NUS
National University of Singapore

School of Computing

Tutorial 4: Hashing

September 12, 2022

Gu Zhenhao

** Partly adopted from tutorial slides by Wang Zhi Jian.*

Map ADT

Why do we need the Map ADT?

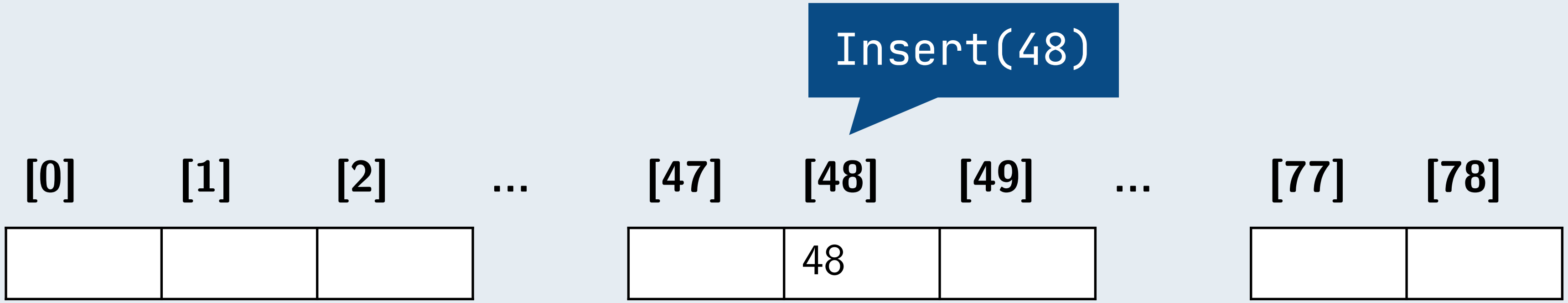
Why Map?

Operations	Array	Linked List
`getItemAtIndex`	$O(1)$	$O(n)$
`getFirst/getLast`	$O(1)$	$O(1)^*$
`addAtIndex/removeAtIndex`	$O(n)$	$O(n)$
`addFront/removeFront`	$O(n)$	$O(1)$
`addBack/removeBack`	$O(n)$ ($O(1)$ amortized)	$O(1)^*$
`contains`	$O(n)$	$O(n)$

- Searching for a key in arrays and linked lists is slow.
- The index of a key is unknown, so we have to search one by one.

How to map?

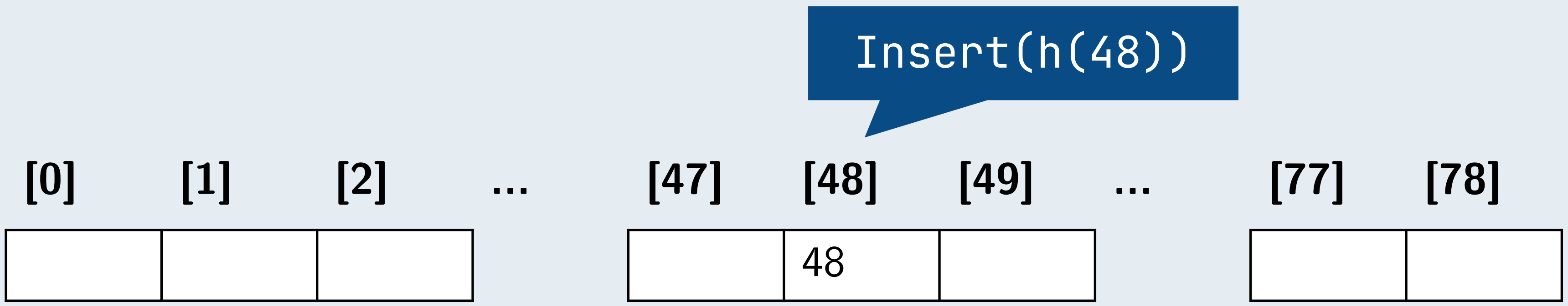
- **Purpose**: we want to infer the index from the key directly.
- **Trivial answer**: directly use the key as index?



