CS2020
Data Structures and Algorithms

Hashing (Part 2)

Today

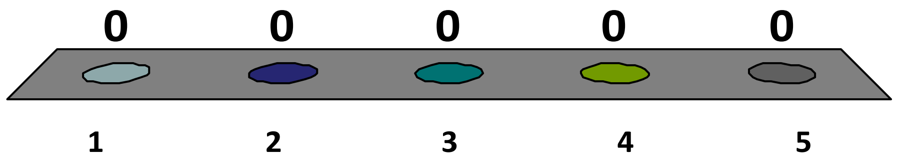
- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Review: Symbol Table Abstract Data Type

Which of the following is *not* typically a symbol table operation?

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

Response Counter

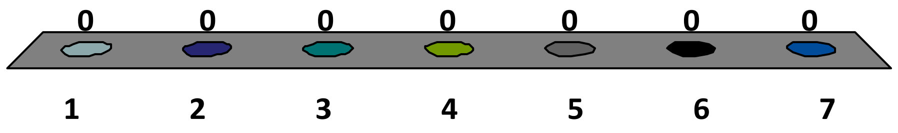


Review: Symbol Table Abstract Data Type

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

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

Response Counter



Today

- Java hashing implementation
- Table sizing
- Resolving Collisions

Hashing in Java

How does your program know which hash function to use?

```
HashMap<MyFoo, Integer> hmap = new ...
MyFoo foo = new MyFoo();

hmap.put(foo, 8);
```

Java Hash Functions

Every object supports the method:

```
int hashCode()
```

Rules:

- Always returns the same value, if the object hasn't changed.
- If two objects are equal, then they return the same hashCode.

Is it legal for every object to return 32?

Hashing in Java

How does your program know which hash function to use?

```
HashMap<MyFoo, Integer> hmap = new ...
MyFoo foo = new MyFoo();

hmap.put(foo, 8);

int hash = foo.hashCode();
```

Java Object

Every class extends Object

public class	**Object**	
Object	clone()	*creates a copy*
boolean	**equals(Object obj)**	***is obj equal to this?***
void	finalize()	*used by garbage collector*
Class	getClass()	*returns class*
int	**hashCode()**	***calculates hash code***
void	notify()	*wakes up a waiting thread*
void	notifyAll()	*wakes up all waiting threads*
String	**toString()**	***returns string representation***
void	wait(…)	*wait until notified*

Java Hash Functions

Every object supports the method:

```
int hashCode()
```

Rules:

- Always returns the same value, if the object hasn’t changed.
- If two objects are equal, then they return the same hashCode.

Is it legal for every object to return 32?

Java Hash Functions

Every object supports the method:

```
int hashCode()
```

Default Java implementation:

- hashCode returns the memory location of the object
- Every object hashes to a different location

Must override `hashCode()` for your class.

Java Integer HashCode

```
public int hashCode() {
  return value;
}
```

Java Long HashCode

```java
public int hashCode() {
   return (int)(value ^ (value >>> 32));
}
```

$$\text{hash}(\overbrace{0\;1\;1\;0\;0\;1\;0\;1\;1\;0\;0\;1\;1\;1\;0\;0}^{\text{32 bits}}\;\overbrace{0\;0\;1\;0\;1\;0\;0\;0\;0\;1\;1\;0\;0\;1\;0\;0}^{\text{32 bits}})$$

$$\begin{array}{rl} & 0\;1\;1\;0\;0\;1\;0\;1\;1\;0\;0\;1\;1\;1\;0\;0 \\ \text{XOR} & 0\;0\;1\;0\;1\;0\;0\;0\;0\;1\;1\;0\;0\;1\;0\;0 \\ \hline & 0\;1\;0\;0\;1\;1\;0\;1\;1\;1\;1\;1\;1\;0\;0\;0 \end{array}$$

Java String HashCode

```java
public int hashCode() {
   int h = hash; // only calculate hash once
   if (h == 0 && count > 0) {  // empty = 0
        int off = offset;
        char val[] = value;
        int len = count;
        for (int i = 0; i < len; i++) {
             h = 31*h + val[off++];
        }
        hash = h;
   }
   return h;
}
```

Java String

HashCode calculation:

```
  hash = s[0]*31^(n-1) +
         s[1]*31^(n-2) +
         s[2]*31^(n-3) +
         ... +
         s[n-2]*31 +
         s[n-1]
```

Why did they choose 31?

Creating a new class

```
public class Pair {
  private int first;
  private int second;

  Pair(int a, int b){
     first = a;
     second = b;
  }
}
```

Creating a new class

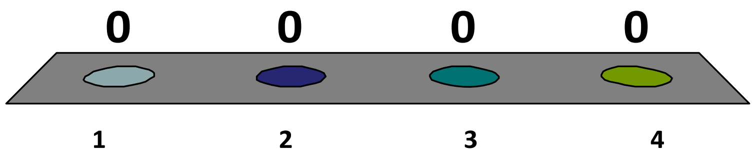
```
public void testPair() {

  HashMap<Pair, Integer> htable =
           new HashMap<Pair, Integer>();
  Pair one = new Pair(20, 20);
  htable.put(one, 7);

  Pair two = new Pair(20, 20);
  int question = htable.get(two);
}
```

htable.get(new Pair(20, 20)) == ?

1. 1
2. 7
3. 11
4. [✓ null]



Response Counter

Creating a new class

```
Pair one = new Pair(20, 20);
Pair two = new Pair(20, 20);

one.hashCode() != two.hashCode()
```

Creating a new class

```
Pair one = new Pair(20, 20);
Pair two = new Pair(20, 20);
htable.put(one, 7);
```

```
htable.get(one) → 7

htable.get(two) → null
```

Creating a new class

```
public class Pair {
  private int first;
  private int second;

  Pair(int a, int b){
     first = a;
     second = b;
  }

  int hashCode(){
     return (first ^ second);
  }
}
```

Creating a new class

```
Pair one = new Pair(20, 20);
Pair two = new Pair(20, 20);
htable.put(one, 7);

htable.get(one) → 7
htable.get(two) → null

one.equals(two) → false
```

Java Hash Functions

Every object supports the method:

```
int hashCode()
```

Rules:

- Always returns the same value, if the object hasn’t changed.
- If two objects are equal, then they return the same hashCode.
- **Must redefine .equals to be consistent with hashCode.**

Creating a new class

```
Pair one = new Pair(20, 20);
Pair two = new Pair(20, 20);
htable.put(one, 7);

htable.get(one) => 7

htable.get(two) => null
```

Java Hash Functions

Every object supports the method:

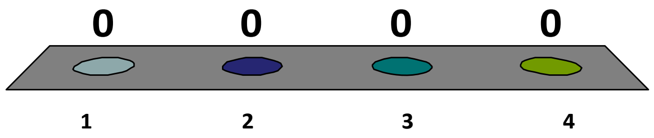
```
boolean equals(Object o)
```

Rules:

- **Reflexive**: x.equals(x) ➔ true
- **Symmetric**: x.equals(y) == y.equals(x)
- **Transitive**: x.equals(y), y.equals(z) ➔ x.equals(z)
- **Consistent**: always returns the same answer
- **Null is null**: x.equals(null) ➔ false

What is wrong here?

1. Does not calculate equality correctly.
2. ✓ Not correct signature.
3. Cannot access private p.first / p.second.
4. Not transitive.



```
boolean equals(Pair p){
  if (p == null) return false;
  if (p == this) return true;
  if (p.first != first) return false;
  if (p.second != second) return false;
  return true;
}
```

Response Counter

Java Hash Functions

Every object supports the method:

```
boolean equals(Object o)
```

```
boolean equals(Object p){
  if (p == null) return false;
  if (p == this) return true;
  if (!(p instanceOf Pair)) return false;
  Pair pair = (Pair)p;
  if (p.first != first) return false;
  if (p.second != second) return false;
  return true;
}
```

Java HashMap

```
public V get(Object key) {
   if (key == null) return getForNullKey();
   int hash = hash(key.hashCode());
   for (Entry<K,V> e = table[indexFor(hash,table.length)];
        e != null;
        e = e.next)
   {
      Object k;
      if (e.hash==hash &&((k=e.key)==key)||key.equals(k)))
         return e.value;
   }
   return null;
}
```

Java HashMap

```
// This function ensures that hashCodes that differ only
// by constant multiples at each bit position have a
// bounded number of collisions (approximately 8 at
// default load factor).

static int hash(int h) {
   h ^= (h >>> 20) ^ (h >>> 12);
   return h ^ (h >>> 7) ^ (h >>> 4);
}
```

Java HashMap

```
public V get(Object key) {
   if (key == null) return getForNullKey();
   int hash = hash(key.hashCode());
   for (Entry<K,V> e = table[indexFor(hash,table.length)];
        e != null;
        e = e.next)
   {
      Object k;
      if (e.hash==hash &&((k=e.key)==key)||key.equals(k)))
         return e.value;
   }
   return null;
}
```

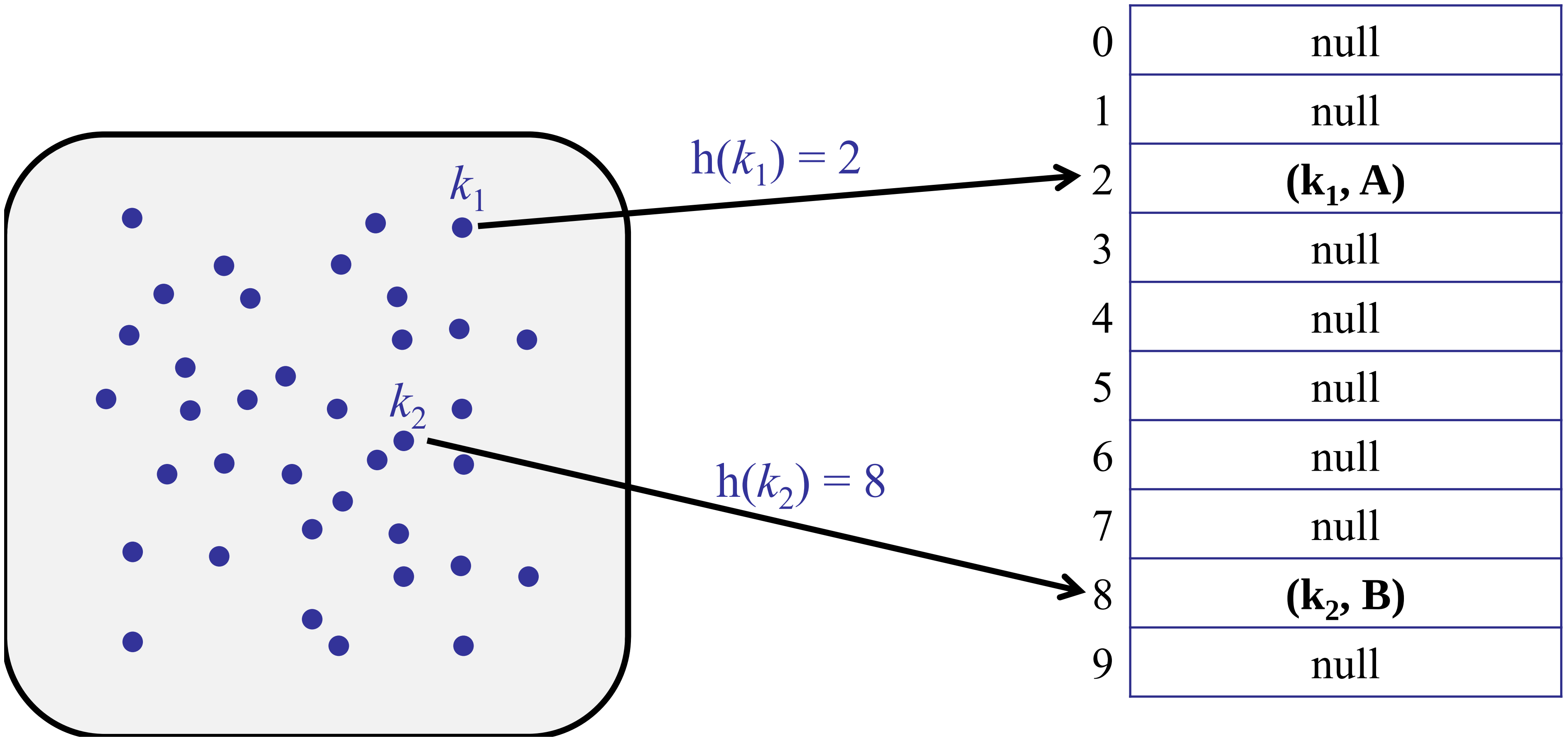
Today

- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Review

Hash Tables

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

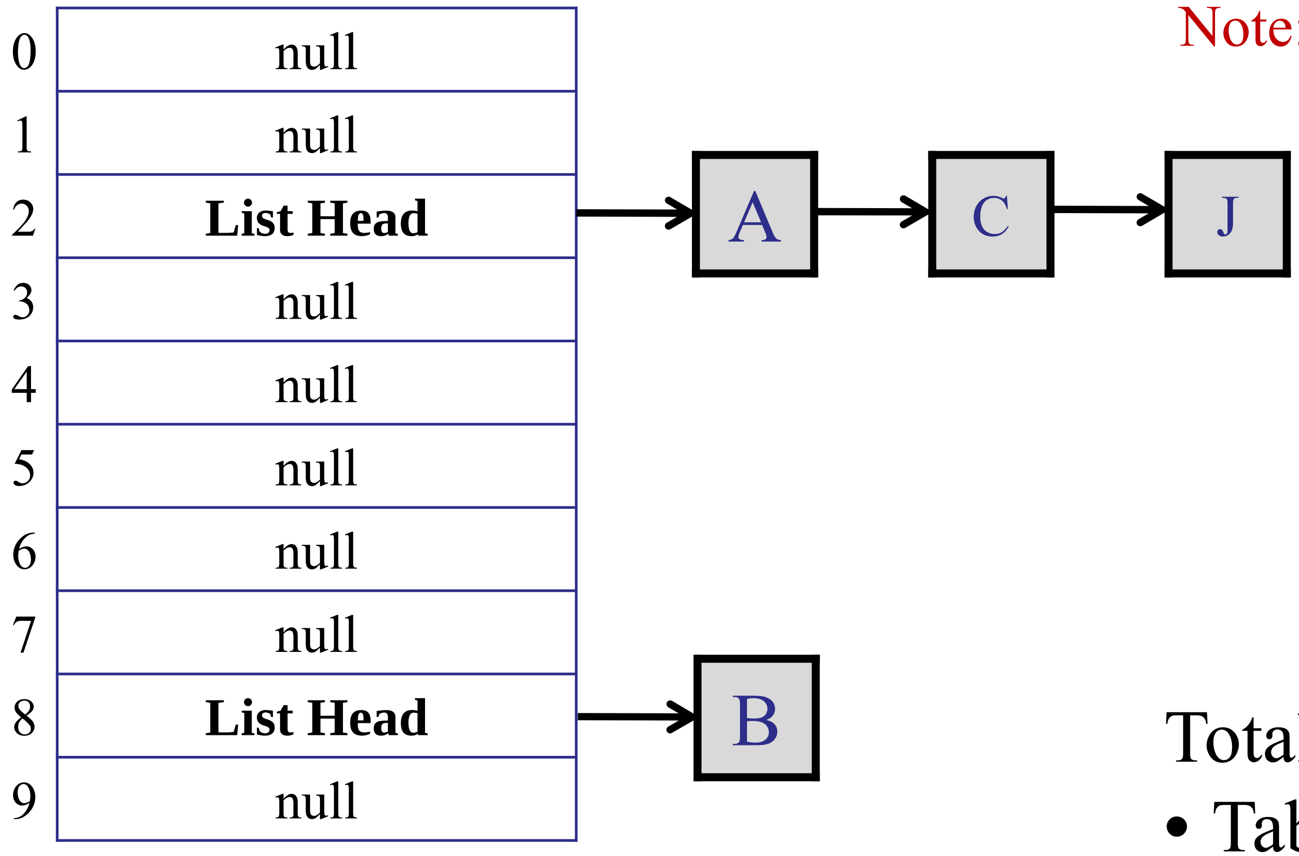


Resolving Collisions

- 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.

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 = $1 + n/m$

Open Addressing

Advantages:

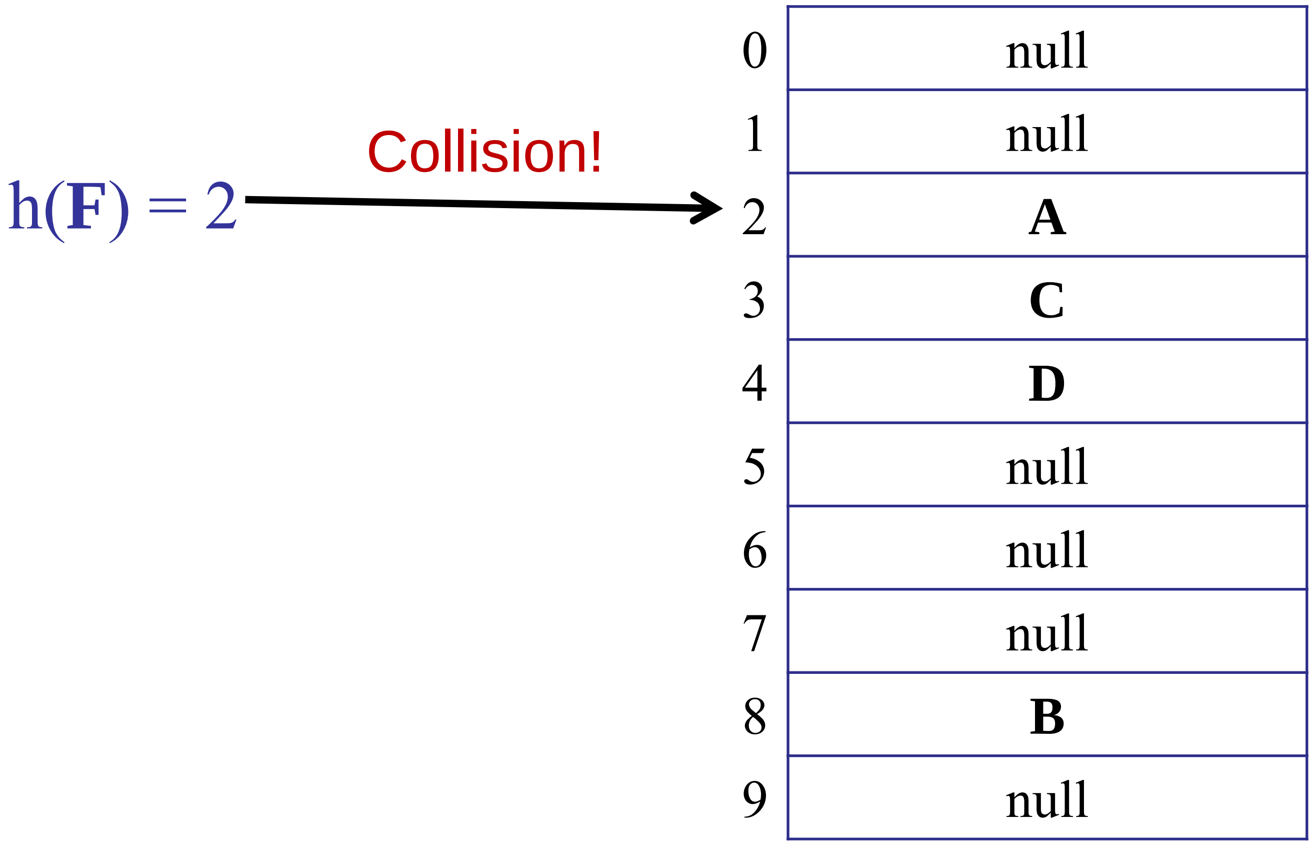
- No linked lists!
- All data directly stored in the table.
- One item per slot.

