Your hash bucket was tasty

CSE 250
Lecture 29

Hash Tables

The University at Buffalo, SUNY

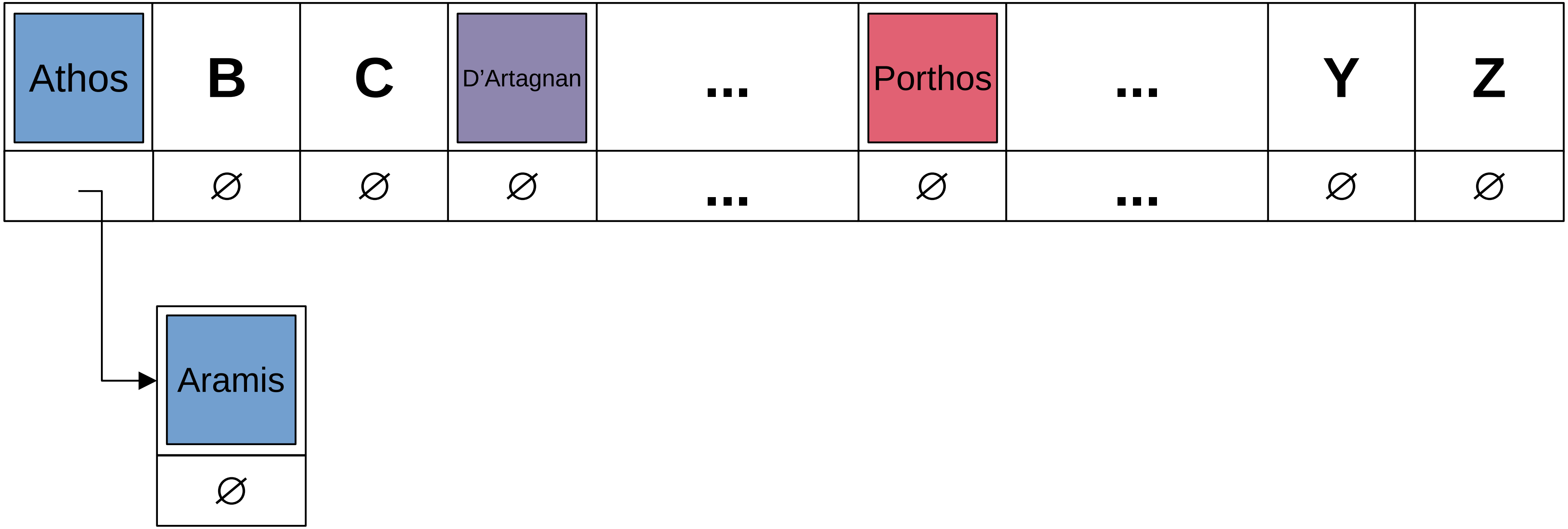
Alternative Idea: Assign Buckets

- Pros
  - O(1) Insert
  - O(1) Find
  - O(1) Remove
- Cons
  - Wasted Space (Only 3/26 slots used)
  - Duplication (What about Aramis?)

Bucket-Based Organization

- Wasted Space
  - Not ideal, but not wrong
  - O(1) access time might be worth it!
  - Also depends on choice of function (more on this later)
- Duplication
  - We need to deal with duplicates!

