ACT 6.2

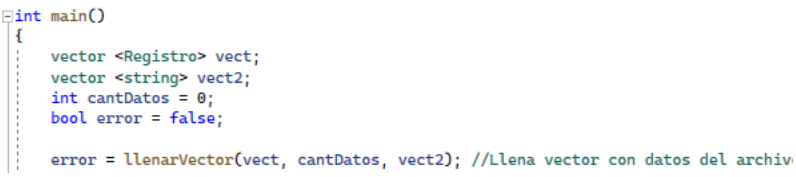
Elías Uriel Velázquez Rojas
A01639716

Elías Uriel Velázquez Rojas

- ¿ Cuáles son las más eficientes?
- ¿Cuáles podrías mejorar y argumenta cómo harías esta mejora?

En nuestra segunda actividad el programa consistía básicamente en tomar cierta cantidad de datos de un txt, para después con estos datos ordenarlos y hacer una búsqueda de algunos de ellos, lo que hicimos fue guardar los datos en un txt para después ordenarlos, esto es muy parecido a la primera actividad que realizamos solo que en la anterior nosotros le insertábamos los datos en esta los agarra de un txt.

Los algoritmos de ordenamiento son aquellos que nos permiten ordenar información o estructuras de datos basándonos en un criterio de ordenamiento como de mayor a menor o viceversa, los algoritmos de búsquedas se usan para encontrar un valor especifico dentro de una estructura de datos.



```
int main()
{
    vector <Registro> vect;
    vector <string> vect2;
    int cantDatos = 0;
    bool error = false;

    error = llenarVector(vect, cantDatos, vect2); //Llena vector con datos del archiv
```

El método de ordenamiento que nosotros usamos fue el de burbuja, pero en nuestro caso no es el mejor algoritmo de búsqueda que pudiera ver, pero en su momento fue el que más entendí, pero ya analizando después creo que pude haber usado el Quicksort ya que este es más eficiente que el que usamos y haría nuestro programa un poco más funcional.

```
//Ordena meses de menor a mayor:
void Funciones::ordenaBurbujaMes(vector<Registro>& vect, vector<string>& vect2)
{
    int i, j;
    for (i = 0; i < vect.size(); i++)
        for (j = i + 1; j < vect.size(); j++)
            if (vect[i].mes > vect[j].mes) {
                swap(&vect2[i], &vect2[j]);
                swap2(&vect[i], &vect[j]);
            }

}

//Ordena dias de menor a mayor:
void Funciones::ordenaBurbujaDia(vector<Registro>& vect, vector<string>& vect2, int ini, int fin)
{
    int i, j;
    for (i = ini; i < fin; i++)
        for (j = i + 1; j < fin; j++)
            if (vect[i].dia > vect[j].dia) {
                swap(&vect2[i], &vect2[j]);
                swap2(&vect[i], &vect[j]);
            }
}
```

Y en el caso de las búsquedas usamos la búsqueda lineal para poder encontrar los numero en este caso el mes y día, en esta ocasión también podíamos implementar la búsqueda binaria ya que los datos ya estaban ordenados y en este caso la búsqueda pude haber sido mucho más eficaz.

En conclusión, podemos decir que los métodos que usamos no fueron lo más eficientes, pero en nuestro caso a hacer nuestro primer intento se puede valer ya que no teníamos mucha experiencia realizando los algoritmos y este fue al primero que le entendimos.

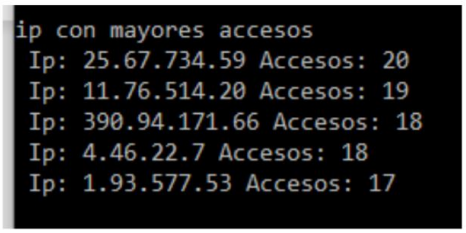
```
//Busca indice del mes y dia ingresados:
int Funciones::busqSecuencial(std::vector<Registro>& vect, int mes, int dia)
{
    for (int i = 0; i < vect.size(); i++)
        if (mes == vect[i].mes)
            for (int j = i; j < vect.size(); j++)
                if (dia == vect[j].dia)
                    return j;
    return -1;
}

int Funciones::busquedaEspecifico(vector<Registro>& vect, int indice, int mes, int dia)
{
    int ind = indice;
    for (int i = indice; i < vect.size(); i++) {
        if (vect[i].dia == dia) {
            ind++;
        }
    }
    return ind - 1;
}
```

En la actividad 3.4 usamos arboles de búsqueda binarios o (BST) el cual consiste que en una estructura de datos se almacenan a través de un nodo raíz y se acomodan los datos hacia la izquierda si es menor y derecha si es mayor, estos tienen una complejidad O(log(n)), en nuestro caso usamos el método de inorder en este caso para nosotros fue la mejor opción ya que queríamos que los datos se mostraran de esa forma con la forma del árbol, en este caso la inserción de los datos sirve solo para que los visualices como tu desees, ya que son igual de eficientes todos son O(log(n)).

Por lo que podríamos decir que nuestro programa fue correcto, porque nosotros buscamos esa implementación, en la estructura se crea un bst para meter los valores de la IP en un tipo string y los accesos para cada ip, después creamos la clase TreeNode que nos va a representar los nodos del árbol, en público esta su constructor y otra función para insertar los valores, los atributos un apuntador izquierda y derecha y un data de estructura bst. Y se imprime así.

CASOS PRUEBA 1:



```
void Tree::inorderTraversal()
{
    int limite = 5;
    cout << "ip con mayores accesos " << endl;
    inorderHelper(root, limite);
    cout << endl;
}
void Tree::inorderHelper(TreeNode* node,int &limit)
{
    if (node != NULL && limit > 0)
    {
        inorderHelper(node->rightNode, limit);
        if (limit > 0) {
            limit--;
            cout << " Ip: " << node->data.dirIP<<"Accesos: "<< node->data.accesos<
        }
        if (limit > 0) {
            inorderHelper(node->leftNode, limit);
        }
    }
}
```

ACTIVIDAD INTEGRAL DE GRAFOS

En esta actividad usamos los grafos para encontrar los fan-out de cada uno, teníamos que determinar cuales son los nodos con mayor fan-out, y definir la dirección de ip donde estaría boot-master, los grafos son una estructura de datos de tipo RED, siendo el caso más general que existe y tiene una relación "Muchos a muchos" (N:M). Existen dos tipos de grafos: el grafo No-Dirigido y el Dirigido.

```
class Graph
{
public:
    vector<vector<src>> adjList;
    Graph(vector<Edge> const& edges, int N)
    {

        adjList.resize(N);

        for (auto& edge : edges)
        {
            src eje = { edge.src, edge.repe };
            src eje2 = { edge.dest, edge.repe };
            adjList[edge.src].push_back(eje2);
            adjList[edge.dest].push_back(eje);
        }
    }
};
```

En nuestro caso realizamos un grafo no dirigido, para que los valores apunten de ida y de regreso, es la clase grafo con sus atributos en público un vector, creado de un vector de tipo src llamado adjList, y luego se crea su constructor, con los atributos un vector de tipo Edge y uno entero, se usa un for hasta Edge, y se declaran ejes, y se usa el push back, para poder hacer el grafo no dirigido.

En este caso de el programa la mejor opción era hacer el grafo no dirigido ya que queríamos contar los fan out, en este caso, por eso si lo hicimos el grafo dirigido cambiaría y sería más difícil pero para algunos casos este tipo de grafo lo favorece porque si no necesitas que apunte al valor anterior puedes hacer un grafo dirigido es favorable dependiendo el caso o lo que quieras solucionar.

Pero se puede representar de distintas formas como

- Matriz de Adyacencias: permite ver si existe o no arista entre dos nodos O(1) y provoca que operaciones sobre grafos sean sencillas.
- Lista de Adyacencias: no necesita saber cantidad de nodos y arcos, hace buen uso de memoria y tiene complejidad O(n).
- Lista de Arcos: es una representación muy eficiente y tiene una complejidad de O(n log n).
- Breadth First: Si hay más de una solución, encontrará una solución con pasos mínimos.
- Depth First: tiene una baja complejidad de tiempo y espacio que BFS.

Nosotros usamos la matriz de adyacencias y la lista de adyacencias para poder agarrar los datos provenientes del txt, en conclusión tu puedes decidir cual es la mejor opción la más eficiente es la matriz de adyacencias y si es menos dirigido será mucho más rápido pero habrá algunas excepciones

Y por último las Hash tables las cuales son las tablas hash es una estructura de datos en la cual se asocian llaves o keys a los valores, su operación principal es la búsqueda de valores ya que permite el acceso a los elementos usando las llaves, el acceso a los datos suele ser muy rápido.

En nuestro caso en nuestro programa para la hash tables creamos una clase llamada Hashtable, a su vez tambien se crea un enum y una estructura, el enum tiene valor, ocupado, y borrado, y la estructura que tiene los atributos de accesos, ip info y el enum estado que se declara como vacío. En la clase los atributos privados, son la lista celda que se llama table, y otra variable entera llamado total_elements y en público se declara el constructor y la función para insertar los elementos, que usa los accesos de la ip y con el push back se guarda, esto es lo principal para poder armar nuestro has tables.

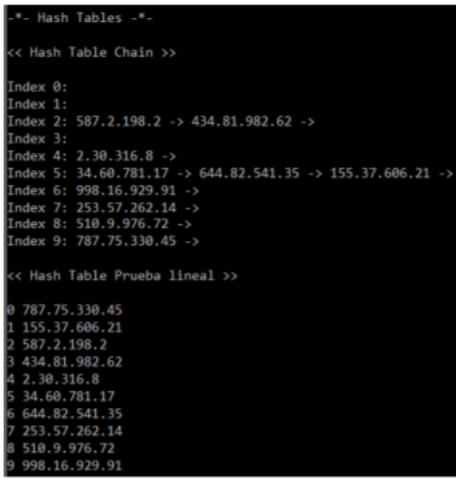
Nosotros asociamos el valor de la ip con el valor de txt, y usamos una función para que se escribieran en el mismo , la función chain como se puede ver en el index 2, hicimos dos tablas la lineal y la chain,

Se llama la función HashF y después se hace un for para insertar los valores recibidos, para después hacer lo mismo con los hash tables, con el arreglo, si se crearon correctamente imprime la pantalla, y se aumenta el contador de listo y se termina esa verificación, lo otro que sucede es que se manda a llamar la función de hashtable para imprimir que es Hash Table Chain y después con la función mostrar la Hash Table Prueba Lineal, otra opción te pide que ingreses la IP y los datos correctamente, después buscar esa ip dentro de la Hash Table Chain, al igual de encontrar la llave en la prueba lineal para después imprimirlo

Para nosotros no pareció la mejor opción esto porque dependía de el numero de llaves y de datos, mientras más eran obviamente era más lento y menos eficiente, pero lo que tratamos de hacer era eficientizar todo, y al final imprimirlo.

En conclusión las hash tabes son muy útiles para el manejo de datos aunque no la mejor forma, ya que es fácil equivocarse, pero es muy eficiente ya que es o(n) y hace rápida su búsqueda la manera que lo implementamos pienso que es la mejor posible y estamos orgullosos de eso.

GRACIAS PROFESOR POR IMPARTIR LA MATERIA 😊


-*- Hash Tables -*-
<< Hash Table Chain >>
Index 0:
Index 1:
Index 2: 587.2.198.2 -> 434.81.982.62 ->
Index 3:
Index 4: 2.30.316.8 ->
Index 5: 34.60.781.17 -> 644.82.541.35 -> 155.37.606.21 ->
Index 6: 998.16.929.91 ->
Index 7: 253.57.262.14 ->
Index 8: 510.9.976.72 ->
Index 9: 787.75.330.45 ->
<< Hash Table Prueba lineal >>
0 787.75.330.45
1 155.37.606.21
2 587.2.198.2
3 434.81.982.62
4 2.30.316.8
5 34.60.781.17
6 644.82.541.35
7 253.57.262.14
8 510.9.976.72
9 998.16.929.91