Outline

- open addressing
- chained hashing
- applications of hashing
- reminder: equality and comparisons in Java
- reminder: sorting
- reminder: selection sort
- reminder: bubble sort
- reminder: insertion sort

open addressing

- when inserting a value in a hash table, if the slot indicated by the hash function is full, insert it into another slot
- this works until the hash table is full (100% load factor), i.e. it works as long as there are open slots
- the *probe sequence* determines where to look next when there is a collision
- when looking up a value in a hash table, the same probe sequence must be followed as when inserting
- when removing a value from a hash table, the hash table must record that the element was removed, so future searches can keep looking when they reach the slot of a deleted element
- each slot must record whether it is empty, full, or deleted
- a slot can only be empty until the first time a value is inserted, after which it can only be full or deleted

probing in open addressing

- linear probing: look at subsequent locations for an open slot
- quadratic probing: instead of looking at the next location, on probe i look i^2 slots further
- double hashing: a second hash function determines the step size

linear probing (linear hashing)

- increment the index (modulo the array size) until a free slot is found
- if many keys hash to similar values, this may lead to long search times, as all those keys hash to (nearly) the same location

quadratic probing (quadratic hashing)

- add (modulo the array size) the square of the probe number to get the next index
- avoids the problem with similar key hashes
- for example,
  - hash 200 would probe locations 200, 201, 204, 209,
  - hash 201 would probe locations 201, 202, 205, 210, with overlap at only one index (201)
  - works well unless the hashes are identical

double hashing

- define two hash functions h_1 and h_2
- h_1(key) determines the initial slot to look into
- h_2(key) determines the step size for the next probe
- h_2(key) != 0
- even if h_1(key1) == h_1(key2), hopefully h_2(key1) != h_2(key2)
- for example, h_1(key) could be the sum of the letters in the string, and h_2(key) could be the sum of the product of the letters in the string with their position, plus 1
- even if two strings sum to the same number, the sum of products of letters and positions would almost certainly be different, as long as the two keys are different

open addressing table size

- load factor cannot exceed 100%, so the table must have at least as many slots as the number of stored elements
- if the table size is a prime number, linear hashing or double hashing will visit the entire table before giving up
- otherwise, for example double hashing in a table of size 100, with step size 10, can only visit 1/10th of the slots
- if the table size is ever changed, each element must be reinserted using the hash function and the new table size, since just copying the old array would map an element to the wrong index:

  24 modulo 11 = 2, but 24 modulo 23 = 1 -- no simple relationship without recomputing the hash value

in-class exercises
groups of 3-5

- what is the probing sequence if the hash table size is 11, and h(key) = 4? answer for each of linear addressing, quadratic addressing, and double hashing where $h_2(key) = 5$
- using h(key) = key mod table size, insert elements with key 56, 48, 40, 13 into a table of size 7
- remove the element with key 48
- locate the element with key 13

alternatives to open addressing

- chaining or chained hashing: each array element refers to a linked list of elements
- buckets: each array element can store up to a fixed number of elements
- either can accomodate load factors greater than 100%
- chaining is more flexible, but requires dynamic memory allocation
- in-class exercise: repeat the previous exercise using chained hashing

applications of hash tables

- hash tables can be used in any application where values (records) will be retrieved based on exact match with a given key
- for example, credit card companies might use a hash table to access their records for a given credit card
- and perhaps another hash table to locate a credit card number given a customer name
- a social networking site might use a hash table to find the information for a given person

using a hash table for a cache

- a cache (pronounced like "cash") is a collection of recently accessed items
- whenever we want to find an item, we first look in the cache, which is fast
- if the object is not there, we go to the underlying data structure, which may be slow but always finds the object
- then, we add to the cache the newly found object
- if the newly found object collides with something already in the cache, the old object can be replaced with the new one
- so in this hash table, operations always take O(1) time
  - we don’t need to use open addressing or chained hashing

