CS2020
Data Structures and Algorithms

Welcome!

Problem Set 5

When is it due??

- Originally: today…
- Tomorrow before 2pm: ok…

Administrative

Coding Quiz

- During Discussion Groups this week.
- Location:
  - Wed. 2-4pm: AS6/425
  - Thurs. 12-2pm: I3/3-46
  - Thurs. 2-4pm: I3/3-46 and I3/3-47
  - Thurs. 5-7pm: I3/3-46

- Don’t skip Discussion Group this week!

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.
  - Difficulty is *not* uniform.
  - Difficulty is not the same as points.

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.

Today

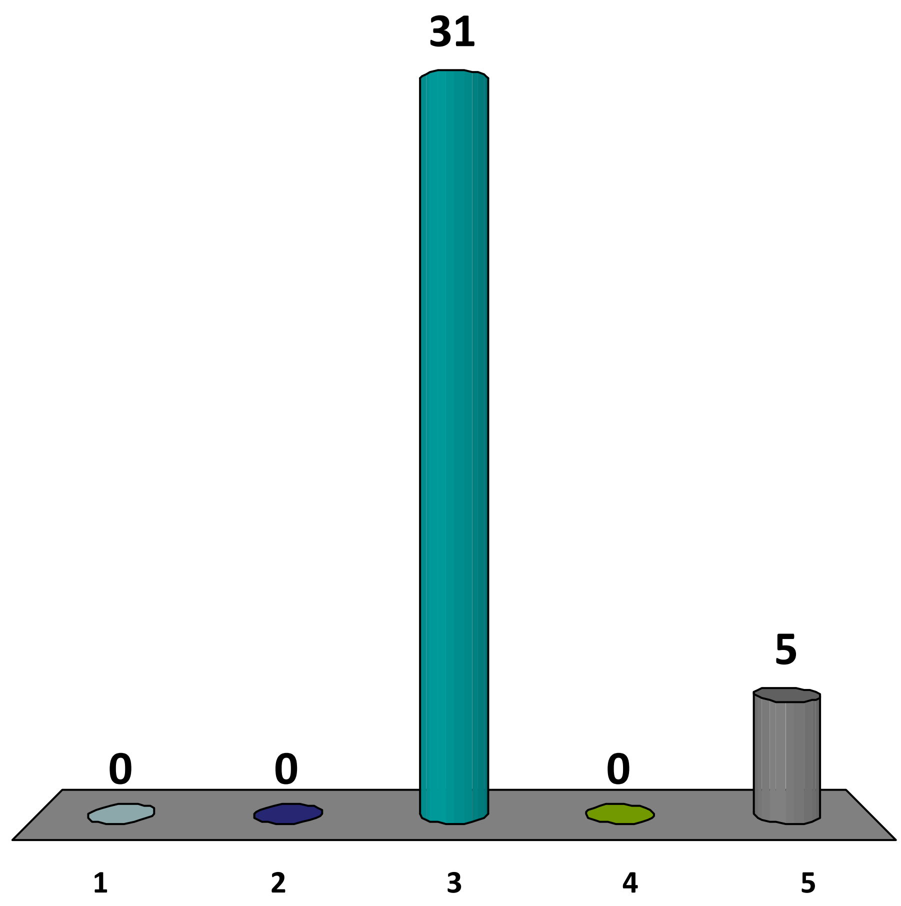
Hash Tables

- Choosing a good table size.
- Amortized Analysis
- Better DNA Analysis

Review: Dictionary Abstract Data Type

Which of the following is *not* typically a dictionary operation?

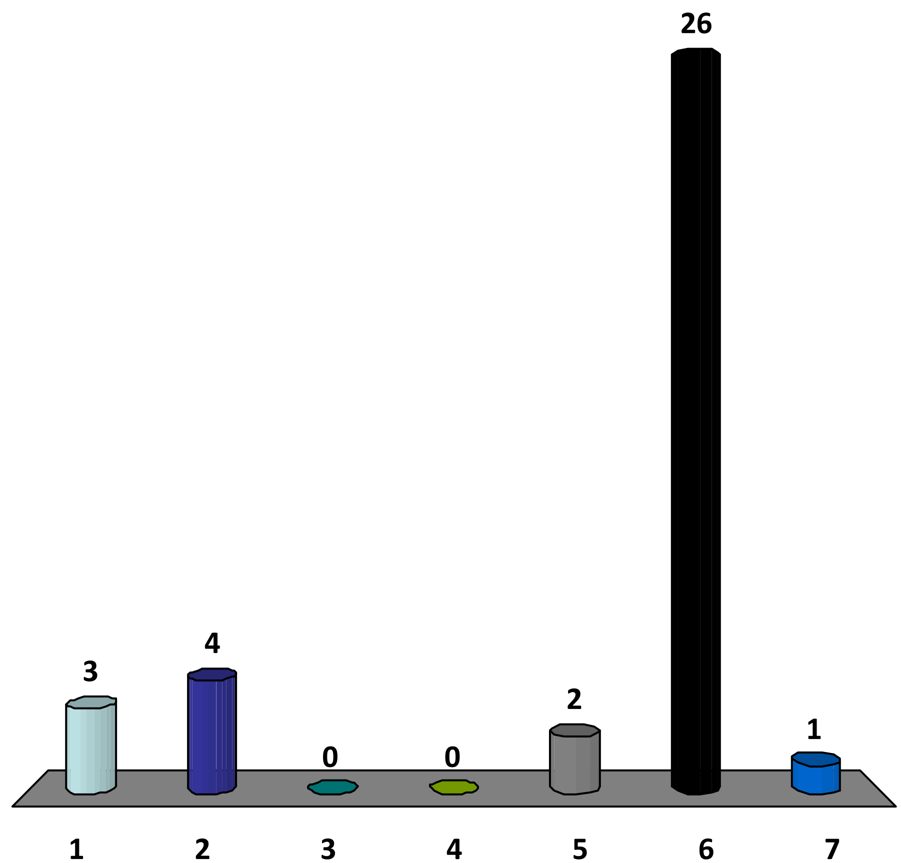
1. insert(key, data)
2. delete(key)
3. successor(key)
4. search(key)
5. None of the above.



Review: Dictionary Abstract Data Type

Which of the following cannot be easily used to implement a dictionary?

