Chapter 15 Stack and Queue

Contents

Syllabus Learning Outcomes

1.3 Data Structures

Understand concept and write algorithms for stack, queue (linear and circular), linear linked list and binary search tree.

1.3.3 Create, insert, and delete operations for stack and queue (linear and circular).

2.3 Implementing Algorithms and Data Structures

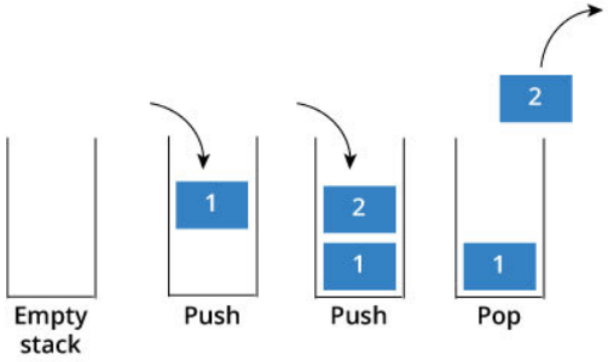
Use programming language elements and constructs to implement sort and search algorithms such as insertion sort, bubble sort, quicksort, merge sort, linear search, binary search and hash table search, as well as data structures such as stacks, queues, linear linked lists and binary search trees.

2.3.3 Write programs to implement operations for stacks, queues (linear and circular), linear linked lists and binary search trees.

1. Stack

Stack is an abstract data type (ADT) which inserts and removes items according to the **Last-In-First-Out** (**LIFO**) principle. A user may insert items into a stack at any time, but may only access or remove the last item inserted.

The name “stack” is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the “pushing” and “popping” of plates on the stack. When we need a new plate from the dispenser, we “pop” the top plate off the stack, and when we add a plate, we “push” it down on the stack to become the new top plate. The same terms are also used in the stack ADT.



Stacks are used in many applications such as the Internet Web browsers storing the addresses of recently visited sites, reversing data as well as matching opening and closing symbols.

1.1 Stack implementation

Stack can be implemented using either array or linked list.

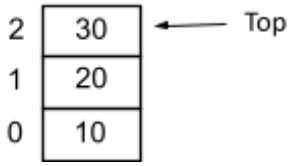
The basic operations of a stack is to

- push(): add an item to the top of the stack
- pop(): remove and return an item from the top of the stack

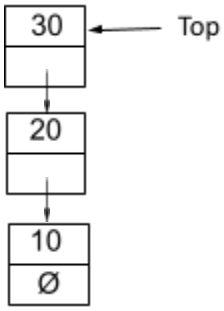
Other supporting operations to be added are

- is_empty(): check if stack is empty
- size(): return number of items in the stack
- peek(): return the top item without removing it
- display(): show all items in the stack

Implementation using array



Implementation using linked list



2. Queue

Queue is another ADT which is a collection of items that are inserted and removed according to the **First-In-First-Out (FIFO)** principle. That is, items can be inserted at any time, but only the item that has been in the queue the longest can be next removed.

We usually say that items enter a queue at the rear and are removed from the front. A metaphor for this terminology is a line of people waiting to get on an amusement park ride. People waiting for such a ride enter at the back of the line and get on the ride from the front of the line. There are many other applications of queues. Stores, theatres, reservation centres, and other similar services typically process customer requests according to the **FIFO** principle. A queue would therefore be a logical choice for a data structure to handle calls to a customer service centre, or a wait-list at a restaurant. Many computing devices, such as a networked printer, or a Web server responding to requests, also use queues.

2.1 Linear queue implementation

Queue can be implemented using either array or linked list.

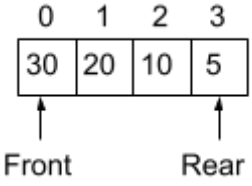
The basic operations of a queue is to

- enqueue(): add an item to the rear of the queue
- dequeue(): remove an item from the front of the queue

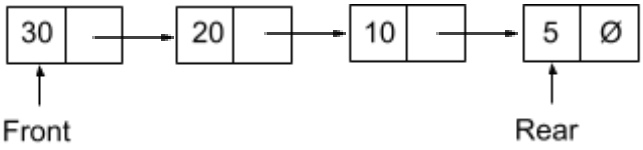
Other supporting operations to be added are

- is_empty(): check if queue is empty
- size(): return number of items in the queue
- peek(): return the next item to be removed without removing it
- display(): show all items in the queue

Implementation using array



Implementation using linked list



2.2 Circular queue implementation

For the stack ADT, we can use a simple mapping to Python list methods. It may be very tempting to use a similar approach for supporting the queue ADT. We could enqueue element e by calling append(e) to add it to the end of the list. We could use the syntax pop(0), as opposed to pop( ), to intentionally remove the first element from the list when dequeuing.

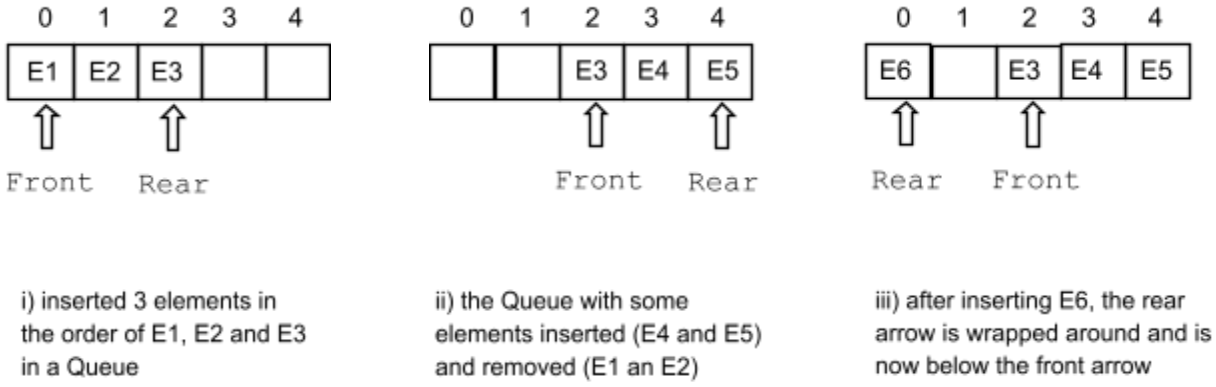
As easy as this would be to implement, it is tragically inefficient. When pop is called on a list with index 0, a loop is executed to shift all elements beyond the specified index, 0 in this case, to the left, to fill the hole in the sequence caused by the pop to make the front pointer always point to the first index in the list. Therefore, a call to pop(0) always causes the worst-case behaviour of O(n) time.

We can improve on the above strategy by avoiding the call to pop(0) entirely. We can replace the dequeued entry in the array with a reference to None, and maintain an explicit variable `Front` to store the index of the element that is currently at the front of the queue. Such an algorithm for dequeue would run in O(1) time. After several dequeue operations, this approach might lead to the configuration portrayed in the diagram below. The `Front` of the queue drifts away from index 0.

We can build a queue that has relatively few elements, yet which are stored in an arbitrarily large list. This occurs, for example, if we repeatedly enqueue a new element and then dequeue another (allowing the front to drift rightward). Over time, the size of the underlying list would grow to O(m) where m is the total number of enqueue operations since the creation of the queue, rather than the current number of elements in the queue.

A method to overcome the problem is to implement the queue as a circular queue, a queue with wrap-around. It can be implemented with a pointer, `Front`, pointing to the front of the queue and a pointer, `Rear`, pointing to the back of the queue. A variable `MaxQueueSize`, holding the maximum capacity of the queue, is needed, and it is useful to have an additional variable `NumberInQueue`, giving the number of elements already in the queue.

Illustration of a circular queue:



i) inserted 3 elements in the order of E1, E2 and E3 in a Queue

ii) the Queue with some elements inserted (E4 and E5) and removed (E1 an E2)

iii) after inserting E6, the rear arrow is wrapped around and is now below the front arrow

Take note that both stacks and queues can be implemented using either array or linked list, whereby both Python list and linked list allow for dynamic memory allocation of its data elements. However, we can also implement stacks and queues having a fixed size, such that we can implement a fixed size array, which allows the static memory allocation of its data elements. In the case of the circular queue implementation, it follows the static memory allocation of its data elements.