CS2020
Data Structures and Algorithms

Welcome!

As of today:

You have completed CS1020.

Semester Roadmap

Where are we?

CS1020

- Searching / Sorting
- Lists
- Trees
- **Hash Tables**

You are here

CS2010

- Graphs
  - Shortest paths
  - Spanning trees
  - Directed acyclic graphs
- Advanced material

Coding Quiz

- Date: March 7/8/10
  - Administered during your Discussion Group
  - Location: TBA
  - Do not skip Discussion Group

- Practice problems:
  - See sample problems
  - See last year's Coding Quiz
  - Talk to your tutor.
  - If you want more practice problems, ask.

Mock Coding Quiz

- Date: Today
  - Time: 2pm and 4pm (two slots)
  - Location: COM1-0120
  - Practice, practice, practice…

Administrative

Advice:

- Coding under time pressure is hard.
  - Don’t rush: read the problem carefully.
  - Don’t rush: plan before you code.
  - Document your code as you go.
  - Don’t get stuck if something doesn’t work.

- Use your time wisely.

Administrative

Advice:

- Test your solution
  - Working code is important.
  - Test "corner-cases."
- Several possible solutions
  - First, ignore efficiency.
  - Develop a solution that works.
  - Test it. Test it. Test it.
  - Then, improve the efficiency.

Administrative

Advice:

- Use good coding style
  - Deductions for code that is badly formatted
- Explain your solution
  - Credit for well-documented code.

Administrative

Advice:

- Don’t submit code that does not even compile!

  If you can not solve the problem correctly, then

  submit simple code that solves the problem simply.

Coding Quiz

- Date: March 7/8/10
  - Administered during your Discussion Group
  - Location: TBA
  - Do not skip Discussion Group

- Practice problems:
  - See sample problems
  - See last year's Coding Quiz
  - Talk to your tutor.
  - If you want more practice problems, ask.

Plan: this week and next

3 Lectures on Hashing

- Applications
- Basic theory
- Handling collisions
- Hashing in Java
- Amortized analysis (doubling/shrinking)
- Sets and Bloom filters

Topic of the Week: Hash Tables

Abstract Data Types

Symbol Table

```
public interface  SymbolTable<Key, Value>
```

`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.

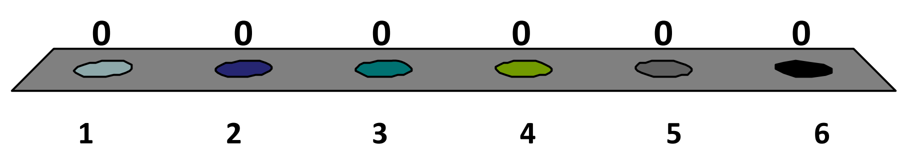
Symbol Table

Examples:

Dictionary:	key = word value = definition
Phone Book	key = name value = phone number
Internet DNS	key = website URL value = IP address
Java compiler	key = variable name value = type and value

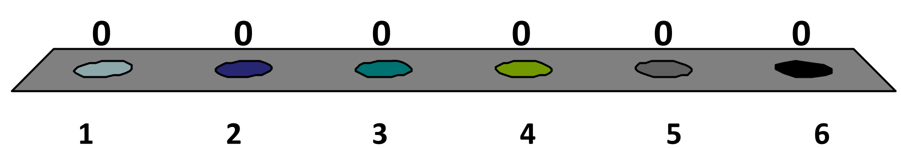
Implement a symbol table with a Linked List:
(C_I= cost insert, C_S=cost search)

1. $C_I = O(1)$, $C_S = O(1)$
2. $C_I = O(1)$, $C_S = O(\log n)$
3. $C_I = O(1)$, $C_S = O(n)$
4. $C_I = O(\log n)$, $C_S = O(\log n)$
5. $C_I = O(n)$, $C_S = O(\log n)$
6. $C_I = O(n)$, $C_S = O(n)$



Implement symbol table with an AVL tree:
(C_I= cost insert, C_S=cost search)

1. $C_I = O(1)$, $C_S = O(1)$
2. $C_I = O(1)$, $C_S = O(\log n)$
3. $C_I = O(1)$, $C_S = O(n)$
4. $C_I = O(\log n)$, $C_S = O(\log n)$
5. $C_I = O(n)$, $C_S = O(\log n)$
6. $C_I = O(n)$, $C_S = O(n)$



Symbol Table

Implement a symbol table with:

- $C_I = O(1)$
- $C_S = O(1)$

Fast, fast, fast….

Dictionaries vs. Symbol Tables

What can you do with a dictionary but not a symbol table?

Dictionaries vs. Symbol Tables

Sorting with a dictionary:

1) Insert every item into the dictionary.

2) Search for the minimum item.

3) Repeat: find successor

Running time to implement sorting:

With an AVL tree/dictionary?

With a symbol table?

Dictionaries vs. Symbol Tables

Sorting with a dictionary:

1) Insert every item into the dictionary.

2) Search for the minimum item.

3) Repeat: find successor

Running time to implement sorting:

With an AVL tree/dictionary? O(n log n)

With a symbol table? O(n^2)

Sorting (aside)

Isn't O(1) impossible?

Sorting takes $\Omega(n \log n)$ comparisons.

- How do you sort with a symbol table?
- Only search/insert/delete.

Sorting (aside)

Isn't O(1) impossible?

Sorting takes $\Omega(n \log n)$ comparisons.

- How do you sort with a symbol table?
- Only search/insert/delete.

(Binary) search takes $\Omega(\log n)$.

- Impossible to search in fewer than $\log(n)$ comparisons.
- But a symbol table finds an item in O(1) steps!!
- Conclusion: symbol table is not ***comparison-based***.

Symbol Tables in Java

Symbol Tables in Java

java.util.Map

public interface java.util.Map<Key, Value>

void	clear()	*removes all entries*
boolean	containsKey(Object k)	*is k in the map?*
boolean	containsValue(Object v)	*is v in the map?*
Value	get(Object k)	*get value for k*
Value	put(Key k, Value v)	*adds (k,v) to table*
Value	remove(Object k)	*remove mapping for k*
int	size()	*number of entries*

Note: no successor / predecessor queries.

Map Interface in Java

java.util.Map<Key, Value>

- No duplicate keys allowed.
- No *mutable* keys
  - If you use an *object* as a key, then you can't modify that object later.

Symbol Tables in Java

java.util.Map

```
        public interface  java.util.Map<Key, Value>
------------------------------------------------------------------------
Set<Map.Entry<Key, Value>>  entrySet()     set of all mappings

                  Set<Key>  keySet()       set of all keys

         Collection<Value>  values()       collection of all values
```

Note: not sorted

not necessarily efficient to work with these sets/collections.

What is wrong here?

Example:

```
Map<String, Integer> ageMap = new Map<String, Integer>();

ageMap.put("Alice", 32);
ageMap.put("Bernice", 84);
ageMap.put("Charlie", 7);

Integer age = ageMap.get("Alice")
```

- Key-type: String
- Value-type: Integer

What is wrong here?

Example:

Map is an interface!
Cannot instantiate an interface.

```
Map<String, Integer> ageMap = new Map<String, Integer>();

ageMap.put(“Alice”, 32);
ageMap.put(“Bernice”, 84);
ageMap.put(“Charlie”, 7);

Integer age = ageMap.get(“Alice”)
```

- Key-type: String
- Value-type: Integer

Map Class in Java

Example: HashMap

```
Map<String, Integer> ageMap = new HashMap<String, Integer>();

ageMap.put("Alice", 32);
ageMap.put("Bernice", 84);
ageMap.put("Charlie", 7);

Integer age = ageMap.get("Alice");
System.out.println("Alice's age is: " + age + ".");
```

- Key-type: String
- Value-type: Integer

Map Class in Java

Example: HashMap

```
Map<String, Integer> ageMap = new HashMap<String, Integer>();

ageMap.put("Alice", 32);
ageMap.put("Bernice", null);
ageMap.put("Charlie", 7);

Integer age = ageMap.get("Bob");
if (age==null){
    System.out.println("Bob's age is unknown.");
}
```

- Returns “null” when key is not in map.
- Returns “null” when value is null.

Map Classes in Java

HashMap

- containsKey
- containsValue
- entrySet
- get
- isEmpty
- keySet
- put
- putAll
- remove
- values

TreeMap

- containsKey
- containsValue
- entrySet
- get
- isEmpty
- keySet
- put
- putAll
- remove
- values

Map Classes in Java

HashMap

TreeMap

- ceilingEntry
- ceilingKey
- descendingKeySet
- firstEntry
- firstKey
- floorEntry
- floorKey
- headMap
- higherEntry
- higherKey
- … (and more)

Symbol Tables are Useful

Examples:

1. Spelling correction (key=misspelled word, data=word)
2. Scheme interpreter (key=variable, data=value)
3. Web server
   - Lots of simultaneous network connections.
   - When a packet arrives, give it to the right process to handle the connection.
   - key=ip address, data = connection handler

In this cases, O(log n) often isn’t fast enough!

Symbol Tables are Useful

Example 1: Pilot Scheduling

1. Check to see if feasible to schedule at time *t*.

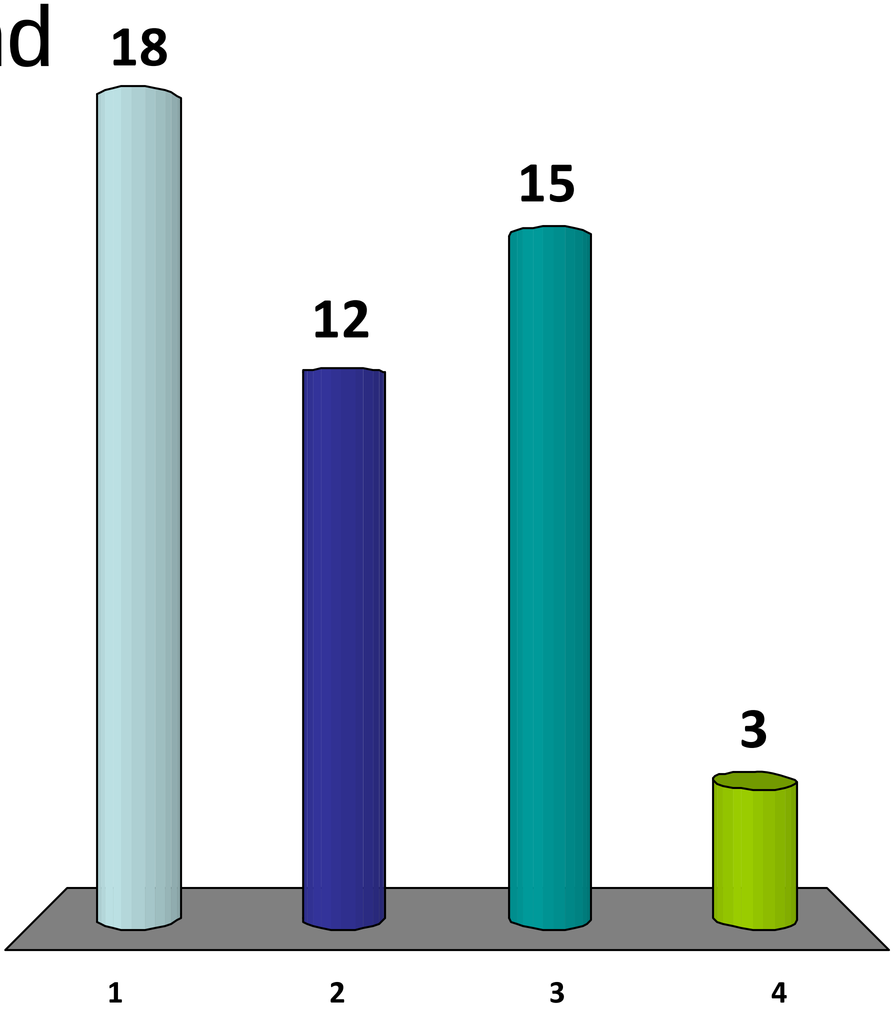
   No two airplanes can land with 3 minutes of each other.

2. Find schedule of pilot *p*.

   Get a list of all the planes that are being flown by a specified pilot.

Which can be efficiently solved with a symbol table?

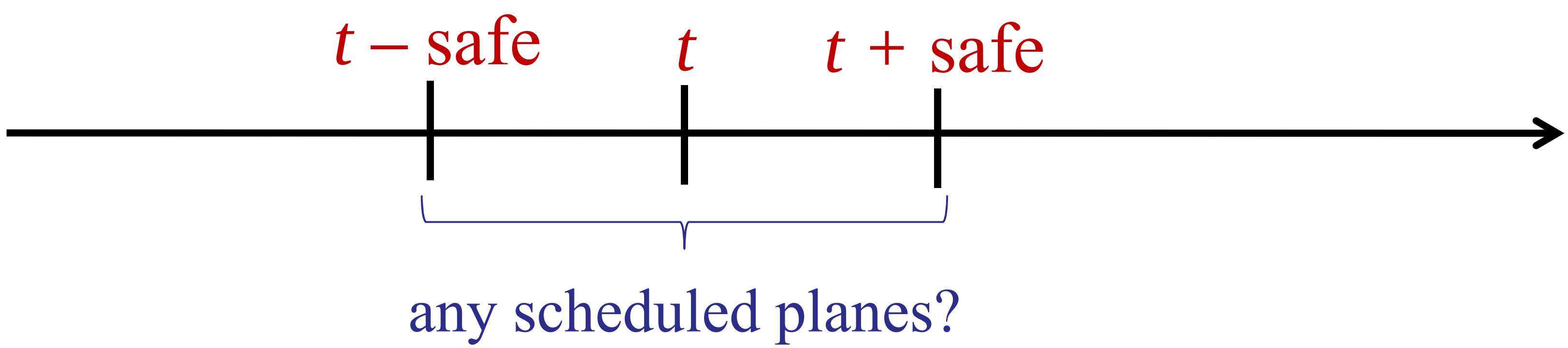
1. Both: scheduling and pilots info.
2. Only scheduling.
3. Only pilot info.
4. Neither.



