ECE385: Homework 03

Farbod Mohammadzadeh
(1008360462)
Connor Ludwig
(1008333028)

13 October 2023

7 Pages

Problem 1

The best way to check which data structure is best for a dictionary is to see which data structure has the best time complexity for the operations.

Linked List

Linked lists are a time inefficient data structure for a dictionary, because they have a time complexity of $O(n)$ for *insert* and *rank*, since you have to go through the entire list each time. Additionally, *search* and *delete* could be $O(n)$ if the list is not sorted, or $O(\log n)$ if it is sorted. However, they do provide a space efficient data structure, since they only require $O(n)$ space.

Array

Arrays are a space inefficient data structure, while they do require $O(n)$ space, it is a non-sparse data structure, so it is not as space efficient as a linked list. However, arrays are time efficient for a dictionary, since they have $O(1)$ time complexity for *insert*, *delete*, and *search*. However, *rank* is $O(n)$, since you have to go through the entire array to find the rank, which is worse than a linked list since the structure could be sparse and have a lot of empty space.

Hashmap

A hashmap is a space efficient data structure, since it can be sparse. However, it is not as space efficient as a linked list. *insert*, *delete*, and *search* are all $O(1)$, which is the best time complexity for a dictionary. However, *rank* is $O(n)$, since you have to go through the entire hashmap to find the rank, which is worse than a linked list since the structure could be sparse and have a lot of empty space.

Self-Balancing Binary Search Tree

A self-balancing BST would provide great time complexity for *insert*, *delete*, *search*, and *rank*, since they are all $O(\log n)$. However, it is more space inefficient than a linked list.

Problem 2

Keys: 7, 9, 88, 11, 25, 23, 22, 28, 14, 21

$$h_p(k) = k \bmod 11$$

$$h_s(k) = 3k \bmod 4$$

We start with an empty table of length 11. The indices of the cells are 0-10 inclusive.

The following insertions happen:

$$k = 7, h_p(7) = 7$$

							7			

$$k = 9, h_p(9) = 9$$

							7		9	

$$k = 88, h_p(88) = 0$$

88							7		9	

$$k = 11, h_p(11) = 0,\ \text{This is a collision, index 0 is already occupied}$$

$$h_s(11) = 33 \bmod 4 = 1,\ \text{so probe with a step size of 1}$$

88	11						7		9	

$$k = 25, h_p(25) = 3$$

88	11		25				7		9	

$$k = 23, h_p(23) = 1, \text{ This is a collision, index 1 is already occupied}$$
$$h_s(23) = 69 \bmod 4 = 1, \text{ so probe with a step size of 1}$$

88	11	23	25				7		9	

$$k = 22, h_p(22) = 0, \text{ This is a collision, index 0 is already occupied}$$
$$h_s(22) = 66 \bmod 4 = 2, \text{ so probe with a step size of 2}$$

88	11	23	25	22			7		9	

$$k = 28, h_p(28) = 6$$

88	11	23	25	22		28	7		9	

$$k = 14, h_p(14) = 3, \text{ This is a collision, index 3 is already occupied}$$
$$h_s(14) = 42 \bmod 4 = 2, \text{ so probe with a step size of 2}$$

88	11	23	25	22	14	28	7		9	

$$k = 21, h_p(21) = 10$$

88	11	23	25	22	14	28	7		9	21

Therefore, the resulting hash table is:

88	11	23	25	22	14	28	7		9	21

Problem 3

Part a

Expression for Losses:

To minimize the waiter's losses, we want to minimize the impatience-weighted waiting time for all clients. Let's define the total loss L as the sum of losses for all clients:

$$L = t_1 * o_1 + t_2 * o_2 + t_3 * o_3 + \cdots + t_n * o_n$$

where:

t_i is the impatience of the i-th client.

o_i is the time taken to take the order from the i-th client.

The goal is to minimize L.

Greedy Algorithm:

1. Sort the clients based on their impatience, in non-decreasing order (i.e., in ascending order of t_i).

2. Take the orders from the clients in this sorted order.

This greedy algorithm sorts the clients by impatience and serves the least impatient clients first.

Part b

After making the first greedy choice (sorting by impatience and taking the orders in this order), the problem reduces to a smaller subproblem. In this subproblem, you need to solve the same problem for the remaining clients, excluding the client with the least impatience (as they have already been served). You can apply the same algorithm recursively to solve this subproblem.

Part c

To prove the greedy choice property, you need to show that there is an optimal solution that agrees with the first greedy choice your algorithm makes.

Let's assume there exists an optimal solution where you don't serve the least impatient client first. Instead, you serve another client with higher impatience. We can argue that this alternative solution is not better. If you serve a client with higher impatience first, it will increase the loss for that client, which is against the goal of minimizing the total losses. So, the first greedy choice of serving the least impatient client is indeed optimal.

Part d

The problem exhibits optimal substructure because the optimal solution to the original problem can be constructed from the optimal solutions to its smaller subproblems. By serving the least impatient client first and solving the subproblem of the remaining clients, you are constructing an optimal solution for the original problem. This is because the greedy algorithm minimizes the losses for the current client, and by recursively applying it, you minimize the losses for the entire set of clients.

In summary, the greedy algorithm sorts clients by impatience and serves them in ascending order of impatience, which minimizes the losses. The problem exhibits the greedy choice property and optimal substructure, making this approach effective and optimal. The runtime of the algorithm is dominated by the sorting step, which is typically O(n log n) for n clients.

Problem 4

Part a)

There are N days, the list of deadlines on each day is: $d = [d_1,\ d_2,\ \ldots,\ d_N]$, with $d[i]$ deadlines on day i.

The given situation is a zero sum game, meaning that any deadlines Xun handles directly takes away from deadlines that Yuntao handles and vice versa, so both students will be attempting to maximize their score which will inherently minimize the other student's score. This type of problem is known as a maximin or minimax problem. In a zero sum game, a maximin and a minimax problem are identical. The minimax problem is formally stated as:

$$v_x = \max_{a_x} \min_{a_y} v_x(a_x, a_y)$$

In this equation, v_x represents the value function of Xun, a_x is the decision made by Xun, and a_y is the decision made by Yuntao. In essence, Xun wants to make a decision that will ensure the maximal value function for herself after Yuntao makes whatever decision minimizes the value for Xun. This becomes a recursion problem as after Xun and Yuntao make one decision each, they are left with the same minimax problem but with a reduced list of deadlines. For example, if Xun and Yuntao both decided to do 1 day of deadlines each, they would be left with a minimax problem on $d = [d_3,\ d_4,\ \ldots,\ d_N]$

To properly evaluate the value function for Xun at any point, the 3D decisions Xun, is able to take must be evaluated, and for each of these decisions, all of Yuntao's decisions must be evaluated, at this point another value function for Xun must be evaluated. Another way to interpret this problem is to recognize that Yuntao's value function follows the same process as Xun's value function, so for the sake of the evaluating runtime, you can view the recursive cycle as: for every value function, evaluate a value function at each of the 3D decisions that can be made. The runtime of making any decision is constant, because of this, the recurrence can be stated as:

$$T\left(n\right) = \sum_{i=1}^{3D} T(n-i) + \theta(1)$$