15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 24

Hash Tables

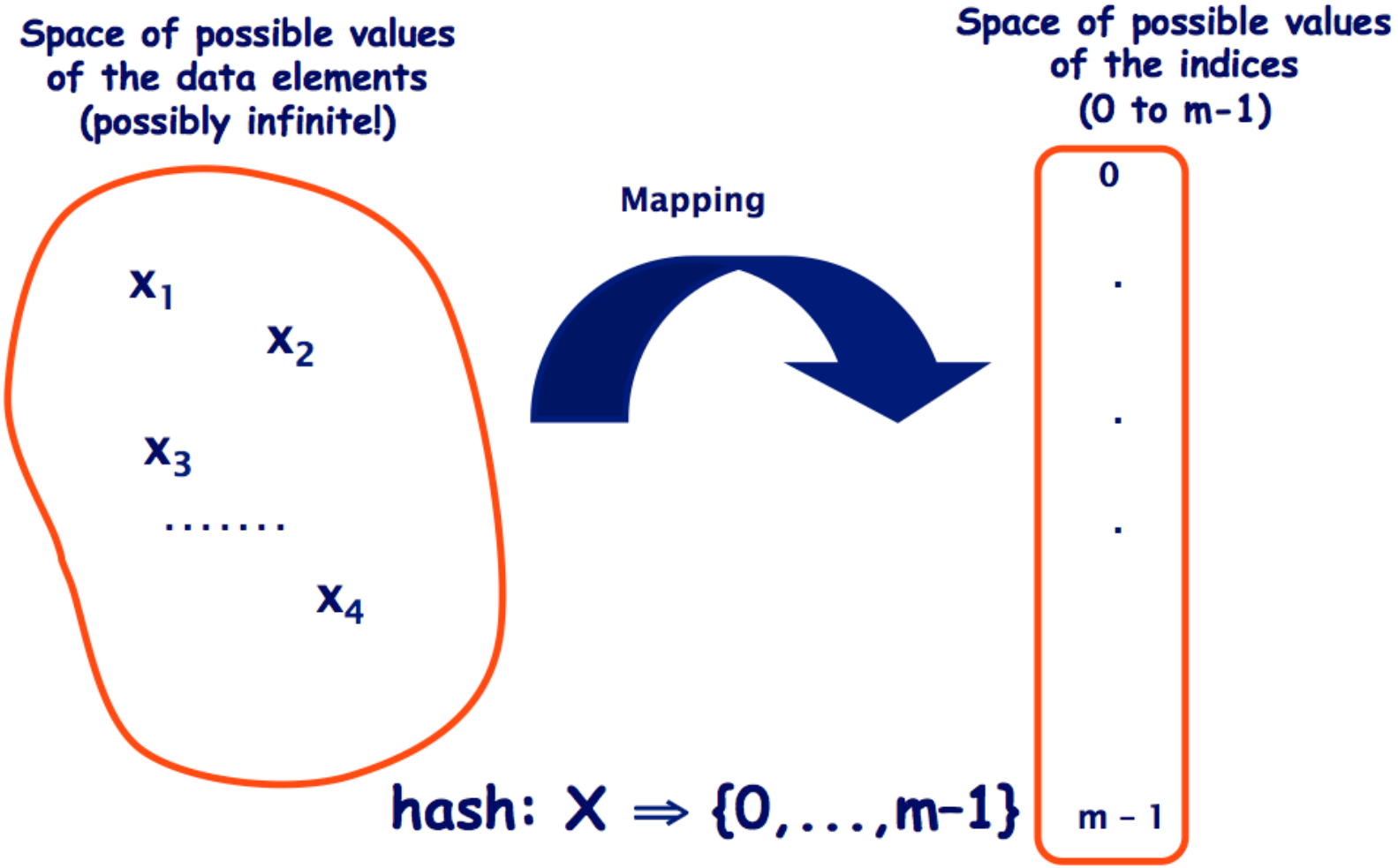
Synopsis

- Hashing and Hash Tables
- Handling Collisions
  - Linear Probing
  - Quadratic Probing

Hash Tables – Basic Ideas

- Data structure that allows you to quickly insert, delete, and retrieve items with expected $O(1)$ work.
- Relies on
  - a fixed size array data structure (of some size m), and
  - a hash function that can map from a potentially infinite space of keys to integer indexes $[0, \ldots, m-1]$
- Disadvantages
  - Collisions
  - Increased memory use to avoid collisions
  - Not work efficient for *findmin*, *findmax*, or *extracting keys in sorted order*

Hash Table - Basic Ideas



Hash Functions

- There is a deep theory behind hash functions.
- We will be interested in some simple functions.
- We will assume hash functions have the idealized property of *simple uniform hashing*:
  - The hash function uniformly distributes keys in range $[0, \ldots, m-1]$
  - Hash value for one key is independent of the hash value for another key.

Hash Functions

- For integers key we can use a linear congruential hash function

$$h(x) = (ax + b) \bmod m$$

where $a \in [1, \ldots, m-1]$, $b \in [0, \ldots, m-1]$, and m is prime.

Hash Functions

- For strings, we can use a polynomial like

$$h(S) = \left(\sum_{i=1}^{|S|} s_i a^i \right) \bmod m$$

Hash Tables

- Support *insert*, *find* and *delete*.
- Can implement abstract data types *Set* and *Table*.
- Do not require total ordering on the universe of keys.
- *Collision* is the main issue
  - Two keys hash to the same location.
  - Impossible to avoid if we do not know the keys in advance
    - Size of key universe $>>$ size of table.

Collisions

- For a table size of 365, one needs 23 keys for a 50% chance of collision and 66 for a 99% chance of collision (Why?)
  - Birthday paradox

Handling Collisions

- **Separate chaining**
  - Store elements not in a table, but in linked lists (containers,bins) hanging off the table.
- **Open addressing**:
  - Put everything into the table, but not necessarily into cell $h(k)$.
- **The perfect hash**:
  - When you know the keys in advance, construct hash functions that avoids collisions entirely.
- **Multiple-choice hashing/Cuckoo hashing**:
  - Consider exactly two locations $h_1(k)$ and $h_2(k)$ only.

Handling Collisions

- We will only consider the first two.
- We will assume we have a set n keys K and a hash function $h : key \rightarrow [0, \ldots, m-1]$ for some m.

Separate Chaining

- Maintain an array of linked lists (buckets).
- Keys that hash to the same value live in the same list at location $h(k)$
- **Insertion**: Insert at the beginning
  - Multiple inserts for the same key $\Rightarrow$ traverse the list
  - May as well insert at the end.
- **Find**: hash to $h(k)$ and search in the list.
- **Delete**: remove from the list.

Separate Chaining

- Costs depend on the *load factor* $\lambda = n/m$ which is also the average length of a list.

Separate Chaining

- Assume $h(k)$ takes $O(1)$ work and we have simple uniform hashing
- *Unsuccessful search* takes expected $\Theta(1 + \lambda)$ work.
  - $O(1)$ for $h(k)$ and λ for traversing the list.

Separate Chaining

- *Successful search* takes expected $\Theta(1 + \lambda)$ work.
- Cost of *Successful search* = Cost of *unsuccessful search* at the time of insertion (Why?)
- With i keys, the unsuccesssful search would take $(1 + i/m)$ work.
- Averaging over i we get

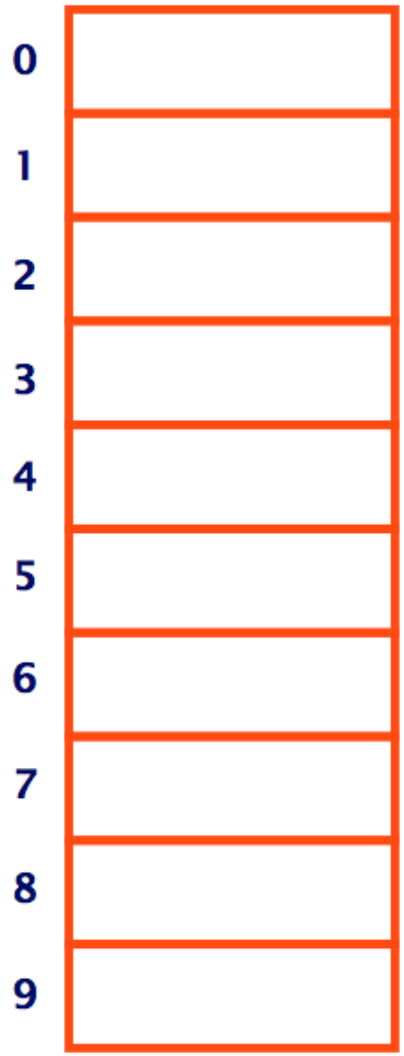
$$\frac{1}{n}\sum_{i=0}^{n-1}(1+i/m) = 1+(n-1)/2m = 1+\lambda/2-\lambda/2m = \Theta(1+\lambda)$$

- Considering constant factors, successful search looks at 1/2 the list on the average.

Open Addressing

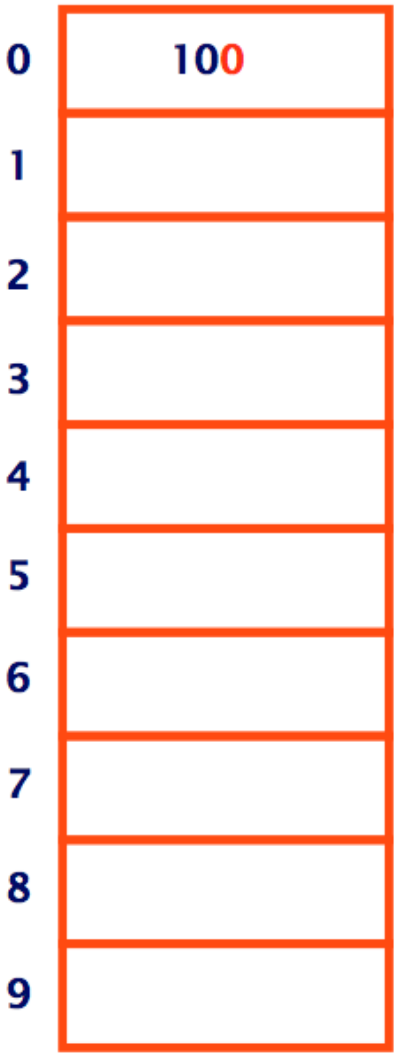
- No lists – everything is stored in the array directly
- The arrays is some constant factor larger than the maximum number of keys we want to store.

An Example



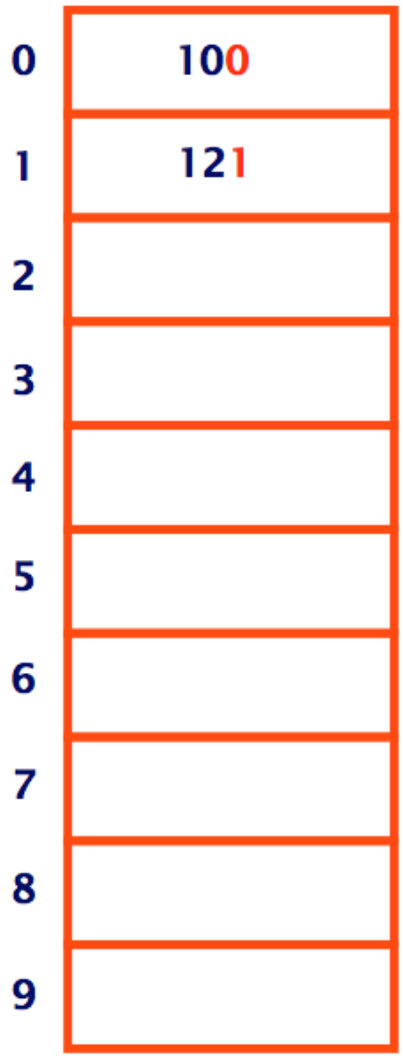
Initially the hash table is empty

An Example



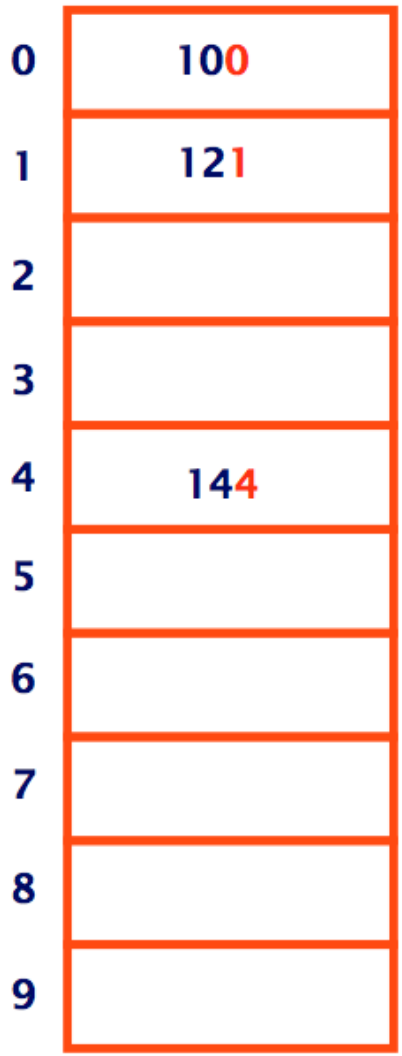
Insert 100

An Example



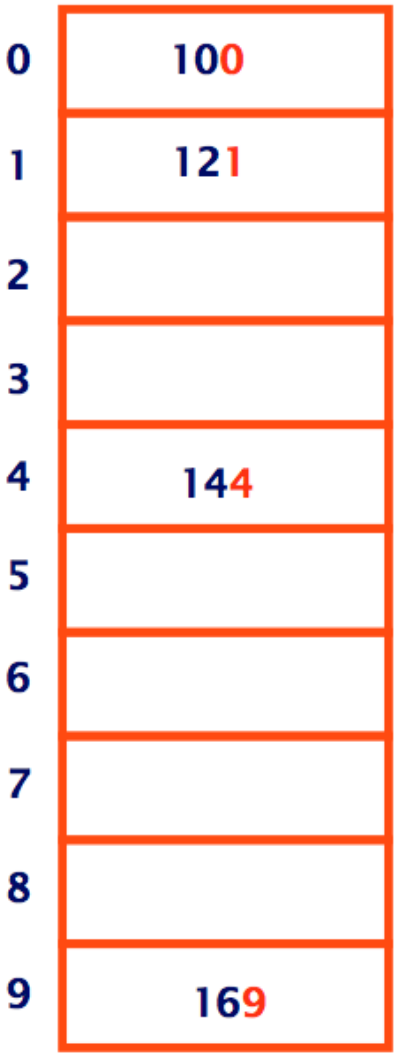
Insert 121

An Example



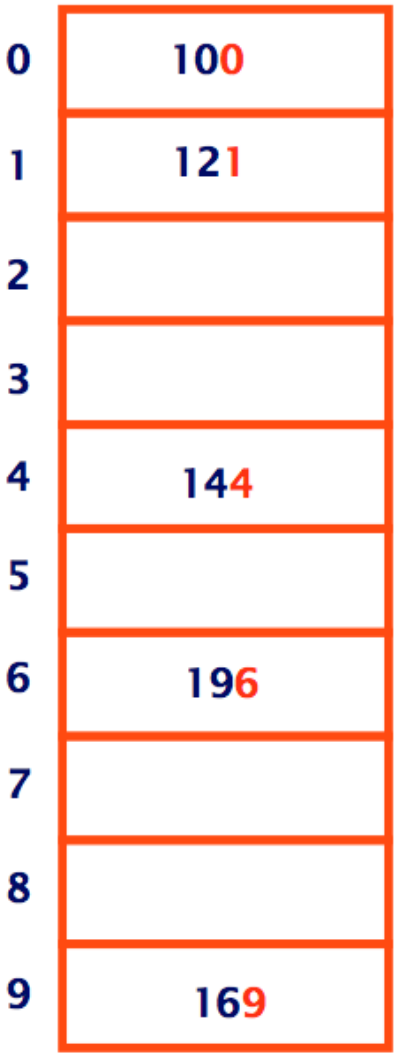
Insert 144

An Example



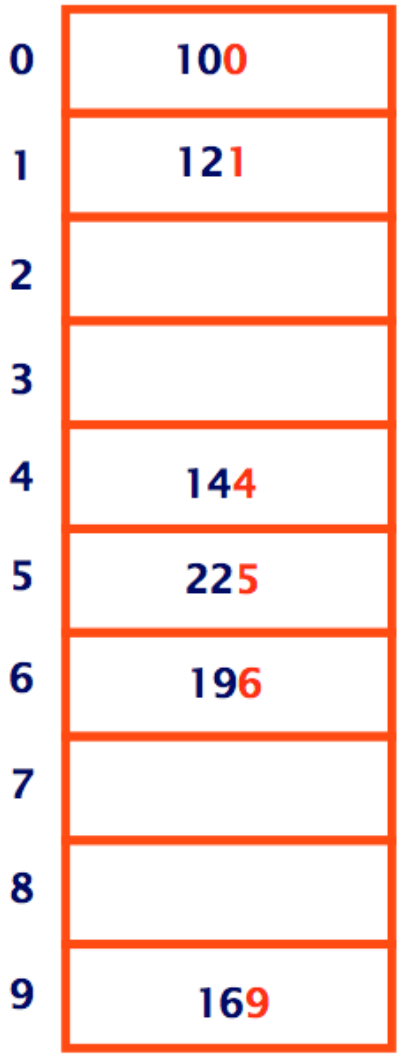
Insert 169

An Example



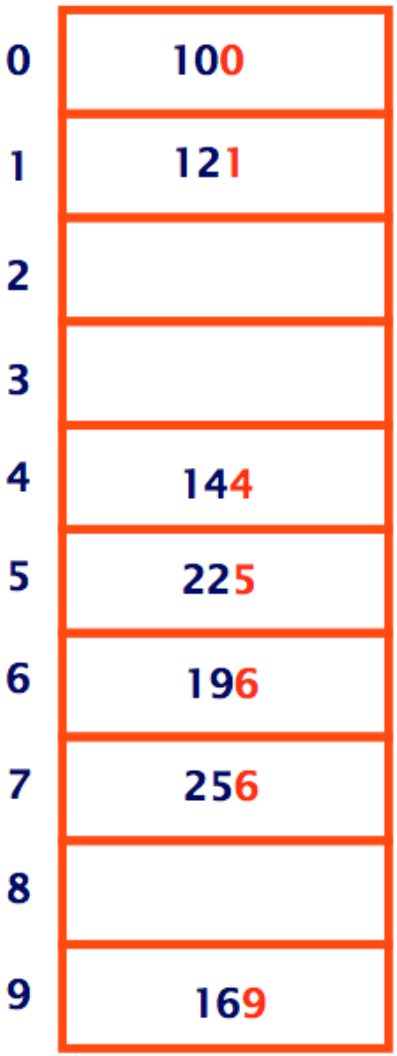
Insert 196

An Example



Insert 225

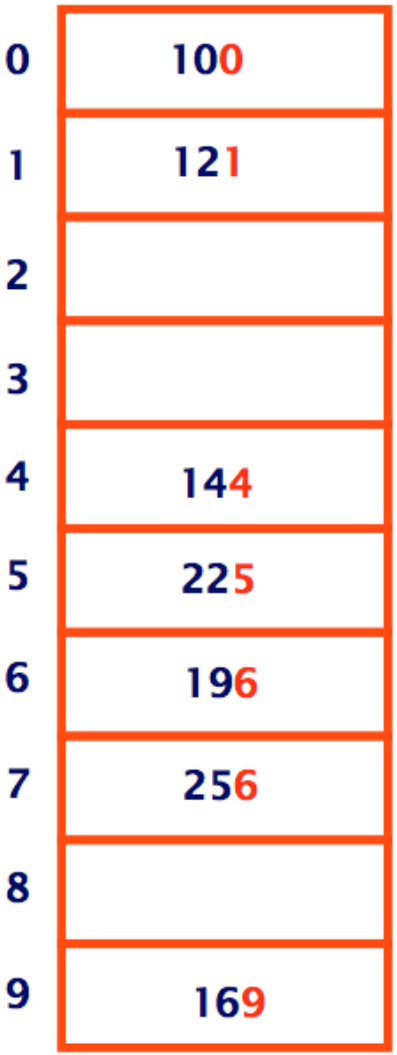
An Example



Insert 256 COLLISION because location

6 is full. Try location 6+1=7

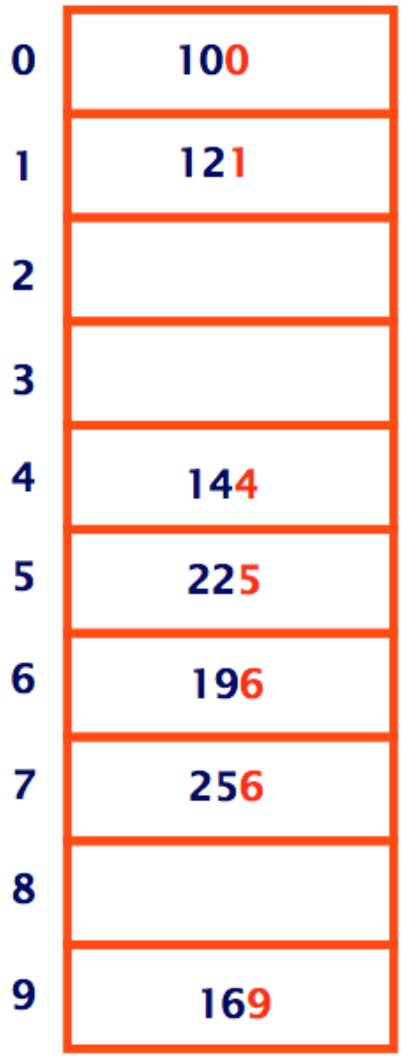
An Example



Insert 289 COLLISION because location

9 is full. Try location (9+1)mod 10=0

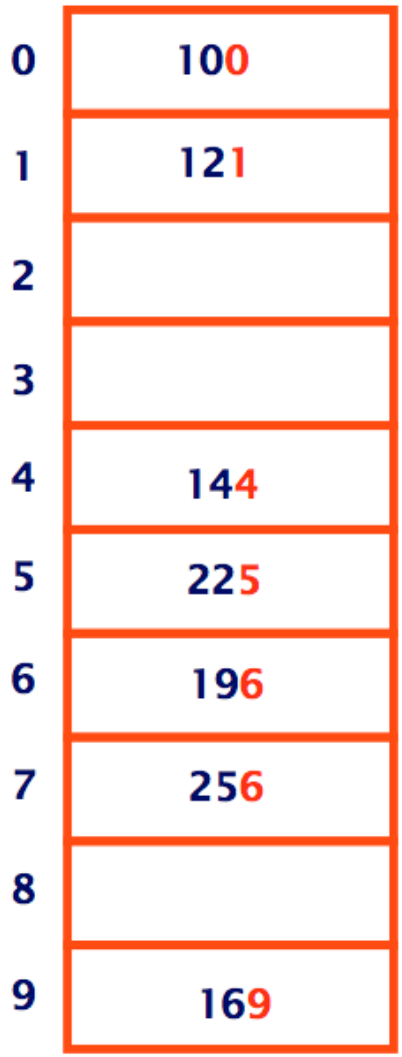
An Example



Insert 289 COLLISION because location 9 is full.

Try location (9+1)mod 10= 0 FULL

An Example

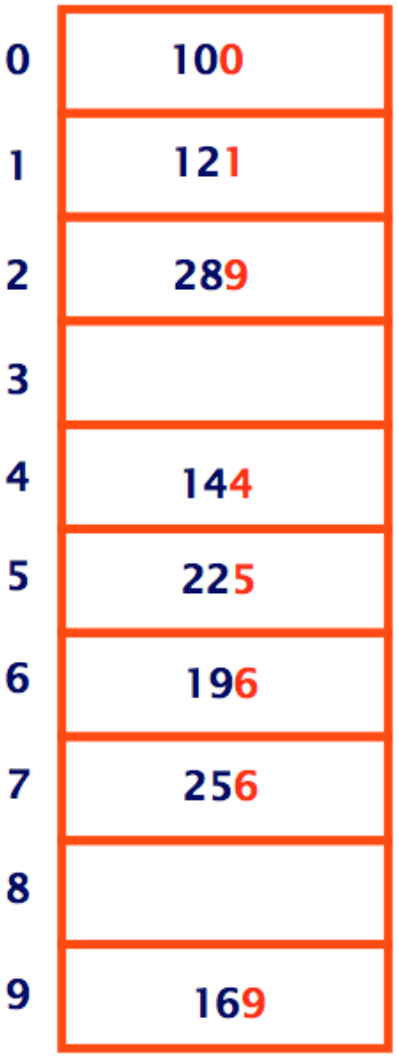


Insert 289 COLLISION because location 9 is full.

Try location (9+1)mod 10= 0 FULL

Try location (9+2)mod 10= 1 FULL

An Example



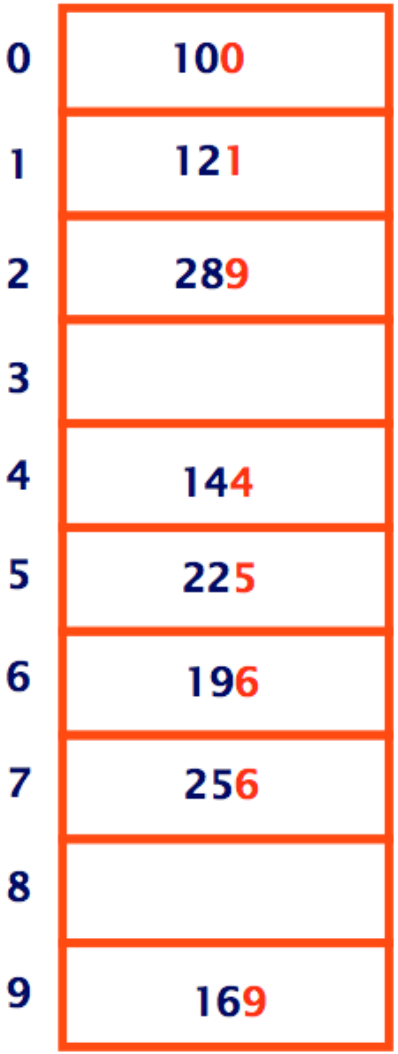
Insert 289 COLLISION because location 9 is full.

Try location (9+1)mod 10= 0 FULL

Try location (9+2)mod 10= 1 FULL

Try location (9+3)mod 10= 2 AVAILABLE

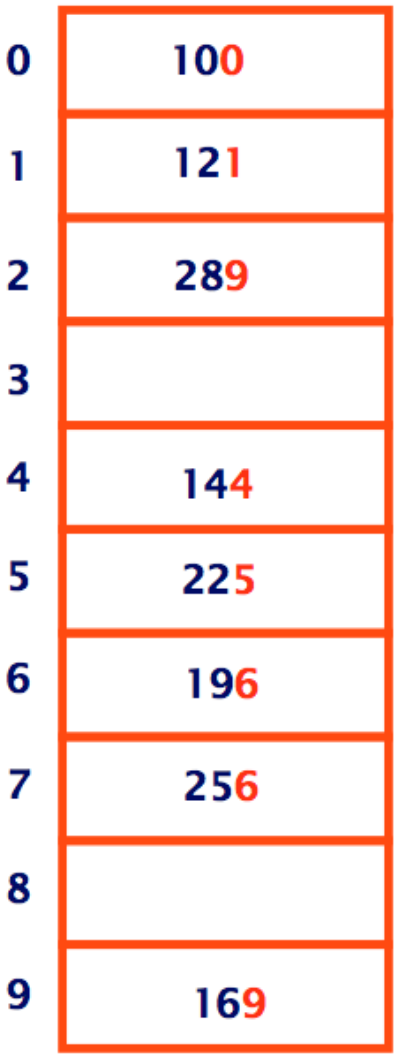
An Example



Insert 324 COLLISION because location 4 is full.

Try location (4+1)mod 10= 5 FULL

An Example

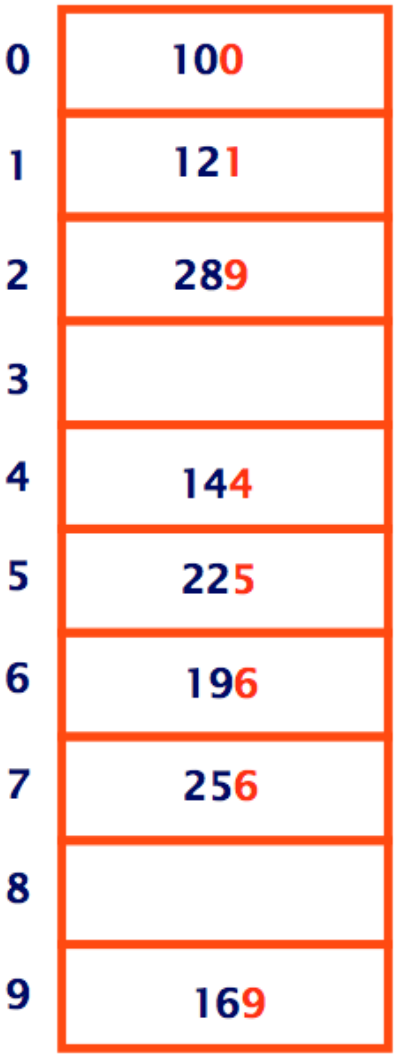


Insert 324 COLLISION because location 4 is full.

Try location (4+1)mod 10= 5 FULL

Try location (4+2)mod 10= 6 FULL

An Example



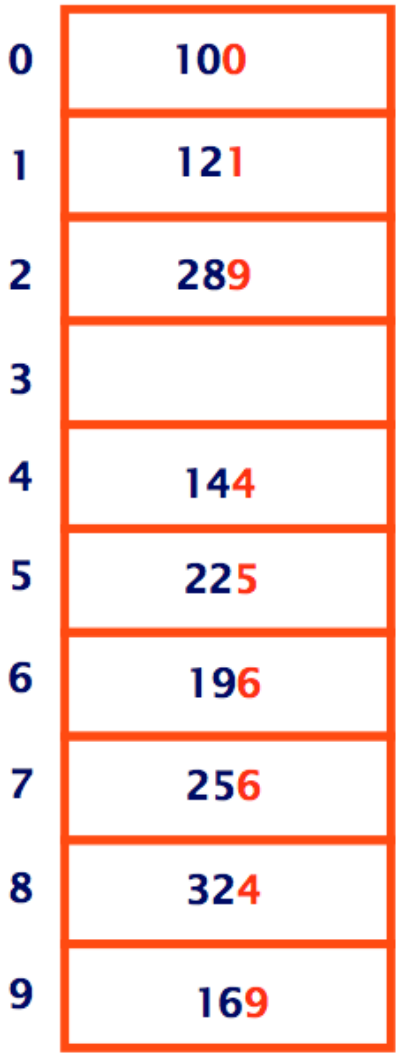
Insert 324 COLLISION because location 4 is full.

Try location (4+1)mod 10= 5 FULL

Try location (4+2)mod 10= 6 FULL

Try location (4+3)mod 10= 7 FULL

An Example



Insert 324 COLLISION because location 4 is full.

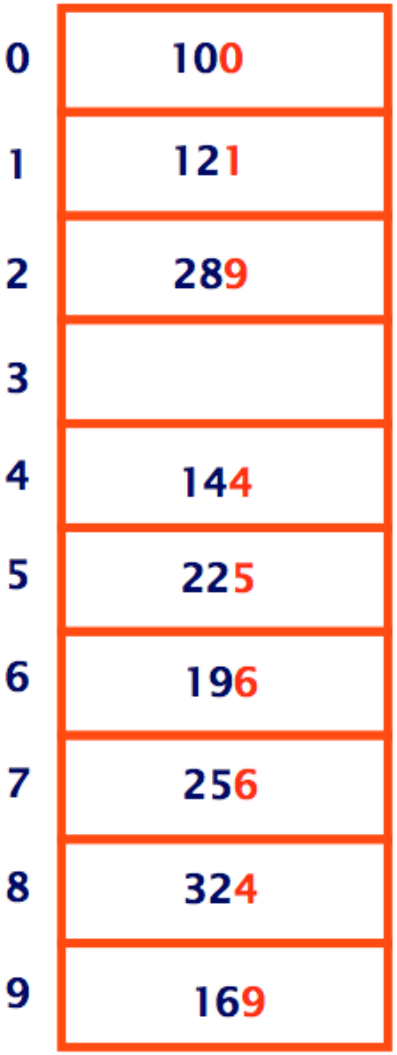
Try location (4+1)mod 10= 5 FULL

Try location (4+2)mod 10= 6 FULL

Try location (4+3)mod 10= 7 FULL

Try location (4+4)mod 10= 8 AVAILABLE

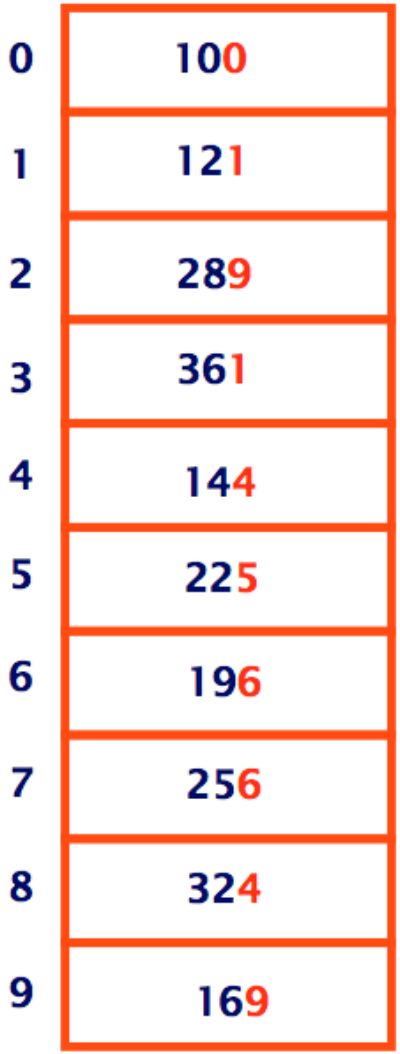
An Example



Insert 361 COLLISION because location 1 is full.

Try location (1+1)mod 10= 2 FULL

An Example



Insert 361 COLLISION because location

1 is full.

Try location (1+1)mod 10= 2 FULL

Try location (1+2)mod 10= 3
AVAILABLE

Open Addressing

- Open addressing uses an ordered sequence of locations.
- $h(k, i)$ gives us the i^{th} location for key k.
- $\langle\, h(k, 0), h(k, 1), h(k, 2), \ldots \rangle$ is the *probe sequence*.
- Try these locations in order until an empty cell is found and insert there.

Open Addressing - Insert

```
1  fun insert(T, k) =
2  let
3     fun insert'(T, k, i) =
4        case nth T h(k, i) of
5           NONE ⇒ update(h(k, i), k) T
6         | _ ⇒ insert'(T, k, i + 1)
7  in
8     insert'(T, k, 1)
9  end
```

- T must be an ST array - otherwise work and span are not constant.
- Need to check if table is full and the key is already in the table or not.

Open Addressing-Search

```
1   fun find(T, k) =
2   let
3      fun find'(T, k, i) =
4         case nth T h(k, i) of
5            NONE ⇒ false
6          | SOME(k') ⇒ if (eq(k, k')) then true
7                       else find'(T, k, i + 1)
8   in
9      find'(T, k, 1)
10  end
```

Open Addressing-Delete

- We can not just delete an items and set its cell to *NONE*! (Why ?)
- *find* will stop searching if it encounters an empty cell.
- Use *lazy delete*
  - Instead of deleting, use a special value HOLD.

```
1  datatype α entry = EMPTY | HOLD | FULL of α
```

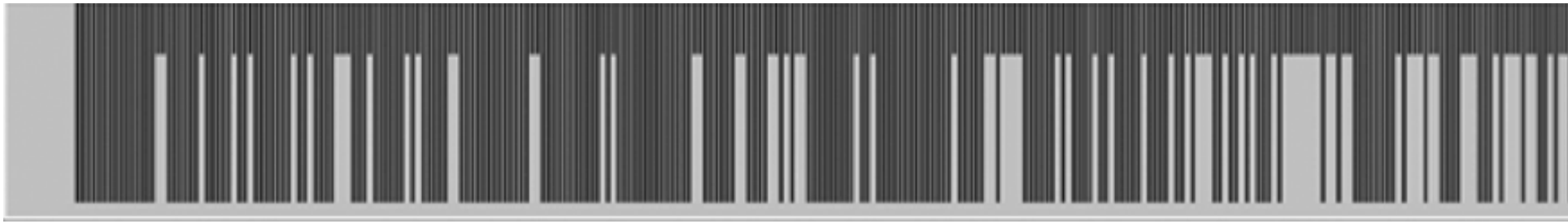
- Find and Insert will need to be changed accordingly.
- Lazy delete effectively increases load factor.
- Rehashing to the rescue!

Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing

Linear Probing

- We check cell at $h(k, i) = (h(k) + i) \mod m$ in i^{th} probe.
- m possible probe sequences.
- Keys tend to cluster – *primary clustering*.
  - Inserts add to a cluster
  - Probe sequences get longer and longer



Impact of Clustering

- Assume table is half full ($\lambda = 1/2$)
- Minimum clustering when every other cell is empty!
- Average probes for insert is 3/2
  - One probe to check cell $h(k)$
  - $+$ with 1/2 chance try the next cell (which by design should be empty)

Impact of Clustering

- Worst case: all keys are clustered to the second half of the array. (Remember $\lambda = 1/2 \Rightarrow m = 2n$)
- How many probes for positions 0 through $n-1$?
  - 1 (Why?)
- How many probes when initial hash is to cell n?
  - n (Why?)
- How many probes when initial hash is to cell $n+1$?
  - $n-1$ (Why?)
- Average is

$$(n+[n+(n-1)+(n-2)+....+1])/m = n/m+n(n+1)/2m \approx n/4$$

- Even though though the average cluster length is 2, the cost is about $n/4$ probes.

Costs for Linear Probing

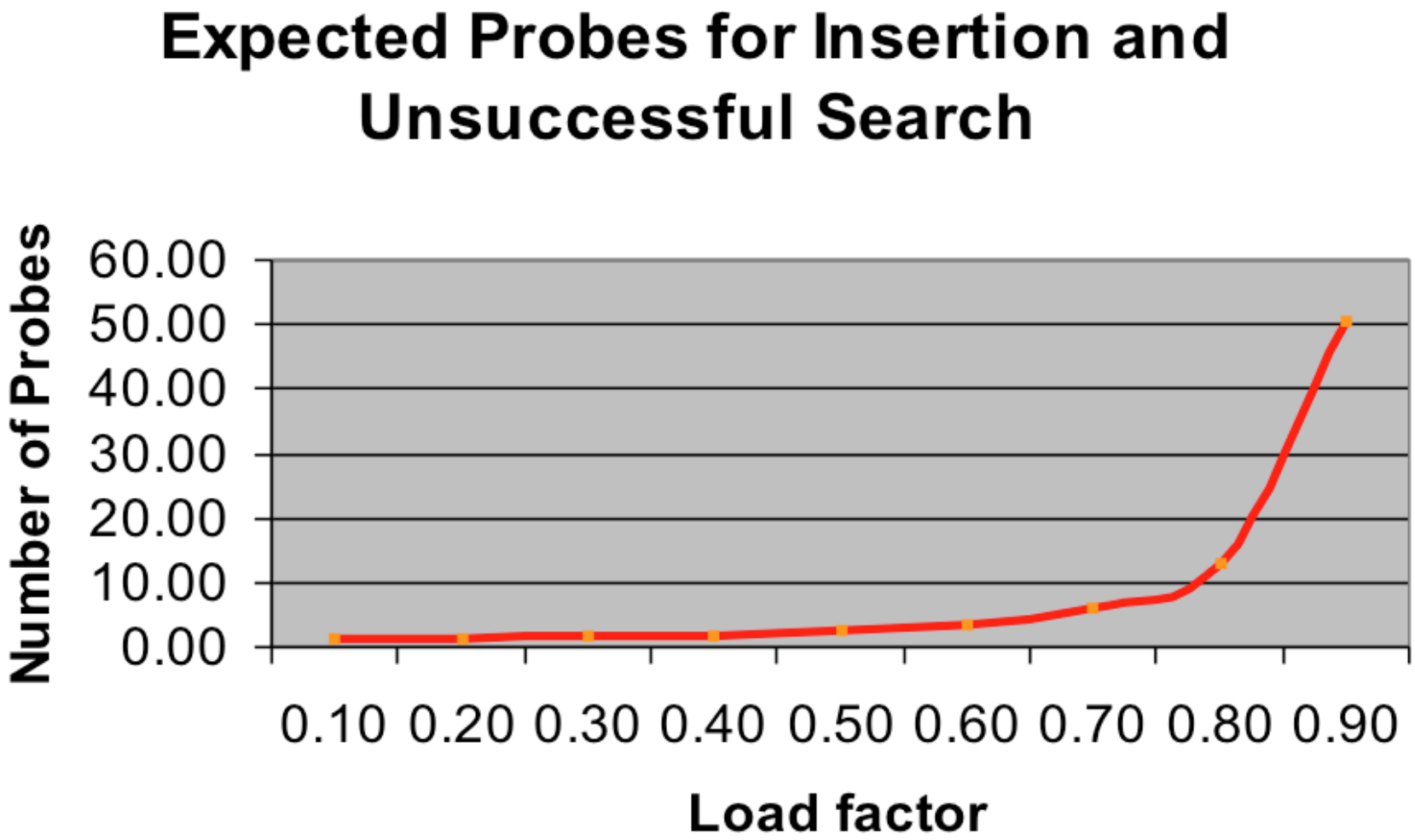
- Given a hash table of size m and with $n = \lambda m$ keys.
- The cost of an unsuccessful search/insert is

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda^2}\right)$$

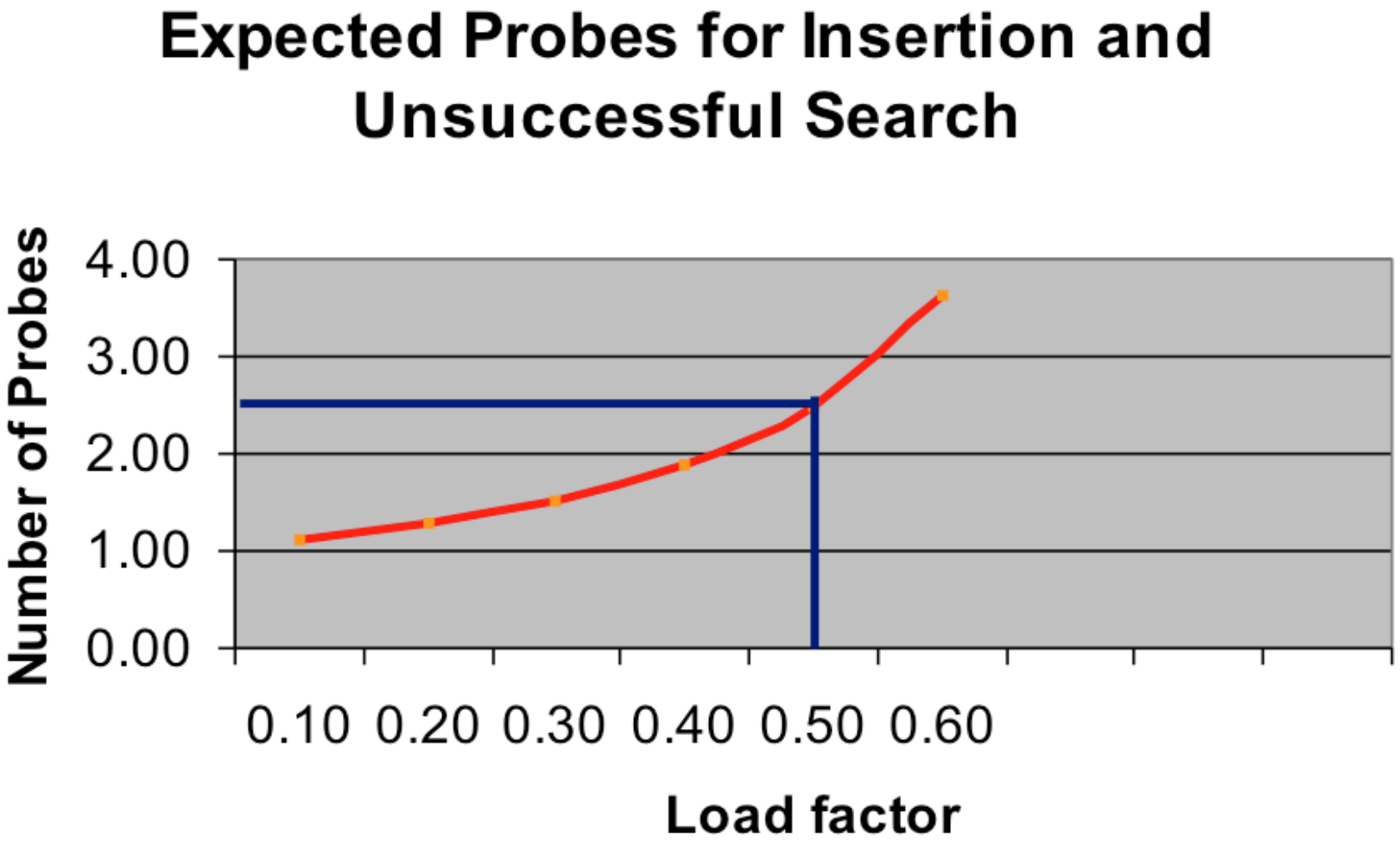
- The cost of an successful search is

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right).$$

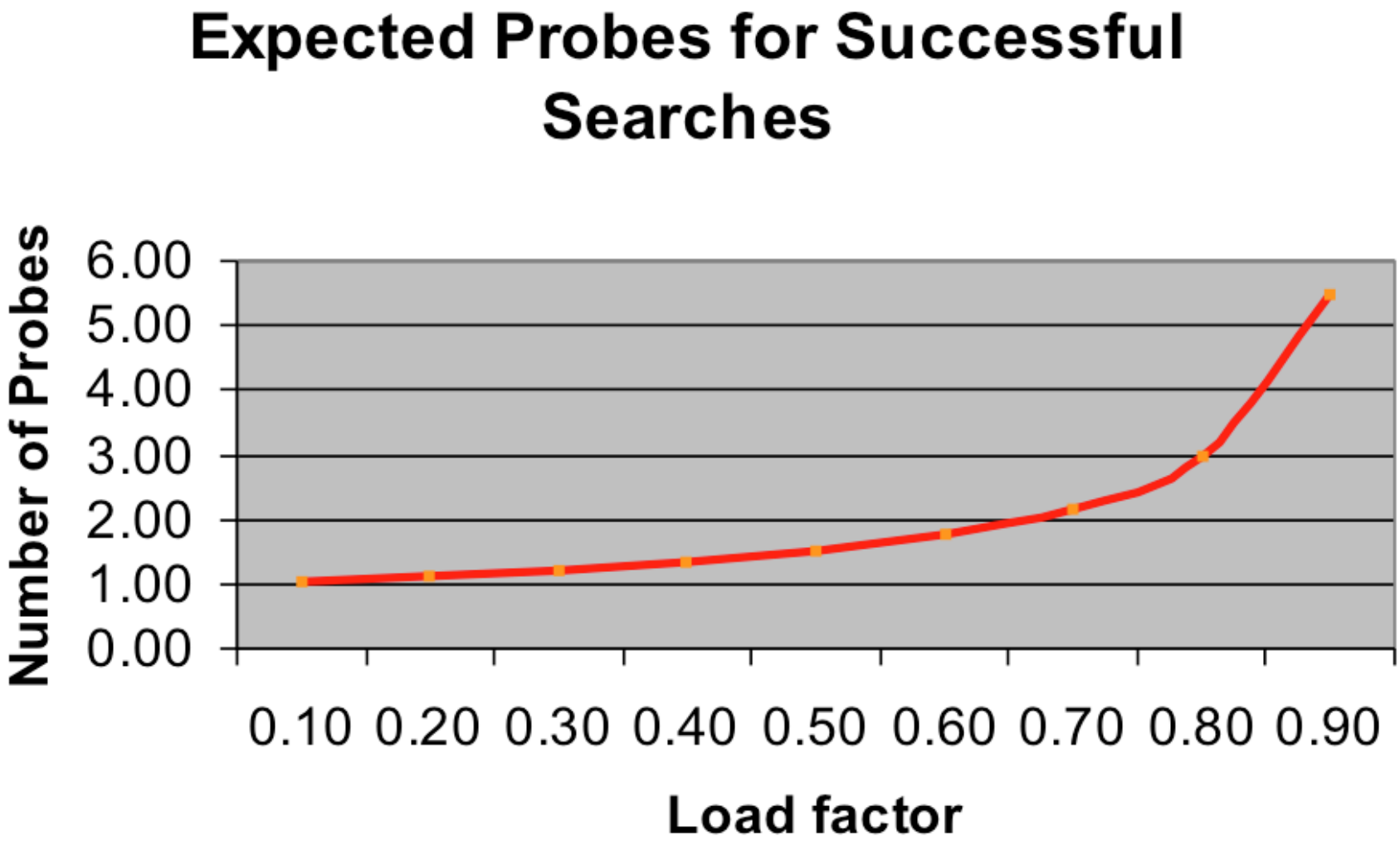
Costs for Linear Probing



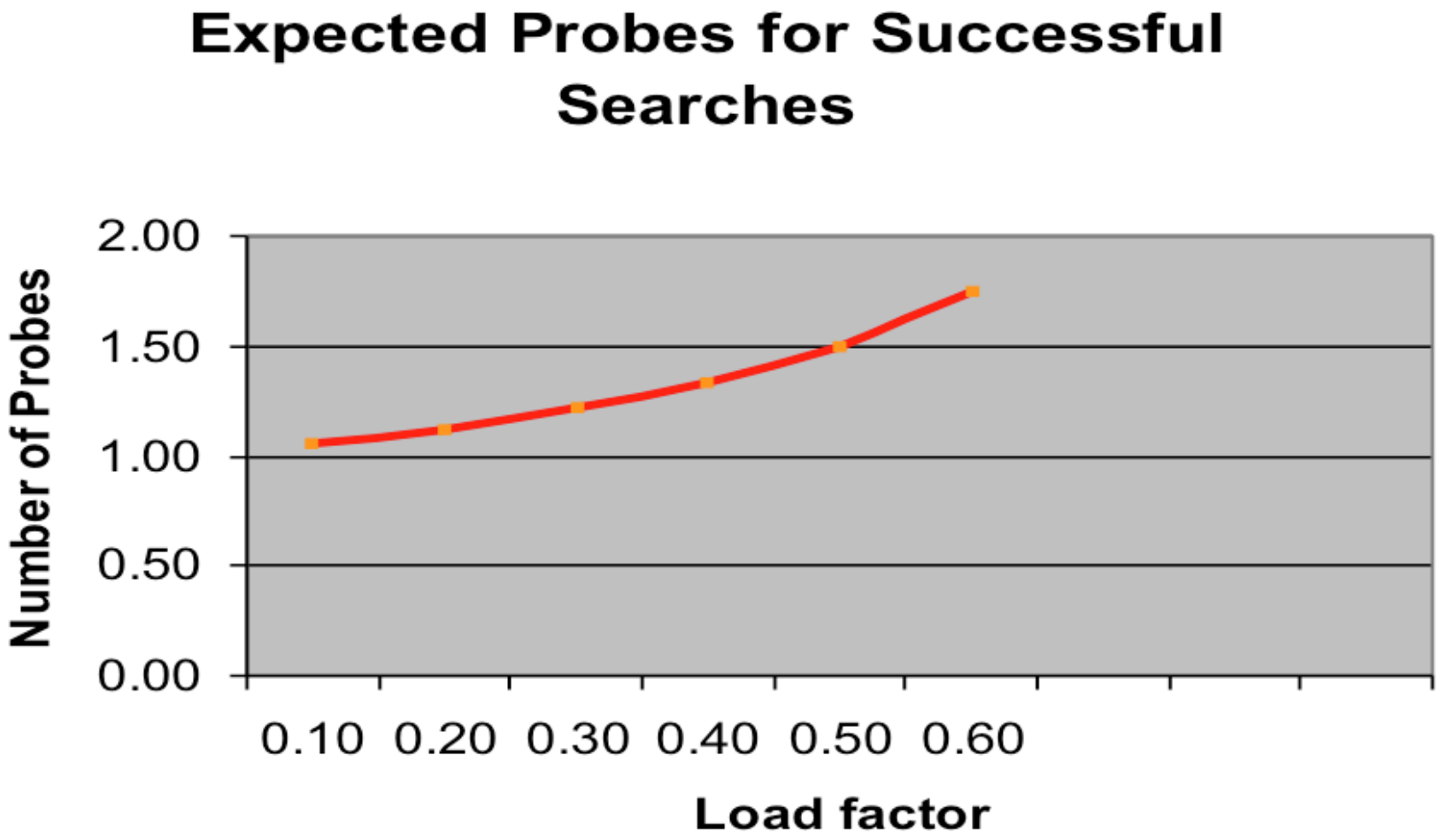
Costs for Linear Probing



Costs for Linear Probing



Costs for Linear Probing



Quadratic Probing

- We check cell at $h(k, i) = (h(k) + i^2) \mod m$ in i^{th} probe.
- Makes longer jumps
- Avoids primary clustering
- But has *secondary clustering*.
- Since there are m possible positions there are m probe sequences.
- Not all available cells get probed (Why?)

Quadratic Probing

- If m is prime and the table is at least half empty, then quadratic probing will always find an empty location.
- Furthermore, no locations are checked twice.

Quadratic Probing

- Consider two probe locations $h(k) + i^2$ and $h(k) + j^2, 0 \leq i, j < \lceil m/2 \rceil$.
- Suppose the locations are the same but $i \neq j$.

$$\begin{aligned} h(k) + i^2 &\equiv (h(k) + j^2) \mod m \\ i^2 &\equiv j^2 \mod m \\ i^2 - j^2 &\equiv 0 \mod m \\ (i - j)(i + j) &\equiv 0 \mod m \end{aligned}$$

- Therefore, either $i - j$ or $i + j$ are divisible by m.
- But since both $i - j$ and $i + j$ are less than m and m is prime, they cannot be divisible by m.
- Thus the first $\lceil m/2 \rceil$ probes are distinct and guaranteed to find an empty location.

Quadratic Probing

- Computing the next hash value is only slightly more expensive

$$h_i - h_{i-1} \equiv (i^2 - (i-1)^2) \mod m$$
$$h_i \equiv (h_{i-1} + 2i - 1) \mod m$$

- If the table gets too full, one can resize and rehash
  - Constant additional overhead

Double Hashing

- Uses two hash-functions:
  - initial location
  - size of the jump
- i^{th} probe is

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m.$$

- Different keys are likely to have different values jump function if they collide.
- Avoids secondary clustering
- $h_2(k)$ should be relatively prime to m to probe each locations.
  - m prime and $0 < h_2(k) < m$ is one option.

Double Hashing

- The average number of probes for an unsuccessful search or an insert is at most

$$1 + \lambda + \lambda^2 + \ldots = \left(\frac{1}{1-\lambda}\right)$$

  - Why?
- The average number of probes for a successful search is

$$\frac{1}{\lambda}\left(1 + \ln\left(\frac{1}{1-\lambda}\right)\right).$$

  - Same argument of averaging over probes at insertion time.

Double Hashing

λ	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.4	1.6	1.8	2.6
unsuccessful	1.3	1.5	2.0	3.0	5.5

- Allows for smaller tables than linear or quadratic probing
- Higher cost for hash function

Parallel Hashing

- $injectCond(IV, S) : (int \times \alpha)seq \times (\alpha option)seq \rightarrow (\alpha option)seq$.
- Conditionally writes each value v_j into location i_j of S
  - if the location is set to NONE

```
1  fun insert(T, K) =
2  let
3     fun insert'(T, K, i) =
4        if |K| = 0 then T
5        else let
6             T' = injectCond({(h(k, i), k) : k ∈ K}, T)
7             K' = {k : k ∈ K | T[h(k, i)] ≠ k}
8        in
9           insert'(T', K', i + 1)        end
10 in
11    insert'(T, k, 1)
12 end
```