**Informe TP2**

**75.41 Algoritmos y Programación II**

**Cátedra Rosa Wachenchauzer**

**Cuatrimestre II, 2016**

**Corrector: Javier Choque**

**Jamilis Netanel David – Padrón: 99093**

**Sinisi Fernando – Padrón: 99139**

**Análisis y diseño de la solución:**

**Dificultades**

Luego de leer el enunciado del trabajo práctico nos encontramos con las

Siguientes dificultades:

En la función uniq\_count:

* Leer el archivo de texto.
* Reconocer la palabra que se esta leyendo.
* Guardar la frecuencia con la que aparecen las palabras.

En la función comm:

* Leer los archivos.
* Tener una referencia de que líneas pertenecen a cada archivo.

En el iterador post order:

* Implementar el iterador interno y externo post order.

En la primitiva abb obtener ítems:

* Devolver los datos del árbol ordenados por clave.

En la función top k:

* Tener en cuenta el caso en que k sea mayor a la cantidad de elementos del arreglo
* Ordenar los elementos de menor a mayor.
* Mantener el orden de O(nlogk).

En la primitiva heap actualizar prioridad:

* Obtener la posición del dato en el heap en orden lineal
* Ordenarlo con respecto a su padre y sus hijos (si los tiene)

**Resolución:**

Analizando la función uniq count:

Comienzo abriendo el archivo, leo la línea actual, guardo e un arreglo las palabras que contiene la línea. Por cada palabra en el arreglo, si la palabra actual no pertenece al hash, la guardo en el hash y también la guardo en una lista que va a contener las palabras (para luego saber en qué orden fueron apareciendo). Luego paso a leer la próxima línea.

Cuando se terminó de leer todo el archivo. Comienzo a leer el arreglo que contiene todas las palabras que fueron apareciendo, por cada palabra en el arreglo la busco en el hash para ver cuántas veces apareció e imprimo la palabra con la cantidad de apariciones.

El orden de este algoritmo es O(n) siendo n la cantidad de palabras del arreglo

Analizando la función comm:

Para comenzar me fijo que la cantidad de parámetros recibidos sea la correcta y que estos sean válidos. Luego abro ambos archivos, creo dos estructuras hash y guardo las líneas de ambos archivos en sus respectivos hash. Creo iteradores para ambos hash.

En el caso en que la cantidad de parámetros sea dos lo que hago es que mientras el iterador del hash con las palabras del primer archivo no este al final, veo la palabra actual de ese iterador, si esa palabra pertenece al hash con las palabras del segundo archivo imprimo esa palabra, luego avanzo el iterador para consultar por la siguiente palabra.

En el caso en que exista el tercer parámetro el algoritmo para decidir si imprimir esa palabra o no es similar al que se implemento en el caso en que este parámetro no existía. Se recibe un iterador y un hash, mientras el iterador no este al final, se analiza la palabra actual del iterador, en este caso para imprimirse la palabra esta no debe pertenecer al hash. Luego se avanza el iterador.

Entonces si el tercer parámetro es un -1 se pasa a la función de imprimir las líneas el iterador del primer hash y el segundo hash.

En el caso en que el tercer parámetro sea un -2 se pasa a la función de imprimir las líneas el iterador del segundo hash y el primer hash.

El orden de este algoritmo es O(n+k) siendo n la cantidad de líneas del primer archivo y k la cantidad de líneas del segundo archivo.

Analizando el iterador post order:

Comienzo analizando el iterador interno:

Comienzo llamando a una función la cual va a realizar los llamados recursivos, a esta función le paso la raíz del árbol, la función visitar y el extra que me pasaron por parámetro.

En este función lo que hago es decir que la raíz que me pasaron es el nodo actual, llamo recursivamente a esta función con el hijo izquierdo y con el hijo derecho, si alguno de estos llamados devuelve falso, devuelvo falso. Sino devuelvo lo que me devuelve la función visitar cuando la llamo con la clave, el dato (del nodo actual) y el valor extra.

El orden de este iterador es O(n) porque a lo sumo recorre el árbol completo, si la función visitar no corta la iteración antes.

Ahora analizo el iterador externo:

Dentro del iterador externo se encuentran las primitivas:

El constructor del iterador post order:

Esta primitiva te devuelve un elemento del tipo de dato iterador post order. Lo que hace es crear una variable del tipo de dato del iterador, a la cual le asigna una pila en la que apiló todos los nodos izquierdos del árbol y también le asigna el árbol. Devuelve el iterador.

Es O(logn) porque recorre una rama del árbol para insertar todos los hijos izquierdos a la raíz en la pila.

Esta al final?

Esta primitiva te devuelve si el iterador está al final, esto se cumple en el caso en que la pila este vacía. En ese caso devuelve verdadero. De no estar vacía devuelve falso.

