Hash Table

- Data Structure for:
  - Insert
  - Search or retrieve
  - Delete
- Very efficient
- Content-based data structure
  - Use value as an index
    - Works if range of values are small
  - Use HASH value as an index
    - Works if HASH function is "good"
- A COLLISION occurs when two values have the same HASH value
- A "good" HASH function is one that causes few or no COLLISIONS.

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

- Perfect hash functions: no collisions.
- Perfect hash functions can be built if the input data is known beforehand. But they are difficult to design.
- For perfect hash functions, all operations can be perfromed in O(1) time.
- If input is not known beforehand, then perfect hash functions are impossible to design.
- So collisions are inevitable.
- How to deal with collisions?
- LINEAR PROBING:
  - If the location where an item is to be inserted is already occupied (COLLISION), then scan sequentially until an empty location is found, and insert new item there.

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

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.

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.

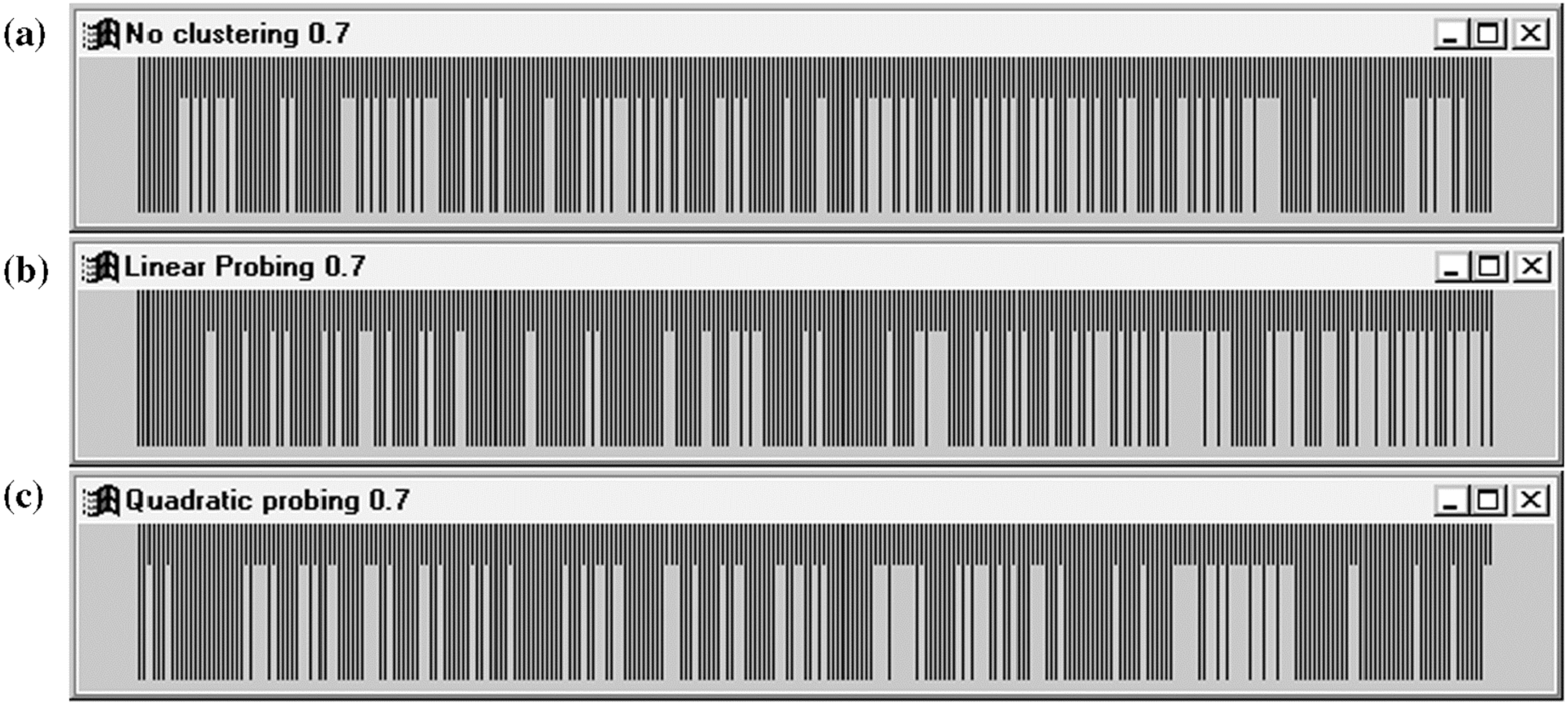

Figure 20.6

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