2024

UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE

MANUAL DE ESTRUCTURAS DE DATOS

INTRODUCCIÓN Y FUNDAMENTOS DE LAS ESTRUCTURAS DE DATOS

TERCER SEMESTRE
NRC: 14675

Contenido

1. USO DE TDA

1.1. Introducción

Los Tipos de Datos Abstractos (TDA) son una herramienta fundamental en el mundo de la programación y la informática en general. Permiten encapsular datos y operaciones relacionadas en una sola entidad, facilitando la modularidad, el reúso de código y la abstracción. En este manual, exploraremos el uso de los TDA, su importancia y cómo implementarlos en diferentes contextos.

1.2. Definición de Tipos de Datos Abstractos (TDA)

Un Tipo de Dato Abstracto (TDA) es una abstracción matemática que define un conjunto de valores y un conjunto de operaciones que pueden realizarse sobre esos valores. Es abstracto en el sentido de que no especifica cómo se implementan los datos ni cómo se realizan las operaciones, sino que se enfoca en definir qué operaciones son posibles y cuál es su comportamiento.

Imagen 1 *Ciclo de vida del TDA*

1.3. Importancia de los TDA

Los TDA son importantes porque promueven la modularidad y la encapsulación en el diseño de software. Al encapsular datos y operaciones relacionadas en una sola entidad, se simplifica la complejidad del sistema y se facilita su mantenimiento y extensión. Además, los TDA promueven la reutilización de código, ya que una vez definido un TDA, puede ser utilizado en diferentes partes del programa sin necesidad de conocer los detalles de su implementación.

1.4. Uso de TDA

Los TDA se utilizan en una amplia variedad de contextos, desde la programación orientada a objetos hasta la programación funcional. A continuación, se presentan algunas formas comunes en las que se utilizan los TDA:

Uso de TDA en programación: En programación, los TDA se utilizan para modelar conceptos abstractos como listas, conjuntos, pilas, colas, árboles, entre otros. Estos TDA proporcionan una interfaz clara y coherente para interactuar con los datos, independientemente de cómo estén implementados internamente. Ejemplos de TDA comunes: Algunos ejemplos comunes de TDA incluyen:

- Listas enlazadas
- Árboles binarios
- Colas y pilas

- Conjuntos y diccionarios
- Grafos

En otras palabras, un TAD es un tipo de datos construido por el programador para resolver una determinada situación. Para definir un TAD, el programador debe comenzar por definir las operaciones que se pueden realizar con él, es decir, qué operaciones son relevantes y útiles para operar con las variables pertenecientes al mismo. Esto se conoce como establecer la interfaz del tipo. La interfaz permite al programador utilizar el tipo (qué se puede hacer frente a cómo está hecho).

1.5. Implementación de TDA

La implementación de un TDA puede variar dependiendo del lenguaje de programación y del contexto en el que se utilice. Sin embargo, hay algunas pautas generales que se pueden seguir:

- Definir una interfaz clara que especifique las operaciones que se pueden realizar sobre el TDA.
- Implementar las operaciones utilizando las estructuras de datos y algoritmos adecuados.
- Ocultar los detalles de implementación detrás de la interfaz, utilizando encapsulamiento u otros mecanismos de protección de datos.

Siguiendo con el ejemplo de las fechas, podemos indicar que las operaciones válidas sobre una fecha son, entre otras:
Crear (dia, mes, año: natural): fecha
Incrementar (fechaInicio: fecha; numDias: entero): fecha
Distancia (fechaInicio, fin: fecha): entero
ObtenerMes (f: fecha): natural

Fecha.h

```
#ifndef FECHA_H
#define FECHA_H

class Fecha {
    private:
        int dia, mes, anio;
    public:
        Fecha(int d, int m, int a);
        Fecha incrementar(int numDias);
        int distancia(Fecha fin);
tiempo.
        int obtenerMes();
    private:
        int diasEnMes(int m, int a);
        int diasDesde1900(int d, int m, int a);
};
```

Fecha.cpp

```
#endif
#include "Fecha.h"
Fecha::Fecha(int d, int m, int a) {
    dia = d;
    mes = m;
    anio = a;
}
Fecha Fecha::incrementar(int numDias) {
    dia += numDias;
    if (dia > diasEnMes(mes, anio)) {
        dia -= diasEnMes(mes, anio);
        mes++;
        if (mes > 12) {
            mes = 1;
            anio++;
        }
    }
    return Fecha(dia, mes, anio);
}
int Fecha::distancia(Fecha fin) {
    int diasInicio = diasDesde1900(dia, mes, anio);
    int diasFin = diasDesde1900(fin.dia, fin.mes, fin.anio);
    return diasFin - diasInicio;
}
int Fecha::obtenerMes() {
    return mes;
}
int Fecha::diasEnMes(int m, int a) {
    if (m == 2) {
        if (a % 4 == 0 && (a % 100 != 0 || a % 400 == 0)) {
            return 29;
        } else {
            return 28;
        }
    } else if (m == 4 || m == 6 || m == 9 || m == 11) {
        return 30;
    } else {
        return 31;
    }
}
int Fecha::diasDesde1900(int d, int m, int a) {
    int dias = 0;
    for (int i = 1900; i < a; i++) {
        if (i % 4 == 0 && (i % 100 != 0 || i % 400 == 0)) {
            dias += 366;
        } else {
            dias += 365;
        }
    }
```

```
    for (int i = 1; i < m; i++) {
        dias += diasEnMes(i, a);
    }
    dias += d - 1;
    return dias;
}
```

1.6. Conclusiones

En resumen, los Tipos de Datos Abstractos (TDA) son una herramienta poderosa para la programación y el diseño de software. Al encapsular datos y operaciones relacionadas en una sola entidad, los TDA promueven el modularidad, el reúso de código y la abstracción, facilitando el desarrollo de sistemas complejos.

2. MANEJO DE PLANTILLAS (CLASES GENÉRICAS)

2.1. Introducción

El manejo de plantillas en C++ permite la creación de clases genéricas que pueden trabajar con diferentes tipos de datos. Esto se logra utilizando plantillas de clases, que son clases parametrizadas por uno o más tipos de datos. Las plantillas permiten escribir código que se adapta automáticamente a diferentes tipos de datos, lo que mejora la reutilización del código y la legibilidad de este.

2.2. Definiciones:

2.2.1. Plantillas (Templates):

Las plantillas son un mecanismo en C++ que permite definir funciones y clases que funcionan con tipos de datos genéricos.
Permiten escribir código que se puede reutilizar con diferentes tipos de datos sin tener que escribir múltiples versiones de la misma función o clase.

2.2.2. Clases Genéricas:

Las clases genéricas, o plantillas de clases, son clases que pueden trabajar con tipos de datos genéricos.
Se definen utilizando la palabra clave template, seguida de la lista de parámetros de tipo entre <>.

2.2.3. Tipos de Datos Abstractos (TDA):

Los TDAs son una abstracción en programación que encapsula datos y operaciones en una estructura cohesiva.
Permiten ocultar los detalles de implementación y proporcionan una interfaz clara para interactuar con los datos.

2.3. Importancia de las plantillas:

Las plantillas permiten escribir código genérico que puede funcionar con múltiples tipos de datos sin necesidad de escribir implementaciones específicas para cada tipo. Esto promueve la reutilización del código y reduce la duplicación.

También permiten la creación de abstracciones de datos y algoritmos independientes del tipo, lo que facilita la creación de estructuras de datos y algoritmos genéricos que pueden adaptarse a diferentes necesidades y tipos de datos.

Las plantillas proporcionan flexibilidad en el diseño del código al permitir la parametrización por tipos y valores. Esto permite adaptar el comportamiento de las funciones y clases a las necesidades específicas del usuario sin tener que cambiar la implementación.

2.4. Ejemplo:

En este ejemplo, crearemos una clase genérica llamada Par que representa un par de valores de cualquier tipo. La clase tendrá métodos para establecer y obtener los valores del par.

Par.h

```
#ifndef PAR_H
#define PAR_H
template <typename T1, typename T2>
class Par {
private:
    T1 primero;
    T2 segundo;

public:
    Par(const T1& p, const T2& s) : primero(p), segundo(s) {}

    T1 obtenerPrimero() const {
        return primero;
    }

    T2 obtenerSegundo() const {
        return segundo;
    }

    void establecerPrimero(const T1& p) {
        primero = p;
    }

    void establecerSegundo(const T2& s) {
        segundo = s;
    }
};

#endif
```

main.cpp

```
#include <iostream>
#include "Par.h"
```

```
int main() {
    Par<int, double> miPar(5, 3.14);

        std::cout << "Primer valor: " << miPar.obtenerPrimero() <<
std::endl;
    std::cout << "Segundo valor: " << miPar.obtenerSegundo() <<
std::endl;

    miPar.establecerPrimero(10);
    miPar.establecerSegundo(6.28);

    std::cout << "Primer valor actualizado: " <<
miPar.obtenerPrimero() << std::endl;
    std::cout << "Segundo valor actualizado: " <<
miPar.obtenerSegundo() << std::endl;

    return 0;
}
```

2.5. Conclusiones

En conclusión, el uso de plantillas genéricas en C++ ofrece una manera poderosa de escribir código flexible y reutilizable al permitir la definición de clases y funciones que pueden trabajar con múltiples tipos de datos.

Esto promueve la reutilización del código al tiempo que proporciona flexibilidad y parametrización, lo que permite adaptar el comportamiento del código a diferentes necesidades sin cambiar su implementación. Además, las plantillas facilitan la abstracción y generalización al permitir la creación de estructuras de datos y algoritmos independientes del tipo, lo que simplifica el desarrollo y promueve una mayor modularidad y eficiencia en el diseño del software.

3. SOBRECARGA DE FUNCIONES Y OPERADORES EN C++

3.1. ¿Qué son?

La sobrecarga de funciones y operadores es una característica en programación que permite definir múltiples versiones de una función u operador con el mismo nombre, pero con diferentes parámetros o tipos de datos. Esto facilita la flexibilidad y la expresividad del código.

3.2. ¿Cómo se definen?

La sobrecarga de funciones implica definir varias funciones con el mismo nombre, pero con diferentes tipos o cantidades de parámetros.

La sobrecarga de operadores, por otro lado, consiste en definir comportamientos específicos para operadores en contextos distintos.

3.3. ¿Para qué sirven?

Sobrecarga de Funciones: Permite utilizar el mismo nombre de función para realizar tareas diferentes según el contexto.

Sobrecarga de Operadores: Facilita la manipulación de tipos de datos personalizados, permitiendo definir el comportamiento de los operadores en esos tipos.

3.4. Importancia o Utilidad dentro de la Programación

La sobrecarga de funciones y operadores mejora la legibilidad y mantenibilidad del código al permitir un uso más natural de las funciones y operadores con diferentes tipos de datos. Esto promueve la reutilización del código y simplifica la implementación de clases y estructuras personalizadas.

3.5. Operadores que se pueden sobrecargar

3.5.1. Operadores Aritméticos:

- + (suma)
- - (resta)
- * (multiplicación)
- / (división)
- % (módulo)

3.5.2. Operadores de Comparación:

- == (igual a)
- != (diferente de)
- < (menor que)
- > (mayor que)
- <= (menor o igual que)
- >= (mayor o igual que)

3.5.3. Operadores Lógicos:

- && (y lógico)
- || (o lógico)
- ! (no lógico)

3.5.4. Operadores de Incremento y Decremento:

- ++ (incremento)
- -- (decremento)

3.5.5. Operadores de Miembro:

- . (punto) - para acceder a miembros de una clase o estructura.
- -> (flecha) - para acceder a miembros a través de un puntero.

3.5.6. Operador de Asignación:

- = (asignación)

3.6. Operadores que no se pueden sobrecargar:

3.6.1. Operador de Punto y Flecha:

- . y → son operadores de acceso a miembros y no pueden ser sobrecargados directamente.

3.6.2. Operador de Resolución de Ámbito:

- :: no puede ser sobrecargado.

3.6.3. Operadores de Ternario:

- ?: (operador ternario condicional) no puede ser sobrecargado.

3.6.4. Operadores de Miembro Puntero a Miembro:

- . * y →* no pueden ser sobrecargados.

3.6.5. Operadores de Flujo de Entrada/Salida:

- << y >> no pueden ser sobrecargados directamente. Sin embargo, se pueden sobrecargar mediante funciones amigas.

3.6.6. Operadores Nuevos y Eliminar:

- ‘new’ y ‘delete’ no pueden ser sobrecargados.

Nota

Es crucial tener en cuenta estas restricciones y utilizar la sobrecarga de operadores con precaución para evitar comportamientos ambiguos o confusos en el código. Además, cuando se sobrecargan operadores, es recomendable seguir las convenciones y mantener la coherencia para mejorar la legibilidad del código.

3.7. Practica

3.7.1. Diseño

Sobrecarga de Funciones

```
1   /* Ejemplo de sobrecarga de función para
2   sumar dos números de diferentes tipos*/
3   int sumar(int a, int b) {
4       return a + b;
5   }
6
7   double sumar(double a, double b) {
8       return a + b;
9   }
```

Imagen 2 *Sobrecarga de funciones*

Sobrecarga de operadores

```
1   // Ejemplo de sobrecarga del operador + para una
2   clase personalizada Fraccion
3   class Fraccion {
4   public:
5       int numerador;
6       int denominador;
7
8       Fraccion operator+(const Fraccion& otra) {
9           Fraccion resultado;
10          // Implementar la suma de fracciones
11          return resultado;
12      }
13  };
```

Imagen 3 *Sobrecarga de Operadores*

3.8. Qué hacer y Qué no hacer durante la codificación

3.8.1. Hacer:

- Utilizar la sobrecarga de funciones y operadores para mejorar la expresividad y reutilización del código.
- Documentar claramente la lógica de la sobrecarga para facilitar su comprensión.

3.8.2. No Hacer:

- Abusar de la sobrecarga, lo que puede llevar a un código confuso y difícil de mantener.
- Sobrecargar funciones u operadores de manera inconsistente o ambigua.

3.9. Implementación en código

El programa presenta un ejemplo de sobrecarga de operadores en el cual se sobrecarga al operador *

Clase main.cpp

```
#include <iostream>
#include "Frac.h"

int main(){
    Frac F1, F2, result;
    system("cls");
    cout << "Ingresar la primera fracion: " << endl;
    F1.in();
    cout << " Ingresar la segunda fracion: " << endl;
    F2.in();
    result = F1 * F2;
    result.out();
    return 0;
}
```

ClasesFrac.h

```
#include <iostream>
using namespace std;

class Frac{
    private:
        int a;
        int b;
        public:
        Frac() : a(0), b(0){}
        void in();
        Frac operator*(const Frac &obj);
        void out();
};
```

ClasesFrac.cpp

```
#include <iostream>
#include "Frac.h"
using namespace std;

void Frac::in(){
    cout << "Ingrese el numerador : ";
    cin >> a;
    cout<< "Ingrese el denominator : ";
    cin >> b;
}

// Overload the * operator
Frac Frac::operator*(const Frac &obj){
    Frac temporal;
```

```
    temporal.a = a * obj.a;
    temporal.b = b * obj.b;

    return temporal;
}
void Frac::out(){
    cout << "La multiplicacion de la fraccion es: " << a << "/"
<< b;
}
```

3.10. Recomendación y conclusiones

La sobrecarga de funciones y operadores es una poderosa herramienta de programación, pero debe utilizarse con moderación y cuidado. Se recomienda aplicarla cuando aporta claridad y coherencia al código, evitando situaciones que puedan llevar a ambigüedades. La documentación y la consistencia son clave para garantizar un uso efectivo de esta característica.

En conclusión, la sobrecarga mejora la legibilidad y la flexibilidad del código, proporcionando una forma elegante de trabajar con diferentes tipos de datos.

4. GESTIÓN DE MEMORIA ESTÁTICA CON TDA

4.1. Definición.

Para implementar alguna estructura de datos, primero es necesario tener muy claro cómo va a ser el manejo de memoria.

La diferencia entre estructuras estáticas y dinámicas está en el manejo de memoria.

En la memoria estática durante la ejecución del programa el tamaño de la estructura no cambia.

La estructura que maneja memoria estática son los vectores. Un vector es una colección finita, homogénea y ordenada de elementos. Un vector es una colección finita, homogénea y ordenada de elementos. Es finita porque todo arreglo tiene un límite, homogénea porque todos los elementos son del mismo tipo y ordenada porque se puede determinar cuál es el enésimo elemento.

Un vector tiene dos partes: Componente e índices.

Los componentes hacen referencia a los elementos que forman el arreglo y los índices permiten referirse a los componentes del arreglo en forma individual.

- Los arreglos se clasifican en:
  - Unidimensionales (vectores o listas).
  - Bidimensionales (matrices o tablas).
  - Multidimensionales

4.2. Ejemplo TDA con Memoria Estática:

main.cpp

```
#include <iostream>
#include "Vector.h"
#include "Validacion.h"
int main(){
    std::cout << "\n\tMEMORIA ESTATICA CON TDA\n\n";
    int tamanoVector = ingresar_enteros("Ingresar el tamanio de la vector:
");
    Vector vec(tamanoVector);
    do {
        system("cls");
        std::cout << "\n\tMEMORIA ESTATICA CON TDA\n\n" <<
            "1. Agregar\n" <<
            "2. Eliminar\n" <<
            "3. Mostrar\n" <<
            "4. Salir\n\n";
        switch (ingresar_enteros("Ingresar la opcion:  ")) {
        case 1:
            vec.insertar(ingresar_enteros("\n\nIngresar un numero:  "));
            break;
        case 2:
            vec.eliminar(ingresar_enteros("\n\nIngresar el numero a eliminar:
"));
            break;
        case 3:
            vec.imprimir();
            break;
        case 4:
            return 0;
        default:
            break;
        }
    } while (true);
}
```

Vector.h

```
#pragma once
#include <iostream>
// Definición del TDA Vector
class Vector {
private:
    int* array; // Puntero al array
    int tamanoActual;   // Tamaño actual del vector
```

```
    int tamanoVector;
public:
    Vector(int tamInicial) : tamanoActual(0) {
        array = new int[tamInicial]; // Asigna memoria para el array
        tamanoVector = tamInicial;
    }
    void insertar(int elemento) {
        if (tamanoActual < tamanoVector) {
            array[tamanoActual++] = elemento;
        }
        else {
            std::cout << "\n\nEl vector esta lleno\n" << std::endl;
            system("pause");
        }
    }
    void eliminar(int dato) {
        int indice = -1;
        for (int i = 0; i < tamanoActual; ++i) {
            if (array[i] == dato) {
                indice = i;
                break;
            }
        }
        if (indice != -1) {
            for (int i = indice; i < tamanoActual - 1; ++i) {
                array[i] = array[i + 1];
            }
            tamanoActual--; // Decrementar el tamaño del vector
        }
        else {
            std::cout << "\n\nNo se encontro el dato\n" << std::endl;
            system("pause");
        }
    }

    void imprimir() {
        std::cout << "\n\nVector:  ";
        for (int i = 0; i < tamanoActual; ++i) {
            std::cout << array[i] << " ";
        }
        std::cout << std::endl << std::endl;
        system("pause");
    }
};
```

Validacion.h

```
#pragma once
#include <conio.h>
#include <iostream>
int borrar(char* datos, int& i) {
    if (i > 0) {
        printf("\b \b");
        i--;
        datos[i] = '\0';
        return i;
    }
    return 0;
}
int ingresar_enteros(std::string msj) {
    char* datos = new char[10];
    char c;
    int i = 0;
    std::cout << msj;
    while ((c = _getch()) != 13 && i < 9) {
        if ((c >= '0' && c <= '9') || c == 8) {
            if (c == 8) {
                i = borrar(datos, i);
            }
            else {
                printf("%c", c);
                datos[i++] = c;
            }
        }
    }
    datos[i] = '\0';
    return atoi(datos);
}
```

5. GESTIÓN DE MEMORIA DINÁMICA CON TDA

En la memoria dinámica durante la ejecución del programa el tamaño de la estructura puede cambiar, las estructuras de datos dinámicas se generan a partir de un dato conocido como referencia (dirección de memoria).

