UNIVERSIDAD DE
COSTA RICA

UNIVERSIDAD DE COSTA RICA

FACULTAD DE INGENIERÍA

ESCUELA DE CIENCIAS DE LA

COMPUTACION E INFORMÁTICA

Estructuras de Datos y Análisis de Algoritmos

CI0116

Grupo 003

Profesora: Sandra Kikut

II Tarea Programada

Elaborado por:

Esteban Quesada Quesada - B96157

Steven Nuñez Murillo - B95614

Sebastián González Varela - B93457

6 de diciembre del 2020

1. Introducción
2. Objetivos
3. Enunciado (Descripción del Problema)
4. Desarrollo
    4.1. Modelos
        4.1.1. Modelo 1
            4.1.1.1. Definición Grafo, no dirigido,con pesos,sin aristas paralelas,sin lazos
            4.1.1.2. Definición y especificación de operadores básicos del Modelo 1
        4.1.3. Modelo 2
            4.1.1.1. Definición Lista
            4.1.1.2. Definición y especificación de operadores básicos del Modelo 2
4.2. Estructura de Datos
        4.2.1. Estructura de Datos 1 (Lista Adyacencia)
            4.2.1.1. Diagrama y descripción E.D. 1
            4.2.1.2. Definición en C++ de la Estructura de Datos
        4.2.2. Estructura de Datos 2 (Matriz Adyacencia)
            4.2.1.1. Diagrama y descripción E.D. 2
            4.2.1.2. Definición en C++ de la Estructura de Datos
4.3. Algoritmos
5. Manual del Usuario
5.1. Requerimientos de Hardware
5.2. Requerimientos de Software
5.3. Arquitectura del programa
5.4. Compilación
5.5. Especificación de las funciones del programa
6. Datos de Prueba
6.1. Formato de los datos de prueba
6.2. Salida esperada
6.3. Salida obtenida (Análisis en caso de fallo)
7. Análisis de Algoritmos
7.1. Listado y justificación de modelos, operadores y estructuras de datos a analizar.
7.2. Casos de estudio, tipos de entrada, tamaños de entrada, diseño de experimentos.
7.3. Datos encontrados, tiempos y espacio presentados en tablas y gráficos.
7.4. Análisis de los datos.
7.5. Comparación de datos reales con los teóricos.
7.6. Conclusiones con respecto al análisis realizado.
8. Listado de Archivos (Estructura de las Carpetas)
9. Referencias o Bibliografía

Introducción:
En el presente trabajo se define, específica e implementa el modelo lógico grafo no dirigido, con pesos, sin aristas paralelas y sin lazos, además de esto se lleva a cabo un análisis de sus operadores. Asimismo, se presentan diferentes estructuras auxiliares, y dos diferentes implementaciones del grafo anteriormente mencionado, una de ellas utilizando la estructura de datos arreglo con matriz de adyacencia y la otra con una lista de adyacencia (L.S.E. y L.S.E)

Por otra parte, se desarrollan e implementan una serie de algoritmos capaces de funcionar con cada uno de los grafos que se implementaron. Existe un menú que es capaz de utilizar cada operador básico de los grafos implementados, así como cada uno de los algoritmos que se desarrollaron para los mismos. Posteriormente se encuentra un análisis empírico, el cual toma las 2 estructuras de grafos para realizar un estudio con 6 algoritmos y 2 operadores básicos . De esta forma se compara el funcionamiento de cada algoritmo según la estructura que se utilice.

Objetivos:
1. Definir, especificar, implementar y usar el modelo Grafo No Dirigido, con pesos, sin aristas paralelas y sin lazos.
2. Usar las estructuras de datos: Matriz de Adyacencia y Lista de Adyacencia (L.S.E. y L.S.E) para implementar el modelo Grafo No Dirigido, con pesos, sin aristas paralelas y sin lazos.
3.Implementar diferentes algoritmos para grafos no dirigidos con pesos, sin aristas paralelas y sin lazos,
4. Hacer un análisis de tiempos reales de ejecución para determinar cómo incide la escogencia de la estructura de datos utilizada para implementar el grafo, en el tiempo de ejecución de algunos algoritmos de grafos.

Enunciado:
Definir, especificar e implementar modelos lógicos. Asimismo realizar una serie de análisis teóricos y un análisis del tiempo real del tiempo de ejecución de diferentes estructuras de datos, operadores y algoritmos utilizados.

Modelos:

Modelo 1 Grafo No Dirigido, con pesos, sin aristas paralelas y sin lazos.

Un grafo no dirigido es un conjunto finito de vértices V de un conjunto finito de aristas A. La principal diferencia entre un grafo no dirigido y un grafo dirigido, es que en grafo no dirigido cada arista en A es un par no ordenado de vértices. El grafo se emplea en distintas disciplinas para modelar relaciones simétricas entre objetos, objetos representados por los vértices y su respectiva conexión está representado por una arista en caso de existir relación entre sí con su respectivo peso, no existen aristas entre un mismo vértice(lazos), ni aristas paralelas.

Definición y especificación de operadores básicos del modelo Grafo no dirigido, con pesos, sin aristas paralelas y sin lazos.

1. Iniciar()

    Parámetros: ninguno

    Efecto: inicializa el grafo G

    Requiere: grafo no inicializado o destruido

    Modifica: nada

2. Destruir(G)

    Parámetros: grafo G

    Efecto: destruye grafo G

    Requiere: grafo G inicializado

    Modifica: nada

3.Vaciar(G)

    Parámetros: grafo G

    Efecto: vacía el grafo G

    Requiere: grafo G inicializado

    Modifica: grafo G

4.Vacío(G)

    Parámetros:grafo G

    Efecto: devuelve falso o verdadero si el grafo G está vacío

    Requiere: grafo G inicializado

    Modifica:nada

5.AgregarVértice(G,e)

    Parámetros: e ->tipo elemento, G tipo grafo

    Efecto: agrega un vértice con el elemento e

Requiere: grafo G inicializado y elemento e válido

Modifica: grafo G

6.EliminarVértice(G,v)

Parámetros: v tipo vértice, G tipo grafo

Efecto: eliminada el vértice v del grafo G

Requiere:grafo G inicializado y vértice v válido

Modifica: grafo V

7.ModificarEtiqueta(G,v,e)

