Algorithmen

Michael Kaufmann

26/10/2021 + 2/11/2021 – 4. Vorlesung
Grundlegende Datenstrukturen

Gliederung

I. Einführung
- Motivation und Notationen
- Laufzeitanalyse, Rekursionen

II. Grundlegende Datenstrukturen
- Arrays und Listen
- Bäume
- Keller und Warteschlangen
- Heaps und Prioritätswarteschlangen
- Hashing

III. Graphenalgorithmen

IV. Sortieren

V. Suchen

VI. Generische algorithmische Methoden

VII. Algorithmen auf Zeichenketten

II. Grundlegende Datenstrukturen
a. Arrays und Listen

II.a Arrays und Listen

Arrays: Die allereinfachste Struktur, sollte jeder kennen

- Array der Länge n ist eine Anordnung von n Objekten des selben Typs in fortlaufenden Adressen im Speicher.
- Standardoperationen: Lese/Schreibe ein Element. Geht in $O(1)$, da Adresse im Speicher bekannt.
- Nachteil: Müssen Speicher für Array im Voraus allozieren. Größe ist fest.

Listen: Können ihre Größe (= Länge) ändern.

- Elemente besteht aus Schlüssel und Zeiger *next* auf das nächste Element. Zusätzlich gibt es noch einen Zeiger auf das Startelement der Liste.
- In doppelt verketteten Listen gibt es jeweils noch Zeiger *prev* auf das Vorgängerelement

.

Operationen für Listen (doppelt verkettet)

- Einfügen ein Element x nach Element e in der Liste: $O(1)$ (falls man Element e hat)
- Löschen eines Elements aus der Liste: $O(1)$
- Suche Element mit Schlüsselwert k: $O(n)$
- Löschen einer Teilliste, wenn erstes/letztes Element bekann ist): $O(1)$
- Füge zweite Liste nach einem speziellen Element in der ersten Liste ein: $O(1)$

Implementierung mit speziell markierten Elementen *head* und *tail* for Anfang und Ende
oder ringförmig mit einem 'Wächter'-Element ?

Operationen für Listen (einfach verkettet)

- Da nur ein Zeiger pro Element, weniger Speicherplatz
- Löschen von Elementen geht nicht so gut, da wir keine prev - Zeiger haben

Vergleich Listen und Arrays

Listen: dynamisch, wir brauchen nicht Platz zu allozieren, aber Zugriff auf bestimmtes Element kann $O(n)$ dauern

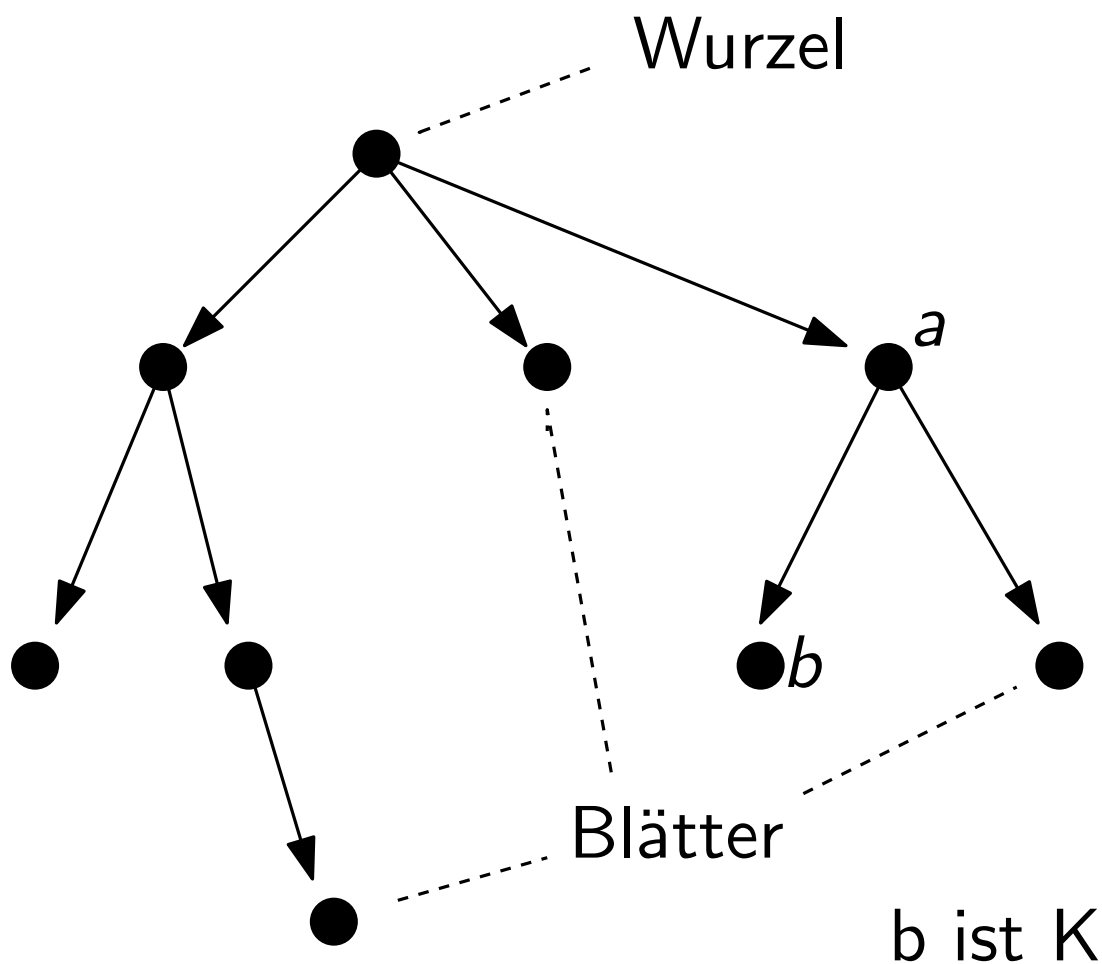
Arrays: Schneller Zugriff über Index, Nachteil: Allozieren. Müssen Größe von vorneherein festlegen

