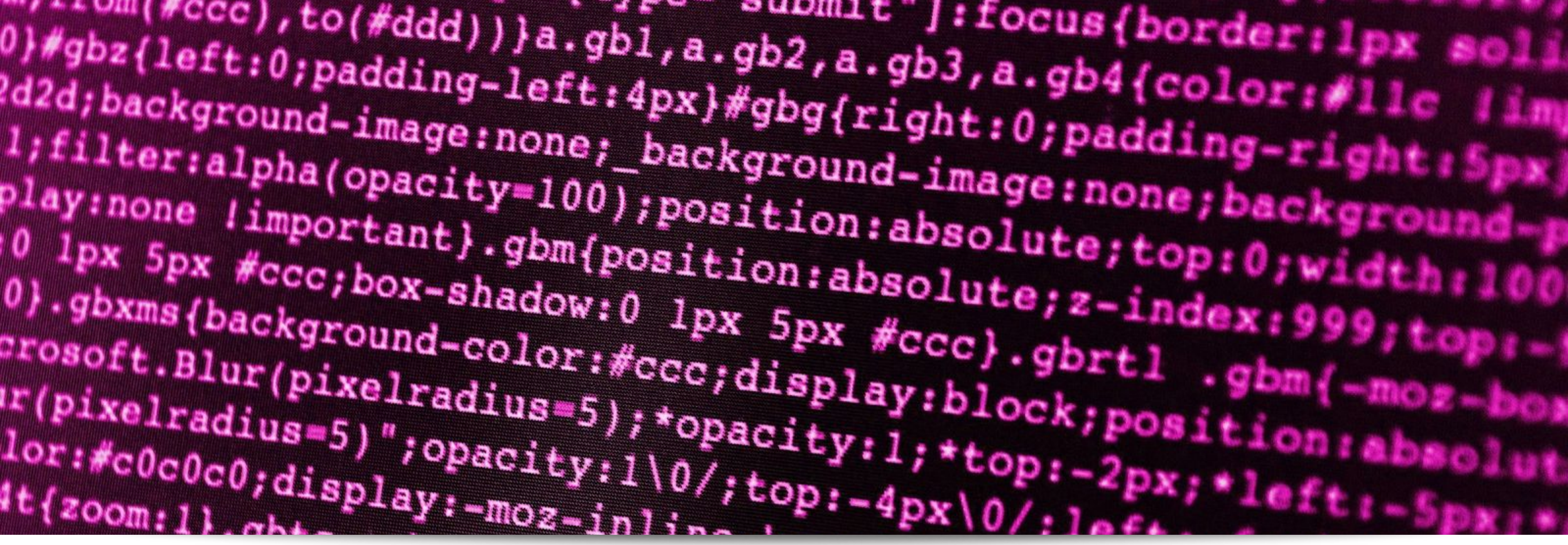


07

Week of March 16, 2020

Hash tables & Collision Resolution Methods

Agenda

- Unordered Maps (Related multiple choice questions_
- Hash Tables and Hashing Functions
- Methods of Collision Resolution
  - Separate Chaining
  - Open Addressing (e.g. forms of probing)
- Examples and Exercises for Handling Collisions
- Analyzing the Performance of a Hash Table
- Dynamic Hashing
- Handwritten Problem

Storing Student Data

- Motivating Problem:
  - you have to store the names of every student in EECS 281.
  - every student has a unique ID in the range [0, 750).
  - querying by student ID must be O(1).
- Solution:
  - store the names of each student in a vector of strings, and index by student ID to get the name of the student!

Storing Student Data

- Motivating Problem:
  - you have to store the names of every student in EECS 281.
  - every student has a **uniqname** that is a string of ASCII characters.
  - querying by **uniqname** must be O(1).
- Can we use the vector method here?
  - Not really.. How would we know which index each uniqname would go to?
  - we would need to "map" each uniqname to some sort of index so that we know where to put each uniqname!

Storing Student Data

- Motivating Problem:
  - you have to store the names of every student in EECS 281.
  - every student has a **uniqname** that is a string of ASCII characters.
  - querying by **uniqname** must be O(1).

- Can we use the vector method here?
  - Not really.. How would we know which index each uniqname would go to?
  - we would need to "map" each uniqname to some sort of index so that we know where to put each uniqname!

- Solution:
  - use an `unordered_map` and index by uniqname!

