



UNIVERSIDAD NACIONAL DEL ALTIPLANO



Facultad de Ingeniería Estadística e Informática

Códigos que estructuran

Alumno: Ruth Karina Apaza Solis

Curso: Estructura de Datos

Docente: Fred Torres Cruz

Índice

- Índice
- Introducción
- 1. ¿Qué es programación?
 - 1.1 ¿Qué es un lenguaje de programación?
 - 1.2 ¿Qué es la estructura de datos?
 - 1.3 ¿Para qué sirven las estructuras de datos?
- 2. Programación en c++
 - 2.1 Mi primer programa
- 3. Operadores
 - 3.1 ¿Para que sirven los operadores?
 - 3.2 Operadores aritméticos:
 - 3.3 Operadores de comparación o relacionales:
- 4. Estructura de control
 - 4.1 ¿Que son las estructuras de control?
 - 4.1.1 Estructuras Condicionales
 - 4.1.2 Estructuras Repetitivas (Bucles)
- 5. Funciones
- 6. Arreglos
- 7. Operadores & y *.
- 8. Operador ->
- 9. Listas enlazadas
- 10. Listas Enlazadas Dobles y Circulares
 - 10.1 Lista Dblemente Enlazada
 - 10.2 Listas circulares
 - 10.2.1. Lista Circular Simple
 - 10.2.3. Lista Circular Doble
 - 10.3. Recorrido de las Listas
- 11. Pilas (Staks)

- [12. Colas \(Queue\)](#)
- [13. Recursividad](#)
- [14. Árboles](#)
 - [14.1 Árbol Binario Simple](#)
 - [14.2 Árbol Binario de Búsqueda \(BST\)](#)
 - [14.3 Árboles Balanceados](#)
 - [14.4 Árboles B y B+](#)
 - [14.5 Árbol Heap \(Montículo\)](#)
 - [14.6 Árbol Rojo-Negro](#)
- [Conclusión](#)

Introducción

Este libro presenta una síntesis ordenada de los principales contenidos abordados durante el curso de programación en C++ y estructuras de datos. Su objetivo es consolidar los conocimientos adquiridos, facilitar su comprensión y servir como material de consulta para futuros estudios.

Se inicia con una introducción a la programación, el lenguaje C++ y los conceptos básicos sobre estructuras de datos. Luego, se desarrollan temas fundamentales como operadores, estructuras de control, funciones y arrays. Posteriormente, se abordan estructuras dinámicas como pilas, colas y listas enlazadas, así como el uso de punteros y operadores específicos del lenguaje. Finalmente, se incluye el concepto de recursión, una técnica clave en la resolución de problemas computacionales.

Cada capítulo combina teoría y ejemplos prácticos, orientando al lector hacia una comprensión integral y aplicada del lenguaje C++.

1. ¿Qué es programación?

La programación informática es el arte del proceso por el cual se limpia, codifica, traza y protege el código fuente de programas computacionales, en otras palabras, es indicarle a la computadora lo que tiene que hacer.

La programación informática es una de las habilidades esenciales que aprendes cuando estudias informática.

Detrás de todos los programas informáticos que conocemos y usamos de manera cotidiana para facilitarnos diversas actividades de nuestro día con día, existe todo un proceso para poderlos crear. Este proceso es conocido como programación, conozcamos un poco más sobre lo que conlleva este proceso.

Por medio de la programación se establecen los pasos a seguir para la creación del código fuente de los diversos programas informáticos.

Este código le indicara al programa informático que tiene que hacer y como realizarlo.

La programación se guía por una serie de normas y un conjunto de órdenes, instrucciones y expresiones que tienden a ser semejantes a una lengua natural acotada. Por lo cual recibe el nombre de lenguaje de programación. Y así como en los idiomas también en la informática existen diversos lenguajes de programación.

Hablando de forma más técnica, la programación se realiza mediante el uso de algoritmos, que se podrían explicar cómo reglas o instrucciones que deben seguirse para resolver el problema y lograr el objetivo.

Algunas de ellas pueden agruparse y de ese modo recibir un nombre para tener la facilidad de ser invocadas con facilidad tantas veces como sea necesario.

 "La programación es cómo haces que las computadoras resuelvan problemas."

1.1 ¿Qué es un lenguaje de programación?

Un **lenguaje de programación** es un conjunto de reglas y símbolos que permiten a los seres humanos escribir instrucciones que una computadora puede entender y ejecutar. Es el medio de comunicación entre el programador y la máquina.

Así como los humanos usamos distintos idiomas para comunicarnos, las computadoras también entienden distintos lenguajes. Cada lenguaje tiene su propia **sintaxis** (forma de escribir) y **semántica** (significado de las instrucciones).

Los lenguajes de programación se utilizan para:

- Crear aplicaciones y programas.
- Controlar el hardware de una computadora.
- Automatizar tareas repetitivas.
- Manipular datos y resolver problemas lógicos.

Algunos ejemplos comunes de lenguajes de programación son:

- **C y C++**
- **Python**
- **Java**
- **JavaScript**
- **Ruby**
- **Go**

1.2 ¿Qué es la estructura de datos?

Una estructura de datos es una forma organizada y eficiente de almacenar, gestionar y acceder a la información dentro de un programa. Su propósito principal es facilitar el procesamiento de grandes cantidades de datos, permitiendo realizar operaciones como inserción, eliminación, búsqueda y ordenamiento de manera óptima.

Las estructuras de datos pueden clasificarse en primitivas (como enteros, caracteres y booleanos) y no primitivas (como arrays, listas, pilas, colas, árboles y grafos). Cada una de ellas responde a diferentes necesidades y se selecciona en función del tipo de problema a resolver.

El estudio de las estructuras de datos es fundamental en la programación, ya que permite diseñar algoritmos más eficientes y comprender cómo se gestiona la memoria y el rendimiento en una aplicación.

1.3 ¿Para qué sirven las estructuras de datos?

Las estructuras de datos cumplen un rol esencial en el desarrollo de programas eficientes y funcionales. Su principal utilidad radica en la organización lógica y estructurada de la información, lo cual permite optimizar el uso de los recursos del sistema y mejorar el rendimiento de los algoritmos.

A través de las estructuras de datos, es posible realizar operaciones complejas como el almacenamiento dinámico, la gestión de grandes volúmenes de información, el acceso rápido a elementos específicos, así como la implementación de técnicas de búsqueda, ordenamiento y recorrido. Además, muchas soluciones informáticas dependen directamente de la correcta elección y manipulación de estas estructuras. Ya sean las más utilizadas comúnmente -como las variables, arrays, conjuntos o clases- o las diseñadas para un propósito específico -árboles, grafos, tablas, etc.-, una estructura de datos nos permite trabajar en un algo nivel de abstracción almacenando información para luego acceder a ella, modificarla y manipularla.

Las estructuras de datos desempeñan un papel fundamental en el desarrollo de programas eficientes y funcionales. Su principal objetivo es organizar la información de manera lógica y estructurada, lo que permite optimizar el uso de los recursos del sistema, como la memoria y el tiempo de procesamiento, y mejorar significativamente el rendimiento de los algoritmos implementados.

Mediante el uso adecuado de estructuras de datos, es posible llevar a cabo operaciones complejas como el almacenamiento dinámico de información, la gestión y manipulación de grandes volúmenes de datos, el acceso rápido y directo a elementos específicos, así como la implementación de técnicas eficientes de búsqueda, ordenamiento y recorrido. La correcta selección y aplicación de estas estructuras es crucial para garantizar que las soluciones informáticas sean efectivas y escalables.

Las estructuras de datos pueden clasificarse en aquellas más comunes y generales, como las variables, arrays, conjuntos y clases, así como en aquellas diseñadas para necesidades específicas, tales como árboles, grafos, tablas hash, entre otras. Estas últimas permiten modelar relaciones y conexiones complejas entre datos, ampliando las capacidades del programa para resolver problemas avanzados.

En esencia, una estructura de datos proporciona un nivel de abstracción que facilita la organización, almacenamiento, acceso, modificación y manipulación de la información. Esta abstracción no solo simplifica el diseño de algoritmos, sino que también contribuye a mantener la claridad y modularidad del código, aspectos fundamentales en el desarrollo de software profesional.

2. Programación en C++

La programación en C++ es el proceso de escribir instrucciones utilizando el lenguaje de programación C++ para que una computadora realice tareas específicas. C++ es un lenguaje de propósito general, compilado y multiparadigma, lo que significa que permite programar de diferentes maneras, incluyendo programación estructurada, orientada a objetos y, en algunos casos, programación funcional.

Fue creado por Bjarne Stroustrup a principios de la década de 1980 como una extensión del lenguaje C, incorporando características adicionales como clases, objetos, herencia y manejo de excepciones.

2.1 Mi primer programa

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

```
int main() {
    cout << "Hola, mundo!" << endl;
    return 0;
}
```

Explicación

1. `#include <iostream>`

¿Qué hace?

Le dice al programa que use una **biblioteca llamada iostream**, que permite mostrar mensajes en la pantalla o recibir datos del teclado.

Ejemplo real: es como decirle a tu programa: *"Usa las herramientas necesarias para poder imprimir mensajes."*

2. `using namespace std;`

¿Qué hace?

Le indica al programa que puede usar funciones como `cout` sin tener que escribir `std::cout`.

Traducción fácil: le dice a C++ que use un "espacio de nombres" donde están los comandos comunes.

3. `int main() {`

¿Qué hace?

Aquí empieza el programa.

`main()` es el punto de inicio: es donde el programa comienza a ejecutarse.

Piensa en esto como el corazón del programa

4. `cout << "Hola, mundo!" << endl;`

¿Qué hace?

Imprime el mensaje `"Hola, mundo!"` en la pantalla.

```
`cout`: Significa "console output" (salida por consola).
`<<`: Se usa para enviar texto a la consola.
`endl`: Hace un salto de línea, como presionar ENTER.
```

- **Lo que verás en pantalla:**

```
Hola, mundo!
```

5. `return 0;`

¿Qué hace?

Le dice al sistema que el programa terminó **correctamente**.

`0` significa “todo bien”.

6. }

¿Qué hace?

Cierra la función `main`.

Es el final del programa.

3. Operadores

Los operadores son símbolos *(+, -, /, <, >, %,) especiales que realizan operaciones sobre uno o más operandos. Los operandos pueden ser variables, valores o expresiones. Los operadores permiten realizar diversas operaciones, como aritméticas, lógicas, relacionales, de asignación, entre otras.

3.1 ¿Para que sirven los operadores?

Los operadores en programación sirven para realizar diversas operaciones sobre datos, variables o valores. Cada tipo de operador tiene una función específica y se utiliza en diferentes contextos para lograr ciertos resultados.

Operadores más comunes:

Operadores Aritméticos: Permiten realizar operaciones matemáticas básicas, como suma, resta, multiplicación, división y módulo. Son esenciales para realizar cálculos numéricos en programas. Operadores de Comparación o Relacionales: Se utilizan para comparar valores y producir resultados booleanos (True o False). Son fundamentales para tomar decisiones en base a condiciones en un programa.

Operadores Lógicos: Permiten combinar o invertir valores booleanos. Son útiles para evaluar expresiones lógicas y tomar decisiones basadas en múltiples condiciones. Estos tres tipos de operadores son los más comunes. Aunque existen algunos más: **Operadores de Asignación:** Sirven para asignar valores a variables. También existen operadores de asignación combinados con operadores aritméticos, facilitando la actualización de variables en una única instrucción. **Operadores de Pertenencia e Identidad:** `in` se utiliza para verificar si un valor está presente en una secuencia (como una lista o cadena). `is` se utiliza para verificar si dos objetos son el mismo objeto en la memoria.

Estos operadores son fundamentales en la construcción de algoritmos y lógica de programación. Permiten a los desarrolladores realizar cálculos, comparaciones y manipulaciones de datos, lo que es esencial para la creación de programas que resuelven problemas específicos.

3.2 Operadores aritméticos:

Se utilizan para realizar operaciones matemáticas en programación. `+` (**suma**): Se utiliza para sumar dos valores.

- (**resta**): Se utiliza para restar el valor de la derecha del valor de la izquierda.

* (**multiplicación**): Se utiliza para multiplicar dos valores.

/ (**división**): Se utiliza para dividir el valor de la izquierda por el valor de la derecha.

% (**módulo**): Devuelve el resto de la división del valor de la izquierda por el valor de la derecha.

// (**división entera**): Devuelve el cociente entero de la división del valor de la izquierda por el valor de la derecha.

3.3 Operadores de comparación o relacionales:

Se utilizan para comparar dos valores y devolver un resultado booleano que indica la relación entre esos valores.

== (**igual a**): Devuelve True si los dos valores son iguales; de lo contrario, devuelve False.

!= (**diferente de**): Devuelve True si los dos valores no son iguales; de lo contrario, devuelve False.

< (**menor que**): Devuelve True si el valor de la izquierda es menor que el valor de la derecha; de lo contrario, devuelve False.

> (**mayor que**): Devuelve True si el valor de la izquierda es mayor que el valor de la derecha; de lo contrario, devuelve False.

