Course: MsDS

Data Structures and Algorithms

Module: 2

Learning Objectives:

1. Understand the essential distinctions between static and dynamic memory allocation, comparing arrays and linked lists in particular.
2. Understand the idea of access-restricted lists and how they may be used in different computing environments.
3. Understand the fundamentals, functionality, and applications of stacks in data structure management.
4. Examine the features, traits, and real-world applications of queues and DEQueues in computing activities.

Structure:

2.1 Static Allocation vs Dynamic Allocation

Memory allocation plays a fundamental role in the fields of software engineering and computer science, acting as a foundational element that underpins the development of efficient and successful programmes. The core of this approach revolves around a crucial differentiation: static memory allocation vs dynamic memory allocation. Comprehending this difference has significant importance for those seeking to become software developers since the choice between these two options has a direct influence on programme performance, maintenance, and scalability.

2.1.1 Static Allocation

Static memory allocation refers to the procedure in which a predetermined quantity of memory is assigned at the time of compilation. The dimensions and arrangement of the data are predetermined before the program's execution, and they stay constant during the program's runtime. The array stands out as the most notable representation of this particular method. When an array is defined, it allocates a certain amount of memory depending on its data type and size. This implies that the amount of memory is predetermined and cannot be altered, resulting in arrays being very simple to create and handle.

Nevertheless, this predictability is not without its own set of obstacles. A notable constraint is the lack of adaptability or rigidity. Once the size of an array is established, it is not possible to alter it without the need to declare it again and, in the majority of situations, recompile the whole programme. Inefficient memory utilisation may occur when an array is defined with excessive size. However, only a portion of it is used, resulting in suboptimal memory allocation. Conversely, an array that has insufficient capacity to accommodate the necessary data may lead to data overflow or need the implementation of intricate solutions. In addition, the process of adding or removing members inside an array necessitates the reordering of components, resulting in potential computational inefficiency.

2.2.2 Dynamic Allocation

In contrast to static memory allocation, dynamic memory allocation enables

programs to request memory spaces during runtime, therefore relieving developers from the need to specify the precise amount of memory required by a data structure prior to execution. The process of allocation is executed via the use of pointers, which are specialised variables that contain the addresses of other variables or memory regions. The linked list, which is composed of nodes interconnected by pointers, serves as a prime illustration of a data structure that derives advantages from the use of dynamic memory allocation.

The primary benefit of this approach lies in its inherent flexibility. The dynamic allocation of memory allows data structures to adjust their size as needed, hence optimising memory utilisation and tolerating unexpected data flows. Moreover, dynamic allocation may enhance the efficiency of operations like insertion and deletion, particularly when performed inside the interior of a data structure. This is due to the fact that these actions often need just pointer alterations.

However, dynamic allocation is not devoid of its disadvantages. The inclusion of manual memory management in programmes offers a heightened degree of complexity, necessitating developers to actively control the allocation and deallocation of memory. Inadequate management of this issue may lead to the occurrence of memory leaks, when memory regions become inaccessible yet remain allocated, squandering important resources. Moreover, the frequent allocation and deallocation of memory might result in memory fragmentation, which has the potential to impede programme execution speed.

2.2 Access Restricted Lists

Within the complex realm of data structures, a category known as access-restricted lists arises as a subset that encompasses a fundamental principle: the imposition of limitations on the methods by which data may be accessed, added, or withdrawn. These structures provide specialised functions in the field of computing, guaranteeing the integrity of data, facilitating efficient operations, and optimising performance for certain use cases. Due to their inherent importance, it is essential to elucidate the fundamental nature, operational mechanisms, and practical relevance of these entities in real-life contexts.

2.2.1 Nature and Variants of Access Restrictions

The fundamental principle behind access-restricted lists is the imposition of limitations on some processes, either by confining them to one extremity of the list or by subjecting them to specified constraints. This kind of access control often streamlines computational procedures by enabling developers to anticipate the locations of insertions or deletions. There are many major categories of access restriction lists that may be identified.

- **Stacks:** These systems function based on the Last In, First Out (LIFO) concept. This implies that the most recent element that has been added to the stack is the first one to be withdrawn from it. Consider a hypothetical scenario where there exists a vertical arrangement of books resembling a stack. In this particular scenario, the only permissible actions are the removal of the uppermost book or the addition of a new book only to the highest position of the stack. The imposition of this constraint guarantees the attainment of efficient O(1) operations, although at the cost of restricting direct access to the centre of the list.
- **Queues:** Queues adhere to the First In, First Out (FIFO) concept, wherein the first element inserted is guaranteed to be the first one to be extracted. Imagine a scenario where individuals form a queue at a financial institution, with the person who arrives earliest occupying the first position and thereby receiving service ahead of others. Queues often play a role in situations that need job scheduling or the execution of breadth-first searches in graph algorithms.
- **Double-Ended Queues (DEQueues):** These ideas generalise the principles of both stacks and queues. Both ends of the list have the capability to undergo the addition or removal of elements. Double-ended queues, also known as DEQueues, are very flexible data structures that may be used in a variety of scenarios. These scenarios often include the need for a mix of stack and queue operations.

The aforementioned lists are well-recognised in the field. However, there are ongoing developments in the form of modifications and hybrids that emerge in response to particular algorithmic requirements and the ever-evolving landscape of computational obstacles.

2.2.2 Practical Implications and Use Cases

The efficacy of access-limited lists extends beyond theoretical implications, as their practical significance is profoundly evident in many computer applications. Stacks play a fundamental role in the evaluation and parsing of expressions inside computer languages. When a method invokes another method inside a programme, the execution of the invoking method is temporarily halted, and its relevant data is stored on a stack until the invoked method finishes its execution. Upon completion, the state of the preceding procedure is removed from the stack and then restarted.

On the contrary, queues play a crucial role in the scheduling of tasks. Operating systems often maintain a queue of processes that are in a state of waiting for specified tasks, such as allocation of CPU time or input/output (IO) activities. The implementation of a queuing mechanism guarantees the sequential processing of processes, hence upholding principles of equity and predictability in the functioning of a system.

DEQueues, also known as double-ended queues, are often used in many algorithms, such as the sliding window approach. This technique necessitates the effective management of a collection of components that may be accessed, added, or deleted from either end of the queue.

2.3 Stacks

Stacks are a basic data structure that operates based on the Last In, First Out (LIFO) principle. In the context of a stack data structure, components are inserted (pushed) and deleted (popped) from a singular end, sometimes referred to as the top. The implementation of limited access in this context guarantees efficient O(1) operations for both push and pop actions. Stacks are a crucial component in a multitude of computing contexts, including the evaluation of expressions in programming languages, the management of execution sequences in recursive function calls, and the maintenance of correct bracketing in code syntax. The inherent simplicity and remarkable efficacy of their structure make them an essential instrument in the realm of algorithm creation and computer science pedagogy.

2.4 Queues

Queues are a kind of linear data structure that adheres to the First In, First Out (FIFO) principle, guaranteeing that the first element enqueued will be the first one dequeued. The concept of a queue may be seen as being equivalent to a real-world scenario where individuals form a queue, with components or objects being added to the back of the queue and removed from the front. The aforementioned systematic arrangement of removal serves to construct queues as a key instrument for situations that need sequential processing.

In the field of computational science, queues play a crucial role in a wide range of applications. Operating systems use queues as a means of task scheduling, facilitating the management of the sequence in which activities are held in anticipation of CPU allocation. In the context of computer networks, the management of data packets that are awaiting transmission or processing often involves the use of queues to maintain a structured and methodical approach. Moreover, within the context of algorithm design, queues play a crucial role in the implementation of breadth-first search methodologies on graphs or trees.

In order to streamline these procedures, a queue generally offers a minimum of two fundamental actions: enqueue (which adds an item to the rear) and dequeue (which removes and returns the front item). Certain implementations may additionally have the capability to perform peek operations, which allow for the viewing of the front item without removing it. Due to its inherent organisation and systematic characteristics, queues play a fundamental role in the field of data structures and their associated applications.

2.5 DEQueues

Double-Ended Queues, sometimes referred to as DEQueues or "decks," are an expanded and adaptable iteration of the conventional queue data structure. In contrast to the traditional queue, which only permits actions at one end, DEQueues enable the addition and removal of components from both the front and the back. The integration of dual functionality combines the capabilities of stacks and queues, hence providing a wider array of applications and use cases.

DEQueues inherit the operational concepts of queues while also introducing a higher degree of flexibility. The DEQueue is often characterised by four major processes, namely, insertion at the front, insertion at the back, deletion from the front, and deletion from the rear. This implies that a DEQueue has the capability to operate as either a stack or a queue, or maybe a hybrid of both, depending upon the specific application. The inherent flexibility of DEQueues renders them very advantageous in scenarios where there is a need to access or modify data from both ends without necessitating the transfer of the whole dataset.

The pragmatic applicability of DEQueues may be noticed in several computing settings. An illustrative instance is the sliding window algorithm, which is used in the processing of arrays and strings. The technique described above demonstrates the effective use of DEQueues to monitor items inside a designated "window" as it traverses the dataset. This approach enables the seamless insertion or removal of components from either end of the window.

Another potential use might be seen in certain cache replacement techniques. In the context of cache management, the "Least Recently Used" (LRU) method uses DEQueues to effectively monitor cache pages. This technique efficiently adjusts the order of pages in the cache depending on their access frequency, eliminating the need to reorganise the whole list.

2.6 Summary

- Module 2 begins by juxtaposing the notions of static allocation, as exemplified by arrays, with dynamic allocation, as epitomised by linked lists. Arrays provide a fixed-size storage mechanism that guarantees contiguous memory allocation and efficient access. In contrast, linked lists have the advantage of dynamic resizing, allowing for flexible growth or reduction in size. However, linked lists may incur additional complexity and entail non-sequential memory access.
- The module explores the concept of access-restricted lists, highlighting the significance of restricting data access to enhance operational efficiency, predictability, and optimisation. The provided lists are customised to suit certain

computing settings, hence guaranteeing the preservation of data integrity and the optimisation of job execution.

- ❖ Stacks and queues are fundamental data structures that adhere to the Last-In-First-Out (LIFO) and First-In-First-Out (FIFO) principles, respectively. Stacks play a crucial role in several contexts, such as managing function calls and evaluating expressions, due to their exclusive top access. On the other hand, queues play a crucial role in several applications, such as work scheduling and breadth-first search algorithms, due to their organised front-rear processes.
- ❖ The module concludes with an examination of DEQueues, which integrate the features of both stacks and queues. DEQueues provide a versatile method for data management by allowing operations to be performed at both ends. This characteristic makes them particularly advantageous in many applications, such as the sliding window technique and cache replacement schemes.

2.7 Keywords

- **Static Allocation:** The memory allocation approach that involves a predefined size, often connected with arrays, is known as static memory allocation.
- **Dynamic Allocation:** Dynamic memory allocation is a versatile technique that allows for the adjustment of memory size as required, often seen in the context of linked lists.
- **Access Restricted Lists:** Data structures that impose restrictions on data access or modification based on specified criteria or locations.
- **Stacks:** A data structure that follows the Last-In-First-Out (LIFO) principle, wherein the most recently inserted element is the first to be deleted.
- **Queues:** The structure follows a First-In-First-Out (FIFO) approach, where the first element inserted is the initial element to be withdrawn.
- **DEQueues:** Double-ended queues, also known as dequeues, are data structures that enable the insertion and removal of elements from both ends. They combine the characteristics of both stacks and queues.

2.8 Self-Assessment Questions

1. Give an explanation of the distinctions between static and dynamic memory allocation, highlighting the benefits and drawbacks of each.
2. Describe the LIFO concept and how it affects the management of data. What role does this concept play in stack functionality?
3. Describe a situation in the real world where queues would be the best data structure to use, and then explain why you made that decision.
4. DEQueues provide a combination of attributes from stacks and queues. Explain DEQueues' operating features and why they may be beneficial in certain computing settings.
5. Access-restricted lists fulfil certain computational requirements. Describe one such kind of list, its special features, and a suggested use case where it would be most effective.

2.9 Case Study

Title: Implementing Efficient Check-in Systems at MegaAir Airport

Introduction:

The difficult challenge of timely and effective management of high numbers of people is one that airports across the globe must overcome. The fundamental choice of data structures in the systems that run these hubs forms the basis of a seamless airport experience.

Case Study:

One of the largest international airports in the world, MegaAir Airport, has lately experienced substantial check-in delays that have left passengers dissatisfied and caused delayed flights. Initial examinations revealed the shortcomings of the data structure decisions made for the present system.

Background:

In the past, MegaAir checked in passengers using an array-based approach (static allocation). The system regularly had overflowing and underutilisation due to the erratic nature of airline reservations and cancellations, resulting in the waste of resources and laborious modifications.

Your Task:

MegaAir Airport has recruited you as a top data structures consultant to update their check-in procedure. In order to provide quick check-ins and efficient resource allocation, you must evaluate the present difficulties and suggest a data structure solution that can dynamically adapt to the varying passenger loads.

Questions to Consider:

1. Would a dynamic allocation strategy be more appropriate, given how unpredictable passenger check-ins are?
2. How can DEQueues simplify the procedure for travellers with various service requirements (such as those travelling in business class, families, or alone)?
3. What operational circumstances at the airport could a queue-based system be most advantageous in?
4. Could a mix of several data structures provide a more complete answer?

Recommendations:

It is advised to build a dynamic allocation method, such as linked lists, after evaluating MegaAir's present system to account for the variable check-in volumes. Additionally, the introduction of DEQueues allows accommodating prioritised check-ins, guaranteeing that premium or special-need customers are handled quickly. A queuing system would speed up the procedure and shorten wait times for general boarding and luggage drop-offs.

Conclusion:

The effectiveness and user experience of complicated systems like airport check-ins may be significantly improved by using the appropriate data structures. MegaAir Airport can greatly reduce delays, maximise resources, and improve passenger happiness by switching to a more flexible and strategic set of data structures.

2.10 References

1. Goodrich, M.T., Tamassia, R. and Goldwasser, M.H., 2014. *Data structures and algorithms in Java*. John Wiley & sons.
2. Mehlhorn, K., 2013. *Data structures and algorithms 1: Sorting and searching* (Vol. 1). Springer Science & Business Media.
3. Storer, J.A., 2002. *An introduction to data structures and algorithms*. Springer Science & Business Media.
4. Drozdek, A., 2012. *Data Structures and algorithms in C++*. Cengage Learning.
5. Lafore, R., 2017. *Data structures and algorithms in Java*. Sams publishing.