

# Universidad Nacional del Altiplano

Facultad de Ingeniería Estadística e Informática

Escuela Profesional de Ingenieria Estadistica e Informatica



**Libro:**

Lenguaje de Programación en C++

**Alumno:**

Jhon Kenedy Chambi Chambi

**Docente:**

Ing. Fred Torres Cruz

**Curso:**

Estructura de Datos

**Semestre:**

Tercer Semestre B

**PUNO-PUNO 2025**

# Índice

<b>1. Introducción</b>	<b>5</b>
<b>2. Definición.-</b>	<b>7</b>
2.1. Definición Formal.- . . . . .	7
<b>3. Características Claves.-</b>	<b>7</b>
<b>4. Uso Comunes.-</b>	<b>8</b>
4.1. Desarrollo de Videojuegos.- . . . . .	8
4.2. Sistemas embebido.- . . . . .	9
4.3. Motores Gráficos.- . . . . .	9
4.4. Sistemas Operativos.- . . . . .	10
4.5. Aplicaciones de Alto Rendimiento.- . . . . .	10
<b>5. FUNDAMENTOS BÁSICOS</b>	<b>11</b>
5.1. Entrada y Salida Estándar . . . . .	11
5.2. cmath- FUNCIONES MATEMÁTICAS.- . . . . .	12
5.2.1. CÓDIGO DE UNA CALCULADORA BÁSICA(SUMA Y RESTA) .	12
5.3. cstdlib - UTILIDADES ESTÁNDAR . . . . .	13
5.4. string - MANIPULACIÓN DE CADENAS . . . . .	14
5.5. algorithm - ALGORITMOS ÚTILES . . . . .	14
5.6. vector - ARREGLOS DINÁMICOS.- . . . . .	15
5.7. map, set, unorderedmap - COLECCIONES CLAVE-CALOR Y CONJUNTOS	15
5.8. ctime - TIEMPO Y FECHAS . . . . .	16
5.9. fstream - ARCHIVOS . . . . .	16
<b>6. UNIDADES DE DESARROLLO</b>	<b>17</b>
6.1. Programas en c++ . . . . .	17
6.2. Hola Mundo . . . . .	17
6.3. Arrays . . . . .	18
6.3.1. ¿Que una array? . . . . .	18
6.3.2. Características Principales . . . . .	18
6.3.3. Un Código como ejemplo: . . . . .	18
6.4. Ventajas y desventajas de usar arrays en C++ . . . . .	19
6.5. Tipos de Arrays en C++ . . . . .	19
6.5.1. Array Estático . . . . .	19
6.5.2. Array Dinámico . . . . .	19
6.6. ¿Cuando se debe usar cada uno?- . . . . .	20
6.7. Ejercicios Prácticos . . . . .	20
6.7.1. Ejercicio 1: Suma de elementos de un array . . . . .	20
6.8. Ejercicio 2: Encontrar el mayor elemento . . . . .	21
6.9. Struct . . . . .	23
6.9.1. ¿Qué es una estructura (struct) en C++? . . . . .	23

6.9.2. Características Principales . . . . .	23
6.9.3. Ventajas y Desventajas del Struct . . . . .	24
6.9.4. Ejercicio 1: Registrar un alumno . . . . .	24
6.9.5. Ejercicio 2: Promedio de varios productos . . . . .	26
<b>7. Listas Enlazadas.-</b>	<b>29</b>
7.0.1. ¿Qué es una Lista Enlazada? . . . . .	29
7.0.2. Ventajas de Listas Enlazadas . . . . .	29
7.1. Estructura de un nodo en C++ . . . . .	29
7.1.1. Ejemplo de básico: Crear y mostrar una lista de 3 elementos . . . . .	29
7.1.2. Ejercicios de Práctica . . . . .	30
<b>8. Listas Dobles y Listas Circulares.-</b>	<b>34</b>
8.1. ¿Qué es una Lista Dblemente Circulada? . . . . .	34
8.1.1. Estructura básica en C++.- . . . . .	34
8.1.2. Aplicaciones Cómunes . . . . .	34
8.1.3. Ventajas . . . . .	34
8.1.4. Desventajas.- . . . . .	35
8.2. ¿Qué es una Lista Circular.- . . . . .	35
8.3. Ejemplo de Nodo en Lista Circular Simple.- . . . . .	35
8.4. Ejemplo de Nodo en Lista Circular Doble.- . . . . .	35
8.5. Aplicaciones . . . . .	36
8.6. Ventajas . . . . .	36
8.7. Cuidados Especiales.- . . . . .	36
8.8. Comparación entre Listas Dobles y Listas Circulares.- . . . . .	36
<b>9. Colas (Queues</b>	<b>37</b>
9.1. Definición y qué es Colas en C++ . . . . .	37
9.2. Operaciones Principales de una Cola . . . . .	37
9.3. Ventajas de las Colas . . . . .	37
9.4. Implementación en C++ . . . . .	37
9.5. Ejercicios Prácticos en C++.- . . . . .	38
9.5.1. Enunciado N° 1.- . . . . .	38
9.5.2. Enunciado N° 2 . . . . .	40
<b>10.Pilas (Stacks).-</b>	<b>43</b>
10.1. Definición.- . . . . .	43
10.2. ¿Cómo funciona una Pila?.- . . . . .	43
10.3. Operaciones Principales.- . . . . .	43
10.4. Ventajas de las Pilas.- . . . . .	43
10.5. Implementación en C++.- . . . . .	44
10.6. Aplicaciones comunes con pilas . . . . .	44
10.7. Ejercicios Prácticos . . . . .	45
10.7.1. Enunciado N° 1 . . . . .	45
10.8. Enunciado N° 2 . . . . .	46

<b>11. Recursión.-</b>	<b>48</b>
11.1. Ejemplo Básico: Factorial.- . . . . .	48
11.1.1. Código en C++ . . . . .	48
11.2. Representación Visual: Llamadas Recursivas del Factorial(3) . . . . .	49
11.3. Ventajas de la Recursión.- . . . . .	49
11.4. Desventajas.- . . . . .	49
11.5. Casos Clásicos donde se usa la Recursión.- . . . . .	49
11.6. Ejercicios Prácticos.- . . . . .	50
11.6.1. Enunciado N° 1.- . . . . .	50
11.6.2. Enunciado N° 2.- . . . . .	51
<b>12. Árboles Binarios</b>	<b>52</b>
12.1. ¿Qué es un árbol binario? . . . . .	52
12.2. Estructura básica . . . . .	52
12.3. Características principales: . . . . .	52
12.4. Código simple de árbol binario en C++ . . . . .	52
12.4.1. Orden Visual . . . . .	54
<b>13. Árboles Balanceados</b>	<b>55</b>
13.1. Definición general . . . . .	55
13.2. ¿Por qué balancear un árbol? . . . . .	55
13.3. Tipos comunes de árboles balanceados . . . . .	55
13.4. Operaciones y complejidad . . . . .	55
13.5. Rotaciones (Árbol AVL) . . . . .	56
13.6. Operaciones y complejidad . . . . .	56
13.7. Rotaciones (Árbol AVL) . . . . .	57
13.7.1. Ejemplo de inserción en un AVL (C++) . . . . .	57
13.7.2. Resultado del programa . . . . .	61
13.8. Ventajas y desventajas . . . . .	61
13.9. Aplicaciones típicas . . . . .	61
<b>14. Árboles B y B++</b>	<b>62</b>
14.1. ¿Qué es un Árbol B . . . . .	62
14.2. Características del Árbol B . . . . .	62
14.3. Estructura de un nodo B (grado t) . . . . .	62
14.3.1. Ejemplo Visual Simplificado . . . . .	62
14.4. Programa en C++ de un árbol B . . . . .	63
14.5. Operaciones Básicas . . . . .	66
14.5.1. Búsqueda . . . . .	66
14.5.2. Inserción . . . . .	66
14.5.3. Eliminación . . . . .	66
14.6. ¿Qué es un árbol B++ . . . . .	67
14.7. Diferencia entre b y B+ . . . . .	67
14.8. Estructura del Árbol B+ . . . . .	68
14.9. ¿Cómo funciona . . . . .	68

14.10 Ejemplo visual (simplificado) . . . . .	69
14.11 Características clave del árbol B . . . . .	69
14.12 Ventajas del árbol B+ . . . . .	69
14.13 Desventajas . . . . .	69
14.14 Aplicaciones del Árbol B+ . . . . .	70
14.15 Programa en C++ . . . . .	70
<b>15. Árboles Heap</b>	<b>74</b>
15.1. ¿Qué es un Árbol Heap . . . . .	74
15.2. Características de un Árbol Heap . . . . .	74
15.3. Tipos de Heap . . . . .	74
15.3.1. Max-Heap . . . . .	74
15.3.2. Min-Heap . . . . .	74
15.4. Ejemplo de un código visual de Min-Heap . . . . .	75
15.5. Representación con Arreglo . . . . .	75
15.6. Aplicaciones del Árbol Heap . . . . .	75
15.7. Ventajas . . . . .	75
15.8. Desventajas . . . . .	76
15.9. Programa en C++ . . . . .	76
15.9.1. Resultado . . . . .	78
<b>16. Árbol Rojo y Negro</b>	<b>79</b>
16.1. Propiedades fundamentales . . . . .	79
16.2. Operaciones en Árbol Rojo-Negro: . . . . .	79
16.3. Rotaciones (Balanceo) . . . . .	80
16.4. Ventajas de los Árboles Rojo-Negro . . . . .	80
16.5. Desventajas . . . . .	81
16.6. Aplicaciones Comunes . . . . .	81

# 1. Introducción

En el mundo de la programación y el desarrollo de software, las estructuras de datos cumplen un papel fundamental. Comprender cómo se organizan, almacenan y manipulan los datos permite a los programadores crear soluciones más eficientes, escalables y robustas. Este libro ha sido creado con el objetivo de brindar una guía clara y detallada sobre las principales estructuras de datos, utilizando el lenguaje de programación C++ como herramienta principal para su implementación.

A lo largo de estas páginas, se explorarán desde las estructuras más básicas, como listas, pilas y colas, hasta estructuras más complejas como árboles, grafos y tablas hash. Se incluirán explicaciones teóricas, diagramas visuales y ejemplos prácticos de código para facilitar el aprendizaje.

El enfoque de este libro es progresivo y didáctico, por lo que está diseñado tanto para estudiantes que se inician en el estudio de las estructuras de datos, como para desarrolladores que deseen reforzar sus conocimientos y mejorar sus habilidades en la programación estructurada y orientada a objetos en C++.

Espero que este material sirva como una herramienta útil en tu formación como programador o ingeniero de software. La comprensión de las estructuras de datos no solo mejora el rendimiento de tus programas, sino que también desarrolla tu capacidad para resolver problemas de manera lógica y eficiente.

**Jhon Kenedy Chambi Chambi**

Autor

© 2025 Jhon Kenedy Chambi Chambi

Todos los derechos reservados.

Ninguna parte de esta publicación puede ser reproducida, almacenada en un sistema de recuperación ni transmitida de ninguna forma ni por ningún medio, ya sea electrónico, mecánico, fotocopia, grabación u otros, sin el permiso previo y por escrito del autor.

Primera edición – 2025

Hecho en Perú

# Lenguaje de Programación en C++

## 2. Definición.-

C++ es un lenguaje de programación de propósito general que fue desarrollado por Bjarne Stroustrup a principios de los años 1980 como una extensión del lenguaje C. Su principal característica es que combina la programación de bajo nivel (cercana al hardware) con características de alto nivel, incluyendo la programación orientada a objetos.

### 2.1. Definición Formal.-

C++ es un lenguaje de programación compilado, de tipado estático, multiparadigma (soporta programación estructurada, orientada a objetos y genérica), que permite el desarrollo eficiente de software tanto a nivel del sistema como de aplicaciones complejas.



## 3. Características Claves.-

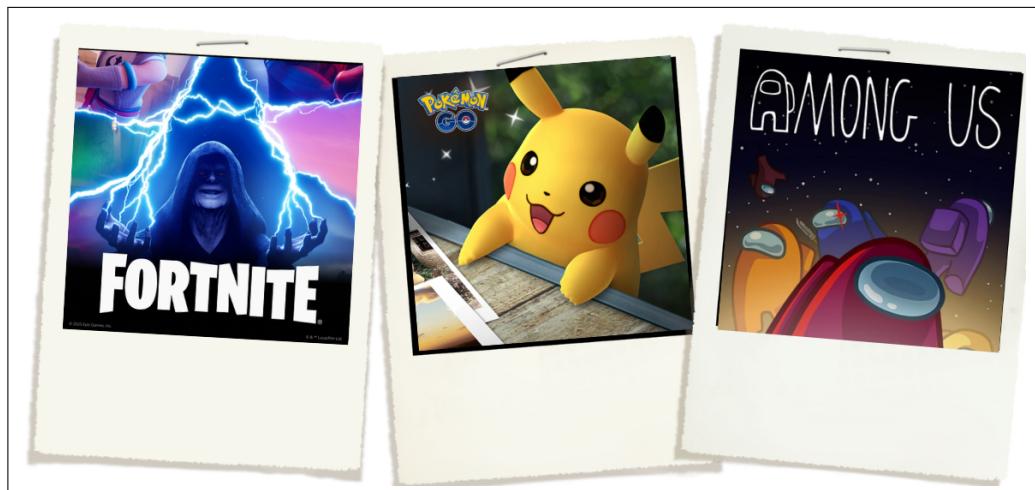
- Compilado: El código fuente se traduce a código máquina mediante un compilador, lo que permite una ejecución rápida.
- Eficiencia y rendimiento: Ofrece control directo sobre la memoria y los recursos del sistema.

- Multiparadigma: Soporta programación estructurada, orientada a objetos y programación genérica.
- Programación orientada a objetos: Incluye conceptos como clases, objetos, herencia, polimorfismo y encapsulamiento.
- Plantillas (templates): Permite escribir código genérico y reutilizable.
- Compatibilidad con C: La mayoría del código en C puede ser utilizado en C++ sin problemas.
- Control manual de memoria: Permite gestionar la memoria de forma explícita usando new y delete.
- Tipado estático: Los tipos de datos se verifican en tiempo de compilación, ayudando a prevenir errores.
- Biblioteca estándar extensa: Incluye la STL (Standard Template Library) con estructuras de datos y algoritmos comunes.

## 4. Uso Comunes.-

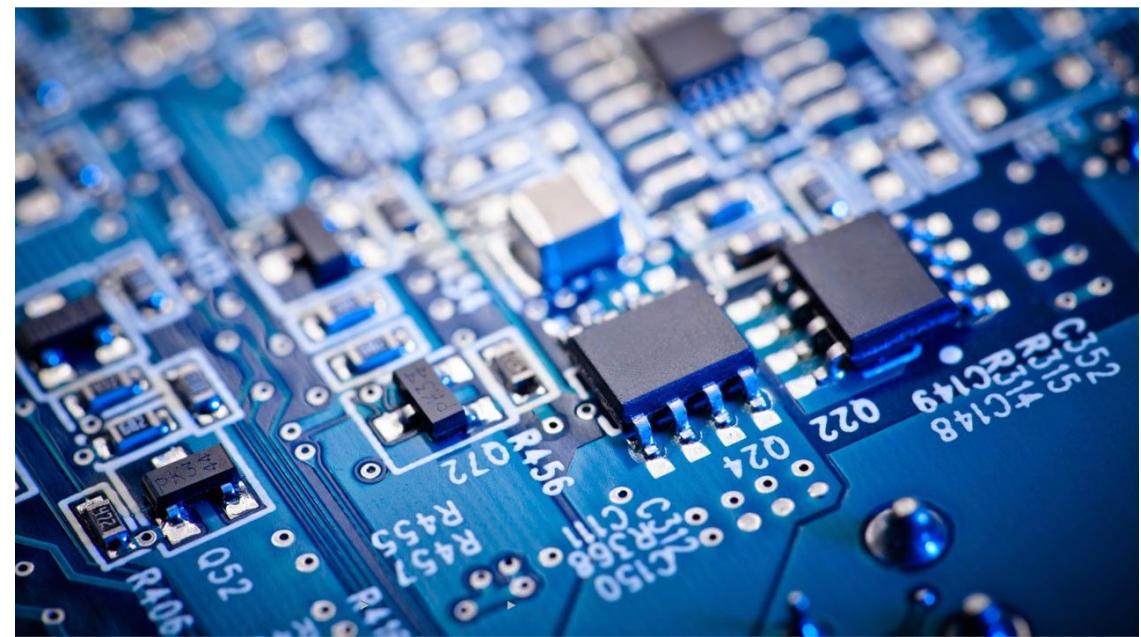
### 4.1. Desarrollo de Videojuegos.-

C++ es muy popular en la industria de los videojuegos por su alto rendimiento y control sobre los recursos del sistema. Los motores de juego más usados, como Unreal Engine, están escritos principalmente en C++. Permite manejar gráficos complejos, físicas en tiempo real y optimización para diferentes plataformas. **Ejemplo:** Pokemon Go, Brawl Stars, Clash Royale, Among Us, Fortnite, Halo, y algunos juegos de Super Mario



## 4.2. Sistemas embebido.-

En sistemas embebidos, donde el hardware es limitado y se requiere máxima eficiencia, C++ es ideal. Se utiliza para programar microcontroladores y dispositivos electrónicos como routers, automóviles, electrodomésticos inteligentes, y otros sistemas que necesitan control preciso y rápido.



## 4.3. Motores Gráficos.-

Los motores gráficos, que son el núcleo para renderizar imágenes en 3D o 2D, están frecuentemente desarrollados en C++. Esto se debe a la necesidad de manejar operaciones matemáticas complejas y optimizar el uso del GPU (unidad de procesamiento gráfico) para mostrar imágenes y animaciones con gran rapidez y calidad.



#### 4.4. Sistemas Operativos.-

C++ se emplea en el desarrollo de partes de sistemas operativos por su capacidad para interactuar directamente con el hardware y ofrecer eficiencia. Aunque los núcleos (kernels) suelen escribirse en C, C++ es usado para componentes que requieren orientación a objetos y estructuras más complejas.



#### 4.5. Aplicaciones de Alto Rendimiento.-

Para software que necesita procesar grandes cantidades de datos o realizar cálculos complejos rápidamente como simulaciones científicas, aplicaciones financieras, motores de búsqueda, y análisis de datos C++ es una opción preferida por su velocidad y eficiencia.



## 5. FUNDAMENTOS BÁSICOS

### 5.1. Entrada y Salida Estándar

Función	Descripción
<code>std::cout</code>	Imprime en pantalla.
<code>std::cin</code>	Recibe entrada desde el teclado.
<code>std::cerr</code>	Muestra errores (sin buffer).
<code>std::clog</code>	Muestra mensajes de log (con buffer).
<code>std::endl</code>	Inserta un salto de línea y vacía el buffer.

La biblioteca `iostream` en C++ permite gestionar la entrada y salida estándar de datos. Se incluye con `#include <iostream>`, y proporciona objetos clave para interactuar con el usuario. El objeto `std::cout` se utiliza para imprimir datos en pantalla, mientras que `std::cin` permite leer información ingresada desde el teclado. Para mostrar mensajes de error, se usa `std::cerr`, que no tiene buffer, por lo que muestra los errores inmediatamente.

Por otro lado, `std::clog` se emplea para mostrar mensajes de registro o depuración, pero con salida almacenada temporalmente en buffer. Finalmente, `std::endl` se utiliza para insertar un salto de línea y limpiar el buffer de salida, asegurando que los datos se impriman en ese momento.

**RECOMENDACIÓN:** La línea `using namespace std;` en C++ **permite usar directamente los elementos del espacio de nombres estándar (std)** sin tener que escribir `std::` cada vez. **EJEMPLO:**



## 5.2. cmath- FUNCIONES MATEMÁTICAS.-

Función	Descripción
<code>sqrt(x)</code>	Raíz cuadrada.
<code>pow(x, y)</code>	x elevado a la y.
<code>abs(x)</code>	Valor absoluto (entero).
<code>fabs(x)</code>	Valor absoluto (decimal).
<code>ceil(x)</code>	Redondea hacia arriba.
<code>floor(x)</code>	Redondea hacia abajo.
<code>round(x)</code>	Redondea al entero más cercano.
<code>sin(x) / cos(x) / tan(x)</code>	Funciones trigonométricas.
<code>log(x)</code>	Logaritmo natural.
<code>log10(x)</code>	Logaritmo base 10.

La biblioteca `cmath` en C++ proporciona una colección de funciones matemáticas para realizar cálculos comunes. Entre ellas, `sqrt(x)` calcula la raíz cuadrada de un número, mientras que `pow(x, y)` eleva x a la potencia y. Para obtener valores absolutos, se usa `abs(x)` para enteros y `fabs(x)` para números decimales. Las funciones `ceil(x)` y `floor(x)` redondean un número hacia arriba y hacia abajo, respectivamente, y `round(x)` redondea al entero más cercano. También incluye funciones trigonométricas como `sin(x)`, `cos(x)` y `tan(x)`. Finalmente, para operaciones con logaritmos, `log(x)` calcula el logaritmo natural y `log10(x)` el logaritmo en base 10.

### 5.2.1. CÓDIGO DE UNA CALCULADORA BÁSICA(SUMA Y RESTA)

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int num1, num2;
5     int suma, resta;
6     suma = num1 + num2;
7     cout << "Ingrese dos numeros: " << endl;
8     cin >> num1 >> num2;
9     cout << "La suma es: " << num1 + num2 << endl;
10    cout << "La resta es " << num1 - num2 << endl;
11    return 0;
12 }
```

```
Ingrese dos numeros:
7 3
La suma es: 10
La resta es 4
```

### 5.3. `cstdlib` - UTILIDADES ESTÁNDAR

Función	Descripción
<code>rand()</code>	Genera un número aleatorio.
<code>srand(x)</code>	Inicializa la semilla del aleatorio.
<code>system("comando")</code>	Ejecuta un comando del sistema.
<code>atoi(str)</code>	Convierte string a entero.
<code>atof(str)</code>	Convierte string a float.

La biblioteca `cstdlib` en C++ proporciona funciones útiles para tareas comunes como generación de números aleatorios, ejecución de comandos del sistema y conversión de cadenas a números. La función `rand()` genera un número entero pseudoaleatorio, mientras que `srand(x)` establece una semilla para inicializar esa generación, lo cual permite obtener diferentes secuencias cada vez que se ejecuta el programa. La función `system("comando")` permite ejecutar instrucciones del sistema operativo desde el programa, como limpiar la pantalla o pausar la consola. Por otro lado, `atoi(str)` convierte una cadena de texto a un número entero (`int`) y `atof(str)` convierte una cadena a un número decimal (`float` o `double`).

## 5.4. string - MANIPULACIÓN DE CADENAS

Método de string	Descripción
<code>s.length()</code>	Longitud del string.
<code>s.size()</code>	Igual a <code>.length()</code> .
<code>s.substr(pos, len)</code>	Substring desde <code>pos</code> , con longitud <code>len</code> .
<code>s.find("txt")</code>	Busca "txt" en el string.
<code>s.replace(p, l, "nuevo")</code>	Reemplaza desde pos <code>p</code> longitud <code>l</code> .
<code>s.append("texto")</code>	Añade al final.
<code>s.erase(pos, len)</code>	Elimina parte del string.
<code>s.insert(pos, "texto")</code>	Inserta en una posición.
<code>s.empty()</code>	Verifica si está vacío.

La clase `string` en C++ incluye varios métodos útiles para manipular cadenas de texto. Para obtener la longitud de un string se usan `s.length()` o `s.size()`, que son equivalentes. Con `s.substr(pos, len)` se extrae una subcadena desde una posición específica y con cierta longitud. La función `s.find("txt")` busca una subcadena dentro del string, y `s.replace(p, l, "nuevo")` permite reemplazar una parte de la cadena. Se puede añadir contenido al final con `s.append("texto")`, eliminar una sección con `s.erase(pos, len)`, o insertar texto en una posición específica con `s.insert(pos, "texto")`. Por último, `s.empty()` sirve para comprobar si la cadena está vacía.

## 5.5. algorithm - ALGORITMOS ÚTILES

Función	Descripción
<code>sort(inicio, fin)</code>	Ordena un arreglo o vector.
<code>reverse(inicio, fin)</code>	Invierte el orden.
<code>max(x, y) / min(x, y)</code>	Máximo / mínimo entre dos valores.
<code>count(inicio, fin, valor)</code>	Cuenta cuántas veces aparece un valor.
<code>find(inicio, fin, valor)</code>	Busca un valor.
<code>binary_search(inicio, fin, valor)</code>	Búsqueda binaria.

La biblioteca `algorithm` en C++ proporciona funciones listas para realizar tareas comunes sobre arreglos y contenedores como vectores. Con `sort(inicio, fin)` se ordenan los elementos en un rango, mientras que `reverse(inicio, fin)` invierte su orden. Las funciones `max(x, y)` y `min(x, y)` devuelven el valor máximo o mínimo entre dos elementos. Para

contar cuántas veces aparece un valor específico, se usa `count(inicio, fin, valor)`, y para buscarlo se puede usar `find(inicio, fin, valor)`. Además, `binary_search(inicio, fin, valor)` permite realizar una búsqueda binaria eficiente en un rango ordenado.

## 5.6. vector - ARREGLOS DINÁMICOS.-

Método	Descripción
<code>v.push_back(x)</code>	Añade elemento al final.
<code>v.pop_back()</code>	Elimina el último elemento.
<code>v.size()</code>	Cantidad de elementos.
<code>v.clear()</code>	Elimina todos los elementos.
<code>v.empty()</code>	Verifica si está vacío.
<code>v.at(i)</code>	Accede al elemento i con verificación.
<code>v[i]</code>	Accede directamente (sin verificación).

El contenedor `vector` en C++ permite trabajar con arreglos dinámicos de forma flexible. Para agregar elementos al final se utiliza `v.push_back(x)`, y para eliminar el último, `v.pop_back()`. El método `v.size()` devuelve la cantidad de elementos que contiene, mientras que `v.clear()` borra todos ellos. Para verificar si el vector está vacío se usa `v.empty()`. Para acceder a un elemento en una posición específica, se puede usar `v.at(i)` con verificación de rango, o directamente `v[i]` sin verificación.

## 5.7. map, set, unorderedmap - COLECCIONES CLAVE-CALOR Y CONJUNTOS

Contenedor	Método / Uso
<code>map[k] = v</code>	Asigna valor v a la clave k.
<code>map.at(k)</code>	Accede al valor de la clave k.
<code>map.find(k)</code>	Busca una clave.
<code>set.insert(x)</code>	Inserta elemento único.
<code>set.erase(x)</code>	Elimina el elemento x.
<code>set.count(x)</code>	Devuelve 1 si existe, 0 si no.

Los contenedores `map` y `set` en C++ son estructuras asociativas eficientes. En un `map`, se puede asignar un valor a una clave con `map[k] = v`, acceder a él con `map.at(k)`, y buscar

una clave usando `map.find(k)`. Por otro lado, el `set` almacena elementos únicos sin repetir. Se puede insertar un elemento con `set.insert(x)`, eliminarlo con `set.erase(x)`, y verificar si existe con `set.count(x)`, que devuelve 1 si el elemento está presente y 0 si no. Ambos contenedores permiten búsquedas rápidas y son muy útiles para organizar y acceder a datos de manera estructurada.

## 5.8. `ctime` - TIEMPO Y FECHAS

Función	Descripción
<code>time(0)</code>	Hora actual en segundos desde 1970.
<code>localtime()</code>	Convierte a formato de fecha local.
<code>asctime()</code>	Convierte fecha a string legible.
<code>difftime(t1, t2)</code>	Diferencia entre tiempos.

La biblioteca `ctime` en C++ permite trabajar con fechas y tiempos. La función `time(0)` devuelve la hora actual en segundos desde el 1 de enero de 1970 (formato Unix). Para convertir ese valor a una fecha local legible se usa `localtime()`, y con `asctime()` se transforma esa fecha en una cadena de texto fácil de leer. Además, `difftime(t1, t2)` permite calcular la diferencia en segundos entre dos tiempos, útil para medir duración o intervalos.

## 5.9. `fstream` - ARCHIVOS

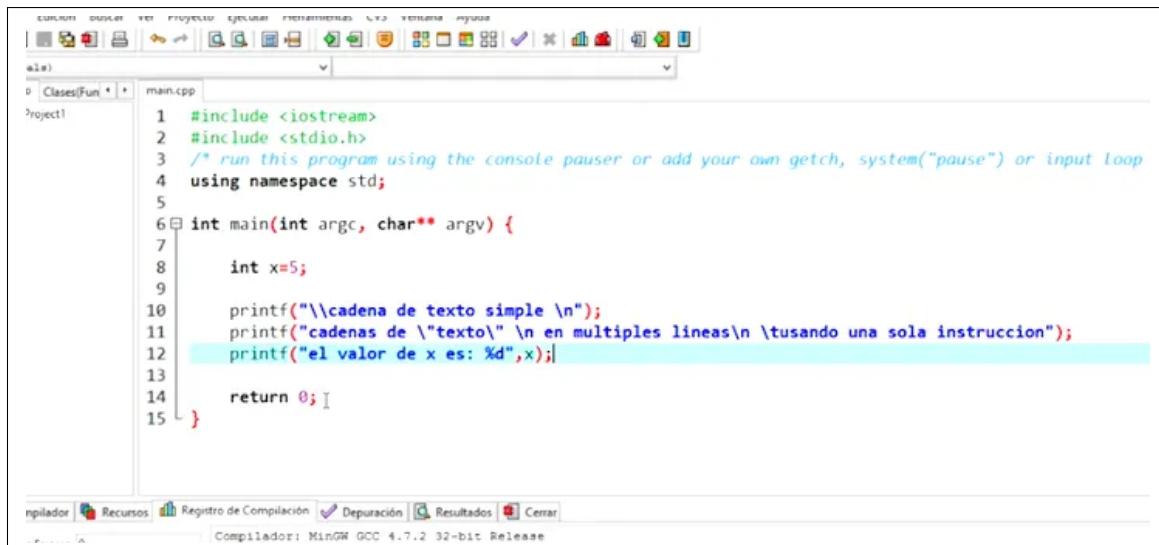
Función / Método	Descripción
<code>ofstream archivo("nombre.txt")</code>	Abre archivo para escritura.
<code>archivo &lt;&lt; "texto"</code>	Escribe en archivo.
<code>ifstream archivo("nombre.txt")</code>	Abre archivo para lectura.
<code>archivo &gt;&gt; variable</code>	Lee del archivo.
<code>archivo.is_open()</code>	Verifica si se abrió bien.
<code>archivo.close()</code>	Cierra el archivo.

En C++, la manipulación de archivos se realiza usando las clases `ofstream` para escritura y `ifstream` para lectura. Para escribir, se crea un objeto `ofstream archivo("nombre.txt")` y se usa `archivo << "texto"` para guardar datos. Para leer, se utiliza `ifstream archivo("nombre.txt")` y luego `archivo >> variable` para extraer información. El método `archivo.is_open()` permite verificar si el archivo se abrió correctamente, y `archivo.close()` se usa para cerrarlo una vez finalizado el proceso.

## 6. UNIDADES DE DESARROLLO

### 6.1. Programas en c++

Un **programa en C++** es una secuencia de instrucciones escritas en el lenguaje de programación C++ que, al ser compiladas y ejecutadas, permiten que una computadora realice tareas específicas. Todo programa en C++ comienza su ejecución desde una función especial llamada `main()`, que actúa como punto de entrada.



The screenshot shows a C++ development environment with the following details:

- Project Structure:** Shows a project named "Project1" containing a file "main.cpp".
- Code Editor:** Displays the following C++ code:

```
1 #include <iostream>
2 #include <stdio.h>
3 /* run this program using the console pauser or add your own getch, system("pause") or input loop
4 using namespace std;
5
6 int main(int argc, char** argv) {
7
8     int x=5;
9
10    printf("\cadena de texto simple \n");
11    printf("cadenas de \"texto\" \n en multiples lineas\n \tusando una sola instruccion");
12    printf("el valor de x es: %d",x);
13
14
15 }
```
- Output Window:** Shows the command prompt window with the following output:

```
C:\WINDOWS\system32\cmd. x + - □ ×
Hola Mundo
Presione una tecla para continuar . . .
```

### 6.2. Hola Mundo

Un **programa "Hola Mundo"** en C++ es el ejemplo más básico utilizado para introducir la sintaxis del lenguaje y demostrar el funcionamiento de un programa completo. Su propósito es mostrar un mensaje sencillo en pantalla (generalmente "Hola, mundo!"), sirviendo como primer paso para aprender a escribir, compilar y ejecutar código en C++.



The screenshot shows a C++ development environment with the following details:

- Code Editor:** Displays the following C++ code:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5
6     cout << "Hola Mundo" << endl;
7     return 0;
8 }
```
- Output Window:** Shows the command prompt window with the following output:

```
C:\WINDOWS\system32\cmd. x + - □ ×
Hola Mundo
Presione una tecla para continuar . . .
```

## 6.3. Arrays

### 6.3.1. ¿Qué es un array?

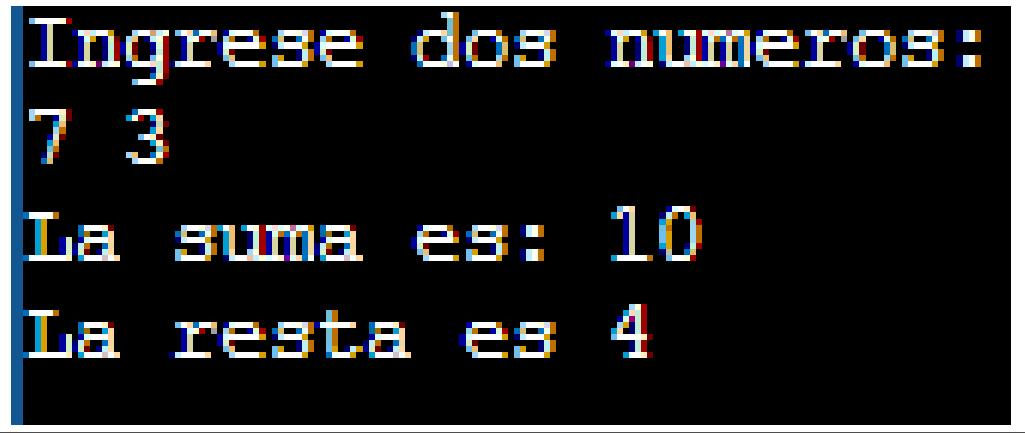
Un **array** (o **arreglo**) en C++ es una estructura de datos que permite almacenar una colección de elementos del mismo tipo en ubicaciones de memoria contiguas. Cada elemento del array se accede mediante un **índice numérico**, comenzando desde cero.

### 6.3.2. Características Principales

- **Tipo fijo:** todos los elementos deben ser del mismo tipo (por ejemplo, `int`, `float`).
- **Tamaño fijo:** el tamaño del array debe definirse al declararlo (en arrays estáticos).
- **Acceso directo:** permite acceder a cualquier elemento de forma rápida usando su índice.

### 6.3.3. Un Código como ejemplo:

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int numeros[5] = {10, 20, 30, 40, 50};
5     cout << "El primer número es: " << numeros[0]
6     << endl;
7     cout << "El tercero es: " << numeros[2] << endl;
8     return 0;
9 }
```



```
Ingrese dos números:
7 3
La suma es: 10
La resta es 4
```

## 6.4. Ventajas y desventajas de usar arrays en C++

Aspecto	Ventajas	Desventajas
Acceso	Acceso rápido a cualquier elemento ( $O(1)$ )	No permite búsquedas inteligentes (como en árboles)
Simplicidad	Fáciles de declarar y usar	Poca flexibilidad si el tamaño cambia frecuentemente
Memoria	Ocupan posiciones continuas de memoria	Difíciles de redimensionar una vez declarados
Tipo fijo	Homogeneidad facilita operaciones	No permiten mezclar diferentes tipos de datos

## 6.5. Tipos de Arrays en C++

### 6.5.1. Array Estático

- Tamaño fijo, definido en tiempo de compilación.
- Se almacena en la pila (stack).

```
1 int numeros [5] = {1, 2, 3, 4, 5};
```

### 6.5.2. Array Dinámico

- Tamaño definido en tiempo de ejecución.
- Se almacena en el heap (memoria dinámica).
- Se gestiona con new y delete.

```
1 int n;
2 cin >> n;
3 int* numeros = new int[n]; // array din mico
4 // Asignar valores
5 for (int i = 0; i < n; i++) {
6     numeros[i] = i * 10;
```

```

7     }
8     // Liberar memoria
9     delete [] numeros;

```

## 6.6. ¿Cuando se debe usar cada uno?.-

Situación	Tipo recomendado
Tamaño conocido y pequeño	Array estático
Tamaño definido por el usuario o variable	Array dinámico
Estructuras más flexibles o redimensionables	Usar <b>vector</b> (tema más avanzado)

## 6.7. Ejercicios Prácticos

### 6.7.1. Ejercicio 1: Suma de elementos de un array

**Enunciado:** Escribe un programa en C++ que:

- Declare un array de 5 enteros.
- Solicite al usuario ingresar los 5 valores.
- Calcule e imprima la suma total de los elementos.

**Objetivo:** Practicar la declaración, llenado, recorrido y suma de elementos de un array.

**Sugerencia de salida esperada:**

```

Ingrese 5 numeros:
10 20 30 40 50
La suma es: 150

```

### Código en C++

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     int numeros[5]; // Declaramos un array de tamaño 5
5     int suma = 0;
6     cout << "Ingrese 5 numeros:" << endl;
7     for (int i = 0; i < 5; i++) {
8         cin >> numeros[i];
9         suma += numeros[i]; // Sumamos cada elemento

```

```

10 }
11 cout << "La suma es: " << suma << endl;
12 return 0;
13 }
```

## 6.8. Ejercicio 2: Encontrar el mayor elemento

**Enunciado:** Crea un programa en C++ que:

- Use un array de n enteros (tamaño ingresado por el usuario).
- Pida al usuario los valores del array.
- Determine e imprima cuál es el número mayor.
- Objetivo: Usar arrays dinámicos, recorrerlos con condiciones y aplicar lógica de comparación.

**Sugerencia de salida esperada:**

```

Cuantos numeros vas a ingresar? 4
Ingresa 4 numeros:
12 45 7 29
El numero mayor es: 45
```

**Solución en C++ (array dinámico)**

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int n;
6     cout << " Cuantos numeros vas a ingresar? ";
7     cin >> n;
8     int* numeros = new int[n]; // Array dinamico
9     cout << "Ingresa " << n << " numeros:" << endl;
10    for (int i = 0; i < n; i++) {
11        cin >> numeros[i];
12    }
13    // Suponemos que el primero es el mayor
14    int mayor = numeros[0];
15    for (int i = 1; i < n; i++) {
```

```
16     if (numeros[i] > mayor) {  
17         mayor = numeros[i];  
18     }  
19 }  
20 cout << "El numero mayor es: " << mayor << endl;  
21 delete[] numeros; // Liberamos memoria dinamica  
22 return 0;  
23 }
```

## 6.9. Struct

Una **estructura** en C++ es un tipo de dato compuesto que permite agrupar **múltiples variables bajo un mismo nombre**, pudiendo ser de diferentes tipos. Es una forma de **organizar y representar datos complejos** que están relacionados entre sí.

Las estructuras se utilizan para modelar objetos del mundo real, como un alumno, un producto o un nodo en una lista enlazada, permitiendo agrupar atributos en un solo bloque lógico.

### 6.9.1. ¿Qué es una estructura (struct) en C++?

Una **estructura** en C++ es un tipo de dato compuesto que permite agrupar **múltiples variables bajo un mismo nombre**, pudiendo ser de diferentes tipos. Es una forma de **organizar y representar datos complejos** que están relacionados entre sí.

Las estructuras se utilizan para modelar objetos del mundo real, como un alumno, un producto o un nodo en una lista enlazada, permitiendo agrupar atributos en un solo bloque lógico.

### 6.9.2. Características Principales

- Se define usando la palabra clave **struct**.
- Puede contener variables (campos) de diferentes tipos.
- Los miembros se acceden con el operador punto (.) si es una variable normal, o flecha (->) si es un puntero.

#### Ejemplo Básico en C++

```
1 #include <iostream>
2 using namespace std;
3 struct Alumno {
4     string nombre;
5     int edad;
6     float promedio;
7 };
8 int main() {
9     Alumno a1;
10    a1.nombre = "Luis";
```

```

11     a1.edad = 20;
12     a1.promedio = 8.9;
13     cout << "Nombre: " << a1.nombre << endl;
14     cout << "Edad: " << a1.edad << endl;
15     cout << "Promedio: " << a1.promedio << endl;
16
17     return 0;
}

```

### Resultado