Java equality reminder

- four kinds of equality:
  - `==` is true if and only if two references are to the *exact same object* (or for basic types)
  - `object1.equals(object2)` is `true` if object1's `equals` method (*in its wisdom*) decides they are the same
  - `object1.compareTo(object2)` is `0` if the `compareTo` method decides they are the same
  - `object1.compare(object2, object3)` is similar
- the behavior of the last three depends on the implementation:
  - ideally, they compare the *contents* of the object
  - ideally they are all consistent with each other

  but:
  - they may not do what you think
  - they may have bugs

properties of the equals method

- the equals method should be:
  - reflexive: a.equals(a) should always be true
  - symmetric: a.equals(b) == b.equal(a)
  - transitive: if a.equals(b) and b.equals(c) iff a.equals(c)
  - consistent: successive identical calls should return the same result
  - if a is not null, a.equals(null) == false
- the equals method can be overridden for any class
- if equals is not overridden, it defaults to ==

ordering objects in Java

- the mathematical function `Integer.signum` returns -1 if its argument is negative, 1 if it is positive, and 0 if it is zero
- in Java, there are two kinds of comparison:
- `Comparable<T>` interface, specifies the `int compareTo(T x)` method
- `Comparator<T>` interface, specifies the `int compare(T x1, T x2)` method (which could be static, but isn't)

compare and compareTo

- both comparison methods should have the following properties
  - symmetric: Integer.signum(compare(a, b)) == -Integer.signum(compare(b, a))
  - transitive: if (((compare(a, b) > 0) && ((compare(b, c) > 0)), then ((compare(a, c) > 0)
  - consistent: if (compare(a, b) == 0), then sgn(compare(a, c)) should be the same as sgn(compare(b, c))
- when building ordered linked lists, heaps, or when sorting, can use either interface, e.g. class Collections has two different methods for sorting,
  - static <T extends Comparable<? super T>> void sort(List<T> list)
  - static <T> void sort(List<T> list, Comparator<? super T> c)

applications of sorting

- finding duplicate elements in a list
  - eliminating duplicates: making a list of elements, where each element must occur at most once
- preparing for future binary search
  - dictionary, phone book
- presenting data in an appropriate format, e.g. for printing (bank statement sorted by date)
- comparing two lists to find out which elements are in one, the other, or both
- merging multiple sorted collections into a new sorted collection, with or without duplicates

in-class exercise

- sort by first name
- what algorithm did you use?

Java sorting

- Java has

```
public static void sort(x[] items);
public static void sort(x[] items, int
fromIndex, int toIndex);
```

- for type x being any one of:
  - `int`
  - `Object` -- the objects must be `Comparable`
  - `Comparator`, in which case the last argument is the comparator object
- as mentioned above, there is also a sort method declared on lists of objects that are either `Comparable`, or can be compared by a `Comparator.compare()` method
  - too much of a good thing might be confusing!

reminder: selection sort

- start with an unsorted array a
- find the smallest element in the array (at index i)
- swap the element at index i with the element at index 0
- now, find the smallest element in 1..a.length-1
- swap the elements at index i and at index 1
- now, the sub-array with elements 0..1 is sorted
- now, find the smallest element in 2..a.length – 1
- ...
- in-class exercise (in groups of 2 or 3): write code to implement selection sort
- in-class exercise: how long does selection sort take?

reminder: bubble sort

- start with an unsorted array a
- loop through all the elements of array a, swapping any that are not sorted
- repeat until the array is sorted
- in-class exercise: how long might this take? how long does it take in the best case? how do these compare to selection sort?
- historical trivia: if the data is stored on a magnetic tape, how many passes of the tape are needed to sort the entire tape? (this is assuming the tape can store much more data than the memory of the computer)
- current trivia: bubble sort can be done in parallel on n processors...

reminder: insertion sort

- in selection sort, find the smallest element, and put it in the next position
- in insertion sort, take the next element, and put it in the right place
- trivia: card players typically use insertion sort to arrange their decks
- the sub-array at the beginning of the array is already sorted, just as in selection sort
- the next element e is always taken from the next index in the array
- elements greater than e (in the sorted part of the array) are shifted up one position, to free the position where element e will be stored
- in-class exercise (in groups of 2 or 3): write code to implement insertion sort
- in-class exercise: how long does insertion sort take?