

Estructura de Datos en C++

Guía para estudiantes

Autor: Flores Aycaya Blanca Beatriz

Año: 2025



Índice general

1. Introducción a la Programación en C++	9
1.1. ¿Qué es C++?	9
1.2. Nuestro primer programa	10
1.3. Entrada y salida de datos	11
1.4. Definición de variables y tipos de datos	11
1.5. Operadores en C++	12
2. Arrays	13
2.1. ¿Qué es un array?	13
2.2. Sintaxis básica	13
2.3. Acceso a los elementos del array	14
2.4. Inicialización de un array	14
2.5. Recorrer un array con bucles	14
2.6. Ejemplo completo: promedio de notas	14
2.7. Actividad sugerida	15
2.8. Resumen visual	16
3. Pilas	17
3.1. ¿Qué es una Pila?	17
3.2. Operaciones básicas de una Pila	18
3.3. Implementación de una Pila con Arreglos	18
3.4. Ejemplo: Uso de la Pila	20
3.5. Implementación de una Pila con Listas Enlazadas	20
3.6. Ejemplo: Uso de la Pila con Listas Enlazadas	22
3.7. Aplicaciones de las Pilas	22
3.7.1. Ejemplo: Evaluación de Expresiones	22
3.8. Actividad sugerida	23
3.9. Resumen visual	24
4. Listas Enlazadas	25
4.1. ¿Qué es una lista enlazada?	25
4.2. Estructura de un Nodo	25
4.3. ¿Por qué usar listas enlazadas?	26
4.4. Tipos de Listas Enlazadas	26
4.5. Implementación de una Lista Enlazada Simple	26
4.6. Ejemplo: Uso de una Lista Enlazada	27
4.7. Lista Dblemente Enlazada	28
4.8. Ejemplo: Uso de la Lista Dblemente Enlazada	30

4.9. Lista Circular	30
4.10. Actividad sugerida	31
4.11. Resumen visual	31
5. Listas Dblemente Enlazadas y Circulares	33
5.1. ¿Qué es una lista doblemente enlazada?	33
5.2. Ventajas de las listas doblemente enlazadas	33
5.3. Estructura de un Nodo en una Lista Doblemente Enlazada	34
5.4. Implementación de una Lista Doblemente Enlazada	34
5.5. Ejemplo: Uso de una Lista Doblemente Enlazada	36
5.6. ¿Qué es una lista circular?	36
5.7. Implementación de una Lista Circular	37
5.8. Ejemplo: Uso de la Lista Circular	39
5.9. Aplicaciones de las Listas Doblemente Enlazadas y Circulares	39
5.10. Actividad sugerida	39
5.11. Resumen visual	40
6. Colas	41
6.1. ¿Qué es una Cola?	41
6.2. Operaciones Básicas de una Cola	41
6.3. Implementación de una Cola con Arreglos	42
6.4. Ejemplo: Uso de una Cola	44
6.5. Cola Circular	44
6.6. Implementación de una Cola Circular	45
6.7. Ejemplo: Uso de una Cola Circular	47
6.8. Ejemplos prácticos de uso de Colas	47
6.8.1. 1. Simulación de Atención al Cliente	47
6.9. 2. Cola para Procesamiento de Tareas en un Sistema Operativo	49
6.9.1. 3. Cola de Impresión en una Oficina	51
7. Pilas	53
7.1. ¿Qué es una Pila?	53
7.2. Operaciones Básicas de una Pila	53
7.3. Implementación de una Pila con Arreglos	54
7.4. Ejemplo: Uso de una Pila	56
7.5. Aplicaciones Comunes de las Pilas	56
7.5.1. 1. Evaluación de Expresiones Aritméticas	56
7.5.2. 2. Deshacer y Rehacer Operaciones en un Editor de Texto	57
7.5.3. 3. Gestión de Funciones en un Programa	59
7.6. Resumen del Capítulo	60
8. Recursividad	61
8.1. ¿Qué es la Recursividad?	61
8.2. ¿Por qué es importante la Recursividad?	62
8.3. Funcionamiento Interno de la Recursividad	63
8.4. Ejemplo clásico: Factorial	63
8.5. Ejemplo 2: Serie de Fibonacci	63
8.6. Ejemplo 3: Búsqueda Binaria Recursiva	64
8.7. Ejemplos prácticos de uso de la Recursividad	65

8.7.1. 1. Caminos en un Laberinto	65
8.7.2. 2. Torres de Hanoi	66
8.7.3. 3. Búsqueda en un Árbol Binario	67
8.8. Resumen del Capítulo	68
9. Algoritmos de Ordenación	69
9.1. ¿Qué es un Algoritmo de Ordenación?	69
9.2. Algoritmos de Ordenación Comunes	69
9.3. Algoritmo de Ordenación por Burbuja (Bubble Sort)	70
9.4. Ejemplo: Ordenación por Burbuja	70
9.5. Algoritmo de Ordenación por Selección (Selection Sort)	71
9.6. Ejemplo: Ordenación por Selección	72
9.7. Algoritmo de Ordenación por Inserción (Insertion Sort)	73
9.8. Ejemplo: Ordenación por Inserción	73
9.9. Algoritmo de Ordenación Rápida (Quick Sort)	74
9.10. Ejemplo: Ordenación Rápida	75
9.11. Resumen del Capítulo	76
10. Unidad 2: Programación Competitiva en C++	77
10.1. Introducción a la Programación Competitiva	77
10.2. ¿Qué es la Programación Competitiva?	77
10.3. Plataformas para Programación Competitiva	78
10.4. Conceptos Clave en Programación Competitiva	78
10.4.1. Algoritmos de Búsqueda y Ordenación	78
10.4.2. Estructuras de Datos	79
10.5. Resolución de Problemas Clásicos	80
10.5.1. Problema: Suma de Subconjuntos	80
10.5.2. Problema: Encuentra el Camino Más Corto en un Grafo	81
10.6. Conclusión	82
11. Ejercicios de Pilas y Colas	83
11.1. Ejercicio 1: Implementación de Pila y Cola	83
11.1.1. Estructuras de Pila y Cola en C++	83
11.2. Ejercicio 2: Transferencia de Datos entre Pila y Cola	85
11.2.1. Código de Transferencia de Pila a Cola y de Cola a Pila	86
11.3. Ejercicio 3: Intercambiar Pila y Cola	87
11.4. Resumen del Capítulo	89
12. Árbol Binario Simple	91
12.1. ¿Qué es un Árbol Binario Simple?	91
12.2. Estructura de un Árbol Binario Simple en C++	91
12.3. Operaciones Básicas en un Árbol Binario	92
12.4. Ejemplo de Inserción en un Árbol Binario	92
12.5. Recorridos de un Árbol Binario	93
12.6. Búsqueda en un Árbol Binario	94
12.7. Eliminación de un Nodo en un Árbol Binario	95

13. Árboles Balanceados	99
13.1. ¿Qué es un Árbol Balanceado?	99
13.2. Propiedades de los Árboles Balanceados	99
13.3. Árboles AVL	99
13.4. Rotaciones en Árboles AVL	100
14. Árboles B / B+	105
14.1. Definición de Árboles B	105
14.1.1. Características de los Árboles B	105
14.1.2. Reglas de los Árboles B	105
14.1.3. Operaciones en Árboles B	105
14.1.4. Ejemplo de Inserción en un Árbol B	106
14.1.5. Gráfico de un Árbol B de Orden 3	106
14.2. Árboles B+	106
14.2.1. Características de los Árboles B+	106
14.2.2. Gráfico de un Árbol B+	106
14.3. Operaciones en Árboles B+	106
14.4. Código C++ para Implementación de un Árbol B	107
15. Árboles Heap	109
15.1. Definición y Concepto	109
15.2. Propiedades Importantes	109
15.3. Operaciones en Árboles Heap	109
15.3.1. Inserción	110
15.3.2. Eliminación (Eliminación del Raíz)	110
15.3.3. Heapificación	110
15.4. Representación de un Árbol Heap	110
15.5. Ejemplo de Árbol Heap	111
15.6. Árbol Heap en la Práctica	111
15.7. Implementación de un Árbol Heap en C++	111
16. Árboles Rojo-Negro	113
16.1. Definición y Concepto	113
16.2. Operaciones en Árboles Rojo-Negro	113
16.2.1. Inserción	113
16.2.2. Eliminación	114
16.2.3. Búsqueda	114
16.3. Rotaciones en Árboles Rojo-Negro	114
16.3.1. Rotación a la Izquierda	114
16.3.2. Rotación a la Derecha	115
16.4. Implementación de un Árbol Rojo-Negro en C++	115
Glosario	121
16.5. Términos en Español	121
16.6. Terms in English	121

Solucionario	123
16.7. Ejercicios del Capítulo 1: Introducción a la Programación en C++	123
16.8. Ejercicios del Capítulo 2: Arrays	124
16.9. Ejercicios del Capítulo 3: Pilas	125
16.10Ejercicios del Capítulo 4: Colas	126
Glosario de la Segunda Unidad	131

Introducción a la Programación en C++

“Programar es decirle a la computadora lo que quieras, de forma que ella entienda.”

¿Qué es programar?

Programar es darle instrucciones a una computadora para que realice tareas. Es como enseñarle paso a paso cómo resolver un problema. La programación es esencial en casi todos los dispositivos que usamos hoy en día, desde teléfonos hasta computadoras y electrodomésticos inteligentes.

Definición: La programación es el proceso mediante el cual se crean aplicaciones, software y sistemas operativos que permiten la interacción con dispositivos tecnológicos.



Figura 1.1:

1.1. ¿Qué es C++?

C++ es un lenguaje de programación muy poderoso y utilizado en todo el mundo. Fue creado para ser rápido, flexible y permitir crear desde videojuegos hasta sistemas

complejos.

¿Por qué aprender C++?

- Tiene una sintaxis clara y lógica.
- Se usa en robótica, videojuegos, apps y sistemas operativos.
- Te prepara para aprender otros lenguajes con facilidad.
- Es uno de los lenguajes más rápidos debido a su cercanía con el hardware.

Historia de C++

C++ fue creado por Bjarne Stroustrup en 1979 en los laboratorios Bell de ATT. Se basó en el lenguaje C y añadió características de programación orientada a objetos, lo que le permitió manejar más fácilmente la complejidad de programas grandes.

1.2. Nuestro primer programa

Ejemplo: Hola Mundo

Este es el programa más clásico para comenzar:

Ejemplo: Primer programa en C++

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

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

Explicación del código

Este programa hace lo siguiente:

- `#include <iostream>` – Incluye la librería estándar de entrada y salida.
- `int main()` – Es la función principal, donde comienza la ejecución del programa.
- `cout` – Imprime el texto "Hola mundo!" en la pantalla.
- `return 0;` – Termina el programa con un valor 0, indicando que se ejecutó correctamente.

1.3. Entrada y salida de datos

Una de las primeras cosas que aprenderás en cualquier lenguaje de programación es cómo recibir datos de los usuarios y cómo mostrar resultados.

Imprimir datos con cout

Ejemplo: Mostrar texto

```
cout << " Bienvenidos - a - C++!" ;
```

Ler datos con cin

El comando `cin` permite recibir datos del usuario. A continuación, mostramos cómo leer el nombre de un usuario:

Ejemplo: Leer el nombre del usuario

```
string nombre;
cout << " Cmo - te - llamas? - ";
cin >> nombre;
cout << " Hola , - " << nombre;
```

1.4. Definición de variables y tipos de datos

En C++, las variables son fundamentales para almacenar datos. Cada variable tiene un tipo, que determina qué tipo de datos puede almacenar.

¿Qué es una variable?

Una variable es como una caja donde guardamos un dato. Cada variable tiene un **nombre**, un **tipo de dato** y un **valor**.

```
int      = 15
```

Esto significa que estamos guardando el número 15 en una variable llamada `edad`, de tipo `int` (entero).

Tipos de Datos en C++

En C++, algunos de los tipos de datos más comunes son:

Tipo	Descripción	Ejemplo
int	Números enteros	int edad = 15;
float	Números decimales	float altura = 1.65;
char	Carácter único	char letra = 'A';
bool	Verdadero o falso	bool aprobado = true;
string	Texto (requiere #include <string>)	string nombre = "Ana";

1.5. Operadores en C++

Los operadores permiten realizar cálculos y manipulaciones sobre las variables.

Operadores Aritméticos

Operador	Función	Ejemplo
+	Suma	a + b
-	Resta	a - b
*	Multiplicación	a * b
/	División	a / b
%	Módulo (residuo)	a % b

Ejemplo: Operaciones Aritméticas

```
"cpp int suma = 10 + 5; // 15 int residuo = 7
```

Lo que aprendiste

- Qué es una variable
- Tipos de datos más comunes
- Declaración e inicialización
- Uso de operadores aritméticos
- Cómo hacer un programa simple con datos personales

Ahora estás listo para trabajar con arreglos (arrays) en el siguiente capítulo!

Arrays

2.1. ¿Qué es un array?

Imagina que tienes que guardar las notas de 30 estudiantes. Podrías crear 30 variables como nota1, nota2, nota3, ..., nota30, pero eso sería una locura. En lugar de eso, podemos usar un **array**, que es como una fila de casillas donde puedes guardar muchos datos del mismo tipo bajo un solo nombre.

Un **array** es una estructura que permite almacenar varios datos del mismo tipo en una sola variable. Cada dato se guarda en una posición llamada **índice**, que empieza desde cero.

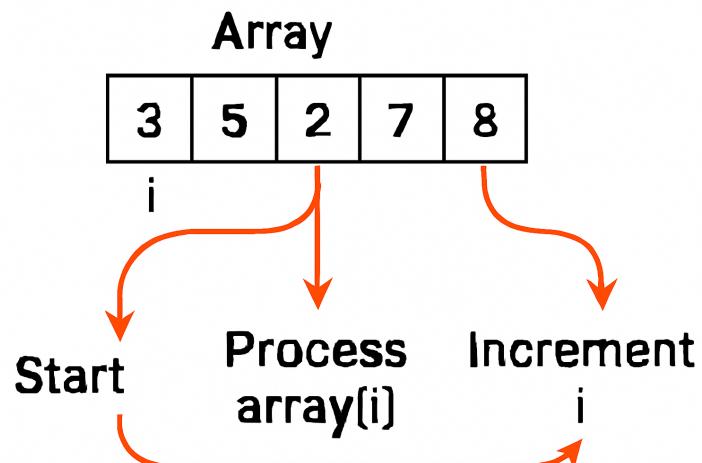


Imagen: Representación gráfica de un array.

2.2. Sintaxis básica

La sintaxis para declarar un array es la siguiente: “`cpp tipo nombreDelArray[tamaño];`

Ejemplo: Declaración de un array

```
int notas [5]; // Crea un array de 5 enteros
```

Este código crea un array de 5 elementos de tipo `int`. Los elementos del array se numeran desde `notas[0]` hasta `notas[4]`.

2.3. Acceso a los elementos del array

Para acceder a los elementos de un array, simplemente usas el índice correspondiente. Recuerda que el índice comienza desde 0.

```
notas[0] = 90; // Asigna 90 a la primera posición
notas[1] = 85; // Asigna 85 a la segunda posición
```

Ejemplo: Acceso a los elementos de un array

```
int notas [5] = {90, 85, 78, 92, 88};
cout << notas [0]; // Imprime 90
```

2.4. Inicialización de un array

También puedes llenar un array desde el principio al declarar sus elementos. `int edades[4] = 12, 15, 17, 14;` Esto guarda 12 en `edades[0]`, 15 en `edades[1]`, y así sucesivamente.

Ejemplo: Inicialización de un array

```
int edades [4] = {12, 15, 17, 14};
```

2.5. Recorrer un array con bucles

Un ejemplo muy común de trabajo con arrays es recorrerlo para realizar alguna operación con cada uno de sus elementos.

Ejemplo: Recorrer un array con bucles

```
int numeros [3] = {5, 8, 12};

for (int i = 0; i < 3; i++) {
    cout << "Elemento -" << i << ":" ->< numeros [i] << endl;
}
```

Este código recorre el array `numeros` e imprime todos sus elementos. La salida será:
Elemento 0: 5 Elemento 1: 8 Elemento 2: 12

2.6. Ejemplo completo: promedio de notas

Supongamos que tenemos un array con las notas de 5 estudiantes y queremos calcular el promedio.

Ejemplo: Promedio de notas

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

int main() {
    int notas[5];
    int suma = 0;

    for(int i = 0; i < 5; i++) {
        cout << "Ingrese la nota #" << (i+1) << ": ";
        cin >> notas[i];
        suma += notas[i];
    }

    float promedio = suma / 5.0;
    cout << "Promedio: " << promedio << endl;

    return 0;
}
```

Este programa pedirá al usuario que ingrese 5 notas y calculará el promedio de esas notas.

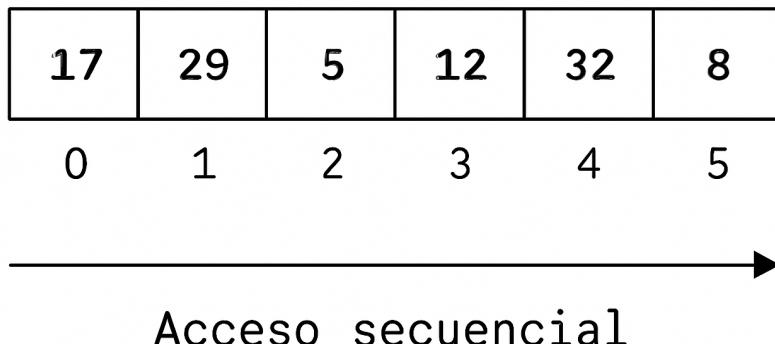
2.7. Actividad sugerida

Actividad

Declara un array de 10 enteros y pídele al usuario que ingrese valores. Luego, imprime todos los valores al revés.

2.8. Resumen visual

Recorriendo un array en C++



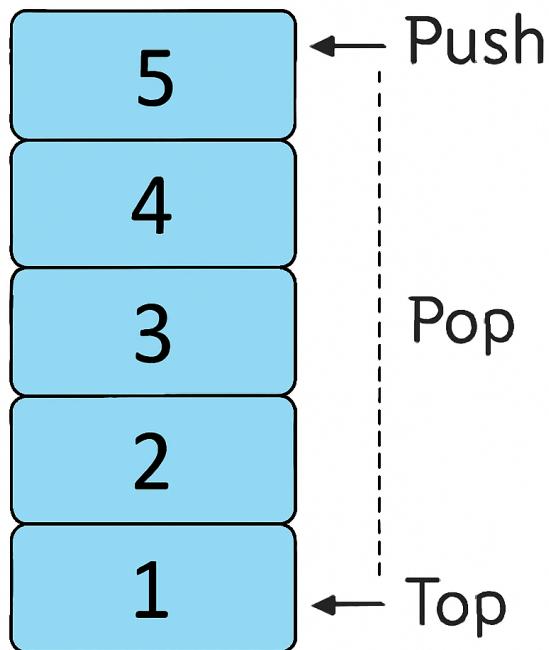
Resumen del capítulo

- Un array guarda varios valores del mismo tipo.
- Cada valor se accede por un índice que comienza en 0.
- Podemos usar bucles para llenar o recorrer un array.
- Son muy útiles cuando trabajamos con listas de datos del mismo tipo.

Pilas

3.1. ¿Qué es una Pila?

Una **pila** (**stack**) es una estructura de datos que sigue el principio **LIFO** (Last In, First Out), lo que significa que el último elemento en ser agregado es el primero en ser retirado. Es similar a una pila de libros: el último libro que pongas en la pila será el primero en ser retirado.



Stack

Imagen: Representación gráfica de una pila.

3.2. Operaciones básicas de una Pila

Las operaciones más comunes en una pila son:

- **push:** Inserta un nuevo elemento en la cima de la pila.

- **pop:** Elimina el elemento que está en la cima de la pila.

- **top:** Muestra el valor del elemento en la cima sin eliminarlo.

- **empty:** Verifica si la pila está vacía.

3.3. Implementación de una Pila con Arreglos

En C++, podemos implementar una pila utilizando arreglos. Aquí tienes la estructura básica para crear una pila con un array.

Ejemplo: Pila con Arreglos

```
#define MAX 100

struct Pila {
    int datos [MAX];
    int tope;

    Pila() {
        tope = -1; // Pila vacia
    }

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

    bool estaLlena() {
        return tope == MAX - 1;
    }

    void push(int valor) {
        if (estaLlena()) {
            cout << "La pila esta llena." << endl;
        } else {
            datos[++tope] = valor;
        }
    }

    int pop() {
        if (estaVacia()) {
            cout << "La pila esta vacia." << endl;
            return -1; // Pila vacia
        } else {
            return datos[tope--];
        }
    }

    int top() {
        if (estaVacia()) {
            cout << "La pila esta vacia." << endl;
            return -1;
        } else {
            return datos[tope];
        }
    }
};
```

Este código define una pila utilizando un arreglo de tamaño fijo **MAX**. La pila tiene un índice **tope** que apunta al último elemento insertado. Las funciones básicas son **push**, **pop** y **top**.

3.4. Ejemplo: Uso de la Pila

A continuación, mostramos cómo usar esta estructura de pila:

Ejemplo: Usar la Pila

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

int main() {
    Pila pila;

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

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

    cout << "Elemento retirado: " << pila.pop() << endl;
    cout << "Elemento retirado: " << pila.pop() << endl;

    cout << "Elemento en la cima - despues de retirar: "
    << pila.top() << endl;

    return 0;
}
```

Este código crea una pila, inserta tres elementos, y luego los va retirando uno por uno, mostrando la cima de la pila antes y después de cada operación.