Es O(1) evalúa solo una condición, si la pila esta vacía o no.

Ver actual:

En el caso de que la pila este vacía, de estar al final, te devuelve NULL. Sino te devuelve la clave del nodo que se encuentra en el tope de la pila.

Es O(1) evalúa una sola condición y realiza operaciones de O(1).

Avanzar:

Te devuelve si se puede avanzar dentro del árbol o no. En el caso en que la pila este vacia, estar al final del árbol, te devuelve falso. Si luego de desapilar el elemento del tope, al cual voy a llamar actual,, la pila se encuentra vacía, te devuelve verdadero. En el caso de no estar vacia la pila, obtiene el tope de esta y llamo a este padre.

Si el actual es el hijo izquierdo del padre, si existe un hijo derecho del padre, llamo a este actual y lo apilo a la pila del iterador. Mientras exista el hijo izquierdo del actual, mi actual pasa a ser el hijo izquierdo de este y lo apilo a la pila.

Es O(logn) porque recorre el árbol en una dirección y va apilando los nodos en la pila.

Destruir:

Destruye la pila del iterador y luego destruye el iterador.

Es O(k) siendo k la cantidad de elementos en la pila.

Analizando la primitiva abb obtener items:

Comienzo creando el tipo de dato que voy a devolver, este se llama abb ítem, cuenta con una clave y un valor. También para esta primitiva se debe crear un nodo el cual contenga un arreglo y una posición.

Ahora ya se puede empezar a crear la primitiva. Comenzamos creando un arreglo del tipo abb ítem con tamaño para la cantidad de nodos del árbol. Al mismo tiempo creamos un nodo para abb ítem con la posición cero y el arreglo de este va a ser el arreglo del tipo abb ítem que creamos arriba.

Para que los ítems estén ordenados por clave utilizamos un recorrido in order, al cual le pasamos el árbol, la función visitar en este caso va a ser guardar calve valor y el extra va a ser el nodo abb iterm.

La función guardar calve valor lo que hace es guardar en la posición del arreglo del nodo su clave y su valor e incrementa la posición para la luego pasar al próximo nodo.

El orden de esta primitiva es O(n) porque al utilizar un recorrido in order se pasa una vez por cada nodo del árbol, y por cada nodo se realizan operaciones de tiempo constante O(1).

Analizando la función top k:

Comienzo creando un heap de máximos y encolando los primeros k elementos (si el tamaño del arreglo es menor que k, por defecto encolo todo el arreglo).

Luego surgen dos posibilidades, que el tamaño del arreglo sea menor que k o que sea mayor que k.

En el caso en que sea menor, creo un vector dinámico en el cual voy a ir insertando al final todos los elementos del heap, hasta que este vacío. Luego ordeno este vector de menor a mayor mediante el método de ordenamiento heap sort. Y por último inserto al final de este vector, una cantidad de punteros a NULL, que es la diferencia que hay entre el tamaño del arreglo y el numero k. Luego devuelvo el arreglo de este vector.

En el caso en que el tamaño del arreglo sea mayor que k, para obtener los k menores del arreglo lo que se hace es ir comparando, desde la posición k del arreglo hasta el final, el máximo del heap con la posición actual del arreglo, si el valor del máximo del heap es mayor que el de la posición del arreglo, desencolo el máximo del heap y encolo la posición actual del arreglo al heap. Al finalizar las comparaciones lo que se hace es ir desencolando el heap e insertando estos valores a un arreglo, el cual se va a ordenador mediante el método de ordenamiento heap sort y va a ser devuelto.

El orden de este algoritmo es O(nlogk) ya que el heap tiene a lo sumo k elementos, encolar y desencolar en el heap cuesta O(logk). Una cota para la cantidad veces que se encola y desencola sería n, siendo n el tamaño del arreglo recibido. El resto de las operaciones dentro de la función son a los sumo O(n) o O(k) (menores que O(nlogk)),porque son recorrer ambos arreglos, ir comparando elementos, insertando, borrando. En conclusión el orden de esta función es O(nlogk).

Analizando la primitiva heap actualizar prioridad:

Lo primero que hacemos es fijarnos que exista el dato, de no existir este no hay forma de actualizar su prioridad, su posición.

Primero lo que hacemos es buscar su posición actual, eso lo hacemos mediante una búsqueda lineal. Guardamos en un arreglo el vector que tiene el heap y vamos comparando cada elemento que tiene este con el dato, si son iguales el elemento del arreglo con el dato ya encontramos la posición del dato en el heap.

A la posición encontrada le aplicamos downheap y upheap, para ordenarla con respecto a su padre y a sus hijos.

El orden de este algoritmo es O(n) ya que buscar la posición del dato en el heap con una búsqueda lineal es O(n) y luego las operaciones de downheap y upheap son O(logn), como O(n) es mayor que O(logn), el orden de la primitiva queda O(n).