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

For homework 2, there are two parts of submission:

1. A pdf file as your solution named as VE281_HW2_[Your Student ID]_[Your name].pdf uploaded to canvas
2. Code for question 3 uploaded to joj (there will be hidden cases but the time restriction is similar to the pretest cases).

For the programming question(question 3), you must make sure that your code compiles successfully on a Linux operating system with g++ and the options:

```
-std=c++1z -Wconversion -Wall -Werror -Wextra -pedantic
```

Estimated time used for this homework: **3-4 hours.**

Great credits to 2020FA VE281 TA Group and enormous thanks to 2021SU VE281 TA Roihn!!!

0 Student Info (0 point)

Your name and student id:

Solution: 陶思郡 Sijun Tao 519021910906

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:

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.

$h_i(x) = (h(x) + i * g(x)) \ \% \ n$

$= (x \% 10 + i * [5 - (x \% 10 \% 5])) \% 10$

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 $H(k) = k$ mod 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: C E are possible.

① As for C, $h_0(76) = 6$, empty; $h_0(24) = 4$, empty;
$h_0(62) = 2$, empty; $h_0(43) = 3$, empty;
$h_0(82) = 2$, not empty → $h_1(82) = 3$, not empty
→ $h_2(82) = 4$, not empty → $h_3(82) = 5$, empty;
$h_0(53) = 3$, not empty ..... $h_4(53) = 7$, empty;

② As for E, $h_0(62) = 2$, empty; $h_0(43) = 3$, empty;
$h_0(24) = 4$, empty;
$h_0(82) = 2$, not empty → $h_1(82) = 3$, not empty
→ $h_2(82) = 4$, not empty → $h_3(82) = 5$, empty;
$h_0(76) = 6$, empty;
$h_0(53) = 3$, not empty ..... $h_4(53) = 7$, empty;

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"; find A2

0	1	2	3
A1	B1		A2

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: After C, the hash table will be invalid.

After C, "0", "1", "3" are occupied and "2" is empty.
As a result, when finding A2, the program will terminate at "2" and returns "not found". However, A2 exists. There exists a conflict and the hash table is not valid.

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:

Suppose we start from an empty hash table of $8M$.

Assume $O(1)$ operation to insert up to $4M$ items $\Rightarrow$ total costs $O(4M)$

Then delete down to M items $\Rightarrow$ total costs $O(3M)$

Create a new hash table of size $4M$. $\Rightarrow$ total costs $O(1)$

Rehash all M items into the new table $\Rightarrow$ total costs $O(M)$

insert new items $\Rightarrow$ total costs $O(1)$

$\Rightarrow$ Total cost for inserting $4M+1$ items is $2O(1)+3O(M)=O(M)$

$\Rightarrow$ The average time complexity of insert will still be $O(1)$

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.

Assume in total there are p elements to be inserted.

At this time, the number of rehashing n is related to p:

$$L[m+(n-1)k] < p \leq L[m+nk],$$

where L is the load factor. Then we get

$$\frac{p-m}{k} \leq n < \frac{p-m}{k} + 1$$

where n is an integer. As a result,

$$C\frac{L(m)+L(m+k)+\cdots+L(m+nk)}{p} = C\frac{L\left(\frac{(2m+nk)n}{2}\right)}{p} = O(p) \neq O(1)$$

$\Rightarrow$ The average time complexity of insert will be $O(p) \neq O(1)$

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. Prove that the average time complexity of your new algorithm is longer $O(1)$.
Hints: previously we rehash when the load factor is larger than a certain number c. What if this c is not a constant?

Solution:

Now, let $c = (\frac{1}{2})^{t+1}$, where t is the current rehashing times.

At first, $c = \frac{1}{2}$, cost for insert into an empty table of size M is $O(M)$.

then create and rehashing, cost is $O(1)+O(M) = O(M)$.

then reach the load factor $c = \frac{1}{4}$, create and rehashing $\Rightarrow O(M)$.

then reach the load factor $c = \frac{1}{8}$, create ..... and so on.

$\Rightarrow$ Total cost for inserting $\frac{M}{2}$ items is $O(M)+\cdots+O(M) = nO(M)$ $(n>1)$

$\Rightarrow$ The average time complexity of insert will be $nO(1) = O(n) \neq O(1)$

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:

$$\begin{cases} S(L) = \frac{1}{L}\ln\frac{1}{1-L} < 2 \\ U(L) = \frac{1}{1-L} < 4 \end{cases} \Rightarrow \quad L < \frac{3}{4}$$

Since load factor $L = \frac{|S|}{n} < 0.75$,

$$n > \frac{1162}{0.75} = 1549.3$$

Pick n as a prime number, then

$$n = 1553$$

5 Bloom Filter (10 points, after Lec9)

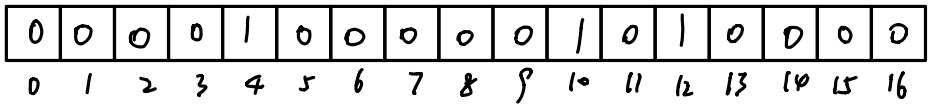
Suppose there is an array A of $n = 17$ bits. Each bit is either 0 or 1. And we have 3 hash functions h_1, h_2, h_3, each mapping inside $\{0, 1, \ldots, n-1\}$:

$$h_1(x) = x \,\%\, n$$
$$h_2(x) = (3 * x + 2) \,\%\, n$$
$$h_3(x) = (5 * x + 1) \,\%\, n$$

Initially the array is all-zero.

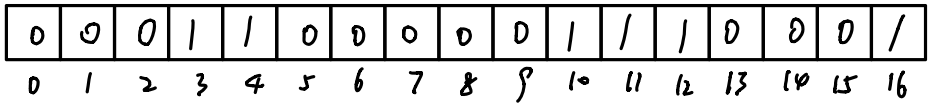
i) sxt first inserts one element 12 into the bloom filter. Please write down the current values of entries of array A. (2 points)

Solution: $h_1(12) = 12$, $h_2(12) = 4$, $h_3(12) = 10$



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

Solution: $h_1(3) = 3$, $h_2(3) = 11$, $h_3(3) = 16$



iii) After inserting 3, he 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:

Yes. He first calculate the $h_1(3)$, $h_2(3)$, $h_3(3)$.

Then, he changes $A[h_i(3)]$ from 1 to 0.

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

Solution: As for 4. $h_1(4) = 4$, $h_2(4) = 14$, $h_3(4) = 4$. $\Rightarrow$ No

As for 20, $h_1(20) = 3$, $h_2(20) = 11$, $h_3(20) = 16$ $\Rightarrow$ Yes

6 Coding Assignment (20 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.