Storing Student Data

```
struct Student {
    string uniqname;
    string full_name;
    vector<double> grades;
};

std::unordered_map<string, Student>
all_students;

string get_full_name(string uniqname) {
    return all_students[uniqname].full_name;
}
```

Storing Student Data

```
struct Student {
    string uniqname;
    string full_name;
    vector<double> grades;
};

std::unordered_map<string, Student> all_students;

void add_grade(string uniqname, double grade) {
    all_students[uniqname].grades.push_back(grade);
}
```

Adding and Accessing Elements

```
using Name = string; // “Name” means “string” now
using FavColor = string;

int main() {
    unordered_map<Name, FavColor> favorite_colors;

    favorite_colors[“mrkevin”] = “orange”;
    favorite_colors[“paoletti”] = “grey”;

    cout << favorite_colors[“paoletti”] << endl; // prints “grey”
    cout << favorite_colors.size() << endl;      // prints 2
    cout << favorite_colors[“nobody”] << endl;   // ???


}
```

Adding and Accessing Elements

```
using Name = string; // “Name” means “string” now
using FavColor = string;

int main() {
    unordered_map<Name, FavColor> favorite_colors;

    favorite_colors[“mrkevin”] = “orange”;
    favorite_colors[“paoletti”] = “grey”;

    cout << favorite_colors[“paoletti”] << endl; // prints “grey”
    cout << favorite_colors.size() << endl;      // prints 2
    cout << favorite_colors[“nobody”] << endl;   // inserts “” into
table!


}
```

Adding and Accessing Elements

```
using Name = string; // “Name” means “string” now
using FavColor = string;

int main() {
    unordered_map<Name, FavColor> favorite_colors;

    favorite_colors[“mrkevin”] = “orange”;
    favorite_colors[“paoletti”] = “grey”;

    cout << favorite_colors[“paoletti”] << endl; // prints “grey”
    cout << favorite_colors.size() << endl;      // prints 2
    cout << favorite_colors[“nobody”] << endl;   // inserts “” into
table!

    cout << favorite_colors.size() << endl; // prints ???

}
```

Adding and Accessing Elements

```
using Name = string; // "Name" means "string" now
using FavColor = string;

int main() {
    unordered_map<Name, FavColor> favorite_colors;

    favorite_colors["mrkevin"] = "orange";
    favorite_colors["paoletti"] = "grey";

    cout << favorite_colors["paoletti"] << endl; // prints "grey"
    cout << favorite_colors.size() << endl;      // prints 2
    cout << favorite_colors["nobody"] << endl;   // inserts "" into
table!

    cout << favorite_colors.size() << endl; // prints 3

}
```

Using find

```
unordered_map<Name, FavColor> favorite_colors;

favorite_colors[“mrkevin”] = “orange”;
favorite_colors[“paoletti”] = “grey”;

Name input_name = argv[1];

auto found_it = favorite_colors.find(input_name);
if (found_it == favorite_colors.end()) {
   cout << “Name not found: ” << input_name << endl;
} else {
   cout << found_it->first << “‘s favorite color is: ” << found_it->second;
}
```

prevents operator[] from adding in keys that do not exist!

Common performance pitfall

```
// Two lookups

if (map.find(x) != map.end()) {
    std::cout << “Found: ” << map[x] << std::endl;
}


// One lookup

auto iter = map.find(x);
if (iter != map.end()) {
    std::cout << “Found: ” << iter->second << std::endl;
}
```

Common performance pitfall

```
// Multiple lookups
employees[x].salary -= 1000;
employees[x].title = “manager”;
employees[x].years_working = 3;

// One lookup
auto iter = employees.find(x);
iter->second.salary -= 1000;
iter->second.title = “manager”;
iter->second.years_working = 3;

// One lookup, better style
auto& employee = employees[x];
employee.salary -= 1000;
employee.title = “manager”;
employee.years_working = 3;
```

watchout for multiple instances of operator[] use with the same key

Another pitfall: forgetting reference specifier on auto

Multiple Choice Practice

Hash Applications

For which of the following applications would a hash table **NOT** be appropriate? Select all that apply.

1. printing out all of the elements in sorted order
2. storing passwords that can be looked up by uniqname
3. returning a person's name given their phone number
4. finding the kth largest element in an array
5. creating an index for an online book

Hash Applications

For which of the following applications would a hash table **NOT** be appropriate?
Select all that apply.

1. **printing out all of the elements in sorted order**
2. storing passwords that can be looked up by uniqname
3. returning a person's name given their phone number
4. **finding the kth largest element in an array**
5. creating an index for an online book

Symmetric Pairs

Two pairs (a, b) and (c, d) are symmetric if b = c and a = d. Suppose you want to find all symmetric pairs in the array. The first element of all pairs is distinct.

For example, if arr1[] = {(14, 23), (11, 2), (52, 83), (49, 38), (38, 49), (2, 11)};

The symmetric pairs are {(11, 2), (2, 11)} and {(49, 38), (38, 49)}.

What is the average-case time complexity of accomplishing this task if you use the most efficient algorithm? If hashing is involved, assume that both search and insert methods work in $\Theta(1)$ time.

