2024

UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE

MANUAL DE ESTRUCTURAS DE DATOS
ORDENACIÓN, BÚSQUEDA Y ESTRUCTURAS DE DATOS AVANZADAS.

TERCER SEMESTRE
NRC: 14675

Contenido

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

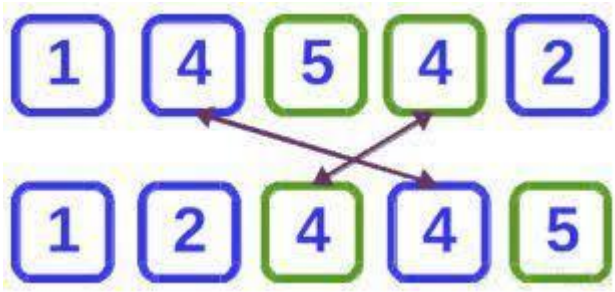


Imagen 1*funcionamiento del algoritmo de intercambio*

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).

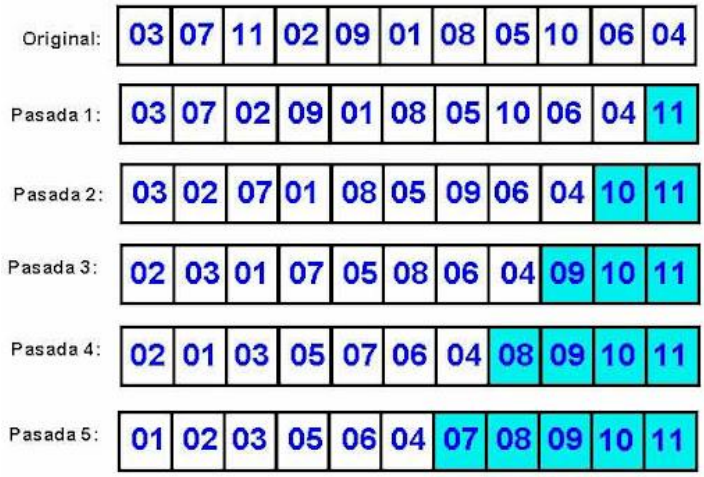


Imagen 2 Funcionamiento de algoritmo Burbuja de forma gráfica.

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.

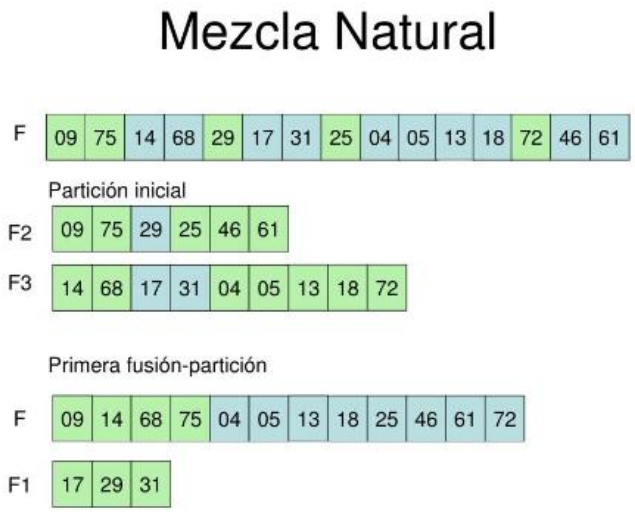


Imagen 3 *Ciclo de vida del TDA*

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 especifico 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;
}
```

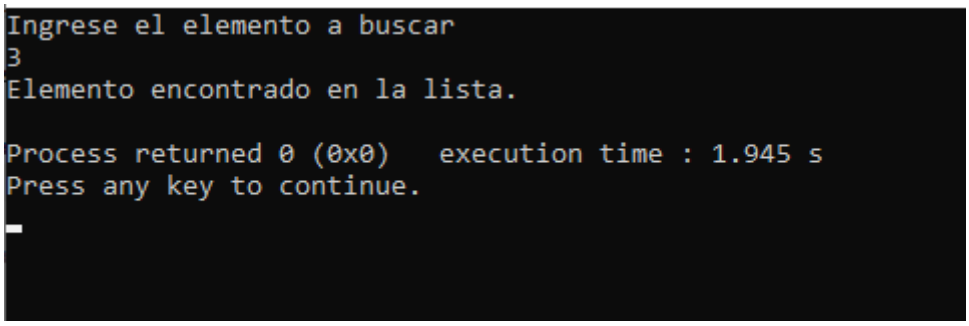


Imagen 4: Ejecución del código ejemplo: Algoritmo de búsqueda secuencial

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.