Symbol Tables are Useful

Example 1: Pilot Scheduling

1. Check to see if feasible to schedule at time *t*.
   - Hard with a symbol table!
   - Need to find out if there are any planes scheduled in the interval [*t*, *t* ± safe_distance]



Symbol Tables are Useful

Example 1: Pilot Scheduling

1. Check to see if feasible to schedule at time *t*.
2. Find schedule of pilot *p*.
   - Perfect for a symbol table!
   - Can insert new pilots.
   - Can search for (and update) existing pilots.
   - Listing all pilots?

Symbol Tables are Useful

Example 2: Document Distance

- Given two documents A and B, how similar are they?
  - Two documents are *similar* if they have similar words in similar frequencies.
  - Formally, define each text as a vector with one entry per word.
  - The distance between the two texts is the angle between the two vectors.

Symbol Tables are Useful

Example 2: Document Distance

- Step 1: Read in each document
  - Read the file as a string.
  - Parse the file into words.
  - Sort the list of words.
  - Count the frequency of each word.
- Step 2: Compare the two documents
  - Calculate the norm $|A|$ and $|B|$ of each vector
  - Calculate the dot product AB.
  - Calculate the angle between A and B.

Symbol Tables are Useful

Example 2: Document Distance

- Step 1: Read in each document

$O(n)$	• Read the file as a string.
$O(n)$	• Parse the file into words.
$O(n \log n)$	• Sort the list of words.
$O(n)$	• Count the frequency of each word.

- Step 2: Compare the two documents

$O(n)$	• Calculate the norm $\|A\|$ and $\|B\|$ of each vector
$O(n)$	• Calculate the dot product AB.
$O(n)$	• Calculate the angle between A and B.

Performance Profiling (Sorting)

(Dracula vs. *Lewis & Clark)*

Step	Function	Running Time
Create vectors:	Read each file	1.03s
	Parse each file	1.23s
	Sort words in each file	2.04s
	Count word frequencies	0.41s
Dot product:		6.10s
Norm:		0.01s
Angle:		0.02s
Total:		**12.75s**

Performance Profiling (Symbol Table)

(*Dracula* vs. *Lewis & Clark*)

Step	Function	Running Time
Create vectors:	Read each file	1.19s
	Parse each file	1.37s
	Sort words in each file	0
	Count word frequencies	0
Dot product:		0.03s
Norm:		0.01s
Angle:		0.02s
Total:		**2.43s**

Symbol Tables are Useful

Example 2: Document Distance

– Step 1: Read in each document

  - Read and parse the file.
  - Put each (word, count) in a HashMap.

Symbol Table:

- key (String) = word
- value (Integer) = count (# times in doc)

Symbol Tables are Useful

Example 2: Document Distance

– Step 1: Read in each document

- Read and parse the file.
- Put each (word, count) in a map.

```
if (word != "")
{
    if (m_WordList.containsKey(word))
    {
        int count = m_WordList.get(word)+1;
        m_WordList.put(word, count);
    }
    else
    {
        m_WordList.put(word, 1);
    }
    word = "";
}
```

Symbol Tables are Useful

– Step 2: Compare documents (dot-product)

```
// Get an iterator for all the keys stored in A
Set<String> ASet = A.m_WordList.keySet();
Iterator<String> AIterator = ASet.iterator();

// Iterate through all the keys in A
while (AIterator.hasNext())
{
    String Key = AIterator.next();

    // If the key from A is also in B
    if (B.m_WordList.containsKey(Key))
    {
        // Add the product of the counts to the sum.
        int AValue = A.m_WordList.get(Key);
        int BValue = B.m_WordList.get(Key);
        sum += AValue*BValue;
    }
}
```

Document Distance

(Dracula vs. *Lewis & Clark)*

Version	Change	Running Time
Version 1		4,311.00s
Version 2	Better file handling	676.50s
Version 3	Faster sorting	6.59s
Version 4	Symbol Table	2.35s

Building a Symbol Table

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.

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

Example: insert(4, Seth)

Direct Access Tables

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

Example: insert(4, Seth)

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

Time: O(1) / insert, O(1) / search

Direct Access Tables

Problems:

- Too much space
  - If keys are integers, then table-size > 4 billion

- What if keys are not integers?
  - Where do you put the key/value “**(hippopotamus, bob)**”?
  - Where do you put 3.14159?

Direct Access Tables

Pythagoras said, "Everything is a number."

"The School of Athens" by Raphael

[Source: MIT 6.006]

Direct Access Tables

Pythagoras said, “Everything is a number.”

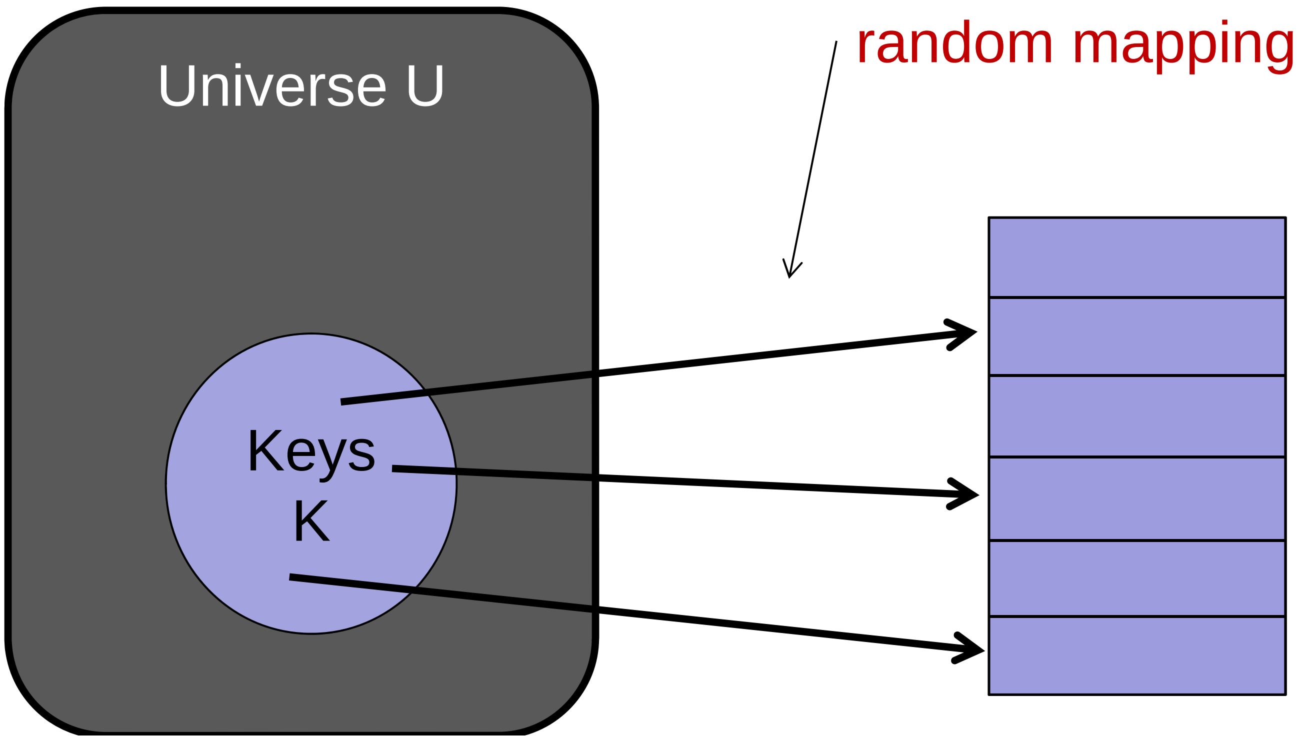
- Everything is just a sequence of bits.
- Treat those bits as a number.

- English:
  - 26 letters => 5 bits/letter
  - Longest word = 28 letters (antidisestablishmentarianism?)
  - 28 letters * 5 bits = 140 bits
  - So we can store any English text in a direct-access array of size 2^{140}.

Hash Functions

Problem:

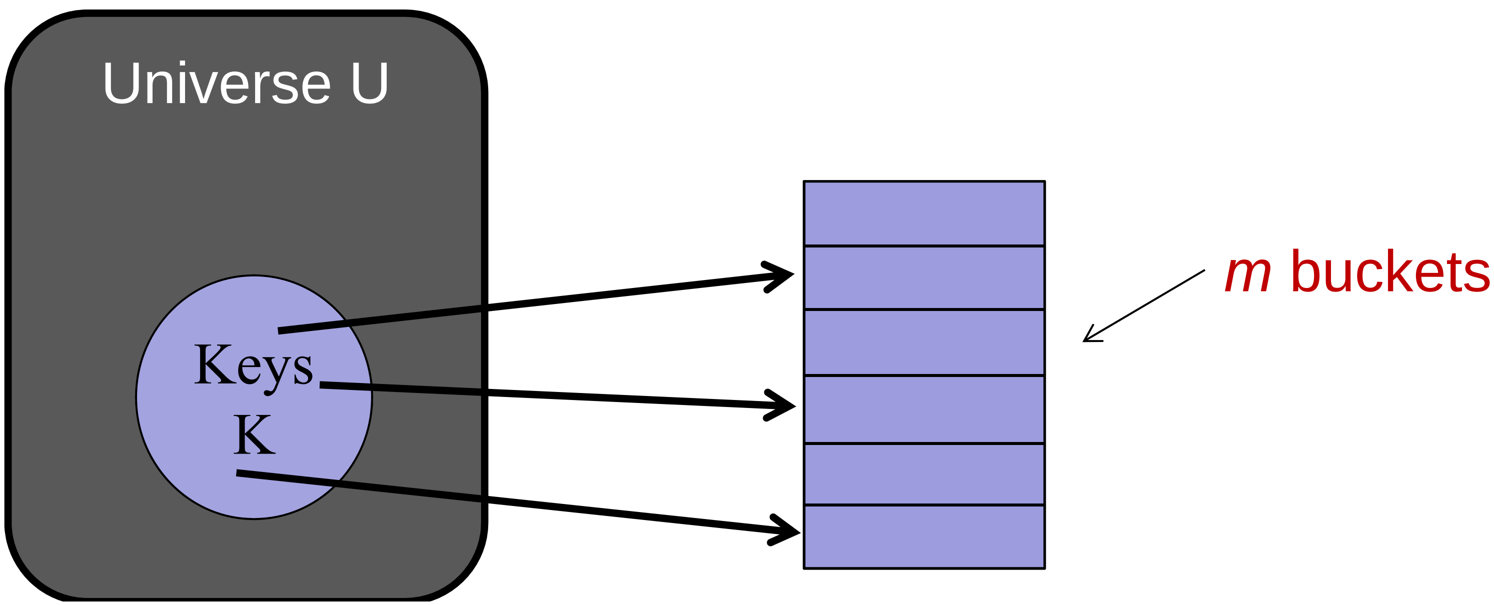
- Huge universe U of possible keys.
- Smaller number n of actual keys.
- How to map n keys to $m \approx n$ buckets?



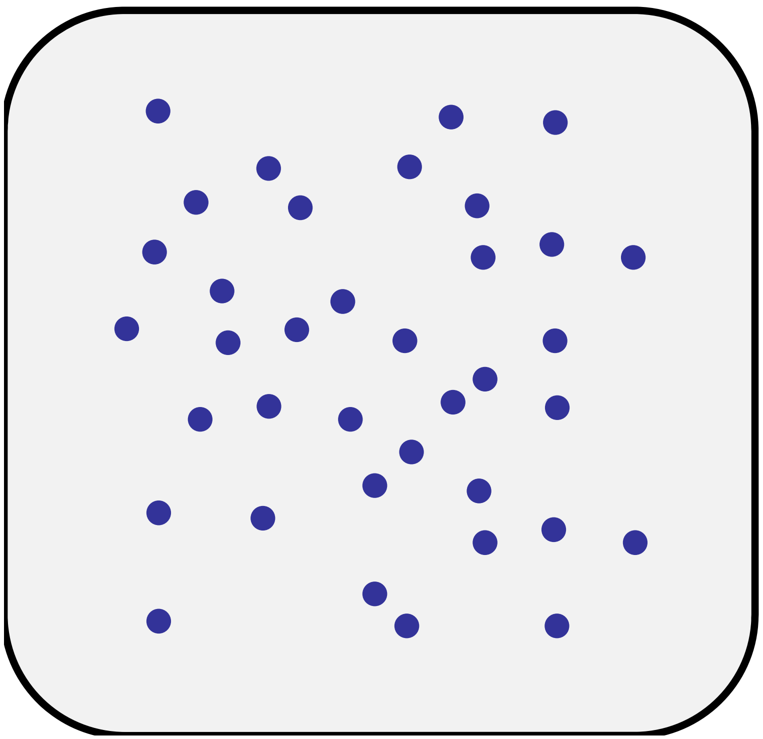
Hash Functions

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

- Store key k in bucket $h(k)$.
- Time complexity:
  - Time to compute h + Time to access bucket
- For now: assume hash function has cost 1 to compute.



