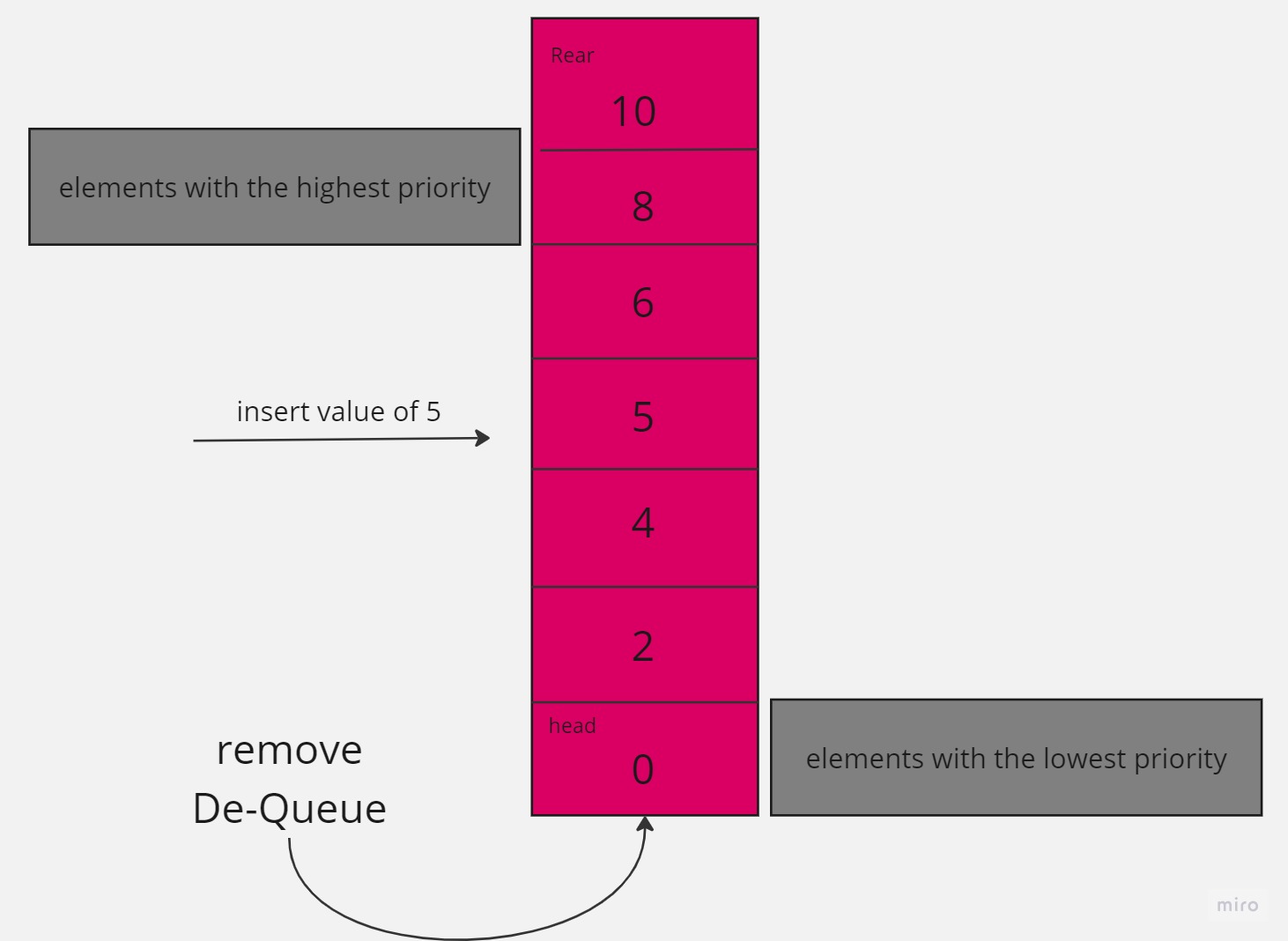
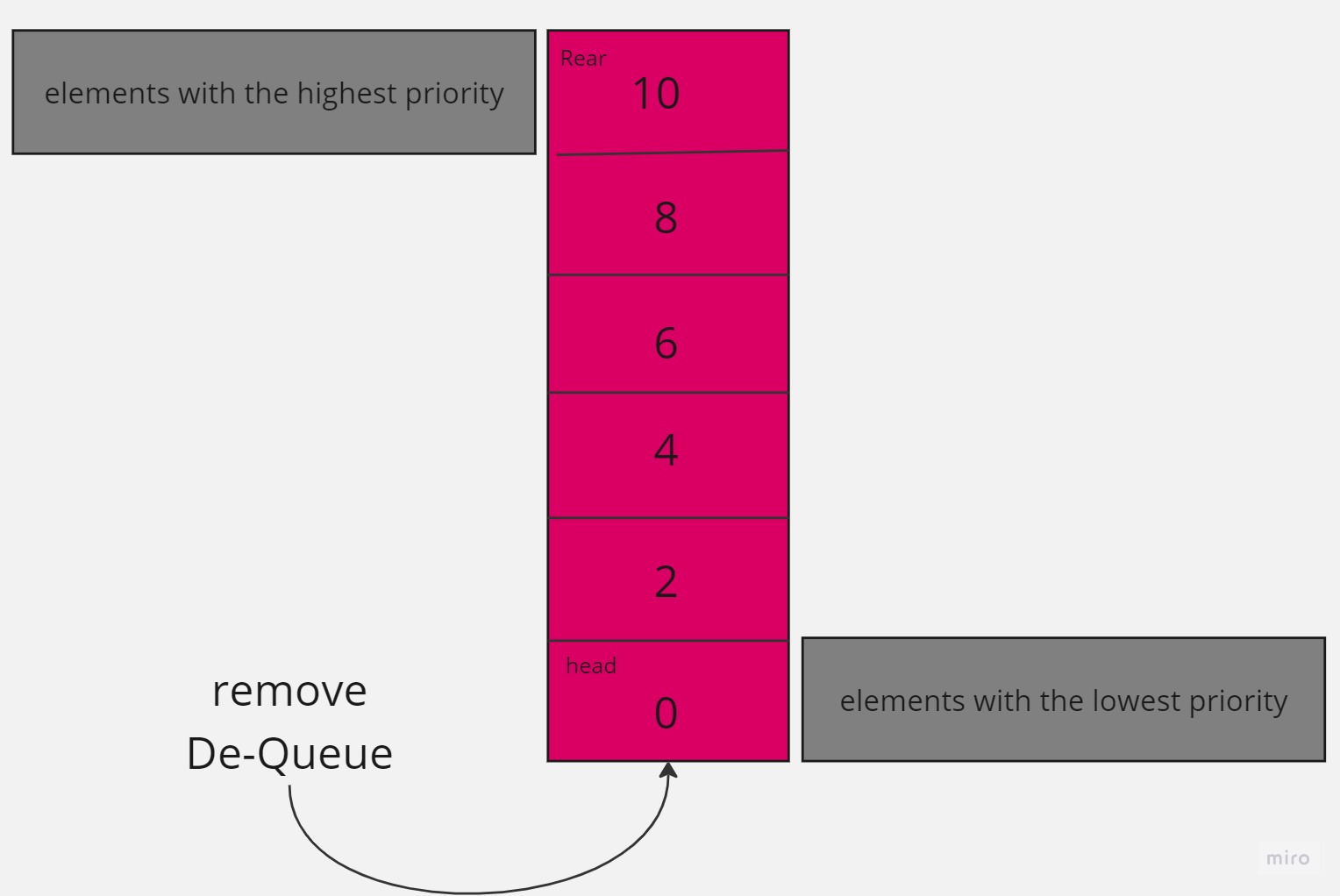
Final data structures

**Part one:-**

**Formal definition of queue ADT**: a list of item in which can be deleted from one end, it is the front of the queue, and an item can be inserted at the end of the queue called the rear of the queue.

The operation part include:

* Insert
* getMain
* removeMain



The procedure in the **insert** adds an element to the min- priority queue, and we must have elements to input to be inserted in. We create a new node with the given element and priority, now if the queue is empty we set the new node as the head, otherwise we Travers the linked-list from the head until finding a node with a higher priority or at the end of the list no higher priority id found. The time complexity O(n) in the worst case, where n is the number of the elements in the priority queue.

**GetMain** returns the element with the minimum priority from the min- priority queue, in which the output should be with the minimum priority. the procedure of the getMain return the element that is stored in the head of the node in the linked list, and its time complexity is O(1).

**RemoveMain** removes the element with the minimum priority from the min- priority queue. The procedure of the removeMain check of the priority of the queue is empty and it gives an error or a return a specified value to indicate the empty state, after that it sets the head nodes nect node as the new head. The time complexity of the removeMain is O(1).

The time complexity in an algorithm expressed in Big-O notation, provides an estimation of the algorithm's running time in relation to the input size. However, the actual running time of an algorithm can vary depending on factors such as the implementation details, hardware, and specific input instances.

1. **Selection sort**: it has a time complexity of O(n^2) in all cases. This means that as the input size increases, the running time of the algorithm grows quadratically.

* **For an already sorted array:** Since Selection Sort scans the entire array and compares elements in each iteration, the algorithm still performs the same number of comparisons and swaps. However, because the array is already sorted, no swaps are required, resulting in a best-case scenario. In this case, the actual running time of Selection Sort on an already sorted array will be shorter than the worst-case scenario.
* **For a reversely sorted array:** Selection Sort still performs the same number of comparisons and swaps as in the general case. The number of comparisons and swaps is maximum when the array is reversely sorted. Hence, the actual running time will be closer to the worst-case scenario, and it will take longer compared to an already sorted array

1. **Merge sort:** Merge Sort has a time complexity of O(n log n) in all cases. This means that as the input size increases, the running time of the algorithm grows logarithmically

* ­**for an already sorted array:** Merge Sort utilizes a divide-and-conquer strategy, splitting the array into halves and then merging them. In the case of an already sorted array, the merge step becomes simpler and more efficient since the subarrays are already sorted. This results in faster execution compared to a random or reversely sorted array. The actual running time will be shorter than the worst-case scenario.
* **for a reversely sorted array:** Merge Sort still performs the same number of comparisons and merging operations as in the general case. Although the merge step is not affected by the reversely sorted order, the divide step will continue to split the array into smaller subarrays. The actual running time will be closer to the worst-case scenario but still faster than a quadratic time complexity algorithm like Selection Sort

**Time complexity analysis:**

**Insert: bigO(n)**

* Best case: O(1), is the priority queue is empty, the new node becomes the head, and the operation takes constant time.
* Worst case: O(n), if the priority queue is not empty and the new node has the highest priority it needs to be inserted at the end of the linked list. This requires traversing the entire list, resulting in a time complexity of O(n), where n is the number of elements in the priority queue.
* Average case: O(n), the new node need to be inserted in different positions depending n its priority.

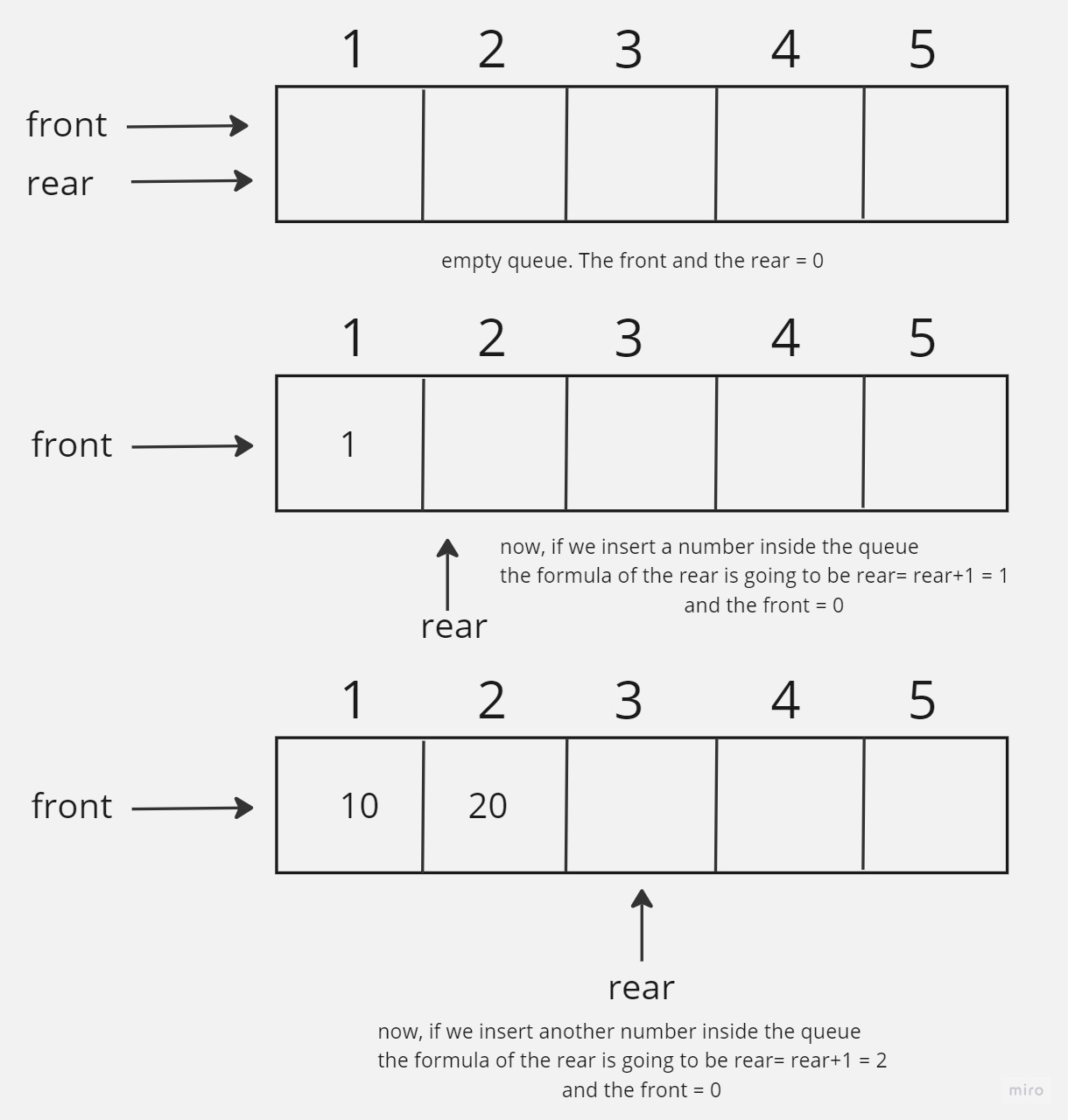
**getMain: bigO(1)**

* Best case: O(1), retrieving elements with the minimum priority involves accessing the head node, which takes constant time.
* Worst case: O(1), the time complexity of getting the minimum element remains constant as it only requires accessing the head of the node
* Average case: O(1), the average case time complexity is also O(1) as the minimum element is always stored in the head node