<= (**menor o igual que**): Devuelve True si el valor de la izquierda es menor o igual al valor de la derecha; de lo contrario, devuelve False.

>= (**mayor o igual que**): Devuelve True si el valor de la izquierda es mayor o igual al valor de la derecha; de lo contrario, devuelve False.

** (**exponenciación**): Se utiliza para elevar el valor de la izquierda a la potencia del valor de la derecha.

Ejemplo

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

int main() {
    int a, b, c;
    cout << "Ingresa tres numeros: " << endl;
    cin >> a >> b >> c;

    double promedio = (a + b + c) / 3.0;

    cout << "El promedio es: " << promedio << endl;
    return 0;
}
```

Explicación

1.

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

Declaración de tres variables enteras que almacenarán los valores ingresados por el usuario.

2.

```
cout << "Ingresa tres numeros: ";
```

Salida estándar. Sigue al usuario ingresar los números.

3.

```
cin >> a >> b >> c;
```

Entrada estándar. Captura tres números consecutivos ingresados desde el teclado.

4.

```
double promedio = (a + b + c) / 3.0;
```

Se realiza la suma de las tres variables y se divide entre 3.0.

- **Uso de 3.0 en vez de 3:** fuerza una división de punto flotante para evitar truncamiento por división entera.
- El resultado se guarda en una variable de tipo double para manejar posibles decimales.

5.

```
cout << "El promedio es: " << promedio << endl;
```

Imprime el resultado del cálculo con una etiqueta descriptiva.

endl agrega un salto de línea al final de la salida.

6.

```
return 0;
```

Indica que el programa finalizó correctamente.

Esto es una convención del sistema operativo para señalar que no hubo errores.

Práctica 1: Calculadora básica Escribe un programa que lea dos números enteros y muestre el resultado de sumarlos, restarlos, multiplicarlos y dividirlos.

```
#include <iostream>
using namespace std;
int main() {
    int num1, num2;

    cout << "Ingrese el primer número: ";
    cin >> num1;
```

```
cout << "Ingrese el segundo número: ";
cin >> num2;

cout << "\nResultados:\n";
cout << "Suma: " << num1 + num2 << endl;
cout << "Resta: " << num1 - num2 << endl;
cout << "Multiplicación: " << num1 * num2 << endl;
cout << "División (entera): " << num1 / num2 << endl;

return 0;
}
```

Ejemplo de ejecución

Ingrese el primer número: 10

Ingrese el segundo número: 2

Resultados:

Suma: 12

Resta: 8

Multiplicación: 20

División: 5

Práctica 2: Par o impar Pide un número entero e indica si es par o impar.

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

int main() {
    int numero;

    cout << "Ingrese un número: ";
    cin >> numero;

    if (numero % 2 == 0) {
        cout << "El número es par." << endl;
    } else {
        cout << "El número es impar." << endl;
    }

    return 0;
}
```

Ejemplo de ejecución

Ingrese un número: 8

El número es par.

Práctica 3: Comparación de edades

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

int main() {
    int edad1, edad2;

    cout << "Ingrese la primera edad: ";
    cin >> edad1;

    cout << "Ingrese la segunda edad: ";
    cin >> edad2;

    if (edad1 > edad2) {
        cout << "La primera edad es mayor que la segunda." << endl;
    } else if (edad1 < edad2) {
        cout << "La segunda edad es mayor que la primera." << endl;
    } else {
        cout << "Ambas edades son iguales." << endl;
    }

    return 0;
}
```

Ejemplo de ejecución

Ingrese la primera edad: 19

Ingrese la segunda edad: 25

La segunda edad es mayor que la primera.

4. Estructura de control

En C++, existen seis estructuras de control fundamentales que permiten controlar el flujo de ejecución de un programa. Estas se dividen en **estructuras condicionales** y **estructuras repetitivas (bucles)**. Conocerlas es esencial para crear programas eficientes y dinámicos.

4.1 ¿Qué son las estructuras de control?

Son herramientas que permiten decidir **qué instrucciones se ejecutan, cuándo y cómo**, basándose en condiciones. Se recomienda dominar el uso de operadores antes de estudiar este tema, ya que son fundamentales para establecer condiciones lógicas.

4.1.1 Estructuras Condicionales

Permiten ejecutar diferentes bloques de código dependiendo del cumplimiento de una condición lógica.

1. `if`

Es la estructura más básica. Evalúa una condición, y si se cumple, ejecuta el bloque correspondiente.

2. `if...else`

Agrega una alternativa para ejecutar un bloque diferente cuando la condición no se cumple.

3. `if...else if...else`

Permite evaluar múltiples condiciones de forma secuencial. Solo se ejecuta el primer bloque cuya condición sea verdadera.

4. `switch...case`

Evalúa una sola variable contra diferentes valores posibles. Es útil para manejar múltiples opciones y mejora la legibilidad del código al evitar repeticiones innecesarias.

4.1.2 Estructuras Repetitivas (Bucles)

Permiten ejecutar un bloque de código de manera repetitiva, ya sea mientras se cumpla una condición o durante un número determinado de veces.

1. `while`

Ejecuta un bloque de instrucciones mientras una condición sea verdadera. La condición se evalúa antes de cada iteración.

2. `do...while`

Ejecuta el bloque al menos una vez, ya que la condición se evalúa después de ejecutar el código. Ideal cuando se necesita que la acción ocurra una vez antes de verificar.

3. `for`

Es un bucle con tres componentes definidos: inicialización, condición y actualización. Es útil cuando se conoce de antemano cuántas veces debe repetirse un proceso.

Ejemplo

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

int main() {
    int nota;
    cout << "Ingresa la nota: ";
    cin >> nota;

    if (nota > 10) {
        cout << "Aprobado";
    } else {
        cout << "Desaprobado";
    }

    return 0;
}
```

Explicación

1.

```
#include <iostream>
```

Permite usar entrada y salida (cin, cout) en C++.

2.

```
using namespace std;
```

Evita tener que escribir std:: antes de cin y cout. 3.

```
int main()
```

Función principal donde empieza la ejecución del programa. 4.

```
int nota;
```

Se declara una variable entera para guardar la nota del estudiante.

5. Entrada del usuario:

```
cout << "Ingresa la nota: ";
cin >> nota;
```

Muestra un mensaje y espera que el usuario escriba un número.

6. Condición if-else:

```
if (nota > 10) {
    cout << "Aprobado";
} else {
    cout << "Desaprobado";
}
```

Si la nota es mayor que 10, el programa muestra "Aprobado".

Si la nota es 10 o menor, muestra "Desaprobado".

7.

```
return 0;
```

Termina el programa correctamente.

Práctica 1: Mayor de dos números Pide dos números al usuario y muestra cuál es mayor, o si ambos son iguales.

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

int main() {
    int num1, num2;

    cout << "Ingrese el primer número: ";
    cin >> num1;

    cout << "Ingrese el segundo número: ";
    cin >> num2;

    if (num1 > num2) {
        cout << "El primer número es mayor." << endl;
    } else if (num2 > num1) {
        cout << "El segundo número es mayor." << endl;
    } else {
        cout << "Ambos números son iguales." << endl;
    }

    return 0;
}
```

Ejemplo de ejecución

Ingrese el primer número: 12

Ingrese el segundo número: 8

El primer número es mayor.

Práctica 2: Calificación con mensaje Pide una nota (de 0 a 20) e imprime un mensaje según el siguiente criterio:

- 18 a 20: Excelente
- 14 a 17: Bueno
- 11 a 13: Regular
- 0 a 10: Desaprobado

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

int main() {
    int nota;
```

```
cout << "Ingrese la nota (0 a 20): ";
cin >> nota;

if (nota >= 18 && nota <= 20) {
    cout << "Excelente" << endl;
} else if (nota >= 14 && nota <= 17) {
    cout << "Bueno" << endl;
} else if (nota >= 11 && nota <= 13) {
    cout << "Regular" << endl;
} else if (nota >= 0 && nota <= 10) {
    cout << "Desaprobado" << endl;
} else {
    cout << "Nota fuera de rango." << endl;
}

return 0;
}
```

Ejemplo de ejecución

Ingrese la nota (0 a 20): 15

Bueno

Práctica 3: Contar hasta N Pide al usuario un número entero positivo **N** y muestra los números del 1 hasta **N**.

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

int main() {
    int N;
    int contador = 1;

    cout << "Ingrese un número entero positivo: ";
    cin >> N;

    while (contador <= N) {
        cout << contador << " ";
        contador++;
    }

    cout << endl;
    return 0;
}
```

Ejemplo de ejecución

Ingrese un número entero positivo: 5

1 2 3 4 5

5. Funciones

una función es un bloque de código independiente que realiza una tarea específica. Puedes pensar en una función como una "mini-máquina" dentro de tu programa: tú le das una entrada (si necesita), ella hace algo con esa entrada, y luego te da una salida (si aplica). Las funciones se utilizan para organizar el código de manera más clara, estructurada y reutilizable. Son una parte fundamental de cualquier lenguaje de programación moderno, incluyendo C++, porque permiten dividir un problema complejo en partes más pequeñas y manejables.

¿Para qué sirven las funciones?

1. Organizar el código En lugar de escribir todo en una sola gran secuencia, puedes dividir tu programa en varias funciones que cada una haga una parte del trabajo. Esto hace que tu código sea más limpio, más legible y más ordenado.
2. Evitar la repetición (reutilizar código) Si necesitas hacer la misma operación varias veces (como sumar dos números o mostrar un saludo), puedes escribir una función una sola vez y llamarla todas las veces que la necesites. Así no repites código innecesario.
3. Dividir problemas grandes Cuando tienes un problema grande, como hacer una calculadora o un sistema de notas, lo mejor es dividir el problema en pasos pequeños. Cada uno de esos pasos se puede convertir en una función diferente.

Por ejemplo:

Una función que lea datos.

Otra que los procese.

Otra que muestre resultados.

Esto se llama "modularidad", y ayuda a trabajar paso a paso.

Palabras reservadas

`| return |` Devuelve un valor desde la función al lugar donde fue llamada
`| | int |` Indica que la función devuelve un número entero
`| | float |` Indica que la función devuelve un número decimal
`| | double |` Similar a `float`, pero con más precisión
`| | char |` Indica que devuelve un carácter
`| | void |` Indica que la función **no devuelve** ningún valor
`| | main() |` Es la función **principal** de todo programa en C++ |

Ejemplo

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

float area(float pi, float radio) {
    return pi * radio * radio;
}

int main() {
    float pi = 3.14;
    float radio;
    float resultado;
```

```
cout << "Ingrese el radio del circulo: ";
cin >> radio;

resultado = area(pi, radio);

cout << "El area del circulo es: " << resultado << endl;

return 0;
}
```

Explicación Este programa en C++ permite calcular el **área de un círculo** usando una **función** que recibe el valor de **pi** y el **radio** como parámetros.

1. Función area

```
float area(float pi, float radio) {
    return pi * radio * radio;
}
```

-Esta función se llama **area**. -Recibe dos parámetros: el valor de **pi** y el **radio** del círculo. -Devuelve un número decimal (tipo **float**). -Usa la fórmula del área del círculo: $[A = \pi \cdot r^2]$ 2. **Función principal main()**

```
int main() {
    float pi = 3.14;
    float radio;
    float resultado;

    cout << "Ingrese el radio del circulo: ";
    cin >> radio;

    resultado = area(pi, radio);

    cout << "El area del circulo es: " << resultado << endl;

    return 0;
}
```

Se declara **pi** con valor 3.14.

Se pide al usuario el radio del círculo.

Se llama a la función **area()** pasando **pi** y **radio**.

Se muestra el resultado con **cout**.

Ejemplo de ejecución

Ingrese el radio del circulo: 5

El area del circulo es: 78.5

Práctica 1: Calcular la potencia de un número (sin usar pow)

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

int potencia(int base, int exponente) {
    int resultado = 1;
    for (int i = 0; i < exponente; i++) {
        resultado *= base;
    }
    return resultado;
}

int main() {
    int base, exponente;

    cout << "Ingrese la base: ";
    cin >> base;

    cout << "Ingrese el exponente: ";
    cin >> exponente;

    int resultado = potencia(base, exponente);

    cout << "El resultado de " << base << "^" << exponente << " es: " << resultado
    << endl;

    return 0;
}
```

Ejemplo de ejecución

Ingrese la base: 2

Ingrese el exponente: 3

El resultado de 2^3 es: 8

Práctica 2: Sumar los dígitos de un número

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

// Función que suma los dígitos de un número
int sumaDigitos(int numero) {
    int suma = 0;
    while (numero != 0) {
        suma += numero % 10;    // Extrae el último dígito
        numero /= 10;           // Elimina el último dígito
    }
}
```

```
        return suma;
    }

int main() {
    int numero;

    cout << "Ingrese un número: ";
    cin >> numero;

    int resultado = sumaDigitos(numero);

    cout << "La suma de los dígitos es: " << resultado << endl;

    return 0;
}
```

Ejemplo de ejecución

Ingrese un número: 1234

La suma de los dígitos es: 10

6. Arreglos

¿Qué es un arreglo? Un arreglo (también llamado vector o array) es una estructura de datos fundamental en programación que permite almacenar y organizar un conjunto de valores del mismo tipo bajo un solo nombre. Estos valores están organizados en una secuencia lineal, y cada valor puede ser accedido mediante un índice o posición numérica.

Características principales de los arreglos: Homogeneidad: Todos los elementos del arreglo deben ser del mismo tipo (por ejemplo, todos enteros, todos caracteres, todos flotantes, etc.).

Acceso por índice: Cada elemento dentro del arreglo está numerado consecutivamente, comenzando en la posición 0, lo que permite acceder directamente a cualquier valor usando su índice.

Tamaño fijo: En lenguajes como C++ y C, el tamaño del arreglo debe definirse al declararlo y no puede cambiar durante la ejecución del programa.

Conceptos claves

Concepto	Ejemplo	Explicación
Declarar arreglo	int a[10];	Arreglo de 10 enteros
Inicializar arreglo	int b[3] = {5,10,15};	Arreglo con valores iniciales
Asignar valor	a[0] = 7;	Asigna 7 al primer elemento
Acceder a valor	int x = b[2];	Guarda 15 (tercer elemento) en x
Índices válidos	0 a tamaño-1	Ejemplo: para a[10], índices 0-9

Ejemplo

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

int main() {
    // Declaración de un arreglo de 5 enteros
    int numeros[5];

    // Asignación de valores a cada posición
    numeros[0] = 10;
    numeros[1] = 20;
    numeros[2] = 30;
    numeros[3] = 40;
    numeros[4] = 50;

    // Acceso y muestra de los valores del arreglo
    for(int i = 0; i < 5; i++) {
        cout << "Elemento en posicion " << i << ": " << numeros[i] << endl;
    }

    return 0;
}
```

Explicación

1. `int numeros[5];` Aquí se declara un arreglo de tipo entero (`int`) llamado `numeros`, con espacio para 5 elementos. Es importante recordar que en C++, los arreglos comienzan desde el índice 0. Por lo tanto, `numeros[0]` es el primer elemento y `numeros[4]` el último.

2. **Asignación de valores**

```
numeros[0] = 10;
numeros[1] = 20;
numeros[2] = 30;
numeros[3] = 40;
numeros[4] = 50;
```

En estas líneas, se asigna manualmente un valor a cada una de las posiciones del arreglo. Cada línea toma una posición específica e introduce un número. 3. **Recorrido con un bucle for**

```
for(int i = 0; i < 5; i++) {
    cout << "Elemento en posicion " << i << ": " << numeros[i] << endl;
}
```

Se utiliza un bucle `for` para recorrer todas las posiciones del arreglo, desde `i = 0` hasta `i = 4`. En cada iteración, se imprime el índice actual (`i`) y el valor almacenado en `numeros[i]`.

Ejecución `numeros[0] = 10; numeros[1] = 20; numeros[2] = 30; numeros[3] = 40; numeros[4] = 50;`

Práctica 1: Invertir los elementos Crear un programa que lea 8 números enteros en un arreglo y luego los muestre en orden inverso (del último al primero).

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

int main() {
    int numeros[8];

    // Lectura de datos
    cout << "Ingrese 8 numeros enteros:" << endl;
    for (int i = 0; i < 8; i++) {
        cin >> numeros[i];
    }

    // Mostrar en orden inverso
    cout << "Los numeros en orden inverso son:" << endl;
    for (int i = 7; i >= 0; i--) {
        cout << numeros[i] << " ";
    }

    cout << endl;
    return 0;
}
```

Ejemplo de ejecución

Ingrese 8 numeros enteros:

10 20 30 40 50 60 70 80

Los numeros en orden inverso son:

80 70 60 50 40 30 20 10

Práctica 2: Suma y producto escalar de dos arreglos Escribe un programa que lea dos arreglos de 5 enteros, calcule un tercer arreglo con la suma elemento a elemento y muestre el producto escalar de ambos arreglos.

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

int main() {
    const int n = 5;
    int a[n], b[n];
    int suma[n];

    cout << "Ingrese los elementos del vector A y B:\n";
    for (int i = 0; i < n; i++) {
        cout << "a[" << i << "]: ";
        cin >> a[i];
        cout << "b[" << i << "]: ";
        cin >> b[i];
    }

    for (int i = 0; i < n; i++) {
        suma[i] = a[i] * b[i];
    }

    cout << "El producto escalar es: " << suma[0];
    for (int i = 1; i < n; i++) {
        cout << ", " << suma[i];
    }
}
```

```
}

int productoEscalar = 0;

cout << "\nSuma de elementos (a[i] + b[i]):\n";
for (int i = 0; i < n; i++) {
    suma[i] = a[i] + b[i];
    productoEscalar += a[i] * b[i];
    cout << "suma[" << i << "] = " << suma[i] << endl;
}

cout << "\nProducto escalar: " << productoEscalar << endl;

return 0;
}
```

Ejemplo de ejecución

Ingrese los elementos del vector A y B: a[0]: 1

b[0]: 2

a[1]: 3

b[1]: 4

a[2]: 5

b[2]: 6

a[3]: 7

b[3]: 8

a[4]: 9

b[4]: 10

Suma de elementos (a[i] + b[i]): suma[0] = 3 suma[1] = 7 suma[2] = 11 suma[3] = 15 suma[4] = 19

Producto escalar: 130

7. Operadores & y *.

En el lenguaje de programación C++, los operadores **&** y ***** son fundamentales para trabajar con **direcciones de memoria** y **punteros**, herramientas clave para el control preciso de los datos y su almacenamiento en la memoria del computador.

6.1. Operador & – Dirección de memoria

El operador **&** se conoce como **operador de dirección**. Su función principal es obtener la **dirección de memoria** de una variable.

Cada vez que se declara una variable, el sistema operativo le asigna una ubicación específica en la memoria. El operador **&** permite acceder a esa ubicación.

Puntos clave:

- Permite trabajar con la posición física donde está almacenada la variable.

- Es utilizado para asignar direcciones a punteros.
- Es el primer paso para manipular datos de forma indirecta mediante punteros.
- Se aplica solamente a **variables**, no a literales o expresiones directas.

6.2. Operador * – Puntero y desreferenciación

El operador * tiene dos usos principales y muy distintos en C++, dependiendo del contexto:

1. Declaración de punteros

Cuando se usa en una declaración de variable, el * indica que esa variable es un **puntero**, es decir, una variable que almacena una **dirección de memoria** en lugar de un valor directo.

2. Desreferenciación

Cuando se usa sobre un puntero ya declarado, el operador * permite **acceder al valor almacenado** en la dirección de memoria que contiene el puntero.

Este proceso se llama **desreferenciación**.

Puntos clave:

Permite leer o modificar datos que están almacenados en otra ubicación de memoria. Es indispensable para la manipulación dinámica de datos y estructuras como arreglos, cadenas y memoria dinámica (`new/delete`). El mal uso del operador * puede causar errores graves como **acceder a zonas de memoria no válidas** (errores de segmentación o *segmentation faults*).

Ejemplo

```
#include <iostream>
int main() {
    int var = 10;
    int* ptr = &var;

    std::cout << "El valor de var: " << var << std::endl;
    std::cout << "La dirección de memoria de var: " << &var << std::endl;
    std::cout << "El valor de ptr: " << ptr << std::endl;
    std::cout << "El valor apuntado por ptr: " << *ptr << std::endl;

    return 0;
}
```

🔍 Explicación

1. `int var = 10;`

Se declara una variable entera llamada `var` y se le asigna el valor `10`.

El sistema operativo reserva una posición de memoria para almacenarla.

2. `int* ptr = &var;`

Se declara un puntero llamado `ptr`, que es de tipo `int*` (puntero a entero). Este puntero almacena la **dirección de memoria** donde se encuentra `var`. El operador `&var` obtiene esa dirección.

3. `std::cout << "El valor de var: " << var << std::endl;`

Se imprime el valor almacenado en `var`, que es `10`.

4. `std::cout << "La dirección de memoria de var: " << &var << std::endl;`

Se muestra la **dirección de memoria** en la que está almacenada la variable `var`. El operador `&` permite acceder a esa dirección física.

5. `std::cout << "El valor de ptr: " << ptr << std::endl;`

Se imprime el contenido del puntero `ptr`.

Como `ptr` almacena la dirección de `var`, este valor coincidirá con `&var`.

6. `std::cout << "El valor apuntado por ptr: " << *ptr << std::endl;`

Aquí se utiliza el operador `*` para **desreferenciar** el puntero `ptr`.

Esto significa que accedemos al **valor almacenado** en la dirección que contiene `ptr`, es decir, el valor de `var`.

Ejemplo de ejecución

El valor de var: 10 La dirección de memoria de var: 0x61ff08 El valor de ptr: 0x61ff08 El valor apuntado por ptr:
10

Práctica 1: Usar un puntero para acceder al valor de una variable

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

int main() {
    int edad = 25;
    int* ptr = &edad; // ptr almacena la dirección de edad

    cout << "Direccion guardada en el puntero: " << ptr << endl;
    cout << "Valor al que apunta el puntero: " << *ptr << endl;

    return 0;
}
```

Ejemplo de ejecución

Direccion guardada en el puntero: 0x61fef8

Valor al que apunta el puntero: 25

Práctica 2: Arreglo y puntero básico

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

int main() {
    int numeros[3] = {5, 10, 15};
    int* ptr = numeros; // ptr apunta al primer elemento del arreglo

    for (int i = 0; i < 3; i++) {
        cout << "Elemento " << i << ": " << *(ptr + i) << endl;
    }

    return 0;
}
```

Ejemplo de ejecución

Elemento 0: 5

Elemento 1: 10

Elemento 2: 15

Práctica 3: Intercambio de valores y uso de punteros para modificar variables

Escribe un programa que reciba dos números enteros, intercambie sus valores usando punteros, y luego muestre los valores intercambiados.

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

void intercambiar(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x, y;

    cout << "Ingrese el primer numero: ";
    cin >> x;
    cout << "Ingrese el segundo numero: ";
    cin >> y;

    intercambiar(&x, &y);

    cout << "Despues del intercambio:\n";
    cout << "Primer numero: " << x << endl;
    cout << "Segundo numero: " << y << endl;

    return 0;
}
```

Ejemplo de ejecución

Ingrese el primer numero: 15

Ingrese el segundo numero: 30

Despues del intercambio:

Primer numero: 30

Segundo numero: 15

8. Operador ->

Es una herramienta fundamental cuando se trabaja con **punteros a objetos**. Su función principal es permitir el acceso directo a los **miembros** (atributos o métodos) de un objeto a través de un puntero.

Este operador es una forma abreviada y mucho más cómoda para manipular objetos cuando se utilizan punteros, evitando la necesidad de desreferenciar manualmente el puntero.

Cuando tienes un puntero que apunta a un objeto, el operador `->` te permite acceder directamente a sus miembros sin usar el operador `*` para desreferenciar el puntero antes.

En términos prácticos, el operador `->` combina dos acciones en una sola:

1. Desreferenciar el puntero:

Equivalente a usar el operador `*` para obtener el objeto al que apunta el puntero.

2. Acceder al miembro del objeto:

Equivalente a usar el operador `.` cuando trabajas con un objeto normal (no puntero).

Se utiliza exclusivamente con punteros a objetos o estructuras. No puede usarse con variables que no sean punteros.

Simplifica la sintaxis al evitar escribir `(*puntero).miembro`. Esto mejora la legibilidad del código y reduce errores.

Permite acceder tanto a atributos (variables miembro) como a métodos (funciones miembro) del objeto al que apunta el puntero.

Es fundamental en la programación orientada a objetos (POO) cuando se manejan punteros a instancias de clases. Por ejemplo, cuando se crean objetos dinámicamente en memoria usando `new`, normalmente se usan punteros, y el operador `->` es la manera estándar de acceder a sus miembros.

¿Qué debes saber? Se utiliza exclusivamente con punteros a objetos o estructuras. No puede usarse con variables que no sean punteros.

Simplifica la sintaxis al evitar escribir `(*puntero).miembro`. Esto mejora la legibilidad del código y reduce errores.

Permite acceder tanto a atributos (variables miembro) como a métodos (funciones miembro) del objeto al que apunta el puntero.

Es fundamental en la programación orientada a objetos (POO) cuando se manejan punteros a instancias de clases. Por ejemplo, cuando se crean objetos dinámicamente en memoria usando `new`, normalmente se usan punteros, y el operador `->` es la manera estándar de acceder a sus miembros

Ejemplo

```
#include <iostream>
struct Persona {
    std::string nombre;
    int edad;
    void saludar() {
        std::cout << "Hola, mi nombre es "
        << nombre << " y tengo "
        << edad << " a~nos."
        << std::endl;
    }
};
int main() {
    Persona* ptrPersona = new Persona();
    ptrPersona->nombre = "Juan";
    ptrPersona->edad = 30;
    std::cout << "Nombre: " << ptrPersona->nombre << std::endl;
    std::cout << "Edad: " << ptrPersona->edad << std::endl;
    ptrPersona->saludar();
    delete ptrPersona;
    return 0;
}
```

Explicación 1.

```
struct Persona {
    std::string nombre;
    int edad;
    void saludar() {
        std::cout << "Hola, mi nombre es "
        << nombre << " y tengo "
        << edad << " a~nos."
        << std::endl;
    }
};
```

Se define una estructura llamada `Persona`, que actúa como una clase simple. Contiene dos miembros de datos: `nombre` (una cadena de texto) `edad` (un número entero). Además, tiene una función miembro llamada `saludar()`, que muestra un mensaje con los datos de la persona.

2.

```
int main() {
    Persona* ptrPersona = new Persona();
```

Se declara un puntero a un objeto `Persona`. Se reserva memoria dinámicamente con `new Persona()`, lo que crea un objeto `Persona` en el heap. `ptrPersona` apunta a ese objeto.

3.

```
ptrPersona->nombre = "Juan";
ptrPersona->edad = 30;
```

Se usan los operadores `->` para acceder a los miembros del objeto apuntado por el puntero `ptrPersona`. Se asignan valores al `nombre` y `edad`.

4.

```
std::cout << "Nombre: " << ptrPersona->nombre << std::endl;
std::cout << "Edad: " << ptrPersona->edad << std::endl;
```

Se imprimen los valores asignados, accediendo a los miembros del objeto a través del puntero usando `->`.

5.

```
ptrPersona->saludar();
```

Se llama a la función `saludar()` del objeto apuntado, usando también el operador `->`.

6.

```
delete ptrPersona;
```

Se libera la memoria asignada con `new` para evitar fugas de memoria (uso correcto de `delete`).

Práctica 1

Define una estructura llamada `Mascota` que contenga dos campos: `nombre` (tipo `string`) y `edad` (tipo `int`). En la función `main`, crea un **puntero** a un objeto `Mascota`, asigna valores a sus miembros utilizando el operador `->` y muestra la información por consola. Luego, libera la memoria asignada.

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

```
struct Mascota {
    string nombre;
    int edad;
};

int main() {
    // Crear un puntero a una estructura Mascota
    Mascota* ptrMascota = new Mascota();

    // Asignar valores usando el operador ->
    ptrMascota->nombre = "Luna";
    ptrMascota->edad = 4;

    // Mostrar los datos
    cout << "Nombre de la mascota: " << ptrMascota->nombre << endl;
    cout << "Edad de la mascota: " << ptrMascota->edad << " años" << endl;

    // Liberar memoria
    delete ptrMascota;

    return 0;
}
```

Ejemplo de ejecución

Nombre de la mascota: Luna

Edad de la mascota: 4 años

Práctica 2 Definir una estructura llamada **Empleado** con dos miembros: **nombre** (tipo **string**) y **sueldo** (tipo **float**). Incluir una función miembro **mostrarDatos()** que imprima ambos valores. En el programa principal, crear un objeto dinámico de tipo **Empleado**, asignar valores usando el operador **->**, invocar el método **mostrarDatos()** usando también **->** y liberar la memoria con **delete**.

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

struct Empleado {
    string nombre;
    float sueldo;

    void mostrarDatos() {
        cout << "Nombre del empleado: " << nombre << endl;
        cout << "Sueldo: $" << sueldo << endl;
    }
};

int main() {
    // Crear un puntero a un objeto Empleado
    Empleado* ptrEmpleado = new Empleado();

    // Asignar valores a través del puntero
```

```
ptrEmpleado->nombre = "Ana Torres";
ptrEmpleado->sueldo = 3500.75;

// Llamar a la función miembro
ptrEmpleado->mostrarDatos();

// Liberar memoria
delete ptrEmpleado;

return 0;
}
```

Ejecución

Nombre del empleado: Ana Torres

Sueldo: \$3500.75

9. Listas enlazadas

Una **lista enlazada** es una estructura de datos fundamental que consiste en una colección de elementos llamados **nodos**, donde cada nodo está conectado con el siguiente a través de un apuntador (o enlace).

¿Qué es un nodo? Un nodo es una pequeña estructura que contiene dos partes principales:

- **Datos:** La información que queremos almacenar (por ejemplo, un número, un texto, u otro tipo de dato).
- **Apuntador o enlace:** Un puntero que indica dónde está el siguiente nodo en la lista.

Cómo funcionan las listas enlazadas

- La lista se inicia con un apuntador al primer nodo llamado habitualmente **cabeza** o **head**.
- Cada nodo "apunta" al siguiente nodo de la lista mediante su enlace.
- Para acceder a un nodo en particular, se parte desde el primer nodo y se sigue el enlace hasta el siguiente, y así sucesivamente.

¿Cómo sabemos que la lista terminó?

El último nodo de la lista tiene un apuntador especial que no apunta a ningún otro nodo. Este apuntador se define con un valor especial llamado **NULL** (o **nullptr** en C++ moderno), que indica que no hay más nodos después.

Almacenamiento dinámico Una característica muy importante de las listas enlazadas es que:

- Los nodos se crean **dinámicamente**, es decir, solo cuando son necesarios.
- Esto permite que la lista pueda crecer o reducirse durante la ejecución del programa.
- Por lo tanto, la lista enlazada es muy flexible y eficiente en el uso de memoria, a diferencia de los arreglos, donde el tamaño debe definirse al inicio y la memoria se asigna de forma fija.

Tipos de datos en los nodos Cada nodo puede almacenar datos de cualquier tipo, incluyendo tipos básicos (como enteros o cadenas) o incluso estructuras más complejas. Esto hace a las listas enlazadas muy versátiles para representar diferentes tipos de información.

Ventajas:

- Una lista enlazada es apropiada cuando no es predecible de inmediato el número de elementos de datos a representarse en la estructura.
- Son dinámicas, por lo que conforme sea necesario, la longitud de la lista puede aumentar o disminuir, mientras que los arreglos que almacenan - las listas de datos no son dinámicos, la memoria del arreglo es asignada en tiempo de compilación.
- Las lista enlazadas sólo se llenan cuando el sistema no tiene suficiente memoria para satisfacer las solicitudes de asignación dinámica de almacenamiento.

Ejemplo

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

// Definición del nodo
struct Nodo {
    int valor;
    Nodo* siguiente;
};

// Función para agregar un nodo al final
void agregarNodo(Nodo*& cabeza, int valor) {
    Nodo* nuevoNodo = new Nodo();
    nuevoNodo->valor = valor;
    nuevoNodo->siguiente = nullptr;

    if (cabeza == nullptr) {
        cabeza = nuevoNodo;
    } else {
        Nodo* temp = cabeza;
        while (temp->siguiente != nullptr) {
            temp = temp->siguiente;
        }
        temp->siguiente = nuevoNodo;
    }
}

// Función para imprimir la lista
void imprimirLista(Nodo* cabeza) {
    Nodo* temp = cabeza;
    while (temp != nullptr) {
        cout << temp->valor << "-> ";
        temp = temp->siguiente;
    }
    cout << "nullptr" << endl;
}

// Función para eliminar toda la lista y liberar memoria
void eliminarLista(Nodo*& cabeza) {
    Nodo* temp;
    while (cabeza != nullptr) {
        temp = cabeza;
        cabeza = cabeza->siguiente;
        delete temp;
    }
}
```

```
        temp = cabeza;
        cabeza = cabeza->siguiente;
        delete temp;
    }
}

// Función principal
int main() {
    Nodo* lista = nullptr;

    agregarNodo(lista, 10);
    agregarNodo(lista, 20);
    agregarNodo(lista, 30);

    cout << "Lista enlazada: ";
    imprimirLista(lista);

    eliminarLista(lista);

    return 0;
}
```

Explicación

1. Estructura Nodo

```
struct Nodo {
    int valor;
    Nodo* siguiente;
};
```

Se define una estructura llamada **Nodo** que representa un nodo de la lista. Cada nodo tiene:

- Un campo **valor** (almacena un número entero).
- Un apuntador **siguiente** que apunta al siguiente nodo en la lista (o **nullptr** si es el último nodo).

2. Función **agregarnodo**

```
void agregarNodo(Nodo*& cabeza, int valor)
```

Esta función agrega un nuevo nodo al final de la lista. **cabeza** es una referencia al puntero que apunta al inicio de la lista (**Nodo*&**). El nuevo nodo se crea dinámicamente con **new**. **Lógica interna:**

- Se crea un nuevo nodo y se le asigna el valor recibido.
- Si la lista está vacía (**cabeza == nullptr**), el nuevo nodo se convierte en el primer nodo.
- Si ya hay nodos, se recorre la lista hasta el último nodo y se enlaza el nuevo nodo al final.

3. Función **imprimirlista**

```
void imprimirLista(Nodo* cabeza)
```

Esta función recorre la lista e imprime cada valor seguido de `->`, hasta llegar al final (`nullptr`).

```
10-> 20-> 30-> nullptr
```

4. Función `eliminarLista`

```
void eliminarLista(Nodo*& cabeza)
```

Libera la memoria de todos los nodos. Recorre la lista nodo por nodo:

- Guarda el nodo actual en un puntero temporal.
- Avanza al siguiente nodo.
- Elimina el nodo temporal con `delete`. Al final, la lista queda vacía y `cabeza` apunta a `nullptr`.

5. Función `main()`

```
int main() {
    Nodo* lista = nullptr;
    agregarNodo(lista, 10);
    agregarNodo(lista, 20);
    agregarNodo(lista, 30);

    std::cout << "Lista enlazada: ";
    imprimirLista(lista);

    eliminarLista(lista);
    return 0;
}
```

- Se crea un puntero `lista` que inicialmente apunta a `nullptr`.
- Se agregan tres nodos con valores 10, 20 y 30.
- Se imprime la lista completa.
- Finalmente, se elimina la lista y se libera la memoria.

Resultado del programa

Lista enlazada: 10-> 20-> 30-> nullptr

Práctica 1: Contar elementos de una lista enlazada Desarrolla un programa que cree una lista enlazada simple e inserte los valores 3, 8, 12 y 20. Luego, el programa debe contar cuántos nodos hay en la lista e imprimir el resultado.

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

struct Nodo {
    int valor;
    Nodo* siguiente;
};

void agregarNodo(Nodo*& cabeza, int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->valor = valor;
    nuevo->siguiente = nullptr;

    if (cabeza == nullptr) {
        cabeza = nuevo;
    } else {
        Nodo* temp = cabeza;
        while (temp->siguiente != nullptr) {
            temp = temp->siguiente;
        }
        temp->siguiente = nuevo;
    }
}

int contarNodos(Nodo* cabeza) {
    int contador = 0;
    while (cabeza != nullptr) {
        contador++;
        cabeza = cabeza->siguiente;
    }
    return contador;
}

void imprimirLista(Nodo* cabeza) {
    Nodo* temp = cabeza;
    while (temp != nullptr) {
        cout << temp->valor << "-> ";
        temp = temp->siguiente;
    }
    cout << "nullptr" << endl;
}

int main() {
    Nodo* lista = nullptr;
    agregarNodo(lista, 3);
    agregarNodo(lista, 8);
    agregarNodo(lista, 12);
    agregarNodo(lista, 20);

    cout << "Lista enlazada: ";
    imprimirLista(lista);

    cout << "Cantidad de nodos: " << contarNodos(lista) << endl;
}
```

```
    return 0;
}
```

Ejecución

Lista enlazada: 3-> 8-> 12-> 20-> nullptr

Cantidad de nodos: 4

Práctica 2: eliminar un nodo por valor en una lista enlazada Crea un programa que construya una lista enlazada simple insertando los valores 4, 7, 10, 7, 15. Luego, el usuario debe ingresar un valor que desea eliminar. El programa debe eliminar solo la primera ocurrencia de ese valor en la lista y mostrar el resultado final.

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

struct Nodo {
    int valor;
    Nodo* siguiente;
};

void agregarNodo(Nodo*& cabeza, int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->valor = valor;
    nuevo->siguiente = nullptr;

    if (cabeza == nullptr) {
        cabeza = nuevo;
    } else {
        Nodo* temp = cabeza;
        while (temp->siguiente != nullptr) {
            temp = temp->siguiente;
        }
        temp->siguiente = nuevo;
    }
}

void eliminarPorValor(Nodo*& cabeza, int valor) {
    if (cabeza == nullptr) return;

    if (cabeza->valor == valor) {
        Nodo* temp = cabeza;
        cabeza = cabeza->siguiente;
        delete temp;
        return;
    }

    Nodo* actual = cabeza;
    while (actual->siguiente != nullptr && actual->siguiente->valor != valor) {
        actual = actual->siguiente;
    }
```

```
}

if (actual->siguiente != nullptr) {
    Nodo* temp = actual->siguiente;
    actual->siguiente = temp->siguiente;
    delete temp;
}
}

void imprimirLista(Nodo* cabeza) {
    Nodo* temp = cabeza;
    while (temp != nullptr) {
        cout << temp->valor << "-> ";
        temp = temp->siguiente;
    }
    cout << "nullptr" << endl;
}

int main() {
    Nodo* lista = nullptr;
    agregarNodo(lista, 4);
    agregarNodo(lista, 7);
    agregarNodo(lista, 10);
    agregarNodo(lista, 7);
    agregarNodo(lista, 15);

    cout << "Lista original: ";
    imprimirLista(lista);

    int valorEliminar;
    cout << "Ingrese valor a eliminar: ";
    cin >> valorEliminar;

    eliminarPorValor(lista, valorEliminar);

    cout << "Lista modificada: ";
    imprimirLista(lista);

    return 0;
}
```

Ejecución

Ejemplo de entrada

Ingrese valor a eliminar: 7

Salida

Lista original: 4-> 7-> 10-> 7-> 15-> nullptr

Lista modificada: 4-> 10-> 7-> 15-> nullptr

Práctica 3: Insertar elementos al inicio de una lista enlazada Desarrolla un programa en C++ que permita insertar los valores enteros 45, 60 y 75 al inicio de una lista enlazada simple. Luego, muestra en pantalla cómo

queda la lista enlazada.