5.1. Se dividen en dos grandes grupos

5.1.1. Lineales: Pilas, Colas, Listas enlazadas.

5.1.2. No Lineales: Arboles y Grafos.

Punteros: Es una variable que contiene una posición de memoria, y por tanto se dice que apunta a esa posición de memoria.

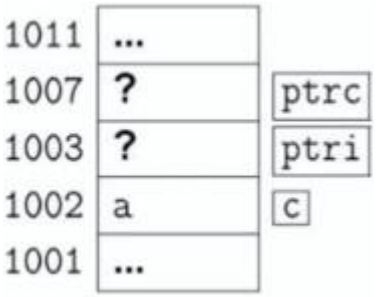

Imagen 4 *Punteros*

Declaración: Declaración <tipo> *<identificador> <tipo> Tipo de dato del objeto referenciado por el puntero <identificador> Identificador de la variable de tipo puntero.

Dirección: Operador & &<id> devuelve La dirección de memoria donde comienza la variable <id>.

El operador & se utiliza para asignar valores a datos de tipo puntero

Indirección: Operador * * devuelve el contenido del referenciado por el puntero. objeto El operador * se usa para acceder a los objetos a los que apunta un puntero

Asignación: Operador = A un puntero se le puede asignar una dirección de memoria concreta, la dirección de una variable o el contenido de otro puntero.

Punteros a punteros: Es un puntero que contiene la dirección de memoria de otro puntero.

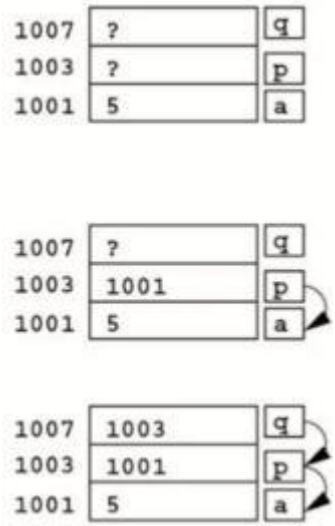

Imagen 5 *Asignación de puntero a puntero*

Para acceder al valor de la variable a podemos escribir a (forma habitual)

*p (a través del puntero p)

**q (a través del puntero a puntero q)

q contiene la dirección de p, que contiene la dirección de a

Operador New: Sirve para reservar memoria Este operador permite crear un objeto de cualquier tipo, incluyendo tipos definidos por el usuario, devuelve un adecuado) al objeto creado.

Operador Delete: Se usa para liberar la memoria dinámica reservada con new. La expresión será normalmente un puntero, el operador delete[] se usa para liberar memoria de arrays dinámicos. Es importante liberar siempre usando delete la memoria reservada con new.

5.2. Implementación de TDA

La implementación de un TDA puede variar dependiendo del lenguaje de programación y del contexto en el que se utilice. Sin embargo, hay algunas pautas generales que se pueden seguir:

- Definir una interfaz clara que especifique las operaciones que se pueden realizar sobre el TDA.
- Implementar las operaciones utilizando las estructuras de datos y algoritmos adecuados.
- Ocultar los detalles de implementación detrás de la interfaz, utilizando encapsulamiento u otros mecanismos de protección de datos.

Siguiendo con el ejemplo de las fechas, podemos indicar que las operaciones válidas sobre una fecha son, entre otras:
Crear (dia, mes, año: natural): fecha
Incrementar (fechaInicio: fecha; numDias: entero): fecha
Distancia (fechaInicio, fin: fecha): entero
ObtenerMes (f: fecha): natural

5.3. Conclusiones

En resumen, los Tipos de Datos Abstractos (TDA) son una herramienta poderosa para la programación y el diseño de software. Al encapsular datos y operaciones relacionadas en una sola entidad, los TDA promueven el modularidad, el reúso de código y la abstracción, facilitando el desarrollo de sistemas complejos.

6. RECURSIVIDAD

6.1. Introducción

La recursividad es una implementación que permite resolver problemas de forma lógica, simplificada y efectiva, su práctica está inmersa en matemáticas, estructura de datos y más.

6.2. Definición de Recursividad

La recursión es un proceso en el que una entidad se define en función de sí misma. Aunque la recursión es poderosa, también puede consumir muchos recursos, por lo que es crucial analizar cuándo y cómo aplicarla. Aunque un

problema pueda ser recursivo por definición, no siempre es el método de solución más adecuado. Antes de implementar un proceso recursivo en aplicaciones prácticas, es necesario demostrar que el nivel máximo de recursión es finito y pequeño, ya que cada llamada recursiva consume memoria para almacenar el estado del proceso. En términos de programación, una función es recursiva cuando se llama a sí misma.

6.3. Importancia de la Recursividad

Permite trabajar de manera simplificada, ya que su naturaleza consiste en convertir a un problema en subproblemas, usando, la menor capacidad posible de espacio, un tiempo rápido de ejecución y un entendimiento mejor de la función.

6.4. Uso de la Recursividad

Cómo se ha mencionado, aunque la recursividad es poderosa, depende totalmente del problema.

Para usar este proceso, se debe,

- Parametrizar las variables que se usarán durante cada llamada.
- Plantear el caso base, mismo que permitirá acabar otra llamada y no caer en un bucle infinito.
- Realizar los cambios respectivos que se harán en las variables de las llamadas.
- Colocar donde se necesiten las llamadas de la misma función.

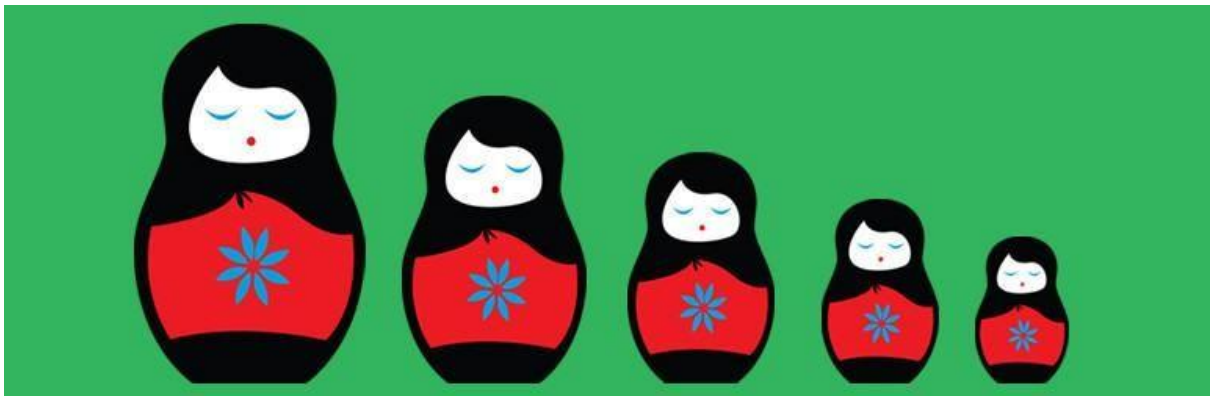

Imagen 6 *Ejemplo de recursividad.*

6.5. Implementación de la Recursividad

Main.cpp

```
#include <iostream>
#include "windows.h"
#include "validaciones.h"
#include "math.h"
```

```
int calcular_potencia(int, int);
int invertir_numero(int, int);

int calcular_potencia(int base, int power){
    if(power != 0){
        return base*calcular_potencia(base,power-1);
    }
    else{
        return 1;
    }
}

int invertir_numero(int numero, int resultado){
    if(numero > 0){
        resultado = resultado*10 + numero%10;
        invertir_numero(numero/10, resultado);
    }
    else{
        return resultado;
    }
}

void mostrar_menu_principal(int opcion){
    system("cls");
    std::cout << "\n----------RECURSIVIDAD---------- \n";
    std::cout << "\nSeleccione una opcion:\n\n";
    std::cout << (opcion == 1 ? "> " : " ") << "Calcule la potencia de un numero entero
positivo\n";
    std::cout << (opcion == 2 ? "> " : " ") << "Invertir un numero entero positivo\n";
    std::cout << (opcion == 3 ? "> " : "  ") << "Salir\n";
}

int main()
{
    int numero1, numero2;
    int opcion = 1;
    char tecla;

    HANDLE consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    CONSOLE_CURSOR_INFO cursorInfo;
    GetConsoleCursorInfo(consoleHandle, &cursorInfo);
    cursorInfo.bVisible = false;
    SetConsoleCursorInfo(consoleHandle, &cursorInfo);

    do{
        mostrar_menu_principal(opcion);
        tecla = _getch();
```

```
        switch(tecla){
            case 72: // Flecha arriba
                if (opcion > 1) opcion--;
                break;
            case 80: // Flecha abajo
                if (opcion < 3) opcion++;
                break;
            case 13: // Enter
                if(opcion == 1){
                    cursorInfo.bVisible = true;
                    SetConsoleCursorInfo(consoleHandle, &cursorInfo);

                    numero1 = ingresar_enteros("Ingrese la base:");
                    printf("\n");

                    numero2 = ingresar_enteros("Ingrese el exponente:");
                    printf("\n");

                    printf("%d ^ %d = %d\n", numero1, numero2, calcular_potencia(numero1,
numero2));

                    system("pause");
                    cursorInfo.bVisible = false;
                    SetConsoleCursorInfo(consoleHandle, &cursorInfo);
                }
                else if(opcion == 2){
                    cursorInfo.bVisible = true;
                    SetConsoleCursorInfo(consoleHandle, &cursorInfo);

                    numero1 = ingresar_enteros("Ingrese el numero el cual quiere invertir:");
                    printf("\n");

                    numero2 = 0;

                    printf("Numero Invertido = %d\n", invertir_numero(numero1, numero2));

                    system("pause");
                    cursorInfo.bVisible = false;
                    SetConsoleCursorInfo(consoleHandle, &cursorInfo);
                }
                else if(opcion == 3){
                    system("cls");
                     std::cout << "Saliendo del programa." << std::endl;
                     exit(0);
                }
                break;
            default:
```

```
            break;
        }

    }while(true);

    system("pause");
    cursorInfo.bVisible = true;
    SetConsoleCursorInfo(consoleHandle, &cursorInfo);
}
```

7. PRINCIPIOS DE LOS ALGORITMOS RECURSIVOS

7.1. Introducción

La recursividad es un concepto fundamental en ciencias de la computación que implica la definición de funciones o algoritmos que se llaman a sí mismos de manera iterativa hasta que se alcanza un caso base, se refiere a la capacidad de una función o procedimiento para llamarse a sí mismo.

Esta técnica se utiliza comúnmente en programación cuando un problema puede ser descompuesto en instancias más simples del mismo problema. La recursividad simplifica la implementación de ciertos algoritmos y puede conducir a soluciones más elegantes y compactas.

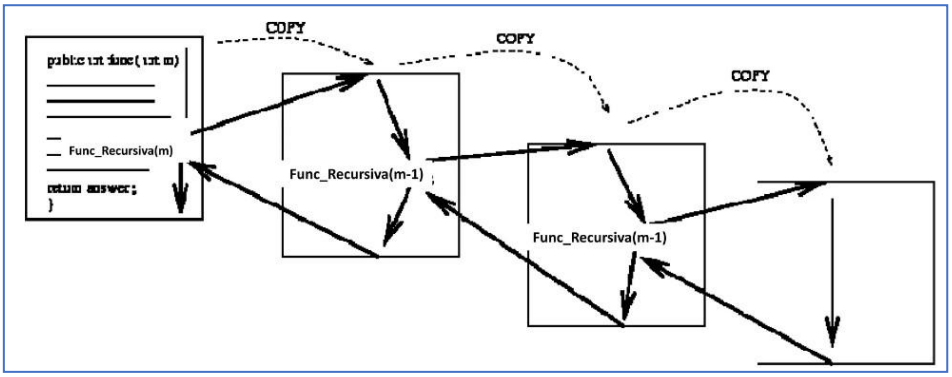

Ilustración 1 *Recursividad de una función*

7.2. Principios de los Algoritmos Recursivos

A continuación, se detallan los principios fundamentales que guían la construcción y comprensión de los algoritmos recursivos:

7.2.1. Caso Base

Todo algoritmo recursivo debe tener al menos un caso base, que es la condición que detiene la recursión. Sin un caso base, la función recursiva continuaría llamándose a sí misma indefinidamente, lo que resultaría en un bucle infinito y un agotamiento de los recursos del sistema. Es crucial definir

claramente este caso base para garantizar la terminación adecuada del algoritmo.

7.2.2. Paso Recursivo

El paso recursivo es el proceso mediante el cual se reduce el problema original a uno más pequeño y manejable. Después de verificar el caso base, el algoritmo debe realizar una llamada a sí mismo con una instancia más simple del problema. Este paso es fundamental para que la recursión avance hacia la resolución del problema original de manera progresiva.

7.2.3. Acumulación de Resultados

En muchos casos, especialmente cuando se trabaja con algoritmos recursivos que procesan estructuras de datos como árboles o listas, es necesario acumular resultados parciales a medida que se avanza en las llamadas recursivas. Esta acumulación puede lograrse mediante parámetros adicionales que se pasan a las llamadas recursivas o mediante el uso de variables globales. Es esencial gestionar correctamente esta acumulación para obtener el resultado correcto al final de la recursión.

7.3. Ejemplos Prácticos

Para ilustrar estos principios, consideremos el ejemplo clásico de la recursión: el cálculo del factorial de un número. El factorial de un número entero no negativo n, denotado como $n!$, se define como el producto de todos los enteros positivos menores o iguales a n. La función factorial se define recursivamente de la siguiente manera:

scssCopy code
factorial(n): si n == 0: retornar 1 sino: retornar n * factorial(n - 1)
En este ejemplo:
El caso base es cuando $n=0$, donde se retorna 1.
El paso recursivo reduce el problema al multiplicar n por el factorial de $n-1$.
No se necesita acumulación de resultados en este caso, ya que el valor del factorial se calcula directamente a partir de las llamadas recursivas.

7.4. Conclusiones

Los principios de los algoritmos recursivos proporcionan una base sólida para la comprensión y la implementación de funciones recursivas. Al entender y aplicar correctamente estos principios, los programadores pueden crear algoritmos recursivos eficientes y libres de errores.

8. TIPOS DE RECURSIVIDAD

La clasificación de la recursividad se basa en cómo se realizan las llamadas y las interacciones entre las funciones recursivas. Aquí se describen algunos tipos comunes:

8.2. Recursividad Directa: En este tipo, una función se llama a sí misma de manera directa. La función realiza una tarea y resuelve un problema más pequeño de manera directa mediante la llamada recursiva.

Ejemplo:

```
#include <iostream>

using namespace std;

class Potencia {
public:
    int calcularPotencia(int base, int exponente) {
        if (exponente == 0) {
            return 1;
        } else {
            return base * calcularPotencia(base, exponente - 1);
        }
    }
};

int main() {
    Potencia potencia;

    int base, exponente;
    cout << "Ingrese la base: ";
    cin >> base;
    cout << "Ingrese el exponente: ";
    cin >> exponente;
    cout << "Potencia: " << potencia.calcularPotencia(base,
exponente) << endl;

    return 0;
}
```

La función calcularPotencia utiliza la recursividad para calcular la potencia de un número. Puede ser más práctico en situaciones donde se necesitan operaciones repetitivas de multiplicación.

8.2 Recursividad Indirecta: En este caso, dos o más funciones se llaman entre sí de manera cíclica para resolver un problema. Cada función realiza una parte del trabajo y delega el resto a las demás funciones.

```
#include <iostream>

using namespace std;

class SecuenciaParImpar {
public:
    void generarSecuencia(int n) {
        if (n > 0) {
            cout << n << " ";
            if (n % 2 == 0) {
                SecuenciaParImpar spi;
                spi.generarSecuencia(n / 2);
 // Llamada recursiva para números pares
            } else {
                SecuenciaParImpar spi;
                spi.generarSecuencia(3 * n + 1);
// Llamada recursiva para números impares
            }
        }
    }
};

int main() {
    SecuenciaParImpar spi;
    cout << "Secuencia para 7: " << endl;
    spi.generarSecuencia(7);
    cout << "\n";
    return 0;
}
```

En este ejemplo, la función generarSecuencia utiliza la recursividad indirecta para generar una secuencia específica. La llamada recursiva depende de si el número es par o impar.

8.3 Recursividad de Cola: En esta variante, la llamada recursiva es la última operación realizada dentro de la función. Especialmente importante en algunos lenguajes de programación que pueden optimizar este tipo de recursividad convirtiéndola en un bucle iterativo.

```
#include <iostream>

using namespace std;

class Factorial {
public:
    int calcularFactorial(int n, int resultado = 1) {
        if (n == 0) {
            return resultado;
        } else {
```

```
            return calcularFactorial(n - 1, n * resultado);
        }
    }
};

int main() {
    Factorial factorial;

    int numero;
    cout << "Ingrese un numero para calcular su factorial: ";
    cin >> numero;
    cout << "Factorial de " << numero << ": " <<
factorial.calcularFactorial(numero) << endl;

    return 0;
}
```

En este ejemplo, la función calcularFactorial utiliza la recursividad de cola para calcular el factorial de un número. La multiplicación se realiza acumulando el resultado en un parámetro adicional.

Estos tipos de recursividad ofrecen diferentes enfoques para abordar problemas y es crucial elegir el tipo que mejor se adapte al contexto del problema y a las características del lenguaje de programación utilizado. La recursividad, cuando se implementa de manera adecuada, puede simplificar la estructura del código y facilitar la comprensión del problema, pero también requiere precaución para evitar problemas como el desbordamiento de la pila de llamadas.

