7 Hashing: chaining

Summer Term 2011
Jan-Georg Smaus

Possible ways of treating 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; all 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.
```

Direct chaining

Keys are stored in overflow lists
\( h(k) = k \mod 7 \)

```
  0 1 2 3 4 5 6
  hash table T
  pointer
  ... colliding elements
```

Implementation in Java

```
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];
    }
    // 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);
    ... // only list operations
}
```
Implementation in Java

```java
class ChainedTableModel extends TableEntry {
    // constructor
    ChainedTableModel(Object key, Object value) {
        super(key, value);
        this.next = null;
    }

    // lookup key in the hash table
    ChainedTableModel p;
    p = (ChainedTableModel) tableEntry[h(key)];
    // Go through the list until end reached or key found
    while (p != null && !p.key.equals(key)) {
        p = p.next;
    }
    // Insert the element (if not there)
    if (!p.key.equals(key))
        p.next = entry;
}
```

```java
class ChainedHashTable extends HashTable {
    // the hash function
    public int h(Object key) {
        return key.hashCode() % capacity;
    }

    // lookup key
    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;
    }

    // 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) {
        if (p == null)
            return null;
        if (p.key.equals(key))
            return p.getNext();
        // otherwise:
        p.next = recDelete(p.next, key);
        return p;
    }
}
```

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();
    }
}
```

Analysis of direct chaining

Uniform hashing assumption:
- All hash addresses are chosen with the same probability, i.e.:
  \( P(h(x) = j) = \frac{1}{m} \)
- Independent from operation to operation

Average chain length for \( n \) entries (load factor): 
\[ \frac{n}{m} \]

Definition:
- \( C_\alpha \) = Expected number of entries inspected during a failed search
- \( C_n \) = Expected number of entries inspected during a successful search

Analysis:
\[ C_n = \frac{1}{2} \]

Analysis of direct chaining

Chaining

Advantages:
- \( C_\alpha \) and \( C_n \) are small
- \( C_n \) > 1 possible
- real distances
- suitable for secondary memory

Efficiency of lookup

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( C_n ) (successful)</th>
<th>( C_\alpha ) (unsuccessful)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>1.450</td>
<td>0.00</td>
</tr>
<tr>
<td>0.25</td>
<td>1.457</td>
<td>0.95</td>
</tr>
<tr>
<td>0.50</td>
<td>1.500</td>
<td>1.00</td>
</tr>
<tr>
<td>0.75</td>
<td>2.000</td>
<td>2.00</td>
</tr>
<tr>
<td>1.00</td>
<td>2.500</td>
<td>3.00</td>
</tr>
</tbody>
</table>

Disadvantages:
- Additional space for pointers
- Colliding elements are outside the hash table
## 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:
    - Complexity (in inspections) ≈ 1 + 0.5 × load factor
  - Failed lookup & insert:
    - Complexity = load factor
  - This holds for **direct chaining**, with **separate chaining** the complexity is a bit higher.
  - **Best case:**
    - Lookup is an immediate success: complexity ≈ \( O(1) \).