Data Structures

CSCI 2270-202: REC 05

Sanskar Katiyar

Logistics

Make-up for Assignment [Interview Grading]

2 opportunities in the semester

1 after each midterm

Office Hours at ECAE 128

Wednesday: 3 pm – 5 pm

Thursday: 5 pm – 6 pm

Friday: 3 pm – 5 pm

Logistics

Still working on Notes for Assignment 4

Will be out tonight! Promise!

Recitation Materials *(Notes, Slides, Code, etc.)*

sanskarkatiyar.github.io/CSCI2270

Logistics: Midterm I

Feb 21 (Fri): 5 pm – 7 pm

Alternate Date

Feb 19 (Wed): 5 pm – 7 pm

Fill form, link on Moodle (**Deadline Today**)

Special Accommodations

Fill form, link on Moodle (**Deadline Tomorrow**)

Recitation Outline

1. Abstract Data Type
2. Stack
3. Queue
4. Exercise

Abstract Data Type

An ADT is composed of:

Collection of **data members**

Operations defined on the ADT

Language independent

Abstract Data Type

Why do we use ADTs?

No perfect Data Structure!

Problem Solving and Algorithms

Efficiency: Lookup, Insertion, Deletion, Storage, etc.

What CSCI 2270 is about?

Understanding popular Data Structures

Implementing these Data Structures

Stack ADT

LIFO: Last In First Out

Data Members*

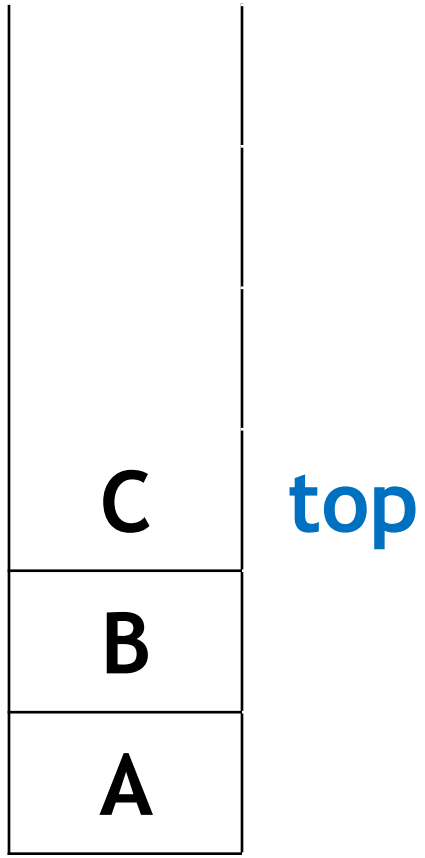
Top: The element at the ***top of the stack***

Operations

Push: ***Insert an element*** at the top of the stack

Pop: ***Delete the top element*** from the stack

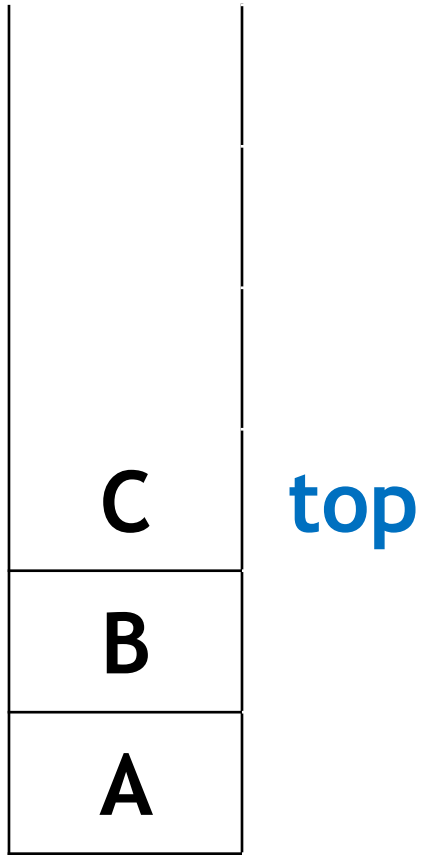
Peek: ***Returns the element at the top*** of the stack



Stack ADT: push

***Pseudocode*: Push**

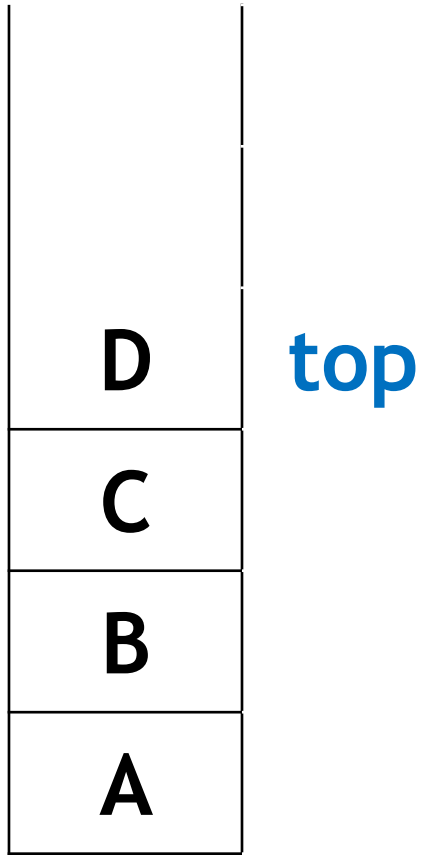
```
Stack.push("D");
```



Stack ADT: push

***Pseudocode*: Push**

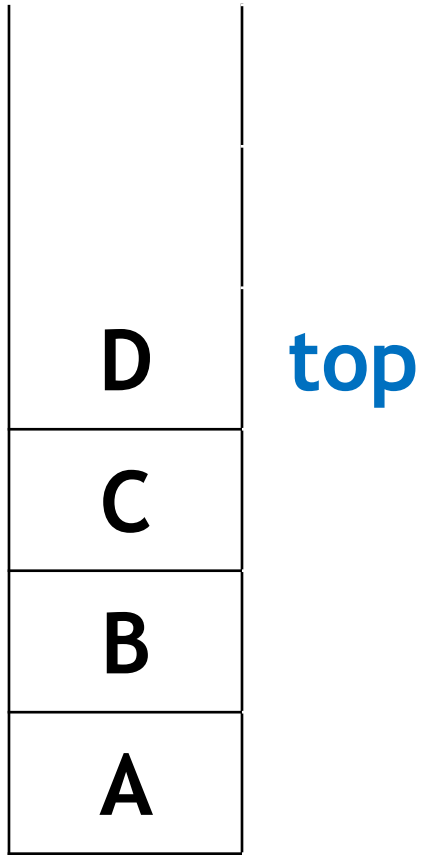
```
Stack.push("D");
```



Stack ADT: peek

***Pseudocode*: Peek**

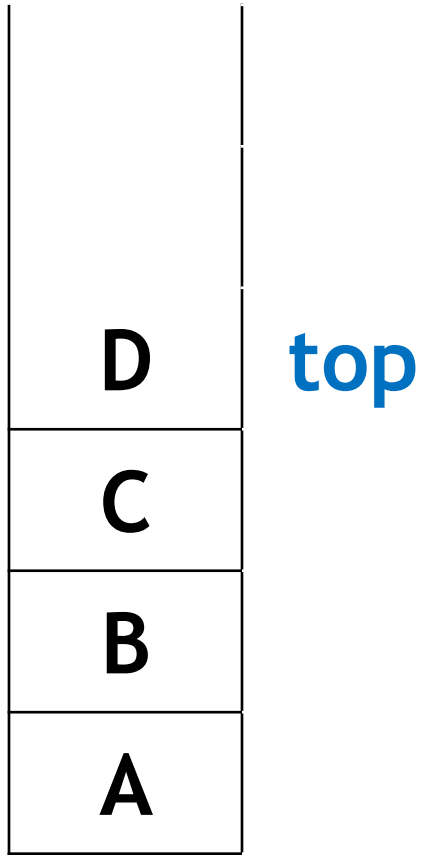
```
var = Stack.peek();

print var;

Output: D
```



Stack ADT: pop

***Pseudocode*: Pop**

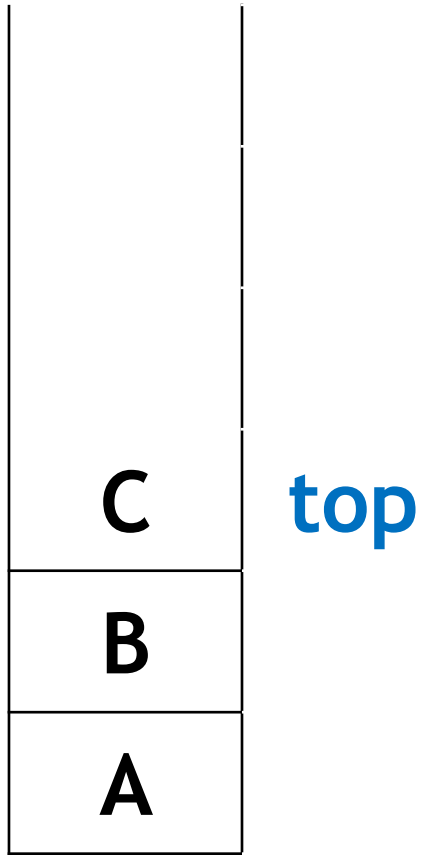
```
Stack.pop();
```



Stack ADT: pop

***Pseudocode*: Pop**

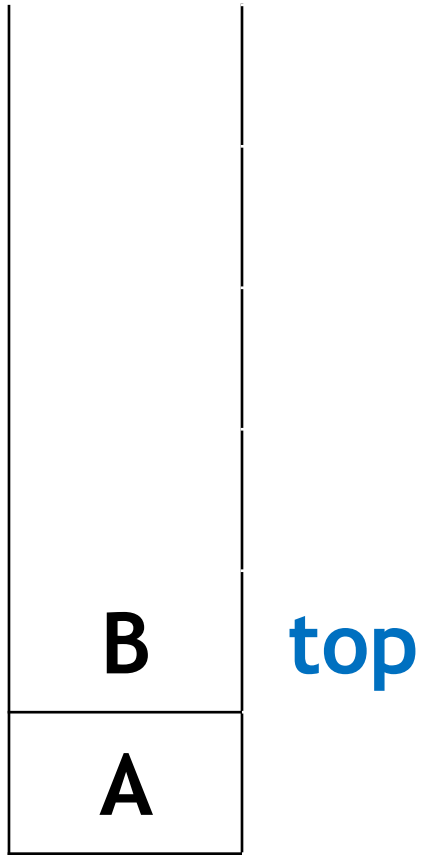
```
Stack.pop();
```



Stack ADT: pop

***Pseudocode*: Pop**

```
Stack.pop();
Stack.pop();
```

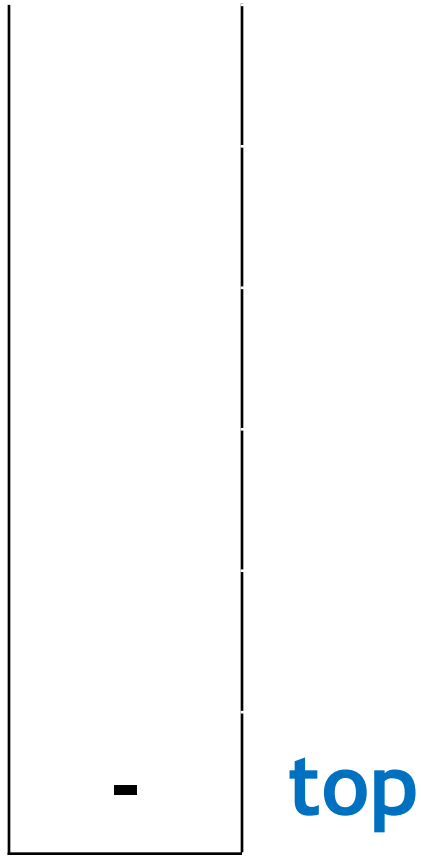


Stack ADT: Empty

Pseudocode

```
// Stack is Empty

Stack.top == NULL;
```

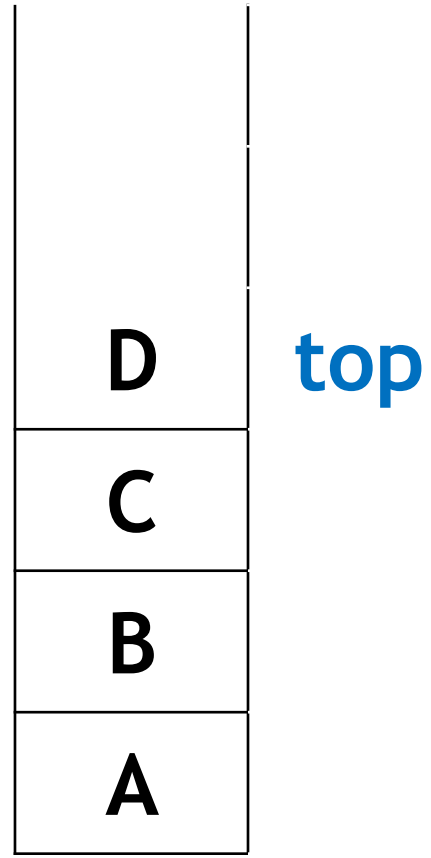
Generally, we can't access `TOP`. Need to use a helper function, such as `isEmpty()`.



