Vorlesung Wirtschaftsinformatik

Prof. Dr. Clemens Espe, MBA

Fakultät für Informatik

Wirtschaftsinformatik

Hochschule Augsburg

Inhaltsverzeichnis

Kapitel 5: Datenstrukturen
5.1. Elementare Datenstrukturen und Listen

Elementare Datenstrukturen
Basis Datentypen

char: Alle darstellbaren Zeichen,

int: Alle ganzen darstellbaren Zahlen,

float: Alle darstellbaren Gleitkommazahlen mit einfacher Genauigkeit,

double: Alle darstellbaren Gleitkommazahlen mit doppelter Genauigkeit,

array: Zusammenfassung zusammengehöriger Daten gleichen Typs,

Basis Datentypen
Arrays

Eindimensionaler Array:

Zusammenfassung eines Blocks von gleichartigen Daten.

Zu beachten ist, dass Arrays mit den Elementnummern 0 bis n-1 versehen sind!!

Beispiel: Array mit 9 string Elementen, string Buchname[9]

Buchname[0]	
Buchname[1]	Informatik
Buchname[2]	
Buchname[3]	
Buchname[4]	Algorithmen
Buchname[5]	
Buchname[6]	
Buchname[7]	Datenstrukturen
Buchname[8]	

Die Elemente eines Arrays liegen immer hinter-einander im Speicher des Rechner (was sie von verketteten Listen unterscheidet, deren Speicherplatz dynamisch zugeteilt wird und durch Zeiger vom Programm verwaltet werden muss)

Mehrdimensionaler Array:

Mehrdimensionale Arrays (in der Mathematik Matrizen genannt) verfügen über x Dimensionen (Vorsicht: Speicherplatzbedarf!!)

Elementare Datenstrukturen: Verkettete Listen
Einfach verkettete Listen

Nicht in jedem Fall ist bereits zum Zeitpunkt der Implementierung bekannt, aus wie vielen Elementen eine Liste bestehen wird (was zur Definition eines Arrays mit zusammenhängendem Speicherplatz nötig ist).

Hier bieten sich verkettete Listen als Datenstruktur an:

Mit jedem Listenelement wird ein Verweis auf den Speicherplatz des folgenden Elements gespeichert -> die Elemente können beliebig im Speicher verteilt sein, der Speicherplatz passt sich also dynamisch der jeweils aktuellen Größe der Liste an.

Einfach verkettete Listen

Implementierung: Die lineare Liste wird als Folge von Knoten implementiert, die jeweils ein Listenelement und einen Zeiger auf das Folgeelement enthalten:

Eine Liste L = <a1, …, an> kann folgendermaßen dargestellt werden:

Einfach verkettete Listen
Implementierung von Listenanfang und - ende

Als nächster Schritt muss festgelegt werden, wie der Listenanfang und das Listenende markiert werden sollen.

Dies soll realisiert werden, indem jeweils ein Dummy Element als Zeiger auf den Anfang (head) und das Ende (tail) hinzugefügt wird:

Einfach verkettete Listen
Initialisierung der Liste

Um die Liste zu initialisieren, werden die Knoten für den Listenanfang und das Listenende eingeführt. Dabei soll der Zeiger Ende-Knoten immer auf das vorherige Element der Liste zeigen:

Einfach verkettete Listen
Suchen eines Elementes in der Liste

Zum Suchen eines Elementes schreibt man das zu suchende Element vor Beginn der Suche in das Dummy Element (tail) am Listenende.

Dann wird von Beginn der Liste Schritt für Schritt jedes Listenelement mit dem gesuchten x verglichen. Kommt x in der Liste vor wird ein Zeiger auf das erste vorkommende Element mit Inhalt x geliefert. Kommt x nicht vor, wird ein Zeiger auf das Dummy Element (tail) geliefert:

Einfach verkettete Listen
Einfügen eines Elementes an gegebener Position p (1)

Zum Einfügen eines neuen Elementes x an der Position p wird das bestehende Element an der Position p an einen neuen Speicherplatz p+1 geschrieben und das Element p durch x ersetzt. Dadurch wird sicher gestellt, dass die jeweiligen Zeiger (insbesondere Element p-1) weiterhin korrekt sind:

