Binäre Heaps

Übersicht

- Bäume in denen Elternknoten größer/kleiner als alle Kindknoten sind
  - größter/kleinster Knoten dementsprechend ganz oben

Übersicht

- Bäume in denen Elternknoten größer/kleiner als alle Kindknoten sind
  - größter/kleinster Knoten dementsprechend ganz oben
- Darstellbar mit Arrays:
  - Elternknoten an Position i, Kindknoten an Positionen $2i$ und $2i + 1$ (Wurzel: $i = 1$)

Übersicht

- Bäume in denen Elternknoten größer/kleiner als alle Kindknoten sind
  - größter/kleinster Knoten dementsprechend ganz oben
- Darstellbar mit Arrays:
  - Elternknoten an Position i, Kindknoten an Positionen $2i$ und $2i + 1$ (Wurzel: $i = 1$)
- Erzeugbar in $O(n)$
- Entfernen der Wurzel in $O(\log n)$ (unter Invariantenerhalt)
- Basis für Heapsort

Heapify

- Basis für Erzeugung
- Hier: $n = 4 \times 2^m$, min-heap

```
fun heapify(Array A, index i):
    assert(A[2*i] and A[2*i+1] are roots of valid heaps)

    // out of range is never minimal:
    index min := index_of_min(A[i], A[2*i], A[2*i+1])
    if min != i:
        swap(A[i], A[min])
        heapify(A, min)
```

Erzeugung

- Heapeigenschaft in rechter Hälfte erfüllt (trivial)

```
fun build_heap(A: Array):
    for i := i/2 to 0:
        heapify(A, i)
```

- Laufzeit: $O(n)$
  - $O\left(\sum_{i=1}^{\log n} \frac{i \times n}{2^i}\right) = O\left(n \times \sum_{i=1}^{\log n} \frac{i}{2^i}\right) = O(n)$

Sift-Up

```
fun sift_up(A: Array, i: index):
    assert(A is valid heap, except maybe for index i)
    if i=0 or A[i/2] <= A[i]:
        return
    swap(A[i], A[i/2])
    sift_up(A, i/2)
```

Insert

```
fun insert(A: Array, val: T):
    A.push_back(T)
    sift_up(a, a.size() - 1)
```

Sift-Down

```
fun sift_down(A: Arrau, i: index):
    assert(A is valid heap except maybe at the
           children of A[i])
    if A[i] is leaf: return
    index min := index_of_min(A[2*i], A[2*i+1])
    if A[min] < A[i]:
        swap(A[min], A[i])
        sift_down(A, min)
```

- Genau ein Kindknoten: Anpassung trivial
- Laufzeit: $O(\log n)$

Remove-min

```
fun remove_min(A: Array): T
    retval := A[0]
    A[0] := A[A.size()-1]
    A.resize(A.size()-1)
    sift_down(A, 0)
    return retval
```

- Laufzeit: $O(\log n)$

Heapsort

- Erzeuge max-Heap in $O(n)$
- n-mal:
  - vertausche Wurzel mit letztem Element des Heaps
  - reduziere Heap-Größe um 1
  - sift-down(0)

Heapsort

- Erzeuge max-Heap in $O(n)$
- n-mal:
  - vertausche Wurzel mit letztem Element des Heaps
  - reduziere Heap-Größe um 1
  - sift-down(0)

- Laufzeit offensichtlich in $O(n \log n)$ (worstcase)
- instabil
- echt inplace (nicht pseudo-inplace wie Quicksort)
- Kombinierbar mit Quicksort (=Introsort)

Kreativaufgaben

Bulk-Insertion

Gegeben sei ein binärer Heap der n Elemente enthält. Es sollen nun k Elemente auf einmal eingefügt werden.

Geben Sie ein Verfahren an (kein Pseudocode), mit dem man die Einfügung in $O(\min\{k \log k + \log n, k + \log n \log k\})$ Schritten erledigen kann.

Sie können davon ausgehen, dass der Heap genau $2m - 1$ Elemente enthält ($m \in \mathbb{N}$).