Programmiertechnik II

Ralf Herbrich

Bäume (*Heaps*)

Überblick

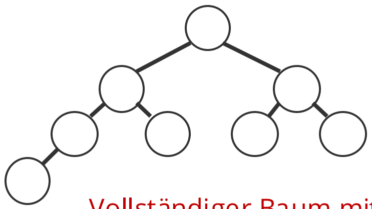
1. Halden (*Heaps*)
2. Haldenbedingung wiederherstellen *(heapify)*
3. Anwendung von Heaps
   - Prioritätswarteschlange
   - *Heapsort*

Überblick

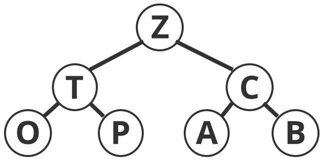
1. **Halden (*Heaps*)**
2. Haldenbedingung wiederherstellen *(heapify)*
3. Anwendung von Heaps
   - Prioritätswarteschlange
   - *Heapsort*

Heaps (Halde)

- **Definition**. Ein *Heap* (auch „Halde“ oder „Haufen“) ist ein Binärbaum, dessen Knoten die folgende drei Eigenschaften erfüllen:
    1. Der Schlüssel jedes Knotens ist größer oder gleich den Schlüsseln aller seiner Kindknoten.
    2. Der Baum ist vollständig (bis auf die letzte Ebene).
    3. Blattebene wird von links nach rechts befüllt.
- **Satz**: Die Höhe eines vollständigen Baums mit n Knoten ist $\lfloor \log_2(n) \rfloor$.
- **Beweis**: Ein vollständiger Baum der Höhe k hat $1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1$ Knoten.
- **Achtung**: Nur partielle Ordnung im Baum; keine Ordnung innerhalb einer Ebene!

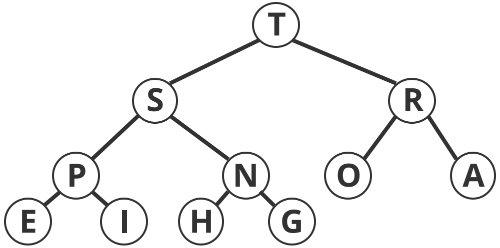


Vollständiger Baum mit 8 Knoten (Höhe = 3)

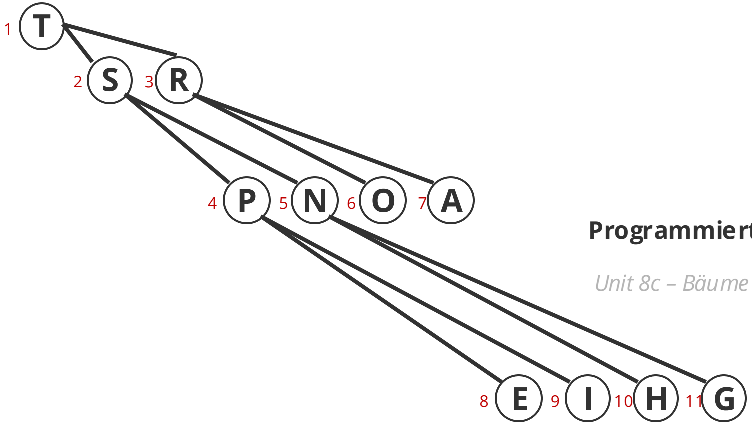


Binary Heaps (Binäre Halde): Repräsentation

- **Baumrepräsentation**:
  - Schlüssel in den Knoten
  - Elternschlüssel nie kleiner als Kindknoten
  - Jeder Knoten enthält Referenz zu 2 Kindern und Referenz zum Elternknoten



- **Feldrepräsentation**:
  - Indizes starten bei 1
  - Knoten sind in *level order* im Feld
  - **Elternknoten von k**: $\lfloor k/2 \rfloor$
  - **Kindknoten von k**: $2k$ und $2k+1$
  - Kein Speicher für explizite Kanten!



