What is a Stack?

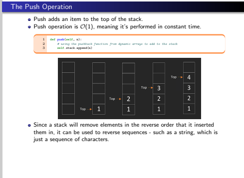
- A stack is a linear data structure that stores items in a Last-In/First-Out (LIFO) manner.
- Think of a stack of plates; you can only add or remove the top plate.
- The last element added is the first one to be removed.

A stack is a data structure that contains a collection of elements where you can add and delete elements from just one end (called the top of the stack). In the physical world, a stack can be conceptualized by thinking of plates at a dinner party buffet. When you go to take a plate, you can only remove from the top and similarly when you finish your meal, the stack of plates can only be built by adding them on top of each other — this is exactly what a stack in the software world does.

Key Characteristics of Stacks

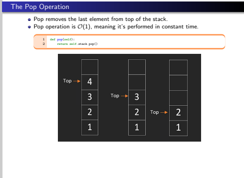
- Stacks can grow and shrink dynamically as items are pushed and popped.
- Primary operations include push, pop, and peek.
- Used in function calls, undo mechanisms in software, and more.

Stacks are a dynamic data structure that operate on a LIFO (Last In First Out) manner. The last element placed inside is the first element that comes out. The stack supports three operations - push, pop, peek.



Push operation adds an element to the top of the stack, which in dynamic array terms would be appending an element to the end. This is an efficient $\mathcal{O}(1)$ operation as discussed in the previous chapters. It helps to visualize a stack as an array that is vertical. The pseudocode demonstrates the concept, along with the visual where we add numbers from 1 to 4 to the top. The top pointer updates to point at the last item added.

Stack, as a data structure, is just an abstract interface and it does not really matter how you implement it - the characteristics are just that you should be able to add and remove elements from the same end.



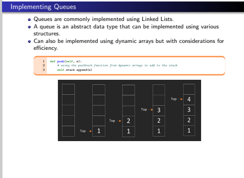
Pop operation removes the last element from top of the stack, which in dynamic array terms would be retrieving the last element. This is also an efficient $\mathcal{O}(1)$ operation as discussed in the previous chapters. Taking the previous example, let's say we wish to pop 4 and 5. The pseudocode demonstrates the concept, along with the visual where we remove 4 and 5 from the top. Again, the top pointer updates to point at the last item.

2024-01-24

What is a Queue?

- A queue is a linear data structure that serves elements in a First-In/First-Out (FIFO) manner.
- Similar to people queuing at a bank or grocery store.
- The first element added to the queue will be the first one to be removed.
- Queues can dynamically grow or shrink as elements are enqueued and dequeued.
- The main operations are enqueue and dequeue.
- Used in scenarios like printer job management.

Queues are similar to stacks, except they follow what is called a FIFO approach (First in First Out). A real world example would be a line at the bank. The first person to come in the line is the first person to be served. An example from the software world would be print jobs. For example, if multiple people are trying to print documents, it will be handled on a first come first serve basis.



The most common implementation of a queue is using a Linked List.
The two operations that queues support are enqueue and dequeue.

A queue is just an abstract interface, similar to a stack and can be implemented by multiple data structures, provided that they fulfill the contract of implementing enqueue and dequeue operations.

The Enqueue Operation

- Enqueue adds an element to the tail of the queue.
- This operation is $\mathcal{O}(1)$ as it involves no shifting of elements.

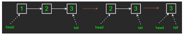
```
def enqueue(self, val):
    newNode = ListNode(val)

    # Queue is non-empty
    if self.right:
        self.right.next = newNode
        self.right = self.right.next
    # Queue is empty
    else:
        self.left = self.right = newNode
```

The enqueue operation adds elements to the tail of the queue until the size of the queue is reached. Since adding to the end of the queue requires no shifting of the elements, this operation runs in $\mathcal{O}(1)$. The pseudocode and visual demonstrates this.

The Dequeue Operation

- Dequeue removes and returns the front element of the queue.
- Always check if the queue is empty before dequeuing.

```python
def dequeue(self):
    # Queue is empty
    if not self.left:
        return None

    # Remove left node and return value
    val = self.left.val
    self.left = self.left.next
    if not self.left:
        self.right = None
    return val
```

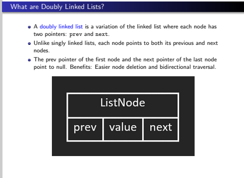


Operation	Big-O Time Complexity
Enqueue	$\mathcal{O}(1)$
Dequeue	$\mathcal{O}(1)$

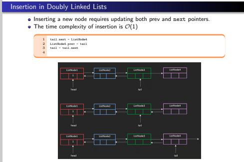
Table: Queue Operations-Worst Case $\mathcal{O}$

Queues could also be implemented by using dynamic arrays, however, it gets a little bit trickier if you want to maintain efficiency of enqueue and dequeue operations. With the array implementation, dequeue would take $\mathcal{O}(n)$ time due to shifting of the elements.

Similar to stacks, it is a good measure to check if the queue is empty before performing the dequeue operation.

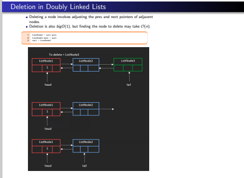


Having learned about singly linked lists, let's next learn about its variation - the Doubly Linked List. As the name implies, it's called doubly because each node now has two pointers. We have a prev pointer which points to the previous node, in addition to the next pointer. If the prev pointer points to null, it is an indication that we are at the start of the linked list.



Similar to the singly linked list, adding a node to a doubly linked list will run in $\mathcal{O}(1)$ time. Only this time, we have to update the prev pointer as well.

For example, looking at the visual below, we have three nodes in our linked list, ListNode1, ListNode2 and ListNode3. Now we have another node, ListNode4, that we wish to insert. We know the we will have to update the next pointer of ListNode3 and the prev pointer of ListNode4. The pseudocode demonstrates this along with the step by step visual.



Going back to the example with the three nodes, deleting is also a $bigO(1)$ operation. There is no shifting or traversal required. Instead, in this case adjusting the prev pointer is required. The pseudocode and visual demonstrate this. Appending and removing from the end of linked lists are both $\mathcal{O}(1)$ operations which is similar to the push and pop operations of the stack. As mentioned earlier, a stack is just an abstract interface that can also be implemented using linked lists. If the target node is not the head or the tail, you must arrive at the node before deletion, which is $\mathcal{O}(n)$.

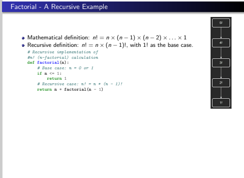
What is Recursion?

- Recursion occurs when a function calls itself.
- A recursive function has a base case and a recursive step.
- Think of recursion like breaking down a problem into smaller, more manageable parts.

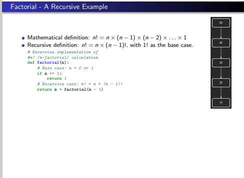
Recursion can be a perplexing concept to wrap your head around so don't be discouraged if you don't get it straightaway. Recursion is when a function calls itself with a smaller output. So while an iterative function will make use of for loop and while loop, a recursive function achieves this by calling itself until a base case is reached. Recursive functions have two parts:

- The base case
- The function calling itself with a different input.

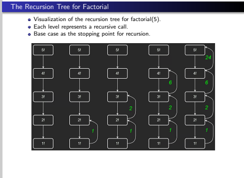
There are two types of recursion, one-branch and two-branch. Let's discuss one-branch recursion first.



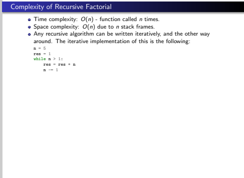
In the last line return statement, we notice that the function is calling itself with a different input. Let's analyze this. We have our two parts: base case and the function calling itself. When the code reaches the last line with the initial input of 5, we get: 5 * factorial(4), which starts executing the function again from line 1, only now with input 4, so we get 4 * factorial(3) and then 3 * factorial(2) and lastly 2 * factorial(1) after which the base case is reached.



But what happens when the base case is reached? When the function is called with 1 as the input, 1 is returned, and now it can be multiplied by 2, which will result in 2, which is the answer to 2!. We have only solved the first sub-problem so far. Now, we compute 3 * factorial(2), which results in 6, then 4 * factorial(3), which is 24, and finally 5 * factorial(4), which is 120 - the ultimate answer to 5! The most important part is that when we trigger the base case, we move back "up" the recursion tree because now we have to "piece" together the answers to our sub-problems to get to the final solution.



