MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

Week 10 Tutorial

Tutorial n. 8

Department of Computing and Software

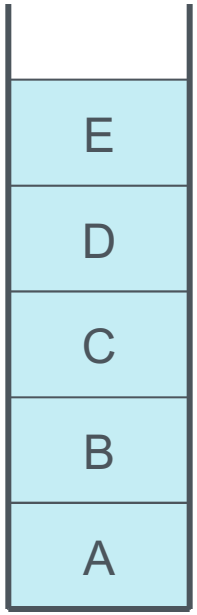
Instructor:

Omid Isfahanialamdari

March 15, 2022

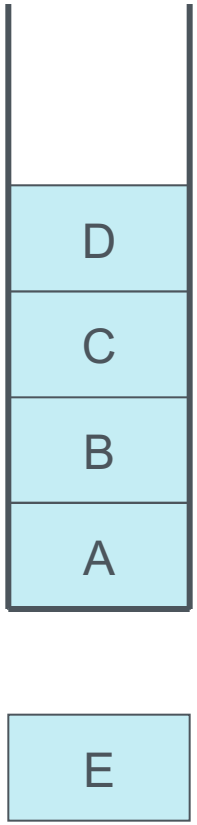
Question 1

- Design an algorithm that given a stack outputs its elements in the order that they have been pushed and restores the stack to its original state.



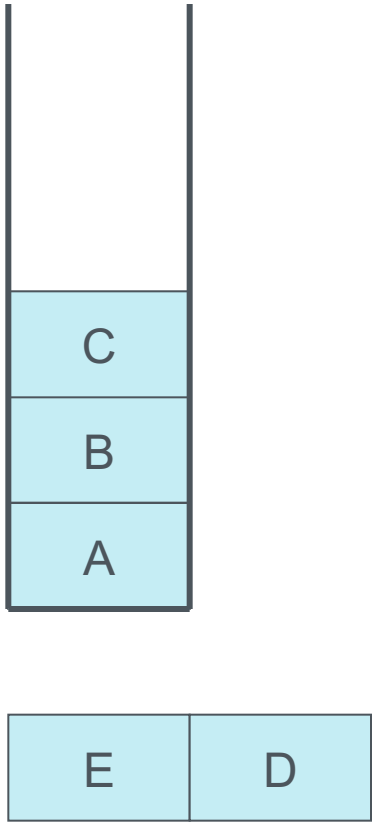
Question 1

- Design an algorithm that given a stack outputs its elements in the order that they have been pushed and restores the stack to its original state.



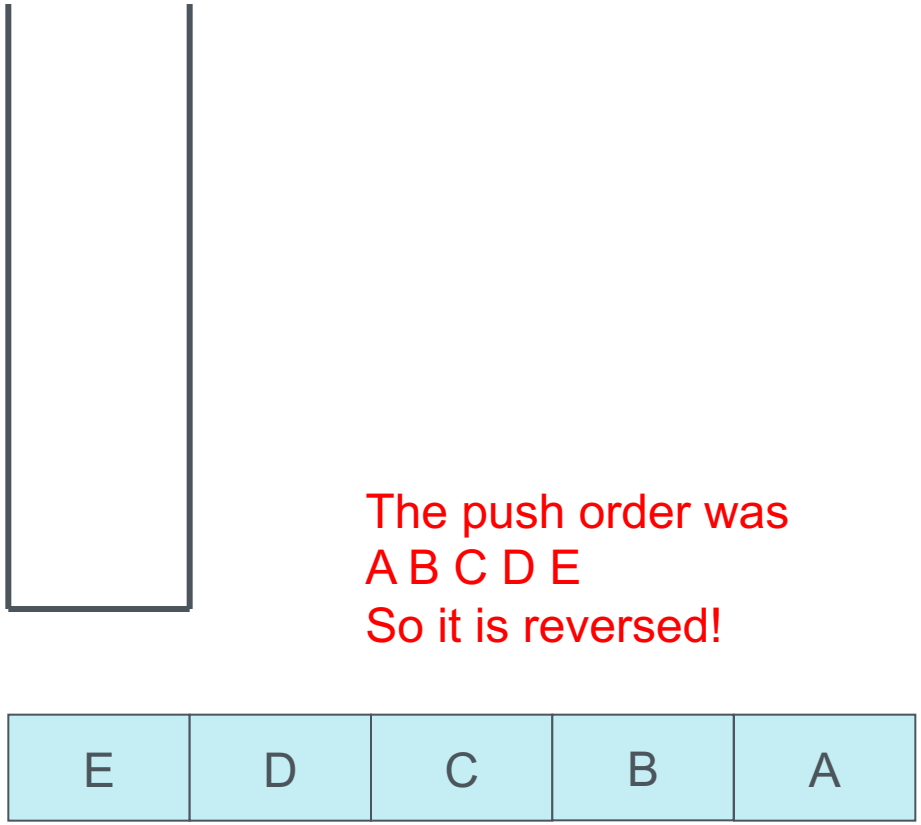
Question 1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



Question 1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



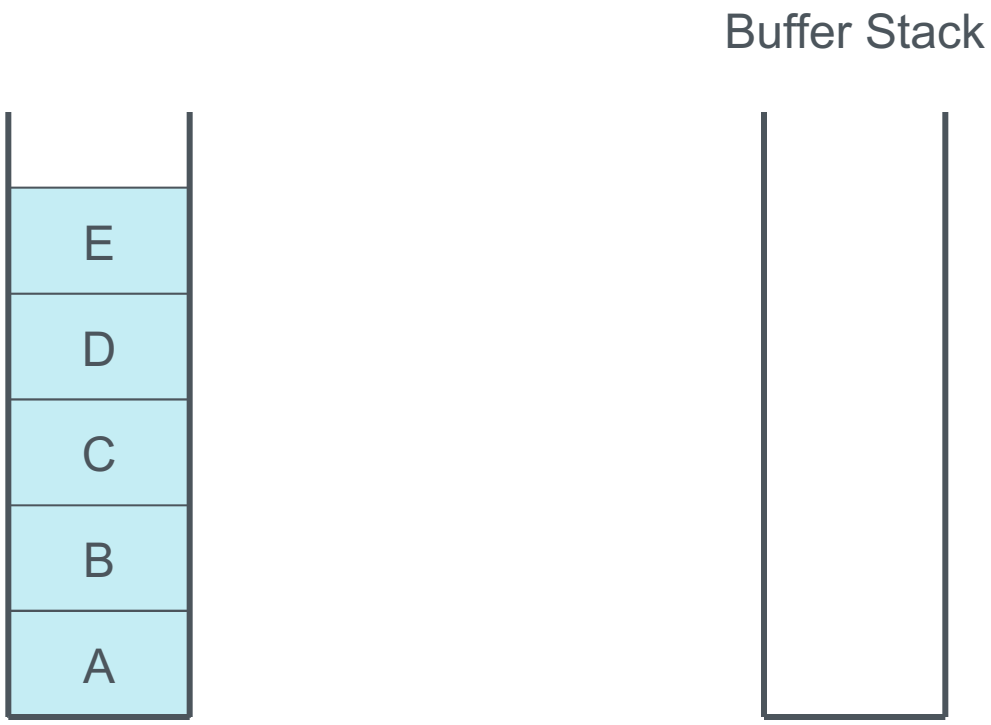
Solution to Q1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.
  - suppose the elements are already on stack **s**
  - define a buffer stack **b**
  - While (!**s**.empty()){
    - **b**.push(**s**.top())
    - **s**.pop()
  - While (!**b**.empty()){
    - topElement ← **b**.top()
    - **s**.push(topElement)
    - **b**.pop()

