

Unit – I : Elementary Data Structures

- Data structures
- Asymptotic complexity
- Abstract data type:
 - Array,
 - **Stacks,**
 - **Queues,**
 - **Linked Lists, and their applications.**

Today's Agenda

Stack

```
graph TD; A[Stack] --> B[Stack Operations]; B --> C[Tower of Hanoi]; C --> D[Expression Parsing];
```

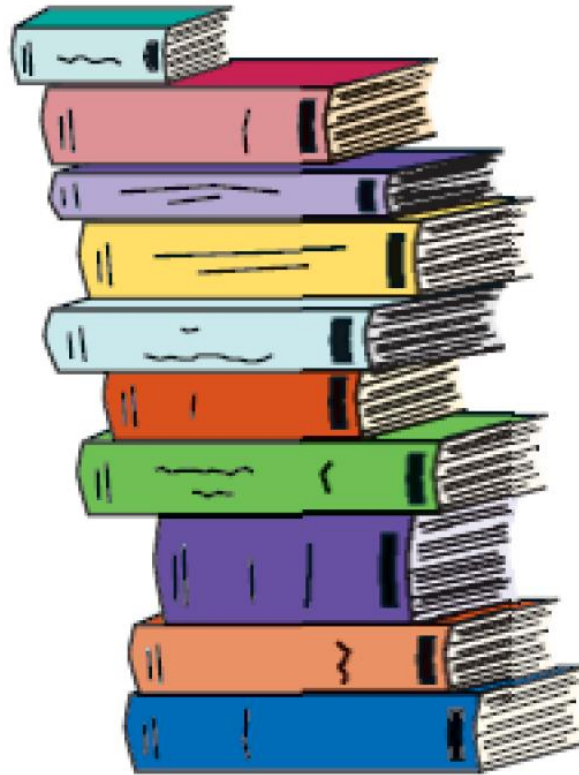
Stack Operations

Tower of Hanoi

Expression Parsing

What is a Stack

- Stack of Books



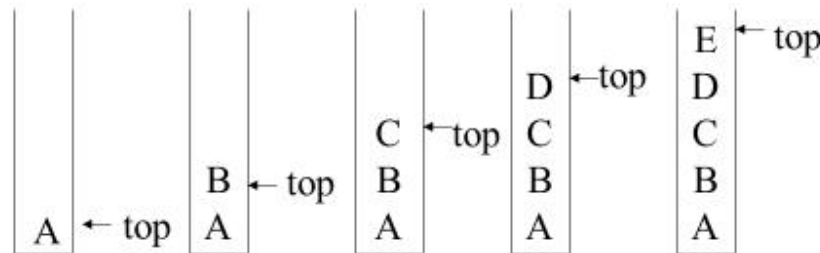
What is a stack?

- Stores a set of elements in a particular order
- Stack principle: **LAST IN FIRST OUT**
- = **LIFO**
- It means: the last element inserted is the first one to be removed
- Example
- Which is the first element to pick up?



Stacks

- What is a Stack?
 - A stack is a data structure of ordered items such that items can be inserted and removed only at one end.



Stacks

- What can we do with a stack?
 - push - place an item on the stack
 - peek - Look at the item on top of the stack, but do not remove it
 - pop - Look at the item on top of the stack and remove it

Stacks

- A stack is a LIFO (Last-In/First-Out) data structure
- A stack is sometimes also called a pushdown store.
- What are some applications of stacks?
 - Program execution
 - Parsing
 - Evaluating postfix expressions

Stack Applications

- Real life
 - Pile of books
 - Plate trays
- More applications related to computer science
 - Program execution stack (read more from your text)
 - Evaluating expressions

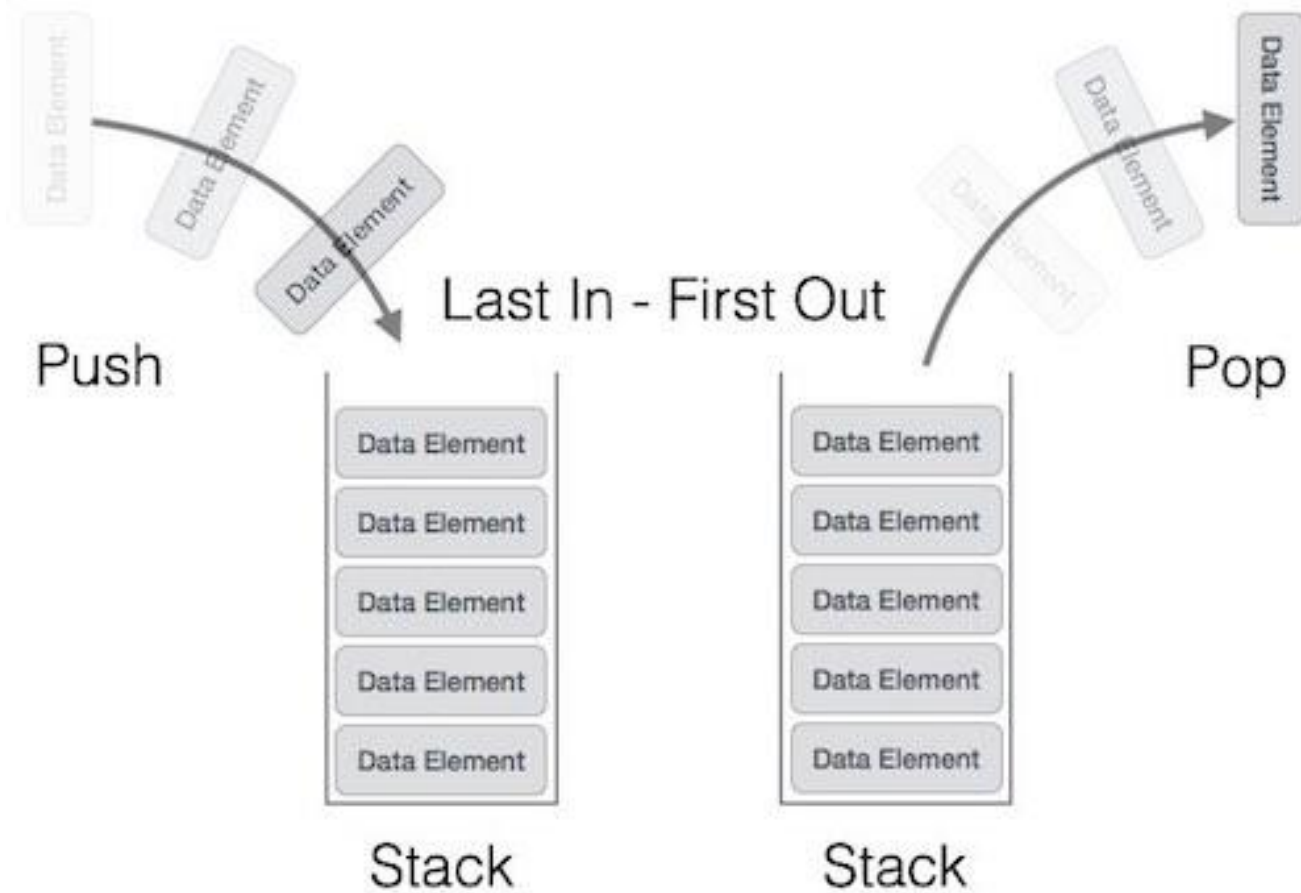
Reversing a Word

- We can use a stack to reverse the letters in a word.
- How?

Reversing a Word

- Read each letter in the word and push it onto the stack
- When you reach the end of the word, pop the letters off the stack and print them out.

Stack Representation



Stack Representation

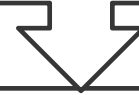
- A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing.
- Most cases (Default) to implement stack using arrays, which makes it a fixed size stack implementation.

Today's Agenda

Stack



Stack Operations



Tower of Hanoi



Expression Parsing

Basic Operations

- Stack operations may involve initializing the stack, using it and then de-initializing it.
- Apart from these basic stuffs, 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 Operations

- When data is PUSHed onto stack.
- 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 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.

peek()

Algorithm of peek() function –

```
begin procedure peek  
  
    return stack[top]  
  
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {  
    return stack[top];  
}
```

isfull()

Algorithm of isfull() function –

```
begin procedure isfull
    if top equals to MAXSIZE
        return true
    else
        return false
    endif
end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
    if(top == MAXSIZE)
        return true;
    else
        return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty  
  
    if top less than 1  
        return true  
    else  
        return false  
    endif  
  
end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

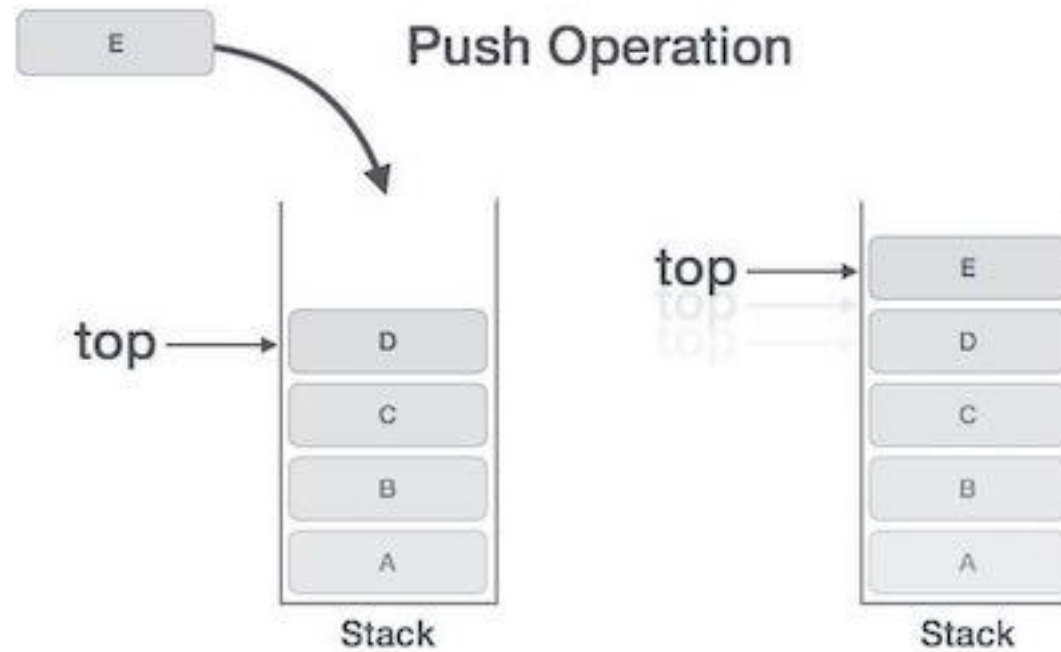
Example

