We say "f(n) is Ω(g(n))", or "f(n) is big-Omega of g(n)", if there exists a real constant c > 0 and an integer constant n0 >= 1 such that f(n) >= c.g(n) for every integer n >= n0.

**Example 1.9:** $3\log n + \log\log n$ *is* $\Omega(\log n)$.

***Proof:*** $3\log n + \log\log n \geq 3\log n$, *for* $n \geq 2$.

We say "f(n) is Θ(g(n))", or "f(n) is big-Theta of g(n)",
if there exists real constants c1, c2 > 0 and an integer constant n0 >= 1
such that f(n) <= c1.g(n) and f(n) >= c2.g(n) for every integer n >= n0.

**Example 1.10:** $3\log n + \log\log n$ *is* $\Theta(\log n)$.

NB - 1) If f(n) is O(g(n)), then g(n) must be Ω(f(n))
2) If f(n) is Ω(g(n)), then g(n) must be O(f(n))
3) If f(n) is Θ(g(n)), then f(n) must be both O(g(n)) and Ω(g(n))
4) If f(n) is Θ(g(n)), then also g(n) must be Θ(f(n))

---

**Theorem 1.7:** *Let $d(n)$, $e(n)$, $f(n)$, and $g(n)$ be functions mapping nonnegative integers to nonnegative reals.*

1. *If $d(n)$ is $O(f(n))$, then $ad(n)$ is $O(f(n))$, for any constant $a > 0$.*
2. *If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)+e(n)$ is $O(f(n)+g(n))$.*
3. *If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, then $d(n)e(n)$ is $O(f(n)g(n))$.*
4. *If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $d(n)$ is $O(g(n))$.*
5. *If $f(n)$ is a polynomial of degree $d$ (that is, $f(n) = a_0 + a_1 n + \cdots + a_d n^d$), then $f(n)$ is $O(n^d)$.*
6. *$n^x$ is $O(a^n)$ for any fixed $x > 0$ and $a > 1$.*
7. *$\log n^x$ is $O(\log n)$ for any fixed $x > 0$.*
8. *$\log^x n$ is $O(n^y)$ for any fixed constants $x > 0$ and $y > 0$.*

---

d(n) = 2n^2  is O(n^2)
e(n) = 4n^3  is O(n^3)
d(n) + e(n) = 2n^2 + 4n^3 is O(n^2 + n^3) = O(n^3)

f(n) = 10^100 . n
g(n) = 0.01 . n^2

efficient => polynomial running time => O(n^k) for some constant k > 0
not efficient => exponential running time => O(2^n)

f(n) = 3 . n^1001
g(n) = 2 . 2^n

It is considered poor taste to include constant factors and lower order terms in the big-Oh notation. For example, it is not fashionable to say that the function $2n^2$ is $O(4n^2 + 6n \log n)$, although this is completely correct. We should strive instead to describe the function in the big-Oh in ***simplest terms***.

| **logarithmic** | **linear** | **quadratic** | **polynomial** | **exponential** |
|---|---|---|---|---|
| $O(\log n)$ | $O(n)$ | $O(n^2)$ | $O(n^k)$ $(k \geq 1)$ | $O(a^n)$ $(a > 1)$ |

A few words of caution about asymptotic notation are in order at this point. First, note that the use of the big-Oh and related notations can be somewhat misleading should the constant factors they "hide" be very large. For example, while it is true that the function $10^{100}n$ is $\Theta(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n \log n$, we should prefer the $\Theta(n \log n)$-time algorithm, even though the linear-time algorithm is asymptotically faster. This preference is because the constant factor, $10^{100}$, which is called "one googol," is believed by many astronomers to be an upper bound on the number of atoms in the observable universe. So we are unlikely to ever have a real-world problem that has this number as its input size. Thus, even when using the big-Oh notation, we should at least be somewhat mindful of the constant factors and lower order terms we are "hiding."

| Some Functions Ordered by Growth Rate | Common Name |
|---|---|
| $\log n$ | logarithmic |
| $\log^2 n$ | polylogarithmic |
| $\sqrt{n}$ | square root |
| $n$ | linear |
| $n \log n$ | linearithmic |
| $n^2$ | quadratic |
| $n^3$ | cubic |
| $2^n$ | exponential |

| $n$ | $\log n$ | $\log^2 n$ | $\sqrt{n}$ | $n \log n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 4 | 2 | 8 | 16 | 64 | 16 |
| 16 | 4 | 16 | 4 | 64 | 256 | 4,096 | 65,536 |
| 64 | 6 | 36 | 8 | 384 | 4,096 | 262,144 | $1.84 \times 10^{19}$ |
| 256 | 8 | 64 | 16 | 2,048 | 65,536 | 16,777,216 | $1.15 \times 10^{77}$ |
| 1,024 | 10 | 100 | 32 | 10,240 | 1,048,576 | $1.07 \times 10^9$ | $1.79 \times 10^{308}$ |
| 4,096 | 12 | 144 | 64 | 49,152 | 16,777,216 | $6.87 \times 10^{10}$ | $10^{1233}$ |
| 16,384 | 14 | 196 | 128 | 229,376 | 268,435,456 | $4.4 \times 10^{12}$ | $10^{4932}$ |
| 65,536 | 16 | 256 | 256 | 1,048,576 | $4.29 \times 10^9$ | $2.81 \times 10^{14}$ | $10^{19728}$ |
| 262,144 | 18 | 324 | 512 | 4,718,592 | $6.87 \times 10^{10}$ | $1.8 \times 10^{16}$ | $10^{78913}$ |

We say that "f(n) is o(g(n))", or "f(n) is little-oh of g(n)", if for any constant c > 0, there exists a constant n0 > 0 such that f(n) <= c.g(n) for n >= n0.

We say that "f(n) is ω(g(n))", or "f(n) is little-omega of g(n)", if g(n) is o(f(n)).

**Example 1.11:** *The function $f(n) = 12n^2 + 6n$ is $o(n^3)$ and $\omega(n)$.*

***Proof:*** *Let us first show that $f(n)$ is $o(n^3)$. Let $c > 0$ be any constant. If we take $n_0 = (12+6)/c = 18/c$, then $18 \le cn$, for $n \ge n_0$. Thus, if $n \ge n_0$,*

$$f(n) = 12n^2 + 6n \le 12n^2 + 6n^2 = 18n^2 \le cn^3.$$

*Thus, $f(n)$ is $o(n^3)$.*

*To show that $f(n)$ is $\omega(n)$, let $c > 0$ again be any constant. If we take $n_0 = c/12$, then, for $n \ge n_0$, $12n \ge c$. Thus, if $n \ge n_0$,*

$$f(n) = 12n^2 + 6n \ge 12n^2 \ge cn.$$

*Thus, $f(n)$ is $\omega(n)$.* ■

For the reader familiar with limits, we note that $f(n)$ is $o(g(n))$ if and only if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0,$$

provided this limit exists. The main difference between the little-oh and big-Oh notions is that $f(n)$ is $O(g(n))$ if ***there exist*** constants $c > 0$ and $n_0 \ge 1$ such that $f(n) \le cg(n)$, for $n \ge n_0$; whereas $f(n)$ is $o(g(n))$ if ***for all*** constants $c > 0$ there is a constant $n_0$ such that $f(n) \le cg(n)$, for $n \ge n_0$. Intuitively, $f(n)$ is $o(g(n))$ if $f(n)$ becomes insignificant compared to $g(n)$ as $n$ grows toward infinity. As previously mentioned, asymptotic notation is useful because it allows us to concentrate on the main factor determining a function's growth.

Example of 'polylogarithmic' function : 11(log n)^4 + 5(log n)^2 + 3(log n) + 2

**Algorithm** recursiveMax$(A, n)$:

*Input:* An array $A$ storing $n \geq 1$ integers.

*Output:* The maximum element in $A$.

**if** $n = 1$ **then**

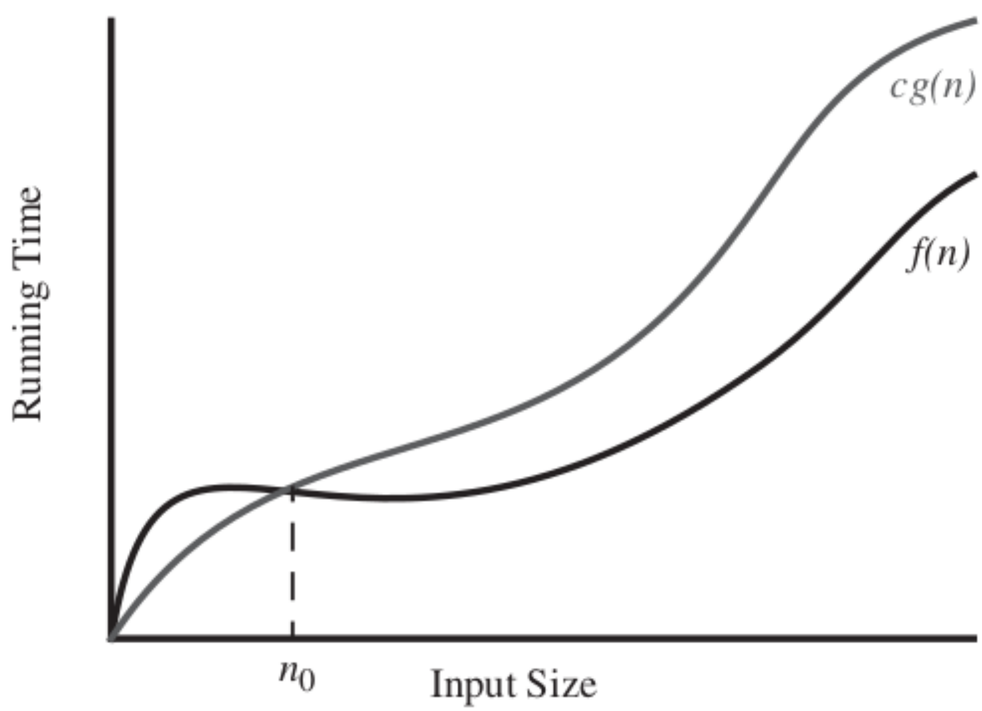
  **return** $A[0]$

**return** $\max\{\text{recursiveMax}(A, n-1),\ A[n-1]\}$

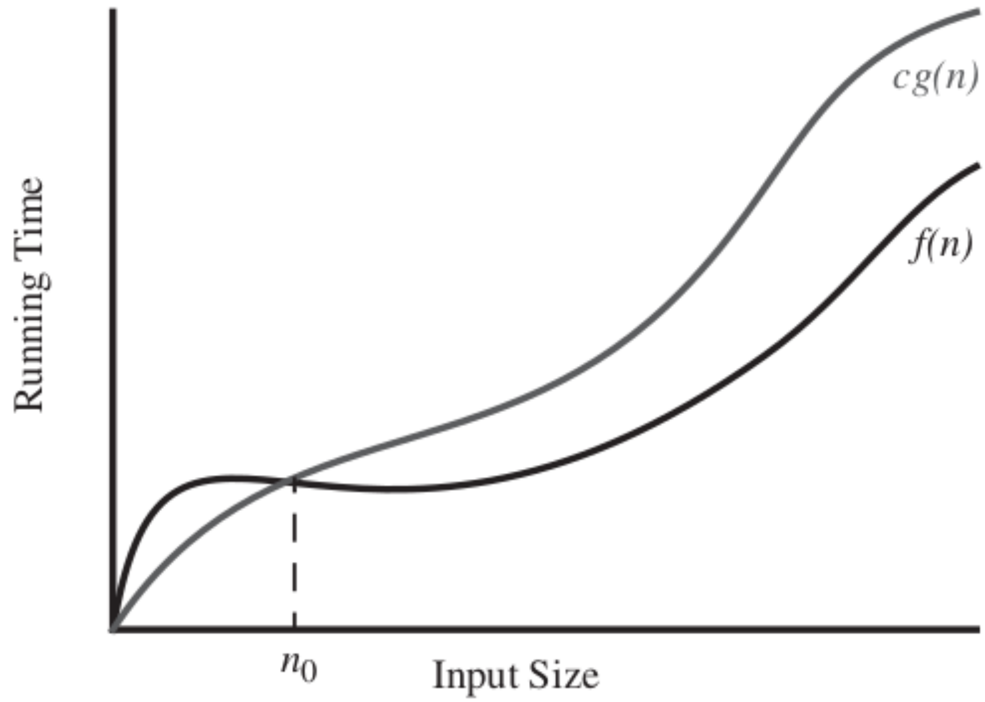
**Algorithm 1.4:** Algorithm recursiveMax.

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

T(n) = 7(n-1) + 3 = 7n - 4




The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.




The function $f(n)$ is $O(g(n))$, for $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

Let f(n) and g(n) be functions mapping non-negative integers to real numbers.

We say "f(n) is O(g(n))", or "f(n) is order of g(n)", if there exists a real constant c > 0 and an integer constant n0 >= 1 such that f(n) <= c.g(n) for every integer n >= n0.

Example 1.1: f(n) = 7n - 2 is O(n).