Überblick

1. Halden (*Heaps*)
2. **Haldenbedingung wiederherstellen *(heapify)***
3. Anwendung von Heaps
   - Prioritätswarteschlange
   - *Heapsort*

Bottom-Up Herstellung der Heapordnung: (*swim up*)

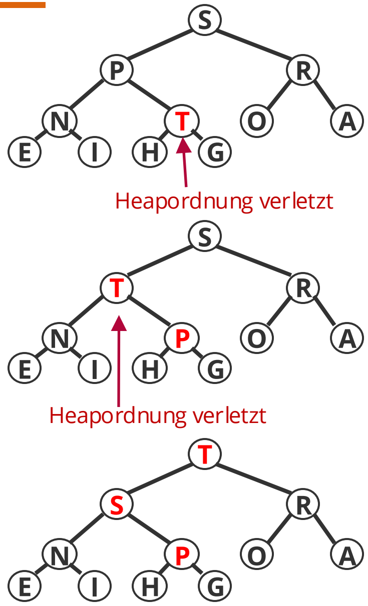
- **Heapordnung**: Elternschlüssel ist größer als Kindschlüssel.
  - Betrachtet aus Sicht des Kindknoten
- **Idee**:
  1. Tausche den Eltern- und Kindknoten um partielle Heapordnung wieder herzustellen. (*swim up*)
  2. Wiederhole Schritt 1 bis die Heapordnung wiederhergestellt ist.

```
// Implements the swim up function of a heap
template <typename Value>
void swim(Value* heap, int k) {
    while (k > 1 && less(heap, k/2, k)) {
        swap(heap, k, k/2);
        k = k/2;
    }
}
```

```
// Implements a swap of element i and j in an array
template <typename Value>
void swap(Value* heap, const int i, const int j) {
    const Value tmp = heap[i - 1];
    heap[i - 1] = heap[j - 1];
    heap[j - 1] = tmp;
    return;
}
```

```
// Implements comparison of two heap elements (assu
template <typename Value>
bool less(Value* heap, const int i, const int j) {
    return (heap[i - 1] < heap[j - 1]);
}
```

heap.h

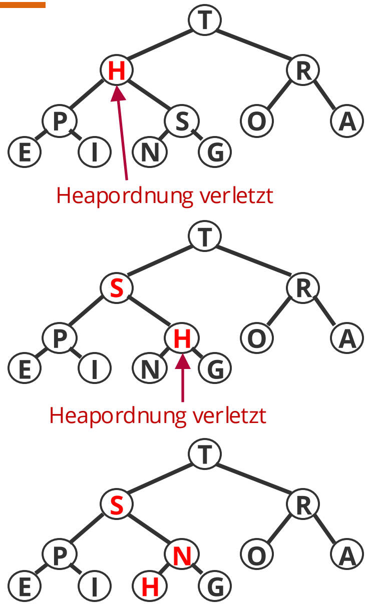


Top-Down Herstellung der Heapordnung: (*sink down*)

- **Heapordnung**: Elternschlüssel ist größer als Kindschlüssel.
  - Betrachtet aus Sicht des Elternknoten
- **Idee**:
  1. Tausche den Elternknoten mit dem *größeren* der beiden Kindknoten um partielle Heapordnung wieder herzustellen. (*sink down*)
  2. Wiederhole Schritt 1 bis die Heapordnung wiederhergestellt ist.

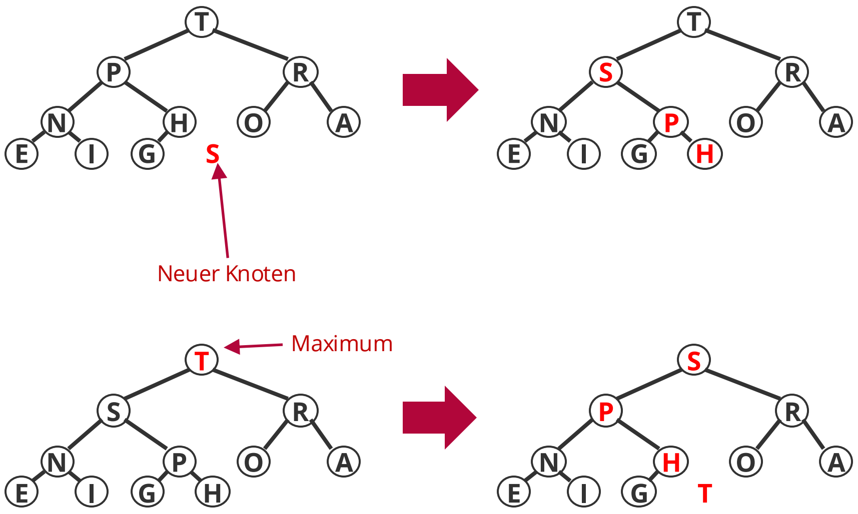
```
// Implements the sink function of a heap        You,
template <typename Value>
void sink(Value* heap, int k, int n) {
    while (2 * k <= n) {
        int j = 2 * k;
        if (j < n && less(heap, j, j + 1)) j++;
        if (!less(heap, k, j)) break;
        swap(heap, k, j);
        k = j;
    }
}
```

heap.h



Heapoperationen

- **Einfügen**: Neuen Knoten am Ende einfügen und dann nach „oben schwimmen“ lassen
  - **Kosten**: Garantiert nicht mehr als $1 + \log_2(n)$ Vergleiche
- **Maximum Entfernen**: Wurzelknoten mit letztem Element tauschen und dann neue Wurzel „nach unten sinken“ lassen
  - **Kosten**: Garantiert nicht mehr als $2 \cdot \log_2(n)$ Vergleiche



Überblick

1. Halden (*Heaps*)
2. Haldenbedingung wiederherstellen *(heapify)*
3. Anwendung von Heaps
   - **Prioritätswarteschlange**
   - *Heapsort*

Anwendung von *Heaps*: Prioritätswarteschlange

- **Prinzip**: Datensammlung, bei dem immer das kleinste (oder größte) Element entfernt wird
- **Anwendungen**:
  - Simulation (z.B., Kunden in Wartschlange, Partikel)
  - Diskrete Optimierung (z.B., Scheduling)
  - Künstliche Intelligenz (z.B., A* Suche)
  - Netzwerke (z.B., Web Cache)
  - Betriebssysteme (z.B., *load balancing*)
  - Suche auf Graphen (z.B., Dijkstra's oder Prim's Algorithmus)

Implementierung	Garantiert			*Average*		
	Einfügen (`put`)	Maximum (`max`)	Löschen (`remove max`)	Einfügen (`put`)	Maximum (`max`)	Löschen (`remove max`)
Sortierte Liste	n	1	1	$\log_2(n)$	1	1
Binärer Heap	$\log_2(n)$	1	$\log_2(n)$	$\log_2(n)$	1	$\log_2(n)$

Operation	Arg	Ergebnis
insert	P	
insert	Q	
insert	E	
remove min		E
insert	X	
insert	A	
insert	M	
remove min		A
insert	P	
insert	L	
insert	E	
remove min		E

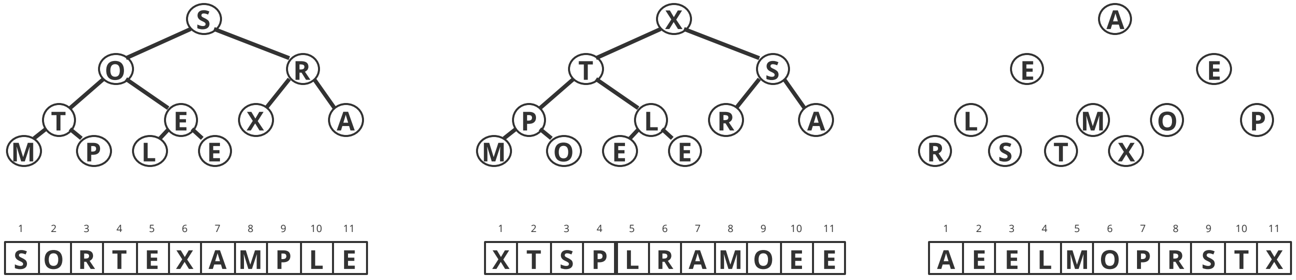
min_pq.h

Überblick

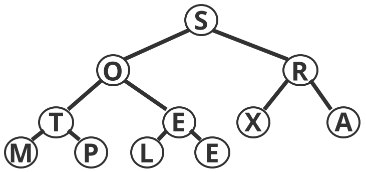
1. Halden (*Heaps*)
2. Haldenbedingung wiederherstellen *(heapify)*
3. Anwendung von Heaps
   - Prioritätswarteschlange
   - ***Heapsort***

Heapsort

- **Grundidee**:
  1. Betrachte das Eingabefeld als einen vollständigen binären Baum
  2. **Heapaufbau**: Baue einen *heap* aus dem Feld mit allen n Schlüsseln
  3. **Runtersortieren**: Entferne nach-und-nach den maximalen Schlüssel.