- At the end **s** will be the same as the beginning.

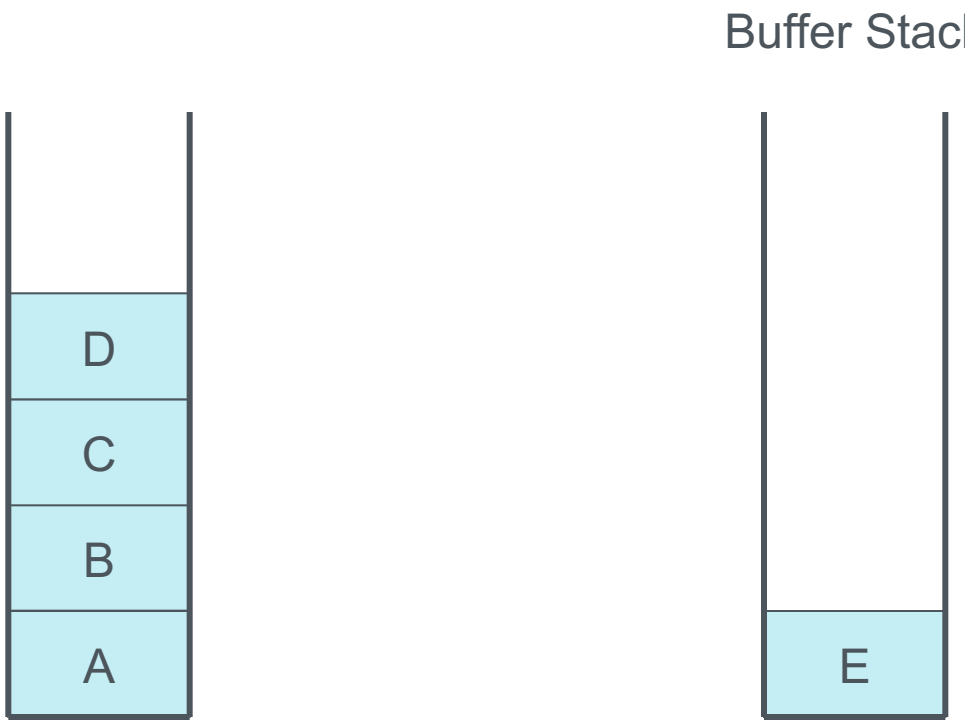
Solution to Q1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



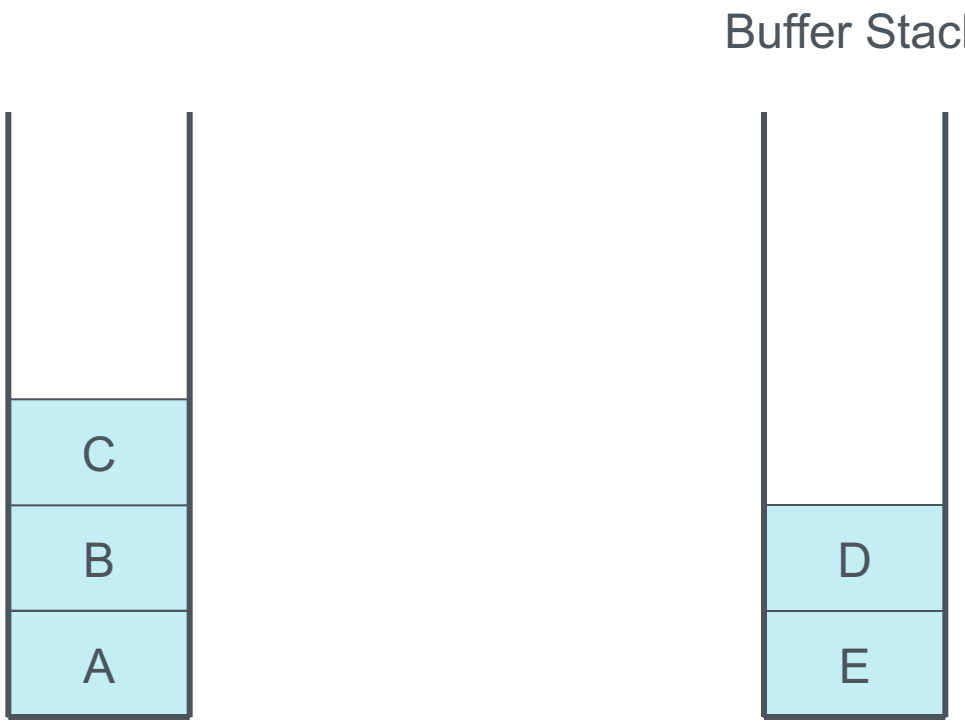
Solution to Q1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



