SOLUCIÓN EVALUACIÓN CONTINUA 7

Autores: Favian Huarca Mendoza, Ana Silvia Cordero Ricaldi, Leonel Leodolfo Campuzano Diestra, Yulinio Zavala Mariño, Sergio Marcelo Ricce Abregú

K-th Smallest Element in an AVL Tree

Solución de alto nivel:

Para poder encontrar el k-esimo valor más pequeño en el AVL tree se hará un recorrido del árbol. Es por ello que se dividirá la búsqueda en 3 partes: Si son menores a root, si es root y si es mayor a root. Para ello se necesita el tamaño del subárbol izquierdo. Se usará la función getSize para ello. Finalmente para recorrer los subárboles se tendrá que llamar a la función de manera recursiva según si es derecha o izquierda. Como observación, al estar usando el tamaño del subárbol izquierdo como punto de comparación, para el uso recursivo de la función en el subárbol derecho se debe restar los nodos del árbol izquierdo y el root.

Complejidad Temporal: O(log(n))
Complejidad Espacial: O(n)

Imagen del Output:

Input:

```
root = insert(root, 20);
root = insert(root, 10);
root = insert(root, 30);
root = insert(root, 5);
root = insert(root, 15);
root = insert(root, 25);
root = insert(root, 35);
```

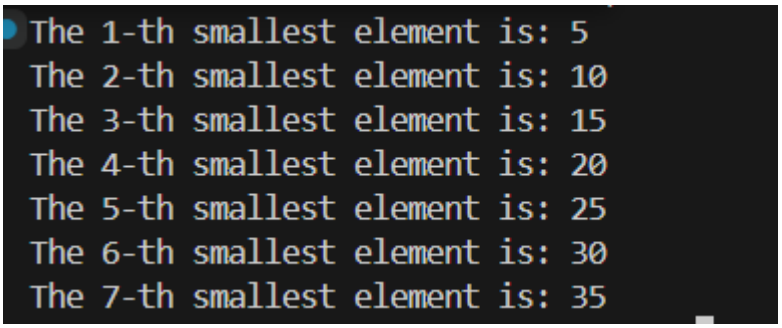
Output:

```
The 1-th smallest element is: 20
The 2-th smallest element is: 30
The 3-th smallest element is: 40
The 4-th smallest element is: 50
The 5-th smallest element is: 60
The 6-th smallest element is: 70
The 7-th smallest element is: 80
```

Input:

```
root = insert(root, 50);
root = insert(root, 30);
root = insert(root, 70);
root = insert(root, 20);
root = insert(root, 40);
root = insert(root, 60);
root = insert(root, 80);
```

Output:



Merge Two AVL Trees (Intersection - Only Common Values)

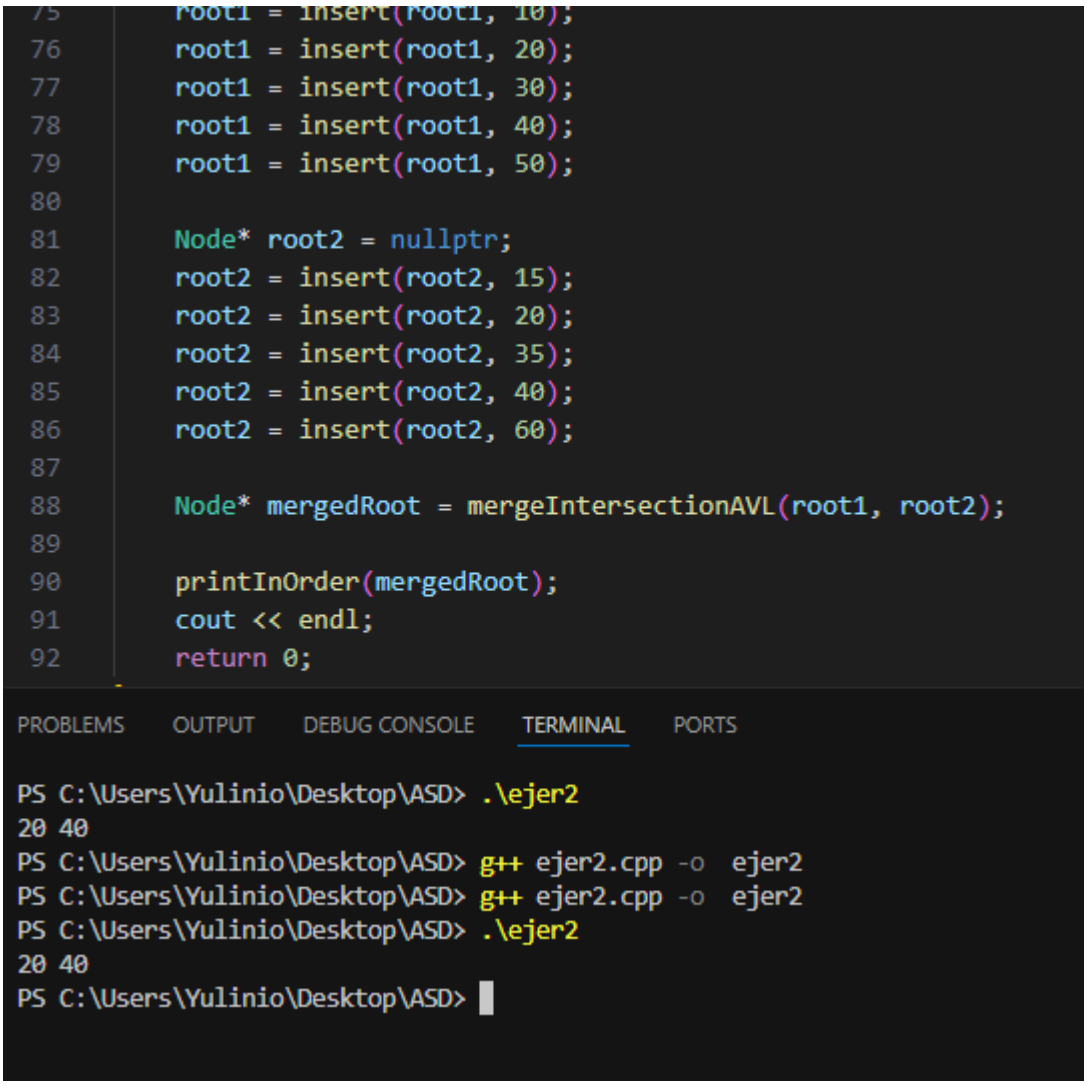
Solución de alto nivel:

Para resolverlo se definirá una rutina de recorrido inorder que extraiga los valores ordenados de cada AVL en dos vectores, luego se compararon estos vectores con un pase dual para generar un tercer vector con los valores comunes en orden creciente, a continuación estos valores se insertarán secuencialmente en un nuevo árbol AVL aplicando las rotaciones necesarias para mantener el balance, y por último se utilizará una función de impresión inorder para visualizar y verificar el árbol de intersección resultante.

Complejidad Temporal:
n1 y n2→ visitas recursivas en “inorder(root1, root2)”
Bucle de inserción→O(klogk) donde k(llamadas)
Por tanto es:
O(n1+n2+klogk)

Complejidad Espacial:
Por los vectores auxiliares(n1 y n2), commonValues(k)
O(n1+n2+k) —> O(n)

Captura de salida:



```
75      root1 = insert(root1, 10);
76      root1 = insert(root1, 20);
77      root1 = insert(root1, 30);
78      root1 = insert(root1, 40);
79      root1 = insert(root1, 50);
80
81      Node* root2 = nullptr;
82      root2 = insert(root2, 15);
83      root2 = insert(root2, 20);
84      root2 = insert(root2, 35);
85      root2 = insert(root2, 40);
86      root2 = insert(root2, 60);
87
88      Node* mergedRoot = mergeIntersectionAVL(root1, root2);
89
90      printInOrder(mergedRoot);
91      cout << endl;
92      return 0;
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\Yulinio\Desktop\ASD> .\ejer2
20 40
PS C:\Users\Yulinio\Desktop\ASD> g++ ejer2.cpp -o  ejer2
PS C:\Users\Yulinio\Desktop\ASD> g++ ejer2.cpp -o  ejer2
PS C:\Users\Yulinio\Desktop\ASD> .\ejer2
20 40
PS C:\Users\Yulinio\Desktop\ASD>
```

AVL Playlist Manager

Solución de alto nivel:

Para añadir una canción, se sigue la lógica de un árbol binario de búsqueda: si el árbol está vacío, se crea el nodo directamente. Si el score de la nueva canción es menor que el del nodo actual, se avanza al subárbol izquierdo; si es mayor, se dirige al subárbol derecho. En caso de que el score sea igual, no se inserta para evitar duplicados. Una vez insertado el nodo, se actualiza la altura del nodo actual, que se calcula como 1 más la altura del hijo más alto (izquierdo o derecho). Finalmente, se determina el factor de balance del nodo restando la altura del subárbol izquierdo menos la del derecho (balance = altura izquierda - altura derecha), lo que permite evaluar si el árbol requiere reequilibrio para mantener su estructura óptima.

Para remover: Buscamos desde la raíz y buscamos el valor del nodo correspondiente, si está en la izquierda, vamos al subnodo izquierdo. Si es derecho, al subnodo derecho. Si el nodo no tiene hijos, solo se elimina. Si tiene un hijo, el nodo se reemplaza por el hijo. Si tiene dos hijos, se reemplaza por el nodo izquierdo.
Para obtener la mas popular:

- Solo es necesario recorrer el subárbol izquierdo hasta que no hayan más nodos, ya que eso representaría el valor más alto que sería el más popular.

y finalmente para obtener el kth popular usamos un orden inverso (derecha a izquierda para garantizar que los nodos sean decrecientes. Se usará un stack que recorra de forma iterativa y usar un contador. Principalmente el stack ayudará a contar cuántos nodos visitó y el contador alcanza el valor de k, ese valor se retornará.

Complejidad Temporal
O(Log n) excepto Kth popular (k+ Log N)
Complejidad espacial
O(n) →Arbol
O(Logn)--> Pila

Capturas: