README

UE1 Hashtable

1) Hashfunktion

- because modulo is an expensive operation we chose to have a table, which is a power of 2(since it is a trivial bitmask)
- should be uniform
- should be fast
- sometimes specific length or value
- chose hash function from slides

2) Kollisionserkennung

- done via quadratic probing as demanded

- implemented with quadratic probing iterator

3) Verwaltung der Kursdaten

- Course data is imported from: http://de.finance.yahoo.com/q/hp?s=MSFT.

- Serializing of course data done via json library: https://github.com/nlohmann/json

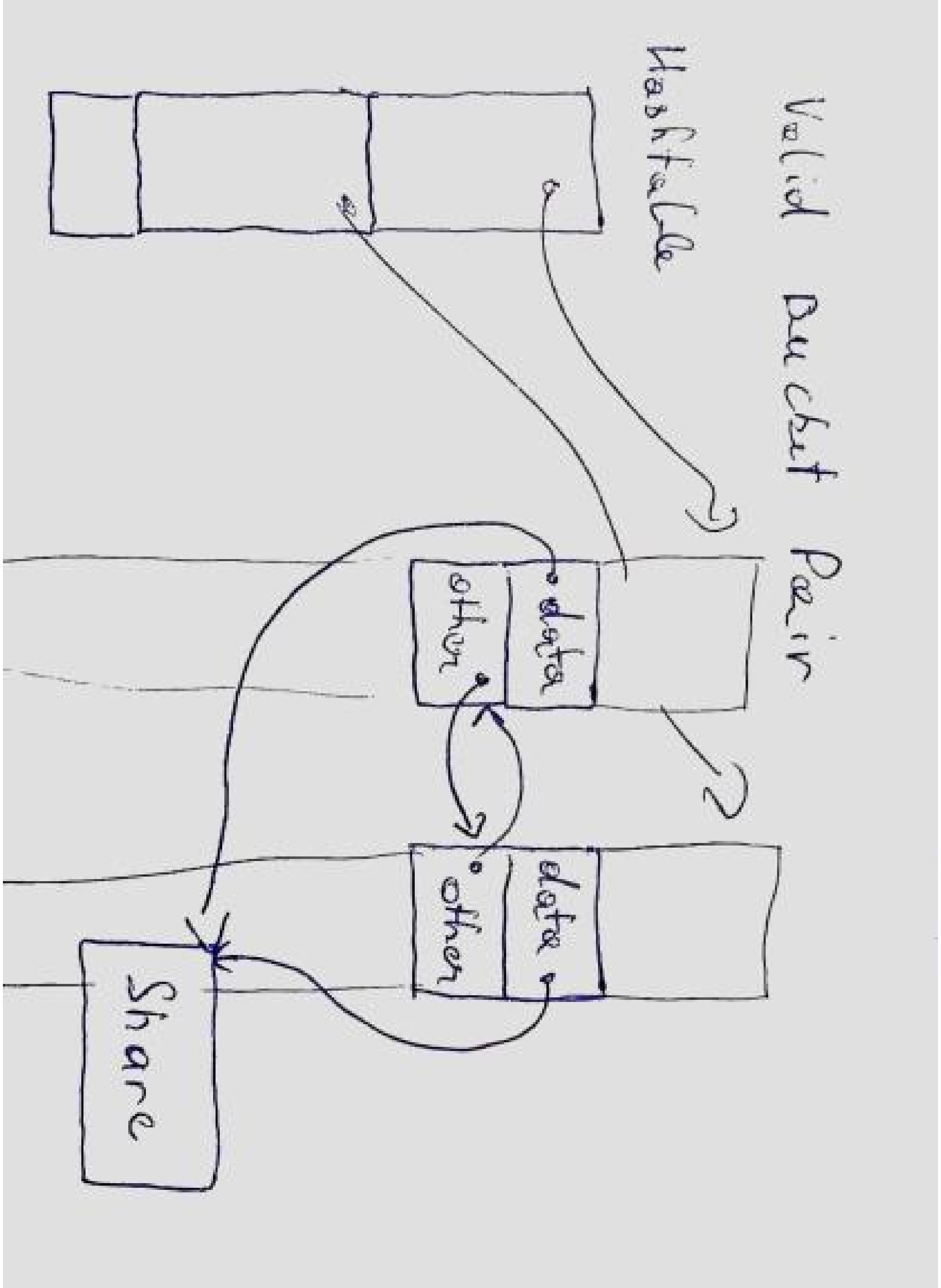
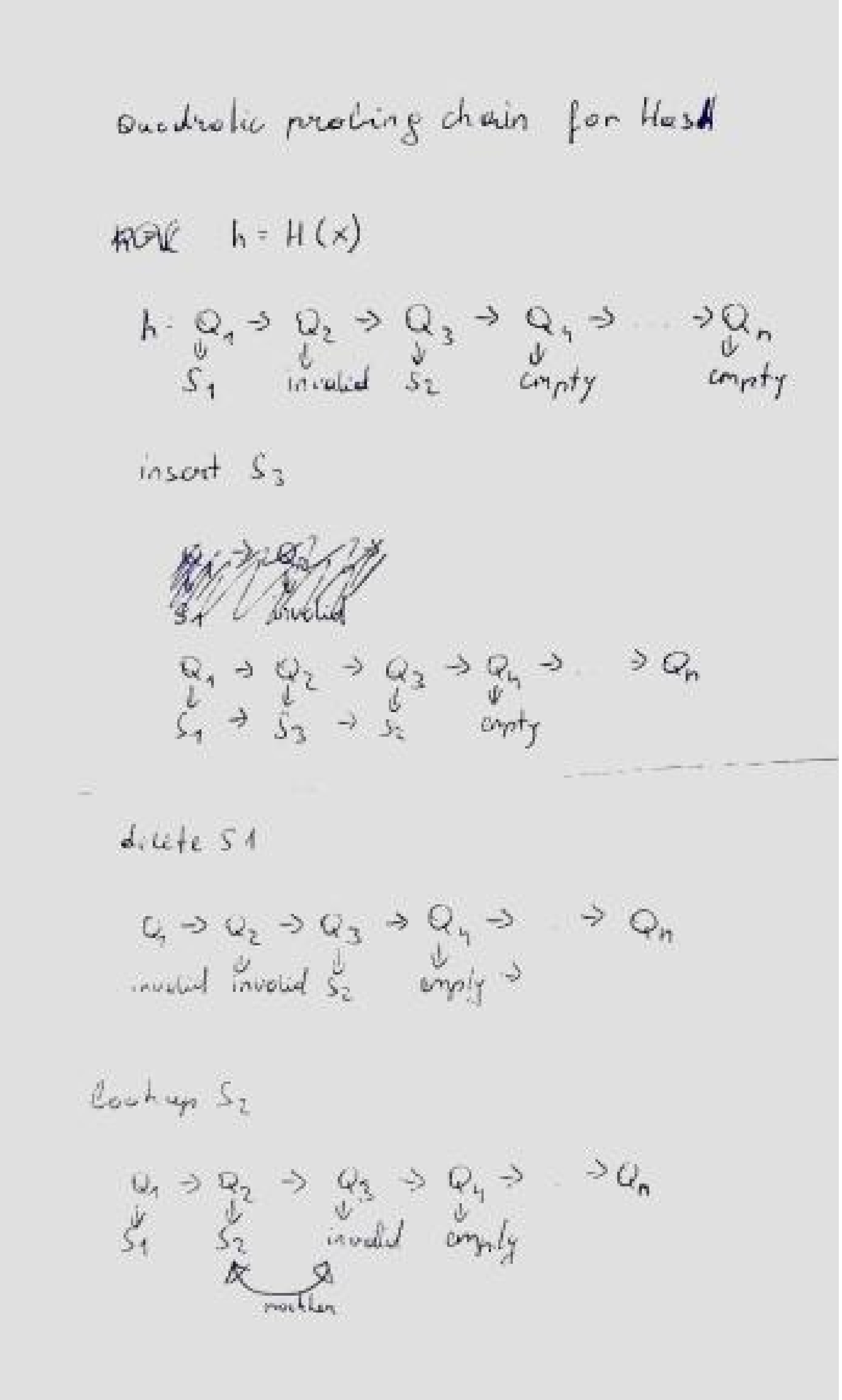