1. $\Theta(1)$
2. $\Theta(\log n)$
3. $\Theta(n)$
4. $\Theta(n \log n)$
5. $\Theta(n^2)$

Symmetric Pairs

Two pairs (a, b) and (c, d) are symmetric if b = c and a = d. Suppose you want to find all symmetric pairs in the array. The first element of all pairs is distinct.

For example, if arr1[] = {(14, 23), (11, 2), (52, 83), (49, 38), (38, 49), (2, 11)};

The symmetric pairs are {(11, 2), (2, 11)} and {(49, 38), (38, 49)}.

What is the average-case time complexity of accomplishing this task if you use the most efficient algorithm? If hashing is involved, assume that both search and insert methods work in $\Theta(1)$ time.

1. $\Theta(1)$
2. $\Theta(\log n)$
3. **$\Theta(n)$**
4. $\Theta(n \log n)$
5. $\Theta(n2)$

Use a hashtable! The first element of each pair is the key, and the second element is the value.

Go through the array once. For each current pair, check if the current second element is in the hashtable, and if it is, check that its value is the same as the current first element. If it matches, it's a symmetric pair.

Interview Problems

Interview Problem

- Given a vector of integers, find the first non-repeated integer.
  - Example:
    - the first non-repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is 10.

Interview Problem

- Given a vector of integers, find the first non-repeated integer.
  - Example:
    - the first non-repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is 10.

- Use an `unordered_map`!
  - scan the vector from left to right and construct an `unordered_map<int,int>` to count the number of appearances for each number
  - after the `unordered_map` is completed, **scan the vector from left to right** and check the count for each character. If an element has a count of 1, return it.

- Time complexity: O(n), Space complexity: O(n)

Modifying the Problem

- Given a vector of integers, find the first **repeated** integer.
  - Example:
    - the first repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is -234.

- How does finding the first repeated (rather than non-repeated) integer change the problem?

Modifying the Problem

- Given a vector of integers, find the first **repeated** integer.
  - Example:
    - the first repeated integer in the vector [-234, 2, 1, -234, 10, 72, 1, 2] is -234.

- How does finding the first repeated (rather than non-repeated) integer change the problem? **We no longer care about the count! As long as the integer is in our hash table, we must have seen it before.**

- As a result, the “value” associated with each key doesn’t really matter to us. Is there an alternative container we could use?

Unordered_Sets

- An `unordered_set` stores unique elements in no particular order (i.e. keys are also values).
- Complexities for finding/inserting are average O(1) and worst-case O(n).

Unordered_Sets

- An unordered_set stores unique elements in no particular order (i.e. keys are also values).
- Complexities for finding/inserting are average O(1) and worst-case O(n).
- The solution to the previous problem:

```
#include <unordered_set>
void find_first_duplicate(const vector<int>& input) {
    unordered_set<int> s;
    for (auto number : input) {
      if (s.find(number) == s.end()) {
          s.insert(number);
       } else {
          cout << “First repeated number is: ” << number << endl;
       }
    }
}
```

once we find an element in the set, we must have seen it before, so we return it!

Hash Tables and Hash Functions

Hash Table and hash Functions

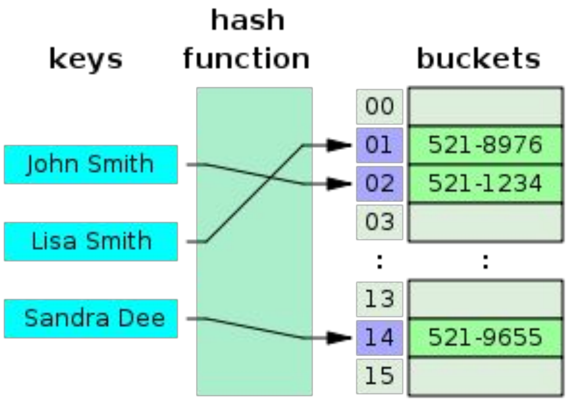
WE WANT YOU

Hash table **maps** keys to associated integer hash values

Aims for **average** case Θ(1) insert, lookup, delete.

You NEED to use the modulo operator (%) to keep hash values within the capacity of the hash table.

```
size_t bucket_index = hash_of(key) %
number_buckets;
```



TO USE %

Hash Tables - Exercise

number of buckets   N=6

Hash function:

```
int hash2(string s){
  if(s.empty())
      return 0;
    else
      return s[0]-'a';
}
```

What does hash function return?

“cat”
“cop”
“ear”
“cartographer”

Hash Tables - Exercise

number of buckets N=6

Hash function:

```
int hash2(string s){
  if(s.empty())
        return 0;
    else
        return s[0]-’a’;
}
```

What does hash function return?

“cat” - **hash:2**
“cop” - **hash:2**
“ear” - **hash:4**
“cartographer” - **hash:2**

Hash Tables - Exercise

number of buckets N=6

Hash function:

```
int hash2(string s){
  if(s.empty())
      return 0;
    else
      return s[0]-'a';
}
```

Which bucket would it be inserted at?