- **Problems**:
  1. What if the key inserted is very large?
  2. What if the key is not an integer?

How to map?

- **Purpose**: we want to infer the index from the key directly.
- **Idea**: Use a function h to map a key to a slot.



- **Problems**:
  1. How to define a good function h?
  2. What if h maps multiple keys to the same slot?

Collision Resolution

- **Separate Chaining**: keep the collided keys in the same slot using *linked list*.

Insert(48)

$h(i) = i\%10$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	11				45 ↓ 35		27	28 ↓ **48**	39

- **Pros**: Inserting a key always cost $O(1)$ time.
- **Cons**: Need $O(n)$ extra space; searching for a key may be slow.

Collision Resolution

- **Linear Probing**: jump to the next slots until we find an empty slot. $(+1, +2, +3, \ldots)$

`Insert(48)`

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	11	**48**			45	35	27	28	39

$h(i) = i\%10$

- **Pros**: Can always find a slot for a key as long as hash table is not full.
- **Cons**: may form primary clusters (consecutive filled slots); both inserting and searching may be slow when large clusters form.

Collision Resolution

- **Quadratic Probing**: gradually increase the length of jumping. $(+1^2, +2^2, +3^2, \ldots)$

`Insert(48)`

$h(i) = i\%10$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	11	**48**			45	35	27	28	39

- **Pros**: can jump through a cluster faster.
- **Cons**: May be unable to find a free slot. (guarantee to find one if load factor < 0.5); May form secondary clusters (same hash value, same probe sequence).

Collision Resolution

- **Double Hashing**: different jumping distance for different keys $(+g(i), +2g(i), +3g(i)\ \ldots)$.

`Insert(48)`

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
0	11			**48**	45	35	27	28	39

$h(i) = i\%10$

$g(i) = (9 - i)\%9$

- **Pros**: can jump through a cluster faster, harder to form a cluster.
- **Cons**: May be stuck in one place or unable to find a slot.

Problem 1: Linear Probing

- Use *linear probing*, hash function $h(\text{key}) = \text{key}\%5$.

[0]	[1]	[2]	[3]	[4]

- **Step 1**: `insert(7)`

We have $7\%5 = 2$, so insert at slot 2.

Problem 1: Linear Probing

- Use *linear probing*, hash function $h(\text{key}) = \text{key}\%5$.

[0]	[1]	[2]	[3]	[4]
		7		

- **Step 2**: `insert(12)`

We have $12\%5 = 2$, and we have a collision!

Check $(12 + 1)\%5 = 3$, an empty slot, so insert at slot 3.

Problem 1: Linear Probing

- Use *linear probing*, hash function $h(\text{key}) = \text{key}\%5$.

[0]	[1]	[2]	[3]	[4]
		7	12	

- **Step 3**: `insert(22)`

We have $22\%5 = 2$, and we have a collision!

Check $(22 + 1)\%5 = 3$, again a collision!

Check $(22 + 2)\%5 = 4$, an empty slot, so we insert at slot 4.

Problem 1: Linear Probing

- Use *linear probing*, hash function $h(\text{key}) = \text{key}\%5$.

[0]	[1]	[2]	[3]	[4]
		7	12	22

- **Step 4**: `delete(12)`

Can we simply set slot 3 as empty slot?

No! In this case we will not be able to find 22.

We mark this slot as deleted instead.

Problem 1: Linear Probing

- Use *linear probing*, hash function $h(\mathrm{key}) = \mathrm{key}\%5$.

[0]	[1]	[2]	[3]	[4]
		7	8	22

- **Step 5**: `insert(8)`

We have $8\%5 = 3$, and we see a **del** symbol!

We can simply insert into slot 3.

Problem 1: Quadratic Probing

- Use *quadratic probing*, hash function $h(\text{key}) = \text{key}\%5$.