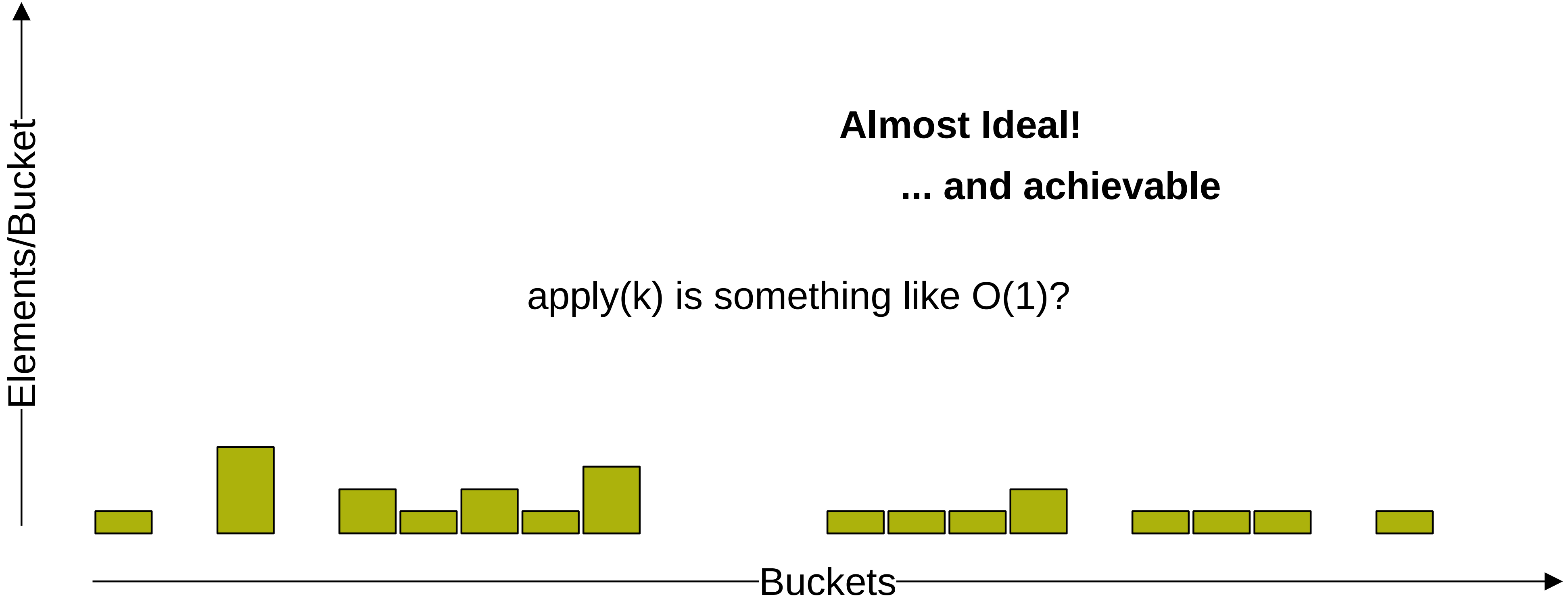
Buckets + Linked Lists



Picking a Lookup Function

- Desirable Features for h(x)
  - Fast
    - needs to be O(1)
  - “Unique”
    - As few duplicate bins as possible

Picking a Lookup Function



Picking a Lookup Function

- **Wacky Idea**: Have h(x) return a random value in [0, N)
  - Random.nextInt % N

    (Yes, it makes apply impossible, but bear with me)

Hash Functions

- Examples
  - SHA256 ← used by GIT
  - MD5, BCRYPT ← used by unix login, apt
  - MurmurHash3 ← used by Scala
- hash(x) is pseudorandom
  1) hash(x) ~ uniform random value in [0, INT_MAX)
  2) hash(x) always returns the same value
  3) hash(x) uncorrelated with hash(y) for x ≠ y

hash(x) is <u>deterministic</u>, but <u>statistically random</u>

Hash Functions

- **Not-so-Wacky Idea**: Use hash function to pick bucket
  - h(x) = hash(x) % N
    - Pseudorandom (“evenly distributed” over N)
    - Deterministic (same value every time)

Expectation

- X is a random variable
  - X = 1 with p = 0.2
  - X = 2 with p = 0.7
  - X = 3 with p = 0.1
- E[X] is the "expectation of X"
  - The average of X taken over all possibilities (weighted by p)
  - E[X] = (1 x 0.2) + (2 x 0.7) + (3 x 0.1)
    - = 0.2 + 1.4 + 0.3 = 1.9

Expected Size of a Bucket

- After n insertions, how many records can we “expect” in the average bucket?
- Let X_j be the number of records in bucket j
  - After n insertions, $0 \leq X_j \leq n$
    - $Xj = 0$ with $p = ???$
    - $Xj = 1$ with $p = ???$
    - ...
    - $Xj = n$ with $p = ???$

what is p?

Expected Size of a Bucket

- Assume N buckets
- Start with 1 insertion (n = 1)
  - $X_j = 0$ with $p = {}^{(N-1)}/_{N}$
  - $X_j = 1$ with $p = {}^{1}/_{N}$
- $E[X] = (0 \times {}^{(N-1)}/_{N}) + (1 \times {}^{1}/_{N}) = {}^{1}/_{N}$

Expected Size of a Bucket

- For n insertions, we repeat the process (n X_js)
  - $X_{1,j}$, $X_{2,j}$, …, $X_{n,j}$
- $E[\ \Sigma_i X_{i,j}\ ] = E[\ X_{1,j}\ ] + \ldots + E[\ X_{n,j}\ ]$
  - $= {}^{1}/_{N} + {}^{1}/_{N} + \ldots + {}^{1}/_{N}$
  - $= {}^{n}/_{N}$
- The <u>expected</u> runtime of insert, apply, remove is $O({}^{n}/_{N})$
- The <u>worst-case</u> runtime of insert, apply, remove is O(n)

Using Hash Functions

- hash(x: Int): Int
  - What about strings?

Arbitrary starting constant
(hash(“”))

```
def hashString(str: String): Int = {
  var accumulator: Int = SEED
  for(character <- str){
    accumulator = hash(accumulator * character.toInt)
  }
  return accumulator
}
```

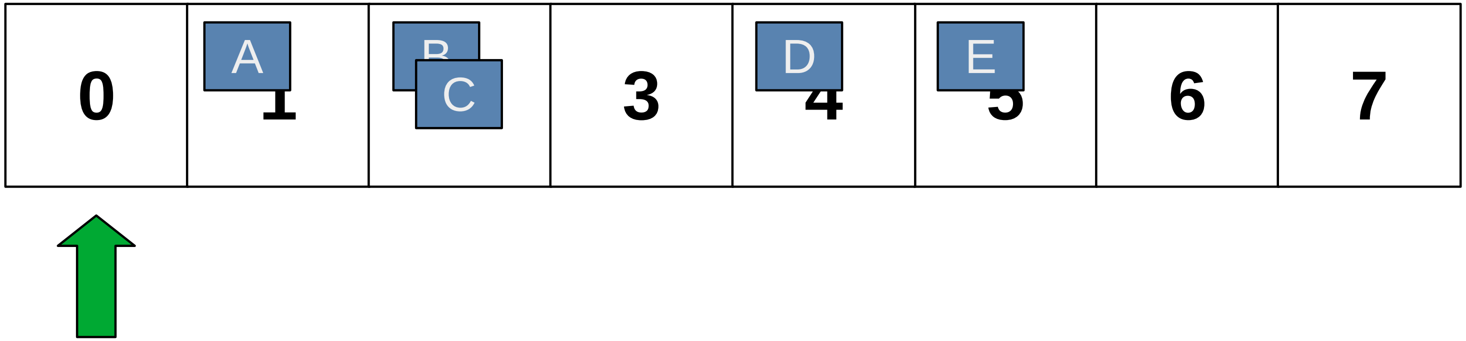
call hash() str.length times

(simplified, don’t actually do exactly this)