Situation vor dem Einfügen von x:

Situation nach dem Einfügen von x:

Einfach verkettete Listen
Löschen eines Elementes an gegebener Position p (1)

Es soll das erste Element der Liste nach dem Dummy Element head gelöscht werden, das x enthält. Um zu verhindern, dass man durch das Löschen des letzten Elements in der Liste keinen Zugriff mehr auf das vorletzte Element hat (das ja jetzt auf das wiederum vorherige Element zeigen müsste), wird mit der Position pos auf das Element gezeigt, dessen next Komponente auf das zu löschende Element zeigt:

Situation vor dem Löschen von $a_p = x$:

Situation nach dem Löschen von $a_p = x$:

Hochschule Augsburg University of Applied Sciences

Einfach verkettete Listen
Löschen eines Elementes an gegebener Position p (2)

Umsetzung in Pseudo-Code:

Doppelt verkettete Listen

Durch die Implementierung von doppelt verketteten Listen kann das Handling der Listen (natürlich auf Kosten des zusätzlichen Speicherbedarfs) vereinfacht werden.

Implementierung:

Die Zeiger in den Elementen der Liste können folgendermaßen dargestellt werden:

Stack (Stapel)
LIFO (last-in-first-out-Prinzip)

Für viele Anwendungen genügt es, wenn Operationen am Ende der Liste durchgeführt werden können. Der Stack ist ein klassisches Anwendungsbeispiel, bei dem typischerweise folgende Operationen definiert sind:

push(): Fügt ein Element am oberen Ende des Stack ab

pop(): Entfernt das oberste Element aus dem Stack

top(): liefert das oberste Element ohne es zu löschen

Wenn die maximale Größe des Stack im vorhinein bekannt ist, bietet sich eine Realisierung als Array an, ansonsten als verkettete Liste.

Hochschule Augsburg University of Applied Sciences

Queue (Warteschlange)
FIFO (first-in-first-out) Prinzip

Die klassischen Operationen einer Warteschlangen:

put(): Fügt ein Element am Ende der Queue hinzu

get(): Entnimmt ein Element am Anfang der Queue und liefert es

Auch bei der Queue bietet sich eine Array-Realisierung an, wenn die maximale Größe der Queue im Vorhinein bekannt ist. Für dynamische Queue-Größen ist eine verkettete Liste empfehlenswert.

5.2. Binäre Suchbäume

Baumstrukturen
Einige Definitionen (1)

Baumstrukturen sind eine der am häufigsten auftretenden Datenstrukturen in der Informatik (z.B. Suchbäume, Entscheidungsbäume, Kodebäume etc.).

Bäume haben im Vergleich zu Listen nicht nur einen Nachfolger, sondern zwei (Binärbaum) bis n. Die Elemente werden *Knoten* genannt und der erste Knoten als *Wurzel* bezeichnet. Die direkten Nachfolgeknoten sind *Söhne* und direkte Vorgängerknoten *Väter.* Jeder Knoten ist mit einem anderen Knoten nur über EINEN *Pfad* verbunden.

Geordnete Bäume zeichnen sich dadurch aus, dass alle Söhne eines Baumes einer gegebenen Struktur (z. B. 1., 2., 3. Sohn) folgen. Knoten ohne Söhne werden als *Blätter* bezeichnet.

Die *Ebene* eines Knotens ergibt sich aus der Anzahl der Knoten auf dem Pfad zur Wurzel (ohne dem genannten Knoten. Die *Höhe* des Baums entspricht seiner höchsten Ebenen Nummer und die *Pfadlänge* eines Baumes ist die Summe der Ebenen aller Knoten.

Baumstrukturen
Einige Definitionen (2)

2012

Baumstrukturen
Einige Definitionen (3)

Baumstrukturen
Zeiger-Realisierung eines Suchbaumes

Bei Suchbäumen gilt für jeden Knoten p:

Sämtliche Schlüssel im linken Teilbaum von p sind kleiner als dessen Schlüssel, während sämtliche Schlüssel im rechten Teilbaum von p größer als dessen Schlüssel sind.