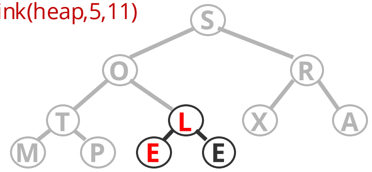
Heapsort: *Top-Down* Heapaufbau



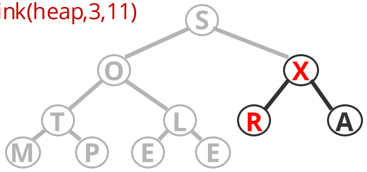

```
// heapify phase          You, last w
for (int k = n / 2; k >= 1; k--)
    sink(heap, k, n);
```

heap.h

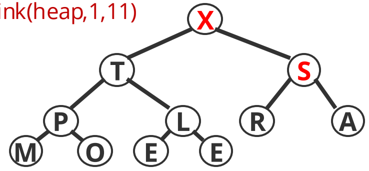
sink(heap,5,11)



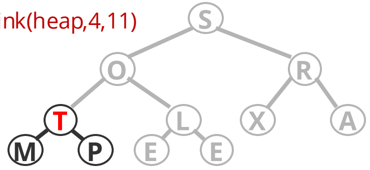
sink(heap,3,11)



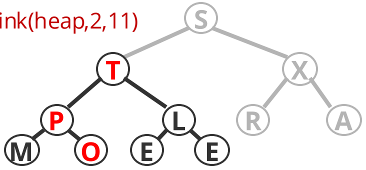
sink(heap,1,11)



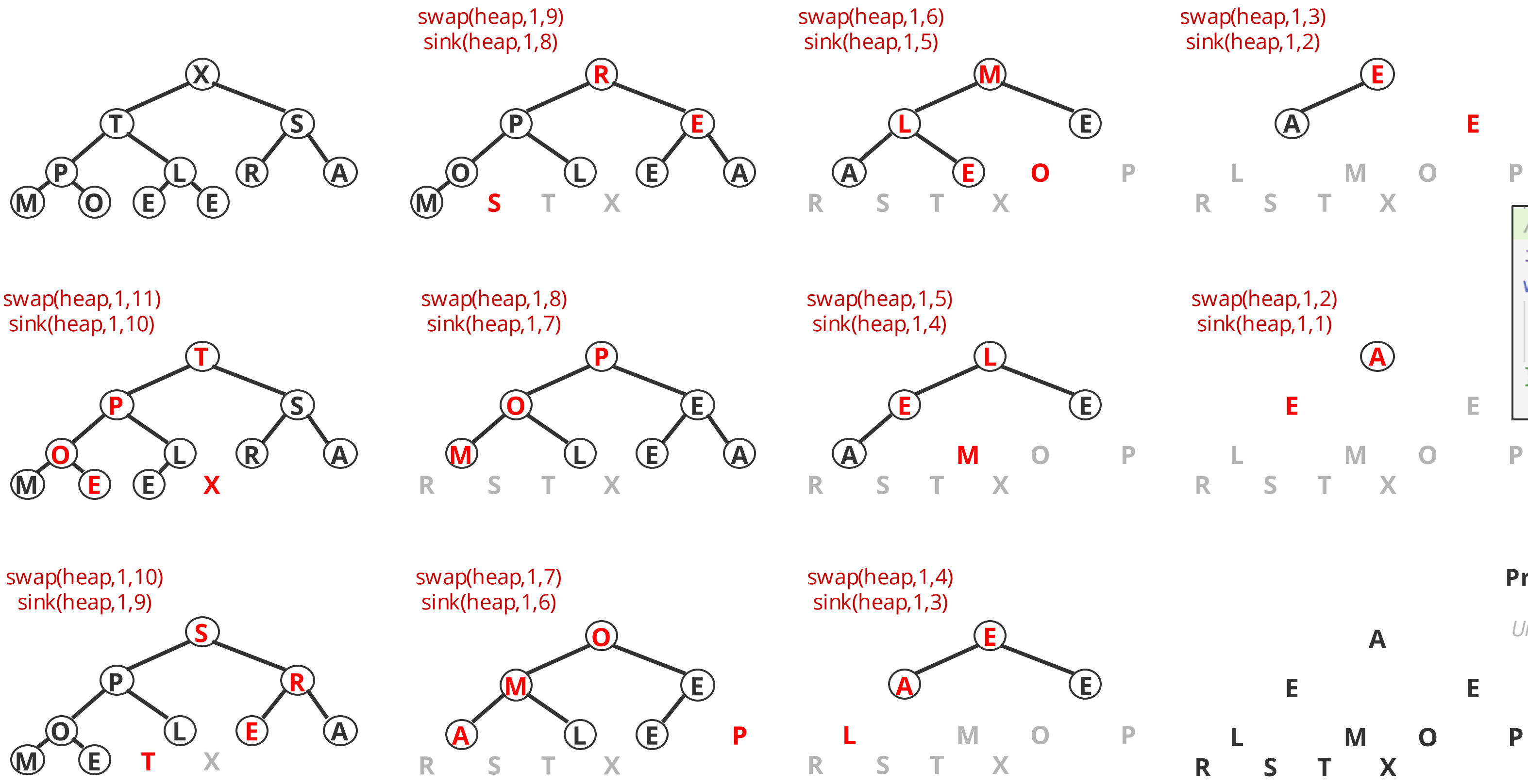
sink(heap,4,11)



