Data Structures
COMP242

Ala' Hasheesh
ahashesh@birzeit.edu

Stacks

Stack

A **stack** is a data structure in which elements are added and removed from one end!

Stack

A **stack** is a data structure in which elements are added and removed from one end!

It can allow us to insert elements and retrieve them in opposite order!

Stack

A **stack** is a data structure in which elements are added and removed from one end!

It can allow us to insert elements and retrieve them in opposite order!

Operations

1. Push: Insert/Add a new element to **top** the stack
2. Pop: Remove/Delete **top** element in the stack
3. Peek: Examine element at the **top** of the stack

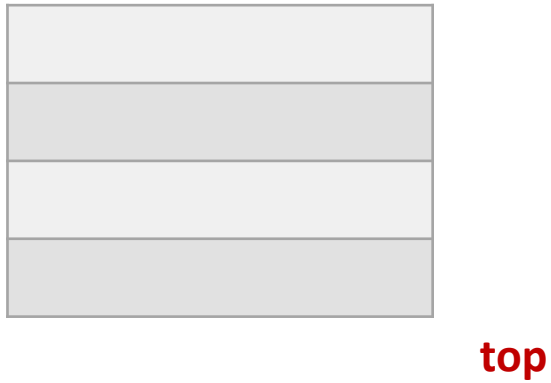
Stack

A **stack** is a data structure in which elements are added and removed from one end!

It can allow us to insert elements and retrieve them in opposite order!

Operations

1. Push: Insert/Add a new element to **top** the stack
2. Pop: Remove/Delete **top** element in the stack
3. Peek: Examine element at the **top** of the stack



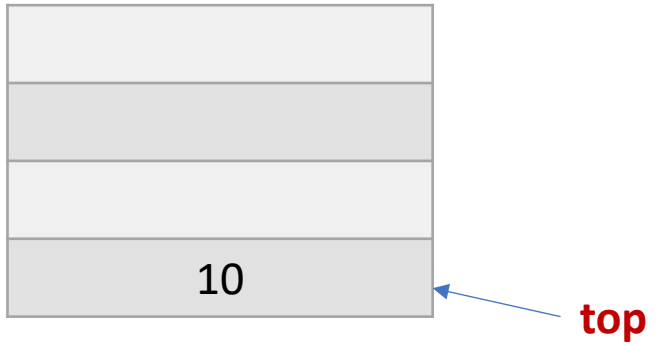
Stack

A **stack** is a data structure in which elements are added and removed from one end!

It can allow us to insert elements and retrieve them in opposite order!

Operations

1. Push: Insert/Add a new element to **top** the stack
2. Pop: Remove/Delete **top** element in the stack
3. Peek: Examine element at the **top** of the stack



stack.push(10);

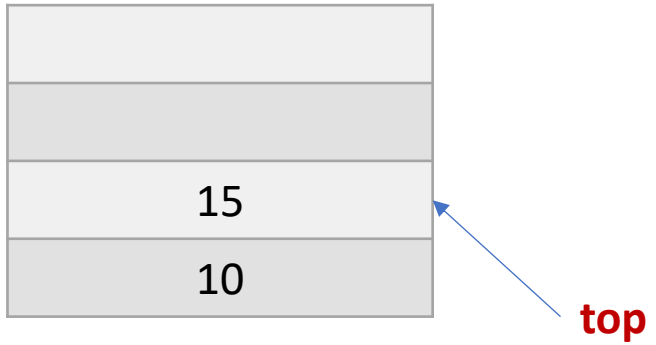
Stack

A **stack** is a data structure in which elements are added and removed from one end!

It can allow us to insert elements and retrieve them in opposite order!

Operations

1. Push: Insert/Add a new element to **top** the stack
2. Pop: Remove/Delete **top** element in the stack
3. Peek: Examine element at the **top** of the stack



stack.push(15);

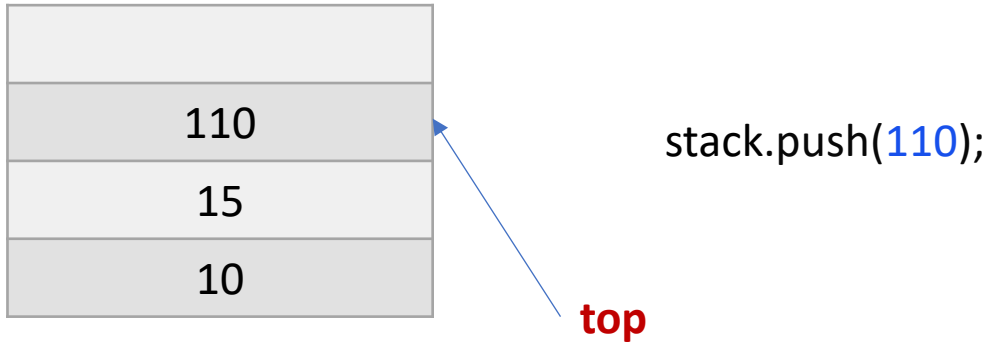
Stack

A **stack** is a data structure in which elements are added and removed from one end!

It can allow us to insert elements and retrieve them in opposite order!

Operations

1. Push: Insert/Add a new element to **top** the stack
2. Pop: Remove/Delete **top** element in the stack
3. Peek: Examine element at the **top** of the stack



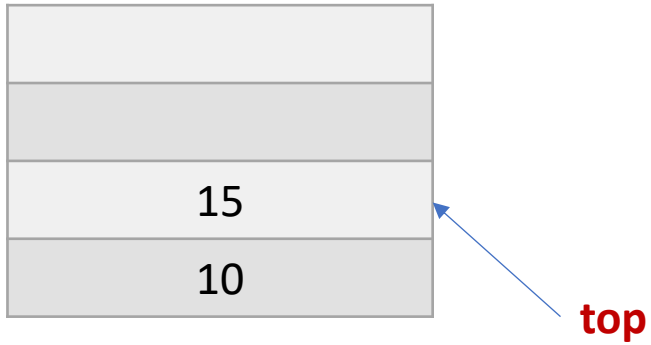
Stack

A **stack** is a data structure in which elements are added and removed from one end!

It can allow us to insert elements and retrieve them in opposite order!

Operations

1. Push: Insert/Add a new element to **top** the stack
2. Pop: Remove/Delete **top** element in the stack
3. Peek: Examine element at the **top** of the stack



stack.pop();

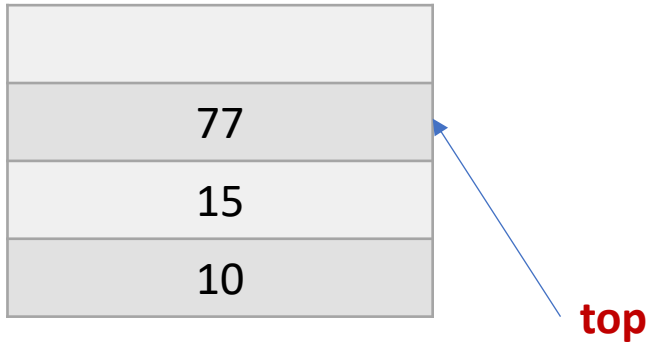
Stack

A **stack** is a data structure in which elements are added and removed from one end!

It can allow us to insert elements and retrieve them in opposite order!

