# Hashing: Collision Resolution Schemes

- Collision Resolution Techniques
- Separate Chaining
- Separate Chaining with String Keys
- Separate Chaining versus Open-addressing
- The class hierarchy of Hash Tables
- Implementation of Separate Chaining
- Introduction to Collision Resolution using Open Addressing
- Linear Probing

# Collision Resolution Techniques

- There are two broad ways of collision resolution:

1. Separate Chaining: An array of linked list implementation.

2. Open Addressing: Array-based implementation.

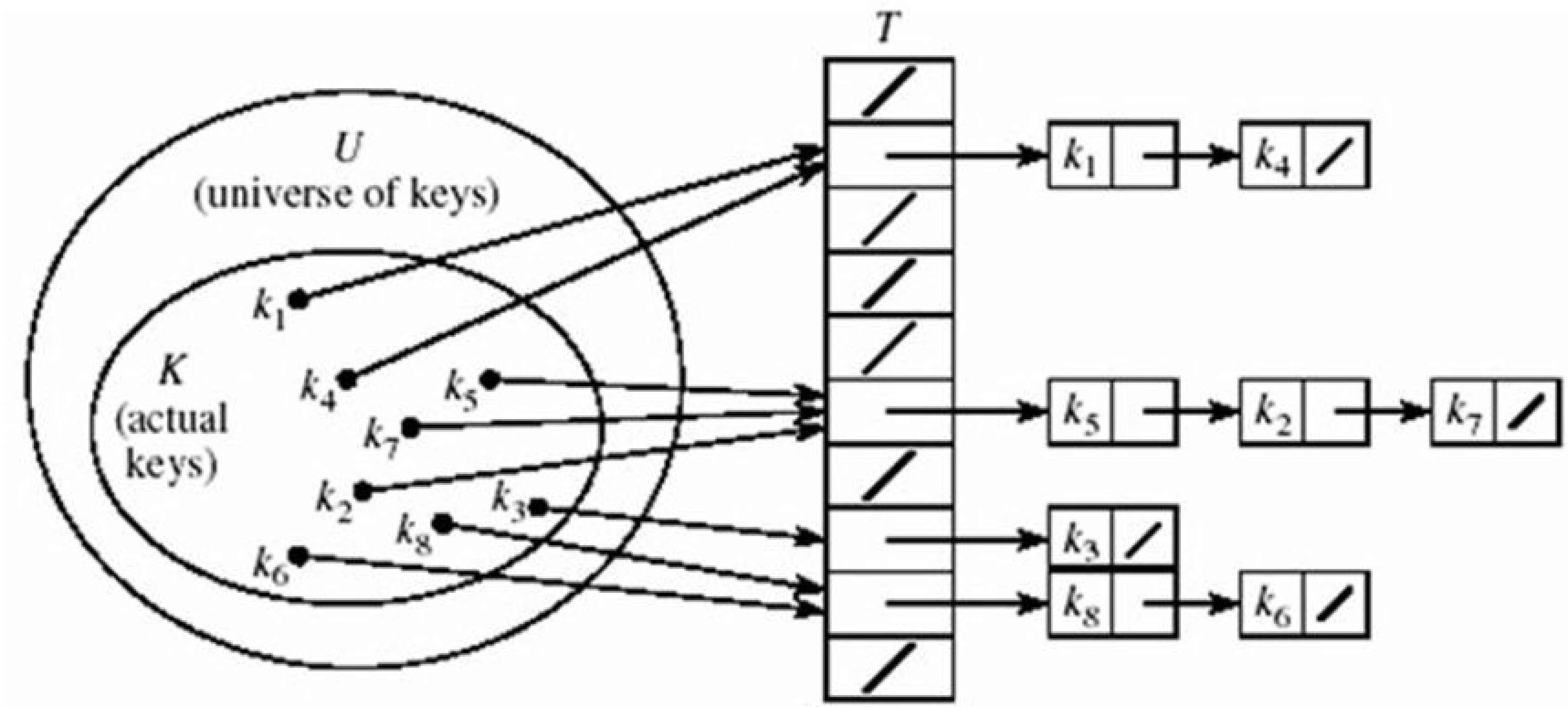
   (i) Linear probing (linear search)
   (ii) Quadratic probing (nonlinear search)
   (iii) Double hashing (uses two hash functions)

# Separate Chaining

- The hash table is implemented as an array of linked lists.

- Inserting an item, **`r`**, that hashes at index **`i`** is simply insertion into the linked list at position **`i`**.

- Synonyms are chained in the same linked list.