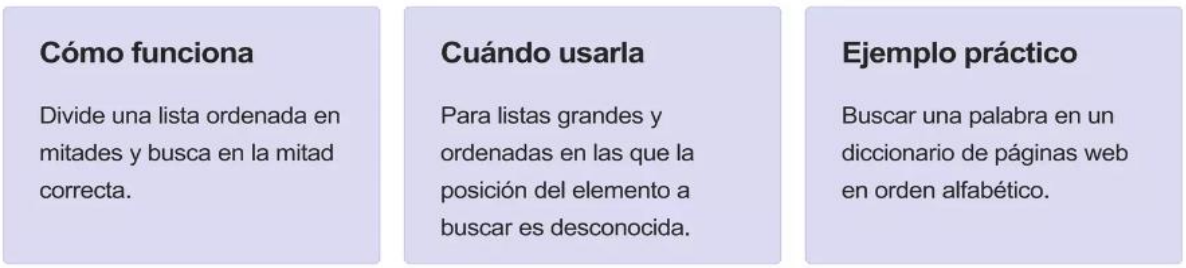


Imagen 5 Descripción del algoritmo

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 <o> 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 log2n, el máximo log2n

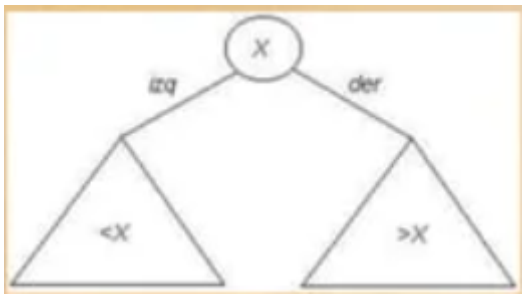


Imagen 6 Comparación de elementos

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.

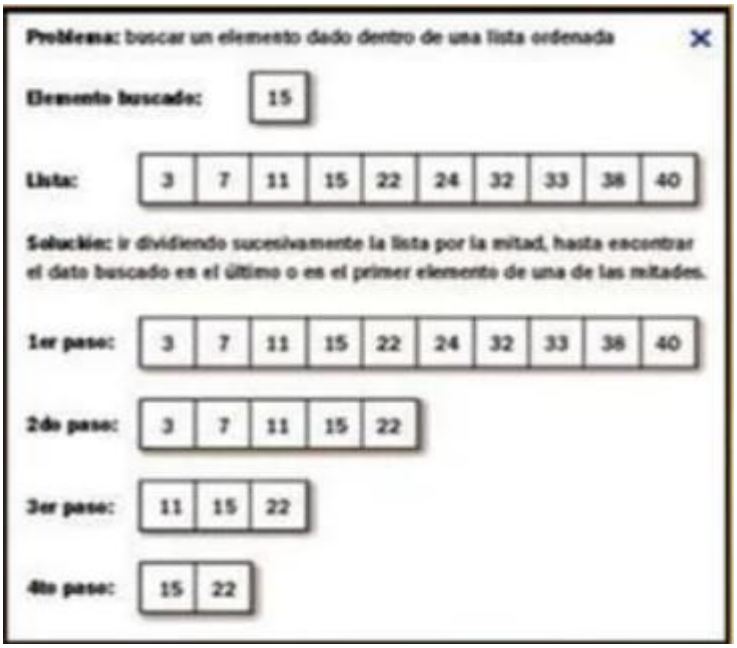


