Vorlesung 29

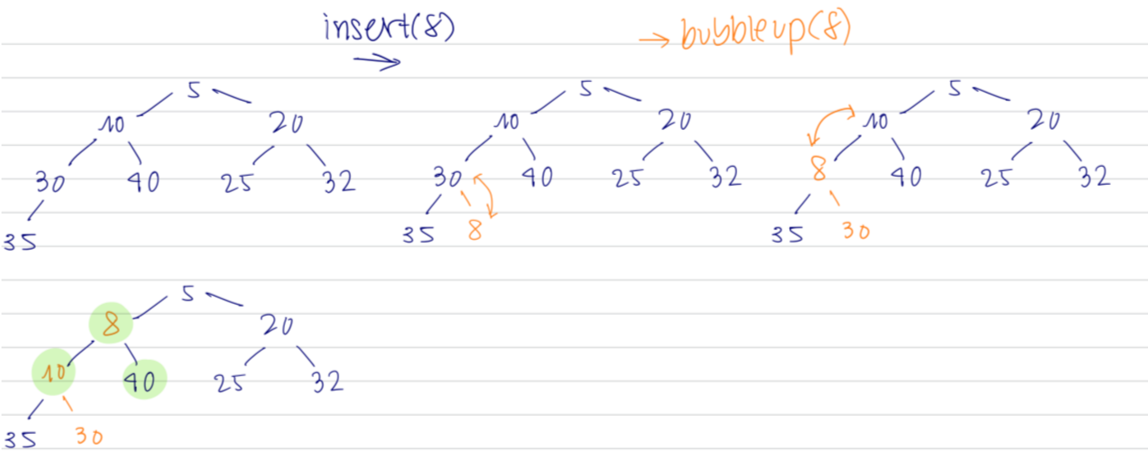
Binärer Heap

Ein binärer Heap ist ein Binärbaum mit:

- Vollständigkeitseigenchaft
- Ordnungseigenschaft: Element in Knoten ist ≤ der Elemente in den Kindern

Einfügen in einen binären Heap

1. Füge Element in unterster Ebene in linkesten freien Knoten ein → jetzt gilt Vollständigkeitseigenschaft
2. `bubbleUp(i)`: Tausche so lange mit Wert im Elternknoten bis der Wert im Elternknoten kleiner ist → anschließend gilt die Ordnungseigenschaft



