Hash Tables

SWE2016-44

Problem

Suppose we are storing employee records, the primary key for which is employee's telephone number.

1) **Insert Employee Record**

2) **Search for an Employee**

3) **Delete Employee Record**

Telephone	Name	City	Dept	…
9864567654	Sam	NYC	HR	
9854354543	Tom	DC	IT	
…				

Solutions

1. **Use an array**
   - **Search takes linear time**

   - **If stores in sorted order, search can be done in O(log n) using binary search but then insertion and deletion becomes costly.**

2. **Use a Linked List**
   - **Search takes linear time**

Solutions

3. **Use balanced BST to store records**
    - **Insertion takes O(log n)**
    - **Search takes O(log n)**
    - **Deletion takes O(log n)**

4. **Create a Direct Access table**
    - **Insertion takes O(1)**
    - **Search takes O(1)**
    - **Deletion takes O(1)**

Solutions

Direct Access Table

Telephone	Add
9864567654	0x1…
9854354543	0x2…
…	

Name	City	Dept	…
Sam	NYC	HR	

Name	City	Dept	…
Tom	DC	IT	

Solutions

Direct Access Table

Telephone	Add
9864567654	0x1…
9854354543	0x2…
…	

Name	City	Dept	…
Sam	NYC	HR	

Name	City	Dept	…
Tom	DC	IT	

Limitations

1) Size of table ($m * 10^n$)

Solutions

Direct Access Table

Telephone	Add
9864567654	0x1…
9854354543	0x2…
…	

Name	City	Dept	…
Sam	NYC	HR	

Name	City	Dept	…
Tom	DC	IT	

Limitations

1) Size of table $(m * 10^n)$

2) Integer may not hold the size of n digits

Solutions

Direct Access Table

(Improvement)

Hashing – provides O(1) time on average for insert, search and delete

Solutions

Direct Access Table

(Improvement)

Hashing – provides O(1) time on average for insert, search and delete

Hash function – hash function maps a big number or string to a small integer that can be used as index in hash table

Solutions

Direct Access Table

(Improvement)

HASH function h(x):

$h(x) = x \bmod 7$

Solutions

Direct Access Table

(Improvement)

HASH function h(x):

$h(x) = x \bmod 7$

$x = 9864567645$
$H(x) = 9864567645 \bmod 7 = 4$

Solutions

Direct Access Table

(Improvement)

HASH function h(x):

$h(x) = x \bmod 7$

$x = 9864567645$

$H(x) = 9864567645 \bmod 7 = 4$

$x = 9854354543$

$H(x) = 9854354543 \bmod 7 = 5$

Solutions

Direct Access Table

(Improvement)

HASH function h(x):

h(x) = x mod 7

x = 9864567645
H(x) = 9864567645 mod 7 = 4

x = 9854354543
H(x) = 9854354543 mod 7 = 5

Good h(x) should

- **Be Efficiently Computable**

Solutions

Direct Access Table

(Improvement)

HASH function h(x):

h(x) = x mod 7

x = 9864567645
H(x) = 9864567645 mod 7 = 4

x = 9854354543
H(x) = 9854354543 mod 7 = 5

Good h(x) should

- **Be Efficiently Computable**
- **Uniformly distribute the keys**

Collision

Collision – Two keys resulting in same value

Collision

Collision – Two keys resulting in same value

HASH function h(x):

h(x) = x mod 7

Collision

Collision – Two keys resulting in same value

HASH function h(x):

h(x) = x mod 7

x = 9864567645
H(x) = 9864567645 mod 7 = 4

x = 9854354542
H(x) = 9854354542 mod 7 = 4

Collision

Collision Handling

- Separate Chaining
- Open Addressing

Separate Chaining

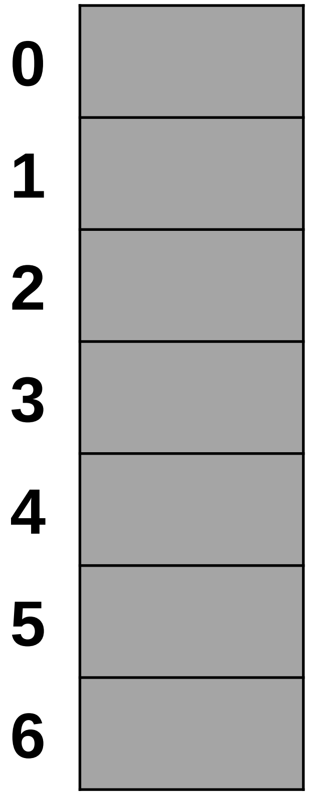
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Separate Chaining

Hash Function : $h(x) = x \bmod 7$

Separate Chaining

Hash Function : h(x) = x mod 7

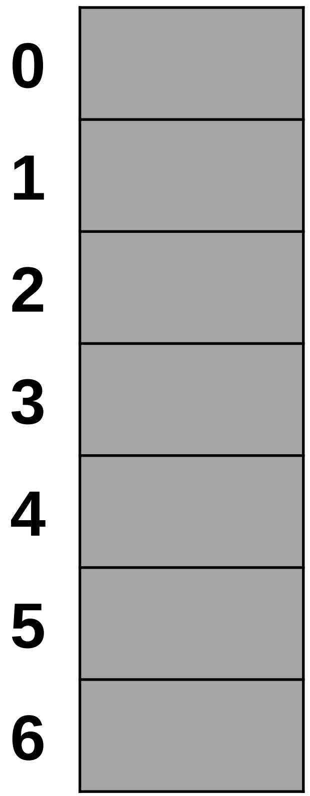


Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101



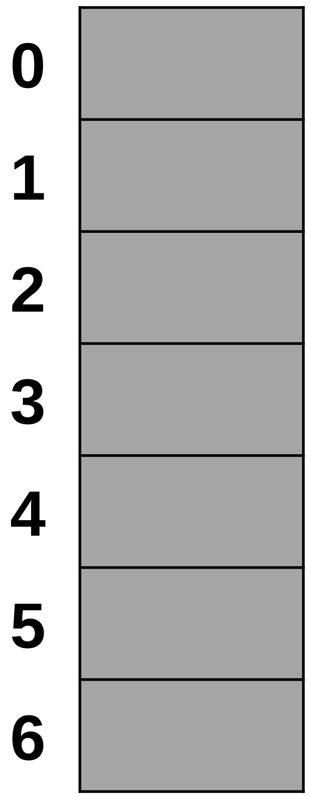
Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

50 mod 7 = 1



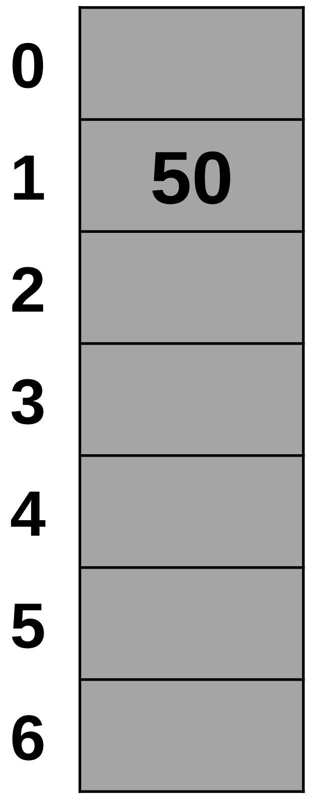
Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

50 mod 7 = 1



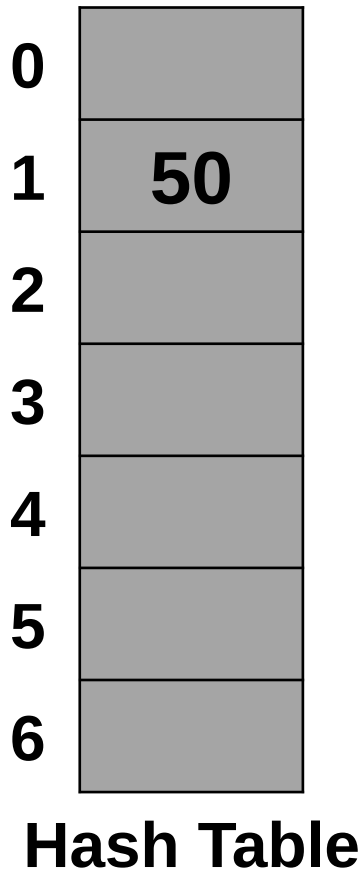
Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

