7 Hashing: chaining

Summer Term 2011

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

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.
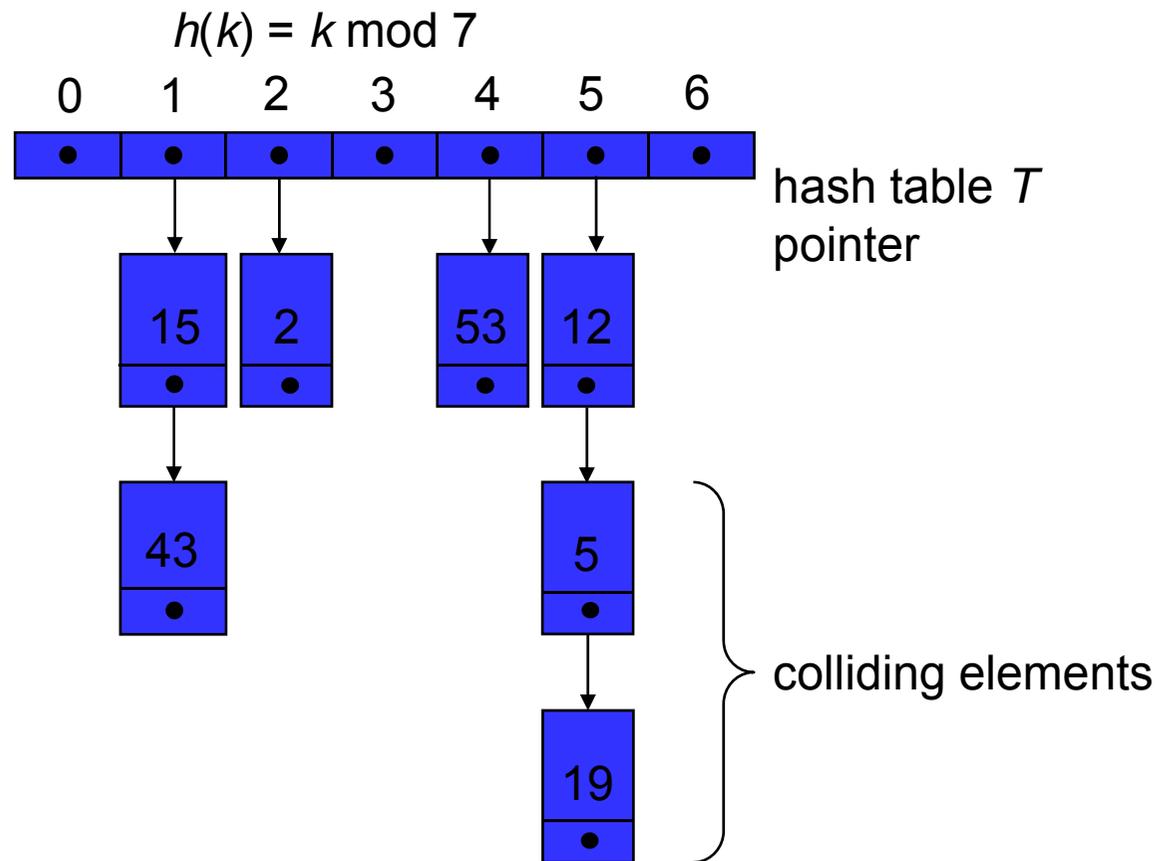
Chaining (1)

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

```
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

$$h(k) = k \bmod 7$$

hash table T
pointer

colliding elements

.

Chaining

Lookup key k
- Compute $h(k)$ and overflow list $T[h(k)]$
- Look for k in the overflow list

Insert a key k
- Lookup k (fails)
- Insert k in the overflow list

Remove a key k
- Lookup k (successfully)
- Remove k from the overflow list

⟶ only list operations

Implementation in Java

```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];
        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

```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 liste 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;
}
```

Implementation in Java

```java
/* 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

```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: -|                           0: -|
1: 15 -> 43 -|                  1: 15 -|
2: 2 -|                         2: 2 -|
3: -|                           3: -|
4: 53 -|                        4: -|
5: 12 -> 5 -> 19 -|             5: 5 -> 19 -|
6: -|                           6: -|
```

Analysis of direct chaining

Uniform hashing assumption:

- All hash addresses are chosen with the same probability, i.e.:

  $Pr(h(k_i) = j) = 1/m$

- independent from operation to operation

Average chain length for n entries (load factor):

$$n/m = \alpha$$

Definition:

C´$_n$ = Expected number of entries inspected during a failed search

C$_n$ = Expected number of entries inspected during a successful search

Analysis:

$$C'_n = \alpha$$
$$C_n \approx 1 + \frac{\alpha}{2}$$

Chaining

Advantages:

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

Efficiency of lookup

α	C_n (successful)	C'_n (unsuccessful)
0.50	1.250	0.50
0.90	1.450	0.90
0.95	1.457	0.95
1.00	1.500	1.00
2.00	2.000	2.00
3.00	2.500	3.00

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 $\in O(1)$.