CSE 250: Hash Tables

Lecture 31

Nov 15, 2023

Reminders

- WA4 due tonight
- PA3 released
  - "Join" two datasets together efficiently.
  - De-anonymize "public" data

The Set ADT

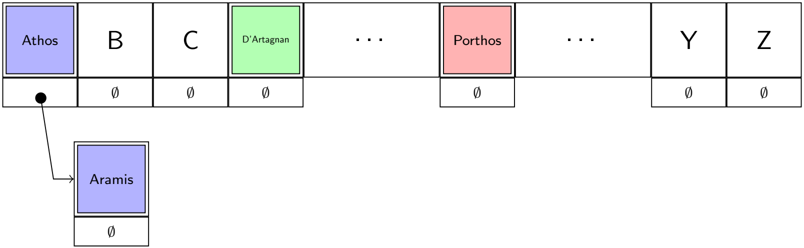
A collection of unique elements (of type `E`)

- `public boolean add(E a)`
  Add an element a to the set and return true. Do nothing and return false if it is already present.
- `public boolean remove(E a)`
  Remove an element a from the set and return true. Do nothing and return false if the element is not in the set.
- `public boolean contains(E a)`
  Return true if and only if the element a is part of the set.
- `public int size()`
  Return the number of elements in the set.

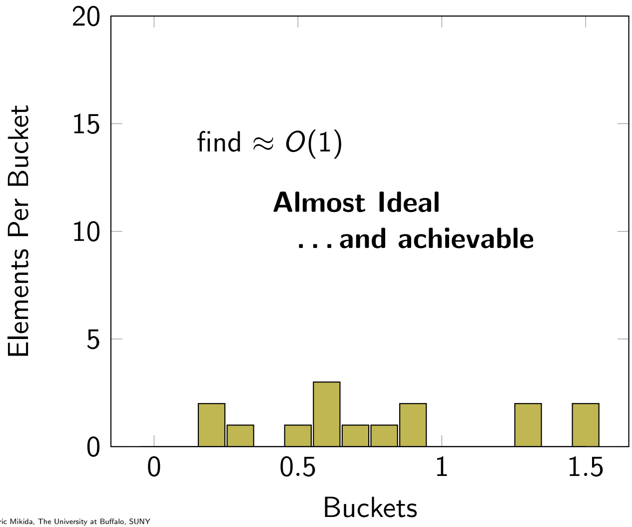
How do we implement a set?

- ~~List (Array or Linked)?~~
- ~~Sorted ArrayList?~~
- Balanced Binary Search Tree (AVL, Red-Black) $O(\log N)$
- Hash Tables

Bucketing Elements with Linked Lists



Picking a lookup function



Hash Functions

Example Hash Functions

- **SHA256** (used by GIT)
- **MD5**, **BCrypt** (used by unix login, apt)
- **MurmurHash3** (used by Scala)

`hash(e)` is pseudorandom

1. `hash(e)` $\sim$ uniform random value in $[0, \texttt{Integer.MAX\_VALUE})$
2. `hash(e)` always returns the same value for the same `e`
3. `hash(e)` is uncorrelated with `hash(e')` for `e` $\neq$ `e'`

Hash Functions

`hash(e)` is ...

- Pseudorandom ("Evenly distributed" over $[0, B)$)
- Deterministic (Same value every time)

Using Hash Functions

Basic Hash: `public int hash(int e)`

- Integers: $\texttt{hash}(e) \mod B$ gets the bucket of e
- Strings: ???

```
public int hashString(String str)
{
  int accumulator = SEED;
  for(c : str.toCharArray())
  {
    accumulator = hash(accumulator + c)
  }
  return accumulator
}
```

(simplified... don't actually do this)

Using Hash Functions in Java

For any object `x`, call `x.hashCode`

HashSet

- `public boolean add(E a)`
  Insert the element into the list at $\texttt{hash}(a) \mod B$.
- `public boolean remove(T a)`
  Find the element in the list at $\texttt{hash}(a) \mod B$ and remove it.
- `public boolean contains(T a)`
  Find the element in the list at $\texttt{hash}(a) \mod B$.
- `public int size()`
  Return a pre-computed size.

Expectation

If X is a variable representing a random outcome, we call the weighted sum of outcomes the **expectation** of X, or $\mathbb{E}[X]$.

If P_i is the probability that $X = x_i$:

$$\mathbb{E}[X] = \sum_i P[X = x_i] \cdot x_i$$

Expected Bucket Size

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 $P[X_j = 0] = ???$
- $X_j = 1$ with $P[X_j = 1] = ???$
- $X_j = 2$ with $P[X_j = 2] = ???$
- $\ldots$
- $X_j = N$ with $P[X_j = N] = ???$

Expected Bucket Size

Assume B buckets.

