Stacks and Easy Problems on Stacks

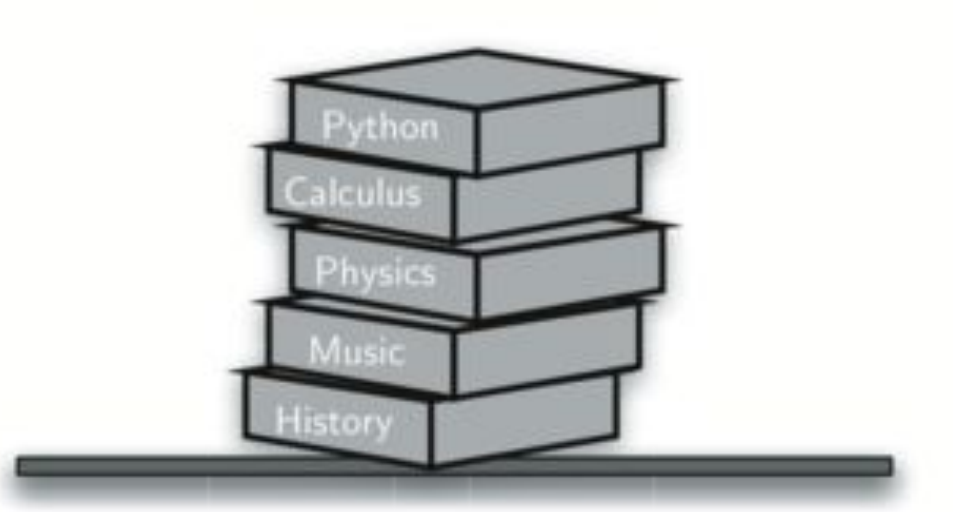
Relevel
by Unacademy

Educator Introduction / Pre-Requisites (10mins)

- In the previous lesson we learnt about dynamic programming and how it can solve complex optimization problems into simpler sub problems and memorization to eventually solve the bigger complex problem .
- In this lesson we will learn about a very useful data structure called stacks and how it can also be used to solve various problems which would have been hard without stacks .
- Pre-requisites : Node js basic programming , understanding of classes and object oriented programming will be useful

Introduction

- A stack is an ordered collection of items where both addition and removal of items takes place at the same end . Consider the following picture of a stack of books on a desk .



Introduction

As you can see here that at any given time , you can only pick the top most book . In this case that will be the book on Python. You cannot pick any book from the middle and if you want any middle book , you have to first remove books from the top to eventually reach that middle book .

- This data structure can be used to solve a variety of use cases e.g solving parenthesis , evaluating an expression , implementing undo - redo functionality , etc .
- Here, a popular notation to describe the stack is **LIFO** (Last In First Out), which means that whatever item is inserted last into the stack, only that item will be picked first, which you can see in our example of books.

Introduction

- We also have another term called **FIFO** (First In First Out) used for another popular data structure called Queue which will be covered in a different lesson, and we will not use that here .
- There are two very important and basic operations on stack which are push and pop. Since we have learnt that stack is implemented as LIFO both insertion and deletion happens from the same end . That end we usually called **Top** . When we insert an element into stack from the top , that operation is called **Push** . When we remove an element from the top of the stack , this operation is called **Pop**.

List of Concepts Involved

In this lesson we will cover following topics on stacks

- **Initialization**
- **Push**
- **isEmpty**
- **Pop**
- **Overflow**
- **Underflow**
- **Easy Problems on Stack**

Initialization : Explanation

- Before implementing various operations on stack , we will first learn how to create a stack .
- Here we will create a stack class which will have two attributes :
  - data : Is an array in which we will store the value
  - Top : Points to the top index of our stack
- Let’s implement this in code as follows

```
class Stack {
 constructor() {
   this.data = [];
   this.top = 0;
 }
}
```

Initialization : Explanation

- In the above approach we have the data attribute as an array , in a similar way we could even implement a stack using a linked list .But there will be few differences which can be described as follows :
    - First we will create a node class which will have value as well as next pointer pointing to the next element
    - We will no longer need the data attribute as all the data will be stored as nodes
    - The top will initially be null and it will always point to the top most node in the stack .
    - Take a look at the following code

Initialization : Explanation

```
class Node {
 constructor(value) {
   this.value = value;
   this.next = null;
 }
}
class Stack {
 constructor() {
   this.top = null;
 }
}
```

Initialization : Explanation

- Since we know that for stacks , both insertion and removal happen from the same position since it follows the principle of LIFO(Last In First Out) , that position is generally referred to as top .
- So we can say here that both insertion and deletion will happen from top and hence we need to know the value of top at all times
- Now we can initialise the tack like this

```
const stack = new Stack();
```

Initialization : Explanation

Try this question :

What will be the output for this

```
console.log(stack.data.length)
```

Answer: []

Explanation: Since we have just initialise the stack with an empty array , hence the length of stack is 0

Push : Explanation

Now that we have created an empty stack , we will learn how to insert an element to this stack.

