Informatik II

Übung 8

FS 2019

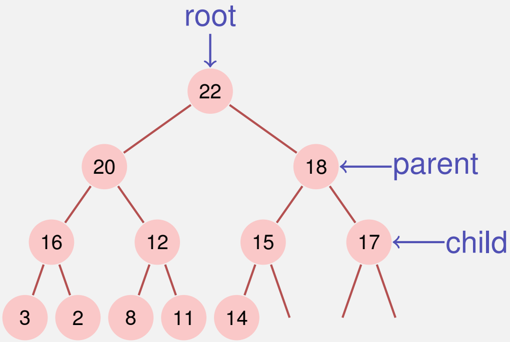
Program Today

1. Repetition Heaps

Max-]Heap[1]

Binary tree with the following properties

1. complete up to the lowest level
2. Gaps (if any) of the tree in the last level to the right
3. *Heap-Condition:* Max-(Min-)Heap: key of a child smaller (greater) that that of the parent node

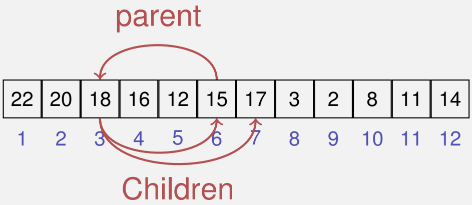


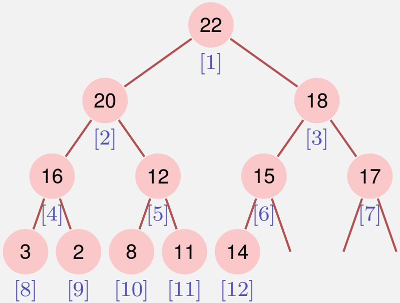
[1] Heap(data structure), not: as in “heap and stack” (memory allocation)

Heap and Array

Tree $\rightarrow$ Array:

- children$(i) = \{2i, 2i+1\}$
- parent$(i) = \lfloor i/2 \rfloor$





Depends on the starting index[2]

[2] For array that start at 0: $\{2i, 2i+1\} \rightarrow \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i-1)/2 \rfloor$

Height of a Heap

A complete binary tree with height[3] h provides

$$1 + 2 + 4 + 8 + ... + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

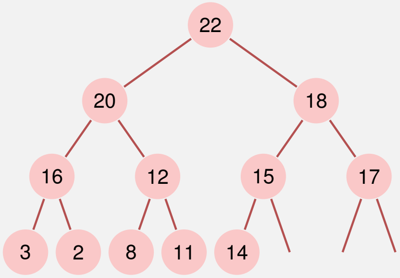
nodes. Thus for a heap with height h:

$$\begin{aligned} & 2^{h-1} - 1 < n \leq 2^h - 1 \\ \Leftrightarrow \quad & 2^{h-1} < n + 1 \leq 2^h \end{aligned}$$

Particularly $h(n) = \lceil \log_2(n+1) \rceil$ and $h(n) \in \Theta(\log n)$.

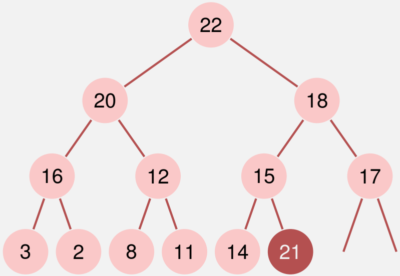
[3]here: number of edges from the root to a leaf

Insert



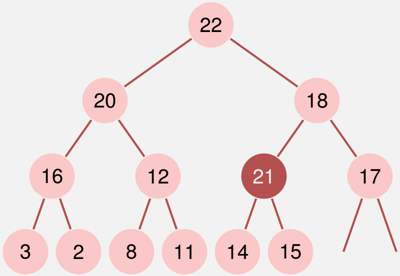
Insert

- Insert new element at the first free position. Potentially violates the heap property.



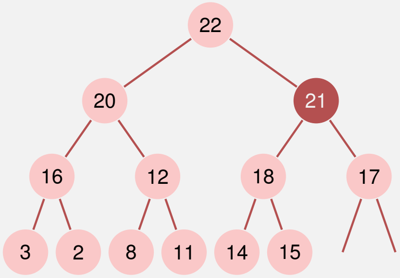
Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively



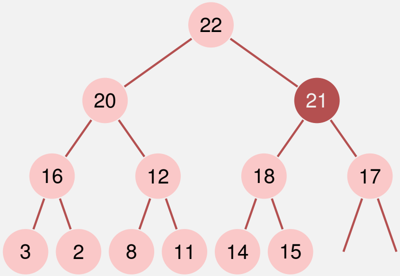
Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively

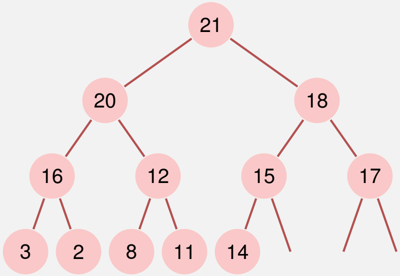


Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively
- Worst case number of operations: $\mathcal{O}(\log n)$

