Introduction to Stack Data Structure

Lesson Plan

PW SKILLS

What is a Stack?

A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear). It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

Some key points related to stack

It is called as stack because it behaves like a real-world stack, piles of books, etc.
A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.

It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack
Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.

DS Stack Introduction

Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

The following are some common operations implemented on the stack:
The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.
- **peek():** It returns the element at the given position.
- **count():** It returns the total number of elements available in a stack.
- **change():** It changes the element at the given position.
- **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

Before inserting an element in a stack, we check whether the stack is full.
If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top=top+1, and the element will be placed at the new position of the top.
The elements will be inserted until we reach the max size of the stack.

POP operation

The steps involved in the POP operation is given below:

Before deleting the element from the stack, we check whether the stack is empty.
If we try to delete the element from the empty stack, then the underflow condition occurs.
If the stack is not empty, we first access the element which is pointed by the top
Once the pop operation is performed, the top is decremented by 1, i.e., top=top-1.

Applications of Stack

The following are the applications of the stack:

Balancing of symbols: Stack is used for balancing a symbol. For example, we have the following program:

```
{
print("Hello");
print("pythont");
}
```

As we know, each program has an opening and closing braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

String reversal: Stack is also used for reversing a string. For example, we want to reverse a "PythonTpoint" string, so we can achieve this with the help of a stack.

First, we push all the characters of the string in a stack until we reach the null character.

After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

UNDO/REDO: It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

Recursion: The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.

DFS(Depth First Search): This search is implemented on a Graph, and Graph uses the stack data structure. Backtracking: Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

Expression conversion: Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:

Infix to prefix
Infix to postfix
Prefix to infix
Prefix to postfix

Postfix to infix

Memory management: The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Stack STL:

The functions associated with stack are:

- empty() – Returns whether the stack is empty – Time Complexity : O(1)
- size() – Returns the size of the stack – Time Complexity : O(1)
- top() – Returns a reference to the top most element of the stack – Time Complexity : O(1)
- push(g) – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)
- pop() – Deletes the most recent entered element of the stack – Time Complexity : O(1)
- empty() – Returns whether the stack is empty – Time Complexity : O(1)
- size() – Returns the size of the stack – Time Complexity : O(1)
- top() – Returns a reference to the top most element of the stack – Time Complexity : O(1)
- push(g) – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)
- pop() – Deletes the most recent entered element of the stack – Time Complexity : O(1)

Q1. Write a program to reverse a stack.

Input: elements present in stack from top to bottom 1 2 3 4
Output: 4 3 2 1

Input: elements present in stack from top to bottom 1 2 3
Output: 3 2 1

Reverse a stack:

The idea of the solution is to hold all values in Function Call Stack until the stack becomes empty. When the stack becomes empty, insert all held items one by one at the bottom of the stack.

Illustration:

Below is the illustration of the above approach

Let given stack be

1
2
3
4

After all calls of reverse, 4 will be passed to function insert at bottom, after that 4 will pushed to the stack when stack is empty

4

Then 3 will be passed to function insert at bottom , it will check if the stack is empty or not if not then pop all the elements back and insert 3 and then push other elements back.

4
3

Then 2 will be passed to function insert at bottom , it will check if the stack is empty or not if not then pop all the elements back and insert 2 and then push other elements back.

4
3
2

Then 1 will be passed to function insert at bottom , it will check if the stack is empty or not if not then pop all the elements back and insert 1 and then push other elements back.

4
3
2
1

Follow the steps mentioned below to implement the idea:

- Create a stack and push all the elements in it.
- Call reverse(), which will pop all the elements from the stack and pass the popped element to function insert_at_bottom()
- Whenever insert_at_bottom() is called it will insert the passed element at the bottom of the stack.
- Print the stack

Below is the implementation of the above approach:

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

    def is_empty(self):
        return self.items == []

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()

    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    def size(self):
        return len(self.items)

def insert_at_bottom(stack, item):
    if stack.is_empty():
        stack.push(item)
    else:
        temp = stack.pop()
        insert_at_bottom(stack, item)
        stack.push(temp)
def reverse(stack):
    if not stack.is_empty():
        temp = stack.pop()
        reverse(stack)
        insert_at_bottom(stack, temp)

