16 - Hash Tables.

Bit Vector Review

- Suppose we want to store a set $S \subseteq [0, d]$, for some $d \in \mathbb{N}$
- A <u>bit vector</u> representation of S is a Boolean array B

  of size $d+1$ s.t. $B[i] \Leftrightarrow i \in S$,

  or $S = \{0 \leq i \leq d : B[i] \text{ is true}\}$

Eg. $d = 20$, $S = \{3, 7, 9\}$:

$B = $ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- Operations <u>member(x), insert(x), remove(x) are all $O(1)$.</u>
- Only practical where d is small
- Space inefficient if $|S| \ll d$
- Copy, Union, Intersection all $\Theta(c)$

Hash Functions

- A hash function for a set D is a function $h : D \to M$ where $|M| < |D|$, ie a map to a smaller set.

Eg $h : [0, MAXINT] \to [0, 12]$, $h(x) = x \bmod 13$
($|M| = 13$, $|D| = 2{,}147{,}483{,}647$)

- There will be values $x, y \in D$ s.t. $x \neq y$ but $h(x) = h(y)$.

- Notation: Define $h(S) = \{ y : y = f(x) \text{ and } x \in S \}$

Eg $h(3) = 3$; $h(7) = 7$; $h(13) = 0$; $h(15) = 2$; $h(20) = 7$
$h(\{3, 7, 13, 15, 20\}) = \{0, 2, 3, 7\}$

- If $h(x) = h(y)$ for $x, y \in S$, we call it a collision (e.g 3, 15)

- We will want hash functions h s.t.
  - ran h = $[0, m-1]$ for $m \in \mathbb{N}$ (array indices)
  - h tends to distribute S uniformly over $[0, m-1]$
  - $m = |M|$ will be prime

Hash Function + Bit Vector

- Let $h : D \to [0, m-1]$, B a Boolean array of size m
- For a set $S \subseteq D$, set

  $B[i] = \text{true}$ iff there is $x \in D$ s.t. $h(x) = i$

  or $\{i : B[i]\} = h(S)$

Eg: $S = \{3, 7, 13, 15, 20\}$

$h(x) = x \bmod 13$; $m = 13$

$h(S) = \{0, 2, 3, 7\}$

$B = \boxed{1|0|1|1|0|0|0|1|0|0|0|0|0}$

<u>now</u>: $\{x : B[h(x)]\} = \{0, 2, 3, 7, 13, 15, 19, 25, 27, 31, \ldots\}$

- $B[h(x)] = 1$ "suggests $x \in S$"
- $B[h(x)] = 0$ <u>implies</u> $x \notin S$.

<u>eg</u>. there may be false positives but never false negatives.

Bloom Filters

- Let $H = \{h_1, h_2, \ldots h_k\}$ be a set of distinct hash functions for a set D, each with range $[0, m-1]$.
- For $S \subseteq D$, set $B[i]$ = true if $h(x) = i$ for some $h \in H$;
  $B[i]$ = false o.w.
- To test for membership in S:
  - if $B[h(x)]$ = true for all $h \in H$, return true
  - o.w. return false.
- We get a false positive only when $h(x)$ is a collision for every $h \in H$.
- B is a Bloom Filter for S
- If m is large enough relative to $|S|$ and the h_i are good quality, independent hash functions, then there will be few false positives

Hash Tables

- Let $h : D \to M$ be a hash function for D with $M = [0, m-1]$
- Let A be an array of size $|M|$ and type $D \cup \{\_\}$

  eg. $A : M \to D \cup \{\_\}$

- For a set $S \subseteq D$, we want

  $A[h(x)] = x$, for each $x \in S$

  $A[i] = \_$ if $h(x) \neq i$ for every $x \in S$.

  Eg: $S = \{2, 12, 17, 21\}$, $h(x) = x \bmod 13$

  $h(S) = \{2, 12, 4, 8\}$

  $A =$

_	_	2	_	17	_	_	_	21	_	_	12

- To check membership in S, return $A[h(x)]$.
- A is a hash table for S
- But what if we have collisions?
- Need collision handling. We will look at a few methods.

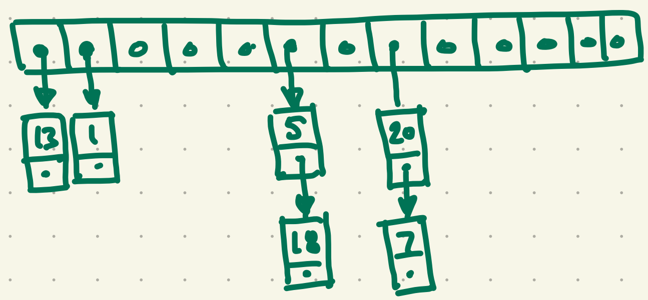
Hashing with Separate Chaining

- Let A be a size-M array of linked lists
- Set $A[i]$ to be a list of the elements $\{x \in S : h(x) = i\}$.
- To test for membership in S:
  - Return true iff x is in the list $A[h(x)]$

Eg. $S = \{1, 5, 7, 13, 18, 20\}$

$h(x) = x \bmod 13$

$A =$



- To insert/remove x: insert/remove x from $A[h(x)]$.
- If h distributes S almost uniformly over M, the lists will be small, and time will be essentially $\Theta(1)$.
- In the worst case, some lists have length $\Omega(n)$ and performance degrades to that of linked lists: $\Omega(n)$.

Hashing with Probing. (Open Addressing)

- Let A be an array of size M and type $D \cup \{-\}$,
  f a hash function $h : D \to [0, m-1]$
- Let f be a function $f : \mathbb{N} \to \mathbb{N}$, that has $f(0) = 0$
  and is monotone increasing (eg. $x > y \Rightarrow f(x) > f(y)$)
- Define, for $i \in \mathbb{N}$, $h_i(x) = (h(x) + f(i)) \bmod m$

Ex. $h(x) = x \bmod 13$, $f(i) = i$

$$h_0(3) = h(3) + 0 = 3$$
$$h_1(3) = h(3) + 1 = 4$$
$$h_2(3) = h(3) + 2 = 5$$

- To resolve collisions, probe the sequence of cells:
  $A[h_0(x)]$, $A[h_1(x)]$, $A[h_2(x)]$, ...

Hashing with Probing. (Open Addressing)

- $h_i(x) = (h(x) + f(i)) \bmod m$

- **To check for membership of x:**
  - Examine the sequence of locations
    $A[h_0(x)], A[h_1(x)], A[h_2(x)], \ldots$
  - Stop at the first location containing x or $\perp$
    - return true if x was found, false otherwise.

- **To insert x:**
  - Examine the sequence of locations
    $A[h_0(x)], A[h_1(x)], A[h_2(x)], \ldots$
  - Stop at the first location containing $-$
    and store x there.

- Choice of $f()$ determines properties.

Hashing with Linear Probing

- Let $f(i) = i$
- The sequence of locations to probe is:

$$A[h(x)],\ A[h(x)+1],\ A[h(x)+2],\ A[h(x)+3],\ \ldots \quad (+\text{ is mod } m)$$

Ex: · Suppose $h(x) = x \bmod 13$, $S = \{2, 9, 18, 36\}$
(so $h(S) = \{2, 5, 9, 10\}$) and A is

_	_	2	_	_	18	_	_	_	9	36	_	_

- To insert 5: · compute $h(5) = 5$;
  · see that $A[5] \neq$ _
  · see that $A[6] =$ _, so set $A[6] = 5$
- Now: $A =$

_	_	2	_	_	18	5	_	_	9	36	_	_

- To check if $5 \in S$: · compute $h(5) = 5$;
  · see that $A[5] \neq$ _, $A[5] \neq 5$
  · see that $A[6] = 5$ and return true
- To check if $31 \in S$: · compute $h(31) = 5$;
  · see that $A[5] \neq 31$, $A[5] \neq$ _
  · see that $A[6] \neq 31$, $A[6] \neq$ _
  · see that $A[7] =$ _ and return false

Hashing with Quadratic Probing.

- Let $f(i) = i^2$
- The sequence of locations to probe is:

  $A[h(x)], A[h(x)+1], A[h(x)+4], A[h(x)+9], \ldots$ (+ is mod m)

Ex: Suppose $h(x) = x \bmod 13$, $S = \{2, 9, 18, 36\}$

(so $h(S) = \{2, 5, 9, 10\}$) and A is

_	_	2	_	_	18	_	_	_	9	36	_	_

- To insert 35:
  - compute $h(35) = 9$
  - see that $A[9] \neq$ _
  - see that $A[10] \neq$ _
  - see that $A[0] =$ _ and store 35 there.
- Now: A is

_	35	2	_	_	18	_	_	_	9	36	_	_

- To check if $35 \in S$:
  - compute $h(35) = 9$
  - see that $A[9] \neq$ _, $A[9] \neq 35$
  - see that $A[10] \neq$ _, $A[10] \neq 35$
  - see that $A[0] = 35$ and return true
- To check if $22 \in S$:
  - compute $h(22) = 9$
  - see that $A[9], A[10], A[0], A[5]$ are not 22 or _
  - see that $A[12] =$ _ and return false