Figure 1: overview

4) Löschalgorithmus



Quadratic probing chain for Hash

$h = H(x)$

$$\begin{tikzcd}
h: Q_1 \arrow[r] \arrow[d] & Q_2 \arrow[r] \arrow[d] & Q_3 \arrow[r] \arrow[d] & Q_4 \arrow[r] \arrow[d] & \cdots \arrow[r] & Q_n \arrow[d] \\
S_1 & \text{invalid} & S_2 & \text{empty} & & \text{empty}
\end{tikzcd}$$

insert S_3

$$\begin{tikzcd}
Q_1 \arrow[r] \arrow[d] & Q_2 \arrow[r] \arrow[d] & Q_3 \arrow[r] \arrow[d] & Q_4 \arrow[r] \arrow[d] & \cdots \arrow[r] & Q_n \\
S_1 \arrow[r] & S_3 \arrow[r] & S_2 & \text{empty} & &
\end{tikzcd}$$

delete S1

$$\begin{tikzcd}
Q_1 \arrow[r] & Q_2 \arrow[r] \arrow[d] & Q_3 \arrow[r] \arrow[d] & Q_4 \arrow[r] \arrow[d] & \cdots \arrow[r] & Q_n \\
\text{invalid} & \text{invalid} & S_2 & \text{empty} \arrow[r] & &
\end{tikzcd}$$

Lookup S_2

$$\begin{tikzcd}
Q_1 \arrow[r] \arrow[d] & Q_2 \arrow[r] \arrow[d] & Q_3 \arrow[r] \arrow[d] & Q_4 \arrow[r] \arrow[d] & \cdots \arrow[r] & Q_n \\
S_1 & S_2 \arrow[r, bend right, leftrightarrow, "\text{marken}"'] & \text{invalid} & \text{empty} & &
\end{tikzcd}$$

