New data structure: Hash table

- Element's index in the table is a function of its value
- Advantage: fast lookup, fast indexing
  - $O(1)$!
- Trick: getting unique values from the keys, or storing keys in unique locations, or both
- Typically, table = array

Data structures, so far

- Linked lists: refer to element starting from the head of the list
- Arrays: refer to element by its numerical position in the memory block
- Queues and stacks: refer to element by its position relative to the start/top
- Trees: refer to element by its relatives (parent and children)

Definitions

- Hash function: maps a key, $k$, to an index (address) in the table
- Perfect hash function: function that maps each key to a unique index
- Collision: occurs when more than one key maps to the same index
A diversion: Bit operations

- XOR: exclusive-OR
  - 0 XOR 1 = 1 XOR 0 = 1
  - 0 XOR 0 = 1 XOR 1 = 0
- In Java: ^ operator

- Shifting:
  - >> right shift  e.g. 11011 >> 1 = 01101
  - << left shift  e.g. 11011 << 1 = 10110

- Masking: use &
  - e.g. 11011010 & 11110000 = 11010000

More on hash functions

- If we have \( n \) items in a table with \( m \) positions, we have \( m^n \) possible hash functions!
- Q: How many of these are perfect hash functions?
  - \( m \) items taken \( n \) at a time: \[ P_{\frac{m}{n}} = \frac{m!}{(m-n)!} \]
  - perfect hash functions are pretty rare

- Several types:
  - division functions
  - folding functions
  - mid-square functions
  - extraction functions
  - radix functions

Hashing by division

- Actually modulo division
- Use size of table as modulus to ensure unique indices (or a prime number greater than the table's size)
- Most other methods utilize modulo division too
- Advantages:
  - simple
  - useful if we don't know much about the keys

Hashing by folding

- Idea: divide the key into parts, then combine ("fold") the parts to create the index
- Shift folding: parts are placed underneath one another, then added
- Boundary folding: same as shift folding, except that every other part is written backwards
- Usually followed by modulo division
Example

- Key is 23459087632
  - Divide into parts: 234 590 876 32
  - Shift folding: 234 + 590 + 876 + 32 = 1732
  - Boundary folding: 234 + 095 + 876 + 23 = 1228

Bit-wise folding

- Replace addition in previous examples with XOR
- Can be applied to strings, etc. too
- Advantage: bit operations can be fast

Hashing by computing the mid-square

- Idea: square the key, use the “middle” as the address
- Example: 96012 -> 9218304144 -> 8304 is hash
- Advantage: entire key is used to calculate the address, reducing chances of collisions
- Can do bit-wise w/ a mask and shifting

Hashing by extraction

- Idea: use only part of the key to compute the hash
- Example: phone numbers in the same city (can omit the area code)
- Advantages: potentially shorter calculation, remove similar data that won’t affect the hash anyway
Hashing by radix transformation

- Idea: convert key into another number base, then divide (modulo)
- Example: convert 23456 to base 7 --> use this as the hash
- May cause collisions!

Detecting and resolving collisions

- Even with the methods introduced previously, collisions may still occur
- We cannot hash two keys to the same location, so we must find a way to resolve collisions
- Choice of hash function and choice of table size may reduce collisions, but will not eliminate them
- Methods for resolving collisions:
  - open addressing: find another empty position
  - chaining: use linked lists
  - bucket addressing: store elements at same location

Open addressing

- Idea: find the next open slot in the table, and place the key there
  - use a “probing function”, \( p(i) \)
  - search the sequence \( h(K) + p(1), h(K) + p(2), \ldots \)
- Several methods:
  - linear probing
  - quadratic probing
  - random probing

Linear probing

- \( p(i) = i \)
- try \( h(K) + 1, h(K) + 2, \ldots \)
- if end of the table reached, wrap around, and continue searching from the beginning of the table until a spot is found or the table is determined to be full
- Advantage: easy
- Disadvantage: creates clusters in the table
Quadratic probing

- \( p(i) = \pm i^2 \)
- try \( h(K) + i^2, h(K) - i^2 \), for \( i=1, 2, \ldots, (\text{size}/2) \)
- Advantage: clustering is diminished
- Disadvantages:
  - if table size is not a prime number, probes will not try all locations in the table
  - may cause secondary clustering (smaller clusters not around the primary key)

Random probing

- \( p(i) = \) some random number sequence
- Use key as the seed for the random number generator
  - can pass as a parameter to `java.util.Random`'s constructor
  - ensures that we will get the same sequence of random numbers for a key
- Reduces secondary clustering

Double hashing

- \( p(i) = i \cdot h_p(K) \)
- try \( h(K) + h_p(K), h(K) + 2h_p(K), \ldots, h(K) + ih_p(K) \)
- Advantage: eliminates secondary clustering in most cases
- Disadvantage: can be time-consuming to compute two hash functions

Efficiency of open addressing

- Depends heavily on the load factor (LF)
  - \( LF = (# \text{ of elements in table})/(\text{table size}) \)
- Interested in both successful and unsuccessful searches
  - because we use collision detection, we might have to look in more than one place for the key
Approximate efficiencies of open addressing methods

- **Linear probing:**
  - success: \[0.5 \left(1 + \frac{1}{1-LF}\right)\]
  - failure: \[0.5 \left(1 + \frac{1}{(1-LF)^2}\right)\]

- **Quadratic probing:**
  - success: \[1 - \ln(1-LF) - 0.5LF\]
  - failure: \[\frac{1}{1-LF} - LF - \ln(1-LF)\]

- **Double hashing:**
  - success: \[\frac{1}{LF} \ln(\frac{1}{1-LF})\]
  - failure: \[\frac{1}{1-LF}\]

Chaining

- Alternative to open addressing
- Idea: allow keys with the same hash to be stored at the same address
  - keys are stored in a linked list
  - address contains pointer to the linked list
- Advantages
  - do not have to search the table
  - can store as many keys as we want at one address
- Disadvantage: linked list operations necessary (insertion, searching, deletion)

Bucket addressing

- Idea: allocate a large block of space at each address of the table, to store keys with the same hash value
- Any overflows can be handled by open addressing or an overflow bucket (linked list)
- Advantage: can (typically) avoid the linked list operations
- Disadvantage: may waste lots of space

Deleting objects from a hash table

- Chaining: just remove node from linked list
- Open addressing: ???
  - if we delete the key at its primary address, but there are keys that hash to the same address stored in other locations in the table, then searches for those keys will come back as unsuccessful!
- Solution: lazy deletion
  - keep the key in the table, but “mark” it as deleted
  - new keys overwrite the deleted keys
Perfect hash functions

- These are rare
- BUT, if the data set is well-defined and known (known size, known keys), we can *design* a hash function for it
  - known data sets: table of reserved words (compilers), dictionary files, ...
- *minimal perfect hash function:* designed hash function that is perfect and results in a full table (i.e., no wasted space in the table)

Hashing in Java

- Java has a built-in `Hashtable` class
- Can specify load factor and capacity (default: 0.75 and 101, respectively) when calling constructor
- Implemented with bucket addressing
- Key objects must implement `hashCode()` method (Java API classes all implement this method already)

Applications of hash tables

- Lots of recent research into using *distributed hash tables* in peer-to-peer networks (searching, lookup)
- Symbol tables (compilers)
- Databases (of phone numbers, IP addresses, etc.)
- Dictionaries