Stacks, Queues, Recursion

Presented by Yasin Ceran

January 24, 2024

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.

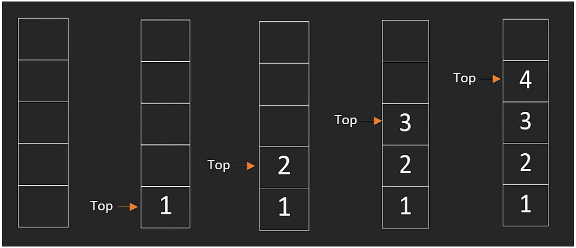
Key Characteristics of Stacks

- 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.

The Push Operation

- Push adds an item to the top of the stack.
- Push operation is $\mathcal{O}(1)$, meaning it's performed in constant time.

```
def push(self, n):
    # using the pushback function from dynamic arrays to add to the stack
    self.stack.append(n)
```

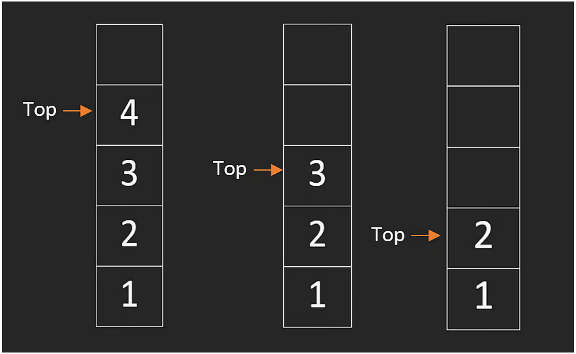


- Since a stack will remove elements in the reverse order that it inserted them in, it can be used to reverse sequences - such as a string, which is just a sequence of characters.

The Pop Operation

- Pop removes the last element from top of the stack.
- Pop operation is $\mathcal{O}(1)$, meaning it's performed in constant time.

```
def pop(self):
    return self.stack.pop()
```



Peek

- Peek is the simplest of three. It just returns, without removing, the top most element.

```python
def peek(self):
    return self.stack[-1]
```

Operation	Big-O Time Complexity
Push	$\mathcal{O}(1)$
Pop	$\mathcal{O}(1)$
Peek/Top	$\mathcal{O}(1)$

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

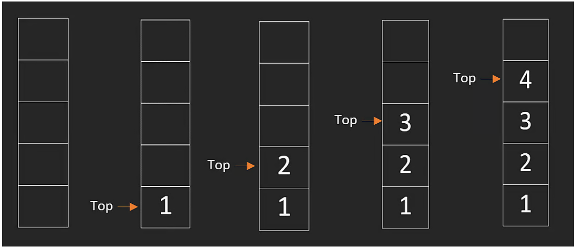
What is a Queue?

- 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.

Implementing Queues

- Queues are commonly implemented using Linked Lists.
- A queue is an abstract data type that can be implemented using various structures.
- Can also be implemented using dynamic arrays but with considerations for efficiency.

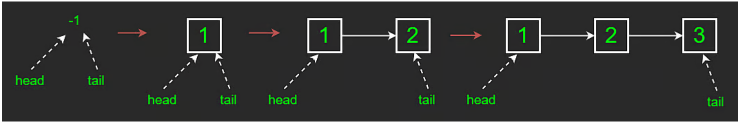
```
def push(self, n):
    # using the pushback function from dynamic arrays to add to the stack
    self.stack.append(n)
```



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.

