Árbol Binario de Búsqueda

Colectivo Estructuras de Datos y Algoritmos

Noviembre 2024

1. Se define *Next* de un nodo como aquel que se ubica inmediatamente después de este en el *array* resultante de llamar al método InOrder. La definición de *Previous* se obtiene de forma análoga. Demuestre que si en un árbol binario de búsqueda un nodo tiene dos hijos entonces su *Next* y su *Previous* son descendientes de dicho nodo.

   a) Demuestre además que su *Next* no tiene hijo izquierdo y su *Previous* no tiene hijo derecho.

Respuesta

Sea x un nodo con dos hijos perteneciente a un **ABB** T, se demostrará que su *Next* es un nodo que pertenece al subárbol de x.

Para esta demostración se definirá *Next* de x como un nodo $y \in T$ tal que se cumple que $y.key > x.key$ y no existe $z \in T$ tal que $y.key > z.key > x.key$. A partir de este momento se hará referencia al nodo *Next* como y.

Sea y un nodo tal que $y.key > x.key$ y pertenece al subárbol de x. De igual forma z será un nodo que cumple que $z.key > x.key$ y no pertenece al subárbol de x.

Se separarán los nodos z en dos conjuntos:

- Aquellos que cumplen que x pertenece a su subárbol izquierdo:
  En este caso, todo z que contenga a x en su subárbol izquierdo también contiene a todo nodo perteneciente al subárbol derecho de x y por tanto se cumple que $z.key > w.key$, para todo w que pertenece al subárbol derecho de x.
- Aquellos que no contienen a x en su subárbol izquierdo:
  Para todos estos nodos existe un z' tal que z' contiene a x en su subárbol izquierdo y a z en su subárbol derecho. Por tanto $z.key > z'.key$ y también se cumple que $z.key > w.key$ para todo w que pertenece al subárbol derecho de x.

Por tanto, los únicos candidatos a ser *Next* de x se encuentran en el subárbol derecho de x y son, por consiguiente, sus descencientes.

a) Por definición de *Next* se tiene que no existe nodo z tal que $z \in T$ y $y.key > z.key > x.key$, dado que y es el *Next* de x.

Demostración por reducción al absurdo:

Se asumirá que el nodo y tiene hijo izquierdo z. Por la propiedad que se cumple en los **ABB** se tiene que $y.key > z.key$. Por el hecho de que z se encuentra en el subárbol derecho de x se tiene que $z.key > x.key$. Por tanto se obtiene $y.key > z.key > x.key$, lo cual resulta en una contradicción con lo previamente expuesto.

Las demostraciones relacionadas con el valor *Previous* se realizan de forma análoga.

2. Sea T un árbol binario cuyas llaves son todas distintas. Sea x un nodo hoja y sea y su padre. Demuestre que $y.key$ es: el menor número de T que es mayor que $x.key$; o el mayor número de T que es menor que $x.key$.

3. Se tiene el siguiente algoritmo que devuelve una lista con los elementos de un árbol recorridos en entreorden.

```
1 def InOrder(T):
2     if |T| = 0:
3         return []
4     return InOrder(T.left) + [T.key] + InOrder(T.right)
```

   a) Demuestre que si $L = InOrder(T)$, entonces L contiene todos los elementos de T ordenados de menor a mayor.

4. Sobre los **ABB** se definen las siguientes funciones:

   Search(T, key) : Busca en el **ABB** T la llave key, si existe devuelve el nodo que la contiene, y en otro caso devuelve $null$ [1]

   Predecessor(T, key) : Busca en el **ABB** T el máx $\{x_{key} \mid x_{key} \in T \wedge x_{key} \leq key\}$. (Devuelve la mayor de las llaves contenidas en T que sean menores o iguales que key, en caso de no existir ninguna se devuelve $null$)

   Successor(T, key) : Busca en el **ABB** T el mín $\{x_{key} \mid x_{key} \in T \wedge x_{key} \geq key\}$. (Devuelve la menor de las llaves contenidas en T que sean mayores o iguales que key, en caso de no existir ninguna se devuelve $null$)

   Rank(T, key) : $|\{x_{key} \mid x_{key} \in T \wedge x_{key} < key\}|$ (Devuelve la cantidad de llaves en el **ABB** T que sean menores a key)

   Select(T, k) $= x_{key}$ donde $x_{key} \in T \wedge Rank(T, x_{key}) = k$ (Devuelve la llave perteneciente a T que tiene exactamente $k-$menores en T). Se asume que siempre $0 \leq k < |T|$.

   En estos métodos se cumple lo siguiente:

   a) $Select(T, Rank(T, key)) = Successor(T, key)$

   b) $Rank(T, Select(T, k)) = k$ cuando $0 \leq k < |T|$

   c) $Select(T, k) = InOrder(T)[k]$. (Asumiendo que $InOrder$ devuelve un array con los key ordenados de menor a mayor e indexado en 0).

   Para cada una de estas funciones diseñe un algoritmo que las resuelva. La complejidad temporal de sus algoritmos en cada caso debe ser $O(h)$, donde h es la altura del **ABB** T. La estructura del **ABB** es la vista en conferencia, pero asumiendo que se tiene para cada nodo una propiedad $size$ que contiene la cantidad de nodos en ese sub-árbol ($x.size = x.left.size + x.right.size + 1$).

Respuesta

Sea T un **ABB** donde cada nodo tiene las siguientes propiedades:

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

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

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

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

Es necesario definir bien estos métodos sobre un **ABB** porque serán las bases para definir nuevas estructuras (con el uso del **AVL**), que nos permitirán resolver problemas aún más complicados. Luego de esta clase pueden usar estos métodos sin necesidad de demostrarlos.
En todos estos ejercicios la idea es utilizar la invariante del orden de las llaves en un **ABB** y en todo momento tomar la decisión de sobre qué hijo del árbol enfocar la búsqueda. Estos se resuelven con un razonamiento inductivo, primero analizando su caso base y luego preparando la entrada para ajustar la respuesta del algoritmo en otro sub-árbol.

[1] En la *Conferencia 4-5 ABB AVL* se brinda una solución a la función *Search*, puede ser útil entender bien este método antes de implementar el resto de las funciones sobre un **ABB** en esta CP

```
1      def Search(T, key):
2          if |T| = 0:      # |null| = 0 y |{}| = 0
3              return null
4          if T.key = key:
5              return T
6          if key < T.key:
7              return Search(T.left, key)
8          return Search(T.right, key)
```

Este es el algoritmo visto en conferencia.

```
1      def Predecessor(T, key):
2          if |T| = 0:
3              return null
4          if T.key = key:
5              return T
6          if key < T.key:
7              return Predecessor(T.left, key)
8          p = Predecessor(T.right, key)
9          if p = null:
10             return T.key
11         return p
```

La idea para buscar en T el antecesor de key es descartar cuales llaves no forman parte del conjunto de las que son menores que key, y de las posibles quedarnos con la mayor.

```
1      def Rank(T, key):
2          if |T| = 0:
3              return 0
4          sizeLeft = |T.left|
5          if key = T.key:
6              return sizeLeft
7          if key < T.key:
8              return Rank(T.left, key)
9          return sizeLeft + 1 + Rank(T.right, key)
```

La idea para resolver esta función es ir directo a la definición de $Rank(t, key)$. Necesitamos saber la cantidad de llaves menores que key en T. Sabiendo que T es un ABB con todas las llaves distintas vamos a resolver esta función con una recurrencia aprovechando la estructura de T. En un primer caso si T es vacío el $Rank$ es 0 porque $\neg\exists x \in T, x < key$. Si $key == T.key$ en este ABB solamente las llaves del hijo izquierdo son menores que key. Si $key < T.key$ recursivamente se deben buscar los menores que key en el hijo izquierdo debido a que todos las llaves en el hijo derecho así como $T.key$, son mayores que key. En el caso en que $key > T.key$ la cantidad de menores que key en T son: todas las llaves en el hijo izquierdo, $T.key$, y las llaves que sean menores que key en el hijo derecho.

```
def Select(T, k):
    sizeLeft = |T.left|
    if k = sizeLeft:
        return T
    if k < sizeLeft:
        return Select(T.left, k)
    return Select(T.right, k − sizeLeft − 1)
```

En esta función la idea es casi construir la inversa de la función *Rank*, nuevamente aprovechando la estructura de T y con un razonamiento recursivo.
En todos estos ejemplos en cada llamado se consume $O(1)$ y la cantidad de llamadas que se hace a la función es un camino entre la raíz de T y un nodo objetivo. Por definición de altura del árbol, todos los caminos entre la raíz y un nodo es siempre menor o igual a esta. Por tanto todas estas funciones son $O(h(T))$.