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

Open Addressing

On collision:

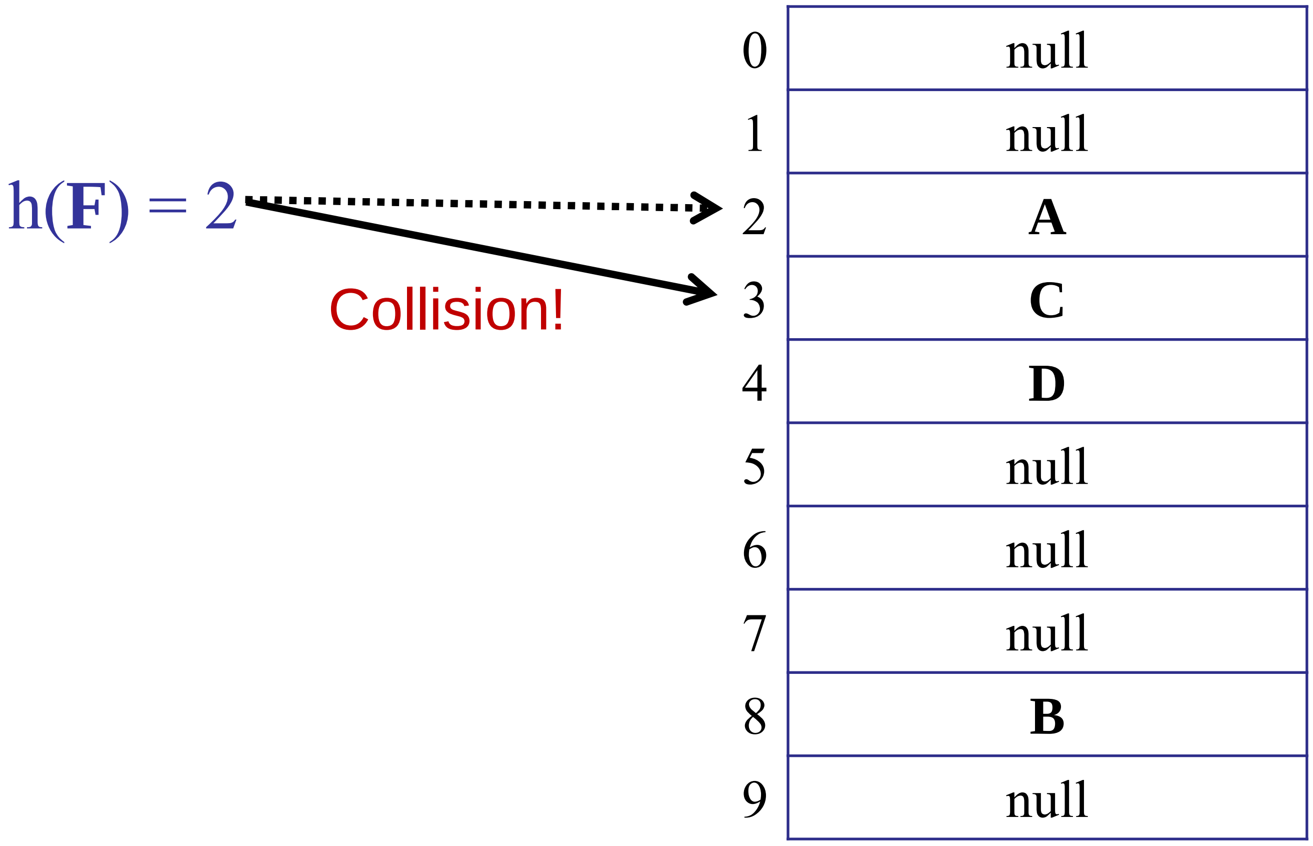
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

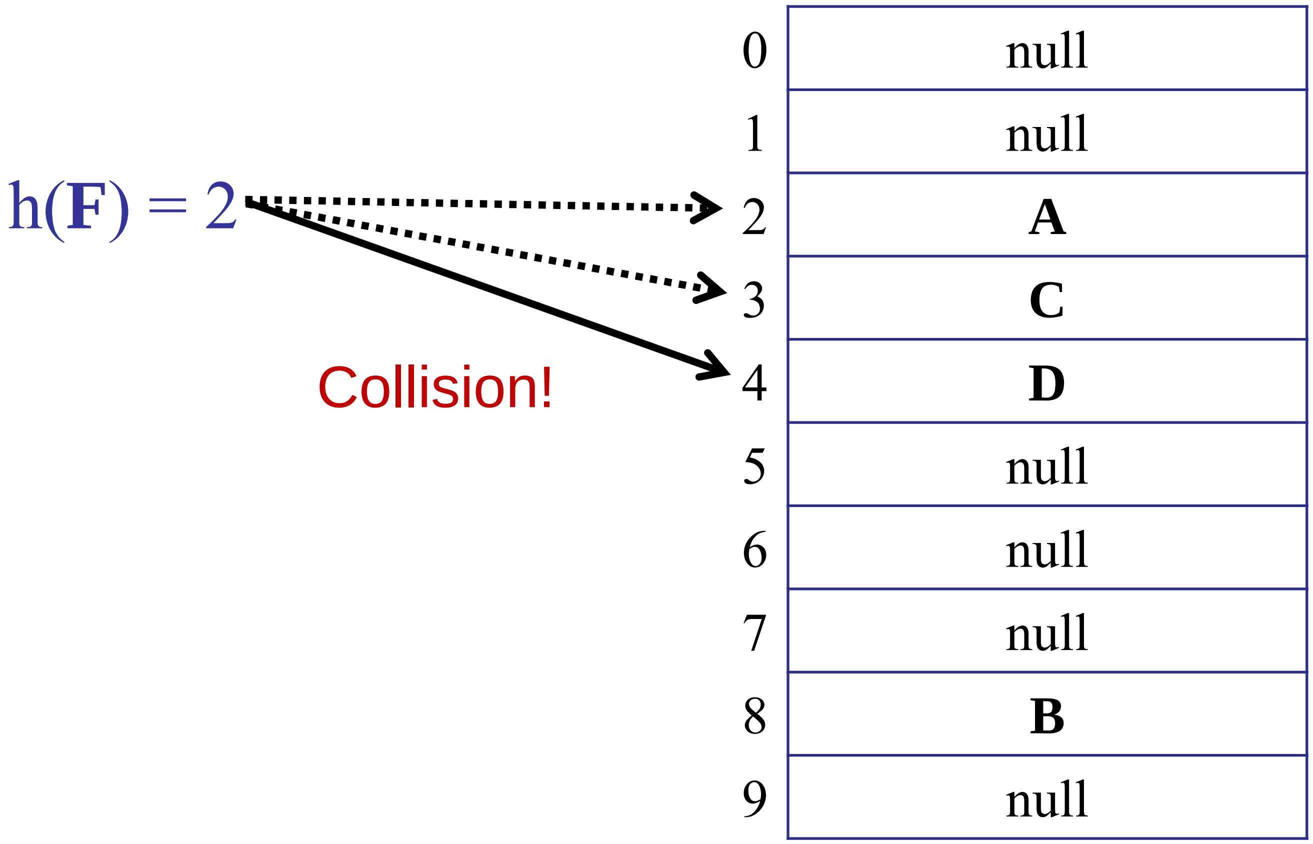
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

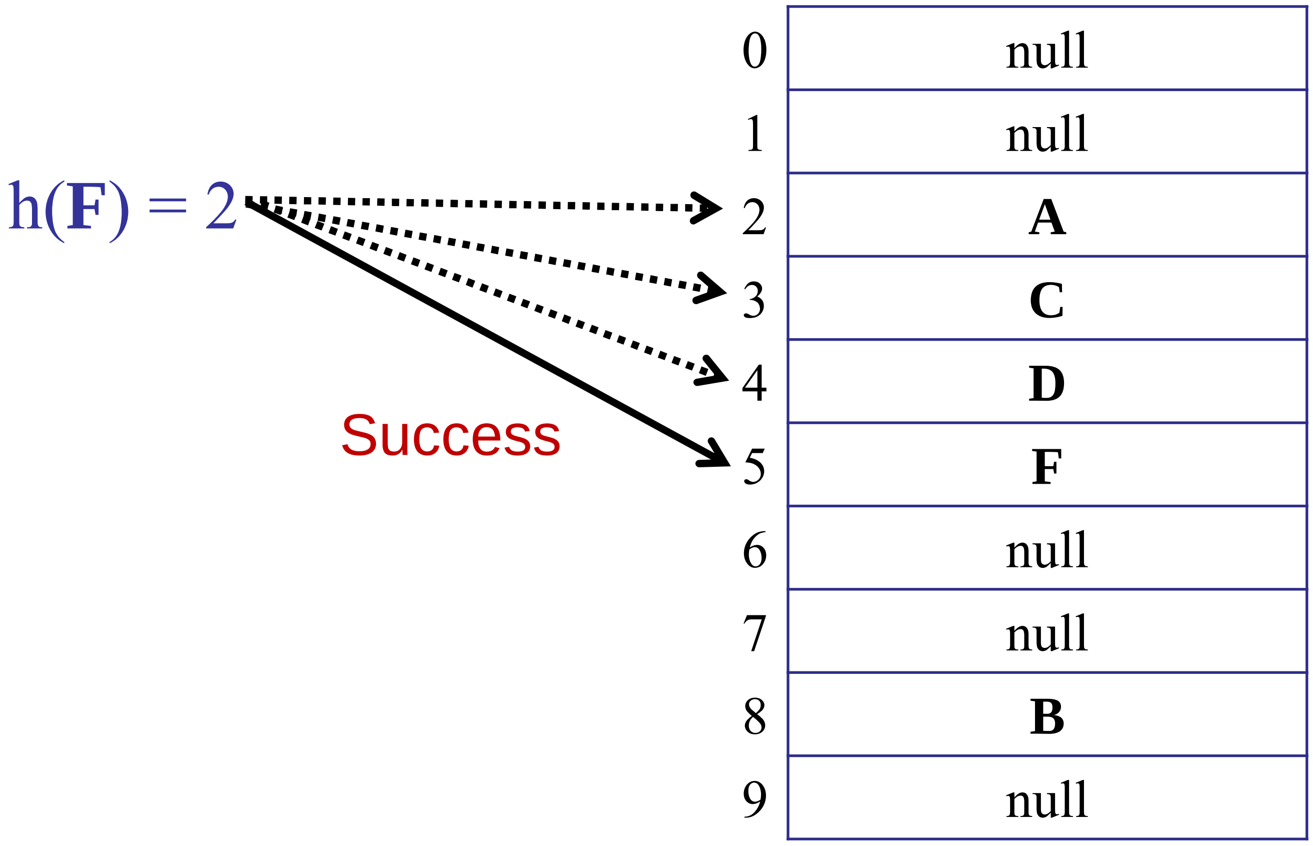
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

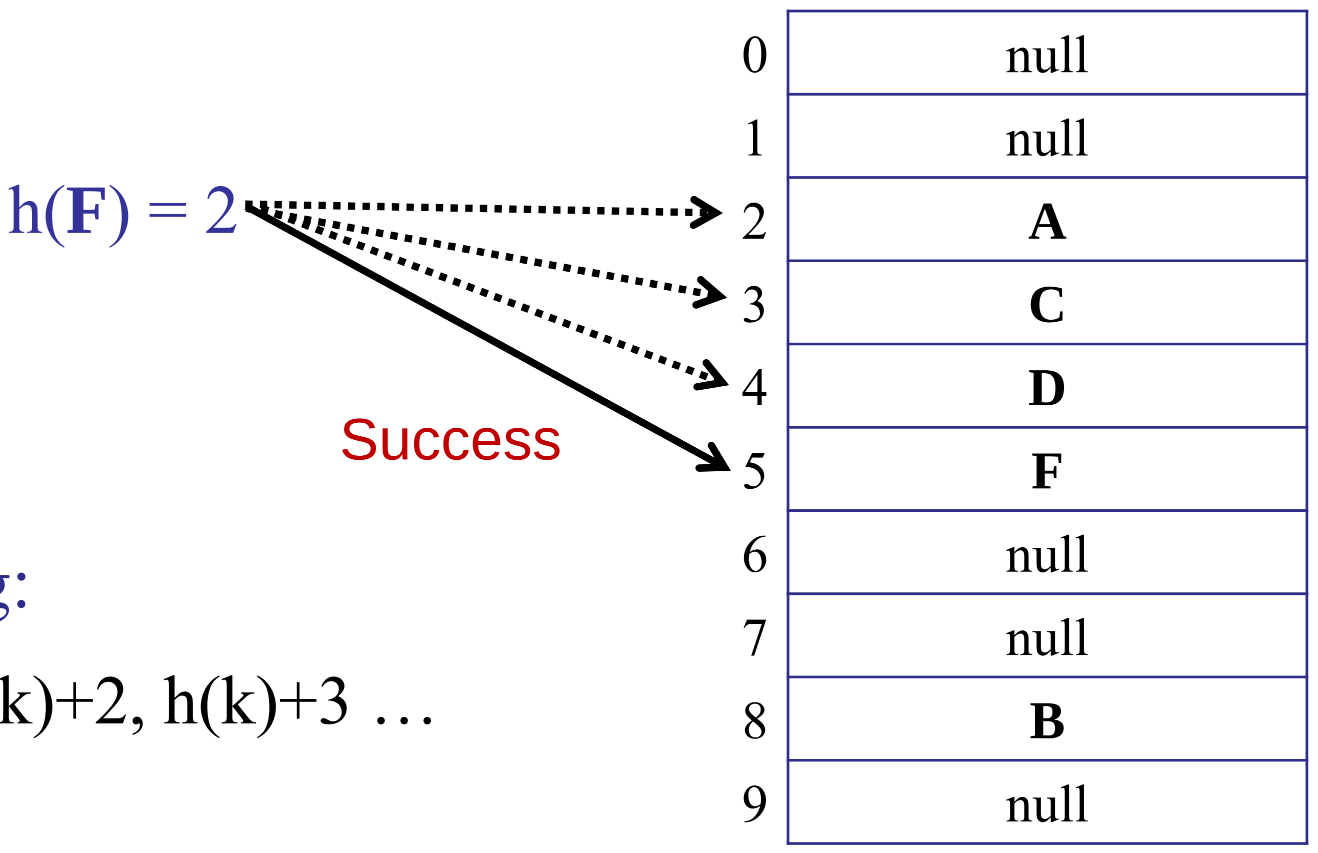
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

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



Linear Probing:

- h(k)+1, h(k)+2, h(k)+3 …

Open Addressing

Hash Function re-defined:

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

Two parameters:

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

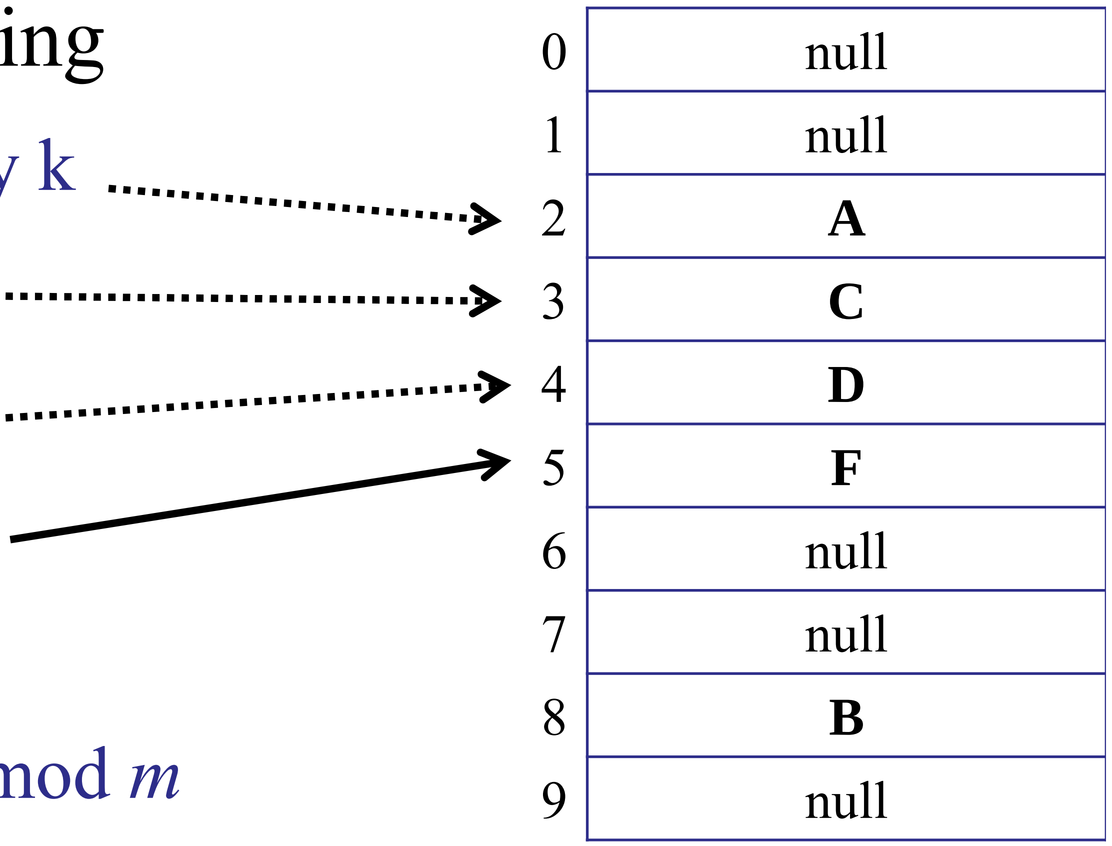
Open Addressing

Hash Function re-defined:

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

Example: Linear Probing

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



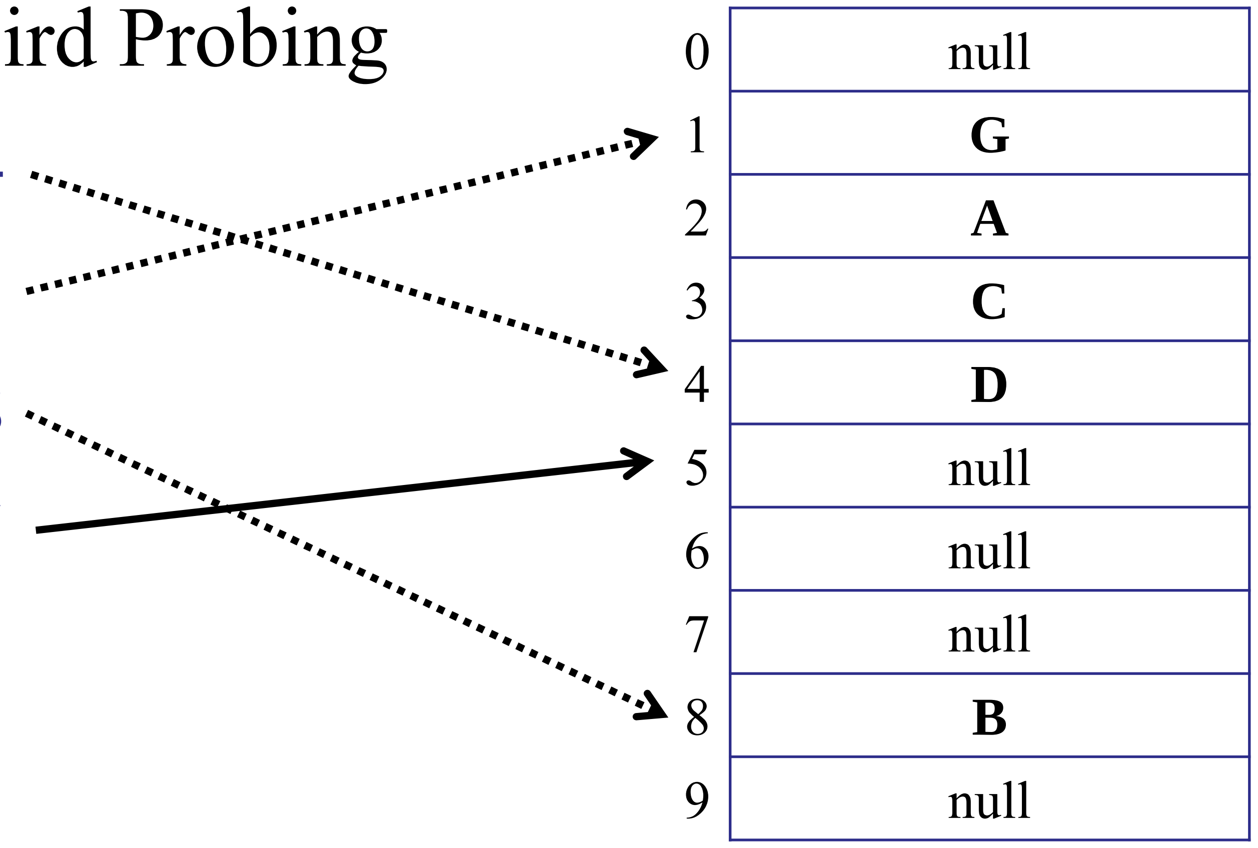
Open Addressing

Hash Function re-defined:

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

Example: Weird Probing

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



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++;
}
throw new TableFullException();           // Table full!
```

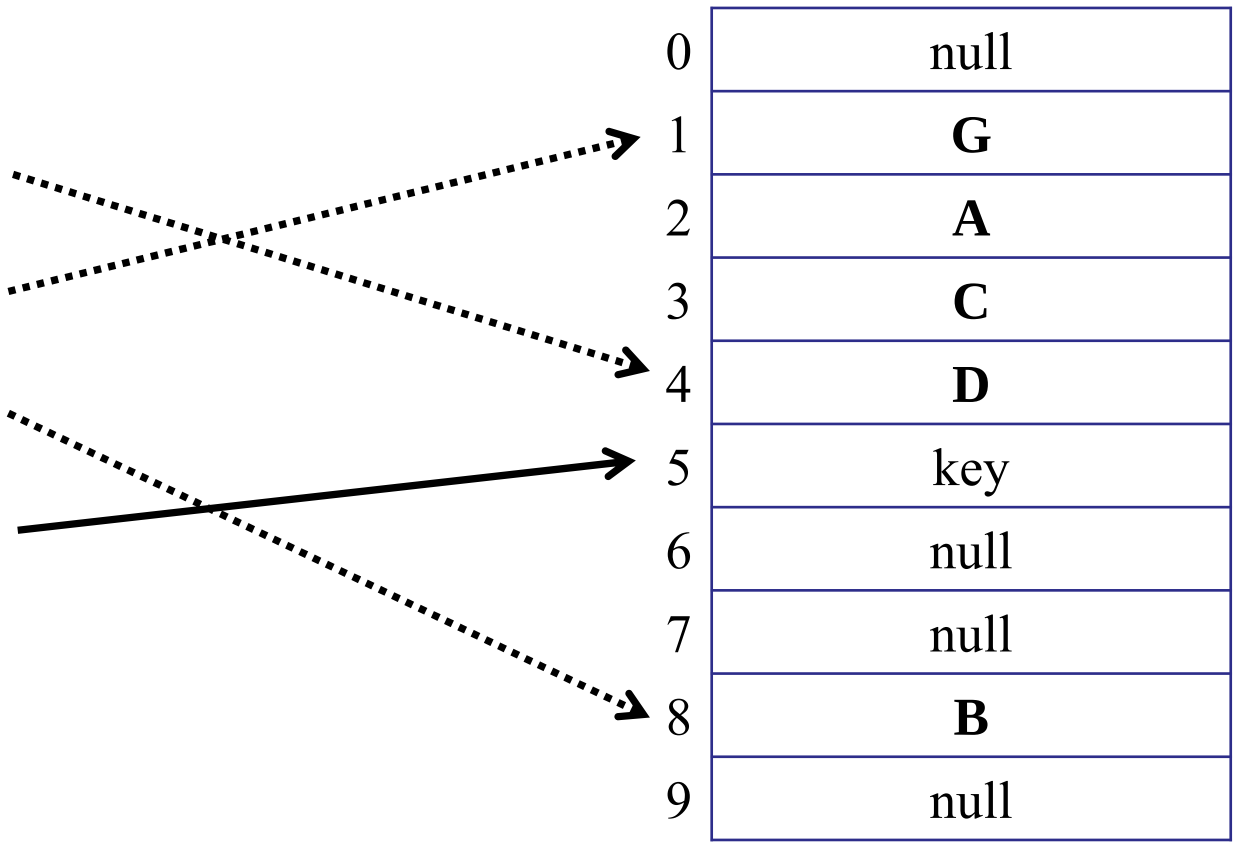
Open Addressing

Hash Function re-defined:

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

search(key)

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



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.
```

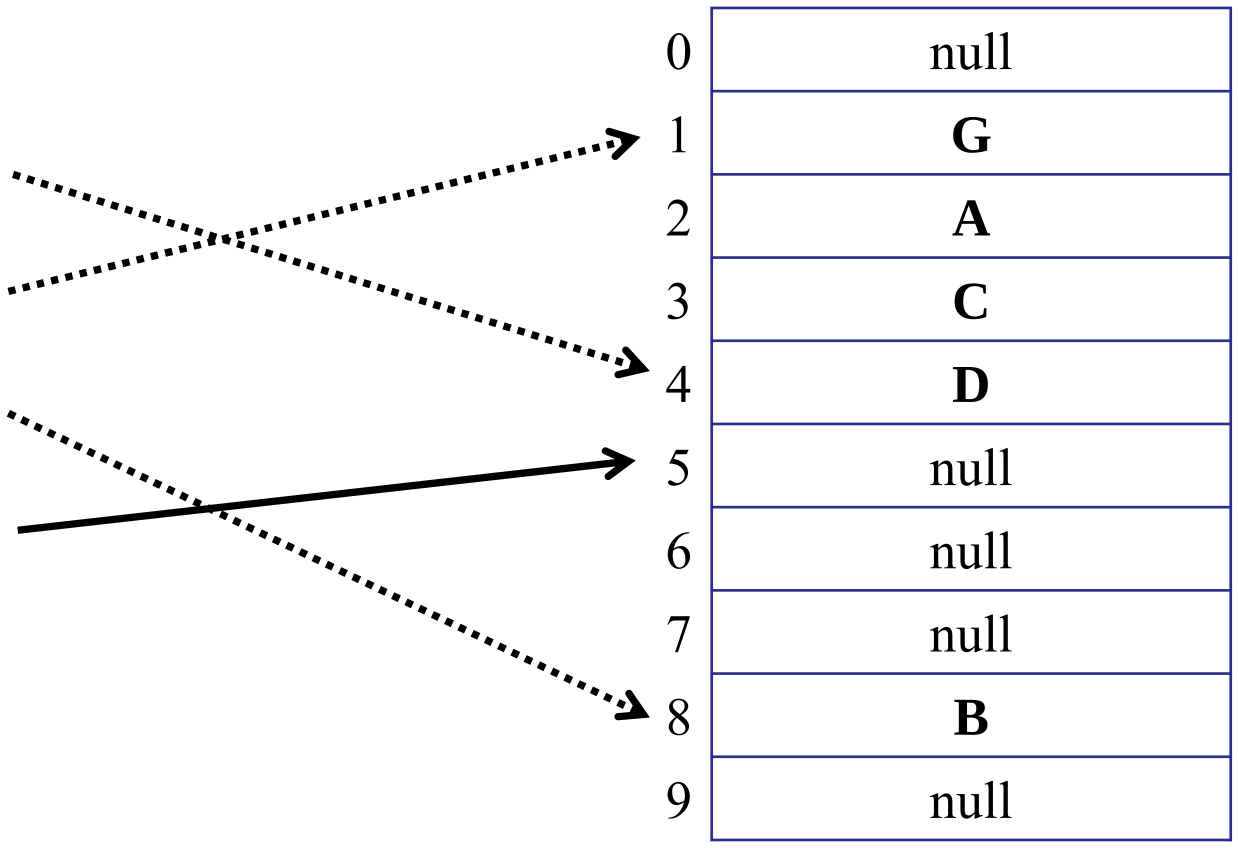
Open Addressing

Hash Function re-defined:

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

search(key)

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



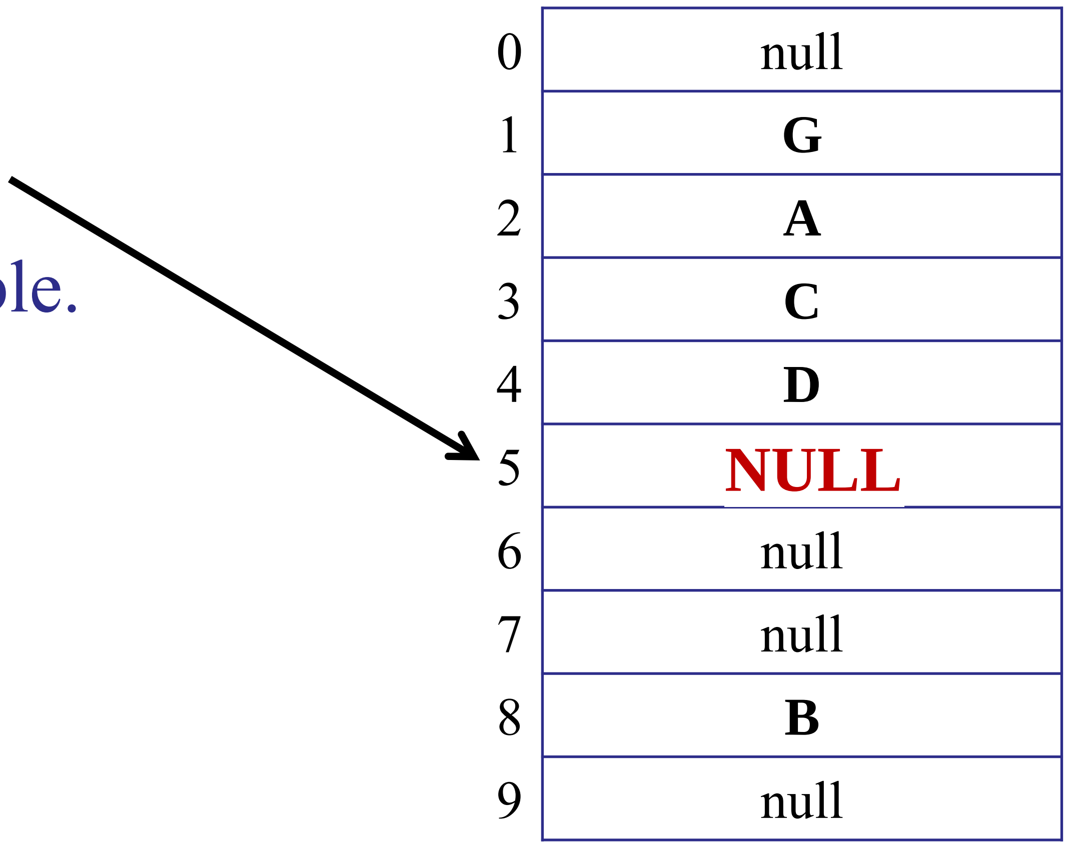
Open Addressing

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.



What is wrong with delete?

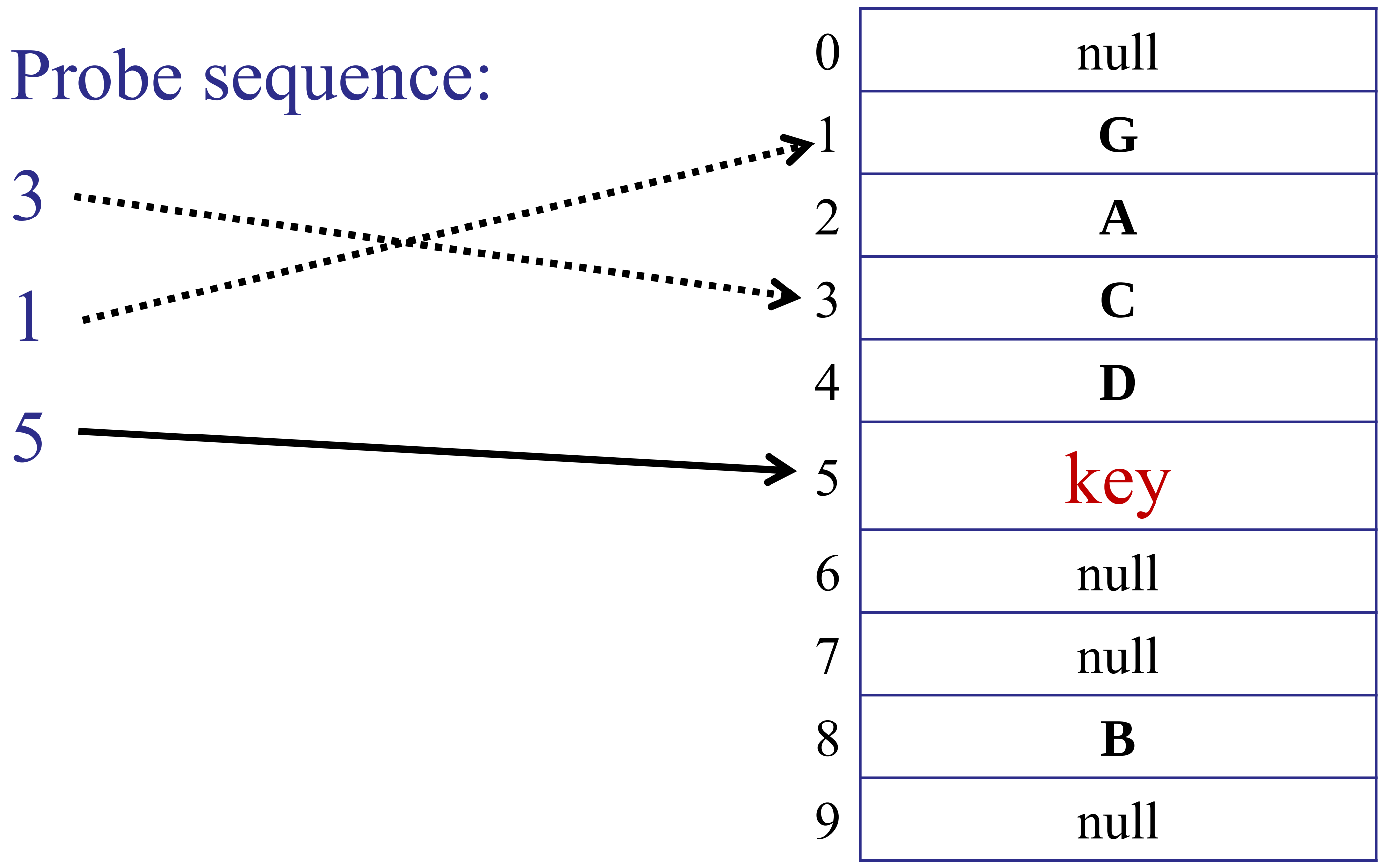
1. ✓ Search may fail to find an element.
2. The table will have gaps in it.
3. Space is used inefficiently.
4. If the key is inserted again, it may end up in a different bucket.