Stack ADT: LIFO

Pseudocode

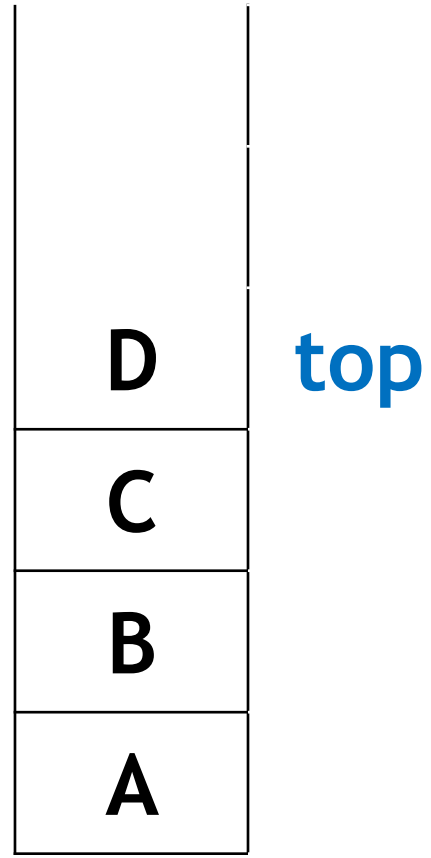
```
Stack.push("A");
Stack.push("B");
Stack.push("C");
Stack.push("D");
```



Stack ADT: LIFO

Pseudocode

```
while(!Stack.isEmpty())
   print Stack.peek();
   Stack.pop();


Output: D, C, B, A
```

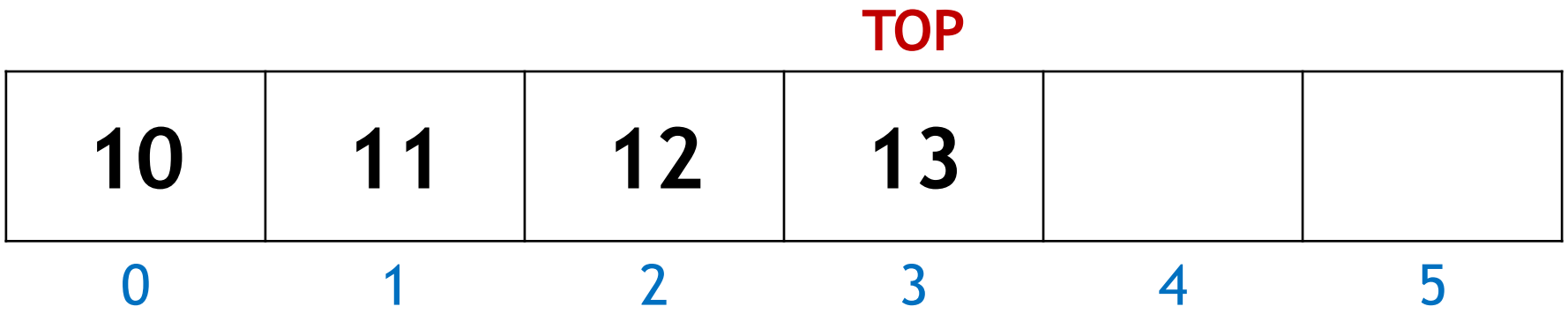


Stack: Array Implementation

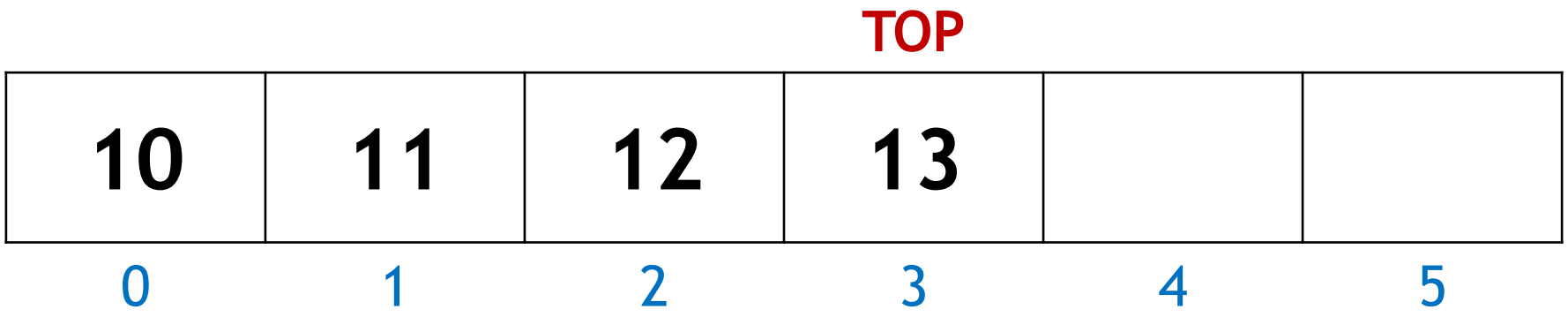
Pro: No Pointers! None! NULL!

Con: Not dynamic. Cannot change size at runtime.

Initialize `TOP = -1`. Check against size of array.



