

# Ceng 111 – Fall 2021
# Week 13a

**Credit**: Some slides are from the “Invitation to Computer Science” book by G. M. Schneider, J. L. Gersting and some from the “Digital Design” book by M. M. Mano and M. D. Ciletti.

# Other notations for computational complexity: $\Omega()$ Notation

f(n) is $\Omega(g(n))$ if and only if there exists a real constant $c > 0$, and positive integer $n_0$, such that

$$c|g(n)| \leq |f(n)| \text{ for all } n \geq n_0$$

- Lower boundary for f(n).
- $2n = \Omega(n)$
- $n^2 = \Omega(n^2)$

Previously on CENG 111!

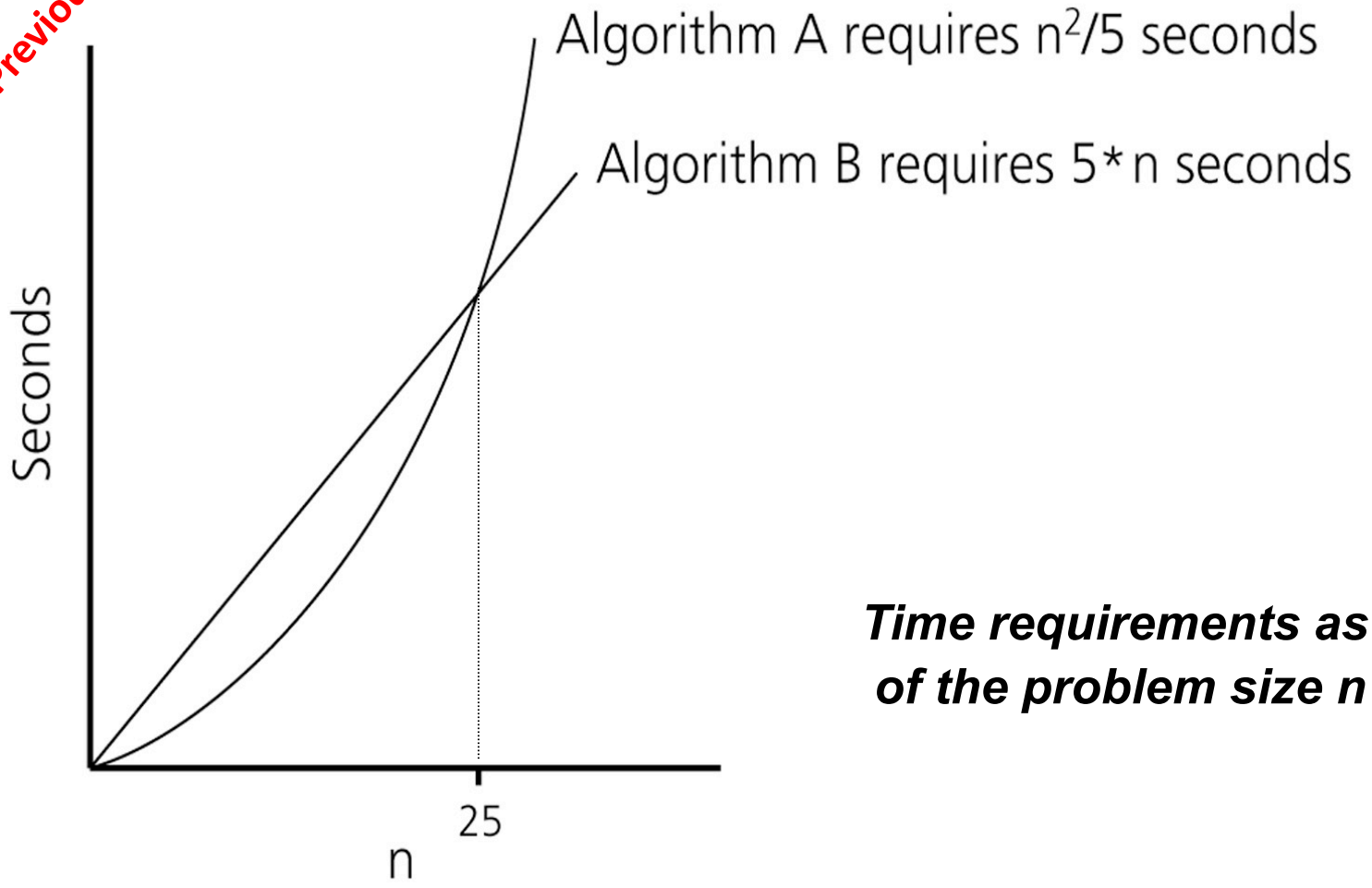
# Other notations for computational complexity: $\Theta()$ notation

