Time Complexity

$$f(n) = O(g(n)) \Leftrightarrow n \geq n_0, \quad c > 0, \quad 0 \leq f(n) \leq cg(n)$$
$$f(n) = \Omega(g(n)) \Leftrightarrow n \geq n_0, \quad c > 0, \quad 0 \leq cg(n) \leq f(n)$$
$$f(n) = \Theta(g(n)) \Leftrightarrow n \geq n_0, \quad c_1, c_2 > 0, \quad 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Transitivity: $f = O(g(n))\ and\ \ g = O(h(n)) \Rightarrow f = O(h(n))$
Symmetry: $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
Reflexivity: $f = O(f), \quad f = \Theta(f), \quad f = \Omega(f)$

<u>Master Theorem</u>
A powerful theorem allowing to solve a large class of recursion relations of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1,\ b > 1$. There are 3 cases to remember:

1. If $f(n) = O(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$

2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$

3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and $af(n/b) \leq cf(n)$ for some c<1 then for all sufficiently large n $T(n) = \Theta(f(n))$

HeapSort

פעולות על מבני נתונים	פעולה יציבה	זמן ריצה	
		ממוצע	רע
BuildHeap	No		$O(n)$
Heapify	No		$O(\log n)$
HeapSort	No		$O(n \log n)$
Insert(S,x)	No		$O(\log n)$
Maximum(S)	No		$O(1)$
ExtractMax(S)	No		$O(\log n)$

Build-Heap(A)

```
heap-size[A] _ length[A]
for i _ length[A]/2 down to 1
do heapify (A,i)
```

Build-Heap(A,i)

```
if i > heapSize
return
Build-Heap(A,2i)
Build-Heap(A,2i+1)
Heapify (A,i)
```

HeapSort(A)

```
Build-Heap(A)
        for i_length[A] downto 2
                do exchange A[1] _A[i]
                        heapSize - -
                        heapify(A,1)
```

Extract-Max(A)

```
A[1] _ A[heapSize]
heapSize--
heapify(A,1)
```

Insert (A, x)

```
heapsize ++
A[heapSize] _x
fixup(A, heapSize)
```

Fixup (A, i)

```
while(A[parent(i)] != null and A[parent(i)] < A[i])
        swap (A[parent(i)],A[i])
        i = parent(i)
```

Linear Sorting

פעולות על מבני נתונים	פעולה יציבה	זמן ריצה	
		ממוצע	רע
Counting Sort		$O(k+n)$	$O(n^2)$
Radix Sort		$O(dn)$	$O(n^2)$
Bucket Sort		$O(n)$	$O(n^2)$
Order Statistic		$O(n)$	$O(n)$
Partition			$O(n)$

CountingSort(A, B, k)

```
for j= 1 to k
        do C[j] = 0
for j=1 to length(A)
        do C[A[j]]=C[A[j]]+1
for j=2 to k
        do C[j]=C[j]+C[j-1]
for j=length(A) downto 1
        do B[C[A[j]]]=A[j]
        C[A[j]]=C[A[j]]-1
```

RadixSort(A, d)

```
for j=1 to d
        do sort A according to the j-th least significant digit
```

BucketSort(A)

```
x = max/n
for j=1 to n
do bucket= [A[j] div x] +1
                e = new(A[j], null)
         InsertOrdered(B, bucket, e)
```

InsertOrdered(B, b, e)

```
if   B[b]=null or key(B[b]) > key(e)
        then next(e)=B[b]
                B[b] =e
        else  r =B[b]
    while (next(r)<>null and key(next(r))<=key(e))
        r=next(r )
    next(e) = next(r )
    next(r )= e
```

Select(A,p,r,i)

```
if  (p=r) then return A[p]
q=Partition(A,p,r)
k=q-p+1
if i<=k
        then return Select(A,p,q,i)
        else return Select(A,q+1,r,i-k)
```

Binary Search Trees

פעולות על מבני נתונים	פעולה יציבה	זמן ריצה	
		ממוצע	רע
Inorder Tree Walk			
Search		$O(\log n)$	$O(n)$
Insert		$O(\log n)$	$O(n)$
Delete		$O(\log n)$	$O(n)$
Minimum		$O(\log n)$	$O(n)$
Maximum		$O(\log n)$	$O(n)$
Successor		$O(\log n)$	$O(n)$
Predecessor		$O(\log n)$	$O(n)$