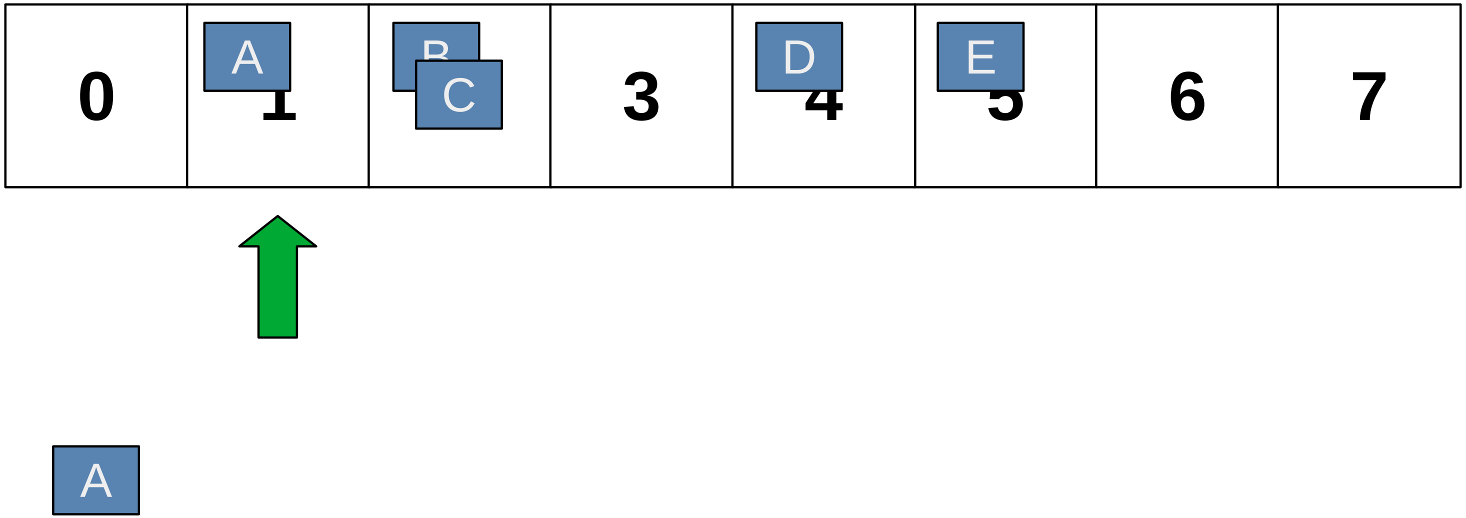
Hash Functions

- hash(x: Object): Int
  - In Java/Scala, call x.hashCode

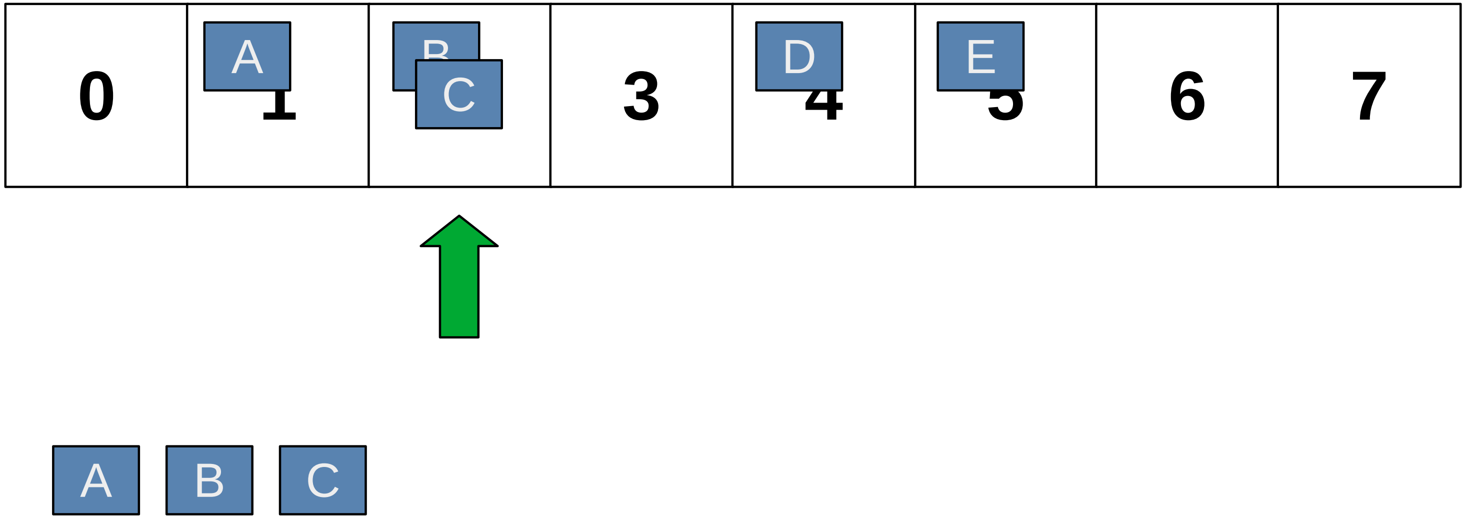
Iterating over a hash table



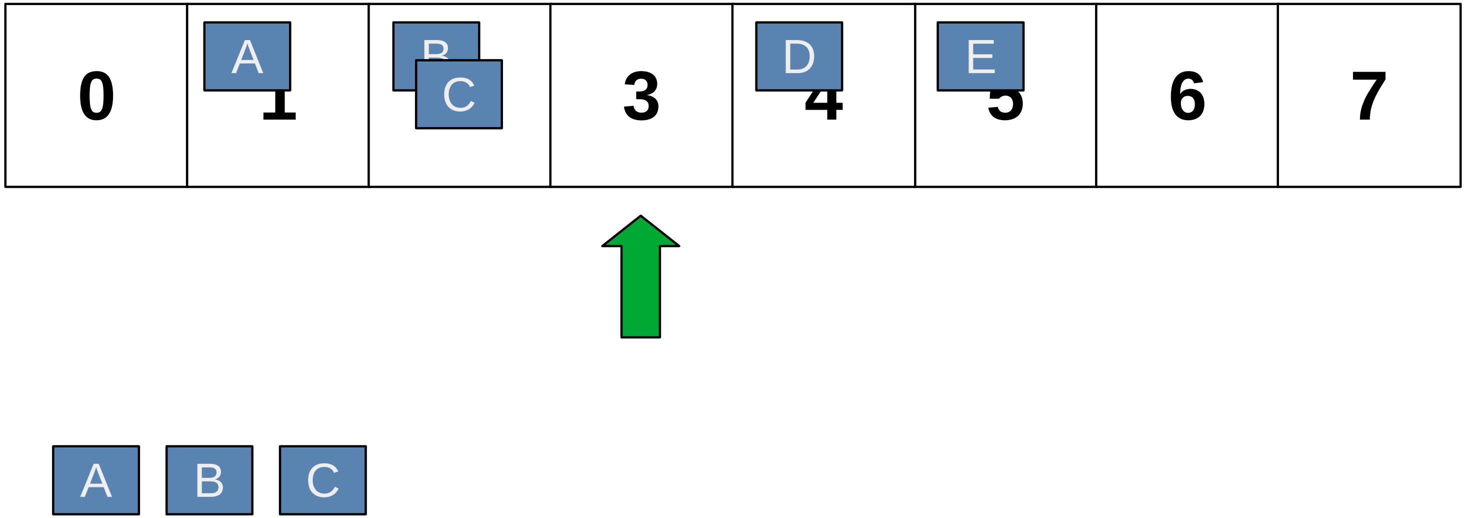
Iterating over a hash table



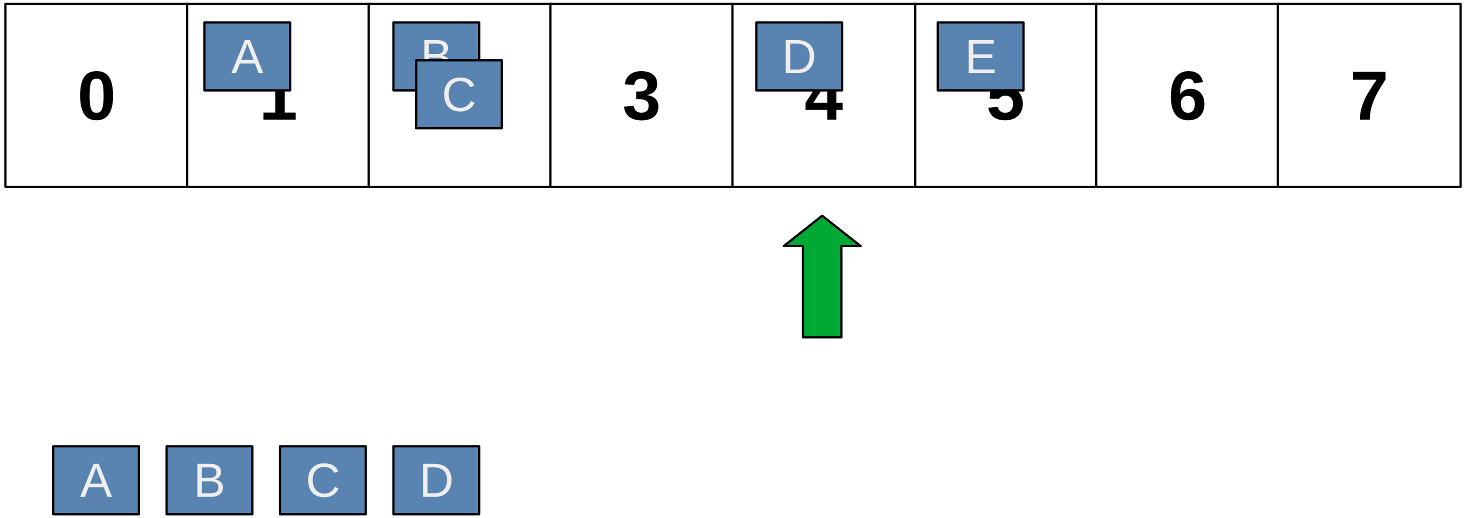
Iterating over a hash table



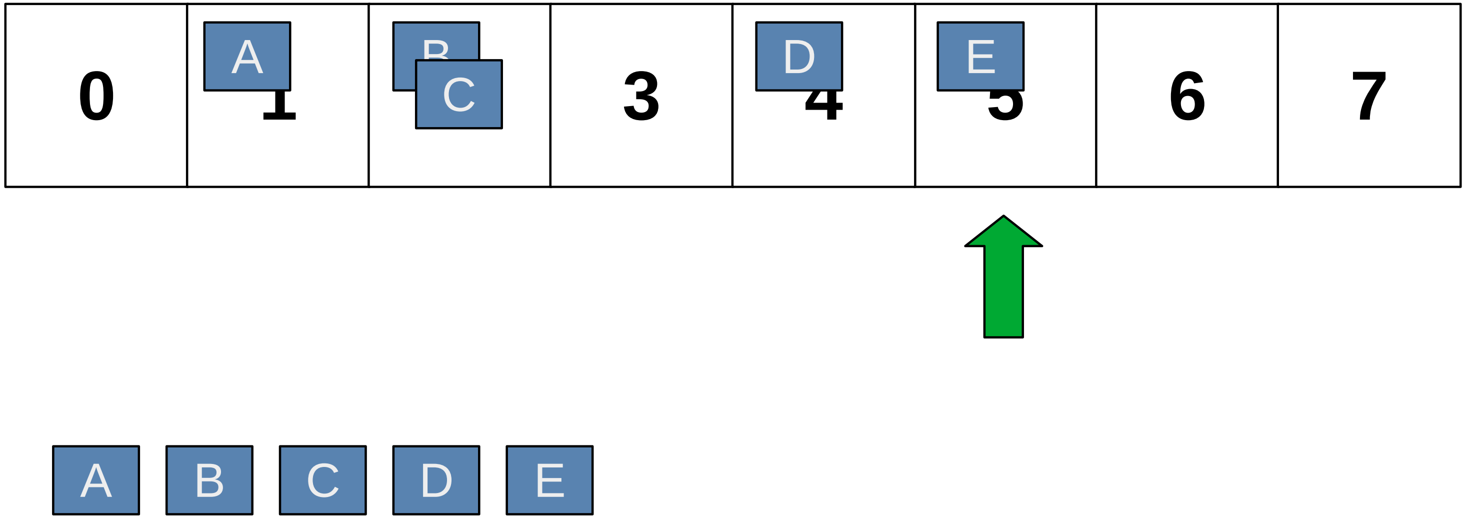
Iterating over a hash table



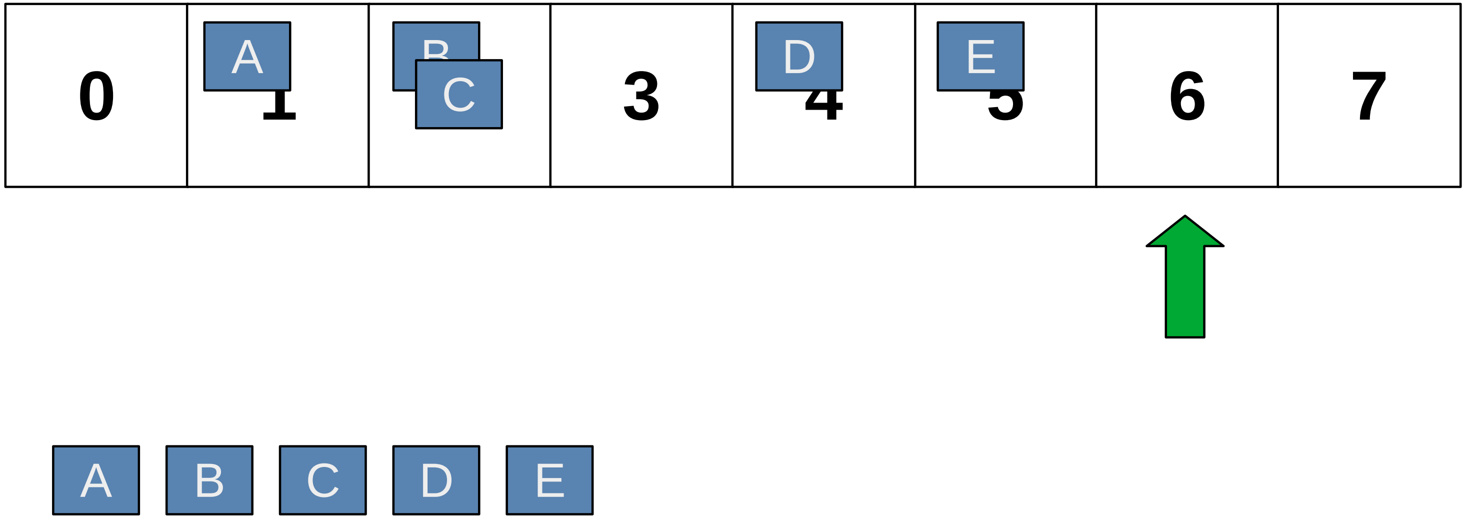
Iterating over a hash table



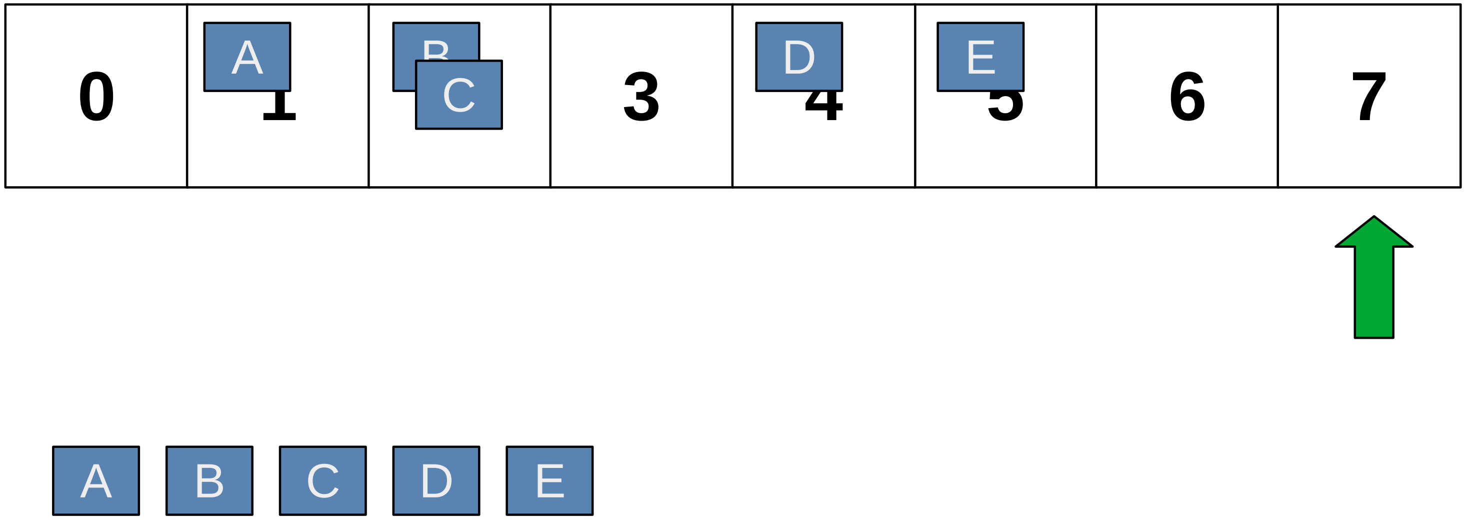
Iterating over a hash table



Iterating over a hash table



