

2. CoMa Übung

17.4.

Heaps, (Iteratoren)

Heaps: Datenstruktur in Baumform mit folgenden Eigenschaften:

- binärer Baum
- Voller Baum, alle (bis auf letzte Schicht) sind voll besetzt

↯ vollständig: jeder Knoten hat 0 oder 2 Kinder
→ Heap ist kein vollst. Baum

- Schlüssel in jedem Knoten ist größer als die seiner Kinder



Operationen:

- extractMax/pop/poll: max. Element entfernen, $O(\log n)$
- top/max/peek: Wert des max. Elements zurückgeben, $O(1)$
- insert/push/offer/add: neues Element einfügen, $O(\log n)$

manchmal:

- delete/remove: entfernt Element mit gegebenem Schlüssel
- decrease key/: ändert Schlüssel für Element mit gegeb. Schlüssel
in — " —

Java bietet:

Class PriorityQueue<E>

```

java.lang.Object
├── java.util.AbstractCollection<E>
│   ├── java.util.AbstractQueue<E>
│   └── java.util.PriorityQueue<E>

```

Type Parameters:

E - the type of elements held in this collection

All Implemented Interfaces:

[Serializable](#), [Iterable<E>](#), [Collection<E>](#), [Queue<E>](#)

```

public class PriorityQueue<E>
extends AbstractQueue<E>
implements Serializable

```

An unbounded priority [queue](#) based on a priority heap. The elements of the priority queue are ordered according to their [natural ordering](#), or by a [Comparator](#) provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in [ClassCastException](#)).

Implementation note: this implementation provides $O(\log(n))$ time for the [enqueueing](#) and [dequeueing](#) methods ([offer](#), [poll](#), [remove\(\)](#) and [add](#)); linear time for the [remove\(Object\)](#) and [contains\(Object\)](#) methods; and constant time for the [retrieval methods](#) ([peek](#), [element](#), and [size](#)).

Constructor Summary

PriorityQueue()

Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their [natural ordering](#).

PriorityQueue(Collection<? extends E> c)

Creates a PriorityQueue containing the elements in the specified collection.

Kopiert anderen Kontainer

PriorityQueue(int initialCapacity)

Creates a PriorityQueue with the specified initial capacity that orders its elements according to their [natural ordering](#).

PriorityQueue(int initialCapacity, Comparator<? super E> comparator)

Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.

spezielle Sortierfkt.

Method Summary

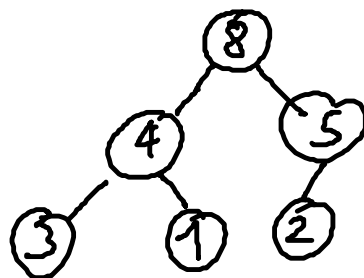
boolean	add(E e)	<i>einfügen</i> Inserts the specified element into this priority queue.
void	clear()	Removes all of the elements from this priority queue.
Comparator<? super E>	comparator()	Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements.
boolean	contains(Object o)	Returns true if this queue contains the specified element.
Iterator<E>	iterator()	Returns an iterator over the elements in this queue.
boolean	offer(E e)	<i>einfügen</i> Inserts the specified element into this priority queue.
E	peek()	<i>max. element ansehen</i> Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E	poll()	<i>max. element entfernen</i> Retrieves and removes the head of this queue, or returns null if this queue is empty.
boolean	remove(Object o)	Removes a single instance of the specified element from this queue, if it is present.
int	size()	Returns the number of elements in this collection.
Object[]	toArray()	

Returns an array containing all of the elements in this queue.

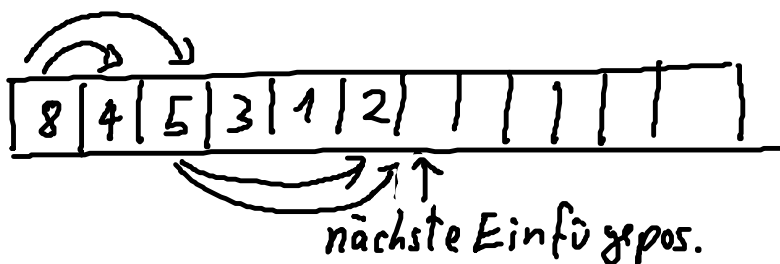
<T> T[] toArray(T[] a)

Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array.

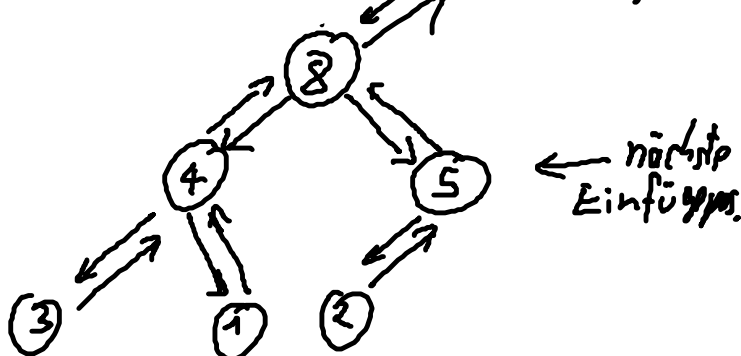
Beispiel:



in VL: als Array



in PA: als Baum dummy



Vorteile: • Baumstruktur implizit gegeben.
(weniger Fehler, weniger Speicher)

• Größe beliebig

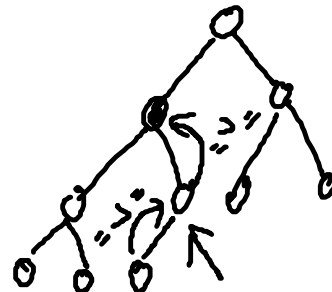
Nachteile: • Größe statisch
→ Kopieren bei Überschreiten der Kapazität nötig

• Aufwand für Konsistenz der Baumstruktur
• Speicherbedarf größer

Operationen in Baum-Variante:

Einfügen:

- Einfügeposition suchen (Vater des neuen Knotens)
- Knoten als Kind anhängen
- Knoten aufsteigen lassen, bis Vater > akt. Knoten



ist (siftUP())

SIFTUP(Node n):

WHILE (n.key > n.parent.key)

 tausche n und n.parent

 n := n.parent

ENDWHILE

tausch der Daten in
schlüssel aus reiheord

Maximum entfernen:

- Maximum ist in Wurzel
- Wurzel mit "letztem" Element tauschen
- neue Wurzel absinken lassen (heapify())



HEAPIFY(Node n):

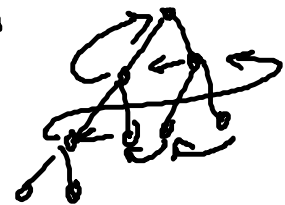
WHILE n.key < max(n.left.key, n.right.key) DO

 tausche n und argmax(n.left.key, n.right.key)

 n := argmax(n.left.key, n.right.key)

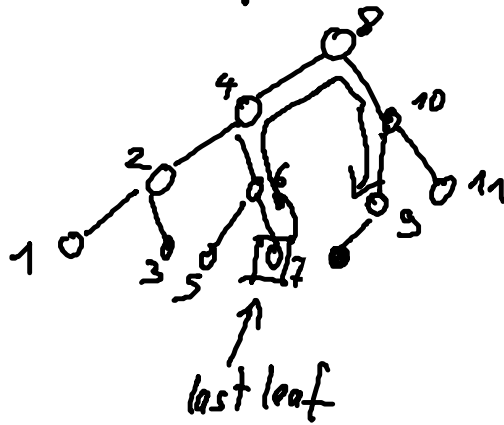
ENDWHILE

Heap erzeugen: - n x 1 Element in leeren Heap einfügen
(oder) - im Baum „rückwärts“ schichtweise für jeden Knoten heapify() aufrufen



Implementation details:

- Wo ist die neue Einfüge-Position:



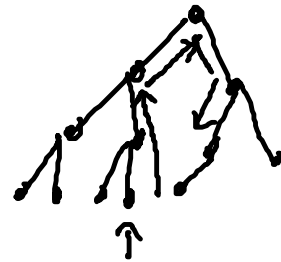
Inorder nummerierung

Fall a) last leaf ist linkes Kind:
 - neuer Knoten ist rechtes Kind vom Vater von last leaf



b) Last leaf ist rechtes Kind:

- 2 Inorder Iterationen
 oder: nextInsert(last leaf)
 nextInsert(Node n):



WHILE n ist rechtes Kind DO
 n := n.parent
 ENDWHILE

} 1 mal Inorder Inkrement

IF n ≠ root DO
 n := n.parent
 n := n.rightChild
 ENDIF

} 1 mal Inorder Inkrement

WHILE n ist kein Blatt DO
 n := n.leftChild
 ENDWHILE

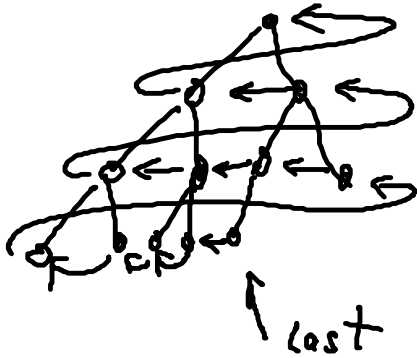
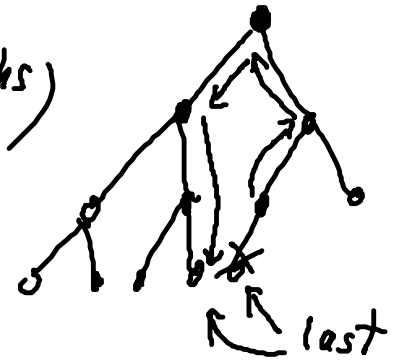
→ neuer Knoten als linkes Kind

— Wo finde ich das vorherige Last-leaf beim Löschen?

— 2 mal Inorder rückwärts,

→ nextInsert() abwandeln, (Rechts und Links tauschen)

andere Methode: zusätzlichen Zeiger



beim Einfügen auf
alten Last leaf zeigen

⇒ schwieriger: BinTreeNode zu
HeapNode erweitern
um neuen Zeiger einzuführen

Programmierhinweise:

- sort(): bekommt Array T[], gibt ^{das selbe} sort. Array zurück
→ Fehlermeldung falls Heap am Anfang nicht leer

falls: Array T[] erstellung nötig

~~T[] arr = new T[4];~~

T[] arr = (T[]) new Object[4];

- " \leq " größer: a.compareTo(b): $< 0 \cong$ a kleiner b
 $= 0 \cong$ a und b gleich
 $> 0 \cong$ a größer b

- Knoten tauschen: Daten tauschen reicht aus