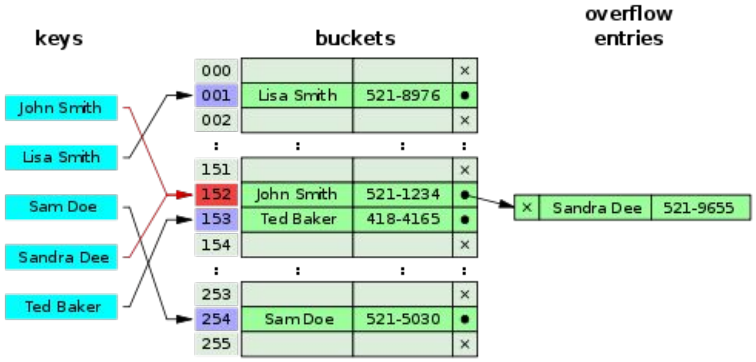
"Cat" - **hash:2** (%6) = 2
"Cop" - **hash:2** (%6) = 2
"Ear" - **hash:4** (%6) = 4
"Cartographer" - **hash:2** (%6) = 2

Collision!!

Collision Resolution

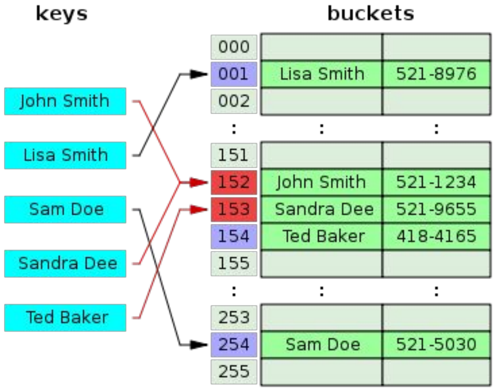
Separate Chaining:

- Store colliding key-value pairs in a linked list for that bucket



Open Addressing:

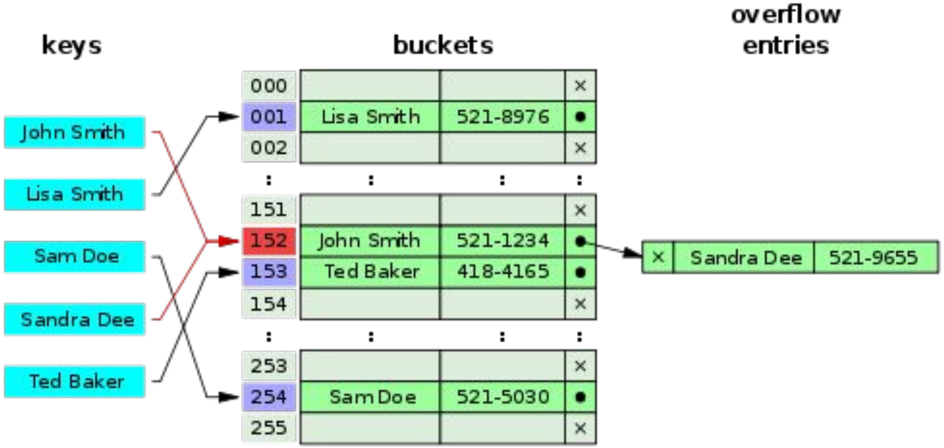
- Store colliding key-value pairs in another bucket/location



Why use Separate Chaining?

- Less sensitive to load factor (even > 1)
- Unknown number of keys/frequency of inserts

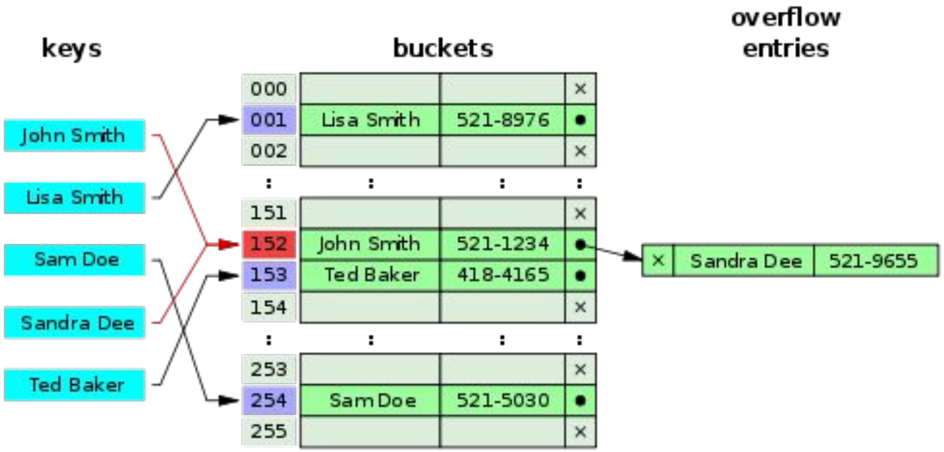
Load factor = Number of elements/size_of_buckets



Separate Chaining

What is the best case, average case, worst case of insert or look up?

