Hash Tables

Set ADT:

boolean find (ElementType x)
void insert (ElementType x)
void remove (ElementType x)

Dictionary ADT:

AnotherType find (ElementType x)
void insert (ElementType x, AnotherType y)
void remove (ElementType x)

Here the dictionary acts as a function or mapping:
find(x) = y

Sets or dictionaries maybe implemented as:

- Linked lists (inefficient)
- Hash tables
- Search trees

Hash tables are based on arrays

An array maps an index to a location
This index is an integer in a bounded range

A hash table uses a *hash function* to map a key to a location
This key can be an unbounded integer, or even a non-integer type such as string or float
The same hash function must be used during all operations (find, insert, and remove)

Hash function: $\{keys\} \rightarrow \{locations\}$

Examples of hash functions

If keys are unbounded integers:

```
H(key) = key % N
int H (int key) { return key % N; }
N = size of hash table (often prime number)
```

If keys are strings:

```
int H (string key) {
    int sum = 0;
    for (int j=0; j<key.length; j++)
        sum = (C*sum + int(key[j]) ) % N;
    }
}
where C = some constant
```

Ideally compute index = H(key) and then access the location or slot in the hash table: Table[index]

However, collisions can occur:

H(key1) = H(key2), where key1 $\neq$ key2

Several kinds of hash tables

The main difference is how collisions are handled

Closed addressing or Separate chaining:

Hash function H(key) = key % 7

Insert keys 50, 55, 39, 27, 100, 18, 76, 40, 71

H(50) = 1
H(55) = 6
H(39) = 4
H(27) = 6 collision
H(100) = 2
H(18) = 4 collision
H(76) = 6 collision
H(40) = 5
H(71) = 1 collision

index	Table	chains (linked lists)
0	null	
1		$\to$ 71 $\to$ 50
2		$\to$ 100
3	null	
4		$\to$ 18 $\to$ 39
5		$\to$ 40
6		$\to$ 76 $\to$ 27 $\to$ 55

Suppose n keys, N slots, H(key) = key % N, separate chaining

Worst-case: $\theta(n)$ time per operation (insert, find, remove)

Average-case: $\theta(n/N)$ time per operation, which is $\theta(1)$ if $n \le cN$

Open addressing:

No separate chains, all keys must be stored in the table
Only one key per slot
Requires $n \le N$
When collision occurs, look in another slot (Which slot?)
Several varieties of open addressing are popular

Linear probing:

Search sequentially

```
for (j=0; j<N; j++)
    look in slot (H(key) + j) % N
```

Hash function H(key) = key % 11
Insert keys 50, 55, 39, 27, 100, 18, 76, 40, 71

H(50) = 6
H(55) = 0
H(39) = 6 occupied
H(27) = 5
H(100) = 1
H(18) = 7 occupied
H(76) = 10
H(40) = 7 occupied
H(71) = 5 occupied

Table

0	55
1	100
2	71
3	
4	
5	27
6	50
7	39
8	18
9	40
10	76

Suppose n keys, N slots, H(key) = key % N, linear probing
Worst-case: $\theta(n)$ time per operation (insert, find, remove)
Average-case: $\theta(1)$ time if $n \leq N/2$

Disadvantages of linear probing:
- Clustering, especially when $n > N/2$
- Removing a key is complicated and inefficient
  - Example: how to remove key 39 ?
  - Usually mark as "deleted" rather than remove

Other kinds of open addressing aim to reduce clustering,
although they cannot eliminate it completely

Quadratic probing:

for (j=0; j<N; j++)
    look in slot (H(key) + j^2) % N

Hash function H(key) = key % 11

Insert keys 29, 14, 51, 36, 73, 58, 55

H(29) = 7
H(14) = 3
H(51) = 7 occupied
    offset 1
H(36) = 3 occupied
    offset 1
H(73) = 7 occupied
    offsets 1, 4
H(58) = 3 occupied
    offsets 1, 4, 9
H(55) = 0 occupied
    offsets 1, 4, 9

Table

0	73
1	58
2	
3	14
4	36
5	
6	
7	29
8	51
9	55
10	

Double hashing:

H is primary hash function, H' is secondary hash function

```
for (j=0; j<N; j++)
    look in slot (H(key) + j*H'(key)) % N
```

Primary hash function H(key) = key % 11
Secondary hash function H'(key) = (key % 7) + 1,
because H'(key) must never evaluate to 0

Insert keys 29, 14, 51, 36, 73, 58, 55

H(29) = 7
H(14) = 3
H(51) = 7 occupied
H'(51) = 3
H(36) = 3 occupied
H'(36) = 2
H(73) = 7 occupied
H'(73) = 4
H(58) = 3 occupied
H'(58) = 3
H(55) = 0 occupied
H'(55) = 7
offsets 7, 14,
21, 28, 35

Table

0	73
1	
2	55
3	14
4	
5	36
6	58
7	29
8	
9	
10	51