Actividad 2.3 - Actividad Integral estructura de datos lineales

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

Thomas Freund Paternostro //A00831997

Alumnos:

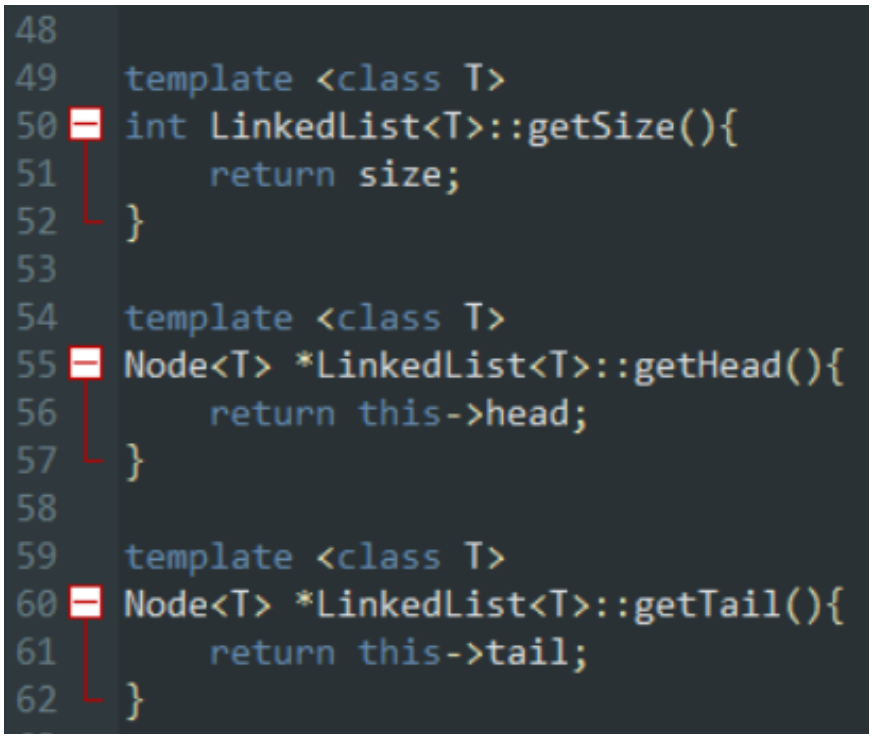
José Ángel Rentería Campos //A00832436

Santiago Andrés Serrano Vacca //A01734988

Thomas Freund Paternostro //A00831997

Fecha de entrega:

08/10/2021



```
48
49  template <class T>
50  int LinkedList<T>::getSize(){
51      return size;
52  }
53
54  template <class T>
55  Node<T> *LinkedList<T>::getHead(){
56      return this->head;
57  }
58
59  template <class T>
60  Node<T> *LinkedList<T>::getTail(){
61      return this->tail;
62  }
```

Línea	Costo	Repeticiones (peor caso)
51	C1	1

Línea	Costo	Repeticiones (peor caso)
56	C1	1

Línea	Costo	Repeticiones (peor caso)
61	C1	1

$$T(n) = C1 * 1$$

$$T(n) = c1 = C = 1$$

$$Complejidad: O\ (1)$$

```
67  template <class T>
68  Node<T> *LinkedList<T>::getSpecificNode(int index){
69      if(index < (size-1 - index)){
70          Node<T>* current = head;
71          int i = 0;
72          while(current != nullptr){
73              if(i == index){
74                  return current;
75              }
76              current = current->getNext();
77              i++;
78          }
79      } else {
80          Node<T>* current = tail;
81          int i = size-1;
82          while(current != nullptr){
83              if(i == index){
84                  return current;
85              }
86              current = current->getPrev();
87              i--;
88          }
89      }
90
91      return nullptr;
92  }
```

Línea	Costo	Repeticiones (peor caso)
53	C1	1
54	C2	1
55	C3	1
56	C4	n
58	C5	n
59	C6	n
60	C7	n

$$T(n) = C1 + C2 + C3(n) + C4(n) + C5(n) + C6(n) + C7(n) + C8$$
$$T(n) = C1 + C2 + C3 + C4n + C5n + C6n + C7n$$
$$T(n) = (C4 + C5 + C6 + C7)n + (C1 + C2 + C3 + C7 + C8)$$
$$a = C4 + C5 + C6 + C7,\ b = C1 + C2 + C3 + C7 + C8$$
$$T(n) = an + b$$