Proof: We need a real constant c > 0 and an integer constant n0 >= 1 such that (7n-2) <= c.n for every integer n >= n0. One possible choice is c = 7, and n0 = 1.

Corollary: The running time of arrayMax is O(n).

**Example 1.3:** $20n^3 + 10n \log n + 5$ *is* $O(n^3)$.

***Proof:*** $20n^3 + 10n \log n + 5 \leq 35n^3$, *for* $n \geq 1$.

Note - If f(n) is a polynomial function of degree k, then f(n) will always be O(n^k).

Example 1.4: f(n) = 3.log(n) + log(log(n)) is O(log n).
Proof: We can choose c=4 and n0 = 2.

**Example 1.5:** $2^{100}$ *is* $O(1)$.

***Proof:*** $2^{100} \leq 2^{100} \cdot 1$, *for* $n \geq 1$. *Note that variable* $n$ *does not appear in the inequality, since we are dealing with constant-valued functions.* ■

We say "f(n) is Ω(g(n))", or "f(n) is big-Omega of g(n)", if there exists a real constant c > 0 and an integer constant n0 >= 1 such that f(n) >= c.g(n) for every integer n >= n0.

**Example 1.9:** $3\log n + \log\log n$ *is* $\Omega(\log n)$.

***Proof:*** $3\log n + \log\log n \geq 3\log n$, *for* $n \geq 2$.

We say "f(n) is Θ(g(n))", or "f(n) is big-Theta of g(n)",
if there exists real constants c1, c2 > 0 and an integer constant n0 >= 1
such that f(n) <= c1.g(n) and f(n) >= c2.g(n) for every integer n >= n0.

**Example 1.10:** $3\log n + \log\log n$ *is* $\Theta(\log n)$.

An algorithm is a step-by-step procedure for performing some task in a finite amount of time.

Experimental studies on running times are useful, but they have some limitations:

- Experiments can be done only on a limited set of test inputs, and care must be taken to make sure these are representative.
- It is difficult to compare the efficiency of two algorithms unless experiments on their running times have been performed in the same hardware and software environments.
- It is necessary to implement and execute an algorithm in order to study its running time experimentally.

Thus, while experimentation has an important role to play in algorithm analysis, it alone is not sufficient. Therefore, in addition to experimentation, we desire an analytic framework that

- Takes into account all possible inputs
- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment
- Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it.

"Algorithm A runs in time proportional to n" =>
If we were to perform experiments, then we would find that the actual running time of algorithm A on any input of size n never exceeds c.n, where c is a constant that depends on the hardware and software environment used.

Given two algorithms A and B, where A runs in time proportional to n and B runs in time proportional to n^2 , we will prefer A to B, since the function n grows at a smaller rate than the function n^2 .

"Algorithm A runs in time proportional to n" =>
If we were to perform experiments, then we would find that the actual running time of algorithm A on any input of size n never exceeds c.n, where c is a constant that depends on the hardware and software environment used.

- A language for describing algorithms
- A computational model that algorithms execute within
- A metric for measuring algorithm running time
- An approach for characterizing running times, including those for recursive algorithms.

**Algorithm** arrayMax$(A, n)$:
  ***Input:*** An array $A$ storing $n \geq 1$ integers.
  ***Output:*** The maximum element in $A$.

```
currentMax ← A[0]
for i ← 1 to n − 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

By inspecting the pseudocode, we can argue about the correctness of algorithm arrayMax with a simple argument. Variable *currentMax* starts out being equal to the first element of $A$. We claim that at the beginning of the $i$th iteration of the loop, *currentMax* is equal to the maximum of the first $i$ elements in $A$. Since we compare *currentMax* to $A[i]$ in iteration $i$, if this claim is true before this iteration, it will be true after it for $i + 1$ (which is the next value of counter $i$). Thus, after $n - 1$ iterations, *currentMax* will equal the maximum element in $A$. As with this example, we want our pseudocode descriptions to always be detailed enough to fully justify the correctness of the algorithm they describe, while being simple enough for human readers to understand.

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method.

Specifically, a primitive operation corresponds to a low-level instruction with an execution time that depends on the hardware and software environment but is nevertheless constant. Instead of trying to determine the specific execution time of each primitive operation, we will simply ***count*** how many primitive operations are executed, and use this number $t$ as a high-level estimate of the running time of the algorithm. This operation count will correlate to an actual running time in a specific hardware and software environment, for each primitive operation corresponds to a constant-time instruction, and there are only a fixed number of primitive operations. The implicit assumption in this approach is that the running times of different primitive operations will be fairly similar. Thus, the number, $t$, of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

RAM (Random Access Machine) Model -
A computer is viewed simply as a CPU connected to a bank of memory cells.
Each memory cell stores a word, which can be a number, a string, or an address.
The term "random access" refers to the ability of the CPU to access an arbitrary memory location using just one single primitive operation.
We assume the CPU in the RAM model can perform any primitive operation in a constant number of steps, which do not depend on the size of the input.

**Algorithm** arrayMax$(A, n)$:

  ***Input:*** An array $A$ storing $n \geq 1$ integers.

  ***Output:*** The maximum element in $A$.

```
currentMax ← A[0]
for i ← 1 to n − 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

- Initializing the variable *currentMax* to $A[0]$ corresponds to two primitive operations (indexing into an array and assigning a value to a variable) and is executed only once at the beginning of the algorithm. Thus, it contributes two units to the count.
- At the beginning of the for loop, counter $i$ is initialized to 1. This action corresponds to executing one primitive operation (assigning a value to a variable).
- Before entering the body of the for loop, condition $i < n$ is verified. This action corresponds to executing one primitive instruction (comparing two numbers). Since counter $i$ starts at 1 and is incremented by 1 at the end of each iteration of the loop, the comparison $i < n$ is performed $n$ times. Thus, it contributes $n$ units to the count.
- The body of the for loop is executed $n - 1$ times (for values $1, 2, \ldots, n - 1$ of the counter). At each iteration, $A[i]$ is compared with *currentMax* (two primitive operations, indexing and comparing), $A[i]$ is possibly assigned to *currentMax* (two primitive operations, indexing and assigning), and the counter $i$ is incremented (two primitive operations, summing and assigning). Hence, at each iteration of the loop, either four or six primitive operations are performed, depending on whether $A[i] \leq$ *currentMax* or $A[i] >$ *currentMax*. Therefore, the body of the loop contributes between $4(n - 1)$ and $6(n - 1)$ units to the count.
- Returning the value of variable *currentMax* corresponds to one primitive operation, and is executed only once.

**Algorithm** arrayMax$(A, n)$:

***Input:*** An array $A$ storing $n \geq 1$ integers.

***Output:*** The maximum element in $A$.

*currentMax* $\leftarrow A[0]$

**for** $i \leftarrow 1$ **to** $n - 1$ **do**

  **if** *currentMax* $< A[i]$ **then**

    *currentMax* $\leftarrow A[i]$

**return** *currentMax*

To summarize, the number of primitive operations $t(n)$ executed by algorithm arrayMax is at least

$$2 + 1 + n + 4(n - 1) + 1 = 5n$$

and at most

$$2 + 1 + n + 6(n - 1) + 1 = 7n - 2.$$

The best case ($t(n) = 5n$) occurs when $A[0]$ is the maximum element, so that variable *currentMax* is never reassigned. The worst case ($t(n) = 7n - 2$) occurs when the elements are sorted in increasing order, so that variable *currentMax* is reassigned at each iteration of the for loop.

We will, for the remainder of this course, typically characterize running times in terms of the worst case. We say, for example, that algorithm arrayMax executes t(n) = 7n − 2 primitive operations in the worst case, meaning that the maximum number of primitive operations executed by the algorithm, taken over all inputs of size n, is 7n − 2.

This type of analysis is much easier than an average-case analysis, as it does not require probability theory; it just requires the ability to identify the worst-case input, which is often straightforward. In addition, taking a worst-case approach can actually lead to better algorithms. Making the standard of success that of having an algorithm perform well in the worst case necessarily requires that it perform well on every input.

A = {1, 3, 2, 5}

recursiveMax(A, 4)
= max( recursiveMax(A, 3), A[3] )
= max( max( recursiveMax(A, 2), A[2] ), A[3] )
= max( max( max( recursiveMax(A, 1), A[1]), A[2] ), A[3] )
= max( max( max( A[0], A[1]), A[2] ), A[3] )
= max( max( max( 1, 3), 2 ), 5 )
= max( max( 3, 2), 5 )
= max( 3, 5 )
= 5

max(a, b)

if a > b
   return a
else
   return b

**Algorithm** recursiveMax$(A, n)$:

*Input:* An array $A$ storing $n \geq 1$ integers.

*Output:* The maximum element in $A$.

**if** $n = 1$ **then** 1

  **return** $A[0]$ 2

**return** max{recursiveMax$(A, n-1)$, $A[n-1]$} T(n-1) + 6

**Algorithm 1.4:** Algorithm recursiveMax.

return: 1, max: 2, operation (n-1): 2, recursive call: T(n-1), indexing A[n-1]: 1

T(n) = no. of primitive operations required to compute recursiveMax(A, n)

$$T(n) = \begin{cases} 3 & \text{if } n = 1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

T(n) = T(n-1) + 7
= T(n-2) + 7 + 7
= ....
= T(1) + 7(n-1)
= 3 + 7(n-1)
= 7n - 4

$f(n) = 1000\ n$

$g(n) = 2\ n^2$

Find smallest n where:
$g(n) >= f(n)$

$2.n^2 >= 1000\ n$
$n >= 500$

| n | f(n) | g(n) | f(n) / f(n-1) | g(n) / g(n-1) |
|---|---|---|---|---|
| 1 | 1000 | 2 | - | - |
| 2 | 2000 | 8 | 2 | 4 |
| 3 | 3000 | 18 | 1.5 | 2.25 |
| 4 | 4000 | 32 | 1.33 | 1.78 |
| 5 | 5000 | 50 | 1.2 | 1.5625 |

.............................

Scalability

1) f(n) is O(g(n)) => g(n) is Ω(f(n))

2) f(n) is Ω(g(n)) => g(n) is O(f(n))

3) f(n) is Θ(g(n)) <=> f(n) is O(g(n)) AND f(n) is Ω(g(n))

To prove: f(n) is Θ(g(n)) <=> g(n) is Θ(f(n))

f(n) is Θ(g(n)) => f(n) is O(g(n)) AND f(n) is Ω(g(n)) [apply rule 3]
=> g(n) is Ω(f(n)) AND g(n) is O(f(n)) [apply rules 1 and 2]
=> g(n) is Θ(f(n)) [apply rule 3]

---

$f(n) = 2.n^2 + 4.n^3$
$= d(n) + e(n)$
$= O(n^2 + n^3)$
$= O(g(n))$

$g(n) = O(n^3)$

Thus, $f(n) = O(n^3)$

$d(n) = 2.n^2$ is $O(n^2)$

$e(n) = 4.n^3$ is $O(n^3)$

$g(n) = n^2 + n^3$
Now, g(n) is a polynomial of degree 3,
g(n) is $O(n^3)$.

---

$f(n) = 3.\log(n) + \log(\log(n))$
$= d(n) + e(n)$
$= O(\log(n) + \log(n))$
$= O(\ 2.\log(n)\ )$
$= O(\ b(n)\ )$

$b(n) = 2.\log(n)$ is $O(\log(n))$

Thus, f(n) is $O(\log(n))$

$d(n) = 3.\log(n)$ is $O(\log(n))$

$e(n) = \log(\log(n))$ is $O(\log(n))$

So, e(n) is $O(\log(n))$

---

$2n^5$ is $O(n^5)$

$n^5 + 3$ is $O(n^5)$

**Algorithm** merge$(S_1, S_2, S)$:

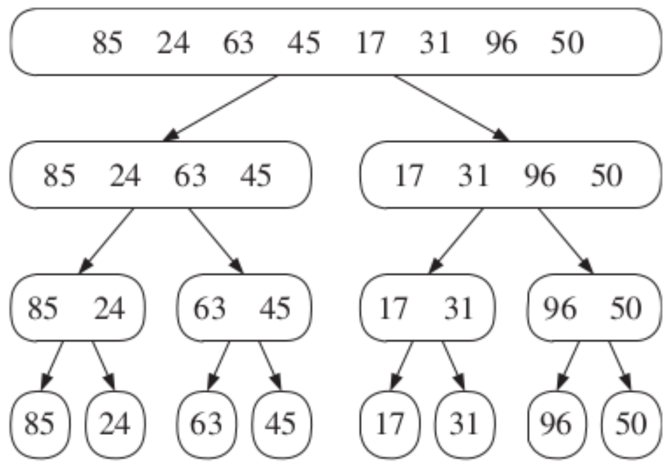
***Input:*** Two arrays, $S_1$ and $S_2$, of size $n_1$ and $n_2$, respectively, sorted in non-decreasing order, and an empty array, $S$, of size at least $n_1 + n_2$

