Hashing

Naive Methode, Telefon–Buch abzuspeichern:

1. Annahme: Alle Namen bestehen aus genau 10 Buchstaben

   kürzere Namen werden mit Blanks aufgefüllt

2. Übersetze Namen in eindeutigen Index:

```
unsigned computeIndex(const char text[]) {
    unsigned index = 0;
    for (unsigned i = 0; i < 10; ++i) {
        index = 128 * index + text[i];
    }
    return index;
}
```

   $$\texttt{index} = \sum_{i=0}^{9} \texttt{text}[i] * 128^{9-i}$$

3. Lege Feld der Größe 128^{10} an:

```
unsigned telephone[1 180 591 620 717 411 303 424];
```

4. Speichere Telefon–Nummer unter berechneten Index:

```
index = computeIndex(name);
telephone[index] = telNumber;
```

5. Suchen: Berechne Index des gegebenen Namens:

```
index = computeIndex(name);
telNumber = telephone[index]
```

Analyse der naiven Methode

1. Sei n Zahl der Einträge im Telefon–Buch.
2. Aufwand, um neuen Eintrag hinzuzufügen:
   $\mathcal{O}(1)$
3. Aufwand, um Eintrag nachzuschlagen:
   $\mathcal{O}(1)$
4. Aufwand, um Telefon–Buch aufzubauen:
   $\mathcal{O}(n)$

$$\boxed{\boxed{\text{Besser gehts nicht!}}}$$

Probleme der naiven Methode:

1. Braucht sehr, sehr viel Speicher!
2. Funktioniert nicht, wenn Namen mehr als 10 Buchstaben haben.
3. In der Funktion `computeIndex` tritt Überlauf ein.

Hashing

Ziel: Entwicklung einer Methode, die

1. im statistischen Mittel vergleichbare Performanz hat wie die naive Methode

2. Speicherverbrauch: $\mathcal{O}(n)$

Erster Versuch:

1. Anlegen eines Feldes der Größe n

   ```
   unsigned telephone[n];
   ```

2. Berechnung des Index

   ```
   unsigned computeIndex(const char text[], unsigned n)
   {
       unsigned index = 0;
       for (unsigned i = 0; i < 10; ++i) {
           index = (128 * index + text[i]) % n;
       }
       return index;
   }
   ```

3. Speichere Telefon–Nummer unter berechneten Index:

   ```
   index = computeIndex(name,n);
   telephone[index] = telNumber;
   ```

4. Suchen: Berechne Index des gegebenen Namens:

   ```
   index = computeIndex(name,n);
   telNumber = telephone[index]
   ```

Analyse des 1. Versuchs

1. Performanz: **ok**
2. Speicher–Verbrauch: Feld der Länge n: **ok**
3. **Korrektheit**? **Nein**!

Problem: Kollisionen

```
computeIndex( "Pfennig",   22 ) = 12
computeIndex( "Schlensog", 22 ) = 12
```

Verschiedene Schüssel können auf den gleichen Wert abgebildet werden!

Lösung: Feld `telephone` enthält nicht

- Telefon–Nummer, sondern
- Zeiger auf verkette Liste mit Paaren $\langle \texttt{Name}, \texttt{telNr} \rangle$
- Diese Listen dienen als Dictionary!

Suchen nun in zwei Stufen:

1. Berechnung des Index

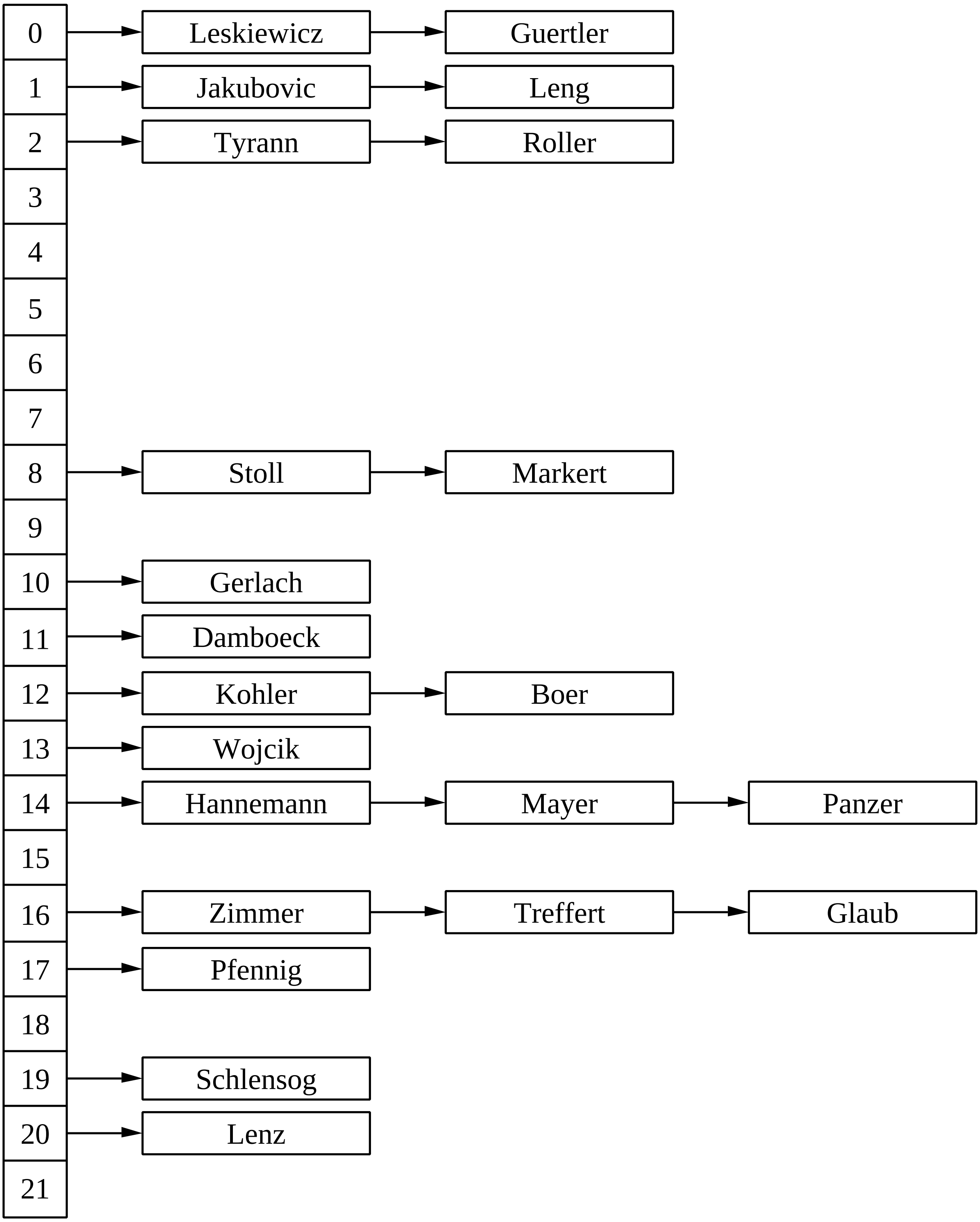
   ```
   computeIndex(key, n);
   ```

2. Suche in verketteter Liste `telephone[index]`

   ```
   search(telephone[index], key)
   ```

Einfügen und Löschen analog

Hash–Tabelle mit getrennter Verkettung



Implementierung in C

Hash–Tabelle benötigt folgende Daten

1. Zahl der Einträge

   ```
   unsigned noEntries;
   ```

2. Größe des angelegten Feldes

   ```
   unsigned noBuckets;
   ```

3. Feld mit Zeigern auf Listen–Knoten

   ```
   List* table;
   ```

Also Definition der Daten–Struktur wie folgt:

```
typedef struct {
    unsigned noEntries;
    unsigned noBuckets;
    List*    table;
} HashTable;
```

Definition: *Auslastungs–Faktor* (load factor)

$$\alpha = \frac{\texttt{noEntries}}{\texttt{noBuckets}}$$

Falls Hash–Funktion die Schlüssel gut über Tabelle verteilt, gibt α mittlere Länge der Listen an.

