universität freiburg

Algorithmen und Datenstrukturen SS 2025

Vorlesung 9: Prioritätswürgeschlangen

Dienstag, 1. Juli 2025

Prof. Dr. Hannah Bast
Professur für Algorithmen und Datenstrukturen
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

Blick über die Vorlesung heute

- Organisatorisches
  - Erfahrungen mit dem Ü8 — Suchbäume + Adressensuche
  - Vorgucker auf das Ü9 — Implementieren + Beweisen
  - Offizielle Evaluation — Es gibt sogar Ü-Punkte dafür
- Inhalt
  - Grundlagen — Definition einer Priority Queue und Anwendungsbeispiele
  - Binärer Heap — Speicherung und Reparieren
  - Implementierung der Op. — insert, get_min, delete_min, …
  - Laufzeit + Optimierungen — Analyse, Heapify, Fibonacci Heaps
  - Bibliotheken — Python, Java, C++

Erfahrungen mit dem Ü8 1/2

- Auszüge aus Ihrem Feedback [halbwegs repräsentativ]

  "Ein Lichtblick, das Blatt hat mega Spaß gemacht"

  "Unfassbar Spaß gemacht, VL gut strukturiert und erleuchtend"

  "Hat sehr viel Spaß gemacht; hat zwar Zeit gekostet, weil ich immer wieder knobeln musste, aber gerade deshalb cool"

  "Vorlesung top, alles gut erklärt und gute Beispiele wie immer"

  "War ein sehr tolles Blatt; Praxisnähe erlaubt es, die Wichtigkeit von AlgoDat und vor allem Laufzeiten Wert zu schätzen"

  "Vorlesung war super verständlich, erase sehr gut erklärt"

  "Coole Idee, aber auch ziemlich aufwendig und komplex"

  "7 Stunden gebraucht, ich war etwas verwirrt von mir selbst"

  "Erst froh wieder zu coden, aber dann hat es nicht funktioniert"

Erfahrungen mit dem Ü8 2/2

- Wo geopolitische Lage auf der Liste Ihrer Sorgen?

  "Persönliche Themen wie Studium stehen im Vordergrund"

  "Nice to know aber hat sonst keine Auswirkungen auf mich"

  "Nicht wirklich, weil ich recht uninformiert bin"

  "Platz 2 nach kommender AlgoDat Klausur"

  "Beschäftigt mich, aber beherrscht nicht dauernd meine Gedanken"

  "Ich würde sagen, so 8192 von 12500 nuklearen Sprengköpfen"

  "Das steht weit oben auf der Liste meiner Sorgen"

  "Ist auf meiner Sorgenliste so hoch wie noch nie"

  "Ziemlich, als ehemaliger Soldat bekommt man nochmals mehr mit, gerade wenn man erfährt, dass diese und jene gefallen sind, welche man vor einem Jahr noch ausgebildet hat" [Ukrainekrieg]

Vorgucker auf das Ü9

- Eine schöne Mischung aus Theorie und Praxis

  A1 Fortsetzung der Implementierung aus der Vorlesung

  Sehr wenig Code; ein bisschen knifflig; aber sollte leichter sein als der binäre Suchbaum vom Ü8

  A2 Heapify = n Elemente auf einen Schlag in eine PQ

  Ebenfalls sehr wenig Code und ein bisschen (nicht zu) knifflig

  A3 Beweis, dass heapify korrekt ist

  Mit Hinweis nicht schwer, den muss man aber erst entschlüsseln; für die, die es erstmal ohne Hinweis probieren wollen

  A4 Beweis, dass heapify in Zeit O(n) läuft

  Sollte mit ein bisschen Nachdenken gut machbar sein

Offizielle Evaluation der Veranstaltung

- Läuft über das zentrale **EvaSys** der Uni
  - Sie sollten am Montag (30. Juni) eine Mail bekommen haben

    Falls nicht, bitte kurz auf dem Forum Bescheid geben, wir haben dafür ein Unterforum "Evaluation" eingerichtet
  - Nehmen Sie sich bitte Zeit und seien Sie **ehrlich**, **konkret**, **fair**, **ausgeschlafen** und **freundlich**

    Sie haben soviel Zeit in die Vorlesung investiert, dann können Sie auch 20 Minuten für die Evaluation aufwenden