700 mod 7 = 0



Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

700 mod 7 = 0

0	700
1	50
2	
3	
4	
5	
6	

Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

76 mod 7 = 6

0	700
1	50
2	
3	
4	
5	
6	

Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

76 mod 7 = 6

0	700
1	50
2	
3	
4	
5	
6	76

Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

85 mod 7 = 1

0	700
1	50
2	
3	
4	
5	
6	76

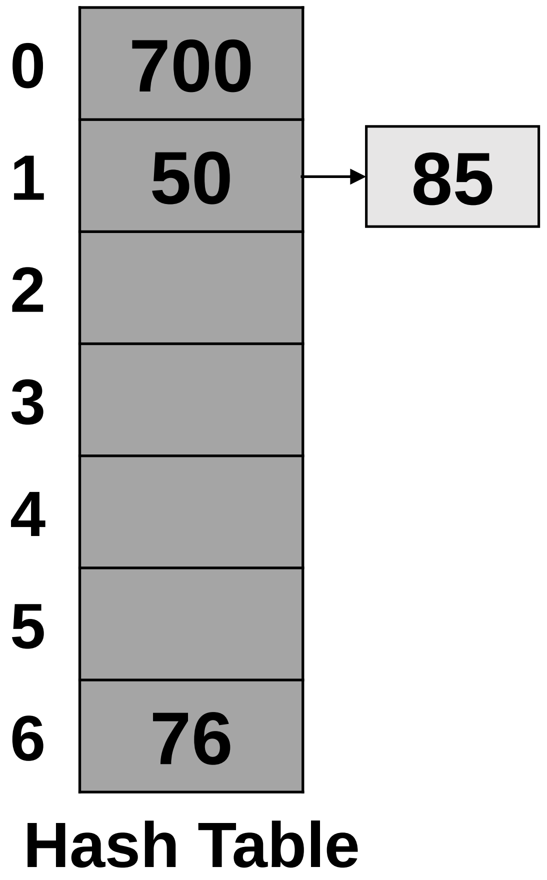
Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

85 mod 7 = 1

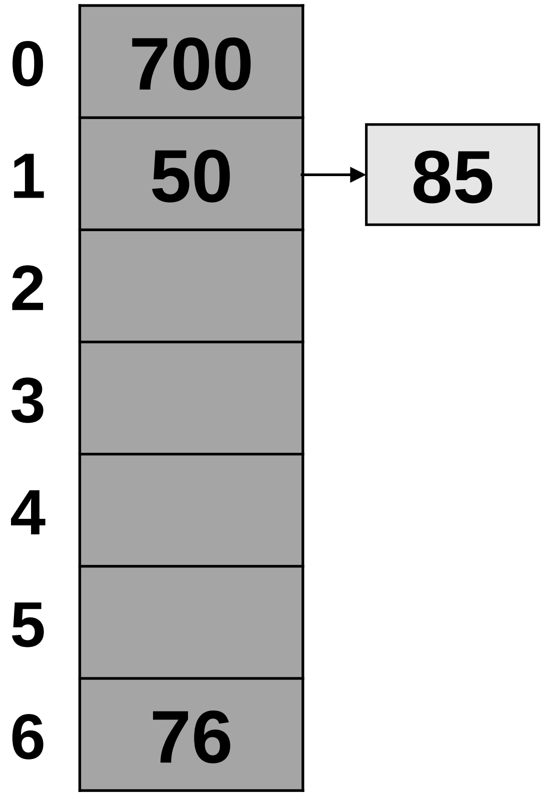


Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

92 mod 7 = 1



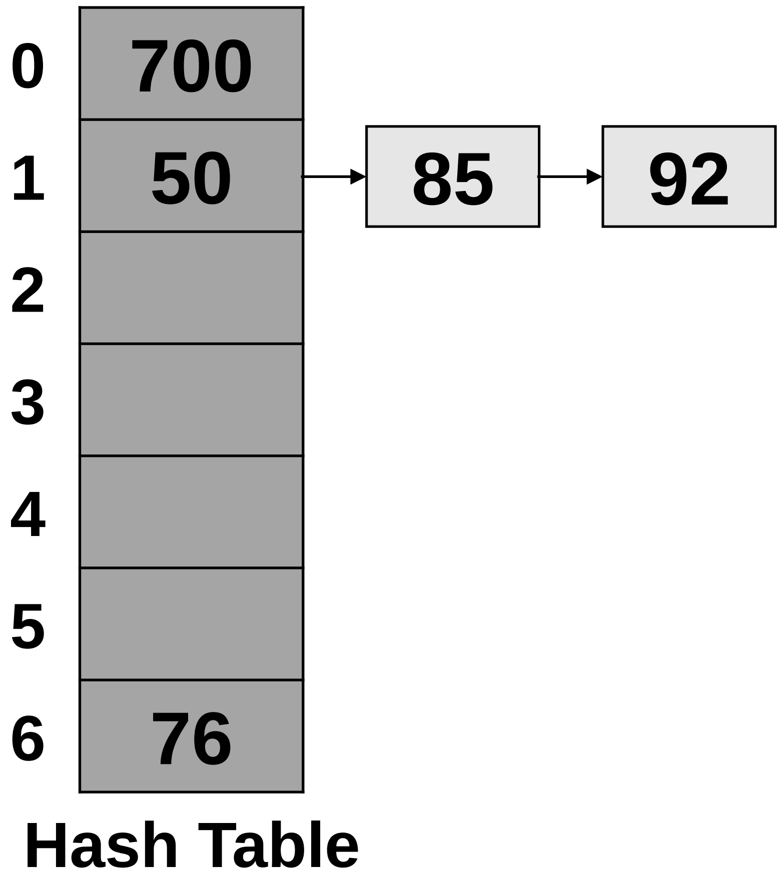
Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

