CS2020
Data Structures and Algorithms

Welcome!

Administrative

- Today:
  - Recitations as usual.

- Next week:
  - Mid-semester recess

- Week after:
  - Coding quiz

Problem Sets

- Problem Set 4:
  - Due Wednesday
  - Sorry about deadline confusion!

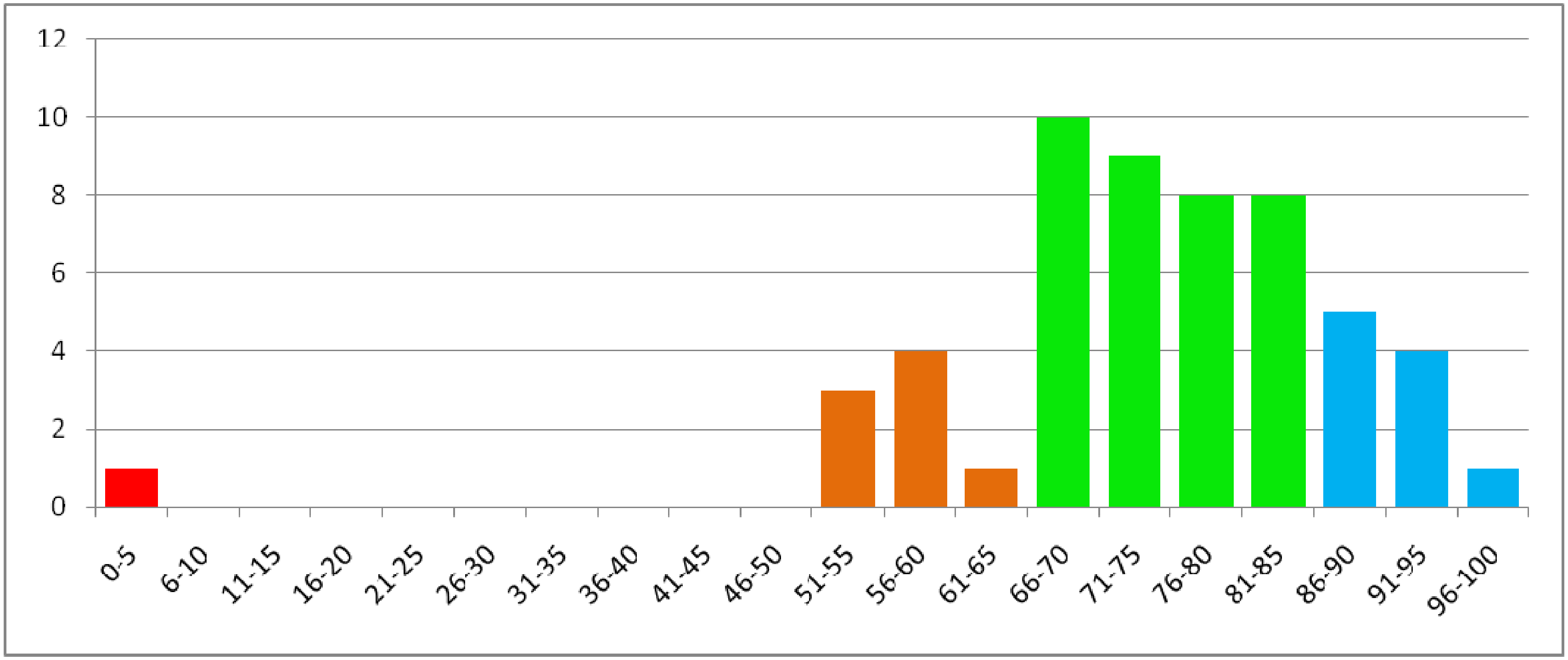
- Problem Set 5:
  - Due after the break
  - Only one (required) problem: extend BST
  - One optional problem: analyzed weight-balanced BST
    - Bonus problem!
    - Example of amortized analysis.

Quiz 1: Results Update

Overall:	Score	Percent
Average:	132 / 180	73

Problem	Max	Average
1. Recurrences	15	12
2. Multiple Choice	40	35
3. Java	40	27
4. Data Structure Basics	40	34
5. Polygonal Search	45	27

Quiz 1: Results Update



Interpretation:

> 85%: Excellent!

> 65%: Good.

< 65%: Time to catch up.

Today: New Topic

Hash Tables

- Dictionaries in Java
- Dictionaries are Useful
- Hash functions
- Chaining
- Simple Uniform Hashing
- Good Hash Functions

Dictionary Data Structure

Abstract data type (ADT) that maintains a set of items, each with a key, subject to:

- insert(*key*, *data*)
  - Adds (*key*, *data*) pair to the dictionary.
- delete(*key*)
  - Removes every (*key*, *) from the dictionary.
- search(*key*)
  - Returns item (*key*, *data*) if it exists in the dictionary.

Dictionary Data Structure

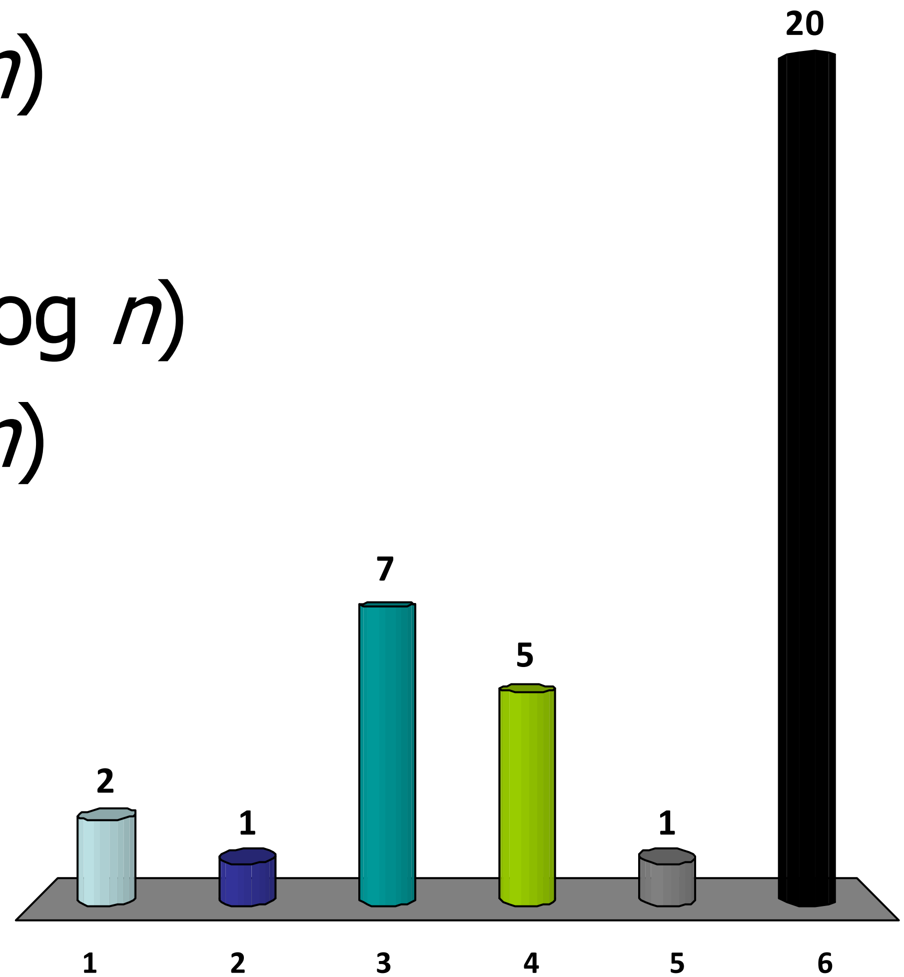
Details:

- What happens with duplicate keys?
  1. Assume no duplicate keys.
  2. Assume new insert overwrites old insert.

- What happens if you delete a non-existent key?
  1. Exception.
  2. Return.

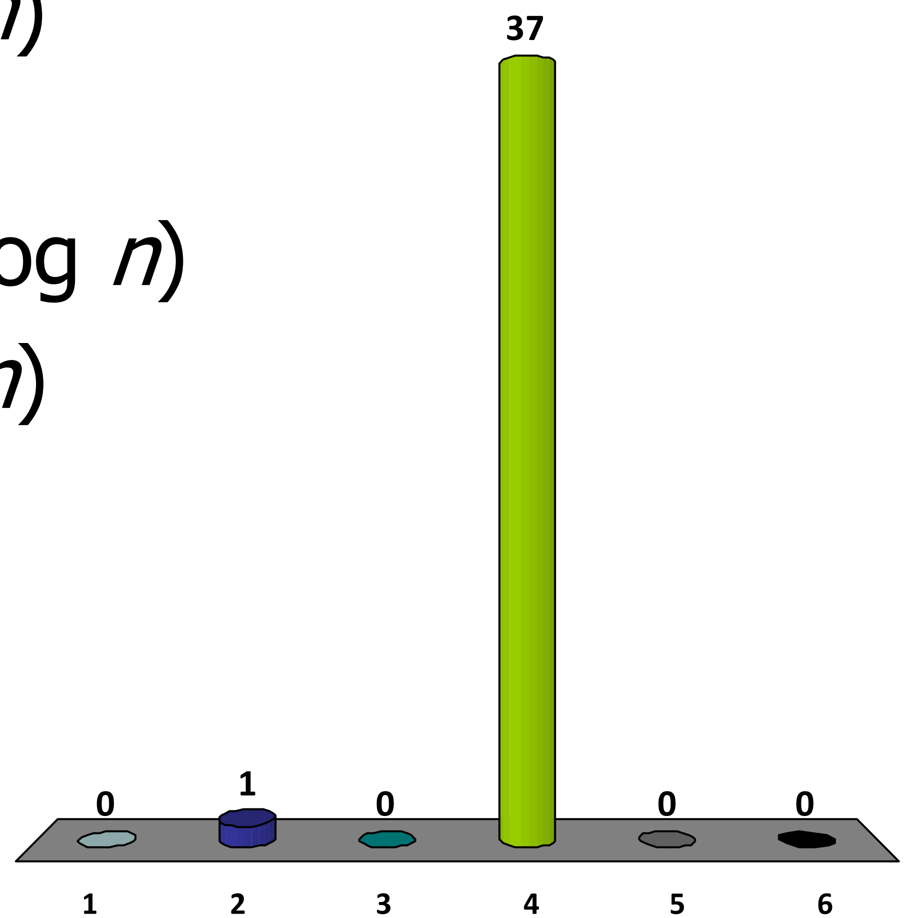
If you implement a dictionary with a Linked List, then: (C_I= cost insert, C_S=cost search)

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. $C_I = O(n)$, $C_S = O(n)$



If you implement a dictionary with an AVL tree, then: (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. $C_I = O(n)$, $C_S = O(n)$



Dictionary Data Structure

Implement a dictionary with:

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

Fast, fast, fast….

Dictionary Data Structure

Isn't O(1) search/insert impossible?

- Sorting takes $\Omega(n \log n)$.
  - How do you sort with a dictionary?
  - Only search/insert/delete.

Dictionary Data Structure

Isn't O(1) search/insert impossible?

- Sorting takes $\Omega(n \log n)$.
  - How do you sort with a dictionary?
  - Only search/insert/delete.

- Binary search takes $\Omega(\log n)$.
  - Impossible to search in fewer than $\log(n)$ steps.
  - But a dictionary finds an item in O(1) steps!!
  - Conclusion: dictionary is not comparison-based.

Dictionaries in Java

Dictionaries in Java

```
public interface  IDictionary<TKey, TData> {

    void insert(TKey key, TData data);

    boolean search(TKey key);

    TData getData(TKey key);

    void delete(TKey key);
}
```

Dictionary Interface in Java

java.util.Map<ktype, vtype>

- Parameterized by two types:
  - ktype (key)
  - vtype (value)
- No duplicate keys allowed.
- No *mutable* keys
  - If you use an *object* as a key, then you can't modify that object later.

Dictionary Interface in Java

java.util.Map<ktype, vtype>

`void clear()`

- Removes all from map.

`boolean containsKey(Object key)`

- Is the key in the map?

`Object get(Object key)`

- Returns the objects associated with the specified key.

`Object put(Object key, Object value)`

- Adds the key/value pair to the map

Dictionary Interface in Java

java.util.Map<ktype, vtype>

`Set entrySet()`

- Returns all entries in the map.

`Set keySet()`

- Returns all keys in the map.

`Collection values()`

- Returns all values in the map.

Note: not sorted, and not necessarily efficient to work with these sets/collections.

