Data Structure and Algorithm

Abstract Data Type

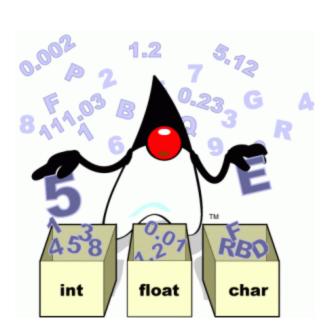Lecturer: Lê Ngọc Thành
Email: lnthanh@fit.hcmus.edu.vn

HCM City

Contents

- **Introduction**
- Some Abstract Data Types
  - Array
  - Linked List
  - Stack
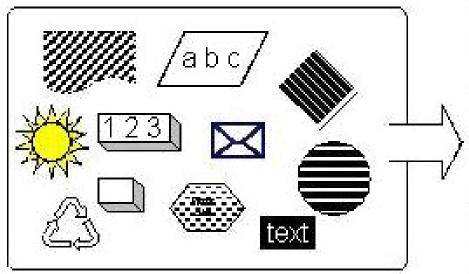 - Queue
- Application Exercises

Datatype

- *Datatype* in a programming language (*primitive data type)* is a data set with pre-defined values.

- For example:
  - Interger: …0, 1, 2, …
  - Floating Point: …
  - Character: …
  - String: …

Data in Real Life

User defined data type

- When system data types are insufficient, most programming languages allow users to define new data types based on available data types.

- Example:

```
struct myType {
    int x;
    float y;
    char z;
};
```

Abstract data type (ADT)

- System data type (int, float, ...) supports basic implementations:

  – +, -, *, /, div, mod, >, <, !=, …

- With user data types, we also need to describe their implementations.

→ They call it abstract data type

```
class Dictionary {
    Dictionary();
    void insert(int x, int y);
    void delete(int x);
    ...
}
```

insert()
delete()
find_min()
find()
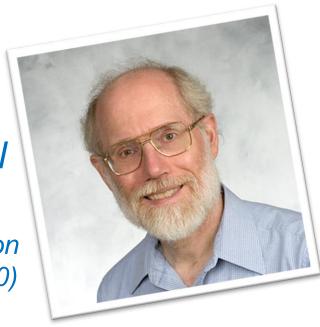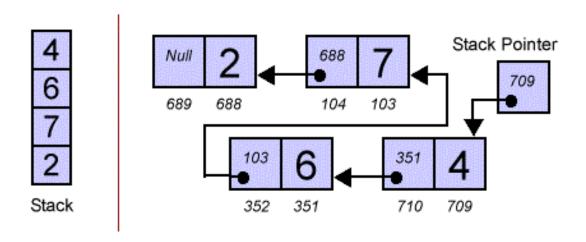
```
int main() {
    D = new Dictionary()
    D.insert(3,10);
    cout << D.find(3);
}
```

Abstract data type
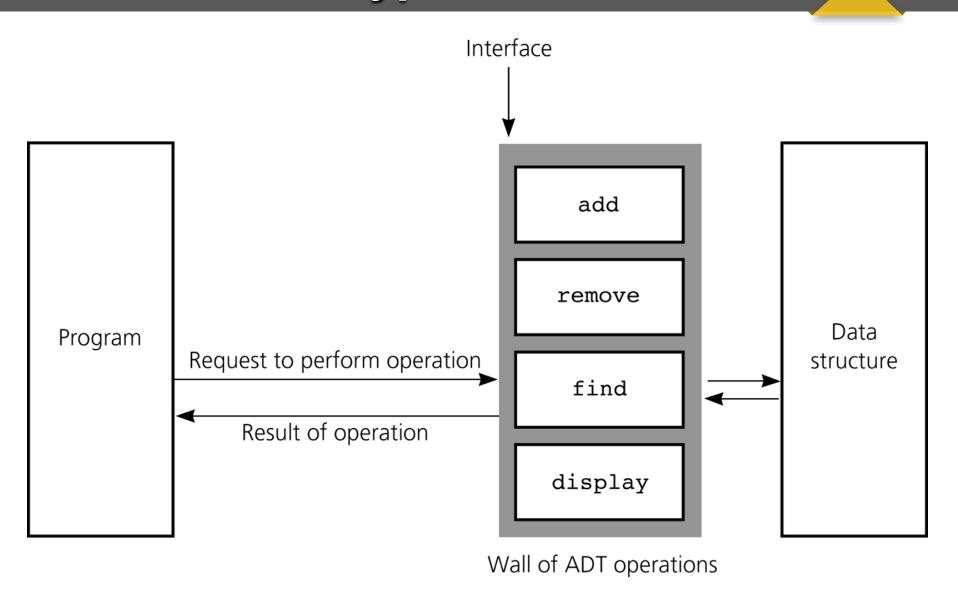
"Get your data structures correct first, and the rest of the program will write itself"

David S.Johnson
(winner of Knuth's Prize in 2010)

Abstract data type
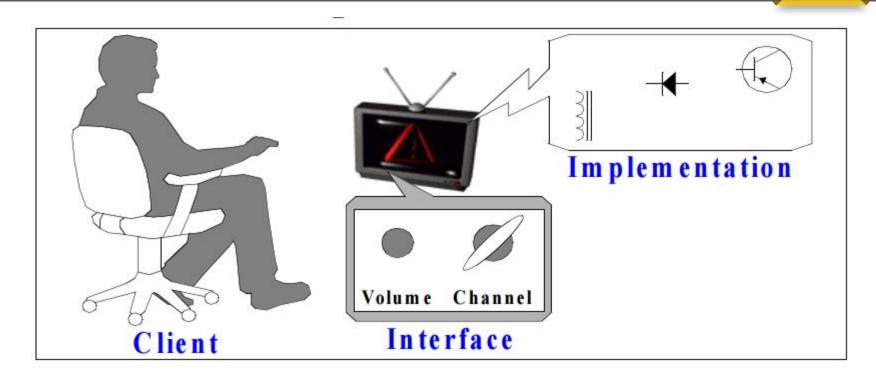
- *Abstract data types* include:
  - Values - V
  - Operations - O
- Focus only on description, not on how to implement details in code.