Praxis: $\alpha \leq 4$ ist gute Wahl.

Implementierung (Fortsetzung)

1. Suchen

```
Value* searchTable(HashTable* htPtr, Key key)
{
    unsigned index = hash(key, htPtr->noBuckets);
    return search(htPtr->table[index], key);
}
```

2. Einfügen

```
void insertTable(HashTable* htPtr, Key key, Value val)
{
    ++(htPtr->noEntries);
    unsigned index = hash(key, htPtr->noBuckets);
    htPtr->table[index] =
        insert(htPtr->table[index], key, val);
}
```

3. Löschen

```
void deleteTable(HashTable* htPtr, Key key)
{
    --(htPtr->noEntries);
    unsigned index = hash(key, htPtr->noBuckets);
    htPtr->table[index] =
        delete(htPtr->table[index], key);
}
```

Anlegen der Hash–Tabelle

Anlegen einer Hash–Tabelle mit `size` Buckets

```
HashTable* makeTable(unsigned size) {
    HashTable* ht = malloc( sizeof(HashTable) );
    unsigned index = 0;
    unsigned count = primes[index];
    while (count < size) {
        count = primes[++index];
    }
    ht->noBuckets = count;
    ht->noEntries = 0;
    ht->table = malloc( sizeof(NodePtr) * count );
    for (unsigned i = 0; i < ht->noBuckets; ++i) {
        ht->table[i] = 0;
    }
    return ht;
}
```

1. `primes`: Feld, was wachsende Primzahlen enthält.

   $\texttt{primes}[i+1] \approx 2 * \texttt{primes}[i]$

2. `index` wird hochgezählt, bis Primzahl `count` gefunden ist, die Größer als `size` ist.

3. Dann wird Feld der Größe `count` angelegt.

4. Feld wird mit 0–Pointern initialisiert.

   (0–Pointer $\widehat{=}$ leere Liste)

Komplexität

Abschätzung der Komplexitäten für naives *Hashing*

n: Zahl der Einträge in Hash–Tabelle

α: Auslastungs–Faktor

1. Suche:
   (a) Worst Case $n \in \mathcal{O}(n)$
   (b) Statistischer Durchschnitt $\frac{\alpha}{2} \in \mathcal{O}(1)$
2. Löschen:
   (a) Worst Case $n \in \mathcal{O}(n)$
   (b) Statistischer Durchschnitt $\frac{\alpha}{2} \in \mathcal{O}(1)$
3. Einfügen:
   (a) Worst Case $\mathcal{O}(1)$
   (b) Statistischer Durchschnitt $\mathcal{O}(1)$

Worst Case:

1. Hash–Funktion berechnet immer selben Index
2. Alle n Einträge in einer Zeile

Statistischer Mittelwert

1. Jede Zeile enthält im Durchschnitt α Einträge

Hash–Funktion

Definition:

Hash–Funktion berechnet zu gegeb. Schlüssel den Index:

$$\textit{hash} : \textit{Key} \to \mathbb{N}$$

Anforderungen:

1. Ist n Größe der Tabelle, so muß gelten:
   $$\textit{hash}(k) < n$$
2. Schlüssel sollen möglichst gleichmäßig auf Tabelle verteilt werden
3. Hash–Funktion soll einfach zu berechnen sein

