

## Back to Simple Search

is OK for small data sets, bad for large.  
 Search would be OK if we could rapidly narrow the search space.

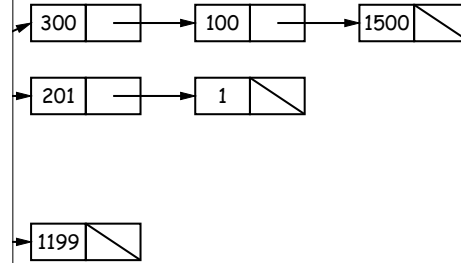
Search in constant time could put any item in our data set into a bucket, where # buckets stays within a constant factor

of buckets that buckets contain roughly equal numbers of keys. Search would be constant time.

## External chaining

buckets.

Each bucket is a list of data items.



Items in buckets have same length, but average is  $N/M = L$ , the load factor

of buckets. Hash function must avoid collisions: keys that "hash" to the same bucket.

## Filling the Table

Search (to be) constant-time lookup, need to keep #buckets proportional to constant factor of #items.

Search is possible when load factor gets higher than some limit.

Search must re-hash all table items.

Search operation constant time per item,

but increasing table size each time, get constant amortized time per item and lookup

Search (that is, that our hash function is good).

## CS61B Lecture #24: Hashing

### Hash functions

Search must have way to convert key to bucket number: a hash function

Search is like "a mixture; a jumble. b a mess." Concise Oxford Dictionary, eighth edition

Search data items.

Search items are long, evenly spread over the range  $0..2^{63} - 1$ .

Search keep maximum search to  $L = 2$  items.

Search hash function  $h(K) = K \% M$ , where  $M = N/L = 100$  is the number of buckets:  $0 \leq h(K) < M$ .

Search 2, 433, and 10002332482 go into different buckets, but 10210, and 210 all go into the same bucket.

### Choosing the Chains: Open Addressing

Search one data item in each bucket.

Search if there is a collision, and bucket is full, just use another.

Search how to do this:

Search probes: If there is a collision at  $h(K)$ , try  $h(K)+m$ ,  $h(K)+2m$ , ... (wrap around at end).

Search probes:  $h(K) + m$ ,  $h(K) + m^2$ , ...

Search probing:  $h(K) + h'(K)$ ,  $h(K) + 2h'(K)$ , etc.

Search  $h(K) = K \% M$ , with  $M = 10$ , linear probes with  $m = 1$ .

Search 11, 3, 102, 9, 18, 108, 309 to empty table.

2	11	3	102	309		18	9
---	----	---	-----	-----	--	----	---

Search search is not so fast, even when table is far from full.

Search search alternative on this technique, but

Search search just settle for external chaining.

## Functions: Other Data Structures I

List, LinkedList, etc.) are analogous to strings: e.g.,

```
int i = 1; Iterator i = list.iterator();
while (i.hasNext()) {
    Object obj = i.next();
    // ...
    int hashCode = obj == null ? 0 : obj.hashCode();
}
```

Don't spend computing hash function by not looking at entire object: look only at first few items (if dealing with a List or String).

Hash collisions, but does *not* cause equal things to go to different buckets.

## Identity Hash Functions

Hash of object ("hash on identity") if distinct (!=) objects are considered equal.

Won't work for Strings, because .equals Strings could be in different buckets:

```
"Hello",
"Hello, world!",
"Hello, world!";
```

hash(S1) != hash(S2), but S1 != S2.

## Special Case: Monotonic Hash Functions

A hash function is *monotonic*: either nonincreasing or nondecreasing.

For any  $k_1 > k_2$ , then  $h(k_1) \geq h(k_2)$ .

Example: time-stamped records; key is the time.

Function is to have one bucket for every hour.

For example, you *can* use a hash table to speed up range queries.

How is it applied to strings? When would it work well?

## Hash Functions: Strings

" $s_0s_1 \dots s_{n-1}$ " want function that takes all characters and their positions into account.

How about with  $s_0 + s_1 + \dots + s_{n-1}$ ?

Java uses

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

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

For a table index in  $0..N-1$ , compute  $h(s) \% N$  (but *don't* use that if  $N$  is not a multiple of  $31!$ )

How to compute as you might think; don't even need multiplication.

```
int i = 0;
for (int i = 0; i < s.length(); i += 1)
    r = (r << 5) - r + s.charAt(i);
```

## 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

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

## What Java Provides

Object, is function hashCode().

hashCode() returns the identity hash function, or something similar. (OK as a default?)

Override it for your particular type.

hashCode() given on last slide, is overridden for type String, as well as in the Java library, like all kinds of List.

HashSet, HashSet, and HashMap use hashCode to give a unique key for each object.

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

## Characteristics

Good hash function, add, lookup, deletion take  $\Theta(1)$  time,

cases where one looks up *equal* keys.

For range queries: "Give me every name between Martin [Why?]"

Probably not a good idea for small sets that you rapidly discard [why?]

## Perfect Hashing

Number of keys is *fixed*.

A perfect hash function might then hash every key to a different slot. This is called *perfect hashing*.

With perfect hashing, there is no search along a chain or in an open-address hash table. The element at the hash value is or is not equal to the key.

One might use first, middle, and last letters of a string to form a 3-digit base-26 numeral. Would work if those letters were unique for all strings in the set.

Similar to the Java method, but tweak the multipliers until all keys have different results.

## Comparing Search Structures

Time complexity,  $k$  is #answers to query.

	Unordered List	Sorted Array	Bushy Search Tree	"Good" Hash Table	Heap
add	$\Theta(N)$	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(N)$
lookup	$\Theta(1)$	$\Theta(N)$	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(\lg N)$
delete	$\Theta(N)$	$\Theta(k + \lg N)$	$\Theta(k + \lg N)$	$\Theta(N)$	$\Theta(N)$
range query	$\Theta(N)$	$\Theta(1)$	$\Theta(\lg N)$	$\Theta(N)$	$\Theta(1)$
insert	$\Theta(N)$	$\Theta(1)$	$\Theta(\lg N)$	$\Theta(N)$	$\Theta(\lg N)$