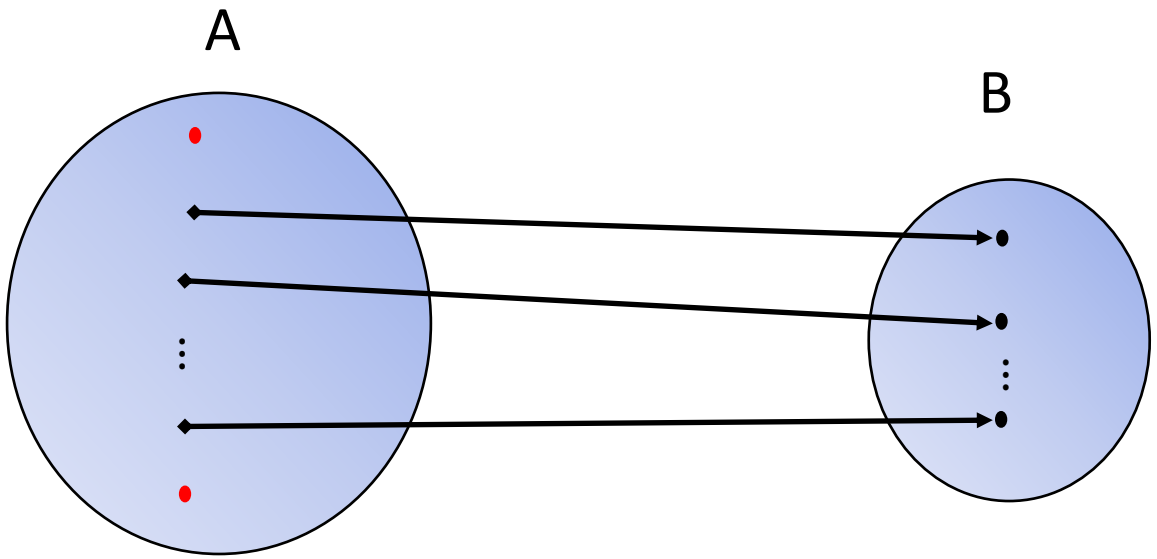
Collision – Resolution strategies for hashing

CMSC 420

Formal problem definition

- Let A and B be two **finite** sets with $|A| > |B|$. Then, there exists no injection $f: A \mapsto B$.



In hashing…

- In our problem, set A is our dataset and B is our available memory.
- $|A| >> |B|$ (one of the major computational problems of modern CS)

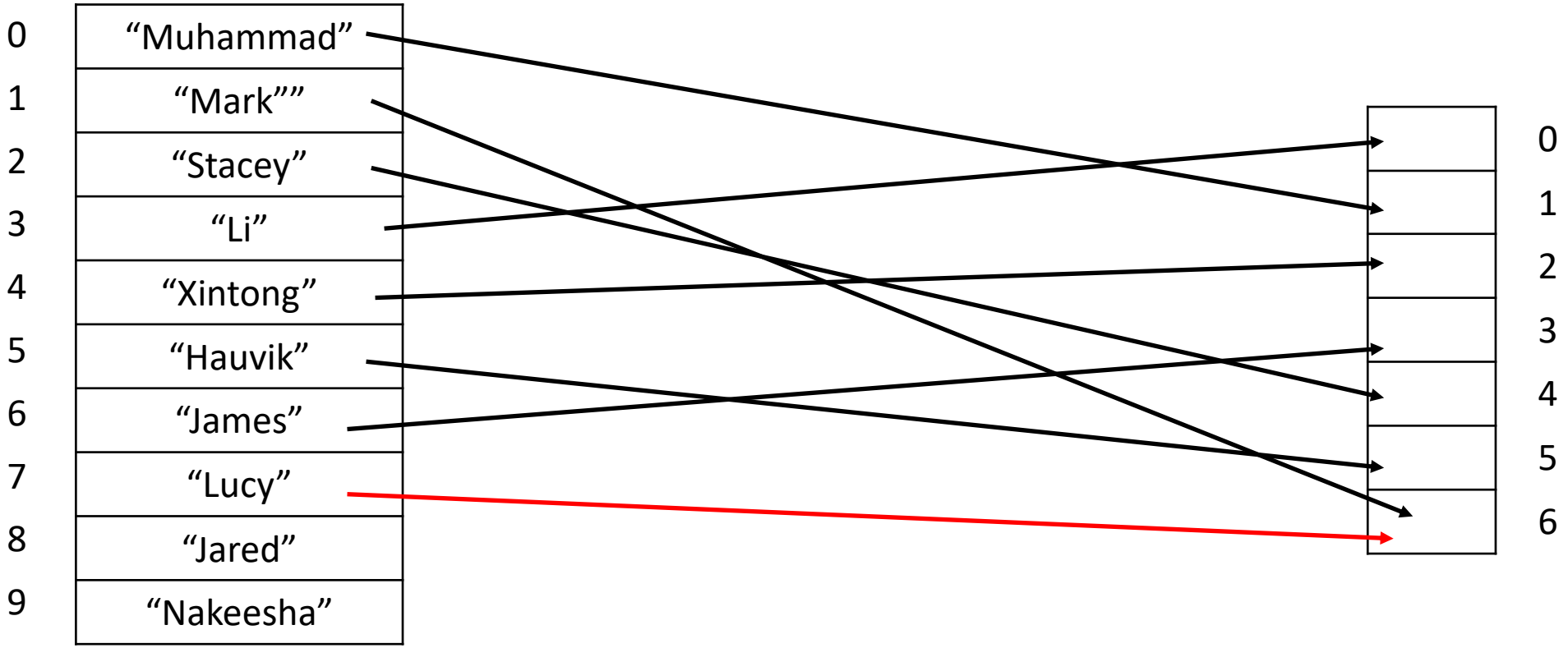
In hashing…

- In our problem, set A is our dataset and B is our available memory.
- $|A| >> |B|$ (one of the major computational problems of modern CS)
- Silver lining: many times we don't need the ***entirety*** of the data stored in a ***single*** machine…

In hashing…

- In our problem, set A is our dataset and B is our available memory.
- $|A| >> |B|$ (one of the major computational problems of modern CS)
- Silver lining: many times we don't need the ***entirety*** of the data stored in a ***single*** machine…
- So we're essentially recognizing the problem, but we will deal with it when it comes (*lazy approach*)
- The absence of a 1-1 mapping between A and B implies that **EVEN WITH THE BEST HASH FUNCTION POSSIBLE, COLLISIONS ARE UNAVOIDABLE.**

Visually


0
1
2
3
4
5
6
7
8
9
“Muhammad”
“Mark””
“Stacey”
“Li”
“Xintong”
“Hauvik”
“James”
“Lucy”
“Jared”
“Nakeesha”
0
1
2
3
4
5
6

Quiz

- Let n be the size of our data (in records) and M be our hash table size, with $n \gg M$.

Quiz

- Let n be the size of our data (in records) and M be our hash table size, with $n \gg M$.
- True or False: our first collision will happen at the $(M + 1)$th insertion (this would mean that we'd need to resize our table if we want more keys to fit)

Quiz

- Let n be the size of our data (in records) and M be our hash table size, with $n \gg M$.
- True or False: our first collision will happen at the $(M + 1)$th insertion (this would mean that we'd need to resize our table if we want more keys to fit)

True

False

Quiz

- Let n be the size of our data (in records) and M be our hash table size, with $n \gg M$.
- True or False: our first collision will happen at the $(M+1)$th insertion (this would mean that we'd need to resize our table if we want more keys to fit)

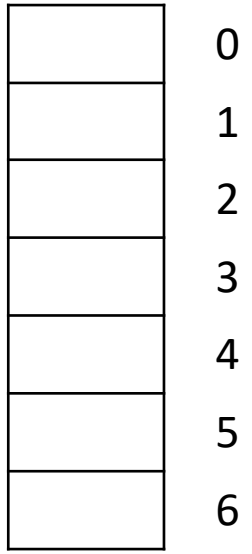
True

False

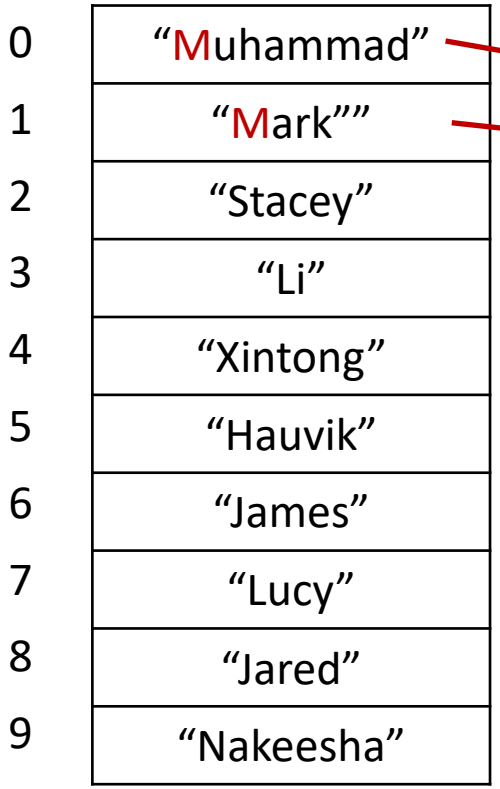
- This would assume a perfect hash function available for any data type!

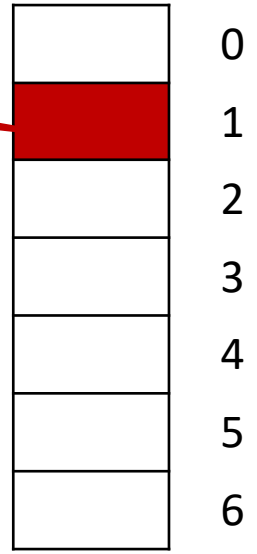
A terrible hash function for strings

0	“Muhammad”
1	“Mark””
2	“Stacey”
3	“Li”
4	“Xintong”
5	“Hauvik”
6	“James”
7	“Lucy”
8	“Jared”
9	“Nakeesha”



A terrible hash function for strings



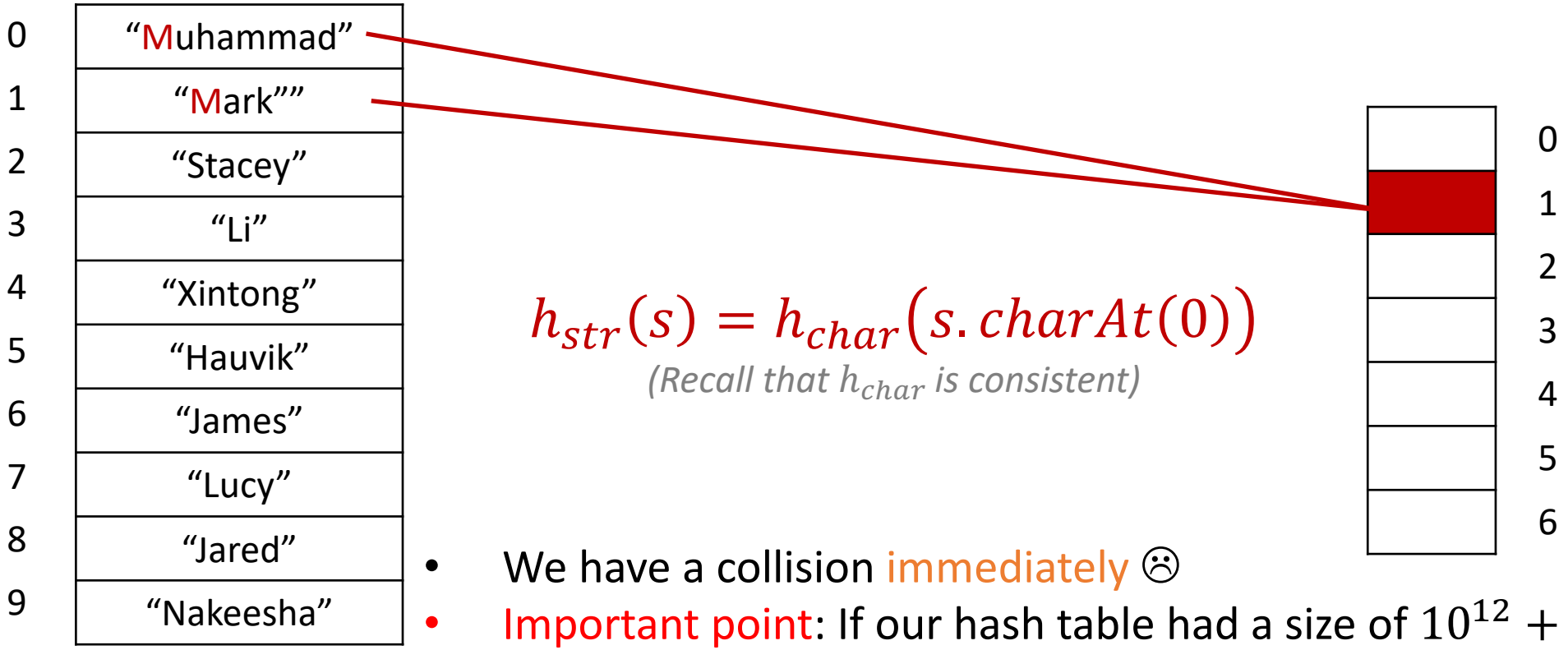


$$h_{str}(s) = h_{char}\big(s.charAt(0)\big)$$

(Recall that h_{char} is consistent)

- We have a collision immediately ☹

A terrible hash function for strings

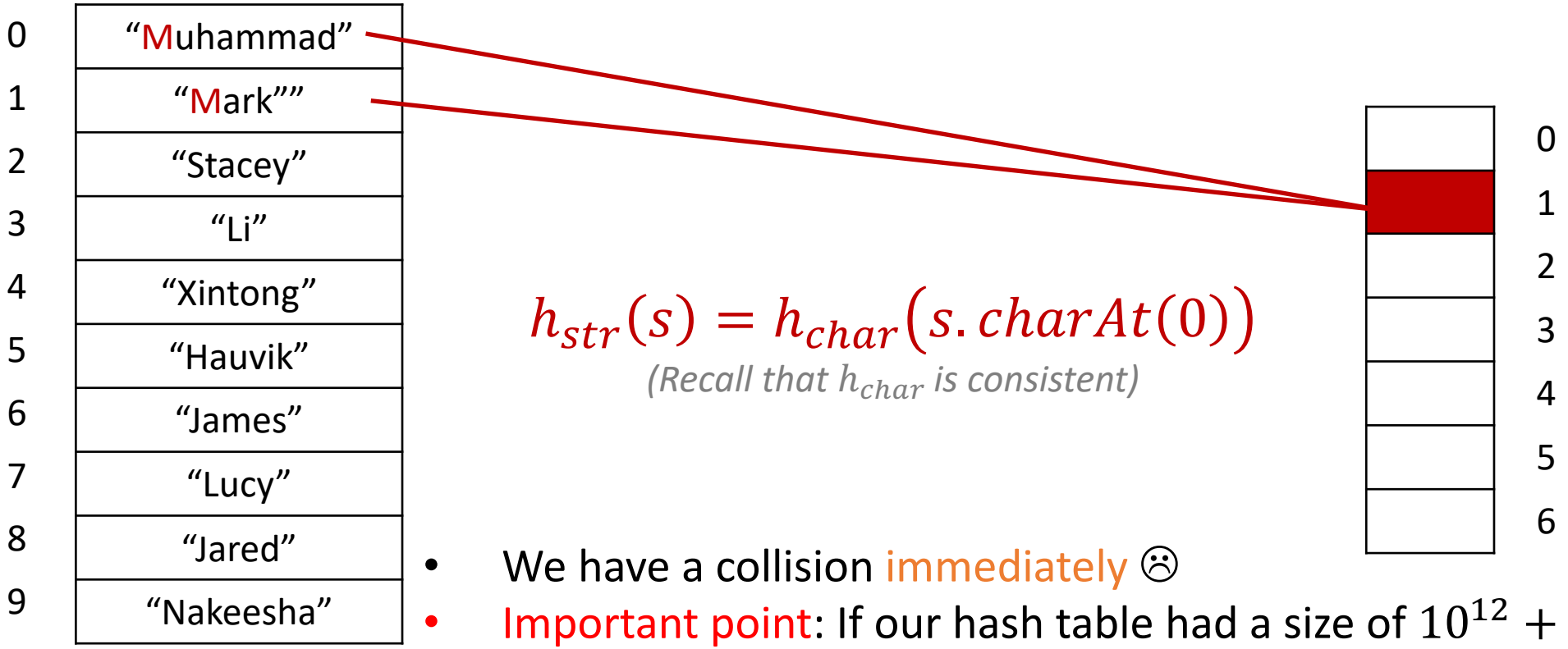


$$h_{str}(s) = h_{char}\big(s.charAt(0)\big)$$

(Recall that h_{char} is consistent)

- We have a collision immediately ☹
- Important point: If our hash table had a size of $10^{12} + 1$, we would **still** have a collision!

A terrible hash function for strings



$$h_{str}(s) = h_{char}\big(s.charAt(0)\big)$$

(Recall that h_{char} is consistent)

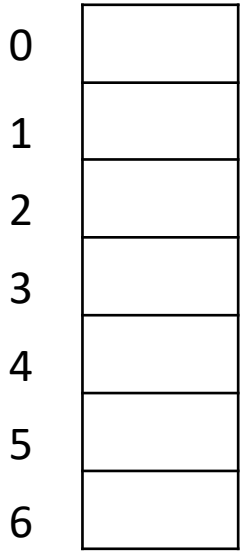
- We have a collision immediately ☹
- Important point: If our hash table had a size of $10^{12} + 1$, we would **still** have a collision!
- Take-home message: **It's not enough to have a lot of memory.**
  - We also need quality, **approximately uniform** hash functions!

Probes

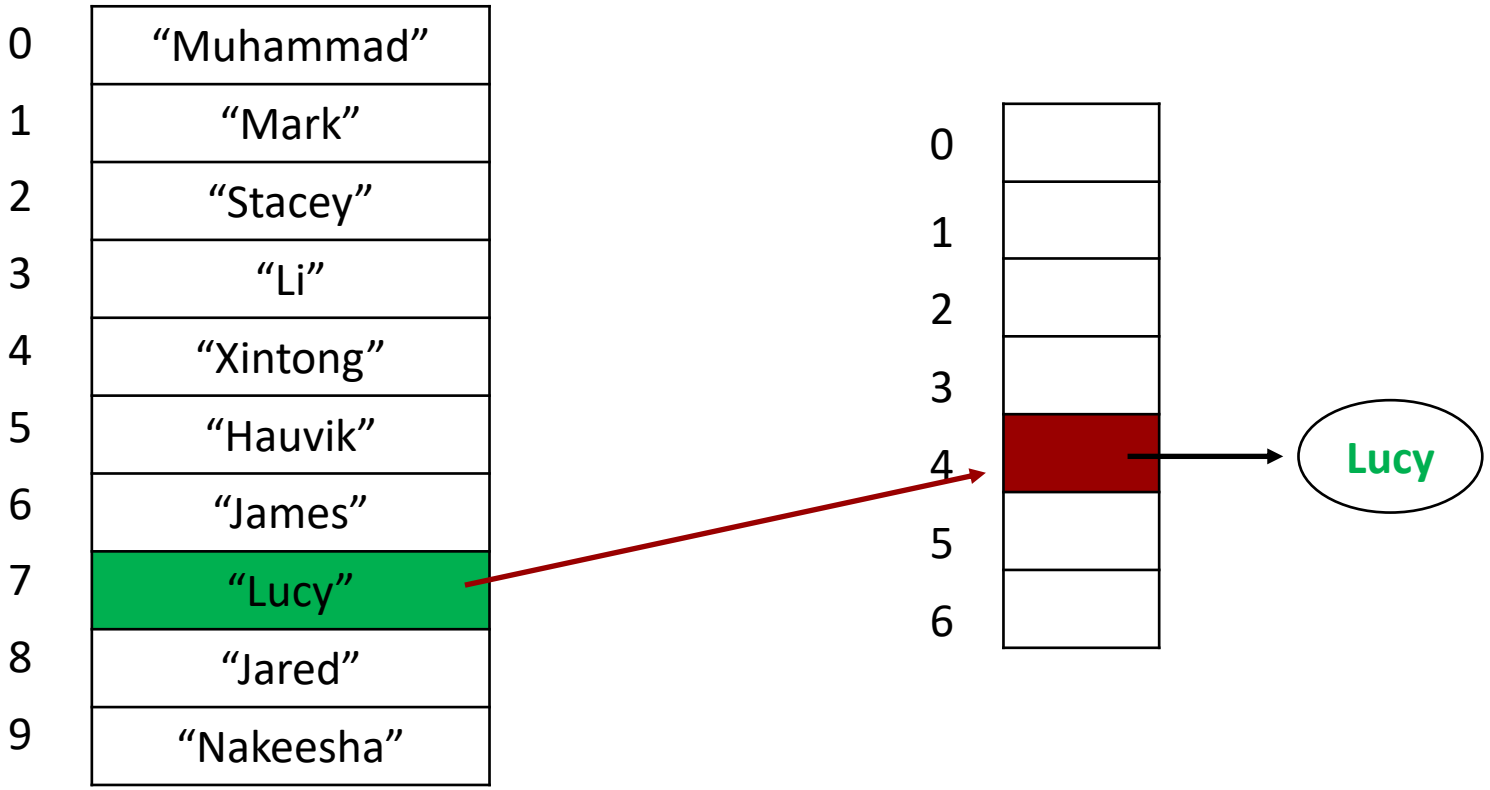
- Suppose that we want to insert a key k in our hash table.
- Every memory access we need to make until we find an empty position *(including the first and last accesses)* we will call a **probe**
- Goal of collision resolution: minimize probes while **not requiring frequent array resizings**
- With two seemingly contradicting goals, it is a challenge to come up with good solutions to this problem!

Separate Chaining

0	“Muhammad”
1	“Mark”
2	“Stacey”
3	“Li”
4	“Xintong”
5	“Hauvik”
6	“James”
7	“Lucy”
8	“Jared”
9	“Nakeesha”

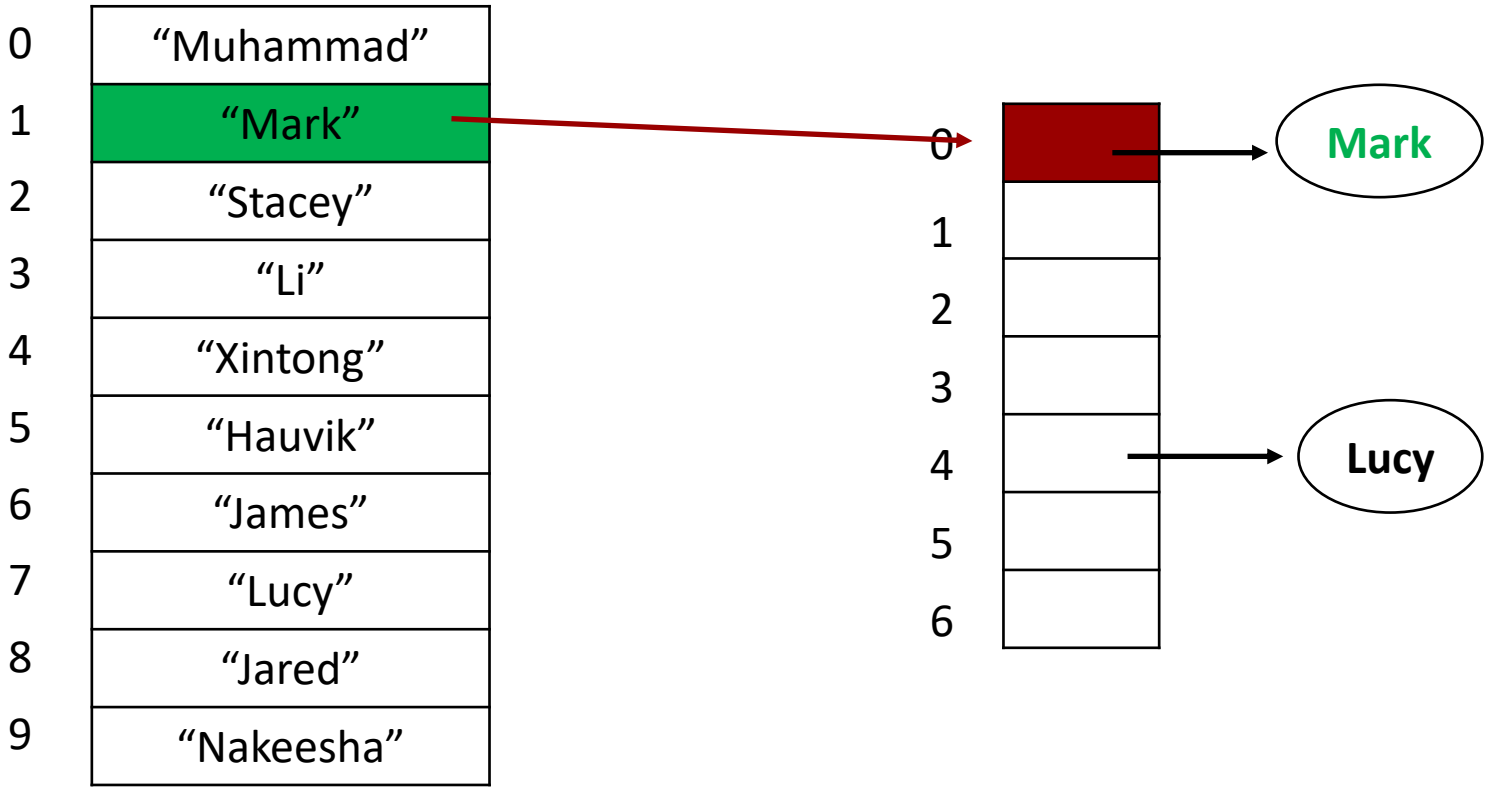


Separate Chaining



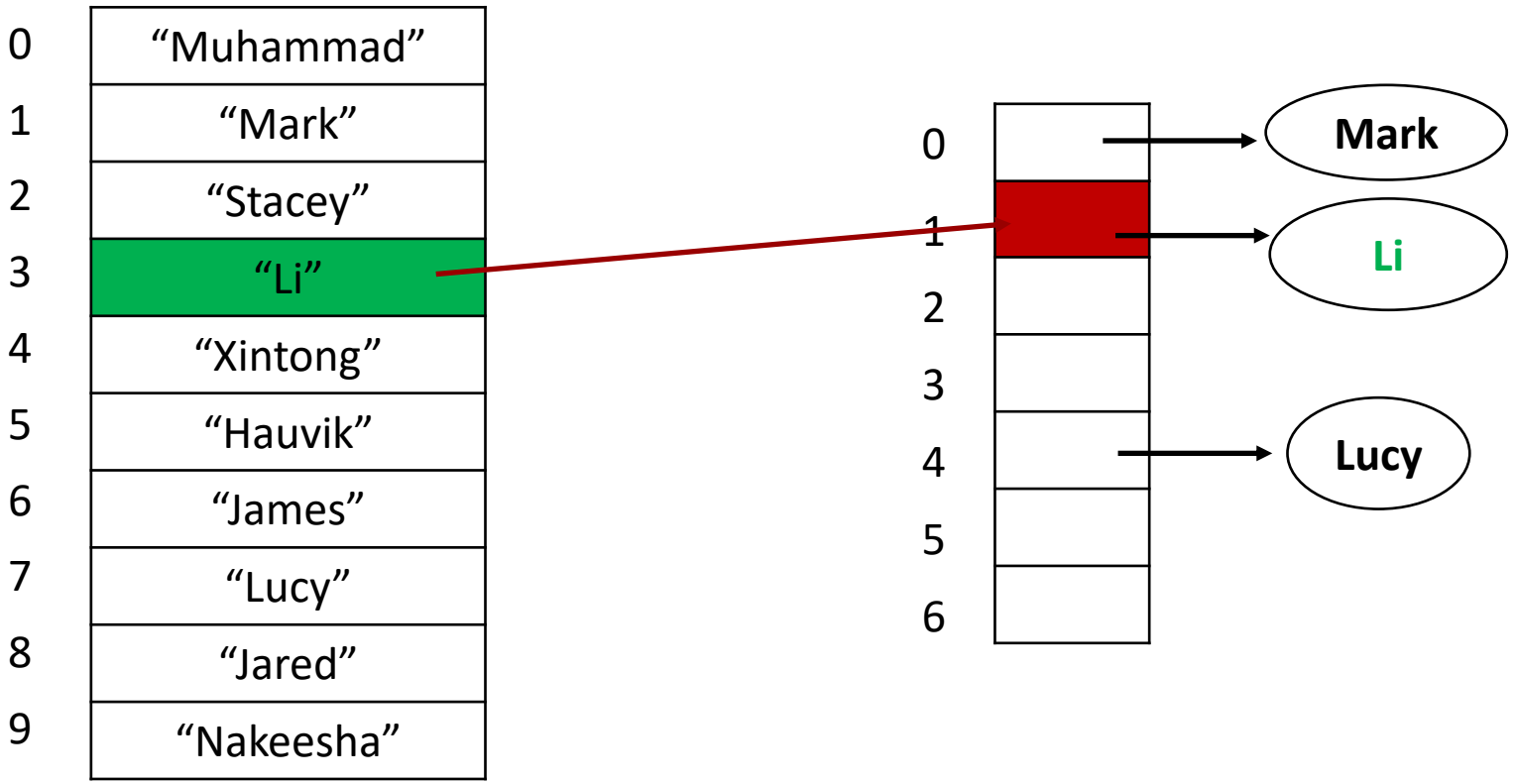
Insert Lucy

Separate Chaining



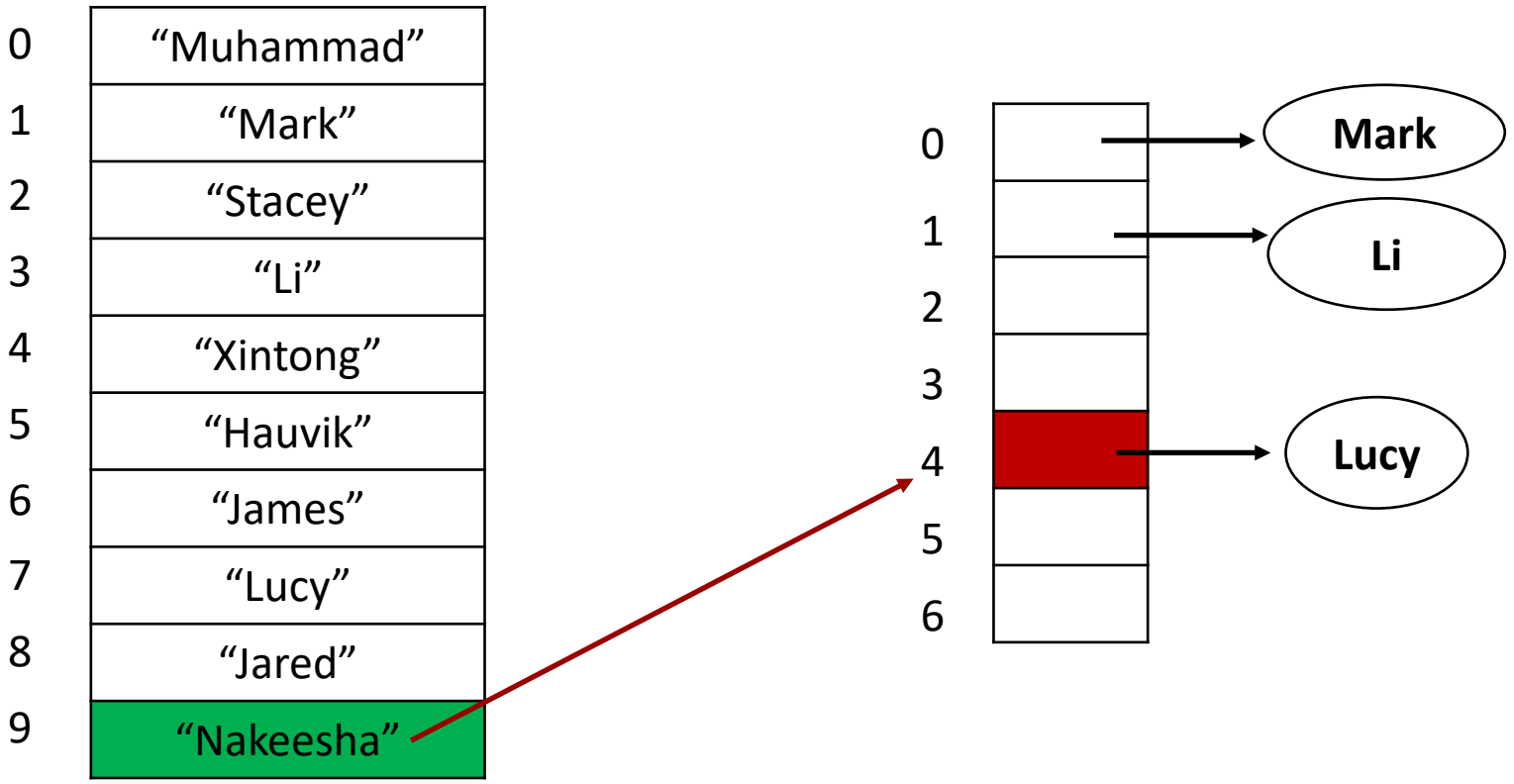
Insert Mark

Separate Chaining



Insert Li

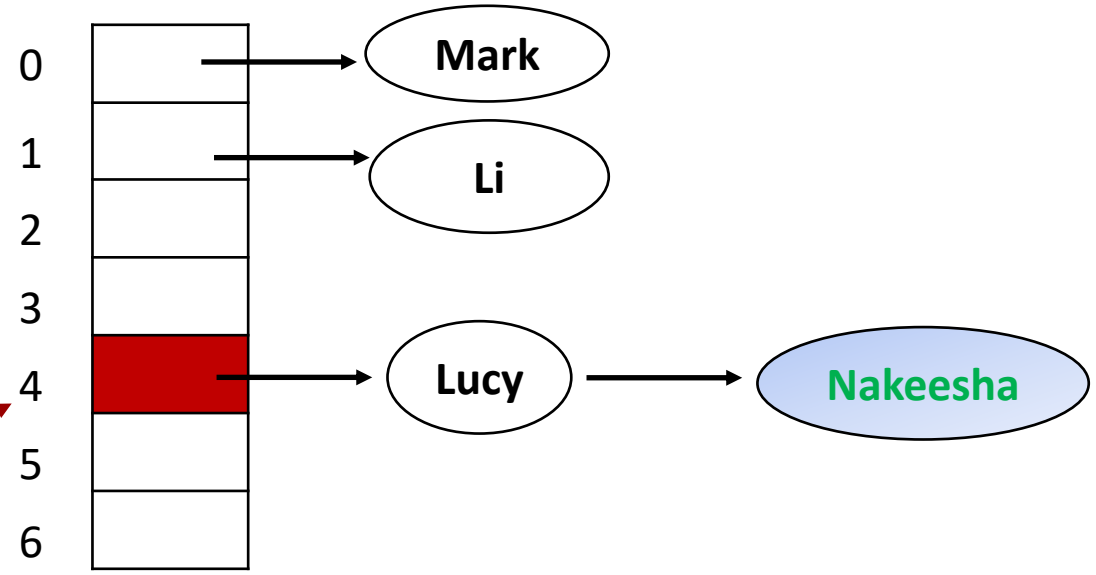
Separate Chaining



Suppose that inserting Nakeesha causes a collision…

Separate Chaining

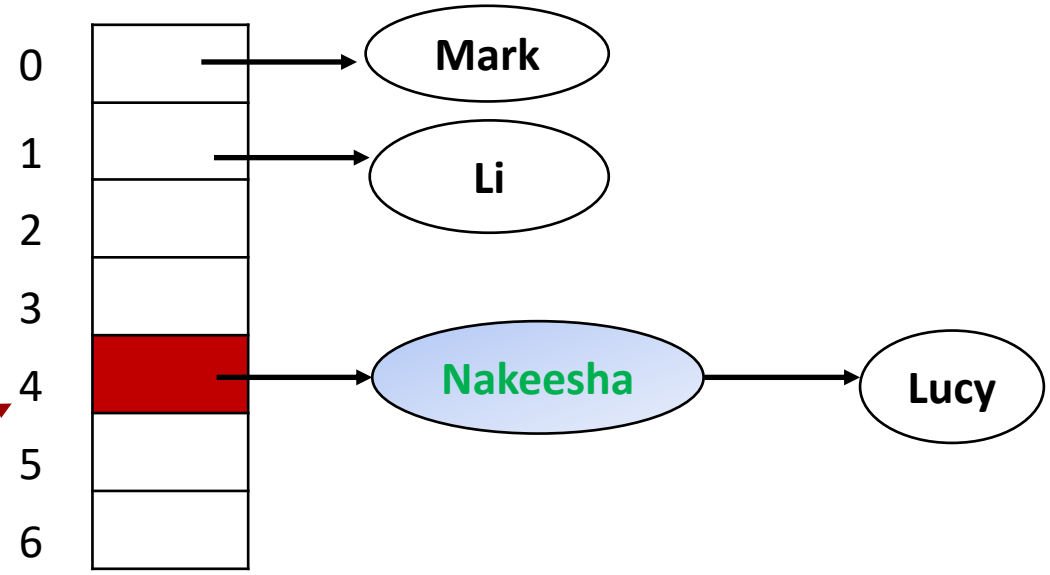
0	“Muhammad”
1	“Mark”
2	“Stacey”
3	“Li”
4	“Xintong”
5	“Hauvik”
6	“James”
7	“Lucy”
8	“Jared”
9	“Nakeesha”



- Trivially solved by inserting at the end of our linked list!
- This can be done in $\mathcal{O}(1)$ if we hold an end pointer at the head of the list

Separate Chaining

0	“Muhammad”
1	“Mark”
2	“Stacey”
3	“Li”
4	“Xintong”
5	“Hauvik”
6	“James”
7	“Lucy”
8	“Jared”
9	“Nakeesha”

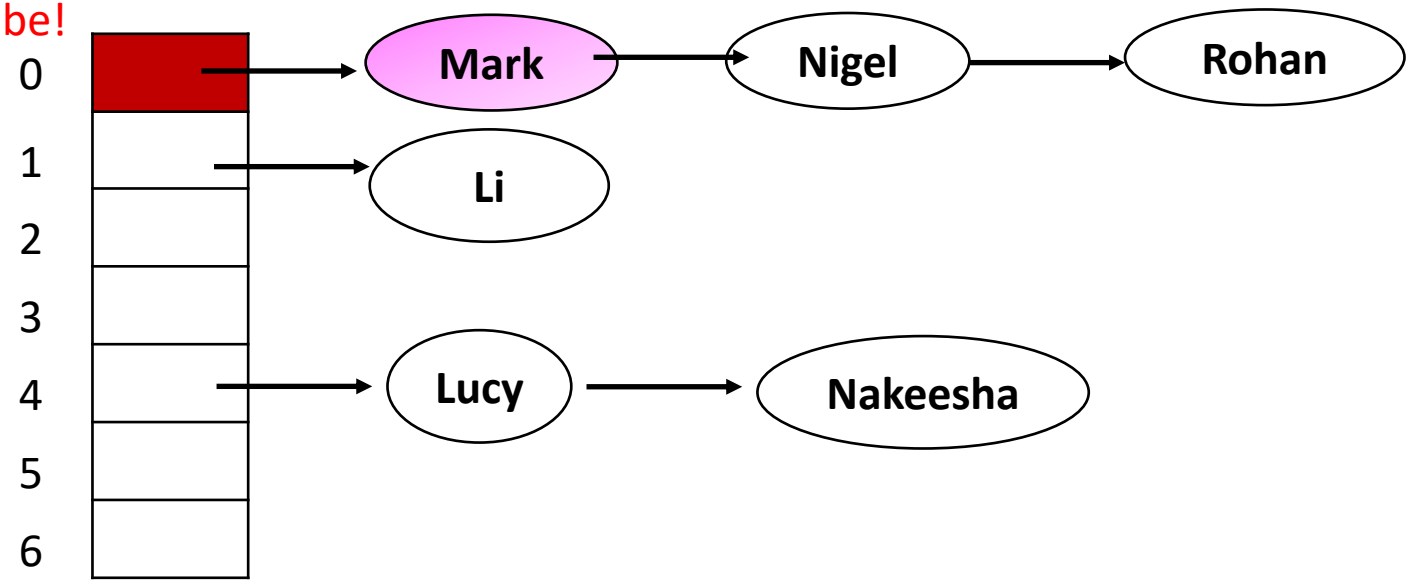


- Or we could just insert at the beginning, again in $\mathcal{O}(1)$ time

(Ordered!) Separate Chaining

0	“Muhammad”
1	“Mark”
2	“Stacey”
3	“Li”
4	“Xintong”
5	“Hauvik”
6	“James”
7	“Lucy”
8	“Jared”
9	“Nakeesha”

Jared probes buffer[0] = node containing Mark, compares name with Mark’s, immediately returns false after a single probe!



- OR, we can accept linear insertion cost to keep the buckets **sorted**
- Then, searches doomed to fail will do so fast!
- This is an example of ordered hashing ☺

Separate Chaining with a perfectly uniform hash function….

- For all keys k, $[h(k) \sim UNIFORM(\{0, 1, 2, \dots, M-1\})]$
- At the limit, all of the lists will have $^{n}/_{M}$ keys per bucket!
- This happens at the limit of infinite insertions (don't fall for the gambler's fallacy)

Average worst-case search in a Separately Chained HT

- Let’s remember something about BSTs…

Average worst-case search in a Separately Chained HT

- Let's remember something about BSTs…
- We were told that the best possible height we could expect to make our search fast would be $\log_2 n$.

Average worst-case search in a Separately Chained HT

- Let's remember something about BSTs…
- We were told that the best possible height we could expect to make our search fast would be $\log_2 n$.
- Then, somebody else said something else: "That's what it's going to be on average!"

Average worst case vs amortized

- Do not confuse the average worst case complexity of an operation with the amortized complexity of an operation.
- Average worst case complexity: I will assume a probability distribution (usually uniform) over my data values (in our case, over the outputs of h), and I will calculate the expected cost of an operation (e.g search, insertion)

Average worst case vs amortized

- Do not confuse the average worst case complexity of an operation with the amortized complexity of an operation.
- Average worst case complexity: I will assume a probability distribution (usually uniform) over my data values (in our case, over the outputs of h), and I will calculate the expected cost of an operation (e.g search, insertion)
- For example, average-case analysis of insertion into a BST, which also provides the logarithmic height property assumes a uniform distribution over your keys, which can also be interpreted in the following manner:

Average worst case vs amortized

- Do not confuse the average worst case complexity of an operation with the amortized complexity of an operation.
- Average worst case complexity: I will assume a probability distribution (usually uniform) over my data values (in our case, over the outputs of h), and I will calculate the expected cost of an operation (e.g search, insertion)
- For example, average-case analysis of insertion into a BST, which also provides the logarithmic height property assumes a uniform distribution over your keys, which can also be interpreted in the following manner:

*Let T be **any non-null subtree** in your BST and k be a new key. Then, if the insertion routine reaches the root r of T and compares k with the data point d* contained by r, $P(k < d) = P(k > d) = 0.5$.

Average worst case vs amortized

- Do not confuse the average worst case complexity of an operation with the amortized complexity of an operation.
- Average worst case complexity: I will assume a probability distribution (usually uniform) over my data values (in our case, over the outputs of h), and I will calculate the expected cost of an operation (e.g search, insertion)
- For example, average-case analysis of insertion into a BST, which also provides the logarithmic height property assumes a uniform distribution over your keys, which can also be interpreted in the following manner:

*Let T be **any non-null subtree** in your BST and k be a new key. Then, if the insertion routine reaches the root r of T and compares k with the data point d* contained by r, $P(k < d) = P(k > d) = 0.5$.

(Same derivation as binary search!)

- So, $E[h(T)] = 0 + 0.5 * E[h(T.left)] + 0.5 * E[h(T.right)] = \log_2 n$

Average worst case vs amortized

- On the other hand, amortized complexity assumes k operations (insert, delete, search, nearest-neighbor, range, longestPrefix…) and answers the question: If I have n objects in my database, what function $f(n)$ is the tightest possible bound for

$$T(all\ k\ operations) = \mathcal{O}(f(n))?$$

- Then, on average, my amortized cost per operation will be $\frac{c \cdot f(n)}{k}$ for the c in the definition of "big-Oh".
  - Recall ArrayLists and Splay Trees (slides on splay trees posted on ELMS).

Pros of Separate Chaining

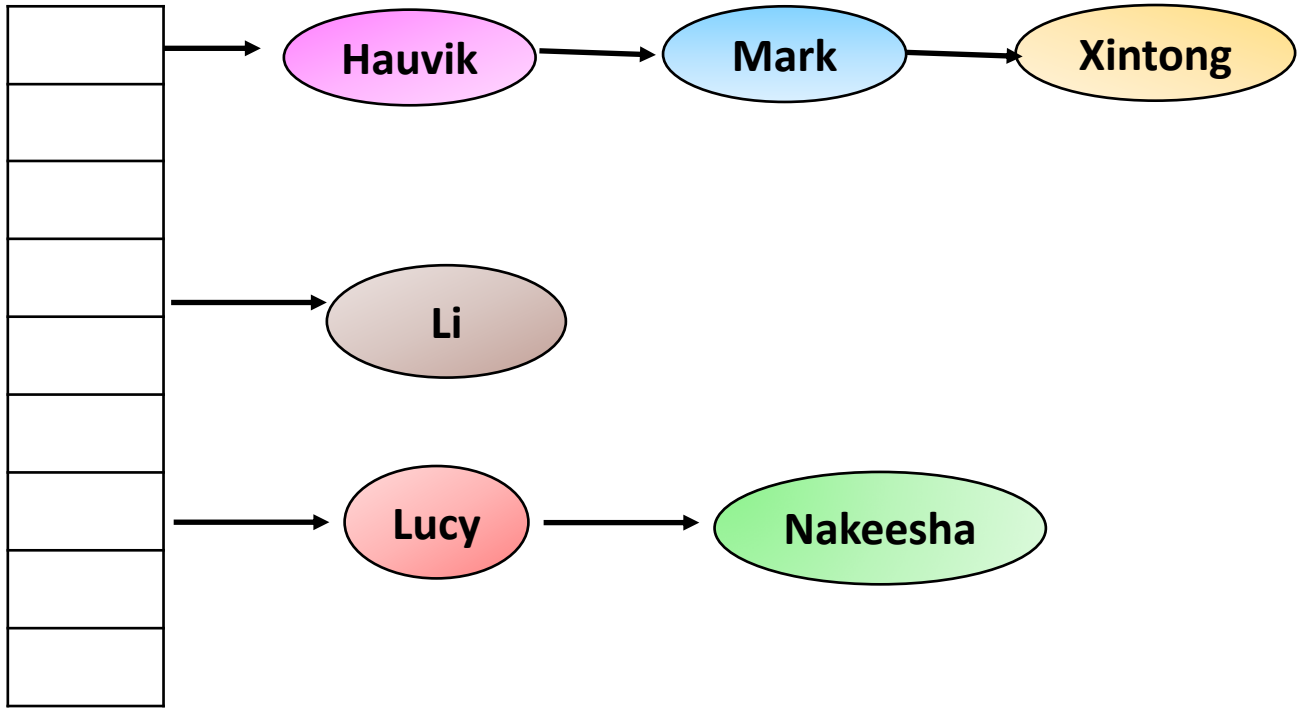
- Implementable in 2 minutes
- Allows us to retrieve pointer to storage medium for K-V pairs (maybe the V is another hash!)
- Great idea for estimating quality of hash function!
- Non-contiguous memory allocation of keys adds flexibility of memory allocation of keys
  - Important when we are memory starved
- Trade-off between insertion speed and (failed) search average probes…
  - Remember: data will be searched much, much more than it will be mutated!

Cons of Separate Chaining

- After a while, we just lose constant search!
- Seems kind of dumb to be spending so much space for just 32-bit pointers instead of the KV-pair itself.
- Non-contiguous storage also means that we cannot take advantage of **caching**! ☹
- Searches doomed to fail can be optimized if we are willing to pay linear time in insertions.
  - But **successful searches do not**: linear time will be required for those no matter what we do in insertions.
  - The constant $\frac{1}{M}$ is good only if M is a large prime (this is not usually a problem).

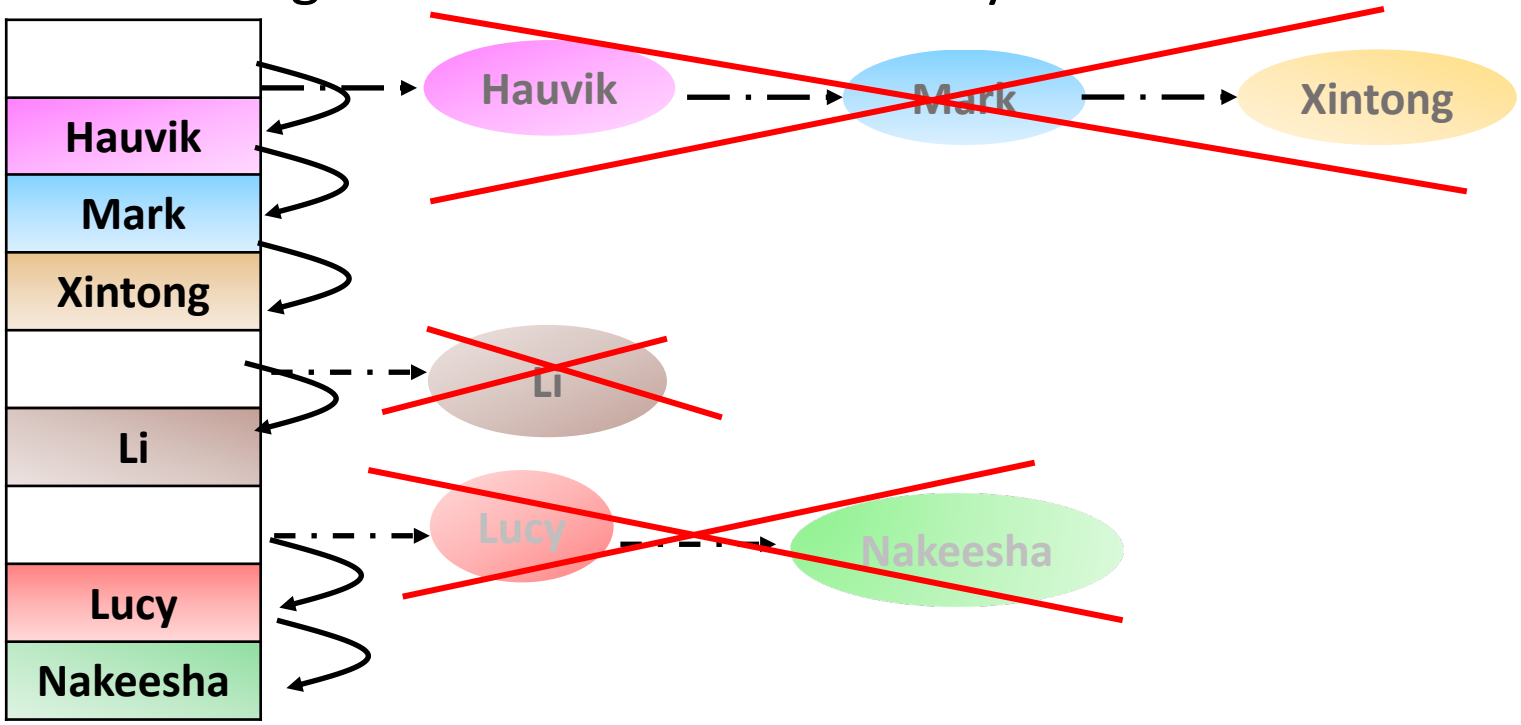
Memory mis—use…

- If we look at a separated chaining – based hash table closely:



Memory mis—use…

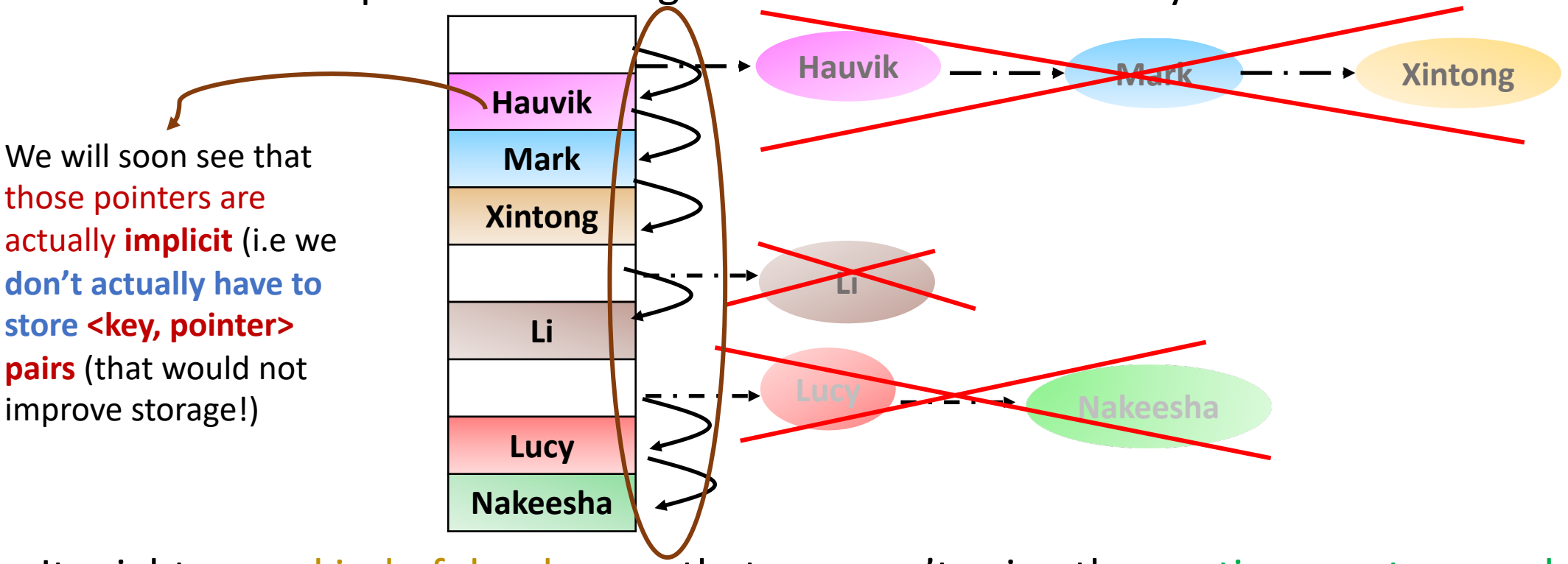
- If we look at a separated chaining – based hash table closely:



- It might seem kind of dumb even that we aren't using the contiguous storage already offered to us to store the keys, instead of wasting space for pointers!

Memory mis—use…

- If we look at a separated chaining – based hash table closely:



We will soon see that those pointers are actually **implicit** (i.e we **don't actually have to store <key, pointer> pairs** (that would not improve storage!)

- It might seem kind of dumb even that we aren't using the contiguous storage already offered to us to store the keys, instead of wasting space for pointers!

Open Addressing methods

- It turns out that this is something done a lot in hashing!
- All methods that perform collision resolution in the internal buffer itself are known as ***open addressing*** or ***closed hashing*** methods (relatively confusing, sorry).

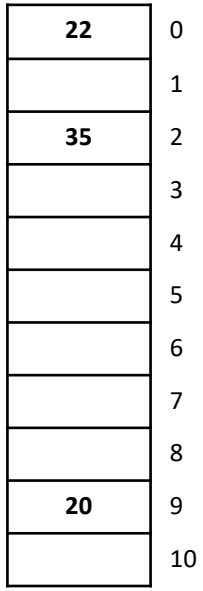
Open Addressing methods

- It turns out that this is something done a lot in hashing!
- All methods that perform collision resolution in the internal buffer itself are known as ***open addressing*** or ***closed hashing*** methods (relatively confusing, sorry).
- The simplest Open Addressing method is linear probing.
- Logic of linear probing:
  - When inserting, if we encounter a collision, keep walking into the buffer by making one step forward at a time, **wrapping around at the end.**
  - The first empty buffer cell is where you will store your key! ☺

Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

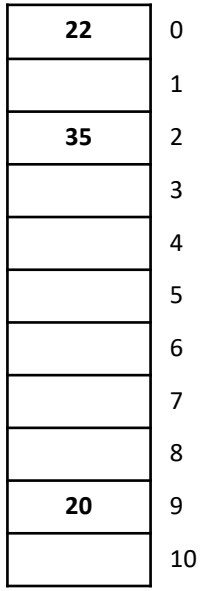


Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

1. Insert 29

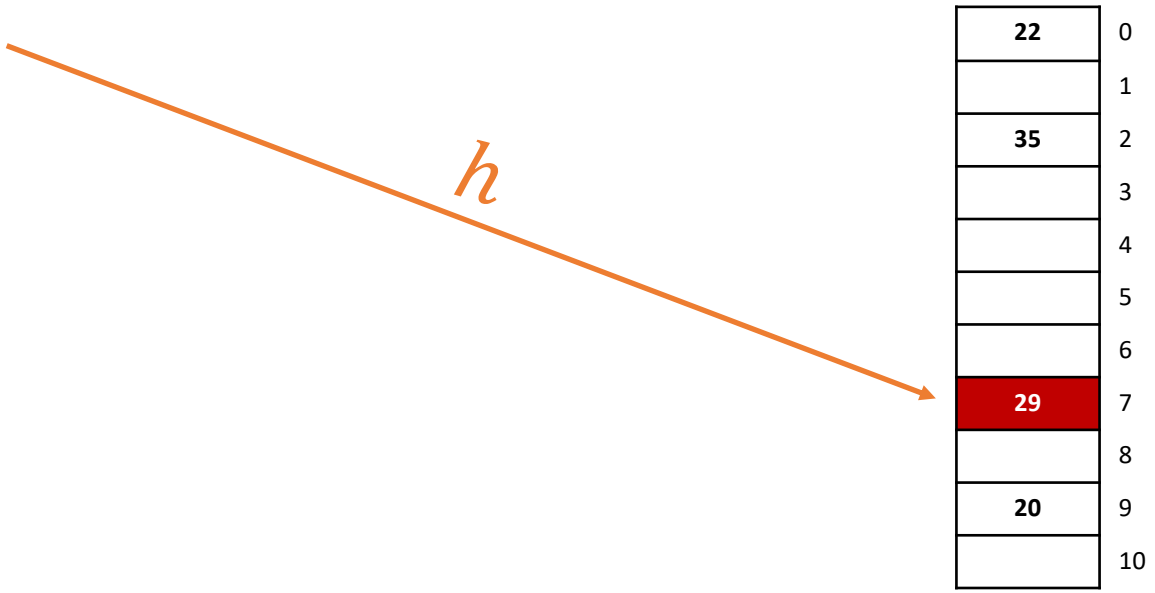


Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

1. Insert 29

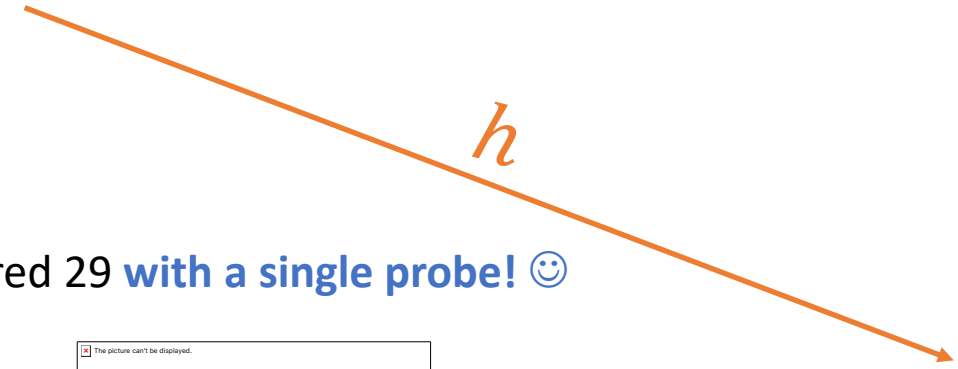


Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

1. Insert 29



22	0
	1
35	2
	3
	4
	5
	6
29	7
	8
20	9
	10

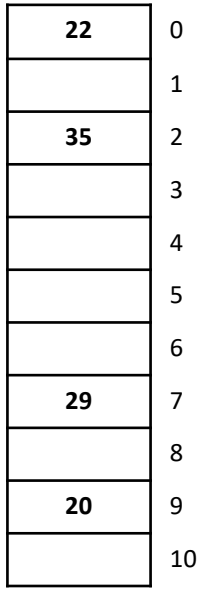
We're done! We stored 29 **with a single probe!** ☺

Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

2. Insert 13



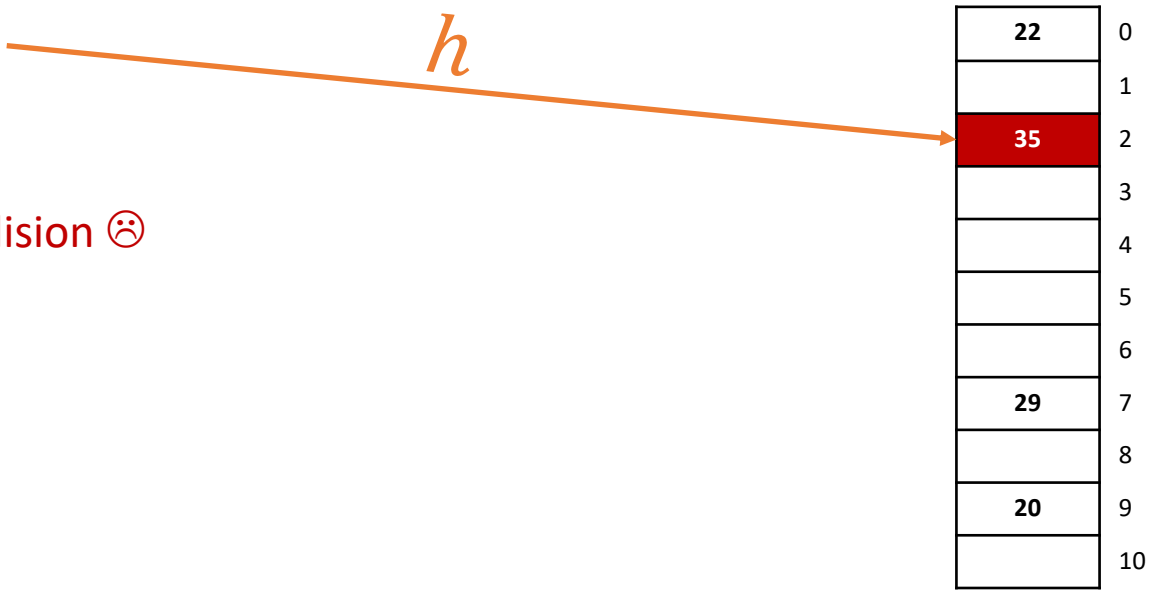
Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

2. Insert 13

We have a collision ☹

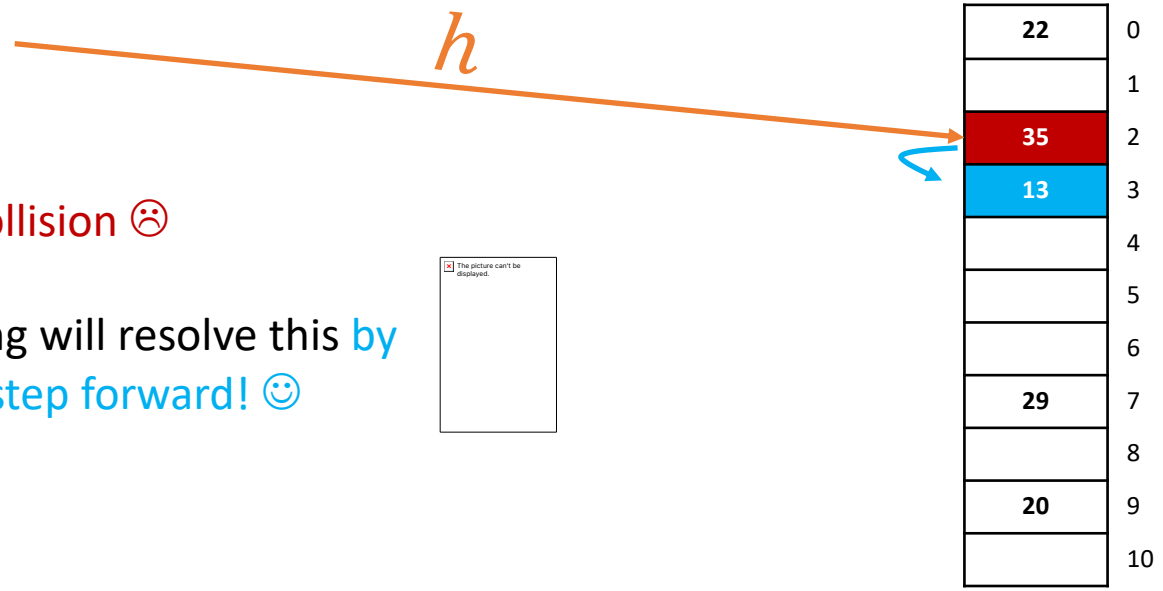


Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

2. Insert 13



We have a collision ☹

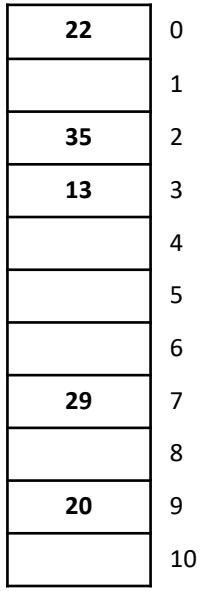
Linear probing will resolve this by moving one step forward! ☺

Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

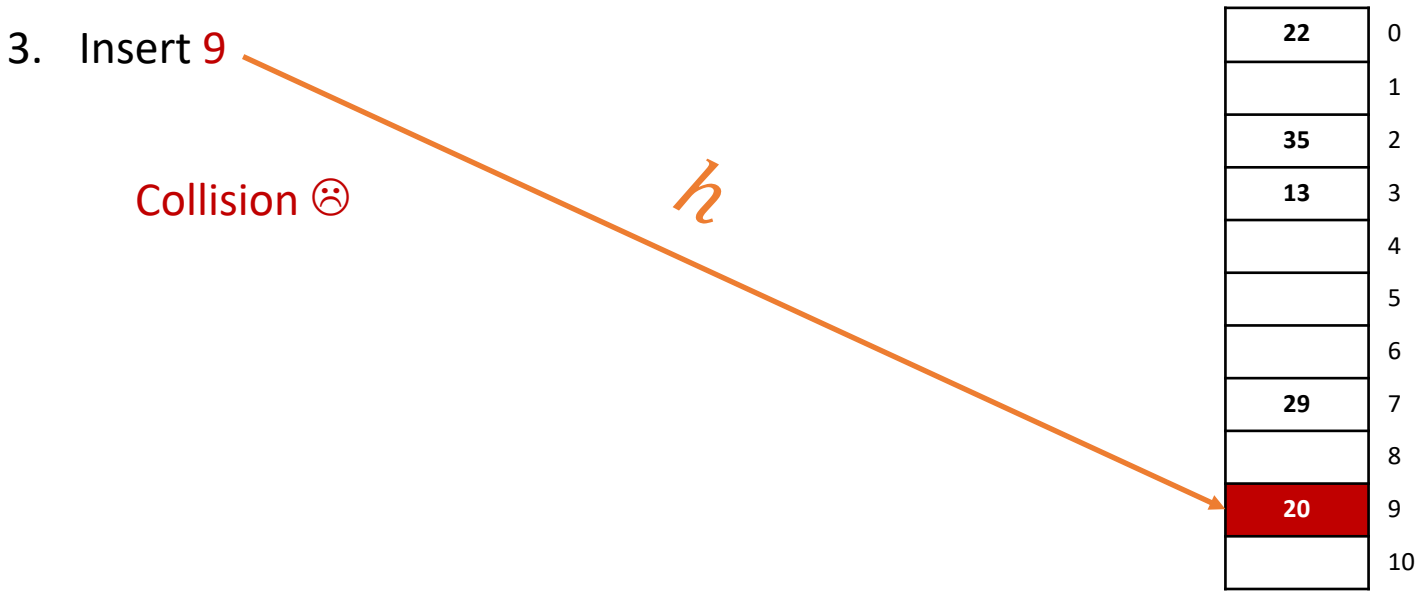
3. Insert 9



Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

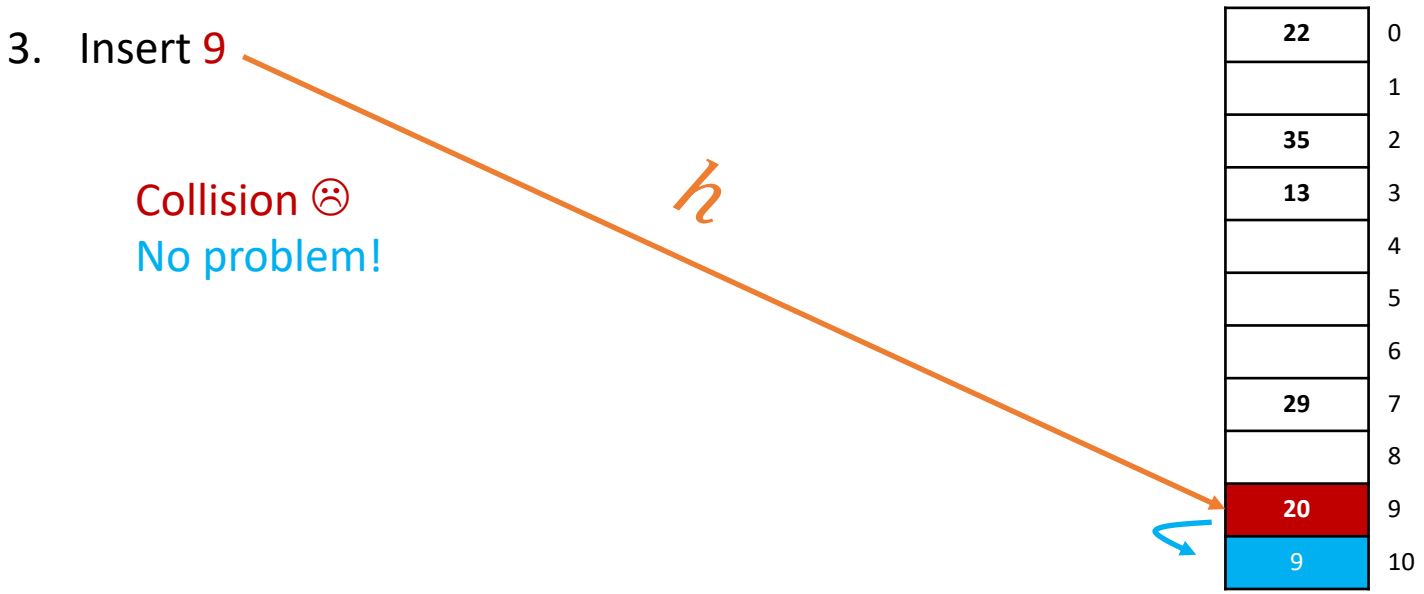
$$h(n) = n \,\%\, M$$



Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

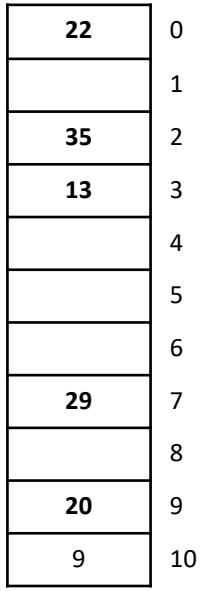


Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

4. Insert 31



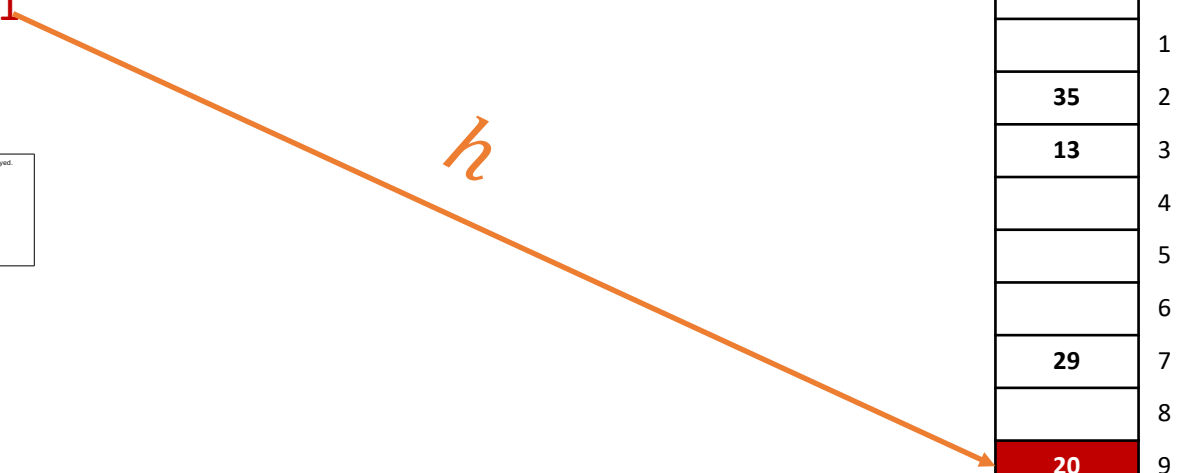
Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

4. Insert 31

Collision



22	0
	1
35	2
13	3
	4
	5
	6
29	7
	8
20	9
9	10

Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

4. Insert 31

Collision

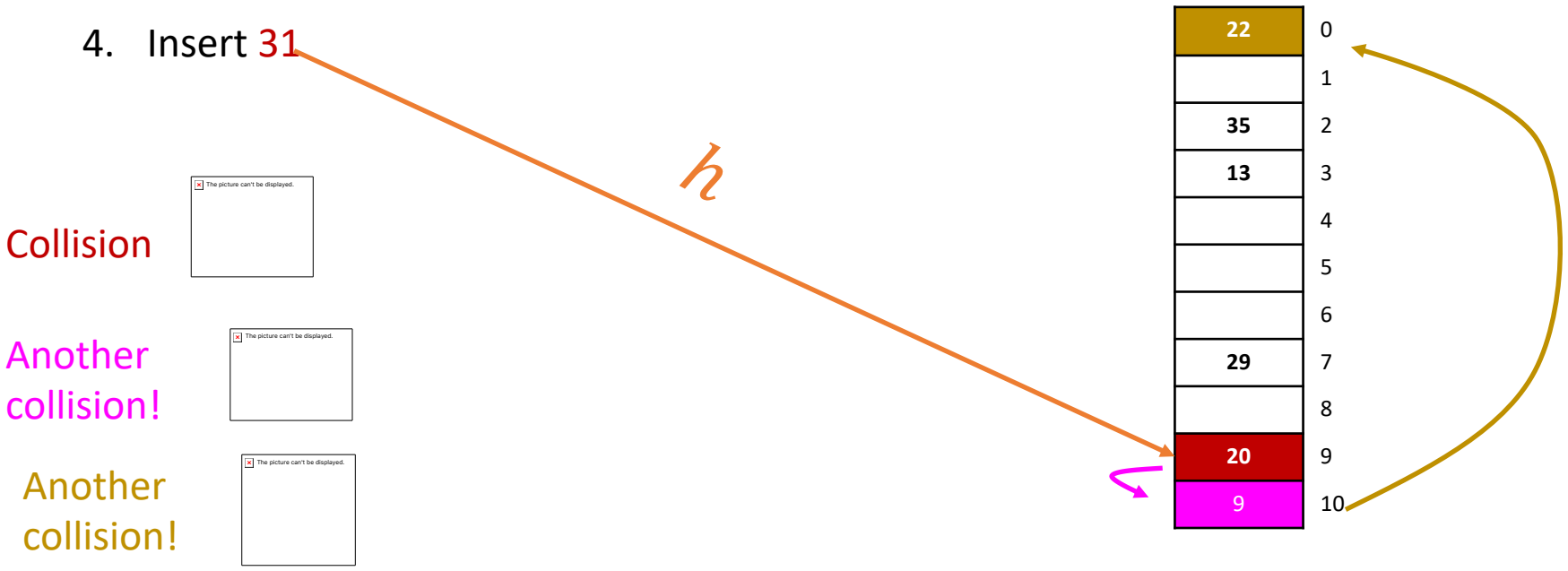
Another collision!

Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

4. Insert 31

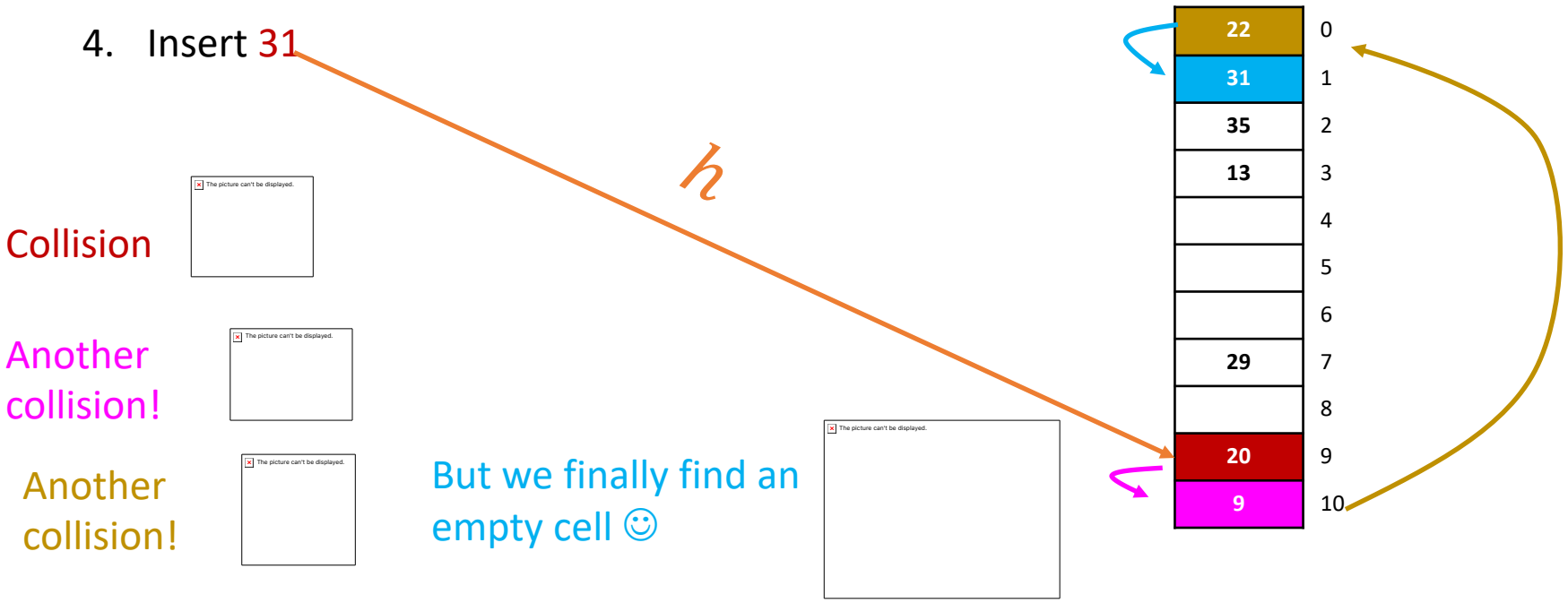


Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

4. Insert 31

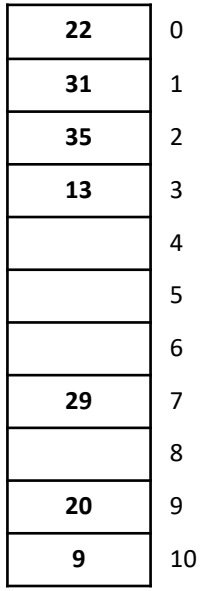


Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

4. Insert 11

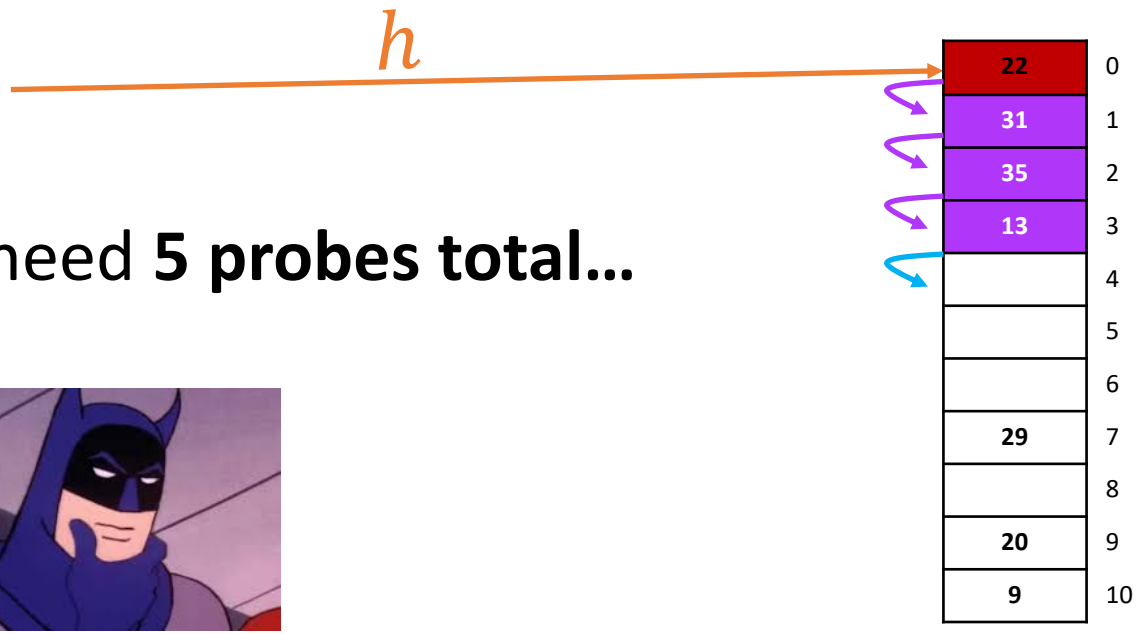


Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

5. Insert 11



We'd need **5 probes total…**

Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

5. Insert 11

22	0
31	1
35	2
13	3
	4
	5
	6
29	7
	8
20	9
9	10

Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

5. Insert 11

Aren't things getting a **bit too crowded** in the buffer?

22	0
31	1
35	2
13	3
	4
	5
	6
29	7
	8
20	9
9	10

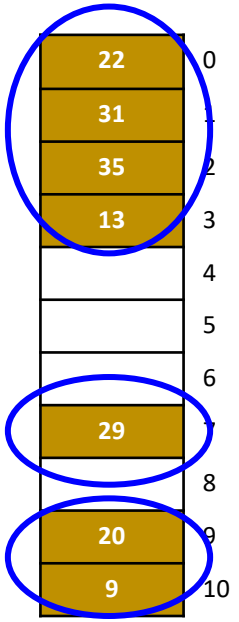
Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

5. Insert 11

The solution: **Expand the size of the buffer** and re-insert all keys, before we insert 11!



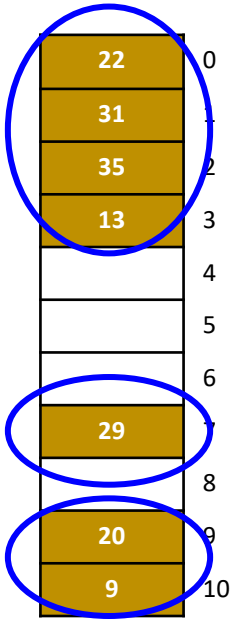
Insertion

- Suppose that we want to hash integers based on a simple modular hash function h:

$$h(n) = n \,\%\, M$$

5. Insert 11

The solution: **Expand the size of the buffer** and re-insert all keys, before we insert 11!



1. *Load factor* $\alpha = \frac{7}{11} = 0.63$
2. In practice, we expand ***immediately after*** we detect $\alpha = 0.5$

Resizing our buffer

- Remember that we want M to be a prime .
- But many primes are *twin* primes, which means that for some primes p, $p + 2$ is also a prime!
  - Examples: $(3, 5)$, $(5\ , 7)$, $(11, 13)$, $(17, 19)$, $(29, 31)$, … , $(347, 349)$, …
  - Those become increasingly rare as $p \to +\infty$.
  - Twin primes conjecture: there are *infinitely many* twin primes.

Resizing our buffer

- Remember that we want M to be a prime .
- But many primes are *twin* primes, which means that for some primes p, $p + 2$ is also a prime!
  - Examples: $(3, 5)$, $(5\ , 7)$, $(11, 13)$, $(17, 19)$, $(29, 31)$, … , $(347, 349)$, …
  - Those become increasingly rare as $p \to +\infty$.
  - Twin primes conjecture: there are *infinitely many* twin primes.
- So, if our enlargement function $e(M)$ is simply:

$$e(M) = first\ prime\ larger\ than\ M$$

- We will have to re-enlarge again pretty fast ☹

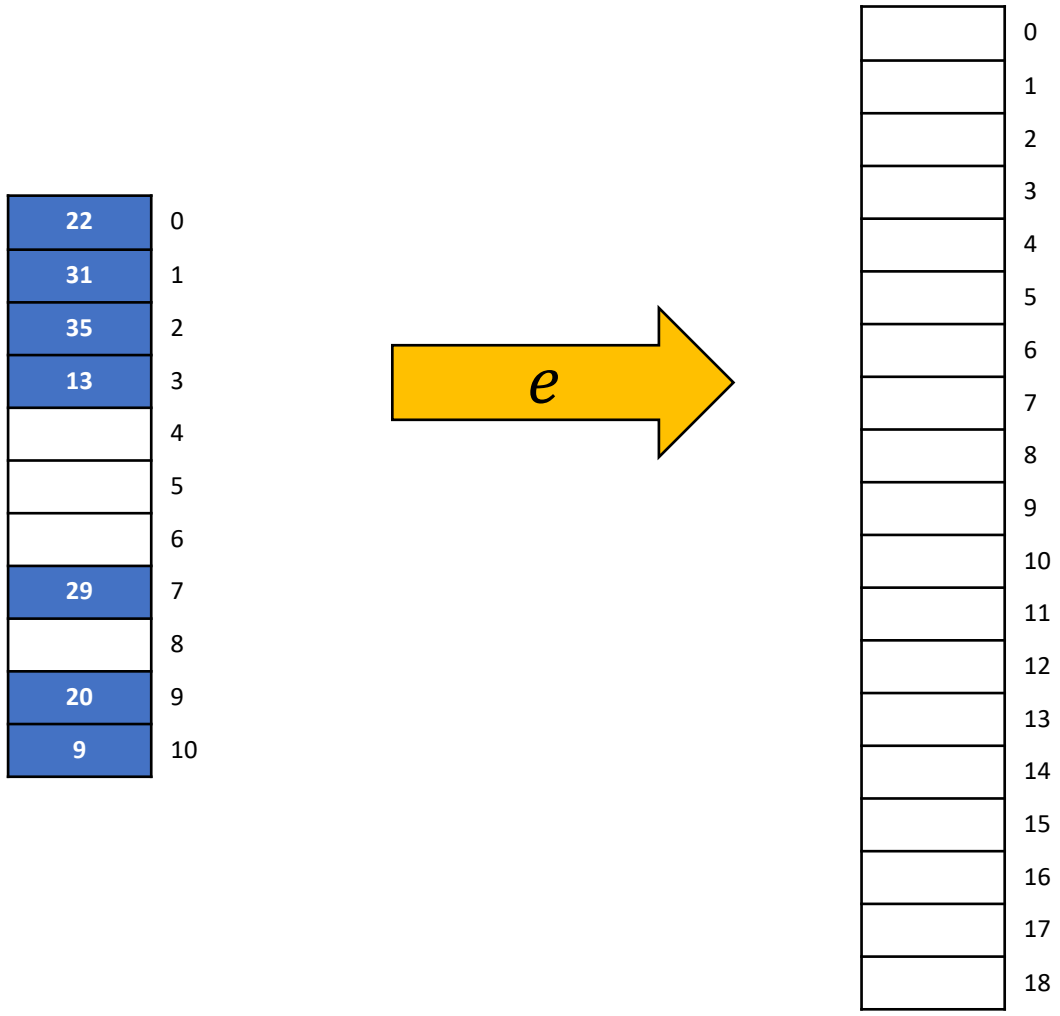
Resizing our buffer

- Here's an idea!

$$e(M) = \textbf{\textit{largest}}\ prime\ \textbf{\textit{smaller}}\ than\ 2 * M$$ 

- Striking a balance between:
  1. Sparse buffer (so more empty cells to insert keys into!)
  2. More random distribution of keys (for modular hash functions, we really want M to be prime).
- This is the approach we will follow with the resizing of this array.

Resizing our buffer

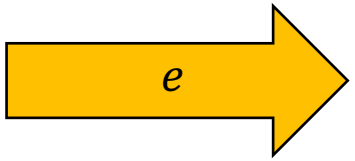


$$h(n) = n \,\%\, M$$

Please reinsert all keys for me real quick ☺

Resizing our buffer

22	0
31	1
35	2
13	3
	4
	5
	6
29	7
	8
20	9
9	10



	0
20	1
	2
22	3
	4
	5
	6
35	7
	8
9	9
29	10
	11
31	12
13	13
	14
	15
	16
	17
	18

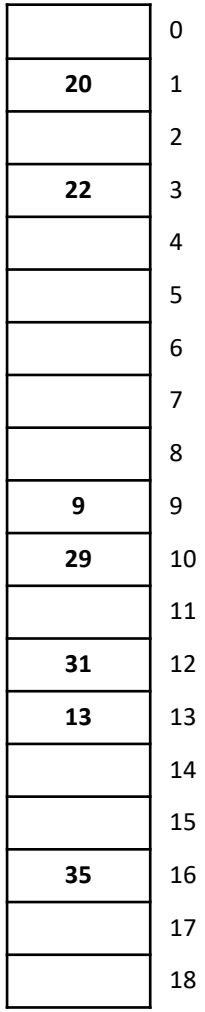
$$h(n) = n \,\%\, M$$

No collisions during re-insertion, and pretty random distribution of keys!

Back to insertion….

$h(n) = n \,\%\, M$

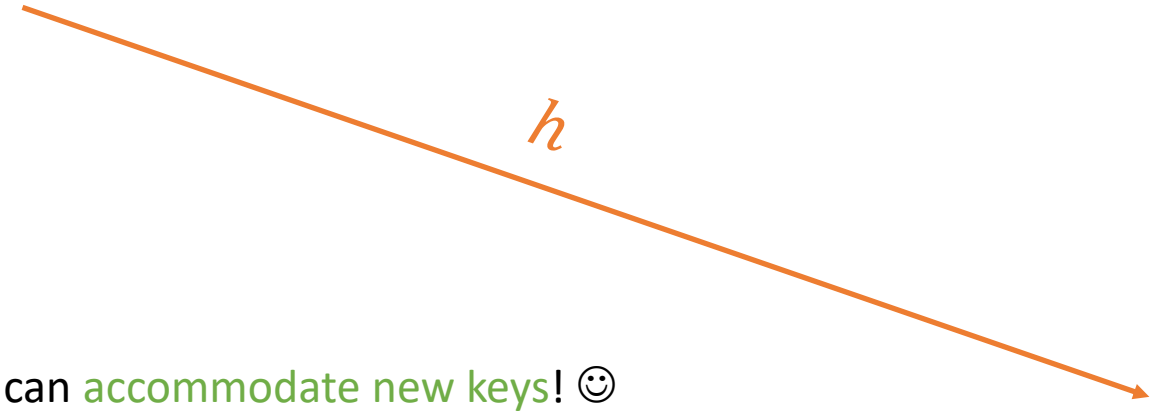
5. Insert 11



Back to insertion….

$$h(n) = n \,\%\, M$$

5. Insert 11



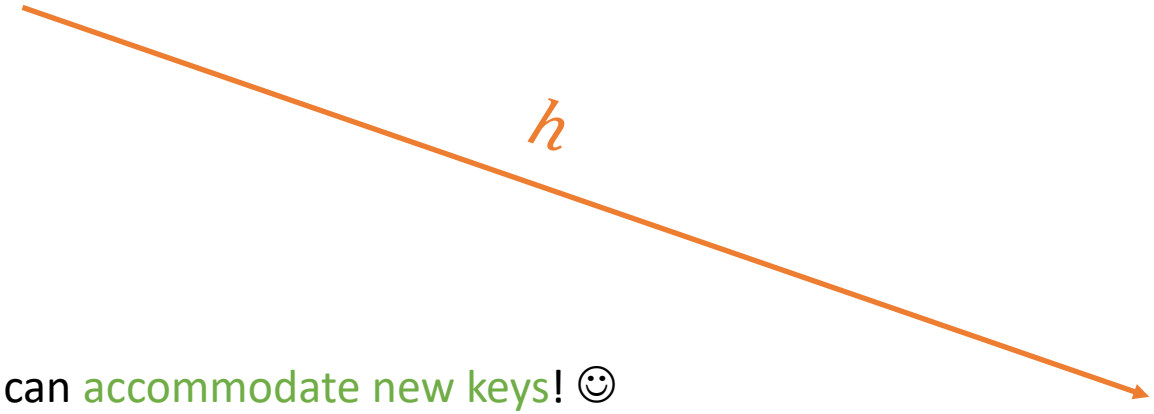
	0
20	1
	2
22	3
	4
	5
	6
35	7
	8
9	9
29	10
11	11
31	12
13	13
	14
	15
35	16
	17
	18

- Now our table can accommodate new keys! ☺
- Until we have to resize again, of course… ☹

Back to insertion….

$h(n) = n \,\%\, M$

5. Insert 11



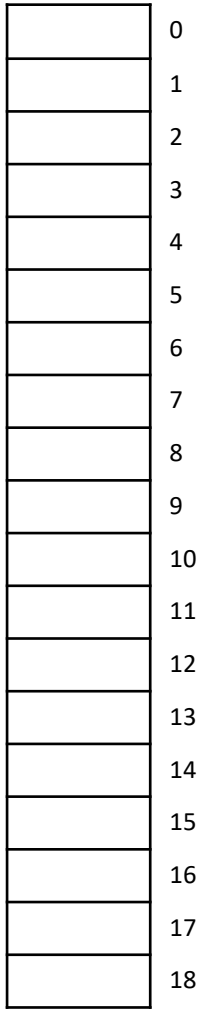
	0
20	1
	2
22	3
	4
	5
	6
35	7
	8
9	9
29	10
11	11
31	12
13	13
	14
	15
35	16
	17
	18

- Now our table can accommodate new keys! ☺
- Until we have to resize again, of course… ☹
  - Heuristic: if $e(M)$ is "close enough" to $2 * M$, we have an `ArrayList`-like resizing
  - Which is exponential, and works quite well in practice ☺

$h(n) = n \,\%\, M$

“Clusters” and “chains”

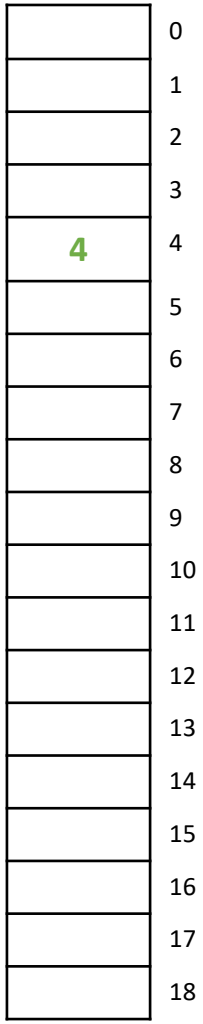
- Insert 4, 23, 5



$h(n) = n \,\%\, M$

“Clusters” and “chains”

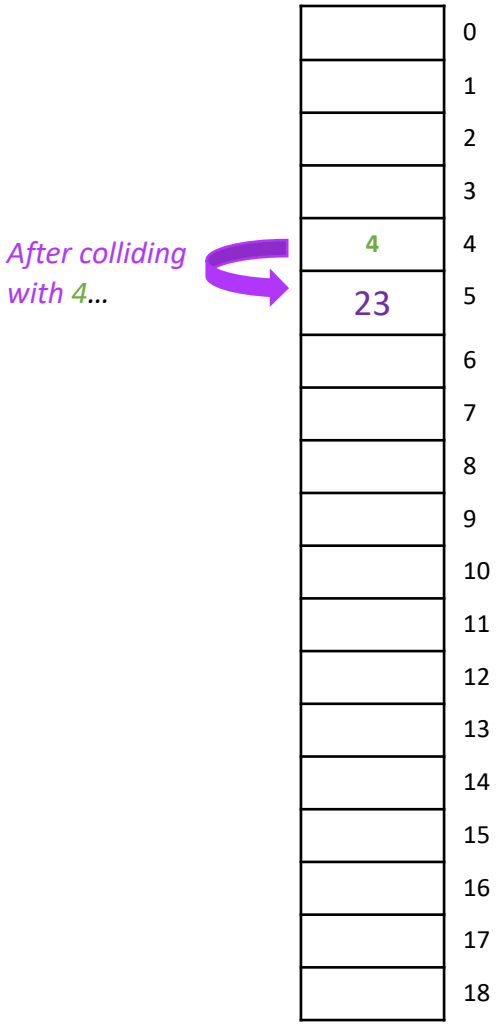
- Insert 4, 23, 5



$h(n) = n \,\%\, M$

“Clusters” and “chains”

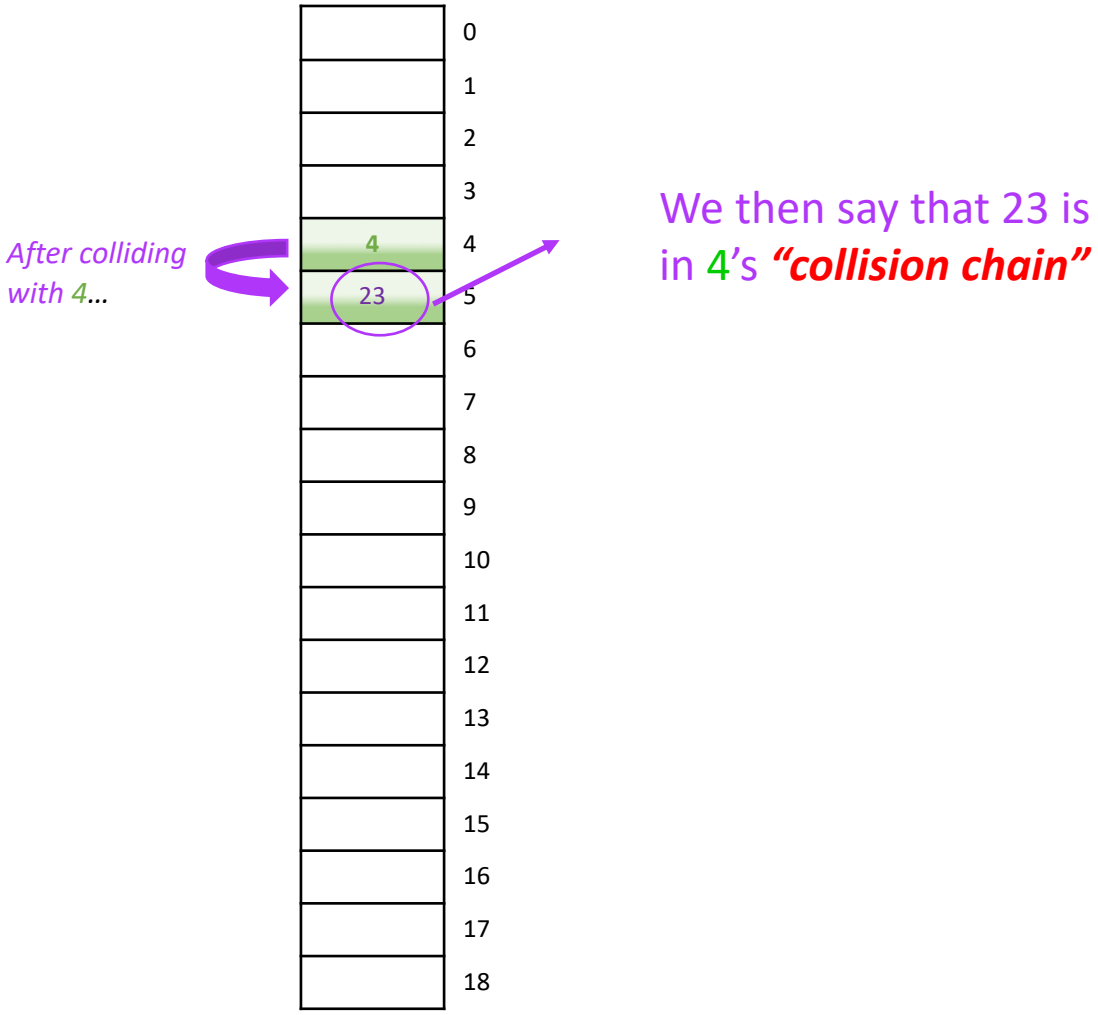
- Insert 4, 23, 5



$h(n) = n \,\%\, M$

“Clusters” and “chains”

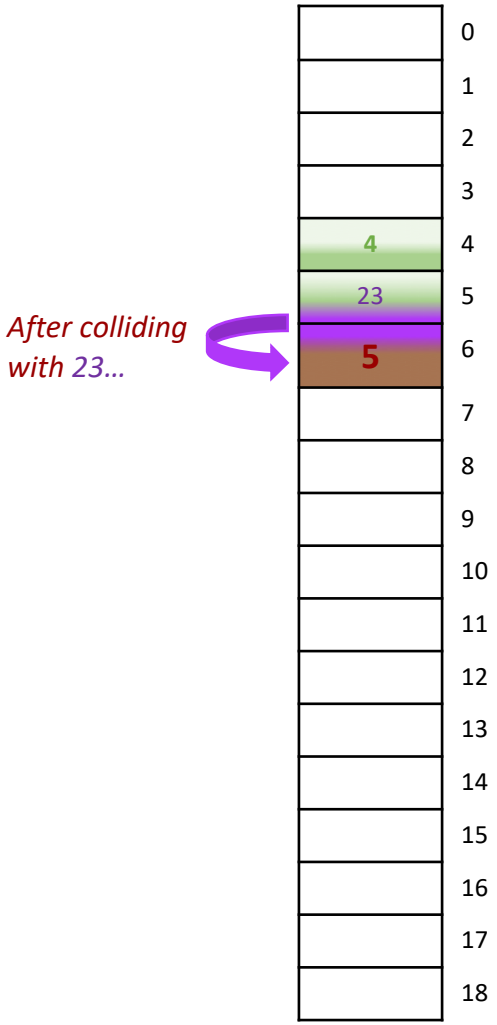
- Insert 4, 23, 5



$h(n) = n \,\%\, M$

“Clusters” and “chains”

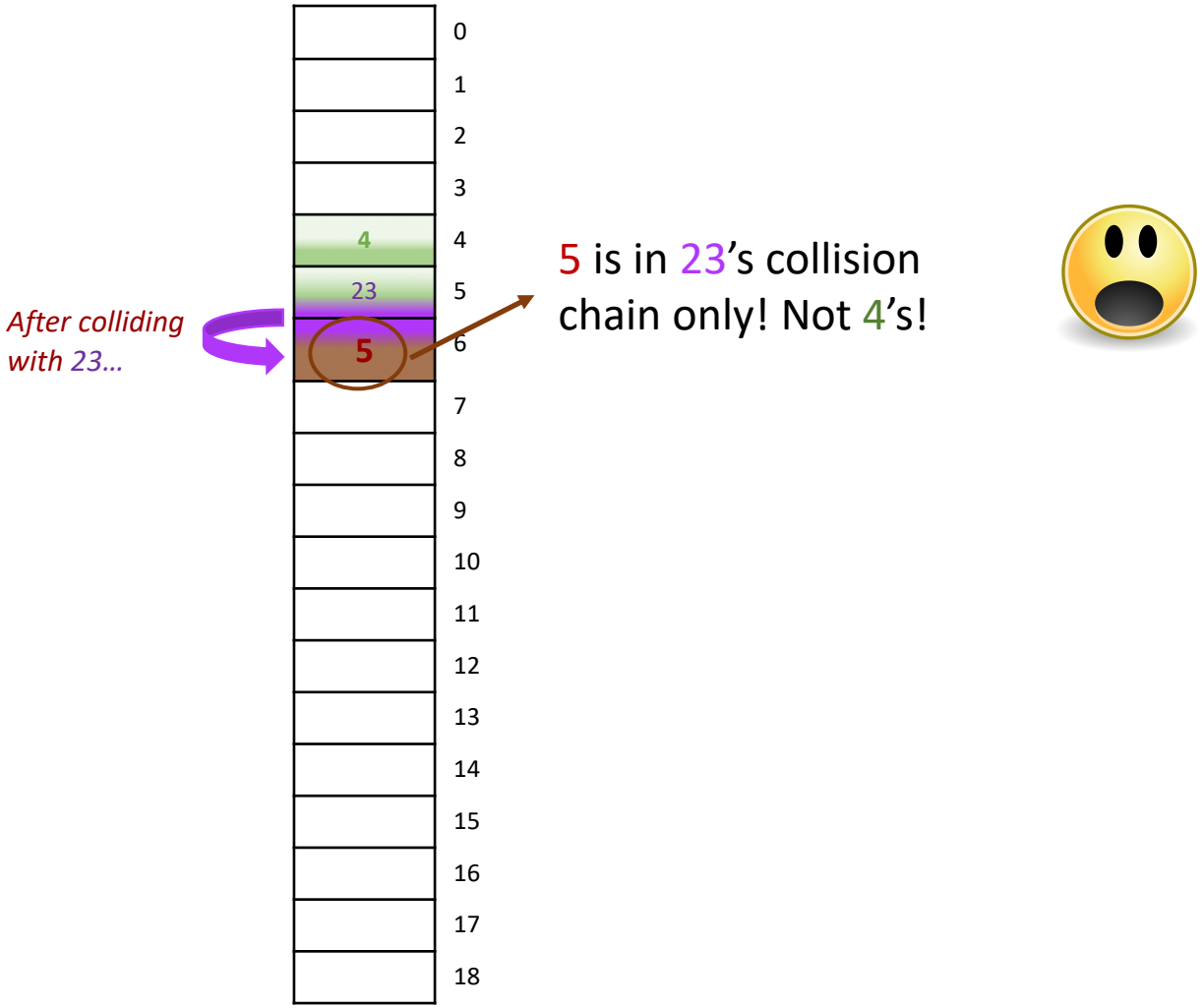
- Insert 4, 23, 5



$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5

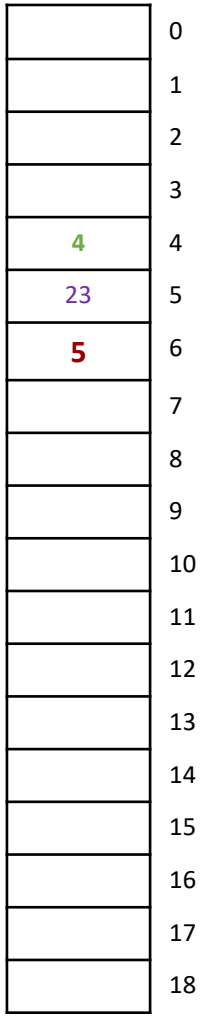


$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5

- Insert 19, 38, 0



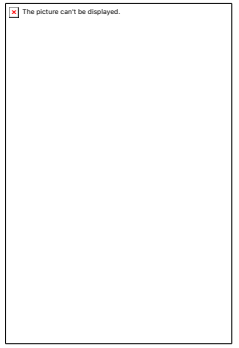
$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0

19	0
38	1
0	2
	3
4	4
23	5
5	6
	7
	8
	9
	10
	11
	12
	13
	14
	15
	16
	17
	18

Which collision chains do these keys belong to?



$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0

Since **38** collides with **19**, it’s in **19**’s collision chain.

Since **0** collides with both **19** and **38** (after one forward step), **it is in both collision chains!**

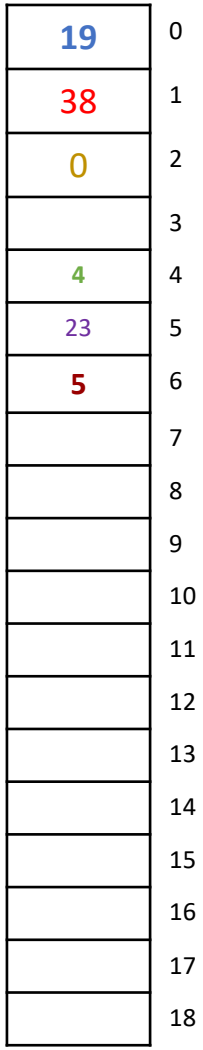
19	0
38	1
0	2
	3
4	4
23	5
5	6
	7
	8
	9
	10
	11
	12
	13
	14
	15
	16
	17
	18

Which collision chains do these keys belong to?

$h(n) = n \,\%\, M$

“Clusters” and “chains”

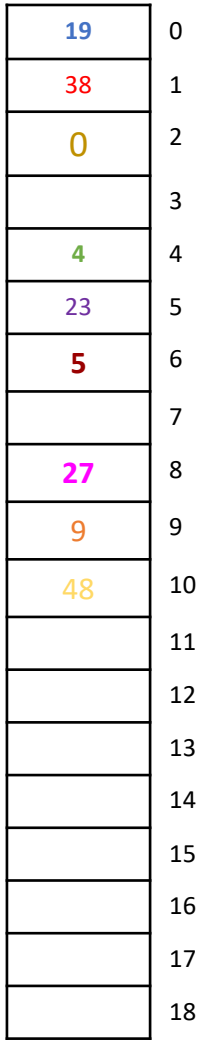
- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48



$h(n) = n \,\%\, M$

“Clusters” and “chains”

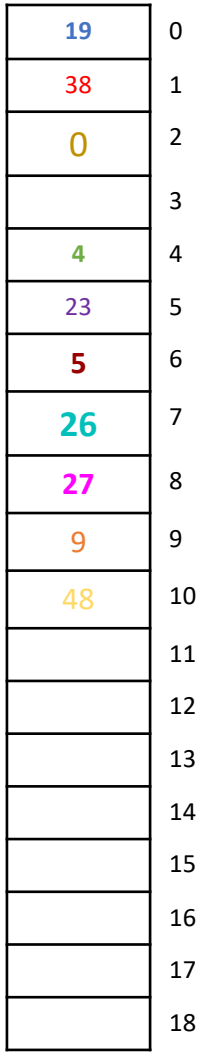
- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26



$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26



$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26

19	0
38	1
0	2
	3
4	4
23	5
5	6
26	7
27	8
9	9
48	10
	11
	12
	13
	14
	15
	16
	17
	18

all those different collision chains have merged into a **cluster of keys**!

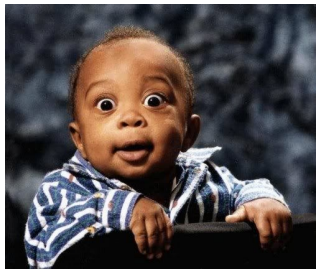
$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26

19	0
38	1
0	2
	3
4	4
23	5
5	6
26	7
27	8
9	9
48	10
	11
	12
	13
	14
	15
	16
	17
	18

all those different collision chains have merged into a **cluster of keys**!



AND IT GETS WORSE!

$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26
- Insert **60**

$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26
- Insert **60**

19	0
38	1
0	2
60	3
4	4
23	5
5	6
26	7
27	8
9	9
48	10
	11
	12
	13
	14
	15
	16
	17
	18

*Inserting 60 led us to a **“mega-cluster”** where all the different collision chains are intermixed!*

$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26
- Insert **60**

Assuming uniformly distributed input, what’s the probability that the next insertion will **enlarge** this cluster?

$\frac{1}{19}$

$\frac{2}{19}$

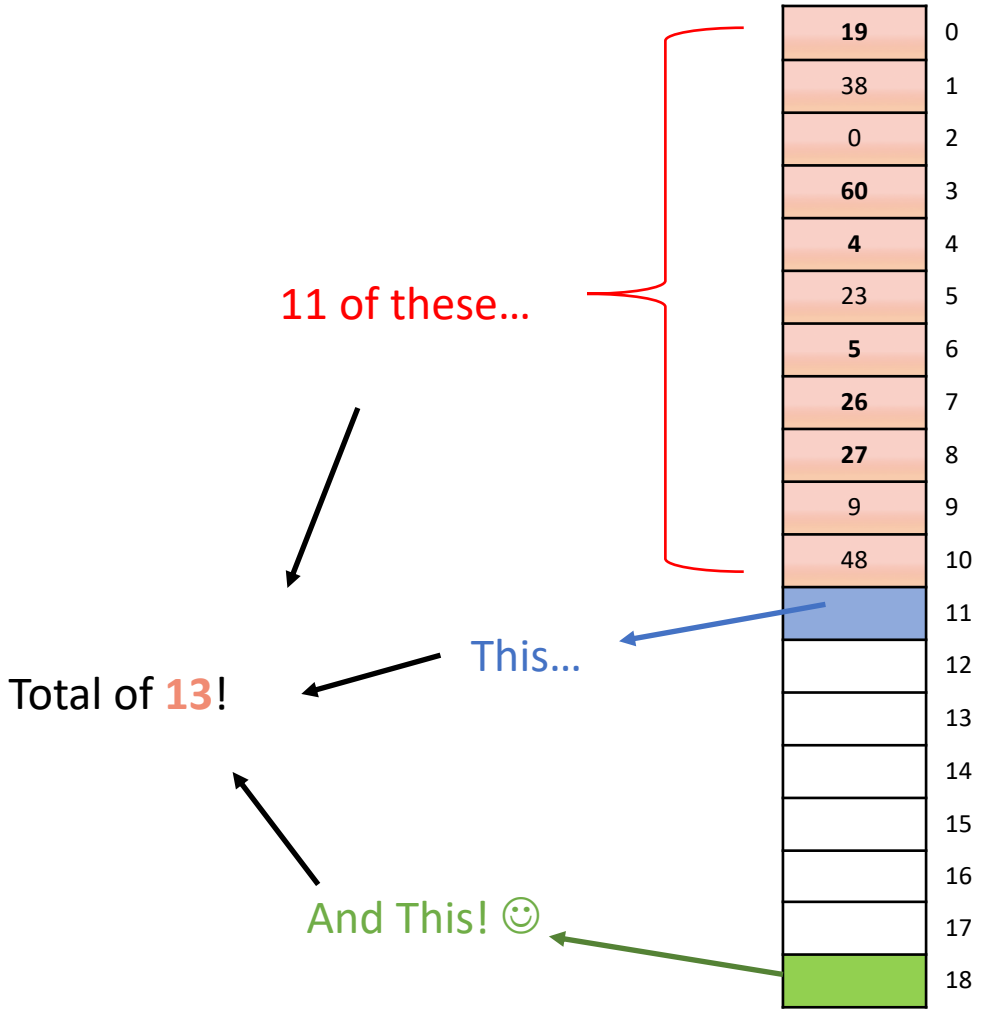
$\frac{12}{19}$

Something else (what?)

$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26
- Insert **60**



Assuming uniformly distributed input, what’s the probability that the next insertion will **enlarge** this cluster?