Abstract data type

Interface

Program

Request to perform operation

Result of operation

add

remove

find

display

Wall of ADT operations

Data structure

Abstract data type

- Each ADT should be like a contract or specification:
  - Define a valid data
  - Identify valid operations:
    - Its name
    - Type of parameters
    - The type of result returned if available
    - Its observable manners
  - Don't identify the following items:
    - Specific datatype
    - Algorithms to implement

Example 1

Client   Interface   Implementation   Volume   Channel

- Separate implementation from specification
  - INTERFACE: specify the allowed operations
  - IMPLEMENTATION: provide code for ops
  - CLIENT: code that uses them

Example 2

- List:
  - Concept:
    - A list is a limited set of elements of the same type.
    - Describe the list as a sequence of its elements: a_1, a_2, . . ., a_n with $n \geq 0$.
      - If n=0, we say the list is empty.
      - If $n > 0$, we call a_1 the first and most secure elements of the list.
      - Number of elements of a list we call the length of a list.
  - How to define ADT for List?

Example 2

- ADT definitions for list:
  - Data: set of elements of the same type
  - Operations:
    - **insertList(x, p, L)**: add element x to position p in the L list
    - **locate(x, L)**: locate element x in the L list
    - **retrieve(p, L)**: get the element in position p
    - **deleteList(p,L)**: delete an element in position p
    - **next(p, L)**: take the next element of position p
    - **previous(p,L)**: get the previous element of position p
    - **first(L)**: get the first one

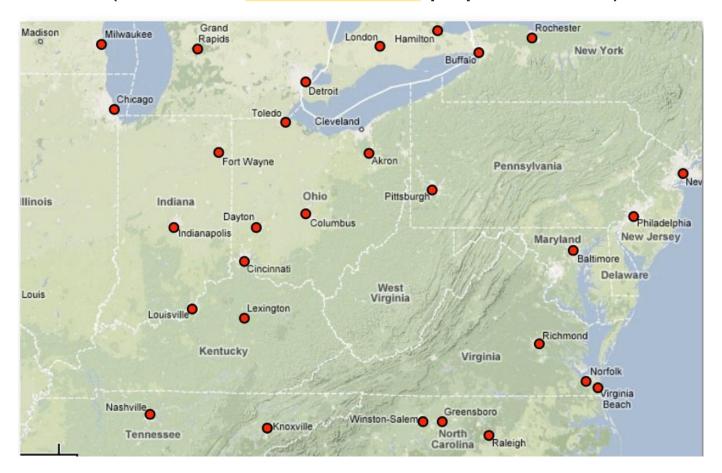- Create ADT for:

Student

Class of Students
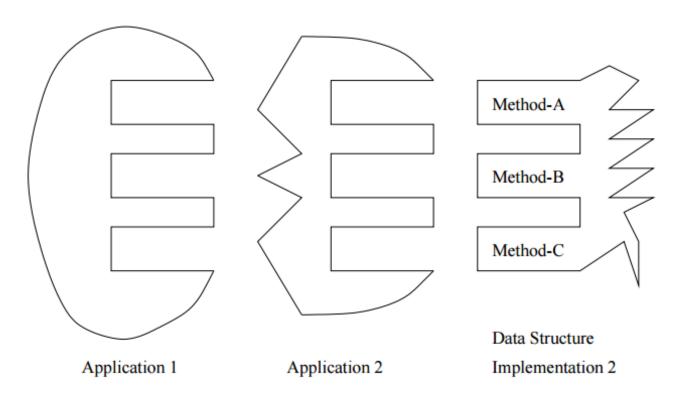
- Let's say you create Google Map and you're containing data about cities (location, coordinates, population,...)

Which operations can your data structure support?

Why uses ADT?

- Help break down the problem
- Hide execution details
- Reuse

Application 1   Application 2   Method-A

Method-B

Method-C

Data Structure
Implementation 2

Java Interface for ADT Dictionary

```
public interface Dictionary {

    public Object find(Object key);

    public void insert(Object key, Object data)
                        throws DuplicatedKeyException;

    public void remove(Object key)
                        throws NoKeyException;

}
```

Common ADTs

- Common abstract data types:
  - Linked list
  - Stacked
  - Queues
  - Priority Queues
  - Tree
  - Dictionary
  - Hash table
  - Graph…

Common ADTs
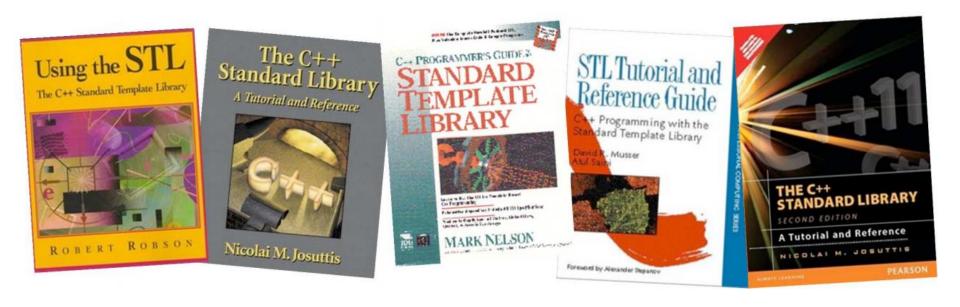
- Some common operations:
  - Returns the specific element
  - Add an element
  - Delete an element
  - Count number of elements
  - Copy
  - Search
  - Sort
  - Get a group of elements
  - ….

Built-in ADTs
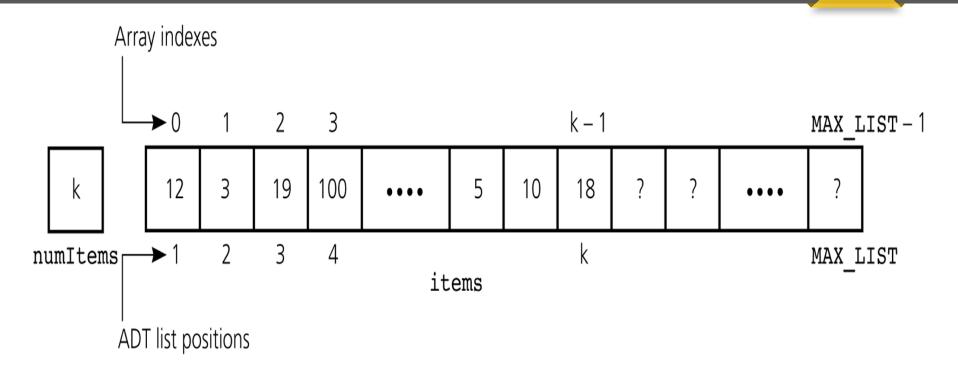
- Programming languages often pre-install command ADTs.
  - Standard Template Library (C++ STL)
  - Java Collections Framework (java.util.Stack, …)

Contents

- Introduction
- **Some Abstract Data Types**
  - Array
  - Linked List
  - Stack
  - Queue
- Application Exercises

Array indexes

0 1 2 3 k – 1 MAX_LIST – 1

| k | 12 | 3 | 19 | 100 | •••• | 5 | 10 | 18 | ? | ? | •••• | ? |

numItems 1 2 3 4 k MAX_LIST

items

ADT list positions

- **Array:**
  - The continuous allocation data structure
  - Fixed size
  - Each element is placed at the specified index / address
- For example, street house with the address is the index

ADT: Array

- *Advantages of array:*
  - Access to the element with constant time: because it only needs to be based on the index to jump to.
  - Space efficiency: due to fixed dimensions.

- *Disadvantages of array:*
  - Cannot resize → deal with dynamic arrays → disadvantage of consuming copy time when allocating more elements.

- *Search complexity:* O(n)
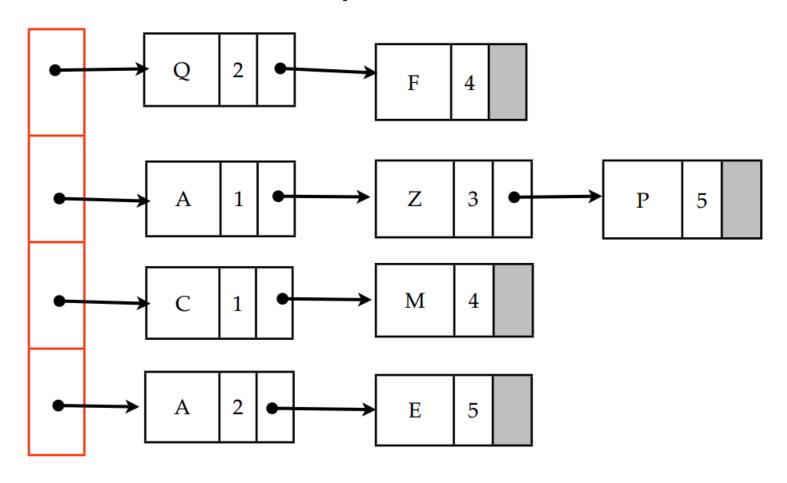
- A lot of positions have no value

$$\begin{pmatrix}
1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\
0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\
0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\
0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\
2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\
0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\
0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0
\end{pmatrix}$$
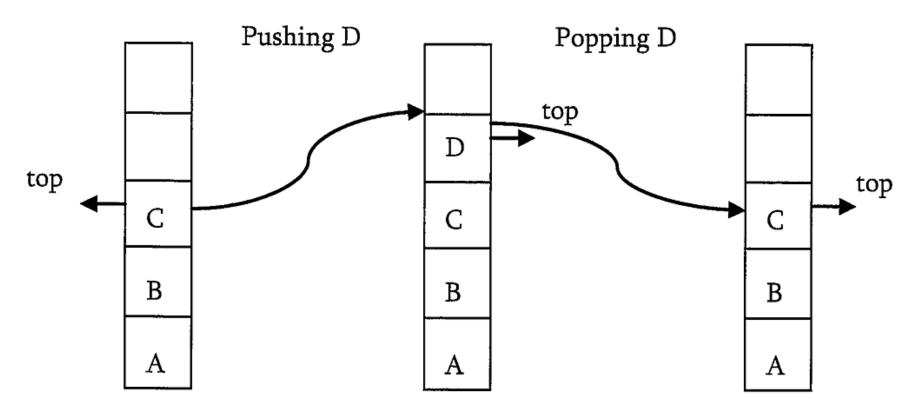
- Use linked list for sparse matrices

ADT: Stack
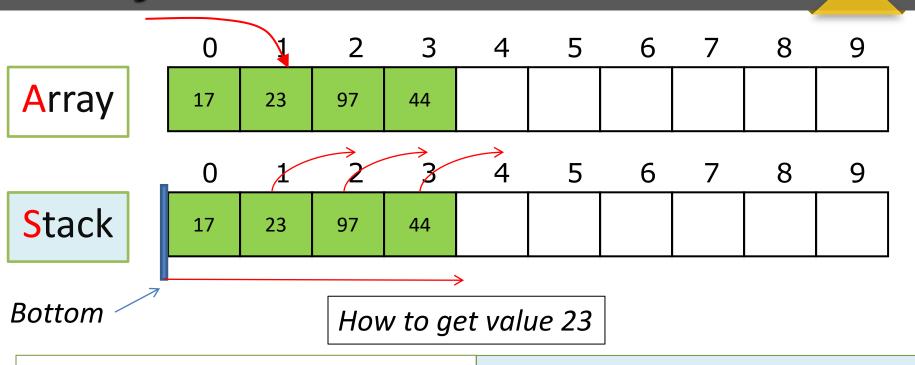
- **Stack**: is an ordered list and elements are entered and retrieved by FILO or LIFO mechanism

Pushing D   Popping D

Introduction to Stack

- **Stack** ADT stores arbitrary objects

- To add (push) an item to the stack, it must be placed on the top of the stack.

- To remove (pop) an item from the stack, it must be removed from the top of the stack too.

- Thus, the last element that is pushed into the stack, is the first element to be popped out of the stack. i.e., **Last In First Out (LIFO)**

- Main stack operations:
  - push: inserts an element
  - pop: removes and returns the last inserted element

Array vs. Stack

	0	1	2	3	4	5	6	7	8	9
Array	17	23	97	44						

	0	1	2	3	4	5	6	7	8	9
Stack	17	23	97	44						

Bottom

How to get value 23
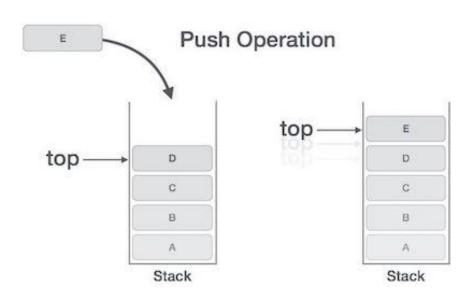
Array:	Stack:
Jump to the target position and perform direct operation. A[1]	- To access element 23, the following elements must be removed from the stack first.

28

Stack operations
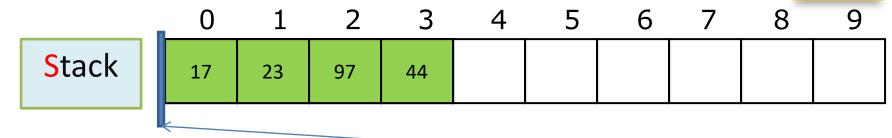
- Basic operations on Stack:
  - Initial
  - Insert an add-in (Push)
  - Take out an (Pop)
  - View top element (Top)
  - Size of Stack (Size)
  - Check Empty Stack (IsEmptyStack)
  - Check Stack Full (IsFullStack)
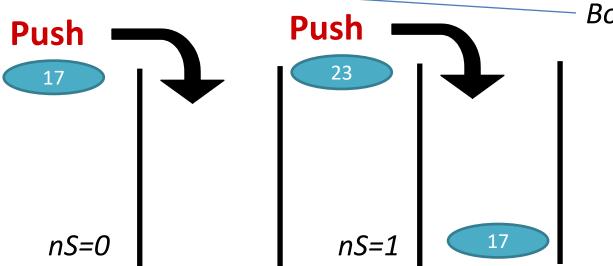
Push Operation
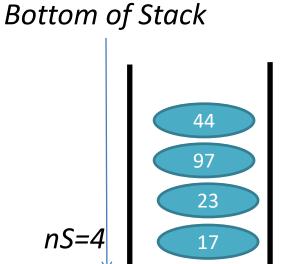
- The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps
  - **Step 1** − Checks if the stack is full.
  - **Step 2** − If the stack is full, produces an error and exit.
  - **Step 3** − If the stack is not full, increments **top** to point next empty space.
  - **Step 4** − Adds data element to the stack location, where top is pointing.
  - **Step 5** − Returns success.

Push Operation

30

	0	1	2	3	4	5	6	7	8	9
Stack	17	23	97	44						

Bottom of Stack

Push

17

Push

23

$nS=0$

17

$nS=1$

44

97

23

17

$nS=4$

```
void Push(int x)
{
        S[nS] = x;
        nS++;
}
```
C++
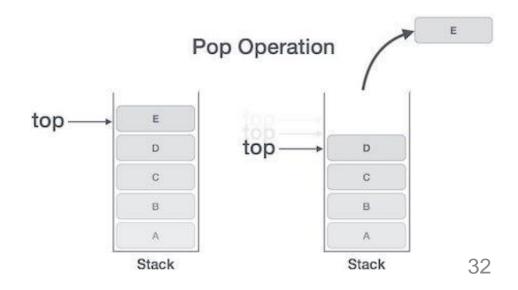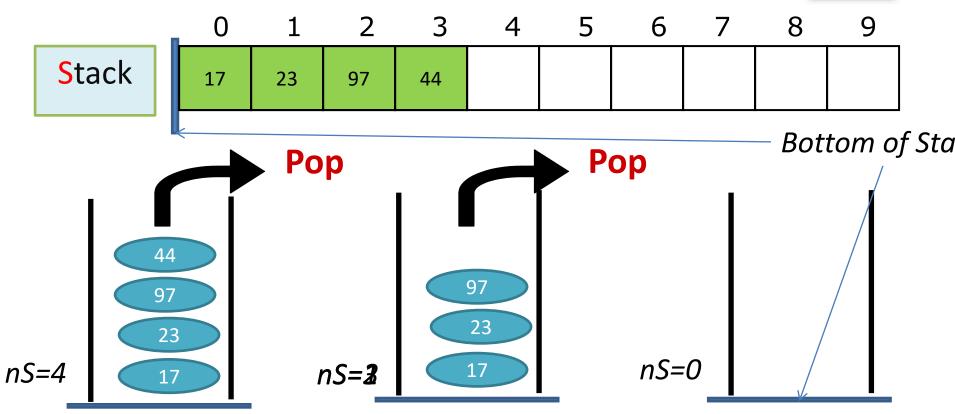
```
public static void Push(int x)
{
        S[nS] = x;
        nS++;
}
```
C#

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

0	1	2	3	4	5	6	7	8	9

Stack

| 17 | 23 | 97 | 44 | | | | | | |

Bottom of Sta

Pop

Pop

44
97
23
17

nS=4

97
23
17

nS=3

nS=0

```
int Pop() {
    if(nS = = 0)        return -1; //empty
    int x = S[nS-1];
    nS--;                                    C++
    return x;          }
```

```
public static int Pop(){
    if(nS = = 0)        return -1; //empty
    int x = S[nS-1];
    nS--;                                    C#
    return x;          }
```

33
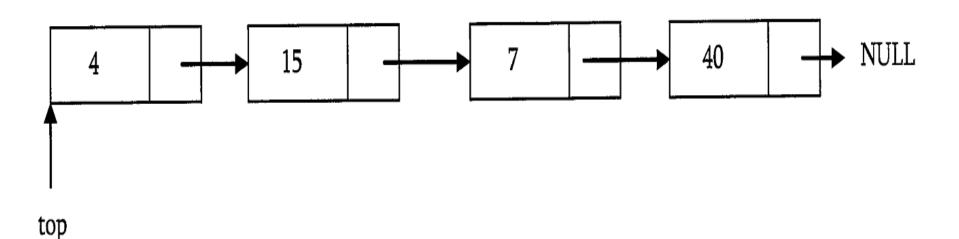
- There are many ways to implement stack:
  - Use arrays
  - Use dynamic arrays (increase in size when stack is full)
  - Use a linked list

