Obligatorio 1
Estructuras de Datos y Algoritmos 2

2025

https://github.com/RominaCarsin/ObligatorioAlgoritmos

326152 - Romina Carsin

246326 - Luis Sanguinetti

Introduccion

Los ejercicios se resolvieron utilizando una estructura de main en el cpp ejercicio y el tad en un archivo separado, por ejemplo AVL para el ejercicio 1. En su mayoría se utilizaron clases ya que son son más sencillas de entender y tiene todas las funciones dentro de sí mismas en vez de ser separadas. También utilizamos como base en algunos los códigos dados en clase antes de adaptarlos para lo que realmente requería cada problema.

Ejercicio 1

Para el ejercicio 1 se solicitó utilizar árbol AVL como estructura principal. Para nuestra solucion 1 no era suficiente ya que solicitaban 2 sistemas de orden, uno por puntos (Rank) y otro por id del árbol (Find y Add).

Justificación de la complejidad temporal

ADD y FIND:

Add tiene que agregar a 2 árboles el de puntos y el de id uno para rank y el otro para find. Para hacer esto se creó una función add que llama a AddId y si es creado recién ahí corre el add points. Ambos de estos árboles se recorren en O(log N) para buscar el elemento del cual están ordenados (punto e id). Para agregar se recorren ambos árboles 1 vez el de puntos y el de id osea la complejidad temporal termina siendo O(log N + log N) porque ambos tienen el mismo y esto resulta en O(log N) para agregar en ambos.
Para el find es lo mismo recorrer solo un árbol esta vez moviéndonos por el id usuario en cada movimiento se reduce a la mitad porque elijo uno de los subárboles y esto resulta en O(log N).

RANK:

O(logN)
Rank es prácticamente lo mismo que add pero con un rango y no un punto fijo busca todos los elementos que cumplan con el punto y cada vez que da un paso compara y sabe para qué lado ir cortando el árbol en dos cada vez lo cual resulta en O(log N)

TOP1 y COUNT:

Pusimos un elemento del jugador que es top lo cual significa que dado un tipo jugador j podemos acceder al top1 en O(1) porque no se busca y se accede directamente.
Count es lo mismo tenemos la función cant que dado un tipo jugador j puede acceder a la cantidad en O(1).

Resultados De Prueba

Se corrieron con los scripts similares a los del practico para no tener que escribir todo cada vez aunque hay que cambiar el nombre del ejercicio a mano.
Se corre con bash ./scripts/compilation.sh y luego bash ./scripts/tests.sh
Ejercicio 1

Ejercicio2

Este ejercicio si se hizo con clase para la estructura central el cache pero para los nodos se utilizó struct ya que no contenían nada más que constructos y variables, por lo tanto no lo consideramos necesario hacer una clase. Se separo en 3 clase cache, struct DomainNode y struct HashNode. La clase contiene la tabla hash, tabla domain como elementos principales, estos son tipo domain node y hash node, el domain node es donde se guarda el domain el path título y las otras variables, el hash node es donde se guarda la relación de hash con el domain node también con path title entre otros. La razón para usar dos hash nodes y tablas es para cumplir con los requerimientos de complejidad bajo 2 diferentes formas.

Justificación de la complejidad temporal

Get, Contains, Count_Domain O(1)promedio:

Es promedio porque es un hash y asume que va a estar en la posición pero si no lo encuentra en la primera posición luego de ir a pos del hash tiene que recorrer hasta encontrarlo, por esta razón es O(1) en promedio y no para peor caso.

Size O(1) peor caso:

Utilice total count para retornar en O(1) como tiene el tamaño en la clase cache lo puede retornar en O(1).

