Hash tables
- HashSet & HashMap
- hash table, hash function
- collisions
  - linear probing, lazy deletion, primary clustering
  - quadratic probing, rehashing
  - chaining

HashSet & HashMap
recall: TreeSet & TreeMap use an underlying binary search tree (actually, a red-black tree) to store values
- as a result, add/put, contains/get, and remove are $O(\log N)$ operations
- iteration over the Set/Map can be done in $O(N)$

the other implementations of the Set & Map interfaces, HashSet & HashMap, use a "magic" data structure to provide $O(1)$ operations*
*legal disclaimer: performance can degrade to $O(N)$ under bad/unlikely conditions however, careful setup and maintenance can ensure $O(1)$ in practice

the underlying data structure is known as a Hash Table
Hash tables

A hash table is a data structure that supports constant time insertion, deletion, and search on average.
- Degenerative performance is possible, but unlikely.
- It may waste some storage.
- Iteration order is not defined (and may even change over time).

Idea: Data items are stored in a table, based on a key.
- The key is mapped to an index in the table, where the data is stored/accessed.

Example: Letter frequency.
- Want to count the number of occurrences of each letter in a file.
- Have an array of 26 counters, map each letter to an index.
- To count a letter, map to its index and increment.

```
<table>
<thead>
<tr>
<th>Letter</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Z</td>
<td>25</td>
</tr>
</tbody>
</table>
```

Mapping examples

Extension: Word frequency.
- Must map entire words to indices, e.g.,

```
<table>
<thead>
<tr>
<th>Word</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>AA</td>
<td>26</td>
</tr>
<tr>
<td>BA</td>
<td>52</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Z</td>
<td>25</td>
</tr>
<tr>
<td>AZ</td>
<td>51</td>
</tr>
<tr>
<td>BZ</td>
<td>77</td>
</tr>
</tbody>
</table>
```

- Problem?

Mapping each potential item to a unique index is generally not practical.
- Even if you limit words to at most 8 characters, need a table of size 217,180,147,158.
- For any given file, the table will be mostly empty!
Table size < data range

since the actual number of items stored is generally MUCH smaller than the number of potential values/keys:

- can have a smaller, more manageable table

  e.g., table size = 26
  possible mapping: map word based on first letter

  "A"* → 0
  "B"* → 1
  ...
  "Z"* → 25

  e.g., table size = 1000
  possible mapping: add ASCII values of letters, mod by 1000

  "AB" → 65 + 66 = 131
  "BANANA" → 66 + 65 + 78 + 65 + 78 + 65 = 417
  "BANANABANANABANANA" → 417 + 417 + 417 = 1251 % 1000 = 251

- POTENTIAL PROBLEMS?

Collisions

the mapping from a key to an index is called a hash function

- the hash function can be written independent of the table size
- if it maps to an index > table size, simply wrap-around (i.e., index % tableSize)

since |range(hash function)| < |domain(hash function)|, can have multiple items map to the same index (i.e., a collision)

"ACT" → 67 + 65 + 84 = 216
"CAT" → 67 + 65 + 84 = 216

techniques exist for handling collisions, but they are costly (LATER)
it's best to avoid collisions as much as possible – HOW?

- want to be sure that the hash function distributes the key evenly

- e.g., "sum of ASCII codes" hash function
  OK if table size is 1000
  BAD if table size is 10,000
  most words are <= 8 letters, so max sum of ASCII codes = 1,016
  so most entries are mapped to first 1/10th of table
Better hash function

A good hash function should

- produce an even spread, regardless of table size
- take order of letters into account (to handle anagrams)

- the hash function used by `java.util.String` multiplies the ASCII code for each character by a power of 31

\[
\text{hashCode()} = \text{char}_0 \times 31^{(\text{len}-1)} + \text{char}_1 \times 31^{(\text{len}-2)} + \text{char}_2 \times 31^{(\text{len}-3)} + \ldots + \text{char}_{\text{len}-1}
\]

where

\[
\text{len} = \text{this.length()}, \ \text{char}_i = \text{this.charAt}(i);
\]

```java
/**
 * Hash code for java.util.String class
 * @return an int used as the hash index for this string
 */
private int hashCode() {
    int hashIndex = 0;
    for (int i = 0; i < this.length(); i++) {
        hashIndex = (hashIndex*31 + this.charAt(i));
    }
    return hashIndex;
}
```

Word frequency example

Returning to the word frequency problem

- pick a hash function
- pick a table size
- store word & associated count in the table
- as you read in words, map to an index using the hash function
  - if an entry already exists, increment
  - otherwise, create entry with count = 1