Parámetros: v tipo vértice, e tipo elemento, G tipo grafo

Efecto: cambia la etiqueta actual del vértice v por el elemento e

Requiere:grafo inicializado, vértice v y elemento e válidos

Modifica: vértice v

8.Etiqueta(G,v)

Parámetros: v tipo vértice, G tipo grafo

Efecto:devuelve la etiqueta actual del vértice v

Requiere: grafo G inicializado y vértice v válido

Modifica: nada

9.AgregarArista(G,v1,v2,p)

Parámetros: v1, v2 tipos vértices , p tipo peso, G tipo grafo

Efecto: agrega una arista a de peso p entre los vértices v1 y v2

Requiere: vértices v1,v2 válidos, peso p válido y que no exista una arista entre v1 y v2

Modifica: grafo G

10. EliminarArista(G,v1,v2)

Parámetros: v1,v2 tipo vértices, G tipo grafo

Efecto: elimina la arista a entre los vértices v1 y v2

Requiere: arista a entre v1 y v2 existente

Modifica: grafo G

11.ModificarPeso(G,v1,v2,p)

Parámetros: v1,v2 tipo vértices, p tipo peso, G tipo grafo

Efecto: modifica el peso p de la arista a entre v1 y v2

Requiere: arista entre v1 y v2 existente, vértices v1,v2 válidos,peso p válido

Modifica: arista a

12. Peso(G,v1,v2)

Parámetros: v1,v2 tipo vértices, G tipo grafo

Efecto: devuelve el peso p de la arista entre v1 y v2

Requiere: vértices v1,v2 válidos, arista existente entre v1 y v2

Modifica:nada

13.Primer Vértice(G)

Parámetros:grafo G

Efecto:devuelve el primer vértice del grafo G

Requiere: grafo G inicializado

Modifica: nada

14. SiguienteVértice(G,v)

Parámetros: v tipo vértice, G tipo grafo

Efecto:devuelve el siguiente vértice de v dentro del grafo G

Requiere: grafo G inicializado, vértice v válido

Modifica: nada

15. PrimerVérticeAdyacente(G, v)

Parámetros: v tipo vértice, G tipo grafo

Efecto: devuelve el primer vértice adyacente del vértice v

Requiere: vértice v válido

Modifica: nada

16.SiguienteVérticeAdyacente(G,v1,v2)

Parámetros: v1,v2 tipo vértices, G tipo grafo

Efecto:devuelve el siguiente vértice adyacente de v1, después de v2

Requiere: vértices v1,v2 válidos

Modifica: nada

17.ExisteArista(G,v1,v2)

Parámetros:v1,v2 tipo vértices, G tipo grafo

Efecto:devuelve verdadero o falso si existe la arista entre v1 y v2

Requiere: vértices v1,v2 válidos

Modifica: nada

18.NumAristas(G)

Parámetros:grafo G

Efecto:devuelve la cantidad total de aristas del grafo G

Requiere: grafo G inicializado

Modifica: nada

19.NumVértices(G)

Parámetros:grafo G

Efecto:devuelve la cantidad total de vértices del grafo G

Requiere:grafo G inicializado

Modifica: nada

20.NumVérticesAdyacentes(G, v)

Parámetros: v tipo vértice

Efecto: devuelve la cantidad de vértices adyacentes del vértice v

Requiere: vértice v válido

Modifica:nada

Modelo Lista

Lista: Se define como una sucesión de elementos que tienen entre sí una relación de procedencia. Pueden ser indexadas, es decir tienen un índice, posteriormente se encuentra la posicional, donde la posición es una noción abstracta y por último la de tipo de elementos .Listas constituyen una estructura flexible en particular, porque pueden crecer y acortarse según se requiera, los elementos son accesibles y se pueden insertar y suprimir en cualquier posición de la lista. Las listas también pueden concatenarse entre sí o dividirse en sublistas, se presentan de manera rutinaria en aplicaciones como recuperación de información, traducción de lenguajes de programación y simulación. Matemáticamente, una lista es una secuencia de cero o más elementos de un tipo determinado (por lo general se denominará de tipo elemento).

1.Iniciar(L):

Parámetros: Lista L

Efecto:inicializa una lista L como lista vacía

Requiere: lista L no inicializada o destruida

Modifica:Lista L

2.Destruir(L):

Parámetros: Lista L

Efecto: deja lista L inutilizable

Requiere:lista L inicializada

Modifica:lista L

3.Vaciar(L):

Parámetros: Lista L

Efecto:vacía lista L. No es necesario

Requiere:lista L inicializada

Modifica: lista L

4.Vacía(L)

Efecto:retorna verdadero si la lista L está vacía o falso si no, requiere que la lista esté inicializada.

Parámetros: Lista L

Requiere:lista L inicializada

Modifica:nada

5.Insertar(e,p,L)

Efecto: agrega un elemento e en la posición p de la lista L

Parámetros:elemento e,lista L

Requiere:lista L inicializada y elemento e y posición p válidos

Modifica: lista L

6. Borrar(e,L)

Parámetros: elemento e , lista L

Requiere:lista L inicializada y elemento e válido

Modifica: lista L

7. Recuperar(p,L)

Parámetros:posición p de la lista,lista L

Requiere:nodo n válido y lista L inicializada

Modifica:nada

8. Primera(L)

Efecto: devuelve la posición p de el primer elemento de la lista L

Parámetros:lista L

Requiere: L inicializada y primer elemento válido

Modifica: nada

10. AgregarAlFinal(e, L)

Efecto:agrega un elemento e al final de la lista L

Parámetros:elemento e,lista L

Requiere:lista L inicializada y elemento e válido

Modifica: lista L

11. Primera(L)

Efecto: Devuelve la primera posición de la lista L

Parámetros:lista L

Requiere: lista L inicializada

Modifica:nada

12. Última(L)

Efecto:Devuelve la última posición de la lista L

Parámetros:lista L

Requiere:lista L inicializada

Modifica:nada

13. Siguiente(p,L)

Efecto:Devuelve la siguiente posición de la posición p en la lista L

Parámetros:posición p,lista L

Requiere:lista L inicializada y posición p valida

Modifica:nada

14. Anterior(p,L)

Efecto:Devuelve la anterior posición de la posición p en la lista L

Parámetros:posición p,lista L

Requiere:lista L inicializada y posición p válida

Modifica:nada

15. Modificar:

Parámetros:p, e, L

Efecto: cambia el elemento e de la p por el elemento e de parámetro

Requiere: lista L inicializada, posición p y elemento e válidos

Modifica: elemento e de la lista L,

16. Intercambiar(p1,p2,L)

Parámetros:lista L, posiciones p1 y p2

Efecto:intercambia las posiciones p1,p2 de la lista L

Requiere:lista L inicializada y posiciones p1,p2 validas

Modifica:lista L

17. NumElem(L)

Parámetros:lista L

Efecto: devuelve la cantidad de elementos de la lista L

Requiere:lista L inicializada

Modifica:nada

Modelos Auxiliares implementados:

- Diccionario.
- Conjunto.
- Árbol parcialmente ordenado.
- Cola.