Put, Remove, List_Domain, Clear_domain O(K):

Put: el hash es O(1) promedio; pero después tiene que recorrer los k elementos de ese dominio e insertar al final
Remove: encontrar el dominio es O(1) promedio, pero después tiene que buscar dentro de los elementos del dominio para eliminarlo.
Clear Domain: eliminar todos los elementos de un dominio, en hash O(1) promedio para encontrar su posición y luego O(K) para eliminar todos los elementos dentro

Clear O(M+N):

Usamos dos for para esto uno recorre la hashtable y el otro recorre la domain, por esta razón es M+N ya que es la suma de ambos largos;

Resultados De Prueba

Se corrieron con los scripts similares a los del practico para no tener que escribir todo cada vez aunque hay que cambiar el nombre del ejercicio a mano.
Se corre con bash ./scripts/compilation.sh y luego bash ./scripts/tests.sh
Ejercicio 2

Ejercicio 3

Similar al ejercicio 2 este es un hash pero la mayor diferencia viene en el tipo de hashing siendo este el abierto el mayor cambio es su manejo de colisiones, siendo esta agregar a una lista en vez de agregar a la siguiente posición disponible en el hash.

Justificación de la complejidad temporal

Get, Contains, Count_Domain O(1)promedio:

Get: Lo consigue en O(1) promedio porque lo busca consiguiendo su posición usando el hash y si tiene más de uno en el domain es lo mismo que en el otro tiene que buscar el correcto, pero en vez de hacerlo en el hash se usa el bucker (lista).
Contains: al igual que el get consigue en promedio O(1) la posición pero tiene que seguir buscando en el bucker (lista) al que realmente busca.
Count_Domain: funciona igual que los otros en complejidad corre a la posición con hash en O(1) promedio y si tiene dominio no eliminado retorna count de domNode.

Size O(1) peor caso:

Es O(1) peor caso porque su llamado lo único que hace es conseguir el totalcount de la clase cache

Put, Remove, List_Domain, Clear_domain O(K):

Put: Busca el dominio en O(1) promedio y luego recorre la lista para insertar, lo cual da O(K) porque siempre recorre la lista de dominio.
Remove: Busca domain en O(1) promedio y recorre la lista para remover que al ser la lista de domain es O(K);
Clear Domain: Busca el domain en O(1) promedio y luego recorre la lista eliminando todos los elementos resultando en O(K).

Clear O(M+N):

Elimina todos los elementos de ambos dominio y recursos lo cual require recorrer ambas tablas y por eso al recorrer ambas tablas es O(M+N).

Resultados De Prueba

```
Administrador@DESKTOP-4NTCO3K MINGW64 /e/Git/ObligatorioAlgoritmos (main)
$ bash ./scripts/tests.sh
./tests/ejercicio3/10.in.txt : 0 segundos | 113 milisegundos
./tests/ejercicio3/10.in.txt - OK
./tests/ejercicio3/especial.in.txt : 0 segundos | 78 milisegundos
./tests/ejercicio3/especial.in.txt - OK
./tests/ejercicio3/100.in.txt : 1 segundos | 78 milisegundos
./tests/ejercicio3/100.in.txt - OK
./tests/ejercicio3/1000.in.txt : 0 segundos | 83 milisegundos
./tests/ejercicio3/1000.in.txt - OK
./tests/ejercicio3/10000.in.txt : 0 segundos | 131 milisegundos
./tests/ejercicio3/10000.in.txt - OK
./tests/ejercicio3/100000.in.txt : 1 segundos | 920 milisegundos
./tests/ejercicio3/100000.in.txt - OK
./tests/ejercicio3/1000000.in.txt : 51 segundos | 51261 milisegundos
./tests/ejercicio3/1000000.in.txt - OK
```

Ejercicio 4

Usamos un min-heap (priority_queue mínima) que mantiene a lo sumo un elemento por lista: el actual “candidato” más chico de cada lista.

Inicialización: insertar en el heap el primer elemento de cada lista no vacía con una tupla (valor, idLista, idxEnLista).
Loop:
Extraer (pop) el mínimo del heap y imprimirlo (o agregarlo al resultado).
Si la lista de la cual salió ese elemento tiene un siguiente, insertar el siguiente en el heap.
Termina cuando el heap queda vacío.