Beispiel:

Baumstrukturen
Zeiger-Realisierung eines Suchbaumes (1)

Grundsätzlich können Bäume programmtechnisch sowohl als Array als auch als Zeiger realisiert werden. Im folgenden soll die Realisierung durch Zeiger näher beleuchtet werden:

Bei der Umsetzung enthält jedes Element des Baumes den Schlüssel und jeweils einen Zeiger zum linken und einen Zeiger zum rechten Sohn. Damit kann jedes Element des Baumes folgendermaßen dargestellt werden:

Implementierung:

Baumstrukturen
Zeiger-Realisierung eines Suchbaumes (2)

Mit der gegebenen Definition ist ein Baum gegeben mit dem Zeiger auf die Wurzel: **var** root: Knotenzeiger

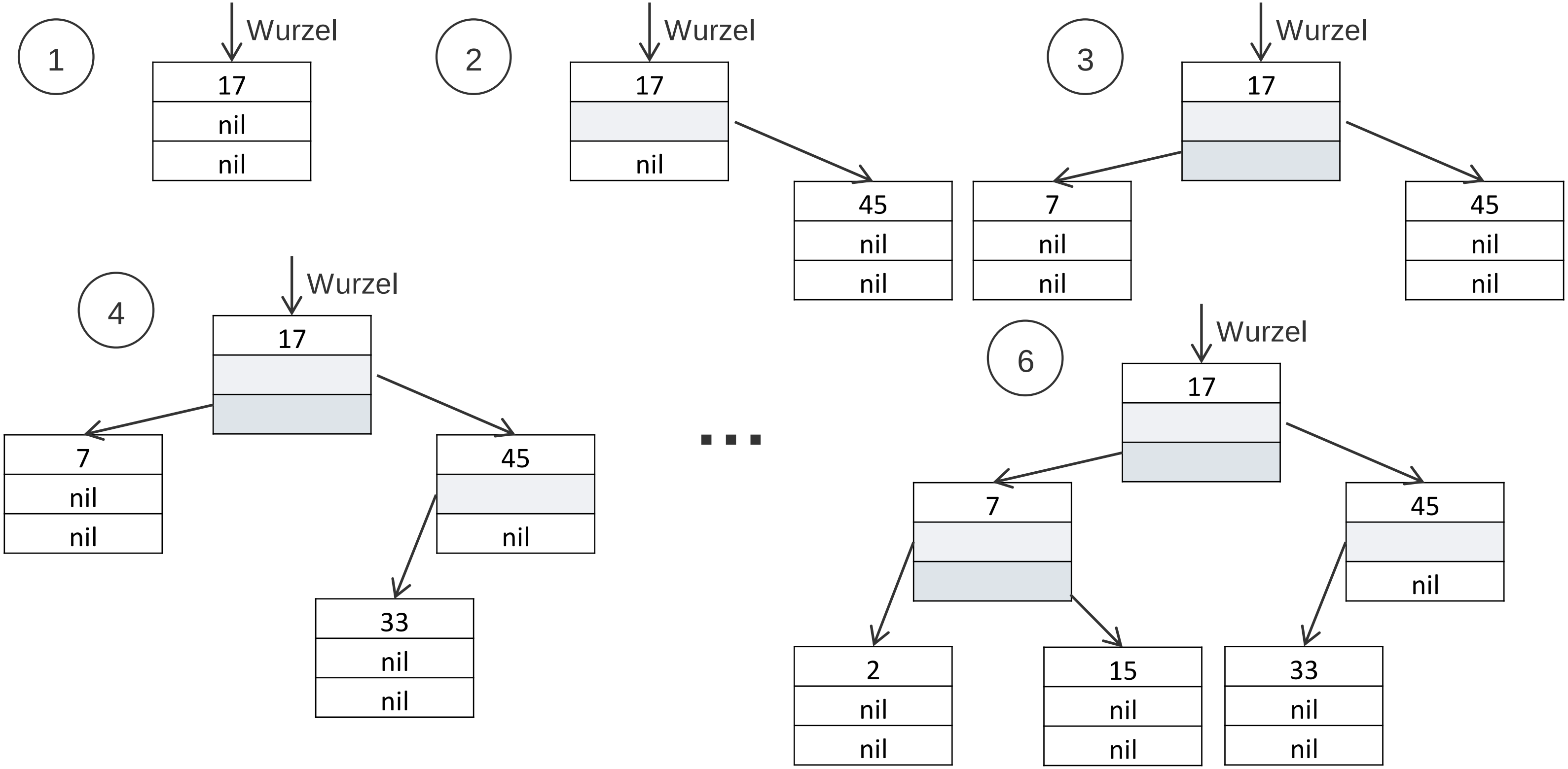
Die Blätter des Baumes beinhalten beim Suchbaum keine Schlüssel und können deshalb als nil-Zeiger repräsentiert werden:

Baumstrukturen
Implementierung zum Aufbau des Baumes

```
program Aufbau_Baum (input, output);
  type
    Knotenzeiger=↑Knoten;
    Knoten=record
            leftson, righson: Knotenzeiger;
            key: integer;
             end;
  var
    wurzel: Knotenzeiger;
    k:integer;

Function NeuerKnoten (var k:integer): Knotenzeiger
  begin
    new(p); {neuen Knoten schaffen}
      p↑.leftson := nil;
      p↑.rightson := nil;
      p↑.key := k;
      NeuerKnoten := p;
      end

Procedure Einfügen (var p:Knotenzeiger,k:integer);
begin
  if k < p↑.key {linker Zweig}
    then begin
      if p↑.leftson = nil then
        p↑.leftson := NeuerKnoten(k);
      else Einfügen(p↑.leftson, k);
    end
    else if k > p↑.key {rechter Zweig}
      then begin
        if p↑.rightson= nil then
          p↑.rightson := NeuerKnoten(k);
        else Einfügen(p↑.rigthson, k);
    else write(´Schlüsses existiert schon´);
end;

begin
  read(k);
  wurzel:= NeuerKnoten (k); {Wurzelknoten}
  while not eof(input) do {bis keine Eingabe mehr}
    begin
      read(k);
      Einfügen(wurzel,k)
    end
end;
```

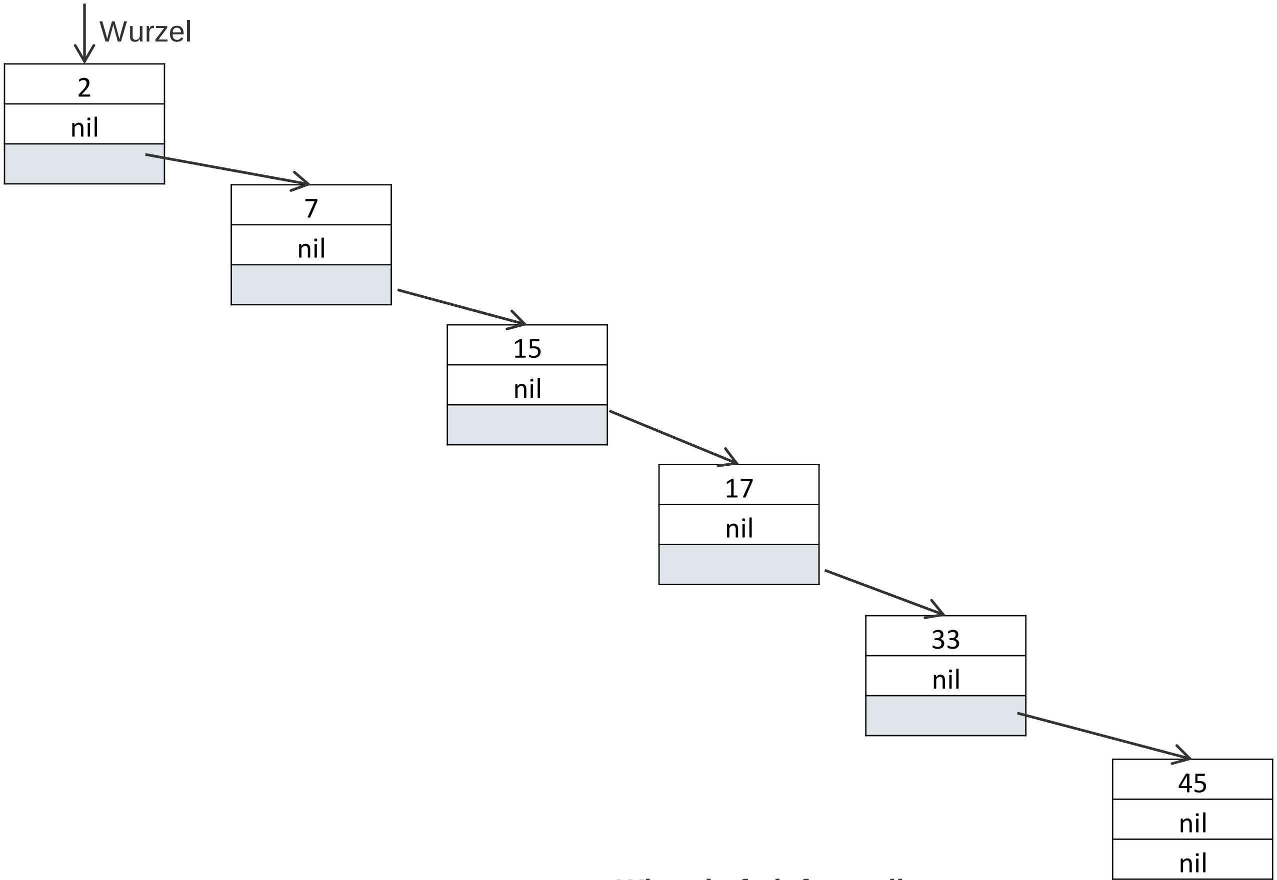
Baumstrukturen
Zahlenbeispiel: 17, 45, 7, 33, 2, 15

Je nachdem, welche Zahlen in welcher Reihenfolge eingegeben werden, können sehr unterschiedliche Baumstrukturen entstehen.



Baumstrukturen
Zahlenbeispiel: 2, 7, 15, 17, 33, 45

Je nachdem, welche Zahlen in welcher Reihenfolge eingegeben werden, können sehr unterschiedliche Baumstrukturen entstehen.



Baumstrukturen
Suchen eines Schlüssels

Auch zum Suchen eines Schlüssels kann hier wieder ein rekursives Vorgehen gewählt werden:

Baumstrukturen - Traversierungsreihenfolgen
Unterschiedliche Untersuchungsreihenfolgen

1. **Pre-order** (depth first/Hauptreihenfolge/Tiefensuche) Traversierung
   Von der Wurzel über den linken zum rechten Teilbaum

2. **Post-order** (Nebenreihenfolge) Traversierung
   Vom linken über den rechten Teilbaum zur Wurzel

3. **In-order** (Symmetrische Reihenfolge) Traversierung
   Vom linken Teilbaum über die Wurzel zum rechten Teilbaum

4. **Reverse in-order** (anti-symmetrische Reihenfolge) Traversierung
   Vom rechten Teilbaum über die Wurzel zum linken Teilbaum

5. **Level-order** (breadth-first/Breitensuche)
   Von der Wurzel wird Ebene für Ebene durchlaufen

5.3. Balancierte Suchbäume
Warum balancierte Suchbäume?

Je nach Eingabe können binäre Suchbäume ggf. zu sehr unsymmetrischen Strukturen führen (z. B. aufsteigende oder absteigende Eingabe), was zu einer schlechten Worst Case Performance führt.

Bei idealer Balance des Suchbaumes beträgt die Höhe des Baumes $log_2 N$ und es werden maximal $log_2 N$ Vergleiche zur Suche eines Elements benötigt.

Der im weiteren dargestellte Ansatz des 2-3 Suchbaumes stellt einen Kompromiss zwischen (aufwendig zu implementierender) perfekter Balance und garantiert logarithmischer Performance dar.

5.3. Balancierte Suchbäume
Der 2-3-Suchbaum: Definition

Ein 2-3 Suchbaum ist entweder leer oder ein:

- 2-Kind-Knoten mit einem Schlüssel und einer linken Referenz zu einem 2-3 Suchbaum mit kleineren Schlüsseln und einer rechten Referenz zu einem 2-3 Suchbaum mit größeren Schlüsseln
- 3-Kind-Knoten mit 2 Schlüsseln und einer linken Referenz zu einem 2-3 Suchbaum mit kleinerem Schlüssel, einer mittleren Referenz zu einem 2-3 Suchbaum mit Schlüsseln zwischen den Schlüsseln des Knotens und einer rechten Referenz zu einem 2-3 Suchbaum mit größerem Schlüssel.

2-3 Suchbaum
Suche eines Eintrages

Suche der Schlüssel 17 und 4 im 2-3 Suchbaum

2-3 Suchbaum
Einfügen in 2-Kind-Knoten

Einfügen des Schlüssels 18 im 2-3 Suchbaum

Hochschule Augsburg University of Applied Sciences

2-3 Suchbaum
Einfügen in 3-Kind-Knoten (I)

Das teilen eines 4er-Knoten ist eine lokale Operation:

Hochschule Augsburg University of Applied Sciences

2-3 Suchbaum
Einfügen in 3-Kind-Knoten (II)

5.3. Balancierte Suchbäume
Gruppenarbeit zum Aufbau balancierter Suchbäume

Aufgabe:

Bauen Sie jeweils den 2-3 Suchbaum bei folgender Eingabe schrittweise auf:

1) 14, 3, 20, 18, 9, 17, 25, 5, 12, 1
2) 1, 3, 5, 9, 12,14, 17, 18, 20, 25

Zeit: 10 Minuten

5.3. Balancierte Suchbäume
Ergebnis Gruppenarbeit

Kapitel 5: Datenstrukturen
5.4. Hashtabellen

5.4. Hashtabellen
Idee der Hashtabelle

- Speichert man Elemente in einem Array oder in einer Liste, so muss man i.A. alle Positionen durchsuchen, um das geeignete Element zu finden
- Wenn man anhand des Elementes selbst auf die Position, an der das Element gespeichert werden soll, schließen kann, lässt sich das Verfahren optimieren
- Effizient wäre dies umzusetzen, wenn sich jedes Objekt eindeutig einem Index in einem Array zuordnen lassen würde
- Einfügen, löschen und suchen: Man muss nur an der entsprechenden Stelle nachsehen und das Element dort einfügen, löschen, oder zurückliefern (Suche)

Warum Hashtabellen?

Verfahren zum Speichern von Datensätzen

5.4. Hashtabellen
Wie funktioniert Hashing?

- **Hashfunktion** berechnet den Arrayindex aus dem Suchschlüssel
- Dies kann zu Kollisionen führen, wenn unterschiedliche Suchschlüssel auf den identischen Arrayindex verweisen
- Mit der **Kollisionsauflösung** wird dieses Problem gelöst
- Somit wird mit Hashverfahren versucht, einen optimalen Kompromiss zwischen Rechenzeit und Speicherbedarf zu bilden

Stabler	17
Meier	20
Kuhls	2
Schneider	18
Bauschulte	23
Tautz	12
Ahrens1	17->19
Schirrmacher	0
Mawlawi	29->
Peters	13
Markiewicz	16
Kawulok	26
Ahrens2	17->21
Horn	5
Fredersdorf	1
Juvczok	28
Foege	1->3
Noll	3->4
Kessler	33-
Celik	18->22
Urbansky	28->30
Mietling	4->6
Hartung	20->24
Bollmeyer	0->7
Wiedebusch	18->25
Lammers	30->31
Szyperrek	16->27
Behrens	1->8

#	N A M E
0	Schirrmacher
1	Fredersdorf
2	Kuhls
3	Foege
4	Noll
5	Horn
6	Mietling
7	Bollmeyer
8	Behrens
9	
10	
11	
12	Tautz
13	Peters
14	
15	
16	Markiewicz
17	Stabler
18	Schneider
19	Ahrens1
20	Meier
21	Ahrens2
22	Celik
23	Bauschulte
24	Hartung
25	Wiedebusch
26	Kawuloc
27	Szyperrek
28	Juvczok
29	
30	Urbansky
31	Lammers
32	
33	Kessler
34	
35	