Iterating over a hash table



Iterating over a hash table

- Runtime
  - Visit every hash bucket
    - O(N)
  - Visit every element in every bucket
    - O(n)

  = O(N + n)

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$ Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Fix an α_{max} and start requiring that $\alpha \leq \alpha_{max}$

What happens when the user inserts n = N x α_{max} + 1 records ?

Rehashing

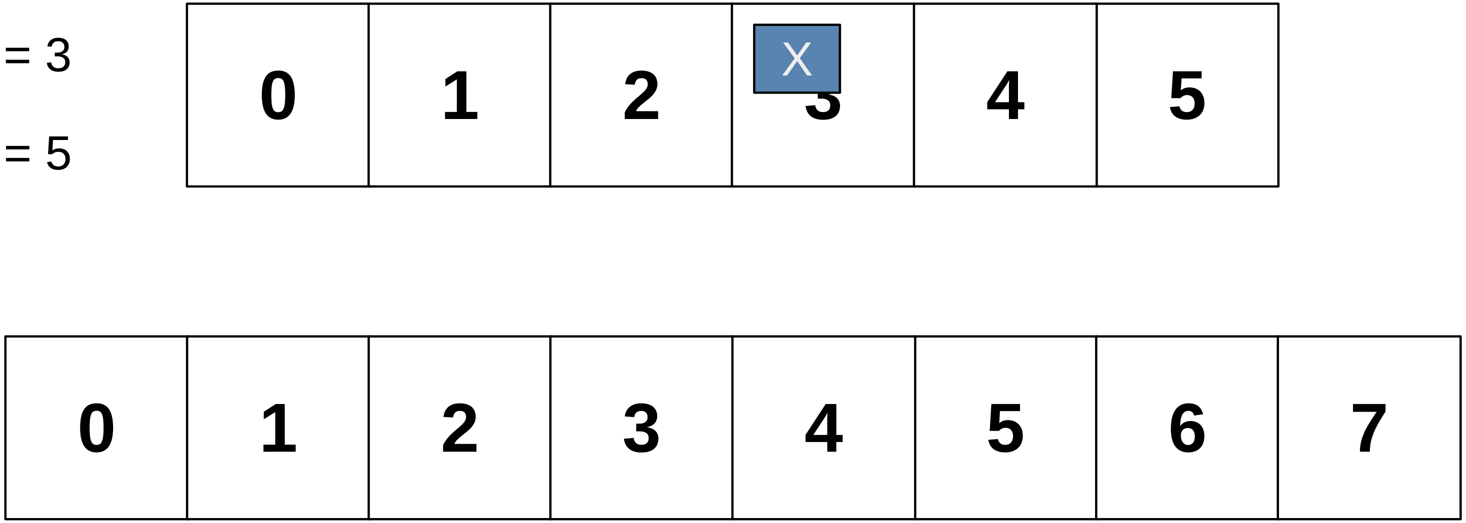
- Resize the array from N_{old} to N_{new}.
  - Element x moves from hash(x) % N_{old} to hash(x) % N_{new}

