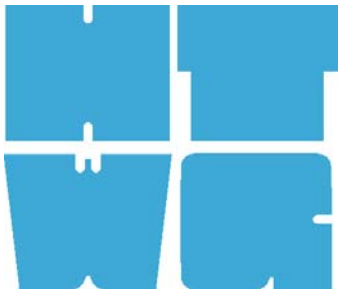


HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Algorithmen und Datenstrukturen WS 2007/2008

Fakultät Informatik
Software-Engineering und Technische Informatik

Prof. Dr. Matthias Franz
mfranz@htwg-konstanz.de
www-home.htwg-konstanz.de/~mfranz/heim.html

nach Folien von Prof. Dr. O.Bittel

Inhaltsverzeichnis (1)

Teil I: Suchen

Inhaltsverzeichnis (2)

Teil II: Graphenalgorithmen

Inhaltsverzeichnis (3)

Teil III: Suchen in Texten

- **Suchen in dynamischen Texten**
  - Knuth-Morris-Pratt-Algorithmus
  - Boyer-Moore-Algorithmus
- **Approximative Suche**
- **Suchen in statischen Texten**
  - Indexierungsverfahren

Teil IV: Graphische Algorithmen

- **Geometrische Datenstrukturen**
- **Distanzprobleme**

Literaturverzeichnis (1)

Standardwerke zu Algorithmen und Datenstrukturen

- **Mark Allen Weiss, *Data Structures and Algorithm Analysis in C++*,**
  Addison-Wesley, 586 Seiten, 2006.

  Deckt Vorlesungsstoff sehr gut ab. Gibt es auch für Java.

- **T. Ottmann und P. Widmayer, *Algorithmen und Datenstrukturen*,**
  Spektrum Akademischer Verlag, 2002.

  Standardwerk; umfangreiche Sammlung von Algorithmen u. Datenstrukturen in Pseudo-Code; zusätzlich Implementierungen in Java;
  Geht über Vorlesungsstoff hinaus. Als Nachschlagewerk sehr empfehlenswert.

- **R. Sedgewick, *Algorithmen in C*,**
  Pearson-Verlag, 2005.

  Standardwerk; umfangreiche Sammlung von Algorithmen u. Datenstrukturen;
  Geht über Vorlesungsstoff hinaus. Die C++-Variante ist leider nicht mehr verfügbar.

  Achtung: Die Neuauflage *Algorithmen in C++: Teil 1-4* behandelt im wesentlichen Suchen und Sortieren und deckt Vorlesungsstoff nur zum Teil ab und ist daher nicht empfehlenswert.

- **Cormen, Leiserson, Rivest und Stein; *Algorithmen – Eine Einführung;***
  Oldenbourg-Verlag, 2004.

  Standardwerk; umfangreiche Sammlung von Algorithmen u. Datenstrukturen;
  Geht über Vorlesungsstoff hinaus.

Literaturverzeichnis (2)

Weiterführende Literatur

- **V. Turau, *Algorithmische Graphentheorie,***
  Oldenbourg-Verlag, 2004

  Algorithmen sind im vorbildlichem Pseudo-Code geschrieben.
  Unbedingt anschauen, falls man selbst fortgeschrittene Algorithmen für Graphen zu entwickeln hat.

- **D. Gusfield, *Algorithms on Strings, Trees and Sequences;***
  Cambridge University , 1997.

  Vertiefung der Algorithmen für Suchen in Texten unter besonderer Berücksichtigung moderner molekulargenetischer Problemstellungen.

Teil 1:
Suchen

- Problemstellung
- Elementare Suchverfahren
  - Lineare Suche
  - Binäre Suche
  - Klasse Dictionary
- Hashverfahren
- Binäre Suchbäume
- Ausgeglichene Bäume
- B-Bäume
- Digitale Suchbäume
- Heaps

Problemstellung (1)

Ziel:

Zur effizienten Verwaltung einer (dynamischen) Menge von Datensätzen sind geeignete Datenstrukturen und Algorithmen gesucht.

Datensätze (records)

bestehen aus

- Suchschlüssel (search key) und
- Nutzdaten (value).

Beispiel Telefonbuch:

Maier	Hauptstr. 1	14234
Baier	Bahnhofstr. 5	24829
Müller	Uferweg 5	53289
Anton	Mozartstr. 2	36478

Datensatz

Suchschlüssel

Nutzdaten

Beispiel Wörterbuch:

sehen	see; look
sprechen	speak; talk
gehen	go; walk
hören	hear; listen

Problemstellung (2)

Operationen:

- **search(key)**
  suche nach Datensatz mit Schlüssel key.
- **insert(key, value)**
  füge neuen Datensatz mit Schlüssel key und Daten value ein.
- **remove(key)**
  Lösche Datensatz mit Schlüssel key.

Datenstrukturen mit diesen Operationen werden in der Literatur auch als **Dictionaries** bezeichnet.

Eventuell zusätzliche Operationen:

- **Traversieren:**
  Gehe über alle Datensätze nach Schlüssel sortiert.
- **Union:**
  Vereinige zwei Mengen von Datensätzen.

Problemstellung (3)

Eindeutigkeit der Schlüssel:

Die Zuordnung von Schlüssel zu einem Datensatz muss nicht eindeutig sein. Es können also mehrere Datensätze zu einem Schlüssel existieren.

Die search-Operation liefert dann mehrere Datensätze zu einem Schlüssel zurück.

Die Forderung der Eindeutigkeit der Schlüssel hängt von der Anwendung ab.

Beispiele:

- Telefonbuch: Schlüssel sind nicht eindeutig
- Wörterbuch: Schlüssel sind eindeutig

Map und Multimap (STL)

Für einfache und effiziente Verwaltungsaufgaben von Datensätze gibt es in der STL schon vorgefertigte Bausteine, die außerdem das sortierte Traversieren gestatten (siehe Prog2):

- Map: Schlüssel müssen eindeutig sein
- Multimap: Schlüssel müssen nicht eindeutig sein

Hier:

Um die Darstellung der Beispiele und Algorithmen einfach zu halten:

- Schlüssel sind eindeutig.
- Im wesentlichen nur int-Werte als Schlüssel. Keine Nutzdaten.

Teil 1:
Suchen

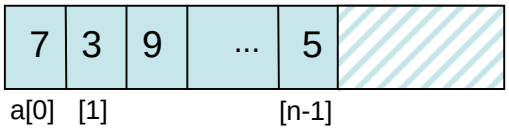
- Problemstellung
- Elementare Suchverfahren
  - Lineare Suche
  - Binäre Suche
  - Klasse Dictionary
- Hashverfahren
- Binäre Suchbäume
- Ausgeglichene Bäume
- B-Bäume
- Digitale Suchbäume
- Heaps

Sequentielle Suche (1)

Datenstruktur:

Halte Datensätze in einem Feld (lückenlos und unsortiert).

(Alternative: linear verkettete Liste.)



Algorithmen:

```
search(int k) {
      for (int i = 0; i < n: i++)
            if (a[i].key == k)
                  breche ab, da k gefunden;
      nicht gefunden;
}
```

```
remove(int k) {
      if (k kommt in a vor) {
            lösche k und schließe Lücke;
            n--;
      }
}
```

```
insert(int k) {
      if (k kommt nicht in a vor) {
            if (a vollständig gefüllt)
                  vergrößere Feld a;
            a[n++] = k;
      }
}
```

Sequentielle Suche (2)

Laufzeit (Worst-Case):

Operation	**T(n)**
search *)	O(n)
insert *)	O(n) **)
remove *)	O(n)
Aufbau einer Menge mit n Elementen (d.h. n insert-Aufrufe)	$O(n^2)$
n search-Aufrufe *)	$O(n^2)$

*) bei einer Menge mit n Elementen.

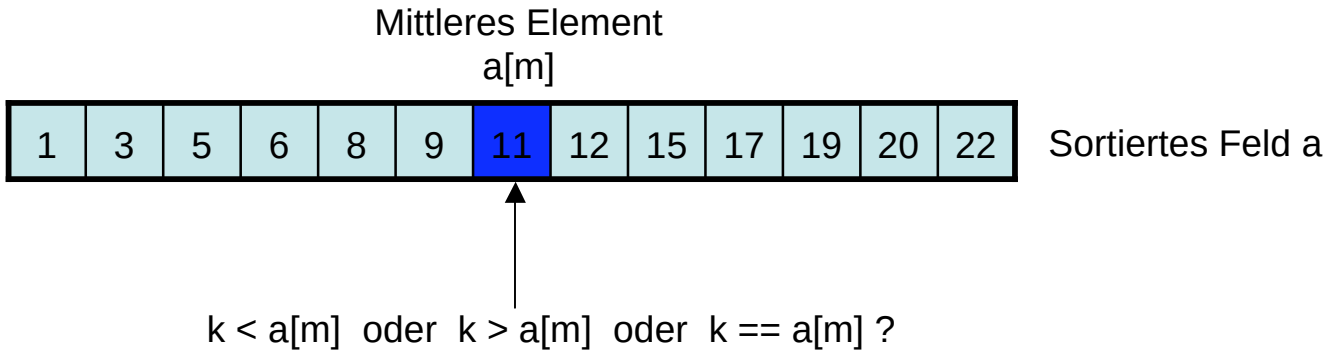
**) Es muss geprüft werden, ob das Element bereits vorkommt (search-Aufruf)

