Algorithmen und Datenstrukturen

B5. Heaps und Heapsort

Marcel Lüthi and Gabriele Röger

Universität Basel

B5.1 Einführung

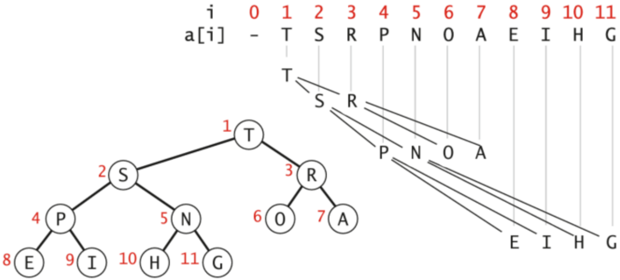
Ausblick auf Vorlesung

- Die Datenstruktur Heap
- Heaps zur Implementation von Priorityqueues
- Heapsort

B5.2 Heaps

Bijektion - Array / Vollständiger Binärbaum

- Jedes Array kann als vollständiger Binärbaum interpretiert werden:
  - Linker Teilbaum: Index Wurzel * 2
  - Rechter Teilbaum: Index Wurzel * 2 + 1

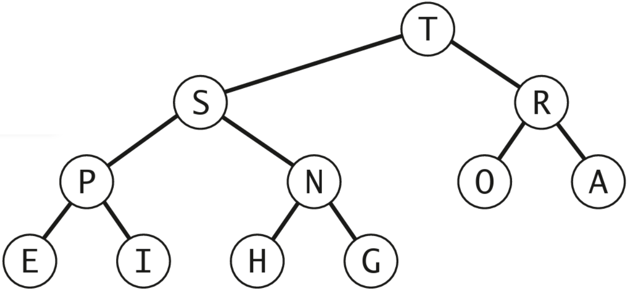


Quelle: Abbildung 2.26, Algorithms, Sedgewick & Wayne

Heap

Definition: Heap

Ein binärer Baum / Array ist Heap geordnet, wenn der Schlüssel in jedem Knoten grösser gleich dem Schlüssel seiner beiden Kindern (sofern vorhanden) ist.



Quelle: Abbildung 2.25, Algorithmen, Wayne & Sedgewick

Heap Ordnung

Theorem

Der grösste Schlüssel in einem Heap-geordneten Binärbaum befindet sich an der Wurzel.

Beweis.

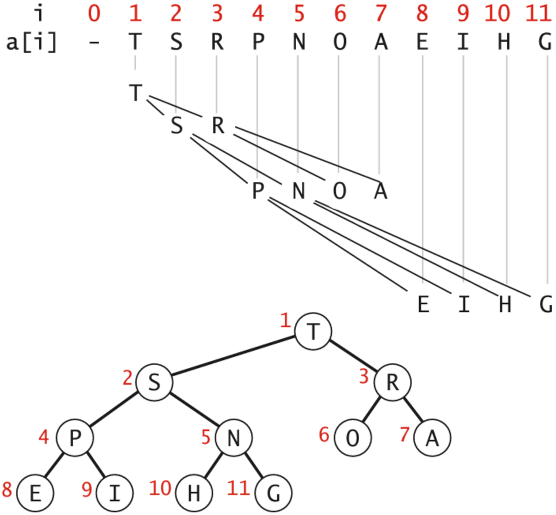
Induktion über die Baumgrösse □

Binärer Heap

Definition: Binärer Heap

Ein binärer Heap ist eine Sammlung von Schlüsseln, die in einem vollständigen Heap-geordneten Binärbaum angeordnet sind und in einem Array ebenenweise repräsentiert werden (das erste Feld des Arrays wird nicht verwendet).

Binärer Heap



Quelle: Abbildung 2.26, Algorithmen, Wayne & Sedgewick

B5.3 Warteschlangen mit Heaps

Priority Queue ADT

```
class MaxPQ[Item]:

    # Element einfuegen
    def insert(k : Item) -> None

    # Groesstes Element zurueckgeben
    def max() -> Item

    # Groesstes Element entfernen und zurueckgeben
    def delMax() -> Item

    # Ist die Queue leer?
    def isEmpty() -> bool

    # Anzahl Elemente in der Priority Queue
    def size() -> int
```

Beobachtung

Array implementation von Max-heap hat grösstes Element immer an Stelle 1 .