Operations

1. Push: Insert/Add a new element to **top** the stack
2. Pop: Remove/Delete **top** element in the stack
3. Peek: Examine element at the **top** of the stack

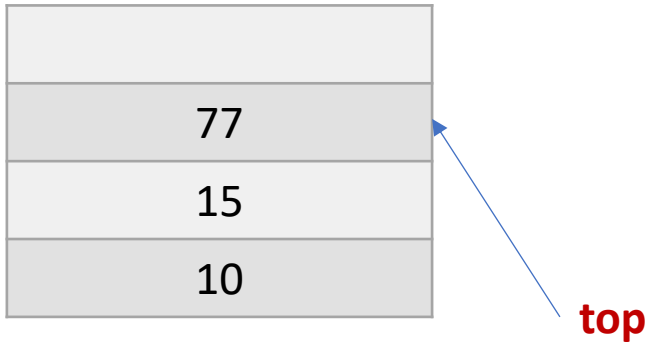


stack.push(77);

Stack

Stacks are called (**LIFO**): Last In First Out

Last element that gets pushed is the first element that gets popped!



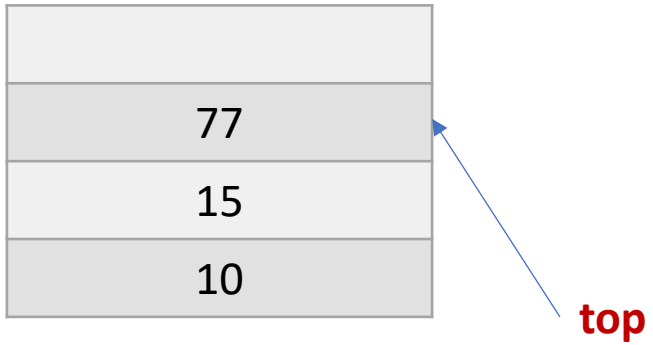
Stack

Stacks are called (**LIFO**): Last In First Out

Last element that gets pushed is the first element that gets popped!

They are also called (**FILO**): First In Last Out

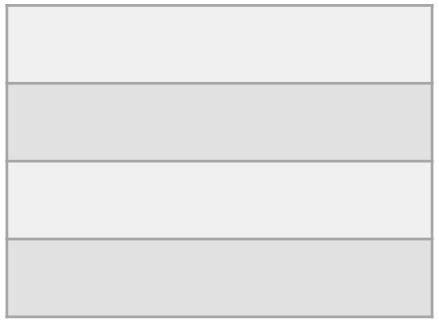
First element that gets pushed is the last element that gets popped!



Stack (Applications)

1. Method calls.

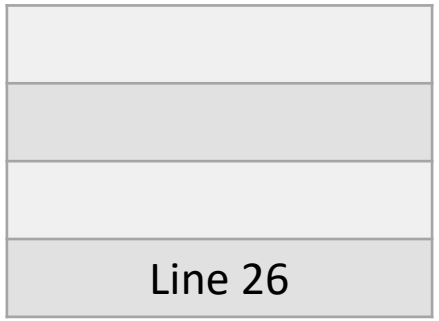
```
private static void m2() {
    return;
}

private static void m1() {
    m2();
    System.out.println(”21");   // Line 21
}

private static void run() {
    m1();
    System.out.println("26");   // Line 26
}
```



Stack (Applications)

1. Method calls.

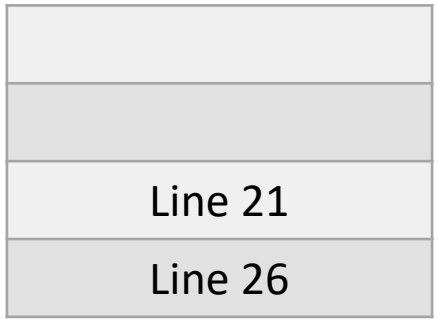
```
private static void m2() {
   return;
}

private static void m1() {
   m2();
   System.out.println(”21");   // Line 21
}

private static void run() {
   m1();
   System.out.println("26");   // Line 26
}
```



Stack (Applications)

1. Method calls.

```
private static void m2() {
   return;
}

private static void m1() {
   m2();
   System.out.println(”21");   // Line 21
}

private static void run() {
   m1();
   System.out.println("26");   // Line 26
}
```



Stack (Applications)

1. Method calls.
2. Evaluating some expressions (used by compilers)
3. Check if string is palindrome
4. Check if brackets match in an expression
   1 + 2 * (3 + 5
   5 + (6 + 7)

5. Undo stack in applications!
6. Many others!

Stack (Array Implementation)

```
public class Stack<T> {
    private int capacity;
    private int top;
    T[] array;

    public Stack() {
        this(10);
    }

    public Stack(int capacity) {
        this.capacity = capacity;
        this.top = -1;
        array = (T[]) new Object[capacity];
    }
}
```

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

top = -1

```
Stack<Integer> stack = new Stack<>();
```

Stack (Array Implementation)

```
public boolean isEmpty() {
    return top == -1;
}
```

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

top = -1

```
Stack<Integer> stack = new Stack<>();
```

Stack (Array Implementation)

```
public boolean isEmpty() {
    return top == -1;
}

public boolean isFull() {
    return top + 1 == capacity;
}
```

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

top = -1

```
Stack<Integer> stack = new Stack<>();
```

Stack (Array Implementation)

```
public boolean isEmpty() {
    return top == -1;
}

public boolean isFull() {
    return top + 1 == capacity;
}

public void push(T element) {
    if (!isFull()) {
        array[++top] = element;
    }
}
```

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

top = -1

```
Stack<Integer> stack = new Stack<>();
```

Stack (Array Implementation)

```
public boolean isEmpty() {
    return top == -1;
}

public boolean isFull() {
    return top + 1 == capacity;
}

public void push(T element) {
    if (!isFull()) {
        array[++top] = element;
    }
}
```

stack.push(10);

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	10

top = 0

Stack<Integer> stack = new Stack<>();

Stack (Array Implementation)

```
public boolean isEmpty() {
    return top == -1;
}

public boolean isFull() {
    return top + 1 == capacity;
}

public void push(T element) {
    if (!isFull()) {
        array[++top] = element;
    }
}
```

stack.push(17);

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	17
0	10

top = 1

Stack<Integer> stack = new Stack<>();

Stack (Array Implementation)

```
public T pop() {
    if (isEmpty()) {
        return null;
    }

    return array[top--];
}
```

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	17
0	10

top = 1

Stack<Integer> stack = new Stack<>();

Stack (Array Implementation)

```
public T pop() {
    if (isEmpty()) {
        return null;
    }

    return array[top--];
}

public T peek() {
    if (isEmpty()) {
        return null;
    }

    return array[top];
}
```

capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	17
0	10

top = 1

```
Stack<Integer> stack = new Stack<>();
```

Stack (Array Implementation)

```
public T pop() {
   if (isEmpty()) {
      return null;
   }

   return array[top--];
}

public T peek() {
   if (isEmpty()) {
      return null;
   }

   return array[top];
}
```

stack.pop();