[0]	[1]	[2]	[3]	[4]

- **Step 1**: `insert(7)`

We have $7\%5 = 2$, so insert at slot 2.

Problem 1: Quadratic Probing

- Use *quadratic probing*, hash function $h(\text{key}) = \text{key}\%5$.

[0]	[1]	[2]	[3]	[4]
		7		

- **Step 2**: `insert(12)`

We have $12\%5 = 2$, a collision!

Check $(12 + 1^2)\%5 = 3$, so we insert at slot 3.

Problem 1: Quadratic Probing

- Use *quadratic probing*, hash function $h(\text{key}) = \text{key}\%5$.

[0]	[1]	[2]	[3]	[4]
		7	12	

- **Step 3**: `insert(22)`

We have $22\%5 = 2$, a collision!

Check $(22 + 1^2)\%5 = 3$, again a collision!

Check $(22 + 2^2)\%5 = 1$, so we insert at slot 1.

Problem 1: Quadratic Probing

- Use *quadratic probing*, hash function $h(\text{key}) = \text{key}\%5$.

[0]	[1]	[2]	[3]	[4]
	22	7	12	

- **Step 4**: `insert(2)`

We have $2\%5 = 2$, a collision!

Check $(2 + 1^2)\%5 = 3$, again a collision!

Check $(2 + 2^2)\%5 = 1$, again a collision!

Problem 1: Quadratic Probing

- Use *quadratic probing*, hash function $h(\text{key}) = \text{key}\%5$.

[0]	[1]	[2]	[3]	[4]
	22	7	12	

- **Step 4**: `insert(2)`

$(2+3^2)\%5 = 1$, $(2+4^2)\%5 = 3$, $(2+5^2)\%5 = 2$, $(2+6^2)\%5 = 3\ldots$

Shall we go on forever?

Patterns in Quadratic Probing

$$
\begin{aligned}
(2+0^2)\%5 &= \boxed{2} \\
(2+1^2)\%5 &= \boxed{3} \\
(2+2^2)\%5 &= \boxed{1} \\
(2+3^2)\%5 &= \boxed{1} \\
(2+4^2)\%5 &= \boxed{3} \\
(2+5^2)\%5 &= \boxed{2} \\
(2+6^2)\%5 &= \boxed{3} \\
(2+7^2)\%5 &= \boxed{1} \\
(2+8^2)\%5 &= \boxed{1} \\
(2+9^2)\%5 &= \boxed{3} \\
(2+10^2)\%5 &= 2
\end{aligned}
$$

Do you notice any pattern?

It seems that the pattern "23113" keeps repeating.

Idea: probably for the values of hash function $h_k(i) = (i + k^2)\%m$ repeats for each m functions?

$$(i + \boldsymbol{k}^2)\%m = (i + (\boldsymbol{k} + \boldsymbol{m})^2)\%m$$

Patterns in Quadratic Probing

$$
\begin{aligned}
(2+0^2)\%5 &= \boxed{2} \\
(2+1^2)\%5 &= \boxed{3} \\
(2+2^2)\%5 &= \boxed{1} \\
(2+3^2)\%5 &= \boxed{1} \\
(2+4^2)\%5 &= \boxed{3} \\
(2+5^2)\%5 &= \boxed{2} \\
(2+6^2)\%5 &= \boxed{3} \\
(2+7^2)\%5 &= \boxed{1} \\
(2+8^2)\%5 &= \boxed{1} \\
(2+9^2)\%5 &= \boxed{3} \\
(2+10^2)\%5 &= 2
\end{aligned}
$$

$$
\begin{aligned}
&(i+(k+m)^2)\%m \\
&= (i+k^2+2km+m^2)\%m \\
&= \left(i+k^2+\underbrace{2km\%m}_{=0}+\underbrace{m^2\%m}_{=0}\right)\%m \\
&= (i+k^2)\%m
\end{aligned}
$$

Indeed! Therefore we only need to evaluate the first m hash functions.

