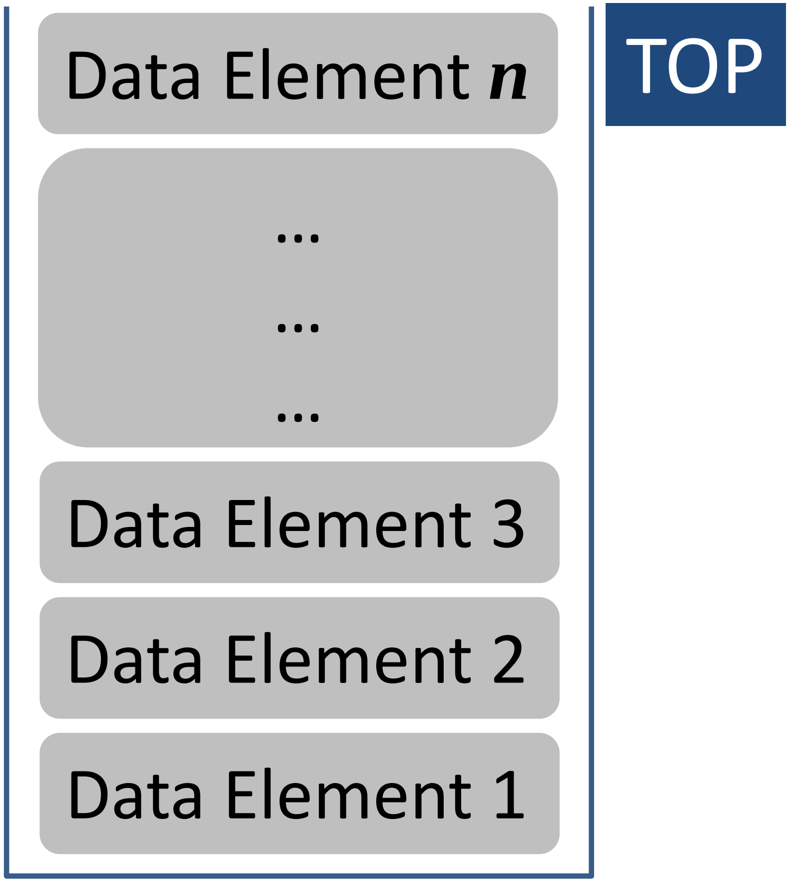
# Lecture 9-10

## Stacks

# Introduction

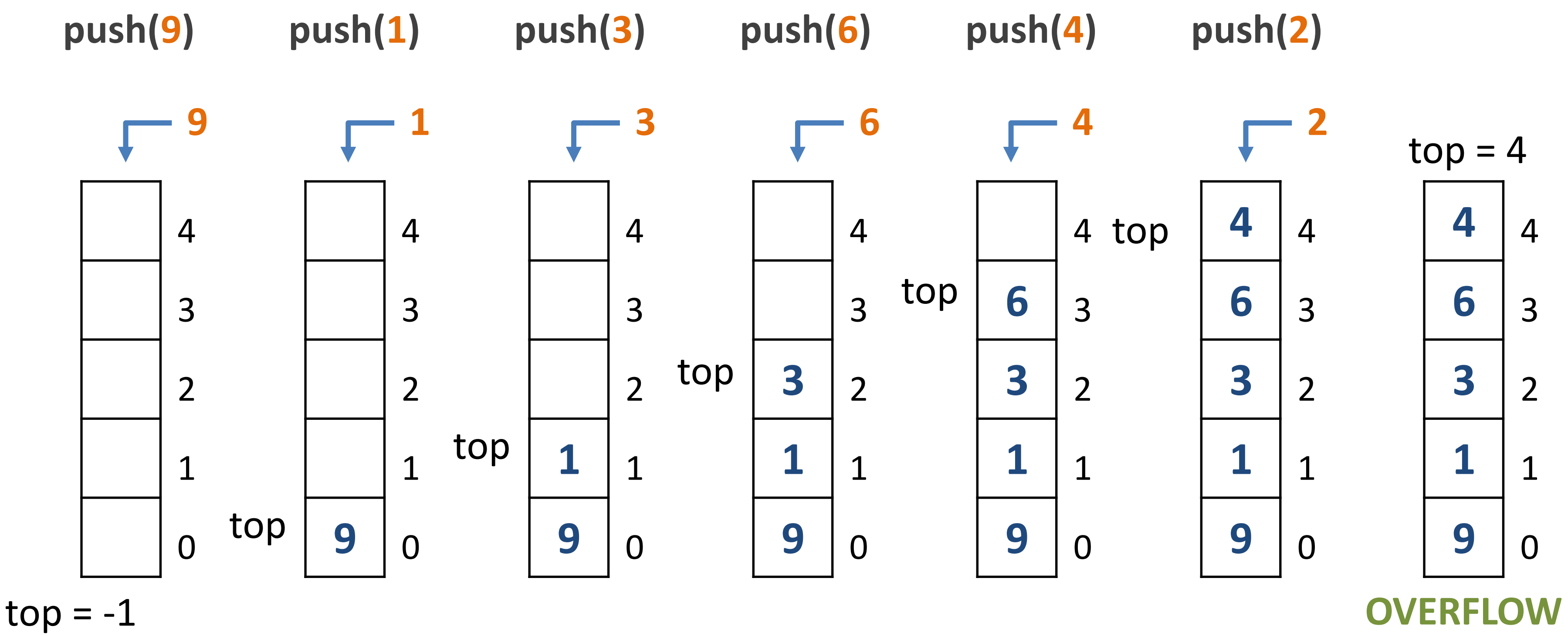
- Non primitive linear data structure.
- Allows operations at one end only.
- The top element can only be accessed at any time.
  - LIFO (Last-in-first-out) data structure.



# Operations

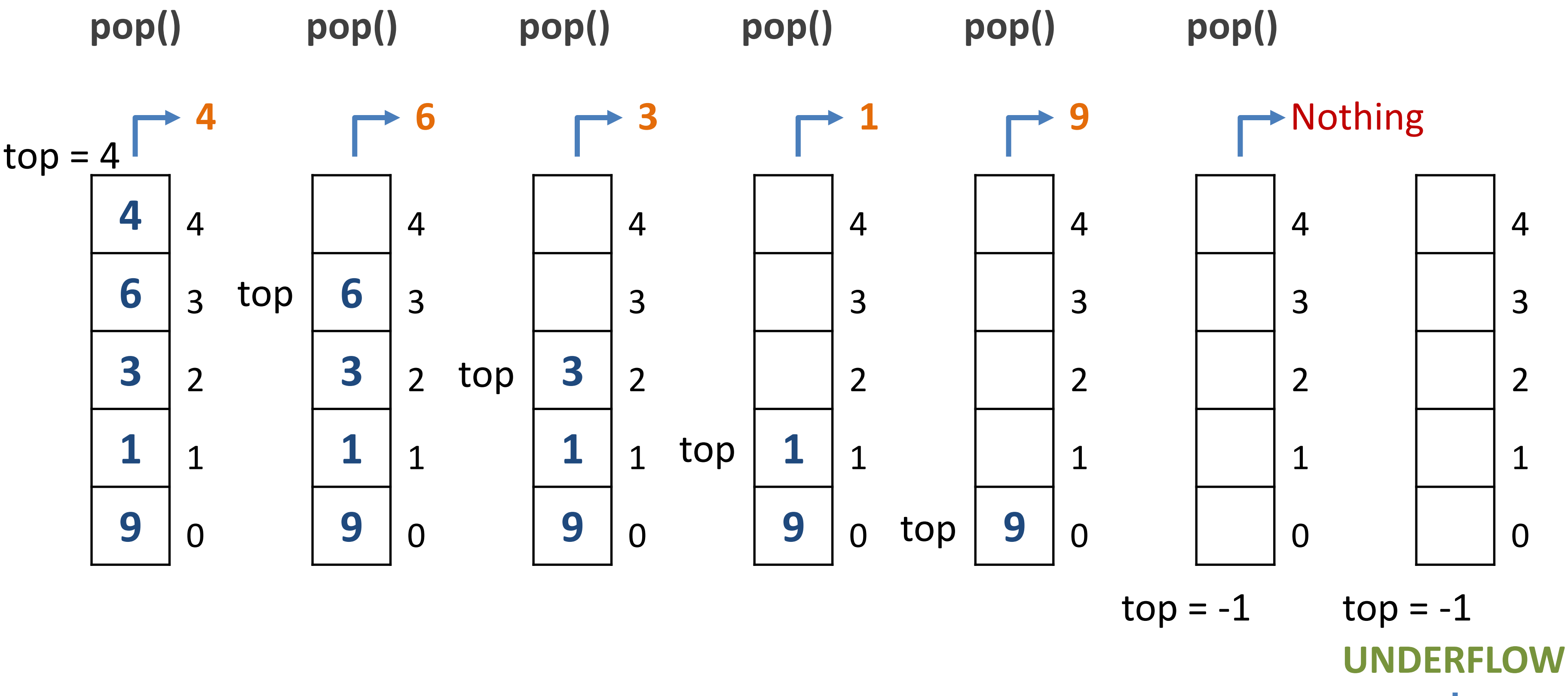
- Two primary operations:
  - **push()** – Pushing (storing) an element on the stack.
  - **pop()** – Removing (accessing) an element from the stack.
- Other operations for effective functionality:
  - **peek()** / **top()** – Get the top data element of the stack, without removing it.
  - **isFull()** – Check if stack is full. ***OVERFLOW***
  - **isEmpty()** – Check if stack is empty. ***UNDERFLOW***

# Stack – Push




The stack is full, no more elements can be added. **OVERFLOW**

# Stack – Pop




The stack is empty, no element can be removed. **UNDERFLOW**

# Stack as an ADT

- A stack is an ordered list of elements of same data type.
- Elements are always inserted and deleted at one end.
- Following are its basic operations:
  - ***S = init()*** – Initialize an empty stack.
  - ***isEmpty(S)*** – Returns "true" if and only if the stack S is empty, i.e., contains no elements.

# Stack as an ADT

- ***isFull(S)*** – Returns "true" if and only if the stack S has a bounded size and holds the maximum number of elements it can.
- ***top(S)*** – Returns the element at the top of the stack S, or error if the stack is empty.
- ***S = push(S,x)*** – Push an element x at the top of the stack S.
- ***S = pop(S)*** – Pop an element from the top of the stack S.
- ***print(S)*** – Prints the elements of the stack S from top to bottom.

# Implementation

- Using static arrays
  - Realizes stacks of a maximum possible size.
  - Top is taken as the maximum index of an element in the array.
- Using dynamic linked lists
  - Choose beginning of the list as the top of the stack.

# Using Static Arrays

# Algorithm for Push

- Let,
  - STACK[SIZE] is a one dimensional array that will hold the stack elements.
  - TOP is the pointer that points to the top most element of the stack.
  - DATA is the data item to be pushed.

1. If TOP == SIZE – 1
2. Display "Overflow condition"
3. Else
4. TOP = TOP + 1
5. STACK [TOP] = DATA

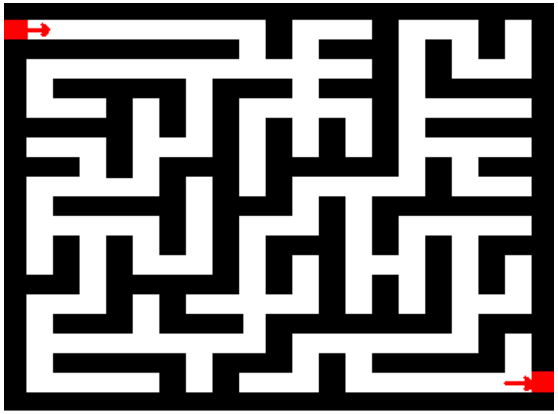
# Algorithm for Pop

- Let,
  - STACK[SIZE] is a one dimensional array that will hold the stack elements.
  - TOP is the pointer that points to the top most element of the stack.
  - DATA is the element popped from the top of the stack.

1. If TOP < 0 (or TOP == -1)
2. Display "Underflow condition."
3. Else
4. DATA = STACK[TOP]
5. TOP = TOP – 1
6. Return DATA

# Applications

- Reverse a word.
  - Push a word letter by letter and then pop letters from the stack.
- "UNDO" mechanism in text editors.
- Backtracking.
  - Game playing, finding paths, exhaustive searching.
- Parsing.
- Recursive function calls.
- Calling a function.
- Expression Evaluation.
- Expression Conversion.



# Stack Applications

# Expression Representation

- Infix – Operator is in-between the operands.
- Prefix – Operator is before the operands. Also known as polish notation.
- Postfix – Operator is after the operands. Also known as suffix or reverse polish notation.

| Infix | Prefix | Postfix |
|---|---|---|
| a + b | + a b | a b + |
| a + b * c | + a * b c | a b c * + |
| (a + b) * (c - d) | * + a b - c d | a b + c d - * |

# Example (Infix to Postfix)

- 5 + 3 * 2
    - 5 3 2 * +
- 3 + 4 * 5 / 6
    - 3 4 5 * 6 / +
- ( 300 + 23 ) * ( 43 – 21 ) / ( 84 + 7 )
    - 300 23 + 43 21 - * 84 7 + /
- ( 4 + 8 ) * ( 6 – 5 ) / ( ( 3 – 2 ) * ( 2 + 2 ) )
    - 4 8 + 6 5 - * 3 2 – 2 2 + * /

# Infix to Postfix Conversion Algorithm

Let Q be any infix expression to be converted in to a postfix expression P.

1. Push left parenthesis onto STACK and add right parenthesis at the end of Q.
2. Scan Q from left to right and repeat step 3 to 6 for each element of Q until the STACK is empty.
3. If an operand is encountered add it to P.
4. If a left parenthesis is encountered push it onto the STACK.
5. If an operator is encountered, then
   i. Repeatedly pop from STACK and add to P each operator which has same precedence as or higher precedence than the operator encountered.
   ii. Push the encountered operator onto the STACK.
6. If a right parenthesis is encountered, then
   i. Repeatedly pop from the STACK and add to P each operator until a left parenthesis is encountered.
   ii. Remove the left parenthesis; do not add it to P.
7. Exit.

| Input | Stack | Output |
|---|---|---|
| A + (B * (C – D) / E) | | |
| A + (B * (C – D) / E)) | ( | |
| + (B * (C – D) / E)) | ( | A |
| (B * (C – D) / E)) | ( + | A |
| B * (C – D) / E)) | ( + ( | A |
| * (C – D) / E)) | ( + ( | A B |
| (C – D) / E)) | ( + ( * | A B |
| C – D) / E)) | ( + ( * ( | A B |

| Input | Stack | Output |
|---|---|---|
| – D) / E)) | ( + ( * ( | A B C |
| D) / E)) | ( + ( * ( – | A B C |
| ) / E)) | ( + ( * ( – | A B C D |
| / E)) | ( + ( * | A B C D – |
| E)) | ( + ( / | A B C D – * |
| )) | ( + ( / | A B C D – * E |
| ) | ( + | A B C D – * E / |
| | | A B C D – * E / + |

# Postfix Evaluation Algorithm

1. Initialize empty stack
2. For every token in the postfix expression (scanned from left to right):
   a. If the token is an operand (number), push it on the stack
   b. Otherwise, if the token is an operator (or function):
      i. Check if the stack contains the sufficient number of values (usually two) for given operator
      ii. If there are not enough values, finish the algorithm with an error
      iii. Pop the appropriate number of values from the stack
      iv. Evaluate the operator using the popped values and push the single result on the stack
3. If the stack contains only one value, return it as a final result of the calculation
4. Otherwise, finish the algorithm with an error

# Example

- Evaluate 2 3 4 + * 6 –
- 2 * (3 + 4) - 6 = 8.

| | Input token | Operation | Stack contents (top on the right) | Details |
|---|---|---|---|---|
| 2 3 4 + * 6 – | 2 | Push on the stack | 2 | |
| 3 4 + * 6 – | 3 | Push on the stack | 2, 3 | |
| 4 + * 6 – | 4 | Push on the stack | 2, 3, 4 | |
| + * 6 – | + | Add | 2, 7 | Pop two values: 3 and 4 and push the result 7 on the stack |
| * 6 – | * | Multiply | 14 | Pop two values: 2 and 7 and push the result 14 on the stack |
| 6 - | 6 | Push on the stack | 14, 6 | |
| - | - | Subtract | 8 | Pop two values: 14 and 6 and push the result 8 on the stack |
| | (End of tokens) | (Return the result) | 8 | Pop the only value 8 and return it |

# Thank You