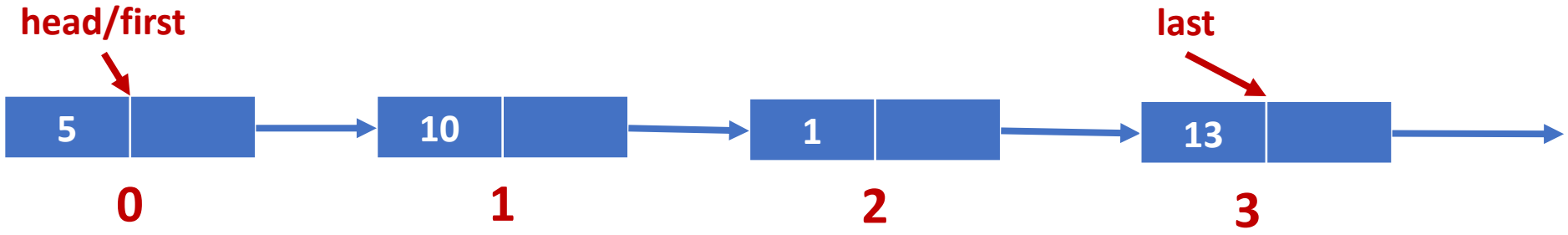
capacity = 10

9	
8	
7	
6	
5	
4	
3	
2	
1	
0	10

top = 0

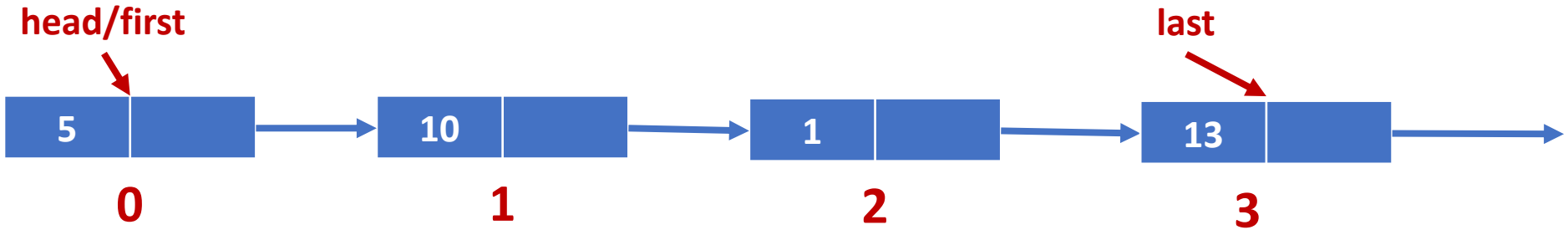
Stack<Integer> stack = new Stack<>();

Stack (LinkedList)



Where is the top?

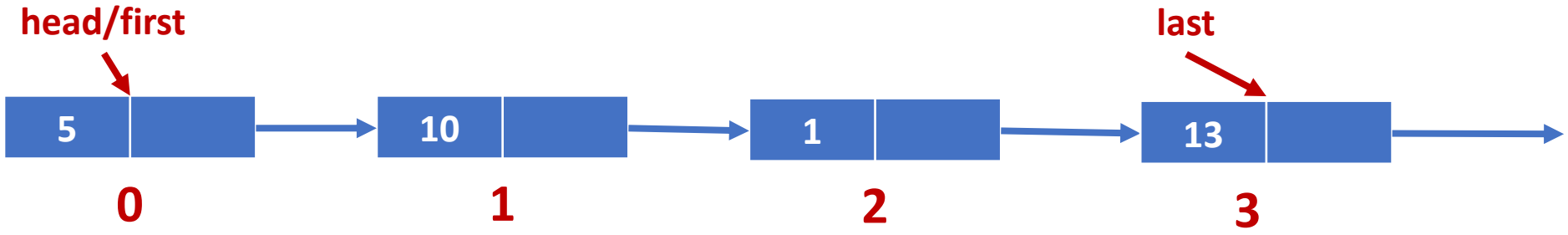
Stack (LinkedList)



Where is the top?

Assume that top = last

Stack (LinkedList)

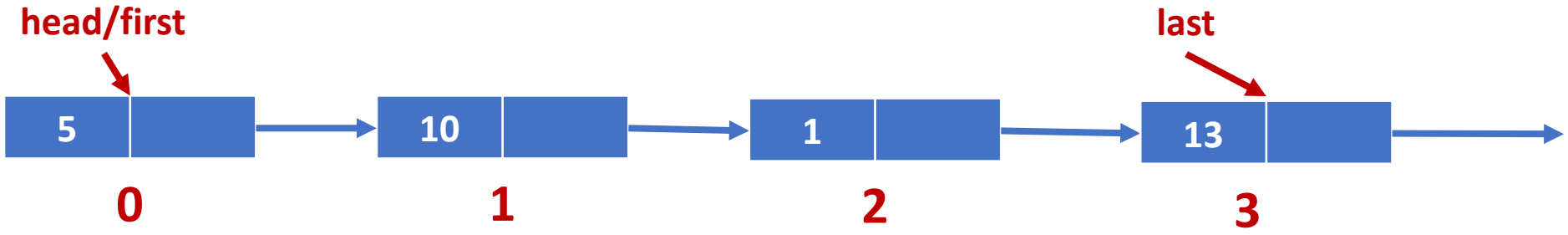


Where is the top?

Assume that top = last

push -> addLast(); *// O(1)*

Stack (LinkedList)



Where is the top?

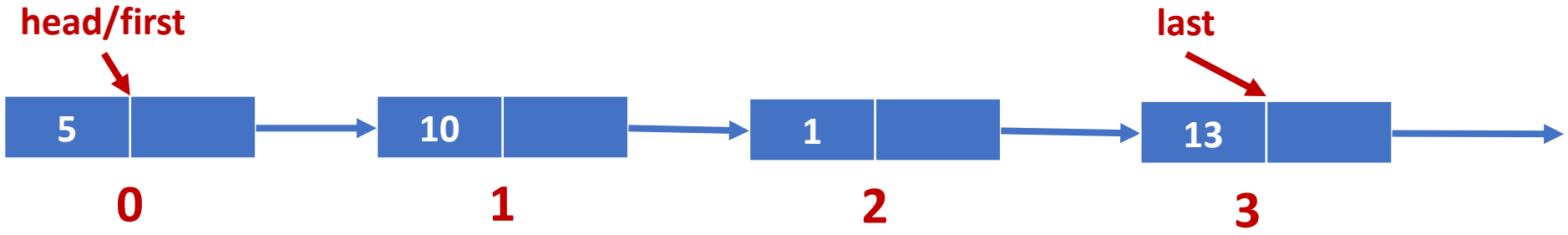
Assume that top = last

push -> addLast(); *// O(1)*

pop -> removeLast(); *// O(n)*

This is valid because it's **LIFO** (Last In First Out)

Stack (LinkedList)



Where is the top?

Assume that top = first

push -> addFirst(); *// O(1)*

pop -> removeFirst(); *// O(1)*

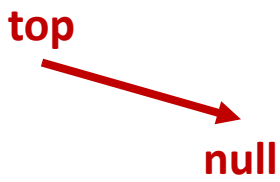
This is valid because it's **LIFO** (Last In First Out)

Stack (LinkedList Implementation)

```
public class Node<T> {
    public T val;
    public Node<T> next;

    public Node(T val) {
        this(val, null);
    }

    public Node(T val, Node<T> next) {
        this.val = val;
        this.next = next;
    }
}
```

Stack (LinkedList Implementation)

```
public class Stack<T> {
    int size = 0;

    Node<T> top;

    public Stack() {
        top = null;
        size = 0;
    }
}
```



Stack (LinkedList Implementation)

```
public boolean isEmpty() {
   return size == 0;
}

public int size() {
   return size;
}
```

size = 0

Stack (LinkedList Implementation)

```
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

public void push(T element) {
    Node<T> node = new Node<>(element);
    node.next = top;
    top = node;
    size++;
}
```

size = 0

Stack (LinkedList Implementation)

```
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

public void push(T element) {
    Node<T> node = new Node<>(element);
    node.next = top;
    top = node;
    size++;
}
```

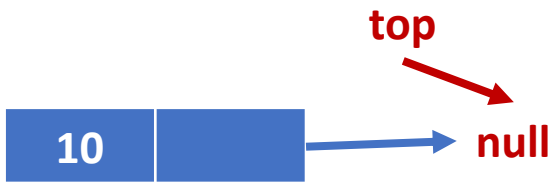
push(10);

size = 0

Stack (LinkedList Implementation)

```
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

public void push(T element) {
    Node<T> node = new Node<>(element);
    node.next = top;
    top = node;
    size++;
}
```

push(10);



size = 0

Stack (LinkedList Implementation)

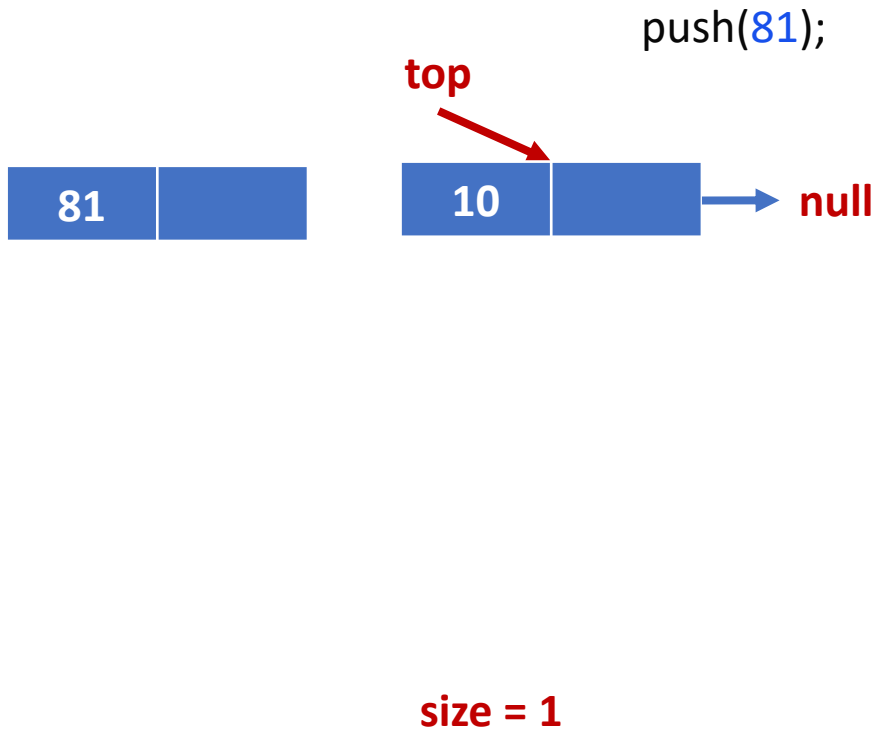
```
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

public void push(T element) {
    Node<T> node = new Node<>(element);
    node.next = top;
    top = node;
    size++;
}
```

push(10);

size = 1

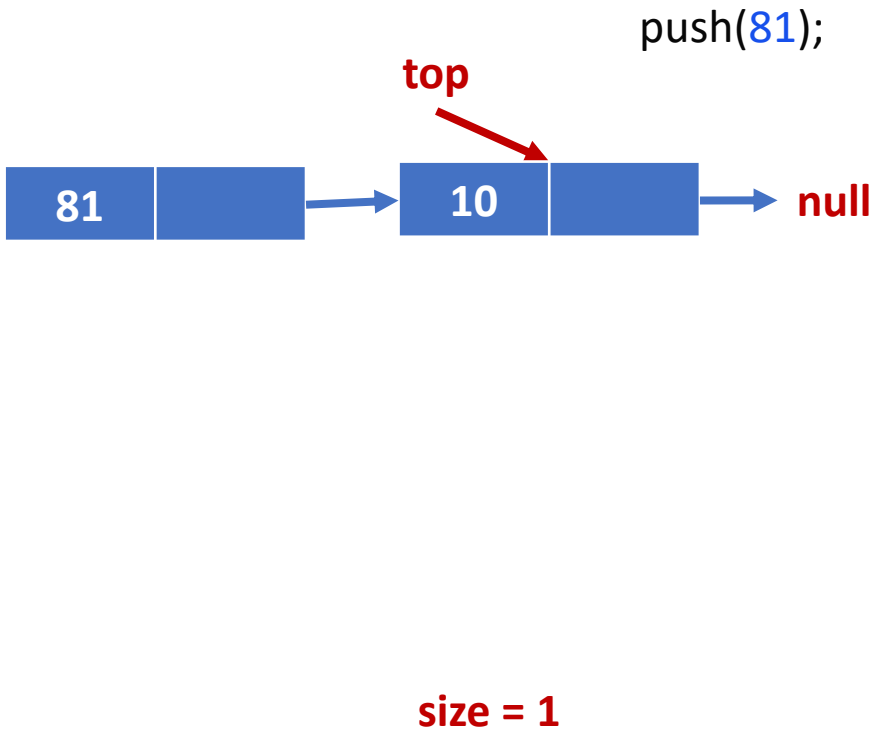
Stack (LinkedList Implementation)

```
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

public void push(T element) {
    Node<T> node = new Node<>(element);
    node.next = top;
    top = node;
    size++;
}
```



Stack (LinkedList Implementation)

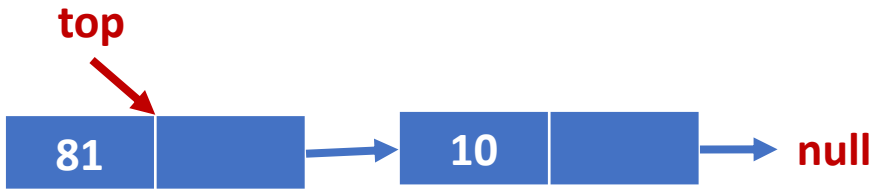
```
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

public void push(T element) {
    Node<T> node = new Node<>(element);
    node.next = top;
    top = node;
    size++;
}
```



Stack (LinkedList Implementation)

```
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

public void push(T element) {
    Node<T> node = new Node<>(element);
    node.next = top;
    top = node;
    size++;
}
```

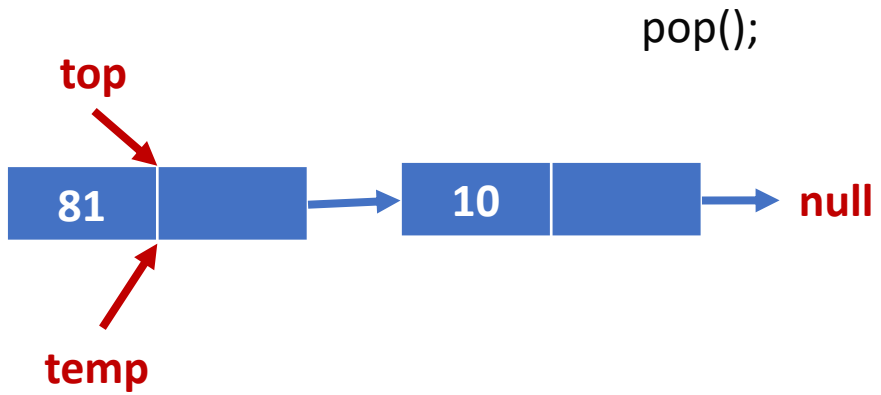
push(81);



size = 2

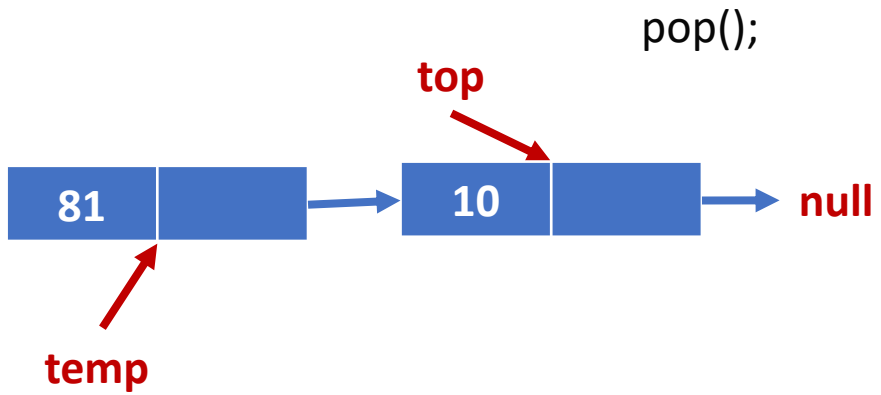
Stack (LinkedList Implementation)

```
public T pop() {
    if (top == null) {
        return null;
    }

    Node<T> temp = top;
    top = top.next;
    temp.next = null; // Not needed
    size--;

    return temp.data;
}
```



