Lecture 24: hashing

- Map interface
- Hash tables
- Hash codes
- Chaining
Map (also a dictionary, hash table)
- Store values associated with keys.
- Put a new value with a given key.
- Check to see if my value associated with a given key.
- Look up value for a given key.

Examples:
- Keys: words, values: definitions.

Possible implementations:
- Linked list
  - Key position in list (wiers)
  - Value: whatever is stored there.

```
  a1 -> a2 -> a3 -> \ldots
  put (5, D)
  - extend list
  - update to add D
  c1 -> c2 -> c3 -> c4 -> c5
```

- Hash???
  - Look up element with a given index?

```
  o
  / \      \\
  e   d    c  b  a

  Doesn't help.  (Why set k(x)?
  Have to search entire list.)
```

- Array: if keys are small integers
  - Put (i, v)
  - exact where
  - Get (i) return a[i]  \( \in O(1) \)

- Binary search tree
  - Good for large, ordered keys.
  - Put: enter into tree \( \in O(\log n) \) time.
  - Look up: follow path to desired key, \( \in O(\log n) \) time (balanced tree).
Big simplifying assumption: assume data is perfectly "spread out"

Ex: assume data are names, assume as many names start with "a" as with "b"

```
'\(a\)  
  \n    \rightarrow Alice \rightarrow Aaron

'\(b\)  
  \n    \rightarrow Carmen \rightarrow Camilla

'\(c\)  
  \n    \rightarrow greg \rightarrow greg

'\(d\)  
  \n    \rightarrow Ted \rightarrow Zoe
```

\(n\) entries in table

\(N\) buckets

- to look up "John", what do I do?
  - go to John's bucket (bucket \(J\))
    - \(O(1)\) time
  - search through \(J\)'s
    - comparing each to "John"
      - By "magic assumption" each bucket has about \(\frac{n}{N}\) elements,
        - \(O(\frac{n}{N}) = O(1)\) time.

idea: let's make \(d\) constant by necessarily \(N\) as \(n\) increases.
Maintain $\alpha \leq 2$.

When adding an element, if $n > 2N$, i.e., $\alpha \geq 2$, we'll double $N$.

- Make a new array of buckets (size $2N$)
- Copy everything from old array to new (in correct bucket)
- For each elt & all links: compute bucket in new & old, add elt to new bucket $O(1)$ times

$O(1)$ per element in table.

Takes $O(n)$ time only after doing $2N$ insertions.

Can do $m$ insertions in $O(n)$ time, even though in worst case, I need $O(n)$ for one insertion
(takes only happens once per insertion)

Amortized Constant Time
In practice, data is not uniformly distributed.

Worst-case: all keys entries are in some bucket, looking takes $O(n)$ for every entry (not $O(n)$)

Cluster: lots of data in the same bucket.

Generalization: need to associate each key in a general way.

Idea: associate hash code with each key.

Example: function assigns a number to each key:

- Alice: 17
- Karen: 32
- Allie: 6
- Ted: 0
- Lee: 1000

- Can easily compute hash of any key.
- If two keys are equal, hashes should be equal.
- Should be "spread out": if two keys are different, they are unlikely to hash to same bucket (i.e. to have same hash code.)
Hash table: array of $N$ buckets, each containing a list of entries.

- Each entry in bucket $b$ has a key whose hash code is $b \mod N$.
  - If we wrap around, i.e. remainder of hash code / $N$ is $b$.

- No special structure within buckets (unsorted linked list).

To lookup key $k$:
- Hash $k$ to find bucket $b$.
- Linear search in bucket $b$ for key $k$.

Amortized constant time $O(1)$.

To insert value with key $k$:
- Check if need to double size (i.e. if $N \leq 2N$).
- Compute bucket by hashing $k$, taking remainder.
- Insert $(k,v)$ in bucket $b$.
Closed chaining

if load factor is < 1, there is
more space in the array of buckets
than elements in table

open chaining

goal: have:

closed chaining
need to find “backup location” in our
storage array, to handle collisions.

strategy for finding backup location:

- if a slot is full, go to
  next slot.

- hash the bucket id to
  find “backup” bucket

bucket of bucket b
(bucket holds x)

N=8

A

B

1
2
3
4
5
6
7

C

D

0