CS2020
Data Structures and Algorithms

Welcome!

Coding Quiz

Coding Under Pressure

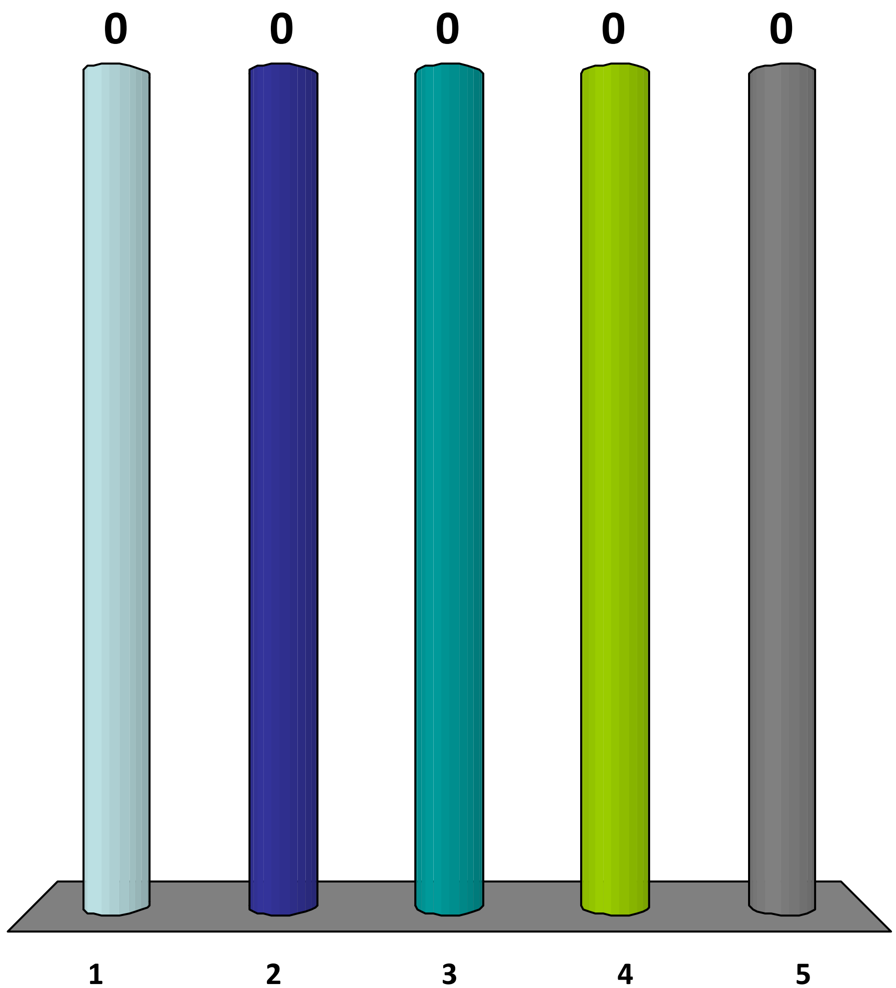
- Hard!
- Stressful!

Goal:

- Questions were not, individually, hard.
- Lots of coding required.
- Lots of places for small mistakes

The Coding Quiz was:

1. Very hard
2. Hard
3. Ok
4. Easy
5. Very easy.



0 of 5

Today

- DNA Analysis
  - Finish the analysis of the Longest-Common-Substring.
- Resolving Collisions
  - Open Addressing
- Advanced Hashing
  - Universal Hashing
  - Perfect Hashing

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.

Longest Common Substring

exists-substring(X1, X2, L)

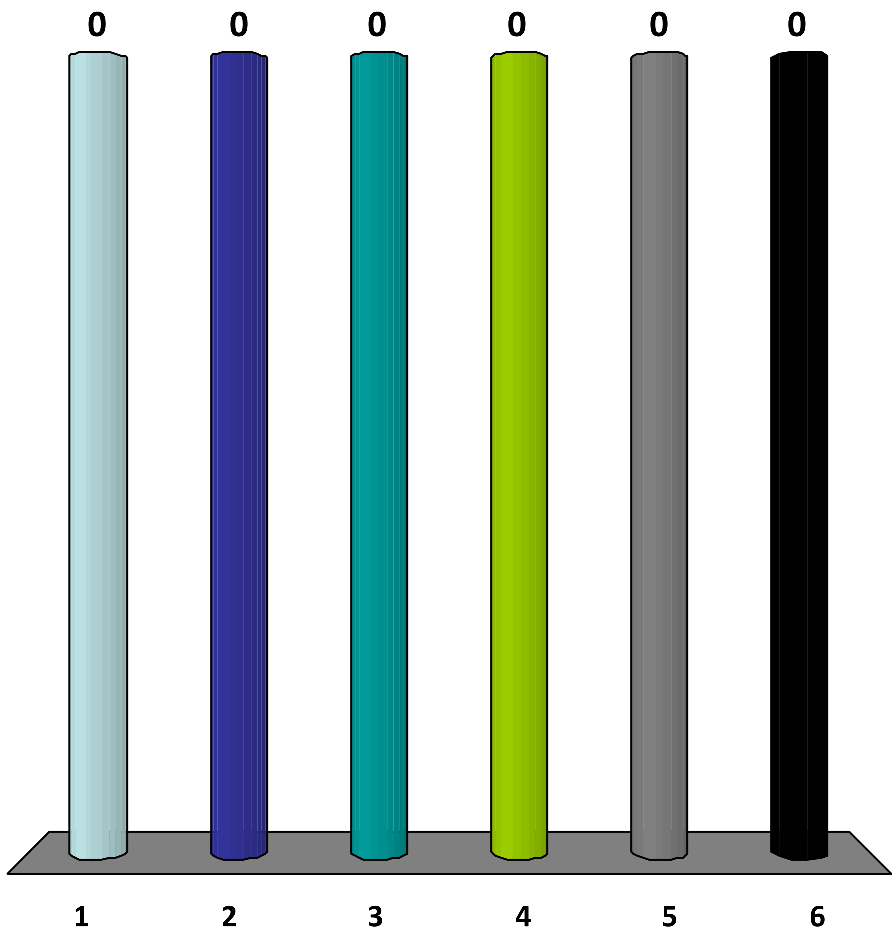
1. for (i = 0 to n – L – 1) do:
2.   *hash* = h(X1[i : i + L])
3.   T.hash-insert(*hash, i*))
4. for (i = 0 to n – L – 1) do:
5.   *hash* = h(X2[i : i + L])
6.   if (T.hash-lookup(*hash* , s)) then
7.   return true.
8. return false

The performance of

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

on strings of length n is:

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



0 of 5

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))
4. …

Insert: O(1)

Assume:

– Simple uniform hashing

– m >= n

Total cost: O(L(n – L)) = 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:

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

- Using a hash function as a *signature.*
  - A hash of a large data structure gives a small signature.
  - Example:
    - Are two databases identical?
    - Compare hash!
  - Think of a hash as a fingerprint.

Faster substring matching

Reduce false positives:

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

hash-insert(s):

Table[$h_1(s)$].LLinsert($h_2(s)$, s)

Faster substring matching

Reduce false positives:

- 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: hash-lookup(s)

- Case 1: string s is in table: O(L)
- Case 2: Table[$h_1(s)$] = null: O(1)
- Case 3: Table[$h_1(s)$] != null: ??

hash-lookup(s):

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

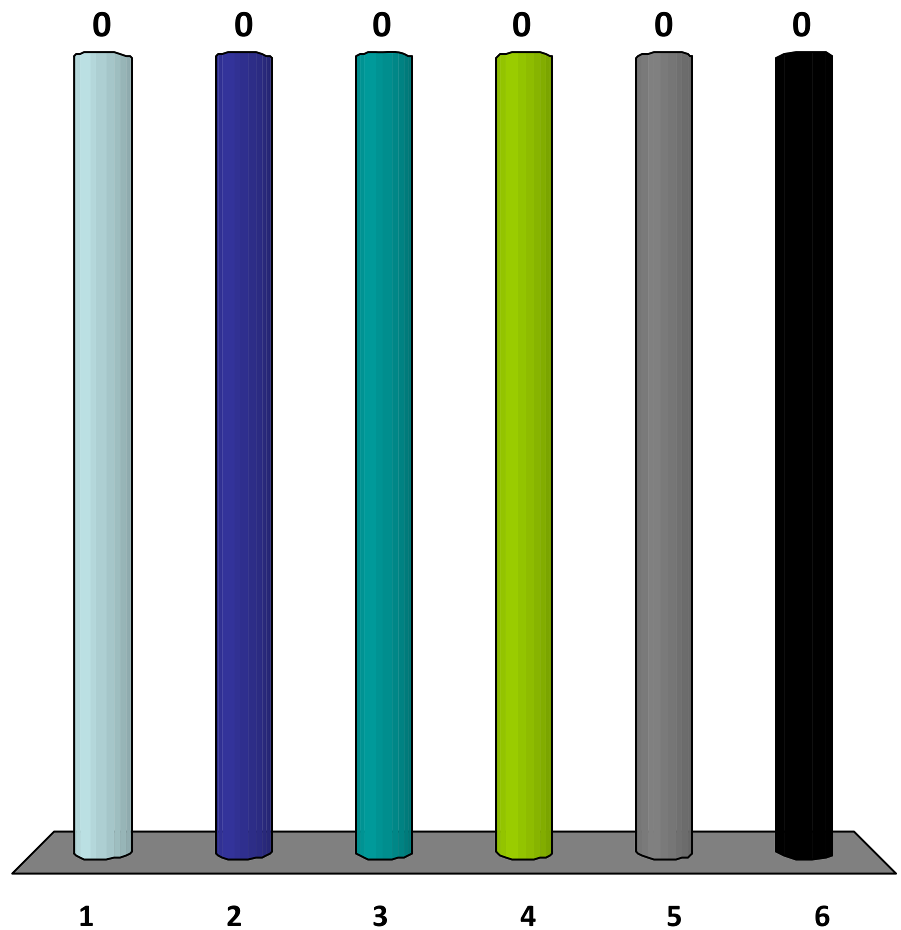
  (sig, t) = Table[$h_1(s)$]

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

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

Let $h_2 : U \rightarrow \{1..n^2\}$ be a hash function. For strings s and t, what is the probability that $h_2(s) == h_2(t)$?