# Separate Chaining (cont'd)

- Retrieval of an item, `r`, with hash address, `i`, is simply retrieval from the linked list at position `i`.
- Deletion of an item, `r`, with hash address, `i`, is simply deleting `r` from the linked list at position `i`.
- **Example:** Load the keys **23, 13, 21, 14, 7, 8, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function: **h(key) = key % 7**

$h(23) = 23 \% 7 = 2$

$h(13) = 13 \% 7 = 6$

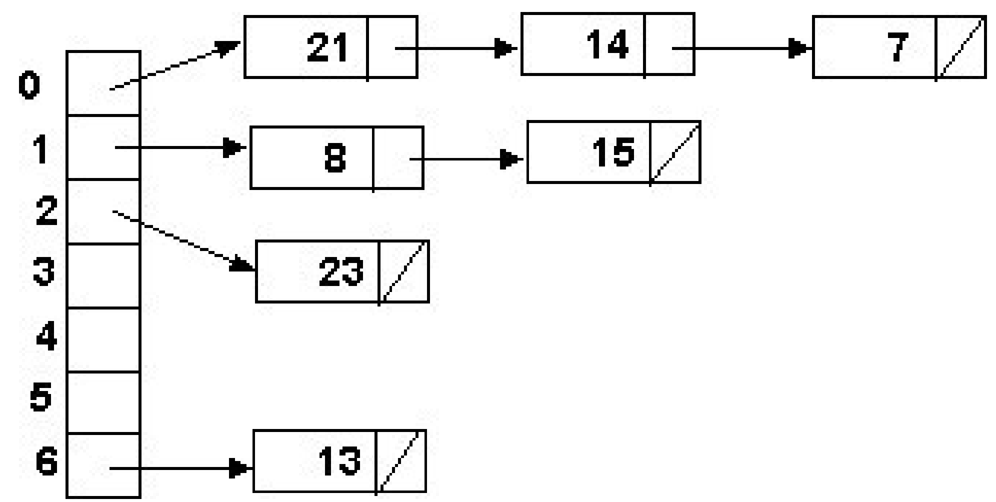
$h(21) = 21 \% 7 = 0$

$h(14) = 14 \% 7 = 0$ collision

$h(7) = 7 \% 7 = 0$ collision

$h(8) = 8 \% 7 = 1$

$h(15) = 15 \% 7 = 1$ collision



# Separate Chaining with String Keys

- Recall that search keys can be numbers, strings or some other object.
- A hash function for a string s = c0c1c2…cn-1 can be defined as:

  $$hash = (c_0 + c_1 + c_2 + \ldots + c_{n-1}) \% tableSize$$

  this can be implemented as:

```
public static int hash(String key, int tableSize){
   int hashValue = 0;
   for (int i = 0; i < key.length(); i++){
       hashValue += key.charAt(i);
   }
   return hashValue % tableSize;
}
```

- Example: The following class describes commodity items:

```
class CommodityItem {
    String name;      // commodity name
    int quantity;     // commodity quantity needed
    double price;     // commodity price
}
```

# Separate Chaining with String Keys (cont'd)

- Use the hash function **hash** to load the following commodity items into a hash table of size **13** using separate chaining:

```
onion        1      10.0
tomato       1      8.50
cabbage      3      3.50
carrot       1      5.50
okra         1      6.50
mellon       2      10.0
potato       2      7.50
Banana       3       4.00
olive        2      15.0
salt         2      2.50
cucumber     3      4.50
mushroom     3      5.50
orange       2      3.00
```

- Solution:

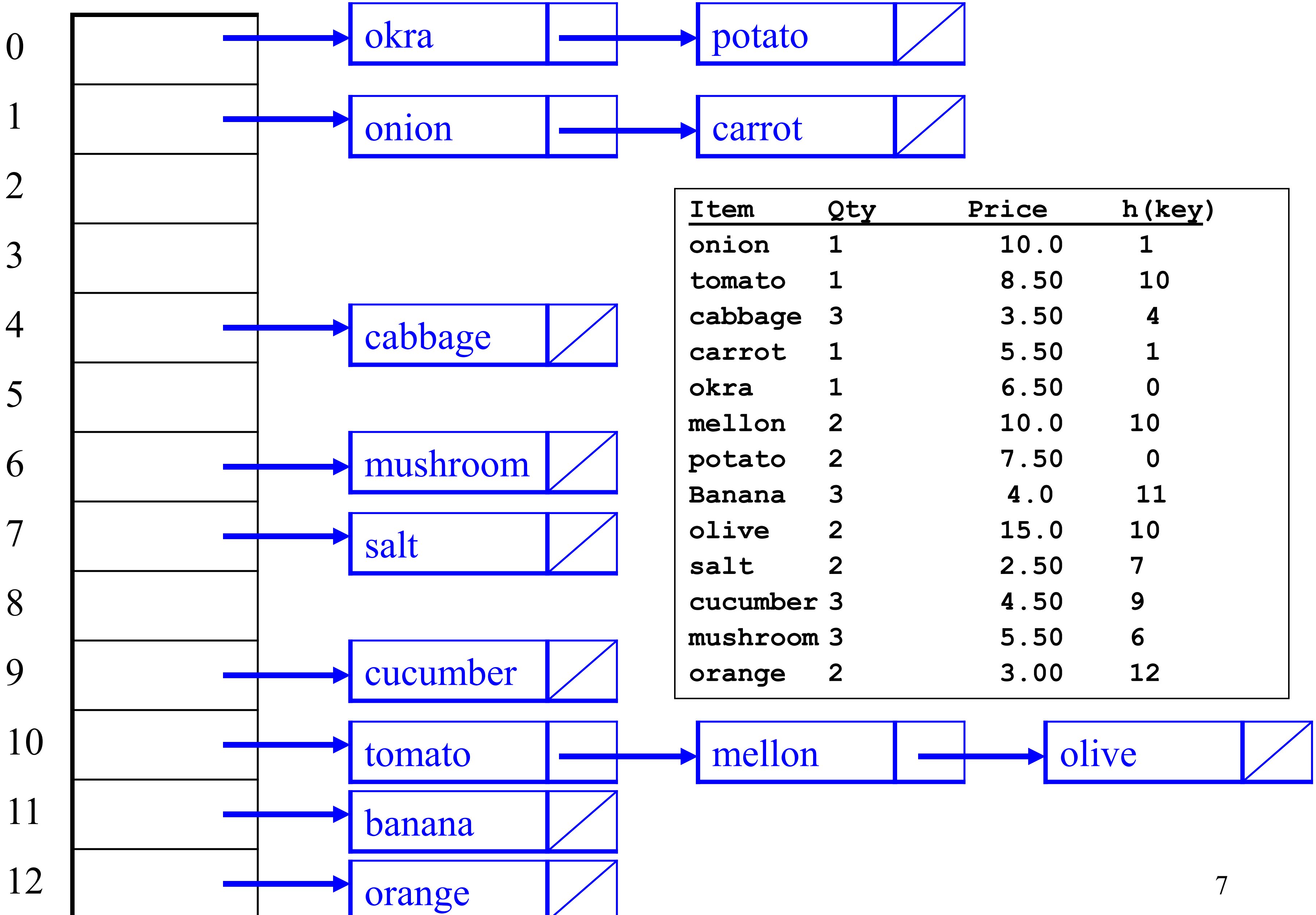
| character | a | b | c | e | g | h | i | k | l | m | n | o | p | r | s | t | u | v |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ASCII code | 97 | 98 | 99 | 101 | 103 | 104 | 105 | 107 | 108 | 109 | 110 | 111 | 112 | 114 | 115 | 116 | 117 | 118 |

hash(onion) = (111 + 110 + 105 + 111 + 110) % 13 = 547 % 13 = 1

hash(salt) = (115 + 97 + 108 + 116) % 13 = 436 % 13 = 7

hash(orange) = (111 + 114 + 97 + 110 + 103 + 101)%13 = 636 %13 = 12

# Separate Chaining with String Keys (cont'd)




| Item | Qty | Price | h(key) |
|---|---|---|---|
| onion | 1 | 10.0 | 1 |
| tomato | 1 | 8.50 | 10 |
| cabbage | 3 | 3.50 | 4 |
| carrot | 1 | 5.50 | 1 |
| okra | 1 | 6.50 | 0 |
| mellon | 2 | 10.0 | 10 |
| potato | 2 | 7.50 | 0 |
| Banana | 3 | 4.0 | 11 |
| olive | 2 | 15.0 | 10 |
| salt | 2 | 2.50 | 7 |
| cucumber | 3 | 4.50 | 9 |
| mushroom | 3 | 5.50 | 6 |
| orange | 2 | 3.00 | 12 |

# Separate Chaining with String Keys (cont'd)

- Alternative hash functions for a string

  $\mathbf{s = c_0c_1c_2 \ldots c_{n-1}}$

exist, some are:

- hash = $(c_0 + 27 * c_1 + 729 * c_2)$ % tableSize
- hash = $(c_0 + c_{n-1} + \text{s.length()})$ % tableSize
- hash = $[\sum_{k=0}^{s.length()-1} 26 * k + \text{s.charAt(k)} - \text{' '}]\%\text{tableSize}$

# Separate Chaining versus Open-addressing

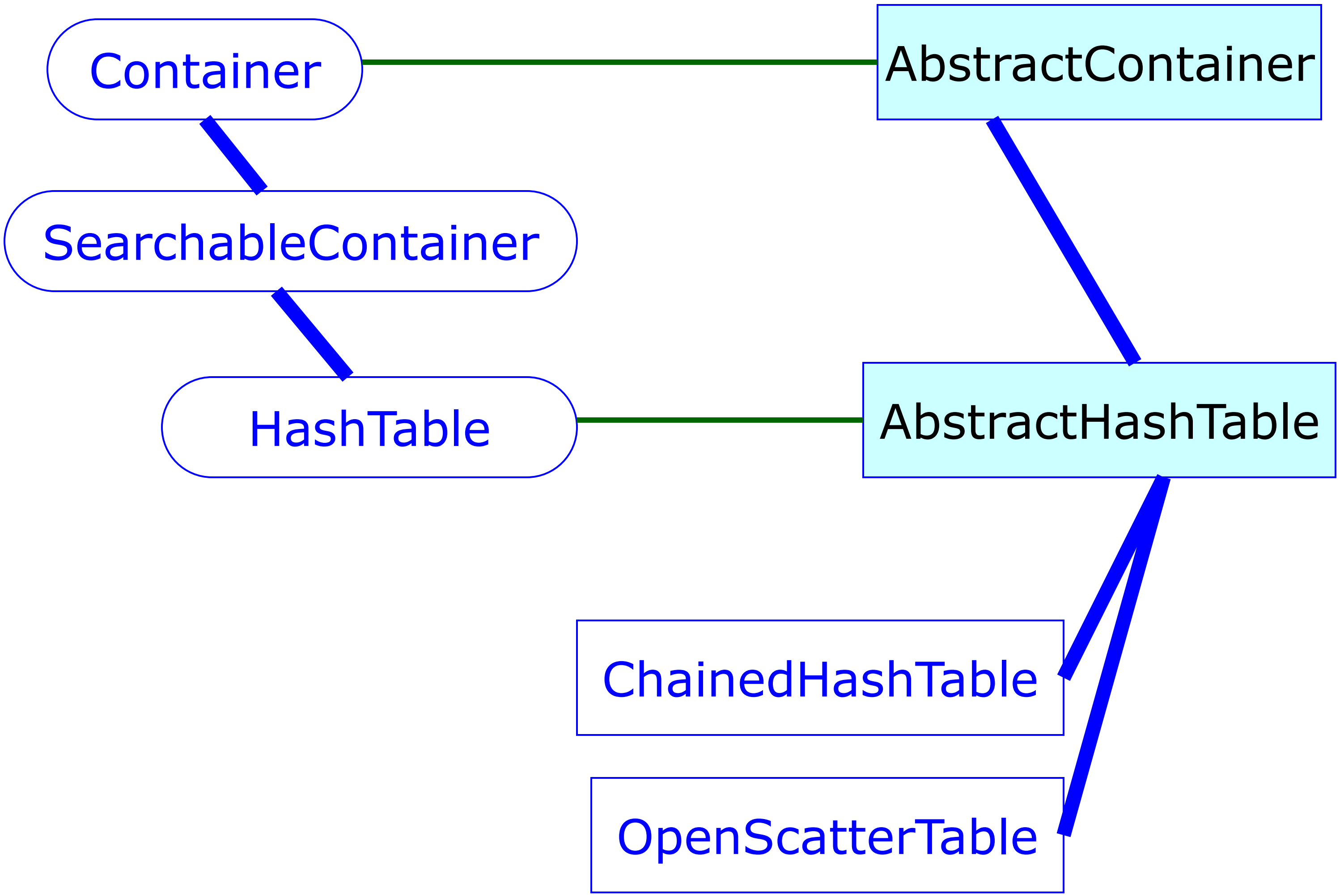
**Separate Chaining has several advantages over open addressing:**

- Collision resolution is simple and efficient.
- The hash table can hold more elements without the large performance deterioration of open addressing (The load factor can be 1 or greater)
- The performance of chaining declines much more slowly than open addressing.
- Deletion is easy - no special flag values are necessary.
- Table size need not be a prime number.
- The keys of the objects to be hashed need not be unique.

