CS2040S
Data Structures and Algorithms

Hashing III

Hashing!

- Introduction to Hashing
- Collision Resolution: chaining
- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

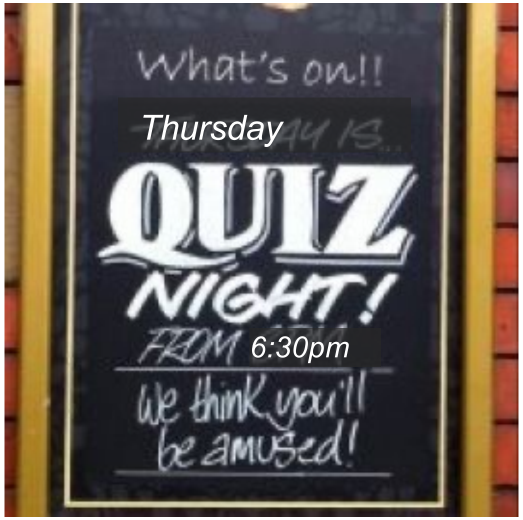
Midterm

Thurs. March 10 6:30pm

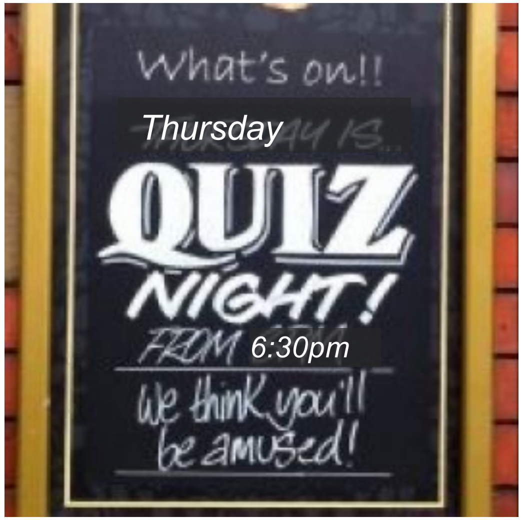
- Location: MPSH
- Room assignment on Coursemology
- Please double-check room

Bring to quiz:

- One double-sided sheet of paper with any notes you like.
- Pens/pencils.
- You may not use anything else. (No calculators, no phones, etc.)



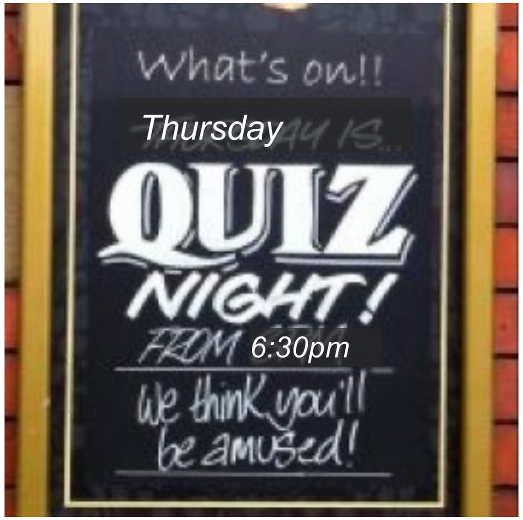
Midterm



Covid Issues

- Please do test before taking the midterm.
- You will need to have the "green pass" on the Univus app to take midterm.
- Please do not come if you feel unwell.

Midterm



What happens if covid positive?

- Upload test to Univus.
- There will be a makeup next week.
- Get well soon!

What if I don't feel well?

- If covid positive, see above.
- If not covid positive, see doctor for MC ➔ Makeup.
- But don't take midterm if unwell!

Hashing!

- Introduction to Hashing
- Collision Resolution: chaining
- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Abstract Data Types

Symbol Table

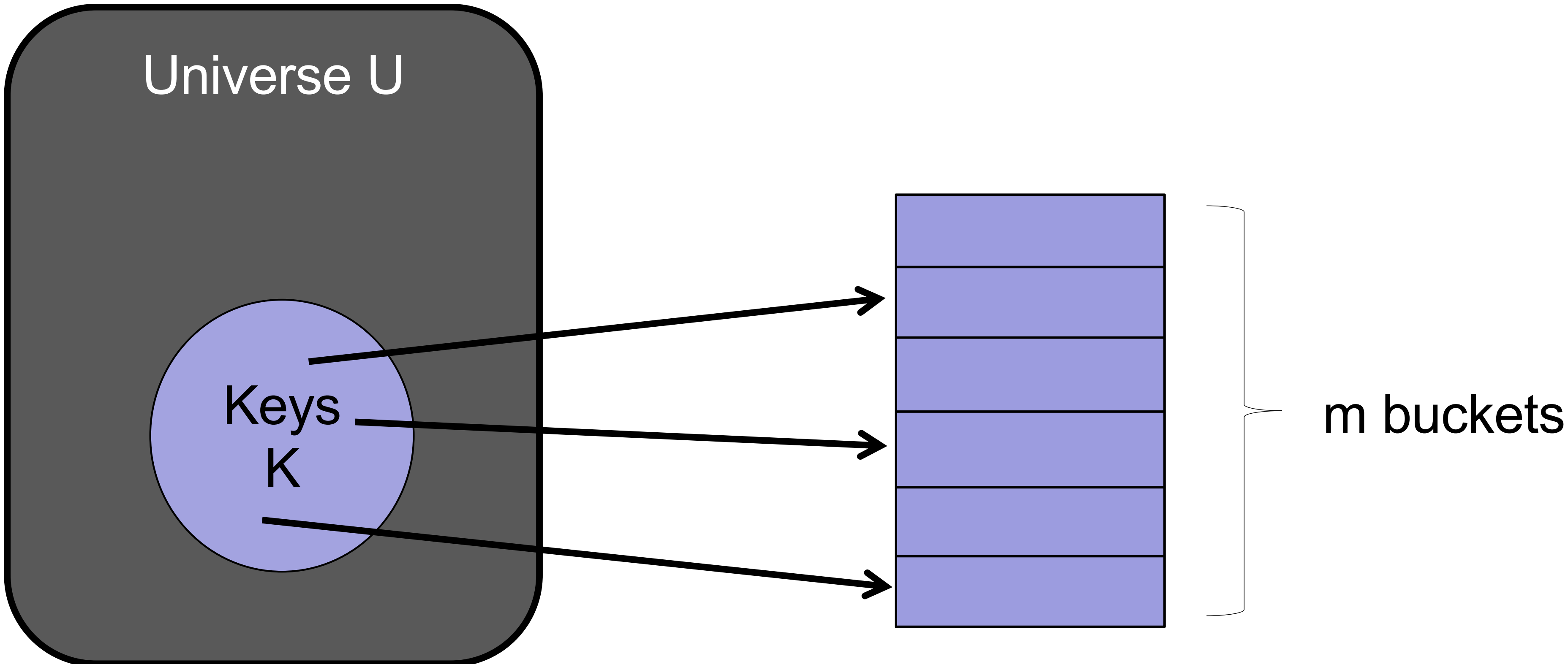
```
public interface  SymbolTable

     void   insert(Key k, Value v)  insert (k,v) into table
    Value   search(Key k)           get value paired with k
     void   delete(Key k)           remove key k (and value)
  boolean   contains(Key k)         is there a value for k?
      int   size()                  number of (k,v) pairs
```

