Data Structures and Object Oriented Programming

Lecture 11

Dr. Naveed Anwar Bhatti

Webpage: naveedanwarbhatti.github.io

Object-Oriented Programming in C++

Stack and Queue

Stack

Stack

- Stack stores arbitrary objects
- Insertions and deletions follow **last-in first-out (LIFO)** scheme

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

- Auxiliary stack operations:
  - **top():** returns the last inserted element without removing it
  - **size():** returns the number of elements stored
  - **isEmpty():** returns a Boolean value indicating whether no elements are stored

Stack (Example)

Operation	Output	Stack
push(8)	**-**	(8)
push(3)	**-**	(3,8)
pop()	**3**	(8)
push(2)	**-**	(2,8)
push(5)	**-**	(5,2,8)
top()	**5**	(5,2,8)
pop()	**5**	(2,8)
pop()	**2**	(8)
pop()	**8**	()
push(9)	**-**	(9)
push(1)	**-**	(1,9)

Applications of Stacks

- **Direct applications**
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another, and so on.
- **Indirect applications**
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array-based Stack

A simple way of implementing the Stack uses an array

- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm push(o)
    if t = S.length - 1 then
        throw FullStackException
    else
        t ← t + 1
        S[t] ← o
```

```
Algorithm pop()
    if isEmpty() then
        throw EmptyStackException
    else
        t ← t - 1
        return S[t + 1]
```

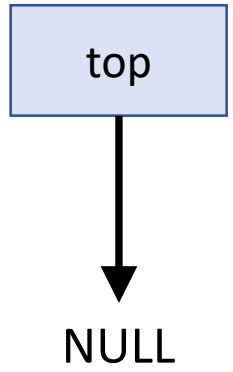
Array-based Stack

Limitations

- The maximum size of the stack must be defined a priori and cannot be changed
- Trying to push a new element into a full stack causes an implementation-specific exception

Stack with a Singly Linked List

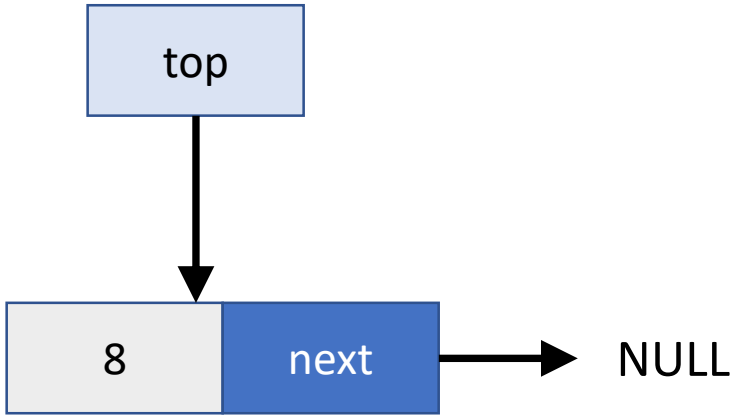
- We can implement a stack with a singly linked list



Stack with a Singly Linked List

- We can implement a stack with a singly linked list

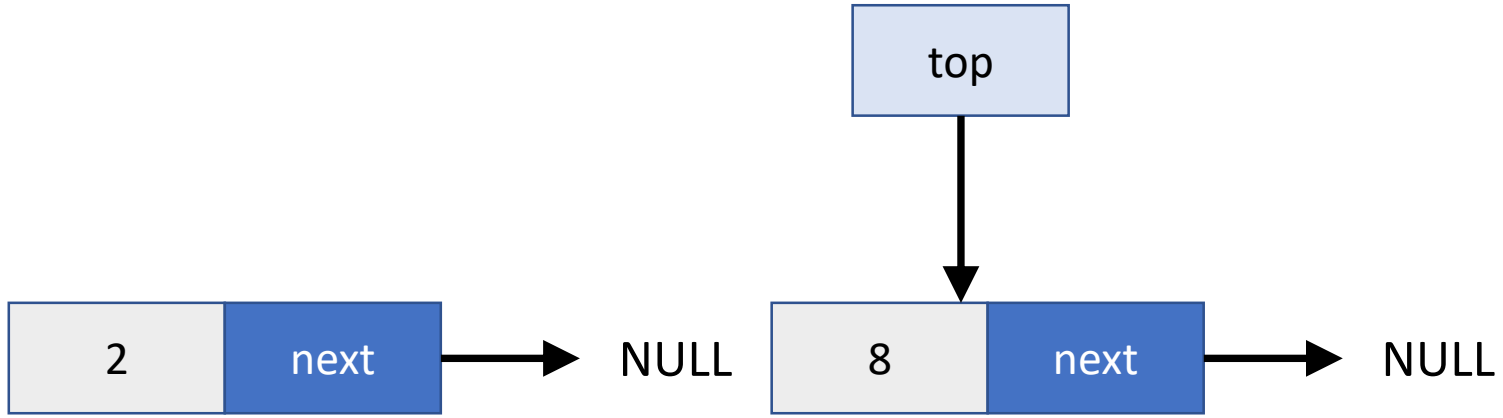
PUSH (8)



Stack with a Singly Linked List

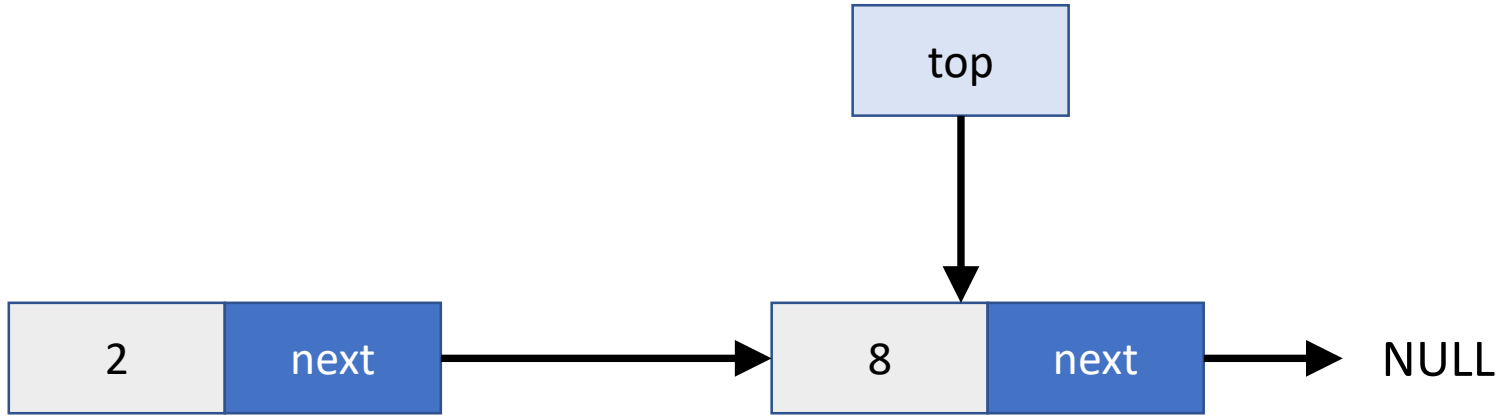
- We can implement a stack with a singly linked list

PUSH (2)



Stack with a Singly Linked List

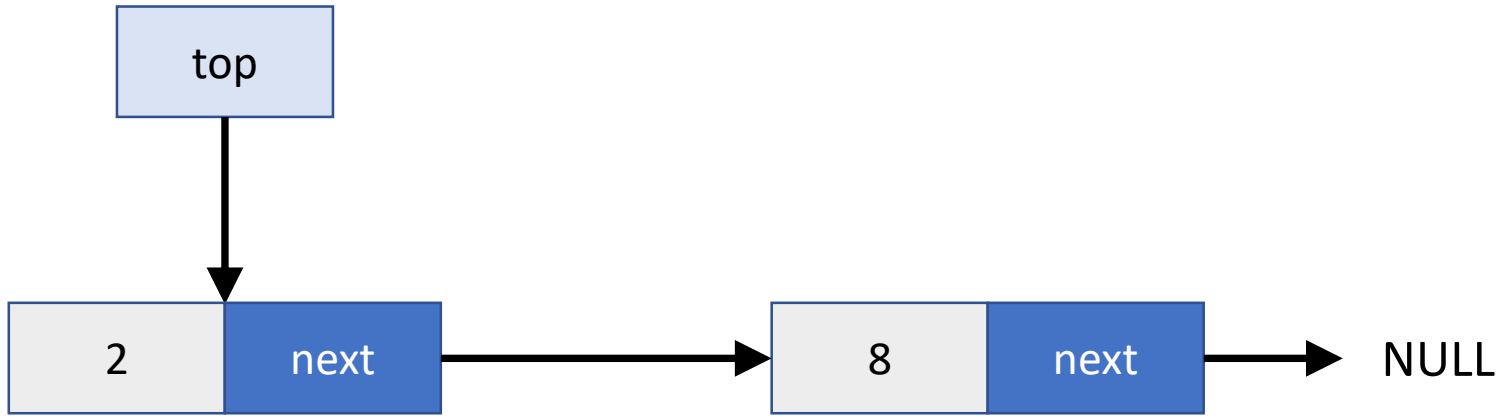
- We can implement a stack with a singly linked list



temp->next=top

Stack with a Singly Linked List

- We can implement a stack with a singly linked list

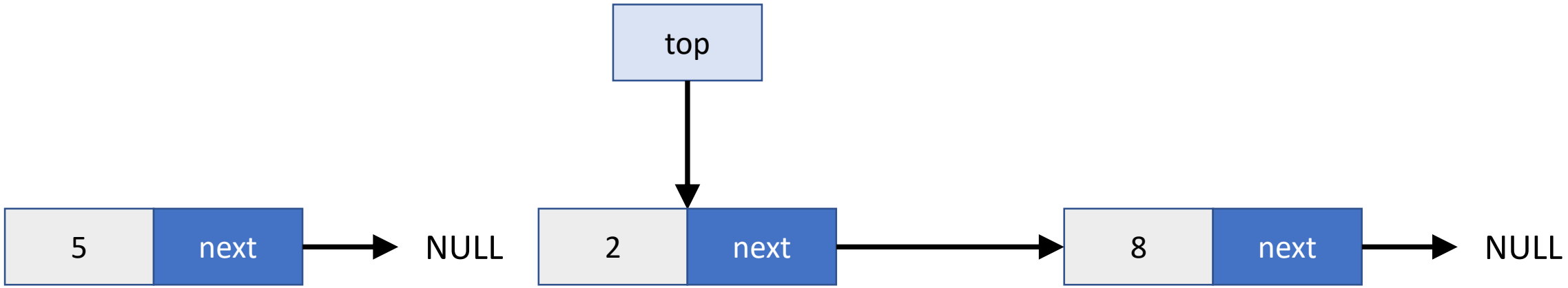