Start with one insertion ($N = 1$)

- $X_j = 0$ with $P[X_j = 0] = \frac{B-1}{B}$
- $X_j = 1$ with $P[X_j = 1] = \frac{1}{B}$

$\mathbb{E}[X_j] = \left(0 \cdot \frac{B-1}{B}\right) + \left(1 \cdot \frac{1}{B}\right) = \frac{1}{B}$

Expected Bucket Size

For N insertions, we repeat the process: $X_{0,j}, X_{1,j}, X_{2,j}, \ldots X_{N,j}$

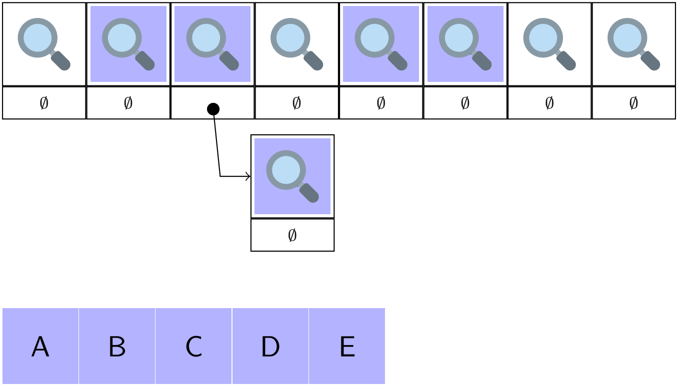
$$\mathbb{E}[\sum_i X_{i,j}] = \mathbb{E}[X_{0,j}] + \mathbb{E}[X_{1,j}] + \ldots + \mathbb{E}[X_{N,j}]$$
$$= \underbrace{\frac{1}{B} + \ldots + \frac{1}{B}}_{N \text{ times}}$$
$$= \frac{N}{B}$$

- **Expected** Runtime of `insert`, `find`, `remove`: $O\left(\frac{N}{B}\right)$
- **Unqualified** Runtime of `insert`, `find`, `remove`: $O(N)$

Hash Table Optimizations

- Improving iteration times
- Resizing the hash table
- Avoiding the linked list

Iterating over a Hash Table



Iterating over a Hash Table

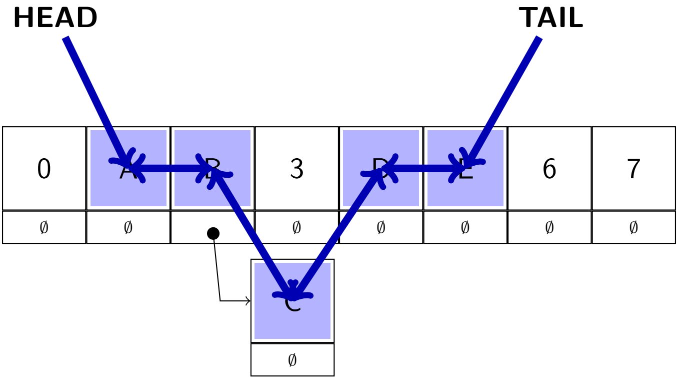
- Visit every hash bucket $O(B)$
- Visit every element in every hash bucket $O(N)$

Total: $O(B + N)$

Linked Hash Table

Idea: Organize the hash table elements in a linked list

Linked Hash Table



Iterating over a Linked Hash Table

- Visit every element via linked list $O(N)$

Total: $O(N)$ (no more $O(B)$ factor)

Insert (Changes only)

- Append the new element to the tail of the linked list. $O(1)$

Remove (Changes only)

- Remove the element from its position in the linked list. $O(1)$

Resizing the Hash Table (Rehashing)

Remember the load factor $\alpha = \frac{N}{B}$

The expected runtime of `insert`, `find`, `remove` is $O(\alpha)$

If we can ensure that $\alpha \leq \alpha_{max}$ for some constant α_{max}, then $O(\alpha) = O(1)$

After enough inserts to make $\alpha > \alpha_{max}$ (with B buckets):

- Create a new hash table with $2B$ buckets.
- Insert every element e from the original table into the new one according to $\texttt{hash}(e) \mod 2B$

Resizing the Hash Table

- Rehash at $N_1 = \alpha_{max} \cdot B$ from B to $2B$ buckets.
- Rehash at $N_2 = \alpha_{max} \cdot 2B$ from $2B$ to $4B$ buckets.
- Rehash at $N_3 = \alpha_{max} \cdot 4B$ from $4B$ to $8B$ buckets.
- ...
- Rehash at $N_j = \alpha_{max} \cdot 2^{j-1}B$ from $2^{j-1}B$ to 2^jB buckets.

Resizing the Hash Table

How many times do we rehash for N insertions?

$$N = 2^{j-1}\alpha_{max}$$

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

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

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

$$j \leq \log(N)$$

Resizing the Hash Table

- Rehashes required: $\leq \log(N)$.
- The ith rehashing $O(2^i)$ work.
- Total work after N insertions is no more than...

