[SWE2015-41] Introduction to Data Structures (자료구조개론)

Stacks

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

(Recap) Arrays

- **An array** is a collection of elements of **the same data type** in **a contiguous block of memory**

- The `i`-th element can be accessed by `arr[i]`

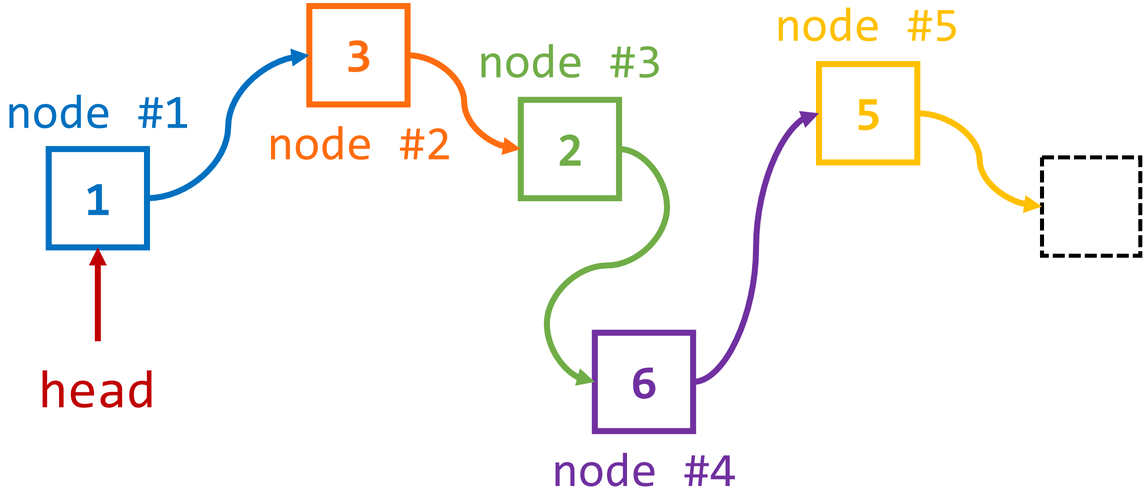
- Time complexity for the access = $O(1)$
  - Address computation requires $O(1)$

```
numbers = &numbers[0] = 0x16aedf320
&numbers[7] = &numbers[0] + 7
            = 0x16aedf33c
```

Index	Address	Value
0	0x16aedf320	1
1	0x16aedf324	5
2	0x16aedf328	9
3	0x16aedf32c	-3
4	0x16aedf330	8
5	0x16aedf334	7
6	0x16aedf338	6
7	**0x16aedf33c**	**10**
8	0x16aedf340	-5
9	0x16aedf344	0

(Recap) Linked Lists

- **A linked list** is a collection of **sequentially-connected** elements
  - The elements are not required to be stored in contiguous memory

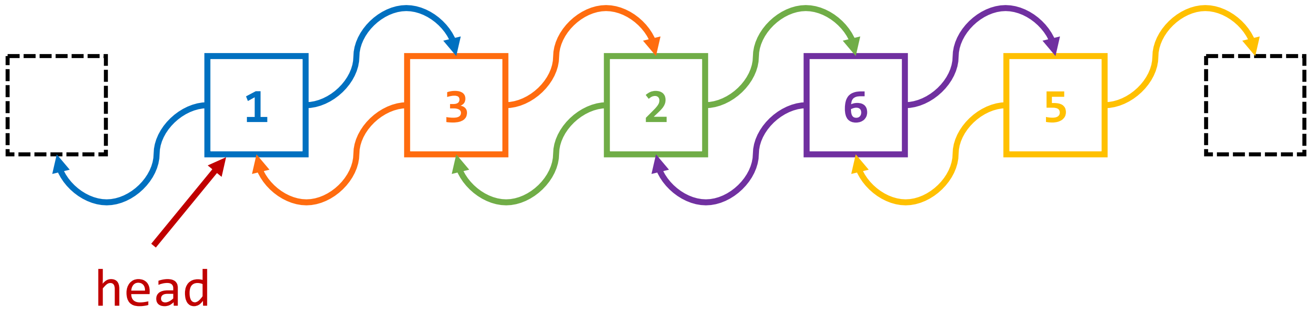


```
typedef struct _Node {
    int value;
    struct _Node *next;
} Node;
typedef struct _LinkedList {
    Node *head;
} LinkedList;
```

(Recap) Doubly Linked Lists

- Doubly Linked Lists are bidirectional
  - Every node has `prev` and `next` pointers for previous and next nodes
  - Bidirectional pointers (`prev`, `next`) enable to move forward and backward

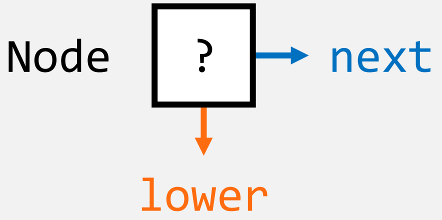
```
typedef struct _Node {
    int value;
    struct _Node *prev, *next;
} Node;
```



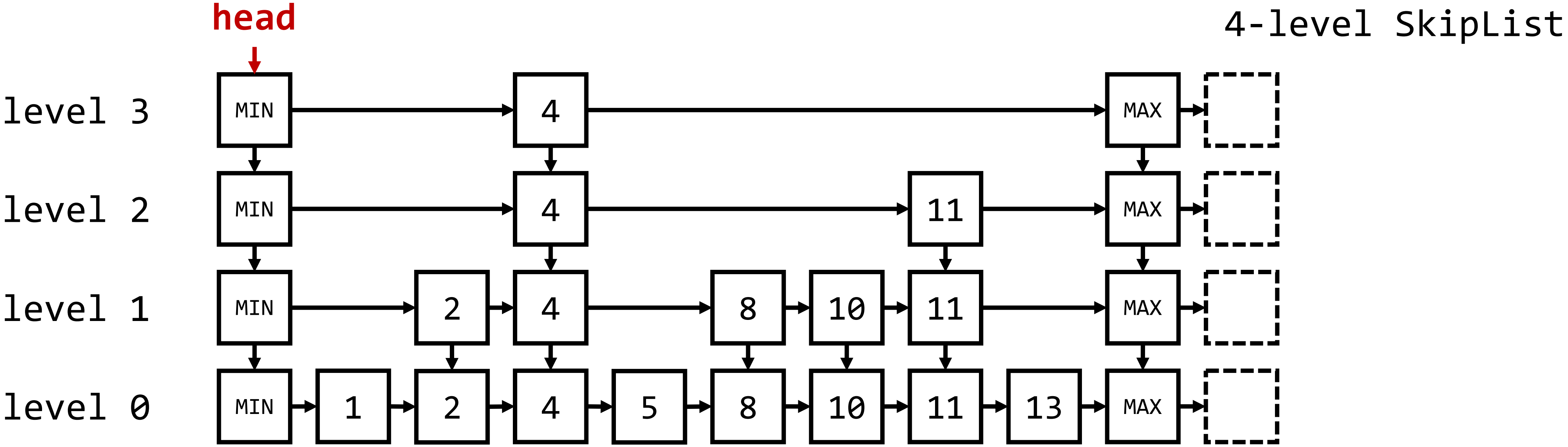
(Recap) Skip Lists

- A **Skip List** is an advanced variant of **ordered/sorted** linked lists
  - This can be implemented by a two-dimensional linked list

```
typedef struct _Node {
    int value;
    struct _Node *next, *lower;
} Node;
```



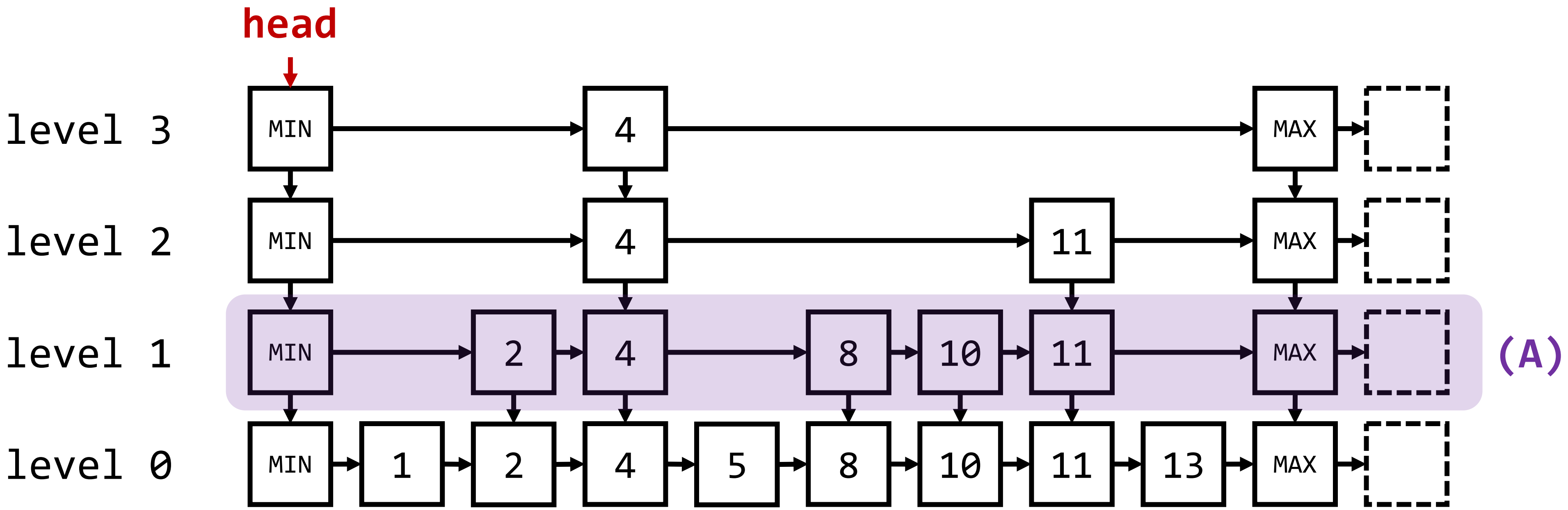
```
typedef struct _SkipList {
    Node *head;
    int num_levels;
} SkipList;
```



(Recap) Skip Lists

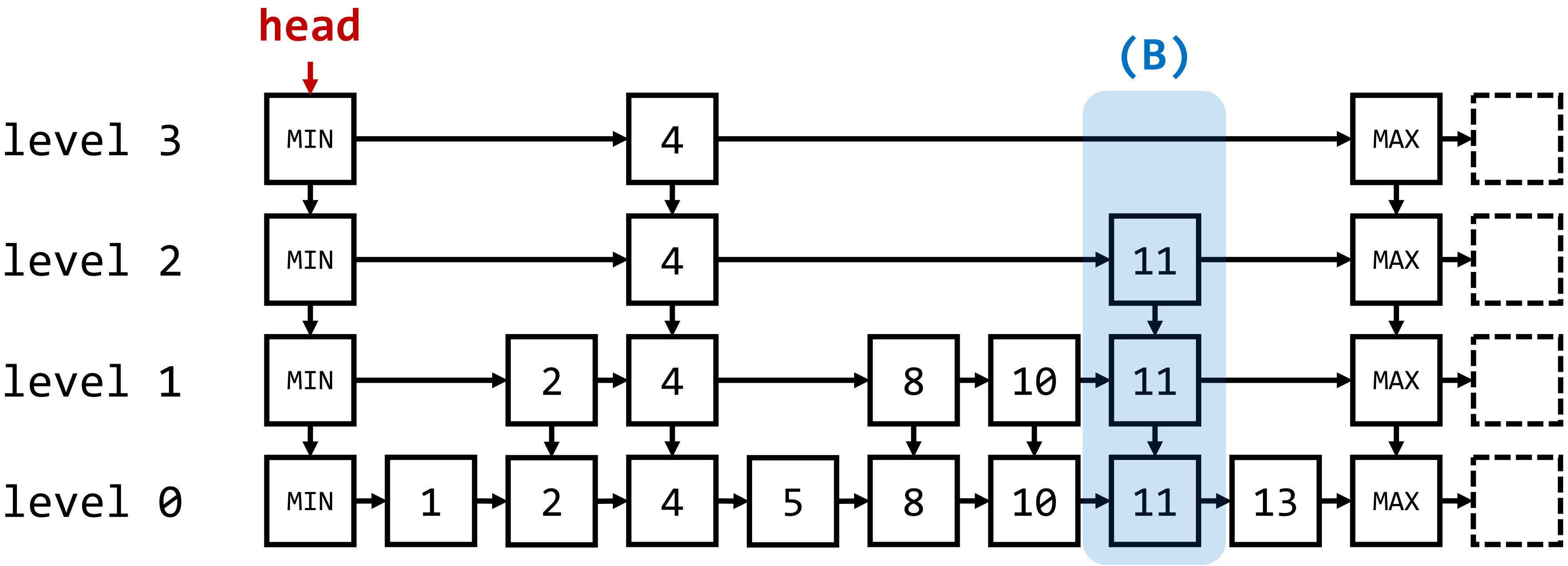
- A **Skip List** is an advanced variant of **ordered/sorted** linked lists
  - This can be implemented by a two-dimensional linked list
  - Two properties:

    (A) Elements at each level (i.e., row) are sorted



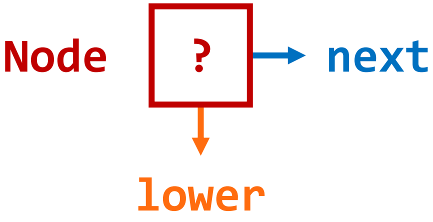
(Recap) Skip Lists

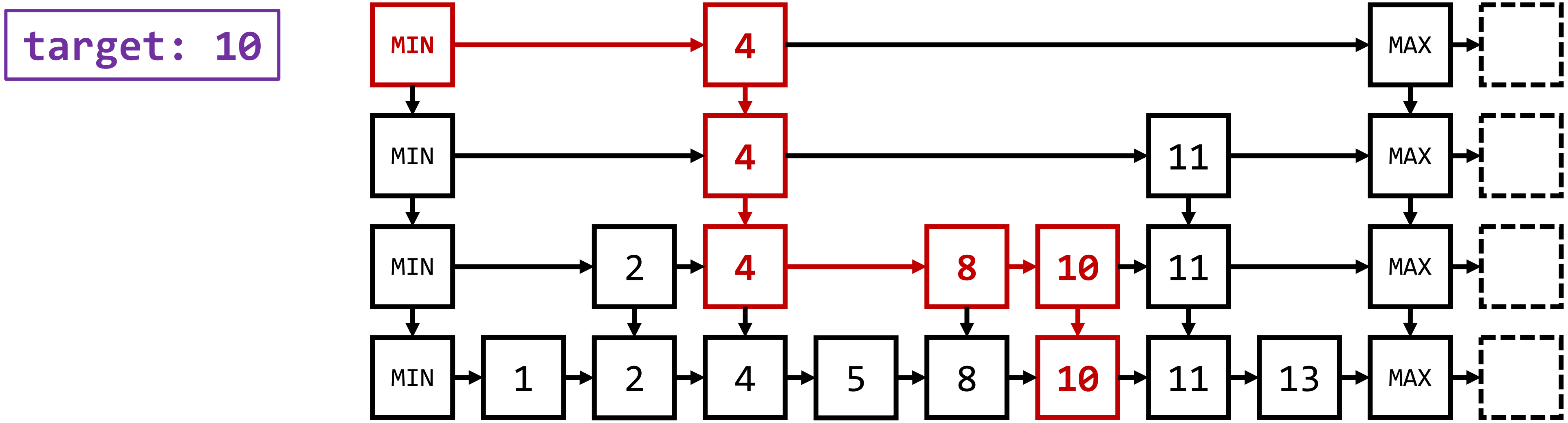
- A **Skip List** is an advanced variant of **ordered/sorted** linked lists
  - This can be implemented by a two-dimensional linked list
  - Two properties:
    - **(A)** Elements at each level (i.e., row) are sorted
    - **(B)** Elements at each column exist consecutively



(Recap) Skip Lists

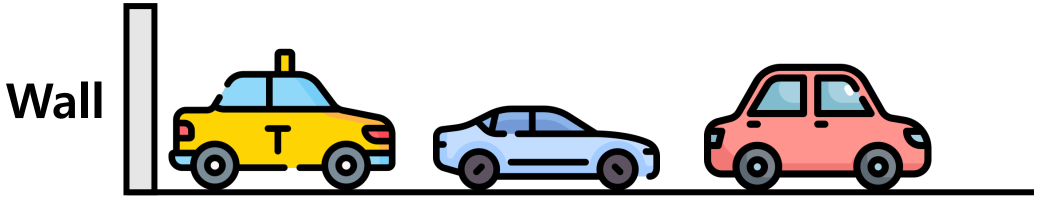
- The search procedure in **Skip Lists**
  1. Starting from **head**,
  2. Compare the **target** value with the **next** node
     1. If **current < next <= target**, then move to the **next** node
     2. If **current <= target < next**, then move to the **lower** node





What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed



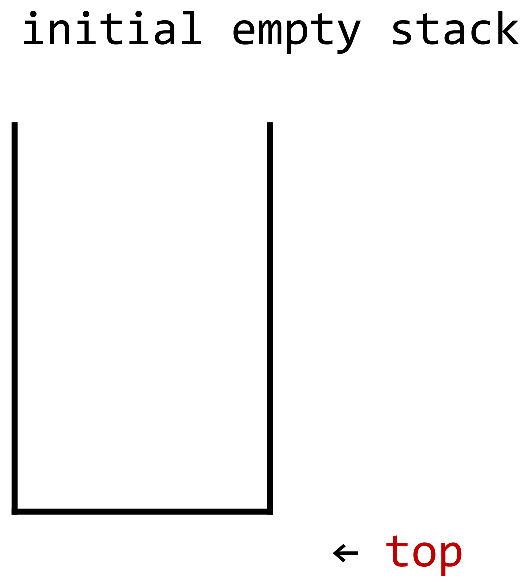
← This blue book was stacked last

← This yellow book was stacked first

What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack

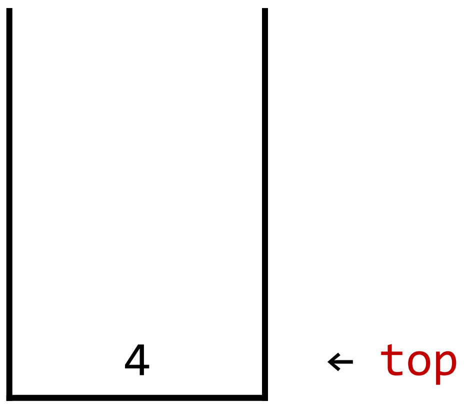


What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack

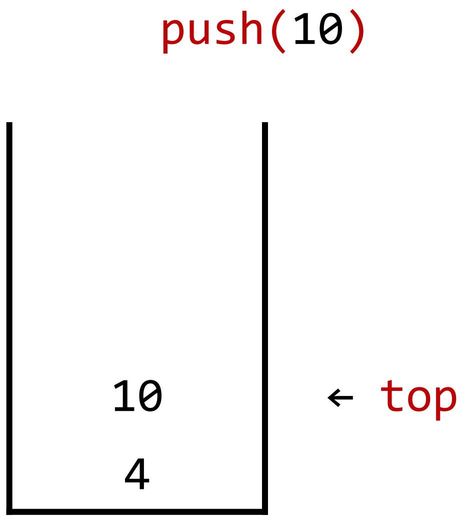
push(4)