```

C:\WINDOWS\system32\cmd. + | v
Nombre: Luis
Edad: 20
Promedio: 8.9

Presione una tecla para continuar . .

```

#### 6.9.3. Ventajas y Desventajas del Struct

Ventaja / Uso	Descripción breve
Organización de datos	Agrupa varios atributos relacionados en una sola unidad.
Legibilidad del código	Hace el código más claro y estructurado.
Base para estructuras más complejas	Se usa para construir listas, pilas, árboles, grafos, etc.
Reutilización de datos	Permite declarar múltiples variables del mismo tipo estructurado.
Compatible con punteros	Facilita estructuras dinámicas como listas enlazadas.

#### 6.9.4. Ejercicio 1: Registrar un alumno

##### Enunciado:

Crea una estructura llamada Alumno que contenga los siguientes campos:

- Nombre (cadena)
- Edad (entero)
- Promedio (número decimal)

Luego, escribe un programa que:

1. Pida al usuario ingresar los datos de un alumno.
2. Los almacene en una variable de tipo Alumno.
3. Imprima los datos en pantalla

**Objetivo:** Practicar la declaración, inicialización y uso de una estructura simple.

### Solución

```
1 #include <iostream>
2 using namespace std;
3 struct Alumno {
4     string nombre;
5     int edad;
6     float promedio;
7 };
8 int main() {
9     Alumno a;
10    cout << "Ingrese el nombre del alumno: ";
11    getline(cin, a.nombre);
12    cout << "Ingrese la edad: ";
13    cin >> a.edad;
14    cout << "Ingrese el promedio: ";
15    cin >> a.promedio;
16    cout << "\nDatos del alumno:" << endl;
17    cout << "Nombre: " << a.nombre << endl;
18    cout << "Edad: " << a.edad << endl;
19    cout << "Promedio: " << a.promedio << endl;
20    return 0;
21 }
```

### Resultado

```
C:\WINDOWS\system32\cmd. + ▾

Ingrese el nombre del alumno: Jhon Kenedy Chambi Chambi
Ingrese la edad: 21
Ingrese el promedio: 14.6

Datos del alumno:
Nombre: Jhon Kenedy Chambi Chambi
Edad: 21
Promedio: 14.6

Presione una tecla para continuar . . . |
```

#### 6.9.5. Ejercicio 2: Promedio de varios productos

**Enunciado:**

Define una estructura llamada Producto con los siguientes campos:

- Nombre del producto
- Precio

**Luego:**

1. Pide al usuario ingresar datos para n productos.
2. Muestra todos los productos.
3. Calcula el promedio de los precios.

**Objetivo: Usar arreglos de estructuras, entrada dinámica, y cálculos básicos.**

**Solución**

```
1 #include <iostream>
2 using namespace std;
3
4 struct Producto {
5     string nombre;
6     float precio;
7 };
8
```

```

9 int main() {
10    int n;
11    cout << " Cuantos productos desea ingresar? ";
12    cin >> n;
13    if (n <= 0) {
14        cout << "Cantidad no valida. El programa terminara." << endl
15        ;
16        return 1;
17    }
18    Producto* productos = new Producto[n];
19    float suma = 0.0;
20    // Entrada de datos
21    for (int i = 0; i < n; i++) {
22        cout << "\nProducto " << i + 1 << ":" << endl;
23        cout << "Nombre: ";
24        cin.ignore(); // Ignorar salto de linea pendiente antes de
25        getline
26        getline(cin, productos[i].nombre);
27        cout << "Precio: ";
28        while (!(cin >> productos[i].precio)) {
29            cout << "Por favor ingrese un numero valido: ";
30            cin.clear();
31            cin.ignore(1000, '\n');
32        }
33        suma += productos[i].precio;
34    }
35    // Mostrar productos
36    cout << "\n      Lista de productos:\n";
37    for (int i = 0; i < n; i++) {
38        cout << "- " << productos[i].nombre << ": $" << productos[i]
39        ].precio << endl;
40    }
41    float promedio = suma / n;
42    cout << "\n      Precio promedio: $" << promedio << endl;
43    delete[] productos;
44    return 0;
45 }
```

```
C:\WINDOWS\system32\cmd. + | 
Cuantos productos desea ingresar? 1

Producto 1:
Nombre: PAPA HUAYRO
Precio: 13 SOLES

-fôi Lista de productos:
- PAPA HUAYRO: $13

-fA|| Precio promedio: $13

Presione una tecla para continuar . . . |
```

Resultado

## 7. Listas Enlazadas.-

### 7.0.1. ¿Qué es una Lista Enlazada?

Una lista enlazada es una estructura de datos dinámica donde los elementos (llamados nodos) están conectados entre sí mediante punteros. Cada nodo contiene dos partes:

- Un dato (valor).
- Un puntero al siguiente nodo en la lista.

Se usa cuando no se conoce de antemano el tamaño de la colección, o cuando se necesitan muchas inserciones o eliminaciones eficientes.

### 7.0.2. Ventajas de Listas Enlazadas

Ventaja	Descripción
Tamaño dinámico	Se puede expandir o reducir durante la ejecución.
Inserciones eficientes	Agregar o eliminar nodos no requiere mover otros elementos.
Uso de memoria flexible	No se desperdicia memoria como en arrays estáticos.

### 7.1. Estructura de un nodo en C++

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

#### 7.1.1. Ejemplo de básico: Crear y mostrar una lista de 3 elementos

```
1 #include <iostream>  
2 using namespace std;  
3  
4 struct Nodo {  
5     int dato;  
6     Nodo* siguiente;  
7 };  
8 int main() {  
9     // Crear nodos
```

```

10 Nodo* primero = new Nodo{10, nullptr};
11 Nodo* segundo = new Nodo{20, nullptr};
12 Nodo* tercero = new Nodo{30, nullptr};
13 // Enlazar los nodos
14 primero->siguiente = segundo;
15 segundo->siguiente = tercero;
16 // Recorrer e imprimir
17 Nodo* actual = primero;
18 while (actual != nullptr) {
19     cout << actual->dato << " -> ";
20     actual = actual->siguiente;
21 }
22 cout << "NULL" << endl;
23 // Liberar memoria
24 delete primero;
25 delete segundo;
26 delete tercero;
27 return 0;
28 }
```

**Resultado del Programa:**

```

C:\WINDOWS\system32\cmd. + | 
10 -> 20 -> 30 -> NULL
Presione una tecla para continuar . . . |
```

### 7.1.2. Ejercicios de Práctica

#### Enunciado 1:

Crea un programa que permita ingresar números enteros y los inserte al inicio de una lista enlazada. El usuario deja de ingresar datos al escribir -1. Luego muestra toda la lista.

**Objetivo:** Practicar inserción al principio de la lista y recorrido dinámico.

## Solución

```
1 #include <iostream>
2 using namespace std;
3
4 struct Nodo {
5     int dato;
6     Nodo* siguiente;
7 };
8 int main() {
9     Nodo* inicio = nullptr;
10    int numero;
11    cout << "Ingrese numeros enteros (use -1 para terminar):" <<
12        endl;
13    while (true) {
14        cin >> numero;
15        if (numero == -1) break;
16        Nodo* nuevo = new Nodo{numero, inicio};
17        inicio = nuevo;
18    }
19    cout << "\nLista generada:\n";
20    Nodo* actual = inicio;
21    while (actual != nullptr) {
22        cout << actual->dato << " -> ";
23        actual = actual->siguiente;
24    }
25    cout << "NULL" << endl;
26    // Liberar memoria
27    while (inicio != nullptr) {
28        Nodo* temp = inicio;
29        inicio = inicio->siguiente;
30        delete temp;
31    }
32    return 0;
}
```

Resultado del Programa:

```
C:\WINDOWS\system32\cmd. + ^

Ingrese numeros enteros (use -1 para terminar):
20 40 60 70
-1

Lista generada:
70 -> 60 -> 40 -> 20 -> NULL

Presione una tecla para continuar . . . |
```

**Enunciado 2:** Usa una lista enlazada simple y cuenta cuántos nodos hay en ella. El usuario ingresa los datos y termina con -1.

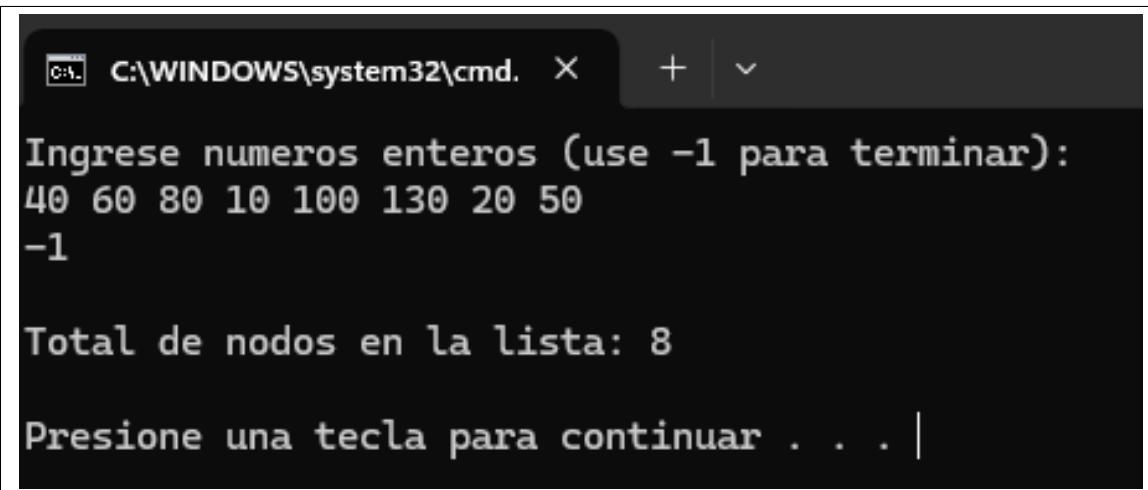
**Objetivo:** Practicar el recorrido y conteo de nodos en una lista dinámica.

**Solución:**

```
1 #include <iostream>
2 using namespace std;
3
4 struct Nodo {
5     int dato;
6     Nodo* siguiente;
7 };
8
9 int main() {
10     Nodo* inicio = nullptr;
11     int numero, contador = 0;
12     cout << "Ingrese numeros enteros (use -1 para terminar):" <<
13         endl;
14     while (true) {
15         cin >> numero;
16         if (numero == -1) break;
17
18         Nodo* nuevo = new Nodo{numero, inicio};
19         inicio = nuevo;
20     }
21     Nodo* actual = inicio;
```

```
21     while (actual != nullptr) {
22         contador++;
23         actual = actual->siguiente;
24     }
25     cout << "\nTotal de nodos en la lista: " << contador << endl;
26     // Liberar memoria
27     while (inicio != nullptr) {
28         Nodo* temp = inicio;
29         inicio = inicio->siguiente;
30         delete temp;
31     }
32     return 0;
33 }
```

Resultado:



```
C:\WINDOWS\system32\cmd. + ▾
Ingrese numeros enteros (use -1 para terminar):
40 60 80 10 100 130 20 50
-1

Total de nodos en la lista: 8

Presione una tecla para continuar . . . |
```

## 8. Listas Dobles y Listas Circulares.-

Las listas enlazadas son una de las estructuras de datos fundamentales en programación, y existen diversas variantes adaptadas a diferentes necesidades. Entre las más versátiles y potentes están las listas doblemente enlazadas y las listas circulares.

### 8.1. ¿Qué es una Lista Dblemente Circulada?

Una **lista doblemente enlazada** (*doubly linked list*) es una estructura de datos lineal compuesta por una secuencia de nodos. A diferencia de la lista simplemente enlazada, donde cada nodo tiene un único puntero al siguiente nodo, aquí **cada nodo contiene dos punteros**:

- Uno apunta al nodo siguiente.
- Otro apunta al nodo anterior.

#### 8.1.1. Estructura básica en C++.-

```
1 struct Nodo {  
2     int dato;  
3     Nodo* siguiente;  
4     Nodo* anterior;  
5 };
```

#### 8.1.2. Aplicaciones Cómunes

- Navegadores web (retroceder y avanzar entre páginas).
- Edición de texto con historial.
- Implementación de estructuras como colas dobles (deques).

#### 8.1.3. Ventajas

- Navegación bidireccional.
- Eliminaciones más eficientes cuando se tiene referencia al nodo.
- Más versátil que una lista simplemente enlazada.

#### 8.1.4. Desventajas.-

- Requiere más memoria (dos punteros por nodo).
- El manejo de punteros es más complejo, propenso a errores si no se actualizan correctamente.

### 8.2. ¿Qué es una Lista Circular.-

Una lista circular es una estructura de datos donde el último nodo no apunta a nullptr (como en listas normales), sino que apunta nuevamente al primer nodo, formando un bucle cerrado o circular. Puede ser simple o doblemente enlazada.

- En una lista circular simple, cada nodo tiene un solo puntero que apunta al siguiente, y el último apunta al primero.
- En una lista circular doblemente enlazada, cada nodo tiene dos punteros (siguiente y anterior), y además:
  - El primero apunta al último como anterior.
  - El último apunta al primero como siguiente.

Esto convierte la lista en una secuencia sin inicio ni final fijo, ideal para representar procesos cíclicos.

### 8.3. Ejemplo de Nodo en Lista Circular Simple.-

```
1 struct Nodo {  
2     int dato;  
3     Nodo* siguiente;  
4 };
```

### 8.4. Ejemplo de Nodo en Lista Circular Doble.-