# Example Usage
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
stack.push(4)

print("Original Stack")
stack.display()
reverse(stack)

print("Reversed Stack")
stack.display()
```

Original Stack
4 3 2 1
Reversed Stack
1 2 3 4

Time Complexity: O(N^2).
Auxiliary Space: O(N) use of Stack

Q2. Given a source stack, copy the contents of the source stack to the destination stack maintaining the same order without using extra space.

Examples:

Input : Source:- |3|
|2|
|1|
Output : Destination:- |3|
|2|
|1|
Input : Source:- |a|
|b|
|c|
Output : Destination:- |a|
|b|
|c|

Approach:

In order to solve this without using extra space, we first reverse the source stack, then pop the top elements of the source stack one by one and push it into the destination stack. We follow the below steps to reverse the source stack:

- Initialize a variable count to 0.
- Pop the top element from the source stack and store it in variable topVal.
- Now pop the elements from the source stack and push them into the dest stack until the length of the source stack is equal to count.
- Push topVal into the source stack and then pop all the elements in the dest stack and push them into the source stack.
- Increment the value of count.
- If count is not equal to length of source stack – 1, repeat the process from step-2.

Below is the implementation of the above approach:

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

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

    def pop(self):
        if not self.is_empty():
            return self.stack.pop
 return None

def length(self):
        return len(self.stack)

    def display(self):
        for i in range(len(self.stack) - 1, -1, -1):
            print(self.stack[i])
        print()
```

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

if __name__ == "__main__":
    source = Stack()  # Source Stack
    dest = Stack()    # Destination Stack

    source.push(1)
    source.push(2)
    source.push(3)

    print("Source Stack:")
    source.display()

    count = 0
# Reverse the order of the values in source stack
    while count ≠ source.length() - 1:
        top_val = source.pop()
        while count ≠ source.length():
            dest.push(source.pop())
        source.push(top_val)
        while not dest.is_empty():
            source.push(dest.pop())
        count += 1
 # Pop the values from source and push into destination stack
    while not source.is_empty():
        dest.push(source.pop())

    print("Destination Stack:")
    dest.display()
```

Output
Source Stack:
3
2
1

Destination Stack:
3
2
1

Complexity Analysis:

Time Complexity: O(n2)
Auxiliary Space: O(n)

Efficient Approach: A better approach would be to represent the stack as a linked list. Reverse the source stack in the same way we reverse a linked list, pop the top elements of the source stack one by one and push it into the destination stack.

Below is the implementation of the above approach:

```
class StackNode:
    def __init__(self, val):
        self.data = val
        self.next = None

class Stack:
    def __init__(self):
        self.top = None

    def push(self, value):
        new_val = StackNode(value)
        if self.top is None:
            self.top = new_val

else:
            new_val.next = self.top
            self.top = new_val

    def pop(self):
        if self.top is None:
            return None
        val = self.top.data
        self.top = self.top.next
        return val

    def display(self):
        current = self.top
        while current is not None:
            print(current.data)
            current = current.next
        print()

    def reverse(self):
        current = self.top
        temp = None
        prev = None
        while current is not None:
            temp = current.next
            current.next = prev
            prev = current
            current = temp
        self.top = prev
 def is_empty(self):
        return self.top is None

if __name__ == "__main__":
    source = Stack()  # Source Stack
    dest = Stack()    # Destination Stack

    source.push(1)
    source.push(2)
    source.push(3)

    print("Source Stack:")
    source.display()

    source.reverse()
# Pop the values from source and push into destination stack
    while not source.is_empty():
        dest.push(source.pop())