3.5. Implementación de una Pila con Listas Enlazadas

Si no conocemos de antemano el tamaño de la pila, podemos implementar una pila utilizando listas enlazadas, lo que nos permite añadir y quitar elementos de forma dinámica.

Ejemplo: Pila con Listas Enlazadas

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

class Pila {
private:
    Nodo* cima;

public:
    Pila() {
        cima = nullptr;
    }

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

    void push(int valor) {
        Nodo* nuevoNodo = new Nodo();
        nuevoNodo->dato = valor;
        nuevoNodo->siguiente = cima;
        cima = nuevoNodo;
    }

    int pop() {
        if (estaVacia()) {
            cout << "La pila est - vacia." << endl;
            return -1;
        } else {
            Nodo* temp = cima;
            int valor = cima->dato;
            cima = cima->siguiente;
            delete temp;
            return valor;
        }
    }

    int top() {
        if (estaVacia()) {
            cout << "La pila est - vacia." << endl;
            return -1;
        } else {
            return cima->dato;
        }
    }
};
```

Esta implementación usa una lista enlazada donde cada nodo guarda un valor y una referencia al siguiente nodo. La **cima** de la pila es el primer nodo de la lista.

3.6. Ejemplo: Uso de la Pila con Listas Enlazadas

A continuación, mostramos cómo usar la pila implementada con listas enlazadas:

Ejemplo: Usar la Pila con Listas Enlazadas

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

int main() {
    Pila pila;

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

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

    cout << "Elemento retirado: " << pila.pop() << endl;
    cout << "Elemento retirado: " << pila.pop() << endl;

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

    return 0;
}
```

Este código es similar al ejemplo anterior, pero ahora la pila se maneja mediante una lista enlazada, lo que permite que los elementos se agreguen y se eliminen de manera dinámica sin límite de tamaño.

3.7. Aplicaciones de las Pilas

Las pilas tienen muchas aplicaciones en programación, especialmente en problemas que requieren que se almacenen y procesen datos en un orden específico, como el procesamiento de expresiones matemáticas y la reversión de secuencias.

3.7.1. Ejemplo: Evaluación de Expresiones

Uno de los usos más comunes de las pilas es en la evaluación de expresiones aritméticas en notación postfija.

Ejemplo: Evaluación de Expresión Postfija

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

int evaluarPostfija(string exp) {
    stack<int> pila;

    for (char c : exp) {
        if (isdigit(c)) {
            pila.push(c - '0');
        } else {
            int b = pila.top(); pila.pop();
            int a = pila.top(); pila.pop();
            if (c == '+') pila.push(a + b);
            if (c == '-') pila.push(a - b);
            if (c == '*') pila.push(a * b);
            if (c == '/') pila.push(a / b);
        }
    }
    return pila.top();
}

int main() {
    string exp = "23+5*7";
    cout << "Resultado: " << evaluarPostfija(exp) << endl;
    return 0;
}
```

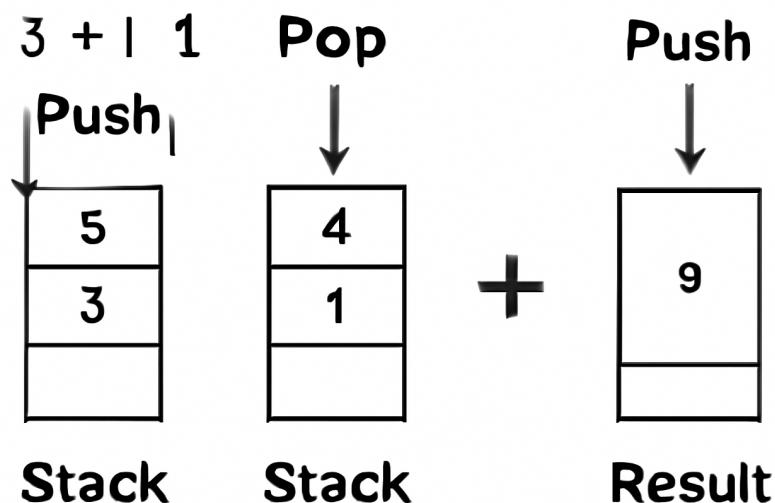
Este código evalúa una expresión postfija como "23+5*7" devuelve el resultado. La pila almacena los operandos mientras procesa los operadores.

3.8. Actividad sugerida

Actividad

Escribe un programa que simule el uso de una pila en un sistema de navegación de páginas web. Cada vez que se navega a una nueva página, se agrega a la pila. Al presionar ".^atrás", se elimina la página más reciente de la pila y se regresa a la página anterior.

3.9. Resumen visual



Resumen del capítulo

- Una pila sigue el principio LIFO.
- Las operaciones principales son push, pop, top, y empty.
- Las pilas pueden implementarse con arreglos o listas enlazadas.
- Son útiles para la evaluación de expresiones y el control de flujo de programas.

Listas Enlazadas

4.1. ¿Qué es una lista enlazada?

Una lista enlazada es una estructura de datos lineal en la que cada elemento (llamado **nodo**) contiene dos partes:

- Un **dato** (por ejemplo, un número o una palabra).
- Un **puntero** que apunta al **siguiente nodo** en la lista.

La principal ventaja de las listas enlazadas sobre los arreglos es que no es necesario conocer de antemano el tamaño de la lista, ya que se pueden ir añadiendo elementos de manera dinámica.

Linked List

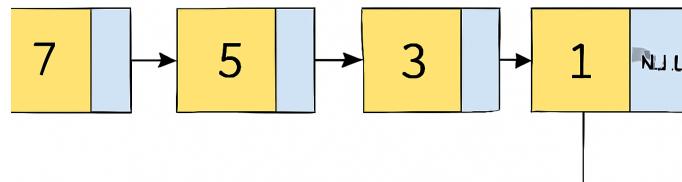


Imagen: Representación gráfica de una lista enlazada.

4.2. Estructura de un Nodo

Un nodo en una lista enlazada generalmente tiene dos partes:

- **dato:** La información que almacena el nodo.
- **siguiente:** Un puntero que apunta al siguiente nodo en la lista.

La estructura de un nodo en C++ puede definirse como:

Ejemplo: Nodo de una Lista Enlazada

```
struct Nodo {  
    int dato;           // Almacena la información  
    Nodo* siguiente;  // Apunta al siguiente nodo  
};
```

El puntero `siguiente` es lo que enlaza un nodo con el siguiente en la lista.

4.3. ¿Por qué usar listas enlazadas?

Las listas enlazadas son útiles cuando:

- No conocemos de antemano el número de elementos que tendremos.
- Queremos insertar o eliminar elementos de manera eficiente en cualquier parte de la lista sin mover otros elementos.
- Necesitamos una estructura dinámica y flexible.

4.4. Tipos de Listas Enlazadas

Existen varios tipos de listas enlazadas:

- **Lista enlazada simple:** Cada nodo apunta al siguiente nodo.
- **Lista doblemente enlazada:** Cada nodo tiene dos punteros, uno al siguiente nodo y otro al nodo anterior.
- **Lista circular:** El último nodo apunta al primer nodo, formando un ciclo.

4.5. Implementación de una Lista Enlazada Simple

En C++, podemos crear una lista enlazada simple con la siguiente estructura:

Ejemplo: Lista Enlazada Simple

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

class ListaEnlazada {
private:
    Nodo* cabeza;

public:
    ListaEnlazada() {
        cabeza = nullptr; // Lista vacía al inicio
    }

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

    void insertarAlPrincipio(int valor) {
        Nodo* nuevoNodo = new Nodo();
        nuevoNodo->dato = valor;
        nuevoNodo->siguiente = cabeza;
        cabeza = nuevoNodo;
    }

    void mostrar() {
        Nodo* actual = cabeza;
        while (actual != nullptr) {
            cout << actual->dato << " -> ";
            actual = actual->siguiente;
        }
        cout << "NULL" << endl;
    }
};
```

Este código define una lista enlazada donde cada nodo contiene un valor y un puntero al siguiente nodo. La función `insertarAlPrincipio` inserta un nodo al principio de la lista.

4.6. Ejemplo: Uso de una Lista Enlazada

Ahora vamos a ver cómo usar la lista enlazada que acabamos de implementar.

Ejemplo: Uso de la Lista Enlazada

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

int main() {
    ListaEnlazada lista;

    lista.insertarAlPrincipio(10);
    lista.insertarAlPrincipio(20);
    lista.insertarAlPrincipio(30);

    cout << "Lista enlazada: ";
    lista.mostrar(); // Muestra la lista

    return 0;
}
```

Este código crea una lista enlazada, inserta tres elementos al principio y luego muestra la lista.

4.7. Lista Dblemente Enlazada

En una lista doblemente enlazada, cada nodo tiene dos punteros: uno que apunta al siguiente nodo y otro que apunta al nodo anterior.

La estructura de un nodo en una lista doblemente enlazada es la siguiente:

Ejemplo: Nodo Dblemente Enlazado

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

class ListaDoble {
private:
    NodoDoble* cabeza;
    NodoDoble* cola;

public:
    ListaDoble() {
        cabeza = nullptr;
        cola = nullptr;
    }

    void insertarAlFinal(int valor) {
        NodoDoble* nuevoNodo = new NodoDoble();
        nuevoNodo->dato = valor;
        nuevoNodo->siguiente = nullptr;
        nuevoNodo->anterior = cola;

        if (cola != nullptr) {
            cola->siguiente = nuevoNodo;
        }
        cola = nuevoNodo;

        if (cabeza == nullptr) {
            cabeza = nuevoNodo;
        }
    }

    void mostrar() {
        NodoDoble* actual = cabeza;
        while (actual != nullptr) {
            cout << actual->dato << " ->- ";
            actual = actual->siguiente;
        }
        cout << "NULL" << endl;
    }
};
```

Este código crea una lista doblemente enlazada, donde los nodos tienen punteros tanto al siguiente nodo como al nodo anterior.

4.8. Ejemplo: Uso de la Lista Dblemente Enlazada

Ahora vamos a ver cómo utilizar la lista doblemente enlazada para insertar elementos al final de la lista y mostrarla:

Ejemplo: Uso de la Lista Dblemente Enlazada

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

int main() {
    ListaDoble lista;

    lista.insertarAlFinal(10);
    lista.insertarAlFinal(20);
    lista.insertarAlFinal(30);

    cout << "Lista - doblemente - enlazada : - ";
    lista.mostrar(); // Muestra la lista

    return 0;
}
```

Este código crea una lista doblemente enlazada, inserta tres elementos al final y luego muestra la lista.

4.9. Lista Circular

Una lista circular es una lista en la que el último nodo apunta al primer nodo, formando un ciclo.

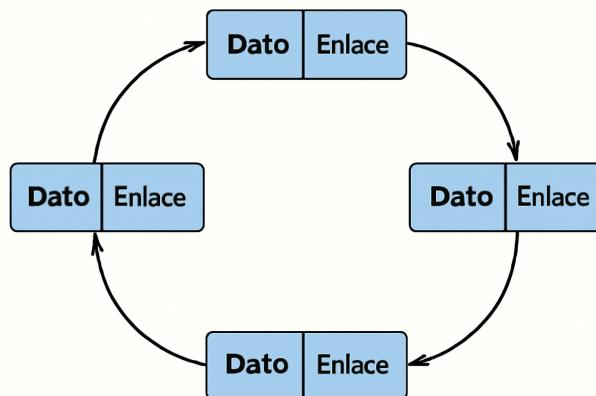


Imagen: Representación gráfica de una lista circular.

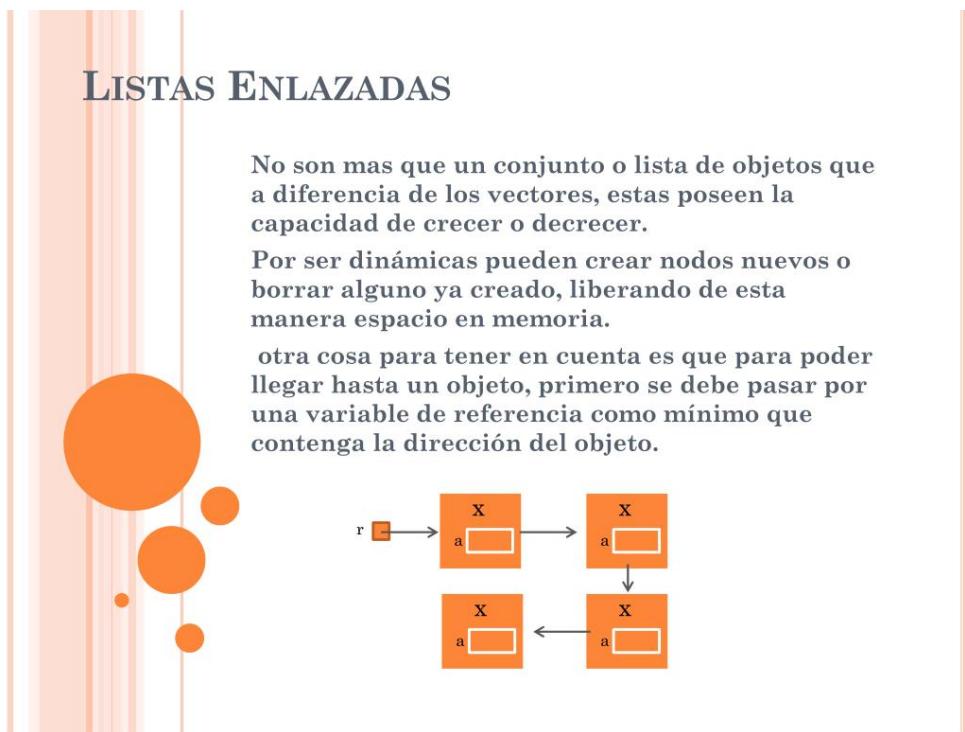
En una lista circular, tanto el primer como el último nodo están enlazados, lo que permite recorrer la lista sin necesidad de saber el tamaño de la misma.

4.10. Actividad sugerida

Actividad

Crea una lista enlazada circular que almacene los nombres de tus amigos. Implementa una función para agregar un nombre al final de la lista y otra para recorrerla e imprimir todos los nombres.

4.11. Resumen visual



Resumen del capítulo

- Una lista enlazada es una estructura de datos donde cada nodo apunta al siguiente.
- Las listas enlazadas pueden ser simples, dobles o circulares.
- Las listas dobles permiten recorrerla en ambas direcciones, y las listas circulares permiten recorrerla de manera infinita.
- Son útiles cuando necesitamos una estructura dinámica de tamaño variable.

Listas Dblemente Enlazadas y Circulares

5.1. ¿Qué es una lista doblemente enlazada?

Una lista doblemente enlazada es una variación de la lista enlazada en la que cada nodo tiene dos punteros:

- Un puntero al siguiente nodo.
- Un puntero al nodo anterior.

Esto permite recorrer la lista tanto hacia adelante como hacia atrás, lo que proporciona una mayor flexibilidad en algunas operaciones.



Imagen: Representación gráfica de una lista doblemente enlazada.

5.2. Ventajas de las listas doblemente enlazadas

Las listas doblemente enlazadas tienen varias ventajas sobre las listas simples:

- Permiten recorrer la lista en ambas direcciones.
- Son útiles para operaciones de inserción y eliminación de nodos en cualquier parte de la lista, sin necesidad de recorrerla desde el principio.
- Son especialmente útiles en estructuras como *colas de doble entrada*, donde se necesita manipular los elementos desde ambos extremos.

5.3. Estructura de un Nodo en una Lista Dblemente Enlazada

El nodo de una lista doblemente enlazada tiene tres componentes:

- **dato:** Almacena la información del nodo.
- **siguiente:** Apunta al siguiente nodo en la lista.
- **anterior:** Apunta al nodo anterior en la lista.

Ejemplo: Nodo en una Lista Doblemente Enlazada

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

En este ejemplo, cada nodo contiene un puntero tanto hacia el siguiente nodo (**siguiente**) como hacia el nodo anterior (**anterior**).

5.4. Implementación de una Lista Doblemente Enlazada

En una lista doblemente enlazada, cada nodo está conectado tanto con el nodo anterior como con el siguiente. Aquí tenemos un ejemplo de cómo implementarlo en C++:

Ejemplo: Lista Dblemente Enlazada

```
class ListaDoble {
private:
    NodoDoble* cabeza;
    NodoDoble* cola;

public:
    ListaDoble() {
        cabeza = nullptr;
        cola = nullptr;
    }

    // Inserta al final
    void insertarAlFinal(int valor) {
        NodoDoble* nuevoNodo = new NodoDoble();
        nuevoNodo->dato = valor;
        nuevoNodo->siguiente = nullptr;
        nuevoNodo->anterior = cola;

        if (cola != nullptr) {
            cola->siguiente = nuevoNodo;
        }
        cola = nuevoNodo;

        if (cabeza == nullptr) {
            cabeza = nuevoNodo;
        }
    }

    // Muestra la lista de cabeza a cola
    void mostrar() {
        NodoDoble* actual = cabeza;
        while (actual != nullptr) {
            cout << actual->dato << "-<->-";
            actual = actual->siguiente;
        }
        cout << "NULL" << endl;
    }

    // Muestra la lista de cola a cabeza
    void mostrarReversa() {
        NodoDoble* actual = cola;
        while (actual != nullptr) {
            cout << actual->dato << "-<->-";
            actual = actual->anterior;
        }
        cout << "NULL" << endl;
    }
};
```

Este código implementa una lista doblemente enlazada con las funciones básicas de inserción al final y recorrido en ambas direcciones.

5.5. Ejemplo: Uso de una Lista Dblemente Enlazada

A continuación, vemos cómo usar la lista doblemente enlazada para insertar elementos al final y recorrerla de ambas maneras (de cabeza a cola y de cola a cabeza):

Ejemplo: Uso de la Lista Doblemente Enlazada

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

int main() {
    ListaDoble lista;

    lista.insertarAlFinal(10);
    lista.insertarAlFinal(20);
    lista.insertarAlFinal(30);

    cout << "Lista -de- cabeza -a- cola:-";
    lista.mostrar(); // Muestra la lista de cabeza a cola

    cout << "Lista -de- cola -a- cabeza:-";
    lista.mostrarReversa(); // Muestra la lista de cola a cabeza

    return 0;
}
```

Este código inserta tres elementos en la lista doblemente enlazada y luego muestra la lista de dos maneras: de cabeza a cola y de cola a cabeza.

5.6. ¿Qué es una lista circular?

Una lista circular es una variación de las listas enlazadas, donde el último nodo apunta al primer nodo, formando un ciclo continuo. Esto permite recorrer la lista de manera infinita sin necesidad de saber el tamaño de la lista.

Lista circular

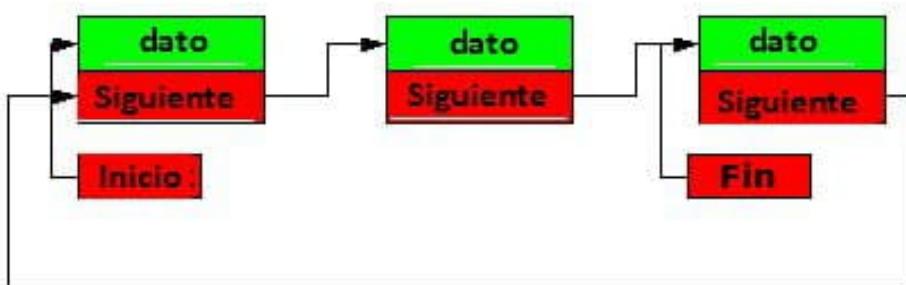


Imagen: Representación gráfica de una lista circular.

En una lista circular, el último nodo tiene un puntero que apunta de nuevo al primer nodo, formando un ciclo.

5.7. Implementación de una Lista Circular

A continuación, veremos cómo implementar una lista circular en C++.

Ejemplo: Lista Circular

```

class ListaCircular {
private:
    NodoDoble* cabeza;
    NodoDoble* cola;

public:
    ListaCircular() {
        cabeza = nullptr;
        cola = nullptr;
    }

    // Inserta al final y conecta la cola con la cabeza
    void insertarAlFinal(int valor) {
        NodoDoble* nuevoNodo = new NodoDoble();
        nuevoNodo->dato = valor;
        nuevoNodo->siguiente = cabeza;
        nuevoNodo->anterior = cola;

        if (cabeza != nullptr) {
            cola->siguiente = nuevoNodo;
        }
        cola = nuevoNodo;

        if (cabeza == nullptr) {
            cabeza = nuevoNodo;
        }
        cabeza->anterior = cola; // La cabeza apunta a la cola
    }

    // Muestra la lista circular
    void mostrar() {
        if (cabeza == nullptr) return;

        NodoDoble* actual = cabeza;
        do {
            cout << actual->dato << " ->- ";
            actual = actual->siguiente;
        } while (actual != cabeza);
        cout << "NULL" << endl;
    }
};

```

Este código implementa una lista circular donde el último nodo apunta de nuevo al primer nodo, formando un ciclo.

5.8. Ejemplo: Uso de la Lista Circular

A continuación, vemos cómo usar la lista circular para insertar elementos y recorrerla:

Ejemplo: Uso de la Lista Circular

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

int main() {
    ListaCircular lista;

    lista.insertarAlFinal(10);
    lista.insertarAlFinal(20);
    lista.insertarAlFinal(30);

    cout << "Lista circular : -";
    lista.mostrar(); // Muestra la lista circular

    return 0;
}
```

Este código crea una lista circular, inserta tres elementos y muestra la lista circular.

5.9. Aplicaciones de las Listas Dblemente Enlazadas y Circulares

Las listas doblemente enlazadas y circulares tienen varias aplicaciones en la programación:

- **Navegadores web:** Usan listas doblemente enlazadas para almacenar el historial de páginas, permitiendo navegar hacia adelante y hacia atrás.
- **Sistemas de mensajería:** Los mensajes pueden ser procesados en una lista circular, donde el último mensaje siempre apunta al primero.
- **Memoria dinámica:** Las listas circulares son útiles para la administración de memoria en sistemas operativos.

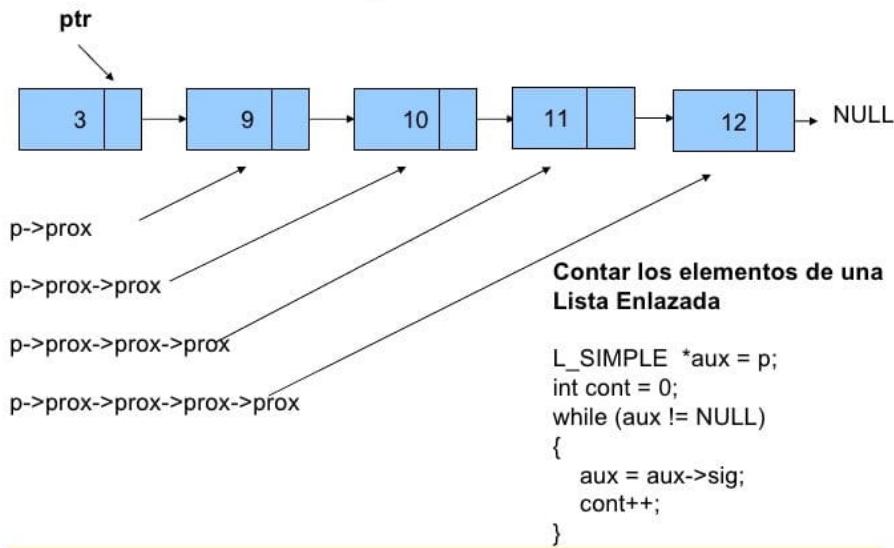
5.10. Actividad sugerida

Actividad

Crea una lista circular que almacene los nombres de tus compañeros de clase. Implementa una función para agregar un nombre al final de la lista y otra para recorrerla e imprimir todos los nombres.

5.11. Resumen visual

Movimiento/búsqueda a través de la lista



Actualizado: Abril 2007

Ing. Julio César Canelón Rangel

Resumen del capítulo

- Las listas doblemente enlazadas permiten recorrer la lista en ambas direcciones.
- Las listas circulares permiten recorrer la lista infinitamente, ya que el último nodo apunta al primero.
- Son útiles cuando necesitamos una estructura dinámica y flexible.

Colas

6.1. ¿Qué es una Cola?

Una **cola** (**queue**) es una estructura de datos lineal que sigue el principio **FIFO** (First In, First Out), lo que significa que el primer elemento en entrar es el primero en salir. Es como una fila en una tienda: el primer cliente que llega es el primero que es atendido.

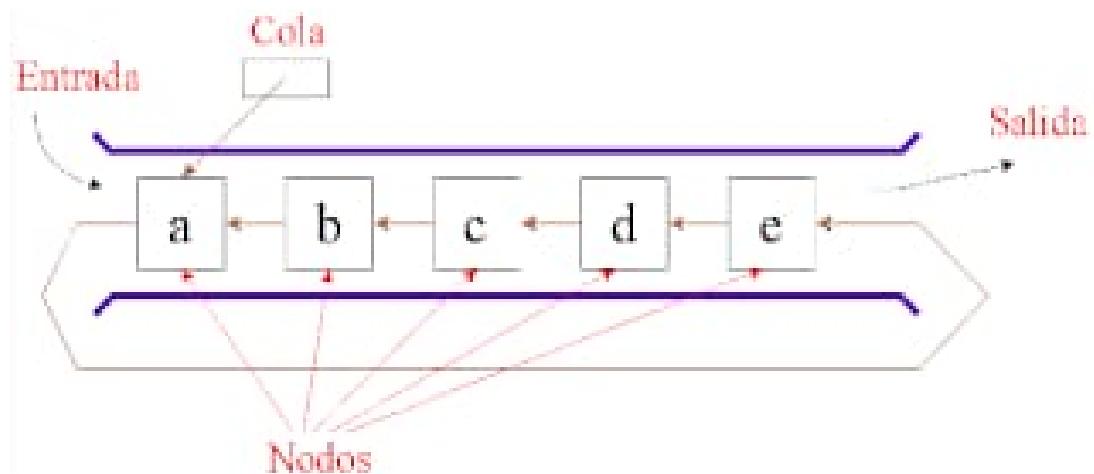


Imagen: Representación gráfica de una cola.

6.2. Operaciones Básicas de una Cola

Las operaciones más comunes en una cola son:

- **enqueue:** Agrega un nuevo elemento al final de la cola.
- **dequeue:** Elimina el elemento del frente de la cola.
- **front:** Muestra el valor al frente de la cola sin eliminarlo.
- **empty:** Verifica si la cola está vacía.

6.3. Implementación de una Cola con Arreglos

En C++, podemos implementar una cola utilizando un arreglo. Aquí tienes la estructura básica para crear una cola con un array.

Ejemplo: Cola con Arreglos

```
#define MAX 100

class Cola {
private:
    int cola[MAX];
    int frente, fin;

public:
    Cola() {
        frente = -1;
        fin = -1;
    }
    bool estaVacia() {
        return frente == -1;
    }
    bool estaLlena() {
        return fin == MAX - 1;
    }
    void enqueue(int valor) {
        if (estaLlena()) {
            cout << "La cola est llena." << endl;
        } else {
            if (frente == -1) frente = 0;
            cola[++fin] = valor;
        }
    }
    int dequeue() {
        if (estaVacia()) {
            cout << "La cola est vacia." << endl;
            return -1;
        } else {
            int valor = cola[frente];
            if (frente >= fin) {
                frente = fin = -1; // Cola vacia
            } else {
                frente++;
            }
            return valor;
        }
    }
    int front() {
        if (estaVacia()) {
            cout << "La cola est vacia." << endl;
            return -1;
        } else {
            return cola[frente];
        }
    }
};
```

Este código implementa una cola con un arreglo. Las funciones básicas de la cola son `enqueue`, `dequeue`, y `front`.

6.4. Ejemplo: Uso de una Cola

A continuación, mostramos cómo usar esta cola para agregar y eliminar elementos:

Ejemplo: Usar la Cola

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

int main() {
    Cola cola;

    cola.enqueue(10);
    cola.enqueue(20);
    cola.enqueue(30);

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

    cout << "Elemento - al - frente - despues de - eliminar : -"
    << cola.front() << endl;

    return 0;
}
```

Este código crea una cola, inserta tres elementos, y luego los va eliminando uno por uno, mostrando el frente de la cola antes y después de cada operación.

6.5. Cola Circular

En una **cola circular**, el primer espacio vacío de la cola se reutiliza cuando se eliminan elementos. Esto permite usar el espacio de manera más eficiente.

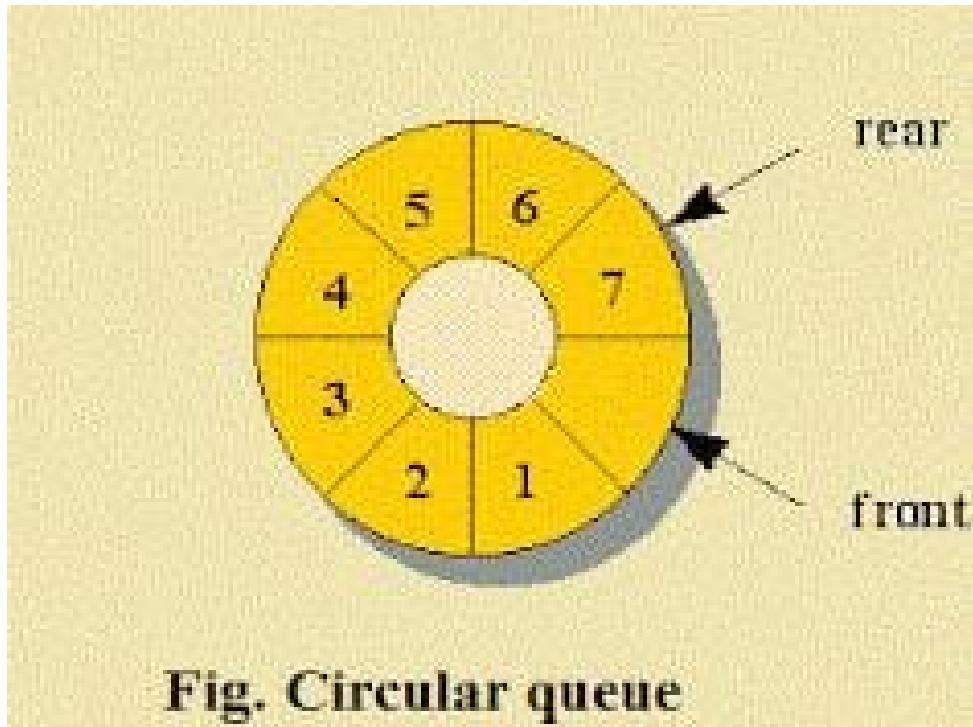


Fig. Circular queue

Imagen: Representación gráfica de una cola circular.

En una cola circular, el último nodo apunta al primero, formando un ciclo. Esto es útil cuando la cola se utiliza de manera continua y no se desperdicia espacio en el arreglo.

6.6. Implementación de una Cola Circular

La implementación de una cola circular en C++ es similar a una cola normal, pero con el agregado de la reutilización de espacios.

Ejemplo: Cola Circular

```
class ColaCircular {  
private:  
    int cola [MAX];  
    int frente , fin;  
  
public:  
    ColaCircular () {  
        frente = -1;  
        fin = -1;  
    }  
    bool estaVacia() {  
        return frente == -1;  
    }  
    bool estaLlena() {  
        return (fin + 1) % MAX == frente ;  
    }  
    void enqueue(int valor) {  
        if (estaLlena()) {  
            cout << "La - cola - circular - esta - llena ." << endl;  
        } else {  
            if (frente == -1) frente = 0;  
            fin = (fin + 1) % MAX;  
            cola [fin] = valor ;  
        }  
    }  
    int dequeue() {  
        if (estaVacia()) {  
            cout << "La - cola - circular - esta - vacia ." << endl ;  
            return -1;  
        } else {  
            int valor = cola [frente];  
            if (frente == fin) {  
                frente = fin = -1; // Cola vacia  
            } else {  
                frente = (frente + 1) % MAX;  
            }  
            return valor ;  
        }  
    }  
    int front () {  
        if (estaVacia()) {  
            cout << "La - cola - circular - esta - vacia ." << endl ;  
            return -1;  
        } else {  
            return cola [frente];  
        }  
    }  
};
```

Este código implementa una cola circular, utilizando el operador modulo (%) para asegurarse de que el puntero al final de la cola vuelva al principio cuando se elimina un elemento.

6.7. Ejemplo: Uso de una Cola Circular

A continuación, vemos como utilizar la cola circular para agregar y eliminar elementos de manera continua:

Ejemplo: Uso de la Cola Circular

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

int main() {
    ColaCircular cola;

    cola.enqueue(10);
    cola.enqueue(20);
    cola.enqueue(30);

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

    cout << "Elemento - retirado : -" << cola.dequeue() << endl;
    cout << "Elemento - retirado : -" << cola.dequeue() << endl;

    cola.enqueue(40); // Reutiliza el espacio vacío
    cout << "Elemento - al - frente - después - de - agregar - 40: -"
    << cola.front() << endl;

    return 0;
}
```

Este código crea una cola circular, inserta y elimina elementos, y muestra cómo se reutiliza el espacio vacío cuando se eliminan elementos.

6.8. Ejemplos prácticos de uso de Colas

Las colas son estructuras muy útiles y se utilizan ampliamente en la programación. A continuación, se presentan tres ejemplos prácticos de su uso:

6.8.1. 1. Simulación de Atención al Cliente

Una aplicación común de las colas es en los sistemas de atención al cliente, donde los clientes esperan su turno para ser atendidos. Aquí, los clientes se agregan a la cola y se atienden en el orden en el que llegaron.

Ejemplo: Cola de Atención al Cliente

```
struct Cliente {
    int id;
    string nombre;
};

class ColaClientes {
private:
    Cliente cola[MAX];
    int frente, fin;

public:
    ColaClientes() {
        frente = -1;
        fin = -1;
    }

    void agregarCliente(int id, string nombre) {
        if (fin == MAX - 1) {
            cout << "No hay m s - espacio - en - la - cola ." << endl;
            return;
        }
        if (frente == -1) frente = 0;
        cola[++fin] = {id, nombre};
    }

    void atenderCliente() {
        if (frente == -1) {
            cout << "No hay clientes - en - la - cola ." << endl;
            return;
        }
        cout << "Atendiendo a " << cola[frente].nombre << endl;
        if (frente == fin) frente = fin = -1;
        // Cola vac a
        else frente++;
    }

    void mostrarCola() {
        if (frente == -1) {
            cout << "No hay clientes ." << endl;
            return;
        }
        for (int i = frente; i <= fin; i++) {
            cout << cola[i].nombre << " - (ID: " <<
            cola[i].id << " ) - - - ";
        }
        cout << endl;
    }
};
```

6.9. 2. Cola para Procesamiento de Tareas en un Sistema Operativo

En sistemas operativos, las colas se utilizan para gestionar las tareas que están a la espera de ser procesadas. El sistema toma la primera tarea en la cola y la procesa.

Ejemplo: Cola de Tareas

```
struct Tarea {
    string descripcion;
    int prioridad;
};

class ColaTareas {
private:
    Tarea cola[MAX];
    int frente, fin;

public:
    ColaTareas() {
        frente = -1;
        fin = -1;
    }
    void agregarTarea(string descripcion, int prioridad) {
        if (fin == MAX - 1) {
            cout << "La cola de tareas est - llena."
            << endl;
            return;
        }
        if (frente == -1) frente = 0;
        cola[++fin] = {descripcion, prioridad};
    }
    void procesarTarea() {
        if (frente == -1) {
            cout << "No hay tareas para procesar." << endl;
            return;
        }
        cout << "Procesando tarea:-" << cola[frente]
        .descripcion
        << endl;
        if (frente == fin) frente = fin = -1;
        // Cola vacia
        else frente++;
    }
    void mostrarCola() {
        if (frente == -1) {
            cout << "No hay tareas pendientes." << endl;
            return;
        }
        for (int i = frente; i <= fin; i++) {
            cout << cola[i].descripcion << " - (Prioridad:-"
            << cola[i].prioridad << ") -<-";
        }
        cout << endl;
    }
};
```

6.9.1. 3. Cola de Impresión en una Oficina

Un ejemplo más es el manejo de trabajos de impresión en una oficina. Los trabajos de impresión se agregan a la cola y se imprimen en el orden en que llegan.

Ejemplo: Cola de Impresión

```
struct TrabajoImpresion {
    string documento;
    int paginas;
};

class ColaImpresion {
private:
    TrabajoImpresion cola[MAX];
    int frente, fin;

public:
    ColaImpresion() {
        frente = -1;
        fin = -1;
    }

    void agregarTrabajo(string documento, int paginas) {
        if (fin == MAX - 1) {
            cout << "La cola de impresión está llena."
            << endl;
            return;
        }
        if (frente == -1) frente = 0;
        cola[++fin] = {documento, paginas};
    }

    void procesarTrabajo() {
        if (frente == -1) {
            cout << "No hay trabajos en la cola."
            << endl;
            return;
        }
        cout << "Imprimiendo: " <<
        cola[frente].documento << "(" << cola[frente].paginas
        <<
        "- paginas)" << endl;
        if (frente == fin) frente = fin = -1;
// Cola vacía
        else frente++;
    }

    void mostrarCola() {
        if (frente == -1) {
            cout << "No hay trabajos pendientes." << endl;
            return;
        }
        for (int i = frente; i <= fin; i++) {
            cout << cola[i].documento << "("
            << cola[i].paginas
            << "- paginas)" << endl;
        }
        cout << endl;
    }
}
```

Pilas

7.1. ¿Qué es una Pila?

Una **pila** (**stack**) es una estructura de datos que sigue el principio **LIFO** (Last In, First Out), lo que significa que el último elemento en entrar es el primero en salir. Es como una pila de platos: el último plato que pongas en la pila será el primero en ser retirado.

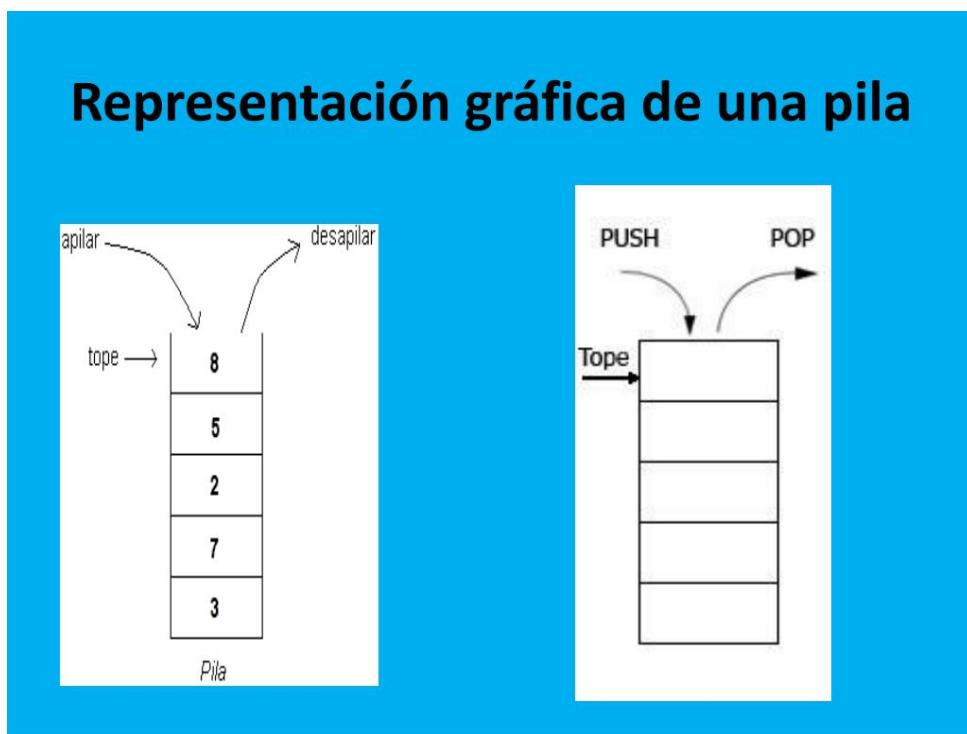


Imagen: Representación gráfica de una pila.

7.2. Operaciones Básicas de una Pila

Las pilas se caracterizan por las siguientes operaciones:

- **push:** Inserta un nuevo elemento en la cima de la pila.
- **pop:** Elimina el elemento en la cima de la pila.
- **top:** Muestra el valor del elemento en la cima sin eliminarlo.
- **empty:** Verifica si la pila está vacía.

7.3. Implementación de una Pila con Arreglos

En C++, podemos implementar una pila utilizando arreglos. Aquí tienes la estructura básica para crear una pila con un arreglo:

Ejemplo: Pila con Arreglos

```
#define MAX 100

class Pila {
private:
    int pila[MAX];
    int cima;

public:
    Pila() {
        cima = -1; // La pila est vaca
    }

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

    bool estaLlena() {
        return cima == MAX - 1;
    }

    void push(int valor) {
        if (estaLlena()) {
            cout << "La pila est llena." << endl;
        } else {
            pila[++cima] = valor;
        }
    }

    int pop() {
        if (estaVacia()) {
            cout << "La pila est vaca." << endl;
            return -1; // Indica que la pila est vaca
        } else {
            return pila[cima--];
        }
    }

    int top() {
        if (estaVacia()) {
            cout << "La pila est vaca." << endl;
            return -1; // Indica que la pila est vaca
        } else {
            return pila[cima];
        }
    }
};
```

Este código define una pila con un arreglo de tamaño MAX. La pila tiene un índice **cima** que apunta al último elemento insertado. Las funciones básicas de la pila son **push**, **pop**, y **top**.

7.4. Ejemplo: Uso de una Pila

A continuación, mostramos cómo usar esta pila para insertar y eliminar elementos:

Ejemplo: Usar la Pila

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

int main() {
    Pila pila;

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

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

    cout << "Elemento retirado: " << pila.pop() << endl;
    cout << "Elemento retirado: " << pila.pop() << endl;

    cout << "Elemento en la cima despues de retirar:
-----" << pila.top() << endl;

    return 0;
}
```

Este código crea una pila, inserta tres elementos y luego los elimina uno por uno, mostrando el valor en la cima de la pila antes y después de cada operación.

7.5. Aplicaciones Comunes de las Pilas

Las pilas son estructuras de datos fundamentales en programación, y se utilizan en una variedad de contextos, como el procesamiento de expresiones matemáticas y la administración de las llamadas a funciones. A continuación, se presentan tres ejemplos prácticos que muestran cómo las pilas se utilizan en situaciones del mundo real.

7.5.1. 1. Evaluación de Expresiones Aritméticas

Las pilas son ampliamente utilizadas para la evaluación de expresiones en notación postfija (también conocida como notación polaca inversa). En esta notación, los operadores siguen a sus operandos, y las pilas se utilizan para almacenar los operandos mientras se procesan los operadores.

Ejemplo: Evaluación de Expresión Postfija

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

int evaluarPostfija(string exp) {
    stack<int> pila;

    for (char c : exp) {
        if (isdigit(c)) {
            pila.push(c - '0');
        } else {
            int b = pila.top(); pila.pop();
            int a = pila.top(); pila.pop();
            if (c == '+') pila.push(a + b);
            if (c == '-') pila.push(a - b);
            if (c == '*') pila.push(a * b);
            if (c == '/') pila.push(a / b);
        }
    }
    return pila.top();
}

int main() {
    string exp = "23+5*";
    cout << "Resultado:-" << evaluarPostfija(exp) << endl;
    return 0;
}
```

Este código evalúa la expresión postfija "23+5*". Devuelve el resultado. La pila almacena los operandos mientras procesa los operadores.

7.5.2. 2. Deshacer y Rehacer Operaciones en un Editor de Texto

En aplicaciones como editores de texto, las pilas se utilizan para implementar las funciones de deshacer (undo) y rehacer (redo). Cada acción del usuario se guarda en una pila de deshacer, y cuando el usuario desea deshacer una acción, el último elemento de la pila se retira.

Ejemplo: Deshacer y Rehacer

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

class Editor {
private:
    stack<string> undoStack;
    stack<string> redoStack;

public:
    void escribir(string texto) {
        undoStack.push(texto);
        cout << "Texto escrito:" << texto << endl;
    }

    void deshacer() {
        if (undoStack.empty()) {
            cout << "No hay mas acciones para deshacer."
            << endl;
        } else {
            string texto = undoStack.top();
            undoStack.pop();
            redoStack.push(texto);
            cout << "Deshecho:" << texto << endl;
        }
    }

    void rehacer() {
        if (redoStack.empty()) {
            cout << "No hay mas acciones para rehacer."
            << endl;
        } else {
            string texto = redoStack.top();
            redoStack.pop();
            undoStack.push(texto);
            cout << "Rehecho:" << texto << endl;
        }
    }
};

int main() {
    Editor editor;
    editor.escribir("Hola");
    editor.escribir("Mundo");
    editor.deshacer();
    editor.rehacer();
    return 0;
}
```

Este código implementa las funcionalidades de deshacer y rehacer utilizando dos pilas, una para las acciones de deshacer y otra para las de rehacer.

7.5.3. 3. Gestión de Funciones en un Programa

Las pilas se utilizan para gestionar las llamadas a funciones en un programa. Cada vez que una función se llama, se coloca un registro en la pila de llamadas. Cuando la función termina, el registro se retira de la pila.

Ejemplo: Pila de Llamadas a Funciones

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

void funcion1() {
    cout << "Ejecutando - func i n - 1." << endl;
}

void funcion2() {
    cout << "Ejecutando - func i n - 2." << endl;
}

void ejecutarFunciones() {
    stack<string> pilaDeLlamadas;

    pilaDeLlamadas.push("funcion1");
    funcion1();
    pilaDeLlamadas.push("funcion2");
    funcion2();

    cout << "Pilas - de - funciones - llamadas : -" ;
    while ( !pilaDeLlamadas.empty() ) {
        cout << pilaDeLlamadas.top() << " -" ;
        pilaDeLlamadas.pop();
    }
    cout << endl;
}

int main() {
    ejecutarFunciones();
    return 0;
}
```

Este código simula cómo las funciones se colocan y retiran de la pila de llamadas a medida que se ejecutan.

7.6. Resumen del Capítulo

En este capítulo, aprendimos sobre las *pilas* y su funcionamiento según el principio **LIFO**. Implementamos pilas con arreglos y vimos ejemplos prácticos de aplicaciones de las pilas, tales como:

1. Evaluación de expresiones aritméticas.
2. Deshacer y rehacer operaciones en un editor de texto.
3. Gestión de funciones en un programa.

Las pilas son estructuras de datos fundamentales en la programación y tienen una amplia gama de aplicaciones en situaciones del mundo real.

Resumen Visual

- Una pila sigue el principio LIFO.
- Las pilas se utilizan en la evaluación de expresiones, deshacer/rehacer operaciones, y la gestión de funciones.
- Pueden implementarse con arreglos o listas enlazadas.

Recursividad

8.1. ¿Qué es la Recursividad?

La **recursividad** es una técnica de programación donde una función se llama a sí misma para resolver un problema más grande, dividiéndolo en subproblemas más pequeños y similares. Es una herramienta poderosa que puede hacer que el código sea más simple y fácil de entender en muchos casos.

Definición clara

La recursividad se resuelve a través de dos principios clave:

- **Caso base:** Es la condición que detiene la recursión. Evita que la función se llame infinitamente.
- **Llamada recursiva:** Es cuando la función se llama a sí misma con una versión más simple del problema.

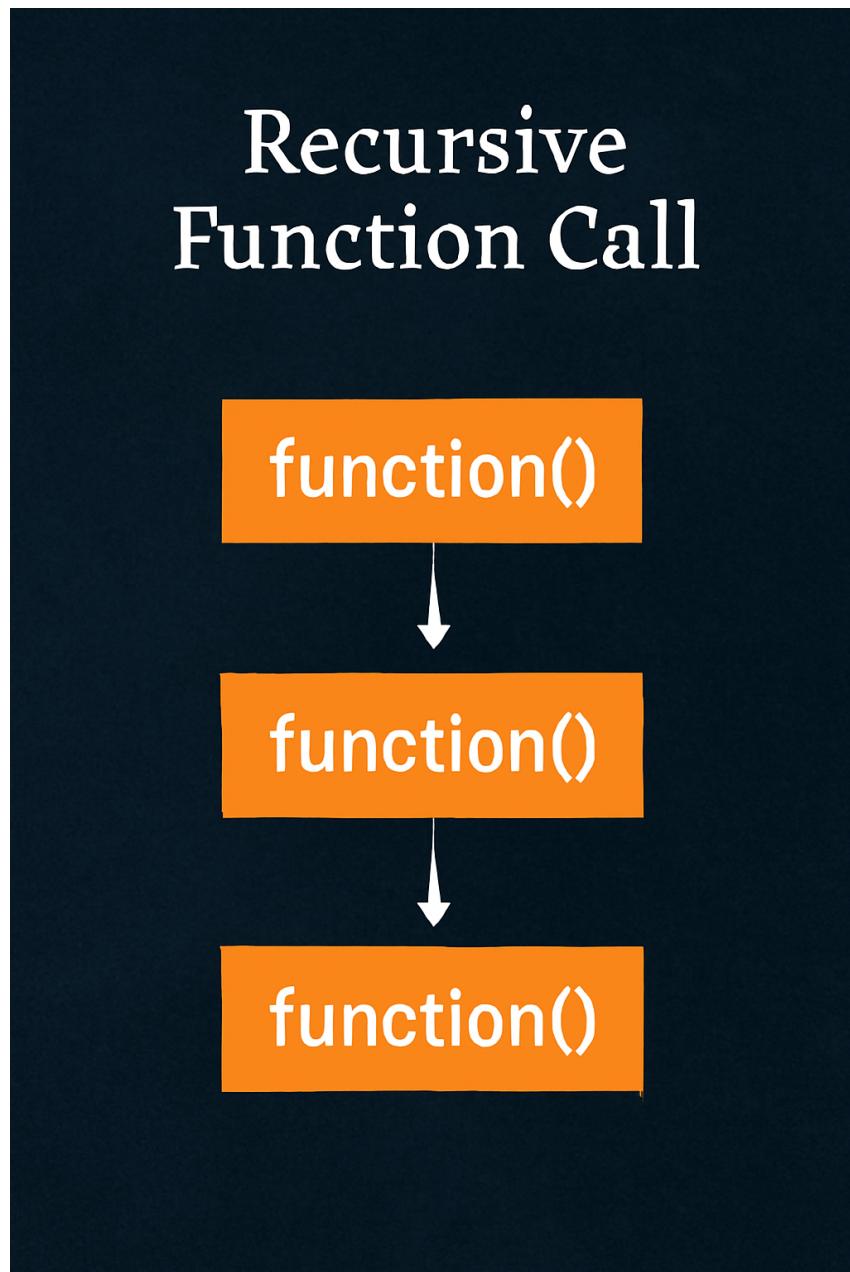


Figura: Ejemplo visual de una función que se llama a sí misma.

8.2. ¿Por qué es importante la Recursividad?

La recursividad es importante porque:

- Permite resolver problemas complejos de manera más elegante.
- Reduce la cantidad de código al dividir el problema en subproblemas más simples.
- Facilita la comprensión de problemas con estructuras de datos como árboles, listas enlazadas, y más.

8.3. Funcionamiento Interno de la Recursividad

Cuando una función se llama a sí misma, se crea una nueva instancia de esa función en la **pila de llamadas** (call stack). Cuando la función llega al **caso base**, las funciones empiezan a resolverse una por una hasta volver a la original.

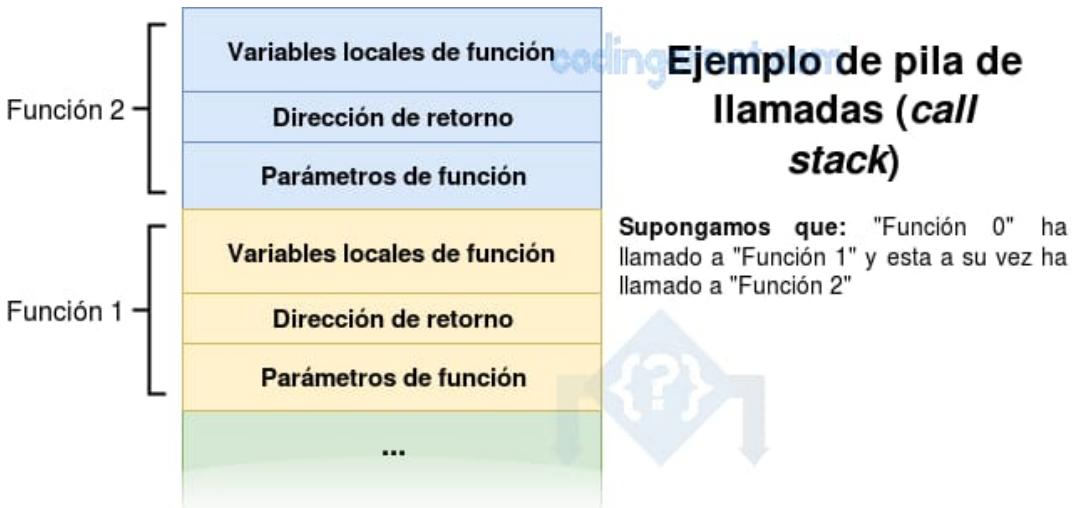


Figura: La pila de llamadas en la recursividad.

8.4. Ejemplo clásico: Factorial

Un ejemplo clásico de recursividad es el cálculo del factorial de un número. El **factorial** de un número n se define como:

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$$

Ejemplo: $4! = 4 \times 3 \times 2 \times 1 = 24$

La versión recursiva en C++ de este cálculo es:

Ejemplo: Factorial Recursivo

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

EXPLICACION:

1. **Caso base:** Cuando $n = 0$, retornamos 1.
2. **Llamada recursiva:** La función se llama a sí misma con $n - 1$ hasta que llega al caso base.

8.5. Ejemplo 2: Serie de Fibonacci

La serie de Fibonacci es una sucesión matemática donde cada número es la suma de los dos anteriores. La serie comienza con 0, 1, 1, 2, 3, 5, 8, 13,

Fórmula:

$$F(n) = F(n - 1) + F(n - 2) \quad \text{con } F(0) = 0 \text{ y } F(1) = 1$$

La versión recursiva de Fibonacci en C++ es:

Ejemplo: Fibonacci Recursivo

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

EXPLICACION:

1. **Caso base:** Cuando $n = 0$, retornamos 0; cuando $n = 1$, retornamos 1.
2. **Llamada recursiva:** La función se llama a sí misma con los valores $n - 1$ y $n - 2$. Este ejemplo ilustra cómo la recursividad puede ser simple y poderosa, pero también ineficiente para valores grandes debido a la repetición de cálculos.

8.6. Ejemplo 3: Búsqueda Binaria Recursiva

La búsqueda binaria es un algoritmo eficiente para encontrar un valor en un arreglo ordenado. El algoritmo divide el rango de búsqueda a la mitad en cada paso, lo que lo convierte en un algoritmo muy eficiente.

Fórmula:

Si es menor, buscamos en la mitad inferior.

La versión recursiva de la búsqueda binaria en C++ es:

Ejemplo: Búsqueda Binaria Recursiva

```
int busquedaBinaria(int arr[], int izquierda, int derecha,
int valor) {
    if (izquierda > derecha) return -1; // No encontrado

    int medio = (izquierda + derecha) / 2;

    if (arr[medio] == valor) return medio;
    if (arr[medio] > valor)
        return busquedaBinaria(arr, izquierda,
                               medio - 1, valor);
    else
        return busquedaBinaria(arr, medio + 1,
                               derecha, valor);
}
```

EXPLICACION:

- 1. Caso base:** Si el valor no se encuentra en el arreglo, se retorna -1.
- 2. Llamada recursiva:** La función se llama con los nuevos límites del arreglo en función de la comparación con el valor medio.

8.7. Ejemplos prácticos de uso de la Recursividad

La recursividad se utiliza en muchos problemas del mundo real. A continuación, se presentan tres ejemplos prácticos donde la recursividad es la solución más eficiente o natural.

8.7.1. 1. Caminos en un Laberinto

Imagina un laberinto donde quieres encontrar un camino desde el inicio hasta la salida. Usando recursividad, puedes explorar cada posible camino hasta encontrar la salida, y si un camino no lleva a la salida, retroceder.

Ejemplo: Resolver un Laberinto con Recursividad

```

bool resolverLaberinto(int laberinto [5][5] , int x ,
int y , int salidaX , int salidaY) {
    if (x == salidaX && y == salidaY) return true;
    // Caso base: salida encontrada

    // Marcar la posición actual como visitada
    laberinto [x] [y] = 2;

    // Intentar mover hacia abajo , arriba , izquierda
    o derecha
    if (x + 1 < 5 && laberinto [x + 1] [y] == 0 &&
        resolverLaberinto(laberinto , x + 1 , y , salidaX ,
        salidaY)) return true;
    if (x - 1 >= 0 && laberinto [x - 1] [y] == 0 &&
        resolverLaberinto(laberinto , x - 1 , y , salidaX ,
        salidaY)) return true;

    if (y + 1 < 5 && laberinto [x] [y + 1] == 0 &&
        resolverLaberinto(laberinto , x , y + 1 , salidaX ,
        salidaY)) return true;
    if (y - 1 >= 0 && laberinto [x] [y - 1] == 0 &&
        resolverLaberinto(laberinto , x , y - 1 , salidaX ,
        salidaY)) return true;

    // Si no se encuentra la salida , retrocedemos
    return false;
}

```

EXPLICACION: Este ejemplo muestra cómo la recursividad puede ser utilizada para explorar todos los caminos posibles en un laberinto y encontrar la salida. Si se encuentra una ruta, la función retorna **true**.

8.7.2. 2. Torres de Hanoi

El problema de las Torres de Hanoi es un clásico problema recursivo que consiste en mover una pila de discos de una torre a otra, siguiendo ciertas reglas.

Ejemplo: Torres de Hanoi

```
void hanoi(int n, char origen, char auxiliar, char destino) {  
    if (n == 1) {  
        cout << "Mover - disco - 1 - de - " << origen << " - a - "  
        << destino << endl;  
        return;  
    }  
    hanoi(n - 1, origen, destino, auxiliar); // Mover  
    n-1 discos  
    cout << "Mover - disco - " << n << " - de - " << origen  
    << " - a - " << destino << endl;  
    hanoi(n - 1, auxiliar, origen, destino); //  
    Mover los discos restantes  
}
```

EXPLICACION: Este problema muestra cómo la recursividad puede descomponer un problema complejo en subproblemas más simples, moviendo discos de una torre a otra. La función se llama a sí misma para mover los discos de la torre auxiliar.

8.7.3. 3. Búsqueda en un Árbol Binario

La recursividad también es útil para recorrer estructuras de datos jerárquicas como los árboles binarios. Aquí se muestra cómo realizar una búsqueda en un árbol binario.

Ejemplo: Búsqueda en un Árbol Binario