Estructura de Datos:

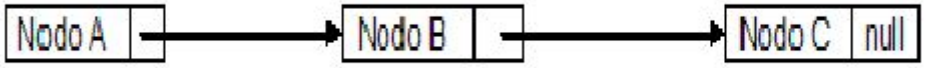
Estructura de datos Lista Simplemente Enlazada

Una lista enlazada simple es una colección de nodos que tienen una sola dirección y que en conjunto forman una estructura de datos lineal. Cada nodo es un objeto compuesto que guarda una referencia a un elemento (dato) y una referencia a otro nodo (dirección).

La referencia que guarda un nodo a otro nodo se puede considerar un enlace o un puntero hacia el segundo nodo y el salto que los relaciona recibe el nombre de salto de enlace o salto de puntero. El primer nodo de una lista recibe el nombre de cabeza, cabecera o primero y el último es llamado final, cola o último (es el único nodo con la referencia a otro objeto como nula).

Un nodo de una lista enlazada simple puede determinar quien se encuentra después de él pero no puede determinar quien se encuentra antes, ya que solo cuenta con la dirección del nodo siguiente pero no del anterior.

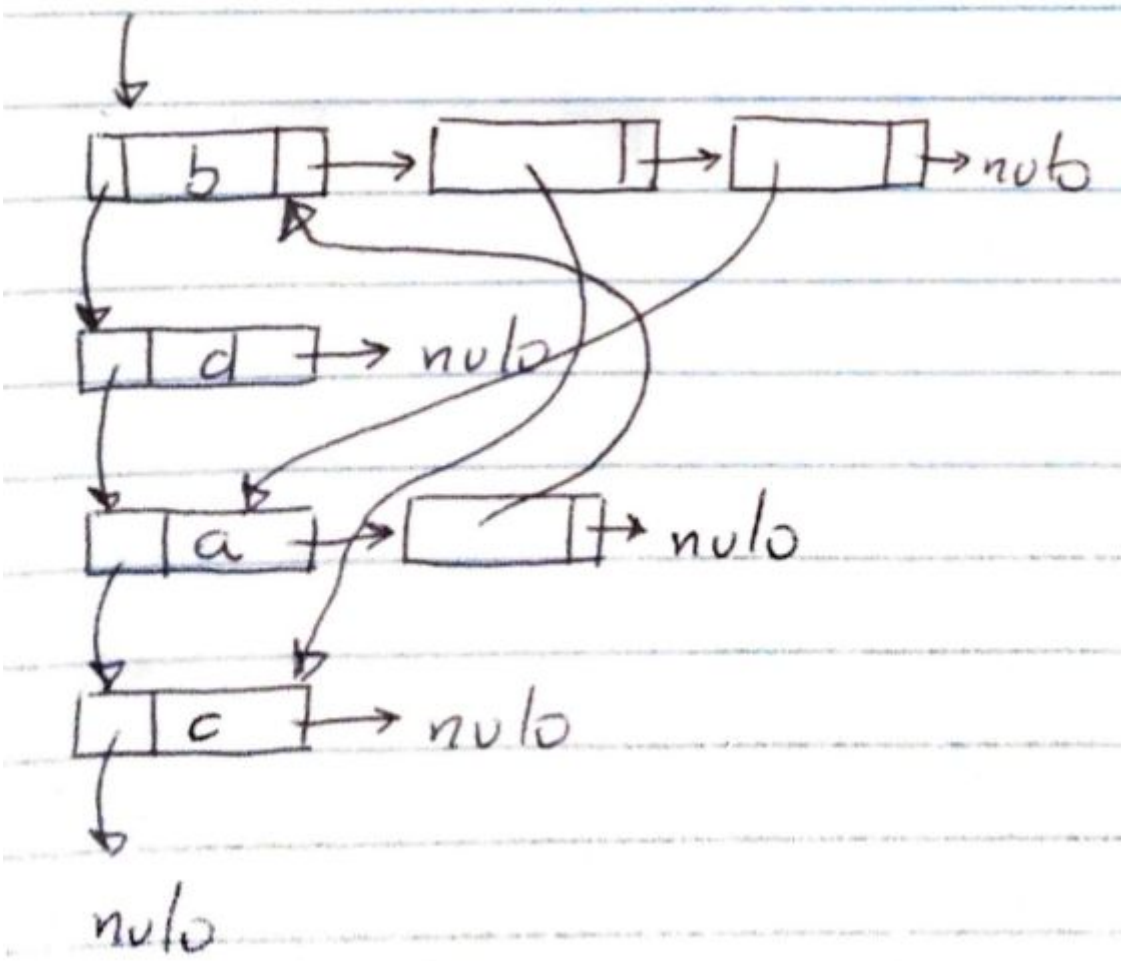
Cabeza final



Los elementos cabeza y final de la figura son referencia creadas a partir de la clase que se crean los nodos.