top=temp;

Stack with a Singly Linked List

- We can implement a stack with a singly linked list

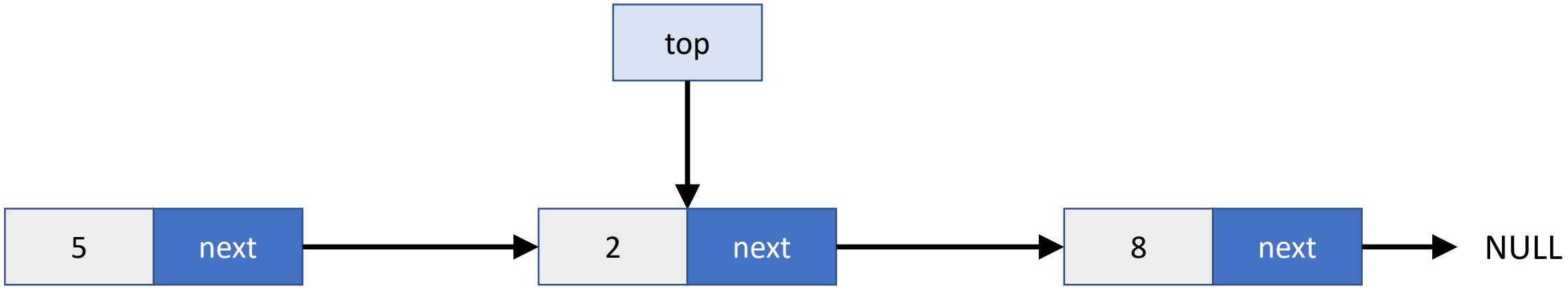
PUSH (5)



top=temp;

Stack with a Singly Linked List

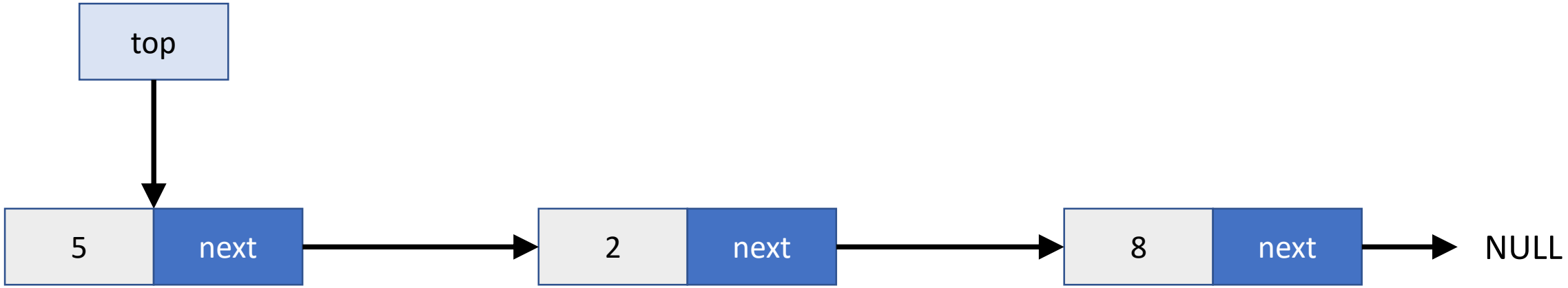
- We can implement a stack with a singly linked list



temp->next=top

Stack with a Singly Linked List

- We can implement a stack with a singly linked list

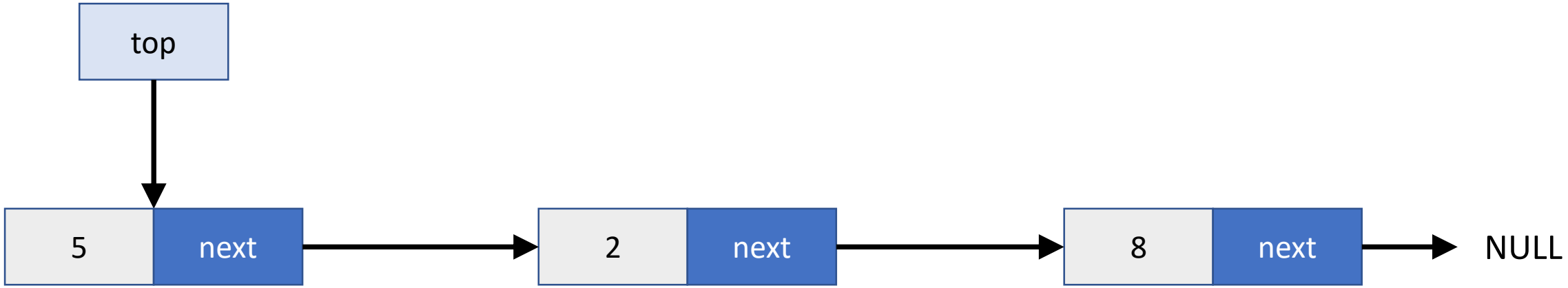


top=temp;

Stack with a Singly Linked List

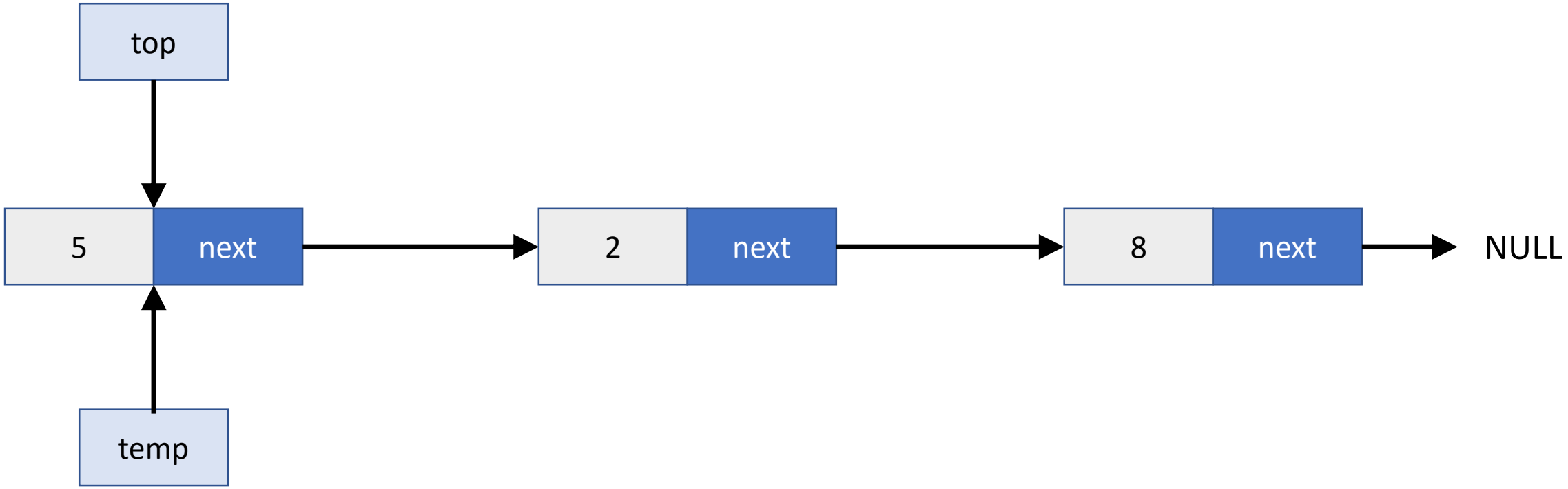
- We can implement a stack with a singly linked list

POP ()



Stack with a Singly Linked List

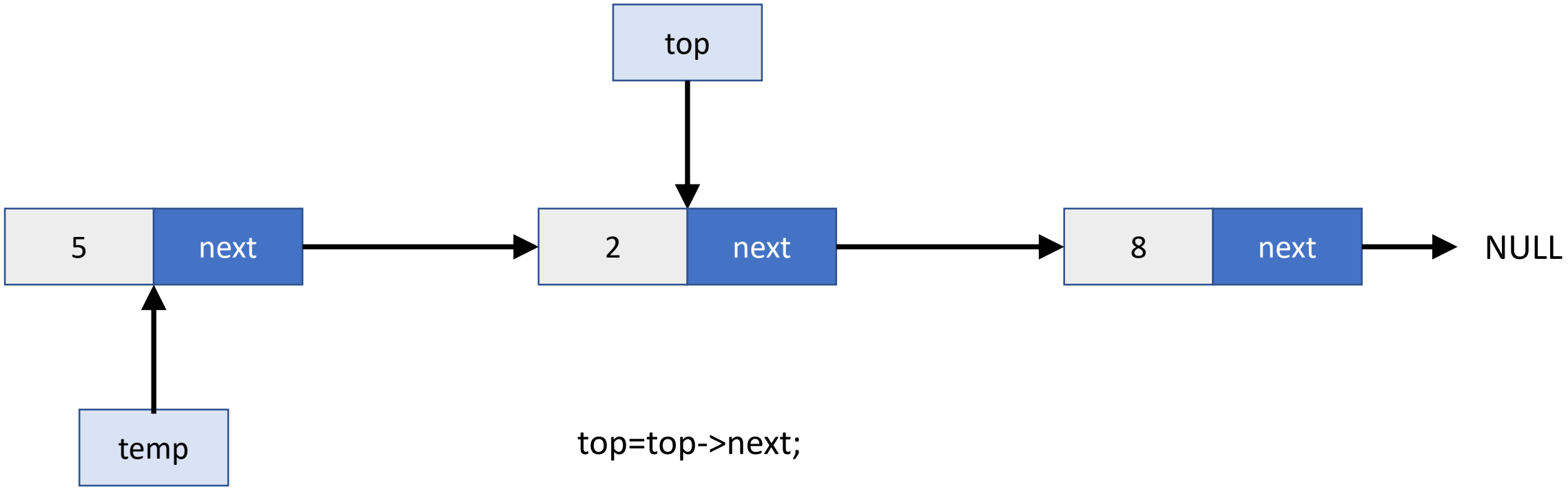
- We can implement a stack with a singly linked list



Node* temp=top;

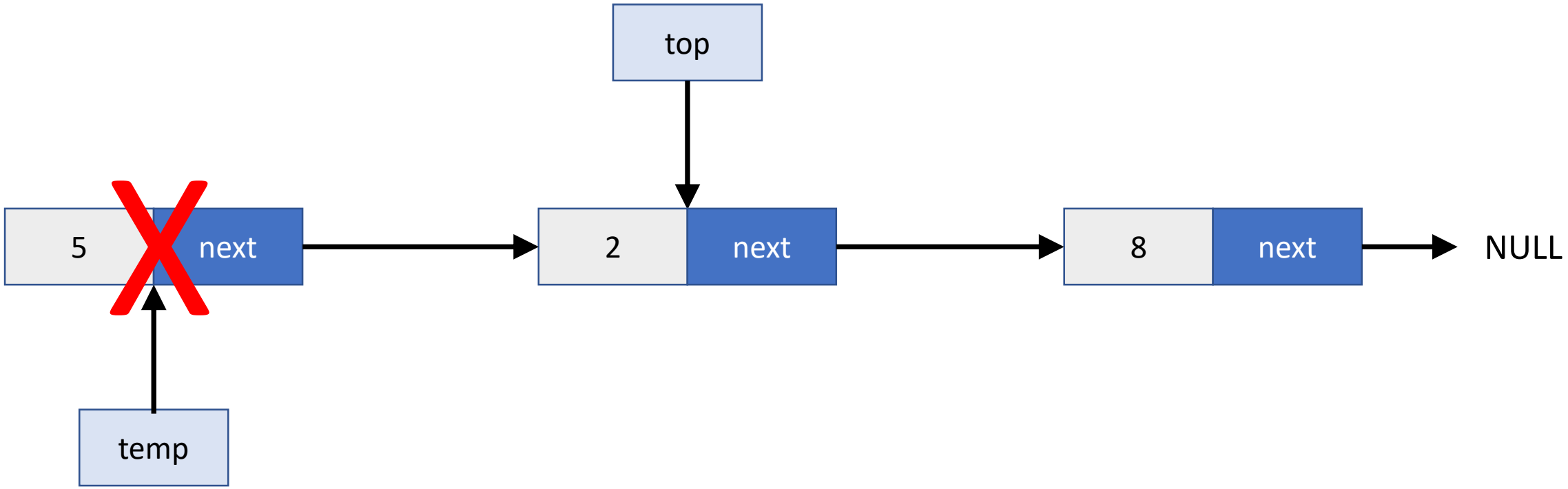
Stack with a Singly Linked List

- We can implement a stack with a singly linked list



Stack with a Singly Linked List

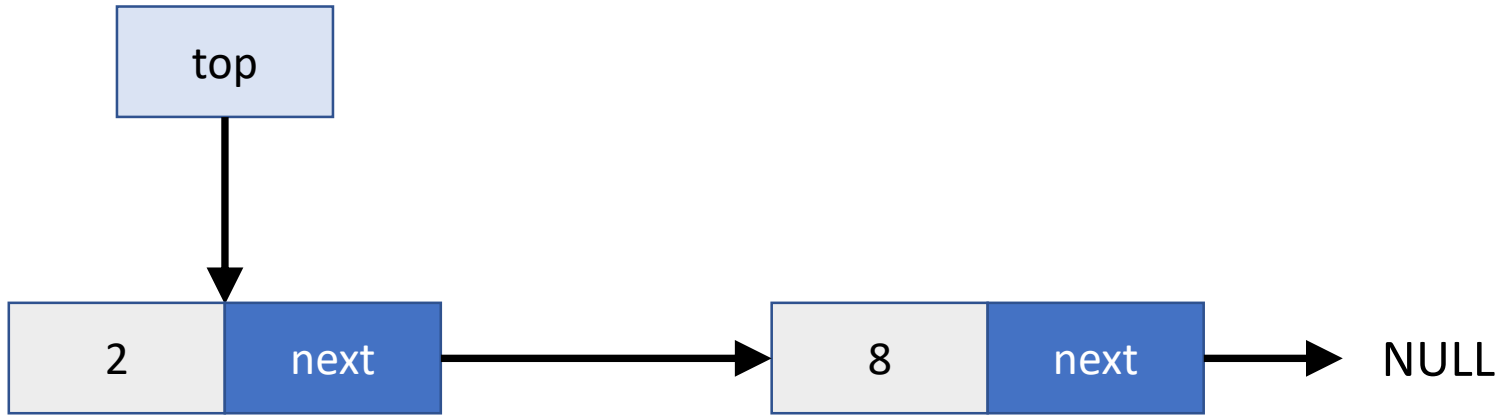
- We can implement a stack with a singly linked list



delete temp;

Stack with a Singly Linked List

- We can implement a stack with a singly linked list



Stack with a Singly Linked List – Example Code