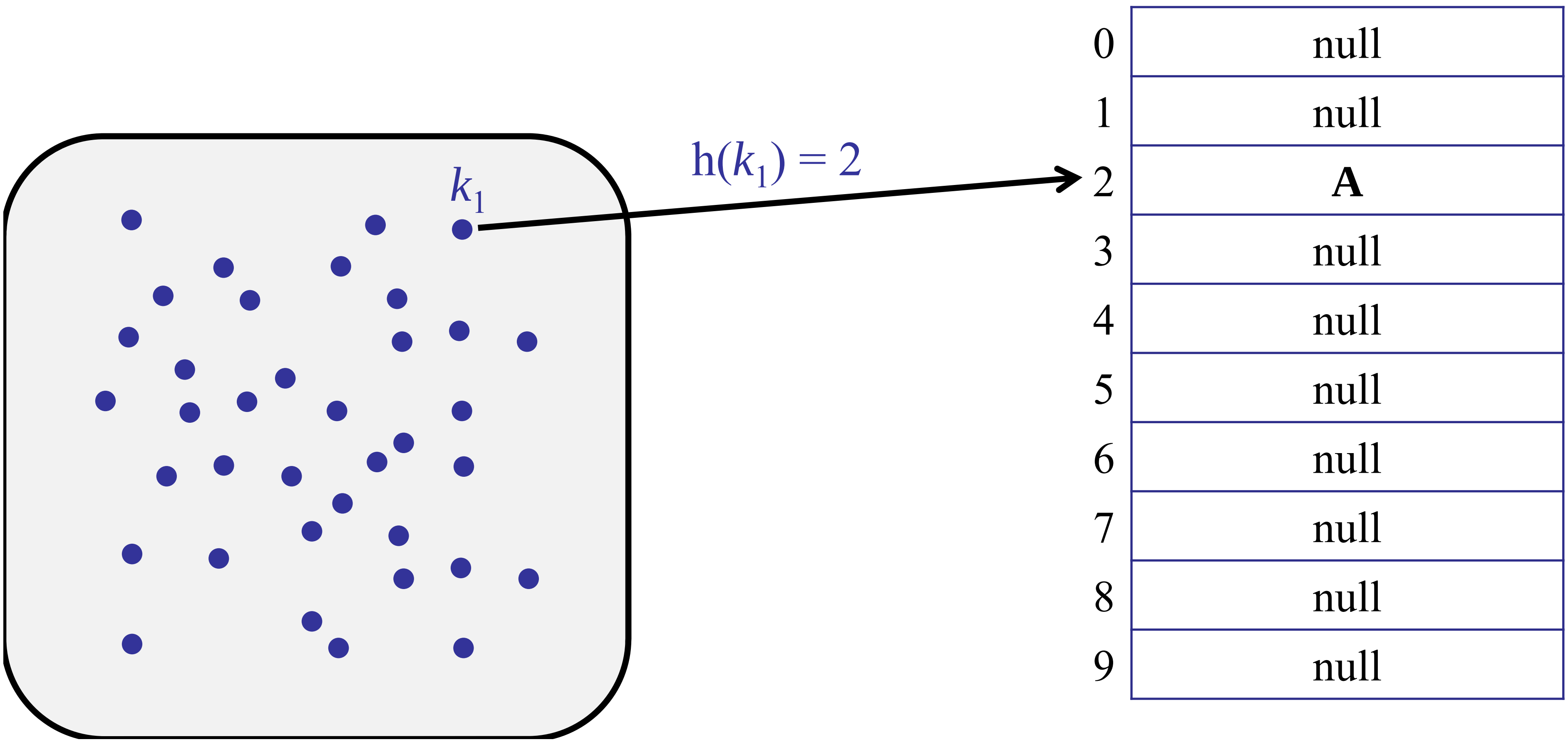
Hash Functions



0	null
1	null
2	null
3	null
4	null
5	null
6	null
7	null
8	null
9	null

Hash Functions

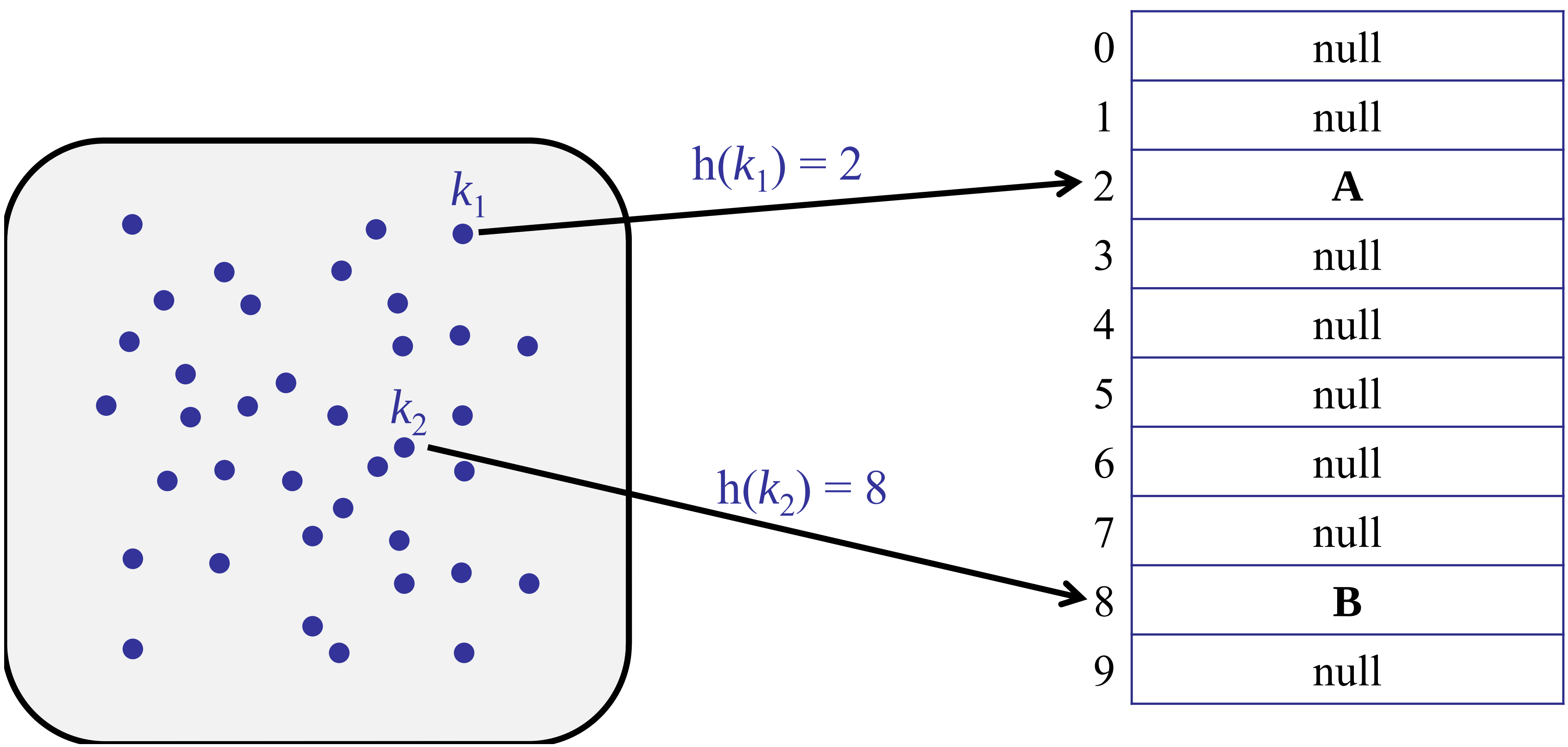
insert(k_1, A)



Hash Functions

insert(k_1, A)

insert(k_2, B)



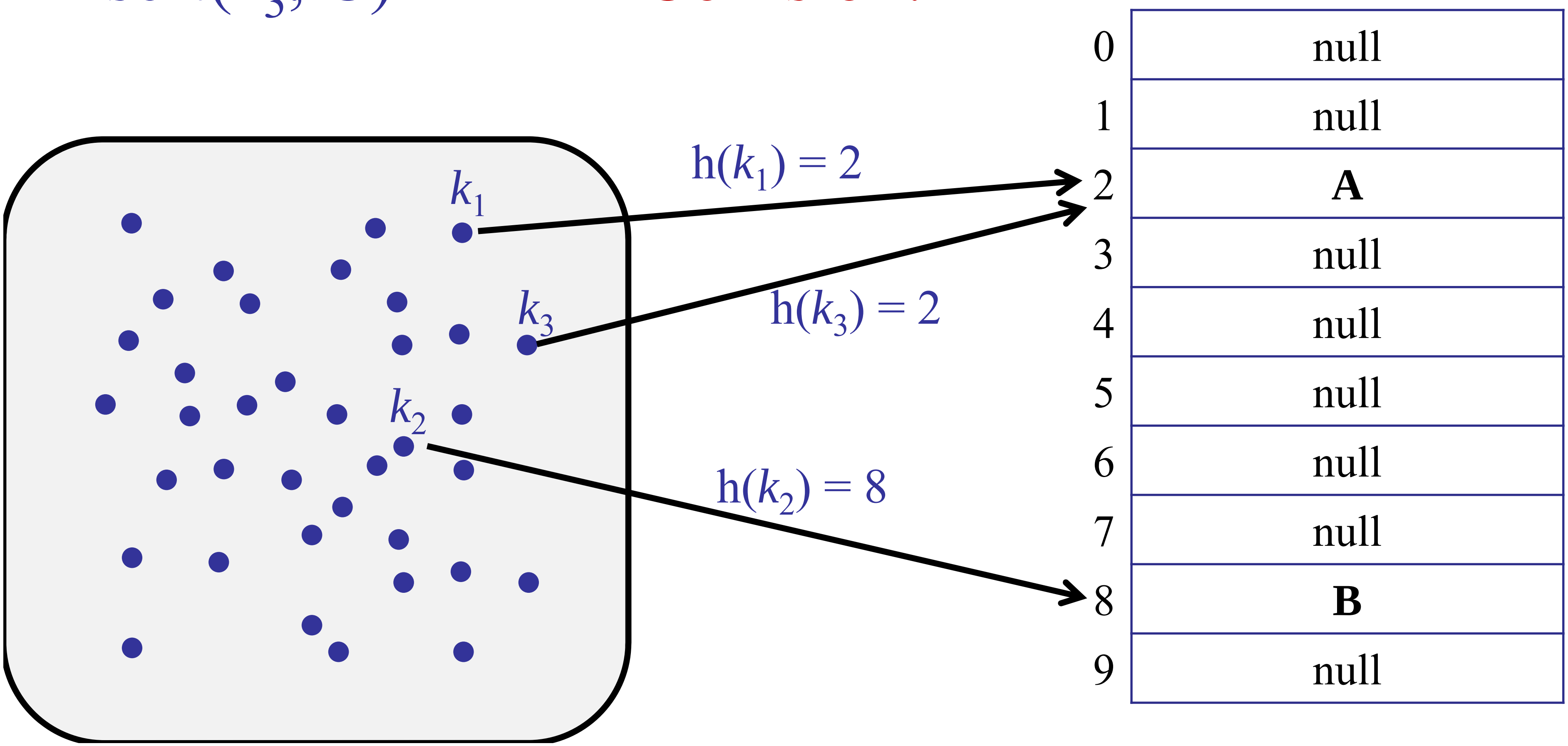
Hash Functions

insert(k_1, A)

insert(k_2, B)

insert(k_3, C)

Collision!



Hash Functions

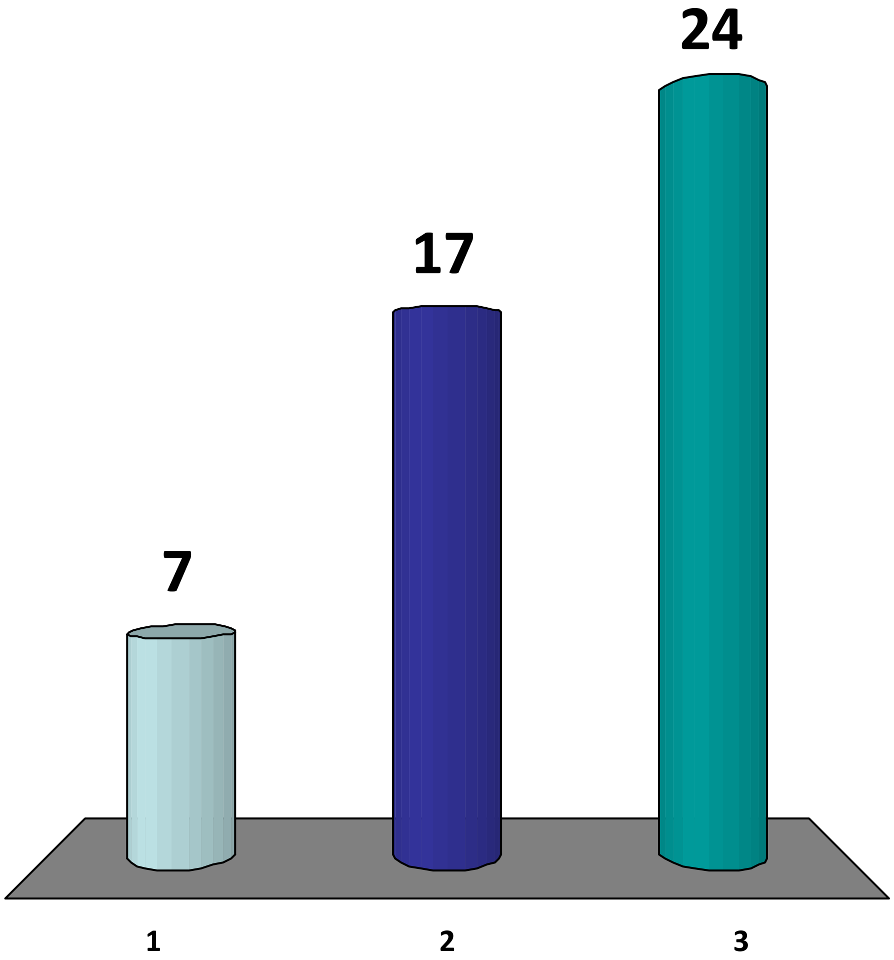
Collisions:

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

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

Can we choose a hash function with no collisions?

1. Yes
2. Sometimes, if we choose carefully
3. ✓ No, impossible



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!

How to Cope with Collisions?

- Idea: choose a new, better hash functions

Coping with Collision

- Idea: choose a new, better hash functions
  - Hard to find.
  - Requires re-copying the table.
  - Eventually, there will be another collision.

Coping with Collision

- Idea: choose a new, better hash functions
  - Hard to find.
  - Requires re-copying the table.
  - Eventually, there will be another collision.
- Idea: chaining (today)
  - Put both items in the same bucket!
- Idea: open addressing (next week)
  - Find another bucket for the new item.

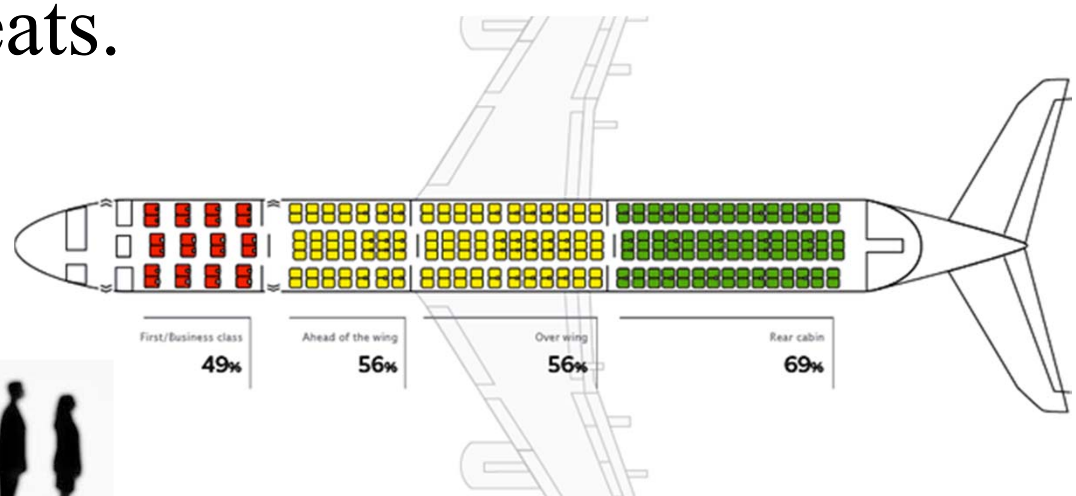
Puzzle Break

An airplane has 100 seats.

100 passengers board the airplane in a random order.

Puzzle Break

An airplane has 100 seats.



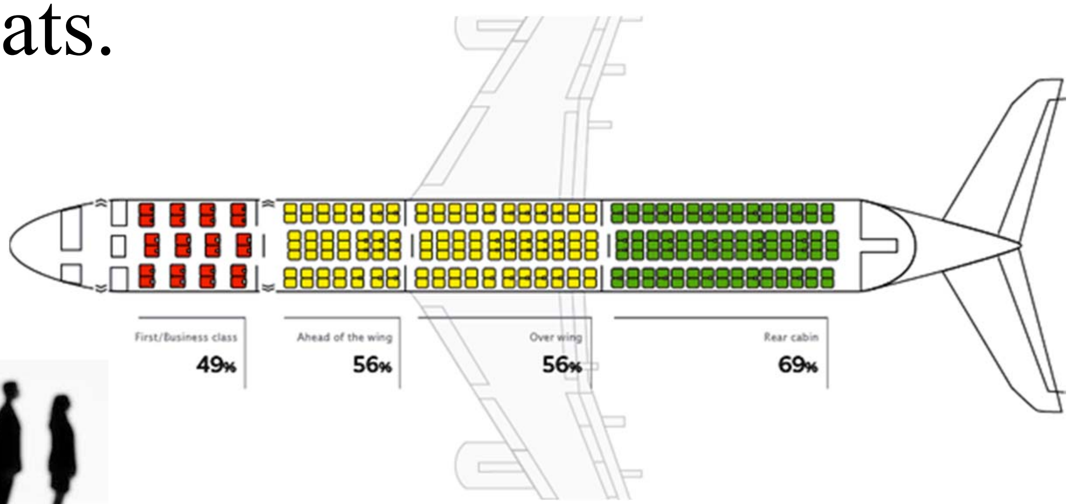
100 passengers board the airplane in a random order.

Passenger 1 is Mr Burns.

Mr Burns sits in a random seat.

Puzzle Break

An airplane has 100 seats.

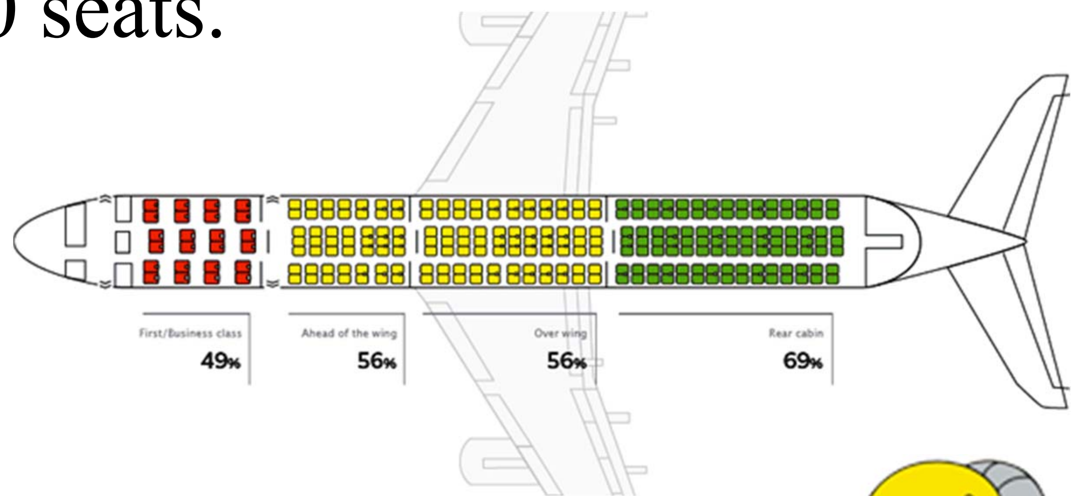


Every other passenger:

- Sits in their assigned seat, if it is free.
- Otherwise, sits in a random seat.

Puzzle Break

An airplane has 100 seats.

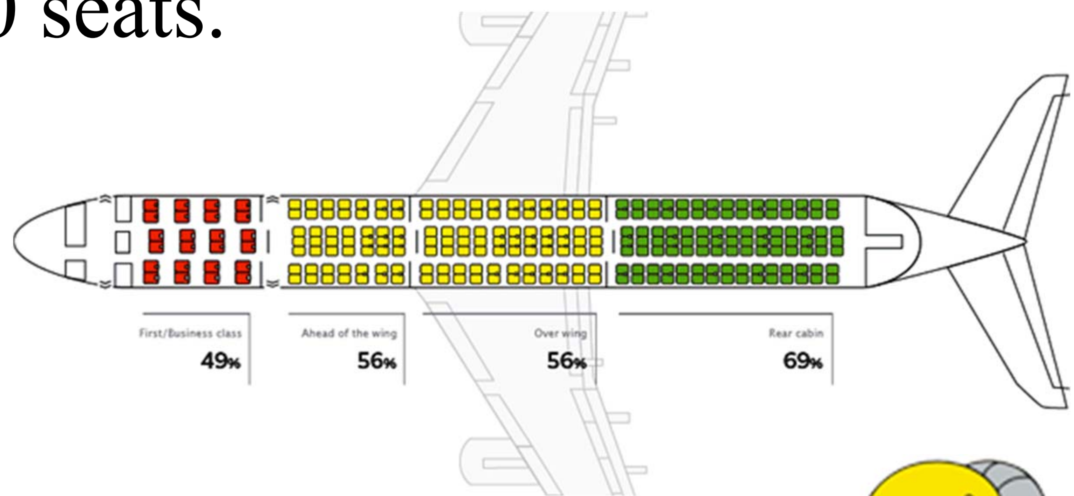


You are passenger #100.

What is the probability your seat is free when you board?

Puzzle Break

An airplane has 100 seats.



What is the probability your seat is free when you board?

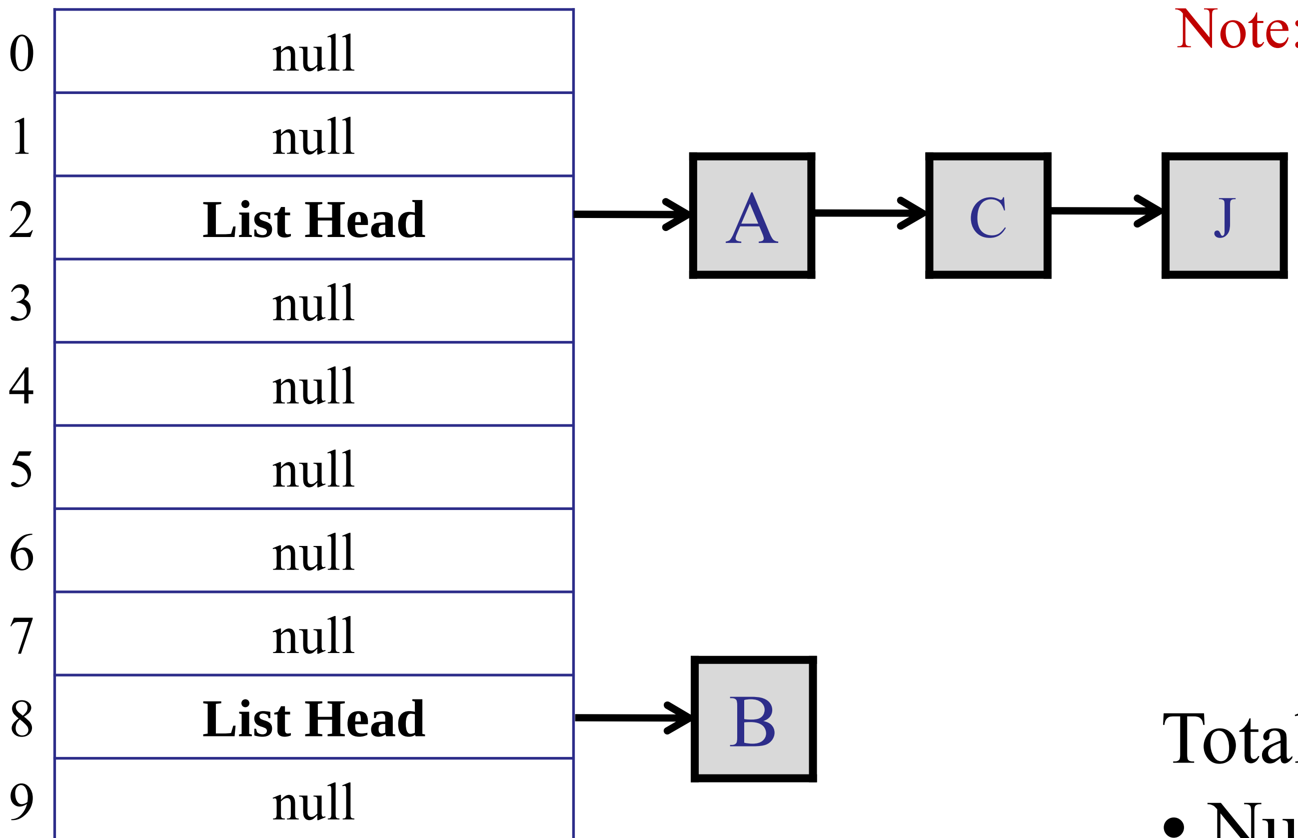
Hint: There are only two possible seats free when you board.

Coping with Collision

- Idea: choose a new, better hash functions
  - Hard to find.
  - Requires re-copying the table.
  - Eventually, there will be another collision.
- Idea: chaining (today)
  - Put both items in the same bucket!
- Idea: open addressing (next week)
  - Find another bucket for the new item.

Chaining

Each bucket contains a linked list of items.



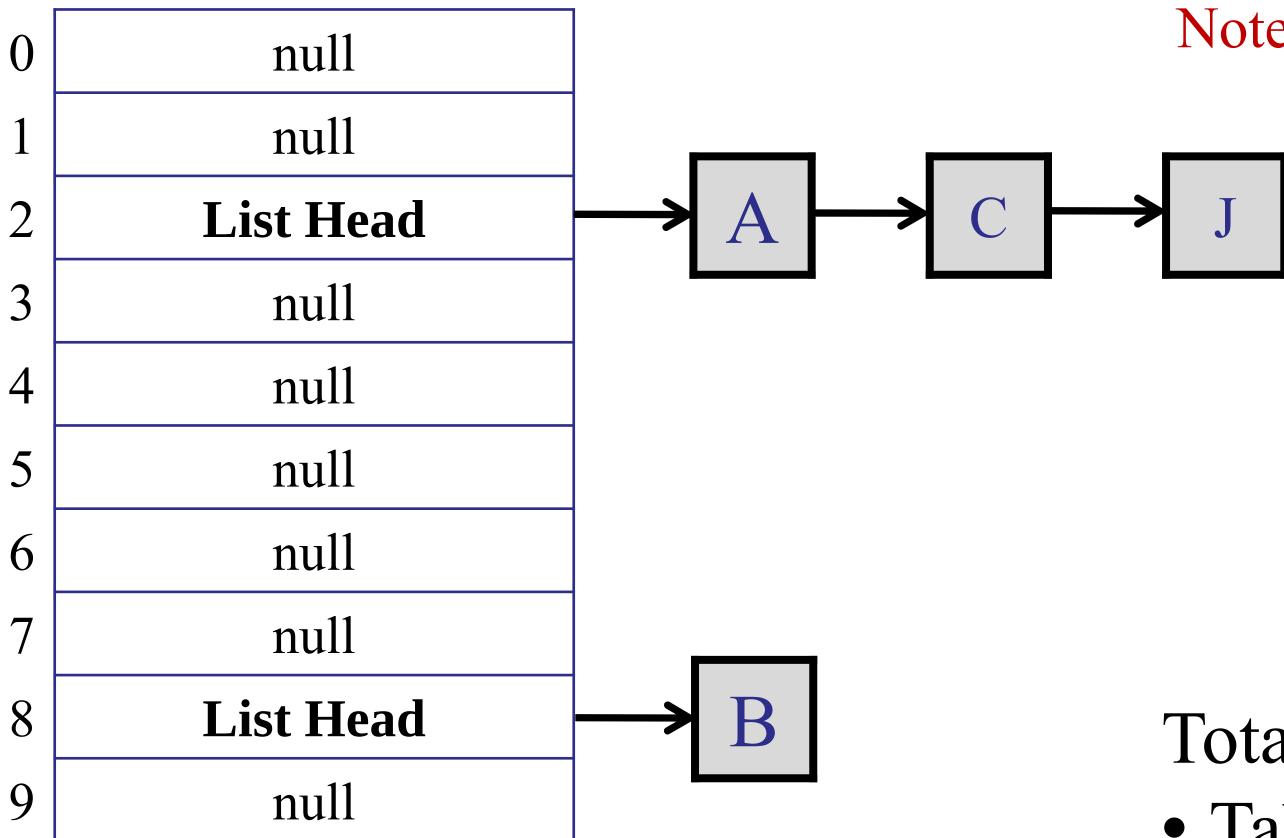
Note: h(A) == h(C) == h(J)

