Stacks

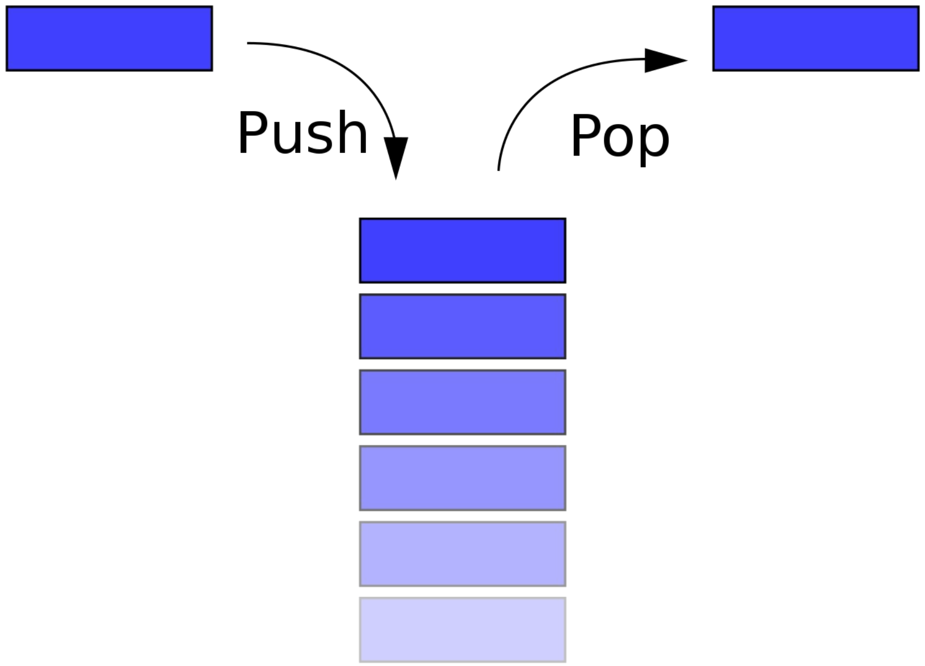
Agenda

- Introduction to stack
- Stacks: ADT
- Stacks: Example
- Stack implementation
- Stack implementation using Array
- Application: Virtual machine
- Application: Time span
- Growable array stack implementation

Introduction to stack

- A stack works on the principle of last-in-first-out (LIFO). The object/data that is added at last will be deleted first.
- Data insertion and deletion in stack will happen from one end only. At any time you can access only the last element of the stack.
- **Operations:**
    - **Push:** Insertion in stack is known as push operation.
    - **Pop:** Deletion of data in stack is known pop operation.

Introduction to stack

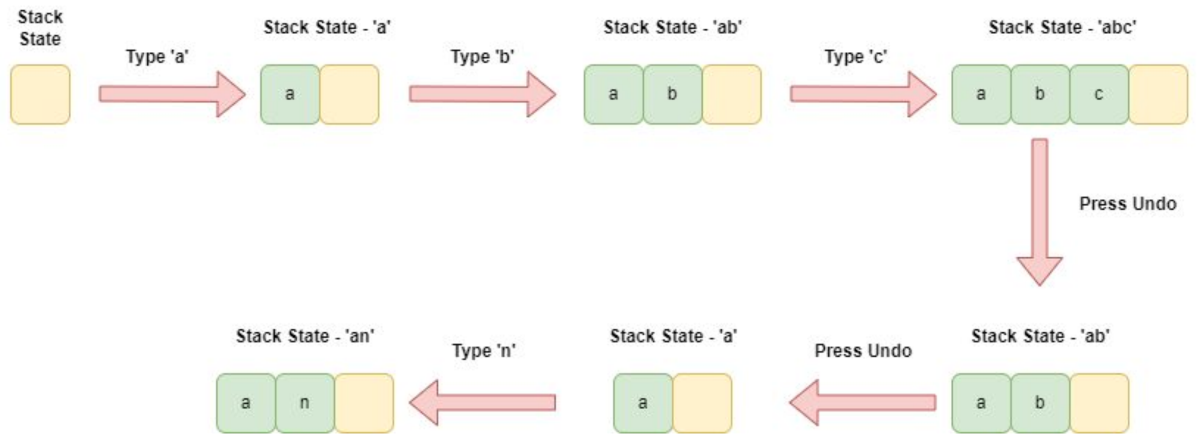


Stack ADT

- Generic operations on Stack data structure.
  - **new():** – Create new stack S.
  - **push():** - All the elements will be inserted at the top of the stack. Top value will be updated before inserting the element
  - **pop():** - Element deletion from the stack.
  - **peek():** – Returns the element that is available at the top of the stack.
- The following support methods could also be defined:
  - **size():** - Returning the number of elements in the stack.
  - **isEmpty():** - This method will be used to check if the given stack is empty, if so, it will return true.
- Axioms
  - **Pop(Push(S, v))** = S
  - **Peek(Push(S, v))** = v

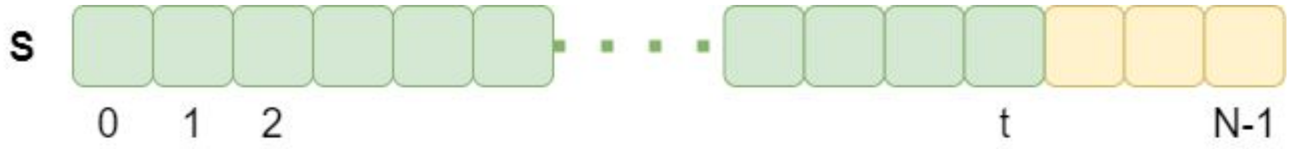
Stack example

- Let’s have a look at one of the common example known as UNDO/REDO feature.
  - Here type means inserting an element or pushing data.
  - After every operation contents of stack is shown.
  - Undo means erasing the last changes and reverting to older state.



Stack implementation using Array

- Implementation of stack using Array or python lists. Below are the operations:
  - Creating an array S of size N. This means that we can not have more than N elements in array.
  - A variable t will be maintained, to help with the ADTs of stack such as push, pop and peek.
  - t will be initialized with -1.



Stack implementation using Array

Stack Pseudocode

```
class Stack:
# Implementation of Stack using Arrays
# Initialize the Stack with default capacity or given capacity
    def __init__(self, capacity = 1024):
        self.capacity = capacity
        self.objects = [0] * self.capacity
        self.top = -1

    def __size(self):                #Return the current stack size
        return (self.top + 1)

    def __isEmpty(self):               #Return true iff the stack is empty
        return (self.top < 0)
```

Stack implementation using Array

Stack Pseudocode

```
def push(self, value):            #Push a new element on the stack
    if (self.__size() == self.capacity):
        print("Stack full")
    else:
        self.top += 1
        self.objects[self.top] = value

def peek(self):                   #Return the top stack element
    if (self.__isEmpty( )):
        print("Stack is empty.")
        return
    return self.objects[self.top]

def pop(self):                    #Pop off the stack element
    if (self.__isEmpty()):
        print("Stack is Empty.");
        return
    elem  = self.objects[self.top]
    self.top = self.top-1         #decrement top
    return elem
```

Stack implementation using Array

- Let’s do the analysis of operations on Array implementation of stack:
    - **push():** constant time; insertion always at the top of stack.
    - **pop():** constant time; deletion always at the top of stack.
    - **peek():** constant time; returning the last inserted element of the stack.

Stack applications

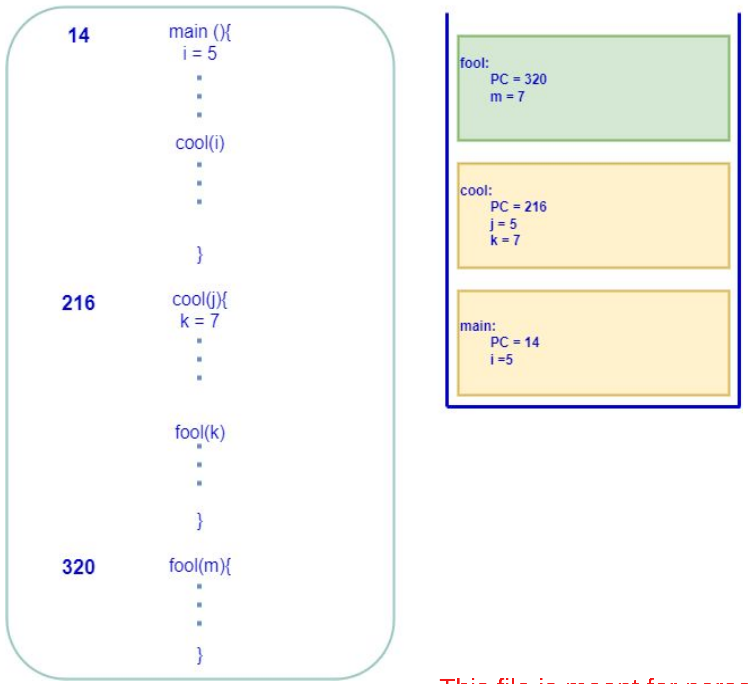
- Redo-Undo in the editors
- Forward-Backward operations in browser
- Memory management
- Implementation of DFS (Depth First Search)
- N-queen’s problem
- Conversion from Infix to postfix expression

Stacks application: Virtual machine

- Every process will maintain its own stack to store data relevant to the process execution.
- Every time a new method is called, it will be inserted into the stack.
- Having stack in these types of situation have several benefits:
  - Implementing recursive method calls.
  - Using stack traces to find error in the program.

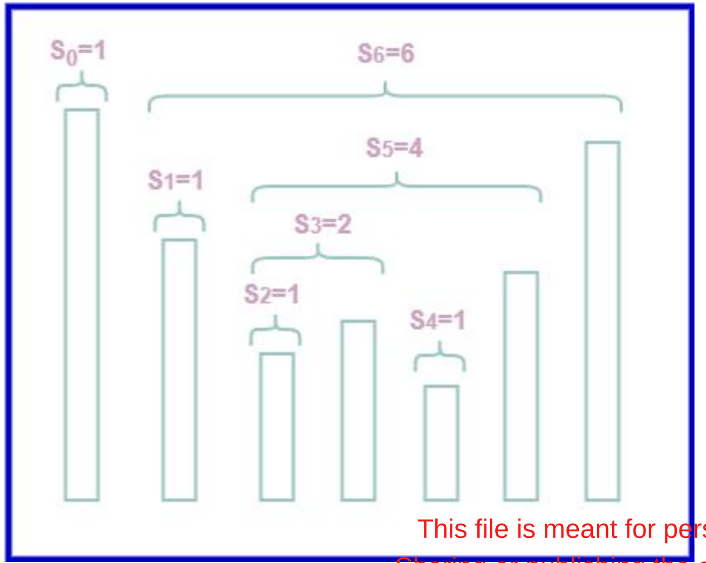
Stacks application: Virtual machine

Method Stack



Application: Time span

- The span s_i of a stock's price on a particular day i is described as the maximum number of consecutive days up to the current day, for which the price of the stock on the current day is less than its price on the given day.
- For any day i, the price of stock should be less than or equal to price on day i.



Application: Time span

- **An inefficient algorithm**

Algorithm ComputeSpans1(P):

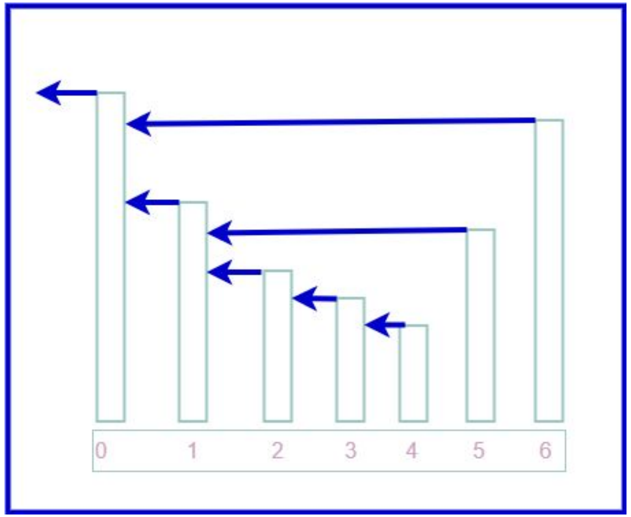
```
Input: an n-element array P of numbers such
that P[i] is  the price of the stock on day i
Output: an n-element array S of numbers such
that S[i]  is the span of the stock on day i

    for i ← 0 to n - 1 do
        k ← 0; done ← false
        repeat
            if P[i - k] ≤ P[i] then k ← k + 1
                else done ← true
        until (k = i) or done
        S[i] ← k
    return S
```

- The above mentioned algorithm is using two loops resulting quadratic n^2 time. Why?

Application: Time span

- **Lets see how stack can be used:**
  - We can calculate s_i easily, if we know the closest day preceding i, on which the price is greater than the price on day i.
  - Let's call the day that has this property as d(i), otherwise by convention of stack we will define it d(i) = -1.
  - Let's have a look at the adjacent figure; In the figure, d(3)=2,

    d(5)=1 and d(6)=0.
  - The span is now be recomputed as s_i = i - d(i).



Application: Time span

- **An efficient algorithm**

Algorithm ComputeSpans2(P):

```
Let D be an empty stack

for i ←0 to n - 1 do
        k ← 0; done ← false
        while not (D.isEmpty() or done) do
                if P[i] ≥ P[D.top()] then D.pop()
                        else done ← true
        if D.isEmpty() then h ← -1
                else h ← D.top()
        S[i] ← i - h
        D.push(i)
return S
```

A growable Array-based stack

- Let’s have a look at the unique issue that occurs in array based stack.
- We know that because size of array is fixed, push operation will have it limitation.
- Instead of returning the StackOverFlow; an efficient approach is to replace the existing array S with a bigger one and continue with push operations.
- How to determine the size of larger array?
    - There are 2 approaches:
        - tight strategy : f(N) = n + c (Adding a constant)
        - growth strategy : f(N) = 2*n (Doubling the size)

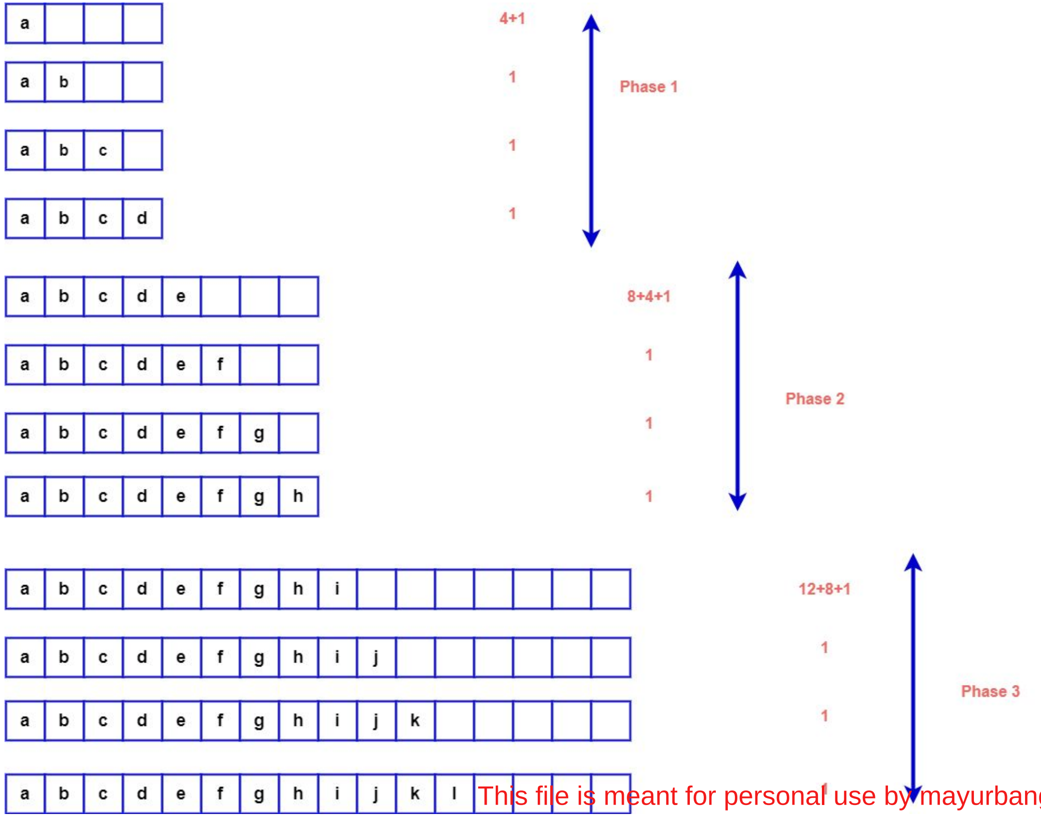
Algorithm push(o):

```
if size() = N then A ← new array of length f(N)
for i ← 0 to N – 1
        A[i] ← S[i]
S ← A; t ← t + 1
S[t] ← o
```

A growable Array-based stack

- **Tight vs growth strategies: Comparison**
  - Let’s compare both strategies by looking at their different cost models:
    - A simple push operation: constant time to add one element in the stack
    - A special push operation:
      1. create an array of size N. ⟶ f(n)
      2. copy N elements, and add one element. ⟶ (n+1)
      3. total cost: f(n)+n+1 units

A growable Array-based stack



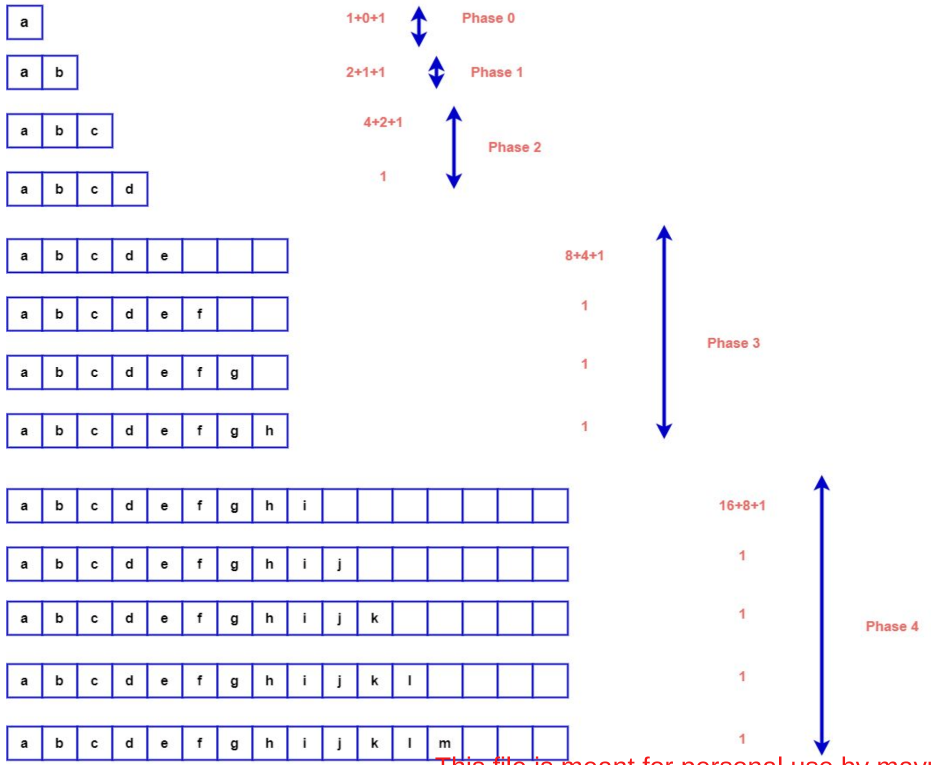
- **Tight Strategy (c = 4)**
  - Initialize an array of size 0.
  - Special push cost when growing array of size n: 2*n + 5

A growable Array-based stack

- **Let’s look at the performance on Tight Strategy**
  - For any arbitrary phase i, the array will have size c*i.
  - Lets calculate the total cost of phase i.
    - Cost creation of array: c*i
    - Cost of copying elements in newly created array: c*(i-1)
    - c * 1 will be the cost of c pushes,
    - Hence, cost of phase i is 2*c*i.
  - For n pushes:
    - Let’s say there are c pushes in each of the phase.
    - n/c phases are required.
    - For n/c phases; total cost will be:

      $= 2*c* (1 + 2 + 3 + ... + n/c) \approx (n^2/c)$

A growable Array stack



- **Growth strategy**
  - Initialize array with size 0.
  - Special push operation cost: 3*n + 1.

A growable Array stack

- **Let’s look at the performance of growth strategy**
  - For any arbitrary phase i the array will have size of 2*i.
  - Let’s explore the total cost of phase i:
    - Cost of creating array: 2^i
    - Cost of copying elements in newly created array: 2^{i-1}
    - Cost of pushes for 2^{i-1} elements: 2^{i-1}
  - Total cost of phase i:$(2^i+2^{i-1}+2^{i-1})$.
  - For n pushes in the reconfigured array,
    - log n phase will be required.
    - Total cost: $2 + 4 + 8 + \ldots + 2^{\log n+1} = 4n - 1$.
  - The growth strategy wins!

Summary

- We explored the concepts of stack along with the involved operations such as push, pop and peek.
- Learnt about the implementation of stack using array.
- We covered the applications of stack in computing paradigm:
  - Virtual machine
  - Time span
  - Growable array stack