    Sie bekommen außerdem 20 Punkte dafür, die die Punkte vom schlechtesten Übungsblatt ersetzen … siehe Folien V1
  - Die **Freitextkommentare** sind besonders interessant für uns

Grundlagen 1/9

Auf Englisch:
Priority Queue

- Definition
  - Eine Prioritätswarteschlange (PW bzw. PQ) verwaltet eine Menge von Elementen, die (wie gehabt) Key-Value Paare sind, mit einer totalen Ordnung $\leq$ auf den Keys
  - Eine PQ sollte die folgende Operationen unterstützen:

insert(item):	füge das gegebene Element ein
get_min():	gebe Element mit kleinstem Key zurück
delete_min():	entferne Element mit dem kleinsten Key
change_key(item):	ändere Key des gegebenen Elementes
remove(item):	entferne das gegebene Element

Wofür das gut ist, sehen wir gleich, man braucht es tatsächlich in der Praxis sehr oft

Grundlagen 2/9

- Vergleich mit Hash Map und binärem Suchbaum
  - Bei der **Hash Map** (Vorlesung 5) sind die Keys in keiner besonderen Ordnung abgespeichert
  - Von daher würden uns getMin und deleteMin dort $\Theta(n)$ Zeit kosten, wobei n = Anzahl Schlüssel
  - Der **binäre Suchbaum** (Vorlesung 8) kann alles was eine PQ kann und noch mehr: lookup von beliebigen Elementen
  - Wir werden sehen, dass die **Priority Queue**, für das was sie kann, effizienter ist als ein binärer Suchbaum

    Und tatsächlich gibt es viele Anwendungen, wo eine Priority Queue ausreicht … siehe Folien 12 – 15

Grundlagen 3/9

- Mehrere Elemente mit dem gleichen Key
  - Das ist für viele PQ-Anwendungen nötig und darf man deswegen **nicht** einfach ausschließen
  - Man muss dann nur klären, welches Element get_min und delete_min auswählen, wenn es mehrere kleinste Keys gibt
  - Das übliche Vorgehen ist so

    get_min: gibt irgendein Element mit kleinstem Key zurück

    delete_min: löscht genau das Element, das von getMin zurückgegeben wird bzw. würde

    Bei unserer Implementierung gleich wird das quasi "von selber" der Fall sein

Grundlagen 4/9

- Argument der Operationen changeKey und remove
  - Eine PQ erlaubt **keinen** Zugriff auf ein beliebiges Element
  - Deshalb machen wir es so, dass wir ein Element vorher erzeugen, bevor wir es an insert übergeben

    ```
    pq = PriorityQueue()
    item = PriorityQueueItem(key, value)
    pq.insert(item)
    ```

    Und eben nicht: pq.insert(key, value)
  - Über das item kann man dann später mittels changeKey bzw. remove den Schlüssel ändern bzw. das Element entfernen

    Der key alleine reicht nicht, weil es ja mehrere Elemente mit demselben Schlüssel geben kann

Grundlagen 5/9

- getMax und deleteMax ?
  - Das lässt sich leicht mit folgenden Trick realisieren:

    1. Ein Schlüssel x wird in der PQ als −x gespeichert
    2. An der Implementierung ändert man nichts (das heißt, wenn man z.B. getMax haben will, ruft man weiterhin getMin auf)

  - Das funktioniert, weil das Minimum der modifizierten Schlüssel dem Maximum der ursprünglichen Schlüssel entspricht:

    $\min\{-x_1, \ldots, -x_n\} = -\max\{x_1, \ldots, x_n\}$

    Der Beweis davon ist eine nette (private) Übungsaufgabe

  - Man muss sich entscheiden, ob man "Min" **oder** "Max" haben möchte; man kann nicht ohne Weiteres beides haben

    Der Heap ist nur auf eine Weise geordnet … sehen wir gleich

Grundlagen 6/9

- Anwendungsbeispiel 1
  - Man kann mit einer PQ einfach n Elemente $x_1, \ldots, x_n$ **sortieren**, und zwar so:

    Alle Elemente einfügen: insert(x_1), insert(x_2), …, insert(x_n)

    Dann wieder rausholen, immer das kleinste was noch da ist: get_min(), delete_min(), get_min(), delete_min(), …