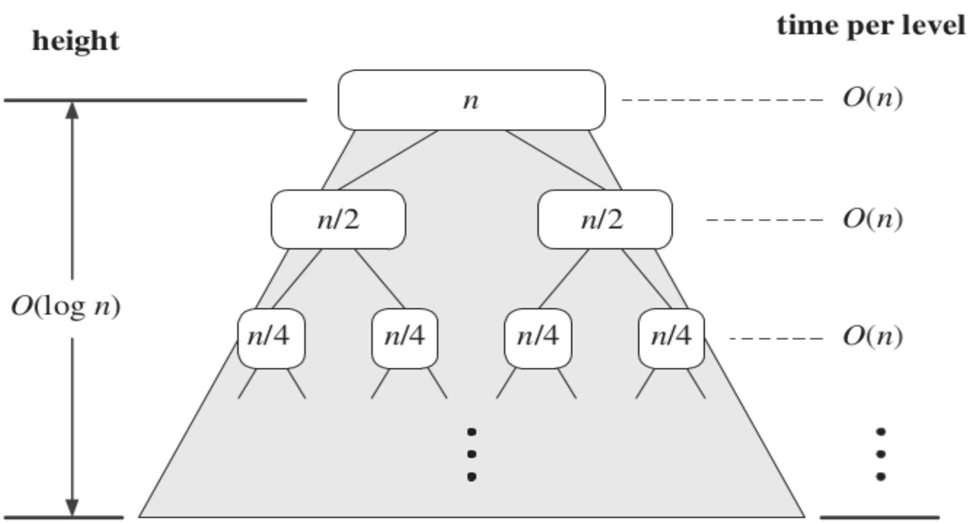
***Output:*** $S$, containing the elements from $S_1$ and $S_2$ in sorted order

$i \leftarrow 1$
$j \leftarrow 1$
**while** $i \leq n$ **and** $j \leq n$ **do**
  **if** $S_1[i] \leq S_2[j]$ **then**
    $S[i+j-1] \leftarrow S_1[i]$
    $i \leftarrow i+1$
  **else**
    $S[i+j-1] \leftarrow S_2[j]$
    $j \leftarrow j+1$
**while** $i \leq n$ **do**
  $S[i+j-1] \leftarrow S_1[i]$
  $i \leftarrow i+1$
**while** $j \leq n$ **do**
  $S[i+j-1] \leftarrow S_2[j]$
  $j \leftarrow j+1$

```
i <- 0
j <- 0
while i < n1 and j < n2, do:
    if S1[i] <= S2[j], then:
        S[i+j] <- S1[i]
        i <- i + 1
    else:
        S[i+j] <- S2[j]
        j <- j + 1
while i < n1, do:
    S[i+j] <- S1[i]
    i <- i + 1
while j < n2, do:
    S[i+j] <- S2[j]
    j <- j + 1
```

```
Algorithm MergeSort(S):
    Input: An array S, of size n
    Output: The sorted version of the array S

    if (n==1):
        return S
    if (n==2):
        if S[0] > S[1]:
            S[0] <-> S[1]    // swap S[0] and S[1]
        return S
    S1 <-  { S[0], S[1], .... , S[n/2] }
    S2 <-  { S[n/2+1], .... , S[n-2], S[n-1] }
    R1 <- MergeSort(S1)
    R2 <- MergeSort(S2)
    merge(R1, R2, R)    // R is an empty array of size n, to be filled in
    return R
```




**Total time:** $O(n \log n)$

n <= 2^k
k <= log_2(n) = O(log n)

Level 0: n
Level 1: n/2
Level 2: n/4
......
......
Level k: n/(2^k) <= 1

Input: { 1, 4, 7, 5, 2, 3 }

Output: { 1, 2, 3, 4, 5, 7 }

Inversions:
(4, 2)
(4, 3)
(7, 5)
(7, 2)
(7, 3)
(5, 2)
(5, 3)

Insertion Sort --

```
Initial :  1  4  7  5  2  3
Pass 1:    1  4 | 7  5  2  3     4>1
Pass 2:    1  4  7 | 5  2  3     7>4
Pass 3:    1  4  5  7 | 2  3     5<7,  5>4
Pass 4:    1  2  4  5  7 | 3     2<7, 2<5, 2<4, 2>1
Pass 5:    1  2  3  4  5  7      3<7, 3<5, 3<4, 3>2
```

Pass 4:  j=3,  j=2, j=1, j=0
Elements at indices (j+1) to (i-1) to be shifted

```
1 4 5 7 2 3
1 4 5 _ 7 3       tmp <- 2
1 4 _ 5 7 3       tmp <- 2
1 _ 4 5 7 3       tmp <- 2
1 2 4 5 7 3
```

InsertionSort( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

```
for i = 1 to (n-1)
{
     j <- i - 1
     while A[i] < A[j]
            j <- j - 1
            if j < 0
                 break
     tmp <- A[i]
     // shift all elements > A[i] by 1 position
     k = i - 1
     while k >= j+1
     {
            A[k+1] <- A[k]
            k <- k - 1
     }
     // insert A[i] in position (j+1)
     A[j+1] <- tmp
}
return A
```

{ 4, 1, ... }

Worst case complexity = O(n^2)
Best case complexity = O(n)

Input: { 7, 5, 4, 3, 2, 1 }

Output: { 1, 2, 3, 4, 5, 7 }

Insertion Sort --

```
Initial :  7  5  4  3  2  1
Pass 1:    5  7 | 4  3  2  1     5<7
Pass 2:    4  5  7 | 3  2  1     4<7, 4<5
Pass 3:    3  4  5  7 | 2  1     3<7, 3<5, 3<4
Pass 4:    2  3  4  5  7 | 1
Pass 5:    1  2  3  4  5  7
```

InsertionSortOptimized( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

```
for i = 1 to (n-1)
{
     inversions = 0
     for j = 0 to (n-1)
         if A[j] > A[j+1]
                inversions <- inversions + 1
      if inversions == 0:
          break

      j <- i - 1
     while A[i] < A[j]
            j <- j - 1
            if j < 0
                break
     tmp <- A[i]
     // shift all elements > A[i] by 1 position
     k = i - 1
     while k >= j+1
     {
            A[k+1] <- A[k]
            k <- k - 1
     }
     // insert A[i] in position (j+1)
     A[j+1] <- tmp
}
return A
```

Worst case complexity = O(n^2)
Best case complexity = O(n)

Input: { 1, 4, 7, 5, 2, 3 }

Output: { 1, 2, 3, 4, 5, 7 }

Inversions:
(4, 2)
(4, 3)
(7, 5)
(7, 2)
(7, 3)
(5, 2)
(5, 3)

Selection Sort --

Initial : 1 4 7 5 2 3

Pass 1: 1 4 3 5 2 7

Pass 2: 1 4 3 2 5 7

Pass 3: 1 2 3 4 5 7

Pass 4: 1 2 3 4 5 7

Pass 5: 1 2 3 4 5 7

```
SelectionSort( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

for i = 1 to (n-1)
{
     currentMax = A[0]
     maxIndex = 0
     for j = 1 to (n-i)
     {
            if A[j] > currentMax
            {
                  currentMax = A[j]
                  maxIndex = j
            }
     }
     // swap A[maxIndex] with A[n-i]
     tmp <- A[maxIndex]
     A[maxIndex] <- A[n-i]
     A[n-i] <- tmp
}
return A
```

Complexity (best and worst case):

$$c * [(n-1) + (n-2) + (n-3) + \ldots + 1]$$
$$= c * [ (n-1)*n/2 ]$$
$$= O(n^2)$$

Input: { 7, 5, 4, 3, 2, 1 }

Output: { 1, 2, 3, 4, 5, 7 }

Selection Sort --

Initial : 7 5 4 3 2 1

Pass 1: 1 5 4 3 2 7

Pass 2: 1 2 4 3 5 7

Pass 3: 1 2 3 4 5 7

Pass 4: 1 2 3 4 5 7

Pass 5: 1 2 3 4 5 7

```
SelectionSortOptimized( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

for i = 1 to (n-1)
{
     inversions = 0
     for j = 0 to (n-1-i)
         if A[j] > A[j+1]
                inversions <- inversions + 1
      if inversions == 0:
         break

     currentMax = A[0]
     maxIndex = 0
     for j = 1 to (n-i)
     {
            if A[j] > currentMax
            {
                  currentMax = A[j]
                  maxIndex = j
            }
     }
     // swap A[maxIndex] with A[n-i]
     tmp <- A[maxIndex]
     A[maxIndex] <- A[n-i]
     A[n-i] <- tmp
}
return A
```

Worst case complexity = $O(n^2)$
Best case complexity = $O(n)$

Input: { 1, 4, 7, 2, 5, 3 }

Output: { 1, 2, 3, 4, 5, 7 }

Bubble Sort --

```
Initial :   1  4  7  2  5  3

Pass 1:   1  4  2  5  3  7     (swaps = 3)

Pass 2:   1  2  4  3  5  7     (swaps = 2)

Pass 3:   1  2  3  4  5  7     (swaps = 1)

Pass 4:   1  2  3  4  5  7     (swaps = 0)
```

```
BubbleSort( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

for i = 1 to (n-1)
{
     for j = 0 to (n-2)
          if A[j] > A[j+1]
          {
                 // swap A[j] with A[j+1]
                 tmp <- A[j]
                 A[j] <- A[j+1]
                 A[j+1] <- tmp
           }
}
return A
```

Complexity (best and worst case):

(c*(n-1)) * (n-1)    = c*(n-1)^2
                     = O(n^2)

c = 1 + 3 + 2 + 3 + 2 + 2 = 13

Input: {cat, mat, bat, ant}

Output: {ant, bat, cat, mat}

Input: { 7, 5, 4, 3, 2, 1 }

Output: { 1, 2, 3, 4, 5, 7 }

Bubble Sort --

```
Initial :   7  5  4  3  2  1

Pass 1:   5  4  3  2  1  7     (swaps = 5)

Pass 2:   4  3  2  1  5  7     (swaps = 4)

Pass 3:   3  2  1  4  5  7     (swaps = 3)

Pass 4:   2  1  3  4  5  7     (swaps = 2)

Pass 5:   1  2  3  4  5  7     (swaps = 1)
```

```
BubbleSortOptimized( int[] A, int n )
  Input: An array A containing n >= 1 integers
  Output: The sorted version of the array A

for i = 1 to (n-1)
{
      swaps = 0
      for j = 0 to (n-1-i)
           if A[j] > A[j+1]
           {
                  // swap A[j] with A[j+1]
                  tmp <- A[j]
                  A[j] <- A[j+1]
                  A[j+1] <- tmp
                  swaps <- swaps + 1
            }
       if swaps == 0:
           break
}
return A
```

Worst case complexity:
    c * [(n-1) + (n-2) + (n-3) + ... + 1]
= c * [ (n-1)*n/2 ]
= O(n^2)

Best case complexity:
    c*(n-1)
= O(n)

| Value: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Frequency: | 0 | 1 | 1 | 2 | 0 | 0 | 1 | 2 | 1 | 1 |

A = { 6, 1, 8, 3, 7, 2, 3, 9, 7}

S = { 1, 2, 3, 3, 6, 7, 7, 8, 9 }

Time complexity:

O(R) + O(n) + O(n+R)
= O(2n + 2R)
= O(n + R)
= O(max(n, R))

F[i] = 0  ->  O(1)
F[i] > 0  ->  O(F[i])

R * O(1) + Sum_i { F[i] }  =  O(n+R)

R * O(1) + O(n)  =  O(n+R)

```
CountSort( int[] A, int n, int a, int b )
   // Input: An array of integers of length n,
             where the values are in [a, b]
   // Output: The sorted version of A

   R = b-a+1
   int F[R]
   for i = 0 to (R-1)
       F[i] = 0

   for i = 0 to (n-1)
       tmp = A[i] - a
       F[tmp] = F[tmp] + 1

   int S[n]
   k = 0
   for i = 0 to (R-1)
        freq = F[i]
        for j = 1 to freq
            S[k] = i + a
            k = k+1
   return S
```

For Binary Search, the time complexity is given by the following recurrence:

T(n) = O(1) + T(n/2)  =>  T(n) = O(log n)

For Merge Sort, the time complexity is given by the following recurrence:

T(n) = 2.T(n/2) + O(n)  =>  T(n) = O(n.log(n))

---

Solving Recurrence Relations --

1) Substitution method: Guess a solution, and then check whether it is correct.

Eg. - Let us guess the solution for Binary Search as T(n) = O(log n), which means that we must have T(n) <= c.(log n) for large enough n (for all n >= n0).

T(n) = O(1) + T(n/2)
<= O(1) + O(log(n/2))
<= c1 + c2.(log(n/2))
= c1 + c2.(log(n) - log(2))
= c1 + c2.log(n) - c2
= c2.log(n) - (c2 - c1)
<= c2.log(n)
= O(log n)

2) Recurrence Tree method: Figure out the solution by studying the recurrence tree.