Patterns in Quadratic Probing

$$(2+0^2)\%5 = \boxed{2}$$
$$(2+1^2)\%5 = \boxed{3}$$
$$(2+2^2)\%5 = \boxed{1}$$
$$(2+3^2)\%5 = \boxed{1}$$
$$(2+4^2)\%5 = \boxed{3}$$
$$(2+5^2)\%5 = \boxed{2}$$
$$(2+6^2)\%5 = 3$$
$$(2+7^2)\%5 = 1$$
$$(2+8^2)\%5 = 1$$
$$(2+9^2)\%5 = 3$$
$$(2+10^2)\%5 = 2$$

Do you notice any other pattern?

It seems that pattern from 3 to 5 is just the pattern from 0 to 2 reversed.

Idea: probably for the values of hash function $h_k(i) = (i + k^2)\%m$ are symmetric w.r.t. $k = m/2$?

$$(i + \boldsymbol{k}^2)\%m = (i + (\boldsymbol{m} - \boldsymbol{k})^2)\%m$$

Patterns in Quadratic Probing

$$
\begin{aligned}
(2+0^2)\%5 &= \boxed{2} \\
(2+1^2)\%5 &= \boxed{3} \\
(2+2^2)\%5 &= \boxed{1} \\
(2+3^2)\%5 &= \boxed{1} \\
(2+4^2)\%5 &= \boxed{3} \\
(2+5^2)\%5 &= \boxed{2} \\
(2+6^2)\%5 &= 3 \\
(2+7^2)\%5 &= 1 \\
(2+8^2)\%5 &= 1 \\
(2+9^2)\%5 &= 3 \\
(2+10^2)\%5 &= 2
\end{aligned}
$$

$$
\begin{aligned}
&(i+(m-k)^2)\%m \\
&= (i+m^2-2km+k^2)\%m \\
&= \left(i+\underbrace{m^2\%m}_{=0}-\underbrace{2km\%m}_{=0}+k^2\right)\%m \\
&= (i+k^2)\%m
\end{aligned}
$$

Indeed! Therefore we only need to investigate the first $\lceil m/2 \rceil$ hash functions.

Problem 1: Double Hashing I

- Use *double hashing*, hash functions $h(\text{key}) = \text{key}\%5$, $g(\text{key}) = \text{key}\%3$.

[0]	[1]	[2]	[3]	[4]

- **Step 1**: `insert(7)`

We have $7\%5 = 2$, so insert at slot 2.

Problem 1: Double Hashing I

- Use *double hashing*, hash functions $h(\mathrm{key}) = \mathrm{key}\%5$, $g(\mathrm{key}) = \mathrm{key}\%3$.

[0]	[1]	[2]	[3]	[4]
		7		

- **Step 2**: `insert(22)`

We have $22\%5 = 2$, a collision!

Check $\big(22 + g(22)\big)\%5 = (22+1)\%5 = 3$, so we insert in slot 3.

Problem 1: Double Hashing I

- Use *double hashing*, hash functions $h(\text{key}) = \text{key}\%5$, $g(\text{key}) = \text{key}\%3$.

[0]	[1]	[2]	[3]	[4]
		7	22	

Better set the second hash function so that **it doesn't evaluate to 0**!

- **Step 3**: `insert(12)`

We have $12\%5 = 2$, a collision!

Check $\big(12 + g(12)\big)\%5 = (12+0)\%5 = 2$, still a collision!

This goes on infinitely as $g(12) = 0$.

Problem 1: Double Hashing II

- Use *double hashing*, hash functions $h(\text{key}) = \text{key}\%5$, $g(\text{key}) = 7 - (\text{key}\%7)$.

[0]	[1]	[2]	[3]	[4]

- **Step 1**: `insert(7)`

We have $7\%5 = 2$, so insert at slot 2.

Problem 1: Double Hashing II

- Use *double hashing*, hash functions $h(\text{key}) = \text{key}\%5$, $g(\text{key}) = 7 - (\text{key}\%7)$.

[0]	[1]	[2]	[3]	[4]
		7		

- **Step 2**: `insert(12)`

We have $12\%5 = 2$, a collision!

Check $\big(12 + g(12)\big)\%5 = (12 + 2)\%5 = 4$, so we insert at slot 4.

Problem 1: Double Hashing II

- Use *double hashing*, hash functions $h(\text{key}) = \text{key}\%5$, $g(\text{key}) = 7 - (\text{key}\%7)$.

[0]	[1]	[2]	[3]	[4]
		7		12

- **Step 3**: `insert(22)`

We have $22\%5 = 2$, a collision!

Check $\big(22 + g(22)\big)\%5 = (22 + 1)\%5 = 3$, so we insert at slot 3.

Problem 1: Double Hashing II

- Use *double hashing*, hash functions $h(\text{key}) = \text{key}\%5$, $g(\text{key}) = 7 - (\text{key}\%7)$.

[0]	[1]	[2]	[3]	[4]
		7	22	12

Better set the second hash function so that **it doesn't evaluate to multiples of m**!

- **Step 4**: `insert(2)`

We have $2\%5 = 2$, a collision!

Check $\big(2 + g(2)\big)\%5 = (2+5)\%5 = 2$, still a collision!

This goes on infinitely as $g(2)\%5 = 0$.

Hash Function

What makes a good hash function?

What makes a hash function good?

1. **Deterministic.**
   Same key always maps to the same slot.
2. **Fast.**
   Time should not depend on size of hash table/total items. Usually $O(1)$ or depends on size of key.
3. **Uniformly distributed.**
   Key should be distributed to *all slots* with equal probability, even if they *share some simple characteristics*.

Problem 2.a

Good or bad: *The hash table has size 100 with positive even integer keys. The hash function is* $h(\mathrm{key}) = \mathrm{key} \,\%\, 100$.

Deterministic? Yes!

Fast? Yes!

Uniformly distributed?

No! our keys are positive even integers, odd numbered slots will never be used!

Problem 2.b

Good or bad: *The hash table has size 49 with positive integer keys. The hash function is* $h(\text{key}) = (\text{key} * 7) \ \% \ 49$.

Deterministic? Yes!

Fast? Yes!

Uniformly distributed?

No! We can only map to slot 0, 7, 14, 21, 28, 35, 42!

