UNIVERSIDAD DE ALMERÍA

Estructuras de datos y Algoritmo I

Práctica 02: ejercicio 02

Curso: 2º Grado en Ingeniería Informática
Grupo docente: B1 **Grupo**: GTA3
Nombre y Apellidos:
Jefferson Max Tomalá Villarreal

Ejercicio 2. Árbol binario de búsqueda AVL. Gestión de ciudades donde empresas de software desarrollan sus proyectos utilizando un AVLTree.

Además, una vez en organizados los datos en la ED indicada anteriormente, se debe responder (con la implementación de la correspondiente función) a las siguientes consultas:

- Devolver las empresas que tienen su sede en una ciudad determinada.
- Devolver los *proyectos* con sede en una ciudad especificada como parámetro de entrada.
- ¿En cuántas ciudades diferentes se desarrollan proyectos de una empresa
- determinada?
- ¿En cuántas ciudades diferentes se desarrolla un determinado proyecto?
- ¿Cuántos proyectos está desarrollando una empresa?

Explique en un documento (memoria) toda implementación suministrada, prestando especial atención a funciones principales de dicha estructura de datos como: **add, remove, clear, contains, isEmpty**, y al iterador iterator del tipo TreeIterator. También se pide, en dicha memoria, que se enumeren las ventajas e inconvenientes de la resolución de este problema mediante el uso de estructuras arbóreas

Un árbol AVL

Un árbol AVL es un tipo especial de árbol binario ideado por los matemáticos rusos Adelson-Velskii y Landis. Fue el primer **árbol de búsqueda binario auto-balanceable** que se ideó.

- Los árboles AVL están siempre equilibrados de tal modo que, para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa.
- Gracias a esta forma de equilibrio (o balanceo), la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad O(log n).
- El factor de equilibrio puede ser almacenado directamente en cada nodo o ser computado a partir de las alturas de los subárboles.

Para conseguir esta propiedad de equilibrio, la inserción y el borrado de los nodos se ha de realizar de una forma especial. Si al realizar una operación de inserción o borrado se rompe la condición de equilibrio, hay que realizar una serie de rotaciones de los nodos.

Rotaciones:

El reequilibrado se produce de abajo hacia arriba sobre los nodos en los que se produce el desequilibrio. Pueden darse dos casos: rotación simple o rotación doble; a su vez ambos casos pueden ser hacia la derecha o hacia la izquierda.

Add: Este método añade elementos al árbol binario devolviendo un booleano “true” en caso de que exista el elemento ítem devolverá un booleano “false” (una de las propiedades del árbol es que no existan elementos repetidos). Por parámetro le pasaremos el elemento parametrizado de la clase <T>. A diferencia del BSTree, el árbol AVLTree al añadir el elemento intenta añadirlo en un subárbol el cual este balanceado con los otros árboles padres anteriores.

Proceso de inserción:

1. Buscar hasta encontrar la posición de inserción o modificación (proceso idéntico a inserción en árbol binario de búsqueda)
2. Insertar el nuevo nodo con factor de equilibrio “equilibrado”
3. Desandar el camino de búsqueda, verificando el equilibrio de los nodos, y re-equilibrando si es necesario

    - El tiempo de ejecución para la inserción es del orden O (log(n)).

Remove: Este método elimina un elemento pasado por parámetro item y devolverá un booleano “true” si existe en el árbol y finalmente si se modifica el mismo. En caso contrario devolverá un “false”. A diferencia del BSTree, el árbol AVLTree al eliminar un elemento intenta añadirlo en un subárbol el cual este balanceado con los otros árboles padres anteriores de este modo cumple la propiedad de árbol balanceado.

- El problema de la extracción puede resolverse en O (log(n)) pasos. Una extracción trae consigo una disminución de la altura de la rama donde se extrajo y tendrá como efecto un cambio en el factor de equilibrio del nodo padre de la rama en cuestión, pudiendo necesitarse una rotación.

Clear: Este método elimina todos los elementos del árbol dejándolo vacío.

Contains: Método que regresa un booleando en caso de que el elemento ítem pasado por parámetro exista/ este contenido dentro de la estructura de datos BSTree. True en caso de que exista, false en caso contrario.

isEmpty: Método que consulta si el árbol está vacio, en caso de que lo esté devolverá un booleano “true” en caso que contenga elementos “false”.

Iterator: Este método regresa un iterador para poder desplazarnos por los elementos del árbol. El tipo de iterador que regresa es de la clase TreeIterator que implementa la interfaz Iterator y contiene todas las funciones de este: hasNext(), next(), current(), remove()…

También se pide, en dicha memoria, que se enumeren las ventajas e inconvenientes de la resolución de este problema mediante el uso de estructuras arbóreas (AVLTree) en lugar de colecciones lineales (ArrayList) como se hizo la práctica 1.

Ventajas AVLTree sobre ArrayList: El tiempo promedio de las operaciones sobre los AVLTree es del orden de Θ (log(n)) y en el peor caso de Θ(n) por lo que podemos decir que la búsqueda a diferencia de una estructura lineal en este caso ArrayList será menor.

Inconvenientes AVLTree sobre ArrayList: Los AVLTree tienen la propiedad de equilibro donde la diferencia de altura de los subárboles de cada uno de sus nodos es como mucho uno, es decir, tiene rotación de nodos y esto requiere un orden de complejidad añadida a diferencia de la inserción y eliminación en la estructura de datos lineal de los ArrayList.