Before you start:

Homework Files

You can download the starter files for coding as well as this *tex* file (you only need to modify *homework2.tex*) on canvas and do your homework with latex. Or you can scan your handwriting, convert to pdf file, and upload it to canvas before the due date. If you choose to write down your answers by hand, you can directly download the pdf file on canvas which provides more blank space for solution box.

Submission Form

A pdf file as your solution named as VE281_HW2_[Your Student ID]_[Your name].pdf uploaded to canvas
Estimated time used for this homework: **3-4 hours.**

0 Student Info

Your name and student id:

Solution: Yinchen Ni 520370910026

1 Selection Algorithms (14 points)

(a) For random selection algorithm, in what cases do we encounter the worst-case or the best-case scenario? Is it related to the input sequence? What are the respective run-time under these circumstances? (7 points)

Solution: The runtime is not related to the input sequence, but depend on the pivot.

- best-case: when we partition the list into two halves and continue with only the half we are interested in. The time complexity is $O(n)$
- worst-case: when we happen to pick up the largest/smallest element as a pivot. The time complexity would be $O(n^2)$.

(b) As for deterministic selection algorithm, we have discussed about the divide-into-groups-of-5 strategy in class. What about we divide the sequence into groups of 7? Find out the recurrence relation as well as the worst-case time complexity of this approach. What are the advantages and disadvantages of having greater group size (i.e. from 5 to 7)? (7 points)

Solution: The overall time complexity does not change.

- Group the array into groups-of-7 and sort to find the median array: $\Theta(n)$
- Apply D-select to the median array: $T(n/7)$
- **At least**(worst case) ~ $(4/7) \times (1/2) = 2/7$ elements is smaller than $x_{k/2}$

Recurrence relation: $T(n) \leq cn + T(\frac{n}{7}) + T(\frac{5}{7}n)$, where we suppose $T(n) \leq an$ and can choose $a = 7c$, so that $T(n) = O(n)$.

- advantages: The D-select for the median array becomes faster.
- disadvantages: It takes more effort to group the arrays and sort to find the median.

2 Hashing Zoo (29 points)

Suppose Prof. Blue Tiger is using a hash table to store information about the grades of his students. The keys are strings and the values are integers. Furthermore, he uses a very simple function t where the hash code of a string is the integer representing its first letter. For example:

- t(”Blue Tiger”) = 1
- t(”Red Flamingo”) = 17
- t(”Glass Frog”) = 6

And we have:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Now, assume we are working with a hash table of size 10 and the hash function $h(t) = t \ \% \ 10$. This means that ”Red Flamingo” would hash to 17, but ultimately fall into bucket 17 % 10 = 7 of our table. For this problem, you will determine where each of the given name lands after inserting a sequence of values using three different collision resolution schemes:

- linear probing
- quadratic probing
- double hashing with $h_i(t) = h(h(t) + ((16 - t) \ \% \ 6) * i)$

For each of these three collision resolution schemes, determine the resulting hash table after inserting the following (*key*, *value*) pairs in the given order:

1. (”Blue Tiger”, ”100”)
2. (”Gold Monkey”, ”65”)
3. (”Red Flamingo”, ”88”)
4. (”Glass Frog”, ”96”)
5. (”Rainbow Horse”, ”80”)
6. (”Honeydew Alligator”, ”70”)
7. (”Pink Elephant”, ”101”)

Every incorrect value counts for 1 point.

2.1 Linear Probing (7 points)

Please use the **linear probing** collision resolution method to simulate the given insertion steps, and then show the final position of each (*key*, *value*) pair inside the related buckets below.

Solution: Keys are in short form. e.g. "BT" stands for "Blue Tiger"

Index	0	1	2	3	4
Key	HA	BT			
Value	70	100			
Index	5	6	7	8	9
Key	PE	GM	RF	GF	RH
Value	101	65	88	96	80

2.2 Quadratic Probing (7 points)

Please use the **quadratic probing** collision resolution method to simulate the given insertion steps, and then show the final position of each (*key*, *value*) pair inside the related buckets below.

Solution:

Index	0	1	2	3	4
Key	GF	BT		HA	
Value	96	100		70	
Index	5	6	7	8	9
Key	PE	GM	RF	RH	
Value	101	65	88	80	

2.3 Double Hashing (7 points)

Please use the **double hashing** collision resolution method to simulate the given insertion steps, with the double hash function $h_i(t) = h(h(t) + ((16 - t) \% 6) * i)$, and then show the final position of each (*key*, *value*) pair inside the related buckets below.

Solution:

Index	0	1	2	3	4
Key	GF	BT	RH	HA	
Value	96	100	80	70	
Index	5	6	7	8	9
Key	PE	GM	RF		
Value	101	65	88		

2.4 Possible Insertion Order (8 points)

Suppose you have a hash table of size 10 storing Blue Tiger's family members' favorite letter which shares the same t(key) and h(t). It uses open addressing with linear probing. After entering six values into the empty hash table, the state of the table is shown below.

Index	0	1	2	3	4	5	6	7	8	9
Key					O	P	G	Q	H	F

How many insertion orders are possible? Explain your answer clearly.

Solution: 24
We first calulate $h(t(key))$ for each key.
h(t('O')) = 4, h(t('P')) = 5, h(t('G')) = 6, h(t('Q')) = 6, h(t('H')) = 7, h(t('F')) = 5.
We find that the letters 'O', 'P', 'G' are in the place it supposed to be. The letters 'H' and 'Q' are shifted by one to the right, while 'F' is shifted by four.
Therefore, 'H' must be inserted after index 7 is taken. 'Q' must be inserted after index 6 is taken. 'F' must be inserted after index 6 to 8 is taken. We conclude that, G < Q < H < F, P < F, where '<' means inserted before.
There are 4 choices to place P and 6 choices to place Q, so in total of 24 possible cases.

3 Basic Hashing Analysis (13 points)

3.1 Lazy Deletion (4 points)

In the lectures, we have mentioned one way to implement **remove()** which is simply marking the bucket as *deleted*. This method is easy to understand and implement. However, do you think there is any backwards of this implementation of **remove()**?

Solution:

1. It increase the time of **find()**.
2. It still occupies extra space and is not totally removed.

3.2 Alternative for Remove (9 points)

Besides the usage of *deleted*, Blue Tiger thinks of another approach to improve the performance. The strategy he proposes is that when we need to remove a certain element x in hash table, we first find the last element y in the cluster where x locates. Then we swap x and y, and simply delete x. Do you think this strategy will work? If not, please propose a way to fix this strategy without marking the bucket as *deleted*. Suppose linear probing is used to avoid collision.

Solution: No, this strategy will not work.
This is because y may be inserted with another h(key), which is different from the key of x. So that the swap would destory the hash table.
Instead, we should continue probing until we find an element z that shares the same h(key) as x, and swap x and z. If we cannot find such z, then just delete x.

4 Hashing with Relocation (24 points)

Hash table is an efficient data structure in general, but it could have poor performance in worst case. After reading hundreds of papers about hash table, Ssy propose a new way to improve the current strategies. In the hash table he proposed, every element is stored along with its probe sequence lengths (PSL). The PSL of an element is defined as the difference between its desired bucket and the bucket it stays. For example, an element x is stored in the hash table, and $h(x) = 2$. But due to collision, it is stored at $hashtable[5]$, then PSL for x is $PSL = 5 - 2 = 3$.
For **insert()**, if the bucket that the key hashes to is empty, we can simply insert the key. If it is occupied, we start linear probing. When encountering an occupied bucket, we compute PSL the new key would have if inserted in that bucket, compare it with PSL of the existing key. If PSL of the new key is greater than the existing key's, we will insert the new key to this bucket and the existing key will be taken out to insert with the same process. Otherwise, we will probe the next bucket. This whole process ends when we encounter an empty bucket.