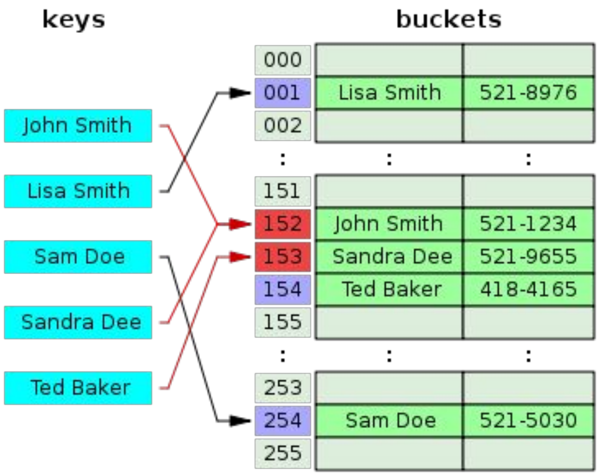
O(1), O(1), O(n)



Why use open addressing?

Much faster because load factor < 1

Provides better cache performance



Open Addressing

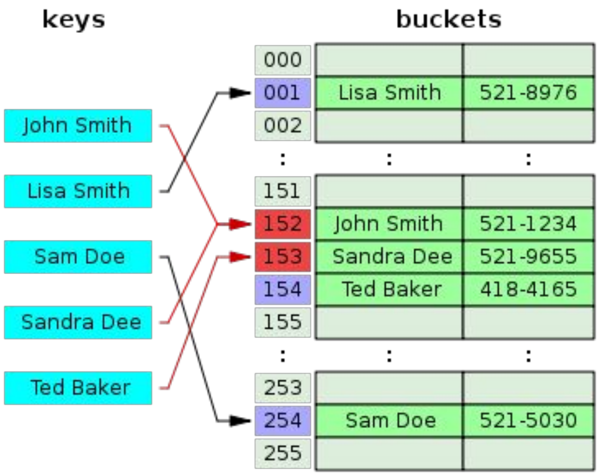
Linear Probing

Quadratic Probing

Double Hashing

Open Addressing:

- Store colliding key-value pairs in <u>another bucket/location</u>



Open Addressing Types

Let k be the key, H(k) be the hash value

Linear Probing $(H(k) + i) \% N$

Quadratic Probing $(H(k) + i^2) \% N$

Double Hashing $(H(k) + i * F(k)) \% N$

H(k) - hash function
F(k) - second hash function
i - collision number (starts at 0)
N - size of our hash table

Open Addressing Types

Let k be the key, H(k) be the hash value

Linear Probing $(H(k) + i) \% N$.... H(k), H(k) + 1, H(k) + 2, H(k) + 3...

Quadratic Probing $(H(k) + i^2) \% N$ .... H(k), H(k) + 1, H(k) + 4, H(k) + 9...

Double Hashing $(H(k) + i * F(k)) \% N$ .... H(k), H(k) + F(k), H(k) + 2F(k)...

H(k) - hash function
F(k) - second hash function
i - collision number (starts at 0)
N - size of our hash table

Linear Probing: Example

number of buckets N=6

Hash function:

```
int hash2(string s){
  if(s.empty())
      return 0;
    else
      return s[0]-'a';
}
```

2%6 = 2

0	1	2	3	4	5
		cat			

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

"cat" **hash: 2**
"cop" **hash: 2**
"ear" **hash: 4**
"cartographer" **hash: 2**

Linear Probing: Example

number of buckets   N=6

Hash function:

```
int hash2(string s){
  if(s.empty())
       return 0;
    else
       return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

"cat" **hash: 2**
"cop" **hash: 2**
"ear" **hash: 4**
"cartographer" **hash: 2**

(2 + 1)%6 = 3%6 = 3

0	1	2	3	4	5
		cat	cop		

Linear Probing: Example

number of buckets N=6

Hash function:

```
int hash2(string s){
  if(s.empty())
      return 0;
    else
      return s[0]-'a';
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

"cat" **hash: 2**
"cop" **hash: 2**
"ear" **hash: 4**
"cartographer" **hash: 2**

4%6 = 4

0	1	2	3	4	5
		cat	cop	ear	

Linear Probing: Example

number of buckets N=6

Hash function:

```
int hash2(string s){
  if(s.empty())
       return 0;
    else
       return s[0]-’a’;
}
```

If we insert these 4 items in this order, using **linear probing** to resolve collisions, where do the keys end up?