Übung: Implementiere die Move-to-front -Regel !

Greife nacheinander auf Elemente zu. Bei Zugriff verschiebe das Element an den Anfang der Liste !

(Häufiges Zugreifen ist billiger, wenn das Element relativ am Anfang der Liste steht.)

II b. Bäume



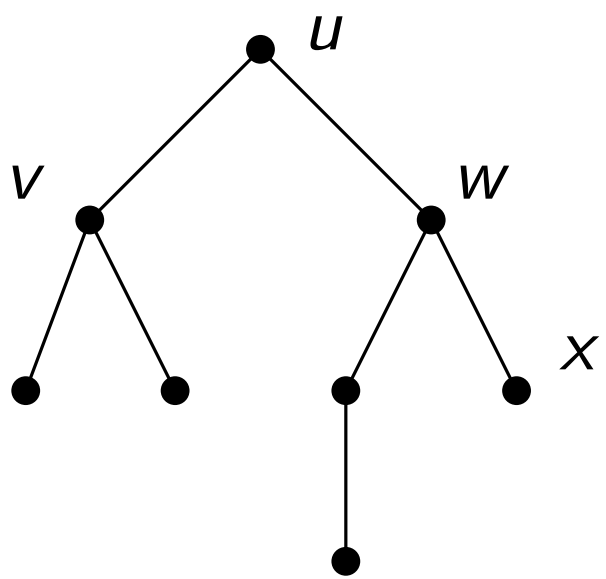
b ist Kind von a.
a ist Vater/parent von b.

Bäume

Sei v Knoten in einem Baum T.

$\text{Tiefe}(v) = 0$ falls v Wurzel,
$= \text{Tiefe}(\text{Vater}(v)) + 1$ falls v nicht Wurzel,

$\text{Höhe}(v) = 0$, falls v Blatt,
$= 1 + max\{\text{Höhe}(w), w \text{ Kind von } v\}$, sonst

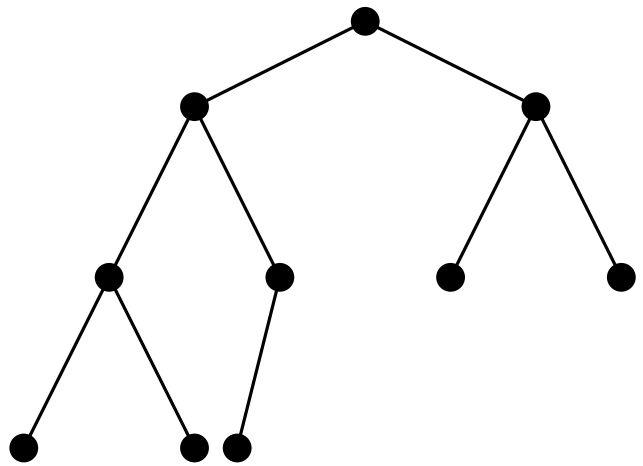


- $\text{Tiefe}(u) = 0$
- $\text{Tiefe}(v) = \text{Tiefe}(w) = 1$
- $\text{Tiefe}(x) = 2$
- $\text{Höhe}(x) = 0$
- $\text{Höhe}(v) = 1$
- $\text{Höhe}(u) = 3$