1. $1/n$
2. $2/n$
3. $1/n^2$
4. $1/\sqrt{n}$
5. $1/2$
6. None of the above.



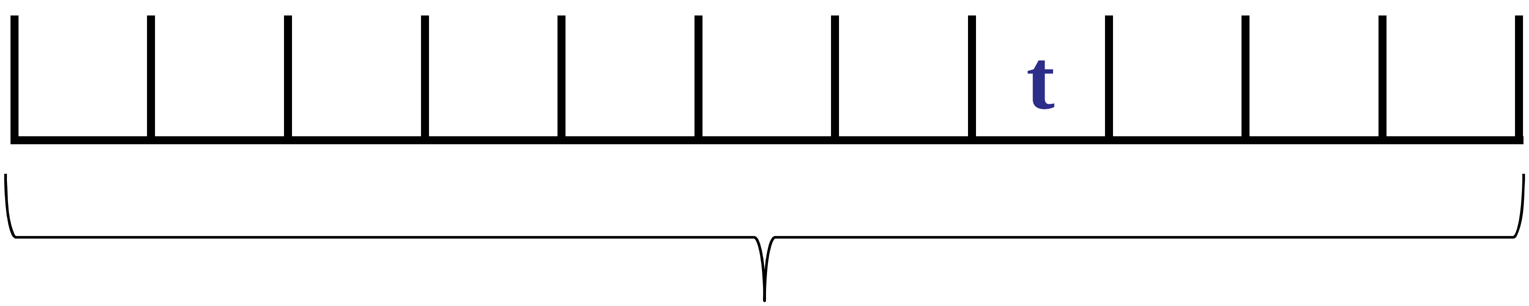
0 of 5

Faster substring matching

Analysis: hash-lookup(s) (Assume SUHA.)

- $h_2 : U \rightarrow \{1..n^2\}$
- For two strings s and t:

  Probability($h_2(s) == h_2(t)$): $1/n^2$



n^2

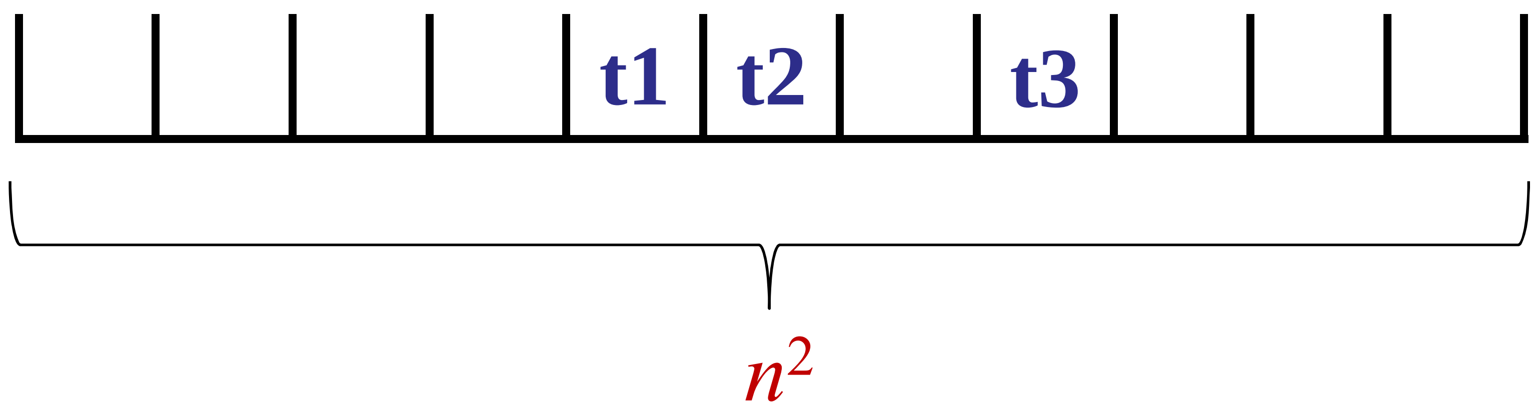
Faster substring matching

Analysis: hash-lookup(s) (Assume SUHA.)

- $h_2 : U \rightarrow \{1..n^2\}$

- For string s:

  Probability($h_2(s) == h_2(t)$ for any string t): $n/n^2 \leq 1/n$



Faster substring matching

Analysis: hash-lookup(s)

- Case 1: string s is in table: O(L)
- Case 2: Table[$h_1(s)$] = null: O(1)
- Case 3: Table[$h_1(s)$] != null: O(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;

with probability ≤ 1/n

Cost: O(L).

Faster substring matching

Analysis:

- Size of signature.
  - $h_2 : U \rightarrow \{1..n^2\}$.
  - $\log(n^2) = 2\log(n)$
- 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 = $O(1)$.

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

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:

Loop n – L times.

Insert: O(1)

3. T.hash-insert(*rollhash*.hash(), i))

4. rollhash.delete-first-letter()

5. rollhash.append-letter(X1[i + L])

Update hash: O(1).

6. …

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}:

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

Rolling Hash

Step 2: Updating the string

Deleting character c_{L-1}:

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

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

Multiplication: O(1)

Shift: O(1)

Subtraction: O(1)

Rolling Hash

Step 2: Updating the string

Appending character c:

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

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:

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

$$
\begin{aligned}
s &= s * 2^8 + c \\
&= (s \ll 8) + c
\end{aligned}
$$

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… $O(n)$….

DNA Analysis Summary

Using Hash Tables

- To get efficient algorithms, you have to be careful!
- Signatures…
  - Hash functions are useful as a “summary” of a longer / bigger document.
- Rolling hashes…
  - Fast way to calculate hashes in an incremental fashion.

Today

- DNA Analysis
  - Finish the analysis of the Longest-Common-Substring.
- Resolving Collisions
  - Open Addressing
- Advanced Hashing
  - Universal Hashing
  - Perfect Hashing

Resolving Collisions

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

Open Addressing

Advantages:

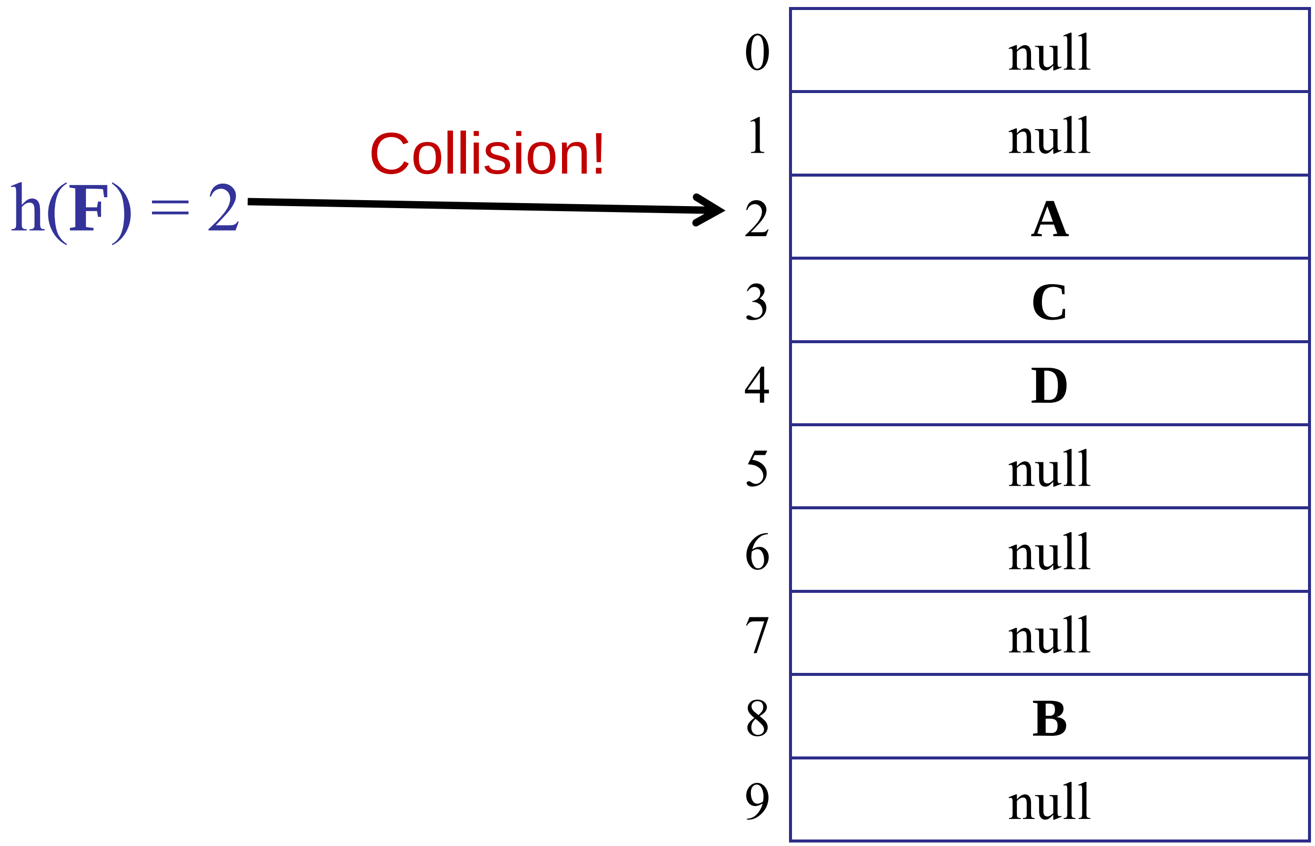
- No linked lists!
- All data directly stored in the table.
- One item per slot.

