



ICS 2105: Data Structures and Algorithms

Stack Data Structure



Acknowledgement

- Notes adapted from:
 - Data Structures Using C++, Second Edition By D.S. Malik
 - tutorialspoint.com
 - studytonight.com



Learning Outcomes

- By the end of this chapter, the learner should be able to:
 - Describe various stack operations
 - Implement a stack as an array
 - Implement a stack as a linked list
 - Describe stack applications.



Introduction

- A stack is a list of homogenous elements in which the **addition** and **deletion** of elements occurs only at one end, called **the top** of the stack.
 - E.g., in a cafeteria, the second tray in a stack of trays can be removed only if the first tray has been removed.



Stack

- A data structure in which the elements are added and removed from one end only.



Stack

- **Stack** is an abstract data type with a bounded (predefined) capacity.
 - It is a simple data structure that allows adding and removing elements in a particular order.
 - Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects

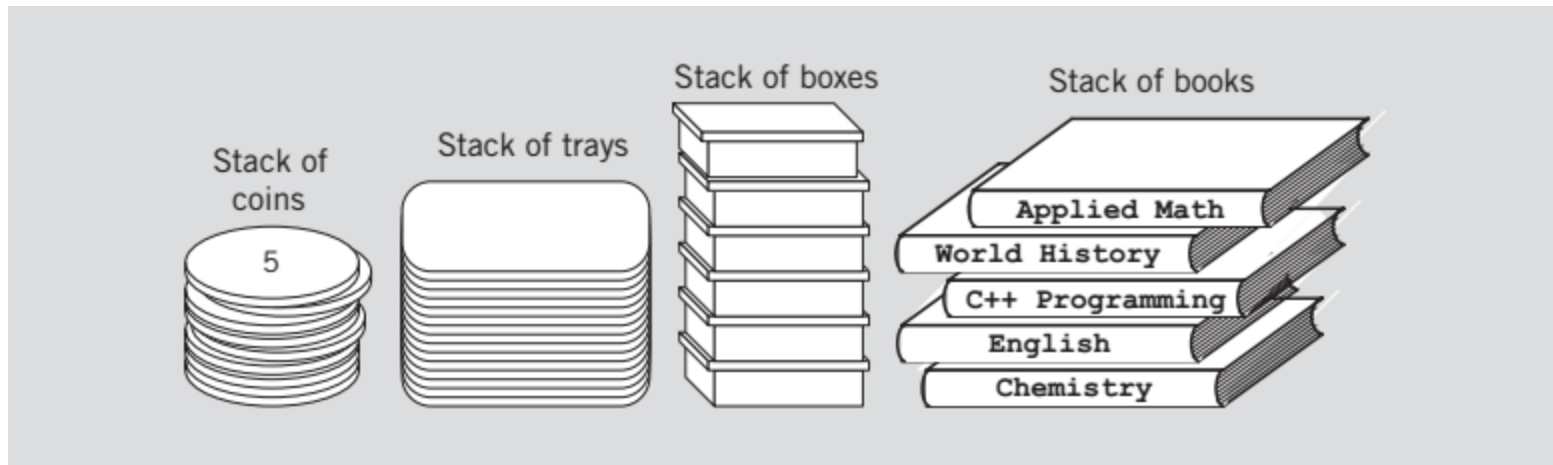


Stack

- It is named stack as it behaves like a real-world stack, for example:
 - A deck of cards or
 - A pile of plates, etc.