Allgemein: $\text{Höhe}(T) = max\{\text{Höhe}(v), v \in T\}$
$\text{Tiefe}(T) = max\{\text{Tiefe}(v), v \in T\}$

Binärbäume

- Jeder Knoten hat höchstens 2 Kinder, ein linkes und ein rechtes (lson/rson)
- In einem *vollständigen/complete* Binärbaum sind alle Level außer dem untersten gefüllt. Dort stehen die Knoten möglichst weit links.
- In einem *vollen/full* Binärbaum ist auch das unterste Level komplett voll.



Vollständig !

Verhältnis Höhe zur Knotenanzahl

Lemma: (a) Ein voller Binärbaum mit n Knoten hat Höhe/Tiefe $\log_2(n+1) - 1 \in \Theta(\log n)$.
(b) Für vollständige Binärbäume ist die Formel $\lceil \log_2(n+1) - 1 \rceil \in \Theta(\log n)$.

Beweis: Anzahl der Knoten in einem vollen Binärbaum der Tiefe h ist

$$1 + 2 + 4 + \ldots + 2^h = 2 \cdot 2^h - 1 = 2^{h+1} - 1$$

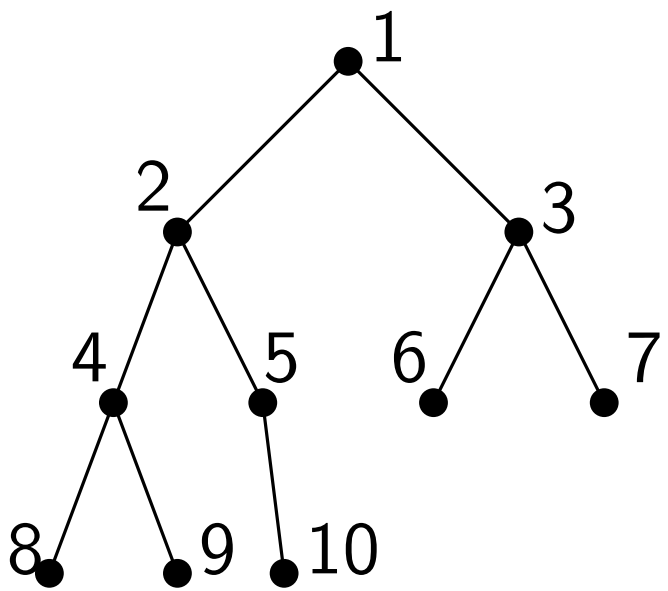
Damit gilt:
$n = 2^{h+1} - 1 \Leftrightarrow h = \log_2(n+1) - 1$ (a)

(b) Im Fall von vollständigen Binärbaum ist das letzte Level eventuell nicht ganz voll. Deshalb wird einfach aufgerundet:
$h = \lceil \log_2(n+1) \rceil - 1$

Wie werden Bäume implementiert ?

Vollständige Binärbäume:

- Alle Blätter haben Tiefe k oder $k+1$.
- Blätter auf Tiefe $k+1$ sitzen 'ganz links'.



Nummerierung schichtweise
von links nach rechts

Für Knoten mit Nummer i gilt:
- Vater hat Nummer $\lfloor i/2 \rfloor$
- Kinder haben Nummern $2i$, $2i+1$.

- wird als Array dargestellt, nur konzeptuell als Baum!
  Indizierungsregeln ergeben Vater-Kind-Relationen

Allgemeine Binärbäume:
Jeder Knoten hat seinen Schlüsselwert, sowie Zeiger auf sein linkes und rechten Kindknoten, sowie einen Zeiger auf seinen Vaterknoten

Ist die Zahl der Kinder pro Knote beschränkt (z.B. 4), kann man das auch so machen.

