Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de datos y algoritmos — 2020-2

Ayudantía 4

Pregunta 1

Demuestra que cualquier ABB de n claves puede ser transformado en cualquier otro ABB arbitrario con las mismas claves usando $O(n)$ rotaciones.

Pregunta 2

Escribe un algoritmo para calcular la cantidad de inversiones de un arreglo de largo n en tiempo $O(n \cdot \log(n))$.

Solución

Pregunta 1

Primero vamos a definir funciones que toman como argumento una key existente en el árbol y retorna otro árbol:

1. $i(k)$: Rotación a la izquierda en la llave k
2. $d(k)$: Rotación a la derecha en la llave k

Notamos que las inversas de estas funciones son:

1. $i^{-1}(k) = d(k)$
2. $d^{-1}(k) = i(k)$

Para esta solución primero vamos a convertir el árbol original en una lista ligada usando rotaciones a la izquierda, para eso se usará la función:

```
1: function toLinkedList(node):
2:     while node → right do          ▷ Si no tengo un hijo a mi derecha no puedo rotar a la izquierda
3:         node ← rotateLeft(node)    ▷ La rotación retorna la raiz del nuevo árbol
4:     end while
5:     if node → left then
6:         toLinkedList(node → left)
7:     end if
8: end function
```

En el nodo raíz. Notamos que este algoritmo es correcto ya que

- Es finito: Notamos que cuando se hace una rotación a la izquierda en un j-ésimo, $j \geq 0$, hijo de la izquierda del nodo raíz, la cantidad de veces se que puede acceder a la izquierda desde el nodo raíz aumenta en uno. Luego, notamos que la profundidad máxima de un árbol es n, por lo que no se pueden hacer más de n-1 rotaciones a la izquierda en nodos que son un j-ésimo hijo izquierdo del nodo raíz. Luego, ya que el único movimiento que se hace desde el nodo raíz es moverse a la izquierda las rotaciones se hacen en j-ésimos hijos izquierdos del nodo raíz por lo que las rotaciones están acotadas por n-1. Además, la cantidad de veces que se puede mover hacia la izquierda está limitada por el tamaño del árbol por lo que los movimientos están acotados por n-1. Por lo tanto el algoritmo es finito.
- Entrega el resultado deseado: al entrar al nodo izquierdo en la llamada recursiva el nodo no tiene un hijo derecho. Luego, porque se empieza el algoritmo en el nodo raíz y se recorre hasta llegar al nodo de más a la izquierda, ocurre que cuando un nodo no tiene un hijo a la izquierda, ninguno de los ancestros de ese nodo tiene un hijo a la derecha, por lo que tenemos una lista ligada.

Luego, como se señaló en la demostración de correctitud, la cantidad máxima de rotaciones que se hacen es n-1, por lo que se puede llegar a una lista ligada en $O(n)$ rotaciones a la izquierda.

Ahora, ya logramos construir una lista ligada a partir de un ABB cualquiera. Digamos que el árbol original es T y queremos llegar al árbol objetivo T'. Tanto T como T' son ABB válidos, por lo que existen secuencias de rotaciones $S(T)$ y $S(T')$ que convierten a cada uno en la misma lista ligada.

Ya vimos que las rotaciones son invertibles, por lo que las secuencias son a su vez invertibles. Por lo tanto a su vez podemos convertir una lista ligada de vuelta a T o a T' mediante las rotaciones $S(T)^{-1}$ y $S(T')^{-1}$ respectivamente.

Por lo tanto, podemos convertir T en T' aplicando primero $S(T)$ y luego $S(T')^{-1}$.

Finalmente, como la cantidad de rotaciones en ambos casos es $O(n)$, entonces la cantidad final de rotaciones es $O(n) + O(n) = O(n)$.

Pregunta 2

Una solucion simple sería que para cada elemento en el arreglo, contamos todos los elementos menores que hay a su derecha y vamos aumentando un contador. Sin embargo, al comparar cada elemento con, en promedio, la mitad de los otros del arreglo, esta solución tendría complejidad $O(n^2)$

El hecho de que la complejidad $O(n \cdot \log(n))$ contenga un logaritmo, indica que en algun minuto estamos dividiendo los datos sucesivamente, por lo que es posible que sea posible aplicar dividir para conquistar. Para eso, tendremos que dividir el arreglo en sub-arreglos, calcular la cantidad de inversiones en cada uno de ellos, y agregar (con un método aún por describir) las cantidades respectivas, al ir juntando los sub-arreglos. Esta definición general del algoritmo se asemeja mucho a MergeSort (falta considerar la propiedad calculada), por lo que será conveniente usar una versión modificada del mismo para cumplir el objetivo, y lograr una opción más eficiente que la solución *naive*.

