HUST

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY.

OBJECT-ORIENTED PROGRAMMING

IT3100E

INSTRUCTOR: PROF. DR TRAN THE HUNG

Class ID: 147839

Demonstration of operations on stack, queue, list

Group Members:
Nguyen Cong Huan - 20227974
Phan Gia Do - 20226026
Nguyen Cong Dat - 20226022
Tran Gia Khanh - 20226048

Abstract

This report demonstrates the fundamental operations performed on three essential data structures: stack, queue, and list. Each data structure is explained in detail, including its characteristics and common applications. The report then presents algorithms and code implementations for operations such as insertion, deletion, and traversal on each data structure. Through practical demonstrations and code examples, readers gain a comprehensive understanding of how these data structures function and how to manipulate them effectively in various programming scenarios.

1 Introduction

The term "data structure" refers to a way of organizing, storing, and managing data in a computer so that it can be accessed and manipulated efficiently. Essentially, it defines a specific layout or arrangement of data elements in memory, along with the operations that can be performed on them.

Data structures are crucial in computer science and programming because they provide a foundation for representing and working with data in algorithms and applications. They enable efficient data storage, retrieval, and manipulation, which are essential for developing efficient software systems.

In summary, a data structure encompasses both the organization of data elements and the algorithms or operations that can be applied to them, facilitating effective data management and computation. Among the plethora of data structures available, three fundamental ones stand out: the stack, queue, and list. Each of these structures serves unique purposes and exhibits distinct characteristics that make them indispensable in various computational tasks and problem-solving scenarios.

In this paper, we aim to explore and demonstrate the basic operations of three fundamental data structures such as: create, insert, sort, find, delete elements in the structures.

2 Linked list

2.1 Introduction

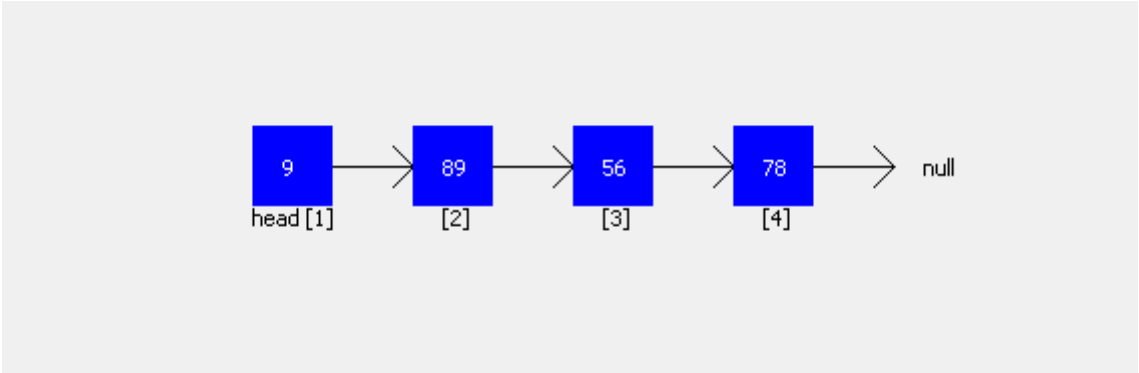


Figure 1: Linked list

A linked list is a fundamental data structure in computer science used for storing and organizing data in a linear fashion. Unlike arrays, which store elements in contiguous memory locations, linked lists utilize a dynamic structure where each element, called a node, consists of two parts: the data itself and a reference (or pointer) to the next node in the sequence. This arrangement allows for flexible memory allocation and efficient insertion and deletion operations, making linked lists a versatile choice for various programming tasks and applications.

Linked lists come in different forms, with the two most common types being singly linked lists and doubly linked lists. In a singly linked list, each node contains data and a pointer to the next node in the sequence, while in a doubly linked list, each node additionally holds a pointer to the previous node, enabling traversal in both forward and backward directions.

The absence of a fixed size and the ability to dynamically allocate memory make linked lists ideal for scenarios where the size of the data structure is unknown or needs to be frequently modified. Furthermore, linked lists offer constant-time insertion and deletion at the beginning or end of the list, which can be advantageous over arrays for certain operations.

In this report, we will delve into the intricacies of linked lists, exploring their characteristics, operations, and applications in depth. Through clear explanations, illustrative examples, and code implementations, readers will gain a comprehensive understanding of linked lists and how to effectively utilize them in various programming scenarios.

2.2 Characteristics

- Dynamic Memory Allocation: Linked lists allow for dynamic memory allocation, meaning nodes can be allocated and deallocated from memory as needed. This flexibility makes linked lists suitable for situations where the size of the data structure may change frequently.

- Sequential Access: Linked lists provide sequential access to elements, meaning traversal through the list is done sequentially from one node to the next. This allows for efficient iteration over the elements of the list.

- Variable Size: Linked lists can grow or shrink in size dynamically. Unlike arrays, which have a fixed size, linked lists can accommodate varying numbers of elements without needing to reallocate memory.

- Node Structure: Each element in a linked list, known as a node, consists of two parts: the data itself and a reference (or pointer) to the next node in the sequence. In the case of doubly linked lists, each node also contains a reference to the previous node.

2.3 Operations

- Insertion: Linked lists support various insertion operations, including insertion at the beginning, end, or at any specific position within the list. These operations involve creating a new node and updating the appropriate pointers to maintain the integrity of the list.

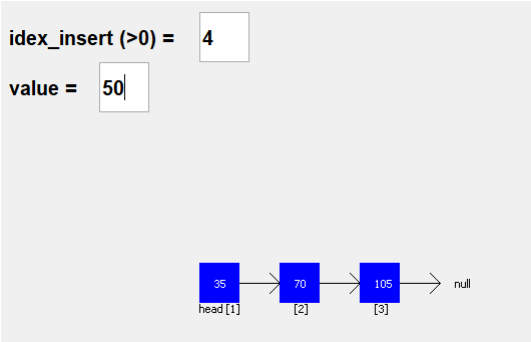


Figure 2: Before inserting

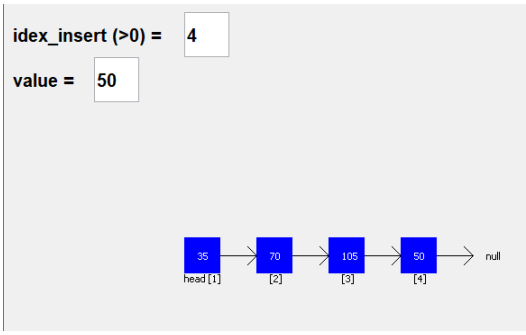


Figure 3: After inserting

- Deletion: Similarly, linked lists allow for deletion operations, such as removing the first node, last node, or a node at a specified position. Deletion involves updating the pointers of neighboring nodes to bypass the node being removed.

- Searching: Linked lists support searching for a specific element by traversing the list until the desired element is found or until the end of the list is reached. However, searching in a linked list typically has linear time complexity O(n), where n is the number of elements in the list

3 Stack

3.1 Introduction

Queues are essential data structures in computer science, facilitating first-in, first-out (FIFO) processing of elements. Unlike arrays or linked lists, queues prioritize elements based on their

arrival time rather than their position. Each element, or task, enters the queue in sequence and exits in the same order.

Queues offer dynamic memory allocation and efficient insertion and deletion operations. They typically consist of nodes, with pointers indicating the front and rear of the queue. Two primary types of queues exist: linear queues, which adhere strictly to FIFO principles, and priority queues, which consider element priority during processing.

Due to their orderly processing and versatility, queues find applications in job scheduling, resource management, and algorithms like breadth-first search.

This report delves into queues' characteristics, operations, and applications, offering clear explanations, examples, and code implementations for effective utilization in programming scenarios.

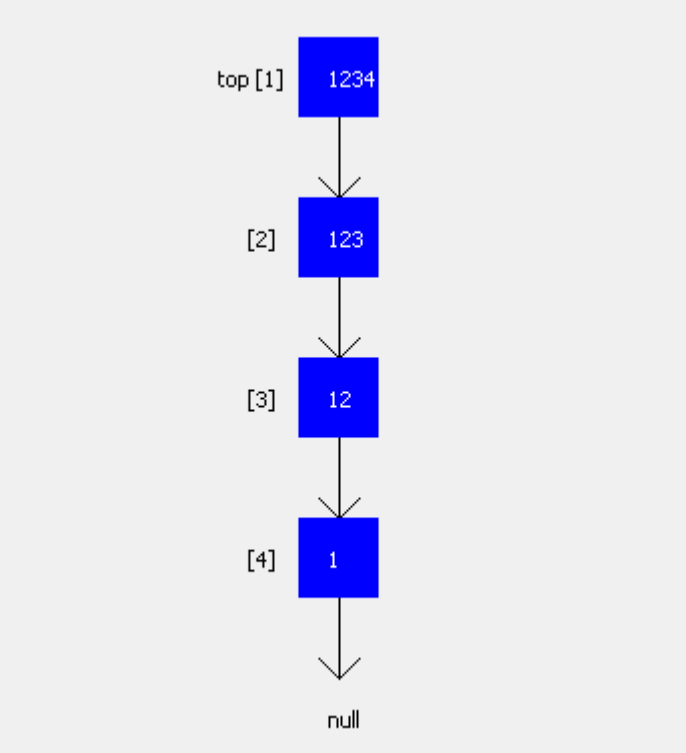


Figure 4: Stack

A stack is a fundamental data structure in computer science used for storing and managing data in a linear fashion based on the Last In, First Out (LIFO) principle. Unlike arrays, which store elements in contiguous memory locations, stacks operate through a dynamic structure where elements are added (pushed) and removed (popped) from the top of the stack. This arrangement allows for efficient management of data and supports operations that require a strict ordering of elements, making stacks a versatile choice for various programming tasks and applications.

Stacks come in different forms, with the two most common implementations being array-based stacks and linked list-based stacks. In an array-based stack, elements are stored in a contiguous block of memory, and the stack's size can be fixed or dynamically resized. In a

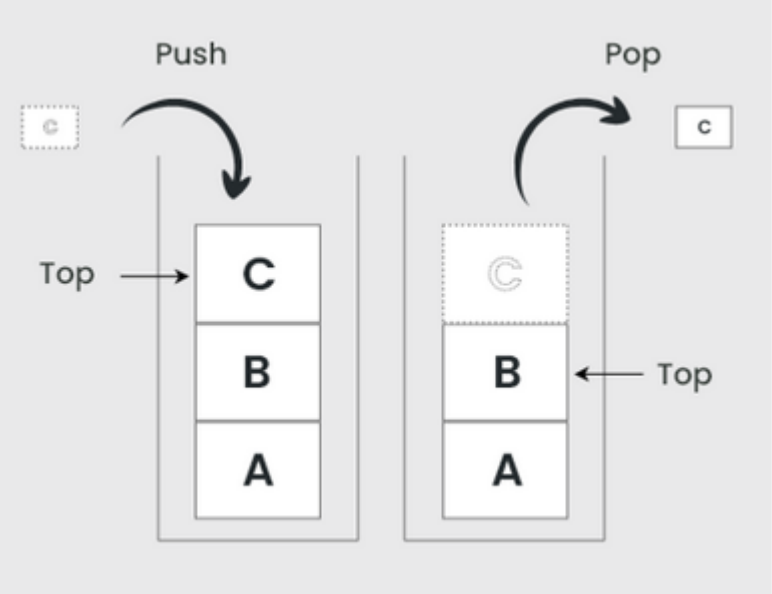
linked list-based stack, each element, called a node, consists of the data itself and a reference (or pointer) to the next node in the sequence, allowing for flexible memory allocation and efficient operations.

The LIFO ordering of elements makes stacks ideal for scenarios where the most recently added element must be accessed first. This is particularly useful in applications such as function call management, expression evaluation, and implementing undo mechanisms. Furthermore, stacks offer constant-time insertion and deletion at the top of the stack, which can be advantageous over arrays for certain operations.

In this report, we will delve into the intricacies of stacks, exploring their characteristics, operations, and applications in depth. Through clear explanations, illustrative examples, and code implementations, readers will gain a comprehensive understanding of stacks and how to effectively utilize them in various programming scenarios.

3.2 Characteristics

- LIFO Principle (Last In, First Out): The last element added to the stack is the first one to be removed. This order of operations is fundamental to the stack's functionality.



- Dynamic Size: The size of the stack can grow or shrink as elements are pushed or popped, making it a flexible data structure.

- Sequential Access: Elements are accessed in a sequential manner, starting from the top of the stack.

- Memory Efficient: Stacks can be implemented in a memory-efficient manner, particularly when using a linked list implementation where nodes are created as needed.

3.3 Operations

- Push: This operation adds an element to the top of the stack. It involves two steps:
  -Incrementing the stack pointer (if implemented using an array).
  -Assigning the new element to the position pointed to by the stack pointer.

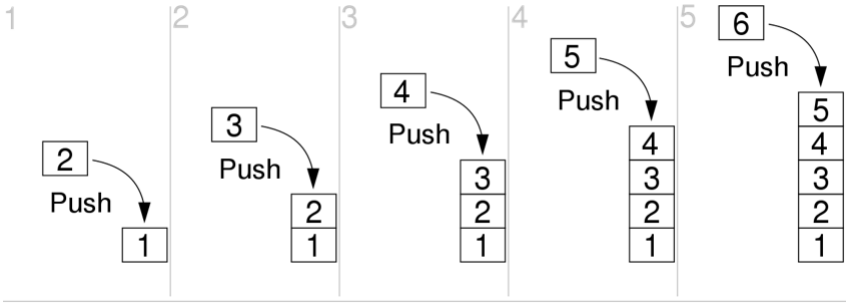


Figure 5: Push

- Pop: This operation removes the top element from the stack. It also involves two steps:
  -Accessing the element at the top of the stack.
  -Decrementing the stack pointer (if implemented using an array).

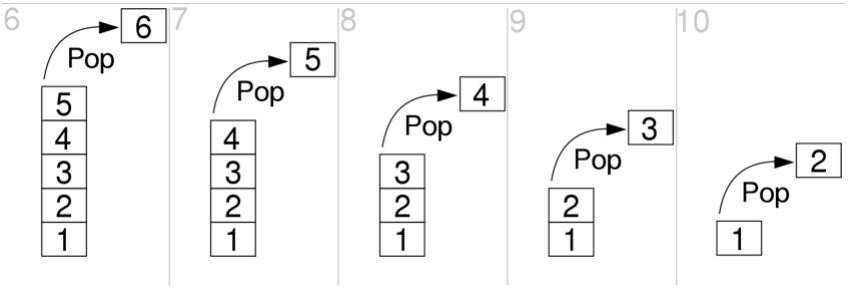


Figure 6: Pop

- Peek (or Top): This operation returns the element at the top of the stack without removing it. It involves accessing the element at the top of the stack without modifying the stack itself.

4 Queue

4.1 Introducing

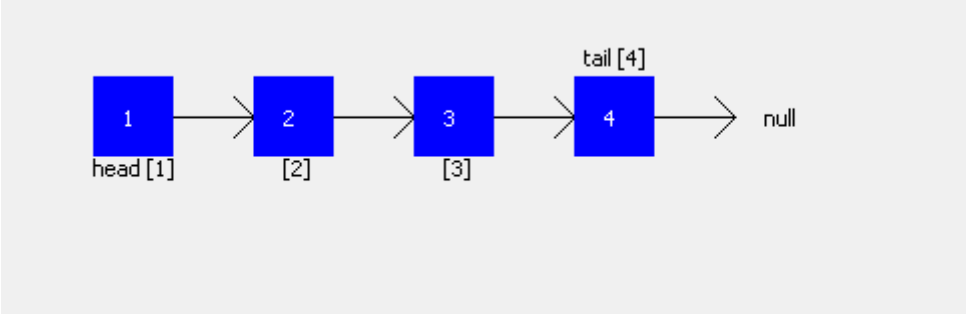


Figure 7: Queue

Queues are essential data structures in computer science, facilitating first-in, first-out (FIFO) processing of elements. Unlike arrays or linked lists, queues prioritize elements based on their arrival time rather than their position. Each element, or task, enters the queue in sequence and exits in the same order.

Queues offer dynamic memory allocation and efficient insertion and deletion operations. They typically consist of nodes, with pointers indicating the front and rear of the queue. Two primary types of queues exist: linear queues, which adhere strictly to FIFO principles, and priority queues, which consider element priority during processing.

Due to their orderly processing and versatility, queues find applications in job scheduling, resource management, and algorithms like breadth-first search.

This report delves into queues' characteristics, operations, and applications, offering clear explanations, examples, and code implementations for effective utilization in programming scenarios.

4.2 Characteristics

- Dynamic Memory Allocation: Queues, like linked lists, offer dynamic memory allocation, allowing nodes to be allocated and deallocated from memory as needed. This adaptability suits scenarios where the queue's size may fluctuate frequently.

- Sequential Access: Queues provide sequential access to elements, facilitating traversal from one node to the next in a linear manner. This inherent sequentiality enables efficient iteration over the queue's elements, supporting orderly processing.

- Variable Size: Unlike arrays with fixed sizes, queues can dynamically grow or shrink in size. They accommodate varying numbers of elements without necessitating memory reallocation, thereby offering flexibility in managing changing workloads or data volumes.

- Node Structure: Each element in a queue, represented as a node, comprises two components: the actual data and a reference (or pointer) to the next node in the sequence. This structure ensures orderly FIFO processing. In the context of priority queues or double-ended queues, additional references may exist, enabling more intricate data management operations.
  Understanding these fundamental characteristics illuminates the versatility and utility of queues across various computational tasks, from task scheduling to resource management and beyond.

4.3 Operations

- Enqueue: This operation adds an element to the rear of the queue, expanding the queue's size.
- Dequeue: Removing an element from the front of the queue, adjusting the queue's size and shifting subsequent elements forward.
- Front (or Peek): Accessing the element at the front of the queue without removing it, enabling inspection of the next item to be dequeued.

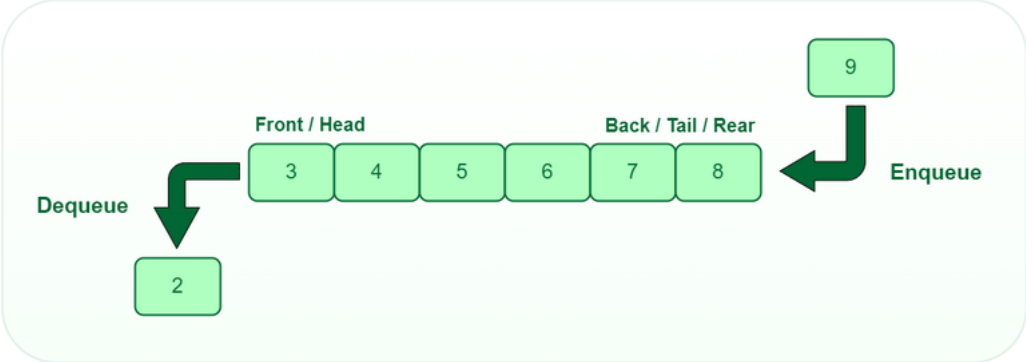


Figure 8: Enqueue and Dequeue

5 Conclusions

In this exploration of three basic data structures—Linked List, Stack, and Queue—we have delved into their fundamental principles, characteristics, and operations that define their usefulness and flexibility in computer science and software engineering.

Linked List: Linked lists offer dynamic memory allocation and flexible sizing, allowing adaptation to changing data structures easily. Their sequential access and node-based structure enable efficient insertion, deletion, and traversal, making them ideal for scenarios requiring frequent modifications in data structures.

Stack: Operating on the "Last In, First Out" (LIFO) principle, stacks are commonly used in tasks that require data processing in the reverse order of their addition. The simplicity of stack

structure and operations like Push, Pop, and Peek make them valuable tools in managing both simple and complex scenarios.

Queue: Following the "First In, First Out" (FIFO) principle, queues provide a way to process data in the order they were added. They are often used in tasks requiring sequential sorting and processing of data, such as job scheduling, resource management, and packet routing in networks.

Each data structure—Linked List, Stack, and Queue—brings its own advantages and trade-offs, serving diverse programming needs and specific situations. Deep understanding of these data structures empowers programmers to make informed design decisions and build robust, efficient solutions for various computational problems