# CMPS 2200 – Fall 2017

# *Heaps*

**Carola Wenk**

# Priority Queue

A **priority queue** is a data structure which supports operations

- Insert
- Find_max
- Extract_max

Several possible implementations:

| | Insert | Find_max | Extract_max |
| --- | --- | --- | --- |
| Unsorted array: | $O(1)$ | $O(n)$ | $O(n)$ |
| Sorted array: | $O(n)$ | $O(1)$ | $O(n)$ or O(1) |
| Balanced BST: | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| Heaps: | $O(\log n)$ | $O(1)$ | $O(\log n)$ |
| Fibonacci Heaps: | $O(1)$ amortized | $O(1)$ amortized | $O(\log n)$ amortized |

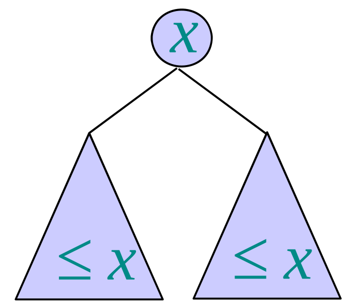
# Heaps

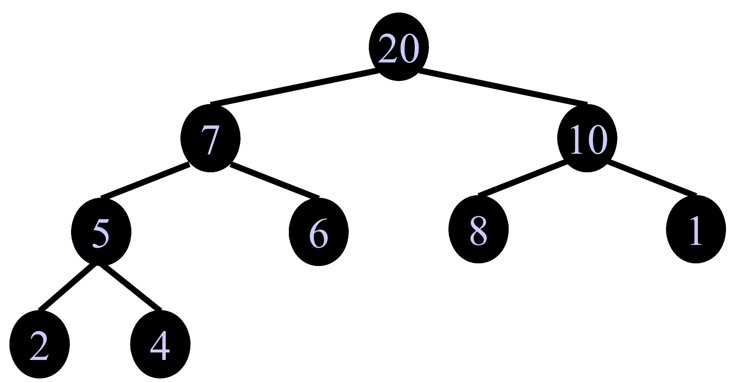
1)

- A max-heap is an almost complete binary tree (flushed left on the last level). Each node stores a key. The tree

2) fulfills the **max-heap property**:

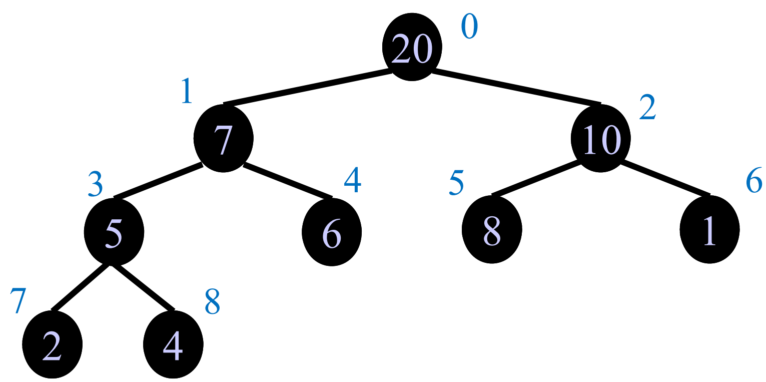
> For every node $x$ holds:
>
> - $y \leq x$ , for all $y$ in any subtree of $x$



# Heap Storage

- Because a max-heap is an almost complete binary tree it can be stored in an array level by level:




| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 20 | 7 | 10 | 5 | 6 | 8 | 1 | 2 | 4 |

- Implement child/parent "pointers":

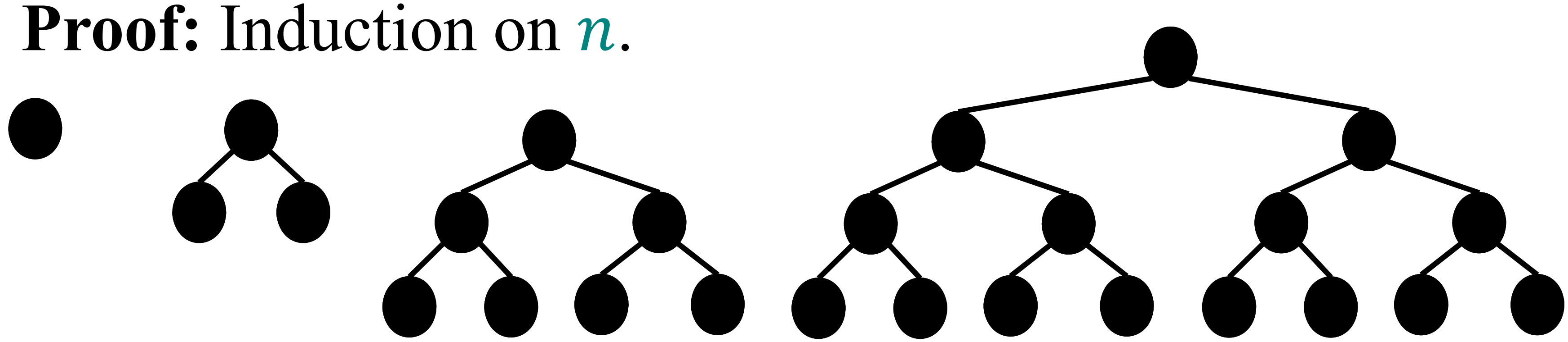
$parent(i) = \left\lfloor \frac{i-1}{2} \right\rfloor \quad left(i) = 2i+1 \quad right(i) = 2i+2$

- Find_max: O(1) time

# Heap Height

- **Lemma:** A complete binary tree of height $h$ has $n = 2^{h+1} - 1$ nodes.

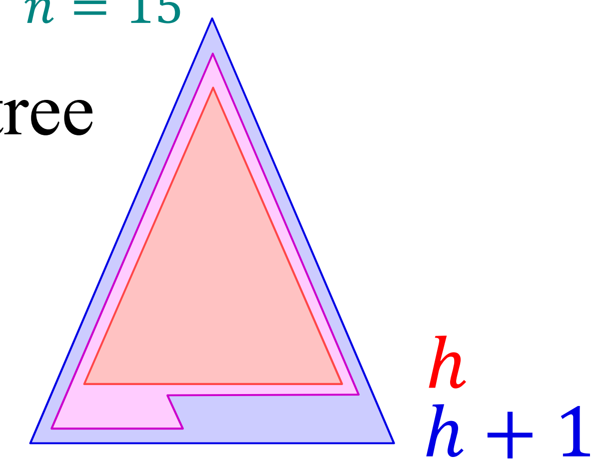
  **Proof:** Induction on $n$.



$h = 0$, $n = 1$ $\quad$ $h = 1$, $n = 3$ $\quad$ $h = 2$, $n = 7$ $\quad$ $h = 3$, $n = 15$

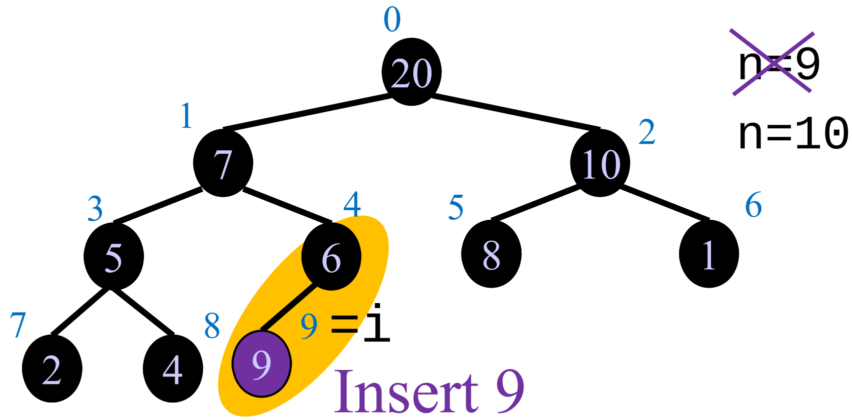
- **Lemma:** An almost complete binary tree with $n$ nodes has height $h = \lfloor \log n \rfloor$.

  **Proof idea:** $2^h - 1 < n \leq 2^{h+1} - 1$.



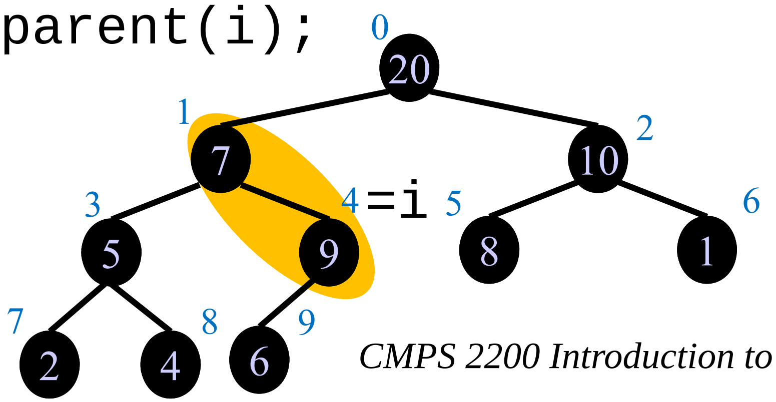
# Insert, Heapify_up : O(*h*)=O(log *n*)

