Vorlesung Datenstrukturen

Sommersemester 2015

Technische Universität Chemnitz

Dr. Frank Seifert

fsei@cs.tu-chemnitz.de

Straße der Nationen 62 - Büro 336g

(Termin nach Vereinbarung)

Vorlesungstermine

Donnerstags:	13:30 - 15:00 Uhr	1/201
Vorlesungsfrei:	14. Mai	
Freitags:	11:30 - 13:00 Uhr	1/201
Vorlesungsfrei:	1. Mai	
	15. Mai	

Übungen

Übungsbeginn

Zweite Vorlesungswoche (ab 13. April 2015)

Termine

Montag:	17:15 - 18:45 Uhr	2/N005	(Pönisch)
Montag:	17:15 - 18:45 Uhr	1/375	(Naumann)
Mittwoch:	11:30 - 13:00 Uhr	1/208a	(Fliege)
Donnerstag:	09:30 - 11:00 Uhr	1/208a	(Pönisch)

Materialien

Vorlesungsmaterialien

Im Internet stehen Kopien der Vorlesungsfolien zum Download bereit

Wo?

Homepage des Lehrstuhl Datenverwaltungssysteme: www.tu-chemnitz.de/informatik/dvs

Aktualität

Folien stehen vor der jeweiligen Vorlesung zur Verfügung, können aber im Semesterverlauf aktualisiert oder ergänzt werden.

Plenarübung

Was?

Um Ihnen mehr Gelegenheit zum Üben des Stoffes zu geben, werden in einigen Vorlesungseinheiten anstelle der Vorlesung Übungen zu einem Schwerpunktthema durchgeführt (siehe Vorlesung Algorithmen & Programmierung).

Wann?

Konkrete Termine werden in der Vorlesung bekanntgegeben.

Wo?

Die Plenarübung findet anstelle der Vorlesung im Vorlesungssaal statt.

Prüfungsvorleistung

Hausaufgaben

Um zur Prüfung Datenstrukturen zugelassen zu werden, ist neben einer bestandenen A&P-Klausur das Erbringen einer **Prüfungsvorleistung** nötig.

Diese Vorleistung besteht in der **Lösung von Hausaufgaben**. Es wird eine Anzahl von Hausaufgaben bereitgestellt, von denen mindestens die Hälfte **richtig** gelöst werden müssen.

Bearbeitung

In der Regel stehen jeweils zwei Wochen zur Bearbeitung der Aufgaben zur Verfügung.

Die Aufgabenstellungen, Abgabemodalitäten und der Bearbeitungszeitraum werden jeweils in der Vorlesung bekanntgegeben.

Algorithmen & Programmierung

Ergebnisse der Klausur

Im Wintersemester 2014 / 2015 haben 60% der Teilnehmer die Klausur „Algorithmen und Programmierung“ bestanden.

Note 1	Note 2	Note 3	Note 4	Note 5
25	12	20	21	53

Wiederholungsklausur

Die Klausur „Algorithmen und Programmierung“ wird **innerhalb der Vorlesungszeit** des Sommersemesters wiederholt.

Der genaue Termin wird in der Vorlesung Datenstrukturen bekannt gegeben.

Inhalt der Veranstaltung

Vorlesung

Behandlung grundlegender Methoden der **Datenorganisation** für die Programmverarbeitung.

Begriff Datenstruktur

- systematische Anordnung von Daten
- Verknüpfung von Daten
- Eigenschaften

Operationen

Aus der Struktur der Daten ergeben sich spezielle Algorithmen zum Durchmustern, Suchen, Einfügen und Löschen eines Elements.

Aus der Semantik einer Datenstruktur ergeben sich häufig noch spezielle zusätzliche Operationen.

Auswahlkriterien

Platzbedarf der Datenstruktur

Entstehen Redundanzen oder bleibt Speicherplatz ungenutzt?

Anordnung der Daten

Welche Beziehung herrscht zwischen einzelnen Datenelementen? Gibt es eine Sortierung?

Zeitbedarf für Zugriff und Verarbeitung

Elementzugriff

Suchen

Einfügen

Löschen

Durchmustern

Sortieren

Schwerpunkte der Vorlesung

Einfache dynamische Datenstrukturen

Werden vom Programmierer zur Laufzeit eines Programms erzeugt, bearbeitet und wieder gelöscht, z.B. Listen, Schlangen und Stapel

Komplexere dynamische Datenstrukturen

Behandelt mächtige und wichtige Datenstrukturen der Informatik, Bäume und Graphen in verschiedenen Ausprägungen.

Suchen

Untersuchung effizienter Wege, Informationen in Datenstrukturen zu finden.

Datenmanipulation

Was ist beim Einfügen oder Löschen von Daten bzgl. verschiedener Datenstrukturen zu beachten?

Sortieren

Es gibt viele verschiedene Sortierverfahren. Aber nur wenige arbeiten effizient. In diesem Schwerpunkt werden verschiedene Verfahren vorgestellt und bezüglich ihrer Leistungsfähigkeit untersucht.

Gliederung der Vorlesung

Komplexität

- Felder
- Listen, Stapel und Schlangen
- Abstrakter Datentyp
- Bäume
- Graphen
- Hashing
- Heaps

Konzepte objektorientierter Programmierung

Programmiersprache

Programmiersprache der Vorlesung

Die in der Vorlesung untersuchten Datenstrukturen werden mit Hilfe von C++ realisiert.

Warum C++

- enthält C als vollständige Teilmenge ➔ kompatibel
- effizient
- unterstützt Datenabstraktion (Thema „Abstrakter Datentyp“)
- objektorientiert (Thema „Konzepte objektorientierter Programmierung“)
- flexibel
- sehr weit verbreitet
- industriell äußerst bedeutend

Sieben Schritte zum Erfolg

1. Teilnahme an der Vorlesung
    - Vorlesungsinhalte nicht nur auf Folien, sondern auch Tafelbilder sowie Klausurtipps
2. Nacharbeiten des Vorlesungsstoffes und Implementierung der Beispiele am Rechner
3. Teilnahme an der Übung und Implementierung der Übungsaufgaben am Rechner
4. Selbstständiges Experimentieren
5. **Programmieren & Experimentieren!**
6. **Programmieren & Experimentieren!!**
7. **Programmieren & Experimentieren!!!**

Vorlesung Datenstrukturen

Rechenaufwand & Komplexität

Die Datenstruktur Feld

Rechenaufwand und Komplexität

Rechenaufwand

Aufwandsabschätzung

Um Datenstrukturen hinsichtlich der Effizienz ihrer Algorithmen miteinander vergleichen zu können, benötigen wir ein Maß für den rechenzeittechnischen Aufwand. Diesen Aufwand messen wir durch Zählung der Anzahl benötigter Rechenschritte.

Rechenschritt

In der Regel bezeichnet man die Durchführung einer elementaren Operation eines Algorithmus als Rechenschritt, z.B. Grundrechenoperationen, Speicherzugriffe. Es können aber durchaus auch komplexere Operationen als ein Rechenschritt angesehen werden.

Rechenzeit

Da bekannt ist, wie lange die einzelnen Schritte bei der Ausführung des Algorithmus in der Form eines Computerprogramms dauern, sprechen wir auch von der Rechenzeit.

Rechenzeit

Unterscheidung der Rechenzeit

Die Rechenzeit ist je nach verwendetem Algorithmus oft auch von der konkreten Verteilung der Daten abhängig. Deshalb unterscheidet man Rechenzeiten für

- den günstigsten Fall ➔ **best case**
- den ungünstigen Fall ➔ **worst case**
- den durchschnittlichen Fall ➔ **average case**

Berechnung der durchschnittlichen Rechenzeit

Bei Vorliegen einer Datenabhängigkeit kann die durchschnittliche Rechenzeit nur berechnet werden, wenn eine bestimmte Wahrscheinlichkeitsverteilung der Daten bekannt ist.

Komplexität eines Algorithmus

Komplexität eines Algorithmus

Da wir die Rechenzeit nicht nur für eine konkrete Datenkonstellation bestimmen wollen, versuchen wir eine Funktion *g(n)* zu bestimmen, die die Abhängigkeit der Rechenzeit von der **Anzahl der Elemente *n*** einer Datenmenge charakterisiert.

Die Elementanzahl kann z.B. durch die Länge einer Eingabefolge bzw. die Elementanzahl eines Feldes bestimmt sein.

Asymptotische Komplexität

In der Regel sind wir nicht genau an der Funktion *g(n)* interessiert, sondern mehr am Verhalten dieser Funktion.

Dahinter steckt die Überlegung, von unwesentlichen Konstanten zu abstrahieren, z.B. Technologieparametern oder Implementierungsdetails.

Beispiel

Analyse der Laufzeiten zweier Funktionen `sum1` und `sum2` zur Berechnung der Summe

$$sum(n) = \sum_{i=1}^{n} i$$

Als relevante Rechenoperationen betrachten wir die Grundrechenarten, Vergleiche und Zuweisungen (eine Inkrement- bzw. Dekrementoperation zählt als eine Rechenoperation).

```
int sum1(int n) {
  int s = 0;
  for (int i = 1; i <= n; i++)
    s = s + i;
  return s;
}
```

```
int sum2(int n) {
  int s;
  s = n * (n+1) / 2;
  return s;
}
```

O-Notation

Zweck

Allgemeine Kennzeichnung der Effizienz eines Algorithmus, unabhängig von konkreter Rechnerarchitektur, Programmiersprache oder Compiler.

Anwendung

Bestimmung des Aufwands an Rechenzeit oder / und Speicherplatz.

