DOCUMENTACIÓN OBLIGATORIO

ESTRUCTURAS DE DATOS Y ALGORITMOS II

2024 – M4C

Sofía Fábrica – 303489
Gino Mazza – 303686

TABLA DE EJERCICIOS

PROBLEMA	RESULTADO
1	Completo.
2	Completo.
3	Completo.
4	Completo.
5	Completo.
6	Completo.
7	Completo.
8	Completo.
9	Completo.
10	Completo.

EJERCICIO 1

Utilizamos como base el código de la implementación de AVL visto en clase, modificándolo para cumplir con el ejercicio. Incluimos en "ejercicio1.cpp" el archivo "avl.cpp" para utilizar sus funciones.

El struct de este archivo tiene los datos necesarios para cada libro (id, título, estado) así como la altura, punteros de izquierda y derecha y constructor.

Definimos la cantidad total de libros y la cantidad de habilitados como variables de clase para cumplir con el orden constante en la consulta (count).

Usamos funciones auxiliares privadas para el funcionamiento de las funciones públicas add, find, toggle (no fue necesario para count gracias a las variables de clase). Nos basamos en la estructura de la función de inserción vista en clase, realizando cambios en las variables que se manejan para la parte de add.
El orden de las operaciones add, find y toggle es O(Log(n)) ya que para insertar y buscar en el árbol no lo recorremos completo, si no que solo recorremos la rama necesaria para la operación.

Además de las auxiliares tenemos funciones privadas para las rotaciones, balance y altura que mantienen correctamente la estructura AVL.

EJERCICIO 2

Incluimos en "ejercicio2.cpp" el archivo "hash.cpp" para utilizar sus funciones. Inicializamos en el main una tabla de hash de tamaño 11 ya que es un número primo (nos ayuda a reducir colisiones y a distribuir las claves más uniformemente).

Para el doble hash implementamos una primera función (que es el modulo de la clave con el tamaño de la tabla) y luego una segunda función diferente a ella (obtiene el número primo más grande que sea menor que el tamaño de la tabla y le resta el módulo de la id con él).

Para evitar errores decidimos hacer dos funciones para insertar: insertarAux que opera sobre la tabla global y actualiza contadores, y una específica para el momento de rehash insRehash que maneja una tabla nueva que luego reemplaza a la global, insertando los elementos directamente (los reorganiza con proximoVacio que utiliza la dos funciones de hash para conseguir el siguiente bucket vacío y distribuir uniformemente considerando las colisiones).

Para el find y toggle, en el caso recursivo usamos proximoParaRevisar, que revisa el siguiente bucket cuando el actual no tiene el ID que se busca (al usar las dos funciones de hash asegura que las colisiones se manejen correctamente).

Para las funciones esPrimo, primoMayorCercano y obtenerPrimoMenor consultamos a ChatGPT.

El orden de ejecución requerido para inserción, búsqueda y actualización de datos cumple ser O(1) cp por la tabla de hash. El de count es O(1) pc gracias a la variable global "habilitados" que permite orden constante al ser consultada.

EJERCICIO 3

Incluimos en “ejercicio3.cpp” el archivo “heap.cpp” para utilizar sus funciones.

Decidimos implementar un minHeap del tamaño de la cantidad de objetos en la tienda para poder ordenarlos todos por precio, utilizando sus funciones como comparar, intercambiar, flotar y hundir.

Para mostrar los k objetos más baratos en orden de precio ascendente, hicimos la función imprimirK que imprime los primeros k elementos del minHeap (los k nodos con precios más bajos) eliminándolos a medida que los imprime, asegurando que no se impriman objetos repetidos con el vector impresos.

Se cumple el orden de ejecución requerido, insertar N elementos en el heap tiene O(log N) pc, imprimirK tiene O(N logN) pc, ya que cada eliminación de las K (en peor caso K=N) tiene una complejidad de O(log N). Por regla de la suma O(log N) + O(N log N) = O(N log N).

EJERCICIO 4

Para este ejercicio implementamos una cola de prioridad extendida, compuesta por un minHeap y una tabla de hash, porque nos permite cambiar la prioridad de los elementos (al modificar el paraLlevar es más prioritario).

En el heap para flotar y hundir usamos una función auxiliar "comparar" que se encarga de definir si es necesario intercambiar siguiendo los criterios que dice la letra sobre prioridades, para llevar, id.

Al final, mostramos el elemento del peek y luego lo eliminamos para pasar al siguiente más prioritario y repetimos este proceso para todos los elementos que aún están en el heap.

Con respecto al orden de ejecución:
En el caso promedio, insertar un pedido en el heap tiene un costo constante O(1) y eliminar un pedido O (log N). Como se realizan N inserciones y eliminaciones, la complejidad total en este caso es O(N * log N).

En el peor caso las operaciones de inserción y eliminación tienen que recorrer toda la tabla hash, lo que resulta en O(N) para las dos. Si esto ocurre en las N operaciones de inserciones y eliminaciones, la complejidad sería O(N * N), es decir, O (N^2).

EJERCICIO 5

Incluimos en el archivo "ejercicio5.cpp", "grafoMisiones.cpp" (que incluye “grafoCiudades.cpp”) para utilizar sus funciones y decidimos implementar dos grafos: uno para las ciudades y otro para las misiones.

Primero, calculamos el orden topológico (OT) de las misiones para determinar el orden en que deben ejecutarse. La función OT utiliza la función calcularGrados, que calcula los grados de entrada de cada vértice (misión), es decir, cuántas misiones deben completarse antes de ejecutar una misión.
También usa una cola de prioridad extendida (CPE), que gestiona las misiones con grado de entrada cero (pueden ejecutarse de inmediato). Además usa Dijkstra para calcular los tiempos de viaje desde la ciudad inicial hasta las demás, así determinamos el orden de las misiones teniendo en cuenta las dependencias y los tiempos de desplazamiento.

Luego recorremos el vector resultante del OT. En cada iteración, aplicamos Dijkstra para calcular el tiempo de viaje mínimo desde la ciudad en la que se encuentra el equipo hasta la ciudad de destino de la misión actual. Los resultados se guardan en el vector tiempos, que contiene los tiempos de viaje más cortos desde la ciudad actual a cada ciudad destino. Durante la ejecución de Dijkstra se llena el vector vengoDe, que tiene información sobre de qué ciudad se viene para llegar a cada ciudad. Este vector nos ayuda a reconstruir el camino más corto desde la ciudad de destino hasta la ciudad de origen.

La función imprimirCamino se usa para reconstruir y mostrar el camino recorrido entre las ciudades recursivamente. Comienza desde la ciudad de destino, retrocede utilizando el vector vengoDe hasta llegar a la ciudad de origen, y luego imprime las ciudades en el orden correcto, desde la ciudad de origen hasta la ciudad de destino.

El orden de ejecución cumple con el requerido. El cálculo del orden topológico de las misiones involucra calcular los grados de entrada de cada misión (M en total) y gestionar sus dependencias (Dep en total) lo que correspone a O (M + Dep). La ejecución de Dijkstra para calcular los tiempos de desplazamiento entre ciudades, lo cual se repite para cada misión en el orden topológico (M en total), con el grafo de ciudades con C vértices y E aristas corresponde a M * ((C + E) * log C).

EJERCICIO 6

Para este ejercicio utilizamos el archivo "grafoCiudades.cpp". Decidimos implementar dos grafos, uno para obtener el mejor camino yendo primero hasta el equipo y el otro para ir primero hasta la entidad. Obtuvimos estos caminos guardando los vectores vengoDe (resultado de Dijkstra) que devuelve la función actualizarCostos (la cual también duplica el costo del camino con duplicarCamino).

También almacenamos el costo total de la ruta empezando por el equipo o por el team en variables que se actualizan en actualizarCostos. Luego comparándolas entre sí para quedarnos con la mejor opción y mostrar el mensaje correspondiente y recreando la ruta con los vengoDe guardados posteriormente.

Implementamos duplicarCamino en el grafo para actualizar los costos de las aristas usadas multiplicándolas por 2. Esto lo hacemos recorriendo el vector “vertices” en el bucket de uno los vértices de la arista hasta encontrar la arista correspondiente y duplicarle el costo, luego lo mismo con el otro vértice de la misma arista, de esta forma duplicamos el costo en un grafo no dirigido.

El orden de ejecución se cumple al utilizar Dijkstra para calcular las distancias mínimas desde cada una de las C ciudades. En cada ejecución, se procesan todas las ciudades (C) y sus conexiones (Con). Las operaciones extraer el nodo con menor distancia y actualizar distancias en la cola de prioridad (implementada como un heap), tienen un costo de log C. Al ejecutarse Dijkstra desde cada ciudad, el orden resulta en (C + Con) * log C.

EJERCICIO 7

Primero inicializamos un vector con las duraciones de cada canción, obtenemos la duración mínima que puede escuchar un estudiante (la canción más larga) y luego una duración total (la suma de todas las canciones). También inicializamos vectores con las posiciones en las que empieza y termina cada canción luego de repartirse (posicionesInicial y posicionesFinal respectivamente) y el tiempo total de escucha de cada uno (tiemposTotales).

Para la búsqueda binaria primero encontramos el medio entre la duración mínima y total encontradas anteriormente (tomamos las mismas como un intervalo y queremos que converjan, por esto el while hace su última iteración cuando los valores se igualan). Si podemos repartir las canciones en este intervalo sin la necesidad de que haya más estudiantes (verificado con la función repartirNoSuperaMax) entonces reemplazamos el resultado y acortamos el intervalo reduciendo el tiempo total (intentando buscar un tiempo menor). Si no podemos, aumentamos el intervalo al aumentar el tiempo máximo permitido.

repartirNoSuperaMax verifica si es posible asignar las canciones consecutivamente sin que el tiempo total que escuche ningún estudiante exceda tiempoMaximo. Recorre las canciones acumulando sus duraciones para cada estudiante y si se supera ese tiempo asigna las canciones acumuladas al estudiante actual y pasa al siguiente. Si necesita más estudiantes de los permitidos, retorna false. Si no, actualiza las posiciones y tiempos asignados a cada estudiante y retorna true.

Respecto al orden de ejecución, la búsqueda binaria va desde el valor de la canción con mayor duración hasta la suma total de las duraciones (T) siendo esto de orden O(log(T)). La función repartirNoSuperaMax en cada iteración de la búsqueda, recorre el vector de duraciones de todas las canciones (N en total) lo que resulta en O(N). Por lo tanto, se verifica el orden de ejecución O(log(T)N).

EJERCICIO 8

Para este ejercicio usamos como base el pseudocódigo visto en clase.

Primero ordenamos las ciudades por sus coordenadas en x ascendentemente (usamos mergesort ya que asegura O (N * log N)) y así en la función de dividir y conquistar, una mitad contiene las ciudades con coordenadas x más pequeñas y la otra con las más grandes.

Los casos base de la función mejorParDAC son cuando hay 2 ciudades (directamente se devuelve ese par) y cuando hay 3 (se devuelve el mejor comparando todas las distancias). Obtenemos el mejor par de la izquierda, el mejor de la derecha, comparamos distancias y nos quedamos con la más chica (menorDistancia).

Construimos una franja para considerar los pares cercanos a la frontera (los que podrían ser el mejor par) y así reducir comparaciones innecesarias. Esto lo hacemos con construirFranja que crea un nuevo arreglo incluyendo todas las ciudades cuya coordenada x está dentro de la distancia mínima desde la línea divisoria. Esto verifica si el mejor par de ciudades se encuentra cruzando la frontera entre las dos mitades.

En verificarFranja ordenamos las ciudades por y para hacer menos comparaciones (también con mergesort). Esto es porque dentro de la franja definida, las ciudades que están a una distancia chica en y son más probables a formar un par cercano. La condición del for evita que se comparen ciudades que están muy separadas en y porque no pueden tener una distancia menor que menorDistancia.

EJERCICIO 9

Para este ejercicio usamos como base la implementación de mochila de tres dimensiones vista en clase.

Modificamos el código implementando una mochila de cuatro dimensiones: cantidad de jugadores totales (1 a J), presupuesto total, cantidad de jugadores extranjeros permitidos, cantidad de jugadores máximos elegibles (1 a 11). Esta función termina con un cout del promedio de valoración, mostrando el resultado de dividir la máxima valoración obtenible (ubicada al final de la mochila) entre 11 (jugadores totales del equipo).

El código cumple con el orden requerido gracias a las dimensiones de la matriz de tabulación y el uso de programación dinámica. Descomponemos el problema en subproblemas almacenando las soluciones parciales en tab, y reutilizamos estas soluciones para construir la respuesta óptima (sin recalcular).

Para cada jugador actual (J veces), recorre todas las combinaciones posibles de presupuesto (P veces), número de extranjeros (E veces), y número de jugadores seleccionados (11 veces). Como 11 es constante, el orden queda O (J * P * E).

EJERCICIO 10

Para este ejercicio usamos como base el pseudocódigo de backtracking visto en clase y una estructura similar a la implementación de una mochila 0-INF.

Inicializamos una matriz que representa el estado actual del jardín, donde cada celda puede estar vacía o tener el color de una flor. Las flores se almacenan en un vector que tiene los colores y restricciones de filas específicas. Con esto y algunos datos llamamos a la función de backtracking floresBT.

En esta tenemos dos casos base: uno para cuando ya terminamos de recorrer toda la matriz (jardín), y otro para cuando no quedan suficientes casillas libres para superar la máxima cantidad de flores puestas almacenada. Este caso se controla con la función noPuedoSuperar, que hace una poda evaluando si las celdas restantes no son suficientes para mejorar la mejor solución.

También calculamos la siguiente celda con la función siguienteCelda, que actualiza las variables filaNueva y columnaNueva. Luego, recorremos todas las flores distintas con un for, probando colocar cada una en la celda actual.

Para colocar una flor verificamos que sea válida con esFlorValida y que cumpla las restricciones usando puedoPonerFlor. Si se puede, agregamos la flor al jardín con ponerFlor, actualizamos el contador de flores puestas y continuamos la búsqueda recursiva. Una vez terminada la rama, deshacemos el cambio con sacarFlor para explorar otras combinaciones. También exploramos la opción de no poner ninguna flor en la celda actual al terminar la recorrida del for.

Por último imprimimos la cantidad máxima de flores que se pueden colocar en el jardín respetando las restricciones.