8.4. Conclusiones:

La recursividad en C++ ofrece una forma poderosa y elegante de abordar problemas al dividirlos en casos más simples y resolverlos de manera iterativa. Sin embargo, su uso requiere precaución debido al riesgo de desbordamiento, ya que cada llamada recursiva agrega una nueva entrada de llamadas, lo que puede consumir recursos de memoria significativos. Además, aunque la recursividad puede ser intuitiva y fácil de entender para ciertos problemas, en otros casos puede ser menos eficiente que enfoques iterativos equivalentes, lo que puede afectar el rendimiento del programa en situaciones críticas. Por lo tanto, al usar recursividad en C++, es importante evaluar cuidadosamente la complejidad del problema y considerar las implicaciones de rendimiento y memoria antes de elegir entre enfoques recursivos o iterativos.

9. PROGRAMAS RECURSIVOS

9.1. Introducción

Los programas recursivos son una parte fundamental de la programación que aprovechan la naturaleza auto-referencial de las funciones para resolver problemas de manera elegante y eficiente. En este documento, exploraremos varios ejemplos de programas recursivos, analizando cómo funcionan y por qué la recursividad puede ser preferible a la iteración en ciertos casos.

9.2. Ejemplos de Programas Recursivos

9.2.1. Cálculo del Factorial

El cálculo del factorial es un ejemplo clásico de un problema que se puede resolver de manera recursiva. La función factorial de un número *n*, denotada como *n!*, se define como el producto de todos los enteros positivos menores o iguales a *n*. Aquí está la versión recursiva de la función factorial:

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

9.2.2. Torres de Hanoi

Las pilas de Hanoi son un sistema recursivo que saca la pila de un lado para llevarlo a otra pila de destino, esto funciona sacando los factores de la pila original para una pila auxiliar y luego a una pila de destino Aquí la representación funcional de las pilas de Hanoi:

```
void hanoi(int n, char origen, char auxiliar, char destino) {
    if (n == 1) {
        std::cout << "Mover disco 1 desde " << origen << " hacia " <<
destino << std::endl;
        return;
    }
    hanoi(n - 1, origen, destino, auxiliar);
    std::cout << "Mover disco " << n << " desde " << origen << " hacia
" << destino << std::endl;
    hanoi(n - 1, auxiliar, origen, destino);
}
```

9.2.3. Cálculo de la Secuencia de Fibonacci

El cálculo de la secuencia de Fibonacci es otro ejemplo clásico de un problema que se puede resolver de manera recursiva. La secuencia de Fibonacci es una serie de números en la que cada número es la suma de los dos anteriores. Aquí está la versión recursiva de la función Fibonacci:

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

9.3. Funcionamiento de los Programas Recursivos

En un programa recursivo, la función se llama a sí misma repetidamente con argumentos diferentes hasta que se alcanza una condición de terminación, también conocida como caso base. En ese punto, las llamadas recursivas se detienen y las funciones comienzan a devolver valores hacia atrás a través de las llamadas anidadas, eventualmente devolviendo un resultado final.

Por ejemplo, en el caso del cálculo del factorial, la función se llama a sí misma con argumentos decrecientes hasta que alcanza el caso base n=0n=0, momento en el cual devuelve 1. Luego, los valores se multiplican recursivamente a medida que se desenrollan las llamadas recursivas hasta llegar al resultado final.

9.4. Ventajas de la Recursividad sobre la Iteración

La recursividad puede ser preferible a la iteración en ciertos casos por varias razones:

Claridad y Simplicidad: En algunos problemas, la solución recursiva puede ser más clara y concisa que su contraparte iterativa, lo que hace que el código sea más fácil de entender y mantener.

Naturaleza del Problema: Algunos problemas tienen una estructura naturalmente recursiva, lo que hace que la solución recursiva sea más intuitiva y directa.

Facilidad de Implementación: En ciertos escenarios, la solución recursiva puede ser más fácil de implementar y menos propensa a errores que la iterativa.

Sin embargo, es importante tener en cuenta que la recursividad también puede tener algunas desventajas, como un mayor consumo de memoria y la posibilidad de desbordamiento de pila en casos de profundidad recursiva excesiva.

10. OPERACIONES BÁSICAS CON LISTAS

Las operaciones con listas son las acciones que se pueden realizar sobre las listas, que son estructuras de datos que permiten almacenar una colección de elementos. Las listas son mutables, lo que significa que se pueden modificar después de su creación (Silvia Guardati, 2017).

Los tipos de operaciones básicas son:

- Insertar: Se pueden agregar elementos al final de la lista, al principio de la lista o en una posición determinada.

- Buscar es una operación que comprueba si un elemento determinado se encuentra en una lista. Si el elemento se encuentra en la lista, la operación devuelve un valor booleano que indica que el elemento está presente. Si el elemento no se encuentra en la lista, la operación devuelve un valor booleano que indica que el elemento no está presente.
- Eliminar: es una operación mutable que elimina un elemento de una lista. Se puede eliminar un elemento por su posición o por su valor.

10.1. Operaciones básicas con listas Simples

10.1.1. Insertar Nodo en una lista vacía

- Suponer que tenemos un nodo a ser insertado.

Imagen 7 *Nodo a insertar*

Un puntero denominado Lista que apunte a Null.

Imagen 8 *Nodo auxiliar llamado lista que apunta a NULL*

Proceso:

- Considerar condiciones iniciales.

Imagen 9 *Lista apunta a NULL*

- Nodo---> siguiente debe apuntar a NULL

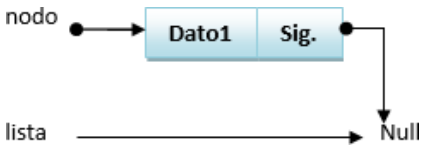

Imagen 10 *Nodo siguiente apunta a NULL*

- Lista apunta a nodo

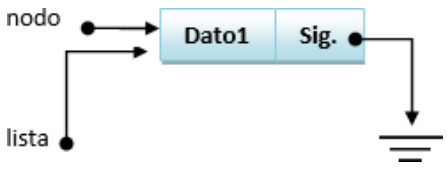

Imagen 11 *Lista apunta al Nodo*

- Lista=nodo

Imagen 12 *Nodo va a hacer igual a lista*

10.1.2. Insertar en la cabeza de la lista

A continuación, se muestra una lista con 4 elementos insertados (2,3,4), y se desea ingresar el numero 1 por la cabeza, para ello se siguen los siguientes pasos:

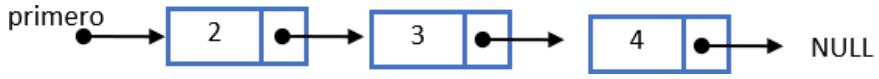

Imagen 13 *Lista simple*

- Crear un nodo e inicializar el campo dato al nuevo elemento. La dirección del nodo creado se asigna a *nuevo*.

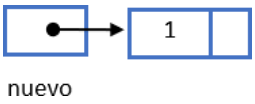

Imagen 14 *Asignación de la dirección del nodo creado a nuevo*

- Hacer que el campo *enlace* del nodo creado apunte a la cabeza (primero) de la lista.

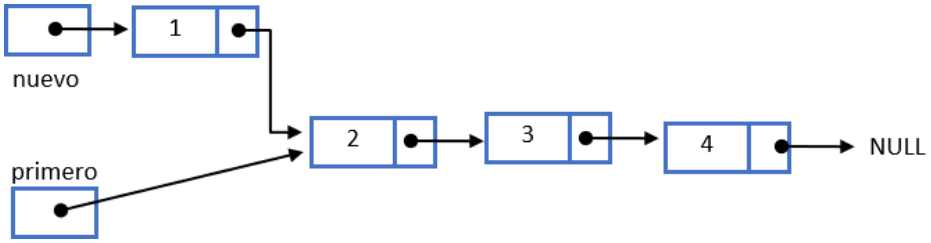

Imagen 15 *Nuevo Nodo apunte a la cabeza*

- Hacer que *primero* apunte al nodo que se ha creado

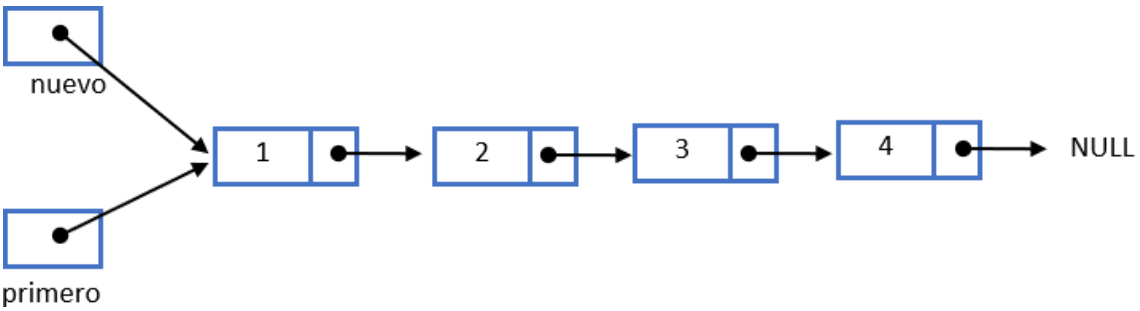

Imagen 16 *Inserción del nuevo nodo*

- En este momento, la función termina su ejecución, la variable local nuevo desaparece y sólo permanece el puntero primero al inicio de la lista.

10.1.3. Insertar en la cola de la lista

Condiciones:

- Disponer de una lista con cierto número de elementos.

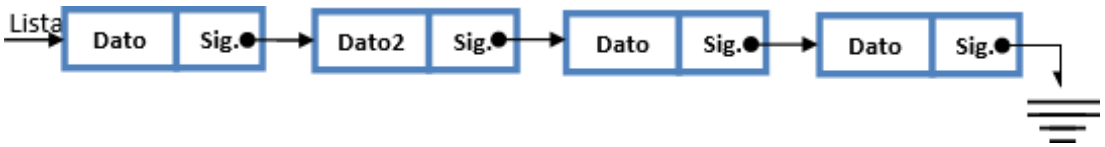

Imagen 17 *Lista Simple con datos*

- Disponer de un nuevo a ser insertado al final de la lista.

Imagen 18 *Nuevo nodo*

- Disponer de un apuntador denominado: "ultimo".

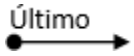

Imagen 19 *Auxiliar denominado Ultimo*

Proceso:

- Integrar en un solo diagrama las condiciones iniciales.

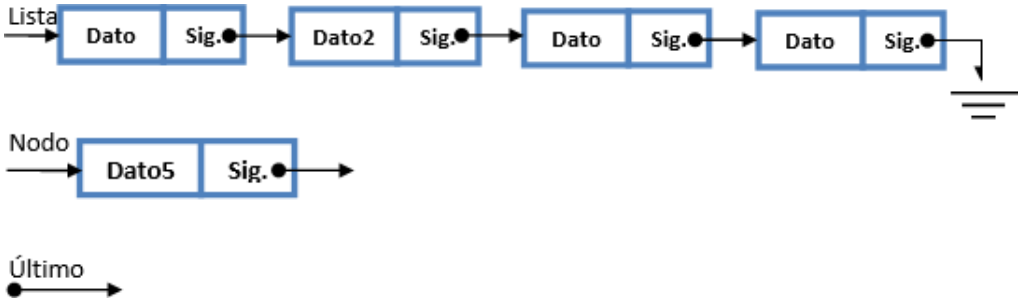

Imagen 20 *Diagrama de condiciones iniciales*

- Hacer que el apuntador "último" apunte al último nodo de la lista.

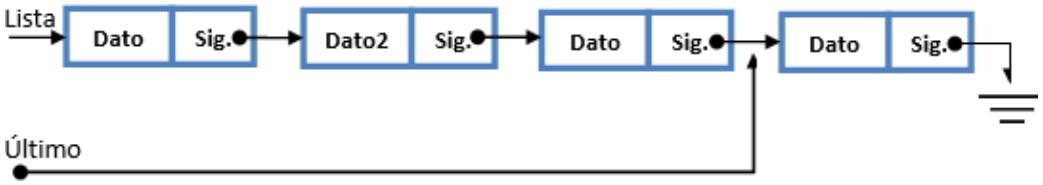

Imagen 21 *Ultimo apunta al último nodo*

- Nodo siguiente y último siguiente apuntan al último nodo de la lista.

Y además el puntero denominado nodo siguiente de la lista que apunta al valor de Null pasa a denominarse último siguiente.

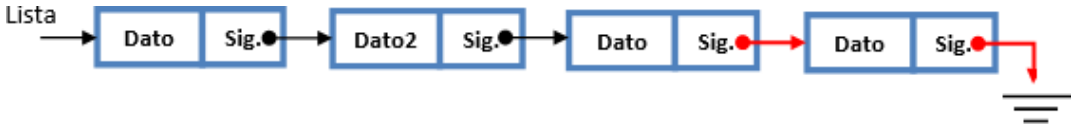

Imagen 22 *nodo->siguiente = ultimo siguiente*

- Hacer que el apuntador nodo->siguiente del nuevo nodo apunte y tome el valor de Null.
  nodo->siguiente = Null
- Hacer que el apuntador ultimo->siguiente que tenía el valor de Null apunte al nuevo nodo.

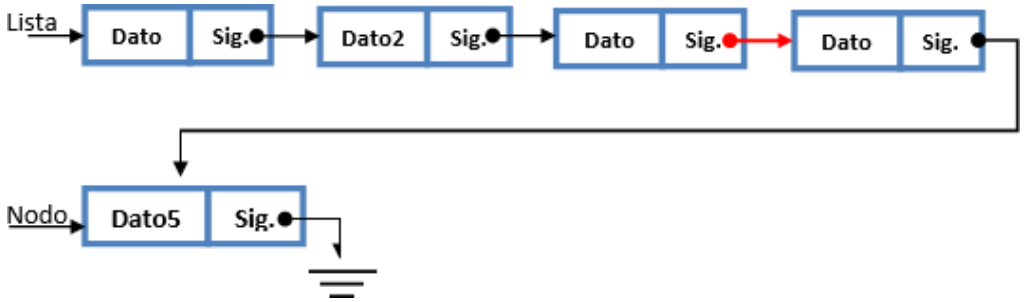

Imagen 23 *ultimo->siguiente y nodo->siguiente apuntan al nuevo nodo a la vez*

- ultimo->siguiente=nodo->siguiente

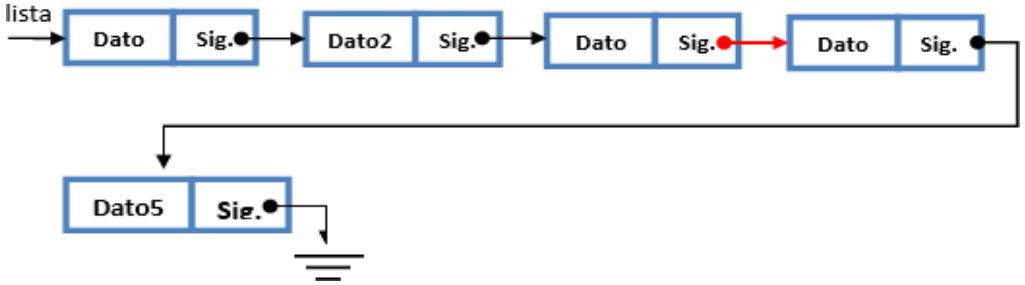

Imagen 24 *Inserción del nuevo Nodo*

10.1.4. Insertar un nuevo nodo en cualquier posición

Condiciones Iniciales:

- Disponer de una lista.

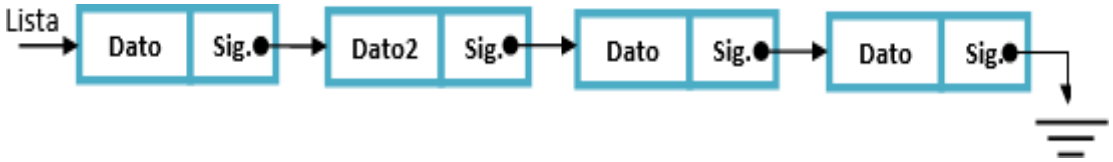

Imagen 25 *Lista Simple*

- Disponer de un nuevo nodo a ser insertado.

Imagen 26 *Nuevo Nodo*

- Disponer de un apuntador denominado anterior

Imagen 27 *Puntero anterior*

Proceso:

- Integrar en un solo diagrama las condiciones iniciales.

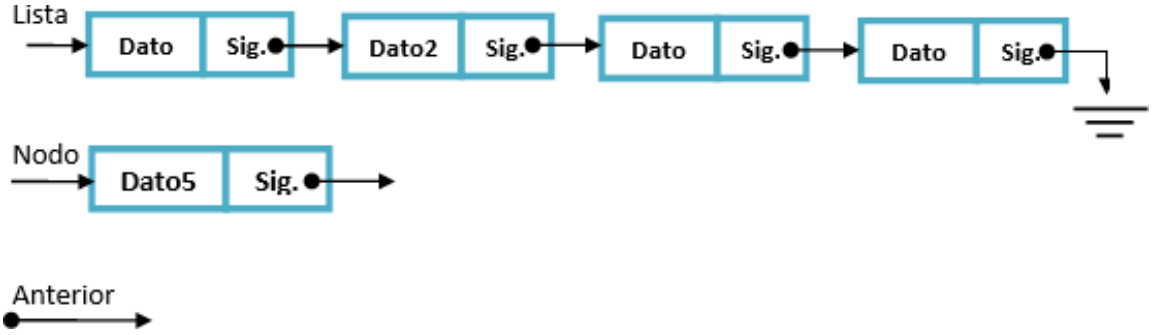

Imagen 28 *Diagrama de condiciones iniciales*

- Determinamos la posición en la cual se va a insertar el nuevo nodo.

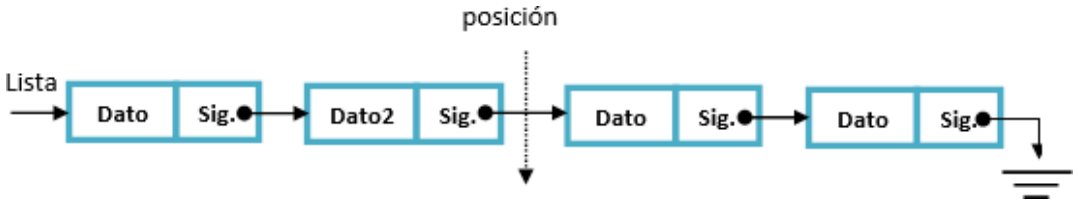

Imagen 29 *Determinación de nuevo nodo a insertar*

- Hacer que el apuntador anterior apunte al nodo->anterior que se encuentra en la lista al cual se va a insertar el nuevo nodo.

