1 Hash Fruits (24 points, after Lec7)

Suppose sxt is using a hash table to store information about the color of different kinds of fruit. The keys are strings and the values are also strings. Furthermore, he uses a very simple hash function where the hash value of a string is the integer representing its first letter (all the letters are in lower case). For example:

- h(”apple”) = 0
- h(”banana”) = 1
- h(”zebrafruit”) = 25

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 compression function $c(x) = x \% 10$. This means that ”zebrafruit” would hash to 25, but ultimately fall into bucket 25 % 10 = 5 of our table. For this problem, you will determine where each of the given fruits lands after inserting a sequence of values using three different collision resolution schemes:

- linear probing
- quadratic probing
- double hashing with $h'(k) = q - (h(k) \% q)$, where $q = 5$

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. (”banana”, ”yellow”)
2. (”blueberry”, ”blue”)
3. (”blackberry”, ”black”)
4. (”cranberry”, ”red”)
5. (”apricot”, ”orange”)
6. (”lime”, ”green”)

Every incorrect value counts for 1 point.

1.1 Linear Probing (8 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: Each wrong position of one key-value pair counts for 2 points.

Index	0	1	2	3	4
Key	apricot	banana	blueberry	blackberry	cranberry
Value	orange	yellow	blue	black	red
Index	5	6	7	8	9
Key	lime				
Value	green				

1.2 Quadratic Probing (8 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	apricot	banana	blueberry	cranberry	
Value	orange	yellow	blue	red	
Index	5	6	7	8	9
Key	blackberry		lime		
Value	black		green		

1.3 Double Probing (8 points)

Please use the **double probing** collision resolution method to simulate the given insertion steps, with the double hash function $h'(k) = q - (h(k) \% q)$ and $q = 5$, and then show the final position of each (*key*, *value*) pair inside the related buckets below.

Solution:

Index	0	1	2	3	4
Key	apricot	banana	cranberry	lime	
Value	orange	yellow	red	green	
Index	5	6	7	8	9
Key	blueberry				blackberry
Value	blue				black

2 Hash! Hash! Hash! (14 points, after Lec8)

2.1 Possible Insertion Order (7 points)

Suppose you have a hash table of size 10 uses open addressing with a hash function $H(k) = k \bmod 10$ and 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			62	43	24	82	76	53		

Which of the following insertion orders is / are possible? Select all that apply and clearly state why it is possible.

A. 76, 62, 24, 82, 43, 53

B. 24, 62, 43, 82, 53, 76

C. 76, 24, 62, 43, 82, 53

D. 62, 76, 53, 43, 24, 82

E. 62, 43, 24, 82, 76, 53

Solution: From the given hash table, we know that:

- 82 should be inserted after 62, 43, and 24 (2 points)
- 53 should be inserted after 43, 24, 82, ad 76 (2 points)

Hence, **C and E** are correct. (2 points)

2.2 Wrong Delete (7 points)

William implements a hash table that uses open addressing with linear probing to resolve collisions. However, his implementation has a mistake: when he erases an element, he replaces it with an empty bucket rather than marking it as deleted! In this example, the keys are strings, with the hash function:

```
size_t hash(string s) {
  return s.empty() ? 0 : s[0] - 'A';
}
```

and the hash table initially contains 100 buckets. After which of the following sequences of operations will the hash table be in an invalid state due to erased items being marked empty rather than as deleted? In this case, a hash table is invalid if subsequence "find" or "size" operations do not return the correct answer.

A. insert "A1"; insert "B1"; insert "C1"; erase "A1"; erase "C1";

B. insert "A1"; insert "A2"; insert "A3"; erase "A3"; erase "A2";

C. insert "B1"; insert "C1"; insert "A1"; insert "A2"; erase "C1";

D. insert "A1"; insert "B1"; insert "A2"; erase "B1"; insert "B2";

E. none of the above

Please clearly state why you choose that answer.

Solution: Answer: **C** (3 points) (if you give C **and** other answers, you will get 2 points.)

When the "deleted" bucket is marked as "empty" by mistake, invalid hash table will exist when there are some elements in those buckets behind the deleted bucket, and those elements are inserted after one or several collisions. In other words, there are some elements that when we want to find those elements, we must pass the deleted bucket.

According to the background knowledge above, we know that **C** is the only one correct answer. After insertion, we have:

Index	0	1	2	3	...
Key	A1	B1	C1	A2	...

After erasing "C1", we have:

Index	0	1	2	3	...
Key	A1	B1		A2	...

Then we see that at index=2, there is an empty "hole". And when we want to find "A2" in this hash table, it will first go to index=0, and does not find "A2"; Then it keeps traversing through the hash table and find that at index=2, the slot is empty, which indicates that there is no "A2" in this hash table, which leads to invalid case. (4 points)

3 Rehashing (22 points)

3.1 Rehash to be smaller (9 points)

After learning the concept of hash table and rehashing, Coned is thinking about the possible optimization in terms of the space usage. His intuitive idea is as following:
In the lecture, by rehashing we always enlarge the hash table. Obviously, if we delete many elements afterwards, the load factor will drop drastically, and we do not need such a large hash table any more.
Prove that if we resize the hash table to be **halved** when the load factor is **less than 0.125**, the average time complexity of insert will still be $O(1)$.
Hints: Assume that such resizing will not result in a hash table smaller than the original one. For example, if initially we have a hash table with 10 buckets, the hash table will **always have at least** 10 buckets.