As observed, we took the original problem, factorial(5) and broke it down into smaller sub-problems, and by combining the answer to those sub-problems, we were able to solve the original problem. It is important to note that if there is no base case in recursion, the last line would execute forever resulting in a stack overflow!



As observed, we took the original problem, factorial(5) and broke it down into smaller sub-problems, and by combining the answer to those sub-problems, we were able to solve the original problem. It is important to note that if there is no base case in recursion, the last line would execute forever resulting in a stack overflow!

What is Two-Branch Recursion?

- Two-branch recursion occurs when a function makes two recursive calls.
- It's often used to solve problems where a current state depends on two previous states.
- A classic example: the Fibonacci sequence.
- To find the n^{th} Fibonacci number, we sum the $(n-1)^{th}$ and $(n-2)^{th}$ numbers.
- The sequence is defined as:

$$\begin{aligned} F(0) &= 0 \quad \text{(base case)} \\ F(1) &= 1 \quad \text{(base case)} \\ F(n) &= F(n-1) + F(n-2) \quad \text{for } n > 1 \end{aligned}$$

Two-branch recursion is a fascinating recursion case.A classic example: the Fibonacci sequence. To find the n^{th} Fibonacci number, we sum the $(n-1)^{th}$ and $(n-2)^{th}$ numbers. The sequence is defined as:

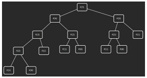
$$\begin{aligned} F(0) &= 0 \quad \text{(base case)} \\ F(1) &= 1 \quad \text{(base case)} \\ F(n) &= F(n-1) + F(n-2) \quad \text{for } n > 1 \end{aligned}$$

This is a classic example of a recurrence relation.

Pseudocode for Recursive Fibonacci

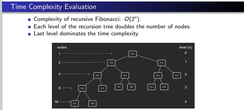
```
# Recursive implementation to calculate the n-th Fibonacci number
def fibonacci(n):
    # Base case: n = 0 or 1
    if n <= 1:
        return n

    # Recursive case: fib(n) = fib(n - 1) + fib(n - 2)
    return fibonacci(n - 1) + fibonacci(n - 2)
```

- Base cases: Return 0 for $n = 0$ and 1 for $n = 1$.
- Recursive case: Return $\text{fib}(n-1) + \text{fib}(n-2)$.



The above pseudocode is similar to our previous example with factorial, except this is a branch factor of two. Notice how we are calling the function twice in the last line, this results in the tree that is shown in the visual.

To analyze, let's follow the same technique we introduced in the previous chapter. We have our base case, we know the function calls itself in the last return statement, and we know that at some point when the base case is reached, we will have to travel back "up" to calculate the ultimate answer. To calculate `fibonacci(5)`, we get `fibonacci(4)` + `fibonacci(3)`. Now, both of these will execute the function from line 1. Looking at our tree, `fibonacci(4)` will call `fibonacci(3)` + `fibonacci(2)` and so on, until n hits 1 or 0 after which it will return the result, and keep going back up all the way until `fibonacci(4)` which will give us an answer of 3. Now, we have the answer to `fibonacci(4)` and do the same for `fibonacci(3)` which results in 2. Add the two together, and the 5^{th} Fibonacci number is 5.



Evaluating the time complexity for this is a little bit more tricky. Let's analyze the tree, and the number of nodes on each one of the levels. On the 1st level (0 indexed), there is 1, on the 2nd level, there are 2, then 4, after which we can see a pattern. Each level has the potential to hold $2\times$ the nodes of the previous level.

That only gives us half the answer. If n is the level we are currently on, this means that to get the number of nodes at any level n, the formula is 2^n. Since we have to potentially traverse all the way to the n^{th} level, and each level has twice as many nodes, we can say the function is upper bounded by 2^n. Recall the power series concept discussed in the dynamic array chapter where the last term is the dominating term. Notice how on the last level (4) there can be at most 16 nodes. Since the last level is in $O(2^n)$, it must be the case that the entire tree is in $O(2^n)$.

Algorithmically speaking, even if we did have 2×2^n or 2^{n-1} operations, it would still belong to $O(2^n)$ because constants do not affect the bound.