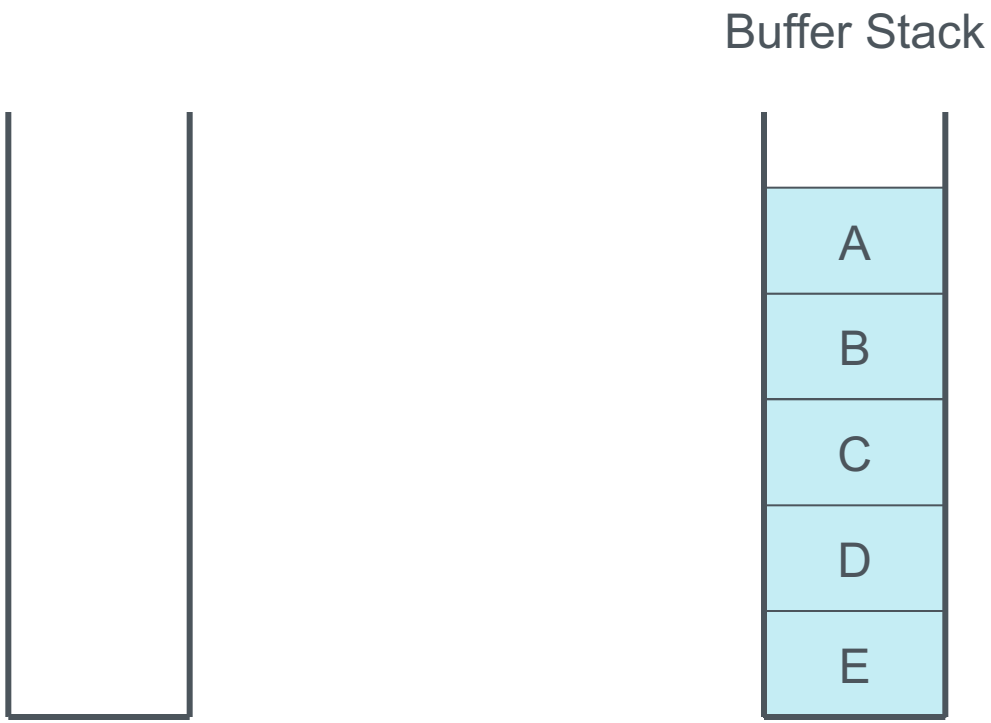
Solution to Q1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



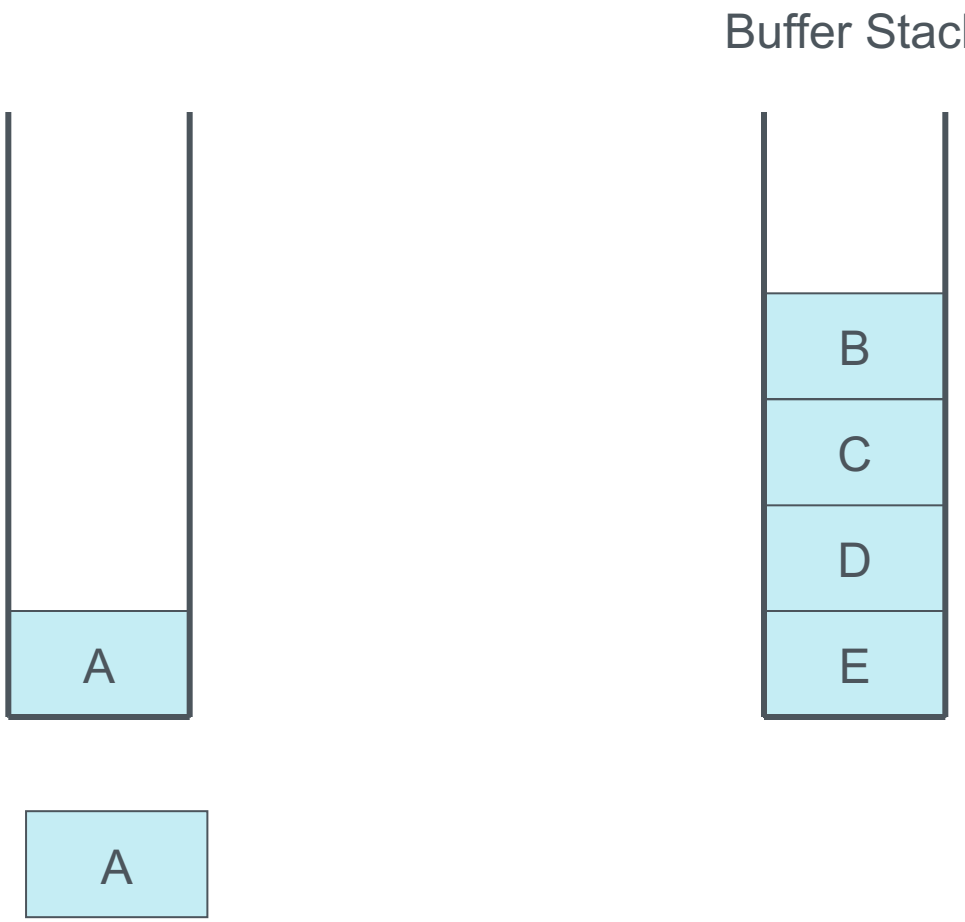
Solution to Q1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



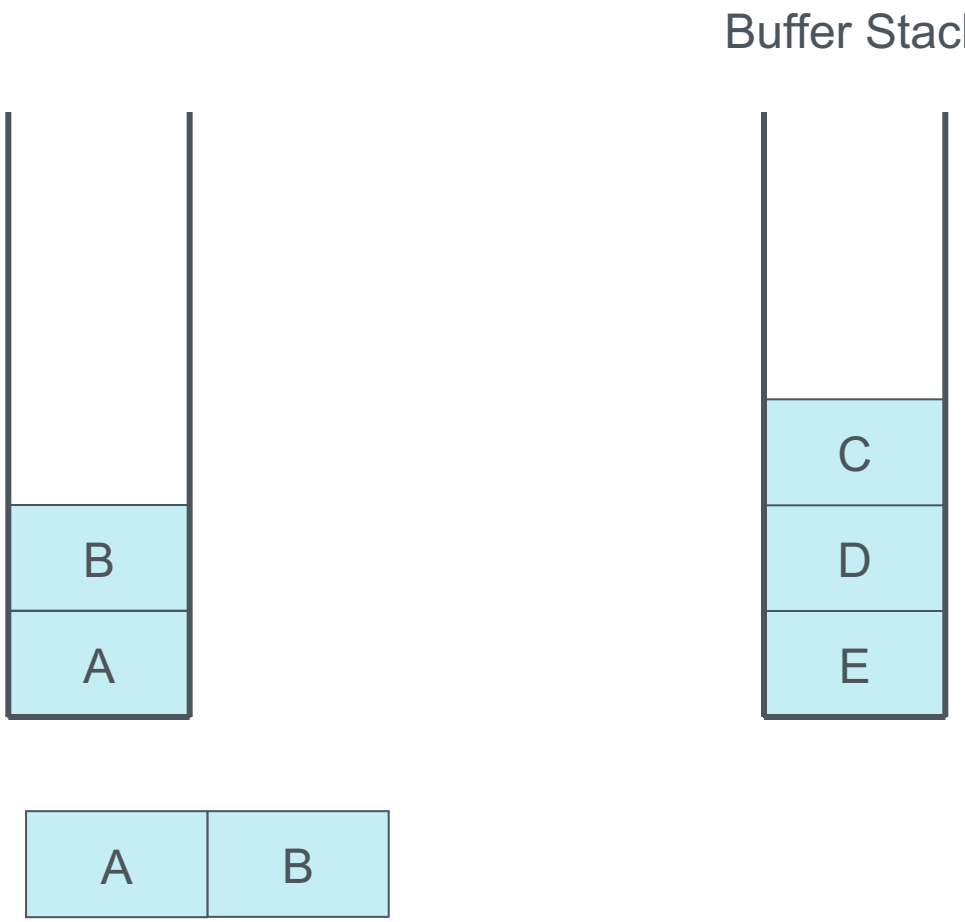
Solution to Q1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



