CS 151
Hash functions & Open addressing
Announcements

Prelab 8 is due now. Lab 8 due Sunday night.

Test #2 is next Wednesday (right before Thanksgiving break)
If you want to take the test on Tuesday, let me know so we can arrange it.
Topics: BSTs (incl. balanced BSTs), PQs, heaps, struct. ind., hashtables.

Structural Induction Bonus Session this Thursday 7:30-9pm. King 243.
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()

Want to implement put, get, and remove in amortized $O(1)$ time.

A **hash table** is an array-based structure that consists of 2 components:

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

Given a key, we put it into and get it from $HT[h(key) \mod S]$.

A **collision** occurs when hash function maps two keys to same bucket. Solutions:

- separate chaining – each bucket contains a list of its elements
- open addressing – to be discussed later.
Hash Functions

$h(key)$ should return a non-neg integer – you can use % S to get a valid bucket
every object in Java has a hashCode() method you can call or override

Every hash function must have the following property in order to work:
  if two entries have the same key, they MUST map to the same integer
  that is, the hash function must depend on the key alone

Therefore, if you ever override an object’s equals method, you should
override the hashCode so that it depends only on the variables the equals
method does.
  ex. suppose the Student class’ equals() checks equality of student ID
      Your hashcode should not depend on height, for example. If you change a
      student’s height, they should still have the same hashcode.

We do not require the converse: it is NOT necessary for two different keys to
map to two different integers (if so, we would never have collisions!)
(Of course, we would love to avoid collisions, but it’s hard to get both $O(1)$
hash function computation and collision avoidance without wasting space!)
Hash Functions

To design your own hash function for a class that represents a key:

• use 1 or 2 “standout” values, e.g. every nth char of string
  • pros: very fast, cons: oblivious to collisions, so you can get lots
• use the sum of values, e.g. the sum of the characters of a string
  • pros: fast, cons: still have collisions (e.g. “bat” and “tab”)
• use a polynomial function of the values, e.g. \( s[0]^2 + s[1] + 3s[3]^4 \)
  • pros: less collisions, cons: how do you choose your coefficients? primes.
• bitwise operations

In Java (fyi):

• `String s.hashCode()`: \( s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \cdots + s[n-2] \cdot 31 + s[n-1] \)
• `Integer i.hashCode()`: i’s integer value
• `Float f.hashCode()`: bits of internal representation, treated as an int
• `Double, long`: xor of both halves
• `List`: \( hc = 1 \)
  
  for( Object o : list )
  
  \( hc = 31 \cdot hc + (\text{obj} = \text{null} ? 0 : \text{obj}.hashCode()) \)
For your own objects that you are using as a key:

- override your equals() method so that you can check key equality
- override the hashcode() method to only use the fields used in equals()
  - Turn each such field into a String (cast it) and get its (String) hashcode()
  - sum (or, xor) the hashcode for the various fields.
- be sure to keep the computation quick (O(1))
Open addressing refers to restricting each bucket to contain at most 1 element. Use probing to find the next empty spot on a collision.

1. linear probing
   • idea: if \( h(k) \mod S = i \) and \( i \) already contains an element, move in one direction space-by-space until you find a free spot (i.e. try \( i+j \) for \( j=1,2,3,\ldots \))
   • when you go to get an item, you calculate \( h(k) \mod S = i \), then you look for the item in bucket \( i \), and then bucket \( i+1, i+2, \text{etc. until you either find it,} \)
     or you find an empty bucket (and thus can conclude the item not in table.)
   • ex:
   • problem: can get large clusters (aka “primary clusters”)
     • the larger the cluster, the longer insert & gets
     • also, when you remove an element, you can’t really remove it (otherwise you mess up subsequent elements’ finds!)
To reduce clustering effects, don’t just move linearly to find an open spot. Skip some spots, in a re-creatable way (so you can find your elements again!)

2. quadratic probing
   • idea: if \( h(k) \mod S = i \) and i already contains an element, move in one direction increasing by squares until you find a free spot (i.e. \( i+1, i+4, i+9, \ldots \))
   • now you don’t get long blocks of clusters merging, but you still get mini-clusters (aka “secondary clusters”) around the squares.
   • ex:

   • problem: if we’re not careful, we may not find an existing open spot
     • ex: \( S=4 \) with spots 0,1,2 filled. If \( h(key)=0 \), we’ll loop forever.
     • Thm (20.4): if \( S \) is prime and \( \lambda < \frac{1}{2} \) then quad. probing always finds an open spot
To reduce secondary clustering effects, move more arbitrarily.
Instead, define some function that tells you how to probe.

3. double hashing
   • idea: if \( h(k) \mod S = i \) and \( i \) already contains an element, move according to some function \( f(k) \) until you find a free spot (i.e. \( i+1f(k),i+2f(k),i+3f(k),... \))
   • i.e. you use a second hash function for collision resolution!
In summary, for open addressing we have the following pseudocode:

**put(key, value)**
- if the hashtable is too small (i.e. if load factor is too big), resize & rehash
- compute i = h(key)
  - add (key, value) to list/bucket at position i % S if using chaining
  - probe to find next open spot or deletion marker if using probing

**get(key)**
- compute i = h(key)
  - search through list/bucket at position i % S if using chaining
  - probe until find element or open spot if using probing

**remove(key)**
- compute i = h(key)
  - remove from list/bucket at position i % S if using chaining
  - probe until find element & put deletion marker if using probing

RT: O(1) if hash function and probing is good, and load factor small (< 1/2).