Hashing: General Framework

- Set of keys $S$
- Universal $U$ of all possible keys
- Hash function $h$ 0, ..., $m-1$
- Hash table $T$

$h(s) = \text{hash address}$
$h(s) = h(s')$  $s$ and $s'$ are synonyms with respect to $h$; address collision

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 $B_{hash(s)}$ is already taken by another data set.
- What can we do with colliding elements?
  1. Chaining: We learned about that in the last chapter.
  2. Open Addressing: Colliding elements are stored in other vacant buckets. During storage and lookup, these are found through so-called probing.

Open addressing

- Idea: Store colliding elements in vacant (“open”) buckets of the hash table
- If $T[h(k)]$ is taken, find a different bucket for $k$ according to a fixed rule

General:
Consider the sequence
$(h(k) - j) \mod m$
$j = 0, \ldots, m-1$

Example:
Consider the bucket with the next smaller index:
$(h(k) - 1) \mod m$

Even more general:
Consider the probe sequence
$(h(k) - s(j,k)) \mod m$

Examples for the function $s(j,k)$
- $s(j,k) = j$ linear probing
- $s(j,k) = (-1)^j + j^2$ quadratic probing

Properties of $s(j,k)$
- Sequence:
  - $(h(k)) = (0,k) \mod m$
  - $(h(k)) = (1,k) \mod m$
  - $(h(k)) = (2,k) \mod m$
  - $(h(k)) = (3,k) \mod m$
  - $(h(k)) = (4,k) \mod m$
- $s(j,k)$ should result in a permutation of 0, ..., $m-1$.

Example: Quadratic probing
$(h(k)) = s(j,k) \mod m$

Critical:
Detection of data sets $\rightarrow$ mark as deleted
(Insert 4, 18, 25; delete 4; lookup 18, 25)
Open addressing

```java
// in HashTable: TableEntry [] T;
private int [] tag;
static final int EMPTY = 0;
static final int OCCUPIED = 1;
static final int DELETED = 2;

// Constructor
OpenHashTable (int capacity) {
    super(capacity);
    tag = new int [capacity];
    for (int i = 0; i < capacity; i++) {
        tag[i] = EMPTY;
    }
}

// The hash function
protected int h (Object key) {...}

// Function s for probe sequence
protected int s (int j, Object key) {
    // quadratic probing
    if (j % 2 == 0)
        return ((j + 1) / 2) * ((j + 1) / 2);
    else
        return -((j + 1) / 2) * ((j + 1) / 2);
}

public int searchIndex (Object key) {
    /* searches for an entry with the given key in the hash table and
    returns the respective index or -1 */
    int i = h(key);
    int j = 1; // next index of probing sequence
    while (tag[i] != EMPTY &&!key.equals(T[i].key)) {
        // Next entry in probing sequence
        i = (h(key) - s(j++, key)) % capacity;
        if (i < 0)
            i = i + capacity;
    }
    if (key.equals(T[i].key) && tag[i] == OCCUPIED)
        return i;
    else
        return -1;
}

public Object search (Object key) {
    /* searches for an entry with the given key in the hash table and
    returns the respective value or NULL */
    int i = searchIndex(key);
    if (i >= 0)
        return T[i].value;
    else
        return null;
}

public void insert (Object key, Object value) {
    // inserts an entry with the given key and value
    int j = 1; // next index of probing sequence
    int i = h(key);
    while (tag[i] == OCCUPIED) {
        i = (h(key) - s(j++, key)) % capacity;
        if (i < 0)
            i = i + capacity;
    }
    T[i] = new TableEntry(key, value);
    tag[i] = OCCUPIED;
}

public void delete (Object key) {
    // deletes entry with given key from the hash table
    int i = searchIndex(key);
    if (i >= 0) {
        // Successful search
        tag[i] = DELETED;
    }
}
```

Probe sequences – linear probing

```java
s(j) = j

Probes sequence for k:
R(k), R(k+1), ..., 0, m-1, ..., R(k)+1,

Problem:
“primary clustering”

Pr(next object ends at position 2) = 4/7
Pr(next object ends at position 1) = 1/7

Long chains are extended with higher probability than short ones.
```
Efficiency of linear probing

Successful search:

\[ C_n = \frac{1}{2} \left( 1 + \alpha \left( \frac{1}{2} \right) \right) \]

Failed search:

\[ C_n = \frac{1}{2} \left( 1 + \alpha \left( \frac{1}{2} \right)^2 \right) \]

<table>
<thead>
<tr>
<th>α</th>
<th>( C_n ) (successful)</th>
<th>( C_n ) (failed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>5.6</td>
<td>50.6</td>
</tr>
<tr>
<td>0.90</td>
<td>10.5</td>
<td>200.5</td>
</tr>
<tr>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Efficiency of linear probing decreases drastically as soon as the load factor \( \alpha \) gets close to the value 1.

Quadratic probing

\[ s(k) = (-1)^j \cdot \left( \frac{j}{2} \right)^2 \]

Probe sequence for \( k, h(k) + 1, h(k) - 1, h(k) + 4, \ldots \)

Permutation, if \( m = p \) is a prime of the form \( 4i + 3 \) for some \( i \).

Problem: secondary clustering, i.e. two synonyms \( k \) and \( k' \) always run through the same probe sequence.

Efficiency of quadratic probing

Successful search:

\[ C_n = 1 - \frac{1}{2} + h \left( \frac{1}{2} \right)^2 \]

Failed search:

\[ C_n = \frac{1}{2} - \alpha + h \left( \frac{1}{2} \right)^2 \]

<table>
<thead>
<tr>
<th>α</th>
<th>( C_n ) (successful)</th>
<th>( C_n ) (failed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>2.5</td>
<td>15.5</td>
</tr>
<tr>
<td>0.90</td>
<td>3.12</td>
<td>23.20</td>
</tr>
<tr>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>