Justificación de la complejidad temporal y espacial

Tiempo:

Hay N extracciones (una por cada elemento total) y hasta N inserciones.

Cada operación del heap cuesta O(log K) porque su tamaño máximo es K.

Por lo tanto, el tiempo total es $O(N \cdot \log K)$, que cumple la cota pedida.

Inicialización del heap: insertar K elementos. Con heapify es O(K) (o O(K log K) si se insertan uno a uno; en cualquier caso queda dominado por $N \cdot \log K$).

Espacio:

El heap guarda a lo sumo K elementos ⇒ O(K) espacio auxiliar.

Si se imprime a medida que se extrae, el resultado no se almacena y el extra se mantiene en O(K).

Resultados De Prueba

```
lucho@LuisLaptop MINGW64 ~/Documents/GitHub/ObligatorioAlgoritmos (main)
$ bash ./scripts/tests.sh
./tests/ejercicio4/10.in.txt : 1 segundos | 185 milisegundos
./tests/ejercicio4/10.in.txt - OK
./tests/ejercicio4/100.in.txt : 0 segundos | 108 milisegundos
./tests/ejercicio4/100.in.txt - OK
./tests/ejercicio4/1000.in.txt : 0 segundos | 85 milisegundos
./tests/ejercicio4/1000.in.txt - OK
./tests/ejercicio4/10000.in.txt : 0 segundos | 149 milisegundos
./tests/ejercicio4/10000.in.txt - OK
./tests/ejercicio4/100000.in.txt : 1 segundos | 202 milisegundos
./tests/ejercicio4/100000.in.txt - OK
./tests/ejercicio4/1000000.in.txt : 1 segundos | 1288 milisegundos
./tests/ejercicio4/1000000.in.txt - OK

lucho@LuisLaptop MINGW64 ~/Documents/GitHub/ObligatorioAlgoritmos (main)
```

Ejercicio 5

Usamos una estructura Arista para representar cada una de las conexiones entre vértices con su peso y las aristas se almacenaron dentro del grafo.
Ordena las aristas por su pero y posteriormente de forma creciente, va uniendo los vértices siempre y cuando no formen ciclos, sumando sus pesos al resultado final.
La estructura DisjointSet se utilizó para determinar de forma eficiente si dos vértices pertenecen al mismo conjunto (misma componente) y, si no, unirlos en uno solo.
Realizamos el algoritmo de Kruskal siguiendo los pasos dados en el curso:

1. Ordenar todas las aristas de menor a mayor según su peso utilizando quicksort
2. Iniciar el conjunto disjunto (DisjointSet) donde cada vértice es su propio representante.
3. Recorrer las aristas en orden creciente y, para cada una:
    a. Si los vértices u y v pertenecen a diferentes conjuntos, unirlos y sumar el peso al total.
    b. Si ya pasamos por una arista no se puede pasar nuevamente
4. Esto hasta pasar por todas las aristas o tener V-1 uniones
5. Se devuelve el peso total del arbol

Justificación de la complejidad temporal

Para ordenar las E aristas se implementó QuickSort, con una función de partición y una de intercambio (intercambiarAristas).

QuickSort tiene una complejidad promedio de O(E log E).

La clase DisjointSet inicializa los arreglos padre y rango en O(V) y utiliza las funciones:

buscar(x): aplica path compression, reduciendo el costo amortizado de búsqueda a $O(\alpha(V)) \approx O(1)$.

unir(a, b): aplica union by rank, uniendo el árbol de menor rango al de mayor, también en $O(\alpha(V)) \approx O(1)$.

Como el recorrido de todas las aristas implica ejecutar buscar y unir a lo sumo una vez por arista, este paso cuesta O(E) en total.

Por lo tanto, la complejidad temporal global es:
$O(ElogE+V+E)=O((V+E)logV)$

Resultados De Prueba