**Report:**

**Advantages of using a Stack ADT for the FerryManagement system:**

A clear advantage is the fact that a Stack ADT adheres to a Last-In-First-Out Principle (LIFO) and applying this to a Ferry management system this is beneficial as the vehicle that was added most recently will be the first to be removed. Vehicles loading involves driving onto the ferry in the order they arrive, establishing a stack like structure and unloading in reverse order.

Additionally, a stack structure is relatively simple in its application, and this contributes to its maintainability. It has a limited set of operations, push, pop, peek etc. making it easy to implement, debug and maintain.

Such limited means to interact with the structure reduces complexity and room for errors in your code. In comparison to other data structures where managing pointers for multiple positions is required, a Stack ADT manages only a single access point (the top).

The predictive behavior of the Stack ADT is beneficial when debugging as it makes it easier to isolate the specific operation causing the occurrence of any errors. It is also beneficial regarding maintenance and scalability. A size limit can be added to a stack with minimal changes, and this is clearly advantageous for a Ferry with a limited amount of space on board (12 in this case).

The operations in a stack operations (push, pop) are efficient with a time complexity of O(1) meaning that the operations can be performed quickly no matter how many elements are in the stack. Regarding scalability, as the number of vehicles increases, the time taken for these operations will remain constant. This is beneficial in a ferry management system where the efficiency of loading and unloading many vehicles is key.

Furthermore, error handling can be enhanced without requiring significant structural changes, for example, the pop operation (remove a vehicle from the ferry) can’t be called on an empty stack or if trying to call the push operation if the stack is full. Pre-checks can be implemented such as checking if the stack is empty before “popping” to proactively check for invalid conditions that may result in an error.

**Other ADTs which could be used as an alternative means of implementation:**

A stack ADT would be ideal for a Ferry Management System that uses a LIFO principle, however there are alternative ADT’s that could also be implemented depending on how the vehicles will be loaded and unloaded onto the ferry.

A Queue ADT, implementing a First-In-First-Out principle, would be simple and useful in a system where the vehicles are loaded from the rear (enqueue) and disembark (dequeue) from the front, using a single lane and entering and leaving in the order they arrive at the port.

There are limitations to using this system as removing vehicles from the middle would not be supported and in the case of an emergency, the entire queue would have to disembark and would require enhancements for vehicles like Ambulances.

There are different kinds of queues that could be utilised such as an array-based queue or a linked list-based queue could be more suitable. Array-based queues could be useful for small ferries with a maximum capacity and similarly to the Stack ADT, its enqueue and dequeue operations have an efficient O(1) time complexity. However, it is limited in flexibility and a linked list-based queue would be more dynamic, and it won’t have a fixed size, it will however, be more complex to implement and maintain.

A linked list is advantageous as it is efficient in handling changes and dynamic scenarios making it useful for managing a ferry system. In comparison to arrays, a linked list provides advantages when adding and removing elements.

For example, in an array, inserting an element into the middle will involve moving the existing elements to accommodate the new element which can be time consuming and less efficient. In the worst-case scenario, inserting an element at the beginning of an array, all elements need to be moved. In contrast, a linked list involves creating a new node and adjusting the pointers of the surrounding nodes and will take a constant time of (O(1)) no matter where the element is being inserted.

It will however require more memory usage for node pointers and managing these pointers will increase complexity and the developer will need to weigh this against the benefits of using a linked list such as efficiency and flexibility. Extra resources such as memory and processing will be required to oversee the pointers connecting the nodes.

Each node requires additional memory to store the pointers, for example when storing integers, the node will hold both the integer value and the pointer to the next node. Extra processing is needed when adding or removing nodes and updating the pointer to ensure the correct sequence is maintained.

If you want to access a specific node on a linked list you must traverse sequentially, starting at the beginning and following the pointers until the desired node is reached, which can be less efficient than in comparison to the direct access offered by arrays. As a result, the time complexity is O(n) meaning that as the number of nodes increases, the time taken for the operation also increases linearly. Efficiency would be dependent on the size of the input data and would be more efficient for smaller datasets as opposed to more complex input.

A successful Ferry Management system relies on an evaluation of the trade-offs between different options of ADTs to achieve optimal performance levels. As we have discussed, a Stack ADT is suited for LIFO operations and its simplicity and efficiency for loading and unloading operations is advantageous. A stack is limited however in its flexibility and if the order needs to be changed for emergencies etc. then it wouldn’t be so suitable. In comparison, we have seen that a Queue ADT works well when following FIFO principles. An array-based ADT has a fixed sized capacity and would be inefficient for dynamic situations, and we have seen that a linked-list ADT provides more flexibility. A linked list can be adapted to manage both FIFO and LIFO loading operations, manage insertions and removals more efficiently and whilst there is a memory overhead required to manage the pointers, they manage it efficiently by only allocating memory for nodes being used.

Ultimately, choosing the correct ADT depends on factors such as loading and unloading patterns, managing dynamic situations and dealing with emergencies. Therefore, when weighing the trade-offs, the most appropriate ADT can be used for optimal efficiency and performance levels.

**Asymptotic analysis.**

This provides a framework for understanding the efficiency and scalability of algorithms. Big O, Big Omega and Big Theta are all forms of asymptotic analysis. By making use of these notations, we can gain a better understanding of the performance of an algorithm in the best, worst and average scenarios.

**Big O Notation:**

It is a mathematical tool used to characterize the efficiency and performance of an algorithm describing how its resource needs, (time or space) runtime and memory, change in proportion to the size of its input. Time complexity describes how the time taken by an algorithm grows as the size of the input increases. Space complexity describes the amount of memory used as the input size increases. Big O expresses the upper bound on the growth rate of an algorithm. The input size refers to the size of the data that is being processed, for example the number of elements in an array.

Examples of common notations include:

* O(n) (Linear): meaning the running time grows linearly with the input size, therefore if the input doubles the time will reflect this by doubling too.
* O(log n) (Logarithmic): time grows logarithmically. This means it grows slowly and is an efficient algorithm particularly for large datasets.
* O(1) (constant): takes the same amount of time regardless of the input. Highly efficient as performance is not affected by an increase in size of the input.
* O(n^2) (quadratic): if the input size doubles the runtime will quadruple. It is inefficient for large datasets.

Big O can therefore provide a prediction on the performance and scalability of an algorithm as the size of the input increases, and this is advantageous for multiple reasons.

It is useful for large data sets, as applications manage large amounts of data it is vital to have efficient algorithms to be able to process and analyze datasets effectively. It helps compare different algorithms that could be used to tackle the same challenges and provides the knowledge to make informed decisions. If you understand the time and space complexity you can therefore select the most effective algorithm. For example, when choosing the best sorting algorithm, Big O can help aid this decision. For a small dataset, Bubble Sort (O(n²)) could be sufficient, but for larger datasets Merge Sort (O(n log n)) would perform at consistently higher level.

As we can see it can predict performance degradation and can therefore help meet the scalability requirements of an application. If large datasets need to be processed then an algorithm with a lower time complexity such as O(n) O(log n) should be selected. By having understandings of high time complexities, it means Big O can help to avoid potential bottlenecks in performance and can help with resource planning needs in terms of memory requirements etc.

**Alternatives to Big O.**

There are of course alternatives to Big O which can also be used to analyze the efficiency of an algorithm. Big Omega (Ω), Big Theta (Θ) are effective methods that can be applied. In contrast to Big O, Big Omega will describe the lower bound of the running time of an algorithm in the most optimal scenario offering the minimum time that it will take whatever the amount of input data. By understanding lower bounds, it is possible to determine the theoretical minimum amount of computational effort for an algorithm to solve a given problem, irrespective of input size or problem complexity.

This is beneficial in real time systems where tasks must be completed in a set time frame and will rely on predictive performance to ensure reliability. A real-world example is in the use of autonomous vehicles such as Tesla which rely on real-time systems for safety and efficiency. Algorithms are in place in the detection of obstacles, i.e. pedestrians and must adhere to strict time constraints to prevent accidents and in other decision-making processes and path planning. Furthermore, best case performance can allow vehicles to plan routes extremely quickly.

Conversely, Big Theta (Θ) notation specifies both upper and lower bounds establishing a “tight bound” on the growth rate of an algorithm. It establishes that the running time of the algorithm will neither grow significantly faster nor slower than the predicted rate, the growth rate is limited to a specific range. This knowledge can help understand a more expected performance rate as opposed to the best- and worst-case scenarios and offers consistency and more precise analysis.

**Static vs. Dynamic ADTs**

Data structures are techniques used for storing and organising data ensuring efficient operations whilst minimising time and memory complexities. These can be defined as static or dynamic.

To illustrate the difference between static and dynamic ADTs an array is good example.

Static, as the name suggests, means the size of the structure is fixed. The values within the structure can be modified like adding or removing elements within the allocated memory however, the amount of memory allocated at compile time can’t change. The size of the structure is established before the program is executed for example “int arr[10];” declares that the array can hold a maximum of ten integers.

This is advantageous as it:

* Ensures a fast access time as the memory address of each element within the structure can be easily calculated based on its index without needing to search through the entire structure.
* Its fixed memory allocation means a predictable memory usage required by the program and can help prevent memory related issues like insufficient memory errors especially in memory constrained environment.
* Such a fixed size and predictable memory can result in better complier optimisations and an improved cache utilisation. For example, compliers can optimise array indexing operations because the exact memory address of the element is known.

Limitations of a static structure:

* Its fixed size results in limited flexibility, if the number of elements that needs to be stored exceeds the pre-defined size, then a static structure is not suitable. In contrast, if the number is less that the pre-defined size, it can lead to memory waste.
* Unsuitable for dynamic data where the elements are unpredictable or changing.
* Potential for errors: buffer overflows when data is written beyond the allocation. Leads to data corruption, program crashes and security vulnerabilities.

In contrast, Dynamic ADTs are flexible in size and can be modified during program execution. This means that this structure is adaptable and can change dynamically to accommodate changing data requirements.