Demo

- **Stack applications**:
  - Character balance
  - Converts the infix to a postfix
  - Calculate postfix expressions
  - Execute function call (including recursion)
  - Web visit history (back button)
  - Recover characters in text editor.
  - Matches tags in HTML and XML
  - Tree-browsing algorithm
  - …

1. Check the character balance.

Example	Valid?	Description
(A+B)+(C-D)	Yes	The expression is having balanced symbol
((A+B)+(C-D)	No	One closing brace is missing
((A+B)+[C-D])	Yes	Opening and immediate closing braces correspond
((A+B)+[C-D]}	No	The last closing brace does not correspond with the first opening parenthesis

2. Converts the infix to postfix

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+ABC	AB+C-
(A+B)*C-D	-*+ABCD	AB+C*D-
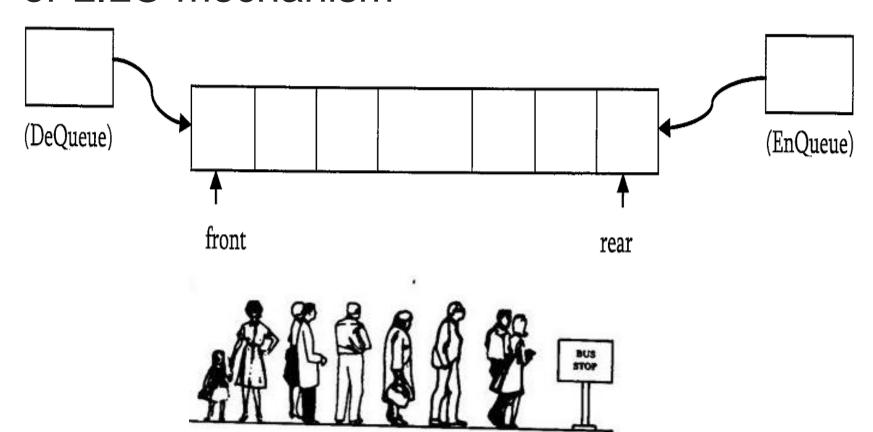
3. Calculate postfix expression (Reverse Polish)

Postfix String : 123*+5-

Result : 2

- *Queue* is an ordered list and elements are entered and retrieved according to the FIFO or LILO mechanism

(DeQueue)   (EnQueue)

front   rear

Queue waiting for a bus.

Queue
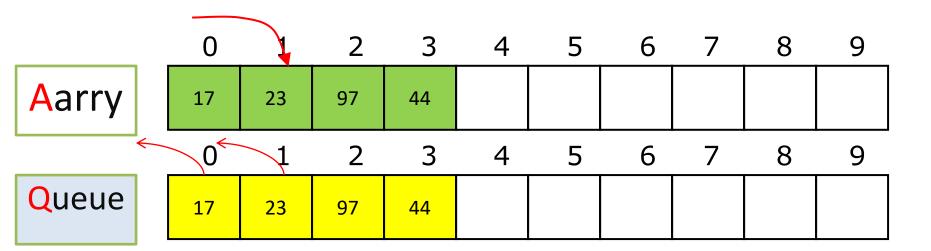
- The Queue ADT stores arbitrary objects

- Insertions and deletions follow the first-in first-out (FIFO) scheme

- Insertions are at the rear and removals are at the front of the queue

Queue operations

- Basic operations on Queue:
  - Initial
  - Insert an enqueue (Enqueue)
  - Removes and returns the element (Dequeue)
  - View the front (Front)
  - Size of Queue (QueueSize)
  - Check Empty Queue (IsEmptyQueue)
  - Check Queue Full (IsFullQueue)

Queue vs Array

	0	1	2	3	4	5	6	7	8	9
Aarry	17	23	97	44						

	0	1	2	3	4	5	6	7	8	9
Queue	17	23	97	44						

How to get the element 23?

Array:
- Jump to the target position and perform direct operation. A[1]
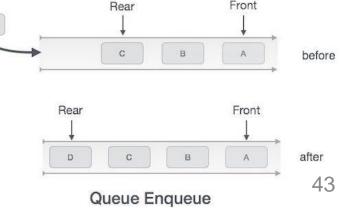
Queue:
- To get access to element 23, the previous elements must be removed from the queue first.
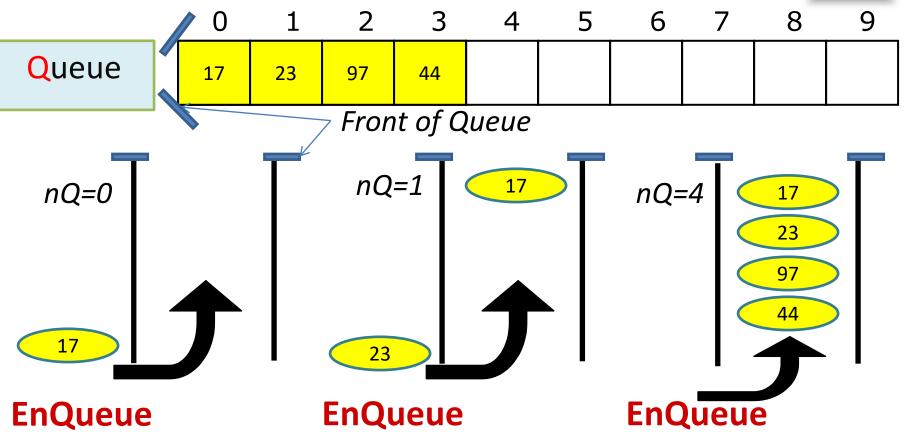
Enqueue Operation

- Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

- The following steps should be taken to enqueue (insert) data into a queue
  - **Step 1** − Check if the queue is full.
  - **Step 2** − If the queue is full, produce overflow error and exit.
  - **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
  - **Step 4** − Add data element to the queue location, where the rear is pointing.
  - **Step 5** − return success

Queue Enqueue

	0	1	2	3	4	5	6	7	8	9
Queue	17	23	97	44						

Front of Queue

nQ=0

17

EnQueue

nQ=1

17

23

EnQueue

nQ=4

17

23

97

44

44

EnQueue

```
void DeQueue(int x)
{
        Q[nQ] = x;          C++
        nQ++;
}
```
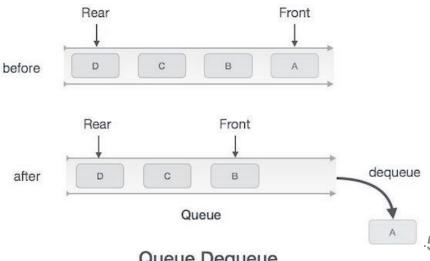
```
public static void EnQueue(int x)
{
        Q[nQ] = x;          C#
        nQ++;
}
```

44

Dequeue Operation
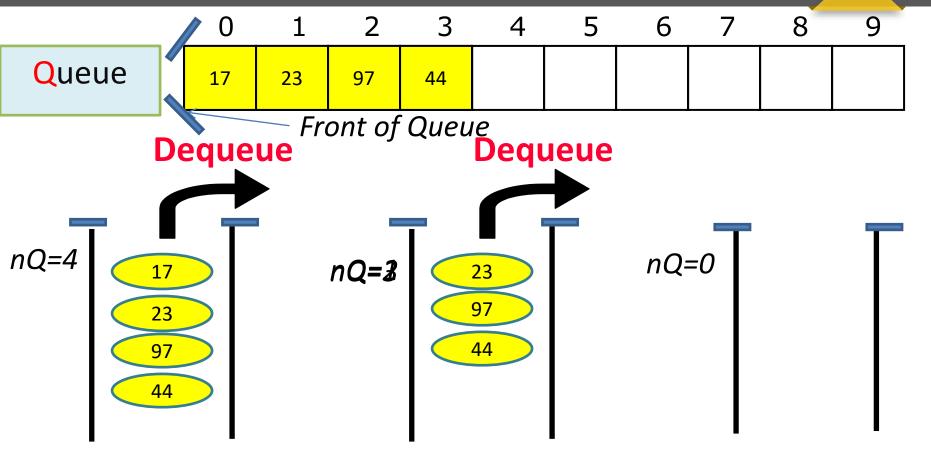
- Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation
  - **Step 1** − Check if the queue is empty.
  - **Step 2** − If the queue is empty, produce underflow error and exit.
  - **Step 3** − If the queue is not empty, access the data where **front** is pointing.
  - **Step 4** − Increment **front** pointer to point to the next available data element.
  - **Step 5** − Return success.

Queue Dequeue

	0	1	2	3	4	5	6	7	8	9
Queue	17	23	97	44						

Front of Queue

Dequeue　　　　　　**Dequeue**

nQ=4

17
23
97
44

nQ=3

23
97
44

nQ=0

```
int DeQueue() {
    if(nQ = = 0)          return -1; //rỗng
    int x = Q[0];
    for (int i = 0; i <n; i ++) Q[i]  = Q[i+1];
    nQ--;
    return x;              }
```
C++

```
public static void DeQueue(int x) {
    if(nQ = = 0)          return -1; //rỗng
    int x = Q[0];
    for (int i = 0; i <n; i ++) Q[i]  = Q[i+1];
    nQ--;
    return x;              }
```
C#
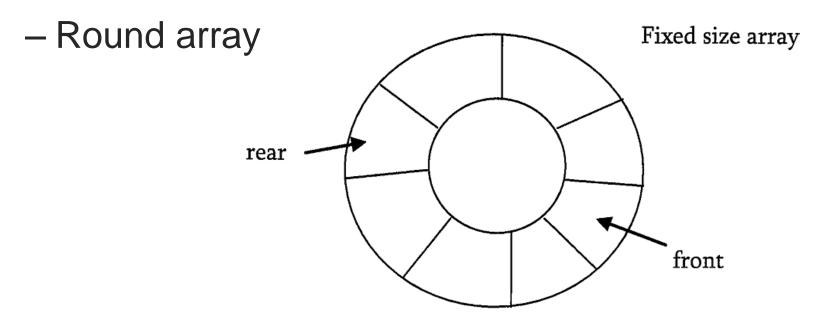
- Similar to Stack, Queue also has many ways to execute such as:
  - Arrays
  - Dynamic array
  - Linked list
  - Round array

Fixed size array

rear

front

Queue applications

- Queue application:
  - Schedule system programs like printing
  - Asynchronous packet transfer
  - Multi-thread
  - Search algorithms
  - ...

Exercise 1

1. Finds the subarray with the maximum sum in the sliding window

Given an array a with a sliding window of size w. The window moves from left to right. Suppose we only see numbers in window w. Each time the window moves sideways one position.

For example: a [1 3 -1 -3 5 3 6 7] and w=3

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Priority Queue

- **Priority_queue** is just like a normal queue except the element removed from the queue is always the greatest among all the elements in the queue

Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.

Priority Queue

nQ=4

Queue 1 (left): 17, 23, 97, 44

Dequeue Max

Dequeue Min

EnQueue

nQ=4

Queue 2 (right): 23, 17, 97, 44

EnQueue

```
public static void DeQueue(int x) {
    if(nQ = = 0)    return -1; //empty
    int x = Q[0];
    for (int i = 0; i <n; i ++) Q[i]  = Q[i+1];
    nQ--;
    return x;          }
```
Queue

```
public static void DePriQueue(int x) {
    if(nPriQ = = 0)         return -1;
    int x = TimMax(PriQ); //TimMin…
    for (int i = 0; i <n; i ++)
            PriQ[i]  = PriQ[i+1];
    nPriQ--;
    return x;          }
```
Priority Queue
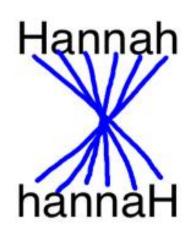
Applications of priority queues

- Priority queues are used in many computing applications.