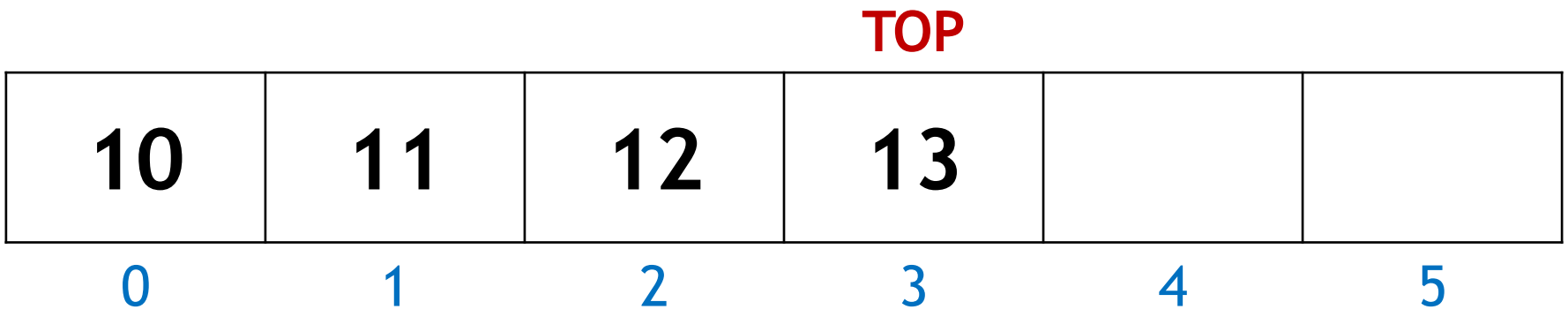
StackArray: push



1. Check if index [TOP + 1] is available.

```
if (TOP + 1 == array_size) {return OVERFLOW};
```

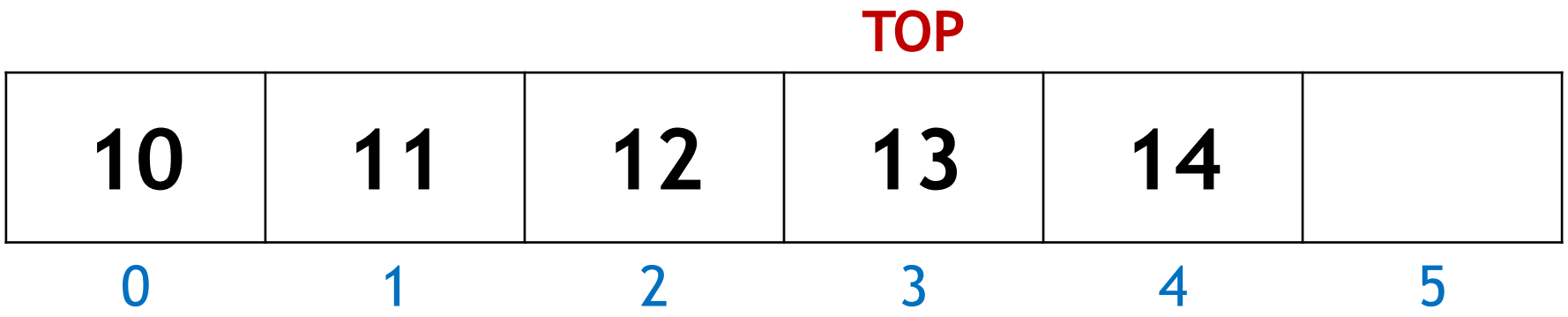
StackArray: push



2. If index [TOP + 1] is available, then insert element.

```
else { Stack[TOP + 1] = 14; }
```

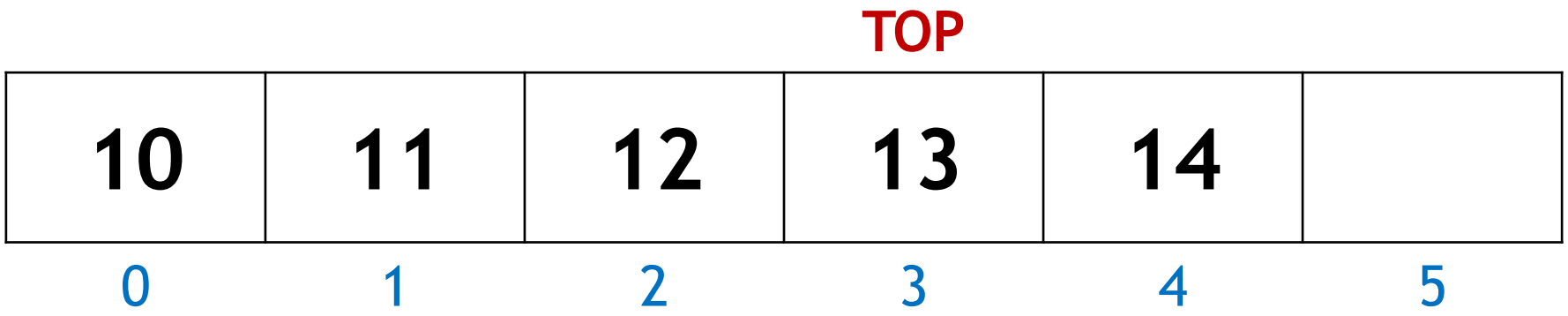
StackArray: push



2. If index [TOP + 1] is available, then insert element.

```
else { Stack[TOP + 1] = 14; }
```

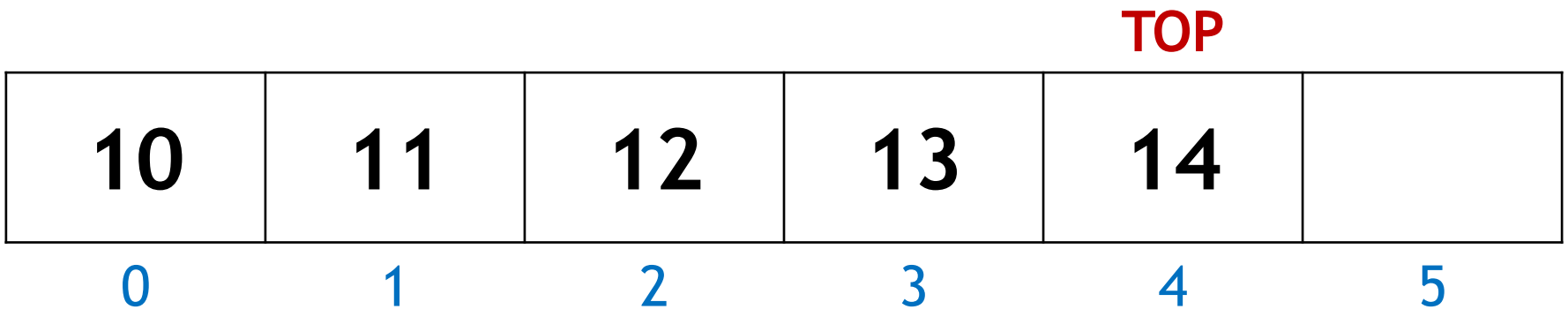
StackArray: push



3. Update TOP to new index.

```
TOP = TOP + 1;
```

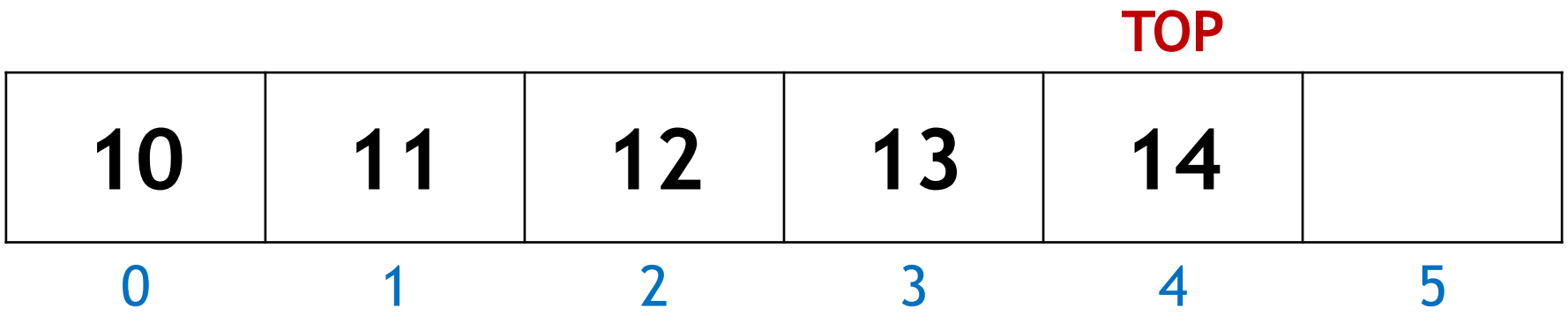
StackArray: push



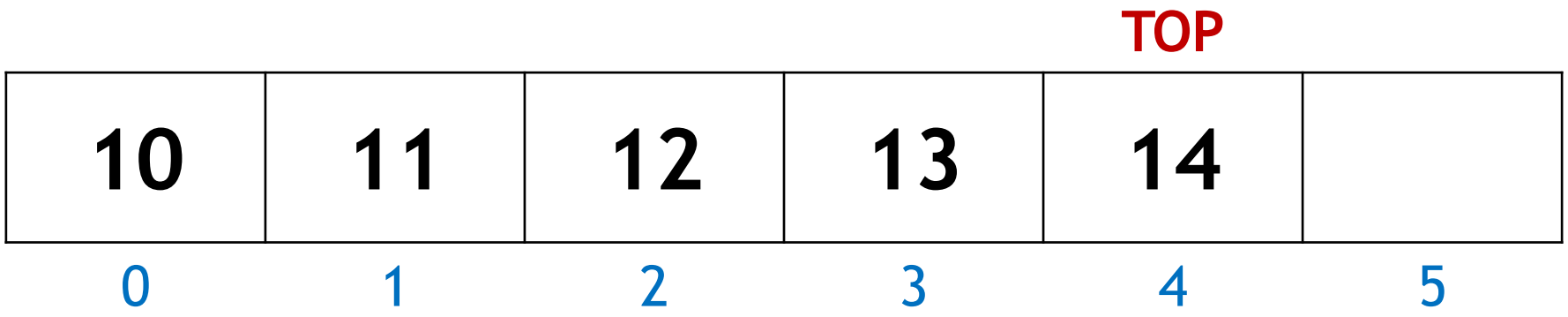
3. Update TOP to new index.

```
TOP = TOP + 1;
```

StackArray: push



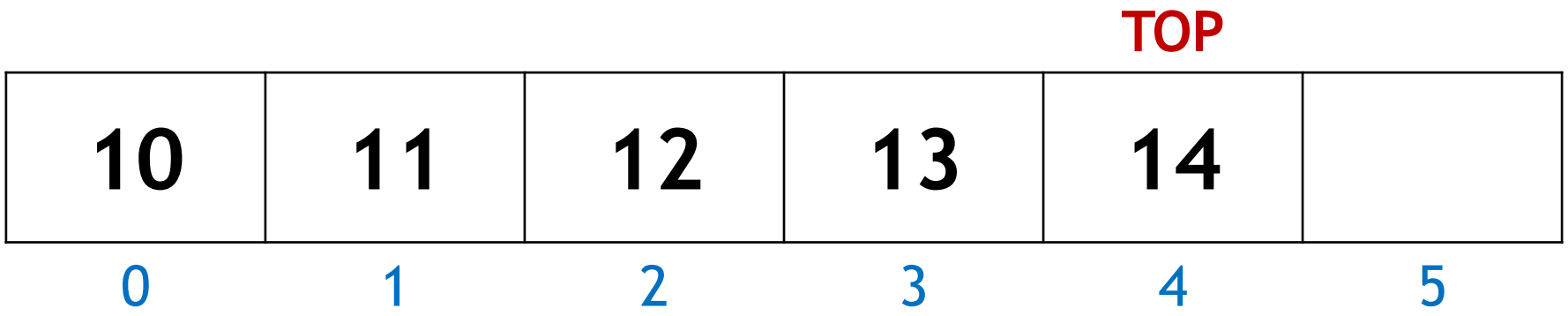
StackArray: pop



1. Check if Stack is Empty.

```
if (TOP == -1) { return UNDERFLOW };
```

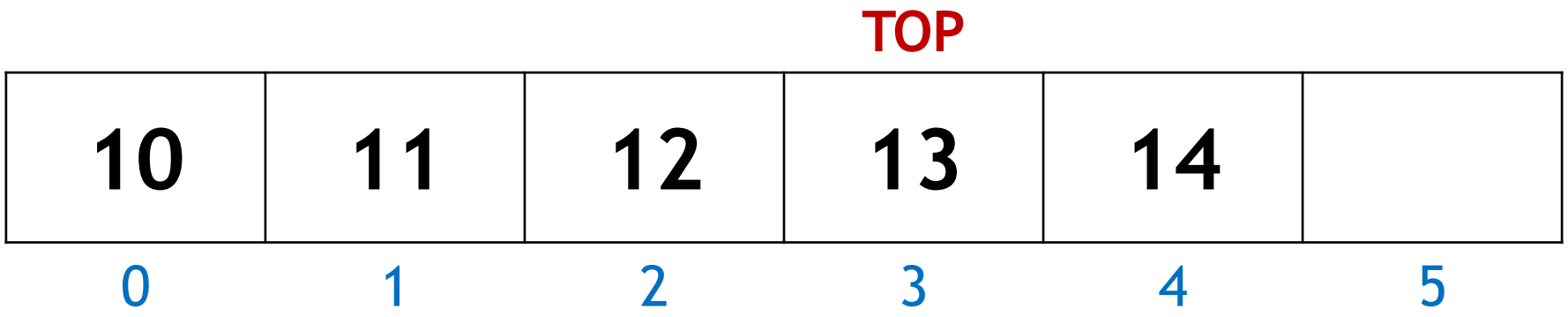
StackArray: pop



2. Decrement TOP.

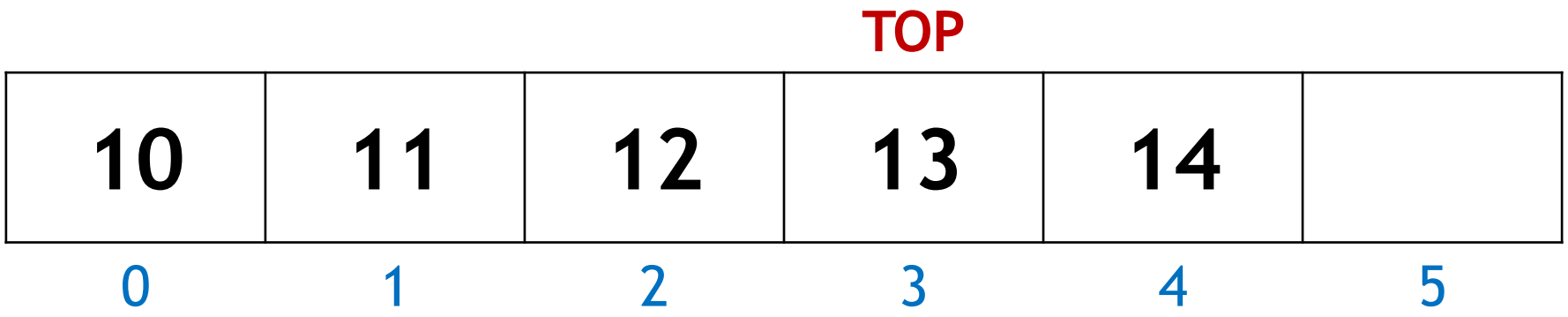
```
else { TOP = TOP - 1; }
```

StackArray: pop



Q. *What about 14, it's still in the array?*

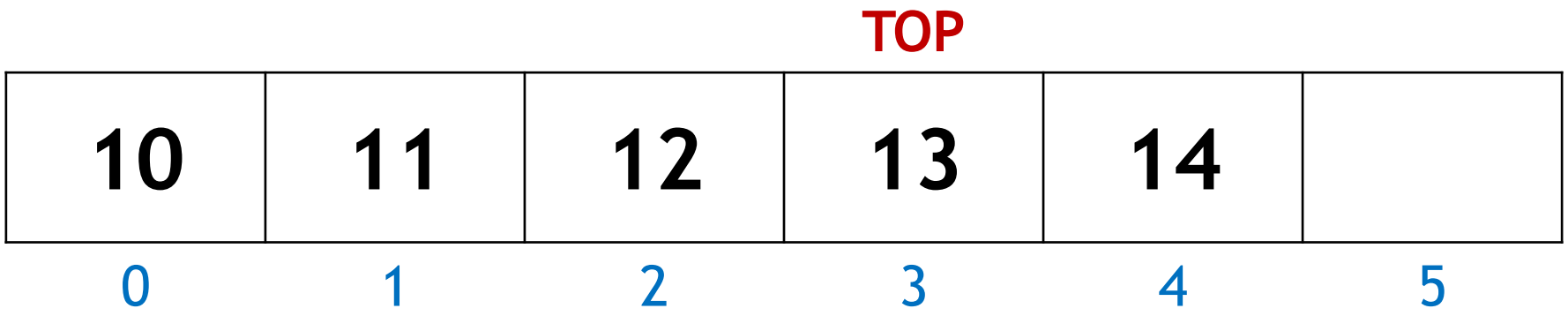
StackArray: peek



1. Check if Stack is Empty.

```
if (TOP == -1) { return EMPTY };
```

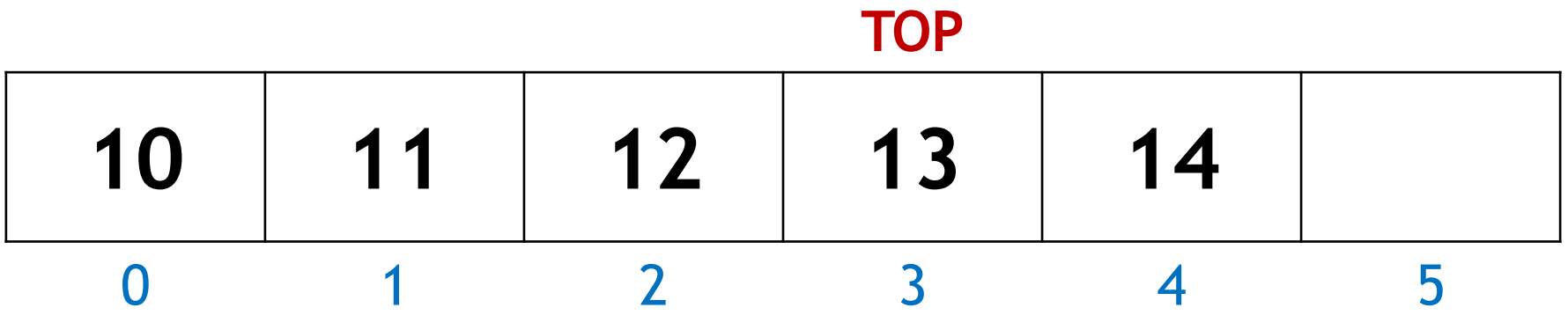
StackArray: peek

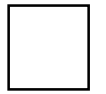


2. Else return the TOP element.

```
else { return Stack[TOP]; }
```

StackArray





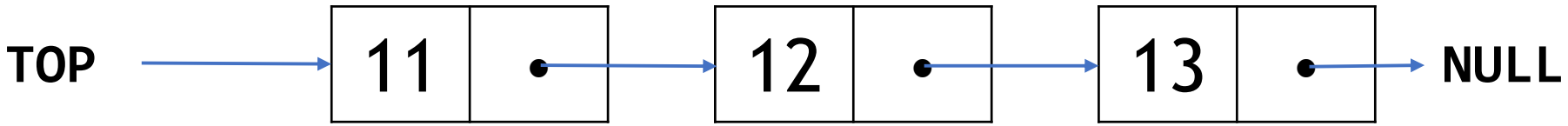
Stack: Linked List Implementation

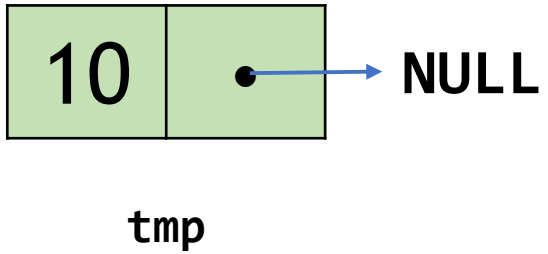
Pro: Dynamically grow and shrink the Stack

Con: Extra memory usage due to pointers

Stack	Stack (Linked List)
top	head
push	Insert at head
pop	Delete at head
peek	Return head's data

StackLL: push

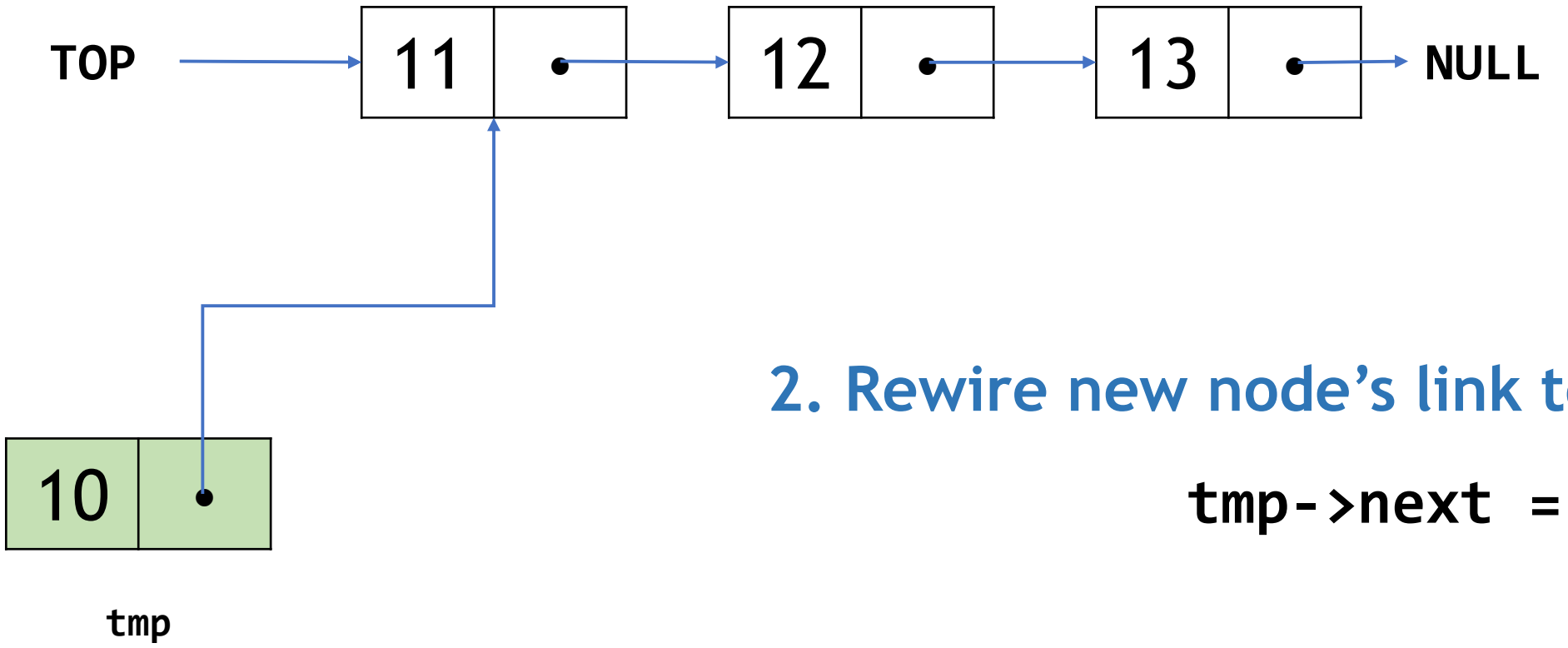




1. Create new node (to be pushed)

```
Node *tmp = new Node({10, NULL});
```

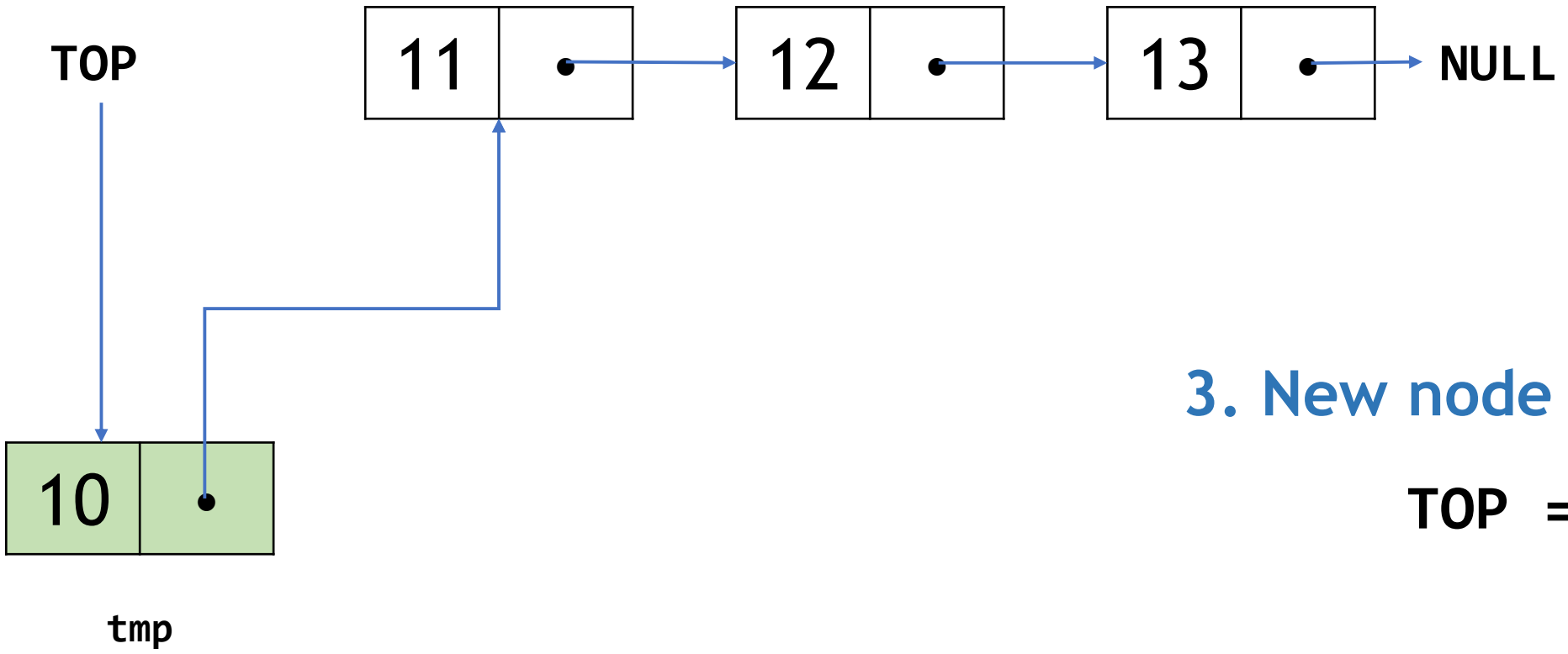
StackLL: push



2. Rewire new node's link to TOP

```
tmp->next = TOP;
```

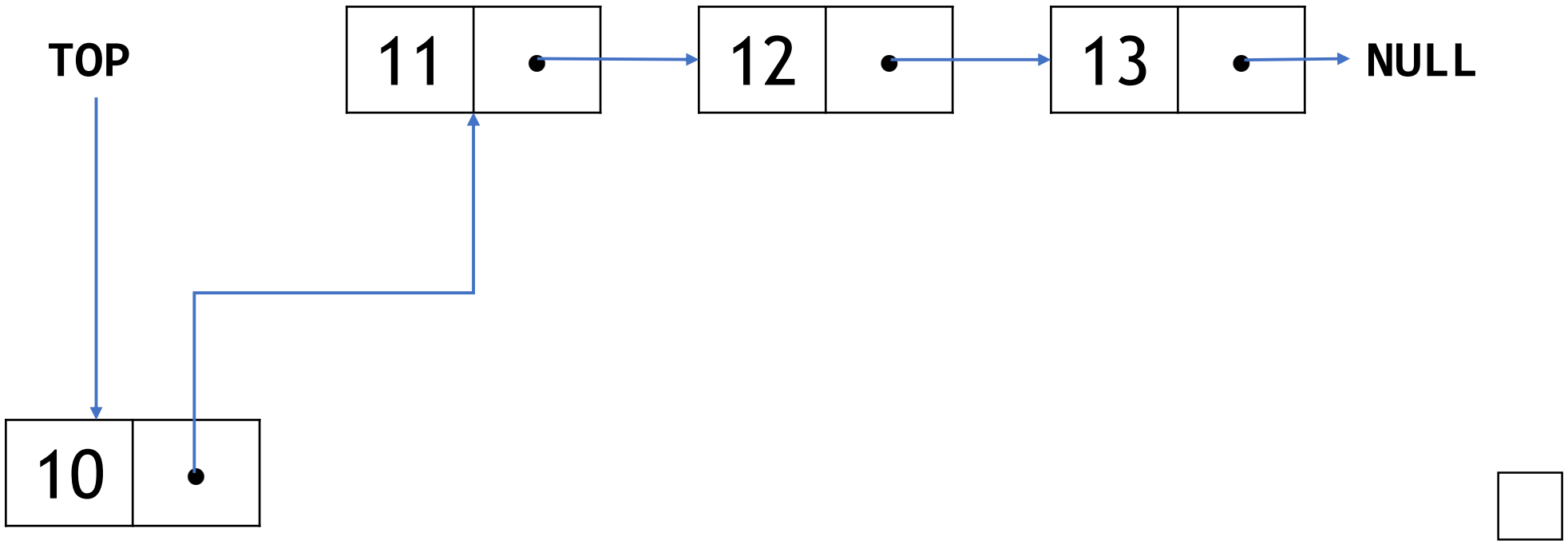
StackLL: push



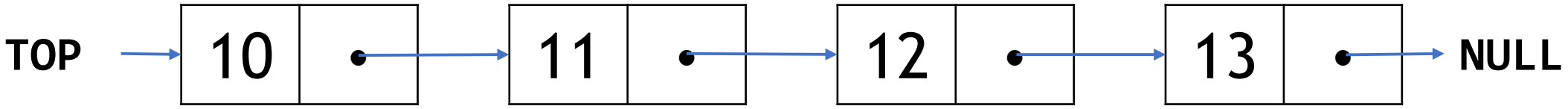
3. New node is TOP

```
TOP = tmp;
```

StackLL: push



StackLL: pop

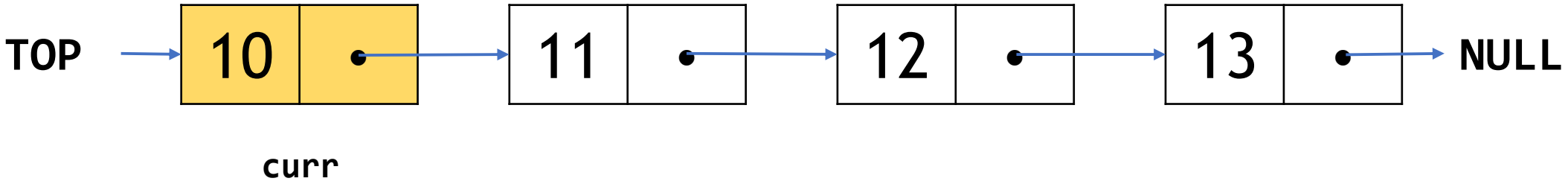


StackLL: pop

0. Check if TOP exists

```
if (!TOP) { return UNDERFLOW };
```

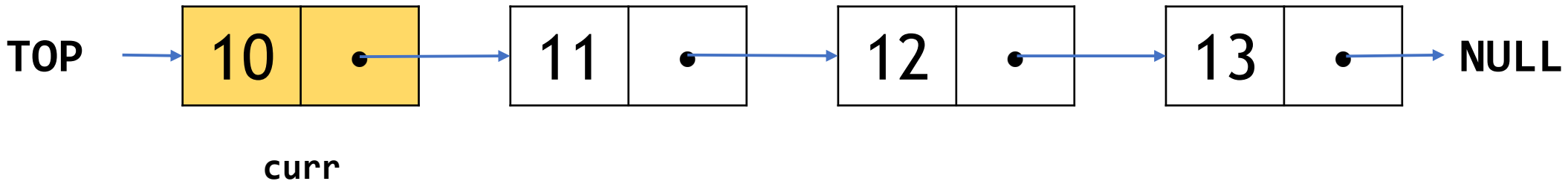
StackLL: pop



1. Create temp (pointer to) TOP

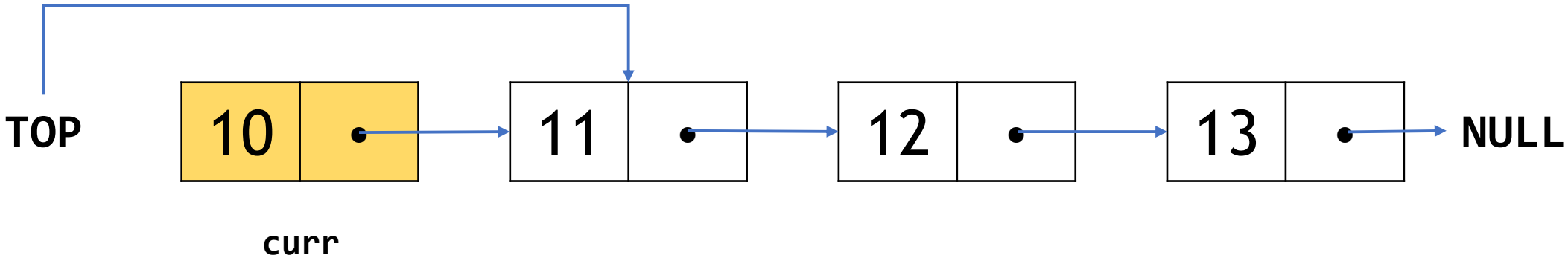
```
Node* curr = TOP;
```

StackLL: pop



2. Modify TOP pointer to next node

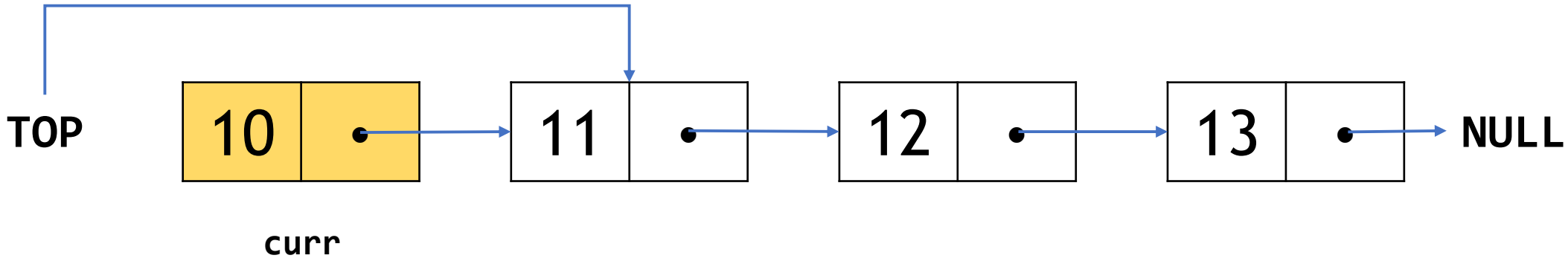
StackLL: pop



2. Modify TOP pointer to next node

```
TOP = TOP->next;
```

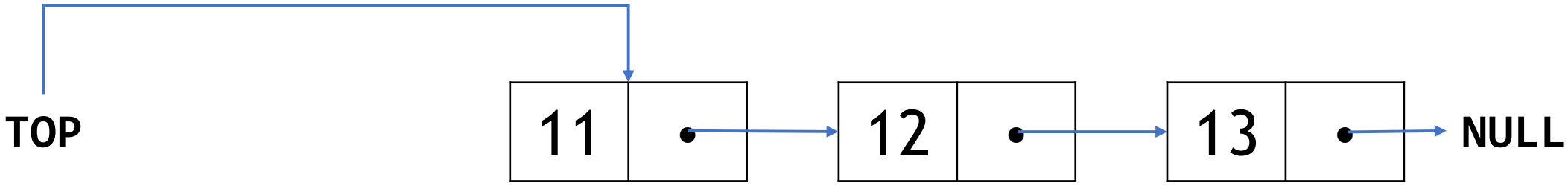
StackLL: pop

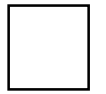


3. Free the node to be deleted

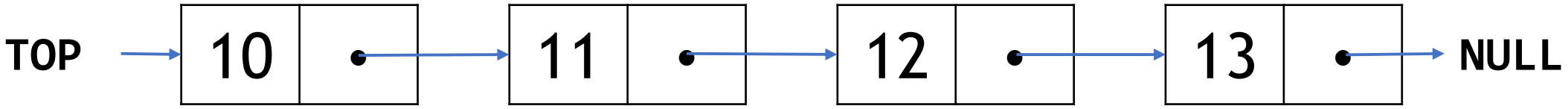
```
delete curr;
```

StackLL: pop





StackLL: peek



Check if TOP exists; Return data

```
if (TOP) {return TOP->data;}
```

Stack: Linked List Implementation

Pro: Dynamically grow and shrink the Stack

Con: Extra memory usage due to pointers

Stack	Stack (Linked List)
top	head
push	Insert at head
pop	Delete at head
peek	Return head's data

Queue ADT

FIFO: First In First Out

Data Members*

- **Head**: The element at the ***front of the queue***
- **Tail**: The element at the ***end of the queue***

Operations

- **Enqueue**: ***Insert an element*** at the end of the queue
- **Dequeue**: ***Delete the element*** at the front of the queue
- **Peek**: ***Returns the element at the front*** of the queue

Queue ADT: Enqueue

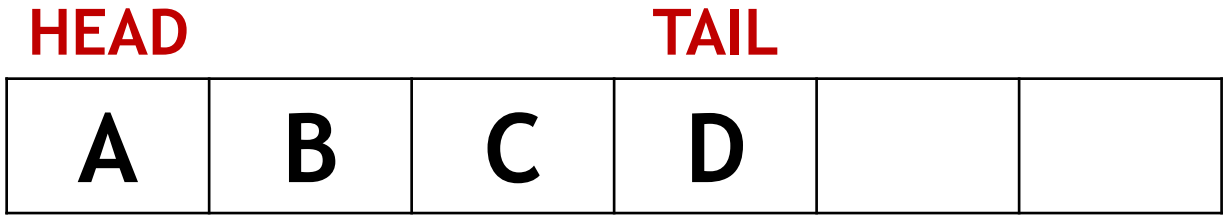
***Pseudocode*: Enqueue**

```
Queue.enqueue("D");
```

Queue ADT: Enqueue

***Pseudocode*: Enqueue**

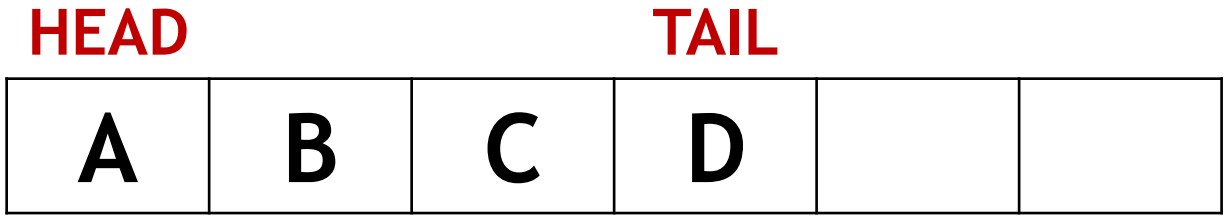
```
Queue.enqueue("D");
```



Queue ADT: Dequeue

***Pseudocode*: Dequeue**

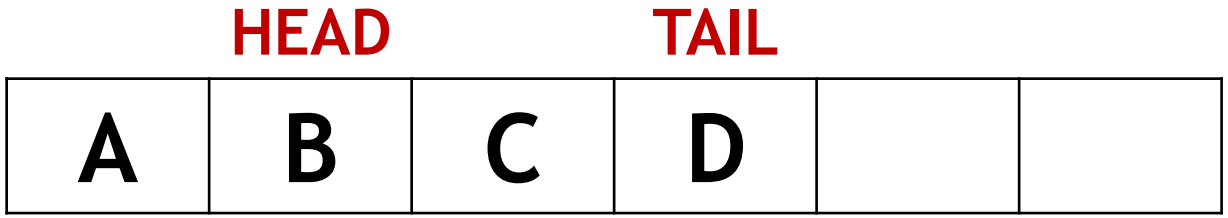
```
Queue.dequeue();
```



Queue ADT: Dequeue

***Pseudocode*: Dequeue**

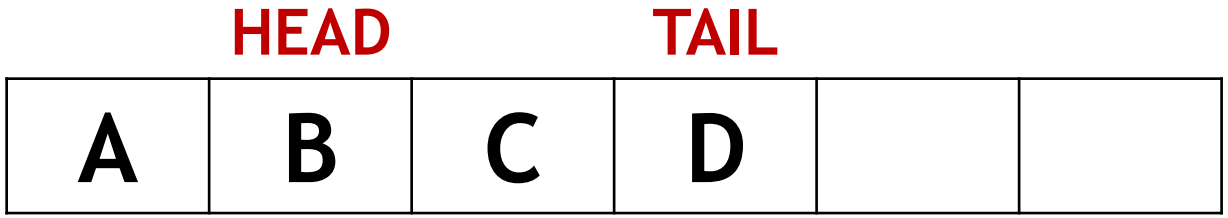
```
Queue.dequeue();
```



Queue ADT: Peek

***Pseudocode*: peek**

```
var = Queue.peek();

print var;

Output: B
```



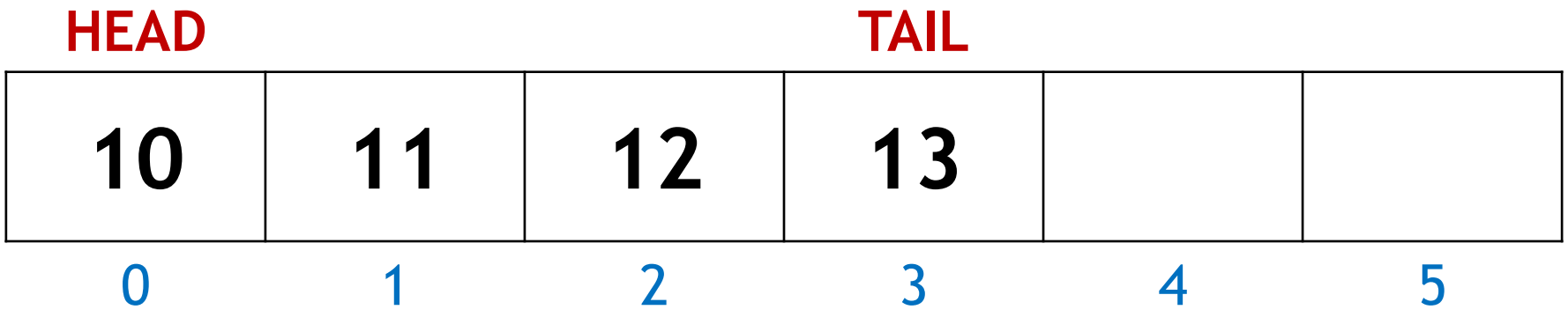
Queue: Array Implementation

[Different from Recitation Writeup]

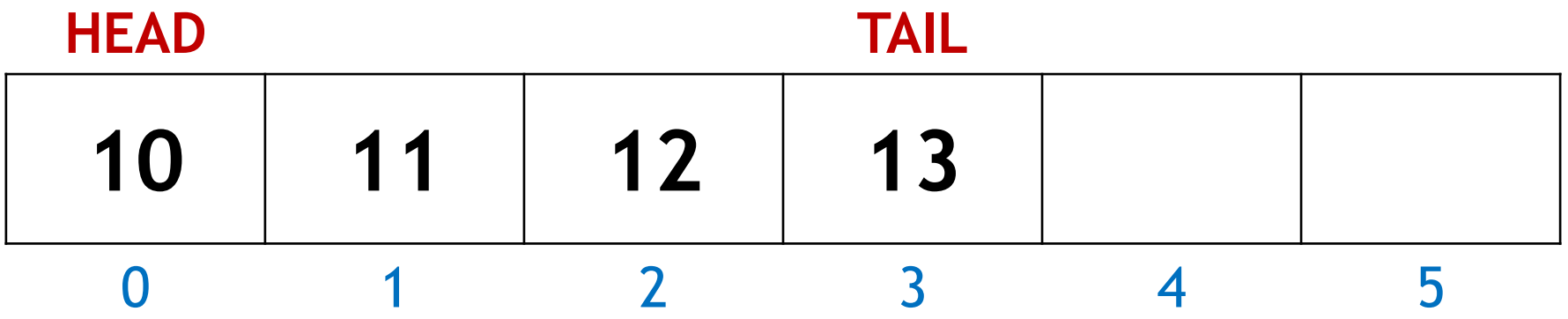
Pro: No Pointers!

Con: Not dynamic. Cannot change size at runtime.

Initialize `HEAD = -1, TAIL = -1`.



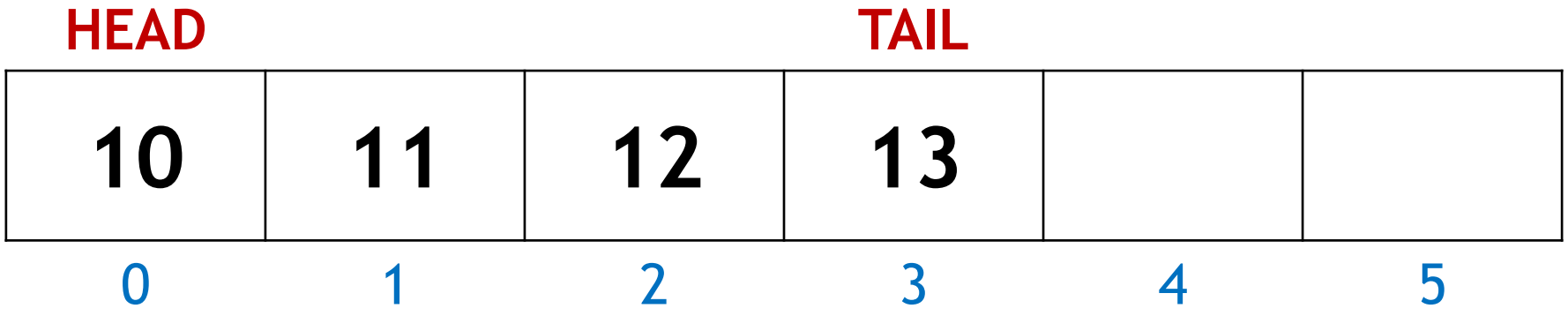
QueueArray: Enqueue



0. Check if there is space in the array

```
if (TAIL + 1 == array_size) { return OVERFLOW; }
```

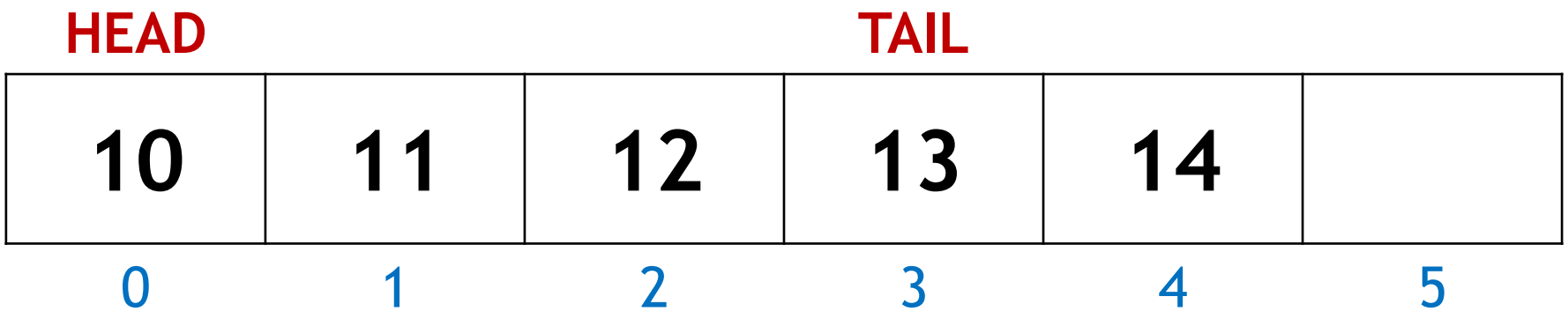
QueueArray: Enqueue



1. Else, Insert the element in the queue

```
Queue[TAIL + 1] = 14;
```

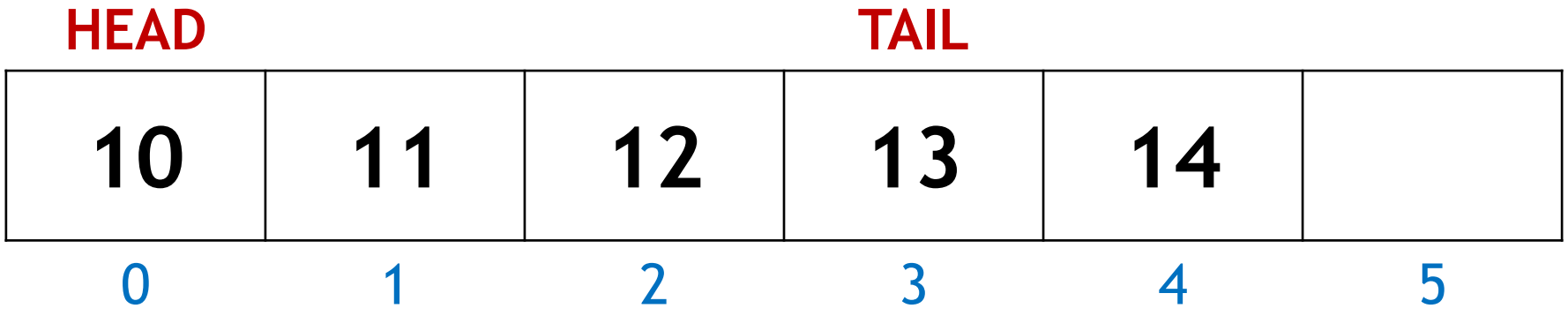
QueueArray: Enqueue



1. Else, Insert the element in the queue

```
Queue[TAIL + 1] = 14;
```

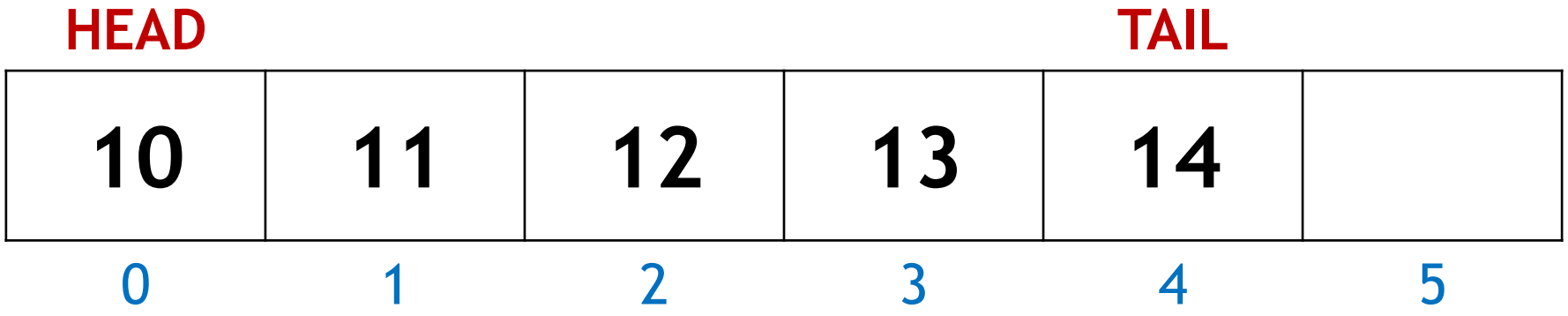
QueueArray: Enqueue



2. Update TAIL

```
TAIL = TAIL + 1;
```

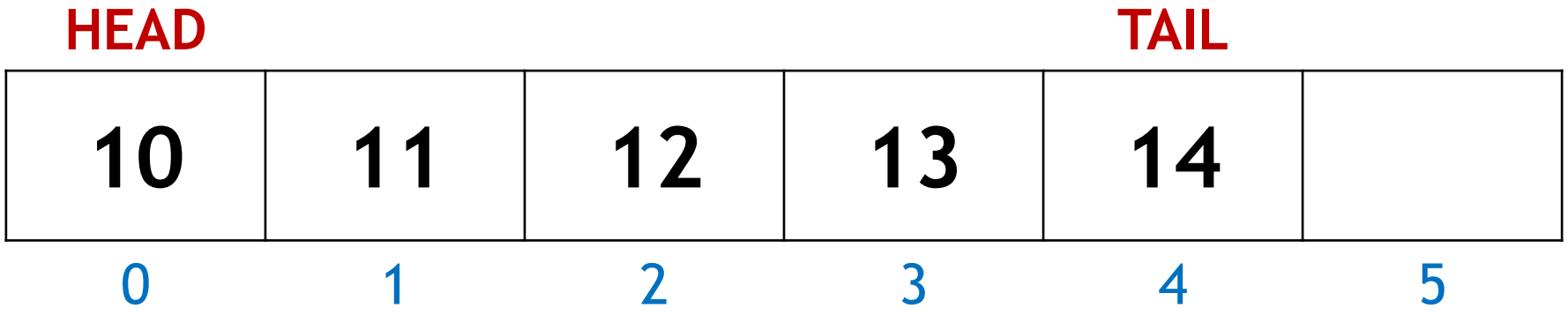
QueueArray: Enqueue



2. Update TAIL

```
TAIL = TAIL + 1;
```

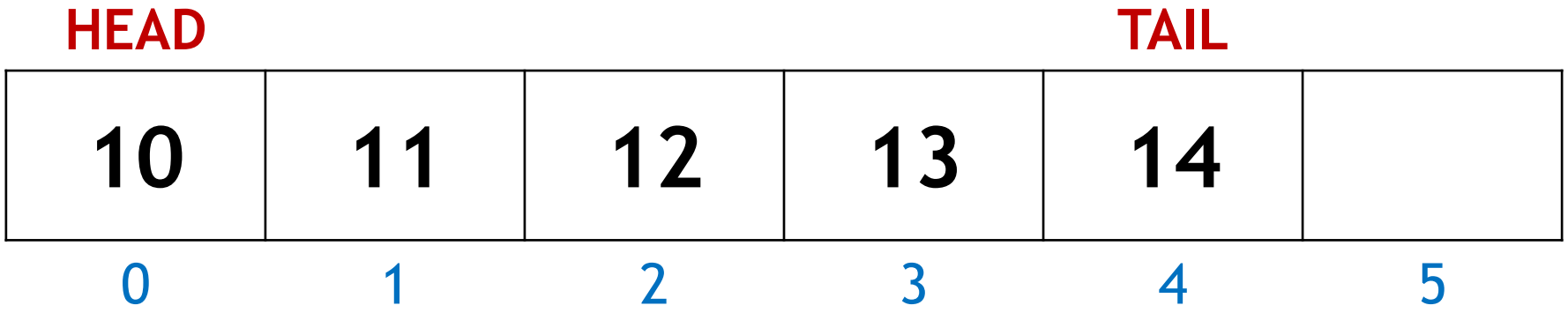
QueueArray: Enqueue

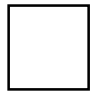


3. Check if HEAD needs to be updated [Edge Case]

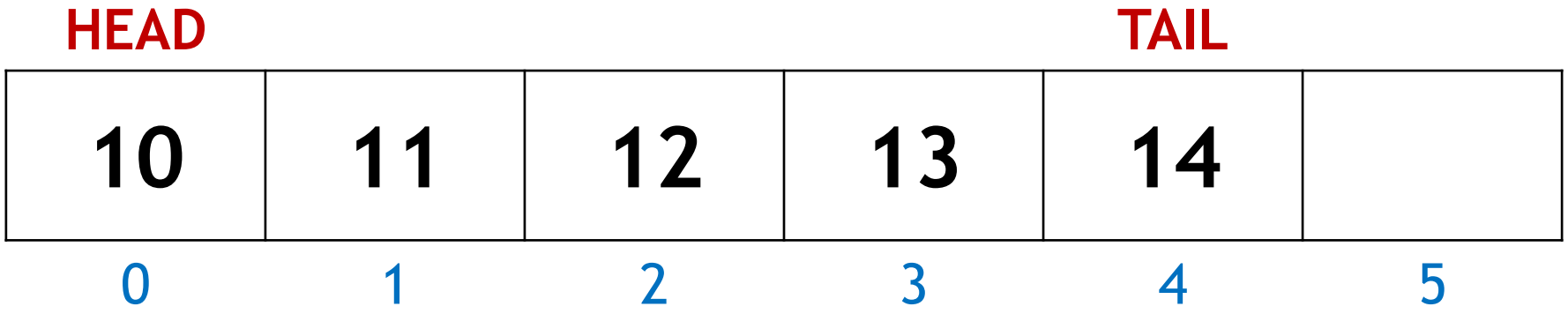
```
if (HEAD == -1) { HEAD = HEAD + 1; }
```

QueueArray: Enqueue





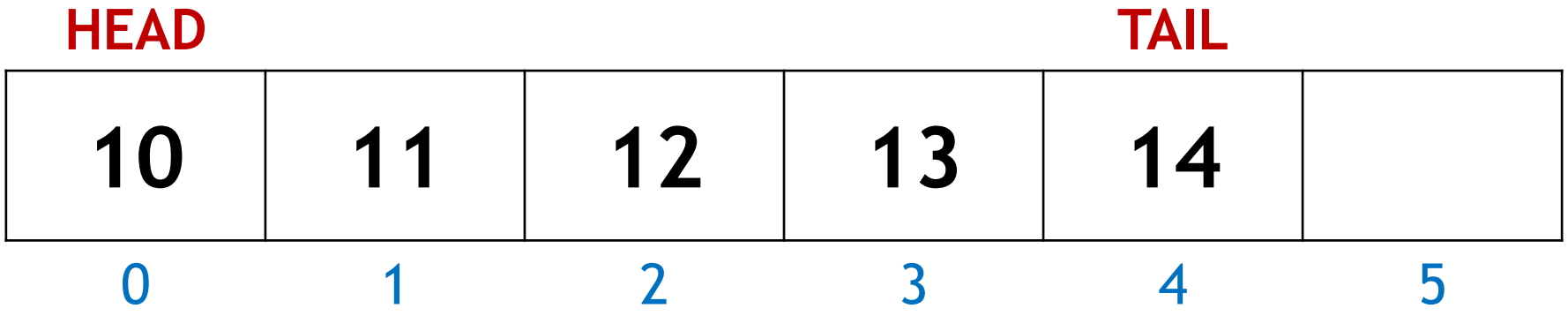
QueueArray: Dequeue



0. Check HEAD is a valid index [Edge Case]

```
if (HEAD < 0) { return UNDERFLOW;}
```

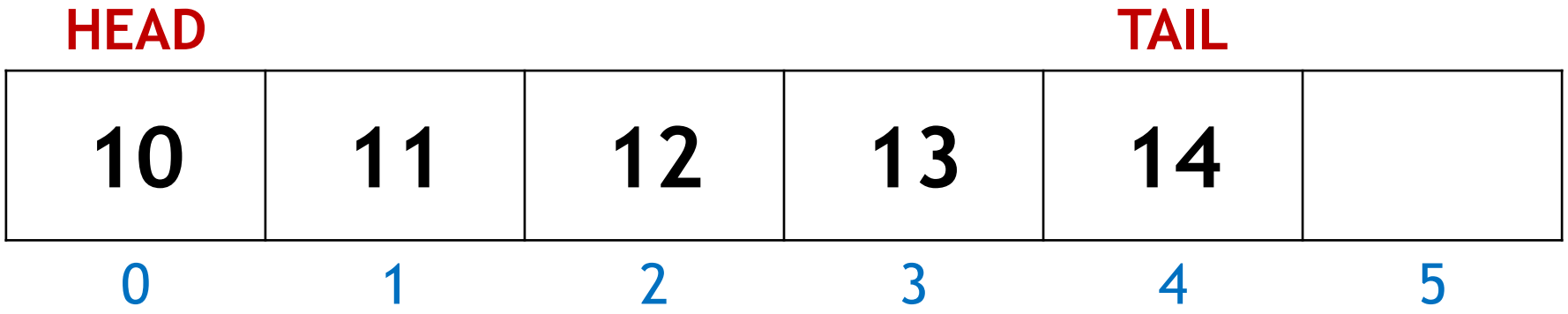
QueueArray: Dequeue



1. Check if there is only one element in queue [Edge Case]

```
if (HEAD == TAIL) { HEAD = -1; TAIL = -1;}
```

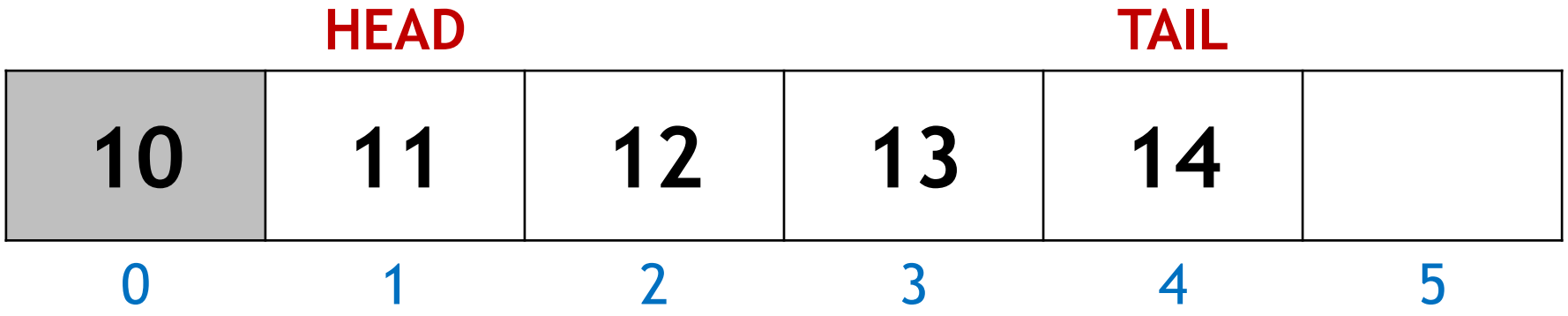
QueueArray: Dequeue



2. Else just update HEAD

```
else { HEAD = HEAD + 1;}
```

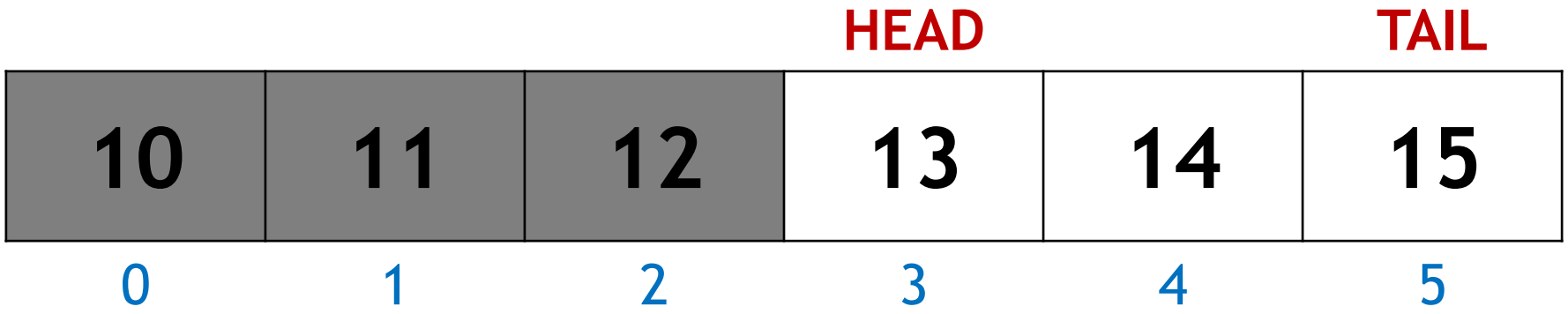
QueueArray: Dequeue



2. Else just update HEAD

```
else { HEAD = HEAD + 1;}
```

Queue: Circular Array Implementation



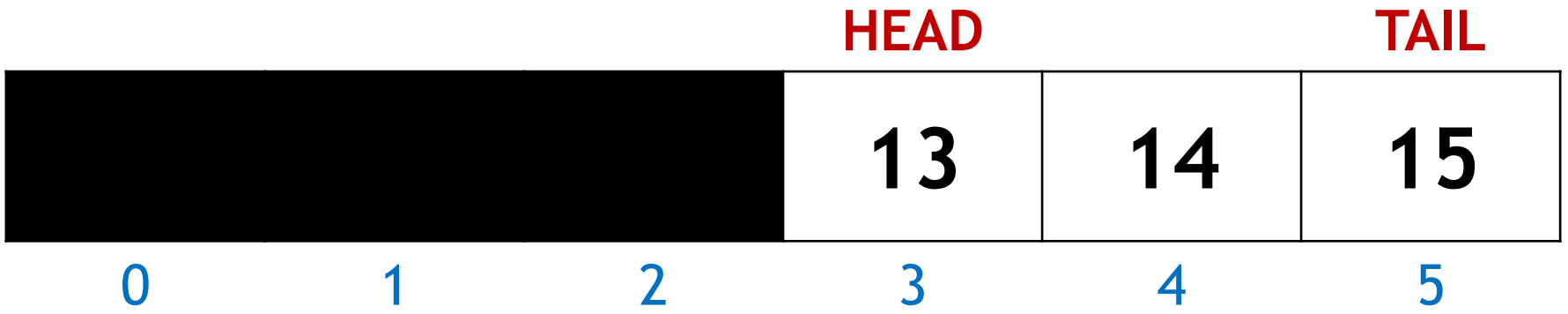
QueueArray seems to waste a lot of space.

