Stack

Table of Contents

Theory

What is Stack data structure?

A stack is a linear data structure in which elements are inserted and removed from one end only. It follows the Last-In-First-Out (LIFO) principle.

The time complexity of the basic stack operations - push, pop, and peek (or top) - is O(1)

What are uses of Stack data structure?

1. Compiler design: The stack is used to store and manage function calls and their local variables in a program's runtime environment. It is also used in implementing the syntax analysis phase of a compiler.
2. Operating systems: The stack is used to manage system calls and interrupts, as well as to allocate and deallocate memory dynamically.
3. Web browsing: The back button in a web browser uses a stack to keep track of the previously visited pages, allowing users to navigate back to the previous page.
4. Undo/Redo functionality: Many software applications use a stack to implement the undo/redo functionality, allowing users to undo and redo their actions.
5. Text editors: Text editors use a stack to implement the undo/redo functionality, as well as to store and manage the history of changes made to a document.
6. Parentheses balancing: The stack is used to check whether a given expression contains balanced parentheses or not.
7. Routing protocols: The stack is used in various routing protocols, such as the Border Gateway Protocol (BGP), to manage the path selection process.
8. Expression evaluation: The stack is used to evaluate expressions, such as postfix and prefix expressions.
9. Backtracking algorithms: The stack is used in backtracking algorithms, such as the N-Queens problem, to keep track of the previous states.

What is stack trace and how it is useful in debugging?

A stack trace is a report that displays the sequence of function calls that led up to an error or exception in a program. It provides a detailed overview of the functions that were called, the order in which they were called, and the arguments passed to them. This information can be used for debugging by helping to identify the root cause of an error or unexpected behaviour in the program.

Stack Implementation

```python
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if self.is_empty():
            return None
        else:
            return self.stack.pop()

    def peek(self):
        if self.is_empty():
            return None
        else:
            return self.stack[-1]

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)
```

How stack is used in function calling?

In programming languages, a stack data structure is commonly used in function calling to keep track of the order in which functions are called and to manage their local variables.

Here's how a stack is used in function calling:

1. When a function is called, the program stores its return address on the stack. This allows the program to know where to return to when the function completes.
2. The program then pushes the function's local variables onto the stack. These variables are only accessible within the function and are automatically popped off the stack when the function completes.
3. If the function calls another function, the program pushes the return address of the current function onto the stack and repeats the process for the new function.
4. When a function completes, the program pops the local variables and return address off the stack and returns to the previous function's return address. This process continues until the main function completes.

By using a stack to manage function calls and local variables, programs can easily handle recursive functions and nested function calls. It also allows programs to manage memory more efficiently by automatically deallocating memory allocated for local variables when the function completes.

Design a browser back button using a stack

To design a browser back button using a stack, we can use the following steps:

1. Create a stack to store the history of visited URLs.
2. Whenever a new page is visited, push the URL of that page onto the stack.
3. When the back button is clicked, pop the top URL from the stack and load that page.
4. If the stack is empty, then disable the back button since there is no previous page to go back to.

Here is some that implements a simple back button using a stack:

```python
import webbrowser

class Browser:
    def __init__(self):
        self.history = []

    def go_to(self, url):
        webbrowser.open(url)
        self.history.append(url)

    def go_back(self):
        if len(self.history) > 1:
            self.history.pop()
            url = self.history[-1]
            webbrowser.open(url)

browser = Browser()

# Visit some websites
browser.go_to('https://www.google.com')
browser.go_to('https://www.python.org')
browser.go_to('https://stackoverflow.com')

# Click the back button twice
browser.go_back()
browser.go_back()
```

How do you handle stack overflow and underflow conditions?

Stack overflow occurs when we try to push an element onto a full stack, and stack underflow occurs when we try to pop an element from an empty stack. To handle these conditions, we can use the following techniques: For stack overflow, we can check if the stack is full before pushing an element, and if it is full, we can either return an error message or resize the stack if possible. For stack underflow, we can check if the stack is empty before popping an element, and if it is empty, we can either return an error message or handle the situation in an appropriate way.

Monotonic Stack

A monotonic stack is a data structure that maintains either a strictly increasing or strictly decreasing order of its elements.Monotonic stacks are primarily used for solving problems involving finding the next greater element, the next smaller element, or similar tasks where you need to maintain a certain order or property in a sequence of elements.