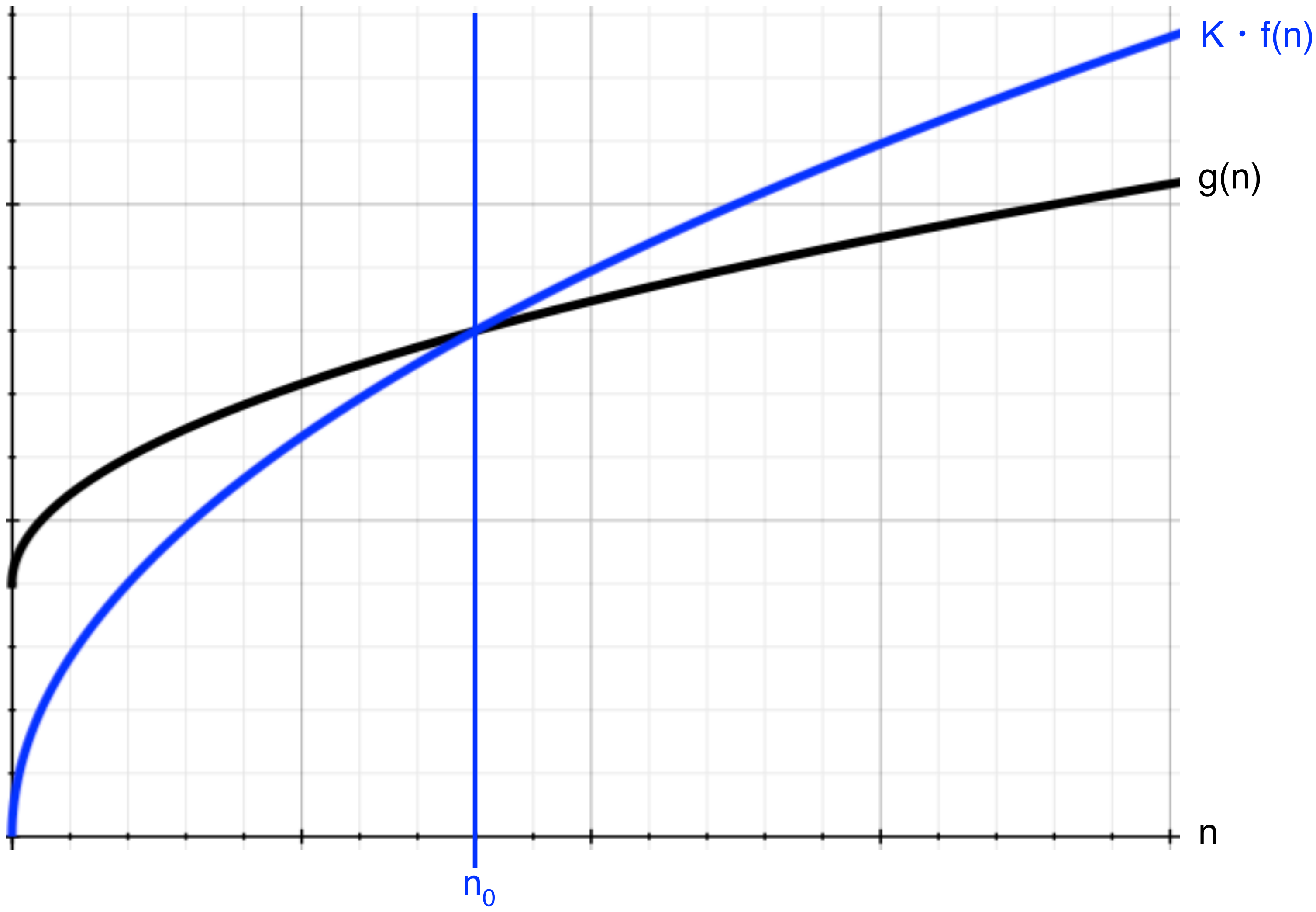
Definition

Die Komplexität eines Algorithmus ist von der Größenordnung $O(f(n))$, wenn für die tatsächliche Komplexität $g(n)$ des Algorithmus gilt: $\exists\ K, n_0 : \forall\ n \geq n_0 : g(n) \leq K\ f(n)$

Bedeutung

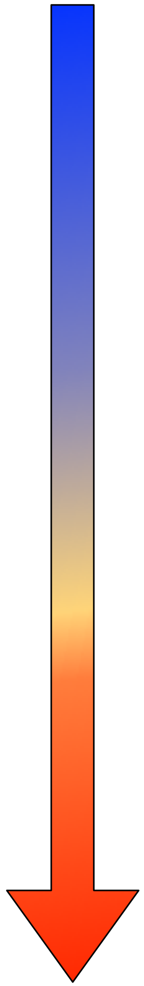
Der Algorithmus erledigt seine Aufgabe in $O(f(n))$ Schritten, d.h. die tatsächliche Anzahl ausgeführter Schritte ist nicht größer als $f(n)$ multipliziert mit einer Konstanten $K>0$.

Veranschaulichung



Typische Zeitkomplexitäten

- konstant $O(1)$
- logarithmisch $O(\log_2 n)$
- linear $O(n)$
- $n \log_2 n$ $O(n \log_2 n)$
- quadratisch $O(n^2)$
- kubisch $O(n^3)$
- exponentiell $O(2^n)$
- Fakultätsfunktion $O(n!)$
- superexponentiell $O(n^n)$

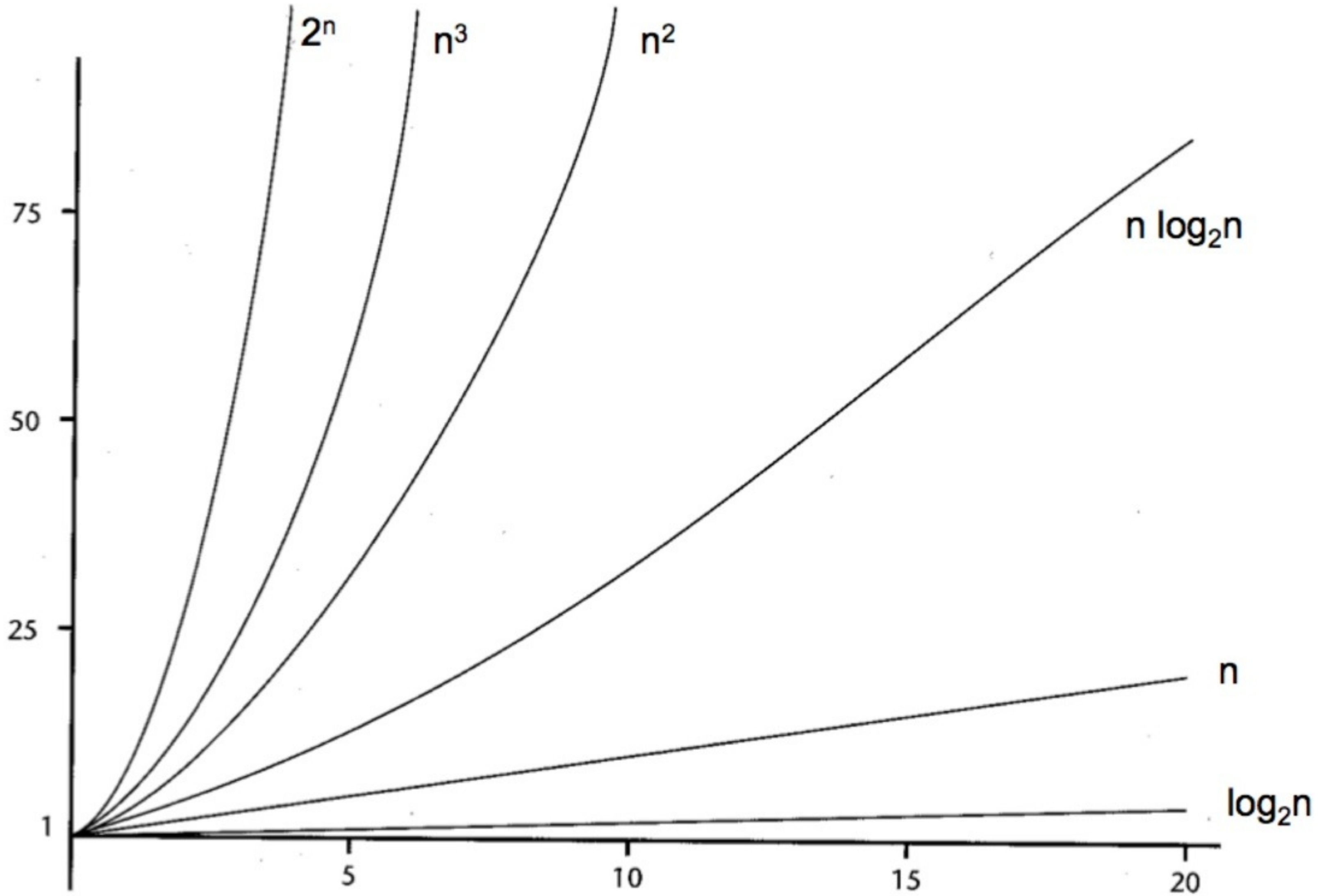


günstig

vertretbar

katastrophal

Typische Zeitkomplexitäten



[Bittel 2007]

Typische Zeitkomplexitäten

Veranschaulichung **des konkreten Zeitbedarfs** wichtiger Komplexitätsklassen unter der Annahme, dass eine Operation 10^{-6} Sekunden benötigt:

Komplexitätsklasse	n = 1 000	n = 1 000 000
$O(1)$	0,000001 sec	0,000001 sec
$O(\log_2 n)$	0,00001 sec	0,00002 sec
$O(n)$	0,001 sec	1,0 sec
$O(n \log_2 n)$	0,01 sec	20 sec
$O(n^2)$	1 sec	11.5 **Tage**
$O(n^3)$	16.6 **min**	31 000 **Jahre**
$O(2^n)$	**größer als das Alter des Weltalls**	

Felder

Eine (bekannte) Datenstruktur

Die Datenstruktur Feld

Eigenschaften

Ein Feld stellt eine **lineare Anordnung** von Elementen des selben Datentyps dar, das im Hauptspeicher durch einen **zusammenhängenden** Speicherblock realisiert wird.

