Suchen & Sortieren

(von Luca Dreiling, Stand: 9. Juli 2019)

1 Anmerkungen:

- Es werden, der Einfachkeit halber, nur Arrays betrachtet.
- Die Länge der Array wird mit n bezeichnet.
- Wir betrachten für die Beispiele immer das Array: | 7 | 1 | 8 | 2 | 11 | 4 |
- Die farblichen Markierungen dienen zur Zuordnung der einzelnen Hilfskonstrukte(z.B. Bubbles, sortierte Listen, ...)

2 Laufzeitenberechnung:

Eines der Wichtigsten Gütekriterien eines Algorithmus ist die Laufzeit. Diese wird approximiert mit der Anhand der Skalierung des Problems mit der Eingabegröße (mit n angegeben) gemessen. Für die Laufzeiten werden $worst, best$ und $average - case$ betrachtet.
Die Approximation erfolgt (bei $\mathcal{O}(g)$) mittels Abschätzung einer Funktion nach oben. Ist die Laufzeit $f \in \mathcal{O}(g)$, so wächst g für $n \to \infty$ stärker als $c \cdot f$. Die Anzahl der Operationen ist dabei unbedeutend, weshalb sie durch eine Konstante c ausgeglichen wir. Zum Vergleichen werden unterschiedliche „Klassen"betrachtet, darunter (aufsteigend): $1(konstant), \log(n), n, n\log(n), n^2, 2^n$.
Es gilt logischerweise, je schneller desto besser!

3 Suchverfahren:

3.1 Suchen mit Listen:

Das ist das primitive Suchen, man betrachtet jedes Element in der Liste, bis man das gesuchte Element findet oder am Ende ist.

3.2 Anwendungsbeispiel:

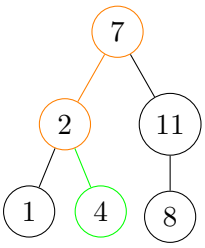
- **Suche 11:** | 7 | 1 | 8 | 2 | 11 | 4 | $\Rightarrow$ 11 gefunden!
- **Suche 42:** | 7 | 1 | 8 | 2 | 11 | 4 | $\Rightarrow$ 42 nicht gefunden!

3.3 Suchen mit Suchbäumen:

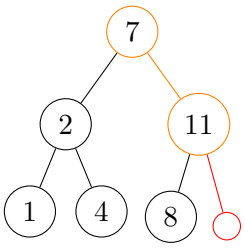
Meist ein binärer Baum (zwei Kindsknoten), mit der Ordnung $linkes\ Kind \leq Elternknoten < rechtes\ Kind$.
Um ein Element darin zu suchen, kann man anhand des Elternknotens entscheiden, ob man für das gesuchte Element zum linken oder rechten Kind absteigen muss.
Zum Erstellen eines Suchbaumes werden, solang für den Wert die Abzweigungen genommen, bis man an einen freien Platz im Baum kommt. Dort wird ein neuer Knoten mit dem Wert erstellt.
Die Anzahl der Vergleiche ergibt sich aus der Länge des Pfades zum gesuchten Knoten.

Im $worst-case$ bildet der Baum eine Liste und man muss jedes Element betrachten. (Um das zu verhindern gibt es Strategien, den Baum flach zu halten. Im Zuge der Vorlesung jedoch nicht wichtig).

3.4 Anwendungsbeispiel (zufälliger Suchbaum):



Beispiel für Suche nach 4



Suche nach 42 (nicht im Baum)

4 Sortierenverfahren:

4.1 Selectionsort:

4.1.1 Vorgehen:

Das Array wird in zwei Teile geteilt: Ein sortiertes (linkes) und ein unsortierten (rechtes) Array. Dann wird für jedes Element (also n Mal) das kleinste Element im unsortierten Array gesucht und hinten an das sortierte Array angefügt.

4.2 Anwendungsbeispiel:

1. | 7 | 1 | 8 | 2 | 11 | 4 | ; Min = 1
2. | 1 | | 7 | 8 | 2 | 11 | 4 | ; Min = 2
3. | 1 | 2 | | 7 | 8 | 11 | 4 | ; Min = 4
4. | 1 | 2 | 4 | | 7 | 8 | 11 | ; Min = 7
5. | 1 | 2 | 4 | 7 | | 8 | 11 | ; Min = 8
6. | 1 | 2 | 4 | 7 | 8 | | 11 | ; Min = 11
7. | 1 | 2 | 4 | 7 | 8 | 11 | ; **Fertig**

4.3 Insertionsort

4.3.1 Vorgehen:

Das Array wird in zwei Teile geteilt: Ein sortiertes (linkes) und ein unsortierten (rechtes) Array. Dann wird für jedes Element (also n Mal) das erste Element des unsortierten Arrays entfernt und an die passende Stelle des sortierten Arrays geschrieben.

4.4 Anwendungsbeispiel:

1. 7 | 1 | 8 | 2 | 11 | 4 ; Nächstes: 7
2. 7 | 1 | 8 | 2 | 11 | 4 ; Nächstes: 1 ($1 < 7$)
3. 1 | 7 | 8 | 2 | 11 | 4 ; Nächstes: 8 ($8 > 1$, $8 > 7$)
4. 1 | 7 | 8 | 2 | 11 | 4 ; Nächstes: 2 ($2 > 1$, $2 < 7$)
5. 1 | 2 | 7 | 8 | 11 | 4 ; Nächstes: 11 ($11 > 1$, $11 > 2$, $11 > 7$, $11 > 8$)
6. 1 | 2 | 7 | 8 | 11 | 4 ; Nächstes: 4 ($4 > 1$, $4 > 2$, $4 < 7$)
7. 1 | 2 | 4 | 7 | 8 | 11 ; **Fertig**

4.5 Bubblesort

4.5.1 Vorgehen:

Man betrachtet sog. Bubbles. Das sind Intervalle/Teillisten einer bestimmten Länge (hier 2). Die Bubbles wandern schrittweise durch das Array, wobei die Bubble in jedem Schritt sortiert wird. So steigen (je nach Implementierung) die größten Objekte nach oben, wo sie ein sortiertes Array bilden. Die Bubble „platzt“beim sortierten Array und beginnt von vorne. Der Algorithmus ist fertig, wenn eine Bubble ohne Vertauschungen aufsteigt.

4.5.2 Anwendungsbeispiel:

1. 7 | 1 | 8 | 2 | 11 | 4 ; Tausche 7 und 1
2. 1 | 7 | 8 | 2 | 11 | 4 ; Bubble bereits sortiert
3. 1 | 7 | 8 | 2 | 11 | 4 ; Tausche 8 und 2
4. 1 | 7 | 2 | 8 | 11 | 4 ; Bubble bereits sortiert
5. 1 | 7 | 2 | 8 | 4 | 11 ; Tausche 11 und 4, Bubble „platzt“
6. 1 | 7 | 2 | 8 | 4 | 11 ; Bubble bereits sortiert
7. 1 | 7 | 2 | 8 | 4 | 11 ; Tausche 7 und 2
8. 1 | 2 | 7 | 8 | 4 | 11 ; Bubble bereits sortiert
9. 1 | 2 | 7 | 8 | 4 | 11 ; Tausche 8 und 4, Bubble „platzt“
10. 1 | 2 | 7 | 4 | 8 | 11 ; Bubble bereits sortiert
11. 1 | 2 | 7 | 4 | 8 | 11 ; Bubble bereits sortiert

12. 1 2 7 4 8 11 ; Bubble bereits sortiert
13. 1 2 4 7 8 11 ; Bubble bereits sortiert, Bubble „platzt“
14. 1 2 4 7 8 11 ; Bubble bereits sortiert
15. 1 2 4 7 8 11 ; Bubble bereits sortiert
16. 1 2 4 7 8 11 ; Bubble bereits sortiert, Bubble „platzt“
17. 1 2 4 7 8 11 ; Keine Vertauschungen durch letzte Bubble “⇒ **Fertig**

4.6 Mergesort

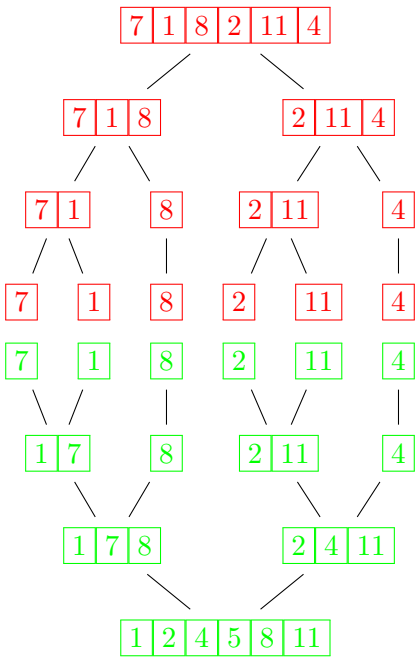
4.6.1 Vorgehen:

Dieses Sortierfverfahren nutzt Divide and Conquer: Es teilt das Array auf und fügt sie anschließend wieder zusammen. So entstehen kleinere, einfache Probleme. Der Aufwand liegt hier beim Zusammenfügen der Listen, dafür ist das Teilen sehr einfach. Das Array wird schrittweise in der Hälfte geteilt, bis nur einzelne Elemente vorliegen. Diese sind per Definition bereits sortiert.
Anschließend wird beim Mergen eine Teilliste in die andere sortiert. Beide Listen bereits sortiert, deshalb kann dies in einer Iteration durchgeführt werden. Fügt man ein Element ein, muss das nächste einzufügende auf jeden Fall rechts davon eingefügt werden.

Beispiel: Sortiere vorsortierte Liste 1 7 8 in ebenfalls vorsortierte Liste 2 4 11 (über den grünen Bereich wurde bereits iteriert, der rote noch nicht. Im Grünen werden keine Elemente mehr hinzugefügt):

1. Füge 1 in 2 4 11 ein → 1 2 4 11
2. Füge 7 in 1 2 4 11 ein → 1 2 4 7 11
3. Füge 8 in 1 2 4 7 11 ein → 1 2 4 7 8 11
4. Alle Elemente eingefügt → 1 2 4 7 8 11

4.6.2 Anwendungsbeispiel:

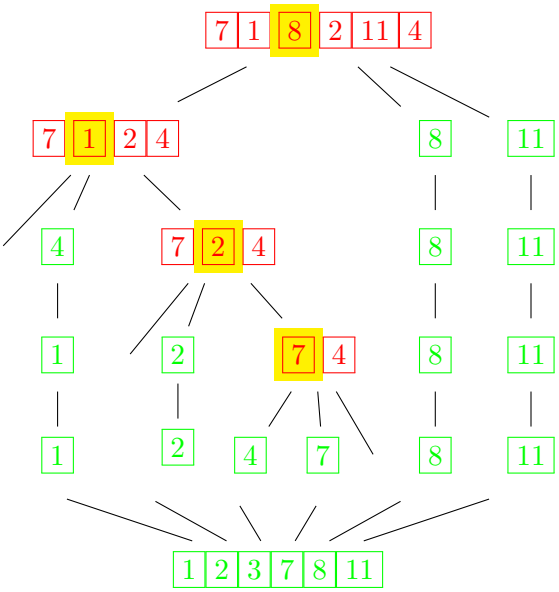


4.7 Quicksort

4.7.1 Vorgehen:

Dieses Sortierfverfahren nutzt Divide and Conquer: Es teilt das Array auf und fügt sie anschließend wieder zusammen. So entstehen kleinere, einfache Probleme. Der Aufwand liegt hier beim Teilen der Listen, dafür ist das zusammenführen sehr einfach. Es wird ein beliebiges Element aus dem Element ausgewählt (meist zufällig, hier das Mittlere), das sog. *Pivot-Element*. Anhand dieses Elements wird das Array in drei Teile geteilt: Die kleiner, gleich und größer als das *Pivot-Element* sind. Dabei werden die Teillisten nicht sortiert. Das wird solange, bis jede Liste in einelementige, sortierte Listen zerfallen sind. Anschließend werden beim Mergen alle Listen „von links nach rechts“zusammengefügt.

4.7.2 Anwendungsbeispiel:



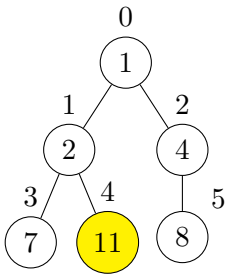
4.8 Heapsort

4.8.1 Vorgehen

Zum Sortieren wird ein Heap benutzt. Das ist letztendlich ein vollständiger binärer Baum, für den für jeden Knoten gilt, dass seine Kindsknoten kleiner oder gleich sind. Vollständig ist ein Baum, wenn jede Ebene komplett und die ggf. nicht volle letzte Ebene von links nach rechts gefüllt ist. Durch diese Heap-Konstruktion ist das kleinste Element immer oben an der Wurzel.
Die Operation *heapify* stellt für einen Knoten die Heap-Eigenschaften her. Sie tauscht bei Verletzung so oft Eltern- und Kindsknoten, bis kein Knoten die Heap-Eigenschaften verletzt.
Die andere Funktion ist das Löschen der Wurzel. Dafür wird die Wurzel mit dem untersten-rechten Element ersetzt und mit *heapify* die Heap-Eigenschaft wiederhergestellt.
Für den Heapsort wird ein Heap erstellt und schrittweise die Wurzel gelöscht, bis der Heap leer ist. Die Wurzeln werden somit in aufsteigender Reihenfolge entfernt.

