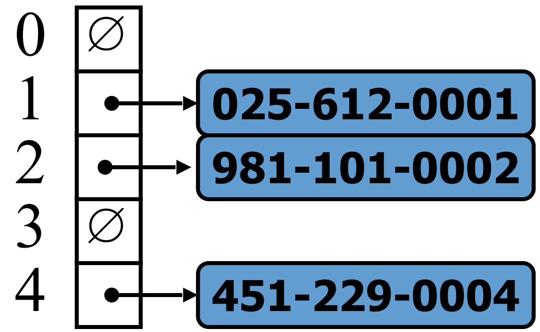
Hash Tables



Dictionary ADT

- Supports search, insert, and delete from a set S.
- Set is a collection of distinguishable objects called members or elements.
- S = {1, 2, 3, 4}
    - 4 is in S, 5 is not
    - Operations: intersection, union, difference
    - Laws: Empty set, idempotent, commutative, associative, distributive, absorption, De Morgan's
- Examples, BSTs and Hash Tables
    - BSTs are sorted sets, Hash Tables are not

Fast Operations

- BST performs searches, inserts, and removes in a remarkable amount of time
    - Log 1,000,000 ≈ 20
    - Log 1,000,000,000 ≈ 30
- Sometimes not fast enough
    - 911 call - search by telephone number
    - Air traffic control - search by flight number
    - Webpage retrievals

Hash Function

- Address calculator performs hashing using a hash function
    - Hash function is user defined
    - Hash table is typically an array
    - Each element maps into an array position
    - Operates in constant or near constant time
    - Perfect hash function maps each item to a unique table location – referred to as *“perfect hashing*

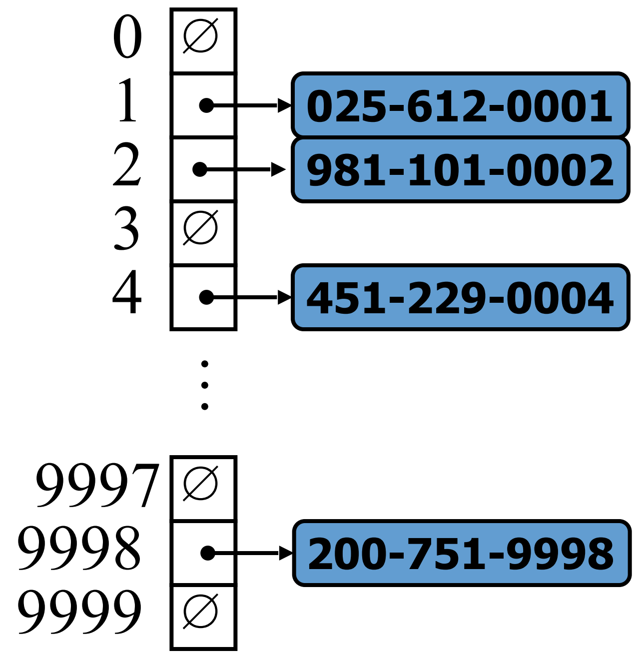
Hash Table

- Components:
  - Hash function h that maps keys of a given type to integers in a fixed interval [0, m-1]
    - Example:

      $h(k) = k \bmod m$
    - Can have hash functions for strings, numbers
  - Comparison function
    - Tells us whether two keys are equivalent
  - Hash Table T: An array of size m

Example

- We design a hash table for storing items where a phone number is a ten-digit positive integer
- Our hash table uses an array of size N = 10,000 and the hash function $\boldsymbol{h}(\boldsymbol{x})$ = last four digits of $\boldsymbol{x}$



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code map:
  $\boldsymbol{h}_1$: keys → integers

  Compression map:
  $\boldsymbol{c}_2$: integers → [0, N - 1]

- The hash code map is applied first, and the compression map is applied next on the result, i.e.,

  $$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{c}_2(\boldsymbol{h}_1(\boldsymbol{x}))$$

Hash Functions

- **Direct addressing** - storing items in an array that is the size of the largest key
  - 350 million social security numbers, 9 digits
  - Wastes space if the number of items is small
  - 000-00-0000 to 999-99-9999, uses 1,000,000,000 cells

Hash Functions

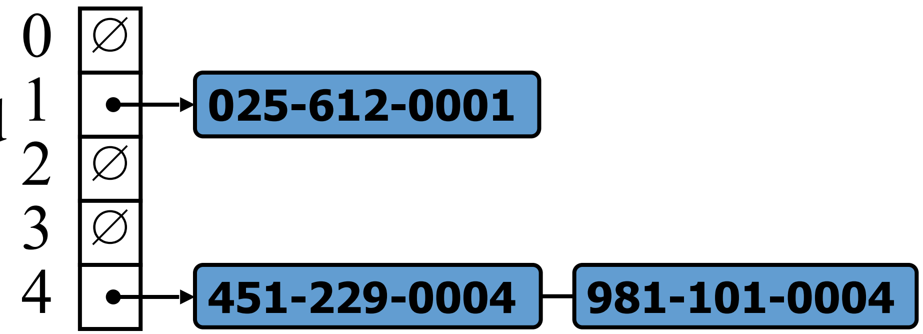
- Good hash function
  - Table size close to the number of items
    - 1 slot for each item
  - Good distribution of items
    - Ideally perfect distribution
  - Easy to compute
    - Speed
  - Ensure any two distinct items get different mappings

Perfect Hashing

- **Perfect hashing** - no two items hash to the same location
- Impossible to give a unique mapping to an infinite number of items stored in a fixed size array.
- Function should attempt to distribute items uniformly.

Collision Resolution

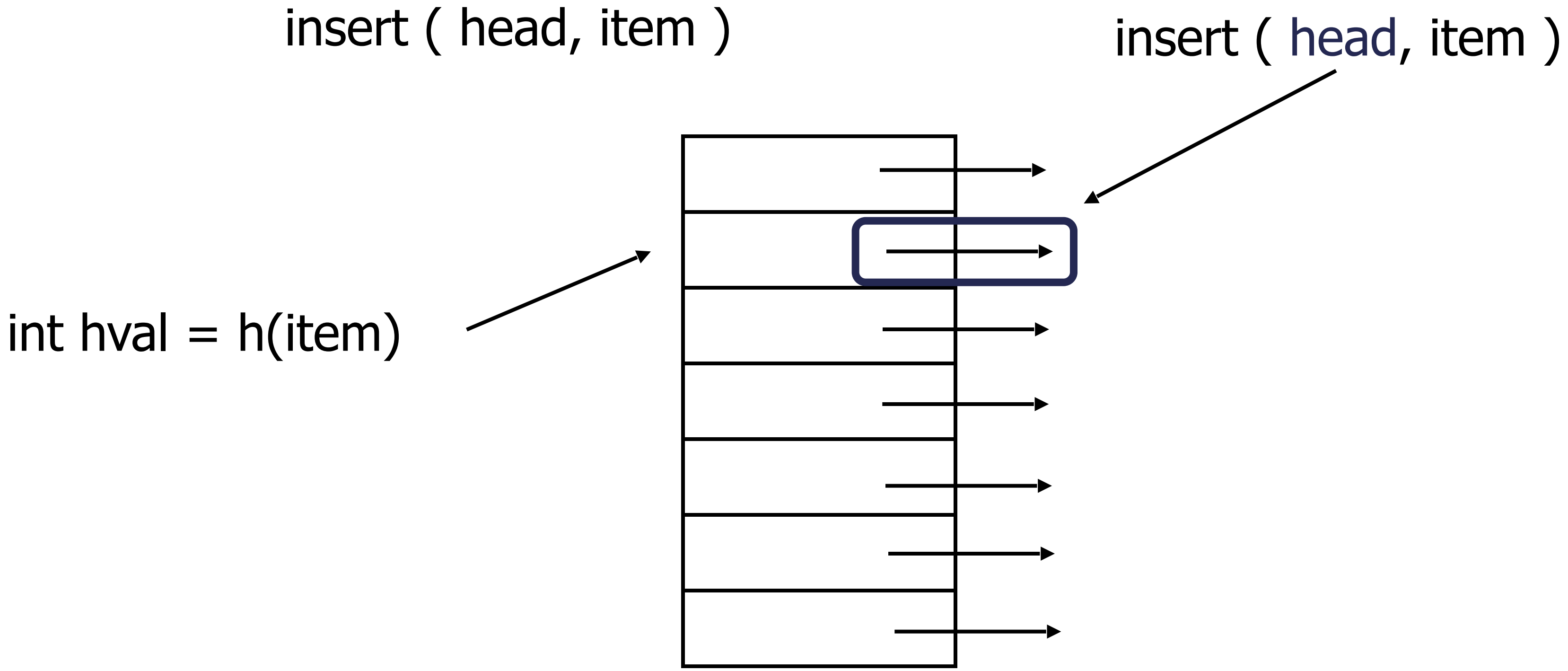
- **Collisions** occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to a linked list of elements that map there



