A picture containing logo

Description automatically generated

Data Structures & Algorithms

T/618/7430

40201201

**Section ()**

**Submitted To:**

Dr. Malek al-lozi

**Submitted By:**

Farouq Hassan

Spring 2023

**Table Of Contents**

* 1. Part 1 …………………………………………………….………………. (2)
  2. 1 …………………………………………………………………….(2)
  3. 2…………………………………………………………………….(3)

2.1. Part 2 …………………….………………………… (3)

2.1. 4 …………………….…………………….………………………… (5)

2.2. 5 …………………….………………………………………………… (6)

2.3. 6 …………………………………………….………………………… (7)

2.4. 7………………………………………………………………………. (7)

2.5. 8………………………………………………………………………. (8)

2.6. 9 …………………….…………………….………………………… (8)

2.7. 10…………………….………………………………………………… (8)

3.1. Part 3 …………………………….…………………….……… (12)

3.1. 11 …………………….…………………….………………………… (12)

3.2. 12 …………………….………………………………………………… (13)

3.3. 13 …………………………………………….………………………… (15)

4.0. Part 4 …………………….…………………………………… (17)

4.1. 14 …………………….…………………….………………………… (17)

4.2. 15 …………………….………………………………………………… (18)

4.3. 16 …………………………………………….………………………… (18)

4.4. 17……………………………………………………………………….(19)

4.5 18……………………………………………………………………….(20)

10.1. Resources ………………………….……………….…... (21)

**------------------------------------------------------------------------------**

**Part 1**

**Q1:**

Design Specification for Min-Priority Queue using linked list: it’s a data structure, it has a node take two elements the data and the priority, every element on it has a priority value and it’s represent the priority depend on the other priorities, it keep that the minimum priority in front of queue, and it has insert value and priority and get to show a output depending on the priority. And remove to get rid of the head node, it has a common application in our lives it could be used in a job scheduling system, when you have many tasks in different priorities, so the highest priority goes with the lowest priority to be done first.

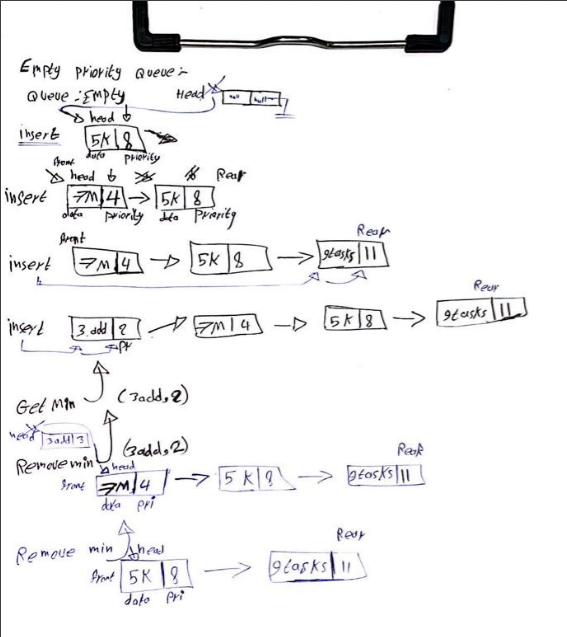
Design Specification for Min-Priority Queue using linked list:

1. Abstract Data Type (ADT): the min priority queue its just a abstract data type, and just for every element have a priority value it can work front of it, its ensure that the min priority in front of it, and you can insert depend on the priority, and back the minimum priority and remove priority, the nodes
2. Data Structure: it’s the main data structure that is used on it, and it’s a singly on it, and in the linked list every element must have data and priority.

