Programmiertechnik II

Ralf Herbrich

Suchen

Überblick

1. Symboltabellen
2. Sequentielle Suche
3. Binäre Suche
4. Fibonacci Suche

Überblick

1. **Symboltabellen**
2. Sequentielle Suche
3. Binäre Suche
4. Fibonacci Suche

Symboltabellen

- Symboltabellen sind eine **Abstraktion** von Schlüssel-Werte Datenbanken (*key value database*)
  - $\textbf{Put}(k, v)$: Füge einen Wert v für einen gegeben Schlüssel k ein
  - $\textbf{Get}(k)$: Finde den Wert v für einen gegeben Schlüssel k
- **Beispiel**: *Domain Name System* (DNS)
  - $\textbf{Put}(k, v)$: Füge eine IP-Adresse v für eine gegebene Domäne k ein
  - $\textbf{Get}(k)$: Finde die IP-Adresse v für eine gegebene Domäne k

Domänenname	IP-Adresse
www.hpi.de	141.89.225.126
www.uni-potsdam.de	141.89.239.5
www.mit.edu	23.10.78.97
www.stanford.edu	151.101.194.133

Schlüssel (*key*) — Wert (*value*)

Anwendungen von Symboltabellen

Anwendung	Suche nach …	Schlüssel	Wert
Wörterbuch	Definition	Wort	Definition
Buchindex	Relevante Seiten	Wort	Liste aller Seiten
Buchhaltung	Transaktion	Konto	Transaktionen
Websuche	Relevante Webseite	Schlagwort	Liste aller Webseiten
Compiler	Eigenschaften von Variablen	Variablenname	Variablentyp und -wert
Genomik	Marker	DNA-Kette	Position im Genom
Dateisystem	Dateiinhalt	Dateiname	Adresse auf dem Speicher

Symboltabellen API

Abstrakte **`ST`** (*Symbol Table*) Klasse

```
// Implements the base class for a symbol table
template <typename Key, typename Value>
class ST {
public:
    // put a key-value pair into the table
    virtual void put(const Key& key, const Value& val) = 0;
    // gets a value for a given key
    virtual const Value* get(const Key& key) const = 0;
    // removes a key from the table
    virtual void remove(const Key& key) = 0;
    // checks if there is a value paired with a key
    virtual bool contains(const Key& key) const = 0;
    // checks if the symbol table is empty
    virtual bool is_empty() const = 0;
    // number of key-value pairs in the table
    virtual int size() const = 0;
};
```

Generische Implementation mit variable Typen für die Schlüssel und Werte

Füge den Wert **`val`** für den Schlüssel **`key`** ein

Suche den Wert für den Schlüssel **`key`**

Entferne den Schlüssel **`key`** und den Wert **`val`**

Überprüfe, ob der Schlüssel **`key`** enthalten ist

Überprüfe, ob die Tabelle leer ist

Größe der Symboltabelle

- **`put`** überschreibt den Wert mit **`val`**, wenn es den Schlüssel **`key`** schon gibt
- **`get`** gibt **`nullptr`** zurück, wenn es den Schlüssel **`key`** noch nicht gibt

Symboltabellen API (*ctd*)

- Die **`contains`** Methode kann man mit Hilfe der **`get`** Methode implementieren

```
// checks if there is a value paired with a key
bool contains(const Key& key) const {
    return (get(key) != nullptr);
}
```

- Die **`is_empty`** Methode kann man mit Hilfe der **`size`** Methode implementieren

```
// checks if the symbol table is empty
bool is_empty() const { return (size() == 0); }
```

- Um Vergleiche durchzuführen, nehmen wir an das die Operatoren **<=**, **==**, und **>=** definiert sind
  - Da wir diese Operationen oft benutzen, sollten die Operationen schnell sein (und $\mathcal{O}(1)$ Laufzeit haben)

Symboltabelle Laufzeitanalyse

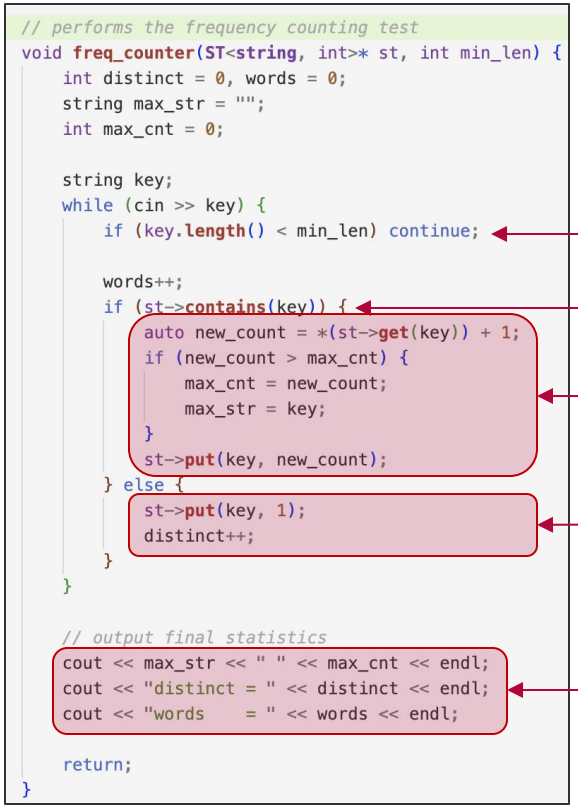
- Eine häufige Anwendung ist ein Häufigkeitszähler (*frequency counter*)
  - Liest eine Sequenz von Zeichenketten (Worten) von der Standardeingabe
  - Gibt das häufigste Wort aus (und die Häufigkeit), sowie die Anzahl von eindeutigen und nicht-eindeutigen Worten

```
→  unit8 git:(main) ✗ more ../data/tinyTale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
→  unit8 git:(main) ✗ ./freq_counter 1 < ../data/tinyTale.txt
it 10
distinct = 20
words    = 60
→  unit8 git:(main) ✗ ./freq_counter 8 < ../data/tale.txt
business 122
distinct = 5131
words    = 14350
→  unit8 git:(main) ✗ ./freq_counter 10 < ../data/leipzig100K.txt
government 2549
distinct = 38468
words    = 160340
```

Testproblem zum Debuggen

Echte Probleme

Symboltabelle Laufzeitanalyse Programm



```
// performs the frequency counting test
void freq_counter(ST<string, int>* st, int min_len) {
    int distinct = 0, words = 0;
    string max_str = "";
    int max_cnt = 0;

    string key;
    while (cin >> key) {
        if (key.length() < min_len) continue;

        words++;
        if (st->contains(key)) {
            auto new_count = *(st->get(key)) + 1;
            if (new_count > max_cnt) {
                max_cnt = new_count;
                max_str = key;
            }
            st->put(key, new_count);
        } else {
            st->put(key, 1);
            distinct++;
        }
    }

    // output final statistics
    cout << max_str << " " << max_cnt << endl;
    cout << "distinct = " << distinct << endl;
    cout << "words    = " << words << endl;

    return;
}
```

Liest eine Zeichenkette und überprüft die minimale Länge

Überprüft, ob die Zeichenkette schon gesehen wurde

Wenn Zeichenkette schon gesehen wurde, wird der Zähler um eins erhöht und eventuell die häufigste Zeichenkette verändert

Wenn nicht, wird ein neuer Eintrag angelegt und der Zähler für eindeutige Zeichenketten inkrementiert