- Implementation von `max` ist trivial

Problem: Wir müssen wenn wir beim `insert` und `delMax` die Heapbedingung erfüllen können.

Beobachtung (2)

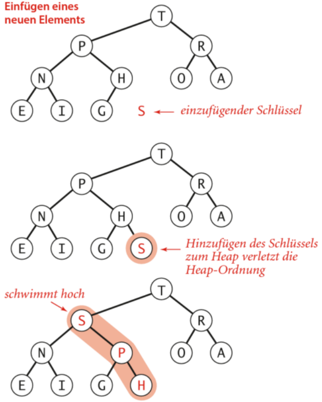
- Array implementation erlaubt uns in konstanter Zeit zu jedem Kind den Elternknoten und von jedem Elternknoten alle Kinder finden ...
  ... ohne dabei explizite Verweise verwalten zu müssen .
- Der Baum hat die Höhe $\lfloor \log_2(N) \rfloor$

Plan

Durch geschicktes Vertauschen der Eltern/Kinder in $O(\log_2(N))$ Operationen nach Entfernen oder Einfügen eines Elements die Heapbedingung wiederherstellen.

Element einfügen

- Blatt wird an letzter Stelle im Array eingefügt
  - entspricht Blatt ganz rechts
- Heap Bedingung wird durch Ausführen von `swim` wiederhergestellt

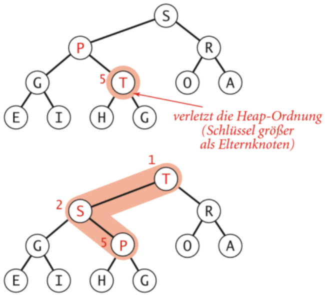


Quelle: Abbildung 2.29: Algorithmen, Sedgewick & Wayne

Die Operation `swim`

- Knoten an Position k in Array a schwimmt nach oben bis Heap Bedingung wieder erfüllt ist.
- Braucht maximal $\log_2(N) + 1$ Vergleiche.

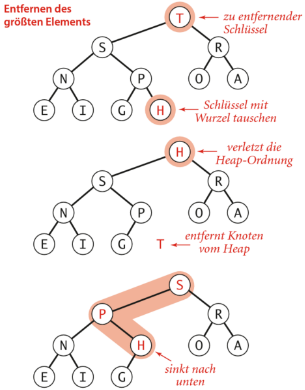
```
def swim(a, k):
  while k > 1 and a[k/2] < a[k]:
    a[k/2], a[k] = a[k], a[k/2]
    k = k/2
```



Quelle: Abbildung 2.29: Algorithmen, Sedgewick & Wayne

Grösstes Element entfernen

- Wurzel (grösstes Element) wird entfernt
- Blatt ganz rechts wird an Wurzel gesetzt
- Heap Bedingung wird durch ausführen von `sink` wiederhergestellt

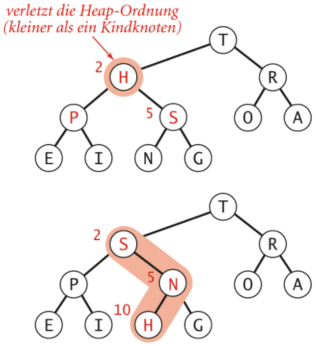


Quelle: Abbiludung 2.29: Algorithmen, Sedgewick & Wayne

Die Operation `sink`

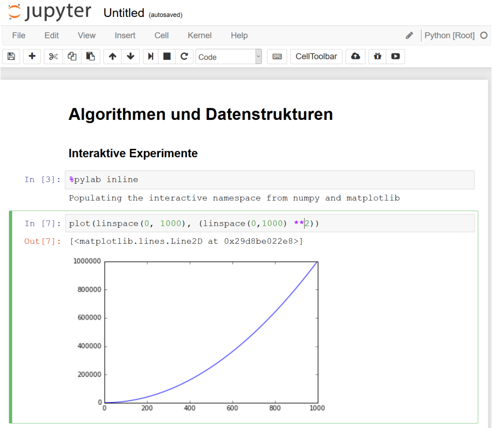
- Knoten an Position k in Array a sinkt nach unten bis Heap Bedingung wieder erfüllt ist.
- Element wird mit grösserem Kind vertauscht.
- Braucht maximal $2\log_2(N)$ Vergleiche.

```
def sink(a, k):
  # Elemente in a[1]..a[N]
  while 2 * k <= N:
    j = 2 * k
    if j < N and a[j] < a[j+1]:
      j += 1
    if not a[k] < a[j]:
      break
    a[j], a[k] = a[k], a[j]
    k = j
```