```

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

bool buscar(Nodo* raiz , int valor) {
    if (raiz == nullptr) return false;
    // Caso base: rbol vac o

    if (raiz->dato == valor) return true;
    // Valor encontrado

    // Buscar en el sub rbol izquierdo o derecho
    return buscar(raiz->izquierdo , valor) ||
    buscar(raiz->derecho , valor);
}

```

EXPLICACION: Este código muestra cómo se puede usar recursividad para recorrer un árbol binario y buscar un valor en cualquiera de sus nodos.

8.8. Resumen del Capítulo

En este capítulo, exploramos la técnica de la **recursividad** y cómo se puede utilizar para resolver problemas complejos. Vimos ejemplos clásicos como el cálculo del factorial, la serie de Fibonacci y la búsqueda binaria. Además, discutimos tres ejemplos prácticos de recursividad en el mundo real:

1. Caminos en un Laberinto.
2. Torres de Hanoi.
3. Búsqueda en un Árbol Binario.

Resumen Visual

- La recursividad resuelve problemas dividiendo un problema grande en subproblemas más pequeños.
- Requiere un caso base para evitar llamadas infinitas.
- Se usa en situaciones como búsquedas, recorridos en árboles, y problemas como las Torres de Hanoi.

Algoritmos de Ordenación

9.1. ¿Qué es un Algoritmo de Ordenación?

Un algoritmo de ordenación es un proceso utilizado para reorganizar los elementos de una lista o arreglo en un orden específico. El orden puede ser ascendente o descendente, dependiendo de los requisitos del problema.

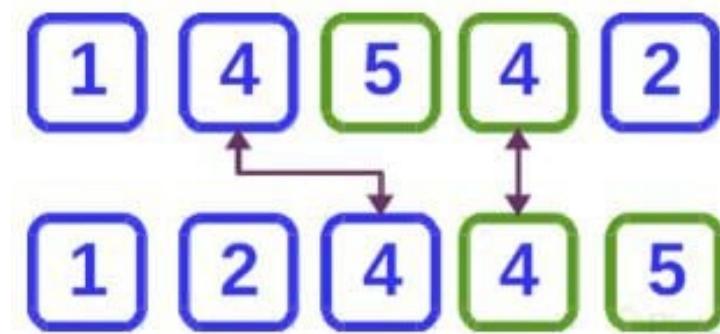


Imagen: Representación visual de un algoritmo de ordenación que organiza los elementos de menor a mayor.

9.2. Algoritmos de Ordenación Comunes

Existen varios algoritmos de ordenación, entre los más conocidos están:

- **Ordenación por burbuja (Bubble Sort):** Es simple pero ineficiente para listas grandes. Compara elementos adyacentes y los intercambia si están en el orden incorrecto.
- **Ordenación por selección (Selection Sort):** Encuentra el mínimo (o máximo) de la lista y lo coloca en la posición correcta.
- **Ordenación por inserción (Insertion Sort):** Construye el arreglo ordenado a medida que recorre la lista, insertando cada elemento en su posición correspondiente.
- **Ordenación rápida (Quick Sort):** Utiliza el enfoque de dividir y vencerás, seleccionando un pivote y organizando los elementos alrededor de él.
- **Ordenación por mezcla (Merge Sort):** También utiliza dividir y vencerás, dividiendo el arreglo en subarreglos más pequeños y luego fusionándolos de manera ordenada.

9.3. Algoritmo de Ordenación por Burbuja (Bubble Sort)

El algoritmo de ordenación por burbuja compara cada par de elementos adyacentes en el arreglo y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que no se requieren más intercambios.

Ejemplo: Ordenación por Burbuja

```
void burbuja(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Intercambiar
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

EXPLICACION:

1. El algoritmo realiza $n-1$ pasadas sobre el arreglo.
2. En cada pasada, compara los elementos adyacentes y los intercambia si están en el orden incorrecto.
3. Al final de cada pasada, el elemento más grande se coloca en su posición correcta.

9.4. Ejemplo: Ordenación por Burbuja

A continuación, vemos cómo usar el algoritmo de burbuja para ordenar un arreglo:

Ejemplo: Usar la Ordenación por Burbuja

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

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

    burbuja(arr, n);

    cout << "Arreglo - ordenado : -" ;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " -" ;
    }
    cout << endl;

    return 0;
}
```

EXPLICACION:

1. El arreglo se ordena utilizando el algoritmo de burbuja.
2. Después de la ordenación, el arreglo se imprime en orden ascendente.

9.5. Algoritmo de Ordenación por Selección (Selection Sort)

El algoritmo de ordenación por selección divide el arreglo en dos partes: una parte ordenada y una parte no ordenada. En cada iteración, selecciona el elemento más pequeño de la parte no ordenada y lo coloca al final de la parte ordenada.

Ejemplo: Ordenación por Selección

```

void seleccion(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        // Intercambiar el mínimo con el primer elemento no ordenado
        int temp = arr[minIdx];
        arr[minIdx] = arr[i];
        arr[i] = temp;
    }
}

```

EXPLICACION:

1. El algoritmo recorre la parte no ordenada del arreglo en cada iteración.
2. Encuentra el elemento más pequeño y lo intercambia con el primer elemento no ordenado.
3. Repite este proceso hasta que todo el arreglo esté ordenado.

9.6. Ejemplo: Ordenación por Selección

A continuación, se muestra cómo usar el algoritmo de selección para ordenar un arreglo:

Ejemplo: Usar la Ordenación por Selección

```

#include <iostream>
using namespace std;

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

    seleccion(arr, n);

    cout << "Arreglo ordenado:-";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}

```

EXPLICACION:

1. El arreglo se ordena utilizando el algoritmo de selección.
2. Después de la ordenación, el arreglo se imprime en orden ascendente.

9.7. Algoritmo de Ordenación por Inserción (Insertion Sort)

El algoritmo de ordenación por inserción construye el arreglo ordenado elemento por elemento, insertando cada nuevo elemento en la posición correcta.

Ejemplo: Ordenación por Inserción

```
void insercion(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        // Mover los elementos del arreglo que son mayores que key
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

EXPLICACION:

1. Comienza desde el segundo elemento y lo compara con los elementos anteriores.
2. Inserta el elemento en su posición correcta dentro de la parte ordenada del arreglo.

9.8. Ejemplo: Ordenación por Inserción

Aquí se muestra cómo usar el algoritmo de inserción para ordenar un arreglo:

Ejemplo: Usar la Ordenación por Inserción

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

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

    insercion(arr, n);

    cout << "Arreglo - ordenado : -" ;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " -" ;
    }
    cout << endl;

    return 0;
}
```

EXPLICACION:

1. El arreglo se ordena utilizando el algoritmo de inserción.
2. Después de la ordenación, el arreglo se imprime en orden ascendente.

9.9. Algoritmo de Ordenación Rápida (Quick Sort)

El algoritmo de ordenación rápida utiliza el enfoque de dividir y vencerás. Se selecciona un pivote y luego se reorganizan los elementos en dos particiones: una con los elementos menores que el pivote y otra con los elementos mayores.

Ejemplo: Ordenación Rápida

```
int particionar(int arr[], int bajo, int alto) {
    int pivote = arr[alto];
    int i = bajo - 1;
    for (int j = bajo; j < alto; j++) {
        if (arr[j] <= pivote) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[alto]);
    return i + 1;
}

void quickSort(int arr[], int bajo, int alto) {
    if (bajo < alto) {
        int pi = particionar(arr, bajo, alto);
        quickSort(arr, bajo, pi - 1);
        quickSort(arr, pi + 1, alto);
    }
}
```

EXPLICACION:

1. El algoritmo selecciona un pivote.
2. Luego divide el arreglo en dos partes, una con elementos menores que el pivote y otra con elementos mayores.
3. Llama recursivamente a `quickSort` para ordenar ambas particiones.

9.10. Ejemplo: Ordenación Rápida

A continuación, se muestra cómo usar el algoritmo de ordenación rápida:

Ejemplo: Usar la Ordenación Rápida

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

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

    quickSort(arr, 0, n - 1);

    cout << "Arreglo ordenado: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

EXPLICACION:

1. El arreglo se ordena utilizando el algoritmo de ordenación rápida.
2. Después de la ordenación, el arreglo se imprime en orden ascendente.

9.11. Resumen del Capítulo

En este capítulo, aprendimos sobre los diferentes algoritmos de ordenación y su implementación en C++. Los algoritmos cubiertos son:

- Ordenación por burbuja (Bubble Sort).
- Ordenación por selección (Selection Sort).
- Ordenación por inserción (Insertion Sort).
- Ordenación rápida (Quick Sort).

Además, proporcionamos ejemplos prácticos para cada uno de ellos, mostrando cómo ordenar un arreglo de manera eficiente.

Resumen Visual

- Los algoritmos de ordenación reorganizan los elementos de un arreglo en un orden específico.
- Diferentes algoritmos tienen ventajas según el tamaño de los datos y la eficiencia deseada.
- La ordenación rápida es más eficiente para arreglos grandes, mientras que la ordenación por burbuja es más fácil de entender pero menos eficiente.

Unidad 2: Programación Competitiva en C++

10.1. Introducción a la Programación Competitiva

La programación competitiva es un campo que implica la resolución de problemas utilizando algoritmos y estructuras de datos eficientes, dentro de un tiempo y espacio limitados. Las competiciones de programación, como las de ICPC o Codeforces, desafían a los programadores a resolver estos problemas bajo presión.

“La programación competitiva es una prueba de habilidad y rapidez para resolver problemas complejos de manera eficiente.”

10.2. ¿Qué es la Programación Competitiva?

La programación competitiva implica el uso de algoritmos clásicos y estructuras de datos para resolver problemas de forma rápida y eficiente. Estos problemas pueden incluir matemáticas, grafos, programación dinámica, etc. El objetivo es encontrar soluciones óptimas bajo condiciones estrictas de tiempo y espacio.

¿Por qué participar en programación competitiva?

- Mejora tus habilidades en algoritmos y estructuras de datos.
- Aumenta tu capacidad para resolver problemas bajo presión.
- Accede a oportunidades en empresas tecnológicas que valoran las habilidades de resolución de problemas.

¿Cómo empezar en programación competitiva?

Para comenzar en la programación competitiva, es importante tener una comprensión sólida de:

- Lenguajes de programación como C++ o Python.
- Conceptos clave en algoritmos y estructuras de datos.
- Participación en plataformas de competiciones como Codeforces, LeetCode, o TopCoder.

10.3. Plataformas para Programación Competitiva

Existen diversas plataformas que permiten a los programadores participar en competiciones de programación. Las más populares incluyen:

- **Codeforces**: Una plataforma popular con múltiples problemas y competiciones.
- **LeetCode**: Ofrece problemas de entrevistas y competencia.
- **TopCoder**: Una de las plataformas más antiguas de programación competitiva.
- **HackerRank**: Ideal para aprender y practicar habilidades en C++, Java, Python, y más.



Imagen: Ejemplo de plataformas donde puedes practicar programación competitiva.

10.4. Conceptos Clave en Programación Competitiva

En la programación competitiva, es esencial comprender ciertos conceptos fundamentales que se utilizan en la mayoría de los problemas.

10.4.1. Algoritmos de Búsqueda y Ordenación

Los algoritmos de búsqueda como la **búsqueda binaria** y los de ordenación como el **quicksort** son esenciales para optimizar la resolución de problemas.

Búsqueda Binaria

La **búsqueda binaria** es un algoritmo eficiente para encontrar un elemento en un arreglo ordenado. Funciona dividiendo el arreglo en dos partes y determinando en cuál de ellas se encuentra el elemento.

Ejemplo: Búsqueda Binaria

```
int busquedaBinaria(int arr[], int n, int target) {
    int inicio = 0, fin = n - 1;
    while (inicio <= fin) {
        int medio = inicio + (fin - inicio) / 2;
        if (arr[medio] == target) return medio;
        if (arr[medio] < target) inicio = medio + 1;
        else fin = medio - 1;
    }
    return -1; // Elemento no encontrado
}
```

Quicksort

El **quicksort** es un algoritmo eficiente de ordenación que utiliza el enfoque "divide y vencerás". Se selecciona un pivote y luego los elementos se reorganizan en torno a él.

Ejemplo: Quicksort

```
void quicksort(int arr[], int low, int high) {
    if (low < high) {
        int pivote = partition(arr, low, high);
        quicksort(arr, low, pivote - 1);
        quicksort(arr, pivote + 1, high);
    }
}
```

10.4.2. Estructuras de Datos

Las estructuras de datos son fundamentales para almacenar y manipular datos de manera eficiente. Las estructuras más utilizadas en programación competitiva incluyen:

- **Vectores y Arreglos:** Se usan para almacenar y acceder a los datos de forma secuencial.
- **Pilas y Colas:** Son útiles para resolver problemas donde se requiere el acceso y eliminación de elementos en un orden específico (LIFO y FIFO).
- **Listas Enlazadas:** Permiten una gestión dinámica de los elementos, con un acceso más eficiente que los arreglos en ciertos casos.
- **Árboles Binarios:** Útiles para almacenar datos jerárquicos y realizar búsquedas eficientes.
- **Grafos:** Usados para modelar redes de elementos interconectados, como rutas entre ciudades o conexiones en redes sociales.

Ejemplo: Pilas

Una pila es una estructura de datos que sigue el principio de Last In, First Out (LIFO). Esto significa que el último elemento agregado es el primero en ser retirado.

Ejemplo: Pila en C++

```
#include <stack>
#include <iostream>
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 << "Elemento - retirado : -" << pila.top() << endl;

    return 0;
}
```

10.5. Resolución de Problemas Clásicos

A continuación, veremos ejemplos clásicos de problemas en programación competitiva.

10.5.1. Problema: Suma de Subconjuntos

Este es un problema clásico que puede resolverse utilizando programación dinámica. Dado un conjunto de números, se debe determinar si existe un subconjunto cuya suma sea igual a un valor específico.

Ejemplo: Suma de Subconjuntos

```
bool sumaSubconjunto(int arr[], int n, int suma) {
    bool dp[n+1][suma+1];
    for (int i = 0; i <= n; i++) dp[i][0] = true;
    for (int i = 1; i <= suma; i++) dp[0][i] = false;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= suma; j++) {
            if (arr[i-1] <= j)
                dp[i][j] = dp[i-1][j] || dp[i-1][j-arr[i-1]];
            else
                dp[i][j] = dp[i-1][j];
        }
    }
    return dp[n][suma];
}
```

10.5.2. Problema: Encuentra el Camino Más Corto en un Grafo

Otro problema clásico en programación competitiva es encontrar el camino más corto entre dos nodos en un grafo. Se puede resolver utilizando el algoritmo de Dijkstra.

Ejemplo: Algoritmo de Dijkstra

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

void dijkstra(int n, vector<vector<int>>& grafo, int src) {
    vector<int> dist(n, INT_MAX);
    dist[src] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push({0, src});

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        for (int v = 0; v < n; v++) {
            if (grafo[u][v] && dist[v] > dist[u]
                + grafo[u][v]) {
                dist[v] = dist[u] + grafo[u][v];
                pq.push({dist[v], v});
            }
        }
    }

    for (int i = 0; i < n; i++) cout << "Distancia - desde -"
        << src << "- a -" << i << "- es : -" << dist[i] << endl;
}
```

10.6. Conclusión

La programación competitiva es una excelente manera de mejorar tus habilidades en algoritmos y estructuras de datos. Al participar en competiciones y resolver problemas, los programadores pueden mejorar su capacidad de pensar de manera eficiente y crear soluciones óptimas.

Ejercicios de Pilas y Colas

En esta sección, vamos a explorar algunos ejercicios prácticos que combinan el uso de pilas y colas, dos estructuras de datos fundamentales en la programación. Los siguientes ejercicios están diseñados para poner en práctica conceptos clave como inserción, eliminación y transferencia de datos entre estas estructuras.

11.1. Ejercicio 1: Implementación de Pila y Cola

En este ejercicio, implementaremos tanto una pila como una cola, y proporcionaremos un menú de opciones para manipular ambas estructuras. El programa debe permitir al usuario insertar elementos en la pila y la cola, eliminar elementos y mostrar el contenido de ambas estructuras.

11.1.1. Estructuras de Pila y Cola en C++

El código a continuación implementa las estructuras de pila y cola utilizando listas enlazadas. Además, se proporciona un menú principal para navegar entre las operaciones.

Ejemplo: Implementación de Pilas y Colas

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

// Estructura para pila
struct Nodo {
    int dato;
    Nodo *siguiente;
};

// Estructura para cola
struct nodo {
    int dato;
    nodo* siguiente;
};

// Variables globales para la cola
nodo* frente = nullptr;
nodo* fin = nullptr;

// Prototipos
void intercambiarPilaCola(Nodo *&, nodo *&, nodo *&);
void menuprincipal();
void menu_pilas(Nodo *&);
void menu_colas();
void insertar(Nodo *&, int);
void mostrar(Nodo* );
void quitar(Nodo *&, int &);
void insertarEnCola(int);
void eliminarDeCola();
void mostrarCola();
void pasarPilaACola(Nodo *&);
void pasarColaAPila(Nodo *&);

// Función principal
int main() {
    menuprincipal();
    return 0;
}

// Menú principal
void menuprincipal() {
    Nodo *pila = nullptr;
    int opcion;
    do {
        cout << "\n=====MENU PRINCIPAL===== \n";
        cout << "1. - Llamar - colas \n";
        cout << "2. - Llamar - pilas \n";
        cout << "3. - Pasar - de - pila - a - cola \n";
        cout << "4. - Pasar - de - cola - a - pila \n";
    }
```

Ejemplo:continacion

```
cout << " 5. - Intercambiar - pila - y - cola \n" ;
// NUEVA OPCIÓN
cout << " 6. - Salir \n" ;
cout << " Opcion : -" ;
cin >> opcion ;

switch (opcion) {
    case 1:
        menu_colas ();
        break;
    case 2:
        menu_pilas (pila);
        break;
    case 3:
        pasarPilaACola (pila);
        system (" pause" );
        break;
    case 4:
        pasarColaAPila (pila);
        system (" pause" );
        break;
    case 5:
        intercambiarPilaCola (pila, frente, fin);
        system (" pause" );
        break;
    case 6:
        cout << " Saliendo - del - programa . . . \n" ;
        break;
    default:
        cout << " Opcion - no - valida . \n" ;
}
system (" cls" );
} while (opcion != 6);
}
```

11.2. Ejercicio 2: Transferencia de Datos entre Pila y Cola

En este ejercicio, implementaremos funciones para transferir datos entre una pila y una cola. La idea es transferir todos los elementos de la pila a la cola y viceversa. Al hacerlo, se debe mantener el orden correcto de los elementos.

11.2.1. Código de Transferencia de Pila a Cola y de Cola a Pila

A continuación, mostramos cómo transferir los elementos entre una pila y una cola utilizando funciones adicionales.

Ejemplo: Transferir Pila a Cola y Cola a Pila

```
// Funciones de transferencia
void pasarPilaACola(Nodo *&pila) {
    Nodo* tempPila = nullptr;
    int valor;

    // Invertimos la pila para mantener orden
    Nodo* actual = pila;
    while (actual != nullptr) {
        insertar(tempPila, actual->dato);
        actual = actual->siguiente;
    }

    // Vaciamos pila original
    while (pila != nullptr) {
        quitar(pila, valor);
    }

    // Insertamos en cola
    while (tempPila != nullptr) {
        quitar(tempPila, valor);
        insertarEnCola(valor);
    }

    cout << "Pila - transferida - a - la
-----cola - exitosamente - y - vaciada .\n";
}

void pasarColaAPila(Nodo *&pila) {
    while (frente != nullptr) {
        int valor = frente->dato;
        eliminarDeCola(); // elimina nodo de la cola
        insertar(pila, valor); // agrega a pila
    }

    cout << "Cola - transferida - a
-----la - pila - exitosamente - y - vaciada .\n";
}
```

11.3. Ejercicio 3: Intercambiar Pila y Cola

Este ejercicio consiste en intercambiar los elementos de una pila y una cola. Los elementos de la pila deben transferirse a la cola, y los elementos de la cola deben transferirse a la pila. A continuación, se muestra cómo hacerlo:

Ejemplo: Intercambiar Pila y Cola