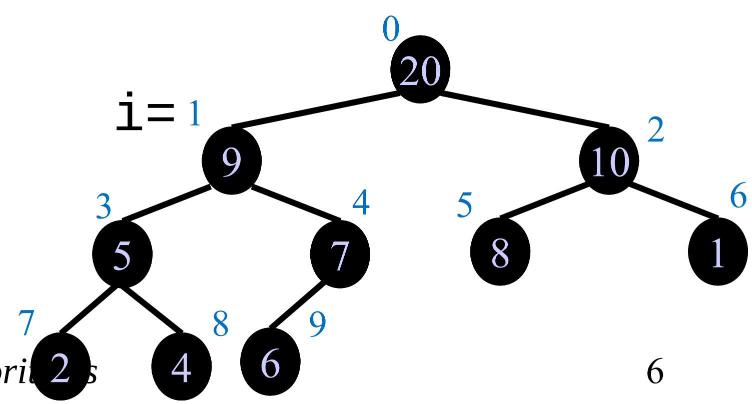
```
Insert(A,n,key){
  n++;
  A[n-1]=key;
  Heapify_up(A,n-1);
}
```




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A: | 20 | 7 | 10 | 5 | 6 | 8 | 1 | 2 | 4 | 9 |

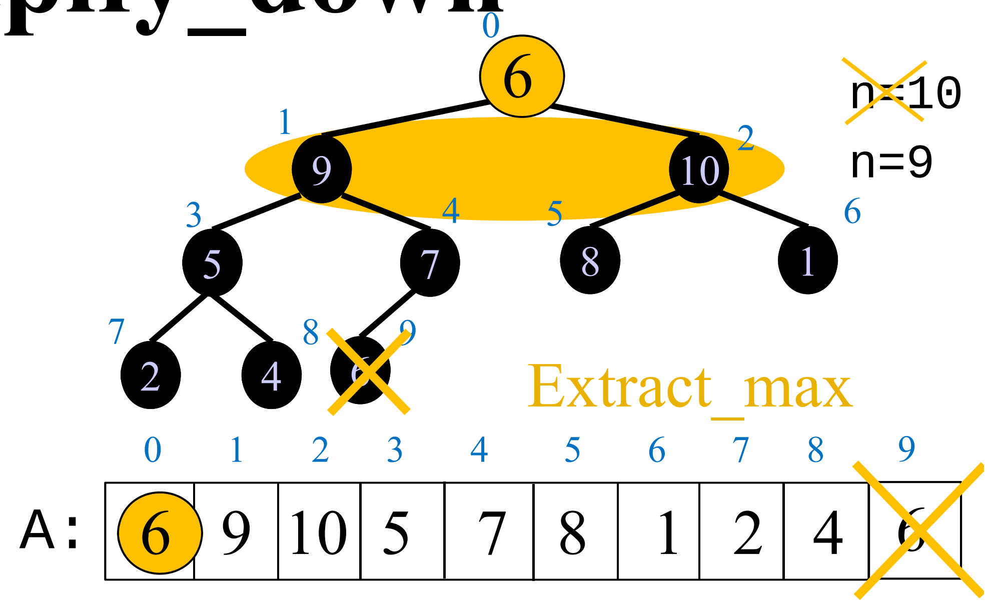
```
Heapify_up(A,i){
  while(i>0 && A[parent(i)]<A[i]){
    swap(A[parent(i)],A[i]);
    i=parent(i);
  }
}
```



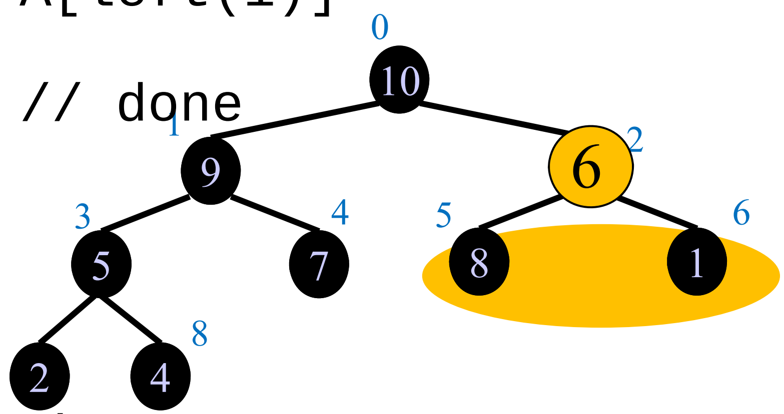



# Extract_max, Heapify_down

```
Extract_max(A,n,key){
  max=A[0];
  A[0]=A[n-1];
  n--;
  Heapify_down(A,n,0);
  return max;
}
```