Q. Why was this not an issue in StackArray?

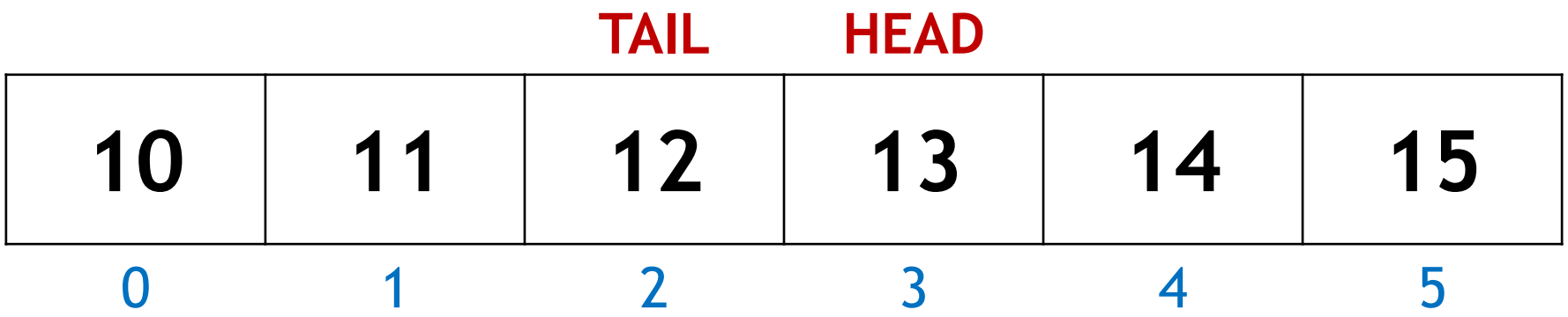
Queue: Circular Array Implementation

Utilize space that is wasted on elements outside the queue

Q. What is **`(TAIL + 1) % array_size`*?***



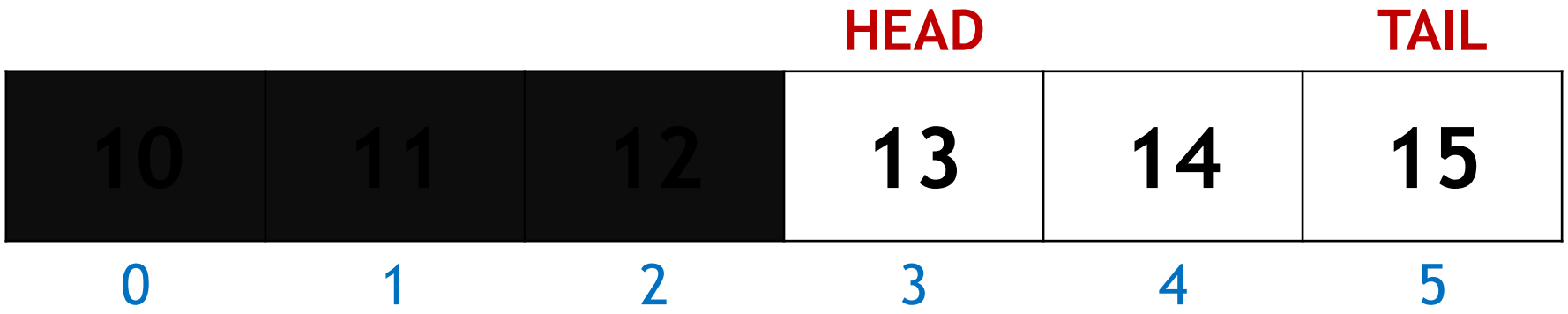
QueueCircular: Enqueue



0. Check if there is enough space [Edge Case]

```
if (HEAD == (TAIL + 1) % array_size) { OVERFLOW; }
```

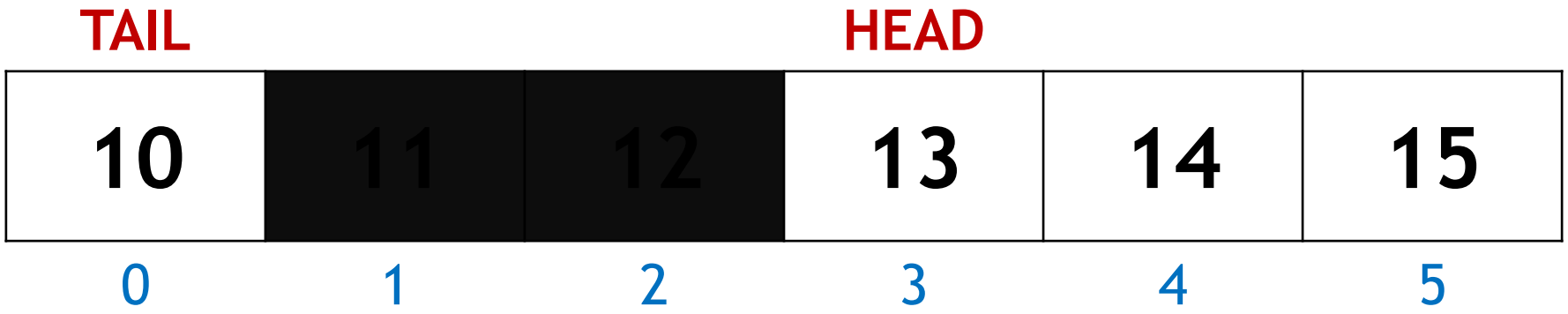
QueueCircular: Enqueue



1. Else, Update TAIL with mod increment

```
TAIL = (TAIL + 1) % array_size;
```

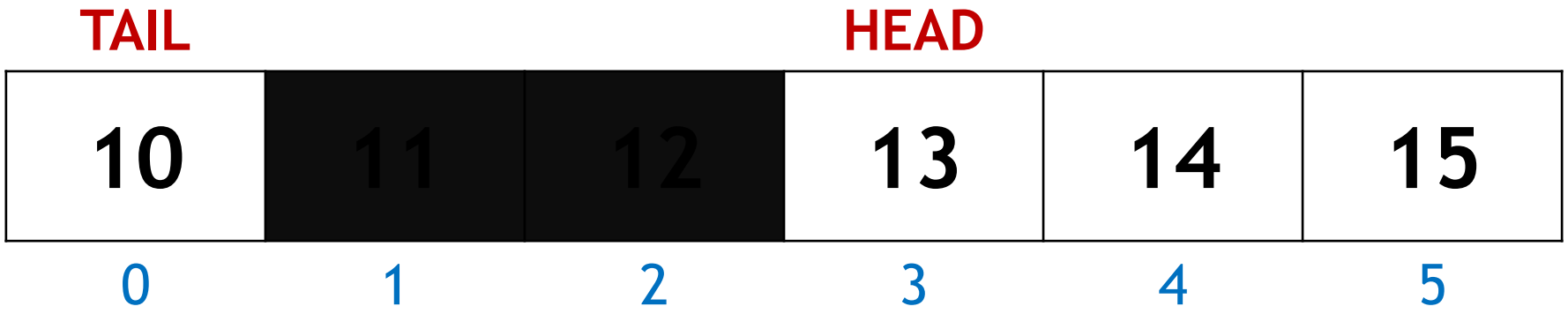
QueueCircular: Enqueue



1. Else, Update TAIL with mod increment

```
TAIL = (TAIL + 1) % array_size;
```

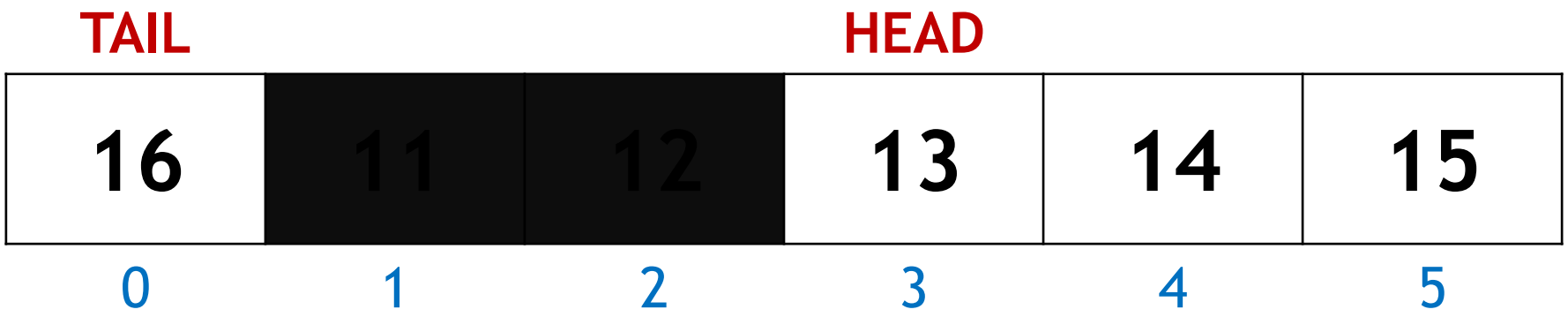
QueueCircular: Enqueue



2. Insert element at TAIL

```
Queue[TAIL] = 16;
```

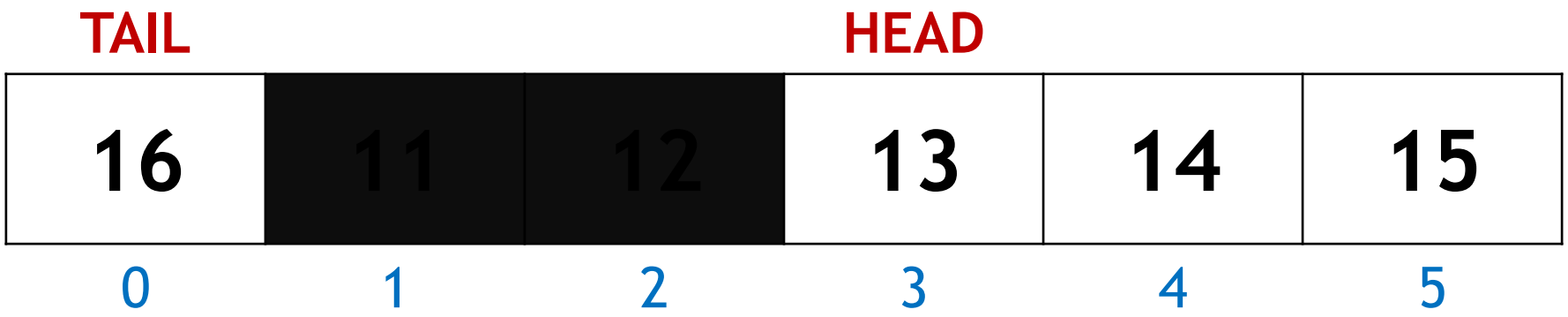
QueueCircular: Enqueue



2. Insert element at TAIL

```
Queue[TAIL] = 16;
```

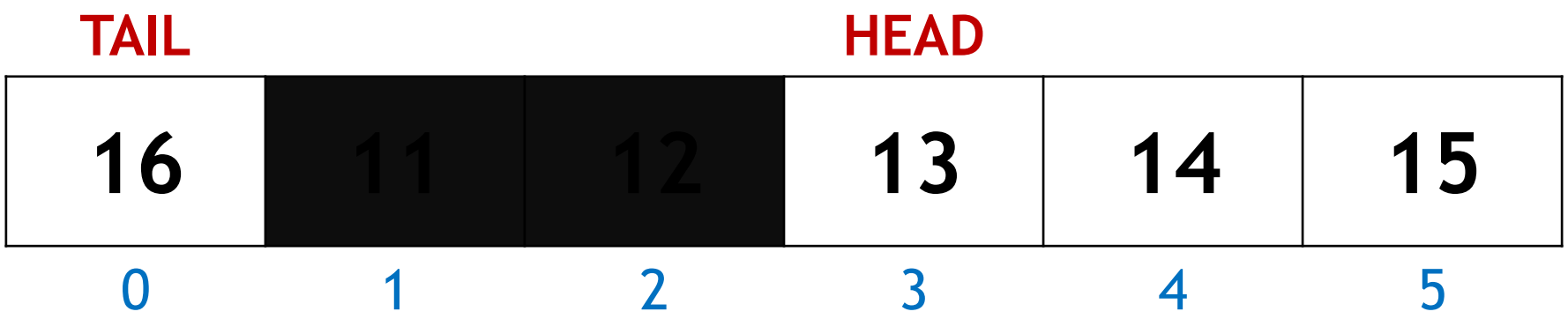
QueueCircular: Enqueue

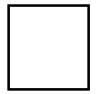


3. Check if HEAD needs to be updated [Edge Case]

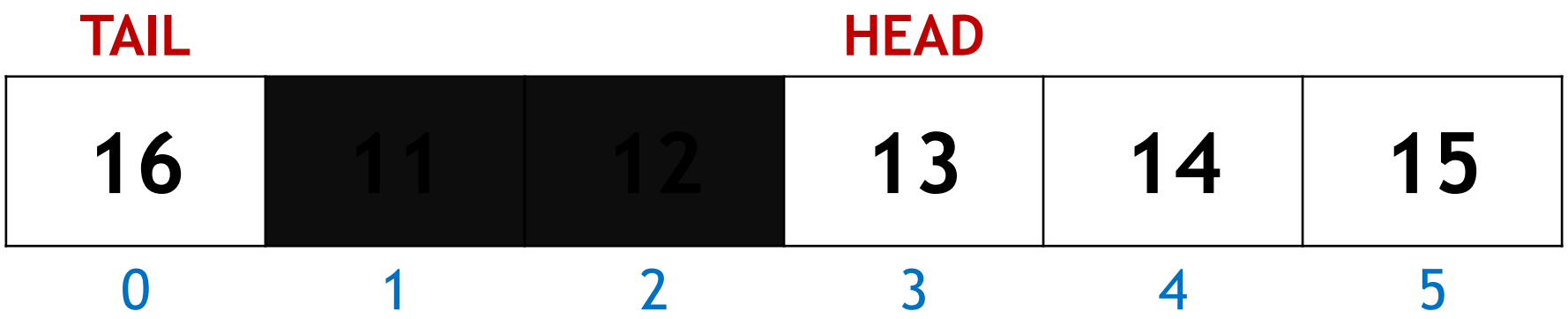
```
if (HEAD == -1) { HEAD = HEAD + 1; }
```

QueueCircular: Enqueue





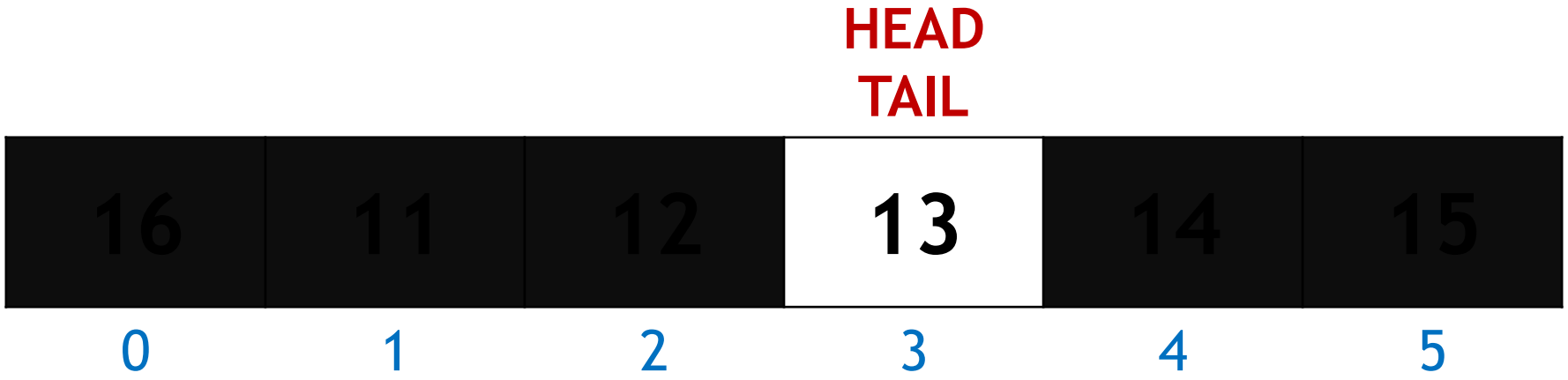
QueueCircular: Dequeue



0. Check HEAD is a valid index [Edge Case]

```
if (HEAD < 0) { return UNDERFLOW;}
```

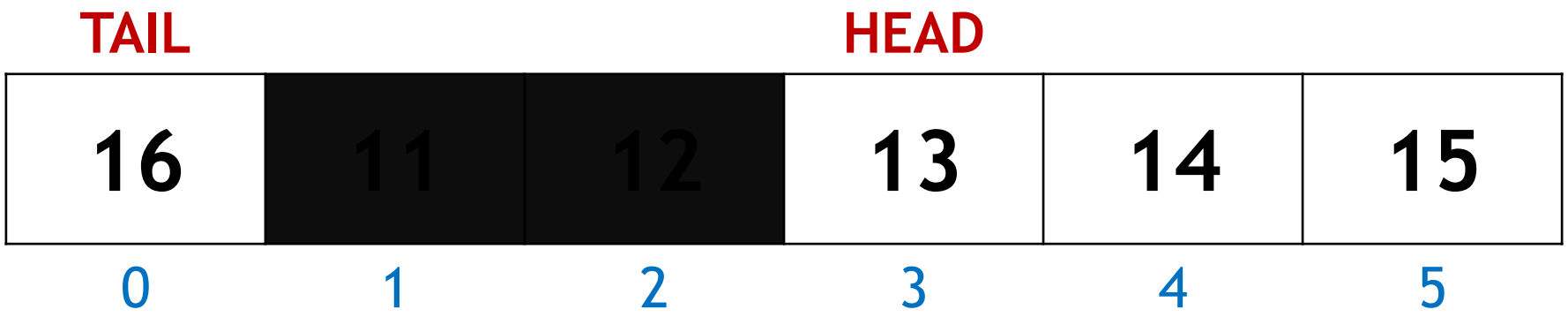
QueueCircular: Dequeue



1. Check if HEAD == TAIL [Edge Case]

```
if (HEAD == TAIL) { HEAD = -1; TAIL = -1; }
```

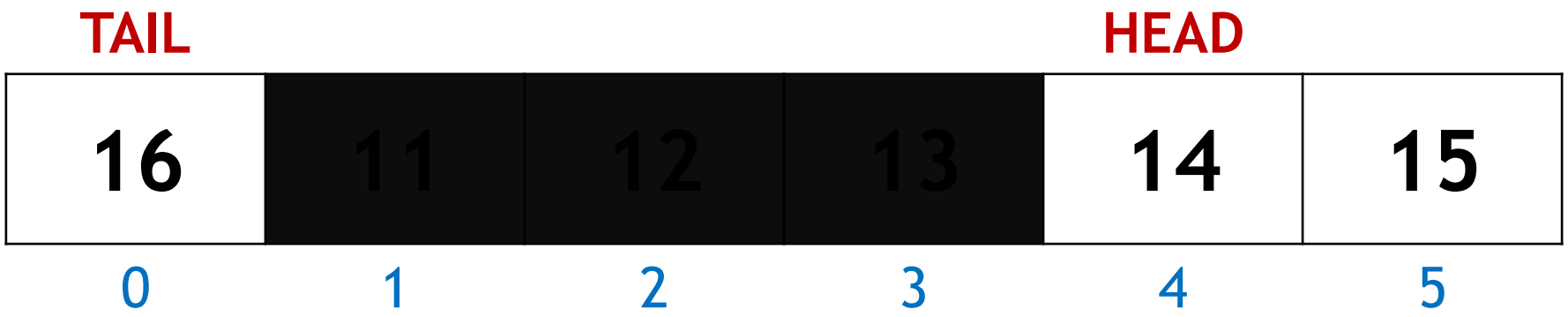
QueueCircular: Dequeue



2. Else, Update HEAD with mod increment

```
else { HEAD = (HEAD + 1) % array_size; }
```

QueueCircular: Dequeue



2. Else, Update HEAD with mod increment

```
else { HEAD = (HEAD + 1) % array_size; }
```

QueueCircular: Dequeue

Queue: Linked List Implementation

Pro: Dynamically grow and shrink the Queue

Con: Extra memory usage due to pointers

Queue	Queue (Linked List)
head, tail	head, tail
enqueue	Insert at tail
dequeue	Delete at head
peek	Return head's data

Mid-term FCQ

Kindly fill the Midterm FCQ.

Check your **@colorado.edu** inbox for link.
Email from ***Rajshree Shrestha***.

Deadline: Feb 19, 2020

Exercise

Stack, Queue

Exercise: Silver

Q: ***Complete Enqueue, Dequeue operations in QueueLL.cpp***

```
void QueueLL::dequeue()
void QueueLL::enqueue(int key)
```

How?

Enqueue: Insertion at end in a LL

Dequeue: Deletion at head in a LL

Exercise: Gold

Q: ***Solve the Balanced Brackets Problem in Driver.cpp***

```
bool isValid(string input);
```

How?

Utilize a stack's LIFO principle

Push open brackets to stack, and figure out when to pop