sink(heap,2,11)



Heapsort: *Top-Down* Runtersortieren




```
// sortdown phase
int k = n;
while (k > 1) {
    swap(heap, 1, k--);
    sink(heap, 1, k);
}
```

heap.h

Heapsort: Mathematische Analyse

- **Satz**: Der Heapaufbau macht $\leq n$ Vertauschungen und $\leq 2n$ Vergleiche.
- **Beweis**: Sei $n = 2^{h+1} - 1$. Dann ist die Anzahl der Vertauschungen
$$h + 2(h-1) + 4(h-2) + 8(h-3) + \cdots + 2^h(0) = 2^{h+1} - h - 2 < n$$

- **Satz**: Das Runtersortieren braucht $\leq 2 \cdot n \log_2(n)$ Vergleiche und Vertauschungen!

- **Signifikanz**: *In-place* Sortieralgorithmus mit $O(n \log_2(n))$ Laufzeitgarantie!
  - *Mergesort*: Nicht *in-place* sondern extra Speicher
  - *Quicksort*: *In-place* aber schlechteste Laufzeit ist $O(n^2)$

- **In der Praxis** oft verwendet weil so einfach zu implementieren und kein zusätzlicher Speicher notwendig!

Zusammenfassung

- **Halden (*Heaps*)**
  - *Heaps* sind spezielle Bäume mit einer partiellen Ordnung
  - *Heaps* lassen sich ohne Kanten direkt in Feldern speichern
- **Haldenbedingung wiederherstellen *(heapify)***
  - Wenn die Haldenbedingung verletzt ist, kann sie durch rekursives Tauschen wieder hergestellt werden
  - Zwei Arten von Rekonstruktion: Aus Sicht des Eltern- oder Kindknoten
  - Korrektur hat höchsten $\log_2(n)$ viele Vergleiche und Vertauschungen
- **Anwendung von Heaps**
  - Ideale Datenstruktur für Prioritätswarteschlange, da Wurzel immer das Maximum
  - Kann auch zum Sortieren benutzt werden, wenn sukzessive das Maximum entfernt wird und die Heapbedingung wiederhergestellt wird
  - *Heapsort* ist perfekt in Komplexität und simpel zu implementieren

Viel Spaß bis zur nächsten Vorlesung!