
# Abstract Data Type - Stacks

Tanvir Ahmed, PhD

Department of Computer Science,

University of Central Florida

# Content

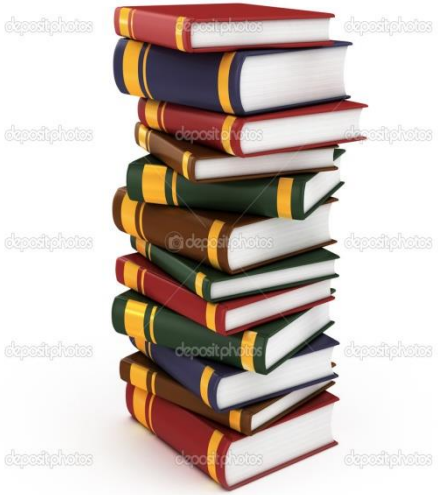
- Abstract Data Type (ADT)
- Stack
  - Applications of Stack
  - Stack Operations and Implementation
  - Examples and Simulation
  - Expression Evaluation, Infix to Postfix

# Abstract Data Type (ADT)

- Can you give me some examples of data type?
- What happens when you declare a variable of a data type? (e.g., int a;)
- What can you do with an integer variable?
- What is an array?
- ADT is a data structure and a set of operations which can be performed on it
  - Example: Stack, Queue, Linked List

# What is Stack?

- A stack is a Last In, First Out (LIFO) ADT
- Anything added goes on the top
- Anything removed from the stack is taken from the “top”
- Things are removed in the reverse order from that in which they were inserted
- Analogy:
  - Stack of plates
  - Which one will you remove first?
  - Another example pile of books
  - Can you think other examples?

# Why stack?

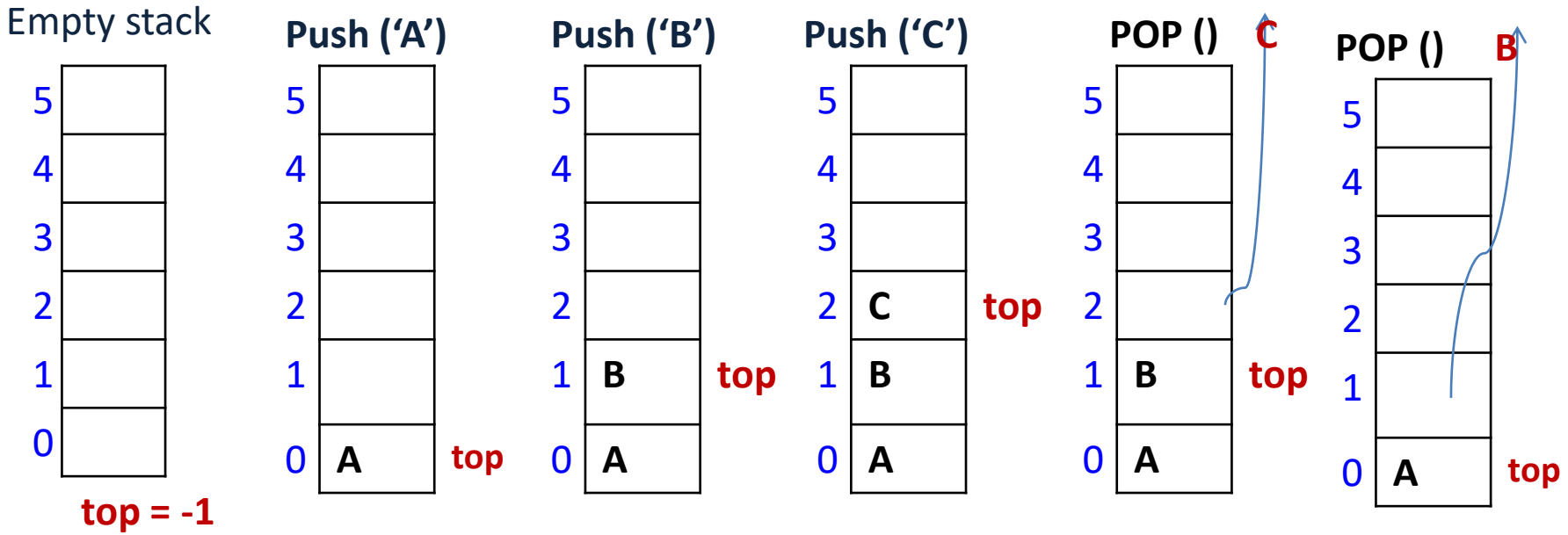
- Variety of applications of stack:
  - Program execution (e.g., function call, recursion)
    - F1() calls F2(), then F2() calls F3(). Where F3() will return?
  - Evaluating expressions (e.g., a / b - c + d * e - a * c)
  - Undo operation (back button of your browser?)
  - Searching path in a graph
  - Validating the balance of parenthesis (we will see example)

