Ines Alejandro Garcia Mosqueda A00834571
Jesús Fong Ruiz A01254062

Tecnológico de Monterrey

Documentación de código

Programming of data structures

Inés Alejandro Garcia Mosqueda A00834571
Jesus Fong Ruiz A01254062

Profesores: Luis Ricardo Peña Llamas
Jorge Gonzalez

10/10/2022

Link del repositorio del proyecto
https://github.com/Ineso1/Data-structure-and-algorithms-proyect

En el código principalmente existen dos diferentes clases: **Node** y **LinkedList**

Clase **Node** es una estructura que encapsula los datos de la lectura de la bitácora.txt (mes, hora, día, ip, razón) aparte contiene dos datos de tipo Nodo en forma de puntero, estos para apuntar un nodo consecutivo y anterior a el nodo instanciado.

```
class Node{
    public:
    std::string month, day, hour, ip, reason;
    Node *nextNode;
    Node *previousNode;
    Node(); //Constructor sin parametros
    Node(std::string&, std::string&, std::string&, std::string&, std::string&, Node*, Node*);  //Constructor con datos del registro
    std::string getData();  //Obtencion de los datos del nodo en formato
    void setData(std::string&, std::string&, std::string&, std::string&, std::string&); //Sobrescritura de los datos del nodo
    ~Node(); //Destructor de Nodo
    friend std::ostream &operator <<(std::ostream& salida, const Node&); //Sobrecarga de operador para imprimir dato del nodo
    int dateIntCode(); //Codificacion de fecha para comparacion
    unsigned long long ipIntCode(); //Codificacion de ip para comparacion
};
```

Clase **LinkedList** es una estructura que contiene un puntero de nodo principal como head de la lista (nodo principal) a partir de este se ejecutan todos los métodos de la lista

```
class LinkedList{
    public:
        LinkedList();
        Node* head; //Nodo pricipal
        Node* tail; //Cola de la lista
        int len = 0; //Longitud de la lista
        Node* insertBefore(Node*, std::string, std::string, std::string, std::string, std::string); //inserta un nodo antes de otro dado la informacion del nodo nuevo
        Node* insertBefore(Node*, Node*); //Inserta un nodo antes de otro dado el nodo
        Node* insertAfter(Node*, std::string, std::string, std::string, std::string, std::string); //Inserta un nodo despues de otro nodo dado la informacion del nuevo nodo
        Node* insertAfter(Node*, Node*); //Inserta un nodo despues de otro nodo dado el nuevo nodo
        void fillList(std::string); //Llena la lista dado un archivo txt
        void printList(); //Imprime todos los datos de la lista
        int search(std::string, std::string, std::string, std::string, std::string); //Busca un nodo dados los datos del nodo
        Node &operator[](int); //Sobrecarga de operador para busqueda de un elemento en un indice dado
        void update(int, std::string, std::string, std::string, std::string, std::string); //Sobrescribe la informacion de un nodo dada la nueva informacion del nodo
        void deleteNode(int); //Algoritmo para borrar un nodo enlazando el resto de nodos
        int sortNodeDate(Node*&); //Algoritmo de ordenamiento por fecha
        int sortNodeIp(Node*&); //Algoritno de ordenamiento por Ip
        int sortNodeIp2(Node*&); //Intento de optimizacion de ordenamiento por Ip (en proceso)
        void saveSortedList(); //Creacion / Sobrescitura de un archivo txt para almacenamiento de informacion
        void findRangeIp(std::string, std::string); //Filtro de busqueda de Ip por rango de Ip
};
```

Listado de métodos implementados en LinkedList

- Node* **insertBefore**(Node*, std::string, std::string, std::string, std::string, std::string);

Inserta un nodo antes de otro dado la información del nodo nuevo

Este método tiene dos formatos de parámetros para facilitar la implementación dentro de los algoritmos que lo necesiten.

La inserción de un nodo dado otro en el método puede considerase complejidad constante, ya que no hay iteraciones internas, solo crea el nodo con los datos dados por los parámetros y enlaza los nodos.

Complejidad O(1)

- Node* **insertBefore**(Node*, Node*);

Inserta un nodo antes de otro dado el nodo

Igual que el método anterior, a comparación que en este caso se le entrega la dirección del nodo que se va a enlazar al nodo dado en el parámetro.

Complejidad O(1)

- Node* **insertAfter**(Node*, std::string, std::string, std::string, std::string, std::string);

Inserta un nodo después de otro nodo dado la información del nuevo nodo.

Similar a los métodos de insertBefore se puede considerar la complejidad de los métodos insertAfter constante, ya que tampoco hay ciclos internos.

Complejidad O(1)

- Node* **insertAfter**(Node*, Node*);

Inserta un nodo después de otro nodo dado el nuevo nodo.

Mismo método de insertAfeter con distintos parámetros para facilitar algunas implementaciones dentro del código.

Complejidad O(1)

- void **fillList**(std::string);

Llena la lista dado un archivo txt

El método abre un documento dado por parámetro en forma de string, el cual contiene internamente la implementación de ordenamiento conforme se van leyendo los datos del archivo

Conforme los datos van entrando a la lista se compara cada dato de la lista para encontrar la posición adecuada de cada dato dentro de la lista y lo inserta, lo cual hace que en el peor caso tenga que recorrer toda la lista, la complejidad de este algoritmo para el posicionamiento de los datos va creciendo conforme se va aumentando la cantidad de datos dentro de la lista dando la siguiente complejidad

Complejidad O(nlogn)

- void **printList**();

Imprime todos los datos de la lista

Dado que se tienen los datos ordenados y se requiere imprimir todos los datos ordenados, se recorre toda la lista dando complejidad n

Complejidad O(n)

- int **search**(std::string, std::string, std::string, std::string, std::string);

Busca un nodo dados los datos del nodo

Al no poder acceder a la LinkedList de forma aleatoria tendremos que recorrer la lista ya sea por el frente de la lista o por la cola de la lista, dando una complejidad n

Complejidad O(n)

- Node &operator**[]**(int);

Sobrecarga de operador para búsqueda de un elemento en un índice dado

Este operador recorre nodo por nodo la cantidad que se tiene obtiene como índice dentro de los corchetes, por lo que el usar el operador en su peor caso se obtiene complejidad n

Complejidad O(n)

- void **update**(int, std::string, std::string, std::string, std::string, std::string);

Sobrescribe la información de un nodo dada la nueva información del nodo

Internamente se utiliza el operador [ ] por lo que la búsqueda en cierto índice otorga complejidad n, para sobrescribir los atributos del nodo se utiliza el método setData() el cual otorga complejidad constante por lo que la complejidad total es

Complejidad O(n)

- void **deleteNode**(int);

Algoritmo para borrar un nodo enlazando el resto de los nodos

Dado un indice de nodo se accede a dicho nodo, se ejecuta el destructor del nodo previamente enlazando los nodos vecinos entre ellos para evitar la pérdida de los datos

- int **sortNodeDate**(Node*&);

Algoritmo de ordenamiento por fecha

Dentro de nodo se tiene un algoritmo que hace una codificación de la fecha de dicho nodo a una cantidad entera, por lo que se considera que la complejidad interna de este algoritmo es constante, ya que no tiene ciclos internos

El algoritmo recorre la lista en comparando los valores de la fecha de cada nodo en la lista hasta encontrar un espacio que le corresponda al valor del nodo dado y lo inserta

Siendo esta de complejidad O(n) ya que busca lineal

Complejidad O(n)

- int **sortNodeIp**(Node*&);

Algoritmo de ordenamiento por Ip

Similar al ordenamiento por fecha se busca nodo por nodo el espacio correcto para insertar el nodo dado, de acuerdo a la comparación de la traducción de la Ip a un código en formato de numero entero

Dicha función que codifica la ip a un número entero es método de la clase Node siendo esta de complejidad constante, ya que las iteraciones que se hacen dentro del código es finita y definida, ya que solo recorre los caracteres de la ip para generar el código

Complejidad O(n)

- void **saveSortedLis**t();

creación / Sobre escritura de un archivo txt para almacenamiento de información

Para poder guardar la lista ordenada dentro de un archivo txt se tiene que recorrer la lista completa, lo que genera una complejidad n

Complejidad O(n)

- void **findRangeIp**(std::string, std::string);

Filtro de búsqueda de Ip por rango de Ip

El método recorre la lista hasta encontrar el primer numero del rango, guarda los datos que satisface la condición del rango dado por el usuario hasta encontrar el otro extremo del rango

Esta función en su peor caso tiene complejidad n

Complejidad O(n)

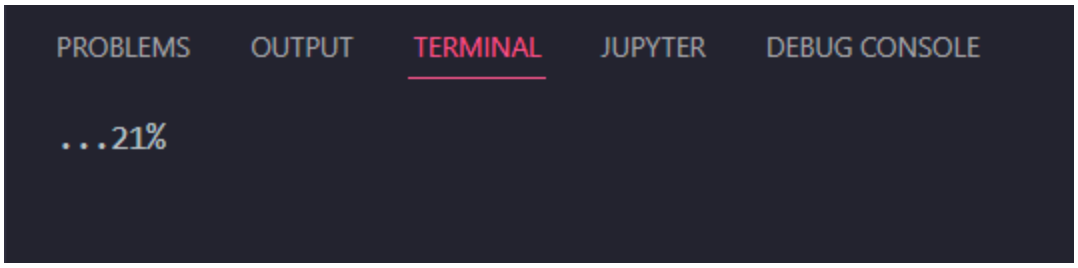
Implementación en Main y caso de prueba

```
int main(){
    LinkedList ll;  //Crea una lista doblemente ligada
    // ll.fillList("bitacoraPrueba.txt");   //Lectura de archivo para carga de lista
    ll.fillList("bitacora.txt");   //Lectura de archivo para carga de lista
    ll.printList(); //Impresion de la lista con datos del archivo
    ll.saveSortedList();
    cout << "El resultado del ordenamiento fue guardado en un archivo llamado sorted.txt";


    string ip1, ip2;
    cout << "\n\t\tLista de peticiones\n";
    cout << "A continuacion se realizara una busqueda\nIntroduce el rango de la busqueda de Ip\nXXX.XXX.XXX.XX:XXXX --> ";
    cin >> ip1;
    cout << "\nXXX.XXX.XXX.XX:XXXX --> ";
    cin >> ip2;
    try{
        ll.findRangeIp(ip1, ip2);
        }
    catch(const std::exception& e)
    {
        std::cerr << e.what() << ": otro error por aqui" << '\n';
        return -1;
    }
    cout << "\nEl resultado del ordenamiento fue guardado en un archivo llamado resultadoBusqueda.txt\n";
    return 0;
}
```

Se instancia el objeto de tipo LinkedList llamado ll, pedimos el llenado de la lista ligada desde el archivo bitácora.txt dentro de la misma carpeta

En la ejecución se tiene una animación de porcentaje de carga de los registros dentro de la lista ligada, para así tener una idea de lo que está haciendo el código



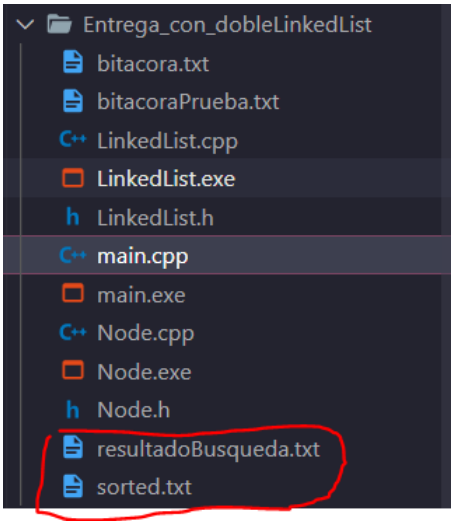
Imprimimos el ordenamiento de la lista por Ip además de guardarlo en un archivo llamado sorted.txt

Se piden dos Ip de las cuales vamos a tener un rango de búsqueda para después guardarlos en un archivo llamado resultadoBusqueda.txt

```
El resultado del ordenamiento fue guardado en un archivo llamado sorted.txt
                Lista de peticiones
A continuacion se realizara una busqueda
Introduce el rango de la busqueda de Ip
XXX.XXX.XXX.XX:XXXX --> 55.2.3.4:4512

XXX.XXX.XXX.XX:XXXX --> 145.5.3.6:4651

El resultado del ordenamiento fue guardado en un archivo llamado resultadoBusqueda.txt
PS C:\Users\nessy\Desktop\entregaXd\Data-structure-and-algorithms-proyect\Entrega_con_dobleLinkedList>
```



Siendo este nuestro principal objetivo del código de ordenamiento y filtrado

main.cpp | resultadoBusqueda.txt | LinkedList.cpp

Data-structure-and-algorithms-proyect > Entrega_con_dobleLinkedList > resultadoBusqueda.txt

```
1  Jul 23 22:21:00 144.96.120.61:5066 Failed password for root
2  Jun 1 05:10:59 144.82.578.19:4054 Failed password for admin
3  Oct 30 12:22:00 144.54.226.84:5530 Failed password for root
4  Jul 2 01:54:38 144.46.993.63:4842 Failed password for illegal user guest
5  Oct 14 07:37:39 144.27.435.63:5858 Failed password for illegal user test
6  Jun 4 00:48:13 144.22.468.3:4245 Illegal user
7  Oct 22 23:24:43 144.10.85.90:6065 Illegal user
8  Aug 12 10:58:12 143.79.346.23:5581 Failed password for illegal user guest
9  Jun 30 10:01:12 143.77.495.12:6890 Failed password for admin
10 Aug 17 02:33:43 143.63.855.89:4365 Failed password for root
11 Jul 11 15:21:02 143.50.302.83:5129 Failed password for root
12 Jun 17 13:54:53 143.44.380.40:6010 Failed password for illegal user test
13 Sep 28 06:57:58 143.17.174.46:6715 Illegal user
14 Jun 29 19:20:42 143.2.76.73:5789 Failed password for admin
15 Jul 24 14:20:33 142.80.588.7:5411 Illegal user
16 Oct 29 17:05:11 142.78.17.6:5970 Failed password for root
17 Sep 12 13:37:47 142.73.830.62:5935 Illegal user
```

```
760   Oct 8 02:01:14 56.79.687.68:5834 Illegal user
761   Sep 22 23:54:08 56.74.138.97:5196 Illegal user
762   Aug 29 13:55:51 56.73.485.71:5820 Failed password for illegal user guest
763   Oct 5 04:17:42 56.68.621.83:4411 Failed password for root
764   Aug 21 07:36:40 56.55.750.44:4799 Failed password for illegal user guest
765   Sep 8 16:16:54 56.43.265.62:6041 Illegal user
766   Aug 1 06:58:36 56.38.783.91:4739 Illegal user
767   Oct 10 04:27:46 56.27.718.12:4749 Illegal user
768   Jul 19 13:47:34 56.25.203.76:5744 Failed password for illegal user guest
769   Jul 1 12:42:25 56.10.792.87:4100 Illegal user
770   Oct 14 01:29:22 55.96.953.60:4612 Failed password for admin
771   Aug 2 07:36:00 55.75.129.71:6989 Failed password for illegal user test
772   Jul 20 22:44:56 55.59.611.97:4881 Illegal user
773   Oct 25 04:26:44 55.26.420.70:5088 Failed password for admin
774   Sep 26 02:21:58 55.11.803.4:4709 Failed password for illegal user guest
775   Jul 21 23:25:40 55.9.236.91:5565 Failed password for illegal user guest
776   Sep 26 10:13:55 55.2.252.2:6498 Failed password for root
777
```

En este caso de prueba se muestran los extremos de la búsqueda de un rango de Ip y se nota que efectivamente, los datos dentro del resultado de búsqueda de la prueba están dentro del rango esperado