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

struct Nodo {
    int valor;
    Nodo* siguiente;
};

void insertarInicio(Nodo*& cabeza, int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->valor = valor;
    nuevo->siguiente = cabeza;
    cabeza = nuevo;
}

void imprimirLista(Nodo* cabeza) {
    Nodo* temp = cabeza;
    while (temp != nullptr) {
        cout << temp->valor << "-> ";
        temp = temp->siguiente;
    }
    cout << "nullptr" << endl;
}

int main() {
    Nodo* lista = nullptr;

    insertarInicio(lista, 45);
    insertarInicio(lista, 60);
    insertarInicio(lista, 75);

    cout << "Lista enlazada: ";
    imprimirLista(lista);

    return 0;
}
```

Ejecución

Lista enlazada: 75-> 60-> 45-> nullptr

10. Listas Enlazadas Dobles y Circulares

10.1 Lista Dblemente Enlazada

Una lista doblemente enlazada es una lista lineal en la que cada nodo tiene dos enlaces, uno al nodo siguiente, y otro al anterior. Las listas doblemente enlazadas no necesitan un nodo especial para acceder a ellas, pueden recorrerse en ambos sentidos a partir de cualquier nodo, esto es porque a partir de cualquier

nodo, siempre es posible alcanzar cualquier nodo de la lista, hasta que se llega a uno de los extremos. El nodo típico es el mismo que para construir las listas que hemos visto, salvo que tienen otro puntero al nodo anterior. **Ventajas:**

- Navegación bidireccional
- Eliminación más eficiente
- Inserción flexible

Desventajas:

- Mayor uso de memoria
- Mayor complejidad en implementación

Operaciones básicas Estructura

```
struct nodo {  
    int dato;  
    struct nodo *siguiente;  
    struct nodo *anterior;  
};
```

Insertar al inicio

```
void insertarInicio ( struct Nodo ** cabeza , int valor ) {  
    struct Nodo * nuevo = ( struct Nodo *) malloc ( sizeof (   
        struct Nodo ) ) ;  
    nuevo - > dato = valor ;  
    nuevo - > siguiente = (* cabeza ) ;  
    nuevo - > anterior = NULL ;  
  
    if ((* cabeza ) != NULL )  
        (* cabeza ) -> anterior = nuevo ;  
  
    (* cabeza ) = nuevo ;  
}
```

¿Qué hace esta función?

- Crea un nuevo nodo con el valor dado.
- Lo inserta al principio de la lista doblemente enlazada.
- Si ya hay nodos en la lista, el antiguo primer nodo actualiza su puntero `anterior` para apuntar al nuevo nodo.
- Finalmente, el puntero `cabeza` apunta ahora al nuevo nodo.

Importante:

- Se usa `malloc` porque este es código estilo C.
- Se usa `struct Nodo** cabeza` para poder modificar el puntero original que apunta al inicio de la lista.

Insertar al Final

```
void insertarFinal ( struct Nodo ** cabeza , int valor ) {  
    struct Nodo * nuevo = ( struct Nodo *) malloc ( sizeof ( struct Nodo ) ) ;  
    struct Nodo * ultimo = * cabeza ;  
  
    nuevo - > dato = valor ;  
    nuevo - > siguiente = NULL ;  
  
    if (* cabeza == NULL ) {  
        nuevo - > anterior = NULL ;  
        * cabeza = nuevo ;  
        return ;  
    }  
  
    while ( ultimo - > siguiente != NULL )  
        ultimo = ultimo - > siguiente ;  
  
    ultimo - > siguiente = nuevo ;  
    nuevo - > anterior = ultimo ;  
}
```

¿Qué hace esta función?

- Crea un nodo nuevo con el valor dado.
- Si la lista está vacía, ese nodo se convierte en el primer nodo.
- Si ya hay nodos, recorre la lista hasta el final.
- Conecta el nuevo nodo al último:
 - El puntero `anterior` del nuevo nodo apunta al último nodo.
 - El `siguiente` del último nodo ahora apunta al nuevo.

Importante:

- Se recorre con `while` hasta encontrar el nodo que tenga `siguiente == NULL`.

Eliminar nodo

```
void eliminarNodo ( struct Nodo ** cabeza , struct Nodo *  
nodoEliminar ) {  
    if (* cabeza == NULL || nodoEliminar == NULL )  
        return ;  
  
    if (* cabeza == nodoEliminar )  
        * cabeza = nodoEliminar - > siguiente ;  
  
    if ( nodoEliminar - > siguiente != NULL )  
        nodoEliminar - > siguiente - > anterior = nodoEliminar  
        -> anterior ;
```

```
if ( nodoEliminar -> anterior != NULL )
nodoEliminar -> anterior -> siguiente = nodoEliminar
-> siguiente ;

free ( nodoEliminar ) ;
}
```

¿Qué hace esta función?

- Elimina un nodo cualquiera de la lista doblemente enlazada.
- Verifica si la lista está vacía o si el nodo a eliminar es `NULL`.
- Si el nodo a eliminar es la `cabeza`, actualiza el puntero `cabeza` al siguiente nodo.
- Ajusta los enlaces del nodo `anterior` y del nodo `siguiente` para saltarse el nodo eliminado.
- Finalmente, libera la memoria del nodo.

Importante:

- No importa si el nodo está al inicio, en medio o al final.
- El uso de `free()` es necesario porque el nodo fue creado con `malloc()`.

10.2 Listas circulares

Una lista circular es una lista lineal en la que el último nodo apunta al primero. Las listas circulares evitan excepciones en las operaciones que se realicen sobre ellas. No existen casos especiales, cada nodo siempre tiene uno anterior y uno siguiente. En algunas listas circulares se añade un nodo especial de cabecera, de ese modo se evita la única excepción posible, la de que la lista esté vacía. El nodo típico es el mismo que para construir listas abiertas. **Ventajas:**

- Útil para aplicaciones cíclicas
- No hay punteros `NULL`
- Acceso continuo a todos los nodos

Desventajas:

- Riesgo de bucles infinitos
- Detección del final más compleja

10.2.1. Lista Circular Simple

Cada nodo contiene un dato y un puntero al siguiente nodo. El último nodo no apunta a `NULL`, sino que apunta al primer nodo, formando un ciclo cerrado. Solo se puede avanzar hacia adelante; no hay punteros hacia atrás (no es doblemente enlazada).

Estructura

```
struct nodo {
    int dato;
    struct nodo *siguiente;
};
```

- Define un nodo con un dato de tipo entero.
- Contiene un puntero llamado `siguiente` que apunta al siguiente nodo en la lista.
- En una lista circular simple, el último nodo apunta al primero, formando un ciclo.

Insertar en Lista Circular

```
void insertarCircular(struct NodoCircular** cabeza, int valor) {
    struct NodoCircular* nuevo = (struct NodoCircular*)
        malloc(sizeof(struct NodoCircular));
    nuevo->dato = valor;

    if (*cabeza == NULL) {
        *cabeza = nuevo;
        nuevo->siguiente = nuevo; // Apunta a sí mismo
        return;
    }

    struct NodoCircular* temp = *cabeza;
    while (temp->siguiente != *cabeza)
        temp = temp->siguiente;

    temp->siguiente = nuevo;
    nuevo->siguiente = *cabeza;
}
```

- Crea un nuevo nodo con el valor dado.
- Si la lista está vacía, el nuevo nodo apunta a sí mismo y se asigna como cabeza.
- Si no, recorre la lista hasta encontrar el último nodo (que apunta a la cabeza).
- Inserta el nuevo nodo después del último, y lo conecta para que apunte a la cabeza, manteniendo el ciclo.

10.2.3. Lista Circular Doble

Cada nodo contiene un dato y dos punteros: Uno hacia el siguiente nodo (enlace hacia adelante). Otro hacia el anterior nodo (enlace hacia atrás). El último nodo apunta al primero, y el primero apunta al último, formando un ciclo cerrado. Puedes recorrer la lista tanto hacia adelante como hacia atrás sin fin.

Estructura

```
struct NodoCircularDoble {
    int dato;
    struct NodoCircularDoble* siguiente;
    struct NodoCircularDoble* anterior;
};
```

- Nodo con un dato entero.
- Dos punteros: siguiente para avanzar al siguiente nodo, anterior para retroceder al nodo previo.

- En lista circular doble, el primer nodo y el último están conectados bidireccionalmente formando un ciclo.
-

Insertar en Lista Circular Doble

```
void insertarCircularDoble(struct NodoCircularDoble** cabeza, int valor) {
    struct NodoCircularDoble* nuevo = (struct NodoCircularDoble*)
    malloc(sizeof(struct NodoCircularDoble));
    nuevo->dato = valor;

    if (*cabeza == NULL) {
        *cabeza = nuevo;
        nuevo->siguiente = nuevo;
        nuevo->anterior = nuevo;
        return;
    }

    struct NodoCircularDoble* ultimo = (*cabeza)->anterior;

    nuevo->siguiente = *cabeza;
    (*cabeza)->anterior = nuevo;
    nuevo->anterior = ultimo;
    ultimo->siguiente = nuevo;
}
```

- Crea un nodo nuevo con el valor dado.
- Si la lista está vacía, el nuevo nodo se enlaza a sí mismo en ambas direcciones y se establece como cabeza.
- Si no, encuentra el último nodo (usando el puntero anterior de la cabeza).
- Inserta el nuevo nodo entre el último y la cabeza, actualizando ambos punteros para mantener el ciclo doble.

10.3. Recorrido de las Listas

Recorrer Lista Doble

```
void recorrerDoble(struct Nodo* nodo) {
    printf("Hacia adelante: ");
    while (nodo != NULL) {
        printf("%d ", nodo->dato);
        nodo = nodo->siguiente;
    }
    printf("\n");
}
```

- Recorre la lista simple desde el nodo dado hasta que el siguiente nodo es NULL.
- Imprime el dato de cada nodo durante el recorrido.

• Recorrer Lista Circular

```
void recorrerCircular(struct NodoCircular* cabeza) {
    if (cabeza == NULL) return;

    struct NodoCircular* temp = cabeza;
    do {
        printf("%d ", temp->dato);
        temp = temp->siguiente;
    } while (temp != cabeza);
    printf("\n");
}
```

- Si la lista está vacía, termina inmediatamente.
- Usa un ciclo do-while para recorrer desde la cabeza, avanzando nodo por nodo.
- Imprime el dato de cada nodo hasta volver al nodo inicial (cabeza), asegurando recorrer todo el ciclo.

Ejemplo

```
#include <stdio.h>
#include <stdlib.h>

// Asumiendo que las estructuras y funciones:
// insertarInicio, insertarFinal, insertarCircular, recorrerDoble y
// recorrerCircular
// ya están definidas en otro lugar.

int main() {
    struct Nodo* listaDoble = NULL;
    struct NodoCircular* listaCircular = NULL;

    // Insertar en lista doble
    insertarInicio(&listaDoble, 10);
    insertarInicio(&listaDoble, 20);
    insertarFinal(&listaDoble, 30);

    // Insertar en lista circular
    insertarCircular(&listaCircular, 1);
    insertarCircular(&listaCircular, 2);
    insertarCircular(&listaCircular, 3);

    printf("Lista Doble: ");
    recorrerDoble(listaDoble);

    printf("Lista Circular: ");
    recorrerCircular(listaCircular);
```

```
    return 0;  
}
```

Explicación 1.

```
#include <stdio.h>  
#include <stdlib.h>
```

Estas líneas incluyen las librerías estándar de entrada/salida (`stdio.h`) y manejo de memoria dinámica (`stdlib.h`). 2.

```
struct Nodo* listaDoble = NULL;  
struct NodoCircular* listaCircular = NULL;
```

Declara dos punteros a estructuras, `listaDoble` y `listaCircular`. Ambos punteros se inicializan en `NULL`, indicando que inicialmente las listas están vacías. `Nodo` es la estructura para la lista doblemente enlazada. `NodoCircular` es la estructura para la lista circular simple. 3.

```
// Insertar en lista doble  
insertarInicio(&listaDoble, 10);  
insertarInicio(&listaDoble, 20);  
insertarFinal(&listaDoble, 30);
```

Estas líneas llaman funciones para agregar nodos a la lista doblemente enlazada.

`insertarInicio(&listaDoble, 10)` inserta un nodo con valor `10` al inicio de la lista.

`insertarInicio(&listaDoble, 20)` inserta un nodo con valor `20` al inicio, que quedará antes del nodo con `10`. `insertarFinal(&listaDoble, 30)` inserta un nodo con valor `30` al final de la lista. 4.

```
// Insertar en lista circular  
insertarCircular(&listaCircular, 1);  
insertarCircular(&listaCircular, 2);  
insertarCircular(&listaCircular, 3);
```

Aquí se agregan nodos a la **lista circular simple**. Cada llamada a `insertarCircular` agrega un nuevo nodo con el valor especificado. Los nodos se van enlazando para que el último apunte al primero, formando un **ciclo**. 5.

```
printf("Lista Doble: ");  
recorrerDoble(listaDoble);
```

Imprime el texto "Lista Doble: ". Luego, la función `recorrerDoble` recorre la lista doble desde el primer nodo hasta el último, imprimiendo los valores almacenados. 6.

```
printf("Lista Circular: ");
recorrerCircular(listaCircular);
```

Imprime el texto "Lista Circular: ". La función `recorrerCircular` recorre la lista circular empezando por el nodo apuntado por `listaCircular` e imprime todos los valores hasta regresar al nodo inicial, mostrando así la lista completa.

Ejecución

Lista Doble: Hacia adelante: 20 10 30

Lista Circular: 1 2 3

Práctica 1:

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

struct Nodo {
    int dato;
    Nodo* siguiente;
    Nodo* anterior;
};

void insertarInicio(Nodo*& cabeza, int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->dato = valor;
    nuevo->siguiente = cabeza;
    nuevo->anterior = nullptr;

    if (cabeza != nullptr)
        cabeza->anterior = nuevo;

    cabeza = nuevo;
}

void recorrerDoble(Nodo* nodo) {
    while (nodo != nullptr) {
        cout << nodo->dato << " ";
        nodo = nodo->siguiente;
    }
    cout << endl;
}

int main() {
    Nodo* lista = nullptr;

    insertarInicio(lista, 30);
    insertarInicio(lista, 20);
```

```
insertarInicio(lista, 10);

cout << "Lista doble: ";
recorrerDoble(lista);

return 0;
}
```

Ejecución

Lista doble: 10 20 30

Práctica 2:

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

struct Nodo {
    int dato;
    Nodo* siguiente;
};

void insertarCircular(Nodo*& cabeza, int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->dato = valor;
    nuevo->siguiente = nullptr;

    if (cabeza == nullptr) {
        cabeza = nuevo;
        nuevo->siguiente = cabeza; // Se apunta a sí mismo
        return;
    }

    Nodo* temp = cabeza;
    while (temp->siguiente != cabeza)
        temp = temp->siguiente;

    temp->siguiente = nuevo;
    nuevo->siguiente = cabeza;
}

void recorrerCircular(Nodo* cabeza) {
    if (cabeza == nullptr) return;

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

```
int main() {
    Nodo* lista = nullptr;

    insertarCircular(lista, 5);
    insertarCircular(lista, 10);
    insertarCircular(lista, 15);

    recorrerCircular(lista);

    return 0;
}
```

Ejecución

Lista circular: 5 10 15

11. Pilas (Staks)

Una **pila** es una estructura de datos lineal que sigue el principio **LIFO** (*Last In, First Out*), es decir, el último elemento que se agrega es el primero en salir.

Se puede imaginar como una **pila de platos**: el último plato que se coloca es el primero que se retira.

Características principales

- Se inserta un elemento con la operación `push()`.
- Se elimina el último elemento insertado con la operación `pop()`.
- Se puede consultar el elemento en la cima con `top()`.
- Solo se puede acceder al elemento en la **cima** (último insertado).
- Es una estructura muy útil en procesos como:
 - Deshacer acciones
 - Recursividad
 - Evaluación de expresiones
 - Seguimiento de llamadas a funciones, entre otros.

Operaciones básicas

Operación	Descripción
<code>push()</code>	Inserta un nuevo elemento en la cima
<code>pop()</code>	Elimina el elemento en la cima
<code>top()</code>	Devuelve el valor del elemento en la cima
<code>empty()</code>	Verifica si la pila está vacía

Ejemplo

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

int main() {
    stack<int> pila;

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

    cout << "Elemento en la cima: " << pila.top() << endl;

    pila.pop();

    cout << "Nuevo elemento en la cima: " << pila.top() << endl;

    return 0;
}
```

Explicación

1. `stack<int> pila;`: Se crea una pila de enteros.
2. `pila.push(10);`: Inserta el número `10`.
3. `pila.push(20);`: Inserta el número `20` encima del `10`.
4. `pila.push(30);`: Inserta el número `30` encima del `20`.
5. `pila.top();`: Retorna el elemento en la cima (`30`).
6. `pila.pop();`: Elimina el `30`.
7. `pila.top();`: Ahora la cima es `20`.

Práctica 1

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

bool esPalindromo(string texto) {
    stack<char> pila;
    int n = texto.length();

    for (int i = 0; i < n / 2; i++) {
        pila.push(texto[i]);
    }

    int inicio = (n % 2 == 0) ? n / 2 : n / 2 + 1;

    for (int i = inicio; i < n; i++) {
        if (pila.top() != texto[i])
            return false;
        pila.pop();
    }
}
```

```
    }

    return true;
}

int main() {
    string texto;
    cout << "Ingrese una palabra: ";
    cin >> texto;

    if (esPalindromo(texto))
        cout << "Es un palindromo" << endl;
    else
        cout << "No es un palindromo" << endl;

    return 0;
}
```

Ejecución

Ingrese una palabra: radar

Es un palindromo

Práctica 2:

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

int main() {
    int numero;
    stack<int> binario;

    cout << "Ingrese un número decimal: ";
    cin >> numero;

    while (numero > 0) {
        binario.push(numero % 2);
        numero = numero / 2;
    }

    cout << "Número en binario: ";
    while (!binario.empty()) {
        cout << binario.top();
        binario.pop();
    }

    cout << endl;
    return 0;
}
```

Ejecución

Ingrese un número decimal: 13

Número en binario: 1101

12. Colas (Queue)

Una **cola** es una estructura de datos lineal que permite almacenar y procesar elementos en orden secuencial, utilizando el principio **FIFO** (*First In, First Out*), lo que significa que **el primer elemento en ingresar es el primero en salir**.

A diferencia de una **pila (LIFO)**, en una cola los elementos se **insertan al final** y se **eliminan desde el inicio**. Esta estructura es útil cuando se requiere mantener un orden de atención o procesamiento entre los datos.

Estructura de una cola

Una cola mantiene dos referencias importantes:

- **Frente (front)**: apunta al primer elemento.
- **Final (rear)**: apunta al último elemento insertado.

Los elementos se **insertan al final** y se **eliminan desde el frente**.

Operaciones fundamentales

Operación	Función
<code>push()</code>	Inserta un elemento al final de la cola
<code>pop()</code>	Elimina el elemento del frente
<code>front()</code>	Retorna el valor del primer elemento sin eliminarlo
<code>empty()</code>	Verifica si la cola está vacía

Características principales

- **Acceso limitado**: solo se puede acceder al elemento en el frente.
- **Orden estricto**: el primer elemento en entrar es el primero en salir.
- Se pueden implementar con **arrays**, **listas enlazadas** o **clases estándar como queue**.

Ejemplo

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

int main() {
    queue<int> cola;

    cola.push(10);
    cola.push(20);
```

```
cola.push(30);

cout << "Elemento al frente: " << cola.front() << endl;

cola.pop();

cout << "Nuevo frente: " << cola.front() << endl;

return 0;
}
```

Explicación

1.

```
queue<int> cola;
```

Se declara una cola de enteros llamada cola. 2.

```
cola.push(10); // Entra 10
cola.push(20); // Entra 20
cola.push(30); // Entra 30
```

Se insertan elementos al final de la cola con la función push():

Primero entra el 10 → cola = [10]

Luego el 20 → cola = [10, 20]

Luego el 30 → cola = [10, 20, 30]

3.

```
cout << "Elemento al frente: " << cola.front() << endl;
```

Se imprime el primer elemento de la cola, es decir, el que va a salir primero.

Salida: Elemento al frente: 10

4.

```
cola.pop(); // Sale 10
```

Se elimina el elemento al frente de la cola, que es el 10.

Ahora la cola queda: [20, 30]

5.

```
cout << "Nuevo frente: " << cola.front() << endl;
```

Se vuelve a mostrar el primer elemento de la cola.

Salida: Nuevo frente: 20

Práctica 1: Promedio de números.

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

int main() {
    queue<int> numeros;
    int n, valor;
    int suma = 0;

    cout << "¿Cuántos números desea ingresar? ";
    cin >> n;

    for (int i = 0; i < n; i++) {
        cout << "Ingrese un número: ";
        cin >> valor;
        numeros.push(valor);
        suma += valor;
    }

    double promedio = (double)suma / n;

    cout << "Promedio: " << promedio << endl;
    return 0;
}
```

Ejecución

¿Cuántos números desea ingresar? 3

Ingrese un número: 5

Ingrese un número: 10

Ingrese un número: 15

Promedio: 10

Práctica 2: Reordenar elementos.

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

int main() {
    queue<int> cola;
    int n, valor;
```

```
cout << "Ingrese la cantidad de elementos: ";
cin >> n;

for (int i = 0; i < n; i++) {
    cout << "Elemento " << i + 1 << ": ";
    cin >> valor;
    cola.push(valor);
}

// Mover los dos primeros al final
if (n >= 2) {
    cola.push(cola.front());
    cola.pop();
    cola.push(cola.front());
    cola.pop();
}

// Mostrar elementos
cout << "Cola final: ";
while (!cola.empty()) {
    cout << cola.front() << " ";
    cola.pop();
}

cout << endl;
return 0;
}
```

Ejecución

Ingrese la cantidad de elementos: 5

Elemento 1: 10

Elemento 2: 20

Elemento 3: 30

Elemento 4: 40

Elemento 5: 50

Cola final: 30 40 50 10 20

13. Recursividad

La recursión consiste en que una función se llame a sí misma para resolver un problema. Se utiliza cuando un problema puede dividirse en subproblemas más pequeños del mismo tipo.

Una función recursiva descompone el problema en partes más simples, resuelve esas partes y luego combina los resultados para obtener la solución final.

13.1 Estructura de una función recursiva

```
tipo nombre_funcion(parámetros) {
    if (condición_base) {
        return valor_base;
    } else {
        return llamada_recursiva; // con valores reducidos
    }
}
```

Donde:

- **condición_base**: condición que detiene la recursión.
- **valor_base**: resultado que se devuelve cuando se llega a la condición base.
- **llamada_recursiva**: llamada a la misma función con parámetros modificados.

13.2 Características importantes de la recursividad

- Utiliza la **pila de ejecución del sistema** para almacenar cada llamada pendiente.
- Cada llamada recursiva genera un **nuevo contexto de ejecución**.
- Finaliza cuando se alcanza el **caso base**.

13.3 Funcionamiento interno de la recursividad

Cada vez que una función se llama a sí misma:

- Se guarda su estado (**parámetros y variables**) en la **pila de llamadas**.
- Se ejecuta una **nueva instancia** de la función.
- Al llegar al **caso base**, se empiezan a resolver las llamadas en **orden inverso** (desde la última hacia la primera).

Este mecanismo se llama **desenrollar la recursión**.

13.4 Tipos de recursión

- **Recursión directa**: cuando una función se llama a sí misma directamente.
- **Recursión indirecta**: cuando una función llama a otra, que eventualmente la vuelve a llamar.
- **Recursión lineal**: cuando hay una sola llamada recursiva por ejecución.
- **Recursión múltiple**: cuando una función se llama a sí misma más de una vez en una misma ejecución (como en la serie de Fibonacci).
- **Recursión de cola (tail recursion)**: cuando la llamada recursiva es la **última operación** en la función (optimizable por el compilador).

Ejemplo Factorial de un número

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

int factorial(int n) {
```

```
if (n == 0) return 1;           // Caso base
else return n * factorial(n - 1); // Llamada recursiva
}

int main() {
    int numero;
    cout << "Ingrese un número: ";
    cin >> numero;
    cout << "Factorial de " << numero << " es: " << factorial(numero) << endl;
    return 0;
}
```

Explicación

1.

```
int factorial(int n) {
```

Se define una función llamada **factorial** que recibe un entero **n**. 2.

```
if (n == 0) return 1;
```

Este es el caso base de la recursión:

Si $n == 0$, retorna 1 (porque $0! = 1$).

3.