Zugriff auf beliebige Feldelemente

Wir benötigen drei Kenngrößen zur Positionsbestimmung eines beliebigen Elements eines Feldes `a`:

- `i` Index des zu bestimmenden Feldelements
- `&a[0]` Adresse des ersten Feldelements (Adresse des Speicherblocks)
- `sizeof(a)` Speicherbedarf **eines** Feldelements

Konsequenz

Der Zugriff auf ein beliebiges Element ist in **konstanter** Zeit, also **O(1)** möglich.

Operationen auf Feldern

Ausgangspunkt

Unsortiertes Feld `a[]` mit n Elementen.

Durchmustern aller Elemente

Jedes Feldelement muss verarbeitet werden:

```
for (int i=0; i<n; i++) {
  process(a[i]); // Verarbeitung von a[i]
}
```

Asymptotische Komplexität

Wir sind nur an der Anzahl der Zugriffe auf Elemente des Feldes a interessiert und wollen diese in Abhängigkeit von n ausdrücken ➔ **Komplexität O(n)**

Operationen auf Feldern

Ausgangspunkt

Unsortiertes Feld `a[]` mit `n` Elementen

Operation Suche (Variante 1)

Teste, ob Element `v` in `a[]` vorkommt (Existenztest)

Anzahl Vergleichsoperationen auf Feldelementen

Minimal: 1

Maximal: n

Durchschnitt: $\frac{n+1}{2}$

Komplexität

O(n)

```
bool match = false;
for (int i=0; i<n && !match; i++) {
    if (a[i] == v) {
        match = true; // gefunden
    }
}
```

Operationen auf Feldern

Ausgangspunkt

Unsortiertes Feld `a[]` mit `n` Elementen

```
int count = 0;
for (int i=0; i<n; i++) {
    if (a[i] == v) {
        count++;
    }
}
```

Operation Suche **(Variante 2)**

Zähle, wie oft Element `v` in `a[]` vorkommt

Anzahl Vergleichsoperationen auf Feldelementen

Es werden immer n Vergleiche auf Feldelementen durchgeführt.

Komplexität

O(n)

Einfügen von Feldelementen (1)

Einfügen eines Elements

Alle Elemente ab Einfügeposition *p* müssen um eine Indexposition „nach hinten“ verschoben werden.

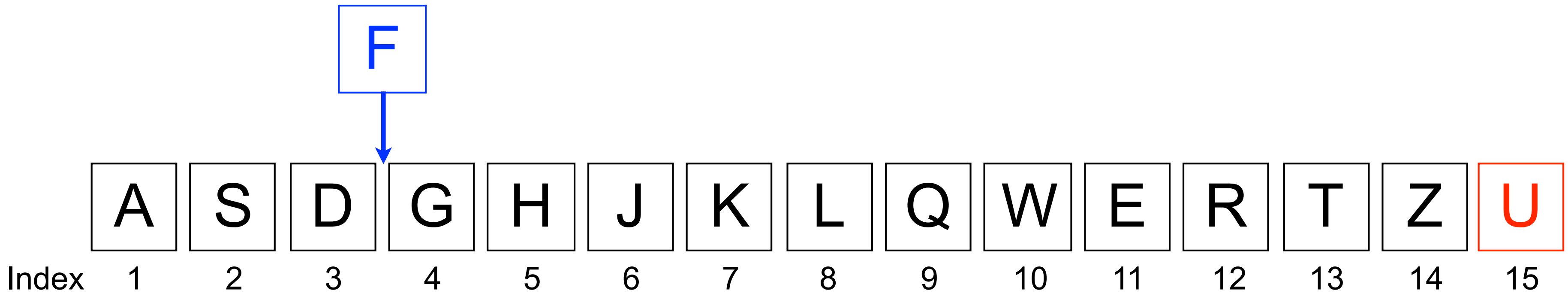
Dann kann das Element in *p* durch das einzufügende ersetzt werden.

Problem

Was passiert mit dem letzten Feldelement?

Beispiel

Feld mit 15 Elementen: Einfügen von F nach dem dritten Feldelement



Einfügen von Feldelementen (1)

Einfügen eines Elements

Alle Elemente ab Einfügeposition *p* müssen um eine Indexposition „nach hinten“ verschoben werden.

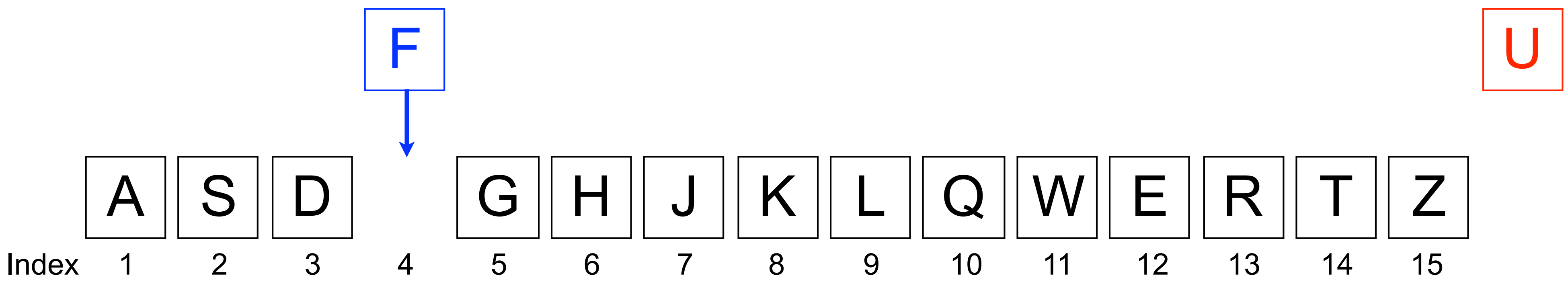
Dann kann das Element in *p* durch das einzufügende ersetzt werden.

Problem

Was passiert mit dem letzten Feldelement?

Beispiel

Feld mit 15 Elementen: Einfügen von F nach dem dritten Feldelement



Einfügen von Feldelementen (1)

Einfügen eines Elements

Alle Elemente ab Einfügeposition *p* müssen um eine Indexposition „nach hinten“ verschoben werden.

Dann kann das Element in *p* durch das einzufügende ersetzt werden.

Problem

Was passiert mit dem letzten Feldelement?

Beispiel

Feld mit 15 Elementen: Einfügen von F nach dem dritten Feldelement

?

	A	S	D	F	G	H	J	K	L	Q	W	E	R	T	Z
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Einfügen von Feldelementen (2)

Problem

Wie kann man den Verlust des letzten Feldelements verhindern? $\boxed{U} \rightarrow \boxed{Z}$

Abhilfe

Einrichtung eines Überlaufpuffers, indem das Feld schon im Vorhinein größer dimensioniert wird.

Mit Hilfe einer zusätzlichen Variablen wird die **tatsächliche** Elementanzahl des Feldes verwaltet.

Nachteil

Speicherplatz wird verschwendet und wir haben keine Garantie, dass das Feld nicht irgendwann doch noch überläuft.

Zeitkomplexität

Im ungünstigsten Fall müssen n Elemente verschoben werden ➔ **Komplexität O(n)**

Löschen von Feldelementen

Löschen eines Elements

Alle Elemente nach der Löschposition *p* müssen um eine Indexposition „nach vorn“ verschoben werden.

Problem

Am Ende des Feldes bleibt ein zunächst undefiniertes Element übrig.

Beispiel:

Feld mit 15 Elementen: Löschen von J an Position *p*=6

	A	S	D	G	H	J	K	L	Q	W	E	R	T	Z	U
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Löschen von Feldelementen

Löschen eines Elements

Alle Elemente nach der Löschposition *p* müssen um eine Indexposition „nach vorn“ verschoben werden.

Problem

Am Ende des Feldes bleibt ein zunächst undefiniertes Element übrig.

Beispiel:

Feld mit 15 Elementen: Löschen von J an Position *p*=6

							K	L	Q	W	E	R	T	Z	U
	A	S	D	G	H	J									
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Löschen von Feldelementen

Löschen eines Elements

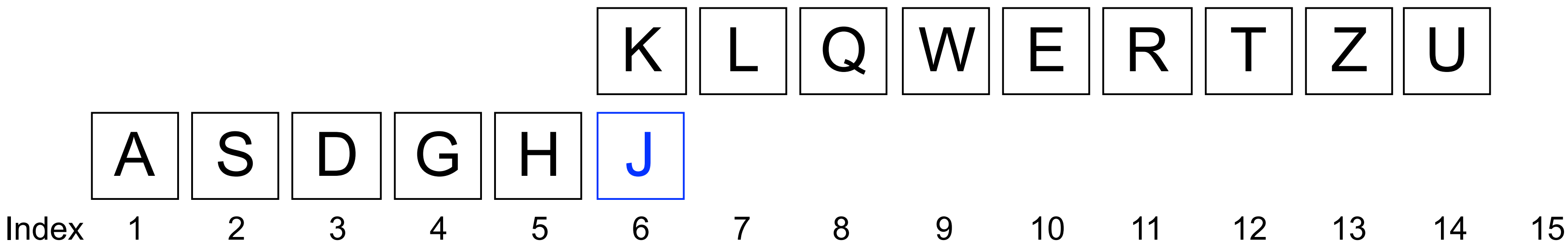
Alle Elemente nach der Löschposition *p* müssen um eine Indexposition „nach vorn“ verschoben werden.

Problem

Am Ende des Feldes bleibt ein zunächst undefiniertes Element übrig.

Beispiel:

Feld mit 15 Elementen: Löschen von J an Position *p*=6



Löschen von Feldelementen

Löschen eines Elements

Alle Elemente nach der Löschposition *p* müssen um eine Indexposition „nach vorn“ verschoben werden.

Problem

Am Ende des Feldes bleibt ein zunächst undefiniertes Element übrig.

Beispiel:

Feld mit 15 Elementen: Löschen von J an Position *p*=6

	A	S	D	G	H	K	L	Q	W	E	R	T	Z	U	
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Löschen von Feldelementen

Löschen eines Elements

Alle Elemente nach der Löschposition *p* müssen um eine Indexposition „nach vorn“ verschoben werden.

Problem

Am Ende des Feldes bleibt ein zunächst undefiniertes Element übrig.

Beispiel:

Feld mit 15 Elementen: Löschen von J an Position *p*=6

	A	S	D	G	H	K	L	Q	W	E	R	T	Z	U	?
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Löschen von Feldelementen

Problem

Wie behandelt man das neu entstandene letzte Feldelement?

Abhilfe

Analog der Einfügeoperation kann mit Hilfe einer zusätzlichen Variablen die tatsächliche Elementanzahl des Feldes verwaltet werden.

Nachteil

Mögliche Speicherplatzverschwendung.

Zeitkomplexität

Im ungünstigsten Fall müssen n–1 Elemente verschoben werden ➔ **Komplexität O(n)**.

	A	S	D	G	H	K	L	Q	W	E	R	T	Z	U	?
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Verwendung von Feldern

Beispiel

Sieb des Eratosthenes

Eratosthenes (ca. 200 v. Chr.) war ein griechischer Mathematiker und Philosoph, der u.a. einen Algorithmus zur Bestimmung von Primzahlen entwickelte:

- Schreibe alle natürlichen Zahlen von 2 bis zu einem frei wählbaren Maximalwert *M* auf. Zunächst sind alle Zahlen unmarkiert.
- Bestimme die kleinste unmarkierte Zahl *P*, deren Vielfache noch nicht markiert wurden. *P* ist immer eine Primzahl.
- Ist das Quadrat von *P* größer als der Maximalwert *M*, ergeben alle nicht markierten Zahlen die Primzahlen von 2 bis *M*
- Markiere alle Vielfachen von *P* und beginne dabei mit dem Quadrat von *P*.

Sieb des Eratosthenes

M = 36

	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

- Schreibe alle natürlichen Zahlen von 2 bis zu einem frei wählbaren Maximalwert *M* auf. Zunächst sind alle Zahlen unmarkiert.
- Bestimme die kleinste unmarkierte Zahl *P*, deren Vielfache noch nicht markiert wurden. *P* ist immer eine Primzahl.
- Ist das Quadrat von *P* größer als der Maximalwert *M*, ergeben alle nicht markierten Zahlen die Primzahlen von 2 bis *M*.
- Markiere alle Vielfachen von *P* und beginne dabei mit dem Quadrat von *P*.

Sieb des Eratosthenes

M = 36

	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

- Schreibe alle natürlichen Zahlen von 2 bis zu einem frei wählbaren Maximalwert *M* auf. Zunächst sind alle Zahlen unmarkiert.
- Bestimme die kleinste unmarkierte Zahl *P*, deren Vielfache noch nicht markiert wurden. *P* ist immer eine Primzahl.
- Ist das Quadrat von *P* größer als der Maximalwert *M*, ergeben alle nicht markierten Zahlen die Primzahlen von 2 bis *M*.
- Markiere alle Vielfachen von *P* und beginne dabei mit dem Quadrat von *P*.

Sieb des Eratosthenes

M = 36

	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

- Schreibe alle natürlichen Zahlen von 2 bis zu einem frei wählbaren Maximalwert *M* auf. Zunächst sind alle Zahlen unmarkiert.
- Bestimme die kleinste unmarkierte Zahl *P*, deren Vielfache noch nicht markiert wurden. *P* ist immer eine Primzahl.
- Ist das Quadrat von *P* größer als der Maximalwert *M*, ergeben alle nicht markierten Zahlen die Primzahlen von 2 bis *M*.
- Markiere alle Vielfachen von *P* und beginne dabei mit dem Quadrat von *P*.

Sieb des Eratosthenes

M = 36

	2	**3**	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

- Schreibe alle natürlichen Zahlen von 2 bis zu einem frei wählbaren Maximalwert *M* auf. Zunächst sind alle Zahlen unmarkiert.
- Bestimme die kleinste unmarkierte Zahl *P*, deren Vielfache noch nicht markiert wurden. *P* ist immer eine Primzahl.
- Ist das Quadrat von *P* größer als der Maximalwert *M*, ergeben alle nicht markierten Zahlen die Primzahlen von 2 bis *M*.
- Markiere alle Vielfachen von *P* und beginne dabei mit dem Quadrat von *P*.

Sieb des Eratosthenes

M = 36

	2	**3**	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

- Schreibe alle natürlichen Zahlen von 2 bis zu einem frei wählbaren Maximalwert *M* auf. Zunächst sind alle Zahlen unmarkiert.
- Bestimme die kleinste unmarkierte Zahl *P*, deren Vielfache noch nicht markiert wurden. *P* ist immer eine Primzahl.
- Ist das Quadrat von *P* größer als der Maximalwert *M*, ergeben alle nicht markierten Zahlen die Primzahlen von 2 bis *M*.
- Markiere alle Vielfachen von *P* und beginne dabei mit dem Quadrat von *P*.