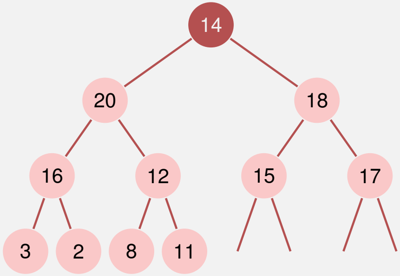


Remove the maximum



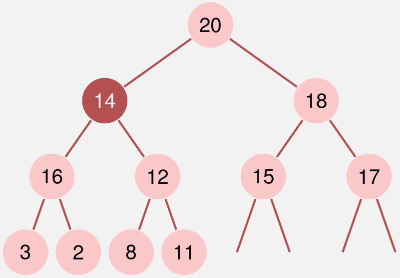
Remove the maximum

- Replace the maximum by the lower right element



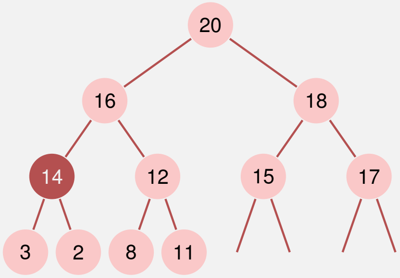
Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sift down successively (in the direction of the greater child)



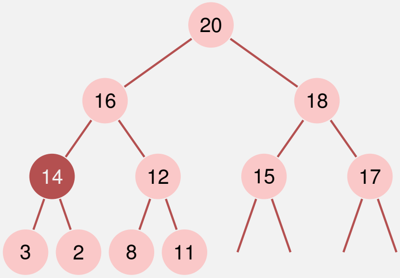
Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sift down successively (in the direction of the greater child)



Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sift down successively (in the direction of the greater child)
- Worst case number of operations: $\mathcal{O}(\log n)$



Algorithm SiftDown(A, i, m)

Input: Array A with heap structure for the children of i. Last element m.

Output: Array A with heap structure for i with last element m.

```
while 2i ≤ m do
    j ← 2i; // j left child
    if j < m and A[j] < A[j + 1] then
        j ← j + 1; // j right child with greater key
    if A[i] < A[j] then
        swap(A[i], A[j])
        i ← j; // keep sinking down
    else
        i ← m; // sift down finished
```

Sort heap

7	6	4	5	1	2

$A[1, ..., n]$ is a Heap.

While $n > 1$

- swap($A[1]$, $A[n]$)
- SiftDown($A, 1, n-1$);
- $n \leftarrow n - 1$

Sort heap

	7	6	4	5	1	2
swap $\Rightarrow$	2	6	4	5	1	7

$A[1, ..., n]$ is a Heap.
While $n > 1$

- swap($A[1]$, $A[n]$)
- SiftDown($A, 1, n-1$);
- $n \leftarrow n - 1$

Sort heap

$A[1, ..., n]$ is a Heap.
While $n > 1$

- swap($A[1]$, $A[n]$)
- SiftDown($A, 1, n-1$);
- $n \leftarrow n - 1$

		7	6	4	5	1	2
swap	$\Rightarrow$	2	6	4	5	1	7
siftDown	$\Rightarrow$	6	5	4	2	1	7

Sort heap

	7	6	4	5	1	2
swap $\Rightarrow$	2	6	4	5	1	7
siftDown $\Rightarrow$	6	5	4	2	1	7
swap $\Rightarrow$	1	5	4	2	6	7

$A[1, ..., n]$ is a Heap.

While $n > 1$

- swap($A[1]$, $A[n]$)
- SiftDown($A, 1, n - 1$);
- $n \leftarrow n - 1$

Sort heap

$A[1, ..., n]$ is a Heap.

While $n > 1$

- swap($A[1]$, $A[n]$)
- SiftDown$(A, 1, n-1)$;
- $n \leftarrow n - 1$

		7	6	4	5	1	2
swap	$\Rightarrow$	2	6	4	5	1	7
siftDown	$\Rightarrow$	6	5	4	2	1	7
swap	$\Rightarrow$	1	5	4	2	6	7
siftDown	$\Rightarrow$	5	4	2	1	6	7
swap	$\Rightarrow$	1	4	2	5	6	7
siftDown	$\Rightarrow$	4	1	2	5	6	7
swap	$\Rightarrow$	2	1	4	5	6	7
siftDown	$\Rightarrow$	2	1	4	5	6	7
swap	$\Rightarrow$	1	2	4	5	6	7

Heap creation

Observation: Every leaf of a heap is trivially a correct heap.

Consequence:

Heap creation

Observation: Every leaf of a heap is trivially a correct heap.

Consequence: Induction from below!

Algorithm HeapSort(A, n)

Input: Array A with length n.
Output: A sorted.

```
// Build the heap.
for i ← n/2 downto 1 do
    SiftDown(A, i, n);
// Now A is a heap.
for i ← n downto 2 do
    swap(A[1], A[i])
    SiftDown(A, 1, i − 1)
// Now A is sorted.
```

Analysis: sorting a heap

SiftDown traverses at most $\log n$ nodes. For each node 2 key comparisons. $\Rightarrow$ sorting a heap costs in the worst case $2 \log n$ comparisons.

Number of memory movements of sorting a heap also $\mathcal{O}(n \log n)$.

Analysis: creating a heap

Calls to siftDown: $n/2$. Thus number of comparisons and movements: $v(n) \in \mathcal{O}(n \log n)$.

Analysis: creating a heap

Calls to siftDown: $n/2$. Thus number of comparisons and movements: $v(n) \in \mathcal{O}(n \log n)$.

But mean length of the sift-down paths is much smaller and it holds that :

$$v(n) \in \mathcal{O}(\mathbf{n})$$

Questions / Suggestions?