- Chaining is simple, but requires additional memory outside the table

Separate Chaining

```
void insertChainedHash ( itemtype item )
        int hval = h(item)
        Node* head = hashTable[hval]
        insert ( head, item )
```



Separate Chaining

- In class exercise - hash the following values:

  6, 45, 754, 34, 4, 66

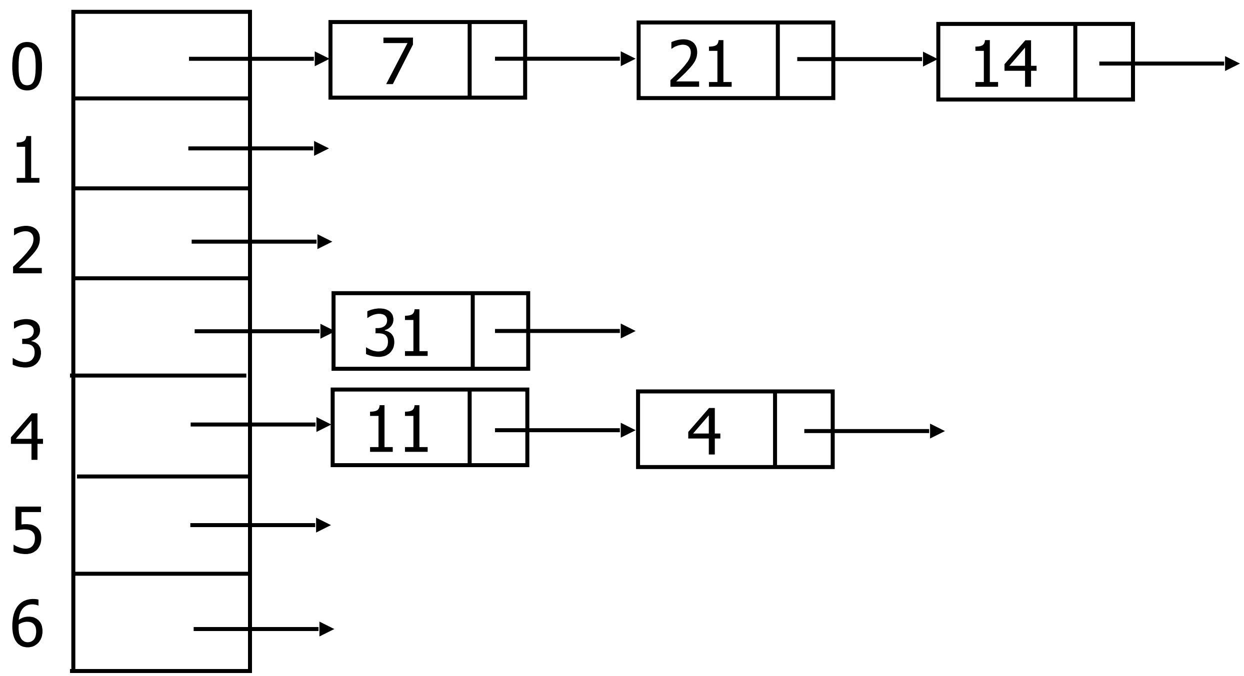
  into a table of size 10, using hash function:

  h(key) = key% SIZE
- Use push back to insert into the chain

Separate Chaining

7, 11, 21, 31, 4, 14

Separate Chaining



Hash Tables – Separate Chaining

- n = number of elements stored in the hash table
- m = size of the hash table
- $\alpha = n/m$ (load factor) - average number of items stored in a chain

Separate Chaining Performance

- **Worst case**
  - All n keys map to the same location creating list of length n
  - Search is $\Theta(n)$, plus time to compute hash function
  - No better than using a linked list
- **Average case**
  - Depends on how uniformly hash function h distributes set of keys among m slots in table
  - **Simple Uniform Hashing** – each key is equally likely to map to any location in m, independent of where any other key has already hashed
  - **Search** - $\Theta(1+\alpha)$, under assumption of Simple Uniform Hashing

