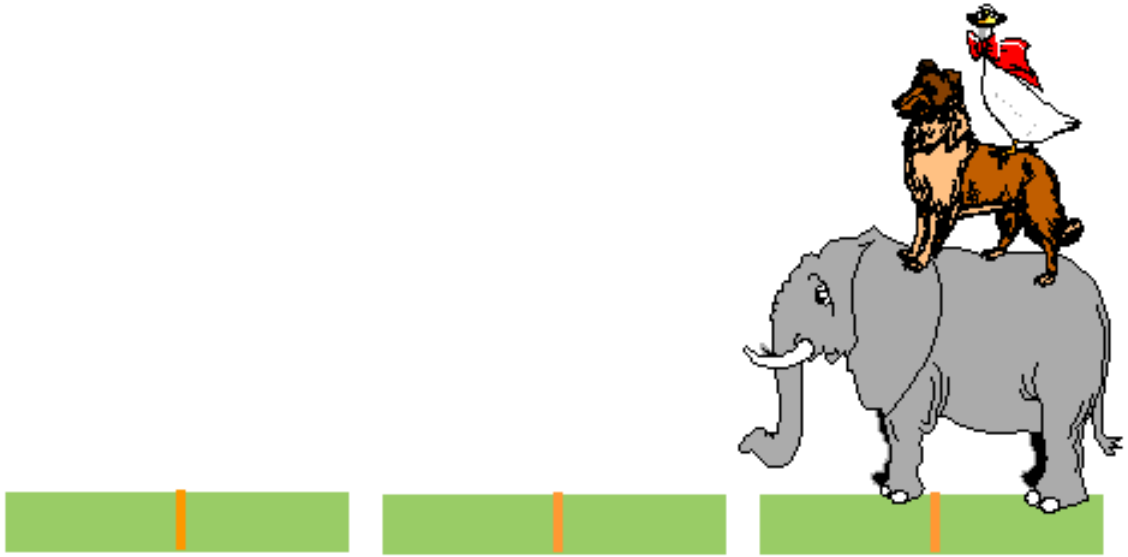
2023/2024(2)
EF234201 Data Structure

Lecture #4

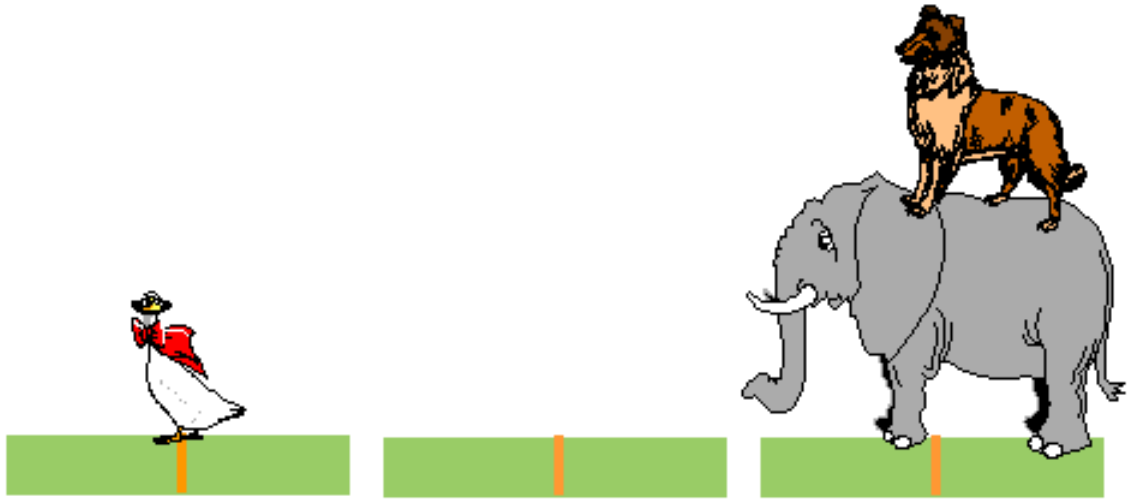
Array: Stack & Queue

Misbakhul Munir IRFAN SUBAKTI
司馬伊凡
Мисбакхул Мунир Ирфан Субакти

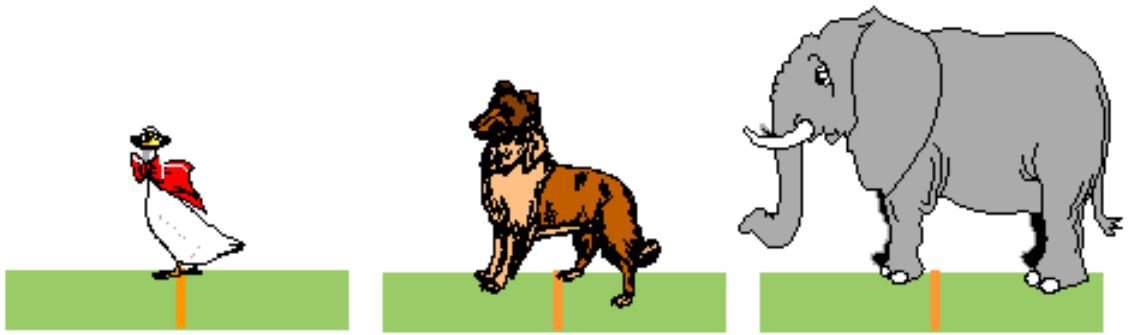
Towers of Hanoi



Towers of Hanoi (continued)



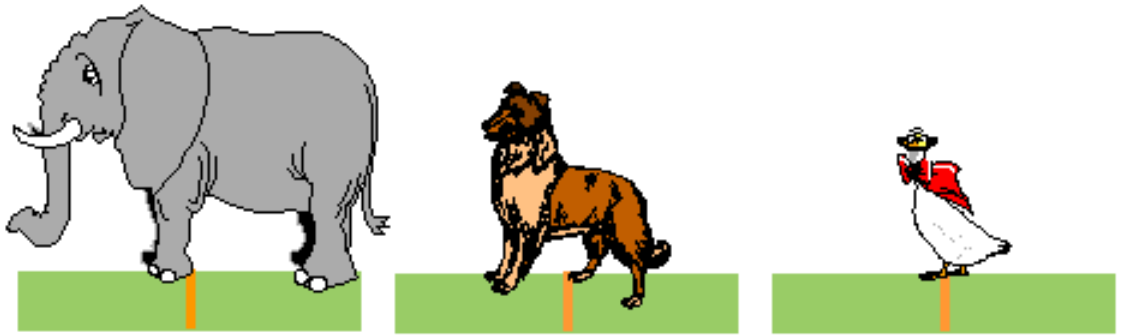
Towers of Hanoi (continued)



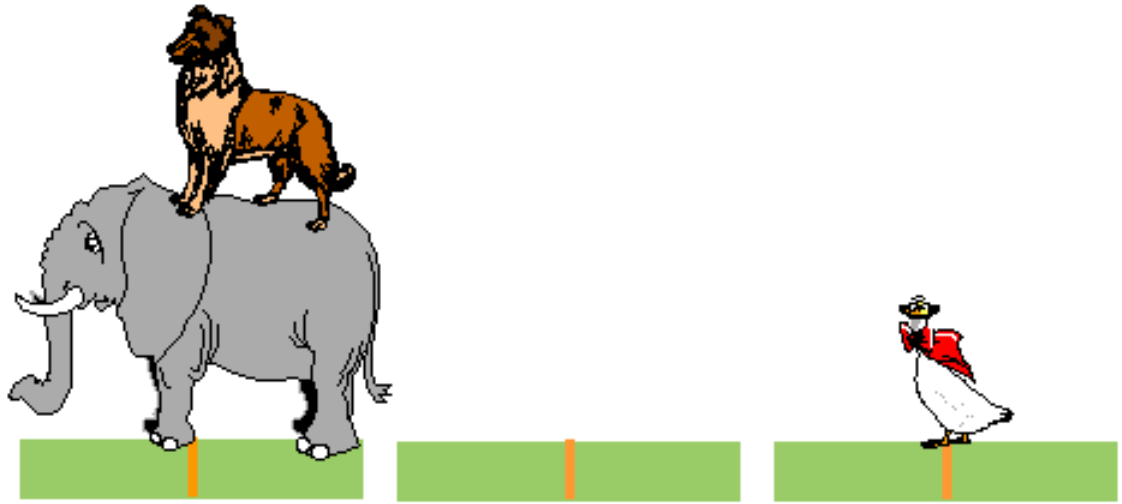
Towers of Hanoi (continued)

Towers of Hanoi (continued)

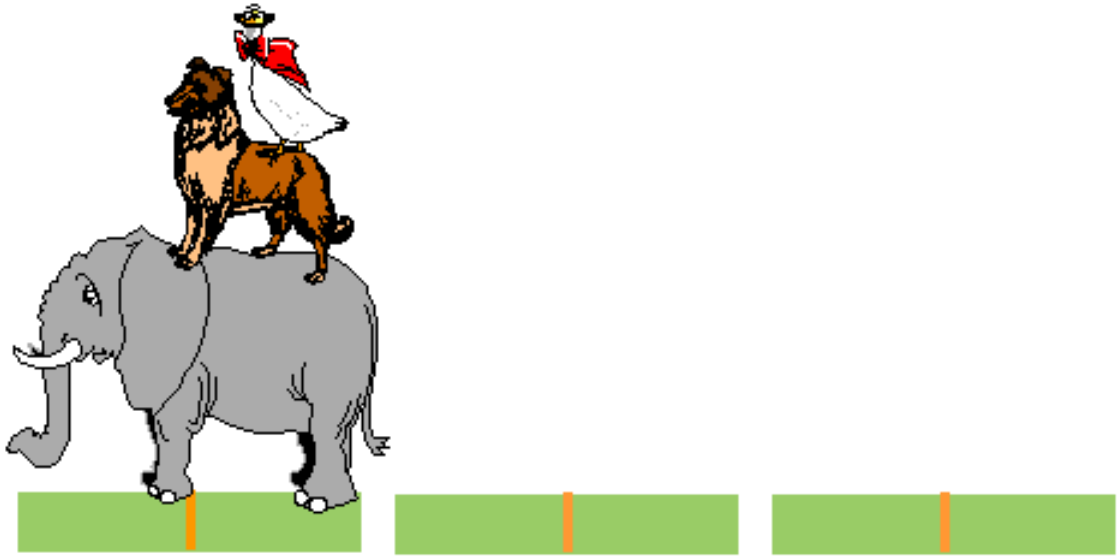
Towers of Hanoi (continued)



Towers of Hanoi (continued)



Towers of Hanoi (continued)