Sieb des Eratosthenes

M = 36

	2	3	4	**5**	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

- Schreibe alle natürlichen Zahlen von 2 bis zu einem frei wählbaren Maximalwert *M* auf. Zunächst sind alle Zahlen unmarkiert.
- Bestimme die kleinste unmarkierte Zahl *P*, deren Vielfache noch nicht markiert wurden. *P* ist immer eine Primzahl.
- Ist das Quadrat von *P* größer als der Maximalwert *M*, ergeben alle nicht markierten Zahlen die Primzahlen von 2 bis *M*
- Markiere alle Vielfachen von *P* und beginne dabei mit dem Quadrat von *P*.

Sieb des Eratosthenes

M = 36

	2	3	4	**5**	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

- Schreibe alle natürlichen Zahlen von 2 bis zu einem frei wählbaren Maximalwert *M* auf. Zunächst sind alle Zahlen unmarkiert.
- Bestimme die kleinste unmarkierte Zahl *P*, deren Vielfache noch nicht markiert wurden. *P* ist immer eine Primzahl.
- Ist das Quadrat von *P* größer als der Maximalwert *M*, ergeben alle nicht markierten Zahlen die Primzahlen von 2 bis *M*.
- Markiere alle Vielfachen von *P* und beginne dabei mit dem Quadrat von *P*.

Sieb des Eratosthenes

M = 36

	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

$7^2 > 36$ ➔ Algorithmus terminiert. Alle nichtmarkierten Felder sind Primzahlen.

- Schreibe alle natürlichen Zahlen von 2 bis zu einem frei wählbaren Maximalwert *M* auf. Zunächst sind alle Zahlen unmarkiert.
- Bestimme die kleinste unmarkierte Zahl *P*, deren Vielfache noch nicht markiert wurden. *P* ist immer eine Primzahl.
- Ist das Quadrat von *P* größer als der Maximalwert *M*, ergeben alle nicht markierten Zahlen die Primzahlen von 2 bis *M*.
- Markiere alle Vielfachen von *P* und beginne dabei mit dem Quadrat von *P*.

Sortierte Felder

Binäre Suche

Wenn die Elemente eines Feldes bereits sortiert sind, können wir den Vorgang der Suche nach einem Element wesentlich beschleunigen.

Prinzip (bei aufsteigend sortierten Werten ohne Duplikate)

1. Betrachte das gesamte Feld

2. Vergleiche das mittlere Element mit dem zu suchenden Wert:
   - Ist dieser größer, betrachte nur das rechts davon liegende Teilfeld.
   - Ist dieser kleiner, betrachte nur das links davon liegende Teilfeld.

3. Gehe zu 2. bis die Werte übereinstimmen oder die Feldgröße 0 ist.

Laufzeit

Durch die jeweilige Halbierung des zu durchsuchenden Feldes pro Schleifendurchlauf beträgt die Zeitkomplexität nur noch **O(log n)**.

Binäre Suche

Beispiel: Suche den Wert 17 in einem aufsteigend sortierten Feld mit acht Elementen

➔ wähle mittleres Element

2	3	5	7	11	13	17	19

17 ist größer als 7 ➔ durchsuche rechte Hälfte

wähle mittleres Element ➔

11	13	17	19

17 ist größer als 13 ➔ durchsuche rechte Hälfte

Vorteil:

Statt sieben Vergleichen bei linearer Suche sind bei binärer Suche nur noch drei Vergleiche notwendig!

wähle mittleres Element ➔

17	19

Element gefunden ➔

Felder - Zusammenfassung

Vorteile

Der Elementzugriff bei bekanntem Index ist mit **O(1)** äußerst schnell und **nicht** von der Elementanzahl des Feldes abhängig.

In sortierten Feldern können Elemente in **O(log n)** sehr schnell gefunden werden.

Nachteile

Einfügeoperationen laufen in **O(n)** und sind aufgrund der starren Feldgröße evtl. nicht durchführbar, falls das Feld keine Kapazität für einzufügende Elemente mehr bietet bzw. bereits vorhandene Elemente überschrieben werden müssten.

Löschoperationen laufen in **O(n)** und werfen die Frage auf, welche Semantik freiwerdende Feldelemente einnehmen.

Die Suche nach Elementen in nicht sortierten Feldern ist mit **O(n)** relativ langsam.

Die meisten Programmiersprachen bieten als eingebaute Datenstruktur nur Felder fester Länge an.

Felder - Zusammenfassung

Pro

Felder eignen sich bei

- Datenansammlungen mit fester Elementanzahl
- Notwendigkeit häufiger wahlfreier Elementzugriffe
- Ausführung gleichartiger Operationen über zusammenhängenden Bereichen bzw. der gesamten Datenmenge, z.B. Vektor- und Matrizenoperationen, Signalverarbeitung

Contra

Felder sind nicht geeignet für

- Datenansammlungen, denen häufig Elemente hinzugefügt oder entnommen werden
- unsortierte Datenansammlungen, in denen oft einzelne Werte gesucht werden müssen

Ende der Vorlesung