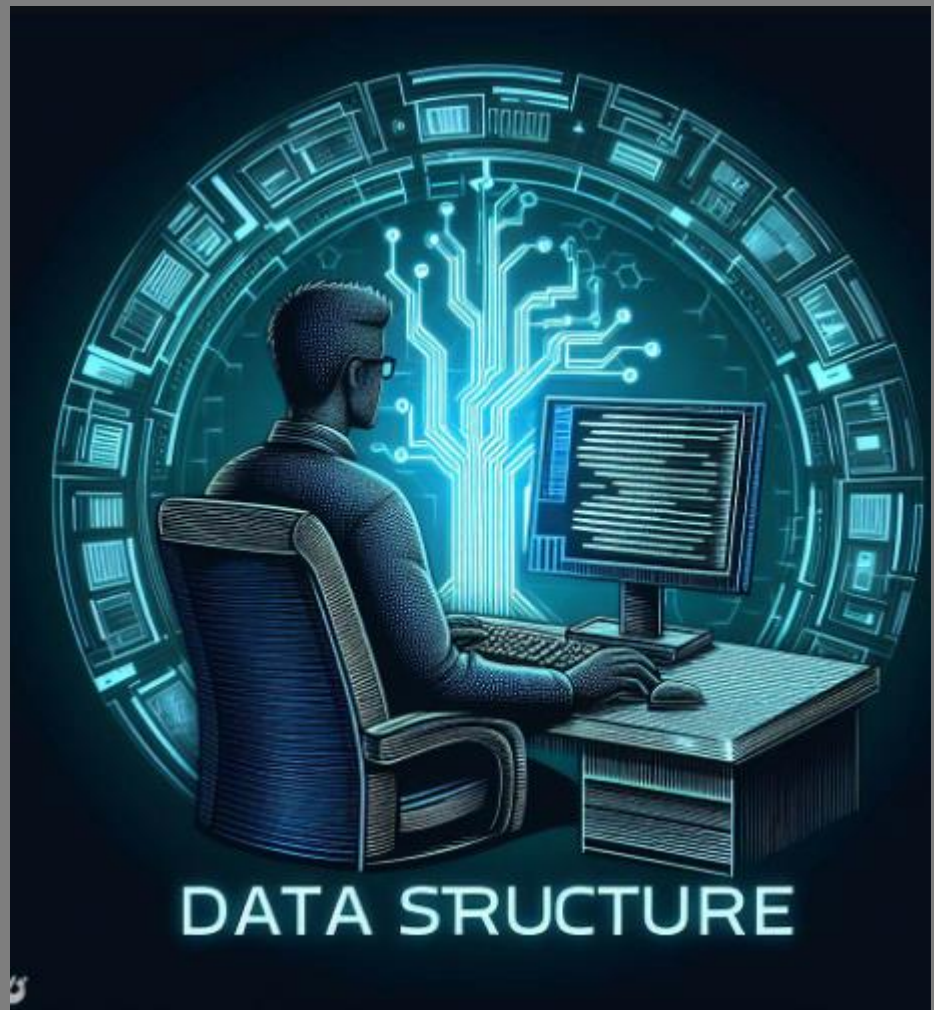


2024

UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE

MANUAL DE ESTRUCTURAS DE DATOS

INTRODUCCIÓN Y FUNDAMENTOS DE LAS ESTRUCTURAS DE DATOS



Contenido

PARCIAL 1

1.	USO DE TDA.....	5
1.1.	Introducción.....	5
1.2.	Definición de Tipos de Datos Abstractos (TDA).....	5
1.3.	Importancia de los TDA	5
1.4.	Uso de TDA	5
1.5.	Implementación de TDA	6
1.6.	Conclusiones.....	8
2.	MANEJO DE PLANTILLAS (CLASES GENÉRICAS)	8
2.1.	Introducción.....	8
2.2.	Definiciones:.....	8
2.2.1.	Plantillas (Templates):	8
2.2.2.	Clases Genéricas:	8
2.2.3.	Tipos de Datos Abstractos (TDA):	8
2.3.	Importancia de las plantillas:	8
2.4.	Ejemplo:	9
	Par.h 9	
	main.cpp	9
2.5.	Conclusiones.....	10
3.	SOBRECARGA DE FUNCIONES Y OPERADORES EN C++	10
3.1.	¿Qué son?.....	10
3.2.	¿Cómo se definen?	10
3.3.	¿Para qué sirven?	11
3.4.	Importancia o Utilidad dentro de la Programación.....	11
3.5.	Operadores que se pueden sobrecargar	11
3.6.	Operadores que no se pueden sobrecargar:	12
3.7.	Practica	13
3.7.1.	Diseño	13
	Sobrecarga de Funciones	13
	Sobrecarga de operadores.....	13
3.8.	Qué hacer y Qué no hacer durante la codificación	13
3.8.1.	Hacer:.....	13

3.8.2. No Hacer:	13
3.9. Implementación en código.....	13
Clase main.cpp.....	14
ClasesFrac.h	14
ClasesFrac.cpp	14
3.10. Recomendación y conclusiones	15
4. GESTIÓN DE MEMORIA ESTÁTICA CON TDA.....	15
4.1. Definición.....	15
4.2. Ejemplo TDA con Memoria Estática:	16
5. GESTIÓN DE MEMORIA DINÁMICA CON TDA.....	18
5.1. Se dividen en dos grandes grupos	18
5.2. Implementación de TDA	20
5.3. Conclusiones	20
6. RECURSIVIDAD	20
6.1. Introducción	20
6.2. Definición de Recursividad	20
6.3. Importancia de la Recursividad	21
6.4. Uso de la Recursividad.....	21
6.5. Implementación de la Recursividad.....	21
7. PRINCIPIOS DE LOS ALGORITMOS RECURSIVOS	24
7.1. Introducción	24
7.2. Principios de los Algoritmos Recursivos	24
7.3. Ejemplos Prácticos	25
7.4. Conclusiones	25
8. TIPOS DE RECURSIVIDAD.....	26
8.4. Conclusiones:	28
9. PROGRAMAS RECURSIVOS.....	29
9.1. Introducción	29
9.2. Ejemplos de Programas Recursivos.....	29
9.2.1. Cálculo del Factorial	29
9.2.2. Torres de Hanoi.....	29
9.2.3. Cálculo de la Secuencia de Fibonacci.....	30
9.3. Funcionamiento de los Programas Recursivos.....	30
9.4. Ventajas de la Recursividad sobre la Iteración	30
10. OPERACIONES BÁSICAS CON LISTAS.....	31

10.1. Operaciones básicas con listas Simples.....	31
Proceso:	34
10.1.4.Insertar un nuevo nodo en cualquier posición	35
10.1.5.Buscar Nodo.....	38
10.2. Operaciones con Listas Doblemente Enlazadas	38
10.2.1.Insertar Nodo en una lista vacía	38
10.2.2.Insertar Nodo en una lista por la Cabeza	39
10.2.3.Insertar Nodo en una lista por la Cola.	39
Insertar un nuevo nodo en cualquier posición	40
10.2.4.Buscar Nodo.....	41
10.2.5.Eliminar Nodo	41
10.3. Listas circulares Simples	42
10.3.1.Insertar un elemento en una lista circular vacía.....	42
10.3.3.Buscar un nodo en una lista circular.....	44
10.3.4.Eliminar Nodo	45
10.4. Listas Dobles circulares.....	46
10.4.1.Insertar Nuevo nodo a la lista	46
10.4.2.Eliminar Nodo	46
10.4.3.Buscar Nodo.....	47
11. TIPOS DE LISTAS	47
12. PILAS	64
12.1. Representación en memoria estática y dinámica	64
12.2. Representación en memoria estática:	65
12.2.1. Ejemplo:.....	65
12.3. Representación en memoria dinámica:	67
12.3.1. Ejemplo:.....	67
12.4. Conclusiones.....	69
13. OPERACIONES BÁSICAS CON PILAS.....	69
13.1. Introducción de Operaciones básicas con pilas.....	69
13.2. Definición.....	69
13.3. Importancia de las Operaciones básicas con pilas	70
13.4. Uso de las Operaciones básicas con pilas	70
13.5. Implementación de Operaciones básicas con pilas	70
13.6. Recursividad con ayuda de pilas.	72
13.7. Conclusiones.....	73

14.	NOTACIÓN INFIJA, PREFIJA Y POSTFIJA.....	73
14.1.	Notación infija	73
14.2.	Notación prefija.....	73
14.2.1.	Ejemplo:	73
14.3.	Notación postfija	74
14.3.1.	Ejemplo:	74
15.	REPRESENTACIÓN EN MEMORIA ESTÁTICA Y DINÁMICA	75
15.1.	Introducción	75
15.2.	Representación en Memoria Estática de Colas	75
15.2.1.	Implementación con Arreglos.....	75
15.2.2.	Ventajas	75
15.2.3.	Desventajas	76
15.3.	Representación en Memoria Dinámica de Colas.....	76
15.3.1.	Implementación con Listas Enlazadas	76
15.3.2.	Ventajas	76
15.3.3.	Desventajas	76
15.4.	Ejemplos Prácticos	76
15.4.1.	Representación Estática con Arreglos	76
15.4.2.	Representación Dinámica con Listas Enlazadas	78
15.5.	Conclusiones	79
16.	OPERACIONES CON COLAS	79
16.1.	Introducción	79
16.2.	Operaciones Básicas.....	79
16.3.	Implementación en Pseudocódigo.....	80
16.4.	Teoría y Ejemplos.....	80
16.5.	Consejos Prácticos	81
16.6.	Conclusiones	81
17.	TIPOS DE COLAS.....	82
17.1.	Introducción	82
17.2.	Colas de Prioridades	82
17.3.	Tipos de Colas de Prioridad:	83
17.4.	Colas circulares	83
17.5.	Colas dobles.....	85
17.6.	Ejemplo de cola circular.....	86
17.7.	Conclusiones	89

PARCIAL 2

1.	ALGORITMO DE BUSQUEDA INTERNA	91
2.	INTERCAMBIO.....	91
2.1.	Análisis.....	92
2.2.	Conclusiones	92
3.	BURBUJA (BUBBLE SORT).	93
3.1.	Definición.	93
3.2.	Propiedades.....	93
3.3.	Conclusiones.	98
4.	ORDENACIÓN POR DISTRIBUCIÓN.....	98
4.1.	Introducción	98
4.2.	Definición:	98
4.3.	Características:	98
4.4.	Ejemplo:.....	98
4.5.	Conclusiones	100
5.	SHELL SORT	100
5.1.	Algoritmo:.....	101
5.2.	Código:.....	101
5.3.	Conclusiones	103
6.	QUICK SORT	103
6.1.	Algoritmo:.....	103
6.2.	Código:.....	103
6.3.	Conclusiones	107
7.	ALGORITMOS DE ORDENACIÓN EXTERNA.....	107
8.	MEZCLA DIRECTA	107
8.1.	Introducción	107
8.2.	Algoritmo de Mezcla Directa	108
8.3.	Implementación del Algoritmo de Mezcla Directa	108
8.4.	Conclusiones	110
9.	MEZCLA NATURAL.	110
9.1.	Definición.	110
10.	ALGORITMO DE BÚSQUEDA - SECUENCIAL.....	111
10.1.	Comportamiento o funcionamiento de una búsqueda secuencial.....	111
10.2.	Ejemplo: Código del algoritmo de Búsqueda - Secuencial	112

11.	ALGORITMOS DE BÚSQUEDA BINARIA	113
11.1.	Introducción	113
11.2.	Definición del algoritmo.....	113
11.3.	Particularidades de los algoritmos de búsqueda binaria	114
11.4.	Uso de los algoritmos de búsqueda binaria	114
11.5.	Algoritmo y codificación de la búsqueda binaria	115
11.6.	Ejemplo:.....	116
11.7.	Conclusiones	118
12.	ALGORITMO DE BÚSQUEDA- HASH.....	118
12.1.	Introducción	118
12.2.	¿Qué es el Hashing?	118
12.3.	Componentes del Hashing.....	119
12.4.	Operaciones de Hashing.....	119
12.5.	Aplicaciones del Hashing en Algoritmos de Búsqueda	119
12.6.	Conclusiones	120
13.	CONCEPTO DE ARBOLES	120
13.1.	Introducción	120
13.2.	Características:	121
13.2.1.	Jerarquía y estructura recursiva:.....	121
13.3.	Ejemplo:	122
13.4.	Conclusiones	124
14.	CLASIFICACIÓN DE ÁRBOLES	124
14.1.	Introducción	124
14.2.	Árbol Binario	125
14.3.	Árbol AVL.....	125
14.4.	Árboles B y B+	125
14.5.	Árbol Rojo-Negro	125
14.6.	Árbol Radix	125
14.7.	Trie.....	125
14.8.	Conclusiones	126
15.	OPERACIONES BÁSICAS CON ÁRBOLES.....	126
15.1.	Introducción.	126
15.2.	Creación, Inserción, Eliminación, Recorridos sistemáticos, Balanceo	126
16.	GRAFOS	133
16.1.	Representación con grafos	134

16.2.	Objetos y punteros	135
16.3.	Matriz de incidencia	135
17.	OPERACIONES BÁSICAS DE GRAFOS.....	135
17.1.	Introducción	135
17.2.	Definición de las operaciones	135
17.3.	Codificación de las operaciones básicas	137
17.4.	Conclusiones	141
18.	ALGORITMO DE DIJKSTRA.....	141
18.1.	Introducción	141
18.2.	Algoritmo de Dijkstra	142
18.3.	Funcionamiento del Algoritmo.....	142
18.4.	Conclusiones	142
19.	GRAFOS BIPARTIDOS.....	142
19.1.	Introducción	142
19.2.	Definición:	143
19.3.	Características:	143
19.4.	Ejemplo:	144
19.5.	Conclusiones	145
20.	REFERENCIAS	145

PARCIAL 3

Búsqueda Exhaustiva.....	148
1.Definición de la Búsqueda Exhaustiva.....	148
2. Consideraciones:	148
3. Conclusiones	150
4. Recomendaciones	150
Algoritmos Voraces	151
1. Introducción	151
2. Funcionamiento	151
3. Estrategias comunes	151
4. Análisis de Algoritmos Voraces.....	152
5. Aplicaciones de Algoritmos Voraces.....	152
6. Ejemplos de Algoritmos Voraces	152
1. Definición de Divide y Vencerás.	152
2. Técnica de Divide y Vencerás.....	153
3. Ejemplo:	154

4.	Conclusiones.	154
5.	Recomendaciones.	155
	Programación Dinámica	155
	Que es la programación dinámica	155
	Características	155
	Ventajas de la programación dinámica:	156
	Desventajas de la programación dinámica:	156
	Problema de la mochila	157
	Código 157	
	Backtracking.....	159
1.	Introducción	159
2.	Funcionamiento	159
3.	Ventajas	160
4.	Desventajas	161
5.	Análisis.....	161
6.	Aplicaciones.....	162
	Eficiencia de los Algoritmos.....	167
1.	Introducción	167
2.	Definición de Eficiencia de Algoritmos	168
3.	Clasificación.....	168
4.	Enfoque de Análisis de Algoritmos	169
5.	Anexo.....	170
6.	Guía de ejercicios prácticos.....	171
	Tiempo de ejecución	172
1.	Concepto.....	172
2.	Notación.....	172
3.	Ejemplo	172
4.	Aporte matemático.....	173
5.	Ejemplo en código	174
6.	Recomendaciones	175
7.	Conclusiones	175
	Notación Asintótica.....	176
1.	Notación θ grande (Big θ):	177
2.	Notación O grande (Big O):.....	178
3.	Notación Ω grande (Big Ω).....	180

Aritmética de notación O.	180
Reglas de Cálculo de Complejidad:	181
Ejemplo.....	182
Complejidad	183
1. Introducción	183
2. Definición de Complejidad Algorítmica	183
3. Reglas elementales para el análisis de algoritmos	185
4. Ejemplos de complejidad	186
5. Ejemplo practico	188
6. Anexo.....	191
7. Guía de ejercicios prácticos	195
Análisis de Recurrencia.....	196
1. Introducción	196
2. Definición	196
3. Proceso.....	197
4. Ejemplo de análisis	197
5. Ventajas	198
6. Desventaja	199
7. Análisis.....	200
8. Ejemplo – Resolución de una Recurrencia por el Método de Expansión.....	201
9. Conclusiones	202
10. Recomendaciones:	203
Bibliografía:	204

1. USO DE TDA

1.1. Introducción

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

1.2. Definición de Tipos de Datos Abstractos (TDA)

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



Imagen 1 Ciclo de vida del TDA

1.3. Importancia de los TDA

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

1.4. Uso de TDA

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

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

Ejemplos de TDA comunes: Algunos ejemplos comunes de TDA incluyen:

- Listas enlazadas
- Árboles binarios
- Colas y pilas

- Conjuntos y diccionarios
- Grafos

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

1.5. Implementación de TDA

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

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

Siguiendo con el ejemplo de las fechas, podemos indicar que las operaciones válidas sobre una fecha son, entre otras:

Crear (dia, mes, año: natural): fecha

Incrementar (fechaInicio: fecha; numDias: entero): fecha

Distancia (fechaInicio, fin: fecha): entero

ObtenerMes (f: fecha): natural

Fecha.h

```
#ifndef FECHA_H
#define FECHA_H

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

Fecha.cpp

```

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

```

```

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

```

1.6. Conclusiones

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

2. MANEJO DE PLANTILLAS (CLASES GENÉRICAS)

2.1. Introducción

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

2.2. Definiciones:

2.2.1. Plantillas (Templates):

Las plantillas son un mecanismo en C++ que permite definir funciones y clases que funcionan con tipos de datos genéricos.

Permiten escribir código que se puede reutilizar con diferentes tipos de datos sin tener que escribir múltiples versiones de la misma función o clase.

2.2.2. Clases Genéricas:

Las clases genéricas, o plantillas de clases, son clases que pueden trabajar con tipos de datos genéricos.

Se definen utilizando la palabra clave `template`, seguida de la lista de parámetros de tipo entre `<>`.

2.2.3. Tipos de Datos Abstractos (TDA):

Los TDAs son una abstracción en programación que encapsula datos y operaciones en una estructura cohesiva.

Permiten ocultar los detalles de implementación y proporcionan una interfaz clara para interactuar con los datos.

2.3. Importancia de las plantillas:

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

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

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

2.4. Ejemplo:

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

Par.h

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

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

    T1 obtenerPrimero() const {
        return primero;
    }

    T2 obtenerSegundo() const {
        return segundo;
    }

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

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

#endif
```

main.cpp

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

```

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

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

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

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

    return 0;
}

```

2.5. Conclusiones

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

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

3. SOBRECARGA DE FUNCIONES Y OPERADORES EN C++

3.1. ¿Qué son?

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

3.2. ¿Cómo se definen?

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

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

3.3. ¿Para qué sirven?

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

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

3.4. Importancia o Utilidad dentro de la Programación

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

3.5. Operadores que se pueden sobrecargar

3.5.1. Operadores Aritméticos:

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

3.5.2. Operadores de Comparación:

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

3.5.3. Operadores Lógicos:

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

3.5.4. Operadores de Incremento y Decremento:

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

3.5.5. Operadores de Miembro:

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

3.5.6. Operador de Asignación:

- = (asignación)

3.6. Operadores que no se pueden sobrecargar:**3.6.1. Operador de Punto y Flecha:**

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

3.6.2. Operador de Resolución de Ámbito:

- :: no puede ser sobrecargado.

3.6.3. Operadores de Ternario:

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

3.6.4. Operadores de Miembro Puntero a Miembro:

- . * y → * no pueden ser sobrecargados.

3.6.5. Operadores de Flujo de Entrada/Salida:

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

3.6.6. Operadores Nuevos y Eliminar:

- 'new' y 'delete' no pueden ser sobrecargados.

Nota

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

3.7. Practica

3.7.1. Diseño

Sobrecarga de Funciones

```

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

```

Imagen 2 Sobrecarga de funciones

Sobrecarga de operadores

```

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

```

Imagen 3 Sobrecarga de Operadores

3.8. Qué hacer y Qué no hacer durante la codificación

3.8.1. Hacer:

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

3.8.2. No Hacer:

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

3.9. Implementación en código

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

Clase main.cpp

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

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

ClasesFrac.h

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

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

ClasesFrac.cpp

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

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

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

```

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

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

```

3.10. Recomendación y conclusiones

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

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

4. GESTIÓN DE MEMORIA ESTÁTICA CON TDA

4.1. Definición.

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

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

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

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

Un vector tiene dos partes: Componente e índices.

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

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

4.2. Ejemplo TDA con Memoria Estática:

main.cpp

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

Vector.h

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

```

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

```

Validacion.h

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

5. GESTIÓN DE MEMORIA DINÁMICA CON TDA

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

5.1. Se dividen en dos grandes grupos

5.1.1. Lineales: Pilas, Colas, Listas enlazadas.

5.1.2. No Lineales: Árboles y Grafos.

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

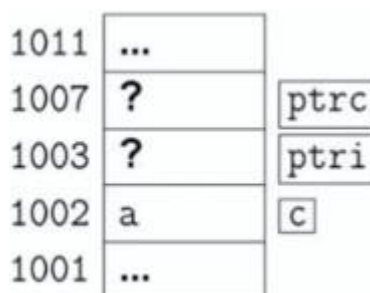


Imagen 4 Punteros

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

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

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

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

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

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

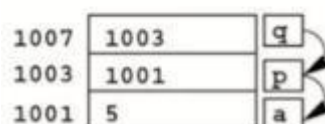
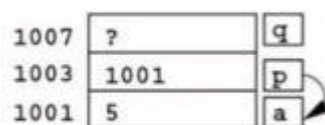
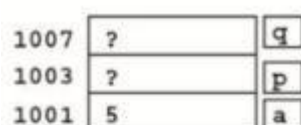


Imagen 5 Asignación de puntero a puntero

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

*p (a través del puntero p)

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

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

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

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

5.2. Implementación de TDA

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

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

Siguiendo con el ejemplo de las fechas, podemos indicar que las operaciones válidas sobre una fecha son, entre otras:

Crear (dia, mes, año: natural): fecha

Incrementar (fechaInicio: fecha; numDias: entero): fecha

Distancia (fechaInicio, fin: fecha): entero

ObtenerMes (f: fecha): natural

5.3. Conclusiones

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

6. RECURSIVIDAD

6.1. Introducción

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

6.2. Definición de Recursividad

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

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

6.3. Importancia de la Recursividad

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

6.4. Uso de la Recursividad

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

Para usar este proceso, se debe,

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



Imagen 6 Ejemplo de recursividad.

6.5. Implementación de la Recursividad

Main.cpp

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

```

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

            numero2 = 0;

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

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

```

```

        break;
    }

}while(true);

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

```

7. PRINCIPIOS DE LOS ALGORITMOS RECURSIVOS

7.1. Introducción

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

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

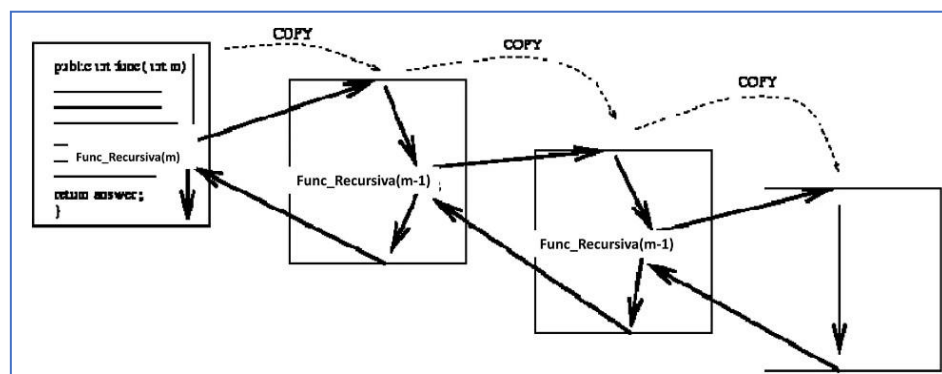


Ilustración 1 Recursividad de una función

7.2. Principios de los Algoritmos Recursivos

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

7.2.1. Caso Base

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

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

7.2.2. Paso Recursivo

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

7.2.3. Acumulación de Resultados

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

7.3. Ejemplos Prácticos

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

scssCopy code

```
factorial(n): si n == 0: retornar 1 sino: retornar n * factorial(n - 1)
```

En este ejemplo:

El caso base es cuando $n=0$, donde se retorna 1.

El paso recursivo reduce el problema al multiplicar n por el factorial de $n-1$.

No se necesita acumulación de resultados en este caso, ya que el valor del factorial se calcula directamente a partir de las llamadas recursivas.

7.4. Conclusiones

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

8. TIPOS DE RECURSIVIDAD

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

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

Ejemplo:

```
#include <iostream>

using namespace std;

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

int main() {
    Potencia potencia;

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

    return 0;
}
```

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

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


```

#include <iostream>

using namespace std;

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

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

```

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

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

```

#include <iostream>

using namespace std;

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

```

```

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

};

int main() {
    Factorial factorial;

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

    return 0;
}

```

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

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

8.4. Conclusiones:

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

9. PROGRAMAS RECURSIVOS

9.1. Introducción

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

9.2. Ejemplos de Programas Recursivos

9.2.1. Cálculo del Factorial

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

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

9.2.2. Torres de Hanoi

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

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

9.2.3. Cálculo de la Secuencia de Fibonacci

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

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

9.3. Funcionamiento de los Programas Recursivos

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

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

9.4. Ventajas de la Recursividad sobre la Iteración

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

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

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

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

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

10. OPERACIONES BÁSICAS CON LISTAS

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

Los tipos de operaciones básicas son:

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

10.1. Operaciones básicas con listas Simples

10.1.1. Insertar Nodo en una lista vacía

- Suponer que tenemos un nodo a ser insertado.



Imagen 7 *Nodo a insertar*

Un puntero denominado Lista que apunte a Null.

lista  Null

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

Proceso:

- Considerar condiciones iniciales.



Imagen 9 *Lista apunta a NULL*

- Nodo---> siguiente debe apuntar a NULL

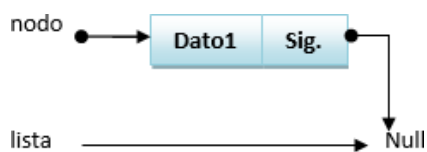


Imagen 10 *Nodo siguiente apunta a NULL*

- Lista apunta a nodo



Imagen 11 *Lista apunta al Nodo*

- Lista=nodo



Imagen 12 *Nodo va a hacer igual a lista*

10.1.2. Insertar en la cabeza de la lista

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

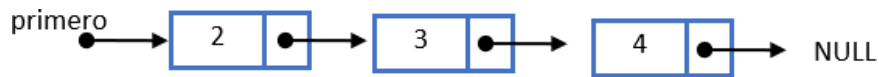


Imagen 13 Lista simple

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

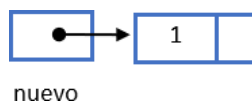


Imagen 14 Asignación de la dirección del nodo creado a nuevo

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

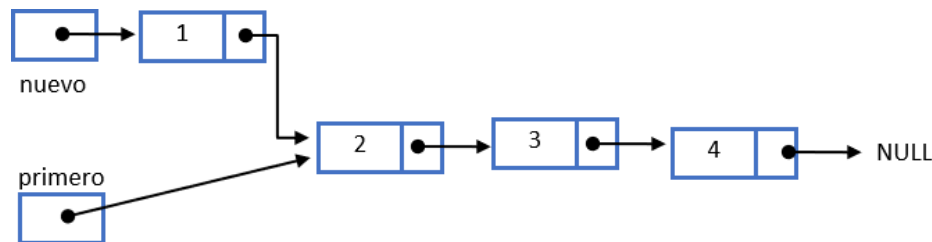


Imagen 15 Nuevo Nodo apunte a la cabeza

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

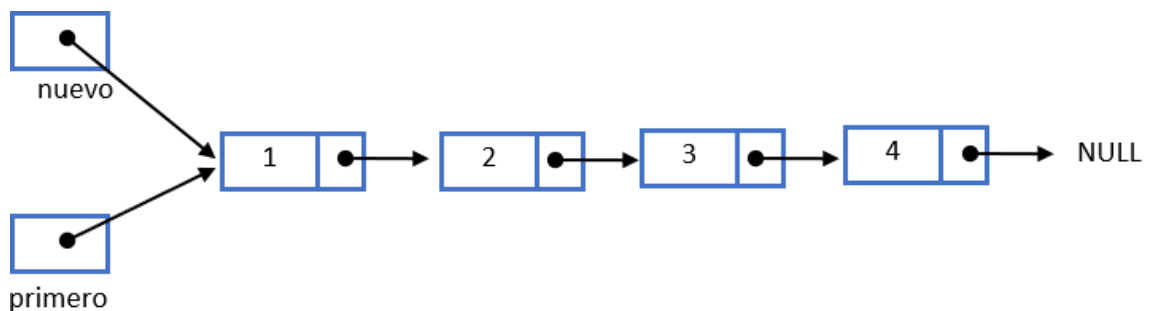


Imagen 16 Inserción del nuevo nodo

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

10.1.3. Insertar en la cola de la lista

Condiciones:

- Disponer de una lista con cierto número de elementos.

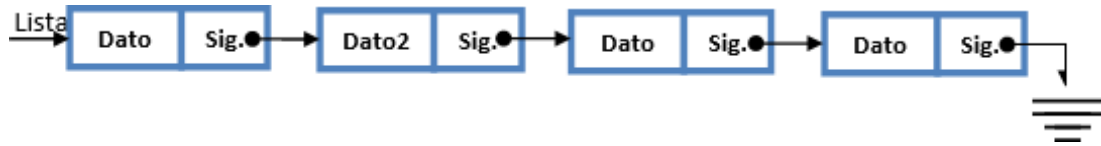


Imagen 17 *Lista Simple con datos*

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



Imagen 18 *Nuevo nodo*

- Disponer de un apuntador denominado: "ultimo".

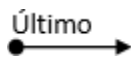


Imagen 19 *Auxiliar denominado Ultimo*

Proceso:

- Integrar en un solo diagrama las condiciones iniciales.

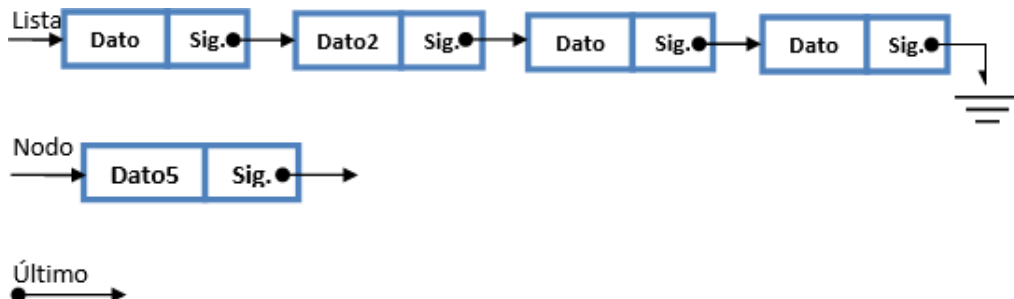


Imagen 20 *Diagrama de condiciones iniciales*

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



Imagen 21 *Ultimo apunta al último nodo*

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

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



Imagen 22 *nodo->siguiente = ultimo siguiente*

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

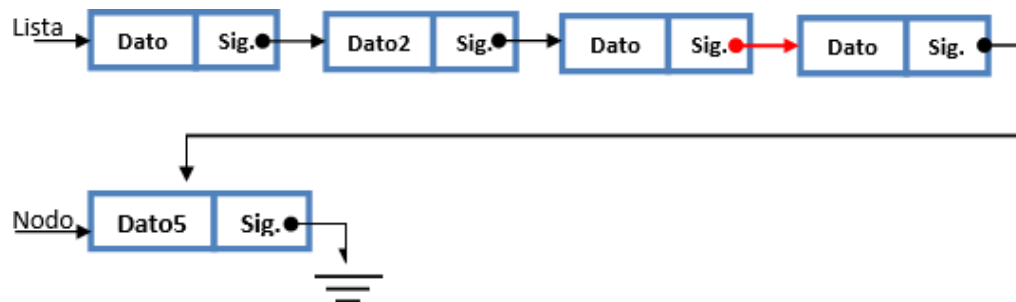


Imagen 23 *ultimo->siguiente y nodo->siguiente apuntan al nuevo nodo a la vez*

- `ultimo->siguiente = nodo->siguiente`

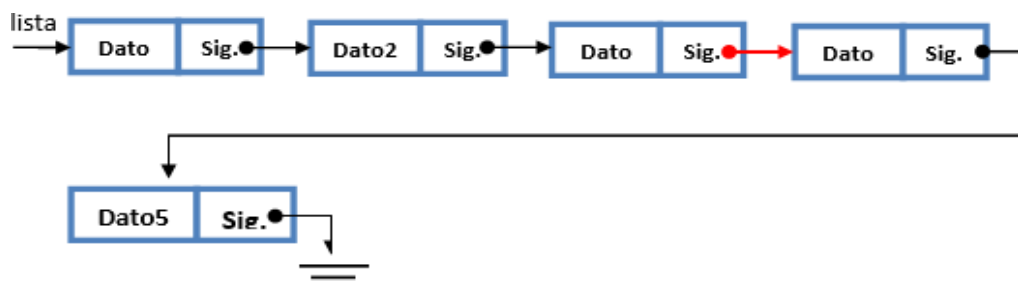


Imagen 24 *Insertión del nuevo Nodo*

10.1.4. Insertar un nuevo nodo en cualquier posición

Condiciones Iniciales:

- Disponer de una lista.



Imagen 25 *Lista Simple*

- Disponer de un nuevo nodo a ser insertado.

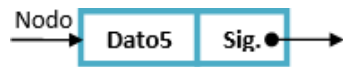


Imagen 26 Nuevo Nodo

- Disponer de un apuntador denominado anterior

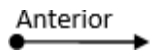


Imagen 27 Puntero anterior

Proceso:

- Integrar en un solo diagrama las condiciones iniciales.