Total space:
- Number buckets: *m*
- Number entries: *n*

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

Hashing with Chaining

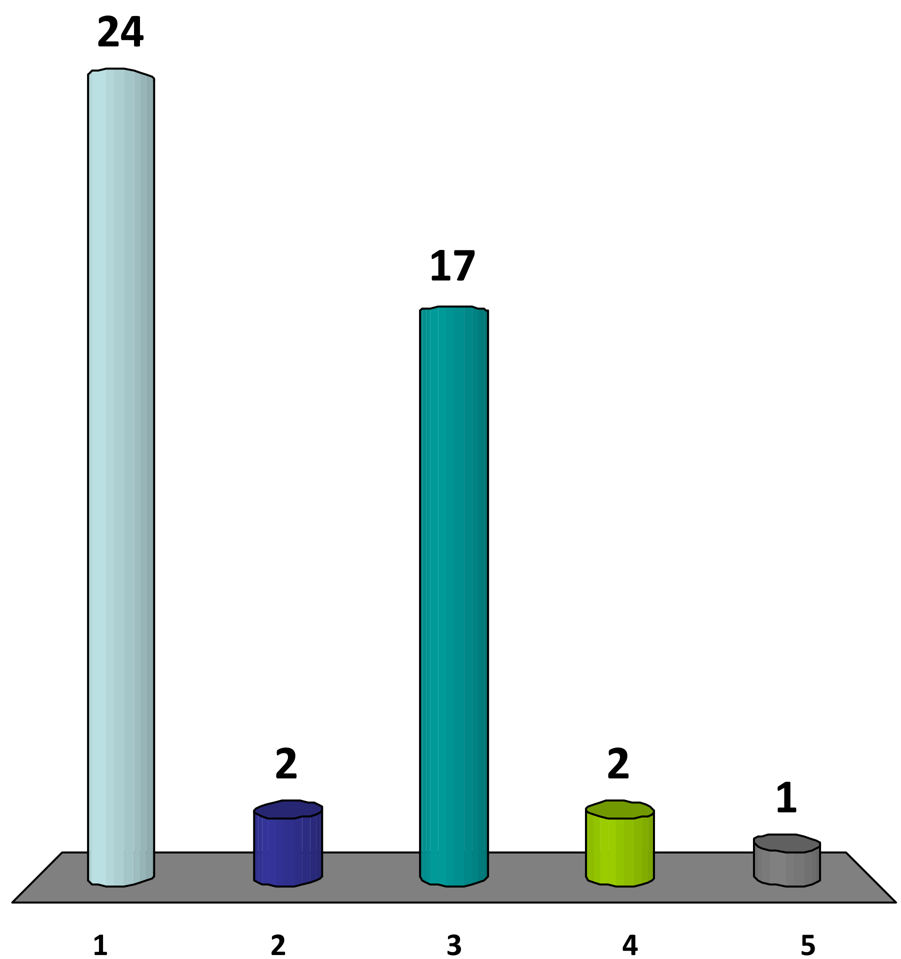
Operations:

- insert(key, value)
  - Calculate h(key)
  - Lookup h(key) and add (key,value) to the linked list.

- search(key)
  - Calculate h(key)
  - Search for (key,value) in the linked list.

What is the worst-case cost of inserting a (key, value)?

1. O(1 + cost(h))
2. O(log *n* + cost(h))
3. O(*n* + cost(h))
4. O(*n* cost(h))
5. We cannot determine it without knowing h.



Hashing with Chaining

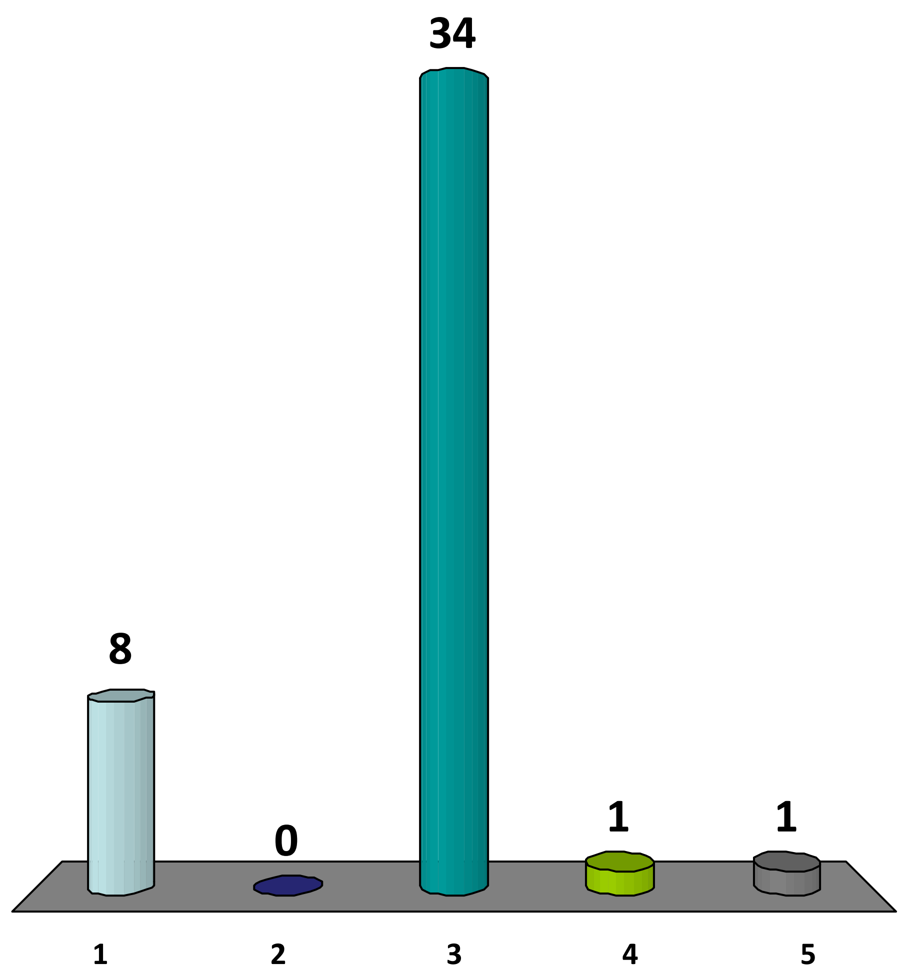
Operations:

- insert(key, value)
  - Calculate h(key)
  - Lookup h(key) and add (key,value) to the linked list.

- search(key)
  - Calculate h(key)
  - Search for (key,value) in the linked list.

What is the worst-case cost of searching a (key, value)?

1. O(1 + cost(h))
2. O(log *n* + cost(h))
3. O(*n* + cost(h))
4. O(*n**cost(h))
5. We cannot determine it without knowing h.



Hashing with Chaining

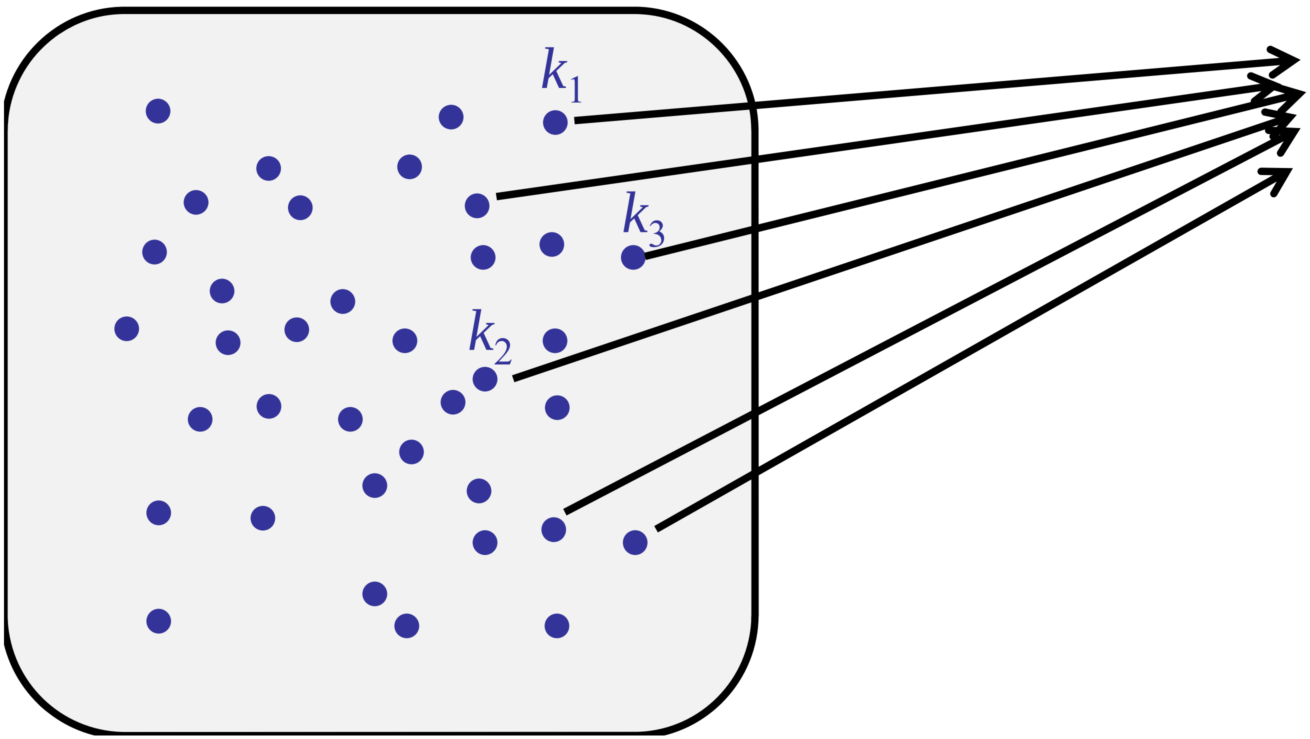
Operations:

- insert(key, value)
  - Calculate h(key)
  - Lookup h(key) and add (key,value) to the linked list.
- search(key) → time depends on length of linked list
  - Calculate h(key)
  - Search for (key,value) in the linked list.

Hashing with Chaining

Assume all keys hash to the same bucket!

– Search costs O(n)!



0	null
1	null
2	**Linked List**
3	null
4	null
5	null
6	null
7	null
8	null
9	null

Let's be optimistic today.

The Simple Uniform Hashing Assumption

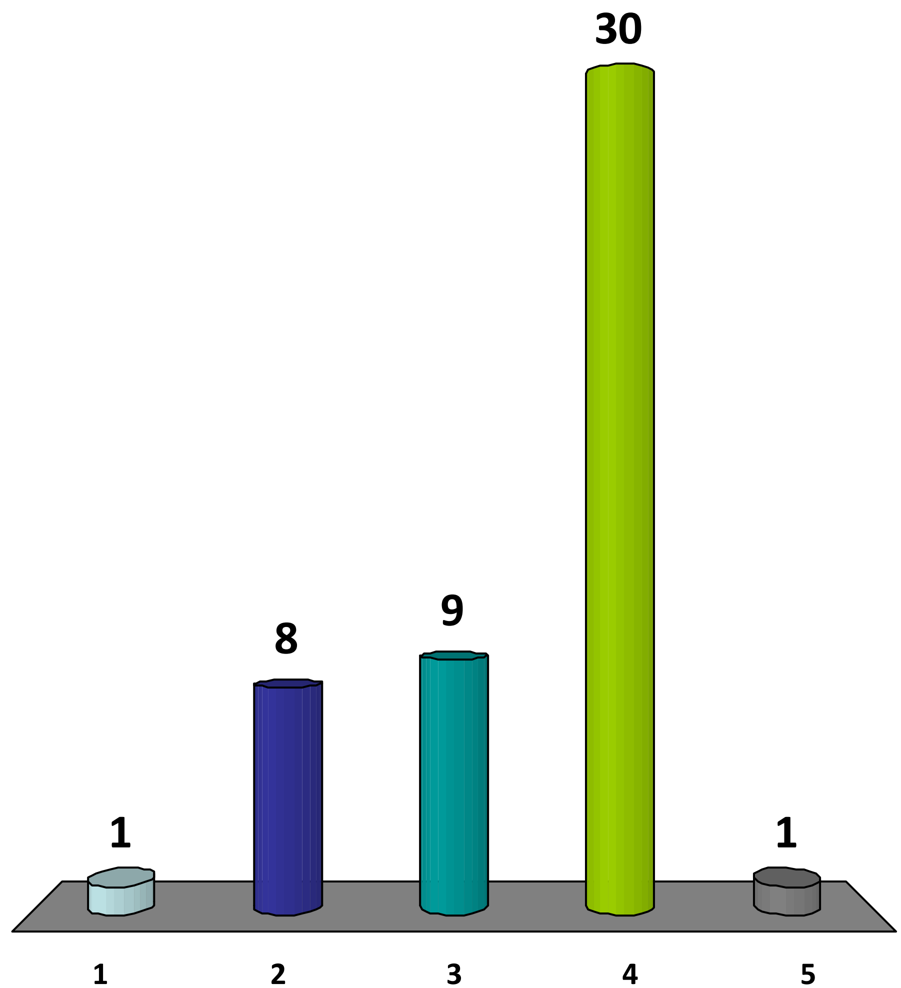
- Every key is equally likely to map to every bucket.
- Keys are mapped independently.

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.

Why don't we just insert each key into a random bucket (instead of using h)?

1. It would be slow to insert.
2. Computers don't have a real source of randomness.
3. By choosing the keys carefully, a user could force the random choices to create many collisions.
4. Searching would be very slow.
5. None of the above.



Let's be optimistic today.

The Simple Uniform Hashing Assumption

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

  = average # items / bucket.

- Expected search time = 1 + *expected # items per bucket*

hash function + array access

linked list traversal

Probability Theory

Set of outcomes for $X = (e_1, e_2, e_3, \ldots, e_k)$:

- $\Pr(e_1) = p_1$
- $\Pr(e_2) = p_2$
- …
- $\Pr(e_k) = p_k$

Expected outcome:

$$E[X] = e_1 p_1 + e_2 p_2 + \ldots + e_k p_k$$

Probability Theory

Linearity of Expectation:

- $E[A + B] = E[A] + E[B]$

Example:

- A = # heads in 2 coin flips
- B = # heads in 2 coin flips
- $A + B$ = # heads in 4 coin flips

$E[A+B] = E[A] + E[B] = 1 + 1 = 2$

Let's be optimistic today.

The Simple Uniform Hashing Assumption

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

  = average # items / bucket.

- Expected search time = 1 + *expected # items per bucket*

hash function + array access

linked list traversal

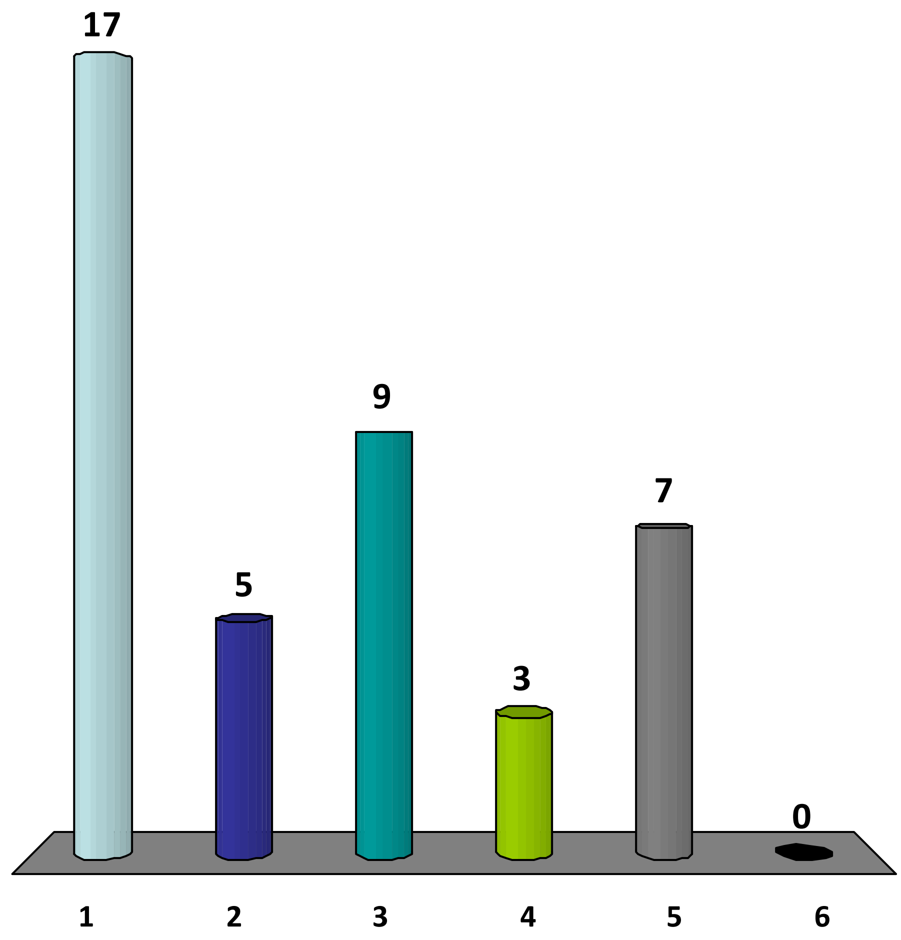
A little more probability

Indicator random variables

$$X(i, j) = 1 \quad \text{if item } i \text{ is put in bucket } j$$
$$= 0 \quad \text{otherwise}$$

Pr(X(i, j) == 1) = ?

1. 1/m
2. 1/n
3. 1/(m+n)
4. m/n
5. n/m
6. log(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.

A little probability

Indicator random variables

$$X(i, j) = 1 \quad \text{if item } i \text{ is put in bucket } j$$
$$= 0 \quad \text{otherwise}$$

$$\mathbf{Pr}(X(i, j) == 1) = 1/m$$

A little probability

Indicator random variables

$X(i, j) = 1$ if item i is put in bucket j

$\quad\quad\quad = 0$ otherwise

Pr$(X(i, j) == 1) = 1/m$

E$(X(i, j))$ = **??**

A little probability

Indicator random variables

$X(i, j) = 1$ if item i is put in bucket j

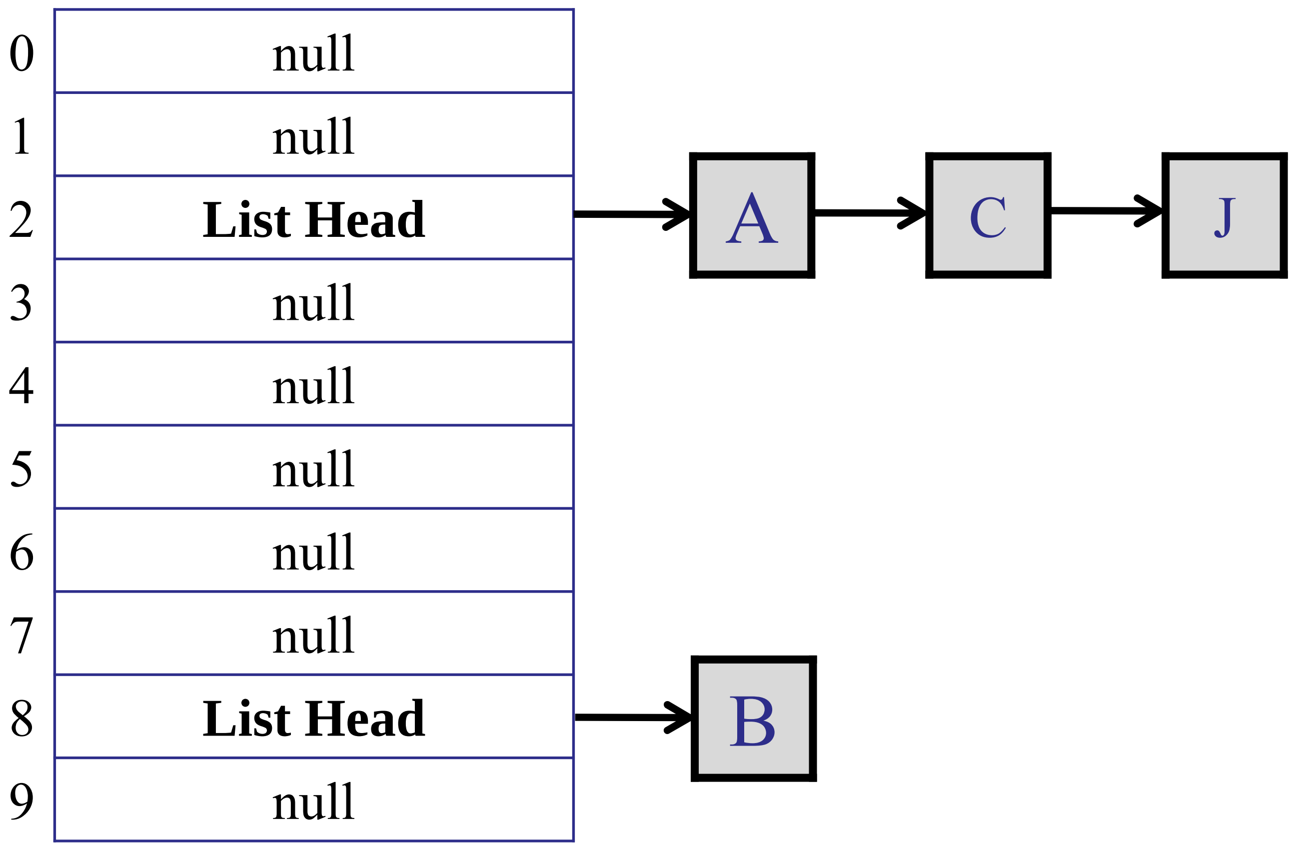
$\quad\quad\quad = 0$ otherwise

$\mathbf{Pr}(X(i, j) == 1) = 1/m$

$$\begin{aligned} E(X(i, j)) &= \mathbf{Pr}(X(i, j) == 1) * 1 + \mathbf{Pr}(X(i, j) == 0) * 0 \\ &= \mathbf{Pr}(X(i, j) == 1) \\ &= 1/m \end{aligned}$$

A little probability

What is the expected number of items in a bucket?



A little probability

Indicator random variables

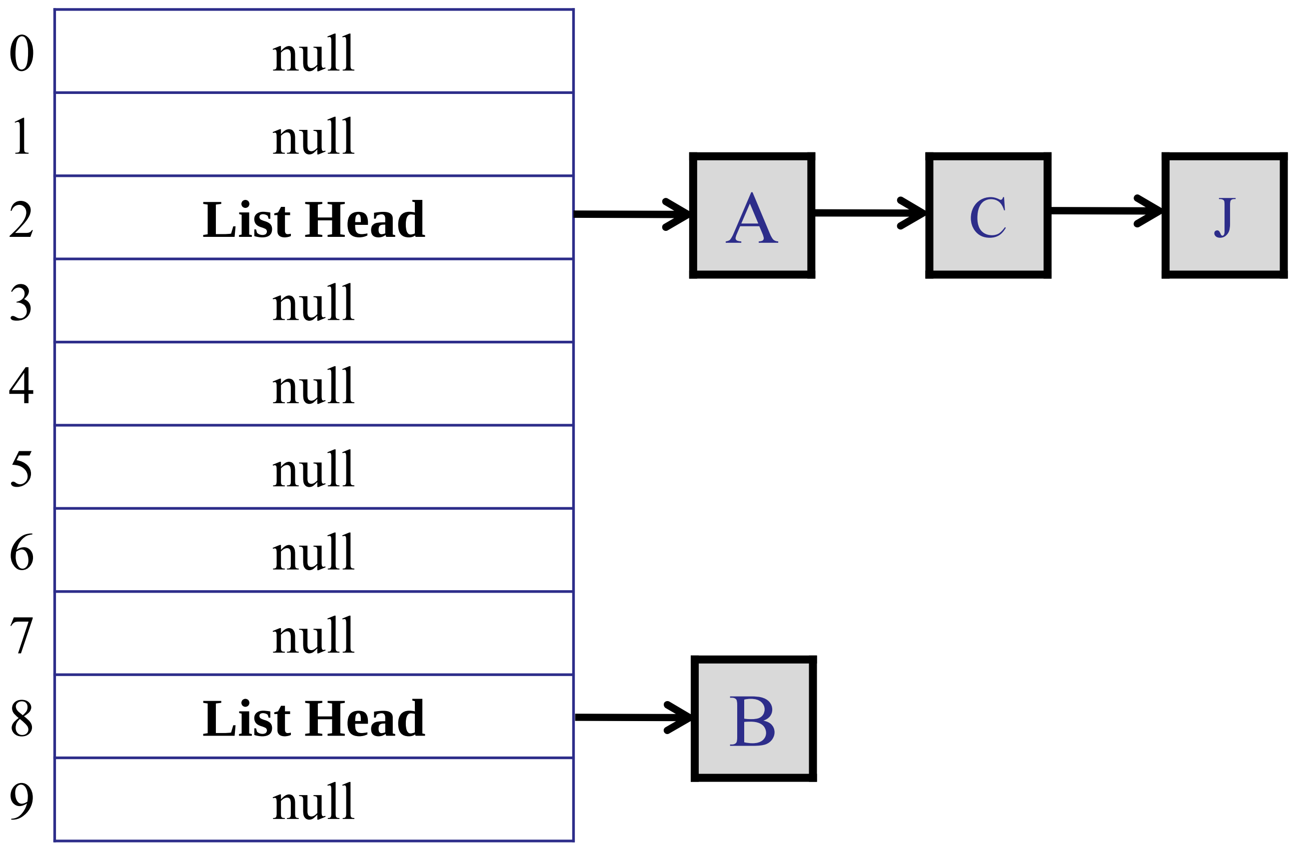
$X(i, j) = 1$ if item i is put in bucket j

$ = 0$ otherwise

$\Sigma_i X(i, b) =$ number of items in bucket b

A little probability

Each item contributes `1' to the bucket it is in..



A little probability

Indicator random variables

$$X(i, j) = 1 \quad \text{if item } i \text{ is put in bucket } j$$
$$= 0 \quad \text{otherwise}$$

$$\Sigma_i\, X(i, b) = \text{number of items in bucket } b$$

A little probability

Calculate expected number of items per bucket:

Expected (Σ_i X(i, b)) =

A little probability

Calculate expected number of items per bucket:

$$\mathbf{E}(\Sigma_i X(i, b)) = \Sigma_i \mathbf{E}(X(i, b))$$

Linearity of expectation: E(A + B) = E(A) + E(B)

A little probability

Calculate expected number of items per bucket:

$$\mathbf{E}(\Sigma_i X(i, b)) = \Sigma_i \mathbf{E}(X(i, b))$$

$$= \Sigma_i 1/m$$

$$= n/m$$

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 $= 1 + n/m$

  $= O(1)$

Hashing with Chaining

Searching:

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

Inserting:

- Worst-case insertion time = O(1)

Hashing with Chaining

What if you insert n elements in your hash table?

What is the expected *maximum* cost?

Hashing with Chaining

What if you insert n elements in your hash table?

What is the expected *maximum* cost?

- Analogy:
  - Throw n balls in $m = n$ bins.
  - What is the maximum number of balls in a bin?

Cost: $O(\log n)$

Hashing with Chaining

What if you insert n elements in your hash table?

What is the expected *maximum* cost?

- Analogy:
  - Throw n balls in $m = n$ bins.
  - What is the maximum number of balls in a bin?

Cost: $\Theta(\log n / \log\log n)$

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!)
- Use linked list to store multiple keys in one bucket.

Analyze performance with simple uniform hashing.

- Expected number of keys / bucket is $O(n/m) = O(1)$.

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

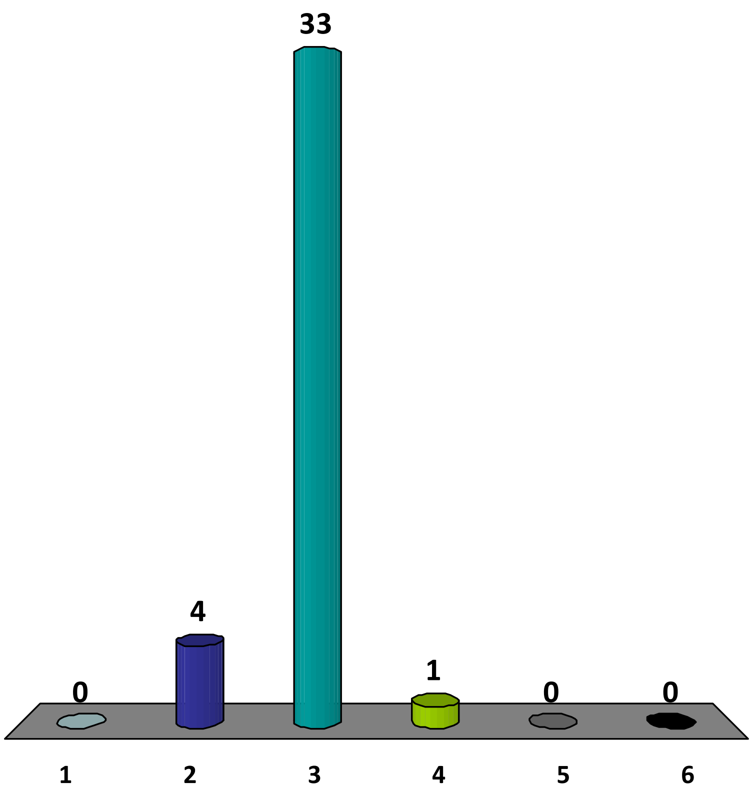
  Recall: 001001 >> 1 = 00100

  001001 >> 2 = 0010

  001001 >> 3 = 001

32 >> 3 = ?