Solution:
We first apply amortized analysis on the time complexity of a **single operation** regardless of *find()*.
Since in the lecture, we have proved that rehashing to be a bigger hash table will not make a difference to average case time complexity, we only focus on situation that may trigger "rehashing to be smaller" here.
Suppose we have n buckets and 0 elements in the table at the beginning of the procedure. During the whole procedure, there are at least totally $m+k$ operations, including m insertions and k deletions. The base case that we consider is:

1. Insert m elements so that the table is rehashed to have a size of $2n$, $O(m)$ time
2. Delete k elements so that the table is to be rehashed again, $O(k)$ time
3. Rehash the $\frac{1}{4}n$ elements, since $\frac{1}{4}n$ is bound to be smaller than m, $O(m)$ time

We have $\frac{2O(m)+O(k)}{m+k} = O(1)$ time for a single operation by amortized analysis for this base case. Then we can extend this conclusion to larger case with multiple rehashing by induction(omitted here, similar to the proof of why rehashing to be large will not make a difference to the time complexity). Till this, we can conclude that the operations totally have an average time complexity of $O(1)$.
Given that the number of insertion m should always be larger than the number of deletion k, suppose the average time complexity is not $O(1)$:

$$T(m+k) = \frac{m * O(f(n)) + k * O(g(n))}{m+k} = \frac{O(m * f(n) + kg(n))}{m+k} = \Omega(\frac{O(m * f(n))}{2m}) = \Omega(f(n))$$

which is not $O(1)$, and we have a conflict here. Therefore, the average time complexity of insertion is $O(1)$.

3.2 Rehash to be not that large (9 points)

After their different ideas in terms of selection algorithm, our master of algorithm, William, again has a different solution to the problem of space usage with Coned. He thinks that the critical point

that makes the hash table to be too big is that it is **enlarged by too much each time**. Typically, we will double the size of the hash table when rehashing.
William wants to change this setting to be: Suppose the original hash table size is m, and the size of the new hash table will be a **prime number close to** $m + k$ after rehashing, where k is a constant. If we rehash as William puts forward, will the average time complexity of insert still be $O(1)$?

Solution:
No. Again we apply the amortized analysis:
The base case doesn't change, but when we are extending it to multiple rehashing:
Suppose we will do t times of rehashing and assume that the hash table size will approximately be equal to $m + i * k$, then the average time complexity of insertion will be(constant coefficient has been omitted):

$$T(n) = \frac{\sum_{i=0}^{t-1} O(m + i * k)}{n} = \frac{O(tm + t^2 k)}{n}$$

and we have $n = (m + tk) * c$, where c is the threshold for the load factor. Since m, k, c are all constants here, $t = O(n)$, then

$$T(n) = O(t) = O(n)$$

3.3 Rehashing always works? (4 points)

Please propose a new strategy of rehashing (similar to what Coned and William has done), so that the average time complexity after amortized analysis is no longer $O(1)$ for insert.
Hints: previously we rehash when the load factor is larger than a certain number c. What if this c is not a constant?

Solution: Rather than rehash when the load factor exceeds its threshold, here we rehash when the number of elements exceed its dynamic threshold. For example, if the threshold just increase by 1 each time, which means that for each insertion we will do the rehashing, then the average time complexity of insertion is $T(n) = O((n^2 + n)/n) = O(n)$, which is no longer $O(1)$.

Grading notes from sxt: actually these exercises in terms of rehashing aim to show you the idea of amortized analysis and the fact that with some very simple modifications, operations in hash table will be less efficient to a great extent(from $O(1)$ to $O(n)$). Don't worry too much about the grade. Trying to solve those problems is sufficient to be rewarded with some points from my perspective:)

4 Hash Table Size (10 points)

Suppose we want to design a hash table containing at most 1162 elements using quadratic probing. We require it to guarantee successful insertions, S(L) < 2 and U(L) < 4. Please determine a proper hash table size and show all intermediate steps.

Solution: Since we have to guarantee the successful insertions for quadratic probing, the load factor can never be less than 1/2.

$$S(L) = \frac{1}{L} ln \frac{1}{1-L}, S(1/2) = 1.39 < 2$$

$$U(L) = \frac{1}{1-L}, U(1/2) = 2 < 4$$

Both satisfied, so finally we have $\frac{1162}{1/2} = 2324$ and we pick the prime number 2333 as the hash table size.