92 mod 7 = 1

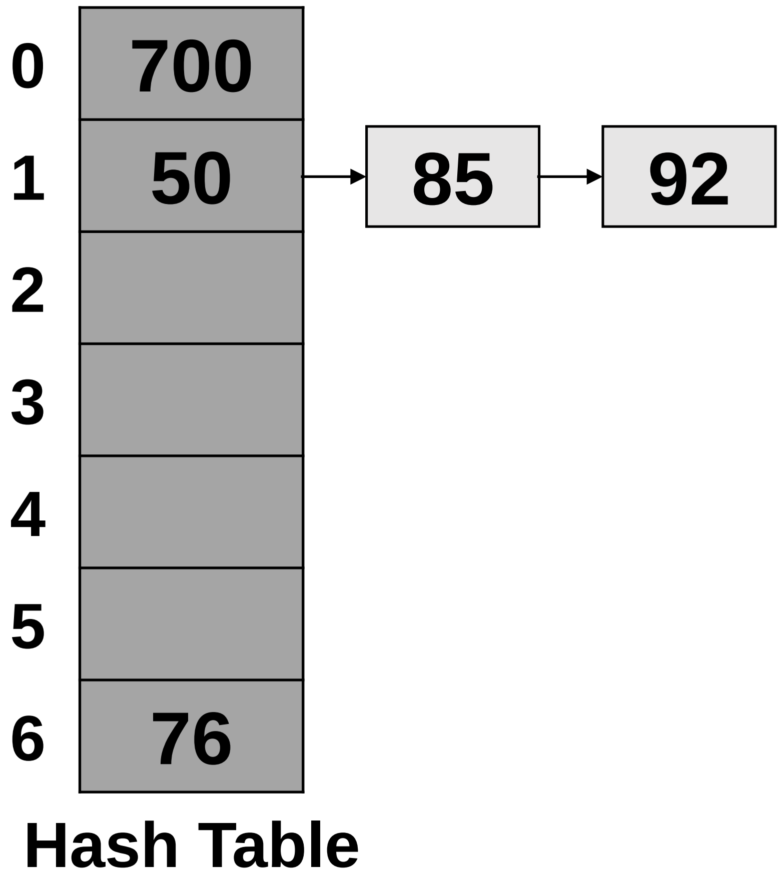


Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

73 mod 7 = 3

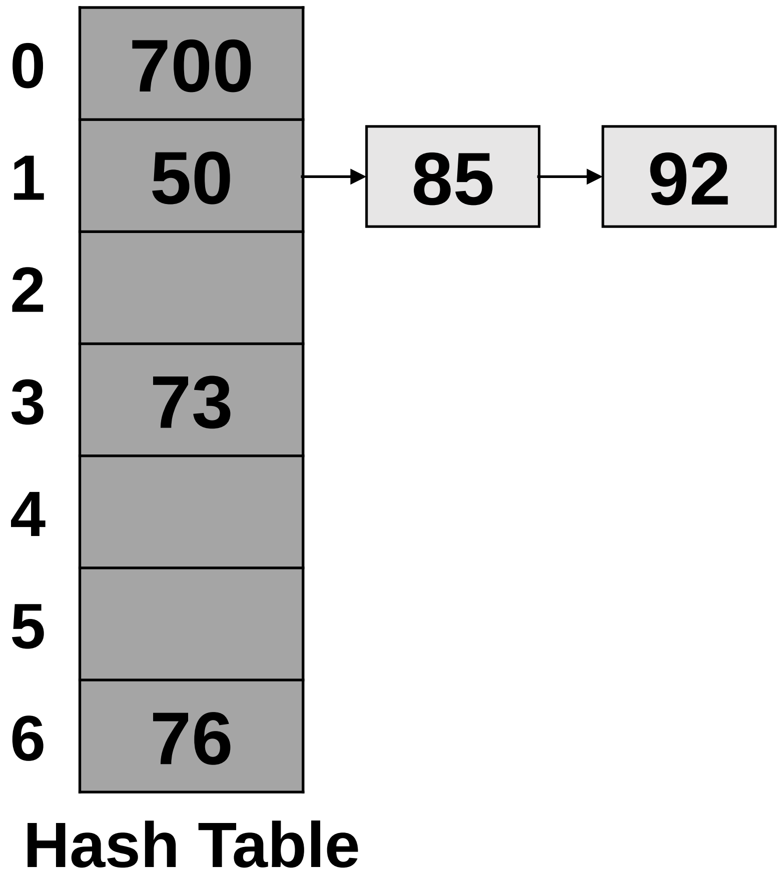


Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

73 mod 7 = 3

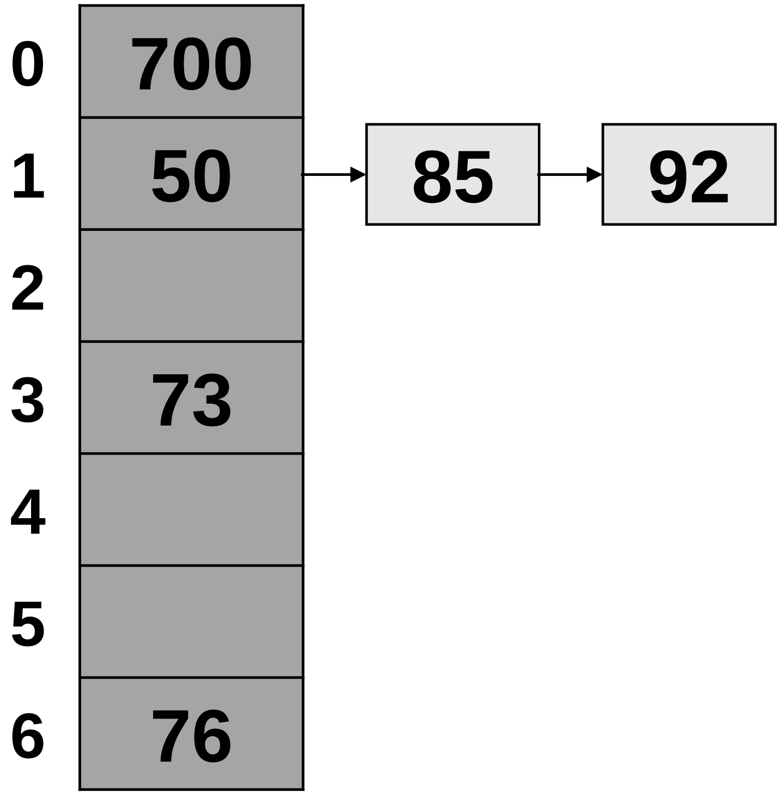


Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

101 mod 7 = 3



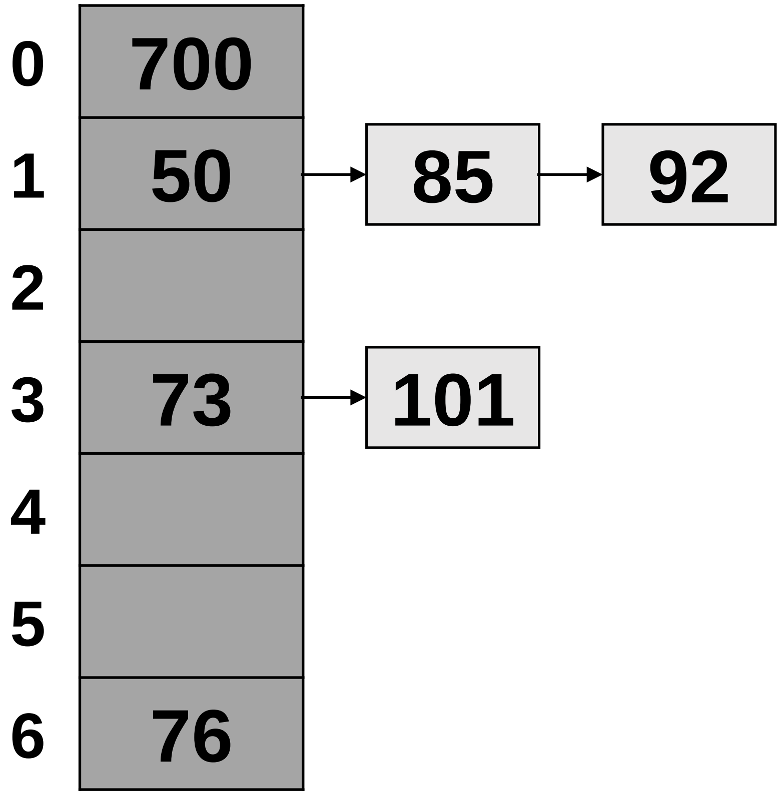
Hash Table

Separate Chaining

Hash Function : h(x) = x mod 7

Keys : 50, 700, 76, 85, 92, 73, 101

101 mod 7 = 3



Hash Table

Separate Chaining

Advantages:

1) **Simple to implement.**

Separate Chaining

Advantages:

1) **Simple to implement.**
2) **Hash table never fills up, we can always add more elements to chain.**

Separate Chaining

Advantages:

1) **Simple to implement.**
2) **Hash table never fills up, we can always add more elements to chain.**
3) **Less sensitive to the hash function or load factors.**

Separate Chaining

Advantages:

1) Simple to implement.
2) Hash table never fills up, we can always add more elements to chain.
3) Less sensitive to the hash function or load factors.
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Separate Chaining

Disadvantages:

1) **Cache performance of chaining is not good as keys are stored using linked list.**

Separate Chaining

Disadvantages:

1) **Cache performance of chaining is not good as keys are stored using linked list.**
2) **Wastage of Space.**

Separate Chaining

Disadvantages:

1) **Cache performance of chaining is not good as keys are stored using linked list.**
2) **Wastage of Space.**
3) **If the chain becomes long, then search time can become O(n) in worst case.**

Separate Chaining

Disadvantages:

1) **Cache performance of chaining is not good as keys are stored using linked list.**
2) **Wastage of Space.**
3) **If the chain becomes long, then search time can become O(n) in worst case.**
4) **Uses extra space for links.**

Separate Chaining

Complexity:

n = number of keys stored in table
m = number of slots in table
α = Average keys per slot or load factor = n/m

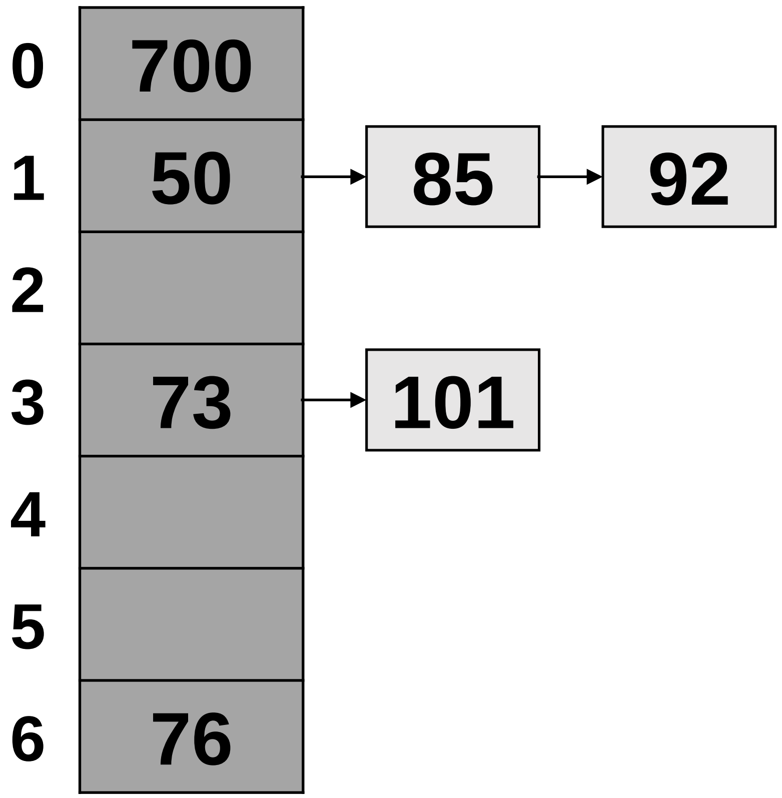
Separate Chaining

Complexity:

n = number of keys stored in table
m = number of slots in table
α = Average keys per slot or load factor = n/m

Expected time to insert/search/delete:

$O(1+\alpha)$



Hash Table

Collision

Collision Handling

- Separate Chaining
- Open Addressing

Open Addressing

A hash collision is resolved by probing

1) **Linear Probing**
2) **Quadratic Probing**
3) **Double Hashing**

Linear Probing

Linear Probing

$$\mathbf{h_i}(X) = (Hash(X) + i) \ \% \ HashTableSize$$

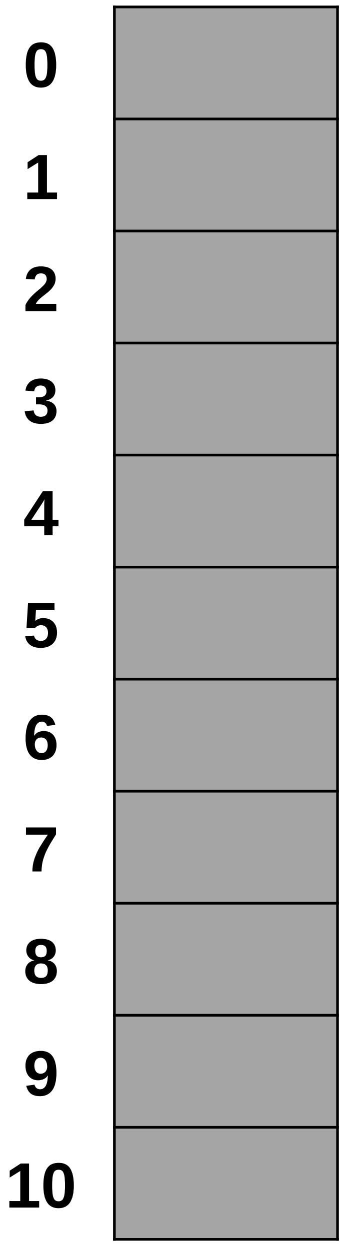
If $\mathbf{h}_0$(X) = (Hash(X) + 0) % HashTableSize is full, we try for $\mathbf{h}_1$

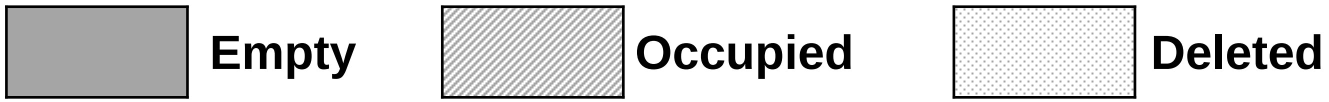
If $\mathbf{h}_1$(X) = (Hash(X) + 1) % HashTableSize is full, we try for $\mathbf{h}_2$

And so on ..

Linear Probing

Keys: 7, 36, 18, 62

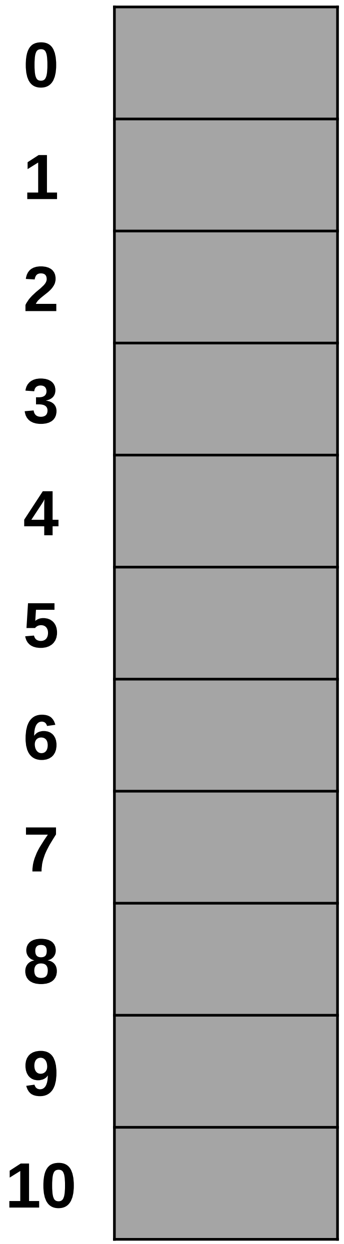


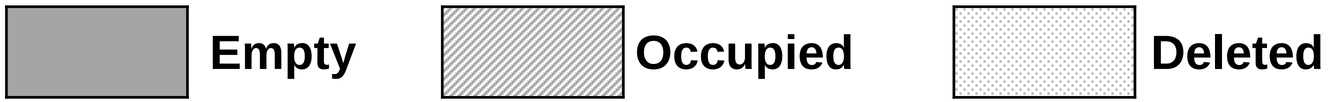


Linear Probing

Keys: 7, 36, 18, 62

Insert(7):
$h_0(7)$ = (7 mod 11) = 7



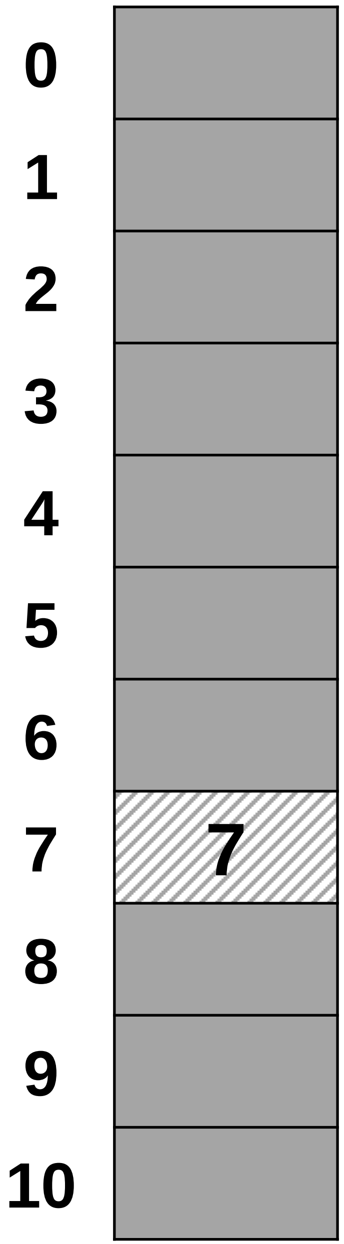


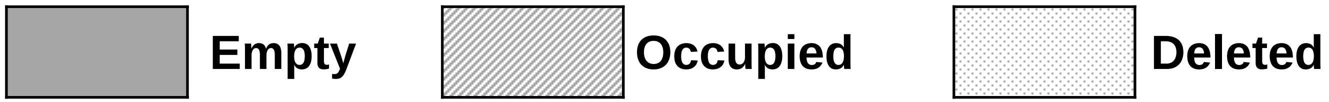
Linear Probing

Keys: 7, 36, 18, 62

Insert(7):

$\mathbf{h}_0(7) = (7 \bmod 11) = 7$

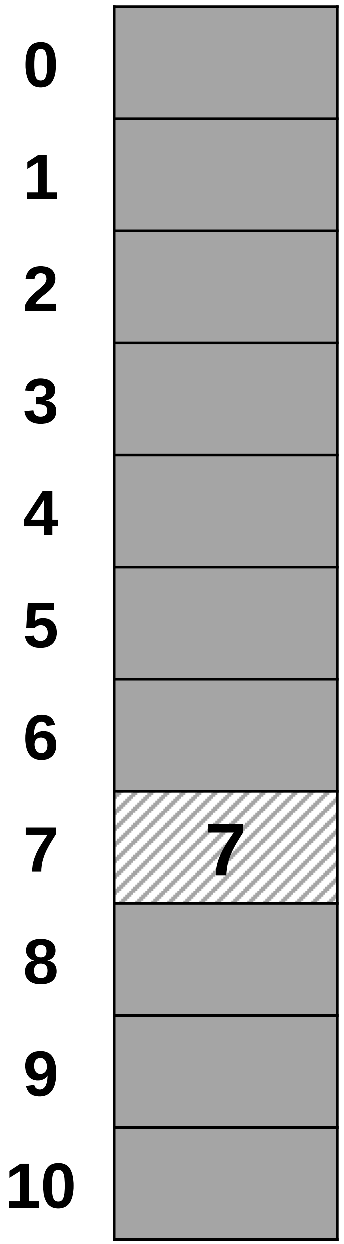


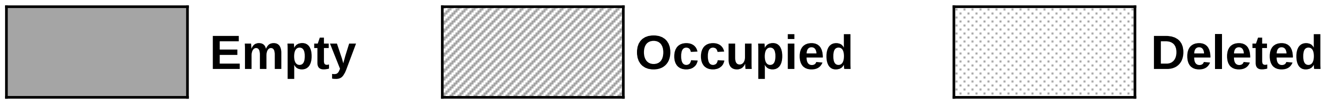


Linear Probing

Keys: 7, 36, 18, 62

Insert(36):
$\mathbf{h}_0$(36) = (36 mod 11) = 3

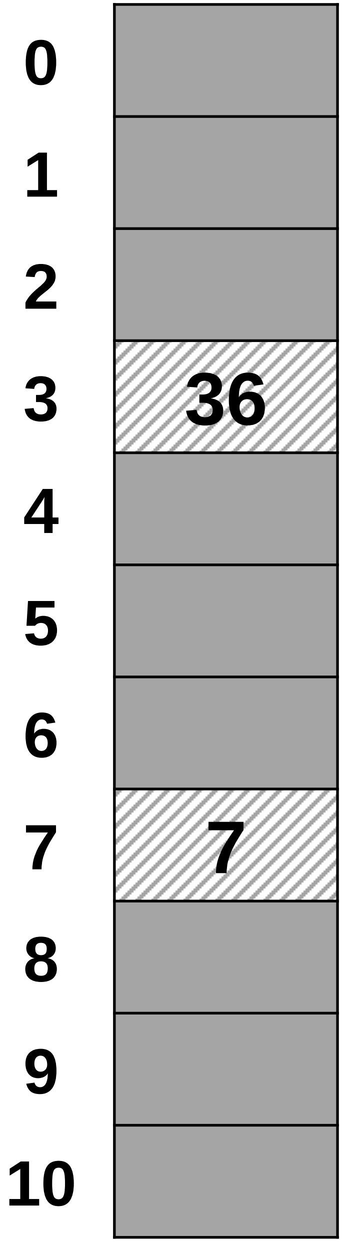


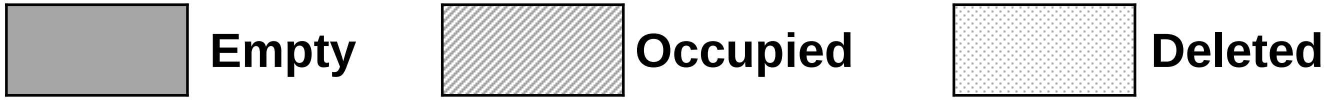


Linear Probing

Keys: 7, 36, 18, 62

Insert(36):
$\mathbf{h}_0$**(36) = (36 mod 11) = 3**

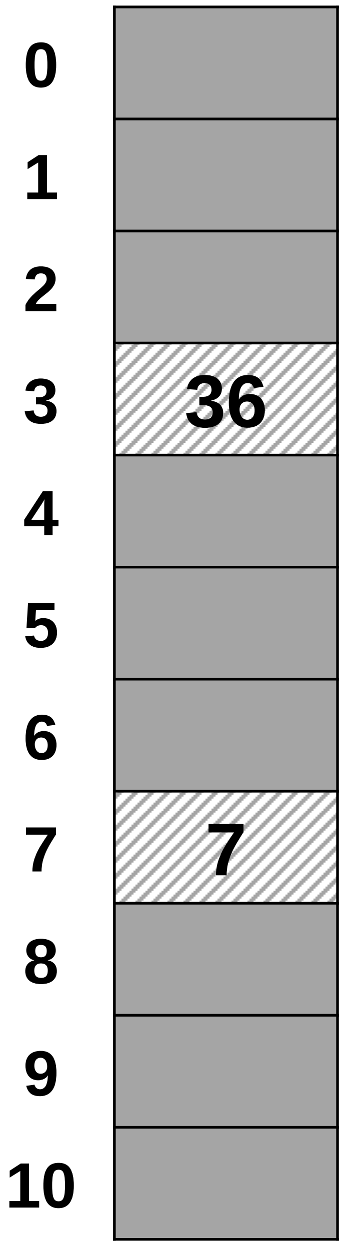


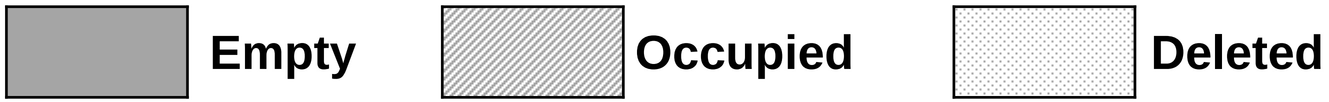


Linear Probing

Keys: 7, 36, 18, 62

Insert(18):
$\mathbf{h}_0$(18) = (18 mod 11) = 7





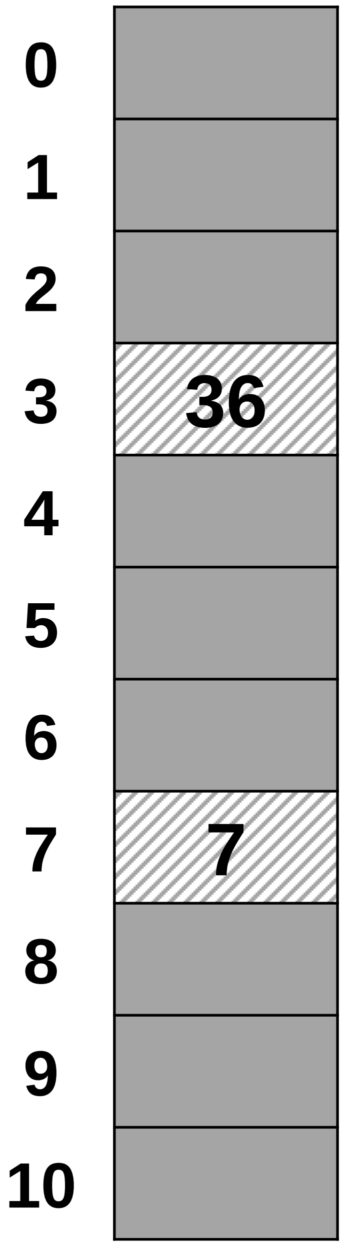
Linear Probing

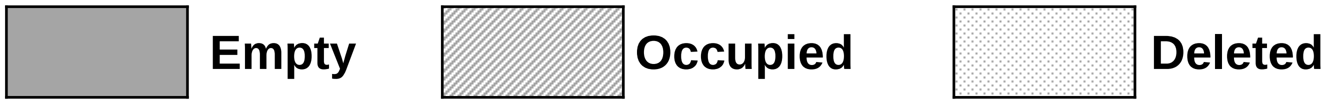
Keys: 7, 36, 18, 62

Insert(18):

$\mathbf{h}_0(18) = (18 \bmod 11) = 7$

$\mathbf{h}_1(18) = ((18+1) \bmod 11) = 8$

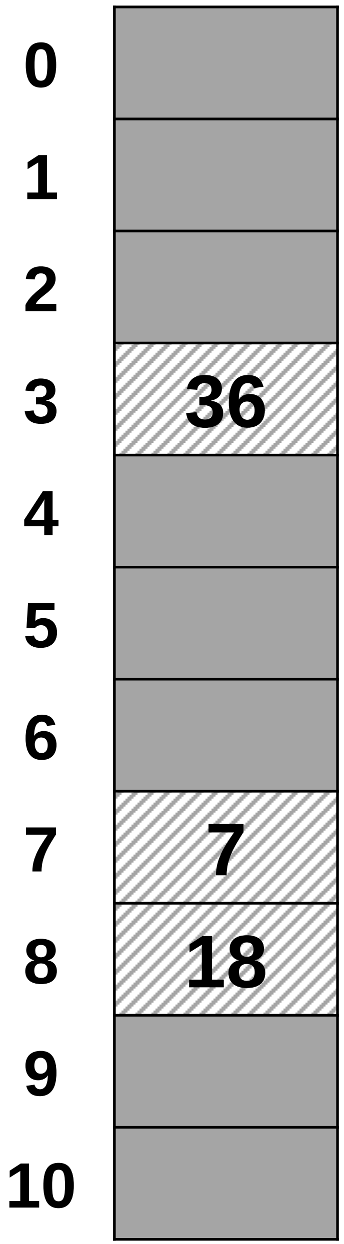


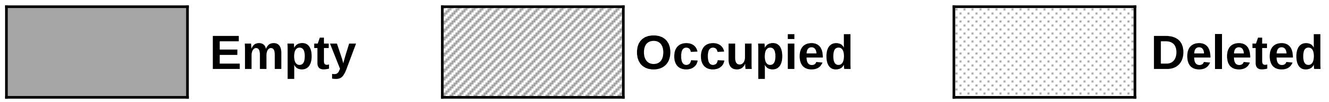


Linear Probing

Keys: 7, 36, 18, 62

Insert(18):
$\mathbf{h}_0$**(18) = (18 mod 11) = 7**
$\mathbf{h}_1$**(18) = ((18+1) mod 11) = 8**

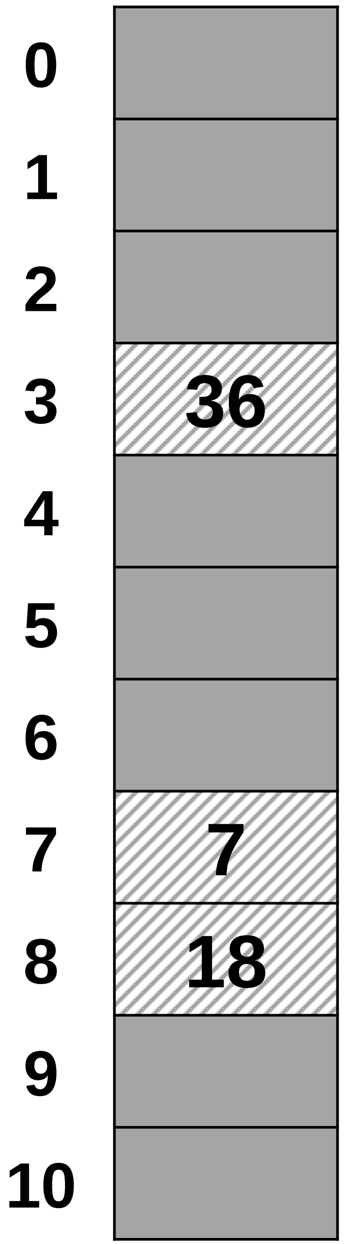


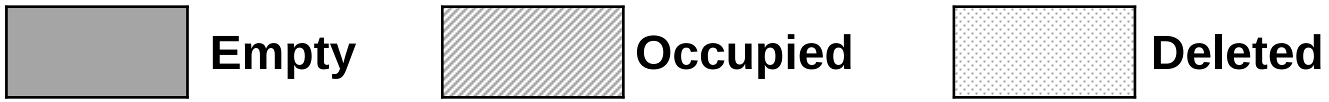


Linear Probing

Keys: 7, 36, 18, 62

Insert(62):
$\mathbf{h}_0$**(62) = (62 mod 11) = 7**





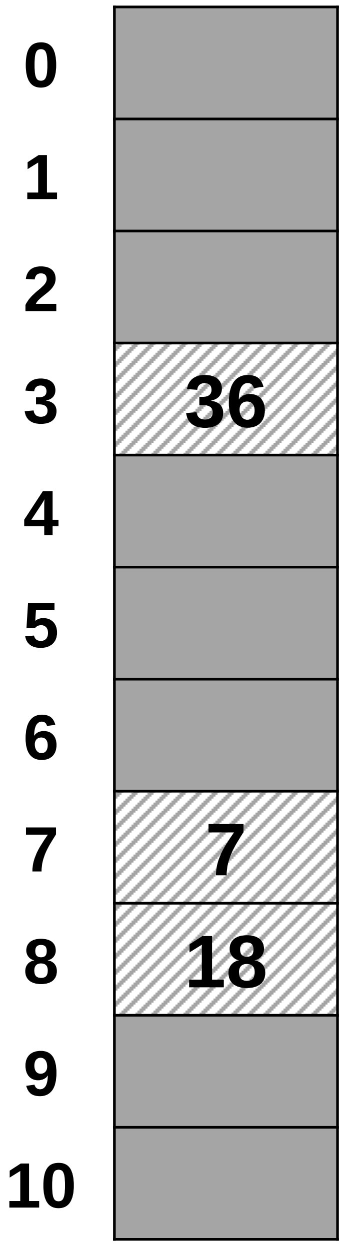
Linear Probing

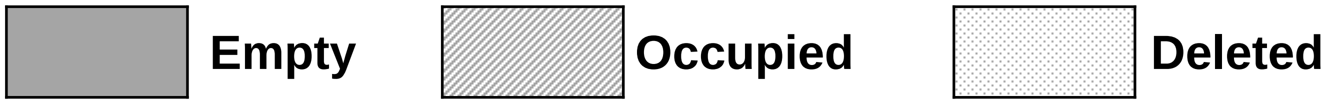
Keys: 7, 36, 18, 62

Insert(62):

$h_0(62) = (62 \bmod 11) = 7$

$h_1(62) = ((62+1) \bmod 11) = 8$

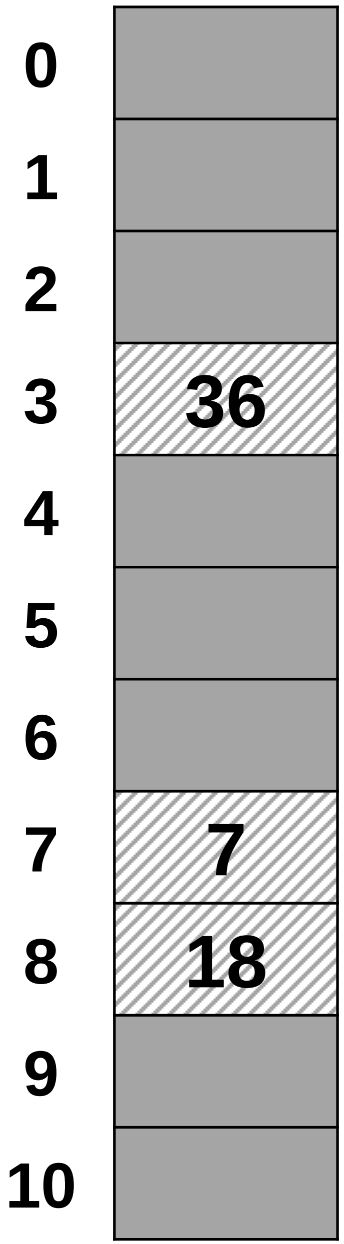


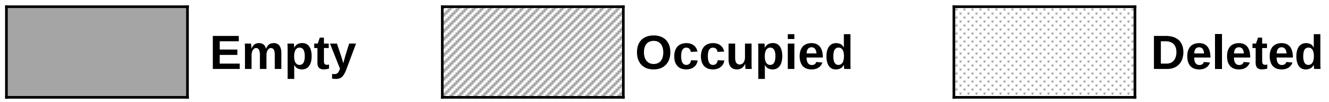


Linear Probing

Keys: 7, 36, 18, 62

Insert(62):
$\mathbf{h}_0$**(62) = (62 mod 11) = 7**
$\mathbf{h}_1$**(62) = ((62+1) mod 11) = 8**
$\mathbf{h}_2$**(62) = ((62+2) mod 11) = 9**





Linear Probing

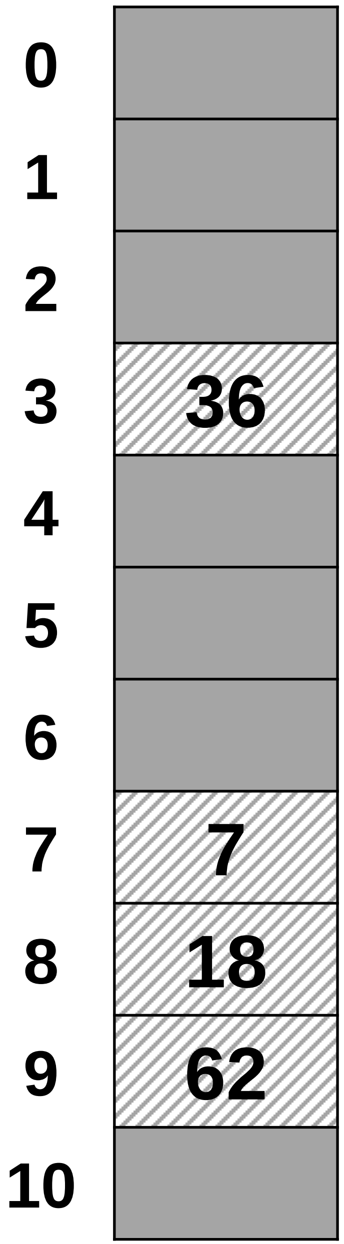
Keys: 7, 36, 18, 62

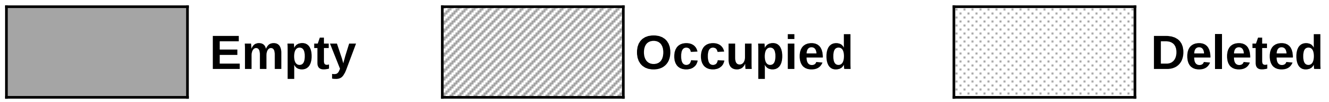
Insert(62):

$h_0(62) = (62 \bmod 11) = 7$

$h_1(62) = ((62+1) \bmod 11) = 8$

$h_2(62) = ((62+2) \bmod 11) = 9$

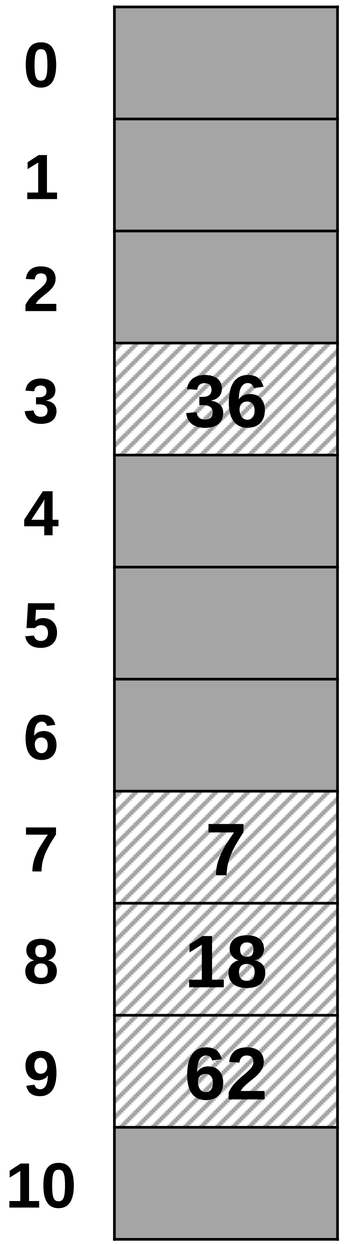


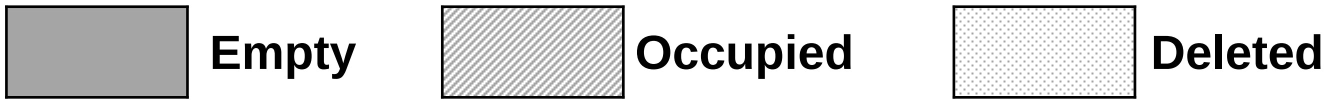


Linear Probing

Keys: 7, 36, 18, 62

Search(18):
$h_0(18) = (18 \bmod 11) = 7$
$h_1(18) = ((18+1) \bmod 11) = 8$

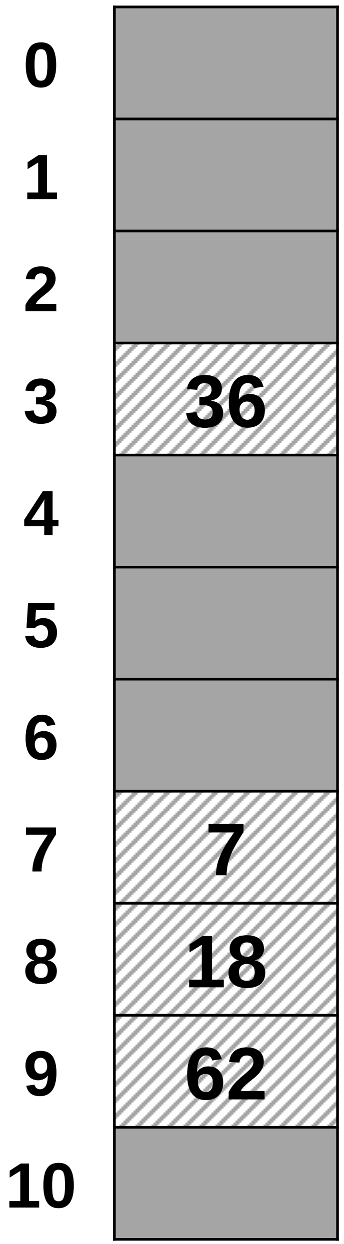


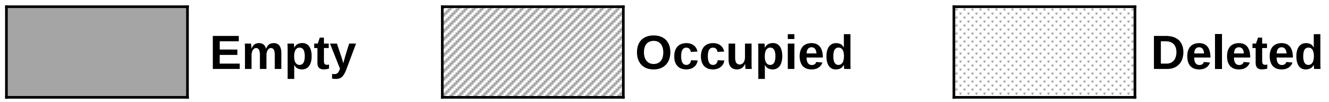


Linear Probing

Keys: 7, 36, 18, 62

Delete(18):

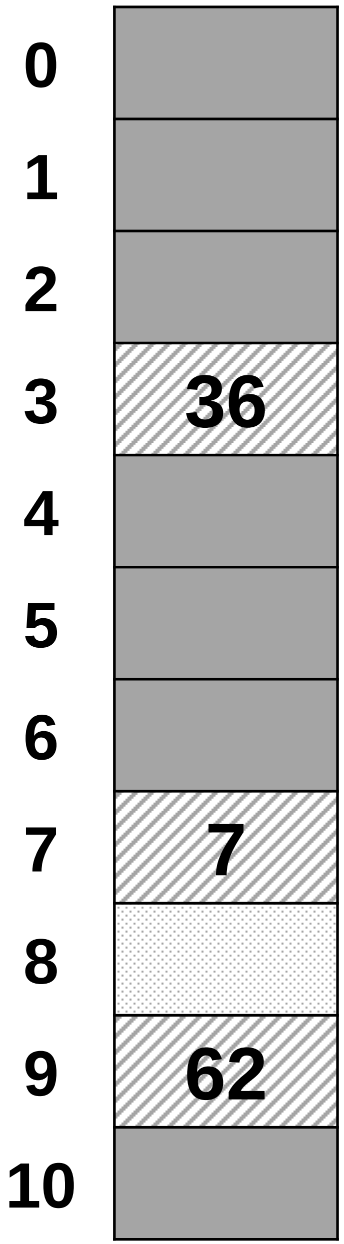


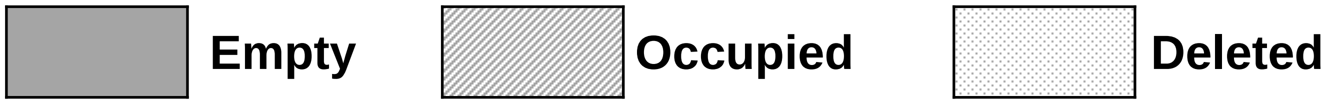


Linear Probing

Keys: 7, 36, 18, 62

Delete(18):





Open Addressing

A hash collision is resolved by probing

1) **Linear Probing**
2) **Quadratic Probing**
3) **Double Hashing**

Quadratic Probing

Quadratic Probing

$$\mathbf{h_i}(X) = (\text{Hash}(X) + \mathbf{i}^2) \,\%\, \text{HashTableSize}$$

If $\mathbf{h}_0$(X) = (Hash(X) + 0) % HashTableSize is full, we try for $\mathbf{h}_1$

If $\mathbf{h}_1$(X) = (Hash(X) + 1) % HashTableSize is full, we try for $\mathbf{h}_2$

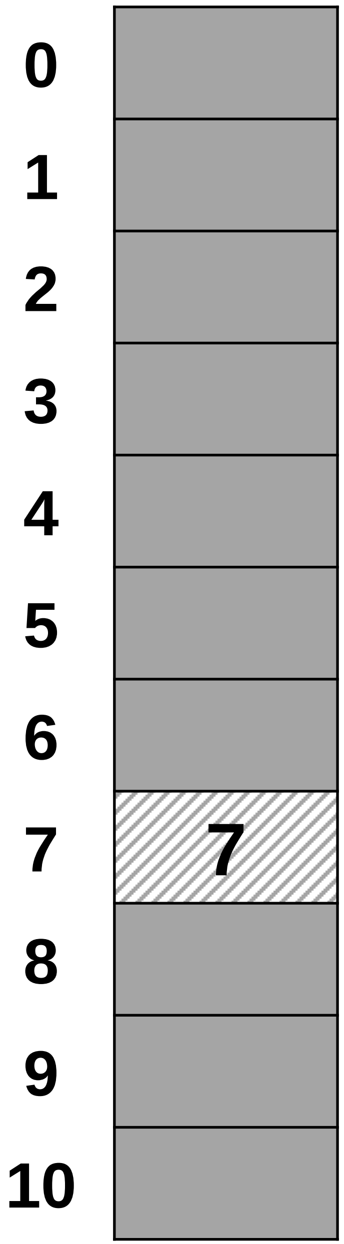
If $\mathbf{h}_1$(X) = (Hash(X) + 4) % HashTableSize is full, we try for $\mathbf{h}_3$

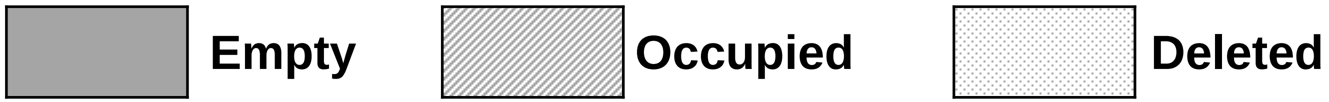
And so on ..

Quadratic Probing

Keys: 7, 36, 18, 62

Insert(7)

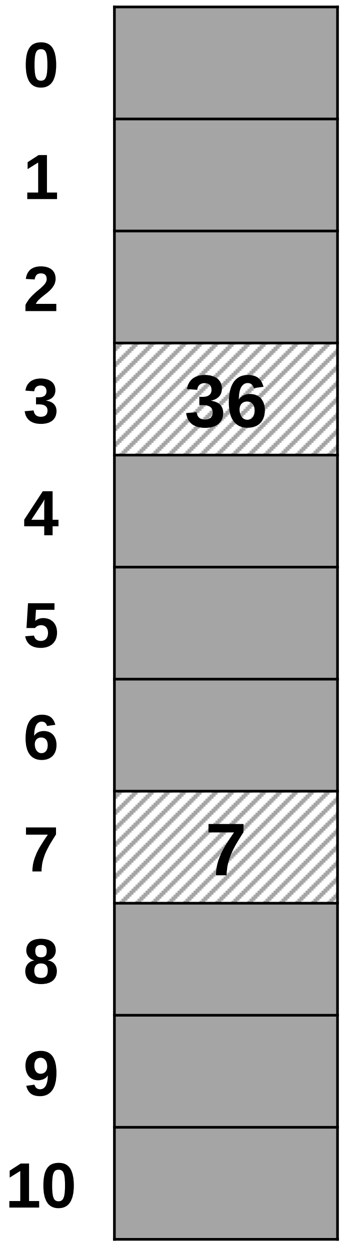


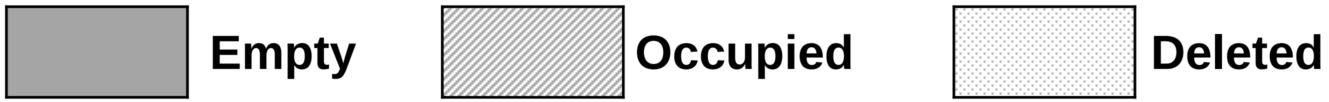


Quadratic Probing

Keys: 7, 36, 18, 62

Insert(36)

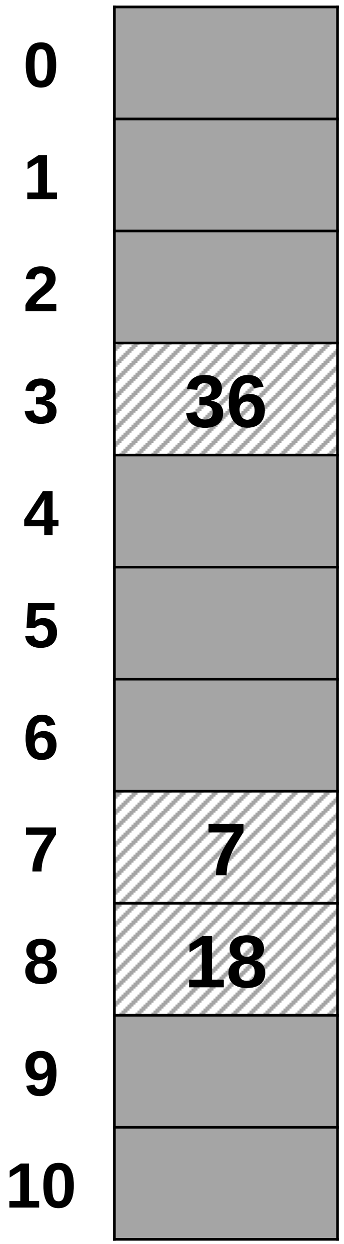


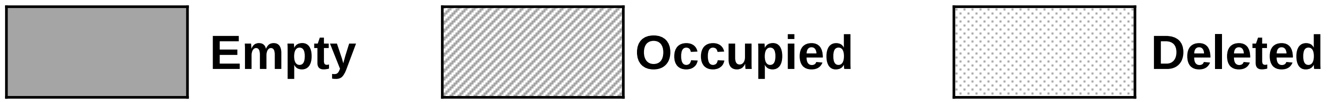


Quadratic Probing

Keys: 7, 36, 18, 62

Insert(18)

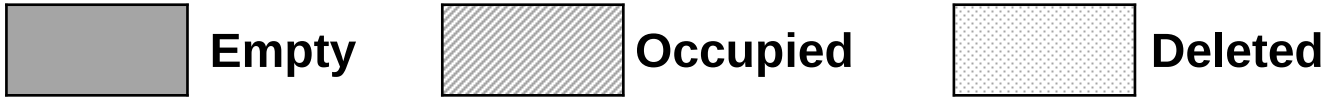


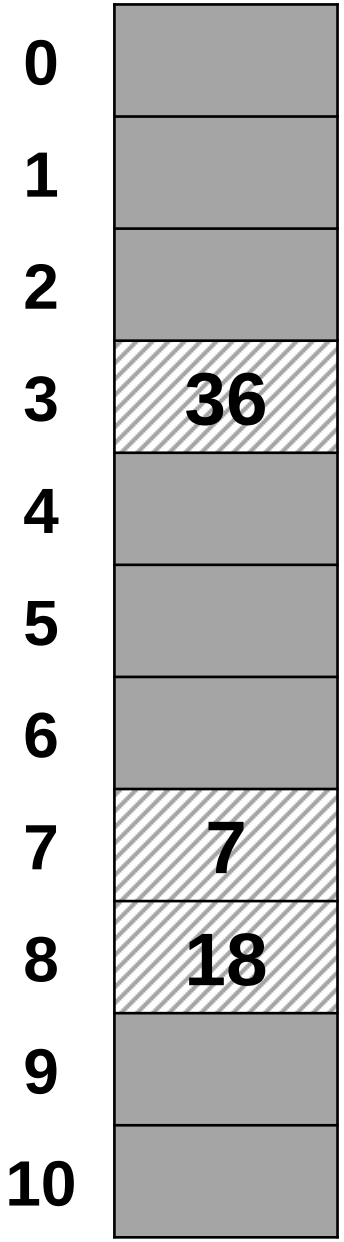


Quadratic Probing

Keys: 7, 36, 18, 62

Insert(62):
$\mathbf{h}_2$**(62) = ((62+4) mod 11) = 0**





Quadratic Probing

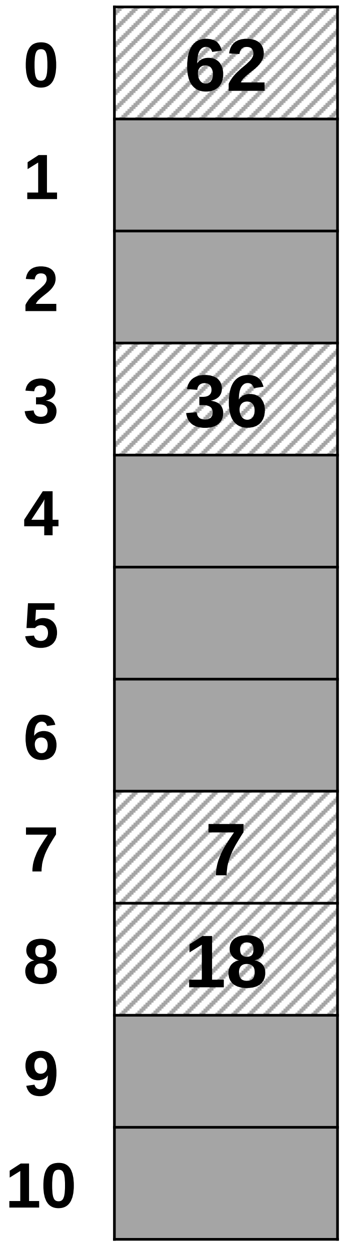
Keys: 7, 36, 18, 62

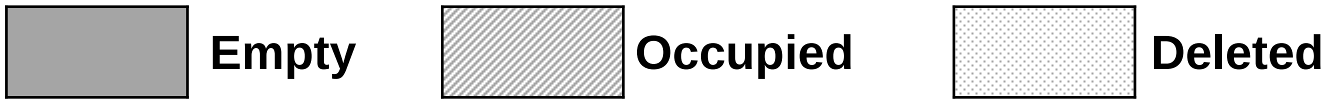
Insert(62):

$h_0(62) = (62 \bmod 11) = 7$

$h_1(62) = ((62+1) \bmod 11) = 8$

$h_2(62) = ((62+4) \bmod 11) = 0$





Open Addressing

A hash collision is resolved by probing

1) **Linear Probing**
2) **Quadratic Probing**
3) **Double Hashing**

Double Hashing

Double Hashing: use another hash function hash2(x) and look for i*hash2(x) slot in i'th iteration.

$$\mathbf{h_i}(X) = (Hash(X) + i * Hash2(X)) \% HashTableSize$$

If $\mathbf{h}_0$(X) = (Hash(X) + 0) % HashTableSize is full, we try for $\mathbf{h}_1$

If $\mathbf{h}_1$(X) = (Hash(X) + 1* Hash2(X)) % HashTableSize is full, we try for $\mathbf{h}_2$

If $\mathbf{h}_1$(X) = (Hash(X) + 4* Hash2(X)) % HashTableSize is full, we try for $\mathbf{h}_3$

And so on ..

Open Addressing

Linear Probing

- Easy to implement
- Best Cache Performance
- Suffers from clustering

Open Addressing

Linear Probing

- Easy to implement
- Best Cache Performance
- Suffers from clustering

Quadratic Probing

- Average Cache Performance
- Suffers a lesser clustering than linear probing

Open Addressing

Linear Probing

- Easy to implement
- Best Cache Performance
- Suffers from clustering

Quadratic Probing

- Average Cache Performance
- Suffers a lesser clustering than linear probing

Double Hashing

- Poor Cache Performance
- No clustering
- Requires more computation time

Open Addressing

Complexity:

n = number of keys to be inserted in hash table
m = number of slots in hash table

Load factor α = n/m (<1)

Open Addressing

Complexity:

n = number of keys to be inserted in hash table
m = number of slots in hash table

Load factor $\alpha = n/m$ (<1)

Theorem. Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Open Addressing

Proof.

- At least one probe is always necessary. With probability n/m, the first probe hits an occupied slot, and a second probe is necessary.
- With probability (n–1)/(m–1), the second probe hits an occupied slot, and a third probe is necessary.
- With probability (n–2)/(m–2), the third probe hits an occupied slot, etc.
- Observe that $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for i = 1, 2, …, n.

Open Addressing

Proof (continued)

- **Therefore, the expected number of probes is**

$$1+\frac{n}{m}\left(1+\left(\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\left(1+\frac{1}{m-n+1}\right)\cdots\right)\right)\right)\right)$$

$$<1+\alpha\left(1+\alpha\left(1+\alpha\left(\cdots\left(1+\alpha\right)\cdots\right)\right)\right)$$

$$\leq 1+\alpha+\alpha^2+\alpha^3+\cdots$$

$$=\sum_{i=0}^{\infty}\alpha^i$$

$$=\frac{1}{1\text{-}\alpha}$$

Open Addressing

Complexity:

n = number of keys to be inserted in hash table
m = number of slots in hash table

Load factor α = n/m (<1)

Expected time to insert/search/delete < $1/(1-\alpha)$

So Search, Insert and Delete take O($1/(1-\alpha)$) time

Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms*”*,* September 29, 2004
- https://www.geeksforgeeks.org