**Disadvantages of Separate Chaining:**

- It requires the implementation of a separate data structure for chains, and code to manage it.
- The main cost of chaining is the extra space required for the linked lists.
- For some languages, creating new nodes (for linked lists) is expensive and slows down the system.

# Implementing Hash Tables: The Hierarchy Tree



# Implementation of Separate Chaining

```
public class ChainedHashTable extends AbstractHashTable {
   protected MyLinkedList [ ] array;
   public ChainedHashTable(int size) {
      array = new MyLinkedList[size];
      for(int j = 0; j < size; j++)
         array[j] = new MyLinkedList( );
   }
   public void insert(Object key) {
      array[h(key)].append(key); count++;
   }
   public void withdraw(Object key) {
      array[h(key)].extract(key); count--;
   }
   public Object find(Object key){
      int index = h(key);
      MyLinkedList.Element e = array[index].getHead( );
      while(e != null){
         if(key.equals(e.getData()) return e.getData();
         e = e.getNext();
      }
      return null;
   }
}
```

# Introduction to Open Addressing

- All items are stored in the hash table itself.
- In addition to the cell data (if any), each cell keeps one of the three states: EMPTY, OCCUPIED, DELETED.
- While inserting, if a collision occurs, alternative cells are tried until an empty cell is found.

- **Deletion**: (lazy deletion): When a key is deleted the slot is marked as DELETED rather than EMPTY otherwise subsequent searches that hash at the deleted cell will fail.

- **Probe sequence**: A probe sequence is the sequence of array indexes that is followed in searching for an empty cell during an insertion, or in searching for a key during find or delete operations.

- The most common probe sequences are of the form:

$$\mathbf{h_i(key) = [h(key) + c(i)] \ \% \ n, \quad for\ i = 0, 1, \ldots, n\text{-}1.}$$

where **h** is a hash function and **n** is the size of the hash table

- The function **c(i)** is required to have the following two properties:

**Property 1:** $c(0) = 0$

**Property 2:** The set of values **{c(0) % n, c(1) % n, c(2) % n, . . . , c(n-1) % n}** must be a permutation of **{0, 1, 2,. . ., n – 1}**, that is, it must contain every integer between **0** and **n - 1** inclusive.

# Introduction to Open Addressing (cont'd)

- The function **c(i)** is used to resolve collisions.

- To insert item r, we examine array location $\mathbf{h_0(r) = h(r)}$. If there is a collision, array locations $\mathbf{h_1(r), h_2(r), ..., h_{n-1}(r)}$ are examined until an empty slot is found.

- Similarly, to find item **r**, we examine the same sequence of locations in the same order.

- **Note**: For a given hash function **h(key)**, the only difference in the open addressing collision resolution techniques (linear probing, quadratic probing and double hashing) is in the definition of the function **c(i)**.
- Common definitions of **c(i)** are:

| Collision resolution technique | c(i) |
|---|---|
| Linear probing | i |
| Quadratic probing | $\pm i^2$ |
| Double hashing | $i * h_p(key)$ |

where $\mathbf{h_p(key)}$ is another hash function.

# Introduction to Open Addressing (cont'd)

- **Advantages of Open addressing:**
  - All items are stored in the hash table itself. There is no need for another data structure.
  - Open addressing is more efficient storage-wise.

- **Disadvantages of Open Addressing:**
  - The keys of the objects to be hashed must be distinct.
  - Dependent on choosing a proper table size.
  - Requires the use of a three-state (Occupied, Empty, or Deleted) flag in each cell.

# Open Addressing Facts

- In general, primes give the best table sizes.
- With any open addressing method of collision resolution, as the table fills, there can be a severe degradation in the table performance.
- Load factors between 0.6 and 0.7 are common.
- Load factors > 0.7 are undesirable.
- The search time depends only on the load factor, *not* on the table size.
- We can use the desired load factor to determine appropriate table size:

$$\text{table size} = \text{smallest prime} \geq \frac{\text{number of items in table}}{\text{desired load factor}}$$

# Open Addressing: Linear Probing

- **c(i)** is a linear function in **i** of the form **c(i) = a*i**.
- Usually **c(i)** is chosen as:

  $c(i) = i$ **for i = 0, 1, . . . , tableSize – 1**

- The probe sequences are then given by:

  **$h_i$(key) = [h(key) + i] % tableSize for i = 0, 1, . . . , tableSize – 1**

- For **c(i) = a*i** to satisfy Property 2, **a** and **n** must be relatively prime.