CMPS 6610/4610 – Fall 2016

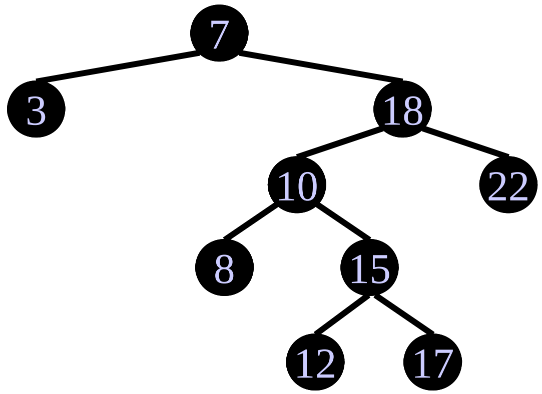
Heaps and Binary Search Trees

Carola Wenk

Dynamic Set

A **dynamic set**, or **dictionary**, is a data structure which supports operations

- Insert
- Delete
- Find



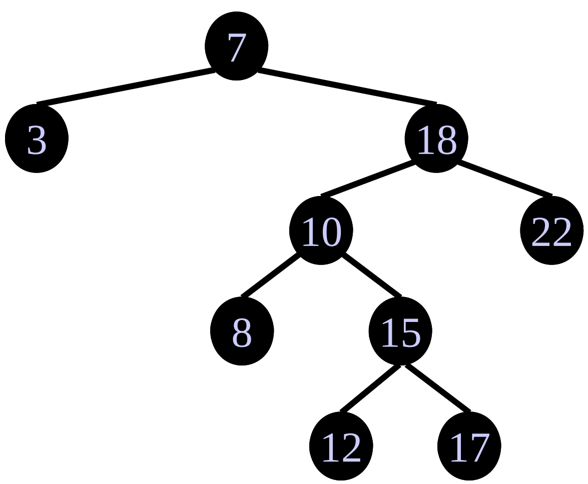
Using **balanced binary search trees** we can implement a dictionary data structure such that each operation takes $O(\log n)$ time.

Search Trees

- A binary search tree is a binary tree. Each node stores a key. The tree fulfills the **binary search tree property**:

For every node x holds:

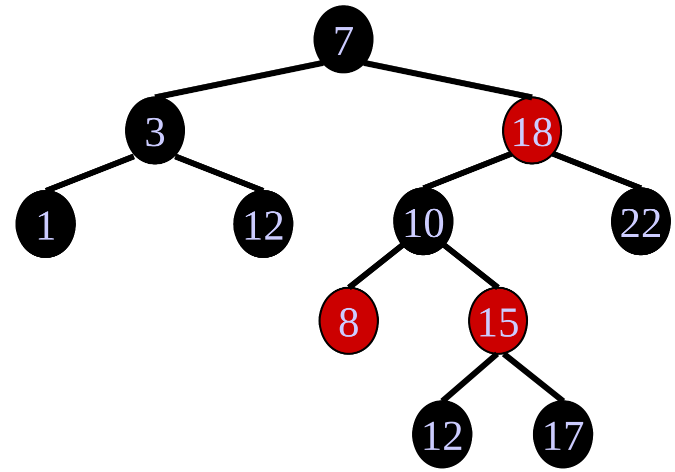
- $y \leq x$, for all y in the subtree left of x
- $x < y$, for all y in the subtree right of x



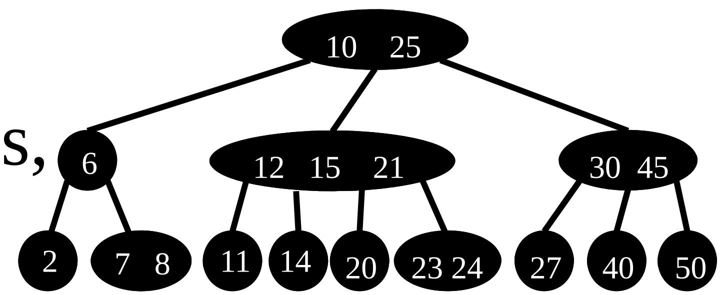
Search Trees

Different variants of search trees:

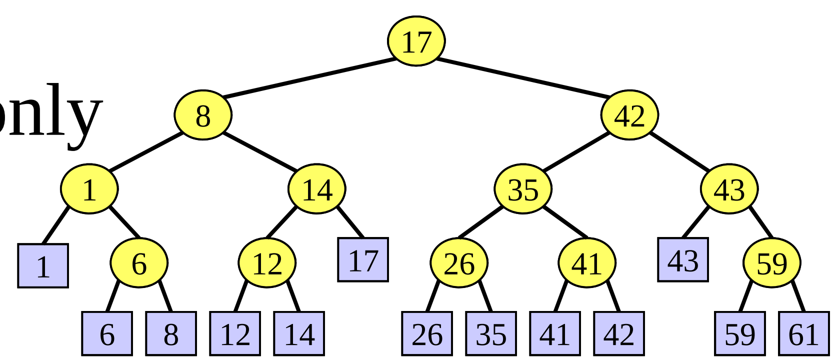
- Balanced search trees (guarantee height of $O(\log n)$ for n elements)



- *k*-ary search trees (such as B-trees, 2-3-4-trees)



- Search trees that store keys only in leaves, and store copies of keys as split-values in internal nodes



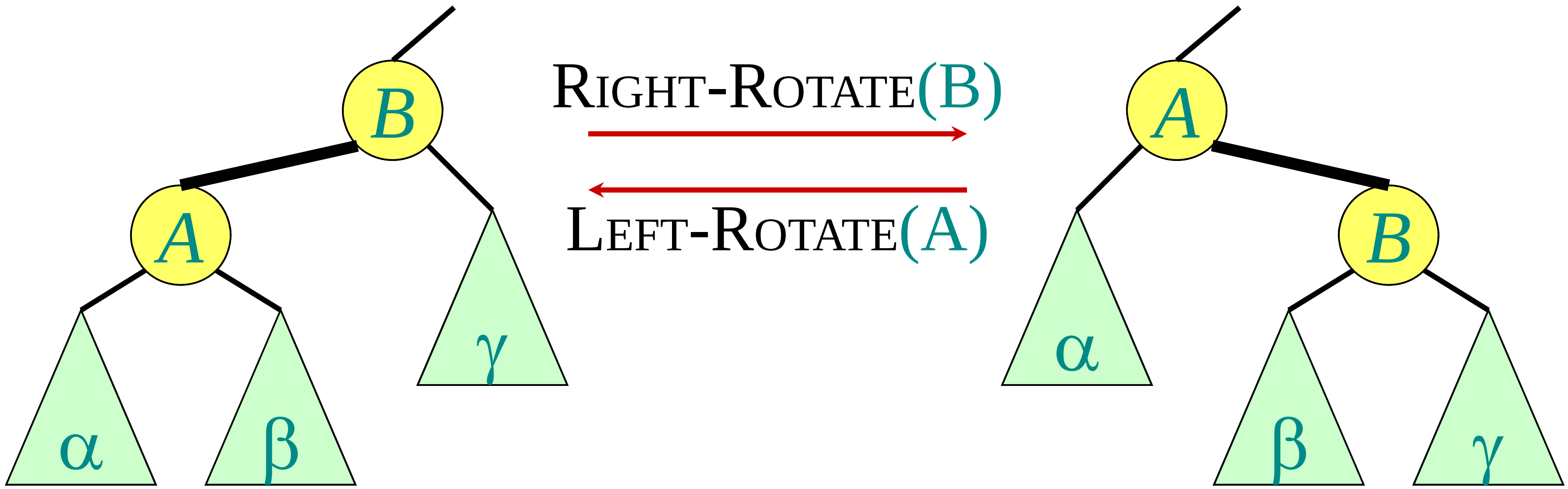
Balanced search trees

Balanced search tree: A search-tree data structure for which a height of $O(\log n)$ is guaranteed when implementing a dynamic set of n items.