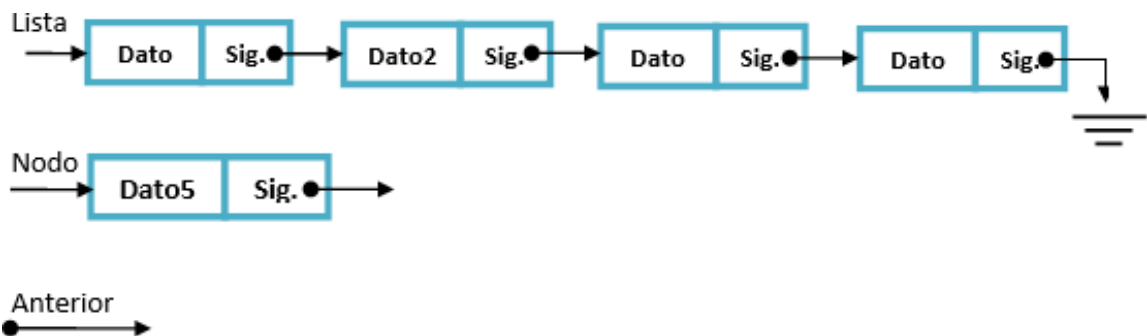


Imagen 28 Diagrama de condiciones iniciales

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

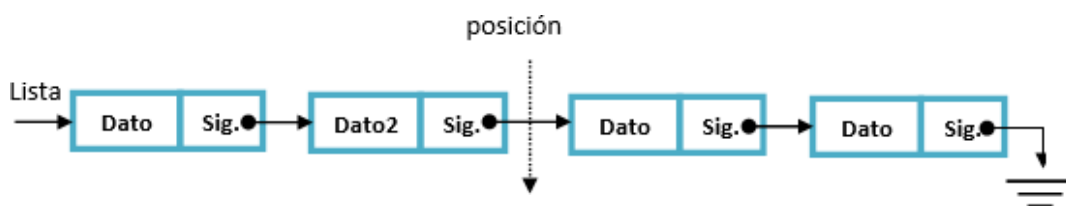


Imagen 29 Determinación de nuevo nodo a insertar

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

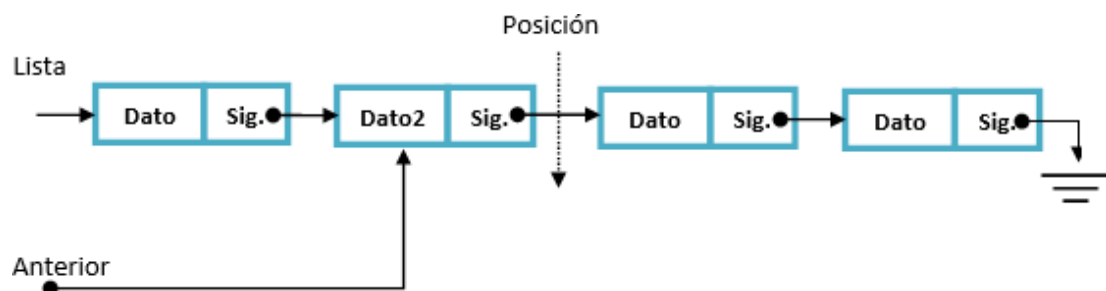


Imagen 30 Apuntador apunta al nodo anterior donde se desea insertar el nodo

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

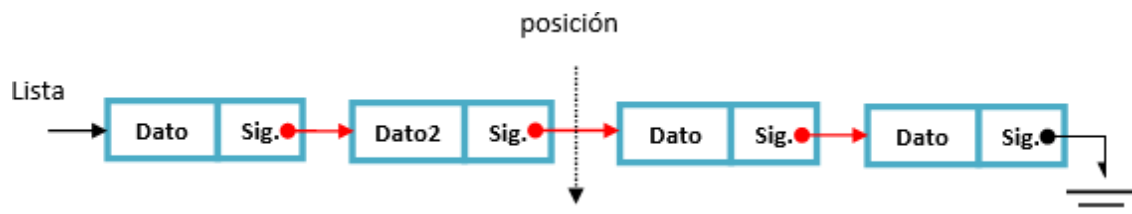


Imagen 31 nodo->siguiente = anterior->siguiente

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

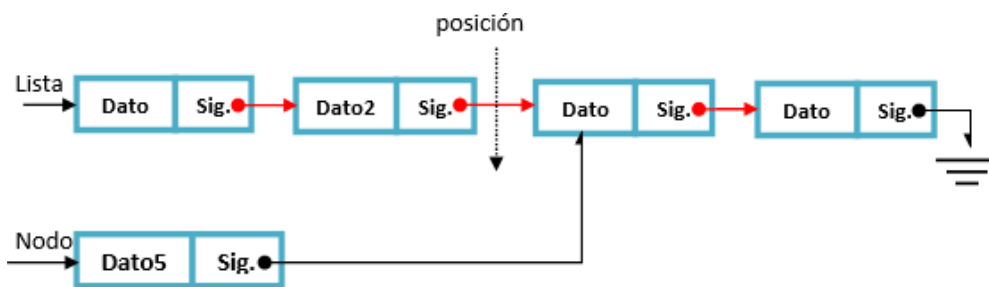


Imagen 32 Nodo Siguiente apunte a la ubicación de la inserción del nuevo nodo

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

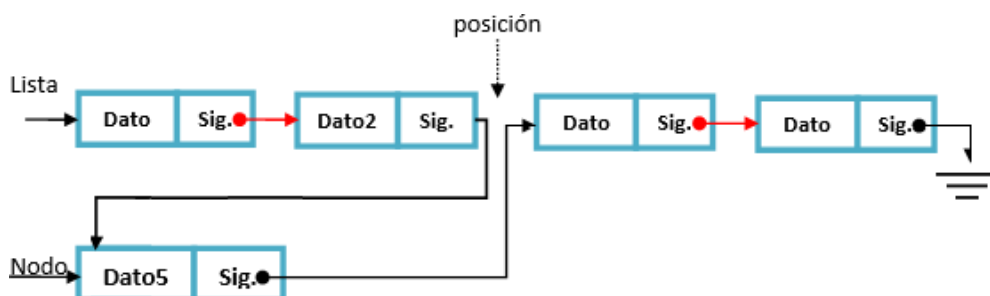


Imagen 33 Nodo Siguiente = nodo->siguiente

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

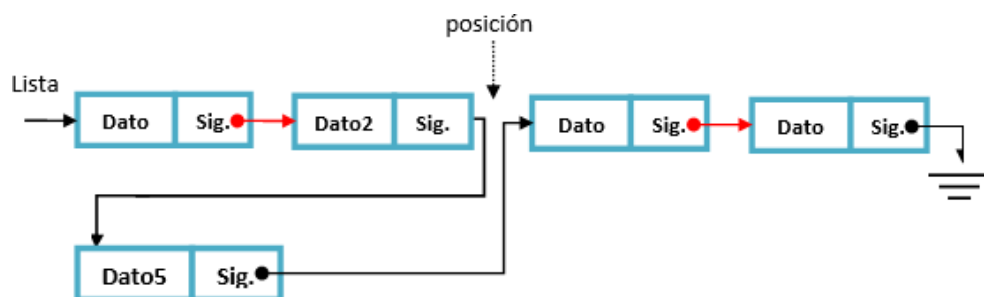


Imagen 35 Inserción del nuevo nodo

10.1.5. Buscar Nodo

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

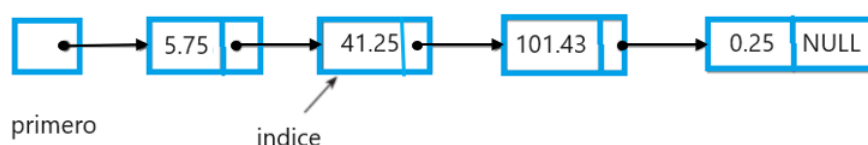


Imagen 35 Búsqueda de un nodo.

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

10.2. Operaciones con Listas Doblemente Enlazadas

10.2.1. Insertar Nodo en una lista vacía

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

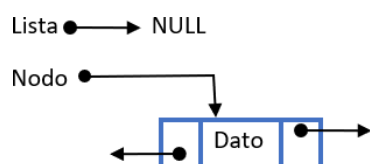


Imagen 36 Diagrama del nuevo nodo y el puntero que apunta a NULL

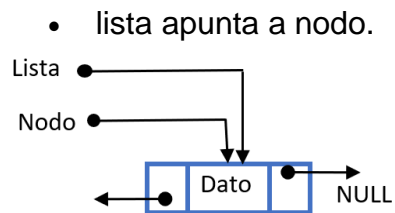


Imagen 37 Lista->siguiente y lista->anterior apunten a NULL

10.2.2. Insertar Nodo en una lista por la Cabeza

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

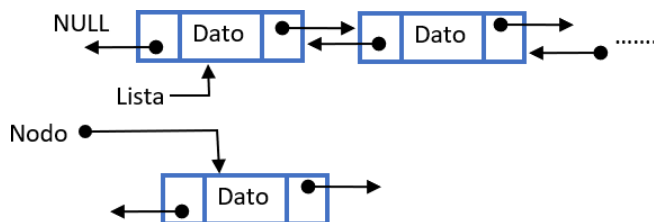


Imagen 38 Diagrama del nuevo nodo a insertar y la lista no vacía

El proceso es el siguiente:

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

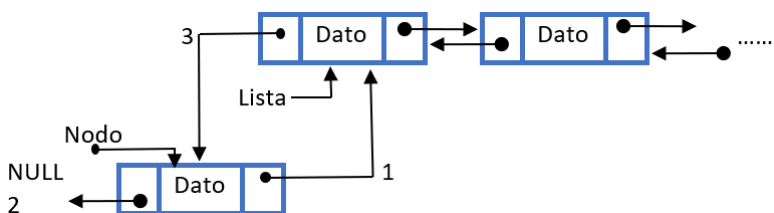


Imagen 39 Diagrama del comportamiento de nuevo nodo

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

10.2.3. Insertar Nodo en una lista por la Cola.

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

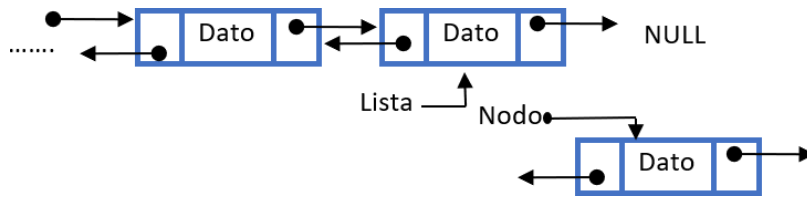


Imagen 40 Diagrama del nuevo nodo a insertar y la lista no vacía

El proceso es el siguiente:

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

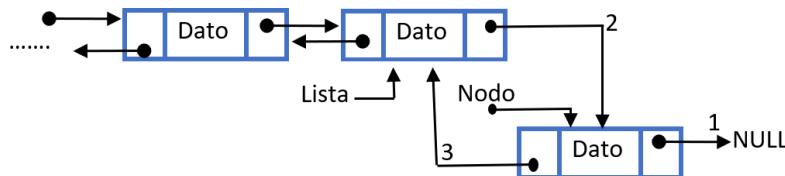


Imagen 41 Diagrama del comportamiento del nodo que se desea insertar

Insertar un nuevo nodo en cualquier posición

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

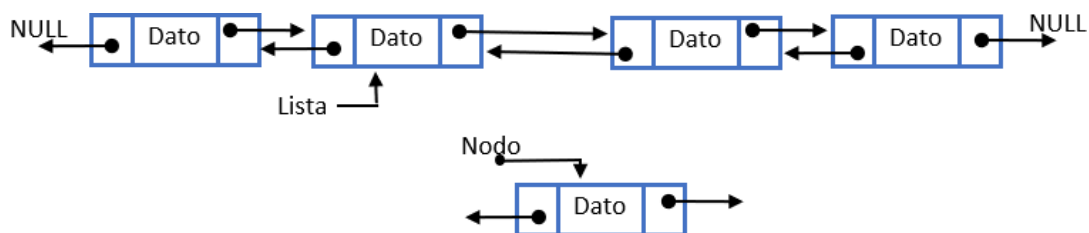


Imagen 42 Diagrama de la lista y la posición donde se desea ingresar el nuevo nodo

El proceso sigue siendo muy sencillo:

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

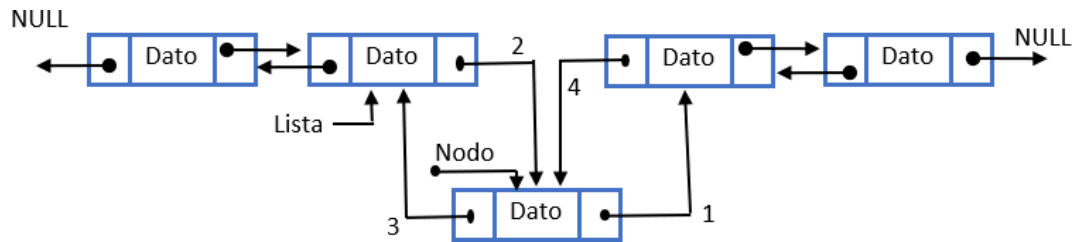


Imagen 43 Diagrama del comportamiento del nodo que se desea insertar

10.2.4. Buscar Nodo

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

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

10.2.5. Eliminar Nodo

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

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

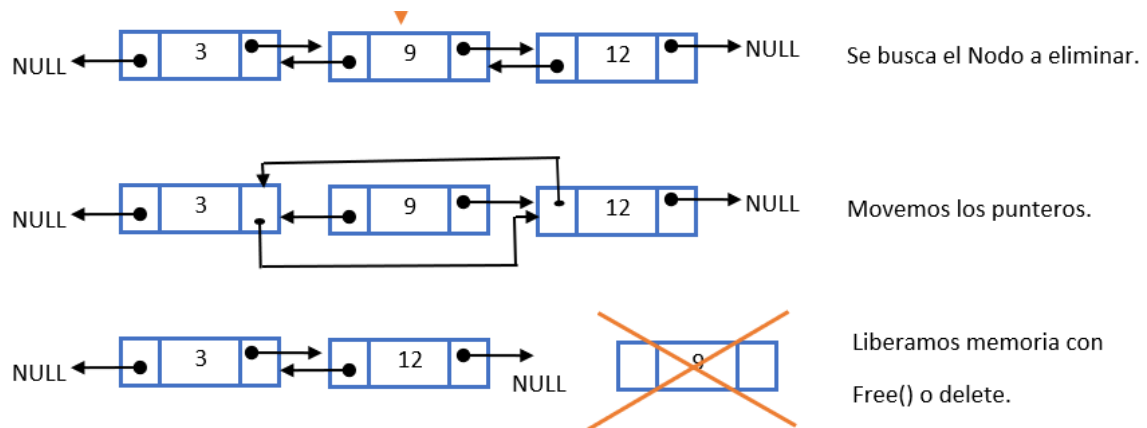


Imagen 44 Diagrama de la eliminación de un nodo de una lista

10.3. Listas circulares Simples

10.3.1. Insertar un elemento en una lista circular vacía.

Condiciones Iniciales:

- Disponer de un nodo a ser insertado.



Imagen 45 Nodo que se desea insertar

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



Imagen 46 Puntero que apunta a Null

Proceso:

- Integrar en un solo diagrama las condiciones iniciales.

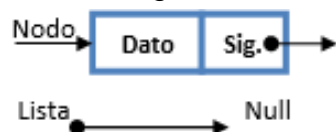


Imagen 47 Diagrama del nodo a ingresar y el puntero

- Hacer que Lista apunte al nodo.

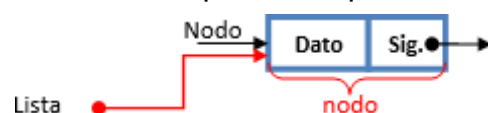


Imagen 48 Puntero apunta al nodo

- Lista=nodo



Imagen 49 El nuevo nodo se apunta a si mismo

- Lista->siguiente apunte al nodo.

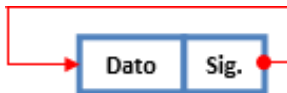


Imagen 50 Nodo insertado en la lista vacía

10.3.2. Insertar un elemento en una lista circular no vacía

Condiciones Iniciales:

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



Imagen 51 Diagrama de la lista circular

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



Imagen 52 Nodo que se desea insertar

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



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

- Integrar las condiciones iniciales.

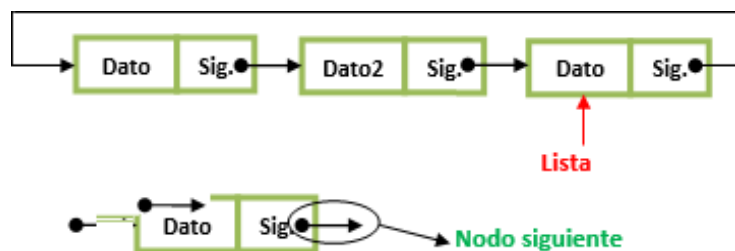


Imagen 54 Diagrama de condiciones iniciales

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

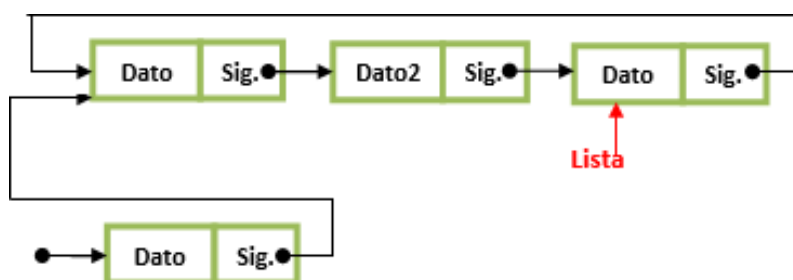


Imagen 55 Nodo nuevo apunta al nodo que siguiente de donde se va a insertar

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

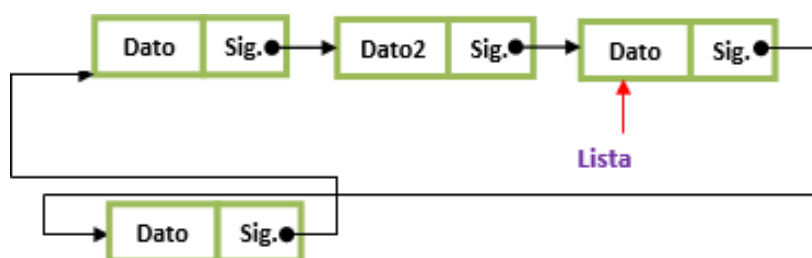


Imagen 56 La lista apunta al nodo insertado

- Lista->siguiente = nodo

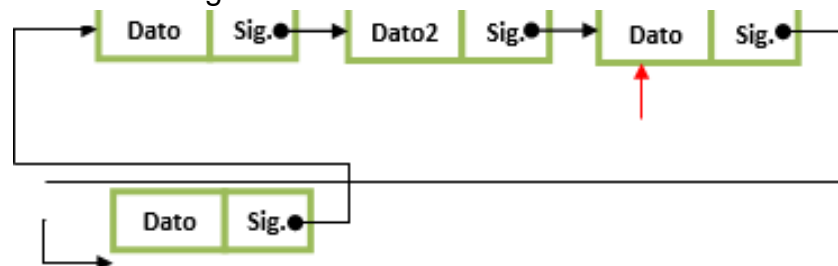


Imagen 57 Inserción del nodo

10.3.3. Buscar un nodo en una lista circular

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

10.3.4. Eliminar Nodo

- Disponer una lista circular

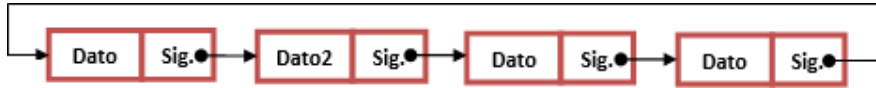


Imagen 56 Lista circular

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

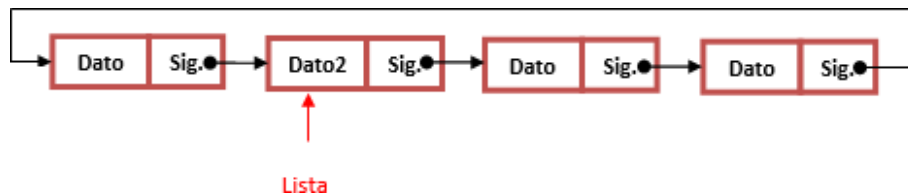


Imagen 58 El puntero auxiliar apunta a la posición anterior donde se desea hacer la inserción

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



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

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



Imagen 60 Diagrama del comportamiento lista

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



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

- Eliminar el nodo

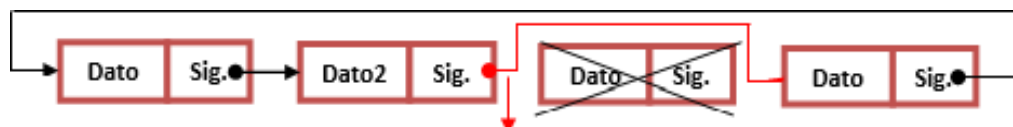


Imagen 62 Eliminación del nodo

- Presentar la nueva lista.

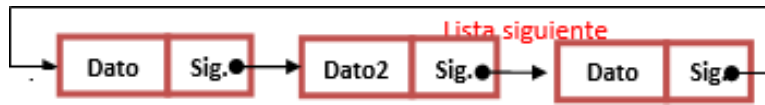


Imagen 63 Diagrama de la lista sin el nodo

10.4. Listas Dobles circulares

10.4.1. Insertar Nuevo nodo a la lista

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

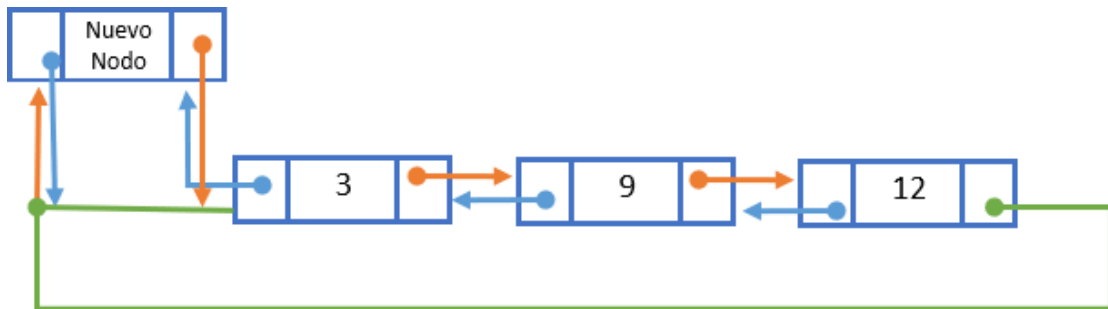


Imagen 64 Diagrama de la inserción de un nodo a una lista doble circular

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

10.4.2. Eliminar Nodo

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

10.4.3. Buscar Nodo

Pasos para buscar un elemento en una lista circular doble:

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

11. TIPOS DE LISTAS

11.1. Listas Simples

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

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

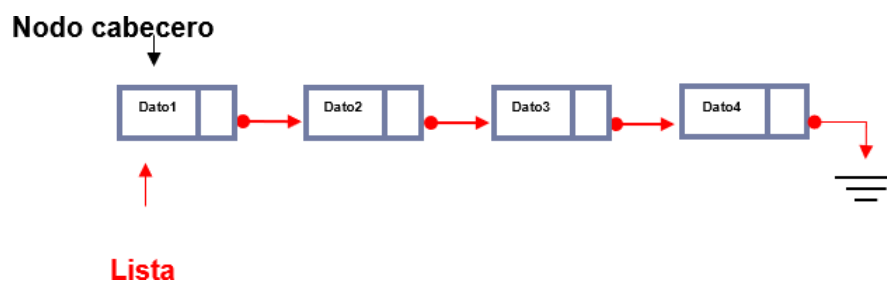


Imagen 8 Lista Simple

Ejemplo:

main.cpp

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

```

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

```

ListaSimple.h

```

#pragma once
#include <iostream>

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

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

```

```

};

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

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

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

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

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

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

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

```

```

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

        delete temp;
    }

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

```

Validacion.h

```

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

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

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

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

```



```

        i = borrar(datos, i);
    }
    else {
        printf("%c", c);
        datos[i++] = c;
    }
}
}
datos[i] = '\0';
return atoi(datos);
}

```

11.2. Listas Dobles Enlazadas.

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

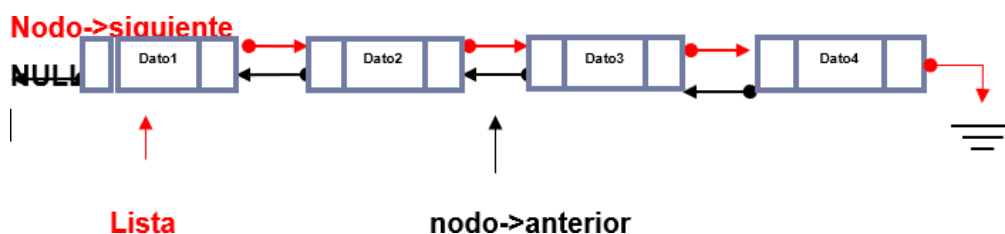


Imagen 65 Lista Doble

Ejemplo:

main.cpp

```

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

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

```

```

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

```

ListaDoble.cpp

```

#pragma once
#include <iostream>

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

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

```

```

};

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

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

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

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

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

```

```

    }

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

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

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

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

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

        delete temp;
    }

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

```

Validacion.h

```

#pragma once
#include <conio.h>

```

```

#include <iostream>

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

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

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

```

11.3. Listas Simples Circulares

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

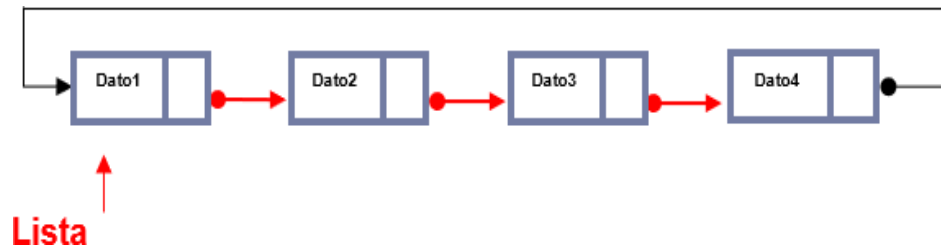


Imagen 66 Lista simple circular

Ejemplo:

Main.cpp

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

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

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

ListaCircular.h

```
#pragma once
#include <iostream>

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

```



```

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

```

Validacion.h

```

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

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

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

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

```

11.4. Listas Dobles Circulares.

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

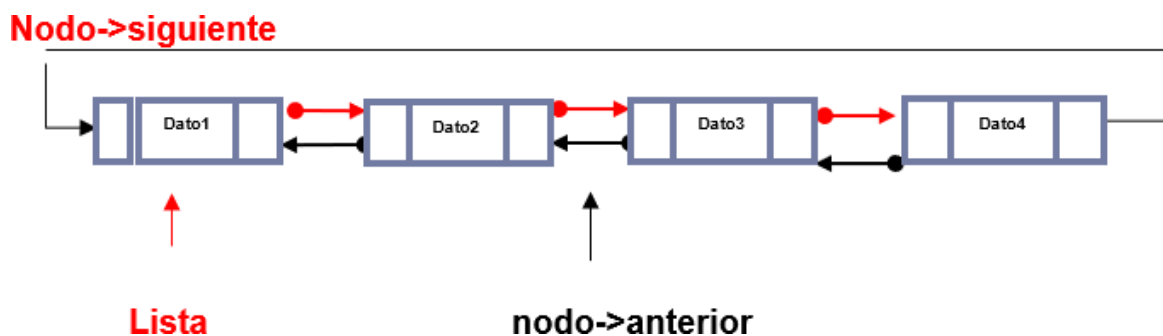


Imagen 67 Lista doble circular

Ejemplo:

Main.h

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

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

```

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

```

ListaDobleCircular.h

```

#pragma once
#include <iostream>

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

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

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

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

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

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

```

```

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

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

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

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

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

```

```

    Nodo* temp = cabeza;
    do {
        if (temp->dato == clave) {
            if (temp == cabeza) {
                cabeza = temp->siguiente;
            }
            temp->anterior->siguiente = temp->siguiente;
            temp->siguiente->anterior = temp->anterior;
            delete temp;
            std::cout << "Elemento " << clave << " eliminado." <<
std::endl;

            return;
        }
        temp = temp->siguiente;
    } while (temp != cabeza);
    std::cout << "Elemento " << clave << " no encontrado en la lista." <<
std::endl;
}

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

```

Validacion.h

```

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

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

```

```

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

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

```

12. PILAS

12.1. Representación en memoria estática y dinámica

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

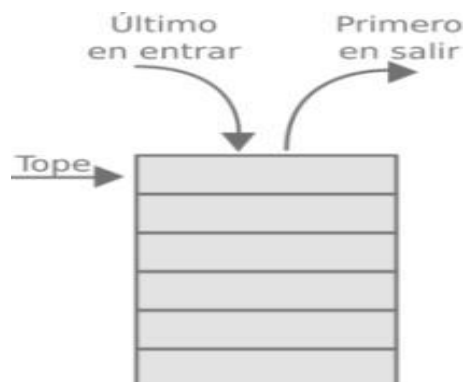


Imagen 68: Estructura de la Pila

12.2. Representación en memoria estática:

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

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

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

12.2.1. Ejemplo:

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

PilaEstatica.h:

```
#ifndef PILA_ESTATICA_H
#define PILA_ESTATICA_H

#include <iostream>

#define TAM_MAX 100 // Tamaño máximo de la pila

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

#endif
```

PilaEstatica.cpp:

```
#include "PilaEstatica.h"

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

```

}

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

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

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

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

```

main.cpp

```

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

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

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

    return 0;}

```


12.3. Representación en memoria dinámica:

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

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

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

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

12.3.1. Ejemplo:

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

PilaDinamica.h

```
#ifndef PILA_DINAMICA_H
#define PILA_DINAMICA_H

#include <iostream>

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

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

```
#endif // PILA_DINAMICA_H
```

PilaDinamica.cpp

```
#include "PilaDinamica.h"

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

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

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

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

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

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

main.cpp

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

```

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

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

    return 0;
}

```

12.4. Conclusiones

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

13. OPERACIONES BÁSICAS CON PILAS

13.1. Introducción de Operaciones básicas con pilas

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

13.2. Definición

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

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

13.3. Importancia de las Operaciones básicas con pilas

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

13.4. Uso de las Operaciones básicas con pilas

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

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

13.5. Implementación de Operaciones básicas con pilas

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

```
#include <iostream>

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

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

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

```

    NodoPila* tope;

public:
    Pila() {
        tope = nullptr;
    }

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

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

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

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

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

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

```

```

    }

    return tope->dato;
}
};

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

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

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

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

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

    return 0;
}

```

13.6. Recursividad con ayuda de pilas.

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

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

13.7. Conclusiones

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

14. NOTACIÓN INFIJA, PREFIJA Y POSTFIJA

14.1. Notación infija

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

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

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

14.2. Notación prefija

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

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

14.2.1. **Ejemplo:**

Transformando el ejemplo de la notación infija:

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

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

Tabla 1. Transformación de Notación Infija a Prefija

14.3. Notación postfija

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

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

14.3.1. Ejemplo:

Transformando el ejemplo de la notación infija:

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

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

Tabla 2. Transformación de Notación Infija a Postfija

15. REPRESENTACIÓN EN MEMORIA ESTÁTICA Y DINÁMICA

15.1. Introducción

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

15.2. Representación en Memoria Estática de Colas

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

15.2.1. Implementación con Arreglos

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

15.2.2. Ventajas

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

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

15.2.3. Desventajas

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

15.3. Representación en Memoria Dinámica de Colas

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

15.3.1. Implementación con Listas Enlazadas

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

15.3.2. Ventajas

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

15.3.3. Desventajas

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

15.4. Ejemplos Prácticos

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

15.4.1. Representación Estática con Arreglos

```
const int MAX_SIZE = 100;
```

```
class StaticQueue {
private:
```

```

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

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

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

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

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

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

```

15.4.2. Representación Dinámica con Listas Enlazadas

```

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

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

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

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

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

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

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

```

```

        final = nullptr;
    }
}

```

15.5. Conclusiones

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

16. OPERACIONES CON COLAS

16.1. Introducción

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

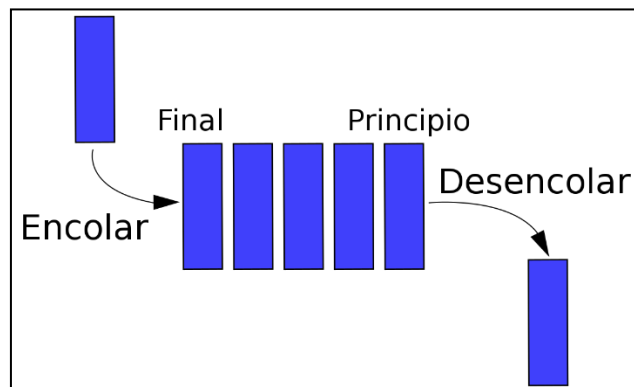


Imagen 69 Procesos de una cola

16.2. Operaciones Básicas

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

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

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

16.3. Implementación en Pseudocódigo

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

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

16.4. Teoría y Ejemplos

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

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

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

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

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

16.5. Consejos Prácticos

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

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

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

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

16.6. Conclusiones

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

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

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

17. TIPOS DE COLAS

17.1. Introducción

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

17.2. Colas de Prioridades

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

Características clave de las colas de prioridad:

- **Prioridad Asociada:**

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

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

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

- **Operaciones de Inserción y Eliminación:**

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

- **Implementación con Estructuras de Datos:**

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

Casos de Uso:

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

17.3. Tipos de Colas de Prioridad:

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

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

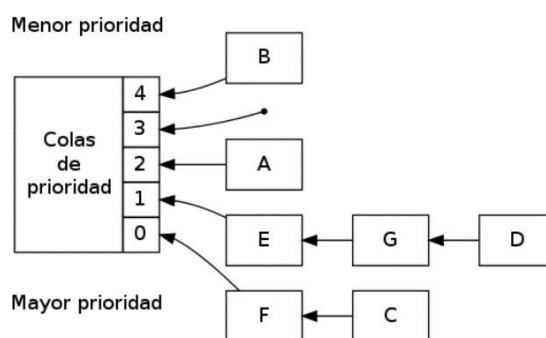


Imagen 70: Cola de prioridad.

17.4. Colas circulares

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

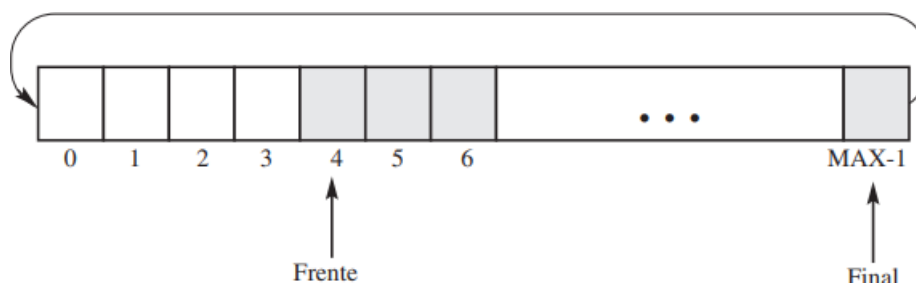


Imagen 71. Estructura de una cola circular.

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

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

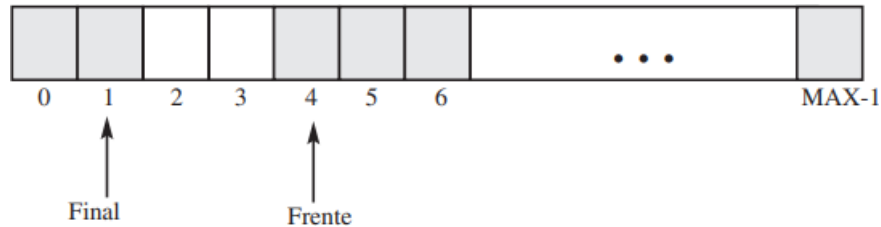


Imagen 72 Cola circular

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

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

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

- **Estructura Circular:**

En una cola circular, los elementos están organizados en una estructura circular, lo que significa que el último elemento está conectado al primer elemento, formando un bucle.

