Hash Tables

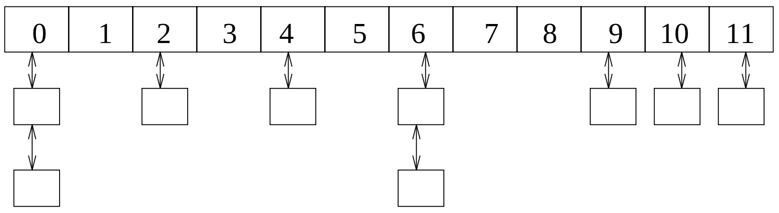
Hash tables are a *very practical* way to maintain a dictionary. As with bucket sort, it assumes we know that the distribution of keys is fairly well-behaved.

The idea is simply that looking an item up in an array is $\Theta(1)$ once you have its index. A hash function is a mathematical function which maps keys to integers.

In bucket sort, our hash function mapped the key to a bucket based on the first letters of the key. "Collisions" were the set of keys mapped to the same bucket.

If the keys were uniformly distributed, then each bucket contains very few keys!

The resulting short lists were easily sorted, and could just as easily be searched!



Hash Functions

It is the job of the hash function to map keys to integers. A good hash function:

1. Is cheap to evaluate

2. Tends to use all positions from $0 \ldots M$ with uniform frequency.

3. Tends to put similar keys in different parts of the tables (Remember the Shifletts!!)

The first step is usually to map the key to a big integer, for example

$$h = \sum_{i=0}^{keylength} 128^i \times char(key[i])$$

This large number must be reduced to an integer whose size is between 1 and the size of our hash table.

One way is by $h(k) = k \bmod M$, where M is best a large prime not too close to $2^i - 1$, which would just mask off the high bits.

This works on the same principle as a roulette wheel!

Good and Bad Hash functions

The first three digits of the Social Security Number

0	1	2	3	4	5	6	7	8	9

The last three digits of the Social Security Number

0	1	2	3	4	5	6	7	8	9

The Birthday Paradox

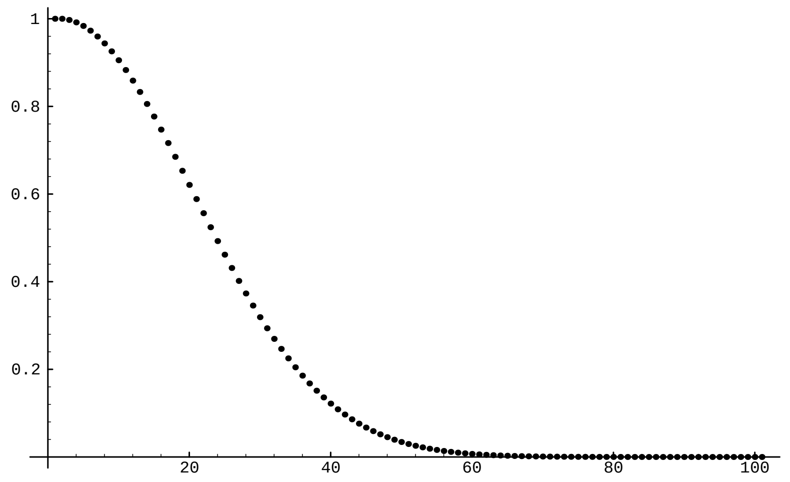
No matter how good our hash function is, we had better be prepared for collisions, because of the birthday paradox.

J	F	M	A	M	J	J1	A	S	O	N	D

The probability of there being *no* collisions after n insertion into an m-element table is

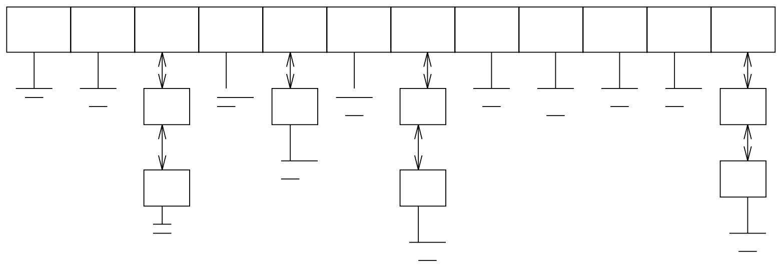
$$(m/m) \times ((m-1)/m) \times \ldots \times ((m-n+1)/m) = \Pi_{i=0}^{n-1} (m-i)/m$$

When $m = 366$, this probability sinks below 1/2 when $N = 23$ and to almost 0 when $N \geq 50$.



Collision Resolution by Chaining

The easiest approach is to let each element in the hash table be a pointer to a list of keys.

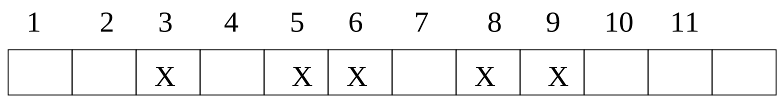


Insertion, deletion, and query reduce to the problem in linked lists. If the n keys are distributed uniformly in a table of size m/n, each operation takes $O(m/n)$ time.

Chaining is easy, but devotes a considerable amount of memory to pointers, which could be used to make the table larger. Still, it is my preferred method.

Open Addressing

We can dispense with all these pointers by using an implicit reference derived from a simple function:



If the space we want to use is filled, we can examine the remaining locations:

1. Sequentially $h, h+1, h+2, \ldots$

2. Quadratically $h, h+1^2, h+2^2, h+3^2 \ldots$

3. Linearly $h, h+k, h+2k, h+3k, \ldots$

The reason for using a more complicated science is to avoid long runs from similarly hashed keys.

Deletion in an open addressing scheme is ugly, since removing one element can break a chain of insertions, making some elements inaccessible.

Performance on Set Operations

With either chaining or open addressing:

- Search - $O(1)$ expected, $O(n)$ worst case
- Insert - $O(1)$ expected, $O(n)$ worst case
- Delete - $O(1)$ expected, $O(n)$ worst case
- Min, Max and Predecessor, Successor $\Theta(n+m)$ expected and worst case

Pragmatically, a hash table is often the best data structure to maintain a dictionary. However, we will not use it much in proving the efficiency of our algorithms, since the worst-case time is unpredictable.

The best worst-case bounds come from balanced binary trees, such as red-black trees.