CS2040S
Data Structures and Algorithms

Hashing III

Hashing overview

- What is a hash function?
- Collision resolution: chaining
- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Hashing overview

- What is a hash function?
- 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.

Direct Access Tables

Attempt #1: Use a table, indexed by keys.

0	null
1	null
2	item1
3	null
4	null
5	item3
6	null
7	null
8	item2
9	null

Universe U={0..9} of size $m = 10$.

<u>(key, value)</u>

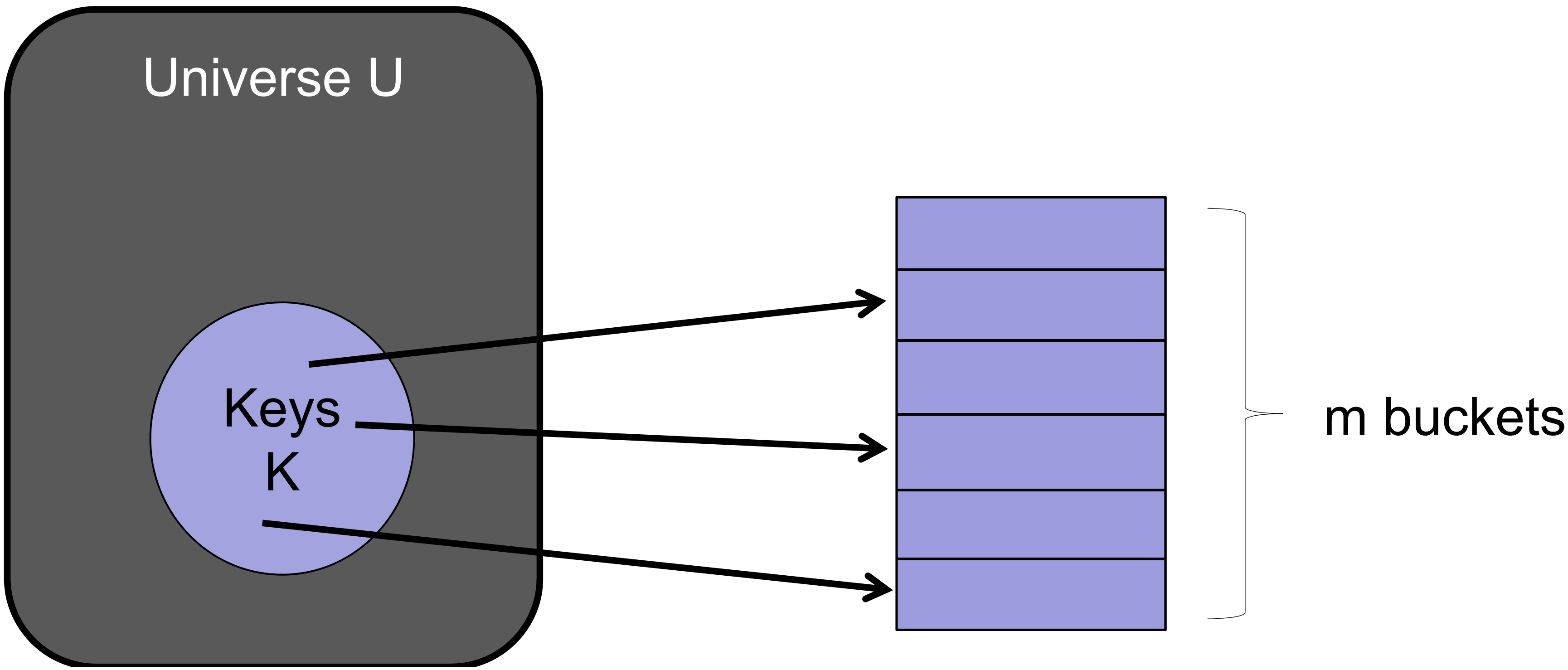
(2, item1)
(8, item2)
(5, item3)

Assume keys are distinct.

Hash Functions

Problem:

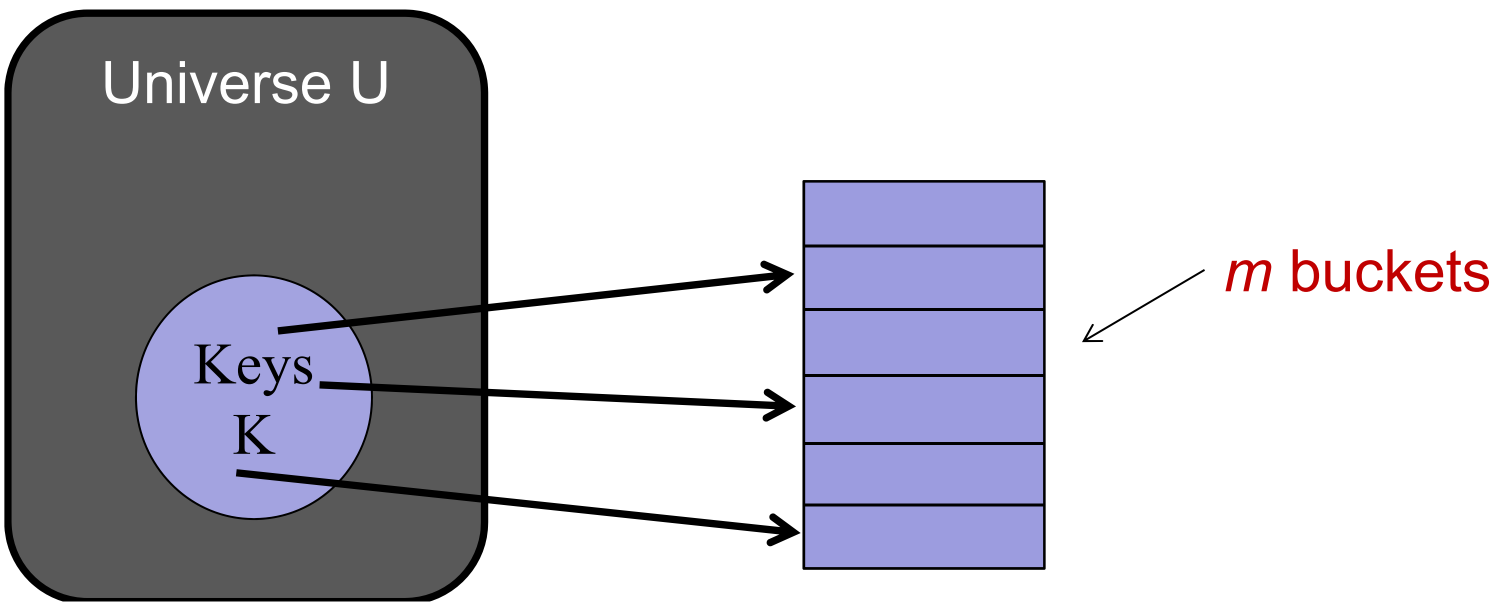
- Huge universe U of possible keys.
- Smaller number n of actual keys.
- How to map n keys to $m \approx n$ buckets?



Hash Functions

Define hash function $h : U \rightarrow \{1..m\}$

– Store key k in bucket $h(k)$.