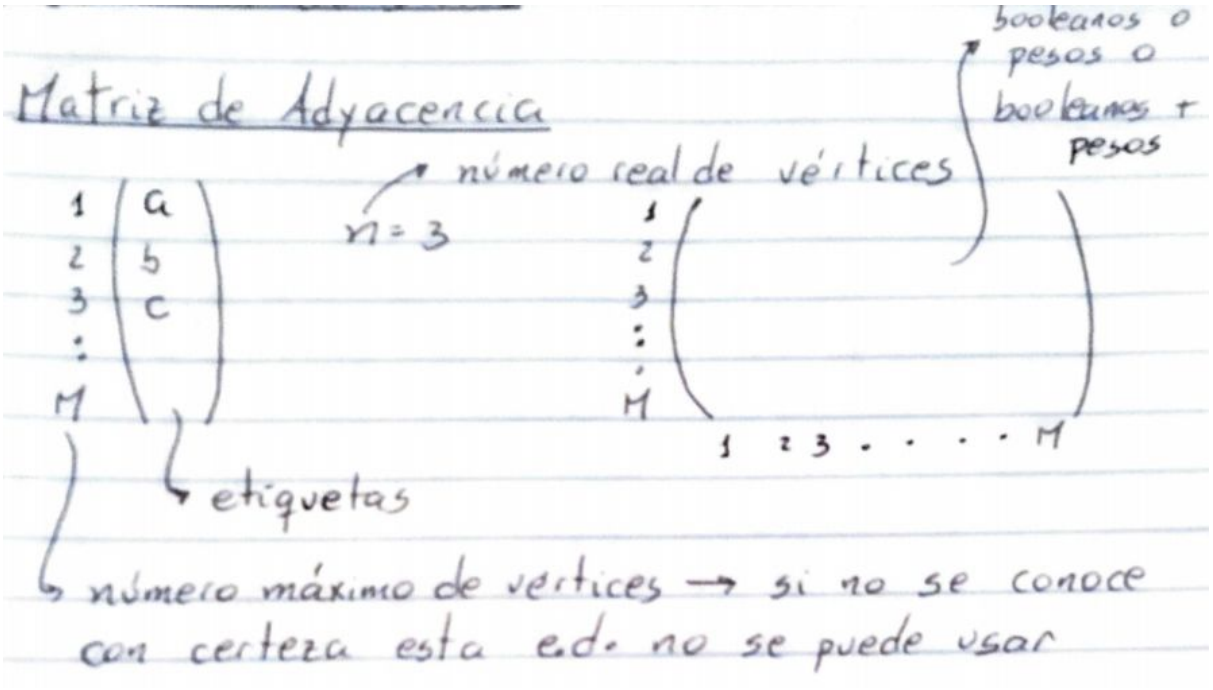
Lista de Adyacencia:

Una lista de adyacencia puede ser una lista simplemente enlazada,arreglo, lista doblemente enlazada, donde se hace una división entre la lista principal de elementos, y cada nodo de la lista tiene una sublista con punteros a los sus vértices adyacentes que se encuentran en la lista principal, los nodos pueden estar en cualquier orden, en las sublistas de vértices adyacentes, los vértices pueden estar en cualquier orden, el orden de los nodos y de los hijos depende del orden en el que se creó el árbol.



Matriz de adyacencia:

Un arreglo con es un conjunto de datos almacenados en memoria estática de manera continua donde los elemento se diferencian por el un índice. Los nodos están en cualquier orden en que el usuario ha creado el grafo lógico, esto debido a que no importa el orden entre vértices y a que el grafo se puede crear en cualquier orden. La estructura cuenta con un arreglo y una matriz representada con un arreglo de arreglos con los datos de las aristas del grafo,en esta estructura el tipo vértice que se guarda en el arreglo principal son chars y los que se almacena en la matriz son booleanos y pesos tipo peso.



Estructuras de datos auxiliares implementadas:

- Arreglo.
- Lista.
- Arreglo de arreglos.
- Vector Heap.

Algoritmos

Nombre Algoritmo	**Parámetros**	**Efecto**	**Requiere**	**Modifica**
Recorrido en Ancho Primero para averiguar si el grafo tiene ciclos	G (tipo grafo) v (tipo vértice)	devuelve verdadero si el grafo tiene ciclos o falso si no tiene ciclos	Grafo inicializado	N/A
Recorrido en Profundidad Primero para averiguar si el grafo tiene ciclos	G (tipo grafo) v (tipo vértice)	devuelve verdadero si el grafo tiene ciclos o falso si no tiene ciclos	Grafo inicializado	N/A

Recorrido en Profundidad Primero para averiguar si un grafo es conexo	G (tipo grafo)	devuelve verdadero si el grafo es conexo o falso si no es conexo	Grafo inicializado	N/A
Averiguar si un grafo es conexo usando un algoritmo basado en Warshall	G (tipo grafo) v (tipo vertice)	devuelve verdadero si el grafo es conexo o falso si no es conexo	Grafo inicializado	N/A
Dijkstra	G(tipo grafo) v (tipo vertice) pesos* tipo vector de ints vertices* tipo vector de int	devuelve el camino más corto a un vértice hacia todos los demás vértices del grafo	Grafo inicializado .	N/A
Floyd	G(tipo grafo) matrizdePesos* tipo vector de ints IntermediateVertices* tipo vector de int	Encuentra el camino más corto entre todo par de vértices	Grafo inicializado	N/A
Circuito de Hamilton de Menor Costo mediante una búsqueda exhaustiva pura	Etiqueta (tipo elemento) G (tipo grafo)	Crea una sucesión de aristas adyacentes sin repetir aristas,ni repetir vértices	Grafo inicializado	N/A
Colorear un grafo, usando la menor cantidad de colores posible, mediante una búsqueda exhaustiva pura	G (tipo grafo)	Asignarle una etiqueta color a cada vértice del grafo, con la mínima cantidad de colores posible	Grafo inicializado	N/A
Prim	G(tipo grafo) v (tipo vertice)	Encuentra el árbol recubridor mínimo en un grafo conexo	Grafo inicializado	N/A

Kruskal	G(tipo grafo) v (tipo vertice)	Encuentra el árbol recubridor mínimo en un grafo conexo	Grafo inicializado	N/A
Aislar un vértice	G (tipo grafo) v (tipo vertice)	Aísla el nodo n eliminando todas sus aristas	Grafo inicializado	Modifica al grafo G
Calcular la suma total de los pesos de todas las aristas	G(tipo grafo) v (tipo vértice)	Suma todos los pesos de las aristas del grafo G	Grafo inicializado .	N/A
Buscar una etiqueta	G(tipo grafo) e(tipo etiqueta)	Devuelve el vértice con la etiqueta e	Grafo inicializado Etiquetas no repetidas	N/A

Manual del Usuario

5.1 Requerimientos de Hardware:
Un ordenador capaz de procesar el programa, los ordenadores actuales, todos en su mayoría son capaces de procesarlo.

