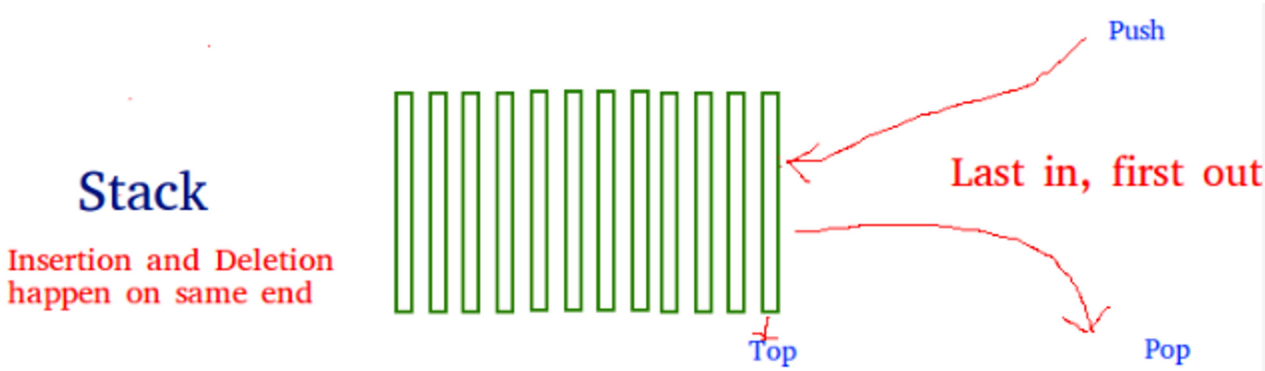
June9

09 June 2020 02:11

Stack in Python

A stack is a linear data structure that stores items in a Last-In/First-Out **(LIFO)** or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called **push** and **pop**.



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 top most element of the stack – Time Complexity : O(1)

Implementation

There are various ways from which a stack can be implemented in Python. This article covers the implementation of stack using data structures and modules from Python library.

Stack in Python can be implemented using following ways:

- list
- collections.deque
- queue.LifoQueue

Implementation using list

Python's builin data structure list can be used as a stack. Instead of **push()**, **append()** is used to add elements to the top of stack while **pop()** removes the element in LIFO order.

Drawback: The biggest issue is that it can run into speed issue as it grows. The items in list are stored next to each other in memory, if the stack grows bigger than the block of memory that currently hold it, then Python needs to do some memory allocations.

Implementation using collections.deque

Python stack can be implemented using deque class from collections module.

Advantage: Deque is preferred over list in the cases where we need quicker append and pop operations from both the ends of the container, as deque provides an O(1) time complexity for append and pop operations as compared to list which provides O(n) time complexity.

- ❖ Same methods on deque as seen in list are used, append() and pop().

Implementation using queue module

Queue module also has a LIFO Queue, which is basically a Stack. Data is inserted into Queue using **put()** function and **get()** takes data out from the Queue.

There are various functions available in this module:

- **maxsize –** Number of items allowed in the queue.
- **empty() –** Return True if the queue is empty, False otherwise.
- **full() –** Return True if there are maxsize items in the queue. If the queue was initialized with maxsize=0 (the default), then full() never returns True.
- **get() –** Remove and return an item from the queue. If queue is empty, wait until an item is available.
- **get_nowait() –** Return an item if one is immediately available, else raise QueueEmpty.
- **put(item) –** Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.
- **put_nowait(item) –** Put an item into the queue without blocking.
- **qsize() –** Return the number of items in the queue. If no free slot is immediately available, raise QueueFull.

Applications of stack:

1. **Expression Handling −**

- Infix to Postfix or Infix to Prefix Conversion −

  The stack can be used to convert some infix expression into its postfix equivalent, or prefix equivalent. These postfix or prefix notations are used in computers to express some expressions. These expressions are not so much familiar to the infix expression, but they have some great advantages also. We do not need to maintain operator ordering, and parenthesis.

- Postfix or Prefix Evaluation −

  After converting into prefix or postfix notations, we have to evaluate the expression to get the result. For that purpose, also we need the help of stack data structure.