Imagen 7 Ejemplo de una búsqueda binaria

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

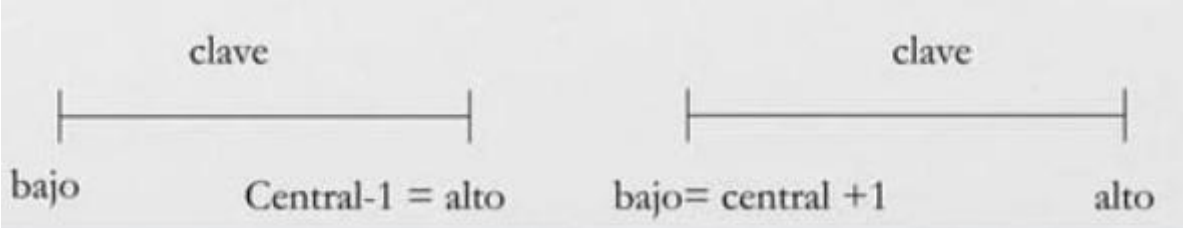


Imagen 8 Comparación del elemento central con la clave

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

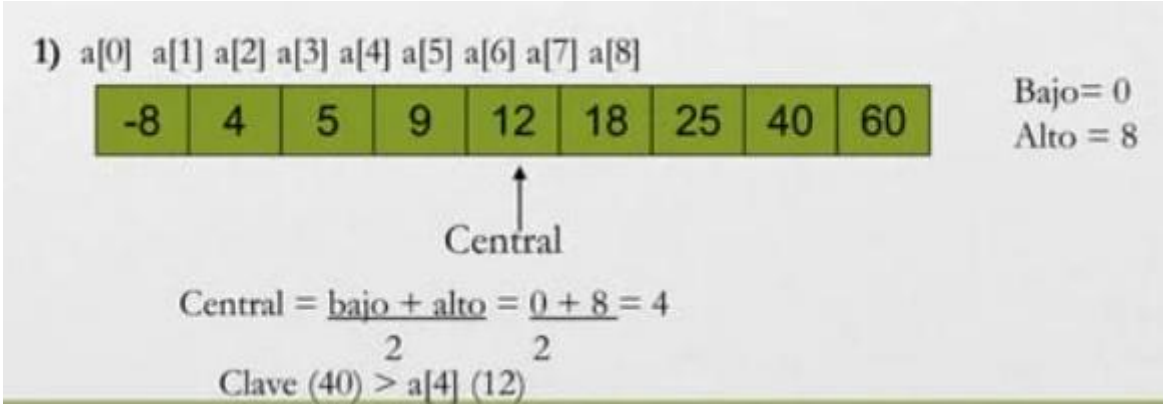


Imagen 9 Cálculo del punto central y primera comparación del algoritmo