Hashing Methods for Strings

- Use ASCII values of characters
- Method 1
  - Hash on ASCII value of the first character
    - Clumping on particular characters and some may be empty
- Method 2
  - Hash on ASCII values of characters added up (sum)
    - If table size is large (i.e. 10,007), then does not take advantage of all slots
    - All keys are string of less than 8 characters
    - ASCII chars have integer value 1-127
    - 127*8 = 1016
    - **1,016 / 10,007 = 0.10152893**

Hashing Methods for Strings

- Method 3
  - Look at the first 3 characters
  - str[0] + str[1]*X + str[2]*X*X
  - Good for random characters and large table sizes
- Method 4
  - Same as method 3 but with all characters
  - Calculation more expensive than method 3.

Polynomial Accumulation

- Partition the bits of the key into a sequence of components of fixed length (8,16, or 32 bits)

  $a_0 a_1 \ldots a_{n-1}$

- Evaluate the polynomial

  $$p(x)=a_0+a_1x+a_2x^2+\ldots+a_{n-1}x^{n-1}$$

  at fixed value x

- Use Horner's rule for polynomial time evaluation

  $p_0(x)=a_{n-1}$

  $p_i(x)=a_{n-i-1}+xp_{i-1}(x)$

Open Address Hashing

- All items are stored in the hash table itself
- If a collision occurs, use probing to try another cell until an empty cell is found
- Techniques for computing probe sequence
    - Linear probing
    - Quadratic probing
    - Double hashing
- Load factor $\alpha = (n/m)$ is always $<= 1$
    - n = number of items in hash table
    - m = size of the hash table

Open Address Hashing Collision Resolution

- Linear probing

  $h(k, i) = (h'(k) + i) \bmod m$

- Quadratic probing

  $h(k, i) = (h'(k) + i^2) \bmod m$

- Double hashing

  $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

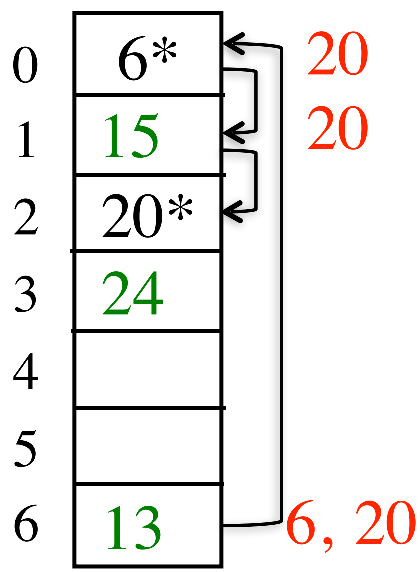
Linear Probing

- $h(k, i) = (h'(k) + i) \bmod m$
- Initial probe goes to $T[h'(k)]$
- $i = 0, 1, 2, \ldots, m-1$
- On collisions probe:
  - $T[h'(k)+1], T[h'(k)+2], \ldots, T[m-1]$
  - Circular-array, wrap around
  - $T[0], T[1], \ldots, T[h'(k)-1]$
- **Primary clustering** - long runs of occupied cells increase average search time

Linear Probing

keys:
13, 15, 24, 6, 20

h(key, i) = (h'(key) + i) % size
h'(key) = key % size



((13 % 7) + 0) % 7 = 6

((15 % 7) + 0) % 7 = 1

((24 % 7) + 0) % 7 = 3

((6 % 7) + 0) % 7 = 6 - COLLISION

((6 % 7) + 1) % 7 = 0 - PROBE*

((20 % 7) + 0) % 7 = 6 - COLLISION

((20 % 7) + 1) % 7 = 0 - PROBE

((20 % 7) + 2) % 7 = 1 - PROBE

((20 % 7) + 3) % 7 = 2 - PROBE*

Linear Probing

- As long as the table is large enough, a free cell will always be found
- Even if the table is relatively empty, it may take many probes
- Primary clustering
  - Results from clumping due to linear probing
  - Long chains of occupied slots fill up
  - Causes an increase in average search time
- Deletions require extra management
  - Items deleted may have caused collisions prior
  - Deleted items open up slots in hash table to store new entries

Primary Clustering

- In class activity - hash the following values using linear probing into a hash table of size 25

keys:
0, 25, 50, 75, 100

Primary Clustering

Primary Clustering

keys:
0, 25, 50, 75, 100

0	0
1	25
2	50
3	75
4	100
5	
⋮	
24	

((0%25) + 0)%25 = 0

((25%25) + 0)%25 = 0 - COLLISION

((25%25) + 1)%25 = 1 - PROBE

((50%25) + 0)%25 = 0 - COLLISION

((50%25) + 1)%25 = 1 - PROBE

((50%25) + 2)%25 = 2 - PROBE

((75%25) + 0)%25 = 0 - COLLISION

((75%25) + 1)%25 = 1 - PROBE

((75%25) + 2)%25 = 2 - PROBE

((75%25) + 3)%25 = 3 - PROBE

((100%25) + 0)%25 = 0 - COLLISION

((100%25) + 1)%25 = 1 - PROBE

((100%25) + 2)%25 = 2 - PROBE

((100%25) + 3)%25 = 3 - PROBE

((100%25) + 4)%25 = 4 - PROBE

Primary Clustering

- In class exercise - hash the following values using linear probing into a hash table of size 25

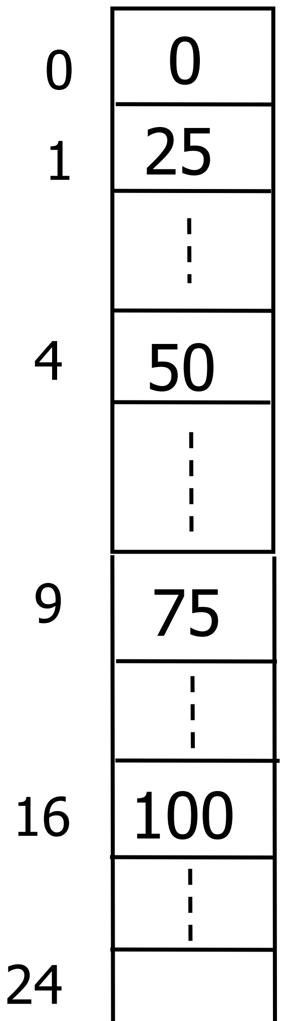
keys:
0, 25, 50, 75, 100

Quadratic Probing

- $h(k, i) = (h'(k) + i^2) \bmod m$
- Initial probe goes to $T[h'(k)]$
- Next probe $T[h'(k)+1^2]$, $T[h'(k)+2^2]$, $T[h'(k)+3^2]$, …
- **Secondary clustering** – Two keys with same initial probe position, will have same probe sequence.

If $h(k_1, 0) == h(k_2, 0)$ then $h(k_1, i) == h(k_2, i)$

Quadratic Probing



$((0\%25) + 0^2)\%25 = 0$

$((25\%25) + 0^2)\%25 = 0$ - COLLISION

$((25\%25) + 1^2)\%25 = 1$ - PROBE

$((50\%25) + 0^2)\%25 = 0$ - COLLISION

$((50\%25) + 1^2)\%25 = 1$ - PROBE

$((50\%25) + 2^2)\%25 = 4$ - PROBE

$((75\%25) + 0^2)\%25 = 0$ - COLLISION

$((75\%25) + 1^2)\%25 = 1$ - PROBE

$((75\%25) + 2^2)\%25 = 4$ - PROBE

$((75\%25) + 3^2)\%25 = 9$ - PROBE

$((100\%25) + 0^2)\%25 = 0$ - COLLISION

$((100\%25) + 1^2)\%25 = 1$ - PROBE

$((100\%25) + 2^2)\%25 = 4$ - PROBE

$((100\%25) + 3^2)\%25 = 9$ - PROBE

$((100\%25) + 4^2)\%25 = 16$ - PROBE

Quadratic Probing

- In class exercise - hash the following values in the order shown into a table of size 7 using quadratic probing

  0, 6, 7, 3, 14, 13

Quadratic Probing

Quadratic Probing

0	0
1	7
2	
3	3
4	14
5	
6	6

$((0\%7) + 0^2)\%7 = 0$

$((6\%7) + 0^2)\%7 = 6$

$((7\%7) + 0^2)\%7 = 0$ - COLLISION

$((7\%7) + 1^2)\%7 = 1$ - PROBE

$((3\%7) + 0^2)\%7 = 3$

$((14\%7) + 0^2)\%7 = 0$ - COLLISION

$((14\%7) + 1^2)\%7 = 1$ - PROBE

$((14\%7) + 2^2)\%7 = 4$ - PROBE

$((13\%7) + 0^2)\%7 = 6$ - COLLISION

$((13\%7) + 1^2)\%7 = 0$ - PROBE

$((13\%7) + 2^2)\%7 = 3$ - PROBE

$((13\%7) + 3^2)\%7 = 1$ - PROBE

$((13\%7) + 4^2)\%7 = 1$ - PROBE

$((13\%7) + 5^2)\%7 = 3$ - PROBE

$((13\%7) + 6^2)\%7 = 0$ - PROBE

$((13\%7) + 7^2)\%7 = 6$ - PROBE

and so on…

Double Hashing

- Have a hash function that produces a series of values and place item in first available slot
- $(i + j\,h'(k))\ mod\ m$ for j = 0,1,…m-1 where m is prime
- i is the value of the original hash function
- Secondary hash function h'(k) can't have zero values
- Table size m must be prime to allow probing of all cells

Double Hashing

- $h(k, i) = (h_1(k) + i\ h_2(k))\ mod\ m$
- Initial probe goes to $T[h_1(k)]$
- Successive probe positions offset from previous position by amount $h_2(k)$, $mod\ m$
- Value $h_2(k)$ must be relatively prime to hash-table size m for all cells to be probed
- Choose m to be prime number and h_2 to always produces positive integer $< m$

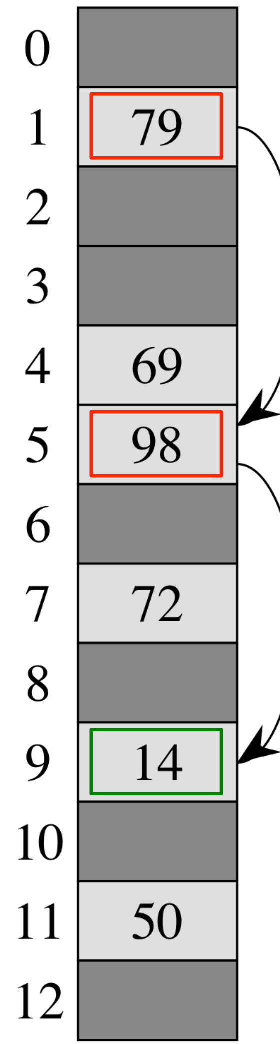
  OR
- Make m power of 2 and h_2 to produce only odd numbers only

Double Hashing

Attempt to insert 14…

$h_1(k) = (k \bmod 13)$

$h_2(k) = (1 + k \bmod 11)$



$h_1(14) = 14 \bmod 13 = 1$

Collision at T[1]

$h_2(14) = (1 + 14 \bmod 11) = 4$

Probe 4 positions away

Collision at T[5]

Probe 4 positions away

Insert 14 at T[9]

Resizing Hash Tables

- **Open Address** - If the table becomes too full, insertions can begin to take a long time or be denied
- Threshold ≈ 70% full
- Double the table size
- Rehash the keys into new table
- De-allocate old table if necessary
- Rehashing takes O(n)

Resizing Hash Tables

- **Separate Chaining** – Resize table when linked lists begin to get long
- Rebuild linked list of key/value pairs for the new table
- De-allocate old table if necessary
- Rehashing takes O(n)

Performance

- Worst case: everything falls into the same bin
- Searches, removals take O(n) time
- Insertions may also take O(n) time
- Load factor alpha=n/m - affects performance
- Can show that expected number of probes is 1/(1-alpha) (for hash tables that store all items in array)
- Expected time is O(1)

Universal Hash Functions

- Idea - hash functions that produce a uniform flat distribution across the array
- Formally, for 0 <= i, j <= M-1, Pr(h(i)=h(j)) <=1/N
- Choose p as a prime between M and 2M.
- Randomly select $0<a<p$ and $0\leq b<p$, and define $h(k)=(ak+b \bmod p) \bmod N$