CS 151
Hash tables & Hash maps
Announcements

Prelab 8 is due Monday morning.

Lab 7 due on Sunday night.

Structural Induction Bonus Session next week, date/time TBA.
   (You guys are ridiculously busy.)
   (Seems that Thursday evening is the best, seconded by Saturday afternoon)
   (I will give you until 5pm to finish the doodle, and then I decide.)
Maps and Dictionaries

We want an ADT that stores information that is reference by key for quick look-up, i.e. we want to store a set of (key,value) pairs.

A map is such as set where there is exactly one value per key; a dictionary is such a set where there could be more than one value per key. We require that the keys be comparable by equality (given two keys, we have to be able to tell if they are equal) but < and > are not necessary.

Operations are
- `put(key, value)` — associate the given value with the given key
- `get(key)` — return the value(s) associated with the given key
- `remove(key)` — return and remove the key (and associated value)s
- `size()`
- `isEmpty()`
- `iterateValues()`, `iterateKeys()`

We want all of the put, get, and remove operations to be $O(1)$, but what we’ll get is amortized (average) $O(1)$, and asymptotic $O(n)$. 
Implementing Maps & Dictionaries

A map is such as set where there is exactly one value per key; a dictionary is such a set where there could be more than one value per key. Operations are

- `put(key, value)` — associate the given value with the given key
- `get(key)` — return the value(s) associated with the given key
- `remove(key)` — return and remove the key (and associated value)s
- `size()`
- `isEmpty()`
- `iterateValues()`, `iterateKeys()`

idea #1: use a TreeMap, as you do on lab 6

Problem: for binary search trees you need comparable keys, but our keys are only comparable using `==`, not `<` and `>` (which we need for BST’s.)

Still, you can’t get amortized $O(1)$ behaviour in trees (but you can get... what?)
A map is such as set where there is exactly one value per key; a dictionary is such a set where there could be more than one value per key. Operations are

- put(key, value) — associate the given value with the given key
- get(key) — return the value(s) associated with the given key
- remove(key) — return and remove the key (and associated value)s
- size()
- isEmpty()
- iterateValues(), iterateKeys()

idea #2: store elements in a list by their key, store them as (key, value) pairs

Then what would the running time of the operations be?

- put(key, value) — O(1) (just add it to the end)
- get(key) — O(n) (takes O(n) time to find key in unordered list)
- remove(key) — O(n) (takes O(n) time to find key in unordered list)
- size() — O(1)
- isEmpty() — O(1)

Note that this takes O(n) space to store the elements, too.
Implementing Maps & Dictionaries

A map is such as set where there is exactly one value per key; a dictionary is such a set where there could be more than one value per key. Operations are

- put( key, value ) — associate the given value with the given key
- get( key ) — return the value(s) associated with the given key
- remove( key ) — return and remove the key (and associated value)s
- size()
- isEmpty()
- iterateValues(), iterateKeys()

Idea #3: store values in array that has an index/bucket for each possible key (ex. if keys range from 0 to 1000: make array of length 1001.)

- put( key, value ) — O(1) (just add it to array[key])
- get( key ) — O(1) (just return array[key], since that’s where it’s stored.)
- remove ( key ) — O(1) (just set array[key] to null)
- size() — O(1)
- isEmpty() — O(1)

Note that this wastes space and can’t handle duplicates.
idea #4: store values in array of buckets that each store a *range* of keys

A hash table is a generalization of an array such that
- the size of the underlying array, $N$, is proportional to $n$ (limit space waste)
- lookup is $O(1)$ on average (amortized) (although $O(n)$ in worst-case)

I.e. We waste some space to obtain faster lookup, but it’s a trade-off.

More specifically, a hash table consists of 2 components:
- $HT[S]$ : an array of $S$ buckets (this is our hash table, like or usual $data[S]$)
- $h(key)$ : a hash function that maps keys to buckets

Given a key, instead of addressing $HT[key]$ as we did in idea #3, address $HT[h(key) \% S]$. A good hash function will “spread out” the keys.

- $HT[7]$: 
  ```
  [ ] [ ] [ ] [ ] [ ] [ ] [ ]
  ```
- $h(s) = (s.charAt(0) \% 7)$ : a hash function that maps keys to buckets
  
  $put( “aaa”, v1 )$: puts $v1$ into $h( “aaa” ) = 97 \% 7 = 6$.  
  $put( “book”, v2 )$: puts $v2$ into $h( “book” ) = 98 \% 7 = 0$.  
  $put( “hello”, v3)$: puts $v3$ into $h( “hello” ) = 104 \% 7 = 6$. oh oh, collision!
A collision occurs when the hash function maps two keys to the same bucket. There are two ways to deal with collisions:
  • separate chaining
  • open addressing
We will discuss both over the next few classes.
Separate Chaining

Separate chaining allows each bucket to contain a list (chain) of values. That is, if \( h(k) = h(j) \) for two keys \( k \neq j \), then \( HT[h(k)] \) contains both.

**put( key, value ):**

- compute \( \text{index} = h(\text{key}) \mod S \)
- add (key, value) to the bucket at \( HT[\text{index}] \)

RT is \( O(1) \) (just add (key,value) to the end of the list).

**get( key ):**

- compute \( \text{index} = h(\text{key}) \mod S \)
- search through bucket at \( HT[\text{index}] \) to see if entry with key exists

RT is \( O(\text{size of bucket}) \) since we have to search through the bucket for the key.

In the worst case, all items are in one bucket, and get is \( O(n) \). On average, there are \( \lambda := n/S \) elements per bucket, call \( \lambda \) the load factor. Then on average, get is \( O(\lambda) \). Want \( S \) such that \( \lambda := n/S \) is small, \( O(1) \) (\( \lambda \approx 0.75 \) works well).

If, on a put, \( n \) increases enough to make \( \lambda \geq \) preset amount, resize the table (i.e. increase \( S \)) to decrease the load factor, rehash all the elts, and continue.
Separate Chaining

Ex. Suppose keys are ints, S=7, h(n)=n, and we want to keep the LF < 0.75.
put: 10, 1, 12, 4, 8.

now we resize to 17.
if we don’t rehash and we go looking for the 10 or 12, we won’t find them.

RT of put with resize: we only rehash when load factor gets high. Rehashing
and resizing takes O(n) time, but it only happens every O(n) puts if you double
resize, so it is amortized O(1). (Note: we usually resize to a prime, actually...)

remove( key ):
    compute index = h( key ) % S
    search through bucket at HT[ index ] and remove entry with key key
RT is O(size of bucket) since we have to search through the bucket for the key.
As with get, this is worst-case O(n), but amortized O(1) if LF is O(1).