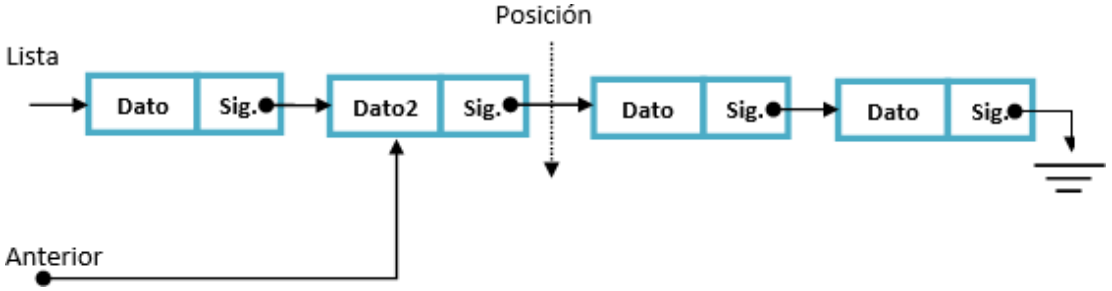

Imagen 30 *Apuntador apunta al nodo anterior donde se desea insertar el nodo*

- Apuntador nodo->siguiente de la lista y anterior->siguiente apuntan a un mismo nodo de la lista.

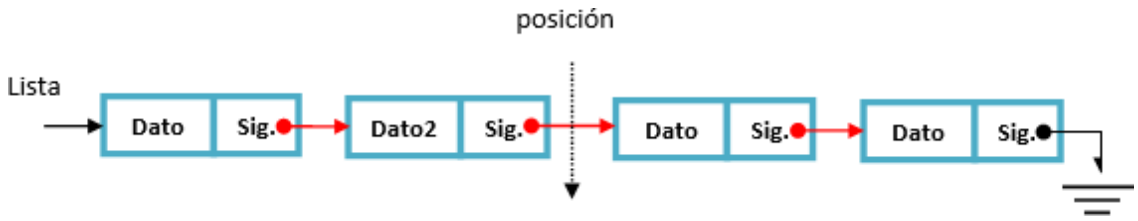

Imagen 31 *nodo->siguiente = anterior->siguiente*

- Hacer que el apuntador nodo->siguiente del nuevo nodo apunte al nodo que se encuentra ubicado en la posición siguiente en la cual se va a ubicar el nuevo nodo**.**

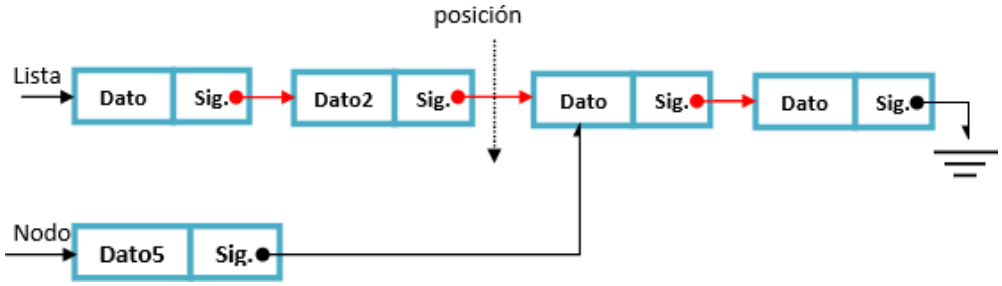

Imagen 32 *Nodo Siguiente apunte a la ubicación de la inserción del nuevo nodo*

- Apuntador anterior->siguiente de la lista y apuntador nodo->siguiente apuntan a un mismo nodo.

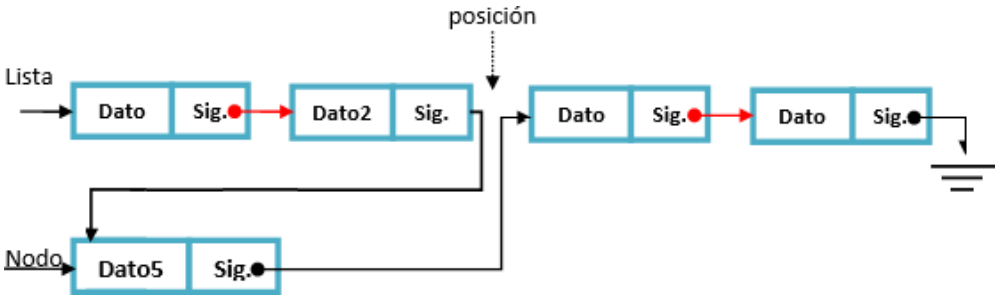

Imagen 33 *Nodo Siguiente = nodo->siguiente*

- Apuntador nodo->siguiente y apuntador nodo->siguiente apuntan al nuevo nodo.
- nodo->siguiente = nodo->siguiente

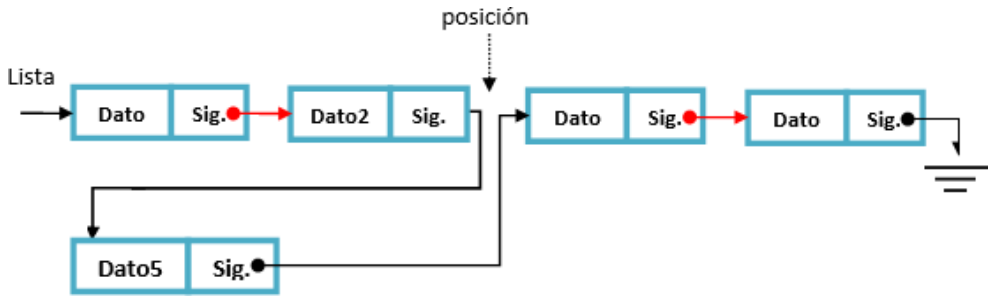

Imagen 35 *Inserción del nuevo nodo*

10.1.5. Buscar Nodo

La operación búsqueda de un elemento en una lista enlazada recorre la lista hasta encontrar el nodo con el elemento. El algoritmo que se utiliza, una vez encontrado el nodo, devuelve el puntero al nodo (en caso negativo, devuelva NULL). Otro planteamiento consiste en devolver true si encuentra el nodo y false si no está en la lista

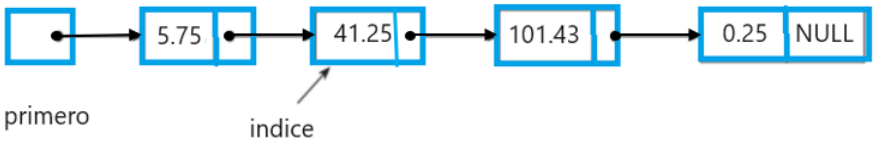

Imagen 35 *Búsqueda de un nodo.*

La función buscar Lista de la clase Lista utiliza el puntero índice para recorrer la lista, nodo a nodo. Primero, se inicializa índice al nodo cabeza (primero), a continuación, se compara el dato del nodo apuntado por índice con el elemento buscado, si coincide la búsqueda termina, en caso contrario índice avanza al siguiente nodo. La búsqueda termina cuando se encuentra el nodo, o bien cuando se ha terminado de recorrer la lista y entonces índice toma el valor NULL.

10.2. Operaciones con Listas Doblemente Enlazadas

10.2.1. Insertar Nodo en una lista vacía

Partiremos de que ya tenemos el nodo a insertar y, por supuesto un puntero que apunte a él, además el puntero que define la lista, que valdrá NULL:

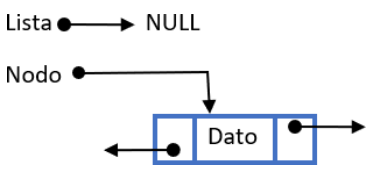

Imagen 36 *Diagrama del nuevo nodo y el puntero que apunta a NULL*

- lista apunta a nodo.

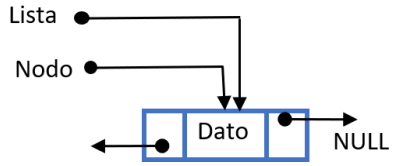

Imagen 37 *Lista->siguiente y lista->anterior apunten a NULL*

10.2.2. Insertar Nodo en una lista por la Cabeza

Partimos de una lista no vacía. Para simplificar, consideraremos que lista apunta al primer elemento de la lista doblemente enlazada:

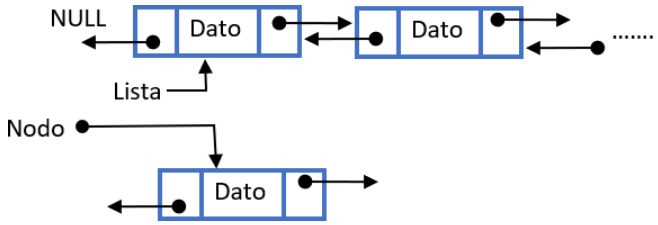

Imagen 38 *Diagrama del nuevo nodo a insertar y la lista no vacía*

El proceso es el siguiente:

- nodo->siguiente debe apuntar a Lista.
- nodo->anterior apuntará a Lista->anterior.
- Lista->anterior debe apuntar a nodo.

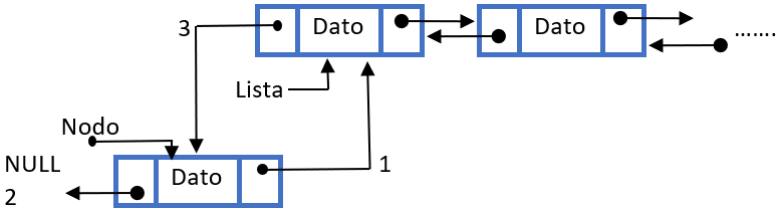

Imagen 39 *Diagrama del comportamiento de nuevo nodo*

Recuerda que Lista no tiene por qué apuntar a ningún miembro concreto de una lista doblemente enlazada, cualquier miembro es igualmente válido como referencia.

10.2.3. Insertar Nodo en una lista por la Cola.

Igual que en el caso anterior, partiremos de una lista no vacía, y de nuevo para simplificar, que Lista está apuntando al último elemento de la lista:

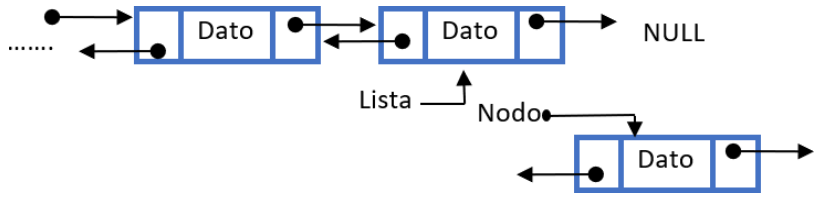

Imagen 40 *Diagrama del nuevo nodo a insertar y la lista no vacía*

El proceso es el siguiente:

- nodo->siguiente debe apuntar a Lista->siguiente (NULL).
- Lista->siguiente debe apuntar a nodo.
- nodo->anterior apuntará a Lista.

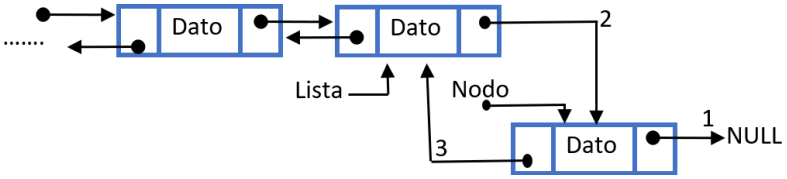

Imagen 41 *Diagrama del comportamiento del nodo que se desea insertar*

Insertar un nuevo nodo en cualquier posición

Bien, este caso es más genérico, ahora partimos de una lista no vacía, e insertaremos un nodo a continuación de uno nodo cualquiera que no sea el último de la lista:

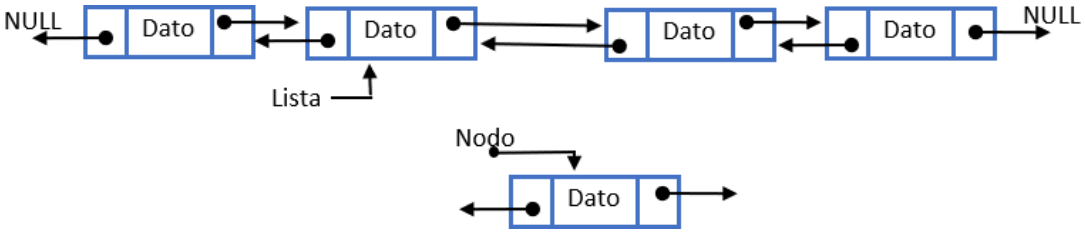

Imagen 42 *Diagrama de la lista y la posición donde se desea ingresar el nuevo nodo*

El proceso sigue siendo muy sencillo:

- Hacemos que nodo->siguiente apunte a lista->siguiente.
- Hacemos que Lista->siguiente apunte a nodo.
- Hacemos que nodo->anterior apunte a lista.
- Hacemos que nodo->siguiente->anterior apunte a nodo

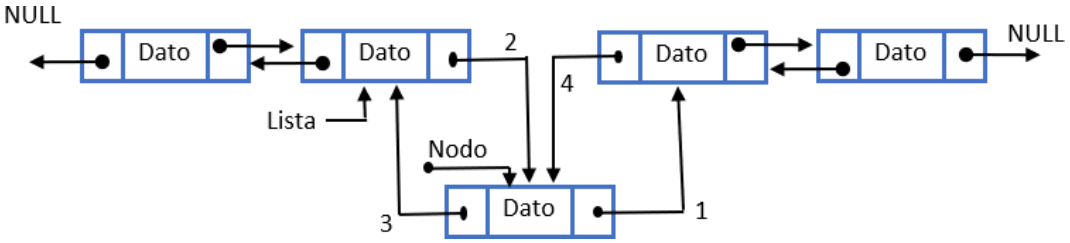

Imagen 43 *Diagrama del comportamiento del nodo que se desea insertar*

10.2.4. Buscar Nodo

Para recorrer una lista procederemos de un modo parecido al que usábamos con las listas abiertas, ahora no necesitamos un puntero auxiliar, pero tenemos que tener en cuenta que Lista no tiene por qué estar en uno de los extremos:

- Retrocedemos hasta el comienzo de la lista, asignamos a lista el valor de lista->anterior mientras lista->anterior no sea NULL.
- Abriremos un bucle que al menos debe tener una condición, que el índice no sea NULL.
- Dentro del bucle asignaremos a lista el valor del nodo siguiente al actual.

10.2.5. Eliminar Nodo

De nuevo tenemos los dos casos posibles, que el nodo a borrar esté apuntado por Lista o que no. Si lo está, simplemente hacemos que Lista sea Lista->anterior, si no es NULL o Lista->siguiente en caso contrario.

- Si nodo apunta a Lista,
  - Si Lista->anterior no es NULL hacemos que Lista apunte a
  - Lista->anterior. Si Lista->siguiente no es NULL hacemos que Lista apunte a NULL
  - Lista->siguiente. Si ambos son NULL, hacemos que Lista sea NULL.
- Si nodo->anterior no es NULL, hacemos que nodo->anterior->siguiente apunte a nodo->siguiente.
- Si nodo->siguiente no es NULL, hacemos que nodo->siguiente->anterior apunte a nodo->anterior.
- Borramos el nodo apuntado por nodo.

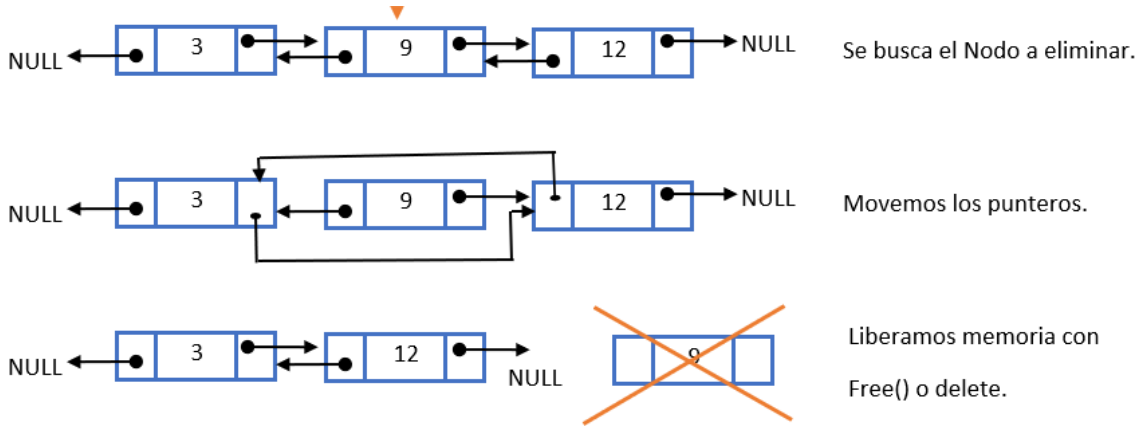

Imagen 44 *Diagrama de la eliminación de un nodo de una lista*

10.3. Listas circulares Simples

10.3.1. Insertar un elemento en una lista circular vacía.

Condiciones Iniciales:

- Disponer de un nodo a ser insertado.

Imagen 45 *Nodo que se desea insertar*

- Disponer de un puntero que defina la lista, el cual valdrá NULL.

Imagen 46 *Puntero que apunta a Null*

Proceso:

- Integrar en un solo diagrama las condiciones iniciales.

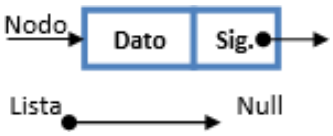

Imagen 47 *Diagrama del nodo a ingresar y el puntero*

- Hacer que Lista apunte al nodo.

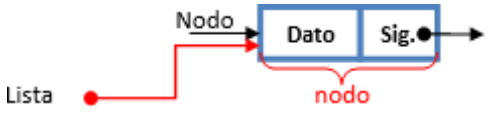

Imagen 48 *Puntero apunta al nodo*

- Lista=nodo

Imagen 49 *El nuevo nodo se apunta a si mismo*

- Lista->siguiente apunte al nodo.

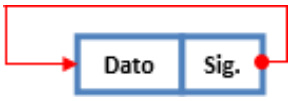

Imagen 50 *Nodo insertado en la lista vacía*

10.3.2. Insertar un elemento en una lista circular no vacía

Condiciones Iniciales:

- Disponer de una lista circular y de un apuntador denominado lista el cual apunte a cualquier nodo de la lista.

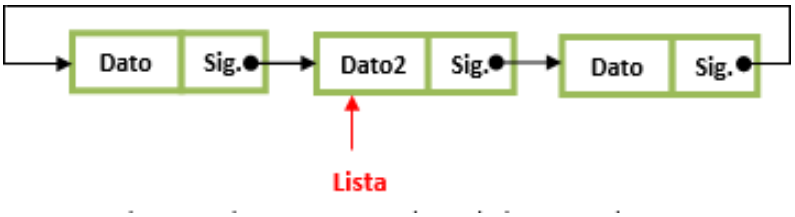

Imagen 51 *Diagrama de la lista circular*

- Disponer de un nodo a ser insertado en la lista circular.