```
#include <iostream> //header file
using namespace std; //standard namespace

struct Node {
    int data;
    Node* next = NULL;

};

class Stack {

    Node* top = NULL;

public:

    void push(int value);
    int pop();
    int peek();
    void printStack();
};
```

Stack with a Singly Linked List – Example Code

```
void Stack::push(int value)
{

    Node* temp = new Node;
    temp->data = value;

    if (top == NULL)
    {
        top = temp;
    }

    else
    {
        temp->next = top;
        top = temp;
    }

}
```

```
int Stack::pop()
{
    if (top == NULL)
    {
        return 0;
    }

    else
    {
        Node* temp = top;
        top = top->next;
        int data=temp->data;
        delete temp;
        return data;

    }

}
```

```
int Stack::peek()
{
    if (top == NULL)
    {
        exit(1);
    }

    else
    {
        return top->data;
    }

}
```

Queue

Queue

- The Queue stores arbitrary objects
- Insertions and deletions follow the **first-in first-out (FIFO)** scheme
- Insertions are at the **end (rear)** of the queue and removals are at the **top (front)** of the queue

Queue

- **Main queue operations:**
  - **enqueue(o)**: inserts element o at the end of the queue
  - **dequeue()**: removes and returns the element at the front of the queue

- **Auxiliary queue operations:**
  - **front()**: returns the element at the front without removing it
  - **size()**: returns the number of elements stored
  - **isEmpty()**: returns a Boolean value indicating whether no elements are stored

Queue (Example)

Operation	Output	Queue
enqueue(5)	**-**	(5)
enqueue(3)	**-**	(3,5)
dequeue()	**5**	(3)
enqueue(7)	**-**	(7,3)
dequeue()	**3**	(7)
front()	**7**	(7)
dequeue()	**7**	()

Applications of Queues

Direct applications

- Waiting lists
- Access to shared resources (e.g., printer)
- Multiprogramming

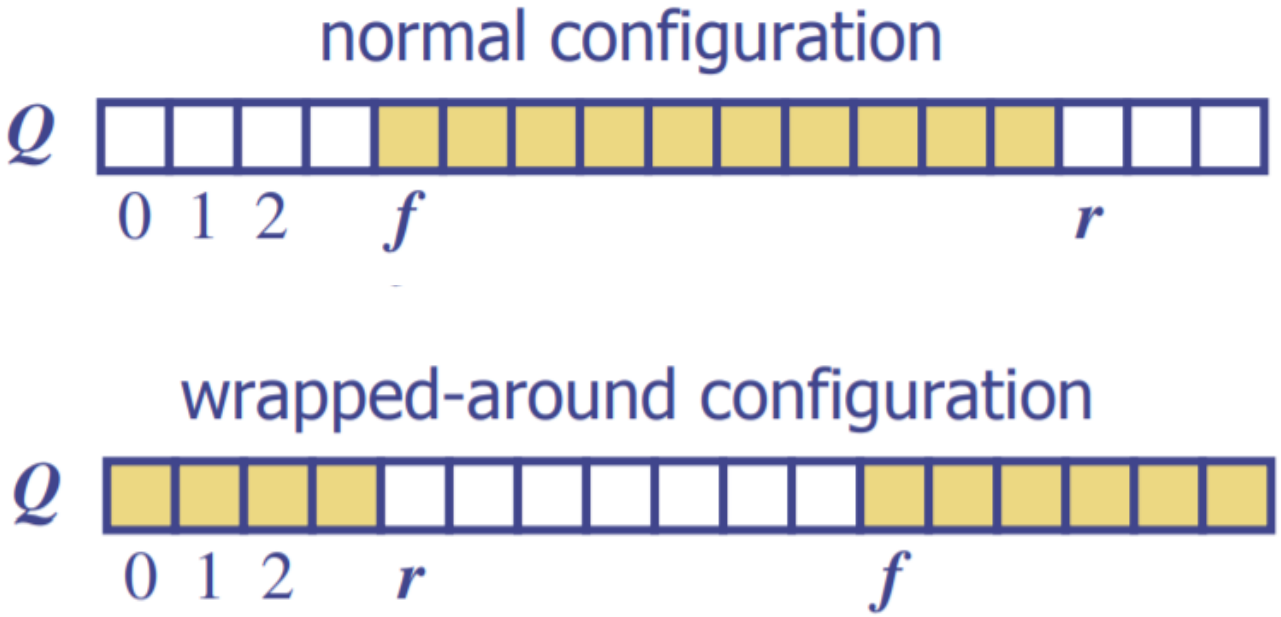
Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Array-based Queue

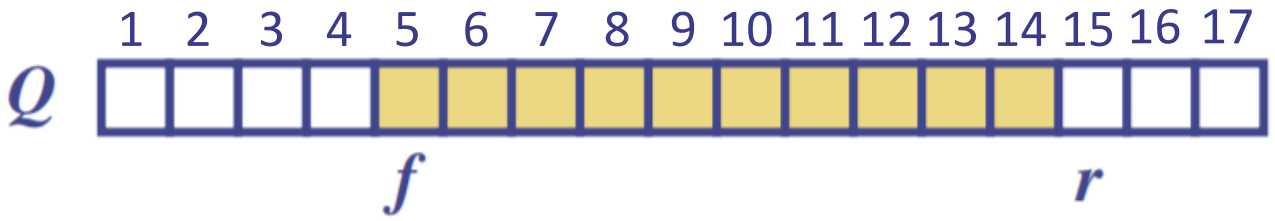
Use an array of size N in a circular fashion

- Two variables keep track of the front and rear
- **f index** of the front element
- **r index** immediately past the rear element
- Array location **r** is kept empty



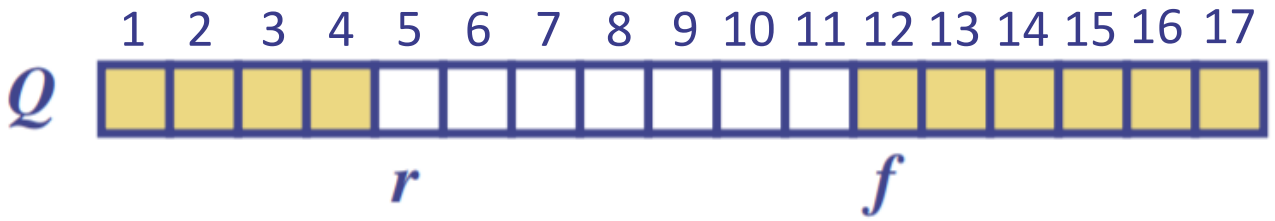
Array-based Queue – Algorithm

We use the ***modulo*** operator (remainder of division) for **size()**, **enqueue()** and **dequeue()** functions



$r - f$

5 – 15 = 10

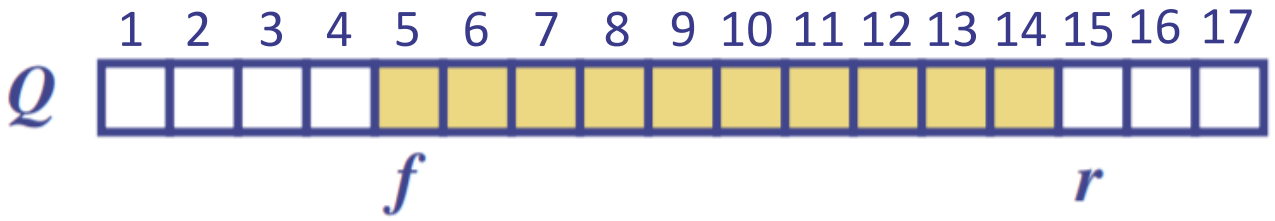


$r - f$

5 – 12 = -7

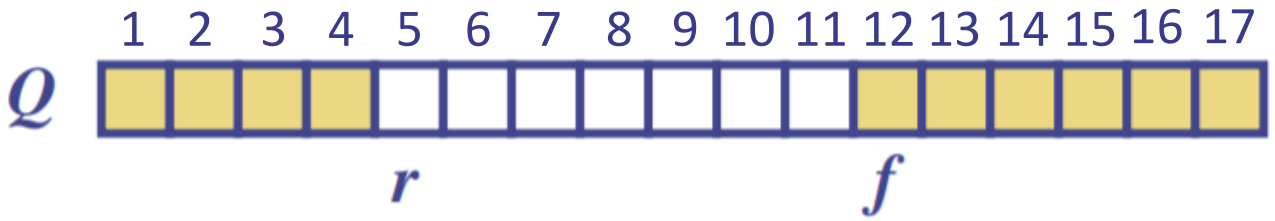
Array-based Queue – Algorithm

We use the ***modulo*** operator (remainder of division) for **size()**, **enqueue()** and **dequeue()** functions



$$N + r - f$$

17 + 15 – 5 = 27

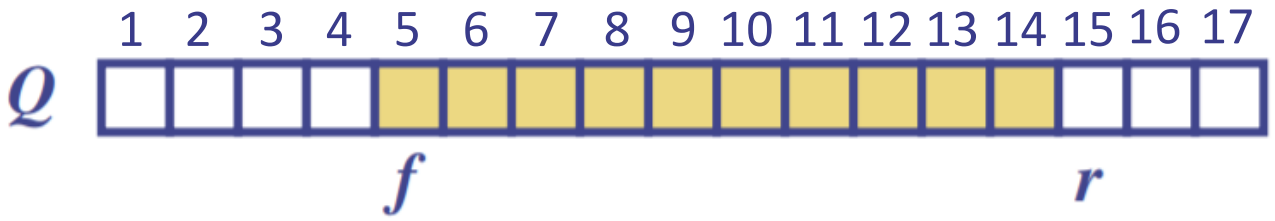


$$N + r - f$$

17 + 5 – 12 = 10

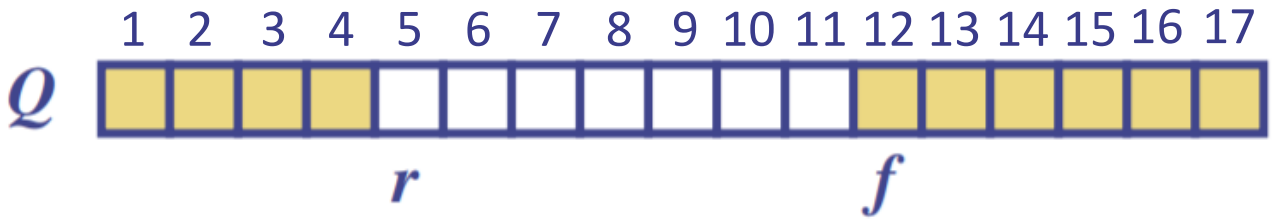
Array-based Queue – Algorithm

We use the ***modulo*** operator (remainder of division) for **size()**, **enqueue()** and **dequeue()** functions



$(N + r - f) \bmod N$

17 + 5 – 15 = 27 % 17 =10



$(N + r - f) \bmod N$

17 + 5 – 12 = 10 % 17 = 10

Algorithm *size*()
  return $(N - f + r) \bmod N$

Algorithm *isEmpty*()
  return $(f = r)$

Array-based Queue – Algorithm