```
else return n * factorial(n - 1);
```

Esta es la llamada recursiva: Si $n > 0$, entonces el factorial se calcula como:

```
n * factorial(n - 1)
```

Por ejemplo, si $n = 3$: $\text{factorial}(3) = 3 * \text{factorial}(2)$

$\text{factorial}(2) = 2 * \text{factorial}(1)$

$\text{factorial}(1) = 1 * \text{factorial}(0)$

$\text{factorial}(0) = 1 \leftarrow \text{caso base}$

Entonces:

$\text{factorial}(1) = 1 \times 1 = 1$

$\text{factorial}(2) = 2 \times 1 = 2$

$\text{factorial}(3) = 3 \times 2 = 6$

4.

```
int main() {
    int numero;
    cout << "Ingrese un número: ";
    cin >> numero;
```

- Se declara una variable numero.
- Se solicita al usuario que escriba un número, y se guarda en numero.

5.

```
cout << "Factorial de " << numero << " es: " << factorial(numero) << endl;
```

- Se llama a la función factorial(numero) y se muestra el resultado.

Ejecución

Ingrese un número: 4

Factorial de 4 es: 24

Práctica 1: Secuencia Fibonacci

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

int fibonacci(int n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    cout << "Ingrese un valor n: ";
    cin >> n;

    cout << "Fibonacci de " << n << " es: " << fibonacci(n) << endl;
    return 0;
}
```

Ejecución:

n = 6 → Fibonacci(6) = 8

Práctica 2: Invertir un número entero

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

```
void invertir(int n) {
    if (n < 10) {
        cout << n;
    } else {
        cout << n % 10;
        invertir(n / 10);
    }
}

int main() {
    int numero;
    cout << "Ingrese un número: ";
    cin >> numero;

    cout << "Número invertido: ";
    invertir(numero);
    cout << endl;
    return 0;
}
```

Ejemplo de ejecución:

Número = 1234 → Resultado: 4321

14. Árboles

Un árbol es una estructura de datos no lineal que organiza la información en forma jerárquica. Está compuesto por nodos conectados entre sí mediante enlaces llamados ramas. Esta estructura permite representar relaciones padre-hijo entre los elementos y se utiliza ampliamente en algoritmos, bases de datos, sistemas de archivos, compiladores, entre otros.

Un árbol es un conjunto de nodos tales que:

- Existe un único nodo llamado raíz (root), que no tiene padre.
- Cada nodo puede tener cero o más nodos hijos.
- No existe ningún ciclo dentro de la estructura.
- Un nodo que no tiene hijos se llama hoja.
- Todo nodo, excepto la raíz, tiene un único parente.

Terminología básica de árboles

Término	Definición
Raíz	Primer nodo del árbol (sin parente)
Nodo	Unidad que contiene datos e información de conexión
Padre	Nodo que tiene hijos
Hijo	Nodo descendiente de otro nodo
Hermano	Nodos con el mismo parente

Término	Definición
Hoja	Nodo que no tiene hijos
Subárbol	Árbol formado por un nodo y sus descendientes
Nivel	Distancia del nodo respecto a la raíz (la raíz está en el nivel 0)
Altura	Número máximo de niveles desde la raíz hasta una hoja
Grado del nodo	Número de hijos que tiene un nodo
Grado del árbol	Grado máximo entre todos los nodos del árbol

Estructura de nodo

```
struct Nodo {
    int dato;
    Nodo* izquierdo;
    Nodo* derecho;
};
```

Cada nodo contiene:

- Un dato (entero, carácter, estructura, etc.).
- Un puntero al hijo izquierdo.
- Un puntero al hijo derecho.

Ejemplo:

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

struct Nodo {
    int dato;
    Nodo* izq;
    Nodo* der;
};

// Crear un nuevo nodo con un valor
Nodo* nuevoNodo(int valor) {
    Nodo* nodo = new Nodo();
    nodo->dato = valor;
    nodo->izq = nullptr;
    nodo->der = nullptr;
    return nodo;
}

// Insertar un valor en el árbol
Nodo* insertar(Nodo* raiz, int valor) {
    if (raiz == nullptr) {
```

```
        return nuevoNodo(valor);
    }

    if (valor < raiz->dato) {
        raiz->izq = insertar(raiz->izq, valor);
    } else if (valor > raiz->dato) {
        raiz->der = insertar(raiz->der, valor);
    }

    return raiz;
}

// Recorrido Inorden: Izquierda → Nodo → Derecha
void inorder(Nodo* raiz) {
    if (raiz != nullptr) {
        inorder(raiz->izq);
        cout << raiz->dato << " ";
        inorder(raiz->der);
    }
}

int main() {
    Nodo* raiz = nullptr;

    // Insertando elementos en el árbol
    raiz = insertar(raiz, 50);
    raiz = insertar(raiz, 30);
    raiz = insertar(raiz, 70);
    raiz = insertar(raiz, 20);
    raiz = insertar(raiz, 40);
    raiz = insertar(raiz, 60);
    raiz = insertar(raiz, 80);

    cout << "Recorrido Inorden del árbol: ";
    inorder(raiz);
    cout << endl;

    return 0;
}
```

Explicación:

1. Estructura del nodo

```
struct Nodo {
    int dato;
    Nodo* izq;
    Nodo* der;
};
```

- Cada nodo del árbol contiene:
- Un dato de tipo int.
- Un puntero al hijo izquierdo izq.
- Un puntero al hijo derecho der.

2. Crear un nuevo nodo

```
Nodo* nuevoNodo(int valor) {
    Nodo* nodo = new Nodo();
    nodo->dato = valor;
    nodo->izq = nullptr;
    nodo->der = nullptr;
    return nodo;
}
```

- Esta función reserva memoria dinámica para un nuevo nodo.
- Asigna el valor recibido (valor) al campo dato.
- Inicializa sus hijos como nullptr (vacíos).

3. Insertar valores en el árbol

```
Nodo* insertar(Nodo* raiz, int valor) {
    if (raiz == nullptr) {
        return nuevoNodo(valor);
    }

    if (valor < raiz->dato) {
        raiz->izq = insertar(raiz->izq, valor);
    } else if (valor > raiz->dato) {
        raiz->der = insertar(raiz->der, valor);
    }

    return raiz;
}
```

- Si la raíz es nula, se crea un nuevo nodo.
- Si el valor es menor que el del nodo actual, se inserta en el subárbol izquierdo.
- Si es mayor, se inserta en el subárbol derecho.
- Esta lógica asegura que se cumpla la propiedad del BST:
- Subárbol izquierdo < nodo < subárbol derecho

4. Recorrido Inorden

```
void inorder(Nodo* raiz) {
    if (raiz != nullptr) {
        inorder(raiz->izq); // Visita subárbol izquierdo
        cout << raiz->dato << " "; // Muestra el dato del nodo actual
```

```

        inorden(raiz->der);           // Visita subárbol derecho
    }
}

```

- El recorrido inorden imprime los elementos en orden ascendente.
- Se visita:
 - Primero el hijo izquierdo
 - Luego el nodo actual
 - Finalmente el hijo derecho.

5. Función main()

```

int main() {
    Nodo* raiz = nullptr;

    // Insertando elementos en el árbol
    raiz = insertar(raiz, 50);
    raiz = insertar(raiz, 30);
    raiz = insertar(raiz, 70);
    raiz = insertar(raiz, 20);
    raiz = insertar(raiz, 40);
    raiz = insertar(raiz, 60);
    raiz = insertar(raiz, 80);

    cout << "Recorrido Inorden del árbol: ";
    inorden(raiz);
    cout << endl;

    return 0;
}

```

- Se crea un puntero raiz y se inicializa con nullptr.
- Se insertan 7 valores en el árbol en este orden: 50, 30, 70, 20, 40, 60, 80.
- Luego se imprime el recorrido inorden del árbol, el cual mostrará los valores ordenados.

Ejecución

20 30 40 50 60 70 80

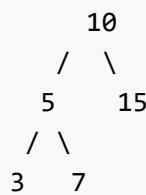
Tipos de árboles

14.1 Árbol Binario Simple

Es un árbol donde cada nodo puede tener como **máximo dos hijos**: uno izquierdo y uno derecho. Sirve como base para estructuras más avanzadas.

- No necesariamente está ordenado.
- Se utiliza en representaciones **básicas de jerarquías o estructuras**.

Práctica 1: Dado el árbol binario simple con los siguientes valores:



Mostrar los elementos del árbol usando recorrido inorden.

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

struct Nodo {
    int dato;
    Nodo* izq;
    Nodo* der;
};

Nodo* crearNodo(int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->dato = valor;
    nuevo->izq = nullptr;
    nuevo->der = nullptr;
    return nuevo;
}

void inorden(Nodo* raiz) {
    if (raiz != nullptr) {
        inorden(raiz->izq);
        cout << raiz->dato << " ";
        inorden(raiz->der);
    }
}

int main() {
    Nodo* raiz = crearNodo(10);
    raiz->izq = crearNodo(5);
    raiz->der = crearNodo(15);
    raiz->izq->izq = crearNodo(3);
    raiz->izq->der = crearNodo(7);

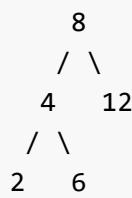
    cout << "Recorrido Inorden: ";
    inorden(raiz);
    cout << endl;

    return 0;
}
```

Ejecución

Recorrido Inorden: 3 5 7 10 15

Práctica 2: Dado el siguiente árbol binario:



Contar cuántos nodos hojas tiene el árbol.

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

struct Nodo {
    int dato;
    Nodo* izq;
    Nodo* der;
};

Nodo* crearNodo(int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->dato = valor;
    nuevo->izq = nullptr;
    nuevo->der = nullptr;
    return nuevo;
}

int contarHojas(Nodo* raiz) {
    if (raiz == nullptr) return 0;
    if (raiz->izq == nullptr && raiz->der == nullptr) return 1;
    return contarHojas(raiz->izq) + contarHojas(raiz->der);
}

int main() {
    Nodo* raiz = crearNodo(8);
    raiz->izq = crearNodo(4);
    raiz->der = crearNodo(12);
    raiz->izq->izq = crearNodo(2);
    raiz->izq->der = crearNodo(6);

    int hojas = contarHojas(raiz);
    cout << "Cantidad de nodos hoja: " << hojas << endl;

    return 0;
}
```

Ejecución

Cantidad de nodos hoja: 3

14.2 Árbol Binario de Búsqueda (BST)

Es un **árbol binario ordenado**, donde para cada nodo se cumple:

- Todos los elementos del **subárbol izquierdo** son **menores** que el nodo.
- Todos los elementos del **subárbol derecho** son **mayores** que el nodo.

Esto permite realizar búsquedas, inserciones y eliminaciones eficientes (en tiempo $O(\log n)$, si está balanceado).

Práctica 1: Construya un Árbol Binario de Búsqueda (BST) que almacene los siguientes valores: 50, 30, 70, 20, 40, 60, 80

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

struct Nodo {
    int dato;
    Nodo* izq;
    Nodo* der;
};

Nodo* crearNodo(int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->dato = valor;
    nuevo->izq = nullptr;
    nuevo->der = nullptr;
    return nuevo;
}

Nodo* insertar(Nodo* raiz, int valor) {
    if (raiz == nullptr) return crearNodo(valor);

    if (valor < raiz->dato)
        raiz->izq = insertar(raiz->izq, valor);
    else if (valor > raiz->dato)
        raiz->der = insertar(raiz->der, valor);

    return raiz;
}

bool buscar(Nodo* raiz, int valor) {
    if (raiz == nullptr) return false;
    if (raiz->dato == valor) return true;
    if (valor < raiz->dato)
        return buscar(raiz->izq, valor);
    else
        return buscar(raiz->der, valor);
}
```

```
void inorder(Nodo* raiz) {
    if (raiz != nullptr) {
        inorder(raiz->izq);
        cout << raiz->dato << " ";
        inorder(raiz->der);
    }
}

int main() {
    Nodo* raiz = nullptr;
    int valores[] = {50, 30, 70, 20, 40, 60, 80};

    for (int valor : valores)
        raiz = insertar(raiz, valor);

    cout << "Recorrido Inorden del BST: ";
    inorder(raiz);
    cout << endl;

    int buscar_valor;
    cout << "Ingrese un número a buscar: ";
    cin >> buscar_valor;

    if (buscar(raiz, buscar_valor))
        cout << "El número SÍ está en el árbol.\n";
    else
        cout << "El número NO está en el árbol.\n";

    return 0;
}
```

Ejecución

Recorrido Inorden del BST: 20 30 40 50 60 70 80

Ingrese un número a buscar: 40

El número SÍ está en el árbol.

Práctica 2: Utilizando la estructura std::set, realizar las siguientes operaciones:

1. Insertar los valores:

50, 30, 70, 20, 40, 60, 80

2. Imprimir los valores del conjunto en orden ascendente (equivalente al recorrido inorden de un árbol binario de búsqueda).