Benefits include:

* Its flexibility is advantageous as without a predetermined size new elements can be added and avoids a buffer overflow that can occur in static arrays. On the other hand, if data is removed, the memory allocation can be reduced or reused, avoiding waste.
* Useful when handling variable data, for example in real world applications data size can fluctuate during the execution of the program. Social media platforms are constantly adding new accounts and users are rapidly generating vast amounts of data. As a result, these platforms must dynamically adjust the size of their structures.
* Their dynamic nature helps to improve efficiency by reducing fragmentation and avoiding wasting memory helps to optimise memory utilisation,

Limitations include:

* There are extra time and resources used by the mechanisms that manage the dynamic memory, and these overheads can impact the performance levels of the program. This is more prevalent in systems with limited resources or in time-constraints.
* They are more complex to implement and maintain. There are challenges around dynamic memory allocation and deallocation, pointer handling, data manipulation and debugging.
* Potential for errors for example, memory leaks which can lead to potential crashes.

As the evidence shows, an array is a good example to highlight the differences between static and dynamic arrays.

In summary, both static and dynamic ADTs offer advantages and disadvantages, a static array has a fixed size which was predetermined at compile time, for example “int arr[10];” declares that this array can hold a maximum of ten integers. Memory has already been allocated and cannot be changed during the execution of the program. If you try and add more elements to store in the array it will result in an overflow as previously discussed. This structure will be inefficient in situations where there are frequent changes.

Dynamic arrays can change size during the execution of the program, memory will be allocated at run time and as we have discussed, its flexibility is beneficial allowing it to grow or shrink to meet the needs of the changing data requirements. These structures can also help to avoid memory waste by only allocating the necessary amount of memory. They are however more complex to implement and maintain and has overheads associated with dynamic memory.

The choice between static and dynamic ADTs will ultimately depend on the requirements of the application. For applications with predictable and smaller data sizes, static structures would be a suitable choice, however, for applications that manage dynamic data, unpredictable data sizes and changes, then dynamic ADTs would be more effective.

**Implementation independent data structures.**

Implementation independent data structures emphasise a separation between the what and the how, meaning the behaviour of the data structure and its implementation. These structure types specify the available options, (add, remove etc) that can be performed, without revealing their internal implementation details.

A queue following the FIFO principle is an example of an implementation data structure, it has operations such as enqueue (adding an element) and dequeue (removing an element) and can be implemented using a circular array. Additionally, a stack ADT with operations such as pop, pop, peek etc can be implemented using an array or a linked list to store elements.

Advantages of using such structures include:

* Flexibility: As mentioned above, the same ADT can be implemented using different structures based on the needs of the application. Different implementations can be utilised taking into consideration factors such as memory constraints, performance requirements etc. This adaptability means that as long as the interface remains consistent, changes to the underlying implementation are isolated and the rest of the program remains unaffected by any modifications.
* Reusability: Abstracting the implementation permits greater code reusability, as the same ADT interface across different parts of the application. This can help to reduce duplication or code redundancy, for example, you can define one stack interface, create a single implementation using an array and reuse it throughout the program. This is advantageous as it helps to reduce the size of your codebase and if any modifications are needed in the implementation, only the implementation class needs changed. The rest of the code that uses the interface will remain unaffected.
* Problem solving: By abstracting away the implementation details, focus can turn to designing solutions to complex problems and this can help improve problem solving Furthermore, it can lead to an improvement in decision-making and improvements in the design of the structure. The scope for adaptability makes implementation independent data structures valuable for building scalable and maintainable systems.

**Bibliography**

Applications, advantages and disadvantages of stack (2022) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-stack/ (Accessed: 2 January 2025).

Vasdeva, R. (2023) Applications, advantages, and disadvantages of stack, Medium. Available at: https://medium.com/@raghav.vasdeva/applications-advantages-and-disadvantages-of-stack-6ede5f8af5f0 (Accessed: 2 January 2025).

Queue Data Structure, Tutorialspoint.com. Available at: https://www.tutorialspoint.com/data\_structures\_algorithms/dsa\_queue.htm (Accessed: 2 January 2025).

Improve, K.F. (2012) Asymptotic analysis, GeeksforGeeks. Available at: https://www.geeksforgeeks.org/asymptotic-notation-and-analysis-based-on-input-size-of-algorithms/ (Accessed: 2 January 2025).

Big O notation tutorial - A guide to big O analysis (2018) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/ (Accessed: 2 January 2025).

Lima, R. (2018) Big O Notation and Other Alternatives, Richardson Lima. Available at: https://www.richardsonlima.com.br/programming/2018/08/19/the-big-O.html (Accessed: 2 January 2025).

Difference between big O vs Big Theta Θ vs Big Omega Ω notations (2020) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/difference-between-big-oh-big-omega-and-big-theta/ (Accessed: 2 January 2025).

Static data structure vs dynamic data structure (2018) GeeksforGeeks. Available at: https://www.geeksforgeeks.org/static-data-structure-vs-dynamic-data-structure/ (Accessed: 3 January 2025).