CS2040S
Data Structures and Algorithms

Hashing!
(Part 3)

Today

- Collision resolution: open addressing
  - Linear Probing: Insert, Lookup, Delete

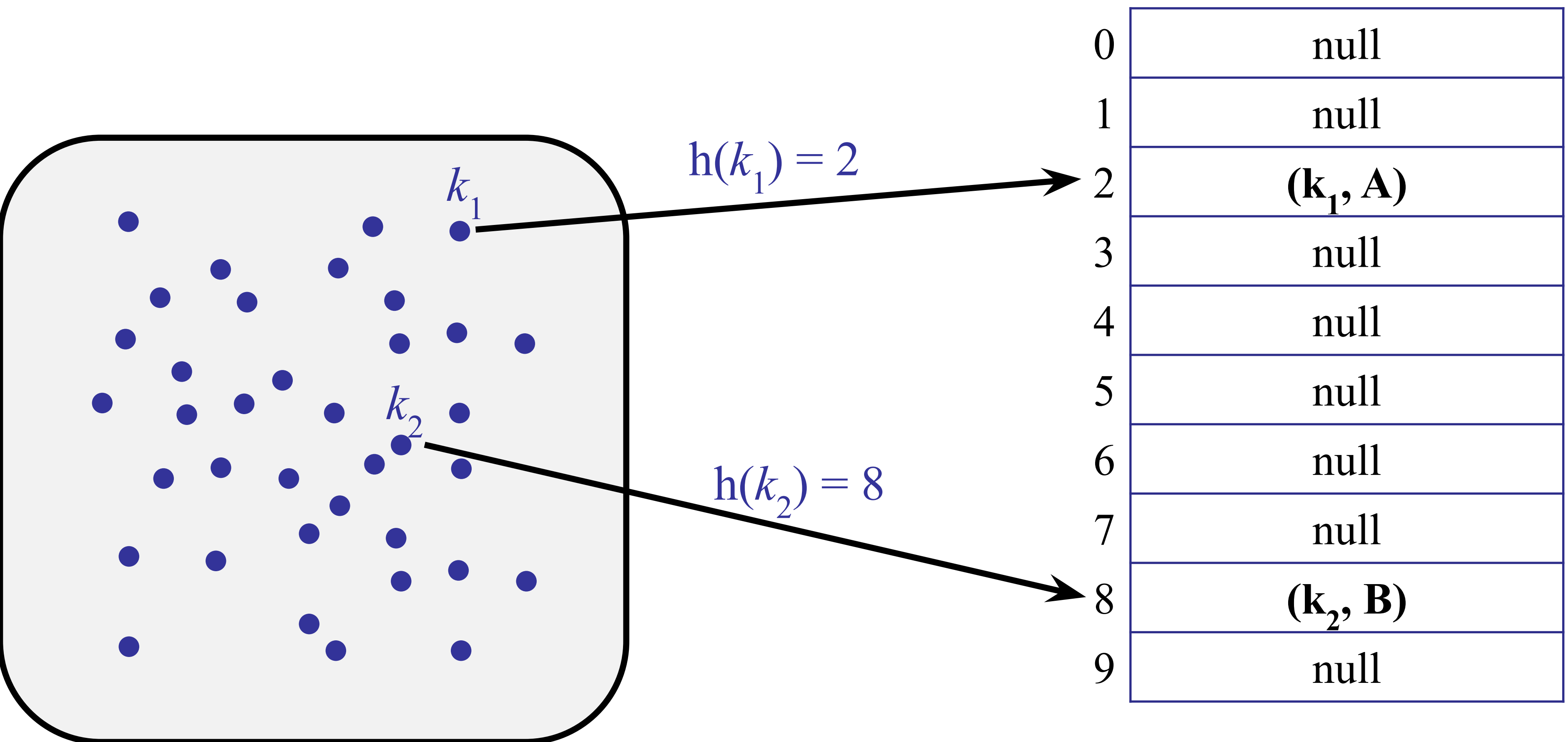
- Table (re)sizing
  - Amortisation

- Other resolutions

Review

Hash Tables

- Store each item from the symbol table in a table.
- Use hash function to map each key to a bucket.



Resolving Collisions

- Basic problem:
  - What to do when two items hash to the same bucket?
- Previously: Chaining
  - Insert item into a linked list.
- Today: Open Addressing
  - Items are inserted into the table directly

Open Addressing

Advantages:

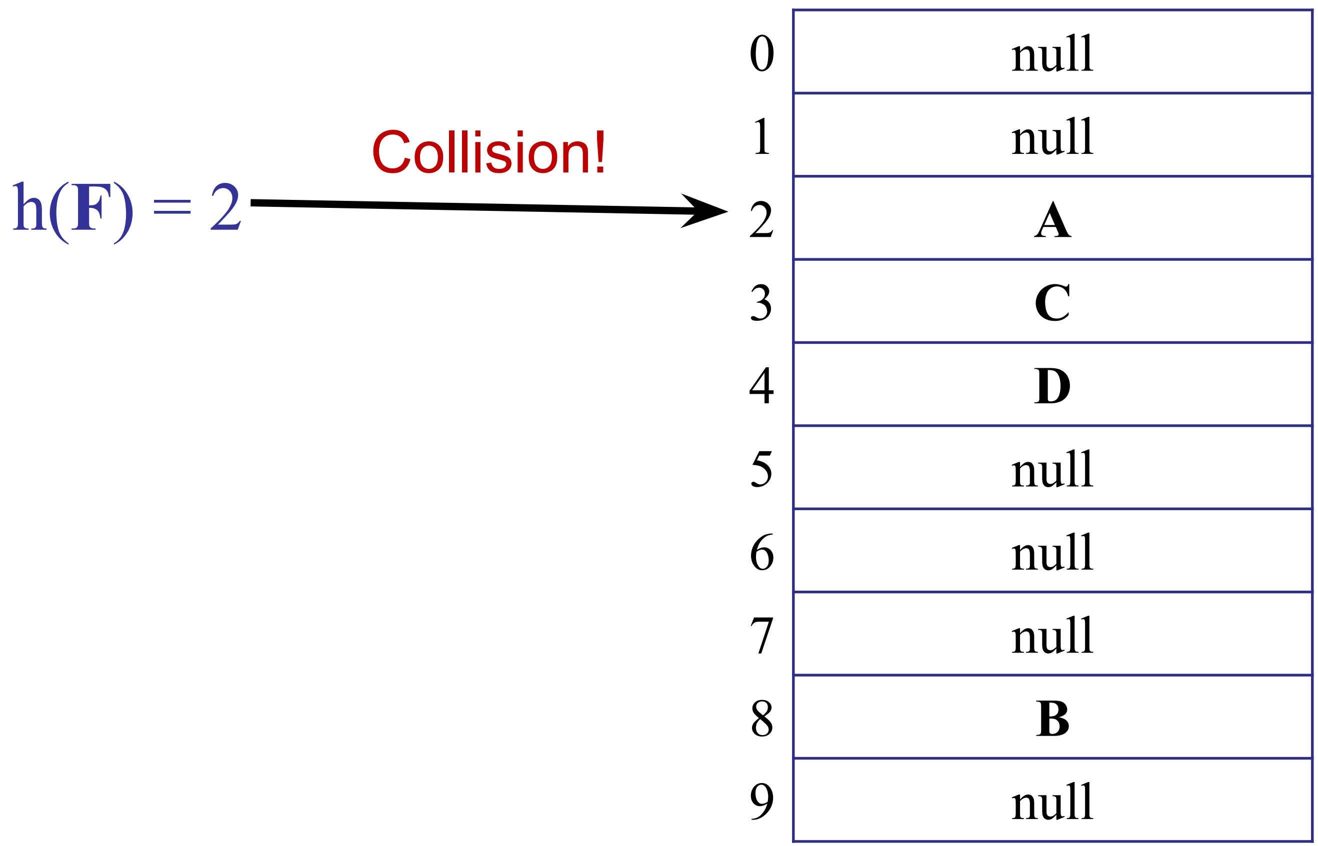
- No linked lists!
- All data directly stored in the table.
- One item per slot.

0	null
1	null
2	**A**
3	null
4	null
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

On collision:

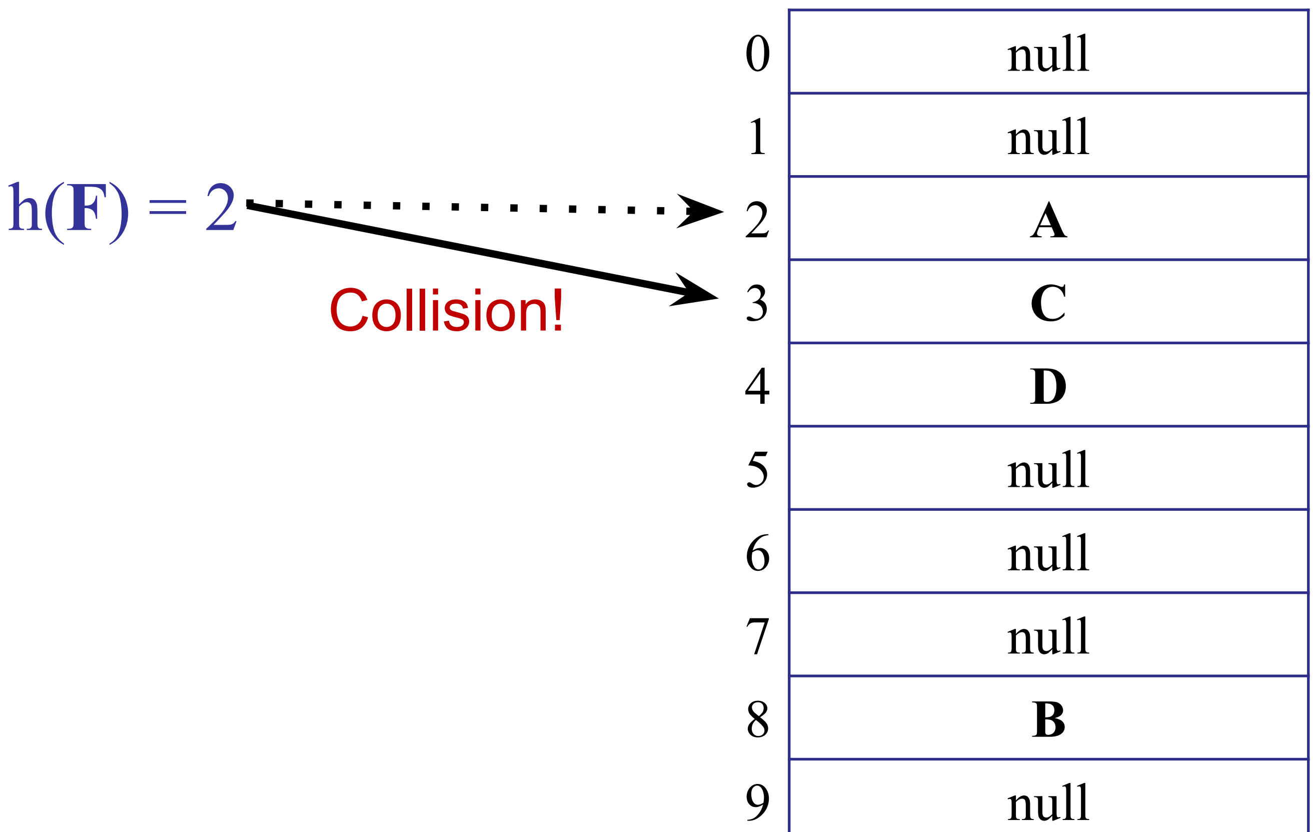
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

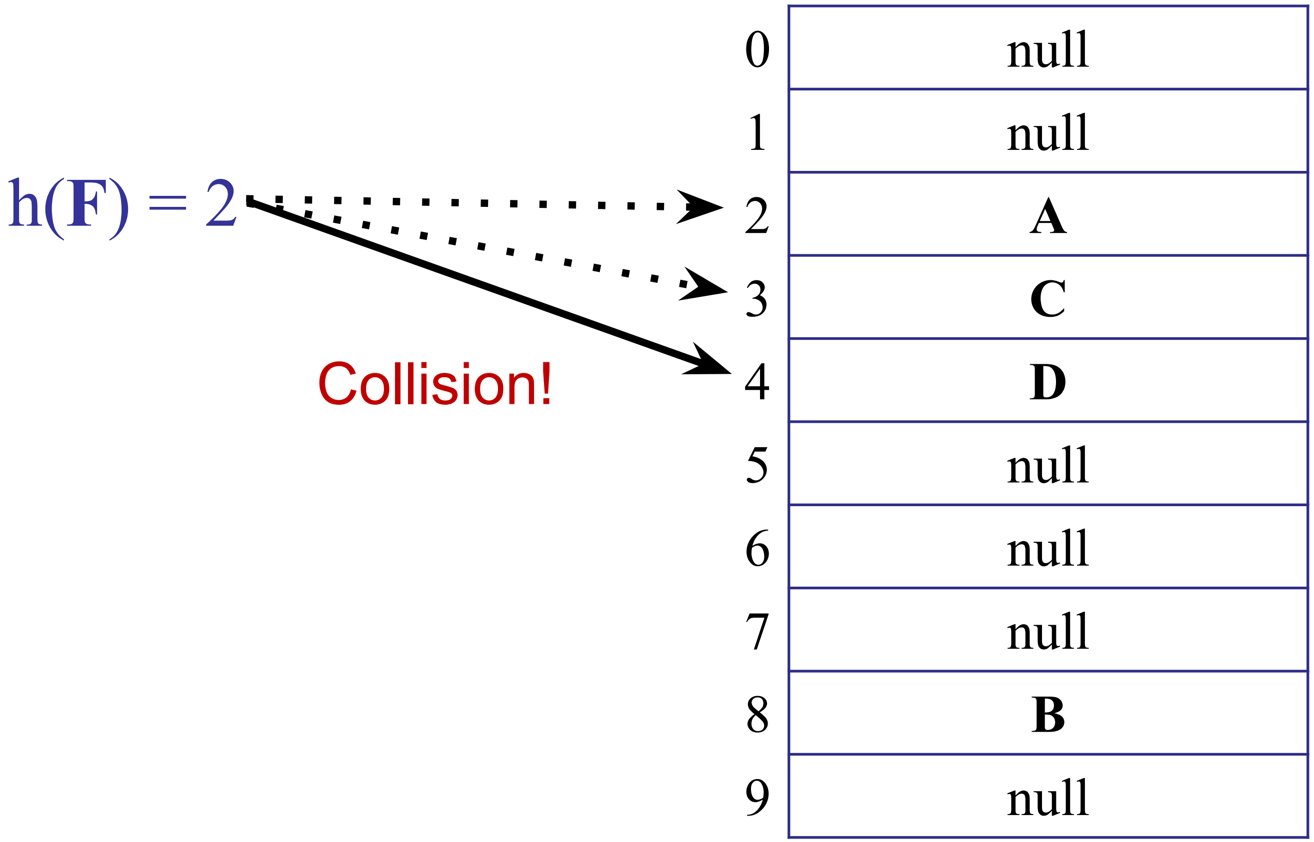
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

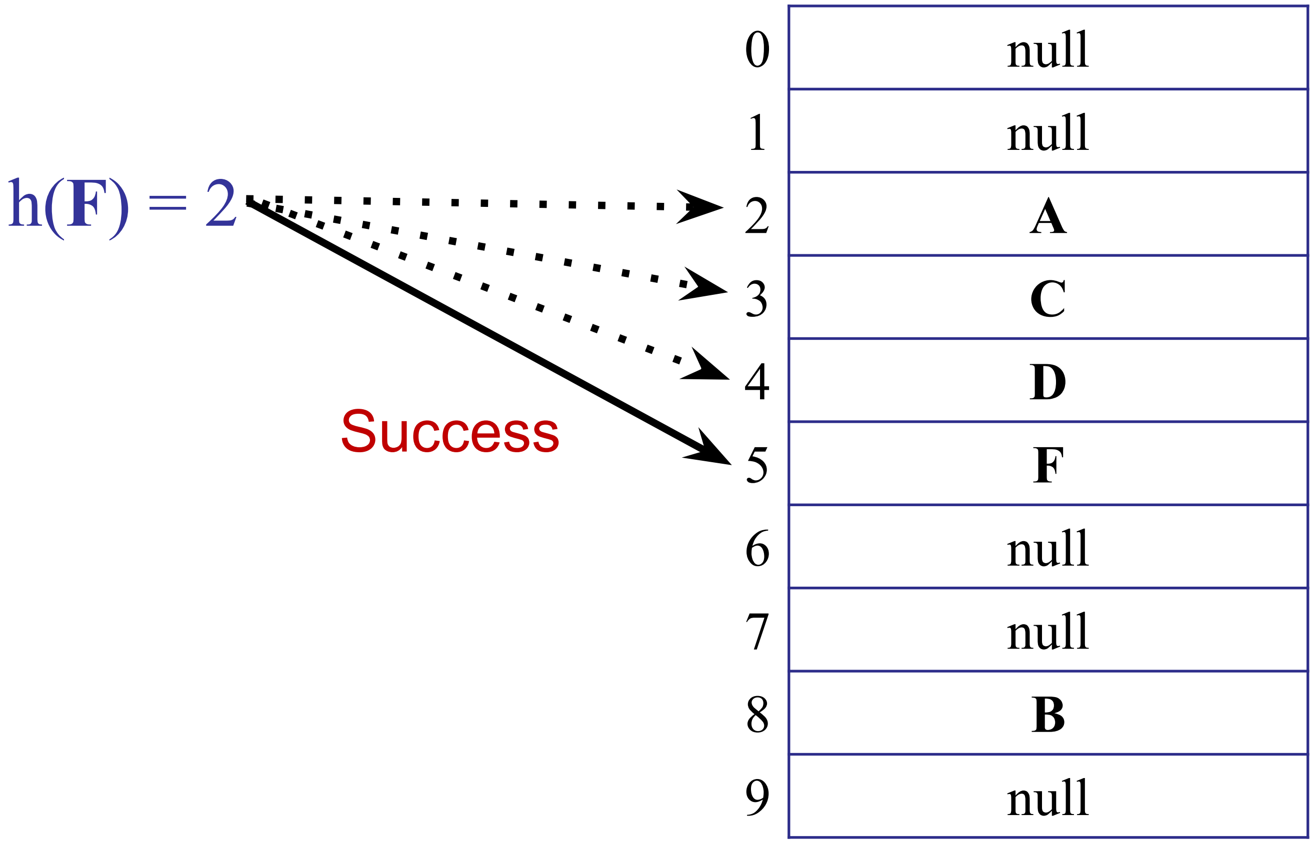
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

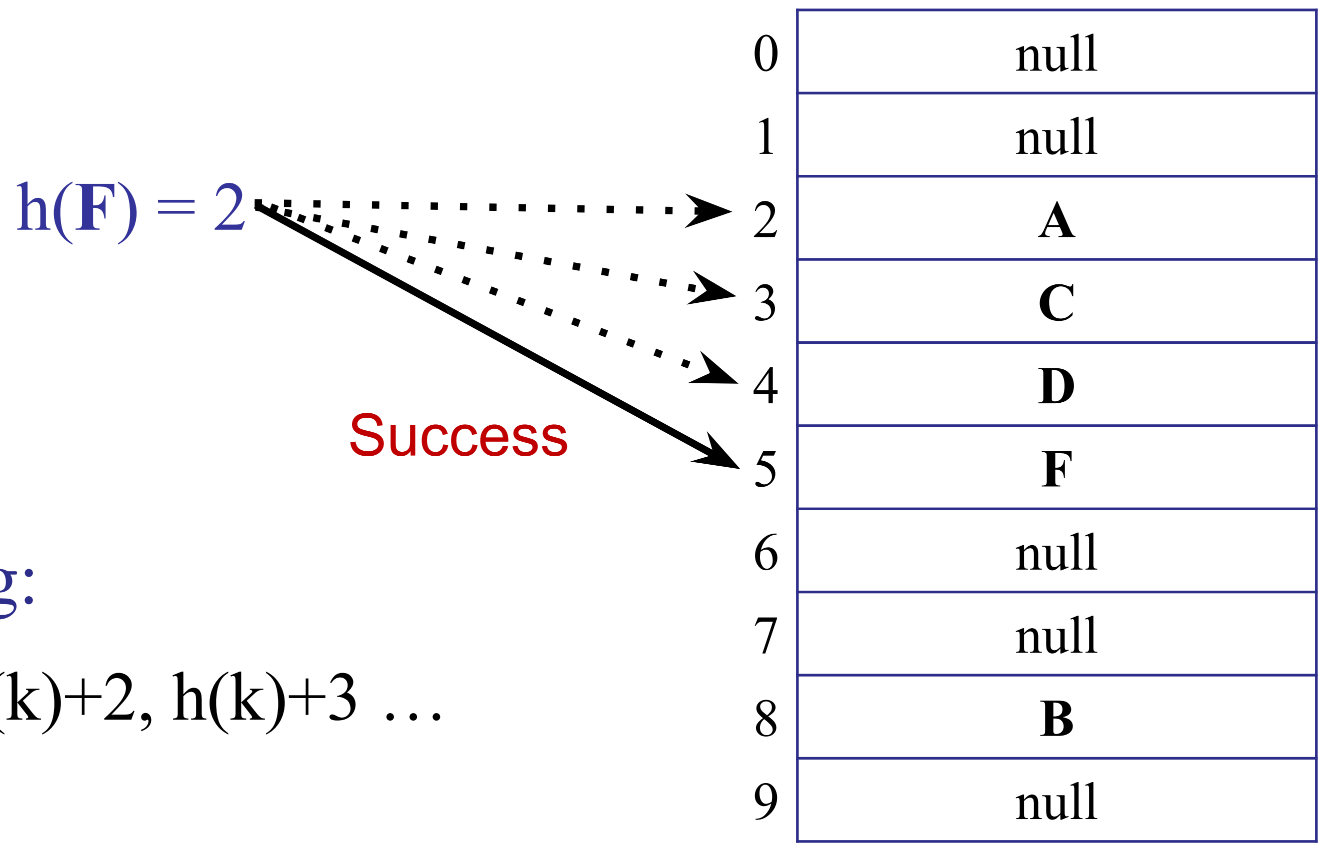
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

Probe a sequence of buckets until you find an empty one.



Linear Probing:

- h(k)+1, h(k)+2, h(k)+3 …

Open Addressing

Example: Inserting an element **F**, first compute

h(**F**)

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

Example: Inserting an element **F**, first compute

$h(\mathbf{F})=3$

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

Example: Inserting an element **F**, first compute

h(**F**)=3

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

Example: Inserting an element **F**, first compute

h(**F**)=3

Try the next location until we find an empty slot

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

Example: Inserting an element **F**, first compute

h(**F**)=3

Try the next location until we find an empty slot

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

Example: Inserting an element **F**, first compute

h(**F**)=3

Try the next location until we find an empty slot

Found one!

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

Example: Inserting an element **F**, first compute

h(**F**)=3

Try the next location until we find an empty slot

Found one!

Insert our item there

0	null
1	null
2	**A**
3	**C**
4	**D**
5	**F**
6	null
7	null
8	**B**
9	null

Open Addressing

```
void insert(K key, V val){
  int m = table.size();
  int hashCode = key.hashCode();
  int probe = 0;

  while(table[(hashCode + probe) % m] != null){
    probe += 1;
  }

  table[(hashcode + probe) % m] = new Pair<K, V>(key, val);
}
```

Is this implementation of insert correct?

1. Yes
2. No

Is this implementation of insert correct?

1. Yes
2. No

What happens if the table is full?

i.e. m = n

Is this implementation of insert correct?

1. Yes
2. No

What happens if the table is full?

i.e. $m = n$

We will have to fix this later.

Open Addressing

When the table is full, we will just keep iterating!

0	**Z**
1	**E**
2	**A**
3	**C**
4	**D**
5	**F**
6	**P**
7	**J**
8	**B**
9	**R**

Assuming n < m, is this implementation correct?

1. Yes
2. No

Open Addressing

What about searching? How do we search?

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about searching? How do we search?

E.g. looking for item **D**

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about searching? How do we search?

E.g. looking for item **D**

Obtain hash h(**D**) = 2

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about searching? How do we search?

E.g. looking for item **D**

Obtain hash h(**D**) = 2

Similarly to as before, start at the hash location, start probing.

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

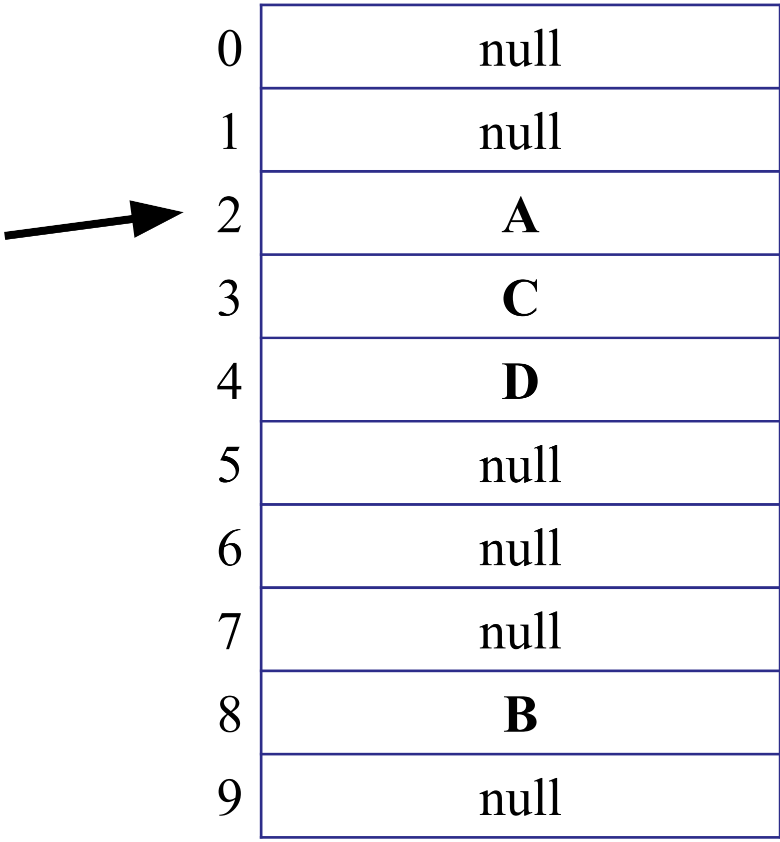
Open Addressing

What about searching? How do we search?

E.g. looking for item **D**

Obtain hash h(**D**) = 2

Similarly to as before, start at the hash location, start probing.



Open Addressing

What about searching? How do we search?

E.g. looking for item **D**

Obtain hash h(**D**) = 2

Similarly to as before, start at the hash location, start probing.

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

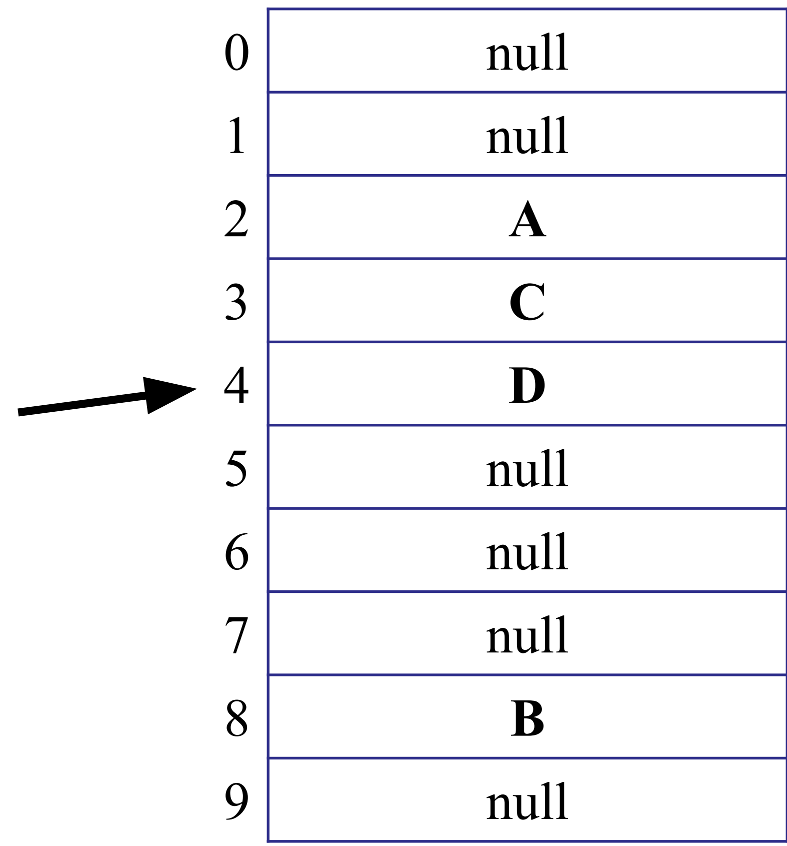
What about searching? How do we search?

E.g. looking for item **D**

Obtain hash h(**D**) = 2

Similarly to as before, start at the hash location, start probing.

Stop when we've found the item.



Open Addressing

What about searching? How do we search?

E.g. looking for item **F**

Obtain hash h(**F**) = 8

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about searching? How do we search?

E.g. looking for item **F**

Obtain hash h(**F**) = 8

What happens if we see *null*?

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about searching? How do we search?

E.g. looking for item **F**

Obtain hash h(**F**) = 8

What happens if we see *null*?

The item is not present.

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

```
V get(Object key){
  int m = table.size();
  int hashCode = key.hashCode();
  int probe = 0;

  while(probe < m && table[(hashCode + probe) % m] != null){
    if(table[(hashcode + probe) % m].getKey().equals(k)){
      return table[(hashcode + probe) % m].getValue();
    }
    probe += 1;
  }
  return null;
}
```

Open Addressing

In case table is full and does not contain the element we want

```
V get(Object key){
  int m = table.size();
  int hashCode = key.hashCode();
  int probe = 0;

  while(probe < m && table[(hashCode + probe) % m] != null){
    if(table[(hashcode + probe) % m].getKey().equals(k)){
      return table[(hashcode + probe) % m].getValue();
    }
    probe += 1;
  }
  return null;
}
```

Is this implementation correct?

1. Yes
2. No

Open Addressing

What about deletions?

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about deletions?

E.g. deleting **C**

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about deletions?

E.g. deleting **C** $\qquad h(\mathbf{C}) = 2$

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about deletions?

E.g. deleting **C**

$h(\mathbf{C}) = 2$

compute $h(\mathbf{C})$

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about deletions?

E.g. deleting **C**

$h(\mathbf{C}) = 2$

compute $h(\mathbf{C})$

find $h(\mathbf{C})$

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about deletions?

E.g. deleting **C**

$h(\mathbf{C}) = 2$

compute $h(\mathbf{C})$

find $h(\mathbf{C})$

0	null
1	null
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

What about deletions?

E.g. deleting **C**

compute h(**C**)

find h(**C**)

Set found position to *null*

0	null
1	null
2	**A**
3	null
4	**D**
5	null
6	null
7	null
8	**B**
9	null

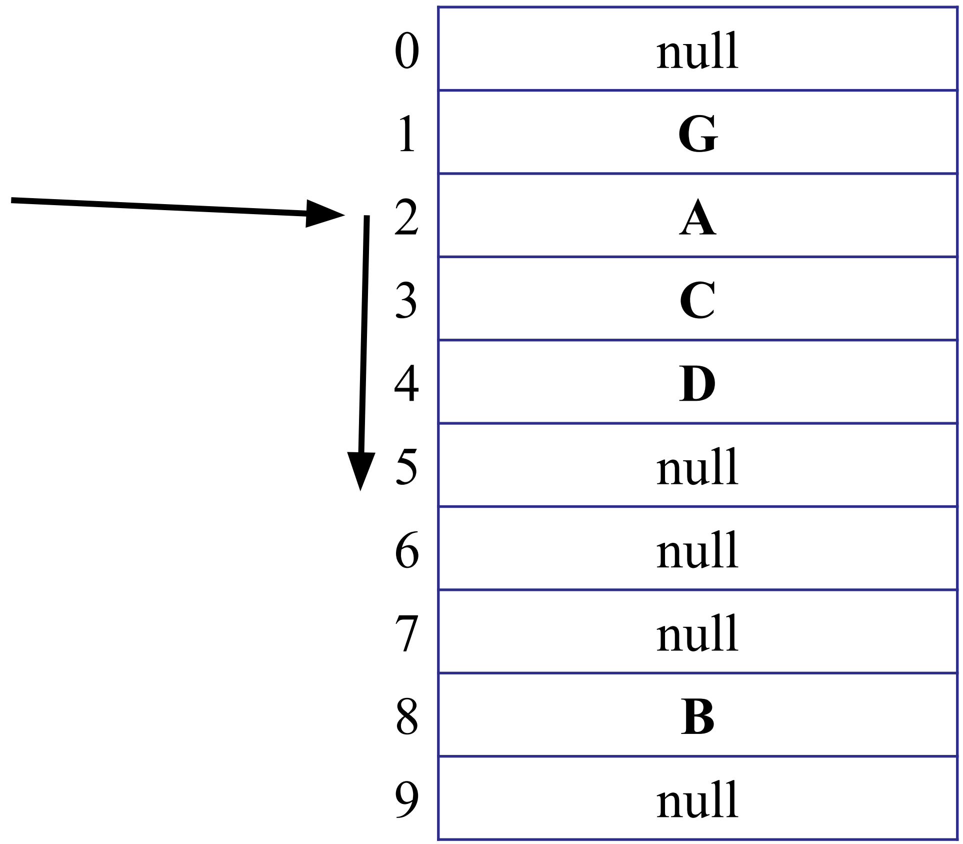
Is delete correct?

1. Yes
2. No

Open Addressing

insert(key)

$h(key) = 2$



Open Addressing

insert(key)

$h(key) = 2$

0	null
1	**G**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

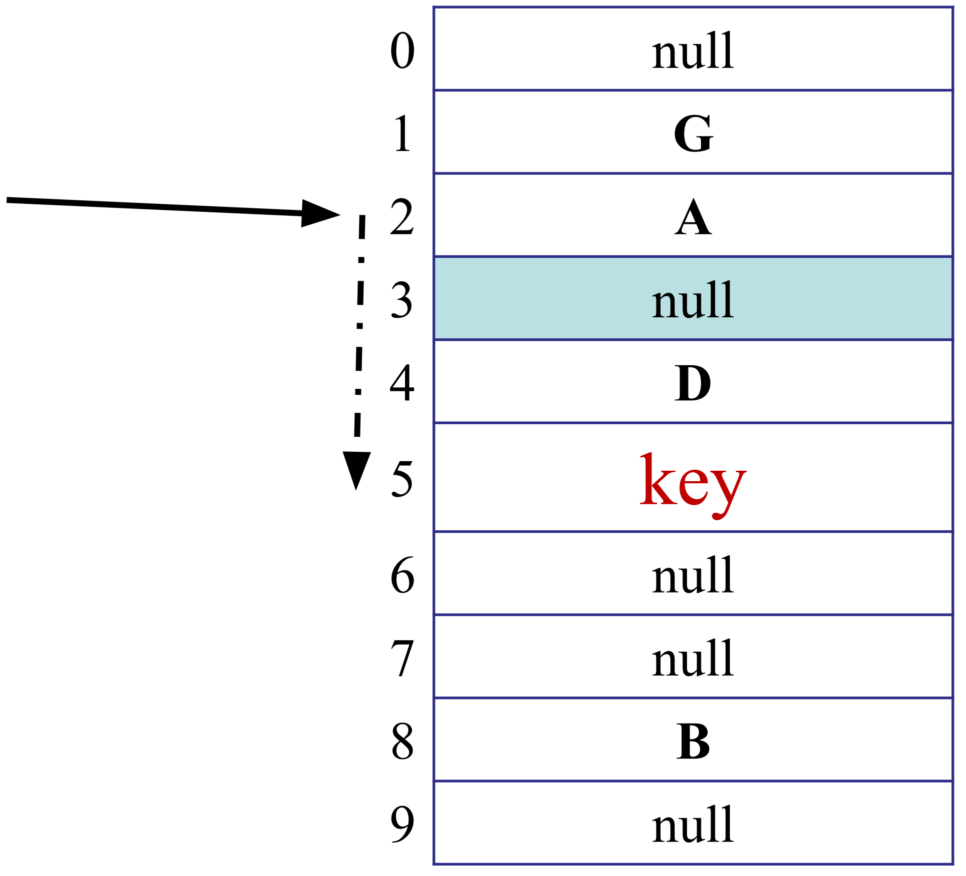
delete(C)

0	null
1	**G**
2	**A**
3	null
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

insert(key)

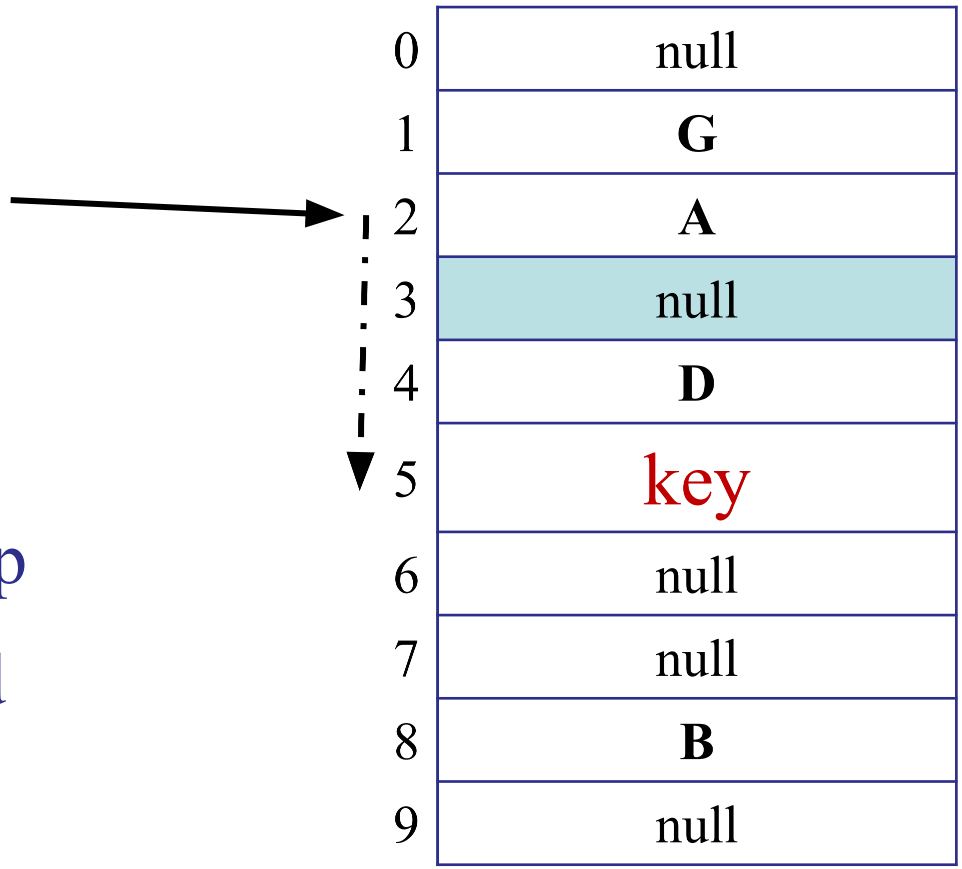
search(key)



Open Addressing

insert(key)

search(key)



We would stop at the null and not continue

Open Addressing

There are a few ways to handle deletions:

- Just probe the entire table during search
    - Not ideal: This makes search crazy expensive
- Tombstoning
    - A quick fix - but need to handle case where too many items are deleted
- Replace it with another element further down
    - Most complicated, but better use of table space

Open Addressing

There are a few ways to handle deletions:

- Just probe the entire table
  - Not ideal: This makes deletion crazy expensive
- Tombstoning
  - A quick fix - but need to handle case where too many items are deleted
- Replace it with another element further down
  - Most complicated, but better use of table space

Open Addressing

Open Addressing

0	null
1	**G**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

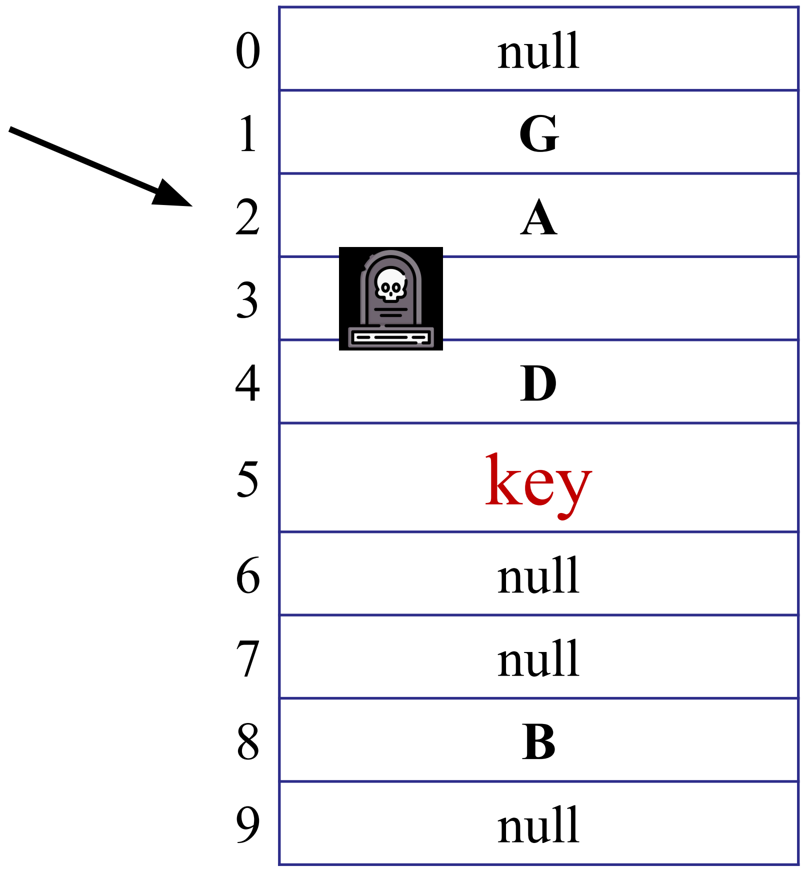
delete(C)

0	null
1	**G**
2	**A**
3	
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

search(key)

h(key) = 2



Open Addressing

search(key)

h(key) = 2

0	null
1	**G**
2	**A**
3	
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

search(k

h(key) =

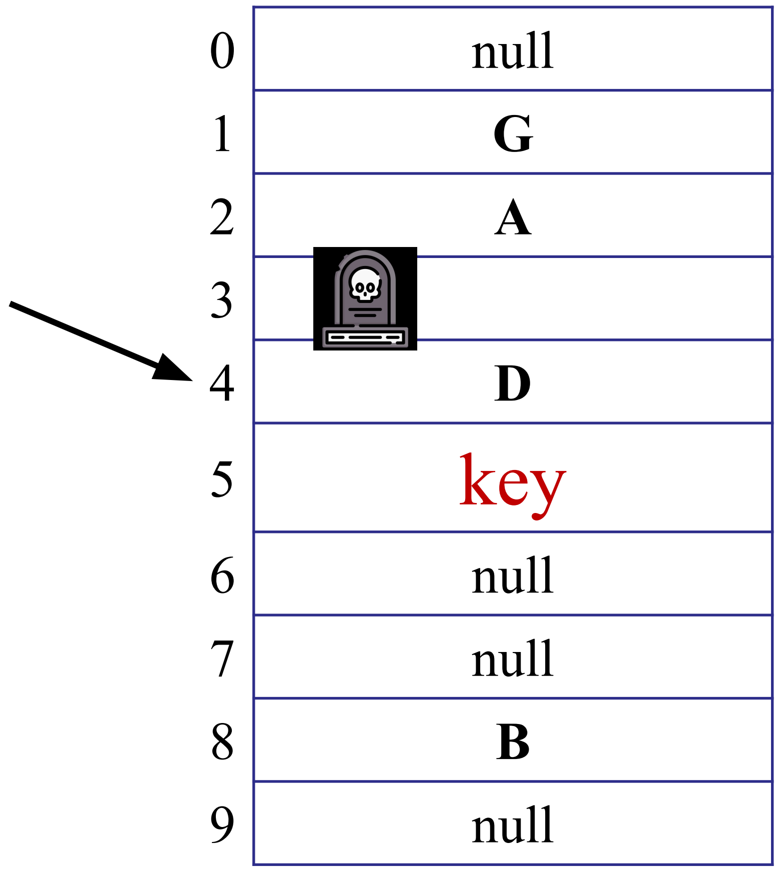
hey watch where
you’re stepping!

0	null
1	**G**
2	**A**
3	
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

search(key)

h(key) = 2



Open Addressing

search(k

kids these days have no respect

h(key) =

0	null
1	**G**
2	**A**
3	
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

search(key)

h(key) = 2

0	null
1	**G**
2	**A**
3	
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

search(key)

h(key) = 2

Now if we ever reach null
we know that the element
doesn't exist in the table

0	null
1	**G**
2	**A**
3	
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

Question 1:

What happens

if as we insert,

we find a tombstone?

