Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2133 - Estructura de Datos y Algoritmos

# Informe Tarea 1

25 de abril de 2024
Lucas Fernández
20639112

---

## Motivación

Este problema nos permite aprender diferentes habilidades que son de alta utilidad para desarrollos futuros. Por una parte, se consolida el correcto uso de C, lenguaje de alta utilidad debido a ser de bajo nivel y, así, conseguir programas de mayor velocidad. Por otra parte, nos entrega la oportunidad de conocer diferentes estructuras y algoritmos que se utilizan usualmente en desarrollos, por ejemplo, el uso de heaps y árboles AVL, algoritmos de inserción y búsqueda recursiva en ellos tanto para valores particulares como rangos, esto nos da la capacidad de expandir nuestros conocimientos y mejorar nuestro desarrollo, estimaciones de tiempo y comportamiento sobre estos y como ciertas técnicas nos permiten mantenernos dentro de cierta complejidad dada. El uso de heaps se puede encuentrar en sistemas que requieran seguir ciertas prioridades u orden, tales como, sistemas operativos para manejo de memoria, filas de atención al cliente, manejo de información en la red, etc. Los árboles AVL se utilizan en sistemas que requieran una rápida respuesta, tales como, bases de datos, compiladores, sistemas de archivos, etc.

## Informe

Para solucionar el problema se utilizaron dos estructuras, Node y Heap, para la parte 1 y tres estructuras, Head, Node y NodeToTree, para la parte 2.

En la primera parte, Node contiene la información relacionada con el usuario, el monto de la compra/venta y la prioridad que tiene en el régimen FIFO. Heap contiene un array de punteros a Node, el tamaño máximo del array y el índice en el que se encuentra actualmente, esta estructura representa un heap en el cual se ordenan por prioridad sus nodos, la inserción y extracción tienen como peor caso una complejidad $O(\log(n))$, la búsqueda es de O(1) dado que se encuentra ordenado por prioridad.

En la segunda parte, Head tiene toda la información relacionada con las cabezas, Luego, Node contiene la información relacionada con la prioridad de la cabeza que representa, el índice en el cual se encuentra el puntero de la cabeza en el array en el cual se guarda, la máxima altura de los hijos izquierdo y derecho del nodo, la altura de cada uno de los lados y una lista ligada y la cantidad de los nodos que tienen la misma prioridad. Esta estructura permite crear un arbol AVL, teniendo así una complejidad de inserción y búsqueda igual a $O(\log(n))$. Por último, NodeToTree es una estructura bastante similar a Node, dado que permite generar un árbol AVL y mantener la complejidad de inserción y búsqueda en este, pero con la sutil diferencia que en vez de guardar una lista ligada de nodos con la misma prioridad guarda el nodo raíz a otro árbol AVL generado con la estructura Node, esto permite manejar las consultas donde existe una doble prioridad.

Las estructuras implementadas se manejan en memoria de distintas formas. Al crear un Heap se reserva un espacio en la memoria igual a la cantidad de los datos ponderado con el tamaño de cada nodo, al momento del liberar la memoria se itera por la cantidad de espacios usados, liberando cada nodo, y luego se libera el array y, finalmente, el Heap. Los nodos pertenecientes al Heap reservan la memoria justo antes de ser insertados. Todo esto

ocupa O(n) memoria. En el caso de los nodos de los árboles AVL se instanciaban para luego insertarlo en el árbol recorriendo como más O(log(n)) para ser guardado dentro de otro nodo. Los nodos guardaban el nodo a la izquierda, derecha y los con mismos prioridad. En el caso de los árboles anidados se hacía el mismo procedimiento con dos tipos de nodos. Ambos requerían O(n) de memoria.

Las complejidades de las operaciones heap markert se explicaron más arriba.

Para los árboles de búsqueda se tomó la estrategia de realizar una lista ligada en los casos con la misma prioridad y ligar los nodos con tal de conseguir el árbol. También se tomó la estrategia de dividir la búsqueda en nodo izquierdo y derecho, partiendo primero con el izquierdo, lo que permite al momento de buscar los rangos partir guardando el menor y luego ir progresivamente guardando el resto en orden de menor a mayor, consiguiendo así lo solicitado. Por último, se tomó la estrategia de ligar árboles a los nodos que debían funcionar para búsquedas con dos parámetros, esto permitió mantener la complejidad de búsqueda igual a O(log(n)).

Para el bonus, utilizando la estrategia de los árboles anidados, primero se busca el menor x, para luego entrar en el árbol ordenado por y. Al momento de encontrar el menor y válido, se revisa que este de acuerdo al rango para posteriormente guardarlo y subir a través de los valores del y con el x fijo. Luego de esto se revisa el siguiente x y cada uno de los y correspondientes. Al tener dos árboles se consigue una complejidad de O(log(n)) en el peor caso por árbol, lo que finalmente se traduce en un algoritmo O(log(n)).

# Conclusión

Concluyendo, el problema planteado nos entregó la oportunidad de aprender diferentes habilidades, algoritmos y estructuras que son de alta utilidad para desarrollos futuros. Nos permitió poner en práctica el uso de heaps y árboles AVL, algoritmos de inserción y búsqueda recursiva en ellos tanto para valores particulares como rangos, expandiendo así nuestros conocimientos y mejorando nuestro desarrollo, estimaciones de tiempo y la comprensión del comportamiento de estos y como ciertas técnicas nos permiten mantenernos dentro de cierta complejidad dada. Todo esto nos da el conocimiento para aplicarlo en proyectos reales.