```
1 struct Nodo {  
2     int dato;  
3     Nodo* siguiente;  
4     Nodo* anterior;  
5 };
```

## 8.5. Aplicaciones

- Sistemas operativos (planificación de procesos cíclica).
- Juegos por turnos.
- Buffers circulares (circular queue).
- Menús o interfaces de navegación en bucle.

## 8.6. Ventajas

- No hay necesidad de verificar si se llegó al final (`nullptr`).
- Perfectas para ciclos repetitivos.
- En la versión doble, permite recorrido completo en ambas direcciones, como una lista cerrada.

## 8.7. Cuidados Especiales.-

- El recorrido debe usar un criterio diferente (`temp != inicio`) para evitar bucles infinitos.
- Requiere manejo riguroso de punteros para no romper el ciclo.

## 8.8. Comparación entre Listas Dobles y Listas Circulares.-

Característica	Lista Dblemente Enlazada	Lista Circular
Dirección de recorrido	Adelante y atrás	Depende: solo adelante o ambas
Inicio y fin definidos	Sí	No (forma un bucle cerrado)
Punteros por nodo	2 (siguiente y anterior)	1 o 2 (según sea simple o doble)
Requiere <code>nullptr</code> final	Sí	No (el último enlaza al primero)
Eficiencia en inserciones	Alta en ambos extremos	Alta, especialmente si es circular doble
Casos de uso típicos	Navegación bidireccional	Procesos cíclicos, estructuras sin fin

## 9. Colas (Queues)

### 9.1. Definición y qué es Colas en C++

Una **cola** (*queue*) es una estructura de datos **lineal** que funciona bajo el principio **FIFO** (*First In, First Out*), lo que significa que el primer elemento en entrar es el primero en salir. Este comportamiento es análogo a una fila en la vida real, como la fila para pagar en una tienda o para abordar un autobús: los primeros en llegar son los primeros en ser atendidos.

En una cola, los elementos se insertan por un extremo llamado **final** (o **rear**) y se eliminan por el otro extremo llamado **frente** (o **front**).

Representación de una Cola

```
1 [10] [20] [30] [40]  
2  
3 Front           Rear
```

### 9.2. Operaciones Principales de una Cola

Operación	Descripción
<code>enqueue(x)</code>	Inserta el elemento <code>x</code> al final de la cola.
<code>dequeue()</code>	Elimina el elemento que está en el frente de la cola.
<code>front()</code>	Muestra el elemento en el frente sin eliminarlo.
<code>isEmpty()</code>	Verifica si la cola está vacía.

### 9.3. Ventajas de las Colas

- Útiles para modelar procesos en orden de llegada.
- Se utilizan en algoritmos como BFS (Búsqueda en Anchura).
- Son fundamentales en sistemas operativos, gestión de tareas y programación concurrente.

### 9.4. Implementación en C++

Las colas pueden implementarse de diferentes formas en C++:

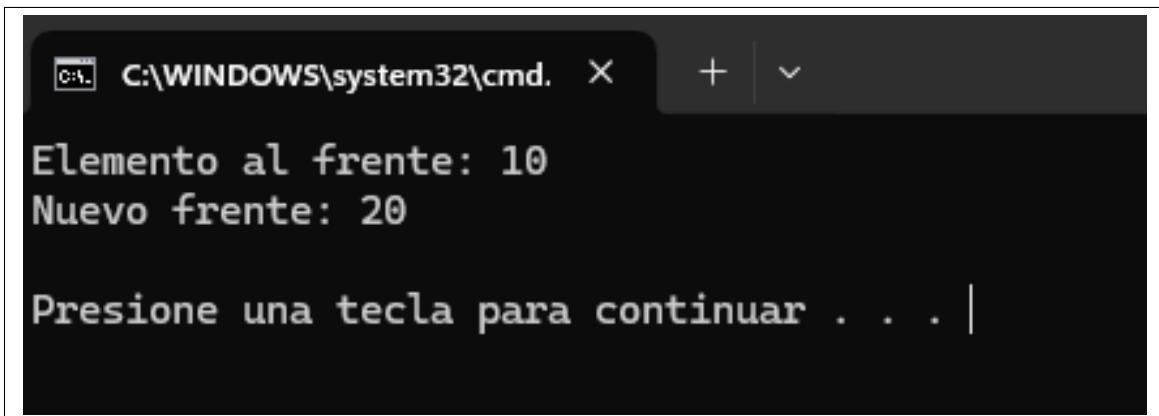
1. Con arreglos (arrays).
2. Con listas enlazadas.

3. Con la clase queue de la STL (Standard Template Library).

### Ejemplo de un Programa

```
1 #include <iostream>
2 #include <queue>
3 using namespace std;
4
5 int main() {
6     queue<int> cola;
7     cola.push(10); // enqueue
8     cola.push(20);
9     cola.push(30);
10    cout << "Elemento al frente: " << cola.front() << endl; // 10
11    cola.pop(); // dequeue
12    cout << "Nuevo frente: " << cola.front() << endl; // 20
13    return 0;
14 }
```

Resultado:



```
C:\WINDOWS\system32\cmd. + | v
Elemento al frente: 10
Nuevo frente: 20

Presione una tecla para continuar . . . |
```

## 9.5. Ejercicios Prácticos en C++.-

### 9.5.1. Enunciado N° 1.-

Simula una fila en una tienda donde los clientes llegan en un orden determinado. El programa debe:

- Insertar los clientes en la cola.
- Atender a los clientes en el orden en que llegaron.

- Mostrar cuántos clientes fueron atendidos.

Código:

```
1 #include <iostream>
2 #include <queue>
3 #include <string>
4 using namespace std;
5
6 int main() {
7     queue<string> clientes;
8     int n;
9     cout << " Cuantos clientes llegaron? ";
10    cin >> n;
11    // Insertar clientes
12    for (int i = 0; i < n; i++) {
13        string nombre;
14        cout << "Nombre del cliente " << (i + 1) << ":" ;
15        cin >> nombre;
16        clientes.push(nombre);
17    }
18    cout << "\nAtendiendo clientes...\n";
19    while (!clientes.empty()) {
20        cout << "Atendido: " << clientes.front() << endl;
21        clientes.pop();
22    }
23    cout << "\nTodos los clientes han sido atendidos.\n";
24    return 0;
25 }
```

Resultado del Programa

The screenshot shows a Windows Command Prompt window with the title bar "C:\WINDOWS\system32\cmd." and a dark background. The window contains the following text:

```
TJCuantos clientes llegaron? 5
Nombre del cliente 1: Juan
Nombre del cliente 2: Carlos
Nombre del cliente 3: Pedro
Nombre del cliente 4: Rechard
Nombre del cliente 5: Fred

Atendiendo clientes...
Atendido: Juan
Atendido: Carlos
Atendido: Pedro
Atendido: Rechard
Atendido: Fred

Todos los clientes han sido atendidos.

Presione una tecla para continuar . . . |
```

### 9.5.2. Enunciado N° 2

Dada una secuencia de números enteros, determinar si forma un palíndromo (se lee igual al derecho y al revés). No se debe alterar el orden de la cola original. **Lógica**

- Usar una queue para almacenar la secuencia original.
- Usar una stack para invertir la secuencia.
- Comparar ambos elementos uno a uno.

#### Código en C++

```
1 #include <iostream>
2 #include <queue>
```

```

3 #include <stack>
4 using namespace std;
5
6 bool esPalindromo(queue<int> q) {
7     stack<int> s;
8     queue<int> copia = q;
9
10    // Llenar la pila para invertir el orden
11    while (!q.empty()) {
12        s.push(q.front());
13        q.pop();
14    }
15    // Comparar la cola original con la pila
16    while (!copia.empty()) {
17        if (copia.front() != s.top())
18            return false;
19        copia.pop();
20        s.pop();
21    }
22    return true;
23 }
24 int main() {
25     queue<int> numeros;
26     int n, num;
27     cout << " Cuantos    numeros vas a ingresar? ";
28     cin >> n;
29     for (int i = 0; i < n; i++) {
30         cout << "Numero " << (i + 1) << ":" ;
31         cin >> num;
32         numeros.push(num);
33     }
34     if (esPalindromo(numeros))
35         cout << "La cola forma un palindromo.\n";
36     else
37         cout << "La cola no forma un palindromo.\n";
38     return 0;
39 }
```

## Resultado del Programa

```
C:\WINDOWS\system32\cmd. X + ▾
Cuantos numeros vas a ingresar? 5
Numero 1: 20
Numero 2: 30
Numero 3: 40
Numero 4: 30
Numero 5: 20
La cola forma un palindromo.

Presione una tecla para continuar . . .
```

## 10. Pilas (Stacks).-

### 10.1. Definición.-

Una pila (stack) es una estructura de datos lineal que sigue el principio LIFO (Last In, First Out), es decir, el último elemento en entrar es el primero en salir. Puedes imaginarla como una pila de platos: solo puedes acceder al último que colocaste arriba.

### 10.2. ¿Cómo funciona una Pila?.-

En una pila, las operaciones se realizan en un solo extremo llamado tope (o top). Los elementos se insertan con la operación push y se eliminan con la operación pop. **Representación Conceptual.-**

```
1      Top
2
3
4      30      ultimo en entrar
5
6      20
7
8      10      primero en entrar
9
```

### 10.3. Operaciones Principales.-

Operación	Descripción
push(x)	Inserta el elemento x en la parte superior.
pop()	Elimina el elemento en la parte superior.
top()	Retorna el elemento superior sin eliminarlo.
isEmpty()	Verifica si la pila está vacía.
size()	Devuelve la cantidad de elementos en la pila.

### 10.4. Ventajas de las Pilas.-

- Sencillas de implementar y entender.
- Útiles para problemas de reversión (deshacer acciones).
- Se utilizan en expresiones matemáticas, compiladores, y más.

## 10.5. Implementación en C++.-

```
1 #include <iostream>
2 #include <stack>
3 using namespace std;
4
5 int main() {
6     stack<int> pila;
7     pila.push(10);
8     pila.push(20);
9     pila.push(30);
10    cout << "Elemento en el tope: " << pila.top() << endl;
11    pila.pop();
12    cout << "Nuevo tope: " << pila.top() << endl;
13    return 0;
14 }
```

Resultado del Programa:

```
C:\WINDOWS\system32\cmd. + ▾
Elemento en el tope: 30
Nuevo tope: 20

Presione una tecla para continuar . . . |
```

## 10.6. Aplicaciones comunes con pilas

- Reversión de cadenas (invertir texto).
- Verificación de expresiones bien formadas (paréntesis, llaves).
- Control de llamadas de funciones (pila de ejecución).
- Navegadores (botón de “atrás”).
- Algoritmos como DFS (búsqueda en profundidad).

## 10.7. Ejercicios Prácticos

### 10.7.1. Enunciado N° 1

Crea un programa que reciba una cadena de texto e invierta su contenido usando una pila. No uses funciones de la STL como reverse() ni arrays auxiliares para invertir. **Lógica** Insertar cada carácter en una pila. Luego, al hacer pop, se recuperan en orden inverso.

Código en C++

```
1 #include <iostream>
2 #include <stack>
3 #include <string>
4 using namespace std;
5
6 int main() {
7     string texto;
8     stack<char> pila;
9     cout << "Ingresa una cadena: ";
10    getline(cin, texto);
11    // Push de caracteres a la pila
12    for (char c : texto) {
13        pila.push(c);
14    }
15    cout << "Cadena invertida: ";
16    while (!pila.empty()) {
17        cout << pila.top();
18        pila.pop();
19    }
20    cout << endl;
21    return 0;
22 }
```

Resultado del Programa

```
C:\WINDOWS\system32\cmd. × + ▾  
Ingresá una cadena: 1 4 5 6 3 9 8 7 10  
Cadena invertida: 01 7 8 9 3 6 5 4 1  
Presione una tecla para continuar . . .
```

## 10.8. Enunciado N° 2

Crea un programa que determine si una expresión matemática tiene los paréntesis correctamente balanceados. Solo considera los caracteres ( y ). **Lógica**

- Cada vez que encuentras un ( lo insertas en la pila.
- Cada vez que encuentras un ), haces un pop().
- Si la pila queda vacía al final, y no hubo errores, está balanceado.

### Código

```
1 #include <iostream>  
2 #include <stack>  
3 #include <string>  
4 using namespace std;  
5  
6 bool estaBalanceado(const string& expr) {  
7     stack<char> pila;  
8  
9     for (char c : expr) {  
10         if (c == '(') {  
11             pila.push(c);  
12         } else if (c == ')') {  
13             if (pila.empty()) return false;  
14             pila.pop();  
15         }  
16     }
```

```
17     return pila.empty();
18 }
19 int main() {
20     string expresion;
21     cout << "Ingresa una expresion: ";
22     getline(cin, expresion);
23     if (estaBalanceado(expresion))
24         cout << "Los parentesis estan balanceados.\n";
25     else
26         cout << "Los parentesis NO est n balanceados.\n";
27     return 0;
28 }
```

### Resultado del Programa

The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.' with a dark theme. The window contains the following text:

```
Ingresa una expresion: (5 + 3) * (2 - (1 + 1))
Los parentesis estan balanceados.