Various Examples of Stacks



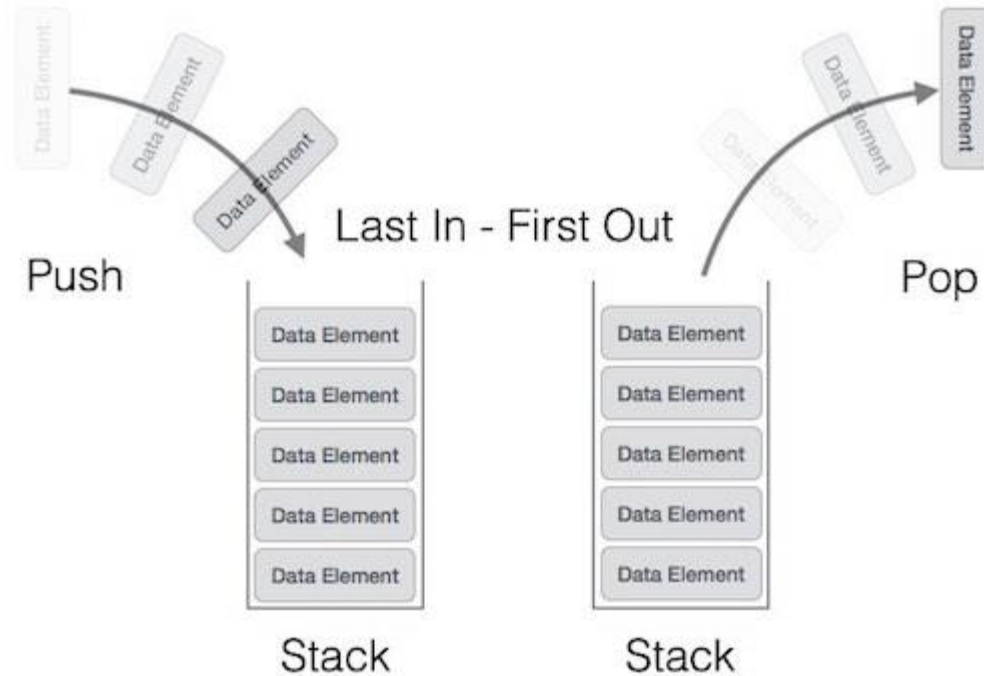


Stack

- This feature makes it LIFO data structure. LIFO stands for Last-in-first-out.
 - Here, the element which is placed (inserted or added) last, is accessed first.
 - In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



Stack Representation





Stack Implementation

- A stack can be implemented by means of:
 - Array
 - Structure
 - Pointer, and
 - Linked List.



Stack Implementation

- Stack can either be a fixed size one or it may have a sense of dynamic resizing.
- In this unit, we are going to implement stack using arrays, which makes it a fixed size stack implementation.



Basic Stack Operations

- Stack operations may involve initializing the stack, using it and then de-initializing it.
- Apart from these basic operations, a stack is used for the following two primary operations:
 - **push()** – Pushing (storing) an element on the stack.
 - **pop()** – Removing (accessing) an element from the stack.



Basic Stack Operations

- To use a stack efficiently, we need to check the status of stack as well.
- For the same purpose, the following functionality is added to stacks –
 - **peek()** – get the top data element of the stack, without removing it.
 - **isFull()** – check if stack is full.
 - **isEmpty()** – check if stack is empty.



Basic Stack Operations

- At all times, we maintain a pointer to the last PUSHed data on the stack.
- As this pointer always represents the top of the stack, hence named **top**.
- The **top** pointer provides top value of the stack without actually removing it.

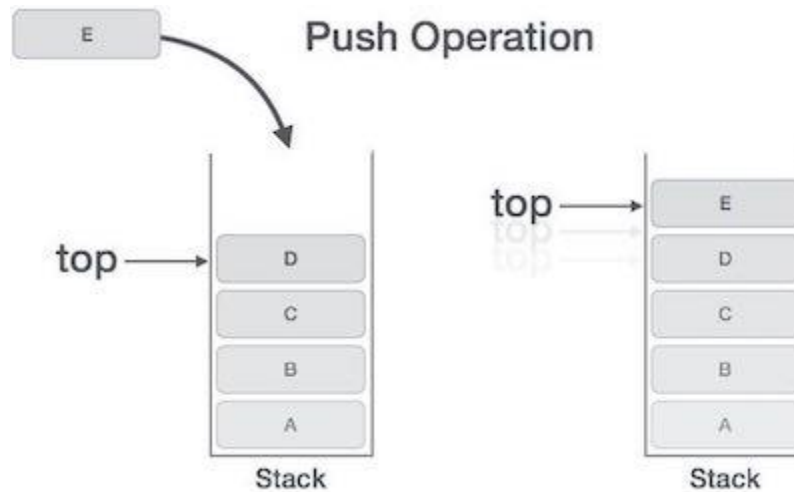


Push Operation

- The process of putting a new data element onto stack is known as a Push Operation.
- Push operation involves a series of steps –
 - **Step 1** – Checks if the stack is full.
 - **Step 2** – If the stack is full, produces an error and exit.
 - **Step 3** – If the stack is not full, increments **top** to point next empty space.
 - **Step 4** – Adds data element to the stack location, where top is pointing.
 - **Step 5** – Returns success.



Push Operation





Pop Operation

- Accessing the content while removing it from the stack, is known as a Pop Operation.
- In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value.



Pop Operation

- But in linked-list implementation, `pop()` actually removes data element and deallocates memory space.