What about collisions?
Linear probing

linear probing is a simple strategy for handling collisions
- if a collision occurs, try next index & keep looking until an empty one is found
  (wrap around to the beginning if necessary)

assume naïve "first letter" hash function

- insert "BOO"
- insert "COO"
- insert "BOW"
- insert "BAZ"
- insert "ZOO"
- insert "ZEBRA"

Linear probing (cont.)

with linear probing, will eventually find the item if stored, or an empty space to add it (if the table is not full)

what about deletions?
- delete "BIZ"

can the location be marked as empty?

can't delete an item since it holds a place for the linear probing
- search "COO"
Lazy deletion

when removing an entry
- mark the entry as being deleted (i.e., mark location)
- subsequent searches must continue past deleted entries (probe until desired item or an empty location is found)
- subsequent insertions can use deleted locations

| ADD "BOO" | 0 |
| ADD "AND" | 1 |
| ADD "BIZ" | 2 |
| ADD "COO" | 3 |
| DELETE "BIZ" | 4 |
| SEARCH "COO" | 5 |
| ADD "COW" | 6 |
| SEARCH "COO" | 7 |

Primary clustering

in practice, probes are not independent
- suppose table is half full
  - maps to 4-7 require 1 check
  - map to 3 requires 2 checks
  - map to 2 requires 3 checks
  - map to 1 requires 4 checks
  - map to 0 requires 5 checks
  - average = 18/8 = 2.25 checks

using linear probing, clusters of occupied locations develop
- known as primary clusters

insertions into the clusters are expensive & increase the size of the cluster
Analysis of linear probing

the load factor $\lambda$ is the fraction of the table that is full

empty table  $\lambda = 0$  half full table  $\lambda = 0.5$  full table  $\lambda = 1$

THEOREM: assuming a reasonably large table, the average number of locations examined per insertion (taking clustering into account) is roughly $(1 + 1/(1-\lambda^2))/2$

- empty table: $(1 + 1/(1 - 0)^2)/2 = 1$
- half full: $(1 + 1/(1 - .5)^2)/2 = 2.5$
- 3/4 full: $(1 + 1/(1 - .75)^2)/2 = 8.5$
- 9/10 full: $(1 + 1/(1 - .9)^2)/2 = 50.5$

as long as the hash function is fair and the table is not too full, then inserting, deleting, and searching are all $O(1)$ operations

Rehashing

it is imperative to keep the load factor below 0.75

if the table becomes three-quarters full, then must resize
- create new table at least twice as big
- just copy over table entries to same locations???
- NO! when you resize, you have to rehash existing entries
  new table size $\rightarrow$ new hash function (+ different wraparound)

LET hashCode = word.length()

ADD "UP"
ADD "OUT"
ADD "YELLOW"

NOW
RESIZE
AND
REHASH
Chaining

there are variations on linear probing that eliminate primary clustering
  - e.g., quadratic probing increases index on each probe by square offset

\[
\text{Hash(key)} \rightarrow \text{Hash(key)} + 1 \rightarrow \text{Hash(key)} + 4 \rightarrow \text{Hash(key)} + 9 \rightarrow \text{Hash(key)} + 16 \rightarrow \ldots
\]

however, the most commonly used strategy for handling collisions is chaining
  - each entry in the hash table is a bucket (list)
  - when you add an entry, hash to correct index then add to bucket
  - when you search for an entry, hash to correct index then search sequentially

Analysis of chaining

in practice, chaining is generally faster than probing
  - cost of insertion is \(O(1)\) – simply map to index and add to list
  - cost of search is proportional to number of items already mapped to same index
    e.g., using naïve “first letter” hash function, searching for “APPLE” might requires traversing a list of all words beginning with ‘A’

if hash function is fair, then will have roughly \(\lambda/\text{tableSize}\) items in each bucket
  \(\Rightarrow\) average cost of a successful search is roughly \(\lambda/(2*\text{tableSize})\)

chaining is sensitive to the load factor, but not as much as probing – WHY?
HashSet & HashMap

java.util.HashSet and java.util.HashMap use chaining

- e.g., HashSet<String>
- HashMap<String, Integer>

- defaults: table size = 16, max capacity before rehash = 75%
  - can override these defaults in the HashSet/HashMap constructor call
- a default hash function is defined for every Object (based on its address)
- a class can define its own hash function by overriding hashCode
  - must ensure that obj1.equals(obj2) \( \Rightarrow \) obj1.hashCode() == obj2.hashCode()