Binäre Suche (1)

Datenstruktur:

Halte Datensätze lückenlos in einem sortierten Feld.

Idee für Such-Algorithmus:



Falls k == a[m], dann gefunden.

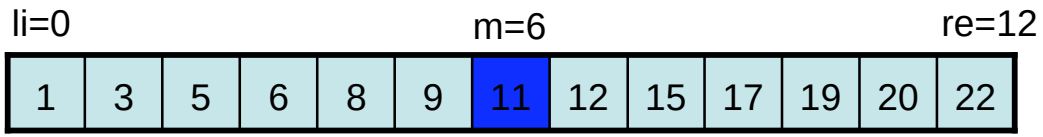
Falls k < a[m], dann suche in linker Hälfte weiter.

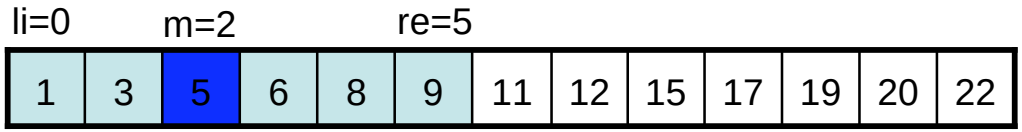
Falls k > a[m], dann suche in rechter Hälfte weiter

Binäre Suche (2)

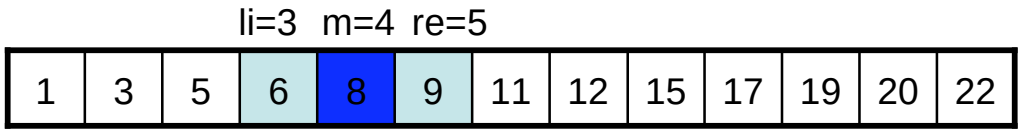
Beispiel:

Suche nach k = 8:





Suche in linker Hälfte



Suche in rechter Hälfte;
k wird gefunden!

Zu durchsuchender Bereich geht von a[li] bis a[re]

Mittleres Element m = (li + re)/2

Binäre Suche (3)

Algorithmen:

```
search(int k) {
      int li =0;
      int re = n-1;

      while (re >= li) {
            int m = (li + re)/2;
            if (k == a[m].key)
                  breche ab, da k gefunden;
            else if (k < a[m].key)
                  re = m – 1;
            else
                  li = m + 1;
      }

      k wurde nicht gefunden;
}
```

Suche weiter in linker Hälfte

Suche weiter in rechter Hälfte

Laufzeit von search:

$T(n) = O(\log n)$

In jedem Schleifendurchlauf wird das zu durchsuchende Teilfeld in etwa halbiert.

```
insert(int k)
// füge in sortiertes Feld ein
```

```
remove(int k)
// wie zuvor
```

Binäre Suche (4)

Laufzeit (Worst-Case):

Operation	**T(n)**
search [1]	O(log n)
insert [1]	O(n)
remove [1]	O(n)
Aufbau einer Menge mit n Elementen (d.h. n insert-Aufrufe)	$O(n^2)$ [2]
n search-Aufrufe [1]	O(n log n)