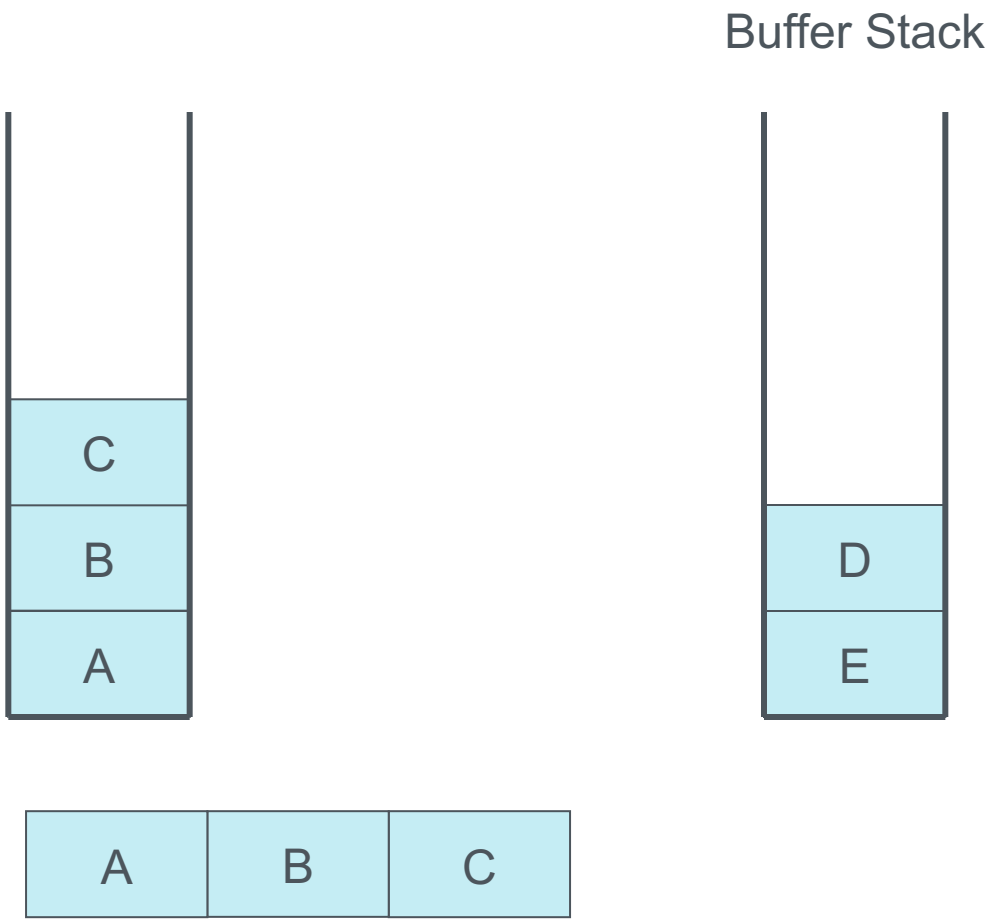
Solution to Q1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



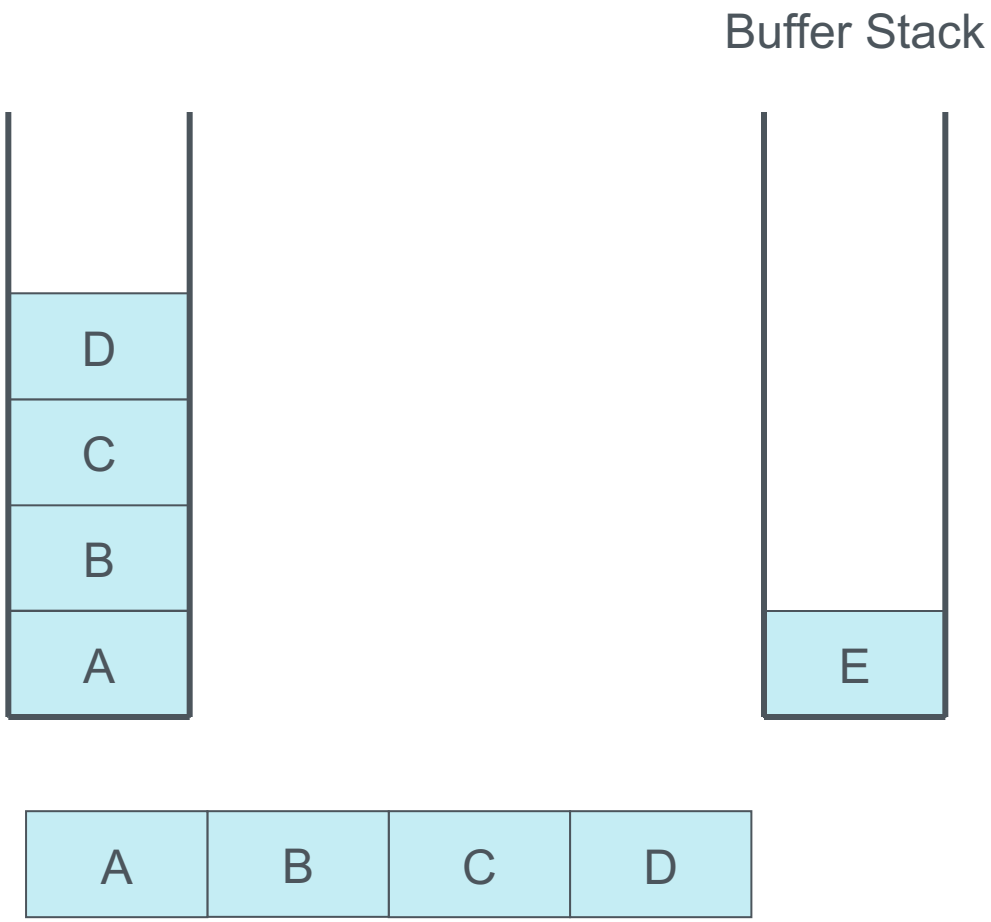
Solution to Q1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



