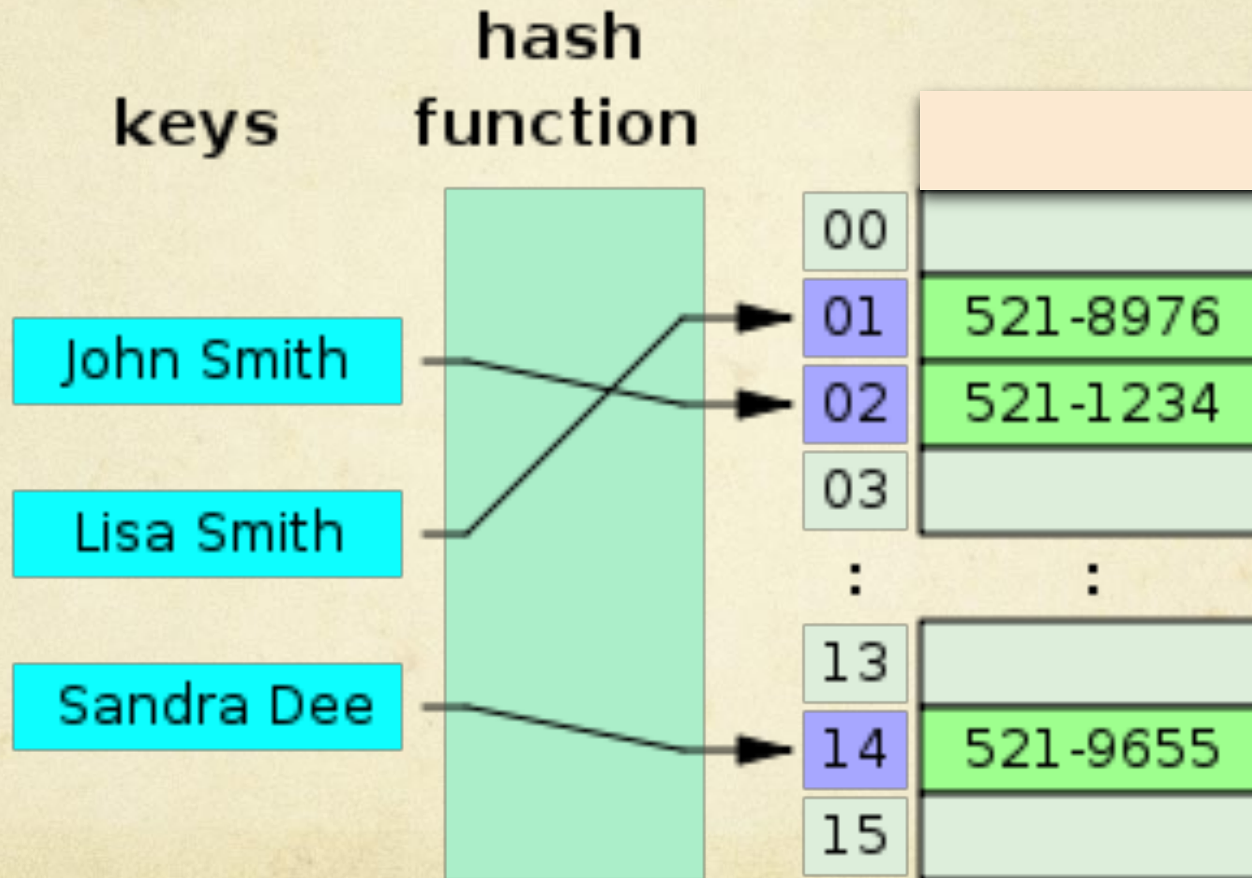
Data Structures

Giri Narasimhan
Office: ECS 254A
Phone: x-3748
giri@cs.fiu.edu

Standard Data Structures

- 3 operations
  - Insert, delete, find
- We want to make them as efficient as possible
- Best we have so far is AVL trees
  - All 3 operations take O(log n) time
  - General idea is to organize data so that
    - Search is easier
    - Insert to and delete from place where you would search
- What if you knew exactly where to search/insert/delete
  - Idea; Use the value to decide where to place

Hashing: Key value to location



http://i.stack.imgur.com/2Saxe.png

Let "value" equal location

- Use SSN or birthdate as location for student record
- Assume chances of "collision" is close to zero
  - Insert: place the record in appropriate location
  - Find: if appropriate location occupied – then found! Else not found
  - Delete: if appropriate location occupied – then delete item. Else nothing to delete
- Each operation O(1) time – incredibly efficient
- Memory: array of size 10,000 or 365 even if only 10 students

