1.1 Lists as Queues and Stacks

Lists as Queues and Stacks

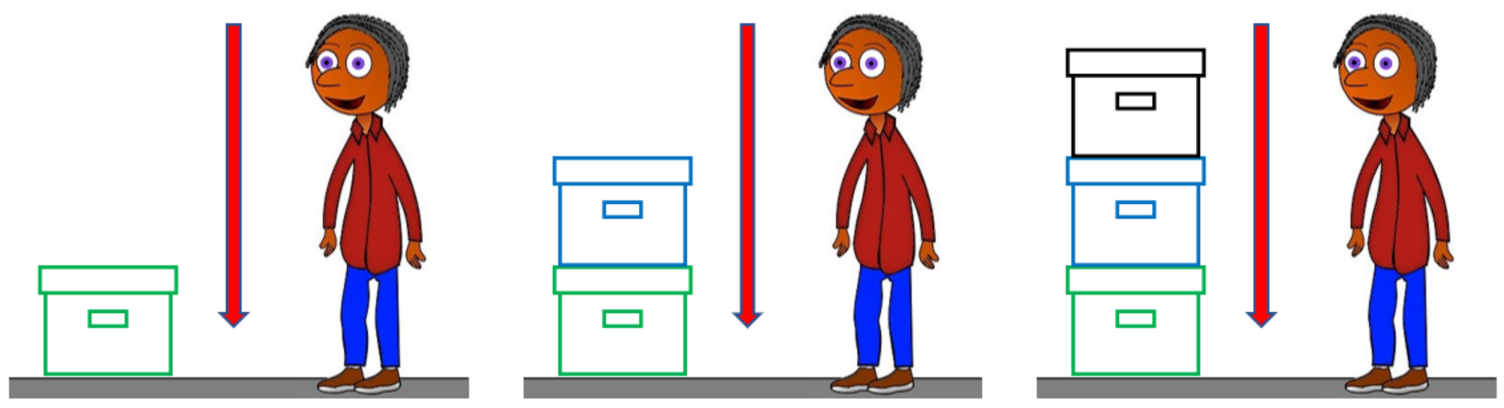
1. Introduction to Queues and Stacks

Queues and stacks are widely used elementary data structures. We call them elementary because they are straightforward and there is nothing complicated about them. One important property of queues and stacks is that their insert and delete operations are prespecified. This means you cannot change how insert and delete operations are carried out.

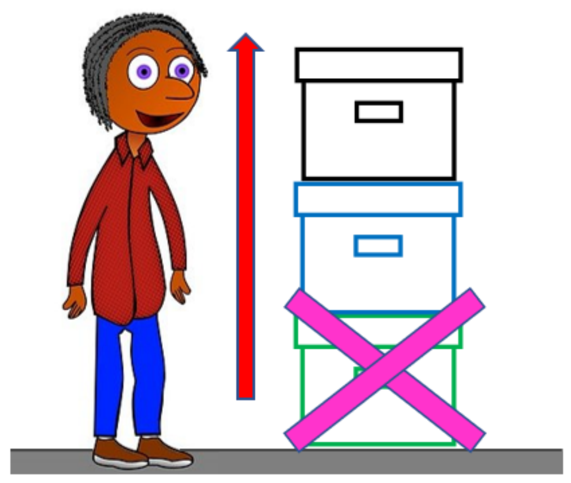
In the first course Python for Beginners, we learned about lists in detail. In this lesson, we will learn how to use lists as stacks and queues.

2. Stacks

Consider a scenario where you have some boxes and you have to place them on top of each other. You get the first box and place it on the floor. You get the second box and place it on top of the first one. Finally, you get the third box and place it on top of the second one. Now we have a stack of three boxes.



Then a friend of yours comes and asks you to hand over the boxes to him/her. How do you remove the boxes? The first box you remove is the third one, which you have placed last. Then the second box. You cannot remove the first box at the bottom without removing the boxes on top.



Placing the boxes on top of each other is similar to insert operation in stacks and removing the boxes is similar to delete operation in stacks.

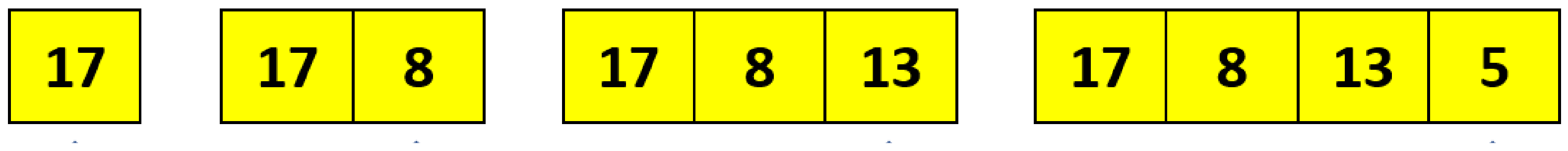
2.1 Properties of a Stack

Stacks have three main properties.

- Stacks are linear structures without any branches.
- The element that is inserted last is deleted first. Last in first out.
- Insertion and deletion cannot be carried out at random locations. They can be carried out only at the end of the stack.

2.2 A Sample Stack

Let us look at an actual stack containing some values. We are going to insert the following four numbers, 17, 8, 13, and 5, into a stack.

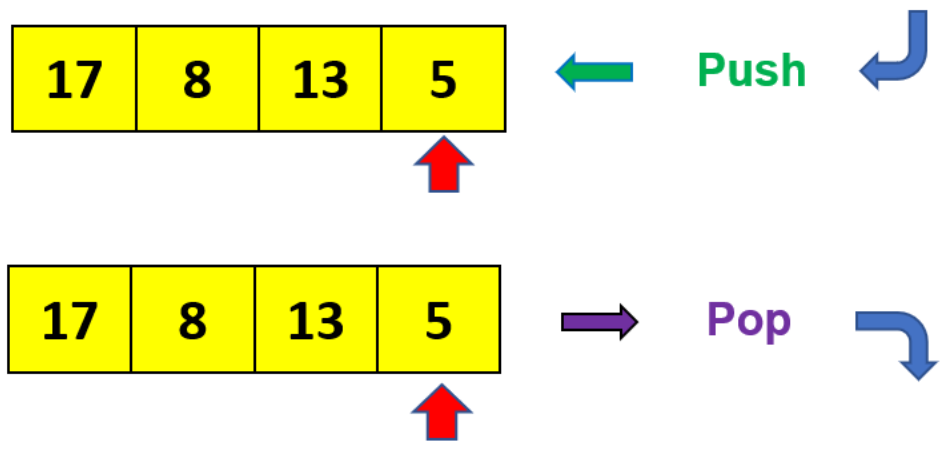


First, we insert 17. In stacks, we need to keep track of the last element inserted because when we delete, we delete the last element inserted and when we insert, we insert to the end of the stack. Now that we know why we need to keep track of the last element, the next question we have to ask is 'how do we achieve it?'.

It's simple. We keep a pointer pointing to the last element. In this example, the pointer named 'top' is performing that task.

Next, we insert 8 and the pointer 'top' is shifted from 17 to 8. Then we insert 13. And finally, we insert 5.

NOTE: In the context of stacks, we use the term PUSH for insert and POP for delete as shown in the following diagram.



2.3 Stack Implementation in Python

To implement a stack using the list data structure, there are three requirements.

- A list
- A pointer to the last element
- Push and Pop functions

```
>>> def push(stack, value):
            stack.append(value)

>>> def pop(stack):
            return stack.pop()

>>> my_stack = []
>>> push(my_stack, 'a')
>>> push(my_stack, 'b')
>>> push(my_stack, 'c')
>>> my_stack
['a', 'b', 'c']
>>> pop(my_stack)
'c'
>>> pop(my_stack)
'b'
>>> pop(my_stack)
'a'
```

Let us start with the implementation of Push and Pop functions.

The PUSH function should be able to insert a given element to the end of the stack. In the implementation given above, the function takes a stack and the new value to be inserted as the inputs. The append method available in lists is used to insert the value at the end of the stack.

The POP function should remove the last element inserted. In the implementation given above, the function takes a stack as input. The pop method, which is a built-in method available to lists, is used to remove the last element of the list. The removed value is returned by the function. Since the list data structure has a built-in method to remove the last element, we do not need a pointer to keep track of it.

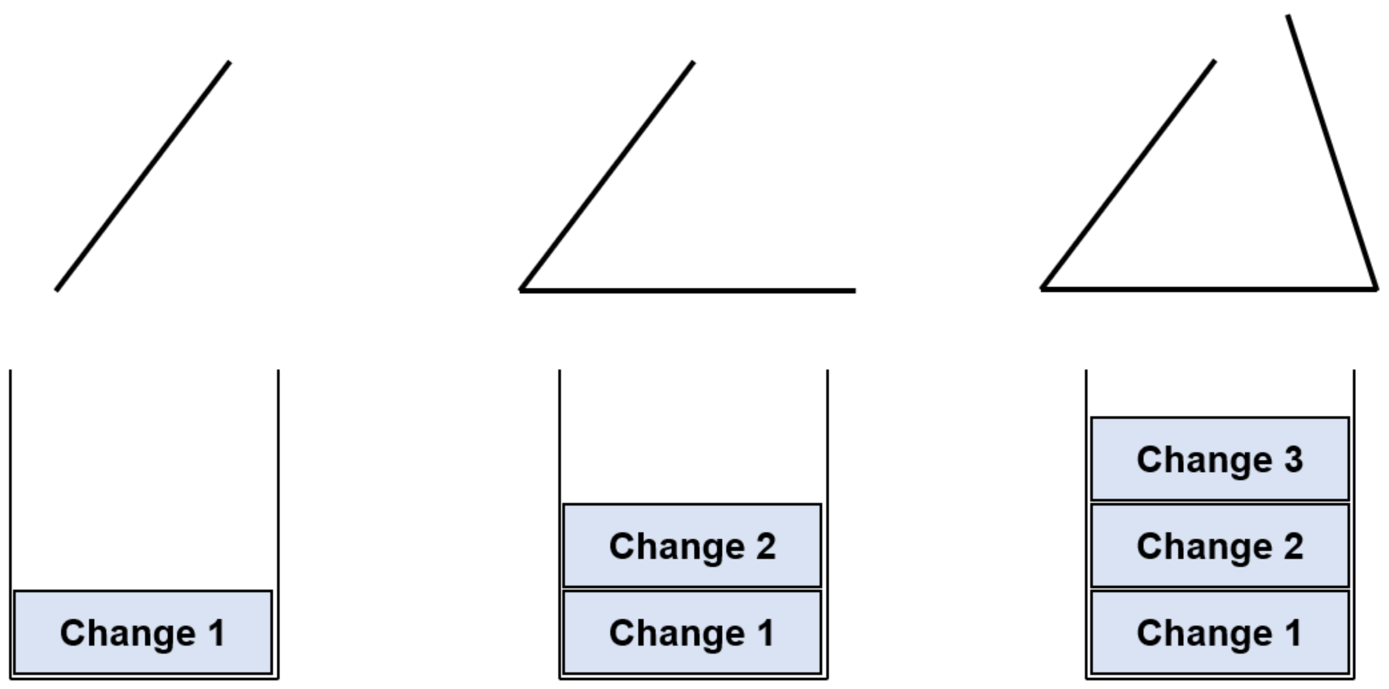
Then create an empty list that is going to be our stack. In the sample code given above, the list is named 'my_stack'.

Then create an empty list that is going to be our stack. In the sample code given above, the list is named 'my_stack'.

To insert ‘a’, we call the push function with ‘my_stack’ and ‘a’ as input parameters. It will insert ‘a’ at index 0 of the list. Next, we call the push function with ‘b’ as an input parameter. ‘b’ will be inserted at index 1 of the list. Similarly, we insert ‘c’ at index 2 of the list. To confirm our insertions, we can simply print the stack. You can see that the values have been inserted correctly.

To delete a value from our stack, we call the pop function we implemented with ‘my_stack’ as the input parameter. As shown here, it returns ‘c’, which is the last element we inserted. When we call pop again, it returns ‘b’ at index 1. And finally, the pop function will return ‘a’ at index 0, leaving us with an empty stack.

