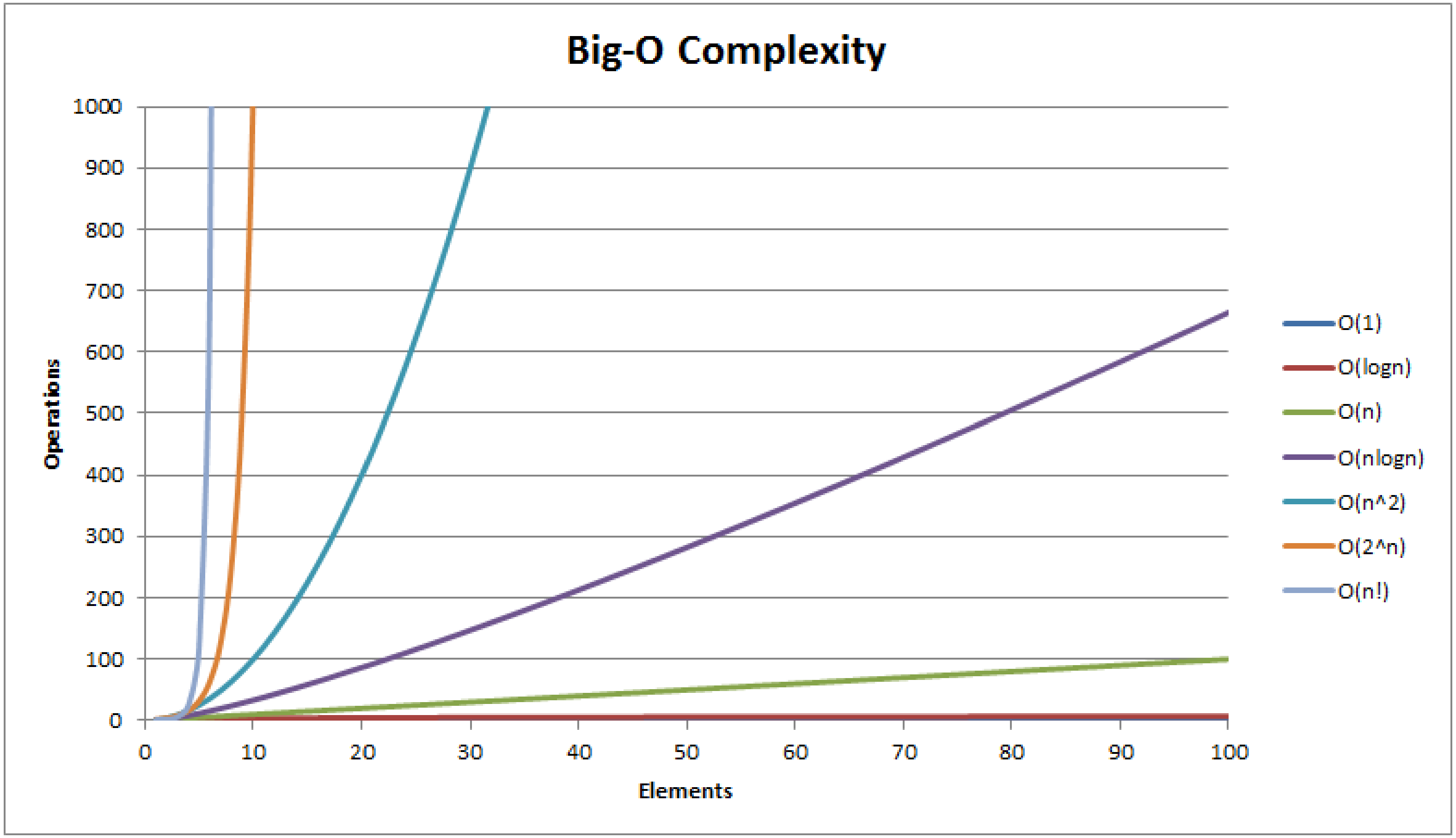
Program Efficiency & Introduction to Complexity Theory



When does implementation matter?