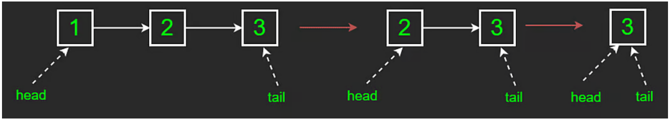
```python
def enqueue(self, val):
    newNode = ListNode(val)

    # 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 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
```

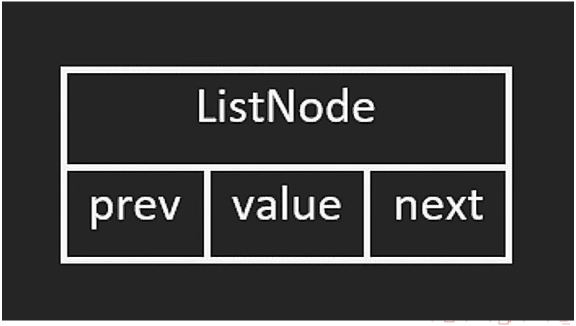


Operation	Big-O Time Complexity
Enqueue	$\mathcal{O}(1)$
Dequeue	$\mathcal{O}(1)$

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

What are Doubly Linked Lists?

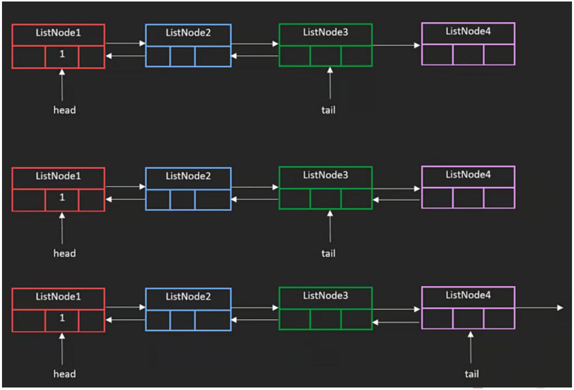
- A doubly linked list is a variation of the linked list where each node has two pointers: `prev` and `next`.
- Unlike singly linked lists, each node points to both its previous and next nodes.
- The prev pointer of the first node and the next pointer of the last node point to null. Benefits: Easier node deletion and bidirectional traversal.



Insertion in Doubly Linked Lists

- Inserting a new node requires updating both prev and `next` pointers.
- The time complexity of insertion is $\mathcal{O}(1)$

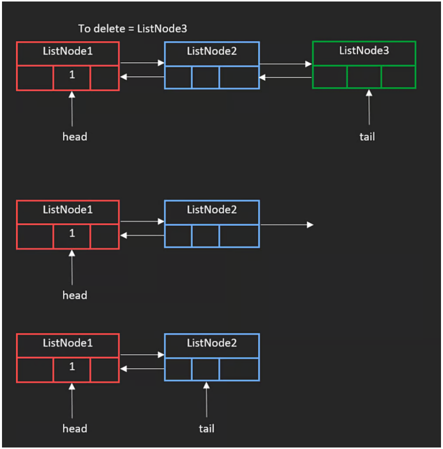
```
tail.next = ListNode4
ListNode4.prev = tail
tail = tail.next

```



Deletion in Doubly Linked Lists

- Deleting a node involves adjusting the prev and next pointers of adjacent nodes.
- Deletion is also *bigO*(1), but finding the node to delete may take $\mathcal{O}(n)$.

```
ListNode2 = tail.prev
ListNode2.next = null
tail = ListNode2
```



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.

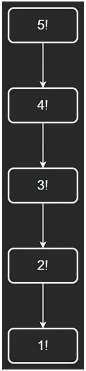
One-Branch Recursion Explained

- In one-branch recursion, the function makes a single recursive call.
- Contrasts with two-branch recursion, which splits into two recursive calls.
- Effective for problems that divide into sub-problems of the same type.

