1. Vorlesung (12.10.10)
Beispiel: Finde das Minimum einer Menge von Schlüsseln
\[ X = \{x_1, x_2, x_3, \ldots, x_n\} \]

(1) \( \text{MAX} = \infty \)
(2) \text{FOR } i = 1 \text{ TO } n \\
(3) \quad \text{IF } x_i < \text{MAX} \text{ THEN } \text{MAX} < x_i \\
(4) \quad \text{END FOR}

Algorithmus I benötigt \( O(n) \) "Schritte"

Als "Schritt" verstehen wir Instruktionen, die ein Rechner in \( O(1) \) Zeit erledigen kann, dies ist für die Vorlesung unser Maß für den Zeit- Aufwand (RAM) (z.B. +, -, *, /, 2^n, quadratische, aber nicht \( n^3 \))

Bessere als Alg I kann eine Anfrage nicht beantwortet werden!

Was aber wenn die Menge \( X \) dynamisch verwaltet werden soll?
Es empfiehlt sich auf eine Datenstruktur zurückzugreifen (HEAP).

Ein Heap unterstützt folgende Operationen:

- `Create_Heap()`: Erstellt einen Heap.
- `Insert(H, x)`: Fügt `x` in den Heap ein.
- `Minimum(H)`: Gibt das Minimum des Heaps zurück.
- `Extract_Min(H)`: Entfernt und gibt das Minimum des Heaps zurück.
- `Decrease_key(H, x, k)`: Verringert den Schlüssel von `x` um `k`.
- `Delete(H, x)`: Entfernt `x` aus dem Heap.
- `Union(H_1, H_2)`: Vereinigt `H_1` und `H_2`.
- `Build_Heap(A)`:

Anmerkung: Wir speichern Elemente `X`, die ein Feld `x.key` haben, welches den Schlüssel enthält
(sowie andere Felder).

Ideal: Wir speichern `X` in einem Binärbäum ab (alle Ebenen, bis auf die letzte
sind hierbei komplett gefüllt!)

Zusätzlich speichern wir mit `X` die `#` der Elemente
fälle von oben nach unten.

von rechts nach links

-2
Diesen Baum speichern wir in einem Array ab:

\[
\begin{array}{cccc}
\hline
4 & 2 & 3 & 4 \\
\hline
| X_5 | X_2 | & \\
\hline
\end{array}
\]

Navigation im Baum:

- `parent(i) = \lfloor i/2 \rfloor` (parent)
- `left(i) = 2i` (linkes Kind)
- `right(i) = 2i+1` (rechtes Kind)

Wir sagen ein Binärbaum hat die **HEAP-EIGENSCHAFT**

\[\text{\(i > 0\)} \implies \text{\(A[i].key \leq A[\text{parent}(i)].key\)}\]

der kleinsten Schlüssel aus \(B_i\)

Teilbaum unter \(i\)
Wichtige Hilfsprozedur:

`HEAPIFY(A, i)`

Stellt die Heap-Eigenschaft für B_i wieder her unter der Voraussetzung, dass die Heap-Eigenschaft für B_left & B_right gilt.

In Pseudo-Code: (Wir ignorieren die Tests ob left(i), right(i) und existieren)

```plaintext
HEAPIFY (A, i)

1) Smallest = minz: A[left(i)].key, left(A[left(i)]).key

2) Smallest = i

3) IF A[LEFT(i)].key < A[i].key
   THEN Smallest = LEFT(i)

4) IF A[RIGHT(i)].key < A[Smallest].key
   THEN Smallest = RIGHT(i)

5) IF Smallest ≠ i THEN EXCHANGE A[i] & A[Smallest]

6) HEAPIFY [A, Smallest]

7) HEAPIFY [A, Smallest]  
```
Beispiel (Zeigt warum der rek. Aufruf (+) notwendig ist)

Nach (1) - (5) erhalten wir: Smallest = 4

Heap Eigenschaft gesichert

Heap Eigenschaft bleibt erhalten

Also müssen wir rekursiv heapify aufrufen (7) wenn getauscht wurde!
Aufwand Heapify:

\[ T(n) = T\left(\frac{2}{3}n\right) + \Theta(1) \]

\[ n = x + y + 1 = x + y + 1 \Rightarrow n \geq \frac{3}{2}x + 1 \geq \frac{3}{2}x \]

\[ \Rightarrow x \leq \frac{2}{3}n \]

\[ \Rightarrow T(n) = O(\log n) \]

Anmerkung (wenn nie weiter das gesagt wird)
Zeit inner — worst case

Bild

Erstellung eines Heaps (A)  (falls fehl) > 15 Minuten

Eingabe: unsortiertes Array A
Ausgabe: A als Heap

Pseudo Code:

\[ \left\lfloor \frac{n}{2} \right\rfloor \]

(1) FOR \( i = \#\text{Elemente} \) \( n \rightarrow A \) DOWTO 1

(2) HEAPIFY (A, i)

(3) END FOR

Anmerkung: wir sperren uns bei (1) die Bäume, und konstruieren die Heap Eigenschaft Bottom-up
Aufwand: Create-Heap?

Große Analyse: \( \frac{n}{2} \times O(\text{Heapify}) \rightarrow O(n \log n) \)

Genauere Analyse: Für einen Knoten \( i \) mit \( \text{height}(i) = k \) braucht \( \text{Heapify} \) \( O(\log k) \), also \( O(h) \).

Tiefe von \( B_i \)

Also

\[ \begin{array}{c}
\text{für } k = \lfloor \log n \rfloor \\
\begin{array}{c}
\downarrow \\
\text{für } k = \lfloor \log n \rfloor \\
\begin{array}{c}
\downarrow \\
\text{für } k = \lfloor \log n \rfloor \\
\end{array}
\end{array}
\end{array} \]

Also

\[ \sum_{j=0}^{k} 2^j (k-j) = \sum_{j=0}^{k} 2^j j = 2^k \sum_{j=0}^{k} j = 2^k \cdot \frac{k(k+1)}{2} \]

Konvergenz

\[ \sum_{j=0}^{k} 2^j (k-j) = 2^k \cdot (k+1) - 2^k = 2^k \cdot (\log n) \]

Die zweite Teil wie \( \log n + \text{was konkreter die Heapelgenschaft for free} \).
Jetzt einfacher:

Minimum (A):

(1) RETURN A[1] → O(1)

Extract-Min (A)

(1) Min ← A[1]
(3) n ← n - 1
(4) HEAPIFY (A, 1)
(5) Return Min

Insert (A)

Decrease Key (A, i, newkey)

(1) A[i] ← min (A[i], newkey)
(2) WHILE (i > 1 & A[Par[i]] > A[i].key)
(3) DO A[i] ← A[Par[i]]
(4) i ← Par[i]
(5) ENDO
Erklärung:

Heap-eigenschaft bleibt für Bi & Bi+i erhalten, aber echt nicht für Bi

\[ A[j] \leq A[x] \text{ für } x \in B_i \}

\[ \forall j \\text{ Tausch } A[j] \leftrightarrow A[i] \text{ sicher Heap-eigenschaft für } B_i \text{ (wenn } i < h) \]

1) jetzt ist die Situation wie eine Decrease_key \( A[i], \text{new key} \)

Aufwand: klar, \( O(\log n) \)

**Insert \((A, \text{key}) X\)**

1. \( \text{key} \leq X, \text{key} \)
2. \( X, \text{key} \leq +\infty \)
3. \( n \leq h+1 \)
4. \( A[Iu] \leq X \)
5. \( \text{Decrease_key } (A, u, \text{key}) \)
Delete \((A, i)\) & Union \((A_1, A_2)\) → Homework

<table>
<thead>
<tr>
<th>Operation</th>
<th>Create Heap</th>
<th>Copy Heap</th>
<th>Insert</th>
<th>Delete</th>
<th>Min/Max</th>
<th>Extract Min/Max</th>
<th>Enqueue, Kost</th>
<th>Union</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Heap</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
<td>Θ(log n)</td>
<td>Θ(1)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>w. Indirect</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ziel</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(4)</td>
<td>Θ(log n)</td>
<td>Θ(1)</td>
<td>Θ(log n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(4)</td>
<td>Θ(log n)</td>
<td>Θ(1)</td>
<td>Θ(log n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
</tr>
</tbody>
</table>