4.1 Optimized find() (9 points)

The simplest way of $find()$ is to check the bucket to which the key hashes. If the key is not found and the bucket is occupied, we start linear probing. However, this strategy is not efficient enough. Base on the strategy of $insert()$ mentioned above, propose a more efficient way to implement $find()$.

Solution: We first check the bucket to which the key hashes. If the key is not found and the bucket is occupied, we start linear probing until the hashed key of the element of that bucket is different from (or is empty) the hashed key of the element we want to find.

4.2 Optimized remove() (9 points)

As simply marking bucket as *deleted* during **remove()** will cause poor performance, please propose a better strategy to implement **remove()** in this special hash table.
Hints: make use of PSLs, the main idea is similar to **insert()**.

Solution: We first remove the element. Then we start linear probing, until we encounter a bucket contained element with PSL = 0, or is empty.

4.3 Other Improvements? (6 points)

Explain why this special hash table has a better performance than the one we talked in the lectures. Can you think of other improvements can be done to improve the performance? Think about how to avoid worst case to happen, answer this question briefly in 2-3 sentences.

Solution: This hash table has a better performance since all the elements with same hash key is put together. Worst case happens when all the elements are clustring together and we are going to delete the first element, then all the elements after need to be moved. Then we may consider change the hash function so that the hash can be more uniform. Perhaps it's also better to do separate chaining...

5 Longest Probing Sequence (20 points)

When we are using linear probing, it is very easy to form large clusters in our hash table. These clusters can affect the performance of our program greatly in turn. So we want to find the expected longest probing sequence and go deep into analysis of linear probing. Finish the four steps below. Suppose we have a hash table of size m with $n \leq m/2$ elements.
a. Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability is at most 2^{-k} that the i th insertion requires strictly more than k probes.
b. Show that for $i = 1, 2, \ldots, n$, the probability is $O\left(1/n^2\right)$ that the i th insertion requires more than $2 \lg n$ probes.
Let the random variable X_i denote the number of probes required by the i th insertion. You have shown in part (b) that $\Pr\left\{X_i > 2 \lg n\right\} = O\left(1/n^2\right)$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.
c. Show that $\Pr\{X > 2 \lg n\} = O(1/n)$.
d. Show that the expected length $\mathrm{E}[X]$ of the longest probe sequence is $O(\lg n)$.
Hints: Recall the formula for expectation is $E[X] = \sum_k P[X = k] \cdot k < P[X \leq k] \cdot k + P[X > k] \cdot X_{max}$.

Solution: To simplify the problem, we consider in worst case $n = m/2$.

a. Since half the bucket is occupied and the other half is empty. So each time we probe, there is 1/2 of the probability that the bucket is empty. So $P[\text{more than } k \text{ probes}] = P[\text{all occupied in those } k \text{ buckets}] = 2^{-k}$.

b. Result from **a.**, $P[\text{more than } 2 \lg n \text{ probes}] = 2^{-2 \lg n} = 1/n^2 = O(1/n^2)$.

c. $P[X > 2\lg n] = 1 - P[X \leq 2\lg n] = 1 - P[\max X_i \leq 2\lg n] = 1 - \prod_{i=1}^{n} P[X_i \leq 2\lg n] = 1 - \prod_{i=1}^{n}(1 - 1/n^2) = 1 - (1 - 1/n^2)^n \approx 1 - (1 - \frac{1}{n^2} \cdot n) = O(1/n)$.

d. Let $k = 2\lg n$, we get $P[X > k] = \exp(-k/2)$. So $E[X] = \sum_k P[X = k] \cdot k < P[X \leq k] \cdot k + P[X > k] \cdot X_{max} = k \cdot (1 - \exp(-k/2)) + \exp(-k/2) \cdot n$(since we would at most do n times probing) $= k - k \cdot \exp(-k/2) + \exp(-k/2) \cdot \exp(k/2) \approx k + 1 \approx 2\lg n = O(\lg n)$.