Hash Functions

Collisions:

- We say that two <u>distinct</u> keys k_1 and k_2 collide if:

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

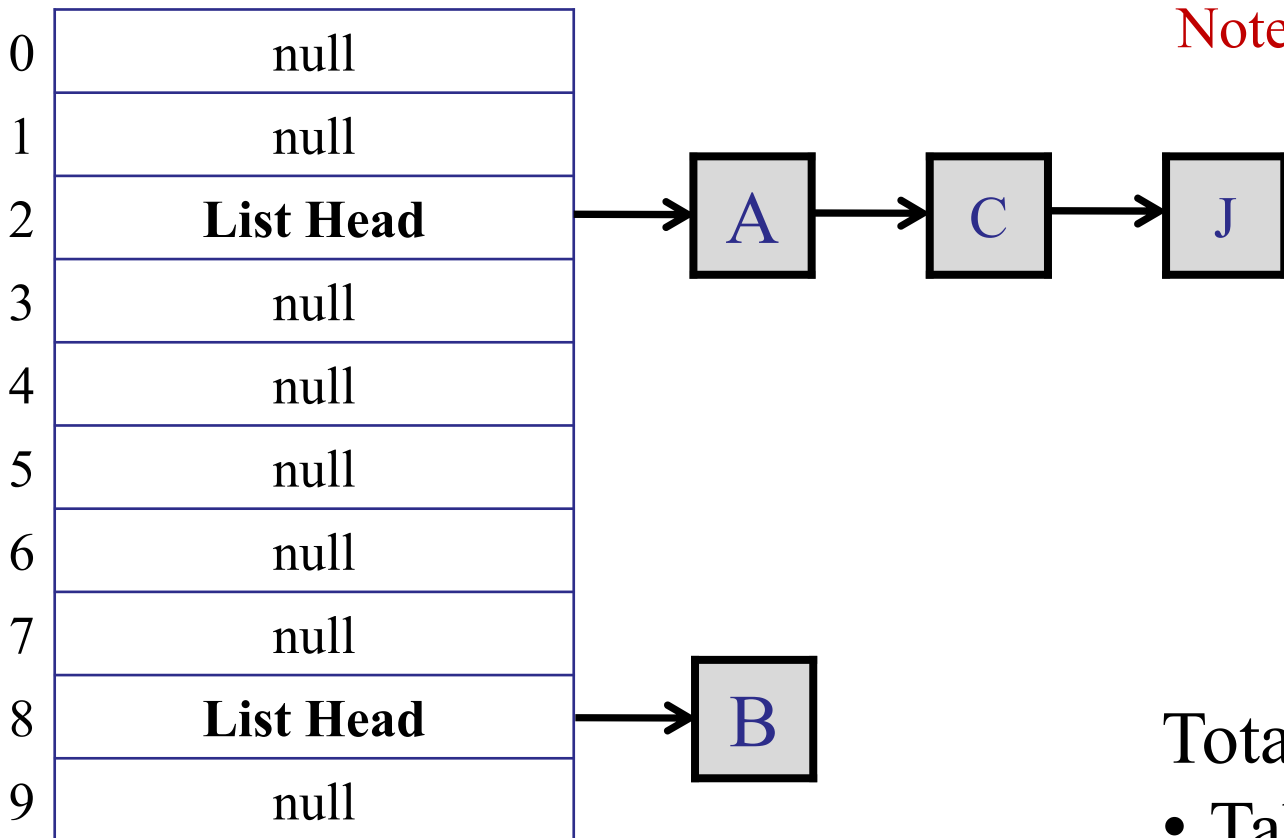
- Unavoidable!
  - The table size is smaller than the universe size.
  - The pigeonhole principle says:
    - There must exist two keys that map to the same bucket.
    - Some keys must collide!

Resolving Collisions

- Basic problem:
  - What to do when two items hash to the same bucket?
- Solution 1: Chaining
  - Insert item into a linked list.
- Solution 2: Open Addressing
  - Find another free bucket.

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.

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.

Let's be optimistic today.

The Simple Uniform Hashing Assumption

- Assume:
  - n items
  - m buckets
- Define: load(hash table) = n/m

  = average # items / buckets.

- Expected search time = $1 + n/m$

hash function + array access

linked list traversal

Let's be optimistic today.

The Simple Uniform Hashing Assumption

– Assume:
  - n items
  - $m = \Omega(n)$ buckets, e.g., $m = 2n$

– Expected search time $= 1 + n/m$

$$= O(1)$$

Resolving Collisions

- Basic problem:
  - What to do when two items hash to the same bucket?
- Solution 1: Chaining
  - Insert item into a linked list.
- Solution 2: Open Addressing
  - Find another free bucket.

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.

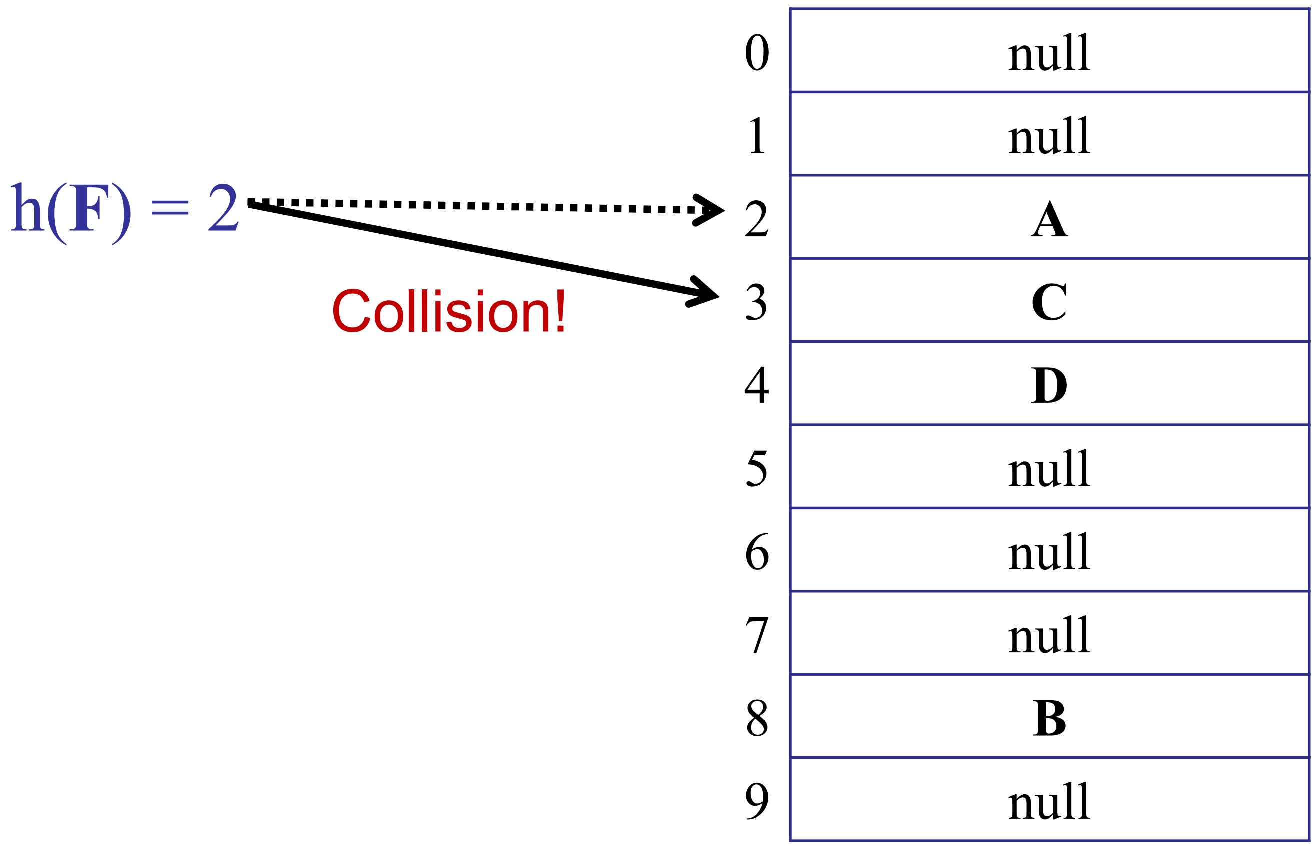
$h(\mathbf{F}) = 2 \xrightarrow{\text{Collision!}} 2$

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

Open Addressing

On collision:

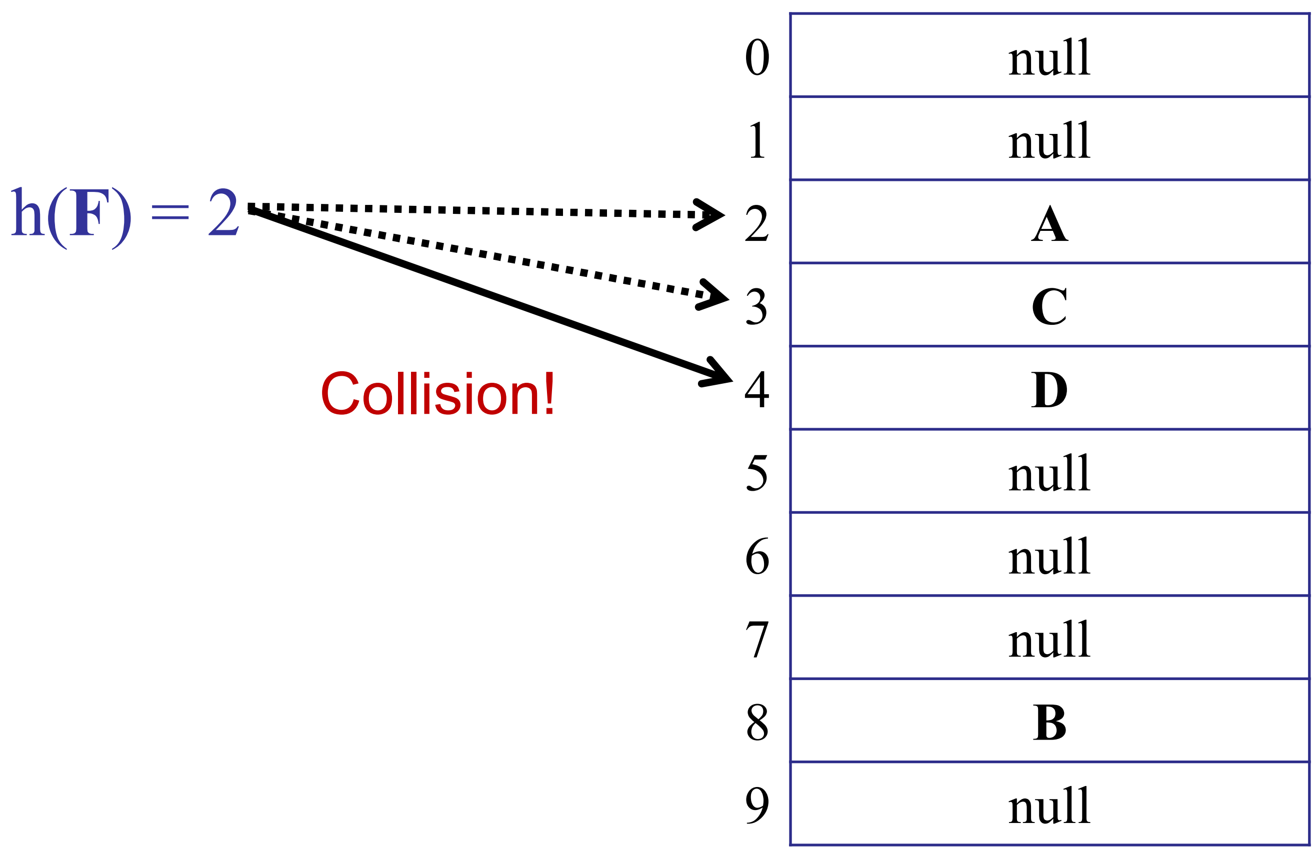
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

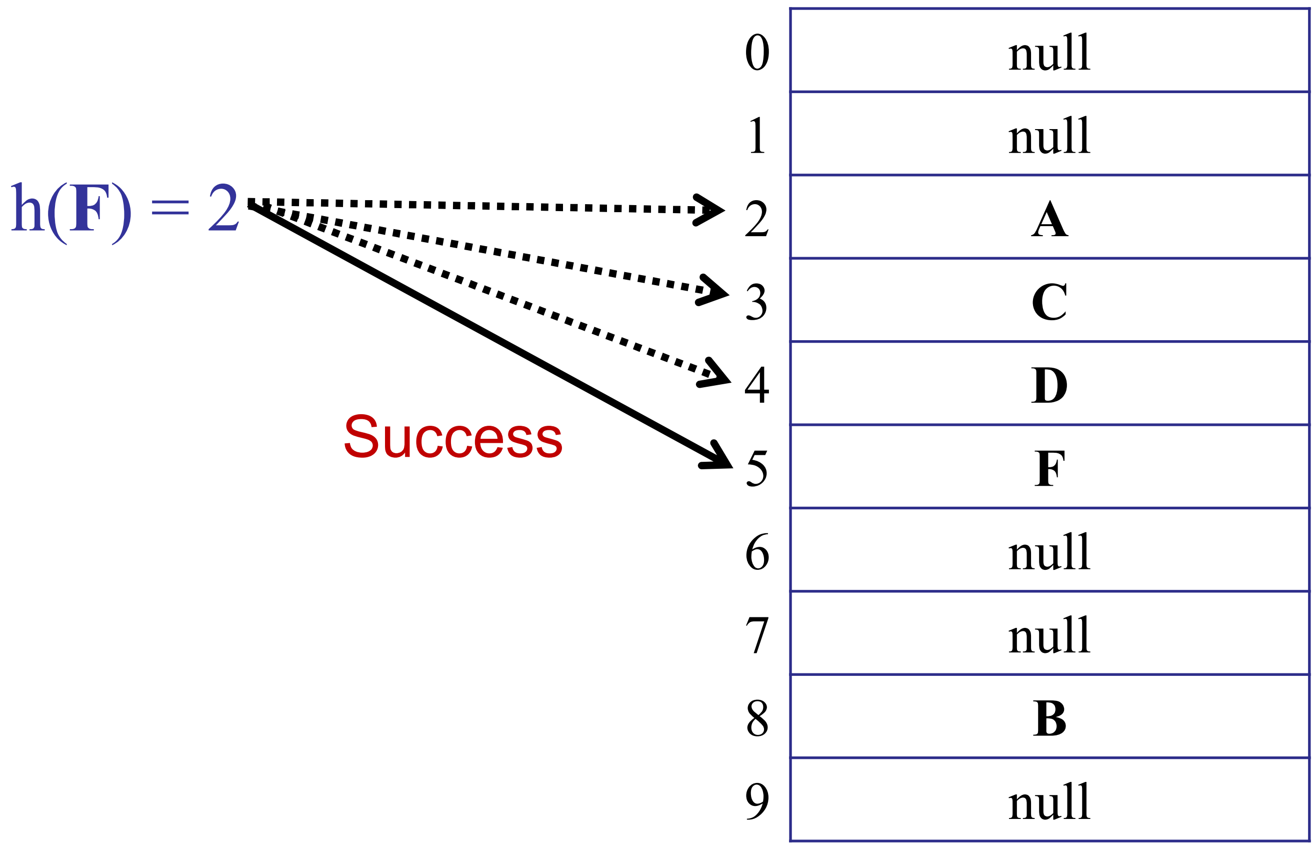
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

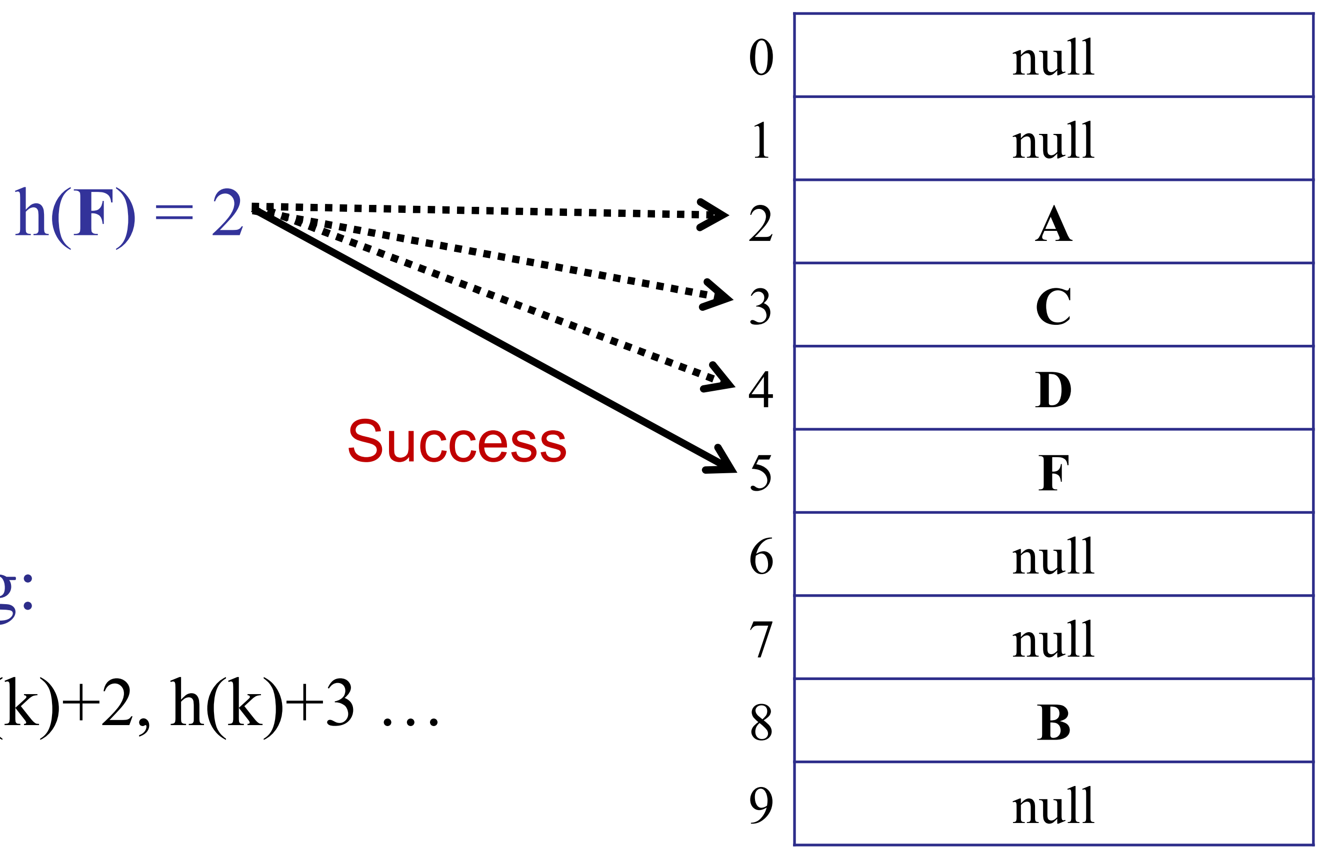
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