0	null
1	**G**
2	**A**
3	
4	**D**
5	key
6	
7	null
8	**B**
9	

Open Addressing

Question 1:

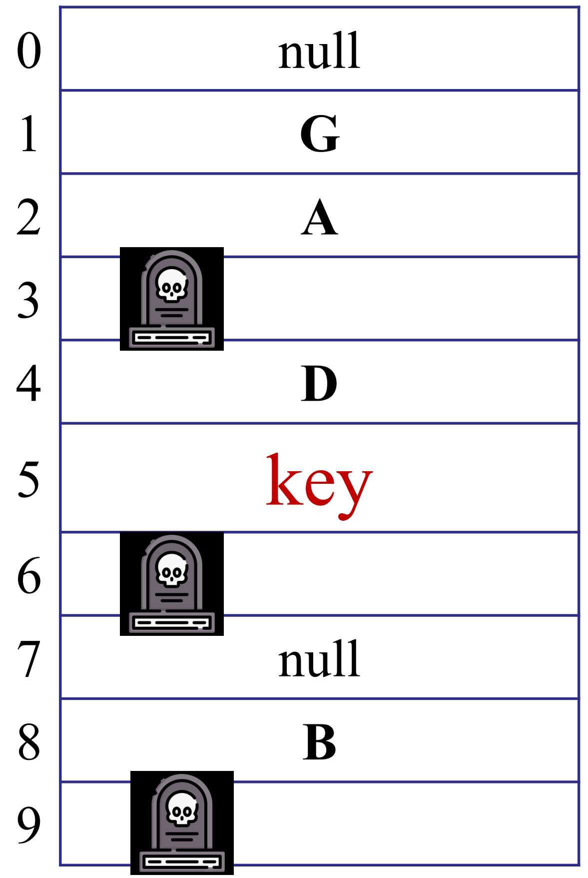
What happens

if as we insert,

we find a tombstone?

E.g. insert(**E**)

h(**E**) = 2



Open Addressing

Question 1:

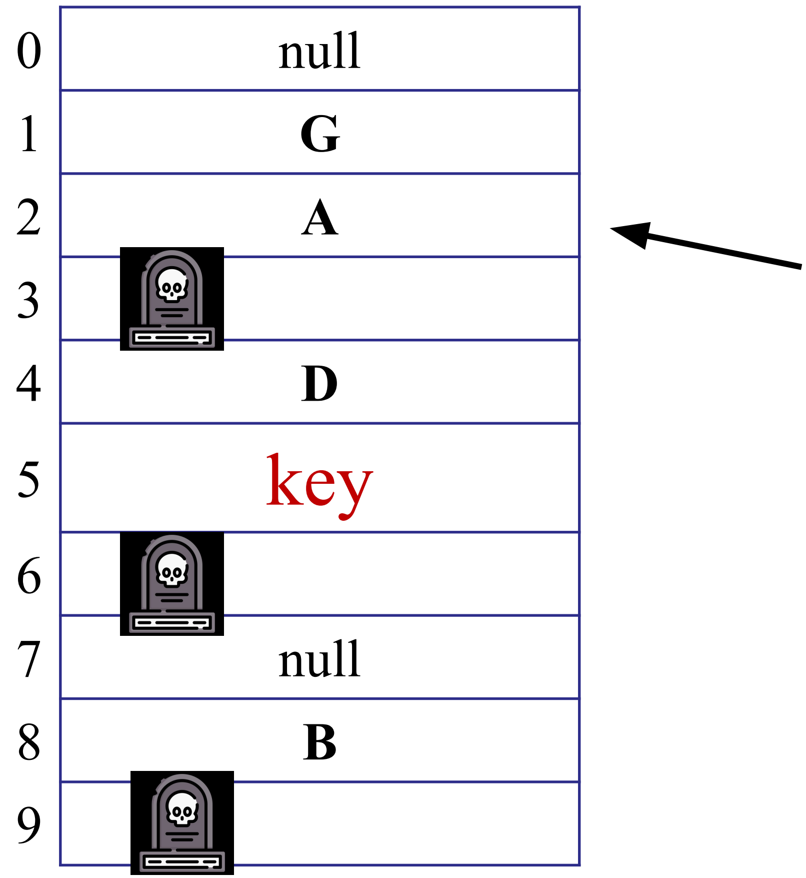
What happens

if as we insert,

we find a tombstone?

E.g. insert(**E**)

h(**E**) = 2



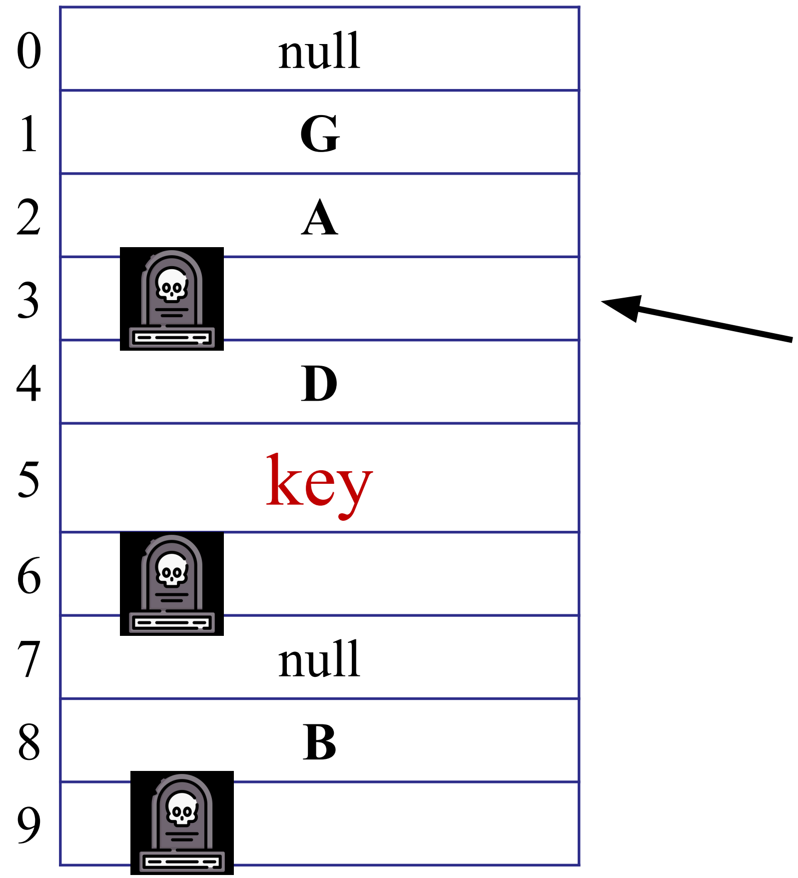
Open Addressing

Question 1:

What happens

if as we insert,

we find a tombstone?

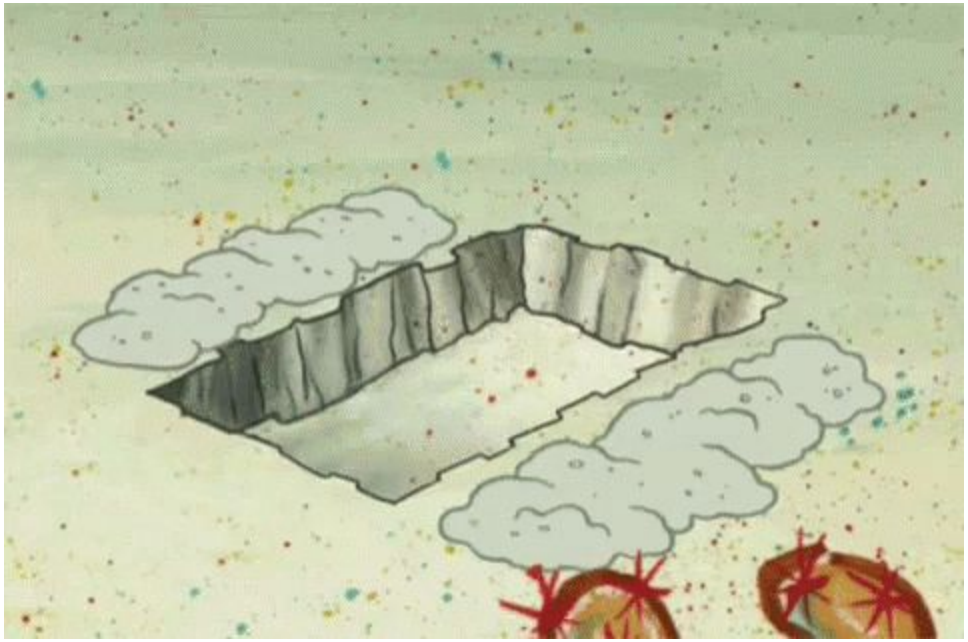


E.g. insert(**E**)

h(**E**) = 2

Open Addressing

POV: When being inserted into a table you find a tombstoned slot



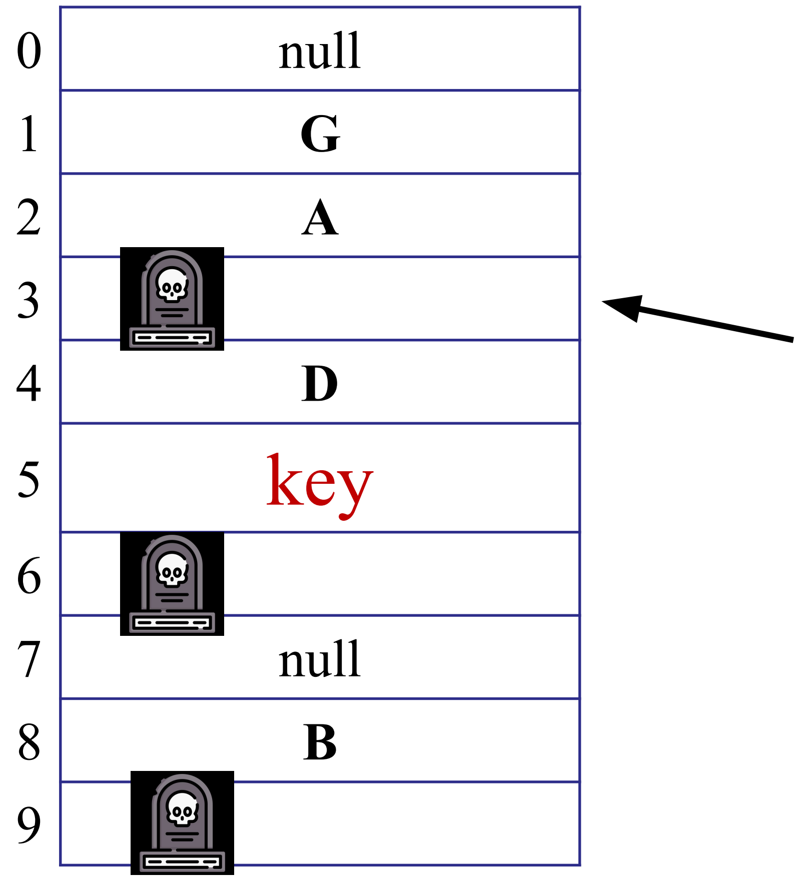
Open Addressing

Question 1:

What happens

if as we insert,

we find a tombstone?



E.g. insert(**E**)

h(**E**) = 2

if we find a tombstone, replace it!

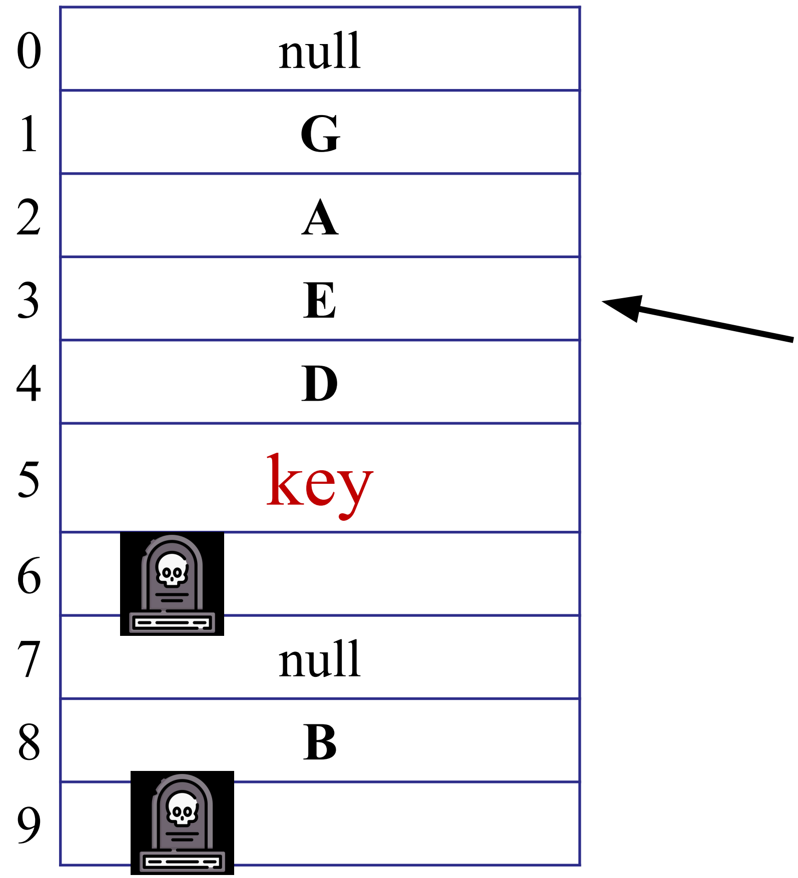
Open Addressing

Question 1:

What happens

if as we insert,

we find a tombstone?



E.g. insert(**E**)

h(**E**) = 2

if we find a tombstone, replace it!

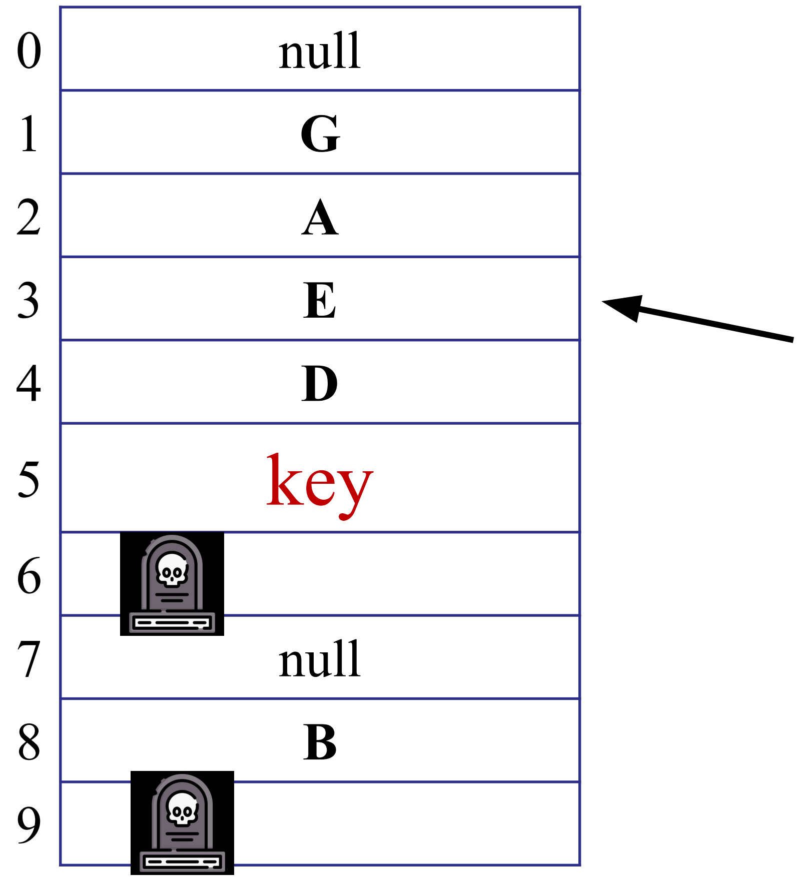
Open Addressing

Question 2:

What happens

if there are too many

tombstones?