- There are SEVERAL algorithms that solve the SAME problem

→ Need to decide which one to choose

Problem	Algorithms
Sort Put elements in a certain order	1. Bucket sort 2. Bubble sort 3. Merge sort 4. Quick sort
Search Retrieve information stored within some data structure	1. Sequential Search 2. Binary Search
Anagrams One string is an anagram of another if the second is a rearrangement of the first	1. Checking Off 2. Sort and compare 3. Brute Force 4. Count and compare

Analysis of Execution Time

```
public static int indexOf(int[] arr, int val) {
    int arrLen = arr.length;
    for (int i = 0; i < arrLen; i++)
      if (arr[i] == val)
            return i;
    return -1;
}
```

In a sequential search of an array:

Skip[Sequential search]

- *worst-case:*

4n+4 → *complexity* is linear

- *best-case:*

7 → *complexity* is constant (independent of input size)

- *average case:*
- *4n/2 +4 = 2n+4* → *complexity* is linear

Why do you need to evaluate an algorithm?

- Find most optimal algorithm for solving given problem, considering various factors and constraints:
  - Execution time
  - Execution space (choosing the correct data structure)
  - Network bandwidth
  - …
- **<u>Goal:</u>** How fast or slow the particular algorithm performs

→**Calculate time *complexity* of the algorithm**

- **<u>Problem:</u>** Several factors impact the actual time
  - Instruction set
  - CPU
  - Brand of compiler…

Asymptotic behavior

To determine **runtime complexity:**

- Calculate T(n) (number of fundamental steps vs. problem size)
- Disregard constants
- Look how running time is affected when input size is quite large.
- Drop the terms that grow slowly (or do not grow at all) and only keep the ones that grow fast as **n** becomes larger
- Examples:
  - $T(n) = 5n + 42$

  → the fastest growing term is ***n*** → linear runtime complexity
  - $T(n) = 37n + 3n^2 + 120$

  → the fastest growing term is $\boldsymbol{n^2}$ →quadratic runtime complexity

Cost of operations: Constant Time Ops

- Each take one foundamental time "step":
  - Simple variable declaration/initialization (`double sum = 0.0;`)
  - Assignment of numeric or reference values (`var = value;`)
  - Arithmetic operation (`+, -, *, /, %`)
  - Array subscripting (`a[index]`)
  - Simple conditional tests (`x < y, p != null`)
  - Operator `new` (NOT including constructor cost)

    Note: `new` takes significantly longer than simple arithmetic or assignment, but its cost is <u>independent</u> of the problem size
- CAUTION: watch out for method calls or constructor invocations that look simple, but might be expensive

Costs of Statements

- Sequential: S1; S2; … Sn

  → sum the costs of S1 + S2 + … + Sn

- Conditional: how long it *might* take to execute the code

```
if (condition) {S1;}
else {S2;}
```

  → max cost (S1, S2) + cost of evaluating the condition

- Loop:

  Calculate cost of each iteration

  Calculate number of iterations

  → Total cost is the product of these

Costs of Statements
Method Calls

- Cost for f(a, b, c) is
  - Cost of actually **calling** the method (constant overhead)

  + cost of **evaluating** the arguments

  + cost of **parameter passing** (normally constant time in Java for both numeric and reference values)

  + cost of **executing** the method body

Analysis of Execution Time

```
public static int indexOf(int[] arr, int val) {
    int arrLen = arr.length;
    for (int i = 0; i < arrLen; i++)
      if (arr[i] == val)
            return i;
    return -1;
}
```

The fundamental instructions:

- Assigning a value to a variable: 2 ‘step’ (`int arrLen=arr.length`)
  +1 ‘step’ (`int i = 0`)
- Return statement : +1 ‘step’ (either `i` or -1)
- `for` loop : ?
  - Accessing array: 1 ‘step’ (`arr[i]`)
  - Comparing two values: + 1 ‘step’ (`arr[i] == val`)
  - Inside () of `for`: + 2 ‘steps’ (`i < arrLen; i++`)

Different types of complexities

- The ***worst-case runtime complexity*** is
  the **maximum number of steps** taken on any instance of size ***n***.

- The ***best-case runtime complexity*** is
  the **minimum number of steps** taken on any instance of size ***n***.

- The ***average case runtime complexity*** is
  an **average number of steps** taken on any instance of size ***n***.

Analysis of Execution Time

```
public static int indexOf(int[] arr, int val) {
    int n = arr.length;
    for (int i = 0; i < n; i++)
            if (arr[i] == val)
                  return i;
    return -1;
}
```

T(n)=
Outside for loop: 4 steps
+
(Inside for loop: 4 steps
*
Number of iterations: ?)