Imagen 52 *Nodo que se desea insertar*

- Ubicar el apuntador a la lista apuntando al nodo en donde va a ser insertado el nuevo nodo.

Imagen 53 *El apuntador apunta a la ubicación donde va a ser insertado*

- Integrar las condiciones iniciales.

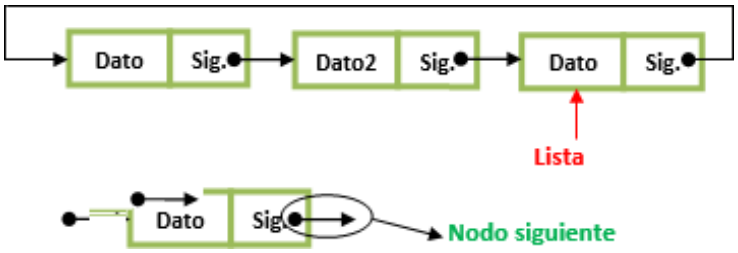

Imagen 54 *Diagrama de condiciones iniciales*

- Hacer que nodo siguiente apunte a la cabecera de lista siguiente.

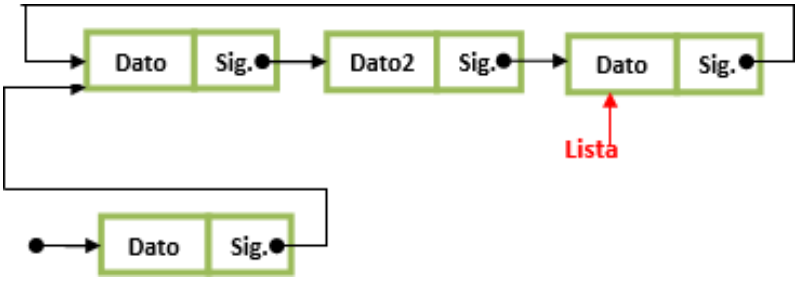

Imagen 55 *Nodo nuevo apunta al nodo que siguiente de donde se va a insertar*

- Hacer que la lista siguiente apunte al nodo a ser insertado

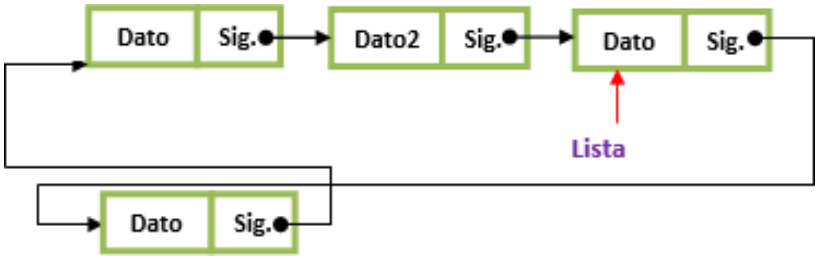

Imagen 56 *La lista apunta al nodo insertado*

- Lista->siguiente = nodo

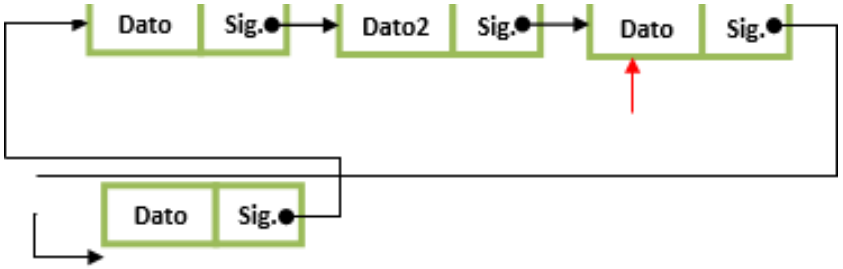

Imagen 57 *Inserción del nodo*

10.3.3. Buscar un nodo en una lista circular

La operación buscar en una lista simple circular es similar a la operación buscar en una lista simple lineal, con la diferencia de que la lista circular no tiene un final. Esto significa que, al llegar al final de la lista, se debe volver al primer nodo para continuar la búsqueda.

10.3.4. Eliminar Nodo

- Disponer una lista circular

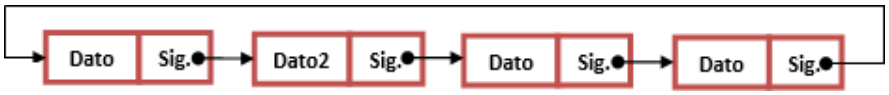

Imagen 56 *Lista circular*

- Hacer que el apuntador auxiliar lista apunte al nodo anterior que deseamos eliminar.

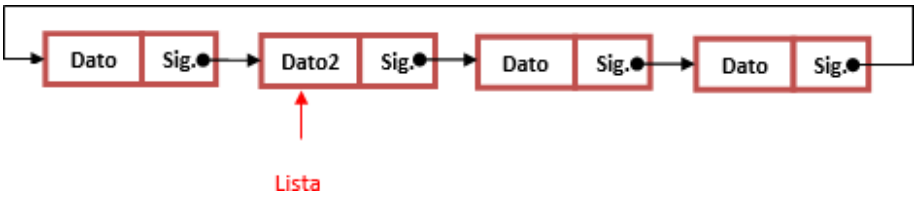

Imagen 58 *El puntero auxiliar apunta a la posición anterior donde se desea hacer la inserción*

- Hacer que el apuntador lista valga lista -> siguiente, mientras lista siguiente sea distinto de nodo.

Imagen 59 *Lista ya no apunta al nodo que se desea eliminar*

- Hacer que lista->siguiente apunte a nodo->siguiente

Imagen 60 *Diagrama del comportamiento lista*

- Lista->siguiente toma la dirección de nodo->siguiente

Imagen 61 *Los nodos dejan de apuntar al nodo que se desea eliminar*

- Eliminar el nodo

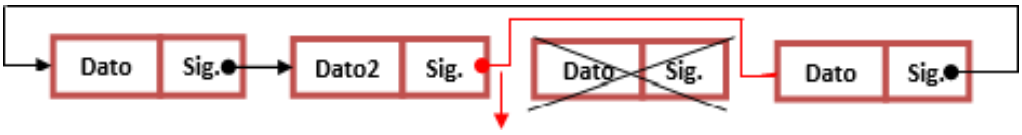

Imagen 62 *Eliminación del nodo*

- Presentar la nueva lista.

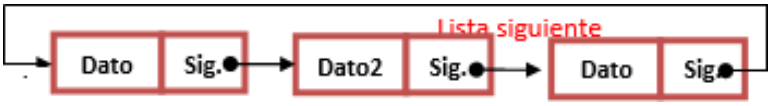

Imagen 63 *Diagrama de la lista sin el nodo*

10.4. Listas Dobles circulares

10.4.1. Insertar Nuevo nodo a la lista

- Crear un nuevo nodo. El nuevo nodo debe tener un puntero al nodo anterior y otro al nodo siguiente.

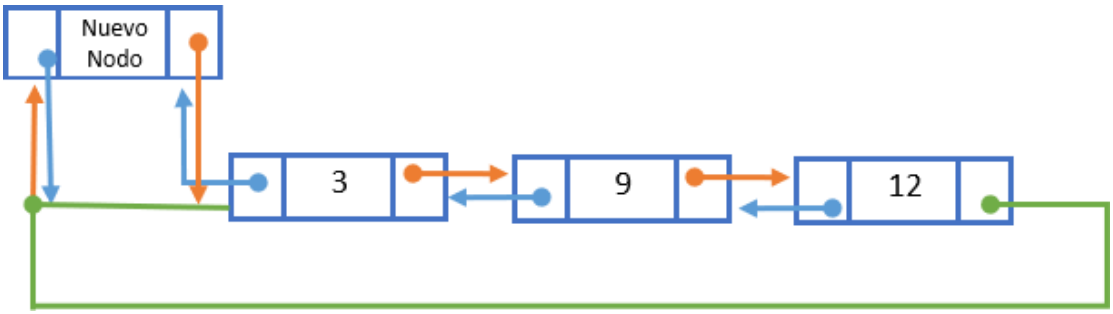

Imagen 64 *Diagrama de la inserción de un nodo a una lista doble circular*

- Obtener el nodo anterior y siguiente del punto de inserción. Podemos obtener estos nodos iterando por la lista hasta llegar al punto de inserción.
- Realizar los enlaces necesarios. Los enlaces necesarios son los siguientes:
    - El puntero al nodo anterior del nuevo nodo se establece en el nodo anterior del punto de inserción.
    - El puntero al nodo siguiente del nuevo nodo se establece en el nodo siguiente del punto de inserción.
    - El puntero al nodo siguiente del nodo anterior del punto de inserción se establece en el nuevo nodo.
    - El puntero al nodo anterior del nodo siguiente del punto de inserción se establece en el nuevo nodo.

10.4.2. Eliminar Nodo

- Obtener el nodo a eliminar. Podemos obtener el nodo a eliminar iterando por la lista hasta encontrar el nodo que queremos eliminar.
- Realizar los enlaces necesarios. Los enlaces necesarios son los siguientes:
    - El puntero al nodo siguiente del nodo anterior del nodo a eliminar se establece en el nodo siguiente del nodo a eliminar.
    - El puntero al nodo anterior del nodo siguiente del nodo a eliminar se establece en el nodo anterior del nodo a eliminar.

10.4.3. Buscar Nodo

Pasos para buscar un elemento en una lista circular doble:

- Inicializar un nodo auxiliar. El nodo auxiliar se usará para iterar por la lista.
- Inicializar el nodo auxiliar al primer nodo de la lista.
- Recorrer la lista. Mientras el nodo auxiliar no sea Nodo y el elemento del nodo auxiliar no sea igual al elemento que buscamos, avanzaremos al siguiente nodo de la lista.
- Si el elemento del nodo auxiliar es igual al elemento que buscamos, devolver el nodo auxiliar.
- Si el nodo auxiliar es Nodo, devolver Nodo.

11. TIPOS DE LISTAS

11.1. Listas Simples

Una lista simple “es una estructura de datos lineal, dinámica, formada por una colección de elementos llamados nodos. Cada nodo está formado por dos partes: la primera de ellas se utiliza para almacenar la información (razón de ser de la estructura de datos), y la segunda se usa para guardar la dirección del siguiente nodo” (Silvia Guardati, 2017,).

Cada nodo (elemento) contiene un enlace que conecta el nodo con el siguiente o el siguiente nodo. Esta lista es eficaz para el recorrido directo ("hacia adelante").

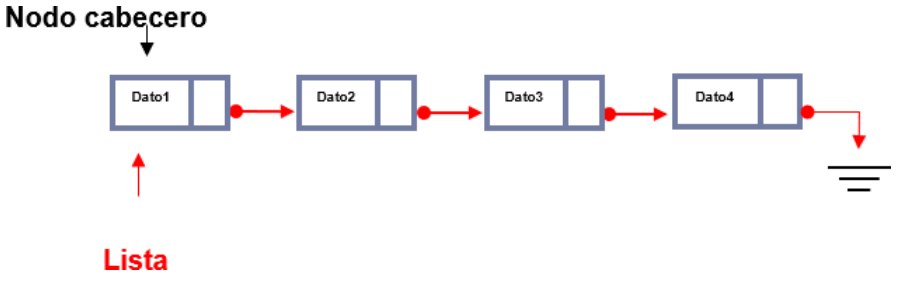

Imagen 8 *Lista Simple*

Ejemplo:

main.cpp

```
#include <iostream>
#include "ListaSimple.h"
#include "Validacion.h"
int main(){
    Lista lista;
```

```
    do {
        system("cls");
        std::cout << "\n\tLISTA SIMPLE\n\n" <<
            "1. Insertar al inicio\n" <<
            "2. Eliminar\n" <<
            "3. Mostrar\n" <<
            "4. Salir\n\n";
        switch (ingresar_enteros("Ingresar la opcion:  ")) {
        case 1:
            lista.insertarAlInicio(ingresar_enteros("\n\nIngresar un numero:
"));
            std::cout << std::endl;
            system("pause");
            break;
        case 2:
            lista.eliminar(ingresar_enteros("\n\nIngresar el numero a
eliminar:  "));
            std::cout << std::endl;
            system("pause");
            break;
        case 3:
            lista.imprimir();
            std::cout << std::endl;
            system("pause");
            break;
        case 4:
            return 0;
        default:
            break;
        }
    } while (true);
}
```

ListaSimple.h

```
#pragma once
#include <iostream>

// Definición de la clase Nodo
class Nodo {

public:
    int dato;
    Nodo* siguiente;

    // Constructor
    Nodo(int valor) : dato(valor), siguiente(nullptr) {}
```

```
};

// Clase Lista que contiene operaciones básicas
class Lista {
private:
    Nodo* cabeza;

public:
    // Constructor
    Lista() {
        cabeza = nullptr;
    }

    // Método para insertar un elemento al inicio de la lista
    void insertarAlInicio(int valor) {
        Nodo* nuevoNodo = new Nodo(valor);
        nuevoNodo->siguiente = cabeza;
        cabeza = nuevoNodo;
    }

    // Método para imprimir la lista
    void imprimir() {
        Nodo* temp = cabeza;
        while (temp != nullptr) {
            std::cout << temp->dato << " -> ";
            temp = temp->siguiente;
        }
        std::cout << "NULL" << std::endl;
    }

    // Método para eliminar un elemento de la lista dada una clave (valor)
    void eliminar(int clave) {
        Nodo* temp = cabeza;
        Nodo* prev = nullptr;

        // Buscar el nodo con la clave dada
        while (temp != nullptr && temp->dato != clave) {
            prev = temp;
            temp = temp->siguiente;
        }

        // Si la clave no se encuentra en la lista, no hay nada que eliminar
        if (temp == nullptr) {
            std::cout << "\nEl elemento " << clave << " no esta en la lista."
<< std::endl;
            return;
        }
```

```
        // Eliminar el nodo encontrado
        if (prev != nullptr)
            prev->siguiente = temp->siguiente;
        else
            cabeza = temp->siguiente;

        delete temp;
    }

    // Destructor para liberar la memoria asignada a los nodos de la lista
    ~Lista() {
        Nodo* temp = cabeza;
        while (temp != nullptr) {
            Nodo* siguiente = temp->siguiente;
            delete temp;
            temp = siguiente;
        }
        cabeza = nullptr;
    }
};
```

Validacion.h

```
#pragma once
#include <conio.h>
#include <iostream>

int borrar(char* datos, int& i) {
    if (i > 0) {
        printf("\b \b");
        i--;
        datos[i] = '\0';
        return i;
    }
    return 0;
}

int ingresar_enteros(std::string msj) {
    char* datos = new char[10];
    char c;
    int i = 0;

    std::cout << msj;
    while ((c = _getch()) != 13 && i < 9) {
        if ((c >= '0' && c <= '9') || c == 8) {
            if (c == 8) {
```

```
                i = borrar(datos, i);
            }
            else {
                printf("%c", c);
                datos[i++] = c;
            }

        }
    }
    datos[i] = '\0';
    return atoi(datos);
}
```

11.2. Listas Dobles Enlazadas.

Una lista enlazada es una lista mundial con dos enlaces, un enlace al nodo siguiente y otro enlace al nodo anterior, además; se recorrido lo podemos hacer en ambos sentidos a partir de cualquier nodo, por esta razón es posible alcanzar cualquier nodo de la lista, hasta llegar a uno de sus extremos. Cada nodo contiene dos enlaces, uno a su nodo predecesor y el otro a su nodo sucesor. La lista es eficiente tanto en recorrido directo (“adelante”) como en recorrido inverso (“atrás”). (Silvia Guardati, 2017)

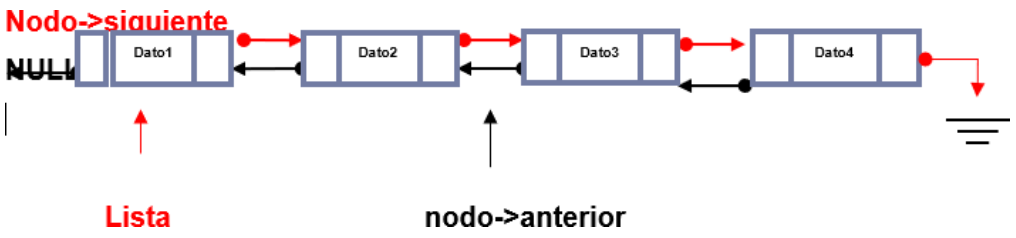

Imagen 65 *Lista Doble*

Ejemplo:

main.cpp

```
#include <iostream>
#include "ListaDoble.h"
#include "Validacion.h"

int main(){
    ListaDoble lista;
    do {
        system("cls");
        std::cout << "\n\tLISTA DOBLE ENLAZADA\n\n" <<
            "1. Insertar al inicio\n" <<
            "2. Eliminar\n" <<
```

```
            "3. Mostrar cabeza\n" <<
            "4. Mostrar cola\n" <<
            "5. Salir\n\n";
        switch (ingresar_enteros("Ingresar la opcion:  ")) {
        case 1:
            lista.insertarAlInicio(ingresar_enteros("\n\nIngresar un numero:
"));
            std::cout << std::endl;
            system("pause");
            break;
        case 2:
            lista.eliminar(ingresar_enteros("\n\nIngresar el numero a
eliminar:  "));
            std::cout << std::endl;
            system("pause");
            break;
        case 3:
            lista.imprimirAdelante();
            std::cout << std::endl;
            system("pause");
            break;
        case 4:
            lista.imprimirAtras();
            std::cout << std::endl;
            system("pause");
            break;
        case 5:
            return 0;
        default:
            break;
        }
    } while (true);
}
```

