Back to Simple Search

• Linear search is OK for small data sets, bad for large.
• So linear search would be OK if we could rapidly narrow the search to a few items.
• Suppose that in constant time could put any item in our data set into a numbered bucket, where # buckets stays within a constant factor of # keys.
• Suppose also that buckets contain roughly equal numbers of keys.
• Then search would be constant time.
Hash functions

• To do this, must have way to convert key to bucket number: a hash function.

  “hash /hæʃ/ 2 a a mixture; a jumble. b a mess.” Concise Oxford Dictionary, eighth edition

• Example:
  - \( N = 200 \) data items.
  - keys are longs, evenly spread over the range \( 0..2^{63} - 1 \).
  - Want to keep maximum search to \( L = 2 \) items.
  - Use hash function \( h(K) = K \% M \), where \( M = N/L = 100 \) is the number of buckets: \( 0 \leq h(K) < M \).
  - So 100232, 433, and 10002332482 go into different buckets, but 10, 400210, and 210 all go into the same bucket.
External chaining

- Array of $M$ buckets.
- Each bucket is a list of data items.

![Diagram showing external chaining]

- Not all buckets have same length, but average is $N/M = L$, the \textit{load factor}.
- To work well, hash function must avoid \textit{collisions}: keys that “hash” to equal values.
Ditching the Chains: Open Addressing

- Idea: Put one data item in each bucket.
- When there is a collision, and bucket is full, just use another.
- Various ways to do this:
  - Linear probes: If there is a collision at $h(K)$, try $h(K) + m$, $h(K) + 2m$, etc. (wrap around at end).
  - Quadratic probes: $h(K) + m$, $h(K) + m^2$, …
  - Double hashing: $h(K) + h'(K)$, $h(K) + 2h'(K)$, etc.
- Example: $h(K) = K \% M$, with $M = 10$, linear probes with $m = 1$.
  - Add 1, 2, 11, 3, 102, 9, 18, 108, 309 to empty table.

| 108 | 1 | 2 | 11 | 3 | 102 | 309 | 18 | 9 |

- Things can get slow, even when table is far from full.
- Lots of literature on this technique, but
- Personally, I just settle for external chaining.
Filling the Table

• To get (likely to be) constant-time lookup, need to keep #buckets within constant factor of #items.

• So resize table when load factor gets higher than some limit.

• In general, must re-hash all table items.

• Still, this operation constant time per item,

• So by doubling table size each time, get constant amortized time for insertion and lookup

• (Assuming, that is, that our hash function is good).
Hash Functions: Strings

• For String, "s_0s_1 \cdots s_{n-1}" want function that takes all characters and their positions into account.

• What’s wrong with $s_0 + s_1 + \ldots + s_{n-1}$?

• For strings, Java uses

$$h(s) = s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \ldots + s_{n-1}$$

computed modulo $2^{32}$ as in Java int arithmetic.

• To convert to a table index in $0..N - 1$, compute $h(s) \% N$ (but don’t use table size that is multiple of 31!)

• Not as hard to compute as you might think; don’t even need multiplication!

```java
int r; r = 0;
for (int i = 0; i < s.length (); i += 1)
    r = (r << 5) - r + s.charAt (i);
```
**Hash Functions: Other Data Structures I**

- **Lists** (ArrayList, LinkedList, etc.) are analogous to strings: e.g., Java uses

  ```java
  hashCode = 1; Iterator i = list.iterator();
  while (i.hasNext()) {
    Object obj = i.next();
    hashCode =
      31*hashCode
    + (obj==null ? 0 : obj.hashCode());
  }
  ```

  - Can limit time spent computing hash function by not looking at entire list. For example: look only at first few items (if dealing with a `List` or `SortedSet`).
  - Causes more collisions, but does **not** cause equal things to go to different buckets.
Hash Functions: Other Data Structures II

- Recursively defined data structures $\Rightarrow$ recursively defined hash functions.

- For example, on a binary tree, one can use something like

```python
hash(T):
    if (T == null)
        return 0;
    else return someHashFunction (T.label ()) ^ hash(T.left ()) ^ hash(T.right ());
```
Identity Hash Functions

- Can use address of object ("hash on identity") if distinct (!=) objects are never considered equal.

- But careful! Won’t work for Strings, because .equals Strings could be in different buckets:

  ```java
  String H = "Hello",
  S1 = H + ", world!",
  S2 = "Hello, world!";
  ```

- Here S1.equals(S2), but S1 != S2.
What Java Provides

- **In class** `Object`, is function `hashCode()`.
- **By default,** returns the identity hash function, or something similar. [Why is this OK as a default?]
- **Can override it** for your particular type.
- **For reasons** given on last slide, is overridden for type `String`, as well as many types in the Java library, like all kinds of `List`.
- **The types** `Hashtable`, `HashSet`, and `HashMap` use `hashCode` to give you fast look-up of objects.

```java
HashMap<KeyType,ValueType> map =
    new HashMap<>(approximate size, load factor);
map.put(key, value);       // Map KEY -> VALUE.
... map.get(someKey)       // VALUE last mapped to by SOMEKEY.
... map.containsKey(someKey) // Is SOMEKEY mapped?
... map.keySet()           // All keys in MAP (a Set)
```
Special Case: Monotonic Hash Functions

• Suppose our hash function is *monotonic*: either nonincreasing or nondecreasing.

• So, e.g., if key $k_1 > k_2$, then $h(k_1) \geq h(k_2)$.

• Example:
  - Items are time-stamped records; key is the time.
  - Hashing function is to have one bucket for every hour.

• In this case, you *can* use a hash table to speed up range queries [How?]

• Could this be applied to strings? When would it work well?
Perfect Hashing

- Suppose set of keys is *fixed*.
- A tailor-made hash function might then hash every key to a different value: *perfect hashing*.
- In that case, there is no search along a chain or in an open-address table: either the element at the hash value is or is not equal to the target key.
- For example, might use first, middle, and last letters of a string (read as a 3-digit base-26 numeral). Would work if those letters differ among all strings in the set.
- Or might use the Java method, but tweak the multipliers until all strings gave different results.
Characteristics

• Assuming good hash function, add, lookup, deletion take $\Theta(1)$ time, amortized.

• Good for cases where one looks up equal keys.

• Usually bad for range queries: “Give me every name between Martin and Napoli.” [Why?]

• Hashing is probably not a good idea for small sets that you rapidly create and discard [why?]
Comparing Search Structures

Here, $N$ is #items, $k$ is #answers to query.

<table>
<thead>
<tr>
<th>Function</th>
<th>Unordered List</th>
<th>Sorted Array</th>
<th>Bushy Search Tree</th>
<th>“Good” Hash Table</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>$find$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(lg N)$</td>
<td>$\Theta(lg N)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(N)$</td>
</tr>
<tr>
<td>$add$ (amortized)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(lg N)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(lg N)$</td>
</tr>
<tr>
<td>$range$ $query$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(k + lg N)$</td>
<td>$\Theta(k + lg N)$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(N)$</td>
</tr>
<tr>
<td>$find$ $largest$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(lg N)$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>$remove$ $largest$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(lg N)$</td>
<td>$\Theta(N)$</td>
<td>$\Theta(lg N)$</td>
</tr>
</tbody>
</table>