Rehashing

hash(x) = 1029

1029 % 6 = 3

1029 % 8 = 5



Rehashing

- Resize the array from N_{old} to N_{new}.
  - Element x moves from hash(x) % N_{old} to hash(x) % N_{new}
- Runtime?
  - Allocate new array: **O(1)**
  - Visit every hash bucket: **$O(N_{old})$**
  - Hash and copy each element to the new array: **O(n)**
  - Free the old array: **O(1)**
  - $O(1) + O(N_{old}) + O(n) + O(1) = O(N_{old}+n)$

Rehashing

- Whenever $\alpha > \alpha_{max}$, rehash to double size
  - Contrast with ArrayBuffer
- Starting with N buckets, after n insertions..
  - Rehash at $n_1 = \alpha_{max} \times N$: From N to 2N Buckets
  - Rehash at $n_2 = \alpha_{max} \times 2N$: From 2N to 4N Buckets
  - Rehash at $n_3 = \alpha_{max} \times 4N$: From 4N to 8N Buckets
  - ...
  - Rehash at $n_j = \alpha_{max} \times 2^jN$: From $2^{j-1}N$ to 2^jN Buckets

Number of Rehashes

With n insertions...

$$n = 2^j \alpha_{max}$$

$$2^j = \frac{n}{\alpha_{max}}$$

$$j = \log\left(\frac{n}{\alpha_{max}}\right)$$

$$j = \log(n) - \log(\alpha_{max})$$

$$j \leq \log(n)$$

Total Work

Rehashes required: ≤ log(n)