2) Buscar en la sublista derecha

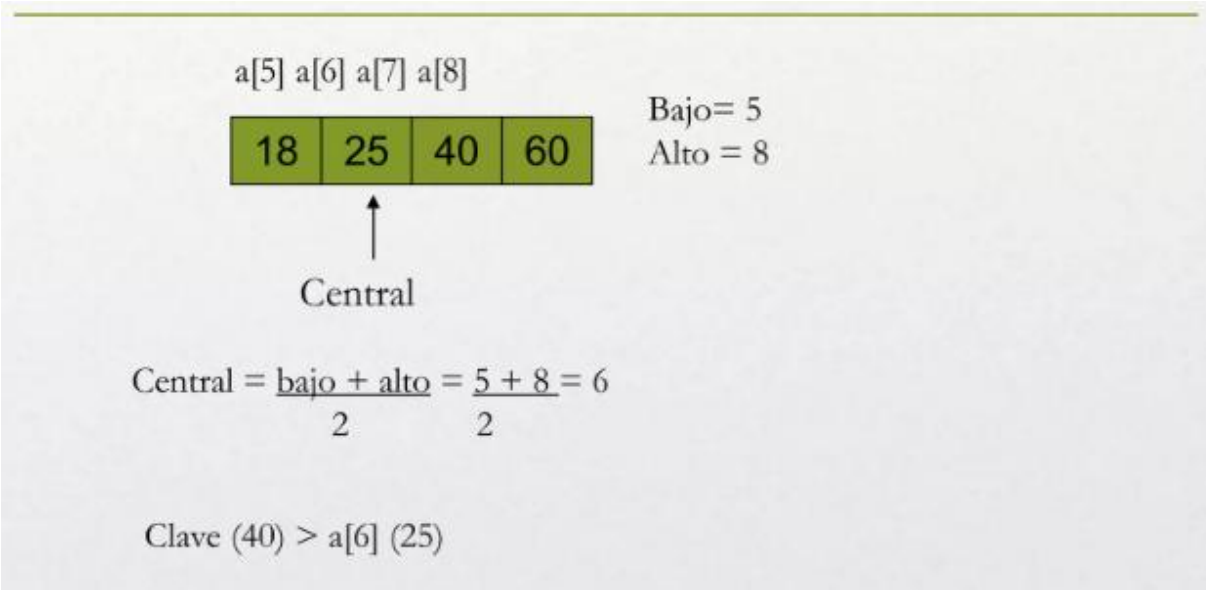


Imagen 10 Segunda comparación del algoritmo

3) Buscar en la sublista derecha

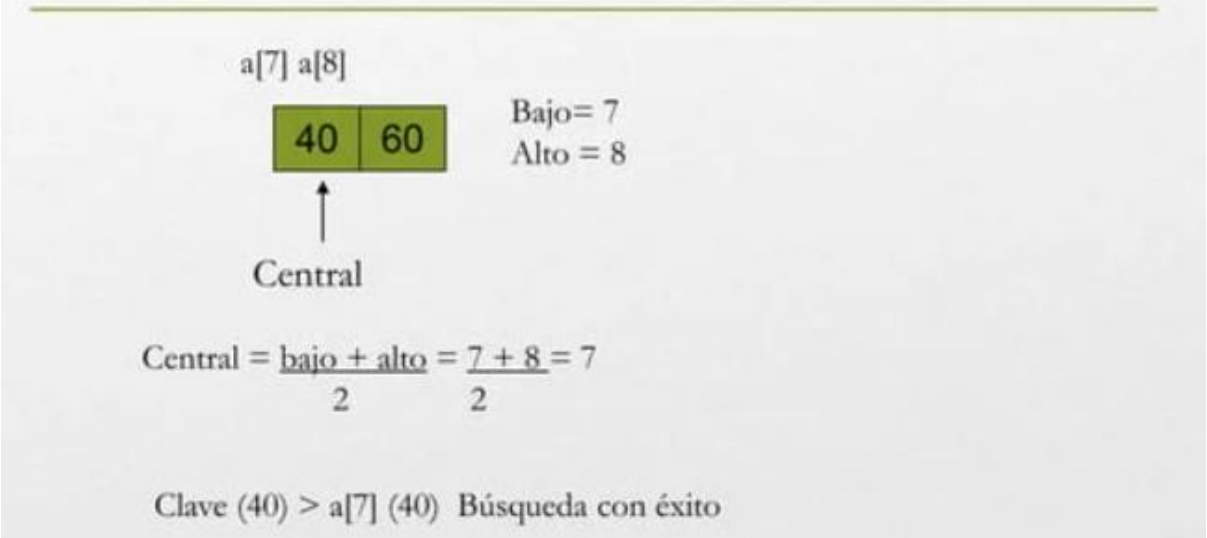


Imagen 11 Segunda comparación del algoritmo

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

Inserción (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.

