Design and Analysis of Algorithms

- Introduction
- Max in an Array and Insertion sort
- Loop Invariant
- Basic Data Structure

Yanlin Zhang & Wei Wang | DSAA 2043 Spring 2025

- Lecture Time: Tue + Thu, 1030 – 1150, W1-222
- Lab Session: Fri, 1030 – 1150, E1-228
- Instructor: Wei Wang
  - weiwcs AT ust.hk
  - http://wei-wang.net/
- TA
  - Liuchang Jing, ljing248 AT connect.hkust-gz.edu.cn
- Course materials
  - Web page: https://dbwangwei.github.io/DSAA2043/
  - Or Canvas

Introduction

Course Goals

- The design and analysis of algorithms
  - They usually appear together

- By taking this course, you will
  - Obtain a good understanding of various data structures and algorithms
  - Learn to think analytically about algorithms
  - Learn to design and apply algorithms to solve computational problems effectively
  - Learn to implement and evaluate algorithms and data structures

Contents (Tentative)

Week	Topic	Content (if any)
1	Course introduction and basic data structures	Basic data structures
2	Analysis of algorithms	Asymptotic Analysis
3	Sorting algorithms	
4	Advanced data structures	
5	Divide-and-conquer	Merge sort, Binary search, Integer multiplication, Master's theorem
6	Dynamic programming I	Knapsack Problem
7	Dynamic Programming II	Longest Common Subsequence, backtracking
8	Greedy and Midterm Exam	Scheduling, MST
9	Graph algorithm I	Graph representation, Graph traversal, Topological sorting, Cycle detection
10	Graph algorithm II	Strongly connect component, Single source and all pairwise shortest path
11	Advanced Graph algorithm	
12	NP	P/NP, NP-complete
13	Computational intractability	Reduction, NPC problems
14	Final exam	

Textbook and Materials

Textbook:

- Introduction to Algorithms. Cormen, Leiserson, Rivest, and Stein

Reference books:

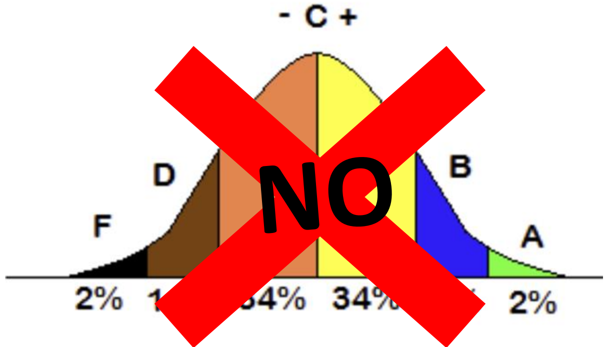
- Algorithm Design. Kleinberg and Tardos
- The Algorithm Design Manual. Steven Skiena

Online resources:

- MIT 6.006 - Introduction to Algorithms
- Stanford CS161 - Design and Analysis of Algorithms

Assessment and Grading (Tentative)

- Class Participation (5%)
- Lab Exercises (10%):
  work on lab exercises and submit by the deadline (each week)
- Individual Project (20%):
  a two-phase programming exercise
- Mid-term exam (25%):
  closed book, on computer, week 8
- Final exam (40%):
  closed book, written
- We assess student performance using criterion-referencing approach. In addition to the criterion written in course syllabus, you can estimate your performance from your course work score:
  - A level: [85, 100]
  - B level: [70, 85)
  - C level: [55, 70]
  - D level: [40, 55)
  - F level: [0, 40)



How to get the most out of this course

Preview, Participate, and Review matters!

- **Before class:**
  - Prepare for the lecture
- **During class:**
  - Class participation: ask any questions anytime
  - Engage with in-class questions and exercises
- **After class:**
  - Review contents timely and ask questions☺ → Don't wait until the day before exam ☹
  - Do exercises
- **Generative AI:**
  - Using Generative AI to prepare and review course content is allowed.
  - Don't use it (brainlessly) to solve exercise.
    - Learning requires generation by you (not AI)
    - Learning algorithm do require learning abstraction, in-depth thinking, and asking critical questions!

What Is an Algorithm

"An **algorithm** is any well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output** in a finite amount of time. An algorithm is thus **a sequence of computational steps that transform the input into the output**."

- CLRS

- Are they algorithms?
  - Problem: Do any two students in a class have the same birthday?
    Solution: Compare each student with others → Any better ideas? ✋
  - Problem: Do any two people in the world share the same birthday?
    Solution: keep asking random people about their birthdays, until you find a pair that matches

algorithm examples

Max in an Array and Insertion sort

Pseudo-Code

- **Pseudo-code:** A mixture of natural language and high-level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm.
- It is more structured than usual prose but less formal than a programming language
- Expressions
  - use standard mathematical symbols to describe numeric and boolean expressions
  - use ← for assignment ("=" in Python)
  - use = for equality relationship ("==" in Python)
- Method declarations
  - algorithm `name(param1,param2)`

Pseudo-Code

- Programming constructs
  - decision structures: **`if ... then ... [else ... ]`**
  - while-loops: **`while ... do`**
  - repeat-loops: **`repeat ... until ...`**
  - for-loop: **`for ... do`**
  - array indexing: **`A[i]`**, **`A[i:j]`**, **`A[i,j]`** or **`A[i][j]`**

    Subarray includes A[j]

    array index starts from 1

    2d array
- Methods
  - calls: object `method(args)`
  - returns: **`return`** `value`

Max in an Array

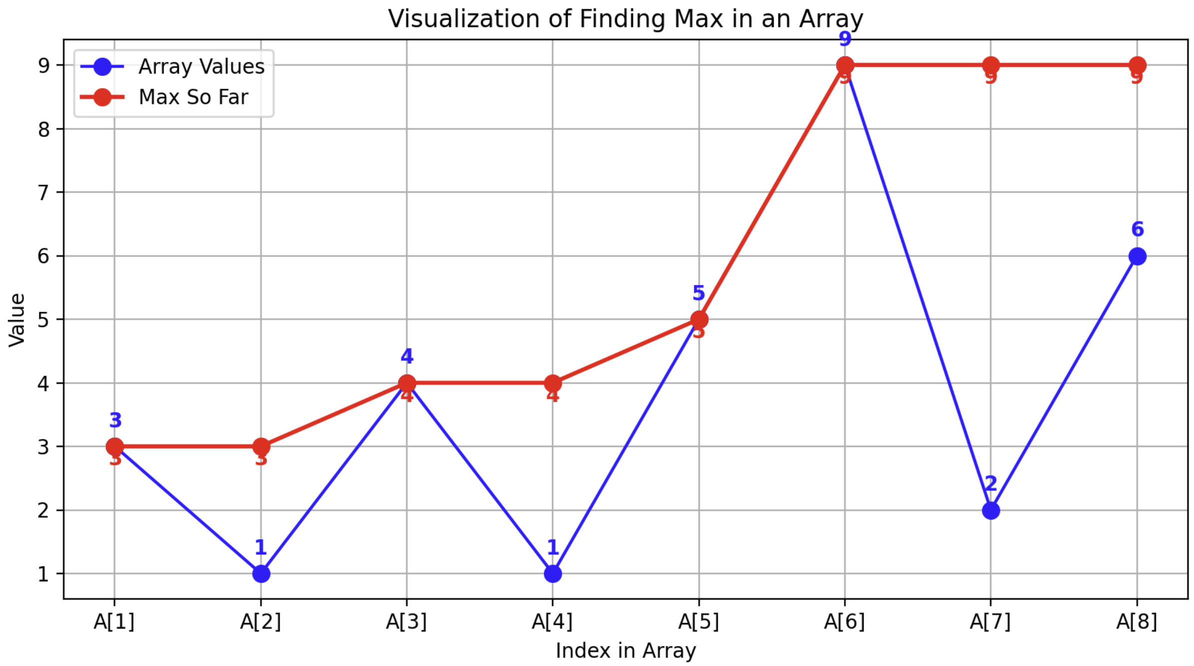
Algorithm findMax(A)