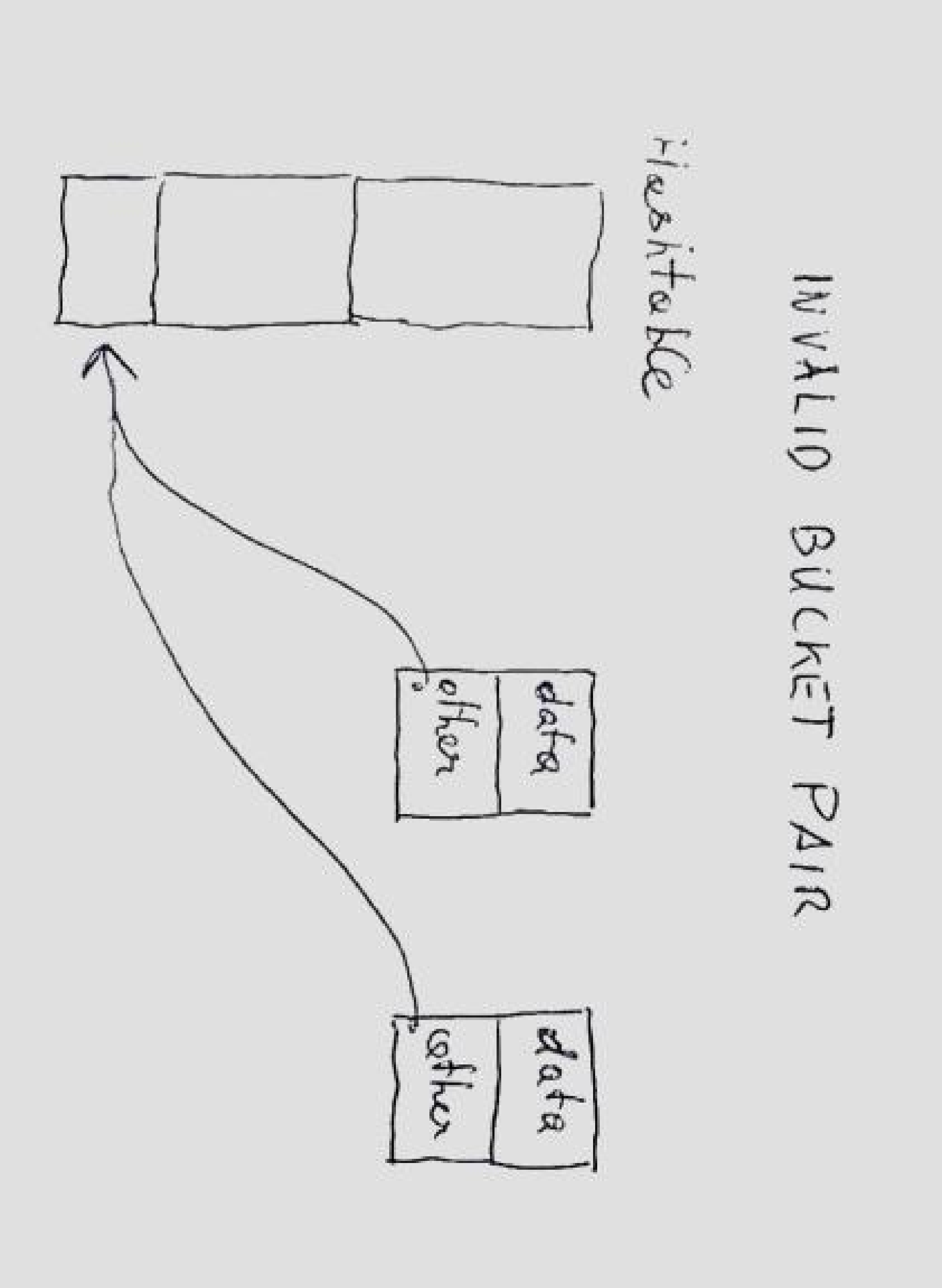


Figure 2: invalid bucket pair

5) Aufwandsabschätzung

5.1 insert, search delete Hashtable

Best case: O(1)
Worst case: O(n)
* higher filling level (probing)
* n = table capacity

5.2 insert array

Worst case O(1)

5.3 delete array

Worst case O(1)

5.4 search array

- search for key
  - **worst case: O(n)**

  - **best case: O(1)**

- search via index
  - **worst case O(1)**

5.5 insert list

- **insert front O(1)**

- **insert back O(n)**

5.6 delete list

- **delete front O(1)**

- **delete back O(n)**

5.7 search list

- **worst case O(n)**

5.8 Benchmarks

```
=== inserting ===
std::unordered_map: average: 298ns from 10000 iterationsto insert 1 random share with load factor:0
std::unordered_map: average: 720ns from 10000 iterationsto insert 1 random share with load factor:0.23497
std::unordered_map: average: 1022ns from 10000 iterationsto insert 1 random share with load factor:0.46994
```

```
std::unordered_map: average: 992ns from 10000 iterationsto insert 1 random share with load factor:0.704911
std::unordered_map: average: 1062ns from 10000 iterationsto insert 1 random share with load factor:0.845801

hashtable: average: 139ns from 10000 iterationsto insert 1 random share with load factor:0
hashtable: average: 240ns from 10000 iterationsto insert 1 random share with load factor:0.25
hashtable: average: 327ns from 10000 iterationsto insert 1 random share with load factor:0.5
hashtable: average: 482ns from 10000 iterationsto insert 1 random share with load factor:0.75
hashtable: average: 678ns from 10000 iterationsto insert 1 random share with load factor:0.9

vector: average: 2392ns from 10000 iterations to insert 1 random share at random position
vector: average: 119ns from 10000 iterations to insert 1 random share at back

list: average: 3103ns from 10000 iterations to insert 1 random share at back
list: average: 338ns from 10000 iterations to insert 1 random share at rand pos

=== look up ===
hashtable: average: 149ns from 10000 iterations to look up 1 random share with load factor: 0.1
hashtable: average: 219ns from 10000 iterations to look up 1 random share with load factor: 0.25
hashtable: average: 312ns from 10000 iterations to look up 1 random share with load factor: 0.5
hashtable: average: 348ns from 10000 iterations to look up 1 random share with load factor: 0.75
hashtable: average: 370ns from 10000 iterations to look up 1 random share with load factor: 0.9

vector: average: 5644ns from 10000 iterations to look up 1 random share by key
list: average: 6056ns from 10000 iterations to look up 1 random share by key
vector: average: 122ns from 10000 iterations to look up 1 random share by pos
list: average: 3088ns from 10000 iterations to look up 1 random share by pos deleting
hashtable: average: 92ns from 10000 iterations to look up 1 element after filling up to 0.95and then deleting
to0.1
```