Maps and Dictionaries

Implementing Maps & Dictionaries

- CS 151
- Hash tables & Hash maps
- Friday, November 9, 12

Announcements

- Prelab 8 is due Monday morning.
- Lab 7 due on Sunday night.
- Lab 7 due on Sunday night.
- Lab 7 due on Sunday night.
- Lab 7 due on Sunday night.

Maps and Dictionaries

Maps & Dictionaries

- CS 151
- Hash tables & Hash maps
- Friday, November 9, 12

Implementing Maps & Dictionaries

- idea 1: use a TreeMap, as you do on lab 6
- remove (key) -- returns and removes the key (and associated value)
- get(key) -- returns the value(s) associated with the given key
- put(key, value) -- associates the given value with the given key
- null(key, value) -- associates the given value with the given key

Operations are

- get is amortized (O(n) and asymptotic O(1).
- We want all of the put, get and remove operations to be O(1), but which will
  - guarantee(key) removekeys()
  - remove(key) -- returns and removes the key (and associated values)
  - get(key) -- returns the value(s) associated with the given key
  - put(key, value) -- associates the given value with the given key
  - null(key, value) -- associates the given value with the given key

Operations are

- get is amortized (O(n) and asymptotic O(1).
- We want all of the put, get and remove operations to be O(1), but which will
  - guarantee(key) removekeys()
  - remove(key) -- returns and removes the key (and associated values)
  - get(key) -- returns the value(s) associated with the given key
  - put(key, value) -- associates the given value with the given key
  - null(key, value) -- associates the given value with the given key

Operations are

- get is amortized (O(n) and asymptotic O(1).
- We want all of the put, get and remove operations to be O(1), but which will
  - guarantee(key) removekeys()
  - remove(key) -- returns and removes the key (and associated values)
  - get(key) -- returns the value(s) associated with the given key
  - put(key, value) -- associates the given value with the given key
  - null(key, value) -- associates the given value with the given key

Operations are

- get is amortized (O(n) and asymptotic O(1).
- We want all of the put, get and remove operations to be O(1), but which will
  - guarantee(key) removekeys()
  - remove(key) -- returns and removes the key (and associated values)
  - get(key) -- returns the value(s) associated with the given key
  - put(key, value) -- associates the given value with the given key
  - null(key, value) -- associates the given value with the given key

Operations are

- get is amortized (O(n) and asymptotic O(1).
- We want all of the put, get and remove operations to be O(1), but which will
  - guarantee(key) removekeys()
  - remove(key) -- returns and removes the key (and associated values)
  - get(key) -- returns the value(s) associated with the given key
  - put(key, value) -- associates the given value with the given key
  - null(key, value) -- associates the given value with the given key

Operations are

- get is amortized (O(n) and asymptotic O(1).
- We want all of the put, get and remove operations to be O(1), but which will
  - guarantee(key) removekeys()
  - remove(key) -- returns and removes the key (and associated values)
  - get(key) -- returns the value(s) associated with the given key
  - put(key, value) -- associates the given value with the given key
  - null(key, value) -- associates the given value with the given key

Operations are

- get is amortized (O(n) and asymptotic O(1).
- We want all of the put, get and remove operations to be O(1), but which will
  - guarantee(key) removekeys()
  - remove(key) -- returns and removes the key (and associated values)
  - get(key) -- returns the value(s) associated with the given key
  - put(key, value) -- associates the given value with the given key
  - null(key, value) -- associates the given value with the given key

Operations are
Implementing Maps & Dictionaries

**Idea #2:** Store elements in a list by their key, store them as (key, value) pairs.

- **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.

**Map** is such a 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.

- **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.

**Map** is such a 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 #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 get faster lookup, but it's a trade-off.

It's a hash table if:

- It's a set where a key maps to a value.
- It's a map where a key maps to a range of values.

**Note:** This idea doesn't support duplicates.

- **put(key, value)** --- O(1)
- **get(key)** --- O(1)
- **remove(key)** --- O(1)
- **size()** --- O(1)
- **isEmpty()** --- O(1)

**Hash tables**

- **Hash table**
  - A hash table is a set where each key maps to a value.
  - A hash table is a generalization of an array such that:
    - It's a set where a key maps to a value.
    - It's a map where a key maps to a range of values.
  - A hash table is a generalization of an array such that:
    - It's a set where a key maps to a range of values.

- **Maps & Dictionaries**
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 $index = h(key) \% S$
- Add $(key, value)$ to the bucket at $HT[index]$

**get(key):**
- Compute $index = h(key) \% S$
- Search through bucket at $HT[index]$ to see if entry with key exists

- RT is $O(1)$ (just add $(key, value)$ to the end of the list).
- RT is $O(size of bucket)$ 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, if we don't resize and we go looking for the $10$ or $12$, we won't find them.

If we don't resize and we go looking for the $10$ or $12$, we won't find them.

If we don't resize and we go looking for the $10$ or $12$, we won't find them.

In the worst case, all items are in one bucket, and get is $O(n)$. On average, if we don't resize and we go looking for the $10$ or $12$, we won't find them.

If, on a put, $n$ increases enough to make $\frac{n}{S}$ preset amount, resize the table (i.e. increase $S$) to decrease the load factor, rehash all the elements, and continue.

**Examples:** Suppose keys are ints, $S=7$, $h(n)=n$, and we want to keep the load factor less than $0.75$.

- put: $10, 1, 12, 4, 8$.
  - now we resize to $17$.

  - If we don't resize 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 enough.

  - Rehashing and resizing takes $O(n)$ time, but $O(1)$ happens every $O(n)$ puts if you double and resizing buckets ($O(n)$ time, but $O(1)$ happens every $O(n)$ puts if you double and resizing buckets ($O(n)$ time, but $O(1)$ happens every $O(n)$ puts if you double)

  - If we don't resize 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 enough.

  - Rehashing and resizing takes $O(n)$ time, but $O(1)$ happens every $O(n)$ puts if you double

  - In the worst case, all items are in one bucket, and get is $O(n)$. On average, if we don't resize and we go looking for the $10$ or $12$, we won't find them.

- 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 $n$ is small.

$\lambda := \frac{n}{S}$

- In the worst case, all items are in one bucket, and get is $O(n)$. On average, if we don't resize and we go looking for the $10$ or $12$, we won't find them.

Separate chaining allows each bucket to contain a list (chain) of values.

Separate Chaining