Hash Tables

Hash tables are generally made up of three components:

1. A **hash function**, $f(k)$ or $h(k)$. The hash function transforms a **key** into an integer (its index in array)
2. An **array** (the hash *table*)
3. The set of **values** associated with each **key.**

Assume n is the number of elements in the hash table, and m is size allocated to the underlying array

Goals of hash function:

- when given two keys with the same value, it should produce the same integer
- uniform distribution of elements (evenly spread across hash table)

Challenges:

Very difficult to achieve the goals without information about the keys ahead of time.

Load factor: $\alpha = \frac{\#\ hashed\ items}{table\ size} = \frac{n}{m}$ $\alpha = 1$ means full; $\alpha = 0.5$ means half full

---

Two strategies to handle collisions:

Separate chaining: store multiple elementsin each slot (often with a linked list)

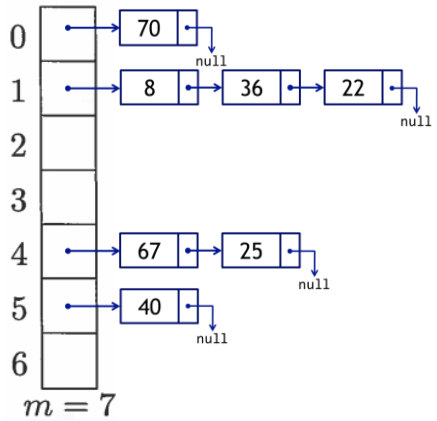
Open Addressing: one item per slot, specify a sequence of slots to try next

Separate Chaining example:

Given a hash function $h(k) = k \bmod m$

Insert the following keys into the hash table:

$8, 40, 36, 67, 22, 25, 70$



Separate Chaining:

Pros: Can have unlimited entries ($n > m$) in the hash table

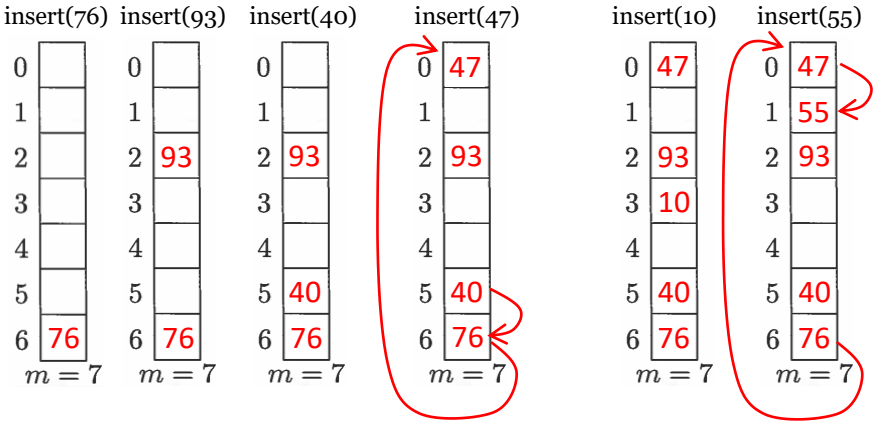
Cons: Memory (a linked list node) is allocated for each insertion (this is not efficient from a memory perspective)

Performance depends on the chain length at each index (what if only one index is used and it's just a large linked list)

Open Addressing

Linear Probing: $h(k, i) = (hash(k) + i) \bmod m$

Open Addressing Exercise:



Searching for a key

Algorithm Find(k):
Input: the target key to search the hash table for
Output: the entry with the given key; null if key not found

$i \leftarrow h(k) \% m$
for $i \leftarrow 1$ to m **do**
  $e \leftarrow$ data[i]
  if e is *null* **then**
    return *null*
  end if
  if e's key is equal to k **then**
    return e
  end if
  $i \leftarrow (i + 1) \% m$
end for
return *null*
end

Search example:

Find entries with the following keys:

93, 8, 47, 12

93 – found immediately
8 – look at indexes 1, 2, 3, return null
47 – look at indexes 5, 6, 0, return entry
12 – look at indexes 5, 6, 0, 1, 2, 3 and finally return null

Linear Probing:

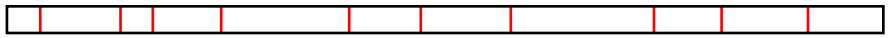
Pros: if load factor < 1, there is always an open location, and no additional memory needs to be allocated

Cons: performance quickly degrades once the list is at least half full
primary clustering negatively affects performance

Primary Clustering

Description: Long, consecutive sequence of filled spots

Remember: the goal is to have entries evenly spaced out across the hash table:



When primary clustering occurs, many entries are together, making it take much longer to find an empty slot for insertion (or searching)

Proposed solution:

Make the jumps to search for next unfilled location jump more than 1 position each time.

This will allow us to get out of the clustered section faster