Outline

- hash tables
- hash functions
- open addressing
- chained hashing

hashing

- hash browns: mixed-up bits of potatoes, cooked together
- hashing: slicing up and mixing together
- a **hash function** takes a larger, perhaps variable amount of data and turns it into a fixed-size integer
- in-class exercise: why is this useful?

hash tables

- a collection class that under optimal conditions gives constant access time to elements in the collection
- how? simply put the elements in an array
- this only works if the key is a small integer, <= the size of the array
- hash functions take arbitrary keys and turn them into small integers
- so we can (a) use the hash of the key to (b) index the array to (c) find out in constant time if the element is present, and if so, (d) get its value
- a **perfect hash function** maps each key to a different array location
- in real life, perfect hash functions are hard to find, in part because the keys may not be known in advance

hash table example

- use the sum of the characters in a string as its hash
- use 1 for "a", 2 for "b", etc
- so the string "edo" hashes to 5 + 4 + 15 = 24
- the string "hello" hashes to 8 + 5 + 12 + 12 + 15 = 52
- the string "world" hashes to 23 + 15 + 18 + 12 + 4 = 72
- if the hash table has size 11, this hash function is a perfect hash function for these three strings:
  - 24 modulo 11 = 2, so "edo" is stored at index 2
  - 52 modulo 11 = 8, so "hello" is stored at index 8
  - 72 modulo 11 = 6, so "world" is stored at index 6

hash table observations

- in each case, computing the index takes time independent of both the table size and the number of elements in the table: O(1)
  - to be accurate, O(key size)
- supposing I wanted to use the same hash function on a table of size 3,
  - 24 modulo 3 = 0, so "edo" is stored at index 0
  - 52 modulo 3 = 1, so "hello" is stored at index 1
  - 72 modulo 3 = 0, so "world" is stored at index 0
- the first and last string now need to be stored in the same location, which is a **collision**
  - an array location can store at most one element!!!!

hash table collisions

- there are two main sources of collision in a hash table:
  - hash function collision: two different keys hash to the same integer
    - using the example function, “edo” and “doe” both hash to 24, which is an inevitable collision
      - the example function is not a very good hash function!
    - a hash function where collisions are really hard to find is useful as a **cryptographic hash function**
  - hash table index collision: two different integers, modulo the hash table length, give the same index
    - the same hash function with the same keys, with table size 11 has no collisions, with table size 3 has collisions
    - so resizing the hash table may change the number of hash table collisions (but not predictably)

ways to handle hash table collisions

- an array location can only hold one item, so what to do in case of a collision? (assume no collisions in the hash function)
- a number of solutions:
  - can store colliding elements (and only those elements) in a separate data structure (e.g. a linked list), which is searched when needed
  - can increase (e.g. double plus one) the size of the array until all collisions go away
  - can look for another place in the same array that is available. This is called **open address hashing**
  - can have each array element refer to a linked list rather than a single element: **chained hashing**
  - or combinations of the above
- in-class exercise: assuming few collisions, what is the average runtime of each of the above strategies?
- in-class exercise: assuming many collisions, what is the worst-case runtime of each of the above strategies?

Hash functions

- finding a perfect hash function is only possible if all keys are known in advance
- for random, evenly distributed keys, good hash functions produce random, evenly distributed hash codes, with a few collisions
- for non-random keys that resemble one another, good hash functions still produce random, evenly distributed hash codes
- so for example, using the first three digits of a telephone number (the area code) as a hash key would give many collisions if all the keys are from the same geographic area
- in-class exercise: using h(key) = key mod table size, insert elements with key 99, 43, 14, 77 into a table of size 10
- for a much more in-depth explanation of hash functions, see http://burtleburtle.net/bob/hash/doobs.html

  which includes a link to a more effective (and more complex) hash function:

  http://burtleburtle.net/bob/c/lookup3.c

Cryptographic Hash Functions

- for a hash table, the most important property of a hash function is that each key be mapped (as much as possible) to a different index
- knowing the hash function, a clever person can create a key that maps to a given index
- with some hash functions, this is hard
  - e.g. if changing any bit of the input changes about ½ the bits of the hash
- such hash functions are useful for identifying data
- checking the hash of an input (for example, a file) can then give assurance that the input has not been maliciously (or accidentally) modified
- SHA-2 cryptographic hashes give 224-bit, 256-bit, 384-bit, or 512-bit hash values
  - bitcoin uses SHA256
- SHA-3 was standardized in 2015, has the same hash sizes
- earlier cryptographic hash functions include SHA-1, SHA-0, MD5 (and similar)
  - for these (now largely obsolete) hash functions, it is possible for an attacker to create a document that has a specific hash value

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