In a sequential search of an array:

- *worst-case: 4n+4*

  → *complexity* is linear
- *best-case: 7*

  → *complexity* is constant (independent of input size)
- *average case: 4n/2 +4 = 2n+4* → *complexity* is linear

What about nested loop?

```
int m=0; //executed in constant time c1
// Outer loop - executed n times
for (int i = 0; i < n; i++)
// Inner loop - be executed n times
   for(int j = 0; j < n; j++)
         sum += i * j; //executed in constant time c2
```

→Runtime complexity is **quadratic**

Rule of thumb: Simple programs can be analyzed by counting the nested loops of the program:
A **single loop** over n items → **linear** complexity
A **loop within a loop** → **quadratic** complexity
A loop within a loop within a loop yields → cubic complexity

What if number of iterations of one loop depends on the counter of the other?

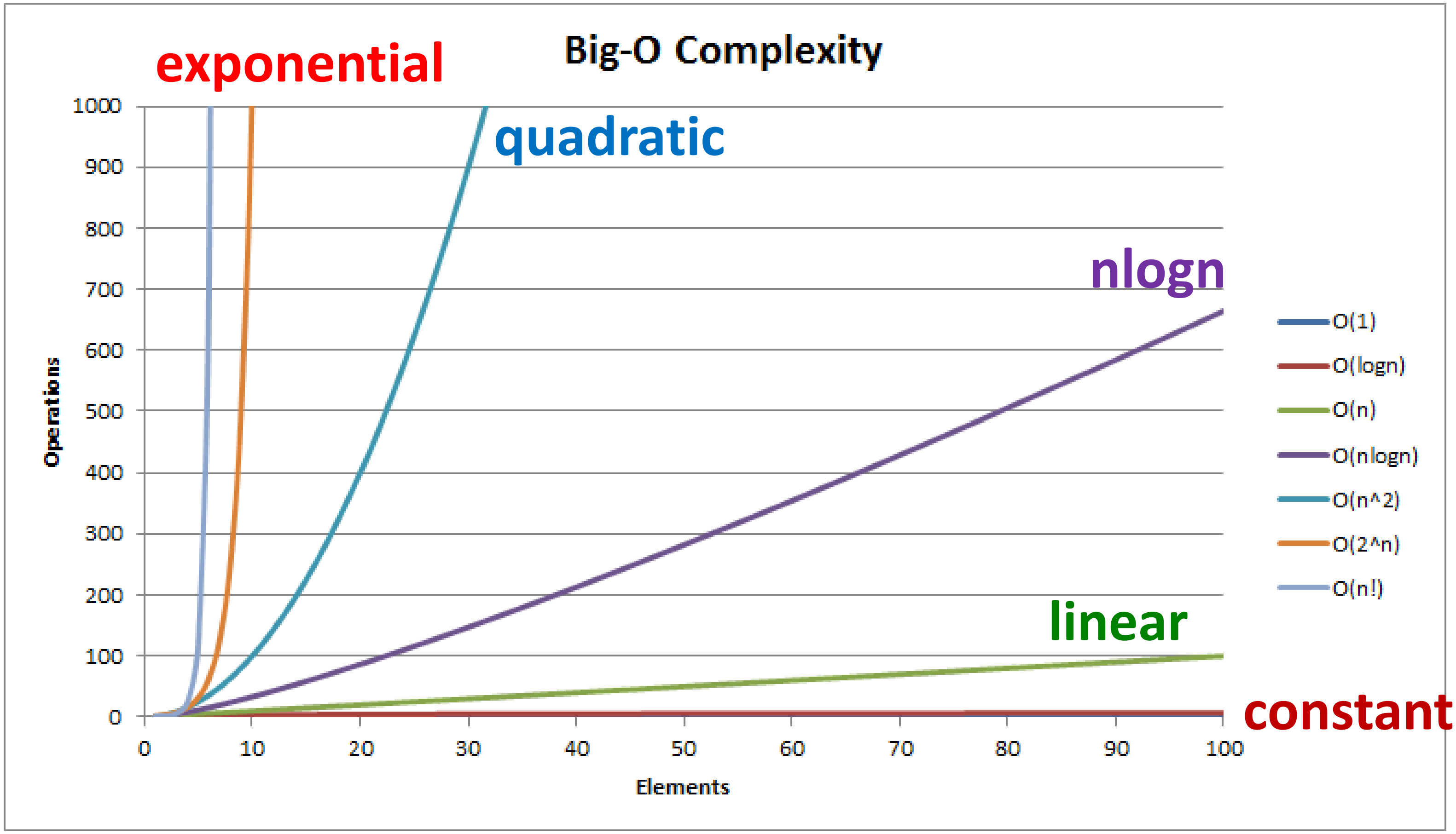
```
 int i,k,sum = 0;
for ( i = 0; i < n; i++ )
  for ( j = 0; j < i; j++ )
    sum +=i * j;
```