Ausgabe aller Statistiken auf dem Bildschirm

Überblick

1. Symboltabellen
2. **Sequentielle Suche**
3. Binäre Suche
4. Fibonacci Suche

Sequentielle Suche

- Einfachste Methode, um eine Symboltabelle zu implementieren
- Basiert auf einer einfach verketteten Liste
  - **Put**(k, v): Beginnend beim Listenkopf werden die Schlüssel verglichen und der Wert verändert; wenn die Liste den Schlüssel nicht enthält, wird ein neuer Listenkopf erstellt, da $\mathcal{O}(1)$!
  - **Get**(k): Beginnend beim Listenkopf werden die Schlüssel verglichen

```
// a helper linked list data type
struct Node {
    Key key;
    Value val;
    Node* next;

    // constructor with values
    Node(const Key& k, const Value& v, Node* n) :
        key(k), val(v), next(n) {}
};
```

Put(k, v)

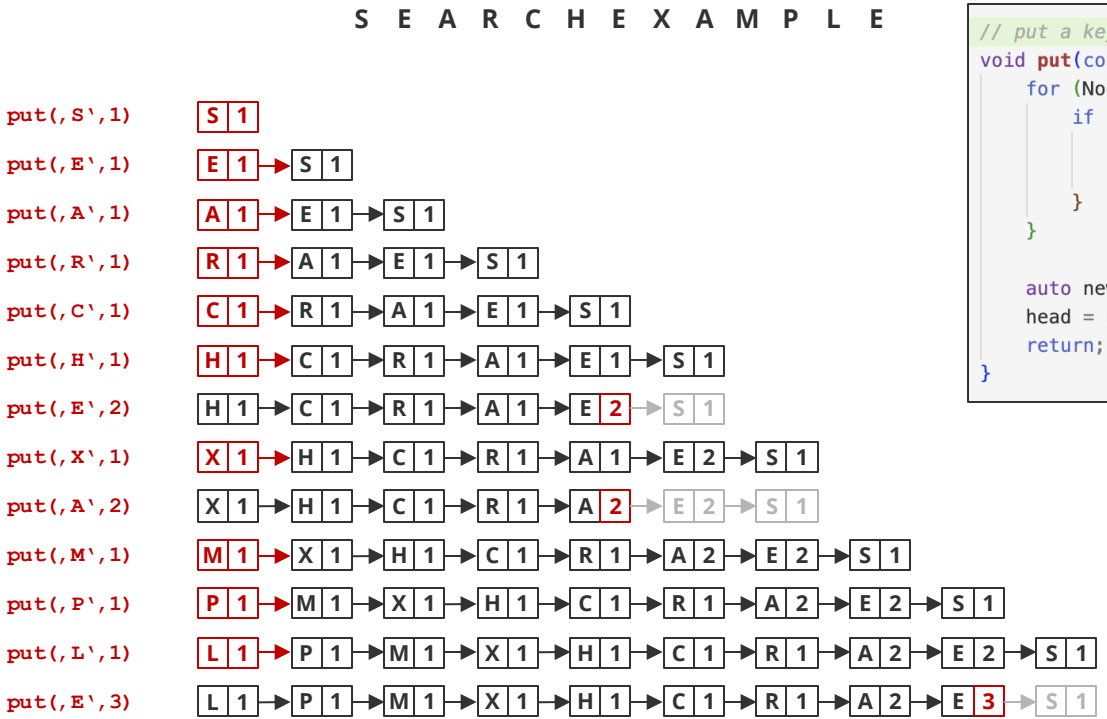
```
// put a key-value pair into the table
void put(const Key& key, const Value& val) {
    for (Node* x = head; x != nullptr; x = x->next) {
        if (x->key == key) {
            x->val = val;
            return;
        }
    }

    auto new_head = new Node(key, val, head);
    head = new_head;
    return;
}
```

Get(k)

```
// gets a value for a given key
const Value* get(const Key& key) const {
    for (Node* x = head; x != nullptr; x = x->next) {
        if (x->key == key) {
            return &(x->val);
        }
    }
    return (nullptr);
}
```

Sequentielle Suche: Ausführung




```
// put a key-value pair into the table
void put(const Key& key, const Value& val) {
    for (Node* x = head; x != nullptr; x = x->next) {
        if (x->key == key) {
            x->val = val;
            return;
        }
    }

    auto new_head = new Node(key, val, head);
    head = new_head;
    return;
}
```

Sequentielle Suche: Laufzeitanalyse

Implementierung	Garantiert			*Average*		
	Suche (`get`)	Einfügen (`put`)	Löschen (`remove`)	Suche (`get`)	Einfügen (`put`)	Löschen (`remove`)
Sequentielle Suche	$\sim n$	$\sim n$	$\sim n$	$\sim \frac{n}{2}$	$\sim n$	$\sim \frac{n}{2}$

```
● ➜  unit6 git:(main) ✗ time ./freq_counter_seq_search 1 < ../data/tinyTale.txt
  it 10
  distinct = 20
  words    = 60
  ./freq_counter_seq_search 1 < ../data/tinyTale.txt  0.00s user 0.00s system 70% cpu 0.004 total
● ➜  unit6 git:(main) ✗ time ./freq_counter_seq_search 8 < ../data/tale.txt
  business 122
  distinct = 5131
  words    = 14350
  ./freq_counter_seq_search 8 < ../data/tale.txt  2.35s user 0.00s system 98% cpu 2.376 total
● ➜  unit6 git:(main) ✗ time ./freq_counter_seq_search 10 < ../data/leipzig100K.txt
  government 2549
  distinct = 38468
  words    = 160340
  ./freq_counter_seq_search 10 < ../data/leipzig100K.txt  233.75s user 0.13s system 98% cpu 3:57.23 total
```

Überblick

1. Symboltabellen
2. Sequentielle Suche
3. **Binäre Suche**
4. Fibonacci Suche

Binäre Suche

- **Idee**: Wenn wir eine sortierte Liste von Schlüsseln haben, dann können wir immer das mittelste Element überprüfen und so den Suchraum halbieren!
  - **Datenstruktur**: Ein sortiertes Array von Schlüsseln und Werten, da beliebiger Zugriff $\mathcal{O}(1)$!
  - **Hilfsfunktion** $\mathbf{rank}(k)$: Gibt zurück, wie viele Schlüssel kleiner als k sind

```int
// returns the rank of the key in the table
int rank(const Key& key) const {
    int lo = 0, hi = n - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (key < keys[mid])
            hi = mid - 1;
        else if (key > keys[mid])
            lo = mid + 1;
        else
            return mid;
    }
    return lo;
}
```

	0	1	2	3	4	5	6	7	8	9
	A	C	E	H	L	M	P	R	S	X
Suche nach P										
lo	hi	mid								
0	9	4	A	C	E	H	**L**	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	**R**	S	X
5	6	5	A	C	E	H	L	**M**	P	R	S	X
6	6	6	A	C	E	H	L	M	**P**	R	S	X
Suche nach Q										
lo	hi	mid								
0	9	4	A	C	E	H	**L**	M	P	R	S	X
5	9	7	A	C	E	H	L	M	P	**R**	S	X
5	6	5	A	C	E	H	L	**M**	P	R	S	X
6	6	6	A	C	E	H	L	M	**P**	R	S	X
7	6		A	C	E	H	L	M	P	R	S	X

```int
// gets a value for a given key
const Value* get(const Key& key) const {
    if (is_empty())
        return (nullptr);

    int i = rank(key);

    if (i < n && keys[i] == key)
        return (&(vals[i]));
    return (nullptr);
}
```

