Inhalt

12 Heaps und Prioriäts-Warteschlangen
- Min- und Max-Heaps
- Prioritäts-Warteschlangen
- HeapSort (*)

Heap: Vorbemerkung

- Wir betrachten einen Heap als eine spezielle Datenstruktur
- in zwei Varianten: MinHeap oder MaxHeap.
- Der Begriff „Heap“ wird (in einem anderen Zusammenhang!) auch als Bezeichnung für eine speziellen Speicherbereich verwendet.
  Das hat nichts (bzw. nur wenig) mit dem hier verwendeten Begriff zu tun. $\leadsto$ nicht verwechseln!

Partiell geordneter Baum

Ein partiell geordneter Baum

- ist ein binärere Baum (mit Einträgen aus einem „vergleichbaren“ Typ)
- mit der zusätzlichen „Heap-Eigenschaft“, dass für jeden Teilbaum gilt:
    - MinHeap:
      Der Wurzelknotenwert ist das Minimum der Werte aller Knotenwerte.

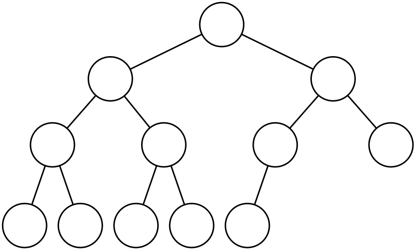
  oder

    - MaxHeap:
      Der Wurzelknotenwert ist das Maximum der Werte aller Knotenwerte.

DS Heap

Ein (Min- oder Max-) Heap ist eine Datenstruktur zur Darstellung

- von partiell geordneten Bäumen,
- die zudem links-vollständig sind.
  Das bedeutet, dass alle Ebenen bis auf die letzte voll besetzt sind und in der letzten Ebene alle Knoten so weit links wie möglich sitzen.
- Die Höhe eines links-vollständigen Baums mit n Knoten ist in $\mathcal{O}(\log n)$

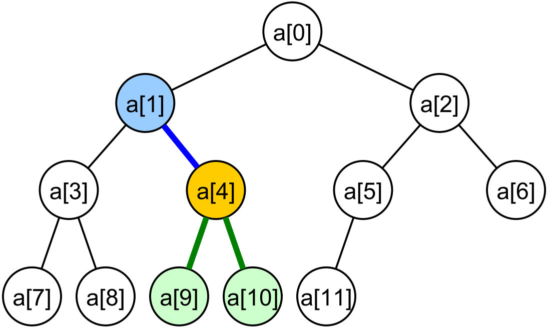


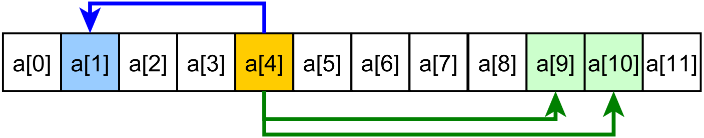
Array-Implementierung

Links-vollständige Bäume lassen sich leicht in einem Array implementieren, denn die Array-Indices der Söhne bzw des Vaters eines Knoten lassen sich leicht aus dem Index des Knotens selber berechnen:

- Wenn die Knoten eines links-vollständigen Baums in „bfs-Reihenfolge“ in ein Array a eingefügt werden, dann gilt:
- $a[0] = root$
- für einen Knoten $k = a[p]$ an der Position p gilt:
  - die Position des Vaters von k ist $(p-1)/2$
  - die Position des linken Sohns von k ist $2 \cdot p + 1$
  - die Position des rechten Sohns von k ist $2 \cdot (p+1)$

Array-Implementierung





Bemerkung und Warnung

Heaps sind keine Suchbäume!

- Die Suche nach einem beliebigen Element (zB `contains(T e)`) wird nicht besonders gut unterstützt.

Aber:

- Der Zugriff auf das kleinste (bzw das größte) Element ist in $\mathcal{O}(1)$ möglich!
  Kosten für `getMin()`, `getMax()` : $\mathcal{O}(1)$

Frage:

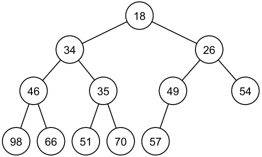
- Lassen sich `insert (T e)` und `remove()` effizient implementieren, sodass
  1. die Heap-Eigenschaft erfüllt bleibt:
     In jedem Teilbaum ist der Wert des Wurzelknotens das Minimum (bzw Maximum) der Werte aller Knoten des Teilbaums.
  2. der Baum linksvollständig bleibt?

Einfügen in einen Heap: upheap

Methode **void** insert (T e):

- Füge den neuen Wert $k.val$ in einem neuen Knoten k an der nächsten freien Position des Baums ein. ($\rightsquigarrow$ Links-Vollständigkeit)
- Stelle durch „upheap-Operation" die Heap-Eigenschaft wieder her:
  - sei f der Vaterknoten von k (sofern $k \neq root$)
  - solange ($k! = root$ und $k.val < f.val$) (bei MinHeap)
    bzw solange ($k! = root$ und $k.val > f.val$) (bei MaxHeap)
    vertausche k mit seinem Vaterknoten f

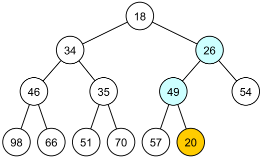
Beispiel MinHeap: insert mit upheap



a[11]

18	34	26	46	35	49	54	98	66	51	70	57

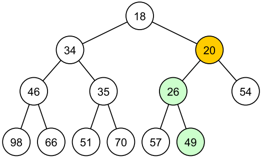
$\Longrightarrow$ **insert(20)**



a[5] a[12]

18	34	26	46	35	49	54	98	66	51	70	57	20

$\Longrightarrow$ **upheap(20)**



a[2] a[5]

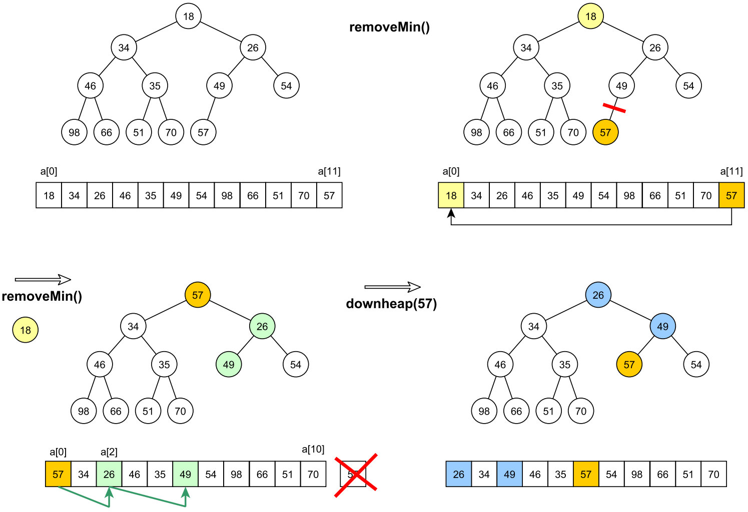
18	34	20	46	35	26	54	98	66	51	70	57	49

Entfernen aus einem Heap: downheap

Methode T remove():

- Speichere den Wurzelwert für die spätere Rückgabe des Wertes.
- Entferne den „letzen“ Knoten des Baums (in der untersten Ebene der am weitesten rechts stehende Knoten) und setze ihn an die Stelle der Wurzel. ($\rightsquigarrow$ Links-Vollständigkeit)
- Stelle durch „downheap-Operation“ die Heap-Eigenschaft wieder her:
  - dh bei MinHeap:
    solange $(k.val > k.lbt.val$ oder $k.val > k.rtb.val)$
    vertausche k mit dem kleineren seiner Söhne
  - dh bei MaxHeap:
    solange $(k.val < k.lbt.val$ oder $k.val < k.rtb.val)$
    vertausche k mit dem größeren seiner Söhne

Beispiel MinHeap: remove() mit downheap



Kosten von insert() und remove()

- Garantie der Links-Vollständigkeit:
  - Das Einsetzen des neuen Wertes an die „letzte“ Posititon bzw
  - das Ersetzen des Wurzelknotens durch den „letzten“ Knoten und dessen Entfernen

  verursacht konstante Kosten: $\mathcal{O}(1)$.
- upheap:
  - Der neu eingefügte Knoten k steigt im Heap nach oben auf, bis die Heap-Eigenschaft wieder hergestellt ist.
  - upheap verfolgt einen Pfad im Baum vom neuen Blatt bis maximal zur Wurzel:
  - insert (T e) arbeitet in $\mathcal{O}(\log n)$
- downheap:
  - Der neue Wurzelwert sinkt im Heap nach unten, bis die Heap-Eigenschaft wieder hergestellt ist.
  - downheap verfolgt einen Pfad im Baum von der Wurzel bis maximal zur letzten Ebene:
  - remove() arbeitet in $\mathcal{O}(\log n)$

Inhalt

Priorität von Objekten

Wir betrachten Sammlungen von Objekten, denen als Priorität ein „Schlüssel“ („key“) zugeordnet ist.

- Die Priorität ist ein numerischer Wert,
- so dass eine totale Ordnungsrelation definiert ist und
- sich Elemente mit höchster Priorität bestimmen lassen.

Anmerkung:

- Prioritäten müssen nicht eindeutig sein, es kann verschiedene Elemente mit gleicher Priorität geben.
- Wir vergeben natürliche Zahlen als Priorität.
- Wir verstehen einen „kleineren Wert“ als „höhere Priorität“.
  Ein Element mit höchster Priorität ist ein Element mit minimalem key.

ADT PrioSchlange

Neben den „Standard“-Operationen **int** size() und **boolean** isEmpty() definiert man für den ADT PrioSchlange folgende Operationen:

- T getMin():
  liefert das Element mit höchster Priorität - also den Eintrag mit minimalem Schlüsselwert (ohne es zu entfernen)
- **void** insert(T v, **int** k):
  fügt ein Element *v* mit Schlüssel *k* in die PrioSchlange ein
- **void** removeMin():
  liefert das Element mit höchster Priorität - also den Eintrag mit minimalem Schlüsselwert - und entfernt es aus der Schlange.

Bemerkung:

Manchmal werden T getMin() und **void** removeMin() auch in einer Methode T removeMin() zusammengefasst, die das Element liefert und entfernt.

Implementierungen und ihre Kosten

Zur Implementierung bieten sich alle Datenstrukturen an, die eine Sortierung der Elemente beim Einfügen erlauben, zB

Kosten für	getMin()	insert ()	removeMin()
(Sortiertes) Array	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
(Sortierte) verkettete Liste	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
(balancierter) Suchbaum	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
MinHeap	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Inhalt

12 Heaps und Prioriäts-Warteschlangen

- Min- und Max-Heaps
- Prioritäts-Warteschlangen
- HeapSort (*)

Prinzip: Sortieren durch Auswählen

HeapSort ist eine Spezialform des SelectionSort

- gegeben: unsortierte Folge *a*
- Ziel: sortierte Folge *b*
- Prinzip:
  - solange *a* nicht leer, wiederhole
    - wähle aus *a* das Minimum aus
    - füge das Minimum in *b* ein (hinten)
- speziell: nutze einen Heap, um schnell das Minimum in *a* zu finden

„Naiver“ HeapSort mit MinHeap

- (insert-Phase): solange a nicht leer, wiederhole
  - entnimm das nächste Element aus a
  - füge es in einen (anfangs leeren) Heap h ein (mit ggf. notwendigen upheap-Operationen)
- (removeMin-Phase): solange h nicht leer, wiederhole
  - entnimm das Minimum aus h
  - füge es in die Folge b ein

Kosten:

- insert-Phase: $n \cdot \mathcal{O}(\log(n)) = \mathcal{O}(n \log n)$
- removeMin-Phase: $n \cdot \mathcal{O}(\log(n)) = \mathcal{O}(n \log n)$

Idee: Verfeinerung des HeapSort: mit MaxHeap

Falls der Heap ausschließlich zum Sortieren einer fest gegeben Folge (bzw eines Arrays) benutzt wird, kann man etwas geschickter vorgehen, denn ...

- die Größe des Heaps ist durch die Länge des Arrays bereits gegeben,
- daher kann das (unsortierte) Array bereits als Darstellung eines links-vollständigen Baums betrachtet werden.
- Betrachte das Array als Darstellung eines Max-Heaps:
- Stelle die Heap-Eigenschaft in den inneren Knoten durch downheap-Operationen her.

In-Situ-HeapSort

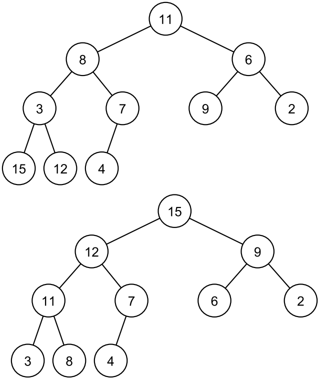
„In Situ“ bzw. „in-place“:
Sortieren eines Arrays ohne zusätzichen Platzbedarf!