- $f(n) \in \Theta(g(n))$ if and only if there exists positive real constants $c_1$ and $c_2$, and a positive integer $n_0$, such that

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$$

- Lower and upper boundary for f(n).
- $2n = \Theta(n)$
- $n^2 = \Theta(n^2)$

# Problems with growth rate analysis




***Time requirements as a function of the problem size n***

| Notation | Name | Example |
|---|---|---|
| $O(1)$ | constant | Determining if a number is even or odd; using a constant-size lookup table or hash table |
| $O(\log n)$ | logarithmic | Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap. |
| $O(n^c),\ 0 < c < 1$ | fractional power | Searching in a kd-tree |
| $O(n)$ | linear | Finding an item in an unsorted list or a malformed tree (worst case) or in an unsorted array; Adding two n-bit integers by ripple carry. |
| $O(n \log n) = O(\log n!)$ | linearithmic, loglinear, or quasilinear | Performing a Fast Fourier transform; heapsort, quicksort (best and average case), or merge sort |
| $O(n^2)$ | quadratic | Multiplying two *n*-digit numbers by a simple algorithm; bubble sort (worst case or naive implementation), shell sort, quicksort (worst case), selection sort or insertion sort |
| $O(n^c),\ c > 1$ | polynomial or algebraic | Tree-adjoining grammar parsing; maximum matching for bipartite graphs |
| $L_n[\alpha, c],\ 0 < \alpha < 1 = e^{(c+o(1))(\ln n)^{\alpha}(\ln \ln n)^{1-\alpha}}$ | L-notation or sub-exponential | Factoring a number using the quadratic sieve or number field sieve |
| $O(c^n),\ c > 1$ | exponential | Finding the (exact) solution to the traveling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search |
| $O(n!)$ | factorial | Solving the traveling salesman problem via brute-force search; finding the determinant with expansion by minors. |



From Wikipedia

# Importance of Developing Efficient Algorithms



Previously on CENG111

Sequential search vs Binary search

| Array size | No. of comparisons by seq. search | No. of comparisons by bin. search |
|---|---|---|
| 128 | 128 | 8 |
| 1,048,576 | 1,048,576 | 21 |
| ~$4.10^9$ | ~$4.10^9$ | 33 |

Execution times for algorithms with the given time complexities:

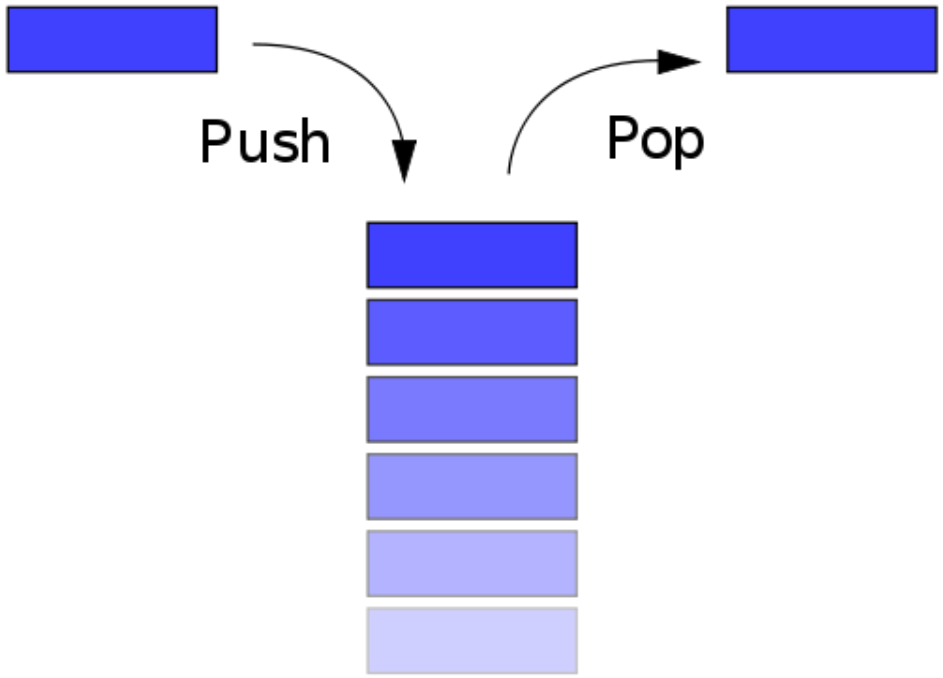
| n | f(n)=n | nlgn | $n^2$ | $2^n$ |
|---|---|---|---|---|
| 20 | 0.02 μs | 0.086 μs | 0.4 μs | 1 ms |
| $10^6$ | 1μs | 19.93 ms | 16.7 min | 31.7 years |
| $10^9$ | 1s | 29.9s | 31.7 years | !!! centuries |

Anaysis of Algorihms, A.Yazici

Previously on CENG111!