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

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

$$= O(N)$$

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

Recap: So Far

Current Design: Hash Table with Chaining

- Array of Buckets
- Each bucket is the head of a linked list (a "chain")

Recap: find(x)

Expected Cost

- Find the bucket $O(c_{hash})$[1]
- Find the record in the bucket $O(\alpha \cdot c_{equals})$[2]

Total: $O(c_{hash} + \alpha c_{equals}) = O(1 + 1) = O(1)$

Unqualified Worst-Case Cost

- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(N \cdot c_{equals})$

Total: $O(c_{hash} + N \cdot c_{equals}) = O(1 + N) = O(N)$

[1] c_{hash} is the cost of the hash function.
[2] c_{equals} is the cost of `.equals`.

Recap: insert(x)

Expected Cost

- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(\alpha \cdot c_{equals})$
- Replace the existing record or append it to the list $O(1)$

Total: $O(c_{hash} + \alpha c_{equals} + 1) = O(1 + 1 + 1) = O(1)$

Unqualified Worst-Case Cost

- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(N \cdot c_{equals})$
- Replace the existing record or append it to the list $O(1)$

Total: $O(c_{hash} + N \cdot c_{equals} + 1) = O(1 + N + 1) = O(N)$

Recap: remove(x)

Expected Cost

- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(\alpha \cdot c_{equals})$
- Remove the record from the linked list $O(1)$

Total: $O(c_{hash} + \alpha c_{equals} + 1) = O(1 + 1 + 1) = O(1)$

Unqualified Worst-Case Cost

- Find the bucket $O(c_{hash})$
- Find the record in the bucket $O(N \cdot c_{equals})$
- Remove the record from the linked list $O(1)$

Total: $O(c_{hash} + N \cdot c_{equals} + 1) = O(1 + N + 1) = O(N)$

HashSet

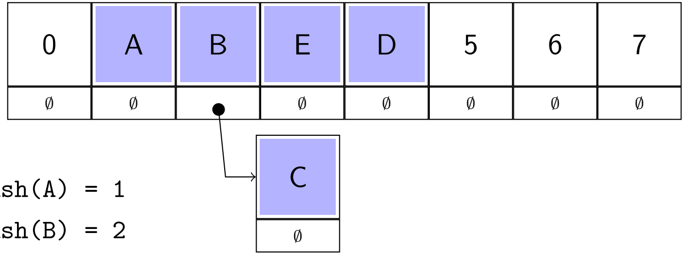
- `public boolean add(E a)`
  Insert the element into the list at $\texttt{hash}(a) \mod B$.
  Expected $O(1)$
- `public boolean remove(T a)`
  Find the element in the list at $\texttt{hash}(a) \mod B$ and remove it.
  Expected $O(1)$
- `public boolean contains(T a)`
  Find the element in the list at $\texttt{hash}(a) \mod B$.
  Expected $O(1)$
- `public int size()`
  Return a pre-computed size. $O(1)$

More Optimizations

Hash Table with Chaining

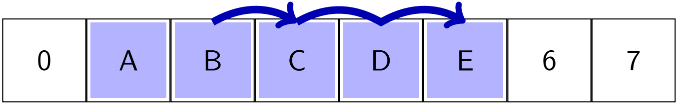
- ... but re-use empty hash buckets instead of linked lists.
  - **Hash Table with Open Addressing**
  - **Cuckoo Hashing** (Double Hashing)
- ... but avoid bursty re-hashing costs
  - **Dynamic Hashing**

Hash Table with Chaining




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

Hash Table with Open Addressing



hash(A) = 1

hash(B) = 2

hash(C) = 2

hash(D) = 4

hash(E) = 3

Open Addressing

insert(a)

- Start at $i = 0$
- While bucket $\texttt{hash}(a) + i \mod N$ is occupied $i = i + 1$
- Insert at bucket $\texttt{hash}(a) + i \mod N$

find(a)

- Start at $i = 0$
- While bucket $\texttt{hash}(a) + i \mod N$ is occupied:
  - If bucket $\texttt{hash}(a) + i \mod N$ holds a, return true
  - Otherwise $i = i + 1$
- Return false

Open Addressing

remove(a)

- Find the bucket containing a.
- For every element in the contiguous block following a:
    - Move the element b into the newly freed spot unless $\texttt{hash}(b) < \texttt{hash}(a) + i$
    - Move to the next element

Open Addressing

Variant Probing Strategies

- **Linear Probing**: Offset to $\texttt{hash}(a) + c \cdot i$ for some constant c
- **Quadratic Probing**: Offset to $\texttt{hash}(a) + c \cdot i^2$ for some constant c

Runtime Costs

- **Chaining**: Runtime dominated by the size of the biggest linked list
- **Open Addressing**: Runtime dominated by probing

With a low enough α_{max}, operations remain expected $O(1)$

Cuckoo Hashing

Let's say we're ok with a more expensive insert/remove.
Can we get $O(1)$ find?

Dynamic Hashing

The amortized cost of a rehash is $O(1)$, but periodic lag spikes can be annoying.

Can we “flatten out” the lag spikes?