1. 1
2. 2
3. 4
4. 8
5. 16
6. 32



Designing Hash Functions

Division Method

– (Bad) Idea: choose $m = 2^x$

Very fast to calculate k mod m via shifts:

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

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.

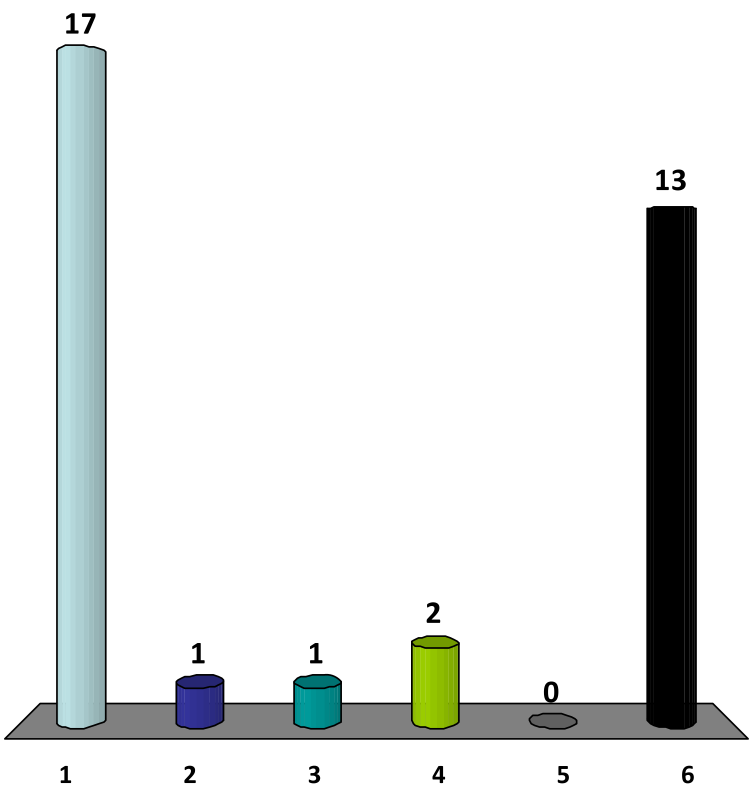
$$\underbrace{k \bmod m + i*m = k}$$

divisible by d

- Implies that $h(k) = k \bmod m$ 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*.

$$k \text{ mod } m + i{*}m = k$$

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.

Designing Hash Functions

Two common hashing techniques…

- Division Method
- Multiplication Method

Designing Hash Functions

Multiplication Method

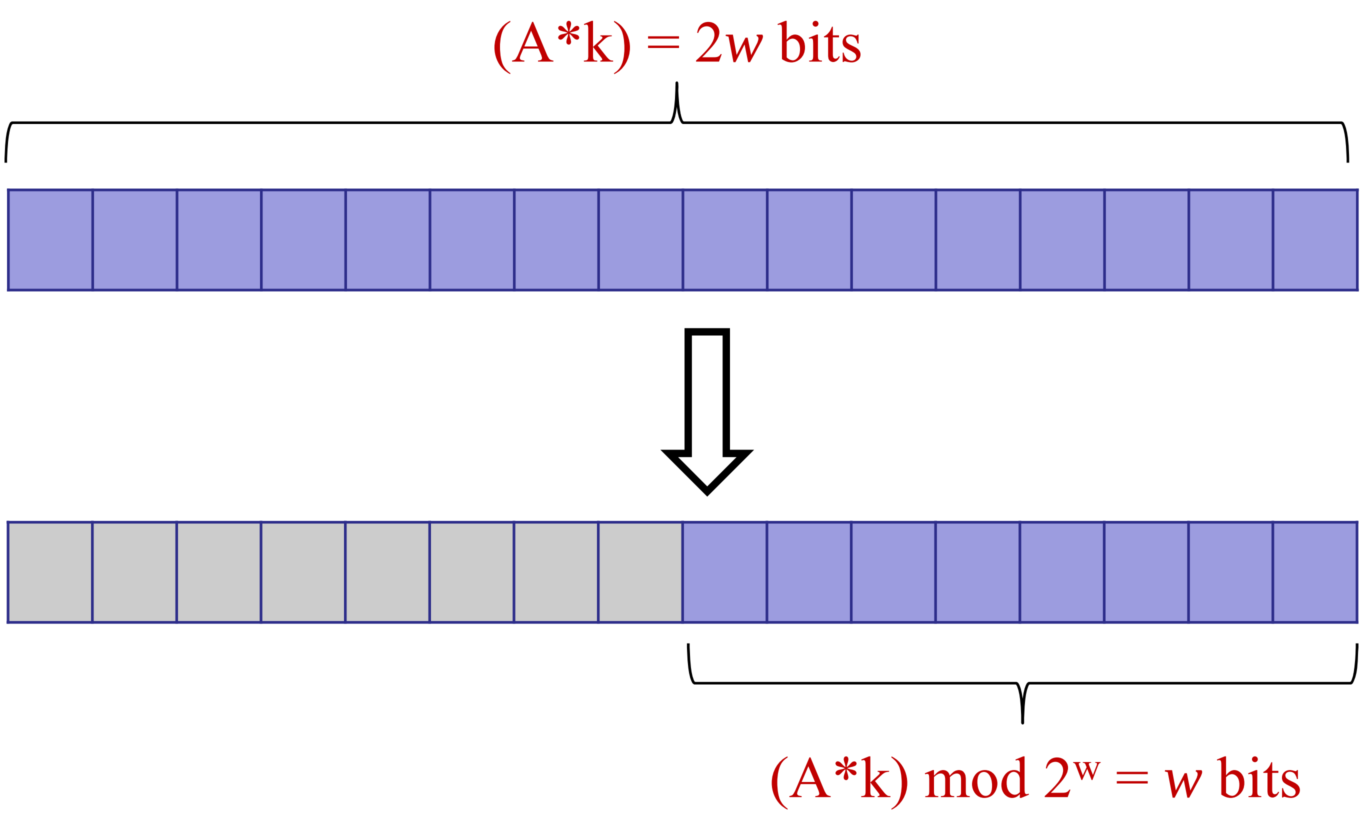
- Fix table size: $m = 2^r$, for some constant r.
- Fix word size: w, size of a key in bits.
- Fix (odd) constant A.

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

Designing Hash Functions

Multiplication Method

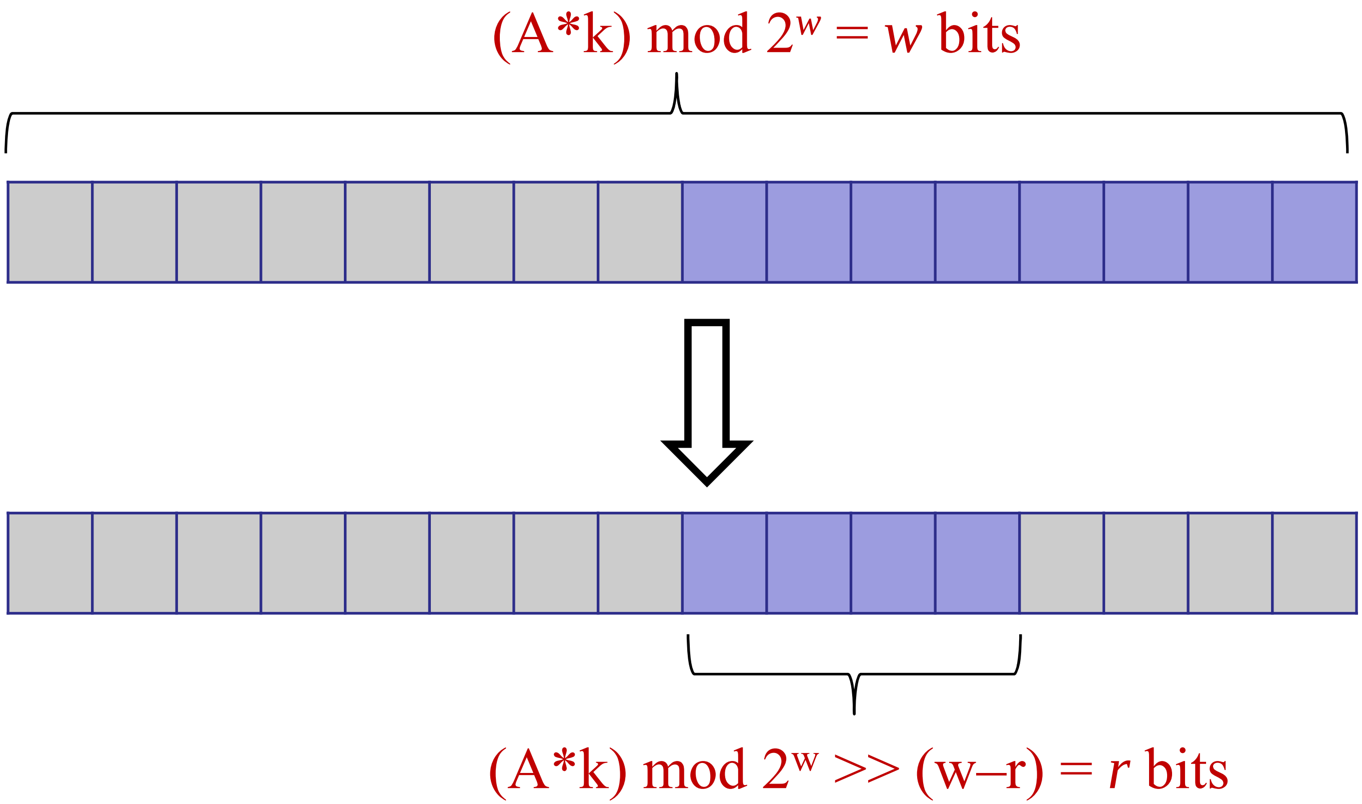
- Given m, w, r, A: $h(k) = (Ak) \bmod 2^w \gg (w - r)$



Designing Hash Functions

Multiplication Method

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



Designing Hash Functions

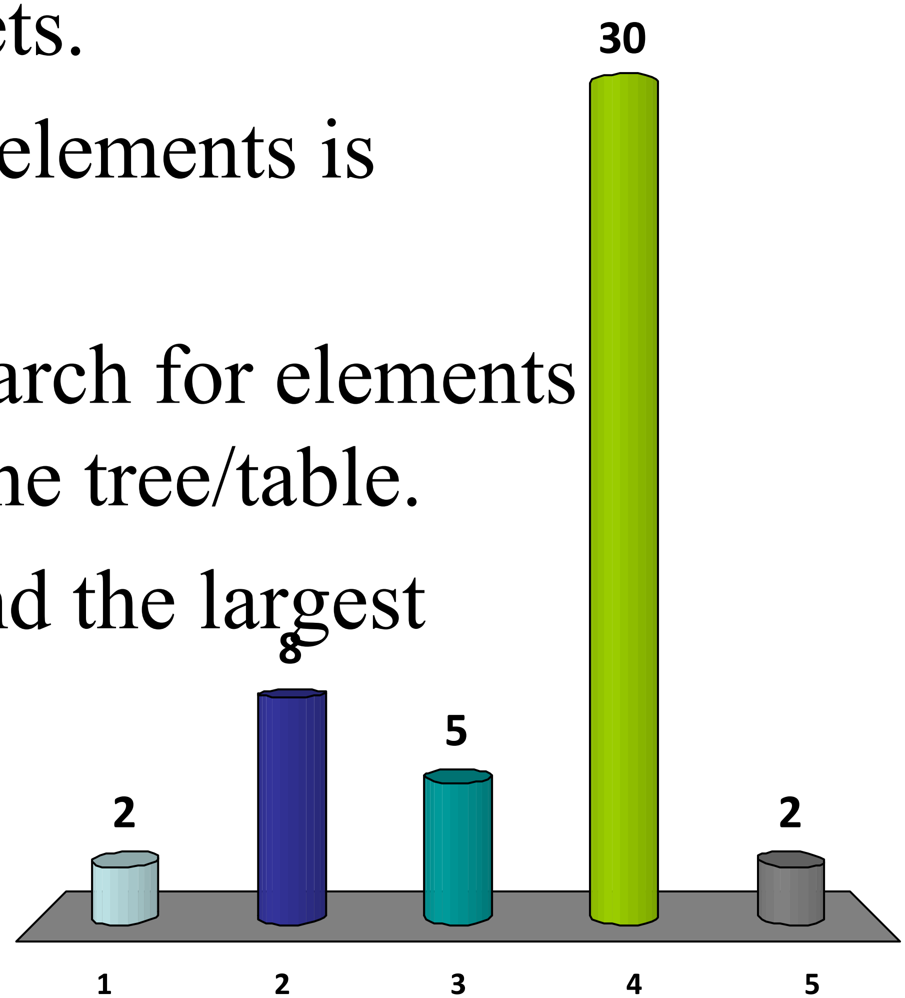
Multiplication Method

- 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.