Dado que se evalúa el peor caso y se lleva al límite donde $\lim_{n \to \infty}$. b se vuelve insignificante.

$$T(n) = an$$
$$Complejidad: O\ (an) = O\ (n)$$

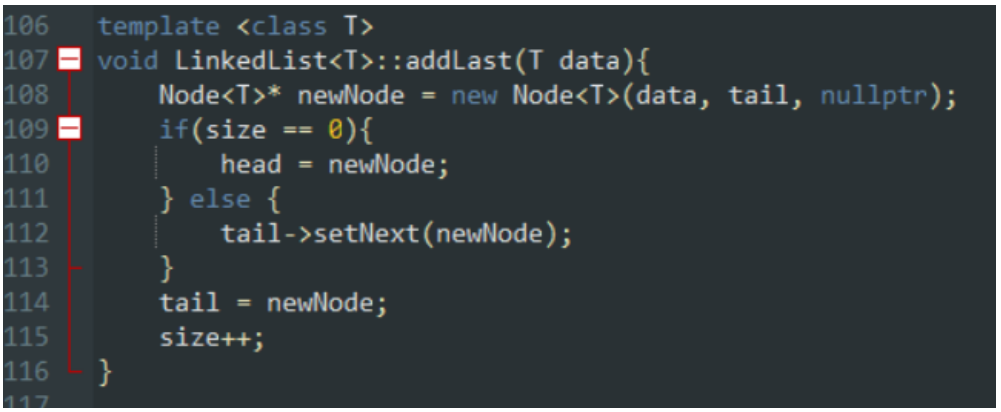
```
94  template <class T>
95  void LinkedList<T>::addFirst(T data){
96      Node<T>* newNode = new Node<T>(data, nullptr, head);
97      if(size == 0){
98          tail = newNode;
99      } else {
100         head->setPrev(newNode);
101     }
102     head = newNode;
103     size++;
104 }
```

Línea	Comp	Caso
96	1	C1
97	1	C2
98	1	C3
100	1	C4
102	1	C5
103	1	C6

$$T(n) = C1 * 1 + C2 * 1 + C3 * 1 + C4 * 1 + C5 * 1 + C6 * 1$$

$$T(n) = c1 + c2 + c3 + c4 + c5 + c6 = C = 1$$

$$Complejidad: O\ (1)$$



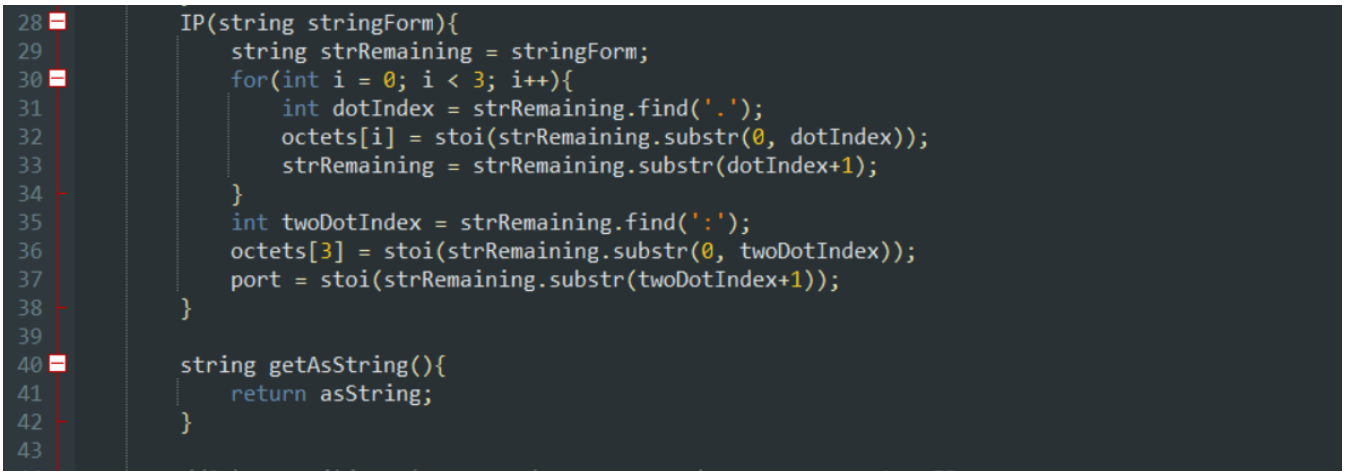
```
106  template <class T>
107  void LinkedList<T>::addLast(T data){
108      Node<T>* newNode = new Node<T>(data, tail, nullptr);
109      if(size == 0){
110          head = newNode;
111      } else {
112          tail->setNext(newNode);
113      }
114      tail = newNode;
115      size++;
116  }
```