5.2. **Requerimientos de Software**

C++ instalado en su ordenador

Si se trabaja en linux solo se necesita tener instalado el compilador «g++» y un editor de texto como «kate». Si no es así, desde el terminal de linux se ejecuta la instrucción:

1. apt-get install g++
2. apt.get install kate

Para crear nuestro programa ejecutaremos el editor:

1. kate nombreprograma.cc

Para usar el compilador solo se debe usar la siguiente instrucción:

g++ nombreprograma.cc -o nombreprograma.x

Una vez compilado ejecutamos el programa usando la instrucción:

./nombreprograma.x

5.3. Arquitectura del programa

Se cuenta con distintas estructuras auxiliares las cuales son necesarias para realizar los diferentes algoritmos, puesto, que en esta entrega, algoritmos como Dijkstra, Kruskal Coloreo, y demás, necesitan de tales estructuras auxiliares. En la arquitectura del programa las definimos en .h, los cuales son apo.h, diccionario.h, vlister.h y conjunto.h.

El programa principal de los grafos está compuesto por una clase main la cual contiene todos los algoritmos de los grafos, asimismo, posee el menú o la clase prueba que utiliza el usuario para poder utilizar las diferentes estructuras de los grafos. Cada grafo está implementado en un .h aparte, es decir:

1. Grafo.h
2. GrafoLSE-LSE.h

5.4. Compilación

Para la ejecución de debe tomar en cuenta que aunque después del compilado se muestran mensaje de precaución pero esto no son más que advertencias por la implementación, después del compilado se puede llamar al ejecutable normalmente y funciona correctamente

1. Para compilar el programa se debe utilizar el comando:
   g++ -o prueba *.cpp
2. Para ejecutar el programa solamente en el cmd se escribe el nombre del ejecutable sin ningún parámetro, es decir: prueba
3. Si se desea utilizar un determinado grafo en el programa principal, es decir en el main.cpp, solamente se debe acceder a esta clase e incluir el grafo que se desea usar y comentar los demás grafos. Una vez se cambia en el .cpp se compila de nuevo el programa, y se ejecuta para que en el menú se esté usando el grafo correcto.

5.5. Especificación de las funciones del programa

- El programa cuenta con un menú principal con múltiples funciones para el grafo

Opción 0:Salir
Ingreso Datos: Ninguno
Salida: Se sale del programa y termina la ejecución
Restricciones:Ninguna

Opción 1:Iniciar Grafo
Ingreso Datos:Ninguno
Salida:Inicializa un árbol

Restricciones:el grafo no debe estar previamente inicializado

Opción 2:Destruir Grafo
Ingreso Datos: ninguno
Salida:se destruye el grafo
Restricciones:el grafo debe estar previamente inicializado

Opción 3:Vaciar Grafo
Ingreso Datos:ninguno
Salida: el grafo queda vacío,sin datos
Restricciones:el grafo, debe estar previamente inicializado

Opción 4:Preguntar Vacío
Ingreso Datos:ninguno
Salida: muestra un verdadero o falso
Restricciones:el grafo debe estar previamente inicializado

Opción 5:Agregar Hijo
Ingreso Datos: por consola se debe ingresar la etiqueta del vértice
Salida: ninguna
Restricciones: el grafo debe estar previamente inicializado

Opción 6:Eliminar Vértice
Ingreso Datos:Por consola se debe ingresar la etiqueta del vértice que se quiere eliminar
Salida: ninguna
Restricciones:El grafo debe estar previamente inicializado, y la etiqueta debe ser válida y debe haber un vértice con la etiqueta

Opción 7:Modificar Etiqueta
Ingreso Datos:Por consola se debe ingresar el vértice al que se le quiere cambiar la etiqueta y la etiqueta
Salida: nada
Restricciones:El grafo debe estar previamente inicializado y la etiqueta y el vértice deben ser válidos

Opción 8:Primer Vértice
Ingreso Datos: Por consola se debe ingresar la etiqueta del nodo que se desea borrar del árbol
Salida: regresa el primer vértice del árbol, nulo si no tiene un primer vértice
Restricciones:El grafo debe estar previamente inicializado

Opción 9:Agregar Arista
Ingreso Datos:Por consola se debe ingresar los vértices a los que se les quiere agregar una arista y el peso de esta arista
Salida:ninguna

Restricciones:El grafo debe estar previamente inicializado, y los vértices deben ser válidos ,igual que el peso,además la arista no debe existir

Opción 10:EliminarArista
Ingreso Datos:por consola se debe ingresar los vértices que tienen la arista que se quiere eliminar
Salida: ninguna
Restricciones:El grafo debe estar previamente inicializado y los vértices deben ser válidos y existir una arista entre estos vértices

Opción 11:Modificar Peso
Ingreso Datos:por consola se debe ingresar la los vértices que tienen la arista a la que se le quiere cambiar el peso y nuevo peso
Salida:ninguna
Restricciones:El grafo debe estar previamente inicializado y los vértices deben ser válidos y tener una arista existente entre estos vértices , el nuevo peso debe ser válido

Opción 12:Peso
Ingreso Datos:vértices que tienen una arista
Salida: peso de la arista
Restricciones:El grafo debe estar previamente inicializado

Opción 13:Existe Arista
Ingreso Datos:por consola se debe ingresar los vértices que se quiere saber si tienen una arista entre sí
Salida:verdadero si existe la arista o falso si n existe
Restricciones:El grafo debe estar previamente inicializado y los vértices deben ser válidos

Opción 14:Siguiente Vértice
Ingreso Datos: vértice que se desea saber el vértice siguiente
Salida: siguiente vértice en caso de existir y nulo si no existe
Restricciones:El grafo debe estar previamente inicializado y el vértice debe ser válido

Opción 15: Primer Vértice Adyacente
Ingreso Datos: por consola se debe ingresar el vértice que se quiere saber si tiene un primer vértice adyacente
Salida: devuelve un vértice si el vértice ingresado tiene vértices adyacentes después del vértice adyacente ingresado sino devuelve nulo
Restricciones:El grafo debe estar previamente inicializado y los vértices deben ser válidos y el segundo debe ser adyacente del primero

Opción 16:Siguiente Vértice Adyacente

Ingreso Datos:por consola se debe ingresar el vértice que se quiere saber si tiene un vértice adyacente y un vértice adyacente
Salida: devuelve un vértice si el vértice ingresado tiene vértices adyacentes después del vértice adyacente ingresado sino devuelve nulo
Restricciones:El grafo debe estar previamente inicializado y los vértices deben ser válidos y el segundo debe ser adyacente del primero

Opción 17:Núm Vértices
Ingreso Datos:ninguno
Salida: número de Vértices del grafo
Restricciones:El grafo debe estar previamente inicializado