```
void intercambiarPilaCola(Nodo *&pila , nodo *&frente ,  
nodo *&fin) {  
    // Copiar pila a cola temporal  
    Nodo *tempPila = nullptr;  
    Nodo *auxPila = pila;  
    while (auxPila != nullptr) {  
        insertar(tempPila , auxPila->dato);  
        auxPila = auxPila->siguiente;  
    }  
    Nodo *invertida = nullptr;  
    int valor;  
    while (tempPila != nullptr) {  
        quitar(tempPila , valor);  
        insertar(invertida , valor);  
    }  
    // Copiar cola a pila temporal  
    nodo *auxCola = frente;  
    Nodo *colaEnPila = nullptr;  
    while (auxCola != nullptr) {  
        insertar(colaEnPila , auxCola->dato);  
        auxCola = auxCola->siguiente;  
    }  
  
    // Vaciar pila original  
    while (pila != nullptr) {  
        quitar(pila , valor);  
    }  
  
    // Vaciar cola original  
    while (frente != nullptr) {  
        eliminarDeCola();  
    }  
  
    // Insertar cola original en pila  
    Nodo *temp = colaEnPila;  
    while (temp != nullptr) {  
        insertar(pila , temp->dato);  
        temp = temp->siguiente;  
    }  
  
    // Insertar pila original (invertida) en cola  
    Nodo *temp2 = invertida;  
    while (temp2 != nullptr) {  
        insertarEnCola(temp2->dato);  
        temp2 = temp2->siguiente;  
    }  
  
    cout << "Pila -y- cola -intercambiadas -exitosamente.\n";  
}
```

Este ejercicio agrega la funcionalidad para intercambiar los elementos de la pila y la cola, utilizando funciones que transfieren los elementos de una estructura a otra y mantienen el orden correcto.

11.4. Resumen del Capítulo

Resumen del Capítulo

- Las pilas y las colas son estructuras de datos fundamentales que se utilizan para almacenar y organizar datos de manera eficiente.
- Las pilas siguen el principio **LIFO** (Last In, First Out), mientras que las colas siguen el principio **FIFO** (First In, First Out).
- Los ejercicios de este capítulo muestran cómo insertar, eliminar y transferir datos entre pilas y colas, así como intercambiar los elementos entre estas estructuras.
- Estos conceptos son esenciales en la implementación de algoritmos y aplicaciones que requieren un manejo eficiente de datos secuenciales.

Árbol Binario Simple

12.1. ¿Qué es un Árbol Binario Simple?

Un árbol binario es una estructura de datos jerárquica en la que cada nodo tiene como máximo dos hijos, denominados hijo izquierdo y hijo derecho. La principal característica de los árboles binarios es que cada nodo tiene como máximo dos hijos, lo que lo hace adecuado para muchas aplicaciones, como la búsqueda de datos, la organización jerárquica y la toma de decisiones.

En un **árbol binario simple**, cada nodo está compuesto por tres elementos:

- Un **dato**: El valor almacenado en el nodo.
- Un **hijo izquierdo**: Puntero al hijo izquierdo.
- Un **hijo derecho**: Puntero al hijo derecho.

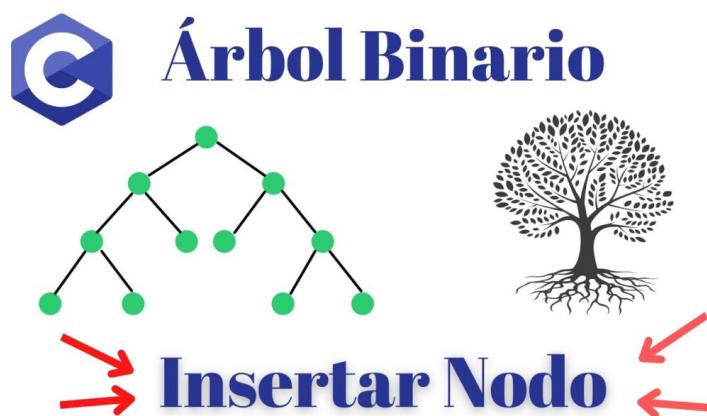


Figura: Representación de un árbol binario simple. El nodo raíz contiene el dato 1, con dos hijos: 2 a la izquierda y 3 a la derecha. Los nodos 2 y 3, a su vez, tienen sus propios hijos.

12.2. Estructura de un Árbol Binario Simple en C++

En C++, la estructura básica de un nodo de un árbol binario es la siguiente:

Ejemplo: Estructura de un Nodo en C++

```
struct Nodo {  
    int dato;           // El valor almacenado en el nodo  
    Nodo* izquierdo;  // Puntero al hijo izquierdo  
    Nodo* derecho;   // Puntero al hijo derecho  
};
```

Esta estructura define un nodo que contiene un valor entero ('dato') y dos punteros a otros nodos: uno al hijo izquierdo y otro al hijo derecho.

12.3. Operaciones Básicas en un Árbol Binario

A continuación, se describen las operaciones más comunes que se pueden realizar en un árbol binario simple:

- **Inserción:** Insertar un nuevo nodo en el árbol.
- **Recorrido (traversal):** Visitar todos los nodos del árbol. Existen tres tipos principales de recorrido:
 - **Preorden:** Se visita primero el nodo raíz, luego el hijo izquierdo y finalmente el hijo derecho.
 - **Inorden:** Se visita el hijo izquierdo, luego el nodo raíz, y por último el hijo derecho.
 - **Postorden:** Se visita primero el hijo izquierdo, luego el hijo derecho y finalmente el nodo raíz.
- **Búsqueda:** Buscar un valor específico en el árbol.
- **Eliminación:** Eliminar un nodo del árbol.

12.4. Ejemplo de Inserción en un Árbol Binario

Veamos cómo insertar elementos en un árbol binario simple. La siguiente función inserta un nuevo nodo en un árbol binario. En este caso, se inserta el nodo de manera recursiva en el árbol.

Ejemplo: Inserción de un Nodo en un Árbol Binario

```
Nodo* insertar(Nodo* raiz , int valor) {  
    // Si el rbol est vac o , crear un nuevo nodo  
    if (raiz == nullptr) {  
        Nodo* nuevoNodo = new Nodo();  
        nuevoNodo->dato = valor;  
        nuevoNodo->izquierdo = nullptr;  
        nuevoNodo->derecho = nullptr;  
        return nuevoNodo;  
    }  
  
    // Si el valor es menor que el valor del nodo actual ,  
    insertamos en el sub rbol izquierdo  
    if (valor < raiz->dato) {  
        raiz->izquierdo = insertar(raiz->izquierdo , valor);  
    }  
    // Si el valor es mayor, insertamos en el sub rbol  
    derecho  
    else {  
        raiz->derecho = insertar(raiz->derecho , valor);  
    }  
  
    return raiz ;  
}
```

Este código inserta un nuevo valor en el árbol binario siguiendo las reglas del árbol binario de búsqueda: los valores menores que el nodo actual se insertan en el subárbol izquierdo, y los valores mayores en el subárbol derecho.

12.5. Recorridos de un Árbol Binario

A continuación se muestran las funciones para realizar los tres tipos de recorrido en un árbol binario: Preorden, Inorden y Postorden.

Ejemplo: Recorridos de un Árbol Binario

```
// Recorrido en Preorden (Raiz, Izquierda , Derecha)
void preorden(Nodo* raiz) {
    if (raiz == nullptr) return;
    cout << raiz->dato << " - " ; // Visitamos el nodo raiz
    preorden(raiz->izquierdo); // Recorremos el subarbol
    izquierdo
    preorden(raiz->derecho); // Recorremos el subarbol
    derecho
}

// Recorrido en Inorden (Izquierda , Raiz , Derecha)
void inorden(Nodo* raiz) {
    if (raiz == nullptr) return;
    inorden(raiz->izquierdo); // Recorremos el subarbol
    izquierdo
    cout << raiz->dato << " - " ; // Visitamos el nodo raiz
    inorden(raiz->derecho); // Recorremos el subarbol
    derecho
}

// Recorrido en Postorden (Izquierda , Derecha , Raiz)
void postorden(Nodo* raiz) {
    if (raiz == nullptr) return;
    postorden(raiz->izquierdo); // Recorremos el subarbol
    izquierdo
    postorden(raiz->derecho); // Recorremos el subarbol
    derecho
    cout << raiz->dato << " - " ; // Visitamos el nodo raiz
}
```

Estos tres métodos permiten recorrer el árbol de diferentes maneras, dependiendo de cuándo se visita el nodo raíz. A continuación, se muestran los resultados de cada tipo de recorrido para un árbol binario con los valores 10, 5, 15, 3 y 7.

Tipo de Recorrido	Resultado
Preorden	10 5 3 7 15
Inorden	3 5 7 10 15
Postorden	3 7 5 15 10

12.6. Búsqueda en un Árbol Binario

La búsqueda en un árbol binario simple se realiza comparando el valor que estamos buscando con el valor del nodo actual. Si el valor es menor, continuamos la búsqueda en el subárbol izquierdo, y si es mayor, la búsqueda continúa en el subárbol derecho.

Ejemplo: Búsqueda en un Árbol Binario

```
bool buscar(Nodo* raiz , int valor) {
    if (raiz == nullptr) return false;
    // El valor no se encuentra
    if (raiz->dato == valor) return true;
    // El valor fue encontrado

    // Si el valor es menor, buscamos en el sub árbol
    // izquierdo
    if (valor < raiz->dato) {
        return buscar(raiz->izquierdo , valor);
    }
    // Si el valor es mayor, buscamos en el sub árbol derecho
    else {
        return buscar(raiz->derecho , valor);
    }
}
```

Esta función recursiva permite buscar un valor en el árbol binario siguiendo las reglas del árbol binario de búsqueda.

12.7. Eliminación de un Nodo en un Árbol Binario

Eliminar un nodo en un árbol binario es una operación que depende de tres casos:

1. El nodo a eliminar es una hoja (sin hijos).
2. El nodo a eliminar tiene un solo hijo.
3. El nodo a eliminar tiene dos hijos.

El caso más complicado es el de los nodos con dos hijos, ya que se debe encontrar el nodo más pequeño en el subárbol derecho (o el más grande en el subárbol izquierdo) para reemplazar el nodo eliminado.

Ejemplo: Eliminación de un Nodo

```

Nodo* eliminar(Nodo* raiz , int valor) {
    if (raiz == nullptr) return raiz;
    // El valor no se encuentra

    // Si el valor es menor, buscamos en el sub rbol izquierdo
    if (valor < raiz->dato) {
        raiz->izquierdo = eliminar(raiz->izquierdo , valor);
    }
    // Si el valor es mayor, buscamos en el sub rbol derecho
    else if (valor > raiz->dato) {
        raiz->derecho = eliminar(raiz->derecho , valor);
    }
    // Si encontramos el nodo a eliminar
    else {
        // Caso 1: El nodo tiene solo un hijo o no tiene hijos
        if (raiz->izquierdo == nullptr) {
            Nodo* temp = raiz->derecho;
            delete raiz;
            return temp;
        }
        else if (raiz->derecho == nullptr) {
            Nodo* temp = raiz->izquierdo;
            delete raiz;
            return temp;
        }

        // Caso 2: El nodo tiene dos hijos
        Nodo* temp = encontrarMinimo(raiz->derecho);
        // Encontramos el nodo m s pequeño en el sub rbol derecho
        raiz->dato = temp->dato; // Reemplazamos el valor
        del nodo a eliminar con el valor del nodo m s pequeño
        raiz->derecho = eliminar(raiz->derecho , temp->dato);
    }
    return raiz;
}

Nodo* encontrarMinimo(Nodo* raiz) {
    Nodo* actual = raiz;
    while (actual && actual->izquierdo != nullptr) {
        actual = actual->izquierdo;
    }
    return actual;
}

```

Resumen de la Sección: En este capítulo, hemos cubierto los aspectos fundamentales de los árboles binarios simples, desde su definición y estructura hasta las operaciones esenciales como inserción, búsqueda, eliminación y recorrido. Además, hemos implementado ejemplos en C++ para cada una de estas operaciones.

Resumen Visual

- Los árboles binarios son estructuras jerárquicas con nodos que tienen hasta dos hijos.
- Las operaciones básicas incluyen inserción, búsqueda, eliminación y recorrido.
- Los recorridos incluyen Preorden, Inorden y Postorden.
- La eliminación de un nodo depende de si el nodo tiene cero, uno o dos hijos.

Árboles Balanceados

13.1. ¿Qué es un Árbol Balanceado?

Un **árbol balanceado** es un tipo de árbol binario donde las alturas de las subramas izquierda y derecha de cada nodo están equilibradas. La diferencia de alturas entre las ramas izquierda y derecha de cualquier nodo no debe ser mayor que un valor específico, comúnmente 1. Este equilibrio asegura que las operaciones de inserción, eliminación y búsqueda se realicen de manera eficiente.

Importancia: Los árboles balanceados permiten que las operaciones se realicen en tiempo logarítmico $O(\log n)$, lo que optimiza el rendimiento de la estructura de datos.

13.2. Propiedades de los Árboles Balanceados

En los árboles balanceados, los nodos se estructuran de tal manera que se minimizan las desventajas de los árboles no balanceados, en los cuales las operaciones pueden llegar a ser $O(n)$ en el peor caso. A continuación, se detallan las propiedades clave de los árboles balanceados:

- **Factor de Equilibrio:** El factor de equilibrio de un nodo en un árbol binario balanceado se calcula como la diferencia de altura entre su subárbol izquierdo y su subárbol derecho. Esta diferencia debe ser ≤ 1 para un árbol AVL.
- **Altura del Árbol:** En un árbol balanceado, la altura del árbol se mantiene controlada, garantizando que las operaciones de búsqueda, inserción y eliminación se realicen en tiempo $O(\log n)$.

13.3. Árboles AVL

Un árbol **AVL** (Adelson-Velsky and Landis) es un tipo de árbol binario de búsqueda balanceado. La propiedad de los árboles AVL establece que el factor de equilibrio de cualquier nodo debe ser ≤ 1 .

Definición Formal del Factor de Equilibrio

El **factor de equilibrio** de un nodo es la diferencia entre las alturas de su subárbol izquierdo y su subárbol derecho. Si el factor de equilibrio es mayor que 1 o menor que -1, se realiza una rotación para restaurar el equilibrio.

$$\text{Factor de Equilibrio} = \text{Altura del Subárbol Izquierdo} - \text{Altura del Subárbol Derecho}$$

Si el **factor de equilibrio** de un nodo es:

- 0: El árbol está equilibrado en ese nodo.
- 1: La subrama izquierda es más alta que la subrama derecha.
- -1: La subrama derecha es más alta que la subrama izquierda.

13.4. Rotaciones en Árboles AVL

Cuando un árbol AVL pierde su equilibrio, se deben realizar **rotaciones** para restablecer el equilibrio. Existen cuatro tipos de rotaciones:

1. Rotación Simple a la Derecha (Right Rotation)
2. Rotación Simple a la Izquierda (Left Rotation)
3. Rotación Doble a la Derecha (Left-Right Rotation)
4. Rotación Doble a la Izquierda (Right-Left Rotation)

Ejemplo: Rotación Simple a la Derecha

```
Nodo* rotacionDerecha(Nodo* y) {
    Nodo* x = y->izquierda;
    Nodo* T2 = x->derecha;

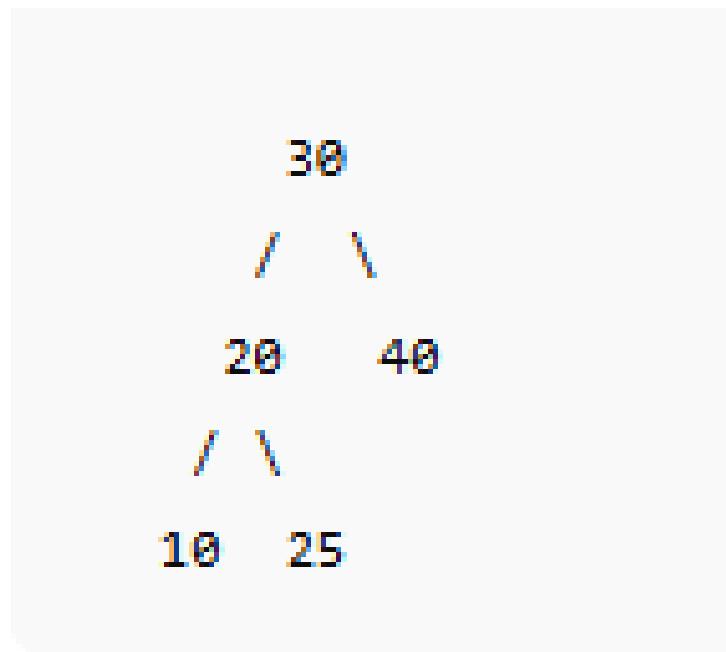
    // Realizamos la rotación
    x->derecha = y;
    y->izquierda = T2;

    // Actualizamos las alturas
    y->altura = max(obtenerAltura(y->izquierda),
                      obtenerAltura(y->derecha)) + 1;

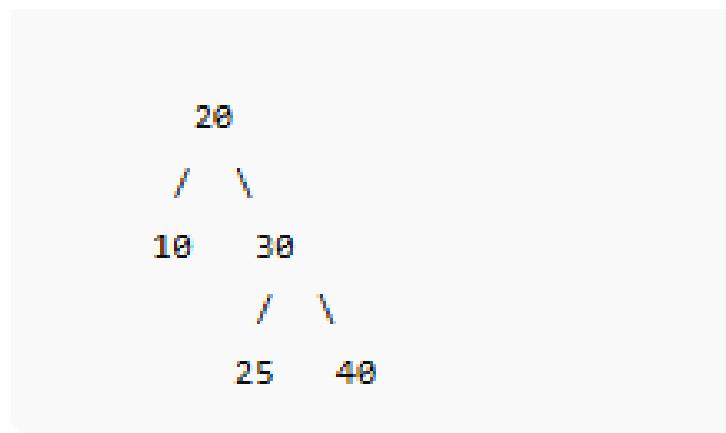
    x->altura = max(obtenerAltura(x->izquierda),
                      obtenerAltura(x->derecha)) + 1;

    return x; // Nueva raíz
}
```

Este es un ejemplo básico de la rotación simple a la derecha en C++.



Después de aplicar la rotación a la derecha sobre el nodo 30, el árbol queda equilibrado.



Código de Implementación de Árbol AVL

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

struct Nodo {
    int dato;
    Nodo *izquierda;
    Nodo *derecha;
    int altura;
};

// Funciones para obtener la altura, el factor de equilibrio, y las rotaciones

// (Implementación similar a lo que se describió antes)

int obtenerAltura(Nodo* nodo) {
    if (nodo == nullptr) return 0;
    return nodo->altura;
}

int obtenerFactorEquilibrio(Nodo* nodo) {
    if (nodo == nullptr) return 0;
    return obtenerAltura(nodo->izquierda) - obtenerAltura(nodo->derecha);
}

// Inserción y rotaciones se implementan aquí...

int main() {
    Nodo* root = nullptr;

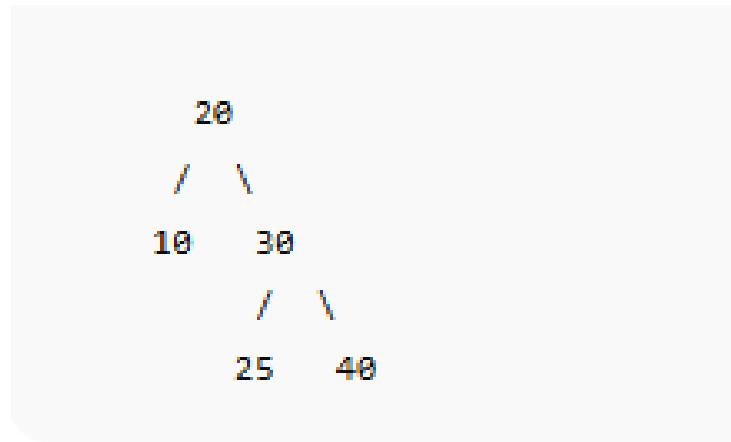
    // Insertar nodos y realizar rotaciones para mantener el equilibrio del árbol
    root = insertar(root, 10);
    root = insertar(root, 20);
    root = insertar(root, 30);
    root = insertar(root, 15);
    root = insertar(root, 25);

    cout << "Recorrido Inorden: ";
    inOrden(root);

    return 0;
}
```

Gráfico de un Árbol AVL Balanceado:

Arbol AVL Balanceado Despues de las Rotaciones:



Árboles B / B+

14.1. Definición de Árboles B

Un **árbol B** es una estructura de datos de árbol balanceado, generalmente utilizada para almacenar datos en sistemas de bases de datos y sistemas de archivos. Los árboles B son un tipo de árbol de búsqueda auto-balanceado en el que cada nodo puede tener más de dos hijos. Los árboles B son especialmente eficientes para el acceso a datos en dispositivos de almacenamiento como discos duros, ya que minimizan el número de accesos a los discos.

