

STACKS AND QUEUES

**SKILLS
FOR LIFE**

SKILLS BOOTCAMPS



Department
for Education

Foundational Sessions Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
(FBV: Mutual Respect.)
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes.
You can submit these questions here:

[SE Open Class Questions](#) or [DS Open Class Questions](#)

Foundational Sessions Housekeeping cont.

- For all **non-academic questions**, please submit a query: www.hyperiondev.com/support
- Report a **safeguarding** incident: www.hyperiondev.com/safeguardreporting
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

Reminders!

Guided Learning Hours

By now, ideally you should have 7 GLHs per week accrued. Remember to attend any and all sessions for support, and to ensure you reach 112 GLHs by the close of your Skills Bootcamp.

Progression Criteria

✓ **Criterion 1: Initial Requirements**

- Complete 15 hours of Guided Learning Hours and the first four tasks within two weeks.

✓ **Criterion 2: Mid-Course Progress**

- Software Engineering: Finish 14 tasks by week 8.
- Data Science: Finish 13 tasks by week 8.

✓ **Criterion 3: Post-Course Progress**

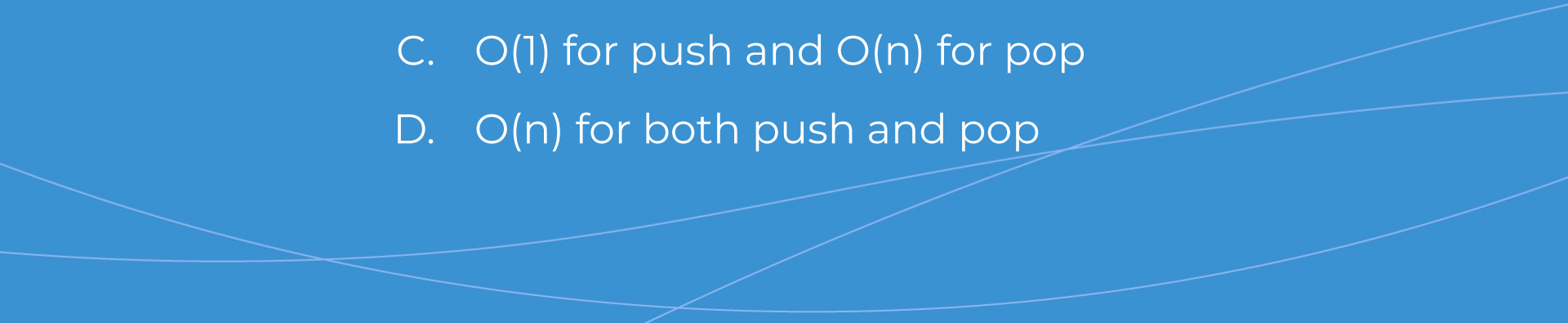
- Complete all mandatory tasks by 24th March 2024.
- Record an Invitation to Interview within 4 weeks of course completion, or by 30th March 2024.
- Achieve 112 GLH by 24th March 2024.


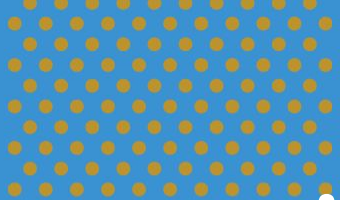
✓ **Criterion 4: Employability**

- Record a Final Job Outcome within 12 weeks of graduation, or by 23rd September 2024.




What is the time complexity of the 'push' and 'pop' operations in a typical stack implementation?

- A. $O(1)$ for both push and pop
 - B. $O(n)$ for push and $O(1)$ for pop
 - C. $O(1)$ for push and $O(n)$ for pop
 - D. $O(n)$ for both push and pop
- 

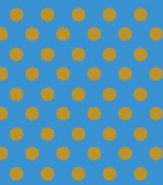


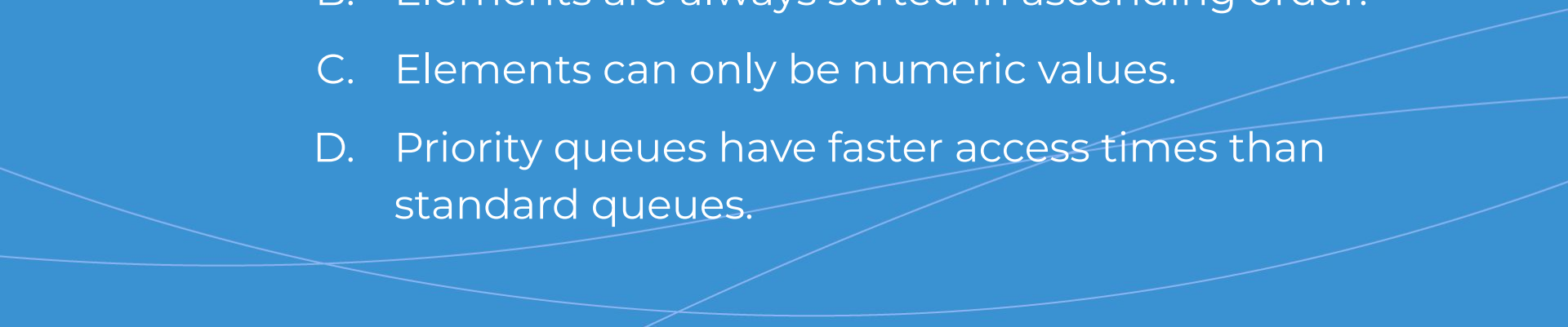
Which of the following data structures is typically used to implement a simple queue?

- A. Array
 - B. Linked List
 - C. Hash Table
 - D. Both Arrays and Linked Lists
- 



What is a distinguishing feature of a priority queue compared to a standard queue?



- A. Elements are removed based on their priority rather than their order in the queue.
 - B. Elements are always sorted in ascending order.
 - C. Elements can only be numeric values.
 - D. Priority queues have faster access times than standard queues.
- 

Recap of Linear Data Structures



Singly Linked List

A singly linked list is a collection of nodes, where each node contains data and a reference to the next node.

- Ideal for **dynamic data storage** where the size of the **data set isn't fixed**.
- A node has two attributes: **data** (the value it holds) and **next** (reference to the next node).

```
class Node:
    def __init__(self, data):
        self.data = data # Store the data
        self.next = None # Initialize the next reference to None
```

Singly Linked List Implementation

- **Initialising** a singly linked list:

```
class SinglyLinkedList:  
    def __init__(self):  
        self.head = None # Initialize the head of the list to None
```

- **Appending** a new element to a singly linked list:

```
# Method to add a new element at the end of the list  
def append(self, data):  
    new_node = Node(data) # Create a new node with the given data  
    if self.head is None:  
        self.head = new_node # If the list is empty, make the new node the head  
        return  
    # If the list is not empty, traverse to the end of the list  
    last_node = self.head  
    while last_node.next:  
        last_node = last_node.next  
    last_node.next = new_node # Link the last node to the new node
```

Singly Linked List Implementation

- **Prepending** a new element to a singly linked list:

```
# Method to add a new element at the beginning of the list
def prepend(self, data):
    new_node = Node(data) # Create a new node with the given data
    new_node.next = self.head # Link the new node to the current head
    self.head = new_node # Make the new node the new head of the list
```

- **Traversing** through elements in a singly linked list:

```
# Method to print all elements in the list
def print_list(self):
    cur_node = self.head
    while cur_node:
        print(cur_node.data, end=" -> ") # Print the data of each node
        cur_node = cur_node.next
    print("None") # Indicate the end of the list
```

Arrays

Arrays are a fixed size data structure for storing elements in sequential memory locations.

- Common operations include **accessing elements**, **updating values**, and **iterating over elements**.

```
# Example of array operations in numpy
import numpy as np
arr = np.array([1, 2, 3, 4])
# Access
print(arr[0]) # Output: 1
# Insertion
arr = np.append(arr, 5)
print(arr) # Output: [1 2 3 4 5]
```

```
# Deletion
arr = np.delete(arr, -1)
print(arr) # Output: [1 2 3 4]
# Update
arr[0] = 0
arr[1] = 1
arr[2] = 2
print(arr) # Output: [0 1 2 4]
# Iterate
for i in arr:
    print(i, end=" ") # Output: 0 1 2 4
```


Numpy Array Operations

- Operations include **mathematical computations**, **reshaping**, **slicing**, and more. Numpy arrays also support **vectorized operations**, making calculations faster.

```
import numpy as np

# Creating a Numpy array
np_arr = np.array([1, 2, 3, 4])

# Mathematical Computations
# Vectorized addition
addition = np_arr + np_arr
print(addition) # Output: [2, 4, 6, 8]
```

```
# Reshaping
reshaped = np_arr.reshape((2, 2))
print(reshaped) # Output: [[1, 2], [3, 4]]

# Slicing
sliced = np_arr[1:3]
print(sliced) # Output: [2, 3]
```

Performance Comparison

Numpy Array

- **Access:** $O(1)$
- **Append:** $O(n)$ - when creating new array
- **Delete:** $O(n)$
- **Iteration:** $O(n)$, but faster due to contiguous memory and cache efficiency
- **Vectorized Operations:** $O(n)$

Python List

- **Access:** $O(1)$
- **Append:** $O(1)$
- **Delete:** $O(n)$
- **Iteration:** $O(n)$, but slower. Especially on large data sets.
- **Vectorized Operations:** Not supported.

Stacks and Queues Topics

1. Introduction to Stacks and Queues
2. Implementing Stacks
3. Implementing Queues
4. Implementing Priority Queues

Task Management

Consider a software development team working on multiple projects at the same time. Each developer is assigned tasks from each project and these tasks need to be stored somewhere.

- For one group of projects, tasks need to be assigned in the order that they were submitted.
- Another group of projects require that the newest tasks be assigned first.
- The other group of projects have prioritised tasks and tasks with the highest priority need to be assigned first.

Task Management

- How can we use different data structures to organise the different lists of tasks in the most efficient way?
- How do we efficiently organise data that will be removed from the structure in a particular order?
- How do we guarantee that data will be removed and added to the structure in the correct manner?
- How can we evaluate the efficiency of these data structures?

Example: Organising Tasks

- **Project A:** This project has been released for a while and users may suggest different features to be added to the project.
 - *These suggestions can be added to a “queue”. Whichever suggestion was made first would be added to the project first.*
- **Project B:** This project is in the process of being developed and suggestions of the most useful features are made by the developers throughout the development of the project.
 - *Features can be added to a “stack”. Whichever feature was most recently suggested would be added to the project first.*
- **Project C:** This project has just been released and a lot of users are experiencing bugs of varying severity.
 - *Issues can be added to a “priority queue” organised by the severity of the bug. Issues that are the most severe would have the highest priority.*

Stacks and Queues

Types of linear data structures that allow for storage and retrieval of data based on a specific method of ordering.

- Data structures allow us to **organise storage** so that data can be accessed **faster and more efficiently**.
- Stacks and queues are simple, easy to implement and widely applicable.
- Each has defined methods of **ordering** and **operations** for adding and removing elements to and from the structure.
- Can be implemented using an Array, Linked List or the deque or queue modules in Python.

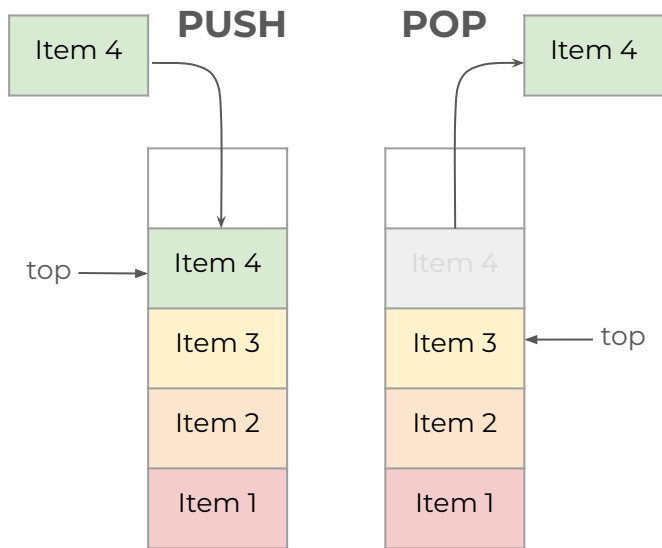
Stacks

Method of Ordering

- **LIFO:** Last element added to the stack is the first to be removed
- Elements are added on **top** of one another in a “stack”
- A pointer points to the element at the **top** of the stack

Operations

- **Push:** Adds an element to the top of the stack
- **Pop:** Removes the element from the top of the stack



Stacks: Array Implementation

```
# Simple stack class with the push and pop functions defined
class Stack:
    # Initialise the stack by creating an array of fixed size
    # and a top pointer
    def __init__(self, max):
        self.max_size = max
        self.stack = [None] * max
        self.top = -1
```

Note: This implementation is the most efficient. Be careful when attempting to implement a stack using the built-in `insert()` and `pop()` methods since it's slower to pop or insert an element at any other position besides the end of the list $[O(n)]$.

Stacks: Array Implementation

```
# Push an element to the stack
# Display a stack overflow error if the stack is full
def push(self, value):
    if self.top == self.max_size-1:
        print("Error: Stack Overflow!")
        return

    self.top += 1
    self.stack[self.top] = value
```

Note: This implementation is the most efficient. Be careful when attempting to implement a stack using the built-in `insert()` and `pop()` methods since its slower to pop or insert an element at any other position besides the end of the list $[O(n)]$.

Stacks: Array Implementation

```
# Pop an element from the stack
# Display a stack underflow error if the stack is empty
def pop(self):
    if self.top == -1:
        print("Error: Stack Underflow!")
        return

    removed = self.stack[self.top]
    self.top -= 1
    return removed
```

Note: This implementation is the most efficient. Be careful when attempting to implement a stack using the built-in `insert()` and `pop()` methods since its slower to pop or insert an element at any other position besides the end of the list $[O(n)]$.

Stacks

Complexity Analysis

- **Push**
 - **Space: $O(1)$**
No extra space is used
 - **Time: $O(1)$**
A single memory allocation done in constant time
- **Pop**
 - **Space: $O(1)$**
No extra space is used
 - **Time: $O(1)$**
Pointer is decremented by 1

Common Uses

- Function and Recursive Calls
- Undo and Redo Mechanisms
- “Most recently used” features
- Backtracking algorithms
- Expression evaluations and syntax parsing

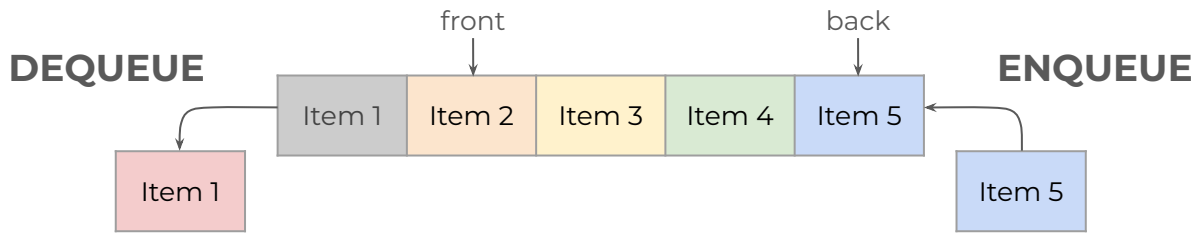
Queues

Method of Ordering

- **FIFO:** First element added to the stack is the first to be removed
- Elements added to **rear** of a “queue” and removed from **front**
- Two pointers point to the **front** and the **rear**

Operations

- **Enqueue:** Adds an element to the end of the queue
- **Dequeue:** Removes the element from the front of the queue



Queues: deque Implementation

- deque makes use of a doubly-linked list

```
# Simple queue class using deque to define operations
from collections import deque

class Queue:
    # Initialise the queue by creating a deque
    # A queue can be created of fixed length as well
    def __init__(self):
        self.queue = deque()
```

Note: Lists can be used to implement a queue, but this will either come at a cost of time, since using the `pop(n)` function has an $O(n)$ time complexity, or at a cost of space, since using pointers would mean that the list would grow infinitely but also have empty elements in the front.

Queues: deque Implementation

```
# Add an element to the end of the queue
def enqueue(self, value):
    self.queue.append(value)

# Remove an element from the front of the queue
def dequeue(self):
    if len(self.queue) == 0:
        print("Queue Underflow!")
        return None
    else:
        return self.queue.popleft()
```

Queues

Complexity Analysis

- **Enqueue**
 - **Space: $O(1)$**
No extra space used
 - **Time: $O(1)$**
Single memory allocation done in constant time
- **Dequeue**
 - **Space: $O(1)$**
No extra space used
 - **Time: $O(1)$**
Front pointer incremented by 1 and node deallocated

Common Uses

- Task Scheduling
- Resource Allocation
- Network Protocols
- Printing Queues
- Web Servers
- Breadth-First Search

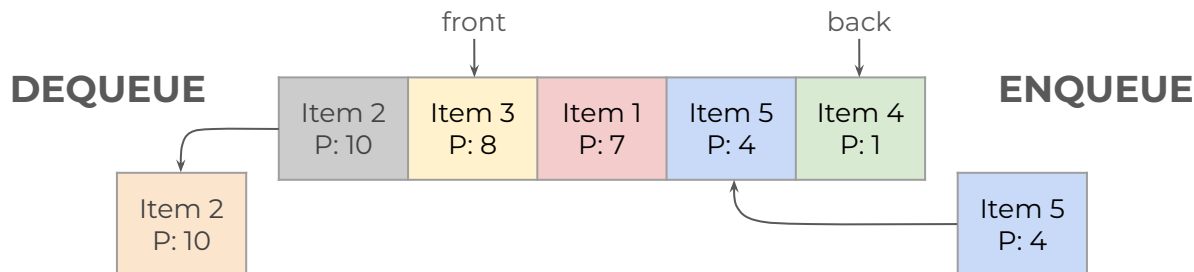
Priority Queues

Method of Ordering

- Arranged according to the assigned **priority** of elements
- Order direction doesn't matter, as long as **the highest priority elements are removed first**

Operations

- **Enqueue:** Adds an element to the queue based on its priority
- **Dequeue:** Removes the highest priority element from the queue



Priority Queues: List Implementation

```
# Simple priority queue class which uses a list

class PriorityQueue:
    # Initialise the queue by creating an empty list
    # Each element will be a pair of values (tuple)
    def __init__(self):
        self.pqueue = []
```

Note: The sort function is used at all times so elements are added to a sorted list which improves time complexities. The list is sorted in ascending order so that the element at the end of the list has the highest priority thus the pop() will have a $O(1)$ complexity.

Priority Queues: List Implementation

```
# Add an element to the end of the queue  
# Sort by priority to queue by priority  
def enqueue(self, value, priority):  
    self.pqueue.append((priority, value))  
    self.pqueue.sort()
```

Note: The sort function is used at all times so elements are added to a sorted list which improves time complexities. The list is sorted in ascending order so that the element at the end of the list has the highest priority thus the pop() will have a $O(1)$ complexity.

Priority Queues: List Implementation

```
# Remove the element with the highest priority
# The element would be at the end of list but
# the beginning of our queue
def dequeue(self):
    if len(self.pqueue) == 0:
        print("Error: Priority Queue Underflow!")
        return None
    else:
        return self.pqueue.pop()[1]
```

Note: The sort function is used at all times so elements are added to a sorted list which improves time complexities. The list is sorted in ascending order so that the element at the end of the list has the highest priority thus the pop() will have a $O(1)$ complexity.