Note: no successor / predecessor queries.

Hash Functions

Problem:

- Huge universe U of possible keys.
- Smaller number n of actual keys.
- How to map n keys to $m \approx n$ buckets?



Hash Functions

Collisions:

- We say that two distinct keys k_1 and k_2 collide if:

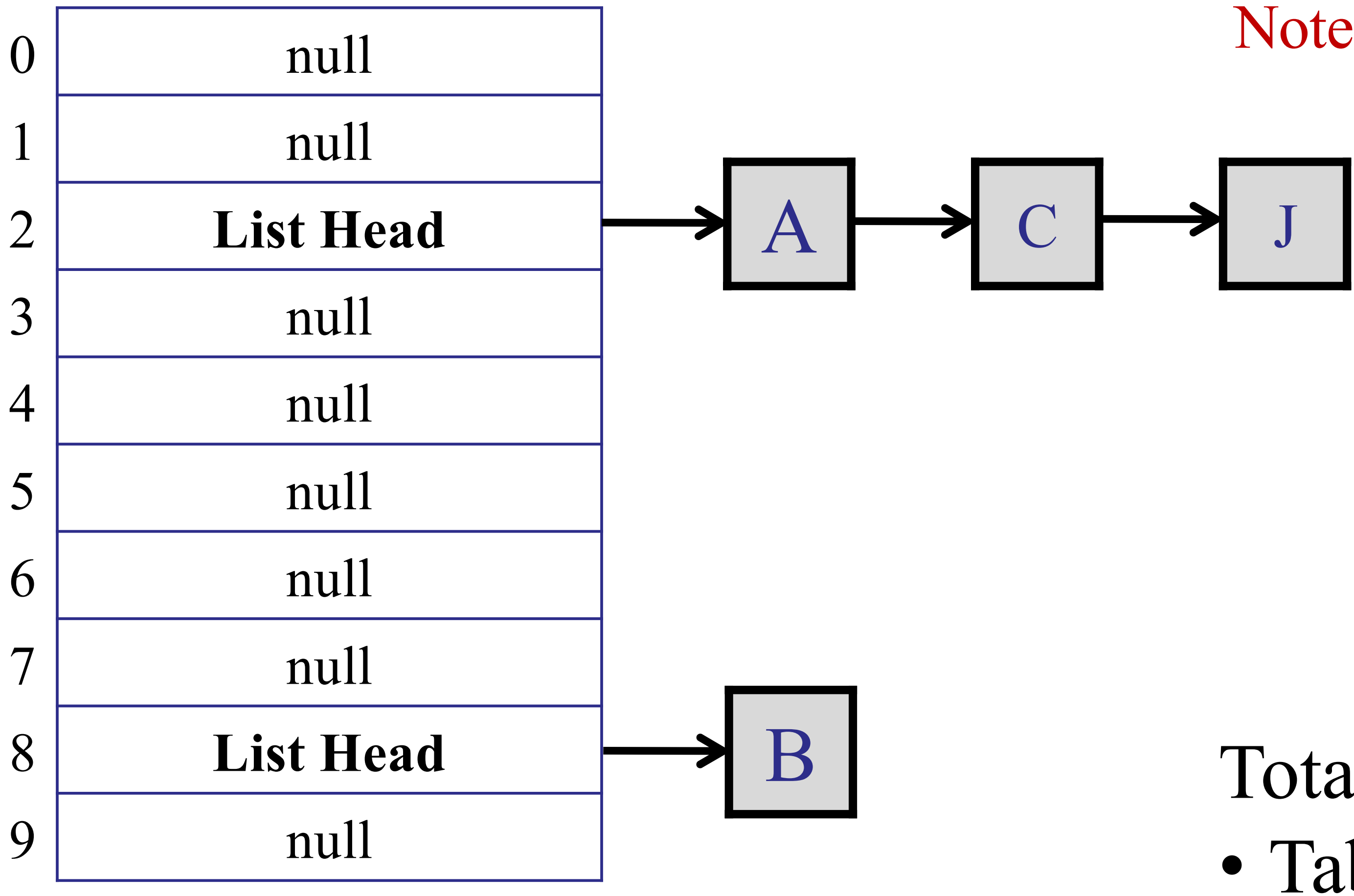
$$h(k_1) = h(k_2)$$

- Unavoidable!
  - The table size is smaller than the universe size.
  - The pigeonhole principle says:
    - There must exist two keys that map to the same bucket.
    - Some keys must collide!

Chaining

Each bucket contains a linked list of items.

Note: h(A) == h(C) == h(J)



Total space: $O(m + n)$

- Table size: m
- Linked list size: n

Let’s be optimistic today.

The Simple Uniform Hashing Assumption

- Every key is equally likely to map to every bucket.
- Keys are mapped independently.

Assume hash function has this property, even if it may not!