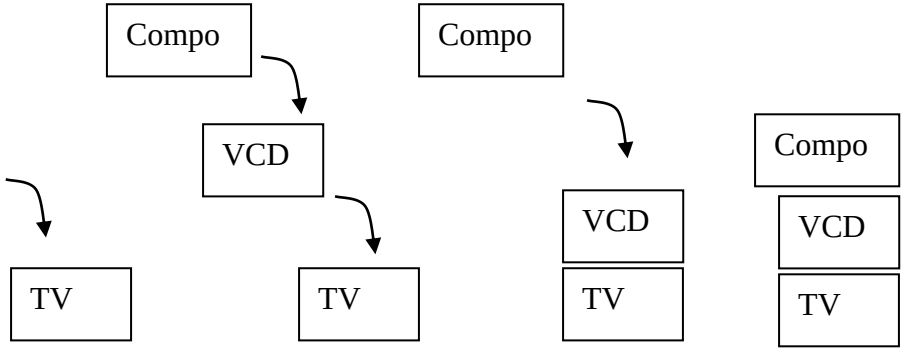
Towers of Hanoi: The Code

```c
#include <stdio.h>

// Function to move a disk from one rod to another
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod) {
    if (n == 1) {
        printf("\nMove disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    printf("\nMove disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

int main() {
    int n = 3; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B, and C are the three rods
    return 0;
}
```

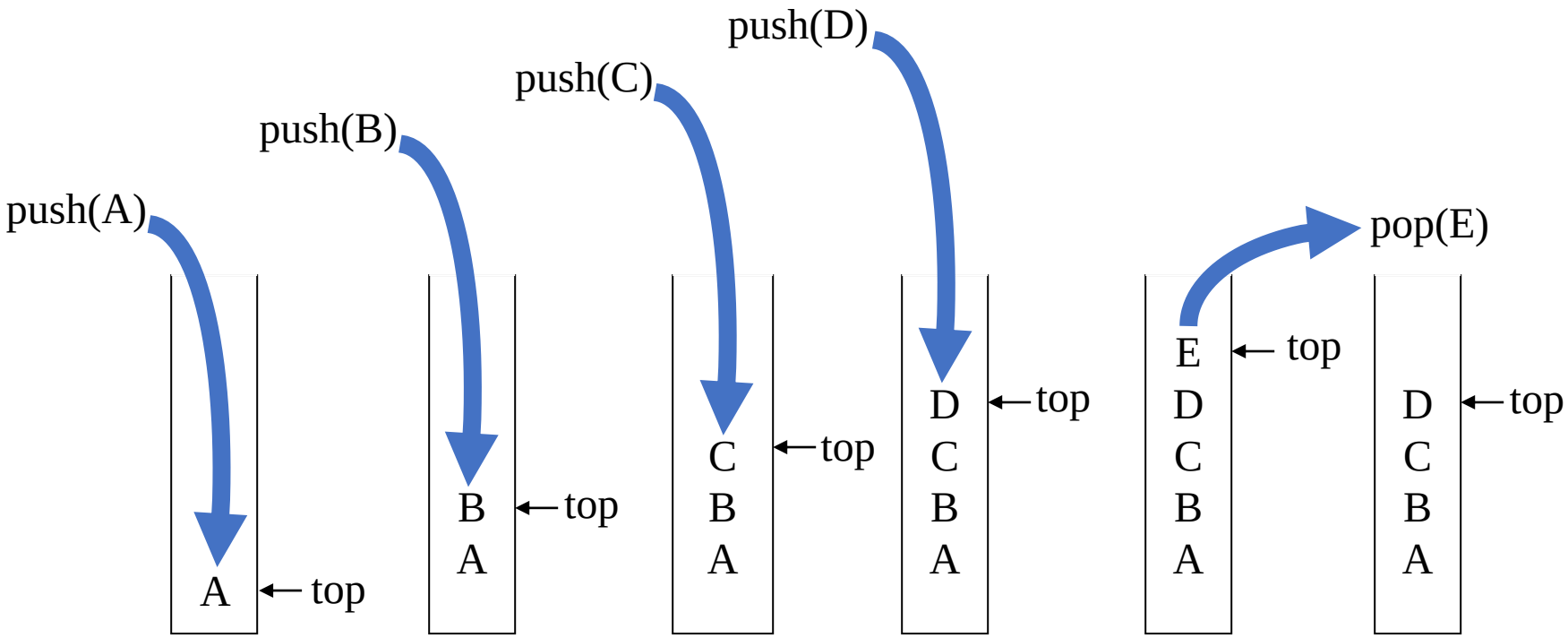
```
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
```

Stack: The Definition

- A data collection arrangement where data can be added and deleted is always done at the end of the data, which is called the top of the stack (TOS)
- Stack is LIFO (Last In First Out)
  - The last object to enter the stack will be the first to leave the stack



Last In First Out

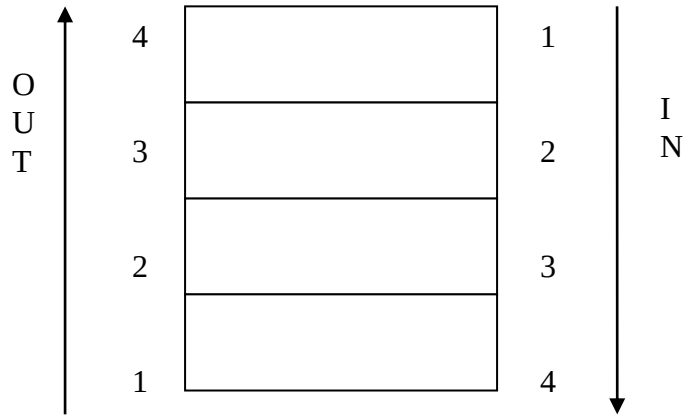


Stack: The Application

- Real life
  - Pile of books (stack of books)
  - Plate trays (stacks of plates)
- More applications related to computer science
  - Program execution stack (read more from your text)
  - Evaluating expressions

Stack: The Operation

- Push: used to add items to the stack at the top of the stack
- Pop: used to take items on the stack at the top of the stack
- Clear: used to clear the stack
- IsEmpty: function used to check whether the stack is empty
- IsFull: function used to check whether the stack is full



Stack with Array of Structs

- Define a Stack using a struct
- Define the MAX_STACK constant to store the maximum contents of the stack
- The Stack struct element is an array of data and the top indicates the top data position
- Create a stacked variable as an implementation of the Stack struct
- Declare the operations/functions above and implement them

Stack: The Program

- Example of MAX_STACK declaration

```
#define MAX_STACK 10
```

- Example of STACK declaration with struct and data array

```
typedef struct myStruct {
   int top;
   int data[10];
} STACK;
```

- Declare/create variables from structs

```
STACK stack;
```

Stack: The Program (continued)

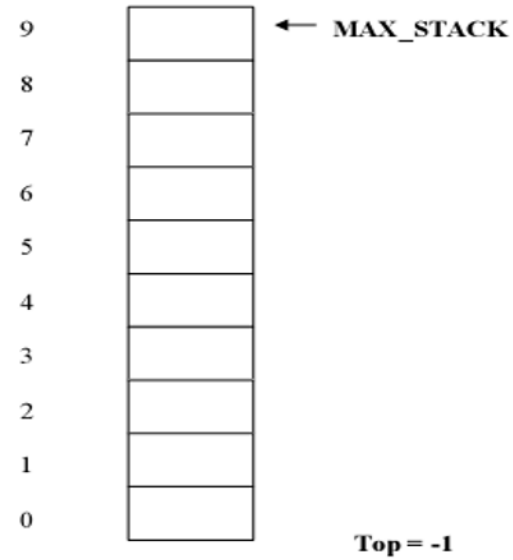
```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK 10

typedef struct myStruct {
    int top;
    int data[10];
} STACK;
STACK stack;

void init() {
    stack.top = -1;
}

int isFull() {
    return (stack.top == MAX_STACK - 1) ? 1 : 0;
}

int isEmpty() {
    return (stack.top == -1) ? 1 : 0;
}

void push(int data) {
    stack.data[++stack.top] = data;
}

void pop() {
    printf("Popped data: %d\n", stack.data[stack.top]);
    stack.top--;
}

void printStack() {
    for (int i = stack.top; i >= 0; i--) {
        printf("Stack data index #%d: %d\n", i, stack.data[i]);
    }
}

void addData() {
    int data;
    while (!isFull()) {
        printf("Add the data: ");
        scanf("%d", &data);
        push(data);
    }
}

void getData() {
    while (!isEmpty()) {
        pop();
    }
}

void main() {
    printf("Initialise the stack.\nSet the top of stack with -1.\nIndex starts from 0.\nSo, it means it's empty stack.\n");
    init();
    printf("\nAdd the stack with 10 numbers.\n");
    addData();
    printf("\nShow the content of the stack.\n");
    printStack();
    printf("\nGet the data from the stack. There're 10 numbers.\n");
    getData();
}
```

```
Initialise the stack.
Set the top of stack with -1.
Index starts from 0.
So, it means it's empty stack.

Add the stack with 10 numbers.
Add the data: 1
Add the data: 2
Add the data: 3
Add the data: 4
Add the data: 5
Add the data: 6
Add the data: 7
Add the data: 8
Add the data: 9
Add the data: 10

Show the content of the stack.
Stack data index #9: 10
Stack data index #8: 9
Stack data index #7: 8
Stack data index #6: 7
Stack data index #5: 6
Stack data index #4: 5
Stack data index #3: 4
Stack data index #2: 3
Stack data index #1: 2
Stack data index #0: 1

Get the data from the stack. There're 10 numbers.
Popped data: 10
Popped data: 9
Popped data: 8
Popped data: 7
Popped data: 6
Popped data: 5
Popped data: 4
Popped data: 3
Popped data: 2
Popped data: 1
```

Stack: The Program (continued)

- Initialization: `init()`
  - Initially fill the top with -1, because arrays in C language start from 0, which means that the data stack is EMPTY!
  - Top is a marker variable in the Stack that indicates the top element of the current Stack data. **Top of Stack** will always move until it reaches the MAX_STACK which causes the stack to be FULL

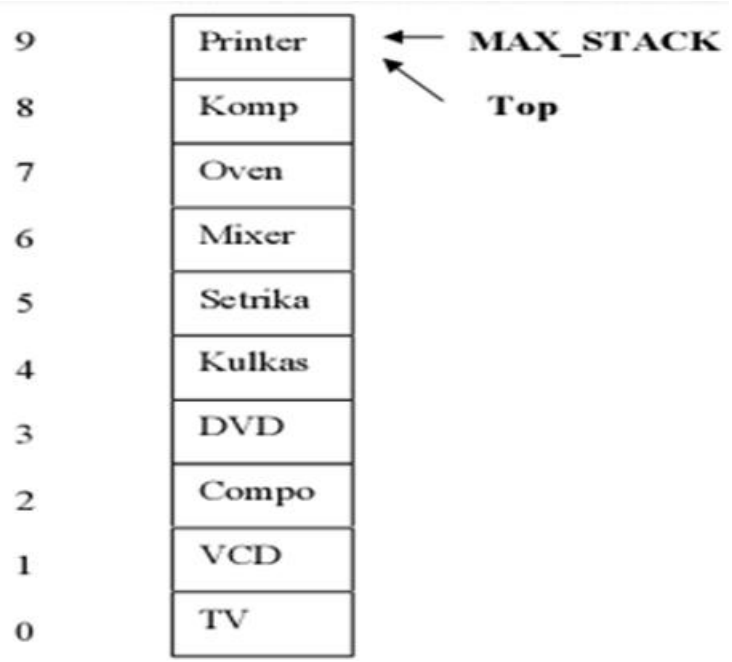
```
11  void init() {
12      stack.top = -1;
13  }
```



Stack: The Program (continued)

- `isFull()`
  - To check whether the stack is full?
  - By checking the **Top of Stack**
    - If it is the same as MAX_STACK-1 then it is full
    - If not (it's still smaller than MAX_STACK-1) then it is not full

```
15  int isFull() {
16      return (stack.top == MAX_STACK - 1) ? 1 : 0;
17  }
```



Stack: The Program (continued)

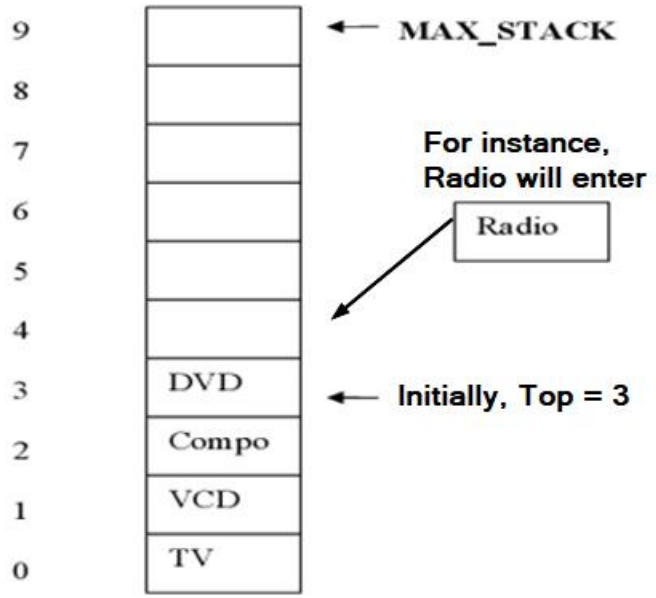
- `isEmpty()`
  - To check whether the Stack data is still empty
  - By checking the **Top of Stack**, if it is still -1 then it means the Stack data is still empty!

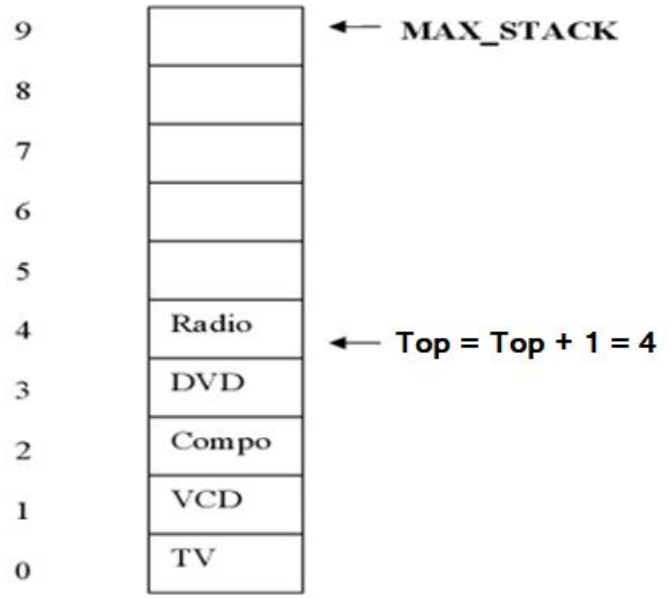
```
19  int isEmpty() {
20      return (stack.top == -1) ? 1 : 0;
21  }
```

Stack: The Program (continued)

- `push()`
  - To insert elements into Stack data. The input data is **always** the top element of the Stack (which is pointed by ToS, **Top of Stack**)
  - If the **data is not yet full**,
    - Add one (increment) value to the **Top of Stack** first every time there is an addition to the Stack data array.
    - Fill the new data into the stack based on the **Top of Stack** index that was previously incremented.
  - If not, output “Full”

Stack: The Program (continued)