Opción 18:Núm Vértices Adyacentes
Ingreso Datos: vértice que se quiere saber el número de vértices adyacentes
Salida: número de vértices adyacentes del vértice que se ingresa
Restricciones:El grafo debe estar previamente inicializado y el vértice debe ser válido

Opción 19:NumAristas
Ingreso Datos:ninguno
Salida: número de aristas del grafo
Restricciones:El grafo debe estar previamente inicializado

Opción 20: Etiqueta
Ingreso Datos: vértice que se quiere saber su etiqueta
Salida: etiqueta del vértice
Restricciones:El grafo debe estar previamente inicializado y el vértice debe ser válido

Opción 21:Es Adyacente
Ingreso Datos: vértices que se quiere saber si son adyacentes
Salida: verdadero si los vértices son adyacentes falso si no lo son
Restricciones:El grafo debe estar previamente inicializado y los vértices deben ser válidos

Opción 22:Es Vértice Nulo
Ingreso Datos: vértice que se quiere saber si existe dentro del grafo
Salida: verdadero si el vértice es nulo falso si no lo es
Restricciones:El grafo debe estar previamente inicializado

Opción 23:Menú de Algoritmos
Ingreso Datos:ninguno
Salida:despliega un submenú
Restricciones: ninguna

- Esta última opción despliega un submenú en el cual se tienen las siguientes opciones

Opción 0:Salir
Ingreso Datos:ninguno
Salida: Vuelve al menú principal
Restricciones: Ninguna

Opción 1: Tiene Ciclos
Ingreso Datos: ninguno
Salida: verdadero si el grafo tiene ciclos, falso si no tiene ciclos
Restricciones:Grafo previamente inicializado

Opción 2: Calcular Pesos
Ingreso Datos:ninguno
Salida: suma total del peso de las aristas del grafo
Restricciones:grafo previamente inicializado

Opción 3: Aislar Vértice
Ingreso Datos:por consola se debe ingresar el vértice que se quiere aislar
Salida:
Restricciones: grafo previamente inicializado y el vértice debe ser válido

Opción 4: Prim
Ingreso Datos:ninguno
Salida: muestra el árbol recubridor mínimo en el grafo
Restricciones: grafo previamente inicializado y conexo

Opción 5: Kruskal
Ingreso Datos:ninguno
Salida:muestra el árbol recubridor mínimo en el grafo
Restricciones: grafo previamente inicializado y conexo

Opción 6:Dijkstra
Ingreso Datos: vértice desde el cual se quiere calcular el camino más corto hacia los otros vértices
Salida: devuelve el camino más corto a un vértice hacia todos los demás vértices del grafo
Restricciones:grafo previamente inicializado y vértice válido

Opción 7:Circuito Hamilton
Ingreso Datos:ninguno
Salida: el camino más corto entre todos los vértices
Restricciones:grafo previamente inicializado

Opción 8:Floyd

Ingreso Datos: ninguno
Salida: el camino más corto entre todo par de vértices
Restricciones: grafo previamente inicializado

Opción 9: Colorear
Ingreso Datos: ninguno
Salida: vértices según el color respectivo que se le asignó en el algoritmo
Restricciones: grafo previamente inicializado

Opción 10:Warshall-Conexos
Ingreso Datos:ninguno
Salida: verdadero si el grafo es conexo, falso si no es conexo
Restricciones: grafo previamente inicializado

Opción 11:Profundidad Primero Ciclos
Ingreso Datos:ninguno
Salida: verdadero si el grafo tiene ciclos , falso si no tiene ciclos
Restricciones: árbol previamente inicializado

Opción 12:Profundidad Primero Conexos
Ingreso Datos: ninguno
Salida:verdadero si el grafo es conexo, falso si no es conexo
Restricciones:grafo previamente inicializado

6.1. Formato de los datos de prueba
Los datos de pruebas utilizados son chars para los vértices de los grafos y ints para los pesos de las aristas, se requiere que se sigan estos tipos de datos para que tanto los grafos,como sus operadores y los algoritmos sirvan de manera adecuada
6.2. Salida esperada
La salida esperada varía según el algoritmo, para algunos la salida esperada es un booleano, o un bosque abarcador,o un recorrido, los algoritmos deben de presentar la salida en pantalla, algunos operadores no tienen salidas y otros presentan salidas de tipo ints,chars,tipo vértices, y tipos pesos.
7. Análisis de Algoritmos
7.1. Listado y justificación de modelos, operadores y estructuras de datos a analizar.

Estructuras de datos:

Matriz de Adyacencia: Es una estructura de datos simple que utiliza un arreglo para representar los vértices y una matriz para representar las aristas y sus respectivos pesos, esta

estructura es muy simple de implementar y para varios operadores el tiempo de ejecución de O(1) por lo que es muy útil, Se puede determinar en un tiempo fijo y constante si un enlace(arco) pertenece o no al grafo.
Es fácil determinar si existe o no un arco o enlace, sólo se debe posicionar en la matriz.
Es fácil determinar si existe un ciclo en el grafo, basta multiplicar la matriz por ella misma n veces hasta obtener la matriz nula(no hay ciclos) o bien una sucesión periódica de matrices(hay ciclo)
Lista de Adyacencia: La lista de adyacencia requiere un espacio proporcional a la suma del número de vértices más el número de enlaces(aristas). Hace buen uso de la memoria. Se utiliza bastante cuando el número de enlaces es mucho menor que(n^2).
La representación con lista de adyacencia es que puede llevar un tiempo O(n) determinar si existe una arista del vértice i al vértice j, ya que pueden haber O(n) vértices en la lista de adyacencia. Para el vértice i.

Operadores:

- Agregar Vértice: este operador lo elegimos para el análisis dado que según la teoría para la estructura de datos matriz de adyacencia el tiempo es de O(n^2) pero para la lista de adyacencia tiene tiempo O(1) por lo que queremos verificar estos tiempos y confirmar la diferencia en los tiempos

- NumVertices: este operador lo elegimos para el análisis dado que según la teoría para la estructura de datos matriz de adyacencia el tiempo es de O(1) pero para la lista de adyacencia tiene tiempo O (N) por lo que queremos verificar estos tiempos y confirmar la diferencia en los tiempos

Modelos:
Grafo No Dirigido, con pesos, sin aristas paralelas y sin lazos.

7.2. Casos de estudio, tipos de entrada, tamaños de entrada, diseño de experimentos.
7.3. Datos encontrados, tiempos y espacio presentados en tablas y gráficos.