$\frac{1}{19}$

$\frac{2}{19}$

$\frac{12}{19}$

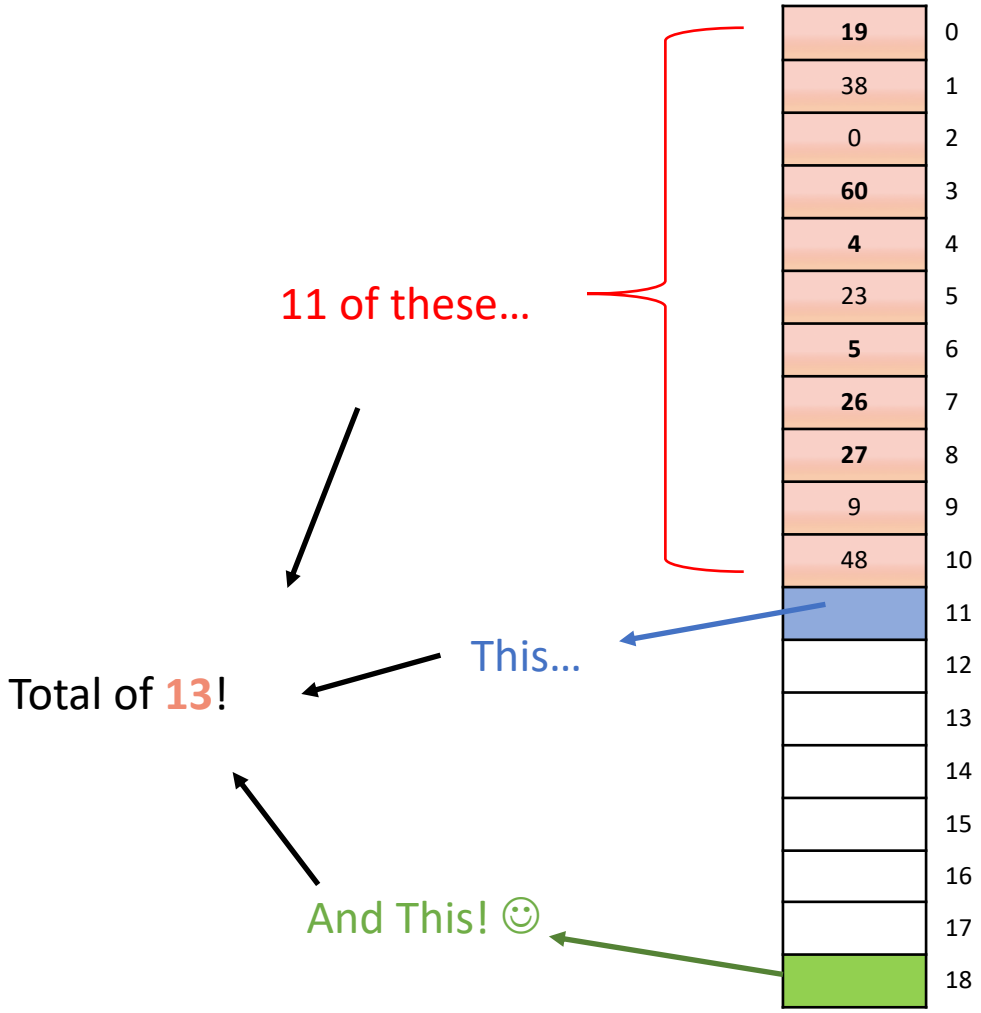
Something else (what?)

$\frac{13}{19}$

$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26
- Insert **60**



Statement: If instead of adding 1 every time we added something else (e.g 3) we **would avoid clusters**

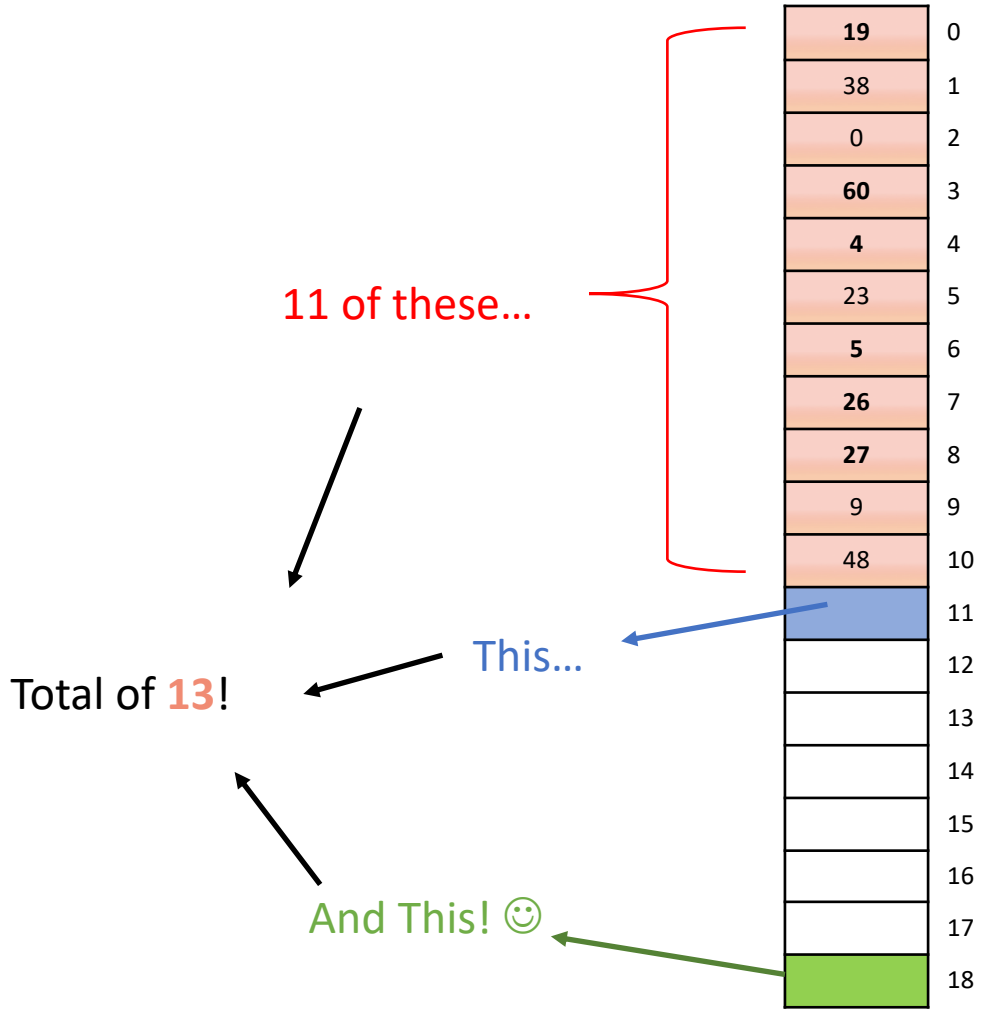
True

False

$h(n) = n \,\%\, M$

“Clusters” and “chains”

- Insert 4, 23, 5
- Insert 19, 38, 0
- Insert 27, 9, 48
- Insert 26
- Insert **60**



Statement: If instead of adding 1 every time we added something else (e.g 3) we **would avoid clusters**

True

False

The “clusters” then end up just being “discontinuous” in the array! :O

Only solution: Resize the array, reinserting all elements afterwards ☹

Take-home message

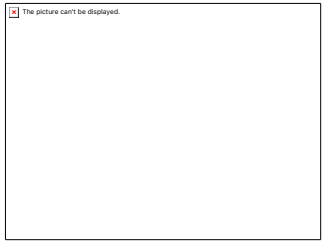
- As the table progressively gets fuller, collision chains form key clusters.
- The probability that those clusters merge into ***mega-clusters*** increases with every insertion that occupies spaces between the clusters… ☹

Take-home message

- As the table progressively gets fuller, collision chains form key clusters.
- The probability that those clusters merge into ***mega-clusters*** increases with every insertion that occupies spaces between the clusters… ☹
- Problem compounded if we remove the uniform hashing assumption.
- Solution offered by linear probing: Resize the table (e.g through our previous approach), re-hash and re-insert all keys.

Take-home message

- As the table progressively gets fuller, collision chains form key clusters.
- The probability that those clusters merge into ***mega-clusters*** increases with every insertion that occupies spaces between the clusters… ☹
- Problem compounded if we remove the uniform hashing assumption.
- Solution offered by linear probing: Resize the table (e.g through our previous approach), re-hash and re-insert all keys.
- *At which value for the load factor α should we resize?*



Knuth's seminal result

- In a linear-probing hash table with M buckets and $n = \alpha \cdot M$ keys to store, the average number of probes required for a search hit is:

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

- Whereas for a search miss it is:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Knuth's seminal result

$a \in (0, 1)$

- In a linear-probing hash table with M buckets and $n = \alpha \cdot M$ keys to store, the average number of probes required for a search hit is:

Psst! Perfectly uniform hash!

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

- Whereas for a search miss it is:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

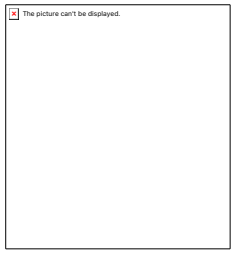
Knuth's seminal result

$\alpha \in (0, 1)$

- In a linear-probing hash table with M buckets and $n = \alpha \cdot M$ keys to store, the average number of probes required for a search hit is:

Psst! Perfectly uniform hash!

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$



Let's consider what happens as $\alpha \to 1$ or $\alpha \to 0$!

- Whereas for a search miss it is:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

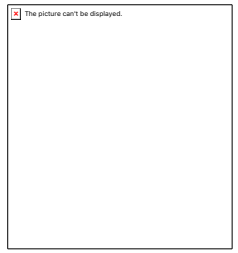
Knuth's seminal result

$\alpha \in (0, 1)$

- In a linear-probing hash table with M buckets and $N = \alpha \cdot M$ keys to store, the average number of probes required for a search hit is:

Psst! Perfectly uniform hash!

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$



Let's consider what happens as $\alpha \to 1$ or $\alpha \to 0$!

- Whereas for a search miss it is:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

- **Larger than the avg #probes for a hit, since $(1-\alpha) \in (0,1) \Rightarrow (1-\alpha)^2 < (1-\alpha)$**
- **Linear Probing favors search hits!**

Search

- Search in a Linear Probing hash table consists of first hashing the key to be searched.
- Then, we loop through the buffer until:
  1. We find the key (search hit) ☺, **OR**
  2. We encounter an empty cell (search miss) ☹

Search

- Search in a Linear Probing hash table consists of first hashing the key to be searched.
- Then, we loop through the buffer until:
  1. We find the key (search hit) ☺, **OR**
  2. We encounter an empty cell (search miss) ☹
- As previously mentioned, linear probing favors search hits over search misses.
  - Is this reasonable? *What can we expect in practice?*

Search

- Search in a Linear Probing hash table consists of first hashing the key to be searched.
- Then, we loop through the buffer until:
  1. We find the key (search hit) ☺, **OR**
  2. We encounter an empty cell (search miss) ☹
- As previously mentioned, linear probing favors search hits over search misses.
  - Is this reasonable? *What can we expect in practice?*

More search hits

More search misses

About the same number of hits and misses

Search

- Search in a Linear Probing hash table consists of first hashing the key to be searched.
- Then, we loop through the buffer until:
  1. We find the key (search hit) ☺, **OR**
  2. We encounter an empty cell (search miss) ☹
- As previously mentioned, linear probing favors search hits over search misses.
  - Is this reasonable? *What can we expect in practice?*

MANY MORE SEARCH HITS!

More search hits

More search misses

About the same number of hits and misses

Recall: **Hashes are key-value stores**. The V in $< K, V >$ can be very complex (entire student record), and can undergo **many mutations** *(new fields for the student, dropping redundant fields, updating fields, etc)* **Every mutation after the first one will occur after a search hit for K!**

Deletion

- Toughest operation!

Deletion

- Toughest operation!
- "Hard" deletion: by setting reference to `null`, you break the path to other keys ☹
  - Only solution: Loop through rest of cluster, nullifying references but simultaneously re-inserting the key in the table

Deletion

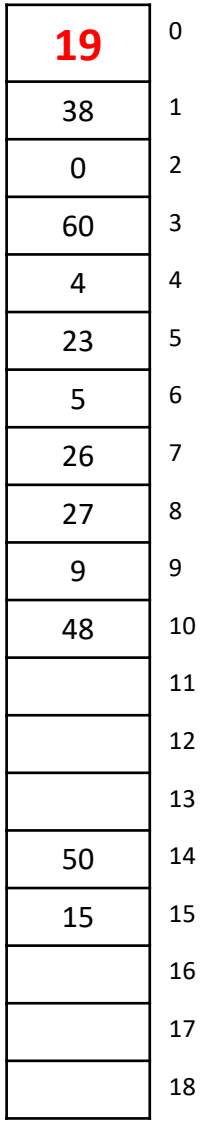
- Toughest operation!
- "Hard" deletion: by setting reference to `null`, you break the path to other keys ☹
  - Only solution: Loop through rest of cluster, nullifying references but simultaneously re-inserting the key in the table.
- "Soft" deletion: Tombstones!
  - A special value which contributes to the current size (key count) of the table, but is ***available*** if a value wants the cell for an insertion!

Deletion

- Toughest operation!
- "Hard" deletion: by setting reference to `null`, you break the path to other keys ☹
  - Only solution: Loop through rest of cluster, nullifying references but simultaneously re-inserting the key in the table.
- "Soft" deletion: Tombstones!
  - A special value which contributes to the current size (key count) of the table, but is ***available*** if a value wants the cell for an insertion!
- In your project, you will implement **both** kinds of deletion.

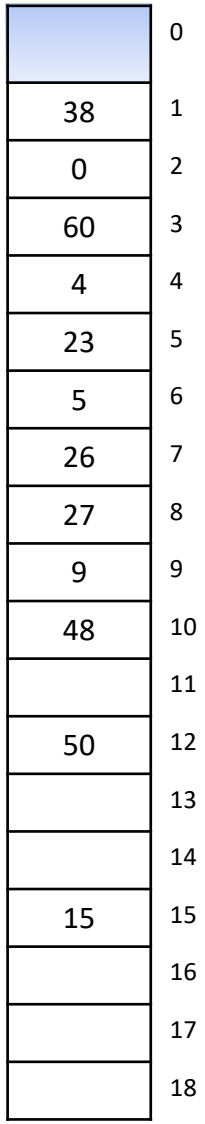
Deletion exercise!

- Let's try to delete 19….



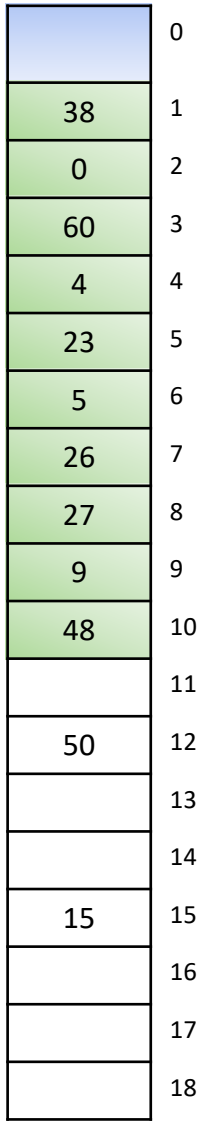
Deletion exercise!

- Let's try to delete 19….
  1. Erase it from the table *(nullify reference if key is an `Object`)*



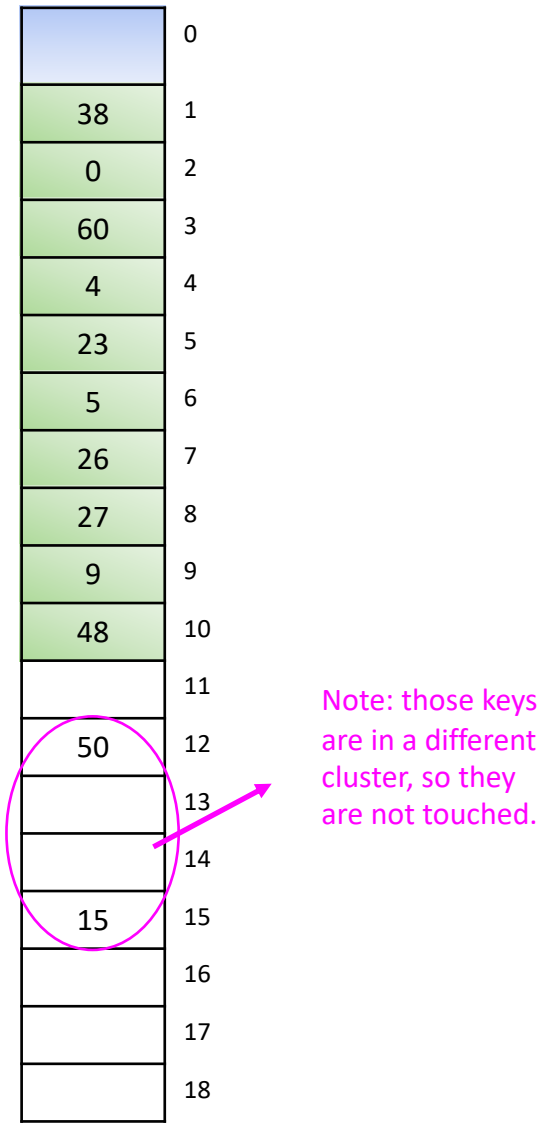
Deletion exercise!

- Let's try to delete 19….
  1. Erase it from the table *(nullify reference if key is an `Object`)*
  2. Loop through the rest of the cluster, erase and re-insert all keys :O



Deletion exercise!

- Let's try to delete 19….
  1. Erase it from the table *(nullify reference if key is an `Object`)*
  2. Loop through the rest of the cluster, erase and re-insert all keys :O



Deletion exercise!

- Let's try to delete 19….
  1. Erase it from the table *(nullify reference if key is an `Object`)*
  2. Loop through the rest of the cluster, erase and re-insert all keys :O
  3. Let's do this exercise together ☺