Quelle: Abbiludung 2.29: Algorithmen, Sedgewick & Wayne

Implementation



Juypter Notebooks: Heap.ipynb

Komplexität

Theorem

In einer Vorrangwarteschlange mit N Elementen benötigen die Heap-Algorithmen zum Einfügen eines neuen Elements nicht mehr als $1 + \log_2(N)$ Vergleiche und zum Entfernen des grössten Elements nicht mehr als $2 \log_2(N)$ Vergleiche.

B5.4 Heapsort

Ein Sortieralgorithmus

- Gegeben, ein unsortiertes Array der länge N .
- Füge alle Elemente der Reihe nach in einen Heap ein.
- Entferne N mal das grösste Element und schreibe es zurück ins Array.

Komplexität

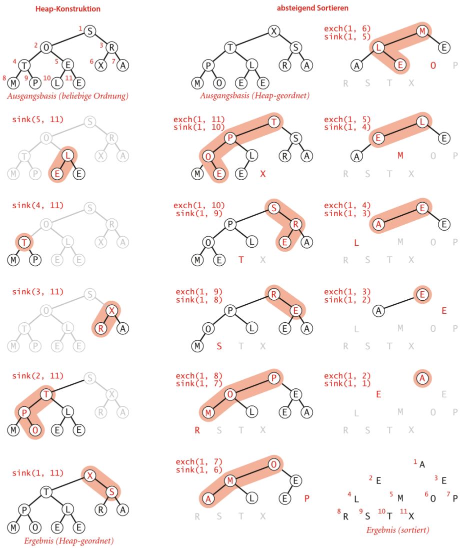
Die Prozedur hat garantierte Laufzeitkomplexität von $O(N \log_2(N))$.

Heapsort

- Idee: Geschicktes verwenden von swim und sink lässt uns heapsort in-place verwenden.
- Prozedur verläuft in zwei Phasen:
  1. Heap Konstruktion (rechts nach Links)
  2. Absteigendes Sortieren durch sukkzesives Tauschen von grösstem Element

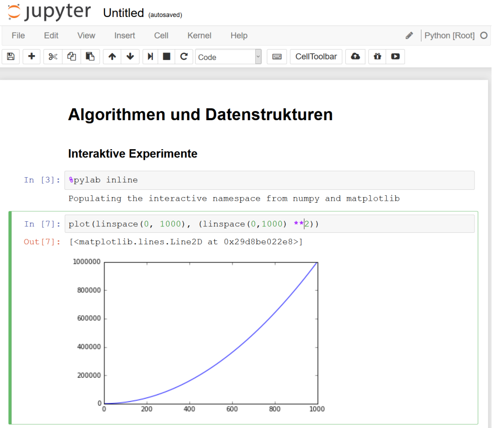
```
def heapsort(a):
    N = len(a) -1
    for k in range(int(N/2), 0, -1):
        sink(a, k)
    while N > 1:
        a[1], a[N] = a[N], a[1]
        N -= 1
        sink(a, 1, N)
```

Heapsort



Quelle: Abbildung 2.31, Algorithmen, Wayne & Sedgewick

Implementation



Jupyter Notebooks: Heaps.ipynb

Bemerkungen

- Heapsort ist theoretisch wichtig:
  - Optimal hinsichtlich Zeit und Speichernutzung
  - Laufzeit $O(n \log n)$.
  - Zusätzlicher Speicher ($O(1)$)
- Praktische Bedeutung eher klein
  - Nutzt CPU Cache nicht effizient, da entfernte Elemente ausgetauscht werden.
- Heaps sind aber für Priority Queues sehr wichtig!

Zusammenfassung

- Heap-sort Algorithmus von Datenstruktur "getrieben"
- Nutzt nicht triviale Zwischenschritte und Hilfsstrukturen
  - Nutzung von Eigenschaften vollständiger binäre Bäume
  - Effiziente Implementation mittels Arrays
  - Heap Bedingung um grösstes Element zu erhalten
- Verständnis von Heap ist zentral für Algorithmus
  - Danach ist Algorithmus einfach zu verstehen
  - Laufzeitanalyse trivial