What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
    - Insertion, deletion, and information access are only possible at the top on the stack
    - **LIFO:** the last added element will be the first element to be removed

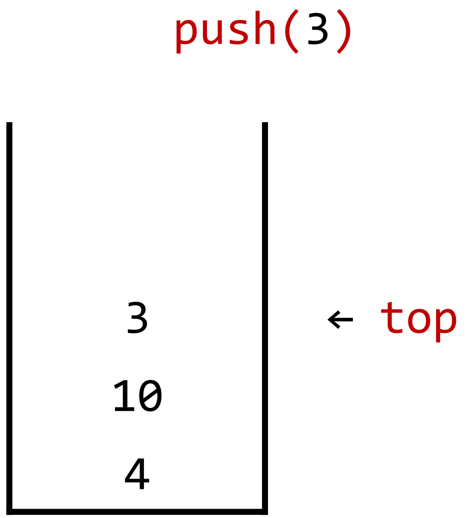
- Main components
    - `top` - represent the top of the stack
        - This can be represented by index or pointer
    - `push()` - insert an element to the top of the stack
    - `pop()` - delete the topmost element from the stack
    - `peek()` - return the value of topmost element of the stack



What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

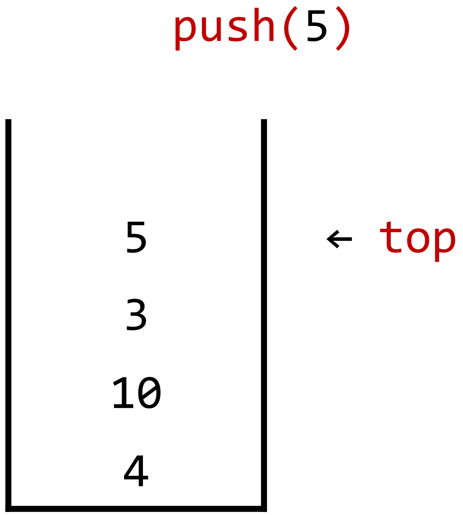
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



push(5)

5 ← top

3

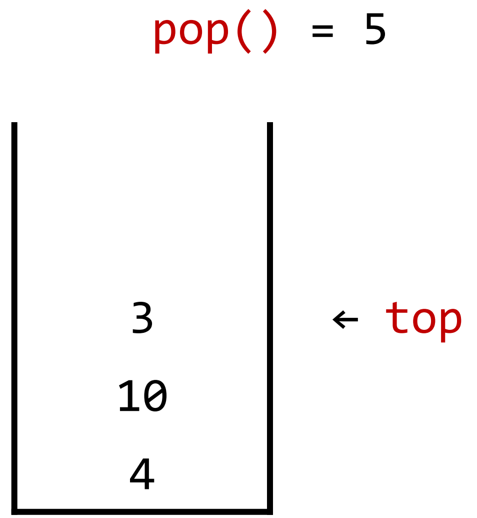
10

4

What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



`pop() = 5`

3 ← `top`

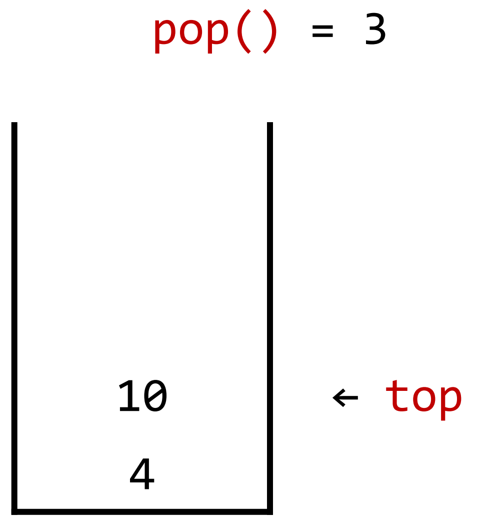
10

4

What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

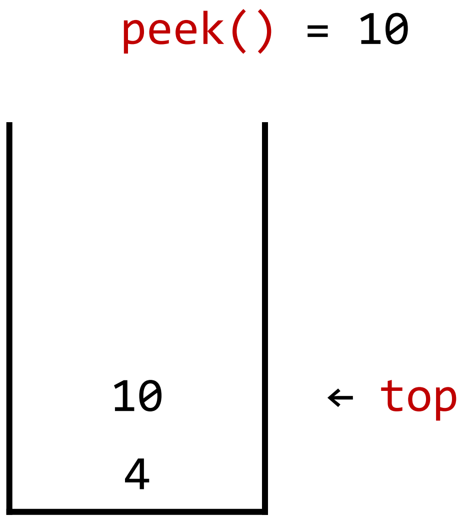
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

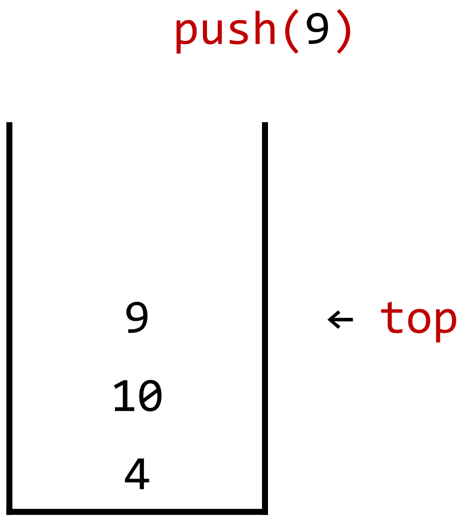
- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack

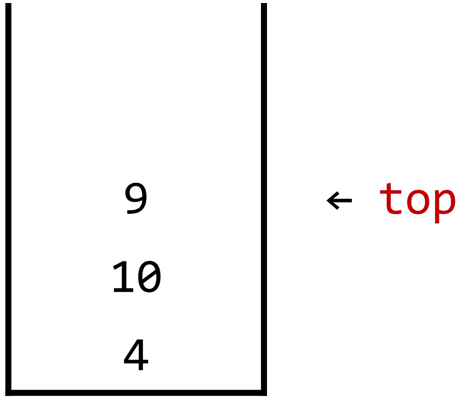


push(9)

What is Stack?

- Stack is a collection of elements that are inserted and removed according to **the last-in first-out (LIFO) principle**
  - Insertion, deletion, and information access are only possible at the top on the stack
  - **LIFO:** the last added element will be the first element to be removed

- Main components
  - `top` - represent the top of the stack
    - This can be represented by index or pointer
  - `push()` - insert an element to the top of the stack
  - `pop()` - delete the topmost element from the stack
  - `peek()` - return the value of topmost element of the stack



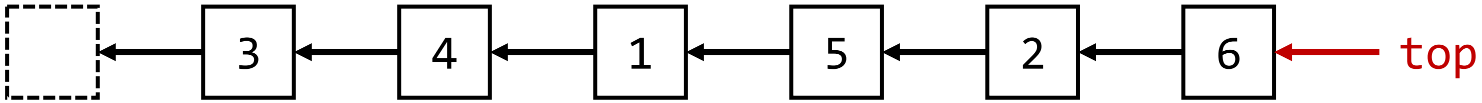
- **Note.** We cannot access items other than at the top (by definition)

Stacks - Implementation

- You have **two options** for stack implementation
  1. **Use Array**

	0	1	2	3	4	5	6	7	...	MAX-1
top = 5	3	4	1	5	2	6				

  2. **Use Linked list**



Stacks - Implementation

- You have **two options** for stack implementation
  1. **Use Array**
     - (+) Implementation is easy
     - (-) Arrays must be declared to have some fixed size
  2. **Use Linked list**
     - (+) Linked Lists can dynamically increase and decrease in size
     - (-) Implementation is (slightly) more difficult than the array-based implementation

Stacks - Array-based Implementation

- How to implement a stack using **the array structure**?

	0	1	2	3	4	5	6	7	...	MAX-1
top = 5	3	4	1	5	2	6				

```
#define MAX_SIZE 100
typedef struct _Stack {
    int top;              // index for the top element
    int items[MAX_SIZE]; // array for stack elements
} Stack;

Stack createStack();
void removeStack(Stack *stack); // nothing to do here

bool isEmpty(Stack *stack);
bool isFull(Stack *stack);
void push(Stack *stack, int item);
int pop(Stack *stack);
int peek(Stack *stack);
```

Stacks - Array-based Implementation

- How to implement a stack using **the array structure**?
  - `top` is the index for the top element
  - What is the empty state?

`top = -1`

0	1	2	3	4	5	6	7	...	MAX-1

-
  - What is the full state?