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

Open Addressing

On collision:

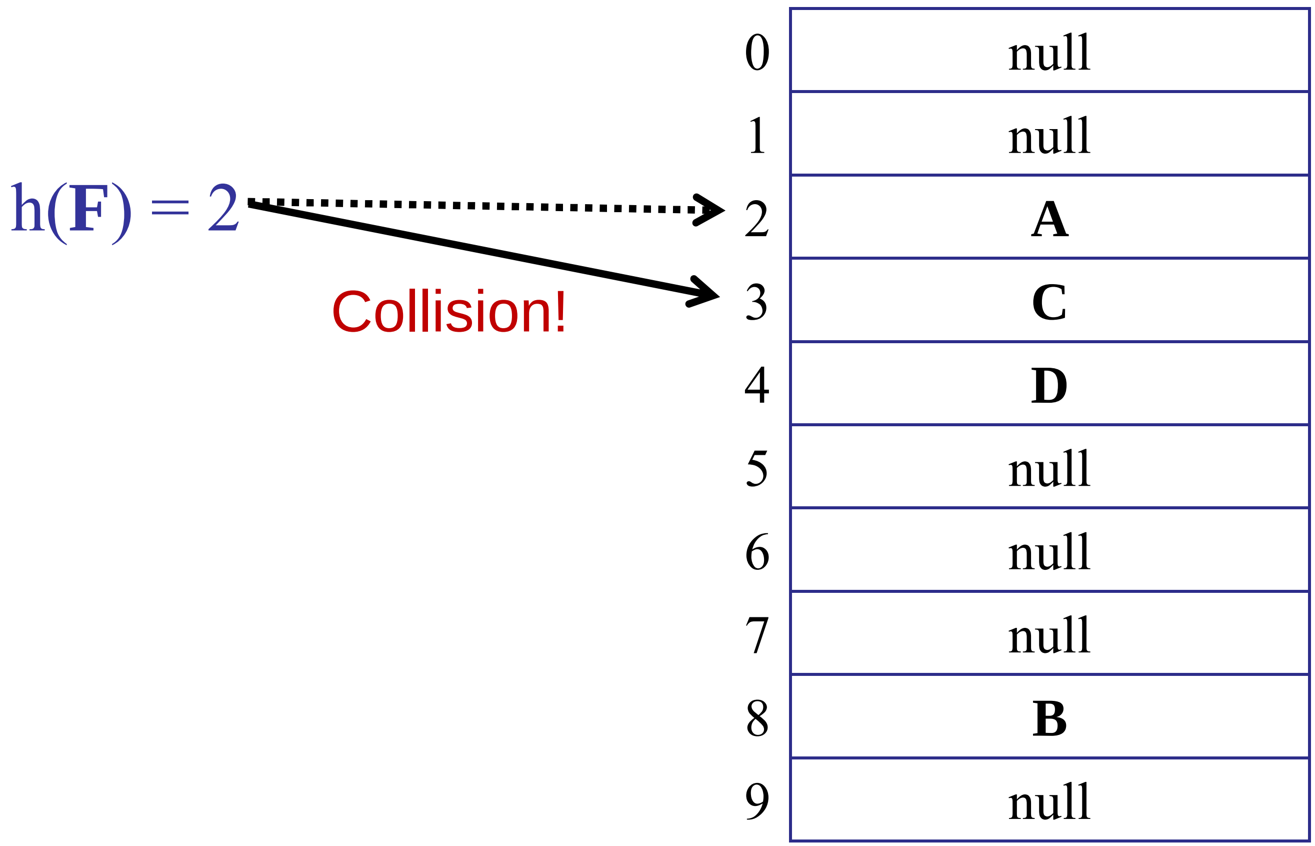
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

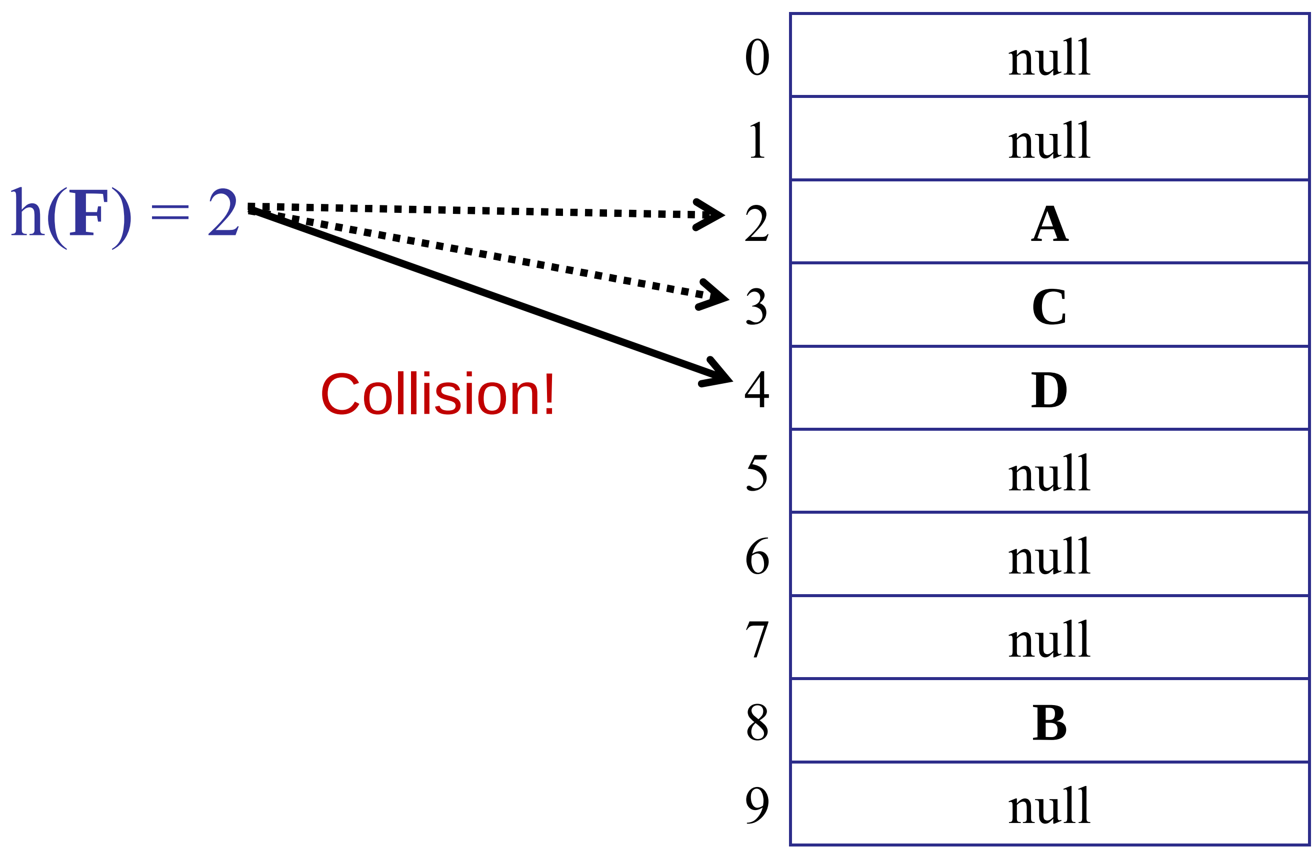
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

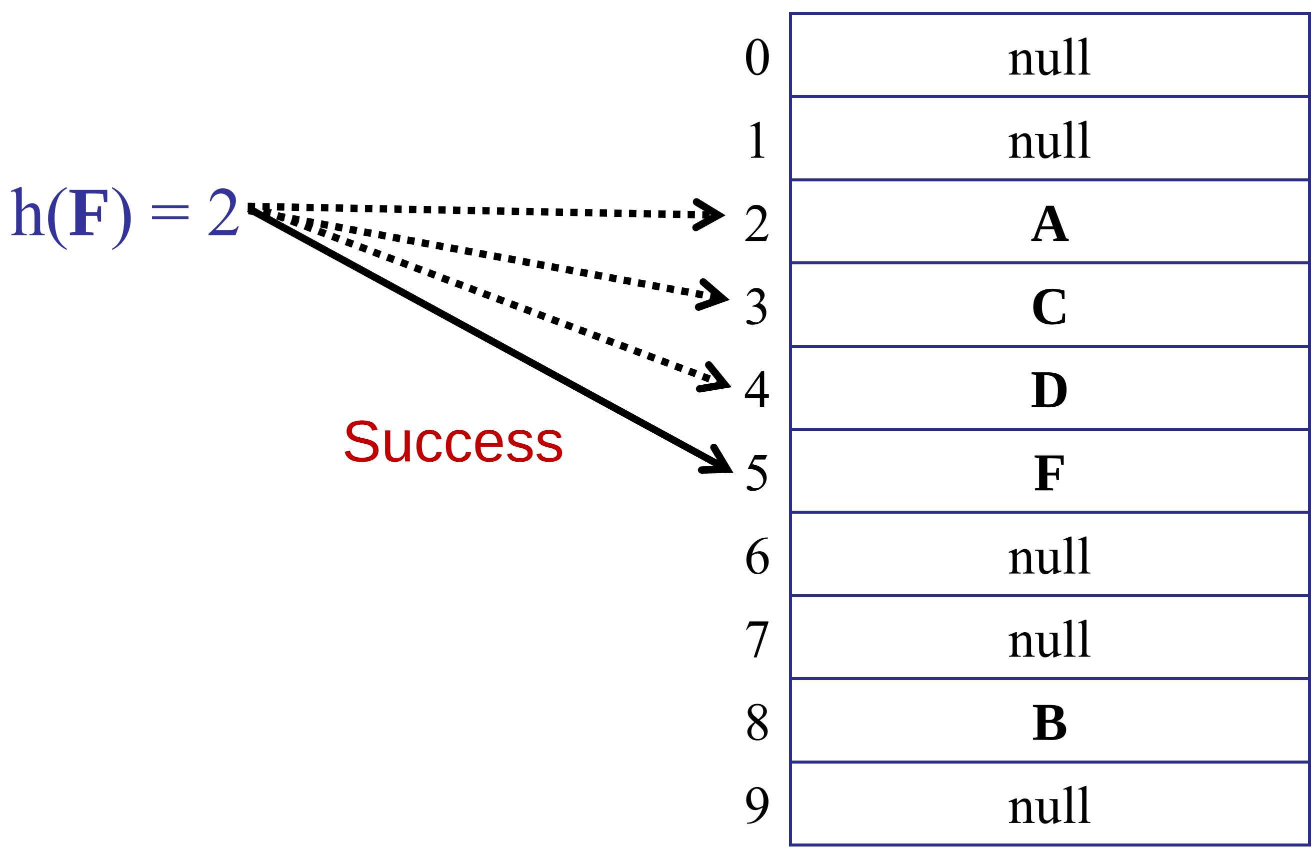
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

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



Open Addressing

On collision:

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

h(**F**) = 2

Success

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

Linear Probing:

- h(k)+1, h(k)+2, h(k)+3 …

Open Addressing

Hash Function re-defined:

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

Two parameters:

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

Open Addressing

Hash Function re-defined:

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

Example: Linear Probing

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

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

Open Addressing

Hash Function re-defined:

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

Example: Weird Probing

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

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

Open Addressing

```
hash-insert(key, data)
int i = 1;
while (i <= m) {                              // Try every bucket
      int bucket = h(key, i);
      if (T[bucket] == null){    // Found an empty bucket
            T[bucket] = {key, data}; // Insert key/data
            return success;           // Return
      }
      i++;
}
return table-full;                           // Table full!
```

Open Addressing

Hash Function re-defined:

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

search(key)

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

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

Open Addressing

```
hash-search(key)
int i = 1;
while (i <= m) {
    int bucket = h(key, i);
    if (T[bucket] == null)  // Empty bucket!
        return key-not-found;
    if (T[bucket].key == key)  // Full bucket.
            return T[bucket].data;
    i++;
}
return key-not-found;  // Exhausted entire table.
```

Open Addressing

Hash Function re-defined:

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

search(key)

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

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

Open Addressing

Hash Function re-defined:

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

delete(key)

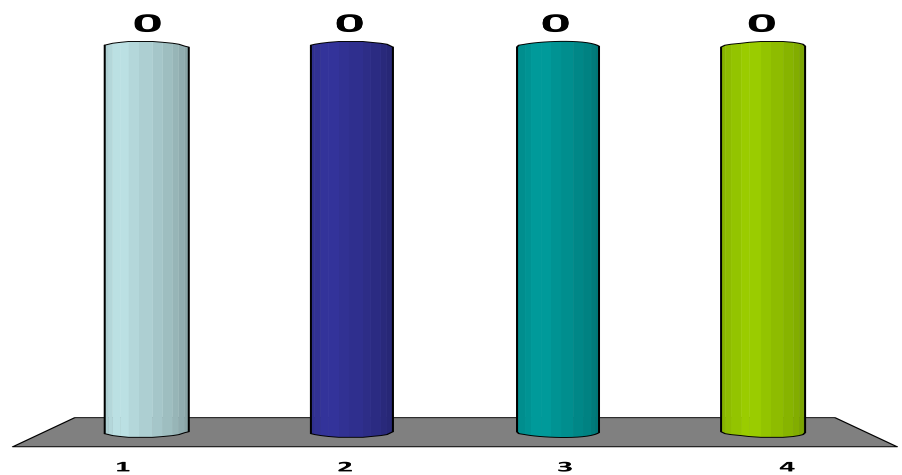
- Find key to delete
- Remove it from table.
- Set bucket to null.

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

What is wrong with delete?

1. Search may fail to find an element.
2. The table will have gaps in it.
3. Space is used inefficiently.
4. If the key is inserted again, it may end up in a different bucket.

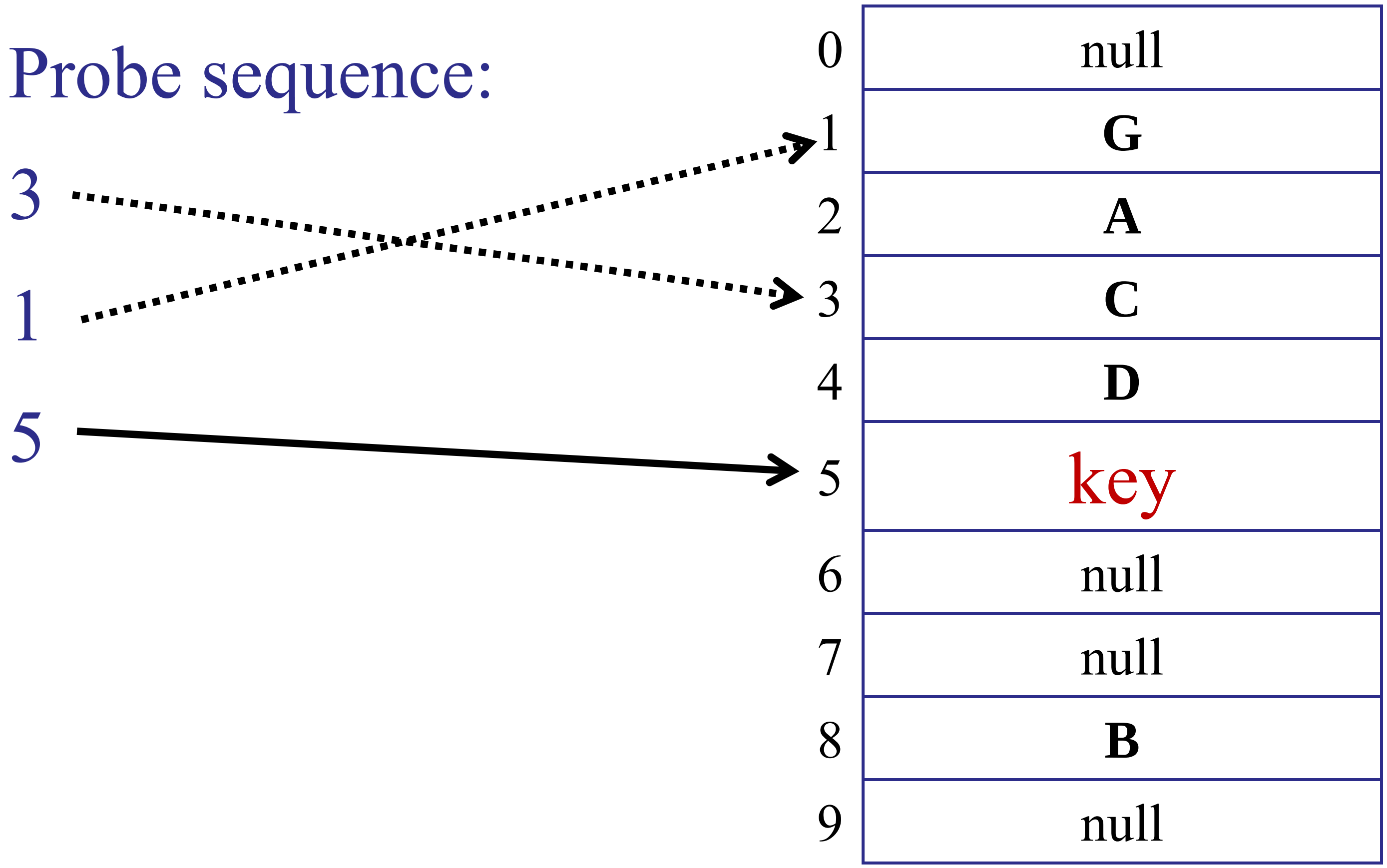
0 of 60



Open Addressing

insert(key)

Probe sequence:



Open Addressing

insert(key)

delete(G)

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

Open Addressing

insert(key)

delete(G)

search(key)

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

Open Addressing

insert(key)

delete(G)

search(key)

Probe sequence:

3

1

5

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

Open Addressing

insert(key)

delete(G)

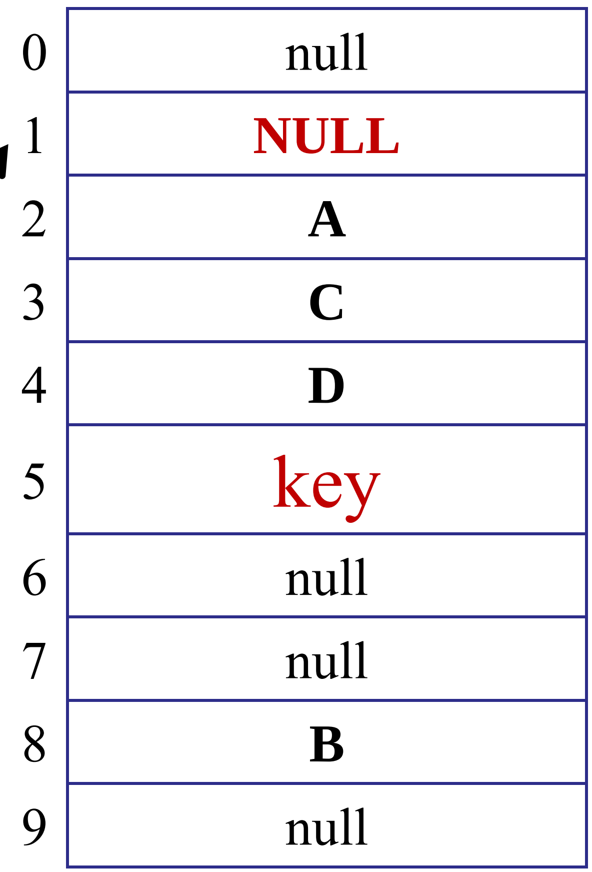
search(key)

Probe sequence:

3

1

Not found!



Open Addressing

Hash Function re-defined:

h(key, i) : U → {1..m}

delete(key)

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

(Tombstone value.)