And we have the operations:

1. Insert: it is including because the priority value needs to be inserted, it needs to insert a data based on its priority value, and need two inputs the data and the priority of it and all of that for the new element not old one, need to create a new node for the data and priority, when it takes the elements if it was empty or the new node have less priority than the head, need to set it as the new head and make it point to the last head, or else it pass all the linked list until it find the right priority, turn it between the nodes when it less than the next one, and the time complexity is O(n) in the worst case, because it will go in all of them to the end of the list.
2. Get: return output the minimum priority of the whole data, and without deleting or editing on the data just output, so if it wasn’t empty return the head node, else point its empty, the time complexity is O (1) because it can be done in constant time.
3. Remove: when it asks to remove, it removes the minimum priority of the whole data, so if It wasn’t empty, store the head as a minimum value and point the head into the next node, time complexity is O (1) because it can be done in constant time.

**Q2:**

****

**Q3: Code**

**Part 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 5000 | 50000 | 500000 |
| Selection sort | sorted | 10ms | 245ms | 22186ms |
| Selection sort | Reversely sorted | 25ms | 2078ms | 212727ms |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | 5000 | 50000 | 500000 |
| Merge sort | Sorted | 2ms | 6ms | 50ms |
| Merge sort | Reversely sorted | 3ms | 4ms | 51ms |

**Q4:**

|  |  |  |
| --- | --- | --- |
|  | Selection sort | Merge Sort |
| Best case | O(n^2) | O(n log n) |
| Worse case | O(n^2) | O(n log n) |

**Q5:**

We may evaluate the effectiveness of the two sorting algorithms on sorted data and reversely sorted data based on the findings I provided:

The selection sort algorithm takes 10ms to sort the sorted array of the size 5000, 245ms for the size 50000, and 22186ms for the size 500000.

The selection sort algorithm takes 25ms to sort the reversely sorted array of size the 5000, 2078ms for the size 50000, and 212727ms for the size 500000.

The merge sort algorithm takes 2ms to sort the sorted array of the size 5000, 6ms for the size 50000, and 50ms for the size 500000.

The merge sort algorithm takes 3ms to sort the reversely sorted array of the size 5000, 4ms for the size 50000, and 51ms for the size 500000.These results show that both techniques outperform reversely sorted data when applied to already sorted data. However, there is a sizable performance gap between the two techniques.

In both the best and worst scenarios, the temporal complexity of Selection Sort is O(n^2). As a result, sorting the reversely sorted arrays takes a long time, especially for bigger sizes. The technique generates a large number of operations since it necessitates several comparisons and swaps for each element.

Merge Sort, on the other hand, has an O(n log n) time complexity in both the best and worst scenarios. Even on data that has been sorted in reverse, it consistently outperforms Selection Sort. The divide-and-conquer strategy used by Merge Sort, in which the array is split into smaller subarrays and then merged back together, results in more effective sorting. The subarrays are successfully combined during the merge stage, which speeds up sorting in general.

In conclusion, when sorting both sorted and reversely sorted data, Merge Sort is often preferred over Selection Sort. Due to its O(n log n) time complexity, it performs more consistently and effectively. Both methods may be used for sorting, although Merge Sort performs better in terms of time complexity and practical performance than Selection Sort.

**Q6:**

An understanding of an algorithm's performance scaling with input size is provided by its theoretical time complexity. The theoretical temporal complexities of Selection Sort and Merge Sort are as follows:

Selection Sort:

Best Case: O(n^2)

Worst Case: O(n^2)

Merge Sort:

Best Case: O(n log n)

Worst Case: O(n log n)

Based on these time complexities, we can now explain the results obtained.

Sorted Data:

When the input data is already sorted, both algorithms exhibit their best-case performance. In this scenario, Selection Sort requires O(n^2) operations regardless of the initial order, resulting in relatively consistent running times. Merge Sort, on the other hand, performs significantly better with its O(n log n) time complexity. The divide-and-conquer approach of Merge Sort allows it to divide the sorted array into smaller subarrays efficiently, resulting in faster sorting times.

Reversely Sorted Data:

When the input data is reversely sorted, the worst-case scenario for both algorithms is encountered. Selection Sort, with its time complexity of O(n^2), requires a large number of comparisons and swaps for each element. As the size of the input increases, the number of operations grows exponentially, leading to significantly longer running times. Merge Sort, with its O(n log n) time complexity, still outperforms Selection Sort on reversely sorted data. While Merge Sort requires more operations than in the sorted case, it scales better due to the efficient merging of sorted subarrays.

In conclusion, the theoretical time complexity offers a foundation for comprehending how different sorting algorithms function. Both Selection Sort and Merge Sort display their anticipated time complexities on sorted and reversely sorted data, however Merge Sort is better at handling big inputs due to its greater time complexity. Merge Sort frequently beats Selection Sort in terms of actual running times, especially as the input amount rises.

**Q7:**

There is other options to figure out that the efficiency between two algorithms and one of these options the space complexity, and it’s the total space taken by the algorithms and it can be different depend on the used language or the machine who runs the algorithm or even the compiler its self, with the depend on the input size, and its includes two of the space that used by the input and the auxiliary space.

**Q8:**

This is a simulate the operations of Dijkstra's algorithm and the Bellman-Ford algorithm on the given graph to find the shortest path from vertex 0 to vertex 5.

Graph:

0 >>> 1 with weight 24

0 >>> 2 with weight 5

0 >>> 3 with weight 10

2 >>> 3 with weight 1

2 >>> 4 with weight 30

2 >>> 5 with weight 10

3 >>> 5 with weight 8

5 >>> 4 with weight 20

Dijkstra's Algorithm:

Set the distance of vertex 0 to 0 and all other vertices to infinity.

Set the previous node of all vertices to null.

Select vertex 0 as the starting vertex.

Visit vertex 0.

Update the distances of its neighbors:

Update vertex 1: Distance from vertex 0 to vertex 1 is 24.

Update vertex 2: Distance from vertex 0 to vertex 2 is 5.

Update vertex 3: Distance from vertex 0 to vertex 3 is 10.

Visit vertex 2 (closest unvisited vertex).

Update the distances of its neighbors:

Update vertex 3: Distance from vertex 0 to vertex 3 (via vertex 2) is 6.

Update vertex 4: Distance from vertex 0 to vertex 4 (via vertex 2) is 35.

Update vertex 5: Distance from vertex 0 to vertex 5 (via vertex 2) is 18.

Visit vertex 3 (closest unvisited vertex).

Update the distance of its neighbor:

Update vertex 5: Distance from vertex 0 to vertex 5 (via vertex 3) is 14.

Visit the remaining vertices:

Visit vertex 1.

Visit vertex 4.

The shortest path from vertex 0 to vertex 5 using Dijkstra's algorithm is: 0 -> 2 -> 3 -> 5, with a total weight of 14.

Bellman-Ford Algorithm:

Set the distance of vertex 0 to 0 and all other vertices to infinity.

Set the previous node of all vertices to null.

Relax all edges |V| - 1 times.

Update the distances of all vertices based on the edges:

Relax edge (0, 1): The distance from vertex 0 to vertex 1 is 24.

Relax edge (0, 2): The distance from vertex 0 to vertex 2 is 5.

Relax edge (0, 3): The distance from vertex 0 to vertex 3 is 10.

Relax edge (2, 3): The distance from vertex 2 to vertex 3 is 6.

Relax edge (2, 4): The distance from vertex 2 to vertex 4 is 35.

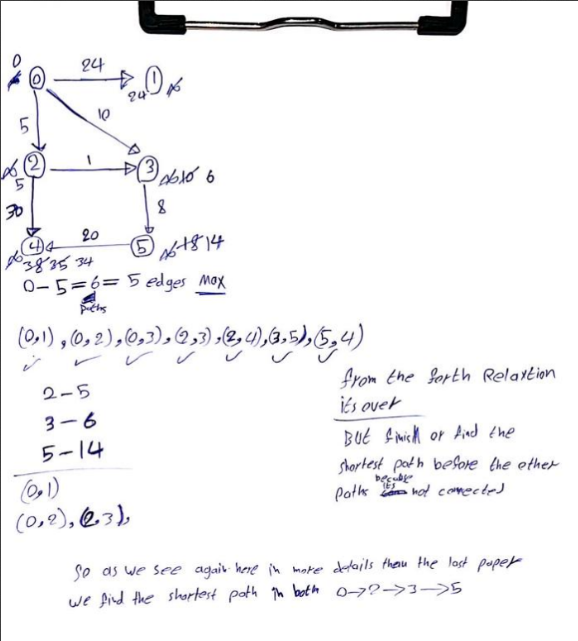
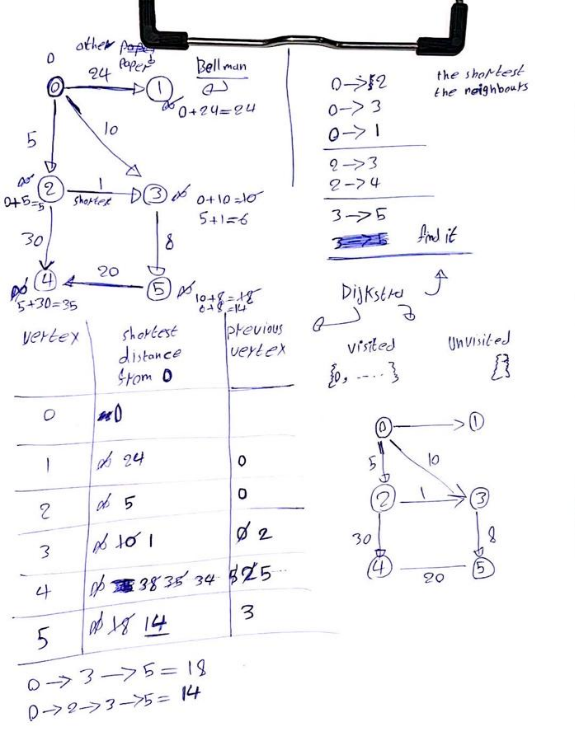
Relax edge (3, 5): The distance from vertex 3 to vertex 5 is 8.

Relax edge (5, 4): The distance from vertex 5 to vertex 4 is 20.

Check for negative weight cycles. (Not applicable in this case)

The shortest path from vertex 0 to vertex 5 using the Bellman-Ford algorithm is: 0 -> 2 -> 3 -> 5, with a total weight of 14.

In this case, both Dijkstra's algorithm and the Bellman-Ford algorithm yield the same shortest path, which is 0 -> 2 -> 5, but with different intermediate steps, and different time with less time to the Dijkstra algorithm.



**Q9: Code**

**Q10: Code**

**Part 3**

**Q11: pseudocode**

stringToCheck = "abccba"

palindromeString = ""

stack = new Stack<Character>()

// Push each character of the string onto the stack

for i = 0 to stringToCheck.length() - 1:

character = stringToCheck.charAt(i)

stack.push(character)

// Pop each character from the stack and append it to palindromeString

while stack is not empty:

palindromeString = palindromeString + stack.pop()

// Compare original and reversed strings

if stringToCheck equals palindromeString:

print "String is a palindrome"

else:

print "String is not a palindrome"

**Q12:**

Abstract Data Type (ADT): Stack

Formal Definition:

Describes a group of components having Last-In-First-Out (LIFO) behavior. It adheres to a certain ordering rule in which the piece that was most recently introduced is also the first to be deleted. The stack is seen as a vertical structure with things piled on top of one another, seeming like a collection of things.

Valid Operations on Stack:

push(element): Adds the given element to the top of the stack.

pop(): Removes and returns the topmost element from the stack.

peek(): Returns the topmost element of the stack without removing it.

isEmpty(): Checks if the stack is empty.

size(): Returns the number of elements in the stack.

Scenario: Checking Palindrome Using Stack

Consider the scenario where we want to check if a given string is a palindrome using a stack. Let's take the string "abccba" as an example.

Logical Representation:

Start with an empty stack and an empty palindromeString.

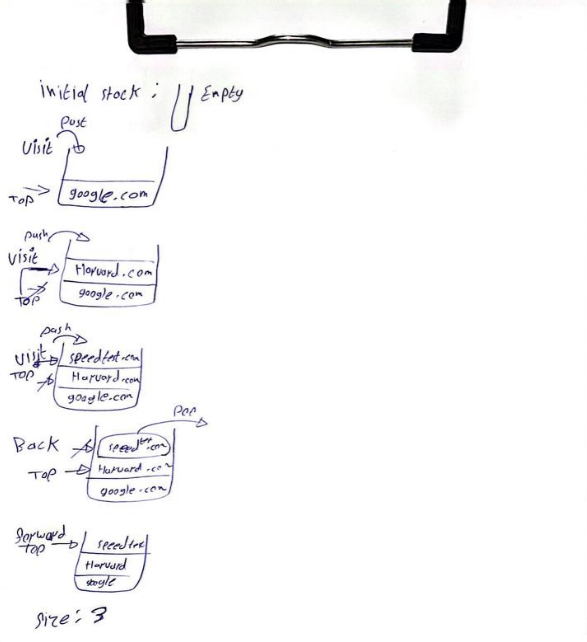
Push each character of the string onto the stack: 'a', 'b', 'c', 'c', 'b', 'a'.

Pop each character from the stack and append it to the palindromeString: 'a', 'b', 'c', 'c', 'b', 'a'.

Compare the original string "abccba" with the reversed string "abccba".

If the two strings are equal, print "String is a palindrome". Otherwise, print "String is not a palindrome".

Here's an illustration of a stack and its operations:



**Q13:**

The call stack plays a crucial role in running the code by keeping track of function calls and their respective execution contexts.

The primary function is called and pushed onto the call stack.

When function1 is called within the main, a new instance of function1 is pushed onto the call stack, creating a new execution context. The current state of the main is saved on the stack.

Inside function1, another call to function1 is made with a different value of i. This new instance of function1 is pushed onto the call stack, and the current state of the previous function1 call is saved.

This process continues as function1 calls itself recursively with decreasing values of i. Each recursive call pushes a new instance of function1 onto the call stack, preserving the execution context of the previous call.

When the base case is reached (i.e., i becomes less than 0), function2 is called. The call to function2 is pushed onto the call stack, and the current state of the previous function1 call is saved.

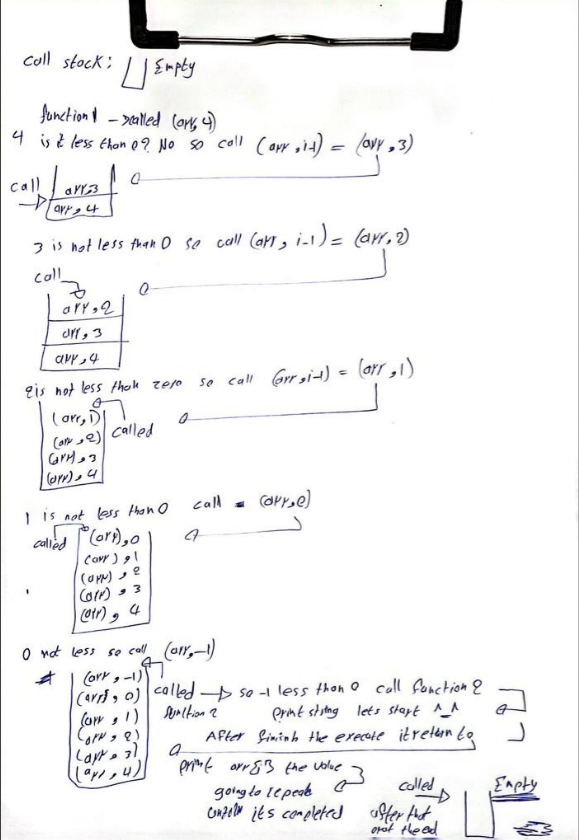
Inside function2, a message is printed, and then function2 returns. The topmost function call (function2) is popped from the call stack, and the execution returns to the previous function1 call.

As each recursive call to function1 reaches its base case, the respective value of arr[i] is printed, and the function returns. The call stack gradually unwinds as each function1 call is completed, returning to the previous function1 call.

Once all the recursive calls to function1 have been completed, the call stack only contains the initial main function call. The code execution continues in the main after the recursive calls.

By using the call stack, the program can manage the execution order of function calls, maintain their respective execution contexts, and correctly return to the appropriate point in the code after each function call completes.

The call stack keeps track of the function calls and their corresponding variables during the execution of the code. It allows for the proper flow of control and helps in managing the recursion in this case.



**Part 14:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linked list | Array unsorted | Array sorted |
| Search | O(n) | O(n) | O(log n) |
| Insert | O(1) | O(1) | O(n) |
| Remove | O(n) | O(n) | O(n) |

**Q14:**

Spell checker application:

Data Structure: Sorted array

Justification: The situation calls for keeping a dictionary on hand and doing a word search. Using an array sorted would enable effective binary search with a time complexity of O(log n), as the words may be stored in sorted order. This would make it possible to quickly search the dictionary for words. By utilizing methods like interpolation search or a modified binary search, the sorted array can also propose words with related prefixes. It is thus a good option for including a spell checker in an application.

Priority queues:

Data Structure: Linked List

Justification: Implementing a priority queue with effective insertion and removal operations is part of the scenario. While an unsorted array might offer insertion with an O(1) time complexity, removal would need an O(n), which is less effective. On the other hand, a sorted array would only require O(log n) time complexity for removal, but O(n) time complexity for insertion because the array must be kept sorted. A linked list is a good option for effectively building priority queues since it offers O(1) time complexity for both insertion and removal. The linked list's head or tail may be used to insert any element, and navigating the list makes it simple to get to the minimum or maximum element. Although a linked list's search operation has an O(n) time complexity, priority queue operations do not need it, making the linked list an effective option.

**Q15:**

So at the first we have two cases and that depend the compare if it less or equal zero and if it check it will return 0 and this means its O(1) in all cases because it’s a constant time, at the second case we have if it was bigger so it will go to the three calls and we can calculate them as their calls because it will be different every time depend on the number, so the first will be O(n-1) the second one O(n-2) the third O(n-3), and at the end of the code we have the comparison and return elements so its O(1) because it’s a constant time, it means to calculate the time complexity for this it has to be as this ((n-1) (n-2) (n-3))

O(1) O(n-1)^3 O(1) so the time complexity of it O(3^n)

**Q16:**

Encapsulation and information hiding offers significant advantages when implementing the stack or any other data structure as an Abstract Data Type (ADT). These benefits can be summarized into three key points.

Firstly, encapsulation provides a layer of security and protection for the data structure. By encapsulating the internal implementation details, access to the data and operations can be restricted to only the necessary interfaces. This prevents unauthorized manipulation of the data, ensuring its integrity. Users can interact with the data structure through well-defined methods, while the implementation details remain hidden, safeguarding the data from unintended modifications or corruption.

Secondly, encapsulation promotes abstraction and simplification. It allows the complexity of the data structure to be hidden, providing users with a high-level interface to interact with. Users don't need to understand the intricacies of the internal workings; they can focus on utilizing the provided methods to perform desired operations. This abstraction simplifies the usage of the data structure, reducing the cognitive load and making the code more readable and maintainable.

Lastly, encapsulation facilitates code modularity and reusability. By encapsulating the data and operations within a self-contained unit, the data structure can be reused in different parts of the code or even in separate projects without modification. The encapsulated unit acts as a black box, providing consistent functionality regardless of the context in which it is used. This promotes modular programming, allowing developers to build on existing implementations and improve code efficiency by avoiding code duplication.

In conclusion, encapsulation and information hiding play a crucial role in implementing data structures as ADTs, such as the stack. The security and protection they provide ensure the integrity of the data, while abstraction and simplification enhance the usability and maintainability of the code. Moreover, encapsulation promotes code modularity and reusability, enabling developers to leverage existing implementations and build robust software systems.

**Q17:**

By looking at how the tenets of Object-Oriented Programming (OOP) connect to ADTs, the idea that imperative Abstract Data Types (ADTs) provide the foundation for object orientation may be supported.

A crucial OOP feature known as polymorphism enables objects to assume many shapes or behaviors. The definition of a common interface for several implementations of the same abstract data structure in the context of ADTs might lead to polymorphism. Because multiple types of stacks, for example, can implement the same set of operations specified by the abstract stack ADT, this promotes code reuse, flexibility, and extension.

Another key concept in OOP is abstraction, which entails separating complicated systems into more manageable, simpler parts. By obscuring implementation specifics and revealing just the fundamental operations and behaviors, ADTs offer a degree of abstraction. An abstract data structure's users do not need to understand how it is internally implemented; instead, they may concentrate on using the supplied interface to carry out required actions. Through this abstraction, modular design, code reuse, and concern separation are all made possible.

The process of inheritance in OOP enables objects to take on traits and behaviors from parent classes. It is possible to use inheritance in the context of ADTs to produce customized variations of an abstract data structure. A "Stack" class, for instance, can be described as an abstract base class having standard stack methods. Subclasses like "ArrayStack" and "LinkedListStack" can then descend from the base class, inheriting the common functions but introducing unique functionality of their own. Modularity, extensibility, and code reuse are encouraged through inheritance.

Another fundamental idea of OOP is encapsulation, which entails combining data and methods into a single unit (object) and regulating access to that unit. ADTs define a particular interface to communicate with the underlying data structure, which serves to naturally encapsulate data and processes. Users can interact with the data structure without having direct access to its internals since the implementation details are wrapped within the ADT. Data integrity, modularity, and the abstraction of implementation details are all guaranteed by encapsulation.

It may be claimed that abstract data structures significantly influenced the concepts and tenets of OOP in response to the issue of whether they were the cause of the development of OOP. ADTs provide a mechanism to focus on the crucial activities while masking implementation details, representing and manipulating data in an organized manner. This idea of abstracting procedures and data fits nicely with the fundamental ideas of OOP, such encapsulation and abstraction. However, it's crucial to keep in mind that OOP covers more abstract ideas than just ADTs, such as the ideas of objects, classes, inheritance, and polymorphism. While abstract data structures laid the foundation for thinking about abstracting data and operations, OOP expanded upon this idea by introducing a holistic approach to software design, focusing on objects, their behaviors, and their interactions.

In conclusion, by aligning with OOP concepts like polymorphism, abstraction, inheritance, and encapsulation, imperative ADTs serve as the foundation for object orientation. Data and actions may be encapsulated using ADTs, which also enable polymorphic behavior, abstract away implementation specifics, and make it easier to reuse code. OOP expands beyond ADTs to include a larger set of ideas and concepts relating to objects, classes, and their interactions, even if ADTs had an effect on how OOP concepts were developed.

**Q18:**

Utilizing implementation-independent data structures, such stacks and queues, has a number of benefits, not the least of which is their capacity to scale efficiently as data volumes increase. These data structures are built to support dynamic resizing and will automatically change their capacity as necessary. This scalability is especially useful in situations where the input size is subject to large fluctuations over time. For instance, depending on user behavior, the amount of incoming messages in a messaging service may rise or fall. Developers may make sure that their programs can effectively process growing volumes of data without generating memory or performance difficulties by employing implementation-independent data structures. Due to their adaptability, programs can easily manage both small and big datasets, allowing for future development without the need for substantial code changes.

The simplicity of testing and debugging associated with implementation-independent data structures is another advantage. These data structures have well defined operations and interfaces that are separate from how they are internally implemented. This separation of the interface from the implementation enables thorough functional testing of the data structure. To make sure that the data structure operates as intended, developers might create test cases that cover a variety of situations and edge cases. Furthermore, the separation of the data structure makes it simpler to troubleshoot any bugs or problems that may occur. Developers may concentrate on the particular data structure and its operations, which makes it easier to identify and fix any problems.

Higher levels of abstraction are offered by implementation-independent data structures, allowing programmers to concentrate on the logical layout of their code. Users of the code are presented with a clear and straightforward interface by developers by obscuring the actual workings of the data structure. The readability and maintainability of the code are both enhanced by this abstraction. Instead of being concerned with the details of its implementation, users simply need to comprehend and interact with the data structure's fundamental activities. The code is simpler to comprehend, less likely to include mistakes, and more likely to be reused as a result of this separation of concerns. Furthermore, if the internal implementation of the data structure has to be improved or changed, it may be done without impacting the external code that depends on it, guaranteeing backward compatibility and reducing the impact on other areas of the program.

**Resources**

(No date a) *Debugging and testing*. Available at: https://www.d.umn.edu/~gshute/softeng/testing.html (Accessed: 18 June 2023).

(No date b) *Code studio*. Available at: https://www.codingninjas.com/codestudio/library/data-structures-for-dictionary-and-spell-checker (Accessed: 18 June 2023).

(No date c) *Code studio*. Available at: https://www.codingninjas.com/codestudio/library/time-and-space-complexities-of-sorting-algorithms-explained (Accessed: 18 June 2023).

*10.4 priority queue abstract data type (ADT) - 10 priority queue abstract data type (ADT) class :* (no date) *Studocu*. Available at: https://www.studocu.com/en-us/document/de-anza-college/beginning-programming-methodologies-in-c/104-priority-queue-abstract-data-type-adt/14818946 (Accessed: 18 June 2023).

Athiramohandas *et al.* (2011) *Priority queue using linked list*, *athiraamazhichira*. Available at: https://athiraamazhichira.wordpress.com/2011/08/22/priority-queue-using-linked-list/ (Accessed: 18 June 2023).

baeldung, W. by: (2023) *When will the worst case of merge sort occur?*, *Baeldung on Computer Science*. Available at: https://www.baeldung.com/cs/merge-sort-time-complexity (Accessed: 18 June 2023).

Janssen, T. (2023) *OOP concept for beginners: What is abstraction?*, *Stackify*. Available at: https://stackify.com/oop-concept-abstraction/ (Accessed: 18 June 2023).

Martin, M. (2023) *Difference between abstraction and encapsulation*, *Guru99*. Available at: https://www.guru99.com/difference-between-abstraction-and-encapsulation.html (Accessed: 18 June 2023).

*Queues and priority queues* (no date) *Chapter 16: Queues and Priority Queues*. Available at: https://www.openbookproject.net/thinkcs/archive/java/english/chap16.htm (Accessed: 18 June 2023).

Rogers, Wm.P. (2001) *Encapsulation is not information hiding*, *InfoWorld*. Available at: https://www.infoworld.com/article/2075271/encapsulation-is-not-information-hiding.html (Accessed: 18 June 2023).

sevugarajansevugarajan                    9 *et al.* (1956) *What is an abstract data type in object oriented programming?*, *Stack Overflow*. Available at: https://stackoverflow.com/questions/1692933/what-is-an-abstract-data-type-in-object-oriented-programming (Accessed: 18 June 2023).

Singh, C. and says, J. (no date) *Home*, *BeginnersBook*. Available at: https://beginnersbook.com/2014/01/java-program-to-check-palindrome-string/ (Accessed: 18 June 2023).

*Stacks.* (no date) *Princeton University*. Available at: https://introcs.cs.princeton.edu/java/43stack/ (Accessed: 18 June 2023).