Hash Tables

DSA, Fall 2022

Dynamic Set ADT

```
type Set
type Cell

NewSet:   ()              → *Set
Search:   (*Set, int)     → *Cell
Insert:   (*Set, *Cell)   → ()
Delete:   (*Set, *Cell)   → ()
Minimum:  *Set            → *Cell
Maximum:  *Set            → *Cell
```

Asymptotic running times

	Balanced Binary Search Tree	Balanced Binary Search Tree (with min / max)
Search	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
Insert	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
Delete	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
Minimum	$\Theta(\log_2 n)$	$\Theta(1)$
Maximum	$\Theta(\log_2 n)$	$\Theta(1)$

Asymptotic running times

	Balanced Binary Search Tree	Balanced Binary Search Tree (with min / max)
Search	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
Insert	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
Delete	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$
Minimum	$\Theta(\log_2 n)$	$\Theta(1)$
Maximum	$\Theta(\log_2 n)$	$\Theta(1)$

Constant-time Search and Insert operations?

Can we make Search and Insert constant-time?

We'd need to know exactly where in the structure an element goes

- we can't afford to look for it

One solution:

Compute the "position" of an element in the structure from the element itself

Constant-time Search and Insert operations?

Intuition:

- suppose elements are integers {0, …, N-1}
- allocate an array of size N
- store element E at position E in the array
- look for element E at position E in the array

What about when elements are not integers {0, …, N-1}?

Hash functions

A hash function is a function from some domain D to {0, …, N-1} for some N

- D is usually taken to be integers
- deterministic — if $h(E_1) \neq h(E_2)$ then $E_1 \neq E_2$

Prototypical hash function

$$h(E) = E \bmod N$$

Hash functions for non-integers:
convert element to an integer then apply a hash function

Hash functions

A hash function is a function from some domain D to {0, …, N-1} for some N

- D is usually taken to be integers
- deterministic — if $h(E_1) \neq h(E_2)$ then $E_1 \neq E_2$

Prototypical hash function

$$h(E) = E \bmod N$$

Hash functions for non-integers:
convert element to an integer t

Terminology:

Element E **hashes** to its **hash value** h(E)

Simple Uniform Hashing

Not all hash functions are created equal

A "good" hash function should spread hash values around $\{0, \ldots, N\text{-}1\}$ equally

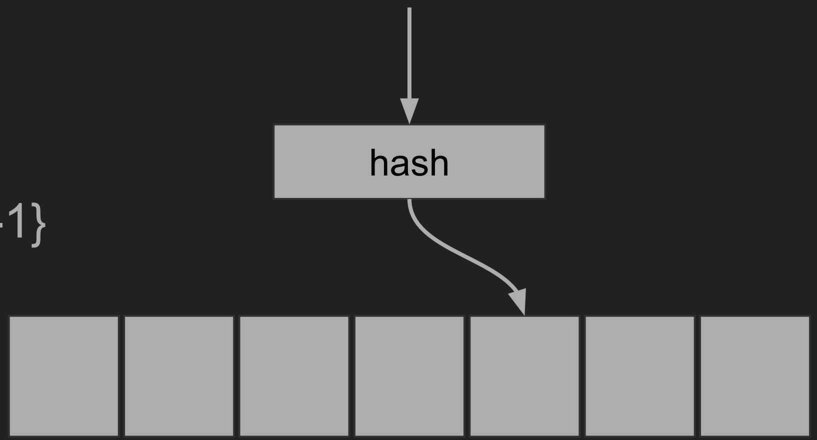
A hash function has the simple uniform hashing property if every element is equally likely to has to any of the N positions

$\sum_{E\,:\,h(E)=j} P(E) = 1/N \quad \text{for } j \in \{0, \ldots, N-1\}$ $\qquad$ $P(E)$ = probability of element E

Hash tables

A hash table (of size N) is:

- an array T of size N
- a hash function h from values to {0,, …, N-1}



Intuition:

- to insert element E, put it at T[h(E)]
- to search for element E, look at T[h(E)]

The Pigeonhole Principle

- if you have M > N elements, then there must exist $E_1 \neq E_2$ with $h(E_1) = h(E_2)$

Hash tables

A hash table (of size N) is:

- an array T of size N
- a hash function h from values to {0,, …, N-1}

Intuition:

- to insert element E, put it at T[h(E)]
- to search for element E, look at T[h(E)]

This is a **collision**

The Pigeonhole Principle

- if you have $M > N$ elements, then there must exist $E_1 \neq E_2$ with $h(E_1) = h(E_2)$

Resolving collisions

Different ways to handle collisions in a hash table

Lead to different flavors of hash tables

Hash tables with chaining:

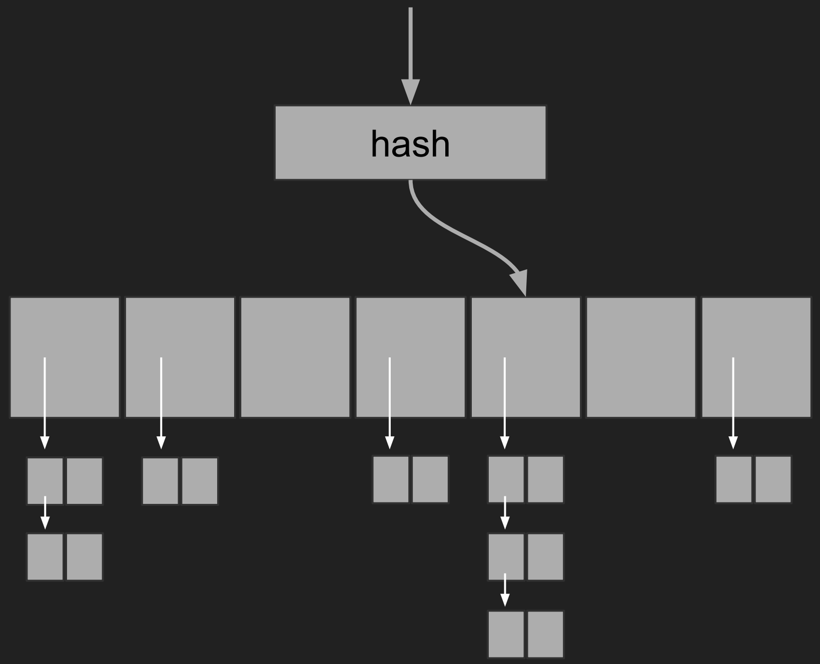
- each cell is a linked list of values
- unlimited capacity

Hash tables with open addressing:

- if a cell is already taken, go to a "backup" cell
- no pointers, but can fill up

Hash tables with chaining

Each cell of the array is a linked list of elements that hash to the same hash value



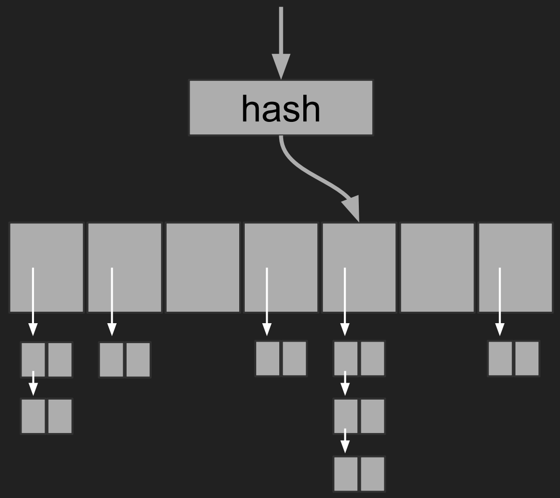
Operations

To insert element E:

- compute h(E)
- add E at the front of the linked list at T[h(E)]

To search for element E:

- compute h(E)
- look for E in the linked list at T[h(E)]



Running time

m = number of elements stored in the table

Load α = m / N

Insertion:

- worst-case: Θ(1)

Search:

- worst-case: Θ(m)
- average case: Θ(1 + α) assuming simple uniform hashing

Hash table with open addressing

Elements are stored directly in the table

If a cell is occupied, deterministically go through (probe) a sequence of "backup cells" finding the first free one ⇒ probing sequence

Various ways of computing a probing sequence:

- linear probing
- quadratic probing
- double hashing

Hash function with probing

$h: U \times \{0, 1, \ldots, N-1\} \to \{0, 1, \ldots, N-1\}$

$h(E, 0), h(E, 1), h(E, 2), \ldots, h(E, N-1)$

should be a permutation of $\{0, 1, \ldots, N-1\}$

Linear probing — given a hash function h' :

$$h(E, i) = (h'(E) + i) \bmod N$$

Quadratic probing — given a hash function h' :

$$h(E, i) = (h'(E) + c_1 i + c_2 i^2) \bmod N$$

Double hashing — given two hash functions h_1 and h_2 :

$$h(E, i) = (h_1(E) + i\, h_2(E)) \bmod N$$

Operations

To insert element E:

- for i = 0, 1, 2, …, N-1, find first free T[h(E, i)] and store E at position h(E, i)

To search for element E:

- for i = 0, 1, 2, …, N-1, search for E in T[h(E, i)]
- abort when you find an empty cell

When α gets close to 1:

- allocate a larger table (increase N) and rehash elements in table

Running time

Open addressing means $\alpha = m / N < 1$

Insert:

- worst-case: $\Theta(m)$
- average case: $\Theta(1 / (1 - \alpha))$ assuming simple uniform hashing

Search:

- worst-case: $\Theta(m)$
- average case: $\Theta(1/\alpha \ln 1/(1-\alpha))$ assuming simple uniform hashing

Running time

Open addressing means α = m / N < 1

Insert:

- worst-case: Θ(m)
- average case: Θ(1 / (1

50% full hash table E[# probes] < 1.4

90% full hash table E[# probes] < 2.6

Search:

- worst-case: Θ(m)
- average case: Θ(1/α ln 1/(1-α)) assuming simple uniform hashing