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

Open Addressing

insert(key)

delete(G)

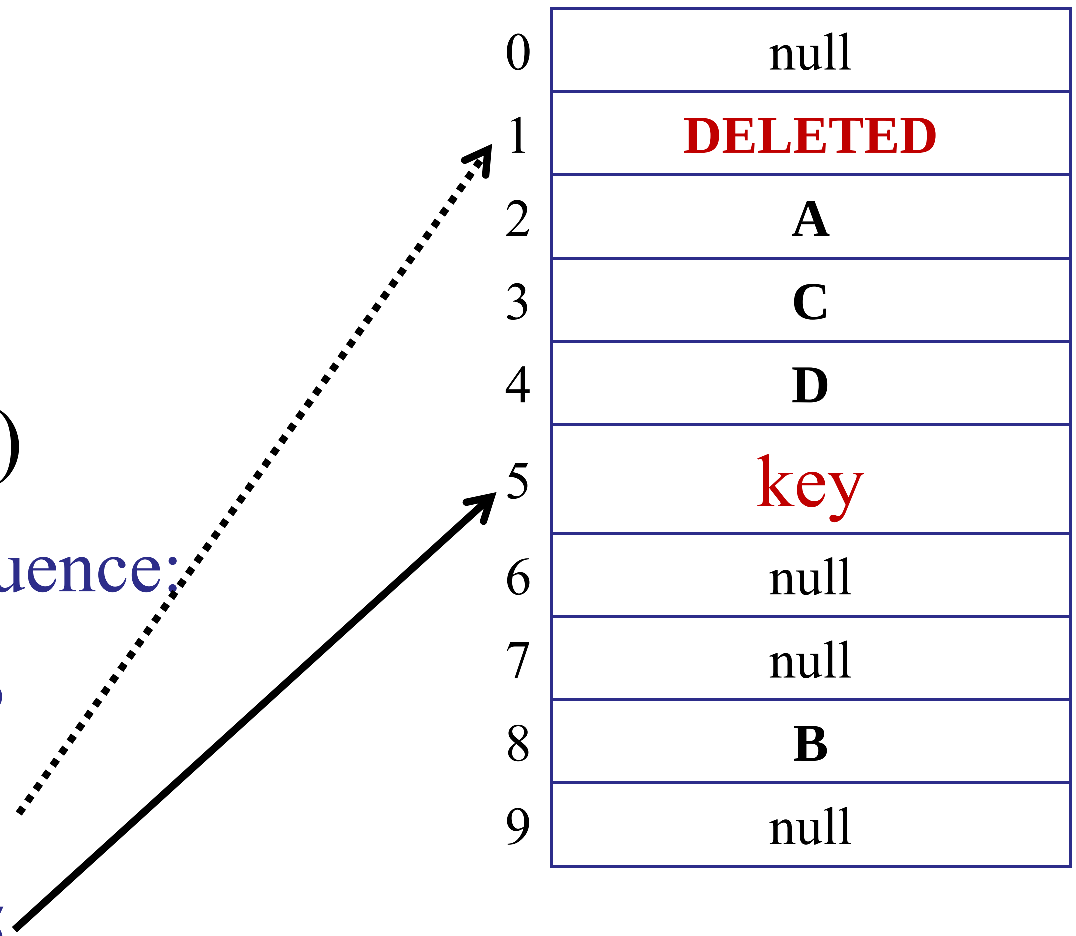
search(key)

Probe sequence:

3

1

5



Open Addressing

Properties of a good hash function:

1. h(*key*, *i*) enumerates all possible buckets.

   For every bucket j, there is some i such that:

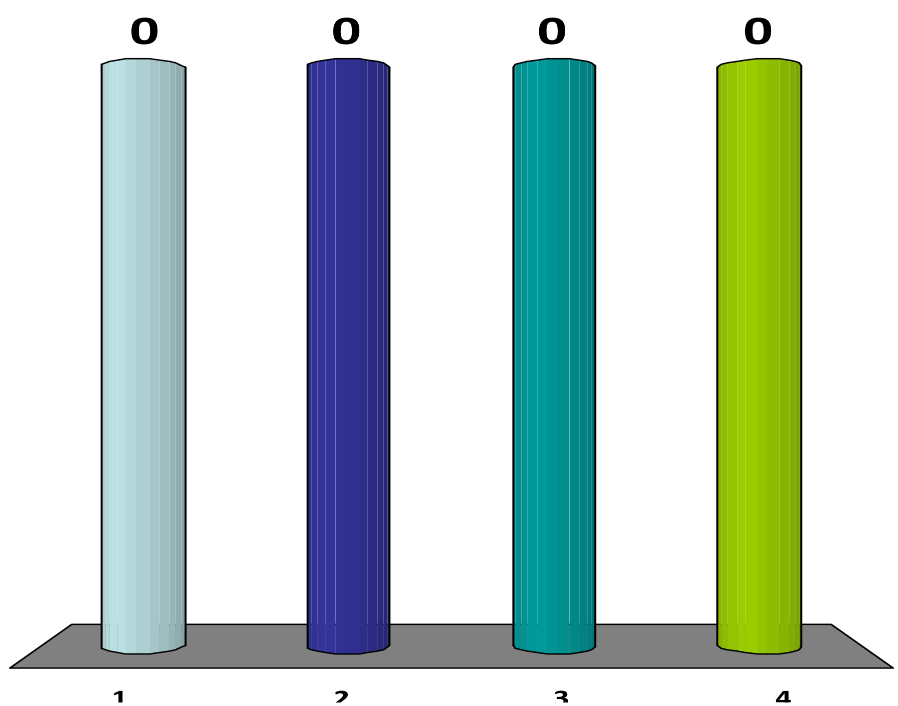
   $$h(key, i) = j$$

   The hash function is permutation of $\{1..m\}$.

   For linear probing: true!

Enter question text...

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



0 of 60

Open Addressing

Properties of a good hash function:

2. Simple Uniform Hashing Assumption

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

For h(*key*, 1)?

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

Open Addressing

Properties of a good hash function:

2. Uniform Hashing Assumption

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

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

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

Open Addressing

Properties of a good hash function:

2. Uniform Hashing Assumption

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

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

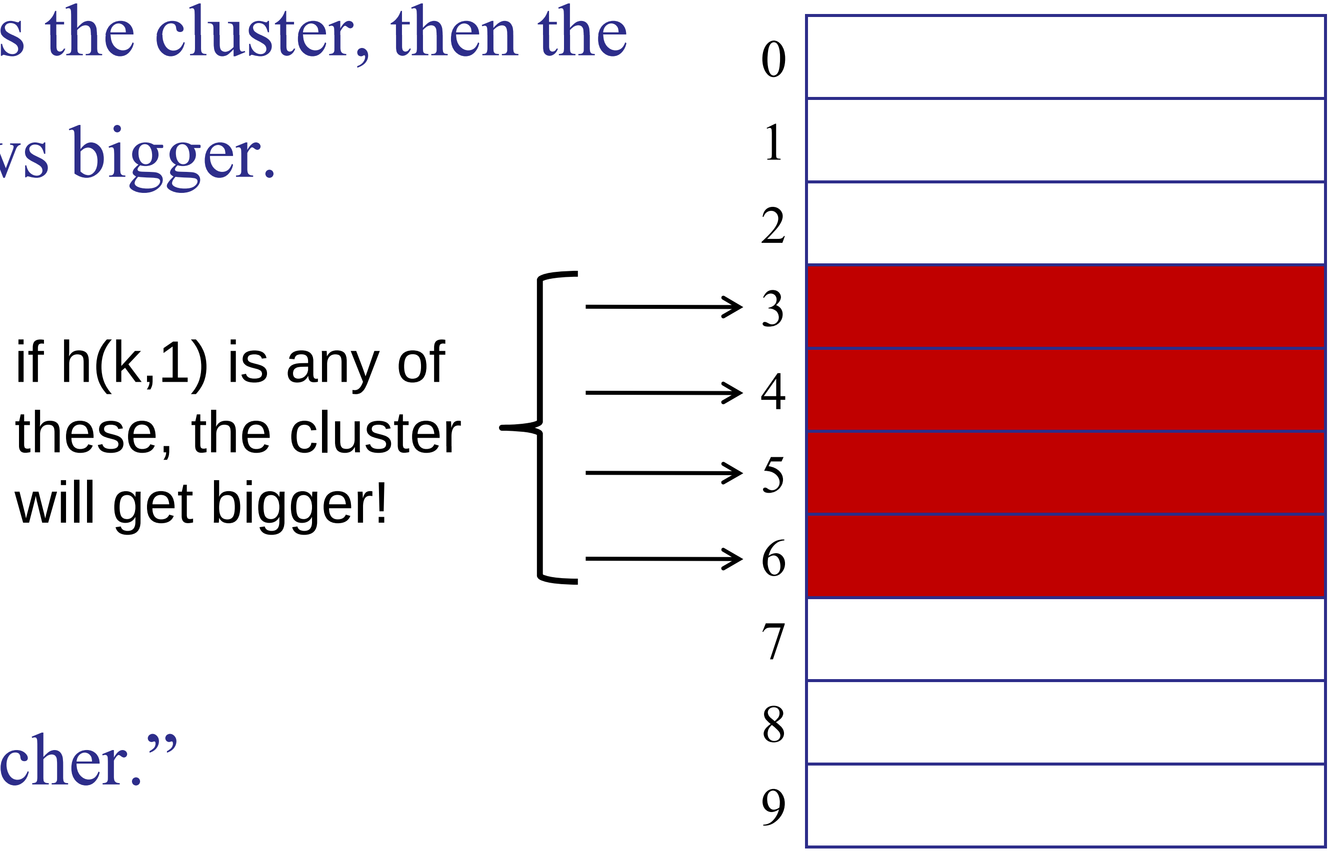
- 1 2 3 4 Pr(1/m)
- 1 2 4 3 Pr(0)
- 1 4 2 3 Pr(0)
- 1 4 3 2 Pr(0)
- …

NOT Linear Probing

Open Addressing

Problem with linear probing: ***clusters***

- If there is a cluster, then there is a higher probability that the next h(k) will hit the cluster.
- If h(k,1) hits the cluster, then the cluster grows bigger.