Intuition:

- Each key is put in a random bucket.
- Then, as long as there are enough buckets, we won’t get too many keys in any one bucket.

Hashing with Chaining

If hash function satisfies Simple Uniform Hashing Assumption

Searching:

- Expected search time = $1 + n/m = O(1)$
- Worst-case search time = O(n)

Inserting:

- Worst-case insertion time = O(1)

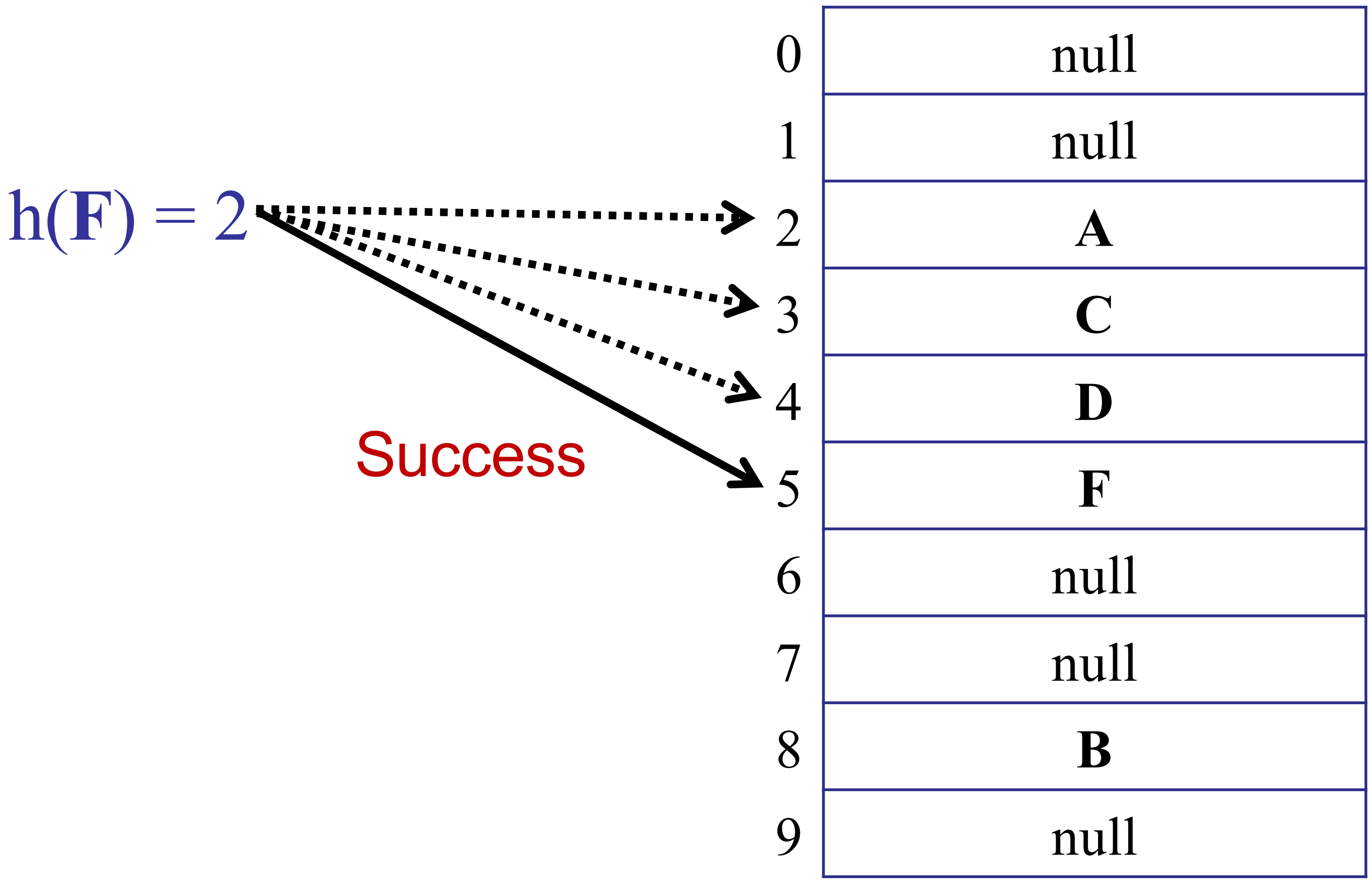
** In this case, inserting allows duplicates…

Preventing duplicates requires searching.

Open Addressing

On collision:

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



Hash Functions

Two properties of a good hash function:

1. h(key, i) enumerates all possible buckets.
   - For every bucket j, there is some i such that:

     $$h(key, i) = j$$

   - The hash function is permutation of $\{1..m\}$.
   - For linear probing: true!

Hash Functions

Two properties of a good hash function:

2. Uniform Hashing Assumption

   Every key is equally likely to be mapped to every ***permutation***, independent of every other key.

n! permutations for probe sequence: e.g.,

- 1 2 3 4
- 1 2 4 3
- 1 4 2 3
- 1 4 3 2
- …

Performance of Open Addressing

- Chaining:
  - When (m==n), we can still add new items to the hash table.
  - We can still search efficiently.

- Open addressing:
  - When (m==n), the table is full.
  - We cannot insert any more items.
  - We cannot search efficiently.

Performance of Open Addressing

Define:

- Load $\alpha = n / m$ Average # items / bucket
- Assume $\alpha < 1$.

Claim:

For n items, in a table of size m, assuming *uniform hashing*, the expected cost of an operation is:

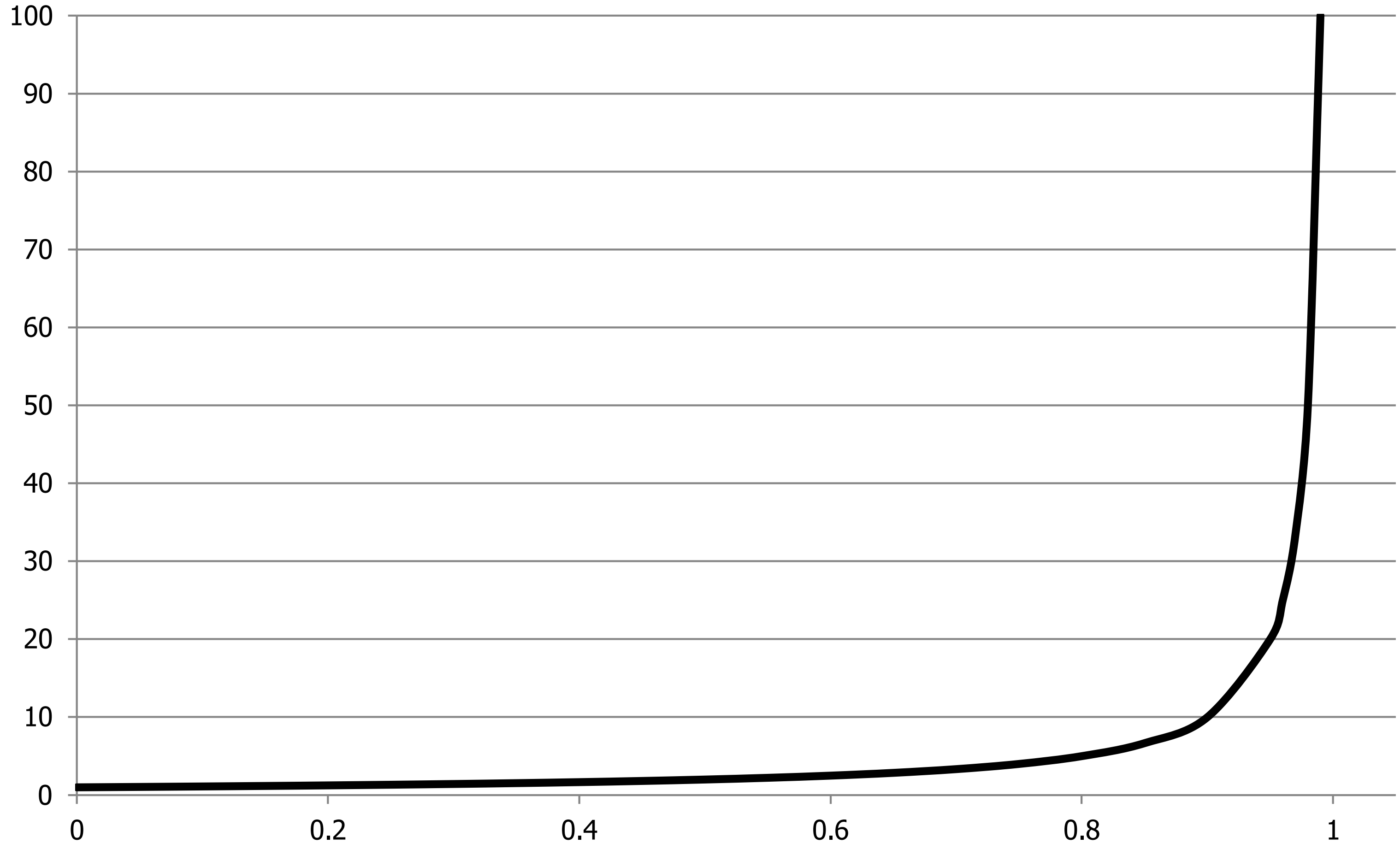
$$\leq \frac{1}{1-\alpha}$$

Example: if (α=90%), then E[# probes] = 10

Disadvantages…

Open addressing:

– Performance degrades badly as $\alpha \rightarrow 1$.



Hashing: Recap

Problem: coping with large universe of keys

- Number of possible keys is very, very large.
- Direct Access Table takes too much space

Hash functions

- Use hash function to map keys to buckets.
- Sometimes, keys collide (inevitably!)

Resolve collisions

- Chaining ➔ SUHA ➔ $O(1 + \alpha)$ expected cost ops
- Open Addressing ➔ UHA ➔ $O(1 / 1 - \alpha)$ expected cost ops

Hashing!

- Introduction to Hashing
- Collision Resolution: chaining
- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Table Size

How large should the table be?

- Assume: Hashing with Chaining
- Assume: Simple Uniform Hashing
- Expected search time: $O(1 + n/m)$
- 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 X takes X time!

– 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: $m = 8$
Increase table size by 1 on resize.
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.

ARCHIPELAGO
is open

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		7	8	1	1	1	1	1	1	1	16	1	1		n

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

How fast to grow

Idea 2: Double table size

Cost of Resizing:

Table size	Total Resizing Cost
8	8
16	(8 + 16)
32	(8 + 16 + 32)
64	(8 + 16 + 32 + 64)
128	(8 + 16 + 32 + 64 + 128)
…	…
m	$<(1+2+4+8+\ldots+m) \leq O(m)$

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: $O(1)$

Design question

Do you care that some insertions take a lot longer than others?

- Most insertions: O(1)
- Some insertions: linear cost (expensive)
- Total cost is good…
- … but what if the slow operation is really, really important / time critical?
- What if YOUR online purchase is the one that triggers a two hour database rebuild?

How fast to grow?

Idea 3: Square table size

- When ($n == m$): $m = m^2$

Table size	Total Resizing Cost
8	?
64	?
4,096	?
16,777,216	?
…	…
m	?

Assume: square table size

What is the cost of inserting n items?

1. $O(\log n)$
2. $O(\sqrt{n})$
3. $O(n)$
4. $O(n \log n)$
5. $O(n^2)$
6. $O(2^n)$
7. None of the above.

ARCHIPELAGO

is open

How fast to grow?

Idea 3: Square table size

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

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

    $= O(n + n^2 + n)$

    $= O(n^2)$

How fast to grow?

Idea 3: Square table size

- When ($n == m$): $m = m^2$

# Items	Total Resizing Cost
8	64
64	(64 + 4,096)
4,096	(64 + 4,096 + …)
…	…
n	$> n^2$
	$= O(n^2)$

How fast to grow?

Idea 3: Square table size

- When ($n == m$): $m = m^2$

# Items	Resizing Cost	Insert Cost
8	64	8
64	(64 + 4,096)	64
4,096	(64 + 4,096 + …)	4,096
…	…	…
n	$> n^2$	n
	$< O(n^2)$	O(n)

How fast to grow?

Idea 3: Square table size

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

- Cost of resize:
  - Total: $O(n^2)$

- Cost of inserts:
  - Total: $O(n)$

Why else is squaring the table size bad?

1. Resize takes too long to find items to copy.
2. Inefficient space usage.
3. Searching is more expensive in a big table.
4. Inserting is more expensive in big table.
5. Deleting is more expensive in a big table.

ARCHIPELAGO

is open

Deleting Elements

Basic procedure: (chained hash tables)

Delete(*key*)

1. Calculate hash of *key*.
2. Let L be the linked list in the specified bucket.
3. Search for item in linked list L.
4. Delete item from linked list L.

Cost:

– Total: $O(1 + n/m)$

Deleting Elements

What happens if too many items are deleted?

- Table is too big!
- Shrink the table…

- Try 1:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/2)$ then $m = m/2$.

Deleting Elements

Rules for shrinking and growing:

– Try 1:

- If $(n == m)$, then $m = 2m$.
- If $(n < m/2)$ then $m = m/2$.

– Example problem:

- Start: n=100, m=200
- Delete: n=99, m=200 → shrink to m=100
- Insert: n=100, m=100 → grow to m=200
- Repeat…

Deleting Elements

Example execution:

- Start: n=100, m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- Repeat…

Deleting Elements

Rules for shrinking and growing:

- Try 2:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/4)$, then $m = m/2$.

- Claim:
  - Every time you double a table of size m, at least $m/2$ new items were added.
  - Every time you shrink a table of size m, at least $m/4$ items were deleted.

Amortized Analysis

Technique for analyzing "average" cost:

– Common in data structure analysis

– Smooths the cost when some operations are expensive and some operations are cheap:

  - E.g., some ops are O(1), some are O(n).
  - What matters is the *total* cost of all the ops.

Definition:

– Operation has amortized cost $T(n)$ if for every integer k, the cost of k operations is $\leq k\,T(n)$

Amortized Analysis

Definition:

- Operation has <u>amortized cost</u> $T(n)$ if for every integer k, the cost of k operations is $\leq k\,T(n)$

Example: amortized cost = 7

1. insert: 5
2. insert: 5
3. insert: 5
4. insert: 13
5. insert: 7

5 <= 7

5+5 <= 2*7 = 14

5+5+5 <= 3*7 = 21

5+5+5+13 <= 4*7 = 28

5+5+5+13+7 <= 5*7 = 35

Amortized Analysis

“amortized” is NOT “average”

Definition:

– Operation has <u>amortized cost</u> $\mathrm{T}(n)$ if for every integer k, the cost of k operations is $\leq k\,\mathrm{T}(n)$

Example: amortized cost **NOT** 7

1. insert: 13
2. insert: 5
3. insert: 5
4. insert: 5
5. insert: 7

13 > 7

13+5 > 2*7 = 14

13+5+5 > 3*7 = 21

13+5+5+5 <= 4*7 = 28

5+5+5+13+7 <= 5*7 = 35

Amortized Analysis

Definition:

– Operation has amortized cost $T(n)$ if for every integer k, the cost of k operations is $\leq k\,T(n)$

Example: (Hash Tables)

– Inserting k elements into a hash table with resizing takes time $O(k)$.

– Conclusion:

The insert operation has amortized cost $O(1)$.

Amortized Analysis

Accounting Method

- Imagine a bank account **B**.
- Each operation adds money to the bank account.
- Every step of the algorithm spends money:
  - Immediate money: to perform the operation.
  - Deferred money: from the bank account.
- Total cost execution = total money
  - Average time / operation = money / num. ops

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account, uses O(1) dollars to insert element.
- A table with *k* new elements since last resize has *k* dollars in bank.

0	null
1	null
2	**(k_1, A)**
3	null
4	null
5	null
6	null
7	null
8	**(k_2, B)**
9	null

Bank account
$2 dollars

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account.

- Claim:
  - Resizing a table of size m takes $O(m)$ time.
  - If you resize a table of size m, then:
    - at least $m/2$ new elements since last resize
    - bank account has $\Theta(m)$ dollars.

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account.
- Pay for resizing from the bank account!
- Strategy:
  - Analyze inserts ignoring cost of resizing.
  - Ensure that bank account always is big enough to pay for resizing.

Amortized Analysis

Total cost: Inserting k elements costs:

- Deferred dollars: $O(k)$ (to pay for resizing)
- Immediate dollars: $O(k)$ for inserting elements in table
- Total (Deferred + Immediate): $O(k)$

Amortized Analysis

Total cost: Inserting k elements costs:

- Deferred dollars: $O(k)$ (to pay for resizing)
- Immediate dollars: $O(k)$ for inserting elements in table
- Total (Deferred + Immediate): $O(k)$

Cost per operation:

- Deferred dollars: $O(1)$
- Immediate dollars: $O(1)$
- Total: $O(1)$ / per operation

Example: Binary Counter

Counter ADT:

- increment()
- read()

0	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Counter ADT:

- increment()
- read()

increment()

0	0	0	0	0	0	0	0	0	1

Example: Binary Counter

Counter ADT:

– increment()

– read()

increment(), increment()

0	0	0	0	0	0	0	0	1	0

Example: Binary Counter

Counter ADT:

- increment()
- read()

increment(), increment(), increment()

0	0	0	0	0	0	0	0	1	1

What is the worst-case cost of incrementing a counter with max-value n?

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n^2)$
5. I have no idea.

Example: Binary Counter

Counter ADT:

- increment()
- read()

Some increments are expensive…

0	1	1	1	1	1	1	1	1	1

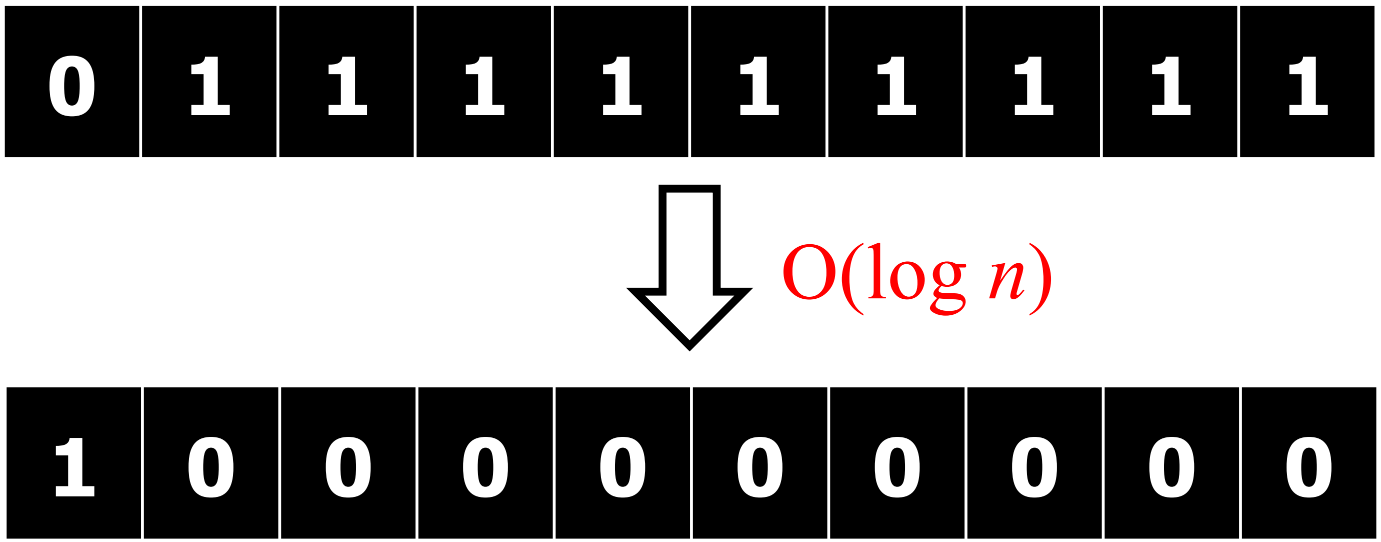
$\Downarrow$ O(log n)

1	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Question: If we increment the counter to n, what is the amortized cost per operation?

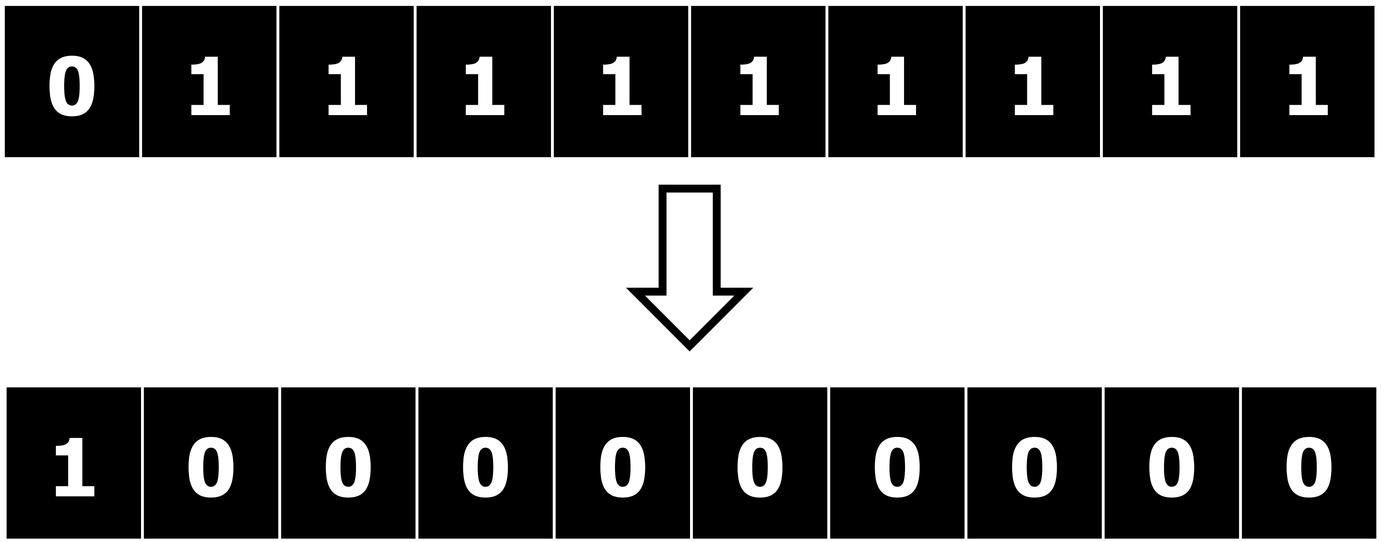
- Easy answer: $O(\log n)$
- More careful analysis….



Example: Binary Counter

Observation:

During each increment, only one bit is changed from: 0 → 1



Example: Binary Counter

Observation:

During each increment, <u>many</u> bits may be changed from: 1 → 0

0	1	1	1	1	1	1	1	1	1

$$\Downarrow$$

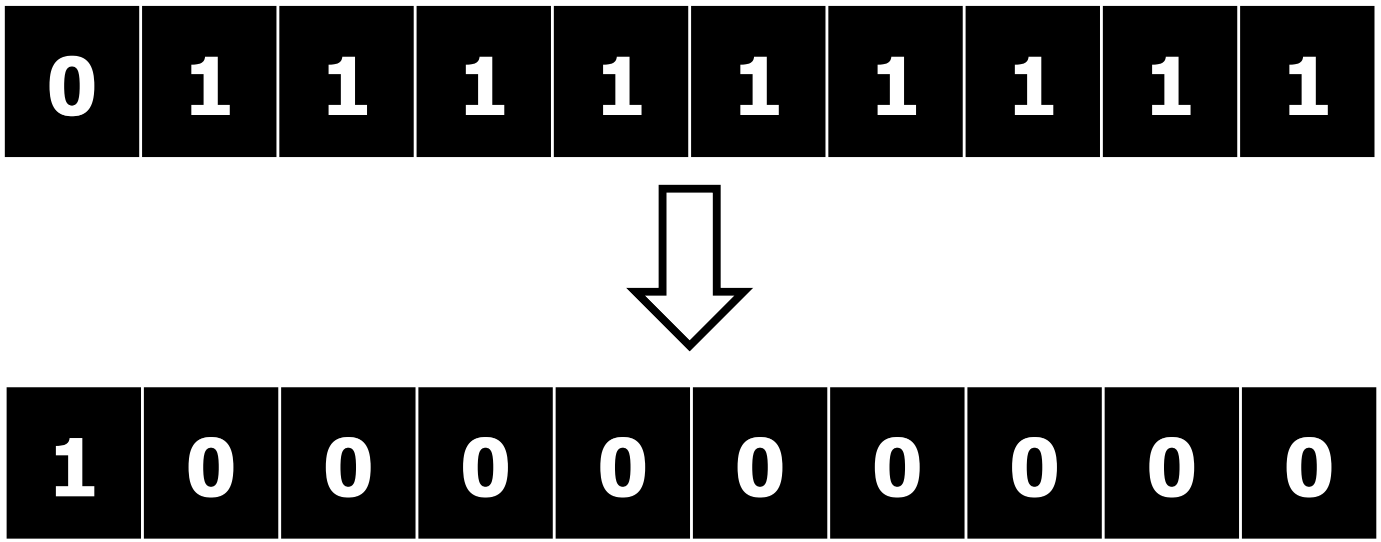
1	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

Whenever you change it from 0→1, add one dollar.



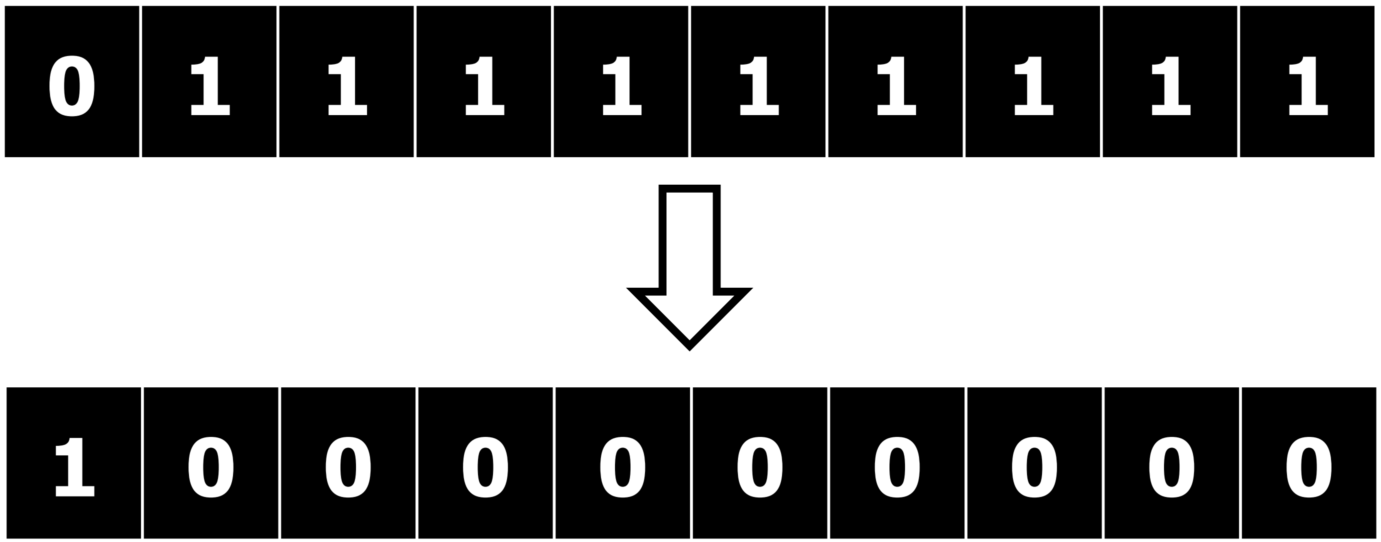
Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

Whenever you change it from 0→1, add one dollar.

Whenever you change it from 1→0, pay one dollar.