- **Frente y Final:**

A diferencia de las colas lineales tradicionales, las colas circulares no tienen un "final" o una "cima" claramente definidos. El frente y el final de la cola están conectados en la estructura circular.

- **Uso Eficiente de Espacio:**

Las colas circulares pueden ser más eficientes en términos de uso de espacio en comparación con las colas lineales, ya que no hay una posición fija al final de la cola que debe permanecer vacía para permitir la expansión.

- **Desbordamiento o Subdesbordamiento:**

Al usar una implementación basada en arreglos, es importante gestionar adecuadamente el desbordamiento (cuando la cola está llena) y el subdesbordamiento (cuando la cola está vacía). Esto se logra utilizando la aritmética modular para realizar operaciones circulares.

- **Operaciones de Enqueue (Encolar) y Dequeue (Desencolar):**

Se pueden realizar operaciones de encolar y desencolar en una cola circular. Al encolar elementos, se insertan al final de la cola, y al desencolar elementos, se eliminan desde el frente.

- **Uso Común en Sistemas Embebidos y Ciclos Continuos:**

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

- **Aplicaciones en Algoritmos de Manejo de Datos:**

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

17.5. Colas dobles

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

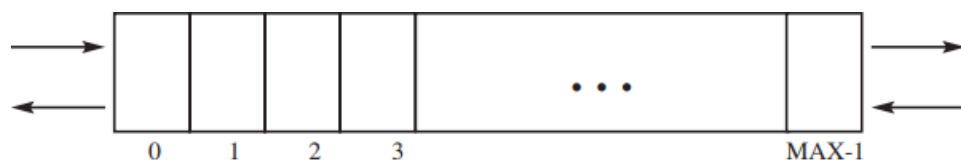


Imagen 73. Cola doble

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

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

- **Operaciones de Inserción y Eliminación en Ambos Extremos:**

Una característica fundamental es la capacidad de realizar operaciones de inserción (**push**) y eliminación (**pop**) tanto en el frente como en el final de la cola. Esto proporciona flexibilidad en la manipulación de los elementos.

- **Estructura Lineal:**

Aunque el término "cola doble" sugiere que tiene dos extremos, la estructura subyacente suele ser lineal. La diferencia clave es que se pueden acceder y manipular ambos extremos.

- **Acceso Eficiente a Ambos Extremos:**

A diferencia de otras estructuras de datos, como las listas enlazadas, las colas dobles permiten un acceso eficiente tanto al frente como al final. Esto

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

- **Implementación con Arreglos o Listas Enlazadas:**

Las colas dobles pueden implementarse utilizando arreglos o listas enlazadas. La elección de la implementación dependerá de los requisitos específicos de la aplicación y las operaciones que se realizarán con mayor frecuencia.

- **Casos de Uso Versátiles:**

Debido a su capacidad para realizar operaciones en ambos extremos, las colas dobles son versátiles y se utilizan en situaciones donde se requiere flexibilidad en la manipulación de elementos, como en algoritmos de procesamiento de datos en tiempo real o sistemas de gestión de eventos.

17.6. Ejemplo de cola circular

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

colasCirculares.h

```
#include <iostream>

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

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

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

ColasCirculares.cpp

```
#include "colasCirculares.h"

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

```

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

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

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

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

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

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

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

```

```

}

main.cpp

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

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

    CircularQueue colaCircular(tamano);

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

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

            case 2:
                colaCircular.desencolar();
                break;

            case 3:
                colaCircular.mostrarContenido();
                break;

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

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

    } while (opcion != 0);
}

```

```
    return 0;  
}
```

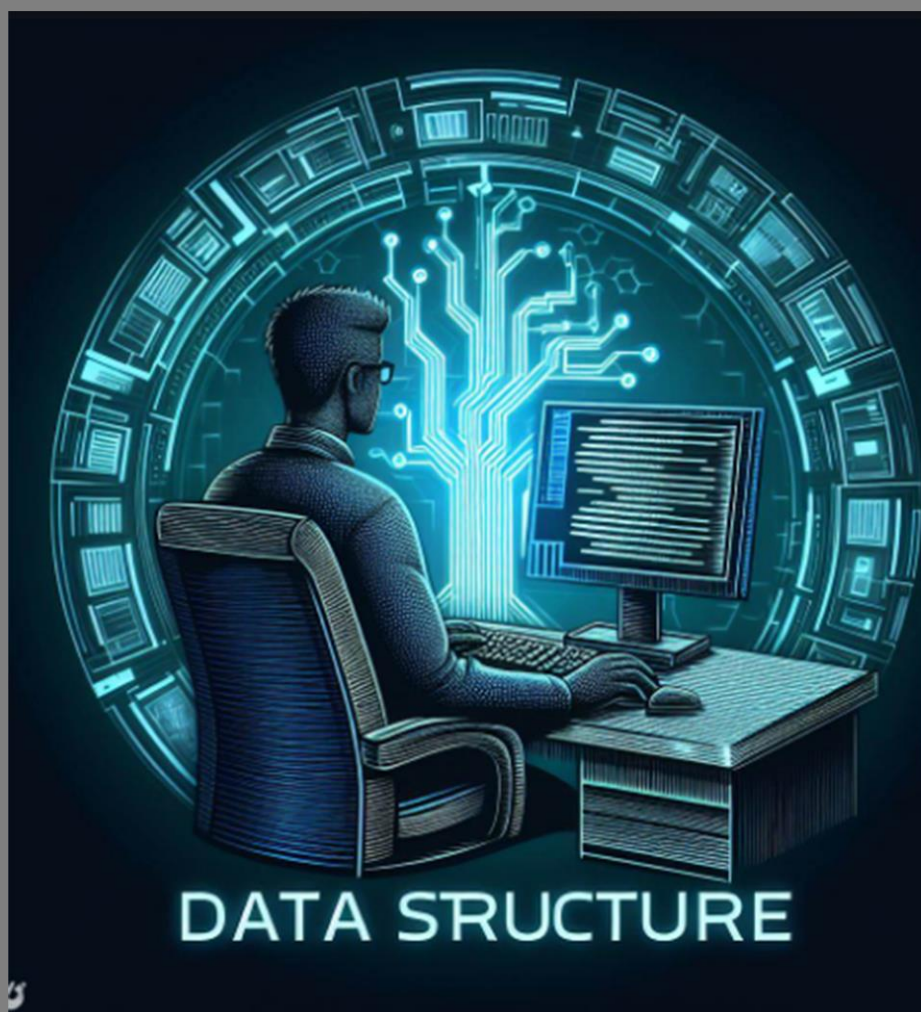
17.7. Conclusiones

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

2024

UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE

MANUAL DE ESTRUCTURAS DE DATOS TEORÍA DE ALGORITMOS



TERCER SEMESTRE
NRC: 14675



1. ALGORITMO DE BUSQUEDA INTERNA

2. INTERCAMBIO

El algoritmo de intercambio (Exchange sort en inglés) funciona intercambiando elementos adyacentes en la lista si están en orden incorrecto (Cormen et al., 2022). Este proceso se repite hasta que no requiere realizar más intercambios dando a entender que la lista está ordenada

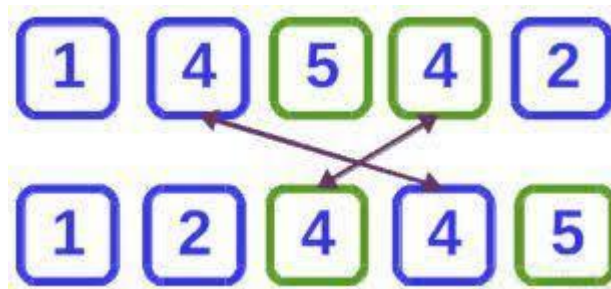


Imagen 1 funcionamiento del algoritmo de intercambio

2.1.1. codificación

```
#include <iostream>
void intercambio(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            // Compara elementos adyacentes
            if (arr[j] > arr[j + 1]) {
                // Intercambia si están en el orden incorrecto
                std::swap(arr[j], arr[j + 1]);
            }
        }
    }
}

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

    std::cout << "Arreglo original: ";
    for (int i = 0; i < n; i++) {
        std::cout << arreglo[i] << " ";
    }
}
```

```
// Llamada a la función de intercambio
intercambio(arreglo, n);

std::cout << "\nArreglo ordenado: ";
for (int i = 0; i < n; i++) {
    std::cout << arreglo[i] << " ";
}

return 0;
}
```

2.1.2. Ejecución

```
PS C:\Users\csar_\Codes-C++\output> & .\'Intercambio.exe'
● Arreglo original: 64 34 25 12 22 11 90
  Arreglo ordenado: 11 12 22 25 34 64 90
```

2.1. Análisis

Función “*Intercambio*”

- La función **intercambio** toma un arreglo **arr** y su longitud **n** como parámetros.
- Utiliza dos bucles anidados para comparar y, si es necesario, intercambiar elementos adyacentes.
- El bucle externo (**for (int i = 0; i < n - 1; i++)**) controla el número total de pasadas a través del arreglo.
- El bucle interno (**for (int j = 0; j < n - i - 1; j++)**) compara y, si es necesario, intercambia elementos.
- La condición **if (arr[j] > arr[j + 1])** verifica si el elemento actual es mayor que el siguiente.
- Si la condición es verdadera, **std::swap(arr[j], arr[j + 1])** intercambia los elementos.

Función principal “*Main*”

- Se declara un arreglo de números desordenados llamado **arreglo**.
- La variable **n** se calcula como la longitud del arreglo.
- Se imprime el arreglo original utilizando un bucle **for**.
- Se llama a la función **intercambio** para ordenar el arreglo.
- Se imprime el arreglo ordenado utilizando otro bucle **for**.

2.2. Conclusiones

En resumen, el algoritmo de ordenamiento por intercambio, es simple pero menos eficiente en comparación con otros algoritmos más avanzados. Funciona intercambiando repetidamente elementos adyacentes hasta que el arreglo esté ordenado. Aunque es fácil de entender e implementar, su rendimiento no es óptimo, especialmente en conjuntos de datos grandes.

3. BURBUJA (BUBBLE SORT).

3.1. Definición.

El algoritmo de ordenamiento Burbuja (bubble sort en inglés) funciona intercambiando repetidamente elementos adyacentes de la lista si están en el orden incorrecto. Es decir, si el elemento de la izquierda es mayor a la derecha, se intercambian. Este proceso se repite hasta que se completa el barrido se realizara ningún intercambio, dando a entender que la lista está ordenada (Cormen et al., 2022).

Específicamente, bubble sort parte del inicio de la lista y compara cada par de elementos adyacentes. Si el primer elemento es mayor que el segundo, los intercambia. Luego pasa al segundo y tercer elemento, intercambiándolos si están desordenados, y así sucesivamente. Cuando llega al final, vuelve a empezar desde el principio. Esto se repite hasta que en una iteración no se realiza ningún intercambio, indicando que la lista está ordenada (Cormen et al., 2022).

Original:	03	07	11	02	09	01	08	05	10	06	04
Pasada 1:	03	07	02	09	01	08	05	10	06	04	11
Pasada 2:	03	02	07	01	08	05	09	06	04	10	11
Pasada 3:	02	03	01	07	05	08	06	04	09	10	11
Pasada 4:	02	01	03	05	07	06	04	08	09	10	11
Pasada 5:	01	02	03	05	06	04	07	08	09	10	11

Imagen 2 Funcionamiento de algoritmo Burbuja de forma gráfica.

3.2. Propiedades.

- Complejidad del espacio: $O(1)$.
- Rendimiento en el mejor caso: $O(n)$.
- Rendimiento promedio de casos: $O(n*n)$.
- Rendimiento en el peor de los casos: $O(n*n)$.
- Estable: Sí.

main.cpp

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

```

int main(){
    Lista lista;
    do {
        system("cls");
        std::cout << "\n\tORDENAMIENTO BURBUJA CON LISTA SIMPLE\n\n" <<
            "1. Insertar al inicio\n" <<
            "2. Eliminar\n" <<
            "3. Mostrar\n" <<
            "4. Ordenamiento metodo burbuja\n" <<
            "5. Salir\n\n";
        switch (ingresar_enteros("Ingresar la opcion: ")) {
        case 1:
            lista.insertarAlInicio(ingresar_enteros("\n\nIngresar un numero:
"));
            std::cout << std::endl;
            system("pause");
            break;
        case 2:
            lista.eliminar(ingresar_enteros("\n\nIngresar el numero a
eliminar: "));
            std::cout << std::endl;
            system("pause");
            break;
        case 3:
            lista.imprimir();
            std::cout << std::endl;
            system("pause");
            break;
        case 4:
            lista.ordenarBurbuja();
            system("pause");
            break;
        case 5:
            return 0;
        default:
            break;
        }
    } while (true);
}

```

ListaSimple.h

```

#pragma once
#include <iostream>
// Definición de la clase Nodo
class Nodo {
public:

```

```

    int dato;
    Nodo* siguiente;

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

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

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

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

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

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

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

        // Si la clave no se encuentra en la lista, no hay nada que eliminar

```

```

        if (temp == nullptr) {
            std::cout << "\nEl elemento " << clave << " no esta en la lista."
<< std::endl;
            return;
        }

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

        delete temp;
    }

    // Método para ordenar la lista usando el algoritmo de burbuja
    void ordenarBurbuja() {
        if (cabeza == nullptr || cabeza->siguiente == nullptr) {
            // La lista está vacía o tiene solo un elemento, no es necesario
ordenar
            return;
        }

        bool intercambiado;
        Nodo* actual;
        Nodo* anterior = nullptr;

        do {
            intercambiado = false;
            actual = cabeza;

            while (actual->siguiente != anterior) {
                if (actual->dato > actual->siguiente->dato) {
                    // Intercambiar los valores de los nodos
                    int temp = actual->dato;
                    actual->dato = actual->siguiente->dato;
                    actual->siguiente->dato = temp;
                    intercambiado = true;
                }
                actual = actual->siguiente;
            }
            anterior = actual;
        } while (intercambiado);
    }

    // Destructor para liberar la memoria asignada a los nodos de la lista
    ~Lista() {

```

```

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

```

Validacion.h

```

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

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

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

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

```

3.3. Conclusiones.

El método de burbuja es fácil de entender e implementar, su rendimiento no es ideal para conjuntos de datos grandes. Otros algoritmos de ordenamiento son generalmente preferidos en la práctica para mejorar la eficiencia.

4. ORDENACIÓN POR DISTRIBUCIÓN

4.1. Introducción

El algoritmo de ordenación por distribución, también conocido como "counting sort", es un método de ordenación eficiente que funciona bien cuando el rango de valores de la entrada es relativamente pequeño en comparación con el tamaño de la entrada. Este algoritmo no compara los elementos de la lista como lo hacen los algoritmos de ordenación comparativa, como el quicksort o el mergesort, lo que lo hace especialmente útil en situaciones donde los valores a ordenar son números enteros dentro de un rango limitado.

4.2. Definición:

El algoritmo de ordenación por distribución funciona contando el número de ocurrencias de cada elemento único en la lista de entrada y utilizando esta información para reconstruir una lista ordenada. Es especialmente eficiente cuando el rango de valores es pequeño y conocido de antemano.

4.3. Características:

- Eficiente para listas con un rango de valores pequeño.
- No compara directamente los elementos de la lista.
- Requiere conocimiento previo del rango de valores de la lista.
- Tiene una complejidad temporal lineal $O(n + k)$, donde "n" es el tamaño de la lista y "k" es el rango de valores.

4.4. Ejemplo:

Supongamos que queremos ordenar una lista de números enteros utilizando el algoritmo de ordenación por distribución implementado en C++ de forma orientada a objetos y separada por archivos.

CountingSort.h

```
#ifndef ORDENAMIENTO_COUNTINGSORT_H
#define ORDENAMIENTO_COUNTINGSORT_H
```



```
#include <list>

class CountingSort {
public:
    static void ordenar(std::list<int>& lst);
};

#endif
```

CountingSort.cpp

```
#include "CountingSort.h"
#include <algorithm>

void CountingSort::ordenar(std::list<int>& lst) {
    int max = *std::max_element(lst.begin(), lst.end());
    int min = *std::min_element(lst.begin(), lst.end());
    int rango = max - min + 1;

    std::list<int> conteo(rango), salida;
    for (int num : lst) {
        conteo[num - min]++;
    }

    for (auto it = conteo.begin(); std::next(it) != conteo.end(); ++it) {
        *std::next(it) += *it;
    }

    for (auto it = lst.rbegin(); it != lst.rend(); ++it) {
        salida.emplace_front(*it);
        conteo[*it - min]--;
    }

    lst = salida;
}
```

main.cpp

```
#include <iostream>
#include <list>
#include "CountingSort.h"
```

```

int main() {
    std::list<int> lst = {4, 2, 2, 8, 3, 3, 1};
    std::cout << "Lista original: ";
    for (int num : lst) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    CountingSort::ordenar(lst);

    std::cout << "Lista ordenada: ";
    for (int num : lst) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

4.5. Conclusiones

El algoritmo de ordenación por distribución, conocido también como "counting sort", representa una herramienta valiosa en el arsenal de algoritmos de ordenación gracias a su eficiencia en situaciones específicas. A diferencia de los algoritmos de comparación, como el quicksort o el mergesort, que operan comparando directamente los elementos de la lista, el counting sort trabaja contando las ocurrencias de cada elemento único y luego utilizando esta información para ordenar la lista. Esta característica lo hace particularmente eficiente cuando se trabaja con un rango de valores limitado, ya que su complejidad temporal es lineal en relación con el tamaño de la lista más el tamaño del rango de valores. Sin embargo, esta eficiencia está condicionada por la necesidad de conocer previamente el rango de valores, lo que puede limitar su aplicabilidad en situaciones donde este rango es desconocido o muy amplio. A pesar de estas limitaciones, el counting sort sigue siendo una opción valiosa en escenarios donde se cumple su suposición fundamental de un rango de valores acotado, proporcionando una alternativa eficiente y sencilla para la ordenación de listas.

5. SHELL SORT

La clasificación Shell es principalmente una variación de la clasificación por inserción. En la ordenación por inserción, movemos los elementos solo una posición hacia adelante. Cuando es necesario mover un elemento mucho más

adelante, intervienen muchos movimientos. La idea de Shell Sort es permitir el intercambio de artículos lejanos. En Shell sort, ordenamos la matriz h para un valor grande de h . Seguimos reduciendo el valor de h hasta que se convierte en 1. Se dice que una matriz está ordenada en h si todas las sublistas de cada h -ésimo elemento están ordenadas.

5.1. Algoritmo:

Paso 1 - Iniciar

Paso 2 - Inicialice el valor del tamaño del espacio. Ejemplo: h

Paso 3 : divida la lista en subpartes más pequeñas. Cada uno debe tener intervalos iguales a h

Paso 4: ordene estas sublistas mediante ordenación por inserción.

Paso 5: repita este paso 2 hasta que la lista esté ordenada.

Paso 6: imprima una lista ordenada.

Paso 7 – Detente.

5.2. Código:

A continuación, se muestra la implementación de ShellSort.

```
// C++ implementation of Shell Sort
#include <iostream>
using namespace std;

/* function to sort arr using shellSort */
int shellSort(int arr[], int n)
{
    // Start with a big gap, then reduce the gap
    for (int gap = n/2; gap > 0; gap /= 2)
    {
        // Do a gapped insertion sort for this gap size.
        // The first gap elements a[0..gap-1] are already
        in gapped order
        // keep adding one more element until the entire
        array is
        // gap sorted
        for (int i = gap; i < n; i += 1)
        {
```

```

        // add a[i] to the elements that have been gap
sorted
        // save a[i] in temp and make a hole at
position i
        int temp = arr[i];

        // shift earlier gap-sorted elements up until
the correct
        // location for a[i] is found
        int j;
        for (j = i; j >= gap && arr[j - gap] > temp; j
-= gap)
            arr[j] = arr[j - gap];
            arr[j] = temp;
        }
    }
    return 0;
}

void printArray(int arr[], int n)
{
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}

int main()
{
    int arr[] = {12, 34, 54, 2, 3}, i;
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Array before sorting: \n";
    printArray(arr, n);
    shellSort(arr, n);
    cout << "\nArray after sorting: \n";
}

```

```

    printArray(arr, n);

    return 0;
}

```

5.3. Conclusiones

En resumen, Shell Sort es una variante eficiente de la clasificación por inserción que permite el intercambio de elementos distantes, mejorando significativamente el rendimiento del algoritmo. Su enfoque de dividir la lista en sublistas y ordenarlas gradualmente resulta en una ordenación más rápida y eficiente en comparación con la clasificación por inserción tradicional.

6. QUICK SORT

El Quicksort es un algoritmo de ordenación eficiente y ampliamente utilizado. se basa en la estrategia de "dividir y conquistar". El algoritmo funciona dividiendo la lista de elementos en particiones, ordenando cada partición de manera independiente y luego combinándolas para obtener la lista ordenada final.

A continuación, se presenta una descripción básica del funcionamiento del Quicksort:

6.1. Algoritmo:

1. Si la longitud de la lista es 0 o 1, entonces ya está ordenada y se devuelve la lista.
2. Elegir un elemento de la lista como pivote.
3. Particionar la lista alrededor del pivote de manera que los elementos menores que el pivote estén a su izquierda y los mayores estén a su derecha.
4. Aplicar Quicksort recursivamente a las sublistas generadas por la partición (elementos menores y mayores).
5. La lista ordenada es la concatenación de la lista de elementos menores, el pivote y la lista de elementos mayores.

6.2. Código:

A continuación, se muestra la implementación de QuickSort.

quicksort.h

```

#include <iostream>

struct Nodo {
    int dato;

```

```

    Nodo* siguiente;

    Nodo* anterior;

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

};

class ListaEnlazada {
private:
    Nodo* cabeza;

public:
    ListaEnlazada();
    ~ListaEnlazada();

    void insertar(int valor);
    void mostrar();
    void quicksort();

private:
    Nodo* obtenerUltimoNodo();
    Nodo* Particionar(Nodo* inicio, Nodo* fin);
    Nodo* QuicksortRecursivo(Nodo* inicio, Nodo* fin);
};

```

quicksort.cpp

```

#include "quicksort.h"

ListaEnlazada::ListaEnlazada() : cabeza(nullptr) {}

ListaEnlazada::~~ListaEnlazada() {
    while (cabeza != nullptr) {
        Nodo* temp = cabeza;
        cabeza = cabeza->siguiente;
        delete temp;
    }
}

void ListaEnlazada::insertar(int valor) {
    Nodo* nuevoNodo = new Nodo(valor);
    if (cabeza == nullptr) {

```

```

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

void ListaEnlazada::mostrar() {
    Nodo* temp = cabeza;
    while (temp != nullptr) {
        std::cout << temp->dato << " ";
        temp = temp->siguiente;
    }
    std::cout << "\n";
}

void ListaEnlazada::quicksort() {
    cabeza = QuicksortRecursivo(cabeza, obtenerUltimoNodo());
}

Nodo* ListaEnlazada::obtenerUltimoNodo() {
    Nodo* temp = cabeza;
    while (temp != nullptr && temp->siguiente != nullptr) {
        temp = temp->siguiente;
    }
    return temp;
}

Nodo* ListaEnlazada::Particionar(Nodo* inicio, Nodo* fin) {
    int pivote = fin->dato;
    Nodo* i = inicio->anterior;

    for (Nodo* j = inicio; j != fin; j = j->siguiente) {
        if (j->dato <= pivote) {
            i = (i == nullptr) ? inicio : i->siguiente;
            std::swap(i->dato, j->dato);
        }
    }
}

```

```

    }

    i = (i == nullptr) ? inicio : i->siguiente;
    std::swap(i->dato, fin->dato);
    return i;
}

Nodo* ListaEnlazada::QuicksortRecursivo(Nodo* inicio, Nodo* fin) {
    if (inicio != nullptr && inicio != fin && inicio != fin->siguiente) {
        Nodo* pivote = Particionar(inicio, fin);
        QuicksortRecursivo(inicio, pivote->anterior);
        QuicksortRecursivo(pivote->siguiente, fin);
    }

    return inicio;
}

```

Main.cpp

```

#include "quicksort.cpp"

int main() {
    ListaEnlazada lista;

    int opcion;
    do {
        std::cout << "\n*** Menu***\n";
        std::cout << "1. Insertar elemento\n";
        std::cout << "2. Mostrar lista\n";
        std::cout << "3. Aplicar Quicksort\n";
        std::cout << "0. Salir\n";
        std::cout << "Ingrese su opcion: ";
        std::cin >> opcion;

        switch (opcion) {
            case 1: {
                int elemento;
                std::cout << "Ingrese un elemento: ";
                std::cin >> elemento;
                lista.insertar(elemento);
                break;
            }
        }
    }
}

```



```

    case 2:
        std::cout << "Lista actual: ";
        lista.mostrar();
        break;
    case 3:
        std::cout << "Aplicando Quicksort...\n";
        lista.quicksort();
        std::cout << "Lista ordenada: ";
        lista.mostrar();
        break;
    case 0:
        std::cout << "Saliendo del programa.\n";
        break;
    default:
        std::cout << "Opción no válida. Inténtelo de nuevo.\n";
        break;
}

} while (opcion != 0);

return 0;
}

```

6.3. Conclusiones

El algoritmo Quicksort destaca por su eficiencia y simplicidad en la implementación. Su estrategia "divide y conquista" permite ordenar listas de manera rápida y es especialmente efectivo en conjuntos de datos grandes. La elección inteligente del pivote y la partición eficiente contribuyen a su rendimiento. Sin embargo, es importante considerar que Quicksort puede tener un rendimiento deficiente en casos específicos, como listas ya ordenadas o casi ordenadas, lo que puede afectar su eficacia en ciertos escenarios. A pesar de esto, Quicksort es una elección popular en la práctica y se ha convertido en un algoritmo de referencia para la ordenación.

7. ALGORITMOS DE ORDENACIÓN EXTERNA

8. MEZCLA DIRECTA

8.1. Introducción

El algoritmo de mezcla directa es un método de ordenación interna que utiliza el enfoque de "divide y conquista" para ordenar una lista de elementos.

En este documento, exploraremos el algoritmo de mezcla directa, su funcionamiento y su implementación en C++.

8.2. Algoritmo de Mezcla Directa

El algoritmo de mezcla directa sigue los siguientes pasos:

1. **División:** Divide la lista en dos sub-listas de tamaño aproximadamente igual.
2. **Ordenación Recursiva:** Ordena cada sub-lista de forma recursiva aplicando el algoritmo de mezcla directa.
3. **Mezcla:** Combina las dos sub-listas ordenadas para obtener la lista ordenada final.

8.3. Implementación del Algoritmo de Mezcla Directa

A continuación, se presenta una implementación del algoritmo de mezcla directa en C++:

```
#include <iostream>
#include <vector>
```

```
void merge(std::vector<int>& arr, int inicio, int medio, int fin) {
    int n1 = medio - inicio + 1;
    int n2 = fin - medio;

    std::vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[inicio + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[medio + 1 + j];

    int i = 0, j = 0, k = inicio;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
    }
}
```

```

        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(std::vector<int>& arr, int inicio, int fin) {
    if (inicio < fin) {
        int medio = inicio + (fin - inicio) / 2;

        mergeSort(arr, inicio, medio);
        mergeSort(arr, medio + 1, fin);

        merge(arr, inicio, medio, fin);
    }
}

void imprimirArray(const std::vector<int>& arr) {
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> arr = {12, 11, 13, 5, 6, 7};
    int n = arr.size();

    std::cout << "Arreglo original:\n";
    imprimirArray(arr);
}

```

```
mergeSort(arr, 0, n - 1);  
  
std::cout << "Arreglo ordenado:\n";  
imprimirArray(arr);  
  
return 0;  
}
```

8.4. Conclusiones

El algoritmo de mezcla directa ofrece una forma eficiente de ordenar listas de elementos utilizando el enfoque de divide y conquista. Su implementación en C++ proporciona una herramienta poderosa para ordenar datos de manera eficiente.

9. MEZCLA NATURAL.

9.1. Definición.

Permiten organizar los elementos de un archivo, de forma ascendente o descendente. Su algoritmo se basa en realizar particiones tomando secuencias ordenadas de máxima longitud en lugar de secuencias ordenadas de tamaño fijo previamente determinadas, como la intercalación directa. Posteriormente se realiza la fusión de esas secuencias ordenadas, alternándolas entre los 2 archivos auxiliares.

Repitiendo este proceso, se logra que el archivo quede completamente ordenado. Para aplicar este algoritmo, se necesitarán 4 archivos, 2 serán considerados de entrada y 2 de salida, alternativamente en cada paso de algoritmo. El proceso termina cuando al finalizar un paso, el segundo archivo de salida quede vacío y el primero queda completamente ordenado.

Mezcla Natural

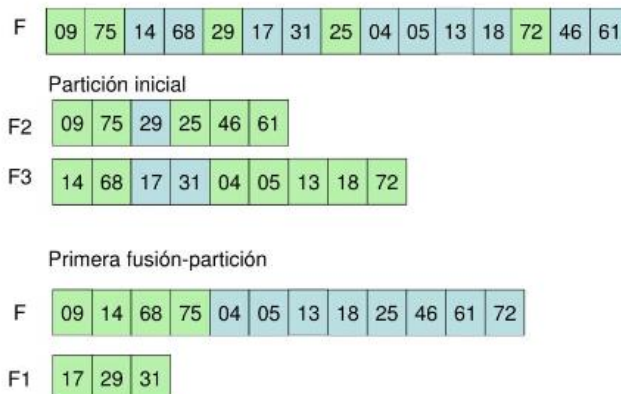


Imagen 3 Ciclo de vida del TDA

10. ALGORITMO DE BÚSQUEDA - SECUENCIAL

El algoritmo de búsqueda secuencial, también conocido como búsqueda lineal, es un método simple y directo para encontrar un elemento específico en una colección de datos. Este algoritmo no requiere que los datos estén ordenados y es fácil de implementar en diferentes tipos de estructuras de datos como arreglos, listas, conjuntos, entre otros. Sin embargo, a medida que aumenta el tamaño de la colección, la búsqueda secuencial puede volverse menos eficiente en comparación con otros algoritmos de búsqueda, especialmente cuando se trata de grandes conjuntos de datos, ya que su tiempo de ejecución es proporcional al tamaño de la colección.

10.1. Comportamiento o funcionamiento de una búsqueda secuencial

La idea principal detrás de la búsqueda secuencial es comparar el elemento buscado con cada elemento de la colección en orden secuencial. Si el elemento buscado coincide con alguno de los elementos de la colección, se retorna la posición donde se encuentra; de lo contrario, se indica que el elemento no está presente en la colección. Esto lo realiza de la siguiente manera:

1. Se toma la colección de datos donde se hará la búsqueda
2. Compara cada elemento actual con el valor buscado
3. **Si** elemento actual coincide con el valor buscado
 - a. se retorna la posición actual del elemento
4. En **caso contrario**: El algoritmo avanza al siguiente elemento en la colección.
5. Se repiten los pasos 2-3 hasta que se encuentre el elemento buscado o se recorra toda la colección sin éxito.
6. Si se encuentra el elemento, se retorna su posición; de lo contrario, se indica que el elemento no está presente en la colección.

10.2. Ejemplo: Código del algoritmo de Búsqueda - Secuencial

Para un mejor entendimiento del mismo se explicará este tema con un código.

busqueda.h

```
#pragma once
#include <list>
std::list<int>::iterator busquedaSecuencial(std::list<int>& lista, int
elemento);
```

Busqueda.cpp

```
// Archivo de implementación (busqueda.cpp)
#include "busqueda.h"
std::list<int>::iterator busquedaSecuencial(std::list<int>& lista, int
elemento) {
    for (auto it = lista.begin(); it != lista.end(); ++it) {
        if (*it == elemento) {
            return it; // Devuelve un iterador apuntando al elemento
encontrado
        }
    }
    return lista.end(); // Devuelve el iterador al final si el
elemento no se encuentra
}
```

Main.cpp

```
#include <iostream>
#include "busqueda.cpp"
int main() {
    std::list<int> miLista = {1, 2, 9, 3, 1, 4, 5,8};
    int elementoBuscado;
    std::cout<<"Ingrese el elemento a buscar"<<std::endl;
    std::cin>>elementoBuscado;
    auto resultado = busquedaSecuencial(miLista, elementoBuscado);
    // Verificar si se encontró el elemento
    if (resultado != miLista.end()) {
        std::cout << "Elemento encontrado en la lista." << std::endl;
    } else {
        std::cout << "Elemento no encontrado en la lista." <<
std::endl;
    }

    return 0;
}
```

```

Ingrese el elemento a buscar
3
Elemento encontrado en la lista.