The i-th rehashing: $O(2^i N)$

Total work after n insertions is no more than...

$$\sum_{i=0}^{\log(n)} O(2^i N) = O\left(N \sum_{i=0}^{\log(n)} 2^i\right)$$

$$= O\left(2^{\log(n)+1} - 1\right)$$

$$= O(n)$$

Work per insertion:
(amortized cost) $O\left(\frac{n}{n}\right) = O(1)$

Recap: So Far

- Current Design: Hash Table with Chaining
  - Array of Buckets
  - Each bucket is the head of a linked list (a “chain”)

Recap: apply(x)

- Expected Cost
  - Find the bucket: $O(c_{hash})$
  - Find the record: $O(\alpha\ c_{equality})$
  - **Total**: $O(c_{hash} + \alpha\ c_{equality}) \approx O(1 + 1) = O(1)$
- Worst-Case Cost
  - Find the record: $O(n\ c_{equality})$
  - **Total**: $O(c_{hash} + n\ c_{equality}) \approx O(1 + n) = O(n)$

Recap: remove(x)

- Expected Cost
  - Find the bucket: $O(c_{hash})$
  - Find the record: $O(\alpha\ c_{equality})$
  - Remove from linked-list: $O(1)$
  - **Total**: $O(c_{hash} + \alpha\ c_{equality} + 1) \approx O(1 + 1 + 1) = O(1)$
- Worst-Case Cost
  - Find the record: $O(n\ c_{equality})$
  - **Total**: $O(c_{hash} + n\ c_{equality} + 1) \approx O(1 + n + 1) = O(n)$

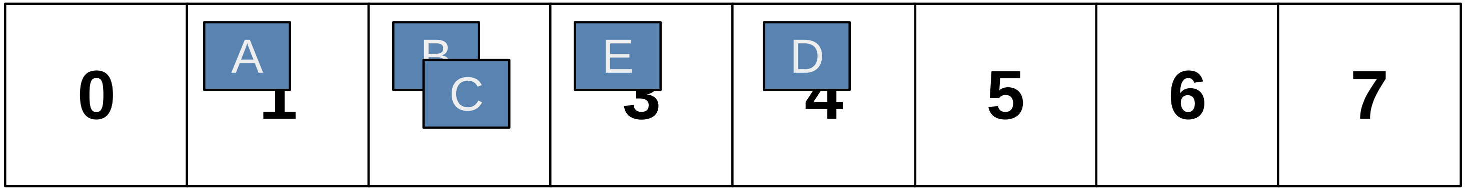
Recap: insert(x)

- Expected Cost
  - Find the bucket: $O(c_{hash})$
  - Remove the key, if present: $O(\alpha\, c_{equality} + 1)$
  - Prepend to linked-list: $O(1)$
  - **Total**: $O(c_{hash} + \alpha\, c_{equality} + 1 + 1) \approx O(1 + 1 + 2) = O(1)$
- Worst-Case Cost
  - Remove the key, if present: $O(n\, c_{equality} + 1)$
  - **Total**: $O(c_{hash} + n\, c_{equality} + 1 + 1) \approx O(1 + n + 2) = O(n)$

Variations

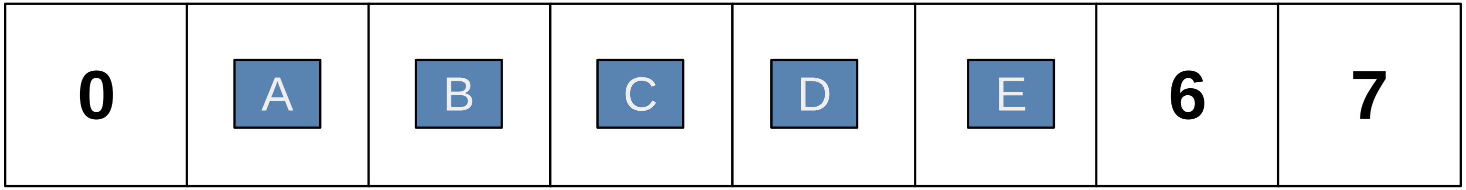
- **Hash Table with Chaining**
  - ... but re-use empty hash buckets instead of chaining
    - **Hash Table with Open Addressing**
    - **Cuckoo Hashing** (Double Hashing)
  - ... but avoid bursty rehashing costs
    - **Dynamic Hashing**
  - ... but avoid O(N) iteration cost
    - **Linked Hash Table**

Chaining



hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

Open Addressing

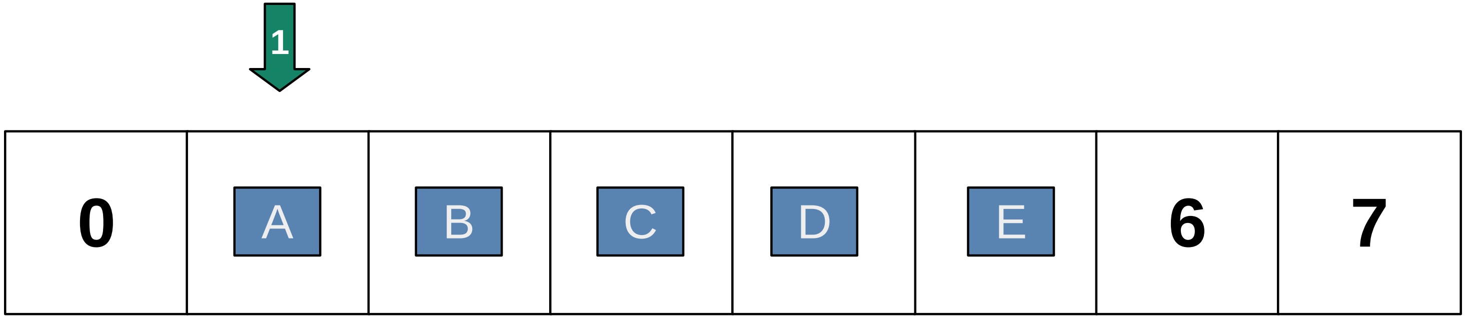


hash(A) = 1
hash(B) = 2
hash(C) = 2 !
hash(D) = 4
hash(E) = 3 !

