7 Hashing: chaining
Possible ways of treating collisions

Treatment of collisions:

- collisions are treated differently in different methods.

- a data set with key $s$ is called a **colliding element** if bucket $B_{h(s)}$ is already taken by another data set.

- what can we do with colliding elements?
  1. **chaining**: implement the buckets as linked lists. Colliding elements are stored in these lists.
  2. **open addressing**: colliding elements are stored in other vacant buckets. During storage and lookup, these are found through so-called probing.
Chaining (1)

- The hash table is an array (length $m$) of lists. Each bucket is implemented by a list

```java
class hashTable {
    List[] ht; // an array of lists
    hashTable (int m) { // Constructor
        ht = new List[m];
        for (int i = 0; i < m; i++)
            ht[i] = new List(); // Construct a list
    }
}
...
```

- Two different ways of using lists:
  1. direct chaining:
     hash table only contains list headers; the data sets are stored in the lists
  2. separate chaining:
     hash table contains at most one data set in each bucket as well as a list header. Colliding elements are stored in the list
Haching by chaining

Keys are stored in overflow lists

\[ h(k) = k \mod 7 \]

This type of chaining is also known as direct chaining.
Direct chaining

**Search** key $k$
- compute $h(k)$ and overflow list $T[h(k)]$
- look for $k$ in the overflow list

**Insert** a key $k$
- search $k$ (fails)
- insert $k$ in the overflow list

**Delete** a key $k$
- search $k$ (successfully)
- remove $k$ from the overflow list

→ only list operations
class TableEntry {
    private Object key, value;
}

abstract class HashTable {
    private TableEntry[] tableEntry;
    private int capacity;
    // Constructor
    HashTable (int capacity) {
        this.capacity = capacity;
        tableEntry = new TableEntry [capacity];
        for (int i = 0; i <= capacity-1; i++)
            tableEntry[i] = null;
    }
    // the hash function
    protected abstract int h (Object key);
    // insert element with given key and value (if not there already)
    public abstract void insert (Object key Object value);
    // delete element with given key (if there)
    public abstract void delete (Object key);
    // lookup element with given key
    public abstract Object search (Object key);
} // class hashTable
Implementation in Java

class ChainedTableEntry extends TableEntry {
    // Constructor
    ChainedTableEntry(Object key, Object value) {
        super(key, value);
        this.next = null;
    }
    private ChainedTableEntry next;
}
class ChainedHashTable extends HashTable {
    // the hash function
    public int h(Object key) {
        return key.hashCode() % capacity;
    }
    // lookup key in the hash table
    public Object search (Object key) {
        ChainedTableEntry p;
        p = (ChainedTableEntry) tableEntry[h(key)];
        // Go through the list until end reached or key found
        while (p != null && !p.key.equals(key)) {
            p = p.next;
        }
        // Return result
        if (p != null)
            return p.value;
        else return null;
    }
}
/* Insert an element with given key and value (if not there) */
public void insert (Object key, Object value) {
    ChainedTableEntry entry = new ChainedTableEntry(key, value);
    // Get table entry for key
    int k = h (key);
    ChainedTableEntry p;
    p = (ChainedTableEntry) tableEntry [k];
    if (p == null){
        tableEntry[k] = entry;
        return ;
    }
    // Lookup key
    while (!p.key.equals(key) && p.next != null) {
        p = p.next;
    }
    // Insert the element (if not there)
    if (!p.key.equals(key))
        p.next = entry;
}
Implementation in Java

// Delete element with given key (if there)
public void delete (Object key) {
    int k = h (key);
    ChainedTableEntry p;
    p = (ChainedTableEntry) TableEntry [k];
    TableEntry[k] = recDelete(p, key);
}

// Delete element with key recursively (if there)
public ChainedTableEntry recDelete (ChainedTableEntry p, Object key) {
    /* recDelete returns a pointer to the start of the list that p points to,
       in which key was deleted */
    if (p == null)
        return null;
    if (p.key.equals(key))
        return p.getNext();
    // otherwise:
    p.next = recDelete(p.next, key);
    return p;
}

public void printTable () {...}
} // class ChainedHashTable
Test program

```java
public class ChainedHashingTest {
    public static void main(String args[]){
        Integer[] t= new Integer[args.length];
        for (int i = 0; i < args.length; i++)
            t[i] = Integer.valueOf(args[i]);
        ChainedHashTable h = new ChainedHashTable(7);
        for (int i = 0; i <= t.length - 1; i++)
            h.insert(t[i], null);
        h.printTable();
        h.delete(t[0]); h.delete(t[1]);
        h.delete(t[6]); h.printTable();
    }
}
```

**Call:**
```
java ChainedHashingTest 12 53 5 15 2 19 43
```

**Output:**
```
0: -|
1: 15 -> 43 -|
2: 2 -|
3: -|
4: 53 -|
5: 12 -> 5 -> 19 -|
6: -|
0: -|
1: 15 -|
2: 2 -|
3: -|
4: -|
5: 5 -> 19 -|
6: -|
```
Analysis of direct chaining

Uniform hashing assumption:

- all hash addresses are chosen with the same probability, i.e.:
  \[ Pr(h(k_i) = j) = \frac{1}{m} \]

- independent from operation to operation

Average chain length for \( n \) entries:

\[ \frac{n}{m} = \alpha \]

Definition:

- \( C'_n \) = expected number of entries inspected during a failed search
- \( C_n \) = expected number of entries inspected during a successful search

Analysis:

\[
\begin{align*}
C'_n & = \alpha \\
C_n & \approx 1 + \frac{\alpha}{2}
\end{align*}
\]
Chaining

Advantages:
+ $C_n$ and $C'_n$ are small
+ $\alpha > 1$ possible
+ suitable for secondary memory

Efficiency of lookup

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$C_n$ (successful)</th>
<th>$C'_n$ (unsuccessful)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>1.250</td>
<td>0.50</td>
</tr>
<tr>
<td>0.90</td>
<td>1.450</td>
<td>0.90</td>
</tr>
<tr>
<td>0.95</td>
<td>1.457</td>
<td>0.95</td>
</tr>
<tr>
<td>1.00</td>
<td>1.500</td>
<td>1.00</td>
</tr>
<tr>
<td>2.00</td>
<td>2.000</td>
<td>2.00</td>
</tr>
<tr>
<td>3.00</td>
<td>2.500</td>
<td>3.00</td>
</tr>
</tbody>
</table>

Disadvantages:
- Additional space for pointers
- Colliding elements are outside the hash table
Summary

Analysis of hashing with chaining:

- **worst case:**
  \( h(s) \) always yields the same value, all data sets are in a list.
  Behavior as in linear lists.

- **average case:**
  - Successful lookup & delete:
    \[ \text{complexity (in inspections)} \approx 1 + 0.5 \times \text{load factor} \]
  - Failed lookup & insert:
    \[ \text{complexity} \approx \text{load factor} \]
  This holds for direct chaining, with separate chaining the complexity is a bit higher.

- **best case:**
  lookup is an immediate success: complexity \( \in O(1) \).