CSE 250
Lecture 29
Hash Tables
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

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
<th>D'Artagnan</th>
<th>...</th>
<th>Porthos</th>
<th>...</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Athos</td>
<td>Ø</td>
<td>Ø</td>
<td>Ø</td>
<td>...</td>
<td>Ø</td>
<td>...</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Athos: Ø
- B: Ø
- C: Ø
- D'Artagnan: Ø
- ...: Ø
- Porthos: Ø
- ...: Ø
- Y: Ø
- Z: Ø

Aramis

∅
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

Almost Ideal!
... and achievable

apply(k) is something like O(1)?
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 deterministic, but statistically random
Hash Functions

• **Not-so-Wacky Idea**: Use hash function to pick bucket
  - \( h(x) = \text{hash}(x) \mod N \)
  - 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)
  - \( E[X] = (1 \times 0.2) + (2 \times 0.7) + (3 \times 0.1) \)
  - \( = 0.2 + 1.4 + 0.3 = 1.9 \)
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, $0 \leq X_j \leq n$
    - $X_j = 0$ with $p = ???$
    - $X_j = 1$ with $p = ???$
    - ...
    - $X_j = n$ 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 \((n X_j)s\)
  - \( X_{1,j}, X_{2,j}, ..., X_{n,j} \)
- \( \mathbb{E}\left[ \sum_i X_{i,j} \right] = \mathbb{E}[X_{1,j}] + ... + \mathbb{E}[X_{n,j}] \)
  - \( = \frac{1}{N} + \frac{1}{N} + ... + \frac{1}{N} \)
  - \( = \frac{n}{N} \)
- The **expected** runtime of insert, apply, remove is \( O\left(\frac{n}{N}\right) \)
- The **worst-case** runtime of insert, apply, remove is \( O(n) \)
Using Hash Functions

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

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

(simplified, don’t actually do exactly this)
Hash Functions

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>P</td>
<td>D</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

↑
Iterating over a hash table

A

0  A | 1  B | 2  C | 3  D | 4  E | 5 | 6 | 7
### Iterating over a hash table

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>3</th>
<th>D</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

- A
- B
- C

- D
- E
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 \( \alpha \) a constant

Fix an \( \alpha_{max} \) and start requiring that \( \alpha \leq \alpha_{max} \)

What happens when the user inserts \( n = N \times \alpha_{max} + 1 \) records?
Rehashing

- Resize the array from $N_{\text{old}}$ to $N_{\text{new}}$.
  - Element $x$ moves from $\text{hash}(x) \mod N_{\text{old}}$ to $\text{hash}(x) \mod N_{\text{new}}$
Rehashing

hash(x) = 1029

1029 % 6 = 3
1029 % 8 = 5
Rehashing

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

- Runtime?
  - Allocate new array: $O(1)$
  - Visit every hash bucket: $O(N_{\text{old}})$
  - Hash and copy each element to the new array: $O(n)$
  - Free the old array: $O(1)$
  - $O(1) + O(N_{\text{old}}) + O(n) + O(1) = O(N_{\text{old}} + n)$
Rehashing

- Whenever $\alpha > \alpha_{\text{max}}$, rehash to double size
  - Contrast with ArrayBuffer

Starting with $N$ buckets, after $n$ insertions...
  - Rehash at $n_1 = \alpha_{\text{max}} \times N$: From $N$ to $2N$ Buckets
  - Rehash at $n_2 = \alpha_{\text{max}} \times 2N$: From $2N$ to $4N$ Buckets
  - Rehash at $n_3 = \alpha_{\text{max}} \times 4N$: From $4N$ to $8N$ Buckets
  - ...
  - Rehash at $n_j = \alpha_{\text{max}} \times 2^jN$: From $2^{j-1}N$ to $2^jN$ Buckets
Number of Rehashes

With $n$ insertions...

\[ n = 2^j \alpha_{\text{max}} \]
\[ 2^j = \frac{n}{\alpha_{\text{max}}} \]
\[ j = \log\left( \frac{n}{\alpha_{\text{max}}} \right) \]
\[ j = \log(n) - \log(\alpha_{\text{max}}) \]
\[ j \leq \log(n) \]
Total Work

Rehashes required: \( \leq \log(n) \)

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

Total work after \( n \) insertions is no more than...

\[
\sum_{i=0}^{\log(n)} O(2^i N) = O \left( \sum_{i=0}^{\log(n)} 2^i \right) = O \left( 2^{\log(n)+1} - 1 \right) = O(n)
\]

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: $O(c_{\text{hash}})$
  - Find the record: $O(\alpha \ c_{\text{equality}})$
  - **Total**: $O(c_{\text{hash}} + \alpha \ c_{\text{equality}}) \approx O(1 + 1) = O(1)$

- **Worst-Case Cost**
  - Find the record: $O(n \ c_{\text{equality}})$
  - **Total**: $O(c_{\text{hash}} + n \ c_{\text{equality}}) \approx O(1 + n) = O(n)$
Recap: remove(x)

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

- **Worst-Case Cost**
  - Find the record: $O(n \ c_{\text{equality}})$
  - **Total**: $O(c_{\text{hash}} + n \ c_{\text{equality}} + 1) \approx O(1 + n + 1) = O(n)$
Recap: \textit{insert}(x)

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

- **Worst-Case Cost**
  - Remove the key, if present: \(O(n \ c_{\text{equality}} + 1)\)
  - **Total:** \(O(c_{\text{hash}} + n \ c_{\text{equality}} + 1 + 1) \approx O(1 + n + 2) = O(n)\)
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 reheashing 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)

"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 \(\text{hash}(X) + ci\) for some constant \(c\)
- **Quadratic Probing**: Offset to \(\text{hash}(X) + ci^2\) 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 \(\alpha_{\text{max}}\), operations still \(O(1)\)
Cuckoo Hashing

- Use two hash functions: hash₁, hash₂
  - 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)