→Analyze inner and outer loops together :

0 + 1 + 2 + … + (n-1) = n(n-1)/2

→ Quadratic complexity

“big-O”



Big-O Complexity
exponential
quadratic
nlogn
linear
constant
1000
900
800
700
600
500
400
300
200
100
0
Operations
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Complexity Classes

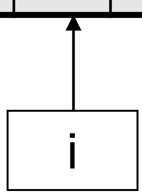
- Several common complexity classes (problem size n)
  - Constant time: O(k) or O(1)
  - Logarithmic time: O(log n) [Base doesn't matter. Why?]
  - Linear time: O(n)
  - "n log n" time: O(n log n)
  - Quadratic time: $O(n^2)$
  - Cubic time: $O(n^3)$
  - Exponential time: $O(k^n)$

- $O(n^k)$ is often called *polynomial time*

Sequential search

- Locates a target value in an array/list by examining each element from start to finish.
  - On Average O(n)
  - Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



Notice that the array is sorted. Could we take advantage of this?

Binary search

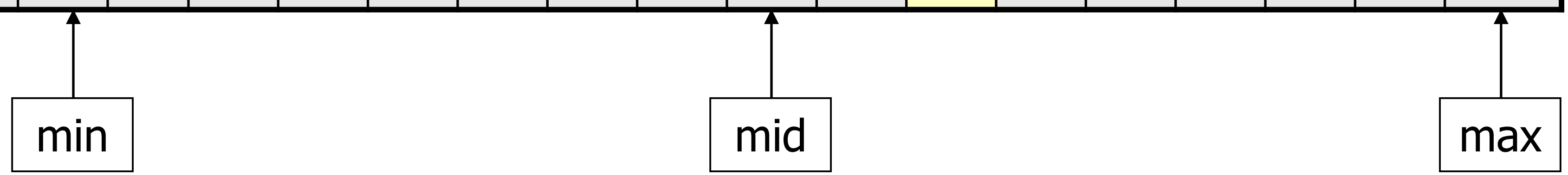
- Locates a target value in a *sorted* array/list
- *Algorithm:* Examine the middle element of the array.

  If it is too big, eliminate the right half of the array and repeat.

  If it is too small, eliminate the left half of the array and repeat.

  Else it is the value we are searching for, so stop
- Example: Searching the array below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103



- How many elements will it need to examine?

What does this function do and what is its complexity ?

```
int mystery (int x) {
   if (x <= 0) throw new IllegalArgumentException();

   if (x == 1) return 0;

   return 1 + mystery (x / 2);
}
```

Try it with arguments of 4, 8 and 2.

Binary search runtime

- For an array of size N, it eliminates ½ until 1 element remains:
  N, N/2, N/4, N/8, ..., 4, 2, 1
  - How many divisions does it take?
- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach N?

    1, 2, 4, 8, ..., N/4, N/2, N
  - Call this number of multiplications "x".