`top = MAX-1`

0	1	2	3	4	5	6	7	...	MAX-1
1	3	2	6	7	4	5	0	...	9

Stacks - Array-based Implementation

- How to implement a stack using **the array structure**?

```
Stack createStack() {
    // Declare a new stack
    // Set the initial value for the top index
    // Return the new stack
}

bool isEmpty(Stack *stack) {
    // Check whether stack is empty or not
}

bool isFull(Stack *stack) {
    // Check whether stack is full or not
}
```

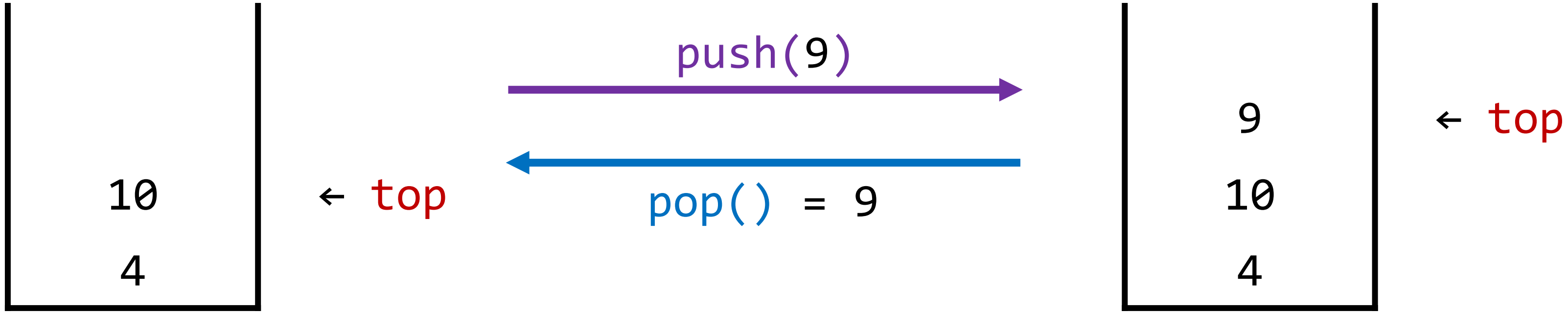
Stacks - Array-based Implementation

- How to implement a stack using **the array structure**?

```
Stack createStack() {
    Stack newStack; // Declare a new stack
    newStack.top = -1; // Set the initial value for the top index
    return newStack; // Return the new stack
}

bool isEmpty(Stack *stack) {
    return stack->top == -1; // Check whether stack is empty or not
}

bool isFull(Stack *stack) {
    return stack->top == MAX_SIZE-1; // Check whether stack is full or not
}
```

Stacks - Array-based Implementation

- How to implement a stack using **the array structure**?
  - push() increases the top index, and then puts an item
  - pop() reads the top item, and then decreases the top index
  - peek() simply reads and returns the top element



Stacks - Array-based Implementation

- How to implement a stack using **the array structure**?

```c
void push(Stack *stack, int item) {
    // Increase top index
    // Put item into stack
}

int pop(Stack *stack) {
    // Read top element
    // Decrease top index
    // Return previous top element
}

int peek(Stack *stack) {
    // Return top element
}
```

Stacks - Array-based Implementation

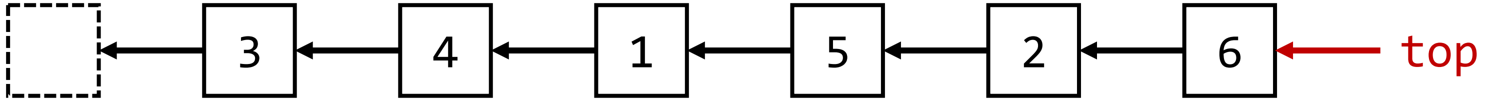
- How to implement a stack using **the array structure**?

```
void push(Stack *stack, int item) {
    stack->top ++; // Increase top index
    stack->items[stack->top] = item; // Put item into stack
}

int pop(Stack *stack) {
    int item = stack[stack->top]; // Read top element
    stack->top --; // Decrease top index
    return item; // Return previous top element
}

int peek(Stack *stack) {
    return stack[stack->top]; // Return top element
}
```

- **Corner cases:** You must check a structure is empty or full when insert or delete an element from a structure

Stacks - List-based Implementation

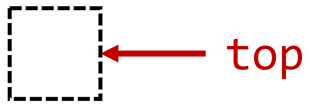
- How to implement a stack using **the linked list structure**?



```
typedef struct _Node { int item; struct _Node *next; } Node;
typedef struct _Stack {
    Node *top; // pointer for the top element
} Stack;

Stack createStack();
void removeStack(Stack *stack); // must remove dynamically allocated variables

bool isEmpty(Stack *stack);
bool isFull(Stack *stack);
void push(Stack *stack, int item);
int pop(Stack *stack);
int peek(Stack *stack);
```

Stacks - List-based Implementation

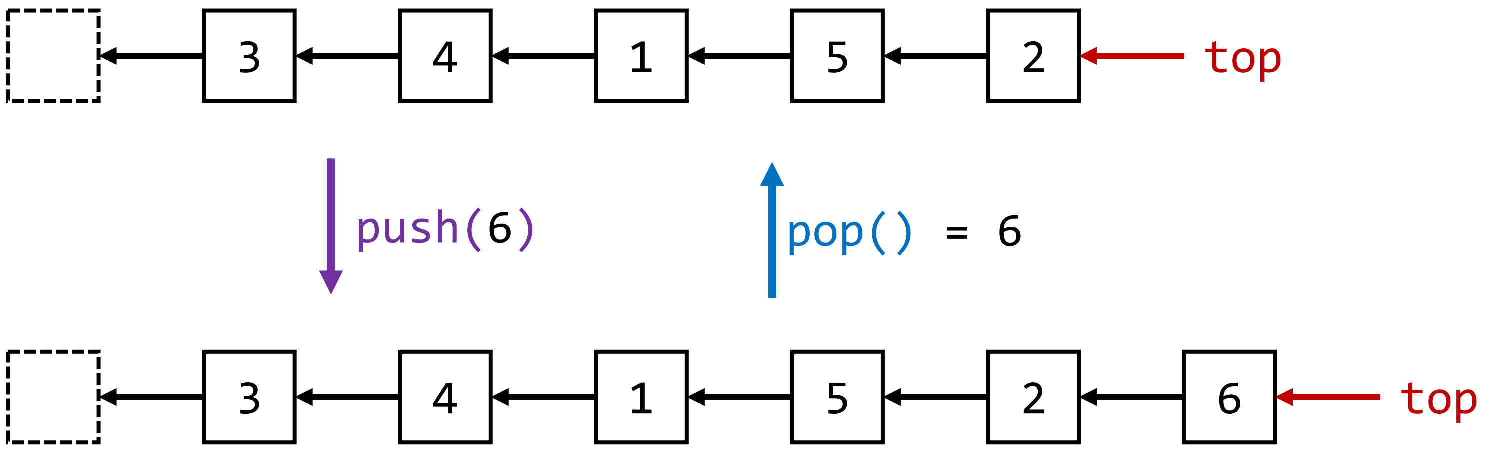
- How to implement a stack using **the linked list structure**?
  - `top` is the pointer for the top element
  - What is the empty state?



  - What is the full state?
    - There is no full state since the size is dynamically increased/decreased

Stacks - List-based Implementation

- How to implement a stack using **the linked list structure**?
  - push() creates a new node, and then link the node with the top node
  - pop() reads the top element, and then remove the top node
  - peek() simply reads and returns the top element



Stacks - Implementation

- What are different and same between **the array-based and list-based implementations**?

```
typedef struct _Stack {
    int top;
    int items[MAX_SIZE];
} Stack;

Stack createStack();
void removeStack(Stack *stack);

bool isEmpty(Stack *stack);
bool isFull(Stack *stack);
void push(Stack *stack, int item);
int pop(Stack *stack);
int peek(Stack *stack);
```

```
typedef struct _Node { ... } Node;
typedef struct _Stack {
    Node *top;
} Stack;

Stack createStack();
void removeStack(Stack *stack);

bool isEmpty(Stack *stack);
bool isFull(Stack *stack);
void push(Stack *stack, int item);
int pop(Stack *stack);
int peek(Stack *stack);
```

Stacks - Implementation

- What are different and same between **the array-based and list-based implementations**?

Different Implementations

```
typedef struct _Stack {
    int top;
    int items[MAX_SIZE];
} Stack;

Stack createStack();
void removeStack(Stack *stack);

bool isEmpty(Stack *stack);
bool isFull(Stack *stack);
void push(Stack *stack, int item);
int pop(Stack *stack);
int peek(Stack *stack);
```

```
typedef struct _Node { ... } Node;
typedef struct _Stack {
    Node *top;
} Stack;

Stack createStack();
void removeStack(Stack *stack);

bool isEmpty(Stack *stack);
bool isFull(Stack *stack);
void push(Stack *stack, int item);
int pop(Stack *stack);
int peek(Stack *stack);
```

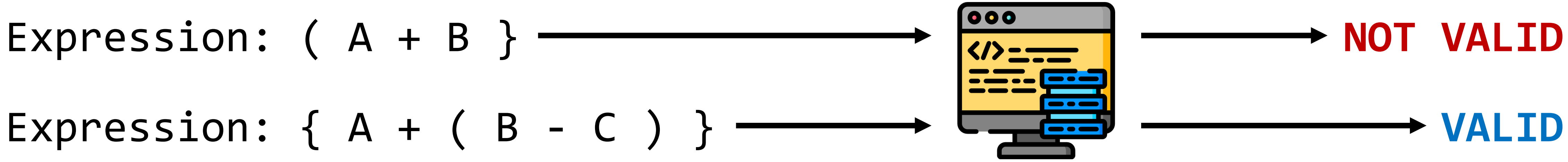
Same User Interface

Stacks - Implementation

- What are different and same between **the array-based and list-based implementations**?
  - **Different implementation** provides different performance for a data structure
  - **Same user interface** allows users not to worry about how it is implemented
  - For better coding,
    you must design **efficient implementation** and **reusable user interface**

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - Check the **validity of parentheses** in any algebraic expression



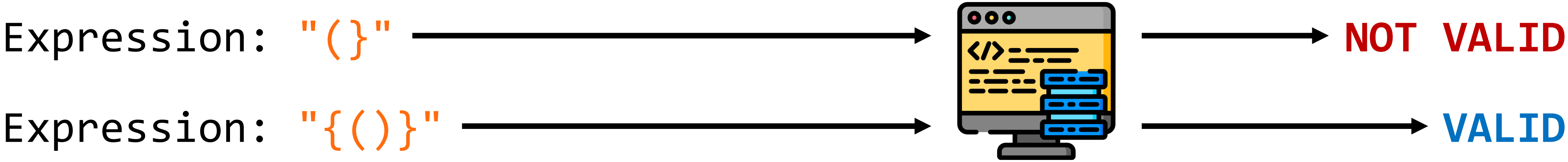
Expression: (A + B } → **NOT VALID**

Expression: { A + (B - C) } → **VALID**

(Q) How to implement the checker?