Dictionary 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

Dictionary 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.

Dictionaries 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!

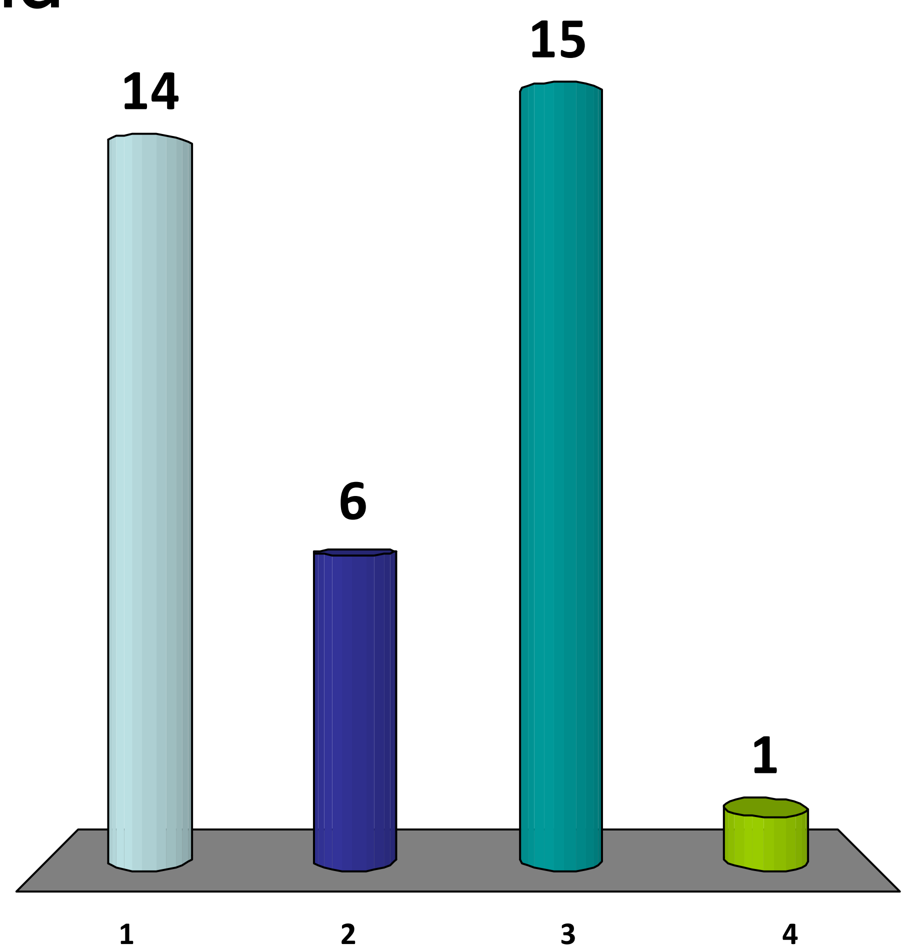
Dictionaries are Useful

Example 1: Pilot Scheduling

1. Check to see if feasible to schedule at time *t*.
2. Find schedule of pilot *p*.

Which can be solved with a dictionary?

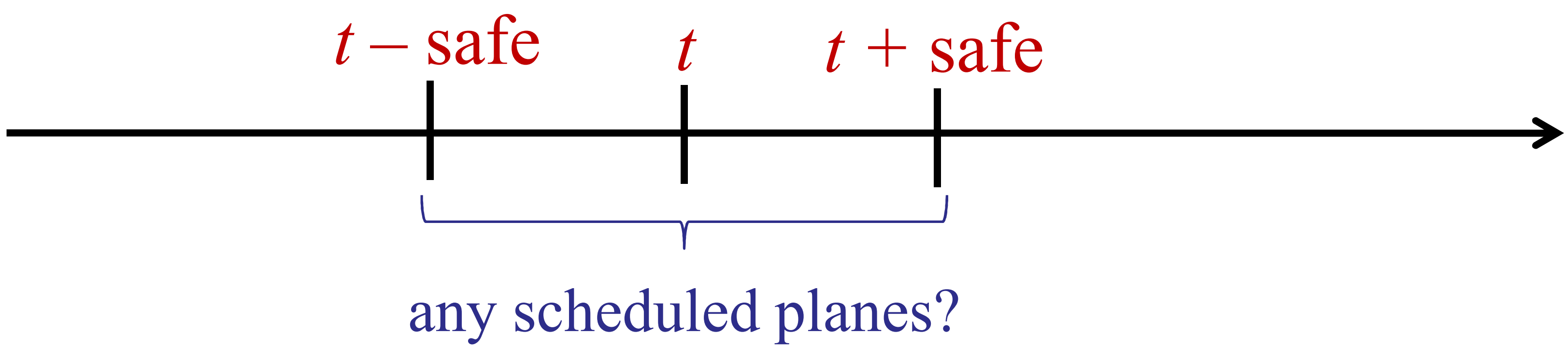
1. Both: scheduling and pilots info.
2. Only scheduling.
3. Only pilot info.
4. Neither.



Dictionaries are Useful

Example 1: Pilot Scheduling

1. Check to see if feasible to schedule at time t.
   - Hard with a dictionary!
   - Need to find out if there are any planes scheduled in the interval $[t, t \pm \text{safe\_distance}]$



Dictionaries 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 dictionary!
   - Can insert new pilots.
   - Can search for (and update) existing pilots.
   - Listing all pilots?

Dictionaries 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.

Dictionaries 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.

Dictionaries 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 $\lvert A \rvert$ and $\lvert B \rvert$ 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 (Dictionary)

(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**

Dictionaries are Useful

Example 2: Document Distance

- Step 1: Read in each document
  - Read and parse the file.
  - Put each (word, count) in a dictionary / HashMap.

Dictionary:

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

Dictionaries 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 = "";
}
```

Dictionaries 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*)

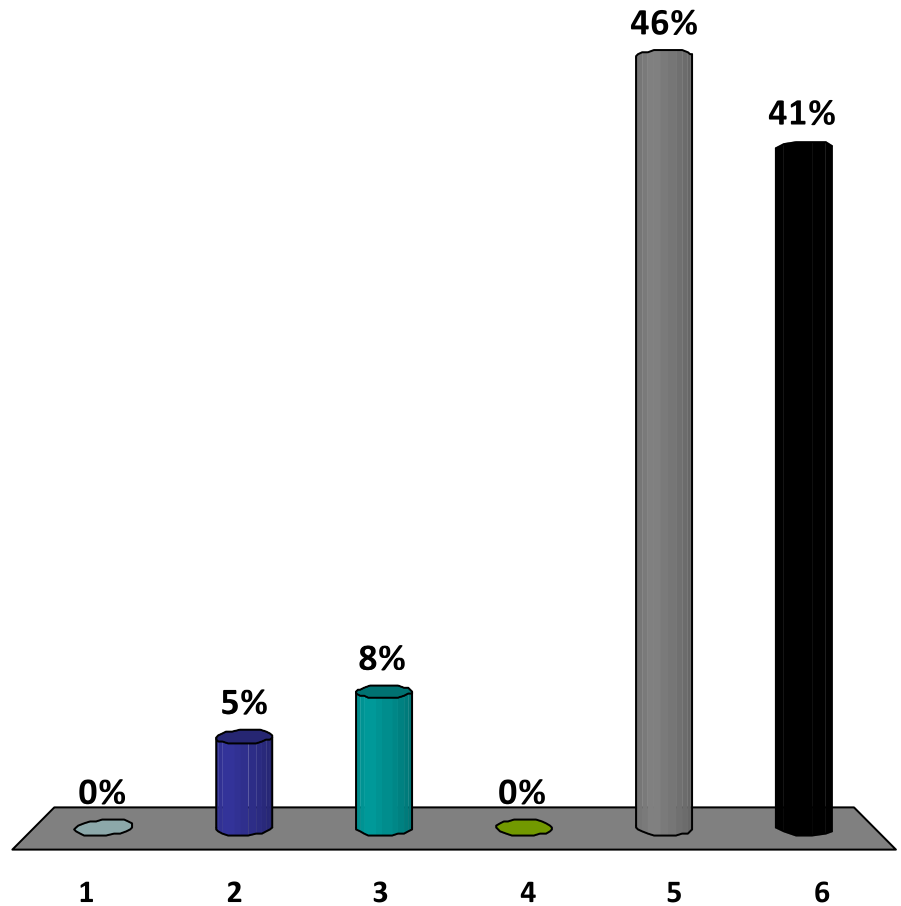
Version	Change	Running Time
Version 1		4,311.00s
Version 2	Better file handling	676.50s
Version 3	Faster sorting	6.59s
Version 4	No sorting!	2.35s

Dictionaries are Useful

Example 3: DNA Analysis

How similar is Chimpanzee DNA to Human DNA?

1. 20-50%
2. 70-79%
3. 80-90%
4. 80-95%
5. 96-99%
6. Who are you calling a chimp, chump?



Dictionaries are Useful

Example 3: DNA Analysis

- How similar is chimp DNA to human DNA?
- Problem:
  - Given human DNA string: ACAAGCGGTAA
  - Given chimp DNA string: CCAAGGGGTAA
  - How similar are they?
- Similarity = longest common substring
  - Implies a gene that is shared by both.
  - Count genes that are shared by both.

Dictionaries are Useful

Example 3: DNA Analysis

- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

Dictionaries are Useful

Naïve Algorithm: strings *A* and *B*

L = length(*A*);

for (L = n down to 1)

for every substring X1 of A of length L:

for every substring X2 of B of length L:

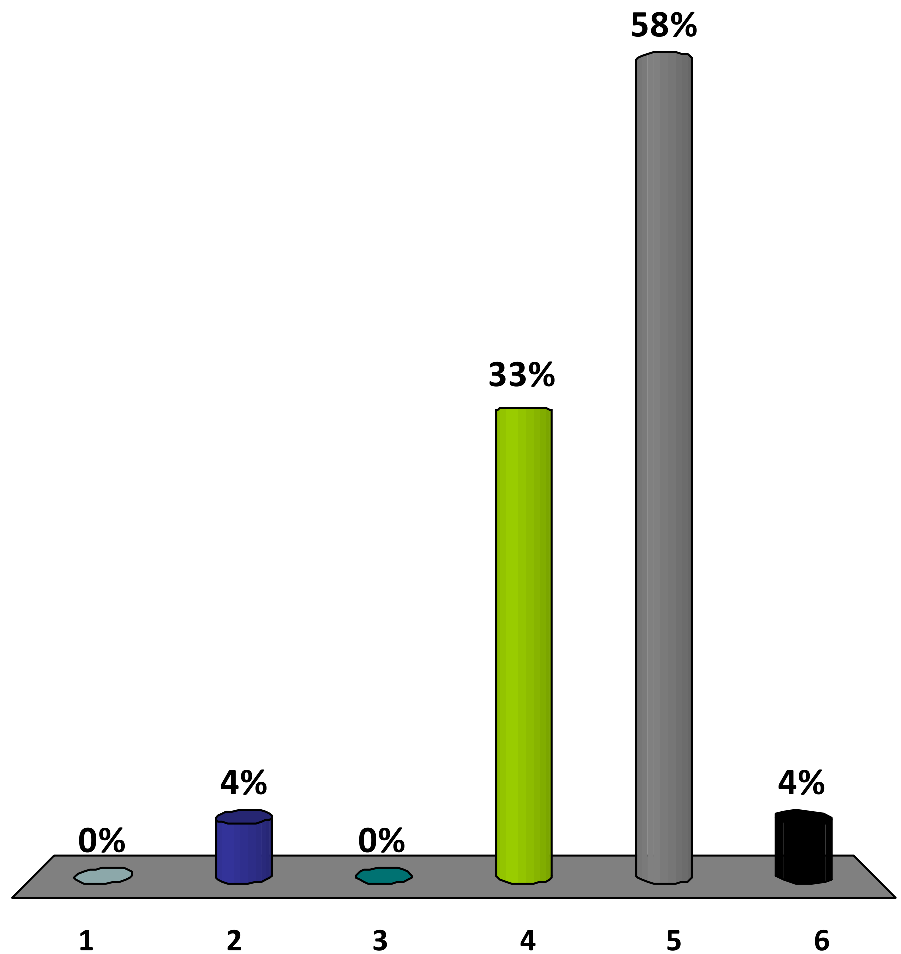
if (X1==X2) then return X1;

Example: ALGORITHM ARITHMETIC

- L=3 : X1= ALG → compare to ARI, ART, ARH, …

What is the running time?

1. $O(\log n)$
2. $O(n)$
3. $O(n \log n)$
4. $O(n^2)$
5. $O(n^3)$
6. $O(n^4)$



Dictionaries are Useful

Naïve Algorithm: strings *A* and *B*

L = length(*A*);

for (L = n down to 1) ← Loop *n* times.

for every substring X1 of A of length L: ← *n* substrings

for every substring X2 of B of length L: ← *n* substrings

if (X1==X2) then return X1; ← comparison costs: $O(n)$

Total cost: $O(n^4)$

Dictionaries are Useful

Example 3: DNA Analysis

- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

- Another idea:
  - Binary search!
  - Don’t search every length L.
  - Start with L = length(A) / 2.
  - Search until you find a match for some length L.

Dictionaries are Useful

Binary Search Algorithm: strings *A* and *B*

repeat until done…

L = length(*A*) /2;

for every substring X1 of A of length L:

for every substring X2 of B of length L:

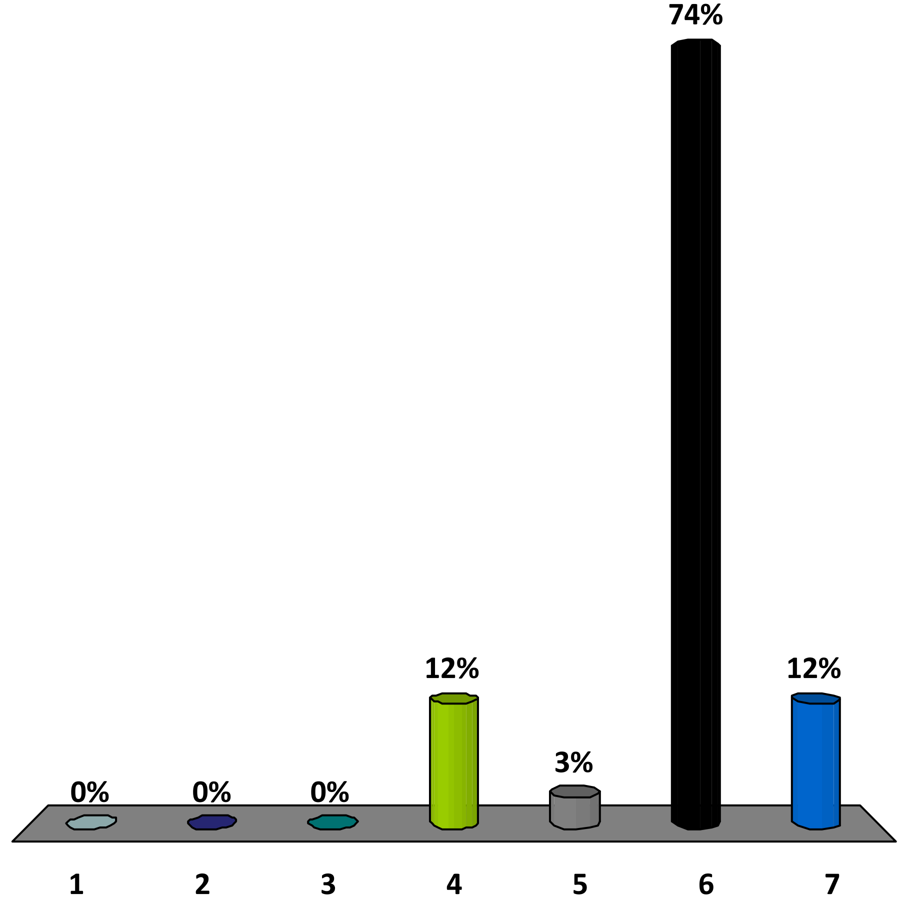
if (X1==X2) then found=true;

if (found) then increase L

else decrease L

What is the running time?

1. O(n)
2. O(n log n)
3. O(n^2)
4. O(n^2log n)
5. O(n^3)
6. O(n^3log n)
7. O(n^4)



Dictionaries are Useful

Binary Search Algorithm: strings *A* and *B*

repeat until done…

L = length(*A*) /2;

for every substring X1 of A of length L:

for every substring X2 of B of length L:

if (X1==X2) then found=true;

if (found) then increase L

else decrease L

Cost: $O(n^3 \log n)$

Dictionaries are Useful

Example 3: DNA Analysis

- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

- Another idea:
  - Put every substring from first string into a dictionary.
  - Lookup every substring from second string in a dictionary.

Dictionaries are Useful

Example 3: DNA Analysis

- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

- Add to dictionary:
  - A, AL, ALG, ALGO, ALGOR, ALGORI, ALGORIT, ALGORITH, …
  - L, LG, LGO, LGOR, LGORI, LGORIT, LGORITH, LGORITHM
  - G, GO, GOR, GORI, GORIT, GORITH, GORITHM
  - …

Dictionaries are Useful

Example 3: DNA Analysis

- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

- Search in dictionary:
  - A, AR, ARI, ARIT, ARITH, ARITHM, ARITHME, ARITHMET, …
  - R, RI, RIT, RITH, RITHM, RITHME, RITHMET, RITHMETI, …
  - I, IT, ITH, ITHM, ITHME, ITHMET, ITHMETI, ITHMETIC
  - …

Dictionaries are Useful

Example 3: DNA Analysis

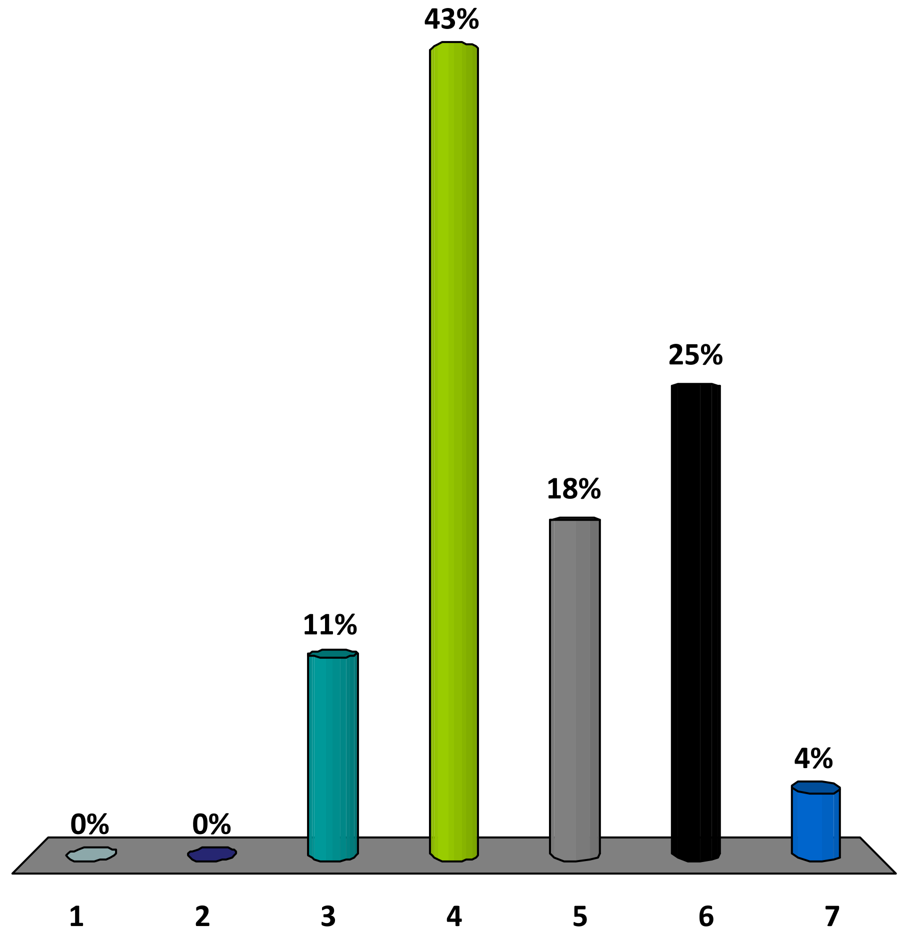
- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

- Search in dictionary:
  - A, AR, ARI, ARIT, ARITH, ARITHM, ARITHME, ARITHMET, …
  - R, RI, RIT, RITH, RITHM, RITHME, RITHMET, RITHMETI, …
  - I, IT, ITH, ITHM, ITHME, ITHMET, ITHMETI, ITHMETIC
  - …

Assume insert/search are O(1). What is the running time of this algorithm?

1. $O(1)$
2. $O(\log n)$
3. $O(n \log n)$
4. $O(n^2)$
5. $O(n^2 \log n)$
6. $O(n^3)$
7. $O(n^3 \log n)$



Dictionaries are Useful

Example 3: DNA Analysis

- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

- There are $O(n^2)$ substrings.
- To add a substring of length k takes time $O(k)$:
  - To add the substring to the dictionary, you have to at least read the whole string!
- Total running time: $O(n^3)$

Dictionaries are Useful

Example 3: DNA Analysis

- Long common substring (text):

  ALGORITHM vs. ARITHMETIC

- Now, binary search again:
  - For log n values of length L:
    - Add all O(n) substrings of length L from A.
    - Search all O(n) substrings of length L from B.
    - Adjust L and continue.
  - Running time: O(n^2 log n).
  - Next lecture: do better!

Coming up next… Hash Tables

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 $m = 9$.

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.

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

Example: insert(4, Seth)

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 word “hippopotamus”?
  - 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."

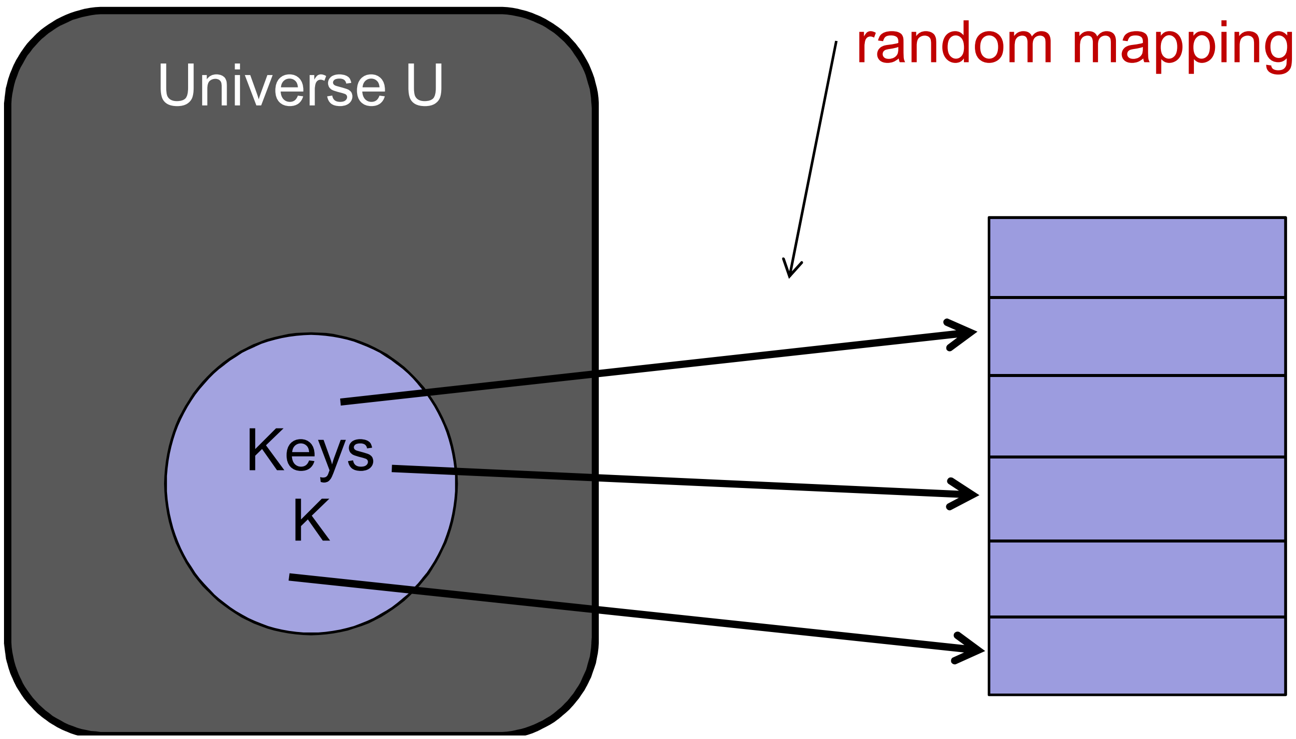
- Everything is just a sequence of bits.
- Treat those bits as a number.

- 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:

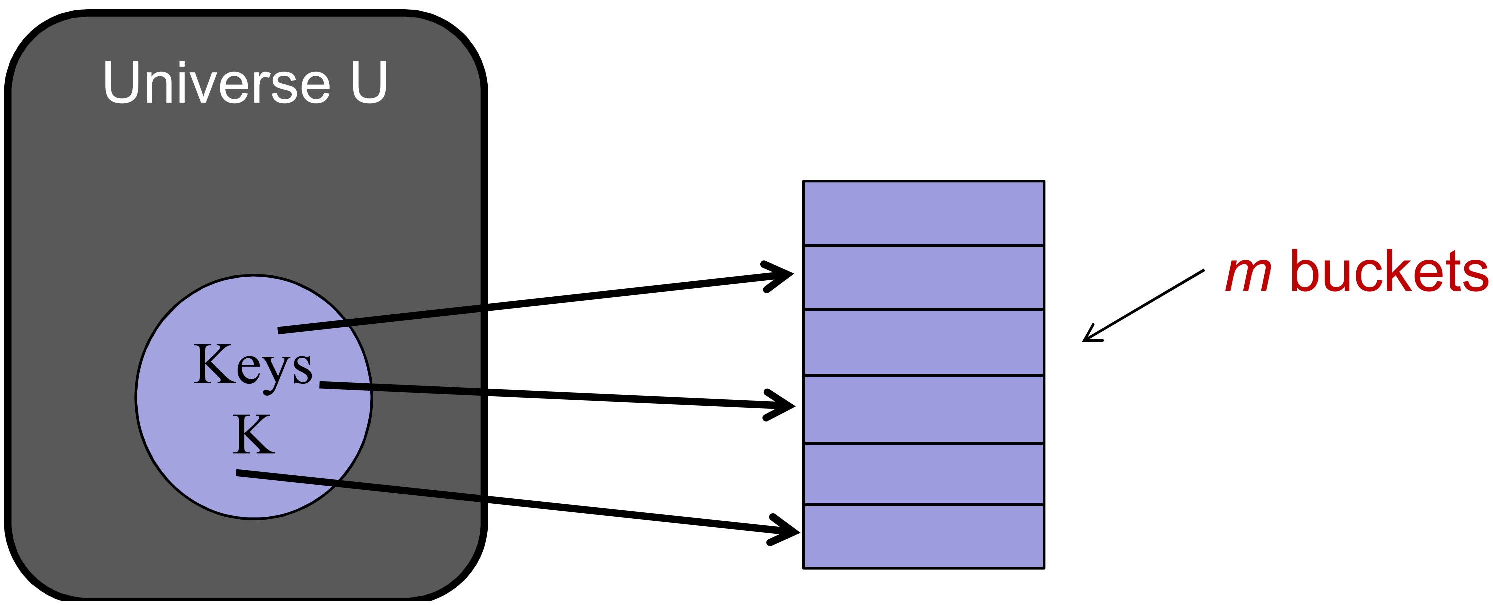
- Huge universe U of possible keys.
- Smaller number n of actual keys.
- How to map n keys to $m \approx n$ buckets?



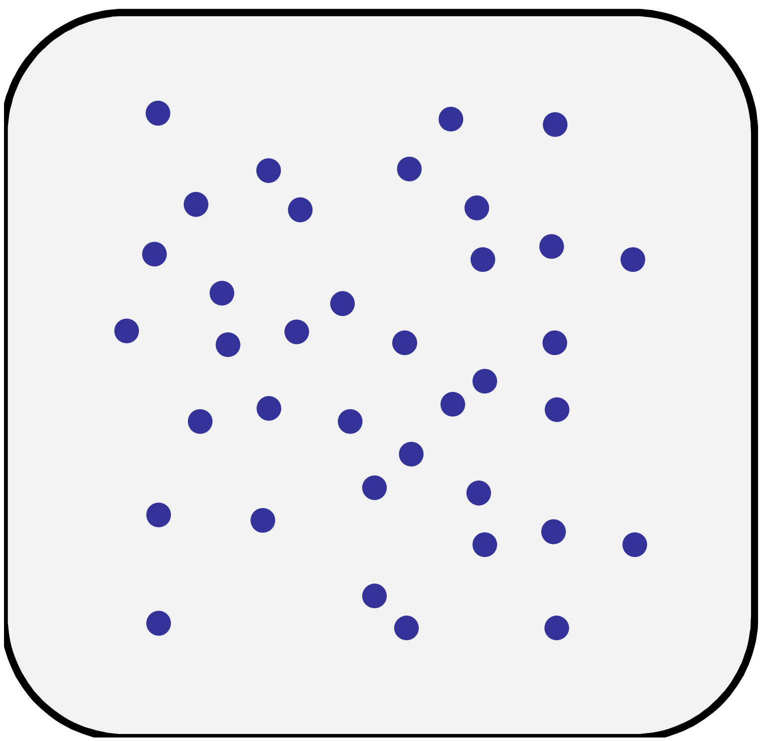
Hash Functions

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

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



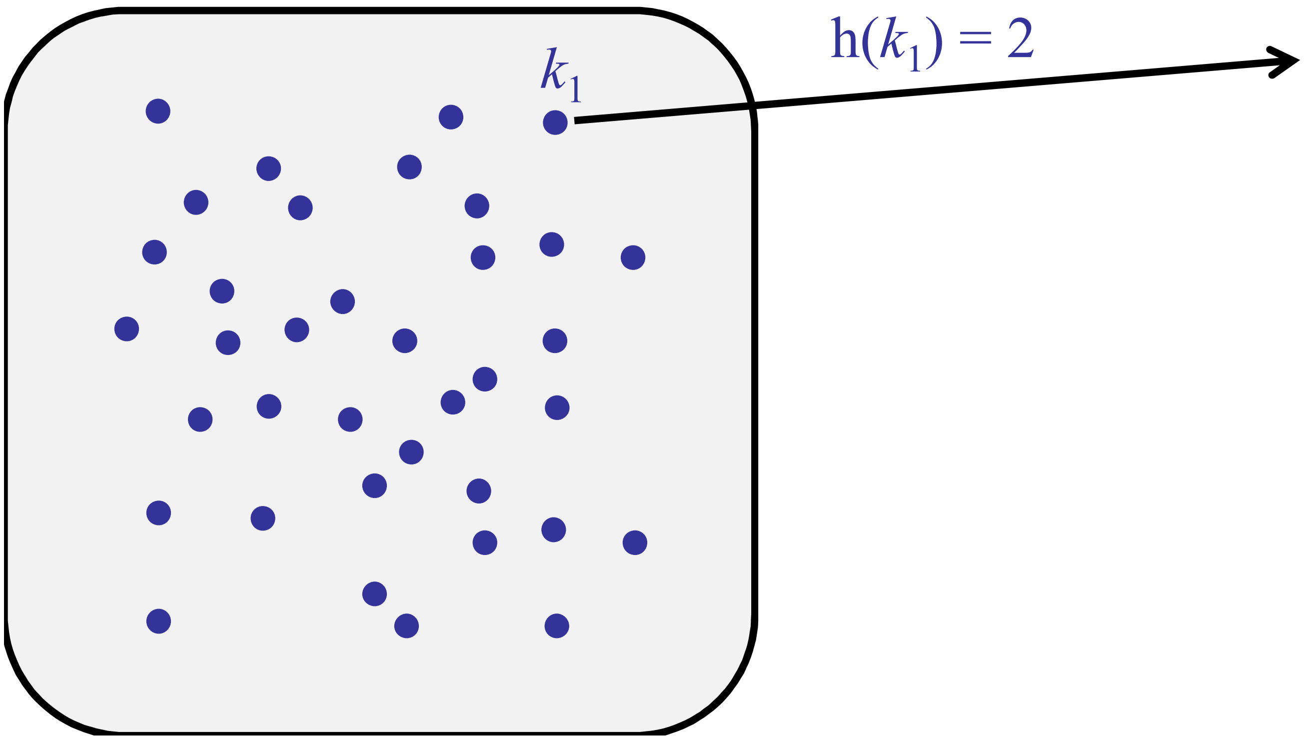
Hash Functions



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)

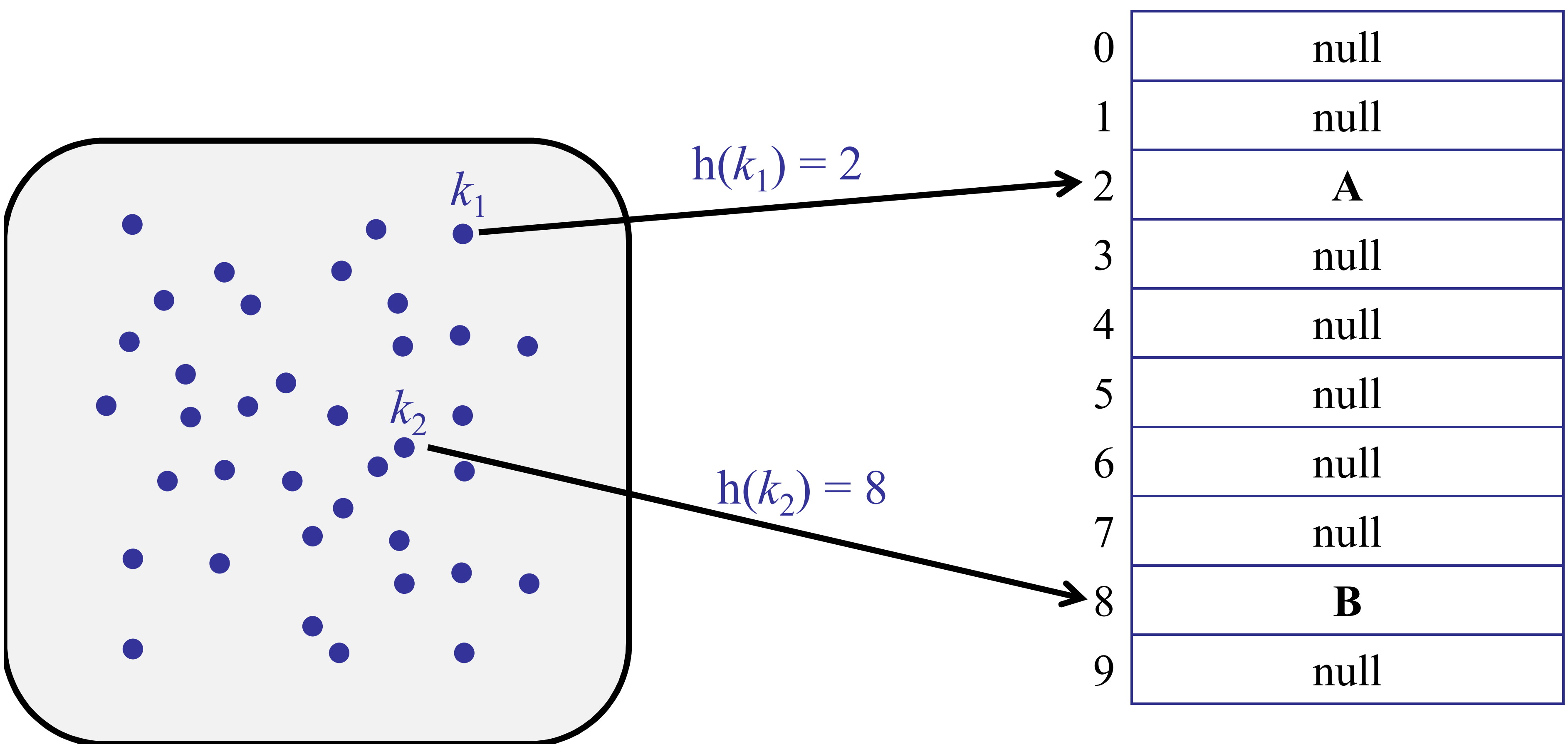


0	null
1	null
2	**A**
3	null
4	null
5	null
6	null
7	null
8	null
9	null

Hash Functions

insert(k_1, A)

insert(k_2, B)



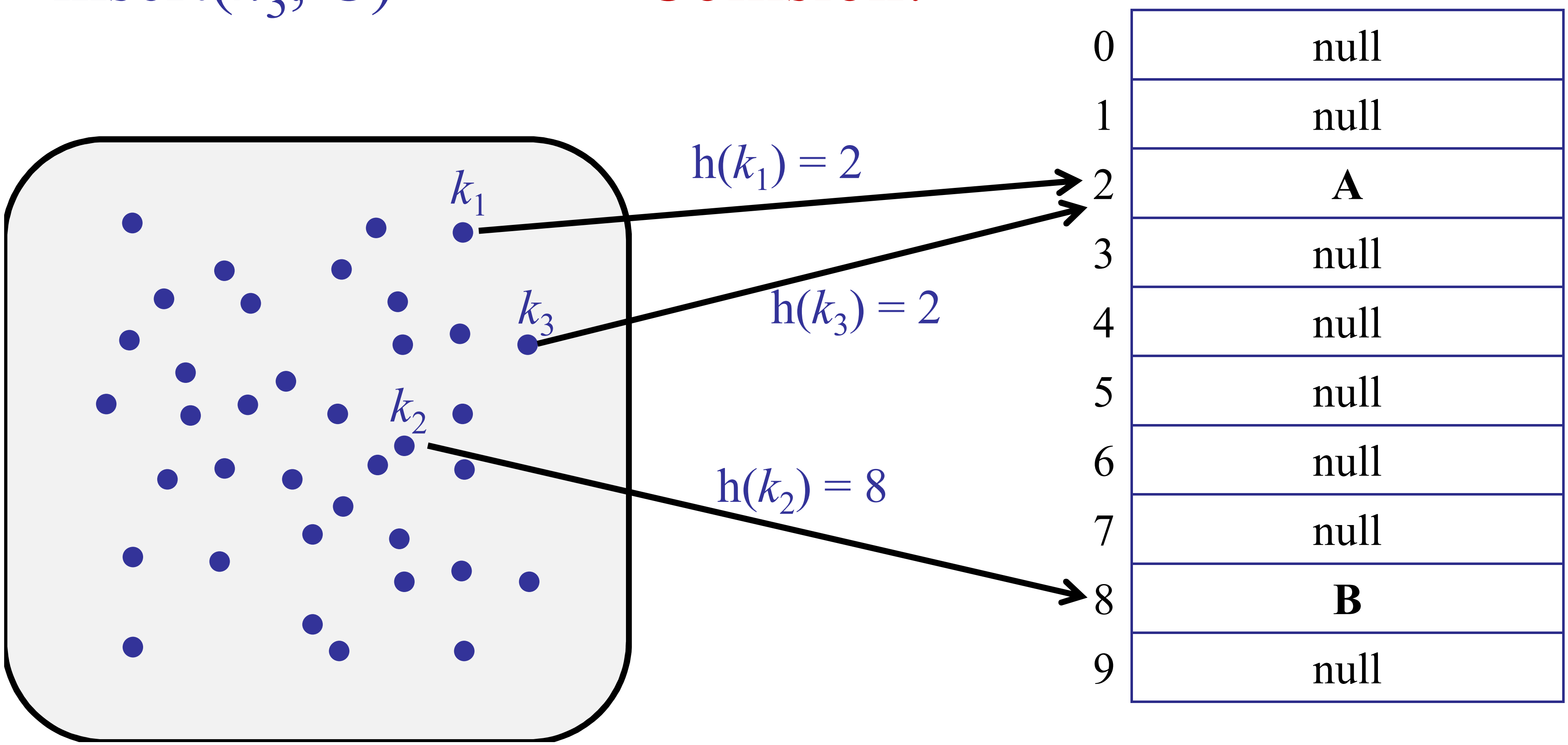
Hash Functions

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:

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

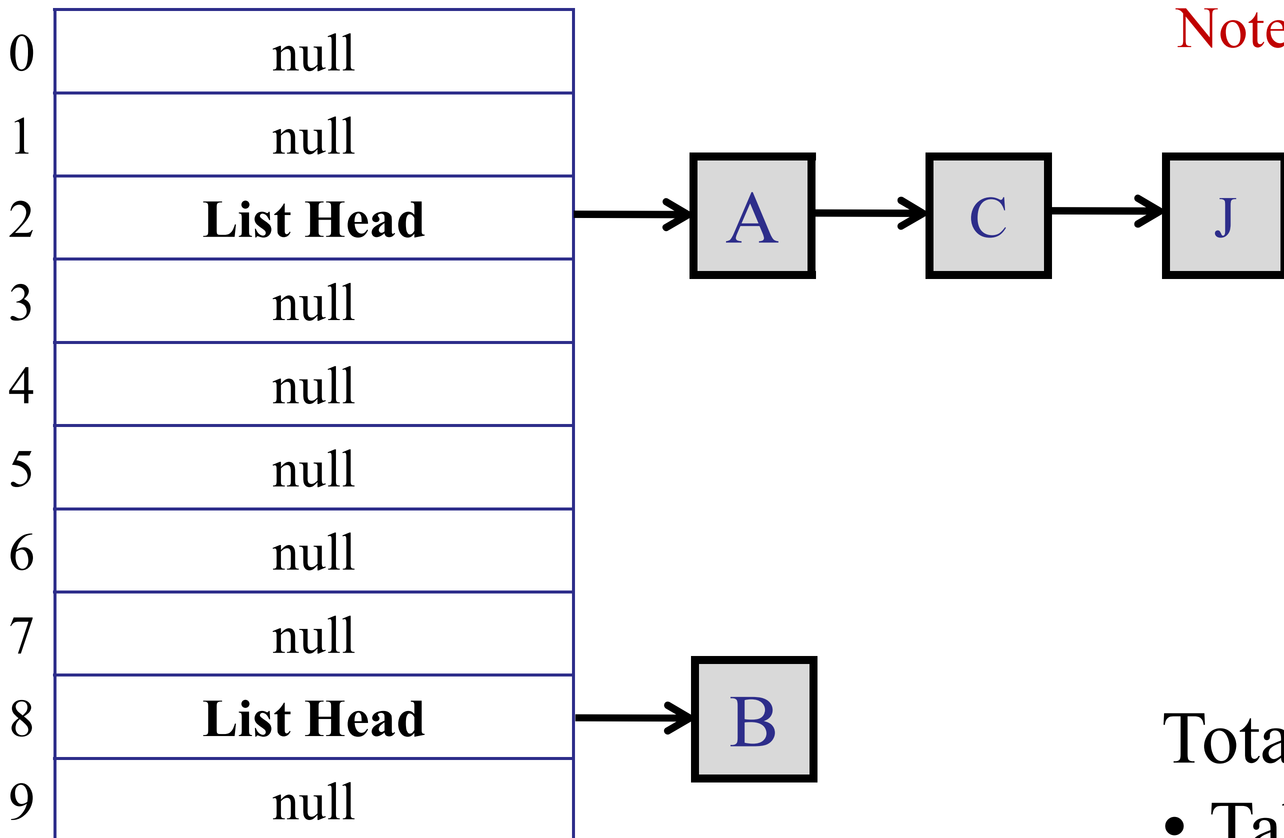
- Unavoidable!
  - The table size is smaller than the universe size.
    - Some keys must collide!
    - Pigeonhole principle.

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 lecture)
  - 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: $O(m + n)$

- Table size: m
- Linked list size: n

Hashing with Chaining

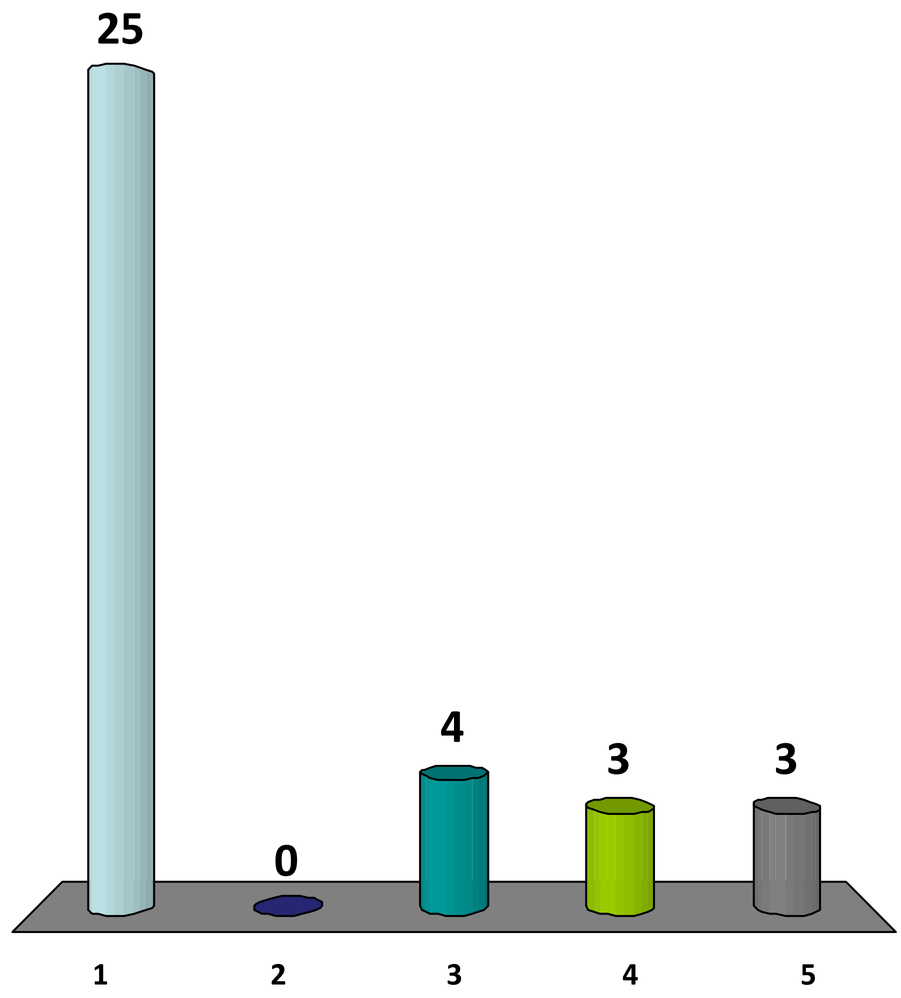
Operations:

- 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 cost of inserting a (key, value)?

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



Hashing with Chaining

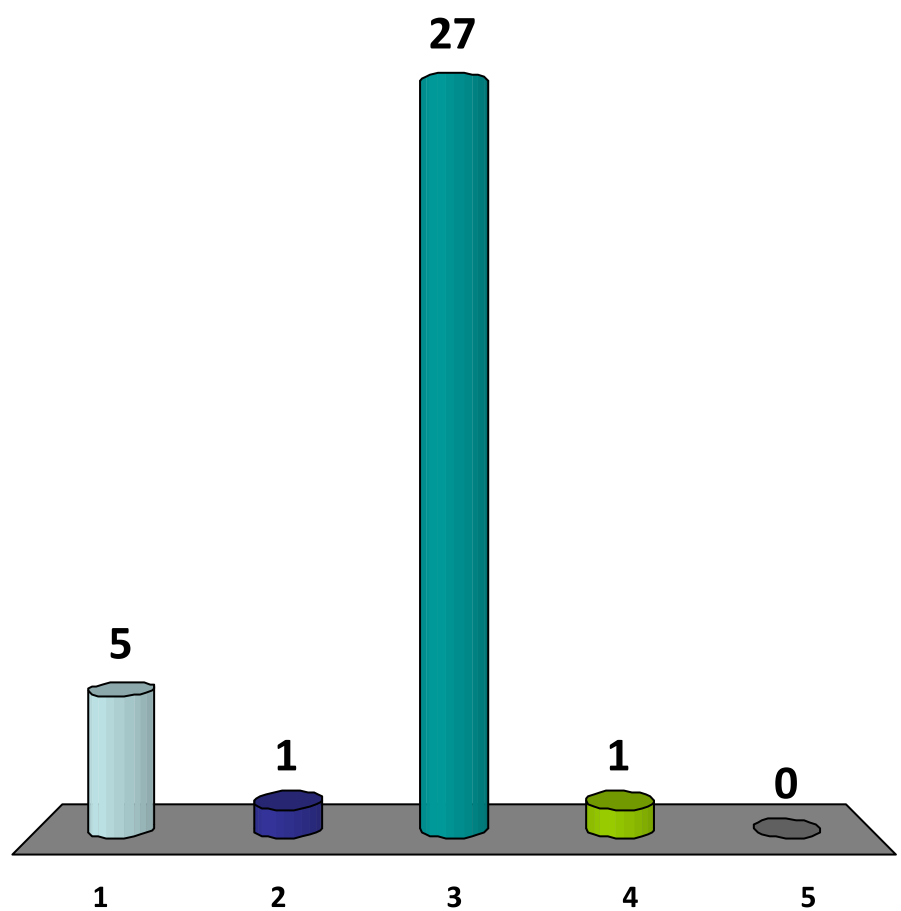
Operations:

- 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 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:

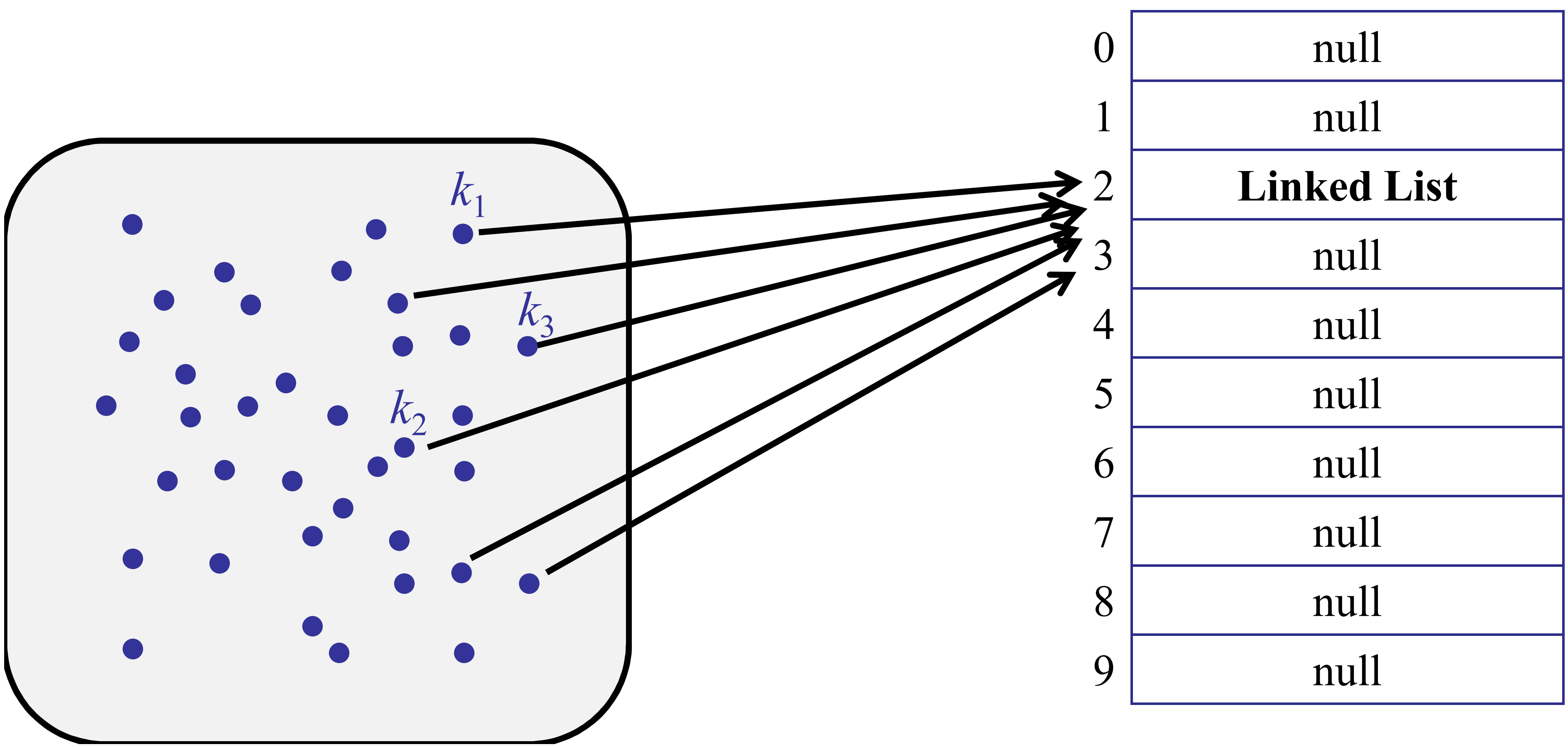
- insert(key, value)
  - Calculate h(key)
  - Lookup h(key) and add (key,value) to the linked list.