Let "value" determine location

- Apply a hash function to value and use it as location
  - Hash value: $h(x) = x \bmod b$
  - Hash value: $h(x) = ax \bmod b$
  - Hash value: $h(x) = h_1(h_2(x))$
  - Middle digits of x^2. For example, $4567^2 = 20857489$
    - h(4567) = 57
- Assume that hash function has following properties:
  - hashes each value to a unique location
  - values in a given domain are hashed to a location uniformly at random in a given range
  - Hash table size ≈ twice number of items to insert

Simple hash functions

$$hashValue(x) = x \% tableSize$$

- Let tableSize = 100
  - X = 173, hashValue(X) = 73
  - X = 3452, hashValue(X) = 52
  - X = 9758, hashValue(X) = 58
  - X = 800, hashValue(X) = 0

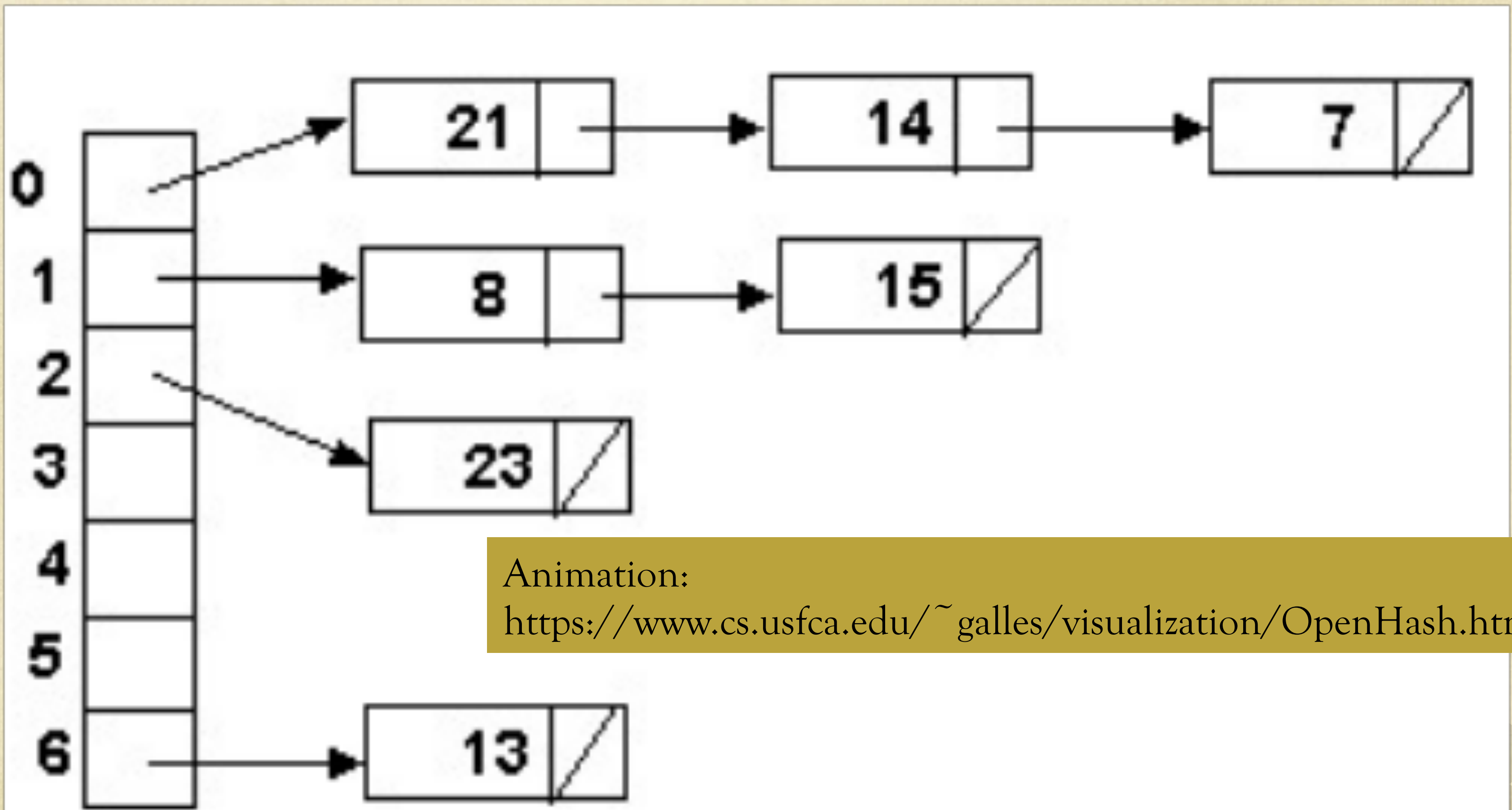
$$hashValue(x) = x_3S^3 + x_2S^2 + x_1S^1 + x_0S^0 \% tableSize$$