Binäre Suche: Einfügen

- Beim Einfügen findet **`rank`** die Position wo das neue Element eingefügt wird

```
// put a key-value pair into the table
void put(const Key& key, const Value& val) {
    int i = rank(key);

    // key is already in table
    if (i < n && keys[i] == key) {
        vals[i] = val;
        return;
    }

    // insert new key-value pair
    if (n == length) resize(2*length);

    for (int j = n; j > i; j--)  {
        keys[j] = keys[j-1];
        vals[j] = vals[j-1];
    }
    keys[i] = key;
    vals[i] = val;
    n++;

    return;
}
```

	keys[]										vals[]									
	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
put(,S',1)	S										1									
put(,E',1)	E	S									1	1								
put(,A',1)	A	E	S								1	1	1							
put(,R',1)	A	E	R	S							1	1	1	1						
put(,C',1)	A	C	E	R	S						1	1	1	1	1					
put(,H',1)	A	C	E	H	R	S					1	1	1	1	1	1				
put(,E',2)	A	C	E	H	R	S					1	1	2	1	1	1				
put(,X',1)	A	C	E	H	R	S	X				1	1	2	1	1	1	1			
put(,A',2)	A	C	E	H	R	S	X				2	1	2	1	1	1	1			
put(,M',1)	A	C	E	H	M	R	S	X			2	1	2	1	1	1	1	1		
put(,P',1)	A	C	E	H	M	P	R	S	X		2	1	2	1	1	1	1	1	1	
put(,L',1)	A	C	E	H	L	M	P	R	S	X	2	1	2	1	1	1	1	1	1	1
put(,E',3)	A	C	E	H	L	M	P	R	S	X	2	1	3	1	1	1	1	1	1	1

Binäre Suche: Laufzeitanalyse

Implementierung	Garantiert			*Average*		
	Suche (`get`)	Einfügen (`put`)	Löschen (`remove`)	Suche (`get`)	Einfügen (`put`)	Löschen (`remove`)
Sequentielle Suche	$\sim n$	$\sim n$	$\sim n$	$\sim \frac{n}{2}$	$\sim n$	$\sim \frac{n}{2}$
Binäre Suche	$\sim \log_2(n)$	$\sim n$	$\sim n$	$\sim \log_2(n)$	$\sim n$	$\sim \frac{n}{2}$

```
● → unit6 git:(main) ✗ time ./freq_counter_binary_search 1 < ../data/tinyTale.txt
  it 10
  distinct = 20
  words    = 60
  ./freq_counter_binary_search 1 < ../data/tinyTale.txt  0.00s user 0.00s system 69% cpu 0.004 total
● → unit6 git:(main) ✗ time ./freq_counter_binary_search 8 < ../data/tale.txt
  business 122
  distinct = 5131
  words    = 14350
  ./freq_counter_binary_search 8 < ../data/tale.txt  0.18s user 0.00s system 98% cpu 0.190 total
● → unit6 git:(main) ✗ time ./freq_counter_binary_search 10 < ../data/leipzig100K.txt
  government 2549
  distinct = 38468
  words    = 160340
  ./freq_counter_binary_search 10 < ../data/leipzig100K.txt  8.27s user 0.01s system 99% cpu 8.320 total
```

Überblick

1. Symboltabellen
2. Sequentielle Suche
3. Binäre Suche
4. **Fibonacci Suche**

Fibonacci Suche

- Bei binärer Suche wird das Suchintervall in jedem Schritt halbiert
  - Erfordert Division um den Mittelpunkt(index) zu bestimmen
  - Division ist 10x langsamer als Addition und Subtraktion!
- **Idee**: Wir benutzen eine exponentielle Zerlegung, die nur Addition benötigt!
- **Fibonacci-Zahlen**:

$$F(n) = F(n-1) + F(n-2) \text{ wobei } F(0) = 0 \text{ und } F(1) = 1$$

  - **Beobachtung 1**: Aus zwei aufeinanderfolgenden Fibonacci-Zahlen kann immer der Vorgänger und Nachfolger durch Addition und Subtraktion berechnet werden

$$F(n) = F(n-1) + F(n-2) \quad \leftarrow \text{Nachfolger}$$
$$F(n-2) = F(n) - F(n-1) \quad \leftarrow \text{Vorgänger}$$

  - **Beobachtung 2**: Die Fibonacci-Zahlen erfüllen (approximativ) diese Beziehung

$$F(n-1) \approx 2/3 \cdot F(n)$$
$$F(n-2) \approx 1/3 \cdot F(n)$$

Fibonacci Suche

```java
// returns the rank of the key in the table
int rank(const Key& key) const {
    if (is_empty()) return (0);

    int fib2 = 0, fib1 = 1, fib = fib1 + fib2;

    // determine the smallest Fibonacci number larger or equal to n
    while (fib < n) {
        fib2 = fib1; fib1 = fib; fib = fib1 + fib2;
    }

    // marks the eliminated range from the front
    int offset = 0;

    // if there are still elements to check
    while (fib > 1) {
        // checks that the offset + fib2 is a valid index
        int i = (offset + fib2 < n - 1) ? offset + fib2 : n - 1;

        // if the key is larger than the keys[i]
        if (key > keys[i]) {
            fib = fib1; fib1 = fib2; fib2 = fib - fib1;
            offset = i;
        }
        // else if the key is smaller than the keys[i]
        else if (key < keys[i]) {
            fib = fib2; fib1 = fib1 - fib2; fib2 = fib - fib1;
        }
        // else return the rank
        else return (i);
    }

    return ((key > keys[offset]) ? offset + 1: offset);
}
```

Bestimmt die kleinste Fibonacci-Zahl die größer gleich n ist

`fib = 13`

	0	1	2	3	4	5	6	7	8	9
	A	C	E	H	L	M	P	R	S	X

Suche nach P

fib2	fib1	offset									
5	8	0	A	C	E	H	L	**M**	P	R	S	X
3	5	5	A	C	E	H	L	M	P	R	**S**	X
1	2	5	A	C	E	H	L	M	**P**	R	S	X

Suche nach Q

fib2	fib1	offset									
5	8	0	A	C	E	H	L	**M**	P	R	S	X
3	5	5	A	C	E	H	L	M	P	R	**S**	X
1	2	5	A	C	E	H	L	M	**P**	R	S	X
1	1	6	A	C	E	H	L	M	P	**R**	S	X
0	1	6	A	C	E	H	L	M	**P**	R	S	X

Binäre Suche: Laufzeitanalyse

Implementierung	Garantiert			*Average*		
	Suche (`get`)	Einfügen (`put`)	Löschen (`remove`)	Suche (`get`)	Einfügen (`put`)	Löschen (`remove`)
Sequentielle Suche	$\sim n$	$\sim n$	$\sim n$	$\sim \frac{n}{2}$	$\sim n$	$\sim \frac{n}{2}$
Binäre Suche	$\sim \log_2(n)$	$\sim n$	$\sim n$	$\sim \log_2(n)$	$\sim n$	$\sim \frac{n}{2}$
Fibonacci Suche	$\sim \log_{1.62}(n)$	$\sim n$	$\sim n$	$\sim \log_{1.62}(n)$	$\sim n$	$\sim \frac{n}{1.62}$

```
● → unit6 git:(main) ✗ time ./freq_counter_fib_search 1 < ../data/tinyTale.txt
  it 10
  distinct = 20
  words    = 60
  ./freq_counter_fib_search 1 < ../data/tinyTale.txt  0.00s user 0.00s system 4% cpu 0.069 total
● → unit6 git:(main) ✗ time ./freq_counter_fib_search 8 < ../data/tale.txt
  business 122
  distinct = 5131
  words    = 14350
  ./freq_counter_fib_search 8 < ../data/tale.txt  0.19s user 0.00s system 98% cpu 0.195 total
● → unit6 git:(main) ✗ time ./freq_counter_fib_search 10 < ../data/leipzig100K.txt
  government 2549
  distinct = 38468
  words    = 160340
  ./freq_counter_fib_search 10 < ../data/leipzig100K.txt  8.30s user 0.01s system 99% cpu 8.345 total
```

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Viel Spaß bis zur nächsten Vorlesung!