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: \( h(key) = \text{key} \mod 7 \)
  
  \[
  \begin{align*}
  h(23) &= 23 \mod 7 = 2 \\
  h(13) &= 13 \mod 7 = 6 \\
  h(21) &= 21 \mod 7 = 0 \\
  h(14) &= 14 \mod 7 = 0 \quad \text{collision} \\
  h(7) &= 7 \mod 7 = 0 \quad \text{collision} \\
  h(8) &= 8 \mod 7 = 1 \\
  h(15) &= 15 \mod 7 = 1 \quad \text{collision}
  \end{align*}
  \]
Separate Chaining with String Keys

- Recall that search keys can be numbers, strings or some other object.
- A hash function for a string \( s = c_0c_1c_2\ldots c_{n-1} \) can be defined as:
  \[
  \text{hash} = (c_0 + c_1 + c_2 + \ldots + c_{n-1}) \ % \ \text{tableSize}
  \]
  this can be implemented as:

  ```java
  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:

  ```java
  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 $\text{hash}$ to load the following commodity items into a hash table of size 13 using separate chaining:

<table>
<thead>
<tr>
<th>commodity</th>
<th>quantity</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>onion</td>
<td>1</td>
<td>10.0</td>
</tr>
<tr>
<td>tomato</td>
<td>1</td>
<td>8.50</td>
</tr>
<tr>
<td>cabbage</td>
<td>3</td>
<td>3.50</td>
</tr>
<tr>
<td>carrot</td>
<td>1</td>
<td>5.50</td>
</tr>
<tr>
<td>okra</td>
<td>1</td>
<td>6.50</td>
</tr>
<tr>
<td>mellon</td>
<td>2</td>
<td>10.0</td>
</tr>
<tr>
<td>potato</td>
<td>2</td>
<td>7.50</td>
</tr>
<tr>
<td>Banana</td>
<td>3</td>
<td>4.00</td>
</tr>
<tr>
<td>olive</td>
<td>2</td>
<td>15.0</td>
</tr>
<tr>
<td>salt</td>
<td>2</td>
<td>2.50</td>
</tr>
<tr>
<td>cucumber</td>
<td>3</td>
<td>4.50</td>
</tr>
<tr>
<td>mushroom</td>
<td>3</td>
<td>5.50</td>
</tr>
<tr>
<td>orange</td>
<td>2</td>
<td>3.00</td>
</tr>
</tbody>
</table>

- Solution:

$$\text{hash}(\text{onion}) = (111 + 110 + 105 + 111 + 110) \mod 13 = 547 \mod 13 = 1$$
$$\text{hash}(\text{salt}) = (115 + 97 + 108 + 116) \mod 13 = 436 \mod 13 = 7$$
$$\text{hash}(\text{orange}) = (111 + 114 + 97 + 110 + 103 + 101) \mod 13 = 636 \mod 13 = 12$$
Separate Chaining with String Keys (cont’d)

<table>
<thead>
<tr>
<th>Item</th>
<th>Qty</th>
<th>Price</th>
<th>h(key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>onion</td>
<td>1</td>
<td>10.0</td>
<td>1</td>
</tr>
<tr>
<td>tomato</td>
<td>1</td>
<td>8.50</td>
<td>10</td>
</tr>
<tr>
<td>cabbage</td>
<td>3</td>
<td>3.50</td>
<td>4</td>
</tr>
<tr>
<td>carrot</td>
<td>1</td>
<td>5.50</td>
<td>1</td>
</tr>
<tr>
<td>okra</td>
<td>1</td>
<td>6.50</td>
<td>0</td>
</tr>
<tr>
<td>mellon</td>
<td>2</td>
<td>10.0</td>
<td>10</td>
</tr>
<tr>
<td>potato</td>
<td>2</td>
<td>7.50</td>
<td>0</td>
</tr>
<tr>
<td>banana</td>
<td>3</td>
<td>4.0</td>
<td>11</td>
</tr>
<tr>
<td>olive</td>
<td>2</td>
<td>15.0</td>
<td>10</td>
</tr>
<tr>
<td>salt</td>
<td>2</td>
<td>2.50</td>
<td>7</td>
</tr>
<tr>
<td>cucumber</td>
<td>3</td>
<td>4.50</td>
<td>9</td>
</tr>
<tr>
<td>mushroom</td>
<td>3</td>
<td>5.50</td>
<td>6</td>
</tr>
<tr>
<td>orange</td>
<td>2</td>
<td>3.00</td>
<td>12</td>
</tr>
</tbody>
</table>

Diagram shows items and their positions using separate chaining.
Separate Chaining with String Keys (cont’d)

- Alternative hash functions for a string
  
  \[ s = c_0c_1c_2\ldots c_{n-1} \]
  
  exist, some are:

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

Container

SearchableContainer

HashTable

AbstractContainer

AbstractHashTable

ChainedHashTable

OpenScatterTable
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:
  \[ h_i(key) = [h(key) + c(i)] \mod n, \quad \text{for } i = 0, 1, \ldots, n-1. \]
  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 \( \{c(0) \mod n, c(1) \mod n, c(2) \mod n, \ldots, c(n-1) \mod n\} \) must be a permutation of \( \{0, 1, 2, \ldots, n-1\} \), 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 \( h_0(r) = h(r) \). If there is a collision, array locations \( h_1(r), h_2(r), ..., h_{n-1}(r) \) 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:

<table>
<thead>
<tr>
<th>Collision resolution technique</th>
<th>( c(i) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear probing</td>
<td>( i )</td>
</tr>
<tr>
<td>Quadratic probing</td>
<td>( \pm i^2 )</td>
</tr>
<tr>
<td>Double hashing</td>
<td>( i \cdot h_p(key) )</td>
</tr>
</tbody>
</table>

where \( h_p(key) \) 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:
  
  $$c(i) = i \quad \text{for } i = 0, 1, \ldots, \text{tableSize} - 1$$

- The probe sequences are then given by:
  
  $$h_i(key) = [h(key) + i] \mod \text{tableSize} \quad \text{for } i = 0, 1, \ldots, \text{tableSize} - 1$$

- For $c(i) = a*i$ to satisfy Property 2, $a$ and $n$ must be relatively prime.
Linear Probing (cont’d)

Example: Perform the operations given below, in the given order, on an initially empty hash table of size 13 using linear probing with \( c(i) = i \) and the hash function: \( h(key) = key \% 13 \):

\[
\text{insert}(18), \text{insert}(26), \text{insert}(35), \text{insert}(9), \text{find}(15), \text{find}(48), \text{delete}(35), \text{delete}(40), \text{find}(9), \text{insert}(64), \text{insert}(47), \text{find}(35)
\]

- The required probe sequences are given by:

\[
h_i(key) = (h(key) + i) \% 13 \quad i = 0, 1, 2, \ldots, 12
\]
### Linear Probing (cont’d)

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>PROBE SEQUENCE</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(18)</td>
<td>( h_0(18) = (18 % 13) % 13 = 5 )</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>insert(26)</td>
<td>( h_0(26) = (26 % 13) % 13 = 0 )</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>insert(35)</td>
<td>( h_0(35) = (35 % 13) % 13 = 9 )</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>insert(9)</td>
<td>( h_0(9) = (9 % 13) % 13 = 9 )</td>
<td>COLLISION</td>
</tr>
<tr>
<td></td>
<td>( h_1(9) = (9+1) % 13 = 10 )</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>find(15)</td>
<td>( h_0(15) = (15 % 13) % 13 = 2 )</td>
<td>FAIL because location 2 has <strong>Empty</strong> status</td>
</tr>
<tr>
<td>find(48)</td>
<td>( h_0(48) = (48 % 13) % 13 = 9 )</td>
<td>COLLISION</td>
</tr>
<tr>
<td></td>
<td>( h_1(48) = (9 + 1) % 13 = 10 )</td>
<td>COLLISION</td>
</tr>
<tr>
<td></td>
<td>( h_2(48) = (9 + 2) % 13 = 11 )</td>
<td>FAIL because location 11 has <strong>Empty</strong> status</td>
</tr>
<tr>
<td>withdraw(35)</td>
<td>( h_0(35) = (35 % 13) % 13 = 9 )</td>
<td>SUCCESS because location 9 contains 35 and the status is <strong>Occupied</strong>. The status is changed to <strong>Deleted</strong>; but the key 35 is not removed.</td>
</tr>
<tr>
<td>find(9)</td>
<td>( h_0(9) = (9 % 13) % 13 = 9 )</td>
<td>The search continues, location 9 does not contain 9; but its status is <strong>Deleted</strong></td>
</tr>
<tr>
<td></td>
<td>( h_1(9) = (9+1) % 13 = 10 )</td>
<td></td>
</tr>
<tr>
<td>insert(64)</td>
<td>( h_0(64) = (64 % 13) % 13 = 12 )</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>insert(47)</td>
<td>( h_0(47) = (47 % 13) % 13 = 8 )</td>
<td>SUCCESS</td>
</tr>
<tr>
<td>find(35)</td>
<td>( h_0(35) = (35 % 13) % 13 = 9 )</td>
<td>FAIL because location 9 contains 35 but its status is <strong>Deleted</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Status</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>O</td>
<td>26</td>
</tr>
<tr>
<td>1</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>O</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>O</td>
<td>47</td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>35</td>
</tr>
<tr>
<td>10</td>
<td>O</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>O</td>
<td>64</td>
</tr>
</tbody>
</table>
Disadvantage of Linear Probing: Primary Clustering

- Linear probing is subject to a primary clustering phenomenon.
- Elements tend to cluster around table locations that they originally hash to.
- Primary clusters can combine to form larger clusters. This leads to long probe sequences and hence deterioration in hash table efficiency.

**Example of a primary cluster:** Insert keys: 18, 41, 22, 44, 59, 32, 31, 73, in this order, in an originally empty hash table of size 13, using the hash function $h(\text{key}) = \text{key} \mod 13$ and $c(i) = i$:

- $h(18) = 5$
- $h(41) = 2$
- $h(22) = 9$
- $h(44) = 5+1$
- $h(59) = 7$
- $h(32) = 6+1+1$
- $h(31) = 5+1+1+1+1+1$
- $h(73) = 8+1+1+1$
Exercises

1. Given that,
   \[ c(i) = a \times i, \]
   for \( c(i) \) in linear probing, we discussed that this equation satisfies Property 2 only when \( a \) and \( n \) are relatively prime. Explain what the requirement of being relatively prime means in simple plain language.

2. Consider the general probe sequence,
   \[ h_i(r) = (h(r) + c(i)) \mod n. \]
   Are we sure that if \( c(i) \) satisfies Property 2, then \( h_i(r) \) will cover all \( n \) hash table locations, \( 0, 1, \ldots, n-1 \)? Explain.

3. Suppose you are given \( k \) records to be loaded into a hash table of size \( n \), with \( k < n \) using linear probing. Does the order in which these records are loaded matter for retrieval and insertion? Explain.

4. A prime number is always the best choice of a hash table size. Is this statement true or false? Justify your answer either way.