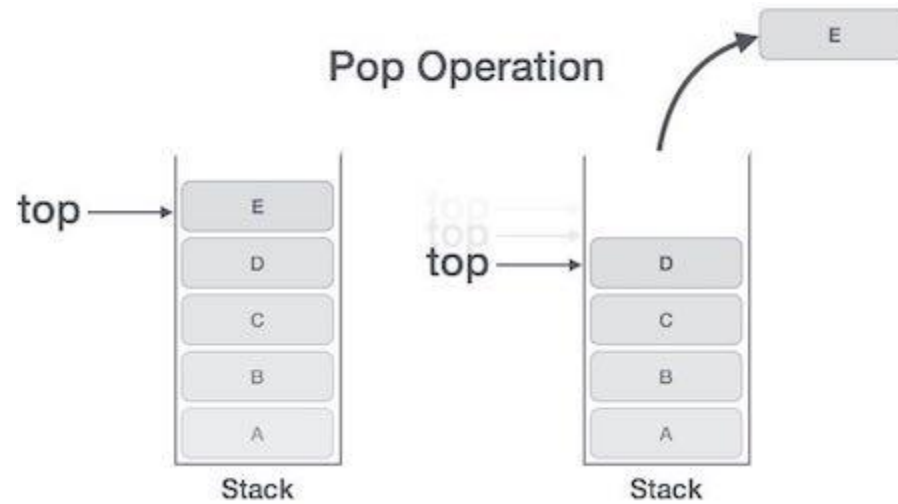


Pop Operation

- A Pop operation may involve the following steps –
 - **Step 1** – Checks if the stack is empty.
 - **Step 2** – If the stack is empty, produces an error and exit.
 - **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
 - **Step 4** – Decreases the value of top by 1.
 - **Step 5** – Returns success.



Pop Operation





Applications of Stack

- The simplest application of a stack is to reverse a word.
 - You push a given word to stack - letter by letter - and then pop letters from the stack.
- There are other uses also like:
 - Parsing
 - Expression Conversion(Infix to Postfix, Postfix to Prefix etc)



Expression Parsing

- The way to write arithmetic expression is known as a **notation**.
- An arithmetic expression can be written in three different but **equivalent notations**, i.e., without changing the essence or output of an expression.



Expression Parsing

- These notations are:
 - Infix Notation
 - Prefix (Polish) Notation
 - Postfix (Reverse-Polish) Notation
- These notations are named as how they use operator in expression



Expression Parsing : Infix Notation

- We write expression in **infix** notation, e.g. $a - b + c$, where operators are used **in-between** operands.
- It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices.



Expression Parsing : Infix Notation

- An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.



Prefix Notation

- In this notation, operator is **prefixed** to operands, i.e. operator is written ahead of operands.
 - E.g., **+ab**. This is equivalent to its infix notation **a + b**.
- Prefix notation is also known as **Polish Notation**.



Postfix Notation

- This notation style is known as **Reversed Polish Notation**.
- In this notation style, the operator is **postfixed** to the operands i.e., the operator is written after the operands.
 - E.g., **ab+**. This is equivalent to its infix notation **a + b**.



Difference In all Three Notations

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	$a + b$	$+ a b$	$a b +$
2	$(a + b) * c$	$* + a b c$	$a b + c *$
3	$a * (b + c)$	$* a + b c$	$a b c + *$
4	$a / b + c / d$	$+ / a b / c d$	$a b / c d / +$
5	$(a + b) * (c + d)$	$* + a b + c d$	$a b + c d + *$
6	$((a + b) * c) - d$	$- * + a b c d$	$a b + c * d -$



Parsing Expressions

- It is not a very efficient way to design an algorithm or program to parse infix notations.
- Instead, these infix notations **are first converted** into either postfix or prefix notations and then computed.
- To parse any arithmetic expression, we need to take care of operator precedence and associativity also.



Parsing Expressions : Precedence

- When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others.
 - For example $a + b * c = a + (b * c)$
- As multiplication operation has precedence over addition, $b * c$ will be evaluated first.



Parsing Expressions: Associativity

- Associativity describes the rule where operators with the same precedence appear in an expression.
 - E.g., in expression $a + b - c$, both $+$ and $-$ have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators.
 - Here, both $+$ and $-$ are left associative, so the expression will be evaluated as $(a + b) - c$.



Parsing Expressions: Associativity

- Precedence and associativity determines the order of evaluation of an expression.



Postfix Evaluation Algorithm

- Step 1 – scan the expression from left to right
- Step 2 – if it is an operand push it to stack
- Step 3 – if it is an operator pull operand from stack and perform operation
- Step 4 – store the output of step 3, back to stack
- Step 5 – scan the expression until all operands are consumed
- Step 6 – pop the stack and perform operation



Implementation of Stack Data Structure

- Stack can be easily implemented using an Array or a Linked List.
- Arrays are **quick**, but are **limited in size** and Linked List **requires overhead** to allocate, link, unlink, and deallocate, but is not limited in size.
- Here we will implement Stack using array



End of lesson