3) Master's Theorem method:

It is a direct method to get solutions for recurrences of the form T(n) = a.T(n/b) + O(n^c), where a>=1 and b>1. Then, the following three cases are used to obtain the solution directly -

(i) If c < log_b(a), then T(n) = Θ( n^(log_b(a)) )
Eg. - For the recurrence T(n) = 16.T(n/4) + O(n), we have: a=16, b=4, c=1
Therefore, 1 = c < log_b(a) = log4(16) = 2, and so we have: T(n) = Θ( n^2 )

(ii) If c = log_b(a), then T(n) = Θ( n^c. log(n) )
Eg. - For binary search recurrence T(n) = 1.T(n/2) + O(1), we have: a=1, b=2, c=0
Therefore, c = log_b(a) = log2(1) = 0, and so we have: T(n) = Θ( n^0. log(n) )
Eg. - For merge sort recurrence T(n) = 2.T(n/2) + O(n), we have: a=2, b=2, c=1
Therefore, c = log_b(a) = log2(2) = 1, and so we have: T(n) = Θ( n^1. log(n) )

(iii) If c > log_b(a), then T(n) = Θ( n^c )
Eg. - For the recurrence T(n) = 2.T(n/4) + O(n^2), we have: a=
Therefore, 2 = c > log_b(a) = log4(2) = 0.5, and so we have: T(n) = Θ( n^2 )

---

T(n) = 2.T(n/2) + O(n)
= 2.(2.T(n/4) + O(n/2)) + O(n)
= 2.(2.(2.T(n/8) + O(n/4)) + O(n/2)) + O(n)
.......
.......
= 2^k.T(n/(2^k))) + (O(n) + 2.O(n/2) + 4.O(n/4) + ... + (2^k).O(n/(2^k)))
= n.T(1) + ( O(n) + O(n) + ..(k times).. + O(n) )
= n + k.O(n)
= n + O(n).log(n) = O(n. log(n))

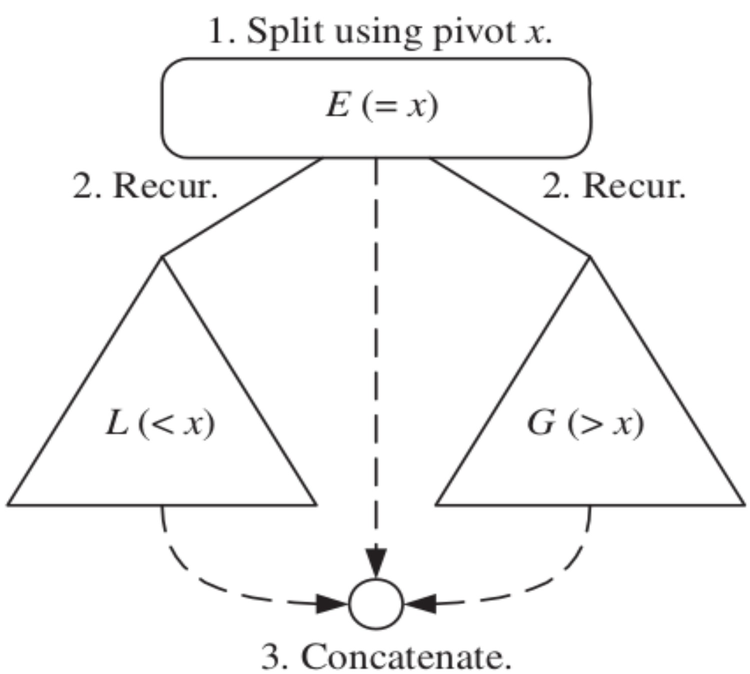
QUICK SORT - Steps:

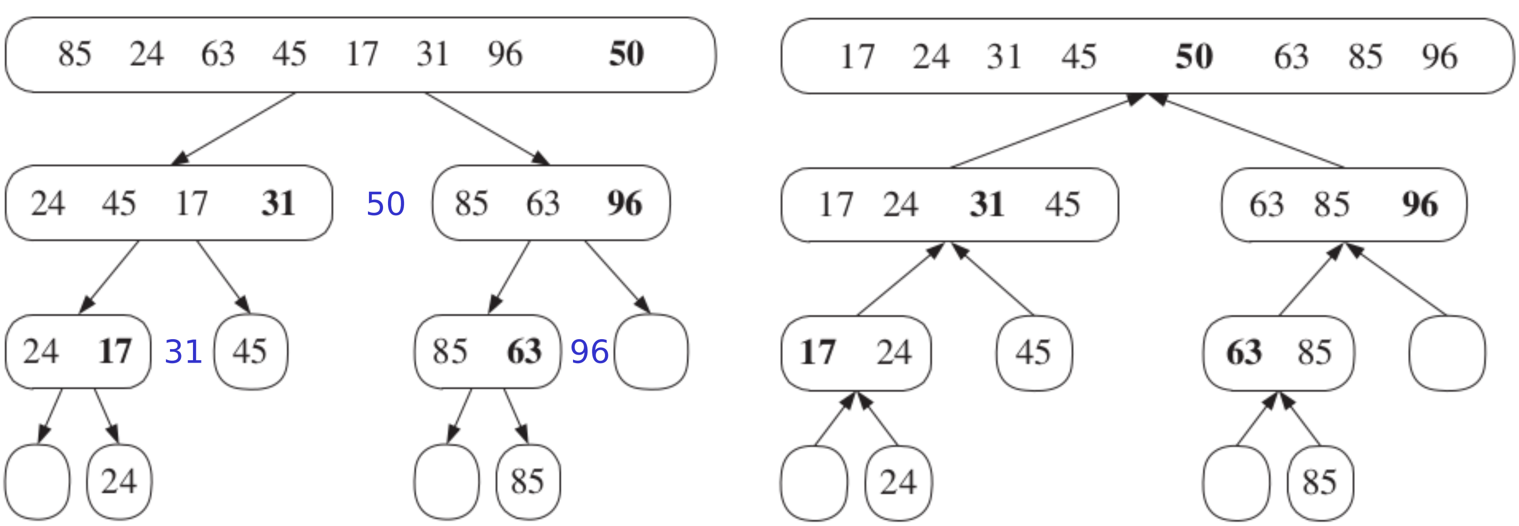
1. ***Divide:*** If $S$ has at least two elements (nothing needs to be done if $S$ has zero or one element), select a specific element $x$ from $S$, which is called the ***pivot***. As is common practice, choose the pivot $x$ to be the last element in $S$. Remove all the elements from $S$ and put them into three sequences:

    - $L$, storing the elements in $S$ less than $x$
    - $E$, storing the elements in $S$ equal to $x$
    - $G$, storing the elements in $S$ greater than $x$.

    (If the elements of $S$ are all distinct, $E$ holds just one element—the pivot.)

2. ***Recur:*** Recursively sort sequences $L$ and $G$.
3. ***Conquer:*** Put the elements back into $S$ in order by first inserting the elements of $L$, then those of $E$, and finally those of $G$.



{85 24 63 45 50 31 96 17}

L={}   E={17}  G={85 24 63 45 50 31 96}

{85 24 63 45 17 31 50 96}

L={85 24 63 45 31 50}   E={96}  G={}

---

```
partition (arr[], low, high)
{
    pivot = arr[high];
    i = (low - 1)
    for (j = low; j < high; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i+1] and arr[high])
    return (i + 1)
}
```

Initial:   85 24 63 45 17 31 96 50
           low = 0,  high = 7

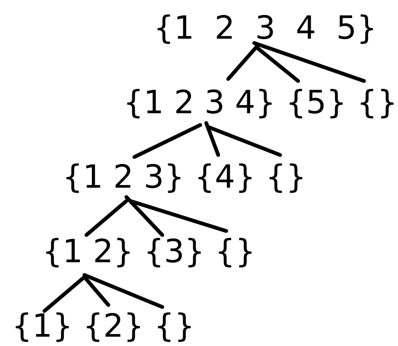
Pivot: arr[7] = 50

i = 3
j = 6

```
85  24  63  45  17  31  96  50
24  85  63  45  17  31  96  50
24  45  63  85  17  31  96  50
24  45  17  85  63  31  96  50
24  45  17  31  63  85  96  50
24  45  17  31  50  85  96  63
```

24  45  17  31  |  50  |  85  96  63

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)    // i.e. if arr[] is of length >= 2
    {
        pi = partition(arr, low, high);
        /* pi is partitioning index, i.e. index of pivot */

        quickSort(arr, low, pi - 1);    // Before pi, i.e. L
        quickSort(arr, pi + 1, high); // After pi, i.e. G
    }
}
```




```
partition (arr[], low, high)
{
    pivot = arr[high];
    i = low
    for (j = low; j < high; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            swap arr[i] and arr[j]
            i++;    // increment index of smaller element
        }
    }
    swap arr[i] and arr[high])
    return (i)
}
```

Worst case time compexity:

(n-1) levels, and
O(n) operations at each level
(counted over all partition functions)
= O(n^2)

Best case time complexity:

O(log(n)) levels, and
O(n) operations at each level
(counted over all partition functions)
= O(n log(n))

Average case time complexity:
O(n log(n))

```
LinearSearch(int[] A, int n, int key)
    pos = -1
    for i = 0 to (n-1)
          if (A[i] == key)
          {
                pos = i
                break
           }
    if pos == -1
           print "Not found!"
    else
           print "Found!"
    return pos
```

A = { 1, 7, 4, 3, 9, 6, 2 }

key = 5 ---> Not found - failed!

key = 4 ---> Found, at index 2!

Worst case time complexity: O(n)

Best case time complexity: O(1)

---

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 4

Comparison 1: key < 6

A' = { 1, 3, 4 }
key = 4

Comparison 2: key > 3

A'' = { 4 }
key = 4

Comparison 3: key == 4

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 8

Comparison 1: key > 6

A' = { 8, 9, 11 }
key = 8

Comparison 2: key < 9

A'' = { 8 }
key = 8

Comparison 3: key == 8

**Algorithm** BinarySearch$(A, k, \text{low}, \text{high})$:

*Input:* An ordered array, $A$, storing $n$ items, whose keys are accessed with method key$(i)$ and whose elements are accessed with method elem$(i)$; a search key $k$; and integers low and high

*Output:* An element of $A$ with key $k$ and index between low and high, if such an element exists, and otherwise the special element ***null***

**if** low $>$ high **then**
  **return** ***null***
**else**
  mid $\leftarrow \lfloor(\text{low} + \text{high})/2\rfloor$
  **if** $k = \text{key}(\text{mid})$ **then**
    **return** elem(mid)
  **else if** $k < \text{key}(\text{mid})$ **then**
    **return** BinarySearch$(A, k, \text{low}, \text{mid} - 1)$
  **else**
    **return** BinarySearch$(A, k, \text{mid} + 1, \text{high})$

Worst case time complexity: O(log n)

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 2

Comparison 1: key < 6

A' = { 1, 3, 4 }
key = 2

Comparison 2: key < 3

A'' = { 1 }
key = 2

Comparison 3: key > 1

A''' = {}
Return "Not found"

```
BinarySearch( A, key, low, high )

if (low > high)          // if A is an empty array
      return (-1)        // "Not found"

mid <-  ⌊ (low + high) / 2 ⌋

if key == A[mid]
      return mid

else if key < A[mid]
      BinarySearch( A, key, low, mid-1 )

else if key > A[mid]
      BinarySearch( A, key, mid+1, high )
```

low = 6
high = 7
mid = (6+7)/2 = 6

key < A[6] =>
low = 6, high = 5

key > A[6] =>
low = 7, high = 7

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 2

low = 0
high = 7
mid = ⌊(0+7)/2⌋ = 3

Comparison 1: key < A[3] = 6

low = 0
high = mid-1 = 3-1 = 2
mid = ⌊(0+2)/2⌋ = 1

Comparison 2: key < A[1] = 3

low = 0
high = mid-1 = 1-1 = 0
mid = ⌊(0+0)/2⌋ = 0

Comparison 3: key > A[0] = 1

low = 0
high = mid-1 = 0-1 = -1

Since (low > high) => "Not found"

A = { 1, 3, 4, 6, 8, 9, 11 }
key = 8

low = 0
high = 7
mid = ⌊(0+7)/2⌋ = 3

Comparison 1: key > A[3] = 6

low = mid+1 = 3+1 = 4
high = 7
mid = ⌊(4+7)/2⌋ = 5

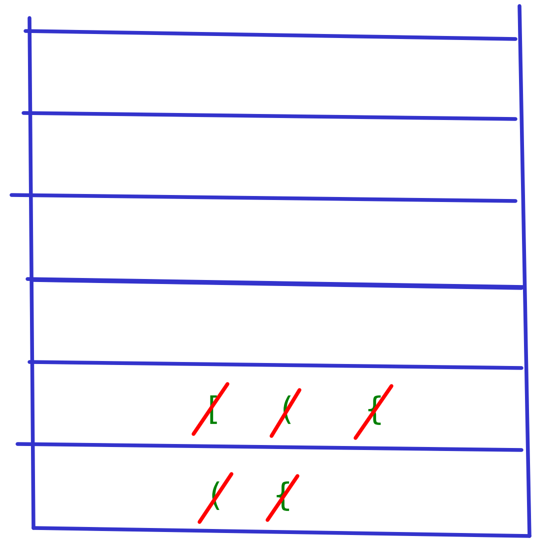
Comparison 2: key < A[5] = 9

low = 4
high = mid-1 = 5-1 = 4
mid = ⌊(4+4)/2⌋ = 4

Comparison 3: key == A[4] = 8

Return (mid) => "Found at index 4"