Question. Given an array, for given index i, find first element a[j] such that a[j]>a[i] and j>i. basically, find first greater element in right of array, if it does not exist, ans is -1:

eg: for [1, 2, 4, 3, 10, 8, 9] => output: [2, 4, 10, 10, -1, 9, -1]

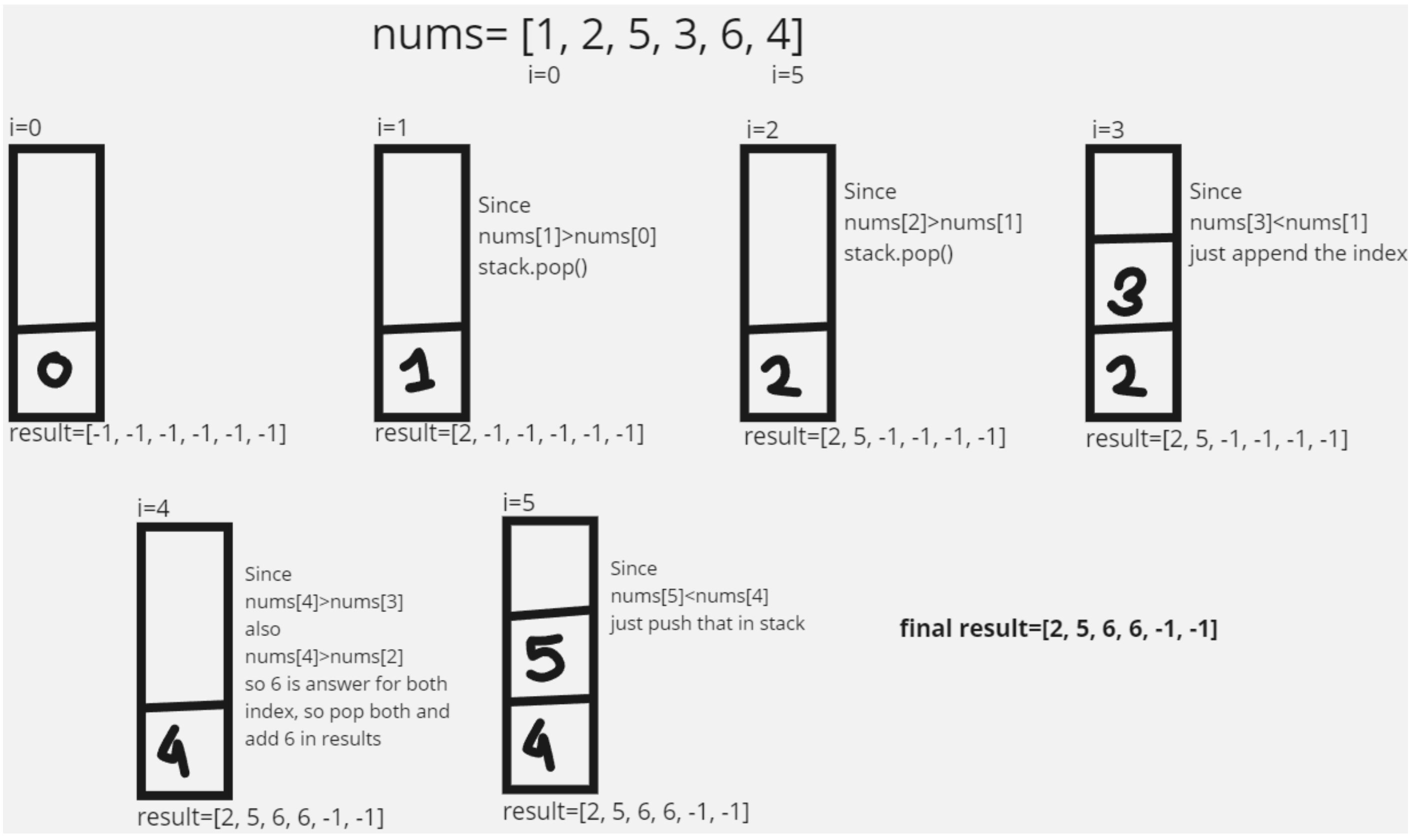
```python
def next_greater_elements(nums):
    stack = []
    result = [-1] * len(nums)

    for i in range(len(nums)):
        while stack and nums[i] > nums[stack[-1]]:
            result[stack.pop()] = nums[i]
        stack.append(i)

    return result

# Test the function
nums = [1, 2, 5, 3, 6, 4]
print(next_greater_elements(nums))  # Output: [2, 5, 6, 6, -1, -1]
```

DRY RUN



Time complexity: O(n)

1. The outer loop iterates through each element of the input list once. This contributes O(n) to the time complexity.
2. Inside the outer loop, we have a while loop that may execute multiple times, but the total number of iterations across all iterations of the outer loop is bounded by the total number of elements in the input list. This is because each element can be pushed onto the stack at most once and popped from the stack at most once. Therefore, the total time complexity of the while loop across all iterations of the outer loop is also O(n).

Since the inner while loop's complexity is not dependent on the outer loop, we can say that both loops contribute O(n) to the overall time complexity.