5. Se tiene una lista de números enteros L, inicialmente vacía y se desea responder Q consultas de tres tipos:
   - Añadir un número entero a L.
   - Eliminar un número de L (se garantiza que este número ya está en L).
   - Responder cuántos números de L pertenecen al intervalo $[a, b]$.

   Diseñe un algoritmo que resuelva este problema (usando un ABB) en tiempo $O(Q * h(T))$. Donde $h(T)$ es la altura del árbol construido.

   Respuesta

   La idea para resolver este problema es mantener un **ABB** con los números que se han insertado hasta el momento, las consultas de tipo 1 y 2 se pueden simular con los métodos *Insert* y *Remove* ya vistos. Para responder la consulta de cuántos números hay en el intervalo $[a, b]$ se puede hacer uso de la siguiente observación:
   Contar cuántos elementos de los presentes hasta el momento en el **ABB** hay en el intervalo $[a, b]$ es equivalente a contar cuántos números son menores que $b + 1$ y restarle cuántos números son menores que a. De esta manera no se estaría contando ningún número mayor que b ni ningún número menor que a.
   Ahora para contar estas cantidades se puede hacer uso del método *Rank* ya visto:
   - $Rank(T, b + 1)$ nos dirá cuántos números son menores que $b + 1$ (o sea menores o iguales que b).
   - $Rank(T, a)$ nos dirá cuántos números son menores que a.

6. Hay N casas y M antenas con posiciones enteras. Todas las antenas transmiten la señal a un radio r, es decir, si una antena está en la posición x le va a transmitir señal a todas las casas con posición y que cumplan $|x - y| \leq r$. Diseñe un algoritmo que determine el mínimo r necesario para transmitir señal a todas las casas.
   Complejidad: $O((N + M) * h(T))$ donde $h(T)$ es la altura del árbol construido.

   Respuesta

   Intuitivamente, para cada casa interesa que la antena más cercana a dicha casa sea la que le transmita señal. Entonces, para cada casa se debe encontrar la antena más cercana y calcular la distancia. El mínimo r será el máximo de esas distancias.
   Sean $H_1, H_2, \ldots, H_N$ las posiciones de las casas y $A_1, A_2, \ldots, A_M$ las posiciones de las antenas. Sin perder generalidad supongamos que para cada casa con posición p existe al menos una antena a su izquierda y al menos una antena a su derecha (en caso de que alguna de ellas no exista, es suficiente considerar la antena que exista), luego se encuentran las antenas más cercanas a la izquierda ($L = \text{máx}\{A_i \mid A_i \leq p\}$) y a la derecha ($R = \text{mín}\{A_i \mid A_i \geq p\}$), y la distancia mínima desde p hasta alguna antena es $\text{mín}(|p - L|, |p - R|)$. Como se mencionó antes, r es igual al máximo entre todas esas distancias.
   Notar que la definición de L y R coinciden con la definición de *Predecessor* y de *Successor* de un **ABB**,

respectivamente.
Demostración de correctitud:
Primero, cada casa necesita una antena dentro de un radio r, por construcción $r \geq \text{mín}(|p-L|, |p-R|)$, por lo que cada casa será cubierta por al menos una antena. Además, no se necesitan considerar antenas que están más a la izquierda de L o más a la derecha de R, ya que si esto ocurriera entonces $A_i < L$ o $A_i > R$, y por lo tanto $|p - A_i| > |p - L|$ o $|p - A_i| > |p - R|$.

```
1   def Solve(N, H, M, A):
2       S = MultiSet()
3       for i in H:
4           Insert(S, i)
5       r = 0
6       for p in A:
7           L = Predecessor(S, p)
8           R = Successor(S, p)
9           if |L| != 0 and |R| != 0:
10              closest = min(|p - L.key|, |p - R.key|)
11          elif |L| != 0:
12              closest = |p - L.key|
13          else:
14              closest = |p - R.key|
15          r = max(r, closest)
16      return r
```

7. Dada una lista de números ordenada de N elementos sin repetición, diseñe un algoritmo que construya un Árbol Binario de Búsqueda (ABB) en tiempo $O(n)$.

   a) Modifique su algoritmo para que el árbol obtenido sea un AVL.