# Stacks

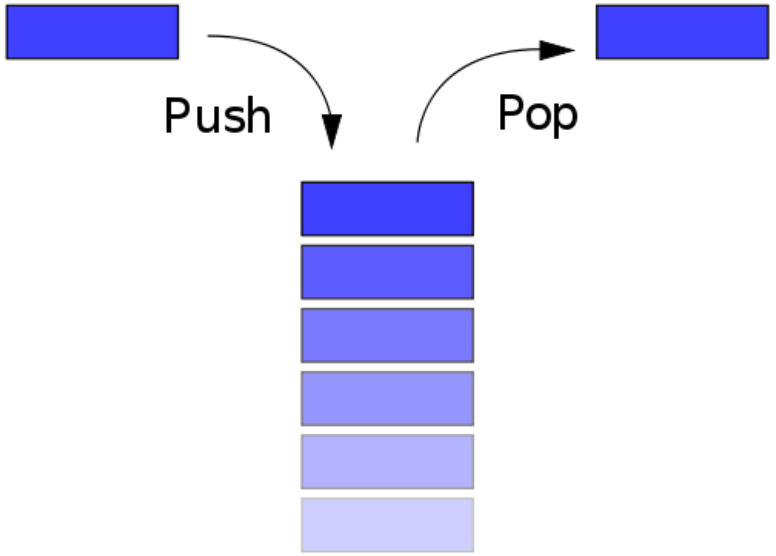
- LIFO:
  - Last In First Out
- We have seen it before (in the Shunting-Yard algorithm)
- Main operations:
  - Push
  - Pop



Previously on CENG111!

# Stacks (cont'd)

- Operations:
  1. Push
  2. Pop
  3. Top/Peek
     - Get the top element without removing it
  4. Is-Empty
     - Checks whether the stack is empty
  5. Length
     - # of elements



Previously on CENG111!

# Stacks: Formal definition

$push(item, stack)$  $item \odot stack$

- $new() \rightarrow \emptyset$
- $popoff(\xi \odot S) \rightarrow S$
- $top(\xi \odot S) \rightarrow \xi$
- $isempty(\emptyset) \rightarrow$TRUE
- $isempty(\xi \odot S) \rightarrow$ FALSE

# Implementing Stacks in Python

Previously on CENG111!



```
def CreateStack():
    """Creates an empty stack"""
    return []

def Push(item, Stack):
    """Add item to the top of Stack"""
    Stack.append(item)

def Pop(Stack):
    """Remove and return the item at the top of the Stack"""
    return Stack.pop()

def Top(Stack):
    """Return the value of the item at the top of the
        Stack without removing it"""
    return Stack[-1]

def IsEmpty(Stack):
    """Check whether the Stack is empty"""
    return Stack == []
```



# Today

- Abstract data types
  - Stack example
  - Queue
  - Priority queue
  - Tree



# Administrative Notes

- THE3:
  - Deadline: 16 January.
- Final:
  - 5 Feb December, Saturday, 13:30



# Stacks in Python (Example)

- Implement postfix implementation in Python using stacks.
  - Given a string like “3 4 + 5 7 + *”, evaluate and return the result.



