

**Data Structures & Algorithms**

**40201201**

**T/618/7430**

**Section (1)**

**Interview Questions**

**Submitted to**

Eng. Hana Al Rasheed

**Submitted on**

June 18th, 2023

**Submitted by**

Marwan Tareq Shafiq Al Farah

**Student ID**

21110011

**Spring 2022 - 2023**

**Table of Content**

[***Part 1*** 3](#_Toc137570620)

[**Q1.** **Design and Time Complexity Analysis of a Min-Priority Queue with Linked List** 3](#_Toc137570621)

[**Q2.** **Illustrating Min-Priority Queue Operations with a Linked List** 4](#_Toc137570622)

[**Q3.** **Implementing a Min-Priority Queue with Linked List** 6](#_Toc137570623)

[***Part 2*** 7](#_Toc137570624)

[**Q4.** **Time Complexity Analysis of Selection Sort and Merge Sort Algorithms** 7](#_Toc137570625)

[**Q5.** **Performance Comparison of Selection Sort and Merge Sort on Sorted and Reversely Sorted Data** 8](#_Toc137570626)

[**Q6.** **Correlation of Experimental Results with Theoretical Time Complexity** 9](#_Toc137570627)

[**Q7.** **Alternative Measures for Comparing Algorithm Efficiency** 11](#_Toc137570628)

[**Q8.** **Step-by-step Simulation of Dijkstra’s and Bellman-Ford Algorithms on a Sample Graph** 12](#_Toc137570629)

[**Q9.** **Implementation of Dijkstra Algorithm** 20](#_Toc137570630)

[**Q10.** **Error Handling in Dijkstra’s Algorithm Implementation** 20](#_Toc137570631)

[***Part 3*** 21](#_Toc137570632)

[**Q11.** **Designing a Pseudocode to Identify Palindromes** 21](#_Toc137570633)

[**Q12.** **Defining and Illustrating Operations of a Data Structure for Palindrome Detection** 22](#_Toc137570634)

[**Q13.** **Understanding Function Calls Through the Call Stack** 23](#_Toc137570635)

[***Part 4*** 28](#_Toc137570636)

[**Q14.** **Choosing the Right Data Structure for Specific Scenarios: Spell Checker and Priority Queue** 29](#_Toc137570637)

[**Q15.** **Analysis of Running Time for Given Code Snippet** 30](#_Toc137570638)

[**Q16.** **The Importance of Encapsulation and Information Hiding in ADT Implementation** 31](#_Toc137570639)

[**Q17.** **Imperative ADTs as the Foundation for Object Orientation: A Discussion** 32](#_Toc137570640)

[**Q18.** **The Benefits of Using Implementation Independent Data Structures: A Critical Evaluation** 33](#_Toc137570641)

[***References*** 36](#_Toc137570642)

# ***Part 1***

1. **Design and Time Complexity Analysis of a Min-Priority Queue with Linked List:**

A priority queue is an Abstract Data Type which is a more advanced type of data structure as it is intended to manage a group of elements, each of which is assigned a specific priority and a key feature of this data structure is that the priority of the elements defines their order. In a Min-Priority Queue the element with the lowest priority takes precedence, implying that it should be removed first.

**Formal Definition:** (GyanBlog, 2019; GeeksforGeeks, 2022; Simplilearn, 2023)

A Min-Priority Queue is a collection of elements, each of which is paired with a distinct priority. The distinct feature of this queue is the maintained order of the elements, which is based on increasing priority, therefore, this arrangement ensures that the element with the lowest priority can be accessed with the highest level of effectiveness. The queue is intended to support three main operations:

* **insert(priority, element):** This operation adds a specified element to the queue while assigning it a specified priority.
* **getMin():** This operation returns the element with the lowest priority from the queue, but it does not remove the element from the queue.
* **removeMin():** This operation removes the element with the lowest priority from the queue.

The Min-Priority Queue can be implemented using a variety of data structures, such as an array, a heap, or a linked list. The Min-Priority Queue will be implemented using a singly linked list in the context of this design specification.

**Application:**

Min-Priority Queues have a wide range of applications in many fields including computer science. They are useful in situations where it is necessary to keep a collection of elements linked to specific priorities. In the field of operating systems, for example, a Min-Priority Queue can be used to schedule tasks or jobs for execution. The tasks with the highest priority are completed first. In a network, a Min-Priority Queue can be used to prioritize packet transmission based on importance. A Min-Priority Queue can also be used effectively in simulations to handle events in the exact order in which they are scheduled to occur. (GeeksforGeeks, 2020; Javatpoint, 2021; KOCHAR, 2022)

**Operations and Time Complexity:**

* **insert(priority, element):** This operation is intended to add a new element to the queue, assigning it a priority. Insertion entails determining the appropriate position for the new element based on its priority and then appending it to the queue. The list will be iterated through in the implementation using a singly linked list until the first node with a priority higher than the new element is located. The new element will then be added before that node. If no nodes with a higher priority are detected, the new element is added to the list’s end. Because we may need to be traverse through the entire list to identify the right location for the new entry the time complexity of the insert operation is O(n), where n is the number of entries in the queue.
* **getMin():** The getMin() method is intended to return the element with the lowest priority from the queue without removing it. Because the queue elements are always kept sorted by priority, the first element in the queue, which also has the lowest priority, can be easily returned. The first node in the implementation using a singly linked list can be accessed in O(1) time, thus the getMin function has a time complexity of O(1).
* **removeMin():** This operation attempts to remove the element with the lowest priority from the queue. In the context of our implementation, which employs a singly linked list, the removal procedure entails finding the first node in the list, which can be removed in O(1) time by simply updating the list’s front pointer.

A Min-Priority Queue is an extremely useful ADT for managing elements with associated priorities which can be implemented using a variety of data structures, each with its own set of advantages and disadvantages. This design specification concentrated on the implementation of the Min-Priority Queue using a singly linked list, and it provided a detailed explanation of the three main operations, as well as their time complexities.

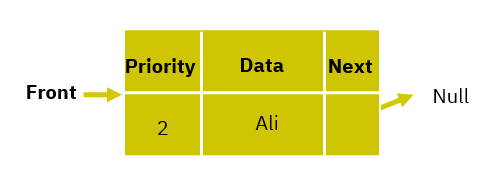
1. **Illustrating Min-Priority Queue Operations with a Linked List:**

The illustration can be found in the “DSA\_Q2\_FinalAssignment.mp4” file attached.

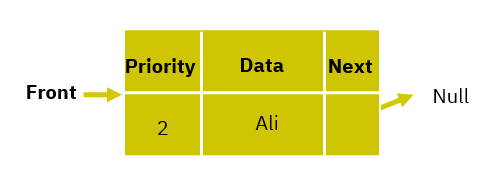
If the Queue is Empty:

**Insert:**

The queue is empty



The New Node is created



Front points to the New Node

**getMin:**

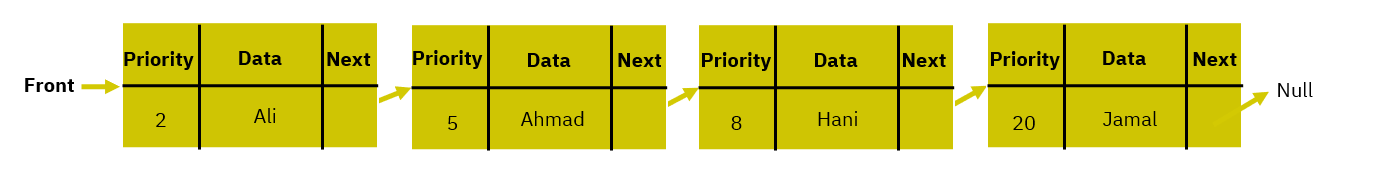
Throws an illegal state exception “The Queue is Empty”

**removeMin:**

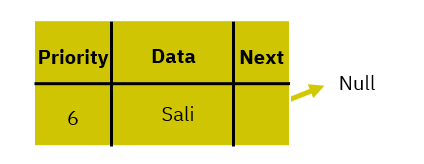
Throws an illegal state exception “The Queue is Empty”

However, if the queue wasn’t empty:

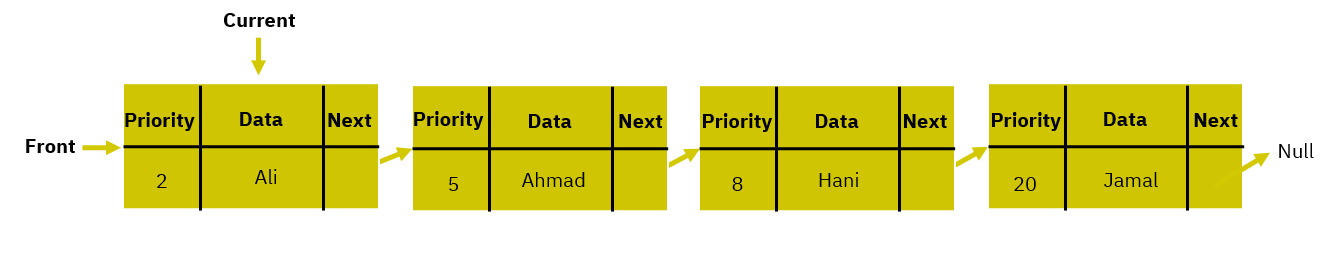
**Insert:**



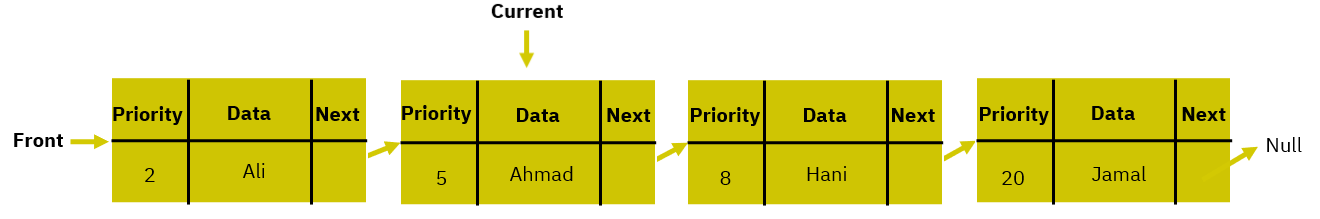
The Min-Priority Queue



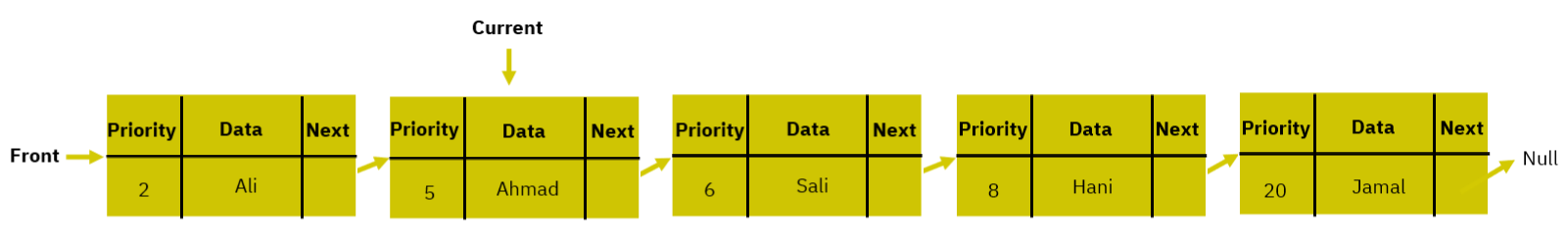
Creating a New Node (newNode) to add it to the Min-Priority Queue



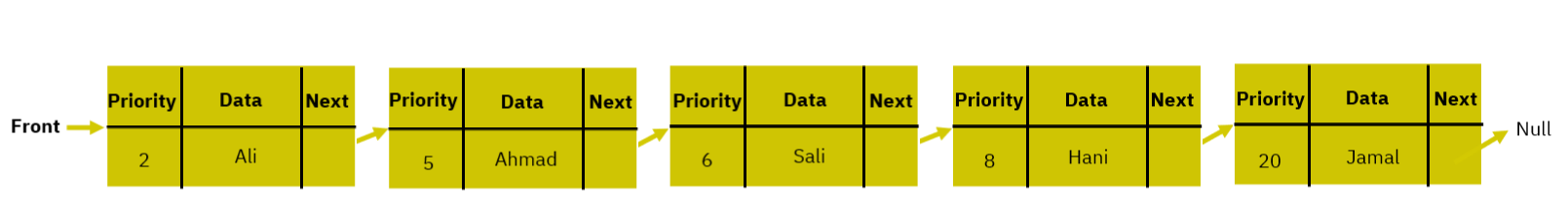
Create a New Node (current) that point to the (front) node to find the correct place to add the (newNode) to the Min-Priority Queue



The (current) node iterating through the Min-Priority Queue to find the optimal location

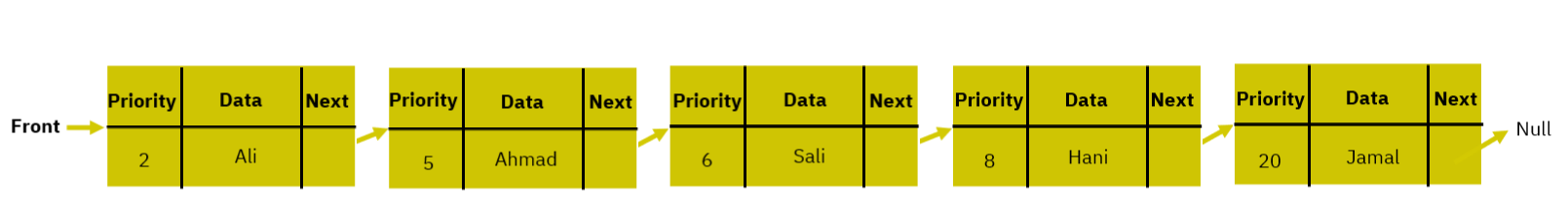


After finding that location the (newNode) gets added to the Min-Priority Queue at that location by making the (current.next) point to the (newNode) and the (newNode.next) point to (current.next)



After adding the (newNode) the code finishes its execution

**getMin():**



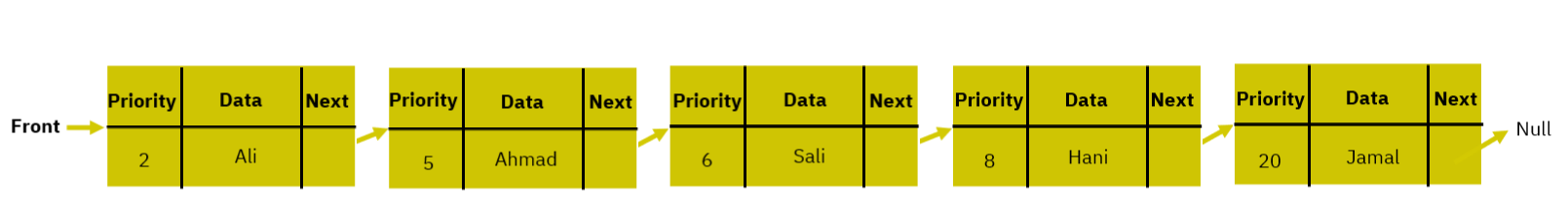
The Min-Priority Queue

A picture containing yellow, line, measuring stick

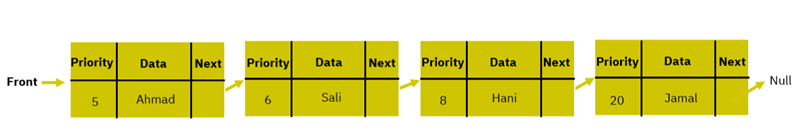
Description automatically generated

Returning (front.data)

**removeMin():**



The Min-Priority Queue



Making the (front) point to (front.next) thus making it point to the following node

1. **Implementing a Min-Priority Queue with Linked List:**

The implementation can be found in the “q3Final” package attached.

# ***Part 2***

**Merge Sort:** Merge sort is a divide-and-conquer algorithm that divides a problem into smaller, more manageable parts, solves each separately, and then combines the solutions to solve the original problem. It divides the input array in half repeatedly until it reaches single-element subarrays that are inherently sorted. It then merges these subarrays in a sorted order, resulting in a sorted version of the original array.

**Selection Sort:** Selection sort is a comparison-based in-place algorithm. It divides the input list into two sections, the sorted and the unsorted section. Initially, the sorted section is empty, and the unsorted section contains the entire list. The algorithm repeatedly selects the smallest (or largest, depending on sorting order) unsorted sublist element, swaps it with the leftmost unsorted element, and moves the sublist boundaries one element to the right.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | 5000 | 50000 | 500000 |
| Merge Sort | **Sorted** | 1.096 ms | 2.027 ms | 12.103 ms |
| **Reversely Sorted** | 0.602 ms | 2.436 ms | 16.812 ms |
| Selection Sort | **Sorted** | 11.126 ms | 2141.357 ms | 328710.631 ms |
| **Reversely Sorted** | 8.614 ms | 2119.969 ms | 256829.479 ms |

1. **Time Complexity Analysis of Selection Sort and Merge Sort Algorithms:**

|  |  |  |
| --- | --- | --- |
|  | Merge Sort | Selection Sort |
| Best Case | O(n log n) | O(n2) |
| Worst Case | O(n log n) | O(n2) |

Let’s examine the time complexity (Big-O) of the sorting algorithms based on the results and determine the best and worst case scenarios.

**Merge Sort:** In both the best and worst case scenarios, the time complexity of the Merge Sort algorithm is O(n log n). Because of this, Merge Sort outperforms Selection Sort for larger input sizes. Merge Sort’s running time increases linearithmically, or in proportion to the size of the input array multiplied by the input size’s logarithm.

**Selection Sort:** In both the best and worst case scenarios, the time complexity of the Selection Sort algorithm is O(n2). This indicates that the size of the input array has a quadratic effect on the running time of Selection Sort.

**Best Case:** The best case for both Selection Sort and Merge Sort occurs when the input array is already sorted. In this case, Selection Sort still necessitates O(n2) comparisons and swaps, whereas Merge Sort necessitates O(n log n) comparisons. As a result, the best-case time complexity of both algorithms remains the same.

**Worst Case:** The worst case for Selection Sort occurs when the input array is in reverse order. In this case, the maximum number of comparisons and swaps are required, resulting in a time complexity of O(n2). The worst case scenario for Merge Sort is when the input array is in reverse order. However, because Merge Sort’s divide-and-conquer strategy ensures that the merging step is always executed in O(n) time, the time complexity remains at O(n log n).

1. **Performance Comparison of Selection Sort and Merge Sort on Sorted and Reversely Sorted Data:**

The primary goal of the analysis is to comprehend the performance of these algorithms in terms of time complexity under various data conditions, specifically sorted and reversely sorted data.

**Merge Sort:**

* **Performance on Sorted Data:** When the input array is already sorted, the merge sort algorithm still performs the entire dividing and merging process in O(n log n) time. The time taken increases linearithmically with the size of the input, as shown in the data: 1.096 ms for 5000 elements, 2.027 ms for 50000 elements, and 12.103 ms for 500000 elements.
* **Performance on Reversely Sorted Data:** Even when the array is reversely sorted, the merge sort algorithm keeps its O(n log n) time complexity. This is because of merge sort’s divide-and-conquer strategy that ensures that the splitting and merging steps are always executed regardless of the input order in O(n log n) time. The data provided validates this: 0.602 ms for 5000 elements, 2.436 ms for 50000 elements, and 16.812 ms for 500000 elements.

**Selection Sort:**

* **Performance on Sorted Data:** Even when the input array is already sorted, the selection sort algorithm requires O(n2) comparisons and swaps because it lacks a mechanism for detecting a sorted array. The time taken quadratically increases with the size of the array, as evidenced by the data: 11.126 ms for 5000 elements, 2141.357 ms for 50000 elements, and 328710.631 ms for 500000 elements.
* **Performance on Reversely Sorted Data:** When the array is reversely sorted, the worst-case scenario occurs. The algorithm here performs the greatest number of comparisons and swaps, resulting in a time complexity of O(n2). The provided data confirms this: 8.614 ms for 5000 elements, 2119.969 ms for 50000 elements, and 256829.479 ms for 500000 elements.

For larger input sizes, regardless of the initial order of the array, Merge Sort outperforms Selection Sort. While the time complexity of Selection Sort remains O(n2) for both sorted and reversely sorted data, Merge Sort has a time complexity of O(n log n), demonstrating its efficiency.

It is important to note that Merge Sort requires additional space to perform the merge operation, whereas Selection Sort is an in-place sorting algorithm that does not. While Merge Sort outperforms Selection Sort in terms of time complexity, it does so at the expense of greater space complexity.

The choice between these two sorting algorithms should be made based on your task’s specific requirements and constraints. If space isn’t an issue and you’re dealing with large data sets, Merge Sort is a better option due to its superior time complexity of O(n log n) in all scenarios. Selection Sort, on the other hand, may be a viable option if memory is a concern and the data set is relatively small. Selection Sort has the advantage of not requiring extra space despite its poor time complexity of O(n2). It is important to note, however, that the performance of Selection Sort degrades significantly with larger data sets, therefore, even with small data sets, if the data is likely to grow in the future, Merge Sort or another more efficient algorithm should be used.

Finally, based on the given data and the time complexity analysis, Merge Sort is generally the superior algorithm due to its consistent performance on larger data sets, both sorted and reversely sorted data. While the implementation of Selection Sort is simpler and does not require extra space, it performs significantly worse on larger datasets, making it less suitable for big data tasks.

1. **Correlation of Experimental Results with Theoretical Time Complexity:**

The relationship between theoretical time complexity and experimental runtimes can provide profound insights into the efficiency and performance of various algorithms. We are evaluating and comparing the performance of two sorting algorithms in this case: Merge Sort and Selection Sort.

**Merge Sort:**

* Merge Sort is a divide-and-conquer algorithm with an O(n log n) time complexity in both the best and worst case scenarios. This linearithmic time complexity means that as the size of the input (n) grows, the algorithm’s runtime grows in proportion to n multiplied by the logarithm of n. This is due to the algorithm’s divide-and-conquer strategy, in which it divides the array until it reaches single-element arrays (which are inherently sorted) and then merges these arrays back together in sorted order.
* The experimental results support this theoretical time complexity. The time required by Merge Sort increased linearithmically with the size of the input array, whether it is already sorted or reversely sorted. For example, when the array size was increased tenfold from 5000 to 50000, the time taken increased within a logarithmic multiple rather than tenfold. Similarly, increasing the array size from 50000 to 500000 increased the time taken, but not in direct proportion to the increase in input size. This pattern was observed in arrays that were both sorted and reversely sorted.
* This experimental observation confirms that the time complexity of Merge Sort is O(n log n), as the increase in runtime is more than linear but less than quadratic, as predicted by the linearithmic time complexity. This consistency in Merge Sort performance, regardless of the initial order of the array, demonstrates the algorithm’s robustness and efficiency, making it a suitable choice for larger data sets.

**Selection Sort:**

* Selection Sort, on the other hand, is a straightforward comparison-based algorithm with an O(n2) time complexity in both the best and worst case scenarios. This quadratic time complexity implies that as the size of the input (n) grows, the algorithm’s runtime grows in proportion to the square of n. This is because Selection Sort scans the remaining unsorted elements for each element in the array to find the smallest and swaps it into the correct position.
* This theoretical time complexity is supported by the experimental results. When the size of the input array increases, Selection Sort takes much less time than Merge Sort. For example, when the array size increased tenfold from 5000 to 50000, Selection Sort took significantly longer than tenfold. This pattern continued when the array size was increased to 500000. This held true for both sorted and reversely sorted data. Even when the input was already sorted, Selection Sort had to traverse the entire array due to its lack of a mechanism to recognize an already sorted list, resulting in a time complexity of O(n2).
* Selection Sort’s theoretical time complexity of O(n2) is confirmed by this quadratic increase in runtime with increasing input size. The algorithm’s performance degrades rapidly as the input size increases, making it less suitable for larger data sets.

In conclusion, the experimental results are consistent with the theoretical time complexities of Merge Sort and Selection Sort. Merge Sort’s linearithmic time complexity make it a better choice for sorting large data sets, whereas Selection Sort’s quadratic time complexity and rapid degradation of performance with increasing input size make it better for sorting smaller data sets. Regardless of these differences, understanding theoretical time complexity and how it translates into actual runtime can aid in selecting the best algorithm for the job.

1. **Alternative Measures for Comparing Algorithm Efficiency:** (faceprep, 2020; GeeksforGeeks, 2021; Topics, 2022; Soni Upadhyay, 2023)

While time complexity is the most commonly used metric for comparing algorithm efficiency, it is not the only one as other critical factors, such as the specific constraints and requirements of the task at hand, can influence the choice of one algorithm over another. One such critical metric is space complexity.

The total amount of memory required to run an algorithm to completion is referred to as its space complexity. The need to consider space complexity arises because the memory of any computational device is limited. An efficient algorithm should have less time complexity as well as less space complexity. When working with large data sets, algorithms that require less memory are preferable, especially when memory resources are scarce or expensive.

Space complexity often falls into two components:

* **Fixed Part:** This is the amount of space that the algorithm and its variables require. This space does not change with the size of the problem and is frequently overlooked in space complexity analysis.
* **Variable Part:** This part is affected by the size of the problem. It is the extra space required by an algorithm to execute, including additional variables, arrays, stacks, heaps, and so on.

Space complexity, like time complexity, is expressed in Big-O notation. For example, an algorithm with space complexity O(n) necessitates linear space. In other words, the memory requirements scale linearly with the size of the input.

The space complexities of the two sorting algorithms we’ve previously discussed, Selection Sort and Merge Sort, differ significantly.

Selection Sort is an in-place sorting algorithm that requires no extra storage space beyond what is required to hold the original list. As a result, it has a space complexity of O(1), implying that it uses constant space. When memory space is limited, Selection Sort is a good option.

Merge Sort, on the other hand, is not an in-place sorting algorithm. It must create new arrays for the merging process, resulting in a space complexity of O(n), implying that it requires space proportional to the size of the input list. Merge Sort hence requires more memory but is more time efficient in terms of time complexity, which may be an issue when working with very large data sets or in systems with limited memory accessibility.

In conclusion, while evaluating the efficacy of an algorithm, it is critical to consider both space complexity and time complexity, especially when resources are limited or when working with enormous volumes of data. You can choose the best algorithm for a certain task by being more balanced and knowledgeable about both of these factors.

1. **Step-by-step Simulation of Dijkstra’s and Bellman-Ford Algorithms on a Sample Graph:**

In this section, we demonstrate how the Bellman-Ford and Dijkstra algorithms compare when simulated on the same directed graph. We begin at vertex zero and finish at vertex five. The graph is described by the following adjacency matrix, where each cell (i,j) denotes the weight of the edge from node i to node j. No direct connection between any of the nodes is indicated by the matrix’s zeros:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Vertex 0 | Vertex 1 | Vertex 2 | Vertex 3 | Vertex 4 | Vertex 5 |
| Vertex 0 | 0 | 24 | 5 | 10 | 0 | 0 |
| Vertex 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Vertex 2 | 0 | 0 | 0 | 1 | 30 | 10 |
| Vertex 3 | 0 | 0 | 0 | 0 | 0 | 8 |
| Vertex 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| Vertex 5 | 0 | 0 | 0 | 0 | 20 | 0 |

When simulating the Dijkstra algorithm, we begin at the source node (node 0) and proceed as follows:

* **Initialization:** We designate a tentative distance value of infinity for all nodes and zero for our initial node and marking all nodes as unvisited.
* **Iteration and Distance Update:** We go from node to node, starting from the source node, updating the distances of those who haven’t been visited yet. We replace it if a newly calculated distance is less than the currently assigned value.
* **Node Finalization:** After considering all unvisited neighbors of the current node, we mark the node as visited. A visited node will not be checked again.
* **Algorithm Termination:** The algorithm terminates when the destination node is marked as visited or when the smallest distance among the nodes in the unvisited set is infinity, or when all nodes have been visited.

**Step by Step Implementation:**

* **Initialization:**

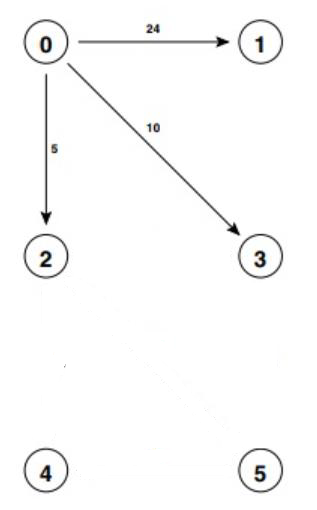
|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | INF | - | - |
| 2 | INF | - | - |
| 3 | INF | - | - |
| 4 | INF | - | - |
| 5 | INF | - | - |

* **Iteration, Distance Update, Node Finalization:**

**Iteration 1:**

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | 24 | 0 | 0 🡪 1 |
| 2 | 5 | 0 | 0 🡪 2 |
| 3 | 10 | 0 | 0 🡪 3 |
| 4 | INF | - | - |
| 5 | INF | - | - |

* Choose the unvisited vertex with the shortest distance: vertex 0.
* If a shorter path is found, update the distances of its neighboring vertices: in this case, the distances from vertex 0 to vertex 1, 2, and 3 are 24, 5, and 10 respectively, which are less than INF; thus, all of their distances are updated.
* Vertex 0 is marked as visited.

****

**Iteration 2:**

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | 24 | 0 | 0 🡪 1 |
| 2 | 5 | 0 | 0 🡪 2 |
| 3 | 6 | 2 | 0 🡪 2 🡪 3 |
| 4 | 35 | 2 | 0 🡪 2 🡪 4 |
| 5 | 15 | 2 | 0 🡪 2 🡪 5 |

* Choose the unvisited vertex with the shortest distance: vertex 2.
* If a shorter path is found, update the distances of its neighboring vertices: in this case, the total distances from vertex 0 to vertex 3, 4, and 5 through vertex 2 are 6, 35, and 15 respectively. For vertex 3 this applies as 6 is less than 10, thus the distance is updated. This also applies for vertex 4 as 35 is less that INF thus the distance is updated. Finally, for vertex 5 this also applies as 15 is less than INF, thus the distance is updated as well.
* Vertex 2 is marked as visited.

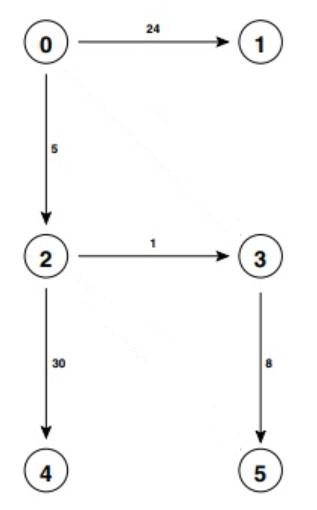
**A picture containing line, circle, diagram

Description automatically generated**

**Iteration 3:**

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | 24 | 0 | 0 🡪 1 |
| 2 | 5 | 0 | 0 🡪 2 |
| 3 | 6 | 2 | 0 🡪 2 🡪 3 |
| 4 | 35 | 2 | 0 🡪 2 🡪 4 |
| 5 | 14 | 3 | 0 🡪 2 🡪 3 🡪 5 |

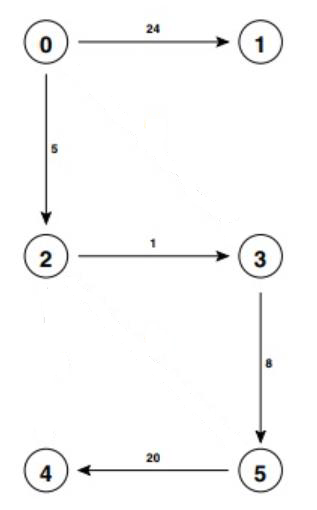
* Choose the unvisited vertex with the shortest distance: vertex 3.
* If a shorter path is found, update the distances of its neighboring vertices: in this case, the total distances from vertex 0 to vertex 5 through vertex 3 is 14 which is less than the distance 15, thus, the distance is updated.
* Vertex 3 is marked as visited.



**Iteration 4:**

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | 24 | 0 | 0 🡪 1 |
| 2 | 5 | 0 | 0 🡪 2 |
| 3 | 6 | 2 | 0 🡪 2 🡪 3 |
| 4 | 34 | 5 | 0 🡪 2 🡪 3 🡪 5 🡪 4 |
| 5 | 14 | 3 | 0 🡪 2 🡪 3 🡪 5 |

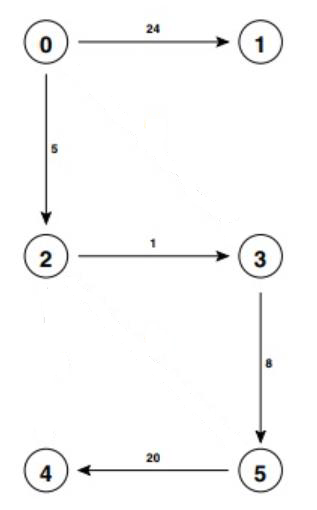
* Choose the unvisited vertex with the shortest distance: vertex 5.
* If a shorter path is found, update the distances of its neighboring vertices: in this case, the total distances from vertex 0 to vertex 4 through vertex 5 is 34 which is less than the distance 35, thus, the distance is updated.
* Vertex 5 is marked as visited.



**Iteration 5:**

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | 24 | 0 | 0 🡪 1 |
| 2 | 5 | 0 | 0 🡪 2 |
| 3 | 6 | 2 | 0 🡪 2 🡪 3 |
| 4 | 34 | 5 | 0 🡪 2 🡪 3 🡪 5 🡪 4 |
| 5 | 14 | 3 | 0 🡪 2 🡪 3 🡪 5 |

* Choose the unvisited vertex with the shortest distance: vertex 1.
* If a shorter path is found, update the distances of its neighboring vertices: in this case, vertex 1 doesn’t have any out-degree vertices thus no changes are made.
* Vertex 1 is marked as visited.