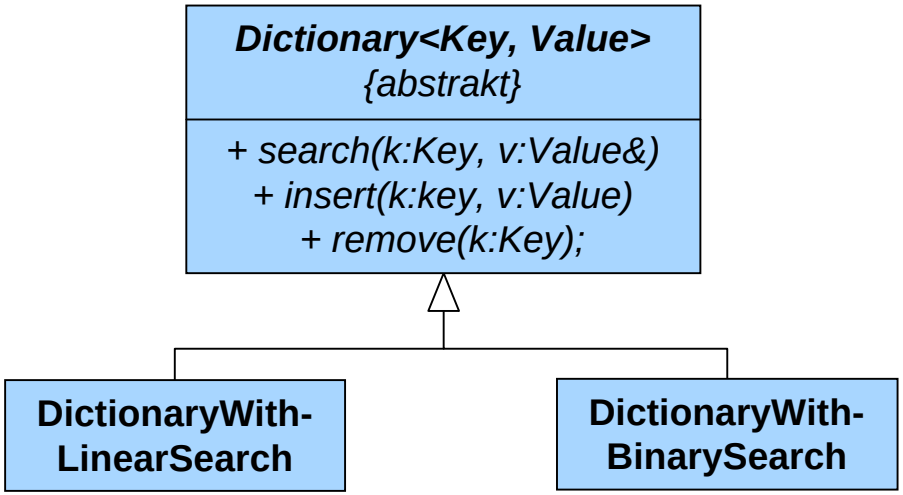
[1] Bei einer Menge mit n Elementen.

[2] Alternativ lassen sich auch n Elemente unsortiert einfügen und dann mit einem schnellen Sortierverfahren sortieren.
Damit würde man T(n) = O(log n) erreichen.

Klasse Dictionary

Konzept:

- Definiere Dictionary als abstrakte Basisklasse.
  Damit wird die Schnittstelle festgelegt.
- Schreibe Implementierungen als abgeleitete Klassen.



Abstrakte Klasse Dictionary in C++

Dictionary.h

```
template <class KeyType, class ValueType>
// KeyType erfordert Vergleichsoperator ==.
// KeyType und ValueType erfordern Zuweisungsoperator =.
class Dictionary
{
public:
      virtual ~Dictionary() { }

      virtual bool search(const KeyType& k, ValueType& v) const = 0;
      // Sucht Schlüssel k und liefert Wert v falls gefunden.
      // Gibt true zurück, falls Schlüssel k gefunden wird, und sonst false.

      virtual bool insert(const KeyType& k, const ValueType& v) = 0;
      // Fügt neuen Schlüssel k mit Wert v ein. Falls Schlüssel bereits vorhanden ist,
      // wird nicht eingefügt und false zurückgeliefert, sonst true.

      virtual bool remove(const KeyType& k) = 0;
      // Löscht Schlüssel k.  Falls Schlüssel gelöscht werden konnte, wird true
      // zurückgeliefert und sonst false (d.h. Schlüssel war nicht vorhanden).

      virtual int getNumber() const = 0;
      // liefert Anzahl der Eintraege.
};
```

Virtueller Destruktor.

Rein virtuelle Methoden

Klasse DictionaryWithLinearSearch in C++ (1)

DictionaryWithLinearSearch.h

```
template <class KeyType, class ValueType>
class DictionaryWithLinearSearch : public Dictionary<KeyType,ValueType>
{
public:
	DictionaryWithLinearSearch(int n = 100); // Größe des Feldes
	virtual ~DictionaryWithLinearSearch ( );
	virtual bool search(const KeyType& k, ValueType& v) const;
	virtual bool insert(const KeyType& k, const ValueType& v);
	virtual bool remove(const KeyType& k);
	virtual int getNumber() const {return number;}

private:
	struct Entry
	{
		KeyType key;
		ValueType value;
	};
	Entry** a;
	int size;
	int number;
};
```

Dynamisches Feld mit Zeigern auf Einträgen.

Damit werden Verschiebungen im Feld effizienter.

Klasse DictionaryWithLinearSearch in C++ (2)

DictionaryWithLinearSearch.cpp

```
template <class KeyType, class ValueType>
DictionaryWithLinearSearch <KeyType, ValueType>::DictionaryWithLinearSearch(int n) {
        number = 0;
        size = n;
        a = new Entry*[size];
}

template <class KeyType, class ValueType>
DictionaryWithLinearSearch <KeyType, ValueType>::~DictionaryWithLinearSearch() {
        for (int i = 0; i <number; i++)
                delete a[i];
        delete [ ] a;
}

template <class KeyType, class ValueType>
bool DictionaryWithLinearSearch <KeyType, ValueType>
::search(const KeyType& k, ValueType& v) const {
        for (int i = 0; i < number; i++)
                if (a[i]->key == k) {
                        v = a[i]->value;
                        return true;    // gefunden;
                }
        return false;                   // nicht gefunden;
}
```

Klasse DictionaryWithLinearSearch in C++ (3)

DictionaryWithLinearSearch.cpp

```
template <class KeyType, class ValueType>
bool DictionaryWithLinearSearch<KeyType, ValueType>
::insert(const KeyType& k, const ValueType& v)
{
      // Key k darf nicht mehrfach vorkommen:
      if (search(k) == true)
            return false;

      if (number >= size) {
            size *= 2;
            Eintrag** aNeu = new Entry*[size];
            for (int i = 0; i < number; i++)
                  aNeu[i] = a[i];
            delete [ ] a;
            a = aNeu;
      }

      a[number] = new Entry;
      a[number]->value = v;
      a[number]->key = k;
      number++;
      return true;
}
```

Feld bei Bedarf verdoppeln.

Klasse DictionaryWithLinearSearch in C++ (4)

DictionaryWithLinearSearch.cpp

```
template <class KeyType, class ValueType>
bool DictionaryWithLinearSearch<KeyType, ValueType>
::remove(const KeyType& k)
{
	int i;
	for (i = 0; i < number; i++)
		if (a[i]->key == k)
			break;
	if (i == number)
		return false;   // nicht gefunden;

	delete a[i];
	for (int j = i; j < number-1; j++)
		a[j] = a[j+1];
	number--;
	return true;
}
```

Klasse DictionaryWithBinarySearch in C++ (1)

DictionaryWithBinarySearch.h

```
template <class KeyType, class ValueType>
// KeyType erfordert zusätzlich Vergleichsoperator <.
class DictionarWithBinarySearch : public Dictionary<KeyType,ValueType>
{
public:
	DictionarWithBinarySearch(int n = 100); // Größe des Feldes
	virtual ~DictionaryWithBinarySearch();
	virtual bool search(const KeyType& k, ValueType& v) const;
	virtual bool insert(const KeyType& k, const ValueType& v);
	virtual bool remove(const KeyType& k);
	virtual int getNumber() const {return number;}
private:
	struct Entry
	{
		KeyType key;
		ValueType value;
	};
	Entry** a;
	int size;
	int number;
};
```

Klasse DictionaryWithBinarySearch in C++ (2)

DictionaryWithBinarySearch.cpp

```
template <class KeyType, class ValueType>
bool DictionarWithBinarySearch<KeyType, ValueType>
::search(const KeyType& k, ValueType& v) const
{
	int li =0;
	int re = n-1;

	while (re >= li) {
		int m = (li + re)/2;
		KeyType km = a[m]->key;
		if (km == k) {
			v = a[m]->value;
			return true;
		} else if (k < v)
			re = m – 1;
		else
			li = m + 1;
	}
	return false;
}
```

Klasse DictionaryWithBinarySearch in C++ (3)

DictionaryWithBinarySearch.cpp

```
template <class KeyType, class ValueType>
bool DictionarWithBinarySearch<KeyType, ValueType>
::insert(const KeyType& k, ValueType& v)   {
      if (search(k) == true)
            return false;
      if (number >= size) {
            size *= 2;
            Eintrag** aNeu = new Entry*[size];
            for (int i = 0; i < number; i++)
                  aNeu[i] = a[i];
            delete [ ] a;
            a = aNeu;
      }
      int j = number - 1;
      while (j >= 0 && a[j]->key > k) {
            a[j+1] = a[j];
            j--;
      }
      a[j+1] = new Entry;
      a[j+1]->value = v;
      a[j+1]-> key = k;
      number++;
      return true;
}
```

Einfügen in sortiertes Feld.

Konstruktor und Destruktor wie bei DictionarWithLinearSearch .

remove wie bei DictionaryWithLinearSearch jedoch mit binärer statt linearer Suche

Anwendung der Klasse Dictionary

Main.cpp

```
int main() {
	Dictionary<string,string>* dict;

	// Implementierung auswählen
	int eingabe; cin >> eingabe;
	switch (eingabe) {
	case 1:
		dict = new DictionarWithBinarySearch<string,string>;
		break;
	case 2:
		dict = new DictionaryWithLinearSearch<string,string>;
		break;
	default:
		return 0;
	}
	dict->insert("Hund", "dog");
	dict->insert("Katze", "cat");
	dict->insert("Tier", "animal");
	string s;
	dict->search("Hund", s);  cout << s << endl;
	return 0;
}
```

Wörterbuch
Deutsch - Englisch