# Stack operations

- Basic stack operations
  - **Push :** Add a new item
  - **Pop :** get and remove the item that was added most recently
  - **IsEmpty:** Determine whether a stack is empty
  - **IsFull:** Determine whether a stack is full
  - **Peek:** Get the most recently added item without deleting it

# Example of stack operations

- Max_Size = 6, Stack[Max_Size], top = -1
- Push(char x): Stack[++top] = x,
- POP(): return Stack[top--], Peek(): return Stack[top]




- Discussion: POP(), POP()
- Stack Overflow? Or Stack Underflow?

# Stack Implementations

- There can be various ways stacks can be implemented.
  - Using global top variables and global array
  - Using structure and array and variables within the structure
    - The use local variable of structure
    - This can facilitate creating many stacks
  - Using linked list
- Two array based stack implementation code will be uploaded in webcourses.
- We will discuss both of them
- Stack also can be implemented in linked list that we will cover in another lecture after learning linked list

# Let us implement a simple Array based stack!

- We will write code in class!
- The code will be available in web courses too!

# Example Stack Implementation- global variable

```
void push(int x)
 {
     if( top >= Max_Size-1)
        printf("Stack Overflow“);
    else
   {
       Stack[++top] = x;
       printf(“Inserted“);
   }
}
```

```
int pop()
{
    if(top < 0)
   {
       printf("Stack Underflow“);
       return -9999;
    }
    else
       return Stack[top--];
}
```

```
int isEmpty()
{
    if(top < 0)
   {
       printf( "Stack is empty“);
       return 1;
   }
   else
   {
       printf( "Stack is not empty“);
       return 0;
    }
}
```

```
int isFull()
{
      what do you
       think?
}
```

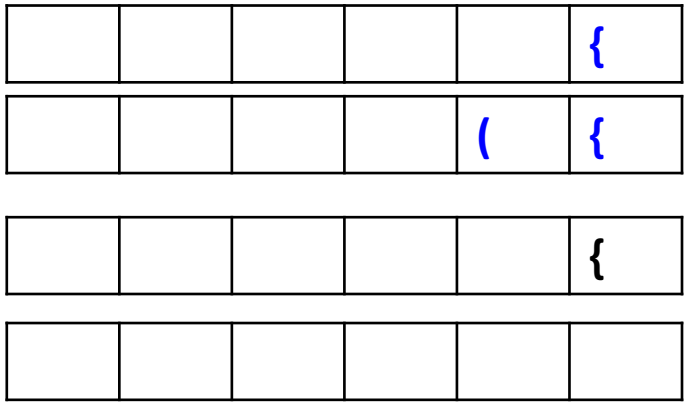
```
int isFull()
{
    if ( top >= Max_Size-1)
   {
          printf(“Full”);
          return 1;
    }
    else
    {
         printf(“Full”);
         return 0;
}
```

# Checking the balance of parenthesis

- Is it balanced?: ( { }[ ] )
- How about this one?: ( ( { } [ ( ) ) ] )
- Simple counting is not enough to check balance
- You can do it with a stack.
- Read the string char by char left to right,
  - If you see a (, [, or {, push it on the stack
  - If you see a ), ], or }, pop the stack and check whether you got the corresponding (, [, or {
  - When you reach the end, check that the stack is empty. If it is empty, it is balanced.

# Simulation

- Input: { ( ) }
- Read {, so push (‘ {‘)
- Read (, so push (‘(‘).
- Read ), so pop. popped item is ‘(‘ which matches ).
- Read }, so pop; popped item is { which matches }.
- End of string; stack is empty, so the string is valid.




**Activity:** Using Stack, check whether the following is balanced or not : “{ ( ) } ) ”

-Assume a stack of Max_Size = 6

-Draw the stack for each step and show the value of “top”

-If any decision is made (valid/invalid), then write the reason

# Exercise

Write a program that will take an expression with parenthesis as input and return whether the parenthesis in the expression is balanced or not.

- Use stack for the solution.

# Expression Evaluation

- Infix Expression:
  - **A + B * C**
  - **(A + B) *C – D * E**
  - Easily understandable by human
  - Hard to parse in a computer program and difficult to evaluate by a program
- Postfix:
  - **Operators** come after the **operand**
  - ***A + B → A B +***
  - **A + B * C → *A B C * +***
  - Operators are ALWAYS in correct evaluation order.

# Infix to Postfix

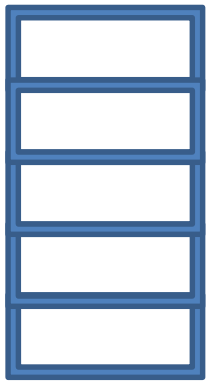
- Consider we have two elements:
  1. An empty expression string, called Postfix
  2. An empty operator stack

# Converting Infix Exp. to Postfix Exp.

- Main Steps (Assuming the expression is valid):
- Start reading the Infix expression from left to right
- **If we encounter an operand** we will append it to the Postfix string, **[Operand doesn't go to stack]**
- **If we encounter an operator let's say current operator:**
- The current operator is compared to the operator that is at the top of the stack
- If the priority of the current operator is **HIGHER** than the priority of the operator at the top of the stack, then the current operator is pushed at the top of the stack.
- If the priority of the current operator is **same or lower** than the priority of the operator at the top of the stack, we keep POPing operators from the stack until the priority of the current operator is higher than the operator at the top of the stack. POPed operators are appended to the Postfix in the order they are POPed. **Finally,** the current operator is PUSHED to Stack.
- **If we encounter an open parenthesis:** immediately pushed in the stack
- **If we encounter a closing parenthesis:** POP until an open parenthesis (inclusive) found in the stack. Append the popped items to the Postfix except the parenthesis.
- ***Priority of the operators is as follows (high to low):***
  - ***1. power ^***
  - ***2. * / %***
  - ***3. + -***

# 4 * 2 + 3 + 8 / 4

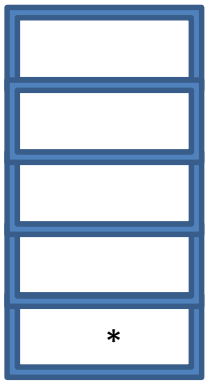
- 4 is an operand:
  - Append it to Postfix
- Stack remains empty



- Postfix Exp.: 4

# 4 * 2 + 3 + 8 / 4

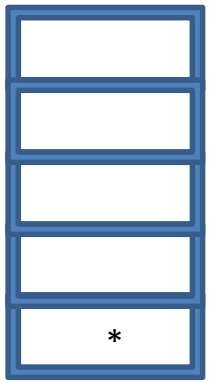
- * is an operator:
  - It is the first operator read
  - It is pushed in the stack




- Postfix Exp.: 4

# 4 * 2 + 3 + 8 / 4

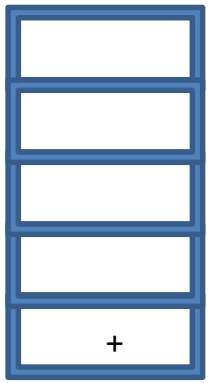
- 2 is an operand:
  - Append it to Postfix




- Postfix Exp.: 4 2

# 4 * 2 + 3 + 8 / 4

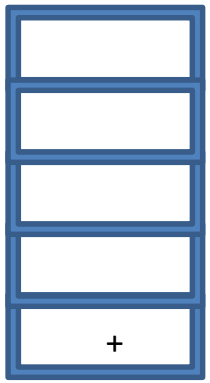
- + is an operator:
  - The stack is not empty
  - '+' has lower priority compared to the operator at the top of the stack (*)
  - * has to POPed out and append to the Postfix
  - + is pushed in the stack




- Postfix Exp.: 4 2 *

# 4 * 2 + **3** + 8 / 4

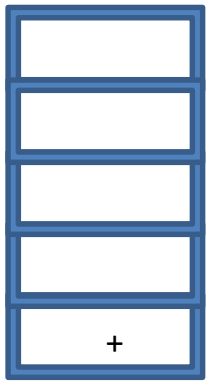
- 3 is an operand:
  - Append it to Postfix




- Postfix Exp.: 4 2 * 3

# 4 * 2 + 3 + 8 / 4

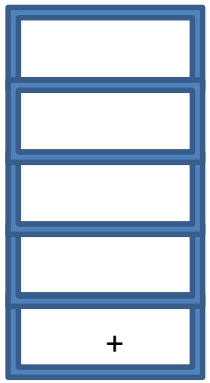
- + is an operator:
  - The stack is not empty
  - '+' has the same priority compared to the operator at the top of the stack (+)
  - + has to POPed out and appended to Postfix
  - + (current operator) is pushed in the stack




- Postfix Exp.: 4 2 * 3 +

# 4 * 2 + 3 + **8** / 4

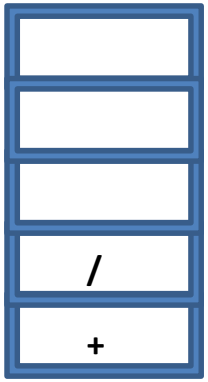
- 8 is an operand:
  - Append it to Postfix




- Postfix Exp.: 4 2 * 3 + 8

# 4 * 2 + 3 + 8 / 4

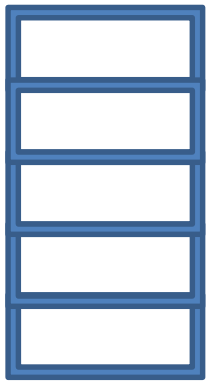
- / is an operator:
  - The stack is not empty
  - '/' has higher priority compared to the operator at the top of the stack (+)
  - * is pushed on top of the stack
  - **Note:** + is not POPed because it has lower priority




- Postfix Exp.: 4 2 * 3 + 8

# 4 * 2 + 3 + 8 / **4**

- 4 is an operand:
  - Append it to Postfix
  - We finished reading the infix Exp.
  - We have to POP everything out of the stack
  - Append all the popped items to the Infix in the order they are popped

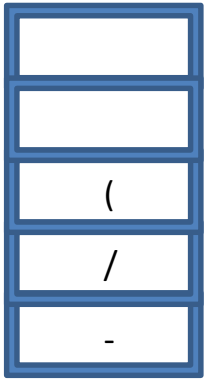


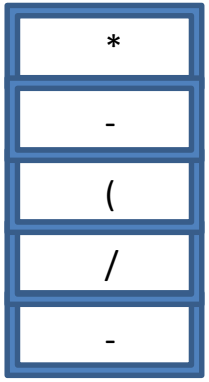
- Postfix Exp.: 4 2 * 3 + 8 4 / +

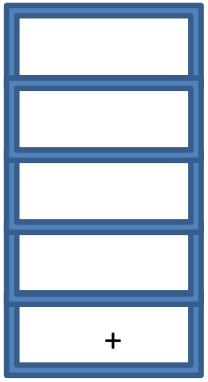
# Example 2: ( ( A + B ) — C * ( D / E ) ) + F

| Step | Symbol | Stack | Postfix |
|---|---|---|---|
| 1 | ( | ( | |
| 2 | ( | ( ( | |
| 3 | A | ( ( | A |
| 4 | + | ( ( + | A |
| 5 | B | ( ( + | A B |
| 6 | ) | ( | A B + |
| 7 | - | ( - | A B + |
| 8 | C | ( - | A B + C |
| 9 | * | ( - * | A B + C |
| 10 | ( | ( - * ( | A B + C |
| 11 | D | ( - * ( | A B + C D |
| 12 | / | ( - * ( / | A B + C D |
| 13 | E | ( - * ( / | A B + C D E |
| 14 | ) | ( - * | A B + C D E / |
| 15 | ) | | A B + C D E / * - |
| 16 | + | + | A B + C D E / * - |
| 17 | F | + | A B + C D E / * - F |
| 18 | | | A B + C D E / * - F + |

# 7 – 12 / (5 – 2 * 1) + 2



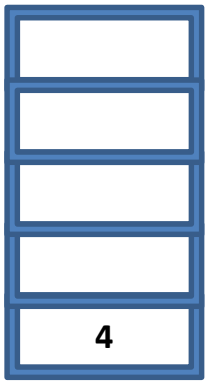

Postfix Exp.: 7 12 5 2 1 * - / - 2 +

# Evaluation from Postfix

- Consider: An empty stack
- Read the expression from left to right
- **If we encounter an operand:** PUSH the operand to the stack
- **If we encounter an operator:**
  - Pop the two operands from the stack,
  - Evaluate two operands with the operator
  - Push the evaluation into the stack
- Repeat it till the end of the expression.

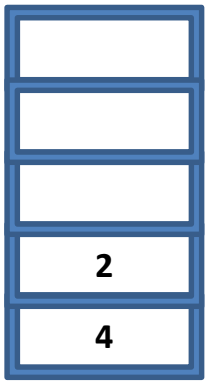
# 4 2 * 3 + 8 4 / +

- 4 is an operand:
- Push(4)



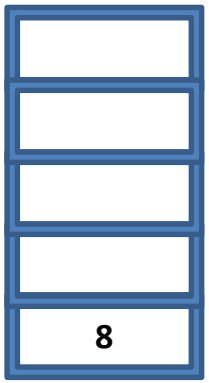
# 4 2 * 3 + 8 4 / +

- 2 is an operand:
- Push(2)



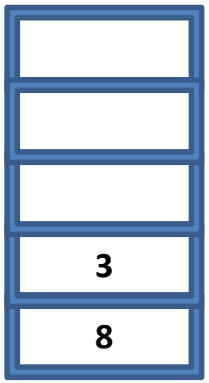
# 4 2 * 3 + 8 4 / +

- * is an operator:
- B = POP() = 2
- A = POP () = 4
- A * B = 2 * 4 = 8
- PUSH(8)



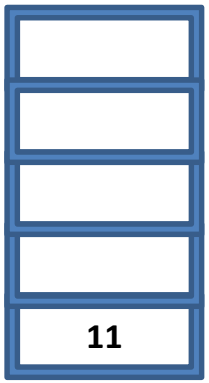
# 4 2 * 3 + 8 4 / +

- 3 is an operand:
- PUSH(3)



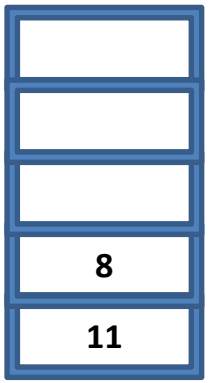
# 4 2 * 3 + 8 4 / +

- + is an operator:
- B = POP() = 3
- A = POP () = 8
- A + B = 8 + 3 = 11
- PUSH(11)



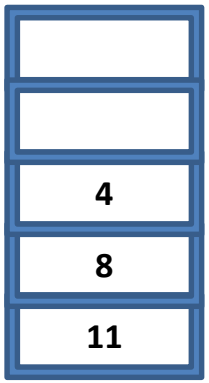
# 4 2 * 3 + 8 4 / +

- 8 is an operand:
- PUSH(8)



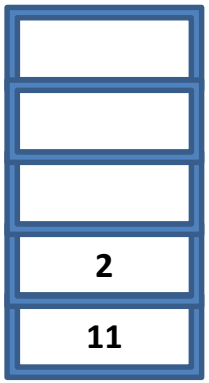
# 4 2 * 3 + 8 4 / +

- 4 is an operand:
- PUSH(4)



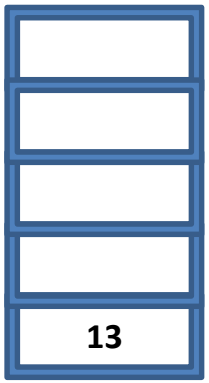
# 4 2 * 3 + 8 4 / +

- / is an operator:
- B = POP() = 4
- A = POP () = 8
- A / B = 8 / 4 = 2
- PUSH(2)



# 4 2 * 3 + 8 4 / +

- + is an operator:
- B = POP() = 2
- A = POP () = 11
- A + B = 11 + 2 = 13
- PUSH(13)




We are end of our postfix expression. So, result = POP() = 13

# All steps in one slide

Evaluate the following postfix expression: 4 2 * 3 + 8 4 / +

| step | symbol | operation | Stack |
|---|---|---|---|
| 1 | 4 | Push(4) | 4 |
| 2 | 2 | Push(2) | 4, 2 |
| 3 | * | B=Pop()=2, A= pop()=4, push(A*B) | 8 |
| 4 | 3 | Puhs(3) | 8, 3 |
| 5 | + | B=pop() = 3, A = pop()= 8, push(A+B) | 11 |
| 6 | 8 | Push(8) | 11, 8 |
| 7 | 4 | Push(4) | 11, 8, 4 |
| 8 | / | B=pop() = 4, A = pop()= 8, push(A/B) | 11, 2 |
| 9 | + | B=pop() = 2, A = pop()= 11, push(A+B) | 13 |
| 10 | | Result= Pop() = 13 | |

# Exercise

1. Evaluate the following postfix expression:

   5 6 + 7 * 8 9 * -

2. Convert the following Infix to Postfix expression:

   a) (A + B)*C-D*E
   b) ( ( A + B ) * D ) ↑ ( E - F)

3. Convert the following Infix to Postfix expressions and evaluate it:

   a) 3 + (4 * 5 – (6 / 7 ↑ 8 ) * 9 ) * 10
   b) 2 ↑ 3 + 5 * 2 ↑ 2 – 12 / 6

# Exercise Answers

1. 5

2. Convert the following Infix to Postfix expression:

a) A B + C * D E * -

b) **A B + D * E F-^**

3. Convert the following Infix to Postfix expressions and evaluate it:

   a) Postfix: **3 4 5 * 6 7 8 ^/ 9 *- 10 *+  Eval: 203**

   b) (2 ↑ 3 + 5 * 2 ↑ 2 – 12 / 6)  Answer: see the next slide

(2 ↑ 3 + 5 * 2 ↑ 2 – 12 / 6)

| Step | Expression | Stack | Postfix |
|---|---|---|---|
| 0 | ( | ( | 2 |
| 1 | 2 | ( | 2 |
| 2 | ^ | (^ | 2 |
| 3 | 3 | (^ | 2 3 |
| 4 | + | (+ | 2 3 ^ |
| 5 | 5 | (+ | 2 3 ^ 5 |
| 6 | * | (+* | 2 3 ^ 5 |
| 7 | 2 | (+* | 2 3 ^ 5 2 |
| 8 | ^ | (+*^ | 2 3 ^ 5 2 |
| 9 | 2 | (+*^ | 2 3 ^ 5 2 2 |
| 10 | - | (- | 2 3 ^ 5 2 2 ^ * + |
| 11 | 12 | (- | 2 3 ^ 5 2 2 ^ * + 12 |
| 12 | / | (-/ | 2 3 ^ 5 2 2 ^ * + 12 |
| 13 | 6 | (-/ | 2 3 ^ 5 2 2 ^ * + 12 6 |
| 14 | ) | | 2 3 ^ 5 2 2 ^ * + 12 6 / - |

# Question?

Than you ☺