LEVEL 1: EASY

1. Valid Parentheses
   Link: https://leetcode.com/problems/valid-parentheses/description/

2. Minimum String Length After Removing Substrings
   Link: https://leetcode.com/problems/minimum-string-length-after-removing-substrings/description/

3. Remove Outermost parentheses
   Link: https://leetcode.com/problems/remove-outermost-parentheses/

4. Number of students unable to eat lunch
   Link: https://leetcode.com/problems/number-of-students-unable-to-eat-lunch/description/

5. Final Prices With a Special Discount in a shop
   Link: https://leetcode.com/problems/final-prices-with-a-special-discount-in-a-shop/description/

LEVEL 2: Medium

LEVEL 3: Difficult

SOLUTIONS:

LEVEL 1:

1. Valid Parenthesis

Detailed Solution

```
class Solution:
    def isValid(self, s: str) -> bool:
        stack=[]
        left = {"(","{","["}

        for i in s:
            if i in left:
                stack.append(i)
            else:
                if not stack:
                    return False

                if (i==")" and stack[-1]!="(") or (i=="}" and stack[-1]!="{") or (i=="]"
and stack[-1]!="[") :
                    return False
                else:
                    stack.pop()

        return not stack
```

2. Minimum String Length After Removing Substrings

Detailed Solution

```

class Solution:
    def minLength(self, s: str) -> int:
        stack = []
        for i in s:
            #can directly do if stack, to check if stack[-1] exist or not
            if stack and ((i=="B" and stack[-1]=="A") or (i=="D" and stack[-1]=="C")):
                stack.pop()
            else:
                stack.append(i)
        return len(stack)
```

3. Remove Outermost Parentheses

Detailed Solution

Here outer_stack will keep track of outermost parentheses,
If flag=0 means in inner most count of “(“ == “)” , thus now pop or push on outer_stack.

```
class Solution:
    def removeOuterParentheses(self, s: str) -> str:
        outer_stack = []
        inner_stack = []
        flag = 0

        for i in s:
            if i=="(":
                if not outer_stack:
                    outer_stack.append(i)
                else:
                    inner_stack.append(i)
                    flag += 1
            else:
                if flag:
                    inner_stack.append(i)
                    flag -=1
                else:
                    outer_stack.pop()

        return "".join(inner_stack)
```

4. Number of students unable to eat lunch

Detailed Solution

```
#Real implementation of scenario
from collections import deque
class Solution:
    def countStudents(self, students: List[int], sandwiches: List[int]) -> int:
        dq_st, dq_sa = deque(students), deque(sandwiches)
        counter = 0
        while dq_st:
            #means we moved one circle, but can't pop any
            if counter == len(dq_st):
                break

            cur_st = dq_st.popleft()
            if cur_st == dq_sa[0]:
                dq_sa.popleft()
                counter = 0
            else:
                dq_st.append(cur_st)
                counter += 1
        return len(dq_st)
```

We use the collections.deque in place of a list to implement a stack in Python because deque is more efficient than a list for implementing a stack. Here are some reasons why deque is preferred over a list for implementing a stack:

- Faster append and pop operations: deque is implemented as a doubly-linked list, which means that adding or removing elements from the beginning or end of the deque is faster than with a list. This makes it more suitable for implementing a stack.
- Thread-safe: deque is thread-safe, which means that it can be used in a multi-threaded environment without requiring additional synchronization.
- Memory efficient: deque is more memory efficient than a list, especially when dealing with large data sets. This is because deque only needs to store pointers to the next and previous elements, while a list needs to store the entire data element.
- Extended functionality: deque provides additional functionality that is not available with a list, such as the ability to efficiently rotate the elements, which can be useful in certain applications.

Approach 2 with logic

```python
class Solution:
    def countStudents(self, students: List[int], sandwiches: List[int]) -> int:
        st0,st1,sa0,sa1=0,0,0,0
        n=len(students)
        for i in range(n):
            if students[i]==0:
                st0+=1
            else:
                st1+=1

            if sandwiches[i]==0:
                sa0+=1
            else:
                sa1+=1

        if st0==sa0:
            return 0
        if sa0>st0:
            c0=0
            for i in range(n):
                if sandwiches[i]==0:
                    c0+=1
                if c0>st0:
                    return n-i
        else:
            c1=0
            for i in range(n):
                if sandwiches[i]==1:
                    c1+=1
                if c1>st1:
                    return n-i
```

5. Final Prices With a Special Discount in a Shop

Detailed Solution

```
class Solution:
    def finalPrices(self, prices: List[int]) -> List[int]:
        #Using monotonic stack concept O(n) solution
        stack=[]
        n=len(prices)
        for i in range(n):
            while stack and prices[i]<=prices[stack[-1]]:
                prices[stack.pop()] -=prices[i]
            stack.append(i)
        return prices
```

LEVEL 2:

LEVEL 3: