CSE 250: Hash Tables

Lecture 32

Nov 17, 2023

Reminders

- PA3 Tests due Weds, Nov 22 **Corrected**
- PA3 Implementation due Sun, Dec 3

Recap: So Far

Current Design: Hash Table with Chaining

- Array of Buckets
- Hash Function assigns each set element to a bucket
- 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

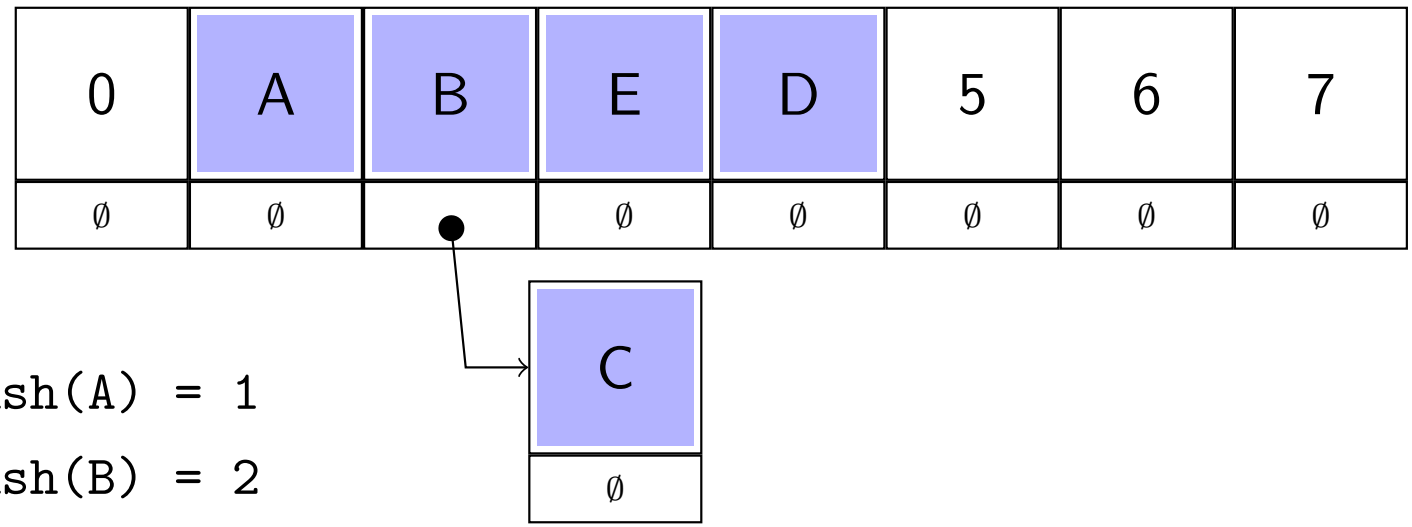
- `public boolean add(E a)`
  Look at array index `hash`(a) mod B; If the element is not in the linked list there, insert it. **Expected** $O(1)$
- `public boolean remove(T a)`
  Look at array index `hash`(a) mod B; If the element is in the linked list there, remove it. **Expected** $O(1)$
- `public boolean contains(T a)`
  Look at array index `hash`(a) mod B; If the element is in the linked list there, return true. **Expected** $O(1)$
- `public int size()`
  Return a pre-computed size. $O(1)$

More Optimizations

Hash Table with Chaining

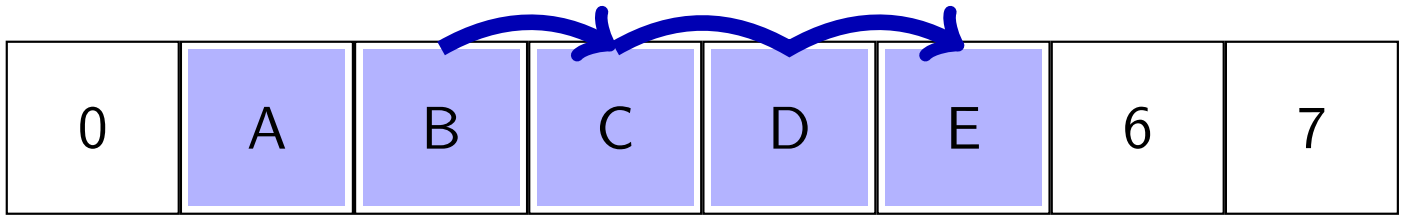
- . . . but re-use empty hash buckets instead of linked lists.
  - **Hash Table with Open Addressing**
  - **Cuckoo Hashing** (Double Hashing)
- . . . but avoid bursty re-hashing costs
  - **Dynamic Hashing**

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

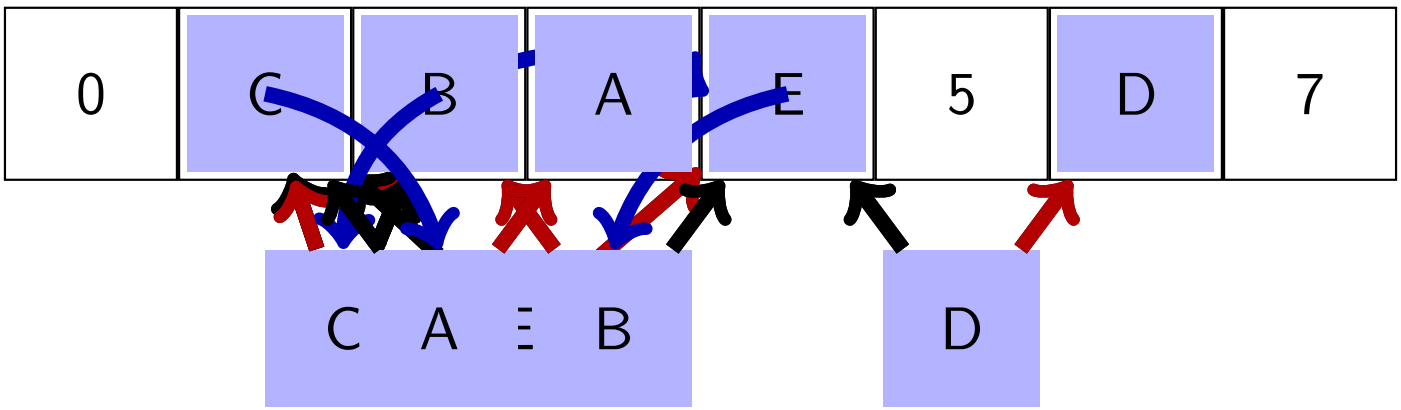
Let's say we're ok with a more expensive insert/remove.
Can we get $O(1)$ find?

Cuckoo Hashing

About 56 of the Old World species and three of the New World cuckoo species (pheasant, pavonine, and striped) are brood parasites, laying their eggs in the nests of other birds and giving rise to the metaphor "cuckoo's egg". These species are obligate brood parasites, meaning that they only reproduce in this fashion.

Wikipedia; Image by JJ Harrison, used under CC-BY 3.0

Cuckoo Hashing



$\texttt{hash}_1\texttt{(A)} = 1;\ \texttt{hash}_2\texttt{(A)} = 3$

$\texttt{hash}_1\texttt{(B)} = 2;\ \texttt{hash}_2\texttt{(B)} = 4$

$\texttt{hash}_1\texttt{(C)} = 2;\ \texttt{hash}_2\texttt{(C)} = 1$

$\texttt{hash}_1\texttt{(D)} = 4;\ \texttt{hash}_2\texttt{(D)} = 6$

$\texttt{hash}_1\texttt{(E)} = 1;\ \texttt{hash}_2\texttt{(E)} = 4$

Cuckoo Hashing

Find $O(1)$

- Look at array index $\texttt{hash}_1 \mod B$
- Look at array index $\texttt{hash}_2 \mod B$

Cuckoo Hashing

- **Find** is unqualified $O(1)$
- **Remove** is unqualified $O(1)$
- **Insert** is expected $O(1)$ (for low values of α)

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?