Gebräuchliche Hash–Funktionen

1. Divisions–Methode
   $$\textit{hash}(k) \;=\; k \,\%\, n$$
   Dabei sollte Tabellen–Größe n Primzahl sein!
2. Multiplikations–Methode: Sei $k \in \mathbb{R}$
   $$\texttt{floor}\big(n * \texttt{fract}(A * k)\big)$$
   (a) $\texttt{floor}(x)$: größte natürliche Zahl kleiner x
   (b) $\texttt{fract}(x) := x - \texttt{floor}(x)$
   (c) $A := \dfrac{\sqrt{5}-1}{2} \approx 0.61803398875$

Hash–Funktion für Strings

```
int hash(const char* key, unsigned size) {
    const    char* ptr;
    unsigned result;
    result = 0;
    ptr    = key;
    while (*ptr != '\0') {
        result = (result << 4) + (*ptr);
        unsigned tmp = result & 0xf0000000;
        if (tmp != 0) {
            result = result ^ (tmp >> 24);
            result = result ^ tmp;
        }
        ptr++;
    }
    return result % size;
}
```

1. Alle Buchstaben werden berücksichtigt
2. Verschiedene Bits der Buchstaben werden durch die Operationen ˆ (exklusives Oder) und Addition vermischt
3. Wird in der Praxis für Symbol–Tabellen in Compilern eingesetzt.

   (siehe auch Aho, Sethi, Ullmann:

   Compilers: Principles, Techniques, and Tools)

Kritik der bisherigen Methode

Probleme der Implementierung:

1. Bisher vorgestelltes Verfahren setzt Kenntnis der maximalen Anzahl von Einträgen in Tabelle voraus

2. Wird Tabelle zu klein gewählt:

   schlechte Performanz

3. Wird Tabelle zu groß gewählt:

   Verschwendung von Speicherplatz

4. Falls Zahl der Einträge mit der Zeit wächst, kann der Auslastungs–Faktor gar nicht optimal eingestellt werden:

   (a) entweder ist die Tabelle am Anfang zu groß

   (b) oder die Tabelle ist später zu klein

Lösung: Tabelle muß dynamisch wachsen können

1. Bei jedem Einfügen kontrollieren des Auslastungs–Faktors α

2. Falls $\alpha > 4$ *Rehashing*:

   (a) Lege doppelt so große neue Tabelle an

   (b) Kopiere alte Tabelle in neue Tabelle

Implementierung Rehashing

```
HashTable* rehash(HashTable* htPtr) {
    HashTable* newTable =
        makeTable(htPtr->noBuckets * 2);
    newTable->noEntries = htPtr->noEntries;
    for (unsigned idx = 0; idx < htPtr->noBuckets; ++idx)
    {
        NodePtr nodePtr = htPtr->table[idx];
        while (nodePtr != 0) {
            Key      key   = nodePtr->key;
            Value    val   = nodePtr->val;
            unsigned index =
                hash(key, newTable->noBuckets);
            newTable->table[index] =
                insert(newTable->table[index], key, val);
            nodePtr = nodePtr->nextPtr;
        }
    }
    freeHashTable(htPtr);
    free(htPtr);
    return newTable;
}
```

Algorithmus

1. neue Tabelle anlegen

2. alte Tabelle Zeile für Zeile durchgehen

3. jeden Eintrag aus alter Tabelle in neue Tabelle einfügen

4. alte Tabelle löschen

Komplexität

Abschätzung der Komplexitäten für *dynamisches Hashing*

Sei n Zahl der Einträge in Hash–Tabelle

1. Suche:
   (a) Worst Case $\mathcal{O}(n)$
   (b) Statistischer Durchschnitt $\mathcal{O}(1)$
2. Löschen:
   (a) Worst Case $\mathcal{O}(n)$
   (b) Statistischer Durchschnitt $\mathcal{O}(1)$
3. Einfügen:
   (a) Worst Case $\mathcal{O}(n)$
   (b) Statistischer Durchschnitt $\mathcal{O}\big(\log(n)\big)$

Worst Case:

1. Hash–Funktion berechnet immer selben Index
2. Alle n Einträge in einer Zeile

Statistischer Mittelwert

1. Jede Zeile enthält α Einträge
2. $\alpha < 4$