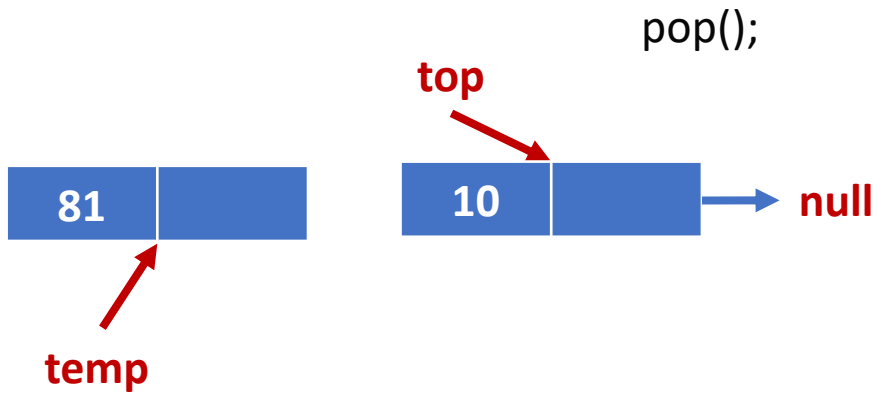
Stack (LinkedList Implementation)

```
public T pop() {
    if (top == null) {
        return null;
    }

    Node<T> temp = top;
    top = top.next;
    temp.next = null; // Not needed
    size--;

    return temp.data;
}
```



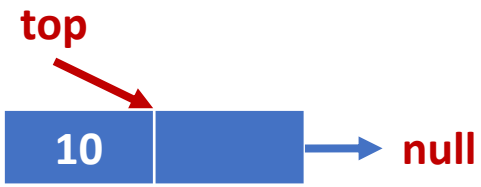
Stack (LinkedList Implementation)

```
public T pop() {
    if (top == null) {
        return null;
    }

    Node<T> temp = top;
    top = top.next;
    temp.next = null; // Not needed
    size--;

    return temp.data;
}
```



Stack (LinkedList Implementation)

```
public T peek() {
    if (top == null) {
        return null;
    }

    return top.data;
}
```



size = 1

Stack (LinkedList Implementation)

How to loop on stack?

Stack (LinkedList Implementation)

How to loop on stack?

```
while (!stack.isEmpty()) {
    int x = stack.pop();
    // Do something with x
}

// stack is empty here!
```

Stack (Exercise)

You have a stack of random integers!

```
Random random = new Random();
Stack<Integer> stack = new Stack<>();
for (int i = 0; i < 10; i++) {
    stack.push(random.nextInt());
}
```

9	**587**
8	473
7	315
6	94
5	37
4	63
3	1025
2	1
1	77
0	10

Stack (Exercise)

You have a stack of random integers!

```
Random random = new Random();
Stack<Integer> stack = new Stack<>();
for (int i = 0; i < 10; i++) {
    stack.push(random.nextInt());
}
```

1. Find max in stack without destroying it!

```
int max = 0;
while (!stack.isEmpty()) {
    max = Math.max(max, stack.pop());
}

// stack is empty here! We don't want that!
```

9	**587**
8	473
7	315
6	94
5	37
4	63
3	1025
2	1
1	77
0	10

Stack (Exercise)

You have a stack of random integers!

```
Random random = new Random();
Stack<Integer> stack = new Stack<>();
for (int i = 0; i < 10; i++) {
    stack.push(random.nextInt());
}
```

1. Find max in stack without destroying it!

```
int max = 0;
while (!stack.isEmpty()) {
    max = Math.max(max, stack.pop());
}

// stack is empty here! We don't want that!
```

Hint: Use Recursion

9	**587**
8	473
7	315
6	94
5	37
4	63
3	1025
2	1
1	77
0	10

Stack (Exercise)

You have a stack of random integers!

```
Random random = new Random();
Stack<Integer> stack = new Stack<>();
for (int i = 0; i < 10; i++) {
    stack.push(random.nextInt());
}
```

1. Find max in stack without destroying it!
2. Print stack without destroying it!

Hint: Use Recursion

9	**587**
8	473
7	315
6	94
5	37
4	63
3	1025
2	1
1	77
0	10

Stack (Application - Infix to postfix)

Infix Notation

We usually use Infix notation as our conventional notation to express our mathematical equations!

1. Operands (i.e. Numbers, Variables and so on) come before operations!
2. Operators appear each two operands (e.g. “2 + 5”)
3. We use brackets to assign priority! (e.g. “(2 + 5) * 7”)

Stack (Application - Infix to postfix)

Infix Notation

We usually use Infix notation as our conventional notation to express our mathematical equations!

1. Operands (i.e. Numbers, Variables and so on) come before operations!
2. Operators appear each two operands (e.g. “2 + 5”)
3. We use brackets to assign priority! (e.g. “(2 + 5) * 7”)

2 + 5 * 7 = 37
(2 + 5) * 7 = 49

Stack (Application - Infix to postfix)

postfix Notation

post notation is easier to evaluate because we don’t deal with brackets!

1. Operators appear after two operands (e.g. “2 5 +”)
2. To evaluate an operator, we apply it to the previous two numbers!

   2 5 + 11 *

Stack (Application - Infix to postfix)

postfix Notation

post notation is easier to evaluate because we don't deal with brackets!

1. Operators appear after two operands (e.g. "2 5 +")
2. To evaluate an operator, we apply it to the previous two numbers!

2 5 + 11 *

Numbers	Token	Result
[]	2	0
[2]	5	0
[2, 5]	+	7
[7]	11	7
[7, 11]	*	77
[77]	null	**77**

Stack (Application - Infix to postfix)

Convert from infix to postfix

Token	Output

1. If you see an **operand** (i.e., **number**), then add it to the output!

Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
2	2

1. If you see an **operand** (i.e., **number**), then add it to the output!

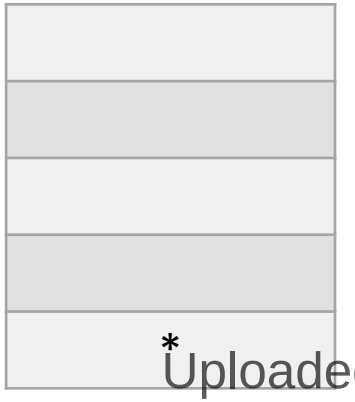
Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
*	2

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.



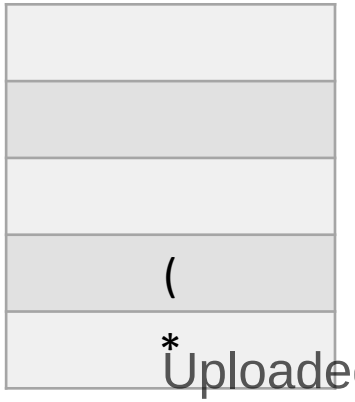
Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
(	2

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.



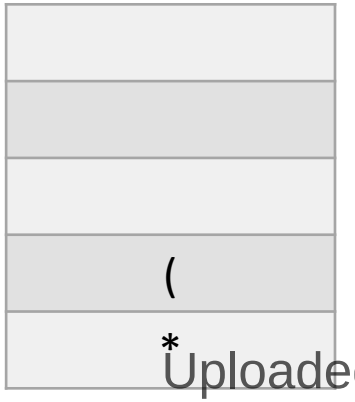
Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
5	2 5

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.



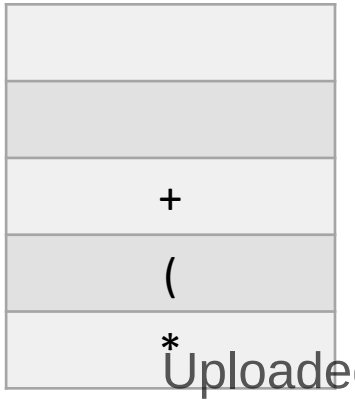
Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
+	2 5

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack



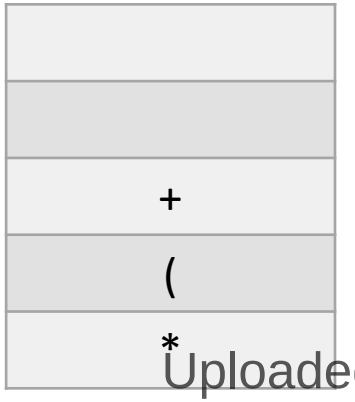
Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
4	2 5 4

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack



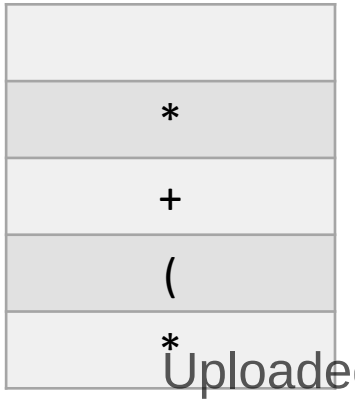
Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
*	2 5 4

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack
   iii. If it has higher priority than top of the stack, then push the operator into the stack (e.g., "*" > "+")
   iv. If it has lower priority, then pop the stack and add the popped element to the output and repeat step 3



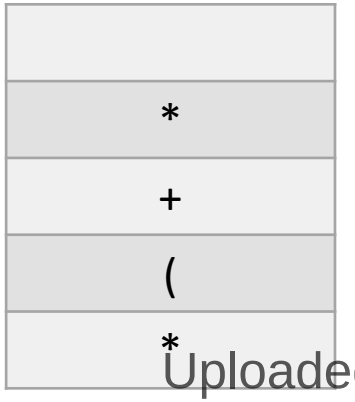
Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
2	2 5 4 2

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack
   iii. If it has higher priority than top of the stack, then push the operator into the stack (e.g., "*" > "+")
   iv. If it has lower priority, then pop the stack and add the popped element to the output and repeat step 3



Stack (Application - Infix to postfix)

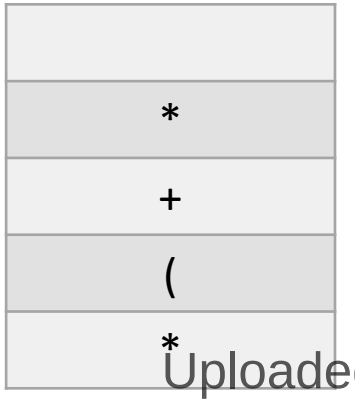
Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
)	2 5 4 2

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack
   iii. If it has higher priority than top of the stack, then push the operator into the stack (e.g., "*" > "+")
   iv. If it has lower priority, then pop the stack and add the popped element to the output and repeat step 3

4. If it's a **closing parenthesis**, then keep popping and adding to output until you reach an **opening parenthesis**.



Stack (Application - Infix to postfix)

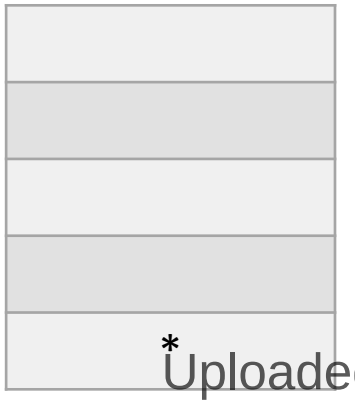
Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
)	2 5 4 2 * +

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack
   iii. If it has higher priority than top of the stack, then push the operator into the stack (e.g., "*" > "+")
   iv. If it has lower priority, then pop the stack and add the popped element to the output and repeat step 3

4. If it's a **closing parenthesis**, then keep popping and adding to output until you reach an **opening parenthesis**.



Stack (Application - Infix to postfix)

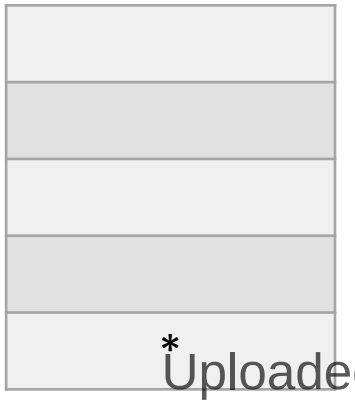
Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
/	2 5 4 2 * +

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack
   iii. If it has higher priority than top of the stack, then push the operator into the stack (e.g., "*" > "+")
   iv. If it has lower priority, then pop the stack and add the popped element to the output and repeat step 3

4. If it's a **closing parenthesis**, then keep popping and adding to output until you reach an **opening parenthesis**.



Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
/	2 5 4 2 * + *

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack
   iii. If it has higher priority than top of the stack, then push the operator into the stack (e.g., "*" > "+")
   iv. If it has lower priority, then pop the stack and add the popped element to the output and repeat step 3

4. If it's a **closing parenthesis**, then keep popping and adding to output until you reach an **opening parenthesis**.

Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
3	2 5 4 2 * + * 3

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack
   iii. If it has higher priority than top of the stack, then push the operator into the stack (e.g., "*" > "+")
   iv. If it has lower priority, then pop the stack and add the popped element to the output and repeat step 3

4. If it's a **closing parenthesis**, then keep popping and adding to output until you reach an **opening parenthesis**.

5. If there is input repeat step 1.

6. If we are done then pop everything and add it to the output!

Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
3	2 5 4 2 * + * 3 /

1. If you see an **operand** (i.e., **number**), then add it to the output!
2. If it's an **opening parenthesis** "**(**" then we push it into a stack!
3. If it's an **operator** (e.g., "+", "-", "*", "/"):
   i. If stack is empty, then push the operator into the stack.
   ii. If the top of the stack is an **opening parenthesis,** then push the operator into the stack
   iii. If it has higher priority than top of the stack, then push the operator into the stack (e.g., "*" > "+")
   iv. If it has lower priority, then pop the stack and add the popped element to the output and repeat step 3

4. If it's a **closing parenthesis**, then keep popping and adding to output until you reach an **opening parenthesis**.

5. If there is input repeat step 1.

6. If we are done then pop everything and add it to the output!

Stack (Application - Infix to postfix)

Convert from infix to postfix

2 * (5 + 4 * 2) / 3

Token	Output
3	2 5 4 2 * + * 3 /

2 * (5 + 4 * 2) / 3 => 2 5 4 2 * + * 3 /

Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token

1. If you see an **operand** (i.e., **number**), then push it to stack!

Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
2

1. If you see an **operand** (i.e., **number**), then push it to stack!

Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
5

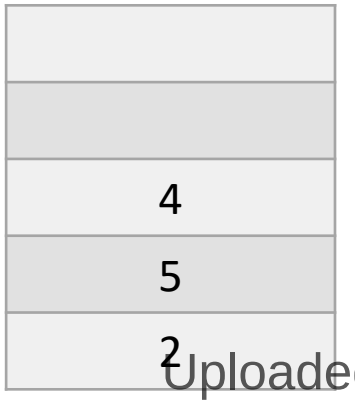
1. If you see an **operand** (i.e., **number**), then push it to stack!

Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
4

1. If you see an **operand** (i.e., **number**), then push it to stack!



Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
2

1. If you see an **operand** (i.e., **number**), then push it to stack!

Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
*

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
*

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop();
int x = stack.pop();
int result = x * y;
stack.push(result);
```

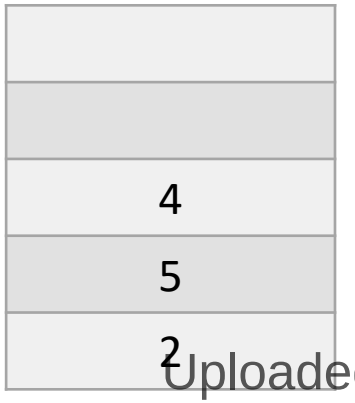
Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
*

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop(); // 2
int x = stack.pop();
int result = x * y;
stack.push(result);
```



Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
*

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop(); // 2
int x = stack.pop(); // 4
int result = x * y;
stack.push(result);
```

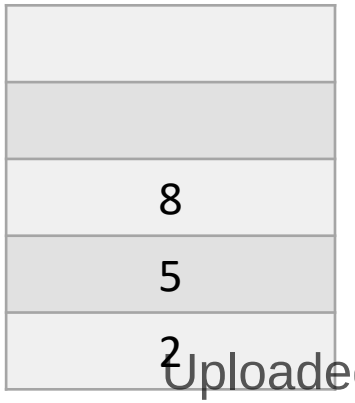
Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
*

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop(); // 2
int x = stack.pop(); // 4
int result = x * y;  // 8
stack.push(result);
```



Stack (Application - Infix to postfix)

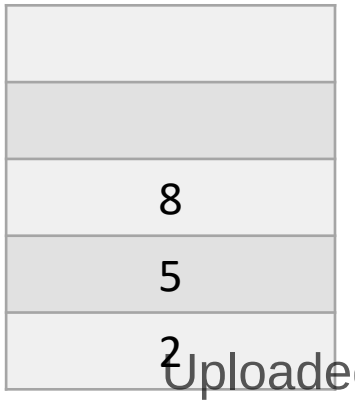
Evaluate postfix

2 5 4 2 * + * 3 /

Token
+

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop(); // 8
int x = stack.pop(); // 5
int result = x + y;  // 13
stack.push(result);
```



Stack (Application - Infix to postfix)

Evaluate postfix

2 5 4 2 * + * 3 /

Token
+

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop(); // 8
int x = stack.pop(); // 5
int result = x + y;  // 13
stack.push(result);
```

Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
*

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop(); // 13
int x = stack.pop(); // 2
int result = x * y;  // 26
stack.push(result);
```

Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
3

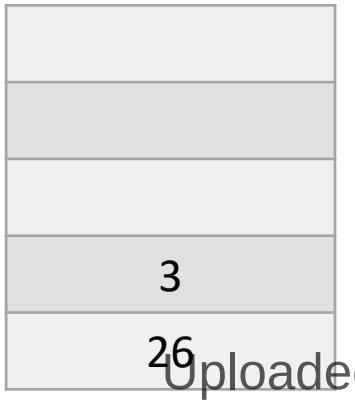
1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
3

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!



Stack (Application - Infix to postfix)

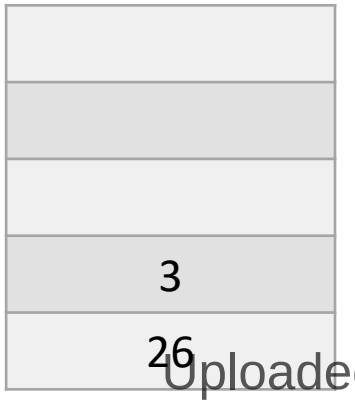
Evaluate postfix

2 5 4 2 * + * 3 /

Token
/

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop(); // 3
int x = stack.pop(); // 26
int result = x / y;  // 8.6
stack.push(result);
```



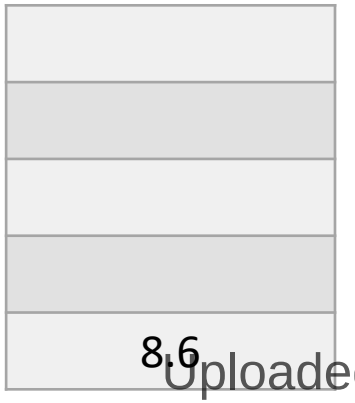
Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
/

1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop(); // 3
int x = stack.pop(); // 26
int result = x / y;  // 8.6
stack.push(result);
```



Stack (Application - Infix to postfix)

Evaluate postfix

2 5 4 2 * + * 3 /

Token
/

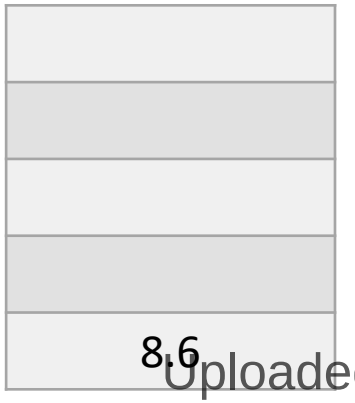
1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

