Trabajo Práctico Nº2

Informe

75.41 Algoritmos y Programación II
Cátedra Wachenchauzer
Cuatrimestre II, 2016

Agustín Manuel Del Torto Herranz. Padrón: 98867.

Víctor Marcelo Belosevich Mugerli. Padrón: 97757

Corrector: Matías Cano

Herramientas:

uniq-count

Esta herramienta itera sobre todas las lineas de un archivo. En cada iteración se separa las lineas en palabras(para esto se utiliza el split realizado en el tp1), las cuales son ingresadas de a una en un hash. Se utiliza este TDA debido a que las operaciones en el son en *O(1)*. Si una palabra ya se encuentra en el hash, solo se modifica su valor. A fin de que las palabras se muestren según el orden de aparición en el archivo se utiliza una cola. El orden de complejidad de esta herramienta es ***O(n*k)*** siendo n la cantidad de palabras y k la cantidad de lineas en el archivo pero como n generalmente es 2 ordenes mayor que k (la cantidad de lineas en promedio es menor que la de palabras), la complejidad realmente es ***O(n)*** .

comm

Esta aplicación hace provecho de las facilidades de la estructura de datos **hash,** específicamente el acceso a los datos en *O(1)*. Se recorren los archivos y se guarda cada línea en hashes independientes. Luego, se recorre uno de ellos con un iterador externo y se consulta, para cada elemento (línea de texto), si pertenece al otro hash. De ser así, se procede a guardar esta línea en un hash distinto (el cual almacenará todas las líneas en común) y se borra de los demás hashes. Acorde a los parámetros ingresados, se recorrerá el hash pertinente, también con iterador externo, y se imprimirá por pantalla.
El orden de complejidad de esta aplicación es de ***O(n)*** siendo n la cantidad de líneas del archivo más largo, pues sólo se hacen tres recorridos de ese orden.

Primitivas:

Árbol Binario de Búsqueda

abb_iter_post_order

Esta primitiva utiliza una pila para poder iterar de forma **post order** el árbol. En un principio el objetivo es apilar la raiz, los hijos izquierdos de la misma y todos los hijos izquierdos de los hijos derechos que no tiene hermanos (es decir, su padre no tiene hijo izquierdo) A esto se lo denomina traza izquierda.
Para avanzar, hay que desapilar y apilar la traza izquierda.
Para ver el actual, hay que ver el tope de la pila y para comprobar que el iterador está al final, solo hace falta ver si la pila está vacía.

abb_obtener_items

Esta primitiva utiliza, para lograr recorrer el árbol de manera **in order**, una **pila** como estructura auxiliar. Al ir apilando las "trazas izquierdas" del árbol y de los hijos derechos de los pertenecientes a estas trazas, se logra obtener los

elementos en el orden deseado y se guardan sus datos y claves en estructuras *abb_item_t*, y estas en arreglo a ser devuelto. Como se recorren todos los elementos del árbol, esta primitiva tiene una complejidad de orden ***O(n).***

Heap

top-k

Con la finalidad de devolver los k menores valores de un vector, esta función utiliza un Heap de máximos. Si el tamaño del vector es menor a k, se procede ocupando los valores desde k-n hasta k con NULL. Luego se llama a heap_crear_arr, la cual crea un heap de tamaño k. (o del tamaño del vector si este es menor a k). Una vez creado el heap, se itera sobre el, agregando nuevos valores y sacando el elemento más grande, para que siempre haya k elementos en el heap (esto tiene una complejidad de *O(log(k))* y *O(1)* respectivamente). Finalmente se procede a desencolar todos los elementos del heap de manera que el vector resultante quede ordenado de forma ascendente.
La complejidad de la función es ***O(nlog(k))*** ya que cuando se itera sobre el heap se recorren todos los elementos del vector (n) y como se menciono antes agregar un elemento a un heap es *O(log(de la cantidad de elementos))*, en este caso, el heap siempre tiene k elementos.

heap_actualizar_prioridad

Para encontrar el elemento que el usuario indica que fue alterado, se itera sobre el heap, comparando punteros. Luego de haber dado con el elemento correcto, se llama tanto a **upheap** como a **downheap** en esa posición para que el apropiado de ellos posicione al elemento en la posición debida.
Si bien tanto upheap como downheap son de orden *O(log(n))*, al tener que recorrer todo el heap para encontrar el elemento, esta primitiva tiene orden ***O(n).***

Para que la primitiva tuviera orden sublineal *(O(log(n))*, se debería tener el heap implementado con un árbol binario de búsqueda, en el que se guarden y actualicen las claves cada vez que se encole y desencole en el heap. Así, en *heap_actualizar_prioridad* se podría acceder al elemento en *O(log(n))*, reduciendo la complejidad del algoritmo.