- Let S = 128
  - X = "comb"
    $hashValue(X) = ('c'\ 128^3 + 'o'\ 128^2 + 'm'\ 128^1 + 'b'\ 128^0) \% tableSize$
  - X = "eye"
    $hashValue(X) = ('e'\ 128^2 + 'y'\ 128^1 + 'e'\ 128^0) \% tableSize$

Collision Resolution

- Collision: when two items hash to the same location
- Many resolution methods exist
  - Chaining
  - Open Addressing
  - Bucketing
  - Double Hashing
  - Overflow

Separate Chaining



Animation:
https://www.cs.usfca.edu/~galles/visualization/OpenHash.html

http://i.stack.imgur.com/CSb6Y.png

Separate Chaining

- Best when stored in main memory. Disk-based separate chaining is not efficient
- If N items stored in table of size M, then average list length is $O(N/M)$ = average time complexity for search
- Average Time Complexity = $O(1)$, if $M = O(N)$
- Worst-Case Time Complexity = length of longest chain
- Theorem: Expected length of longest chain = $O(\log N)$

Bucket Hashing

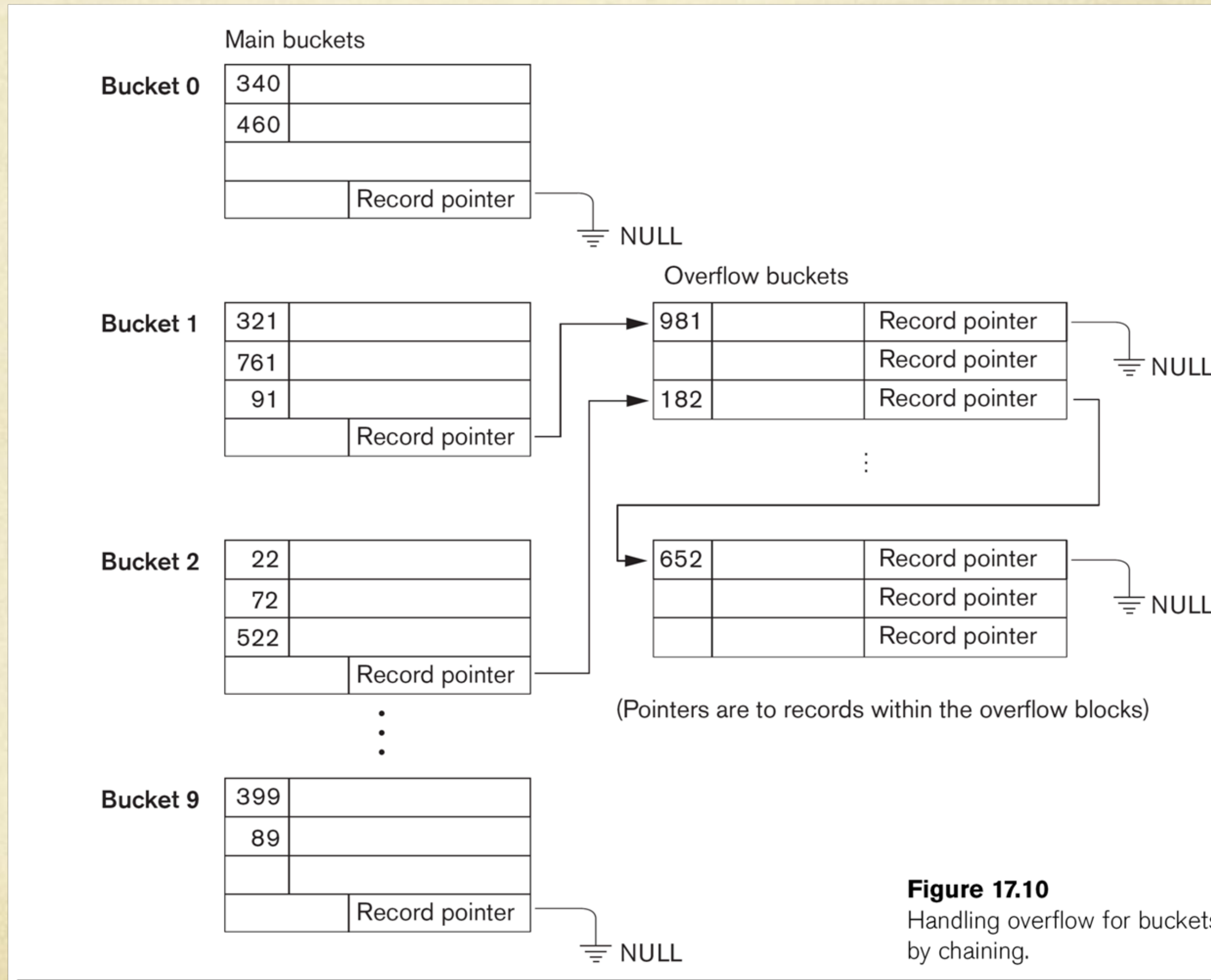


Figure 17.10
Handling overflow for buckets by chaining.

http://ulam2.cs.luc.edu/353/spr13/notes/images/fig17.10.png

Open Addressing / Linear Probing

HASHING with LINEAR PROBING

m = 20, f(k) = k % m

Initial Situation

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Value	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1	−1

After inserting 34, 55, 12, 8, 45, 37, 32, 88, 98, 54

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Value	−1	−1	−1	−1	−1	45	−1	−1	8	88	−1	−1	12	32	34	55	54	37	98	−1
Hops						0			0	1			0	1	0	0	2	0	0	

After inserting 21, 42, 56, 74, 52, 33, 16

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Value	74	21	42	52	33	45	16	−1	8	88	−1	−1	12	32	34	55	54	37	98	56
Hops	6	0	0	11	11	0	10		0	1			0	1	0	0	2	0	0	3

Red numbers give number of hops in probe sequence

https://www8.cs.umu.se/~jopsi/dinf504/hashing_probe.gif

Open Addressing / Linear Probing

- Insert: If hash location is "occupied", place item in first empty location scanning from hash location
- Find: If item is not in correct location, search for item by scanning from hash location until first empty location