```
int y = stack.pop(); // 3
int x = stack.pop(); // 26
int result = x / y;  // 8.6
stack.push(result);
```

3. Repeat until input is finished!

4. When there is no more input the result will be at the top of the stack!

2 * (5 + 4 * 2) / 3 => 2 5 4 2 * + * 3 /



Stack (Application - Infix to postfix)

Evaluate postfix 2 5 4 2 * + * 3 /

Token
/

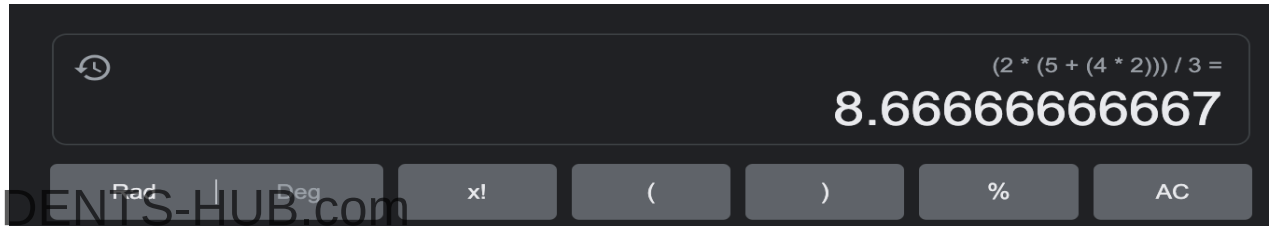
1. If you see an **operand** (i.e., **number**), then push it to stack!
2. If you see an **operator,** then pop two numbers from the stack and evaluate the **operator**!
   Then you push the result back into the stack!

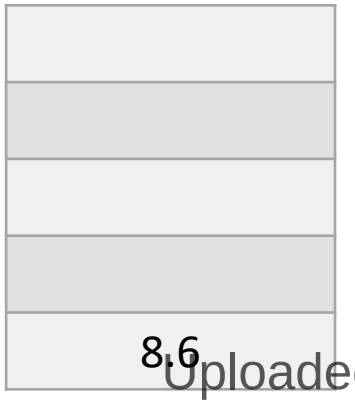
```
int y = stack.pop(); // 3
int x = stack.pop(); // 26
int result = x / y;  // 8.6
stack.push(result);
```

3. Repeat until input is finished!

4. When there is no more input the result will be at the top of the stack!

2 * (5 + 4 * 2) / 3 => 2 5 4 2 * + * 3 /





Stack (Exercises)

1. **Evaluate “2 3 * 2 1 - / 5 3 * +”**
   Result should be **21**

2. **Convert “(((3 + 5) * (7 - 9)) / (11 + 13))” to postfix.**

3. **Evaluate the following:**
   1. **“6 2 3 + - 3 8 2 / + * 2 + 3 +”**
   2. **“4 5 + 7 2 - *”**
   3. **“7 5 2 * + 4 3 4 + * 4 * -”**

Stack (Exercises)

1. **Write java code to evaluate postfix input.**

2. **Compare between Array and LinkedList implementation of Stack.**

   You should compare between methods (push, pop and peek)
   You should compare implementations based on Memory and time!

Stack (Application - Balanced Brackets)

- 1 + (4 * 5)
- 7 + (2 + 3
- 3 * 2 / 5)

How to tell if brackets are balanced or not?

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token

1. If you anything other than a bracket, then ignore it

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
2

1. If you anything other than a bracket, then ignore it

Ignore it!

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
*

1. If you anything other than a bracket, then ignore it

Ignore it!

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
(

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!

stack.push(‘(‘);

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
5

Ignore it!

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
+

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!

Ignore it!

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
7

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!

Ignore it!

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
)

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!
3. If you see an **opening bracket (e.g., ‘)‘, ’}‘, ‘]‘)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
)

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!
3. If you see an **opening bracket (e.g., ‘)‘, ’}‘, ‘]‘)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).

‘)’ matches with ‘(‘
so, we pop the top.

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
)

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!
3. If you see an **opening bracket (e.g., ‘)‘, ’}‘, ‘]‘)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
)

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!
3. If you see an **opening bracket (e.g., ‘)‘, ’}‘, ‘]‘)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).

4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).

Stack (Application - Infix to postfix)

Check Brackets

2 * (5 +7))

Token
)

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., '(', '{', '[')**, then push it into the stack!
3. If you see an **opening bracket (e.g., ')', '}', ']')**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).

4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).

5. Otherwise return true, (brackets are **balanced**).

Stack (Application - Infix to postfix)

Check Brackets

2 * ((5 +7)

Token
2

Ignore it!

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!
3. If you see an **opening bracket (e.g., ‘)‘, ’}‘, ‘]‘)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).
4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).
5. Otherwise return true, (brackets are **balanced**).

Stack (Application - Infix to postfix)

Check Brackets

2 * ((5 +7)

Token
*

Ignore it!

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!
3. If you see an **opening bracket (e.g., ‘)‘, ’}‘, ‘]‘)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).
4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).
5. Otherwise return true, (brackets are **balanced**).

Stack (Application - Infix to postfix)

Check Brackets

2 * ((5 +7)

Token
(

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g.,** ‘(‘, ’{‘, ‘[‘**)**, then push it into the stack!
3. If you see an **opening bracket (e.g.,** ‘)‘, ’}‘, ‘]‘**)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).

4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).

5. Otherwise return true, (brackets are **balanced**).

Stack (Application - Infix to postfix)

Check Brackets

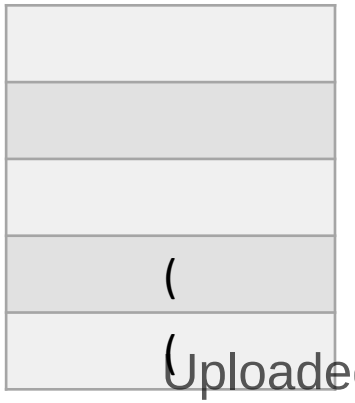
2 * ((5 +7)

Token
(

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g.,** ‘(‘, ’{‘, ‘[‘**)**, then push it into the stack!
3. If you see an **opening bracket (e.g.,** ‘)‘, ’}‘, ‘]‘**)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).

4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).

5. Otherwise return true, (brackets are **balanced**).



Stack (Application - Infix to postfix)

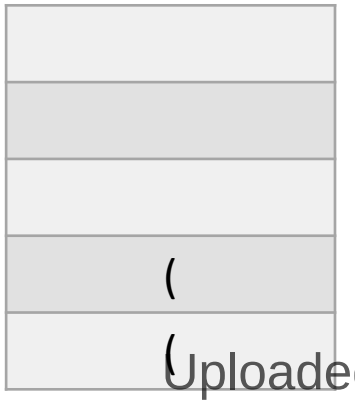
Check Brackets

2 * ((5 +7)

Token
5

Ignore it!

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g.,** ‘(‘, ’{‘, ‘[‘**)**, then push it into the stack!
3. If you see an **opening bracket (e.g.,** ‘)‘, ’}‘, ‘]‘**)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).
4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).
5. Otherwise return true, (brackets are **balanced**).



Stack (Application - Infix to postfix)

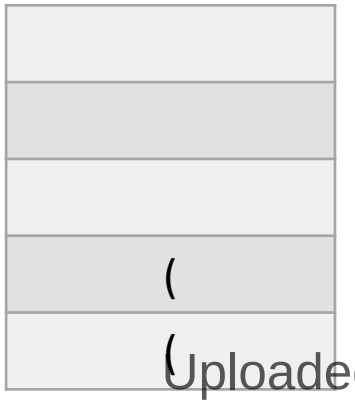
Check Brackets

2 * ((5 +7)

Token
+

Ignore it!

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g.,** ‘(‘, ’{‘, ‘[‘**)**, then push it into the stack!
3. If you see an **opening bracket (e.g.,** ‘)‘, ’}‘, ‘]‘**)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).
4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).
5. Otherwise return true, (brackets are **balanced**).



Stack (Application - Infix to postfix)

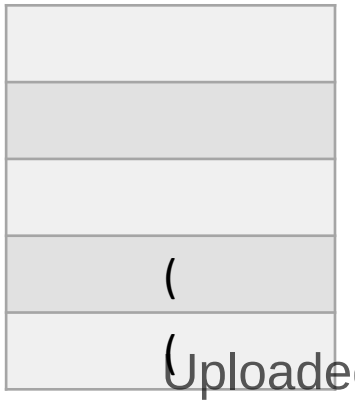
Check Brackets

2 * ((5 +7)

Token
7

Ignore it!

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g.,** ‘(‘, ’{‘, ‘[‘**)**, then push it into the stack!
3. If you see an **opening bracket (e.g.,** ‘)‘, ’}‘, ‘]‘**)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).
4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).
5. Otherwise return true, (brackets are **balanced**).



Stack (Application - Infix to postfix)

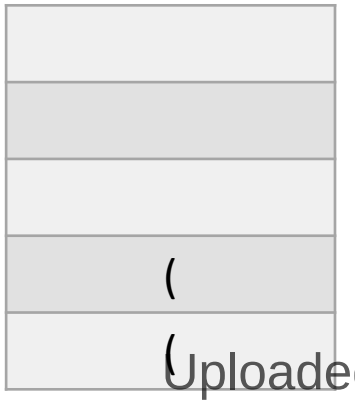
Check Brackets

2 * ((5 +7)

Token
)

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., ‘(‘, ’{‘, ‘[‘)**, then push it into the stack!
3. If you see an **opening bracket (e.g., ‘)‘, ’}‘, ‘]‘)**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).
4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).
5. Otherwise return true, (brackets are **balanced**).

‘)’ matches with ‘(‘
so, we pop the top.



Stack (Application - Infix to postfix)

Check Brackets

2 * ((5 +7)

Token

1. If you anything other than a bracket, then ignore it.
2. If you see an **opening bracket (e.g., '(', '{', '[')**, then push it into the stack!
3. If you see an **opening bracket (e.g., ')', '}', ']')**:
    1. If the closing **matches** with the top of the Stack, then pop the top.
       If it does **not match**, then return false (brackets are **unbalanced**).
    2. If the the stack is empty, then return false (brackets are **unbalanced**).
4. When finished if the stack is not empty, then return false (brackets are **unbalanced**).
5. Otherwise return true, (brackets are **balanced**).

Since stack is not empty then we return false!