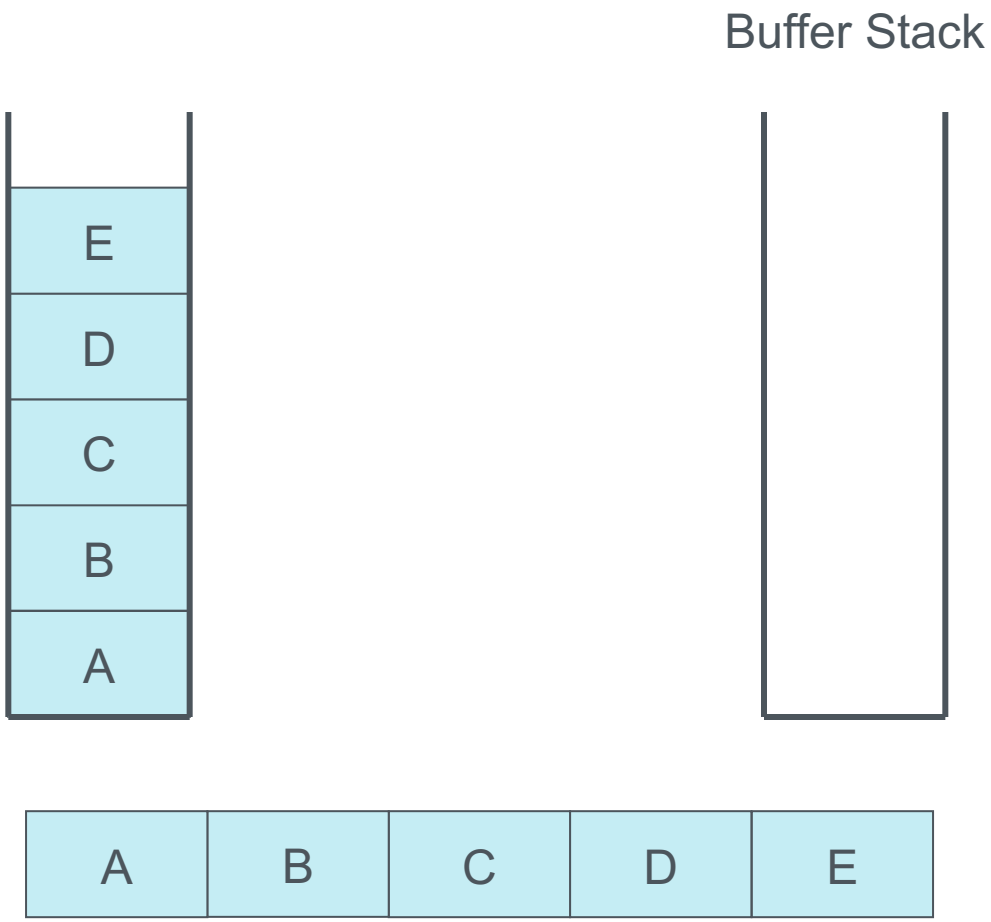
Solution to Q1

- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



Solution to Q1

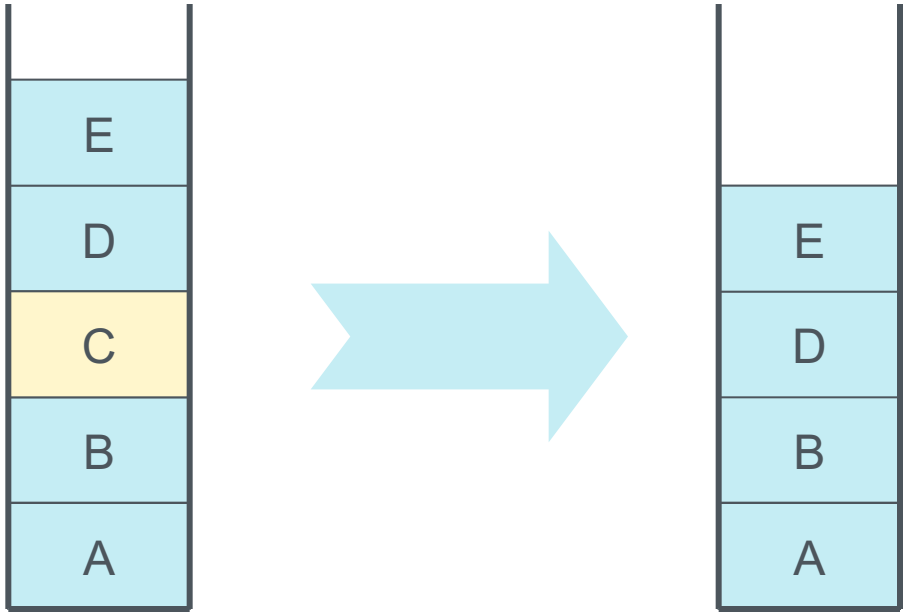
- Design an algorithm that given a stack outputs its elements in the in the order that they have been pushed and restores the stack to its original state.



Question 2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We want to remove ‘C’



Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.
  - suppose the elements are already on stack **s**
  - define a buffer stack **b**
  - While (!**s**.empty()){
    - topElement ← **s**.top()
    - **s**.pop()
    - if (topElement = c)

      break

      else

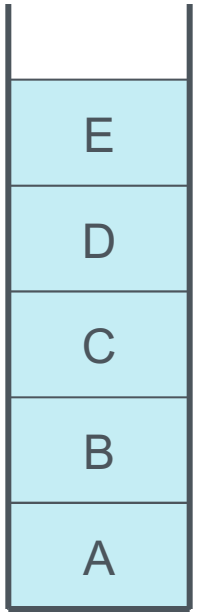
      b.push(topElement)
  - While (!**b**.empty()){
    - topElement ← **b**.top()
    - **b**.pop()
    - **s**.push(topElement)

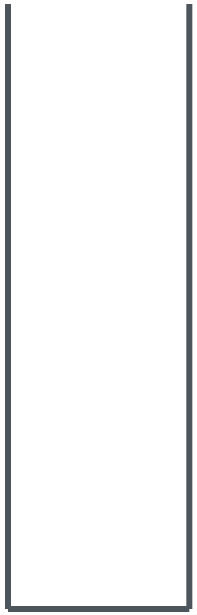
Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We are looking for ‘C’

Buffer Stack

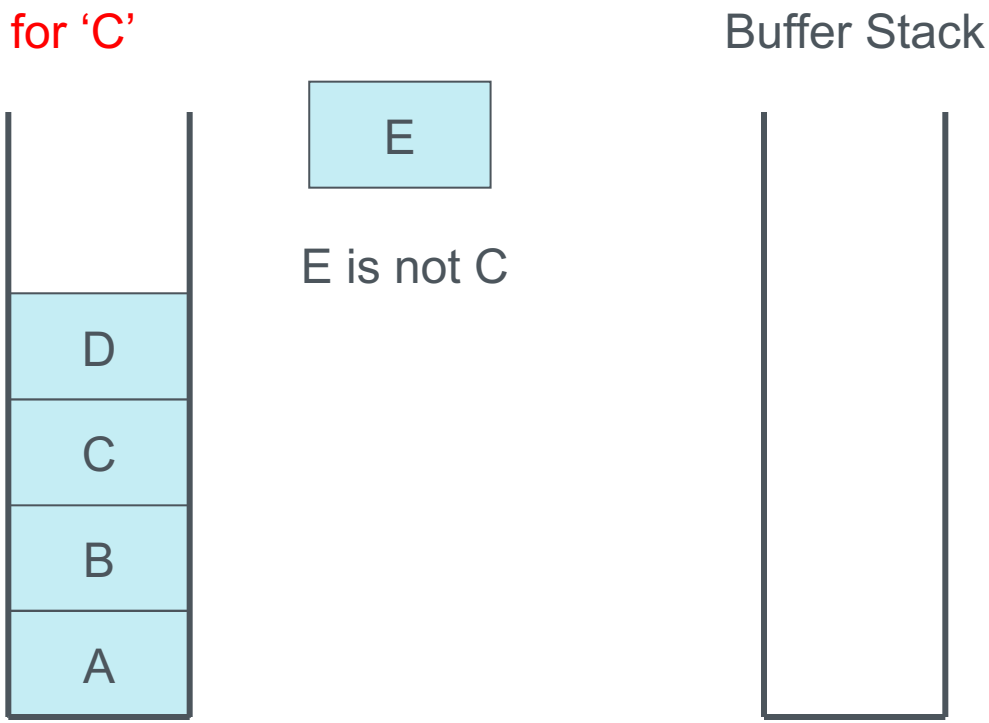




Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We are looking for ‘C’

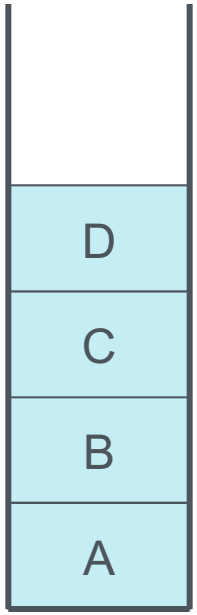


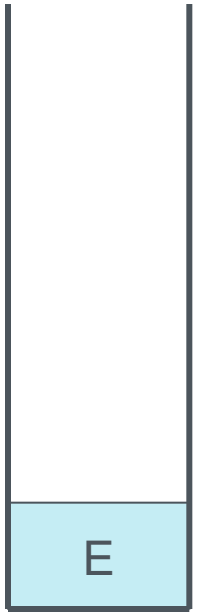
Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We are looking for ‘C’

Buffer Stack



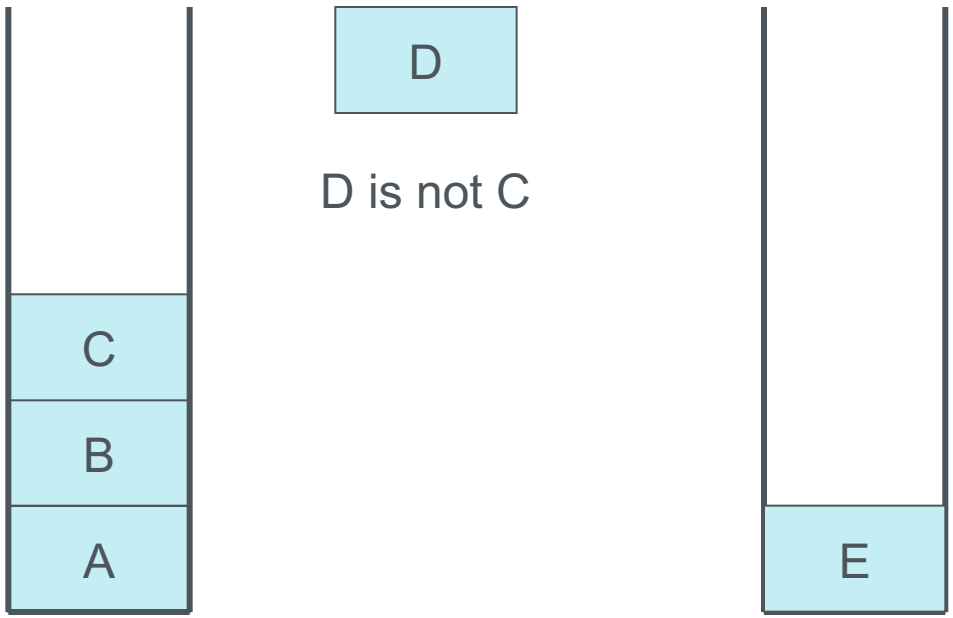


Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We are looking for ‘C’

Buffer Stack

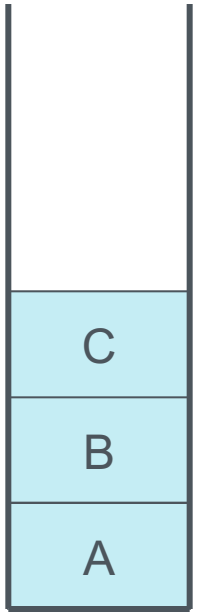


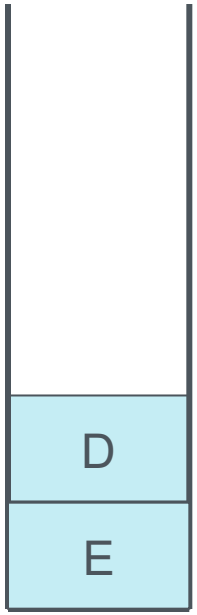
Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We are looking for ‘C’