Response Counter

Open Addressing

insert(key)

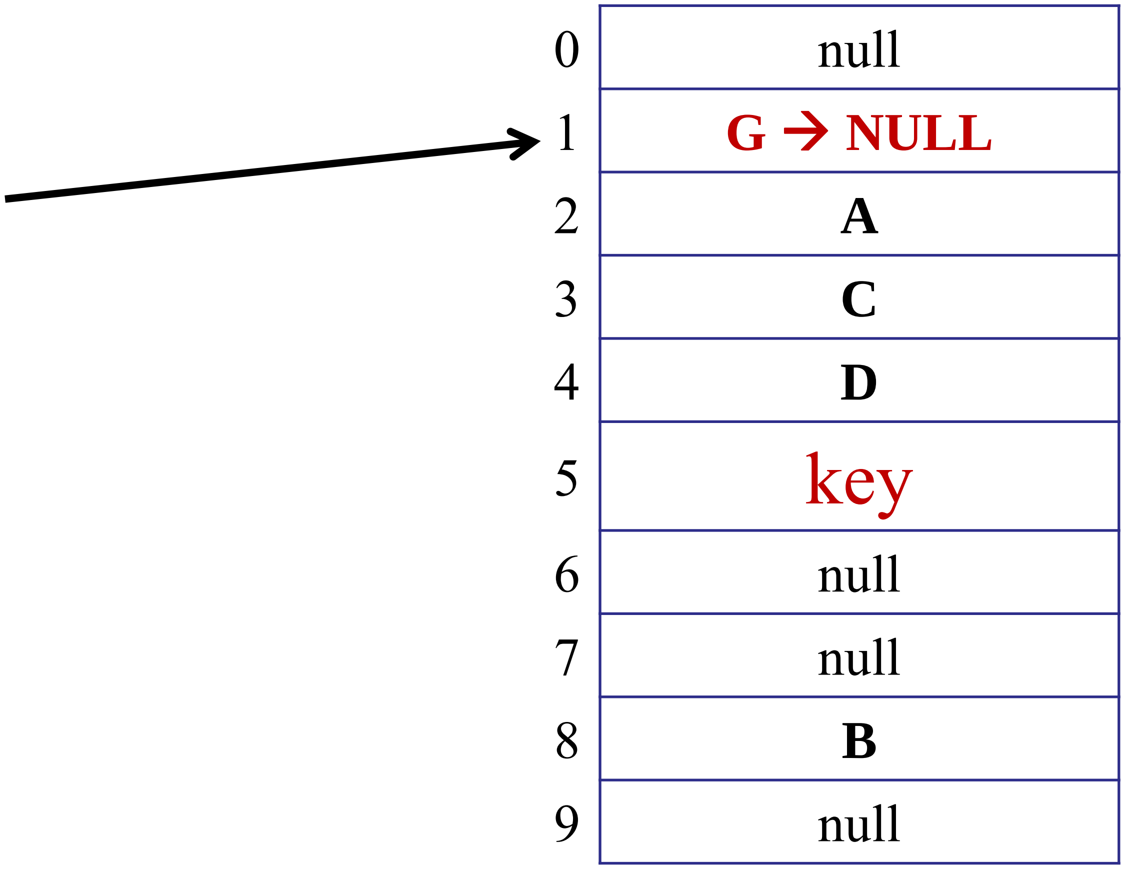
Probe sequence:



Open Addressing

insert(key)

delete(G)



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)

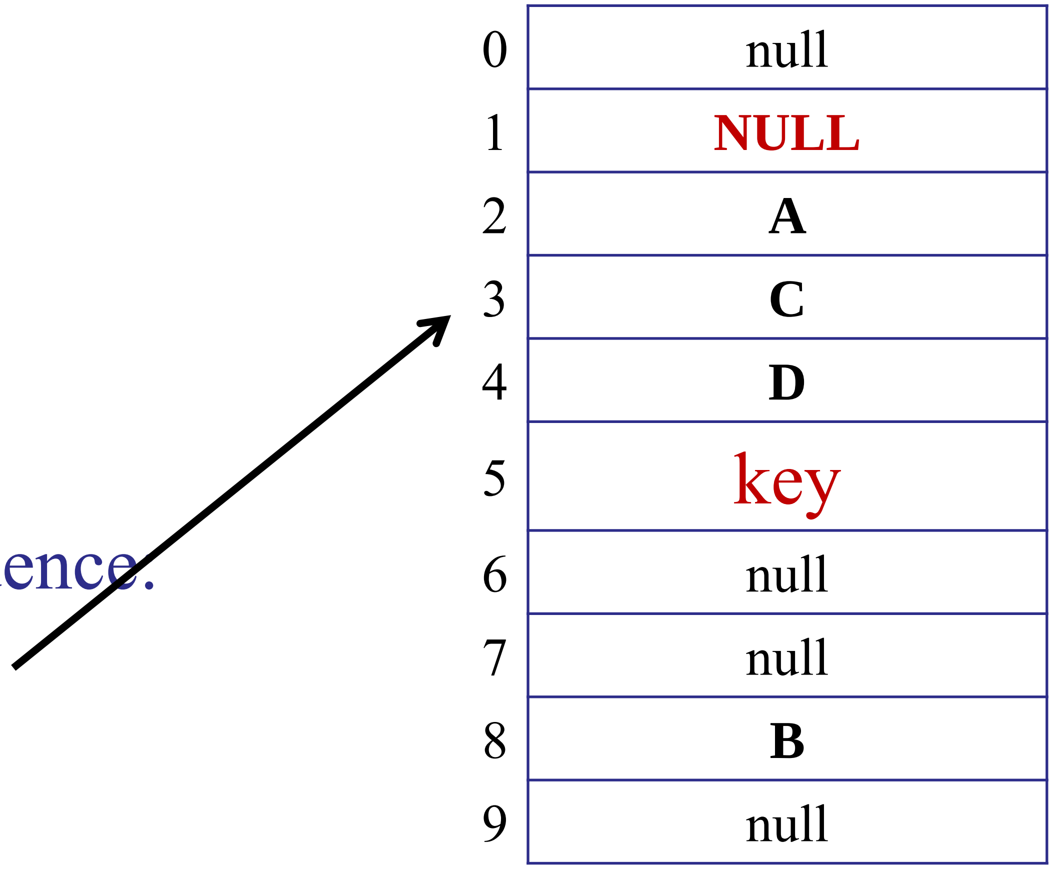
search(key)

Probe sequence:

3

1

5



Open Addressing

insert(key)

delete(G)

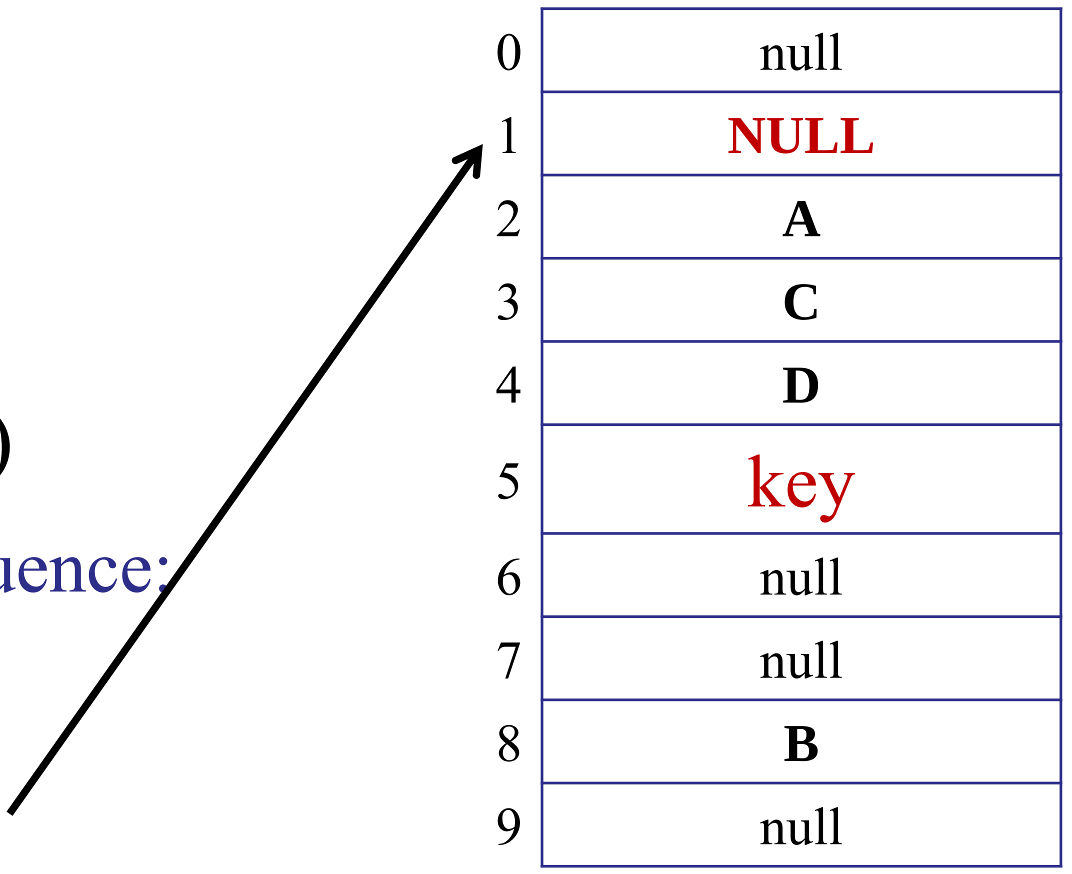
search(key)

Probe sequence:

3

1

Not found!



Open Addressing

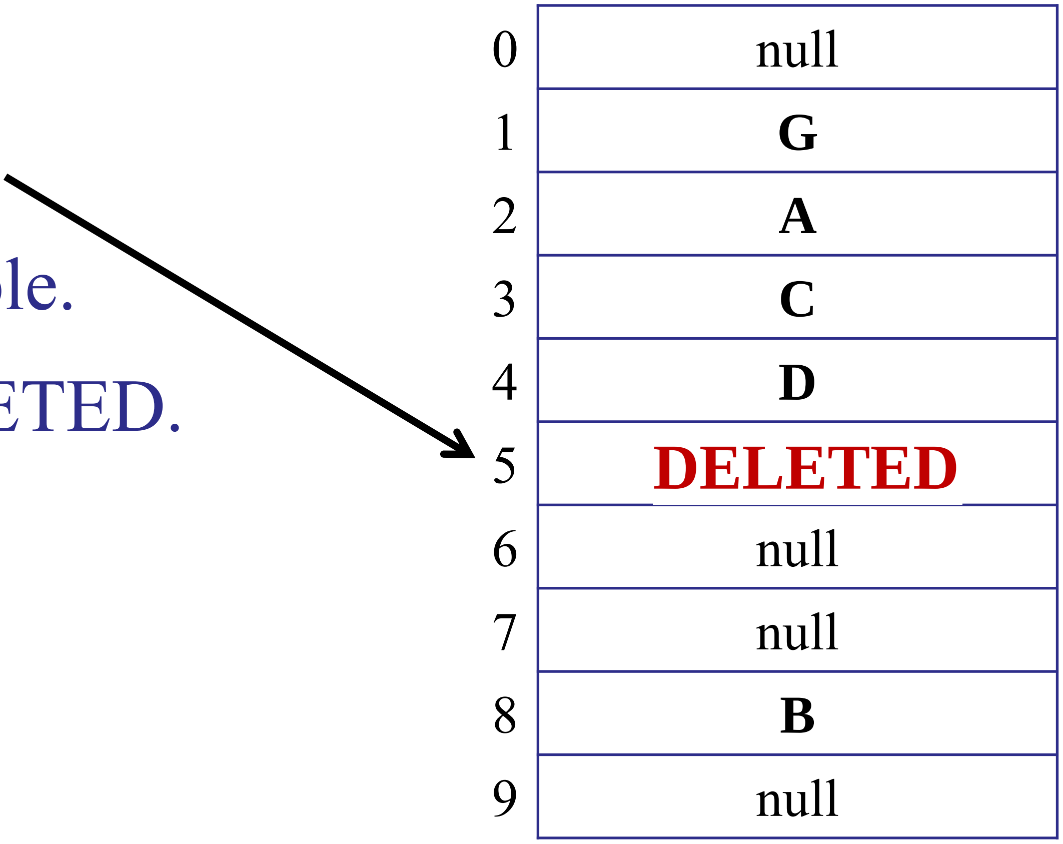
Hash Function re-defined:

h(key, i) : U → {1..m}

delete(key)

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

(Tombstone value.)



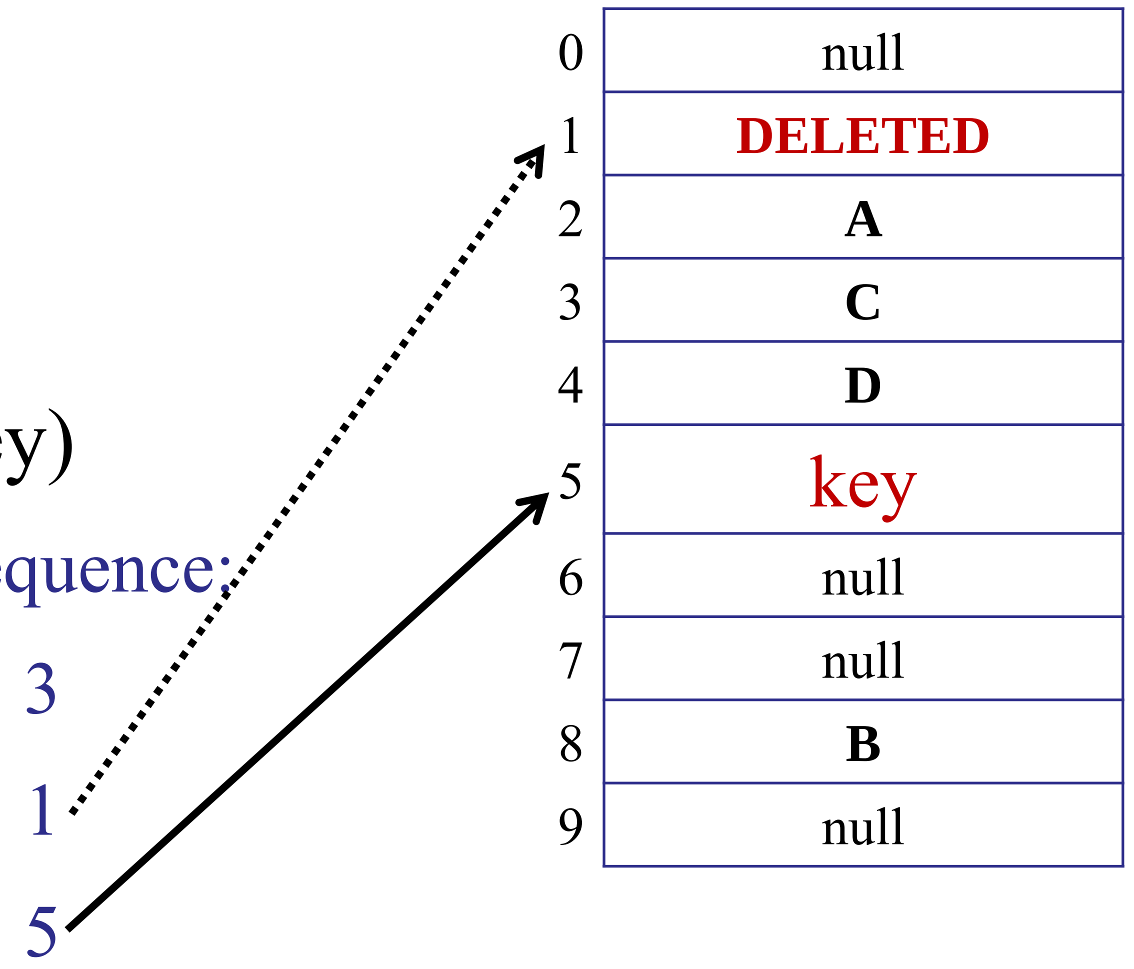
Open Addressing

insert(key)

delete(G)

search(key)

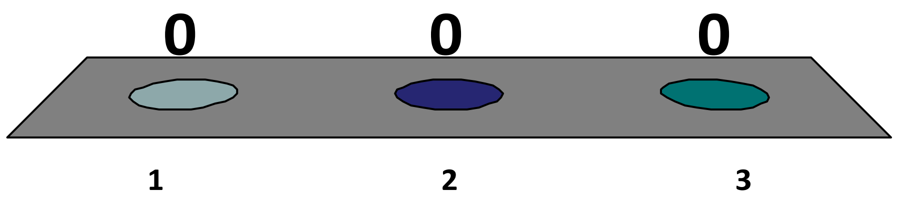
Probe sequence:



What happens when an insert finds a DELETED cell?

1. [✓ Overwrite the deleted cell.]
2. Continue probing.
3. Fail.

Response Counter



Hash Functions

Two properties of a good hash function:

1. h(*key*, *i*) enumerates all possible buckets.
   - For every bucket *j*, there is some *i* such that:

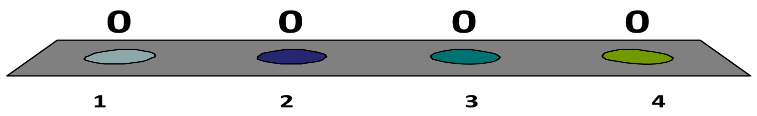
     $$h(key, i) = j$$

   - The hash function is permutation of $\{1..m\}$.
   - For linear probing: true!

What goes wrong if the sequence is not a permutation?

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

Response Counter



Hash Functions

Two properties of a good hash function:

2. Simple Uniform Hashing Assumption

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

For h(*key*, 1)?