```
Algorithm enqueue(o)
    if size() = N - 1 then
        throw FullQueueException
    else
        Q[r] ← o
        r ← (r + 1) mod N
```

Array-based Queue – Algorithm

```
Algorithm dequeue()
    if isEmpty() then
        throw EmptyQueueException
    else
        o ← Q[f]
        f ← (f + 1) mod N
        return o
```

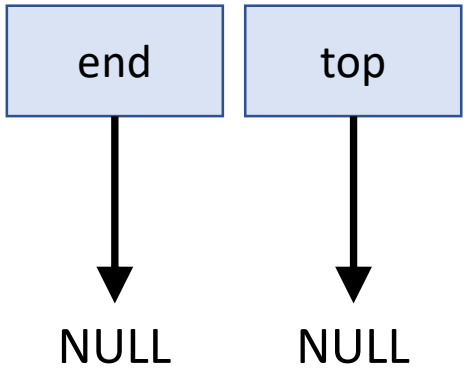
Array-based Queue – Limitations

- **The maximum size of the queue must be defined a priori , and cannot be changed**
- **Trying to push a new element into a full queue causes an implementation-specific exception**

Queue with a Singly Linked List

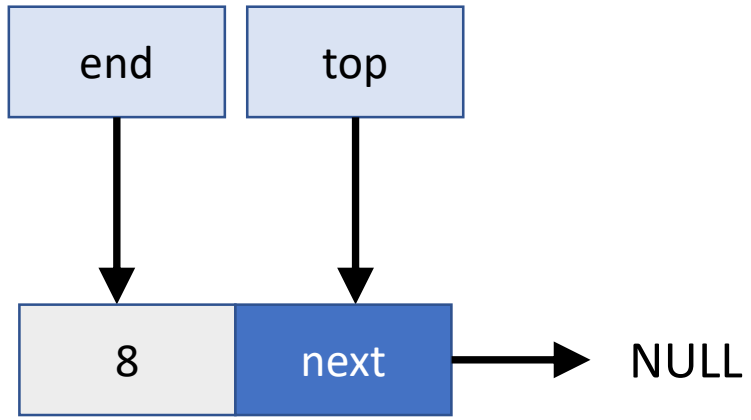
- **We can implement a queue with a singly linked list**