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - Check the **validity of parentheses** in any algebraic expression



```
Expression: "(}"
Expression: "{()}"
```

NOT VALID

VALID

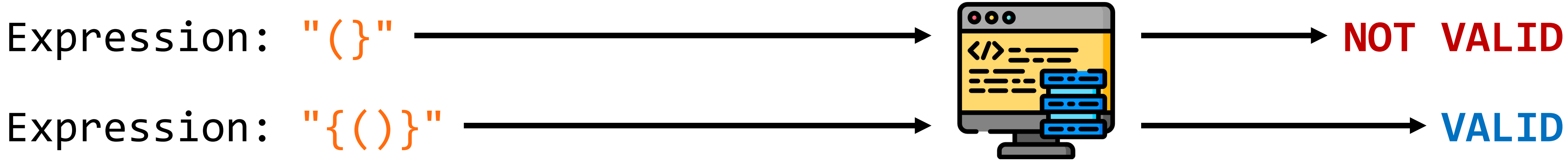
(Q) How to implement the checker?

(Step 1) Simplify the problem: remove everything except for the parentheses

- `( A + B } → "(}"`
- `{ A + ( B - C ) } → "{()}"`

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - Check the **validity of parentheses** in any algebraic expression



Expression: "(}" → NOT VALID

Expression: "{()}" → VALID

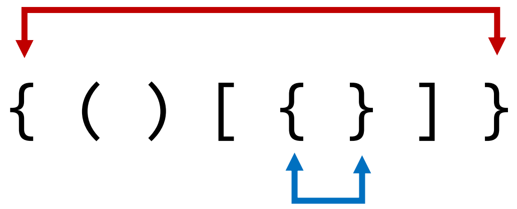
(Q) How to implement the checker?

(Step 1) Simplify the problem: remove everything except for the parentheses

- (A + B } → "(}"
- { A + (B - C) } → "{()}"

(Step 2) Think the property of parentheses

- The **first-open** parenthesis matches the **last-closed** parenthesis
- The **last-open** parenthesis matches the **first-closed** parenthesis



{ ( ) [{ }] }

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work

{ ( ) [{ }] }

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work

{ ( ) [{ }] }

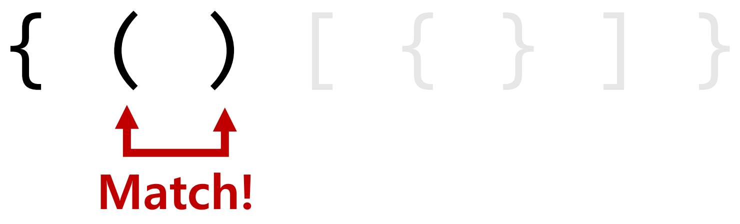
Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work

{ ( ) [{ }] }

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work

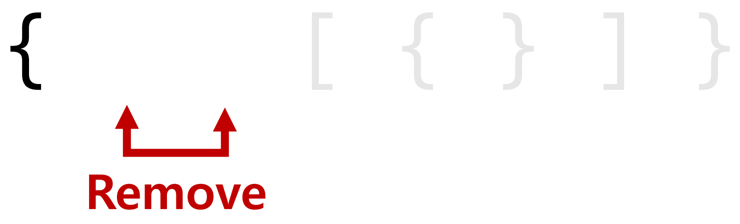


{ ( ) [{ }] }

Match!

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work



{ [{ }] }

Remove

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work

{ [{ }] }

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**

(Q) How to implement the checker?

(Step 1) Simplify the problem: remove everything except for the parentheses

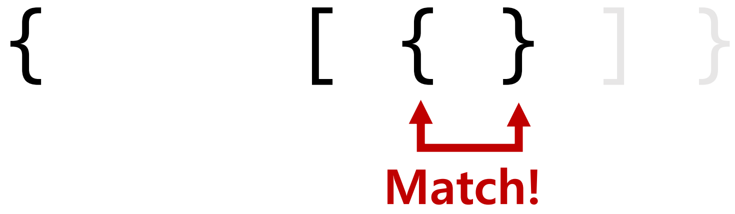
(Step 2) Think the property of parentheses

(Step 3) Imagine how does the checker work

{   [{ }] }

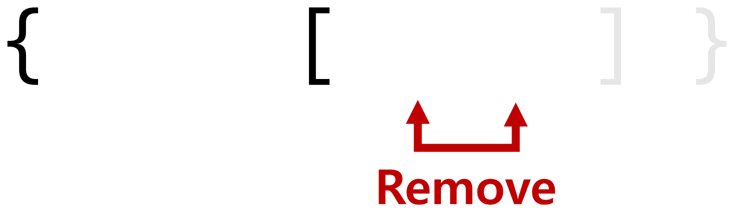
Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work



Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work

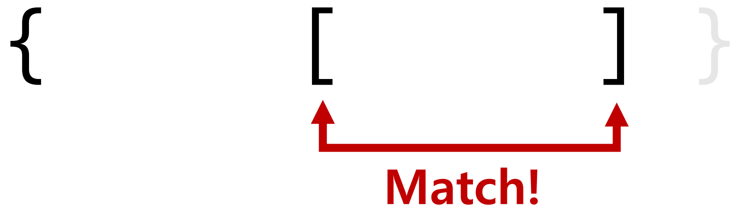


{ [ ] }

Remove

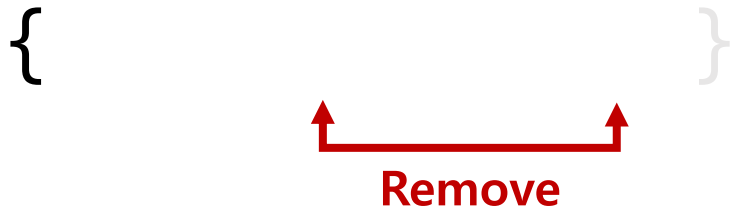
Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work



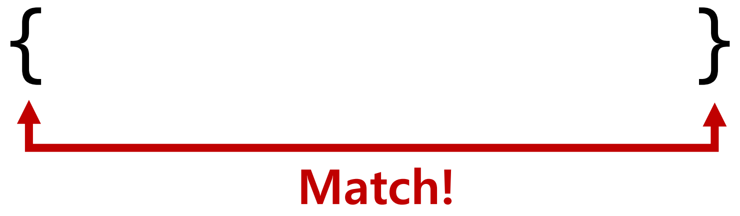
Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work



Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work



Stacks - Problem Solving Practice

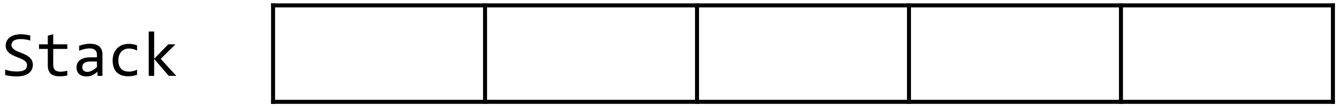
- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work

Remove

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work

{ ( ) [{ }] }



Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work

{ ( ) [{ }] }

Stack	{				

push({)

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work

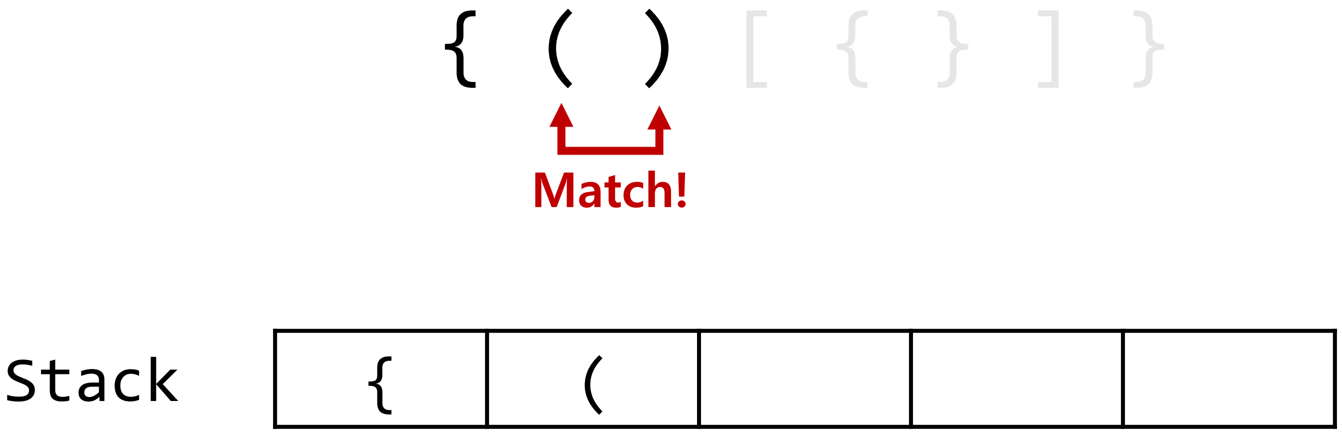
{ ( ) [{ }] }

Stack	{	(			

push(( )

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work



{ ( ) [{ }] }

Match!

Stack

{	(			

Stacks - Problem Solving Practice

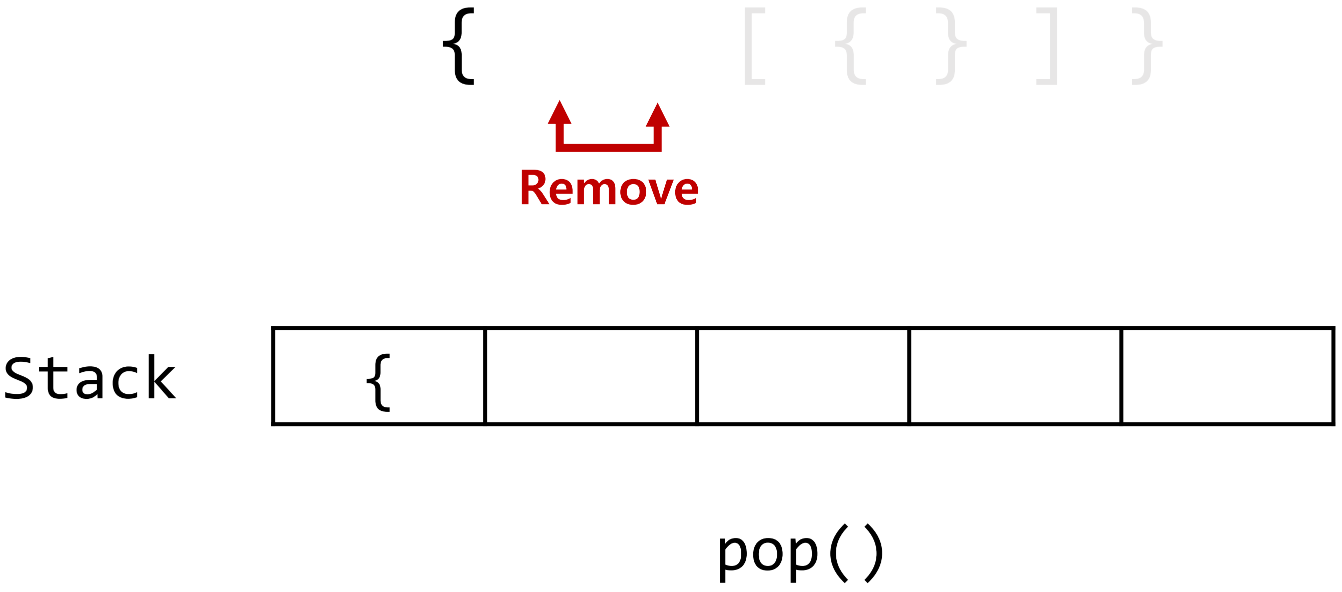
- Problem: **Parentheses Checker**

  (Q) How to implement the checker?

  (Step 1) Simplify the problem: remove everything except for the parentheses

  (Step 2) Think the property of parentheses

  (Step 3) Imagine how does the checker work



Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
    - **(Q)** How to implement the checker?
    - **(Step 1)** Simplify the problem: remove everything except for the parentheses
    - **(Step 2)** Think the property of parentheses
    - **(Step 3)** Imagine how does the checker work

{ [{ }] }

Stack	{	[			

push([ )

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work

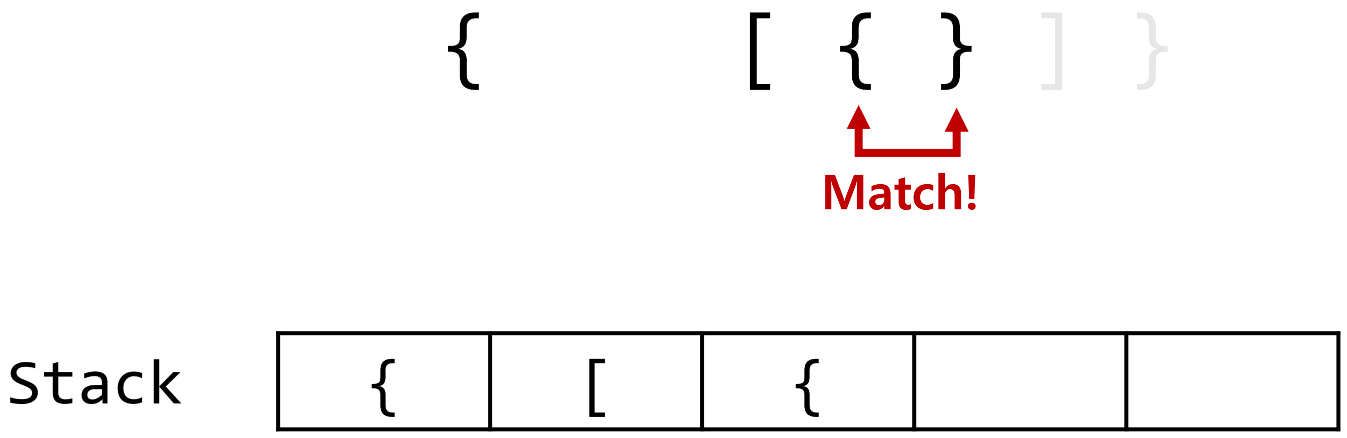
```
{     [ { } ] }
```

Stack	{	[	{		

```
push( { )
```

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work



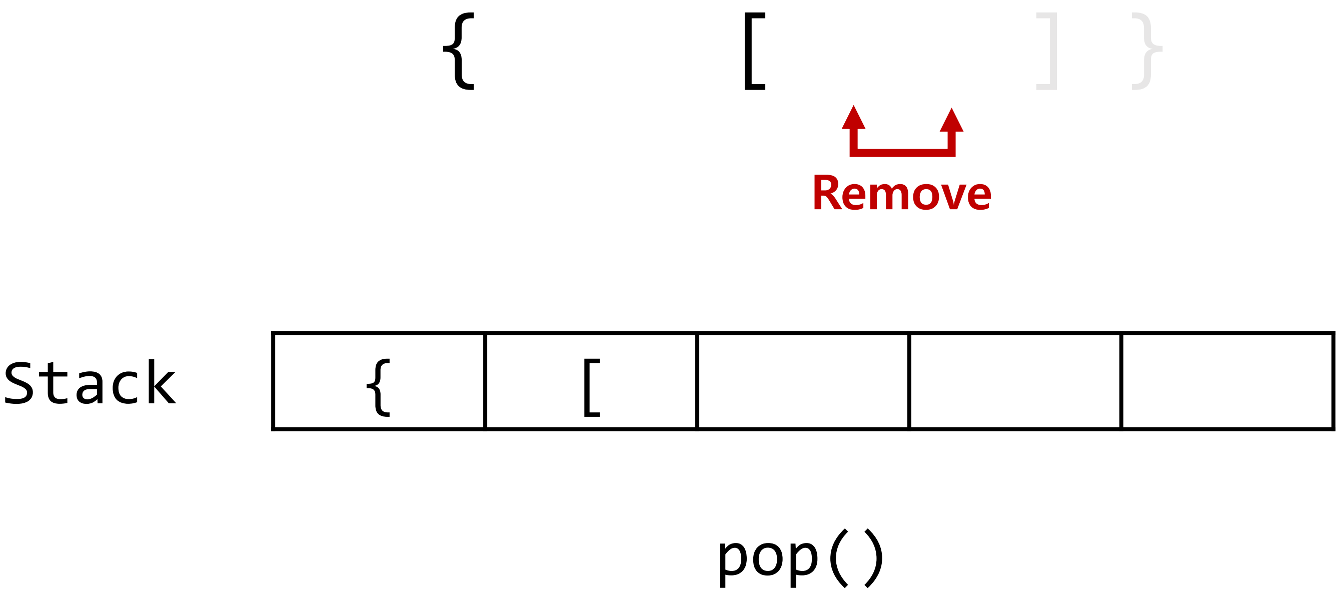
{ [{ }] }

Match!

Stack	{	[	{		

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work



{ [ ] }

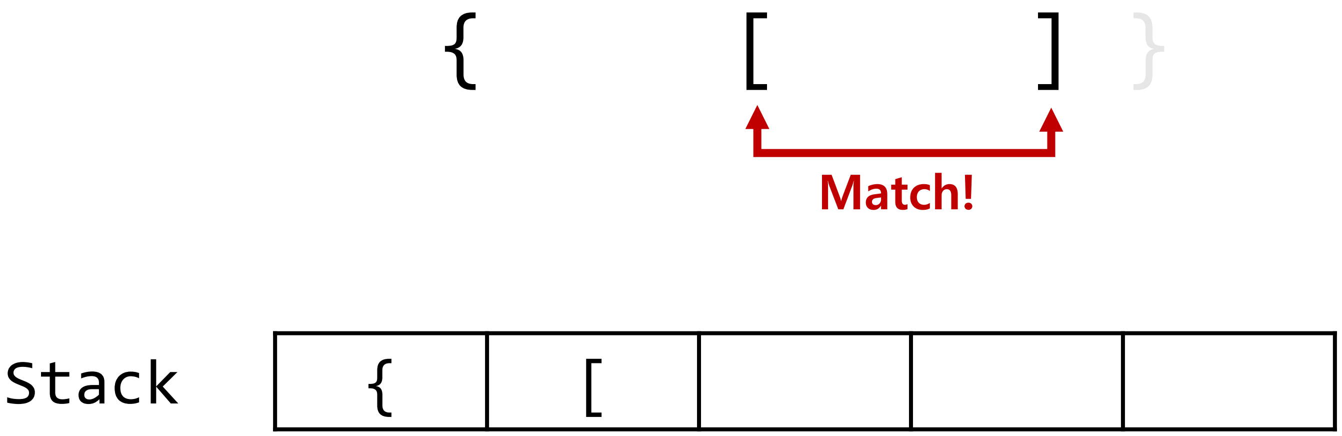
Remove

Stack	{	[			

pop()