Problem 2.c

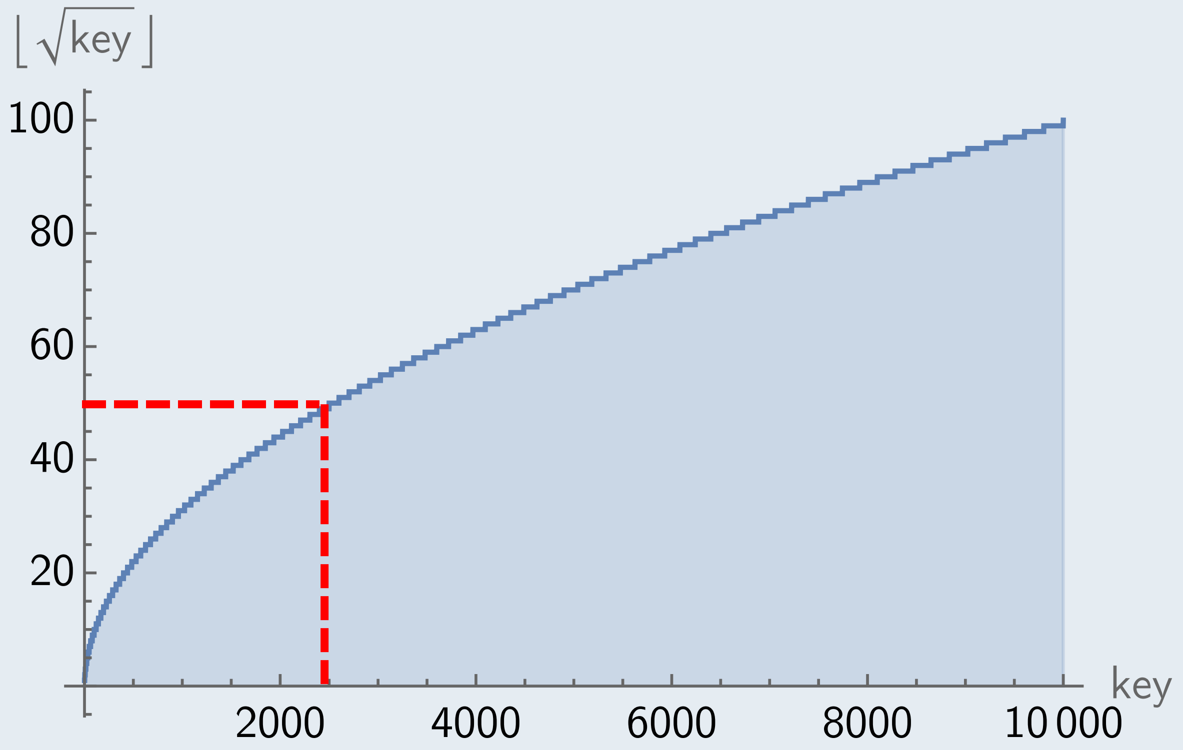
Good or bad: *The hash table has size 100 with non-negative integer keys in the range [0, 10000]. The hash function is* $h(\text{key}) = \lfloor\sqrt{\text{key}}\rfloor \,\%\, 100$.

Deterministic? Yes!

Fast? Yes!

Uniformly distributed?

No! We are more likely to map to higher numbered slots.



Problem 2.d

Good or bad: *The hash table has size 1009, and keys are valid email addresses. The hash function is* $h(\text{key}) = (\text{sum of ASCII values of each of the last 10 characters}) \ \% \ 1009.$

Deterministic? Yes!

Fast? Yes!

Uniformly distributed?

No! All addresses with same long domain name, e.g. @comp.nus.edu.sg are mapped to the same slot!

Problem 2.e

Good or bad: *The hash table has size 101 with integer keys in the range of [0, 1000]. The hash function is* $h(\text{key}) = \lfloor \text{key} \times \text{random} \rfloor \% \, 101$, *where* $0.0 \leq \text{random} \leq 1.0$.

Deterministic? No! We generate a random number each time... so multiple evaluation of same $h(\text{key})$ will give different result!

Problem 2.f

Good or bad: *The hash table has size 54 with String keys, with the hash function:*

```
int hash(String key) {
    h = 0
    for (int i = 0; i <= key.length() - 1; i++)
        h += 9 * (int) key.charAt(i)
    h = (h mod 54)
    return h
}
```

Deterministic? Yes!

Fast? Yes!

Uniformly distributed? No! h will be multiples of 9, so h can only be among 0, 9, 18, 27, 36, 45.

How to set a good hash function?

1. **Deterministic.**
   Never use random numbers in hash function.
2. **Fast.**
   Infer slot index only from the key itself.
3. **Uniformly distributed.**
   Use prime numbers in hash functions to ensure even distribution!

How to set good hash function(s)? *

Many standard ways to set hash functions... e.g.

1. Tabulation Hashing.
2. Binary Matrix Technique.
3. Prime Field: choose a prime number $p > m$, two random integers $1 \le a \le p-1$, $0 \le b \le p-1$, and define $h(x) = \big((ax+b) \bmod p\big) \bmod m$.

A Way to choose a good prime number (for hash table size):

1. Table Lookup. (a table used by standard C++ library)

Application of Map

How to use the fast searching of Map ADT properly?