Recordemos que el algoritmo (MergeSort)[1] divide recursivamente las listas, para luego unirlas *ordenadamente*. Esto lo realiza la subrutina *Merge*. El funcionamiento de Merge es el siguiente. Sin pérdida de generalidad, asumamos que el algoritmo ordena de forma ascendente (menor a mayor). Merge recibe dos listas **ordenadas**, y para cada lista, compara el primer elemento de cada una. Como las listas ya están ordenadas, necesariamente el primer elemento de cada lista será el menor de cada una, por lo tanto, el menor entre ambos será el menor de todos los elementos de ambas listas, por lo que lo extrae de la lista correspondiente y lo coloca en el primer lugar de la lista resultante. Una vez extraído el valor, las listas siguen estando ordenadas, por lo que se puede repetir el procedimiento, comparando el primer elemento de cada lista resultante. Merge realiza $n + m$ inserciones a la lista resultante, donde n y m son los largos de cada lista.

Es en cada comparación de *Merge* donde se debe calcular la cantidad de inversiones. Al tomar el primer elemento la lista izquierda (cabeza izquierda) y el de la lista derecha (cabeza derecha), se extrae el menor.

1. **Caso 1: Cabeza derecha es la menor** Como las listas están ordenadas, y la cabeza derecha es menor que el primer elemento de la lista izquierda, es menor que todos los elementos de la lista izquierda. Entonces al ordenarla (extraer e insertar) se debe sumar el largo de la lista izquierda a la cantidad de inversiones, ya que hay una inversión de la cabeza derecha con cada uno de sus elementos.
2. **Caso 2: Cabeza izquierda es la menor** Como la cabeza izquierda es menor a la derecha, esto significa que están ordenadas correctamente. Por lo tanto, se extrae e inserta sin sumar nada al contador de inversiones.

Veamos esto con un ejemplo concreto:

Imaginemos que estamos en una etapa intermedia de `MergeSort()` en la cual llamamos a `Merge()`. Si se considera que se tienen dos sub-arrays involucrados en la fusión: `[x,x,x,x,7,9,12]` y `[x,x,5,6,8,10]`. En este ejemplo 7 y 5 son las cabezas (elementos del arreglo siendo comparados) de los respectivos sub-arrays a ser fusionados. En este caso, como $7 > 5$, (7,5) es una inversión en el arreglo original

Además, es claro que los elementos que siguen a 7 en el arreglo (9 y 12) también generan una inversión con el elemento 5 debido a que `Mergesort()` va generando arreglos ordenados a medida que va fusionando arreglos. Es decir, (9,5) y (12,5). Entonces podemos decir que para el elemento 5, el número total de inversión es 3, lo que es exactamente igual al número de elementos que quedan en los subarreglos de la izquierda.

En cambio, si se tienen los sub arreglos: `[x,x,x,x,7,9,12]` y `[x,x,x,x,8,10]`, este caso $7 < 8$ por lo que, no habría ninguna inversión ya que, 7 va a ser menor que todos los otros elementos del sub arreglo derecho.

[1] https://www.youtube.com/watch?v=JSceec-wEyw

A continuación se adjunta el pseudocódigo de Mergesort, junto con las modificaciones que permiten contar la cantidad de inversiones. (En Merge, B es el arreglo auxiliar).

```
MergeSort(A, i, f)
I = 0
if f > i :
    m = ⌊(i + f)/2⌋
I += MergeSort(A, i, m)
I += MergeSort(A, m + 1, f)
I += Merge(A, i, m, m+1, f)
ret I
```

```
Merge (A, i, m, j, f)
  I = 0
  B[i:f] = A[i:f]
  x = i
  while i < m and j < f :
      if B[i] > B[j]:
          A[x] = B[j]
          j++
          I += m - i + 1
      else
          A[x] = B[i]
          i++
      x++
```

Pasar los elementos sobrantes a $A[x:f]$

```
  ret I
```