AVL

Colectivo Estructuras de Datos y Algoritmos

Noviembre 2024

1. Determine si las siguientes proposiciones son verdaderas(**V**) o falsas(**F**). Argumente su respuesta en cada caso.

   a) ___ La menor cantidad de nodos necesarios para construir un árbol **AVL** de altura 6 es 34.

   b) ___ La mayor altura de un árbol **AVL** con 10 nodos es 3.

   c) ___ El recorrido entreorden en un árbol **AVL** es $\Theta(n)$.

   d) ___ En todo árbol **AVL** la diferencia entre la longitud del camino más largo desde la raíz hasta una hoja y la longitud del camino más corto desde la raíz hasta una hoja es siempre menor o igual a 2.

2. Diseñe un algoritmo que dado un árbol binario de búsqueda T determine si es un árbol **AVL**. La complejidad temporal del algoritmo debe ser $O(n)$, donde n es la cantidad de nodos en T.

3. Dada una lista ordenada L de n elementos diseñe un algoritmo que construya un árbol **AVL** que posea todos los elementos de L. La complejidad temporal del algoritmo debe ser $O(n)$.

4. Implemente los métodos **Insert(T, key)** y **Erase(T, key)** para árboles **AVL**. La complejidad temporal de cada método debe ser $O(\log n)$, donde n es la cantidad de nodos en T. Muestre cómo actualizar correctamente la propiedad *size* de los nodos involucrados luego de una inserción/eliminación.

5. Sea la estructura de datos **Set** que simula el comportamiento de un conjunto con las funciones:
   Find(S, v): Devuelve *True* si $v \in S$, *False* en caso contrario.
   Insert(S, v): Añade el elemento v a S si $v \notin S$, hace nada en caso contrario.
   Erase(S, v): Elimina el elemento v de S si $v \in S$, hace nada en caso contrario.
   Rank(S, v): $|\{u \mid u \in S \wedge u < v\}|$ (Devuelve la cantidad de elementos $u \in S$ que sean menores que v).
   Select(S, k): Devuelve $u \in S$ con exactamente k elementos menores en S, o *null* si u no existe.
   SumLessThan(S, v): Devuelve la suma de todos los elementos $u \in S$ que sean menores que v.

   Diseñe los métodos de la estructura de datos **Set** con complejidad temporal $O(\log |S|)$.

6. Sea la estructura de datos **MultiSet** que simula el comportamiento de un multiconjunto con las funciones:
   Find(M, v): Devuelve *True* si $v \in M$, *False* en caso contrario.
   Insert(M, v): Añade el elemento v a M.
   Erase(M, v): Elimina una ocurrencia del elemento v en M si $v \in M$, hace nada en caso contrario.
   Rank(M, v): $|\{u \mid u \in M \wedge u < v\}|$ (Devuelve la cantidad de elementos $u \in M$ que sean menores que v).
   Select(M, k): Devuelve $u \in M$ con exactamente k elementos distintos menores en M, o *null* si u no existe.
   Kth(M, k): Devuelve el k-ésimo menor elemento en M. Se asume que $0 \leq k < |M|$.
   SumLessThan(M, v): Devuelve la suma de todos los elementos $u \in M$ que sean menores que v.

   Diseñe los métodos de la estructura de datos **MultiSet** con complejidad temporal $O(\log |M|)$.

   Respuesta

   Sea M la estructura de datos tipo **MultiSet**, M tiene el atributo *myavl* de tipo *AVL* cuyo objetivo es almacenar los elementos del multiconjunto.

   Cada nodo T de *myavl* tiene las siguientes propiedades:

   key : Llave del nodo (En un **AVL** todas las llaves son distintas pues es un **ABB**)

   left : **AVL** hijo izquierdo de T (*null* en caso de no existir).

right : **AVL** hijo derecho de T (*null* en caso de no existir).

size : Número de nodos en el árbol ($T.size = |T|$)

amount : Cantidad de veces que el elemento *key* está en M.

total : Suma de *amount* de los nodos en el árbol.

sum : Suma de $key \times amount$ de los nodos en el árbol.

En la guía solo se ejemplifica el método **Select**, sin embargo, mostrando como actualizar las propiedades exclusivas de *myavl* que no están en un **ABB** es posible resolver el resto de las funciones.

```
1      def Select(T, k):
2          if k < 0 or k >= |T|:
3              return null
4          sizeLeft = |T.left|
5          if k = sizeLeft:
6              return T.key
7          if k < sizeLeft:
8              return Select(T.left, k)
9          return Select(T.right, k − sizeLeft − 1)
```

El pseudocódigo aprovecha la propiedad del orden relativo de las llaves en un **ABB** (un **AVL** es un **ABB**) para calcular el elemento correcto. En esencia es el mismo método para un **ABB** mostrado en la clase práctica pasada con la particularidad de que los extremos son desconocidos.

La complejidad temporal es $O(h)$, donde h es la altura de *myavl* y, por lo tanto, es $O(\log |myavl|)$ dado que la altura de un **AVL** es $\Theta(\log n)$, donde n es la cantidad de elementos del **AVL**. Como $|myavl| \leq |M|$ entonces la complejidad temporal es a su vez $O(\log |M|)$.

7. Sea la estructura de datos **Dictionary** que maneja pares <llave, valor> con las funciones:
Get(D, key): Devuelve el valor asociado a la llave *key* en D. Retorna *null* si la llave *key* no existe en D.
Set(D, key, value): Actualiza en D, o crea en caso que no exista, el valor asociado a la llave *key* con *value*.
Diseñe los métodos de la estructura de datos **Dictionary** con complejidad temporal $O(\log |D|)$.

Respuesta

Sea D la estructura de datos tipo **Dictionary**, D tiene el atributo *myavl* de tipo **AVL** cuyo objetivo es almacenar los pares <llave, valor> que sean añadidos/actualizados en D. Los nodos en *myavl* poseen las propiedades **key** y **value**. La adición de *value* no afecta la correctitud del **AVL** ni la complejidad temporal y espacial de sus métodos pues el valor solo es una variable relevante a un dato en el nodo y no interfiere o es afectado por las rotaciones. Además, se asumen implementados los métodos **Search**, **Insert** y **Erase** de **AVL** vistos en conferencias. El método **Insert** recibe un parámetro adicional asociado a la propiedad *value* que es usado solamente en la creación de un nuevo nodo. El uso de tales métodos del **AVL** permite la resolución de las funciones **Get** y **Set** como muestran los pseudocódigos a continuación.

```
1      def Get(D, key):
2          nodo = Search(myavl, key)
3          if nodo = null:
4              return null
5          else:
6              return nodo.value
```

El método **Search** de **ABB** busca la llave *key* en *myavl*, si existe devuelve el nodo que la contiene

pudiendo obtener así el valor asociado a la llave, y en otro caso devuelve *null* que indica la no existencia de la llave en la estructura.
La complejidad temporal de **Search** es $O(\log |D|)$ porque es $O(h)$ donde h al ser la altura del árbol que es un **AVL** es $\Theta(\log |myavl)$ y $|myavl| = |D|$. Luego, la complejidad temporal del método **Get** es $O(\log |D|)$ dado que hace un llamado a la función **Search** y posteriormente deduce la respuesta en $O(1)$ en dependencia de la nulidad del resultado de la llamada.

```
1          def Set(D, key, value):
2              node = Search(myavl, key)
3              if node = null:
4                  Insert(myavl, key, value)
5              else:
6                  node.value = value
```

El método **Set** cumple la invariante de conservar actualizados los pares <llave, valor> en la estructura de datos D y, por consiguiente, en *myavl*. Una llave es usada por primera vez en el método **Get** si y solo si no existe un nodo que la posea en *myavl* y, por tanto, hay que crearlo para almacenar el nuevo par <llave, valor>. En el caso de que la llave sea usada por segunda vez o posteriores entonces el valor anterior asociado a la llave es obsoleto por lo que se puede reutilizar el nodo que posee la llave en *myavl* y actualizar el campo *value* con el nuevo valor.
En el pseudocódigo de **Set** se realiza un llamado a la función **Search** ($O(\log |D|)$ por la explicación anterior) y luego, en dependencia de la nulidad del resultado de la llamada, se procede a resolver uno de los casos mencionados anteriormente (la creación del nodo (inserción) con una complejidad temporal $O(\log |D|)$ o la actualización en $O(1)$). Luego, por la regla de la suma, la complejidad temporal del método **Set** es $O(\log |D|)$.