1. Array
2. Binary Search Tree
3. Direct Access Table
4. Hash Table
5. Linked List
6. Stack
7. None of the above.



Review

Dictionary Abstract Data Type

- insert(key, data)
- search(key)
- delete(key)

Typical Implementations:

- Array
- Linked List
- (Binary) Search Tree
- Hash Table

Review

Applications of Dictionaries:

- Pilot scheduling
- Document distance
- DNA Analysis (longest common substrong)

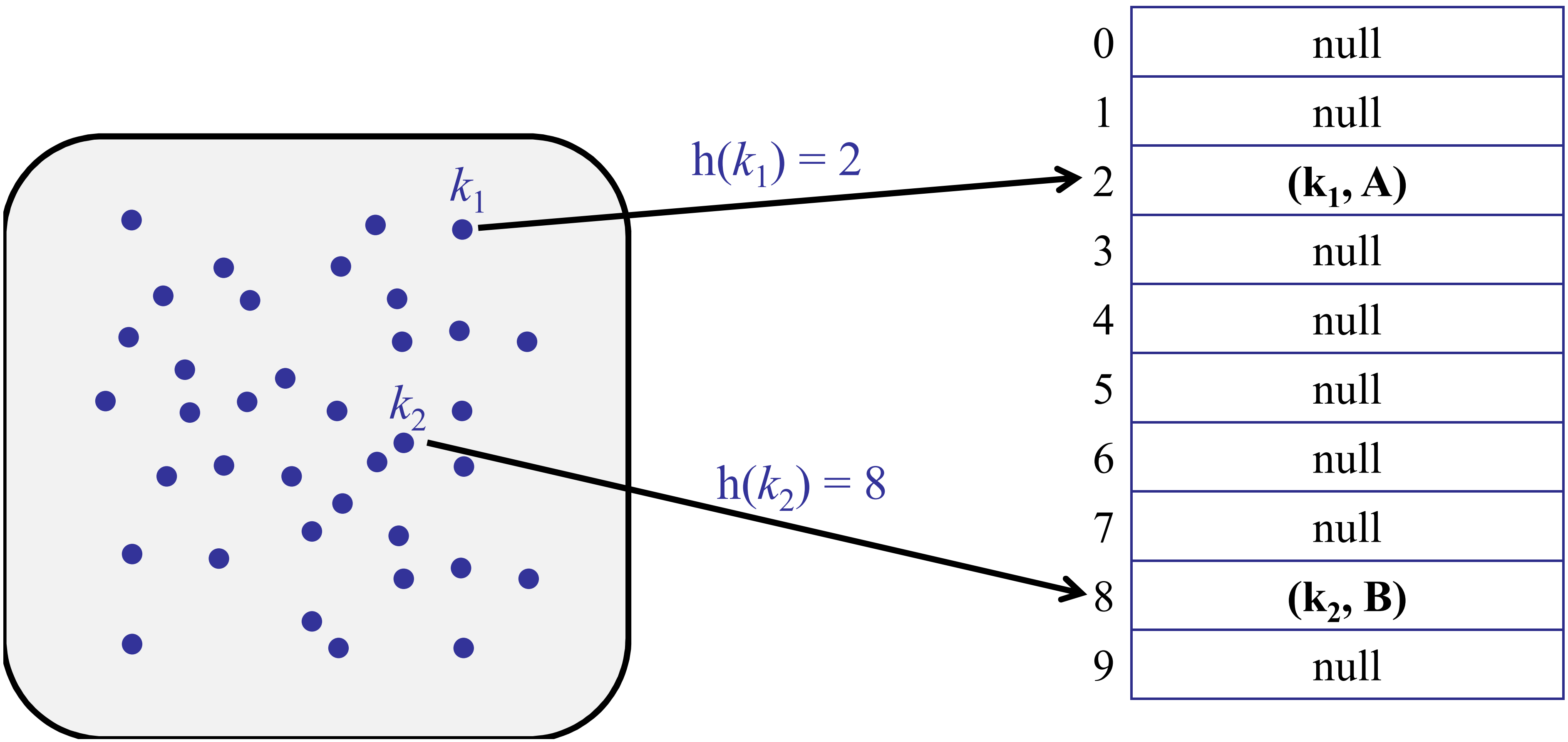
Dictionaries in Java:

- HashMap<keyType, dataType>

Review

Hash Tables

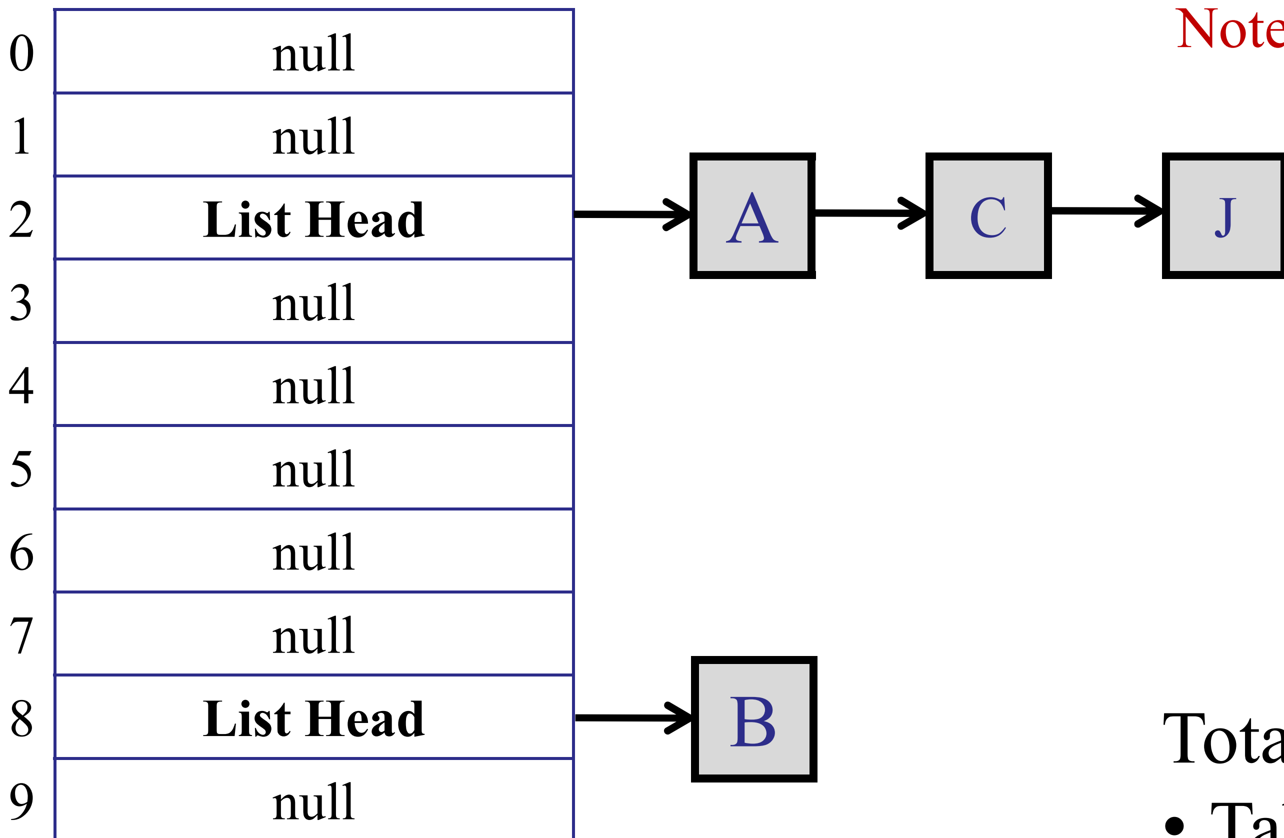
- Store each item from the dictionary in a table.
- Use hash function to map each key to a bucket.