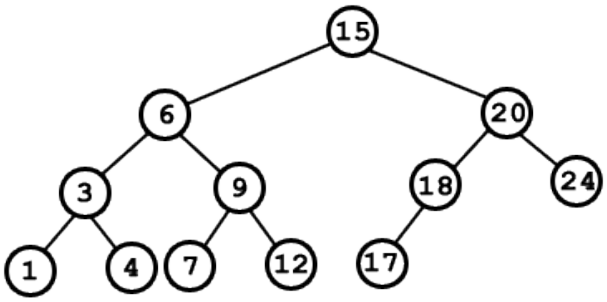


Imagen 12: Árbol básico

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.

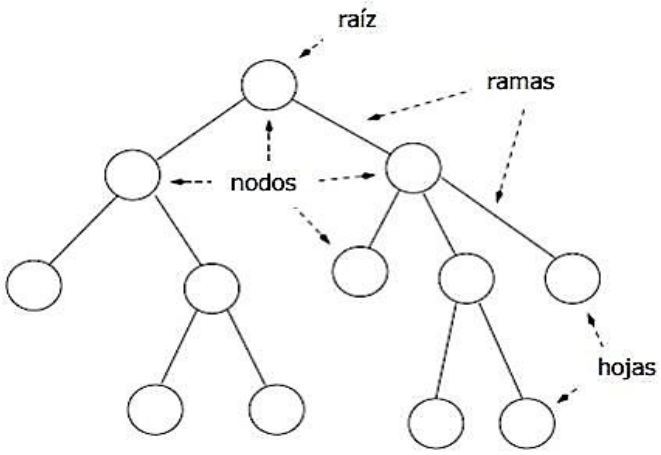


Imagen 13: Características del Árbol

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.

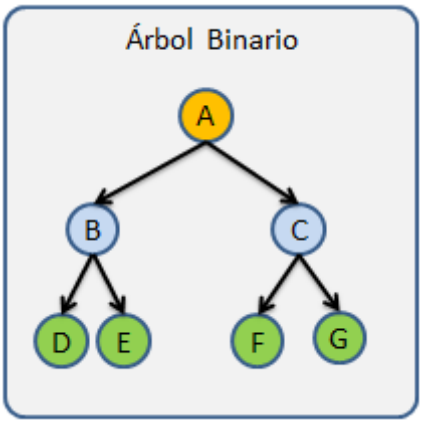


Imagen 15 *Ciclo de vida del TDA*

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.

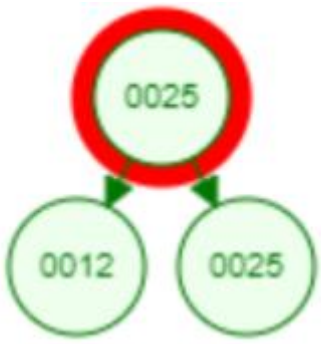


***Imagen** 17 Raíz Actual*

b. Comparación del Valor:

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

***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.

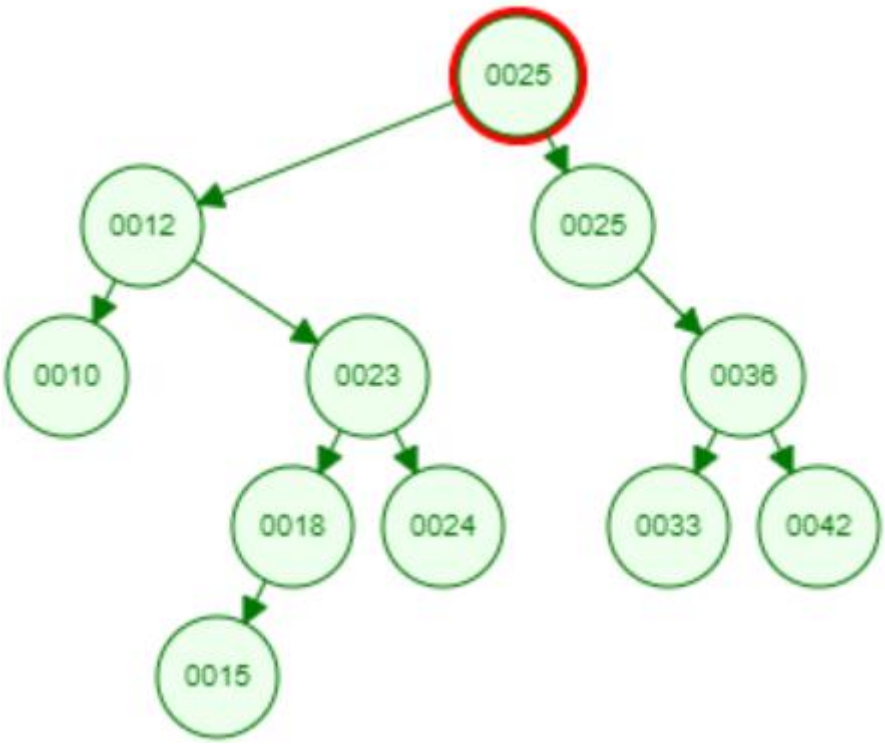


Imagen 21 *Buscar el nodo a eliminar*

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.

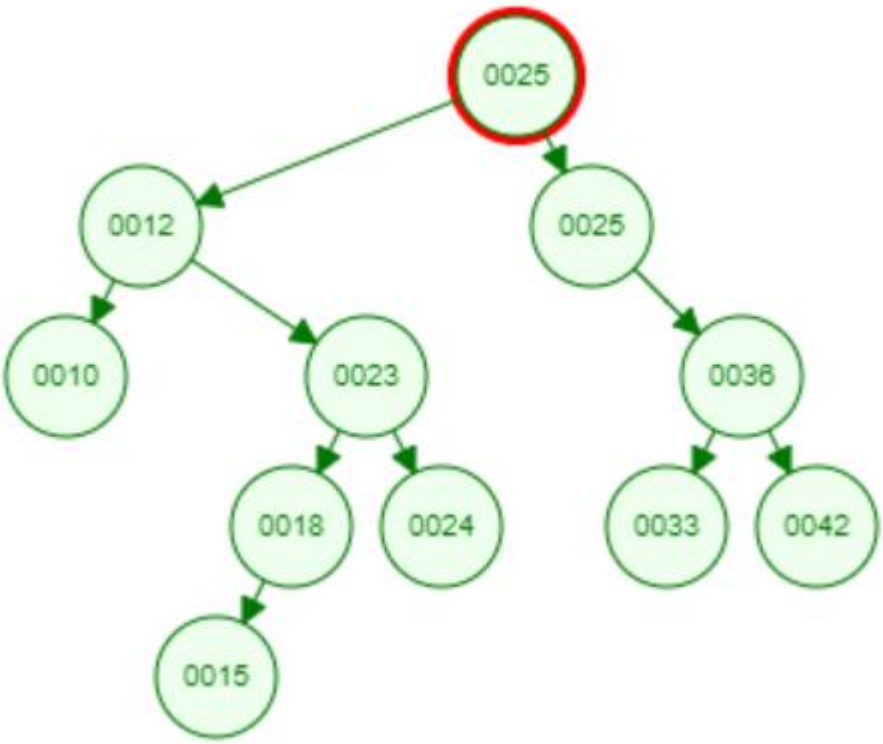


Imagen 22 *Buscar el nodo a eliminar*

- **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).

