CLASE 7

Para medir la eficiencia, se utiliza el modelo del peor caso en función del número de elementos (n) de la estructura. Es fundamental comprender las diferentes clasificaciones de costos:

- **Constante (O(1))**: El costo es siempre el mismo, sin importar la cantidad de elementos.
- **Logarítmico (O(log n))**: Típicamente, se asocia con recorrer una rama de un árbol balanceado.
- **Lineal (O(n))**: Implica realizar operaciones constantes por cada elemento en la estructura.
- **"N log N" (O(n log n))**: Representa la ejecución de operaciones logarítmicas por cada elemento.
- **Cuadrático (O(n^2))**: Implica realizar operaciones lineales por cada elemento.

Árboles Binarios de Búsqueda (BSTs)

Los BSTs son una forma de implementar Tipos de Datos Abstractos (TADs) como los Sets con costos logarítmicos, siempre que los elementos estén ordenados y organizados en un árbol.

Invariante de Representación de un BST: Para cualquier nodo con valor x, todos los elementos en su subárbol izquierdo deben ser menores que x, y todos los elementos en su subárbol derecho deben ser mayores que x. Esta propiedad es la clave para su eficiencia.

- **Búsqueda (belongs)**: Se compara el elemento buscado con la raíz. Si es menor, se busca en el subárbol izquierdo; si es mayor, en el derecho. Este proceso tiene un costo de O(log n) en un árbol balanceado.
- **Inserción (adds)**: Se sigue la misma lógica de búsqueda para encontrar la posición correcta y luego se inserta el nuevo nodo. El costo también es de O(log n) en un árbol balanceado.
- **Borrado (removeS)**: Al eliminar un nodo, pueden quedar dos subárboles. Para mantener la estructura de BST, el nodo eliminado se reemplaza por el elemento máximo del subárbol izquierdo o el mínimo del subárbol derecho. Esta operación tiene un costo de O(log n) si el árbol está balanceado.

Peor Caso para BSTs: Si los elementos se insertan en orden (ascendente o descendente), el árbol se degenera en una estructura similar a una lista, y el costo de las operaciones se vuelve lineal (O(n)).

Árboles AVL

Los árboles AVL son BSTs autobalanceados que resuelven el problema del peor caso de los BSTs.

Invariante de Representación de un AVL: Para cualquier nodo, la diferencia de alturas entre su subárbol izquierdo y su subárbol derecho no puede ser mayor que 1. Además, debe mantener el invariante de BST.

- **Balanceo**: Para mantener el invariante de AVL durante las inserciones y eliminaciones, se realizan operaciones de "rotación" (simple o doble) que reestructuran el árbol para mantenerlo balanceado.
- **Eficiencia**: Gracias al autobalanceo, la altura de un AVL se mantiene siempre logarítmica, garantizando un costo de O(log n) en el peor de los casos para las operaciones de búsqueda, inserción y borrado.

Heaps (Montículos)

Los Heaps son estructuras de datos basadas en árboles, ideales para implementar PriorityQueues (Colas de Prioridad) con eficiencia logarítmica.

Invariantes de Representación de un Heap:

1. **Propiedad de Heap**: La raíz del árbol es siempre el elemento mínimo (o de máxima prioridad) de todo el conjunto.
2. **Estructura de Árbol Lleno**: Todos los niveles del árbol están completos, excepto posiblemente el último, que se llena de izquierda a derecha sin dejar huecos.

- **Encontrar el Mínimo (findMinPQ)**: Gracias al invariante de heap, esta operación es de costo constante (O(1)), ya que el elemento mínimo siempre se encuentra en la raíz.
- **Inserción (insertPQ)**: Un nuevo elemento se añade en la próxima posición disponible para mantener el árbol lleno. Esto puede romper el invariante de heap, por lo que el elemento "flota" hacia arriba hasta encontrar su posición correcta. Esta operación tiene un costo de O(log n).
- **Borrado del Mínimo (deleteMinPQ)**: Se elimina la raíz. Para llenar el vacío, el último elemento del árbol se mueve a la raíz. Esto puede violar el invariante de heap, por lo que este elemento se "hunde", intercambiándose con el menor de sus hijos hasta que se restaura la propiedad de heap.

Observación Clave: BSTs, AVLs y Heaps no son TADs por sí mismos, sino estructuras de datos eficientes que se utilizan para implementar TADs como Sets, Maps y Priority Queues. El uso correcto de sus invariantes es crucial para garantizar la eficiencia.