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

int main() {
    set<int> bst;
```

```

// 1. Insertar los valores
bst.insert(50);
bst.insert(30);
bst.insert(70);
bst.insert(20);
bst.insert(40);
bst.insert(60);
bst.insert(80);

// 2. Mostrar los elementos en orden ascendente
cout << "Recorrido Inorden del BST (usando std::set): ";
for (int valor : bst) {
    cout << valor << " ";
}
cout << endl;

return 0;
}

```

Ejecución

Recorrido Inorden del BST (usando std::set): 20 30 40 50 60 70 80

14.3 Árboles Balanceados

Son árboles donde la **altura de los subárboles izquierdo y derecho** de cada nodo difiere como máximo en uno.

Esto **evita que el árbol se degrade a una lista lineal**, manteniendo su eficiencia.

Ejemplo clásico: Árbol **AVL**, donde se realizan **rotaciones** para mantener el equilibrio después de cada inserción o eliminación.

Práctica 1: Crear un árbol binario desbalanceado a la izquierda y aplicar una rotación simple a la derecha para balancearlo. Mostrar el recorrido inorden antes y después de la rotación.

```

#include <iostream>
using namespace std;

struct Nodo {
    int dato;
    Nodo* izq;
    Nodo* der;
};

// Crear nodo nuevo
Nodo* crearNodo(int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->dato = valor;
    nuevo->izq = nullptr;
    nuevo->der = nullptr;
    return nuevo;
}

```

```

}

// Recorrido inorden
void inorder(Nodo* raiz) {
    if (raiz != nullptr) {
        inorder(raiz->izq);
        cout << raiz->dato << " ";
        inorder(raiz->der);
    }
}

// Rotación simple a la derecha
Nodo* rotarDerecha(Nodo* y) {
    Nodo* x = y->izq;
    Nodo* T2 = x->der;

    x->der = y;
    y->izq = T2;

    return x;
}

int main() {
    Nodo* raiz = crearNodo(30);
    raiz->izq = crearNodo(20);
    raiz->izq->izq = crearNodo(10); // Árbol desbalanceado a la izquierda

    cout << "Recorrido Inorden antes de rotar: ";
    inorder(raiz);
    cout << endl;

    // Aplicamos rotación
    raiz = rotarDerecha(raiz);

    cout << "Recorrido Inorden después de rotar: ";
    inorder(raiz);
    cout << endl;

    return 0;
}

```

Ejecución

Recorrido Inorden antes de rotar: 10 20 30

Recorrido Inorden después de rotar: 10 20 30

Práctica 2: Dado un árbol binario, verificar si está balanceado, es decir, si la diferencia de altura entre los subárboles izquierdo y derecho de cada nodo es como máximo 1.

```

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

```

```
struct Nodo {
    int dato;
    Nodo* izq;
    Nodo* der;
};

Nodo* crearNodo(int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->dato = valor;
    nuevo->izq = nullptr;
    nuevo->der = nullptr;
    return nuevo;
}

// Altura del árbol
int altura(Nodo* raiz) {
    if (raiz == nullptr) return 0;
    int altIzq = altura(raiz->izq);
    int altDer = altura(raiz->der);
    return 1 + max(altIzq, altDer);
}

// Verificar balance
bool estaBalanceado(Nodo* raiz) {
    if (raiz == nullptr) return true;

    int altIzq = altura(raiz->izq);
    int altDer = altura(raiz->der);

    int diferencia = abs(altIzq - altDer);

    return (diferencia <= 1)
        && estaBalanceado(raiz->izq)
        && estaBalanceado(raiz->der);
}

int main() {
    Nodo* raiz = crearNodo(40);
    raiz->izq = crearNodo(20);
    raiz->der = crearNodo(60);
    raiz->izq->izq = crearNodo(10);
    raiz->izq->der = crearNodo(30);

    if (estaBalanceado(raiz))
        cout << "El árbol está balanceado" << endl;
    else
        cout << "El árbol NO está balanceado" << endl;

    return 0;
}
```

Ejecución

El árbol está balanceado

14.4 Árboles B y B+

Se utilizan en **sistemas de bases de datos y archivos**.

- **Árbol B:** Árbol de búsqueda generalizado donde cada nodo puede tener **múltiples claves e hijos**.
 - Mantiene los datos ordenados.
 - Permite **búsquedas, inserciones y eliminaciones** en tiempo **logarítmico**.
- **Árbol B+:** Variante del árbol B donde:
 - Los **datos reales** solo se almacenan en las **hojas**.
 - Las hojas están **enlazadas secuencialmente**, lo que **facilita las búsquedas por rango**.

Son altamente eficientes para operaciones en **disco** o **almacenamiento externo**.

Práctica 1: Crear un nodo de un árbol B que permita almacenar hasta 3 claves enteras. Luego, ingresar manualmente 3 valores y almacenarlos en orden dentro del nodo. Finalmente, mostrar las claves ingresadas.

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

struct NodoB {
    int claves[3];
    int n; // cantidad de claves utilizadas
};

// Insertar clave en orden dentro del nodo
void insertarClave(NodoB &nodo, int valor) {
    if (nodo.n < 3) {
        nodo.claves[nodo.n] = valor;
        nodo.n++;

        // Ordenar las claves
        for (int i = 0; i < nodo.n - 1; i++) {
            for (int j = i + 1; j < nodo.n; j++) {
                if (nodo.claves[i] > nodo.claves[j]) {
                    int aux = nodo.claves[i];
                    nodo.claves[i] = nodo.claves[j];
                    nodo.claves[j] = aux;
                }
            }
        }
    }
}

void mostrarClaves(NodoB nodo) {
    cout << "Claves almacenadas: ";
}
```

```
        for (int i = 0; i < nodo.n; i++) {
            cout << nodo.claves[i] << " ";
        }
        cout << endl;
    }

int main() {
    NodoB nodo;
    nodo.n = 0;

    insertarClave(nodo, 40);
    insertarClave(nodo, 10);
    insertarClave(nodo, 20);

    mostrarClaves(nodo);

    return 0;
}
```

Ejecución

Claves almacenadas: 10 20 40

Práctica 2: Crear un nodo B que admite hasta 3 claves. Ingresar valores uno por uno e indicar si el nodo se encuentra lleno (es decir, si ya no puede almacenar más claves).

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

struct NodoB {
    int claves[3];
    int n;
};

bool estaLleno(NodoB nodo) {
    return nodo.n == 3;
}

void insertarClave(NodoB &nodo, int valor) {
    if (!estaLleno(nodo)) {
        nodo.claves[nodo.n] = valor;
        nodo.n++;
    } else {
        cout << "El nodo ya está lleno. No se puede insertar " << valor << endl;
    }
}

int main() {
    NodoB nodo;
    nodo.n = 0;

    insertarClave(nodo, 5);
```

```

insertarClave(nodo, 15);
insertarClave(nodo, 25);
insertarClave(nodo, 35); // esta no debería entrar

cout << "Estado del nodo: ";
if (estaLleno(nodo)) {
    cout << "Lleno" << endl;
} else {
    cout << "Aún hay espacio" << endl;
}

return 0;
}

```

Ejecución

El nodo ya está lleno. No se puede insertar 35

Estado del nodo: Lleno

14.5 Árbol Heap (Montículo)

Es un **árbol completo** que cumple la **propiedad de montículo**:

- En un **Max-Heap**, el valor de cada nodo es **mayor o igual** que el de sus hijos.
- En un **Min-Heap**, el valor de cada nodo es **menor o igual** que el de sus hijos.

Usos principales:

- **Algoritmos de prioridad**
- Implementación de **colas de prioridad**
- **HeapSort** (algoritmo de ordenamiento) **Práctica 1:** Crear una función que transforme un arreglo de enteros en un Max-Heap. Mostrar el arreglo antes y después de aplicar heapify.

```

#include <iostream>
using namespace std;

// Reorganiza el subárbol con raíz en i para cumplir propiedad de Max-Heap
void heapify(int A[], int n, int i) {
    int mayor = i;
    int izq = 2 * i + 1;
    int der = 2 * i + 2;

    if (izq < n && A[izq] > A[mayor]) mayor = izq;
    if (der < n && A[der] > A[mayor]) mayor = der;

    if (mayor != i) {
        int temp = A[i];
        A[i] = A[mayor];
        A[mayor] = temp;

        heapify(A, n, mayor);
    }
}

```

```

}

// Construye un Max-Heap desde un arreglo
void construirHeap(int A[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(A, n, i);
    }
}

// Muestra el arreglo
void mostrar(int A[], int n) {
    for (int i = 0; i < n; i++) {
        cout << A[i] << " ";
    }
    cout << endl;
}

int main() {
    int A[] = {20, 15, 8, 10, 5, 7, 6};
    int n = 7;

    cout << "Arreglo original: ";
    mostrar(A, n);

    construirHeap(A, n);

    cout << "Max-Heap construido: ";
    mostrar(A, n);

    return 0;
}

```

Ejecución

Arreglo original: 20 15 8 10 5 7 6

Max-Heap construido: 20 15 8 10 5 7 6

Práctica 2: Utilizar la estructura priority_queue de la biblioteca estándar de C++ para simular el comportamiento de un Max-Heap. Insertar los siguientes valores: 20, 15, 30, 5 Luego, mostrar:

1. El valor máximo almacenado (elemento en la cima del heap).
2. Todos los elementos del heap, extrayéndolos uno por uno en orden descendente.

```

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

int main() {
    priority_queue<int> heap;

    heap.push(20);
    heap.push(15);

```

```

heap.push(30);
heap.push(5);

cout << "Elemento mayor (top): " << heap.top() << endl;

cout << "Elementos del heap (de mayor a menor): ";
while (!heap.empty()) {
    cout << heap.top() << " ";
    heap.pop();
}
cout << endl;

return 0;
}

```

Ejecución Elemento mayor (top): 30

Elementos del heap (de mayor a menor): 30 20 15 5

14.6 Árbol Rojo-Negro

Es un **árbol binario de búsqueda auto-balanceado** donde cada nodo tiene un **color** (rojo o negro) y se cumplen ciertas reglas que garantizan el equilibrio del árbol.

Características clave:

- Las operaciones de **inserción, eliminación y búsqueda** se realizan en **O(log n)**.
- **Requiere menos rotaciones** que un árbol AVL.
- Muy utilizado en bibliotecas estándar, como:
 - `std::map`
 - `std::set`

Práctica 1: Definir la estructura de un nodo para un Árbol Rojo-Negro, que incluya:

- Un campo entero para almacenar el valor.
 - Un campo de color (rojo o negro).
 - Punteros a hijo izquierdo, derecho y padre.
- Luego, crear e imprimir un nodo de valor 10 de color rojo.

```

#include <iostream>
using namespace std;

enum Color { ROJO, NEGRO };

struct NodoRN {
    int dato;
    Color color;
    NodoRN* izq;
    NodoRN* der;
    NodoRN* padre;
};

```

```
NodoRN* crearNodo(int valor) {
    NodoRN* nuevo = new NodoRN();
    nuevo->dato = valor;
    nuevo->color = ROJO;
    nuevo->izq = nullptr;
    nuevo->der = nullptr;
    nuevo->padre = nullptr;
    return nuevo;
}

void mostrarNodo(NodoRN* nodo) {
    cout << "Dato: " << nodo->dato << endl;
    cout << "Color: " << (nodo->color == ROJO ? "Rojo" : "Negro") << endl;
}

int main() {
    NodoRN* raiz = crearNodo(10);
    mostrarNodo(raiz);
    return 0;
}
```

Ejecución

Dato: 10

Color: Rojo

Práctica 2: Utilizar la estructura std::set para almacenar claves enteras. Insertar los siguientes valores: 40, 10, 60, 20, 50

Luego, imprimir los elementos ordenados y verificar si el número 50 se encuentra en el conjunto.

Nota: std::set implementa internamente un árbol rojo-negro balanceado.

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

int main() {
    set<int> arbol;

    arbol.insert(40);
    arbol.insert(10);
    arbol.insert(60);
    arbol.insert(20);
    arbol.insert(50);

    cout << "Elementos en orden: ";
    for (int x : arbol) {
        cout << x << " ";
    }
    cout << endl;

    if (arbol.find(50) != arbol.end()) {
```

```
    cout << "El número 50 sí está en el árbol." << endl;
} else {
    cout << "El número 50 no está en el árbol." << endl;
}

return 0;
}
```

Ejecución

Elementos en orden: 10 20 40 50 60

El número 50 sí está en el árbol.

Conclusión

Este trabajo ha sido desarrollado con el objetivo de repasar y reforzar los principales temas del curso de estructuras de datos, partiendo desde los fundamentos básicos de la programación en C++, como operadores, funciones y arreglos, hasta llegar a estructuras más complejas y específicas como listas enlazadas, pilas, colas, árboles y montículos.

Cada capítulo ha sido elaborado de forma ordenada, incluyendo teoría explicada, ejemplos claros y ejercicios resueltos paso a paso.

Durante el desarrollo del documento he podido aplicar muchos de los conceptos vistos en clase y darles una estructura clara que también puede servir como material de repaso para mí o para otros compañeros. Me ayudó a practicar bastante la lógica, la implementación manual y a tener una visión más completa de cómo se manejan los datos en memoria.

Con este trabajo concluyo el desarrollo de los temas propuestos en el curso, cumpliendo con los objetivos planteados.

Trabajo elaborado por: Ruth Karina Apaza Solis

Presentado al curso de: Estructuras de Datos

Escuela Profesional de Estadística e Informática – 2025