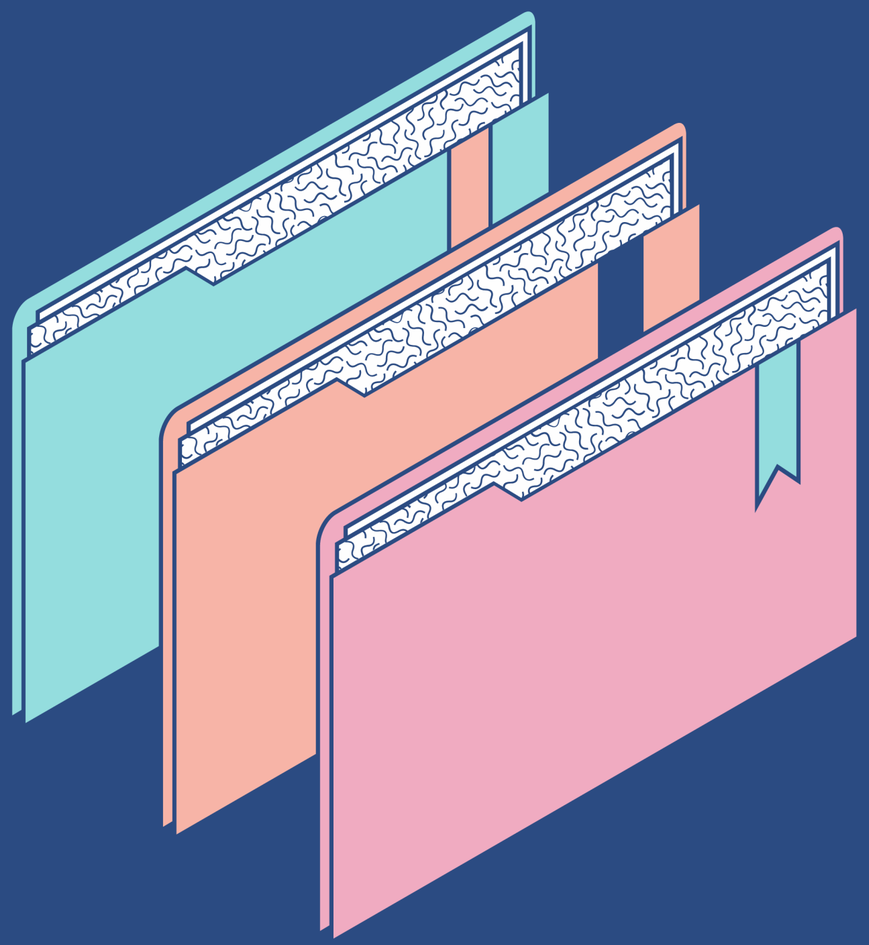
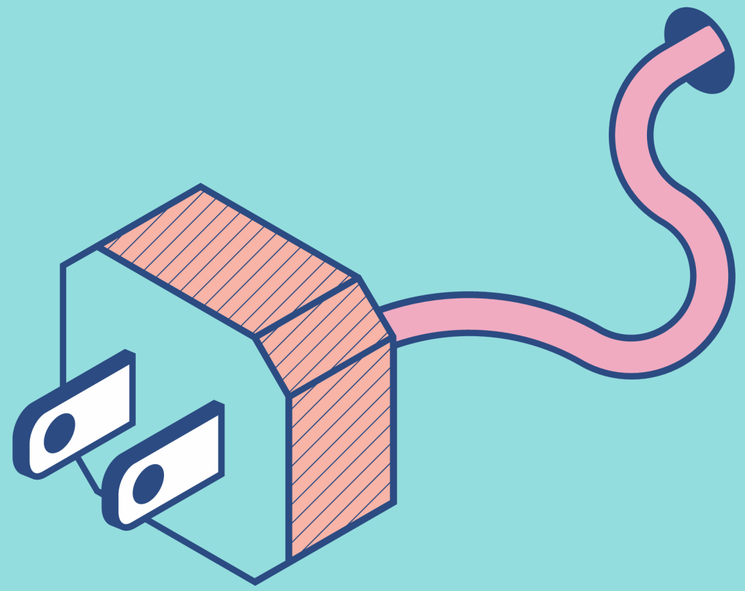


# program

- Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.
- Determine the operations of a memory stack and how it is used to implement function calls in a computer.
- Illustrate, with an example, a concrete data structure for a First in First out (FIFO) queue.
- Compare the performance of two sorting algorithms.
- Analyse the operation, using illustrations, of two network shortest path algorithms, providing an example of each.

# Create a design specification for data structures, explaining the valid operations that can be carried out on the structures.

- Identify the Data Structures
- Define the Operations
- Specify Input Parameters
- Define Pre- and Post-conditions
- Discuss Time and Space Complexity
- Provide Examples and Code Snippets (if applicable)



Array

- Description:
- An array is a collection of elements stored in contiguous memory locations. Each element is indexed, allowing for constant time access.
- Operations:
- Access (Retrieve an element)
- Insert (Add an element)
- Delete (Remove an element)
- Input Parameters:
- For Access: Index (integer)
- For Insert: Index (integer), Value (same data type as array elements)
- For Delete: Index (integer)

**Pre-conditions:**

- **Access/Insert/Delete: The index must be within the array bounds (0 ≤ index < size).**

**Post-conditions:**

- **Access: The element at the given index is returned.**
- **Insert: The value is added at the given index, shifting elements to the right if necessary.**
- **Delete: The element at the given index is removed, and subsequent elements shift left to fill t gap.**

**Time Complexity:**

- **Access: O(1)**
- **Insert: O(n) (worst-case, when inserting at the beginning)**
- **Delete: O(n) (worst-case, when deleting at the beginning)**

**Space Complexity: O(n)**

2. Linked List

Description:

A linked list is a dynamic data structure consisting of nodes, where each node contains data and a pointer to the next node in the sequence.

Operations:

- Access (Retrieve an element by index)
- Insert (Add a node)
- Delete (Remove a node)

Input Parameters:

- For Access: Index (integer)
- For Insert: Index (integer), Value (data to insert)
- For Delete: Index (integer)

Pre-conditions:

- Access/Insert/Delete: Index must be within the list bounds (0 ≤ index < size).

Post-conditions:

- Access: Returns the value stored in the node at the given index.
- Insert: A new node is created and inserted at the specified index, adjusting pointers accordingly.
- Delete: The node at the given index is removed, and pointers are adjusted.

Time Complexity:

- Access: O(n)
- Insert: O(n) (worst-case when inserting at the end)
- Delete: O(n) (worst-case when deleting at the end)

Space Complexity: O(n) (for storing n elements)

3. Stack (LIFO)

Description:

A stack is a linear data structure following the Last In, First Out (LIFO) principle. Elements are added and removed from only one end, known as the "top."

Operations:

- Push (Insert an element onto the stack)
- Pop (Remove the top element)
- Peek (Retrieve the top element without removing it)

Input Parameters:

- For Push: Value (element to add)
- For Pop/Peek: None

Pre-conditions:

- Push: The stack should not be full (if a size limit is set).
- Pop/Peek: The stack must not be empty.

Post-conditions:

- Push: The element is added to the top of the stack.
- Pop: The top element is removed and returned.
- Peek: The top element is returned without removal.

Time Complexity:

- Push/Pop/Peek: O(1)

Space Complexity: O(n)

4. Queue (FIFO)

Description:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. Elements are added at the rear and removed from the front.

Operations:

- Enqueue (Insert an element at the rear)
- Dequeue (Remove the front element)
- Front (Retrieve the front element without removing it)

Input Parameters:

- For Enqueue: Value (element to add)
- For Dequeue/Front: None

Pre-conditions:

- Enqueue: The queue should not be full (if a size limit is set).
- Dequeue/Front: The queue must not be empty.

Post-conditions:

- Enqueue: The element is added to the rear.
- Dequeue: The front element is removed and returned.
- Front: The front element is returned without removal.

Time Complexity:

- Enqueue/Dequeue/Front: O(1)

Space Complexity: O(n)

5. Hash Table

Description:

A hash table is a data structure that maps keys to values. It uses a hash function to compute an index where the value is stored, providing efficient lookups.

Operations:

- Insert (Add a key-value pair)
- Delete (Remove a key-value pair)
- Search (Find the value associated with a key)

Input Parameters:

- For Insert: Key (unique identifier), Value (data associated with the key)
- For Delete: Key (unique identifier)
- For Search: Key (unique identifier)

Pre-conditions:

- Insert: The key should not already exist (for non-duplicate keys).
- Delete/Search: The key must exist in the hash table.

Post-conditions:

- Insert: The key-value pair is added.
- Delete: The key-value pair is removed.
- Search: The value associated with the key is returned.

Time Complexity:

- Insert/Delete/Search: O(1) on average (with good hash function), O(n) in the worst case due to collisions.

Space Complexity: O(n)

6. Binary Search Tree (BST)

Description:

A binary search tree is a hierarchical data structure where each node has at most two children (left and right). The left child has a smaller value than the parent, and the right child has a larger value.

Operations:

- Insert (Add a node)
- Delete (Remove a node)
- Search (Find a node)

Input Parameters:

- For Insert: Value (data to add)
- For Delete/Search: Value (data to remove or find)

Pre-conditions:

- Insert: The tree should not already contain the value.
- Delete/Search: The value must exist in the tree.

Post-conditions:

- Insert: The node is added to the appropriate position.
- Delete: The node is removed, and the tree is rebalanced (if applicable).
- Search: The node is found and returned.

Time Complexity:

- Insert/Delete/Search: O(log n) on average (in a balanced tree), O(n) in the worst case (unbalanced tree).

Space Complexity: O(n)

# Determine the operations of a memory stack and how it is used to implement function calls in a computer.

1. Memory Stack Definition

A memory stack (call stack) is a region of memory that stores information about the active subroutines (functions) in a computer program. It is primarily used to manage function calls, return addresses, local variables, and control flow. The stack grows as new function calls are made, and shrinks as functions return.

2. Operations of a Memory Stack

The primary operations performed on a memory stack are:

- Push: Adds a new element (e.g., function call data) to the top of the stack.
- Pop: Removes the topmost element from the stack (i.e., function return).
- Peek: Retrieves the top element without modifying the stack (used to access current function details).

These operations are LIFO (Last In, First Out), meaning the last function pushed onto the stack is the first one to return.

# Determine the operations of a memory stack and how it is used to implement function calls in a computer.

3. Function Call Implementation Using the Stack

When a function is called, a stack frame is created and pushed onto the memory stack. This stack frame typically contains:

- Return Address: Where the program should continue after the function returns.
- Function Parameters: Arguments passed to the function.
- Local Variables: Variables local to the function.
- Saved Registers: Any CPU registers that need to be restored after the function executes.

Upon returning from a function, the stack frame is popped, restoring the previous state of the program.

Steps:

1. The return address and parameters are pushed onto the stack.
2. The local variables and other data are pushed onto the stack.
3. When the function ends, the stack frame is popped, and control is transferred to the return address.

# Determine the operations of a memory stack and how it is used to implement function calls in a computer.

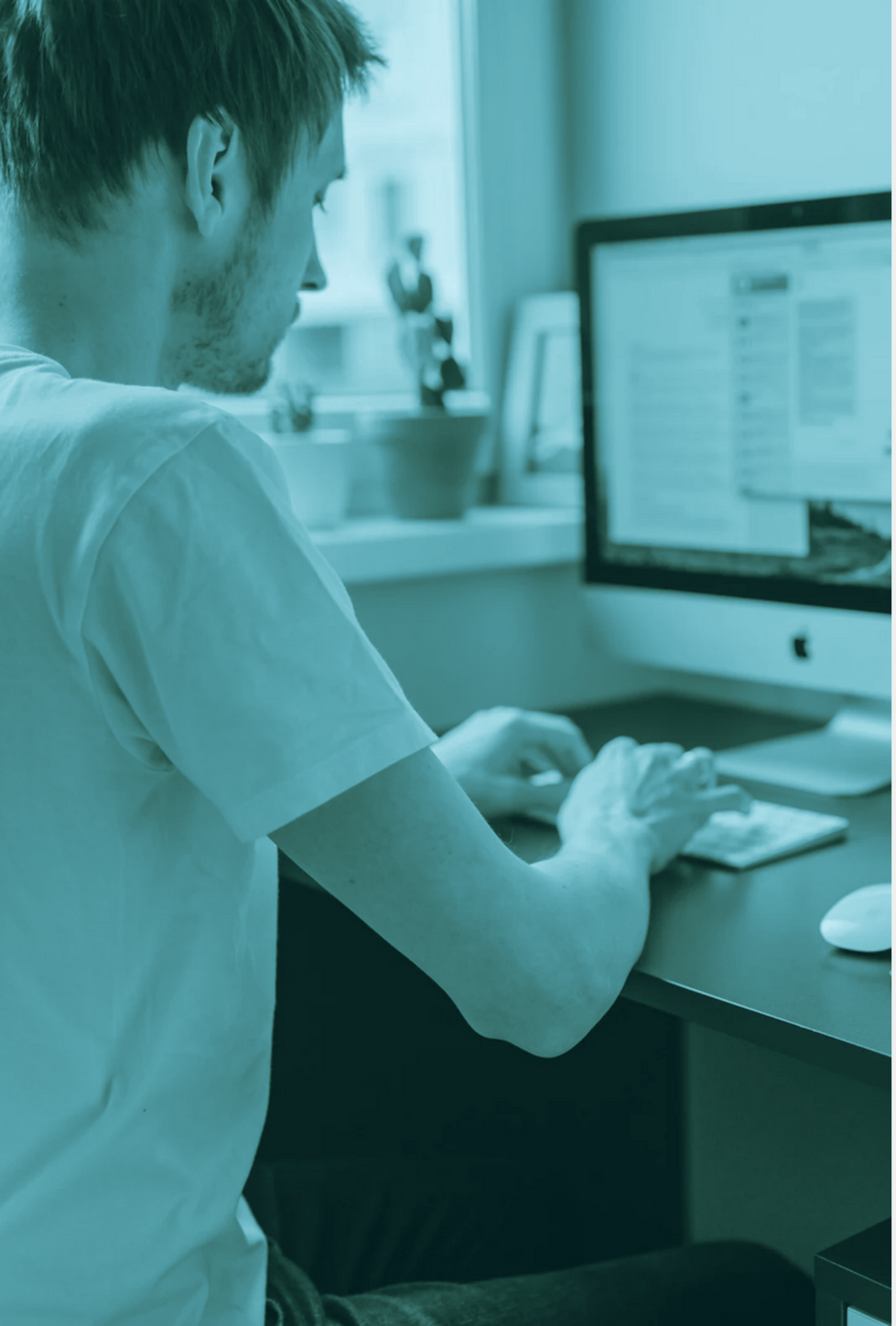
Memory Stack during execution:

1. Initial call to factorial(5):
   - The stack frame for main() is at the bottom.
   - Stack frame for factorial(5) is pushed, storing the return address and n = 5.
2. Recursive call to factorial(4):
   - Stack frame for factorial(4) is pushed, storing return address and n = 4.
3. Continuing recursion factorial(3), factorial(2), factorial(1):
   - Each recursive call adds a new stack frame on top, with return addresses and the respective n values.
4. Return from factorial(1):
   - Stack frame for factorial(1) is popped.
   - Control returns to factorial(2), and the result is used for multiplication.
5. Final result returned to main():
   - Once the recursion unwinds, all stack frames are popped until the result is passed to main().

# Determine the operations of a memory stack and how it is used to implement function calls in a computer.

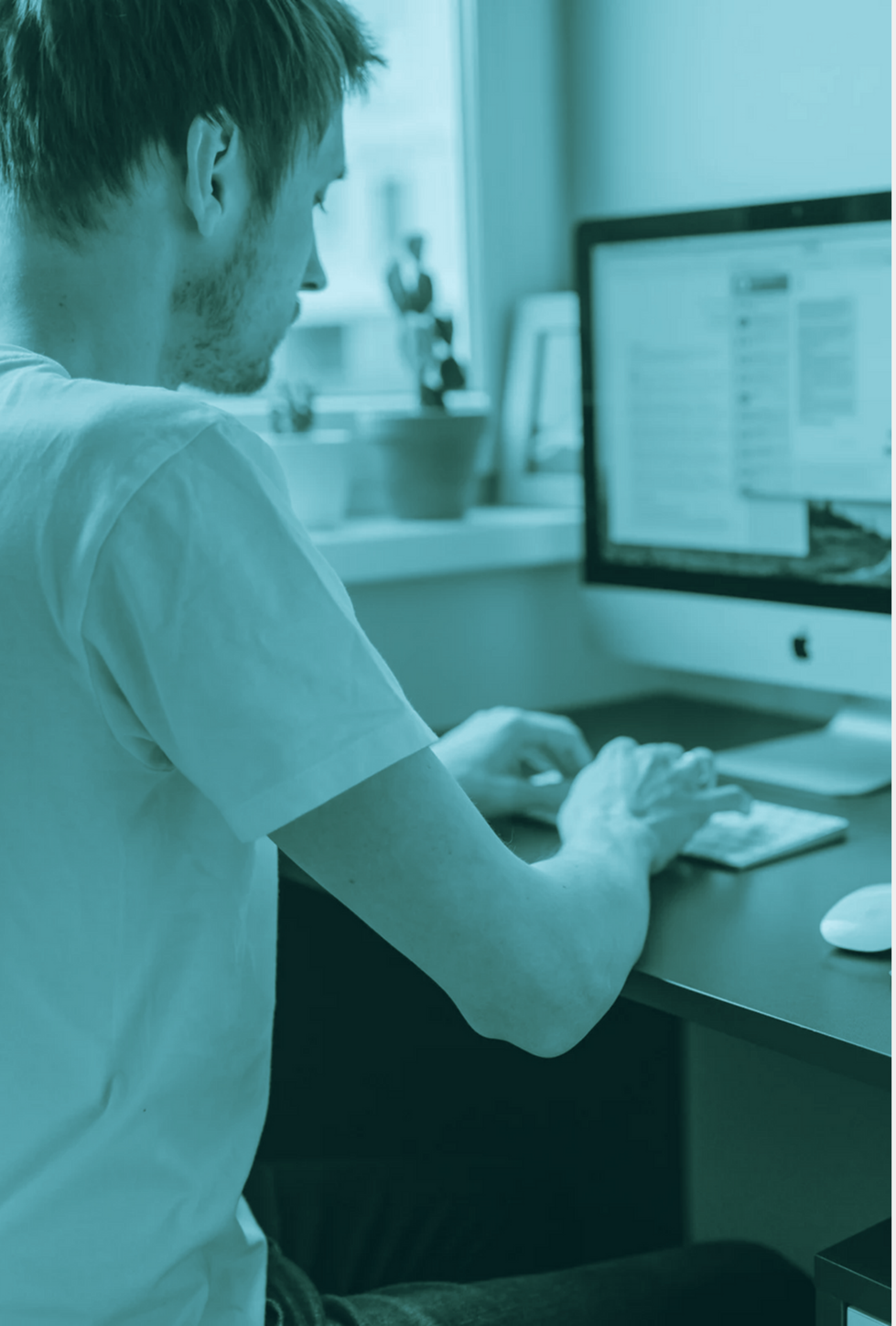
5. Importance of the Memory Stack
   - Managing Function Calls: The memory stack ensures each function's data (parameters, local variables, etc.) is isolated from others, making nested and recursive calls possible.
   - Tracking Execution Flow: The return addresses ensure that after completing a function, the program can continue from the correct location.
   - Memory Efficiency: Since stack frames are automatically popped when a function returns, the memory used for local variables is efficiently reclaimed.

Key Points:
   - A memory stack is essential for recursion, as it allows multiple function calls to be handled independently.
   - Errors like stack overflow occur when the stack grows beyond its allocated memory (e.g., deep recursion).



## Introduction to FIFO

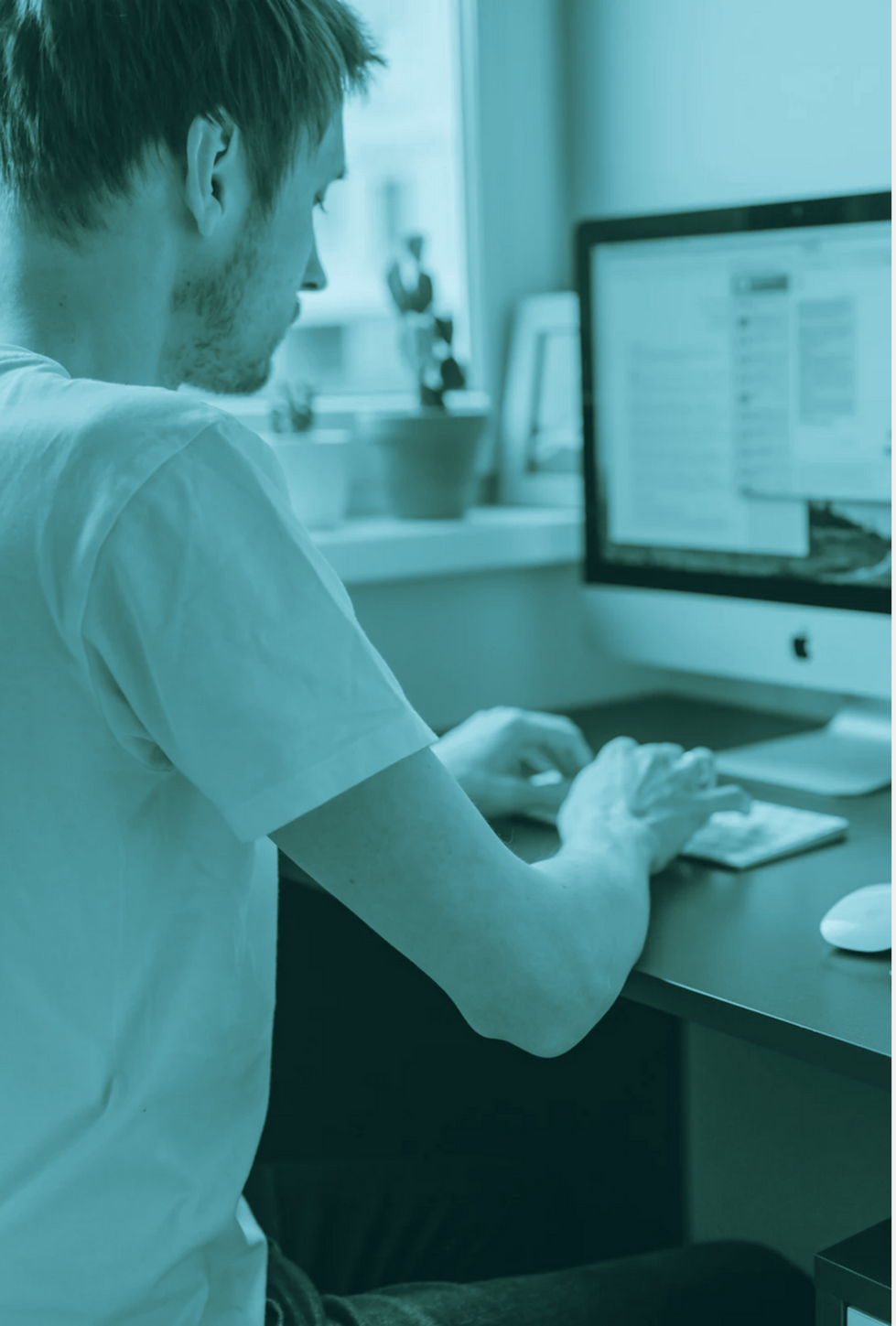
FIFO (First In, First Out) is a data structure that processes elements in the order they are added. The first element added to the queue will be the first one to be removed, much like a line of people waiting for service. FIFO queues are commonly used in scenarios like scheduling tasks, managing requests in servers, and buffering data.



## 1. Define the Structure

A FIFO queue typically supports the following operations:

- Enqueue: Add an element to the end of the queue.
- Dequeue: Remove an element from the front of the queue.
- Peek: View the front element without removing it.
- IsEmpty: Check if the queue is empty.

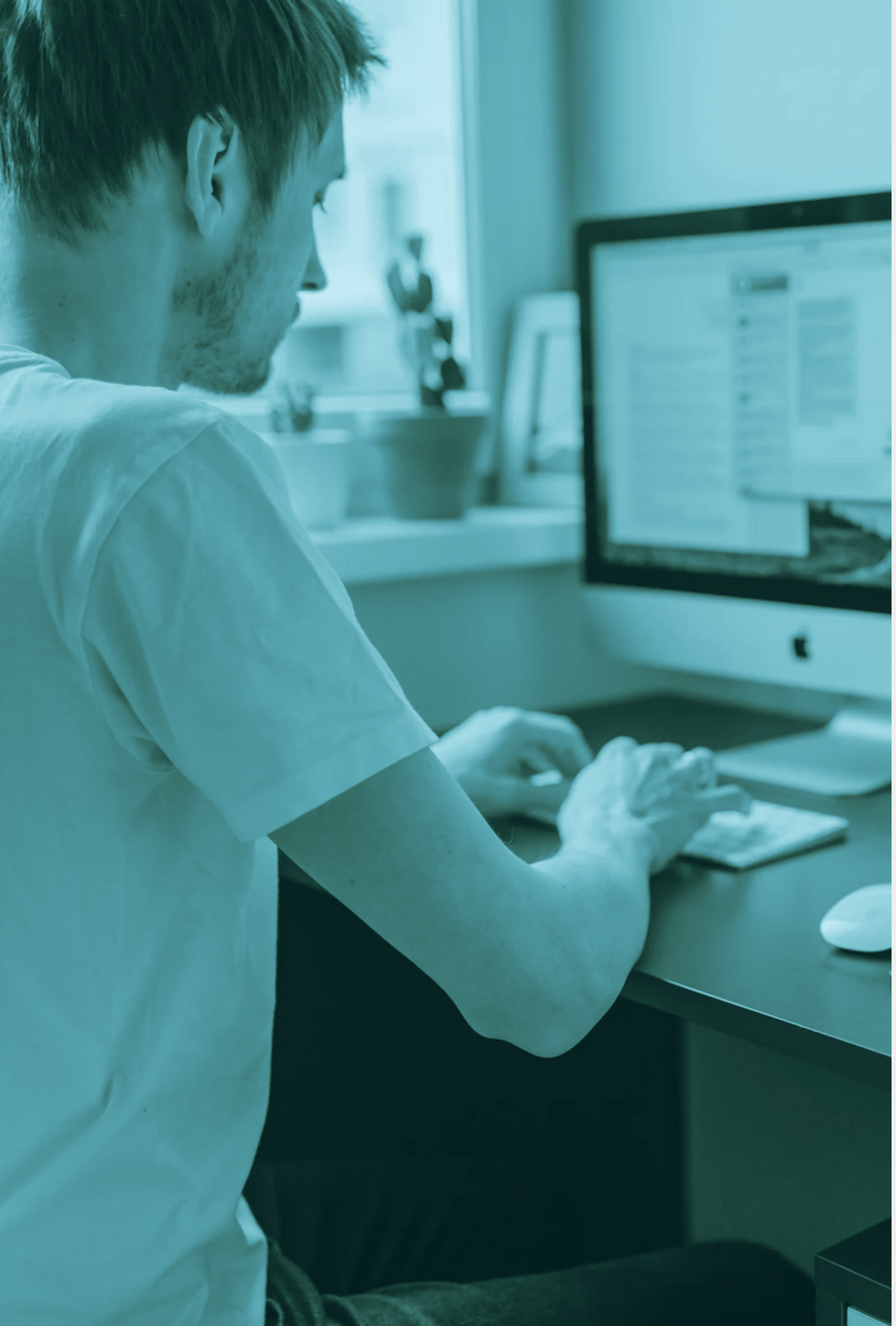


## 2. Array-Based Implementation

In an array-based implementation, a fixed-size array is used to store the elements. Two pointers, front and rear, are maintained to track the beginning and end of the queue.

## 3. Linked List-Based Implementation

In a linked list-based implementation, each element is stored in a node, and the queue maintains pointers to the front and rear nodes. This allows for dynamic sizing without worrying about capacity.



**4. Concrete Example to Illustrate How the FIFO Queue Works**

Let's illustrate a FIFO queue with a concrete example using both implementations.

Example Scenario:

Consider a queue of customers at a ticket counter.

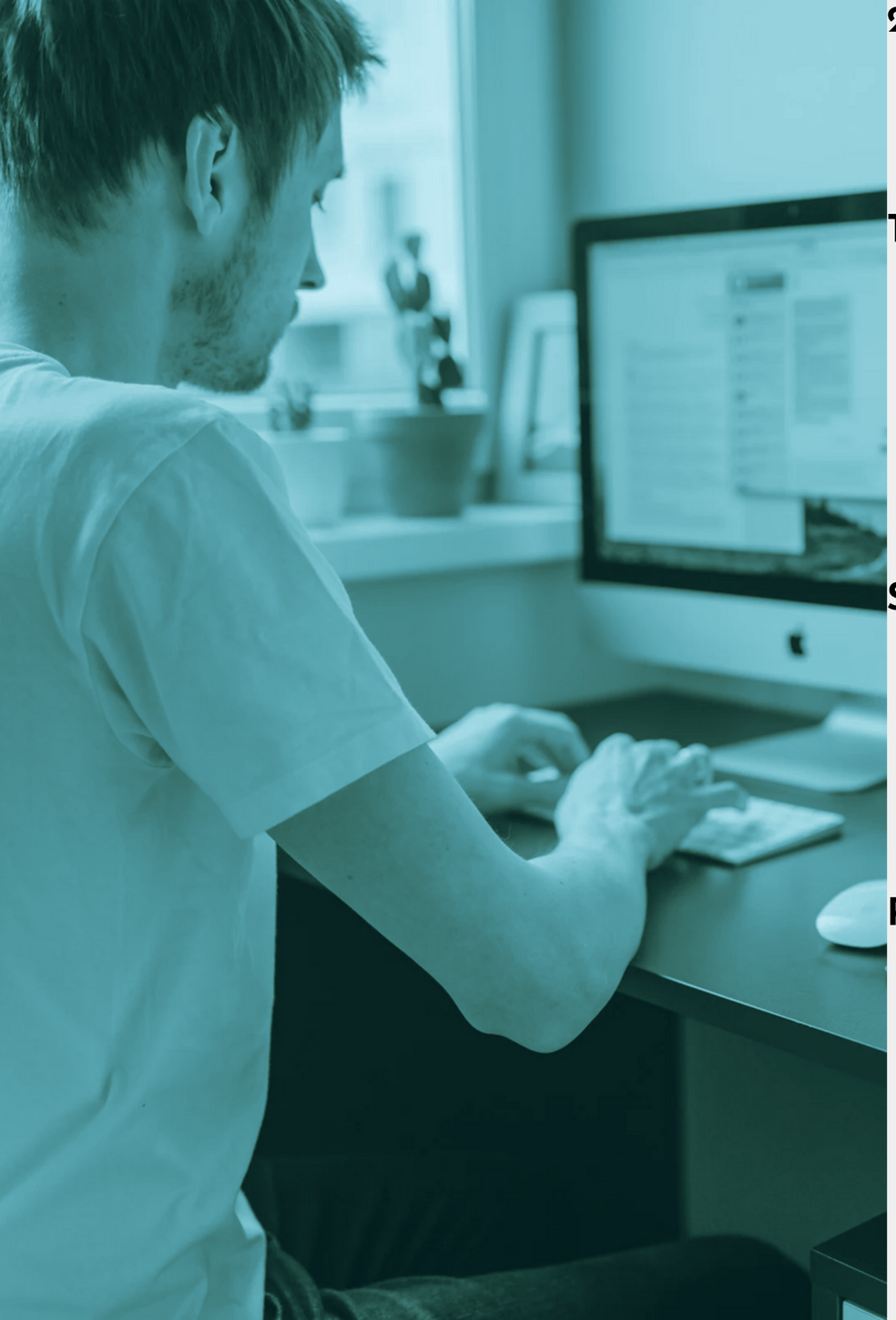
1. Enqueue Operations:
   - Customer A (ID: 1) arrives: enqueue(1)
   - Customer B (ID: 2) arrives: enqueue(2)
   - Customer C (ID: 3) arrives: enqueue(3)
2. Queue State:
   - Array Implementation: [1, 2, 3]
   - Linked List Implementation: 1 -> 2 -> 3
3. Dequeue Operations:
   - Customer A is served: dequeue() returns 1
   - Customer B is served: dequeue() returns 2
4. Queue State After Dequeue:
   - Array Implementation: [3] (front index moves to 1)
   - Linked List Implementation: 3
5. Remaining Operations:
   - Peek at front: The front customer (Customer C) can be checked using peek(), which will return 3 without removing it from the queue.

## Introduction to the Two Sorting Algorithms

For this comparison, we will examine Merge Sort and Quick Sort, two widely used sorting algorithms that have different approaches and performance characteristics.

### 1. Merge Sort

- Type: Comparison-based, divide-and-conquer sorting algorithm.
- Approach: Divides the input array into two halves, sorts each half recursively, and merges the sorted halves back together.



## 2. Quick Sort

- Type: Comparison-based, divide-and-conquer sorting algorithm.
- Approach: Selects a 'pivot' element, partitions the array into elements less than and greater than the pivot, and recursively sorts the partitions.

Time Complexity Analysis

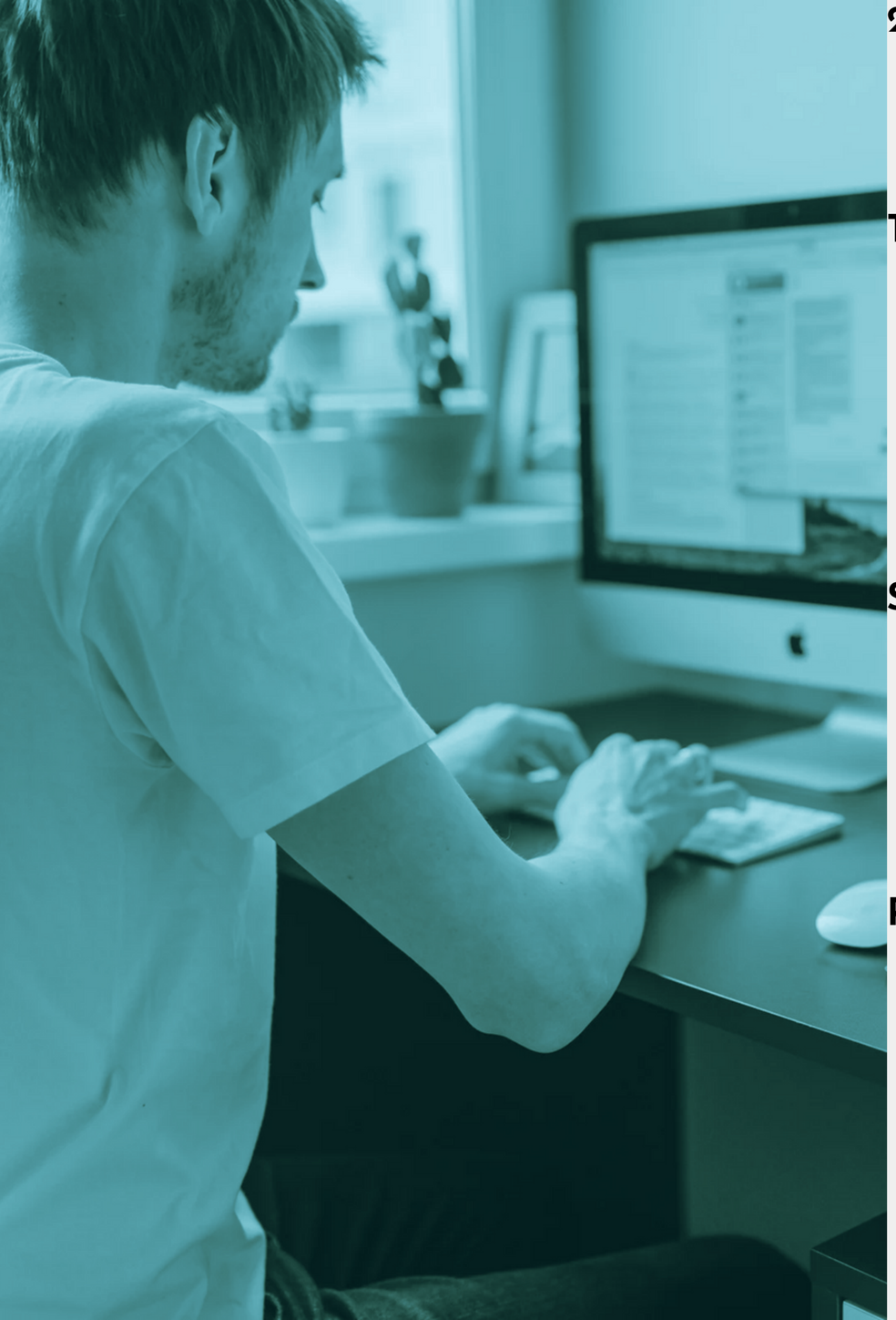
- Merge Sort has a consistent time complexity of O(nlogn)O(n \log n)O(nlogn) across all cases because it always divides the array in half.
- Quick Sort has a best and average case of O(nlogn)O(n \log n)O(nlogn), but its worst case is O(n2)O(n^2)O(n2) when the smallest or largest element is consistently chosen as the pivot (e.g., for already sorted arrays).

Stability

- Merge Sort maintains the relative order of records with equal keys (stability).
- Quick Sort does not guarantee stability, as the relative order may change when elements are swapped during partitioning.

Performance Comparison

- Efficiency: Merge Sort is consistently efficient with its O(nlogn)O(n \log n)O(nlogn) time complexity, making it suitable for large datasets and linked lists. Quick Sort is faster on average but can degrade to O(n2)O(n^2)O(n2) in the worst-case scenario.
- Space Usage: Merge Sort uses more memory, while Quick Sort operates in-place, which can be a crucial factor in memory-limited environments.



## 2. Quick Sort

- Type: Comparison-based, divide-and-conquer sorting algorithm.
- Approach: Selects a 'pivot' element, partitions the array into elements less than and greater than the pivot, and recursively sorts the partitions.

Time Complexity Analysis

- Merge Sort has a consistent time complexity of O(nlogn)O(n \log n)O(nlogn) across all cases because it always divides the array in half.
- Quick Sort has a best and average case of O(nlogn)O(n \log n)O(nlogn), but its worst case is O(n2)O(n^2)O(n2) when the smallest or largest element is consistently chosen as the pivot (e.g., for already sorted arrays).

Stability

- Merge Sort maintains the relative order of records with equal keys (stability).
- Quick Sort does not guarantee stability, as the relative order may change when elements are swapped during partitioning.

Performance Comparison

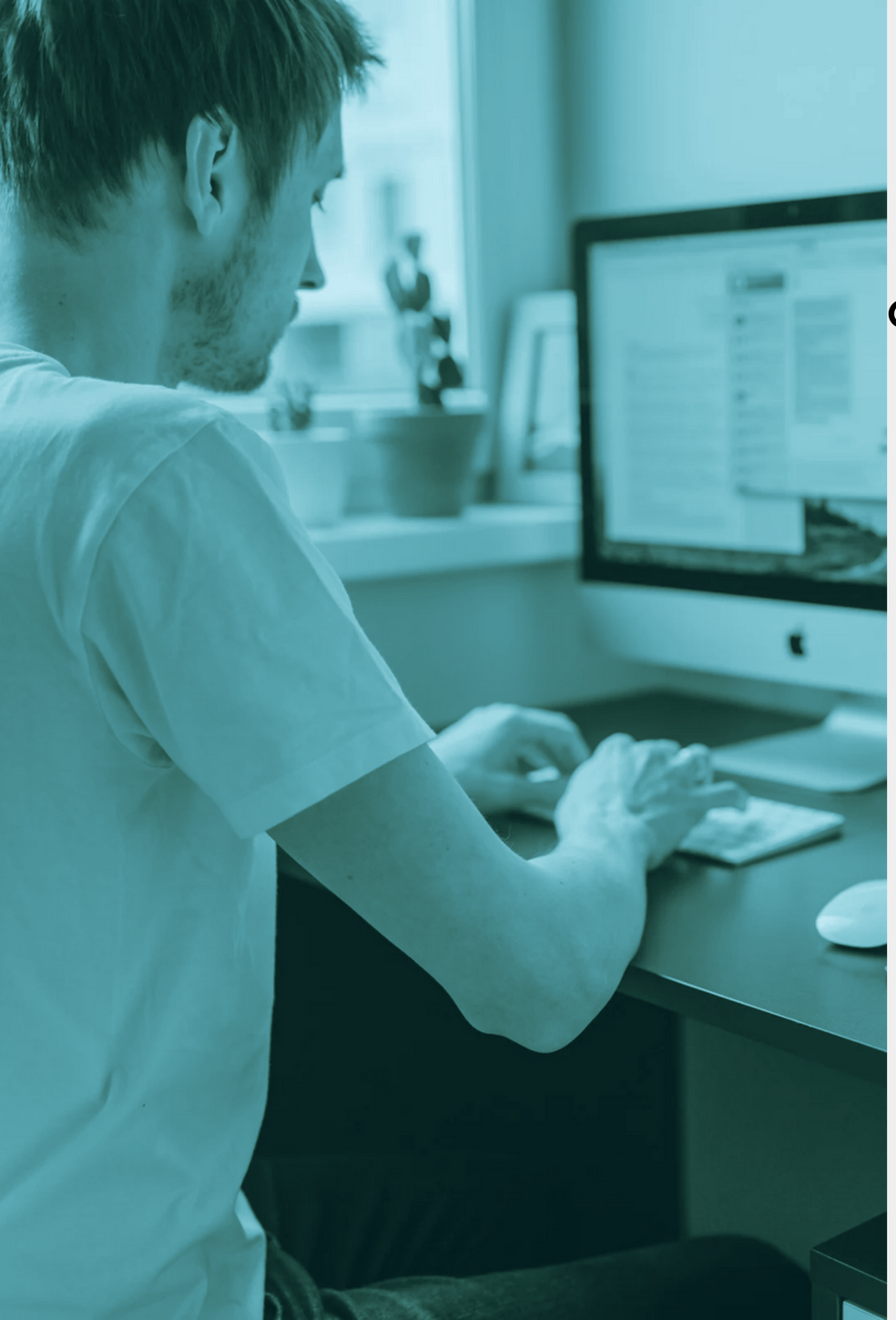
- Efficiency: Merge Sort is consistently efficient with its O(nlogn)O(n \log n)O(nlogn) time complexity, making it suitable for large datasets and linked lists. Quick Sort is faster on average but can degrade to O(n2)O(n^2)O(n2) in the worst-case scenario.
- Space Usage: Merge Sort uses more memory, while Quick Sort operates in-place, which can be a crucial factor in memory-limited environments.

**Concrete Example to Demonstrate Performance Differences**

**Input Array: [38, 27, 43, 3, 9, 82, 10]**

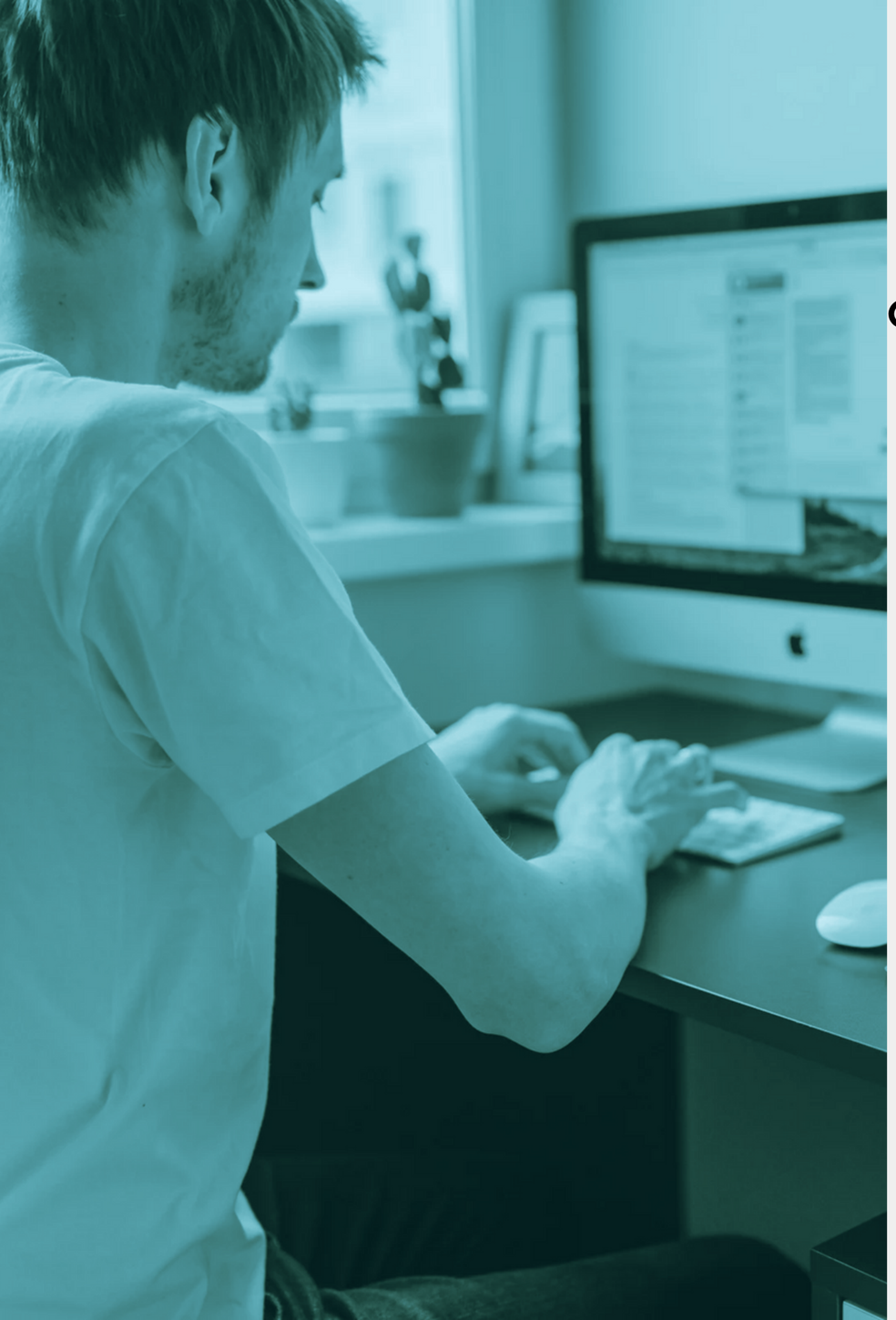
**Merge Sort Steps:**

1. **Split the array:**
   - **[38, 27, 43] and [3, 9, 82, 10]**
2. **Further split:**
   - **[38] [27] [43] and [3] [9] [82] [10]**
3. **Merge sorted arrays:**
   - **Merge [27] and [38] to get [27, 38], merge with [43] to get [27, 38, 43].**
   - **Merge [3] and [9] to get [3, 9], then merge with [82, 10] to get [3, 9, 10, 82].**
4. **Final merge:**
   - **Merge [27, 38, 43] and [3, 9, 10, 82] to get [3, 9, 10, 27, 38, 43, 82].**



**Quick Sort Steps:**

1. **Choose a pivot (let's say 10):**
    - **Rearrange to get [3, 9, 10, 38, 27, 43, 82].**
2. **Recursively sort the left partition [3, 9] (already sorted) and the right partition [38, 27, 43, 82].**
3. **Choose a new pivot from the right partition (e.g., 27):**
    - **Rearrange to get [3, 9, 10, 27, 38, 43, 82].**



**Quick Sort Steps:**

1. **Choose a pivot (let's say 10):**
    - **Rearrange to get [3, 9, 10, 38, 27, 43, 82].**
2. **Recursively sort the left partition [3, 9] (already sorted) and the right partition [38, 27, 43, 82].**
3. **Choose a new pivot from the right partition (e.g., 27):**
    - **Rearrange to get [3, 9, 10, 27, 38, 43, 82].**

## Introduction to Network Shortest Path Algorithms

Network shortest path algorithms are used to find the shortest path between nodes in a graph. These algorithms are vital in various applications, such as routing in networks, urban traffic navigation, and geographic information systems. They help minimize travel time, distance, or cost.

## Algorithm 1: Dijkstra's Algorithm

Overview: Dijkstra's Algorithm is designed to find the shortest paths from a starting node to all other nodes in a weighted graph with non-negative edge weights. It operates on the principle of maintaining a priority queue of vertices to explore.

Steps:

1. Initialize the distance to the starting node as 0 and all other nodes as infinity.
2. Add the starting node to the priority queue.
3. While the priority queue is not empty:
    - Remove the node with the smallest distance.
    - For each neighbor of this node, calculate the potential distance from the start node.
    - If this distance is shorter than the recorded distance, update it and add the neighbor to the priority queue.

**Illustration: Consider the following weighted graph:**

```
scss
Sao chép mã
      (2)
A -------- B
 | \       |
 |  \      | (1)
 |   \     |
(1)  (4)   |
 |     \   |
 |      \  |
C -------- D
      (3)
```

1. **Initialization:**
   - **Distances: A: 0, B: ∞, C: ∞, D: ∞**
   - **Priority Queue: {A: 0}**
2. **Processing:**
   - **Visit A: Update distances to B and C**
     - **Distances: B: 2, C: 1, D: ∞**
     - **Priority Queue: {B: 2, C: 1}**
   - **Visit C: Update distances to D**
     - **Distances: B: 2, C: 1, D: 4**
     - **Priority Queue: {B: 2, D: 4}**
   - **Visit B: Update distances to D**
     - **Distances: B: 2, C: 1, D: 3**
     - **Priority Queue: {D: 3}**
   - **Visit D: No updates needed.**
   - **Final distances from A: A: 0, B: 2, C: 1, D: 3.**

## Algorithm 2: Prim-Jarnik Algorithm

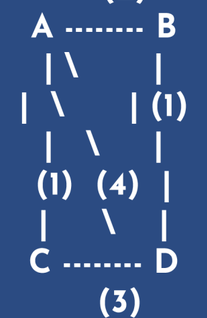
**Overview:** The Prim-Jarnik Algorithm (commonly referred to as Prim's Algorithm) is used for finding the Minimum Spanning Tree (MST) of a connected, weighted graph. It works by starting from a single vertex and growing the MST by adding the smallest edge from the tree to a vertex not yet in the tree.

**Steps:**

Initialize the MST with a starting vertex and mark it as part of the tree.
While there are vertices not yet included in the MST:
Find the edge with the smallest weight that connects a vertex in the MST to a vertex outside of it.
Add this edge and the corresponding vertex to the MST.

**Illustration:** Consider the following weighted graph:

scss
Sao chép mã




1. **Initialization:**
   - Start with vertex A. MST: {A}.
2. **Processing:**
   - Add the smallest edge (1) to C: MST becomes {A, C} with edges A-C.
   - Add the smallest edge (1) to B: MST becomes {A, B, C} with edges A-B, A-C.
   - Add the smallest edge (2) to D: MST becomes {A, B, C, D} with edges A-B, A-C, C-D.
   - Final MST edges: {A-B, A-C, C-D} with total weight 2 + 1 + 3 = 6.

Performance Analysis

| Algorithm | Time Complexity | Space Complexity | Applicability |
| --- | --- | --- | --- |
| Dijkstra's Algorithm | O((V + E) log V) (using priority queue) | O(V) | Shortest path in graphs with non-negative weights |
| Prim-Jarnik Algorithm | O(E log V) (using priority queue) | O(V) | Minimum Spanning Tree in connected graphs |

- Dijkstra's Algorithm is efficient for finding the shortest path from a single source to multiple destinations. It is generally used in routing and geographical applications.
- Prim-Jarnik Algorithm is efficient for constructing a Minimum Spanning Tree, essential for network design and clustering applications.