- 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)!



Let's be optimistic today.

The Simple Uniform Hashing Assumption

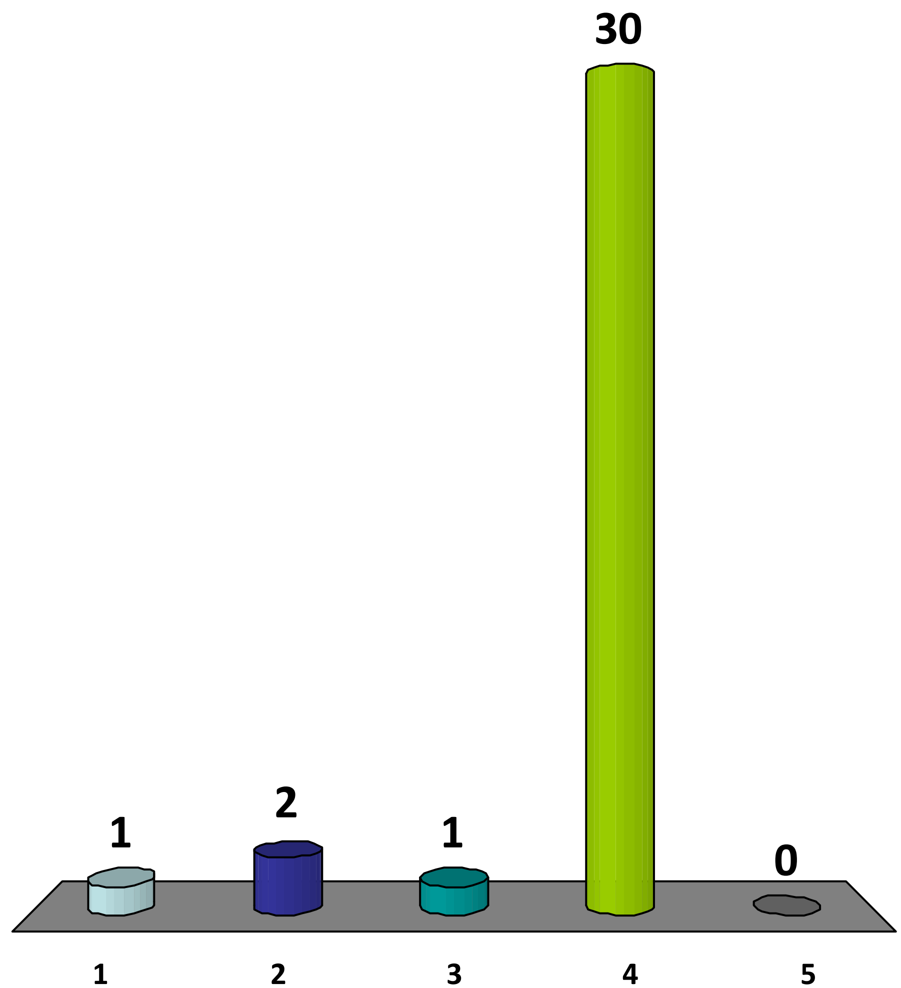
- Every key is equally likely to map to every bucket.
- Keys are mapped independently.

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 <u>Simple Uniform Hashing</u> Assumption

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

  = average #items / bucket.

- Expected search time = 1 + m/n

hash function + array access

linked list traversal

Let's be optimistic today.

The <u>Simple Uniform Hashing</u> Assumption

- Assume:
  - n items
  - $m = \Omega(n)$ buckets

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

$$= O(1)$$

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 [1..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: 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, really, 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

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

- Idea: choose $m = 2^x$

  Very, very fast to calculate $k \bmod m == k >> x$

Designing Hash Functions

Division Method

– Idea: choose $m = 2^x$

Very, very fast to calculate $k \bmod m == k >> x$

– Problem: Regularity

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

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

even

even

Designing Hash Functions

Division Method

– Assume k and m have common divisor d.

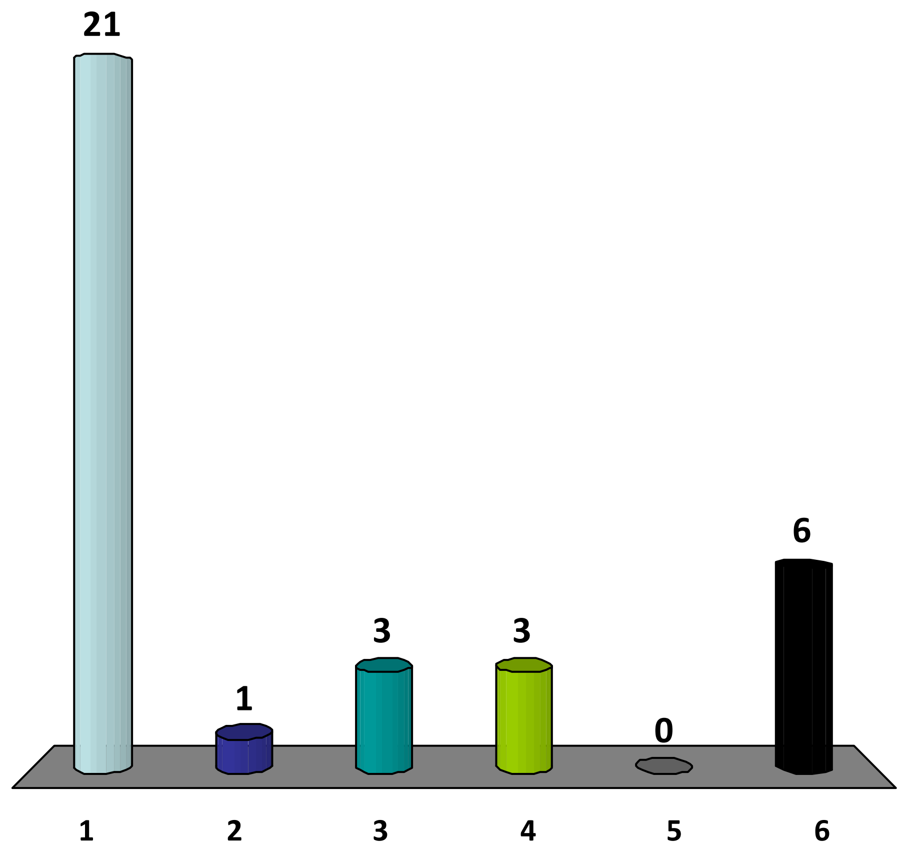
$$k \bmod m + i{*}m = k$$

divisible by d

– Implies that h(k) is divisible by d.

If every d is a divisor of m and every 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.

Designing Hash Functions

Multiplication Method

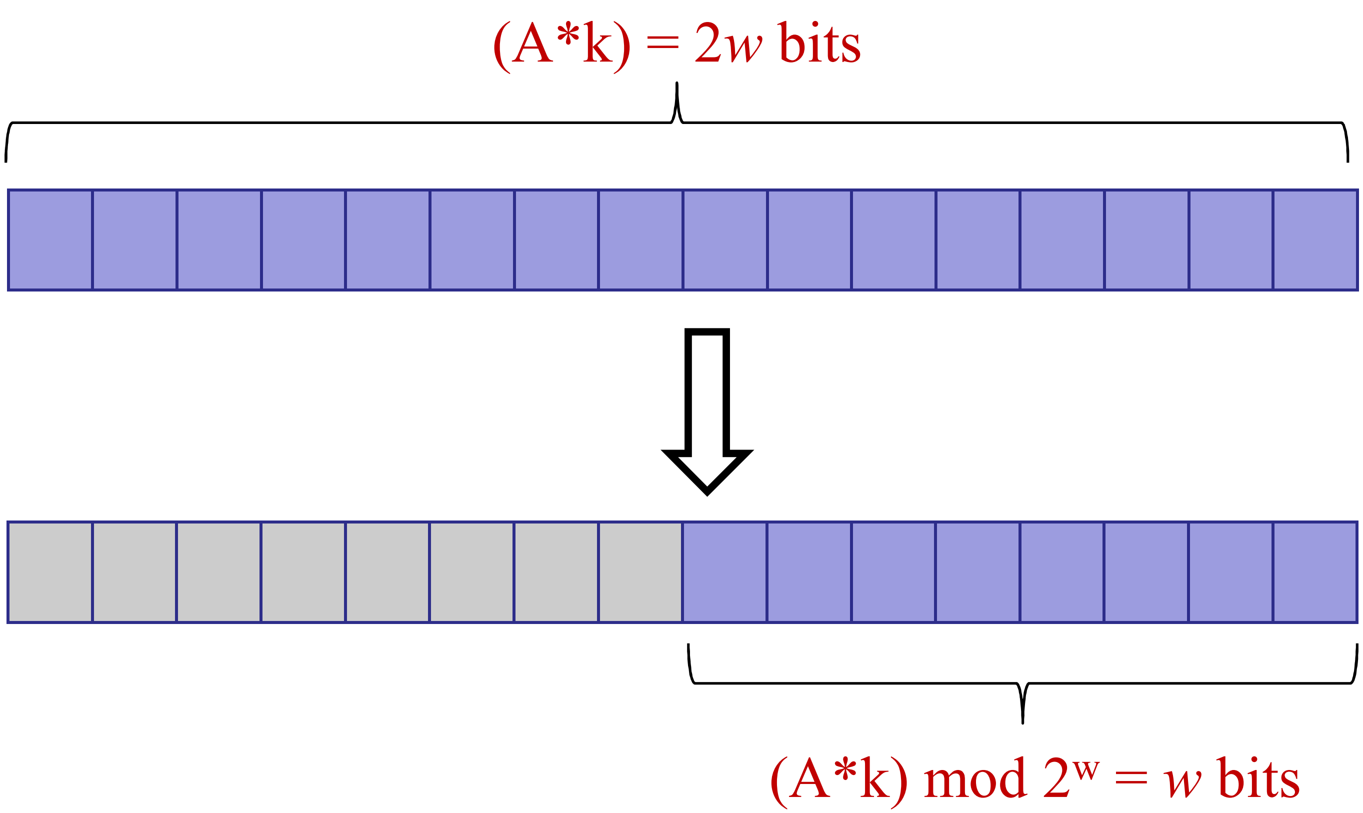
- Fix table size: $m = 2^r$, for some constant r.