“cat” **hash: 2**
“cop” **hash: 2**
“ear” **hash: 4**
“cartographer” **hash: 2**

2

2 + 1 = 3

2 + 2 = 4

2 + 3 = 5

0	1	2	3	4	5
		cat	cop	ear	cartographer

Linear Probing: Example

number of buckets N=6

Hash function:

```
int hash2(string s){
  if(s.empty())
      return 0;
    else
      return s[0]-'a';
}
```

Your turn!

If we now insert “duck” into the hash table, where does it end up?

0	1	2	3	4	5
		cat	cop	ear	cartographer

Linear Probing: Example

number of buckets N=6

Hash function:

```
int hash2(string s){
  if(s.empty())
      return 0;
    else
      return s[0]-'a';
}
```

Your turn!

If we now insert "duck" into the hash table, where does it end up?

hash2("duck") = 3

(3 + 0) % 6 = 3 = FULL
(3 + 1) % 6 = 4 = FULL
(3 + 2) % 6 = 5 = FULL
(3 + 3) % 6 = 0 = EMPTY! Insert here

0	1	2	3	4	5
duck		cat	cop	ear	cartographer

Linear Probing Erasing Example (Ghost)

number of buckets N=10

Hash function:

```
int hash2(string s){
   if(s.empty())
        return 0;
     else
        return s[0]-'a';
}
```

How would we do these 3 operations in this order, using **linear probing** to resolve collisions?

erase “cat” hash: 2
Insert “cop” hash: 2
erase “cartographer” hash: 2
insert “cartographer" hash: 2

0	1	2	3	4	5	6	7	8	9
			cop	ear	cartographer				

Linear Probing Erasing Example

number of buckets N=10

Hash function:

```
int hash2(string s){
   if(s.empty())
         return 0;
     else
         return s[0]-'a';
}
```

How would we do these 3 operations in this order, using **linear probing** to resolve collisions?

erase "cat" hash: 2
erase "cartographer" hash: 2
insert "cartographer" hash: 2

2

0	1	2	3	4	5	6	7	8	9
		{ghost}	cop	ear	cartographer				

Linear Probing Erasing Example

number of buckets N=10

Hash function:

```
int hash2(string s){
   if(s.empty())
        return 0;
     else
        return s[0]-'a';
}
```

How would we do these 3 operations in this order, using **linear probing** to resolve collisions?

erase “cat” hash: 2
erase “cartographer” hash: 2
insert “cartographer" hash: 2

Must continue looking until an empty spot or the key is found!

2   2 + 1 = 3   2 + 2 = 4   2 + 3 = 5

0	1	2	3	4	5	6	7	8	9
		{ghost}	cop	ear	{ghost}				

Linear Probing Erasing Example

number of buckets N=10

Hash function:

```
int hash2(string s){
   if(s.empty())
        return 0;
     else
        return s[0]-’a’;
}
```

How would we do these 3 operations in this order, using **linear probing** to resolve collisions?

erase “cat” hash: 2
erase “cartographer” hash: 2
insert “cartographer" hash: 2

Should we insert “cartographer” here?

2

0	1	2	3	4	5	6	7	8	9
		{ghost}	cop	ear	{ghost}				

Linear Probing Erasing Example

number of buckets N=10

Hash function:

```
int hash2(string s){
   if(s.empty())
         return 0;
     else
         return s[0]-'a';
}
```

How would we do these 3 operations in this order, using **linear probing** to resolve collisions?

erase "cat" hash: 2

erase "cartographer" hash: 2

insert "cartographer" hash: 2

Should we insert "cartographer" here?

No!

2

0	1	2	3	4	5	6	7	8	9
		{ghost}	cop	ear	{ghost}				

Linear Probing Erasing Example

number of buckets N=10

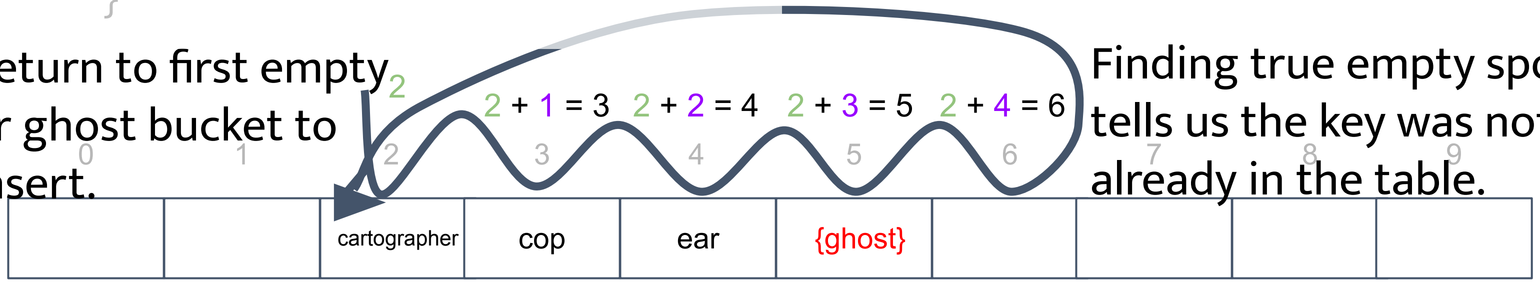
Hash function:

```
int hash2(string s){
   if(s.empty())
      return 0;
    else
      return s[0]-'a';
}
```