Allgemeine Bäume:
Geschwisterknoten sind nicht beschränkt, deshalb:
Kinder eines Knotens v werden in einer Liste organisiert.
v hat einen Zeiger auf sein linkestes Kind. Jeder Knoten hat einen Zeiger auf seinen rechten Geschwisterknoten. Dies ergibt dann die Liste der Geschwister von v, von links nach rechts.

Keller + Warteschlangen (Stacks + Queues)

Keller:
Dynamische Datenstruktur, bei der immer nur das oberste Element zugreifbar ist.
Operationen: *Push(x)* fügt Element x oben auf den Keller ein.
Pop löscht das oberste Element vom Keller.
Prinzip: LIFO (Last in First out)

Implementierung:
– Array mit Zeiger auf das augenblickliche Top-Element.
Operationen gehen in $O(1)$.

– Bei doppelt verkettete Liste: Zeiger einfach ans Ende der Liste. Dort steht das Top-Element.

– Bei einfacher Verkettung füge die Element vorne ein! Und halte den Zeiger auf das erste Element!

Warteschlangen (Queues)

Dynamische Datenstruktur, die nach FIFO (First in first out) funktioniert.
Operationen: *Einfügen(x)* fügt Element x ans Ende der Queue (Enqueue). *Löschen* löscht das erste Element der Queue.

Implementierung:
– Array mit Zeiger auf das augenblickliche End- sowieTop-Element. Operationen gehen in $O(1)$.

– Bei doppelt verkettete Liste ....

– Bei einfacher Verkettung ....

Heaps

Was sind Heaps ?

– Einfache baumartige Datenstruktur zur Verwaltung einer Menge mit Ordnungsrelation.

– Unterstützt Operationen wie Insert, IncreaseKey, DecreaseKey, ExtractMin

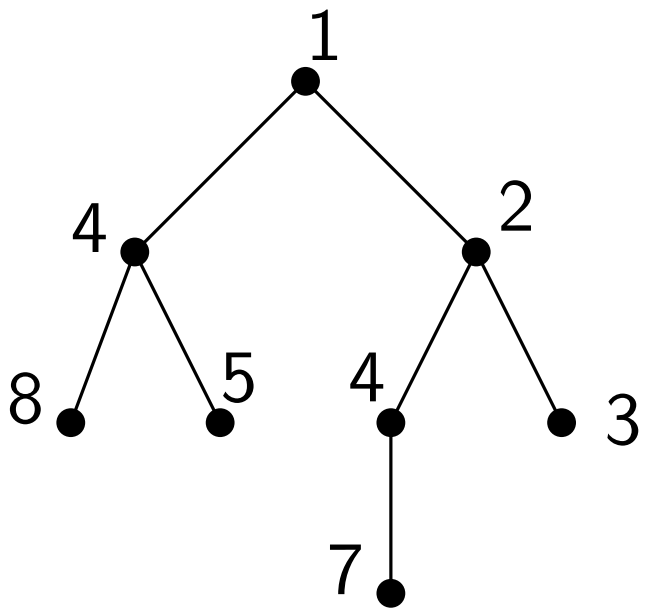
– Kompromiss zwischen ungeordnetem Array und geordnetem Array (Überlege Unterschiede, Vorteile, Nachteile

Heaps: Grundlagen

Speichern Menge S im Heap T ab.
Knoten in T entsprechen Elementen in S.

Heapeigenschaft: Für Knoten u und v mit Vater$(v) = u$ gilt:

$$S[u] \leq S[v]$$

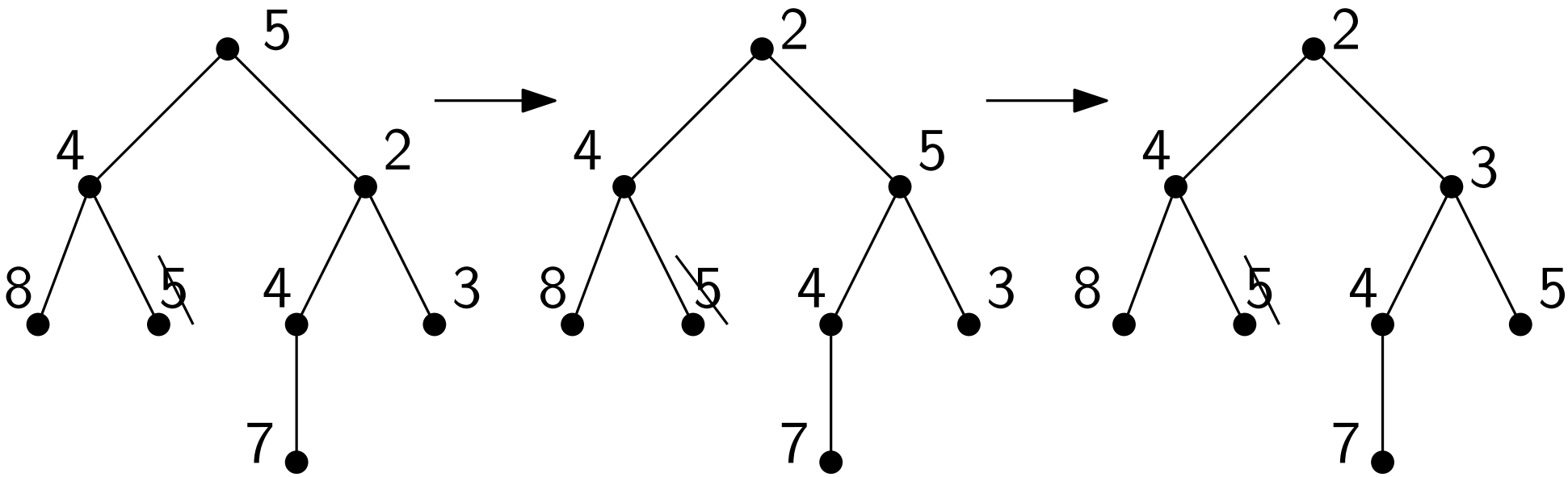


$$S = \{1, 2, 3, 4, 5, 4, 7, 8\}$$

Operationen: ExtractMin

Im Heap steht Minimum in der Wurzel. Also:

1. Suche Minimum $\rightarrow$ Wurzel $O(1)$
2. Entferne Minimum aus Heap. Repariere Heap:
   - Nimm Blatt und füge es in Wurzel ein.
   - Lass es nach unten sinken (Vergleich mit kleinstem Kind)

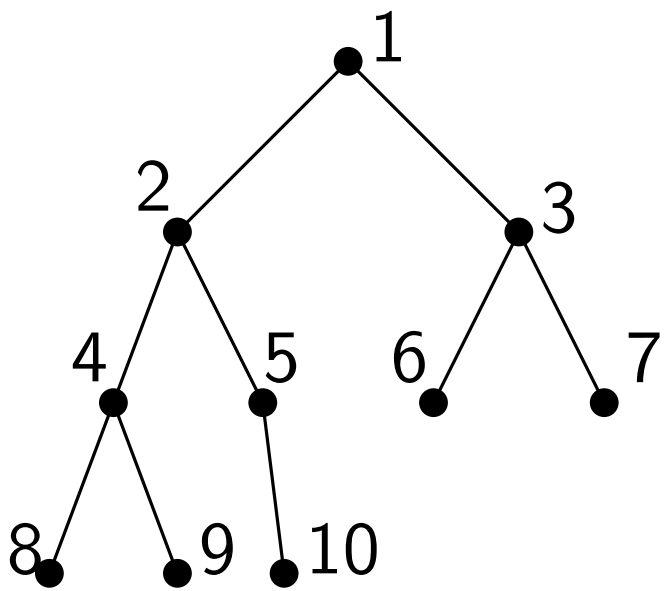


Laufzeit: $O(\text{Höhe}(T))$ **Problem**: Mache Höhe klein!

Wie werden Heaps implementiert ?

Als vollständige Binärbäume:

- Alle Blätter haben Tiefe k oder $k+1$.
- Blätter auf Tiefe $k+1$ sitzen 'ganz links'.



Nummerierung schichtweise
von links nach rechts

Für Knoten mit Nummer i gilt:
- Vater hat Nummer $\lfloor i/2 \rfloor$
- Kinder haben Nummern $2i$, $2i+1$.