- **We do not have the size-limitation of the array based implementation, i.e., the queue is NEVER full**

Queue with a Singly Linked List



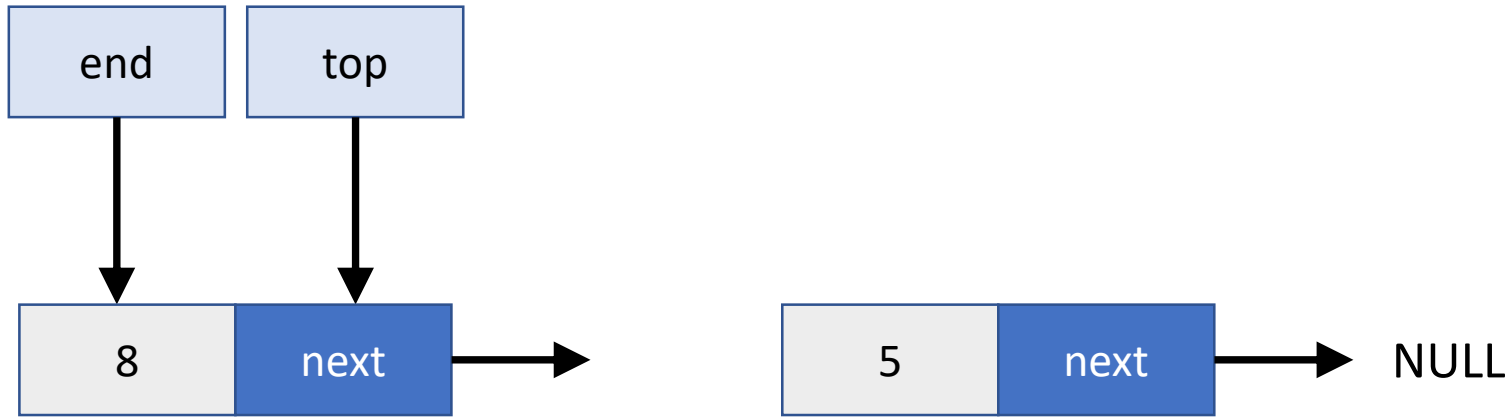
Queue with a Singly Linked List

enqueue (8)



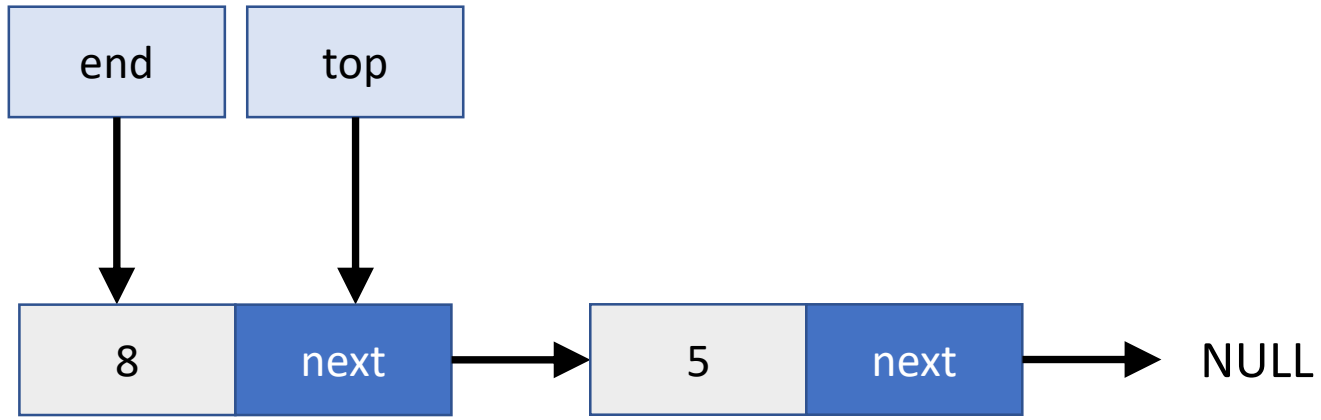
Queue with a Singly Linked List

enqueue (5)

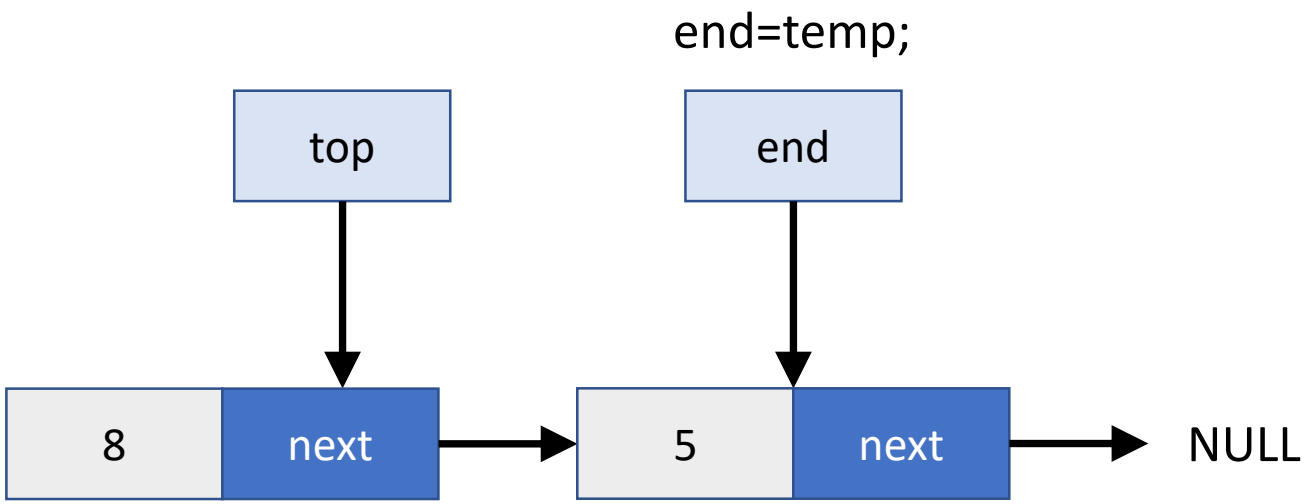


Queue with a Singly Linked List

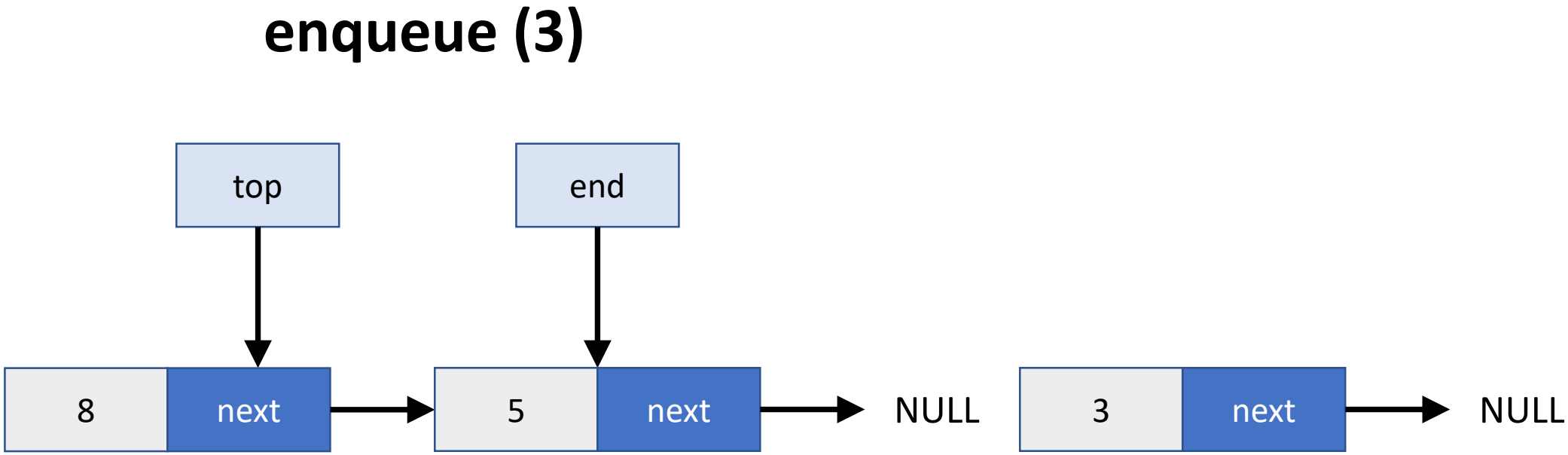
end->next=temp;



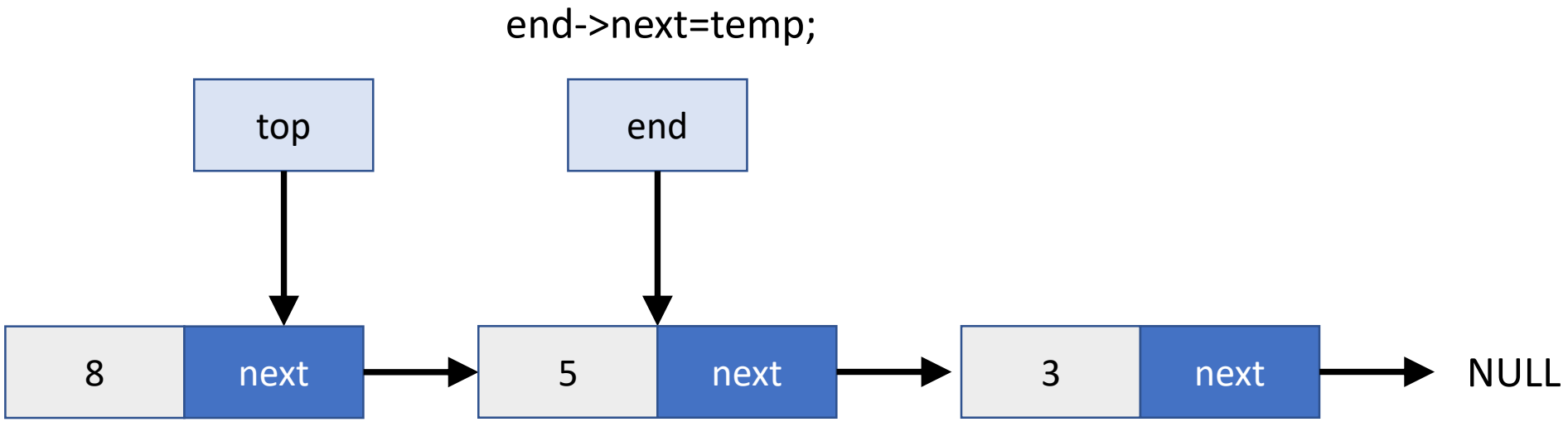
Queue with a Singly Linked List



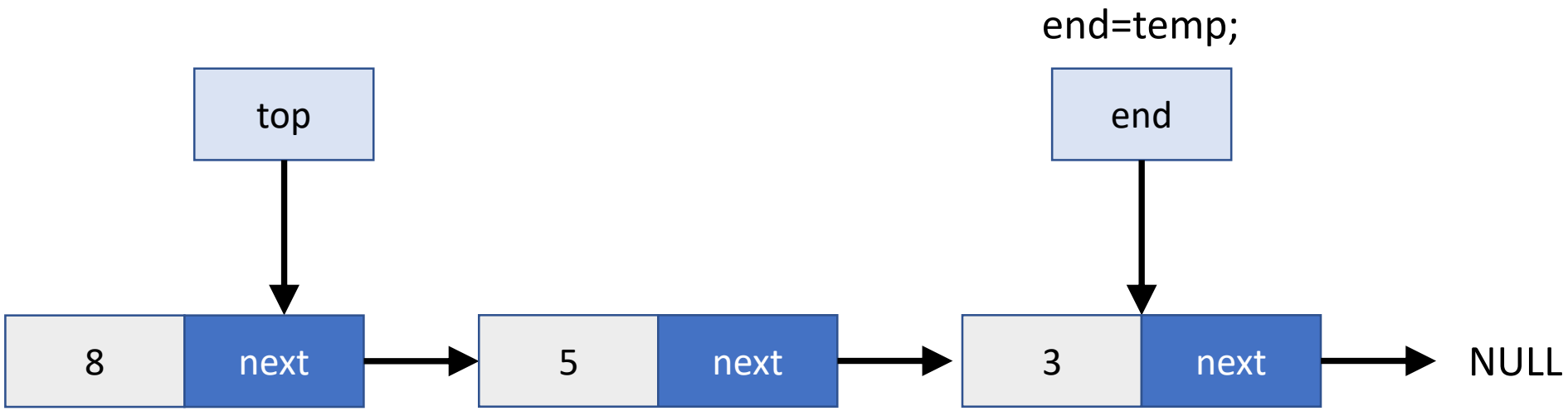
Queue with a Singly Linked List



Queue with a Singly Linked List

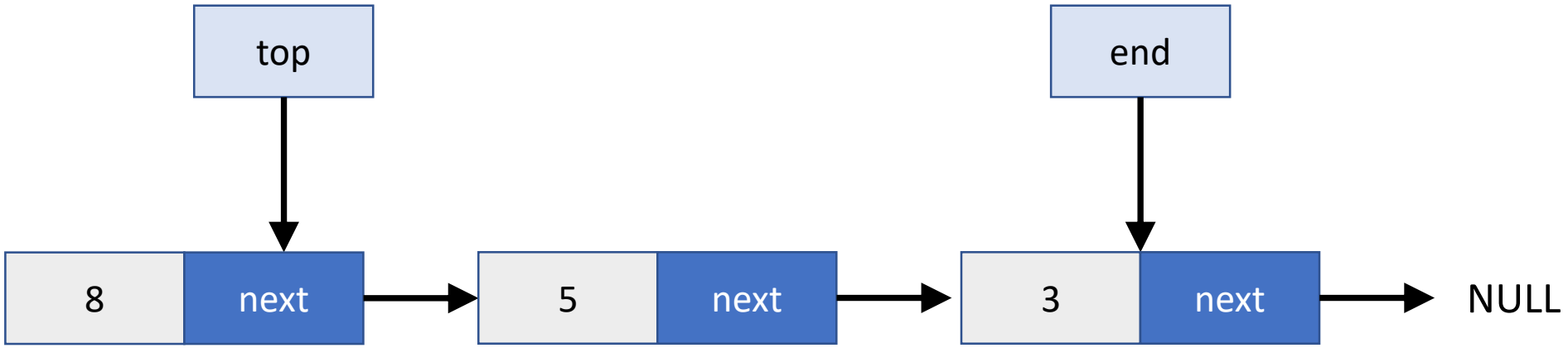