Rekursive Implementierung

```
```

Imperative Implementierung

```
```

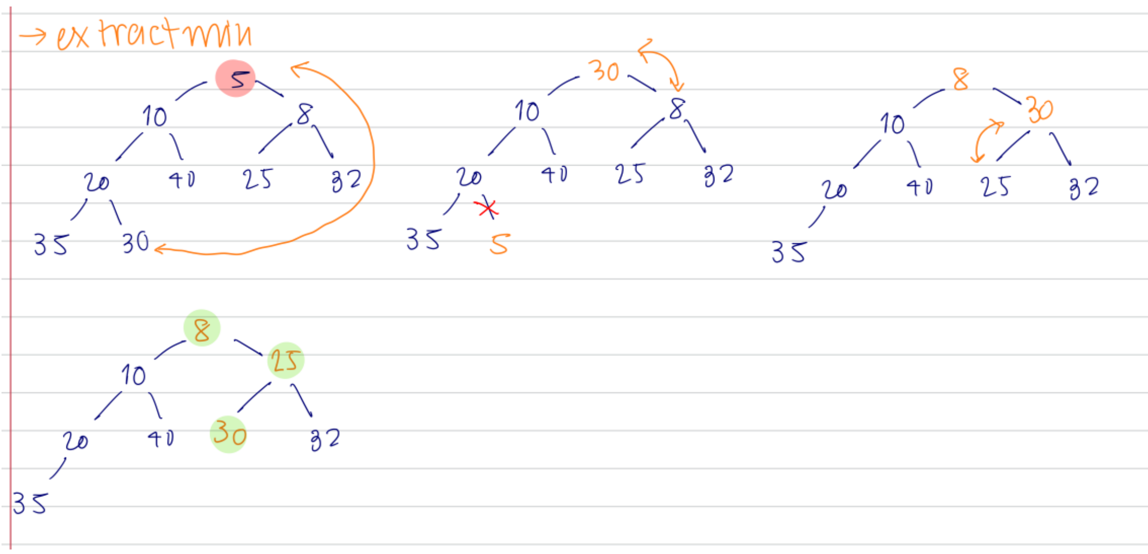
Laufzeit

Im schlechtesten Fall wird ein neues Minumum eingefügt und bubble up endet entsprechend in der Wurzel. Das sind h Schritte für die Höhe des BinBaums, also entspricht die Laufzeit $O(h) \in O(logn)$

Löschen des Minimums aus einem binären Heap

Um das Minimum zu löschen/extrahieren müsste man die Wurzel löschen. Das würde aber den Baum zerstören. Stattdessen:

1. Tausche Element in der Wurzel mit untersten rechtesten Blatt und lösche dieses
2. `bubbleDown(i):` Tausche so lange mit kleinerem Kind, bis beide Kinder größer sind.



Laufzeit

Da der erste Schritt des Tausches $O(1)$ hat dominiert due Laufzeit von `bubbleDown(i)`. Wie bei `bubbleUp(i)` muss im worst-case entlang der gesamten Baumhöhe getauscht werden, hier von oben nach unten. Das heißt die Laufzeit ist $O(h) \in O(logn)$

Rekursive Implementierung

```
```

Imperative Implementierung

```
```

Nochmal sortieren mit Prioritätswarteschlangen

In einer Implementierung mit `PrioQueue` haben wir einen vergleichsbasierten Sortieralgorithmus. → Erst werden alle Elemente eingefügt und dann wieder gelöscht und eine sortierte Liste wird zurück gegeben. Weil es ein vergleichsbasierter Sortieralgorithmus ist, ist der best-case $O(nlogn)$. Des wegen muss einer der beiden Algorithmen (`insert(i)` oder `extractMin()`) amortisiert eine Laufzeit von $O(logn)$ haben.

Scala Code für allgemeine Implementierung:

```
def prioQueuesort[K:Ordering](array = Array[K])Unit =
    val n = array length
    val pq = new PrioQueueImp[K]()
    for a <- array do pq insert(a)
    for i <- 0 to n-1 do array(i) = pq extractMin()
```

Allgemeine Laufzeit

- $T_{in}(n)$: Laufzeit für `insert` in PQ der Größe n
- $T_{out}(n)$: Laufzeit für `extractMin` auf PQ der Größe n

Dann entspricht die allgemeine Laufzeit der obigen Implementierung:

$$T(n) = c + T_{in}(0) + T_{in}(1) + ... + +T_{in}(n-1) + T_{out}(n) + +T_{out}(n-1) + ... + +T_{out}(1)$$

Wobei c → Konstante Laufzeit für die ersten beiden Zeilen, und dann die Folgenden Summen entsprechen den Laufzeiten für die `insert` und `extractMin` Algorithmen, bei denen `insert` erst in eine leere und dann sich nach und nach füllenden Heap durchgeführt wird und `extractMin` bei einem vollen Heap beginnt, welcher sich dann aber immer weiter leert.

Man kann also einfacher schreiben:

$$= c + \sum_{i=0}^{n-1} T_{in}(i) + \sum_{i=1}^{n} T_{out}(i) = c + \sum_{i=1}^{n} T_{in}(i-1) + T_{out}(i)$$

$$\leq c + n * (T_{in}(n) + T_{out}(n)) \in O(n * (T_{in}(n) + T_{out}(n))$$

Verkettete Knoten

Sortierte Reihenfolge

$T_{in}(n) \in O(n)$ und $T_{out}(n) \in O(1)$ → Sortiert wird in $O(n^2)$ prinzipiell und von der Laufzeit entspricht dies Insertionsort.

Unsortierte Reihenfolge

$T_{in}(n) \in O(1)$ und $T_{out}(n) \in O(n)$ → Sortiert wird dann in $O(n^2)$ und entspricht von der Durchführung Selectionsort