Open Addressing

Question 2:

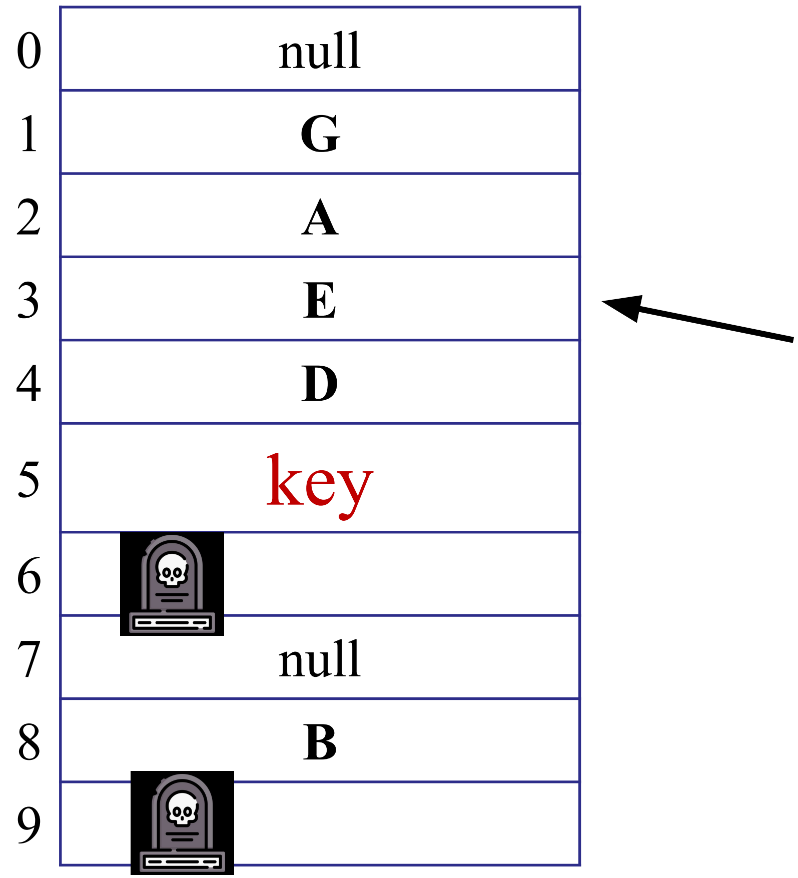
What happens

if there are too many

tombstones?

We can re-hash

all the elements.



Open Addressing

Question 2:

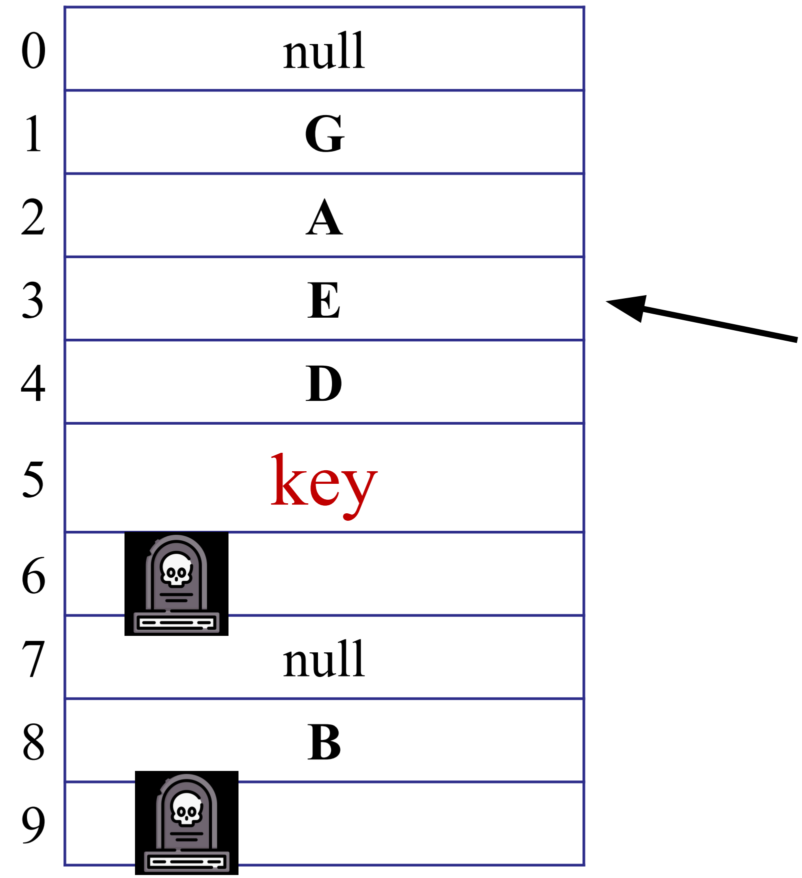
What happens

if there are too many

tombstones?

We can re-hash

all the elements.



How many is too many?

For us to know that, we first need to do the next section.

Important Observation:

The cost of insert/delete/lookup now depends on the size of the run that we hash into!

Important Observation:

The cost of insert/delete/lookup now depends on the size of the run that we hash into!

e.g. inserting into here requires us to probe further

0	null
1	**G**
2	**A**
3	**T**
4	null
5	**E**
6	null
7	null
8	**B**
9	null

Important Observation:

The cost of insert/delete/lookup now depends on the size of the run that we hash into!

0	null
1	**G**
2	**A**
3	**T**
4	null
5	**E**
6	null
7	null
8	**B**
9	null

e.g. hashing into empty slot is the cheapest

Today

- Collision resolution: open addressing
  - Linear Probing: Insert, Lookup, Delete
- Table (re)sizing
  - Amortisation
- Analysis of Linear Probing

Hash Functions

Let’s go back and fix the full table issue.

Hash Functions

Let’s go back and fix the full table issue.

Recall: When a table of size m already has m elements, we cannot insert one more.

Hash Functions

Let’s go back and fix the full table issue.

Recall: When a table of size m already has m elements, we cannot insert one more.

How should we handle this?

Table Size

How large should the table be?

- Assume: Hashing with Chaining
- Assume for now: Insertion costs $\Theta(1)$ in expectation
- Optimal size: $m = \Theta(n)$
  - if $(m < 2n)$: too many collisions.
  - if $(m > 10n)$: too much wasted space.
- Problem: we don't know n in advance.

Table Size

uses the fact that
(n/m) is constant

How large should the table be?

- Assume: Hashing with Chaining
- Assume for now: Insertion costs $\Theta(1)$ in expectation
- Optimal size: $m = \Theta(n)$

  - if $(m < 2n)$: too many collisions.
  - if $(m > 10n)$: too much wasted space.

- Problem: we don't know n in advance.

Table Size

Idea:

- Start with small (constant) table size.
- Grow (and shrink) table as necessary.

Example:

- Initially, $m = 10$.
- After inserting 6 items, table too small! Grow…
- After deleting n-1 items, table too big! Shrink…

Table Size

How to grow the table:

1. Choose new table size m.
2. Choose new hash function h.
   - Hash function depends on table size!
   - Remember: $h : U \rightarrow \{1..m\}$
3. For each item in the old hash table:
   - Compute new hash function.
   - Copy item to new bucket.

Table Size

Time complexity of growing the table:

- Assume:
  - Let m_1 be the size of the old hash table.
  - Let m_2 be the size of the new hash table.
  - Let n be the number of elements in the hash table.
- Costs:
  - Scanning old hash table: $O(m_1)$
  - Inserting each element in new hash table: $O(1)$
  - Total: $O(m_1 + n)$

Table Size

Time complexity of growing the table:

- Assume:
  - Size $m_1 < n$.
  - Size $m_2 > 2n$
- Costs:
  - Total: $O(m_1 + n)$.

    $= O(n)$

Table Size

Time complexity of growing the table:

Wait! What is the cost of initializing the new table?

- Initializing a table of size m_2 takes m_2 time!
  - when we make the new table, need to make sure every slot is set to *null*
- Costs:

  Total: $O(m_1 + m_2 + n)$

Table Size

Time complexity of growing the table:

- Assume:
  - Let m_1 be the size of the old hash table.
  - Let m_2 be the size of the new hash table.
  - Let n be the number of elements in the hash table.
- Costs:
  - Scanning old hash table: $O(m_1)$
  - Creating new hash table: $O(m_2)$
  - Inserting each element in new hash table: $O(1)$
  - Total: $O(m_1 + m_2 + n)$

How fast to grow?

Idea 1: Increment table size by 1

- if ($n == m$): $m = m+1$

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = n+1$.
  - Total: $O(n)$

Initially: $m = 8$

What is the cost of inserting n items?

1. O(n)
2. O(n log n)
3. O(n^2)
4. O(n^3)
5. None of the above.

How fast to grow?

Idea 1: Increment table size by 1

- When ($n == m$): $m = m+1$
- Cost of each resize: O(n)

Table size	8	8	9	10	11	12	…	n+1
Number of items	0	7	8	9	10	11	…	n
Number of inserts		7	1	1	1	1	…	1
Cost		7	8	9	10	11		n

- Total cost: $(7 + 8 + 9 + 10 + 11 + \ldots + n) = O(n^2)$

How fast to grow?

Idea 2: Double table size

- if ($n == m$): $m = 2m$

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = 2n$.
  - Total: $O(n)$

How fast to grow?

Idea 2: Double table size

- When $(n == m)$: $m = 2m$
- Cost of each resize: $O(n)$

Table size	8	8	16	16	16	16	16	16	16	16	32	32	32	…	2n
# of items	0	7	8	9	10	11	12	13	14	15	16	17	18	…	n
# of inserts		7	1	1	1	1	1	1	1	1	1	1	1	…	1
Cost		1	16	1	1	1	1	1	1	1	32	1	1		n

- Total (extra) resizing cost: $(7 + 15 + 31 + \ldots + n) = O(n)$

How fast to grow?

Idea 2: Double table size

- if $(n == m)$: $m = 2m$

  - Cost of resize: $O(n)$
  - Cost of inserting n items + resizing: $O(n)$

- Most insertions: $O(1)$
- Some insertions: linear cost (expensive)
- Average cost: $O(1)$

How fast to grow?

Idea 2: Double table size

– if ($n == m$): $m = 2m$

- Cost of resize: O(n)
- Cost of inserting n items + resizing: O(n)

– Most insertions: O(1)

– Some insertions: linear cost (expensive)

– Average cost: O(1)

Do not get confused with **average-case** complexity

How fast to grow?

Idea 2: Double table size

- if $(n == m)$: $m = 2m$

  - Cost of resize: O(n)
  - Cost of inserting n items + resizing: O(n)

- Most insertions: O(1)
- Some insertions: linear cost (expensive)
- Average cost: O(1)

Dividing total cost of all insertions by number of insertions made

Do not get confused with **average-case** complexity

How fast to grow?

What about deletion?

Idea: When $n = m/2$, we halve the table capacity from m to $m/2$

Does our resizing policy for deletion work?

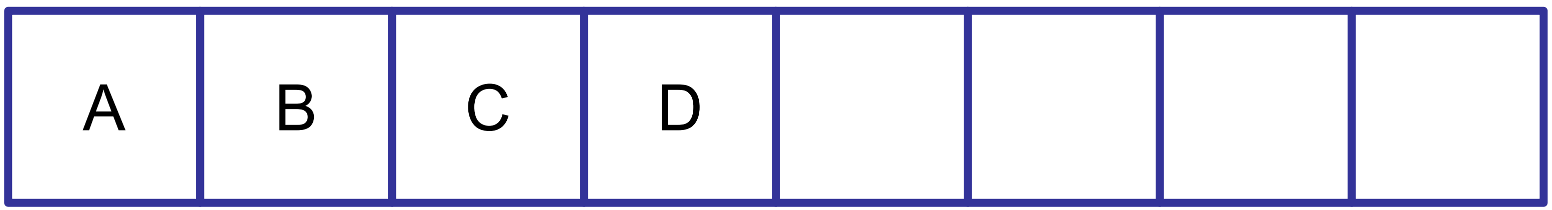
1. Yes
2. No

How fast to grow?

A	B	C	D	E			

How fast to grow?

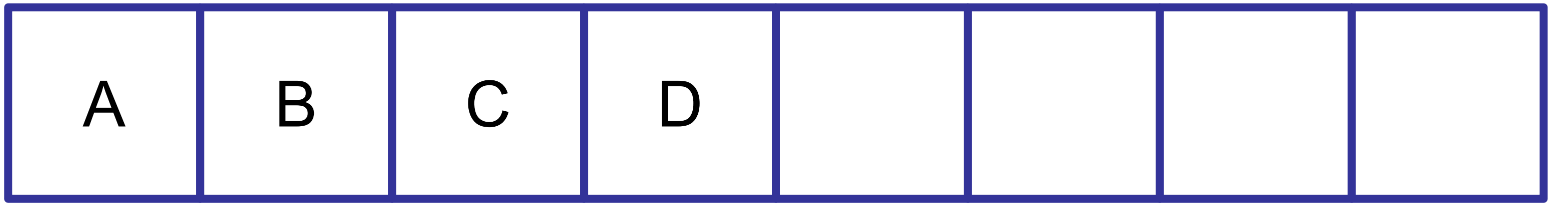
delete(**E**)



How fast to grow?

delete(**E**)

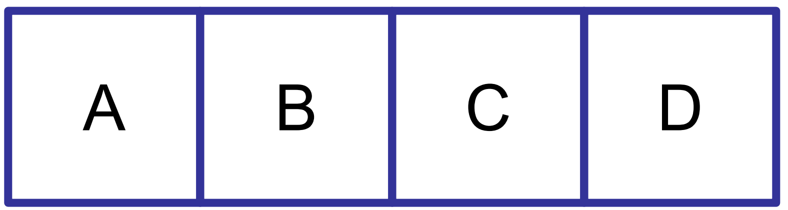
resize triggered!



How fast to grow?

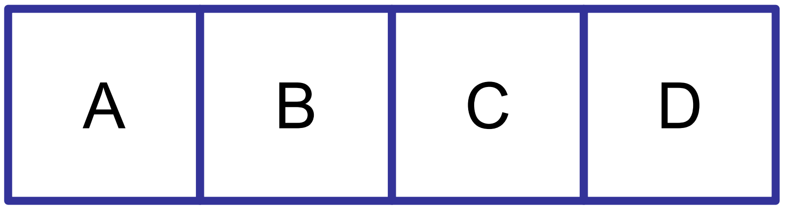
delete(**E**)

resize triggered!



How fast to grow?

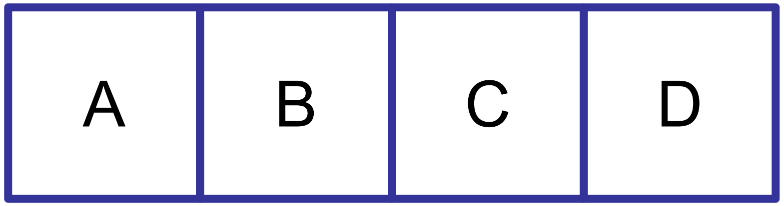
insert(**E**)



How fast to grow?

insert(**E**)

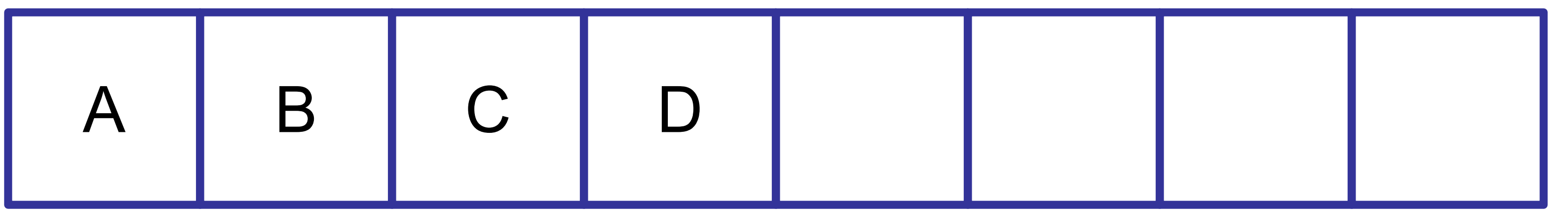
resize triggered!



How fast to grow?

insert(**E**)

resize triggered!



How fast to grow?

insert(**E**)

resize triggered!

A	B	C	D	E			

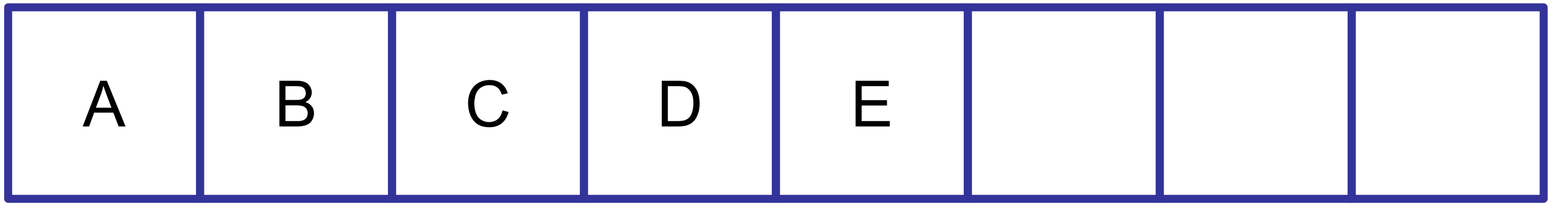
How fast to grow?

What happens if we repeatedly delete and insert **E**?

A	B	C	D	E			

How fast to grow?

What happens if we repeatedly delete and insert **E**?



It costs O(n) time per operation!

How fast to grow?

What happens if we repeatedly delete and insert **E**?

A	B	C	D	E			

How fast to grow?

So what should our resizing policy be?

How fast to grow?

So what should our resizing policy be?

- On inserts, when table is full:
  - Double the table size: $m = 2m$
- On deletes, when table is quarter full:
  - Halve the table size: $m = m/2$

How fast to grow?

But why does this work?

How fast to grow?

Informal observation:

Most inserts/deletes will cost O(1).

If we don't resize too often, then after **n** hashtable operations, the **sum total** cost of the **n** operations costs in total **O(n)**.

How fast to grow?

In fact: we only double on every power of 2.

Table size	8	8	16	16	16	16	16	16	16	16	32	32	32	…	2n
# of items	0	7	8	9	10	11	12	13	14	15	16	17	18	…	n
# of inserts		7	1	1	1	1	1	1	1	1	1	1	1	…	1
Cost		1	16	1	1	1	1	1	1	1	32	1	1		n

How fast to grow?

For the sake of analysis:
Let’s treat resize as a separate operation.

- Table full? Double table size
- Table quarter full? Halve table size

– We cannot call resize ourselves
– Resize is called based on our resizing policy

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 0
Table size: 2

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 2
Table size: 2

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + \underline{4} + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 2
Table size: 2

Table resize to 4

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 2
Table size: 4

Table resize to 4

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 3
Table size: 4

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + …

Number of items: 4
Table size: 4

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 4
Table size: 8

Table resize to 8

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 5

Table size: 8

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + …

Number of items: 6
Table size: 8

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 7
Table size: 8

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 8
Table size: 8

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Number of items: 8
Table size: 16

Table resize to 16

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Informal Claim:

Ignoring the first 3 terms, for every power of 2 term **x** in the sum, there are exactly **x/4** many terms that are +1 behind it

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + \boxed{1 + 1 + 8} + 1 + 1 + 1 + 1 + 16 + \ldots$$

Informal Claim:

Ignoring the first 3 terms, for every power of 2 term **x** in the sum, there are exactly **x/4** many terms that are +1 behind it

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + \boxed{1 + 1 + 1 + 1 + 16} + \ldots$$

Informal Claim:

Ignoring the first 3 terms, for every power of 2 term **x** in the sum, there are exactly **x/4** many terms that are +1 behind it

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Redistributing the sums we get:

$$1 + 1 + 4 + 5 + 5 + 0 + 5 + 5 + 5 + 5 + 0 + \ldots$$

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

We have as many terms in the sum as there are insertions. So total cost <= 5n + 1 + 1 + 4 = O(n)

$$1 + 1 + 4 + 5 + 5 + 0 + 5 + 5 + 5 + 5 + 0 + \ldots$$

How fast to grow?

If we had to insert **n** items into a table whose size **m** was initially 2. The total cost:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

We have as many terms in the sum as there are insertions. So total cost <= 5n + 1 + 1 + 4 = O(n)

$$1 + 1 + 4 + 5 + 5 + 0 + 5 + 5 + 5 + 5 + 0 + \ldots$$

But the story gets complicated if we have a mix of operations. What if we wanted to prove the same for an sequence of insertions and deletes?

Amortised analysis

Idea: We like the idea of re-arranging the terms in the summation for the **total cost**. Can we somehow do that again?

We pretended our insertions cost more than they should have

Actual sum:

$$1 + 1 + 4 + \boxed{1 + 1} + 8 + \boxed{1 + 1 + 1 + 1} + 16 + \ldots$$

The sum we analysed:

$$1 + 1 + 4 + \boxed{5 + 5} + 0 + \boxed{5 + 5 + 5 + 5} + 0 + \ldots$$

Amortised analysis

Idea: We like the idea of re-arranging the terms in the summation for the **total cost**. Can we somehow do that again?

We pretended our insertions cost more than they should have

Actual sum:

$$1 + 1 + 4 + 1 + 1 + \boxed{8} + 1 + 1 + 1 + 1 + \boxed{16} + \ldots$$

In exchange, we could pretend all the resizing was free!

The sum we analysed:

$$1 + 1 + 4 + 5 + 5 + \boxed{0} + 5 + 5 + 5 + 5 + \boxed{0} + \ldots$$

Amortised analysis

Idea: We like the idea of re-arranging the terms in the summation for the **total cost**. Can we somehow do that again?

Crucially, the expensive operation needs to happen infrequently enough

Actual sum:

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

so that the cheap operations can **"pay" in advance** for the **future** expensive one.

The sum we analysed:

$$1 + 1 + 4 + 5 + 5 + 0 + 5 + 5 + 5 + 5 + 0 + \ldots$$

Amortised analysis

Idea: We like the idea of re-arranging the terms in the summation for the **total cost**. Can we somehow do that again?

New analytical tool:

Operation	Actual Cost	Amortised Cost
Insert	1	1 + 4 = 5
Delete	1	1 + 4 + 5
Table Resize	n	0

Amortised analysis

Idea: We like the idea of re-arranging the terms in the summation for the **total cost**. Can we somehow do that again?

New analytical tool:

Think of these as us paying the cost in advance

Operation	Actual Cost	Amortised Cost
Insert	1	1 + 4 = 5
Delete	1	1 + 4 + 5
Table Resize	n	0

Amortised analysis

Idea: We like the idea of re-arranging the terms in the summation for the **total cost**. Can we somehow do that again?

New analytical tool:

So that later on this operation is “free”

Operation	Actual Cost	Amortised Cost
Insert	1	1 + 4 = 5
Delete	1	1 + 4 + 5
Table Resize	n	0

Amortised analysis

This means that <u>if we did **n** operations of inserts and deletes</u> (and all the resizes that needed to be done),

the total cost is n * O(5) = n * O(1) = O(n)

New analytical tool:

Operation	Actual Cost	Amortised Cost
Insert	1	1 + 4 = 5
Delete	1	1 + 4 + 5
Table Resize	n	0

Amortised analysis

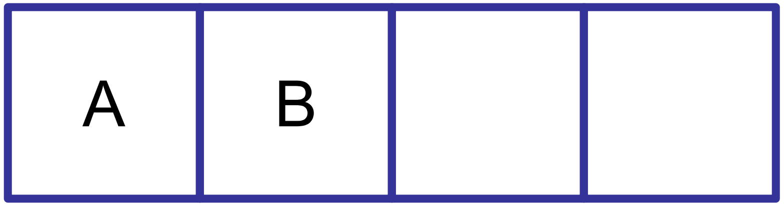
Idea:

A	B		

Let's say we *just* did a table resize. So n = m /2

Amortised analysis

Idea:



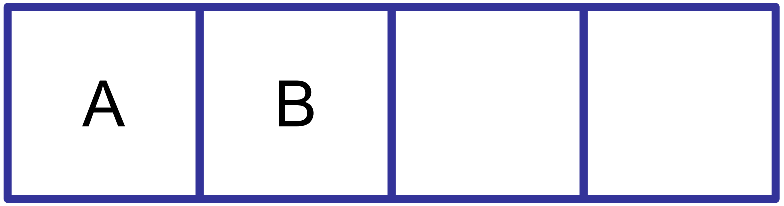
Let's say we *just* did a table resize. So n = m /2

There are 2 possible ways for us to do a table resize:

1. We insert another n elements until it is full

Amortised analysis

Idea:



Let's say we *just* did a table resize. So n = m /2

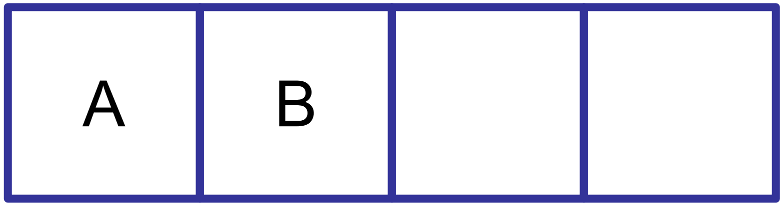
There are 2 possible ways for us to do a table resize:

1. We insert another n elements until it is full

2. We delete another n/2 elements until it is quarter full

Amortised analysis

Idea:

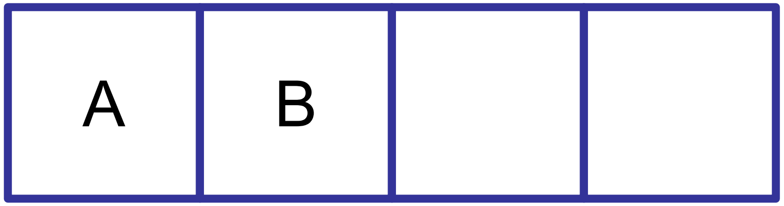


Let's say we *just* did a table resize. So n = m /2

Case 1: We insert another n elements until it is full

Amortised analysis

Idea:



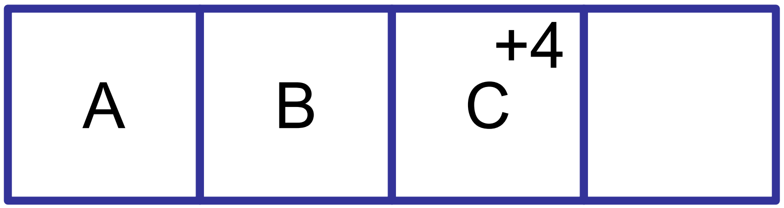
Let’s say we *just* did a table resize. So n = m /2

Case 1: We insert another n elements until it is full

The actual cost of insert was 1, if we “pretend” it is 5,

Amortised analysis

Idea:



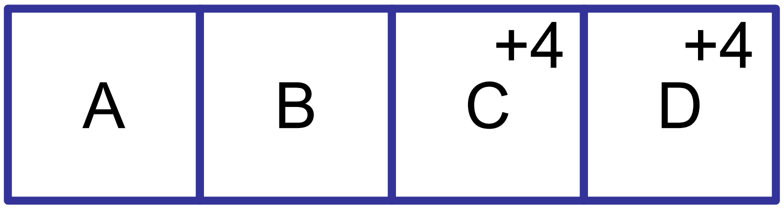
Let's say we *just* did a table resize. So n = m /2

Case 1: We insert another n elements until it is full

The actual cost of insert was 1, if we "pretend" it is 5,

Amortised analysis

Idea:



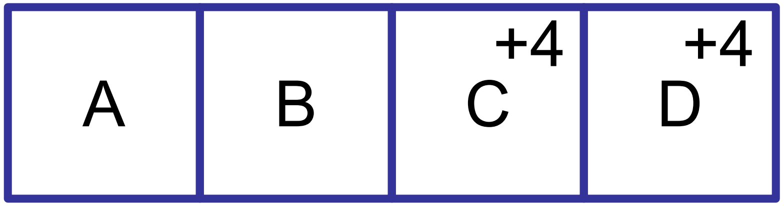
Let’s say we *just* did a table resize. So n = m /2

Case 1: We insert another n elements until it is full

The actual cost of insert was 1, if we “pretend” it is 5,

Amortised analysis

Idea:



Let's say we *just* did a table resize. So n = m /2

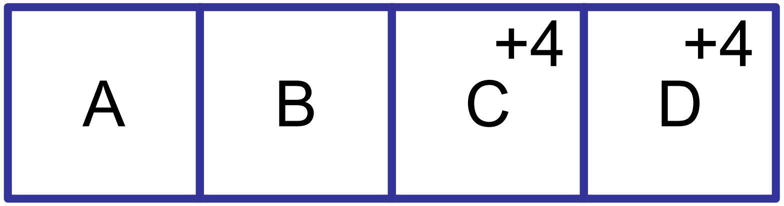
Case 1: We insert another n elements until it is full

The actual cost of insert was 1, if we "pretend" it is 5,

the "extra 4" units of time that <u>we didn't actually use</u> to insert the items now pays for the resizing

Amortised analysis

Idea:



Let's say we *just* did a table resize. So n = m /2

Case 1: We insert another n elements until it is full

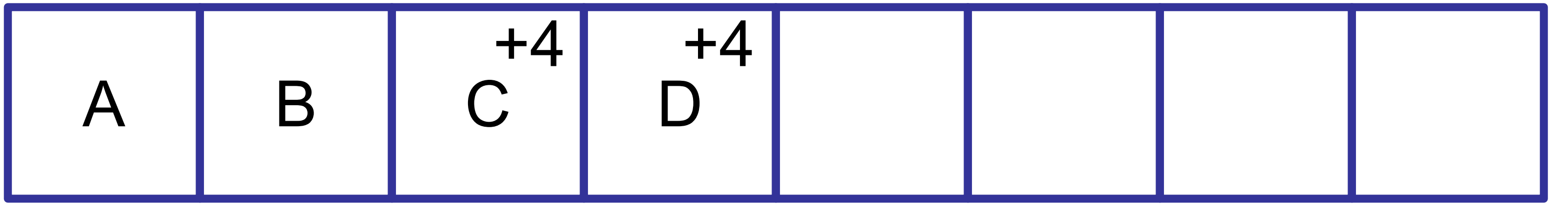
The actual cost of insert was 1, if we "pretend" it is 5,

the "extra 4" units of time that we didn't actually use to insert the items now pays for the resizing

So we have 4n surplus

Amortised analysis

Idea:



Let's say we *just* did a table resize. So n = m /2

Case 1: We insert another n elements until it is full

The actual cost of insert was 1, if we "pretend" it is 5,

the "extra 4" units of time that <u>we didn't actually use</u> to insert the items now pays for the resizing

<u>So we have 4n surplus</u>

new table size = 2m = 4(m/2) = 4n

Amortised analysis

Idea:

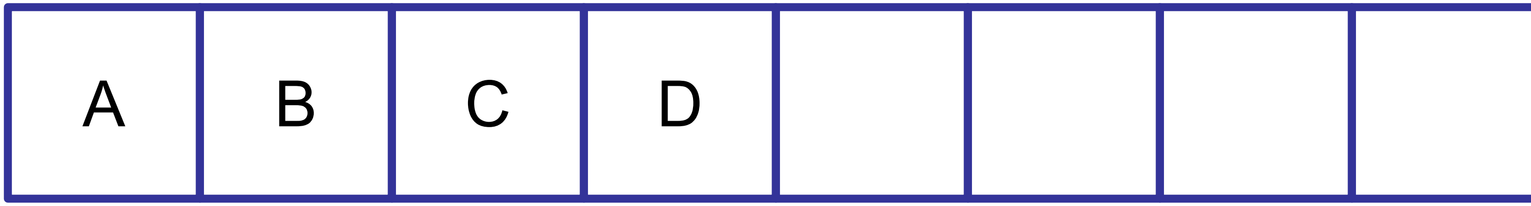
A	B	C	D				

Let’s say we *just* did a table resize. So n = m /2

Case 2: We delete another n/2 elements until it is m/4

Amortised analysis

Idea:



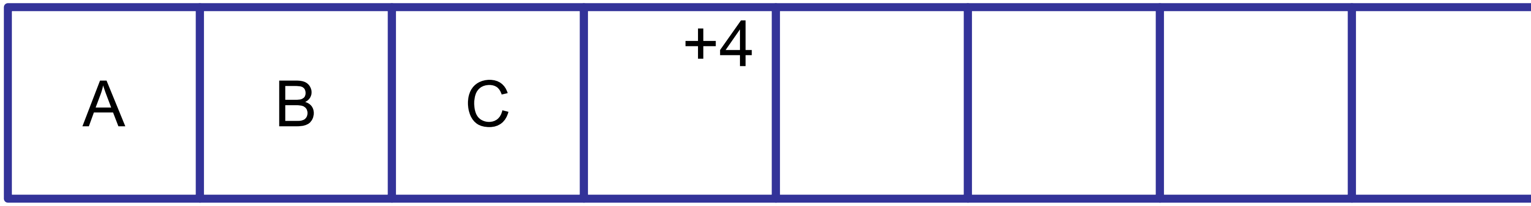
Let's say we *just* did a table resize. So n = m /2

Case 2: We delete another n/2 elements until it is m/4

Likewise, we pretend the deletion costs an extra +4 units.

Amortised analysis

Idea:



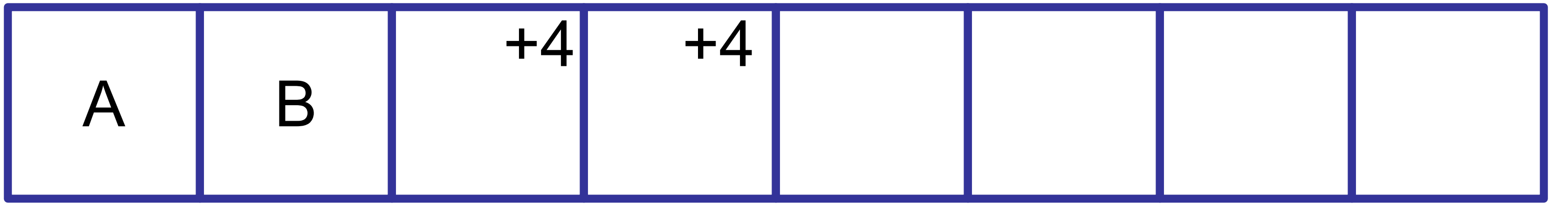
Let's say we *just* did a table resize. So n = m /2

Case 2: We delete another n/2 elements until it is m/4

Likewise, we pretend the deletion costs an extra +4 units.

Amortised analysis

Idea:



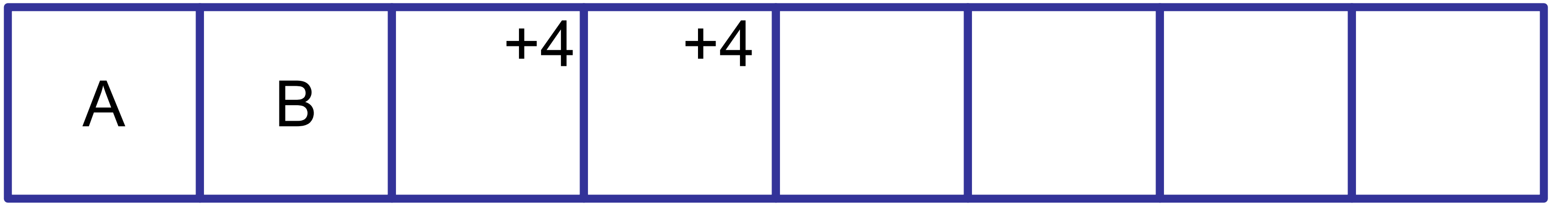
Let's say we *just* did a table resize. So n = m /2

Case 2: We delete another n/2 elements until it is m/4

Likewise, we pretend the deletion costs an extra +4 units.

Amortised analysis

Idea:



Let’s say we *just* did a table resize. So n = m /2

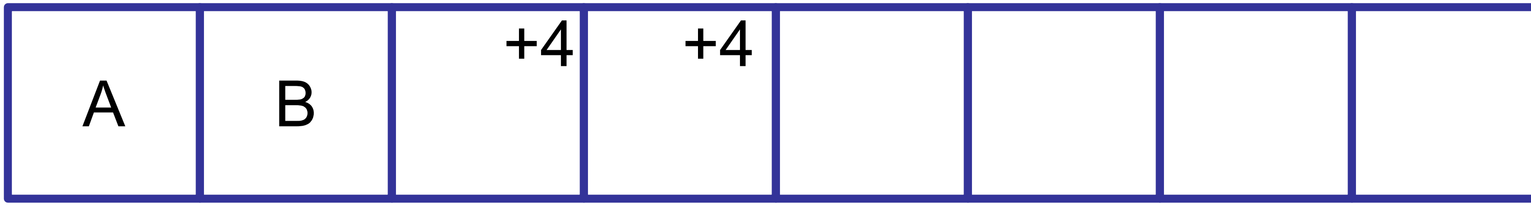
Case 2: We delete another n/2 elements until it is m/4

Likewise, we pretend the deletion costs an extra +4 units.

Then the surplus helps us again.

Amortised analysis

Idea:



Let's say we *just* did a table resize. So n = m /2

Case 2: We delete another n/2 elements until it is m/4

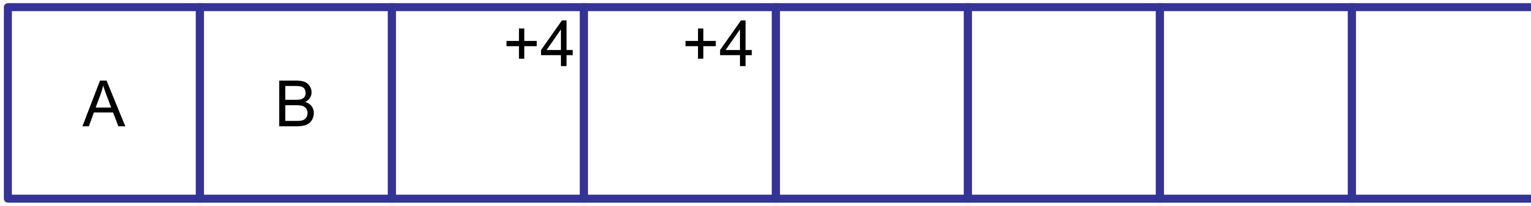
Likewise, we pretend the deletion costs an extra +4 units.

Then the surplus helps us again. We had to delete n/2 elements

new table size = m/2 = 2(m/4) =

Amortised analysis

Idea:



Let's say we *just* did a table resize. So n = m /2

Case 2: We delete another n/2 elements until it is m/4

Likewise, we pretend the deletion costs an extra +4 units.

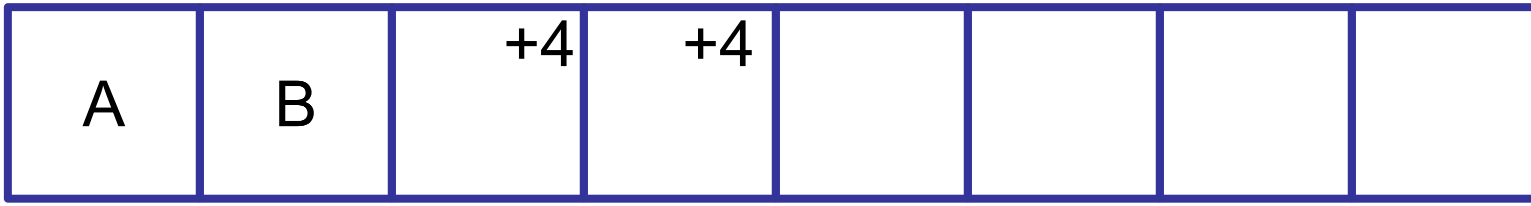
Then the surplus helps us again. We had to delete n/2 elements

So we had 2n surplus.

new table size = m/2 = 2(m/4) = 2n

Amortised analysis

Idea:



Let’s say we *just* did a table resize. So n = m /2

In both cases, because we had the surplus, we know this means that any sequence of **n** operations of inserts, and deletes, we can always pretend any required table resizing is free!

This means that the total cost is **n * O(1) = O(n)**!

How fast to grow?

Remember: "Pretending" the cost here is just a fancy way of letting us redistribute the terms in the sum

PREVIOUSLY

$$1 + 1 + 4 + 1 + 1 + 8 + 1 + 1 + 1 + 1 + 16 + \ldots$$

Redistributing the sum we get:

$$1 + 1 + 4 + 5 + 5 + 0 + 5 + 5 + 5 + 5 + 0 + \ldots$$

Amortised analysis

Important note:

The difference between actual cost and amortised cost.

Operation	Actual Cost	Amortised Cost
Insert	1	1 + 4 = 5
Delete	1	1 + 4 + 5
Table Resize	n	0

Amortised analysis

Important note:

When considered as a single operations, the worst-case cost is still O(n)! **Sometimes** an insert/delete will still take O(n) time due to the table resizing that happens.

Operation	Actual Cost	Amortised Cost
Insert	1	1 + 4 = 5
Delete	1	1 + 4 + 5
Table Resize	n	0

Amortised analysis

Important note:

On the other hand, when considering **a sequence** of **n** operations, some of which are inserts, and some of which are deletes, the **sum total cost** of all of the operations are **O(1)** for any insert/deletes + **0** for any resizing that happens in the sequence.

Operation	Actual Cost	Amortised Cost
Insert	1	1 + 4 = 5
Delete	1	1 + 4 + 5
Table Resize	n	0

Amortised analysis

Why is the distinction important?

Imagine two algorithms that are used to process server requests:

Algorithm	Longest time taken for single request	Time taken for a batch of **n** requests
Algorithm 1	0.001s	5n+0.001s
Algorithm 2	1s	0.01n+1s

Amortised analysis

Why is the distinction important?

Imagine two algorithms that are used to process server requests:

Algorithm	Longest time taken for single request	Time taken for a batch of **n** requests
Algorithm 1	0.001s	5n+0.001s
Algorithm 2	1s	0.01n+1s

Worst case time per single operation: Tail Latency!

Amortised analysis

Why is the distinction important?

Imagine two algorithms that are used to process server requests:

Algorithm	Longest time taken for single request	Time taken for a batch of **n** requests
Algorithm 1	0.001s	5n+0.001s
Algorithm 2	1s	0.01n+1s

Time taken for sequence of operations: Throughput!

Amortised analysis

If you are worried about the spike in runtime, and cannot afford any single expensive operation, maybe consider using a tree.

Operation	Actual Cost	Amortised Cost
Insert	1	1 + 4 = 5
Delete	1	1 + 4 + 5
Table Resize	n	0

Amortised analysis

If you are okay, but prefer higher throughput, perhaps a hashtable/symbol table is ok.

Operation	Actual Cost	Amortised Cost
Insert	1	1 + 4 = 5
Delete	1	1 + 4 + 5
Table Resize	n	0

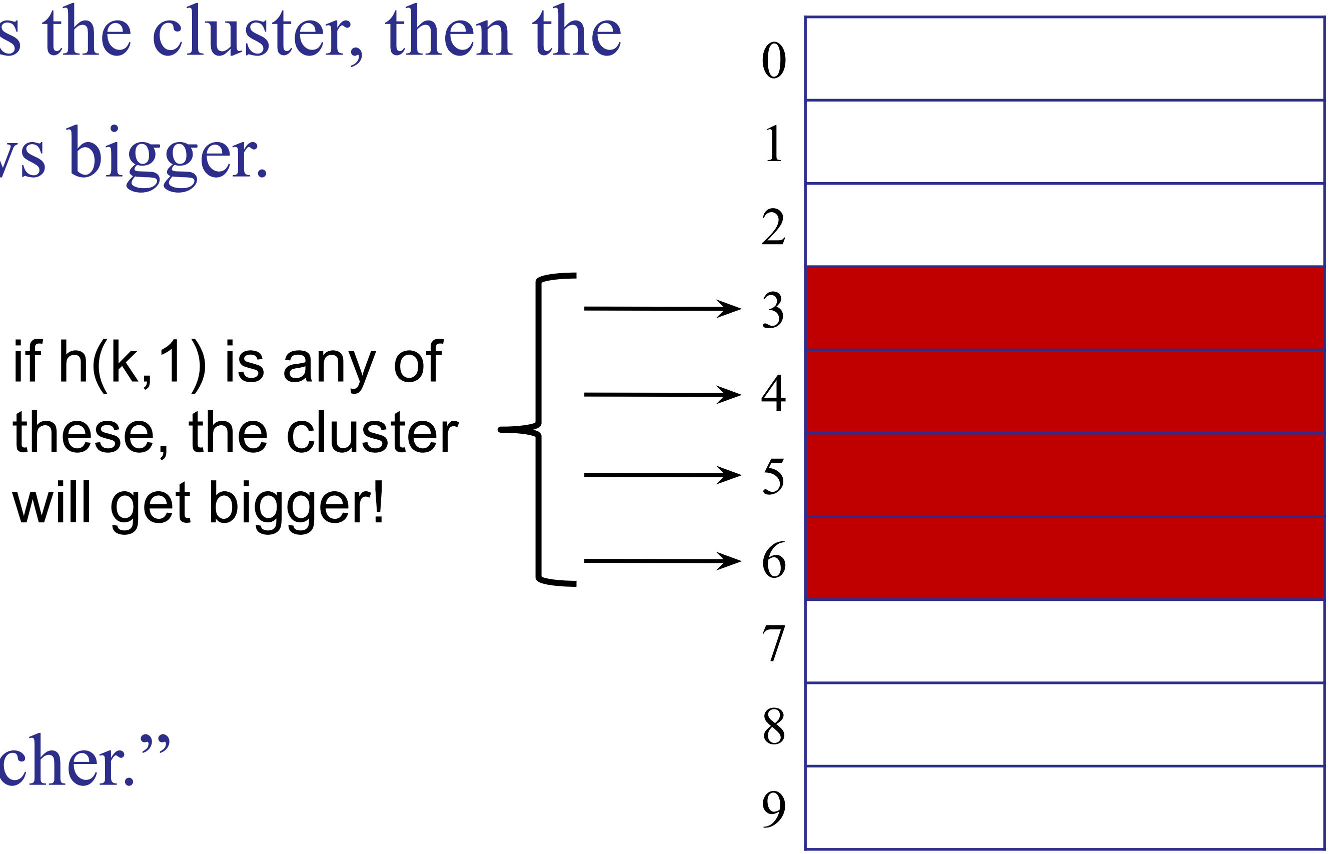
Today

- Collision resolution: open addressing
  - Linear Probing: Insert, Lookup, Delete
- Table (re)sizing
  - Amortisation
- Analysis of Linear Probing

Linear Probing

Problem with linear probing: ***clusters***

- If there is a cluster, then there is a higher probability that the next h(k) will hit the cluster.
- If h(k,1) hits the cluster, then the cluster grows bigger.



- "Rich get richer."

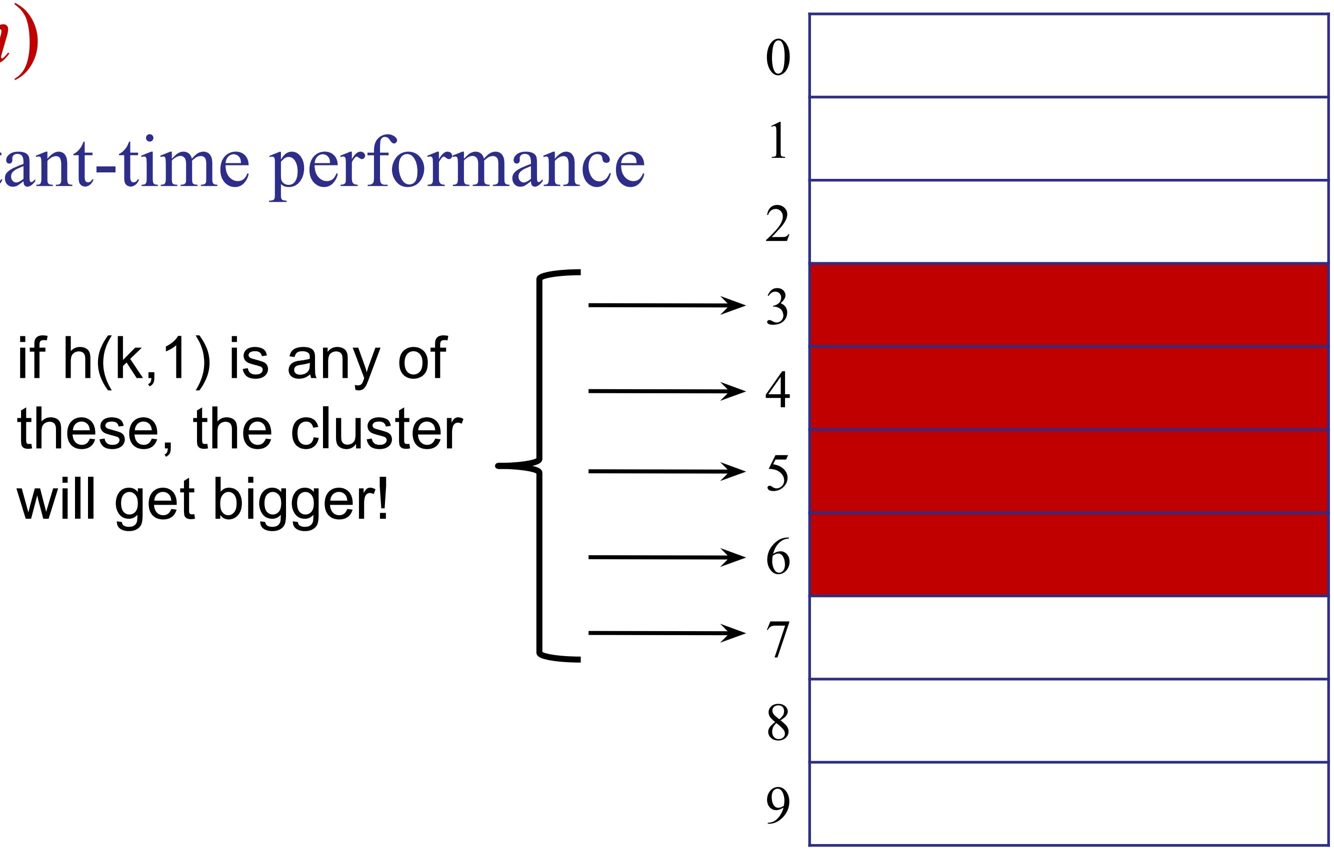
Linear Probing

Problem with linear probing: ***clusters***

- If the table is 1/4 full, then there will be clusters of size:

  $\Omega(\log n)$

- Ruins constant-time performance



That conversation again…

Professor (for the last 30 years):

"Linear probing is bad because it leads to clusters and bad performance. We need uniform hashing."

Punk in the front row:

"But I ran some experiments and linear probing seems really fast."

Professor:

"Maybe your experiments were too small, or just weren't very well done. Let me prove to you that uniform hashing is good."

That conversation again…

Professor (for the last 30 years):

"Linear probing is bad because it leads to clusters and bad performance. We need uniform hashing."

Punk in the front row:

"But I ran some experiments and linear probing seems really fast."

Professor:

"Maybe your experiments were too small, or just weren't very well done. Let me prove to you that uniform hashing is good."

Punk in the front row goes and starts a billion dollar startup doing high performance data processing.

Student sitting next to punk in the front row goes to grad school and proves that linear probing really is faster.

Linear probing: In Practice

In practice, linear probing is faster!

- **Reason #1:** Caching!
- It is *cheap* to access nearby array cells.
  - Example: access T[17]
  - Cache loads: T[10..50]
  - Almost 0 cost to access T[18], T[19], T[20], …
- If the table is 1/4 full, then there will be clusters of size: $\Omega(\log n)$
  - Cache may hold entire cluster!
  - No worse than wacky probe sequence.

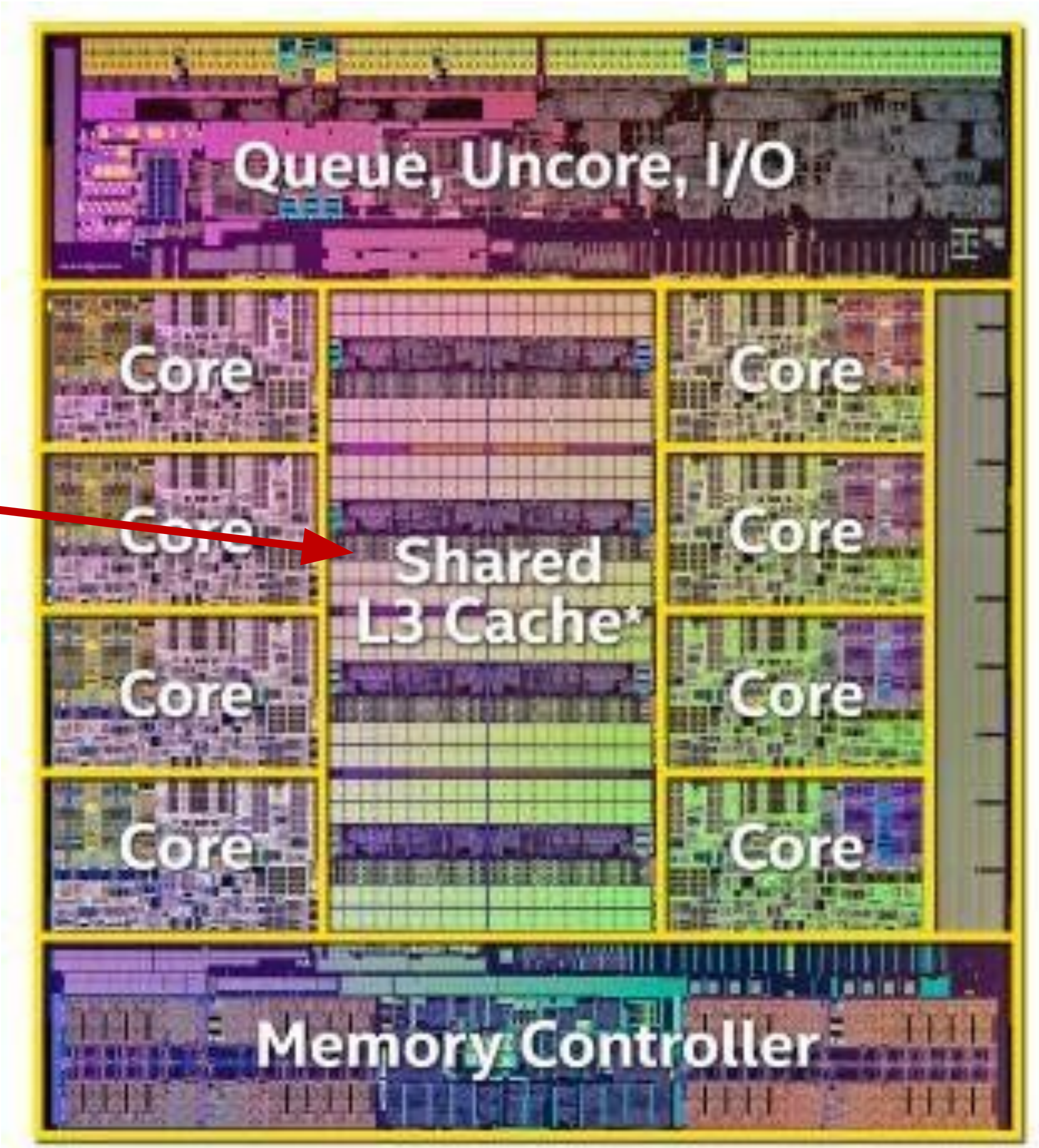
Linear probing: In Practice

In practice, linear probing is faster!

- **Reason #2:** Prefetching!
- If your memory access pattern is easily "predictable", your CPU can fetch the cache line even before you use it.
- A linear probe is as predictable as it gets. While you are processing the current cache line, your CPU will pre-fetch the next one for you.

Linear probing: In Practice

Intel spent all this real estate on it,
we might as well make use of it



Linear Probing: In Theory

- Highly non-trivial to show expected running time of O(1) with constant load factor (and good hash functions).
  - Pagh, Pagh, and Ružić in 2011
  - Patrascu, and Thorup in 2013
  - See Seth's notes:
    https://www.comp.nus.edu.sg/~gilbert/CS5330/2019/lectures/03.Hashing.pdf

Linear Probing: In Theory

- Highly non-trivial to show expected running time of O(1) with constant load factor (and good hash functions).
  - Pagh, Pagh, and Ružić in 2011
  - Patrascu, and Thorup in 2013
  - See Seth's notes: https://www.comp.nus.edu.sg/~gilbert/CS5330/2019/lectures/03.Hashing.pdf

- **Takeaway**: Even though linear probing looks bad, we can take it to be O(1) expected running time for insert/delete/lookup, assuming a special time of efficient hash functions.

Linear Probing: Conclusion

- Linear probing is actually great with both:
  - Theoretical O(1) guarantees under reasonable assumptions
  - Practical support from how modern CPUs work

* In CS2040S the takeaway is that we are happy to take insert and delete (without table resizing) as O(1) **in expectation**. But still **O(n)** worst case because of either resizing, or probing a long run of keys.

Today

- Collision resolution: open addressing
  - Linear Probing: Insert, Lookup, Delete
- Table (re)sizing
  - Amortisation
- Analysis of Linear Probing
- Other methods of resolution.

Resolving Collisions

- Open Addressing
  - Items are inserted into the table directly

What if we did not just probe linearly?

Resolving Collisions

- (Previously) Chaining

Resolving Collisions

- (Previously) Chaining

`std::unordered_map` is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal of elements have average constant-time complexity.

Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its key. Keys with the same hash code appear in the same bucket. This allows fast access to individual elements, since once the hash is computed, it refers to the exact bucket the element is placed into.

Two keys are considered *equivalent* if the map's key equality predicate returns true when passed those keys. If two keys

https://en.cppreference.com/w/cpp/container/unordered_map

```c
/**
 * hash_add - add an object to a hashtable
 * @hashtable: hashtable to add to
 * @node: the &struct hlist_node of the object to be added
 * @key: the key of the object to be added
 */
#define hash_add(hashtable, node, key)                                    \
        hlist_add_head(node, &hashtable[hash_min(key, HASH_BITS(hashtable))])

```

https://github.com/torvalds/linux/blob/master/include/linux/hashtable.h

Resolving Collisions

- Open Addressing
  - Items are inserted into the table directly

What if we did not just probe linearly?

How about $h(x) + i^2$ (for i collisions)

Quadratic probing!

Resolving Collisions

- Open Addressing

Type ‘S’ or ‘/’ to search, ‘?’ for more options...

std::collections

Struct HashMap

Since 1.0.0 · Source

Settings

```
pub struct HashMap<K, V, S = RandomState> { /* private fields */ }
```

A hash map implemented with quadratic probing and SIMD lookup.

By default, `HashMap` uses a hashing algorithm selected to provide resistance against HashDoS attacks. The a
seeded, and a reasonable best-effort is made to generate this seed from a high quality, secure source of rando
the host without blocking the program. Because of this, the randomness of the seed depends on the output q
random number coroutine when the seed is created. In particular, seeds generated when the system’s entrop
low such as during system boot may be of a lower quality.

Quadratic probing!

Resolving Collisions

- Open Addressing
  - Items are inserted into the table directly

What if we did not just probe linearly?

How about using another “random” function $g(i)$?

First probe $h(x) + g(0)$, then $h(x) + g(1)$,

then $h(x) + g(2)$, and so on….

Resolving Collisions

- Open Addressing
  - Items are inserted into the table directly

What if we did not just probe linearly?

How about using another "random" function $g(i)$?

First probe $h(x) + g(0)$, then $h(x) + g(1)$,

then $h(x) + g(2)$, and so on….

Random probing!

Resolving Collisions

```
The first half of collision resolution is to visit table indices via this
recurrence:

    j = ((5*j) + 1) mod 2**i

For any initial j in range(2**i), repeating that 2**i times generates each
int in range(2**i) exactly once (see any text on random-number generation for
proof).  By itself, this doesn't help much:  like linear probing (setting
```

https://hg.python.org/cpython/file/5620691ce26b/Objects/dictobject.c#l105

Costs Per Operation:

Hashing with linear probing:

Operation	Actual Cost	Amortised Cost
Insert	O(1)	O(1)
Delete	O(1)	O(1)
If table resize triggered	O(n)	0
Search	O(1)	O(1)

Resolving Collisions

Other cool tables we do not have time to cover:

1. Robin hood hashtables

2. Cuckoo hashtables

Resolving Collisions

Other cool tables we do not have time to cover:

1. Robin hood hashtables

2. Cuckoo hashtables

Fun Fact: these and linear probing are Eldon’s favourite hashing methods.

Next Week

Heaps, and Priority Queues