Process returned 0 (0x0)   execution time : 1.945 s
Press any key to continue.

```

Imagen 4: Ejecución del código ejemplo: Algoritmo de búsqueda secuencial

11.ALGORITMOS DE BÚSQUEDA BINARIA

11.1. Introducción

La búsqueda binaria es una técnica fundamental en el campo de la informática y la ciencia de la computación. Este manual se enfoca en proporcionar una comprensión detallada de los algoritmos de búsqueda binaria, sus aplicaciones y su implementación práctica.

11.2. Definición del algoritmo

El algoritmo de búsqueda binaria es una técnica eficiente para encontrar un elemento específico en una lista ordenada de elementos. Funciona dividiendo repetidamente el conjunto de datos en mitades y descartando la mitad en la que se sabe que el elemento buscado no puede estar presente, hasta que se encuentre el elemento deseado o se determine que no está en la lista. Este enfoque aprovecha el hecho de que la lista está ordenada para reducir el número de comparaciones necesarias, logrando así una complejidad temporal de $O(\log n)$, donde n es el número de elementos en la lista. El algoritmo de búsqueda binaria es ampliamente utilizado en diversos campos, como la informática, la matemática y la ingeniería, debido a su eficiencia y su capacidad para manejar grandes conjuntos de datos de manera rápida y efectiva.

Cómo funciona	Cuándo usarla	Ejemplo práctico
Divide una lista ordenada en mitades y busca en la mitad correcta.	Para listas grandes y ordenadas en las que la posición del elemento a buscar es desconocida.	Buscar una palabra en un diccionario de páginas web en orden alfabético.

Imagen 5 Descripción del algoritmo

11.3. Particularidades de los algoritmos de búsqueda binaria

- La lista debe estar ordenada de acuerdo al valor de la clave, para realizar la búsqueda.
- Se reduce el vector a la mitad en comparación a $<0>$ al elemento buscado.
- Al estar ordenada se obtiene el número total de registros.
- Es aplicable a árboles binarios y listas.
- El esfuerzo mínimo es 1, el medio es $1 \log_2 n$, el máximo $\log_2 n$

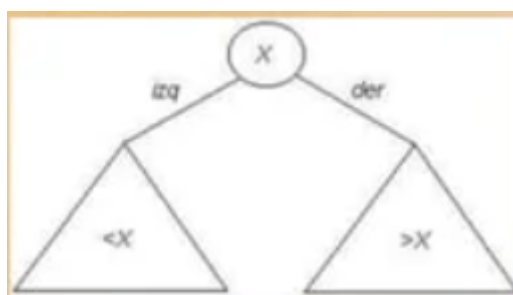


Imagen 6 Comparación de elementos

11.4. Uso de los algoritmos de búsqueda binaria

Los algoritmos de búsqueda binaria se utilizan en una amplia variedad de aplicaciones en informática y otros campos. Algunos de los usos más comunes incluyen:

1. Búsqueda en bases de datos: En sistemas de gestión de bases de datos, la búsqueda binaria se utiliza para recuperar rápidamente información almacenada en índices ordenados, lo que acelera la recuperación de datos y mejora el rendimiento de las consultas.

2. Búsqueda en árboles binarios de búsqueda: Los árboles binarios de búsqueda son estructuras de datos comunes que se utilizan para almacenar y organizar datos de manera eficiente. Los algoritmos de búsqueda binaria se utilizan para recorrer estos árboles de manera eficiente y recuperar información.

3. Búsqueda en listas ordenadas: Cuando se trabaja con listas ordenadas de elementos, la búsqueda binaria permite encontrar rápidamente un elemento específico sin tener que recorrer la lista completa, lo que resulta en un tiempo de búsqueda más rápido en comparación con otros métodos de búsqueda.

4. Algoritmos de ordenación: Algunos algoritmos de ordenación, como el algoritmo de mezcla (Merge Sort) y el algoritmo de ordenación rápida (Quick Sort), utilizan la búsqueda binaria como parte de su proceso de ordenación.

5. Búsqueda en espacios de claves: En aplicaciones como la criptografía y la seguridad informática, la búsqueda binaria se utiliza para buscar y comparar claves de manera eficiente en grandes espacios de claves.

6. Búsqueda en juegos y simulaciones: Los algoritmos de búsqueda binaria se utilizan para realizar búsquedas eficientes en estructuras de datos que representan entidades y objetos en juegos y simulaciones, lo que permite una interacción más rápida y fluida con el entorno virtual.

Problema: buscar un elemento dado dentro de una lista ordenada

Elemento buscado: 15

Lista: 3 7 11 15 22 24 32 33 38 40

Solución: ir dividiendo sucesivamente la lista por la mitad, hasta encontrar el dato buscado en el último o en el primer elemento de una de las mitades.

1er paso: 3 7 11 15 22 24 32 33 38 40

2do paso: 3 7 11 15 22

3er paso: 11 15 22

4to paso: 15 22

Imagen 7 Ejemplo de una búsqueda binaria

11.5. Algoritmo y codificación de la búsqueda binaria

Suponiendo que la lista esté almacenada como un arreglo, donde los índices de la lista son bajo=0 y alto=n-1, donde n es el número de elementos del arreglo, los pasos a seguir son:

1. Calcular el índice del punto central del arreglo:

$$\text{Central} = (\text{bajo} + \text{alto}) / 2 \quad (\text{División entera})$$
2. Comparar el valor de este elemento central con la clave:
 - Si $a[\text{central}] < \text{clave}$, la nueva sublista de búsqueda tiene por valores extremos de su rango bajo=central+1...alto
 - Si $\text{clave} < a[\text{central}]$, la nueva sublista de búsqueda tiene por valores extremos de su rango bajo...central-1.

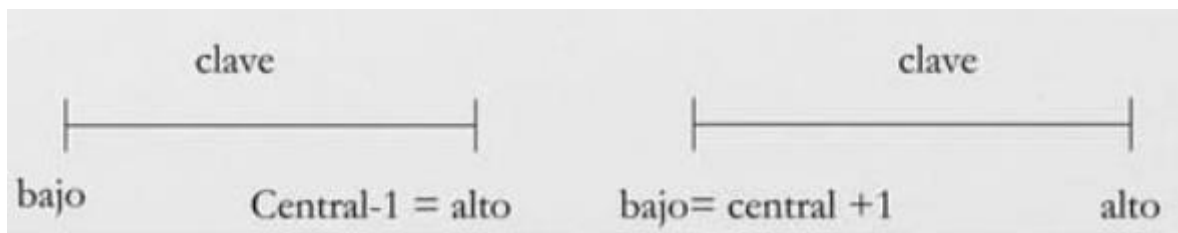


Imagen 8 Comparación del elemento central con la clave

El algoritmo termina o bien porque se ha encontrado la clave, o bien porque el valor de bajo excede a alto y el algoritmo devuelve el indicador de fallo -1 (Búsqueda no encontrada)

11.6. Ejemplo:

Sea el arreglo $A\{-8,4,5,9,12,18,25,40,60\}$, buscar la clave 40:

1) Cálculo del punto central

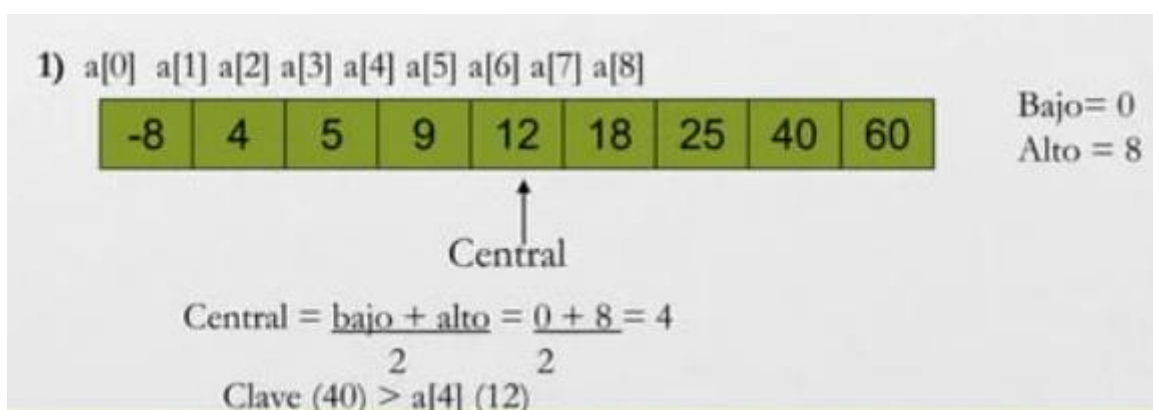


Imagen 9 Cálculo del punto central y primera comparación del algoritmo

2) Buscar en la sublista derecha

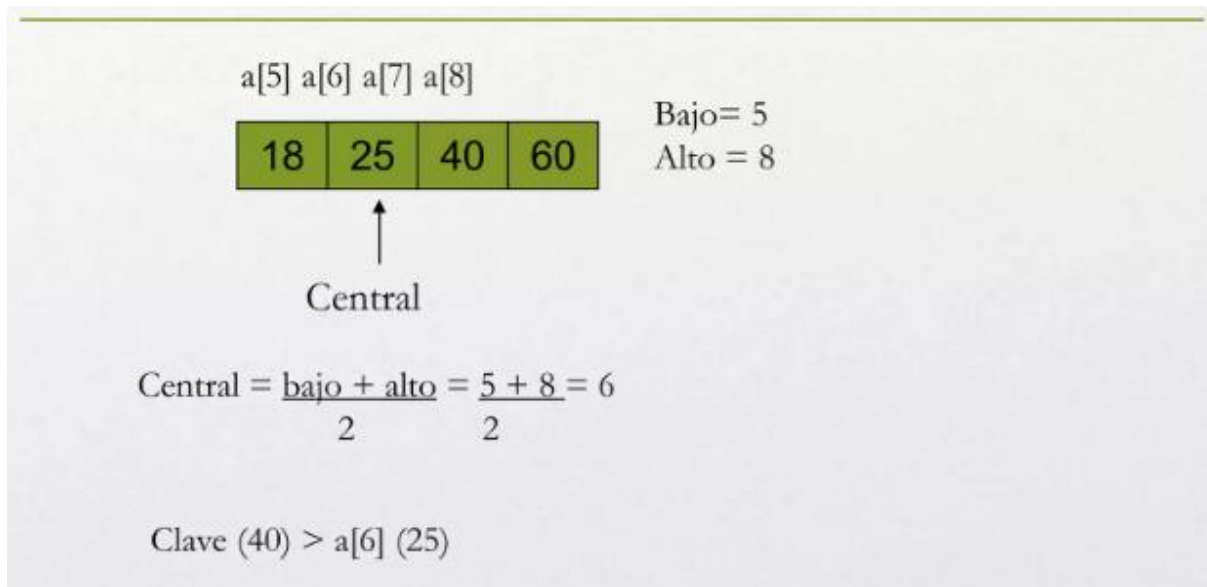


Imagen 10 Segunda comparación del algoritmo

3) Buscar en la sublista derecha

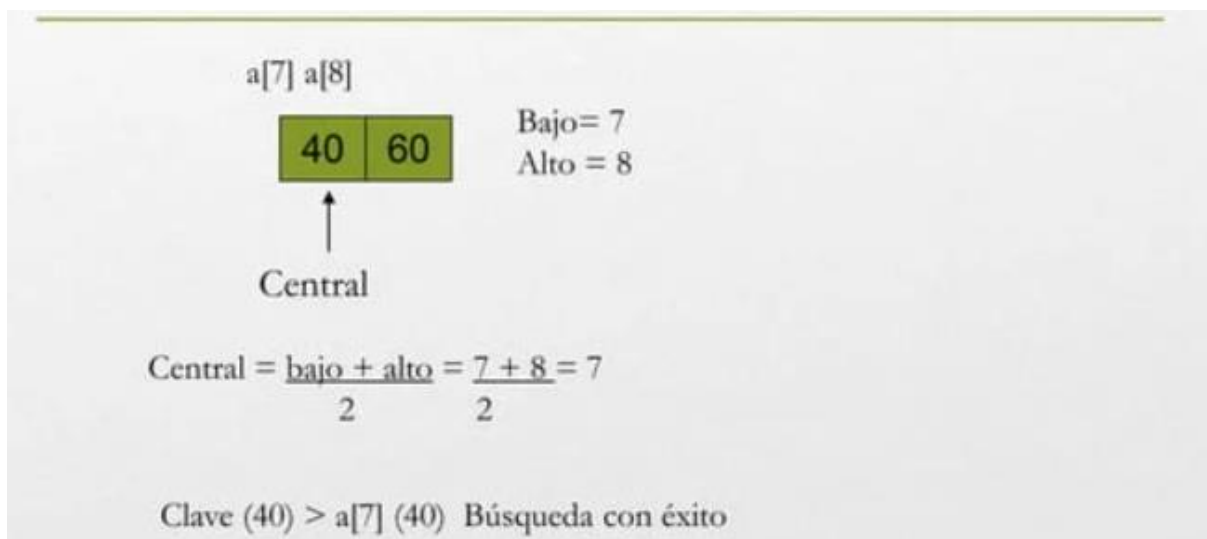


Imagen 11 Segunda comparación del algoritmo

Como se puede observar, este algoritmo ha requerido 3 comparaciones, frente a las 8 que hubiera necesitado la búsqueda secuencial.

```

int busquedaBin(int a[], int n, int clave)
{
    int central, bajo, alto;

```

```

int valorCentral;
bajo = 0;
alto = n - 1;
while (bajo <= alto)
{
    central = (bajo + alto)/2;           // índice de elemento central
    valorCentral = a[central];          // valor del índice central
    if (clave == valorCentral)
        return central;                // encontrado, devuelve
posición
    else if (clave < valorCentral)
        alto = central - 1;             // ir a sublista inferior
    else
        bajo = central + 1;             // ir a sublista superior
}
return -1;                             //elemento no encontrado
}

```

11.7. Conclusiones

En conclusión, los algoritmos de búsqueda binaria ofrecen una notable mejora en términos de eficiencia y rendimiento en comparación con los algoritmos de búsqueda secuencial. Mientras que la búsqueda secuencial implica recorrer cada elemento de una lista uno por uno, lo que resulta en un tiempo de búsqueda proporcional al tamaño de la lista ($O(n)$), la búsqueda binaria aprovecha la estructura ordenada de los datos para reducir significativamente el número de comparaciones necesarias.

12. ALGORITMO DE BÚSQUEDA- HASH

12.1. Introducción

En el ámbito de la informática y la programación, los algoritmos de búsqueda juegan un papel fundamental en la organización y recuperación eficiente de datos. Uno de los métodos más importantes utilizados en estos algoritmos es el Hashing, una técnica que permite asociar claves con valores de forma rápida y eficiente. Este manual tiene como objetivo proporcionar una comprensión clara y concisa del concepto de Hashing y su aplicación en los algoritmos de búsqueda.

12.2. ¿Qué es el Hashing?

El Hashing es una técnica que se utiliza para mapear datos de tamaño variable a datos de tamaño fijo. Esto se logra mediante una función denominada función hash, que toma una entrada (o 'clave') y la transforma en una dirección única en una tabla de dispersión (o 'hash table'). Esta dirección

se utiliza entonces para acceder al valor asociado con esa clave de forma rápida.

12.3. Componentes del Hashing

Función Hash: Es el corazón del proceso de Hashing. Esta función toma una clave como entrada y devuelve una dirección única en la tabla de dispersión. Una buena función hash debe ser rápida de calcular y producir distribuciones uniformes de direcciones.

Tabla de Dispersión (Hash Table): Es una estructura de datos que almacena los pares clave-valor. Está compuesta por un conjunto de 'buckets' o 'slots' donde se almacenan estos pares. La dirección devuelta por la función hash determina en qué 'bucket' se almacenará el par clave-valor.

Colisión: Ocurre cuando dos claves diferentes generan la misma dirección hash. Esto puede ocurrir debido a limitaciones en la función hash o a la naturaleza del conjunto de claves. Las colisiones deben ser manejadas adecuadamente para garantizar la integridad de la tabla de dispersión.

12.4. Operaciones de Hashing

Insertión (Insert): Consiste en añadir un nuevo par clave-valor a la tabla de dispersión. Para ello, se calcula la dirección hash de la clave y se inserta el par en el 'bucket' correspondiente. En caso de colisión, se utiliza una estrategia de resolución de colisiones.

Búsqueda (Search): Implica recuperar el valor asociado a una clave dada. Se calcula la dirección hash de la clave y se busca en el 'bucket' correspondiente. Si se encuentran colisiones, se resuelven de acuerdo a la estrategia establecida.

Eliminación (Delete): Permite eliminar un par clave-valor de la tabla de dispersión. Se busca la clave en la tabla, se elimina el par si se encuentra y se ajusta la estructura de la tabla si es necesario.

12.5. Aplicaciones del Hashing en Algoritmos de Búsqueda

Búsqueda Rápida: El hashing permite una búsqueda rápida y eficiente de elementos en grandes conjuntos de datos. La complejidad temporal de las operaciones de búsqueda, inserción y eliminación es $O(1)$ en promedio, lo que significa que el tiempo de ejecución no depende del tamaño de los datos.

Gestión de Colisiones: A través de diversas técnicas de resolución de colisiones, como el encadenamiento o la exploración lineal, el hashing permite

gestionar eficazmente las colisiones y mantener un rendimiento óptimo incluso en presencia de claves duplicadas.

Estructuras de Datos Eficientes: Muchas estructuras de datos populares, como los conjuntos y los mapas, se implementan utilizando tablas de dispersión basadas en hashing. Estas estructuras ofrecen un equilibrio entre la rapidez de acceso y la flexibilidad en la manipulación de datos.

12.6. Conclusiones

El Hashing es una técnica poderosa y versátil utilizada en una amplia gama de aplicaciones informáticas, especialmente en algoritmos de búsqueda. Proporciona una manera eficiente de asociar claves con valores y permite realizar operaciones de búsqueda, inserción y eliminación en tiempo constante en promedio. Comprender los fundamentos del hashing y su aplicación en los algoritmos de búsqueda es esencial para cualquier desarrollador de software que busque optimizar el rendimiento y la eficiencia de sus aplicaciones.

13. CONCEPTO DE ARBOLES

13.1. Introducción

Los árboles en C++ son una estructura de datos fundamental que se utiliza para organizar y almacenar datos de manera jerárquica. Se componen de nodos interconectados, donde cada nodo puede tener cero o más nodos hijos, excepto el nodo superior, conocido como raíz, que no tiene nodos padre. Esta estructura de datos refleja la relación de padres e hijos de una manera natural y se utiliza en una amplia gama de aplicaciones, desde la organización de datos en bases de datos hasta la implementación de algoritmos de búsqueda y ordenación.

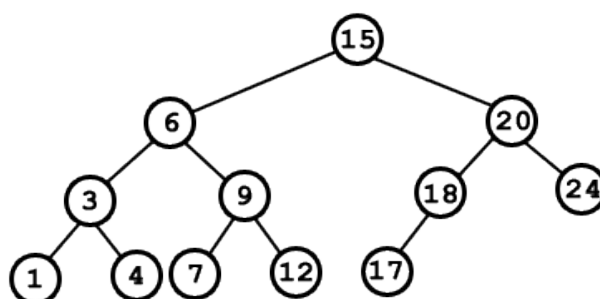


Imagen 12: Árbol básico

13.2. Características:

13.2.1. [Jerarquía y estructura recursiva:](#)

Los árboles tienen una estructura jerárquica donde cada nodo puede tener cero o más nodos hijos. Esta estructura se define de forma recursiva, ya que cada nodo puede ser considerado como la raíz de un subárbol que incluye a sus propios hijos y subárboles.

13.2.2. **Nodo raíz:**

Es el nodo superior del árbol y actúa como punto de partida para acceder a todos los demás nodos. Este nodo no tiene un nodo padre.

13.2.3. **Nodos internos y hojas:**

Los nodos internos son aquellos que tienen al menos un hijo, mientras que las hojas son nodos que no tienen hijos. Los nodos internos se utilizan para estructurar y organizar los datos, mientras que las hojas contienen la información real o los valores almacenados.

13.2.4. **Grado de un nodo:**

Se refiere al número de hijos que tiene un nodo. El grado de un nodo puede ser cero (en el caso de las hojas), uno, dos o más.

13.2.5. **Profundidad y altura:**

La profundidad de un nodo en un árbol es la longitud del camino desde la raíz hasta ese nodo. La altura de un árbol es la longitud máxima de todos los caminos desde la raíz hasta las hojas. Estas métricas son útiles para medir la eficiencia de ciertas operaciones en un árbol, como la búsqueda.

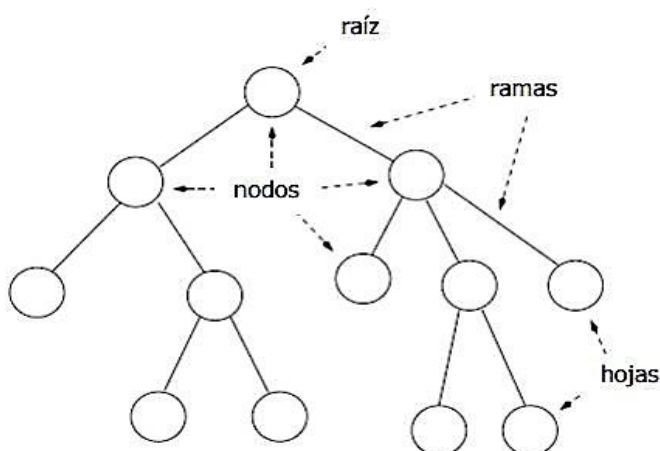


Imagen 13: Características del Árbol

13.3. Ejemplo:

NodoArbol.h

```
#ifndef NODOARBOL_H
#define NODOARBOL_H

class NodoArbol {
private:
    int dato;
    NodoArbol* izquierdo;
    NodoArbol* derecho;
public:
    NodoArbol(int valor);
    int obtenerDato();
    void establecerDato(int valor);
    NodoArbol* obtenerIzquierdo();
    void establecerIzquierdo(NodoArbol* nodo);
    NodoArbol* obtenerDerecho();
    void establecerDerecho(NodoArbol* nodo);
};

#endif
```

NodoArbol.cpp

```
#include "NodoArbol.h"

NodoArbol::NodoArbol(int valor) : dato(valor), izquierdo(nullptr),
derecho(nullptr) {}

int NodoArbol::obtenerDato() {
    return dato;
}

void NodoArbol::establecerDato(int valor) {
    dato = valor;
}

NodoArbol* NodoArbol::obtenerIzquierdo() {
    return izquierdo;
}

void NodoArbol::establecerIzquierdo(NodoArbol* nodo) {
```



```

        izquierdo = nodo;
    }

    NodoArbol* NodoArbol::obtenerDerecho() {
        return derecho;
    }

    void NodoArbol::establecerDerecho(NodoArbol* nodo) {
        derecho = nodo;
    }

```

ArbolBinario.h

```

#ifndef ARBOLBINARIO_H
#define ARBOLBINARIO_H
#include "NodoArbol.h"

class ArbolBinario {
private:
    NodoArbol* raiz;
    void destruirArbol(NodoArbol* nodo);

public:
    ArbolBinario();
    ~ArbolBinario();
    void insertar(int valor);
};

#endif

```

ArbolBinario.cpp

```

#include "ArbolBinario.h"

ArbolBinario::ArbolBinario() : raiz(nullptr) {}

ArbolBinario::~ArbolBinario() {
    destruirArbol(raiz);
}

```

```

void ArbolBinario::destruirArbol(NodoArbol* nodo) {
    if (nodo) {
        destruirArbol(nodo->obtenerIzquierdo());
        destruirArbol(nodo->obtenerDerecho());
        delete nodo;
    }
}

```

```

void ArbolBinario::insertar(int valor) {
}

```

main.cpp

```

#include <iostream>
#include "ArbolBinario.h"

```

```

int main() {
    ArbolBinario arbol;

    arbol.insertar(5);
    arbol.insertar(3);
    arbol.insertar(7);

