Before you start:

Homework Files

You can download the starter files for coding as well as this *tex* file (you only need to modify *homework1.tex*) on canvas and do your homework with latex (recommended). 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 1, you need to upload a **tar ball** with two files in the following format:

- VE281_HW1_[Your Student ID]_[Your name].tar
  - ⌊ VE281_HW1_[Your Student ID]_[Your name].pdf
  - ⌊ hashtable.h

Please strictly follow the format given above!!! Everyone who does not obey the format will get **2 points** deduction!!!
Notes: No extra folders (extracting this tar should only give you two files), no space in your name (use underscore(_) instead), no brackets. One example for name of tar:
VE281_HW1_518370910000_Run_Peng.tar

For all coding questions (question 3), you need to successfully compile your code, or otherwise you will at most get half of the whole score(we will give you partial points if you implement some of the functionalities). We will have some simple test cases to test whether your code can correctly work. 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.**

0 Student Info (1 point)

Your name and student id:

Solution:

1 Hash Fruits (24 points, after Lec7)

Suppose Roihn 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					
Value					
Index	5	6	7	8	9
Key					
Value					

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					
Value					
Index	5	6	7	8	9
Key					
Value					

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					
Value					
Index	5	6	7	8	9
Key					
Value					

2 Hash! Hash! Hash! (25 points, after Lec8)

2.1 Possible Insertion Order (6 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:

2.2 Time Complexity of Hashing (12 points)

i) In a few sentences, explain why a hash table with open addressing and rehashing (doubling the size every time when the load factor exceeds a threshold) achieves insertion in $O(1)$ time. (3 points)

Solution:

ii) When rehashing is not triggered, is the insertion time $O(1)$ (does it have an upper bound)? Why? (3 points)

Solution:

iii) Show that at any given point, the amortized (distributed to all items in the hash table) cost of rehashing is $O(1)$ for each item in the hash table. (3 points)

Solution:

iv) What if during rehashing, the hash table is expanded by a factor x with $x > 2$? Is the amortized cost of rehashing still $O(1)$? If no, explain. If yes, show your work. If it depends, show both. (3 points)

Solution:

2.3 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:

3 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*.

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

Solution:

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

Solution:

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:

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 clear show your steps. (4 points)

Solution:

Reference

Assignment 2, VE281, FA2020, UMJI-SJTU.
Lab Assignment 7, EECS281, FA2020, University of Michigan.