    $2^x = N$

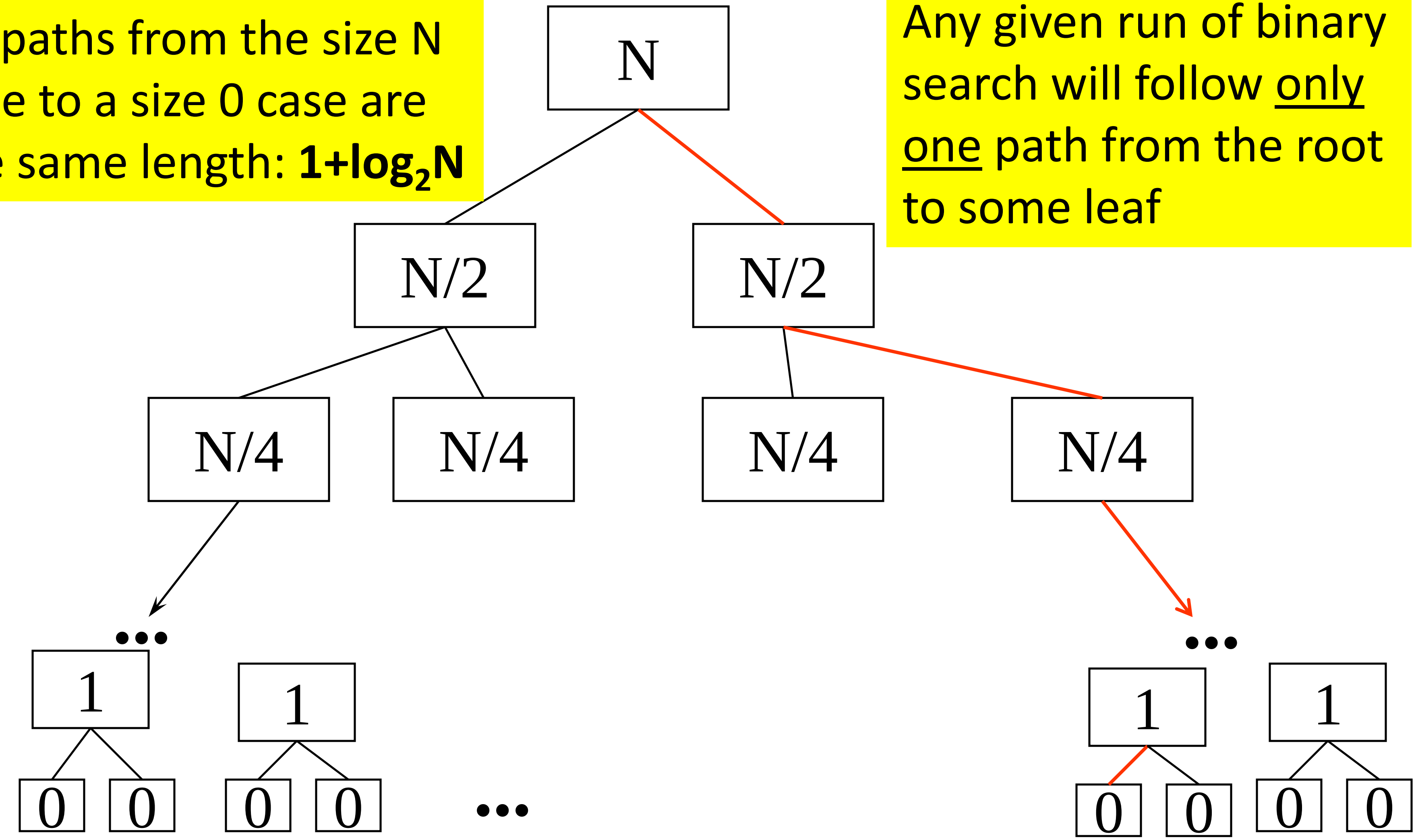
    $\mathbf{x = \log_2 N}$

→ Binary search has **logarithmic** complexity - O(logN)

Picture the Execution

All paths from the size N case to a size 0 case are the same length: **$1+\log_2 N$**

Any given run of binary search will follow <u>only one</u> path from the root to some leaf



ArrayList vs. LinkedList* in Java

	ArrayList (dynamic array)	LinkedList*
`get(int index)` **Indexing**	*O* (1) (main benefit)	O(n)
`add (E element)` **Inserting at the end**	O(n) (dynamically growing) O(1) (on average input)	O(1)
`add (int index, E element)` **Inserting at the index**	O(n) Unless at the end	*O*(1) (index ==0, main benefit) O(n)

*** with head, tail, and size**

ArrayList vs. LinkedList* in Java

	ArrayList (dynamic array)	LinkedList*
`remove(int index)` **Delete from index**	$O(1)$(index) O(n)	$O(1)$ (index ==0, index ==size , main benefit) O(n)

*** with head, tail, and size**