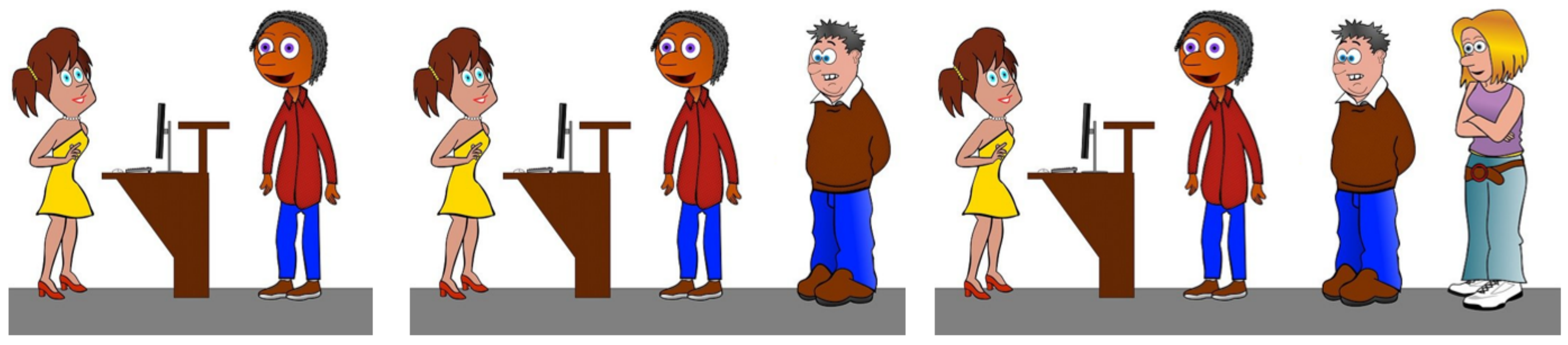
2.4 Application of Stacks



One of the simplest examples is the undo function. Suppose, you want to draw a triangle. You draw the first line, which is the first change you make. Then you draw the second line which is the second change. And finally, when you try to draw the third and last line, you make a mistake (see the diagram above). You can correct it by using the undo function which will remove the last change you made. It is similar to pop operation in a stack. Now, you can redraw the last line correctly. This example clearly illustrates how useful stacks are.

3. Queues

Assume a scenario at a bank where people are coming to deposit money. People form a QUEUE in the order in which they come in. The person who comes first stays at the front of the queue. The person who comes in second stays behind the first person. Similarly, the person who comes in third stays behind the second person. Now we have a queue with three people.



I believe you all are familiar with how queues work in real life. The first person in the queue, who entered the queue first, deposits the money and leaves. Then, the remaining people in the queue move one position forward. Now, the person who entered the queue second deposits the money and leaves. This pattern continues until the last person. The queue data structure also works in a similar fashion.

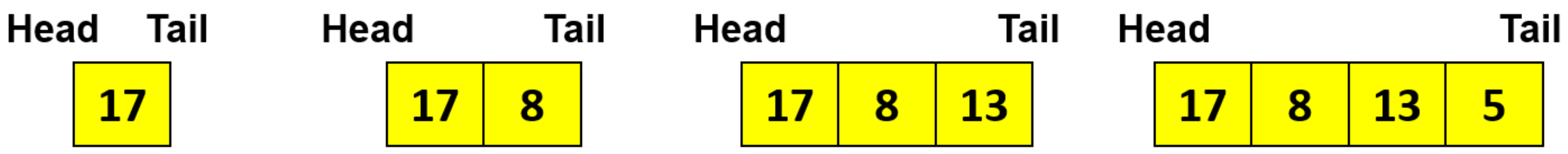
3.1 Properties of a Queue

Queues have three main properties.

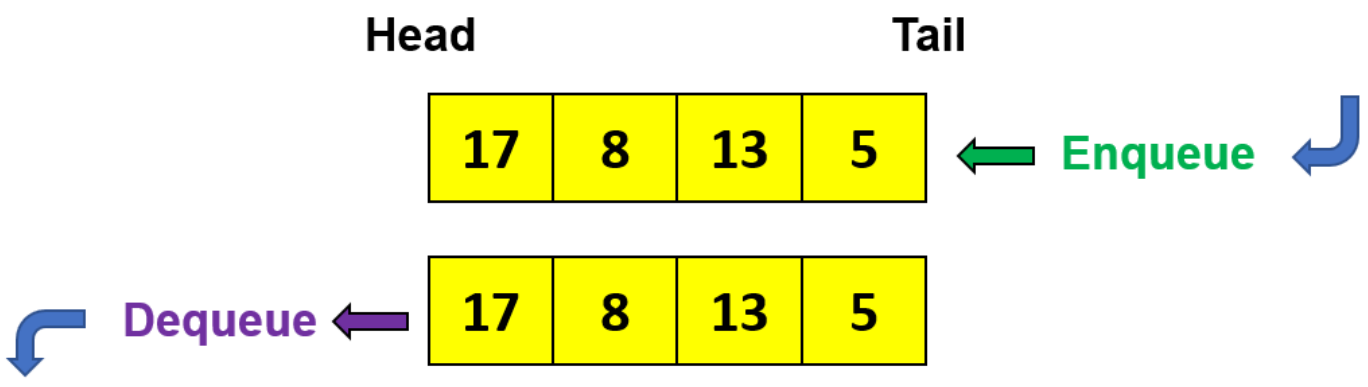
- Queues are linear. There are no branches.
- The element that is inserted first is removed first. First-in first-out.
- Insertion and deletion cannot be carried out at random locations. Insertion can only be carried out at the end of the queue and deletion only happens at the front of the queue.

3.2 A Sample Queue

Let us build a sample queue. We are going to insert the numbers 17, 8, 13, and 5, into a queue.



First, we insert 17. As shown in the diagram above, a queue has a head and a tail. Insertion takes place at the tail of the queue and deletion happens at the head of the queue. Next, we insert 8, 13, and finally 5.



In the context of queues, we use the term ENQUEUE for insert and DEQUEUE for delete. Enqueue operation adds an element to the tail of the queue. Dequeue operation removes an element from the head of the queue.

3.3 Queue Implementation in Python

To implement a queue using the list data structure, there are three requirements.

- A list
- Pointers to the head and tail
- Enqueue and Dequeue functions

```
>>> def enqueue(queue, value):
      queue.append(value)

>>> def dequeue(queue):
      return queue.pop(0)

>>> my_queue = []
>>> enqueue(my_queue, ‘a')
>>> enqueue(my_queue, ‘b')
>>> enqueue(my_queue, ‘c')
>>> my_queue
[‘a’, ‘b’, ‘c’]
>>> dequeue(my_queue)
‘a’
>>> dequeue(my_queue)
‘b’
```

Let us start with the implementation of Enqueue and Dequeue functions. The enqueue function should append the given value to the tail of the queue. In out implementation here, the enqueue function takes the queue and the value to be inserted as input parameters. The append method available for list data structure is used to insert the value at the tail of the queue. The dequeue function should remove the first element in the queue. In this implementation, the function takes the queue as input parameter. The pop built-in method available for list data structure is used to remove the first element of the list. To remove the first element, the pop method should be called with parameter 0 to specify index 0. The removed value is returned by the function. Since Python list has built-in methods to remove from the head and append to the tail, we do not need separate pointers to the head and tail of the queue.

Then create an empty list that is going to be our queue. In the sample code given above, the list is named ‘my_queue’.

To insert ‘a’, we call the enqueue function with ‘my_queue’ and ‘a’ as input parameters. It will insert ‘a’ at index 0 of the list. Next, we call the enqueue function with ‘b’ as an input parameter. ‘b’ will be inserted at index 1 of the list. Similarly, we insert ‘c’ at index 2 of the list.

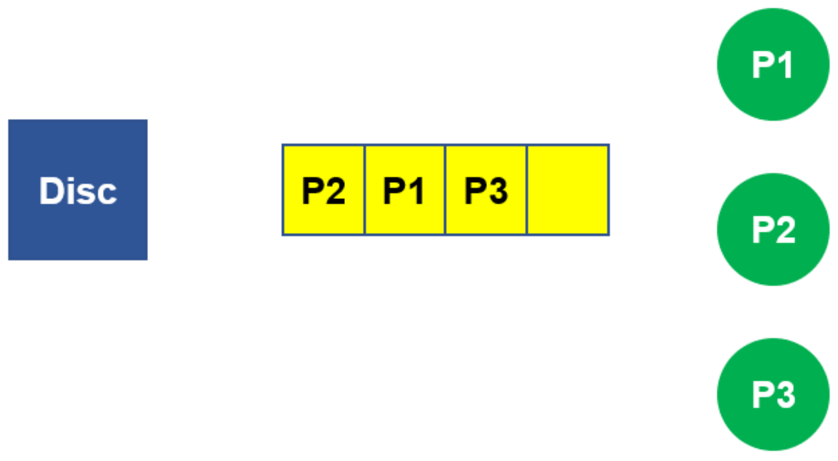
To confirm our insertions, we can simply print the queue. You can see in the example that the values have been inserted correctly.

To delete a value from our queue, we call the dequeue function we implemented with ‘my_queue’ as the input parameter. As shown here, it returns ‘a’, which is the first element we inserted.

When we call dequeue again, it returns ‘b’ which is the next element we inserted.

3.4 Application of Queues

File reading in computers is one occasion where queues are useful.



Consider a scenario where we have several files stored in a disc. There are three programs running in parallel and they need to access multiple files. However, our hard disc can read only one file at a time. Hence, what it does is, it keeps a queue where all the incoming requests are stored. In this example, program P2 has made the first request to read a file. P1 has made the second request and P3 has made the third one. Using a queue ensures that the program which makes the request first will receive the file first, which ensures fairness. This example clearly illustrates how useful queues are.

Data retention summary

mora_logo.png Sponsored By logo.png