**Iteration 6:**

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | 24 | 0 | 0 🡪 1 |
| 2 | 5 | 0 | 0 🡪 2 |
| 3 | 6 | 2 | 0 🡪 2 🡪 3 |
| 4 | 34 | 5 | 0 🡪 2 🡪 3 🡪 5 🡪 4 |
| 5 | 14 | 3 | 0 🡪 2 🡪 3 🡪 5 |

* Choose the unvisited vertex with the shortest distance: vertex 4.
* If a shorter path is found, update the distances of its neighboring vertices: in this case, vertex 4 doesn’t have any out-degree vertices thus no changes are made.
* Vertex 4 is marked as visited.

A picture containing circle, line

Description automatically generated

* **Algorithm Termination:** after iterating through all vertices the algorithm is terminated and we can obtain the shortest distances between a source vertex and any other vertex.

Following these steps, we discover that the shortest path from vertex 0 to vertex 5 is 0 🡪 2 🡪 3 🡪 5 resulting in a total distance of 14.

Let us now proceed to the simulation of the Bellman-Ford algorithm. This algorithm is more general than Dijkstra’s in that it can handle graphs with negative edge weights. Bellman-Ford employs dynamic programming by repeatedly relaxing the edges, resulting in successive approximations of the shortest-path weights.

The Bellman-Ford algorithm, like Dijkstra’s, is initialized by setting the distance from the source to itself to zero and the distance to all other vertices to infinity. However, unlike Dijkstra, Bellman-Ford does not use a priority queue to select the next vertex. Instead, for V-1 iterations, it relaxes all edges simultaneously, where V is the number of vertices in the graph.

**Step by Step Implementation:**

* **Initialization:**

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | INF | - | - |
| 2 | INF | - | - |
| 3 | INF | - | - |
| 4 | INF | - | - |
| 5 | INF | - | - |

* **Iterations:**

Going through each iteration in this order of edges (0,1), (0,2), (0,3), (2,3), (2,4), (2,5), (3,5), (5,4).

**Iteration 1:**

* **Edge (0,1):** INF is the current shortest known distance to Vertex 1. If we go from Vertex 0 to Vertex 1, the distance is 24, which is less than INF, so we update the distance to Vertex 1 to 24.
* **Edge (0,2):** As with the previous example, the current shortest known distance to Vertex 2 is INF. The distance between Vertex 0 and Vertex 2 is 5, which is less than INF, so we update the distance to Vertex 2 to be 5.
* **Edge (0,3):** The current shortest known distance to Vertex 3 is INF. If we go from Vertex 0 to Vertex 3, the distance is 10, which is less than INF, so we update the distance to Vertex 3 to be 10.
* **Edges (2,3):** The current shortest known distance to Vertex 3 is 10. If we go from Vertex 2 (current distance 5) to Vertex 3, the distance is 5 + 1 = 6, which is less than 10, so we update the distance to Vertex 3 to be 6.
* **Edge (2,4):** The current shortest known distance to Vertex 4 is INF. If we go from Vertex 2 to Vertex 4, the distance is 5 + 30 = 35, which is less than INF, so we update the distance to Vertex 4 to 35.
* **Edges (2,5):** The current shortest known distance to Vertex 5 is INF. If we go from Vertex 2 to Vertex 5, the distance is 5 + 10 = 15, which is less than INF, so we update the distance to Vertex 5 to 15.
* **Edge (3,5):** The current shortest known distance to Vertex 5 is 15. If we go from Vertex 3 (current distance 6) to Vertex 5, the distance is 6 + 8 = 14, which is less than 15, so we update the distance to Vertex 5 to be 14.
* **Edges (5,4):** The current shortest known distance to Vertex 4 is 35. If we go from Vertex 5 (current distance 14) to Vertex 4, the distance is 14 + 20 = 34, which is less than 35, so we update the distance to Vertex 4 to 34.

A picture containing circle, line

Description automatically generated

The final distance table after processing all edges in the given order in the first iteration is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | 24 | 0 | 0 🡪 1 |
| 2 | 5 | 0 | 0 🡪 2 |
| 3 | 6 | 2 | 0 🡪 2 🡪 3 |
| 4 | 34 | 5 | 0 🡪 2 🡪 3 🡪 5 🡪 4 |
| 5 | 14 | 3 | 0 🡪 2 🡪 3 🡪 5 |

**Iteration 2:**

Let’s move on to the second iteration. We go over all the edges again in the order you specified:

* **Edge (0,1):** The current shortest known distance to Vertex 1 is 24. Going from Vertex 0 to Vertex 1 does not result in a shorter path, so no update is performed.
* **Edge (0,2):** The current shortest known distance to Vertex 2 is 5. Going from Vertex 0 to Vertex 2 does not result in a shorter path, so no update is performed.
* **Edge (0,3):** The current shortest known distance to Vertex 3 is 6. Going from Vertex 0 to Vertex 3 does not result in a shorter path, so no update is performed.
* **Edges (2,3):** The current shortest known distance to Vertex 3 is 6. Going from Vertex 2 to Vertex 3 does not result in a shorter path, so no update is performed.
* **Edges (2,4):** The current shortest known distance to Vertex 4 is 34. Going from Vertex 2 to Vertex 4 does not result in a shorter path, so no update is performed.
* **Edge (2,5):** The current shortest known distance to Vertex 5 is 14. Going from Vertex 2 to Vertex 5 does not result in a shorter path, so no update is performed.
* **Edge (3,5):** The current shortest known distance to Vertex 5 is 14. Going from Vertex 3 to Vertex 5 does not result in a shorter path, so no update is performed.
* **Edge (5,4):** The current shortest known distance to Vertex 4 is 34. Going from Vertex 5 to Vertex 4 does not result in a shorter path, so no update is performed.

A picture containing circle, line

Description automatically generated

The distance table remains unchanged after the second iteration:

|  |  |  |  |
| --- | --- | --- | --- |
| Vertex | Distance | Previous Vertex | Path |
| 0 | 0 | 0 | 0 |
| 1 | 24 | 0 | 0 🡪 1 |
| 2 | 5 | 0 | 0 🡪 2 |
| 3 | 6 | 2 | 0 🡪 2 🡪 3 |
| 4 | 34 | 5 | 0 🡪 2 🡪 3 🡪 5 🡪 4 |
| 5 | 14 | 3 | 0 🡪 2 🡪 3 🡪 5 |

We can stop the algorithm here because there were no updates in the second iteration. Because the graph contains no negative cycles, no further iterations can improve the shortest paths. As a result, the shortest distances between Vertex 0 and the other vertices are shown in the table above.

1. **Implementation of Dijkstra Algorithm:**

The implementation can be found in the “part2Dijkstra” package attached.

1. **Error Handling in Dijkstra’s Algorithm Implementation:**

The implementation can be found in the “part2Dijkstra” package attached.

# ***Part 3***

1. **Designing a Pseudocode to Identify Palindromes:**

In order to determine if a string is a palindrome we use a Stack data structure. The process involves creating a reverse string using a Stack and comparing it to the original string to determine whether it is a palindrome or not.

Our isPalindrome(str) function takes a string as an input and returns a boolean result indicating whether or not the text is a palindrome. The comparable pseudocode for our function is as follows:

**Pseudocode:**

Function isPalindrome(String str):

    Define Stack stack

    Define String reverseStr = ““

    Convert str to lower case

    For each character in str:

        Push character to stack

    While stack is not empty:

        Pop character from stack and append it to reverseStr

    If str is equal to reverseStr:

        Return TRUE

    Else:

        Return FALSE

To account for case sensitivity, we first convert the input string, str, to lowercase. Then we loop through the string, pushing each character onto the stack.

We form a reverse string once the stack is full of characters from the string. This is accomplished by sequentially popping each character from the stack and appending it to our previously defined reverseStr.

Finally, we compare the original string to the reverseStr. If they match, we know the string is a palindrome and return TRUE; otherwise, we return FALSE.

Thus, the provided pseudocode demonstrates a simple and robust method for identifying palindromes by utilizing a unique data structure - the Stack.

1. **Defining and Illustrating Operations of a Data Structure for Palindrome Detection:**

We selected the Stack Abstract Data Type (ADT) as our data structure in the previous question. A popular data structure in computer science called a stack adheres to the Last-In-First-Out (LIFO) principle, which makes it helpful in a variety of computing methods.

A linear data structure called a “Stack” performs equivalent operations to a real stack, such as a collection of plates. Pieces can be pushed or popped from one end of a stack, which is commonly referred to as the top. The last dish added to the pile is the first to be removed in a similar fashion. The core of Stack’s functionality is this particular LIFO (Last-in-First-out) order of operations.

It’s necessary to delve into Stack ADT’s basic operations in order to comprehend its usage and operations in a more thorough manner:

* **Push:** By performing this action, an element is added to the top of the stack. It is like adding a fresh plate on top of the stack. The newly added element then takes over as the stack’s current top element.
* **Pop:** Using the pop operation, an element is essentially taken off the top of the stack. This operation retrieves and removes the top element, making the next element (if it exists) the new top, analogous to lifting the topmost plate from our pile.
* **Top:** Without actually removing the top element from the stack, this operation returns its value. It gives us the chance to see what is at the top of our stack (or pile), much like how we can see what is on the top plate without moving the stack.
* **isEmpty:** The isEmpty operation simply determines whether the Stack is empty. If there are no plates in the pile, the Stack is empty, and otherwise, it returns the boolean value false.

After going over the operations, let’s use an illustration to show how they might be used. Let’s say our goal is to reverse the string “hello” using a stack:

* **Push:** Because the last character, a “o,” is the last to be pushed after the process is completed, it becomes the top element of the stack. After this procedure, the Stack resembles a stack of letters, with the letter “o” at the top and the letters “l,” “l,” “e,” and “h” at the bottom.
* **Top:** If the Top operation were to be performed at this point, the result would be the top element of the stack, which is ‘o’.
* **Pop:** When performing the Pop operation, we begin by removing elements from the top of the stack and adding them to a new string. The Pop operation resembles taking the top letter out of our stack. The first element to be popped and added to the new string is ‘o’ because it was the top element. Then comes ‘l’, ‘l’, ‘e’, and finally ‘h’. We get a reversed string because we are removing elements in the opposite direction from how they were added.
* **IsEmpty:** After popping every character and adding them to the end of our new string, the IsEmpty operation would return true, indicating that the Stack is now empty of all elements, much like having no more letters in our hypothetical stack.

**Logical Representation:**

* A Stack is logically expressed as a vertical stack of things. In our string reversal example, the string “hello” would be shown in the Stack as [‘o’, ‘l’, ‘l’, ‘e’, ‘h’]. The arrangement of the letters represents the order of the elements that will be popped ‘o’ above ‘l’, ‘l’ above ‘l’, ‘l’ above ‘e’, ‘e’ over ‘h’.
* Pushing elements onto the stack causes them to be added to the top, changing the top element every time. On the other hand, popping elements from the stack entails removing elements from the top and changing the top element as a result.

These core characteristics and functions of the Stack ADT are effectively used in the context of our palindrome detection algorithm. By pushing characters from a string onto a stack and then popping those characters to construct a new string, we successfully created a reversed string. Our capacity to compare this reversed string to the original string determines if the original string is a palindrome.

Finally, the Stack ADT is an important component of a wide range of algorithms and data manipulations, and it is complemented by its core set of operations. The practical application of Stack ADT’s properties and operations in resolving complex computational issues is illustrated by our palindrome checking algorithm. It demonstrates how knowledge of basic operations, properties, and simple data structures can help us create effective algorithms and manipulate data in a more orderly and efficient way.

1. **Understanding Function Calls Through the Call Stack:**

The call stack is a crucial data structure in computer execution. It maintains track of a program’s active functions or methods, as well as the variables and parameters associated with those methods.

public class Main {

  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 (String[] args) {

    int arr[] = {1, 2, 3, 4, 5};

    function1 (arr, 4);

    function2 ();

  }

}

**Illustration of the Call Stack of the Above Code:**

1. The first function to be called is main(). This will be the first function added to the empty call stack.

**Call Stack:**

main()

1. The first function called within main() is function1(). This is appended to the call stack.

**Call Stack:**

function1(arr, 4)  
main()

1. function1() is called again (recursion) within function1(), with i decremented by one.

**Call Stack:**

function1(arr, 3)  
function1(arr, 4)  
main()

1. This recursive call will continue until the value of i is less than zero.

**Call Stack:**

function1(arr, -1)  
function1(arr, 0)  
function1(arr, 1)  
function1(arr, 2)  
function1(arr, 3)  
function1(arr, 4)  
main()

1. When i is less than zero, function1() calls function2().

**Call Stack:**

function2()  
function1(arr, -1)  
function1(arr, 0)  
function1(arr, 1)  
function1(arr, 2)  
function1(arr, 3)  
function1(arr, 4)  
main()

1. After function2() completes, it is removed from the stack.

**Call Stack:**

function1(arr, -1)  
function1(arr, 0)  
function1(arr, 1)  
function1(arr, 2)  
function1(arr, 3)  
function1(arr, 4)  
main()

**Output:**

>>> let’s start ^\_^

1. Then function1() with i = -1 completes and is removed from the stack.

**Call Stack:**

function1(arr, 0)  
function1(arr, 1)  
function1(arr, 2)  
function1(arr, 3)  
function1(arr, 4)  
main()

**Output:**

>>> let’s start ^\_^

1. Following that, the next line of the last function1() call (with i = 0) is executed, printing arr[0]. This process of finishing function1() calls continues, printing the corresponding arr[i] each time.

**Call Stack:**

function1(arr, 4)  
main()

**Output:**

>>> let’s start ^\_^  
>>> 1  
>>> 2  
>>> 3  
>>> 4

1. function1(arr, 4) is now completed and removed from the stack.

**Call Stack:**

main()

**Output:**

>>> let’s start ^\_^  
>>> 1  
>>> 2  
>>> 3  
>>> 4  
>>> 5

1. Then, from main(), function2() is called.

**Call Stack:**

function2()  
main()

**Output:**

>>> let’s start ^\_^  
>>> 1  
>>> 2  
>>> 3  
>>> 4  
>>> 5

1. When function2() is finished, it is removed from the stack. Finally, the only function left on the stack is main().

**Call Stack:**

main()

**Output:**

>>> let’s start ^\_^  
>>> 1  
>>> 2  
>>> 3  
>>> 4  
>>> 5  
>>> let’s start ^\_^

1. When main() finishes, it is removed from the stack, leaving the call stack empty.

**Call Stack:**

(empty)

**Output:**

>>> let’s start ^\_^  
>>> 1  
>>> 2  
>>> 3  
>>> 4  
>>> 5  
>>> let’s start ^\_^

Therefore, this is the call stack’s state following each function call in the code. The call stack is helpful because it tracks the execution of function calls, which makes it simpler to comprehend the flow of code execution, particularly in situations where recursion and nested function calls are present. In the provided code, the call stack assists us in: (IBM, 2019; MDN, 2020; Cake, 2021)

* **Function Call Tracking:** The call stack keeps track of the functions that are currently active in your program. When a function or method is called, it is added to the call stack. It is popped off the stack when it returns. This enables the program to determine which function is currently being executed and where to return once the function has completed execution.
* **Recursion Control:** The code contains a recursive function (function1). Recursive functions call themselves, and it would be impossible to keep track of how many recursive calls have been made and where to return after each recursive call completes without a call stack. Each recursive call to function1 adds a new call stack entry. Once the call is completed, the entry is removed, returning to the previous recursive call.
* **Debugging and error handling:** If one of the functions fails, the call stack can assist you in determining the sequence of function calls that resulted in the failure. By looking at the call stack’s status at the moment of the error, you may determine the “path” of execution that led to the failure. When working with sophisticated programs that have several nested function calls.

# ***Part 4***

|  |  |  |  |
| --- | --- | --- | --- |
|  | Linked List | Array Unsorted | Array Sorted |
| Search | O(n) | O(n) | O(n)/O(log n) |
| Insert | O(n) 🡪 Searching for the element or position O(1) 🡪 Insertion | O(n) 🡪 Shifting the elements in the array to open a place for the added element | O(n) 🡪 Shifting the elements in the array to open a place for the added element  Or  (for keeping the array sorted)  O(n)/O(log n) 🡪 Searching for the position  O(n) 🡪 Shifting the elements in the array to open a place for the added element |
| Remove | O(n) 🡪 Searching for the element or position O(1) 🡪 Removal | O(n) 🡪 Shifting the elements in the array in place of the removed element | O(n) 🡪 Shifting the elements in the array in place of the removed element |

**Linked List:**

* **Searching:** To find a specific element in a linked list, you must traverse the list, which in the worst case takes O(n) time. ‘n’ denotes the number of items in the list.
* **Insertion:** In the worst-case scenario, finding the position to insert an element takes up O(n) time, whereas inserting at the first position only takes constant time, or O(1). If you need to insert an element at a specific position other than the first, the overall time complexity will end up being O(n).
* **Removal:** Time complexity for removing a specific element or position from a linked list can be O(n) in the worst-case scenario. However, removal from the first element maintains an O(1) operation similar to insertion. The overall complexity can be determined to be O(n).

**Unsorted Array:**

* **Searching:** To find a specific element in an unsorted array, you must traverse the array, using Linear Search which in the worst case takes O(n) time. The number ‘n’ denotes the number of elements in the array.
* **Insertion:** If you’re seeking to add a new element to a specific position within an array, you’ll have to shift around items to accommodate for the new addition. This can take up to O(n) time in the worst-case scenario, whereas simply appending to the end of the array is a constant time operation, clocking in at O(1). Considering the former approach’s complexity, the overall resulting time complexity is O(n).
* **Removal:** In the case of removing an element from the end of an array, the time needed is O(1) just like insertion. However, if removal is required from a particular position, it results in having to move the following elements to cover the vacant spot. This motion can take O(n) time in the worst-case scenario. As a result, the time complexity would be O(n).

**Sorted Array:**

* **Searching:** You can use binary search to find a specific element in a sorted array, which takes O(log n) time in the worst case. A linear search, on the other hand, would take O(n) time. The number ‘n’ denotes the number of elements in the array.
* **Insertion:** To add an element to the end of an array, it only takes O(1) time. But if the element needs to be placed at a specific spot, it requires shifting the subsequent elements, making the time complexity O(n). Despite this, if you want to keep the array sorted, you will have to pinpoint the appropriate position for the new element. One option could be a binary search that takes approximately O(log n) time. Once you’ve decided where to put the new element, the following components must be shifted to accommodate it. This action could take up to O(n) time in the worst-case scenario. Therefore, the insertion’s overall time complexity is O(n).
* **Removal:** Just like insertion, O(1) is the time to remove at the end of the array. But if you have a specific element’s position, removing it needs to shift the array, which takes O(n) time. Hence, overall the time complexity is O(n). Keeping the array sorted requires finding the element’s spot, which implies O(log n) time for binary search. Once found, the gap created by removing the element needs to fill in through shifted elements. Removing takes O(n) time, in the event of the worst-case scenario. Consequently, the O(n) total time complexity is a result of this operation’s shift.

1. **Choosing the Right Data Structure for Specific Scenarios: Spell Checker and Priority Queue:**

**Spell Checker Application:**

A sorted array would be an excellent choice in this case. The reason for this is that spell checkers require efficient search operations, and we can perform a binary search in a sorted array, which has a time complexity of O(log n), making it faster than a linear search in an unsorted array or a linked list.

Although insertion and deletion operations in a sorted array are inefficient (O(n) in the worst-case scenario), they are rarely used in a spell checker scenario. Because the primary operation is word searching and potential misspellings are not inserted into the dictionary, the cost of insertion is insignificant in this case.

Furthermore, sorted arrays allow us to quickly find words that are lexically close to the misspelled word, which can help us suggest corrections. If a word is missing from the array, the position where it should have been, can be used to suggest alternatives. The words immediately preceding and following this location will be lexically close to the misspelled word and can be used to suggest possible corrections.

While Hash Tables, Tries, and Bloom Filters are most commonly used for this scenario, the assignment brief specifically requested that we choose the data structure based on our analysis of Linked List, Sorted Array, and Unsorted Array above, with the sorted array being the best choice among these three. (Baeldung, 2023)

**Priority Queues:**

A linked list would be an appropriate choice among the given options for a priority queue. Priority queues must support efficient insertion and deletion operations, which a linked list can do well.

Elements in a priority queue are frequently removed from the front (highest priority), which is an O(1) operation in a linked list. The insertion operation in a linked list is O(n), as this requires keeping the list sorted by priority. Searching for the right position to insert in a sorted linked list is O(n), but given that the search operation isn’t as frequent as insertions and deletions in a typical priority queue, this trade-off may be acceptable.

An unsorted array would have inefficient deletion (because it involves shifting elements), and while a sorted array could also work, insertion operations (which are common in a priority queue) in a sorted array are also inefficient due to the need to shift elements to keep the array sorted.

While Binary Heaps are most commonly used for this scenario, the assignment brief specifically requested that we choose the data structure based on our analysis of Linked List, Sorted Array, and Unsorted Array above, with the linked list being the best choice among these three. (Programiz, 2021)

1. **Analysis of Running Time for Given Code Snippet:**

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 = fun1(arr, index - 4);

    if (m1 > m2)

        return m1;

    else if (m2 > m3)

        return m3;

    else

        return m1;

}

The fun1 function is a compelling example of a recursive function that provides an intriguing case for time complexity analysis. To accurately assess its time complexity, it is necessary to investigate the function’s inner workings, comprehend its recursive nature, and comprehend the implications of the varying decrements in the recursive calls.

The function takes two parameters: an array, arr, and an integer, index. The method’s functionality is dependent on recursive calls to itself with three distinct arguments: index - 1, index - 2, and index - 4. This branching recursion is at the heart of our investigation.

To unravel the time complexity, we establish a recursive relation, a mathematical expression that denotes the running time of the function T(n) in terms of its smaller versions. The recursive expression is based on the fact that the function calls itself three times with different arguments. As a result, it can be described as:

In this equation, O(1) represents the time spent performing comparison operations and returning the result. These operations are thought to execute in constant time, which means that their execution time remains constant regardless of the size of the input.

Due to the three recursive calls with varying decrements, the equation does not translate directly into standard Big O notation form. However, a rough approximation can be obtained by considering the T(n-1) branch with the highest frequency of recursive calls. This branch has a significant impact on time complexity because the number of recursive calls made by T(n-1) far outnumbers the other branches when n is large.

=

Each call to fun1 has the potential to result in three additional calls to fun1. In the worst-case scenario, the number of calls roughly triples with each level of recursion, culminating in 3n calls to fun1 for an input size n.

As a result, we are able to estimate an approximation for this function’s time complexity as O(3n). This approximation represents the time complexity’s worst-case scenario. Due to the possibility of fewer calls being made when the index becomes small, the actual time complexity may be less than this approximation.

1. **The Importance of Encapsulation and Information Hiding in ADT Implementation:** (cs.csub.edu, 2017; University of Wisconsin, 2020; Oracle, 2022)

Containing both data and the operations that can be carried out on that data, an ADT provides a top-level framework for data manipulation. While a Stack is one such ADT that allows for push, pop, top, and isEmpty operations, its specific implementation details - such as usage of an array or linked list - are kept hidden away from the user.

Let’s now examine the benefits of information hiding and encapsulation in this context:

* **Modularity:** Encapsulation helps to separate concerns and promotes modularity. The codebase is easier to manage and is more logically organized because each module or class has its own responsibilities. This division makes it simpler to debug and maintain a stack or any other ADT because each module can be tested and changed separately.
* **Reusability and Maintainability:** By isolating the code into various classes, we can reuse these classes whenever necessary without making any changes. Once we have implemented the push, pop, top, and isEmpty operations in the context of a stack, we can use them anywhere in our program. Additionally, if we need to change the implementation specifics, we only need to do so once, lowering the risk of errors and improving the code’s maintainability.
* **Information hiding:** Information hiding aids in data protection. Only a defined set of functions (like push and pop for a stack) can alter the internal state of a stack (or any other ADT). This stops the user from accidentally accessing data in a way that wasn’t intended or from putting the data structure in an invalid state.
* **Usefulness and Security:** We improve the data structure’s usability and safety by encapsulating the implementation details and offering a clear interface. The likelihood of errors decreases when a user employs the stack, which requires only awareness of available operations rather than full comprehension of the implementation process. Consequently, safety and ease of use are enhanced.
* **Flexibility and Efficiency:** Encapsulation enables a data structure’s implementation to be changed without having an impact on the code that utilizes the data structure. For better performance, we might switch a stack’s underlying data structure from an array to a linked list. As long as the interface, the available methods to interact with the stack, remains unchanged, none of the code using the stack needs to be altered.

In conclusion, encapsulation and information hiding are two fundamental concepts in object-oriented programming that offer a number of advantages when data structures are implemented as ADTs, as they increase safety and flexibility while also enhancing modularity, maintainability, and reusability.

1. **Imperative ADTs as the Foundation for Object Orientation: A Discussion:**

I strongly agree that imperative Abstract Data Types (ADTs) serve as the foundation for object-oriented programming (OOP). This argument is based on the inherent features and purposes of both imperative ADTs and OOP.

To begin with, imperative ADTs are an important component of the imperative programming paradigm. They define a data type through its operations it shields the user from the actual implementation. Data abstraction is a critical concept in OOP, where an ADT user can use the operation functionalities without any knowledge of the internal workings. The focus lies solely on the operator function and not on its inner workings.

Instances of classes in OOP are akin to objects, while classes themselves function as ADTs. These classes also determine the various functions or methods available to manipulate and modify data, similar to how an ADT establishes both operations and data. However, OOP expands on this concept by allowing for polymorphism and inheritance, which let one function work on objects belonging to various classes. Intuitive programming with ADTs generally does not support these concepts.

However, the fundamental idea of OOP is the combination of data and operations into a single unit, which is provided by ADTs. Using classes and objects, OOP implements this encapsulation. The development of sophisticated software systems is made possible when every object, viewed as a distinct entity, contains both data and methods. By doing this, each object is equipped with its own state and behavior, making it a unique instance of a class.

First steps toward OOP involve recognizing the significance of ADTs. Thus, understanding the importance of ADTs is easy. They introduce the ideas of data encapsulation and abstraction, which are later developed and improved upon in OOP. Additionally, a lot of object-oriented languages permit the use of ADTs because they support imperative programming. This emphasizes the connection between the two even more.

It’s crucial to keep in mind, though, that ADTs and OOP differ despite having concepts in common. OOP adds extra features like inheritance and polymorphism to the ADT principles. ADTs can therefore be a starting point for understanding OOP, but they do not cover all of its features and capabilities. I firmly believe that imperative ADTs lay a strong foundation for OOP, but they are only a small component of the overall picture.

1. **The Benefits of Using Implementation Independent Data Structures: A Critical Evaluation:** (Linode Docs, 2020; MIT OpenCourseWare, 2022)

Data structures are essential elements in computer science and software development because they provide effective methods for managing and organizing data. Implementation dependent and implementation independent are the two broad categories into which these structures can be divided. Implementation-dependent data structures, like arrays and linked lists, are reliant on a particular method of data organization and storage. In contrast to linked lists, which use nodes connected by pointers, arrays use contiguous memory locations to store data. Their particular traits and behaviors are closely related to how they were implemented differently.

Implementation independent data structures such as stacks and queues, are characterized more by their behavior and operations than by their actual implementation. They may be implemented in a variety of ways depending on the demands of the work at hand since they can be formed using a variety of underlying structures such as arrays or linked lists. Being “implementation independent” emphasizes how adaptable and versatile these data structures are, enabling their use in a variety of contexts and programming environments.

Let’s examine the advantages of using implementation independent data structures in more detail now:

* **Adaptability and Versatility:**

Stacks and queues are two implementation-independent data structures that offer a great level of flexibility and adaptability. Each underlying structure that may be used to construct these structures, such as arrays or linked lists, has its own set of benefits and drawbacks.

Take a stack as an example. Implementing it as an array might be a wise decision if you are aware of the stack’s maximum size beforehand. If the size is known in advance, memory allocation can be done only once, minimizing memory usage and preventing reallocation. In addition to that, arrays provide constant-time access to any element.

However, if the stack’s maximum size is unknown and there could be significant size fluctuations, a linked list might be a better option for the stack’s implementation. Dynamic size adjustment is possible with linked lists without the need to reallocate memory or move elements as with arrays.

Due to their implementation independence, these data structures offer a high degree of adaptability by enabling you to select the underlying structure that best suits your problem’s requirements and constraints.

* **Easy to Use and Abstraction:**

The fact that implementation independent data structures abstract away the difficulties of the underlying implementation is one of their main benefits. As a result, developers may focus on finding a solution to the present problem rather than becoming bogged down in the details of how the data structure is implemented.

As an example, consider a queue. A queue’s basic operations are enqueue (insertion at the end) and dequeue (removal from the front). These operations are consistent regardless of how the queue is implemented. Whether the queue is established with an array or a linked list, the user may enqueue and dequeue things without having to grasp the precise details of how these actions are executed.

This degree of abstraction simplifies and clarifies the code greatly. It reduces cognitive load for developers, making code maintenance and debugging easier. It also promotes better software design because developers can focus on the program’s architecture and logic rather than low-level implementation details.

* **Reusability and portability:**

Programming environments and languages become compatible with data structures that are independent in implementation, providing an extensive degree of reusability and portability. The reason for this high portability is because these data structures can be utilized in different languages and programming environments.

Consider a stack as an example. The stack concept, LIFO (Last In, First Out), is shared by all programming languages. If you’ve implemented a stack in C++, you can easily port that concept to Java, Python, or any other language. Only the syntax and specific language constructs change, while the underlying principles of the stack remain unchanged.

In software development, the time and effort required can be drastically reduced through reusability. When code is written in one language, it can be easily translated and implemented in a different language using it as a reference or guide. This creates a substantial opportunity for algorithms and data structures to be utilized across various projects and programming languages.

If the data structures are created in a way that is not specific to any one language, they become extremely portable and can be reused across various environments and programming languages. This further increases their reusability and usefulness.

In conclusion, the adaptability, abstraction abilities, and portability of implementation independent data structures like stacks and queues are what give them their power. Programmers may choose the optimal underlying structure for a specific problem thanks to their adaptability, which promotes efficient and effective coding. By being able to abstract away difficulties, they lessen the cognitive strain on developers and free them up to focus on higher-level problem-solving strategies and program design. The flexibility to apply these structures across a variety of settings and programming languages also encourages code reuse, which improves time and effort efficiency.

# ***References***

Baeldung (2023) *Best Data Structure for Dictionaries, Baeldung on Computer Science.* Available at: <https://www.baeldung.com/cs/language-dictionary-data-structure>

Cake, I. (2021) *Call Stack, Interview Cake.* Available at: <https://www.interviewcake.com/concept/java/call-stack>

cs.csub.edu (2017) *Abstract Data Types and Encapsulation, cs.csub.edu.* Available at: <https://www.cs.csub.edu/~melissa/cs350-f15/notes/ch11_notes.html>

faceprep (2020) *Space Complexity of Algorithms with Examples | FACE Prep, Faceprep.in.* Available at: <https://www.faceprep.in/data-structures/space-complexity/>

GeeksforGeeks (2020) *Applications of Priority Queue, GeeksforGeeks.* Available at: <https://www.geeksforgeeks.org/applications-priority-queue/>

GeeksforGeeks (2021) *What does ‘Space Complexity’ mean?, GeeksforGeeks.* Available at: [https://www.geeksforgeeks.org/g-fact-86/](https://www.geeksforgeeks.org/g-fact-86/%20)

GeeksforGeeks (2022) *Priority Queue in C++ Standard Template Library (STL), GeeksforGeeks.* Available at: [https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/](https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/%20)

GyanBlog (2019) *Min Priority Queue Implementation with Heap Data structure, GyanBlog.* Available at: [https://www.gyanblog.com/coding-interview/min-priority-queue-with-heap/](https://www.gyanblog.com/coding-interview/min-priority-queue-with-heap/%20)

IBM (2019) *The Call Stack, IBM.* Available at: <https://www.ibm.com/docs/en/i/7.2?topic=overview-call-stack>

Javatpoint (2021) *Priority Queue (Data Structures), javatpoint.* Available at: <https://www.javatpoint.com/ds-priority-queue>

KOCHAR, A. (2022) *Applications of Priority Queue | Queue, Prepbytes. Available* at: <https://www.prepbytes.com/blog/queues/applications-of-priority-queue/>

Linode Docs (2020) *Understanding Data Structures: Definition, Uses & Benefits, Linode Docs.* Available at: [https://www.linode.com/docs/guides/data-structure/](https://www.linode.com/docs/guides/data-structure/%20)

MDN (2020) *Call stack - MDN Web Docs Glossary: Definitions of Web-related terms | MDN, MDN.* Available at: <https://developer.mozilla.org/en-US/docs/Glossary/Call_stack>

MIT OpenCourseWare (2022) *Reading 12: Abstract Data Types, MIT OpenCourseWare.* Available at: <https://ocw.mit.edu/ans7870/6/6.005/s16/classes/12-abstract-data-types/>

Oracle, D. (2022) *Encapsulation and Abstract Data Types (ADT), DBA Oracle.* Available at: <http://www.dba-oracle.com/t_object_encapsulation_abstract.htm>

Programiz (2021) *Priority Queue Data Structure, Programiz. Available* at: <https://www.programiz.com/dsa/priority-queue>

Simplilearn (2023) *Priority Queue in Data Structure: Implementation & Types by Simplilearn, Simplilearn.* Available at: <https://www.simplilearn.com/tutorials/data-structure-tutorial/priority-queue-in-data-structure>

Soni Upadhyay (2023) *Time and Space complexity in Data Structure, Simplilearn.* Available at: [https://www.simplilearn.com/tutorials/data-structure-tutorial/time-and-space-complexity](https://www.simplilearn.com/tutorials/data-structure-tutorial/time-and-space-complexity%20)

Topics, S. (2022) *Space Complexity in Data Structure, Scaler Topics.* Available at: <https://www.scaler.com/topics/data-structures/space-complexity-in-data-structure/>

University of Wisconsin (2020) *Introduction: Abstract Data Types, Cs367.* Available at: <https://pages.cs.wisc.edu/~skrentny/cs367-common/readings/Introduction/>