The University of Michigan
Electrical Engineering & Computer Science
EECS 281: Data Structures and Algorithms
Fall 2024

Lab 7: Introduction to Collision Resolution

Instructions:
Work on this document with your group, then enter the answers on the canvas quiz. **This assignment is due November 11th, 11:59pm.**

Note:
This lab contains both a Canvas quiz and an autograder portion. For the Canvas quiz, you will have **three** attempts, and the best score will be kept. See the last few pages for instructions on the coding portion. **You should start early on this lab - you will likely NOT be able to finish on time if you start late. Over 100 lines of code will be needed to complete the assignment.** For additional reading, read sections 11.1 and 11.2 of the CLRS Introduction to Algorithms textbook, and visit datastructures.maximal.io/hash-tables.

You MUST **include the following assignment identifier at the top of every file you submit to the autograder as a comment. This includes all source files, header files, and your Makefile (if there is one).**

Assignment identifier: 2C4A3C53CD5AD45A7BEA3AE5356A3B1622C9D04B

1 Part A: Collision Resolution (2 points)

1. Which of the following statements is/are **FALSE?** Select all that apply.
   A. Hash tables with linear probing will require less rehashing than hash tables using separate chaining.
   B. Linear probing is better than double hashing at preventing keys in a table from clustering together.
   C. Hash tables that use linear probing have a better worst-case time complexity than hash tables that use separate chaining.
   D. Linear probing will have a better average-case time complexity than separate chaining for lookup.
   E. For both linear probing and separate chaining, collisions only occur between elements that have the same hash value.

2. Which of the following collision resolution methods is NOT a form of open addressing?
   A. separate chaining
   B. linear probing
   C. quadratic probing
   D. double hashing
   E. all of the above methods use open addressing

3. You're implementing a hash table that uses open addressing with linear probing to resolve collisions. However, your implementation has a mistake: when you erase an element, you replace it with an empty bucket rather than marking it as deleted! In this example, your 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 subsequent "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

4. Suppose you have a hash table of size 10 that uses open addressing with a hash function H(k) = k and linear probing. After entering six values into the empty hash table, the state of the table is shown below. Remember to use the modulo operator to keep hash values within the capacity of the hash table!

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.

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

2 Part B: Hash Fruits (6 points)

Suppose you are 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, let’s say you use a very simple hash function where the hash value of a string is the integer representing its first letter. For example:

```
h("apple") = 0
h("banana") = 1
h("zebrafruit") = 25
```

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

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”)

Answer questions 5-8 using the **linear probing** collision resolution method. Each question is worth 0.5 points. You may use the following table to help you solve these problems.

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
key										
value										

5. After all six fruits are inserted, where did the key ”blueberry” land? Enter a numeric index between 0 and 9.

6. After all six fruits are inserted, where did the key ”blackberry” land? Enter a numeric index between 0 and 9.

7. After all six fruits are inserted, where did the key ”cranberry” land? Enter a numeric index between 0 and 9.

8. After all six fruits are inserted, where did the key ”lime” land? Enter a numeric index between 0 and 9.

For questions 9-12, repeat the process using the **quadratic probing** collision resolution method. Each question is worth 0.5 points. The list of (key, value) pairs to insert has been reproduced below for your convenience.

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

You may use the following table to help you solve these problems.

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
key										
value										

9. After all six fruits are inserted, where did the key ”blueberry” land? Enter a numeric index between 0 and 9.

10. After all six fruits are inserted, where did the key ”blackberry” land? Enter a numeric index between 0 and 9.

11. After all six fruits are inserted, where did the key ”cranberry” land? Enter a numeric index between 0 and 9.

12. After all six fruits are inserted, where did the key ”lime” land? Enter a numeric index between 0 and 9.

For questions 13-16, repeat the process using the **double hashing** collision resolution method, with the double hash function $h'(k) = q - (h(k) \% q)$ and $q = 5$. Each question is worth 0.5 points. You may use the following table to help you solve these problems.

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
key										
value										

13. After all six fruits are inserted, where did the key ”blueberry” land? Enter a numeric index between 0 and 9.

14. After all six fruits are inserted, where did the key ”blackberry” land? Enter a numeric index between 0 and 9.

15. After all six fruits are inserted, where did the key ”cranberry” land? Enter a numeric index between 0 and 9.

16. After all six fruits are inserted, where did the key ”lime” land? Enter a numeric index between 0 and 9.

3 Part C: Coding Assignment (5 points)

For this question, you will be tasked with implementing a hash table function, loosely inspired by the STL's own unordered_map container. Download the starter file hashtable.h from Canvas.

Your hash table will support the following four operations:

- `bool insert(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 with 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 a private function called rehash_and_grow. You can earn some, but not all, available points without implementing this function. You must put logic for increasing the number of buckets into this function to earn all available points.

Implementation Requirements

Your hash table will be implemented using linear 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>
struct Bucket {
   BucketType type = BucketType::Empty;
   Key key;
   Value value;
};
```

A bucket has a type (whether it be empty, occupied, or deleted), a key and a value. The key and value are only meaningful if the bucket is occupied. 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 `BucketType::Empty`, `BucketType::Occupied`, and `BucketType::Deleted`. These values can be compared for equality.

When your hash table receives a key k, in order to find where it goes, you will use the STL's hashing functor, `std::hash`, and mod it by the current number of buckets:

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

Your hash table should probe forward from the desired bucket if that bucket is already occupied (and wrap around if it reaches the end of the buckets vector).

When your hash table becomes too full, it should increase the number of buckets. Exactly when and how you do this is up to you! A recommended default would be to double the number of buckets when the hash table's load factor exceeds 0.5, but you can play with these parameters. (Note that using prime bucket counts is **not** likely to improve your code's performance on the autograder due to its unusual hash function. You don't need anything complicated to pass all the test cases).

Submission

To submit to the autograder, create a .tar.gz file containing just hashtable.h.

tar -czvf lab7.tar.gz hashtable.h

Your hash table cannot assume anything about its keys or values, other than that they can be assigned and copied, and that the key type can be hashed and compared with == and !=.

You can work on these problems by yourself or with your group, but each individual must submit a solution to the autograder. Only students who submit code to the autograder will receive points. Every source code and header file must contain the following project identifier in a comment in the file's preamble:
// Project Identifier: 2C4A3C53CD5AD45A7BEA3AE5356A3B1622C9D04B

The names of the test cases describe which operations they'll perform:

- **I**nsert
- **B**rackets (`operator[]`)
- **E**rase

All of the tests check size.

When your program gets incorrect output, the autograder will attempt to explain why your program didn't do the right thing, so take a look at the detailed output! These won't always be helpful, but they can sometimes help pinpoint your problems. If a message is unclear, ask about it on Piazza. Time and memory constraints will be lenient for this lab. As long as you are under 3x over time and 10x over memory, you will receive full credit for each test case.

Helpful tips:

- Remember to account for deleted elements!

- Ensure that your operations are actually expected $\Theta(1)$. This means not looping over every bucket for every single operation.
- Feel free to add additional private methods to Hashtable. You shouldn't need or want any new member variables.
- Carefully look at the lab and lecture slides on each function as you implement them.