Algorithmen und Datenstrukturen

- Zusammenfassung -

Jörn Spichal

Contents

1 Was fehlt? 2

2 Laufzeitanalyse 2
2.1 Laundau-Notation . . . 2
2.2 Master-Theorem . . . 2

3 Sortierverfahren 2
3.1 Counting Sort . . . 2
3.2 Quicksort . . . 3
3.3 Selection Sort . . . 5
3.4 Heapsort . . . 6

4 Bäume 10
4.1 Traversierung Binärbaum . . . 10
4.2 B^+ Baum . . . 10

5 Hashing 12
5.1 Seperate Chaining / Verkettung . . . 12
5.2 Lineares Sondieren / Linear Probing . . . 12
5.3 (alternierendes) Quadratisches Sondieren . . . 12
5.4 Double Hashing . . . 12

6 Dynamisches Hashing 13
6.1 Linear Hashing . . . 13
6.2 Extendible Hashing . . . 14

7 Graphen 14
7.1 Dijkstra-Algorithmus . . . 14
7.2 Topologische Ordnung . . . 15

1 Was fehlt?

- Bucketsort, Mergesort, Radixsort
- Laufzeitvergleich der Sortierverfahren
- Beispiele (Algorithmen) für Laufzeiten
- Binärer Suchbaum
- Datenstrukturen B^+-Baum, binärer Suchbaum
- Algorithmus von Kruskal (minimal spannender Baum)

2 Laufzeitanalyse

2.1 Laundau-Notation

$f \in o(g)$	f wächst langsamer als g
$f \in \mathcal{O}(g)$	f wächst nicht schneller als g (obere Schranke)
$f \in \Theta(g)$	f wächst genauso schnell wie g
$f \in \Omega(g)$	f wächst nicht langsamer als g (untere Schranke)
$f \in \omega(g)$	f wächst schneller als g

2.2 Master-Theorem

Das Master-Theorem kann bei rekursiven Laufzeitberechnungen angewandt werden, die folgender Form entsprechen:

$a \cdot T(n/b) + n^c \qquad a \geq 1, b > 1, c \geq 0$

Dann ergibt sich für die Laufzeit:

$$T(n) = \begin{cases} \Theta(n^{log_b a}), & \text{falls } c < log_b a \\ \Theta(n^c \ log \ n), & \text{falls } c = log_b a \\ \Theta(n^c), & \text{falls } c > log_b a \end{cases}$$

3 Sortierverfahren

3.1 Counting Sort

- stabil (da identische Elemente bei identischem Key)
- linear zur Eingabelänge des Arrays
- beschränktes Intervall von Zahlen
- nur natürliche Zahlen
- nicht vergleichsorientiert
- Laufzeit $O(n + k)$, $k =$ Wertebereich

Algorithmus

```
countingsort(A, k)
  C = array(0, k)
  for (i=0; i<=k; i=i+1)
    C[i] = 0
  for (i=1; i<=A.size; i=i+1)
    C[A[i]] = C[A[i]] + 1
  j=0
  B = array(1, A.size)
  for (i=0; i<=k; i=i+1)
    for ( ; C[i]>0; C[i]=C[i]-1)
      B[j] = i
      j = j+1
  return B
```

Beispiel

Eingabearray A der Länge 10 mit Elementen aus dem Intervall {0,9}, also ist das Array C 9+1 Elemente groß (um das gesamte Intervall abzudecken)

1	7	4	6	9	0	2	8	4	1

Nun werden die Zahlen in A der Reihe nach gelesen und das entsprechende Feld in C um 1 erhöht.:

0	1	2	3	4	5	6	7	8	9
1	2	1	0	2	0	1	1	1	1

In Array B werden jetzt der Reihe nach die Indizes von C geschrieben (entsprechend des Wertes)

B:	0	1	1	2	4	4	6	7	8	9

3.2 Quicksort

- nicht stabil (es existieren aber stabile Varianten)
- Laufzeit Best Case O($n \cdot log(n)$)
- Laufzeit im Worst Case ist O(n^2) (z.B. Pivot ist immer letztes Element einer sortierten Liste)
- Rekursionstiefe Best Case von $log(n)$
- benötigt Platz auf dem Stack.

Algorithmus in C

```
void quicksort (int a[], int l, int r){
  // Ein oder weniger Elemente nicht sortieren
  if (r<=l+1) return;
  int p,i,j,t;
  p = a[l]; // linkes Element ist Pivot
  i = l+1; j = r;
  do{
    while ((a[i]<=p)&&(i<r)) i++; // groesseres Element suchen
    while ((a[j]>=p)&&(j>l)) j--; // kleineres Element suchen
```

```
    if (i>=j) break;
    t=a[i]; a[i]=a[j]; a[j]=t; // Tauschen
  }while(i<j)

  if (a[j]<a[l]){
    t=a[j]; a[j]=a[l]; a[l]=t; // Pivot tauschen
  }
  // quicksort auf Elemente links und rechts vom Pivot
  quicksort(a,l,j-1);
  quicksort(a,j+1,r);
}
```

Beispiel

Eingabearray A[0..8].

quicksort(0,8)
Die Indizes werden wie folgt gesetzt:
p=0; l=p+1; r=8;

5	7	4	3	5	9	7	8	6
p	l							r

Pivotelement ist das erste Element, also 0. Dann beginnt die Suche, l sucht von 1..8 ein größeres Element als das Pivot und r von 8..0 ein kleineres bis l≥r ist

5	7	4	3	5	9	7	8	6
p	l		r					

Da l<r wird A[l] mit A[r] getauscht und die Suche fortgesetzt.

5	3	4	7	5	9	7	8	6
p		r	l					

Jetzt ist r<l. Die Suche wird abgebrochen. Da A[r]<A[p] wird A[r] mit A[p] getauscht. Das Element an der Stelle r ist danach richtig einsortiert. Jetzt starten die Rekursionsaufrufe:
quicksort(0,1) und quicksort(3,8)

quicksort(0,1)

4	3	5	7	5	9	7	8	6
p	lr	-						

Jetzt wird gesucht. Index l steht schon ganz rechts und wird nicht verschoben. Index r steht bereits auf einem kleineren Wert als das Pivotelement. Da A[r]<A[p] wird A[r] mit A[p] getauscht. Jetzt starten die Rekursionsaufrufe:
quicksort(0,0) und quicksort(2,1)
Beide Aufrufe sind obsolet, weil im ersten Fall nur ein Element sortiert werden müsste und die Indizes im zweiten Fall ungültig sind. Also erinnern wir uns an den gemerkten Rekursionsaufruf und fahren mit quicksort (3,8) fort.

quicksort (3,8)

4	3	5	7	5	9	7	8	6
-	-	-	p	l				r

Wieder beginnt die Suche.

4	3	5	7	5	9	7	8	6
-	-	-	p		l			r

Es wird A[l] mit A[r] getauscht und die Suche fortgesetzt.

4	3	5	7	5	6	7	8	9
-	-	-	p		r		l	

Suche abbrechen, da r<l. A[r]<A[p], also tauschen. Danach ist A[r] richtig sortiert. Rekusrionsaufrufe: quicksort(3,4) und quicksort(6,8).

quicksort(3,4)

4	3	5	6	5	7	7	8	9
-	-	-	p	rl	-			

Die Suche verändert die Indizes r und j nicht, da l wieder ganz rechts steht und r auf ein kleineres Element als das Pivot steht und die Suche wird abgebrochen. A[r]<A[p], also tauschen.Danach ist A[r] richtig sortiert. Rekusrionsaufrufe:
quicksort(3,3) und quicksort(5,4). Wieder sind beide obsolet, also ist der Bereich richtig sortiert.

quicksort(6,8)

4	3	5	5	6	7	7	8	9
-	-	-	-	-	-	p	l	r

Suche.

4	3	5	5	6	7	7	8	9
-	-	-	-	-	-	pr	l	

Es wird nichts getauscht, da r<l und p=r. Rekursionsaufrufe:
quicksort(6,5) und quicksort(7,8). Erster ist obsolet.

4	3	5	5	6	7	7	8	9
-	-	-	-	-	-	-	p	rl

Suche.

4	3	5	5	6	7	7	8	9
-	-	-	-	-	-	-	pr	l

Es wird wieder nichts getauscht, da r<l und p=r. Rekusrionsaufrufe:
quicksort(7,6) und quicksort(8,8). Beide sind obsolet. Das Feld ist sortiert.

3.3 Selection Sort

- instabil
- Laufzeit n^2

Algorithmus

```
selectionsort(A){
  n = count(A)-1; // letztes Element
```

```
  for (int p=0; p<n-1; p++){
    m = p
    for (int i=p+1;i<=n;i++)
      if (A[i] < A[m]) m = i
    tausche(A[m],A[p]);
    p++;
  }
}
```

Beispiel

Array A[0..8]; n=8;
Im ersten Schritt ist p=0. min=p. Nun im Bereich i=1..8 das Minimum gesucht.

5	7	4	3	5	9	7	8	6
pm								

Minimum suchen.

5	7	4	3	5	9	7	8	6
p			m					

Das Minimum ist A[3] und wird mit A[p] getauscht. p wird um 1 erhöht und der Vorgang wiederholt.

3	7	4	5	5	9	7	8	6
	p	m						

3	4	7	5	5	9	7	8	6
		p	m					

3	4	5	7	5	9	7	8	6
			p	m				

3	4	5	5	7	9	7	8	6
				p				m

3	4	5	5	6	9	7	8	7
					p	m		

3	4	5	5	6	7	9	8	7
						p		m

3	4	5	5	6	7	7	8	9
							pm	

Der Vorgang wird für das letzte Element nicht wiederholt, da es automatisch einsortiert ist. Das Array ist sortiert.

3.4 Heapsort

- Laufzeit O($n \cdot log(n)$)
- nicht stabil
- in-place, überschreibt den Eingabespeicher

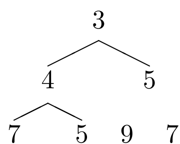
Algorithmus

```
public class HeapSorter{
    private int[] a;
    private int n;

    public void sort(int[] a){
        this.a=a;
        n=a.length;
        heapsort();
    }

    private void heapsort(){
        buildheap();
        while (n>1){
            n--;
            exchange (0, n);
            downheap (0);
        }
    }

    private void buildheap(){
        for (int v=n/2-1; v>=0; v--)
            downheap (v);
    }

    private void downheap(int v){
        int w=2*v+1;            // erster Nachfolger von v
        while (w<n){
            if (w+1<n)          // gibt es einen zweiten Nachfolger?
                if (a[w+1]>a[w]) w++;
            // w ist der Nachfolger von v mit maximaler Markierung

            if (a[v]>=a[w]) return;  // v hat die Heap-Eigenschaft

            exchange(v, w);     // vertausche Markierungen von v und w
            v=w;                // fahre mit v=w fort
            w=2*v+1;
        }
    }

    private void exchange(int i, int j){
        int t=a[i];
        a[i]=a[j];
        a[j]=t;
    }
}    // end class HeapSorter
```

Beispiel

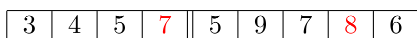
3	4	5	7	5	9	7	8	6

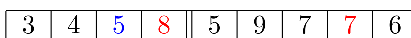
Die Zahlenfolge wird der Reihe nach als Binärbaum interpretiert, wobei die ersten $\lfloor n/2 \rfloor$ Zahlen Knoten und die letzteren Blätter darstellen.

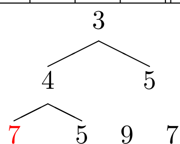


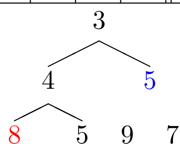
8 6

Buildheap.
Nun werden für alle Knoten vom letzten bis zur Wurzel die Heapeigenschaften überprüft (alle Kinder sind kleinergleich) und ggf. korrigiert. Bei falscher Heapeigenschaft wird das größte Kind *nach oben* verschoben und für das ersetzte Kind erneut die Heapeigenschaft wiederhergestellt (rekursiv), bis sie bis hin zu den Blättern erfüllt ist.
Wir beginnen also beim letzten Knoten, welcher die Heapeigenschaft nicht erfüllt. Also wird der Wert des Knotens mit dem größten Kind getauscht.



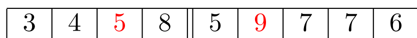


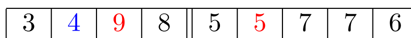


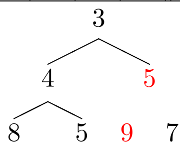


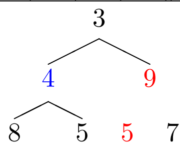
8 6 7 6

Jetzt ist die Heapeigenschaft für diesen Knoten erfüllt. Der nächste Knoten (5) erfüllt die Heapeigenschaft nicht und wird korrigiert.



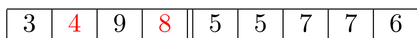


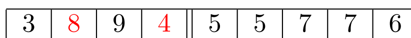


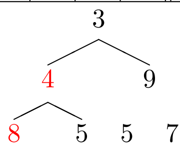


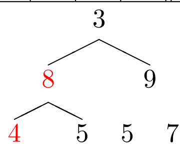
7 6 7 6

Weiter mit Knoten (4). Auch hier ist die Heapeigenschaft nicht erfüllt und der Wert wird mit (8) getauscht. Dadurch wird auch in diesem Knoten die Heapeigenschaft nicht mehr erfüllt und muss getauscht werden.

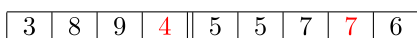


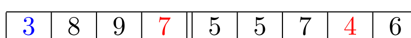


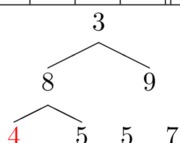


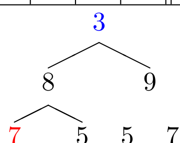


7 6 7 6



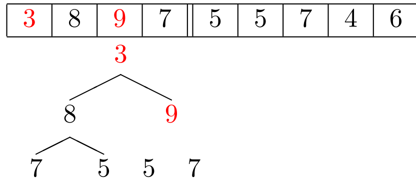
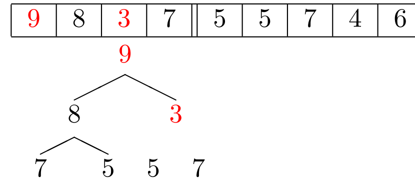






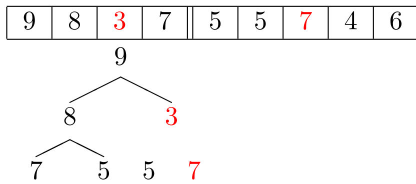
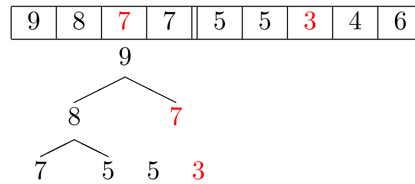
7 6 4 6

Nächster Knoten ist (3).

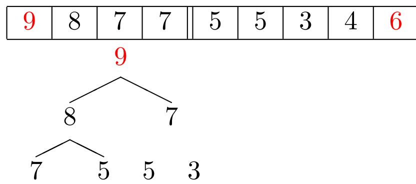
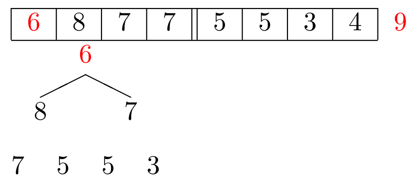
4 6 4 6

Heapeigenschaften für Knoten (3) wiederherstellen.

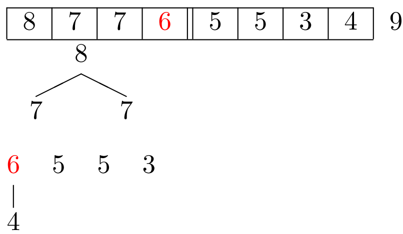
4 6 4 6

Nun ist in allen Knoten die Heapeigenschaft erfüllt. Jetzt wird der erste Knoten (9) mit dem letzten Blatt (6) getauscht, die Größe des Heaps um 1 reduziert (damit steht (9) am Ende der Liste und außerhalb des Heaps) und der Knoten (6) nach unten *versickert*, sodass die Heapeigenschaft im gesamten Heap wiederhergestellt ist. Dies wird so lange wiederholt, bis die Größe des Heaps 0 ist.

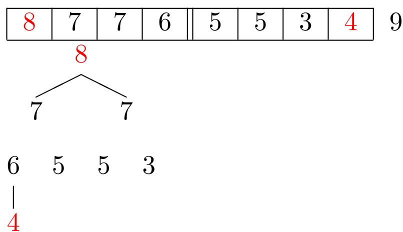
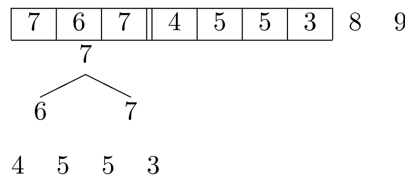



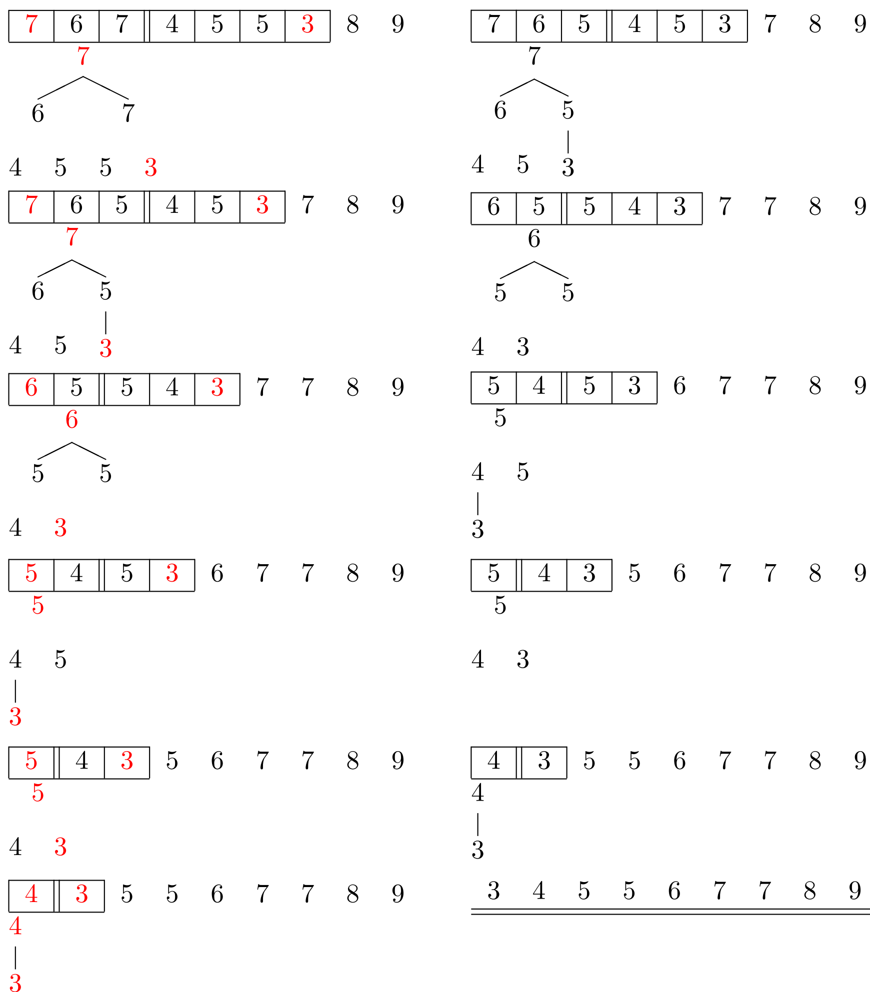
4 6

Versickern von (6).



Und wiederholen. Fortan ohne Zwischenschritte.



4 Bäume

4.1 Traversierung Binärbaum

Traversierung eines Baumes bedeutet das systematische Besuchen seiner Knoten. Es gibt drei interessante Reihenfolgen beim rekursiven durchlaufen des Baumes:

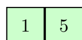
	preorder	*inorder*	*postorder*
1.	**node**	left	left
2.	left	**node**	right
3.	right	right	**node**

4.2 B^+ Baum

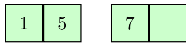
Beispiel Ordnung $k = 1$

Einfügen der Zahlenfoge 1 5 7 9 3 6 4 2

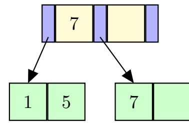
Einfügen von: 1 und 5



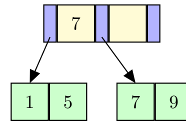
Einfügen von: 7



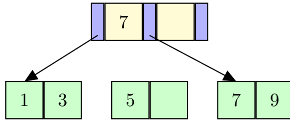
Erzeuge Wurzel



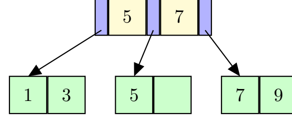
Einfügen von 9



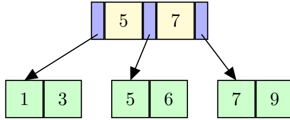
Einfügen von 3



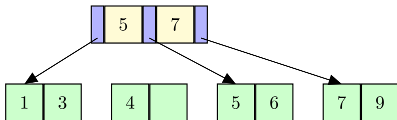
Verlinken



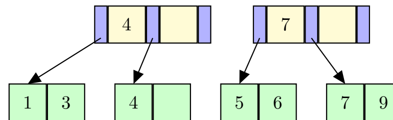
Einfügen von 6



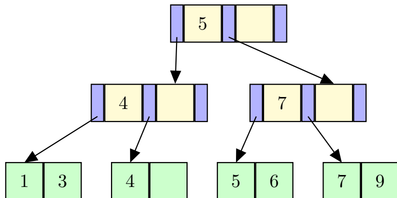
Einfügen von 4



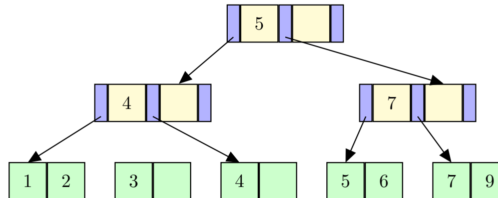
Wurzel-Split



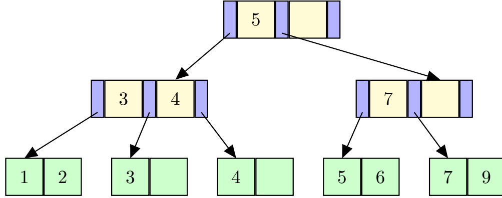
Neue Wurzel erzeugen.



Einfügen von 2

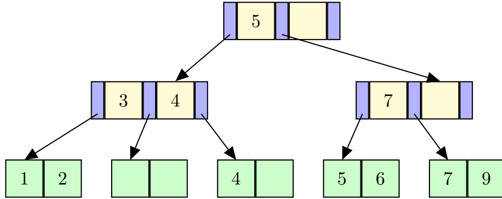


Verlinken

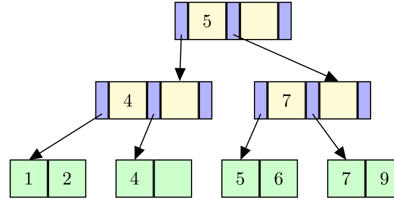


Jetzt werden die Werte 3 4 5 6 in Reihenfolge gelöscht

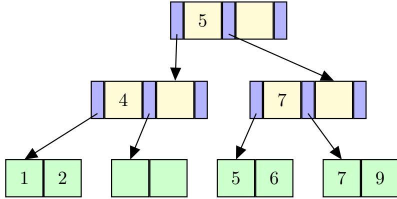
Löschen von 3



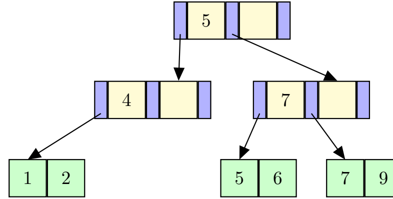
Leeres Blatt entfernen



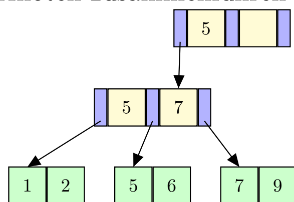
Löschen von 4



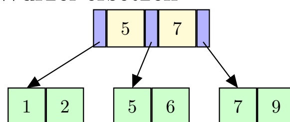
Leeres Blatt entfernen



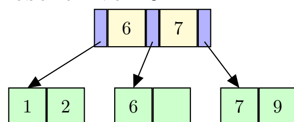
Knoten zusammenführen



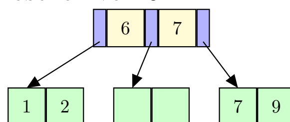
Wurzel ersetzen



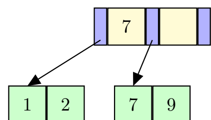
Löschen von 5



Löschen von 6



Leeres Blatt entfernen



5 Hashing

Die grundlegende Idee von Hashing ist es, lese-schreibe-suche-Zugriffe in $O(1)$ zu bewerkstelligen. Anhand einer Haschfunktion wird die Position eines Elements in einem Vektor berechnet. Dabei kann es zu Kollisionen kommen, wenn zwei Elemente demselben Hashwert zugeordnet werden.

5.1 Seperate Chaining / Verkettung

Es werden Lineare Listen (Überlaufketten) an den Buckets gebildet.

5.2 Lineares Sondieren / Linear Probing

Es wird bei einer Kollision der nächste freie Platz ausgehend von der Position der Kollision bestimmt ($g(k) = 1$). Dadurch können Cluster etnstehen. Laufzeit von höchstens m.

5.3 (alternierendes) Quadratisches Sondieren

$h_i(k) = \left(h(k) + (-1)^{i+1} \cdot \lceil i/2 \rceil^2\right)\ mod\ m = h(k)+1, h(k)-1, h(k)+2, h(k)-2, h(k)+4, \ldots$
Alternierend, wenn Vorzeichenwechsel. Verhindert im Vergleich zur Linearen Sondierung die Clusterbildung. Es werden mindestens $m/2$ Buckets besucht, wenn m schlecht gewählt. Möglichkeit einer sekundären Clusterbildung.

5.4 Double Hashing

Beim Doublehashing wird als Kollisionsbehandlung eine zweite Hashfunktion angewendet.

Beispiel

Es werden in Reihenfolge folgende Werte in eine Hashtabelle der Größe $m = 9$ eingefügt: 9 12 6 4 7 25.
1. Hashfunktion: $h(k) = k\ mod\ 9$
2. Hashfunktion: $g(k) = (k\ mod\ 10) \cdot 3$
$a_0 = h(k)$, $a_{i+1} = (a_i + g(k)) mod\ m$

k	9	12	6	4	7	25
$h(k)$	0	3	6	4	7	7
$g(k)$	27	6	18	12	21	15

Beim Einfügen der Werte 9 12 6 4 7 tritt keine Kollision auf. Es ergibt sich also folgende Hashtabelle:

0	1	2	3	4	5	6	7	8
9			12	4		6	7	

Da Platz 7 bereits belegt ist, kann 25 dort nicht eingefügt werden und es muss eine alternative Position berechnet werden:

$a_0 = h(k) = 7$
$a_1 = (a_0 + g(k))\ mod\ m = (7 + 15)\ mod\ 9 = 4$ (belegt)
$a_2 = (4 + 15)\ mod\ 9 = 1$
Also ergibt sich 1 als Position für 25 und somit folgende Hashtabelle

0	1	2	3	4	5	6	7	8
9	25		12	4		6	7	

Der Wert 81 könnte nicht mehr eingefügt werden, weil sich folgender Kollisionpfad ergibt:

$a_0 = h(81) = 0$
$a_1 = (a_0 + k(81))\ mod\ 9 = 3\ mod\ 9 = 3$
$a_2 = 6$
$a_3 = 0$
$\dots$

Suchen des Wertes 15.

$a_0 = 6$ (belegt und ungleich 15, also Kollisionspfad verfolgen)
$a_1 = 3$ (belegt ungleich 15)
$a_2 = 0$ (belegt ungleich 15)
$a_3 = 6 = a_0$

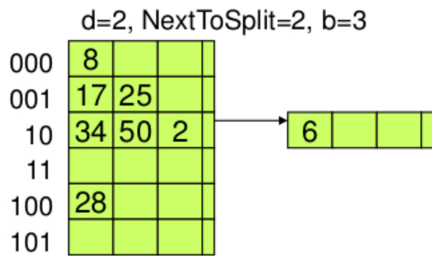
15 könnte nur an den Positionen 0 3 6 vorkommen. Die zyklische Berechnung von a führt zum Abbruch.

6 Dynamisches Hashing

Dynamische Hashverfahren können ihren Speicherbereich nach Bedarf dynamisch erweitern.

6.1 Linear Hashing

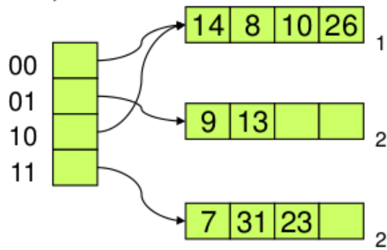
Es werden Blöcke der Größe b verwaltet, die bei Überlauf in einer linearen Liste organisiert sein können. Es existiert eine Hashfunktion h, die die Elemente in Buckets einsortiert. Dabei bestimmt eine Tiefe d, wieviele Bits (von hinten) der Hashfunktion zur Bestimmung des Buckets verwendet werden.
$NextToSplit$ notiert sich das Bucket, welches beim nächsten Überlauf (Erzeugung eines Überlaufblock in der verketteten Liste) erweitert wird. Die Tiefe der Blöcke ist $[NextToSplit, 2^d - 1] = d$ und sonst $d + 1$.
Bei einem Split wird der Block $NextToSplit$ in zwei Blöcke mit erhöhter Tiefe aufgeteilt und alle (auch in der Überlaufliste) enthaltenen Elemente auf beide entsprechend der Hashfunktion auf die beiden Verteilt.



- Ungünstig, wenn die letzten Stellen vieler Elemente gleich sind, sodass sie auch nach mehreren Split-Runden in denselben Bucket einsortiert werden.
- Splitten z.B. bei jedem Überlauf oder nach Belegungsfaktor (Nach Einfügen von $L \cdot b$ Elementen seit letztem Überlauf, L=Load)

6.2 Extendible Hashing

Es wird ein Index zur Verwaltung der Blöcke der Größe b verwendet. Verwendung einer Hashfunktion ähnlich zu Linear Hashing. Bei Überlauf wird der Indexbereich so lange verdoppelt, bis Platz für das einzufügende Element vorhanden ist. Ein Bucket kann von mehreren Indizes referenziert werden und hat eine vom Index unabhängige Tiefe.



- Mehrfache Indexverdoppelung kann auftreten
- Im Worst-Case exponentieller Indexwachstum (Clustered Data)
- löschen leerer Blöcken nicht einfachen (Indexverkleinerung)
- Nur ein externer Zugriff, wenn Index in Speicher passt
- keine garantierte Mindesauslastung

7 Graphen

7.1 Dijkstra-Algorithmus

Gegeben ist ein gerichteter Graph G, welcher durch folgende Adjazenzmatrix dargestellt wird (zur Übersichtlichkeit als Tabelle):

	v_1	v_2	v_3	v_4	v_5
v_1	-	2	7	15	20
v_2	-	-	4	-	5
v_3	-	7	-	1	6
v_4	-	-	-	-	4
v_5	-	-	-	9	-

Der Zeilenindex steht für *von*, der Spaltenindex für *nach*. Zeile 1 Spalte 3 wäre also ein Weg von 7 von v_1 nach v_3. Wir wollen nun die kürzesten Wege von v_1 zu allen anderen Knoten herausfinden. Dazu merken wir uns die besuchten Knoten, den aktuellen Knoten, die minimalen Wege und die Vorgänger f.d. kürzesten Weg (Wenn ein kürzester Weg angegeben werden soll).

Zu Beginn besuchen wir den Ausgangsknoten und setzen die minimalen Entfernungen entsprechend der vorhandenen Kanten und unendlich bei unerreichbaren Knoten. Die Vorgänger werden entsprechend gesetzt.

Algorithmus:

1. Startknoten besuchen und Distanzen der vorhandenen Kanten merken
2. solange es unbesuchte Knoten gibt:
   (a) besuche unbesuchten Knoten mit niedrigster Distanz
   (b) berechne Distanz zu allen unbesuchten Konten. Wenn diese kleiner ist, Distanz und Vorgänger merken

Besucht	**v**	**Min**	**Vorgänger**
{1}	-	[0,2,7,15,20]	[-,1,1,1,1]
{1,2}	2	[0,2,6,15,7]	[-,1,2,1,2]
{1,2,3}	3	[0,2,6,7,7]	[-,1,2,3,2]
{1,2,3,4}	4	[0,2,6,7,7]	[-,1,2,3,2]
{1,2,3,4,5}	5	[0,2,6,7,7]	[-,1,2,3,2]

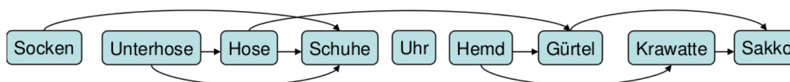
Der kürzeste Weg von v_1 nach v_4 Beträg demnach 7 und kann über die Vorgänger angegeben werden: $v_1 \to v_2 \to v_3 \to v_4$.

Mögliche Datenstruktur einer Adjazenzliste:

```
class node {
 public:
  int v;
  node *next;
};

node *adjliste[maxV];
```

7.2 Topologische Ordnung

Eine Topologische Ordnung eines gerichteten azyklischen Graphs G (directed acyclic graph, DAG) ist eine lineare Anordnung aller Knoten, sodass u vor v in der Anordnung steht, wenn G eine Kante $< u, v >$ enthält. Falls der Graph nicht azyklisch ist, gibt es keine topologische Ordnung.



Algorithmus

1. Falls kein Knoten mehr vorhanden, halte an.

2. Kein Knoten ohne eingehende Kante? $\Rightarrow$ anhalten, Graph nicht azyklisch
3. Knoten ohne eingehende Kante in der topologischen Ordnung notieren und aus dem Graphen entfernen
4. Weiter bei 1.