Actividad 3.4 - Actividad Integral de BST (Evidencia Competencia)

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

Alumnos:

José Ángel Rentería Campos //A00832436

Santiago Andrés Serrano Vacca //A01734988

Thomas Freund Paternostro //A00831997

Fecha de entrega:

22/10/2021

addIPToAccessNum:

```
18      void addIPToAccessNum(IP ip, int accessNum){
19          Node *current = root;
20          Node *father = nullptr;
21
22          while(current != nullptr){
23              if(accessNum == current->getAccessNum()){
24                  current->addIP(ip);
25                  return;
26              }
27
28              father = current;
29              current = (accessNum > current->getAccessNum()) ? current->getRight() : current->getLeft();
30          }
31
32          if(father == nullptr){
33              root = new Node(accessNum, {ip});
34          } else {
35              (father->getAccessNum() > accessNum)? father->setLeft(new Node(accessNum, {ip})) : father->setRight(new Node(accessNum, {ip}));
36          }
37      }
```

Línea	Costo	Repeticiones (peor caso)
19	C1	1
20	C2	1
22	C3	h
23	C4	h
24	C5	h
25	C6	h
28	C7	h
29	C8	h
32	C9	1
33	C10	1
35	C11	1

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

$$T(h) = (C3 + C4 + C5 + C6 + C7 + C8)h + (C1 + C2 + C3 + C9 + C10 + C11)$$
$$a = C3 + C4 + C5 + C6 + C7 + C8,\ b = C1 + C2 + C3 + C9 + C10 + C11$$
$$T(h) = ah + b$$

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

$$T(h) = ah$$
$$Complejidad: O(ah) = O(h)$$

O(h) es la altura del árbol donde h <= n, n es la cota superior.

visitByInverseInorder:

```
139         void visitByInverseInorder(Node* node){
140             if(node == nullptr)
141                 return;
142 
143             visitByInverseInorder(node->getRight());
144 
145             cout << node->getAccessNum() << " accesos: " << endl;
146             for(IP ip : node->getIPs()){
147                 cout << ip.getAsStringNoPort() << endl;
148             }
149             cout << "------------------------------" << endl;
150 
151             visitByInverseInorder(node->getLeft());
152         }
```

Línea	Costo	Repeticiones (peor caso)
140	C1	1
141	C2	1
143	C3	1
145	C4	1
146	C5	n
147	C6	n
149	C7	1
151	C8	1

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

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

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

printXIPsWithMoreAccesses:

```
197     void printXIPsWithMoreAccesses(int &x, Node* node){
198         if(node == nullptr)
199             return;
200 
201         printXIPsWithMoreAccesses(x, node->getRight());
202 
203         for(IP ip : node->getIPs()){
204             if(x <= 0) return;
205             cout << ip.getAsStringNoPort() << "-> " << node->getAccessNum() << " accesos" << endl;
206             x--;
207         }
208 
209         printXIPsWithMoreAccesses(x, node->getLeft());
210     }
```

Línea	Comp	Caso
198	C1	1
199	C2	1
201	C3	1
204	C4	n
205	C5	n
206	C6	n
207	C7	n

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

$$a = C1 + C2 + C3,\ b = C4 + C5 + C6 + C7$$
$$T(n) = an + b$$

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

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

Ciclo que manda las IPs con su número de repeticiones a su respectivo BST:

```
31      //Ahora creamos el BST, donde:
32      //Key: número de repeticiones
33      //Value: IPs que se repiten ese número de veces
34      BST repeticiones;
35
36      //Vamos populando el BST:
37      int eqIPCount; //Equal IP Count
38      IP prevIP;
39
40      for(IP ip : ips){
41          if(prevIP != ip){
42              if(prevIP != IP())
43                  repeticiones.addIPToAccessNum(prevIP, eqIPCount);
44              eqIPCount = 1;
45          } else {
46              eqIPCount++;
47          }
48          prevIP = ip;
49      }
```