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work



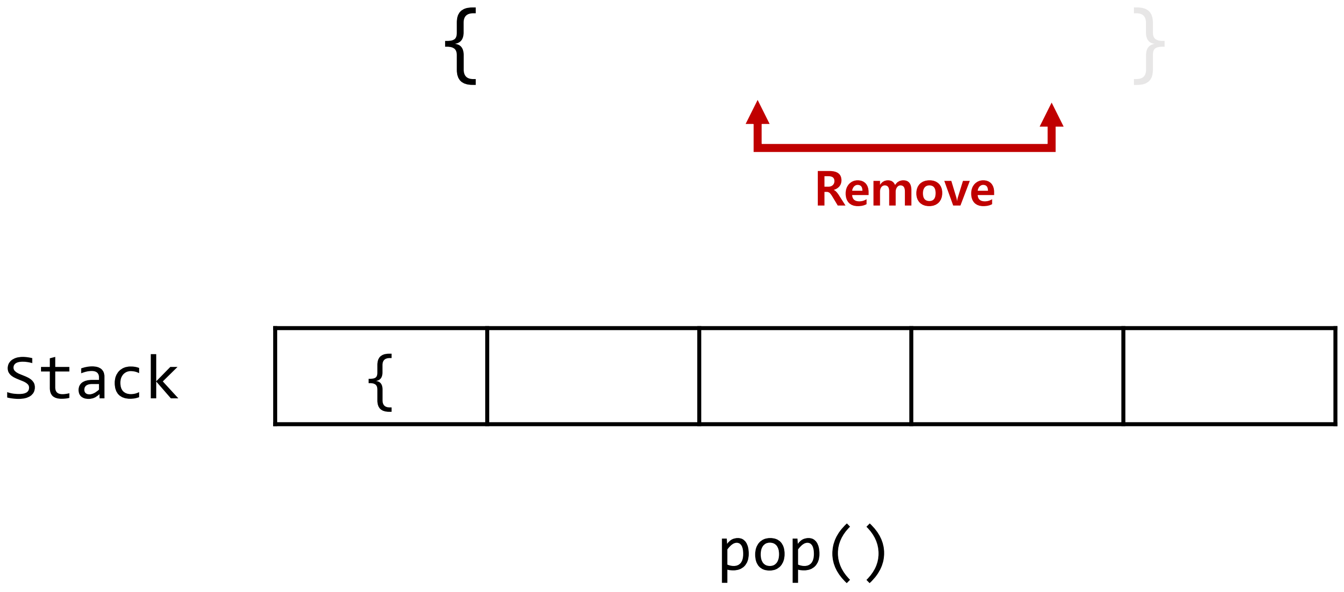
{ [ ] }

Match!

Stack	{	[			

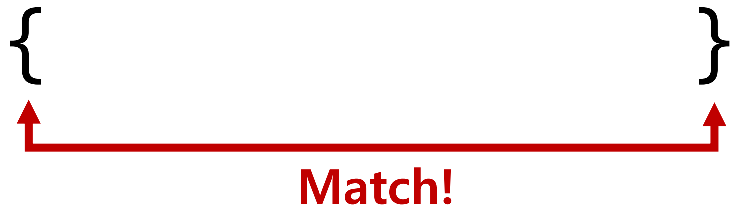
Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(Step 1)** Simplify the problem: remove everything except for the parentheses
  - **(Step 2)** Think the property of parentheses
  - **(Step 3)** Imagine how does the checker work



Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - (Q) How to implement the checker?
  - (Step 1) Simplify the problem: remove everything except for the parentheses
  - (Step 2) Think the property of parentheses
  - (Step 3) Imagine how does the checker work



Stack

{				

Stacks - Problem Solving Practice

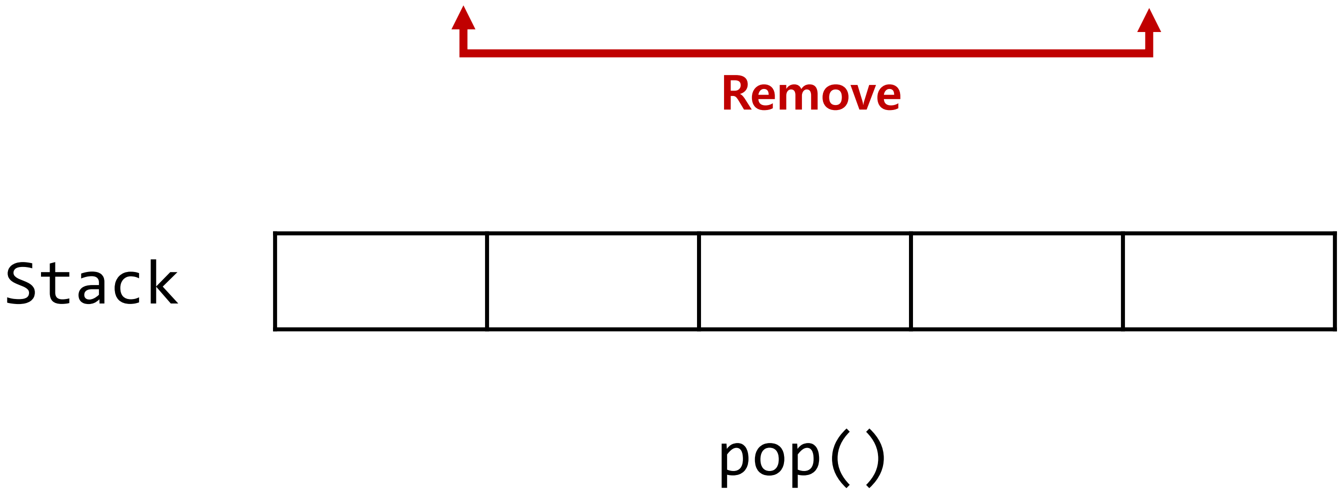
- Problem: **Parentheses Checker**

  (Q) How to implement the checker?

  (Step 1) Simplify the problem: remove everything except for the parentheses

  (Step 2) Think the property of parentheses

  (Step 3) Imagine how does the checker work



Remove

Stack

pop()

Stacks - Problem Solving Practice

- Problem: **Parentheses Checker**
  - **(Q)** How to implement the checker?
  - **(A)** Use the stack structure!

```
bool checkParentheses(char str[]) {
    int i;
    bool validity = true;
    Stack stack = createStack();
    for (i = 0; str[i] != '\0'; i ++) {
        // Write your own code
    }
    removeStack(&stack);
    return validity;
}
```

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**

  (Q) What is the postfix expression?

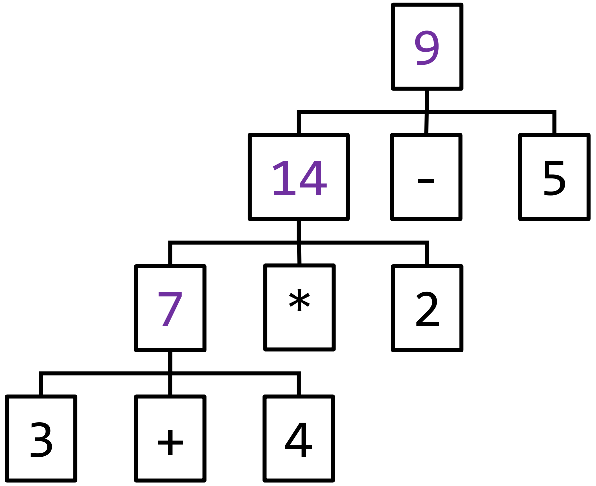
  - **Infix** expression: the operator is placed in between the operands (e.g., A + B)
  - **Postfix** expression: the operator is placed after the operands (e.g., A B +)

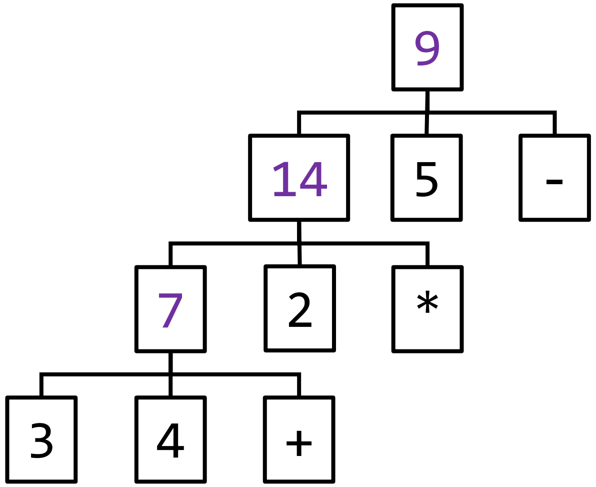
Infix Expression

```
((3 + 4) * 2) - 5
```

Postfix Expression

```
3 4 + 2 * 5 -
```





Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**

  (Q) What is the postfix expression?

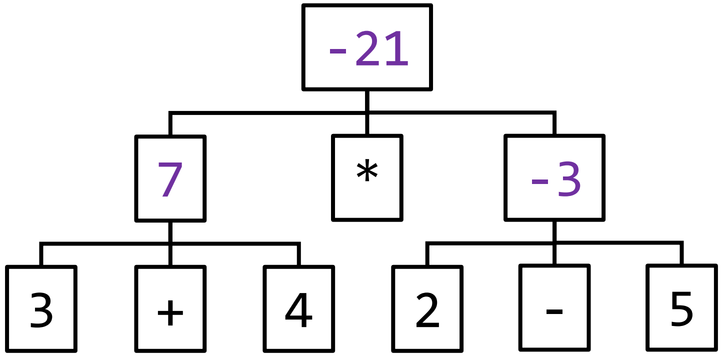
  - **Infix** expression: the operator is placed in between the operands (e.g., `A + B`)
  - **Postfix** expression: the operator is placed after the operands (e.g., `A B +`)