```
int main ( int argc, char *argv[] )
{
        int i;
        for(i=0; i<6; i++)
        { ... }
}
```




( [ ] ) { ( ) { } }  ---  VALID

( [ ) ]  ---  INVALID

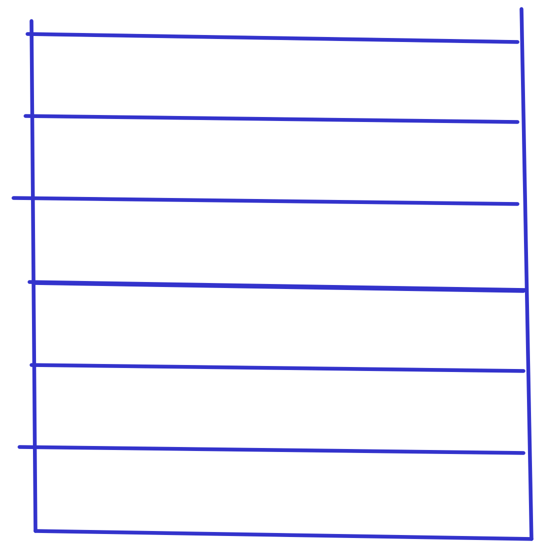
( [ ]  ---  INVALID (Stack not empty)

( [ )  ---  INVALID

[ ] )  ---  INVALID

{ { } } }  ---  INVALID

```
bool bracketCheck(const std::string& s){
        STACK st;
        int i, n;
        char ch, tmp;
        n = s.length();
        for(i=0; i<n; i++){
                ch = s[i];
                if( (ch=='(') || (ch=='{') || (ch=='[') )
                        st.push(ch);
                else if( ch == ')' ){
                        if( st.isEmpty() )
                                return false;
                        tmp = st.pop();
                        if( tmp != '(' )
                                return false;
                }
                else if( ch == '}' ){
                        if( st.isEmpty() )
                                return false;
                        tmp = st.pop();
                        if( tmp != '{' )
                                return false;
                }
                else if( ch == ']' ){
                        if( st.isEmpty() )
                                return false;
                        tmp = st.pop();
                        if( tmp != '[' )
                                return false;
                }
        }
        if( !st.isEmpty() )
                return false;
        return true;
}
```



( ( ) ( ) )

ct = 1, 2, 1, 2, 1, 0

( ( ) ( ) ( )

ct = 1, 2, 1, 2, 1, 2, 1

( ( ) ) ) ( ( )

ct = 1, 2, 1, 0, -1, 0, 1, 0

( [ ] ) [ ( ] )

ct1 = 1, 0
ct2 = 1, 0

1 + (2 * 3) - 4 -> Infix expression ( <operand1> <operator> <operand2> )

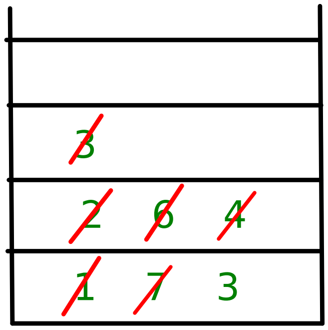
1 2 3 * + 4 - -> Postfix expression ( <operand1> <operand2> <operator> )

1 2 3 * + 4 -

1 6 + 4 -

7 4 -

3




op2 = st.pop() = 3
op1 = st.pop() = 2
result = op1 * op2 = 2 * 3 = 6
st.push(result)

op2 = st.pop() = 6
op1 = st.pop() = 1
result = op1 + op2 = 1 + 6 = 7
st.push(result)

1 2 3 * + 4 - $

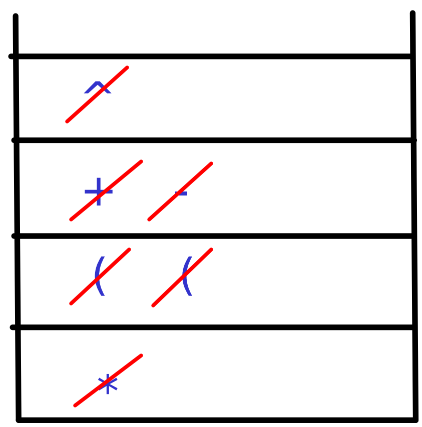
op2 = st.pop() = 4
op1 = st.pop() = 7
result = op1 - op2 = 7 - 4 = 3
st.push(result)

-2 4 + -5 -
=> (-2 + 4) - (-5)

2 * (1 + 3) * (5 - 4) = 2 * 4 * 1 = 8

2 1 3 + 5 4 - * * = 2 4 5 4 - * * = 2 4 1 * * = 2 4 * = 8

I/p: 2 * ( 1 + 3 ^ 7 ) * ( 5 - 4 ) $




O/p: 2 1 3 7 ^ + * 5 4 - *

1 + 2 * 3 - 4 -> Infix expression ( <operand1> <operator> <operand2> )

- + * 2 3 1 4 -> Prefix expression ( <operator> <operand1> <operand2> )

- + * 2 3 1 4

- + 6 1 4

- 7 4

3

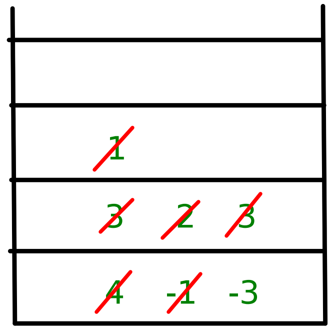
(1 + 2) * (3 - 4)

* + 1 2 - 3 4

(1 + 2) * (3 - 4) = 3 * -1 = -3

* + 1 2 - 3 4

4 3 - 2 1 + * $




op1 = st.pop() = 3
op2 = st.pop() = 4
result = op1 - op2 = 3 - 4 = -1
st.push(result)

op1 = st.pop() = 1
op2 = st.pop() = 2
result = op1 + op2 = 1 + 2 = 3
st.push(result)

op1 = st.pop() = 3
op2 = st.pop() = -1
result = op1 * op2 = 3 * (-1) = -3

Infix Expression: ( 1 + 2 ) * ( 3 - 4 )

Reverse of Infix: ( 4 - 3 ) * ( 2 + 1 )

Postfix of Reverse: 4 3 - 2 1 + *

Reverse of above: * + 1 2 - 3 4

Infix Expression: ( 1 + 2 * 3 ) / ( 3 - 2 * 4 )

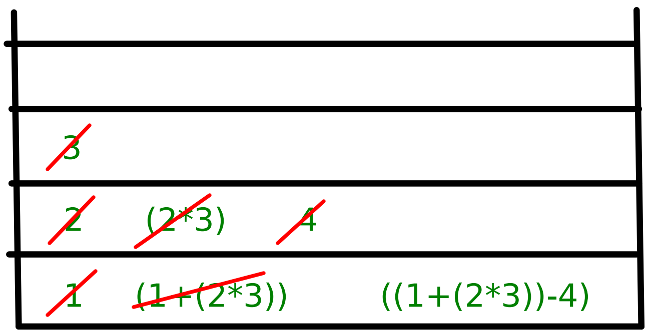
Reverse of Infix: ( 4 * 2 - 3 ) / ( 3 * 2 + 1 )

Postfix of Reverse: 4 2 * 3 - 3 2 * 1 + /

Reverse of above: / + 1 * 2 3 - 3 * 2 4

1 2 3 * + 4 - -> Postfix expression ( <operand1> <operand2> <operator> )

1 + (2 * 3) - 4 -> Infix expression ( <operand1> <operator> <operand2> )




1 2 3 * + 4 - $

((1+(2*3))-4)

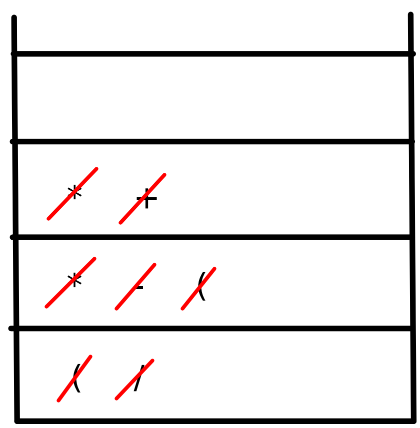
exp2 = st.pop() = 3
exp1 = st.pop() = 2
result = "(<exp1> * <exp2>)" = "(2 * 3)"
st.push(result)

exp2 = st.pop() = "(2 * 3)"
exp1 = st.pop() = 1
result = "(<exp1> + <exp2>)" = "(1 + (2 * 3))"
st.push(result)

exp2 = st.pop() = 4
exp1 = st.pop() = "(1 + (2 * 3))"
result = "(<exp1> - <exp2>)" = ((1 + (2 * 3)) - 4)
st.push(result)

Input: ( 4 * 2 - 3 ) / ( 3 * 2 + 1 ) $

Output: 4 2 * 3 - 3 2 * 1 + /



first(): Return the position of the first element of $S$; an error occurs if $S$ is empty.

last(): Return the position of the last element of $S$; an error occurs if $S$ is empty.

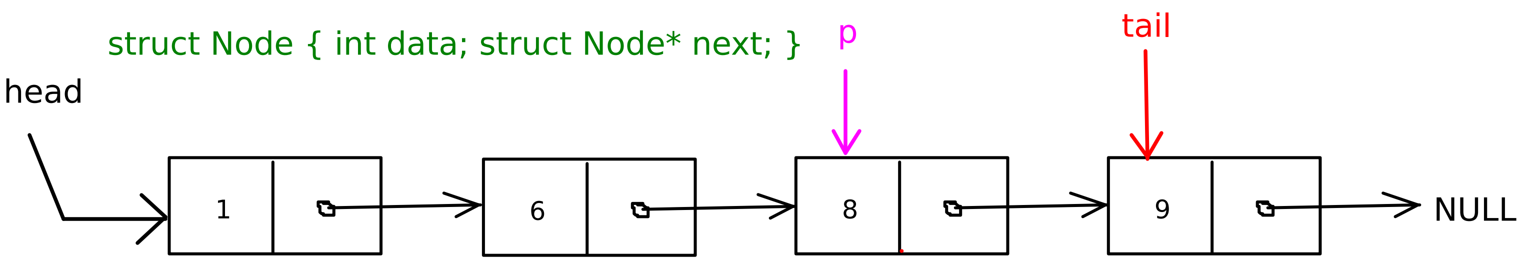
before($p$): Return the position of the element of $S$ preceding the one at position $p$; an error occurs if $p$ is the first position.

after($p$): Return the position of the element of $S$ following the one at position $p$; an error occurs if $p$ is the last position.

```
struct Node* before(p) {
   struct Node* currNode = head;
   if (p == NULL || p == head)
       return NULL;
   while (currNode->next != p)
       currNode = currNode->next;
   return currNode; }

struct Node* after(p) { return p->next; }
```

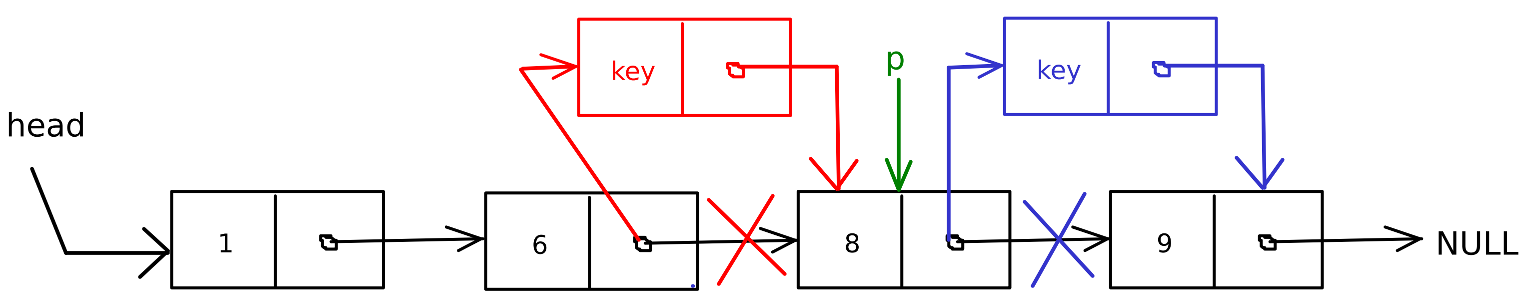
```
struct Node* first() { return head; }

struct Node* last() {
   struct Node* currNode;
   currNode = head;
   if (head == NULL)
      return NULL;
   while (currNode->next != NULL)
        currNode = currNode->next;
   return currNode;
}

struct Node* last() { return tail; }

int getDataAt(struct Node* p) {
     return p->data;
}
```

```
struct Node { int data; struct Node* next; }
```




```
int getDataAtPos(int k) {
i = 1;
struct Node* currNode = head;
while( i < k  && currNode->next != NULL ) {
    currNode = currNode->next;  i = i + 1; }
if(i==k)     return currNode->data;
else         return NULL; }
```