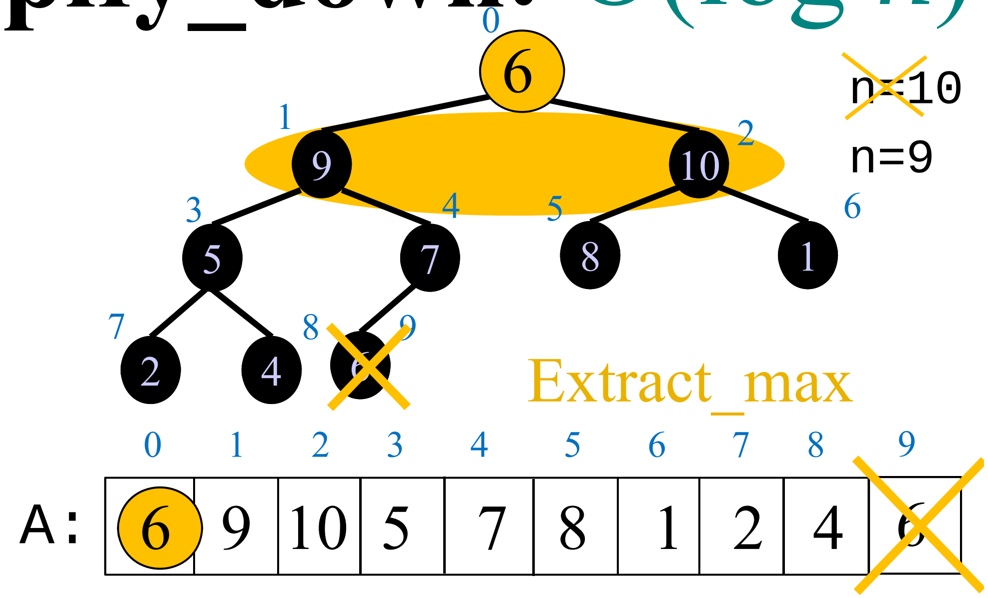



```
Heapify_down(A,n,i){
  while(left(i)<n){//left child exists
    maxchild=left(i);
    if(right(i)<n && A[right(i)]>A[left(i)]
      maxchild =right(i);
    if(A[maxchild]<=A[i]) break; // done
    swap(A[i], A[maxchild]);
    i=maxchild;
  }
}
```

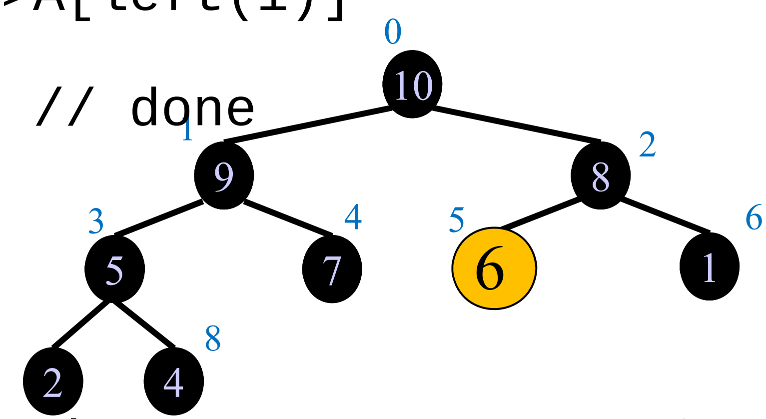


# Extract_max, Heapify_down: O(log *n*)

```
Extract_max(A,n,key){
  max=A[0];
  A[0]=A[n-1];
  n--;
  Heapify_down(A,n,0);
  return max;
}
```




```
Heapify_down(A,n,i){
  while(left(i)<n){//left child exists
    maxchild=left(i);
    if(right(i)<n && A[right(i)]>A[left(i)]
      maxchild =right(i);
    if(A[maxchild]<=A[i]) break; // done
    swap(A[i], A[maxchild]);
    i=maxchild;
  }
}
```



# Heapsort : O($n$ log $n$)

- Insert all numbers in a max-heap
- Repeatedly extract max

```
Heapsort(A,n){
  Build_heap(A); //Insert all elements
  for(i=n-1; i>=1; i--){
    swap(A[0],A[i]); // moves max to A[n]
    n--;
    Heapify_down(A,n,0);
  }
}
```

O($n$ log $n$) (Build_heap)

O($n$ log $n$) (for loop)