How would we do these 3 operations in this order, using **linear probing** to resolve collisions?

erase "cat" hash: 2
erase "cartographer" hash: 2
insert "cartographer" hash: 2

Return to first empty or ghost bucket to insert.



Finding true empty spot tells us the key was not already in the table.

Comparison of Collision Resolution Methods

Separate Chaining:

- Extra memory is in form of linked list ptrs
- Uses more dynamic memory
- Handling collisions is faster (appending to list is faster than math + reindexing)
- Can have a load factor greater than 1
- Linked lists used due to need for fast insert and delete in middle of list, and no need for random access
- Most common implementation

Open Addressing:

- Extra memory is in form of empty spaces
- Simpler storage
- Requires "ghosts" for erase
- Cache locality improves runtime
- Linear probing suffers from clusters
- Load factor must be < 1
  - Quadratic probing needs load factor < .5
- Double hashing wastes time on 2nd hash

Resizing Hash Tables

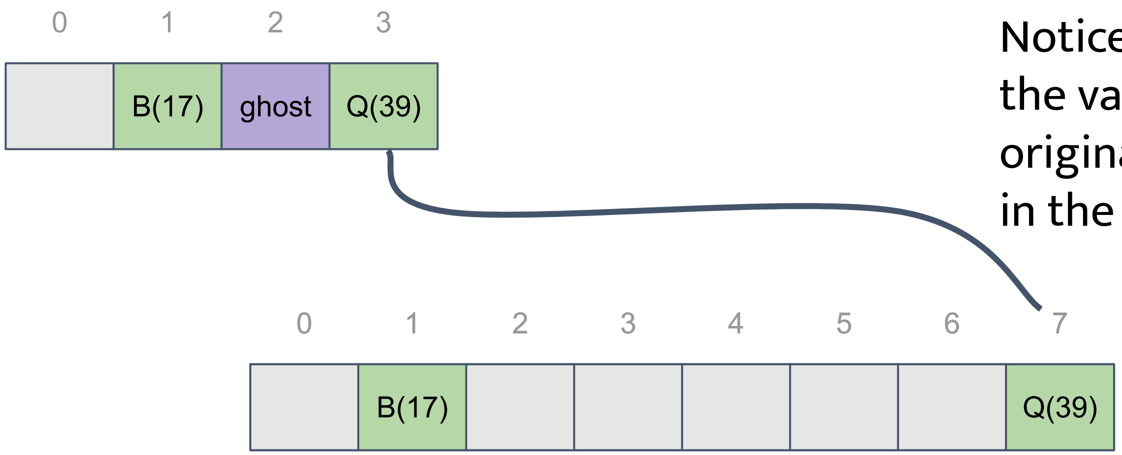
How to rehash:

Create a larger, empty hash table.

Insert all non-ghost elements (With same hash function)

Some keys will stop colliding in this new, larger table. Some new collisions may also occur.

Resizing Hash Tables



39 % 8 = 7

Notice, we did NOT just copy the value at index 3 in the original hash table to index 3 in the new hash table.

Exam style questions

Practice Problem 1

Which of the following statements is/are TRUE about a hash table that uses open addressing?

I. A hash function could map multiple keys to the same integer value
II. The average number of probes to find an element in the hash table may change when the load factor increases
III. Some keys may rehash to the same bucket when the size of a hash table's underlying array increases

A) I only

B) I and II only

C) I and III only

D I, II, and III

E) None of the above

Practice Problem 1

Which of the following statements is/are TRUE about a hash table that uses open addressing?

I. A hash function could map multiple keys to the same integer value
II. The average number of probes to find an element in the hash table may change when the load factor increases
III. Some keys may rehash to the same bucket when the size of a hash table's underlying array increases

A) I only

B) I and II only

C) I and III only

D) I, II, and III

E) None of the above

Practice Problem 2

What is the best definition of a hash collision?

A) Two entries are identical except for their keys.
B) Two entries with different data have the exact same key.
C) Two entries with different keys have the same exact hash value.
D) Two entries with the exact same key have different hash values.

Practice Problem 2

What is the best definition of a hash collision?

A) Two entries are identical except for their keys.
B) Two entries with different data have the exact same key.
C) Two entries with different keys have the same exact hash value.
D) Two entries with the exact same key have different hash values.