Problem 3

- **Goal**: Find the time each k-letter words appear in the text.
- **Trivial Answer**: for the given k-letter word, traverse through the text and count it appearance. Each query takes $O(nk)$ time.

`mississippi`

The 4-letter word `issi` appear twice.

Redundant work: no need to go through the text again and again if we store the count!

** This is a classic problem in Computational Biology: k-mer counting in Genome.*

Problem 3

- **Goal**: Find the time each k-letter words appear in the text.
- **Idea**: pre-process the text and store all the counts.

`mississippi`

key	value
miss	1
issi	2
ssis	1
siss	1
ssip	1
sipp	1
ippi	1

Problem 3

Idea: Store ⟨key, value⟩ pair in a hash table.

1. **Pre-processing**: for each of the $(n - k + 1)$ k-letter words,
   - If it exists in hash table, increment the value.
   - Otherwise set the value as 1.
2. **Query**: for the given word, search if it is in the table. If yes, return the value.

key	value
`miss`	1
`issi`	2
`ssis`	1
`siss`	1
`ssip`	1
`sipp`	1
`ippi`	1

* *This technique is commonly used in Computational Biology, for [k-mer] counting algorithms.*

Problem 3

Each search takes in average $O(k)$ time and in worst case $O(nk)$ time (when can it happen?).

1. **Pre-processing**: $O(nk)$ time in average.
2. **Query**: $O(k)$ time in average.

key	**value**
`miss`	1
`issi`	2
`ssis`	1
`siss`	1
`ssip`	1
`sipp`	1
`ippi`	1

* *This technique is commonly used in Computational Biology, for k-mer counting algorithms.*

Problem 4

- **Goal**: choose one item from each category such that they sum to \$100.

Appetizers	**A** \$40	**B** \$20	**C** \$50
Soups	**D** \$10	**E** \$60	
Mains	**F** \$30	**H** \$70	
Desserts	**I** \$10	**J** \$20	**K** \$30

Problem 4

- **Simplified Goal**: choose one item from each of the 2 categories such that they sum to \$100.

Appetizers	**A** \$40	**B** \$20	**C** \$50
Soups	**D** \$10	**E** \$60	

Problem 4

Appetizers	**A** \$40	**B** \$20	**C** \$50
Soups	**D** \$10	**E** \$60	

Trivial answer: Try out all $O(n^2)$ combinations.

A D	A E	B D	B E	C D	C E
\$50	\$100	\$30	\$80	\$60	\$110

Problem 4

Appetizers	**A** \$40	**B** \$20	**C** \$50
Soups	**D** \$10	**E** \$60	

Trivial answer: Try out all $O(n^2)$ combinations.

A D	A E	B D	B E	C D	C E
\$50	\$100	\$30	\$80	\$60	\$110

No need to consider B or C as there's no \$80 or \$50 soup.

Problem 4

Appetizers	A \$40	B \$20	C \$50
Soups	D \$10	E \$60	

Checking still costs $O(n)$!

Idea:

- For each $\$k$ appetizer **check** whether there is a $\$100 - k$ soup.
- Use price as keys, put $\langle 10, D\rangle$, $\langle 60, E\rangle$ into a **Hash Map** to make checking $O(1)$! In total we need just $O(n)$ time.

Problem 4

What if we have 4 categories? Can we transfer them into 2 categories?

Appetizers	**A** \$40	**B** \$20	**C** \$50
Soups	**D** \$10	**E** \$60	
Mains	**F** \$30	**H** \$70	
Desserts	**I** \$10	**J** \$20	**K** \$30

Problem 4

Idea: Merge each two categories together and use the original algorithm!

Both merging and searching cost $O(n^2)$ time.

Appetizer + Soup Set	**AD** \$50	**BD** \$30	**CD** \$60
	AE \$100	**BE** \$80	**CE** \$110
Main + Dessert Set	**FI** \$40	**FJ** \$50	**FK** \$60
	HI \$80	**HJ** \$20	**HK** \$100

Bloom Filter

How does it work?

