Data structures and Algorithms

Note05-Stack, Queue and Deque

Edited By

Arshia Gharooni

2023

0.1 Stack

A stack is an abstract data type that represents a collection of elements, with two primary operations: push and pop. The push operation adds an element to the top of the stack, and the pop operation removes and returns the topmost element. Stacks are commonly used in computer science, particularly in algorithms and programming languages, for example, in parsing, recursion, and memory management.

0.1.1 Implementation of stack

A stack can be implemented using different data structures. The most common data structures used to implement a stack are an array and a linked list.

Array implementation

In an array implementation of a stack, we create an array of fixed size, and then we use two variables, namely top and size, to keep track of the elements in the stack. The top variable represents the index of the topmost element in the stack, and the size variable represents the number of elements in the stack.

When we push an element onto the stack, we increment the top variable and insert the element at the top index of the array. When we pop an element from the stack, we remove the element at the top index of the array and decrement the top variable.

The time complexity of push and pop operations in an array implementation of a stack is O(1), which means that the time it takes to perform these operations does not depend on the number of elements in the stack. However, the space complexity of an array implementation of a stack is O(n), where n is the size of the array. This means that we need to allocate a fixed amount of memory for the stack, regardless of the number of elements in the stack.

Let's consider a simple example of implementing a stack using an array. Suppose we have an array of size 10, and we want to create a stack of integers using this array. We can use the following steps to implement the stack:

1. Initialize the stack: We start by initializing an integer variable called top to -1. This variable will be used to keep track of the topmost element in the stack.

2. Push an element: To push an element onto the stack, we first check if the stack is full or not. If the stack is not full, we increment the top

variable and insert the new element at the top of the stack. If the stack is full, we display an error message saying that the stack is full.

3. Pop an element: To pop an element from the stack, we first check if the stack is empty or not. If the stack is not empty, we remove the topmost element from the stack and decrement the top variable. If the stack is empty, we display an error message saying that the stack is empty.

4. Peek the top element: To peek at the top element of the stack, we simply return the element at the index pointed to by the top variable.

Linked List Implementation

In a linked list implementation of a stack, we use a linked list data structure to represent the stack. The linked list consists of nodes, with each node containing an element and a pointer to the next node in the list.

To implement a stack using a linked list, we use a pointer variable, called top, to point to the topmost node in the linked list. When we push an element onto the stack, we create a new node and insert it at the beginning of the linked list, making it the new top node. When we pop an element from the stack, we remove the top node from the linked list and update the top variable to point to the next node in the list.

The time complexity of push and pop operations in a linked list implementation of a stack is also O(1), which means that the time it takes to perform these operations does not depend on the number of elements in the stack. However, the space complexity of a linked list implementation of a stack is O(n), where n is the number of elements in the stack. This means that we need to allocate memory dynamically as we add elements to the stack.

0.1.2 Stack Operations

Push

Push operation is used to insert an element at the top of the stack. In this operation, we first check if the stack is full or not. If the stack is not full, we increment the top index and insert the element at the top index. The time complexity of push operation is O(1) as it involves only one operation.

The push operation involves only one operation, so the time complexity of push operation is O(1).

Pop

Pop operation is used to remove an element from the top of the stack. In this operation, we first check if the stack is empty or not. If the stack is not empty, we remove the element at the top index and decrement the top index. The time complexity of pop operation is also O(1) as it involves only one operation.

The pop operation involves only one operation, so the time complexity of pop operation is also O(1)

Peek

Peek operation is used to return the element at the top of the stack without removing it. In this operation, we first check if the stack is empty or not. If the stack is not empty, we return the element at the top index. The time complexity of peek operation is also O(1) as it involves only one operation.

The peek operation involves only one operation, so the time complexity of peek operation is also O(1).

Size

Size operation is used to return the number of elements in the stack. In this operation, we simply return the value of the top index plus one. The time complexity of size operation is also O(1) as it involves only one operation.

The size operation involves only one operation, so the time complexity of size operation is also O(1).

0.2 Queue

In computer science, a queue is an abstract data type that represents a collection of elements, where the addition of new elements happens at one end, and the removal of elements happens at the other end. It follows the principle of first-in, first-out (FIFO), meaning that the element that has been in the queue for the longest time is the first one to be removed. Queues are commonly used in programming to manage resources or to simulate real-world scenarios such as a line at a supermarket.

In this lecture, we will discuss the various implementations of a queue data structure and the complexity of each implementation. We will also look at some common operations performed on queues and their corresponding time complexities.

0.2.1 Implementation of Queue

There are several ways to implement a queue data structure. The choice of implementation depends on the specific needs of the application and the constraints of the system. Here are some common implementations of a queue:

1. Array-based Queue: An array-based queue is implemented using an array data structure. The front and rear of the queue are represented by two indices, and new elements are added to the rear of the queue and removed from the front of the queue.

   The time complexity of inserting an element into the queue is O(1) since we can simply append the element to the end of the array. The time complexity of removing an element from the queue is also O(1) since we can simply remove the element at the front of the array. However, in the worst case scenario when the array becomes full, we need to resize the array which takes O(n) time where n is the size of the array.

2. Linked List-based Queue: A linked list-based queue is implemented using a linked list data structure. The front and rear of the queue are represented by two pointers, and new elements are added to the rear of the queue and removed from the front of the queue.

   The time complexity of inserting an element into the queue is O(1) since we can simply add a new node to the end of the linked list. The time complexity of removing an element from the queue is also O(1) since we can simply remove the first node from the linked list.

3. Circular Queue: A circular queue is implemented using an array data structure with a fixed size. The front and rear of the queue are represented by two indices that wrap around to the beginning of the array when they reach the end. New elements are added to the rear of the queue and removed from the front of the queue.

   The time complexity of inserting an element into the queue is O(1) since we can simply append the element to the end of the array. The time complexity of removing an element from the queue is also O(1) since we can simply remove the element at the front of the array. However, in the worst case scenario when the array becomes full, we cannot insert any more elements even if there are empty spaces in the array.

0.2.2 Queue Operations

A queue supports two basic operations, enqueue, and dequeue.

Enqueue

The enqueue operation adds an item to the end of the queue. The new item becomes the last item in the queue.

- Algorithm:

```
def  Enqueue(Queue, item)
    add item to the end of the Queue
```

- Time complexity:
  The time complexity of the enqueue operation is O(1) because it involves only one operation, which is adding an item to the end of the queue.

Dequeue

The dequeue operation removes the item at the front of the queue. The item is removed from the queue, and the next item in the queue becomes the front item.

- Algorithm:

```
def Dequeue(Queue)
    if Queue is not empty
        remove the first item from the Queue
```

- Time complexity:
  The time complexity of the dequeue operation is also O(1) because it involves only one operation, which is removing the first item from the queue.

Additional Queue Operations

In addition to the basic operations, a queue can also support other operations, such as checking if the queue is empty and finding the size of the queue.

1. IsEmpty:
   The IsEmpty operation checks if the queue is empty. If the queue is empty, it returns true; otherwise, it returns false.

   - Algorithm:

```
def IsEmpty(Queue)
    if Queue is empty
        return true
    else
        return false
```

- Time complexity:
  The time complexity of the IsEmpty operation is O(1) because it involves only one operation, which is checking if the queue is empty.

2. Size:
   The Size operation returns the number of items in the queue.

   - Algorithm:

```
def Size(Queue)
    return the number of items in the Queue
```

   - Time complexity:
     The time complexity of the Size operation is O(1) because it involves only one operation, which is returning the number of items in the queue.

0.3 Deque

A deque, short for double-ended queue, is a data structure that allows insertion and removal of elements from both ends. In other words, it is a linear collection of elements that supports adding and removing elements from both the front and the back of the queue. The deque data structure is useful in many situations, including implementing algorithms for breadth-first search, simulation of queueing systems, and for implementing data structures such as stacks and queues.

0.3.1 Deque Implementations

There are several ways to implement a deque data structure, each with its own advantages and disadvantages. Here are some of the most common implementations:

Array-based Deque

An Array-based Deque is a data structure that is used to store a collection of elements in a particular order. The term Deque is an abbreviation for Double Ended Queue, and it refers to a data structure that allows insertion and deletion of elements from both ends of the queue. In this lecture note, we will discuss the implementation and usage of Array-based Deque.

- Implementation of Array-based Deque
  Array-based Deque is implemented using an array data structure. In this implementation, the array is used to store the elements of the Deque. The two ends of the Deque are represented by two indices, front and rear, which point to the first and last elements of the Deque, respectively. Initially, both front and rear are set to -1, which indicates that the Deque is empty.

- Operations on Array-based Deque
  The following operations can be performed on an Array-based Deque.

  1. Insertion:
     Elements can be inserted into an Array-based Deque from either end. When an element is inserted into the Deque, the front and rear indices are updated accordingly. If the Deque is empty, then both front and rear are set to 0. If the Deque is full, then no more elements can be inserted.
  2. Deletion:
     Elements can be deleted from an Array-based Deque from either end. When an element is deleted from the Deque, the front and rear indices are updated accordingly. If the Deque becomes empty after deletion, then both front and rear are set to -1.
  3. Peek:
     The Peek operation is used to retrieve the element at either end of the Deque without removing it. This operation does not modify the Deque in any way.
  4. Size:
     The Size operation is used to determine the number of elements in the Deque.

Linked-list-based Deque

A Linked-list-based Deque is a Deque implementation using a linked list as the underlying data structure. A Linked-list-based Deque has two pointers: head and tail. The head pointer points to the first node of the list, while the tail pointer points to the last node of the list.

- Insertion
  Insertion at the head or the tail of the deque can be done by simply adding a new node to the linked list and changing the pointers accordingly. To insert at the head, we create a new node and make it the new head by changing the next pointer of the new node to the current head and updating the head pointer to point to the new node. To insert at the tail, we create a new node and make it the new tail by changing the

next pointer of the current tail to the new node and updating the tail pointer to point to the new node. In both cases, the time complexity is O(1).

- Removal
  Removal from the head or the tail of the deque can be done by simply removing the corresponding node from the linked list and changing the pointers accordingly. To remove from the head, we remove the current head node and update the head pointer to point to the next node. To remove from the tail, we remove the current tail node and update the tail pointer to point to the previous node. In both cases, the time complexity is O(1).