Línea	Comp	Caso
108	1	C1
109	1	C2
110	1	C3
112	1	C4
114	1	C5
105	1	C6

$$T(n) = C1 * 1 + C2 * 1 + C3 * 1 + C4 * 1 + C5 * 1 + C6 * 1$$

$$T(n) = c1 + c2 + c3 + c4 + c5 + c6 = C = 1$$

$$Complejidad: O(1)$$



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

Línea	Costo	Repeticiones (peor caso)
29	C1	1
30	C2	3
31	C3	1
32	C4	1
34	C5	1
34	C6	1
35	C7	1
36	C4	1
37	C5	1
40	C6	1
41	C7	1

$$T(n) = C1 + C2 * 3 + C3 + C4 + C5 + C6 + C7 + C8 = C = 1$$

$$Complejidad: O\ (1)$$

```
160 Node<Bitacora>* buscarPorIPSecuencial(LinkedList<Bitacora>& bitacoras, IP ipABuscar, bool empezarDesdeFinal){
161     Node<Bitacora>* current = empezarDesdeFinal ? bitacoras.getTail() : bitacoras.getHead();
162     int i = empezarDesdeFinal ? bitacoras.getSize()-1 : 0;
163 
164     if(empezarDesdeFinal){
165         while(current != nullptr){
166             if(current->getData().ip <= ipABuscar){
167                 return current;
168             }
169             i--;
170             current = current->getPrev();
171         }
172     } else {
173         while(current != nullptr){
174             if(current->getData().ip >= ipABuscar){
175                 return current;
176             }
177             i++;
178             current = current->getNext();
179         }
180     }
181     return nullptr;
182 }
```

Línea	Costo	Repeticiones (peor caso)
161	C1	1
162	C2	1
164	C3	1
165	C4	n
166	C5	n
167	C6	n
169	C7	n
170	C8	n
172	C9	1
173	C10	n
174	C11	n
175	C12	n
177	C13	n
178	C14	n
181	C15	1

$$T(n) = C1 + C2 + C3 + C4(n) + C5(n) + C6(n) + C7(n) + C8(n) + C9 + C10(n) + C11(n) + C12(n) + C13(n) + C14(n) + C15$$

$$T(n) = C1 + C2 + C3 * n + C4 * n + C5 * n + C6 * n + C7 * n + C8 * n + C9 + C10 * n + C11 * n + C12 * n + C13 * n + C14 * n + C15$$

$$T(n) = (C4 + C5 + C6 + C7 + C8 + C11 + C12 + C13 + C14)n + (C1 + C2 + C3 + C9 + C15)$$

$$a = C4 + C5 + C6 + C7 + C8 + C11 + C12 + C13 + C14,\ b = C1 + C2 + C3 + C9 + C15$$

$$T(n) = an + b$$

Dado que se evalúa el peor caso y se lleva al límite donde $\lim_{n \to \infty}$, b se vuelve insignificante.

$$T(n) = an$$

$$Complejidad: O(an) = O(n)$$

```
143  void quickSort(LinkedList<Bitacora> &v, int inicio, int fin)
144  {
145    if (inicio < fin)
146    {
147      int p = Particion(v, inicio, fin);
148      quickSort(v, inicio, p - 1);
149      quickSort(v, p + 1, fin);
150    }
151  }
```

Al ser el quicksort un método de ordenamiento recursivo, tenemos que:

$$T(n) = n + T(n-1)$$
$$T(n-1) = (n-1) + T(n-2)$$
$$T(n-2) = (n-2) + T(n-3)$$
$$T(n-3) = (n-3) + T(n-4)$$
$$\dots$$
$$T(3) = 3 + T(2)$$
$$T(2) = 2 + T(1)$$
$$T(1) = 0$$

