Act 5.2 - Actividad Integral sobre el uso de códigos hash (Evidencia Competencia)

Programación de estructuras de datos y algoritmos fundamentales (Gpo 4)

Alumno:

Thomas Freund Paternostro //A00831997

Fecha de entrega:

26/11/2021

Complejidad:

main:

```
8   int main(){
9       //Alimentamos los datos dentro del HashTable:
10      IPHashTable hTable;
11      ifstream bitacora("bitacoraACT5_2.txt");
12      string currentLine;
13      while(getline(bitacora, currentLine)){
14          stringstream clStream(currentLine);
15          string mes, dia, hora, ip, razonFalla;
16          clStream >> mes >> dia >> hora >> ip;
17          getline(clStream, razonFalla);
18          razonFalla = razonFalla.substr(1);
19          string fecha = mes + " " + dia + " " + hora;
20          hTable.addIPAccess(ip, fecha, razonFalla);
21      }
22
23      //Le pedimos al usuario la información que necesita:
24      string ipSolicitada;
25      do{
26          cout << "Escribe la direccion IP de la que quieres el resumen (escribe -1 para terminar): ";
27          cin >> ipSolicitada;
28          cout << endl;
29          hTable.printIPSummary(ipSolicitada);
30      } while (ipSolicitada != "-1");
31
32      return 0;
33  }
```

De la línea 10 a las 12 la complejidad de es O(1), de la línea 13 a la 21 lee todas las líneas de la bitácora, **por lo tanto la complejidad es O(n) (siendo n el número de líneas en el archivo)**, la línea 25 a la 30 es también O(n), pero en este caso n es igual al número de IPs que solicite ver el usuario. Aquí se utiliza el método dónde se solicita el IP con sus atributos y sus repeticiones, que tiene complejidad constante O(1).

getIPIndex y addIPAccess:

```
class IPHashTable{
    private:
        forward_list<IPSummary> ipSummaries[TABLE_SIZE];
        //Esta es la función hash, que con la IP (el key) determina el índice dentro del arreglo
        int getIPIndex(string ipAsString){
            int octets[2];
            for(int i = 0; i < 2; i++){
                int dotIndex = ipAsString.find('.');
                octets[i] = stoi(ipAsString.substr(0, dotIndex));
                ipAsString = ipAsString.substr(dotIndex+1);
            }

            return octets[0]*256 + octets[1];
        }
    public:
        void addIPAccess(string ipAsString, string accessDate, string failReason){
            int ipIndex = getIPIndex(ipAsString);
            if(ipSummaries[ipIndex].empty()){
                ipSummaries[ipIndex].push_front(IPSummary(ipAsString));
                ipSummaries[ipIndex].front().addAccess(ipAsString, accessDate, failReason);
            } else {
                forward_list<IPSummary>::iterator it = ipSummaries[ipIndex].begin();
                while(it != ipSummaries[ipIndex].end() && !(it->isSameIP(ipAsString))){
                    advance(it, 1);
                }
                if(it == ipSummaries[ipIndex].end()){
                    ipSummaries[ipIndex].push_front(IPSummary(ipAsString));
                    ipSummaries[ipIndex].front().addAccess(ipAsString, accessDate, failReason);
                } else {
                    it->addAccess(ipAsString, accessDate, failReason);
                }
            }
        }
```

La función “getIPindex” es una función de complejidad constante, o sea O(1). Esto es porque, aunque hay un ciclo for, este ciclo siempre se ejecuta dos veces (se podría haber conseguido el mismo efecto simplemente copiando el código dos veces, sin usar un for), **La complejidad de esta función es O(1) que esencialmente es la complejidad de el HashTable.**

En cuanto a la función “addIPAccess”, ésta **tiene una complejidad de O(1)** (si es que no hay colisiones), al no tener ciclos ni otras líneas que se repitan más de una vez dependiendo de una variable.