Queue with a Singly Linked List

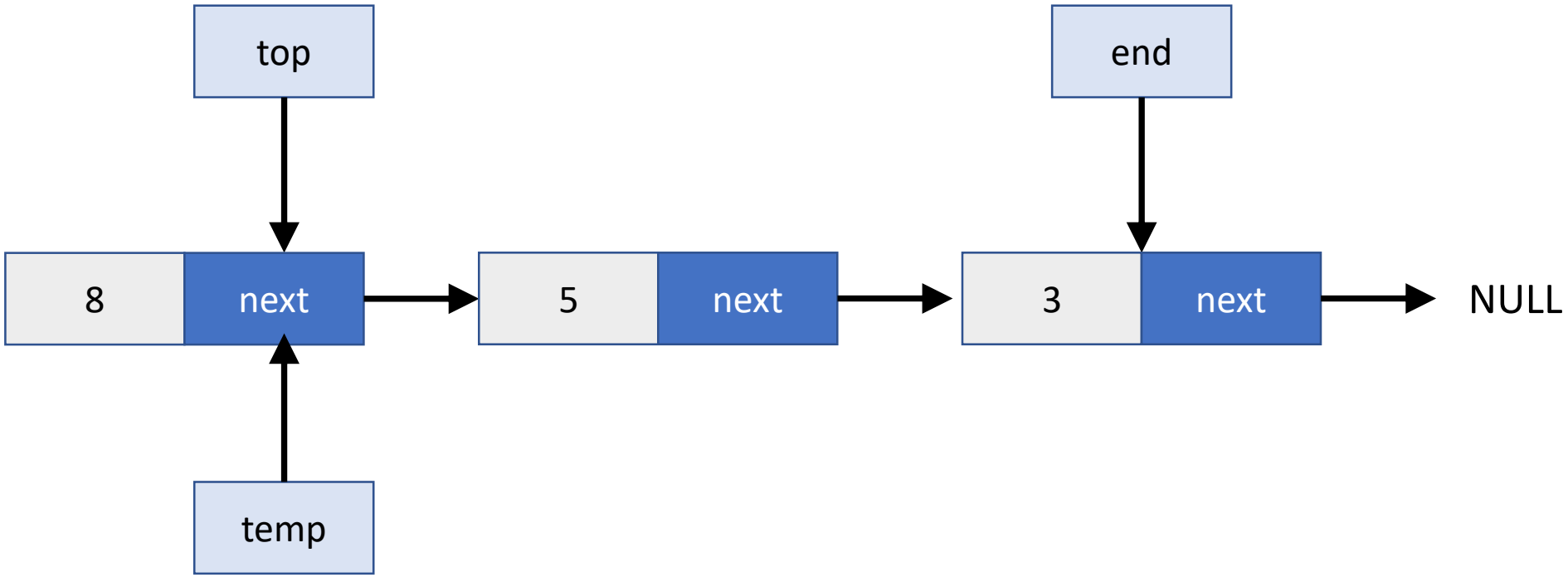


Queue with a Singly Linked List

dequeue ()

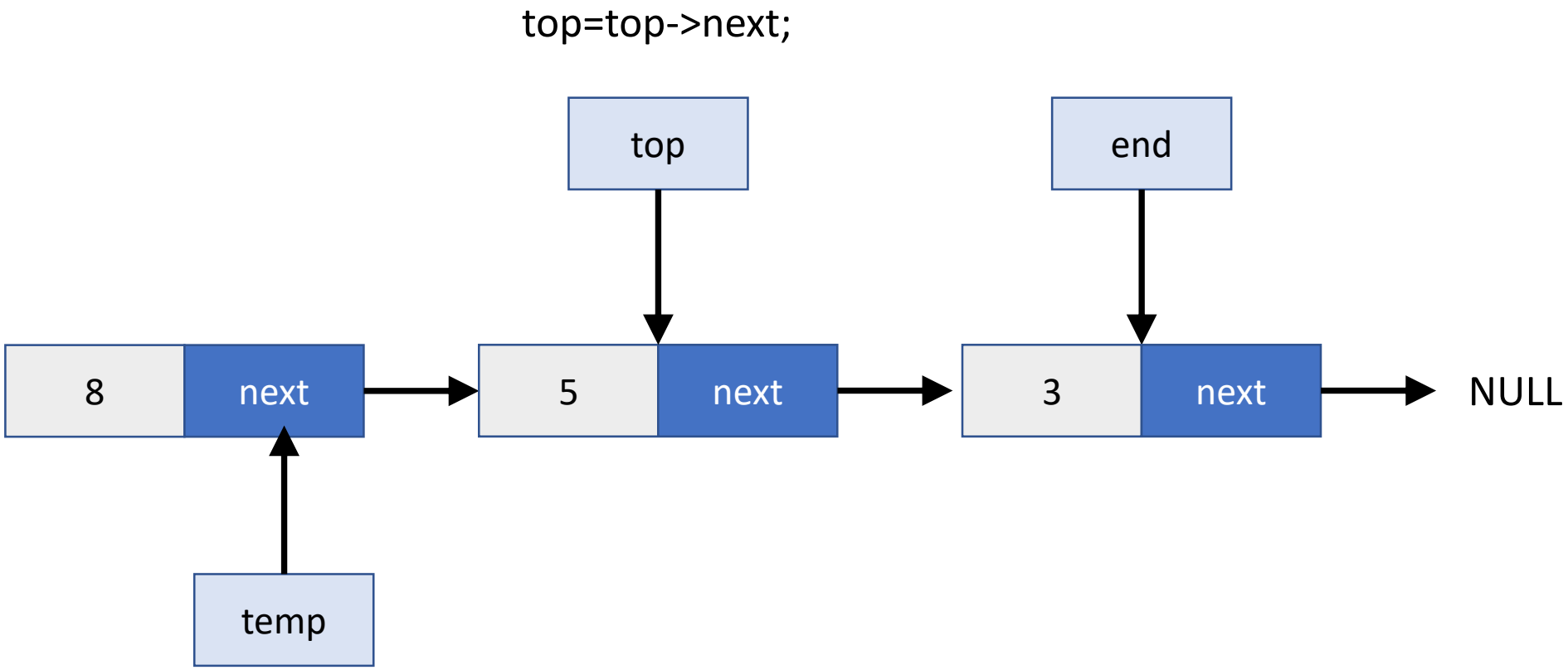


Queue with a Singly Linked List

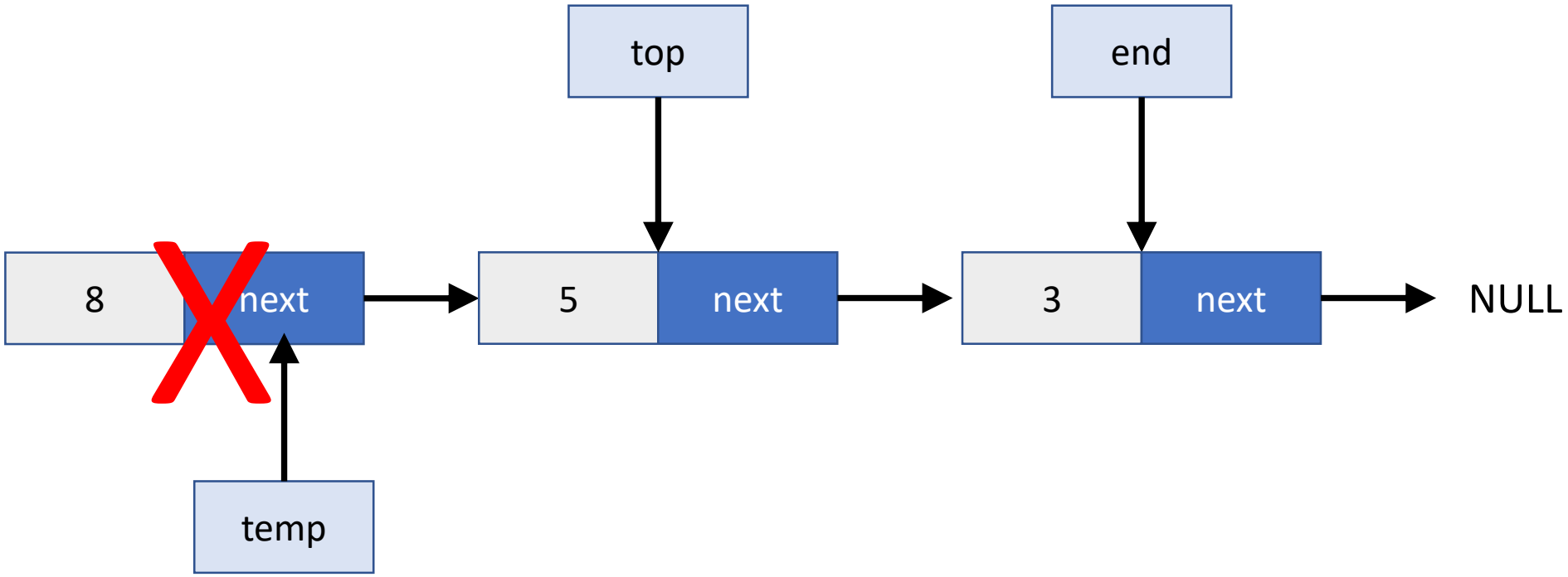


Node* temp=top;

Queue with a Singly Linked List

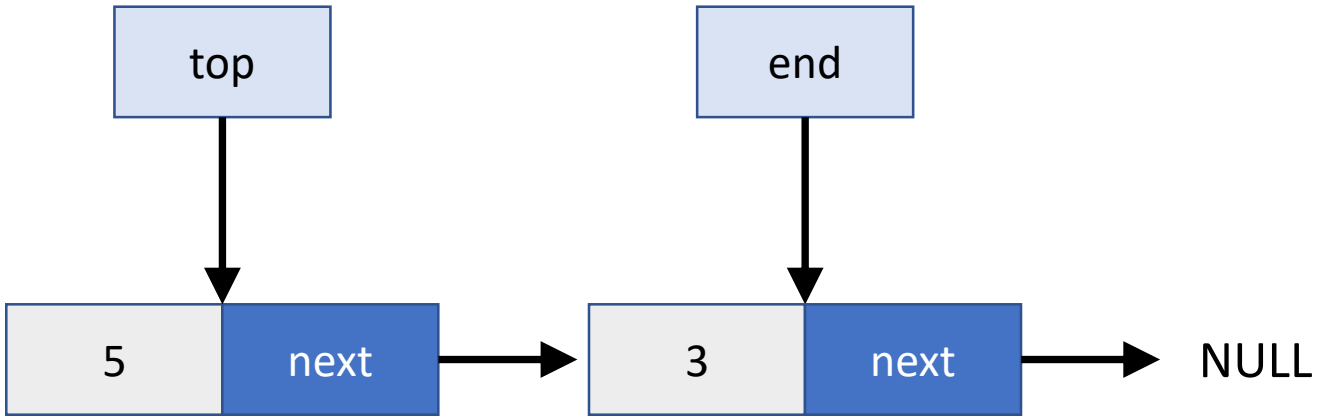


Queue with a Singly Linked List



delete temp;

Queue with a Singly Linked List



Queue with a Singly Linked List – Example Code

```
#include <iostream> //header file
using namespace std; //standard namespace

struct Node {
    int data;
    Node* next = NULL;

};

class queue {

    Node* top = NULL;
    Node* end = NULL;

public:

    void enqueue(int value);
    int dequeue();
    int front();
};
```

Queue with a Singly Linked List – Example Code

```
void queue::enqueue(int value)
{

    Node* temp = new Node;
    temp->data = value;

    if (end == NULL)
    {
        end = top = temp;
    }

    else
    {
        end->next = temp;
        end = temp;
    }

}
```

```
int queue::dequeue()
{
    if (top == NULL)
    {
        return 0;
    }

    else
    {
        Node* temp = top;
        top = top->next;

        if (top == NULL)
            end = NULL;
        int data = temp->data;
        delete temp;
        return data;
    }

}
```

Thanks a lot

If you are taking a Nap, **wake up**.....Lecture Over