Review: 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

Review

The Simple Uniform Hashing Assumption

- Every key is equally likely to map to every bucket.

Load of a Hash Table:

- # elements: n
- # buckets: m
- Define: load(hash table) = n/m

  = average #items / bucket.
- Expected search time = $1 + n/m$

Note: error on slides from last lecture!

Review

Division method:

- Choose table of size p, for some prime p.
- $h(k) = k \bmod p$

Multiplication method:

- Choose odd integer A.
- Let w be the word size.
- Let 2^r be the table size.

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

Table Size

How large should the table be?

- Assume: Simple Uniform Hashing
- Expected search time: $O(1 + n/m)$
- Optimal size: $m = \Theta(n)$

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

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

Table Size

Idea:

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

Example:

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

Table Size

How to grow the table:

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

Table Size

Time complexity of growing the table:

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

Table Size

Time complexity of growing the table:

- Assume:
  - Size $m_1 = \Theta(n)$.
  - Size $m_2 = \Theta(n)$.
- Costs:
  - Total: $O(m_1 + n(1+n/m_2))$.

    $= O(n)$

How fast to grow?

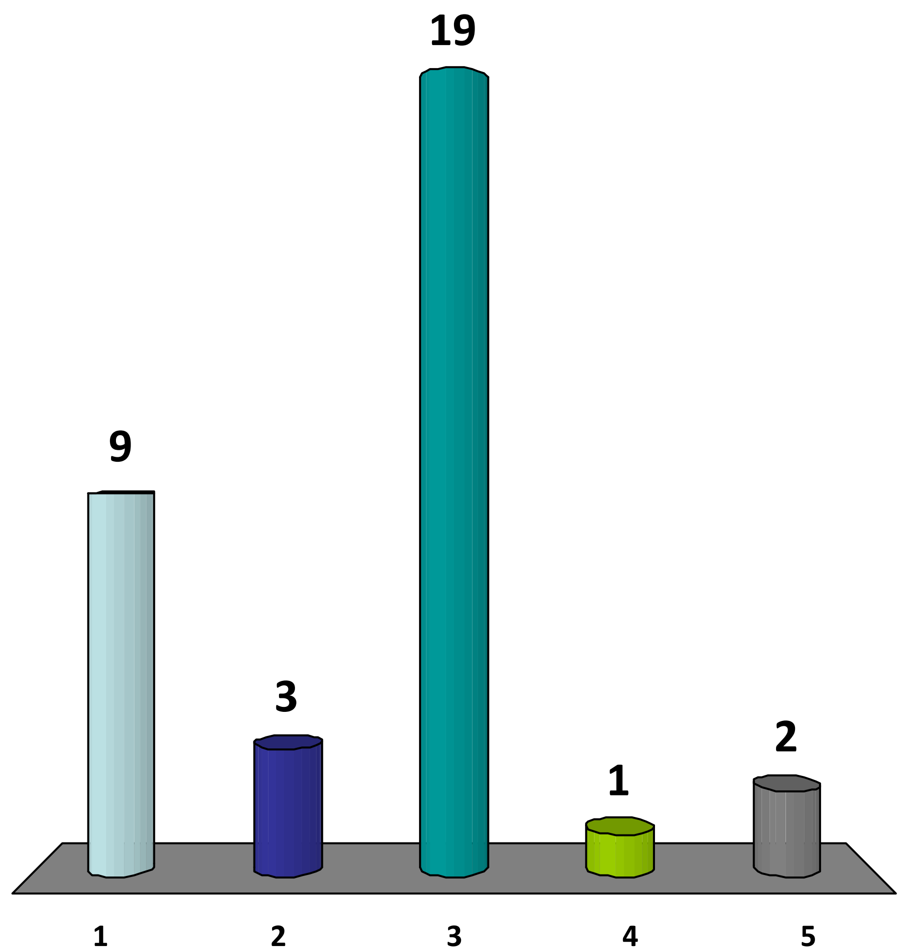
Idea 1: Increment table size by 1

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

- 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		7c	8c	9c	10c	11c		cn

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

How fast to grow?

Idea 2: Double table size

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

- 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		7c	8c	1	1	1	1	1	1	1	16c	1	1		cn

- Total cost: $c(8 + 16 + 32 + \ldots + n) = O(n)$

How fast to grow

Idea 2: Double table size

Cost of Resizing:

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

How fast to grow?

Idea 2: Double table size

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

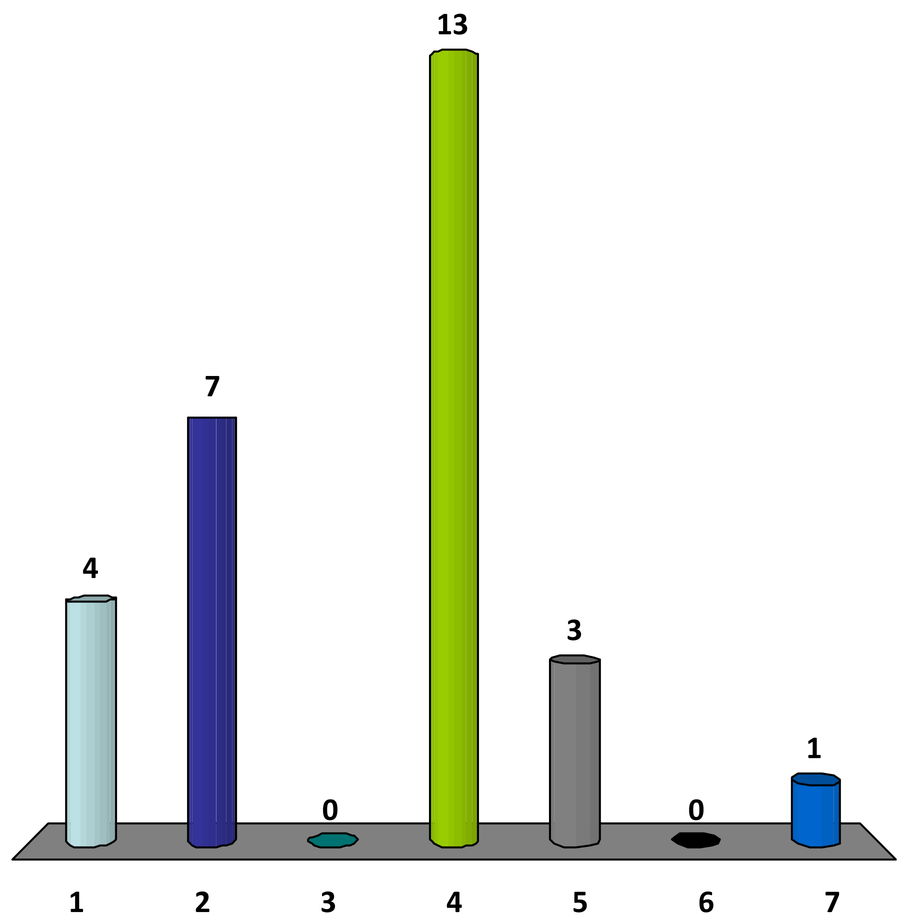
- When ($n == m$): $m = m^2$

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 / log n)
4. O(n)
5. O(n log n)
6. O(n^2)
7. None of the above.



How fast to grow?

Idea 3: Square table size

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

- Cost of resize:
  - Size $m_1 = n$.
  - Size $m_2 = n^2$.
  - Total: $O(m_1 + n(1+n/m_2))$

    $= O(n + n(1+1/n))$

    $= O(n)$

How fast to grow?

Idea 3: Square table size

– When ($n == m$): $m = m^2$

# Items	Total Resizing Cost
8	8c
64	(8 + 64)c
4,096	(8 + 64 + 4,096)c
…	…
n	> $c\sqrt{n}$
	< O(n)

How fast to grow?

Idea 3: Square table size

– When ($n == m$): $m = m^2$

# Items	Resizing Cost	Insert Cost
8	8c	8c
64	(8 + 64)c	64c
4,096	(8 + 64 + 4,096)c	4,096c
…	…	…
n	$> c\sqrt{n}$	cn
	$< O(n)$	O(n)

How fast to grow?

Idea 3: Square table size

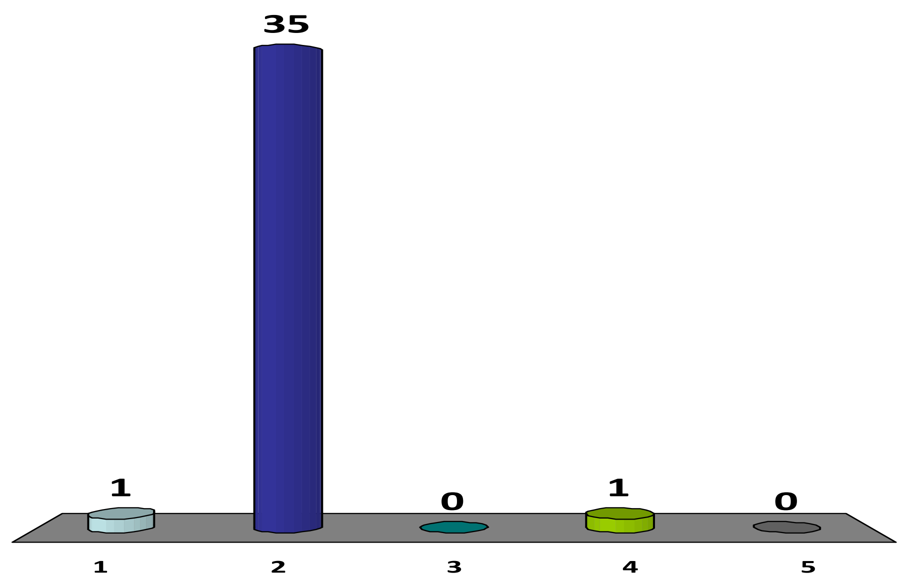
- When ($n == m$): $m = m^2$

- Cost of resize:
  - Total: O(n)

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

Why not square the table size?

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

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

Amortized Analysis

Definition:

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

Example: (Hash Tables)

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

– Conclusion:

The <u>insert operation</u> has amortized cost $O(1)$.

Amortized Analysis

Definition:

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

Example: (Problem Set 5)

- Inserting n elements into a weight-balanced search tree costs $O(\log n)$.
- Conclusion:

  The insert operation has amortized cost $O(\log n)$.

Amortized Analysis

Accounting Method (paying rent)

- Each operation adds money to the system.
- Every step of the algorithm either:
  - Costs money.
  - Costs time.
- Total cost execution = time + money
  - Average time / operation = initial money + time cost

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.
- A table with k new elements since last resize has $\Theta(k)$ dollars.

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!
- Analyze inserts ignoring cost of resizing.

Total cost: Inserting k elements costs:

- Dollars: \$O(k) (used to pay for resizing)
- Time: O(k) for inserting elements into table
- Total (Time + Money): O(k)

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!
- Analyze inserts ignoring cost of resizing.

Average cost:

- Dollars: \$O(*1*) (initial dollars per operation)
- Time: O(*1*) for inserting element into table
- Total (Time + Money): 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()

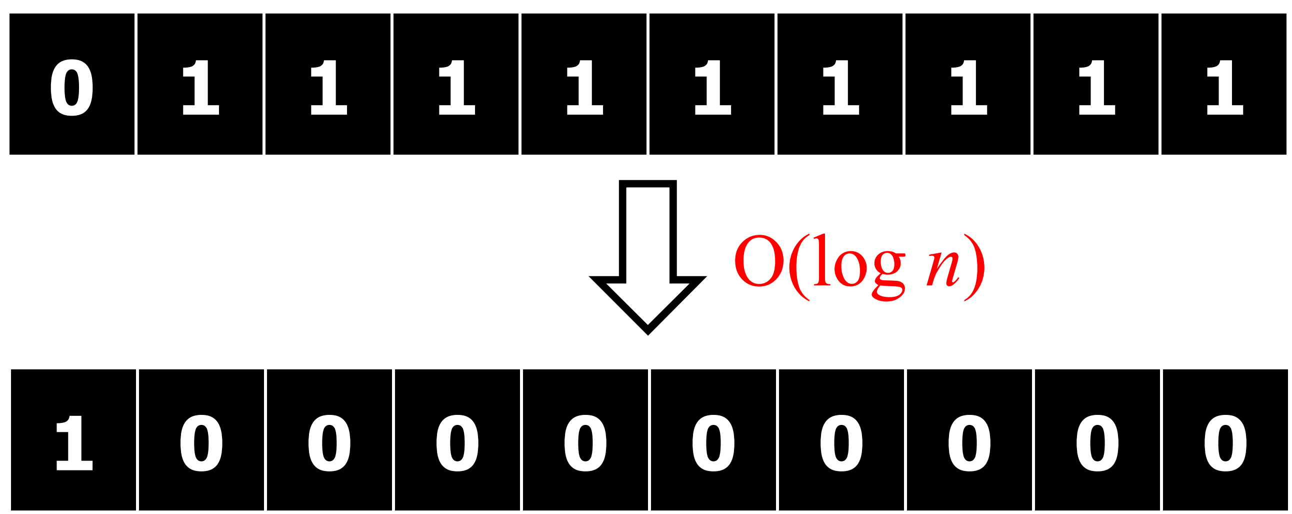
0	0	0	0	0	0	0	0	1	1

Example: Binary Counter

Counter ADT:

- increment()
- read()

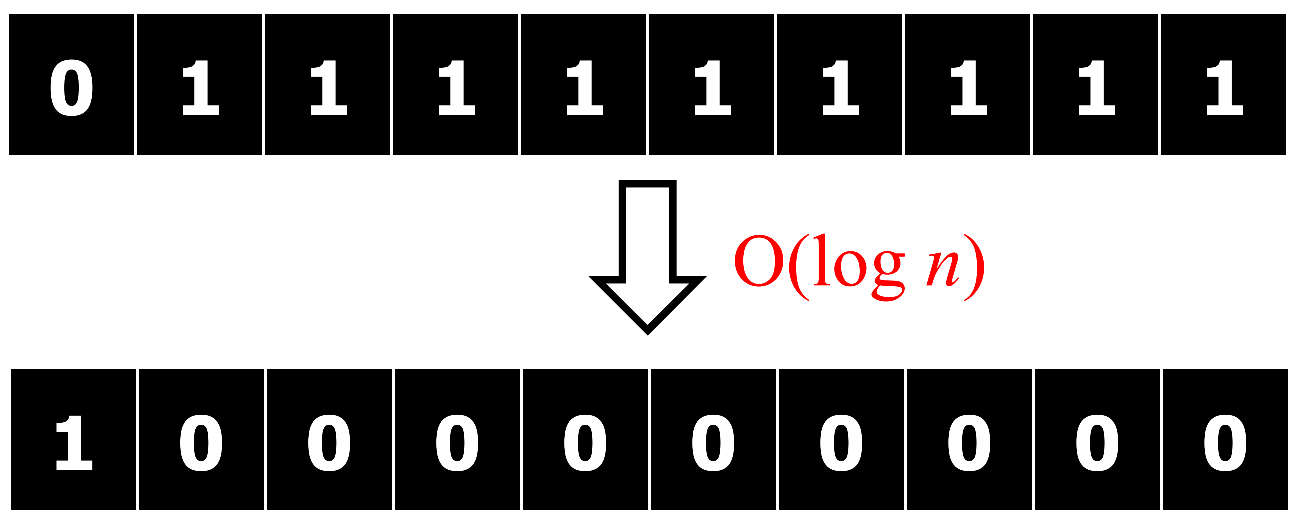
Some increments are expensive…



Example: Binary Counter

Question: If we increment the counter to n, what is the average 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

0	1	1	1	1	1	1	1	1	1

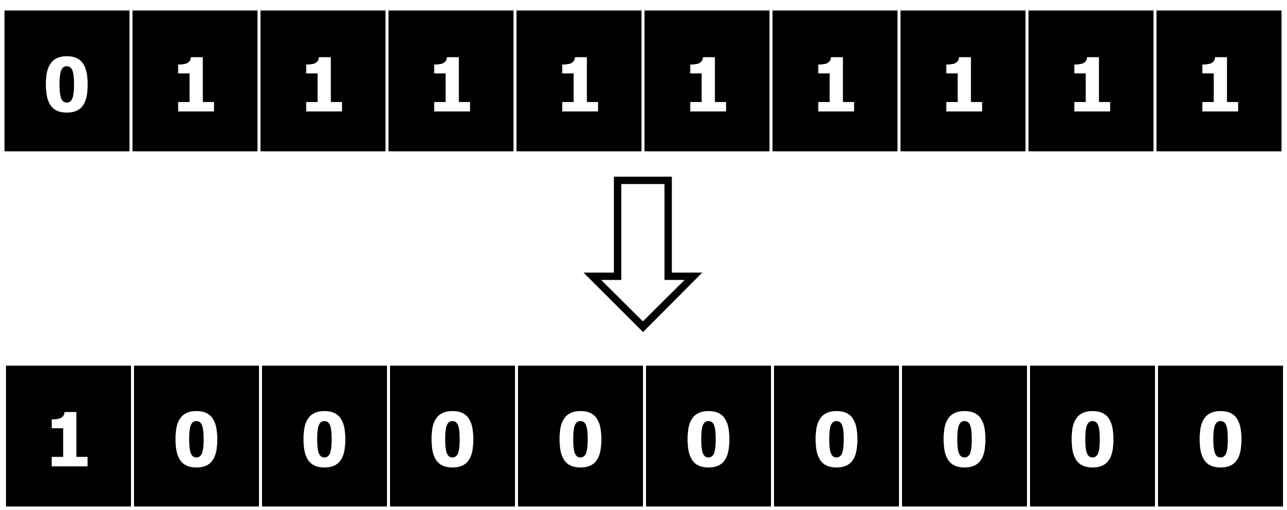
$$\Downarrow$$

1	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Observation:

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

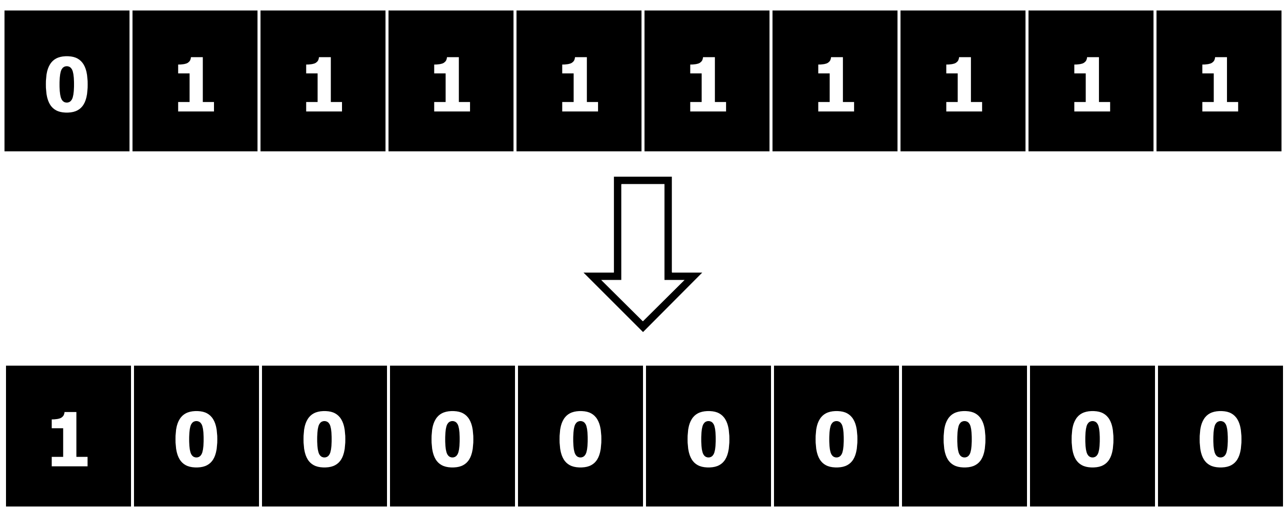


Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

Whenever you change it from $0 \rightarrow 1$, add one dollar.



Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

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

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

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:

Average 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

DNA Analysis

Find the longest common substring of two DNA sequences.

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.

DNA Analysis

Long common substring (text):

ALGORITHM vs. ARITHMETIC

Assume both strings are the same length…

Computer Scientist's View of Biology

"I have some exciting new results to tell you about today. To start out, let's assume:

- A gene is an integer.
- A chromosome is a set of integers.
- An organism is a set of sets.

Now that we are all on the same page…"

DNA Analysis

Naïve Algorithm: strings *A* and *B*

```
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;
```

DNA Analysis

Naïve Algorithm: strings A and B

```
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;
```

Loop n times.

n substrings

n substrings

comparison costs: $O(n)$

Total cost: $O(n^4)$

DNA Analysis

Improvements:

1. Binary search: $O(n^3 \log n)$
   - Given L, is there a common substring of length L?
   - Search for largest L where the answer is yes.

DNA Analysis

Improvements:

2. Hash table + binary search: $O(n^2 \log n)$
   - Put all n strings of length L from string A in a hash table.
   - Search for all n strings of length L from string B.
   - Report success if found.

DNA Analysis

Long common substring (text):

ALGORITHM vs. ARITHMETIC

Consider (L == 3):

- Add to dictionary:
  - ALG, LGO, GOR, ORI, RIT, ITH, THM
- Search in dictionary:
  - ARI, RIT, ITH, THM, HME, MET, ETI, TIC
- Matches:
  - RIT, ITH, THM

Longest Common Substring

exists-substring(X1, X2, L)

1. for (i = 0 to n – L – 1) do:
2. $hash$ = h(X1[i : i + L])
3. T.hash-insert($hash$, i))
4. for (i = 0 to n – L – 1) do:
5. $hash$ = h(X2[i : i + L])
6. if (T.hash-lookup($hash$, s)) then
7. return true.
8. return false

Longest Common Substring

exists-substring(X1, X2, L)

1. for (i = 0 to n – L – 1) do: Loop $n – L$ times.
2. $hash$ = h(X1[i : i + L]) Calculate hash: O(L).
3. T.hash-insert($hash$, i)) Insert: O(1)
4. …

Assume:

– Simple uniform hashing

– m >= n

Total cost: O(L(n – L)) = O(n^2)

Longest Common Substring

exists-substring(X1, X2, L)

1. …
2. for ($i = 0$ to $n - L - 1$) do:
3. $\quad\quad$ *hash* = h(X2[i : i + L])
4. $\quad\quad$ if (T.hash-lookup(*hash* , s)) then
5. $\quad\quad\quad\quad$ return true.

Loop $n - L$ times.

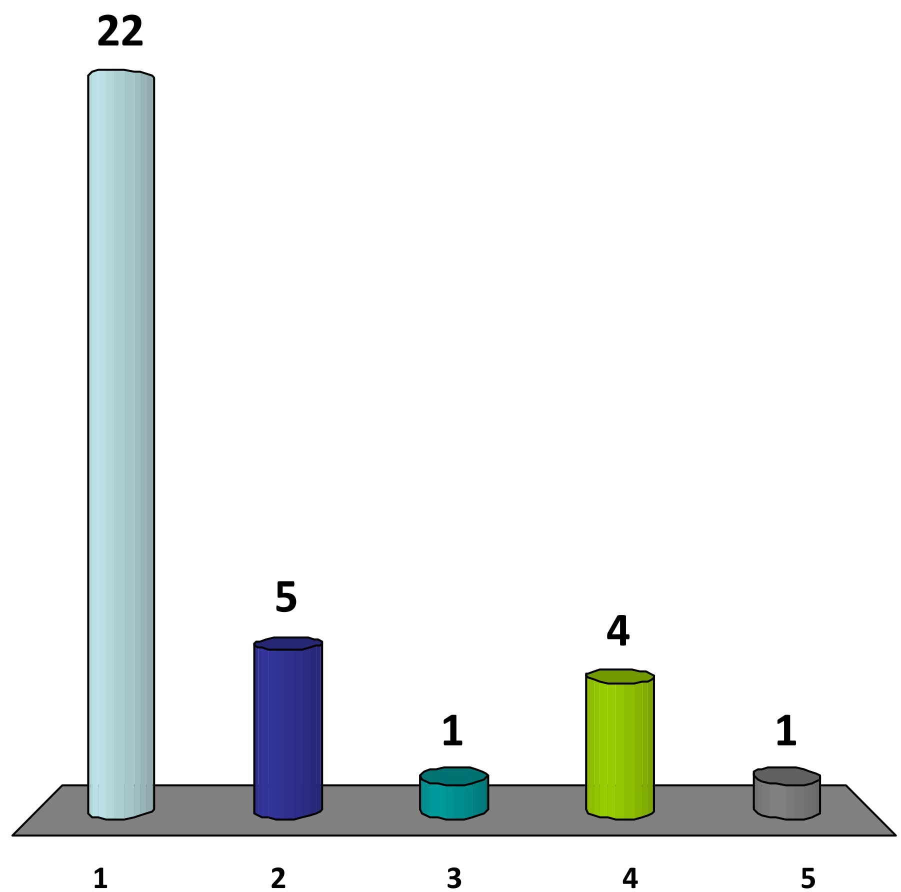
Calculate hash: O(L).

Lookup: ???

What is the cost of hash-lookup??

Assume simple-uniform hashing and m>=n. The (expected) cost of hash-lookup is:

1. O(1)
2. O(L)
3. O(log n)
4. O(L log n)
5. O(n)



Longest Common Substring

hash-lookup(*hash*, *s*)

Case 1: string *s* is in the table at position *hash*.

- Cost: O(L)
  - O(1) lookup.
  - O(L) comparison (*s* == T[*hash*])
- How many times can this happen?
  - Once!
  - If we find a string of length L, then we return true.

0	null
1	null
hash	**(hash, s)**
3	null
4	null
5	null
6	null
7	null
8	**(k_2, B)**
9	null

Longest Common Substring

hash-lookup(*hash*, *s*)

Case 2: No element is in the table at position *hash*.

- Cost: O(1)
  - O(1) lookup.
  - O(1) comparison (*null* == T[*hash*])

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

Longest Common Substring

hash-lookup(*hash*, *s*)

Case 3: string *s* is **NOT** in the table at position *hash*.

- Cost: O(L)
  - O(1) lookup.
  - O(L) comparison (*s* != T[*hash*])

- How often does this happen?
  - Under simple uniform hashing, there is a collision with probability >= n/m

0	null
1	null
hash	**(hash, t)**
3	null
4	null
5	null
6	null
7	null
8	**(k_2, B)**
9	null

Longest Common Substring

hash-lookup(*hash*, *s*)

Case 3: string *s* is **NOT** in the table at position *hash*.

- How often does this happen?
  - Under simple uniform hashing, there is a collision with probability $\geq n/m$
  - Assume $(n > m/4)$ then: $\geq 1/4$
  - For each hash-lookup: E[cost] $\geq L / 4$

0	null
1	null
hash	**(hash, t)**
3	null
4	null
5	null
6	null
7	null
8	**(k_2, B)**
9	null

Longest Common Substring

exists-substring(X1, X2, L)

1. …
2. for ($i = 0$ to $n - L - 1$) do:
3. $\quad\quad hash = h(X2[i : i + L])$
4. $\quad\quad$ if (T.hash-lookup($hash$, s)) then
5. $\quad\quad\quad\quad$ return true.

Loop $n - L$ times.

Calculate hash: O(L).

Lookup: E[cost] >= L/4

Total cost: $O((n - L)(L + L/4)) = O(n^2)$

DNA Analysis

In order to speed up `exists-substring`:

1. Reduce false positives
   - If the hash is in the table, then it is very likely that the string is in the hash table.

2. Compute hash faster
   - It is too slow to re-compute the hash function (n – L) times.

Faster substring matching

Reduce false positives:

- Idea 1: Make the hash table bigger.
  - Problem: probability(collision) $= n/m >= 1/4$
  - Solution: set $m = n^2$

- Analysis:
  - Probability(collision) $= n/m <= n/n^2 <= 1/n$
  - Expected cost of a false positive $= L/n$.
  - Expected cost of a lookup $= 1 + L/n$

Longest Common Substring

exists-substring(X1, X2, L)

1. …
2. for ($i = 0$ to $n - L - 1$) do:
3. $\quad$ *hash* = h(X2[i : i + L])
4. $\quad$ if (T.hash-lookup(*hash* , s)) then
5. $\quad\quad$ return true.

Loop $n - L$ times.

Calculate hash: O(L).

Lookup: E[cost] = 1 + L/n

Total cost: $O((n - L)(L + 1 + L/n)) = O(n^2)$

Faster substring matching

Reduce false positives:

- Idea 1: Make the hash table bigger.
  - Problem: probability(collision) $= n/m >= 1/4$
  - Solution: set $m = n^2$
- Problem:
  - Table is too big!!

Faster substring matching

Reduce false positives:

- Idea 2: Use two different hash functions.
  - $h_1 : U \rightarrow \{1..m\}$, $m < 4n$.
  - $h_2 : U \rightarrow \{1..n^2\}$.

- Store pair: $(h_2(s), s)$ in hash table at location $h_1(s)$.
  - Call $h_2(s)$ the *signature*.

Faster substring matching

Reduce false positives:

- Idea 2: Use two different hash functions.
  - $h_1 : U \rightarrow \{1..m\}$, $m < 4n$.
  - $h_2 : U \rightarrow \{1..n^2\}$.

hash-lookup(s):

if (Table[$h_1(s)$] != null) then

(sig, t) = Table[$h_1(s)$]

if ($h_2(s) == sig$) then

if ($s == t$) then return true;

Faster substring matching

Analysis:

- Size of signature.
  - $h_2 : U \rightarrow \{1..n^2\}$.
  - log(n2) = 2log(n)
- Assume that we can read/write/compare $\log(n)$ bits in time $O(1)$.
  - Why? A machine word is > $\log(n)$.
- Cost of comparing two signatures = $O(1)$.

Faster substring matching

Analysis:

- By simple uniform hashing:
- if (s != t) then:

  probability(h(s) == h(t)) <= $1/n$
- Expected(cost of hash-lookup) <= $1 + L/n$.

---

hash-lookup(s):

if (Table[$h_1(s)$] != null) then

(*sig*, t) = Table[$h_1(s)$]

if ($h_2(s)$ == *sig*) then

if (s == t) then return true;

Longest Common Substring

exists-substring(X1, X2, L)

1. …
2. for ($i = 0$ to $n - L - 1$) do:
3. $\quad hash = h(X2[i : i + L])$
4. $\quad$ if (T.hash-lookup($hash$, s)) then
5. $\qquad$ return true.

Loop $n - L$ times.

Calculate hash: O(L).

Lookup: E[cost] = 1 + L/n

Total cost: $O((n - L)(L + 1 + L/n)) = O(n^2)$

DNA Analysis

In order to speed up `exists-substring`:

1. Reduce false positives
   - Use second hash function as a signature.
   - Reduce cost of collisions.

2. Compute hash faster
   - It is too slow to re-compute the hash function (n – L) times.

Rolling Hash Function

Abstract data type:

- insert(s) : sets string equal to string s
- delete-first-letter()
- append-letter(c)
- hash() : returns hash of current string

Rolling Hash Function

Example:

- insert(“arith”)

  string == “arith”

- hash() → 17
- delete-first-letter()

  string == “rith”

- hash() → 47
- append-letter(‘m’)

  string == “rithm”

- hash() → 4

Rolling Hash Function

Costs:

- insert(s) : O(|S|)
- delete-first-letter() : O(1)
- append-letter(c) : O(1)
- hash() : O(1)

Example:

- insert("arith") : 5c
- delete-first-letter(), append-letter(m) : O(1) = c

  string == "rithm"
- delete-first-letter(), append-letter(e) : O(1) = c

  string == "ithme"
- delete-first-letter(), append-letter(t) : O(1) = c

  string == "thmet"
- delete-first-letter(), append-letter(i) : O(1) = c

  string == "hmeti"
- delete-first-letter(), append-letter(c) : O(1) = c

  string == "metic"

Conclusion: n – L = 6 hashes for cost 10c = O(n).

Longest Common Substring

exists-substring(X1, X2, L)

1. *rollhash*.insert(X1[i : i + L])
2. for (i = 0 to n – L – 1) do:
3. T.hash-insert(*rollhash*.hash(), i))
4. rollhash.delete-first-letter()
5. rollhash.append-letter(X1[i + L])
6. …

Longest Common Substring

exists-substring(X1, X2, L)

1. *rollhash*.insert(X1[i : i + L])
2. for (i = 0 to n – L – 1) do:
3. T.hash-insert(*rollhash*.hash(), i))
4. rollhash.delete-first-letter()
5. rollhash.append-letter(X1[i + L])
6. …

Loop $n - L$ times.

Insert: O(1)

Update hash: O(1).

Total cost: $O(n - L + L) = O(n)$

Longest Common Substring

exists-substring(X1, X2, L)

1. …
2. *rollhash*.insert(X2[i : i + L])
3. for (i = 0 to n – L – 1) do:
4. if (T.hash-lookup(*rollhash*.hash() , s)) then
5. return true.
6. rollhash.delete-first-letter()
7. rollhash.append-letter(X1[i + L])

Longest Common Substring

exists-substring(X1, X2, L)

1. …
2. *rollhash*.insert(X2[i : i + L])
3. for (i = 0 to n – L – 1) do:
4. if (T.hash-lookup(*rollhash*.hash() , s)) then
5. return true.
6. rollhash.delete-first-letter()
7. rollhash.append-letter(X1[i + L])

Loop $n - L$ times.