Presione una tecla para continuar . . .
```

## 11. Recursión.-

La recursión es una técnica de programación donde una función se llama a sí misma para resolver un problema. Se usa comúnmente cuando un problema puede dividirse en subproblemas más pequeños y de la misma naturaleza.

Una función recursiva siempre debe tener al menos dos componentes:

1. **Caso base (condición de parada)**: Define cuándo debe detenerse la recursión.
2. **Llamada recursiva**: Es la llamada que la función hace a sí misma con un valor modificado que lo acerque al caso base.

### 11.1. Ejemplo Básico: Factorial.-

La función factorial se define como:

$$n! = n \times (n-1)! \text{ y } 0!=1$$

#### 11.1.1. Código en C++

```
1 #include <iostream>
2 using namespace std;
3
4 int factorial(int n) {
5     if (n == 0) // Caso base
6         return 1;
7     else
8         return n * factorial(n - 1); // Llamada recursiva
9 }
10 int main() {
11     int num;
12     cout << "Ingresa un numero: ";
13     cin >> num;
14     cout << "Factorial de " << num << " es: " << factorial(num) <<
15         endl;
16     return 0;
17 }
```

#### Resultado del Programa.-

The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.' with the following text displayed:

```
Ingresá un numero: 5
Factorial de 5 es: 120

Presione una tecla para continuar . . .
```

## 11.2. Representación Visual: Llamadas Recursivas del Factorial(3)

```
1     factorial(3)
2         3 * factorial(2)
3             2 * factorial(1)
4                 1 * factorial(0)
5                     1           caso base
```

## 11.3. Ventajas de la Recursión.-

- Código más limpio y legible para problemas recursivos (como árboles, torres de Hanoi).
- Permite resolver problemas complejos con menor esfuerzo de codificación.

## 11.4. Desventajas.-

- Consume más memoria por las llamadas anidadas en la pila.
- Puede ser menos eficiente que las soluciones iterativas.
- Si no hay caso base, genera desbordamiento de pila (stack overflow).

## 11.5. Casos Clásicos donde se usa la Recursión.-

- Cálculo de factoriales, potencias, sumas.
- Series de Fibonacci.
- Búsqueda en estructuras jerárquicas (como árboles).

- Algoritmos de ordenamiento como QuickSort y MergeSort.
- Problemas como Torres de Hanoi, laberintos y backtracking.

## 11.6. Ejercicios Prácticos.-

### 11.6.1. Enunciado N° 1.-

Implementa una función recursiva que sume los números del 1 al N.

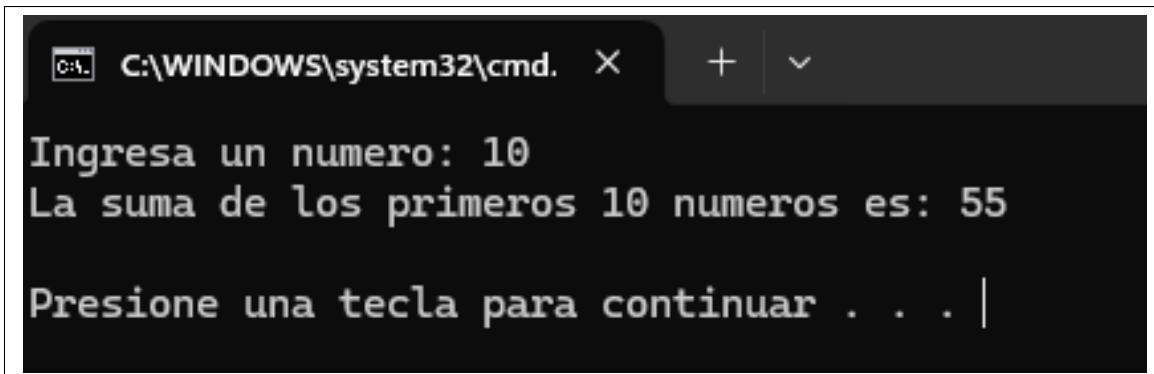
Código en C++

```

1      #include <iostream>
2  using namespace std;
3
4  int suma(int n) {
5      if (n == 1)
6          return 1;
7      else
8          return n + suma(n - 1);
9  }
10
11 int main() {
12     int n;
13     cout << "Ingresa un numero: ";
14     cin >> n;
15     cout << "La suma de los primeros " << n << " numeros es: " <<
16         suma(n) << endl;
17     return 0;
}

```

Resultado del Programa.-



```

C:\WINDOWS\system32\cmd. + | 
Ingresá un numero: 10
La suma de los primeros 10 numeros es: 55
Presione una tecla para continuar . . .

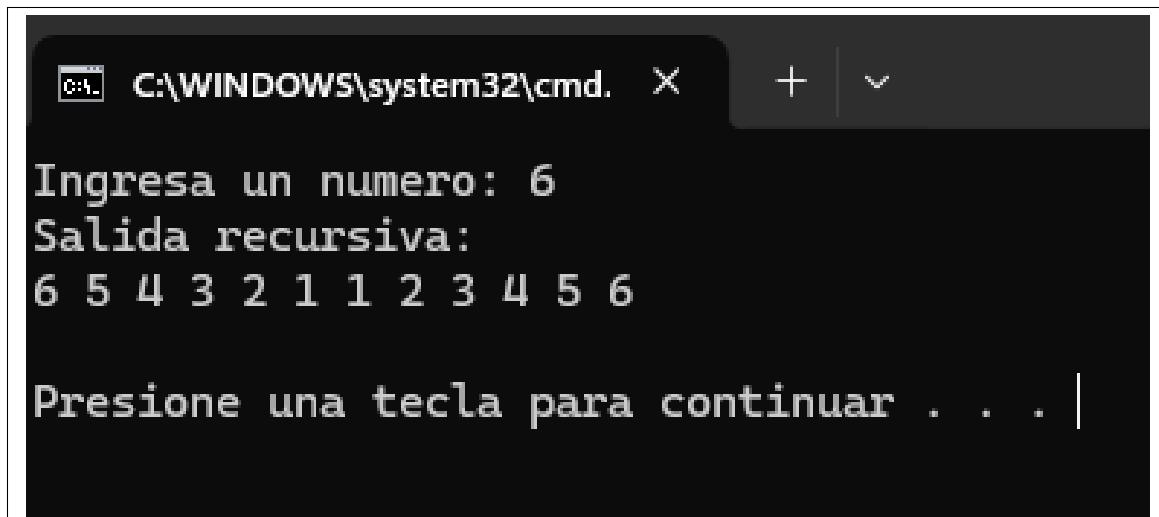
```

### 11.6.2. Enunciado N° 2.-

Crea una función recursiva que imprima los números del 1 al N y luego del N al 1. **Código en C++**

```
1 #include <iostream>
2 using namespace std;
3
4 void imprimir(int n) {
5     if (n == 0)
6         return;
7     cout << n << " ";
8     imprimir(n - 1);
9     cout << n << " ";
10 }
11
12 int main() {
13     int n;
14     cout << "Ingresa un numero: ";
15     cin >> n;
16     cout << "Salida recursiva:\n";
17     imprimir(n);
18     cout << endl;
19     return 0;
20 }
```

Resultado del Programa



The screenshot shows a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.' with a dark theme. The window contains the following text:

```
Ingresá un numero: 6
Salida recursiva:
6 5 4 3 2 1 1 2 3 4 5 6

Presione una tecla para continuar . . . |
```

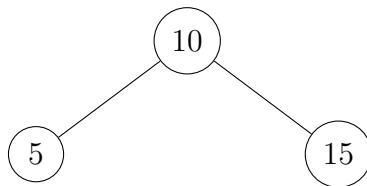
## 12. Árboles Binarios

### 12.1. ¿Qué es un árbol binario?

Un árbol binario es una estructura de datos en forma de jerarquía, donde cada elemento se llama nodo, y cada nodo puede tener como máximo dos hijos: uno a la izquierda y otro a la derecha.

El primer nodo del árbol se llama raíz, y los nodos que no tienen hijos se llaman hojas.

### 12.2. Estructura básica



### 12.3. Características principales:

1. Cada nodo puede tener 0, 1 o 2 hijos.
2. Se usa para representar datos jerárquicos (como carpetas o decisiones).
3. Sirve de base para estructuras más avanzadas como árboles de búsqueda, árboles balanceados, y árboles AVL.

### 12.4. Código simple de árbol binario en C++

```
1 #include <iostream>
2 using namespace std;
3
4 // Definición del nodo del rbol
5 struct Nodo {
6     int dato;
7     Nodo* izquierda;
8     Nodo* derecha;
9 };
10
11 // Función para crear un nuevo nodo
12 Nodo* crearNodo(int valor) {
```

```

13     Nodo* nuevo = new Nodo();
14     nuevo->dato = valor;
15     nuevo->izquierda = nullptr;
16     nuevo->derecha = nullptr;
17     return nuevo;
18 }
19
20 // Insertar un valor en el rbol binario (BST)
21 Nodo* insertar(Nodo* raiz, int valor) {
22     if (raiz == nullptr)
23         return crearNodo(valor);
24
25     if (valor < raiz->dato)
26         raiz->izquierda = insertar(raiz->izquierda, valor);
27     else
28         raiz->derecha = insertar(raiz->derecha, valor);
29
30     return raiz;
31 }
32
33 // Recorrido inorden: izquierda - ra z - derecha
34 void inOrden(Nodo* raiz) {
35     if (raiz != nullptr) {
36         inOrden(raiz->izquierda);
37         cout << raiz->dato << " ";
38         inOrden(raiz->derecha);
39     }
40 }
41
42 int main() {
43     Nodo* raiz = nullptr;
44
45     // Insertar algunos valores
46     raiz = insertar(raiz, 10);
47     raiz = insertar(raiz, 5);
48     raiz = insertar(raiz, 15);
49     raiz = insertar(raiz, 3);
50     raiz = insertar(raiz, 7);

```

```
51
52     cout << "Recorrido en orden del arbol: ";
53     inOrden(raiz);
54     cout << endl;
55
56     return 0;
57 }
```

#### 12.4.1. Orden Visual

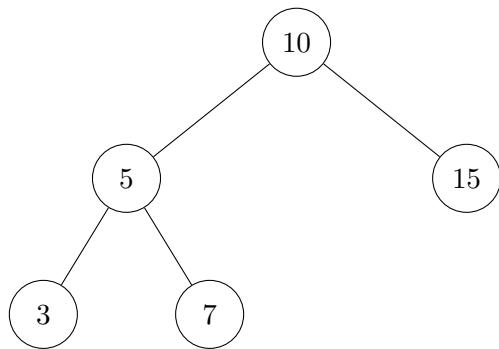


Figura 1: Árbol binario simple usado en el ejemplo de código C++.

## 13. Árboles Balanceados

### 13.1. Definición general

Un árbol balanceado es un tipo especial de árbol binario donde se mantiene controlada la altura del árbol para evitar que crezca desproporcionadamente hacia un solo lado. En términos simples, un árbol balanceado trata de asegurar que todos los nodos estén lo más cerca posible de la raíz, de forma que las operaciones de búsqueda, inserción o eliminación se realicen rápidamente.

### 13.2. ¿Por qué balancear un árbol?

- **Eficiencia:** Mantener baja la altura asegura tiempos logarítmicos.
- **Evitar degeneración:** Un BST sin balance puede transformarse en una lista.
- **Estabilidad:** El árbol responde mejor ante grandes volúmenes de datos.

### 13.3. Tipos comunes de árboles balanceados

Tipo	Descripción breve
AVL	Árbol binario de búsqueda auto-balanceado que realiza rotaciones simples o dobles para corregir desequilibrios.
Rojo-Negro	Usa colores para asegurar que la ruta más larga nunca supere el doble de la más corta.
Splay	Tras cada acceso, lleva el nodo consultado a la raíz para acelerar accesos repetidos.
B/B+	Árboles multi-camino empleados en discos y bases de datos; mantienen alto factor de ramificación.
Heap	Árbol casi completo usado para montículos ( <i>priority queues</i> ) y heapsort.

Cuadro 1: Resumen de árboles balanceados más utilizados.

### 13.4. Operaciones y complejidad

Operación	Árbol balanceado	Árbol desbalanceado
Búsqueda	$O(\log n)$	$O(n)$
Inserción	$O(\log n)$	$O(n)$
Eliminación	$O(\log n)$	$O(n)$

### 13.5. Rotaciones (Árbol AVL)

- Rotación simple a la derecha
- Rotación simple a la izquierda
- Rotación doble izquierda-derecha
- Rotación doble derecha-izquierda

### 13.6. Operaciones y complejidad

Operación	Árbol balanceado	Árbol desbalanceado
Búsqueda	$O(\log n)$	$O(n)$
Inserción	$O(\log n)$	$O(n)$
Eliminación	$O(\log n)$	$O(n)$

## 13.7. Rotaciones (Árbol AVL)

- Rotación simple a la derecha
- Rotación simple a la izquierda
- Rotación doble izquierda-derecha
- Rotación doble derecha-izquierda

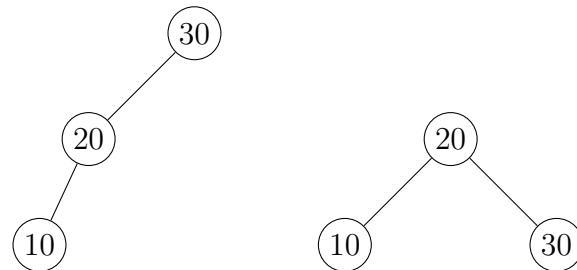


Figura 2: Ejemplo de desequilibrio Izquierda–Izquierda y rotación simple a la derecha.

### 13.7.1. Ejemplo de inserción en un AVL (C++)

```
1 #include <iostream>
2 #include <algorithm> // std::max
3 using namespace std;
4
5 //=====
6 // Nodo del rbol AVL
7 //=====
8 struct Node {
9     int key;           // Valor almacenado
10    int height;        // Altura del sub rbol
11    Node *left;        // Hijo izquierdo
12    Node *right;       // Hijo derecho
13
14    explicit Node(int k)
15        : key(k), height(1), left(nullptr), right(nullptr) {}
16 };
17 //-----
```

```

19 // Funciones auxiliares
20 //-----
21 int height(Node* n) { return n ? n->height : 0; }
22
23 int balance(Node* n) { return n ? height(n->left) - height(n->right)
24 : 0; }
25 //-----
26 // Rotaci n simple a la derecha
27 //-----
28 Node* rotateRight(Node* y) {
29     Node* x = y->left;
30     Node* T2 = x->right;
31
32     // Rotar
33     x->right = y;
34     y->left = T2;
35
36     // Actualizar alturas
37     y->height = 1 + max(height(y->left), height(y->right));
38     x->height = 1 + max(height(x->left), height(x->right));
39
40     return x; // Nueva ra z
41 }
42 //-----
43 // Rotaci n simple a la izquierda
44 //-----
45 Node* rotateLeft(Node* x) {
46     Node* y = x->right;
47     Node* T2 = y->left;
48     // Rotar
49     y->left = x;
50     x->right = T2;
51     // Actualizar alturas
52     x->height = 1 + max(height(x->left), height(x->right));
53     y->height = 1 + max(height(y->left), height(y->right));
54     return y; // Nueva ra z
55 }

```

