8 Hashing: open addressing
Hashing: general framework

Set of keys $S$

Universe $U$ of all possible keys

hash function $h$

hash table $T$

$h(s) = \text{hash address}$

$h(s) = h(s') \iff s \text{ and } s' \text{ are synonyms with respect to } h$

address collision

$(H(u) \subseteq [-2^{31}, 2^{31}])$
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.
Hashing by chaining

Keys are stored in **overflow lists**

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

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
15 & 2 & 53 & 12 & \bullet & \bullet & \bullet \\
\bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
43 & 5 & 19 & \bullet & \bullet & \bullet & \bullet \\
\end{array}
\]

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

Example:
consider the bucket with the next smaller index:

$\left( h(k) - 1 \right) \mod m$

0  1  h(k)  m-2  m-1
...

General:
consider the sequence

$\left( h(k) - j \right) \mod m$

$j = 0, \ldots, m-1$
Even more general:

consider the probe sequence

\[(h(k) - s(j,k)) \mod m\]

\[j = 0, ..., m-1, \text{ for a given function } s(j,k)\]

**Examples** for the function

\[s(j, k) = j\]  \hspace{1cm} (linear probing)

\[s(j, k) = (-1)^j \cdot \left\lfloor \frac{j}{2} \right\rfloor^2\]  \hspace{1cm} (quadratic probing)

\[s(j, k) = j \cdot h'(k)\] \hspace{1cm} (double hashing)
Probe sequence

**Properties** of \( s(j,k) \) sequence

\[
(h(k) - s(0,k)) \mod m,
(h(k) - s(1,k)) \mod m,
(h(k) - s(m-2,k)) \mod m,
(h(k) - s(m-1,k)) \mod m
\]

should result in a permutation of 0, ..., \( m-1 \).

**Example**: quadratic probing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>h(11) = 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Critical**: \( s(j,k) = -1,1,-4,4,-9,9 \)

deletion of data sets \( \rightarrow \) mark as deleted

(insert 4, 18, 25; delete 4; lookup 18, 25)
class OpenHashTable extends HashTable {
    // 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);
    }
Open addressing - lookup

```java
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;
}
```
Open addressing - insert

```java
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;
    }
}
public class OpenHashingTest {
    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]);
        OpenHashTable h = new OpenHashTable(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 OpenHashingTest 12 53 5 15 2 19 43

Output (quadratic probing):
[ ] [ ] [ ] [ ] [ ] [ ] (12) [ ]
[ ] [ ] [ ] [ ] [ ] (53) (12) [ ]
[ ] [ ] [ ] [ ] [ ] (53) (12) (5)
[ ] (15) [ ] [ ] (53) (12) (5)
[ ] (15) (2) [ ] (53) (12) (5)
[ ] (15) (2) (19) (53) (12) (5)
(43) (15) (2) (19) (53) (12) (5)

{43} (15) (2) (19) {53} {12} (5)
Probe sequences – linear probing

\[ s(j,k) = j \]

Probe sequence for \( k \):

\[ h(k), h(k)-1, ..., 0, m-1, ..., h(k)+1, \]

Problem:

“primary clustering”

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\text{5} & \text{53} & \text{12} & & & & \\
\end{array}
\]

\[ Pr \text{ (next object ends at position 2) } = \frac{4}{7} \]

\[ Pr \text{ (next object ends at position 1) } = \frac{1}{7} \]

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

Successful search:

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

Failed search:

\[ C'_n \approx \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right) \]

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( C_n ) (successful)</th>
<th>( C'_n ) (failed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>1.5</td>
<td>2.5</td>
</tr>
<tr>
<td>0.90</td>
<td>5.5</td>
<td>50.5</td>
</tr>
<tr>
<td>0.95</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

\[ s(j,k) = (-1)^j \times \left\lfloor \frac{j}{2} \right\rfloor^2 \]

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

Permutation, if \( m = 4l + 3 \) is prime.

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

Successful search:

\[ C_n \approx 1 - \frac{\alpha}{2} + \ln \left( \frac{1}{1 - \alpha} \right) \]

Failed search:

\[ C'_n \approx \frac{1}{1 - \alpha} - \alpha + \ln \left( \frac{1}{1 - \alpha} \right) \]

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( C_n ) (successful)</th>
<th>( C'_n ) (failed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>1.44</td>
<td>2.19</td>
</tr>
<tr>
<td>0.90</td>
<td>2.85</td>
<td>11.40</td>
</tr>
<tr>
<td>0.95</td>
<td>3.52</td>
<td>22.05</td>
</tr>
<tr>
<td>1.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Uniform probing

\[ s(j, k) = \pi_k(j) \]

\( \pi_k \) one of the \( m! \) permutations of \( \{0, ..., m-1\} \)
- only depends on \( k \)
- same probability for each permutation

\[ C'_n \leq \frac{1}{1-\alpha} \]

\[ C_n \approx \frac{1}{\alpha} \times \ln \left( \frac{1}{(1-\alpha)} \right) \]

<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>1.39</td>
<td>2</td>
</tr>
<tr>
<td>0.90</td>
<td>2.56</td>
<td>10</td>
</tr>
<tr>
<td>0.95</td>
<td>3.15</td>
<td>20</td>
</tr>
<tr>
<td>1.00</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Double hashing

Idea: Choose another hash function $h'$

$$ s(j,k) = j \cdot h'(k) $$

Probe sequence for $k$:

$$ h(k), h(k) - h'(k), h(k) - 2h'(k), ... $$

Requirement:

probing sequence must correspond to a permutation of the hash addresses.

Hence:

$$ h'(k) \neq 0 \text{ and } h'(k) \text{ no factor of } m, \text{ i.e. } h'(k) \text{ does not divide } m. $$

Example:

$$ h'(k) = 1 + (k \mod (m-2)) $$
## Example

Hash functions: \( h(k) = k \mod 7 \)
\[ h'(k) = 1 + k \mod 5 \]

Insert sequence: 15, 22, 1, 29, 26

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \( h'(22) = 3 \)
- \( h'(1) = 2 \)
- \( h'(29) = 5 \)
- \( h'(26) = 2 \)

In this example we can do with a single probing step almost every time.

- Double hashing is **as efficient as uniform probing**.
- Double hashing is **simpler to implement**.