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

Hash Functions

Two properties of a good hash function:

2. Uniform Hashing Assumption

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

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

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

Hash Functions

Two properties of a good hash function:

2. Uniform Hashing Assumption

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

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

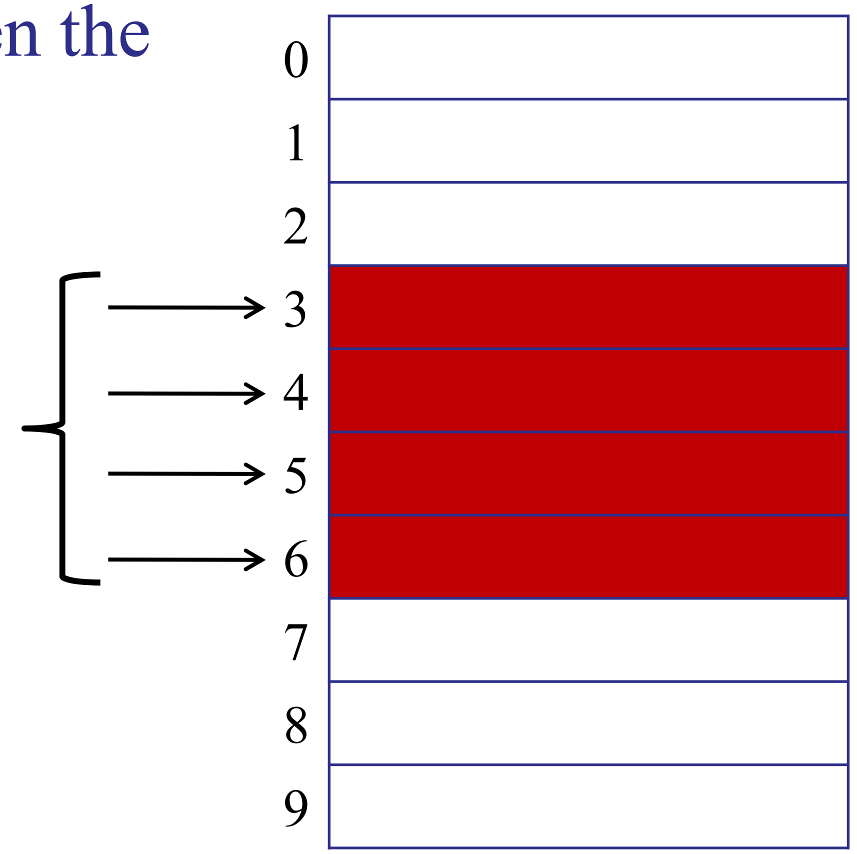
- 1 2 3 4 Pr(1/m)
- 1 2 4 3 Pr(0)
- 1 4 2 3 Pr(0)
- 1 4 3 2 Pr(0)
- …

NOT Linear Probing

Linear Probing

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."

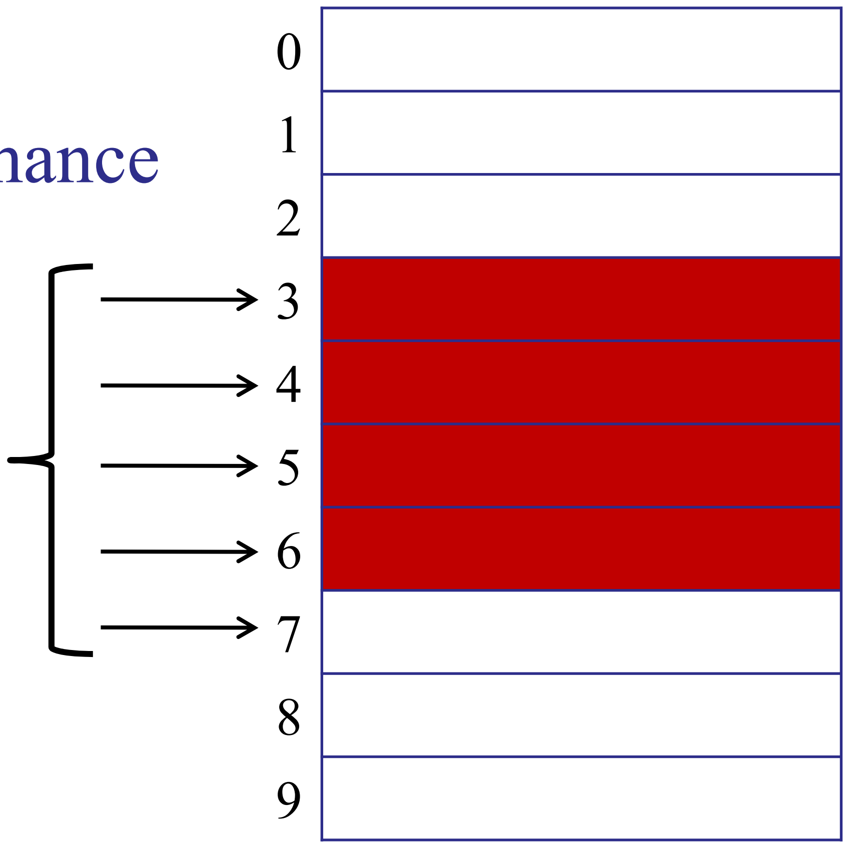
Linear Probing

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



if h(k,1) is any of these, the cluster will get bigger!

Linear probing

In practice, linear probing is very fast!

- Why? Caching!
- It is *cheap* to access nearby array cells.
  - Example: access T[17]
  - Cache loads: T[10..50]
  - Almost 0 cost to access T[18], T[19], T[20], …
- If the table is 1/4 full, then there will be clusters of size: $\theta(\log n)$
  - Cache may hold entire cluster!
  - No worse than wacky probe sequence.

Open Addressing

Properties of a good hash function:

2. Uniform Hashing Assumption

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

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

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

Double Hashing

- Start with two ordinary hash functions:

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

- Define new hash function:

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

- 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

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

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

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

$$f(k) + i\cdot g(k) = f(k) + j\cdot g(k) \quad \text{mod } m$$

$\Rightarrow$ $i\cdot g(k) = j\cdot g(k) \quad \text{mod } m$

$\Rightarrow$ $(i - j)\cdot g(k) = 0 \quad \text{mod } m$

$\Rightarrow$ 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.

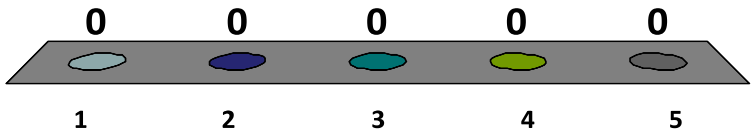
Example: if $(m = 2^r)$, then choose $g(k)$ odd.

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.

Response Counter



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 $\alpha = n / m$ $\leftarrow$ 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:

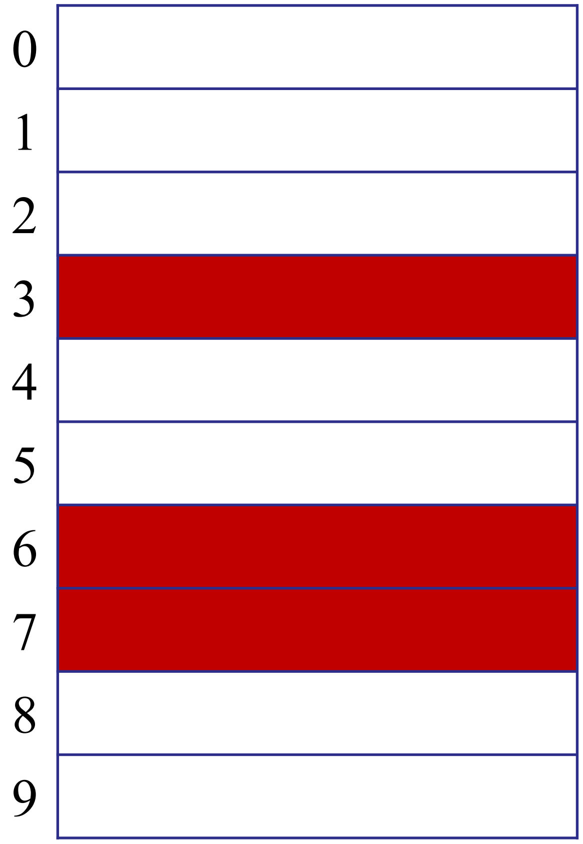
$$\le \frac{1}{1-\alpha}$$

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*

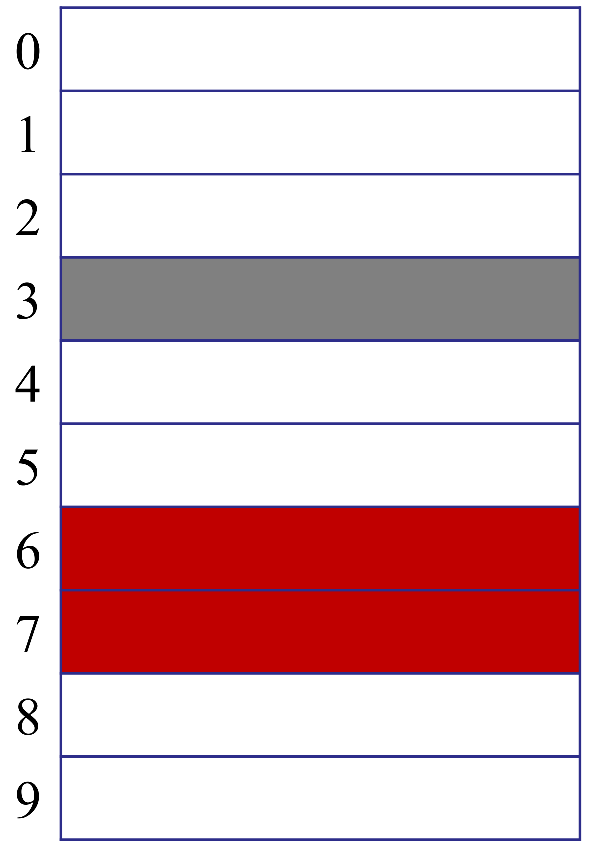


Performance of Open Addressing

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: $(n-1)/(m-1)$



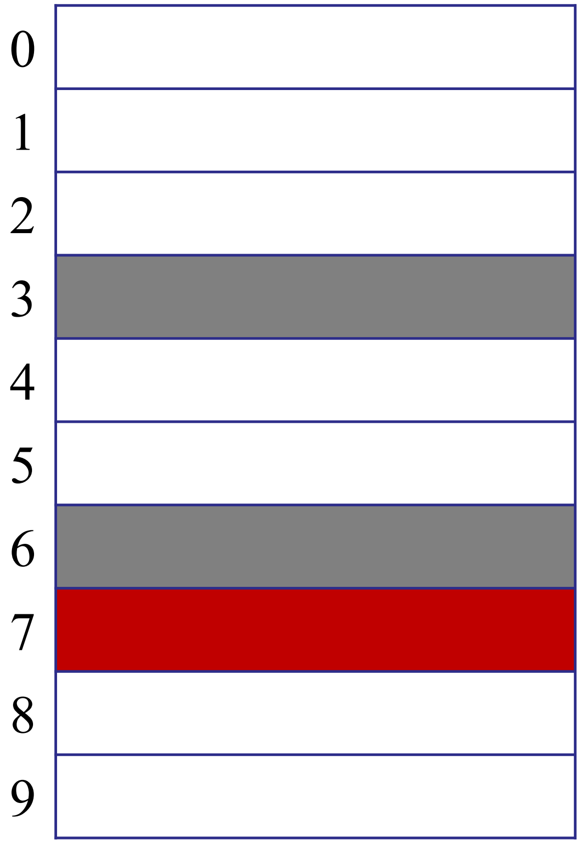
Performance of Open Addressing

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:

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

First probe

Probability of collision on first probe

Performance of Open Addressing

Proof of Claim:

- Expected cost:

$$\underbrace{1}_{\text{First probe}}+\underbrace{\frac{n}{m}}_{\substack{\text{Probability}\\\text{of collision}\\\text{on first probe}}}\left(1+\underbrace{\frac{n-1}{m-1}}_{\substack{\text{Probability}\\\text{of collision}\\\text{on second probe}}}\left(\boxed{\begin{array}{c}\text{Expected cost of}\\\text{remaining probes}\end{array}}\right)\right)$$

Performance of Open Addressing

Proof of Claim:

- Expected cost:

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

First probe   Second probe   Third probe

Performance of Open Addressing

Proof of Claim:

- Expected cost:

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

- Note:

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

Performance of Open Addressing

Proof of Claim:

- Expected cost:

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

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

Performance of Open Addressing

Proof of Claim:

- Expected cost:

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

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

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

Performance of Open Addressing

Proof of Claim:

- Expected cost:

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

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

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

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

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

Disadvantages…

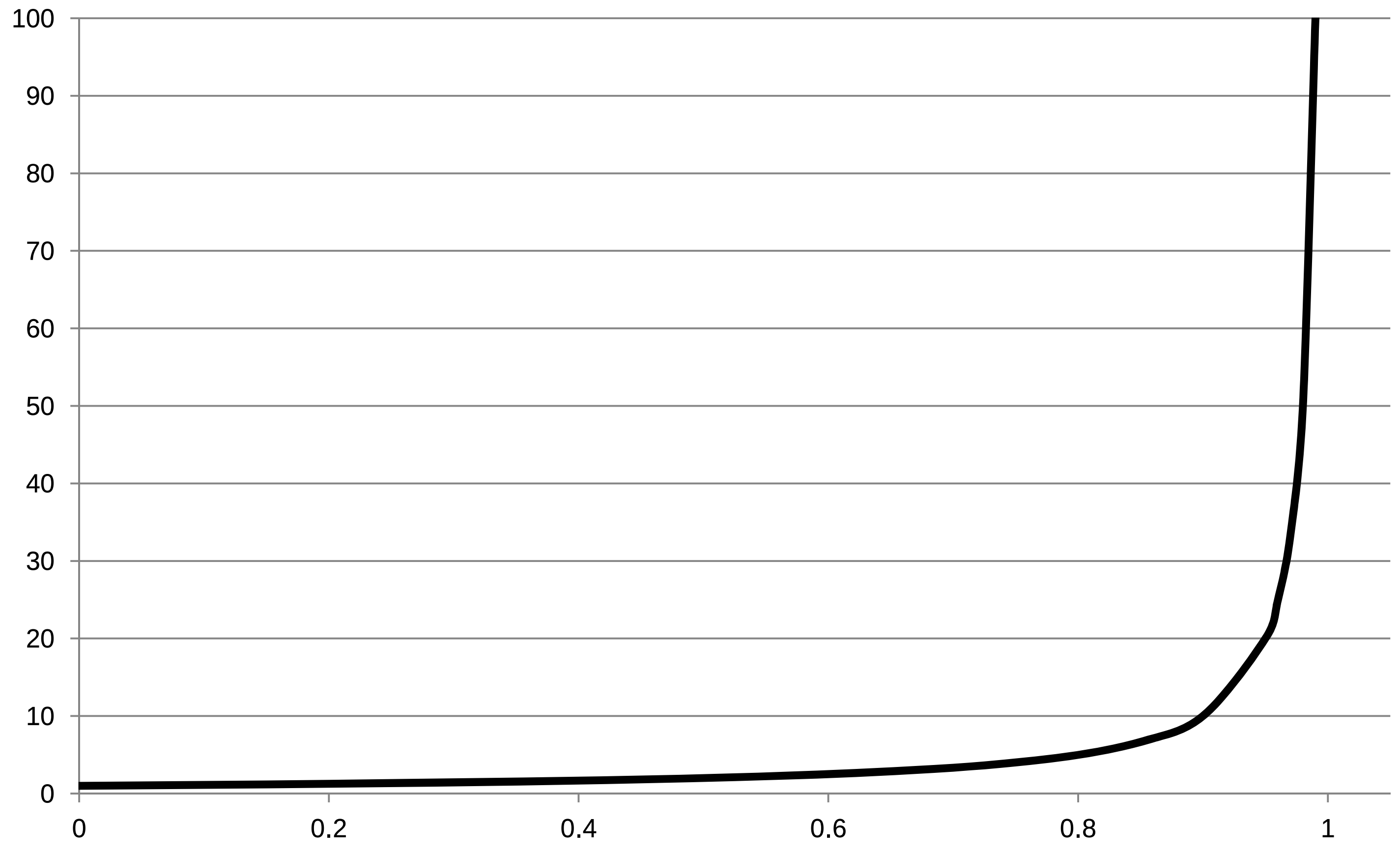
Open addressing:

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

Disadvantages…

Open addressing:

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



Today

- Java hashing
- Collision resolution: open addressing
- Table (re)sizing

Table Size

How large should the table be?

- Assume: Hashing with Chaining
- Assume: Simple Uniform Hashing
- Expected search time: $O(1 + n/m)$
- Optimal size: $m = \Theta(n)$
  - if $(m < 2n)$: too many collisions.
  - if $(m > 10n)$: too much wasted space.

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

Table Size

Idea:

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

Example:

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

Table Size

How to grow the table:

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

Table Size

Time complexity of growing the table:

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

Table Size

Time complexity of growing the table:

- Assume:
  - Size $m_1 < n$.
  - Size $m_2 > 2n$
- Costs:
  - Total: $O(m_1 + n)$.

    $= O(n)$

Table Size

Time complexity of growing the table:

Wait! What is the cost of initializing the new table?

- Initializing a table of size x takes x time!

- Costs:

  Total: $O(m_1 + m_2 + n)$

Table Size

Time complexity of growing the table:

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

How fast to grow?

Idea 1: Increment table size by 1

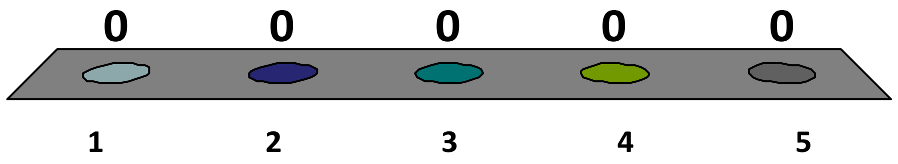
- if ($n == m$): $m = m+1$

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

Initially: $m = 8$
What is the cost of inserting n items?