Problems with Linear Probing

- Clustering – also called Primary Clustering
  - Clusters tend to get larger because probability of collision increases with cluster size.
    - http://www.cs.armstrong.edu/liang/animation/web/LinearProbing.html
    - https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html
  - Small clusters merge to become large clusters, causing secondary clustering.
  - Making table larger will reduce collisions, but is wasteful
  - Handling deletions is a problem

Problems with Linear Probing

- PRIMARY CLUSTERING
  - Large blocks of occupied cells are formed.
  - Amount of clustering and size of clusters is dependent on LOAD FACTOR (fraction of table that is occupied).
  - It deteriorates the performance.

- NAÏVE ANALYSIS:
  - If load factor is F, and table size is T, then the average time for search is FT.
    - INCORRECT !!
  - If load factor is F, then the average time for search is:
    - $1 + 1/(1-F)^2)/2$
  - If F = 50%, then the average cluster time is 2.5
  - If F = 90%, then the average cluster time is 50.5

Clustering

- Linear Probing leads to primary clustering
- LINEAR PROBING: Try H, H+1, H+2, H+3, ...
- QUADRATIC PROBING: Try H, $H+1^2$, $H+2^2$, $H+3^2$, ...
  - Seems to eliminate primary clustering
- Linear Probing also leads to secondary clustering
  - This is when large clusters merge to become larger clusters.
  - It is not clear if quadratic probing eliminates it.
- DOUBLE HASHING: Try $H_1(x)$, $H_1(x) + H_2(x)$, $H_1(x) + 2H_2(x)$, $H_1(x) + 3H_2(x)$, ...
  - This is an improvement over quadratic probing. But more expensive to implement.
- SEPARATE CHAINING: need linked list or dynamic arrays.

Handling Deletions

- Straightforward in Separate Chaining
- Challenges in Open Addressing
  - Upon collision, the value is stored in first open location.
  - Problem: if an item is deleted, it might appear as if there is no other item that mapped to that location, and a find operation would return "NOT FOUND"
  - Solution: Upon deletion, leave a place holder to indicate this used to be occupied.

