Instituto Tecnológico y de Estudios Superiores de Monterrey

Jesús Eduardo Escobar Meza - A01743270

Programación de estructuras de datos y algoritmos fundamentales (Gpo 603)

Tarea 5.2 Documento reflexión

15 de noviembre de 2024

1) ¿Mi autoevaluación del desempeño en esta actividad integradora es? (Numérica entre 1 y 100))

100

2) Para cada integrante del equipo una coevaluación

¿La coevaluación en esta actividad integradora que doy a mi compañero [*Ángel Gabriel Camacho Pérez*] es? (Numérica entre 1 y 100)

La calificación es de 100

¿La coevaluación en esta actividad integradora que doy a mi compañero [*Ana Paula Navarro Hernández*] es? (Numérica entre 1 y 100)

La calificación es de 100

Importancia y eficiencia de usar Hash Table:

Gracias a las tablas hash podemos tener un acceso casi directo al dato que estamos buscando ya que esta la identifica con una llave (en este caso no fue única ya que usamos la IP para generar dicha llave) y con esto podemos lograr una mayor eficiencia haciendo que su complejidad sea de O(1) esto también funciona para la eliminación, e inserción.

Ventajas: Como anteriormente se mencionó al momento de tener una clave para cada IP pudimos tener una búsqueda mucho mas rápida, ya que esta no necesita que estarse recorriendo o partiendo un arreglo (esto también fue mas eficiente ya que la tabla hash que se agrego fue una con dirección por encadenamiento haciendo que cada IP tuviera su propia posición).

Además de que se nos fue de una mayor facilidad su implementación ya que son muy complejas y su función nos permite hacer consultas basadas en claves (parte fundamental de esta entrega)

Desventajas: El problema que tienen las hash es el calculo para poder calcular su posición ya que cuando tienes muchos datos la probabilidad con la que las claves de estos colisionen es muy alta y esto podría hacer que la búsqueda en lugar de ser de O(1) se vaya haciendo de o(n) (en nuestro programa no fue el caso ya que lo hicimos por encadenamiento).

Otra cosa que puede llegar a ser de desventaja es el uso de memoria debido a que cuando está llena hasta cierto porcentaje de su arreglo se hace un rehashing (función de la cual se hablara después, pero en pequeñas palabras aumenta el tamaño del arreglo) hace que el arreglo aumente hasta esto a veces es bueno pero otras veces puede llegar a ser malo debido a que si se aumenta demasiado podría queda un arreglo con muchos espacios vacíos.

Explicación de las clases, funciones, estructura del código y cambios:

Para esta actividad se nos pidió implementar una Hash Table para acomodar el archivo “bitacora3.txt” que sería entregada por el profesor las tabla hash nos ayudaría para que cada IP se acomodará en cada punto de un arreglo, esto es importante ya que el programa le pedirá al usuario una IP y en consola se imprimirá una lista con todas fechas de forma ascendente las cuales contengan la misma IP (cabe aclarar que como se mencionó anteriormente se uso una hash table por encadenamiento solo que el encadenamiento no se hizo como normalmente se hace que sería con una linked list si no que se hizo con un BST ya que sería mejor por la complejidad del ordenamiento). El BST en esta actividad se hizo para las fechas de la IP que se pidió seria insertada y ordenada gracias a este tipo de algoritmo.

Los cambios principales para este programa fue que tuvimos fue la implementación de un nuevo tipo de código con el que pudimos hacer un nuevo almacenamiento una nueva forma de búsqueda con una menor complejidad haciendo que el programa fuera mucho más eficiente.

Hash Table:

Esta clase se usó para el almacenamiento de cada IP y la impresión de las fechas

Se usaron varias funciones para esta clase, pero las principales fueron getPos (la cual nos sirve para calcular la posición en la que debe de estar una IP, su complejidad es de O(1), put (Este método nos sirve para insertar un nuevo nodo en la posición hash calculada con la función antes mencionada además de que también verifica si la tabla ya fue llenada por más del 75% si es así hace un rehashing dicha función será explicada más adelante, su complejidad es de O(log n)), showList (Esta función nos permite mostrar la lista de las fechas las cuales son mandas por la función mostrar que está en la clase BST, su complejidad es de O(n)), búsqueda (Eta función es la que se encarga de hacer la búsqueda de la IP ingresada por el usuario y usa la función antes mencionada parar mostrar todas fechas que tienen esa IP, su complejidad es de O(n)), rehashing(Esta función nos permite aumentar de tamaño el arreglo de la hash table y restructurarla para que se puedan ingresar más valores con la siguiente formular tamañoArreglo * 2 + 1, su complejidad es de O(n)), obtenerFechaID(Esta función nos permite obtener el ID de la fecha esto se hace obteniendo el número de segundos que han pasado tras esa fecha, con esto podemos ubicar de manera más rápida su llamado al momento de leer el archivo, su complejidad es de O(1)), insertLine(Esta función nos permite leer una línea del texto dividirla entre cada uno de sus componentes y después de eso insertarla en la hash para así poder hacer el BST de forma correcta ya que para obtener bien en donde debe de estar y poder ser enviado de forma correcta al momento de ser llamada se utilizaron las funciones del BST que se verán a continuación, su complejidad de O(log n)) y leerArchivo(Esta fusión se encarga de leer el archivo asignado línea por línea después de eso manda a llamar a la función anterior insertLine la cual ya se encarga de hacer la inserción em la hash table, su complejidad de O(n log n)).

BST:

Esta clase hace el almacenamiento de cada IP repetida y ordena y manda las fechas de manera ascendente a la clase que se encarga de imprimir las fechas en el hash table.

Se usaron varias funciones para esta clase, pero las principales fueron insert (función la cual nos permite ingresar un nuevo nodo en el BST, este código fue diseñado en una tarea, actividad y practica anteriores por lo que fue de fácil implementación, su complejidad es de O(log n)), getRaizData (Esta función nos devuelve la data de la raíz, cuenta con una complejidad de O(1)), getRaizKey (Esta función solo nos devuelve la key de la raíz, su complejidad es de O(1)), getRaizID (Esta función nos devuelve el ID de la raíz su complejidad seria de O(1)), mostrar (Esta función muestra los nodos del BST y se manda a llamar en la función showList de la hashtable, su complejidad es de O(n)), inorder (Esta funcion nos permite recorrer el árbol y mostrar sus nodos, su complejidad es de O(n)), eliminar (Esta funcion nos permite eliminar todos los nodos del BST recorriendo cada nodo del BST, su complejidad es de O(n)) y eliminarRaiz (Esta función nos permite eliminar el nodo raíz de BST además de que lo restructura de manera correcta, su complejidad es de O(log n)).

Ahora porque se eligió un BST y no una linkedlist simple al principio si se implementó con una linkedlist pero después de ver la complejidad de los métodos se tomó la decisión de cambiar por un BST ya que para el acomodo de las fechas al momento de mostrarlo sería más rápido y eficiente con un BST al igual que la entrega anterior por lo que después de consultar distintas paginas se tomo la decisión que por temas de complejidad e implementación se haría con un BST.