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



Linear Probing:

- h(k)+1, h(k)+2, h(k)+3 …

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

Two parameters:

- key : the thing to map
- i : number of collisions

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

Example: Linear Probing

- $h(k, 1)$ = hash of key k
- $h(k, 2) = h(k, 1) + 1$
- $h(k, 3) = h(k, 1) + 2$
- $h(k, 4) = h(k, 1) + 3$
- …
- $h(k, i) = h(k, 1) + i \bmod m$

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

Open Addressing

Hash Function re-defined:

$h(\text{key}, i) : U \rightarrow \{1..m\}$

Example: Weird Probing

- $h(k, 1) = 4$
- $h(k, 2) = 1$
- $h(k, 3) = 8$
- $h(k, 4) = 5$

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

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

search(key)

- $h(key, 1) = 4$
- $h(key, 2) = 1$
- $h(key, 3) = 8$
- $h(key, 4) = 5$

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

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

delete(key)

- Find key to delete
- Remove it from table.
- Set bucket to DELETED.

(Tombstone value.)

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

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!

What goes wrong if the sequence is not a permutation?

1. Search incorrectly returns key-not-found.
2. Delete fails.
3. Insert puts a key in the wrong place
4. Returns table-full even when there is still space left.

Hash Functions

Two properties of a good hash function:

2. Simple Uniform Hashing Assumption

   Every key is equally likely to be mapped to every bucket, independently of every other key.

For h(*key*, 1)?

For every h(*key*, *i*)?

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
- …

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 Pr(1/m)
- 1 2 4 3 Pr(0)
- 1 4 2 3 Pr(0)
- 1 4 3 2 Pr(0)
- …

NOT Linear Probing

Linear probing

In practice, linear probing is very fast!

- Why? 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: $\theta(\log n)$
  - Cache may hold entire cluster!
  - No worse than wacky probe sequence.

Open Addressing

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
- …

Double Hashing

- Start with two ordinary hash functions:

$$f(k),\ g(k)$$

- Define new hash function:

$$h(k, i) = f(k) + i \cdot g(k) \mod m$$

- Note:
  - Since f(k) is good, f(k, 1) is "almost" random.
  - Since g(k) is good, the probe sequence is "almost" random.

Double Hashing

Hash function

$$h(k, i) = f(k) + i \cdot g(k) \mod m$$

Claim: if g(k) is relatively prime to m, then $h(k, i)$ hits all buckets.

- Assume not: then for some distinct $i, j < m$:

$$f(k) + i \cdot g(k) = f(k) + j \cdot g(k) \mod m$$

$$\Rightarrow i \cdot g(k) = j \cdot g(k) \mod m$$

$$\Rightarrow (i - j) \cdot g(k) = 0 \mod m$$

$\Rightarrow$ g(k) not relatively prime to m, since $(i,j < m)$

Double Hashing

Hash function

$$h(k, i) = f(k) + i \cdot g(k) \mod m$$

Claim: if $g(k)$ is relatively prime to m, then $h(k, i)$ hits all buckets.

Example: if $(m = 2^r)$, then choose $g(k)$ odd.

Performance of Open Addressing

If (m==n), what is the expected insert time, under uniform hashing assumption?

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

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 $\alpha < 1$.

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:

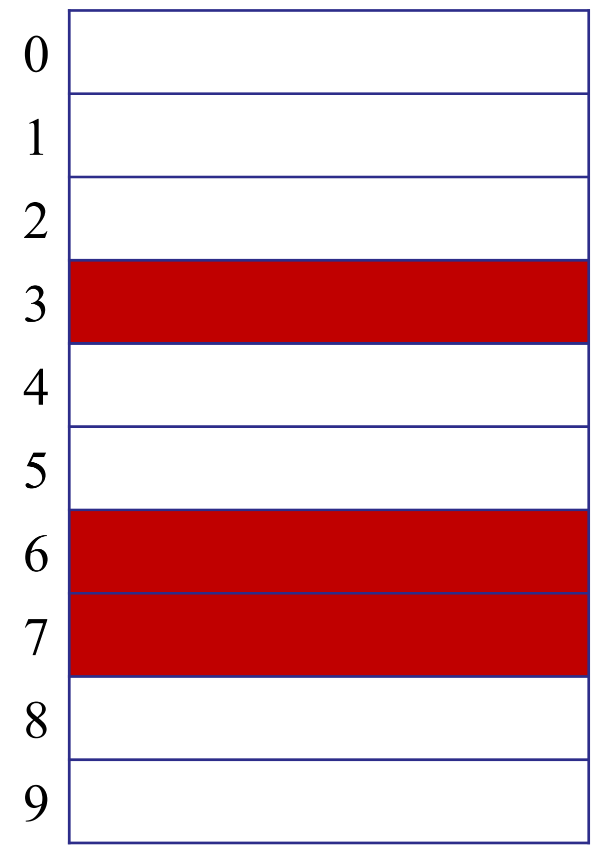
$$\leq \frac{1}{1-\alpha}$$

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

Performance of Open Addressing

Proof of Claim:

– First probe: probability that first bucket is full is: *n/m*

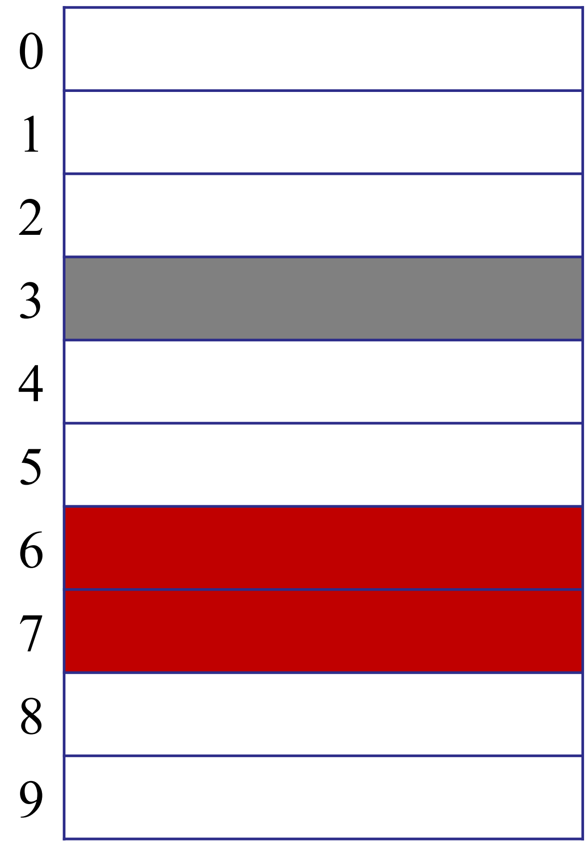


Performance of Open Addressing

Proof of Claim:

- First probe: probability that first bucket is full is: n/m

- Second probe: if first bucket is full, then the probability that the second bucket is also full: $(n-1)/(m-1)$



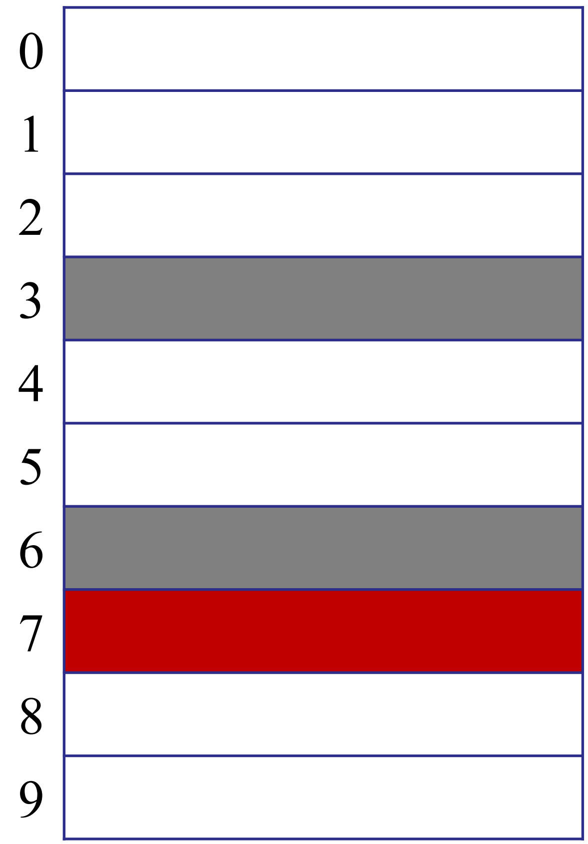
Performance of Open Addressing

Proof of Claim:

- First probe: probability that first bucket is full is: n/m

- Second probe: if first bucket is full, then the probability that the second bucket is also full: $(n - 1) / (m - 1)$

- Third probe: probability is full: $(n - 2) / (m - 2)$



Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$1+\frac{n}{m}\left(\text{Expected cost of remaining probes}\right)$$

First probe

Probability
of collision
on first probe

Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(\cdots\right)\right)$$

Expected cost of remaining probes

First probe

Probability of collision on first probe

Probability of collision on second probe

Performance of Open Addressing

Proof of Claim:

- Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\right)\right)\right)$$

First probe Second probe Third probe

Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\right)\right)\right)$$

– Note:

$$\frac{n-i}{m-i}\leq\frac{n}{m}\leq\alpha$$

Performance of Open Addressing

Proof of Claim:

- Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdot \cdot \cdot \right)\right)\right)$$

$$\leq 1+\alpha\left(1+\alpha\left(1+\alpha\left(\cdots\cdots\right)\right)\right)$$

Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\right)\right)\right)$$

$$\leq 1+\alpha\left(1+\alpha\left(1+\alpha\left(\cdots\cdots\right)\right)\right)$$

$$\leq 1+\alpha+\alpha^2+\alpha^3+\cdots$$

Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\square\ \square\ \square\right)\right)\right)$$

$$\leq 1+\alpha\left(1+\alpha\left(1+\alpha\left(\cdots\cdots\right)\right)\right)$$

$$\leq 1+\alpha+\alpha^2+\alpha^3+\cdots$$

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

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

Advantages…

Open addressing:

- Saves space
  - Empty slots vs. linked lists.
- Rarely allocate memory
  - No new list-node allocations.
- Better cache performance
  - Table all in one place in memory
  - Fewer accesses to bring table into cache.
  - Linked lists can wander all over the memory.

Disadvantages…

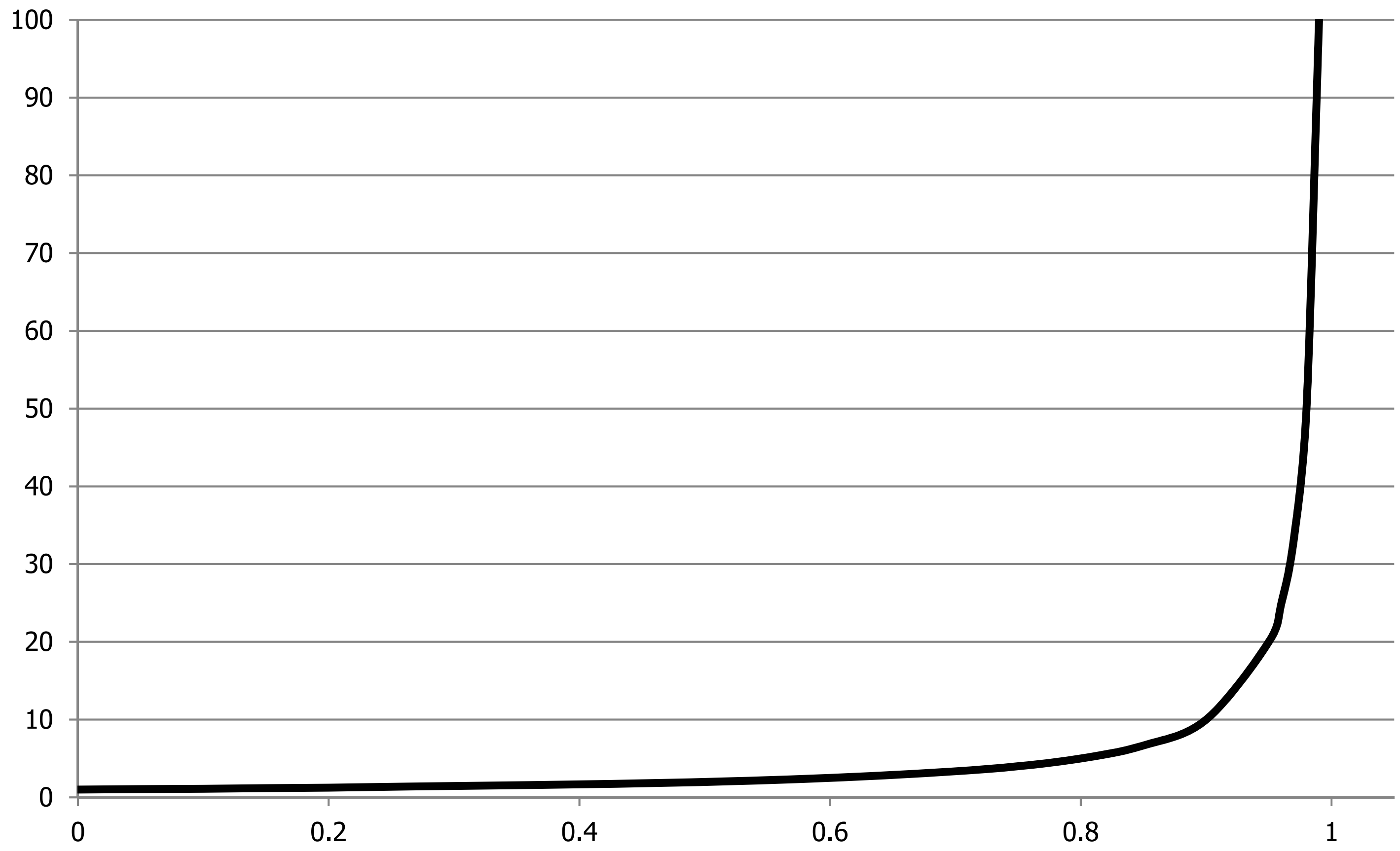
Open addressing:

- More sensitive to choice of hash functions.
  - Clustering is a common problem.
  - See issues with linear probing.
- More sensitive to load.
  - Performance degrades badly as $\alpha \rightarrow 1$.

Disadvantages…

Open addressing:

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



Hashing overview

- What is a hash function?
- 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.

Not like Java hashCode!

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: $m = 8$
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		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)

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.

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.

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
- Like paying rent:
  - You don't pay rent every day!
  - Pay \$900/month = \$30/day.

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 takes time $O(k)$.
- Conclusion:

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

Amortized Analysis

Accounting Method (paying rent)

- 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?

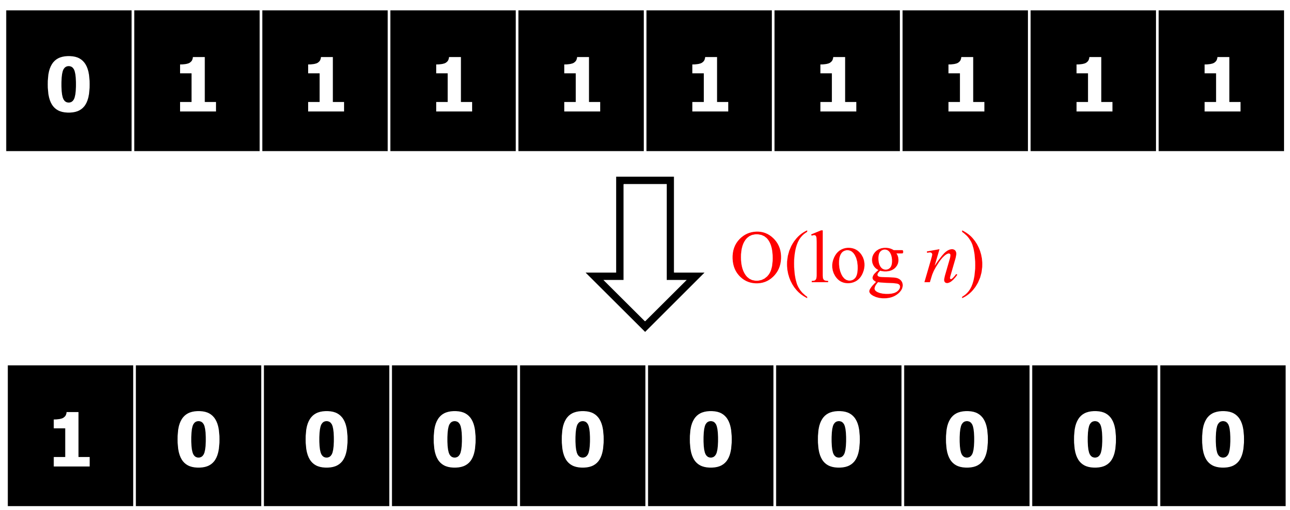
1. O(1)
2. O(log n)
3. O(n)
4. O(n^2)
5. I have no idea.

Example: Binary Counter

Counter ADT:

- increment()
- read()

Some increments are expensive…

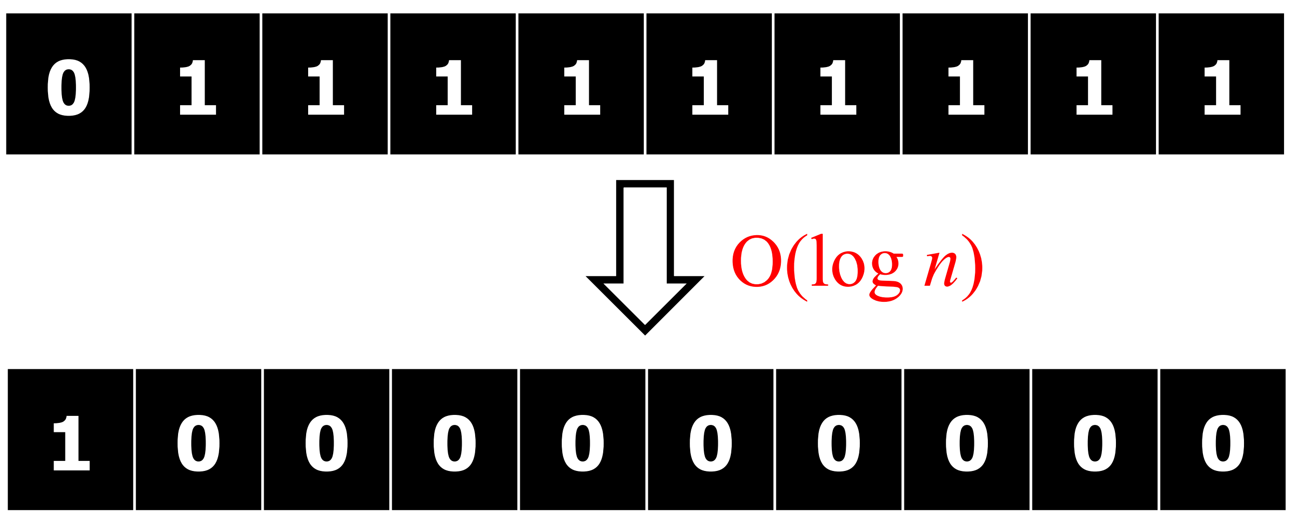


Example: Binary Counter

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

– Easy answer: $O(\log n)$

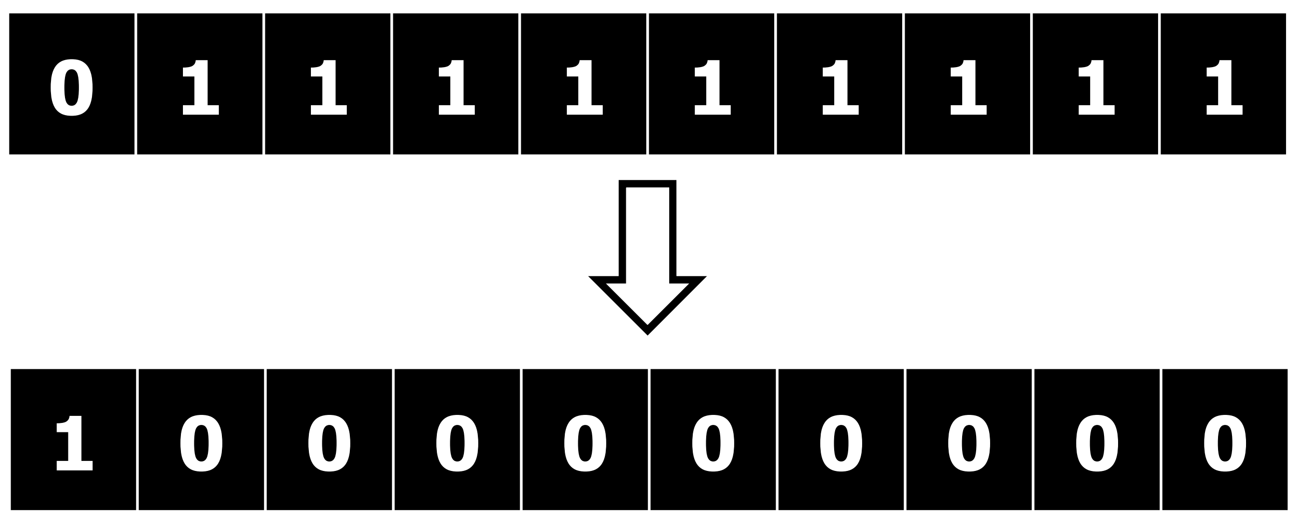
– More careful analysis….



Example: Binary Counter

Observation:

During each increment, only <u>one</u> 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

⇓

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.

0	1	1	1	1	1	1	1	1	1

$$\Downarrow$$

1	0	0	0	0	0	0	0	0	0

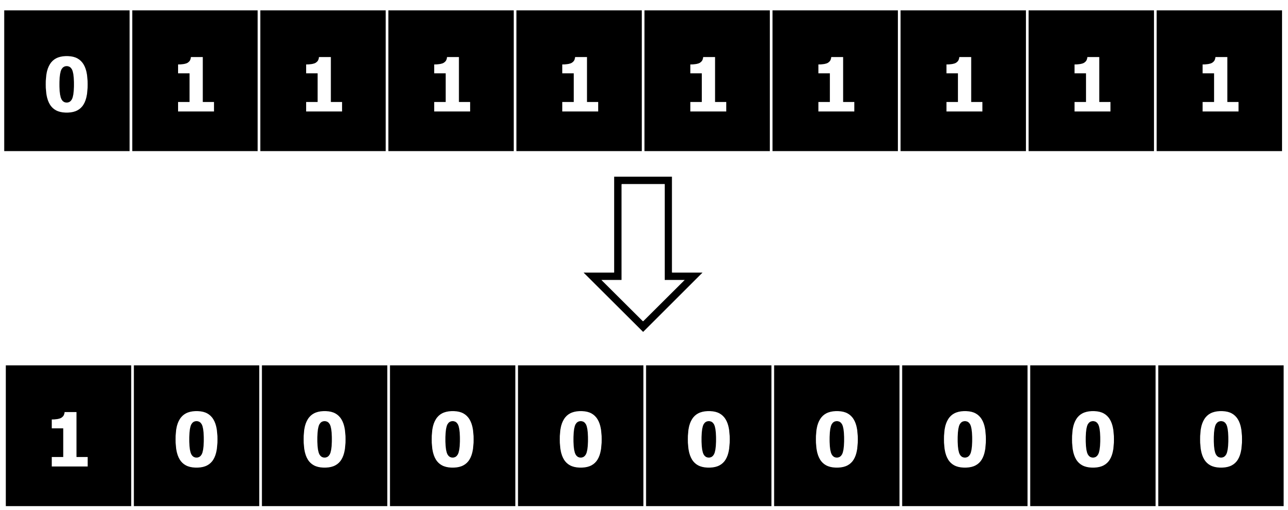
Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

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

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()

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

$\Downarrow$

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

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.)

0	1	1	1	1	1	1	1	1	1

⇩

1	0	0	0	0	0	0	0	0	0

Table Size Rules

Rules for shrinking and growing:

- 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

Accounting Method

- 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

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

Hash Table Resizing

Conclusion: Hashing with Chaining

– with Simple Uniform Hashing Assumption (SUHA)

Cost per operation:

– Insert operation: amortized O(1)

– Search operation: expected O(1)

Notes:

– Inserts are amortized because of table resizing.

– Inserts are not randomized (because no searching for duplicates).

– Searches are expected (but not amortized) since no resizing on a search.

Hashing overview

- What is a hash function?
- Collision resolution: chaining
- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Reality Fights Back

Simple Uniform Hashing doesn't exist.

- Keys are not random.
  - Lots of regularity.
  - Mysterious patterns.
- Patterns in keys can induce patterns in hash functions unless you are very careful.

Problem Hash Functions

Example:

- One bucket for each letter [a..z]
- Hash function: h(string) = first letter.
    - E.g., h(“hippopotamus”) = h.

- Bad hash function: why??

Problem Hash Functions

Example:

- One bucket for each letter [a..z]
- Hash function: h(string) = first letter.
  - E.g., h(“hippopotamus”) = h.

- Bad hash function: many fewer words start with the letter *x* than start with the letter *s*.

Problem Hash Functions

Example:

- One bucket for each number from [1..26*28]
- Hash function: h(string) = sum of the letters.
  - E.g., h("hat") = 8 + 1 + 20 = 29.

- Bad hash function: why??

Problem Hash Functions

Example:

- One bucket for each number from [1..26*28]
- Hash function: h(string) = sum of the letters.
  - E.g., h("hat") = 8 + 1 + 20 = 29.

- Bad hash function: lots of words collide, and you don't get a uniform distribution (since most words are short).

Problem Hash Functions

But pretty good hash functions do exist…

- Optimism pays off!

Moral of the story:

- Don't design your own hash functions.
- Ever.
- Unless you really need to.

Designing Hash Functions

Goal: find a hash function whose values *look* random.

- Similar to pseudorandom generators:
  - When you use Java random, there is no real randomness.
  - Instead, it generates a sequence of numbers that looks random.
- For every hash function, some set of keys is bad!
- If you know the keys in advance, you can choose a hash function that is always good!
  - But if you change the keys, then it might be bad again.

Designing Hash Functions

Two common hashing techniques…

- Division Method
- Multiplication Method

Designing Hash Functions

Division Method

- $h(k) = k \bmod m$
  - For example: if m=7, then h(17) = 3
  - For example: if m=20, then h(100) = 0
  - For example: if m=20, then h(97) = 17

- Two keys k_1 and k_2 collide when:

$$k_1 = k_2 \bmod m$$

- Collision unlikely if keys are random.

Designing Hash Functions

Division Method

– (Bad) idea: choose $m = 2^x$

Very fast to calculate $k \bmod m$ via shifts

Recall: 001001 >> 1 = 00100

001001 >> 2 = 0010

001001 >> 3 = 001

Designing Hash Functions

Division Method

- (Bad) Idea: choose $m = 2^x$

  Very fast to calculate $k \bmod m$ via shifts:

$$\mathrm{k} \bmod 2^x = \mathrm{k} - ((k >> x) << x)$$

Designing Hash Functions

Division Method

- (Bad) Idea: choose $m = 2^x$

  Very fast to calculate $k \bmod m$ via shifts

- Problem: Regularity
  - Input keys are often regular
  - Assume input keys are even.
  - Then $h(k) = k \bmod m$ is even!

$$k \bmod m + \underbrace{i(m)}_{\text{even}} = \underbrace{k}_{\text{even}}$$

Designing Hash Functions

Division Method

– Assume k and m have common divisor d.

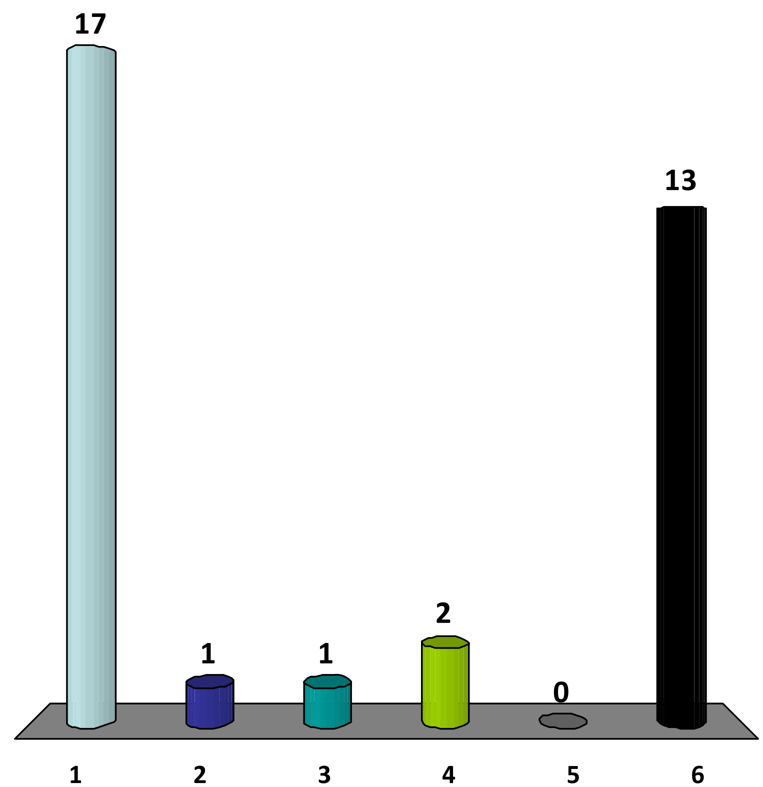
$$\underbrace{k \bmod m + i * m = k}$$

divisible by d

– Implies that $\mathrm{h}(k) = k \bmod m$ is divisible by d.

If d is a divisor of m and every key k, then what percentage of the table is used?

1. $1/d$
2. $1/k$
3. $1/m$
4. d/n
5. m/n
6. d/m



Designing Hash Functions

Division Method

- Assume *k* and *m* have common divisor *d*.

$$k \bmod m + i * m = k$$

divisible by *d*

- Implies that h(*k*) is divisible by *d*.

- If all keys are divisible by *d*, then you only use 1 out of every *d* slots

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

Designing Hash Functions

Division Method

- $h(k) = k \bmod m$
- Choose m = prime number
  - Not too close to a power of 2.
  - Not too close to a power of 10.
- Division method is popular (and easy), but not always the most effective.
- Division is slow.