Quick Review…

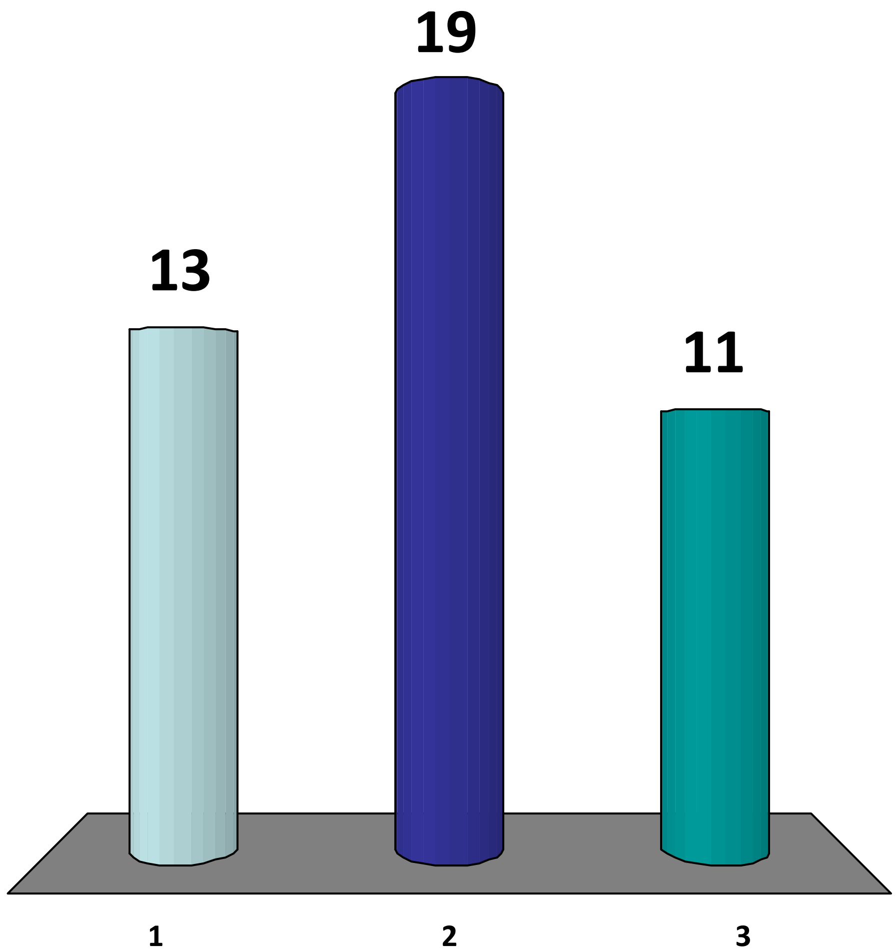
When is a BST better than a Hash Table?

1. For very large data sets.
2. When the number of elements is unknown in advance.
3. When you need to search for elements that might not be in the tree/table.
4. When you need to find the largest element.
5. Never.



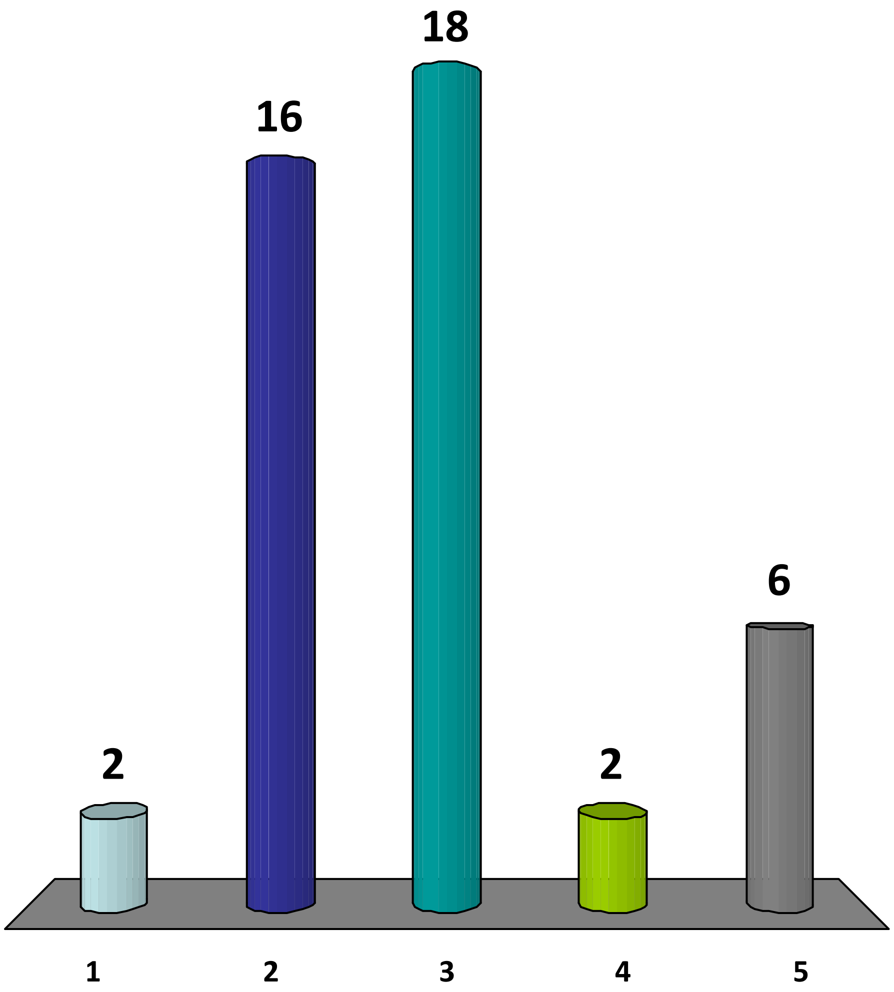
Can you easily extend a symbol table / hash table to maintain the order in which items are <u>inserted</u>??

1. Yes.
2. No.
3. Only if you are really, really clever.



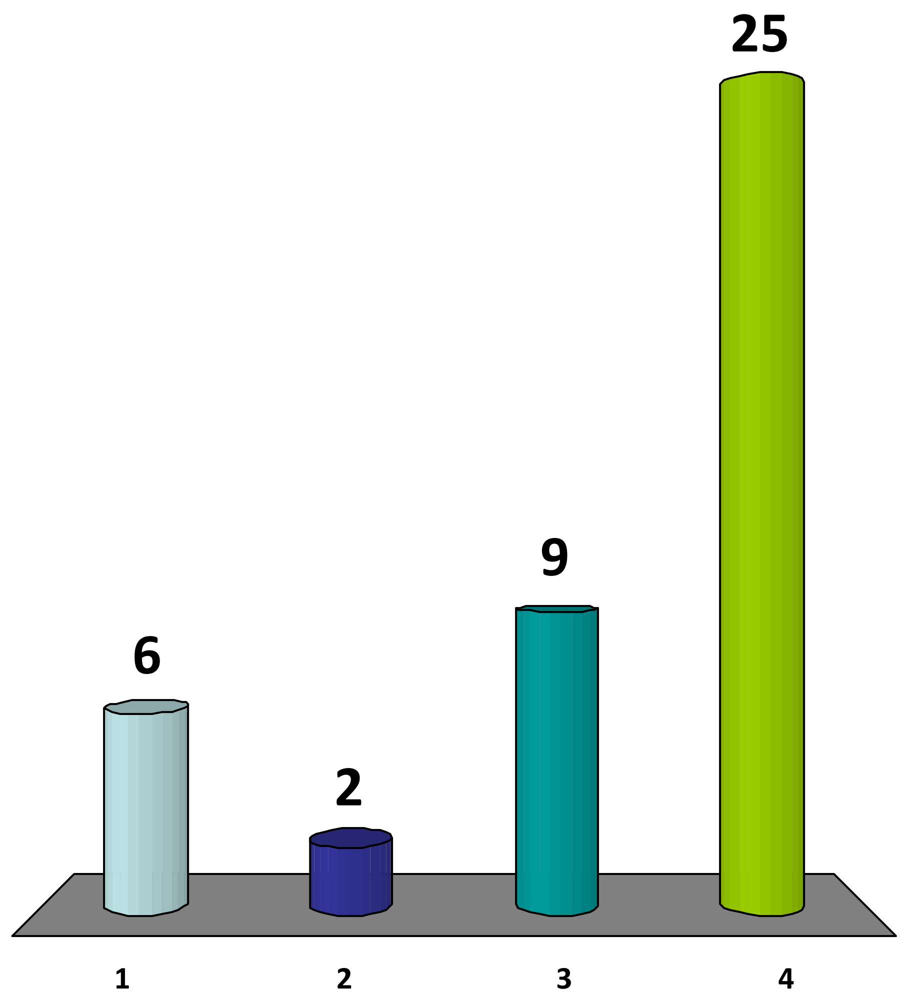
Which of the following is *not* a problem with a direct access table?

1. It takes up too much space.
2. Keys must be integers.
3. Searching it is slow.
4. Enumerating all elements is slow.
5. None of the above.



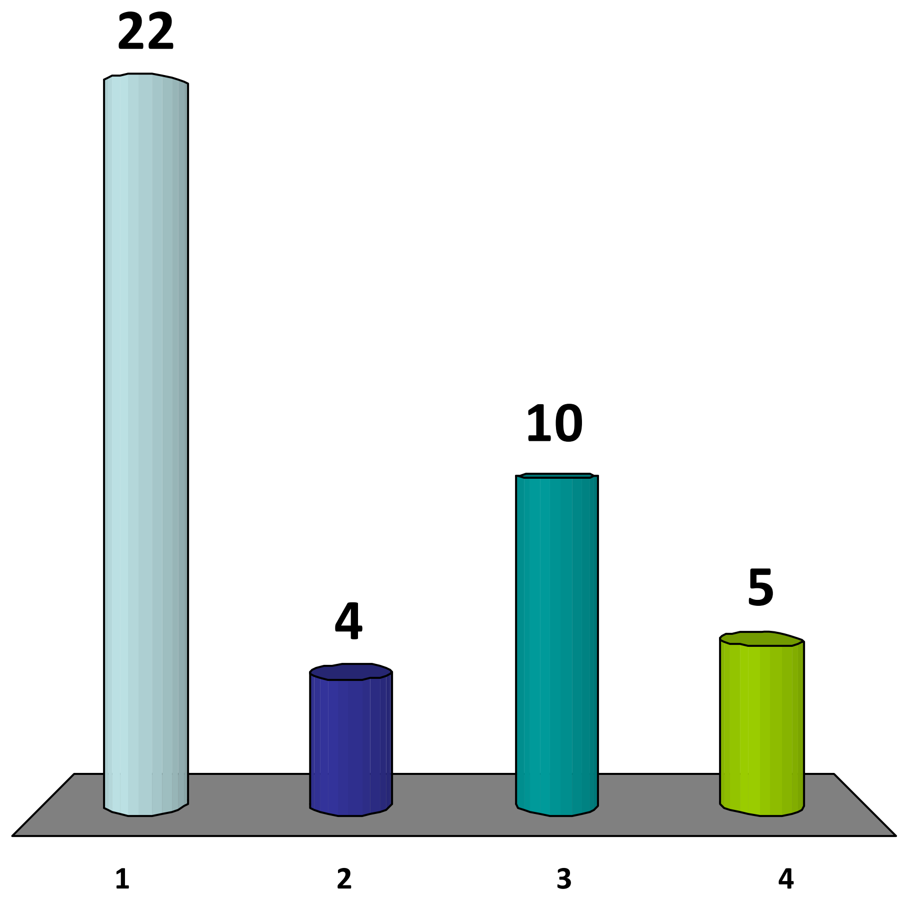
Enumerating keys in a hash table is fastest when:

1. m > n
2. m > n^2
3. n >= m
4. It doesn't matter.



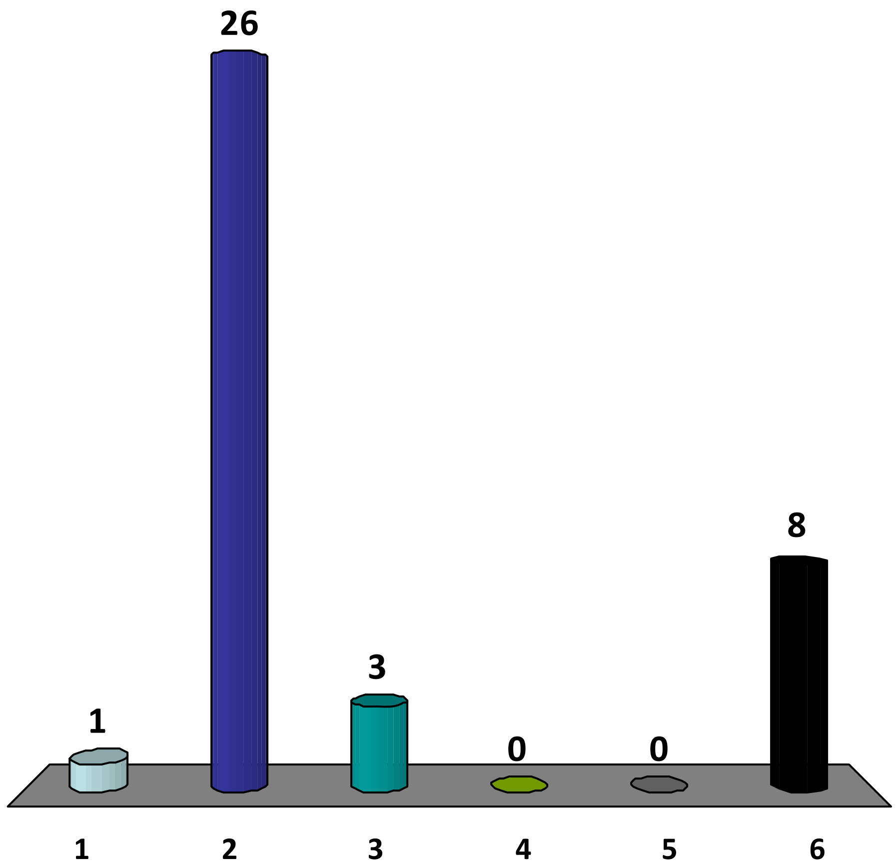
Searching in a hash table is fastest when:

1. $m > n$
2. $n > m$
3. $n \approx m$
4. It doesn't matter.



For the division method, which of the following is a good table size?

1. 102
2. 103
3. 104
4. 105
5. 106
6. None of the above.



Summary

Symbol Tables are pervasive

- You find them everywhere!

Hash tables are fast, efficient symbol tables.

- Under optimistic assumptions, provably so.
- In the real world, often so.
- But be careful!

Beats BSTs:

- Operate directly on keys (i.e., indexing)
- Gave up: successor/predecessor/etc.