8. Sea la estructura de datos **ValueSortedDictionary** que maneja pares <llave, valor> con las funciones:
Get(D, key): Devuelve el valor asociado a la llave *key* en D. Retorna *null* si la llave *key* no existe en D.
Set(D, key, value): Actualiza en D, o crea en caso que no exista, el valor asociado a la llave *key* con *value*.
Max(D): $\max\{v \mid < k, v > \in D\}$ (Devuelve el mayor valor en D).
Min(D): $\min\{v \mid < k, v > \in D\}$ (Devuelve el menor valor en D).

   Diseñe la estructura de datos **ValueSortedDictionary** del tal forma que los métodos **Get** y **Set**tengan complejidad temporal $O(\log |D|)$ y los métodos **Max** y **Min** tengan complejidad temporal $O(1)$.

9. Dada una lista L de n elementos. Sea $d_{i,j} = |i - j|$ para $1 \leq i, j \leq n$ encontrar $max\ d_{i,j}$ tal que $L_i = L_j$. (Encontrar la mayor distancia entre 2 elementos iguales). La complejidad temporal del algoritmo debe ser $O(n \log n)$.

Respuesta

Para solucionar el problema sólo importa la distancia entre los índices de aquellos elementos con el mismo valor. Por lo tanto, se puede usar una instancia D de **Dictionary** que almacene para cada valor de la lista L, la lista de índices en los que aparece, luego por cada valor de L se calcula la distancia entre los índices de los elementos con ese valor y se actualiza el máximo de las distancias calculadas. Pero esta solución tendría una complejidad temporal de $\Omega(n^2)$, lo cual no cumple con la restricción dada.

Se necesita un enfoque diferente al problema, para ello es importante preguntarse:

- ¿Para un índice j con $L_j = x$ realmente se necesita analizar todos los índices i con $L_i = x$?

La respuesta es no, ya que solo se necesita el índice i más lejano a j (en otras palabras el menor índice i tal que $L_i = L_j$).

Demostración:
Sean i, j dos índices tales que $L_i = L_j = x$ (sin pérdida de generalidad asumamos que $i < j$) y k un índice tal que $L_k = x$ y $i < k < j$. Entonces, la distancia entre i y j es $j - i$ y la distancia entre k y j

es $j-k$. Como $k > i$ entonces $j-k < j-i$ lo que implica que la distancia entre k y j es menor que la distancia entre i y j.

Por lo demostrado anteriormente, para un mismo valor x solo se necesita almacenar el primer índice en el que aparece porque otro valor solo minimizaría la respuesta. Por lo que solo es necesario almacenar en el diccionario D un solo índice (el primer índice en el que aparece el valor).
y el problema se vuelve más simple de resolver.

Por último, se debe recorrer la lista de izquierda a derecha para encontrar la primera ocurrencia de cada valor. Luego actualizar la distancia entre el índice actual y el índice de la primera ocurrencia.

Pseudocódigo:

```
1  def max_distance(L):
2      ans = 0
3      D = Dictionary()
4      for i = 0 to |L| − 1:
5          furthest_index = Get(D, L[i])
6          if furthest_index is null:
7              Set(D, L[i], i)
8          else:
9              ans = max(ans, i − furthest_index)
10     return ans
```

Insertar y buscar de **Dictionary** tiene complejidad temporal $O(\log n)$, por lo que la complejidad temporal del algoritmo es $O(n \log n)$.

Nota: Alternativamente basado en la demostración anterior podemos almacenar el primer y el último índice en el que aparece el valor x en la lista L. Luego, para cada valor x se calcula la distancia entre el primer y el último índice en el diccionario D y se actualiza el máximo de las distancias calculadas.

10. Dada una lista $L = L_1, L_2, ..., L_n$ de enteros. Diseñe un algoritmo que cuente la cantidad de pares de índices i, j con $i < j$ tales que $\frac{L_i}{j} = \frac{L_j}{i}$. La complejidad de su algoritmo debe ser $O(n \log n)$

Respuesta

Nótese que la igualdad $\frac{L_i}{j} = \frac{L_j}{i}$ es equivalente a $L_i \cdot i = L_j \cdot j$. Por lo tanto, se puede almacenar en una instancia D de **Dictionary** la cantidad de veces que aparece el producto $L_i \cdot i$ para cada índice i en la lista L.
Luego por cada índice j de la lista se busca en el diccionario D la cantidad de veces que aparece el producto $L_j \cdot j$, se suma a la respuesta y se actualiza en el diccionario D la frecuencia del producto $L_j \cdot j$ para que contribuya a la cantidad que aportarán a la respuesta los posibles valores $k > j$ tales que $L_k \cdot k = L_j \cdot j$.