- “Rich get richer.”

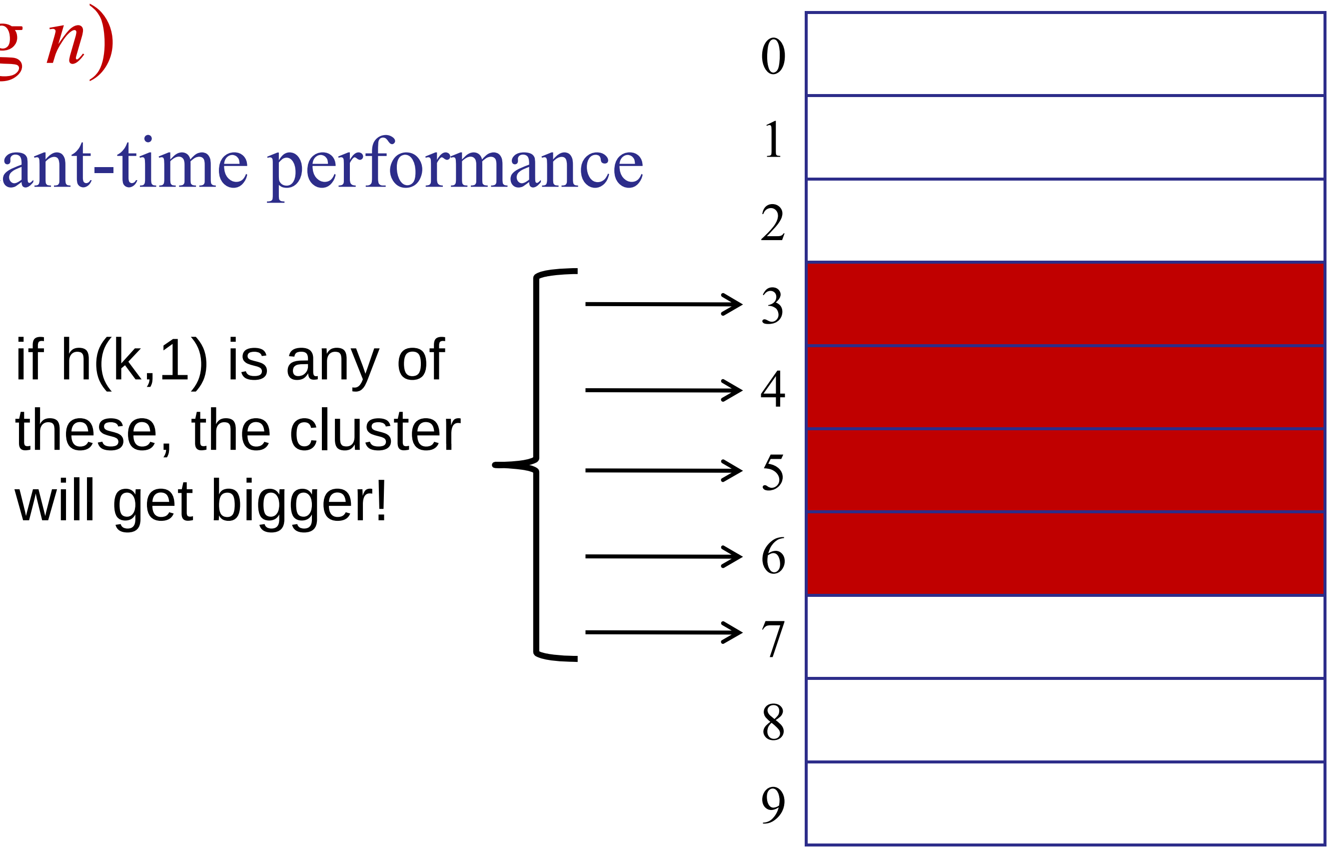
Open Addressing

Problem with linear probing: ***clusters***

- If the table is 1/4 full, then there will be clusters of size:

  $\theta(\log n)$

- Ruins constant-time performance



Open Addressing

Properties of a good hash function:

2. Uniform Hashing Assumption

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

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

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

Double Hashing

- Start with two ordinary hash functions:

$$\mathrm{f}(k),\ \mathrm{g}(k)$$

- Define new hash function:

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

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

Double Hashing

Hash function

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

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

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

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

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

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

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

Double Hashing

Hash function

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

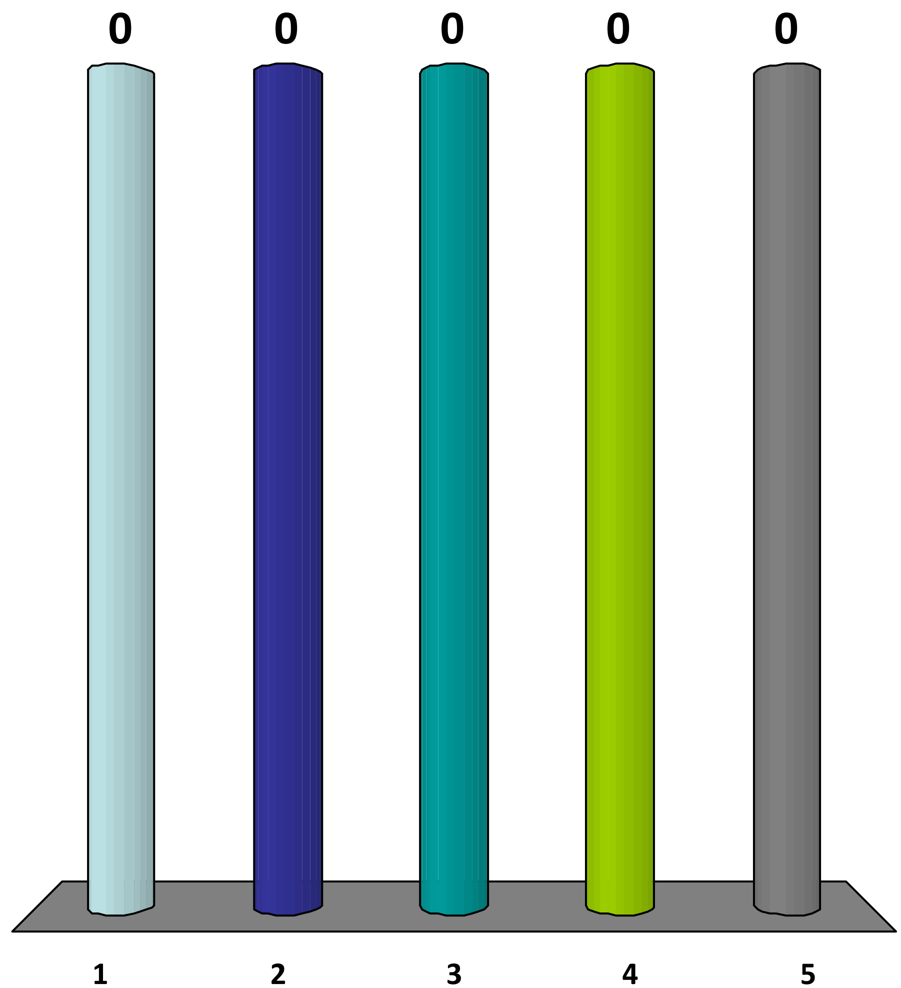
Claim: if $g(k)$ is relatively prime to m, then $h(k, i)$ hits all buckets.

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

Performance of Open Addressing

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

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



0 of 60

Performance of Open Addressing

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

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

Performance of Open Addressing

Define:

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

Performance of Open Addressing

Define:

– Load $\alpha = n / m$ Average # items / bucket

– Assume $\alpha < 1$.

Claim:

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

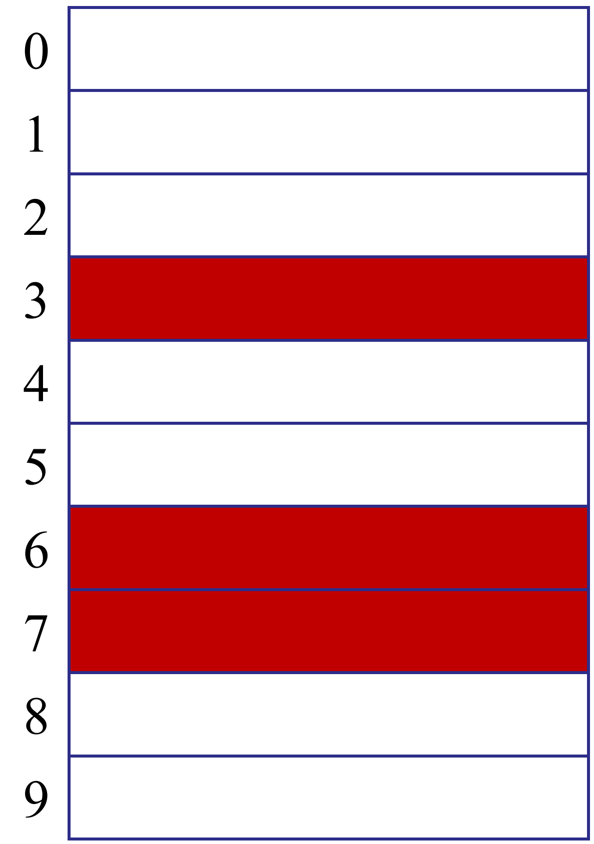
$$\leq \frac{1}{1-\alpha}$$

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

Performance of Open Addressing

Proof of Claim:

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

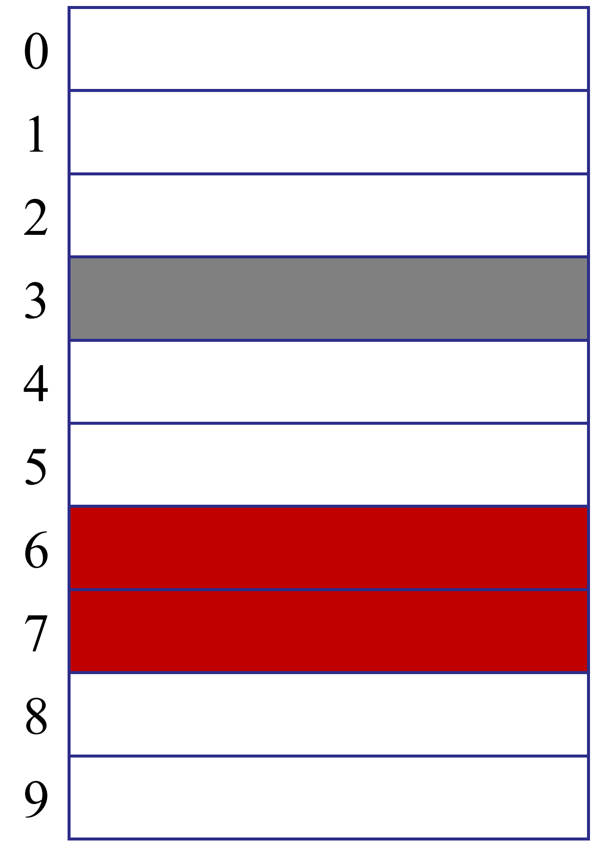


Performance of Open Addressing

Proof of Claim:

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

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



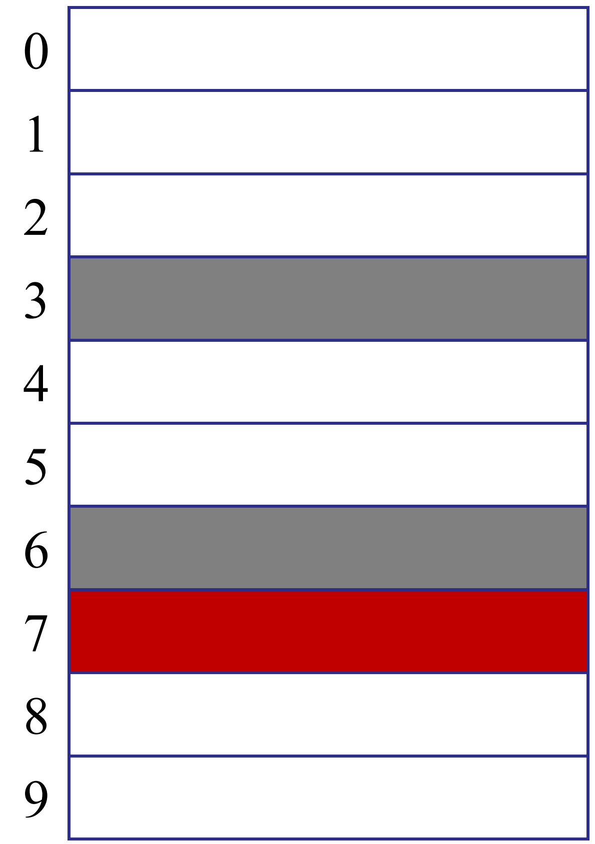
Performance of Open Addressing

Proof of Claim:

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

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

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



Performance of Open Addressing

Proof of Claim:

– Expected cost:

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

First probe   Second probe   Third probe

Performance of Open Addressing

Proof of Claim:

- Expected cost:

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

- Note:

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

Performance of Open Addressing

Proof of Claim:

– Expected cost:

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

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

Performance of Open Addressing

Proof of Claim:

- Expected cost:

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

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

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

Performance of Open Addressing

Proof of Claim:

– Expected cost:

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

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

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

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

Performance of Open Addressing

Define:

- Load $\alpha = n / m$

Average # items / bucket

- Assume $\alpha < 1$.

Claim:

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

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

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

Advantages…

Open addressing:

- Saves space (no linked lists)
- Rarely allocate memory
- Better cache performance
  - Table all in one place in memory
  - Fewer accesses to bring table into cache.
  - Linked lists can wander all over the memory.

Disadvantages…

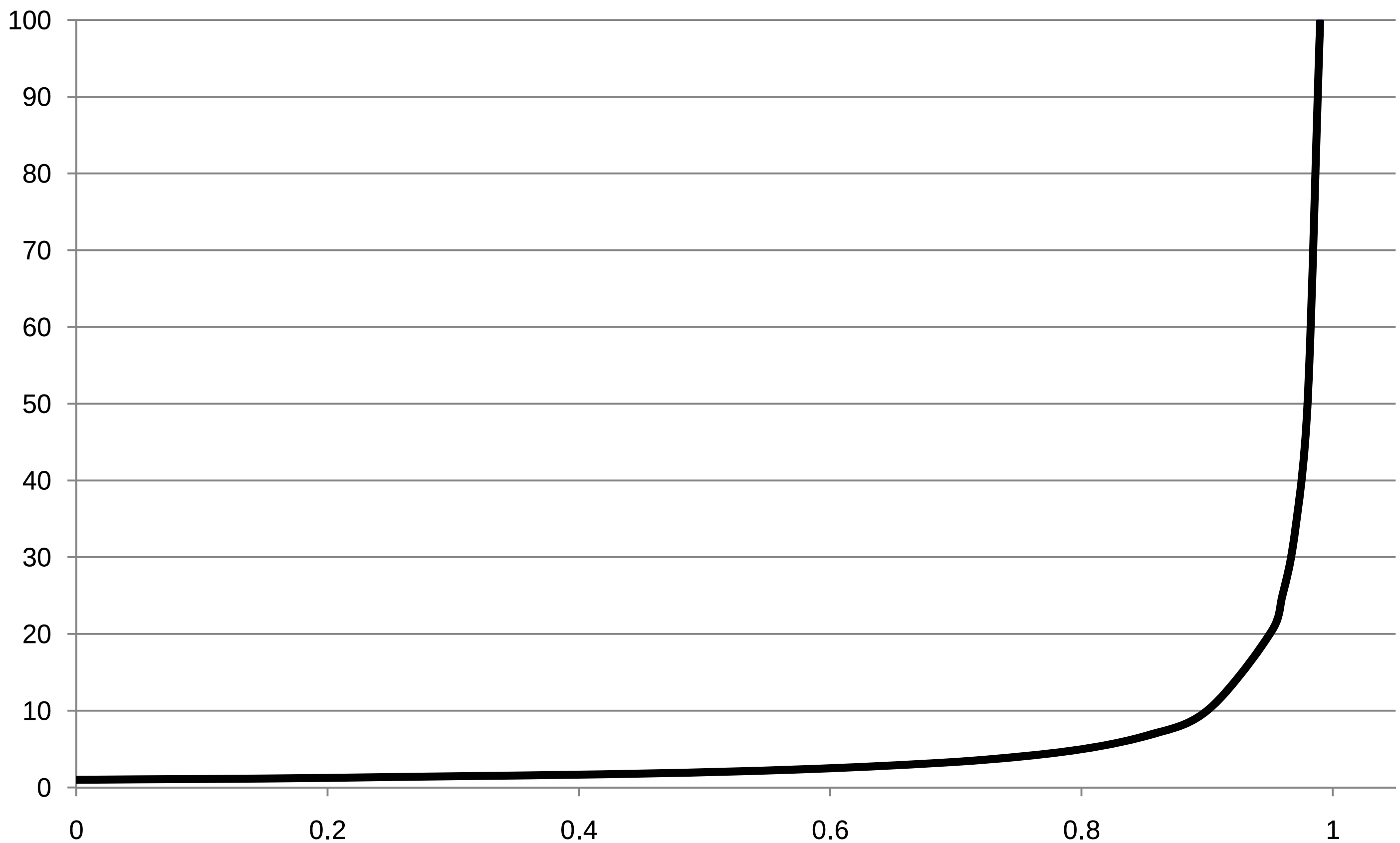
Open addressing:

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

Disadvantages…

Open addressing:

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



Advanced Hashing

Universal Hashing

- Any one hash function might be bad.
- Imagine:
  - I choose hash function h today.
  - I ship my application.
  - Bad luck! Function h is not a good hash function.

Advanced Hashing

Universal Hashing

- Choose a “universal family” of hash functions:

  $h_1, h_2, h_3, h_4, \ldots, h_k$

- Select a hash function at random every time you build a hash table.

- Example: Multiplication Method parameterized by A. Choose A at random.

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

Advanced Hashing

Universal Hashing

- Choose a "universal family" of hash functions:

  $$h_1, h_2, h_3, h_4, \ldots, h_k$$

- Select a hash function at random every time you build a hash table.
- Prove that for every x, y,: $\Pr(h(x){==}h(y)) \leq 1/m$
- No simple uniform hashing assumption!
- O(1) expected cost per operations.

Advanced Hashing

Universal Hashing

Fun exercise: prove that the following is a universal family of hash functions:

Let m be a prime number.

Define $h_{ab}(k) = a \cdot k + b \pmod{m}$

Advanced Hashing

Even Better: **Perfect Hashing**

- Hash table with ***zero*** collision.
- No linked lists, no probe sequences.
- Guarantee O(1) worst-case search.

Only for a static set of keys:

- Given a fixed set of elements to store in a hash table.