    Der entsprechende Algorithmus heißt **HeapSort** YouTube
  - Wir sehen später: alle Operationen gehen in $O(\log n)$ Zeit … damit läuft HeapSort in Zeit $O(n \cdot \log n)$

    Also asymptotisch optimal für vergleichsbasiertes Sortieren

    Insbesondere genauso gut wie MergeSort (im allgemeinen Fall) und QuickSort (im besten Fall)

Grundlagen 7/9

- Anwendungsbeispiel 2
  - Berechnen der k kleinsten Elemente einer Menge:

    Wie auf der Folien vorher, fügt man erst alle n Elemente ein

    Danach aber nur k Aufrufe von get_min() und delete_min()
  - Das Einfügen von n Elementen "auf einen Schlag" geht mittels "heapify" in Zeit $O(n)$

    Siehe Folie 29 + das sind Aufgaben 2, 3 und 4 vom Ü9
  - Die Laufzeit ist dann also $O(n + k \cdot \log n)$

    Alternativ ginge das auch mit einem Algorithmus für "partielles Sortieren", aber der PQ-Algorithmus funktioniert auch bei einer sich dynamisch ändernden Menge

Grundlagen 8/9

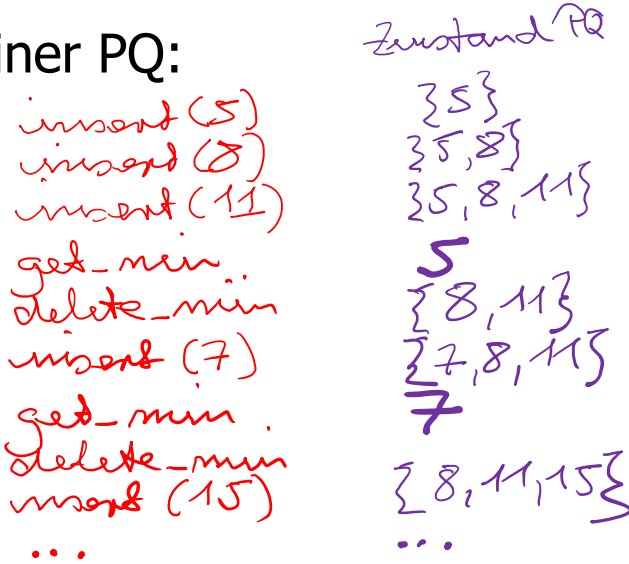
- Anwendungsbeispiel 3
  - Mischen von k sortierten Listen … Englisch: k-way merge
  - Dazu k Laufvariablen, für jede Liste eine … in jeder Iteration wird das kleinste von den betreffenden Elementen berechnet
  - Das geht mit einer PQ in Zeit O(log k) pro Iteration, also insgesamt Zeit O(n · log k), wobei n = Gesamtzahl Elemente
  - Beispiel für Mischen von drei Listen mit einer PQ:

L1: 5, 7, 15, 23

L2: 8, 9, 10

L3: 11, 17, 19, 25

R: 5, 7, …



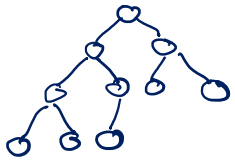
Grundlagen 9/9

- Anwendungsbeispiel 4
  - Die PQ ist die grundlegende Datenstruktur bei **Dijkstra's Algorithmus** zur Berechnung kürzester Wege
  - Das ist insbesondere der Grundalgorithmus hinter jedem Routenplaner und Navigationsgerät

Das ist das Thema der V10 nächste Woche … und das Ü10 wird sein, einen einfachen Routenplaner zu implementieren

Binärer Heap 1/7

- Grundidee

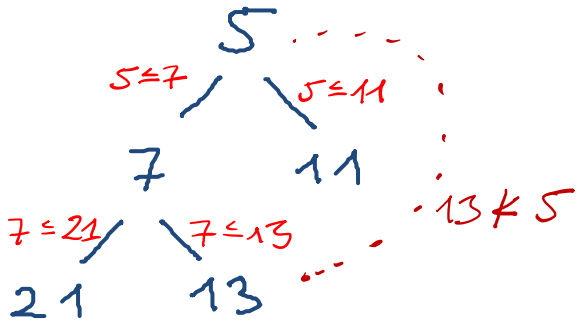


  - Wir speichern die Elemente in einem **binären Heap**, der die folgenden Eigenschaften hat:

    Es ist ein fast **vollständiger binärer Baum** (siehe V8)

    Es gilt die **Heap-Eigenschaft**: Der Key jedes Knotens ist ≤ die Keys seiner Kinder (sofern sie existieren)

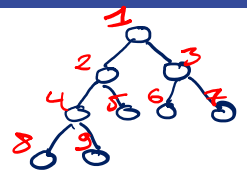
  - Das ist eine **schwächere** Eigenschaft als beim binären Suchbaum, insbesondere sind die Blätter nicht sortiert



ABER z.B. nicht:

13 < 5

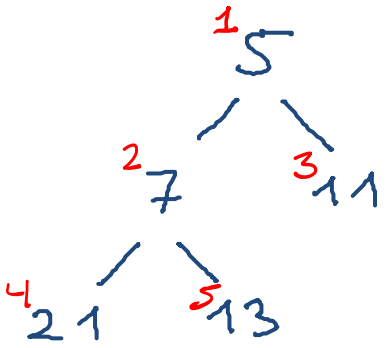
Binärer Heap 2/7

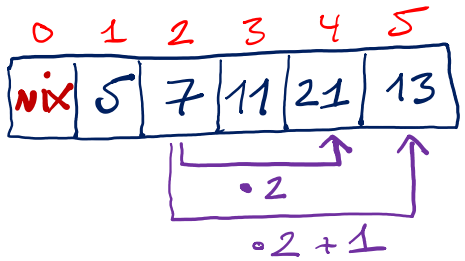


- Wie speichert man einen binären Heap
  - Anders als beim BinarySearchTree geht das **ohne Zeiger**
  - Wir nummerieren die Knoten von oben nach unten und von links nach rechts durch, beginnend mit **1**
  - Wir können die Elemente dann in einem **Feld** speichern, und leicht zu Kinder- bzw. Elternknoten springen:

    Die Kinder von Knoten i sind Knoten $2 \cdot i$ und $2 \cdot i + 1$

    Der Elternknoten von einem Knoten i ist Knoten $\lfloor i/2 \rfloor$





Binärer Heap 3/7

- Änderung an einer Stelle im Heap
  - Alle Operationen außer get_min ändern **genau eine** Stelle im Heap, an der dann evtl. die Heap-Eigenschaft (HE) verletzt ist
  - Das folgende **Lemma** zeigt, dass die HE dann entweder "nach oben" oder "nach unten" verletzt ist (aber nicht beides)
  - Entsprechend brauchen wir zwei Reperaturmethoden

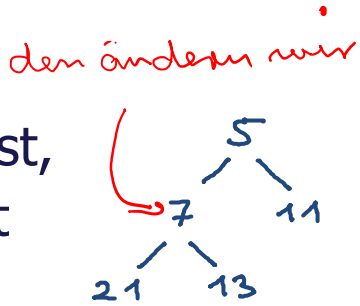
    repair_heap_upwards

    repair_heap_downwards
  - Wie die realisiert werden, sehen wir gleich (Folien 20 + 21)

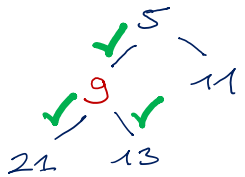
    Vorher die Formulierung des Lemmas (nächste Folie) und der Beweis (übernächste Folie)

Binärer Heap 4/7

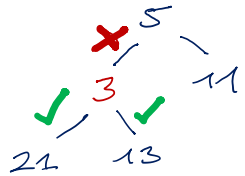
- Es gilt folgendes **Lemma**:
  - Wird in einem binären Heap, in dem die HE erfüllt ist, der Key x an einem Knoten geändert zu y, dann gilt **genau einer** der folgenden drei Fälle



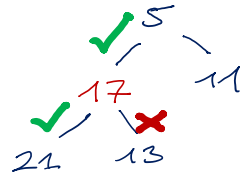
1. Die HE ist weiterhin erfüllt, d.h. falls es einen Elternknoten u gibt, dann key(u) ≤ y und für jedes Kind v gilt, y ≤ key(v)