- In stack , we always insert the element in the top position . The process of inserting the element into the stack at top position is called push.
- Since we are implementing the stack using an array , we need to increment the top variable every time we insert the element into the data array .
- Note we could have also implemented stack using linked list but in this lesson we will only focus on arrays method.
- Lets understand this by the following code

```
console.log(stack.data.length)
```

Push : Explanation

```
push(element) {
    this.data[this.top] = element;
    this.top = this.top + 1;
```

Now we can insert the element into the stack as follows

```
stack.push(100);
```

Push : Explanation

Try this question :

What will be the output for this

```
const stack = new Stack();

 stack.push(100);
 stack.push(200);

 console.log(stack.top);
```

Answer : 2

Explanation : Since we have inserted two elements into the stack , the top value will be 2 pointing to element 200

IsEmpty : Explanation

Let’s create one more method called isEmpty which will be used for various others operations on stack.

- In stack , let’s say for some reason we want to determine if the stack is empty or not.
- One can say that we could determine this by just checking if the length of data is 0 or not
- But the correct way to determine if the stack is empty or not is by looking at the value of stack
- Remember when we initialize the empty stack , we have used the top value as 0, we can do the same by checking if the top value is 0 , then it means the stack is empty otherwise it is not empty .
- The reason we do it this way is that it is consistent with other approaches of stack where even when we implement stack using linked list , we could determine the emptiness of stack just by looking at the top variable without worrying about data.
- Lets understand this by the following code

IsEmpty : Explanation

```
push(element) {
    this.data[this.top] = element;
    this.top = this.top + 1;
  }
```

Now we can check whether the stack is empty or not like this

```
stack.isEmpty()
```

IsEmpty : Explanation

Try this question :

What will be the output for this

```
const stack = new Stack();
 console.log(stack.isEmpty());
```

Answer : true

Explanation : Since the stack is empty and and the value of top is 0 , hence it will return true

Pop : Explanation

Now we will learn how to remove elements from the stack .

- In stack , just like insert we will remove elements from the same position. The process of removing the element from top is called pop operation .
- Here we have to remove elements in the top position of data and need to decrement the top variable.
- The good thing is that we already have an inbuilt javascript array method called pop which removes the last element from the array.
- But before popping the element , first we need to check if the stack is empty or not which we already created previously
- Lets understand this by the following code

Pop : Explanation

```
pop() {
    if (this.isEmpty() === false) {
      this.top = this.top - 1;
      return this.data.pop();
    }
  }
```

Now we can pop the element from stack like this

```
Const element = stack.pop()
```

Pop : Explanation

Try this question :
What will be the output for this

```
const stack = new Stack();
 stack.push(100);
 stack.push(200);
 console.log(stack.pop());
```

Answer : 200
Explanation : First we pushed 100 , so the top was pointing at 100 . Then we push 200 and the top is pointing at 200 . Now when we pop , we get the value of top which is 200 and the top of the stack now points at 100.

Overflow : Explanation

Now lets understand overflow condition for stack

- Let’s create the stack and when we keep pushing elements in the stack. At some point we don't have any memory left to push any more elements in the stack .
- This condition when are stack is full and you cannot push any more element in the stack is called overflow i.e. stack overflow
- In practical scenarios when we execute the code , we have a call stack to save values in memory and if the program is stuck at executing code , this call stack gets full and the program crashes .
- We don't really implement any specific logic in this lesson just remember that for advanced use cases this logic needs to be implemented as part of push method in stack.

Underflow : Explanation

Just like overflow there is a condition called underflow which we are going to learn now .

- Remember when we implemented the pop method , we checked if the stack was empty or not .
- We only allowed pop operation if the stack is not empty but we did not handle the case if the stack is empty.
- In the case when we try to remove the element from stack and if the stack is already empty , then this situation can be described as stack underflow.
- So we need to modify our popup function to handle the stack underflow .
- Lets understand this by the following code

Underflow : Explanation

```
pop() {
    if (this.isEmpty()) {
      throw new Error("Stack Underflow");
    }

    this.top = this.top - 1;
    return this.data.pop();
  }
```

Underflow : Explanation

Try this question :

What will be the output for this

```
const stack = new Stack();
 stack.push(100);
 stack.pop();
 stack.pop();
```

Answer : Throw Error Stack Underflow

Explanation : Here we first pushed 100 , so stack has one element . Now we performed pop operation and the stack becomes empty . Now when we try to perform one more pop operation , since the stack is empty , we will get the stack underflow error.

Peek : Explanation

Now we will learn about peek operation in stack.

- Let's say at any point of time , you want to get the top most element in stack , we call this operation as peek
- We can implement the peek functionality by just getting the value of the data at top position
- But before returning the top most element , we need to first check if the stack is empty or not
- If the stack is empty , we will return null otherwise we will return the top most element
- Lets understand this by the following code

```
peek() {
    if (this.isEmpty()) {
      return null;    }
    return this.data[this.top - 1];
  }
```

Peek : Explanation

Try this question :

What will be the output for this

```
const stack = new Stack();
 stack.push(100);
 stack.push(200);
 stack.push(300);
 console.log(stack.peek());
```

Answer : 300

Explanation : Here we have constructed the stack like this 100 → 200 → 300 and the topmost element is 300. Hence the result we will get back is 300

Print : Explanation

Now we will learn about print operation in stack.