Lookup: E[cost] = 1 + L/n

Update hash: O(1).

Total cost: $O((n - L)(1 + L/n) + L) = O(n)$

Rolling Hash Function

Abstract data type:

- insert(s) : sets string equal to string s
- delete-first-letter()
- append-letter(c)
- hash() : returns hash of current string

Rolling Hash

Basic idea:

- Initially (on “insert”), calculate hash of string.
- Whenever the string is updated, update the hash.
- When a hash() is requested, output the pre-computed hash.

Rolling Hash

Step 1: Represent a string as a number

- Assume all letters in a string are 8-bit chars.
- Given a sequence of letters:

  $c_{L-1}\, c_{L-2} \ldots c_1\, c_0$

- Define: 8L bit integer

$$s = \underbrace{00101001}_{c_{L-1}}\ \underbrace{10110111}_{c_{L-2}}\ \ldots\ \underbrace{10010000}_{c_1}\ \underbrace{10010000}_{c_0}$$

Rolling Hash

Step 1: Represent a string as a number

- Assume all letters in a string are 8-bit chars.
- Given a sequence of letters:

  $c_{L-1}\ c_{L-2} \ldots c_1\ c_0$

- Define: 8L bit integer

$$s = \underbrace{00101001}_{c_{L-1}}\ \underbrace{10110111}_{c_{L-2}} \ldots \underbrace{10010000}_{c_1}\ \underbrace{10010000}_{c_0}$$

$$s = \sum_{i=0}^{L-1} c_i \cdot 2^{8i}$$

Rolling Hash

Step 1: Represent a string as a number

- Assume all letters in a string are 8-bit chars.
- Given a sequence of letters:

$c_{L-1}\ c_{L-2} \ldots c_1\ c_0$

- Define: 8L bit integer

$$s = \underbrace{00101001}_{c_{L-1}}\ \underbrace{10110111}_{c_{L-2}} \ldots \underbrace{10010000}_{c_1}\ \underbrace{10010000}_{c_0}$$

$$s = \sum_{i=0}^{L-1} c_i \cdot 2^{8i} = \sum_{i=0}^{L-1} c_i \ll 8i$$

Rolling Hash

Step 2: Updating the string

Deleting character c_{L-1}:

$$
\begin{array}{rl}
s = & 00101001\ \ 10110111\ \ \ldots\ 10010000\ \ 10010000 \\
- & 00101001\ \ 00000000\ \ \ldots\ 00000000\ \ 00000000 \\
\hline
 & \ \ 10110111\ \ \ldots\ 10010000\ \ 10010000
\end{array}
$$

Rolling Hash

Step 2: Updating the string

Deleting character c_{L-1}:

$$
\begin{array}{rllll}
s = & 00101001 & 10110111 & \ldots\ 10010000 & 10010000 \\
- & 00101001 & 00000000 & \ldots\ 00000000 & 00000000 \\
\hline
 & & 10110111 & \ldots\ 10010000 & 10010000
\end{array}
$$

$$
\begin{aligned}
s &= s - c_{L-1} \cdot 2^{8(L-1)} \\
&= s - c_{L-1} \ll 8(L-1)
\end{aligned}
$$

Multiplication: O(1)

Shift: O(1)

Subtraction: O(1)

Rolling Hash

Step 2: Updating the string

Appending character c:

$$
\begin{array}{rr}
s = 00000000\ \ 10110111\ \ \ldots\ 10010000\ \ 10010000 & \\
* \qquad\qquad\qquad\qquad\qquad\qquad\qquad 1\ \ 00000000 & \\
\hline
10110111\ \ \ldots\ 10010000\ \ 10010000\ \ 00000000 &
\end{array}
$$

Rolling Hash

Step 2: Updating the string

Appending character c:

$$
\begin{array}{rr}
s = & 00000000\ \ 10110111\ \ \ldots\ 10010000\ \ 10010000 \\
* & 1\ \ 00000000 \\
\hline
 & 10110111\ \ \ldots\ 10010000\ \ 10010000\ \ 00000000 \\
+ & 10101101 \\
\hline
 & 10110111\ \ \ldots\ 10010000\ \ 10010000\ \ 10101101
\end{array}
$$

Rolling Hash

Step 2: Updating the string

Appending character c:

$$
\begin{array}{rr}
s = 00000000\ \ 10110111\ \ \ldots\ 10010000\ \ 10010000 & \\
* \qquad\qquad\qquad\qquad\qquad\qquad\qquad 1\ \ 00000000 & \\
\hline
10110111\ \ \ldots\ 10010000\ \ 10010000\ \ 00000000 & \\
+ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 10101101 & \\
\hline
10110111\ \ \ldots\ 10010000\ \ 10010000\ \ 10101101 &
\end{array}
$$

$$s = s * 2^8 + c$$

$$= (s \ll 8) + c$$

Shift, addition: O(1)

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = s \bmod p$$

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = s \bmod p$$

Appending a character:

$$\mathrm{h}(\mathrm{s} \ll 8 + \mathrm{c})$$

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = s \bmod p$$

Appending a character: O(1)

$$\begin{aligned} & \mathrm{h}(\mathrm{s} \ll 8 + \mathrm{c}) \\ & = [(\mathrm{s} \ll 8) + \mathrm{c}] \bmod \mathrm{p} \\ & = [(\mathrm{s} \bmod p) \ll 8) \bmod p + \mathrm{c}] \bmod \mathrm{p} \\ & = [\mathrm{h}(\mathrm{s}) \ll 8 + \mathrm{c}] \bmod \mathrm{p} \end{aligned}$$

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = s \bmod p$$

Deleting the first character:

$$h\left(s - \left(c_{L-1} \ll 8(L-1)\right)\right)$$

Rolling Hash

Step 3: The Hash Function

The Division Method

$$h(s) = s \bmod p$$

Deleting the first character: O(1)

$$h\left(s - \left(c_{L-1} \ll 8(L-1)\right)\right)$$

$$= [h(s) - (c_{L-1} \ll 8(L-1) \bmod p)] \bmod p$$

Rolling Hash Function

Costs:

- insert(s) : O(|S|)
- delete-first-letter() : O(1)
- append-letter(c) : O(1)
- hash() : O(1)

DNA Analysis

Longest Common Substring

For any length L:

`exists-substring`(X1, X2, L)

has cost $O(n)$.

Using binary search to find maximum value of L, we find the longest common substring in time:

$O(n \log n)$

DNA Analysis

Longest Common Substring

For any length L:

`exists-substring`(X1, X2, L)

has cost $O(n)$.

Using binary search to find maximum value of L, we find the longest common substring in time:

$O(n \log n)$

The story continues… suffix-trees… $O(n)$….

Summary

Amortized Analysis

- Sometimes, it is better to look at the average cost per operation.

Using Hash Tables

- To get efficient algorithms, you have to be careful!
- Signatures…
- Rolling hashes…
- Etc.