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

Response Counter



How fast to grow?

Idea 1: Increment table size by 1

- When ($n == m$): $m = m+1$
- Cost of each resize: $O(n)$

Table size	8	8	9	10	11	12	…	n+1
Number of items	0	7	8	9	10	11	…	n
Number of inserts		7	1	1	1	1	…	1
Cost		7	8	9	10	11		n

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

How fast to grow?

Idea 2: Double table size

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

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

How fast to grow?

Idea 2: Double table size

- When ($n == m$): $m = 2m$
- Cost of each resize: $O(n)$

Table size	8	8	16	16	16	16	16	16	16	16	32	32	32	…	2n
# of items	0	7	8	9	10	11	12	13	14	15	16	17	18	…	n
# of inserts		7	1	1	1	1	1	1	1	1	1	1	1	…	1
Cost		7	8	1	1	1	1	1	1	1	16	1	1		n

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

How fast to grow

Idea 2: Double table size

Cost of Resizing:

Table size	Total Resizing Cost
8	8
16	(8 + 16)
32	(8 + 16 + 32)
64	(8 + 16 + 32 + 64)
128	(8 + 16 + 32 + 64 + 128)
…	…
m	<(1+2+4+8+…+m) ≤ O(m)

How fast to grow?

Idea 2: Double table size

- if ($n == m$): $m = 2m$
  - Cost of resize: O(n)
  - Cost of inserting n items + resizing: O(n)
- Most insertions: O(1)
- Some insertions: linear cost (expensive)
- Average cost: O(1)

How fast to grow?

Idea 3: Square table size

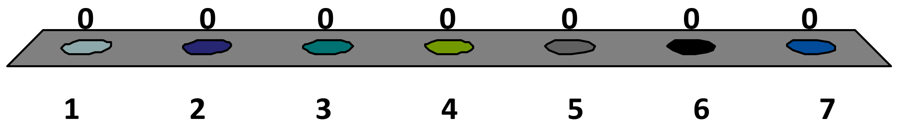
- When ($n == m$): $m = m^2$

Table size	Total Resizing Cost
8	?
64	?
4,096	?
16,777,216	?
…	…
m	?

Assume: square table size
What is the cost of inserting *n* items?

1. O(log n)
2. O($\sqrt{n}$)
3. O(n)
4. O(n log n)
5. O(n^2)
6. O(2^n)
7. None of the above.

Response Counter



How fast to grow?

Idea 3: Square table size

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

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

    $= O(n + n^2 + n)$

    $= O(n^2)$

How fast to grow?

Idea 3: Square table size

– When ($n == m$): $m = m^2$

# Items	Total Resizing Cost
8	64
64	(64 + 4,096)
4,096	(64 + 4,096 + …)
…	…
n	$> n^2$
	$< O(n^2)$

How fast to grow?

Idea 3: Square table size

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

# Items	Resizing Cost	Insert Cost
8	64	8
64	(64 + 4,096)	64
4,096	(64 + 4,096 + …)	4,096
…	…	…
n	$> n^2$	n
	$< O(n^2)$	O(n)

How fast to grow?

Idea 3: Square table size

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

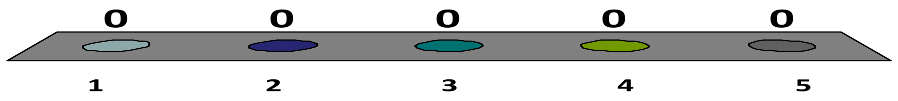
- Cost of resize:
  - Total: $O(n^2)$

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

Why else is squaring the table size bad?

1. Resize takes too long to find items to copy.
2. Inefficient space usage.
3. Searching is more expensive in a big table.
4. Inserting is more expensive in big table.
5. Deleting is more expensive in a big table.

Response Counter



Deleting Elements

Basic procedure: (chained hash tables)

Delete(*key*)

1. Calculate hash of *key*.
2. Let *L* be the linked list in the specified bucket.
3. Search for item in linked list *L*.
4. Delete item from linked list *L*.

Cost:

– Total: $O(1 + n/m)$

Deleting Elements

What happens if too many items are deleted?

- Table is too big!
- Shrink the table…

- Try 1:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/2)$ then $m = m/2$.

Deleting Elements

Rules for shrinking and growing:

- Try 1:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/2)$ then $m = m/2$.

- Example problem:
  - Start: n=100, m=200
  - Delete: n=99, m=200 → shrink to m=100
  - Insert: n=100, m=100 → grow to m=200
  - Repeat…

Deleting Elements

Example execution:

- Start: n=100, m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- cost=100 • Delete: n=99, m=200 → shrink to m=100
- cost=100 • Insert: n=100, m=100 → grow to m=200
- Repeat…

Deleting Elements

Rules for shrinking and growing:

- Try 2:
  - If $(n == m)$, then $m = 2m$.
  - If $(n < m/4)$, then $m = m/2$.

- Claim:
  - Every time you double a table of size m, at least $m/2$ new items were added.
  - Every time you shrink a table of size m, at least $m/4$ items were deleted.

Amortized Analysis

Technique for analyzing "average" cost:

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

Definition:

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

Amortized Analysis

Definition:

– Operation has amortized cost T(n) if for every integer k, the cost of k operations is $\leq k\,\mathrm{T}(n)$

Example: amortized cost = 7

1. insert: 5
2. insert: 5
3. insert: 5
4. insert: 13
5. insert: 7

5 <= 7

5+5 <= 2*7 = 14

5+5+5 <= 3*7 = 21

5+5+5+13 <= 4*7 = 28

5+5+5+13+7 <= 5*7 = 35

Amortized Analysis

"amortized" is NOT "average"

Definition:

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

Example: amortized cost **NOT** 7

1. insert: 13
2. insert: 5
3. insert: 5
4. insert: 5
5. insert: 7

13 > 7

13+5 > 2*7 = 14

13+5+5 > 3*7 = 21

13+5+5+5 <= 4*7 = 28

5+5+5+13+7 <= 5*7 = 35

Amortized Analysis

Definition:

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

Example: (Hash Tables)

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

– Conclusion:

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

Amortized Analysis

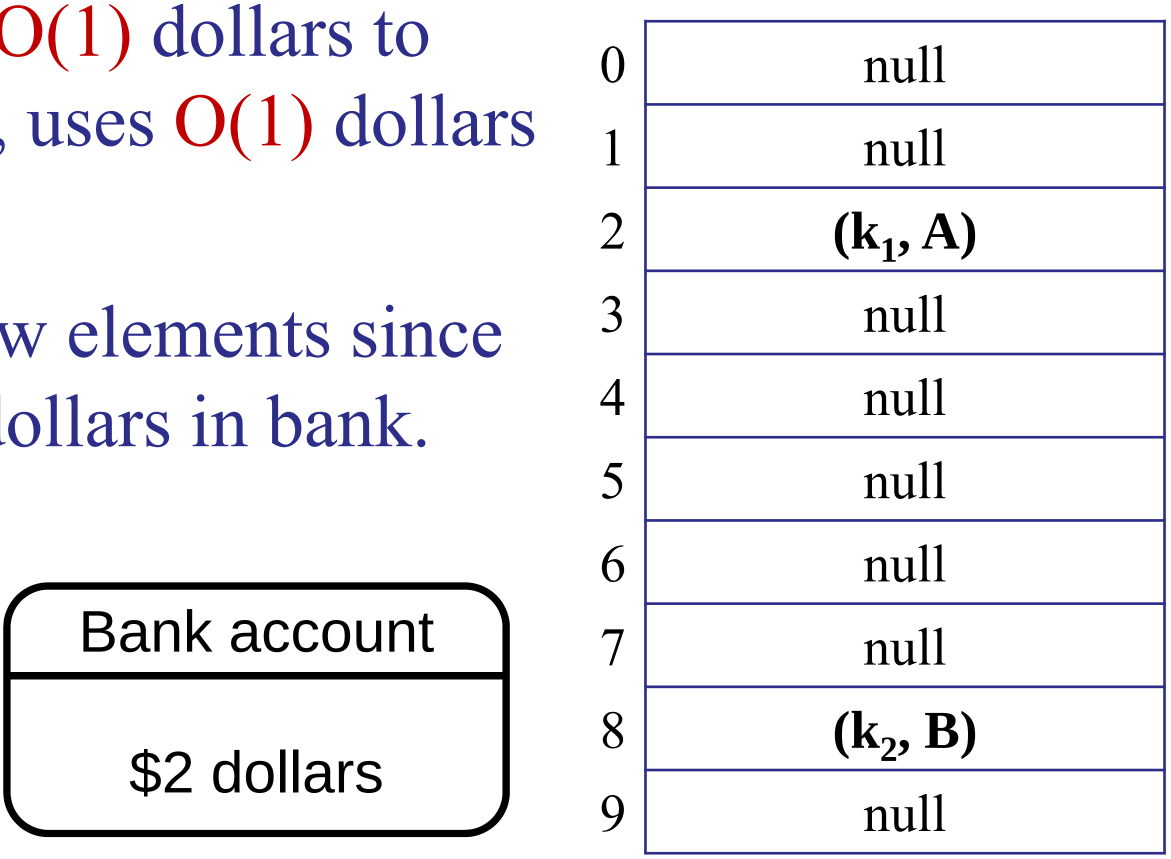
Accounting Method (paying rent)

- Imagine a bank account **B**.
- Each operation adds money to the bank account.
- Every step of the algorithm spends money:
  - Immediate money: to perform the operation.
  - Deferred money: from the bank account.
- Total cost execution = total money
  - Average time / operation = money / num. ops

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account, uses O(1) dollars to insert element.
- A table with *k* new elements since last resize has *k* dollars in bank.



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 Θ(*m*) dollars.

Amortized Analysis

Accounting Method Example (Hash Table)

- Each table has a bank account.
- Each time an element is added to the table, it adds O(1) dollars to the bank account.
- Pay for resizing from the bank account!
- Strategy:
  - Analyze inserts ignoring cost of resizing.
  - Ensure that bank account always is big enough to pay for resizing.

Amortized Analysis

Total cost: Inserting k elements costs:

- Deferred dollars: $O(k)$ (to pay for resizing)
- Immediate dollars: $O(k)$ for inserting elements in table
- Total (Deferred + Immediate): $O(k)$

Amortized Analysis

Total cost: Inserting *k* elements costs:

- Deferred dollars: O(*k*) (to pay for resizing)
- Immediate dollars: O(*k*) for inserting elements in table
- Total (Deferred + Immediate): O(*k*)

Cost per operation:

- Deferred dollars: O(1)
- Immediate dollars: O(1)
- Total: O(1) / per operation

Example: Binary Counter

Counter ADT:

- increment()
- read()

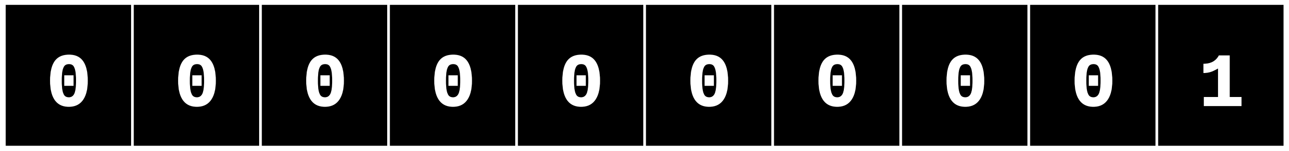
0	0	0	0	0	0	0	0	0	0

Example: Binary Counter

Counter ADT:

- increment()
- read()

increment()



Example: Binary Counter

Counter ADT:

- increment()
- read()

increment(), increment()

0	0	0	0	0	0	0	0	1	0

Example: Binary Counter

Counter ADT:

- increment()
- read()

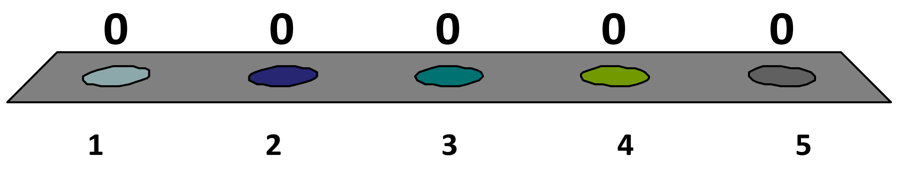
increment(), increment(), increment()

0	0	0	0	0	0	0	0	1	1

What is the worst-case cost of incrementing a counter with max-value n?

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

Response Counter

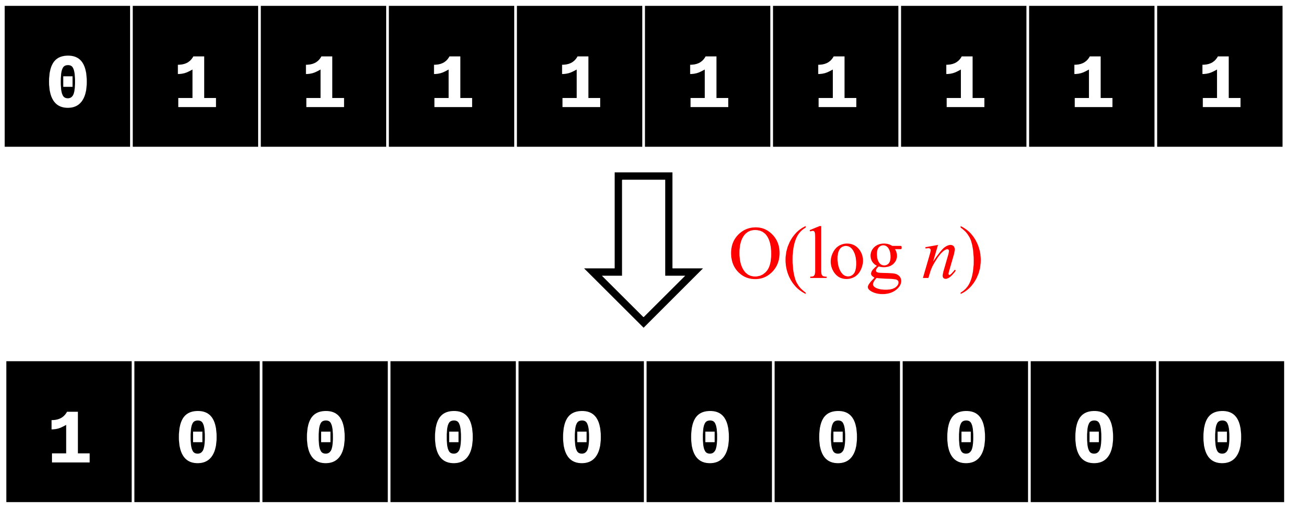


Example: Binary Counter

Counter ADT:

- increment()
- read()

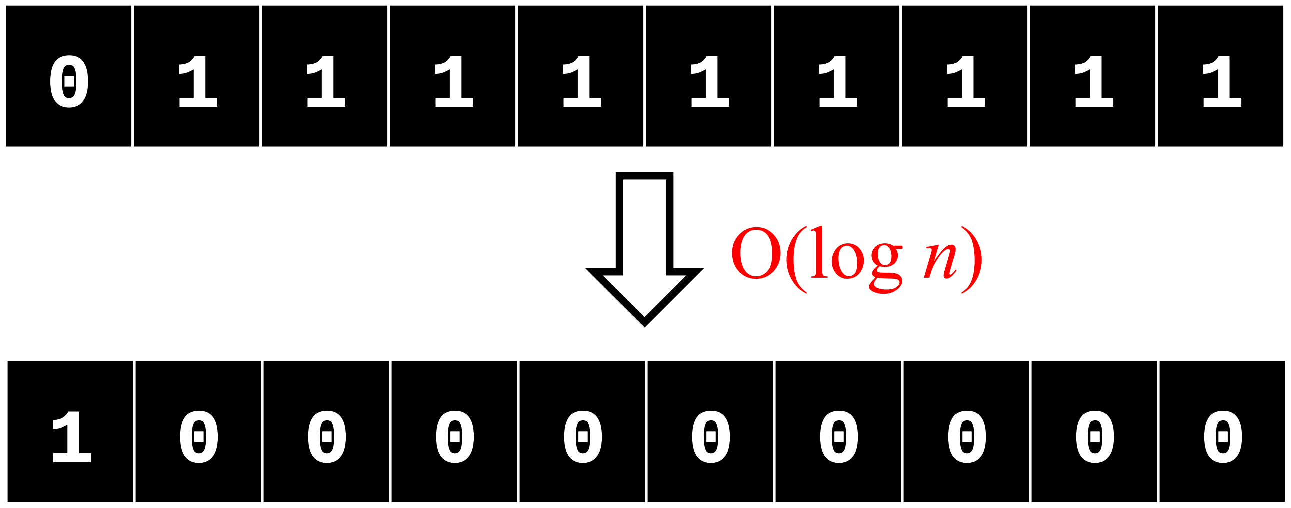
Some increments are expensive…



Example: Binary Counter

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

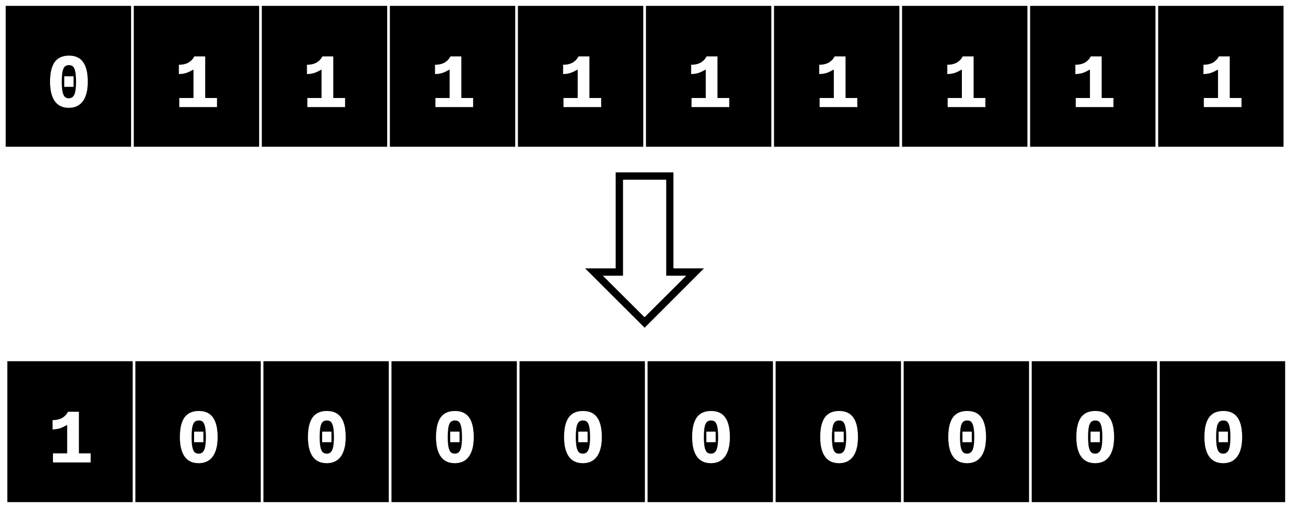
- Easy answer: O(log n)
- More careful analysis….