- For example, many operating systems used a scheduling algorithm where the next process executed is the one with the shortest execution time or the highest priority.

- For another example, consider the problem of sorting a file of data representing persons.
  - We can use a priority queue to sort the data by:
    1. first adding all of the persons to a queue,
    2. sort data in certain order according to the specified priority, and then
    3. removing them in turn from the priority queue (Front element first)

Contents

- Introduction
- Some Abstract Data Types
  - Array
  - Linked List
  - Stack
  - Queue
- **Application Exercises**

Palindromes

- A **palindrome** is a word, phrase, number, or other sequence of characters which reads the same backward as forward, such as madam or racecar.

"Madam, I'm Adam."

"Enid and Edna dine."
"A man, a plan, a canal – Panama!"

Notes: Capitalization, spacing, and punctuation are usually ignored.

Evaluating Expressions

$$(5+9)*2+6*5$$

- An ordinary arithmetical expression like the above is called infix-expression -- binary operators appear in between their operands.

- The order of operations evaluation is determined by the precedence rules and parenthesis.

Evaluating Postfix Expression

- Expressions can also be represented using **postfix notation** - where an operator comes after its two operands.

- The advantage of postfix notation is that the order of operation evaluation is unique without the need for precedence rules or parenthesis.

Infix	Postfix
16 / 2	16 2 /
(2 + 14)* 5	2 14 + 5 *
2 + 14 * 5	2 14 5 * +
(6 – 2) * (5 + 4)	6 2 - 5 4 + *

Why Postfix?

- Does not require parentheses!

- Some calculators make you type in that way

- Easy to process by a program

- The processing algorithm uses a stack for operands (data)
  - simple and efficient

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 -

Postfix Notation is known as Reversed Polish Notation.

- Convert infix to postfix, then apply postfix evaluation algorithm.

Infix to Postfix

- Example :
- (a) Infix: 2 + 3 − 4   Postfix: 2 3 + 4 −

- (b) Infix: 2 + 3 * 4   Postfix: 2 3 4 * +

- The operators of the same priority appear in the same order, operator with higher priority appears before the one with lower priority.

- Rule: hold the operators in a stack, and when a new operator comes, push it on the stack if it has higher priority. Else, pop the stack off and move the result to the output until the stack is empty or an operator with a lower priority is reached. Then, push the new operator on the stack.

Infix to Postfix

- Algorithm:
  - Read a token
  - If operand, output (enqueue) to queue
  - If '(', push the '(' on stack
  - If operator:
    - if stack top is an op of >= precedence: pop and output to queue until '(' is on top or stack empty (need updated!!!!!) ➔ wrong
    - push the new operator
  - If ')', pop and output until '(' has been popped
  - Repeat until end of input
    - pop rest of stack

Operand1   Operand2   Operator

ab+

Infix to Postfix

- Applying the correct rules on when to pop
  - Assign a priority: * and / have a higher priority than + and –
  - Suppose st.top() is + and next token is *, then * is pushed on the stack.
  - However, "(" behaves differently. When it enters the stack, it has the highest priority since it is pushed on top no matter what is on stack. However, once it is in the stack, it allows every symbol to be pushed on top.

Operator precedence

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

Dealing with parentheses

- An opening parenthesis is pushed on the stack (always). It is not removed until a matching right parenthesis is encountered. At that point, the stack is popped until the matching (is reached.

Example: (a + b * c + d)* a
Stack: (Output to queue: a
Stack: (+   Output to queue : a
Stack: (+   Output to queue : a b
Stack: (+ *   Output to queue : a b
Stack: (+ *   Output to queue : a b c
Stack: (+ +   Output to queue : a b c *
Stack: (+   Output to queue : a b c * + d
Stack   Output to queue : a b c * + d +
Stack *   Output to queue : a b c * + d +
Stack *   Output to queue : a b c * + d + a
Stack   Output to queue : a b c * + d + a *

More on Postfix

- **3 4 5 * -** means same as (3 (4 5 *) -)
  - infix: **3 - (4 * 5)**
- Parentheses aren't needed!
  - When you see an operator:
    - both operands must already be available.
    - Stop and apply the operator, then go on
- Precedence is implicit
  - Do the operators in the order found, period!
- Practice converting:
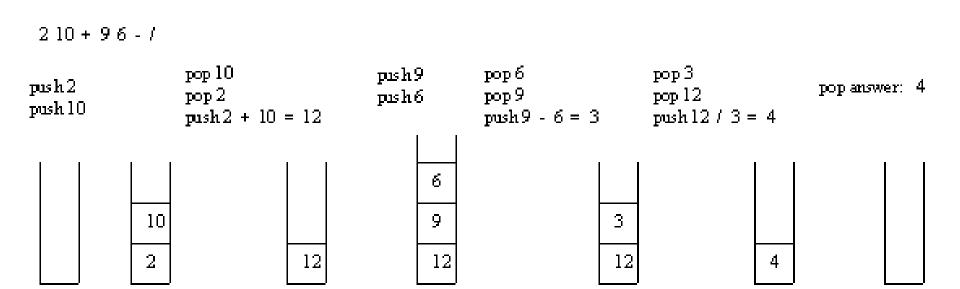 - 1 2 + 7 * 2 %
  - (3 + (5 / 3) * 6) - 4

Evaluating Postfix Expression

- The following algorithm uses a stack to evaluate a postfix expressions.
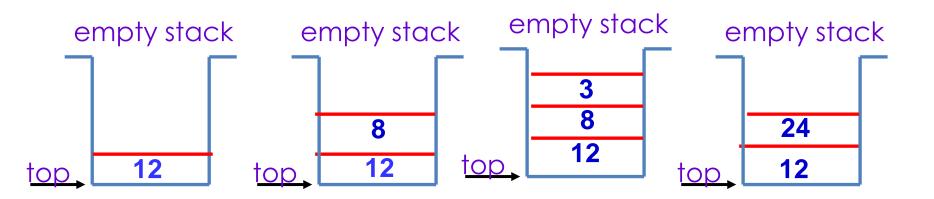
```
Start with an empty stack
for (each item in the expression) {
        if (the item is a number)
            Push the number onto the stack
        else if (the item is an operator){
            Pop two operands from the stack
          Apply the operator to the operands
            Push the result onto the stack
        }
}
Pop the only number from the stack – that's the result of evaluation
```

Evaluating Postfix Expression
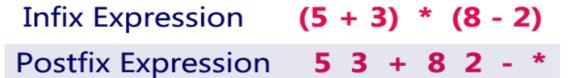
- Example: Consider the postfix expression, **2 10 + 9 6 - /,** which is **(2 + 10) / (9 - 6)** in infix, the result of which is 12 / 3 = 4.