Perfect Hashing

Idea 1: Use a very big table

- If ($m = n^2$), then an element collides with probability:

  $$n/n^2 \le 1/n$$

- The probability that no two elements collide:

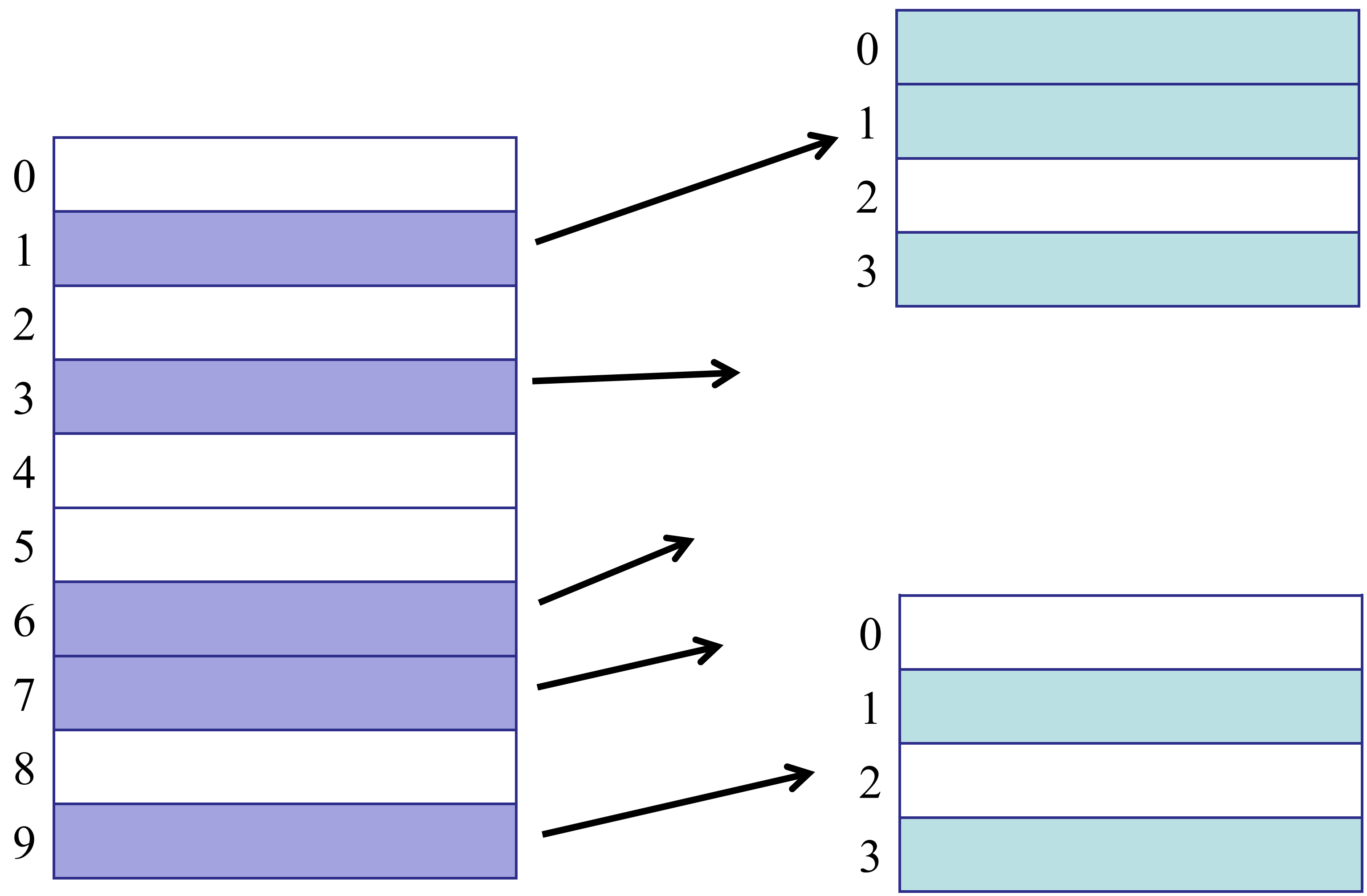
  $$\le \left(1 - \frac{1}{n}\right)^n \le \frac{1}{2}$$

- Repeatedly choose new hash function until no collision.

  Expected number of re-tries: O(1)

Perfect Hashing: Hierarchical Approach

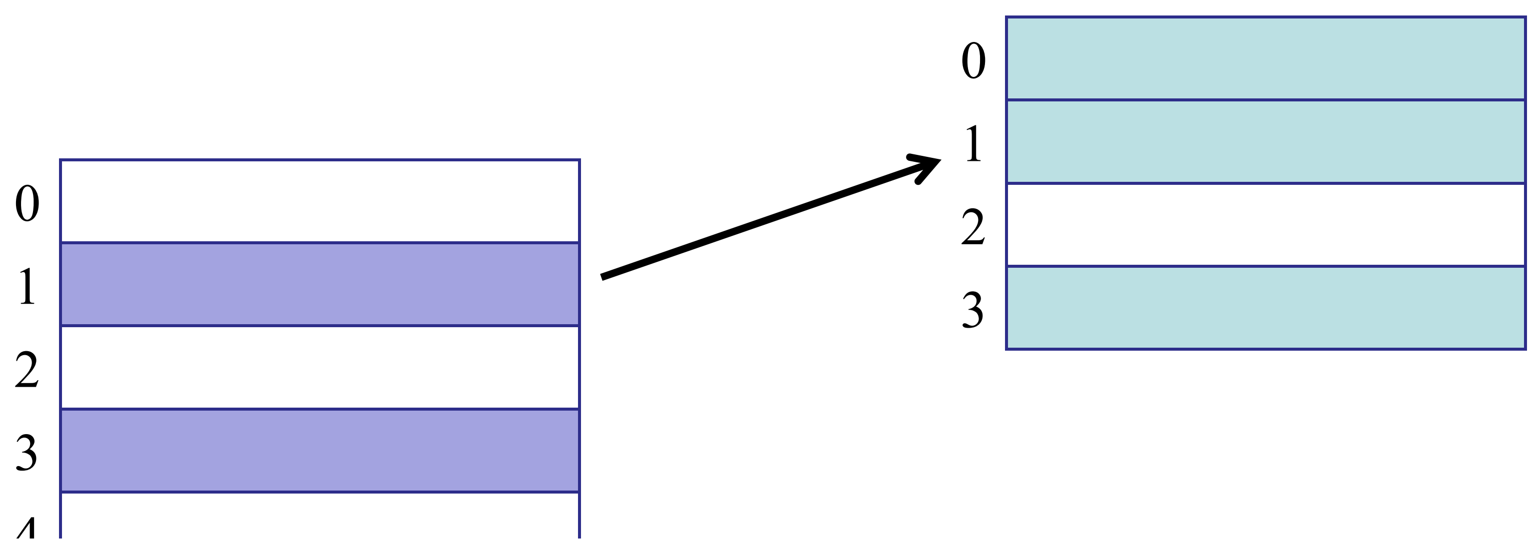
Idea 2: Use a two-level table



Perfect Hashing

Idea 2: Use a two-level table

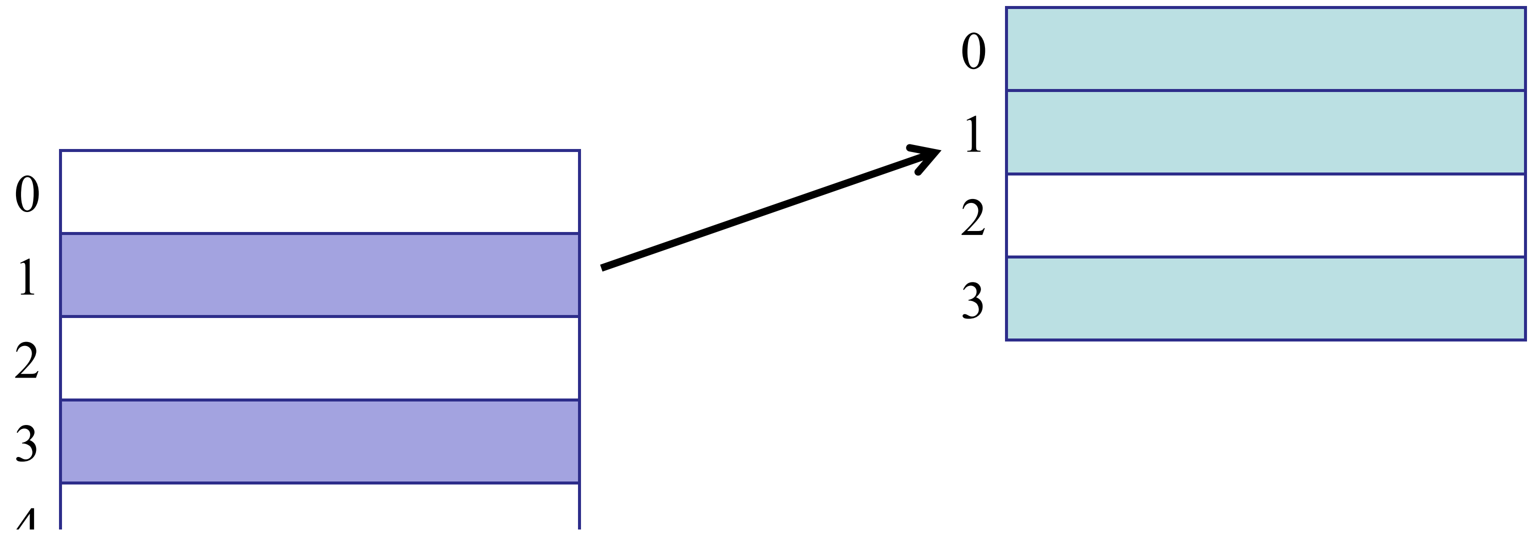
- Table 1: size = $\Theta(n)$
- Expected number of items / bucket: $\Theta(1)$



Perfect Hashing

Idea 2: Use a two-level table

- Assume table contains k elements.
- Table 2: size = $\Theta(k^2)$
- On average: constant # of items in Table 2.
- On average: constant size…



Perfect Hashing

Idea 2: Use a two-level table

- Prove: expected space is O(n)

- What happens when new items are inserted?
  - Collision in table 1, with constant probability.
  - Collision in table 2, with constant probability.
  - Every so often, rebuild all tables.
  - Amortized analysis??

Summary

Open Addressing

- Efficient technique for keeping items in the table.
- Many different techniques for probing.
  - Quadratic Probing
  - Cuckoo hashing

Advanced Hashing

- Universal Hashing
- Perfect Hashing