```

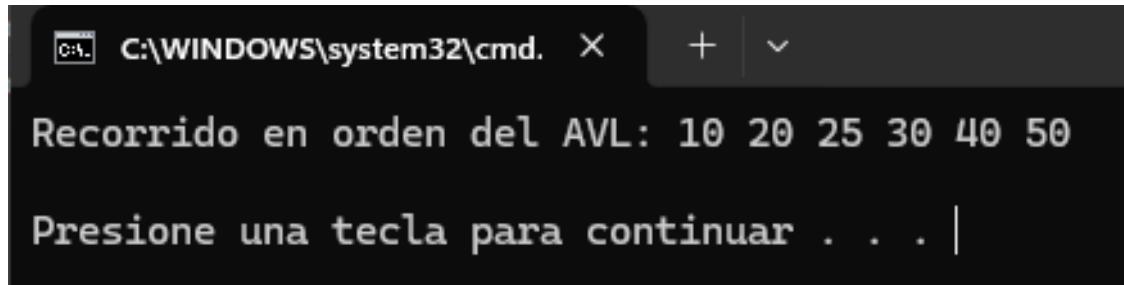
56
57 // -----
58 // Inserci n en el rbol AVL
59 // -----
60 Node* insert(Node* node, int key) {
61     // 1. Inserci n normal de BST
62     if (!node) return new Node(key);
63     if (key < node->key)
64         node->left = insert(node->left, key);
65     else if (key > node->key)
66         node->right = insert(node->right, key);
67     else
68         return node; // Duplicados no permitidos
69     // 2. Actualizar altura del ancestro
70     node->height = 1 + max(height(node->left), height(node->right));
71     // 3. Obtener factor de balance
72     int bal = balance(node);
73     // 4. Casos de desequilibrio y rotaciones
74     // Izquierda - Izquierda
75     if (bal > 1 && key < node->left->key)
76         return rotateRight(node);
77
78     // Derecha - Derecha
79     if (bal < -1 && key > node->right->key)
80         return rotateLeft(node);
81
82     // Izquierda - Derecha
83     if (bal > 1 && key > node->left->key) {
84         node->left = rotateLeft(node->left);
85         return rotateRight(node);
86     }
87     // Derecha - Izquierda
88     if (bal < -1 && key < node->right->key) {
89         node->right = rotateRight(node->right);
90         return rotateLeft(node);
91     }
92     // Ya balanceado
93     return node;

```

```

94  }
95 // -----
96 // Recorrido en orden ( i n order )
97 // -----
98 void inOrder(Node* root) {
99     if (root) {
100         inOrder(root->left);
101         cout << root->key << ' ';
102         inOrder(root->right);
103     }
104 }
105 // -----
106 // Liberar memoria ( p o s t orden )
107 // -----
108 void destroy(Node* root) {
109     if (root) {
110         destroy(root->left);
111         destroy(root->right);
112         delete root;
113     }
114 }
115 // =====
116 //          main()
117 // =====
118 int main() {
119     Node* root = nullptr;
120     // Inserta algunos valores de prueba
121     int valores[] = { 10, 20, 30, 40, 50, 25 };
122     for (int v : valores)
123         root = insert(root, v);
124     cout << "Recorrido en orden del AVL: ";
125     inOrder(root);
126     cout << '\n';
127     destroy(root); // Liberar memoria
128     return 0;
129 }
```

### 13.7.2. Resultado del programa



The screenshot shows a Windows Command Prompt window with the title bar "C:\WINDOWS\system32\cmd.". The window contains the following text:  
Recorrido en orden del AVL: 10 20 25 30 40 50  
Presione una tecla para continuar . . . |

## 13.8. Ventajas y desventajas

- **Ventajas:** operaciones garantizadas en  $O(\log n)$ , evita degeneración, base para otras estructuras.
- **Desventajas:** implementación más compleja, sobrecoste de rotaciones y almacenamiento extra (altura, color, etc.).

## 13.9. Aplicaciones típicas

1. **Bases de datos:** árboles B/B+ para índices en disco.
2. **Bibliotecas estándar C++:** std::map y std::set.
3. **Sistemas de archivos y buscadores:** estructura jerárquica y búsquedas rápidas.
4. **Motores de juegos:** árboles de decisión y particiones espaciales.

## 14. Árboles B y B++

### 14.1. ¿Qué es un Arbol B

Un árbol B es una estructura de datos jerárquica y balanceada diseñada para almacenar grandes cantidades de datos y realizar operaciones como búsqueda, inserción y eliminación de forma eficiente, especialmente cuando los datos están almacenados en memoria secundaria (como discos duros).

Fue diseñado para minimizar el número de accesos al disco, por eso es ampliamente utilizado en:

- Sistemas de gestión de bases de datos
- Sistemas de archivos
- Índices de búsqueda

### 14.2. Características del Arbol B

- Es un árbol de búsqueda balanceado, pero no binario: cada nodo puede tener más de dos hijos.
- Los nodos internos pueden contener varias claves (key) y tener varios punteros a hijos.
- Todos los nodos hoja están al mismo nivel (la altura está controlada).
- Las claves dentro de cada nodo están ordenadas.

### 14.3. Estructura de un nodo B (grado t)

- De 2 a 5 claves
- De 3 a 6 hijos

#### 14.3.1. Ejemplo Visual Simplificado

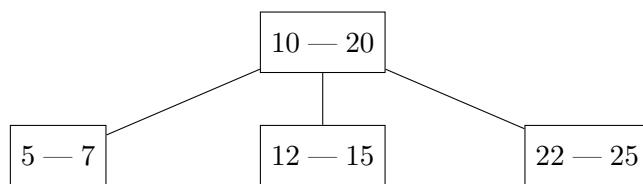


Figura 3: Ejemplo de árbol B de orden 3.

#### 14.4. Programa en C++ de un árbol B

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const int T = 3; // Orden (grado mínimo)
5
6 struct BNode {
7     vector<int> keys; // Claves almacenadas
8     vector<BNode*> child; // Punteros a hijos
9     bool leaf;
10
11     BNode(bool isLeaf) : leaf(isLeaf) {
12         keys.reserve(2 * T - 1);
13         child.resize(2 * T, nullptr);
14     }
15 };
16
17 // -----
18 // Función para recorrer el rbol (in order)
19 // -----
20 void traverse(BNode* node) {
21     if (!node) return;
22
23     int i = 0;
24     for (; i < (int)node->keys.size(); ++i) {
25         // Visitar sub rbol izquierdo antes de imprimir la clave i
26         if (!node->leaf)
27             traverse(node->child[i]);
28         cout << node->keys[i] << " ";
29     }
30     // ltimo hijo
31     if (!node->leaf)
32         traverse(node->child[i]);
33 }
34
35 // -----
36 // Divide el hijo completo y sube la clave media
37 // -----
```

```

38 void splitChild(BNode* parent, int idx, BNode* fullChild) {
39     auto *newChild = new BNode(fullChild->leaf);
40
41     // Copiar las últimas T-1 claves al nuevo hijo
42     for (int j = 0; j < T - 1; ++j)
43         newChild->keys.push_back(fullChild->keys[j + T]);
44
45     // Copiar los hijos correspondientes
46     if (!fullChild->leaf) {
47         for (int j = 0; j < T; ++j)
48             newChild->child[j] = fullChild->child[j + T];
49     }
50
51     // Reducir el tamaño del hijo original
52     fullChild->keys.resize(T - 1);
53
54     // Insertar el nuevo hijo en el padre
55     parent->child.insert(parent->child.begin() + idx + 1, newChild);
56     parent->child.pop_back(); // mantener tamaño 2T
57
58     // Insertar la clave media en el padre
59     parent->keys.insert(parent->keys.begin() + idx, fullChild->keys[T - 1]);
60 }
61
62 //-----
63 // Inserta una clave en un nodo que NO esté lleno
64 //-----
65 void insertNonFull(BNode* node, int k) {
66     int i = node->keys.size() - 1;
67
68     if (node->leaf) {
69         // Insertar la nueva clave manteniendo orden
70         node->keys.push_back(0);
71         while (i >= 0 && k < node->keys[i]) {
72             node->keys[i + 1] = node->keys[i];
73             --i;
74         }

```

```

75     node->keys[i + 1] = k;
76 } else {
77     // Encontrar el hijo apropiado
78     while (i >= 0 && k < node->keys[i]) --i;
79     ++i;
80
81     // Si el hijo est lleno, dividirlo
82     if (node->child[i]->keys.size() == 2 * T - 1) {
83         splitChild(node, i, node->child[i]);
84         if (k > node->keys[i]) ++i;
85     }
86     insertNonFull(node->child[i], k);
87 }
88 }
89
90 // -----
91 // Inserci n p blica: maneja divisi n de la ra z
92 // -----
93 void insert(BNode*& root, int k) {
94     if (!root) {
95         root = new BNode(true);
96         root->keys.push_back(k);
97         return;
98     }
99     // Si la ra z est llena, crear nueva ra z
100    if (root->keys.size() == 2 * T - 1) {
101        auto *s = new BNode(false);
102        s->child[0] = root;
103        splitChild(s, 0, root);
104        int i = (k > s->keys[0]) ? 1 : 0;
105        insertNonFull(s->child[i], k);
106        root = s;
107    } else {
108        insertNonFull(root, k);
109    }
110 }
111
112 // -----

```

```

113 // Programa de demostraci n
114 //-----
115 int main() {
116     BNode* root = nullptr;
117     vector<int> valores = {10, 20, 5, 6, 12, 30, 7, 17, 15, 22, 25};
118
119     for (int v : valores)
120         insert(root, v);
121
122     cout << "Recorrido en orden del rbol B: ";
123     traverse(root);
124     cout << endl;
125
126     return 0;
}

```

## Resultado

Listing 1: Salida del programa en C++

Recorrido en orden del rbol B: 5 6 7 10 12 15 17 20 22 25 30

## 14.5. Operaciones Básicas

### 14.5.1. Búsqueda

- Similar a una búsqueda binaria, pero dentro de un nodo con múltiples claves.
- Muy eficiente en árboles de gran tamaño porque reduce la altura del árbol.

### 14.5.2. Inserción

- Se agrega la clave en el nodo correspondiente
- Si se sobrepasa el número máximo de claves, el nodo se divide y la clave del medio sube.

### 14.5.3. Eliminación

- Se realiza reorganizando claves entre nodos vecinos o combinando nodos si hay pocas claves.

## **14.6. ¿Qué es un arbol B++**

El árbol B+ es una estructura de datos jerárquica y balanceada que se utiliza principalmente para organizar grandes cantidades de datos en sistemas de archivos y bases de datos.

Es una variación del árbol B, diseñada para ser más eficiente en búsquedas secuenciales y por rangos, así como en sistemas que utilizan almacenamiento externo (discos duros o SSDs).

## **14.7. Diferencia entre b y B+**

- **¿Dónde se almacenan los datos?**

En el Árbol B, los datos pueden estar tanto en los nodos internos como en las hojas.

En cambio, en el Árbol B+, todos los datos reales se almacenan únicamente en los nodos hoja. Los nodos internos solo contienen claves de navegación.

- **Contenido de los nodos internos**

En el Árbol B, los nodos internos pueden contener tanto claves como datos. En el Árbol B+, los nodos internos contienen exclusivamente claves para guiar la búsqueda.

- **Enlace entre hojas**

En el arból B, no necesariamente hay conexión entre los nodos hoja, En el árbol B+, las hojas están enlazadas entre sí (en una lista ordenada), lo que facilita el recorrido secuencial.

- **Eficiencia en recorrido secuencial**

El arbol B no esta optimizado para recorridos ordenados. En cambio, el árbol B+ permite recorridos mucho más eficientes gracias a que sus hojas están conectadas en orden.

- **Uso recomendado**

El árbol b es más adecuado para búsquedas por clave exacta. El árbol B+ es ideal cuando se requiere realizar búsquedas por rangos o recorridos ordenados, como en bases de datos e índices.

## 14.8. Estructura del Árbol B+

### 1. Nodos Internos:

- Contienen solo claves para guiar la búsqueda.
- No contienen los datos reales.

### 2. Nodos Hoja:

- Contienen todos los datos reales (claves + registros).
- Están enlazados entre sí como una lista (generalmente una lista doble).

## 14.9. ¿Cómo funciona

El árbol B+ divide su estructura en dos capas:

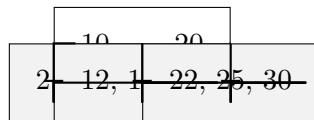
### 1. Navegación por el árbol (nodos internos):

- Se utiliza para buscar la hoja correcta.
- Es eficiente se comporta como un árbol de búsqueda.

### 2. Acceso a los datos (nodos hoja):

- Una vez llegas a la hoja adecuada, puedes **leer el dato exacto o continuar a la siguiente hoja** sin necesidad de volver a subir en el árbol.
- Esto es ideal para consultas como "tráeme todos los valores entre 50 y 100".

#### **14.10. Ejemplo visual (simplificado)**



#### **14.11. Características clave del árbol B**

- Es un árbol balanceado: todas las hojas están al mismo nivel.
- Reduce la altura del árbol menor número de accesos a disco.
- Permite búsquedas rápidas y eficientes, tanto exactas como por rango.
- Los recorridos secuenciales son más rápidos que en un árbol B.
- Muy usado en motores de bases de datos e índices.