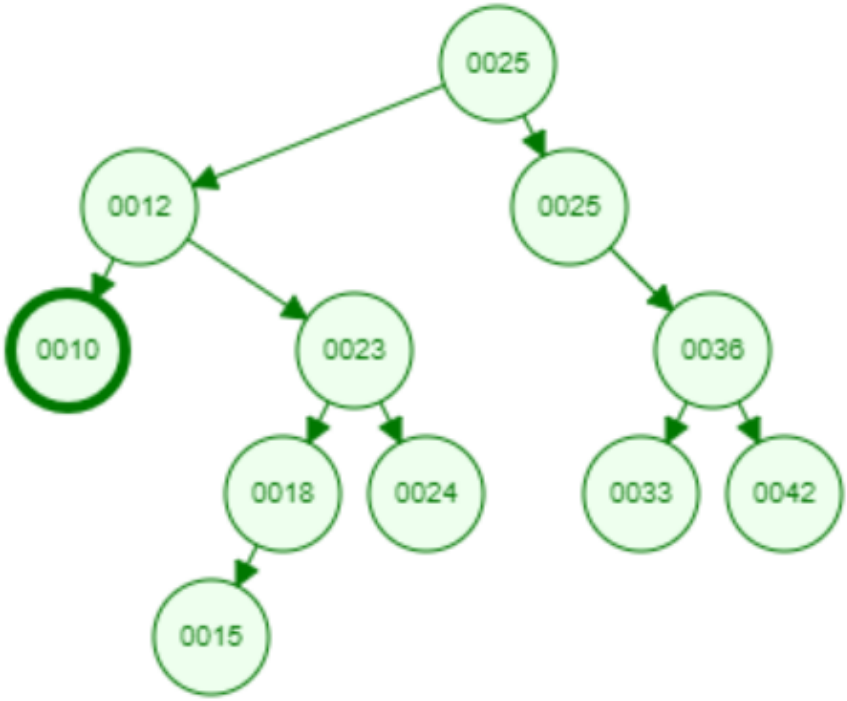


Imagen 23 *Seleccionar el Nodo a eliminar*

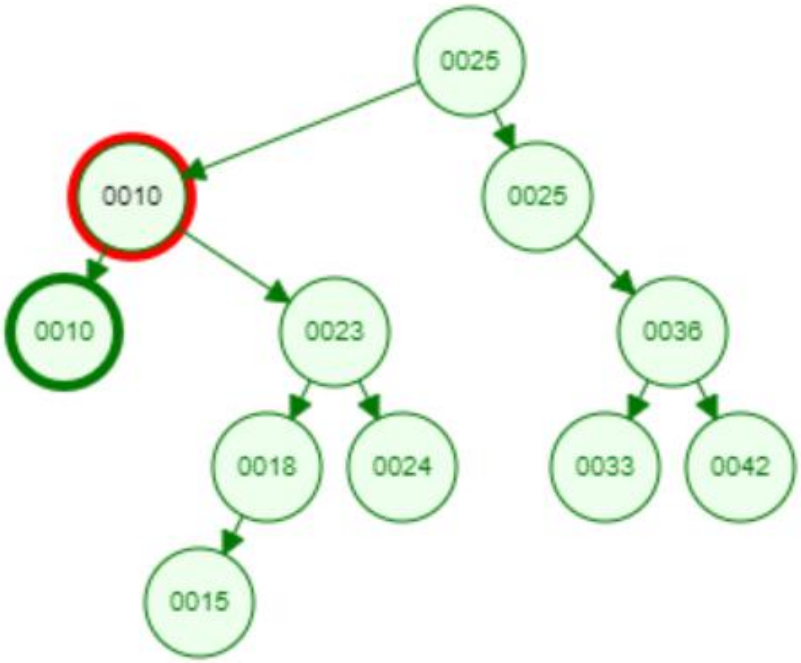


Imagen 24 Seleccionar el tipo de caso

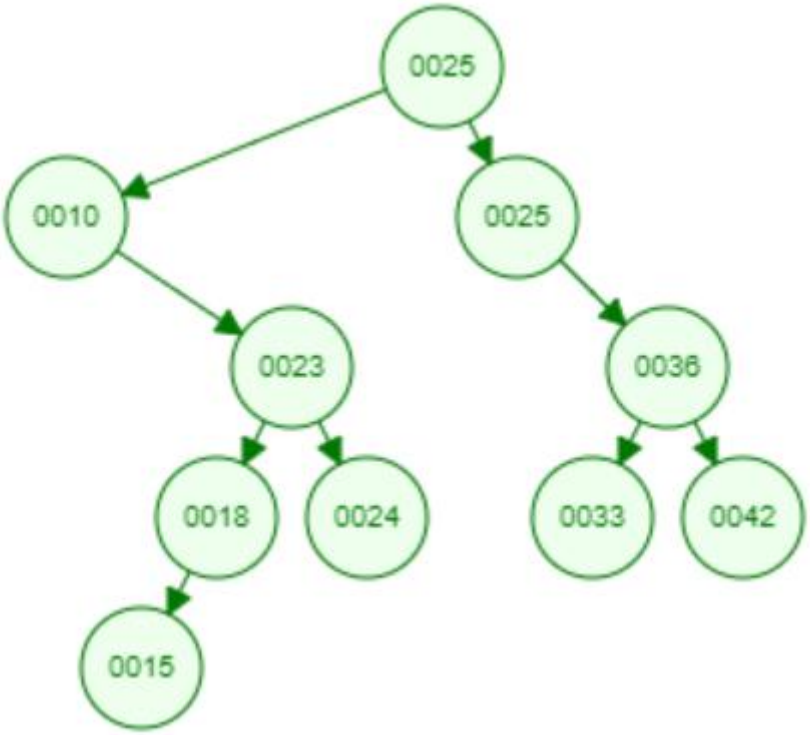


Imagen 25 *Nuevo Árbol*

- **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

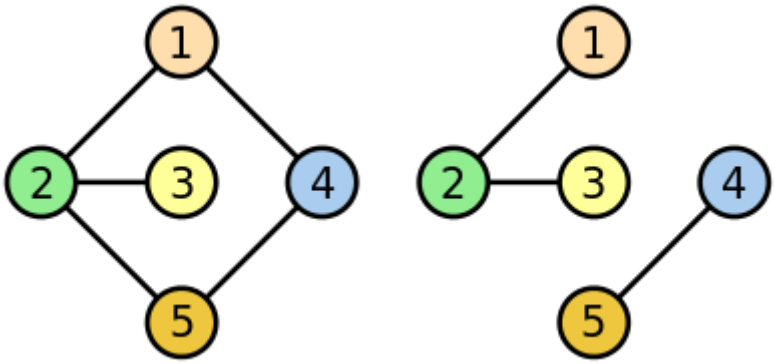


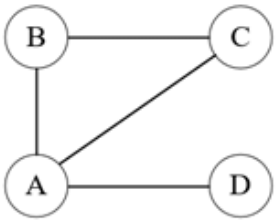
Figura 27: Representación de un grafo

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



	A	B	C	D
A	0	1	1	1
B	1	0	1	0
C	1	1	0	0
D	1	0	0	0

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

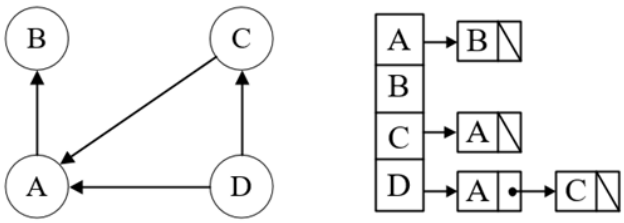


Figura29: Grafo con lista de adyacencia

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 (matiz 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"
<< std::endl;
    }
}

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.

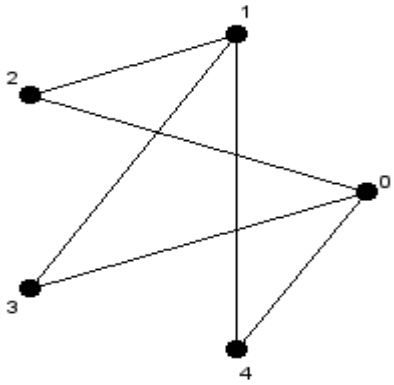


Imagen 30: Grafo Bipartido

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 V1 y V2, de tal manera que cada arista en E conecta un vértice en V1 con un vértice en V2.

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