Hochschule Augsburg University of Applied Sciences

Hashfunktionen
Positive Integer: modulares Hashing

1. Im ersten Schritte gilt es festzulegen, wie groß die maximale Schlüssel-menge M ist, die in dem zum Speichern der Daten verwendeten Array abgelegt werden
2. Dividiert man nun den Schlüsselwert durch M-1 und nimmt den Rest der Division (Modulo-Operation), so ist dieser immer ein Wert zwischen 0 und M-1 und kann damit zur Adressierung des Arrays verwendet werden
3. Eine ideale Verteilung wird erreicht, wenn für M eine Primzahl verwendet wird. Ansonsten ist die Wahrscheinlichkeit groß, dass nicht alle Bits des Schlüssels berücksichtigt werden und damit keine optimal gleichmäßige Verteilung über das Array gegeben ist.

Schlüssel	M-1=100	M-1=101
685	85	79
921	21	12
455	55	51
379	79	76
772	72	65
845	45	37
680	80	74
229	29	27
302	2	100
322	22	19
169	69	68
280	80	78
263	63	61
877	77	69
372	72	69
536	36	31
366	66	63
280	80	78
202	2	0
444	44	40
108	8	7
115	15	14

Hashfunktionen
Strings: modulares Hashing auch möglich

- In diesem Fall wird der String wie ein riesiger Integer behandelt, indem die ASCII Werte der einzelnen Char Werte ausgelesen werden
- Um alle Buchstaben des Strings unabhängig von ihrer Stelle im String gleichwertig zu behandeln, werden sie mit einer Konstanten K multipliziert und char für char aufaddiert:
- Der folgende Code wendet dies beispielsweise für K = 47 und eine Arraygröße M = 100 an:
- Dabei ergeben sich beispielsweise folgende Hash-Werte:

Hashing mit Verkettung
Problemstellung

Das Hashverfahren teilt sich grundsätzlich in die zwei Schritte auf:

1. Hashfunktion zur Umwandlung von Schlüsseln in Array-Indizes
2. **Kollisionsauflösung** für den Fall, dass mehrere Schlüssel in den identischen Index aufgelöst werden:

Eine effektive und einfach zu implementierende Kollisionsauflösung ist das Hashing mit Verkettung.

Hashing mit Verkettung
Grundprinzip

Nutze ein Array mit M < N
verketteten Listen:

- Hash:
- Einfügen:
- Suche:

Hochschule Augsburg University of Applied Sciences

Hashing mit linearer Sondierung
Grundprinzip

Bei den Verfahren des Hashing mit offener Adressierung wird beim Eintreten einer Kollision in dem Array nach dem nächsten freien Arrayindex gesucht. Bei der lineare Sondierung wird im Falle einer Kollision schrittweise der nächste Eintrag geprüft mit folgenden Alternativen:

Die Implementierung erfolgt typischerweise mit parallelen Arrays für Schlüssel und Werte. Der Index zum Identifizieren der Daten im Arrays wird über den Index durch die Hashfunktion ermittelt.

Hashing mit linearer Sondierung
Konkretes Beispiel

Schlüssel	Hash	Wert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	6	0																
E	10	1																
A	4	2																
R	14	3																
C	5	4																
H	4	5																
E	10	6																
X	15	7																
A	4	8																
M	1	9																
P	14	10																
L	6	11																
E	10	12																

Hashing mit linearer Sondierung
Performance

Annahme

- „Perfekte“ Hash-Funktion
- m Tabelleneinträge
- n Anzahl der gespeicherten Einträge

Dann gilt:

- Anzahl Vergleiche S, um einen Eintrag zu finden

Hash-Verfahren
Anmerkungen

Hashing ist ein in der Praxis häufig eingesetztes Verfahren, um auf Objekte schnell zugreifen zu können. Beispiele:

Eine Datenstruktur „Hash-Tabelle“ ist in vielen Programmiersprachen in Form von Bibliotheken verfügbar. Beispiele:

Es gibt eine Vielzahl unterschiedlicher Hash-Verfahren und Hash-Funktionen (z. B. optimiert für bestimmte Anwendungsbereiche)

Eine aussagekräftige Komplexitätsanalyse ist oft schwierig, weil man die Häufigkeit von Kollisionen nur abschätzen kann, wenn man die Anwendung hinreichend gut kennt.

Hochschule Augsburg University of Applied Sciences