Deletions & Performance

- DELETES:
  - Need to be careful to leave a "marker".
- OPTIMAL VALUES OF LOAD FACTORS
- Doubling table size if load factors become high.
- REHASHING
- Hashing works very well in practice, and is widely used.
- Used to implement SYMBOL TABLES in compilers and various software systems.
- How does it compare to BST?
  - O(log N) versus O(1)

Figure 20.5
Illustration of primary clustering in linear probing (b) versus no clustering (a) and the less significant secondary clustering in quadratic probing (c). Long lines represent occupied cells, and the load factor is 0.7.

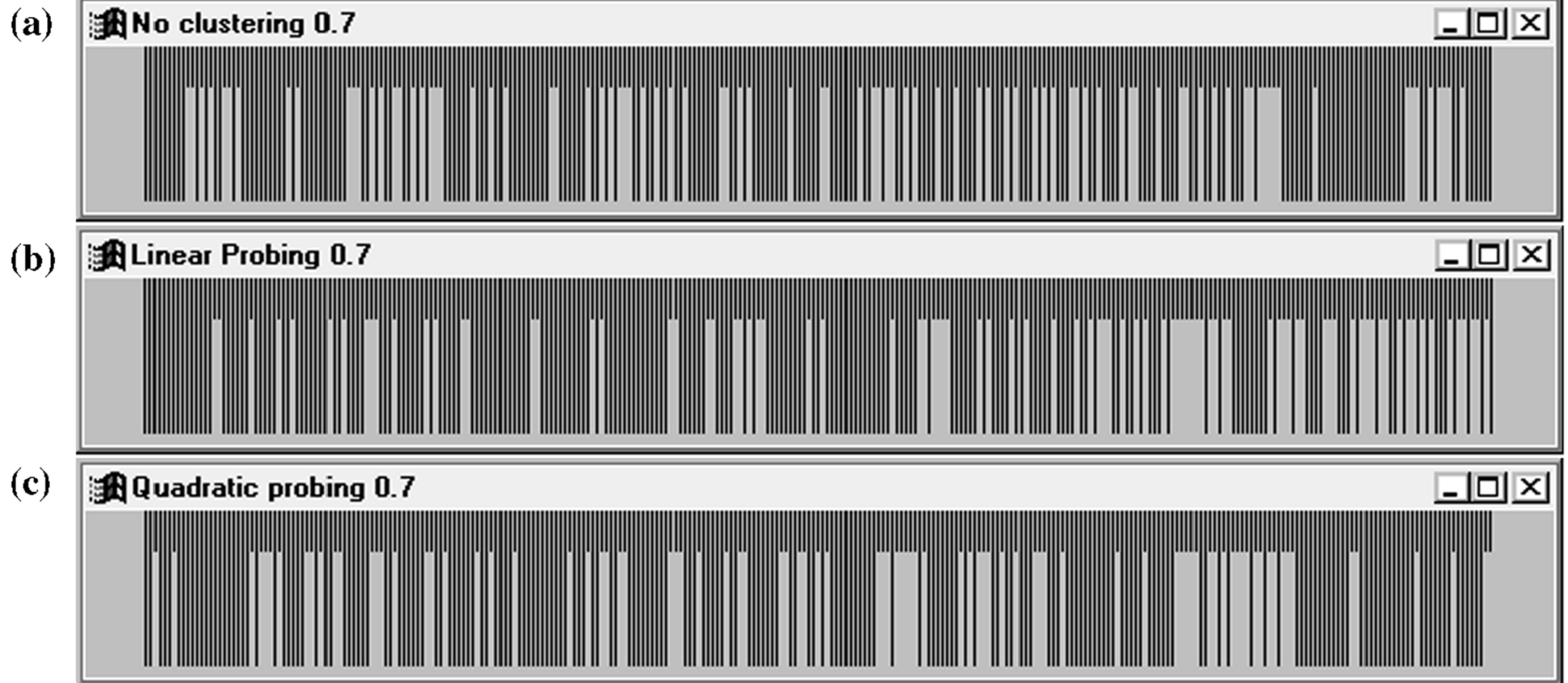


Figure 20.4
Linear probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

 © 2002 Addison Wesley

Figure 20.6
A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

 © 2002 Addison Wesley