Entonces…

$$T(n) = n + (n-1) + (n-2) + (n-3) + (n-4)\dots + 3 + 2$$
$$= 1/2(n(n+1)) - 1 = O(n^2)$$

```
117  int Particion(LinkedList<Bitacora> &v, int inicio, int fin)
118  {
119      int j = inicio;
120
121      Node<Bitacora>* nodeAtI = v.getSpecificNode(inicio);
122      Node<Bitacora>* nodeAtMitad = v.getSpecificNode(fin);
123      Node<Bitacora>* nodeAtJ = nodeAtI;
124
125    for (int i = inicio; i < fin; ++i)
126    {
127      //si la IP del Ã-ndice actual es menor a la del medio
128      if (nodeAtI->getData() < nodeAtMitad->getData())
129        {
130          //swap con el incial
131          swapNodeData(nodeAtI, nodeAtJ);
132          ++j;
133          nodeAtJ = nodeAtJ->getNext();
134        }
135
136      nodeAtI = nodeAtI->getNext();
137    }
138    //swap con el de la mitad
139    swapNodeData(nodeAtJ, nodeAtMitad);
140    return j;
141  }
```

Línea	Costo	Repeticiones (peor caso)
119	C1	1
121	C2	1
122	C3	1
123	C4	1
125	C5	n+1
128	C6	n
131	C7	n
132	C8	n
133	C9	n
136	C10	n
139	C11	1
140	C12	1

$$T(n) = C1 + C2 + C3 + C4 + C5(n+1) + C6(n) + C7(n) + C8(n) + C9(n) + C10(n) + C11 + C12$$

$$T(n) = C1 + C2 + C3 + C4 + C5 * n + C5 + C6 * n + C7 * n + C8 * n + C9 * n + C10 * n + C11 + C12$$

$$T(n) = (C5 + C6 + C7 + C8 + C9 + C10)n + (C1 + C2 + C3 + C4 + C5 + C11 + C12)$$

Complejidad: **O(n)**

Incluir de manera específica si el uso de un lista doblemente ligada para esta situación problema es la más adecuada o no así como sus ventajas y desventajas

La lista doblemente ligada tiene como ventaja poder retroceder y moverse a lo largo de esta, tanto de principio a fin como de fin a principio, además de colocar y crear memoria acorde a la necesidad que se requiera, sin acaparar más de la necesaria. Se trabajó para hacer el arreglo utilizando un algoritmo quicksort, el cual en el caso promedio es O(n log n) y el peor caso es O(n^2). No consideramos que en el peor caso de la función Partición que tiene complejidad O(n) afecte la eficiencia o procesamiento de los archivos de texto. En donde claramente estamos trabajando con un algoritmo deficiente, sin embargo encontramos que para la carga de 16807 IPs, la velocidad de ordenamiento supera lo esperado.

La gran desventaja de las listas ligadas es el tiempo que toma acceder a uno de sus elementos por índice. Para, por ejemplo, acceder al quinto elemento de la lista, tendríamos que iterar 5 veces, accediendo a los 4 nodos anteriores al quinto.

Aunque, aprovechando que se trata de una lista doblemente ligada, es posible comenzar a buscar desde el final (para, por ejemplo, obtener el elemento 99 de una lista de 100 por medio de 2 iteraciones y no de 99), sigue siendo supremamente eficiente, por ejemplo, acceder al elemento 50. Esta ineficiencia se va volviendo cada vez más preocupante conforme crece el tamaño de la lista.

El problema aquí es que algoritmos de ordenamiento como QuickSort requieren acceder repetidamente a elementos por su índice, lo cual ralentiza enormemente el proceso de ordenamiento, especialmente en listas muy largas. Es por esto que, para efectos de ordenamiento de datos, las listas doblemente ligadas son altisimamente ineficientes.

Una ventaja de la lista doblemente ligada como fue previamente mencionado es que puede analizar una cantidad indefinida de nodos, a base de la cantidad de líneas leídas en un documento; esto le permite al programa tener la flexibilidad de trabajar con diferentes tipos de archivos y, mientras estén ordenados con la estructura predefinida, sea posible presentar una solución que uno o varios individuos tardarían una cantidad exponencialmente grande de tiempo en producir y comprobar su veracidad. Sin embargo, esta ventaja será provechosa solo en el caso de que la información no se tenga que ordenar.

Reflexión:
En esta actividad se trabajó con una lista doblemente ligada y se utilizó uno de los ordenamientos para trabajar con ips. Fuera del contexto de esta actividad se consideró que esta estructura de datos es muy ineficiente o no es la apropiada para hacer un ordenamiento. Esto se debe a la flexibilidad pero también a la rigidez que tiene una lista entrelazada doble. Independiente a eso se pudo entender a profundidad cómo los apuntadores y trabajar cómo hacer las modificaciones al archivo 1.3 para poder correr este código de manera eficiente.