Pseudocódigo:

```
def count_pairs(L):
    ans = 0
    D = Dictionary()
    for j = 1 to |L|:
        cur_value = Get(D, L[j] * j)
        if cur_value is not null:
            ans += cur_value
            Set(D, L[j] * j, cur_value + 1)
        else:
            Set(D, L[j] * j, 1)
    return ans
```

Buscar, obtener y actualizar de **Dictionary** tiene complejidad temporal $O(\log n)$, por lo que la complejidad temporal del algoritmo es $O(n \log n)$.

11. Una inversión en una lista L de elementos ordenables, es un par ordenado i, j de índices de L, tales que $i < j$ y $L[i] > L[j]$. Diseñe un algoritmo que dada una lista de enteros L, devuelva la cantidad de inversiones en dicha lista. La complejidad temporal de su algoritmo debe ser $O(n \log n)$, siendo $n = |L|$.

Respuesta

Matemáticamente el problema consiste en calcular

$$S = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} f(L[i], L[j]), \text{ donde } f(a,b) = \begin{cases} 1 & \text{si } a > b \\ 0 & \text{en otro caso} \end{cases}$$

Nótese que al evaluar cada término dentro de las sumatorias es posible calcular S con complejidad temporal $\Omega(n^2)$ lo que tal solución no cumple con la restricción de complejidad temporal. Sin embargo, nótese que para un i fijo el valor de $T_i = \sum_{j=i+1}^{n-1} f(L[i], L[j])$ es igual a la cantidad de elementos menores que $L[i]$ a la derecha de la i-ésima posición en L. Esta observación y una instancia M de **Multiset** permiten la resolución adecuada del ejercicio dado que, al mantener dentro de la estructura M todos los valores de las posiciones a la derecha de la posición i a analizar, la ejecución de **Rank(M, L[i])** devuelve exactamente T_i.
El multiconjunto M cuando se analizan las posiciones x y $x+1$, con $0 \leq x \leq n-2$, tiene solo un elemento de diferencia $L[x+1]$ (el resto son iguales) así que actualizar M de la posición x a la siguiente (anterior) consiste en eliminar (añadir) el elemento $L[x+1]$ de M con un costo temporal $O(\log n)$ (n es cota superior de $|M|$ en cualquier instante). Luego, al recorrer los elementos de izquierda a derecha (derecha a izquierda) y en cada posición i actualizar M consecuentemente en $O(\log n)$ y hallar el valor de T_i en $O(\log n)$ se obtiene el valor de $S = \sum_{i=0}^{n-1} T_i = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} f(L[i], L[j])$. La complejidad temporal del algoritmo es $O(n \log n)$.

El siguiente pseudocódigo usa la alternativa de insertar los elementos de derecha a izquierda.

```
def calculate_inversions(L):
    S = 0
    M = Multiset()
    for i = |L| − 1 to 0:
        Ti = Rank(M, L[i])
        S += Ti
        Insert(M, L[i])
    return S
```

Nótese que en el pseudocódigo se mantiene la invariante de que al analizar el índice i de la lista L se encuentran en M todos los elementos de L a la derecha de esa posición lo que permite calcular correctamente T_i usando **Rank** y posteriormente S.

12. Se tiene una lista L de números (inicialmente vacía). Sobre la cual se realizan Q consultas de tres tipos:

- **Insertar:** Se inserta un número x en la lista L (posiblemente repetido).
- **Eliminar:** Se elimina un número x de la lista L (se garantiza que el número ya está insertado en L).
- **Consultar:** Devolver la frecuencia del número que más se repite en L.

Su algoritmo debe tener una complejidad temporal de $O(Q \log Q)$.

Respuesta

Cada vez que se inserta o se elimina un número de la lista solamente cambia la frecuencia de un único valor. Para resolver este problema se utilizarán dos instancias M y MF de **Multiset**, M almacenará los valores que se encuentran actualmente en la lista L, mientras que MF almacenará las frecuencias de esos valores.

Si se mantiene actualizado correctamente MF lo que se necesita en cada **consulta** es la máxima llave de MF, pues sería precisamente la mayor de las frecuencias de los números que aparecen en L.