Priority Queues: queue Implementation

```
# Simple priority queue class which uses the queue module
import queue

class PriorityQueue:
    # Initialise the PQ with an instance of the PQ class
    # A PQ of fixed length can be created as well
    def __init__(self):
        self.pqueue = queue.PriorityQueue()
```

Note: This implementation makes use of a structure called a min-heap to improve efficiency. A min-heap maintains that all parent nodes are less than or equal to all child nodes.

Priority Queues: queue Implementation

```
# Insert an element into the PQ based on its priority
# This PQ gives the lowest values the highest priority
# We change the sign of the priorities to fit our need
def enqueue(self, value, priority):
    self.pqueue.put((-1*priority, value))
```

Note: This implementation makes use of a structure called a min-heap to improve efficiency. A min-heap maintains that all parent nodes are less than or equal to all child nodes.

Priority Queues: queue Implementation

```
# Remove the element with the highest priority
def dequeue(self):
    if self.pqueue.qsize() == 0:
        print("Error: Priority Queue Underflow!")
        return None
    else:
        return self.pqueue.get()[1]
```

Note: This implementation makes use of a structure called a mini-heap to improve efficiency. A min-heap maintains that all parent nodes are less than or equal to all child nodes.

Priority Queues

Complexity Analysis

- **Enqueue**

- **Space:**

- **List - $O(1)$**
No extra space is used
- **PriorityQueue - $O(1)$**
No extra space is used

- **Time:**

- **List - $O(n)$**
Each element's priority must be checked and compared
- **PriorityQueue - $O(\log n)$**
Adding node to heap

- **Dequeue**

- **Space:**

- **List - $O(1)$**
No extra space is used
- **PriorityQueue - $O(1)$**
No extra space is used

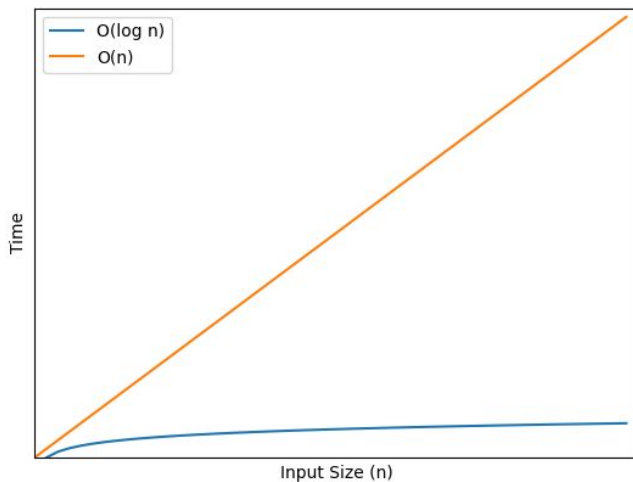
- **Time:**

- **List - $O(1)$**
Pointer is updated or last element is removed
- **PriorityQueue - $O(\log n)$**
Removing node from heap

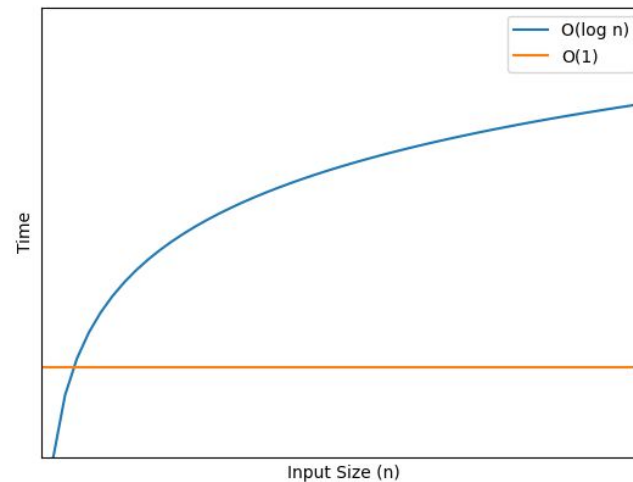
Priority Queues

Complexity Analysis Visualisation

Enqueue Time Complexity



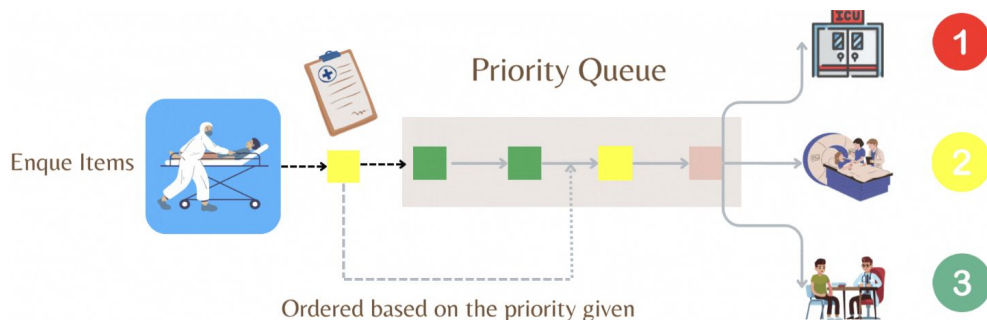
Dequeue Time Complexity



Priority Queues

Common Uses

- Dijkstra's Shortest Path Algorithm
- Data Compression
- Artificial Intelligence
- Load Balancing in OSs
- Optimisation Problems
- Robotics
- Medical Systems
- Event-driven simulations



Source: [Priority Queues in Healthcare \(Medium\)](#)

Worked Example

Consider a simple scheduler designed to manage the execution of tasks of varying priority.

We will consider two different implementations, one using a stack and another which uses a priority queue.

1. Discuss the choice between a stack or a priority queue for this implementation based on each data structure's performance and flexibility.
2. Based on the answer in 1, determine under what conditions it would be better to use each structure.
3. Implement a simple scheduler using either a stack or a priority queue.

Worked Example

Consider a simple scheduler designed to manage the execution of tasks of varying priority.

We will consider two different implementations, one using a stack and another which uses a priority queue.

1. Discuss the choice between a stack or a priority queue for this implementation based on each data structure's performance and flexibility.

Performance: Better speed when using a stack [$O(1)$ vs $O(\log n)$] but better suited order of execution with a priority queue.

Flexibility: Priority queues have better flexibility since priority can influence order of execution.

2. Based on the answer in 1, determine the best suited conditions for each data structure.

Stack: Performance is important and task prioritisation is not critical

Priority Queue: Task prioritisation critically impacts order of execution.

Worked Example

Consider a simple scheduler designed to manage the execution of tasks of varying priority.

We will consider two different implementations, one using a stack and another which uses a priority queue.

3. Implement a simple scheduler using either a stack or a priority queue.

Examples of both implementations can be found with the source code for this lecture.

Summary

Stacks

- ★ **Order:** LIFO
- ★ **Operations:** Push() and Pop()
- ★ **Time complexity:** $O(1)$
- ★ **Space complexity:** $O(1)$

Queues

- ★ **Order:** FIFO
- ★ **Operations:** Enqueue() and Dequeue()
- ★ **Time complexity:** $O(1)$
- ★ **Space complexity:** $O(1)$

Summary

Priority Queues

- ★ **Order:** Based on the priority of each element
- ★ **Operations:** Enqueue() and Dequeue()

- ★ **Time complexity [List]:** $O(n)$
- ★ **Space complexity [List]:** $O(1)$

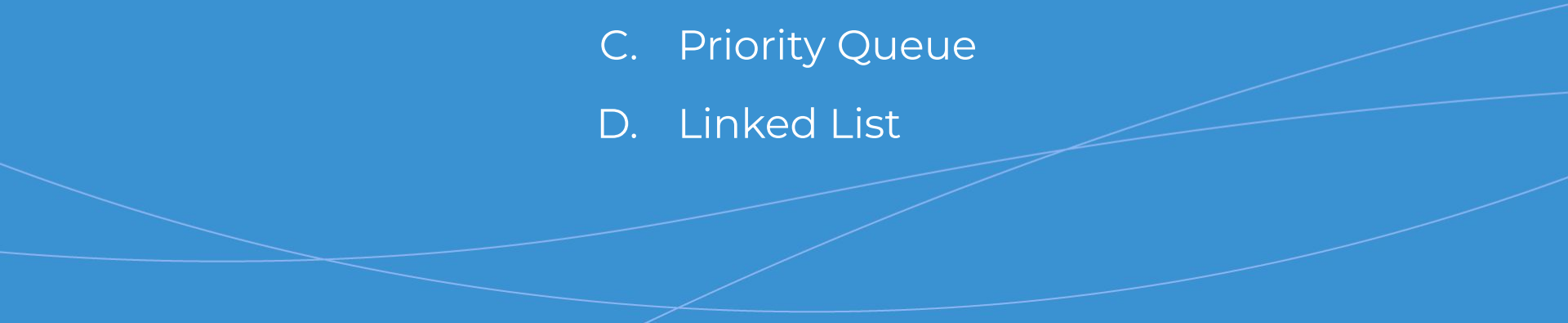
- ★ **Time complexity [PriorityQueue]:** $O(\log n)$
- ★ **Space complexity [PriorityQueue]:** $O(\log n)$


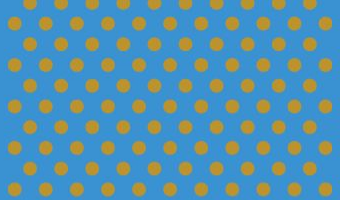
Further Learning

- [StackAbuse](#) - Stacks and Queues Basics
- [GeeksforGeeks](#) - Data Structures includes in depth information and implementation of Stacks, Queues and Priority Queues
- [GeeksforGeeks](#) - Stacks Complexity Analysis
- [GeeksforGeeks](#) - Queues Complexity Analysis

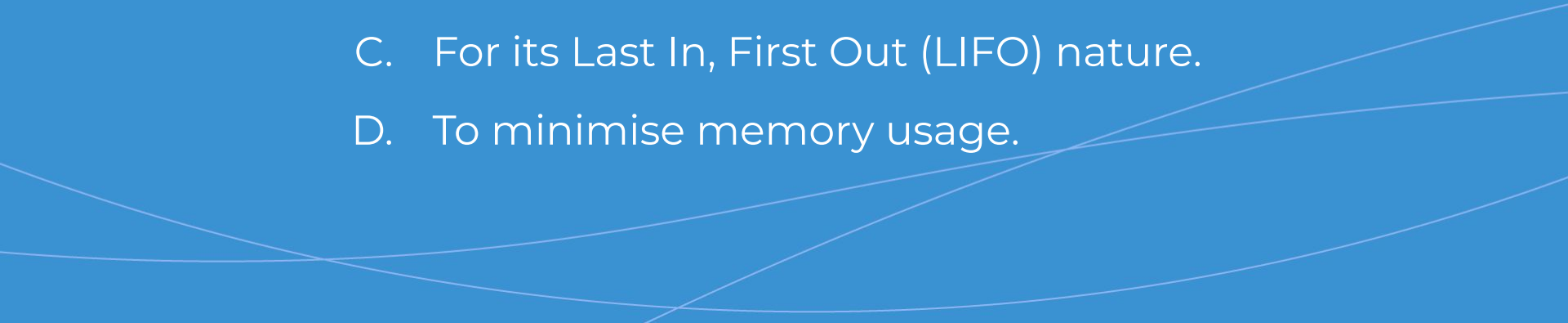


Which data structure would be most appropriate for implementing a backtracking algorithm?

- A. Stack
 - B. Queue
 - C. Priority Queue
 - D. Linked List
- 



In a task scheduling system, why might a priority queue be preferred over a simple queue?

- A. For faster access to elements.
 - B. To process tasks based on their priority.
 - C. For its Last In, First Out (LIFO) nature.
 - D. To minimise memory usage.
- 



Questions and Answers

Questions around Stacks and Queues