    print("Destination Stack:")
    dest.display()
```

Output
Source Stack:
3
2
1

Destination Stack:
3
2
1

Complexity Analysis:

Time Complexity: O(n)
Auxiliary Space: O(n)

Q3. Given a Stack S, the task is to print the elements of the stack from top to bottom such that the elements are still present in the stack without their order being changed.

Examples:

Input: S = {2, 3, 4, 5}
Output: 5 4 3 2

Input: S = {3, 3, 2, 2}
Output: 2 2 3 3

Recursive Approach: Follow the steps below to solve the problem:

- Create a recursive function having stack as the parameter.
- Add the base condition that, if the stack is empty, return from the function.
- Otherwise, store the top element in some variable X and remove it.
- Print X, call the recursive function and pass the same stack in it.
- Push the stored X back to the Stack.

Below is the implementation of the above approach:

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

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

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return None
```

```
def is_empty(self):
        return len(self.items) == 0

def print_stack(s):
    if s.is_empty():
        return
    x = s.peek()
    s.pop()
    print(x, end=" ")
    print_stack(s)
    s.push(x)

if __name__ == "__main__":
    s = Stack()
    s.push(1)
    s.push(2)
    s.push(3)
    s.push(4)
print_stack(s)
```

Given a stack S and an integer N, the task is to insert N at the bottom of the stack.

Examples:

Input: N = 7
S = 1 <- (Top)
2
3
4
5
Output: 1 2 3 4 5 7

Input: N = 17
S = 1 <- (Top)
12
34
47
15
Output: 1 12 34 47 15 17

Program to insert an element at the Bottom of a Stack

The simplest approach would be to create another stack. Follow the steps below to solve the problem:

Initialize a stack, say temp.
Keep popping from the given stack S and pushing the popped elements into temp, until the stack S becomes empty.

Push N into the stack S.
Now, keep popping from the stack 'temp' and push the popped elements into the stack S, until the stack temp becomes empty.

Below is the implementation of the above approach:

```
class Stack:
    def __init__(self):
        self.items = []
def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return None

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

def insert_to_bottom(S, N):
temp = Stack()

    # Transfer all elements from S to temp
    while not S.is_empty():
        temp.push(S.pop())

    # Push the new element to the bottom of the original stack
    S.push(N)

    # Transfer all elements back from temp to S
    while not temp.is_empty():
        S.push(temp.pop())
# Print the stack elements
    while not S.is_empty():
        print(S.peek(), end=" ")
        S.pop()
    print()

if __name__ == "__main__":
    S = Stack()
    S.push(5)
    S.push(4)
    S.push(3)
    S.push(2)
    S.push(1)

    N = 7
insert_to_bottom(S, N)
```

Output:
1 2 3 4 5 7

Time Complexity: O(N)
Auxiliary Space: O(N)

Stack Underflow — An error called when an item is called from a stack, but the stack is empty.
Stack Overflow — An error called when an item is pushed onto a stack, but the stack is full.
Arraylist implementation of stack:

push – To push the elements(Objects) into the Stack.
pop – To pop the top Object from the Stack
peek – To view the Top Object . This will not modify the Stack.
size – Get the size of the Stack
isEmpty – to check if the Stack is empty or not.

Code:

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

    def __str__(self):
        return f"Stack [elements={self.elements}]"

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

    def pop(self):
        if self.is_empty():
            return None
        return self.elements.pop()

def push(self, element):
        self.elements.append(element)

    def size(self):
        return len(self.elements)

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

if __name__ == "__main__":
    stack = Stack()
    print(f"Is Stack Empty: {stack.is_empty()}")

    stack.push("Gyan")
    stack.push("Vivek")
    stack.push("Rochit")
    stack.push("Panda")

print(f"Is Stack Empty: {stack.is_empty()}")
    print(stack)

    print(f"Stack Size: {stack.size()}")
    print(f"Peek Top Element: {stack.peek()}")
    print(f"After peek: {stack}")

    print(f"Pop Top Element: {stack.pop()}")
    print(f"After pop: {stack}")
    print(f"Stack Size now: {stack.size()}")
```

Output

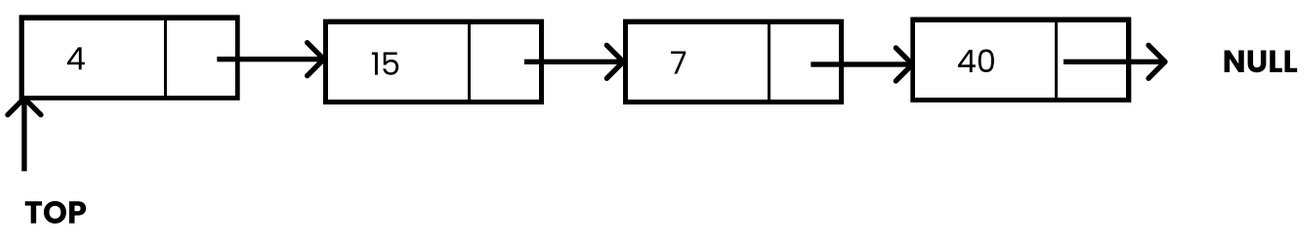
Is Stack Empty:true
Is Stack Empty:false
Stack [elements=[Gyan, Vivek, Rochit, Panda]]

Stack Size:4
Peek Top Element: Panda
After peek: Stack [elements=[Gyan, Vivek, Rochit, Panda]]
Pop Top Element: Panda
After pop: Stack [elements=[Gyan, Vivek, Rochit]]

Stack Size now:3
Stack Implementation using Linked List in Python
We will discuss Stack implementation using Linked List in Python. Instead of using an array, we can also use a linked list to implement a Stack. The linked list allocates the memory dynamically. However, time complexity in both scenarios is the same for all the operations i.e. push, pop, and peek.

In the linked list implementation of a Stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the Stack. A Stack is said to be overflown if the space left in the memory heap is not enough to create a node.

A push operation is implemented by inserting an element at the beginning of the list.
A pop operation is implemented by deleting the node from the beginning (the header/top node).



Why Linked List?

Using a linked list to implement a stack provides us with the ability to have a dynamically sized stack. This means we won't face the "stack overflow" issue if we reach a certain size, unlike array implementations.
Implementation of a Stack using Linked List in Python

Code:

```
class ListNode:
    def __init__(self, data=None):
        self.next = None
        self.data = data if data is not None else float('-inf')

    def get_next(self):
        return self.next

    def set_next(self, node):
        self.next = node

    def get_data(self):
        return self.data

    def set_data(self, data):
```

```
  self.data = data

    def __str__(self):
        return str(self.data)


class EmptyStackException(Exception):
    pass


class LinkedStack:
    def __init__(self):
        self.length = 0
        self.top = None

    def push(self, data):
        temp = ListNode(data)
        temp.set_next(self.top)
        self.top = temp
        self.length += 1

    def pop(self):
        if self.is_empty():
            raise EmptyStackException("Stack is empty")
        result = self.top.get_data()
        self.top = self.top.get_next()
        self.length -= 1
        return result

    def peek(self):
        if self.is_empty():
            raise EmptyStackException("Stack is empty")
        return self.top.get_data()

    def is_empty(self):
        return self.length == 0

    def size(self):
        return self.length

    def __str__(self):
        result = ""
        current = self.top
        while current is not None:
            result += str(current) + "\n"
            current = current.get_next()
        return result

if __name__ == "__main__":
    stack = LinkedStack()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    stack.push(4)
    stack.push(5)

    print(stack)

    print(f"Size of stack is: {stack.size()}")

    stack.pop()
    stack.pop()

    print(f"Top element of stack is: {stack.peek()}")
```

Output:
Size of stack is: 5
Top element of stack is: 3

Time and space complexity analysis:

Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

- Space Complexity (for n push operations) O(n)
- Time Complexity of create Stack: DynArrayStack() O(1)
- Time Complexity of push() O(1) (Average)
- Time Complexity of pop() O(1)
- Time Complexity of top() O(1)
- Time Complexity of isEmpty() O(1)
- Time Complexity of deleteStack() O(n)

Comparing Array Implementation & Linked List Implementation
Array Implementation

- Operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of n operations (starting from an empty stack) - “amortized” bound takes time proportional to n.

Linked List Implementation

- Grows and shrinks gracefully.
- Every operation takes constant time O(1).
- Every operation uses extra space and time to deal with references.