```
bool isempty() {  
    if(top == -1)  
        return true;  
    else  
        return false;  
}
```

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



Algorithm for PUSH Operation

- begin procedure push: stack, data
- if stack is full
- return null
- endif
- $\text{top} \leftarrow \text{top} + 1$
- $\text{stack}[\text{top}] \leftarrow \text{data}$
- end procedure

C Code

```
void push(int data) {  
    if(!isFull()) {  
        top = top + 1;  
        stack[top] = data;  
    } else {  
        printf("Could not insert data, Stack is full.\n");  
    }  
}
```

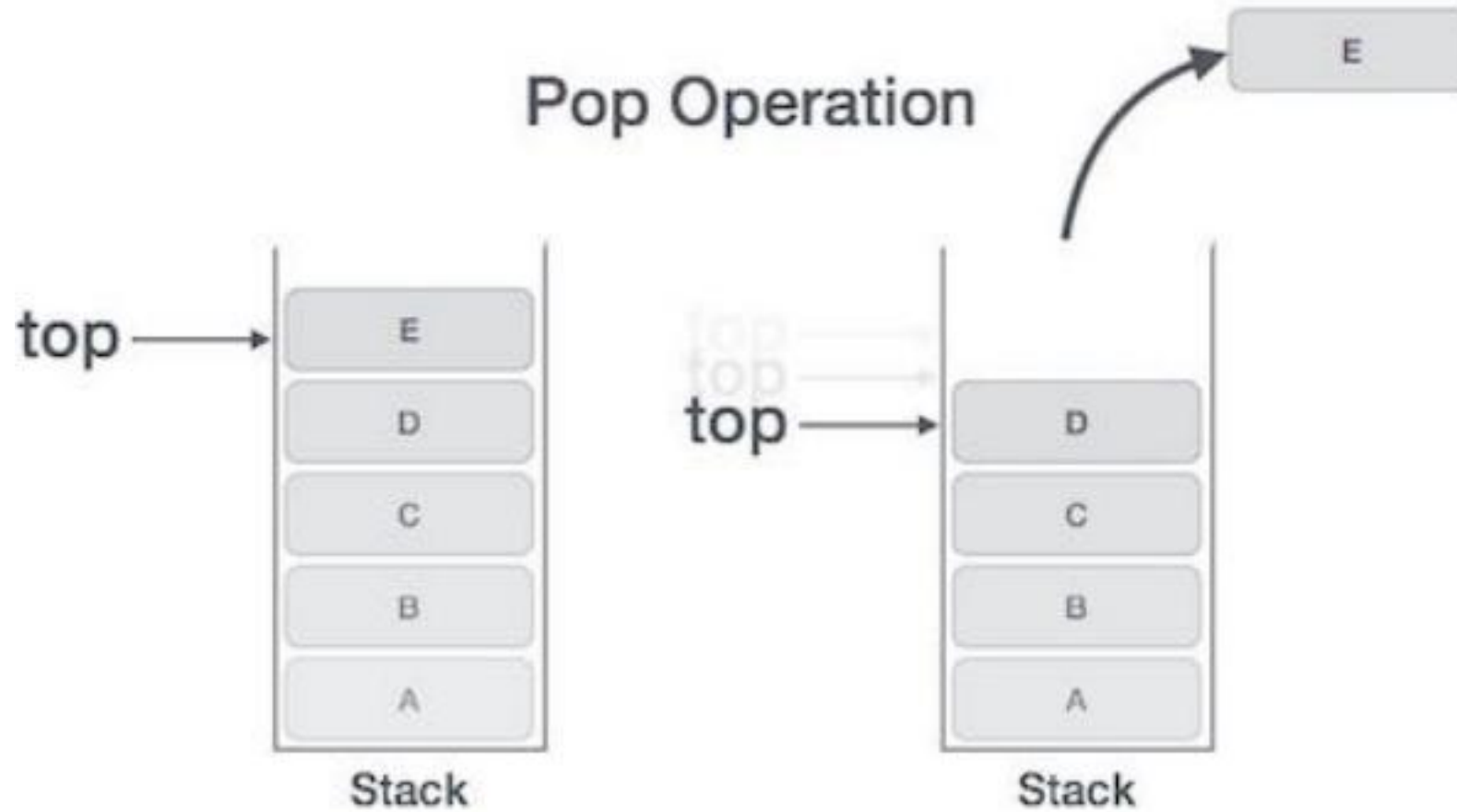
Pop Operation

- 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.
- 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



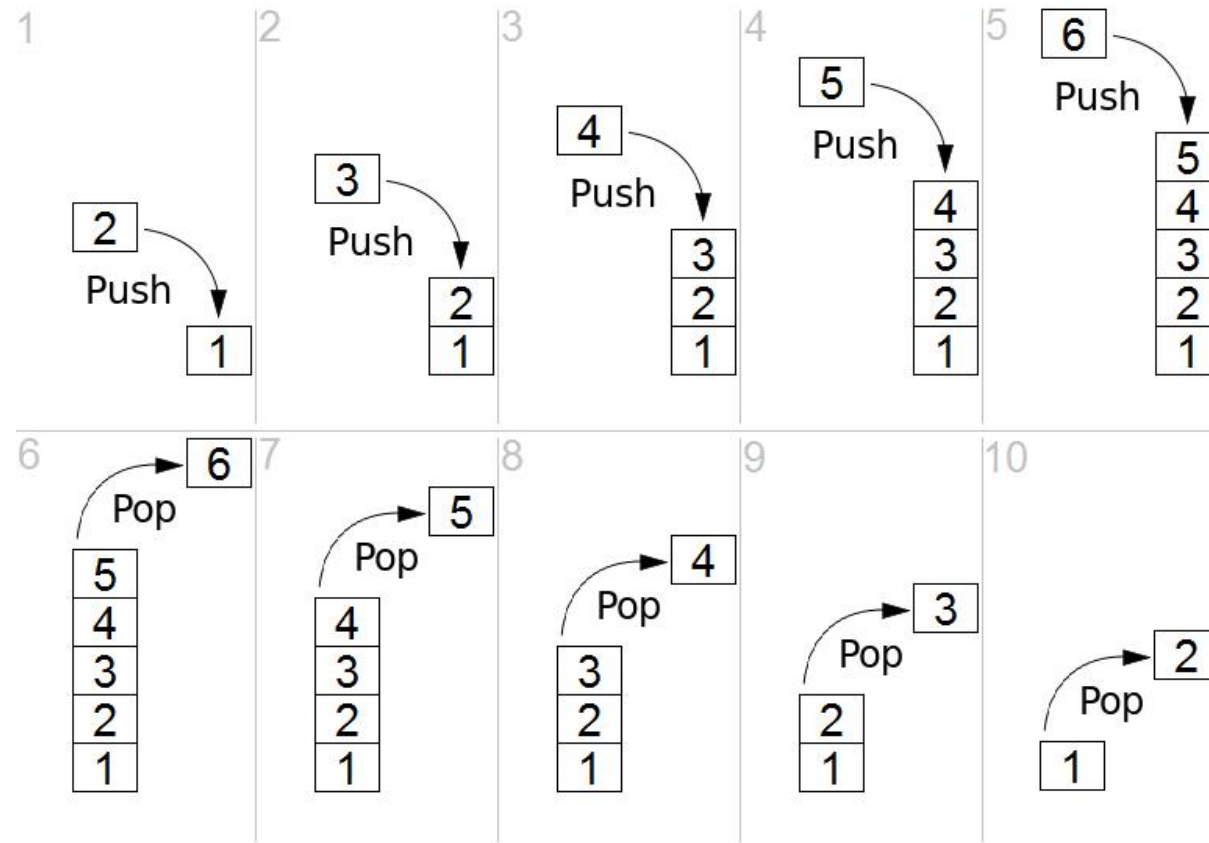
Pop Operation

- begin procedure pop: stack
- if stack is empty
- return null
- endif
- data \leftarrow stack[top]
- top \leftarrow top - 1
- return data
- end procedure

Pop Code in C

```
int pop(int data) {  
    if(!isempty()) {  
        data = stack[top];  
        top = top - 1;  
        return data;  
    } else {  
        printf("Could not retrieve data, Stack is empty.\n");  
    }  
}
```

PUSH and POP



Practical Applications - Stack

- Expression Processing
 - Often used to parse and evaluate various forms of expression statements.
 - A stack to evaluate a mathematical expression that is written using reverse Polish notation.
- Backtracking: browser backtrack
 - Back button in your browser.
 - Click on various links in the page, the browser adds each link to a stack in order to maintain the order in which they were visited.
 - The user clicks on the back button, the most recently added URL is popped off the stack and then revisited.

Practical Applications - Queue

- Order Processing

- It ensures that orders for a product are fulfilled in the order in that they were received by the system.

- Messaging

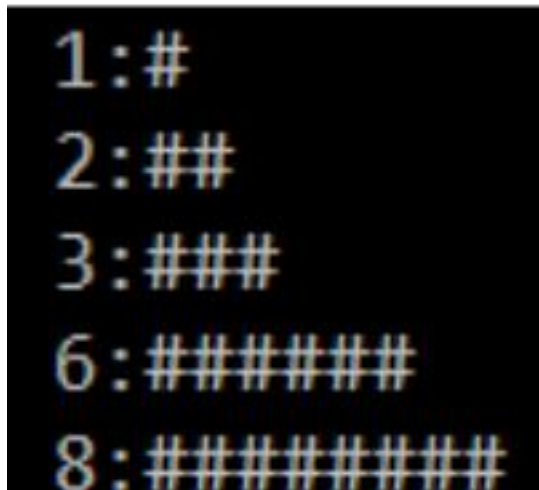
- SMS messenger app and entering each message, you want to make sure that each of those messages is sent in the order they were written.
- The messenger app might use a queue to maintain each message and make sure that they are sent in the order they were placed into the queue.

A Simple Challenge – Create a bar chart

```
1: #  
2: ##  
3: ###  
6: #####  
8: #####
```


SOLUTION TO CHALLENGE: CREATE A BAR CHART

```
void showBarChart(void);
void showOneBar(int barLength);
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    showBarChart();
}
```



```
1: #
2: ##
3: ###
6: #####
8: #####
```

```
void showBarChart(void)
{
    showOneBar(1);
    showOneBar(2);
    showOneBar(3);
    showOneBar(6);
    showOneBar(8);
}

void showOneBar(int barLength)
{
    printf("%3d:", barLength);
    for(int i=0; i<barLength; i++)
    {
        putchar('#');
    }
    putchar('\n');
}
```

Challenge Continued

- Generate the random data set that's an improvement over the previous program.
 - The data set should consist of integer numbers in a given range.
 - For example, between zero and 50
-
- Clue:
 - `#include <time.h>`
 - `rand()`

Today's Agenda

Stack



Stack Operations



Tower of Hanoi

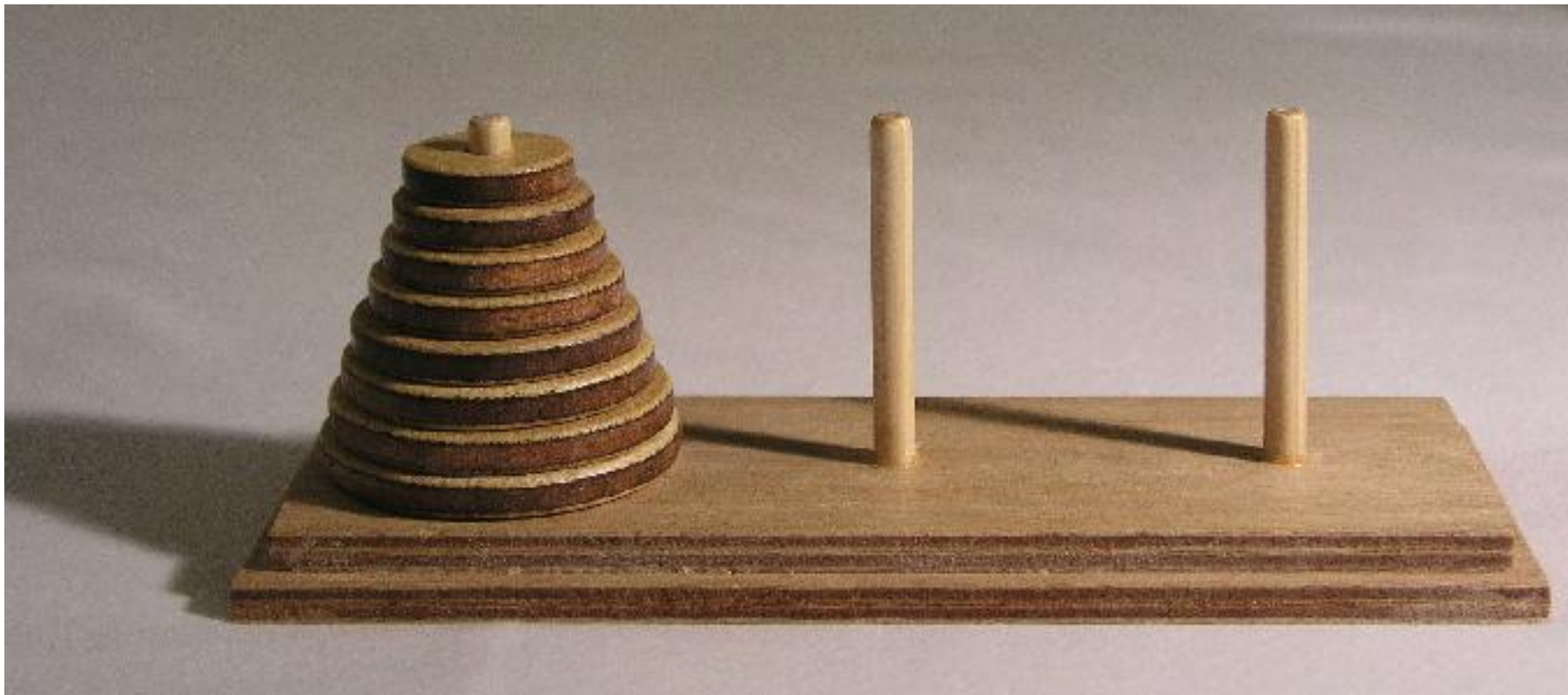


Expression Parsing

Towers of Hanoi

- One of the most interesting applications of stacks can be found in solving a puzzle called Tower of Hanoi.
- According to an old story, the existence of the universe is calculated in terms of the time taken by a number of monks, who are working all the time, to move 64 disks from one pole to another. But there are some rules about how this should be done, which are:
 - You can move only one disk at a time.
 - For temporary storage, a third pole may be used.
 - You cannot place a disk of larger diameter on a disk of smaller diameter

The Towers of Hanoi



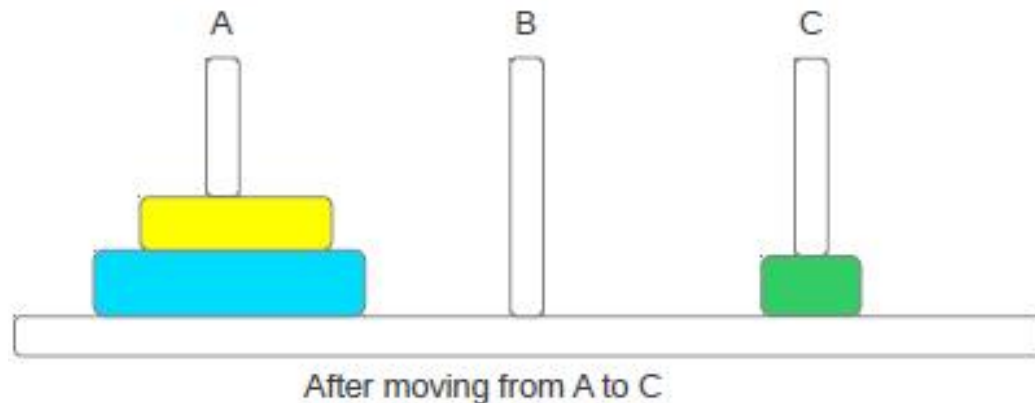
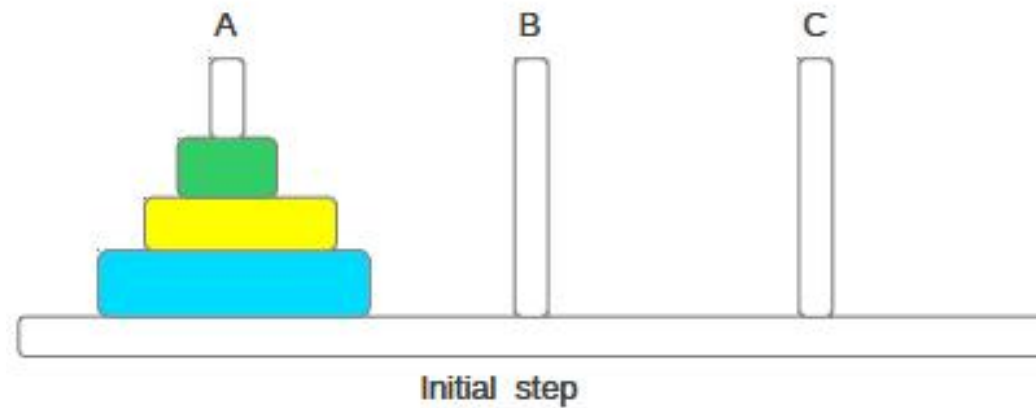
The Towers of Hanoi

A Stack-based Application

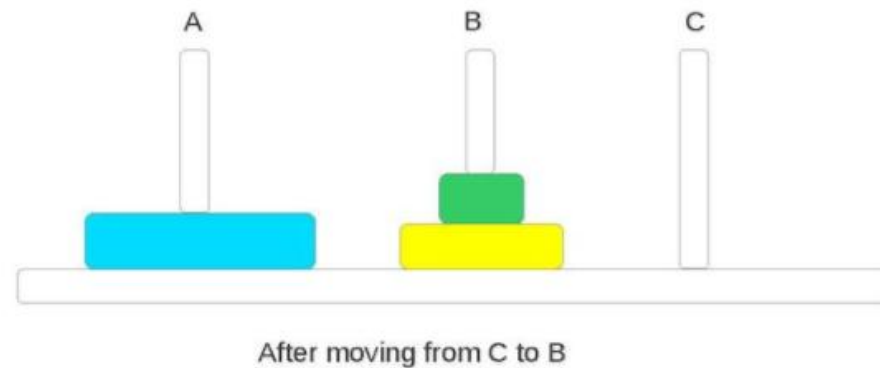
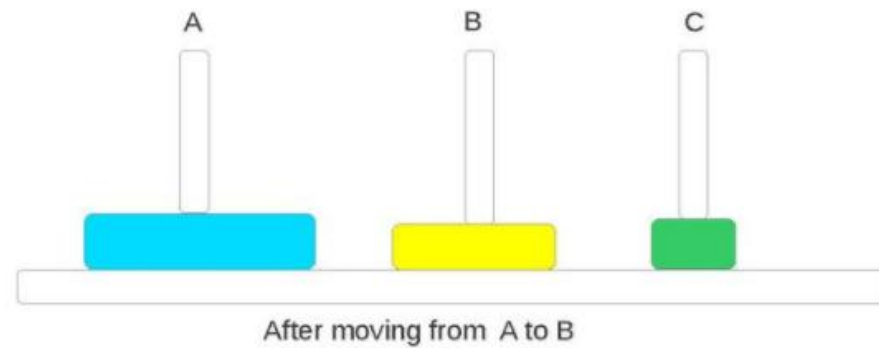
- GIVEN: three poles
- a set of discs on the first pole, discs of different sizes, the smallest discs at the top
- GOAL: move all the discs from the left pole to the right one.
- CONDITIONS: only one disc may be moved at a time.
- A disc can be placed either on an empty pole or on top of a larger disc.



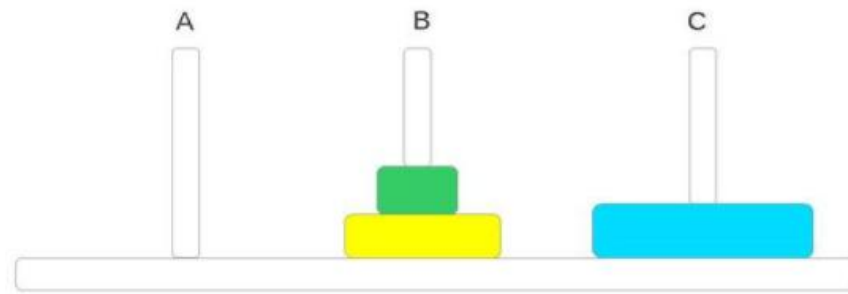
Here we assume that A is first tower, B is second tower & C is third tower.



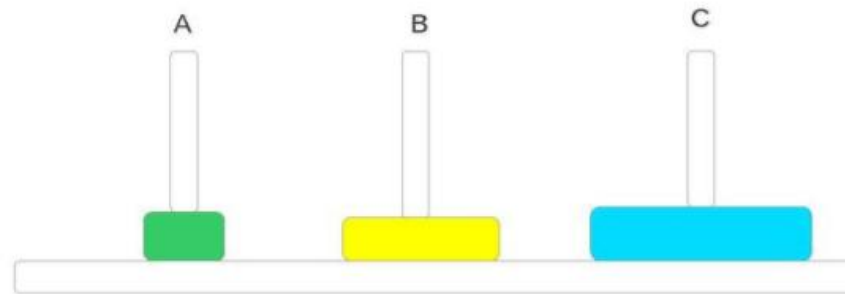
Step 3



Step 3

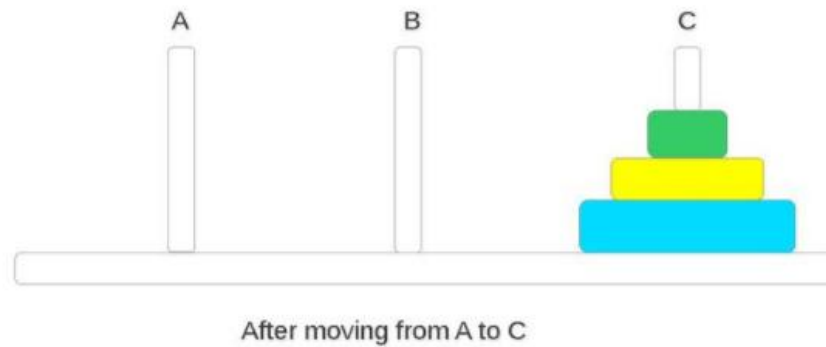
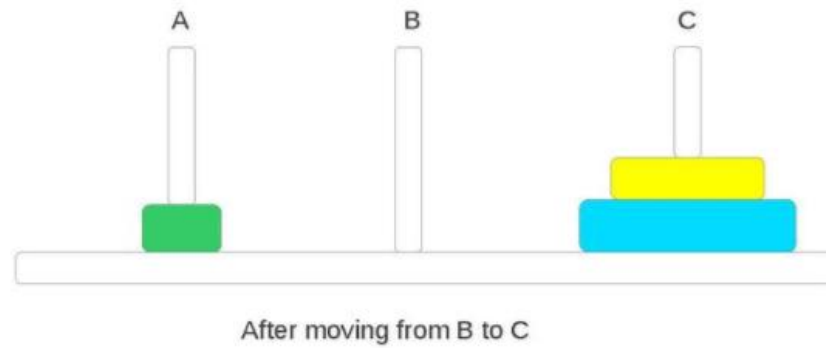


After moving from A to C



After moving from B to A

Step 4

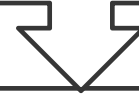


Today's Agenda

Stack



Stack Operations



Tower of Hanoi



Expression Parsing

Notation

- 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. These notations are –
 - Infix Notation
 - Prefix (Polish) Notation
 - Postfix (Reverse-Polish) Notation

All 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.

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

$$a + b * c \quad \Rightarrow \quad a + (b * c)$$

- As multiplication operation has precedence over addition, $b * c$ will be evaluated first

operator precedence and associativity

Sr. No.	Operator	Precedence	Associativity
1	Exponentiation \wedge	Highest	Right Associative
2	Multiplication ($*$) & Division ($/$)	Second Highest	Left Associative
3	Addition ($+$) & Subtraction ($-$)	Lowest	Left Associative

Token	Operator	Precedence ¹	Associativity
() [] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement ²	16	left-to-right
-- ++ ! - - + & * sizeof	decrement, increment ³ logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	Left-to-right

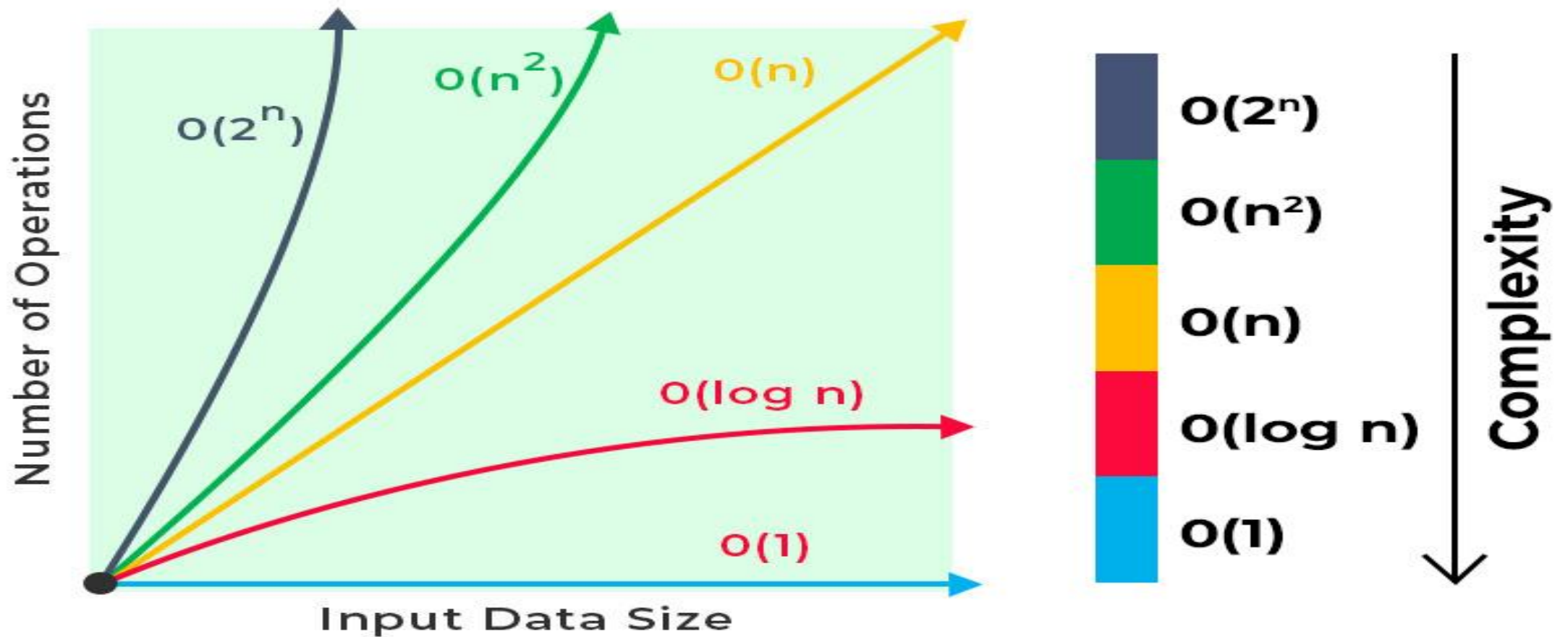
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
ξξ	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= ♠	assignment	2	right-to-left
,	comma	1	left-to-right

Time Complexity

- To access or edit any element stored in a stack, the time taken is $O(N)$ as to reach any specific element, all the elements before it has to be removed.
- The searching operation also takes a total time of $O(N)$, as reaching any specific element isn't possible without popping the elements stored before it.
- Operations like insertion or deletion in a stack take constant time i.e. $O(1)$.

Basic Chart



Notations

$O(1)$ Constant

$O(\log N)$ Logarithmic

$O(N)$ Linear time

$O(N * \log N)$ Log linear

$O(N^2)$ Quadratic

$O(N^3)$ Cubic

$O(2^N)$ Exponential

$O(N!)$ Factorial