Example: Binary Counter

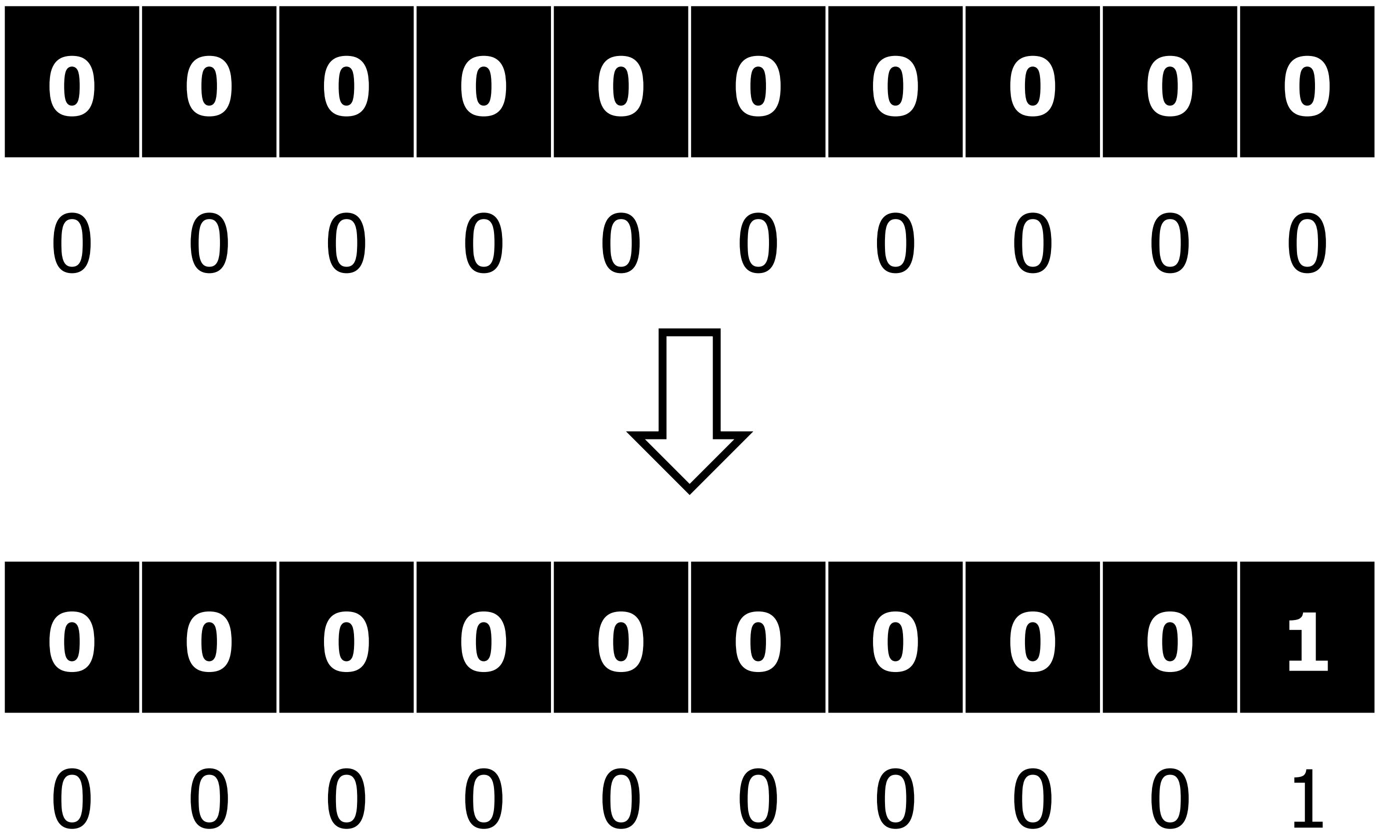
Counter ADT

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
0	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Counter ADT

increment()



Example: Binary Counter

Counter ADT

increment(), increment()

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
0	0	0	0	0	0	0	0	0	1

⇓

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
0	0	0	0	0	0	0	0	1	0

Example: Binary Counter

Counter ADT

increment(), increment(), increment()

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
0	0	0	0	0	0	0	0	1	0

⇩

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
0	0	0	0	0	0	0	0	1	1

Example: Binary Counter

Counter ADT

increment()

0	1	1	1	1	1	1	1	1	1

0 1 1 1 1 1 1 1 1 1

1	0	0	0	0	0	0	0	0	0

1 0 0 0 0 0 0 0 0 0

Example: Binary Counter

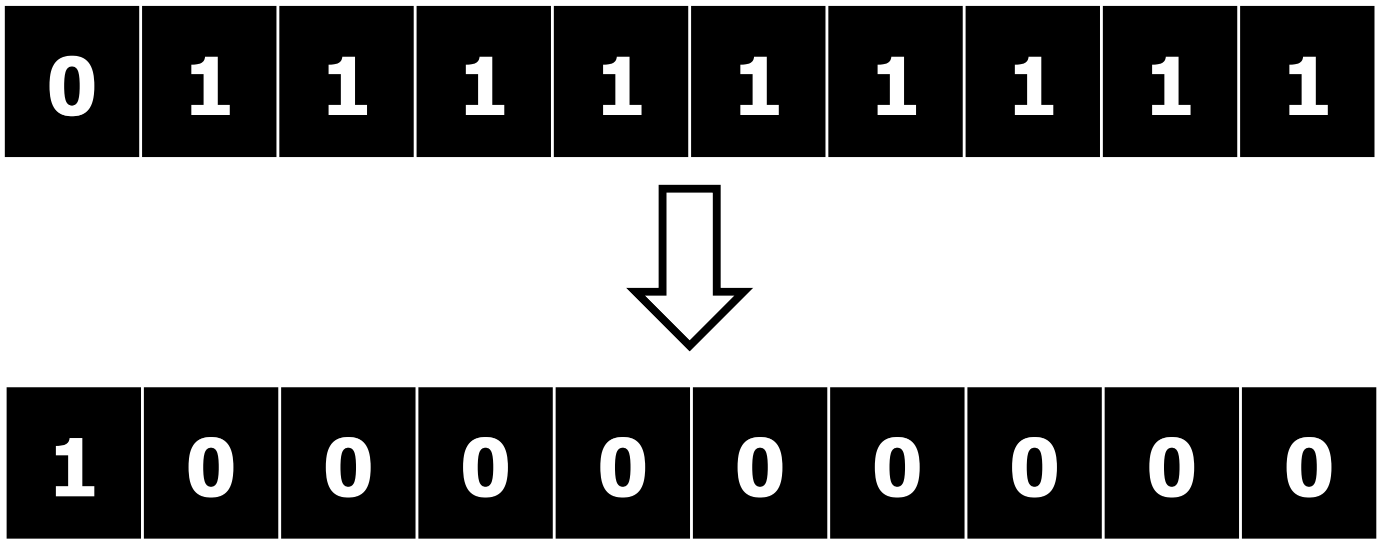
Observation:

Amortized cost of increment: 2

– One operation to switch <u>one</u> 0→1

– One dollar (for bank account of switched bit).

(All switches from 1→0 paid for by bank account.)



Design question

Do you care that some insertions take a lot longer than others?

- Most insertions: O(1)
- Some insertions: linear cost (expensive)
- Total cost is good…
- … but what if the slow operation is really, really important / time critical?
- What if YOUR online purchase is the one that triggers a two hour database rebuild?

Hash Table Resizing

Rules for shrinking and growing:

- If $(n == m)$, then $m = 2m$.
- If $(n < m/4)$, then $m = m/2$.

Key facts:

- Every time you double a table of size m, at least $m/2$ new items were added ➔ can pay for doubling.
- Every time you shrink a table of size m, at least $m/4$ items were deleted ➔ can pay for shrinking.

➔ Operations cost O(1) amortized, expected cost!

Hashing!

- Introduction to Hashing
- Collision Resolution: chaining
- Java hashing
- Collision resolution: open addressing
- Table (re)sizing