Para mantener actualizados los dos **Multisets** se necesita que cada vez que se inserte o elimine un número x de la lista L se actualice la frecuencia de x en M y se actualice el valor de la frecuencia de x en MF de la forma siguiente:

- Antes de insertar un número x en la lista L este aparece cierta cantidad de veces, sea esta cantidad fq_x entonces al insertar x en L también debemos hacerlo en M, como resultado de esto ahora x aparece $fq_x + 1$ veces por lo que debemos remover la cantidad de veces que aparecía antes (fq_x) de MF e insertar $fq_x + 1$ en MF.
- Análogamente si se elimina un número x de la lista L debemos remover una ocurrencia de x de M, así como también una ocurrencia de fq_x de MF e insertar $fq_x - 1$ en MF.

Entonces ya se tiene una forma de mantener la cantidad de ocurrencias de cada valor, así como la cantidad de ocurrencias de cada frecuencia para cada uno de esos valores. Solo queda pedir el máximo elemento de MF en cada **consulta**.

Pseudocódigo:

```
1  def solve(Queries):
2      answer = []
3      M = Multiset()
4      MF = Multiset()
5      for query in Queries:
6          if query.type = 'Insertar':
7              old_frequency = Search(M, query.x).amount
8              Erase(MF, old_frequency)
9              Insert(M, query.x)
10             new_frequency = Search(M, query.x).amount
11             Insert(MF, new_frequency)
12         elif query.type = 'Eliminar':
13             old_frequency = Search(M, query.x).amount
14             Erase(MF, old_frequency)
15             Erase(M, query.x)
16             new_frequency = Search(M, query.x).amount
17             Insert(MF, new_frequency)
18         elif query.type = 'Consultar':
19             answer.append(Max(MF))
20     return answer
```

Por cada una de las Q consultas se realizan tres operaciones de `Insert` o `Erase` en un **Multiset**, por

lo que cada una de ellas tiene una complejidad de $O(\log Q)$, como esto se realiza a lo sumo Q veces la complejidad temporal del algoritmo es $O(Q \log Q)$.

13. Dada una lista L y un número K devolver la mayor cantidad de elementos distintos en un subarreglo de tamaño K.

    La complejidad temporal de su algoritmo debe ser $O(n \log n)$, siendo $n = |L|$.

Respuesta

Obsérvese que la cantidad de elementos en los subarreglos deseados no varía por lo que se puede mantener una ventana de tamaño K mientras se itera por las posiciones de la lista.

Definamos por ventana en un arreglo o lista L como un conjunto de posiciones consecutivas $i_1, i_2, ..., i_k$ tales que $i_j + 1 = i_{j+1}$ y $k \leq |L|$

¿Cómo mantener correctamente calculada esta cantidad?

Si se usa una instancia M de **Multiset** que almacene los elementos de la ventana actual, al consultar el *size* de la raíz de M se tendría la cantidad de nodos del mismo, lo que es equivalente a la cantidad de elementos distintos en la ventana actual, debido a que en **AVL** no hay nodos con llaves repetidas.

¿Cómo podemos actualizar el conjunto al mover la ventana?

Al mover la ventana de izquierda a derecha se tiene que eliminar el primer elemento de la ventana de M y añadir el siguiente elemento de la lista L a M.
Nótese que el tamaño de la ventana cambió su tamaño en $-1 + 1 = 0$ (se eliminó un elemento y se añadió otro), por lo que solo se está iterando por los subarreglos de tamaño K.

Una vez que se tiene fijada la ventana de tamaño K solo queda actualizar la respuesta con la máxima cantidad de elementos distintos en cada una.

Pseudocódigo:

```
1  def max_distinct_elements(L, K):
2      ans = 0
3      M = Multiset()
4      for i = 0 to K − 1:
5          Insert(M, L[i])
6      ans = M.size
7      for i = K to |L| − 1:
8          Insert(M, L[i])
9          Erase(M, L[i − K])
10         ans = max(ans, M.size)
11     return ans
```

Veamos que cada elemento es insertado y eliminado del conjunto a lo sumo una vez, por lo que la complejidad temporal del algoritmo es $O(n \log n)$.

14. Sea S una cadena de ceros y unos, y $n = |S|$. Se harán m modificaciones del tipo: invertir el bit en la posición i de S. Luego de cada modificación se debe devolver la máxima cantidad de caracteres iguales consecutivos. Diseñe un algoritmo que resuelva el problema, la complejidad temporal del mismo debe ser $O((n + m) \log n)$.