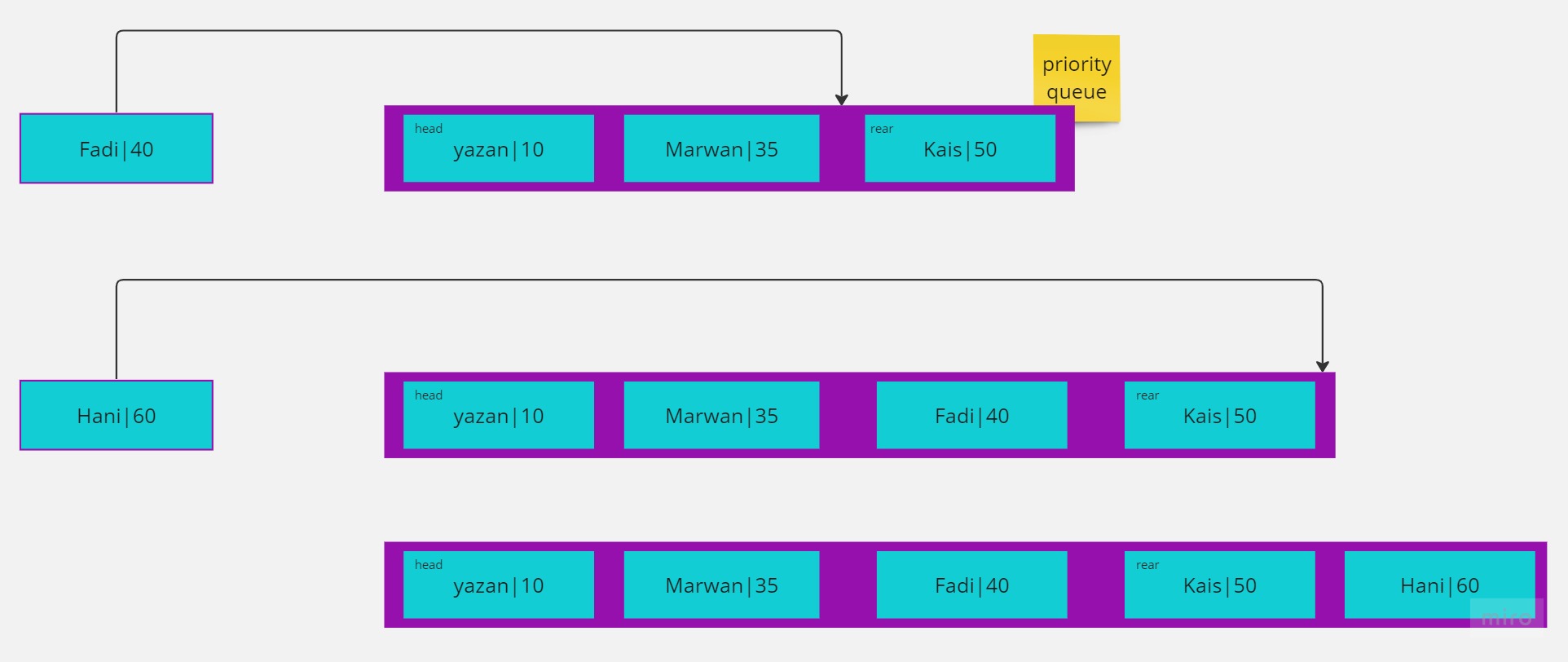
**RemoveMain: bigO(1)**

* best case: O(1), removing the element with the minimum priority involves updating the head node, which takes constant time.
* Worst case: O(1), the time complexity of removing the minimum element remains constant as it only requires updating the head of the node.
* Average case: O(1), the average case time complexity is also O(1) as removing the minimum element involves the head node, which is a constant time operation.

**Q2**



The attached pictures shows how the priority queue look like when its empty and how it looks like when implementing values in it.



The attached photo shows when adding an item to a full queue, the item will be inserted in the queue at the appropriate position based on its value.

**Part 2:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Selection sort** | 5000 | 50000 | 500000 |
| Sorted | Time to sort array 5000 sorted using selection sort in millis: 8 | Time to sort array 50000 sorted using selection sort in millis: 695 | Time to sort array 500000 sorted using selection sort in millis: 449013 |
| reversely sorted | Time to sort array 5000 unsorted using selection sort in millis: 21 | Time to sort array 50000 unsorted using selection sort in millis: 2608 | Time to sort array 500000 unsorted using selection sort in millis: 2105629 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Merge sort** | 5000 | 50000 | 500000 |
| Sorted | Time to sort array 5000 sorted using merge sort in millis: 2 | Time to sort array 50000 sorted using merge sort in millis: 13 | Time to sort array 500000 sorted using merge sort in millis:57 |
| Reversely sorted | Time to sort array 5000 unsorted using merge sort in millis: 1 | Time to sort array 50000 unsorted using merge sort in millis: 12 | Time to sort array 500000 unsorted using merge in millis: 36 |

**Q4:**

|  |  |  |
| --- | --- | --- |
| **Time complexity of (big-O)** | Selection sort | Merge sort |
| Best case | O(n^2) | O(n log n) |
| Worst case | O(n^2) | O(n log n) |

**Q5:**

**Comparing the performance on sorted algorithms:**

* **Selection sort**When the data is already sorted, the selection sort algorithm still requires the same number of comparisons and swaps as it would in the worst-case scenario. This is because the algorithm does not exploit the fact that the data is sorted and continues to search for the minimum element in each iteration. As a result, the time complexity of selection sort remains O(n^2) in all cases, regardless of the order of the input.
* **Merge sort**: It takes advantage of the sorted order of the input, making it highly beneficial in terms of performance. When the data is already sorted, the algorithm utilizes a divide and conquer strategy to avoid unnecessary comparisons and optimizations. This results in simplified merging steps, leading to faster execution. Despite the sorted data, the time complexity of merge sort remains O(n log n) in all cases, indicating its efficiency

**Comparing the performance on reversely sorted data:**

* Selection sort: it perform the same weather the data is sorted or reverse. It still requires the same number of comparisons and swaps in both cases. However, the time complexity on reversely sorted data is also O(n^2).
* Merge sort: its performance unaffected by the reverse order of the input. It still recursively splits he data and merges the sorted sub problems, resulting in a time complexity of O(n log n) even in reversely sorted data.
* In conclusion, merge sort is better than selection sort in terms of the time complexity. In addition, it has a significantly better worst case time complexity of O(n log n) compared to selection sorts that has O(n^2). Merge sort performance is consistent regardless the input order, whereas selection sort performance remains the same regardless of wather the data is sorted or reversely sorted.

**Q7:**

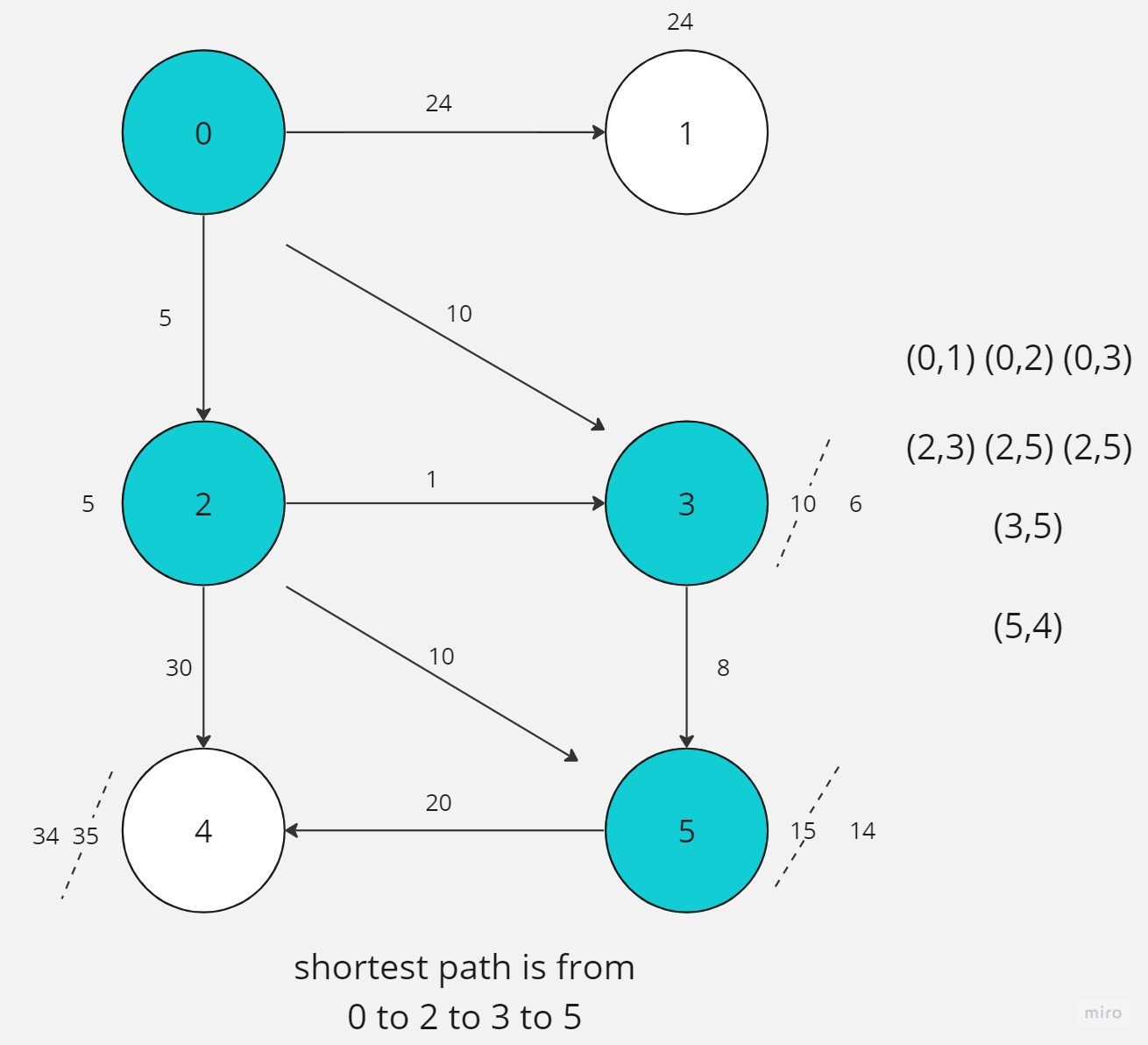
There is another measure that can be used to compare the efficiency of algorithms, which is space complexity. Space complexity refers to the amount of memory or space that required by an algorithm to solve a problem. Thus, it can be solved by using one of the various ways like:

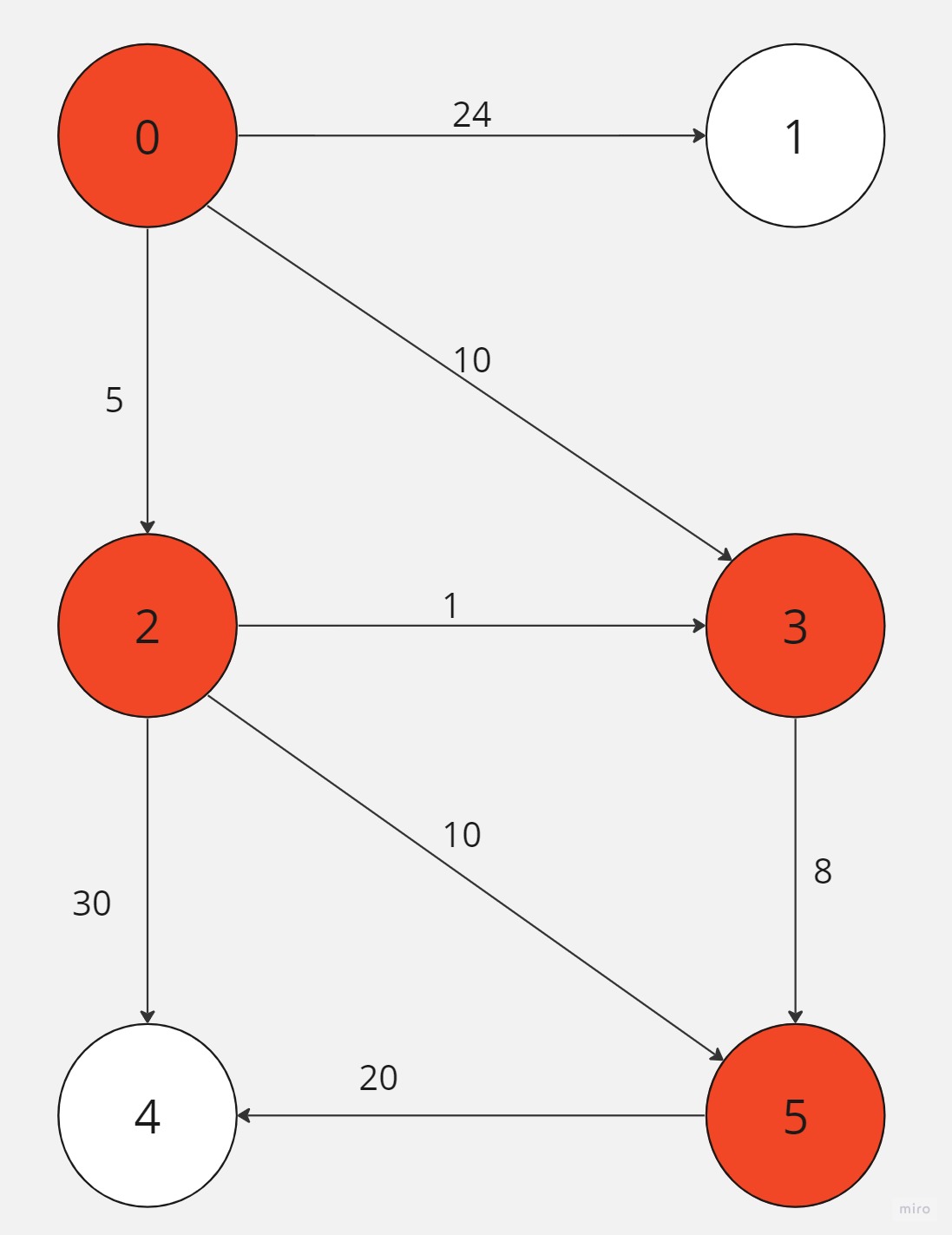
* **In-place storing:** Some sorting algorithms, such as bubble sort and selection sort, can sort the input data directly without using any extra memory apart from the original array. These algorithms are called in-place sorting algorithms and only require a constant amount of additional space. (parkhar, 2022)
* **Additional data structures:** When using sorting algorithms like merge sort or quick sort, sometimes we need extra storage, like temporary arrays or stacks, to help with the sorting. The space complexity of these algorithms considers the space used by these extra storage structures. (geeks, 2022)
* **Auxiliary space:** It is the extra memory that an algorithm uses, aside from the input. It includes memory for variables, data structures, and recursive functions. It helps us know how much extra memory the algorithm needs. (OREILLY, 2023)

**Q8:**

**Bellman-ford algorithm:-**

**Went into 5 iteration**





**Shortest path is: 0,2,3,5**

**Part 3:**

**Q11:**

1. scanner = Scanner()
2. word = scanner.getInput()
3. reversedWord = ""
4. stack = Stack()
5. for i in range(len(word)):
6. stack.push(word[i])
7. while not stack.isEmpty():
8. reversedWord += stack.pop()
9. if reversedWord == word:
10. print("palindrome")
11. print("not a palindrome")

**Q12:** The data structure used in question number 11 is a stack. Stack is an abstract data type that represents a group collection of elements where the last element added is the first one to be removed, in which it follows the last in fist out.

The valid operations on a stack ADT are:

* Push (item): adds an item to the top of the stack
* Pop(): removes and return the topmost item of the stack
* Peek(): returns the topmost item of the stack
* isEmpty(): checks if the stack is empty
* size(): returns the number of elements in the stack

Scenario: palindrome checking. The operations of the stack ADT in the context of checking a string is a palindrome or not.

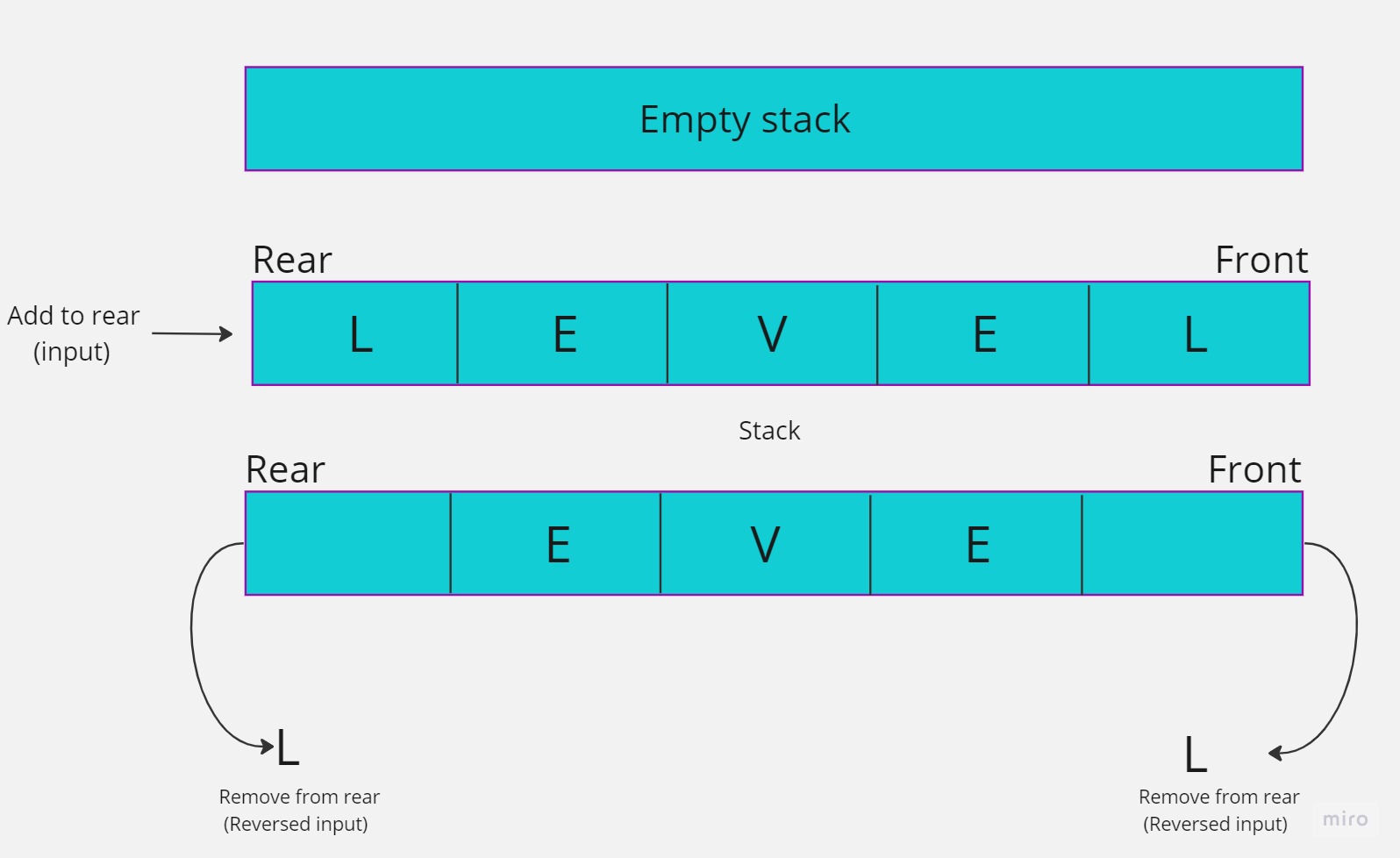
1. Create an empty stack
2. Push each character of the input onto the stack

* Stack: [ ]
* Input = “ level”
* Stack: [‘l’, ‘e’, ‘v’, ‘e’, ‘l’]

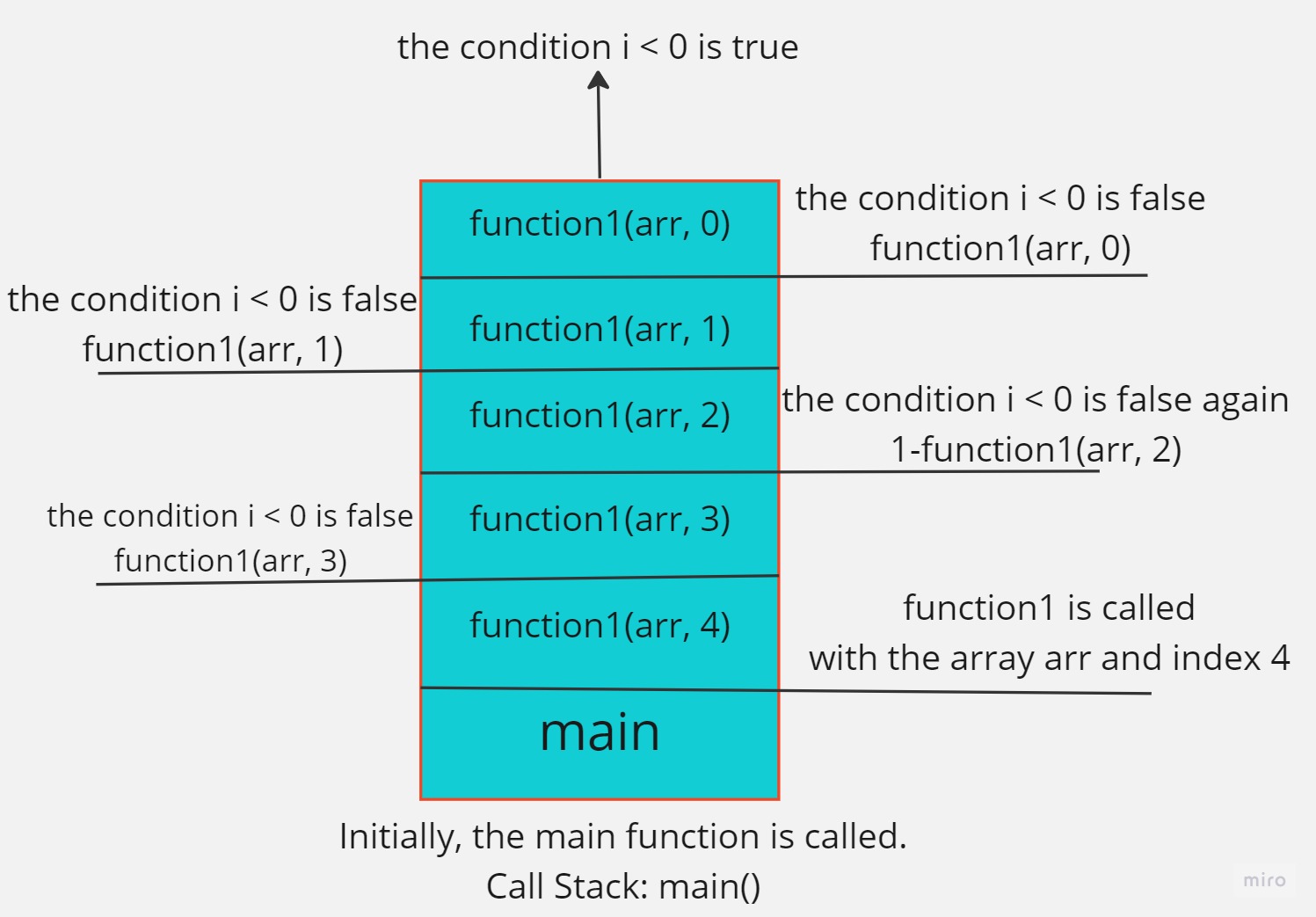
1. Pop characters from the stack and concatenate them to form the reversed input

* Stack: [‘l’, ‘e’, ‘v’, ‘e’, ‘l’]
* Reversed input: “l”
* Stack: ['e', 'v', 'e', 'l']
* Reversed Input: "le"
* Stack: ['v', 'e', 'l']
* Reversed Input: "lev"
* Stack: ['e', 'l']
* Reversed Input: "leve"
* Stack: ['l']
* Reversed Input: "level"

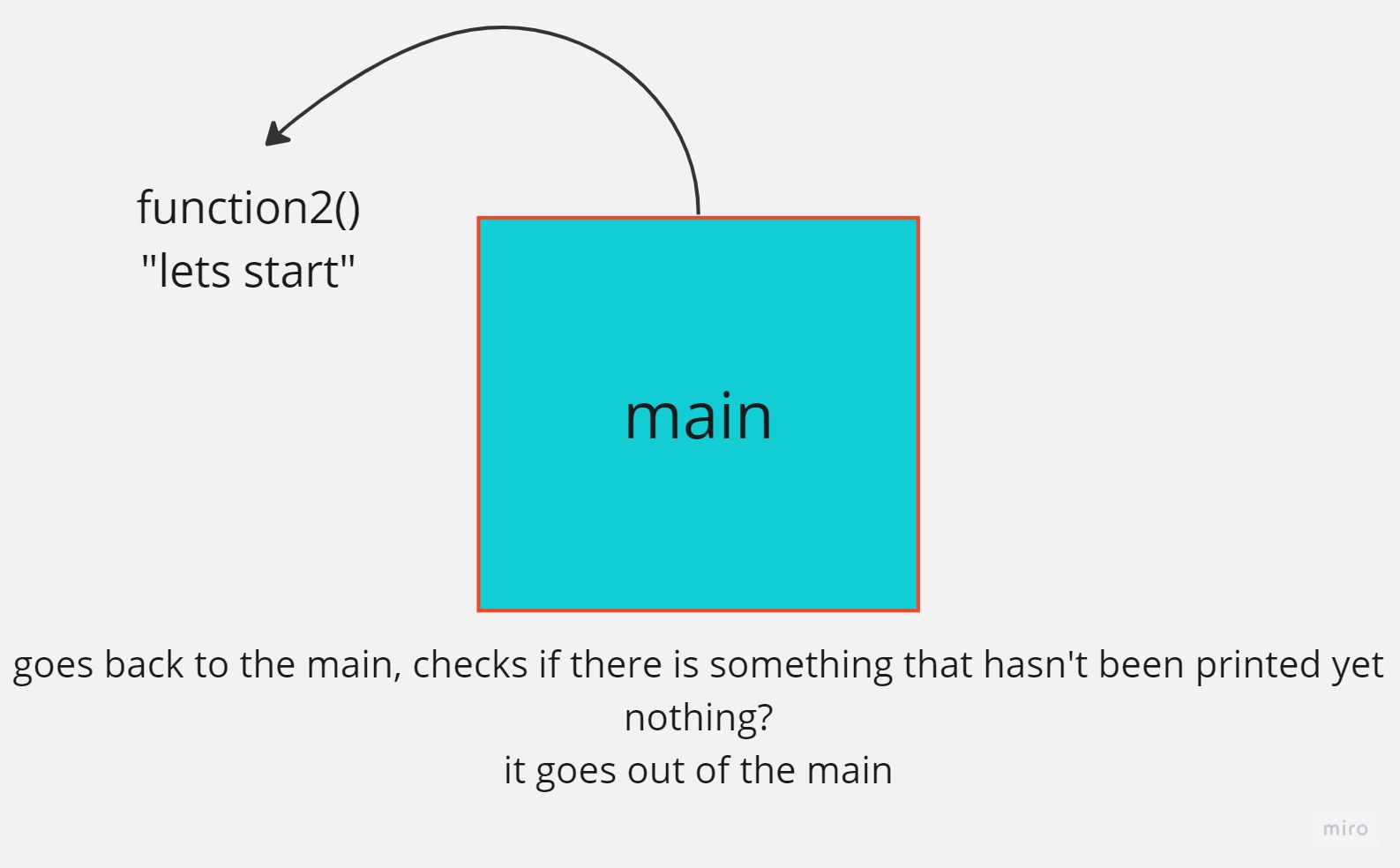
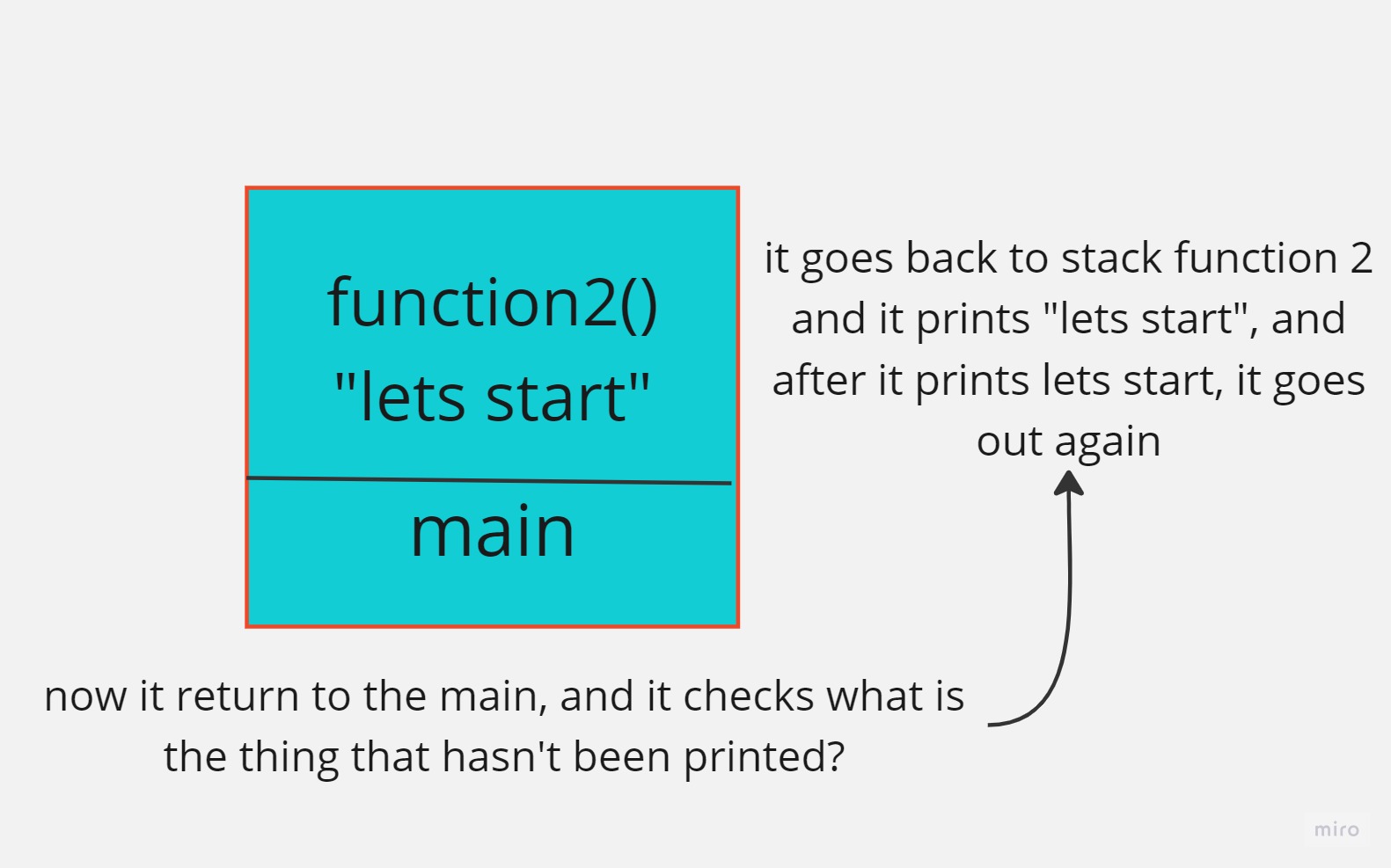
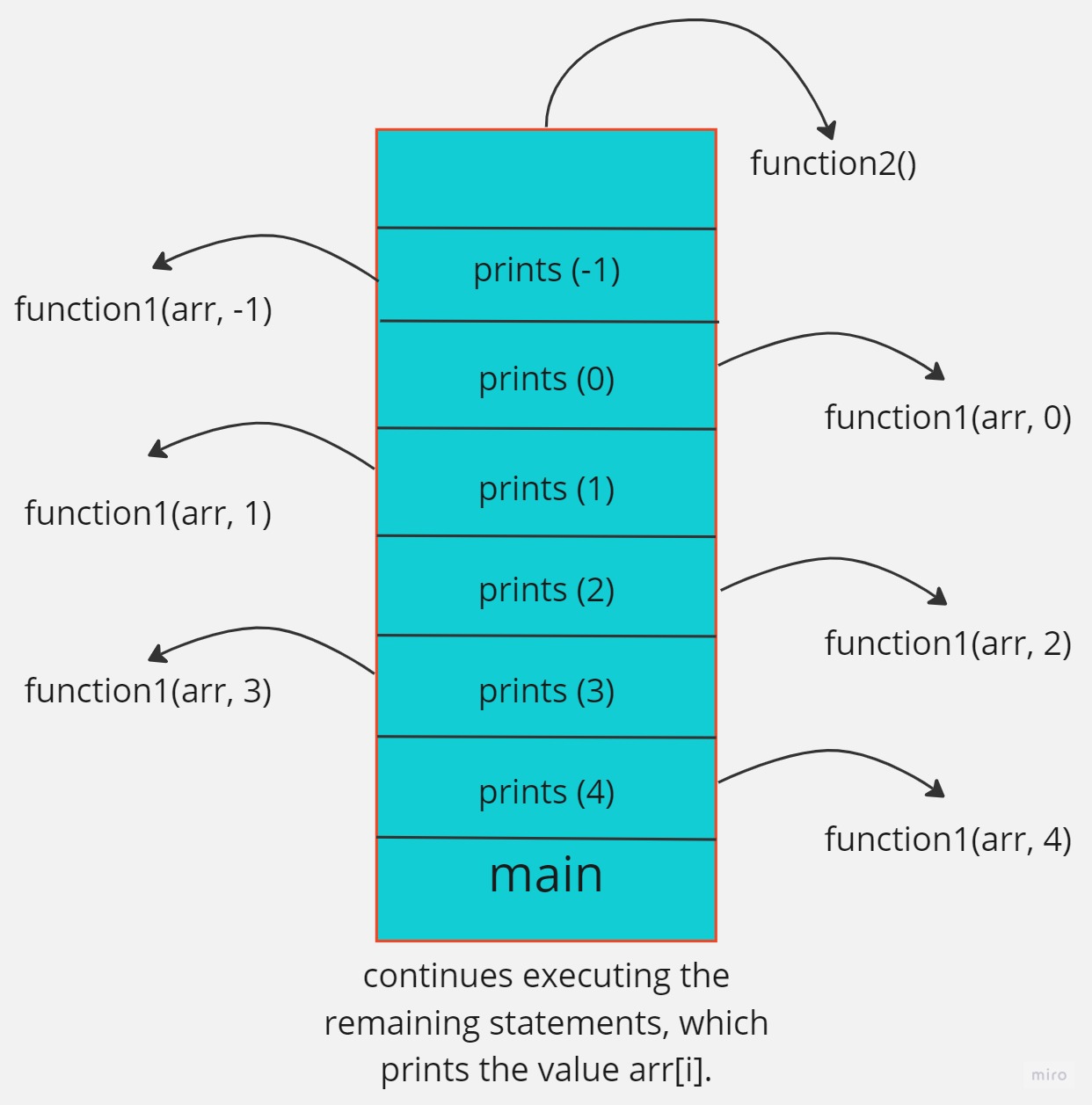
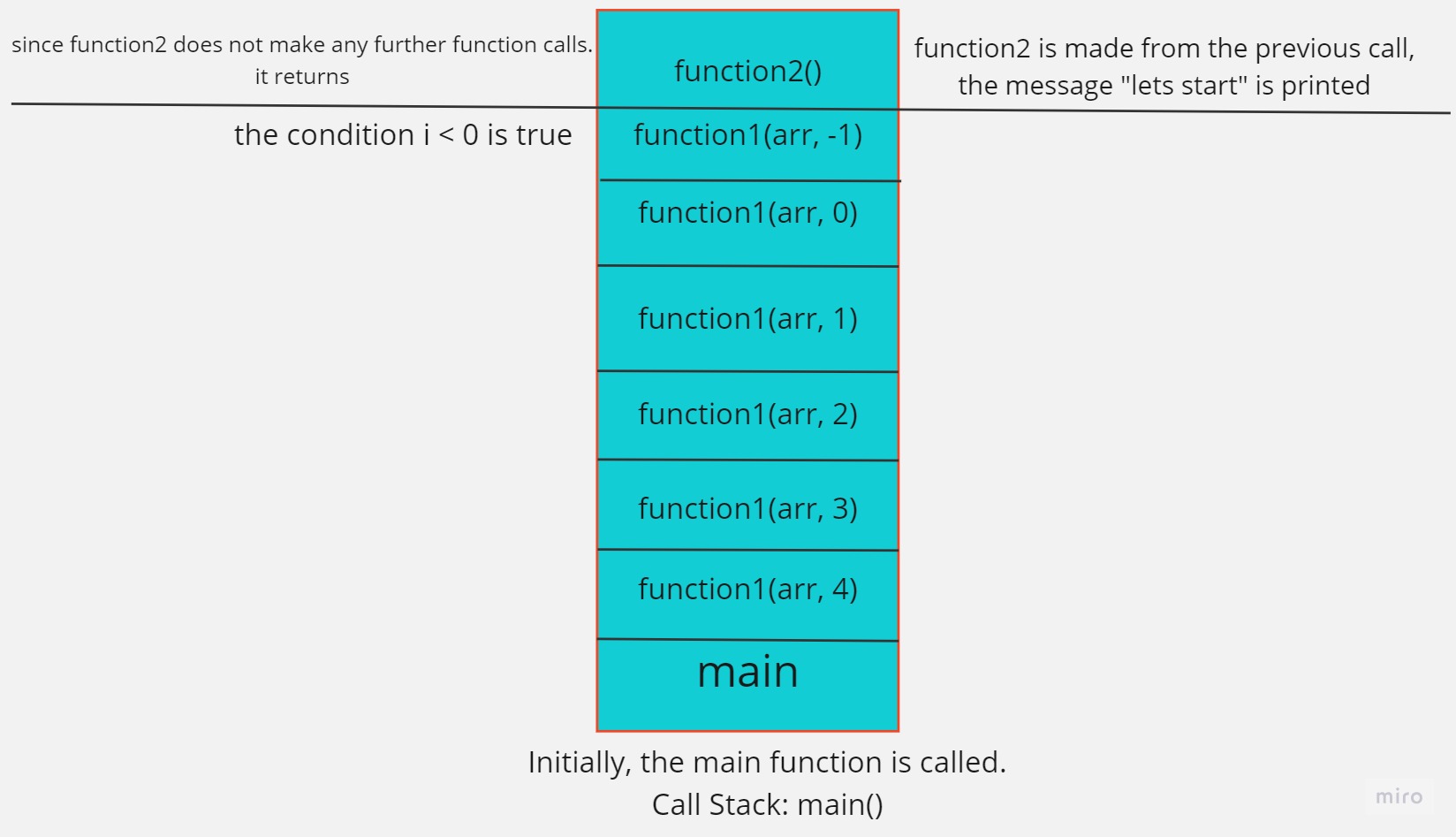
The following picture shows how the stack works.



**Q13:** public static void function2(){system.out.println ("lets start")}public static void function1 (int arr[]. int i){if (i<0){function2;}else{function1 (arr, i-1);system.out.println (arr [i])}public static void main (Strings [] args){int arr [] = {1,2,3,4 ,5};function1 (arrr,4);function2 (arr);}



The condition I<0 is true when -1 occurs, so it becomes function1 (arr,i-1) where I =0



**Part 4:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linked list | Array unsorted | Array sorted |
| Search | O(n) | O (n) | O (log n) |
| Insert | O(n) | O(n) | O(n) |
| Remove | O(n) | O(n) | O(n) |

**Q14:**

Spell checker application: for this scenario, a suitable data structure to store the dictionary of words would be the(prefix tree), it allows for efficient searching and suggestions of words based on prefix. Each node in the Trie represents a letter, and the edges represents the connections between to form words. In which, by traversing the Trie based on the letters of the typed word, the spell checker can quickly determine if the word exists in the dictionary. However, if the word is not found, it can efficiently suggest a similar words by traversing the Trie from corresponding prefix. The time complexity for searching and suggestion words in a Trie is typically O(k), where K is the length of the word being searched.

Priority queues: for implementing efficient priority queues, a suitable data structure would be the binary heap for a Fibonacci heap. Both of these data structure offer efficient insertion and deletion operations in the worst case.

* Binary heap provides O(log n) time complexity for both insertion and deletion of elements. They are easy to implement and maintain and are suitable for scenarios where the ordering of is based on their priority value.
* Fibonacci heap offer even better worst case time complexity for some operations, such as O(1) for insertion and O(log n) for deletion. In addition, they are particularly useful when there are frequent insertion and deletions of elements in a priority queue

**Q15:**

public static int fun1(int [] arr, int index) { If (index <=0) { Return arr[0]; } int m1 = fun1 (arr, index -1); int m2= fun1(arr, index-2); int m3= fin1(arr, index-4); If (m1>m2) { Return m1; } Else if (m2>m3) { Return m3; } Else { Return m1; } }

To determine the time complexity of a recursive function, we need to analyze each component of the function individually in order to assess its runtime. These components aim to simplify complex problems by breaking them down into smaller, more manageable ones. They achieve this by recursively calling themselves according to a defined pattern. To evaluate the time complexity, the user provides the function with two parameters: an array and an index. The recursive calls within the function are the primary factors to consider when estimating the runtime.Let us assume the initial value of the index is 'n' and track the calls accordingly to determine the time complexity for each call. We examine each call within the recursive function's sequence to assess its time complexity. The initial call of the main function has a time complexity of T(N). The second call, which occurs inside the recursive function, has a time complexity of T(N-1). The third call follows the same pattern, with a time complexity of T(N-2). This pattern continues with the fourth call having a time complexity of T(N-3). We repeat this process until we reach the best case. By analyzing the time complexity of each call in the sequence, we can estimate the total time required to execute the function and address the given problem.

We must consider the time required for any conditions or variable initialization, which can be expressed as T(N) = O(1), in addition to the recursive calls.By examining each component of the code and calculating the time complexity for each call, we can estimate the total time required to execute the function and solve a specific problem.The time complexity, T(N), can be expressed as T(N) = T(N) + T(N-1) + T(N-2) + T(N-4) + O(1) by summing the time complexities of each call. In the worst-case scenario, there are three function calls for each call made. Considering that there will be approximately 3N calls to function1, we can estimate the time complexity of this function to be approximately 3N based on this observation. From this approximation, we can deduce that the time complexity of this recursive function is roughly O(3^n).

**Q16**:

**The advantages of encapsulation and information hiding when implementing the stack:-**

The utilization of encapsulation and information hiding in implementing a stack offers numerous benefits. These include modularity, abstraction, data protection, code reusability, maintainability, ease of debugging, and future flexibility. These techniques enhance the stack implementation, making it stronger, more efficient, and adaptable. By applying the principles of encapsulation and information hiding, developers can create a well-organized, dependable, and scalable stack implementation that contributes to the overall achievement of software development projects.  
(Cosentino, 2019)

Other data structure as ADT:-

1. **Modularity and code organization**: Encapsulation enables the containment of the inner workings of a data structure within a clearly defined interface. By establishing a distinct separation between the implementation and the interface, it simplifies the structuring and control of code. Additionally, the Abstract Data Type (ADT) offers a broader perspective of the data structure, facilitating comprehension and upkeep.(niharika123, 2020) (geeks, 2023)
2. **Information hiding:** Encapsulation allows for the practice of concealing information, which involves limiting direct entry to the internal state of an object. In the case of a stack Abstract Data Type (ADT), this implies that the internal implementation specifics, like the array or linked list employed to store the elements, are concealed from ADT users. This safeguard shields the data structures from unintended alterations and guarantees the preservation of its integrity. (tylor & taylor, 2022)
3. **Flexibility and extensibility:** Encapsulation permits the modification of the internal implementation of a data structure without impacting the code utilizing the Abstract Data Type (ADT). As long as the interface remains unchanged, users can seamlessly interact with the data structure. This flexibility empowers us to optimize the implementation or substitute it with a more efficient one, all while ensuring that the existing codebase remains unaffected. (hillard, 2019)
4. **Security and data integrity:** Encapsulation enhances security by limiting direct entry to internal data and operations, safeguarding against unauthorized modifications to the data structure. It ensures that only appropriate operations are performed, promoting data integrity and minimizing the likelihood of errors or unintended side effects. (Gaithersburg, 2022)

**Q17:**

The view that imperative ADTs serve as a basis for object orientation is a valid perspective, and I agree with this view due to object oriented programming that has its roots in procedural programming and imperative ADTs, and it builds upon the concepts and principles of imperatives programming.

Some justifications for this view are:

1. **Data abstraction:** Imperative Abstract Data Types (ADTs) facilitate data abstraction through the establishment of a distinct division between the interface and the implementation. This segregation enables ADT users to interact with the abstracted data and execute operations without requiring knowledge of the underlying specifics. Object-oriented programming builds upon this concept by introducing classes and objects, where data and behavior are combined into reusable entities. (wiki, 2023) (junker, 20218)
2. **Inheritance and polymorphism:** Imperative Abstract Data Types (ADTs) possess the ability to be extended and specialized through inheritance, a fundamental characteristic of object-oriented programming (OOP). In OOP, inheritance facilitates the creation of class hierarchies, enabling subclasses to inherit and expand upon the behavior of their parent classes. This concept of building upon existing functionality finds its roots in imperative ADTs, where diverse data structures can also be derived from a shared base. (Library, unkown)

**Q18: \*\***

By leveraging implementation independent data structures like stacks and queues, developers can write code that is more portable, adaptable and maintainable. These data structures provides a level of abstraction that simplifies programming tasks, promotes code reuse, and enables interoperability across different platforms and environments.

**Abstraction**: Implementation-agnostic data structures introduce a layer of abstraction between the user and the underlying implementation of the data structure. This implies that the user does not need to be aware of how the data structure is implemented, making it easier to comprehend and utilize. For instance, when using a stack, the user can perform operations like push and pop without concerning themselves with whether the stack is implemented using an array or a linked list.**Portability**: Implementation-agnostic data structures are applicable across different programming languages and platforms. This enhances their portability compared to implementation-specific data structures, which are limited to a particular programming language or platform. For instance, the stack data structure can be implemented in any programming language that supports object-oriented programming, enabling its usage in various languages like Java, C++, and Python.**Flexibility**: Implementation-agnostic data structures can be easily modified to cater to the specific requirements of a particular application. This grants them more flexibility than implementation-specific data structures, which are typically rigid in their implementation. For instance, the stack data structure can be effortlessly adapted to handle different data types such as integers, strings, or objects, making it versatile and adaptable to diverse applications.