P

Consider the following hashing function and hash table. If the contents of the hash table were produced by inserting `Elk`, `Ant`, `It`, `Is`, `Elf`, and `Elm` in this order (with no other operations), which collision resolution method does the hash table use? *Hint: 'A' words hash to 0, 'E' words hash to 4, and 'I' words hash to 8.*

```
int hash_1(string s) {
  if (s.empty())
    return 0;
  else
    return s.front() - 'A';
} // hash_1()
```

Ant	Elm			Elk	Elf			It	Is
0	1	2	3	4	5	6	7	8	9

A) Linear probing
B) Quadratic probing
C) Cubic probing
D) Separate chaining
E) None of the above

Consider the following hashing function and hash table. If the contents of the hash table were produced by inserting `Elk`, `Ant`, `It`, `Is`, `Elf`, and `Elm` in this order (with no other operations), which collision resolution method does the hash table use? *Hint: 'A' words hash to 0, 'E' words hash to 4, and 'I' words hash to 8.*

```
int hash_1(string s) {
  if (s.empty())
    return 0;
  else
    return s.front() - 'A';
} // hash_1()
```

0	1	2	3	4	5	6	7	8	9
Ant	Elm			Elk	Elf			It	Is

A) Linear probing

B) Quadratic probing

C) Cubic probing

D) Separate chaining

E) None of the above

Practice Problem 4

In which of the following situations would you want to use a hash table?

A) You have a classroom full of students each with a 10-digit ID number, and you want to find them by their ID number.
B) You want to find the highest priority thread to execute, each with its own assigned priority.
C) You have a rectangular floor composed of square tiles, and you want to know the color of a tile given a set of coordinates.
D) You are given a set of names, and want to print them out in alphabetical order

Practice Problem 4

In which of the following situations would you want to use a hash table?

A) You have a classroom full of students each with a 10-digit ID number, and you want to find them by their ID number.
B) You want to find the highest priority thread to execute, each with its own assigned priority.
C) You have a rectangular floor composed of square tiles, and you want to know the color of a tile given a set of coordinates.
D) You are given a set of names, and want to print them out in alphabetical order

Hash Tables Practice Problem

- Suppose you're given a set of N distinct words, all of the same length M. We want to find a pair of words that are "similar", meaning that you can obtain one from the other by changing one letter. For example, "cart" and "cast" are similar, and "cast" and "cost" are similar (but "cart" and "cost" are not).
- We know that N is much, much bigger than M. You can also assume that $M \geq 3$. The algorithm may use any amount of memory. (Hint: Use a hash table.)

Hash Tables Practice Problem

- Go through each word and replace each letter with a “blank” letter (*). Search for this “blankified” word in the hash table. If you find it, that’s your pair of similar words. Otherwise, insert it and keep going!
- For each i = 0, 1, ..., M-1, for each string s, set s[i] = ‘*’ and put it into the hash set
- What’s the best possible worst-case running time of this algorithm to find a pair of similar words in the provided set, in terms of length M and number of distinct words N? (Note that it takes O(M) look up a string of length M in a hash table because it must be hashed and compared for equality).

Hash Tables Practice Problem

- Go through each word and replace each letter with a "blank" letter (*). Search for this "blankified" word in the hash table. If you find it, that's your pair of similar words. Otherwise, insert it and keep going!
- For each i = 0, 1, ..., M-1, for each string s, set s[i] = '*' and put it into the hash set
- What's the best possible worst-case running time of this algorithm to find a pair of similar words in the provided set, in terms of length M and number of distinct words N? (Note that it takes O(M) look up a string of length M in a hash table because it must be hashed and compared for equality). **$\Theta(M^2N)$**
- **For every letter (M) in every word (N), you must do an O(M) look up.**

Feedback

- Please leave comments/suggestions on piazza post @3807

Handwritten Problem

Handwritten Problem

Prefixes are words that can be followed by some other letters to form a longer word - let's call this final word the successor. For example, the prefix “an” followed by “other” forms the word “another”.

Now, given a dictionary consisting of many prefixes and a sentence, you need to replace all the successors in the sentence with the prefix forming it. If a successor has many prefixes that can form it, replace it with the prefix with the shortest length.

The input will only have lower-case letters. Return the new sentence in a vector of strings.
P prefixes**, N** words**, M** length**: O(PM + NM^2) (Hashing/looking up a string of length M costs O(M))**

Example:
Prefixes: ["cat", "bat", "rat"]
Sentence: [“the”, “cattle”, “was”, “rattled”, “by”, “the”, “battery”]
Output: ["the”, “cat”, “was”, “rat”, “by”, “the”, “bat"]

```
vector<string> replace_words(const vector<string>& prefixes,
                             const vector<string>& sentence);
```