Bloom Filter

- **Purpose**: save space while allowing us to check if a key was inserted or not.
- **Idea**:
  1. Maintain a sequence of bits.
  2. Use multiple hash functions, e.g. h_1, h_2, h_3.

```
000000000000000000000000000000000000
```

Bloom Filter

- **Purpose**: save space while allowing us to check if a key was inserted or not.
- **Idea**:
  1. Maintain a sequence of bits.
  2. Use multiple hash functions, e.g. h_1, h_2, h_3.
  3. When inserting a key x, set bits at $h_1(x), h_2(x), h_3(x)$ to 1.

`Insert(48)`

```
000001000000000001001000000000000000
```

$h_1(48)$ $h_2(48)$ $h_3(48)$

Bloom Filter

- **Purpose**: save space while allowing us to check if a key was inserted or not.
- **Idea**:
  1. Maintain a sequence of bits.
  2. Use multiple hash functions, e.g. h_1, h_2, h_3.
  3. When inserting a key x, set bits at $h_1(x), h_2(x), h_3(x)$ to 1.

`Insert(32)`

`000001010000000001001000000010000000`

$h_2(32)$ $h_3(32)$ $h_1(32)$

Already set!

Bloom Filter

- **Purpose**: save space while allowing us to check if a key was inserted or not.
- **Idea**:
  1. Maintain a sequence of bits.
  2. Use multiple hash functions, e.g. h_1, h_2, h_3.
  3. When inserting a key x, set bits at $h_1(x), h_2(x), h_3(x)$ to 1.
  4. When checking if a key x is inserted, check if $h_1(x), h_2(x), h_3(x)$ are <u>**all**</u> 1.

```
000001010000000001001000000010000000
```

$h_1(48)$ $h_2(48)$ $h_3(48)$

48 is present!

Problem 5

If we enable remove/delete:

`delete(48)`

`000001010000000001001000000010000000`

$h_1(48)$ $h_2(48)$ $h_3(48)$

Problem 5

If we enable remove/delete:

- We can remove some or all of those set bits.
- Check if 48 is in the bloom filter? No, which is correct!

`delete(48)`

`0000000100000000000000000000000010000000`

$h_1(48)$ $h_2(48)$ $h_3(48)$

Problem 5

If we enable remove/delete:

- We can remove some or all of those set bits.
- Check if 48 is in the bloom filter? No, which is correct!
- Check if 32 is in the bloom filter? No, $h_3(32)$ is not set.
- We will have **false negative** results!

`query(32)`

```
000000010000000000000000000010000000
```

$h_2(32)$ $h_3(32)$ $h_1(32)$

Appendix

Tips on Midterm Exam

In general:

- Skip problems you are unsure of, finish easiest ones first.
- If you don't understand a question:
  - ~~**Step 1**: Cry.~~
  - ~~**Step 2**: Hand in the paper to us.~~
  - ~~**Step 3**: Cry louder outside.~~

Ask us to clarify!

Analysis Questions

Some statements may be stating some general rules, in that case...

- Unless you can easily see that a statement is true, first try to prove that it is false by **finding a counter-example**!
  - e.g. try some boundary cases & special cases.
- If you can't find such counter-examples, then consider why it might be true.

Structured Questions

- If you do not modify the data structures/algorithms in the lecture, you can simply say the names of them without implementing them again.
    - e.g. Write “we use Merge Sort on array A” instead of writing out the whole Merge Sort algorithm.
- Define your variables clearly.
    - e.g. write “Let S be a stack of integers” before using S.
- Pseudocode is OK unless Java is explicitly required.
- Be clear!

Structured Questions

- If you cannot come up with a solution that meets the time complexity requirement. Just write down the best solution you can think of.
- You can start from some **trivial answers**, and see whether you can improve them.
- A slow but correct solution is always better than a fast but wrong solution.

End of File

Enjoy(?) the recess week and good luck with the exam :-)