    return 0;
}

```

13.4. Conclusiones

Los árboles son una estructura de datos fundamental en informática, caracterizada por su organización jerárquica donde cada nodo puede tener cero o más nodos hijos, excepto la raíz, que no tiene nodos padres. Su versatilidad y amplia aplicabilidad se reflejan en su uso en diversos campos, desde la representación de datos en bases de datos y sistemas de archivos hasta la implementación de algoritmos de búsqueda y ordenación.

14. CLASIFICACIÓN DE ÁRBOLES

14.1. Introducción

Los árboles son estructuras de datos fundamentales en informática que se utilizan para organizar y almacenar datos de manera jerárquica. En este documento, exploraremos diferentes tipos de árboles, incluidos los árboles binarios, AVL, B, B+, rojo-negro, Radix y Trie, analizando sus características, ventajas y aplicaciones.

14.2. Árbol Binario

Un árbol binario es una estructura de datos en la que cada nodo puede tener hasta dos hijos: un hijo izquierdo y un hijo derecho. Estos árboles son utilizados en una variedad de aplicaciones, incluidas las búsquedas y las representaciones de expresiones aritméticas.

14.3. Árbol AVL

Los árboles AVL son árboles binarios de búsqueda balanceados en los que la diferencia de altura entre los subárboles izquierdo y derecho de cualquier nodo es como máximo uno. Esta propiedad asegura que el árbol esté siempre balanceado y proporciona operaciones de búsqueda, inserción y eliminación eficientes en tiempo logarítmico.

14.4. Árboles B y B+

Los árboles B y B+ son estructuras de datos utilizadas principalmente en bases de datos y sistemas de archivos. Estos árboles están diseñados para manejar grandes cantidades de datos y admiten operaciones eficientes de búsqueda, inserción y eliminación, así como un acceso rápido a través de índices.

14.5. Árbol Rojo-Negro

Un árbol rojo-negro es un tipo de árbol binario de búsqueda balanceado en el que cada nodo tiene un atributo de color que puede ser rojo o negro. Estos árboles mantienen un equilibrio entre la altura de los subárboles izquierdo y derecho, lo que garantiza operaciones eficientes en tiempo logarítmico.

14.6. Árbol Radix

Un árbol radix es una estructura de datos especializada diseñada para almacenar y buscar datos cuyas claves son cadenas de caracteres o números. Estos árboles utilizan la posición de los dígitos o caracteres en las claves para organizar y buscar datos de manera eficiente.

14.7. Trie

Un trie, también conocido como árbol de prefijos, es una estructura de datos eficiente para almacenar y buscar conjuntos de cadenas de caracteres. Cada nodo en un trie representa un carácter, y las rutas desde la raíz hasta los nodos hoja representan las cadenas almacenadas en la estructura.

14.8. Conclusiones

La variedad de árboles disponibles en informática proporciona herramientas poderosas para organizar, almacenar y recuperar datos de manera eficiente en una amplia gama de aplicaciones. La elección del tipo de árbol adecuado depende de las características específicas del problema y los requisitos de rendimiento.

15. OPERACIONES BÁSICAS CON ÁRBOLES.

15.1. Introducción.

Como ya se ha visto en semanas anteriores un árbol es una estructura jerárquica que consta de nodos conectados entre sí mediante enlaces o aristas. Cada nodo en un árbol tiene un padre (excepto el nodo superior, llamado raíz) y cero o más nodos hijos. Esta puede realizar varias operaciones: Crear, Insertar, Recorrer o Balancear si estos son necesarios.

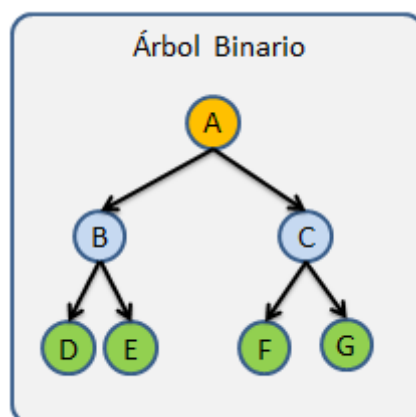


Imagen 15 Ciclo de vida del TDA

15.2. Creación, Inserción, Eliminación, Recorridos sistemáticos, Balanceo

• Creación

Para crear un árbol necesitamos insertar un nodo, llamado nodo raíz, es decir se hará lo siguiente:

- Se inserta un valor
- Este valor tomar la posición de la raíz y desde aquí partirá muchas de las operaciones del árbol.



Imagen 16 Creación de la Raíz de un Árbol Binario

- **Inserción**

Una vez creado nuestro árbol, vamos a insertar más nodos. Se basa en un árbol binario de búsqueda, es decir, el nodo contiene datos de tipo numéricos. La idea es no repetir elementos, sin embargo, se verificará primeramente si existe un nodo con el mismo dato que se quiere insertar. Si el caso es afirmativo, el nodo: será insertado por la derecha del que halló su coincidente, y así si encuentra más. Si no existen repetidos, el procedimiento menciona que si el número es mayor se inserta por la derecha, pero si es menor, se inserta por la izquierda. (Gonzales,2021)

a. Comienza en la Raíz.

- Comienza desde la raíz del árbol.



Imagen 17 Raíz Actual

b. Comparación del Valor:

- Compara el valor del nodo que deseas insertar con el valor del nodo actual.

0 < 0025. Looking at left subtree



*Imagen 18 Comparación Nodo Ingresado con Raíz actual***c. Decisión de Movimiento:**

- Si el valor del nodo a insertar es menor que el valor del nodo actual, muévete al subárbol izquierdo.
- Si el valor es mayor, muévete al subárbol derecho.
- Si el valor es igual se mueve al lado derecho

*Imagen 19 Decisión de movimiento de nodo actual***d. Repetición:**

- Repite los pasos 2 y 3 hasta que encuentres un lugar vacío donde puedas insertar el nuevo nodo.

e. Inserción:

- Inserta el nuevo nodo en el lugar vacío encontrado en el paso anterior.

*Imagen 20 Inserción*

- **Eliminación**

Este proceso se simplifica cuando el nodo a eliminar es terminal o tiene solo un descendiente. Sin embargo, surge dificultad al intentar borrar un nodo con dos descendientes, ya que un solo puntero no puede apuntar en dos direcciones. En este caso, el nodo a eliminar se reemplaza con el nodo más a la derecha de su subárbol izquierdo o el nodo más a la izquierda de su subárbol derecho. La elección del nodo de sustitución

asegura que la propiedad de árbol de búsqueda se conserve en el árbol resultante.

a. Encontrar el Nodo a Eliminar:

- Comienza buscando el nodo que deseas eliminar. Si el nodo no está presente en el árbol, entonces no hay nada que eliminar.

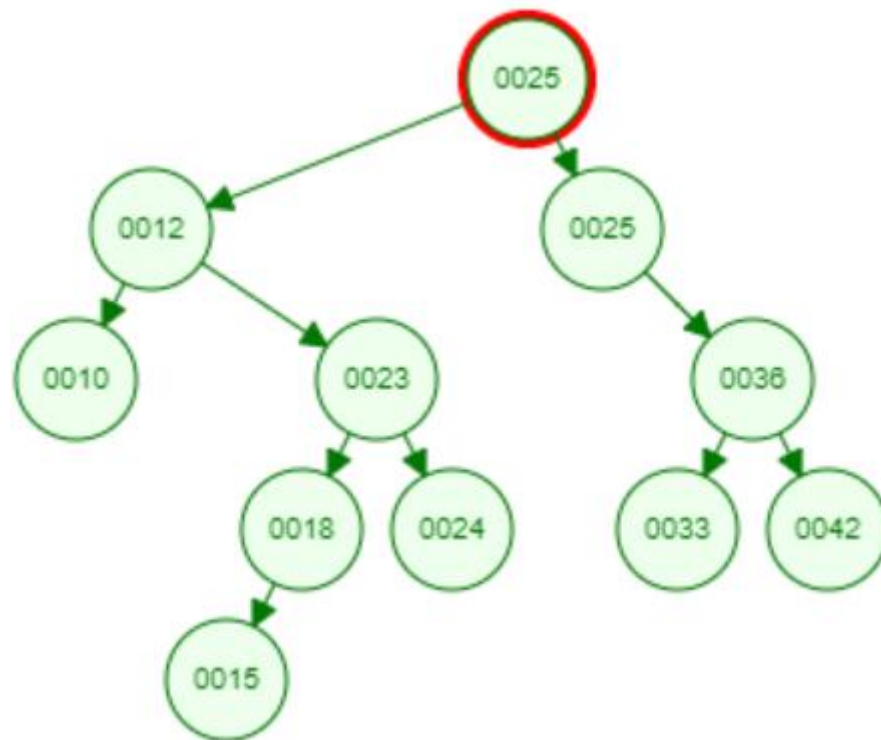


Imagen 21 Buscar el nodo a eliminar

b. Identificar el Caso del Nodo a Eliminar:

- Un nodo puede estar en una de las siguientes situaciones:
- **Caso 1: Nodo Hoja (sin hijos):**
 - Puedes eliminar el nodo directamente.
- **Caso 2: Nodo con un Solo Hijo:**
 - Puedes eliminar el nodo y conectar su hijo directamente con el padre del nodo a eliminar.
- **Caso 3: Nodo con Dos Hijos:**
 - Necesitas encontrar el sucesor inmediato (el nodo más pequeño en el subárbol derecho del nodo a eliminar) o el predecesor inmediato (el nodo más grande en el subárbol izquierdo del nodo a eliminar). Luego, reemplaza el nodo a eliminar con el sucesor o predecesor y elimina el sucesor o predecesor original.

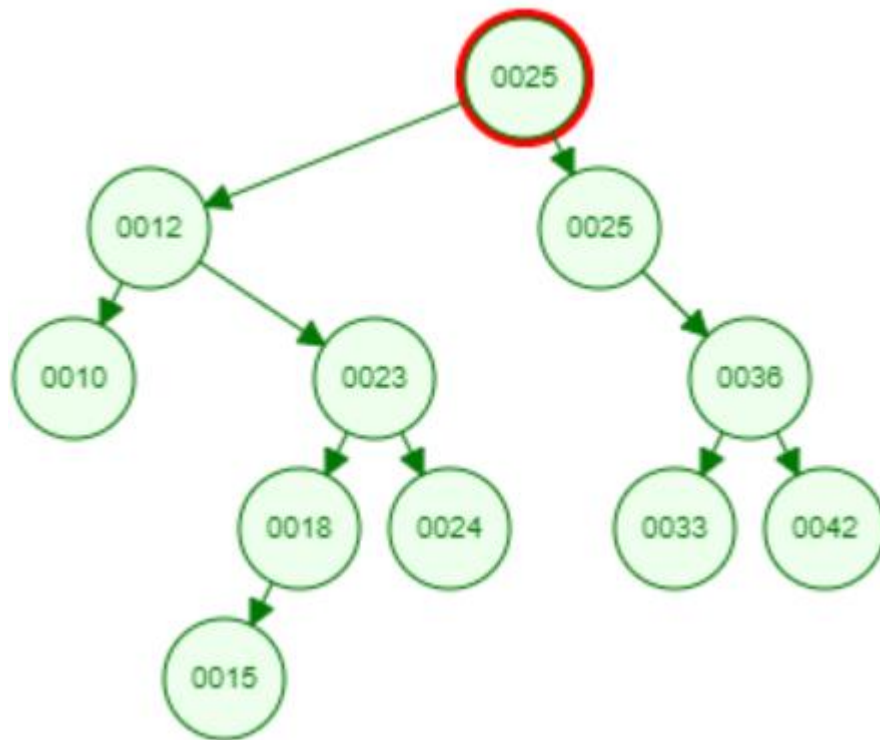


Imagen 22 Buscar el nodo a eliminar

- **Eliminar el Nodo:**
- Aplica las acciones correspondientes según el caso identificado en el paso anterior.
- Caso 1:
 - Simplemente elimina el nodo.
- Caso 2:
 - Conecta el hijo no nulo del nodo a eliminar directamente con el padre del nodo a eliminar.
- Caso 3:
 - Encuentra el sucesor o predecesor inmediato.
 - Reemplaza el valor del nodo a eliminar con el valor del sucesor o predecesor.
 - Elimina el sucesor o predecesor original (puedes aplicar los pasos de eliminación nuevamente para este nodo).

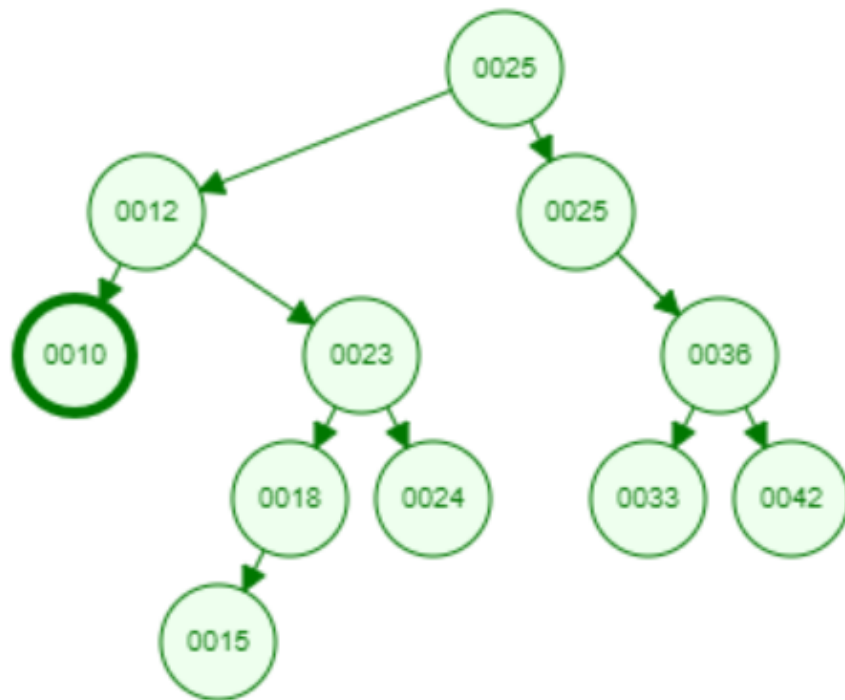


Imagen 23 Seleccionar el Nodo a eliminar

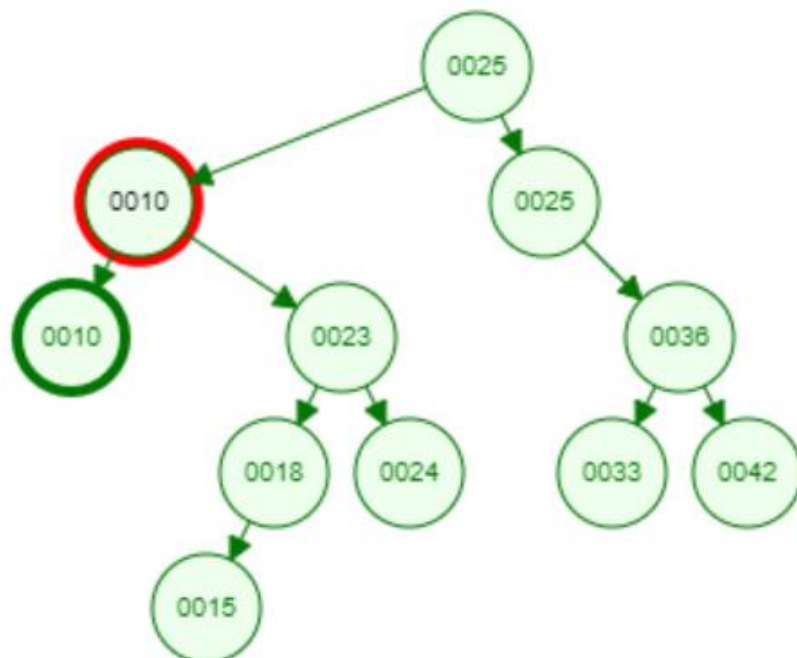


Imagen 24 Seleccionar el tipo de caso

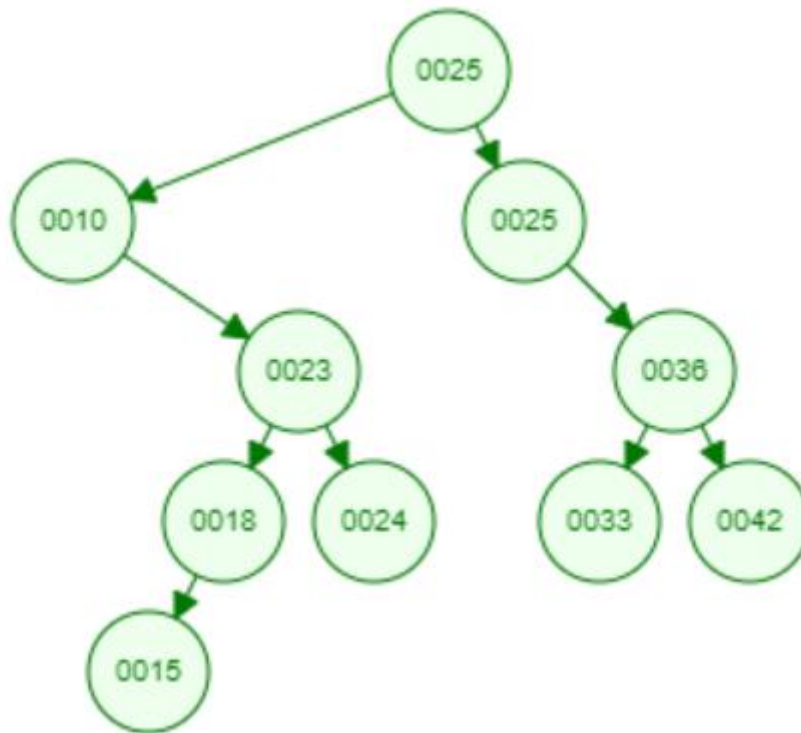


Imagen 25 Nuevo Árbol

- **Recorridos Sistemáticos.**

Existen varios métodos de recorrido de nodos en un árbol, y los tres principales son el recorrido en orden (in-order), el recorrido en preorden (pre-order) y el recorrido en postorden (post-order). A continuación, te proporciono los pasos para realizar cada uno de estos recorridos sistemáticos en un árbol:

Recorrido en Orden (In-Order):

- **Recorrer el Subárbol Izquierdo:**
 - Realiza un recorrido en orden del subárbol izquierdo del nodo actual.
- **Procesar el Nodo Actual:**
 - Procesa el nodo actual.
- **Recorrer el Subárbol Derecho:**
 - Realiza un recorrido en orden del subárbol derecho del nodo actual.
- **Recorrido en Preorden (Pre-Order):**
- **Procesar el Nodo Actual:**
 - Procesa el nodo actual.
- **Recorrer el Subárbol Izquierdo:**
 - Realiza un recorrido en preorden del subárbol izquierdo del nodo actual.
- **Recorrer el Subárbol Derecho:**

- Realiza un recorrido en preorden del subárbol derecho del nodo actual.
- **Recorrido en Postorden (Post-Order):**
- **Recorrer el Subárbol Izquierdo:**
 - Realiza un recorrido en postorden del subárbol izquierdo del nodo actual.
- **Recorrer el Subárbol Derecho:**
 - Realiza un recorrido en postorden del subárbol derecho del nodo actual.
- **Procesar el Nodo Actual:**
 - Procesa el nodo actual.
- **Balanceo.**

Un árbol binario está perfectamente equilibrado si, para todo nodo, el número de Y nodos en el subárbol izquierdo y el número de nodos en el subárbol derecho difieren como mucho en una unidad. Sin embargo, esto no es necesario realizar en todos los tipos de árboles, Principalmente en los árboles AVL y en los Rojo-Negro es obligatorio realizar estos balanceos.

Esto puede realizarse fácilmente distribuyendo los nodos, según se leen, equitativamente a la izquierda y a la derecha de cada nodo.



Imagen 26: Árbol Actual

16. GRAFOS

Los grafos son una estructura de datos fundamental en ciencia de la computación, con múltiples aplicaciones en el modelado de problemas complejos de software y hardware (Shaffer, 2022). En programación, los grafos

permiten representar relaciones y conexiones entre distintos objetos, usuarios, servidores, páginas web, funciones de código, etc. (Dai et al., 2021).

Por ejemplo, el grafo de llamadas entre funciones y métodos muestra cómo se interconectan las distintas partes de un programa (Shaffer, 2022). Asimismo, las estructuras de datos típicas como listas enlazadas y árboles son casos especiales de grafos. Y los algoritmos de grafos se aplican en optimización de código, análisis de rendimiento y testing (Han et al., 2022).

En base de datos los grafos representan redes de información como relaciones muchos-a-muchos. Y en aprendizaje automático se usan para patrones de interconexión en redes neuronales (Dai et al., 2021). En resumen, desde la programación de bajo nivel en ensamblador y compiladores, hasta el desarrollo de aplicaciones web y mobile, inteligencia artificial, big data y computación distribuida en la nube, los grafos son una abstracción indispensable para modelar problemas computacionales complejos (Shaffer, 2022).

Figura 1

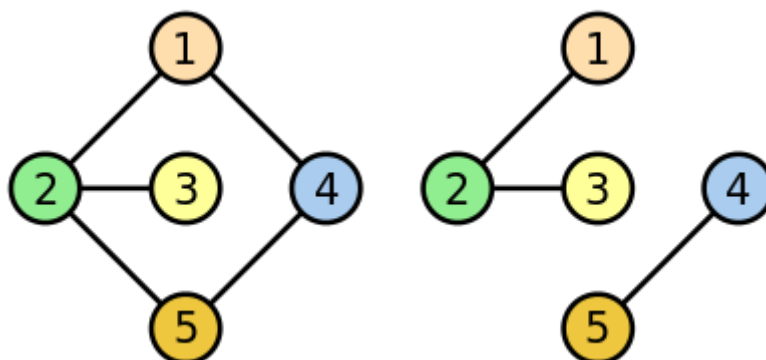


Figura 27: Representación de un grafo

16.1. Representación con grafos

16.1.1. Matrices de adyacencia

Es una matriz cuadrada donde el elemento i, j es 1 si existe una arista entre el vértice i y j , o 0 en caso contrario. Permite representar grafos dirigidos y valorados de manera sencilla (Cormen et al., 2022).

Figura 2

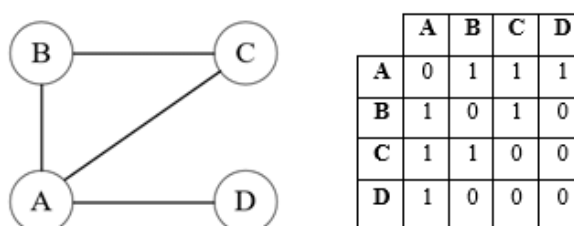


Figura 28: Matriz de adyacencia

16.1.2. Listas de adyacencia

Cada vértice tiene una lista vinculada con los vértices adyacentes. Es de uso eficiente en memoria y se adapta bien a grafos dispersos (con pocas aristas) (Aho et al., 2022).

Figura 3

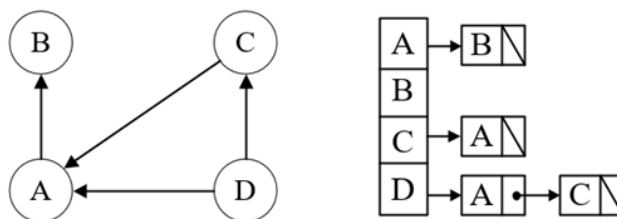


Figura29: Grafo con lista de adyacencia

16.2. Objetos y punteros

Consiste en implementar los grafos como objetos de nodos y aristas, con punteros que enlazan unos a otros. Es muy utilizado en programación orientada a objetos (Han et al., 2022).

16.3. Matriz de incidencia

Relaciona vértices con aristas, indicando las incidencias de cada arista sobre los vértices. Útil en varios algoritmos de grafos.

17. OPERACIONES BÁSICAS DE GRAFOS

17.1. Introducción

Los grafos son estructuras fundamentales en el ámbito de la teoría de grafos y son ampliamente utilizados en diversas áreas, desde la informática hasta la logística, pasando por la biología y la ingeniería. Un grafo es una representación abstracta de un conjunto de objetos interconectados, donde estos objetos se denominan nodos o vértices, y las conexiones entre ellos se llaman aristas.

El estudio de operaciones con grafos es esencial para comprender y resolver una amplia gama de problemas, desde encontrar la ruta más corta entre dos puntos en una red de transporte hasta modelar relaciones sociales en redes sociales. Este manual tiene como objetivo proporcionar una guía completa para entender y utilizar operaciones básicas y avanzadas en grafos.

17.2. Definición de las operaciones

Un grafo consta de un conjunto de objetos o elementos junto a sus relaciones. Es preciso definir las operaciones básicas para construir la estructura y, en general, modificar sus elementos. En definitiva, especificar el tipo abstracto de datos grafo.

Ahora se definen operaciones básicas, a partir de las cuales se construye el grafo. Su realización depende de la representación elegida (matriz de adyacencia, o listas de adyacencia).

Las operaciones con grafos son manipulaciones o transformaciones que se realizan sobre estructuras de datos de grafos. Un grafo es una estructura que consta de nodos (o vértices) y aristas (o bordes) que conectan estos nodos.

Las operaciones más básicas con grafos son la inserción de Nodos y Aristas, eliminación de nodos y aristas, recorridos, búsqueda e impresión, a continuación, se describen las operaciones básicas:

17.2.1. [Agregar un Nodo \(Vértice\)](#):

Esta operación implica agregar un nuevo nodo al grafo. Un nodo representa una entidad individual en el grafo, como una ubicación en un mapa o una persona en una red social. Al agregar un nuevo nodo, se debe actualizar la estructura de datos del grafo para incluir el nodo agregado.

17.2.2. **Eliminar un Nodo (Vértice):**

La eliminación de un nodo implica eliminar un nodo existente del grafo. Esto también implica eliminar todas las conexiones asociadas con ese nodo, es decir, eliminar todas las aristas que salen o entran en el nodo.

17.2.3. **Agregar una Arista:**

Una arista es una conexión entre dos nodos en el grafo. Agregar una arista implica establecer una conexión entre dos nodos existentes en el grafo. Esta operación puede ser dirigida o no dirigida, dependiendo de si la conexión tiene una dirección específica o no.

17.2.4. **Eliminar una Arista:**

La eliminación de una arista implica eliminar una conexión entre dos nodos existentes en el grafo. Al eliminar una arista, se rompe la conexión entre los nodos y ya no se consideran adyacentes entre sí.

Estas son algunas de las operaciones básicas que se realizan comúnmente en grafos. Cada una de estas operaciones tiene

aplicaciones prácticas en una variedad de campos, incluidas las redes sociales, la logística, la optimización de rutas, la planificación de proyectos y mucho más.

17.3. Codificación de las operaciones básicas

En esta sección, exploraremos los algoritmos y la codificación de estas operaciones básicas en grafos. Desde la implementación de estructuras de datos adecuadas hasta la escritura de algoritmos eficientes, nos sumergiremos en el mundo práctico de trabajar con grafos a través de ejemplos detallados y código explicativo.

GrafoNoDirigido.h

```
#ifndef GRAFONODIRIGIDO_H
#define GRAFONODIRIGIDO_H

#include <vector>
#include <stack>

class GrafoNoDirigido {
private:
    int numNodos;
    std::vector<std::vector<int>> adyacencia;

public:
    GrafoNoDirigido(int n);
    void agregarNodo();
    void eliminarNodo(int nodo);
    void agregarArista(int nodoA, int nodoB);
    void eliminarArista(int nodoA, int nodoB);
    void imprimirGrafo();
    void busquedaDFS(int nodoInicial);
};

#endif // GRAFONODIRIGIDO_H
```

GrafoNoDirigido.cpp

```
#include "GrafoNoDirigido.h"
```

```

#include <iostream>
#include <algorithm> // Necesario para std::find

GrafoNoDirigido::GrafoNoDirigido(int n) : numNodos(n), adyacencia(n,
std::vector<int>()) {}

void GrafoNoDirigido::agregarNodo() {
    ++numNodos;
    adyacencia.resize(numNodos);
}

void GrafoNoDirigido::eliminarNodo(int nodo) {
    nodo--;

    if (nodo >= 0 && nodo < numNodos) {
        // Creamos un nuevo vector de adyacencia sin las conexiones asociadas
        // al nodo que queremos eliminar
        std::vector<std::vector<int>> nuevoAdyacencia(numNodos - 1,
std::vector<int>());

        // Copiamos las conexiones del vector de adyacencia original al nuevo
        // vector, excluyendo las conexiones asociadas al nodo que queremos eliminar
        int nuevoIndice = 0;
        for (int i = 0; i < numNodos; ++i) {
            if (i != nodo) {
                for (int vecino : adyacencia[i]) {
                    if (vecino != nodo + 1) {
                        nuevoAdyacencia[nuevoIndice].push_back(vecino > nodo + 1 ?
vecino - 1 : vecino);
                    }
                }
                nuevoIndice++;
            }
        }

        // Actualizamos el vector de adyacencia con el nuevo vector creado
        adyacencia = nuevoAdyacencia;
        numNodos--;

        std::cout << "\nNodo " << nodo + 1 << " eliminado correctamente.\n" <<
std::endl;
    }
}

```



```

    } else {
        std::cerr << "\nError: El nodo a eliminar está fuera del rango del grafo.\n"
    }
}

void GrafoNoDirigido::agregarArista(int nodoA, int nodoB) {
    nodoA--;
    nodoB--;

    if (nodoA >= 0 && nodoA < numNodos && nodoB >= 0 && nodoB <
numNodos) {
        // Verificamos si los nodos existen en el grafo
        if (adyacencia.size() >= nodoA && adyacencia.size() >= nodoB) {
            adyacencia[nodoA].push_back(nodoB);
            adyacencia[nodoB].push_back(nodoA);
            std::cout << "\nArista entre " << nodoA + 1 << " y " << nodoB + 1 << "
agregada correctamente.\n" << std::endl;
        } else {
            std::cerr << "\nError: Al menos uno de los nodos ingresados no existe
en el grafo.\n" << std::endl;
        }
    } else {
        std::cerr << "\nError: Al menos uno de los nodos ingresados esta fuera del
rango del grafo. \n" << std::endl;
    }
}

void GrafoNoDirigido::eliminarArista(int nodoA, int nodoB) {
    nodoA--;
    nodoB--;

    if (nodoA >= 0 && nodoA < numNodos && nodoB >= 0 && nodoB <
numNodos) {
        // Verificamos si los nodos existen en el grafo
        if (adyacencia.size() >= nodoA && adyacencia.size() >= nodoB) {
            // Verificamos si hay una arista entre los nodos ingresados
            auto itA = std::find(adyacencia[nodoA].begin(),
adyacencia[nodoA].end(), nodoB);
            auto itB = std::find(adyacencia[nodoB].begin(),
adyacencia[nodoB].end(), nodoA);

```

```

        if (itA != adyacencia[nodoA].end() && itB != adyacencia[nodoB].end()) {
            adyacencia[nodoA].erase(itA);
            adyacencia[nodoB].erase(itB);
            std::cout << "\nArista entre " << nodoA + 1 << " y " << nodoB + 1 << "
eliminada correctamente.\n" << std::endl;
        } else {
            std::cerr << "\nError: No hay una arista entre los nodos
ingresados.\n" << std::endl;
        }
    } else {
        std::cerr << "\nError: Al menos uno de los nodos ingresados no existe
en el grafo.\n" << std::endl;
    }
} else {
    std::cerr << "\nError: Al menos uno de los nodos ingresados esta fuera del
rango del grafo.\n" << std::endl;
}
}

```

```

void GrafoNoDirigido::imprimirGrafo() {
    std::cout << "Lista de adyacencia del grafo:" << std::endl;
    for (int i = 0; i < numNodos; ++i) {
        std::cout << "Nodo " << i + 1 << " esta conectado con:";
        for (int vecino : adyacencia[i]) {
            std::cout << " " << vecino + 1;
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

```

```

void GrafoNoDirigido::busquedaDFS(int nodoInicial) {
    std::vector<bool> visitado(numNodos, false);
    std::stack<int> pila;

    pila.push(nodoInicial);

    std::cout << "Recorrido DFS desde el nodo " << nodoInicial << ": ";
}

```

```

while (!pila.empty()) {
    int nodoActual = pila.top();
    pila.pop();

    if (!visitado[nodoActual]) {
        std::cout << nodoActual << " ";
        visitado[nodoActual] = true;

        for (int vecino : adyacencia[nodoActual]) {
            if (!visitado[vecino]) {
                pila.push(vecino);
            }
        }
    }
}

std::cout << std::endl;
}

```

17.4. Conclusiones

Las operaciones básicas con grafos, como agregar y eliminar nodos y aristas, son fundamentales para manipular la estructura y la conectividad del grafo, afectando directamente su capacidad para representar relaciones entre entidades. Estas operaciones son la base sobre la cual se construyen algoritmos más complejos y aplicaciones prácticas en una variedad de campos, desde la optimización de rutas hasta el análisis de redes sociales. La eficiencia en la implementación de estas operaciones es crucial para garantizar un rendimiento óptimo en grafos de gran escala.

18. ALGORITMO DE DIJKSTRA

18.1. Introducción

El algoritmo de Dijkstra es un algoritmo clásico de búsqueda de caminos más cortos en un grafo ponderado dirigido o no dirigido con pesos no negativos. En este documento, exploraremos el algoritmo de Dijkstra, su funcionamiento y su implementación en C++.

18.2. Algoritmo de Dijkstra

El algoritmo de Dijkstra se ejecuta en un grafo ponderado dirigido o no dirigido y encuentra el camino más corto desde un nodo de inicio hacia todos los demás nodos en el grafo. Utiliza una estrategia de búsqueda de costo mínimo, explorando los nodos en orden de menor costo acumulado desde el nodo de inicio.

18.3. Funcionamiento del Algoritmo

El algoritmo de Dijkstra opera de la siguiente manera:

1. Se inicializa un conjunto de distancias acumuladas para cada nodo, estableciendo la distancia al nodo de inicio como 0 y el resto como infinito.
2. Se selecciona el nodo con la distancia acumulada mínima y se lo marca como visitado.
3. Se actualizan las distancias acumuladas de los nodos adyacentes al nodo seleccionado, si el camino a través del nodo seleccionado es más corto que el anteriormente registrado.
4. Se repiten los pasos 2 y 3 hasta que se hayan visitado todos los nodos o se haya alcanzado el nodo de destino.

18.4. Conclusiones

El algoritmo de Dijkstra es una herramienta poderosa para encontrar caminos más cortos en grafos ponderados. Su implementación eficiente permite su aplicación en una amplia gama de problemas de optimización de rutas y redes.

19. GRAFOS BIPARTIDOS

19.1. Introducción

Los grafos bipartidos son una estructura fundamental en teoría de grafos, ampliamente utilizada en diversas áreas como la informática, la biología, la economía y la sociología. Su importancia radica en su capacidad para modelar relaciones entre dos conjuntos de elementos de manera clara y eficiente.

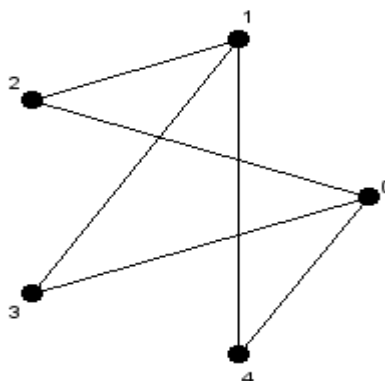


Imagen 30: Grafo Bipartido

19.2. Definición:

Un grafo bipartido es un tipo de grafo cuyos vértices pueden ser divididos en dos conjuntos disjuntos de manera que cada arista del grafo conecta un vértice de un conjunto con un vértice del otro conjunto. Formalmente, un grafo $G = (V, E)$ se considera bipartido si el conjunto de vértices V puede ser particionado en dos conjuntos disjuntos V_1 y V_2 , de tal manera que cada arista en E conecta un vértice en V_1 con un vértice en V_2 .

19.3. Características:

- **Bipartición clara:** La característica principal de los grafos bipartidos es que permiten una división clara y distinta de los vértices en dos conjuntos disjuntos, lo que facilita el análisis y la comprensión de las relaciones entre los elementos del grafo.
- **Estructura simple:** Debido a su naturaleza bipartita, los grafos bipartidos tienden a tener una estructura simple y regular, lo que los hace especialmente útiles en aplicaciones prácticas como la modelización de sistemas complejos.
- **Modelado de relaciones:** Son ideales para modelar relaciones entre dos conjuntos de entidades diferentes. Por ejemplo, en un grafo bipartido que modela la relación entre estudiantes y cursos, cada vértice de un conjunto representa un estudiante, mientras que cada vértice del otro conjunto representa un curso, y las aristas representan la inscripción de estudiantes en cursos.
- **Algoritmos eficientes:** Existen algoritmos eficientes para el análisis de grafos bipartidos, como el algoritmo de emparejamiento máximo, que encuentra el mayor conjunto de aristas disjuntas en un grafo bipartido, y

el algoritmo de coloreo de vértices, que asigna colores a los vértices de manera que vértices adyacentes tengan colores diferentes.

19.4. Ejemplo:

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

bool esBipartido(vector<vector<int>>& grafo, int src) {
    int V = grafo.size();
    vector<int> color(V, -1); // Colores de los vértices (-1: no
                             // coloreado, 0: color 1, 1: color 2)
    color[src] = 0; // Se colorea el vértice de origen con el color 0

    queue<int> q;
    q.push(src);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : grafo[u]) {
            if (color[v] == -1) { // Si el vértice no está coloreado
                color[v] = 1 - color[u]; // Se asigna un color
                // diferente al del vértice actual
                q.push(v);
            } else if (color[v] == color[u]) { // Si el vértice
                // adyacente tiene el mismo color que el actual
                return false; // No es bipartido
            }
        }
    }

    return true;
}

int main() {
    int V, E;
    cout << "Ingrese el número de vértices y aristas del grafo: ";
    cin >> V >> E;

    vector<vector<int>> grafo(V);

    cout << "Ingrese las aristas del grafo (por ejemplo, para la
    arista 0->1, ingrese '0 1'): \n";
    for (int i = 0; i < E; ++i) {
```

```

    int u, v;
    cin >> u >> v;
    grafo[u].push_back(v);
    grafo[v].push_back(u);
}

if (esBipartido(grafo, 0)) {
    cout << "El grafo es bipartido.\n";
} else {
    cout << "El grafo no es bipartido.\n";
}

return 0;
}

```

Este programa solicita al usuario que ingrese el número de vértices y aristas del grafo, así como las aristas del grafo. Luego, utiliza la función `esBipartido` para verificar si el grafo es bipartido o no, comenzando desde el vértice 0.

19.5. Conclusiones

En general, la teoría de grafos bipartidos proporciona un marco sólido y versátil para modelar relaciones entre dos conjuntos de elementos, con aplicaciones extendidas en una variedad de campos. La capacidad de dividir los vértices en dos conjuntos disjuntos y la estructura clara que ofrecen hacen que los grafos bipartidos sean fundamentales en problemas de optimización, diseño de algoritmos y análisis de redes complejas. Además, algoritmos eficientes como el de búsqueda en anchura (BFS) permiten verificar fácilmente si un grafo es bipartido, lo que contribuye a su utilidad práctica en la resolución de problemas del mundo real. En resumen, la teoría de grafos bipartidos ofrece un marco conceptual poderoso y aplicable en una amplia gama de contextos, facilitando el análisis y la comprensión de relaciones entre conjuntos de elementos.

20. REFERENCIAS

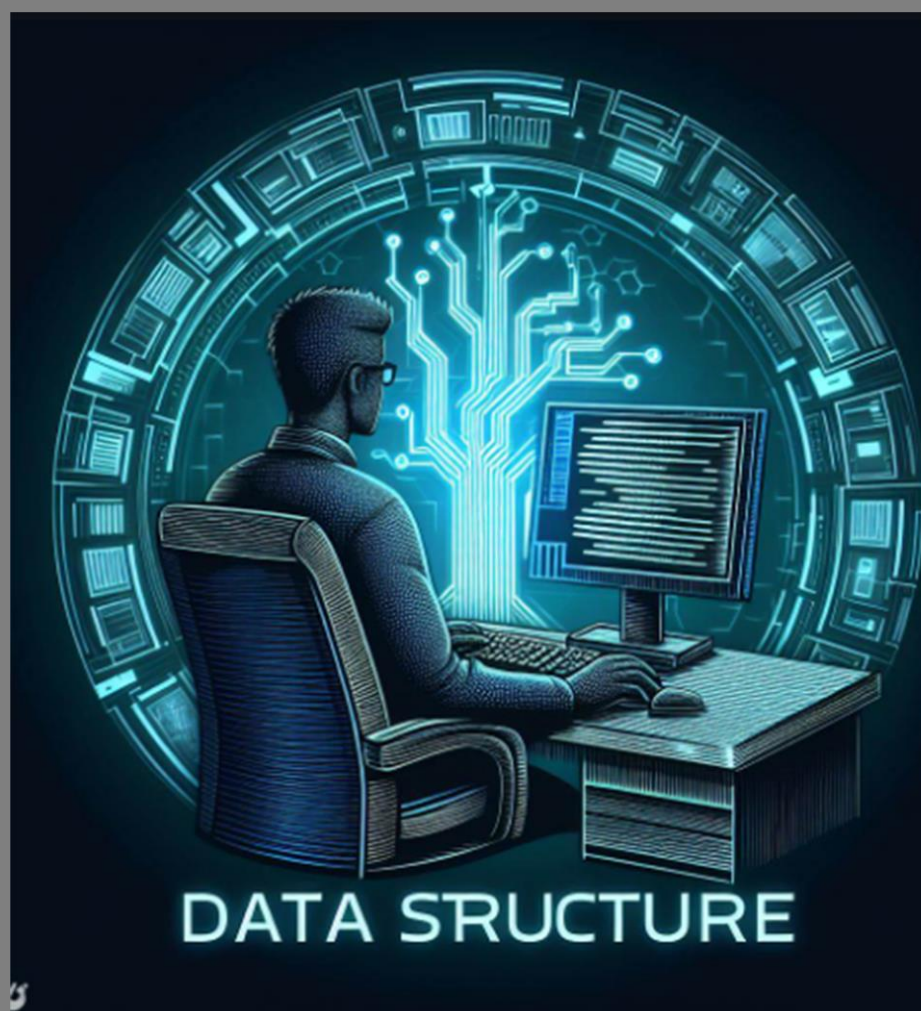
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.
- <https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf>
- Drozdek, Adam. "Data Structures and Algorithms in C++." 5th Edition, Cengage Learning, 2019.
- https://itlectures.ro/wpcontent/uploads/2016/04/AdamDrozdek__DataStructures_and_Algorithms_in_C_4Ed.pdf

- Jain, S. (2023, January 10). *ShellSort*. GeeksforGeeks. Retrieved February 4, 2024, from <https://www.geeksforgeeks.org/shellsort/>
- Edelkamp, S., Weiß, A., & Wild, S. (2020). QuickXsort: A fast sorting scheme in theory and practice. *Algorithmica*, 82(3), 509-588.
- Hossain, M. S., Mondal, S., Ali, R. S., & Hasan, M. (2020, March). Optimizing complexity of quick sort. In *International conference on computing science, communication and security* (pp. 329-339). Singapore: Springer Singapore.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). The MIT Press.
- Himadri, M. (2016) Data structures with C - by Schaum Series, Academia.edu. Available at: https://www.academia.edu/27906978/Data_Structures_With_C_by_schaum_series (Accessed: 10 January 2024).
- Estructura de datos en C++, Luis Joyanes e Ignacio Zahonero.
- Data Structures and Algorithms in Java, Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser.
- Sahni, Sartaj, and Sanguthevar Rajasekaran. "Data Structures, Algorithms, and Applications in Java." 2nd Edition, Wiley, 2018.
- Lafore, Robert. "Data Structures and Algorithms in Java." 2nd Edition, Sams Publishing, 2019.
- Drozdek, Adam. "Data Structures and Algorithms in C++." 5th Edition, Cengage Learning, 2019.
- Shaffer, C. A. (2022). A practical introduction to data structures and algorithm analysis. (4th ed.). Prentice Hall.
- Han, W., Miao, Q., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakar, R., Cheng, W., & Yu, W. (2022). Chronos: A graph engine for temporal graphs. ACM SIGMOD Conference 2022, 2335-2349
- Drozdek, Adam. "Data Structures and Algorithms in C++." 5th Edition, Cengage Learning, 2019.
https://itlectures.ro/wpcontent/uploads/2016/04/AdamDrozdek__DataStructures_and_Algorithms_in_C_4Ed.pdf

2024

UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE

MANUAL DE ESTRUCTURAS DE DATOS TEORÍA DE ALGORITMOS



Búsqueda Exhaustiva

1. Definición de la Búsqueda Exhaustiva

La búsqueda exhaustiva, también conocida como búsqueda completa o fuerza bruta, es una técnica algorítmica utilizada para encontrar soluciones óptimas o satisfactorias a un problema examinando sistemáticamente todas las posibles soluciones en un espacio de búsqueda. Aunque puede ser ineficiente en términos de tiempo de ejecución, la búsqueda exhaustiva es útil cuando el espacio de búsqueda es pequeño o cuando no hay una estructura lógica clara que permita optimizar la búsqueda.

En C++, la búsqueda exhaustiva se implementa típicamente utilizando bucles anidados o recursión, dependiendo de la naturaleza del problema.

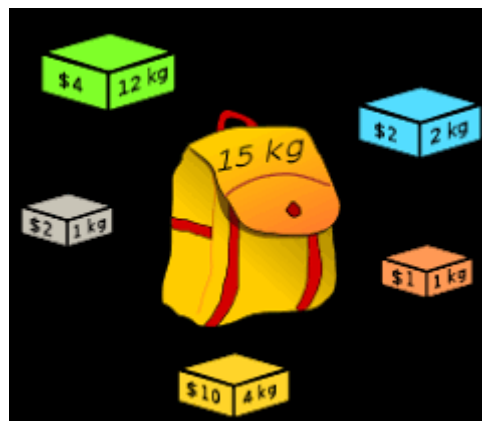


Imagen 1: Problema del Viajero resuelto con búsqueda exhaustiva

2. Consideraciones:

Las colas de prioridad tienen una amplia variedad de aplicaciones en el desarrollo de software. Algunos ejemplos destacados incluyen:

1. Eficiencia: La eficiencia de la búsqueda exhaustiva puede verse afectada significativamente por el tamaño del espacio de búsqueda y la complejidad del problema. Para problemas con un espacio de búsqueda grande, la búsqueda exhaustiva puede volverse prohibitivamente lenta y poco práctica.

2. Optimización: En algunos casos, puede ser necesario explorar otras técnicas de optimización, como la programación dinámica o el uso de algoritmos heurísticos, para mejorar el rendimiento y encontrar soluciones más rápidas y eficientes.
3. Limitaciones: La búsqueda exhaustiva puede no ser la mejor opción para todos los problemas, especialmente aquellos con un espacio de búsqueda muy grande o donde existen estructuras lógicas que permiten optimizaciones significativas.

2.1 Ejemplo:

A continuación, presentamos un ejemplo sencillo de implementación de búsqueda exhaustiva en c++ realizando sumas.

```
#include <iostream>

// Función para calcular la suma de los números en un rango dado
int sumaEnRango(int inicio, int fin) {
    int suma = 0;
    for (int i = inicio; i <= fin; ++i) {
        suma += i;
    }
    return suma;
}

int main() {
    int inicio, fin;
    std::cout << "Ingrese el inicio del rango: ";
    std::cin >> inicio;
    std::cout << "Ingrese el final del rango: ";
    std::cin >> fin;

    int resultado = sumaEnRango(inicio, fin);
    std::cout << "La suma de los números en el rango dado es: " <<
    resultado << std::endl;

    return 0;
}
```

Tenemos otro ejemplo más detallado de Búsqueda Exhaustiva en la carpeta de Código Búsqueda Exhaustiva en el Git Hub.

3. Conclusiones

La búsqueda exhaustiva es una técnica simple pero poderosa para abordar problemas algorítmicos en C++. Aunque puede ser ineficiente en algunos casos, sigue siendo útil para resolver problemas con un espacio de búsqueda pequeño o cuando otras técnicas de optimización no son aplicables. Es importante considerar las limitaciones y explorar otras estrategias cuando sea necesario para lograr soluciones más eficientes.

4. Recomendaciones

- Evalúa la complejidad del problema: Antes de usar búsqueda exhaustiva, considera el tamaño del espacio de búsqueda y la complejidad del problema. Si el espacio de búsqueda es grande, otras técnicas pueden ser más eficientes.
- Explora técnicas de optimización: Si el problema es complejo, considera otras técnicas como programación dinámica o algoritmos heurísticos para mejorar el rendimiento.
- Usa búsqueda exhaustiva con moderación: Evalúa si hay alternativas más eficientes antes de recurrir a búsqueda exhaustiva.
- Prueba exhaustivamente tu implementación: Asegúrate de probar tu código en diversos casos para verificar su corrección y rendimiento.
- Considera la paralelización: Para mejorar la eficiencia, piensa en paralelizar la búsqueda exhaustiva usando bibliotecas como OpenMP o MPI si es posible.

- **Documenta tu código:** Documenta tu implementación para facilitar la comprensión y el mantenimiento del código.

Algoritmos Voraces

1. Introducción

Un algoritmo voraz, también conocido como algoritmo greedy, es una estrategia para resolver problemas complejos mediante la toma de decisiones localmente óptimas en cada paso. Aunque no siempre garantizan una solución globalmente óptima, estos algoritmos son populares por su simplicidad, eficiencia y capacidad para encontrar soluciones satisfactorias en muchos casos.

2. Funcionamiento

Los algoritmos voraces se basan en un ciclo iterativo que comprende tres etapas:

- **Evaluación:** En cada iteración, se evalúan las opciones disponibles en ese momento.
- **Selección:** Se elige la opción que parece ser la mejor según la estrategia voraz definida.
- **Actualización:** Se actualiza el estado del problema para reflejar la decisión tomada.

El ciclo se repite hasta que se alcanza un estado final que satisface las condiciones de terminación.

3. Estrategias comunes

Existen diferentes estrategias voraces que se pueden utilizar en la selección de la mejor opción en cada iteración. Algunas de las más comunes son:

- **El mejor ajuste:** Se elige la opción que mejor se ajusta a un criterio específico, como el menor costo, la mayor ganancia o la menor distancia.
- **El primero en llegar, el primero en ser atendido:** Se elige la opción que ha estado disponible durante más tiempo.
- **El más cercano:** Se elige la opción que está más cerca de un punto de referencia específico.

4. *Análisis de Algoritmos Voraces*

El análisis de un algoritmo voraz implica evaluar su eficiencia y su capacidad para encontrar soluciones óptimas.

- **Eficiencia:** La eficiencia de un algoritmo voraz se mide por su tiempo de ejecución y su complejidad espacial. En general, los algoritmos voraces son bastante eficientes y pueden ejecutarse en tiempo polinomial.
- **Optimalidad:** No todos los algoritmos voraces encuentran soluciones globalmente óptimas. La optimalidad de un algoritmo voraz depende de la naturaleza del problema y de la estrategia voraz utilizada.

5. *Aplicaciones de Algoritmos Voraces*

Los algoritmos voraces se utilizan en una amplia variedad de aplicaciones, incluyendo:

- **Ruteo:** Encontrar la ruta más corta entre dos puntos en un mapa.
- **Asignación:** Asignar tareas a recursos de manera eficiente.
- **Programación:** Seleccionar el mejor conjunto de tareas para ejecutar en un procesador.
- **Optimización:** Encontrar la mejor solución a un problema de optimización, como el problema del vendedor ambulante.

6. *Ejemplos de Algoritmos Voraces*

Algoritmo de Dijkstra: Es la ruta más corta entre un vértice y todos los otros vértices de un grafo.

Divide y Vencerás.

1. *Definición de Divide y Vencerás.*

Divide y vencerás es un paradigma algorítmico (a veces llamado por error Divide y Concurrir - una adaptación en broma), similar a los paradigmas de programación Dinámica y Algoritmos ávidos o glotones. Un algoritmo Divide y Vencerás típico resuelve un problema siguiendo estos 3 pasos.

1. **Dividir:** Descomponer el problema en sub-problemas del mismo tipo.
Este paso involucra descomponer el problema original en pequeños sub-problemas. Cada sub-problema debe representar una parte del problema original. Por lo general, este paso emplea un enfoque

recursivo para dividir el problema hasta que no es posible crear un sub-problema más.

2. **Vencer:** Resolver los sub-problemas recursivamente. Este paso recibe un gran conjunto de sub-problemas a ser resueltos. Generalmente a este nivel, los problemas se resuelven por sí solos.
3. **Combinar:** Combinar las respuestas apropiadamente. Cuando los sub-problemas son resueltos, esta fase los combina recursivamente hasta que estos formulan la solución al problema original. Este enfoque algorítmico trabaja recursivamente y los pasos de conquista y fusión trabajan tan a la par que parece un sólo paso.

Usualmente este método nos permite hacer una reducción bastante significativa en la complejidad tiempo del algoritmo a emplear.

2. Técnica de Divide y Vencerás.

- Se divide el problema en subproblemas. Por eficiencia debemos intentar que los subproblemas tengan un tamaño similar. Si es posible resolver el problema, porque este es indivisible, se resuelve directamente.

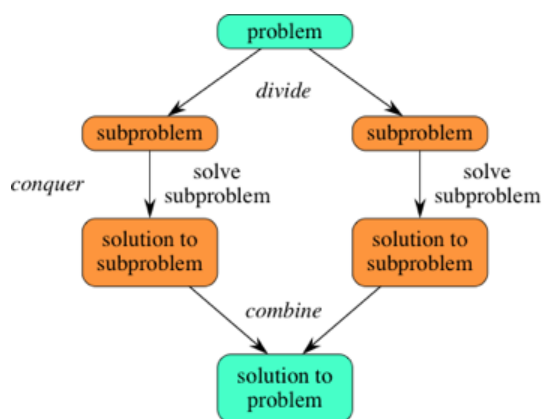


Imagen 1: Técnica Divide y vencerás.

- Se resuelve el problema para cada una de las instancias más pequeñas.
- Se combinan las soluciones obtenidas para obtener una solución global.

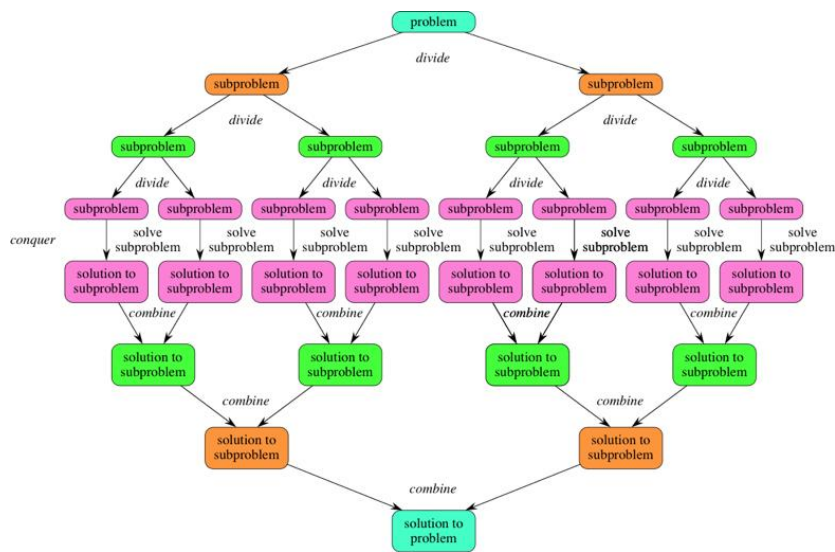


Imagen 2: Combinan las soluciones obtenidas para obtener una solución global

3. Ejemplo:

En este ejemplo, la función encontrarMaximo divide la matriz en mitades más pequeñas, encuentra el máximo en cada mitad y luego combina los resultados para obtener el máximo global.

Divide Y Venceras.cpp

```

#include <iostream>
#include <vector>

// Función para encontrar el máximo elemento en una matriz
int encontrarMaximo(const std::vector<int>& arr, int izquierda,
int derecha) {
    if (izquierda == derecha) {
        return arr.at(izquierda);
    }
    else {
        int mitad = (izquierda + derecha) / 2;
        int maxIzquierda = encontrarMaximo(arr, izquierda, mitad);
        int maxDerecha = encontrarMaximo(arr, mitad + 1, derecha);
        return std::max(maxIzquierda, maxDerecha);
    }
}

int main() {
    std::vector<int> numeros = { 10, 5, 8, 15, 3, 20, 12 };
    int maximoElemento = encontrarMaximo(numeros, 0,
numeros.size() - 1);
    std::cout << "El maximo elemento es: " << maximoElemento <<
std::endl;
    return 0;
}

```

4. Conclusiones.

- La técnica de "Divide y Vencerás" es efectiva para abordar problemas complejos al descomponerlos en subproblemas más simples.
- La fase de combinación es crucial para obtener la solución global de manera correcta y eficiente.
- El manejo adecuado de la recursividad es fundamental para evitar problemas como el desbordamiento de la pila de llamadas.

5. Recomendaciones.

- La técnica de "Divide y Vencerás" es efectiva para abordar problemas complejos al descomponerlos en subproblemas más simples.
- La fase de combinación es crucial para obtener la solución global de manera correcta y eficiente.
- El manejo adecuado de la recursividad es fundamental para evitar problemas como el desbordamiento de la pila de llamadas.

Programación Dinámica

Que es la programación dinámica

La programación dinámica es una técnica esencial de optimización utilizada para abordar problemas combinatorios y de toma de decisiones. Su importancia radica en su capacidad para descomponer problemas complejos en subproblemas más manejables y resolverlos de forma independiente. Esto no solo hace posible resolver problemas que de otro modo serían difíciles o incluso imposibles de abordar, sino que también permite una solución eficiente para problemas complejos como el recorrido del viajante, el problema de la mochila y la búsqueda de la subsecuencia común más larga. entre otros.

Características

- Almacenamiento de resultados previos: La programación dinámica almacena los resultados de los subproblemas previamente resueltos para evitar cálculos redundantes.
- Divide y vencerás: La técnica se basa en el análisis del problema en subproblemas más pequeños y manejables.

- Soluciones óptimas: La programación dinámica proporciona soluciones óptimas a los problemas, garantizando la mejor solución posible dentro del conjunto de todas las posibles soluciones.
- Problemas solapados: Los subproblemas en la programación dinámica se solapan, lo que significa que los mismos subproblemas se resuelven varias veces. Esto permite una optimización al evitar recalcular soluciones ya conocidas.

Ventajas de la programación dinámica:

- Eficiencia: La programación dinámica puede ofrecer soluciones eficientes para problemas complejos, reduciendo significativamente el tiempo de ejecución en comparación con otros enfoques.
- Optimalidad: Proporciona soluciones óptimas garantizadas para problemas de optimización, lo que significa que la solución encontrada es la mejor posible dentro del conjunto de todas las soluciones posibles.
- Reutilización de resultados: Almacenando y reutilizando los resultados de subproblemas previamente resueltos, se evita el cálculo redundante, lo que mejora aún más la eficiencia del algoritmo.
- Aplicabilidad: La técnica de programación dinámica es aplicable a una amplia gama de problemas combinatorios y de toma de decisiones, lo que la hace versátil y útil en diversas situaciones.

Desventajas de la programación dinámica:

Complejidad de implementación: La implementación de algoritmos de programación dinámica puede ser complicada y requerir un entendimiento profundo del problema y de la técnica en sí misma. Esto puede hacer que sea difícil para los programadores novatos.

Requisitos de memoria: Al almacenar los resultados de subproblemas previamente resueltos, la programación dinámica puede requerir una cantidad significativa de memoria, especialmente para problemas con muchos subproblemas.

Dificultad para identificar subproblemas: Identificar correctamente los

subproblemas y definir la recurrencia puede ser desafiante en algunos casos, lo que puede dificultar la aplicación efectiva de la programación dinámica.

No siempre es la mejor opción: Aunque la programación dinámica puede proporcionar soluciones eficientes y óptimas para muchos problemas, no es la mejor opción para todos los problemas. En algunos casos, otros enfoques pueden ser más adecuados o más fáciles de implementar.

Problema de la mochila

El problema de la mochila es un desafío clásico en optimización combinatoria que implica seleccionar la combinación óptima de objetos para maximizar el valor total, sin exceder la capacidad de una mochila dada. La técnica de programación dinámica es una forma eficaz de resolver este problema al descomponerlo en subproblemas más pequeños y reutilizar las soluciones parciales. La relación de recurrencia, que define cómo calcular el valor máximo posible para diferentes capacidades de mochila y diferentes conjuntos de objetos, se utiliza para construir una tabla de programación dinámica. Esta tabla almacena las soluciones parciales y se actualiza gradualmente durante la resolución del problema, lo que permite encontrar la solución óptima de manera eficiente.

Código

Mochila.cpp

```
#include <iostream>
#include <vector>
#include "validaciones.h"

using namespace std;

// Estructura de un objeto
struct Objeto {
    int peso;
    int valor;
};

// Función para calcular el máximo entre dos números
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Función que resuelve el problema de la mochila utilizando
programación dinámica
```

```

int problemaMochila(int capacidad, vector<int>& pesos, vector<int>&
valores, int n) {
    vector<vector<int>> V(n + 1, vector<int>(capacidad + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= capacidad; j++) {
            if (pesos[i - 1] <= j) {
                V[i][j] = max(V[i - 1][j], valores[i - 1] + V[i - 1][j]
- pesos[i - 1]);
            } else {
                V[i][j] = V[i - 1][j];
            }
        }
    }

    return V[n][capacidad];
}

int main() {
    system("cls");
    validaciones v;

    int n;
    // Validación para el número de objetos
    do {
        n = v.ingresar_enteros("\nIngrese el numero de objetos (menos
de 25): ");
        if (n <= 0 || n > 25) {
            cout << "\nPor favor, ingrese un numero valido de
objetos." << endl;
        }
    } while (n <= 0 || n > 25);

    vector<int> pesos(n);
    vector<int> valores(n);

    // Ingreso de los pesos y valores de los objetos
    for (int i = 0; i < n; ++i) {
        cout << "\nObjeto " << i + 1 << endl;
        // Validación para el peso del objeto
        do {
            pesos[i] = v.ingresar_enteros("\nIngrese el peso del
objeto (mayor que 0 y menor que 100): ");
            if (pesos[i] <= 0 || pesos[i] >= 100) {
                cout << "\nPor favor, ingrese un peso mayor que 0 y
menor que 100" << endl;
            }
        } while (pesos[i] <= 0);

        // Validación para el valor del objeto
        do {
            valores[i] = v.ingresar_enteros("\nIngrese el valor del
objeto (mayor que 0): ");
            if (valores[i] <= 0) {
                cout << "\nPor favor, ingrese un valor mayor que 0."
<< endl;
            }
        } while (valores[i] <= 0);
    }
}

```

```

    }

    int capacidadMaxima;
    capacidadMaxima = v.ingresar_enteros("\nIngrese la capacidad
maxima de la mochila: ");

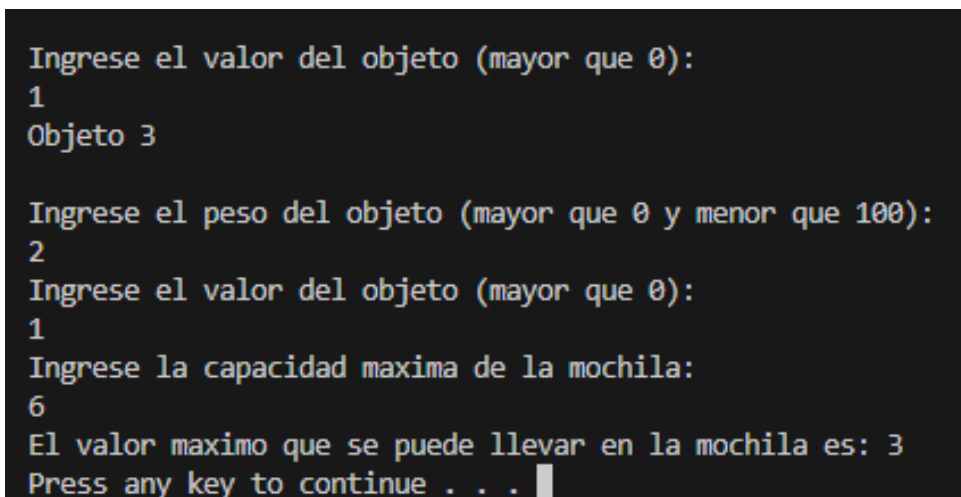
    // Resolución del problema de la mochila
    int maxValor = problemaMochila(capacidadMaxima, pesos, valores,
n);

    cout << "\nEl valor maximo que se puede llevar en la mochila es: "
<< maxValor << endl;
    system("pause");

    return 0;
}

```

a. **Ejecución del código:**



```

Ingrese el valor del objeto (mayor que 0):
1
Objeto 3

Ingrese el peso del objeto (mayor que 0 y menor que 100):
2
Ingrese el valor del objeto (mayor que 0):
1
Ingrese la capacidad maxima de la mochila:
6
El valor maximo que se puede llevar en la mochila es: 3
Press any key to continue . . .

```

Figura 1. Ejecución de código de Mochila

Backtracking

1. Introducción

El backtracking es una técnica algorítmica utilizada para encontrar soluciones a problemas computacionales, especialmente en situaciones donde se requiere buscar todas las posibles soluciones dentro de un espacio de búsqueda. Se basa en la idea de construir sistemáticamente todas las posibles configuraciones, descartando aquellas que no cumplen ciertas condiciones o restricciones, y retrocediendo (backtracking) cuando se llega a un punto donde ya no es posible seguir adelante sin violar alguna restricción.

2. Funcionamiento

El algoritmo de retroceso normalmente funciona de la siguiente

manera:

- Elija una solución inicial y colóquela en la raíz del árbol de búsqueda.
- Examine la siguiente opción, seleccione una y agréguela a la solución actual.
- Si la solución actual es válida, pase a la siguiente opción y repita el segundo paso
- Si la solución actual no es válida, retroceda hasta el último punto de decisión y pruebe una opción diferente.

Continúe el proceso hasta que se encuentre una solución o se hayan examinado todas las soluciones posibles.

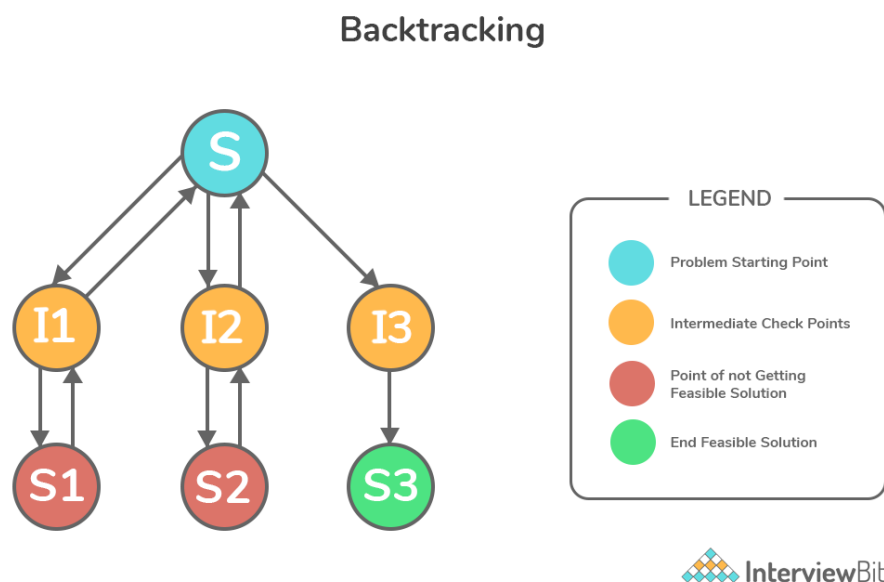


Figura 1: Funcionamiento del Backtracking

3. Ventajas

- Permite encontrar todas las soluciones posibles a un problema.
- Es una técnica de resolución de problemas general que se puede aplicar a muchos tipos de problemas diferentes.
- El backtracking es una técnica fácil de entender y de implementar.
- El backtracking puede ser muy efectivo cuando se aplica correctamente.

4. Desventajas

- Puede ser ineficiente para problemas con un gran número de soluciones posibles.
- La complejidad temporal del backtracking puede ser muy alta en algunos casos, lo que hace que el algoritmo sea impráctico o incluso imposible de aplicar.
- Puede requerir una gran cantidad de memoria para almacenar el árbol de decisiones completo.
- No garantiza encontrar la solución óptima a un problema, a menos que se utilicen técnicas de poda y optimización adicionales.

5. Análisis

Complejidad temporal:

- La complejidad temporal del algoritmo de backtracking depende en gran medida del tamaño del espacio de búsqueda y de la cantidad de restricciones y condiciones que se aplican a cada paso.
- En el peor de los casos, donde se exploran todas las posibles combinaciones, la complejidad temporal puede ser exponencial, $O(b^d)$, donde 'b' es el factor de ramificación (número de opciones disponibles en cada paso) y 'd' es la profundidad del árbol de decisiones (número de pasos).
- Sin embargo, con técnicas de poda y optimización adecuadas, la complejidad temporal puede reducirse significativamente.

Complejidad espacial:

- La complejidad espacial del algoritmo de backtracking también depende del tamaño del espacio de búsqueda y de la cantidad de memoria necesaria para almacenar las soluciones parciales y las restricciones.
- En el peor de los casos, la complejidad espacial puede ser lineal, $O(d)$, donde 'd' es la profundidad del árbol de decisiones y representa la cantidad máxima de memoria necesaria para almacenar las soluciones parciales en un momento dado.

6. Aplicaciones

- Resolución de problemas de búsqueda exhaustiva: El backtracking se utiliza para encontrar todas las posibles soluciones a problemas donde se necesita explorar todo el espacio de búsqueda, como la generación de permutaciones, combinaciones, subconjuntos, etc.
- Problemas de optimización: En muchos problemas de optimización, el backtracking se puede utilizar para encontrar la mejor solución posible explorando exhaustivamente todas las posibles configuraciones y eligiendo la que optimice ciertas métricas o criterios.
- Problemas de satisfacción de restricciones (CSP): El backtracking se utiliza ampliamente en problemas de satisfacción de restricciones, donde se busca encontrar una solución que cumpla con un conjunto de restricciones o condiciones, como el problema de asignación de horarios, el problema de coloración de grafos, etc.
- Juegos y rompecabezas: El backtracking se utiliza en la resolución de juegos y rompecabezas, como el sudoku, el juego del 8, el ajedrez, etc., donde se busca encontrar una secuencia de movimientos que conduzca a una solución o victoria.
- Compiladores y análisis de código: En la compilación y el análisis de código, el backtracking se utiliza en la resolución de problemas de análisis sintáctico y semántico, como la generación de árboles de sintaxis abstracta, la resolución de conflictos de reducción y desplazamiento en el análisis LR, etc.

2. Ejemplo – N Reinas

Tablero.h

```
#pragma once

class Tablero {
public:
    Tablero();
    Tablero(int);
    void set_TAM(int);
    int get_TAM();
    void set_piezas_strings(std::string***);
    std::string*** get_piezas_strings();
    void mostrar_tablero();
    bool esta_ocupado(int, int);
```



```

        ~Tablero();
private:
    int TAM;
    std::string*** piezas_strings;
};

                                Tablero.cpp

#include <iostream>

#include "Tablero.h"
#include <iomanip>

Tablero::Tablero() {
    TAM = 0;
    piezas_strings = nullptr;
}

Tablero::Tablero(int _TAM) {

    //ENCERO MI MATRIZ DE PUNTEROS A strings CON DIRECCIONES NULAS

    TAM = _TAM;

    piezas_strings = (std::string***)malloc(TAM *
sizeof(std::string**));
    if (piezas_strings != nullptr) {
        for (int i = 0; i < TAM; i++) {
            *(piezas_strings + i) = (std::string**)malloc(TAM *
sizeof(std::string*));
            for (int i = 0; i < TAM; i++) {
                if (*(piezas_strings + i) != nullptr) {
                    for (int j = 0; j < TAM; j++) {
                        (*(piezas_strings + i) + j) = new std::string("-
");
                    }
                }
            }
        }
    }

}

void Tablero::set_TAM(int _TAM) {
    TAM = _TAM;
}

int Tablero::get_TAM() {
    return TAM;
}

void Tablero::set_piezas_strings(std::string*** _piezas_strings) {
    piezas_strings = _piezas_strings;
}

```

```

std::string*** Tablero::get_piezas_strings() {
    return piezas_strings;
}

bool Tablero::esta_ocupado(int _x, int _y) {
    if (*(*(piezas_strings + _x) + _y) != nullptr) {
        return true;
    }
    return false;
}

void Tablero::mostrar_tablero() {

    if (piezas_strings != nullptr) {
        for (int i = 0; i < TAM; i++) {
            if (*(piezas_strings + i) != nullptr) {
                for (int j = 0; j < TAM; j++) {
                    std::cout << std::setw(10) << (*(*(piezas_strings
+ i) + j)) << std::setw(10);
                }
                printf("\n");
            }
        }
    }

}

Tablero::~Tablero() {
}

```

Problema_Reinas.h

```

#pragma once

#include "Tablero.h"

bool validar_posiciones(int, int*, int, int*, int);
void fun(Tablero& tablero, int x, int* _x, int, int*, int&);

bool validar_posiciones(int x, int* _x, int y, int* _y, int TAM) {
    for (int i = 0; i < TAM; i++) {
        if (y == *(_y + i) || std::abs((x - *(_x + i)) ==
std::abs(y - *(_y + i)))) {
            return false;
        }
    }
    return true;
}

void fun(Tablero& tablero, int x, int* _x, int y, int* _y, int&
contador) {

    if (x != tablero.get_TAM() && y != tablero.get_TAM()) {

        if (validar_posiciones(x, _x, y, _y,
tablero.get_TAM())) {

```

```

        *((*(tablero.get_piezas_strings() + x) + y)) =
"Reina";
        *(_x + x) = x;
        *(_y + x) = y;
        fun(tablero, x + 1, _x, 0, _y, contador);
        if ((x+1 != tablero.get_TAM() && *(_y +
tablero.get_TAM() - 1) == 10)) {
            *((*(tablero.get_piezas_strings() + x)
+ y)) = "-";
        }
        else {
            contador++;
        }
    }

    fun(tablero, x, _x, y + 1, _y, contador);

    if (*(_y + tablero.get_TAM()-1) != 10) {

        tablero.mostrar_tablero();
        printf("SOL: %i, -----
-----\n", contador);
    }

    *((*(tablero.get_piezas_strings() + x) + y)) = "-";
    *(_y + x) = 10;
    *(_x + x) = 20;
}

}

```

N_Reinas.cpp

```

#include <iostream>
#include <conio.h>
#include "windows.h"

#include "Menu.h"
#include "Tablero.h"
#include "validaciones.h"
#include "Problema_Reinas.h"

int main()
{
    int opcion = 1;
    char tecla;
    int contador = 0;

    int TAM;
    Tablero* ajedrez = nullptr;

    int* _x = nullptr;
    int* _y = nullptr;

    HANDLE consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);

```

[illegible]

```

        *(_x + j) = 20;
    }
    fun(*ajedrez, 0, _x, 0, _y, contador);
}
}

delete ajedrez;
delete _x;
delete _y;
contador = 0;

cursorInfo.bVisible = false;
SetConsoleCursorInfo(consoleHandle, &cursorInfo);

system("pause");

}
else if (opcion == 2) {

    system("cls");
    std::cout << "Saliendo del programa." << std::endl;
    exit(0);

}
break;
default:
break;

}

} while (true);

}

```

Eficiencia de los Algoritmos

1. Introducción

La eficiencia de los algoritmos es un aspecto crucial en el diseño y análisis de algoritmos. Se refiere a la medida en que un algoritmo utiliza recursos, como tiempo y espacio, para resolver un problema dado. En el estudio de la eficiencia de los algoritmos, nos preocupamos por comprender cómo varía el tiempo de ejecución o el uso de recursos a medida que cambian las entradas del problema.

Lo más importante de un algoritmo es que sea fácil de entender, y por tanto, de mantener en una situación normal. Pero si presenta problemas de eficiencia, ya tendremos que valorar otras opciones que, aunque puedan restar legibilidad, puede que mejoren en velocidad y en uso de otros recursos. Tenemos que prestar especial atención a los bucles, por ejemplo, si hay

operaciones dentro de un bucle que son idénticas en todas las operaciones y que se podrían hacer una sola vez fuera del bucle. Anidar bucles también es una práctica de riesgo, pues nos acerca a complejidades exponenciales. Evita calcular la longitud del elemento que recorres en cada iteración; si no va a variar, calcúlalo antes de entrar en el bucle, guárdalo en una variable y úsala para controlar el recorrido.

2. Definición de Eficiencia de Algoritmos

Cuando se enfrenta un problema, es común encontrar múltiples algoritmos adecuados para su resolución, y la elección del mejor es fundamental. ¿Cómo decidir entre diferentes opciones algorítmicas para abordar un mismo problema?

En situaciones donde se debe resolver un número limitado de casos de un problema simple, la elección del algoritmo puede no ser crítica. En estos casos, la preferencia puede inclinarse hacia la rapidez de implementación o la disponibilidad de algoritmos preexistentes, sin necesidad de considerar sus propiedades teóricas. Sin embargo, cuando se enfrentan numerosos casos o problemas de mayor complejidad, la selección del algoritmo requiere un enfoque más meticuloso y fundamentado (Walter B, 2020).

Existen dos enfoques principales para la selección de algoritmos:

1. Enfoque empírico (o a posteriori): Implica la implementación de diferentes técnicas y su evaluación mediante pruebas en la computadora, utilizando una variedad de casos de prueba.
2. Enfoque teórico (o a priori): Consiste en determinar, mediante análisis matemático, los recursos necesarios para cada algoritmo en función del tamaño de los casos considerados. Los recursos de interés principal suelen ser el tiempo de ejecución (el más crítico) y el espacio de almacenamiento, principalmente la memoria principal de la computadora.

Se compararán los algoritmos principalmente desde el enfoque teórico, basándose en sus tiempos de ejecución. La eficiencia de un algoritmo se evaluará principalmente en función de su velocidad de ejecución, considerando como más eficiente aquellos que ejecuten las tareas en menos tiempo.

3. Clasificación

Dado que no hay una unidad de medida estándar para expresar la eficiencia de un algoritmo, se suele evaluar en términos del tiempo requerido por el algoritmo para ejecutarse. En este sentido, se puede afirmar que un algoritmo para resolver un problema necesita un tiempo aproximado de $t(n)$, donde t es una función dada, si existe una constante positiva c y una implementación del algoritmo que pueda resolver todos los casos de tamaño n en un tiempo no mayor a $ct(n)$ unidades de tiempo. Es importante destacar que

la unidad de tiempo es arbitraria y puede ser definida según el contexto, ya sea años, minutos, días, horas, segundos, milisegundos, etc.

Esto mismo se puede definir formalmente:

$T(n) = O(f(n))$ si existe una constante c y un valor n_0 tales que $T(n) \leq c \cdot f(n)$ cuando $n > n_0$

Algunos órdenes de complejidad son tan comunes que tienen nombres propios, como se detalla a continuación. En la figura 1 se muestran las funciones de crecimiento para valores de tamaño del uno al diez. Aunque este rango de valores es limitado, es adecuado para visualizar las diferencias entre los distintos órdenes de complejidad y cómo cambia el tiempo de ejecución en función del tamaño de la entrada:

1. en el orden de $O(c)$, o de tiempo constante;
2. en el orden de $O(\log n)$, o de tiempo logarítmico;
3. en el orden de $O(n)$, o de tiempo lineal;
4. en el orden de $O(n \log n)$, o de tiempo casi lineal;
5. en el orden de $O(n^2)$, o de tiempo cuadrático;
6. en el orden de $O(n^3)$, o de tiempo cúbico;
7. en el orden de $O(n^k)$, o de tiempo polinómico;
8. en el orden de $O(c^n)$, o de tiempo exponencial;
9. en el orden de $O(n!)$, o de tiempo factorial.

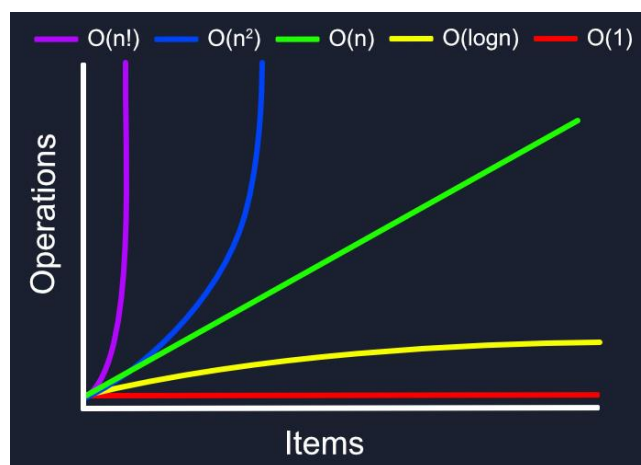


Imagen 1 Eficiencia basado en la complejidad de las operaciones (undefinedworld.com)

4. Enfoque de Análisis de Algoritmos

Cuando se analiza un algoritmo se lo puede realizar desde tres enfoques útiles:

1. Mejor caso, poco útil dada la poca o escasa ocurrencia de dicho caso.
2. Caso medio, útil cuando los casos a resolver son muy volátiles e igualmente probables que ocurran (es decir equiprobables). Requiere de mucha información a priori respecto de los casos a resolver. Esto en general es difícil de conocer.
3. Peor caso, es adecuado para problemas críticos donde se debe conocer el tiempo requerido para el peor caso. No requiere información respecto de los casos a resolver.

Es necesario abordar dos conceptos fundamentales:

- **Operación elemental:** Se refiere a una instrucción básica en la computación, como una suma o una asignación, que se puede realizar en un tiempo constante, independientemente de la implementación específica. Por simplicidad, se considera que estas operaciones tienen un costo unitario.
- **Notación asintótica:** Esta notación se centra en el comportamiento de las funciones para valores muy grandes de sus parámetros. Permite comparar algoritmos incluso para tamaños moderados o grandes de datos. En esta notación, "n" representa el tamaño del conjunto de datos y "t(n)" indica la cantidad de recursos necesarios para implementar un algoritmo en función de ese tamaño.

5. Anexo

Ejemplo 1

```
#include <iostream>
void imprimir_pares(int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            std::cout << "(" << i << ", " << j << ") ";
        }
        std::cout << std::endl;
    }
}
int main() {
    int numero;
    std::cout << "Ingrese un número: ";
    std::cin >> numero;
    imprimir_pares(numero);
    return 0;
}
```

Ejemplo 2

```
if num > 10:
    print('es mayor que 10')
    print('su cuadrado es 100')
    aux = num % 2
else:
    print('es menor que 10')
    print('su cubo es 1000')
    aux = num // 2
    print(aux)
    producto = num * 2
```


Ejemplo 3

```

if num > 10:
    print('es mayor que 10')
    print('su cuadrado es 100')
    aux = num % 2

if(num%20):
    print('es divisible por 2')
else:
    print('es divisible por 2')
    num = num + 1
    aux = num * num * num
else:
    print('es menor que 10')
    print('su cubo es 1000')
    aux= num // 2
    print(aux)
    producto = num * 2
    if(producto % 3 == 0):
        print('es divisible por 3')

```

6. Guía de ejercicios prácticos

A continuación, se plantean una serie de fragmentos de códigos. Habrá que realizar un análisis de la complejidad de cada una de las diferentes funciones.

Ejercicio 1

```

numero= int(input('ingrese un número'))
while (numero!=0) and (len(lista) < 10000):
    lista.append(numero)
    numero = int(input('ingrese número'))

for i in range(0, len(lista)):
    print(lista[i])

```

Ejercicio 2

```

while (p ≤ u) and (pos == -1):
    med = (p+u) // 2
    if(lista[med] < x):
        pos = med
else:
    if(x > lista[med]):

```

```

        p = med + 1
else:
    p = med 1

```

Tiempo de ejecución

1. Concepto

El tiempo de ejecución se refiere a la cantidad de tiempo que tarda un algoritmo en completar su tarea para una entrada de datos dada. Es una métrica fundamental para evaluar la eficiencia de un algoritmo y determinar su escalabilidad a medida que aumenta el tamaño de los datos de entrada.

2. Notación

Existen diferentes notaciones para expresar el tiempo de ejecución de un algoritmo, siendo la notación asintótica la más común. La notación asintótica se enfoca en el comportamiento del algoritmo cuando el tamaño de la entrada crece indefinidamente, ignorando factores constantes y términos menos significativos. Las notaciones más utilizadas son:

- **Notación O grande (O):** Representa el peor caso o el límite superior del tiempo de ejecución del algoritmo.
- **Notación Omega (Ω):** Representa el mejor caso o el límite inferior del tiempo de ejecución del algoritmo.
- **Notación Theta (Θ):** Representa el caso promedio o el orden exacto del tiempo de ejecución del algoritmo.

3. Ejemplo

Supongamos que tenemos un algoritmo de búsqueda lineal en un arreglo de n elementos. En el mejor caso, el elemento buscado se encuentra en la primera posición, por lo que el tiempo de ejecución es $\Omega(1)$. En el peor caso, el elemento buscado se encuentra en la última posición o no está presente, por lo que el tiempo de ejecución es $O(n)$. En el caso promedio, el tiempo de ejecución es $\Theta(n)$.

4. Aporte matemático

1.1.1. Complejidad temporal

La complejidad temporal se refiere al tiempo de ejecución de un algoritmo en función del tamaño de la entrada. Proporciona una medida cuantitativa de la eficiencia del algoritmo.

El análisis de la complejidad temporal se realiza considerando tres casos principales: el mejor caso, el caso promedio y el peor caso. Esto permite comprender el comportamiento del algoritmo en diferentes escenarios.

Se utilizan herramientas matemáticas, como el álgebra y el cálculo, para determinar la complejidad temporal de un algoritmo. Esto implica analizar el número de operaciones realizadas en función del tamaño de la entrada y expresar la complejidad mediante notaciones asintóticas.

1.1.2. Algoritmos

Los algoritmos pueden ser representados de manera formal utilizando notaciones matemáticas, como pseudocódigo, diagramas de flujo o descripciones matemáticas precisas.

Se utilizan técnicas de demostración matemática, como la inducción, para probar la corrección de un algoritmo, asegurando que produzca resultados correctos para todas las entradas válidas. Además, se demuestra la terminación del algoritmo, garantizando que finalice en un tiempo finito.

Muchos algoritmos recursivos pueden ser analizados mediante relaciones de recurrencia, que describen la relación entre el costo de una instancia del problema y el costo de sus subproblemas. Se utilizan métodos matemáticos, como el método de sustitución, el método de iteración y el método del árbol de recursión, para resolver estas relaciones de recurrencia.

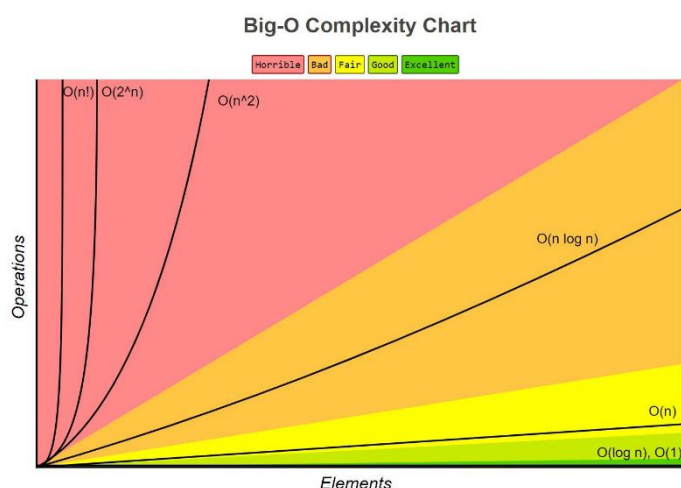
1.1.3. Notación Big O

La notación Big O se utiliza para describir el comportamiento asintótico de la complejidad temporal de un algoritmo a medida que el tamaño de la entrada crece. Proporciona una cota superior en el peor caso.

Existen reglas y propiedades matemáticas que rigen el cálculo y la manipulación de la notación Big O, como la aditividad, la multiplicación y la transitividad.

La notación Big O permite comparar la eficiencia de diferentes algoritmos que resuelven el mismo problema, al observar cómo se comporta su complejidad temporal a medida que aumenta el tamaño de la entrada.

Figura 1: Tabla de notación Big O



5. Ejemplo en código

Main.cpp

```
int main() {
    Validaciones validador;
    ListaSimple lista;

    int cantidad;
    cout << "    **** TIEMPO DE EJECUCION DE UN ALGORITMO ****\n" << endl;
    cout << "    ** Ordenamiento Burbuja ** \n" << endl;
    cout << "- Ingresa la cantidad de elementos en la lista: ";
    cantidad = validador.validarEnteros();
    cout << "\n";

    if (cantidad <= 0) {
        cout << "La cantidad de elementos debe ser mayor que 0." << endl;
        return 1;
    }

    cout << ">> Ingresa " << cantidad << " numeros:" << endl;
    for (int i = 0; i < cantidad; i++) {
        int valor;
        valor = validador.validarEnteros();
        cout << "\n";
        lista.agregarElemento(valor);
    }

    auto inicio = std::chrono::high_resolution_clock::now();

    cout << "\nLista original: ";
    lista.mostrarLista();

    lista.ordenarBurbuja();

    auto fin = std::chrono::high_resolution_clock::now();

    cout << "Lista ordenada: ";
    lista.mostrarLista();

    std::chrono::duration<double> tiempo = fin - inicio;
    std::cout <<

    "\n\n=====
    ==\n\n";
    std::cout << "- TIEMPO DE EJECUCION DEL ALGORITMO: " << std::endl;
    std::cout << " " << tiempo.count() << " segundos\n" << std::endl;
    std::cout <<

    "=====
    =\n\n";

    return 0;
}
```

Ejecución

Figura 2

```
- Ingresa la cantidad de elementos en la lista: 2
>> Ingresa 2 numeros:
3
4
```

```
Lista original: 4 3
Lista ordenada: 3 4
```

```
=====

- TIEMPO DE EJECUCION DEL ALGORITMO:
0.0001374 segundos

=====
```

Figura 2: Ejecución de código

6. Recomendaciones

Analizar cuidadosamente el tiempo de ejecución de los algoritmos propuestos utilizando la notación asintótica y compararlo con soluciones alternativas.

Priorizar algoritmos con mejores complejidades de tiempo, especialmente aquellos que son $O(\log n)$ o $O(n \log n)$, ya que tienden a ser más escalables que algoritmos cuadráticos o exponenciales.

Optimizar el código implementado para minimizar operaciones innecesarias y aprovechar técnicas como memorización, división y conquista, o programación dinámica cuando sea posible.

7. Conclusiones

El análisis del tiempo de ejecución es fundamental para evaluar la eficiencia de un algoritmo. Nos permite comprender cómo se comporta un algoritmo a medida que crece el tamaño de la entrada de datos, lo que es crucial para determinar su escalabilidad y viabilidad en situaciones del mundo real.

La notación big-O, en particular, es ampliamente utilizada para expresar el tiempo de ejecución en el peor caso, lo que nos brinda una cota superior del rendimiento del algoritmo. Esto es crucial para garantizar que un algoritmo pueda manejar las entradas más desafiantes dentro de un tiempo razonable.

Al comparar el tiempo de ejecución de diferentes algoritmos, podemos identificar aquellos que son más eficientes y escalables, lo que nos permite tomar decisiones informadas al seleccionar el enfoque más apropiado para un problema dado.

Notación Asintótica

Para entender el concepto de la notación asintótica debemos comprender dos conceptos básicos. Primero, el tiempo que se tarda un algoritmo dependiendo del tamaño de su entrada. Para esto tomamos en cuenta que el número máximo de intentos en las búsquedas lineales y las búsquedas binarias aumentan a medida que la longitud del arreglo aumenta. Siguiendo esta idea podemos concluir que el tiempo de ejecución del algoritmo es una función del tamaño de su entrada.

Segundo debemos saber en qué tan rápido crece una función por su tamaño de entrada, esto se llama tasa de crecimiento del tiempo de ejecución. Para ser mas eficaces con este concepto, tenemos que simplificar la función y extraer la parte mas importante. Tomemos de ejemplo un algoritmo que corre con una entrada de tamaño n , se tarda $n^2 + 10n + 500$ instrucciones de maquina es ser completado. El termino n^2 , sin importar el valor de n , va a crecer a un ritmo mas acelerado que los términos $10n + 500$.

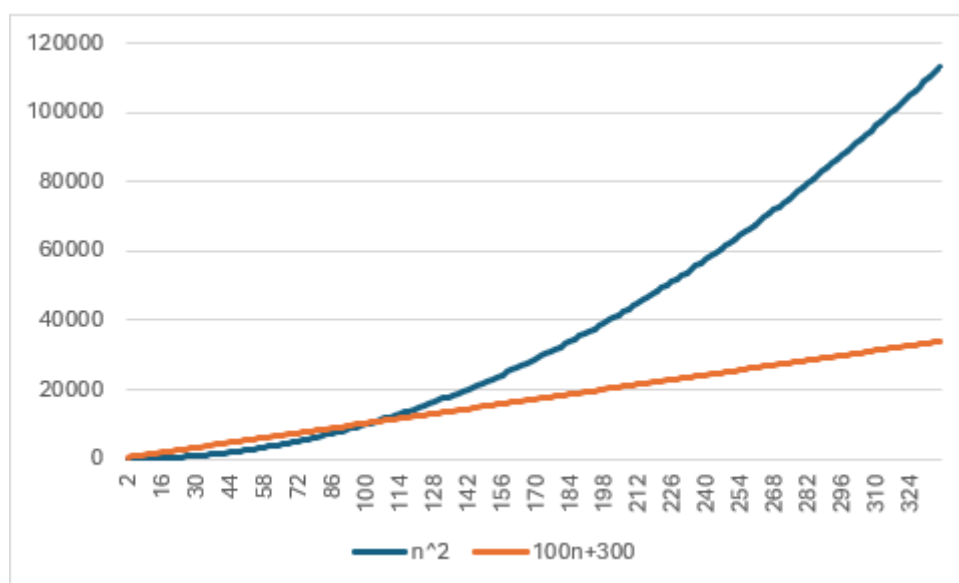


Ilustración 2: Gráfico tiempo de ejecución

Se puede concluir que el tiempo de ejecución crece como n^2 , descartando el

coeficiente y los términos restantes, siempre que el tiempo de ejecución sea $an^2 + bn + c$ para $a > 0$, b y c siempre habrá un valor de n para que an^2 sea mayor al resto de términos.

Al descartar estos términos no significativos y con coeficientes constantes, nos enfocamos en la tasa de crecimiento de un algoritmo, en este momento utilizamos la notación asintótica. Tiene tres formas, la notación Θ grande (Big Θ), notación O grande (Big O) y notación Ω grande (Big Ω).

1. Notación Θ grande (Big Θ):

Tenemos como ejemplo el código:

```
void busqueda_lineal(std::string funcion, char valor) {
    for (int intentos = 0; intentos < funcion.length(); guess++) {
        if (funcion[intentos] == valor) {
            return intentos; //valor encontrado y su posición
        }
    }
    return -1; // no se encontro el valor
};
```

En este código el tamaño del string sería n el cual es el número máximo de veces que se puede ejecutar el bucle for.

Cada que el bucle for itera, tiene que hacer varios cálculos:

- Compara "intentos" con el tamaño del string.
- Compara el valor de la posición del string con el valor.
- Regresar el valor de intentos (si es que encuentra el valor).
- Incrementar el valor de intentos.

Cada calculo toma un tiempo constante y si el bucle itera n veces, entonces el tiempo para todas las iteraciones sería $c_1 * n$ donde c_1 es la suma de los tiempos para el cálculo en una iteración del bucle. c_1 no tiene un valor fijo dado que depende de muchos factores. c_2 es el tiempo que el bucle for y también es una constante. Por tanto, el tiempo de búsqueda lineal, en el caso de que no se encuentre el valor, es de $c_1 * n + c_2$.

Para valores menores de n , no es importante el tiempo de ejecución. Pero cuando n es lo suficientemente grande utilizamos Theta de n , o $\Theta(n)$, y se concluye que el tiempo de ejecución estará en el intervalo inferior de $k_1 * n$ y $k_2 * n$.

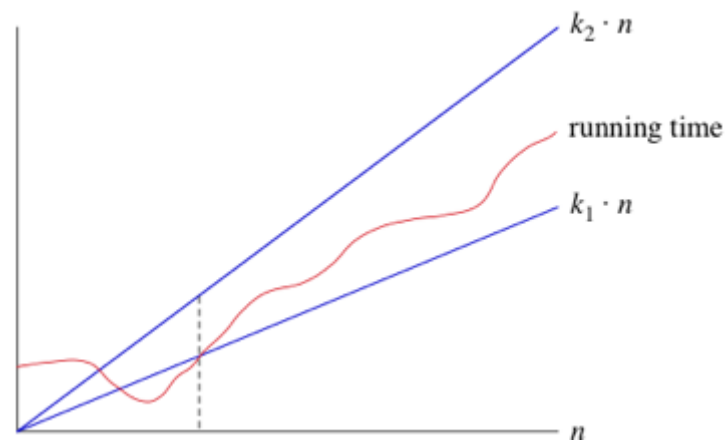


Ilustración 3: Tasa de crecimiento Big Theta.

En el cálculo de Theta, no importan las unidades de tiempo que usemos, al utilizar la notación Θ grande, decimos que tenemos una *cota asintóticamente ajustada* sobre el tiempo de ejecución. Asintóticamente porque solo importan los valores grandes de n , Cota ajustada porque ajustamos el tiempo de ejecución dentro del rango de una constante mayor y menor.

Las funciones en notación asintótica se clasifican de mayor a menor dependiendo de qué tan rápido crecen. Es así como la primera crece a un ritmo lento y la última a un ritmo bastante acelerado:

1. $\Theta(1)$
2. $\Theta(\log_2 n)$
3. $\Theta(n)$
4. $\Theta(n \log_2 n)$
5. $\Theta(n^2)$
6. $\Theta(n^2 \log_2 n)$
7. $\Theta(n^3)$
8. $\Theta(2^n)$
9. $\Theta(n!)$

2. Notación O grande (Big O):

Utilizamos Big O para acotar asintóticamente el crecimiento de factores dentro de constantes por arriba y por abajo. Aunque también se pueden acotar solo por arriba.

Por ejemplo, el peor tiempo de ejecución de una búsqueda binaria es $\Theta(\log_2 n)$,

aunque no siempre se ejecuta en ese tiempo. Si se encuentra el valor en la primera iteración entonces es válido decir que se ejecutó en un tiempo $\Theta(1)$. En conclusión, el tiempo de ejecución nunca es peor que $\Theta(\log_2 n)$ pero sí puede ser mejor.

Usamos Big O significa que el tiempo de ejecución crece a lo más por una constante dada, pero este tiempo también puede crecer más lentamente.

*Un tiempo de ejecución $O(f(n))$ para una n lo suficientemente grande, el tiempo de ejecución es $k * f(n)$ para una constante k de esta forma:*

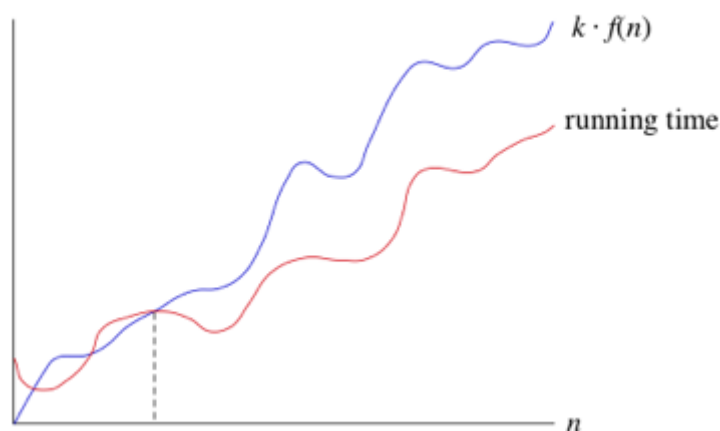


Ilustración 4: Tasa de crecimiento Big O.

Utilizamos Big O para cotas superiores asintóticas, ya que la cota es para el tiempo de ejecución por arriba de entradas con entradas suficientemente grandes.

Entonces, si comparamos la definición de Big Theta y Big O vemos que tienen similitudes, pero Big O se concentra únicamente en la cota de arriba. Se puede decir que el tiempo de ejecución en una búsqueda binaria va a ser siempre $O(\log_2 n)$ para cualquier valor de n .

Una búsqueda binaria puede ejecutarse siempre en un tiempo $O(\log_2 n)$ pero no siempre se ejecuta en un tiempo $\Theta(\log_2 n)$.

Big O dará un valor que puede ser mayor al tiempo de ejecución, pero esta en el rango, podemos utilizar una analogía como ejemplo. Si tenemos 10 dólares en el bolsillo y afirmamos a un amigo que tenemos menos de 100 dólares, la afirmación va a ser correcta pero no muy precisa. Así mismo funciona la notación Big O.

3. Notación Omega grande (Big Ω)

En la notación Omega grande decimos que el algoritmo toma por lo menos una cierta cantidad de tiempo, para una n suficientemente grande, el tiempo de ejecución es por lo menos $k * f(n)$ para una constante k . El tiempo de ejecución $\Omega(f(n))$ se grafica de la forma:

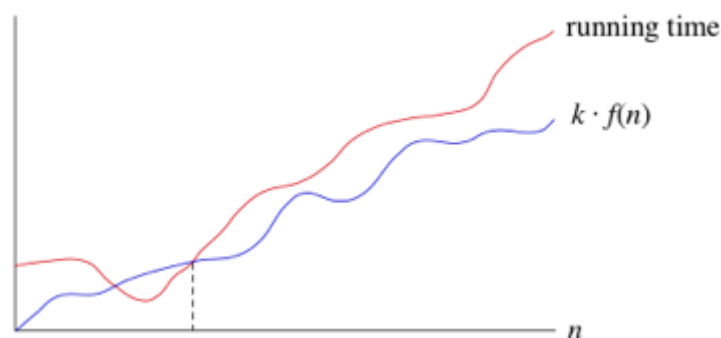


Ilustración 4: Tasa de crecimiento Big Ω .

A diferencia de Big O, Big Ω utiliza los límites asintóticos inferiores, ya que acota los crecimientos por debajo de un n lo suficientemente grande.

Así como $\theta(f(n))$ implica $O(f(n))$, también implica $\Omega(f(n))$, por ende, el peor caso de búsqueda binaria para Big Ω es $\Omega(\log_2 n)$.

Con la misma analogía del dinero en el bolsillo utilizado en Big O, podemos decir que, “tengo por lo menos 1 dólar en el bolsillo”. Sigue siendo correcto, pero no muy preciso, de igual forma es el caso de la búsqueda binaria de Big Ω al afirmar que el mejor de los casos es $\Omega(1)$, es correcto, pero no preciso pues sabemos que hay al menos un tiempo constante.

Aritmética de notación O.

Decimos que un algoritmo tiene un orden $O(n)$ si existen un n_0 y un c siendo $c > 0$, tal que para todo $n \geq n_0$, $T(n) \leq c * f(n)$. La relación O denota una dominancia de funciones, en donde la función f está acotada superiormente por un múltiplo de la función g (f es dominada por $c.g(n)$). Así, la expresión $f = O(g)$ refleja que el orden de crecimiento asintótico de la función f es inferior o igual al de la función g . (Díaz, N., 2014).

A continuación, se enumeran las propiedades básicas de la notación O:

- $k * O(f(n)) = O(f(n))$
- $O(f(n) + g(n)) = \max \{f(n); g(n)\}$
- $O(f(n)) * O(g(n)) = O(f(n) * g(n))$
- $O(O(f(n))) = O(f(n))$

Reglas de Cálculo de Complejidad:

- El tiempo de ejecución de cada sentencia simple, por lo común puede tomarse como $O(1)$. Algunas operaciones matemáticas no deben ser tratadas como operaciones elementales. Por ejemplo, el tiempo necesario para realizar sumas y productos crece con la longitud de los operandos. Sin embargo, en la práctica se consideran elementales siempre que los datos que se usen tengan un tamaño razonable.
- El tiempo de ejecución de una secuencia de proposiciones se determina por la regla de la suma. Es el máximo tiempo de ejecución de una proposición de la sentencia.
- Para las sentencias de bifurcación (IF, CASE) el orden resultante será el de la bifurcación con mayor orden.
- Para los bucles es el orden del cuerpo del bucle sumado tantas veces como se ejecute el bucle. Este tiempo debería incluir: el tiempo propio del cuerpo y el asociado a la evaluación de la condición y su actualización, en su caso. Si se trata de una condición sencilla (sin llamadas a función) el tiempo es $O(1)$, que no es relevante para al cálculo asintótico. Cuando se trata de un bucle donde todas las iteraciones son iguales, entonces el tiempo total será el producto del número de iteraciones por el tiempo que requiere cada una.
- Para los bucles anidados el orden corresponde a la multiplicación del orden del bucle interno por el orden del bucle externo.
- El orden de una llamada a un subprograma no recursivo es el orden del subprograma. (Díaz, N., 2014).
- Algunos de órdenes de complejidad que se presentan con frecuencia son:
 - Lineal: n
 - Cuadrático: n^2
 - Polinómico: n^k , con k que pertenece a los naturales.
 - Logarítmico: $\log(n)$
 - Exponencial: c^n

En la figura 1 se presenta una comparación entre los principales órdenes

de complejidad. Es de destacar el excesivo costo que representa una complejidad de orden 2^n , frente a la estabilidad de la complejidad logarítmica.

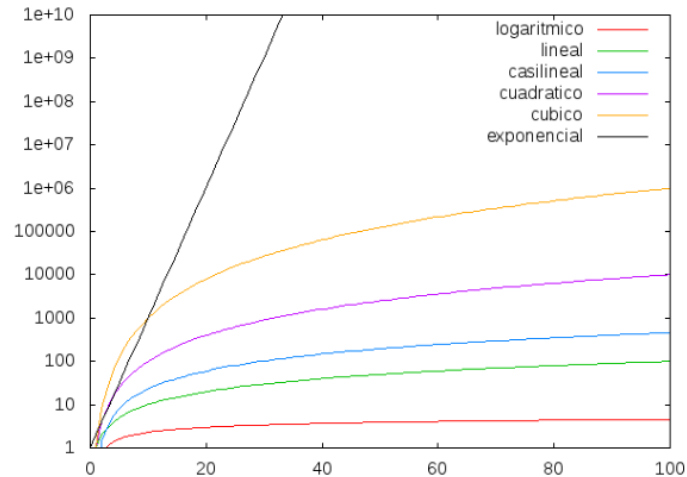


Imagen 1. Aritmética de notación O.

Ejemplo.

En el ejemplo se utiliza un bucle for para recorrer la matriz y encontrar el máximo elemento. La notación O para este algoritmo es $O(n)$, donde n es el tamaño de la matriz. Esto significa que el tiempo de ejecución crece linealmente con respecto al número de elementos en la matriz.

main.cpp

```
#include <iostream>
#include <vector>

// Función para encontrar el máximo elemento en una matriz
int encontrarMaximo(const std::vector<int>& arr) {
    int maxElemento = arr.at(0);
    for (int i = 1; i < arr.size(); ++i) {
        if (arr.at(i) > maxElemento) {
            maxElemento = arr.at(i);
        }
    }
    return maxElemento;
}

int main() {
    std::vector<int> numeros = { 10, 5, 8, 15, 3, 20, 12 };
    int maximoElemento = encontrarMaximo(numeros);
    std::cout << "El máximo elemento es: " << maximoElemento <<
std::endl;
    return 0;
}
```

Complejidad

1. Introducción

Un algoritmo es un método para resolver un problema, debe presentarse como una secuencia ordenada de instrucciones que siempre se ejecutan en un tiempo finito y con una cantidad de esfuerzo también finito. En un algoritmo siempre debe haber un punto de inicio y uno de terminación, estos deben ser únicos y deben ser fácilmente identificables (LUDA UAM-Azc., 2024).

Todo algoritmo debe cumplir con las siguientes características:

1. Debe ser **Preciso**; es decir, debe especificar sin ambigüedad el orden en que se deben ejecutar las instrucciones.
2. Debe ser **Definido**; esto es, que cada vez que se ejecute bajo las mismas condiciones, la secuencia de ejecución deberá ser la misma y entregar los mismos resultados.
3. Debe ser **Finito**; es decir, que siempre se realizarán un número finito de instrucciones en un tiempo y esfuerzo finitos.

2. Definición de Complejidad Algorítmica

Existen muchas alternativas de solución para un problema, debemos seleccionar el algoritmo más eficiente con el mejor conjunto de pasos, que su tiempo de ejecución sea el menor y cuyas líneas de código sean las menos posibles.

A simple vista parece algo muy simple, pero a medida que un programa crece, se requiere una medición más exacta y apropiada, para esto se realizan ciertas operaciones matemáticas que establecen la eficiencia teórica del programa, al estudio de estos casos se denomina **Complejidad Algorítmica**.

La **complejidad algorítmica** representa la cantidad de recursos (temporales y espaciales) que necesita un algoritmo para resolver un problema y por tanto permite determinar la eficiencia de dicho algoritmo, pero son medidas relativas al tamaño del problema. Un algoritmo será más eficiente comparado con otro, siempre que consuma menos recursos, como el tiempo y espacio de memoria necesarios para ejecutarlo.

1. Complejidad Temporal: Tiempo de cómputo necesario para ejecutar algún programa.

2. Complejidad Espacial: Memoria que utiliza un programa para su ejecución, indica la cantidad de espacio requerido para ejecutar el algoritmo; es decir, el espacio en memoria que ocupan todas las variables propias al algoritmo. Para calcular la memoria estática de un algoritmo se suma la memoria que ocupan las variables declaradas en el algoritmo. Para el caso de la memoria dinámica, el cálculo no es tan simple ya que, este depende de cada ejecución del algoritmo.

Nosotros estudiaremos las complejidades temporales, con este fin, para cada problema determinaremos una medida N , que llamaremos tamaño de la entrada o número de datos a procesar por el programa, intentaremos hallar

respuestas en función de dicha N .

Esto dependerá de la naturaleza del problema, por ejemplo, si hablamos de un arreglo se puede ver a N como el rango del arreglo, para una matriz, el número de elementos que la componen; para un grafo, podría ser el número de nodos o arcos, no se puede establecer una regla para N , pues cada problema tiene su propia complejidad.

La familia $O(f(n))$ representa un **Orden de Complejidad** de donde $f(n)$ es la función más sencilla de la familia.

Las funciones de complejidad algorítmica más habituales en las cuales el único factor del que dependen es el tamaño de la muestra de entrada n , ordenadas de mayor a menor eficiencia son:

$O(1)$	Orden constante
$O(\log n)$	Orden logarítmico
$O(n)$	Orden lineal
$O(n \log n)$	Orden cuasi-lineal
$O(n^2)$	Orden cuadrático
$O(n^3)$	Orden cúbico
$O(n^a)$	Orden polinómico
$O(2^n)$	Orden exponencial
$O(n!)$	Orden factorial

Imagen 2 Tipos de Complejidad algorítmica

A continuación, se mostrará su definición de cada tipo de complejidad

Complejidad	Definición
$O(1)$	Complejidad constante. Las instrucciones se ejecutan una vez.
$O(\log n)$	Complejidad logarítmica. Aparece en algoritmos como búsqueda binaria.
$O(n)$	Complejidad lineal. Se encuentra en bucles simples con instrucciones constantes.
$O(n \log n)$	Complejidad cuasi-lineal. Presente en algoritmos de tipo divide y vencerás.
$O(n^2)$	Complejidad cuadrática. Se da en bucles doblemente anidados.
$O(n^3)$	Complejidad cúbica. Se da en bucles con triple anidación.
<u>$O(n^a)$</u>	<u>Complejidad polinómica ($a > 3$). El tiempo de ejecución aumenta con n^a.</u>
$O(2^n)$	Complejidad exponencial. Tiempos de ejecución muy altos, poco prácticos.

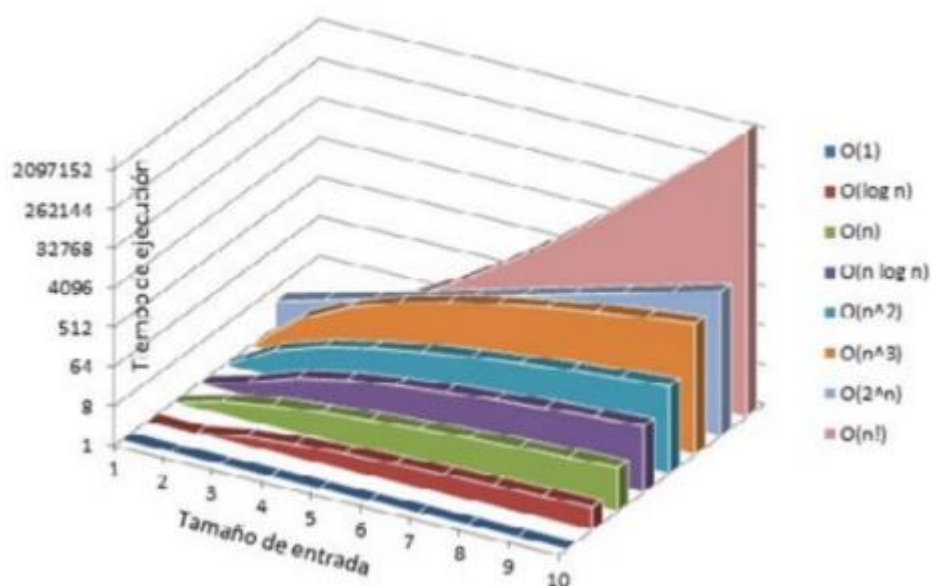


Imagen 2 Grafica de comportamiento de ordenes de crecimiento de algoritmos

3. Reglas elementales para el análisis de algoritmos

Cuando se dispone de distintos algoritmos para resolver un problema, es necesario definir cuál de ellos es el más eficiente. Una herramienta esencial para este propósito es el análisis de algoritmos. Este análisis suele efectuarse desde dentro del algoritmo hacia afuera, determinando el tiempo requerido por las instrucciones secuenciales y cómo se combinan estos tiempos con las estructuras de control que enlazan las distintas partes del algoritmo. A continuación, se presentan las reglas básicas para el análisis de algoritmos:

Operaciones elementales: el coste computacional de una operación elemental es 1 (asignación, en trata/salida, operación aritmética que no desborde el tamaño de la variable utilizada). La mayoría de las instrucciones de un algoritmo son operaciones elementales que están en el orden de $O(1)$.

Secuencia: en un fragmento de código, su coste computacional será la suma del coste individual de cada una de sus operaciones o fragmentos de código, aplicando la regla anterior. Es decir, si se cuenta con un fragmento de código compuesto de cinco operaciones elementales, el coste de dicho fragmento será $O(5)$.

Condicional *if-else* (decisión): si g es el coste de la evaluación de un condicional *if*, su valor es de $O(1)$ por cada condición que se deba evaluar (considerando siempre el peor caso, en el que se deba realizar todas las comparaciones dependiendo del operador lógico usado). A esto

se le suma el coste de la rama de mayor valor, sea esta la verdadera o falsa (considerando las anidadas si corresponde), aplicando las reglas anteriores.

Es decir el coste de un fragmento condicional es del orden de $O(g + \text{máximo}(\text{rama verdadera}, \text{rama falsa}))$.

Ciclo for: El coste de dicho fragmento se considera como una constante. Y el mismo se considera despreciable o de $O(n)$ respecto al número de iteraciones del ciclo.

Es decir el coste total estará en el orden de $O(n * O(1))$ lo que significa que es de $O(n)$. Si dentro del *F1* tenemos la presencia de otro ciclo del mismo tipo, el coste total estará en el orden de $O(n * n * O(1))$ lo que significa que es de $O(n^2)$.

Ciclo while: en este tipo de ciclos se aplica la regla anterior, solo se debe tener en cuenta que a veces no se cuenta con una variable de control numérica, sino que depende del tamaño de las estructuras con las que se esté trabajando y su dimensión, o del tipo de actividad que se realice dentro de dicho ciclo.

Recursividad: el cálculo del coste total de un algoritmo recursivo no es tan sencillo como los casos que vimos previamente. Para realizar este cálculo existen dos técnicas comúnmente utilizadas: ecuaciones recurrentes y teoremas maestros. Para realizar la primera de ellas se busca una ecuación que elimine la recursividad para poder obtener el orden de dicha función, muchas veces esto no es una tarea sencilla. En cambio, el segundo utiliza funciones condicionales y condiciones de regularidad para realizar el cálculo del coste. Estas técnicas antes mencionadas son avanzadas y exceden los alcances de este libro por lo que solo se explicarán algunos ejemplos utilizando la técnica de ecuaciones recurrentes para determinar el orden de funciones recurrentes.

4. Ejemplos de complejidad

Tenemos el siguiente ejemplo donde vamos a analizar la complejidad

Ejemplo 1:

Complejidad Lineal

```
void numeros_pares_impares(int numero) {
    int cont_imp = 0; // O(1)
    for (int i = 1; i <= numero; ++i) { // O(n)
        if (i % 2 == 0) { // O(1)
            std::cout << i << " Es Par" << std::endl; // O(1)
        } else { // O(1)
            std::cout << i << " Es Impar" << std::endl; // O(1)
            cont_imp++; // O(1)
        }
    }
}
```



```

        std::cout << "Cantidad de Números Impares: " << cont_imp <<
std::endl; // O(1)
    }

```

Para calcular la complejidad algorítmica de este código, primero analicemos los diferentes bloques de código:

1. El bucle **for** itera desde **1** hasta **numero** inclusive, donde **numero** es el parámetro de entrada.
2. Dentro del bucle, se realiza una verificación condicional **if (i % 2 == 0)** para determinar si **i** es par o impar.
3. En función de la paridad de **i**, se imprime un mensaje correspondiente.
4. Además, se incrementa el contador **cont_imp** cuando **i** es impar.
5. Finalmente, se imprime la cantidad de números impares encontrados.

Vamos a analizar la complejidad de cada uno de estos bloques:

- El bucle **for** ejecuta **numero** veces, por lo que su complejidad es $O(\text{numero})$.
- La verificación **if (i % 2 == 0)** se realiza en cada iteración del bucle, lo cual es una operación de tiempo constante $O(1)$.
- La impresión de un mensaje es una operación de tiempo constante $O(1)$.
- El incremento de **cont_imp** es una operación de tiempo constante $O(1)$.

Tenemos:

$$O(1) + O(n) * (O(1) + O(1) + O(1) + O(1)) + O(1) = O(n)$$

Por lo tanto, la complejidad total del programa es $O(n)$, donde n es el número ingresado como parámetro. Esto significa que el tiempo de ejecución del algoritmo crecerá linealmente con el valor de entrada **numero**.

Ejemplo 2:

Complejidad Cuadrática

```

void imprimir_pares(int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            std::cout << "(" << i << ", " << j << ") "; // O(1)
        }
        std::cout << std::endl; // O(1)
    }
}

```

Este código tiene un bucle anidado dentro de otro bucle, lo que resulta en una complejidad cuadrática. Ahora, hagamos el análisis de la complejidad:

- El bucle externo ejecuta n veces, donde n es el valor ingresado por el usuario.
- Dentro de cada iteración del bucle externo, el bucle interno también se ejecuta n veces.
- Por lo tanto, el número total de iteraciones del bucle interno es $n * n$, lo que resulta en una complejidad cuadrática.

Tenemos:

$$O(n) * (O(n) * O(1) + O(1)) = O(n^2)$$

Por lo tanto, la complejidad total del programa es $O(n^2)$, donde n es el valor de

entrada. Esto significa que el tiempo de ejecución del algoritmo crecerá cuadráticamente con el valor de entrada n .

Ejemplo 3: Complejidad algorítmica

```
int potencia_logaritmica(int base, int exponente) { // O(1)
    if (exponente == 0) // O(1)
        return 1; // O(1)

    int mitad_potencia = potencia_logaritmica(base, exponente /
2); // O(log n)
    int resultado = mitad_potencia * mitad_potencia; // O(1)

    if (exponente % 2 == 1) // O(1)
        resultado *= base; // O(1)

    return resultado; // O(1)
}
```

El análisis de complejidad de tu función **potencia_logaritmica** es diferente al ejemplo del bucle anidado, pero vamos a desglosarlo:

1. La verificación **if (exponente == 0)** y la operación de retorno **return 1;** son operaciones de tiempo constante $O(1)$. Esto se debe a que no importa el valor de **exponente**, estas operaciones siempre toman una cantidad constante de tiempo.
2. La llamada recursiva **int mitad_potencia = potencia_logaritmica(base, exponente / 2);** tiene una complejidad de $O(\log n)$. En cada llamada recursiva, el exponente se reduce a la mitad, lo que resulta en aproximadamente logarítmicamente muchas llamadas recursivas.
3. La multiplicación **int resultado = mitad_potencia * mitad_potencia;** es una operación de tiempo constante $O(1)$.
4. La verificación **if (exponente % 2 == 1)** y la multiplicación adicional **resultado *= base;** también son operaciones de tiempo constante $O(1)$.
5. La operación de retorno **return resultado;** es una operación de tiempo constante $O(1)$.

Entonces, podemos resumir la complejidad de esta función como la suma de todas estas operaciones:

$$O(1) + O(1) + O(\log n) + O(1) + O(1) + O(1) = O(\log n)$$

Por lo tanto, la complejidad total de tu función **potencia_logaritmica** es $O(\log n)$, donde n es el exponente. Esto significa que el tiempo de ejecución del algoritmo aumentará logarítmicamente con el valor del exponente.

5. Ejemplo practico

Como calcular el tiempo de ejecución de diferentes tipos de complejidad y la respectiva grafica

Para ello se utiliza Code::Blocks donde se tendrá que cambiar el compilador a 32bits para que el código funcione correctamente (esto se debe a que se utiliza la librería `"#include <graphics.h>"`, a continuación se mostrar como configurar su Code:: Blocks:

Primero abrimos Stting y Compiler para modificar:

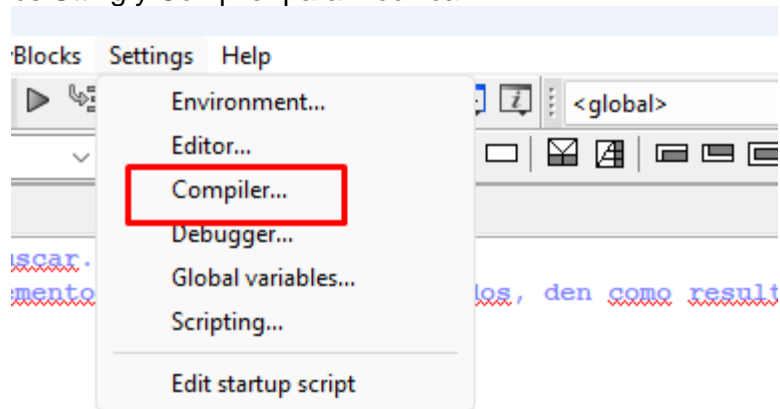


Imagen 3 Configuración de compilador.

A continuación tendrá que descargarse y cargar el compilador de 32bits a Code :: Blocks y colocar en linker settings lo siguiente: -lbg -lgdi32 -lcomdlg32 -luuid -loleaut32 -ole32

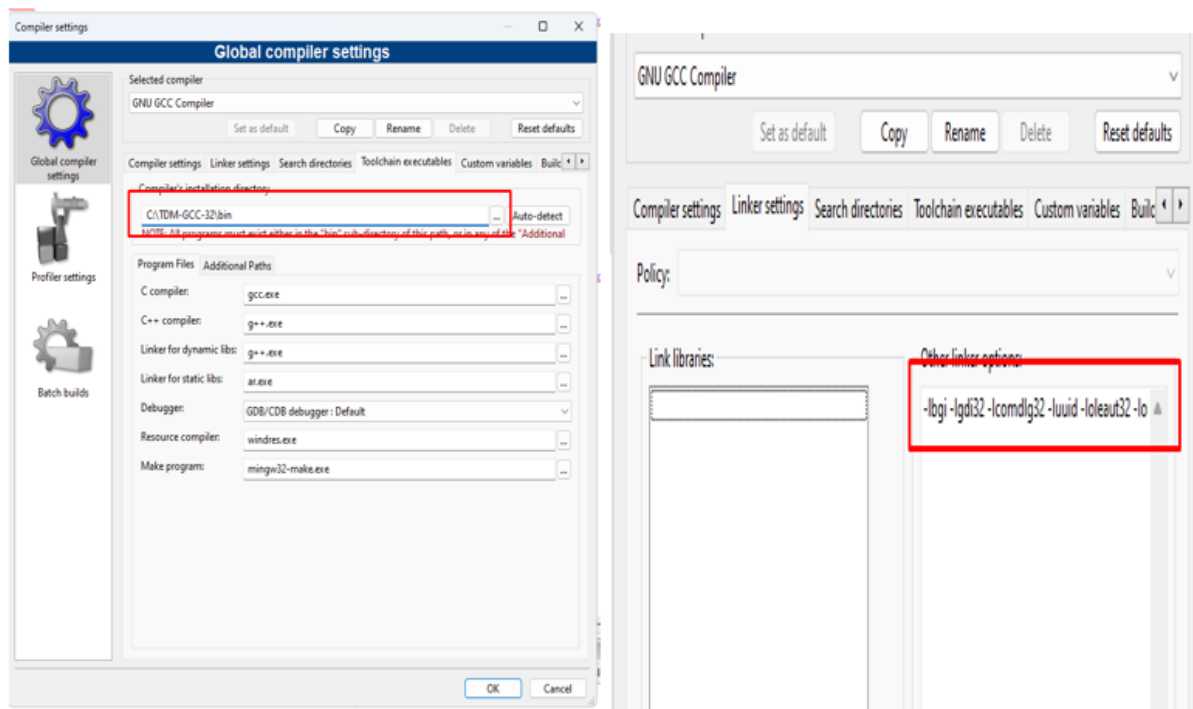


Imagen 4 Configuración de compilador a 32bits

Y ejecutaremos se presentará un fragmento del código donde se podrá observar la diferencia de tiempo en de dos funciones con el mismo propósito, pero con diferente tipo de complejidad (Lineal y Cuadrática)

función Cuadrática $O(n^2)$

```
vector<int> sumandos_version1(const vector<int> &numeros, int
suma) {

    for (auto a : numeros) {
```

```

        int diferencia = suma - a;
        for (auto b : numeros) {
            if ((b != a) && (b == diferencia))
                return {a, b};
        }
    }
    return {0, 0};
}

```

Función Lineal $O(n)$

```

vector<int> sumandos_version2(const vector<int> &numeros, int
suma) {

    set<int> conjunto;
    for (auto a : numeros) {
        int diferencia = suma - a;
        if (conjunto.find(diferencia) != conjunto.end())
            return {a, diferencia};
        else
            conjunto.insert(a);
    }
    return {0, 0};
}

```

Ejecución

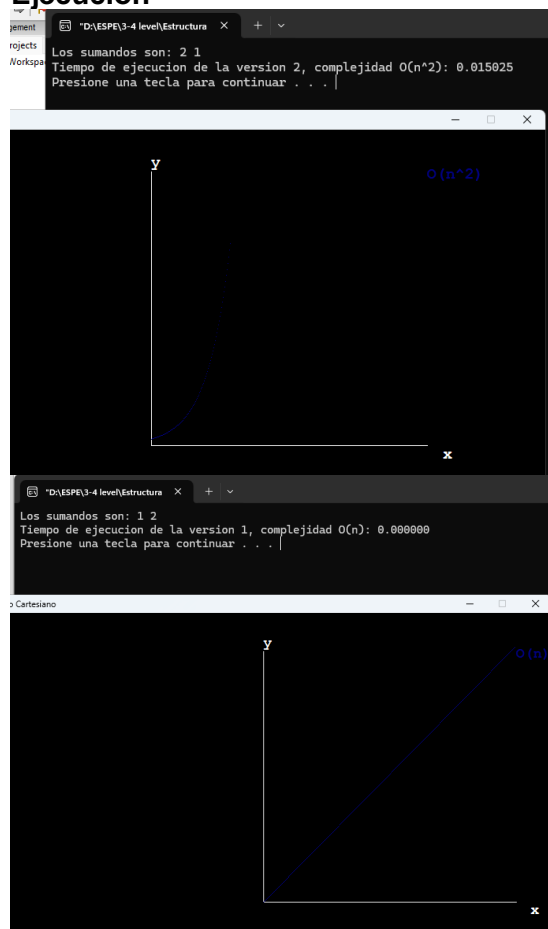


Imagen 5 Compilación función $O(n^2)$

Imagen 6 Compilación función $O(n)$

Donde podemos observar que el tiempo de ejecución de la función de complejidad $O(n^2)$ es mayor a la función de complejidad $O(n)$.

6. Anexo

Ejemplo 1:

```
#include <iostream>
void imprimir_pares(int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            std::cout << "(" << i << ", " << j << ") ";
        }
        std::cout << std::endl;
    }
}
int main() {
    int numero;
    std::cout << "Ingrese un número: ";
    std::cin >> numero;
    imprimir_pares(numero);
    return 0;
}
```

Ejemplo 2

```
#include <iostream>
void imprimir_pares(int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            std::cout << "(" << i << ", " << j << ") ";
        }
        std::cout << std::endl;
    }
}
int main() {
    int numero;
    std::cout << "Ingrese un número: ";
    std::cin >> numero;
    imprimir_pares(numero);
    return 0;
}
```

Ejemplo 3

```
#include <iostream>

int potencia_logaritmica(int base, int exponente) {
    if (exponente == 0)
        return 1;

    int mitad_potencia = potencia_logaritmica(base, exponente /
2);
    int resultado = mitad_potencia * mitad_potencia;
    if (exponente % 2 == 1)
        resultado *= base;
    return resultado;
}

int main() {
    int base, exponente;
    std::cout << "Ingrese la base: ";
    std::cin >> base;
    std::cout << "Ingrese el exponente: ";
```

```

    std::cin >> exponente;
    int resultado = potencia_logaritmica(base, exponente);
    std::cout << "El resultado de " << base << " elevado a " <<
exponente << " es: " << resultado << std::endl;
    return 0;
}

```

Ejemplo 5

Ejemplo practico

```

#include <iostream>
#include <vector>
#include <set>
#include <bits/stdc++.h>
#include <sys/time.h>
#include <windows.h>
#include <locale.h>
#include <conio.h>
#include <string>
#include <limits>
#include <graphics.h>
#include <math.h>

using namespace std;

vector<int> sumandos_version1(const vector<int> &numeros, int
suma) {
    for (auto a : numeros) {
        int diferencia = suma - a;
        for (auto b : numeros) {
            if ((b != a) && (b == diferencia))
                return {a, b};
        }
    }
    return {0, 0};
}

vector<int> sumandos_version2(const vector<int> &numeros, int
suma) {
    set<int> conjunto;
    for (auto a : numeros) {
        int diferencia = suma - a;
        if (conjunto.find(diferencia) != conjunto.end())
            return {a, diferencia};
        else
            conjunto.insert(a);
    }
    return {0, 0};
}

void graficarCuadraticaO_n2() {
    initwindow(900, 900, "Plano Cartesiano", 200, 100);
    // Dibujar ejes x e y
    setcolor(WHITE);
    line(400, 400, 750, 400); // Eje x
    line(400, 400, 400, 50); // Eje y
    // Etiquetas de los ejes
    settextstyle(DEFAULT_FONT, HORIZ_DIR, 0);
    outtextxy(770, 400, "x");
    outtextxy(400, 30, "y");
    // Dibujar la línea que representa la complejidad O(n^2) en el

```

```

primer cuadrante
    settextstyle(DEFAULT_FONT, HORIZ_DIR, 0);
    int x2, y2;
    double c = 8;
    for (x2 = 0; x2 <= 100; x2++) {
        y2 = 400 - c * pow(2, x2 / 20.0);
        if (y2 >= 50) { // Asegurarse de que y esté en el primer
cuadrante
            putpixel(x2 + 400, y2, BLUE); // Desplazar x2 para que
esté en el primer cuadrante
        }
    }
    // Etiqueta de la complejidad
    setcolor(BLUE);
    outtextxy(750, 45, "O(n^2)");
}

void graficarLinealO_n() {
    initwindow(800, 800, "Plano Cartesiano", 200, 100);

    // Dibujar ejes x e y
    setcolor(WHITE);
    line(400, 400, 750, 400); // Eje x
    line(400, 400, 400, 50); // Eje y
    // Etiquetas de los ejes
    settextstyle(DEFAULT_FONT, HORIZ_DIR, 0);
    outtextxy(770, 400, "x");
    outtextxy(400, 30, "y");
    // Dibujar la línea que representa la complejidad O(n)
    setcolor(BLUE);
    line(400, 400, 750, 45);
    // Etiqueta de la complejidad
    outtextxy(750, 45, "O(n)");
}

int main() {
    struct timeval inicio, fin;
    int totalOpciones = 3;
    int opcion = 1;

    while (true) {
        system("cls"); // Limpiar la pantalla antes de mostrar las
opciones
        cout << " \n\t COMPLEJIDAD ALGORITMICA? " << endl;
        for (int i = 1; i <= totalOpciones; i++) {
            if (i == opcion) {
                cout << " ";
                if (i == 1) cout << ">> ";
            } else {
                cout << " ";
            }
            switch (i) {
                case 1:
                    cout << "1: Lineal O(n)" << endl;
                    break;
                case 2:
                    cout << "2: Cuadratica O(n^2)" << endl;
                    break;
                case 3:

```

```

        cout << "3: Salir" << endl;
        break;
    }
}
int tecla = _getch(); // Utilizamos getch() en lugar de
_getch()
if (tecla == 224) { // Tecla de flecha
    tecla = _getch(); // Obtenemos la segunda parte del
código de la tecla de flecha
    switch (tecla) {
        case 72: // Flecha arriba
            opcion = (opcion > 1) ? opcion - 1 :
totalOpciones;
            break;
        case 80: // Flecha abajo
            opcion = (opcion < totalOpciones) ? opcion + 1
: 1; // Mover la opción hacia abajo, volver al inicio si está en
la parte inferior
            break;
    }
} else if (tecla == 13) { // Tecla Enter
    if (opcion == 3) {
        system("cls");
        cout << "Saliendo del programa..." << endl;
        break;
    } else if (opcion == 1) {
        system("cls");
        gettimeofday(&inicio, NULL);
        ios_base::sync_with_stdio(false);
        vector<int> resultado = sumandos_version1({6, 1,
4, 2, 9, 7, 3, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30}, 3);
        cout << "Los sumandos son: ";
        for (int num : resultado) {
            cout << num << " ";
        }
        cout << endl;
        gettimeofday(&fin, NULL);
        double tiempo_ejecucion = (fin.tv_sec -
inicio.tv_sec) * 1e6;
        tiempo_ejecucion = (tiempo_ejecucion +
(fin.tv_usec - inicio.tv_usec)) * 1e-6;
        cout << "Tiempo de ejecucion de la version 1,
complejidad O(n): " << fixed << tiempo_ejecucion << endl;
        graficarLinealO_n(); // Supongo que esta función
la tienes definida en otro lugar
        system("pause");
    } else if (opcion == 2) {
        system("cls");
        gettimeofday(&inicio, NULL);
        vector<int> resultado = sumandos_version2({6, 1,
4, 2, 9, 7, 3, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30}, 3);
        cout << "Los sumandos son: ";
        for (int num : resultado) {
            cout << num << " ";
        }
        cout << endl;
    }
}

```



```

        gettimeofday(&fin, NULL);
        double tiempo_ejecucion = (fin.tv_sec -
inicio.tv_sec) * 1e6;
        tiempo_ejecucion = (tiempo_ejecucion +
(fin.tv_usec - inicio.tv_usec)) * 1e-6;
        cout << "Tiempo de ejecucion de la version 2,
complejidad O(n^2): " << fixed << tiempo_ejecucion << endl;
        graficarCuadraticaO_n2();
        system("pause");
    }
}
}
cout << "Presione cualquier tecla para continuar...";
_getch();
return 0;
}

```

7. Guía de ejercicios prácticos

A continuación, se plantean una serie de fragmentos de códigos. Habrá que realizar un análisis de la complejidad de cada una de las diferentes funciones.

Ejercicio 1:

```

int suma_arreglo(const std::vector<int>& arr) {
    int suma = 0;
    for (int num : arr) {
        suma += num;
    }
    return suma;
}

```

Ejercicio 2:

```

bool buscar_elemento(const std::vector<int>& lista, int elemento)
{
    for (int num : lista) {
        if (num == elemento)
            return true;
    }
    return false;
}

```

Ejercicio 3:

```

int fibonacci_recursivo(int n) {
    if (n <= 1)
        return n;
    return fibonacci_recursivo(n - 1) + fibonacci_recursivo(n -
2);
}

```

Análisis de Recurrencia

1. Introducción

En el vasto dominio de la ciencia de la computación, el análisis de recursos destaca como una herramienta vital para la eficiencia y la toma de decisiones en el desarrollo de software. Enfocado en algoritmos y estructuras de datos, este análisis desentraña la complejidad computacional, evaluando recursos desde tiempo de ejecución hasta espacio de memoria. Este proceso no es meramente teórico; es una brújula pragmática que guía decisiones cruciales en el diseño de algoritmos, superando desafíos prácticos como simplificaciones teóricas y variabilidades de hardware. Exploraremos estos fundamentos, resaltando su relevancia y desafíos en el contexto real, revelando cómo convergen las decisiones informadas y el diseño eficiente en la intersección de tiempo, espacio y algoritmos.

2. Definición

El análisis de recursos, en el contexto de algoritmos y estructuras de datos, se refiere a la evaluación de los recursos computacionales utilizados por un algoritmo. Estos recursos pueden incluir tiempo de ejecución, espacio de memoria, ancho de banda de red u otros recursos relevantes para el problema en cuestión. El objetivo principal del análisis de recursos es entender y cuantificar cómo un algoritmo utiliza estos recursos en función del tamaño de la entrada.

El análisis de recursos es esencial para tomar decisiones informadas sobre la eficiencia de los algoritmos y estructuras de datos, y para diseñar soluciones que optimicen el uso de recursos en aplicaciones del mundo real. Hay dos aspectos principales del análisis de recursos:

1. **Análisis de Tiempo:** Se centra en medir o estimar el tiempo de ejecución de un algoritmo en función del tamaño de la entrada. Esto implica la evaluación del rendimiento temporal en términos de operaciones básicas o complejidad algorítmica. Las relaciones recurrentes y las notaciones como la notación Big O son comúnmente utilizadas en el análisis de tiempo para describir la eficiencia temporal de los algoritmos.

2. **Análisis de Espacio:** Examina la cantidad de memoria o espacio de almacenamiento requerido por un algoritmo en función del tamaño de la entrada. Este análisis es fundamental para determinar la escalabilidad y eficiencia en términos de la utilización de recursos de memoria. La complejidad espacial también se expresa comúnmente utilizando notaciones como la notación Big O.

3. Proceso

Para realizar el análisis de recurrencia en estructuras de datos, se sigue generalmente este proceso:

1. **Definición de la recurrencia:** Se establece una ecuación que describe la relación entre el tiempo de ejecución de una operación en la estructura de datos y el tamaño del problema.
2. **Resolución de la recurrencia:** Se intenta resolver la ecuación de recurrencia para obtener una expresión cerrada que represente el tiempo de ejecución en términos del tamaño del problema.
3. **Análisis del resultado:** Se analiza el resultado obtenido para comprender el comportamiento temporal del algoritmo o estructura de datos en función del tamaño del problema.

Este análisis es fundamental para comprender la eficiencia y complejidad temporal de algoritmos y estructuras de datos, ya que proporciona información sobre cómo el rendimiento del algoritmo evoluciona a medida que el tamaño del problema crece.

Es importante destacar que el análisis de recurrencia no solo se aplica a algoritmos y estructuras de datos recursivos, sino que también puede utilizarse en contextos no recursivos donde haya una relación recurrente entre el tamaño del problema y el tiempo de ejecución.

Imagina que estás jugando con bloques de construcción y quieres entender cuánto tiempo te llevará construir una torre de bloques.

4. Ejemplo de análisis

Un ejemplo común en el análisis de recurrencia es la relación para el tiempo de ejecución de algoritmos de ordenación, como el Merge Sort o el QuickSort.

Estas relaciones recurrentes suelen tomar la forma de:

$$T(n)=a \cdot T(bn)+f(n)$$

Donde:

- $T(n)$ es el tiempo de ejecución del algoritmo para un problema de tamaño n .
- a es el número de sub problemas en los que se divide el problema principal.
- b es el factor por el cual se reduce el tamaño del problema en cada iteración.
- $f(n)$ es la función que representa el tiempo necesario para dividir el problema, combinar las soluciones de los sub problemas y cualquier trabajo adicional realizado en cada nivel de recursión.

El análisis de recurrencia es un paso importante para comprender la eficiencia de los algoritmos y estructuras de datos, y puede ayudar a tomar decisiones informadas sobre qué enfoques son más eficientes para problemas particulares.

5. Ventajas

1. **Diseño Eficiente:** Ayuda en el diseño eficiente de algoritmos, permitiendo a los desarrolladores comprender y optimizar el rendimiento en términos de recursos computacionales.
2. **Comparación de Algoritmos:** Facilita la comparación objetiva de diferentes algoritmos para un problema dado, permitiendo identificar cuál es más eficiente en términos de tiempo y espacio.
3. **Escalabilidad:** Permite evaluar la escalabilidad de un algoritmo, es decir, cómo se comporta a medida que el tamaño de la entrada aumenta, proporcionando una visión clara de su rendimiento a gran escala.
4. **Identificación de Problemas:** Ayuda a identificar y corregir posibles problemas de rendimiento antes de implementar un algoritmo en un entorno de producción.

5. Comunicación Efectiva: Facilita la comunicación entre los desarrolladores al proporcionar una métrica objetiva para discutir y comparar el rendimiento de los algoritmos.

6. Desventaja

1. Simplificaciones Teóricas: A menudo, el análisis de recursos se basa en simplificaciones teóricas y asume un modelo de máquina abstracta, lo que puede no reflejar con precisión las complejidades del hardware y del entorno de ejecución real.
2. Variabilidad del Hardware: El rendimiento real puede variar según el hardware subyacente, el compilador utilizado y otros factores específicos de la implementación.
3. Omisión de Factores Constantes: La notación Big O y otras formas de análisis asintótico tienden a ignorar factores constantes, lo que significa que dos algoritmos con la misma complejidad asintótica pueden tener diferencias significativas en el rendimiento real.
4. No Considera Constantes Ocultas: A veces, la notación Big O no tiene en cuenta constantes ocultas que podrían afectar el rendimiento en situaciones de pequeña escala.
5. Complejidad Espacial vs. Temporal: El análisis de recursos a menudo se centra más en la complejidad temporal y puede pasar por alto la importancia de la complejidad espacial en algunos contextos.
6. No Toma en Cuenta Factores Externos: No considera factores externos como la variabilidad en la entrada de datos o el impacto de operaciones de entrada/salida, que pueden afectar el rendimiento real.

1. Aplicaciones

1. Diseño de Algoritmos Eficientes: Ayuda en el diseño de algoritmos que utilizan eficientemente los recursos disponibles, minimizando el tiempo y el espacio requeridos para resolver un problema.

2. Desarrollo de Software: Es esencial en el desarrollo de software para garantizar que las aplicaciones sean eficientes y respondan de manera rápida, especialmente en entornos donde los recursos son limitados.
3. Optimización de Bases de Datos: En el diseño y la optimización de bases de datos, el análisis de recursos es crucial para garantizar consultas rápidas y eficientes.
4. Computación en la Nube: En entornos de computación en la nube, donde los recursos son compartidos y pagados según el uso, el análisis de recursos es fundamental para controlar costos y mejorar la eficiencia.
5. Sistemas Empotrados: En el desarrollo de sistemas empujados con recursos limitados, el análisis de recursos es crucial para garantizar que el software funcione eficientemente en hardware con restricciones.

7. Análisis

El análisis de recursos en algoritmos y estructuras de datos es un tema fundamental en la ciencia de la computación que busca entender y cuantificar cómo los algoritmos utilizan los recursos computacionales, como el tiempo y el espacio, en función del tamaño de la entrada. Este análisis es esencial para diseñar algoritmos eficientes y tomar decisiones informadas en el desarrollo de software. A continuación, se presentan algunos aspectos clave del análisis del tema:

Importancia del Análisis de Recursos:

Eficiencia del Software: Permite diseñar software eficiente que resuelva problemas de manera rápida y con un uso óptimo de los recursos disponibles.

Toma de Decisiones: Facilita la toma de decisiones informadas al elegir entre diferentes algoritmos o enfoques para resolver un problema particular.

Optimización: Ofrece una base teórica para la optimización de algoritmos y estructuras de datos, mejorando su rendimiento en situaciones del mundo real.

Aspectos Clave del Análisis:

Análisis de Tiempo: Evalúa el tiempo de ejecución del algoritmo en función del tamaño de la entrada. Utiliza notaciones como Big O para expresar la

complejidad temporal.

Análisis de Espacio: Examina la cantidad de memoria o espacio de almacenamiento necesario en función del tamaño de la entrada. La complejidad espacial también se expresa con notación Big O.

Comparación de Algoritmos: Permite comparar diferentes algoritmos y seleccionar el más eficiente para un problema específico.

Consideración de Casos Especiales: Analiza el rendimiento en mejores casos, peores casos y casos promedio para comprender el comportamiento del algoritmo en diversas situaciones.

Validación Empírica: Complementa el análisis teórico con pruebas prácticas para validar las predicciones y garantizar que la eficiencia se mantenga en situaciones reales.

8. Ejemplo – Resolución de una Recurrencia por el Método de Expansión

Ejemplo

Suponga que tenemos la siguiente relación de recurrencia

$$T(n) = T(n - 1) + 3$$

$$T(0) = 2$$

Suponga que queremos hallar $T(7)$:

$$\begin{aligned} T(7) &= T(6) + 3 = T(5) + 3 + 3 = T(5) + 2*3 \\ &= T(4) + 3 + 2*3 = T(4) + 3*3 = T(3) + 3 + 3*3 = T(3) + 4*3 \\ &= T(2) + 3 + 4*3 = T(2) + 5*3 = T(1) + 3 + 5*3 = T(1) + 6*3 \\ &= T(0) + 3 + 6*3 = T(0) + 7*3 = 2 + 7*3 = 23 \end{aligned}$$

Figura 1: Primera Parte de la resolución de una recurrencia por expansión.

Realizado por Cordova, J. Extraído de: <https://www.youtube.com/watch?v=CWiUBI2C9Q0>

Ejemplo

En general, para la relación:

$$T(n) = T(n - 1) + 3$$

$$T(0) = 2$$

$$T(n) = T(n - 1) + 3 = T(n - 2) + 2 \cdot 3 = T(n - 3) + 3 \cdot 3$$

$$= T(n - 4) + 4 \cdot 3 = \dots = T(n - n) + n \cdot 3$$

$$= T(0) + 3n = 3n + 2$$

Por tanto, la solución es $T(n) = 3n + 2$

Note que $T(7) = 23$

Figura 2: Segunda Parte de la resolución de una recurrencia por expansión

Realizado por Cordova, J. Extraído de: <https://www.youtube.com/watch?v=CWiUBI2C9Q0>

9. Conclusiones

- **Importancia del Análisis de Recursos:** El análisis de recursos es fundamental para entender cómo los algoritmos utilizan el tiempo y el espacio, permitiendo tomar decisiones informadas en el diseño de software y la elección de algoritmos.
- **Optimización de Algoritmos:** A través del análisis de recursos, se puede lograr la optimización de algoritmos, diseñándolos de manera que utilicen eficientemente los recursos disponibles y mejoren el rendimiento general del software.
- **Comparación de Algoritmos:** Permite comparar diferentes enfoques para resolver un problema, facilitando la elección del algoritmo más adecuado en función de las necesidades específicas y las restricciones de recursos.
- **Escalabilidad:** El análisis de recursos proporciona información sobre cómo el rendimiento de un algoritmo escala a medida que aumenta el tamaño de la entrada, lo que es crucial para aplicaciones que deben manejar conjuntos de datos cada vez más grandes.

- Validación Empírica: Complementar el análisis teórico con pruebas prácticas es esencial para garantizar que las predicciones sean consistentes en situaciones del mundo real y para confirmar la eficiencia del software.

10. Recomendaciones:

- Considerar Factores Prácticos: Aunque el análisis teórico es valioso, es importante considerar factores prácticos como la implementación específica, la arquitectura del hardware y el entorno de ejecución, ya que estos pueden influir en el rendimiento real.
- Actualización Constante: Dado que la tecnología y las plataformas evolucionan, es recomendable actualizar regularmente el análisis de recursos para asegurarse de que las decisiones de diseño sigan siendo válidas y eficientes en contextos cambiantes.
- Uso de Herramientas de Perfilado: Para validar las predicciones teóricas, se recomienda utilizar herramientas de perfilado para realizar un seguimiento del rendimiento real del software en diferentes situaciones y entornos.
- Experimentación en Diversas Escalas: Realizar experimentos con conjuntos de datos de diferentes tamaños y enfoques de entrada para comprender completamente el comportamiento del algoritmo en diversas escalas.
- Documentación Clara: Documentar el análisis de recursos junto con las decisiones de diseño tomadas proporcionará una guía valiosa para el mantenimiento futuro y para otros desarrolladores que trabajen en el proyecto.

Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.
- Skiena, S. S. (2008). *The Algorithm Design Manual*.

- Sedgewick, R., & Wayne, K. (2011). *Algorithms*. Addison-wesley professional.
- Sedgewick, R., & Wayne, K. (2016). *Computer science: An interdisciplinary approach*. Addison-Wesley Professional.
- Javier Cordova. (2015, November 29). *solucion de relaciones de recurrencia* [Video]. YouTube. <https://www.youtube.com/watch?v=CWiUBI2C9Q0>

Bibliografía:

- Cormen T, Leiserson, Rivest and Stein. (2002) Introduction to Algorithms, 2nd edition. Disponible en: <https://athena.nitc.ac.in/summerschool/Files/clrs.pdf>
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.
- Sedgewick, R., & Wayne, K. (2021). Algorithms (4th ed.). Addison-Wesley Professional.
- Knuth, D. E. (1997). The art of computer programming, volume 1: Fundamental algorithms (3rd ed.). Addison-Wesley.
- GeeksforGeeks. (n.d.). Backtracking algorithms. Retrieved March 10, 2023, from <https://www.geeksforgeeks.org/backtracking-algorithms/>
- Estructura de datos en C++, Luis Joyanes e Ignacio Zahonero.
- (2010) Programación Dinámica. Available at: https://ocw.ehu.eus/pluginfile.php/40172/mod_resource/content/1/disenio_alg/contenidos/programacion-dinamica.pdf (Accessed: 06 March 2024).
- TOMi.digital - Programación dinámica. (s. f.). TOMi.Digital. https://tomi.digital/es/27819/programacion-dinamica?utm_source=google&utm_medium=seo
- inDrive.Tech. (2023, 4 septiembre). Comprender la programación dinámica para poder utilizarla de forma eficaz. HackerNoon. <https://hackernoon.com/es/comprender-la-programacion-dinamica-para-que-puedas-usarla-eficazmente>

- Frias S, (04 de abril de 2021), freeCodeCamp, Significado del algoritmo divide y venceras, <https://www.freecodecamp.org/espanol/news/significado-del-algoritmo-divide-y-venceras/>
- Yañez S, (02 de marzo de 2019), Ciberaula, Divide y Vencerás, https://www.ciberaula.com/cursos/java/divide_venceras.php
- PhinX Lab. (2023, 22 de febrero). ¿Qué es un algoritmo voraz o greedy? Medium. Recuperado de: <https://phinxlab.medium.com/qu%C3%A9-es-un-algoritmo-voraz-o-greedy-6c47ee3b7281>
- Olimpiada Informática. (s. f.). Algoritmia voraz. <https://olimpiada-informatica.org/>
- de la Cueva, V., González, L., & Salinas, E. (2020). Estructuras de datos y algoritmos fundamentales (pp. 51–58). Editorial Digital del Tecnológico de Monterrey. https://www.google.com.ec/books/edition/Estructuras_de_datos_y_algoritmos_fundam/MXf1DwAAQBAJ?hl=es-419&gbpv=0
- hu, y. (2021). Algoritmos C++: Explica Los Algoritmos de C++ con Bellas Imágenes. Aprende de Forma Fácil y Mejor. (n.p.): Independently Published.
- <https://www.amazon.com/s?k=9798592521423&i=stripbooks&linkCode=qs>