Designing Hash Functions

Two common hashing techniques…

- Division Method
- Multiplication Method

Designing Hash Functions

Multiplication Method

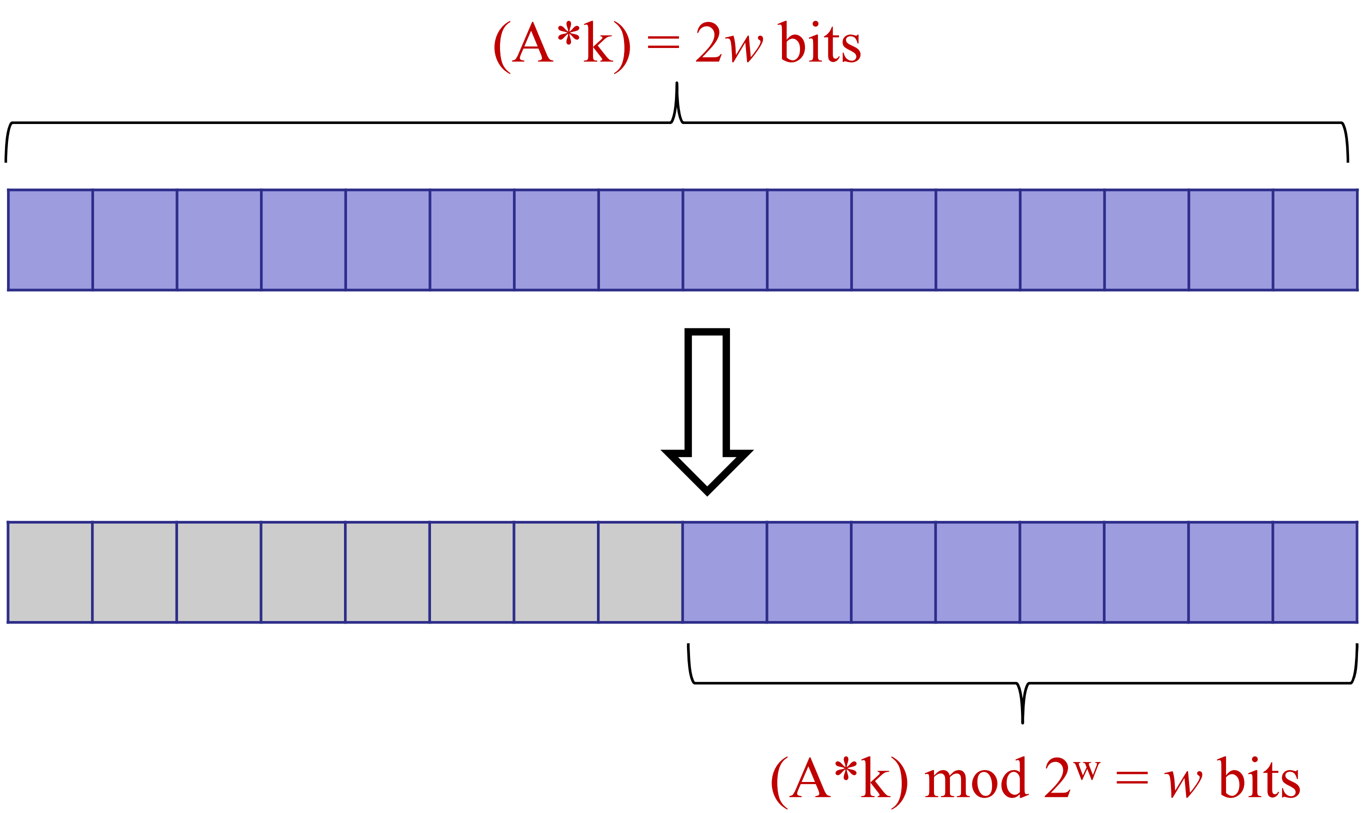
- Fix table size: $m = 2^r$, for some constant r.
- Fix word size: w, size of a key in bits.
- Fix (odd) constant A.

$$h(k) = (Ak) \bmod 2^w \gg (w - r)$$

Designing Hash Functions

Multiplication Method

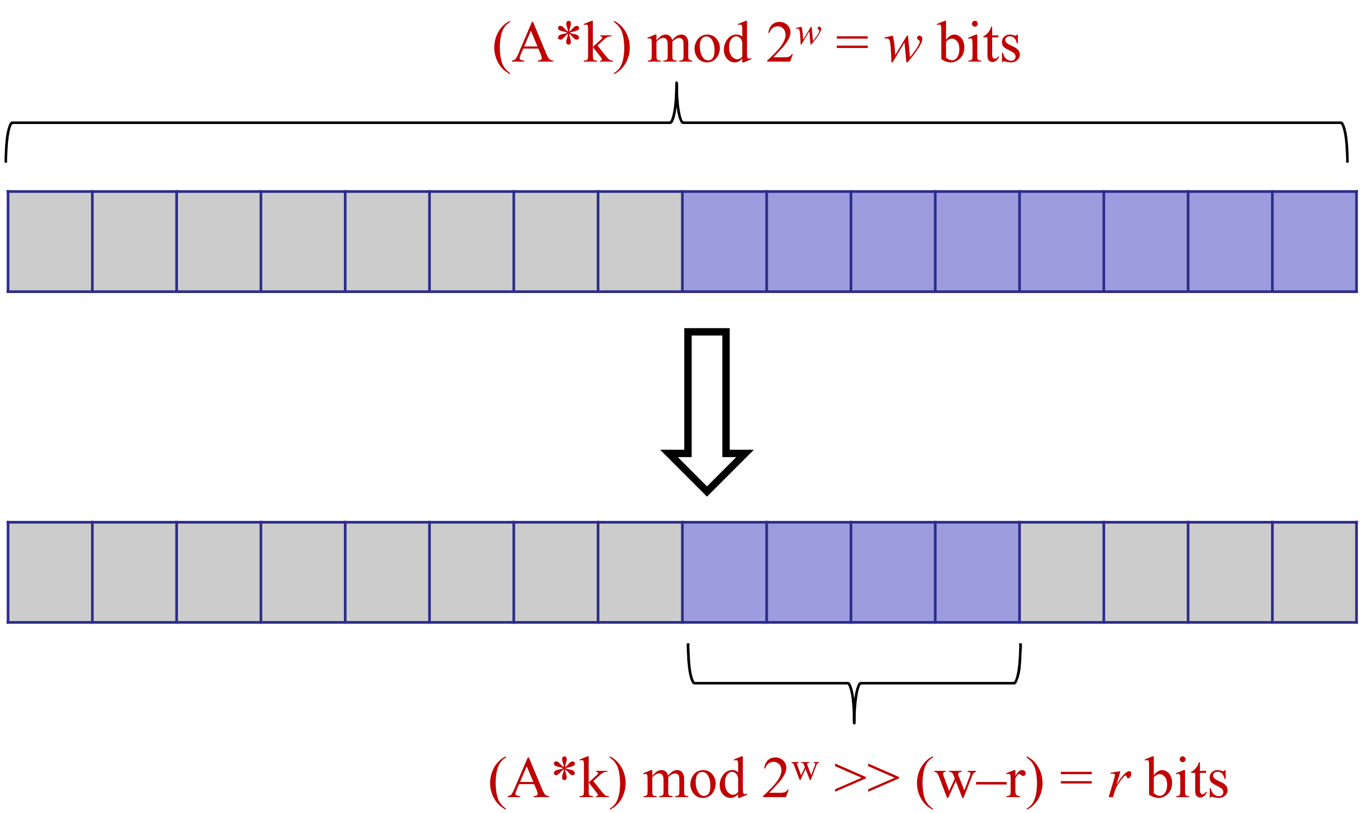
- Given m, w, r, A: $h(k) = (Ak) \bmod 2^w \gg (w - r)$



Designing Hash Functions

Multiplication Method

- Given m, w, r, A: $h(k) = (Ak) \bmod 2^w \gg (w - r)$



Designing Hash Functions

Multiplication Method

- Faster than Division Method
  - Multiplication, shifting faster than division
- Works reasonably well when A is an odd integer $> 2^{w-1}$
  - Odd: if it is even, then lose at least one bit's worth
  - Big enough: use all the bits in A.

Hashing overview

- What is a hash function?
- Collision resolution: chaining
- Java hashing
- Collision resolution: open addressing
- Table (re)sizing