- wird als **Array** dargestellt, nur konzeptuell als Baum!
  Indizierungsregeln ergeben Vater-Kind-Relationen

HeapAufbau/Initialisierung: Algorithmus

Initialisierung des Heaps H:
für alle $a \in S$ {**Insert**(a, H)}

Insert(a, H)**:**
sei n die Größe von H;
$n \leftarrow n + 1$;
$H(n) \leftarrow a$;
$i \leftarrow j \leftarrow n$;
while $(i > 1)$ {
$j \leftarrow \lfloor j/2 \rfloor$;
if $(H[j] > H[i])$ {tausche $H[i]$ und $H[j]$; $i \leftarrow j$};
else $i \leftarrow 1$;
};

Beispiel!

HeapAufbau: Analyse

Laufzeit:
Initialisierung (Einfügen, Hochsteigen lassen): $O(n \cdot \text{Höhe}(H))$

Höhe(H) is $O(\log n)$, da Baum vollständig ist

Lemma:
Ein vollständiger Baum der Höhe k hat mindestens 2^k Knoten.

Beweis:
Der kleinste Baum der Höhe k hat nur einen Knoten auf Tiefe k.
Auf Stufe i gibt es 2^i Knoten für $0 \leq i < k$.
$\Rightarrow \sum_{i=0}^{k-1} 2^i + 1 = 2^k$.

$\Rightarrow$ Für Heap mit n Knoten der Höhe k gilt $n \geq 2^k$
$\Rightarrow \log n \geq k = \text{Höhe}(H)$

Heapaufbau: Geht es besser ?

Warum dieser Algorithmus nicht schneller sein kann:
Füge Schlüssel in absteigender Reihenfolge an, also z.B. $n, n-1, n-2, \ldots, 2, 1$. Bei jedem Hochsteigen muss der jeweilige Schlüssel bis ganz hoch in die Wurzel.
Das sind bei fast allen Schlüsseln mehr als $1/2 \cdot \log n$ Schritte.
Also insgesamt $\Omega(n \log n)$.

Alternative: Füge die Schlüssel nacheinander ein, OHNE gleich hochsteigen zu lassen.
Dann mache alle Unterbäume zu Heaps, startend bei den Blättern levelweise.
Genauer: Betrachte v. Dann sind beide Unterbäume von v schon Heaps. Nur wenn die Heapeigenschaft für v nicht gilt, lasse v runtersinken in Zeit $O(h(v))$, falls v die Höhe h hat.
Danach ist Unterbaum mit Wurzel v ein Heap.

Alternative hat nur $O(n)$ Laufzeit:

Überlegung: Behandeln eines Knotens auf Tiefe i hat Laufzeit $O(\log n - i)$.
Es gibt 2^i Knoten davon. Also

Laufzeit: $O(\sum_{i=0}^{\log n}(\log n - i) \cdot 2^i) = ... = O(n)$.

Wieder was verbessert, was offensichtlich vernünftig und nicht zu verbessern war !

Operation: IncreaseKey

Sei H ein Heap von n Knoten. Jetzt wird der Schlüssel von Knoten v erhöht. Deshalb kann es sein, dass v jetzt die Heapeigenschaft verletzt.

Beachte: Alle anderen Knoten erfüllen die Heapeigenschaft. Lasse den Schlüsselwert nach unten sinken: Tausche den Platz mit dem Kind w, das den kleineren Schlüsselwert hat.

Beobachtung: v erfüllt die Heapeigenschaft. Eventuell erfüllt w immer noch nicht die Heapeigenschaft. Dann lasse den Schlüsselwert weiter nach unten sinken.

Laufzeit: $O(h) = O(\log n)$ für Höhe h des Heaps

Operation DecreaseKey

Sei H ein Heap von n Knoten. Jetzt wird der Schlüssel von Knoten v erniedrigt. Deshalb kann es sein, dass jetzt die Heapeigenschaft für parent(v) verletzt ist.

Beachte: Alle anderen Knoten erfüllen die Heapeigenschaft. Lasse den Schlüsselwert nach oben steigen: Tausche den Platz mit dem parent(v).

Beobachtung: v erfüllt die Heapeigenschaft. Eventuell erfüllt $parent(v)$ immer noch nicht die Heapeigenschaft. Dann lasse den Schlüsselwert weiter nach oben steigen.

Laufzeit: $O(h) = O(\log n)$ für Höhe h des Heaps

Anwendungen später, z.B. HeapSort

Einige Operationen gehen noch besser: z.B. FibonacciHeaps (kürzeste Wege)

Hat man viele billige Operationen, aber nur wenig teuere, lohnt es sich oft, genauer hinzuschauen

Jetzt : **Prioritätswarteschlangen**