```
int size() {
size = 0;
struct Node* currNode = head;
while( currNode != NULL ) {
       currNode = currNode->next;
       size = size + 1; }
return size; }
```




```
insertBefore(struct Node* p, int key)
{
       struct Node* newNode;
       newNode = ....;  // allocate memory
       newNode->data = key;
       newNode->next = p;
       struct node* currNode = head;
       while(currNode->next != p)
              currNode = currNode->next;
       currNode->next = newNode;
}
```

```
insertAfter(struct Node* p, int key)
{
       struct Node* newNode;
       newNode = ....;  // allocate memory
       newNode->data = key;
       newNode->next = p->next;
       p->next = newNode;
}
```

first(): Return the position of the first element of $S$; an error occurs if $S$ is empty.

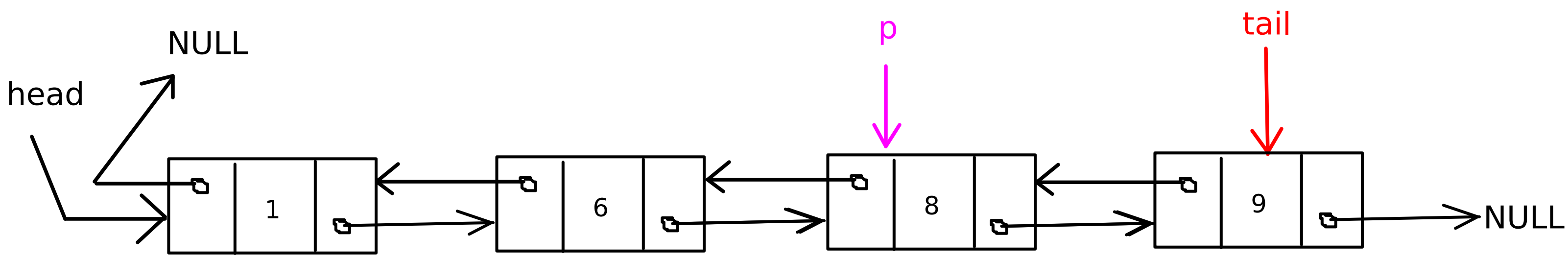
last(): Return the position of the last element of $S$; an error occurs if $S$ is empty.

before($p$): Return the position of the element of $S$ preceding the one at position $p$; an error occurs if $p$ is the first position.

after($p$): Return the position of the element of $S$ following the one at position $p$; an error occurs if $p$ is the last position.

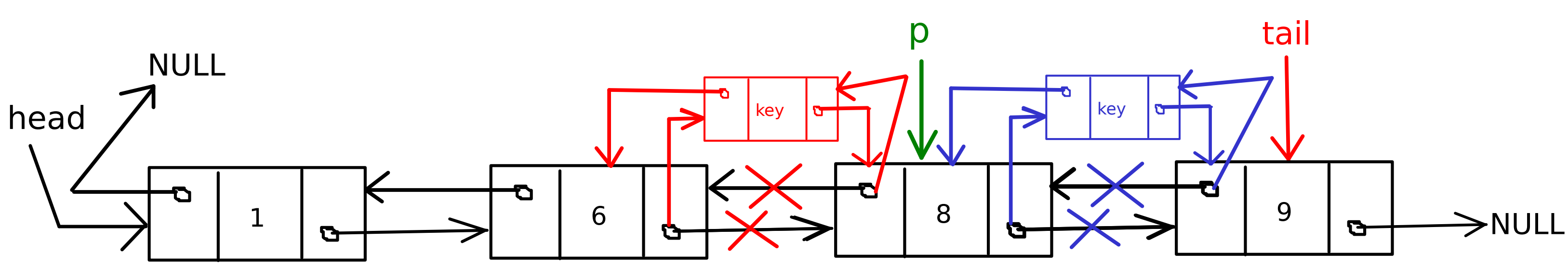
```
struct Node* first() { return head; }

struct Node* last() {
   struct Node* currNode;
   currNode = head;
   if (head == NULL)
      return NULL;
   while (currNode->next != NULL)
        currNode = currNode->next;
   return currNode;
}

struct Node* last() { return tail; }

int getDataAt(struct Node* p) {
     return p->data;
}
```

```
struct Node {
        int data;
        struct Node* prev;
        struct Node* next;
}

struct DLL {
        int size;
        struct Node* head;
        struct Node* tail;
}

struct Node* before(p) {  return p->prev;  }

struct Node* after(p) {  return p->next;  }
```




```
int getDataAtPos(int k) {
i = 1;
struct Node* currNode = head;
while( i < k  && currNode->next != NULL ) {
    currNode = currNode->next;  i = i + 1; }
if(i==k)     return currNode->data;
else         return NULL; }
```

```
int size() {
size = 0;
struct Node* currNode = head;
while( currNode != NULL ) {
       currNode = currNode->next;
       size = size + 1; }
return size; }
```




```
insertBefore(struct Node* p, int key)
{
       struct Node* newNode;
       newNode = ....;  // allocate memory
       newNode->data = key;
       newNode->prev = p->prev;
       newNode->next = p;
       p->prev->next = newNode;
       p->prev = newNode;
}
```

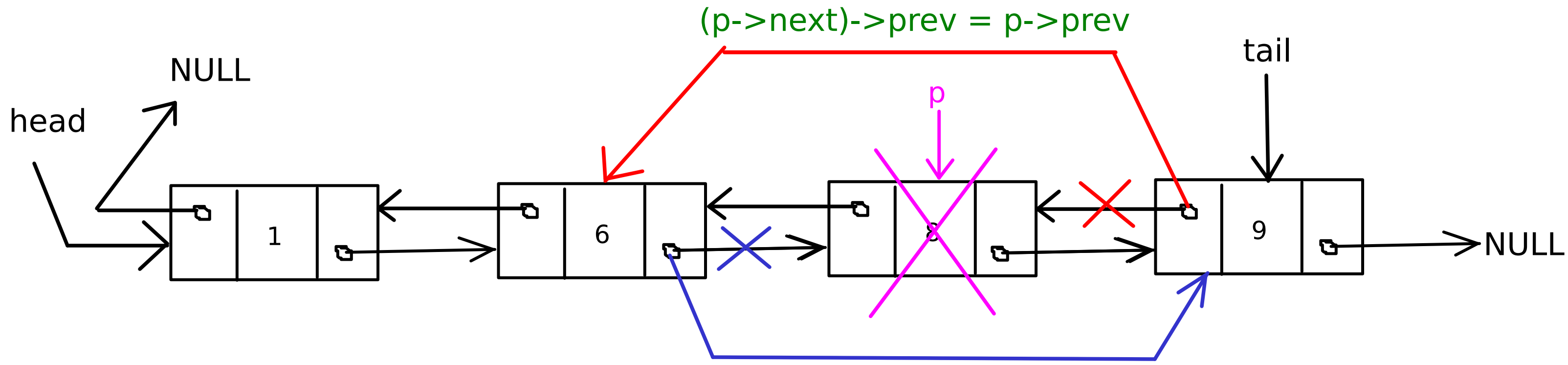
```
insertAfter(struct Node* p, int key)
{
       struct Node* newNode;
       newNode = ...;  // allocate memory
       newNode->data = key;
       newNode->prev = p;
       newNode->next = p->next;
       p->next->prev = newNode;
       p->next = newNode;
}
```

```
void insertAtBeginningSLL(head, key) {
	struct Node* newNode;
	newNode = ...;  // allocate memory
	newNode->data = key;
	newNode->next = head;
	head = newNode;
}
```

```
void insertAtBeginningDLL(head, key) {
	struct Node* newNode;
	newNode = ...;  // allocate memory
	newNode->data = key;
	newNode->prev = NULL;
	newNode->next = head;
	if( head != NULL)
		head->prev = newNode;
	head = newNode;
}
```

```
void insertAtEndSLL(head, key) {
	struct Node* newNode;
	newNode = ...;  // allocate memory
	newNode->data = key;
	newNode->next = NULL;
	if( head == NULL ) {
		head = newNode;
		// tail = newNode;
	}
	else {
		struct Node* currNode = head;
		while( currNode->next != NULL)
			currNode = currNode->next;
		currNode->next = newNode;
		// tail->next = newNode;
		// tail = newNode;
	}
}
```

```
void insertAtEndDLL(head, key) {
	struct Node* newNode;
	newNode = ...;  // allocate memory
	newNode->data = key;
	newNode->next = NULL;
	if( head == NULL ) {
		head = newNode;
		// tail = newNode;
	}
	else {
		struct Node* currNode = head;
		while( currNode->next != NULL)
			currNode = currNode->next;
		currNode->next = newNode;
		// tail->next = newNode;
		// newNode->prev = tail;
		// tail = newNode;
	}
}
```

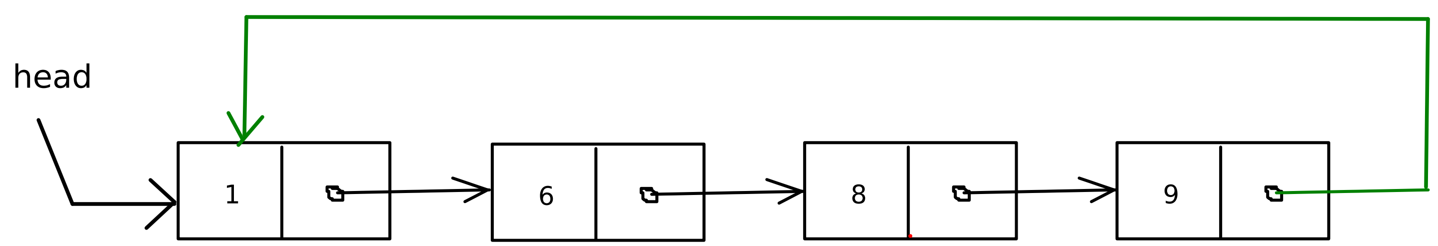


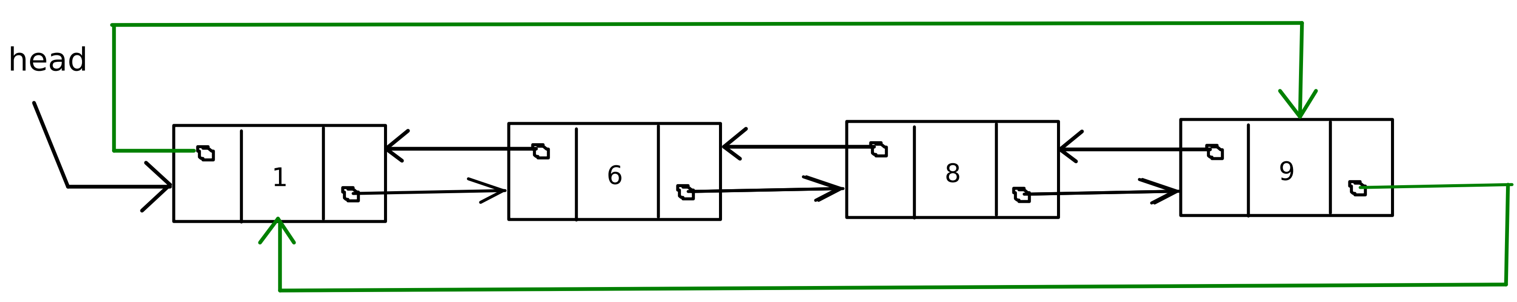

```
void deleteDLL(struct Node* p) {
        (p->prev)->next = p->next;
        (p->next)->prev = p->prev;
        free(p);  // de-allocate Node p
}
```

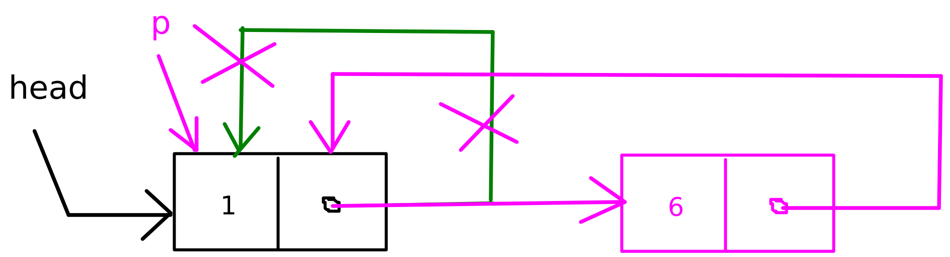
```
void deleteSLL(struct Node* p) {
        struct Node* currNode = head;
        while(currNode->next != p)
                currNode = currNode->next;
        currNode->next = p->next;
        free(p);  // de-allocate Node p
}
```

```
void swapNodes(Node* p, Node* q) {
        (p->prev)->next = q;
        (p->next)->prev = q;
        (q->prev)->next = p;
        (q->next)->prev = p;
        struct Node* tmp;
        tmp = p->next;
        p->next = q->next;
        q->next = tmp;
        tmp = p->prev;
        p->prev = q->prev;
        q->prev = tmp;
}
```

## Circular Lists (both Singly-linked and Doubly-linked)




```
newNode->next = p->next
p->next = newNode;
```

Abstract Data Type (ADT) --

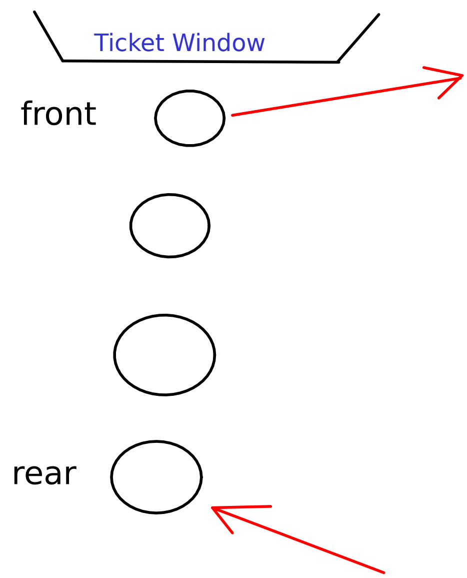
In computer science, an abstract data type (ADT) is a mathematical model for data types. An abstract data type is defined by its behavior (semantics) from the point of view of a user, of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

Eg. - Queue, Stack, Hash Table, Linked List, Binary Search Tree etc.

---

Queue<type> : FIFO (First In First Out) linear data structure,
it supports the operations of enqueue (at rear end) and dequeue (at front end)

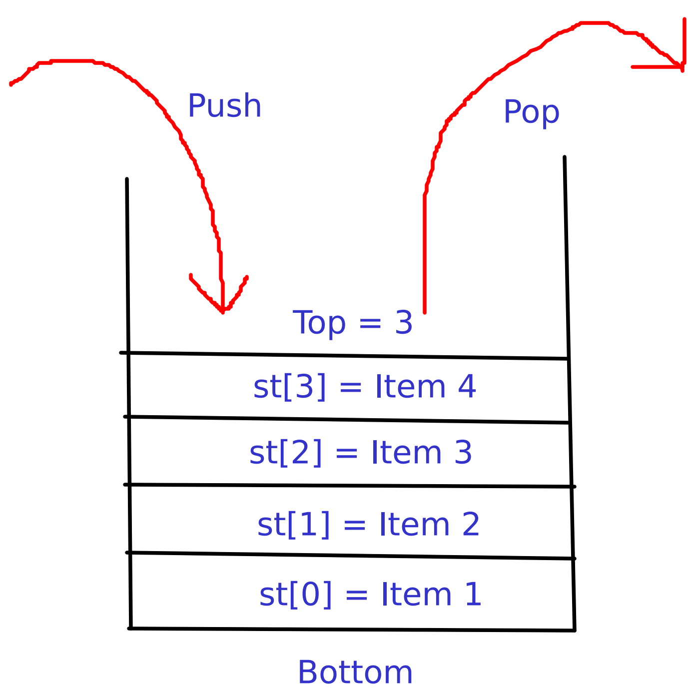
Operations:
void enqueue(<Type> key)
<Type> dequeue()
int size()
<Type> peek()
bool is_empty()
bool is_full()




---

Stack<type> : LIFO (Last In First Out) linear data structure,
it supports the operations of push (insertion) and pop (deletion), both at the same end

Operations:
void push(<Type> key)
<Type> pop()
int size()
<Type> top() / peek()
bool is_empty()
bool is_full()



Array Implementation of Queue:

```
class Queue<Type T>
{
        T q[MAXSIZE];
        int front, rear;

        Queue() { rear = -1; front = 0; }

        void enqueue(T data) {
                if( is_full() == true )
                        return "Error: Queue is full!";
                rear = (rear + 1) % MAXSIZE;
                q[rear] = data;
        }

        T dequeue() {
                if( is_empty() == true )
                        return "Error: Queue is empty!";
                tmp = q[front];
                front = (front + 1) % MAXSIZE;
                return tmp;
        }

        T peek() { return q[front]; }

        int size() {
                int size = (rear - front + 1);
                if(size < 0)
                        size = size + MAXSIZE;
                return size;
        }

        bool is_empty() { return ( size()==0 ); }

        bool is_full() { return ( size()==MAXSIZE ); }
}
```

Array Implementation of Stack:

```
class Stack<Type T>
{
        T st[MAXSIZE];
        int top;

        Stack() { top = -1; }

        void push(T data) {
                if( is_full() == true )
                        return "Error: Stack is full!";
                top++;
                st[top] = data;
        }

        T pop() {
                if( is_empty() == true )
                        return "Error: Stack is empty!";
                tmp = st[top];
                top--;
                return tmp;
        }

        T top() { return st[top]; }

        int size() { return (top+1); }

        bool is_empty() { return (top==-1); }

        bool is_full() { return (top==MAXSIZE-1); }
}
```

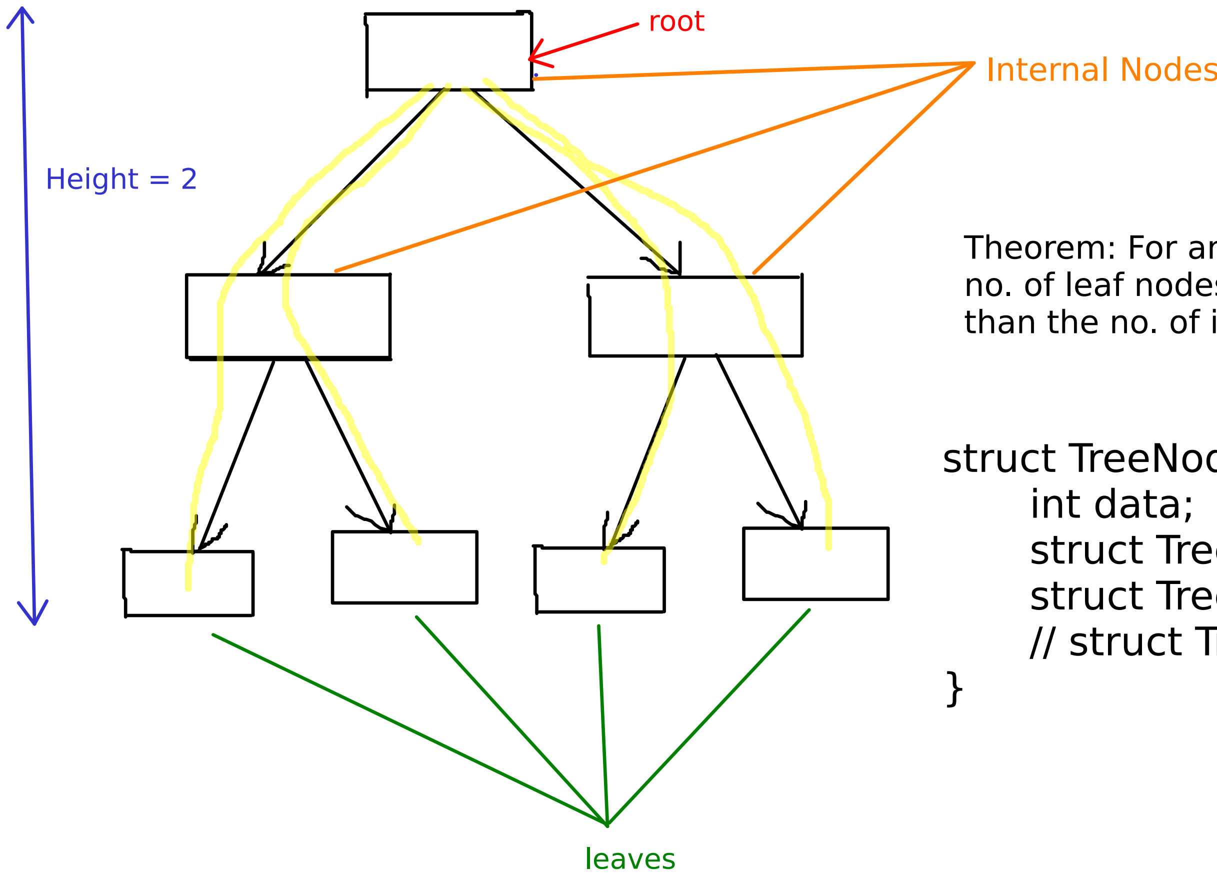
Operations to be performed (in sequence):
enqueue(44), enqueue(52), enqueue(69), enqueue(72), dequeue(), dequeue(), dequeue(), enqueue(37), dequeue(), enqueue(28), enqueue(13), dequeue(), dequeue()

q:

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 13 | ~~52~~ | ~~69~~ | ~~72~~ | ~~37~~ | ~~28~~ |

rear = ~~-1~~ ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 0

front = ~~0~~ ~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ 0

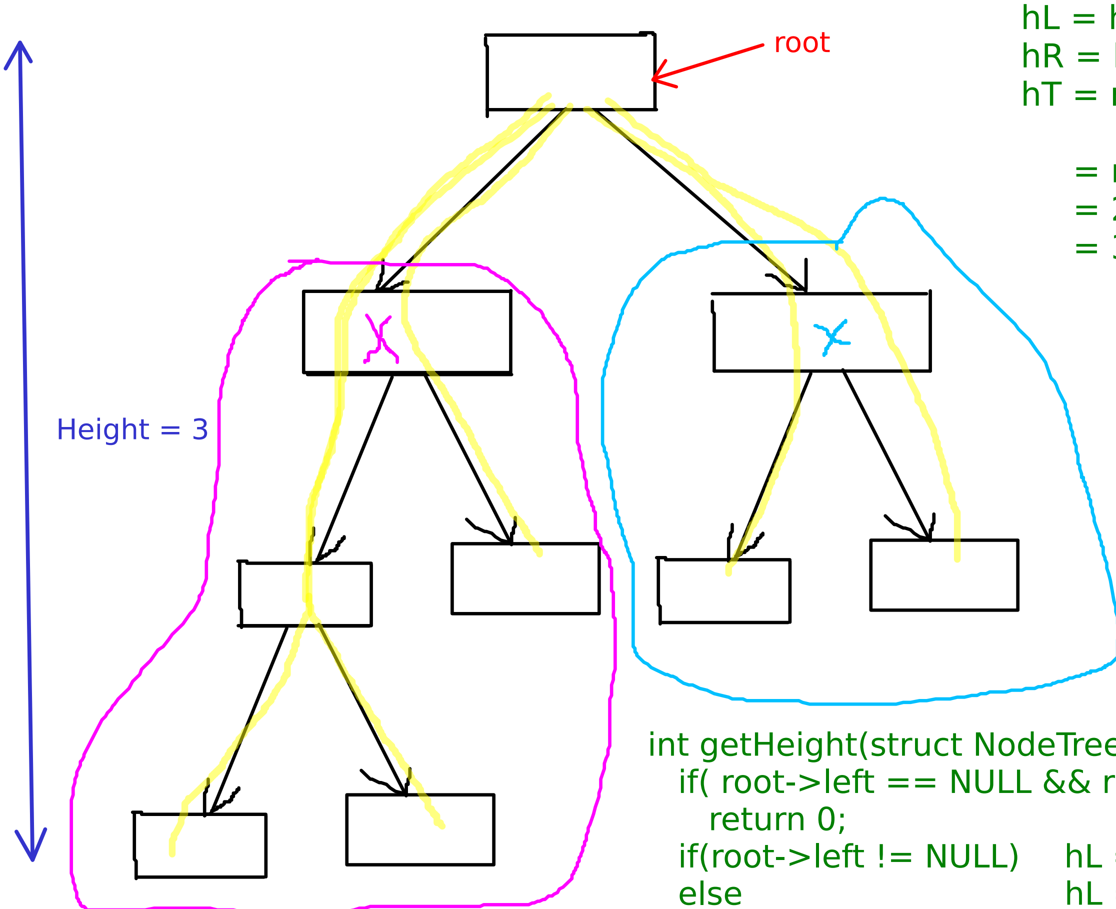



Binary Tree

N-ary Tree

Theorem: For any binary tree,
no. of leaf nodes is always 1 more
than the no. of internal nodes.

```
struct TreeNode {
        int data;
        struct TreeNode* left;
        struct TreeNode* right;
        // struct TreeNode* parent;
}
```




hL = height of left subtree
hR = height of right subtree
hT = max(hL, hR) + 1

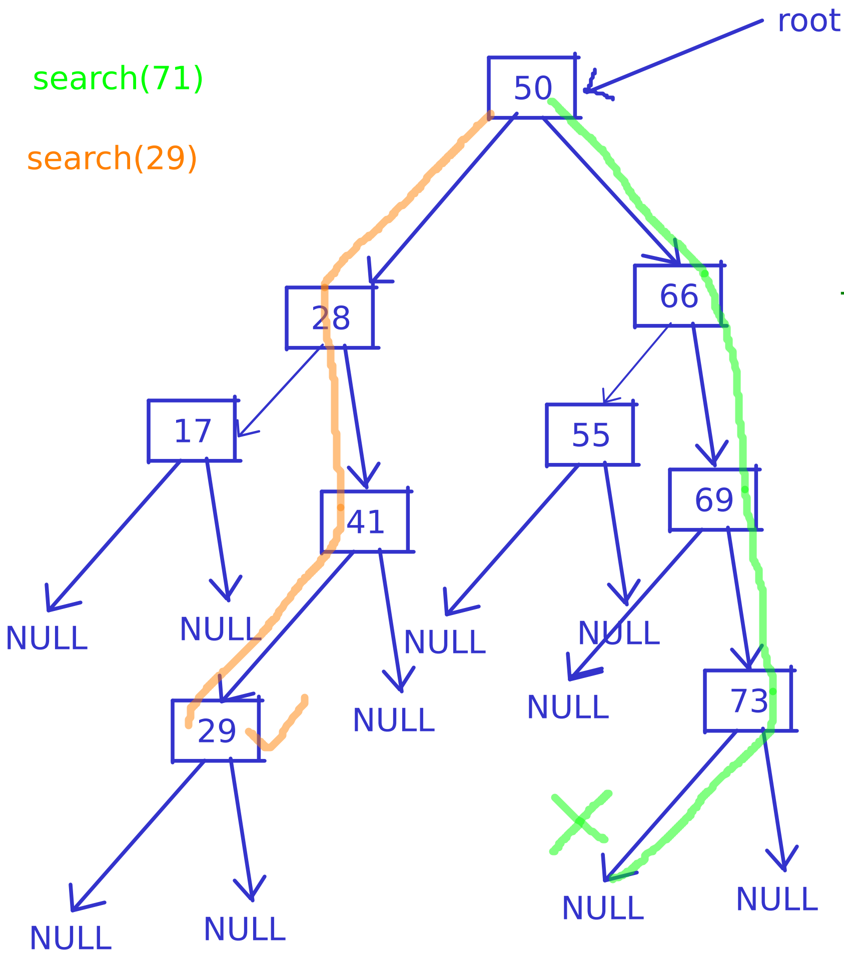
= max(2,1) + 1
= 2 + 1
= 3

| h | #leaves | #total |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 2 | 3 |
| 2 | 4 | 7 |
| 3 | 8 | 15 |

```
int getHeight(struct NodeTree* root) {
   if( root->left == NULL && root->right == NULL)
      return 0;
   if(root->left != NULL)    hL = getHeight( root->left );
   else                      hL = 0;
   if(root->right != NULL)   hR = getHeight( root->right );
   else                      hR = 0;
   if (hL > hR)          return (hL + 1);
   else                  return (hR + 1);
}
```

Theorem: For a binary tree having
height h, the maximum possible no.
of leaf nodes is 2^h, and the total no.
of nodes can be at most 2^(h+1)-1.

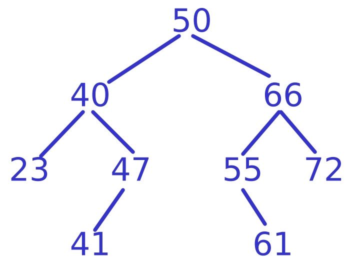
Create a Binary Search Tree (BST) by inserting the following elements (in sequence):
50, 28, 41, 66, 69, 73, 55, 17, 29




```
struct TreeNode {
        int data;
        struct TreeNode* left;
        struct TreeNode* right;
}
```

```
TreeNode* search(TreeNode* root, int k) {
        if( root == NULL ) {
                print("Not found!");
                return NULL;
        }
        if( k == root->data )
                return root;
        if( k < root->data )
                return search( root->left, k );
        if( k > root->data )
                return search( root->right, k );
}
```

```
Treenode* insert(TreeNode* root, int k) {
   if( root == NULL ) {
        struct TreeNode* newNode = ...;
        newNode->data = k;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
   }
   else {
        if( k <= root->data )
              root->left = insert( root->left, k );
        if( k > root->data )
              root->right = insert( root->right, k );
        return root;
   }
}
```




Pre-order: ROOT, LEFT Subtree, RIGHT Subtree

Post-order: LEFT Subtree, RIGHT Subtree, ROOT

In-order: LEFT Subtree, ROOT, RIGHT Subtree

---

( ... ) 50 ( ... )

( ( ... ) 40 ( ... ) ) 50 ( ... )

( ( 23 ) 40 ( ... ) ) 50 ( ... )

( ( 23 ) 40 ( ( ... ) 47 ) ) 50 ( ... )

( ( 23 ) 40 ( ( 41 ) 47 ) ) 50 ( ... )

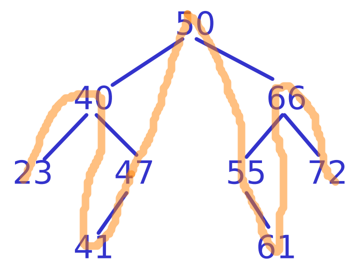
( ( 23 ) 40 ( ( 41 ) 47 ) ) 50 ( ( ... ) 66 ( ... ) )

( ( 23 ) 40 ( ( 41 ) 47 ) ) 50 ( ( 55 ( ... ) ) 66 ( ... ) )

( ( 23 ) 40 ( ( 41 ) 47 ) ) 50 ( ( 55 ( 61 ) ) 66 ( ... ) )

( ( 23 ) 40 ( ( 41 ) 47 ) ) 50 ( ( 55 ( 61 ) ) 66 ( 72 ) )

```
void inorder(struct TreeNode* root) {
        if( root == NULL ) {
                print("Tree is empty!");
                return;
        }
        if( root->left != NULL )
                inorder( root->left );
        print( root->data );
        if( root->right != NULL )
                inorder( root->right );
}
```




---

( ... ) ( ... ) 50

( ( ... ) ( ... ) 40 ) ( ... ) 50

( ( 23 ) ( ... ) 40 ) ( ... ) 50

( ( 23 ) ( ( ... ) 47 ) 40 ) ( ... ) 50

( ( 23 ) ( ( 41 ) 47 ) 40 ) ( ... ) 50

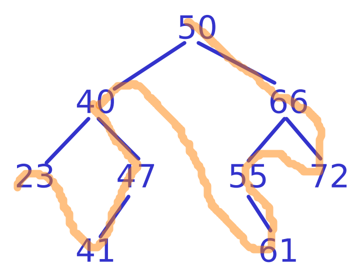
( ( 23 ) ( ( 41 ) 47 ) 40 ) ( ( ... ) ( ... ) 66 ) 50

( ( 23 ) ( ( 41 ) 47 ) 40 ) ( ( ( ... ) 55 ) ( ... ) 66 ) 50

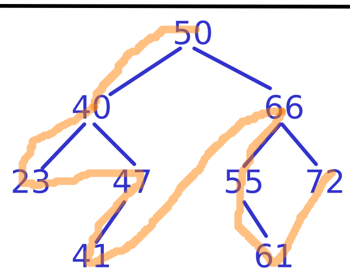
( ( 23 ) ( ( 41 ) 47 ) 40 ) ( ( ( 61 ) 55 ) ( ... ) 66 ) 50

( ( 23 ) ( ( 41 ) 47 ) 40 ) ( ( ( 61 ) 55 ) ( 72 ) 66 ) 50

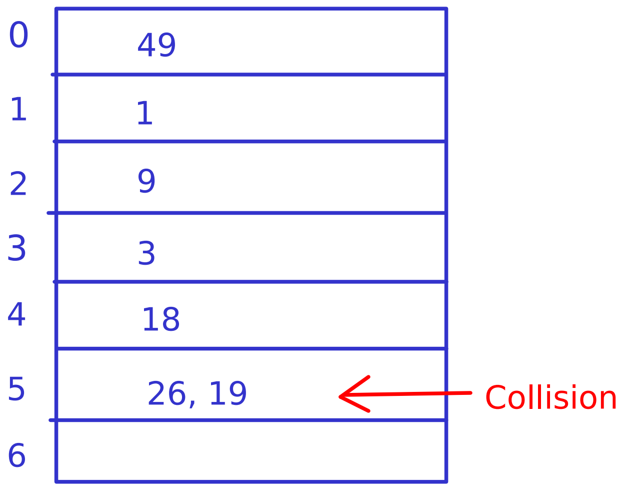
```
void postorder(struct TreeNode* root) {
        if( root == NULL ) {
                print("Tree is empty!");
                return;
        }
        if( root->left != NULL )
                postorder( root->left );
        if( root->right != NULL )
                postorder( root->right );
        print( root->data );
}
```




---




```
void preorder(struct TreeNode* root) {
        if( root == NULL ) {
                print("Tree is empty!");    return;  }
        print( root->data );
        if( root->left != NULL )          preorder( root->left );
        if( root->right != NULL )         preorder( root->right );
}
```



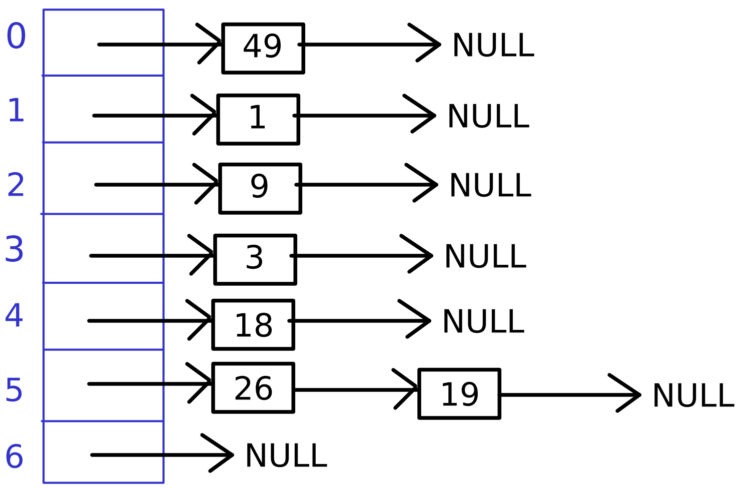

1, 3, 9, 26, 49, 18, 19

If k is a key, the define a function:

h(k) = k % 7

```
find(x) {
    pos = x % 7;
    if( T[pos] == x )
        return True;
    else
        return False;
}
```

```
#define EMPTY -1

insert(x) {
    pos = x % 7;
    if ( T[pos] = EMPTY )
        T[pos] = x;
    else
        // handle collision;
}
```




Open Addressing

Separate Chaining

```
find(x) {
    pos = x % 7;
    currNode = T[pos];
    while(currNode != NULL) {
        if (currNode -> data == x)
                return True;
        currNode = currNode->next;
    }
    return False;
}
```

```
insert(x) {
   pos = x % 7;
   struct Node* newNode = ...;
   newNode->data = x;
   newNode->next = NULL;
   if( T[pos] == NULL )
       T[pos] = newNode;
   else {
       currNode = T[pos];
       while(currNode->next != NULL)
           currNode = currNode->next;
       currNode->next = newNode;
   }
```

| | |
|---|---|
| 0 | 40 |
| 1 | 1 |
| 2 | 68 |
| 3 | 3 |
| 4 | |
| 5 | 26 |
| 6 | 19 |

1, 26, 19, 3, 40, 68

h(k) = k % 7

h(k) = (2k + 5) % 7

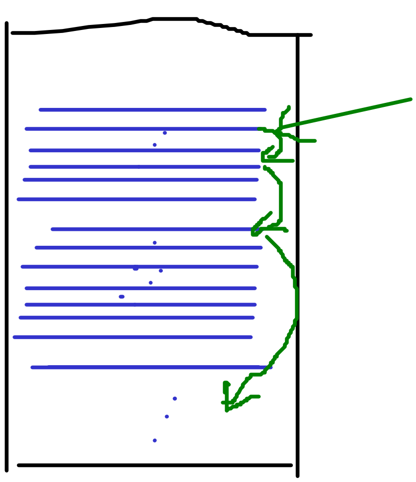
| | |
|---|---|
| 0 | 68 |
| 1 | 1 |
| 2 | 40 |
| 3 | 3 |
| 4 | |
| 5 | 26 |
| 6 | 19 |

Linear Probing: h(k, i) = ( h(k) + i ) % 7

Quadratic Probing: h(k, i) = ( h(k) + i^2 ) % 7

```
insert(k) {
        i = 0;
        pos = k % 7;
        while( T[pos] != EMPTY ) {
                i = i + 1;
                pos = ( (k % 7) + i ) % 7;
                // if( pos == k % 7 )
                //    return "Error! Hash table is FULL!"
        }
        T[pos] = k;
}
```

```
find(k) {
   i = 0;
   pos = k % 7;
   while( T[pos] != k ) {
          i = i + 1;
          pos = ( (k % 7) + i ) % 7;
          if( T[pos] == EMPTY )
                 return "Not present!"
    }
    return pos;
}
```



h(k) = k % 4

2, 4, 6, 8, 10

2 -> 2
4 -> 0
6 -> 2
8 -> 0
10 -> 2

h(k) = k % 5

2, 4, 6, 8, 10

2 -> 2
4 -> 4
6 -> 1
8 -> 3
10 -> 0

Load factor of hash table = (No. of elements inserted) / (Size of the table)