Hash Tables

Another implementation for the dictionary ADT, but has (under suitable assumptions) constant time operations

Relevant sets

- the universe: $U = \{k : k \text{ is a possible key}\}$
- keys in use: $K = \{k : k \text{ is used to store something}\}$
  - size of the dictionary is $n = |K|$

Naive Implementation - Direct Addressing

Have an array `S` of size $|U|$, then for a key $k \in \{0, \ldots, |U| - 1\}$, store its value in `S[k]`, or if that key is not present, store a flag in `S[k]` marking it empty

e.g. if $|U| = 4$ and we `insert(2, 1)`, then `arr = [-, -, 1, -]`

But what if $|U|$ is too big?

- what if $|U|$ is too big for an array of its size to be stored
- what if $|U| >> |K|$, so an array of size $|U|$ would be incredibly wasteful

Hash Functions

A hash function maps a large set of inputs to a small set of things to use as keys

If we have an array `S` of size m, we can use a hash function $h : U \to \{0, \ldots, m - 1\}$ to "hash" keys into places in `S`

Don't worry about actual hash functions, they are mysterious.

"People go to Hogwarts for years to learn how to do this sort of thing."

- Danny Heap, on proving the effectiveness of hash functions

Collisions

If $h : U \to K$ and $|K| < |U|$, then h cannot be injective, so there could be a *collision*: two different keys share the same hash

Solutions:

- chaining: each slot of array is a (doubly) linked list storing all keys that hash to that slot
- open addressing (commonly used)

Chaining

- `insert(S, k)`
  - if nothing else has taken slot $h(k)$ thus far, store a node with k as the `head` of `S[h(k)]`
  - otherwise, store k at `S[h(k)]` and point to node that was previously there
- `delete(S, k_ptr)`
  - given a pointer to the node `n` storing k, set `n.prev.next = n.next` (linked list deletion)
- `search(S, k)`
  - check if `S[h(k)]` stores `k`
  - if not, check `S[h(k)].next`
    - if not, repeat this step
  - if the end of a linked list is reached, then `k` is not in the dictionary

SUHA - Simple Uniform Hashing Assumption

We assume that $h(k)$ is equally likely to take on all values in $\{0, \ldots, m-1\}$

Application to chaining

SUHA is equivalent to assuming that $\forall i, j \in \{0, \ldots, m-1\}, n_i = n_j$ where n_x is the number of keys that are hashed to x

$\sum_{i=0}^{m-1} E[n_i] = n$, and by SUHA all $n_i \approx n_j$, so each $n_i \approx \frac{n}{m}$ (we call $\alpha = n/m$ the *load factor*)

Performance is bad if α gets too big (as n grows), but we can improve this by expanding the hash table (increasing m)