Línea	Comp	Repeticiones(peor caso)
34	C1	1
37	C2	1
38	C3	1
40	C4	n
41	C5	n
42	C6	n
43	h	n
44	C8	n
45	C9	n
46	C10	n
48	C11	n

$$T(h,\ n) = C1 + C2 + C3 + C4(n) + C5(n) + C6(n) + h(n) + C8(n) + C9(n) + C10(n) + C11(n)$$

$$T(h,\ n) = (C4 + C5 + C6 + C8 + C9 + C10 + C11)n + (C1 + C2 + C3) + hn$$

$$a = C4 + C5 + C6 + C7 + C8 + C9 + C10 + C11,\ b = C1 + C2 + C3$$

$$T(h,\ n) = an + b + hn$$

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

$$T(h,\ n) = hn$$

$$Complejidad\text{: } O\ (hn) \simeq O(n\ log\ n)$$

O(h) es la altura del árbol donde h <= n, n es la cota superior.

FUNCIONES NUEVAS (Explicación):

Esta función sirve para añadir una dirección IP al BST, teniendo en cuenta su número de accesos. En el caso de que ya exista un nodo con su número de accesos, simplemente añadimos esa dirección IP al nodo (cada nodo tiene un vector con sus direcciones IP; por ejemplo, el nodo con key 35 tiene un vector con todas las direcciones IP que salen 35 veces). En caso de que no exista un nodo con el número de accesos de esta IP, lo creamos.

```
18      void addIPToAccessNum(IP ip, int accessNum){
19          Node *current = root;
20          Node *father = nullptr;
21
22          while(current != nullptr){
23              if(accessNum == current->getAccessNum()){
24                  current->addIP(ip);
25                  return;
26              }
27
28              father = current;
29              current = (accessNum > current->getAccessNum()) ? current->getRight() : current->getLeft();
30          }
31
32          if(father == nullptr){
33              root = new Node(accessNum, {ip});
34          } else {
35              (father->getAccessNum() > accessNum)? father->setLeft(new Node(accessNum, {ip})) : father->setRight(new Node(accessNum, {ip}));
36          }
37      }
```

Esta función traversa el BST en inorden inverso (o reverse inorder, como se le conoce en inglés). Sirve para imprimir las IPs en forma descendente por cantidad de accesos.

```
139         void visitByInverseInorder(Node* node){
140             if(node == nullptr)
141                 return;
142
143             visitByInverseInorder(node->getRight());
144
145             cout << node->getAccessNum() << " accesos: " << endl;
146             for(IP ip : node->getIPs()){
147                 cout << ip.getAsStringNoPort() << endl;
148             }
149             cout << "------------------------------" << endl;
150
151             visitByInverseInorder(node->getLeft());
152         }
```

Esta función imprime las *x* direcciones IP con más accesos, en orden descendente por número de accesos.

```
197     void printXIPsWithMoreAccesses(int &x, Node* node){
198         if(node == nullptr)
199             return;
200 
201         printXIPsWithMoreAccesses(x, node->getRight());
202 
203         for(IP ip : node->getIPs()){
204             if(x <= 0) return;
205             cout << ip.getAsStringNoPort() << "-> " << node->getAccessNum() << " accesos" << endl;
206             x--;
207         }
208 
209         printXIPsWithMoreAccesses(x, node->getLeft());
210     }
```

Esta no es una función, pero de todos modos es importante. Con este código vamos iterando a través del vector con las IPs, ya ordenado previamente, y vamos contando cuántas veces se repite una IP. Cuando la IP deja de repetirse, llamamos a *addIPToAccessNum(...)* para añadir la información al BST. Posteriormente, reiniciamos el contador para ir contando ahora cuántas veces se repite la siguiente IP, y así hasta terminar con todo el vector.

```
31      //Ahora creamos el BST, donde:
32      //Key: número de repeticiones
33      //Value: IPs que se repiten ese número de veces
34      BST repeticiones;
35 
36      //Vamos populando el BST:
37      int eqIPCount; //Equal IP Count
38      IP prevIP;
39 
40      for(IP ip : ips){
41          if(prevIP != ip){
42              if(prevIP != IP())
43                  repeticiones.addIPToAccessNum(prevIP, eqIPCount);
44              eqIPCount = 1;
45          } else {
46              eqIPCount++;
47          }
48          prevIP = ip;
49      }
```