14.1.1. Características de los Árboles B

1. **Balanceado:** Todos los nodos hoja se encuentran a la misma profundidad, garantizando una altura mínima del árbol.
2. **Múltiples hijos:** A diferencia de los árboles binarios, un nodo en un árbol B puede tener más de dos hijos.
3. **Ordenado:** Los datos dentro de cada nodo están ordenados, y los nodos tienen un rango definido de elementos que pueden almacenar.
4. **Eficiencia:** Debido a que cada nodo puede tener múltiples elementos, los árboles B tienen una estructura muy eficiente para operaciones de búsqueda, inserción y eliminación.

14.1.2. Reglas de los Árboles B

Un árbol B de orden **m** cumple con las siguientes propiedades:

- Cada nodo tiene al menos $\lceil m/2 \rceil$ hijos (excepto la raíz, que puede tener menos).
- Cada nodo puede tener hasta m hijos.
- Los nodos internos almacenan las claves que permiten dividir el espacio de búsqueda entre los hijos. Las claves dentro de un nodo están ordenadas de manera ascendente.
- Las claves dentro de un nodo son tales que cualquier clave en el subárbol izquierdo de una clave es menor que ella, y cualquier clave en el subárbol derecho es mayor.

14.1.3. Operaciones en Árboles B

Las operaciones básicas en un árbol B incluyen:

- **Búsqueda:** Se realiza una búsqueda de forma similar a un árbol binario de búsqueda, pero comparando claves dentro de los nodos y decidiendo en qué subárbol continuar la búsqueda.

- **Inserción:** La inserción en un árbol B puede requerir la división de nodos si el nodo está lleno. Esto se realiza de manera recursiva hasta que se encuentra el lugar adecuado para la clave.

- **Eliminación:** La eliminación es más compleja y puede implicar la fusión de nodos, el préstamo de claves de nodos hermanos, o la reorganización de nodos.

14.1.4. Ejemplo de Inserción en un Árbol B

Consideremos un árbol B de orden 3. Un nodo puede tener hasta 2 claves y hasta 3 hijos.

- Inserción de 10, 20, 30, 40, 50: La inserción en un árbol B se realiza de manera ordenada. Si un nodo está lleno, se divide y se promueve una clave al nodo superior.

14.1.5. Gráfico de un Árbol B de Orden 3

14.2. Árboles B+

Un **árbol B+** es una variante de los árboles B que se utiliza ampliamente en sistemas de bases de datos. La principal diferencia es que, en un árbol B+, **todas las claves se almacenan en las hojas**. Los nodos internos sirven únicamente para dirigir la búsqueda. Esto tiene varias ventajas, especialmente cuando se realizan rangos de búsqueda o recorridos en los datos.

14.2.1. Características de los Árboles B+

1. **Solo las hojas contienen datos:** Los nodos internos contienen solo claves, mientras que las hojas contienen los datos completos o referencias a los datos.

2. **Conexión entre hojas:** Las hojas en un árbol B+ están conectadas entre sí en una lista enlazada. Esto facilita las búsquedas de rango, ya que es posible recorrer las hojas de manera eficiente.

3. **Alto rendimiento en consultas de rango:** Debido a que todas las claves están almacenadas en las hojas y están ordenadas, las consultas de rango pueden realizarse de manera muy eficiente.

14.2.2. Gráfico de un Árbol B+

14.3. Operaciones en Árboles B+

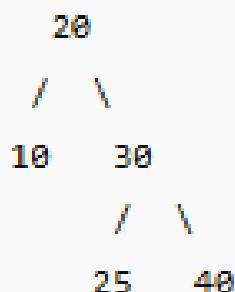
Las operaciones en un árbol B+ son muy similares a las de los árboles B, pero con algunas diferencias importantes:

1. **Búsqueda:** La búsqueda se realiza como en un árbol B, pero siempre se lleva a cabo en las hojas, ya que las hojas son las que contienen los datos completos. 2. **Inserción:** La inserción en un árbol B+ también puede requerir la división de nodos y la promoción de claves, pero siempre las claves se insertan en las hojas. 3. **Eliminación:** La

eliminación de un nodo se realiza de manera similar a la de un árbol B, pero al eliminar un nodo hoja, se debe reorganizar la lista enlazada de hojas si es necesario.

14.4. Código C++ para Implementación de un Árbol B

Aquí tienes un ejemplo básico de implementación de un árbol B para realizar operaciones de inserción y búsqueda:



Ejemplo: Rotación Simple a la Derecha

```
Nodo* rotacionDerecha(Nodo* y) {
    #include <iostream>
    #include <vector>
    using namespace std;

    class Nodo {
    public:
        vector<int> claves;
        vector<Nodo*> hijos;
        bool hoja;

        Nodo(bool hoja) {
            this->hoja = hoja;
        }
    };
    class ArbolB {
    private:
        Nodo* raiz;
        int orden;

    public:
        ArbolB(int orden) {
            this->orden = orden;
            raiz = new Nodo(true);
        }

        void insertar(int clave) {
            Nodo* nodo = raiz;
            if (nodo->claves.size() == orden - 1) {
                Nodo* nuevoNodo = new Nodo(false);
                raiz = nuevoNodo;
                nuevoNodo->hijos.push_back(nodo);
                dividir(nuevoNodo, 0);
            }
            insertarNoLleno(raiz, clave);
        }

        void insertarNoLleno(Nodo* nodo, int clave) {
            int i = nodo->claves.size() - 1;
            if (nodo->hoja) {
                nodo->claves.push_back(clave);
                sort(nodo->claves.begin(), nodo->claves.end());
            } else {
                while (i >= 0 && clave < nodo->claves[i]) {
                    i--;
                }
                i++;
                if (nodo->hijos[i]->claves.size() == orden - 1) {
                    dividir(nodo, i);
                    if (clave > nodo->claves[i]) {
                        i++;
                    }
                }
                insertarNoLleno(nodo->hijos[i], clave);
            }
        }
    };
}
```

Árboles Heap

15.1. Definición y Concepto

Un **árbol Heap** es una estructura de datos completa en forma de árbol binario en la que se cumplen las siguientes propiedades:

- **Propiedad de Completeness:** Un árbol Heap debe ser completo, lo que significa que todos los niveles del árbol están completamente llenos, excepto tal vez el último nivel, que debe ser llenado de izquierda a derecha.
- **Propiedad de Heap:** Dependiendo del tipo de Heap, el valor de cada nodo debe ser mayor o menor que los de sus hijos:
 - **Heap Máximo:** El valor de un nodo debe ser mayor o igual que el valor de sus hijos. Es decir, el nodo raíz tiene el valor máximo en el árbol.
 - **Heap Mínimo:** El valor de un nodo debe ser menor o igual que el valor de sus hijos. Es decir, el nodo raíz tiene el valor mínimo en el árbol.

15.2. Propiedades Importantes

Los árboles Heap tienen varias propiedades que los hacen útiles en diversas aplicaciones:

- **Eficiencia en la Inserción y Eliminación:** La inserción y eliminación de elementos en un árbol Heap son operaciones eficientes, especialmente para operaciones de prioridad.
- **Acceso al Elemento Máximo o Mínimo:** En un Heap máximo, el acceso al valor máximo se realiza en tiempo constante ($O(1)$), y lo mismo ocurre en un Heap mínimo para el valor mínimo.
- **Estructura Completa:** La estructura de árbol binario completo garantiza que el árbol esté equilibrado, lo que lleva a tiempos logarítmicos en operaciones como la inserción y eliminación.

15.3. Operaciones en Árboles Heap

Las operaciones básicas en un árbol Heap incluyen:

15.3.1. Inserción

Para insertar un elemento en un Heap, se sigue el siguiente procedimiento:

1. Agregar el elemento al final del árbol. Esto asegura que el árbol sigue siendo completo.

2. Restaurar la propiedad del Heap. Después de insertar, el árbol puede dejar de cumplir la propiedad del Heap, por lo que debemos restaurarla. Esto se hace mediante el proceso de heapificación hacia arriba (o up-heapify), que compara el nodo insertado con su padre y lo intercambia si es necesario, repitiendo el proceso hasta que la propiedad del Heap se restaure.

15.3.2. Eliminación (Eliminación del Raíz)

La eliminación de la raíz de un Heap (ya sea máximo o mínimo) se realiza de la siguiente forma:

1. Reemplazar la raíz con el último nodo. El último nodo del árbol (en el último nivel) se mueve a la raíz. 2. Restaurar la propiedad del Heap. Se realiza el proceso de heapificación hacia abajo (o down-heapify), que compara el nodo con sus hijos y lo intercambia con el hijo más grande (en un Heap máximo) o más pequeño (en un Heap mínimo). Este proceso se repite hasta que el árbol cumple nuevamente la propiedad de Heap.

15.3.3. Heapificación

La heapificación es el proceso de reorganizar el árbol para que cumpla la propiedad del Heap. Esto puede hacerse tanto hacia arriba (cuando se inserta un nuevo nodo) como hacia abajo (cuando se elimina la raíz).

- Heapificación hacia arriba: Se utiliza durante la inserción de un nodo. Comienza desde el nodo recién insertado y lo compara con su parente, realizando intercambios hasta que el árbol cumpla con la propiedad de Heap.

- Heapificación hacia abajo: Se utiliza después de eliminar la raíz o durante la construcción de un Heap. Comienza desde la raíz y compara el nodo con sus hijos, realizando intercambios con el hijo más grande o más pequeño hasta que el árbol cumpla con la propiedad de Heap.

15.4. Representación de un Árbol Heap

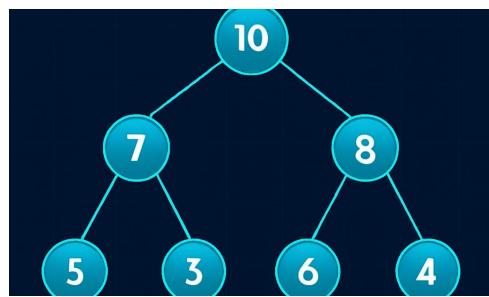
La representación más común de un árbol Heap es mediante un arreglo o vector. En este arreglo:

- El elemento en la posición 0 es la raíz del árbol.
- Los elementos en las posiciones 1 y 2 son los hijos de la raíz.
- En general, el elemento en la posición i tiene:
 - Su hijo izquierdo en la posición $2i + 1$.
 - Su hijo derecho en la posición $2i + 2$.
 - Su parente en la posición $\left\lfloor \frac{i-1}{2} \right\rfloor$.

15.5. Ejemplo de Árbol Heap

A continuación, se presenta un ejemplo visual de un árbol Heap máximo:

Árbol Heap Máximo :



Explicación: En este árbol Heap máximo, la raíz es el elemento máximo del árbol, y cada nodo padre es mayor o igual que sus hijos.

15.6. Árbol Heap en la Práctica

Los árboles Heap son comúnmente utilizados en aplicaciones como:

- Colas de prioridad: Las colas de prioridad son estructuras de datos que permiten extraer el elemento con la mayor (o menor) prioridad. Los Heaps son ideales para implementarlas debido a su eficiencia en la inserción y eliminación de elementos.
- Algoritmo de Heapsort: Heapsort es un algoritmo de ordenación basado en árboles Heap. Funciona construyendo un Heap máximo (o mínimo) a partir de los datos y luego extrayendo repetidamente el elemento máximo (o mínimo), lo que produce una secuencia ordenada.
- Estructuras de datos balanceadas: Los Heaps también se utilizan en otras estructuras de datos balanceadas, como los árboles binarios balanceados, para asegurar un tiempo de acceso eficiente.

15.7. Implementación de un Árbol Heap en C++

A continuación, se presenta la implementación básica de un Heap máximo utilizando un arreglo:

implementación básica de un Heap máximo

```
Nodo* rotacionDerecha(Nodo* y) {  
  
    #include <iostream>  
    #include <vector>  
    using namespace std;  
  
    // Función para realizar la heapificación hacia abajo  
    void heapify(vector<int>& heap, int n, int i) {  
        int largest = i; // Inicializamos el mayor como raíz  
        int left = 2 * i + 1; // Hijo izquierdo  
        int right = 2 * i + 2; // Hijo derecho  
  
        // Si el hijo izquierdo es más grande que la raíz  
        if (left < n && heap[left] > heap[largest]) {  
            largest = left;  
        }  
  
        // Si el hijo derecho es más grande que el mayor hasta ahora  
        if (right < n && heap[right] > heap[largest]) {  
            largest = right;  
        }  
  
        // Si el mayor no es la raíz, intercambiamos y seguimos heapificando  
        if (largest != i) {  
            swap(heap[i], heap[largest]);  
            heapify(heap, n, largest);  
        }  
    }  
  
    // Función para insertar un nuevo valor en el Heap  
    void insert(vector<int>& heap, int value) {  
        heap.push_back(value); // Insertamos el nuevo valor en el final  
        int i = heap.size() - 1;  
  
        // Restauramos la propiedad de Heap moviendo el valor hacia arriba  
        while (i != 0 && heap[(i - 1) / 2] < heap[i]) {  
            swap(heap[i], heap[(i - 1) / 2]);  
            i = (i - 1) / 2;  
        }  
    }  
  
    // Función para eliminar el valor máximo (raíz) del Heap  
    int extractMax(vector<int>& heap) {  
        if (heap.size() <= 1) return heap[0];  
  
        int root = heap[0];  
        heap[0] = heap.back();  
        heap.pop_back();  
        heapify(heap, heap.size(), 0);  
  
        return root;  
    }  
  
    int main() {  
        // Implementación de la función main  
    }  
}
```

Árboles Rojo-Negro

16.1. Definición y Concepto

Un Árbol Rojo-Negro es un tipo especial de árbol binario de búsqueda equilibrado, que asegura un buen rendimiento en operaciones de búsqueda, inserción y eliminación, al mantener el árbol balanceado mediante un conjunto de reglas que definen el color de los nodos. Cada nodo del árbol puede ser de color rojo o negro, y estas reglas son las que permiten que el árbol se mantenga equilibrado y, por lo tanto, garantizar un tiempo de ejecución logarítmico en las operaciones más comunes.

Propiedades de los Árboles Rojo-Negro:

Los árboles Rojo-Negro tienen las siguientes propiedades clave:

1. Cada nodo es rojo o negro.
2. La raíz es negra.
3. Cada hoja (nulo) es negra.

4. Si un nodo es rojo, entonces sus hijos son negros (no puede haber dos nodos rojos consecutivos).

5. Para cada nodo, todos los caminos desde ese nodo hasta las hojas descendientes tienen el mismo número de nodos negros (esto se conoce como el camino negro.^o "black-height").

Estas propiedades son esenciales para que el árbol mantenga un balance adecuado, garantizando una altura logarítmica con respecto al número de elementos, lo que implica un tiempo de operación $O(\log n)$ para las operaciones más comunes.

16.2. Operaciones en Árboles Rojo-Negro

Los árboles Rojo-Negro soportan las siguientes operaciones fundamentales:

16.2.1. Inserción

La inserción en un árbol Rojo-Negro sigue el proceso de inserción estándar en un árbol binario de búsqueda, pero con algunos pasos adicionales para restaurar las propiedades del árbol Rojo-Negro. El algoritmo de inserción se realiza en dos fases:

1. Inserción estándar en un árbol binario de búsqueda. El nuevo nodo se inserta como un nodo rojo, siguiendo las reglas del árbol binario de búsqueda.
2. Restauración de las propiedades del árbol Rojo-Negro. Después de la inserción, el árbol puede violar alguna de las propiedades del árbol Rojo-Negro. La restauración se

realiza mediante una serie de rotaciones y cambios de color, conocidas como **rotaciones izquierda y derecha.

16.2.2. Eliminación

La eliminación en un árbol Rojo-Negro es más compleja debido a la necesidad de restaurar el balance del árbol tras la eliminación de un nodo. Al igual que la inserción, se realiza una eliminación estándar en un árbol binario de búsqueda, y luego se corrige cualquier violación de las propiedades del árbol mediante una serie de rotaciones y cambios de color. La eliminación se realiza en tres fases:

1. Eliminación del nodo: El nodo a eliminar se elimina de manera similar a la eliminación en un árbol binario de búsqueda.
2. Restauración de las propiedades del árbol Rojo-Negro**: El árbol puede violar alguna de las propiedades del árbol Rojo-Negro tras la eliminación. Para restaurar el balance, se aplican rotaciones y cambios de color.
3. Ajustes de color y rotaciones**: Si se eliminó un nodo negro, esto podría afectar la propiedad de los caminos negros. En este caso, se hacen ajustes mediante rotaciones y cambios de color para restaurar la propiedad del árbol.

16.2.3. Búsqueda

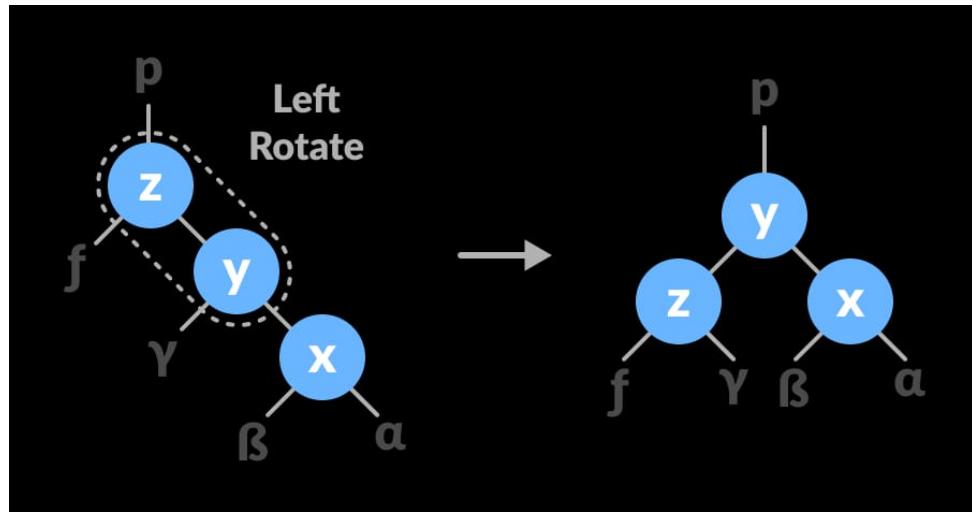
La búsqueda en un árbol Rojo-Negro sigue el mismo proceso que en cualquier otro árbol binario de búsqueda. El tiempo de búsqueda es $O(\log n)$, ya que la altura del árbol se mantiene equilibrada debido a las propiedades del árbol Rojo-Negro.

16.3. Rotaciones en Árboles Rojo-Negro

Las rotaciones son operaciones clave en los árboles Rojo-Negro para mantener el balance del árbol. Existen dos tipos de rotaciones:

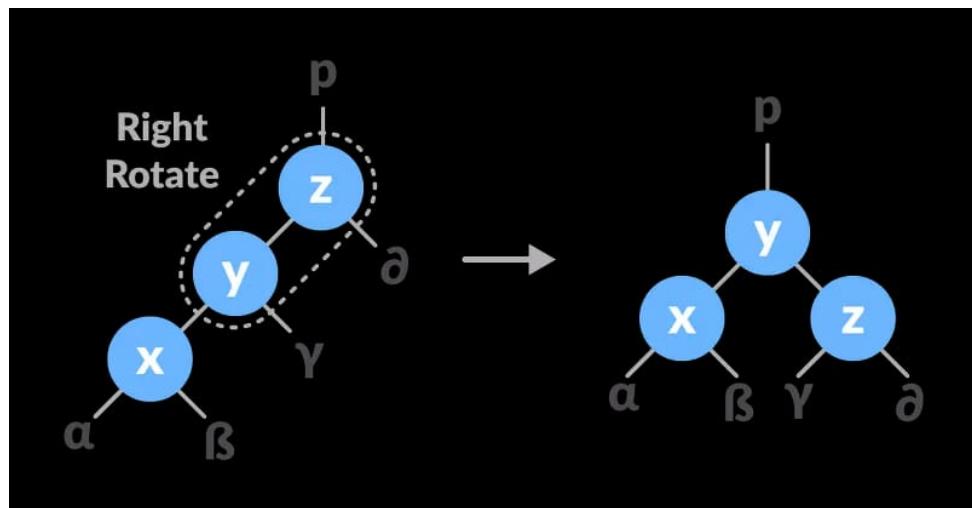
16.3.1. Rotación a la Izquierda

La rotación a la izquierda es una operación que toma un nodo y lo mueve hacia la izquierda, mientras ajusta las relaciones entre sus nodos hijos. Se utiliza cuando un nodo rojo tiene un hijo derecho rojo, lo que viola la regla que establece que no puede haber dos nodos rojos consecutivos.



16.3.2. Rotación a la Derecha

La rotación a la derecha es el proceso inverso de la rotación a la izquierda. Se utiliza cuando un nodo rojo tiene un hijo izquierdo rojo, lo que también viola la propiedad de los árboles Rojo-Negro.



16.4. Implementación de un Árbol Rojo-Negro en C++

A continuación, se presenta la implementación básica de un árbol Rojo-Negro en C++. Esta implementación cubre las operaciones básicas de inserción y rotaciones.

implementación básica de un árbol Rojo-Negro en C++.

```
{  
#include <iostream>  
using namespace std;  
  
enum Color { RED, BLACK };  
  
struct Node {  
    int data;  
    Node *left, *right, *parent;  
    Color color;  
};  
  
class RedBlackTree {  
private:  
    Node* root;  
  
protected:  
    void leftRotate(Node*& root, Node*& x) {  
        Node* y = x->right;  
        x->right = y->left;  
  
        if (y->left != nullptr) {  
            y->left->parent = x;  
        }  
        y->parent = x->parent;  
  
        if (x->parent == nullptr) {  
            root = y;  
        } else if (x == x->parent->left) {  
            x->parent->left = y;  
        } else {  
            x->parent->right = y;  
        }  
  
        y->left = x;  
        x->parent = y;  
    }  
  
    void rightRotate(Node*& root, Node*& y) {  
        Node* x = y->left;  
        y->left = x->right;  
  
        if (x->right != nullptr) {  
            x->right->parent = y;  
        }  
        x->parent = y->parent;  
  
        if (y->parent == nullptr) {  
            root = x;  
        }  
    }  
};
```

continuacion...

```
    } else if (y == y->parent->right) {
        y->parent->right = x;
    } else {
        y->parent->left = x;
    }

    x->right = y;
    y->parent = x;
}

public:
    RedBlackTree() {
        root = nullptr;
    }

    // Función de inserción en el árbol
    void insert(int data) {
        Node* newNode = new Node();
        newNode->data = data;
        newNode->color = RED;
        newNode->left = newNode->right = newNode->parent = nullptr;

        // Inserción estándar en un BST
        Node* y = nullptr;
        Node* x = root;

        while (x != nullptr) {
            y = x;
            if (newNode->data < x->data) {
                x = x->left;
            } else {
                x = x->right;
            }
        }

        newNode->parent = y;
        if (y == nullptr) {
            root = newNode;
        } else if (newNode->data < y->data) {
            y->left = newNode;
        } else {
            y->right = newNode;
        }

        // Restaurar las propiedades del árbol Rojo-Negro
        fixInsert(root, newNode);
    }
}
```

continuacion...

```
// Método para restaurar las propiedades del árbol Rojo-Negro
void fixInsert(Node*& root, Node*& newNode) {
    Node* uncle;
    while (newNode->parent && newNode->parent->color == RED) {
        if (newNode->parent == newNode->parent->parent->left) {
            uncle = newNode->parent->parent->right;
            if (uncle && uncle->color == RED) {
                newNode->parent->color = BLACK;
                uncle->color = BLACK;
                newNode->parent->parent->color = RED;
                newNode = newNode->parent->parent;
            } else {
                if (newNode == newNode->parent->right) {
                    newNode = newNode->parent;
                    leftRotate(root, newNode);
                }
                newNode->parent->color = BLACK;
                newNode->parent->parent->color = RED;
                rightRotate(root, newNode->parent->parent);
            }
        } else {
            uncle = newNode->parent->parent->left;
            if (uncle && uncle->color == RED) {
                newNode->parent->color = BLACK;
                uncle->color = BLACK;
                newNode->parent->parent->color = RED;
                newNode = newNode->parent->parent;
            } else {
                if (newNode == newNode->parent->left) {
                    newNode = newNode->parent;
                    rightRotate(root, newNode);
                }
            }
        }
    }
}
```

continuacion...

```
        newNode->parent->color = BLACK;
        newNode->parent->parent->color = RED;
        leftRotate(root, newNode->parent->parent);
    }
}
root->color = BLACK;
}

// Función para mostrar el árbol en orden
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}

void display() {
    inorder(root);
}
};

int main() {
    RedBlackTree tree;
    tree.insert(10);
    tree.insert(20);
    tree.insert(30);
    tree.insert(15);

    cout << "Árbol Rojo-Negro en orden: ";
    tree.display();
    return 0;
}
```


Glosario

16.5. Términos en Español

- **Algoritmo:** Un conjunto de pasos o instrucciones finitas que resuelven un problema o realizan una tarea.
- **Recursividad:** Técnica de programación en la que una función se llama a sí misma para resolver problemas de manera más eficiente.
- **Pila:** Estructura de datos que sigue el principio **LIFO** (Last In, First Out), donde el último elemento agregado es el primero en ser removido.
- **Cola:** Estructura de datos que sigue el principio **FIFO** (First In, First Out), donde el primer elemento agregado es el primero en ser removido.
- **Algoritmos de Ordenación:** Algoritmos que reorganizan los elementos de una lista o arreglo en un orden específico (ascendente o descendente).
- **Burbuja (Bubble Sort):** Algoritmo de ordenación que compara y organiza los elementos adyacentes, repitiendo el proceso hasta que el arreglo esté ordenado.
- **Selección (Selection Sort):** Algoritmo de ordenación que selecciona el elemento más pequeño y lo coloca en la posición correcta, repitiendo el proceso para el resto del arreglo.
- **Inserción (Insertion Sort):** Algoritmo de ordenación que construye el arreglo ordenado insertando cada nuevo elemento en la posición correcta.
- **Ordenación rápida (Quick Sort):** Algoritmo eficiente de ordenación que divide y organiza los elementos alrededor de un pivote.

16.6. Terms in English

- **Algorithm:** A set of finite steps or instructions that solve a problem or perform a task.
- **Recursion:** A programming technique where a function calls itself to solve problems more efficiently.
- **Stack:** A data structure that follows the **LIFO** (Last In, First Out) principle, where the last element added is the first to be removed.

- **Queue:** A data structure that follows the **FIFO** (First In, First Out) principle, where the first element added is the first to be removed.
- **Sorting Algorithms:** Algorithms that rearrange the elements of a list or array in a specific order (ascending or descending).
- **Bubble Sort:** A sorting algorithm that compares and sorts adjacent elements, repeating the process until the array is sorted.
- **Selection Sort:** A sorting algorithm that selects the smallest element and places it in the correct position, repeating the process for the rest of the array.
- **Insertion Sort:** A sorting algorithm that builds the sorted array by inserting each new element in the correct position.
- **Quick Sort:** An efficient sorting algorithm that divides and organizes the elements around a pivot.

Solucionario

16.7. Ejercicios del Capítulo 1: Introducción a la Programación en C++

- **Ejercicio 1: "Hola Mundo"**: El programa debe mostrar en pantalla el mensaje "Hola Mundo!".

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

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

- **Ejercicio 2: Nombre del Usuario**: El programa debe pedir al usuario su nombre e imprimirla en pantalla.

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

int main() {
    string nombre;
    cout << " Cul - es - tu - nombre? - ";
    cin >> nombre;
    cout << "Hola , -" << nombre << " ! " << endl;
    return 0;
}
```

- **Ejercicio 3: Cálculo de Edad en 5 Años**: El programa debe calcular cuántos años tendrá el usuario en 5 años.

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

int main() {
    int edad;
    cout << " Cuntos - a os - tienes? - ";
    cin >> edad;
    cout << "En - 5 - a os - tendr s -" << edad + 5 << " - a os ." << endl;
```

```

    return 0;
}

```

16.8. Ejercicios del Capítulo 2: Arrays

- **Ejercicio 1: Invertir el Arreglo:** El programa debe invertir un arreglo de 10 números y mostrarlo.

```

#include <iostream>
using namespace std;

int main() {
    int arr[10];
    cout << "Ingresa 10 números: ";
    for (int i = 0; i < 10; i++) {
        cin >> arr[i];
    }
    cout << "Arreglo invertido: ";
    for (int i = 9; i >= 0; i--) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}

```

- **Ejercicio 2: Mayor de Números:** El programa debe encontrar el número mayor entre 5 valores ingresados por el usuario.

```

#include <iostream>
using namespace std;

int main() {
    int arr[5], mayor;
    cout << "Ingresa 5 números: ";
    for (int i = 0; i < 5; i++) {
        cin >> arr[i];
    }
    mayor = arr[0];
    for (int i = 1; i < 5; i++) {
        if (arr[i] > mayor) {
            mayor = arr[i];
        }
    }
    cout << "El mayor número es: " << mayor << endl;
    return 0;
}

```

- **Ejercicio 3: Precio Total de Productos:** El programa debe calcular el total de precios de 4 productos y mostrar el resultado.

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

int main() {
    float arr[4], total = 0;
    cout << "Ingresa el precio de 4 productos: ";
    for (int i = 0; i < 4; i++) {
        cin >> arr[i];
        total += arr[i];
    }
    cout << "El total a pagar es: " << total << endl;
    return 0;
}
```

16.9. Ejercicios del Capítulo 3: Pilas

- **Ejercicio 1: Pila de Enteros:** El programa debe crear una pila de enteros y realizar operaciones de inserción y eliminación.

```
#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 << "Elemento retirado: " << pila.top() << endl;

    pila.pop();
    cout << "Elemento retirado: " << pila.top() << endl;

    return 0;
}
```

- **Ejercicio 2: Factorial Recursivo:** El programa debe calcular el factorial de un número utilizando recursividad.

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

int factorial(int n) {
```

```

if (n == 0) return 1;
return n * factorial(n - 1);
}

int main() {
    int n;
    cout << "Ingresa un n mero para calcular su factorial : ";
    cin >> n;
    cout << "El factorial de " << n << " es : " << factorial(n)
    << endl;
    return 0;
}

```

- **Ejercicio 3: Búsqueda en Árbol Binario:** El programa debe buscar un valor en un árbol binario.

```

#include <iostream>
using namespace std;

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

bool buscar(Nodo* raiz, int valor) {
    if (raiz == nullptr) return false;
    if (raiz->dato == valor) return true;
    return buscar(raiz->izquierdo, valor) || buscar(raiz->derecho,
    valor);
}

int main() {
    Nodo* raiz = new Nodo{10, nullptr, nullptr};
    raiz->izquierdo = new Nodo{5, nullptr, nullptr};
    raiz->derecho = new Nodo{15, nullptr, nullptr};

    cout << " El valor 5 est en el rbol ? " << (buscar(raiz, 5)
    ? " S " : "No") << endl;

    return 0;
}

```

16.10. Ejercicios del Capítulo 4: Colas

- **Ejercicio 1: Cola de Atención al Cliente:** El programa debe simular una cola de atención al cliente.

```

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

int main() {
    queue<string> cola;

    cola.push("Juan");
    cola.push("Maria");
    cola.push("Pedro");

    cout << "Atendiendo a: " << cola.front() << endl;
    cola.pop();

    cout << "Atendiendo a: " << cola.front() << endl;
    cola.pop();

    return 0;
}

```

- **Ejercicio 2: Cola Circular:** El programa debe crear una cola circular que reutilice espacio cuando se eliminan elementos.

```

#include <iostream>
using namespace std;

#define MAX 5
class ColaCircular {
private:
    int cola[MAX];
    int frente, fin;

public:
    ColaCircular() {
        frente = -1;
        fin = -1;
    }

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

    bool estaLlena() {
        return (fin + 1) % MAX == frente;
    }

    void enqueue(int valor) {
        if (estaLlena()) {
            cout << "La cola circular est llena." << endl;
        }
    }
}

```

```

        return ;
    }
    if (frente == -1) frente = 0;
    fin = (fin + 1) % MAX;
    cola[fin] = valor ;
}

int dequeue() {
    if (estaVacia()) {
        cout << "La -cola -circular -est - vac a ." << endl;
        return -1;
    }
    int valor = cola[frente];
    if (frente == fin) {
        frente = fin = -1;
    } else {
        frente = (frente + 1) % MAX;
    }
    return valor;
}
};

int main() {
    ColaCircular cola;

    cola.enqueue(10);
    cola.enqueue(20);
    cola.enqueue(30);
    cola.enqueue(40);

    cout << "Elemento - retirado : -" << cola.dequeue() << endl;

    return 0;
}

```

- **Ejercicio 3: Cola de Impresión:** El programa debe simular una cola de impresión.

```

#include <iostream>
using namespace std;

struct TrabajoImpresion {
    string documento;
    int paginas;
};

class ColaImpresion {
private:
    TrabajoImpresion cola[MAX];
    int frente , fin ;

```

```
public:
    ColaImpresion() {
        frente = -1;
        fin = -1;
    }

    void agregarTrabajo(string documento, int paginas) {
        if (fin == MAX - 1) {
            cout << "La cola de impresion est llena." << endl;
            return;
        }
        if (frente == -1) frente = 0;
        cola[++fin] = {documento, paginas};
    }

    void procesarTrabajo() {
        if (frente == -1) {
            cout << "No hay trabajos en la cola." << endl;
            return;
        }
        cout << "Imprimiendo: " << cola[frente].documento << " - (" 
        << cola[frente].paginas << " - paginas)" << endl;
        if (frente == fin) frente = fin = -1;
        else frente++;
    }

    void mostrarCola() {
        if (frente == -1) {
            cout << "No hay trabajos pendientes." << endl;
            return;
        }
        for (int i = frente; i <= fin; i++) {
            cout << cola[i].documento << " - (" << cola[i].paginas << " - paginas) - - - ";
        }
        cout << endl;
    }
};

int main() {
    ColaImpresion cola;

    cola.agregarTrabajo("Documento1", 10);
    cola.agregarTrabajo("Documento2", 20);
    cola.procesarTrabajo();

    return 0;
}
```

{}

Glosario de la Segunda Unidad

Árbol Binario

Un **árbol binario** es una estructura de datos en la que cada nodo tiene a lo sumo dos hijos. Los nodos se organizan de forma jerárquica con un único nodo raíz.

Propiedades:

- Cada nodo tiene un valor y un puntero a sus hijos izquierdo y derecho.
- Se utilizan para representar estructuras jerárquicas o tomar decisiones (como en los árboles de decisión).

Árbol Binario de Búsqueda (BST)

Un **árbol binario de búsqueda (BST)** es un tipo especial de árbol binario en el que, para cada nodo, los valores de los nodos en su subárbol izquierdo son menores y los de su subárbol derecho son mayores.

Propiedades:

- Permite realizar búsquedas, inserciones y eliminaciones eficientes.

Árbol AVL

Un **árbol AVL** es un árbol binario de búsqueda auto-balanceado. Su diferencia clave es que, para cualquier nodo, la diferencia de alturas entre sus subárboles izquierdo y derecho no puede ser mayor a 1.

Propiedades:

- Garantiza una búsqueda eficiente, ya que se mantiene equilibrado.

Árbol Rojo-Negro

Un **árbol rojo-negro** es un árbol binario de búsqueda auto-balanceado en el que cada nodo tiene un color (rojo o negro) y se deben cumplir ciertas reglas para mantener el balance del árbol.

Propiedades:

- Las reglas del árbol rojo-negro aseguran que el árbol esté equilibrado y sus operaciones (búsqueda, inserción y eliminación) se realicen en tiempo logarítmico.

Pilas (Stack)

Una **pila (stack)** es una estructura de datos que sigue el principio **LIFO (Last In, First Out)**. El último elemento agregado es el primero en ser removido.

Operaciones clave:

- **Push:** Insertar un elemento en la pila.
- **pop:** Eliminar el elemento superior de la pila.

Colas (Queue)

Una **cola (queue)** es una estructura de datos que sigue el principio **FIFO (First In, First Out)**. El primer elemento agregado es el primero en ser removido.

Operaciones clave:

- **Enqueue:** Insertar un elemento en la cola.
- **Dequeue:** Eliminar el primer elemento de la cola.

Lista Enlazada

Una **lista enlazada** es una estructura de datos en la que cada elemento (nodo) contiene un valor y un puntero (referencia) al siguiente nodo.

Tipos:

- **Lista enlazada simple:** Cada nodo apunta al siguiente.
- **Lista doblemente enlazada:** Cada nodo apunta tanto al siguiente como al anterior.
- **Lista circular:** El último nodo apunta al primer nodo.

Recursividad

La **recursividad** es una técnica de programación donde una función se llama a sí misma para resolver un problema descomponiéndolo en subproblemas más pequeños.

Características:

- **Caso base:** Condición que detiene la recursión.
- **Llamada recursiva:** La función se llama a sí misma con una versión más simple del problema.

Árbol B

Un **árbol B** es una estructura de datos de árbol auto-balanceado que mantiene los elementos ordenados y permite búsquedas, inserciones y eliminaciones eficientes.

Propiedades:

- Es utilizado para almacenar grandes cantidades de datos en estructuras de almacenamiento como bases de datos y sistemas de archivos.

Árbol B+

Un **árbol B+** es una variante del árbol B donde los nodos hoja están enlazados, permitiendo un recorrido secuencial eficiente.

Propiedades:

- Los nodos internos no almacenan datos, solo claves para direccionar a los nodos hoja.

Árbol Heap

Un **árbol heap** es un árbol binario completo que cumple con la propiedad de heap. En un **heap máximo**, el valor de cada nodo es mayor que el valor de sus hijos, mientras que en un **heap mínimo**, el valor de cada nodo es menor que el valor de sus hijos.

Propiedades:

- Utilizado en algoritmos de ordenación como **Heap Sort** y estructuras como **colas de prioridad**.

Árboles Rojo-Negro vs Árboles B

Ambos son árboles binarios de búsqueda auto-balanceados, pero tienen diferencias clave en sus criterios de balanceo:

- **Árbol Rojo-Negro:** El balanceo se mantiene mediante reglas de color, lo que permite menos rotaciones en algunos casos.
- **Árbol B:** Es utilizado principalmente en bases de datos, donde se mantiene un árbol balanceado de múltiples nodos, permitiendo búsquedas rápidas en discos.

Nodo

Un **nodo** es un elemento básico de muchas estructuras de datos. En árboles y listas, un nodo contiene datos y punteros que vinculan a otros nodos.

Puntero

Un **puntero** es una variable que almacena la dirección de memoria de otra variable. Los punteros se utilizan para manipular directamente los datos y gestionar estructuras dinámicas como listas enlazadas y árboles.

Operaciones de Árboles Binarios

- **Inserción:** Colocar un nuevo nodo en el árbol siguiendo las reglas de la estructura (por ejemplo, en un árbol binario de búsqueda).
- **Búsqueda:** Localizar un nodo dentro del árbol.
- **Eliminación:** Eliminar un nodo del árbol mientras se mantienen las propiedades de la estructura.
- **Recorridos:** Procesar todos los nodos del árbol en un orden determinado (preorden, inorden, postorden).

Árboles de Expresión

Los **árboles de expresión** son utilizados para representar expresiones matemáticas de manera jerárquica, donde los nodos internos son operadores y las hojas son operandos.

Árbol Binario Completo

Un **árbol binario completo** es un árbol en el que todos los niveles, excepto posiblemente el último, están completamente llenos, y todos los nodos están lo más a la izquierda posible.