Example: Binary Counter

Observation:

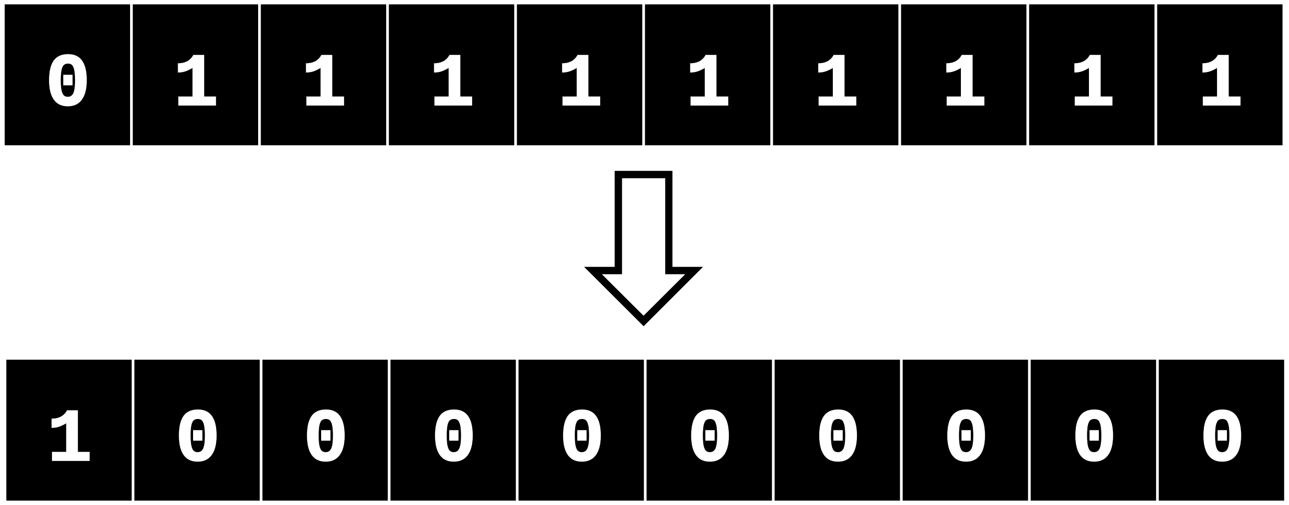
During each increment, only one bit is changed from: 0 → 1



Example: Binary Counter

Observation:

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

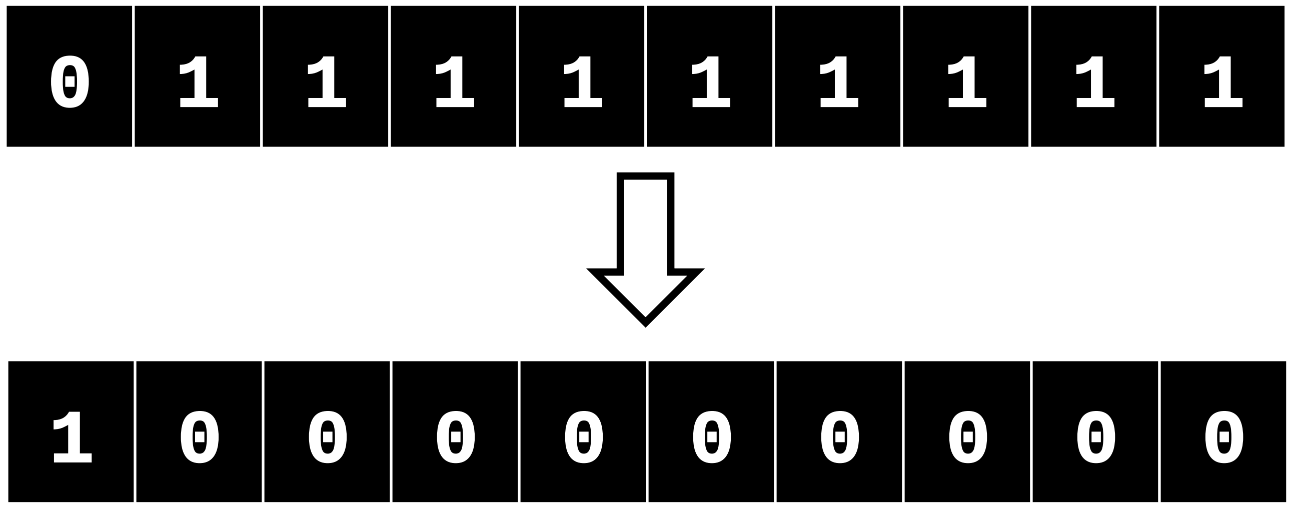


Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

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



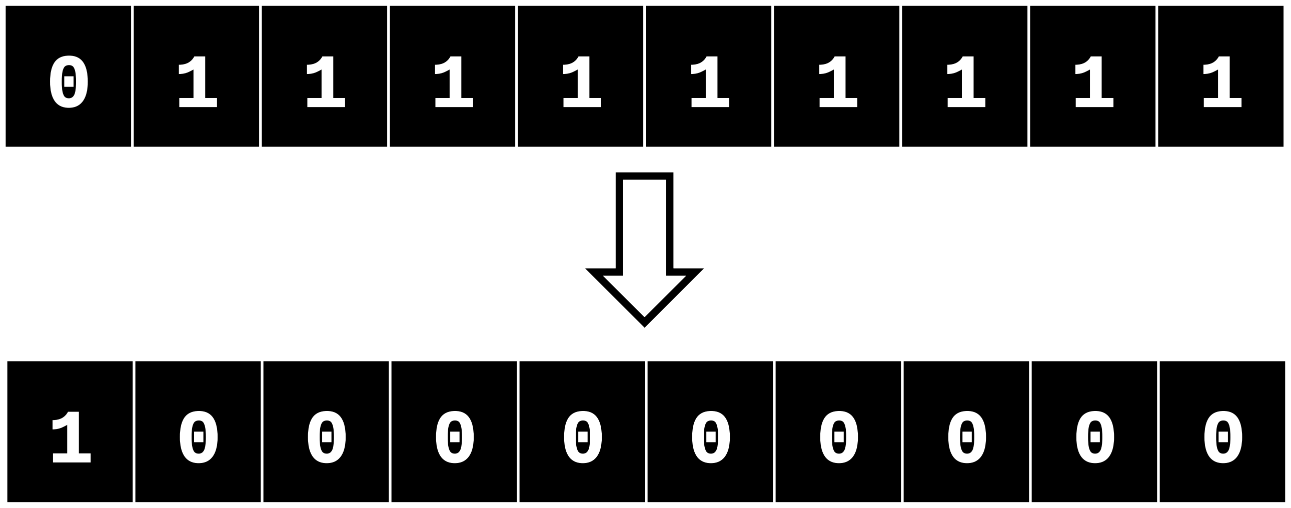
Example: Binary Counter

Observation:

Accounting method: each bit has a bank account.

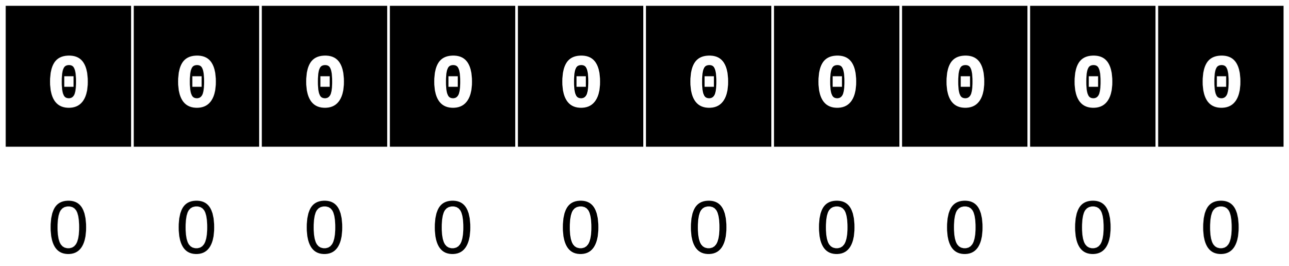
Whenever you change it from 0→1, add one dollar.

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



Example: Binary Counter

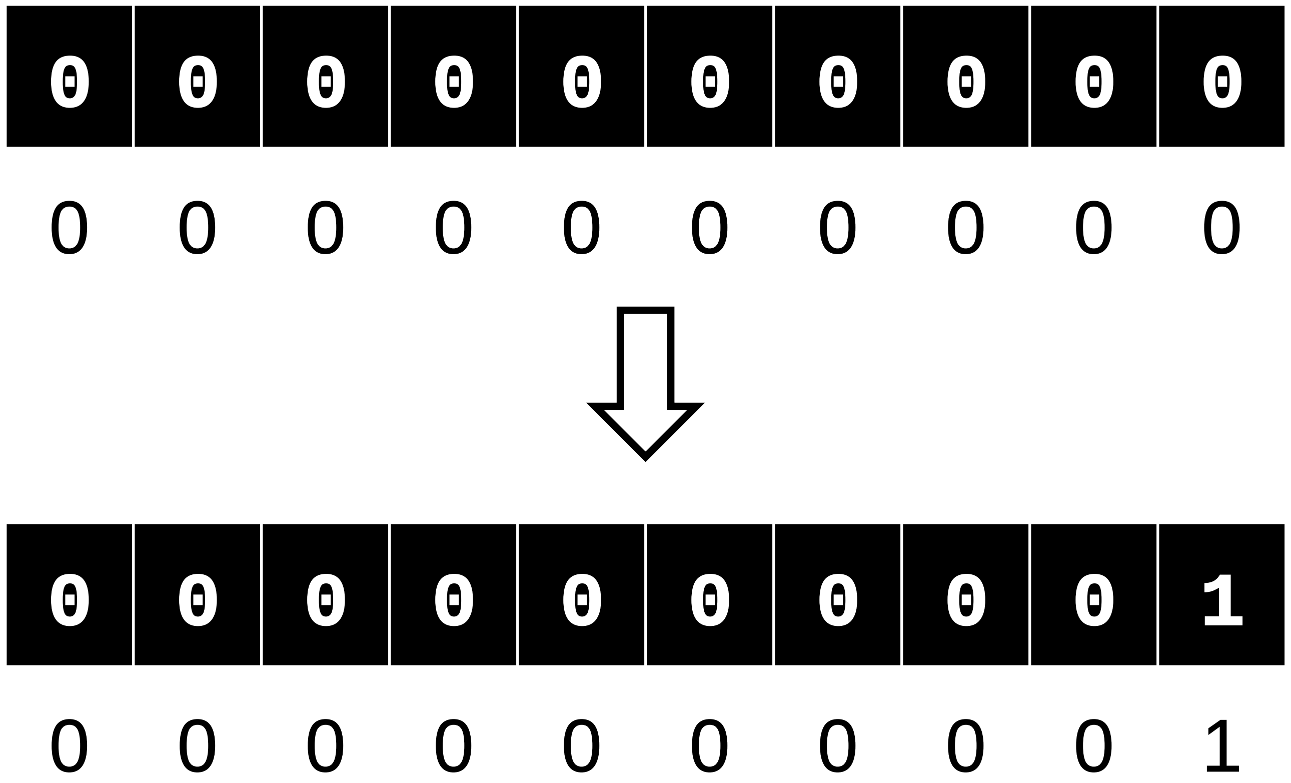
Counter ADT



Example: Binary Counter

Counter ADT

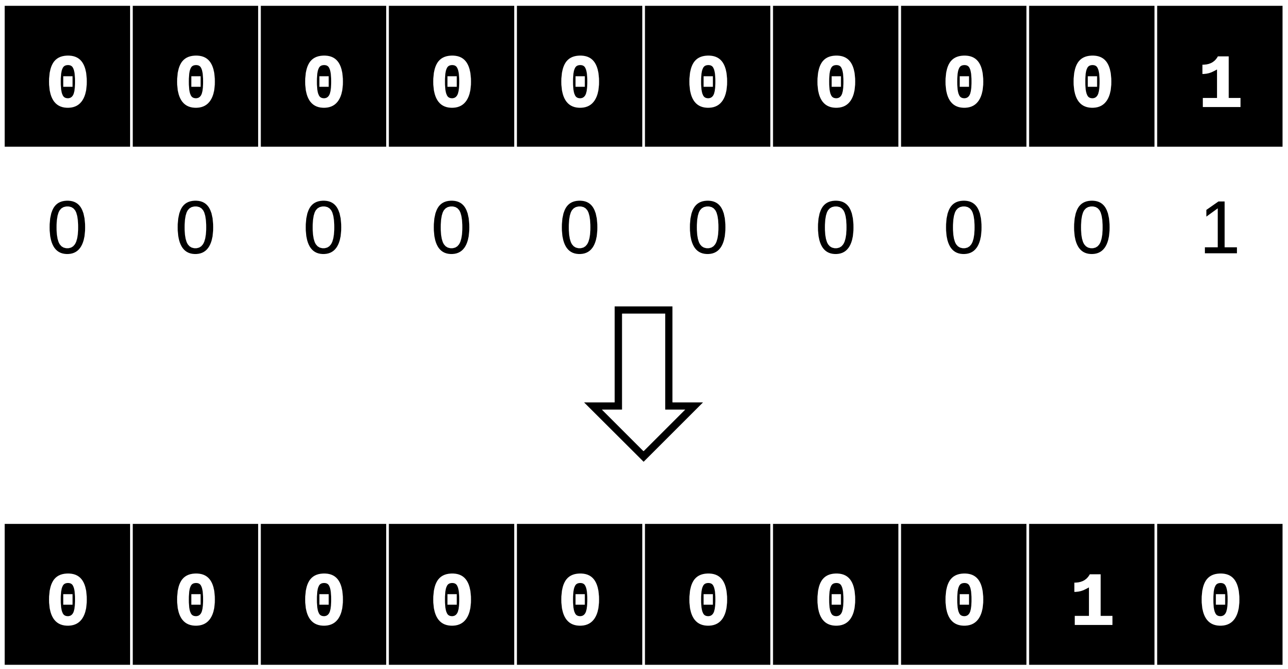
increment()



Example: Binary Counter

Counter ADT

increment(), increment()



