UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA
Año 2016 - 2do Cuatrimestre

ALGORITMOS Y PROGRAMACIÓN II (95.15)

Trabajo práctico Nº 2

Tema: Utilidades con hashes, ABBs y heaps

Nombre: Klaus Lungwitz Padrón: 96416

Nombre: Ramiro Santos Padrón: 99772

Ayudante asignado: Martín Buchwald

Uniq-count

Para esta utilidad se pensó usar un hash, ya que con esta estructura es posible acceder a una clave en O(1). Esto es útil por varias operaciones que vamos a tener que hacer al ir guardando en el hash las palabras encontradas en el archivo. Primero que nada, todo el tiempo se estará verificando si la clave pertenece, es una operación de búsqueda. Esto es muy importante destacar, ya que en otras estructuras como listas o arreglos no es posible realizar esta operación en O(1), sino que se tiende a O(n) en el peor caso. Si no pertence, se guarda con un contador como valor (inicializado en 1), por lo que se realiza una operación de orden O(1). Si pertenece, se obtiene el valor (el contador) (otra operación de orden O(1)) y se lo incrementa en 1.

El orden del programa es la suma de varias operaciones. Se va leyendo de a una línea por vez con getline, por lo que la lectura del archivo se realiza en O(n). Luego de leerse cada línea, se separan las palabras usando la función split, y luego se recorre el arreglo de cadenas devuelto por split, por lo que, como se ve, en realidad en cada lectura se lee tres veces la entrada, resultando en O(3n). Como ya se detalló en el primer párrafo, en cada lectura también se realizan operaciones O(1) que no aumentan el orden hasta aquí. Al final de todo se obtienen y se imprimen todas las claves, que en el peor caso se tarda O(n). Por lo tanto, el orden total del programa es O(4n) (más varias operaciones O(1)), que resulta ser O(n).

Una estructura de datos adicional que fue utilizada fue una lista. Se usó para almacenar las palabras en el orden en el que van apareciendo en el texto, para luego poder imprimirlas en ese mismo orden. Y una biblioteca adicional utilizada fue strutil, hecha en el TP anterior.

Comm

A la hora de empezar con este ejercicio, es necesario tener en claro los dos casos que se presentan. Por un lado, si el usuario ingresa -1 o -2 (se deberá imprimir las líneas únicas del archivo que corresponda) o si no ingresa parámetro (se deberán ingresar las líneas que coincidan). Para ello se utilizará un hash, en donde se guardarán primero que nada todas las líneas del primer archivo recibido por parámetro (se guardarán como claves, el valor de ellas es indiferente ya que no se usará). Luego, para el primer caso mencionado, se recorrerá el segundo archivo y se verificará que no sus líneas no se encuentren en el hash, es decir, que no se encuentren repetidas en el primer archivo, y en caso de que así suceda se imprimirá la línea y luego se la agregará al hash (teniéndola ahora como ya incluida). Ya sea que el usuario ingrese -1 o -2 lo único que cambiará eso es que archivo se pone primero por parámetro y en consecuencia cuál segundo. Para el segundo caso, al igual que antes, se llenará el hash con las líneas del primer archivo, y, a diferencia de antes, se verificará que cada línea del segundo archivo sí esté. En caso que sea verdadero, la línea se imprimirá y se borrará del hash para evitar imprimirla de vuelta. En la función main (donde se llamará a comm) se deberá tener en cuenta al igual que en el ejercicio uniq-count, que, al utilizarse archivos, se deberá corroborar que no haya ningún error ya sea de lectura de archivos como de cierre.
Teniendo en cuenta que se recorrerá una vez el primer archivo pasado por parámetro (para meter sus líneas en el hash como clave) y luego una vez el segundo archivo, verificando que sus líneas están o no el hash según lo que se requiera hacer, el algoritmo será de orden n+k , siendo n la longitud del primer archivo y k la del segundo. En caso de tener longitudes similares será 2n, es decir n (el 2 es despreciable). Si k llegase a ser mucho más grande que n, el orden del algoritmo sería k (o n en el caso contrario).

Iterador post order

Para realizar este iterador de modo externo se tendrán en cuenta los tips dados en clase:
- al crear el iterador debo apilar la raíz y la traza izquierda (borde izquierdo del árbol)
- al avanzar debo desapilar y apilar la traza izquierda si y solo sí el desapilado es hijo izquierdo del nuevo tope.

Luego el resto es como el iterador in order: para ver el actual se verá el tope de la pila y para saber si el iterador está al final se verificará que la pila está vacía.
Con el iterador interno se prodecedrá igual que con el iterador in order, salvo que, obviamente, se modificará el orden en que se recorre el árbol, es decir que primero se irá recorriendo recursivamente los hijos izquierdos, luego los derechos y finalmente las raíces en cada caso (aplicándo la función de visitar).
El orden, al igual que en el iter in order nuevamente, será de n, ya que se recorren todos los nodos del árbol, siendo entonces n la cantidad de nodos.

abb_obtener_items

Esta primitiva del abb se realiza en orden O(n) ya que se recorre una vez el abb entero usando el iterador externo. En cada iteración se asignan a punteros la clave y el valor de cada item, devolviéndose un arreglo de estos items al usuario.

Como el enunciado pide que el arreglo de items esté ordenado por clave, se utiliza el iterador in-order, ya que el mismo recorre el abb en orden ascendente según la clave.

Top-k

Para poder realizar este ejercicio cumpliendo el orden pedido, esto es, n log k (siendo n la cantidad de elementos del vector recibido y k la cantidad de elementos que tendrá el arreglo a devolver) se podría hacer lo siguiente: ya que se piden los k números menores, y el orden debe ser n log k, se usará un heap en el que se guardarán k números. Para ello, se ha pensado recorrer los primeros k elementos del vector recibido y ubicarlos en el heap (este heap deberá ser de máximos, ahora se verá el porqué) . Después, lo indicado sería terminar de recorrer el arreglo e ir fijándose por cada elemento si es menor al máximo del heap, que se puede obtener viendo el tope del heap, puesto que es de máximos como se mencionó antes, y, en caso de ser menor que el tope del heap, desencolar el tope y encolar el nuevo elemento. Lo bueno es que al usar el heap, al incluir el nuevo elemento todo se reordenará y se obtendrá de nuevo el número de mayor valor en el heap al revisar el tope. Este procedimiento se repetirá hasta recorrer todo el arreglo, por lo que se tendrá orden n aquí y log k por lo mencionado anteriormente sobre el heap. Finalmente, y no hay que olvidar, es que se deberá ubicar en un nuevo arreglo (ara el cual se pedirá memoria y así se podrá devolver) todos los elementos que queden en el heap luego de recorrer todo el arreglo. Esto agregará al orden un +k de esta manera: n log + k, pero esto no modifica el orden, de modo que quedaría igualmente n log k.

heap_actualizar_prioridad

Esta primitiva trabaja en orden O(n log n). Esto es una suma de las siguientes operaciones.

Primero se busca el dato indicado en el heap, esta operación en el peor caso tarda O(n).

Si se encuentra el dato, se lo compara con su padre y sus hijos para descifrar cómo ordenarlo en el heap. Si el dato es más chico que alguno de sus hijos, se hace downheap para acomodar el dato (O(log n)). Por otro lado, si el dato es más grande que su padre, se hace upheap para acomodar el dato (O(log n)). Por lo tanto la primitiva tarda O(n log n), pero es importante observar que, una vez encontrado el dato, el orden de las operaciones siguientes para acomodar el dato baja a O(log n).