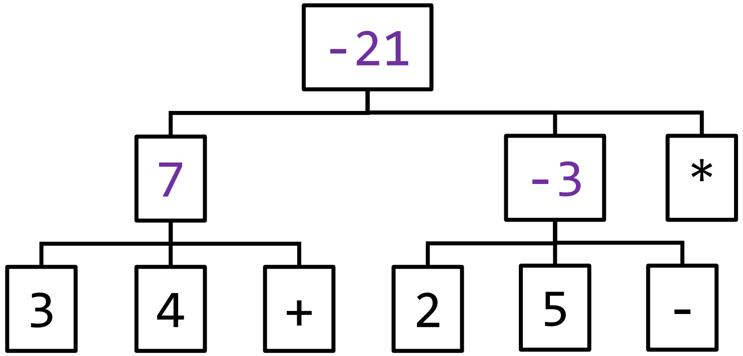
Infix Expression

`(3 + 4) * (2 - 5)`

Postfix Expression

`3 4 + 2 5 - *`





Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**

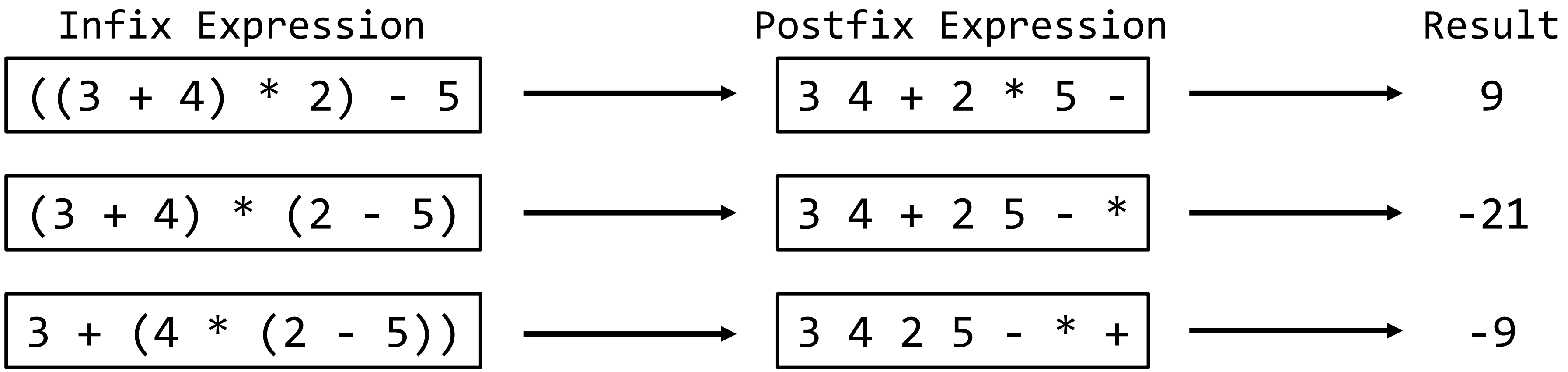
  (Q) What is the postfix expression?

  - **Infix** expression: the operator is placed in between the operands (e.g., `A + B`)
    - **(+)** Easy to understand and familiar to us
    - **(-)** Need parentheses for operation priority
  - **Postfix** expression: the operator is placed after the operands (e.g., `A B +`)
    - **(+)** Priority is simply left-to-right, so easy to implement
    - **(-)** parenthesis-free, i.e., no parenthesis is required for operation priority

Infix Expression	Postfix Expression
`((3 + 4) * 2) - 5`	`3 4 + 2 * 5 -`
`(3 + 4) * (2 - 5)`	`3 4 + 2 5 - *`
`3 + (4 * (2 - 5))`	`3 4 2 5 - * +`

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**
  - **(Q1)** How to evaluate a postfix expression?
  - **(Q2)** How to convert an infix expression into a postfix one?



Infix Expression	Postfix Expression	Result
`((3 + 4) * 2) - 5`	`3 4 + 2 * 5 -`	9
`(3 + 4) * (2 - 5)`	`3 4 + 2 5 - *`	-21
`3 + (4 * (2 - 5))`	`3 4 2 5 - * +`	-9

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**

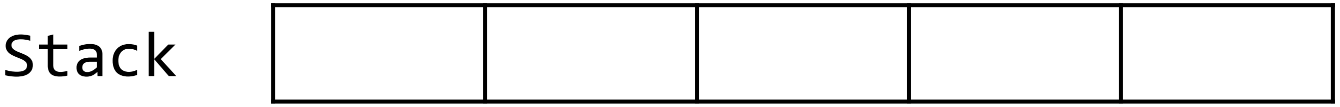
  (Q1) How to evaluate a postfix expression?

  (A1) Read operands and perform operators from left to right

  1. For operands, put the value into stack
  2. For operators, perform operators with two topmost elements

Postfix Expression

3 4 + 2 5 - *



Stack

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**

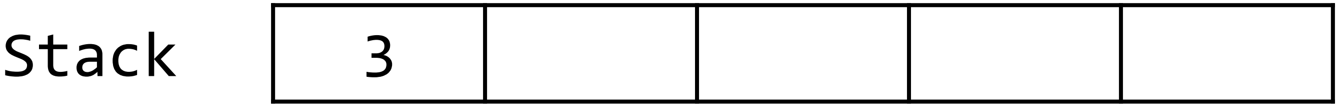
  (Q1) How to evaluate a postfix expression?

  (A1) Read operands and perform operators from left to right

  1. For operands, put the value into stack
  2. For operators, perform operators with two topmost elements

Postfix Expression

3 4 + 2 5 - *



Stack

3				

push(3)

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**

  (Q1) How to evaluate a postfix expression?

  (A1) Read operands and perform operators from left to right

  1. For operands, put the value into stack
  2. For operators, perform operators with two topmost elements

```
Postfix Expression

3 4 + 2 5 - *
```

Stack	3	4			

```
push( 4 )
```

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**
  - (Q1) How to evaluate a postfix expression?
  - (A1) Read operands and perform operators from left to right
    1. For operands, put the value into stack
    2. For operators, perform operators with two topmost elements

Postfix Expression

3 4 + 2 5 - *

Stack

7				

```
pop() → 4 , pop() → 3
push( 3 + 4 )
```

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**
  - **(Q1)** How to evaluate a postfix expression?
  - **(A1)** Read operands and perform operators from left to right
    1. For operands, put the value into stack
    2. For operators, perform operators with two topmost elements

Postfix Expression

3 4 + 2 5 - *

Stack

7	2			

push(2)

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**
  - (Q1) How to evaluate a postfix expression?
  - (A1) Read operands and perform operators from left to right
    1. For operands, put the value into stack
    2. For operators, perform operators with two topmost elements

Postfix Expression

3 4 + 2 5 - *

Stack	7	2	5		

push(5)

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**
  - **(Q1)** How to evaluate a postfix expression?
  - **(A1)** Read operands and perform operators from left to right
    1. For operands, put the value into stack
    2. For operators, perform operators with two topmost elements

```
Postfix Expression

3 4 + 2 5 - *
```

Stack	7	-3			

```
pop() → 5 , pop() → 2
push( 2 - 5 )
```

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**
  - **(Q1)** How to evaluate a postfix expression?
  - **(A1)** Read operands and perform operators from left to right
    1. For operands, put the value into stack
    2. For operators, perform operators with two topmost elements

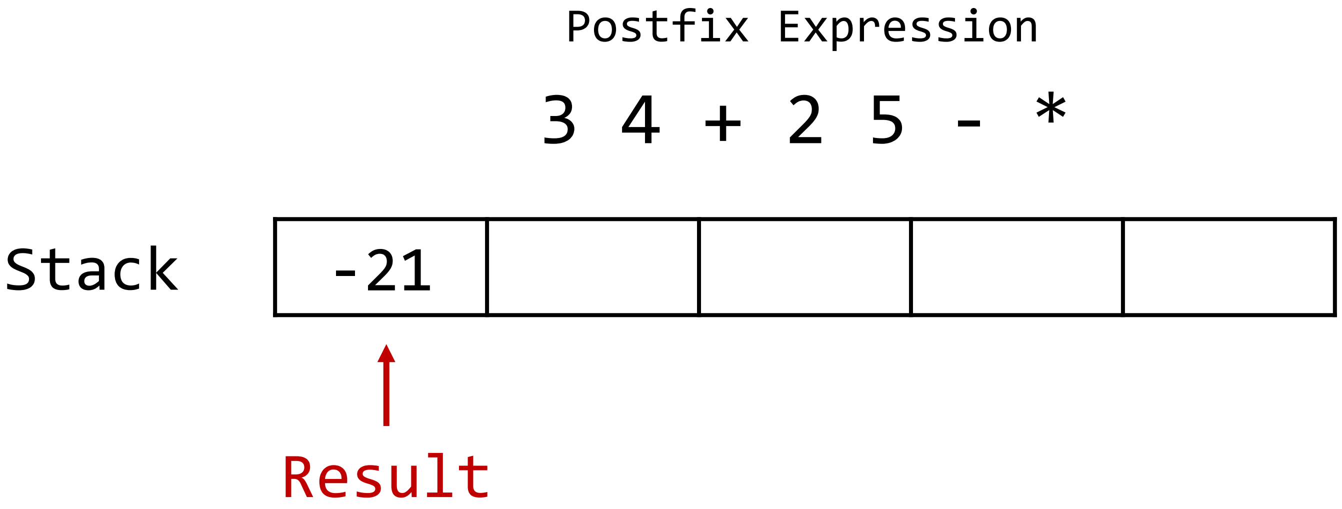
```
Postfix Expression

3 4 + 2 5 - *
```

Stack	-21				

```
pop() → -3 , pop() → 7
push( 7 * -3 )
```

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**
  - **(Q1)** How to evaluate a postfix expression?
  - **(A1)** Read operands and perform operators from left to right
    1. For operands, put the value into stack
    2. For operators, perform operators with two topmost elements



Postfix Expression

3 4 + 2 5 - *

Stack

-21

Result

Stacks - Problem Solving Practice

- Problem: **Postfix Expression Evaluation**

  (Q1) How to evaluate a postfix expression?

  (A1) Read operands and perform operators from left to right

```
int evaluatePostfix(char str[]) {
    int i, result;
    Stack stack = createStack();
    for (i = 0; str[i] != '\0'; i ++) {
        if (str[i] == '+') { }      // Write your own code
        else if (str[i] == '-') { } //
        else if (str[i] == '*') { } //
        else if (str[i] == '/') { } //
        else { }                    //
    }
    result = peek(&stack);
    removeStack(&stack);
    return result;
}
```

Any Questions?

1398