#### **14.12. Ventajas del árbol B+**

- Alto rendimiento en lectura de grandes volúmenes de datos
- Ideal para almacenamiento en disco
- Búsquedas por rango muy eficientes
- Hojas enlazadas para facilitar recorrido ordenado
- Los nodos internos solo se usan para navegación (más ligeros)

#### **14.13. Desventajas**

- Estructura más compleja que árboles binarios o AVL
- Insertar y eliminar puede requerir redistribuir nodos y rebalancear
- Requiere mantener punteros entre hojas

## 14.14. Aplicaciones del Árbol B+

- Motores de bases de datos: MySQL, PostgreSQL, Oracle
- Sistemas de archivos: NTFS, HFS+, ext4
- Índices en sistemas de búsqueda
- Dispositivos de almacenamiento (por bloques)

## 14.15. Programa en C++

```
1 #include <bits/stdc++.h>
2 using namespace std;
3
4 const int T = 3; // Orden mínimo (t)
5
6 struct Node {
7     bool isLeaf;
8     vector<int> keys;
9     vector<Node*> child; // Hijos (internos) o punteros a
10    hojas
11    Node* next; // Solo para hojas: siguiente hoja
12
13    Node(bool leaf) : isLeaf(leaf), next(nullptr) {
14        keys.reserve(2 * T - 1);
15        child.resize(2 * T, nullptr);
16    }
17
18 //----- Utilidades -----
19 void traverseLeaves(Node* root) {
20     // Ir a la hoja más a la izquierda
21     while (root && !root->isLeaf)
22         root = root->child[0];
23
24     // Recorrer hojas usando 'next'
25     while (root) {
26         for (int k : root->keys) cout << k << ' ';
27         root = root->next;
```

```

28     }
29     cout << '\n';
30 }
31
32 //----- Divisi n de un hijo -----
33 void splitChild(Node* parent, int idx, Node* y) {
34     auto* z = new Node(y->isLeaf);
35     // Transferir ltimas T-1 claves a z
36     z->keys.assign(y->keys.begin() + T, y->keys.end());
37     y->keys.resize(T);
38
39     if (y->isLeaf) {
40         // Hojas: copiar puntero 'next'
41         z->next = y->next;
42         y->next = z;
43     } else {
44         // Internos: mover T hijos
45         z->child.assign(y->child.begin() + T, y->child.end());
46         y->child.resize(T);
47     }
48
49     // Insertar z en el padre
50     parent->child.insert(parent->child.begin() + idx + 1, z);
51     parent->child.pop_back(); // mantener tama o 2T
52     // Subir la clave m nima de z
53     parent->keys.insert(parent->keys.begin() + idx, z->keys.front())
54     ;
55 }
56 //----- Inserci n en nodo no lleno -----
57 void insertNonFull(Node* node, int k) {
58     if (node->isLeaf) {
59         // Insertar ordenadamente en la hoja
60         node->keys.insert(lower_bound(node->keys.begin(), node->keys
61             .end(), k), k);
62     } else {
63         // Elegir hijo apropiado
64         int i = upper_bound(node->keys.begin(), node->keys.end(), k)

```

```

        - node->keys.begin();

64    if (node->child[i]->keys.size() == 2 * T - 1) {
65        splitChild(node, i, node->child[i]);
66        if (k >= node->keys[i]) ++i;
67    }
68    insertNonFull(node->child[i], k);
69}
70}

71 //----- Inserci n p blica -----
72 void insert(Node*& root, int k) {
73    if (!root) {
74        root = new Node(true);
75        root->keys.push_back(k);
76        return;
77    }
78    if (root->keys.size() == 2 * T - 1) {
79        auto* s = new Node(false);
80        s->child[0] = root;
81        splitChild(s, 0, root);
82        int i = (k >= s->keys[0]) ? 1 : 0;
83        insertNonFull(s->child[i], k);
84        root = s;
85    } else {
86        insertNonFull(root, k);
87    }
88}
89}

90 //----- Demo -----
91 int main() {
92    Node* root = nullptr;
93    vector<int> valores = {10, 20, 5, 6, 12, 30, 7, 17, 15, 22, 25,
94                           2, 9, 18};
95
96    for (int v : valores) insert(root, v);
97
98    cout << "Recorrido de hojas (ordenado): ";
99    traverseLeaves(root); // Muestra: 2 5 6 7 9 10 12 15 17 18 20

```

```
22 25 30
100     return 0;
101 }
```

## 15. Arboles Heap

### 15.1. ¿Qué es un Arbol Heap

Un árbol Heap es una estructura de datos completa y parcialmente ordenada, que se utiliza principalmente para implementar colas de prioridad. Se trata de un árbol binario completo, donde cada nodo tiene una relación específica con sus hijos. Existen dos tipos principales:

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

### 15.2. Características de un Arbol Heap

- Es un árbol binario completo: todos los niveles están completamente llenos excepto posiblemente el último, que está lleno de izquierda a derecha.
- Se representa eficientemente como un arreglo (array), no como una estructura enlazada.
- Se usa principalmente en colas de prioridad, algoritmos como heapsort, y en la implementación de estructuras tipo scheduler o administración de tareas.

### 15.3. Tipos de Heap

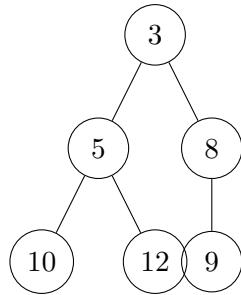
#### 15.3.1. Max-Heap

- Cada nodo es mayor o igual que sus hijos.
- El elemento más grande está en la raíz.
- Ideal para obtener rápidamente el máximo valor.

#### 15.3.2. Min-Heap

- Cada nodo es menor o igual que sus hijos.
- El elemento más pequeño está en la raíz.
- Ideal para obtener rápidamente el mínimo valor.

#### 15.4. Ejemplo de un código visual de Min-Heap



#### 15.5. Representación con Arreglo

Un arbol Heap puede representar con un array. Por ejemplo, el Min-Heap anterior se representa como: **Índices: 0 1 2 3 4 5**

**VAlores: 3 5 8 10 12 9**

Relaciones:

- Para un nodo en índice **i**
  - Hijo izquierdo **2i + 1**
  - Hijo derecho **2i + 2**
  - Padre **(i - 1) / 2**

#### 15.6. Aplicaciones del Árbol Heap

- Colas de prioridad (priority queues)
- Heapsort (algoritmo de ordenamiento)
- Planificadores de procesos (en sistemas operativos)
- K mayores/menores elementos de un conjunto
- Grafos: Algoritmos como Dijkstra (usando Min-Heap)

#### 15.7. Ventajas

- Eficiencia: operaciones clave como inserción y extracción son logarítmicas.
- Muy útil cuando necesitas obtener continuamente el mínimo o el máximo.

## 15.8. Desventajas

- No es tan rápido para búsquedas arbitrarias (no está completamente ordenado).
- Mantener el orden parcial requiere operaciones de "flotar." "hundir." elementos.

## 15.9. Programa en C++

```
1      #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  class MinHeap {
6  private:
7      vector<int> heap;
8
9      // Devuelve índice del padre
10     int parent(int i) { return (i - 1) / 2; }
11
12     // Devuelve índice de los hijos
13     int left(int i) { return 2 * i + 1; }
14     int right(int i) { return 2 * i + 2; }
15
16     // Sube el valor si es menor que su padre
17     void heapifyUp(int i) {
18         while (i > 0 && heap[i] < heap[parent(i)]) {
19             swap(heap[i], heap[parent(i)]);
20             i = parent(i);
21         }
22     }
23
24     // Baja el valor si es mayor que alguno de sus hijos
25     void heapifyDown(int i) {
26         int smallest = i;
27         int l = left(i), r = right(i);
28
29         if (l < heap.size() && heap[l] < heap[smallest])
30             smallest = l;
31         if (r < heap.size() && heap[r] < heap[smallest])
```

```

32         smallest = r;
33
34     if (smallest != i) {
35         swap(heap[i], heap[smallest]);
36         heapifyDown(smallest);
37     }
38 }
39
40 public:
41     void insert(int key) {
42         heap.push_back(key);
43         heapifyUp(heap.size() - 1);
44     }
45
46     int getMin() {
47         if (!heap.empty())
48             return heap[0];
49         throw out_of_range("Heap vac o");
50     }
51
52     int extractMin() {
53         if (heap.empty())
54             throw out_of_range("Heap vac o");
55
56         int minValue = heap[0];
57         heap[0] = heap.back();
58         heap.pop_back();
59         heapifyDown(0);
60         return minValue;
61     }
62
63     void printHeap() {
64         for (int val : heap)
65             cout << val << " ";
66         cout << endl;
67     }
68 };
69

```

```

70 //----- MAIN -----
71 int main() {
72     MinHeap h;
73     h.insert(10);
74     h.insert(5);
75     h.insert(12);
76     h.insert(3);
77     h.insert(9);

78     cout << "Heap actual: ";
79     h.printHeap();
80
81     cout << "Elemento m nimo: " << h.getMin() << endl;
82
83     cout << "Extraer m nimo: " << h.extractMin() << endl;
84
85     cout << "Heap despues de extraer m nimo: ";
86     h.printHeap();
87
88     return 0;
89 }

```

### 15.9.1. Resultado

```

C:\WINDOWS\system32\cmd. x + v
Heap actual: 3 5 12 10 9
Elemento minimo: 3
Extraer minimo: 3
Heap despues de extraer m nimo: 5 9 12 10

Presione una tecla para continuar . . .

```

## 16. Árbol Rojo y Negro

Un árbol Rojo-Negro es un tipo de árbol binario de búsqueda (ABB) auto-balanceado que garantiza un tiempo logarítmico ( $O(\log n)$ ) para operaciones como inserción, eliminación y búsqueda.

Se llama así porque cada nodo tiene un atributo de color: rojo o negro, lo cual ayuda a mantener el árbol equilibrado.

### 16.1. Propiedades fundamentales

Para que un árbol Rojo-Negro funcione correctamente, debe cumplir 5 reglas estrictas:

1. **Cada nodo es rojo o negro.**
2. **La raíz siempre es negra**
3. **Todas las hojas (nodos nulos) se consideran negras.**
4. **Si un nodo es rojo, sus hilos deben ser negros.**
  - Es decir, no puede haber dos nodos rojos consecutivos.
5. **Todo camino desde un nodo hasta sus hojas descendientes nulas contiene el mismo número de nodos negros.**
  - A esto se le llama **.altura negra**.

### 16.2. Operaciones en Árbol Rojo-Negro:

#### 1. Inserción:

- Similar a la inserción en un árbol binario de búsqueda.
- El nodo se inserta como **rojo**.
- Luego, se realiza un proceso de **reparación** para mantener las reglas, lo cual puede requerir:
  - **Cambio de colores**
  - **Rotaciones** (izquierda o derecha)

#### 2. Eliminación

- Más compleja que la inserción.

- Si se elimina un nodo negro, se puede romper el balance de color.
- Se requieren múltiples pasos de **ajuste**, incluyendo rotaciones y recoloreos para restaurar las propiedades del árbol.

### 3. Búsqueda

- Igual que un árbol binario de búsqueda.
- Comparación de claves hasta encontrar el nodo deseado.

### 16.3. Rotaciones (Balanceo)

Son operaciones clave para restaurar las reglas después de insertar o eliminar nodos:

- **Rotación a la izquierda:** Cambia la estructura local del árbol cuando el hijo derecho está "desbalanceado".
- **Rotación a la derecha:** Se aplica cuando el hijo izquierdo está desbalanceado.
- En algunos casos se realizan rotaciones dobles (izquierda-derecha o derecha-izquierda).

### 16.4. Ventajas de los Árboles Rojo-Negro

1. **Mantienen el árbol balanceado automáticamente** Gracias a sus reglas de color y altura negra, el árbol se mantiene equilibrado sin necesidad de un rebalanceo constante como en otros árboles (por ejemplo, AVL).
2. **Operaciones eficientes ( $O(\log n)$ )** Las operaciones de búsqueda, inserción y eliminación siempre tienen un tiempo logarítmico en el peor caso, lo que lo hace ideal para estructuras dinámicas con muchos datos.
3. **Evitan el desequilibrio extremo** Las reglas del árbol garantizan que la altura máxima del árbol esté limitada a aproximadamente el doble del logaritmo del número de nodos, evitando caminos muy largos.
4. **Requiere menos rotaciones que un árbol AVL** Aunque no está tan estrictamente balanceado como un AVL, en la práctica necesita menos rotaciones, lo que puede ser más eficiente en ciertas aplicaciones.
5. **Estabilidad en rendimiento** A diferencia de otros árboles como el binario de búsqueda simple, el árbol Rojo-Negro nunca degenera en una lista (como puede ocurrir si los datos llegan ordenados).

**6. Muy utilizado en implementaciones estándar** Es la estructura base de contenedores como:

- std::map y std::set en C++
- TreeMap y TreeSet en Java
- Algunos sistemas de archivos y bases de datos

**7. Buen rendimiento en inserciones/eliminaciones frecuentes** Es especialmente útil cuando se necesita modificar constantemente la estructura, ya que mantiene el equilibrio sin grandes costos.

### 16.5. Desventajas

- Implementación más compleja que otros árboles como Heap o simples ABB.
- Requiere manejar múltiples casos en inserción/eliminación debido a las reglas de color.
- Menos estrictamente balanceado que un árbol AVL (lo cual puede ser una ventaja o desventaja, según el caso).

### 16.6. Aplicaciones Comunes

- Implementación de contenedores ordenados en STL (std::map, std::set en C++).
- Sistemas de bases de datos y archivos.
- Programas de planificación de tareas.
- Compiladores, sistemas de red y sistemas operativos (como el completely fair scheduler en Linux).