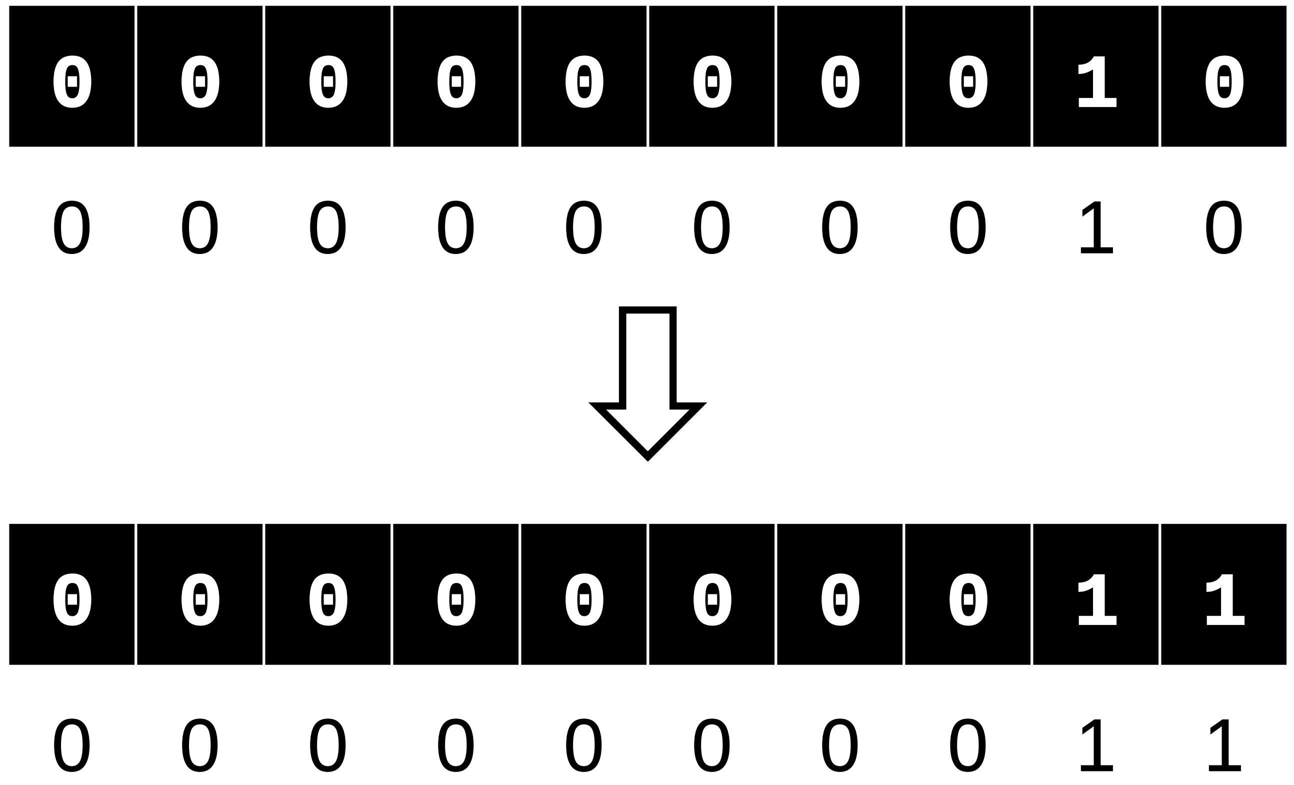
0 0 0 0 0 0 0 0 0 1

0 0 0 0 0 0 0 0 1 0

Example: Binary Counter

Counter ADT

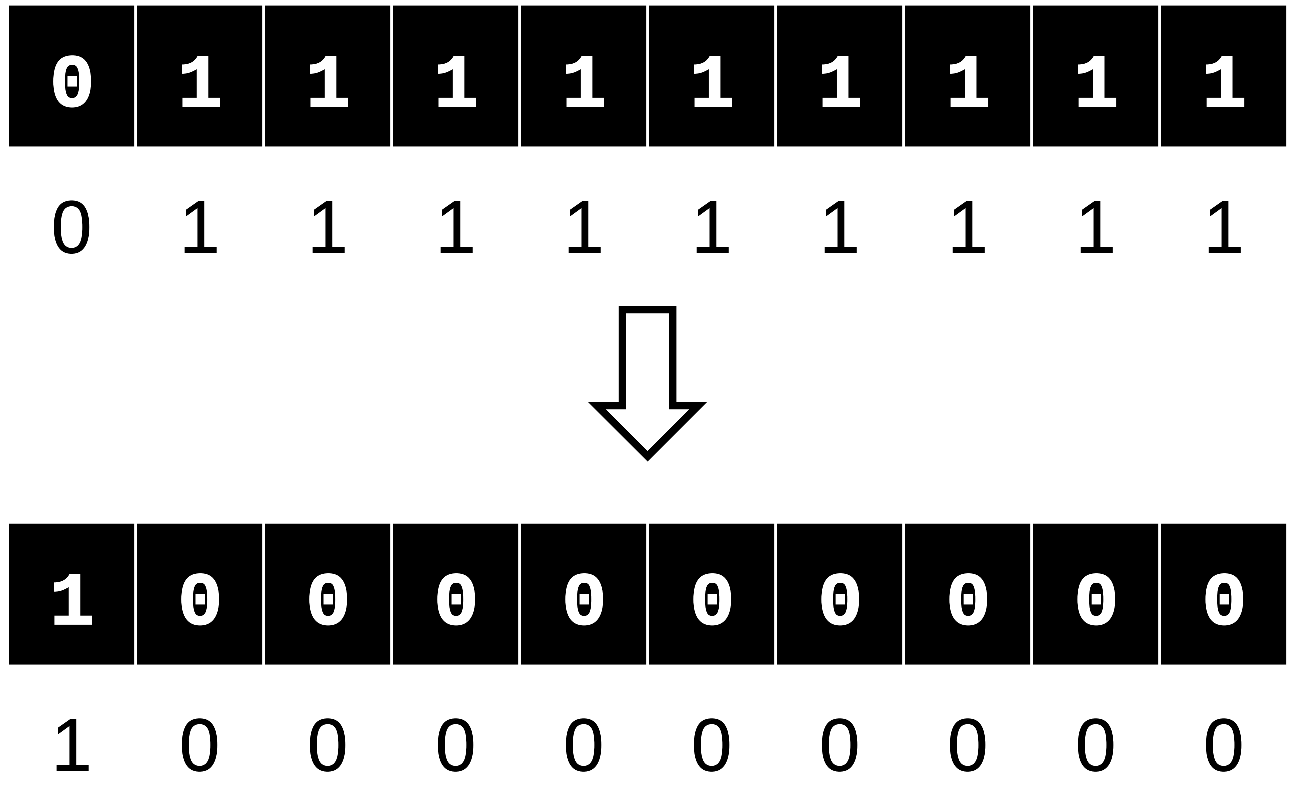
increment(), increment(), increment()



Example: Binary Counter

Counter ADT

increment()



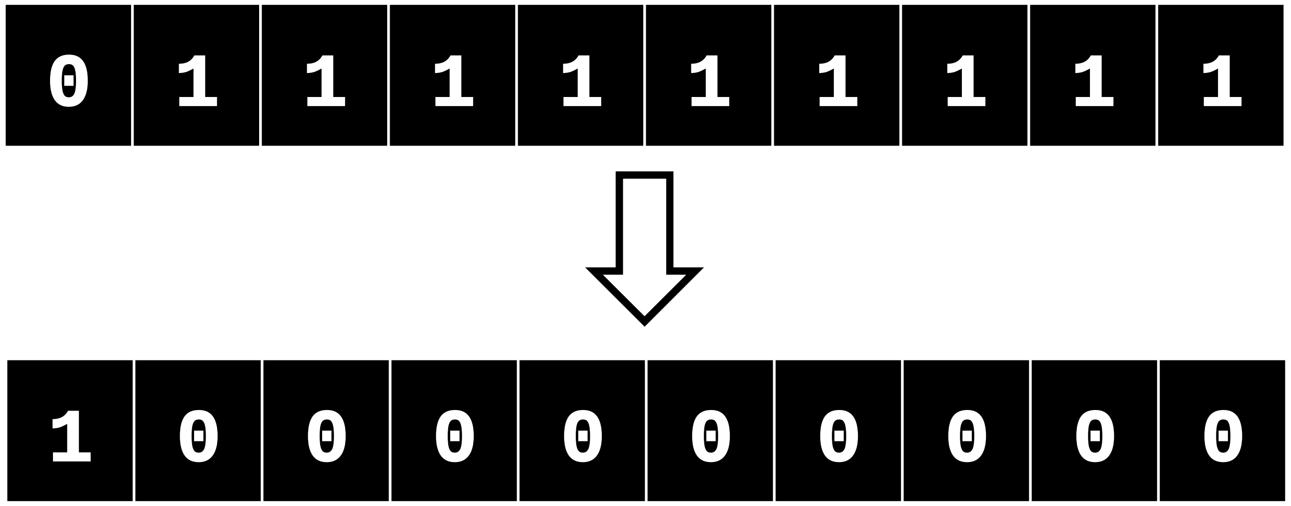
Example: Binary Counter

Observation:

Amortized cost of increment: 2

- One operation to switch <u>one</u> 0→1
- One dollar (for bank account of switched bit).

(All switches from 1→0 paid for by bank account.)



Today

- Java hashing
- Resolving collisions: open addressing
- Table (re)sizing

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:

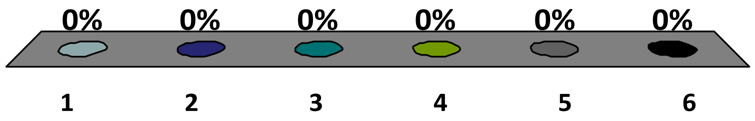
- Operate directly on keys (i.e., indexing)
- Gave up: successor/predecessor/etc.

Symbol Tables 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?



Response Counter

Symbol Tables 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.

Symbol Tables are Useful

Example 3: DNA Analysis

– Longest common substring (text):

ALGORITHM vs. ARITHMETIC

Symbol Tables are Useful

Naïve Algorithm: strings *A* and *B*

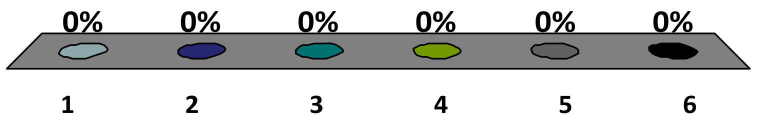
```
L = length(A);
for (L = n down to 1)
    for every substring X1 of A of length L:
        for every substring X2 of B of length L:
            if (X1==X2) then return X1;
```

Example: ALGORITHM   ARITHMETIC

– L=3 : X1= ALG → compare to ARI, RIT, ITH, …

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)



Symbol Tables 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-L substrings
        for every substring X2 of B of length L:       n-L substrings
            if (X1==X2) then return X1;                comparison costs: O(L)
```

Total cost: $O(n^4)$

Symbol Tables are Useful

Example 3: DNA Analysis

- Longest 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.

Symbol Tables 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
```

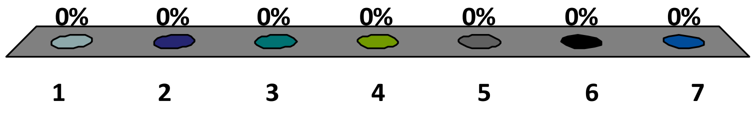
Symbol Tables are Useful

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

```
low = 0;
high = length(A);
repeat until (low >= high-1):
    L = low + (high-low)/2;
    found = **substring**(A, B, L);
    if (found) then low = L;
    else high = L;
return low;
```

What is the running time?

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



Symbol Tables are Useful

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

```
substring(A, B, L)
    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 true;           ← comparison costs: O(n)
    return false
```

Cost: $O(n^3)$

Symbol Tables are Useful

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

```
low = 0
high = length(A)/2;
repeat until (low >= high-1):
    L = (high+low)/2;
    found = **substring**(A, B, L);
    if (found) then low = L;
    else high = L;
return low;
```

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

Symbol Tables are Useful

Example 3: DNA Analysis

– Longest common substring (text):

  ALGORITHM vs. ARITHMETIC

– Another idea:

  - Put every substring from first string into a symbol table.
  - Lookup every substring from second string in the symbol table.

Symbol Tables are Useful

Example 3: DNA Analysis

– Longest common substring (text):

  ALGORITHM vs. ARITHMETIC

– Add to symbol table:

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

Symbol Tables are Useful

Example 3: DNA Analysis

- Longest common substring (text):

  ALGORITHM vs. ARITHMETIC

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

Symbol Tables are Useful

Example 3: DNA Analysis

- Longest common substring (text):

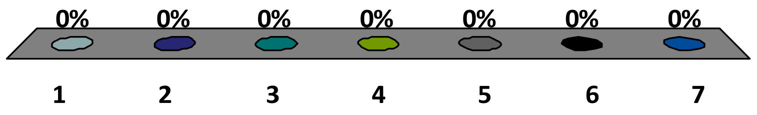
  ALGORITHM vs. ARITHMETIC

- Search in symbol table:
  - 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 a properly sized hash table. 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^3log n)

Response Counter



Symbol Tables are Useful

Search for substring of length L:

```
substring(A, B, L)
    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 true;           comparison costs: O(n)
    return false
```

Cost: $O(n^3)$

Symbol Tables 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 symbol table, you have to at least read the whole string!
- Total running time: O(n^3)

Symbol Tables are Useful

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

```
substring(A, B, L)
    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 true;           ← comparison costs: O(n)
    return false
```

n substrings

n substrings

comparison costs: *O(n)*

Cost: $O(n^3)$

Symbol Tables are Useful

Example 3: DNA Analysis

- Longest common substring (text):

  ALGORITHM vs. ARITHMETIC

- Basic idea:
  - Put every substring from first string into a symbol table.
  - Lookup every substring from second string in the symbol table.

Symbol Tables are Useful

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

```
substring(A, B, L)
    for every substring X1 of A of length L:
            Add X1 to the symbol table.
    for every substring X2 of B of length L:
            if X2 is in the symbol table then return true;
    return false;
```

Cost: $O(Ln) = O(n^2)$

Symbol Tables are Useful

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

```
low = 0
high = length(A)/2;
repeat until (low >= high-1):
    L = (high+low)/2;
    found = **substring**(A, B, L);
    if (found) then low = L;
    else high = L;
return low;
```

Cost: $O(n^2 \log n)$

Symbol Tables are Useful

Example 3: DNA Analysis

- Longest 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).
  - Better hashing: O(n log n).

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
```

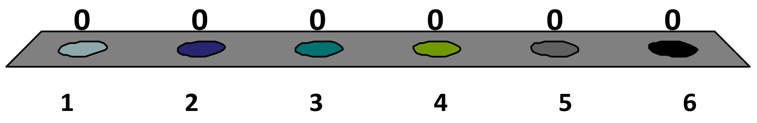
The performance of

`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. $O(n^2\log(n))$
6. $O(n^3)$

Response Counter



Longest Common Substring

```
exists-substring(X1, X2, L)
    1. for ($i = 0$ to $n - \mathrm{L} - 1$) do:
    2.         $hash = \mathrm{h}(\mathrm{X1}[i : i + \mathrm{L}])$
    3.         T.hash-insert($hash$, $i$))
    4. …
```

Loop $n - L$ times.

Calculate hash: O(L).

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)

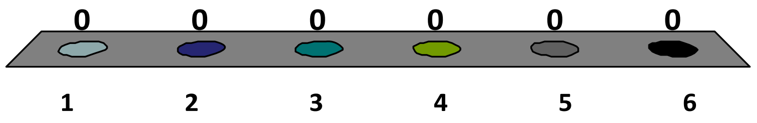
- Case 1: string s is in table: O(L)
- Case 2: Table[$h_1(s)$] = null: O(1)
- Case 3: Table[$h_1(s)$] != null: ??

```
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.

Response Counter

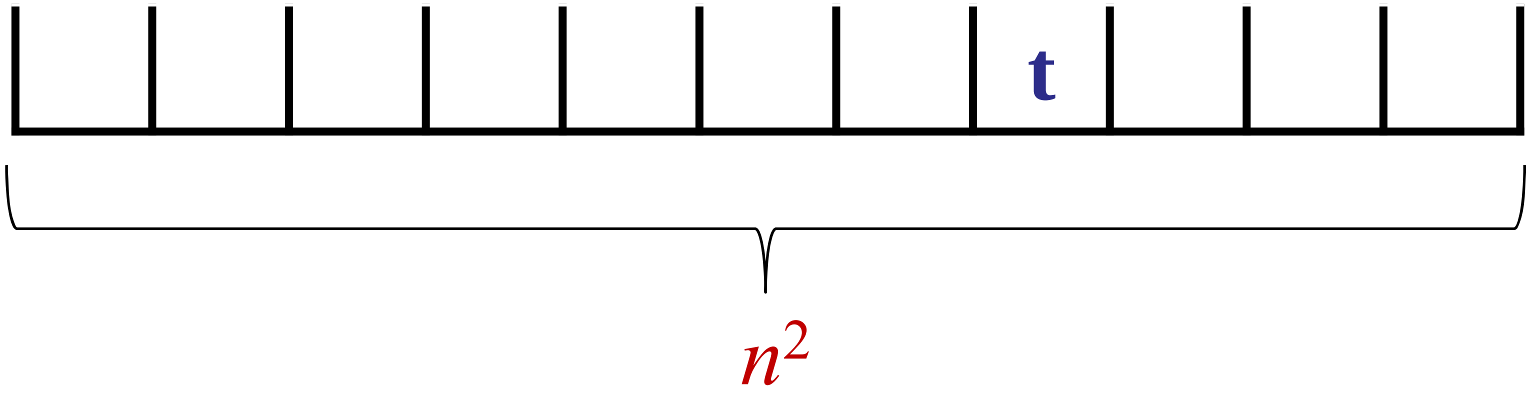


Faster substring matching

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

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

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



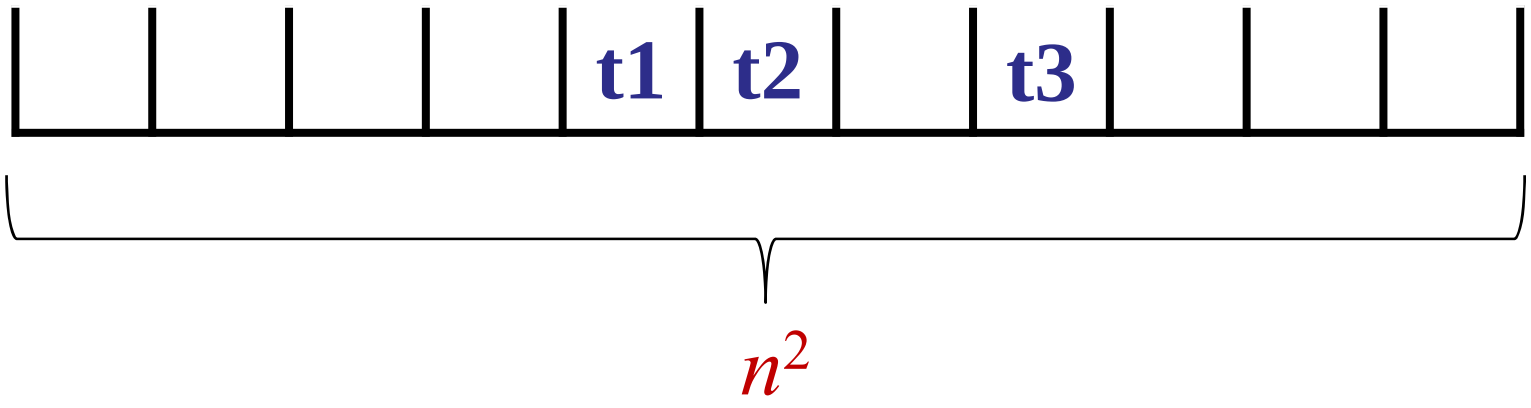
Faster substring matching

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.          hash = h(X2[i : i + L])
4.          if (T.hash-lookup(hash , s)) then
5.                  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() $\rightarrow$ 17
- delete-first-letter()

  string == “rith”
- hash() $\rightarrow$ 47
- append-letter(‘m’)

  string == “rithm”
- hash() $\rightarrow$ 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:

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

- 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}:

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

Rolling Hash

Step 2: Updating the string

Deleting character c_{L-1}:

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

$$s = s - c_{L-1} \cdot 2^{8(L-1)}$$

$$= s - c_{L-1} \ll 8(L-1)$$

Multiplication: O(1)

Shift: O(1)

Subtraction: O(1)

Rolling Hash

Step 2: Updating the string

Appending character c:

$$\begin{array}{lr}
s = & 00000000\ \ 10110111\ \ \dots\ 10010000\ \ 10010000 \\
\quad * & 1\ \ 00000000 \\
\hline
 & 10110111\ \ \dots\ 10010000\ \ 10010000\ \ 00000000
\end{array}$$

Rolling Hash

Step 2: Updating the string

Appending character c:

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

Rolling Hash

Step 2: Updating the string

Appending character c:

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

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

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)

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

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

Review

Symbol Table Abstract Data Type

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

Typical Implementations:

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

Review

Applications of Symbol Tables:

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

Symbol Tables in Java:

- HashMap<keyType, dataType>