Verfahren in zwei Phasen:

1. Aufbau des Max-Heaps aus dem Ursprungs-Array a:
   - Interpretiere das (unsortierte) Array a als linksvollständigen Baum
   - Führe von hinten nach vorne für jeden inneren Knoten die Operation downheap durch
     dh für die Elemente $a[(a.length - 1)/2], \ldots, a[0]$
2. „Auslesen“ des Max-Heaps:
   - Wiederhole $(a.length - 1)$-mal:
     - Führe removeMax() mit anschließendem downheap aus ...
     - ... wobei in jedem Schritt das maximale Element an das Ende des Arrays getauscht wird,
     - und im folgenden Schritt der betrachtete Array-Ausschnitt um 1 verkürzt wird.

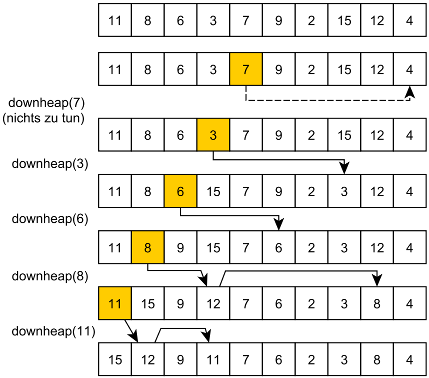
Beispiel HeapSort (1): Aufbau des MaxHeaps

linksvollständiger Baum



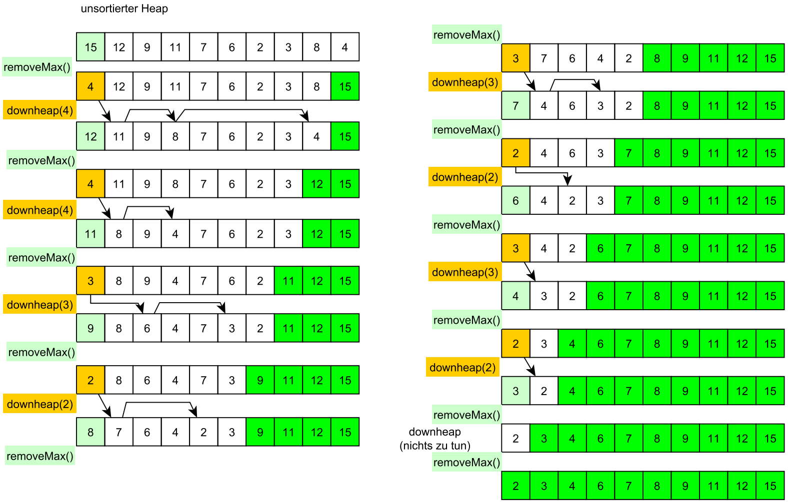
unsortierter Heap

unsortiertes Array



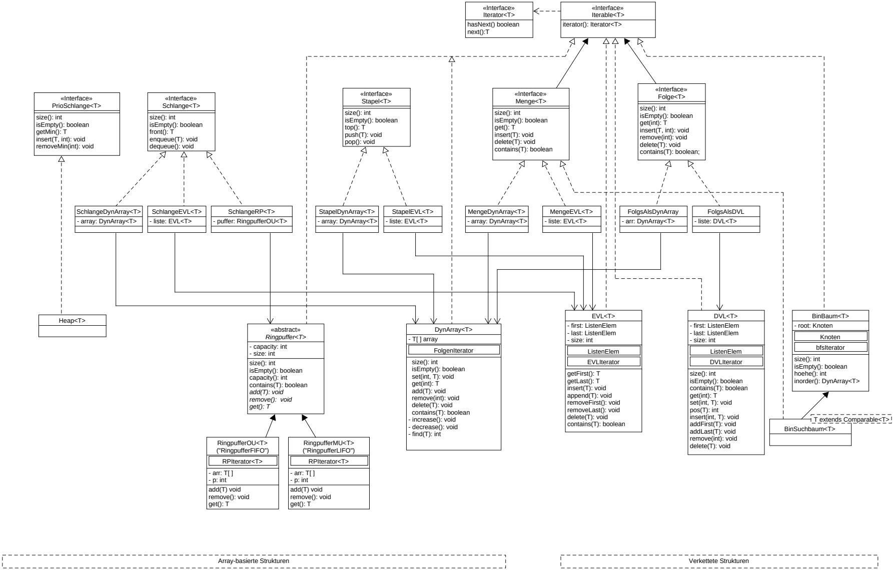
unsortierter Heap

Beispiel HeapSort (2): Auslesen des MaxHeaps



Inhalt

UML - final


«Interface»
Iterator<T>
hasNext() boolean
next():T
«Interface»
Iterable<T>
iterator(): Iterator<T>
«Interface»
PrioSchlange<T>
size(): int
isEmpty(): boolean
getMin(): T
insert(T, int): void
removeMin(int): void
«Interface»
Schlange<T>
size(): int
isEmpty(): boolean
front(): T
enqueue(T): void
dequeue(): void
«Interface»
Stapel<T>
size(): int
isEmpty(): boolean
top(): T
push(T): void
pop(): void
«Interface»
Menge<T>
size(): int
isEmpty(): boolean
get(): T
insert(T): void
delete(T): void
contains(T): boolean
«Interface»
Folge<T>
size(): int
isEmpty(): boolean
get(int): T
insert(T, int): void
remove(int): void
delete(T): void
contains(T): boolean;
SchlangeDynArray<T>
- array: DynArray<T>
SchlangeEVL<T>
- liste: EVL<T>
SchlangeRP<T>
- puffer: RingpufferOU<T>
StapelDynArray<T>
- array: DynArray<T>
StapelEVL<T>
- liste: EVL<T>
MengeDynArray<T>
- array: DynArray<T>
MengeEVL<T>
- liste: EVL<T>
FolgsAlsDynArray
- arr: DynArray<T>
FolgsAlsDVL
- liste: DVL<T>
Heap<T>
«abstract»
Ringpuffer<T>
- capacity: int
- size: int
size(): int
isEmpty(): boolean
capacity(): int
contains(T): boolean
add(T): void
remove(): void
get(): T
RingpufferOU<T>
("RingpufferFIFO")
RPIterator<T>
- arr: T[ ]
- p: int
add(T) void
remove(): void
get(): T
RingpufferMU<T>
("RingpufferLIFO")
RPIterator<T>
- arr: T[ ]
- p: int
add(T) void
remove(): void
get(): T
DynArray<T>
- T[ ] array
FolgenIterator
size(): int
isEmpty(): boolean
set(int, T): void
get(int): T
add(T): void
remove(int): void
delete(T): void
contains(T): boolean
- increase(): void
- decrease(): void
- find(T): int
EVL<T>
- first: ListenElem
- last: ListenElem
- size: int
ListenElem
EVLIterator
getFirst(): T
getLast(): T
insert(T): void
append(T): void
removeFirst(): void
removeLast(): void
delete(T): void
contains(T): boolean
DVL<T>
- first: ListenElem
- last: ListenElem
- size: int
ListenElem
DVLIterator
size(): int
isEmpty(): boolean
contains(T): boolean
get(int): T
set(int, T): void
pos(T): int
insert(int, T): void
addFirst(T): void
addLast(T): void
remove(int): void
delete(T): void
BinBaum<T>
- root: Knoten
Knoten
bfsIterator
size(): int
isEmpty(): boolean
hoehe(): int
inorder(): DynArray<T>
T extends Comparable<T>
BinSuchbaum<T>
Array-basierte Strukturen
Verkettete Strukturen

Fazit

- Die Abstrakten Datentypen
  Folge, Menge, Stapel, Schlange, Prioritäts-Warteschlange
  verfügen - in verschiedenen Varianten - alle über Operationen zum
  - Zugriff auf ein Element: get(), contains(), getMin(), front(), top()
  - Einfügen von Elementen: insert(), push(), enqueue()
  - Löschen von Elementen:
    delete(), remove(), pop(), dequeue(), removeMin()
- Zur Implementierung bieten sich unterschiedliche Datenstrukturen an:
  Dynamisches Array, Verkettete Liste, Suchbaum, Ringpuffer, Heap
- Je nach gewählter Datenstruktur haben die Implementierungen der Methoden unterschiedliche (worst-case-) Laufzeit.

Ihre Zukunft?

Aufgabe für Software-Entwickler:innen:

- Je nach Anforderungen des speziellen Anwendungsfalls
- wollen wir einen geeigneten Datentyp,
- dh. die „beste“ Kombination aus ADT und Datenstruktur

finden.

Viel Glück dabei ;)