- Fix word size: w, size of a key in bits.
- Fix (odd) constant A.

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

Designing Hash Functions

Multiplication Method

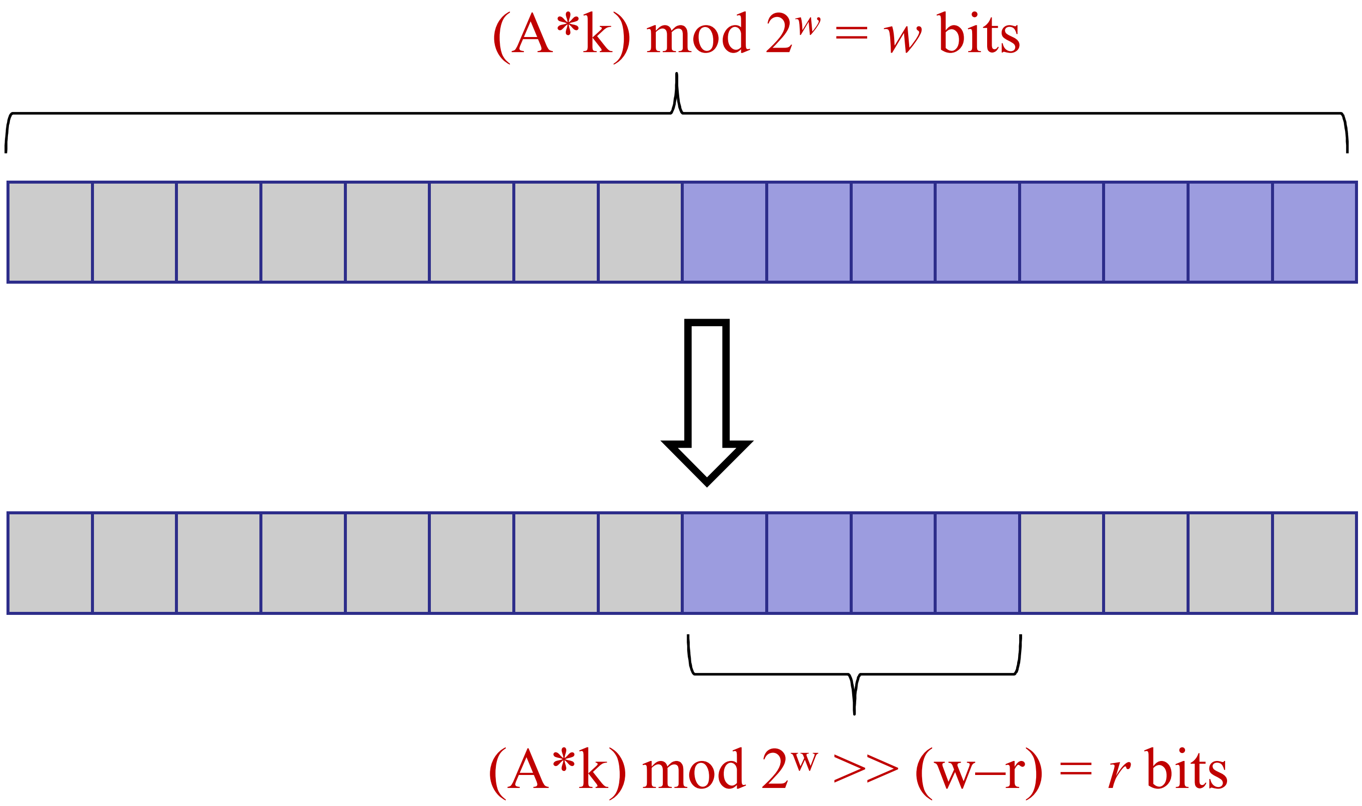
- Given m, w, r, A: $h(k) = (Ak) \bmod 2^w \gg (w - r)$



Designing Hash Functions

Multiplication Method

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



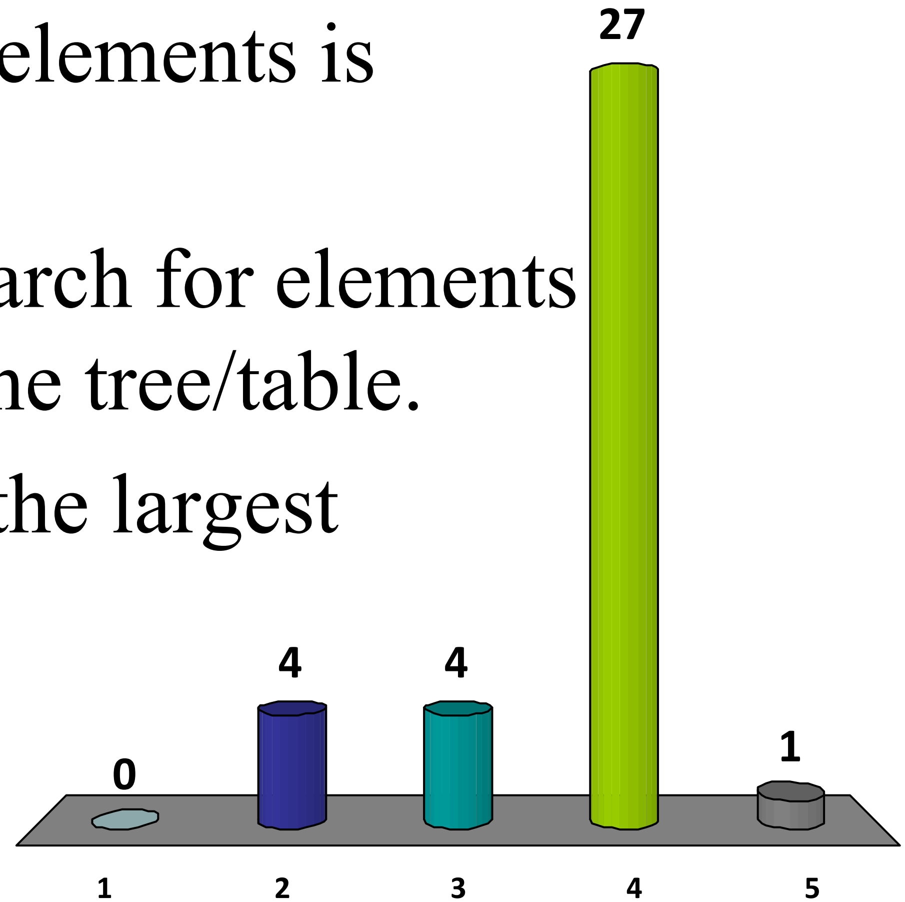
Designing Hash Functions

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.

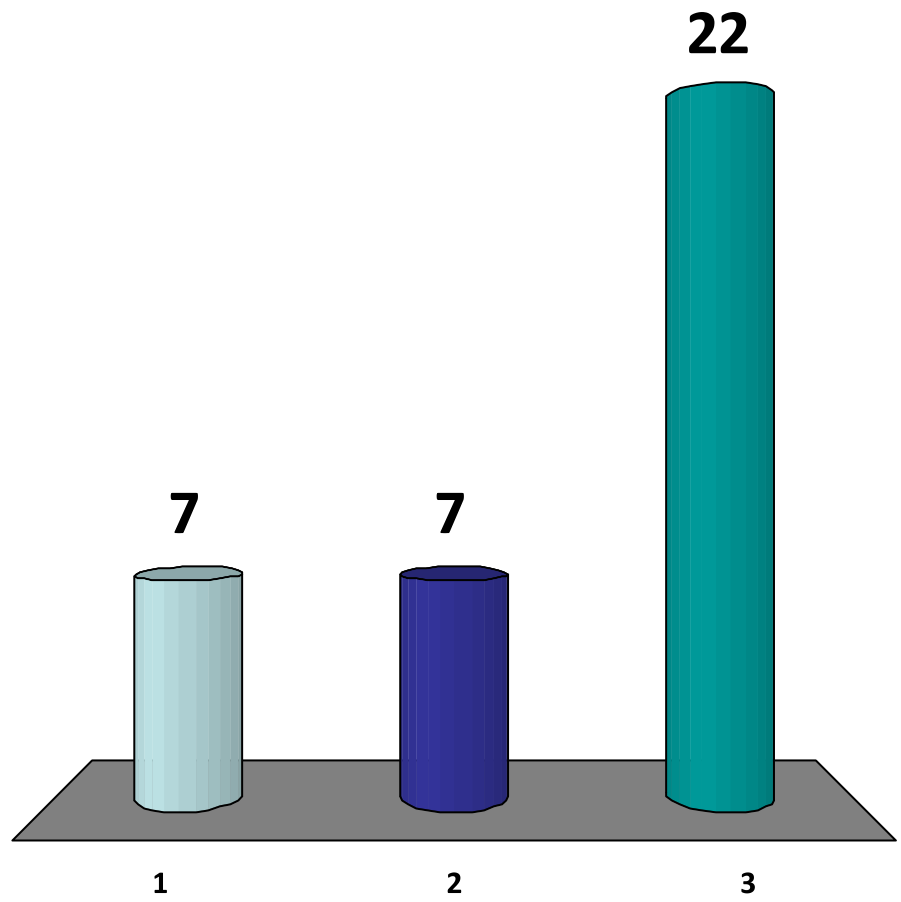
When is a BST better than a Hash Table?

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 find the largest element.
5. Never.



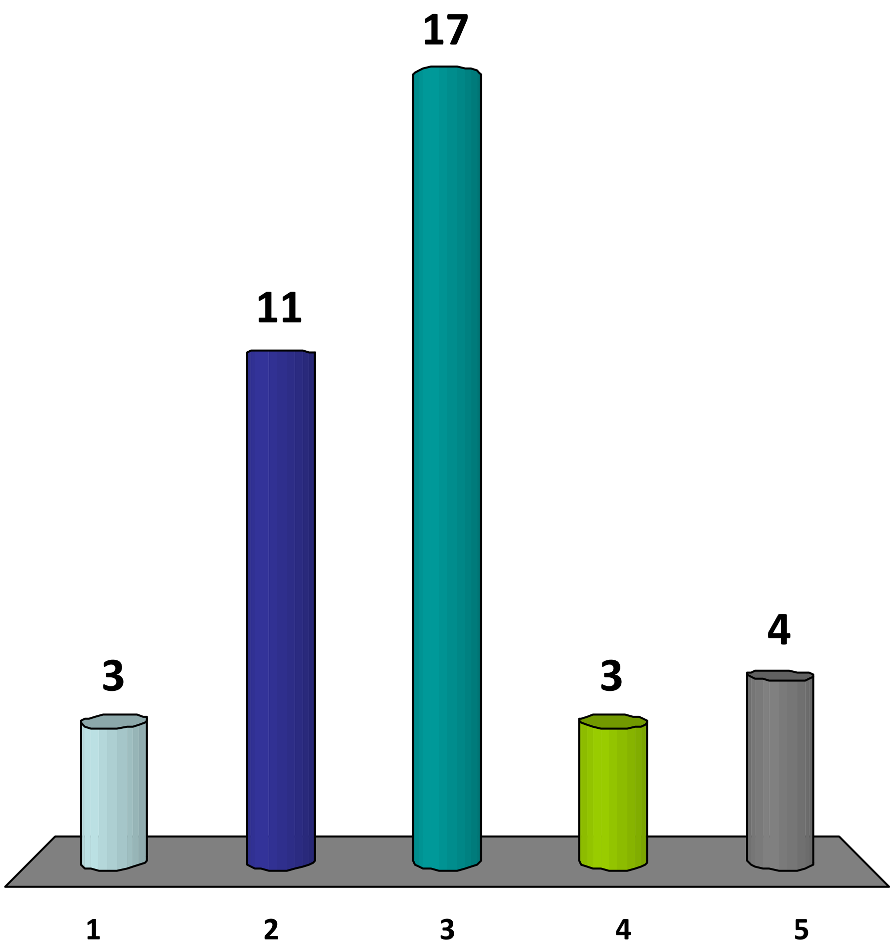
Can you easily extend a dictionary / hash table to maintain the order in which items are inserted??

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



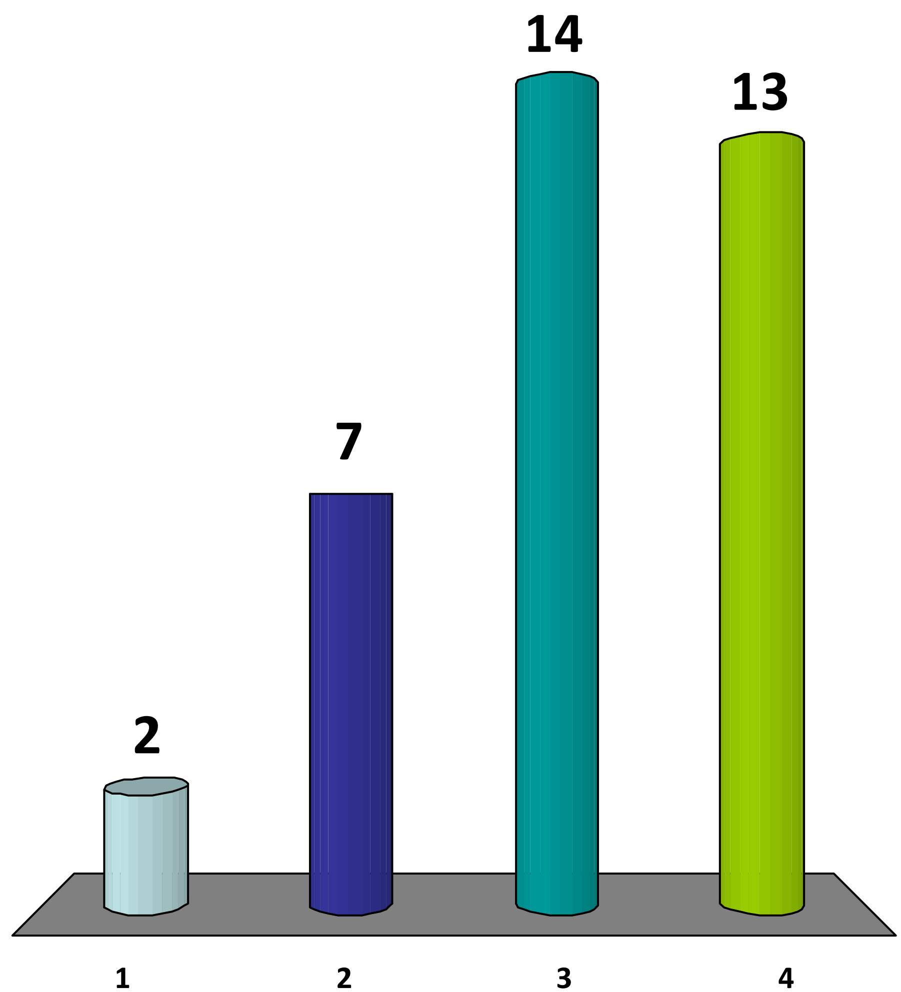
Which of the following is *not* a problem with a direct access table?

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.



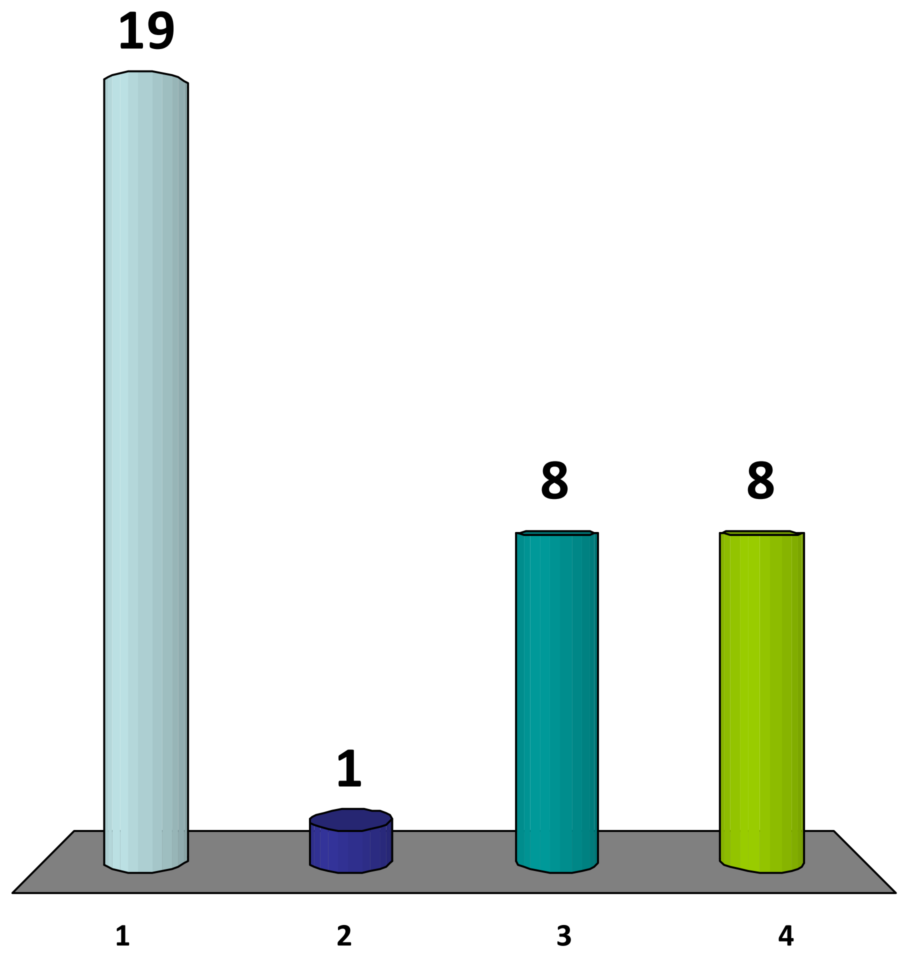
Enumerating keys in a hash table is fastest when:

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



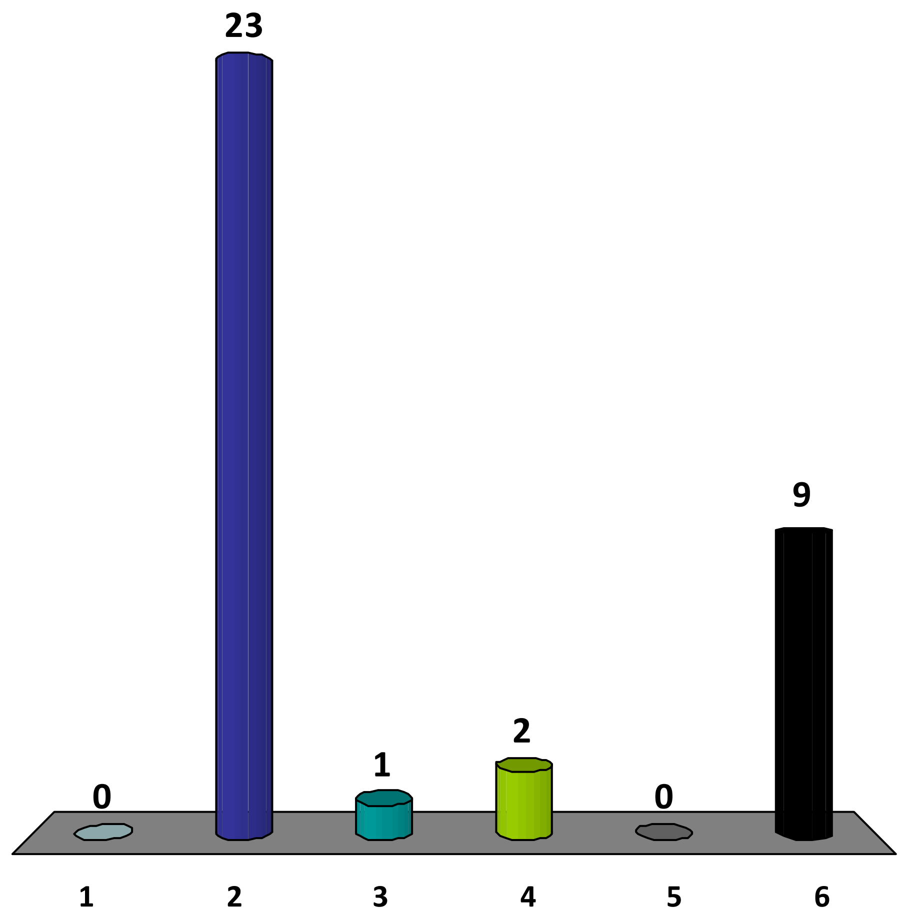
Searching in a hash table is fastest when:

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



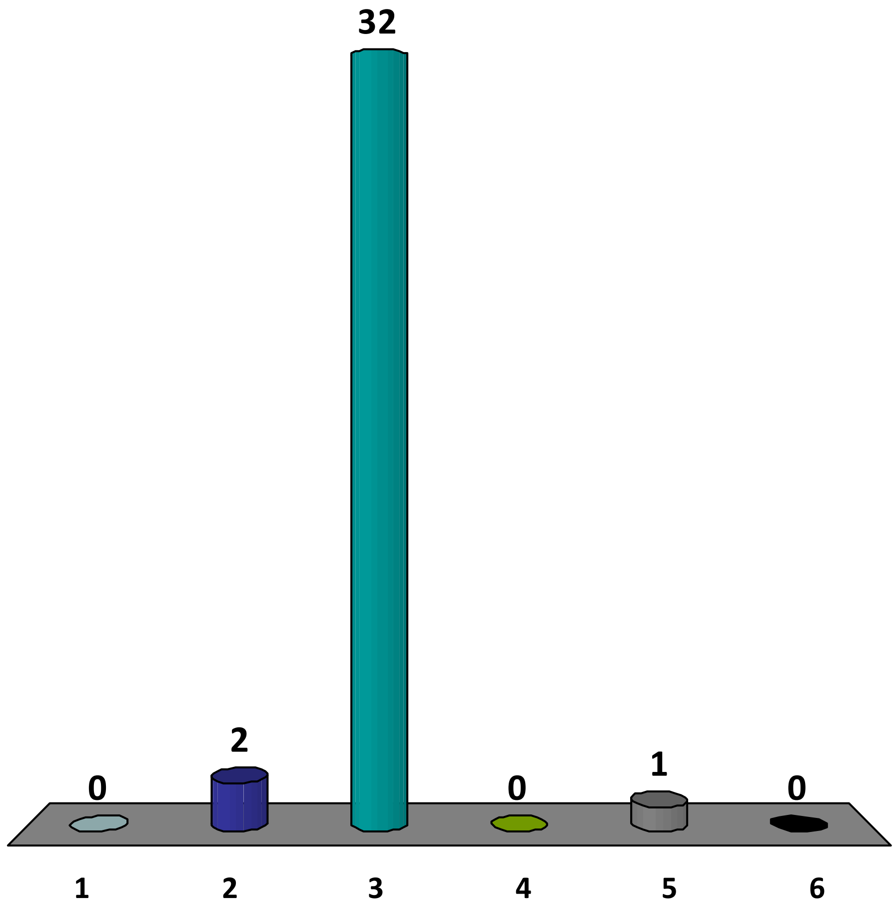
For the division method, which of the following is a good table size?

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



32 >> 3 = ?

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



Summary

Dictionaries are pervasive

- You find them everywhere!

Hash tables are fast, efficient dictionaries.

- 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.