- The following is a trace of the postfix evaluation algorithm for the above.

2 10 + 9 6 - /

push 2
push 10

pop 10
pop 2
push 2 + 10 = 12

push 9
push 6

pop 6
pop 9
push 9 - 6 = 3

pop 3
pop 12
push 12 / 3 = 4

pop answer: 4

10		
2		

	12

6
9
12

3
12

4

- 12 8 3 * +

empty stack   empty stack   empty stack   empty stack

top	12
top	8 / 12
top	3 / 8 / 12
top	24 / 12
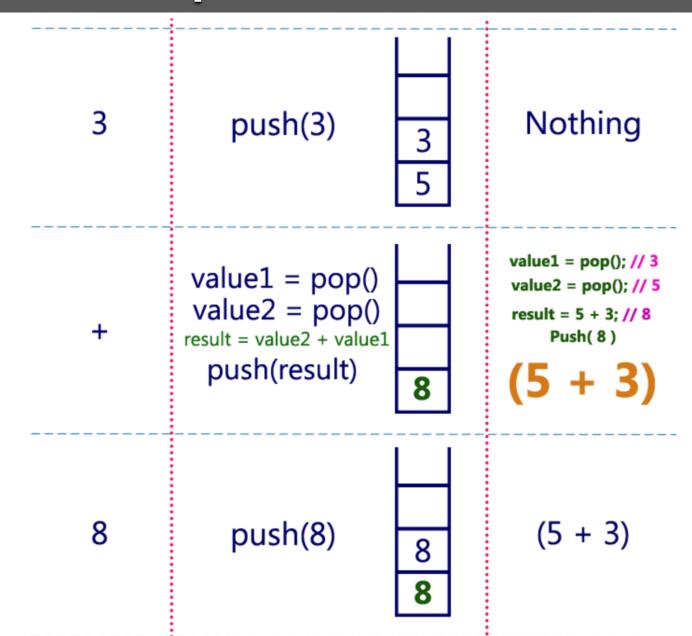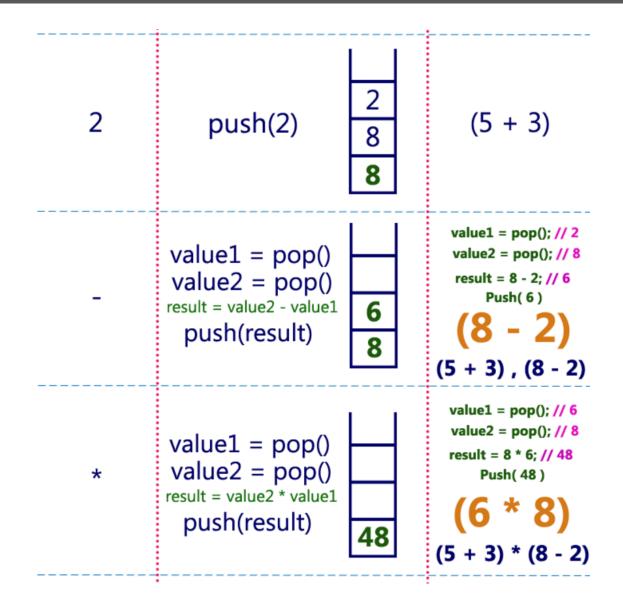
Finally, the result 36 is pushed back on the stack.

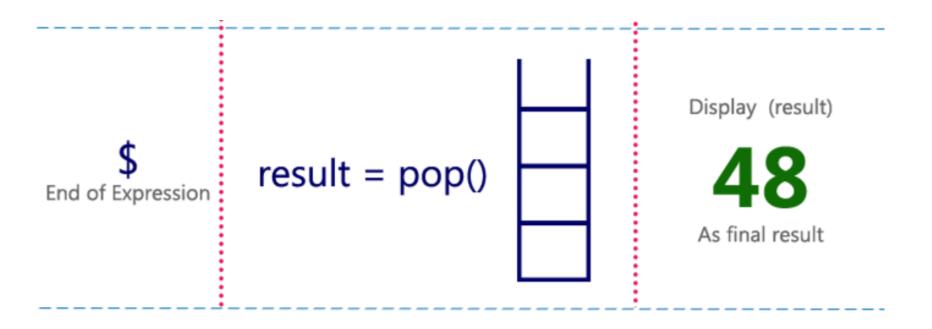| Infix Expression	(5 + 3) * (8 - 2)
Postfix Expression	5 3 + 8 2 - *

Above Postfix Expression can be evaluated by using Stack Data Structure as follows...

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty	Nothing
5	push(5)	Nothing

2	push(2)	2 8 **8**	(5 + 3)

-	value1 = pop()
	value2 = pop()
	result = value2 - value1
	push(result)
		6
		8
			value1 = pop(); // 2
			value2 = pop(); // 8
			result = 8 - 2; // 6
			Push(6)
			(8 - 2)
			(5 + 3) , (8 - 2)

*	value1 = pop()
	value2 = pop()
	result = value2 * value1
	push(result)
		48	value1 = pop(); // 6
			value2 = pop(); // 8
			result = 8 * 6; // 48
			Push(48)
			(6 * 8)
			(5 + 3) * (8 - 2)

$
End of Expression

result = pop()

Display (result)

48

As final result

Infix Expression **(5 + 3)** * **(8 - 2)** = **48**

Postfix Expression **5 3 + 8 2 - *** value is **48**

The End.