Caso de Estudio	Tipo de Entrada	Tamaños de Entrada	Diseño de Experimentos	Estructura de Datos utilizada	Tiempo de duración	Promedio de la Cantidad de tiempo entre el tamaño de entrada
Operador Agregar Vértice	Fueron agregados a los grafos elementos de tipo char.	Se realizaron 400 agregados para cada prueba.	Se realiza una función en el main que se encarga de procesar 400 veces la operación mencionada.	1)Matriz Adyacencia 2)Lista Adyacencia	1) 0.0001553s 2) 8.35e-005 s	1)0.0001553s / 400 2) 8.35e-005s / 400
Operador Num Vertices()	Fueron utilizados grafos de tipo char.	Se utilizó un grafo con 400 agregados.	Se realiza una función en el main que se encarga de procesar 1 veces la operación mencionada.	1)Matriz Adyacencia 2)Lista Adyacencia	1) 0.0001405s 2) 0.0011584s	1) 0.0001405s / 400 2) 0.0011584s / 400
Tiene Ciclos-Recorrido en	Fueron utilizados grafos de tipo char.	Se utilizó un grafo con 5 agregados de vértices y	Se realiza una función en el main que se	1)Matriz Adyacencia	1) 0.0061153	1) 0.0061153/5

ancho primero		con 7 aristas	encarga de procesar 1 veces la operación mencionada.	2)Lista Adyacencia	2) 0.0058332	2) 0.0058332/5
Averiguar si un grafo es conexo	Fueron utilizados grafos de tipo char.	Se utilizó un grafo con 5 agregados de vértices y con 7 aristas	Se realiza una función en el main que se encarga de procesar 1 veces la operación mencionada.	1)Matriz Adyacencia 2)Lista Adyacencia	1) 0.0186777 2) 0.0196952	1)0.0186777/5 2) 0.0196952/5
Prim	Fueron utilizados grafos de tipo char.	Se utilizó un grafo con 5 agregados de vértices y con 7 aristas	Se realiza una función en el main que se encarga de procesar 1 veces la operación mencionada.	1)Matriz Adyacencia 2)Lista Adyacencia	1) 0.0264786 2) 0.0301583	1)0.0264786/5 2)0.0301583/5
Floyd	Fueron utilizados grafos de tipo char.	Se utilizó un grafo con 5 agregados de vértices y con 7 aristas	Se realiza una función en el main que se encarga de procesar 1 veces la operación mencionada.	1)Matriz Adyacencia 2)Lista Adyacencia	1) 1.46e-005 2) 3.9e-006	1) 1.46e-005/5 2) 3.9e-006/5

Colorear	Fueron utilizados 3 grafos de tipo char.	a-)Se utilizó un grafo con 4 agregados de vértices y con 5 aristas, b-)Se utilizó un grafo con 5 agregados de vértices y con 7 aristas c-)Se utilizó un grafo con 6 agregados de vértices y con 9 aristas	Se realiza una función en el main que se encarga de procesar 1 veces la operación mencionada.	1)Matriz Adyacencia 2)Lista Adyacencia	1-a) 1.6e-006 1-b) 6.3e-006 1-c) 7.9e-006 2-a) 2.7e-005 2-b) 7.89e-005 2-c) 9.2e-006	1-a) 1.6e-006/4 1-b)6.3e-006/5 1-c) 7.9e-006/6 2-a) 2.7e-005/4 2-b) 7.89e-005/5 2-c) 9.2e-006/6
Hamilton	Fueron utilizados 3 grafos de tipo char.	a-)Se utilizó un grafo con 4 agregados de vértices y con 5 aristas, b-)Se utilizó un grafo con 5 agregados de vértices y con 7 aristas c-)Se utilizó un grafo con 6 agregados de vértices y con 9 aristas	Se realiza una función en el main que se encarga de procesar 1 veces la operación mencionada.	1)Matriz Adyacencia 2)Lista Adyacencia	1-a) 3.24e-005 1-b) 5.6e-006 1-c) 9.1e-006 2-a) 3.4e-006 2-b) 4.4e-006 2-c) 9.4e-006	1-a) 3.24e-005/4 1-b) 5.6e-006/5 1-c) 9.1e-006/6 2-a) 3.4e-006/4 2-b) 4.4e-006/5 2-c) 9.4e-006/6

7.4. Análisis de los datos.
7.5. Comparación de datos reales con los teóricos.
7.6. Conclusiones con respecto al análisis realizado.

Comparación, análisis y conclusiones de datos reales con los teóricos:

Operadores

Operador agregar vértice: Como se puede observar en las tablas de datos el tiempo de ejecución para este operador varía para las dos distintas estructuras de grafos, ya que en la estructura de datos matriz de adyacencia el tiempo es de O(n2) mientras que para la lista de adyacencia tiene tiempo O(1). Si comparamos esta información con los datos obtenidos y mostrados en la tabla logramos notar que efectivamente existe una diferencia al utilizar este operador en estas estructuras, sin embargo, debido a que los datos utilizados no tienen un tamaño tan considerable se puede notar un poco menos la comparación.
Como conclusión se puede observar como el operador Agregar Vértice es sumamente rápido y no gasta mucho tiempo a la hora de utilizarlo en un grafo implementado con la estructura de una lista de adyacencia y al utilizarlo con una matriz de adyacencia puede generar mayores costos ya que al hacer un agregado en la matriz este se tiene que hacer en la última posición siempre y además de esto agregar los espacios correspondientes.

Operador número Vértices: Como se puede observar en las tablas de datos este operador varía en su tiempo de ejecución, en la estructura de datos matriz de adyacencia el tiempo de ejecución fue el de menor tiempo, lo que concuerda con la teoría ya que el tiempo para esta estructura es O(1), en cambio para la estructura lista de adyacencia se tiene el mayor tiempo de ejecución, lo cual es válido porque esta posee un tiempo de ejecución de O(n), esto se debe a que para conocer la cantidad de elementos en una lista ocupa recorrer esta en cambio al trabajar en una matriz esta se maneja con índices por lo que se cuenta con un tope el cual corresponde a la cantidad de elementos.

Algoritmos

Tiene Ciclos-Recorrido en ancho primero: En este algoritmo los operadores básicos usados son Primer Vértice,Primer Vértice Adyacente, Peso,siguiente Vértice adyacente,siguiente Vértice, para el caso de los operadores básicos Siguiente Vértice y Primer Vértice, en la teoría para ambas estructuras de datos el orden de duración es igual O(1), por lo que esto no debería de representar cambios significativos en los tiempos de ejecución, en el caso del operador Peso, segun la teoria la estructura matriz de adyacencia tiene un orden de duración menor, contrario al operador primer vértice adyacente que según la teoría la estructura que tiene menor orden de duración es la lista de adyacencia, por lo que estos últimos dos operadores significan un cambio en los tiempo, además el último operador es Siguiente Vértice Adyacente el cual en ocasiones tiene menor orden de duración para la estructura matriz de adyacencia, por eso en la tabla los menores tiempos de ejecución son los de la estructura de datos matriz de adyacencia.

Averiguar si un grafo es conexo: Averiguar si un grafo es conexo es un algoritmo de tipo recursivo, esto implica que por lo general conlleva un mayor consumo del espacio, con respecto al tiempo al utilizar un grafo con cualquiera de las dos estructuras los tiempos son muy similares y esto se puede apreciar con facilidad en la tabla de tiempos mostrada anteriormente, la razón de esto es que la mayoría de operadores utilizado en esta implementación son O(1), a excepción de PrimerVerticeAdyacente() y SiguienteVerticeAdyacente() que son O(n), pero PrimerVerticeAdyacente() es O(n) en una matriz adyacente y O(1) en la otra, lo contrario de SiguienteVerticeAdyacente() que es O(n) en una lista de adyacencia y O(1) en la otra, en conclusión usar una implementación del grafo con cualquiera de las dos estructuras va a ser dependiente de las necesidades del programador en ese momento ya que los costos son muy similares.

Prim: En este algoritmo podemos observar según las tablas de tiempos que para las diferentes estructuras de datos los tiempo de ejecución varían muy poco, sin embargo al utilizar la estructura de lista de adyacencia se ve un aumento del tiempo lo cual si se analiza es justificable debido a que este utiliza diversos operadores básicos que aunque la mayoría tienen un orden de duración similar en ambas estructuras si hay una predominancia de una mayor duración de los utilizados al usar la lista de adyacencia como por ejemplo el operador Peso() con un orden O(min(N,A)), NumVertices con un orden O(N) o SteVerticeAdyacente() con un orden de duración de O(min(N,A)). Por lo tanto, ya sea que se utilice una matriz de adyacencia o la otra estructura este algoritmo sale con un costo muy similar y con velocidades semejantes, solo quedaría considerar que estructura se amolda más a la situación para cual se requiera el grafo.

Floyd: En este algoritmo los operadores básicos utilizados son Primer Vértice, Siguiente Vértice, Núm Vértices y Peso, en el caso de primer vértice según la teoría para ambas estructuras de datos el orden de duración es de O(1), por lo que este operador no debería de ocasionar una diferencia significativa en los tiempos de ejecución, esto mismo pasa con el operador basico Siguiente vértice, pero con los operadores Peso y Núm Vértices según la teoría para la estructura lista de adyacencia los órdenes de duración son O(n) y O min(n,a) respectivamente, en cambio con la estructura de datos matriz de adyacencia para ambos operadores, el orden de duración es de O(1), en las tablas de tiempos de ejecución vemos que los tiempos para la matriz de adyacencia son menores, y concluimos que la razón de esto es por el par de operadores básicos Peso y Núm Vértices

Colorear: Para ambas estructuras podemos notar que conforme se incrementa la cantidad de vértices como la cantidad de aristas el tiempo de ejecución aumenta, esto es lógico dado que el algoritmo tarda más en recorrer por completo el grafo, en este algoritmo los operadores que se utilizan son siguiente Vértice y Núm Vértices, en el caso de Siguiente Vértice para ambas estructuras el orden de duracion segun la teoria es de O(1) por lo que esto no debe significar cambio en los tiempo, pero el Num Vértices tiene una diferencia dado que según la teoría la matriz de adyacencia tiene un orden de duración O(1) , mientras que la lista de adyacencia tiene un orden de duracion O(n), por esto concluimos que la posible razón por la que los

tiempos de la estructura lista de adyacencia sean menores a los tiempos de la matriz de adyacencia es por este operador.

Hamilton: Para ambas estructuras podemos notar que conforme se incrementa la cantidad de vértices como la cantidad de aristas el tiempo de ejecución aumenta, esto es lógico dado que el algoritmo tarda más en recorrer por completo el grafo, en este algoritmo los operadores que se utilizan son PrimerVerticeAdyacente,Peso,NumVertices,SiguienteAdyacente
en el caso de Siguiente Vértice ambas estructuras tienen en la teoría un orden de duración Ó(n), por lo que esto no debería generar cambios significativos en los tiempos, en el caso de Primer Vértice Adyacente el orden de duración es mayor en el caso de la matriz de adyacencia, pero esto se ve contrarrestado por NumVertices y Peso que son mayores según la teoría el orden de duración son mayores para la estructura de datos lista de adyacencia, por esto concluimos que la diferencia entre los tiempos de ejecución está dada por estos factores, siendo mayores los tiempos de ejecución de la estructura de datos lista de adyacencia

8. Listado de Archivos (Estructura de las Carpetas)

Estructuras Auxiliares:

- apo.h
- conjunto.h
- diccionario.h
- Diccionario2.h
- vlister.h

Implementaciones de Grafos:

GrafoLSE-LSE:

- Nodo.h
- Nodo.cpp
- NodoLH.h
- NodoLH.cpp
- GrafoLSE.h
- GrafoLSE-LSE.h

GrafoMatrizdeAdyacencia:

- Grafo.h

Main:

main.cpp contiene todos los algoritmos de los grafos y el menú de uso.

9. Referencias o Bibliografía

AHO, V., ALFRED, H., JOHN, U., & JEFFREY. (1998). ESTRUCTURAS DE DATOS Y ALGORITMOS. MEXICO, D.F.: IMPRESIONES ALDINA, S.A.

Carl Reynolds & Paul Tymann (2008). *Schaum's Outline of Principles of Computer Science*. McGraw-Hill. ISBN 978-0-07-146051-4.

Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2009). *Introduction to algorithms*. Cambridge, Massachusetts: The MIT Press. ISBN 978-0-262-53305-8.

Hernández, R., Lázaro, J. C., Dormido, R., & Ros, S. (2001). Estructuras de datos y Algoritmos. Prentice Hall.