“Cascade” collisions to the next available spot

Open Addressing

apply(A)

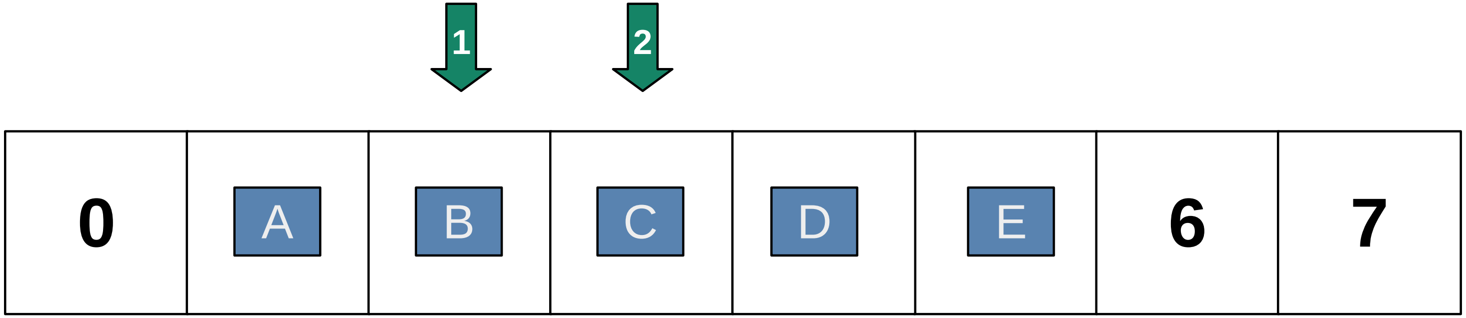


hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

“Cascade” collisions to the next available spot

Open Addressing

apply(C)

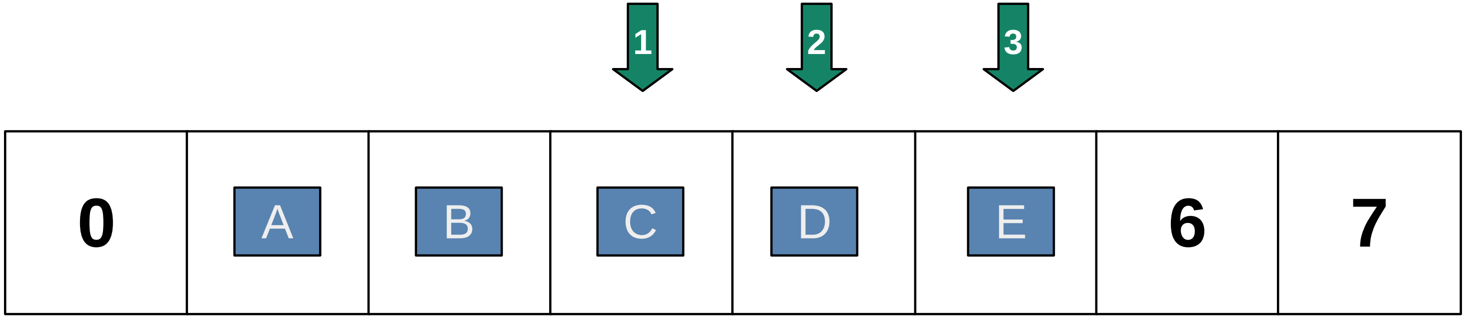


hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

"Cascade" collisions to the next available spot

Open Addressing

apply(E)



hash(A) = 1
hash(B) = 2
hash(C) = 2
hash(D) = 4
hash(E) = 3

"Cascade" collisions to the next available spot

Open Addressing

- insert(X)
  - While bucket hash(X)+i %N is occupied, i = i + 1
  - Insert at bucket hash(X)+i %N
- apply(X)
  - While bucket hash(X)+i %N is occupied
    - If the element at bucket hash(X)+i %N is X, return it
    - Otherwise i = i + 1
  - Element not found

Open Addressing

- remove(X)
  - While bucket hash(X)+i is occupied
    - If the element at bucket hash(X)+i is X, remove it
    - Otherwise i = i + 1

What about elements that were cascaded ?

Removals Under Open Addressing

- Check each element in a contiguous block, starting at hash(X)
  - Move elements up
    - Don't move any element Y ahead of hash(Y)

Open Addressing

- **Linear Probing**: Offset to hash(X)+ci for some constant c
- **Quadratic Probing**: Offset to hash(X)+ci^2 for some constant c
- Follow Probing Strategy to find the next bucket

- Runtime Costs
  - Chaining: Dominated by following chain
  - Open Addressing: Dominated by probing
- With a low enough α_{max}, operations still O(1)

Cuckoo Hashing

- Use two hash functions: $hash_1$, $hash_2$
  - Each record is stored at one of the two
- insert(x)
  - If both buckets are available: pick at random
  - If one bucket is available: insert record there
  - If neither bucket is available, pick one at random
    - “Displace” the record there, move it to the other bucket
    - Repeat displacement until an empty bucket is found

apply(x) and remove(x) is guaranteed O(1)