Dynamic Hashing

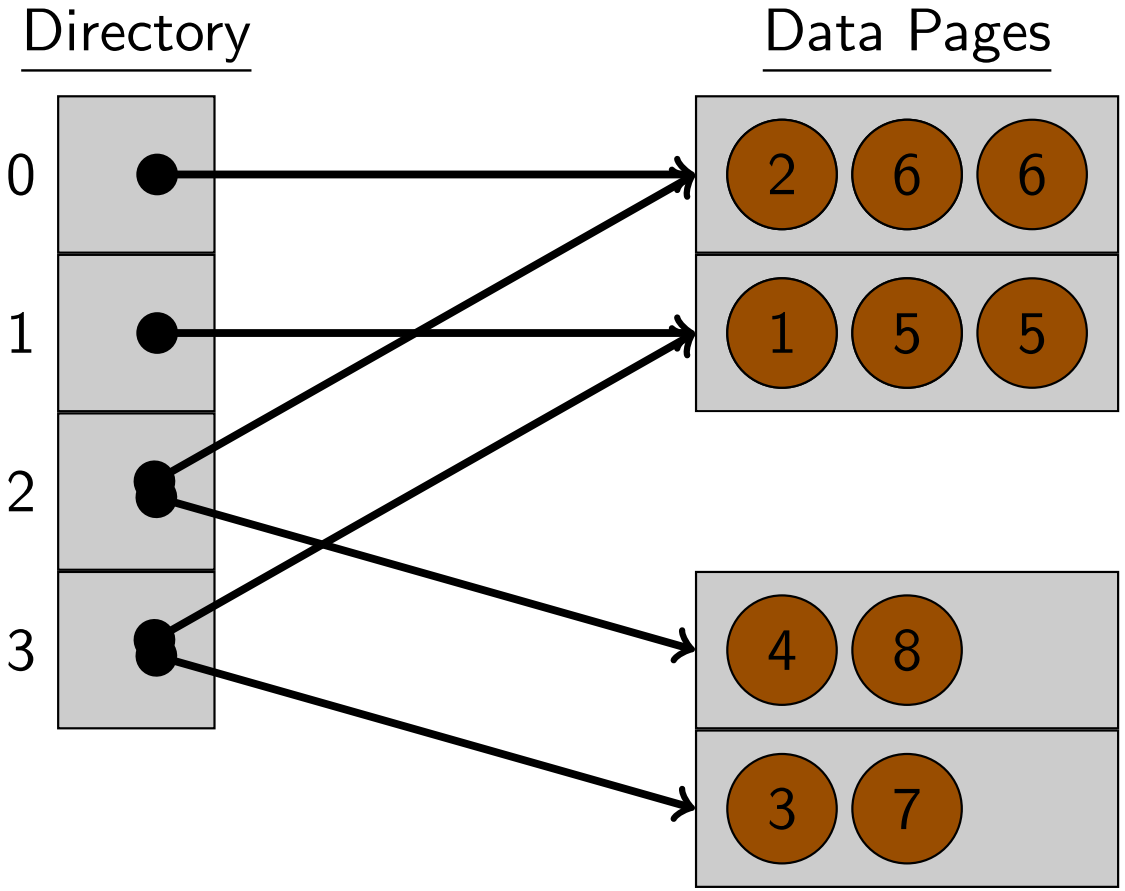
Observation: If $a = N \mod B$ then either

- $a = N \mod 2B$, or
- $a + B = N \mod 2B$

Doubling the size of the hash table always rehashes every element in a specific bucket to one of two places.

Dynamic Hashing

Example: $\texttt{hash}(N) = N$



Dynamic Hashing

- An array (of size B) of pointers to arrays (each of size α). (and some book-keeping metadata)
- When doubling the array size, only copy the array pointers. (faster than rehashing the entire hash table)
- Only split one bucket at a time
- Only double the array when a bucket being split has only one pointer to it.

A Dynamic Hash Table does not have better asymptotic complexity than a Hash Table with Chaining (but has a better constant factor).

The Map ADT

A collection of key-value pairs (key type `K`, value type `V`) with unique keys.

- `public void put(K key, V value)`
  Insert the pair `key, value` into the map, replacing any existing pair with key `key`.
- `public V remove(K key)`
  Remove the pair with key `key`, returning the pair's value if it is present.
- `public boolean contains(K key)`
  Return true if the map contains a pair with key `key`.
- `public int size()`
  Return the number of pairs in the map.

HashMap

- `public void put(K key, V value)`
  Look at array index `hash`(*key*) mod B; Remove the pair with the same key if present; Insert the new pair. **Expected** $O(1)$
- `public V remove(K key)`
  Look at array index `hash`(*key*) mod B; Remove the pair with the same key if present. **Expected** $O(1)$
- `public boolean contains(K key)`
  Look at array index `hash`(*key*) mod B; Return true if there is an existing pair with the key. **Expected** $O(1)$
- `public int size()`
  Return a pre-computed size. $O(1)$