CS61B Lecture #22: Hashing

Speeding Up Searching

- 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 we 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 in number of comparisons.

Hash functions

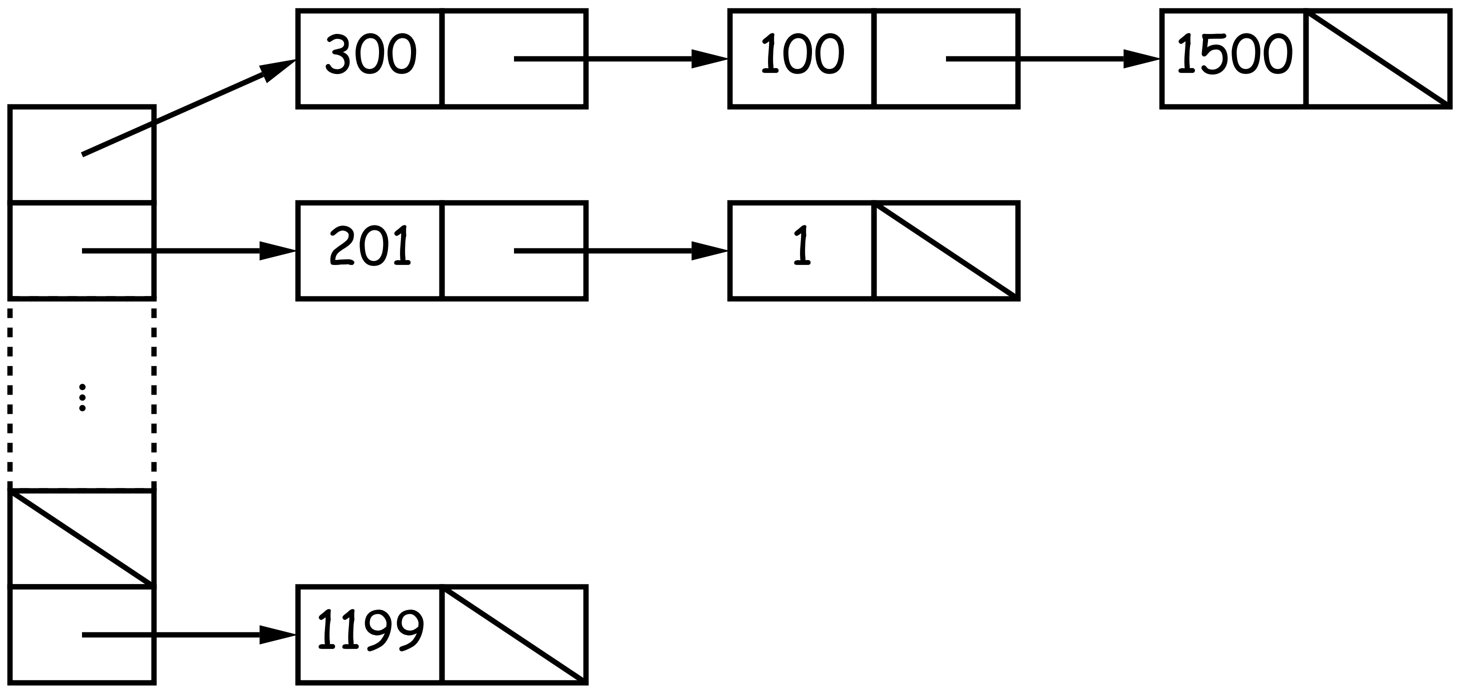
- To do this, must have way to convert key to bucket number: a *hash function*.

  "**hash** /hæʃ/ *2a* 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: $0 \le h(K) < M$.
  - 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.



- Not all buckets have same length, but average is $N/M = L$, the *load factor*.
- To work well, hash function must avoid too many *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.
- So you actually use a hash function with two arguments: $h(K, i)$, where K is the key, and the buckets you try are $h(K, 0), h(K, 1), \ldots$.
- Various possibilities (here S is the table size):
  - Linear probes: $h(K, i) = (h(K, 0) + C \cdot i) \bmod S$, for constant C.
  - Quadratic probes: $h(K, i) = (h(K, 0) + C_1 \cdot i + C_2 \cdot i^2) \bmod S$.
  - Double hashing: $h(K, i) = (h(K, 0) + i \cdot h'(K) \bmod S$, where h' is a different hash function.
- Example: $h(K, i) = (K + i)\% M$, with $M = 10$:
  - 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).

Sets vs. Dictionaries

- So far, we've really been talking about hash tables containing *sets of keys*. The important operations are testing for membership, adding keys, and iterating through the keys.
- The term "hash table" can also refer to a *dictionary*, in which each key is associated with a *value*.
- The associated value does not affect in the search process.
- The important operations are testing for whether a key is in the table, finding what value is associated with a given key, and setting the value associated with a key.
- Here, we'll concentrate on sets of keys, since the addition of associated values is a conceptually small change.

Hash Functions: Strings

- For String, "$s_0 s_1 \cdots s_{n-1}$" 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!

```
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 analagous to strings: e.g., Java uses

```
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

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

Identity Hash Functions

- We can use the address of object as its hashed value ("hash on identity") if distinct (`!=`) objects are never considered equal.
- But careful! Won't work for Strings, because `.equal` Strings could be in different buckets:

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

- Here `S1.equals(S2)`, but `S1 != S2`.
- Java provides the type `java.util.IdentityHashMap` for this purpose:

```
IdentityHashMap<Object, String> map = new IdentityHashMap<>();
Thing x = ...;
map.put(x, "Thing #1");     // Works if Thing does not override .equals.
map.put("Frank", "President"); // Probably not a good idea.
```

What Java Provides

- The class `Object` defines 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, it 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.

```
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 nondescreasing.
- 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.
- One technique: use a hash table of hash tables with two hash functions. One chooses a small hash table that uses the second hash function. With a few random selections of hash functions, you can soon find one where there is never a collision in the second hash table.
- For example, suppose we store key/value pairs in type `Pair`. Then:

```
Pair[][] table = ...;
// Find which table X must be in:
int h0 = hash0(X);
// Use different hash function for each h0.
Pair[] pair = table[h0][hash2(h0, X)];
return pair.key.equals(X) ? pair.value : null;
```

Characteristics

- Assuming good hash function, add, lookup, deletion take $\Theta(1)$ time, amortized.
- Good for cases where one looks up *equal* keys.
- For range queries such as "Give me every name between Martin and Napoli," hash tables are usually bad. [Why?]
- Hashing is probably not a good idea for small sets that you rapidly create and discard. [Why?]