Double Hashing

- Let $f(i) = i \cdot hash_2(x)$,
  where $hash_2(x)$ is a hash function for D that is different from h, and with $ran(hash_2) \subseteq [1, m]$
- The sequence of locations to probe is:

  $(+ \text{ is mod } m)$

  $A[h(x)], A[h(x) + hash_2(x)], A[h(x) + 2 \cdot hash_2(x)], \ldots$

Ex: Suppose: $h(x) = x \bmod 13$, $hash_2(x) = (7 - (x \bmod 7))$

$S = \{2, 9, 18, 36\}$, so $h(S) = \{2, 5, 9, 10\}$

and A is

_	_	2	_	_	18	_	_	_	9	36	_	_

- To insert 15:
  - compute $h(x) = 2$
  - see that $A[2] \neq$ _
  - compute $hash_2(x) = 6$
  - see that $A[8] =$ _ and store 15 there
- Now: A is

_	_	2	_	_	18	_	_	15	9	36	_	_

- To check if $15 \in S$, check $A[2]$, then $A[8]$, and return true
- To check if $10 \in S$:
  - compute $h(10) = 10$
  - see that $A[10] \neq 10$, $A[10] \neq$ _
  - compute $hash_2(10) = 4$
  - see that $A[1] =$ _ and return false

Removal with Open Addressing.

- Suppose we have a hash table H for a set S containing x, and want to remove x.
  - If H uses separate chaining, we just delete x.
  - If H uses open addressing, we cannot, because x affects the probe sequence for other elements.

Ex: · Suppose $h(x) = x \bmod 13$, $S = \{2, 5, 9, 18, 36\}$ and A was obtained as in our Linear Probing example: $A =$

–	–	2	–	–	18	5	–	–	9	36	–	–

· Suppose we now delete 18, so

$A =$

–	–	2	–	–	–	5	–	–	9	36	–	–

· Now, searching for 5 fails, because

$$A[h(5)] = \_ \quad !$$

- One solution is to <u>mark</u> cells where we have deleted elements.

Removal with Open Addressing.

<u>Ex</u>: In the previous example, to remove 18 we replace it with d:

A =

_	_	2	_	_	d	5	_	_	9	36	_	_

Now, search & insert procedures perform as if $A[5]$ has some key that we will never use.

- <u>To remove x:</u>
  - examine the sequence of locations

    $A[h_0(x)], A[h_1(x)], A[h_2(x)], \ldots$
  - when x is found, replace it with d
- Notice that search & insert work correctly as they are
- Insert can be modified to reclaim space:

  To insert x: - Examine the sequence of prob locations

  - Stop at the first one containing _ or d and store x there.
- NB: In implementation, d and _ could be special values, or A could be an array of objects or structs with "empty" and "deleted" variables/fields.

Load Factor

- The <u>load factor</u> of a hash table H is:

$$\lambda = \frac{(\#\text{ of keys}) + (\#\text{ of elements marked } d)}{m}$$

(If H uses separate chaining, there are no d's)

- Good performance requires λ not too large.
- For separate chaining: λ should not be much larger than 1, so average list length is about 1.
- For open addressing, want $\lambda < 0.5$, so that it is not too hard to find a place to make an insertion.

Some Properties with Open Addressing.

- Linear Probing:
  - Insertion always succeeds if $\lambda < 1$
  - Primary Clustering is a serious problem.

- Quadratic Probing:
  - Avoids primary clustering
  - Exhibits secondary clustering – but less problematic
  - Insertion alway succeeds if $\lambda \leq 0.5$, but may fail if $\lambda > 0.5$ (even if there is space).

- Double Hashing:
  - Requires design of a second suitable hash function
  - Requires computing 2 hash functions whenever probing beyond $A[h_0(x)]$ is needed.

Rehashing

- Rehashing hash table H means constructing a completely new hash table for the contents of H.

- We may want to do it if:
  - λ is too large (close to 0.5 for open addressing, much larger than 1 for separate chaining)
  - Performance has become poor (which may result from clustering, from long linked lists, or from many removals)

- Takes time $\Theta(n)$ under the assumption that insert is $\Theta(1)$.

Hashing Properties

- Well-designed hash tables are effective in practice, with fast insert, member, remove operations
- Require a good hash function for the domain of application
- Operations O(1) <u>on average</u>, under assumptions that may not hold in practice:
  - all keys equally likey
  - hash function distributes keys uniformly
  - λ small
- Do not support operations based on order of keys, such as:
  - enumerate in order
  - min, max, range lookups
  - union + intersection

(These are efficient with AVL Trees & B-Trees).

End