Examples:

- AVL trees
- Red-black trees
- 2-3 trees
- 2-3-4 trees
- B-trees

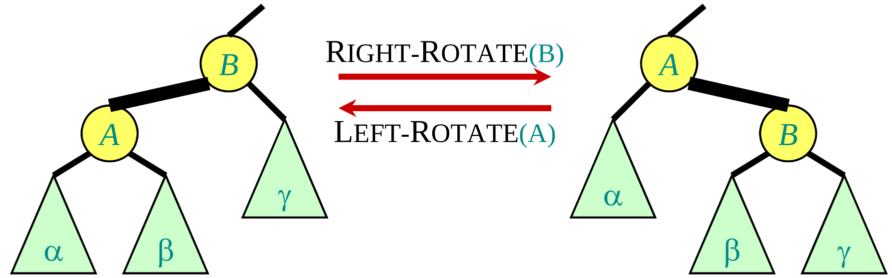
Rotations

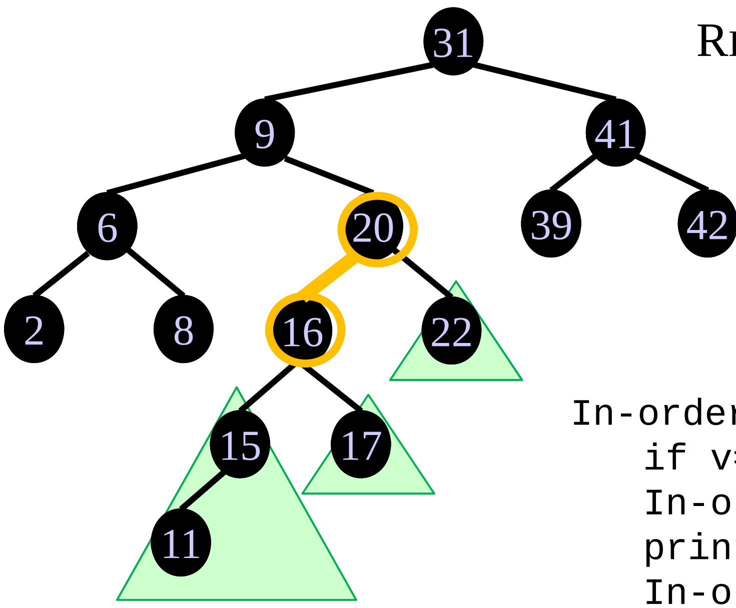


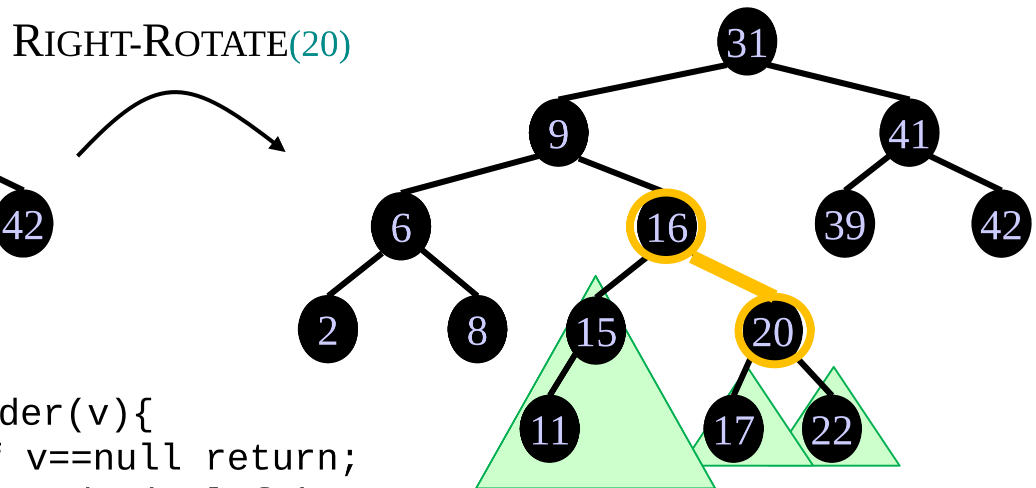
- Rotations maintain the inorder ordering of keys:

  $a \in \alpha, b \in \beta, c \in \gamma \;\Rightarrow\; a \le A \le b \le B \le c.$
- Rotations maintain the binary search tree property
- A rotation can be performed in $O(1)$ time.

Rotation Example








```
In-order(v){
    if v==null return;
    In-order(v.left);
    print(v.key+" ");
    In-order(v.right);
}
```

In-order traversal:
2 6 8 9 11 15 16 17 20 22 31 39 41 42

In-order traversal:
2 6 8 9 11 15 16 17 20 22 31 39 41 42

⇒ Maintains sorted order of keys, and can reduce height

Priority Queue

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

- Insert
- Find_max
- Extract_max

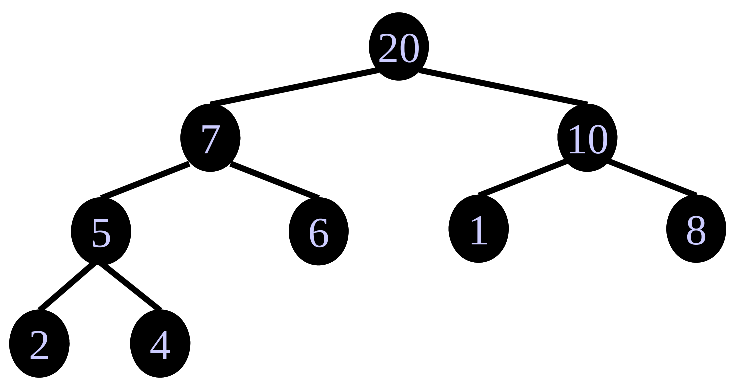
Several possible implementations:

	Insert	Find_max	Extract_max
Unsorted array:	$O(1)$	$O(n)$	$O(n)$
Sorted array:	$O(n)$	$O(1)$	$O(n)$
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

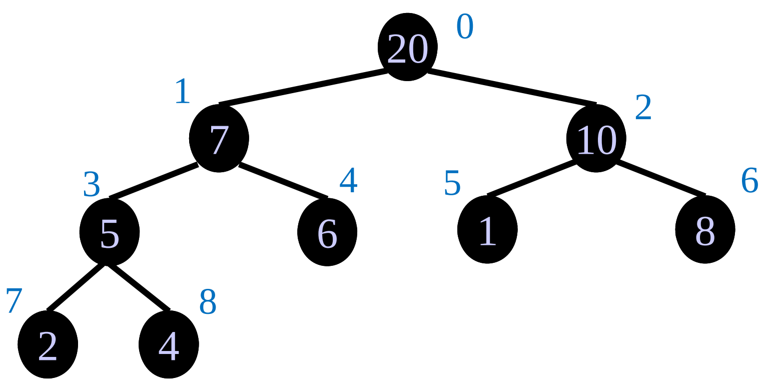
- A max-heap is an almost complete binary tree (flushed left on the last level). Each node stores a key. The tree fulfills the **max-heap property**:

> 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 it can be stored in an array level by level:



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

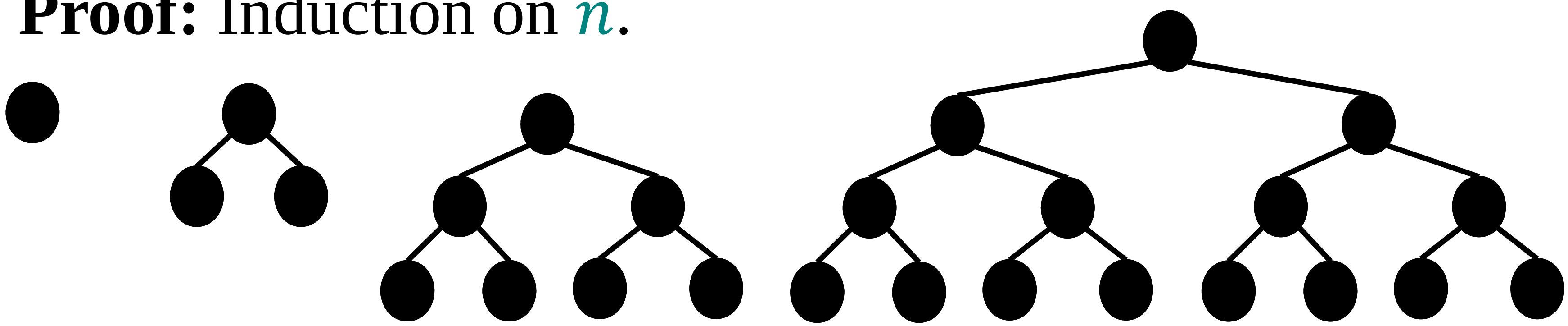
- Implement child/parent "pointers":

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

- Find_max: O(1) time

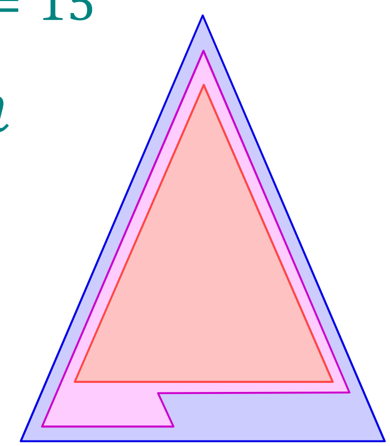
Heap Height

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



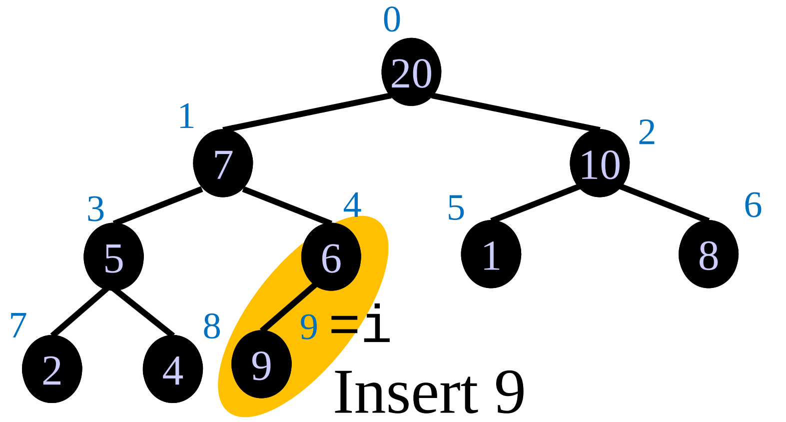
$h = 0$, $n = 1$ $h = 1$, $n = 3$ $h = 2$, $n = 7$ $h = 3$, $n = 15$

- **Lemma:** An almost complete 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)

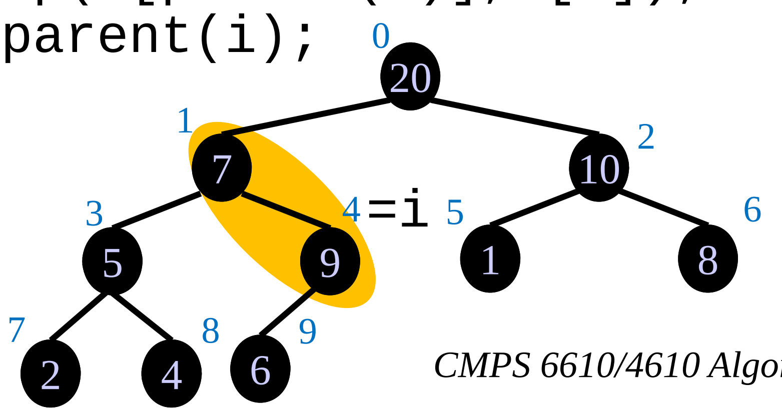
```
Insert(A,n,key){
  n++;
  A[n-1]=key;
  Heapify_up(A,n-1);
}
```

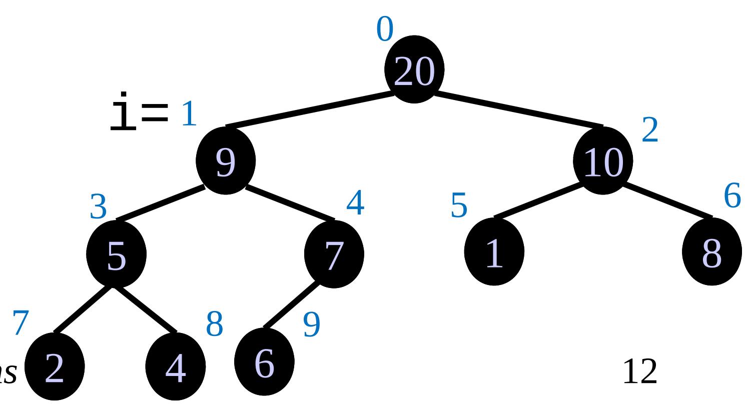


A:

0	1	2	3	4	5	6	7	8	9
20	7	10	5	6	1	8	2	4	9

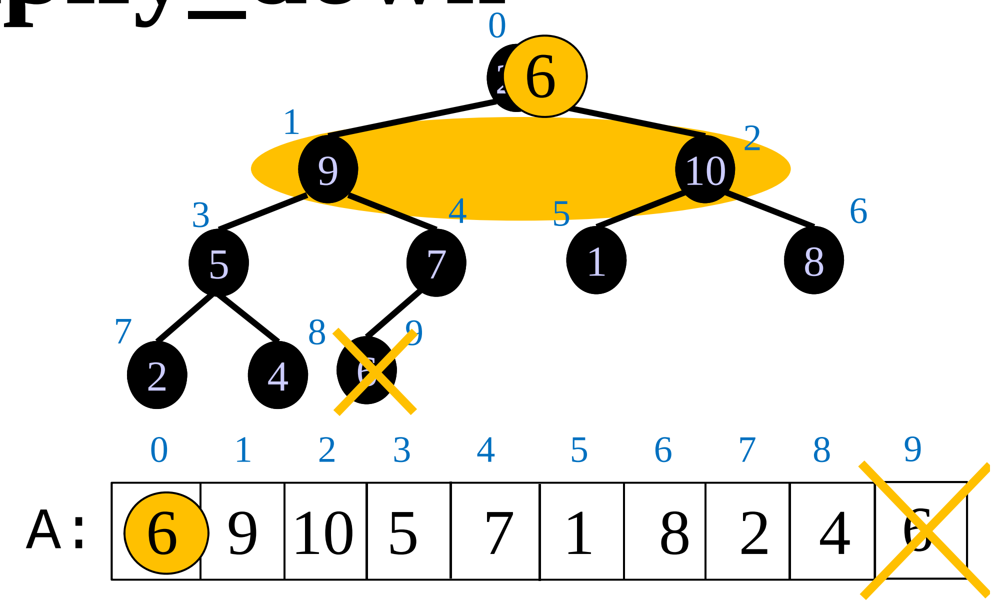
```
Heapify_up(A,i){
  while(i>0 && A[parent(i)]<A[i]){
     swap(A[parent(i)],A[i]);
     i=parent(i);
  }
}
```



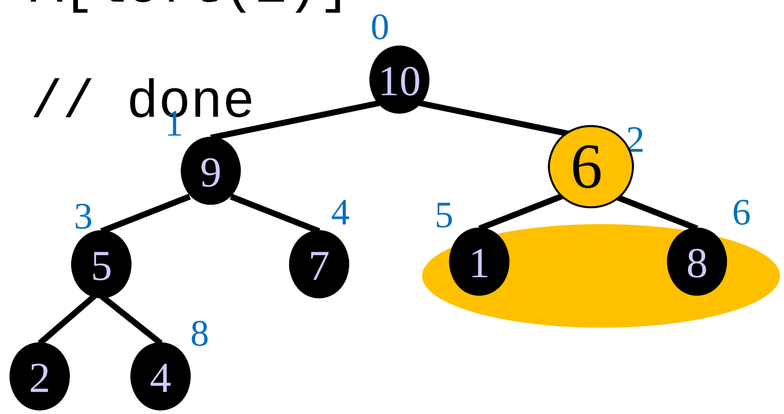


Extract_max, Heapify_down

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

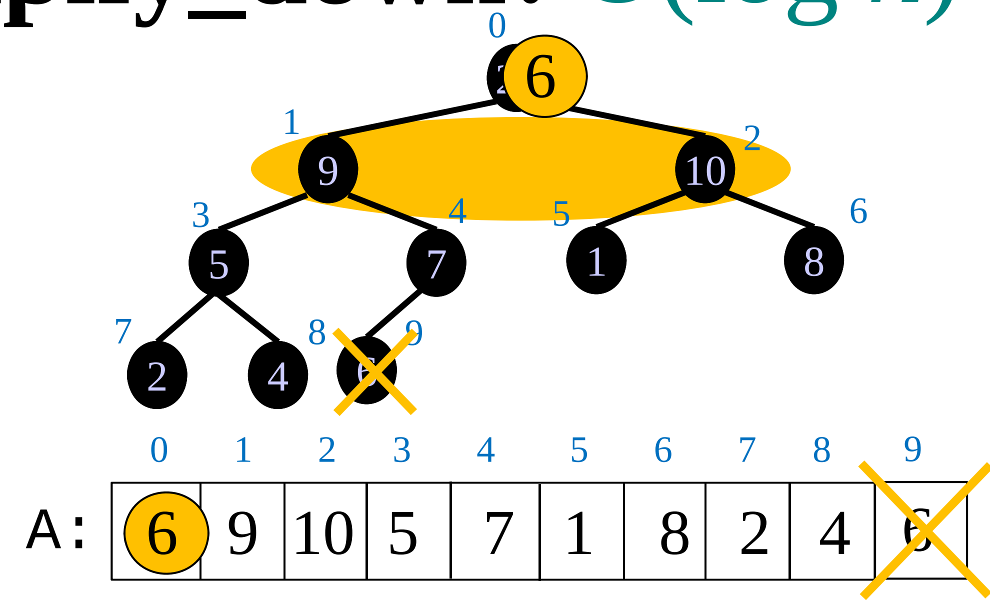



```
Heapify_down(A,n,i){
  while(left(i)<n){//left child exists
    maxchild=left(i);
    if(right(i)<n && A[right(i)]>A[left(i)]
      maxchild =right(i);
    if(A[maxchild]<=A[i]) break; // done
    swap(A[i], A[maxchild]);
    i=maxchild;
  }
}
```

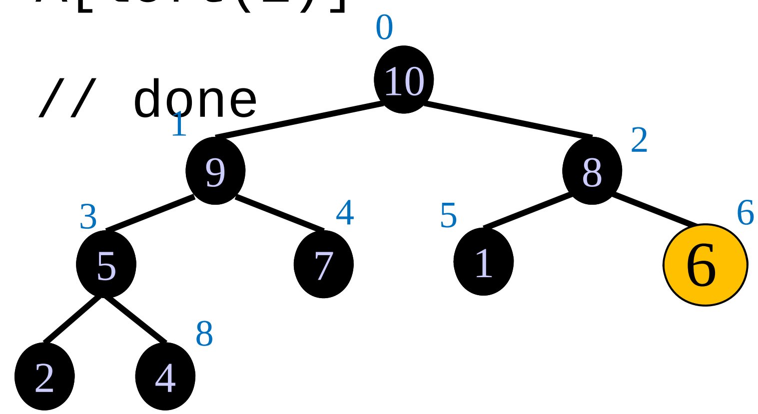


Extract_max, Heapify_down: O(log *n*)

```
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]);
    Heapify_down(A,n-1,0);
  }
}
```

O(n log n)

O(n log n)