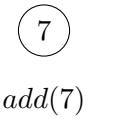
4.8.2 Darstellung als Array:

Heaps sind vollständig, somit können sie ebenweise, von links nach rechts durchlaufen werden, ohne ein Loch zu finden.
Außderdem hält jede volle Ebene auf mit Tiefe h 2^h Elemente (Wurzel hat Tiefe 0). Das x-te (von 0) Element auf Ebene h hat bekommt somit den Index $2^h + x$.
Der Heap kann so durch ein Array realisiert werden, wobei jedes Element wie eben beschrieben addressiert wird.

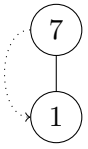


Index $= 2^2 + 0 = 4$✓

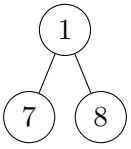
4.8.3 Anwendungsbeispiel: „Heapen“des Array



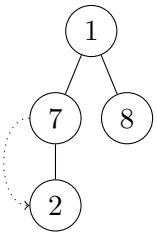
$add(7)$



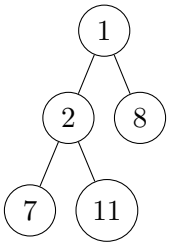
$add(1)$, $reheap()$



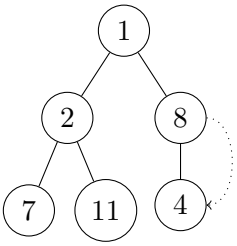
$add(8)$



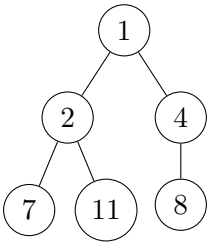
$add(2)$, $reheap()$



$add(11)$

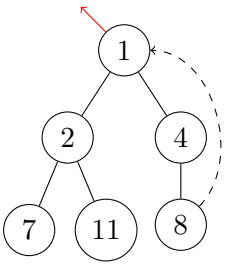


$add(4)$, $reheap()$



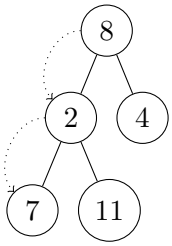
Fertig

Anwendungsbeispiel: Sortieren



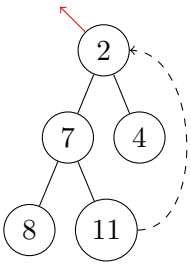
$remove(1)$

1



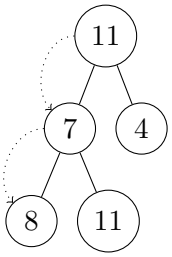
$reheap()$

1



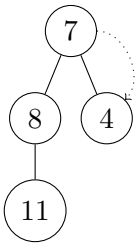
$remove(2)$

1	2



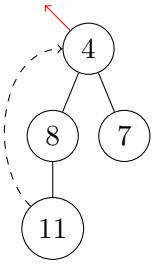
$reheap()$

1	2



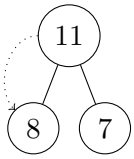
$reheap()$

1	2



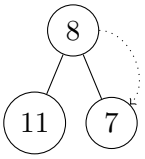
$remove(4)$

1	2	4



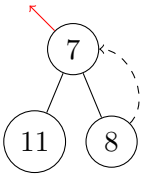
$reheap()$

1	2	4



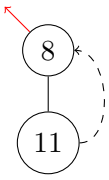
$reheap()$

1	2	4



$remove(7)$

1	2	4	7



$remove(8)$

1	2	4	7	8

$remove(11)$

1	2	4	7	8	11

Heap leer ⇒ Fertig!

1	2	4	7	8	11

4.9 Laufzeiten der Sortierverfahren[Q1]:

	best case	*average case*	*worst case*
Selectionsort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertionsort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Bubblesort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Mergesort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$
Quicksort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n^2)$
Heapsort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$

5 Quellen: (Stand: 07.07.2019)

Q1: `https://de.wikipedia.org/wiki/Sortierverfahren#Vergleichsbasiertes_Sortieren`