# Stacks in Python (Example - Solution)

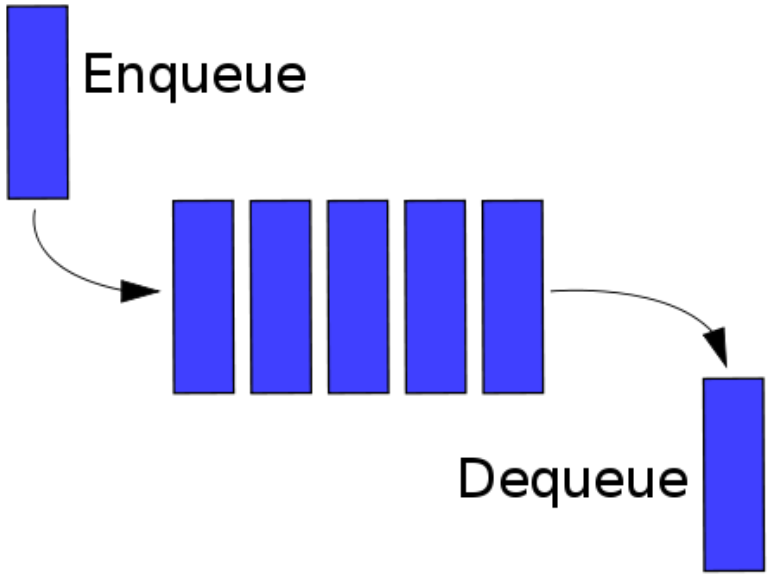
```
def postfix_eval(Exp):
        # Example Exp: "3 4 + 5 6 + *"
        Stack = CreateStack()
        Exp = Exp.split(' ')

        for token in Exp:
                if token.isdigit(): Push(token, Stack)
                else:
                        op2 = Pop(Stack)
                        op1 = Pop(Stack)
                        result = str(eval(op1 + token + op2))
                        Push(result, Stack)

        return Pop(Stack)
```

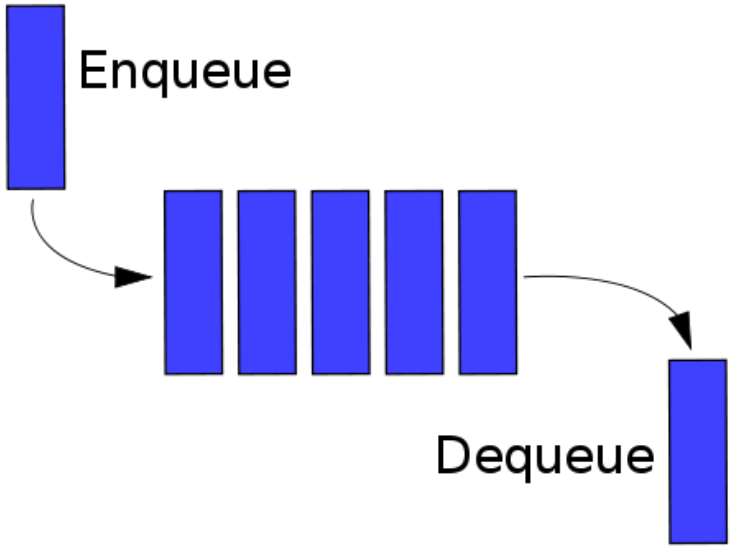
# Queues

- FIFO:
  - First In First Out
- The item that was inserted first is removed first.
- Main operations:
  - Add (enqueue)
  - Remove (dequeue)



# Queues (cont'd)

- Operations:
  1. Add (enqueue)
  2. Remove (dequeue)
  3. Front/Peek
  4. Is-Empty
  5. Length





# Queues in Python

| Queue Operation | Corresponding Python Op. |
| --- | --- |
| Add (enqueue) | L.append(item) |
| Remove (dequeue) | L.pop(0) |
| Front/Peek | L[0] |
| Is-Empty | L == [] |
| Length | len(L) |

# Implementing Queues in Python

```
def CreateQueue():
        """Creates an empty queue"""
        return []

def Enqueue(item, Queue):
        """Add item to the end of Queue"""
        Queue.append(item)

def Dequeue(Queue):
        """Remove and return the item at the front of the Queue"""
        return Queue.pop(0)

def IsEmpty(Queue):
        """Check whether the Queue is empty"""
        return Queue == []

def Front(Queue):
        """Return the value of the current front item without removing it"""
        return Queue[0]
```

# Queues: Formal Definition

$add(item, queue)$  $item \boxplus queue$

- $new() \rightarrow \varnothing$
- $front(\xi \boxplus \varnothing) \rightarrow \xi$
- $front(\xi \boxplus Q) \rightarrow front(Q)$
- $remove(\xi \boxplus \varnothing) \rightarrow \varnothing$
- $remove(\xi \boxplus Q) \rightarrow \xi \boxplus remove(Q)$
- $isempty(\varnothing) \rightarrow$TRUE
- $isempty(\xi \boxplus Q) \rightarrow$ FALSE



# Queues in Python (Example)

These two functions should run on two "threads" that share some memory together

```
CustomerQueue = CreateQueue()

def bank_queue():
        while True:
                if NewCustomerArrived() == True:
                        new_customer = GetCustomerInfo()
                        Enqueue(new_customer, CustomerQueue)

def serve_customers():
        while True:
                if CustomerQueue.IsEmpty() == False:
                        customer = Dequeue(CustomerQueue)
                        ServeCustomer(customer)
```