IPSummary:

```
29  IPSummary(string stringForm){
30      string strRemaining = stringForm;
31      for(int i = 0; i < 3; i++){
32          int dotIndex = strRemaining.find('.');
33          octets[i] = stoi(strRemaining.substr(0, dotIndex));
34          strRemaining = strRemaining.substr(dotIndex+1);
35      }
36      int twoDotIndex = strRemaining.find(':');
37
38      if(twoDotIndex == string::npos){
39          octets[3] = stoi(strRemaining.substr(0, strRemaining.size()));
40      } else {
41          octets[3] = stoi(strRemaining.substr(0, twoDotIndex));
42      }
43  }
```

Todas las líneas de esta función tienen una complejidad constante, ya que a pesar de que también haya un ciclo for aquí, sólo se repite una cantidad predefinida de veces. **Por lo tanto la complejidad de toda la función es O(1).**

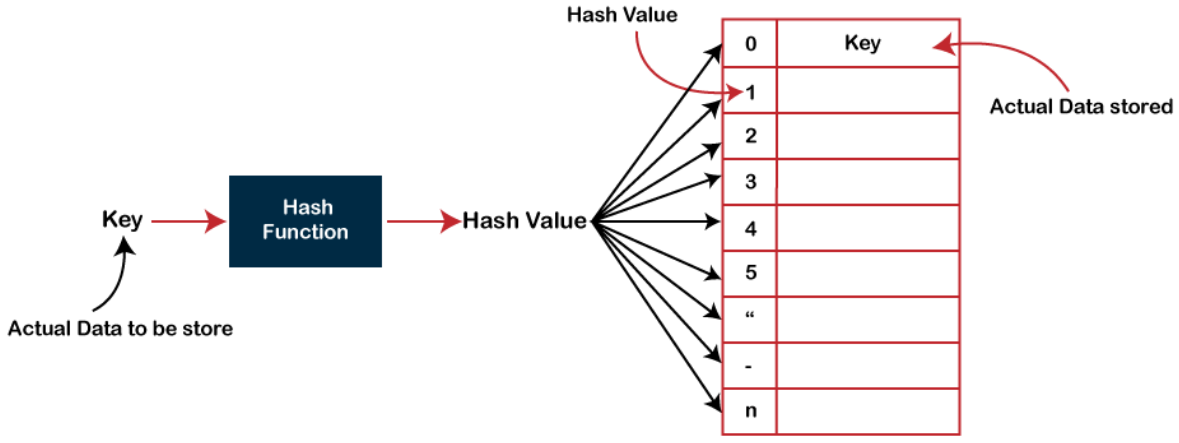
isSameIP:

```
45  bool isSameIP(string stringForm){
46      string strRemaining = stringForm;
47      for(int i = 0; i < 3; i++){
48          int dotIndex = strRemaining.find('.');
49          if(octets[i] != stoi(strRemaining.substr(0, dotIndex))){
50              return false;
51          }
52          strRemaining = strRemaining.substr(dotIndex+1);
53      }
54      int twoDotIndex = strRemaining.find(':');
55
56      if(twoDotIndex == string::npos){
57          if(octets[3] != stoi(strRemaining.substr(0, strRemaining.size()))) return false;
58      } else {
59          if(octets[3] != stoi(strRemaining.substr(0, twoDotIndex))) return false;
60      }
61
62      return true;
63  }
```

En las líneas 47 a la 50 se tiene una complejidad de O(1), aunque use un ciclo for que divide en octetos la IP que recibió la función como parámetro (igual que en las situaciones anteriores, se repite un número constante de veces). El resto de las líneas tiene una complejidad de O(1). **Entonces, esta función tiene complejidad O(1).**

Realiza una investigación y reflexión de la importancia y eficiencia del uso de las tablas hash para tales fines

Los hash tables son una lista de valores pares donde el primer valor es una llave y el segundo es el valor, donde se puede acceder de manera rápida al valor por medio de la llave. Si se desea hacer búsquedas rápidas, esta estructura es una de las más eficientes y por sus propiedades también es útil para eliminar o insertar elementos con una complejidad de O(n) en el peor de los casos.



Como asigna una hashtable sus valores

Sin embargo esta estructura puede ser no la más eficiente si se requiere estructura dado que esta no mantiene un orden en específico, pero como fue analizado previamente para estas entregas si se considera ideal.

En el caso del trabajo realizado por el equipo se llegó a la misma solución utilizando unorder_map pero se consideró por motivos de aprendizaje y también de investigación más útil demostrar su uso desarrollando las tablas hash y no una librería de C++ que prácticamente hace todo el hashing sin hacer realmente la actividad.

Se puede considerar que la bitácora tiene más de 91000 entradas y no es posible que este tenga la misma cantidad de ips si estos se repiten varias veces. Al mismo tiempo en los primeros dos octetos de cada ip hay una posibilidad de 256 combinaciones del 0-255 haciendo 65536 posibles combinaciones como se puede ver en la imagen inferior del código.

```cpp
#define TABLE_SIZE 65536 //256^2

using namespace std;

class IPHashTable{
    private:
        forward_list<IPSummary> ipSummaries[TABLE_SIZE];
        //Esta es la función hash, que con la IP (el key) determina el índice dentro del arreglo
        int getIPIndex(string ipAsString){
            int octets[2];
            for(int i = 0; i < 2; i++){
                int dotIndex = ipAsString.find('.');
                octets[i] = stoi(ipAsString.substr(0, dotIndex));
                ipAsString = ipAsString.substr(dotIndex+1);
            }

            return octets[0]*256 + octets[1];
        }
```

Para poder asignar cada ip a una “key” lo que se realizó en el ip es multiplicar el primer octeto por del ip por 256, el número de combinaciones y sumarle el valor del segundo octeto generando una llave particular que sería llamada cada vez que esta aparezca y donde se pueda guardar los otros datos en la lista que se creó.

En los hash tables se tiene que considerar que no es posible saber de antemano cuántas colisiones pueden haber con una función hash. Sin embargo, prácticamente todas las implementaciones de tablas hash ofrecen O(1) en la gran mayoría de insertos. Esto es lo mismo que insertar matriz: es O(1) a menos que necesite cambiar el tamaño, en cuyo caso es O (n), más la incertidumbre de colisión. Donde para prácticamente todos los casos de uso, las tablas hash tienen una complejidad O(1) donde la complejidad de inserción es O(1) y la de búsqueda es O(1).

Como fue previamente mencionado, fuera del caso que se tenga dos datos en una memoria que es cabe resaltar que es bastante raro, la complejidad de cada operación es O(n) y por lo tanto esta estructura de datos es altamente eficiente en términos de complejidad y utilidad dado que es una estructura de datos ideal para resolver problemas de esta índole.

En la parte inferior se puede ver como se encontró la cantidad de veces que fue llamado un IP arbitrario en la entrega que en este caso sería la llave y los datos que aparecen como las fechas y razones de la falla.

```
Escribe la direccion IP de la que quieres el resumen (escribe -1 para terminar): 195.44.205.157

Accesos a la IP 195.44.205.157 (8 accesos):
Puerto: 4169 | Fecha: Jul 09 16:19:58 | Razon de falla: Illegal user
Puerto: 3742 | Fecha: Jun 09 01:32:22 | Razon de falla: Failed password for illegal user guest
Puerto: 6129 | Fecha: Jul 02 16:36:33 | Razon de falla: Failed password for admin
Puerto: 3355 | Fecha: Oct 13 20:44:49 | Razon de falla: Failed password for illegal user guest
Puerto: 9040 | Fecha: Jun 10 03:20:15 | Razon de falla: Failed password for admin
Puerto: 3098 | Fecha: Jul 26 04:39:18 | Razon de falla: Failed password for illegal user guest
Puerto: 4581 | Fecha: Aug 26 04:22:30 | Razon de falla: Failed password for illegal user guest
Puerto: 4021 | Fecha: Sep 25 18:11:34 | Razon de falla: Failed password for illegal user root
```