2. **Backtracking Procedure −**

   Backtracking is one of the algorithm designing technique. For that purpose, we dive into some way, if that way is not efficient, we come back to the previous state and go into some other paths. To get back from current state, we need to store the previous state. For that purpose, we need stack. Some examples of backtracking is finding the solution for Knight Tour problem or N-Queen Problem etc.

3. Another great use of stack is during the function call and return process. When we call a function from one other function, that function call statement may not be the first statement. After calling the function, we also have to come back from the function area to the place, where we have left our control. So we want to resume our task, not restart. For that reason, we store the address of the program counter into the stack, then go to the function body to execute it. After completion of the execution, it pops out the address from stack and assign it into the program counter to resume the task again.

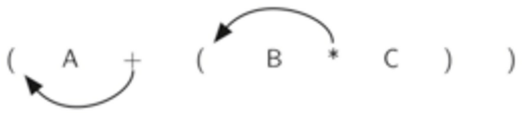
- **Expression Handling**

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Infix to Postfix

Infix expression: The expression of the form a op b. When an operator is in-between every pair of operands.
Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.

Moving Operators to the Right for Postfix Notation



': Moving Operators to the Left for Prefix Notation

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d
If op1 = +, op2 = *, op3 = +

The compiler first scans the expression to evaluate the expression b * c, then again scan the

expression to add a to it. The result is then added to d after another scan.

The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.

The corresponding expression in postfix form is: abc*+d+. The postfix expressions can be evaluated easily using a stack.

Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
…..3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
…..3.2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is '(', push it to the stack.
5. If the scanned character is ')', pop the stack until '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Evaluation of Postfix Expression

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.

Algorithm:

1) Create a stack to store operands (or values).
2) Scan the given expression and do following for every scanned element.
…..a) If the element is a number, push it into the stack
…..b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
3) When the expression is ended, the number in the stack is the final answer

Example:

Let the given expression be "**2 3 1 * + 9 -**". We scan all elements one by one.

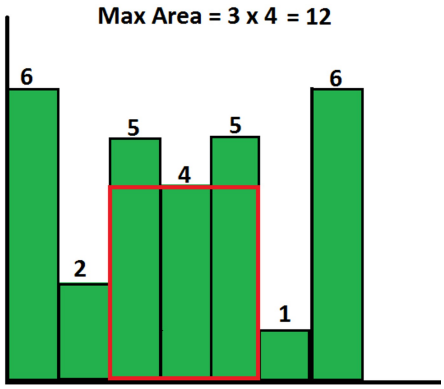
- Scan '2', it's a number, so push it to stack. Stack contains '2'
- Scan '3', again a number, push it to stack, stack now contains '2 3' (from bottom to top)
- Scan '1', again a number, push it to stack, stack now contains '2 3 1'
- Scan '*', it's an operator, pop two operands from stack, apply the * operator on operands, we get 3*1 which results in 3. We push the result '3' to stack. Stack now becomes '2 3'.
- Scan '+', it's an operator, pop two operands from stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result '5' to stack. Stack now

becomes ‘5’.

- Scan ‘9’, it’s a number, we push it to the stack. Stack now becomes ‘5 9’.
- Scan ‘-‘, it’s an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4. We push the result ‘-4’ to stack. Stack now becomes ‘-4’.
- There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

Questions to implement

1. **Infix to Postfix Conversion, and postfix evaluation using stack.**
2. **Given a stack, how to reverse the elements of the stack using only stack operations (push & pop)**
3. **Given n non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram. For example, consider the following histogram with 7 bars of heights {6, 2, 5, 4, 5, 1, 6}. The largest possible rectangle possible is 12 (see the below figure, the max area rectangle is highlighted in red) Note: Solve the problem in both complexity O(nlog) and O(n)**



4. **Given a non-negative integer num represented as a string, remove k digits from the number so that the new number is the smallest possible. Note: The length of num is less than 10002 and will be ≥ k. The given num does not contain any leading zero.**

Example 1:

```
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new
number 1219 which is the smallest.
```

Example 2:

```
Input: num = "10200", k = 1
Output: "200"
Explanation: Remove the leading 1 and the number is 200. Note that
the output must not contain leading zeroes.
```

Example 3:

```
Input: num = "10", k = 2
Output: "0"
Explanation: Remove all the digits from the number and it is left
with nothing which is 0.
```