ListaDoble.cpp

```
#pragma once
#include <iostream>

// Definición de la clase NodoDoble
class NodoDoble {
public:
    int dato;
    NodoDoble* siguiente;
    NodoDoble* anterior;

    // Constructor
    NodoDoble(int valor) : dato(valor), siguiente(nullptr), anterior(nullptr)
{}
```

```
};

// Clase ListaDoble que contiene operaciones básicas
class ListaDoble {
private:
    NodoDoble* cabeza;

public:
    // Constructor
    ListaDoble() {
        cabeza = nullptr;
    }

    // Método para insertar un elemento al inicio de la lista
    void insertarAlInicio(int valor) {
        NodoDoble* nuevoNodo = new NodoDoble(valor);
        if (cabeza == nullptr) {
            cabeza = nuevoNodo;
        }
        else {
            nuevoNodo->siguiente = cabeza;
            cabeza->anterior = nuevoNodo;
            cabeza = nuevoNodo;
        }
    }

    // Método para imprimir la lista de principio a fin
    void imprimirAdelante() {
        NodoDoble* temp = cabeza;
        while (temp != nullptr) {
            std::cout << temp->dato << " <-> ";
            temp = temp->siguiente;
        }
        std::cout << "NULL" << std::endl;
    }

    // Método para imprimir la lista de fin a principio
    void imprimirAtras() {
        NodoDoble* temp = cabeza;
        while (temp->siguiente != nullptr) {
            temp = temp->siguiente;
        }
        while (temp != nullptr) {
            std::cout << temp->dato << " <-> ";
            temp = temp->anterior;
        }
        std::cout << "NULL" << std::endl;
```

```
    }

    void eliminar(int valor) {
        NodoDoble* temp = cabeza;
        NodoDoble* prev = nullptr;

        // Buscar el nodo con el valor dado
        while (temp != nullptr && temp->dato != valor) {
            prev = temp;
            temp = temp->siguiente;
        }

        // Si el valor no se encuentra en la lista, no hay nada que eliminar
        if (temp == nullptr) {
            std::cout << "El elemento " << valor << " no esta en la lista."
<< std::endl;
            return;
        }

        // Eliminar el nodo encontrado
        if (prev != nullptr)
            prev->siguiente = temp->siguiente;
        else
            cabeza = temp->siguiente;

        if (temp->siguiente != nullptr)
            temp->siguiente->anterior = prev;

        delete temp;
    }

    // Destructor para liberar la memoria asignada a los nodos de la lista
    ~ListaDoble() {
        NodoDoble* temp = cabeza;
        while (temp != nullptr) {
            NodoDoble* siguiente = temp->siguiente;
            delete temp;
            temp = siguiente;
        }
        cabeza = nullptr;
    }
};
```

Validacion.h

```
#pragma once
#include <conio.h>
```

```
#include <iostream>

int borrar(char* datos, int& i) {
    if (i > 0) {
        printf("\b \b");
        i--;
        datos[i] = '\0';
        return i;
    }
    return 0;
}

int ingresar_enteros(std::string msj) {
    char* datos = new char[10];
    char c;
    int i = 0;

    std::cout << msj;
    while ((c = _getch()) != 13 && i < 9) {
        if ((c >= '0' && c <= '9') || c == 8) {
            if (c == 8) {
                i = borrar(datos, i);
            }
            else {
                printf("%c", c);
                datos[i++] = c;
            }

        }
    }
    datos[i] = '\0';
    return atoi(datos);
}
```

11.3. Listas Simples Circulares

Una lista circular simple es una lista en la cual el nodo que sigue al último es el primero. Es decir, el último nodo tiene como sucesor al primero de la lista, logrando con ello tener acceso nuevamente a todos los miembros de la lista. Esta característica permite que desde cualquier nodo de esta estructura de datos se tenga acceso a cualquiera de los otros nodos de esta (Silvia Guardati, 2017).

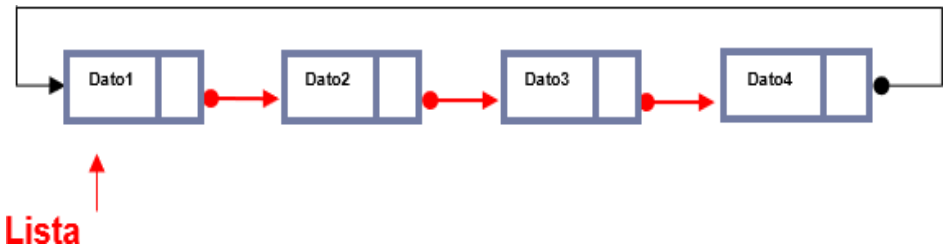

Imagen 66 *Lista simple circular*

Ejemplo:

Main.cpp

```
#include <iostream>
#include "ListaCircular.h"
#include "Validacion.h"

int main(){
    ListaCircular lista;
    do {
        system("cls");
        std::cout << "\n\tLISTA SIMPLE CIRCULAR\n\n" <<
            "1. Insertar al inicio\n" <<
            "2. Eliminar\n" <<
            "3. Mostrar\n" <<
            "4. Salir\n\n";
        switch (ingresar_enteros("Ingresar la opcion:  ")) {
        case 1:
            lista.insertarAlInicio(ingresar_enteros("\n\nIngresar un numero:
"));
            std::cout << std::endl;
            system("pause");
            break;
        case 2:
            lista.eliminar(ingresar_enteros("\n\nIngresar el numero a
eliminar:  "));
            std::cout << std::endl;
            system("pause");
            break;
        case 3:
            lista.imprimir();
            std::cout << std::endl;
            system("pause");
            break;
        case 4:
            return 0;
        default:
            break;
        }
```

```
    } while (true);
}
```

ListaCircular.h

```
#pragma once
#include <iostream>

// Definición de la clase Nodo
class Nodo {
public:
    int dato;
    Nodo* siguiente;

    // Constructor
    Nodo(int valor) : dato(valor), siguiente(nullptr) {}
};

// Clase ListaCircular que contiene operaciones básicas
class ListaCircular {
private:
    Nodo* cabeza;

public:
    // Constructor
    ListaCircular() : cabeza(nullptr) {}

    // Método para verificar si la lista está vacía
    bool estaVacia() {
        return cabeza == nullptr;
    }

    // Método para insertar un elemento al inicio de la lista
    void insertarAlInicio(int valor) {
        Nodo* nuevoNodo = new Nodo(valor);
        if (estaVacia()) {
            nuevoNodo->siguiente = nuevoNodo;
            cabeza = nuevoNodo;
        }
        else {
            nuevoNodo->siguiente = cabeza->siguiente;
            cabeza->siguiente = nuevoNodo;
        }
    }

    // Método para imprimir la lista
    void imprimir() {
        if (estaVacia()) {
```

```
            std::cout << "La lista está vacía." << std::endl;
            return;
        }

        Nodo* temp = cabeza->siguiente;
        std::cout << "Lista: ";
        do {
            std::cout << temp->dato << " -> ";
            temp = temp->siguiente;
        } while (temp != cabeza->siguiente);
        std::cout << "(Cabeza)" << std::endl;
    }

    // Método para eliminar un elemento de la lista dada una clave (valor)
    void eliminar(int clave) {
        if (estaVacia()) {
            std::cout << "La lista está vacía." << std::endl;
            return;
        }

        Nodo* temp = cabeza;
        Nodo* prev = nullptr;

        // Buscar el nodo con la clave dada
        do {
            prev = temp;
            temp = temp->siguiente;
            if (temp->dato == clave) {
                prev->siguiente = temp->siguiente;
                if (temp == cabeza)
                    cabeza = prev;
                delete temp;
                std::cout << "Elemento " << clave << " eliminado." <<
std::endl;
                return;
            }
        } while (temp != cabeza->siguiente);

        std::cout << "Elemento " << clave << " no encontrado en la lista." <<
std::endl;
    }

    // Destructor para liberar la memoria asignada a los nodos de la lista
    ~ListaCircular() {
        if (!estaVacia()) {
            Nodo* temp = cabeza->siguiente;
            while (temp != cabeza) {
```

```
                Nodo* siguiente = temp->siguiente;
                delete  temp;
                temp = siguiente;
            }
            delete cabeza;
            cabeza = nullptr;
        }
    }
};
```

Validacion.h

```
#pragma once
#include <conio.h>
#include <iostream>

int borrar(char* datos, int& i) {
    if (i > 0) {
        printf("\b \b");
        i--;
        datos[i] = '\0';
        return i;
    }
    return 0;
}

int ingresar_enteros(std::string msj) {
    char* datos = new char[10];
    char c;
    int i = 0;

    std::cout << msj;
    while ((c = _getch()) != 13 && i < 9) {
        if ((c >= '0' && c <= '9') || c == 8) {
            if (c == 8) {
                i = borrar(datos, i);
            }
            else {
                printf("%c", c);
                datos[i++] = c;
            }

        }
    }
    datos[i] = '\0';
    return atoi(datos);
}
```

11.4. Listas Dobles Circulares.

Las listas doblemente ligadas son otra variante de las estructuras vistas en las secciones previas. En las listas simplemente ligadas cada nodo conoce solamente la dirección de su nodo sucesor. De ahí la importancia de no perder el puntero al primer nodo de esta. Por su parte, en las listas doblemente ligadas, cada nodo conoce la dirección de su predecesor y de su sucesor. La excepción es el primer nodo de la lista que no cuenta con predecesor, y el último que no tiene sucesor. Debido a esta característica, se puede visitar a todos los componentes de la lista a partir de cualquiera de ellos (Silvia Guardati, 2017).

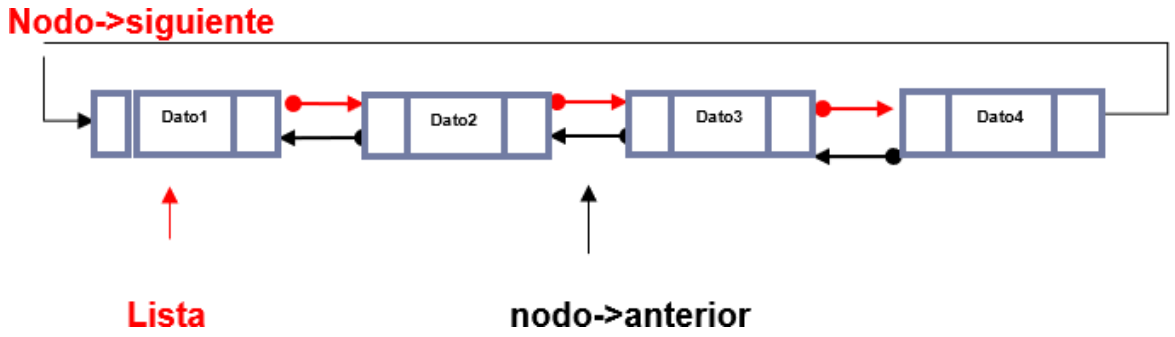

Imagen 67 *Lista doble circular*

Ejemplo:

Main.h

```
#include <iostream>
#include "ListaDobleCircular.h"
#include "Validacion.h"

int main(){
    ListaDobleCircular lista;
    do {
        system("cls");
        std::cout << "\n\tLISTA DOBLE CIRCULAR\n\n" <<
            "1. Insertar al inicio\n" <<
            "2. Eliminar\n" <<
            "3. Mostrar\n" <<
            "4. Salir\n\n";
        switch (ingresar_enteros("Ingresar la opcion:  ")) {
        case 1:
            lista.insertarAlInicio(ingresar_enteros("\n\nIngresar un numero:
"));

            std::cout << std::endl;
            system("pause");
            break;
        case 2:
            lista.eliminar(ingresar_enteros("\n\nIngresar el numero a
```

```
eliminar:  "));
            std::cout << std::endl;
            system("pause");
            break;
        case 3:
            lista.imprimir();
            std::cout << std::endl;
            system("pause");
            break;
        case 4:
            return 0;
        default:
            break;
        }
    } while (true);
}
```

ListaDobleCircular.h

```
#pragma once
#include <iostream>

// Definición de la clase Nodo
class Nodo {
public:
    int dato;
    Nodo* siguiente;
    Nodo* anterior;

    // Constructor
    Nodo(int valor) : dato(valor), siguiente(nullptr), anterior(nullptr) {}
};

// Clase ListaDobleCircular que contiene operaciones básicas
class ListaDobleCircular {
private:
    Nodo* cabeza;

public:
    // Constructor
    ListaDobleCircular() : cabeza(nullptr) {}

    // Método para verificar si la lista está vacía
    bool estaVacia() {
        return cabeza == nullptr;
    }

    // Método para insertar un elemento al inicio de la lista
```

```
void insertarAlInicio(int valor) {
    Nodo* nuevoNodo = new Nodo(valor);
    if (estaVacia()) {
        nuevoNodo->siguiente = nuevoNodo;
        nuevoNodo->anterior = nuevoNodo;
        cabeza = nuevoNodo;
    }
    else {
        nuevoNodo->siguiente = cabeza;
        nuevoNodo->anterior = cabeza->anterior;
        cabeza->anterior->siguiente = nuevoNodo;
        cabeza->anterior = nuevoNodo;
        cabeza = nuevoNodo;
    }
}

// Método para imprimir la lista
void imprimir() {
    if (estaVacia()) {
        std::cout << "La lista está vacía." << std::endl;
        return;
    }

    Nodo* temp = cabeza;
    std::cout << "Lista hacia adelante: ";
    do {
        std::cout << temp->dato << " <-> ";
        temp = temp->siguiente;
    } while (temp != cabeza);
    std::cout << "(Cabeza)" << std::endl;

    temp = cabeza->anterior;
    std::cout << "Lista hacia atrás: ";
    do {
        std::cout << temp->dato << " <-> ";
        temp = temp->anterior;
    } while (temp != cabeza->anterior);
    std::cout << "(Cabeza)" << std::endl;
}

// Método para eliminar un elemento de la lista dada una clave (valor)
void eliminar(int clave) {
    if (estaVacia()) {
        std::cout << "La lista está vacía." << std::endl;
        return;
    }
```

```
        Nodo* temp = cabeza;
        do {
            if (temp->dato == clave) {
                if (temp == cabeza) {
                    cabeza = temp->siguiente;
                }
                temp->anterior->siguiente = temp->siguiente;
                temp->siguiente->anterior = temp->anterior;
                delete temp;
                std::cout << "Elemento " << clave << " eliminado." <<
std::endl;
                return;
            }
            temp = temp->siguiente;
        } while (temp != cabeza);
        std::cout << "Elemento " << clave << " no encontrado en la lista." <<
std::endl;
    }

    // Destructor para liberar la memoria asignada a los nodos de la lista
    ~ListaDobleCircular() {
        if (!estaVacia()) {
            Nodo* temp = cabeza->siguiente;
            while (temp != cabeza) {
                Nodo* siguiente = temp->siguiente;
                delete  temp;
                temp = siguiente;
            }
            delete cabeza;
            cabeza = nullptr;
        }
    }
};
```

Validacion.h

```
#pragma once
#include <conio.h>
#include <iostream>

int borrar(char* datos, int& i) {
    if (i > 0) {
        printf("\b \b");
        i--;
        datos[i] = '\0';
        return i;
    }
    return 0;
}
```

```
int ingresar_enteros(std::string msj) {
    char* datos = new char[10];
    char c;
    int i = 0;

    std::cout << msj;
    while ((c = _getch()) != 13 && i < 9) {
        if ((c >= '0' && c <= '9') || c == 8) {
            if (c == 8) {
                i = borrar(datos, i);
            }
            else {
                printf("%c", c);
                datos[i++] = c;
            }

        }
    }
    datos[i] = '\0';
    return atoi(datos);
}
```

12. PILAS

12.1. Representación en memoria estática y dinámica

Una pila es una estructura de datos lineal que sigue el principio de LIFO (Last In, First Out), lo que significa que el último elemento agregado a la pila es el primero en ser eliminado. Las operaciones principales en una pila son la inserción (push), que agrega un elemento a la parte superior de la pila, y la eliminación (pop), que elimina el elemento superior de la pila.

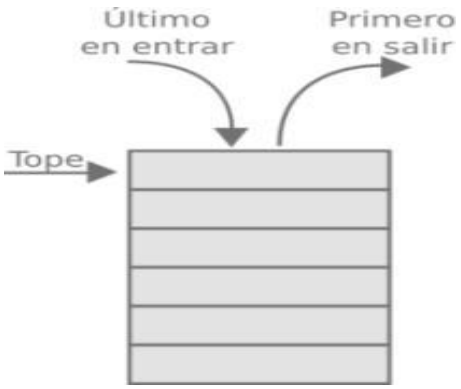

Imagen 68: Estructura de la Pila

12.2. Representación en memoria estática:

En la representación estática de una pila, se reserva un bloque de memoria de tamaño fijo durante la compilación del programa.

- La cantidad de memoria asignada para la pila se determina antes de tiempo y no cambia durante la ejecución del programa.
- Se utiliza un arreglo estático para almacenar los elementos de la pila.
- El acceso a los elementos se realiza mediante índices.

Una desventaja es que la capacidad de la pila está limitada por el tamaño del arreglo estático, lo que puede llevar a problemas de desbordamiento si se intenta agregar más elementos de los que el arreglo puede contener.

12.2.1. Ejemplo:

En la implementación de una pila utilizando memoria estática en C++, se puede utilizar un arreglo para almacenar los elementos de la pila.

PilaEstatica.h:

```
#ifndef PILA_ESTATICA_H
#define PILA_ESTATICA_H

#include <iostream>

#define TAM_MAX 100 // Tamaño máximo de la pila

class PilaEstatica {
private:
    int arreglo[TAM_MAX];
    int tope; // Índice del elemento superior de la pila
public:
    PilaEstatica();
    void empujar(int dato);
    void sacar();
    int elementoTope();
    bool estaVacia();
};

#endif
```

PilaEstatica.cpp:

```
#include "PilaEstatica.h"

PilaEstatica::PilaEstatica() {
    tope = -1; // Inicializar la pila como vacía
```

```
}

void PilaEstatica::empujar(int dato) {
    if (tope == TAM_MAX - 1) {
        std::cout << "Error: La pila está llena." << std::endl;
        return;
    }
    arreglo[++tope] = dato;
}

void PilaEstatica::sacar() {
    if (tope == -1) {
        std::cout << "Error: La pila está vacía." << std::endl;
        return;
    }
    tope--;
}

int PilaEstatica::elementoTope() {
    if (tope == -1) {
        std::cout << "Error: La pila está vacía." << std::endl;
        return -1;
    }
    return arreglo[tope];
}

bool PilaEstatica::estaVacia() {
    return tope == -1;
}
```

main.cpp

```
#include <iostream>
#include "PilaEstatica.h"

int main() {
    PilaEstatica pila;
    pila.empujar(10);
    pila.empujar(20);
    pila.empujar(30);

    std::cout << "Elemento superior de la pila: " <<
pila.elementoTope() << std::endl;
    pila.sacar();
    std::cout << "Elemento superior de la pila después de sacar(): "
<< pila.elementoTope() << std::endl;

    return 0;}
```

12.3. Representación en memoria dinámica:

En la representación dinámica de una pila, se utiliza la asignación de memoria dinámica durante la ejecución del programa.

- No se reserva un tamaño fijo de memoria durante la compilación; en su lugar, la memoria se asigna y libera según sea necesario durante la ejecución del programa.
- Se utiliza una estructura de datos enlazada, como una lista enlazada, para almacenar los elementos de la pila.
- Cada elemento de la pila se almacena en un nodo de la lista enlazada, y los nodos están enlazados entre sí mediante punteros.

Esta representación permite que la pila crezca y se reduzca dinámicamente según sea necesario, lo que es útil cuando la cantidad de elementos de la pila puede variar durante la ejecución del programa.

Una desventaja potencial es el costo adicional de la asignación y liberación de memoria dinámica, así como la necesidad de gestionar correctamente la liberación de la memoria para evitar fugas de memoria.

12.3.1. Ejemplo:

En la implementación de una pila utilizando memoria dinámica en C++, se puede utilizar una lista enlazada

PilaDinamica.h

```
#ifndef PILA_DINAMICA_H
#define PILA_DINAMICA_H

#include <iostream>

class PilaDinamica {
private:
    class Nodo {
    public:
        int dato;
        Nodo* siguiente;
        Nodo(int valor) : dato(valor), siguiente(nullptr) {}
    };

    Nodo* tope; // Puntero al elemento superior de la pila
public:
    PilaDinamica();
    void empujar(int dato);
    void sacar();
    int elementoTope();
    bool estaVacia();
    ~PilaDinamica(); // Destructor
};
```

```
#endif // PILA_DINAMICA_H
```

PilaDinamica.cpp

```
#include "PilaDinamica.h"

PilaDinamica::PilaDinamica() {
    tope = nullptr; // Inicializar la pila como vacía
}

void PilaDinamica::empujar(int dato) {
    Nodo* nuevoNodo = new Nodo(dato);
    nuevoNodo->siguiente = tope;
    tope = nuevoNodo;
}

void PilaDinamica::sacar() {
    if (tope == nullptr) {
        std::cout << "Error: La pila está vacía." << std::endl;
        return;
    }
    Nodo* temp = tope;
    tope = tope->siguiente;
    delete temp;
}

int PilaDinamica::elementoTope() {
    if (tope == nullptr) {
        std::cout << "Error: La pila está vacía." << std::endl;
        return -1;
    }
    return tope->dato;
}

bool PilaDinamica::estaVacia() {
    return tope == nullptr;
}

PilaDinamica::~PilaDinamica() {
    while (tope != nullptr) {
        sacar();
    }
}
```

main.cpp

```
#include <iostream>
#include "PilaDinamica.h"
```

```
int main() {
    PilaDinamica pila;
    pila.empujar(10);
    pila.empujar(20);
    pila.empujar(30);

    std::cout << "Elemento superior de la pila: " <<
pila.elementoTope() << std::endl;
    pila.sacar();
    std::cout << "Elemento superior de la pila después de sacar(): "
<< pila.elementoTope() << std::endl;

    return 0;
}
```

12.4. Conclusiones

- La implementación utilizando memoria estática es más simple y directa, ya que no requiere la gestión manual de la memoria.
- Sin embargo, la implementación utilizando memoria dinámica permite que la pila crezca dinámicamente según sea necesario, lo que puede ser más eficiente en términos de uso de memoria en casos donde la cantidad de elementos de la pila es variable.
- La elección entre memoria estática y dinámica depende de los requisitos específicos del programa y del rendimiento deseado. En general, si la cantidad máxima de elementos de la pila es conocida y limitada, la memoria estática puede ser preferible por su simplicidad y eficiencia. Por otro lado, si la cantidad de elementos puede variar o es desconocida, la memoria dinámica puede ser más adecuada.

13. OPERACIONES BÁSICAS CON PILAS

13.1. Introducción de Operaciones básicas con pilas

Las pilas son estructuras de datos fundamentales en informática que siguen el principio de "último en entrar, primero en salir" (LIFO, por sus siglas en inglés). Las operaciones básicas con pilas son esenciales para manipular y gestionar estos elementos de manera eficiente.

13.2. Definición

Una pila es una colección de elementos donde las operaciones de inserción y eliminación se realizan solo en un extremo llamado "cima" o "tope". Las operaciones básicas incluyen:

- Push: Agrega un elemento a la cima de la pila.
- Pop: Elimina el elemento en la cima de la pila.
- Peek o Top: Observa el elemento en la cima sin eliminarlo.
- isEmpty: Verifica si la pila está vacía.

13.3. Importancia de las Operaciones básicas con pilas

Las operaciones básicas con pilas son cruciales en muchas áreas de la informática, incluyendo algoritmos, compiladores, sistemas operativos y manejo de memoria. Proporcionan un mecanismo simple pero poderoso para gestionar datos de manera eficiente, permitiendo la reversión de operaciones y el control de flujo en muchas aplicaciones *Stein, C. (2009)*

13.4. Uso de las Operaciones básicas con pilas

Las pilas se utilizan en situaciones donde la reversión de operaciones o la gestión de tareas de manera inversa es necesaria. Algunos ejemplos comunes de uso incluyen:

- Evaluación de expresiones matemáticas.
- Implementación de algoritmos de búsqueda en profundidad.
- Manejo de llamadas a funciones en la ejecución de programas.
- Implementación de la funcionalidad "deshacer" en aplicaciones de software.

13.5. Implementación de Operaciones básicas con pilas

La implementación de operaciones básicas con pilas puede realizarse utilizando estructuras de datos como matrices o listas enlazadas. Es crucial mantener un seguimiento del tope de la pila y garantizar que las operaciones se realicen correctamente para evitar desbordamientos o subdesbordamientos *Tamassia, R. (2011).*

```
#include <iostream>

// Definición de la clase NodoPila
class NodoPila {
public:
    int dato;
    NodoPila* siguiente;

    NodoPila(int dato) {
        this->dato = dato;
        this->siguiente = nullptr;
    }
};

// Definición de la clase Pila
class Pila {
private:
```

```
    NodoPila* tope;

public:
    Pila() {
        tope = nullptr;
    }

    ~Pila() {
        while (!isEmpty()) {
            pop();
        }
    }

    bool isEmpty() {
        return tope == nullptr;
    }

    void push(int dato) {
        NodoPila* nuevo_nodo = new NodoPila(dato);
        nuevo_nodo->siguiente = tope;
        tope = nuevo_nodo;
    }

    int pop() {
        if (isEmpty()) {
            std::cerr << "Error: La pila está vacía\n";
            return -1; // Valor de retorno para indicar un
error
        }

        int dato_pop = tope->dato;
        NodoPila* temp = tope;
        tope = tope->siguiente;
        delete temp;
        return dato_pop;
    }

    int peek() {
        if (isEmpty()) {
            std::cerr << "Error: La pila está vacía\n";
            return -1; // Valor de retorno para indicar un
error
```

```
        }

        return tope->dato;
    }
};

// Función principal
int main() {
    Pila pila;
    std::cout << "La pila está vacía: " << std::boolalpha
<< pila.isEmpty() << "\n";

    pila.push(10);
    pila.push(20);
    pila.push(30);

    std::cout << "Elemento en la cima de la pila: " <<
pila.peek() << "\n";

    std::cout << "Sacando elementos de la pila:\n";
    while (!pila.isEmpty()) {
        std::cout << pila.pop() << "\n";
    }

    std::cout << "La pila está vacía: " << std::boolalpha
<< pila.isEmpty() << "\n";

    return 0;
}
```

13.6. Recursividad con ayuda de pilas.

Las pilas se usan comúnmente para indicar en qué punto se encuentra un programa cuando contiene procedimientos que se llaman a sí mismos. Estos procedimientos se conocen como procedimientos recursivos.

La recursividad es una técnica en la que un procedimiento o función se hace llamadas a sí mismo en el proceso de realización de sus tareas. La recursividad se puede definir mediante un clásico ejemplo de la función factorial. La recursividad es una técnica de programación muy potente que puede ser usada en lugar de una iteración (Bucles o ciclos). Ello implica una forma diferente de ver las acciones repetitivas permitiendo que un subprograma se llame a sí mismo para resolver una operación más pequeña del programa original.

13.7. Conclusiones

Las operaciones básicas con pilas son fundamentales para la computación y ofrecen un mecanismo eficiente para gestionar datos en muchos contextos. Comprender y dominar estas operaciones es esencial para desarrollar algoritmos y aplicaciones eficientes y robustas.

14. NOTACIÓN INFIJA, PREFIJA Y POSTFIJA

14.1. Notación infija

"En notación infija, los operadores se escriben entre los operandos, lo que requiere la definición de las reglas para determinar el orden de las operaciones" (Knuth, 1974).

Esta notación es la más habitual, tiene la jerarquía de operadores dando prioridad al operador que este dentro del paréntesis, luego al exponente, seguido de multiplicación y división y por último la suma y la resta.

14.1.1. **Ejemplo:** (1 * (2 - 3)) + (4 + 5)

14.2. Notación prefija

"La notación prefija es un sistema de escritura matemática en la que cada operador sigue a sus operandos y no hay necesidad de paréntesis para indicar el orden de las operaciones" (Łukasiewicz, 1929).

Esta notación el operador mas cercano al operando es el de mayor jerarquía, es una operación binaria, es decir, toma de a dos operandos y un operador para realizar los cálculos. Para resolver la notación prefija o polaca tomamos al primer operador que tenga dos operandos seguidos, el resultado se guarda y se sigue al siguiente operador que tenga dos operadores seguidos y así sucesivamente.

14.2.1. Ejemplo:

Transformando el ejemplo de la notación infija:

Entrada	Pila	Salida
(1 * (2 - 3)) + (4 + 5)		
(1 * (2 - 3)) + (4 + 5	)	
(1 * (2 - 3)) + (4 +	)	5
(1 * (2 - 3)) + (4	) +	5
(1 * (2 - 3)) + (	) +	5 4

(1 * (2 - 3)) +	)+(	5 4
(1 * (2 - 3))	+	5 4 +
(1 * (2 - 3)	+)	5 4 +
(1 * (2 - 3	+))	5 4 +
(1 * (2 -	+))	5 4 + 3
(1 * (2	+)) -	5 4 + 3
(1 * (	+)) -	5 4 + 3 2
(1 *	+)) - (	5 4 + 3 2
(1	+) *	5 4 + 3 2 -
(	+) *	5 4 + 3 2 - 1
	+) * (	5 4 + 3 2 - 1
	+	5 4 + 3 2 - 1 *
		5 4 + 3 2 - 1 * +
Invertimos el resultado		
		+ * 1 - 3 2 + 4 5

Tabla 1. Transformación de Notación Infija a Prefija

14.3. Notación postfija

"En notación postfija, los operadores siguen a sus operandos, eliminando así la necesidad de paréntesis y reglas de precedencia" (Hamblin, 1957).

Esta notación no es necesariamente el contrario de la prefija, aunque tengan sus similitudes con respecto a la jerarquía de operadores. Para la transformación cambia de sentido, mientras en la prefija se lee de derecha a izquierda en la postfija se lee de izquierda a derecha, tomando como refería el operando binario más cercano al operador.

14.3.1. Ejemplo:

Transformando el ejemplo de la notación infija:

Entrada	Pila	Salida
(1 * (2 - 3)) + (4 + 5)		
1 * (2 - 3)) + (4 + 5)	(	
* (2 - 3)) + (4 + 5)	(	1
(2 - 3)) + (4 + 5)	(*	1
2 - 3)) + (4 + 5)	(* (	1
- 3)) + (4 + 5)	(* (	1 2
3)) + (4 + 5)	(* (-	1 2
)) + (4 + 5)	(* (-	1 2 3

) + (4 + 5)	(* (-)	1 2 3
+ (4 + 5)	(*)	1 2 3 -
(4 + 5)	+	1 2 3 - *
4 + 5)	+ (	1 2 3 - *
+ 5)	+ (	1 2 3 - * 4
5)	+ (+	1 2 3 - * 4
)	+ (+	1 2 3 - * 4 5
	+ (+)	1 2 3 - * 4 5
	+	1 2 3 - * 4 5 +
		1 2 3 - * 4 5 + +
Resultado		
		1 2 3 - * 4 5 + +

Tabla 2. Transformación de Notación Infija a Postfija

15. REPRESENTACIÓN EN MEMORIA ESTÁTICA Y DINÁMICA

15.1. Introducción

La representación en memoria, tanto estática como dinámica, es fundamental en la implementación de estructuras de datos como las colas. En este documento, exploraremos cómo se puede representar una cola en memoria estática y dinámica, así como las ventajas y desventajas de cada enfoque.

15.2. Representación en Memoria Estática de Colas

En la representación estática de una cola, se utiliza un arreglo de tamaño fijo para almacenar los elementos. Se asigna una cantidad de memoria predeterminada en tiempo de compilación para la cola. A continuación, se describen los aspectos clave de la representación estática de colas:

15.2.1. Implementación con Arreglos

Una cola estática se puede implementar utilizando un arreglo de tamaño fijo. Se requieren dos índices adicionales, frente y final, para realizar las operaciones de inserción y eliminación.

15.2.2. Ventajas

- Simplicidad: La implementación con arreglos es simple y directa.

- Eficiencia: Acceso rápido a los elementos debido a la indexación directa en el arreglo.

15.2.3. Desventajas

- Tamaño Fijo: La cola está limitada por el tamaño máximo del arreglo, lo que puede llevar a desbordamientos o subutilización de la memoria.

15.3. Representación en Memoria Dinámica de Colas

En la representación dinámica de una cola, se utiliza la asignación de memoria dinámica para permitir que la cola crezca o disminuya según sea necesario. A continuación, se describen los aspectos clave de la representación dinámica de colas:

15.3.1. Implementación con Listas Enlazadas

Una cola dinámica se puede implementar utilizando una lista enlazada, donde cada nodo contiene un elemento y un puntero al siguiente nodo.

15.3.2. Ventajas

- Flexibilidad: La cola puede crecer o disminuir dinámicamente según la demanda.
- Uso Eficiente de la Memoria: Se asigna memoria solo cuando se necesite, lo que evita el despilfarro de memoria.

15.3.3. Desventajas

- Overhead: La gestión de punteros y la asignación dinámica de memoria pueden aumentar la complejidad y el overhead del código.

15.4. Ejemplos Prácticos

A continuación, se presentan ejemplos de implementaciones estáticas y dinámicas de colas en lenguaje C++ para ilustrar los conceptos discutidos anteriormente.

15.4.1. Representación Estática con Arreglos

```
const int MAX_SIZE = 100;

class StaticQueue {
private:
```

```
    int data[MAX_SIZE];
    int frente;
    int final;

public:
    StaticQueue() {
        frente = -1;
        final = -1;
    }

    bool estaVacia() {
        return frente == -1;
    }

    bool estaLlena() {
        return (final + 1) % MAX_SIZE == frente;
    }

    void encolar(int elemento) {
        if (estaLlena()) {
            std::cout << "La cola está llena. No se puede encolar más
elementos.\n";
            return;
        }
        if (estaVacia()) {
            frente = 0;
        }
        final = (final + 1) % MAX_SIZE;
        data[final] = elemento;
        std::cout << "Elemento " << elemento << " encolado con
éxito.\n";
    }

    void desencolar() {
        if (estaVacia()) {
            std::cout << "La cola está vacía. No se puede
desencolar.\n";
            return;
        }
        std::cout << "Desencolando elemento " << data[frente] <<
".\n";
        if (frente == final) {
            frente = -1;
            final = -1;
        } else {
            frente = (frente + 1) % MAX_SIZE;
        }
    }
}
```

15.4.2. Representación Dinámica con Listas Enlazadas

```
class Nodo {
public:
    int dato;
    Nodo* siguiente;

    Nodo(int d) {
        dato = d;
        siguiente = nullptr;
    }
};

class DynamicQueue {
private:
    Nodo* frente;
    Nodo* final;

public:
    DynamicQueue() {
        frente = nullptr;
        final = nullptr;
    }

    bool estaVacia() {
        return frente == nullptr;
    }

    void encolar(int elemento) {
        Nodo* nuevoNodo = new Nodo(elemento);
        if (estaVacia()) {
            frente = nuevoNodo;
        } else {
            final->siguiente = nuevoNodo;
        }
        final = nuevoNodo;
        std::cout << "Elemento " << elemento << " encolado con
éxito.\n";
    }

    void desencolar() {
        if (estaVacia()) {
            std::cout << "La cola está vacía. No se puede
desencolar.\n";
            return;
        }
        Nodo* temp = frente;
        frente = frente->siguiente;
        std::cout << "Desencolando elemento " << temp->dato << ".\n";
        delete temp;
        if (frente == nullptr) {
```

```
                final = nullptr;
            }
        }
    }
```

15.5. Conclusiones

Tanto la representación estática como la dinámica de colas tienen sus propias ventajas y desventajas. La elección entre una u otra dependerá de las necesidades específicas del proyecto, incluidos los requisitos de rendimiento, la flexibilidad y la eficiencia en el uso de la memoria.

16. OPERACIONES CON COLAS

16.1. Introducción

Las colas son estructuras de datos fundamentales en informática y programación. Siguen el principio de "primero en entrar, primero en salir" (FIFO, por sus siglas en inglés: First In, First Out), lo que significa que el primer elemento en ser añadido a la cola será el primero en ser eliminado. En este manual, exploraremos en detalle las operaciones básicas que se pueden realizar con colas, así como algunos consejos prácticos para su implementación y uso eficiente.

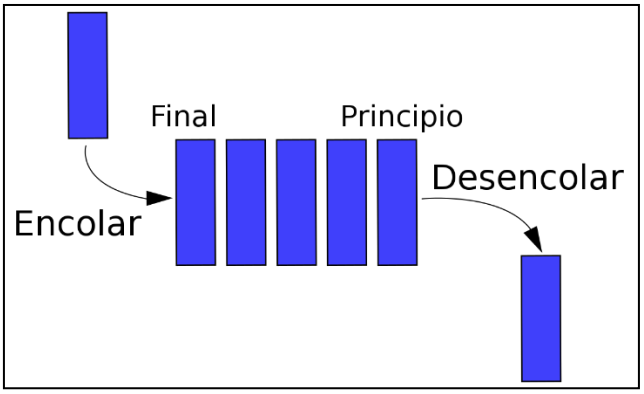

Imagen 69 *Procesos de una cola*

16.2. Operaciones Básicas

Las operaciones básicas que se pueden realizar con una cola son:

- **Encolar**: Añade un elemento al final de la cola.
- **Desencolar**: Elimina y devuelve el elemento al principio de la cola.
- **Frente**: Devuelve el elemento que está al principio de la cola, sin eliminarlo.

- **Tamaño**: Devuelve el número de elementos en la cola.
- **Vacío**: Verifica si la cola está vacía o no.

16.3. Implementación en Pseudocódigo

A continuación, presentamos una implementación básica de las operaciones de una cola en pseudocódigo:

```
cola = ColaVacía()
Encolar(cola, elemento):
   Agregar elemento al final de la cola
Desencolar(cola):
   Si la cola está vacía, devolver error
   Sino:
      elemento = Primer elemento de la cola
      Eliminar primer elemento de la cola
      Devolver elemento
Frente(cola):
   Si la cola está vacía, devolver error
   Sino:
      Devolver primer elemento de la cola
Tamaño(cola):
   Devolver el número de elementos en la cola
Vacío(cola):
   Devolver verdadero si la cola está vacía, falso en caso contrario
Tamaño(cola):
   Devolver el número de elementos en la cola
Vacío(cola):
   Devolver verdadero si la cola está vacía, falso en caso contrario
```

16.4. Teoría y Ejemplos

Encolar: Esta operación añade un elemento al final de la cola. Es similar a añadir una persona al final de una fila. Por ejemplo, si estamos simulando una fila en un banco, cada vez que alguien llegue a la fila, se le añadirá al final para ser atendido en el orden en el que llegó.

Desencolar: La operación de desencolar elimina y devuelve el elemento al principio de la cola. Siguiendo con el ejemplo de la fila en el banco, cuando un cajero atiende a un cliente, este cliente es el primero en ser desencolado de la fila para ser atendido.

Frente: Devuelve el elemento que está al principio de la cola, sin eliminarlo. Es como mirar quién es el siguiente en la fila del banco sin hacer que esa persona salga de la fila.

Tamaño: Devuelve el número de elementos en la cola. Es útil para saber cuántas personas hay en la fila del banco en un momento dado.

Vacío: Verifica si la cola está vacía o no. Si no hay nadie en la fila del banco, la cola está vacía.

16.5. Consejos Prácticos

Elección de la Estructura de Datos Correcta: Antes de utilizar una cola, asegúrate de que es la estructura de datos adecuada para tu problema. Por ejemplo, si necesitas acceso rápido a elementos específicos, una lista enlazada o un arreglo podrían ser más apropiados.

Gestión de Errores: Siempre verifica si la cola está vacía antes de intentar desencolar elementos para evitar errores.

Eficiencia: Considera el rendimiento de las operaciones encolar y desencolar, especialmente en grandes conjuntos de datos. Algunas implementaciones de colas, como las colas de doble extremo, pueden proporcionar un mejor rendimiento en ciertos casos.

Uso de Colas en Procesos en Paralelo: Las colas son útiles en entornos de programación concurrente o paralela para coordinar el trabajo entre múltiples hilos o procesos.

16.6. Conclusiones

Las colas son una estructura de datos esencial con una amplia gama de aplicaciones en informática y programación. Con una comprensión clara de sus operaciones básicas y algunos consejos prácticos para su implementación eficiente, puedes utilizar colas de manera efectiva para resolver una variedad de problemas en tus proyectos de desarrollo de software.

Recuerda siempre considerar el contexto específico de tu aplicación y elegir la estructura de datos que mejor se adapte a tus necesidades. Con la

práctica y la experiencia, dominarás el uso de colas y otras estructuras de datos, mejorando así la calidad y eficiencia de tus programas.

17. TIPOS DE COLAS

17.1. Introducción

Previamente en el curso habíamos visto un concepto básico de colas, en este entendimos que se trata de un tipo de estructura de datos FIFO puesto a que el primer dato ingresado también será el primer dato en salir, sin embargo, estas también se clasifican dependiendo de ciertas características, en este capítulo se nombraran algunas de ellas, junto con una breve descripción gráfica y un código representativo de estas. Se explicará de la manera más detallada posible para una mejor comprensión del lector.

17.2. Colas de Prioridades

Las colas de prioridad son estructuras de datos que gestionan elementos según su prioridad. A diferencia de una cola tradicional, donde el primer elemento en entrar es el primero en salir (FIFO), en una cola de prioridad, los elementos se eliminan de acuerdo con su nivel de prioridad. Es decir, el elemento con la mayor (o en algunos casos, menor) prioridad es el próximo en ser eliminado.

Características clave de las colas de prioridad:

- **Prioridad Asociada:**

Cada elemento de la cola de prioridad tiene asociado un valor de prioridad que determina su orden de eliminación.

- **Acceso al Elemento de Máxima Prioridad:**

La principal operación en una cola de prioridad es acceder al elemento con la máxima (o mínima) prioridad sin eliminarlo.

- **Operaciones de Inserción y Eliminación:**

Se pueden insertar nuevos elementos en la cola de prioridad, y cuando se elimina un elemento, generalmente es el que tiene la mayor (o menor) prioridad.

- **Implementación con Estructuras de Datos:**

Las colas de prioridad se pueden implementar utilizando diversas estructuras de datos, como montículos (heaps), árboles binarios de búsqueda balanceados, o arreglos con técnicas de ordenamiento específicas.

Casos de Uso:

Son útiles en situaciones donde es necesario manejar elementos en función de su urgencia, importancia o algún otro criterio de prioridad. Se utilizan en algoritmos y aplicaciones donde la planificación o el procesamiento deben realizarse en un orden específico.

17.3. Tipos de Colas de Prioridad:

Existen colas de prioridad de máxima y de mínima. En una cola de prioridad de máxima, el elemento con la mayor prioridad se elimina primero. En una cola de prioridad de mínima, el elemento con la menor prioridad se elimina primero.

Ejemplos de aplicaciones de colas de prioridad incluyen la planificación de tareas, algoritmos de búsqueda de caminos mínimos, gestión de procesos en sistemas operativos, entre otros. La flexibilidad y eficiencia en la gestión de prioridades hacen que las colas de prioridad sean una herramienta valiosa en diversas áreas de la informática y la ingeniería de software.

.

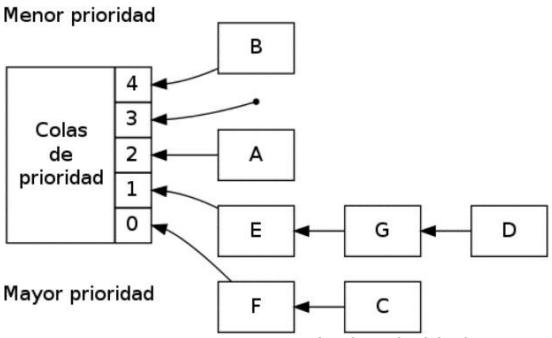

Imagen 70: *Cola de prioridad.*

17.4. Colas circulares

Una cola circular es aquella en la cual el sucesor del último elemento es el primero. Por lo tanto, el manejo de las colas como estructuras circulares permite un mejor uso del espacio de memoria reservado para la implementación de las mismas. La Imagen 71 corresponde a la representación gráfica de una cola circular. Observe que el siguiente elemento del último es el primero. (Silvia Guardati, 2017, p 231).

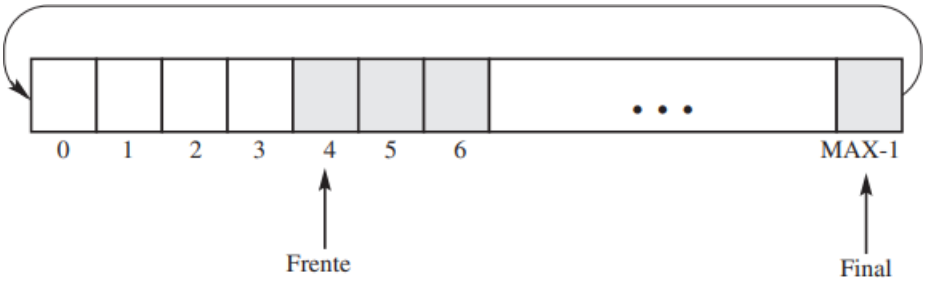

Imagen 71. *Estructura de una cola circular.*

La Imagen 72 presenta el esquema correspondiente a una cola circular, en la cual el final se movió hacia el inicio de la cola, teniendo un valor menor al frente. En este ejemplo, (Silvia Guardati, 2017) la cola tiene las posiciones 4 a MAX-1 y

0 a 1 ocupadas, siendo el primero elemento a salir el que está en la posición 4 y el último insertado el que está en la posición 1.

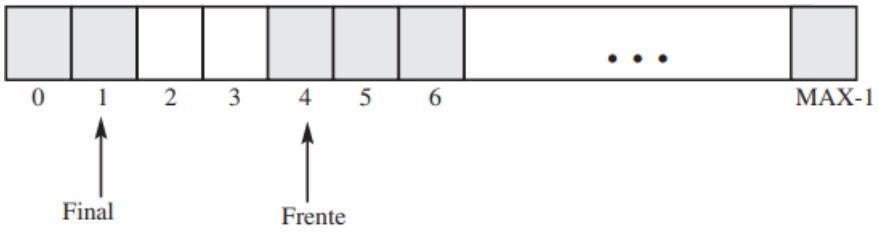

Imagen 72 *Cola circular*

Los algoritmos correspondientes a las operaciones de inserción y eliminación varían al tratarse de colas circulares, es lo referente a la actualización de los punteros. Asimismo, (Silvia Guardati, 2017) la condición para determinar si la cola está llena debe considerar todos los casos que puedan presentarse, que son:

1. El frente en la posición 0 y el final en la posición (MAX -1),0.
2. El (Final+1) es igual al Frente.

Ambos se evalúan por medio de la expresión: (Final +1) % MAX == Frente.

- **Estructura Circular:**
  En una cola circular, los elementos están organizados en una estructura circular, lo que significa que el último elemento está conectado al primer elemento, formando un bucle.
- **Frente y Final:**
  A diferencia de las colas lineales tradicionales, las colas circulares no tienen un "final" o una "cima" claramente definidos. El frente y el final de la cola están conectados en la estructura circular.
- **Uso Eficiente de Espacio:**
  Las colas circulares pueden ser más eficientes en términos de uso de espacio en comparación con las colas lineales, ya que no hay una posición fija al final de la cola que debe permanecer vacía para permitir la expansión.
- **Desbordamiento o Subdesbordamiento:**
  Al usar una implementación basada en arreglos, es importante gestionar adecuadamente el desbordamiento (cuando la cola está llena) y el subdesbordamiento (cuando la cola está vacía). Esto se logra utilizando la aritmética modular para realizar operaciones circulares.
- **Operaciones de Enqueue (Encolar) y Dequeue (Desencolar):**
  Se pueden realizar operaciones de encolar y desencolar en una cola circular. Al encolar elementos, se insertan al final de la cola, y al desencolar elementos, se eliminan desde el frente.
- **Uso Común en Sistemas Embebidos y Ciclos Continuos:**

Debido a su eficiencia y estructura cíclica, las colas circulares son comunes en sistemas embebidos y situaciones donde las operaciones de encolar y desencolar son frecuentes y deben ocurrir de manera continua.

- **Aplicaciones en Algoritmos de Manejo de Datos:**
  Se utilizan en algoritmos y estructuras de datos donde se necesita un acceso cíclico a los elementos, como en ciertos tipos de algoritmos de manejo de datos o en situaciones donde la cola debe mantener un estado constante de actividad.

17.5. Colas dobles

Otra variante de las estructuras tipo cola son las colas dobles. Como su nombre lo indica, estas estructuras permiten realizar las operaciones de la inserción y eliminación por cualquiera de sus extremos. (Silvia Guardati, 2017, p 232). Gráficamente una cola doble se representa de la siguiente manera Imagen 4:

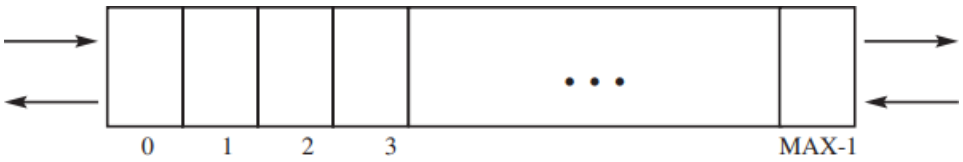

Imagen 73. *Cola doble*

Debido a que este tipo de estructura es una generalización del tipo cola no se presentan aquí las operaciones, (Silvia Guardati, 2017). Al respecto, sólo se menciona que será necesario definir métodos que permitan insertar por el frente y por el final, así como métodos que permitan eliminar por ambos extremos. Las condiciones para determinar el estado de la cola no varían.

Por último, es importante señalar que una doble cola también puede ser circular. En dicho caso, será necesario que los métodos de la inserción y eliminación (sobre cualquier de los extremos) consideren el movimiento adecuado de los punteros. (Silvia Guardati, 2017).

- **Operaciones de Inserción y Eliminación en Ambos Extremos:**
  Una característica fundamental es la capacidad de realizar operaciones de inserción (**push**) y eliminación (**pop**) tanto en el frente como en el final de la cola. Esto proporciona flexibilidad en la manipulación de los elementos.
- **Estructura Lineal:**
  Aunque el término "cola doble" sugiere que tiene dos extremos, la estructura subyacente suele ser lineal. La diferencia clave es que se pueden acceder y manipular ambos extremos.
- **Acceso Eficiente a Ambos Extremos:**
  A diferencia de otras estructuras de datos, como las listas enlazadas, las colas dobles permiten un acceso eficiente tanto al frente como al final. Esto

es beneficioso para operaciones que involucran inserciones o eliminaciones en ambos extremos.

- **Implementación con Arreglos o Listas Enlazadas:**
  Las colas dobles pueden implementarse utilizando arreglos o listas enlazadas. La elección de la implementación dependerá de los requisitos específicos de la aplicación y las operaciones que se realizarán con mayor frecuencia.
- **Casos de Uso Versátiles:**
  Debido a su capacidad para realizar operaciones en ambos extremos, las colas dobles son versátiles y se utilizan en situaciones donde se requiere flexibilidad en la manipulación de elementos, como en algoritmos de procesamiento de datos en tiempo real o sistemas de gestión de eventos.

17.6. Ejemplo de cola circular

A continuación, plantearemos un ejemplo en C++ de una cola circula funcional y adjunto a esto el código del mismo.

colasCirculares.h

```
#include <iostream>

class CircularQueue {
private:
    int *array;
    int frente, final, capacidad;

public:
    CircularQueue(int size);
    ~CircularQueue();

    bool estaVacia();
    bool estaLlena();
    void encolar(int elemento);
    void desencolar();
    void mostrarContenido();

};
```

ColasCirculares.cpp

```
#include "colasCirculares.h"

CircularQueue::CircularQueue(int size) {
```

```
    capacidad = size + 1; // El tamaño real es uno más para
diferenciar entre frente y final.
    array = new int[capacidad];
    frente = final = 0;
}

CircularQueue::~CircularQueue() {
    delete[] array;
}

bool CircularQueue::estaVacia() {
    return frente == final;
}

bool CircularQueue::estaLlena() {
    return (final + 1) % capacidad == frente;
}

void CircularQueue::encolar(int elemento) {
    if (!estaLlena()) {
        array[final] = elemento;
        final = (final + 1) % capacidad;
        std::cout << "Elemento " << elemento << " encolado.\n";
    } else {
        std::cout << "La cola está llena. No se puede encolar.\n";
    }
}

void CircularQueue::desencolar() {
    if (!estaVacia()) {
        int elemento = array[frente];
        frente = (frente + 1) % capacidad;
        std::cout << "Elemento " << elemento << " desencolado.\n";
    } else {
        std::cout << "La cola está vacía. No se puede desencolar.\n";
    }
}

void CircularQueue::mostrarContenido() {
    if (!estaVacia()) {
        std::cout << "Contenido de la cola: ";
        int i = frente;
        while (i != final) {
            std::cout << array[i] << " ";
            i = (i + 1) % capacidad;
        }
        std::cout << "\n";
    } else {
        std::cout << "La cola esta vacía.\n";
    }
```

```
}
```

main.cpp

```
#include <iostream>
#include "ColasCirculares.cpp"

int main() {
    int tamano;
    std::cout << "Ingrese el tamaño de la cola circular: ";
    std::cin >> tamano;

    CircularQueue colaCircular(tamano);

    int opcion;
    do {
        std::cout << "\n*** Menu de Operaciones ***\n";
        std::cout << "1. Encolar\n";
        std::cout << "2. Desencolar\n";
        std::cout << "3. Mostrar Contenido\n";
        std::cout << "0. Salir\n";
        std::cout << "Ingrese su opcion: ";
        std::cin >> opcion;

        switch (opcion) {
            case 1:
                int elemento;
                std::cout << "Ingrese el elemento a encolar: ";
                std::cin >> elemento;
                colaCircular.encolar(elemento);
                break;

            case 2:
                colaCircular.desencolar();
                break;

            case 3:
                colaCircular.mostrarContenido();
                break;

            case 0:
                std::cout << "Saliendo del programa.\n";
                break;

            default:
                std::cout << "Opción no válida. Inténtelo de
nuevo.\n";
                break;
        }

    } while (opcion != 0);
```

```
    return 0;
}
```

17.7. Conclusiones

- Las Colas Tradicionales Siguen el principio FIFO (Primero en entrar, primero en salir) y son útiles en situaciones donde se requiere un orden estricto de procesamiento.
- Las Colas de Prioridad: Permiten gestionar elementos según su prioridad, facilitando la atención de los elementos más importantes antes que los menos importantes.
- Las Colas Dobles: Proporcionan flexibilidad al permitir operaciones de inserción y eliminación tanto al frente como al final, siendo ideales para escenarios donde se necesita acceso eficiente a ambos extremos.
- Las Colas Circulares Presentan una estructura cíclica que permite un uso eficiente del espacio y son ideales para aplicaciones que requieren operaciones continuas de encolar y desencolar, como en sistemas embebidos o situaciones con ciclos de operación repetitivos.

18. REFERENCIAS

1. Abstraction and Specification in Program Development, Barbara Liskov and John Guttag.
2. Data Structures and Algorithms in Java, Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser.
3. Drozdek, Adam. "Data Structures and Algorithms in C++." 5th Edition, Cengage Learning, 2019.https://itlectures.ro/wpcontent/uploads/2016/04/AdamDrozdek DataStructures_and_Algorithms_in_C_4Ed.pdf
4. Hernandez, A. (15 de marzo de 2017). Análisis de algoritmos. GoConqr. https://www.goconqr.com/p/14581124/note_page/742718
5. Bhadaniya, S. (2023, May 29). *Recursion in C++: Types, Examples & Advantages*. FavTutor. https://favtutor.com/blogs/recursion-cpp
6. Ceballos, F. (2020) C/C++ Curso de programación. 5ª. Edición.
7. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press. https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf
8. Drozdek, Adam. "Data Structures and Algorithms in C++." 5th Edition, Cengage Learning, 2019. https://itlectures.ro/wpcontent/uploads/2016/04/AdamDrozdek DataStructures_and_Algorithms_in_C_4Ed.pdf

9. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2019). Algorithms Unlocked: The Basic Tools for Computer Programming (4th ed.). MIT Press. Disponible en: https://dahlan.unimal.ac.id/files/ebooks/2013%20Algorithms_Unlocked.pdf
10. Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
11. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2019). "Data Structures and Algorithms in Python". John Wiley & Sons.
12. Lafore, R. (2019). "Data Structures and Algorithms in Java". Sams Publishing.
13. Weiss, M. A. (2020). "Data Structures and Algorithm Analysis in Java". Pearson.
14. Kleinberg, J., & Tardos, É. (2020). "Algorithm Design". Pearson.