

Homework 11

Problem 1:(a)

Homework 11 stds

Problem 11.1

(a) We are given the sequence $\langle 3, 10, 2, 4 \rangle$

$m = 5$ (size of hashtable)

$$h_1(k) = k \bmod 5, \quad h_2(k) = 4k \bmod 8$$

Insertion 1: $k = 3$

$$h_1(3) = 3 \bmod 5 = 3$$

$$h_2(3) = (4 \cdot 3) \bmod 8 = 12 \bmod 8 = 4$$

$i = 0$

$$h(3, 0) = (h_1(3) + 0 \cdot h_2(3)) \bmod 5 = (3 + 0) \bmod 5 = 3$$

slot 3 is free \Rightarrow 3 inserted at index 3

$$\Rightarrow$$

0	
1	
2	
3	3
4	

Insertion 2: $k = 10$

$$h_1(10) = 10 \bmod 5 = 0$$

$$h_2(10) = (4 \cdot 10) \bmod 8 = 40 \bmod 8 = 0$$

$i = 0$

$$h(10, 0) = (h_1(10) + 0 \cdot h_2(10)) \bmod 5 = (0 + 0) \bmod 5 = 0$$

slot 0 is free \Rightarrow 10 inserted at index 0

0	10
1	
2	
3	3
4	

Insertion 3: $k = 2$

$$h_1(2) = 2 \bmod 5 = 2$$

$$h_2(2) = (4 \cdot 2) \bmod 8 = 8 \bmod 8 = 0$$

$i = 0$

$$h(2, 0) = (h_1(2) + 0 \cdot h_2(2)) \bmod 5 = 2 \bmod 5 = 2$$

slot 2 is free \Rightarrow 2 inserted at index 2

0	10
1	
2	2
3	3
4	

Insertion 4: $k = 4$

$$h_1(4) = 4 \bmod 5 = 4$$

$$h_2(4) = (4 \cdot 4) \bmod 8 = 16 \bmod 8 = 0$$

$i = 0$

$$h(4, 0) = (h_1(4) + 0 \cdot h_2(4)) \bmod 5 = 4 \bmod 5 = 4$$

\Rightarrow

0	10
1	
2	2
3	3
4	4

(b) the hashCode function in our code is defined as follows:

```
int hashCode(int key) {  
    return key % maxSize;  
}
```

->modular hash function

In this case, $h'(key) = key \% maxSize$ where the default maxSize we declared is 11.

The modular hash function is effective for various reasons:

1. It is efficient : the modulo operation($key \% maxSize$) is not computationally expensive, working in a constant time of $O(1)$ (making it efficient for frequent insertions or lookups).
2. It is uniformly distributed: For small integer keys, the modulo operation tends to provide a good distribution of keys(especially when the keys are randomly distributed). This is helping with the minimization of collisions and keeps a balance of the load across the table slots. The effectiveness depends on the choice of maxSize. A prime number is optimal, as prime numbers tend to produce uniform distributions, reducing clustering, when dealing with keys that are multiples of smaller numbers in particular.
3. Easy to understand implementation: it does not require additional complex computations.

Testing with the data I gave as an example in the implementation part in main:

Keys: 1, 2, 3, 1(inserted the 2nd time, we will need to update the value for this key)

Values: 10, 20, 30, 40

1.Inserting Key 1:

$hashCode(1) = 1 \% 11 = 1$

The key 1 is inserted at index 1

2.Inserting Key 2:

$hashCode(2) = 2 \% 11 = 2$

The key 2 is inserted at index 2

3. Inserting Key 3:

$hashCode(3) = 3 \% 11 = 3$

The key 3 is inserted at index 3

4. Inserting Key 1(again) :

$hashCode(1) = 1 \% 11 = 1$

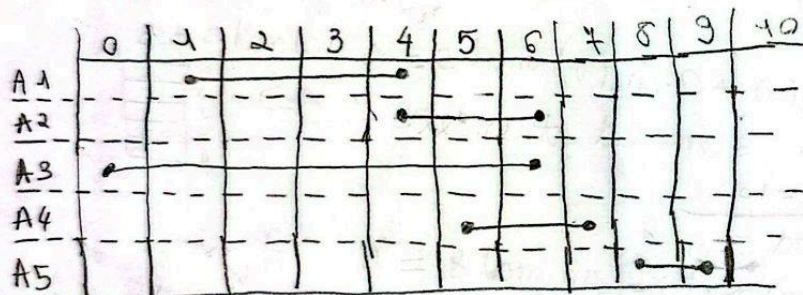
Since the key already exists, it updates the value to 40 (as described in insertNode).

No collisions occur during these insertions as all the keys all hash to unique positions in the table.

Problem 2:

Problem 11.2

(a) Given a set of activities with start and finish time, our goal is to select the maximum number of not overlapping activities. The Greedy Approach would be to always pick the activity with the shorter duration (duration = finish - start). We can show that this is not always the globally optimal solution, by providing a counter example.



Sorting by duration:

- 1) A5 (8-9) duration = 1, start = 8
- 2) A2 (4-6), duration = 2, start = 4
- 3) A4 (5-7) duration = 2, start = 5
- 4) A1 (1-4), duration = 3, start = 1
- 5) A3 (0-6) duration = 6, start = 0

Optimal solution (=) Maximum number of overlapping activities. That would be: A1 (from 1 to 4), A4 (from 5 to 7), A5 (from 8 to 9) } \Rightarrow 3 activities

Using the greedy approach, we would choose the first three activities based on duration (in ascending order):

A5 (8-9)
A2 (4-6)
A4 (5-7) } these 2 are overlapping \Rightarrow Max here is 2 activities (A5 (8-9), A4 (5-7)) which is worse than optimal

In conclusion, the "shortest duration first" does not lead to the global optimal solution.

CS Scanned with CamScanner

(b)

GREEDY_LATEST_START(A)

1. selected = emptyList

/*initialize the list of selected activities*/

```

2.    current_start_limit = 109
/*this will keep track of the latest time an activity can finish*/

3.    repeat
4.        best_activity = NIL
        /*reset best activity after each iteration*/
5.        for each activity in A do
6.            (start, finish) = activity
            /*divide each activity into its start and finish time*/
7.            if finish <= current_start_limit then
                /*check whether the activity finishes before the current
limit*/
8.                if best_activity = NIL or start > best_activity.start then
9.                    best_activity = activity
                        /*select the one with the latest start time that
fits*/
10.            if best_activity = NIL then
                /*if no suitable activity was found*/
11.                break
                /*we stop the loop as no more non-overlapping activities can be
added*/
12.            append best_activity to selected
                /*add the selected activity at the end of the result list*/
13.            current_start_limit = best_activity.start
                /*update the limit so the next selected activity finishes before this
one starts*/
14.    until false
/*infinite loop controlled by the break condition above*/
15.    return selected
/*return the list of non-overlapping activities chosen*/

```

Pseudocode breakdown

- step 1: we select the first activity => the one with the latest start time that fits the current time window(initially very large: 10⁹)
- step 2: update the time window: after we select an activity, current_start_limit gets updated to the start of the selected activity=> the next activity we choose must finish before this start time
- step 3: then we select the next activity(that finishes before current_start_limit). if it fits that requirement => we add it at the end of

the selected list and update the curent_start_limit to the start time of this new activity

- step 4: repeat this process until no more activities finish before the current_start_limit)