Factorial - A Recursive Example

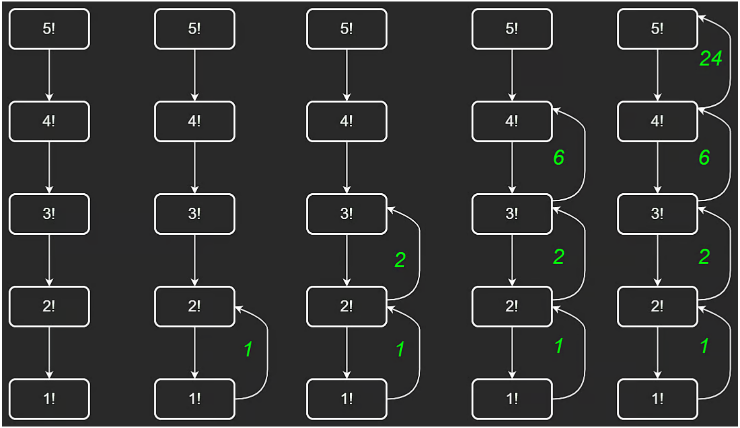
- Mathematical definition: $n! = n \times (n-1) \times (n-2) \times \ldots \times 1$
- Recursive definition: $n! = n \times (n-1)!$, with 1! as the base case.

```python
# Recursive implementation of
#n! (n-factorial) calculation
def factorial(n):
    # Base case: n = 0 or 1
    if n <= 1:
        return 1
    # Recursive case: n! = n * (n - 1)!
    return n * factorial(n - 1)
```



The Recursion Tree for Factorial

- Visualization of the recursion tree for factorial(5).
- Each level represents a recursive call.
- Base case as the stopping point for recursion.



Complexity of Recursive Factorial

- Time complexity: $O(n)$ - function called n times.
- Space complexity: $O(n)$ due to n stack frames.
- Any recursive algorithm can be written iteratively, and the other way around. The iterative implementation of this is the following:

```
n = 5
res = 1
while n > 1:
    res = res * n
    n -= 1
```

What is Two-Branch Recursion?

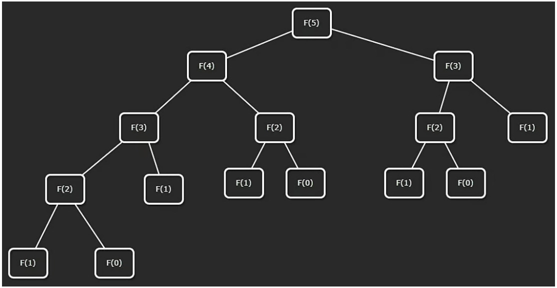
- Two-branch recursion occurs when a function makes two recursive calls.
- It's often used to solve problems where a current state depends on two previous states.
- A classic example: the Fibonacci sequence.
- To find the n^{th} Fibonacci number, we sum the $(n-1)^{th}$ and $(n-2)^{th}$ numbers.
- The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

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

Pseudocode for Recursive Fibonacci

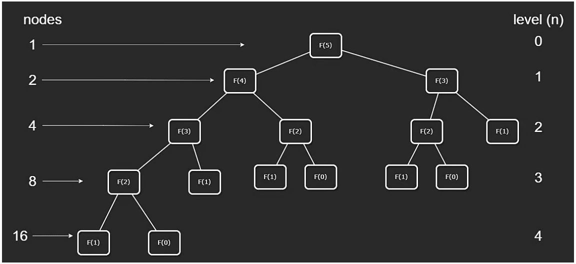
```
# Recursive implementation to calculate the n-th Fibonacci number
def fibonacci(n):
    # Base case: n = 0 or 1
    if n <= 1:
        return n

    # 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 $\text{fib}(n-1) + \text{fib}(n-2)$.



Time Complexity Evaluation

- Complexity of recursive Fibonacci: $O(2^n)$.
- Each level of the recursion tree doubles the number of nodes.
- Last level dominates the time complexity.



Summary:

- **Stacks**:
  - LIFO (Last In, First Out) principle.
  - Operations: push (add), pop (remove), peek (top element).
  - Use case: Undo mechanisms, function call stacks.
- **Queues**:
  - FIFO (First In, First Out) principle.
  - Operations: enqueue (add), dequeue (remove).
  - Use case: Printer job scheduling, breadth-first search.
- **Doubly Linked Lists**:
  - Nodes with two pointers: next and previous.
  - Operations: insertion, deletion at any position.
  - Use case: Navigable sequences, undo functionality in applications.
- **Recursion**:
  - Function calling itself to solve smaller instances of the problem.
  - Types: Single-branch (e.g., factorial), Multi-branch (e.g., Fibonacci).
  - Use case: Tree traversals, divide and conquer algorithms.