5 Bloom Filter (10 points, after Lec9)

Suppose there is an array A of $n = 17$ bits. Each bit is either 0 or 1. And we have 3 hash functions h_1, h_2, h_3, each mapping inside $\{0, 1, \dots, n-1\}$:

$$\begin{aligned} h_1(x) &= x \ \% \ n \\ h_2(x) &= (3 * x + 2) \ \% \ n \\ h_3(x) &= (5 * x + 1) \ \% \ n \end{aligned}$$

Initially the array is all-zero.

i) Roihn first inserts one element 12 into the bloom filter. Please write down the current values of entries of array A. (2 points)

Solution:

$$\begin{aligned} h_1(x) &= 12 \ \% \ 17 = 12 \\ h_2(x) &= (3 * 12 + 2) \ \% \ 17 = 4 \\ h_3(x) &= (5 * 12 + 1) \ \% \ 17 = 10 \quad \text{(1 point)} \end{aligned}$$

Hence, the array $A = [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0]$. (1 point)

ii) Then Roihn inserts 3 into the bloom filter. Please write down the current values of entries of array A. (2 points)

Solution:

$$\begin{aligned} h_1(x) &= 3 \ \% \ 17 = 3 \\ h_2(x) &= (3 * 3 + 2) \ \% \ 17 = 11 \\ h_3(x) &= (5 * 3 + 1) \ \% \ 17 = 16 \quad \text{(1 point)} \end{aligned}$$

Hence, the array $A = [0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1]$. (1 point)

iii) After inserting 3, Roihn finds that it is actually a mistake, and would like to remove 3 from the bloom filter. Can he remove the element 3 from the fliter? If so, how it can be achieved? (2 points)

Solution: He cannot remove element from the bloom filter. (1 point)

The reason is: actually bloom filter does not support remove function, since we cannot assure that the hash values calculated for element 3 are not used by other elements. If we change those 1s to 0s, this may cause false negative judgement. (1 point)

iv) Now Roihn wants to find out whether some of the elements are in this filter. Does element 4 in this filter? What about 20? Please clearly show your steps. (4 points)

Solution: For $x = 4$,

$$h_1(x) = 4 \ \% \ 17 = 4$$
$$h_2(x) = (3 * 4 + 2) \ \% \ 17 = 14$$
$$h_3(x) = (5 * 4 + 1) \ \% \ 17 = 4 \quad \text{(1 point)}$$

Since $A[14] = 0$, element 4 is **NOT** in the filter. (1 point)

For $x = 20$,

$$h_1(x) = 20 \ \% \ 17 = 3$$
$$h_2(x) = (3 * 20 + 2) \ \% \ 17 = 11$$
$$h_3(x) = (5 * 20 + 1) \ \% \ 17 = 16 \quad \text{(1 point)}$$

Since $A[3] = A[11] = A[16] = 1$, element 20 is in the filter. (1 point)

6 Coding Assignment (40 points, after Lec8)

For this question, you will be tasked with implementing a hash table function. Download the starter file *hashtable.h* from Canvas.

Your hash table will support the following four operations:

```
bool insert(pair<Key, Val>);

size_t erase(Key);

Val& operator[](Key);

size_t size();
```

insert takes a key and a value, and inserts them into the hash table. If the new key is already in the hash table, then the operation has no effect. insert returns whether the key was inserted.

erase takes a key, and removes it from the hash table. If the key isn't already in the hash table, then the operation has no effect. erase returns how many items were deleted (either 0 or 1).

operator[] takes a key, and returns a reference to the associated value in the hash table. If the key was not previously present in the table, it will be inserted, associated to a default-constructed value.

size returns the number of key-value pairs currently stored in the hash table. insert and operator[] can both increase the hash table's size, while erase can decrease it.

The provided code includes an optional private function called **rehash_and_grow**. You may find it useful to put logic for increasing the number of buckets into this function, but you are not required to.

Implementation Requirements:

Your hash table will be implemented using **quadratic probing**, with deleted elements to support erasing keys. The key-value pairs will be stored in *buckets*, a member variable in *Hashtable* of type *std::vector<Bucket>*.

```
template<typename Key, typename Value, typename Hasher = std::hash<Key>>
struct Bucket {
  BucketType type = BucketType::Empty;
  Key key;
  Value value;
};
```

A bucket has a type, a key and a value. Here the *BucketType* is an enum class type:

```
enum class BucketType {
  Empty, // bucket contains no item
  Occupied, // contains an item
  Deleted // is a deleted element
};
```

To refer to its three possible values, write as *BucketType::Empty*, *BucketType::Occupied*, and *BucketType::Deleted* for comparison.

For the hash function, you will use the STL's hashing functor, *std::hash*, and mod it by the current number of buckets. Assume the given key is k, then:

```
Hasher hasher;
size_t desired_bucket = hasher(k) % buckets.size();
```

You can get more details in the given starter file *hashtable.h*, and we also provide you a simple testing cpp file, which can give you a good start for your testing.

For submission, you only need to upload the header file *hashtable.h*.

Reference

Assignment 2, VE281, FA2020, UMJI-SJTU.
Homework 1, VE281, SU2021, UMJI-SJTU.
Lab Assignment 7, EECS281, FA2020, University of Michigan.
Lecture Notes, CS312, Cornell University.