- Let's say we want to print all the elements present in the stack .
- For this we have to traverse the data array from top to the last element of stack
- We will create an iterator called index and initialise it to top and keep decrementing the index till it is greater than or equal to 0 as we know indexes in array starts from 0.
- Lets understand this by the following code

```
print() {
    let index = this.top - 1;
    while (index >= 0) {
      console.log(this.data[index]);
      index--;    }
  }
```

Print : Explanation

Try this question :

What will be the output for this

```
const stack = new Stack();
 stack.push(100);
 stack.push(200);
 stack.push(300);
 stack.print();
```

Answer : 300

200

100

Explanation : Here we have constructed the stack like this 100 ➜ 200 ➜ 300 and the topmost element is 300. So we will keep printing the element from 300 , then 200 and finally 100.

Easy Problems on stack (P1)

Problem:

Implement a function called insertElementAtBottom which takes a number as input and it inserts the value at the bottom of that stack.

Scenario:

```
const stack = new Stack();
 stack.push(4);
 stack.push(3);
 stack.push(2);
 stack.push(1);

 stack.insertElementAtBottom(0);
 stack.print();
```

Output:

1
2
3
4
0

Easy Problems on stack (P1)

Code

```
 insertElementAtBottom(value) {
if (this.isEmpty()) {
  this.push(value);
  return;
}

const tmp = this.peek();
this.pop();
this.insertElementAtBottom(value);
this.push(tmp);
return;
```

Easy Problems on stack (P1)

Code WalkThough:

We are implementing this using a recursive function where we need to pass stack as a parameter . We will recursively call this insertElementAtBottom function till the stack is empty. When it is empty we will push our element which will be in the top as it will ne the only element at that point . Till the time stack is not empty , we will store the top most value in temp variable , pop the value and call the insertElementAtBottom recursively and again push that element back to stack . In this way our element will be inserted at the bottom of the stack.

Easy Problems on stack (P2)

Problem:

Recursively reverse elements in stack

Scenario:

```
const stack = new Stack();
 stack.push(4);
 stack.push(3);
 stack.push(2);
 stack.push(1);
 stack.reverse();
 stack.print();
```

Output:

4
3
2
1

Easy Problems on stack (P2)

Code

```
insertElementAtBottom(value) {
   if (this.isEmpty()) {
     this.push(value);
     return;
   }
   const tmp = this.peek();
   this.pop();
   this.insertElementAtBottom(value);
   this.push(tmp);
   return;
 }
```

Easy Problems on stack (P2)

Code

```
reverse() {
  if (!this.isEmpty()) {
    const temp = this.peek();
    this.pop();
    this.reverse();
    this.insertElementAtBottom(temp);
  }
}
```

Easy Problems on stack (P2)

Code WalkThough:

To implement this logic ,We can reuse our previous function called insertElementAtBottom where we will use this method till our stack is empty . For every iteration , we will pop it , call the reverse method and at the insert the popped element which we got from peek at the bottom .

Easy Problems on stack (P3)

Problem :

- Implement a function that takes a string which contains various parenthesis like ( ) { } [ ] as an input and returns whether it checks if it is the valid parenthesis or not .
- E.g. when we pass the following string to the function "({[]})” it should return true and if we pass this string “{[(]}]” , it should return false
- Try to implement this logic using following stack functions : push , pop , isEmpty

Easy Problems on stack (P3)

Code

```
class Stack {
 constructor() {
   this.data = [];
   this.top = 0;
 }
 push(element) {
   this.data[this.top] = element;
   this.top = this.top + 1;
 }
 isEmpty() {
   return this.top === 0;
 }
```

Easy Problems on stack (P3)

Code

```
 pop() {
   if (this.isEmpty()) {
     throw new Error("Stack Underflow");
   }
   this.top = this.top - 1;
   return this.data.pop();
 }
function matches(open, close) {
 const matchTable = {
   "(": ")",
   "[": "]",
   "{": "}"   };
```

Easy Problems on stack (P3)

Code

```
 return close === matchTable[open];
}
function isOpening(value) {
 const opens = ["(", "[", "{"];
 return opens.indexOf(value) > -1;
}
function isClosing(value) {
 const closes = [")", "]", "}"];
 return closes.indexOf(value) > -1;
}
```

Easy Problems on stack (P3)

Code

```
function checkParenthesis(str) {
 const stack = new Stack();
 for (const char of str) {
   if (isOpening(char)) {
     stack.push(char);
   } else if (isClosing(char)) {
     const open = stack.pop();
     const close = char;
     if (!matches(open, close)) {
       return false;     }   } }
 return stack.isEmpty();  }
```

Practice / HW assignment

1. Sort the elements of stack recursively . You should use stack methods like pop, push and peek . You should not create a new array but modify the existing array .
2. Write a function that returns the maximum and minimum element present in the stack.

Upcoming Class Teasers

In the next class we will use this knowledge on stack and we will learn about applying this on complex problems like Next Greater Element , Previous Greater Element and variations about it .

Thank you!