Buffer Stack



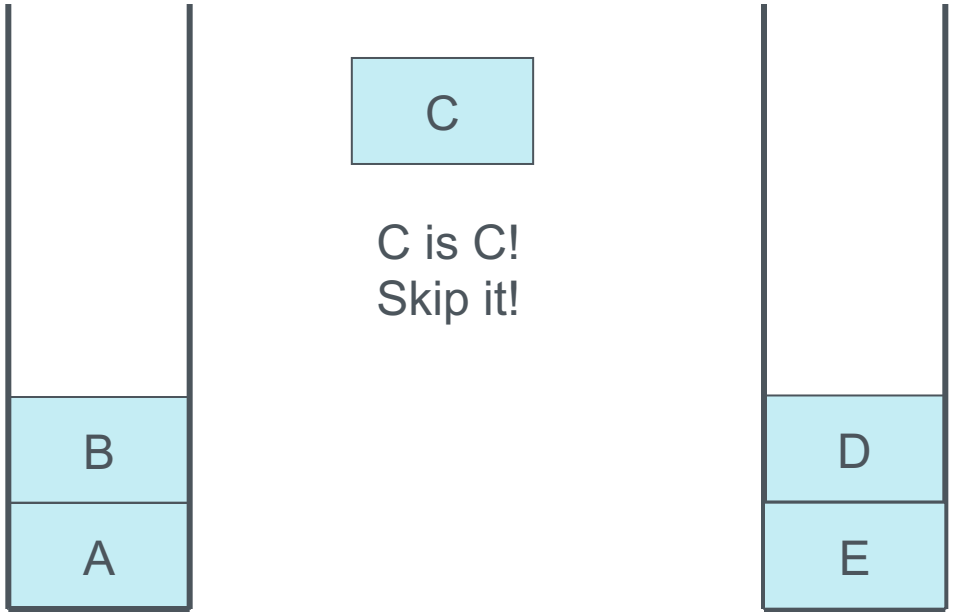


Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We are looking for ‘C’

Buffer Stack

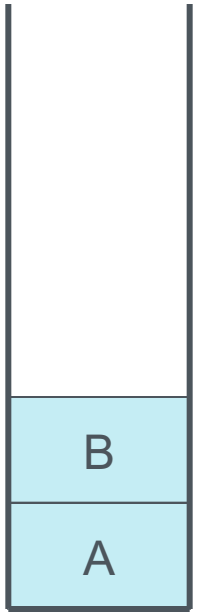


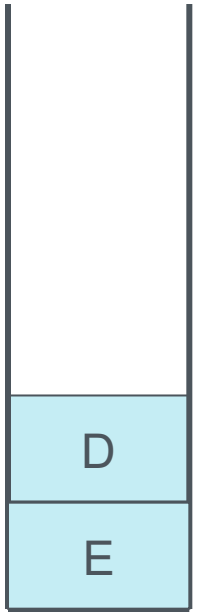
Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We are looking for ‘C’

Buffer Stack



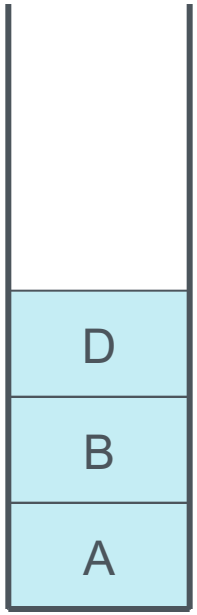


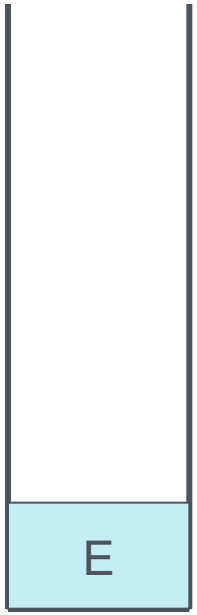
Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We are looking for ‘C’

Buffer Stack



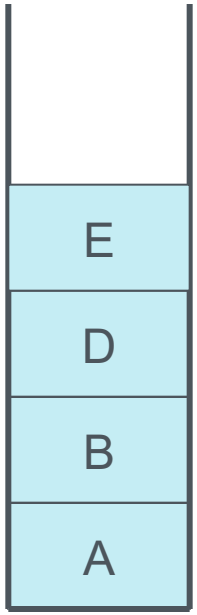


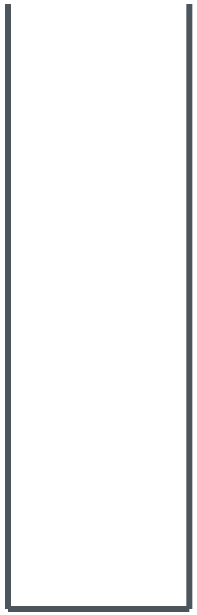
Solution to Q2

- Design an algorithm that given a stack, and an element **c** to be removed, removes the element **c** from the stack.

We are looking for ‘C’

Buffer Stack





Question 3

- We are going to talk about evaluating expressions
- Expressions
  - An expression is made up of operands, operators and delimiters.
  - Example:

    A / B – C + D * E – A * C

    - Operands: A, B, C, D, E (or constants).
    - Operators: /, –, +, *.
    - Delimiter: (,).

- In this question, we want to convert expressions to the Postfix notation

How to Evaluate an Expression in the Right Order?

- Assign each operator a priority
  - Priority of operators in C++

Priority	Operator
1	Unary minus, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	==, !=
6	&&
7	||

High priority

Low priority

- How to evaluate A * B / C? Evaluate the expression from left to right.
- Use parentheses to define computation priority.

Postfix Notation

- Each operator appears after its operands.
- The way of how compiler evaluate an expression.
  - Example:

Infix Notation	Postfix Notation
A + B	A B +
A * B / C	A B * C /
A/ B – C + D * E – A * C	A B / C – D E * + A C * –

Postfix Notation

1. Use parentheses to group operands according to the operator they use.
2. Change the position of operator.
3. Remove all the parentheses.
   - Example (Try yourself)
     - A * B / C
       - ((A * B) / C)
       - ((A B) * / C)
       - ((A B) * C) /
       - A B * C /

Postfix Notation

1. Use parentheses to group operands according to the operator they use.
2. Change the position of operator.
3. Remove all the parentheses.
   - Example:

$$A / B - C + D * E - A * C$$

Step 1. $$(((A / B) - C) + (D * E)) - (A * C))$$

Postfix Notation

1. Use parentheses to group operands according to the operator they use.
2. Change the position of operator.
3. Remove all the parentheses.
   - Example:

A / B – C + D * E – A * C

Step 1. ((((A / B) – C) + (D * E)) – (A * C))

Postfix Notation

1. Use parentheses to group operands according to the operator they use.
2. Change the position of operator.
3. Remove all the parentheses.
   - Example:

A / B – C + D * E – A * C

Step 1. ((((A / B) – C) + (D * E)) – (A * C))

Step 2. ((((A B) / C) – (D E) *) + (A C) *) –

Postfix Notation

1. Use parentheses to group operands according to the operator they use.
2. Change the position of operator.
3. Remove all the parentheses.
   - Example:

A / B – C + D * E – A * C

Step 1. ((((A / B) – C) + (D * E)) – (A * C))

Step 2. ((((A B) / C) – (D E) *) + (A C) *) –

Step 3. A B / C – D E * + A C * –

Reasons to Convert Expressions to Postfix Notation

- Parentheses are eliminated.
  - Example:

Infix Notation	Postfix Notation
A * (B + C)	A B C + *
A/ B – (C + D) * E	A B / C D + E * –

- Easier than infix evaluation.
  - The priority of the operators is no longer relevant.

Evaluate an Infix Expression

- Issues:
  - How to convert infix notation to the postfix?
  - How to evaluate an postfix expression?
  - Clue:
    - Using stack.

Convert from Infix to Postfix

- Observation:
  - The order of operands is unchanged
    - Output operands immediately.

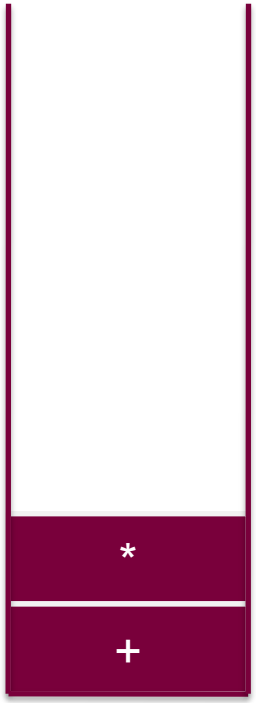
      A + B * C – D → A B C * + D –
      - The order of A, B, C and D is unchanged.
- Stack operators until it is time to pass them to the output.

Example

A + B * C

Stack

	Next	Stack	Output	
Init.	none	empty	none	
	A	empty	A	Output operand
	+	+	A	Stack operator
	B	+	AB	
	*	+*	AB	'*' has a priority higher than '+'.
	C	+*	ABC	
		empty	ABC*+	Clear the stack



* Pop out stacked operators that has higher priority.

Output:

A	B	C	*	+

Observation

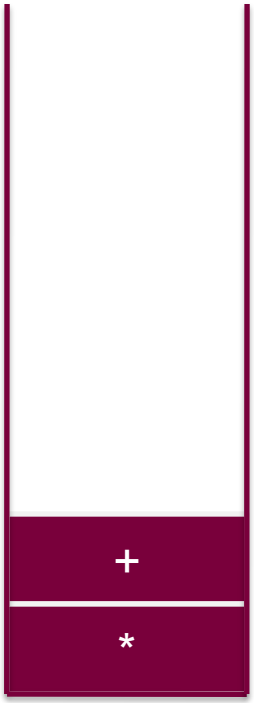
- The stack is used for operators.
- The more upper the operator is, the higher its priority is.
  - When a new operator *p* is coming, operators that has higher priority than *p* will be popped out first before *p* is pushed.

Example 2

(A + B) * C

Next	Stack	Output	
none	empty	none	
(	(	none	'(' is pushed directly
A	(	A	
+	(+	A	No operator except ')' can pop '('
B	(+	AB	
)	empty	AB+	')' pops out all operators until encountering the first '('; parentheses will not be output.
*	*	AB+	
C	empty	AB+C	
		AB+C*	

Stack



Output:

A	B	+	C	*

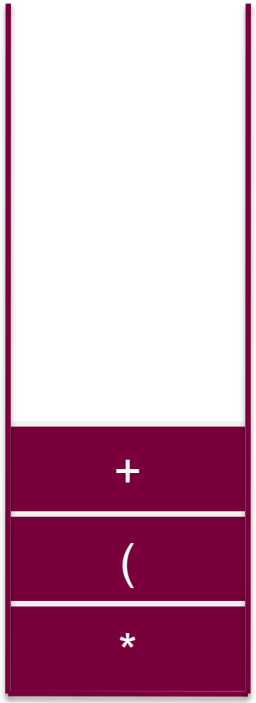
Example 3

A * (B + C) * D

nit.

Next	Stack	Output	
none	empty	none	
A	empty	A	
*	*	A	
(	*(	A	
B	*(	AB	
+	*(+	AB	
C	*(+	ABC	
)	*	ABC+	
*	*	ABC+*	The same priority. Pop out the old one and push the new one.
D	*	ABC+*D	
		ABC+*D*	

Stack



Output:

A	B	C	+	*	D	*

Algorithm

```
bool isOperand(const char c){
    if((c >='a' && c<='z') || (c >='A' && c<='Z')){
        return true;
    }
    return false;
}
```

```
int priority(const char c){
    if(c == '+' || c =='-')
        return 1;

    if(c == '*' || c =='/')
        return 2;

    if(c == '^')
        return 3;

    return 0;
}
```

```
stack<char> s;
string exp = "A*(B+C)*D";
for(int i = 0; i < exp.size(); ++i) {
    if (isOperand(exp[i])){
        cout << exp[i] << endl;
    }
    else if (exp[i] == '('){
        s.push(exp[i]);
    }
    else if (exp[i] != ')'){
        while (!s.empty()) {
            char y = s.top(); s.pop();
            if (y != '(' && priority(y) >= priority(exp[i]))
                cout << y << endl;
            else {
                s.push(y);
                break;
            }
        }
        s.push(exp[i]);
    }
    else{  // exp[i] is ')'
        while (!s.empty()) {
            char y = s.top(); s.pop();
            if (y != '(')
                cout << y << endl;
            else {
                break;
            }
        }
    }
}
while (!s.empty()) {
    char y = s.top(); s.pop();
    cout << y <<endl;
}
```

Analysis of Algorithm

Suppose the input expression has length of *n*.

- Space complexity: $O(n)$.
  - The stack used to buffer operators at most requires $O(n)$ elements.
- Time complexity:
  - The function make only a left-to-right pass across the input.
  - The time spent on each operand is $O(1)$.
  - The time spent on each operator is $O(1)$.
    - Each operator is stacked and unstacked at most once.
  - Thus, the time complexity of InfixToPostfix() is $O(n)$.

```
stack<char> s;
string exp = "A*(B+C)*D";
for(int i = 0; i < exp.size(); ++i) {
    if (isOperand(exp[i])){
        cout << exp[i] << endl;
    }
    else if (exp[i] == '('){
        s.push(exp[i]);
    }
    else if (exp[i] != ')'){
        while (!s.empty()) {
            char y = s.top(); s.pop();
            if (y != '(' && priority(y) >= priority(exp[i]))
                cout << y << endl;
            else {
                s.push(y);
                break;
            }
        }
        s.push(exp[i]);
    }
    else{  // exp[i] is ')'
        while (!s.empty()) {
            char y = s.top(); s.pop();
            if (y != '(')
                cout << y << endl;
            else {
                break;
            }
        }
    }
}
while (!s.empty()) {
    char y = s.top(); s.pop();
    cout << y <<endl;
}
```

Questions?