Input: An array A storing n values

Output: The maximum element in A

findMax pseudo-code

```
Algorithm FindMax(A)
Input: An array A with n elements
Output: The maximum value in A

max_so_far ← A[1]  // Initialize max with the first element
for i ← 2 to length(A) - 1 do
    if A[i] > max_so_far then
        max_so_far ← A[i]  // Update max if a larger value is found
    end if
end for
return max_so_far
```



Sorting an Array

Algorithm sort(A)

Input: An array A storing n values

Output: A permutation of A where elements are ordered in increasing sequence

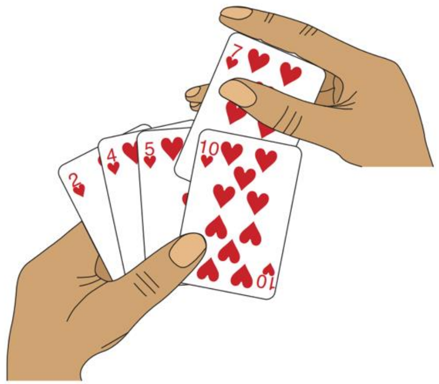
Sort

INPUT

sequence of numbers

$a_1, \quad a_2, \quad ...., \quad a_n$

2 5 4 10 7



OUTPUT

a permutation of the sequence of numbers

$b_1, b_2, \quad ...., \quad b_n$

2 4 5 7 10

Correctness (requirements for the output)

For any given input the algorithm halts with the output:

- $b_1 < b_2 < b_3 < .... < b_n$
- $b_1, b_2, b_3, ..., b_n$ is a permutation of $a_1, a_2, a_3, ..., a_n$

Running time

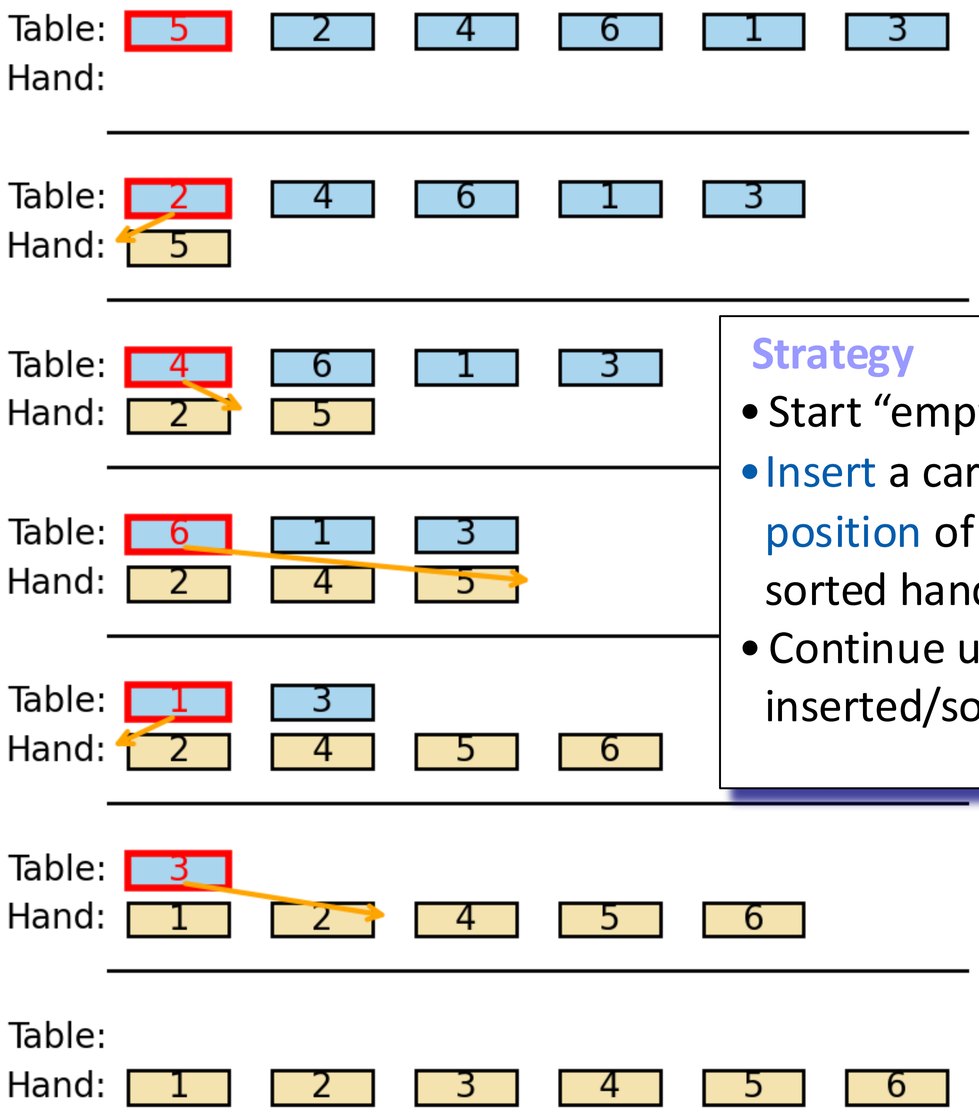
Depends on

- number of elements (n)
- how (partially) sorted they are
- algorithm

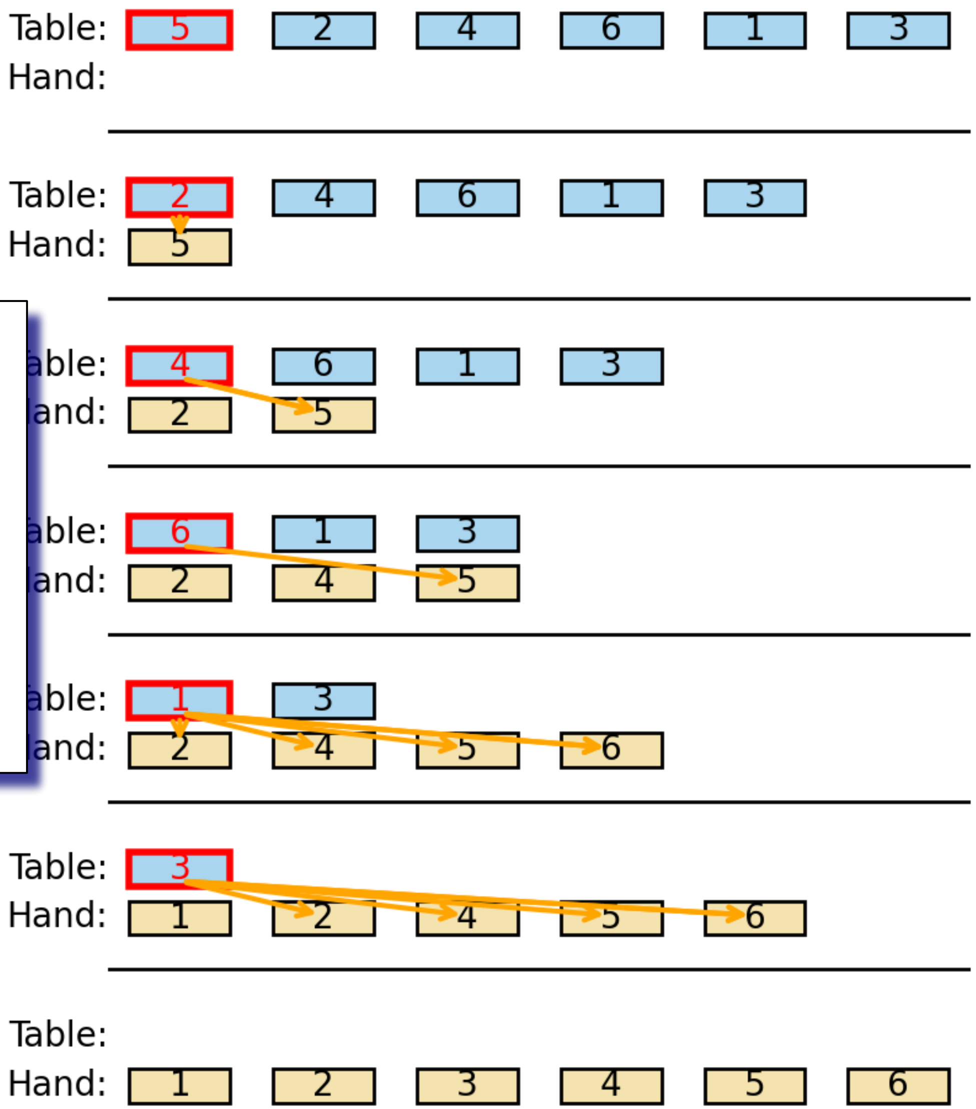
Picking and Placing Cards at Hand

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Poker-Style Insertion Sort (magic power)



Poker-Style Insertion Sort (no magic power)



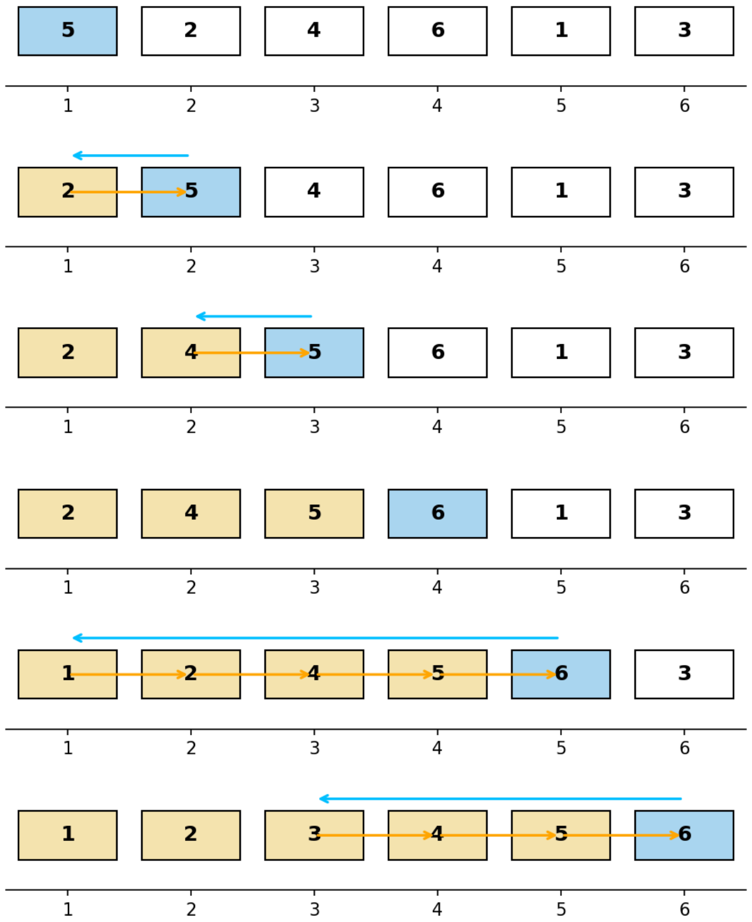
Strategy

- Start "empty handed"
- **Insert** a card in the **right position** of the already sorted hand
- Continue until all cards are inserted/sorted

Insertion Sort

```
Algorithm InsertionSort(A)
Input: An array A with n elements
Output: A sorted in non-decreasing order

for i ← 2 to length(A) do
    key ← A[i]  // Current element to be inserted
    j ← i - 1   // Start comparing from the previous element
    while j > 0 and A[j] > key do
        A[j + 1] ← A[j]  // Shift element to the right
        j ← j - 1
    end while
    A[j + 1] ← key  // Place key in the correct position
end for

return A  // Sorted array
```

Insertion Sort Step-by-Step



Is our algo. correct?

Loop Invariant

Proving the Correctness of FindMax

```
Algorithm FindMax(A)
Input: An array A with n elements
Output: The maximum value in A

1. max_so_far ← A[1]
2. for i ← 2 to length(A) - 1 do
3.     if A[i] > max_so_far then
4.         max_so_far ← A[i]
5.     end if
6. end for
7. return max_so_far
```

Proof of Correctness

Goal: Showing that max_so_far stores the maximum value of A.

1. Before the Loop Starts (Initialization)

- Before entering the loop, max_so_far = A[1]. Since we've only seen one element, this is correct.

2. During Each Iteration (Maintenance)

- If A[i] > max_so_far, we update max_so_far = A[i], ensuring it holds the maximum seen so far.
- Otherwise, max_so_far remains unchanged, which is still correct.

3. After the Loop Ends (Termination)

- At the end, max_so_far contains the maximum of all A[1] to A[n].

Loop Invariant

- A loop invariant is a property or condition that holds true before and after each iteration of a loop.
- Purpose:
  - To show that an algorithm maintains a specific condition throughout its execution.
  - To help prove that the algorithm works correctly (via initialization, maintenance, and termination).
- The way that we prove FindMax correct is Loop Invariant.
  - Loop invariant: max_so_far holds the maximum value among A[1:i].

Structure of a Loop Invariant Proof

- Initialization: Show that the invariant holds **before** the first iteration (base case).
- Maintenance: Assuming the invariant holds at the beginning of any iteration, **prove that it still holds** after executing the loop body.
- Termination: When the loop terminates, the invariant (plus the loop's exit condition) gives a useful property that helps prove the algorithm's correctness.

FYI. The three steps is introduced in CLRS. In other books, you may find Establishment (i.e. Initialization), Preservation (i.e. Maintenance), Postcondition and Termination: Postcondition ensures the final goal is achieved if the loop stops; Termination guarantees that the loop will stop.

Proving Insertion-sort

```
Algorithm InsertionSort(A)
Input: An array A with n elements
Output: A sorted in non-decreasing
        order

for i ← 2 to length(A) do
    key ← A[i]
    j ← i - 1
    while j > 0 and A[j] > key do
        A[j + 1] ← A[j]
        j ← j - 1
    end while
    A[j + 1] ← key
end for

return A
```

Proof of Correctness

Loop Invariant: A[1:i-1] is a sorted list of elements in the original A[1:i-1]

1. Initialization (Note: i ← 2 is not part of the loop)

- When i=2, A[1:1] contains one value → sorted.

2. Maintenance

- Within the loop-body, A[1:i-1] is assumed to be sorted. We move A[i-1], A[i-2], … toward the right, until we find a position for A[i]. Once A[i] is inserted, A[1:i] remains sorted.

3. Termination

- As i goes from 2 to length(A), and the loop body does not modify i. The loop will terminate. By termination, i=length(A) means A[1:length(A)] is sorted.

Discussion: a Loop Invariant for Linear Search

香港科技大学(广州) THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

```
Algorithm LinearSearch(A, x)
Input: An array A with n elements, target value x
Output: Index of x in A, or -1 if not found

loc = -1
for i ← 1 to length(A) do
    if A[i] == x then
        loc = i  // Found x at index i
    end if
end for
return val
```

Proof of Correctness

Loop Invariant: What is true and related to this task?

1. Initialization: What is true about A[1:i] before the loop starts?

2. Maintenance: How does each iteration preserve the correctness of the search?

3. Termination: When the loop exits, why can we be sure the correct index or -1 is returned?

Using the Loop Invariant Proof Structure, prove that LinearSearch(A, x) is correct.

Discussion: a Loop Invariant for Linear Search

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

```
Algorithm LinearSearch(A, x)
Input: An array A with n elements, target value x
Output: Index of x in A, or -1 if not found

loc = -1
for i ← 1 to length(A) do
    if A[i] == x then
        loc = i  // Found x at index i
    end if
end for
return loc
```

Proof of Correctness

Loop Invariant:

1. Initialization:

2. Maintenance:

3. Termination:

Using the Loop Invariant Proof Structure, prove that LinearSearch(A, x) is correct.

Discussion: a Loop Invariant for Linear Search

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

```
Algorithm LinearSearch2(A, x)
Input: An array A with n elements, target value x
Output: Index of x in A, or -1 if not found

for i ← 1 to length(A) do
    if A[i] == x then
        return i  // Found x at index i
    end if
end for
return -1  // x is not in A
```

Proof of Correctness

Loop Invariant:

1. Initialization:

2. Maintenance:

3. Termination:

Using the Loop Invariant Proof Structure, prove that LinearSearch2(A, x) is correct.

Basic Data Structures

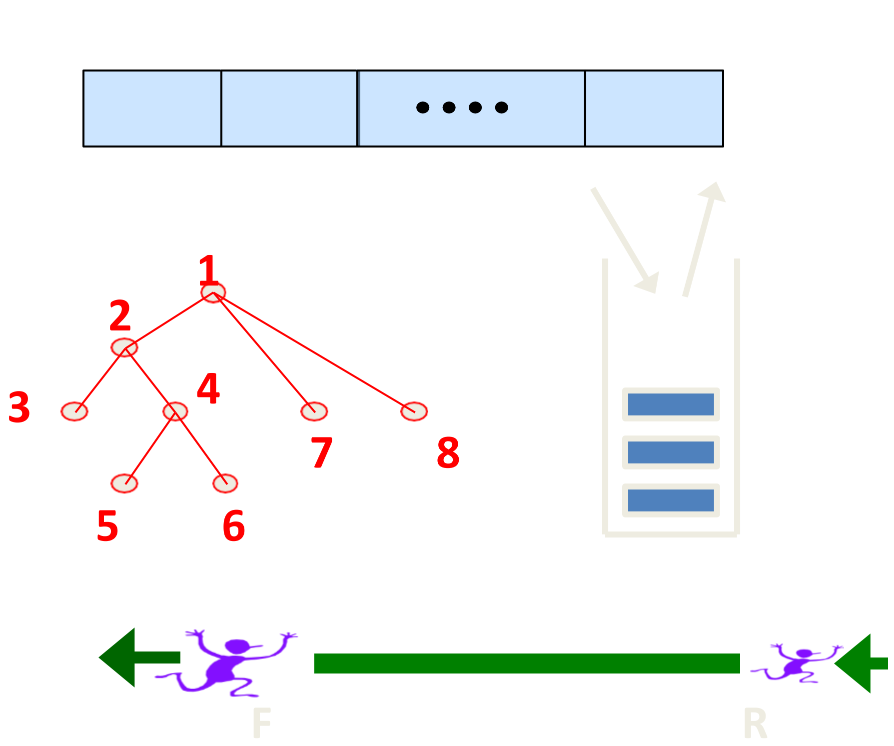
What Is a Data Structure

"A **data structure** is **a way to store and organize data** in order to facilitate **access** and **modifications**. Using the appropriate data structure or structures is an important part of algorithm design. *No single data structure works well for all purposes*, and so you should know the strengths and limitations of several of them."

- CLRS

Basic Data Structures

- Arrays
- Lists
- Stacks
- Queues
- Trees



Array

- An array is a linear data structure that stores a fixed-size collection of elements of the same type in contiguous memory locations.
  - **Fixed Size** – Its size is declared at initialization and cannot be changed.
  - **Contiguous Memory Allocation** – Elements are stored sequentially in memory, making access fast (O(1) for direct access by index).
  - **Homogeneous** – All elements in an array must be of the same data type.
  - **Index-Based Access** – Elements are accessed using an index, starting from 1.

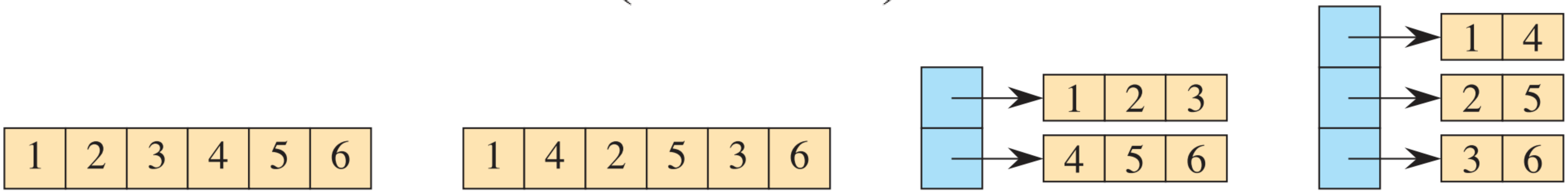
Array:	**10**	**25**	**30**	**40**	**50**
Index:	1	2	3	4	5

Array-based Matrix

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- We can use an array or arrays to store a matrix.

$$M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



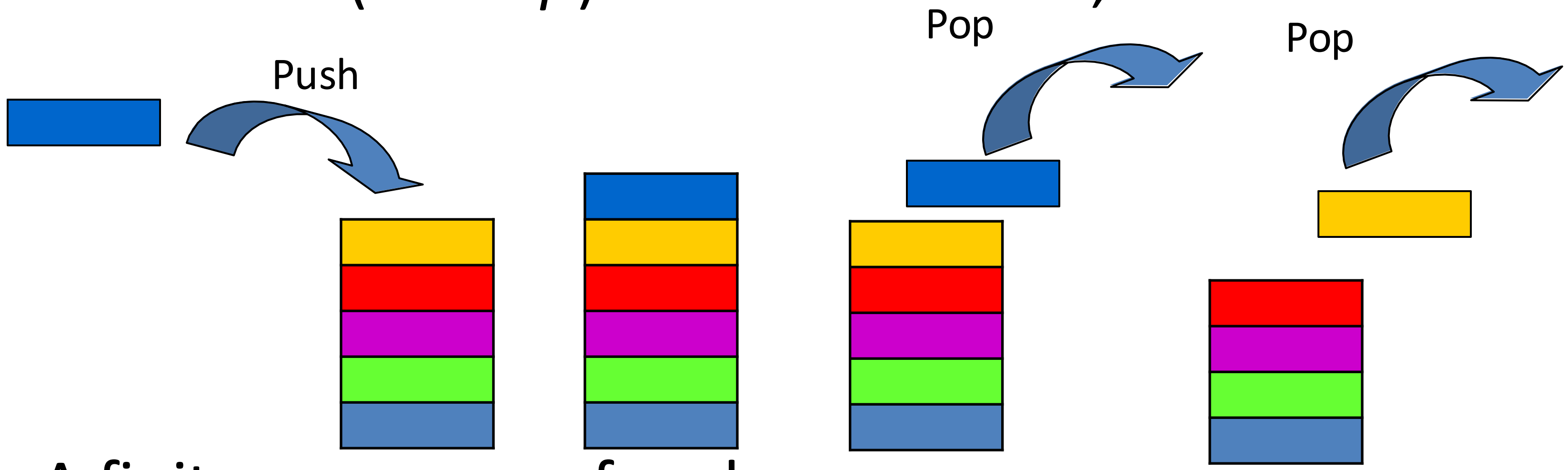
Row-major ordering | Column-major ordering | **Row-major ordering** | Column-major ordering

Blue: Array of pointers to arrays

M[1] = [1, 2, 3]
M[1][2] = 2 } Python's 2d array (list of list)
M[1, 2] = 2 Numpy's 2d array

Stack

- **def.** A list for which Insert (push) and Delete (pop) are allowed only at one end of the list (the *top*) → **LIFO** – Last in, First out



- **Objects:** A finite sequence of nodes
- **Operations:**
  - Push: Insert element at top
  - Pop: Remove and return top element
- **Applications:** undo operations

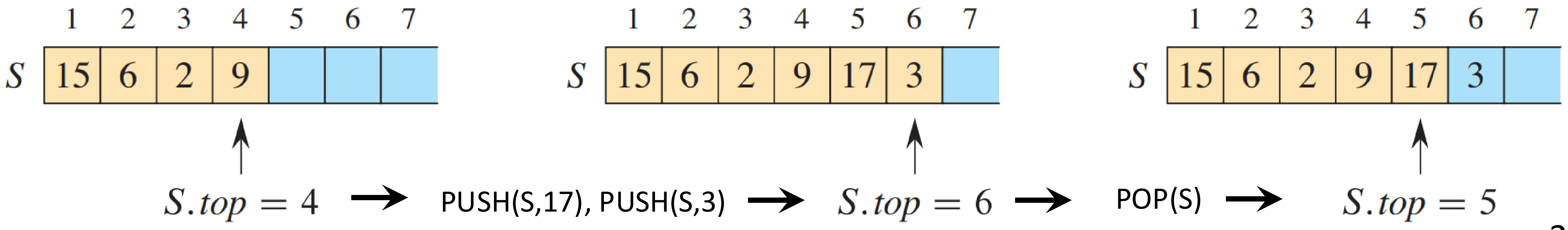
Exercise: Stack

- Describe the output of the following series of stack operations
  - Push(8)
  - Push(3)
  - Pop()
  - Push(2)
  - Push(5)
  - Pop()
  - Pop()
  - Push(9)
  - Push(1)

Array-based Stack


```
INITIALIZE(S, size)
    S.size = size
    S.array = new array of size S.size
    S.top = 0  # Stack starts empty

STACK-EMPTY(S)
    if S.top == 0
        return TRUE
    else return FALSE
```

```
PUSH(S, x)
    if S.top == S.size
        error "overflow"
    else S.top = S.top + 1
        S[S.top] = x

POP(S)
    if STACK-EMPTY(S)
        error "underflow"
    else S.top = S.top - 1
        return S[S.top+1]
```



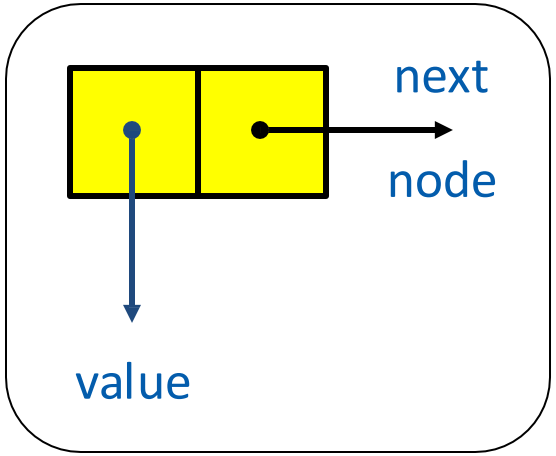
Growable Array-Based Stack

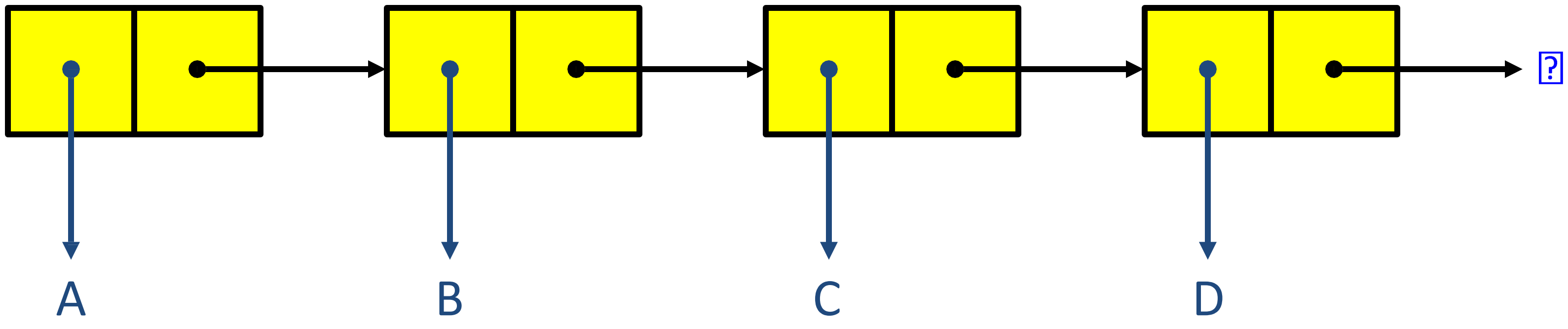

```
INITIALIZE(S, size)
S.size = size
S.array = new array of size S.size
S.top = 0  # Stack starts empty

STACK-EMPTY(S)
if S.top == 0
    return TRUE
else return FALSE

GROW(S)
new_size = 2 * S.size   # Double the size
new_array = new array of size new_size
for i = 1 to S.size:
    new_array[i] = S.array[i]  # Copy elements
S.array = new_array
S.size = new_size
```

```
PUSH(S, x)
if S.top == S.size:
    GROW(S)  # Expand the array
S.top = S.top + 1
S.array[S.top] = x

POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top+1]
```

Singly Linked List

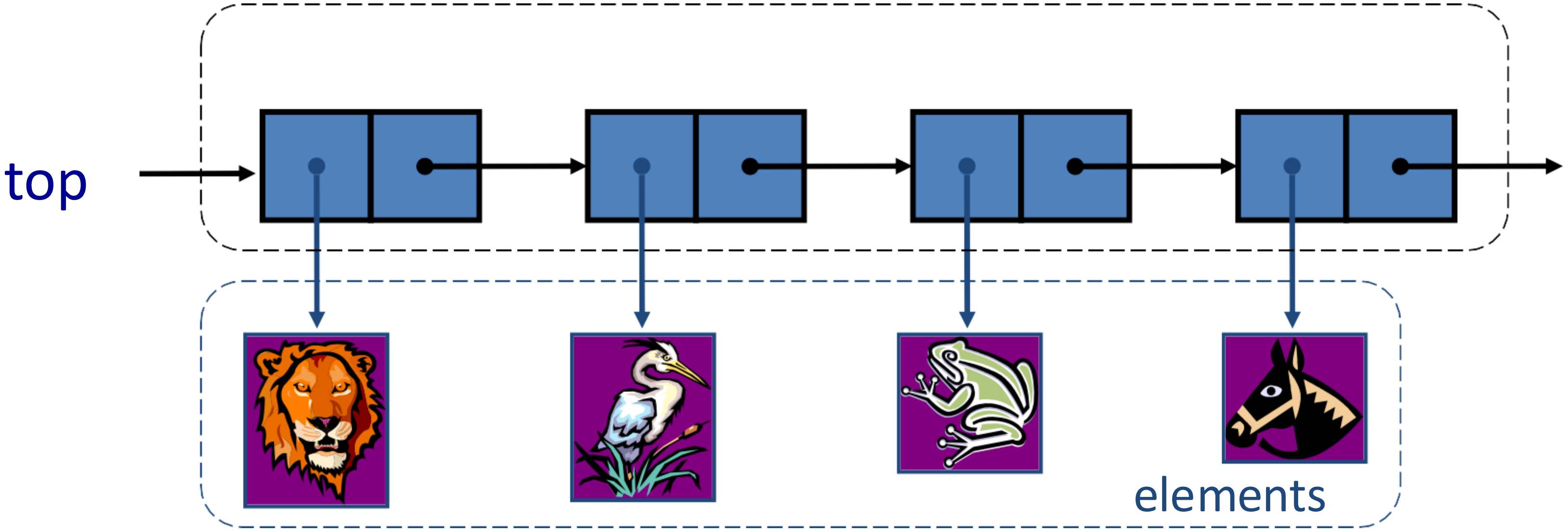
- A singly linked list is a dynamic data structure consisting of a sequence of nodes
- Each node contains two parts:
  - Data – Stores the actual value.
  - Next Pointer – Points to the next node in the list.
  - The last node's next pointer is NULL, indicating the end of the list.





Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $\boldsymbol{O}(\boldsymbol{n})$ and each operation of the Stack takes $\boldsymbol{O}(1)$ time



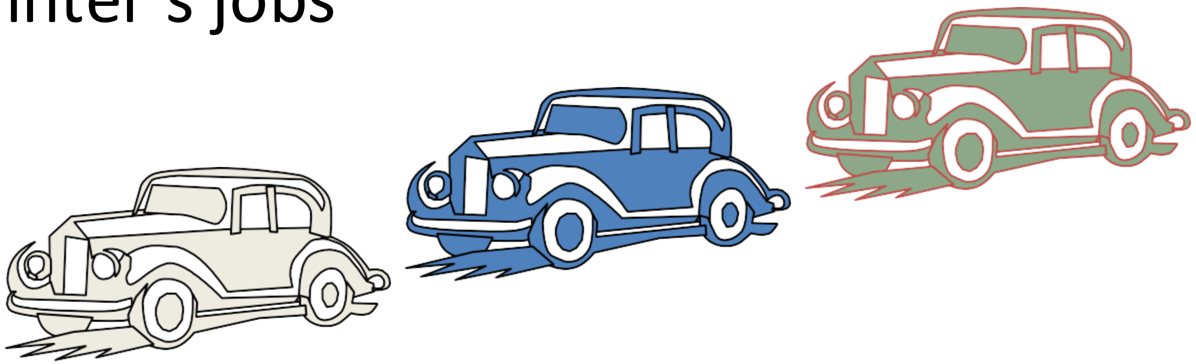
Stack Summary

Implementation	Push	Pop	isEmpty	Top	Space
Fixed-size Array	O(1)	O(1)	O(1)	O(1)	O(n) (Extra capacity overhead)
Growable Array	O(1) amortized/O(n) worst	O(1)	O(1)	O(1)	O(n) (Extra capacity overhead)
Linked List	O(1)	O(1)	O(1)	O(1)	O(n) (Extra pointer overhead)

- Fixed-Size Array: Best for known, small-sized stacks but has wasted memory when underutilized.
- Growable Array: Balances flexibility and speed, but resizing incurs occasional O(n) cost.
- Linked List: No need to predefine size, but higher space overhead (extra pointers for each node).

Queues

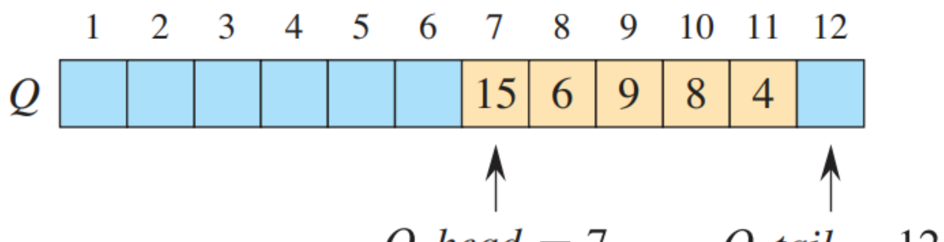
- **def.:** A Queue is a linear data structure that follows the FIFO (First In, First Out) principle. This means that elements are added at the rear (enqueue) and removed from the front (dequeue).
- **Operations:**
  - Enqueue(x) → Adds x to the rear.
  - Dequeue() → Removes and returns the front element.
- **Applications:** printer's jobs



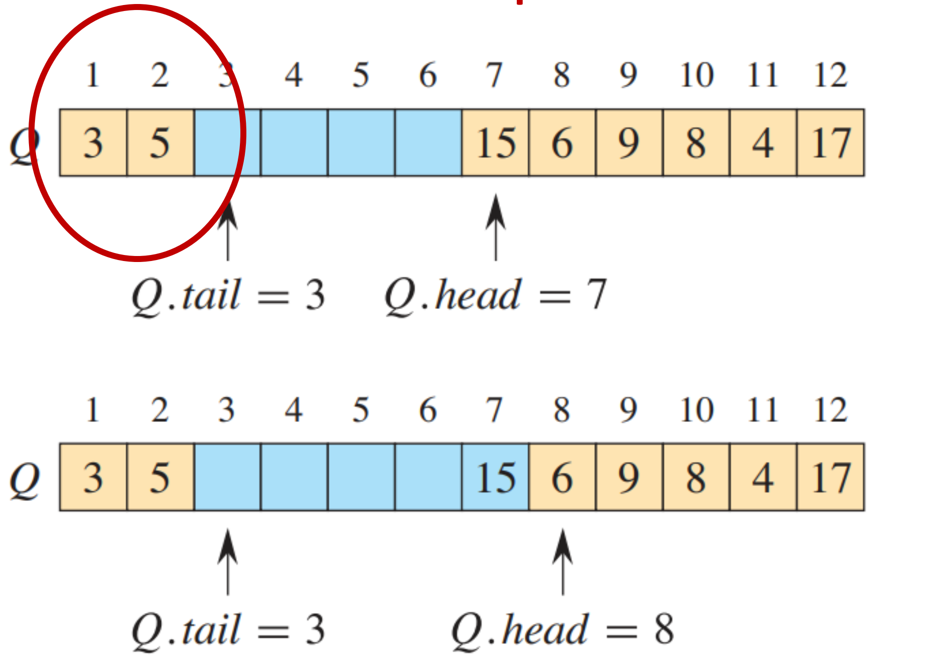
Exercise: Queues

- Describe the output of the following series of queue operations
  - enqueue(8)
  - enqueue(3)
  - dequeue()
  - enqueue(2)
  - enqueue(5)
  - dequeue()
  - dequeue()
  - enqueue(9)
  - enqueue(1)

Circular Array based Queue

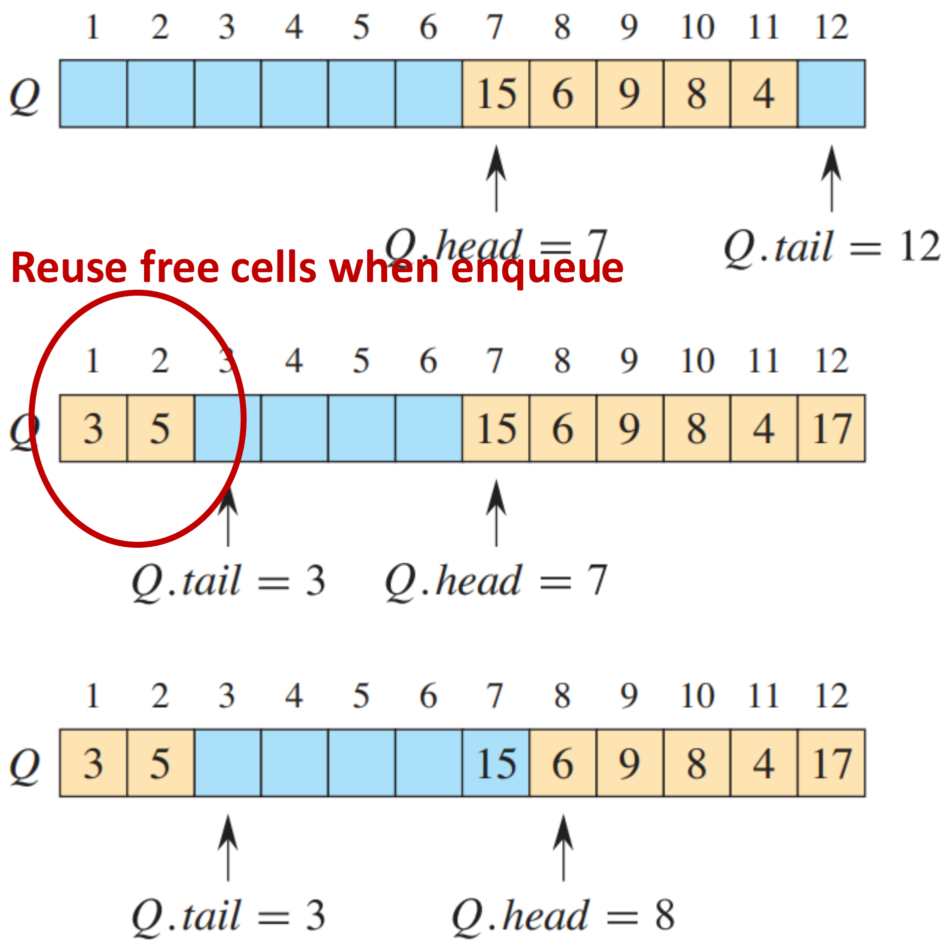


Reuse free cells when enqueue




```
INITIALIZE(Q, size)
   Q.size = size
   Q.array = new array of size Q.size
   Q.head = -1  # Indicates an empty queue
   Q.tail = -1

IS-EMPTY(Q)
   return Q.head == -1

IS-FULL(Q)
   return (Q.tail + 1) % Q.size == Q.head

ENQUEUE(Q, x)
   if IS-FULL(Q):
       error "Queue is full"
   else if IS-EMPTY(Q):
       Q.head = Q.tail = 0  # First element in queue
   else:
       Q.tail = (Q.tail + 1) % Q.size # wrap around
   Q.array[Q.tail] = x
```

Circular Array based Queue (cont.)



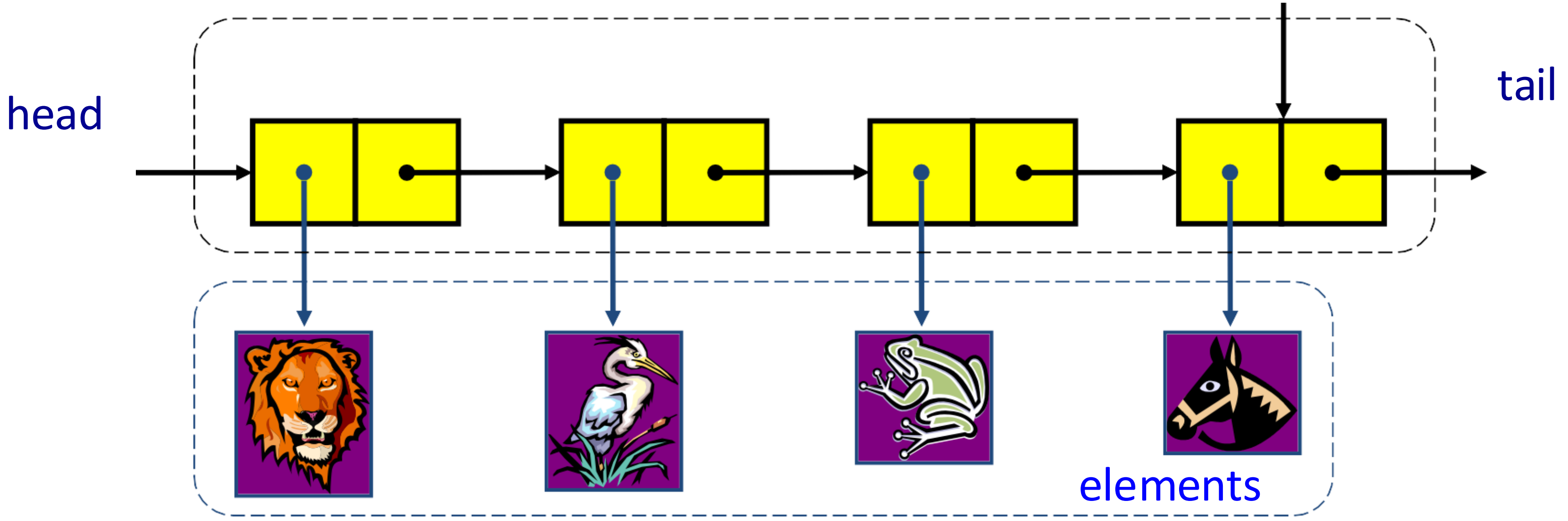

```
DEQUEUE(Q)
if IS-EMPTY(Q):
    error "Queue is empty"
else:
    temp = Q.array[Q.head] #Store the head element
    if Q.head == Q.tail: #Only 1 element was present
        Q.head = Q.tail = -1  # Reset queue
    else:
        Q.head = (Q.head + 1) % Q.size #wrap around
    return temp

FRONT(Q)
if IS-EMPTY(Q):
    error "Queue is empty"
return Q.array[Q.head]
```

Growable Array-based Queue

- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with doubled sized
- Similar to what we did for an array-based stack
- The enqueue operation has amortized running time $O(1)$

Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the head of the list
  - The rear element is stored at the tail of the list
- The space used is $\boldsymbol{O}(\boldsymbol{n})$ and each operation of the Queue takes $\boldsymbol{O}(1)$ time
- the queue is NEVER full



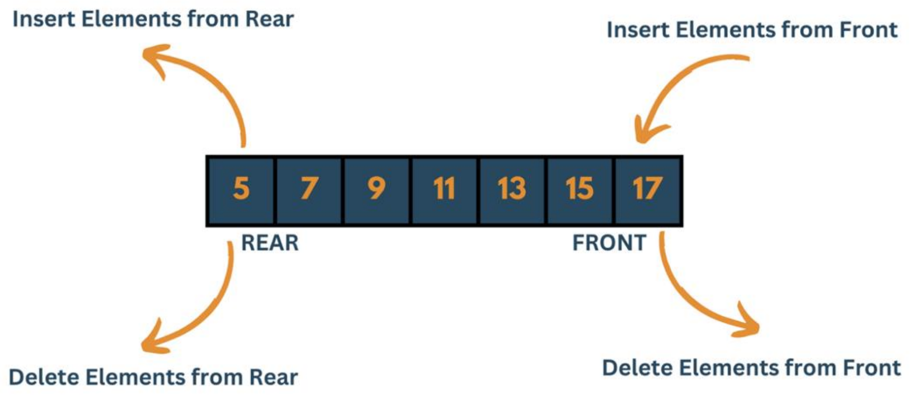
Queues Summary

Implementation	Enqueue	Dequeue	isEmpty	Head	Space
Circular Array	O(1)	O(1)	O(1)	O(1)	O(n) (Fixed size, extra capacity overhead)
Growable Array	O(1) amortized/O(n) worst	O(1)	O(1)	O(1)	O(n) (Extra capacity overhead)
Linked List	O(1)	O(1)	O(1)	O(1)	O(n) (Extra pointer overhead)

- Circular Array: Best for known, small-sized queues but has wasted memory when underutilized.
- Growable Array: Balances flexibility and speed, but resizing incurs occasional O(n) cost.
- Linked List: No need to predefine size, but higher space overhead (extra pointers for each node).

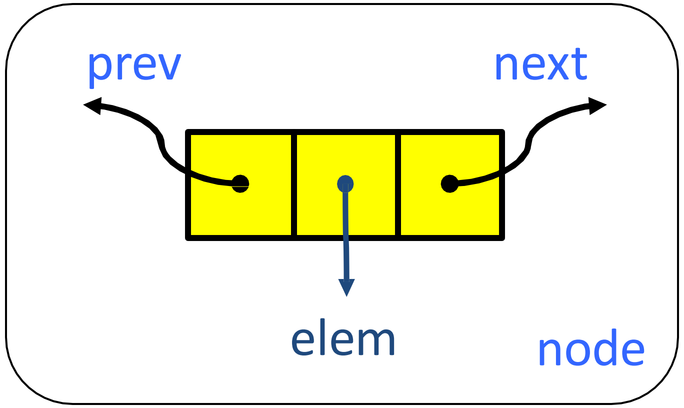
Deque: double-ended queue

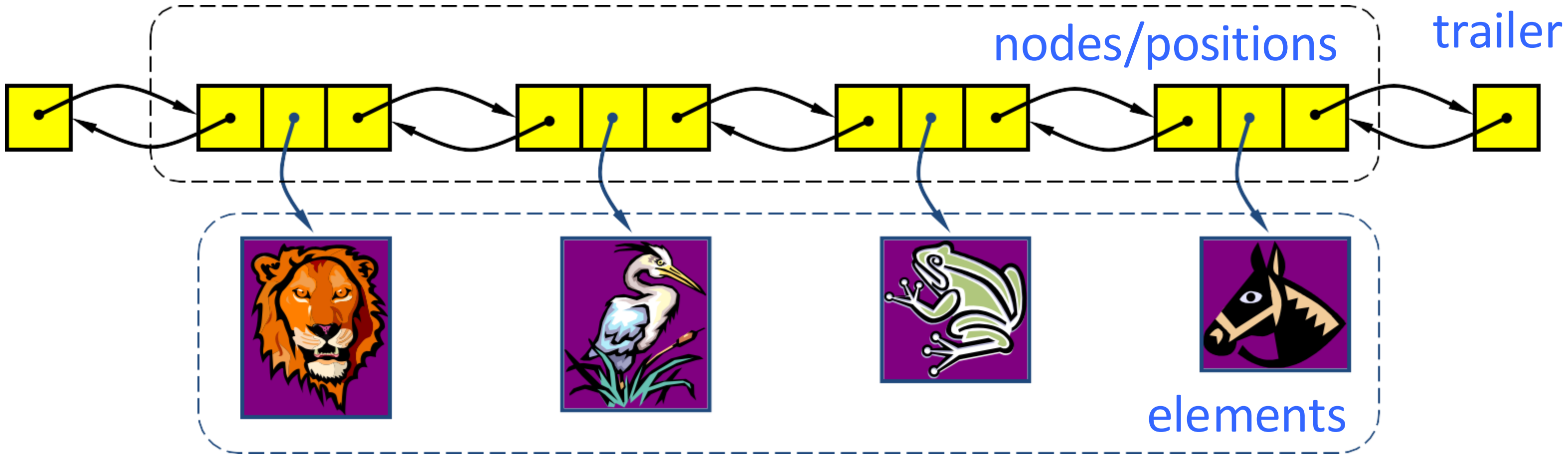
- **Def.** A Deque (Pronounced ‘deck’) is a linear data structure that allows insertion and deletion from both ends (front and rear).
- Supports both FIFO and LIFO operations.
  - Insert and delete from both front and rear.
- More flexible than a normal queue.
  - Efficient O(1) insertion & deletion at both ends



Doubly Linked List

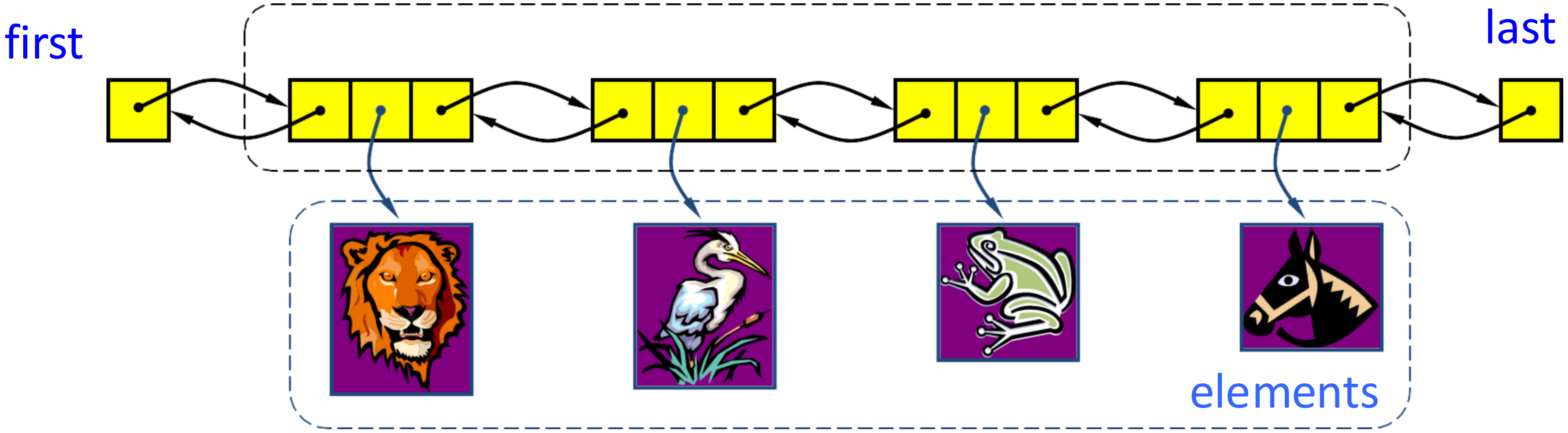
- A doubly linked list provides a natural implementation of the Deque
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





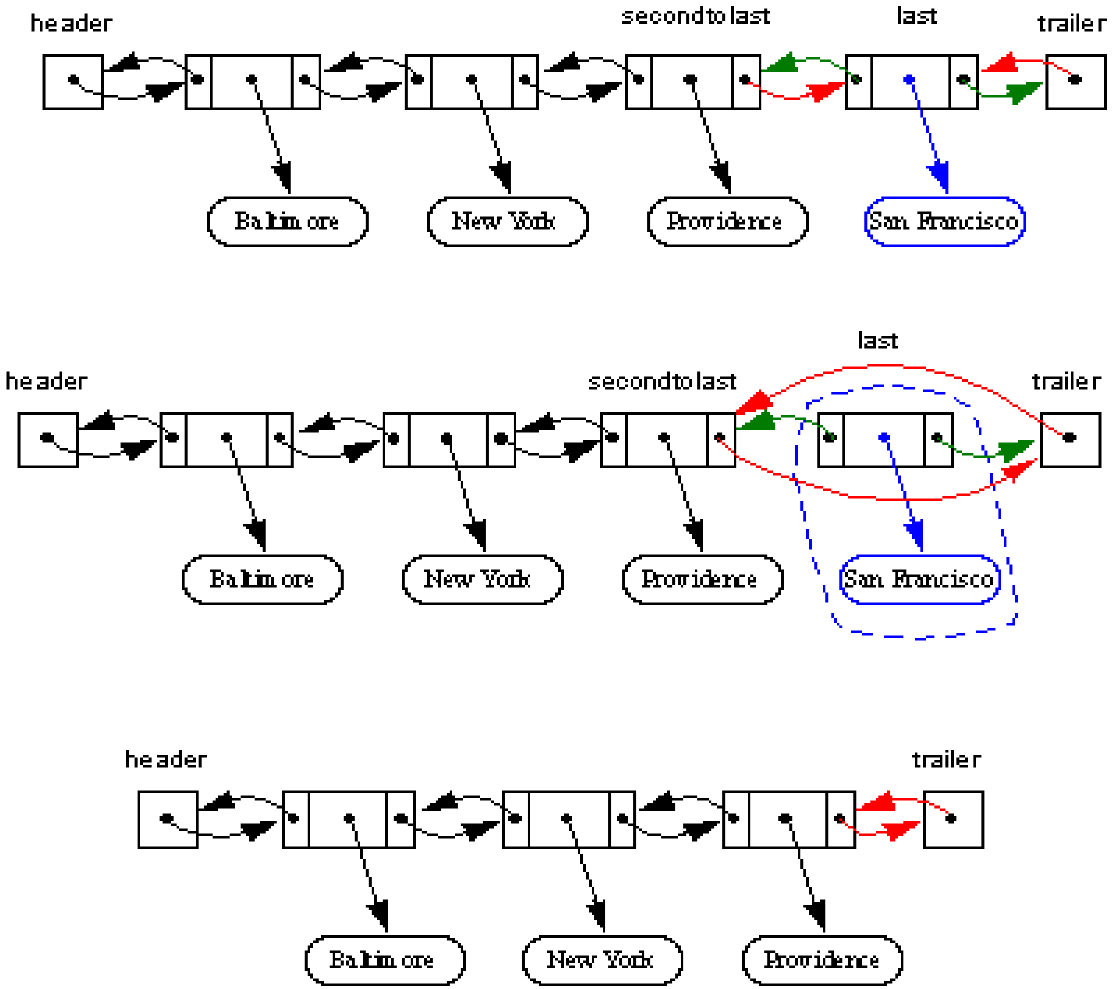
Deque with a Doubly Linked List

- We can implement a deque with a doubly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Deque ADT takes $O(1)$ time



Implementing Deques with Doubly Linked Lists

Here's a visualization of the code for removeLast().



Abstract Data Type (ADT)

- ADT: A mathematical definition of objects, with operations defined on them → an ADT specifies what a data structure should do, but not how it does it.
- Key Characteristics of ADTs:
  - Encapsulation: ADTs hide internal representations, exposing necessary operations.
  - Implementation Independent: ADTs can be implemented using different underlying structures.
  - Operations-Oriented: ADTs define operations, not implementations.
- ADT Example - List:
  - collection of elements.
  - Common operations: insert(index, value), delete(index), get(index), size().
  - Implementation: Arrays, Linked Lists.

Implementing Stacks and Queues with Deque

Stacks ADT with Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

Queues ADT with Deques:

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()