2. Die HE ist "nach oben" verletzt, d.h. es gibt einen Elternknoten u und es gilt key(u) > y



3. Die HE ist "nach unten" verletzt, d.h. es gibt einen Kindknoten v für den gilt y > key(v)

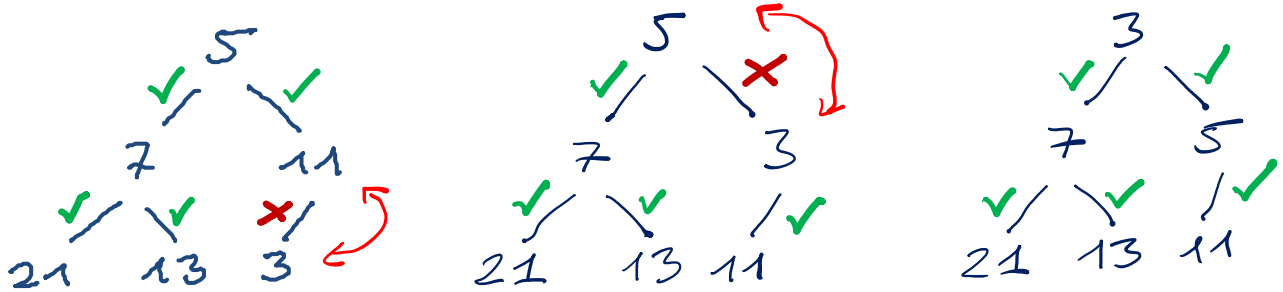


Der Beweis ist eine schöne Klausurvorbeitungsübung

Binärer Heap 5/7

- Methode repair_heap_upwards Live-Coding in der VL
  - Knoten x mit dem Elternknoten y tauschen

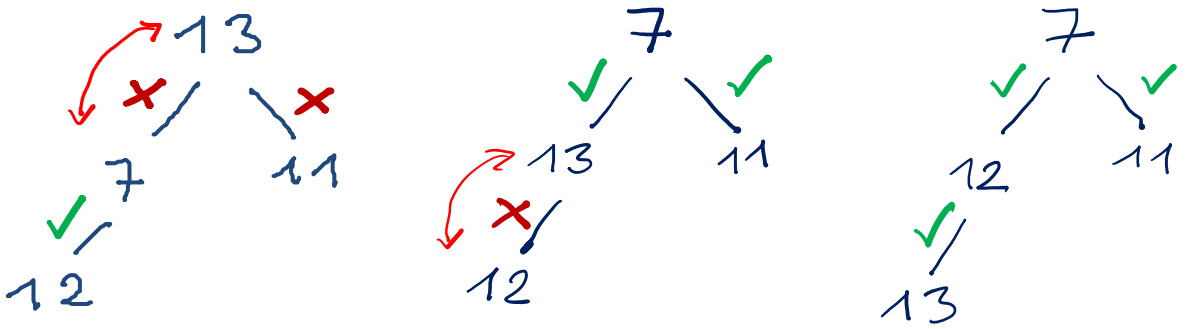
    Zu den Kindknoten hin stimmt dann mit Sicherheit alles, weil der Schlüssel am Elternknoten kleiner wird als vorher
  - Jetzt ist bei dem Elternknoten eventuell die HE verletzt
  - In dem Fall einfach da dasselbe nochmal, usw.



Binärer Heap 6/7

- Methode repair_heap_downwards Aufgabe 1 vom Ü9
  - Knoten x mit dem Kind y tauschen, das den kleineren Key von den beiden Kindern hat

    Zum Elternknoten hin stimmt dann mit Sicherheit alles
  - Jetzt ist bei diesem Kind evtl. die Heap-Eigenschaft verletzt, wenn sein Key größer ist als der von einem der beiden Kinder
  - In dem Fall einfach da dasselbe nochmal, usw.



Binärer Heap 7/7

- Index eines Elementes in der PQ
  - **Achtung:** für change_key und remove muss ein Element wissen, wo es im Heap steht

    Und es ist auch zum Debuggen und für die Unit Tests nützlich, wie wir im Folgenden sehen werden
  - Lösung: jedes Element hat eine zusätzliche Variable heap_index, die angibt, wo es im internen Feld steht
  - Wann immer das Element im Feld verschoben wird, müssen wir darauf achten, heap_index entsprechend anzupassen

    Ein Element wird nur innerhalb von repair_heap_downwards und repair_heap_upwards verschoben

Implementierung der Operationen 1/5

- Die Operation insert

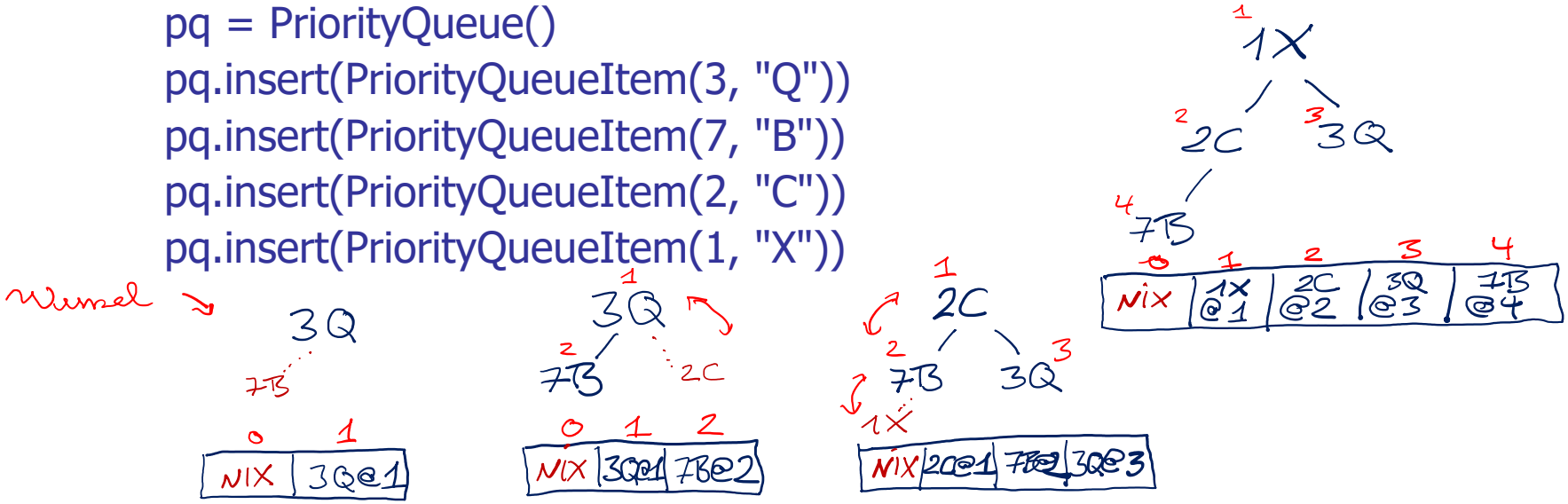
  Live-Coding in der VL

  – Wir fügen das Element erstmal am Ende des Feldes hinzu; danach kann die Heapeigenschaft verletzt sein

  Aber nur genau an dieser (letzten) Position

  – Wir stellen die Heap-Eigenschaft durch die Funktion **repair_heap_upwards** (Folie 20) wieder her … Beispiel:

```
pq = PriorityQueue()
pq.insert(PriorityQueueItem(3, "Q"))
pq.insert(PriorityQueueItem(7, "B"))
pq.insert(PriorityQueueItem(2, "C"))
pq.insert(PriorityQueueItem(1, "X"))
```

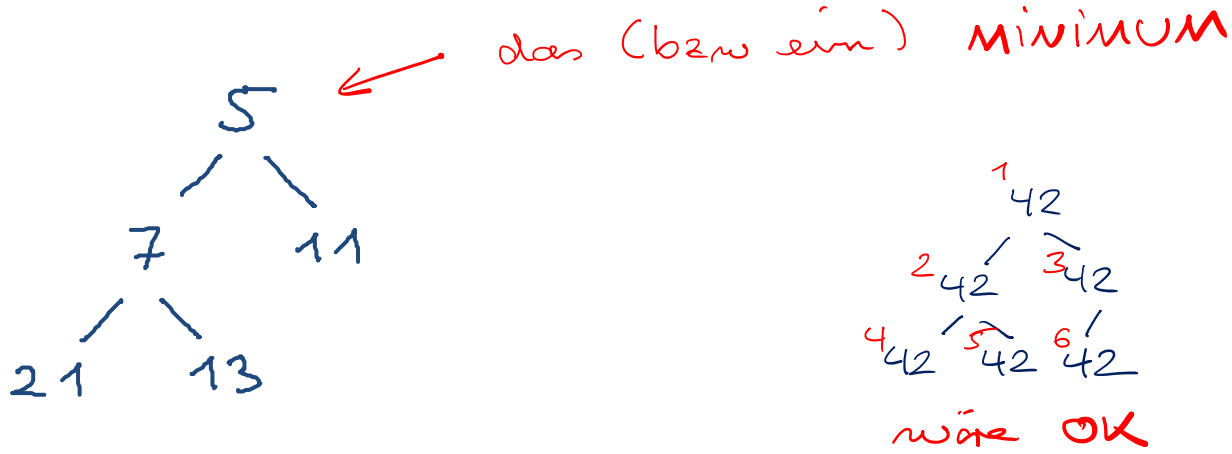


Implementierung der Operationen 2/5

- Die Operation get_min — Live-Coding in der VL
  - Einfach das oberste Element zurückgeben

    Im Feld ist das einfach das Element an Position 1

    Falls der Heap leer ist, ist eine übliche Konvention ein Element wie **None** oder **Null** zurückgeben

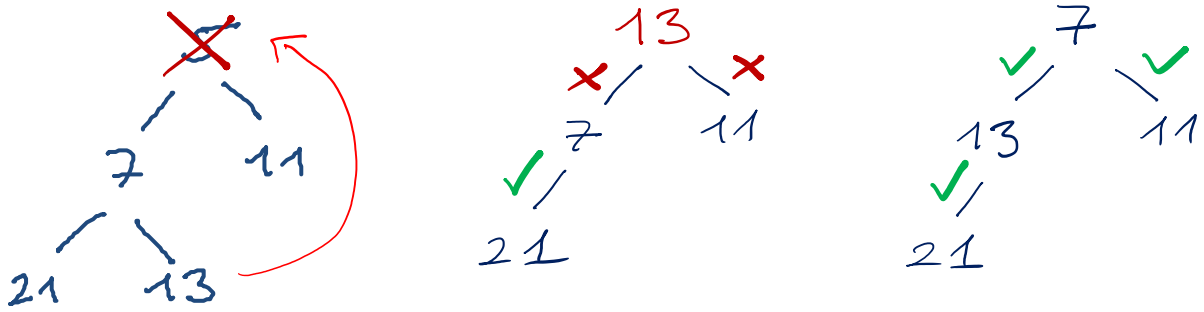


Implementierung der Operationen 3/5

- Die Operation delete_min Aufgabe 1 vom Ü9
  - Wir setzen einfach das Element von der letzten Position an die erste Position (falls der Heap nicht leer ist)

    Element an der ersten Stelle wird dabei überschrieben
  - Danach kann die Heapeigenschaft verletzt sein

    Aber wieder nur genau an dieser (ersten) Position
  - Wiederherstellung der Heap-Eigenschaft mittels der Funktion **repair_heap_downwards** (Folie 21)



Implementierung der Operationen 4/5

- Die Operation change_key Zusatzaufgabe Ü9
  - Das Element wird als Argument übergeben … wir können also einfach seinen Schlüssel ändern
  - Danach tritt laut dem Lemma von Folie 19 genau einer dieser drei Fälle ein (und wir können leicht feststellen, welcher):

    1. Die Heap-Eigenschaft (HE) ist weiter erfüllt

    Dann brauchen wir nichts weiter tun

    2. HE an dem geänderten Knoten "nach oben" verletzt

    Dann rufen wir für den Knoten repair_heap_upwards auf

    3. HE an dem geänderten Knoten "nach unten" verletzt

    Dann rufen wir für den Knoten repair_heap_downwards auf

Implementierung der Operationen 5/5

- Die Operation remove — Zusatzaufgabe Ü9
  - Das Element wird als Argument übergeben
  - Element von der letzten Position an diese Stelle setzen
  - Dann sind wir in der gleichen Situation wie bei change_key
  - Weiteres Vorgehen, siehe dort

Laufzeit und Optimierungen 1/3

- Laufzeit
  - Die Laufzeit von repair_heap_downwards und repair_heap_upwards ist jeweils O(d), wobei d die Tiefe des Baumes ist
  - Aus Vorlesung 8 wissen wir: $d = O(\log n)$ sogar $d = \lfloor \log_2 n \rfloor$

    Nach Konstruktion haben im binären Heap alle inneren Knoten zwei Kinder, außer vielleicht die auf Tiefe d – 1
  - Die Laufzeit von **insert**, **delete_min**, **change_key** und **remove** ist deshalb O(log n)

    Bei jeder dieser Operationen wird potenziell die Heapeigenschaft verletzt und muss wiederhergestellt werden
  - Die Operation **get_min** läuft sogar in O(1) Zeit

    Das Minimum ist einfach das oberste Element im Heap

Laufzeit und Optimierungen 2/3

$$\sum_{i=1}^{\infty} 2^{-i} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 1$$

$$\sum_{i=1}^{\infty} i\,2^{-i} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \frac{5}{32} + \cdots = 2$$

- "Heapify"
  - Herstellen der Heap-Eigenschaft in einem Feld mit n Elementen, mit beliebiger Anordnung gemäß der Keys
  - Das geht trivial mit n insert Operationen in $O(n \cdot \log n)$ Zeit
  - Es geht aber auch effizienter in $O(n)$ Zeit und zwar indem man **repair_heap_downwards** für alle inneren Knoten aufruft und zwar in umgekehrter Reihenfolge, wie sie im Feld stehen

    Ü9, Aufgabe 2: Implementierung dieser heapify Funktion

    Ü9, Aufgabe 3: Beweis, dass dieser Algorithmus korrekt ist

    Ü9, Aufgabe 4: Beweis, dass die Laufzeit $O(n)$ ist

Laufzeit und Optimierungen 3/3

- Fibonacci Heaps
  - Grundidee: Ein "Wald" von (im Allg. nicht mehr vollständigen) binären Bäumen, die im Verlauf ineinander gehängt werden
  - Theoretische bessere Laufzeit als einfach binäre Heaps:

get_min in Zeit $O(1)$	wie beim binären Heap
insert in Zeit $O(1)$	binärer Heap $O(\log n)$
decrease_key in amort. Zeit $O(1)$	binärer Heap $O(\log n)$
delete_min in amort. Zeit $O(\log n)$	binärer Heap $O(\log n)$

  - In der Praxis ist der binäre Heap aufgrund seiner Einfachheit und guten Cache-Effizienz (Feld) aber schwer zu schlagen

    Selbst für $n = 2^{20} \approx 1.000.000$ ist $\log_2 n$ ja nur 20

Bibliotheken 1/3

- Benutzung in **Java** `import java.util.PriorityQueue;`
  - Element-Typ unterscheidet nicht zwischen Key und Value

    `PriorityQueue<T> pq;`
  - Defaultmäßig wird die Ordnung $\leq$ auf T genommen

    Eigene Ordnung über einen Comparator, wie bei sort
  - Operationen:

insert	**add**
delete_min	**poll**
get_min	**peek**
remove	**remove**
change_key	gibt es nicht, aber lässt sich leicht simulieren mit remove und insert

Bibliotheken 2/3

- Benutzung in **C++** `#include <queue>;`
  - Element-Typ unterscheidet nicht zwischen Key und Value

    `std::priority_queue<T> pq;`
  - Es wird die Ordnung $\geq$ auf T genommen, und nicht $\leq$
  - Beliebige Vergleichsfunktion wie bei `std::sort`

insert	**push**
delete_min	**pop**
get_min	**top**
remove	gibt es nicht, aus Effizienzgründen
change_key	gibt es nicht, aus Effizienzgründen

In Vorlesung 10 sehen wir, wie man ohne remove und change_key zurechtkommt, wenn man sie doch braucht

Bibliotheken 3/3

- Benutzung in **Python** from queue import PriorityQueue
  - Elemente sind beliebige Tupel

```
pq = PriorityQueue()
pq.put((key, value))
```

  - Es wird die Ordnung auf den Tupeln genommen

insert	**put**
delete_min	**get**
get_min	**pq.queue[0]** … internes Feld, Beginn bei 0
remove	gibt es nicht … weil in AlgoDat gepennt
change_key	gibt es nicht … weil in AlgoDat gepennt

In Vorlesung 10 sehen wir, wie man ohne remove und change_key zurechtkommt, wenn man sie doch braucht

Literatur / Links

- Prioritätswarteschlangen
  - In Mehlhorn/Sanders:

    6 Priority Queues [einfache und fortgeschrittenere Varianten]
  - In Wikipedia

    http://de.wikipedia.org/wiki/Vorrangwarteschlange

    http://en.wikipedia.org/wiki/Priority_queue