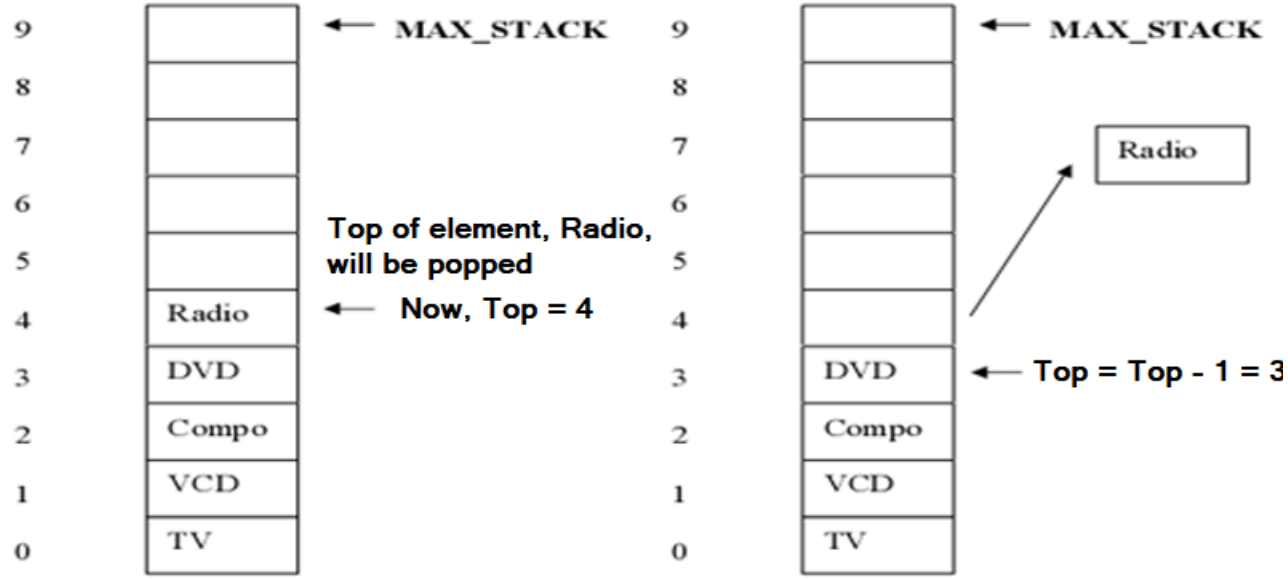



```
23  void push(int data) {
24      stack.data[++stack.top] = data;
25  }
```

Stack: The Program (continued)

- `pop()`
  - To retrieve Stack data at the top (data which is pointed by ToS, **Top of Stack**)
  - First, display the value of the top element of the stack by accessing its index according to the **Top of Stack**, then decrease the value of the **Top of Stack** so that the number of stack elements is reduced

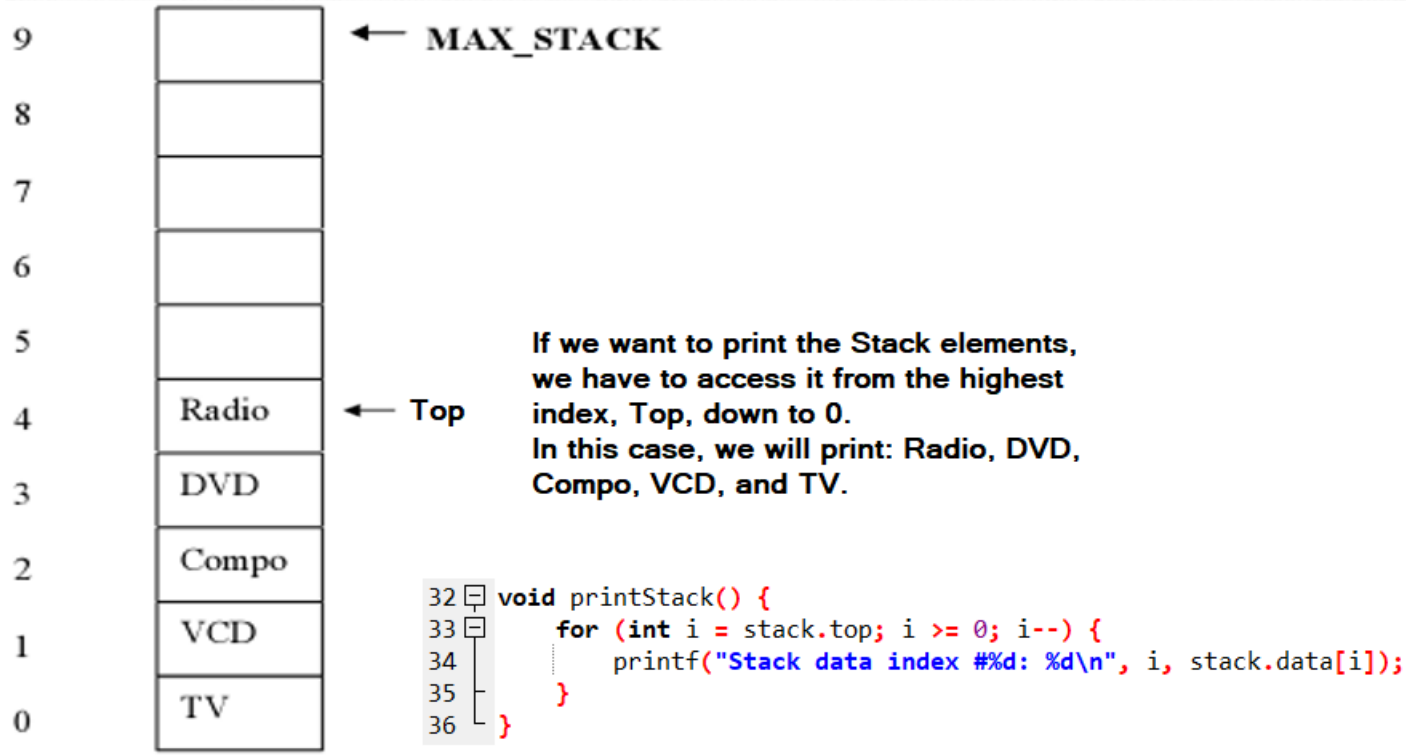
```
27  void pop() {
28      printf("Popped data: %d\n", stack.data[stack.top]);
29      stack.top--;
30  }
```



Stack: The Program (continued)

- `printStack()`
  - To display all of Stack data elements
  - By looping all the array values in reverse, because we have to access from the highest array index first and then to the smaller index!

Stack: The Program (continued)



If we want to print the Stack elements, we have to access it from the highest index, Top, down to 0.
In this case, we will print: Radio, DVD, Compo, VCD, and TV.

```
32  void printStack() {
33      for (int i = stack.top; i >= 0; i--) {
34          printf("Stack data index #%d: %d\n", i, stack.data[i]);
35      }
36  }
```

Stack: The Program (continued)

- `peek()`
  - Used to see/pick the ToS, **Top of Stack**

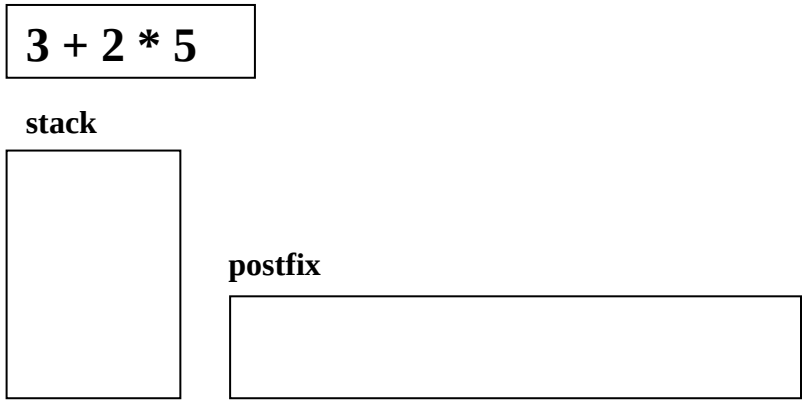
```
53  void main() {
54      printf("Initialise the stack.\nSet the top of stack with -1.\nIndex starts from 0.\nSo, it means it's empty stack.\n");
55      init();
56      printf("\nAdd the stack with 10 numbers.\n");
57      addData();
58      printf("\nShow the content of the stack.\n");
59      printStack();
60      printf("\nToS, the Top of Stack: %d\n", peek());
61      printf("\nGet the data from the stack. There're 10 numbers.\n");
62      getData();
63  }
64
65  int peek() {
66      return stack.data[stack.top];
67  }
```

```
Show the content of the stack.
Stack data index #9: 10
Stack data index #8: 9
Stack data index #7: 8
Stack data index #6: 7
Stack data index #5: 6
Stack data index #4: 5
Stack data index #3: 4
Stack data index #2: 3
Stack data index #1: 2
Stack data index #0: 1

ToS, the Top of Stack: 10

Get the data from the stack. There're 10 numbers.
Popped data: 10
```

Case Study: Scientific Calculator

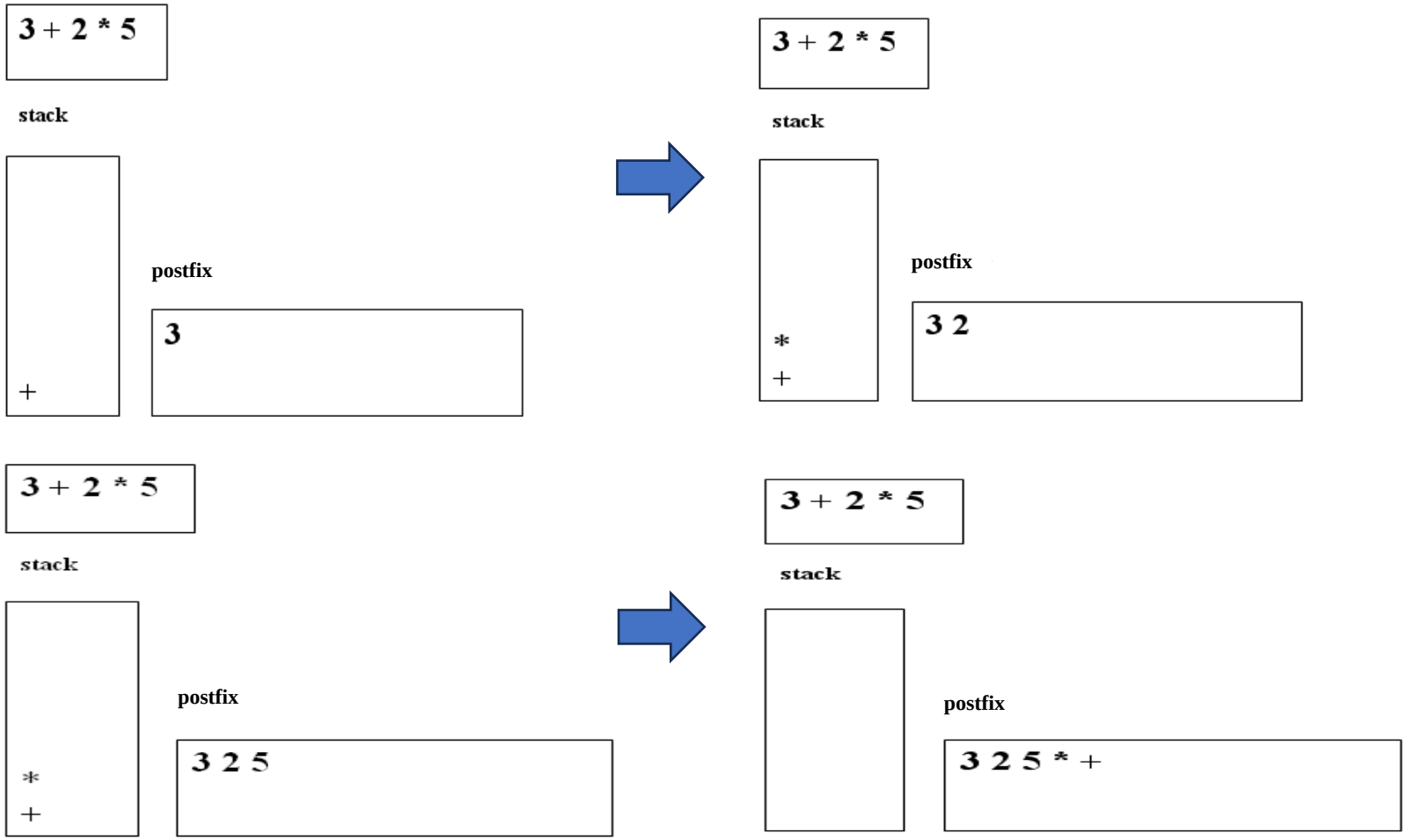
- Suppose the operation is: `3 + 2 * 5`
- The above operation is called **infix** notation
- The **infix** notation must first be changed to **postfix** notation
- `3 + 2 * 5` → postfix notation is `3 2 5 * +`

Case Study: Scientific Calculator (continued)



- Read the input from the front to the back
- If it is an **operand**, then enter it into **postfix**
- If it is an **operator**, then:
  - If the **stack** is still empty, push to the stack
  - If the degree of the problem operator > the degree of the ToS (**Top of Stack**) operator
    - Push the input operator to the stack
  - As long as the problem operator degree <= ToS operator degree
    - Pop the ToS and insert it into the postfix
    - After everything is done, push the input operator to the stack
- If you have read all the input, pop all the contents of the stack and push them to postfix in the correct order

Case Study: Scientific Calculator (continued)



Another example

- a+b*c-d
  - Stack (empty) and Postfix (empty)
- Scan a
  - Postfix: a
- Scan +
  - Stack: +
- Scan b
  - Postfix: ab
- Scan *, because ToS (+) < *, then add to Stack
  - Stack: +*

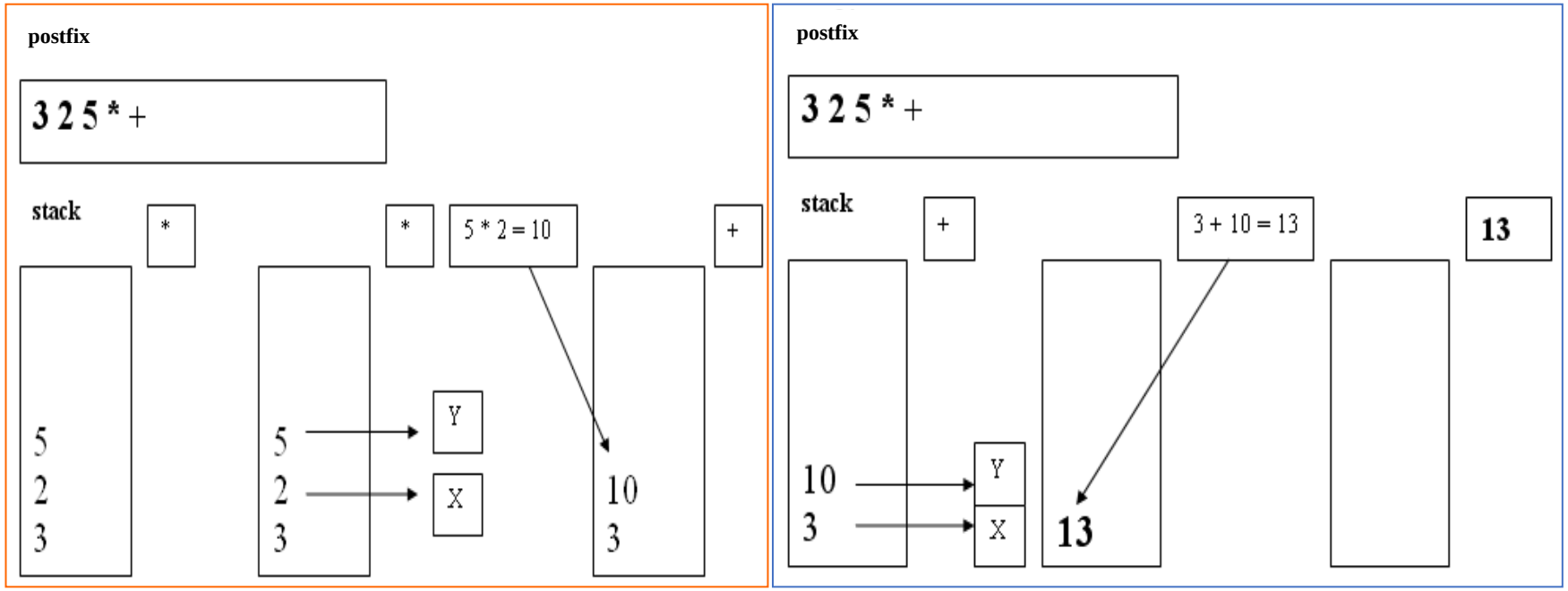
Another example (continued)

- Scans c
  - Postfix: abc
- Scan –, because * > -, then pop Stack, and add to Postfix
  - Stack: +
  - Postfix: abc*
  - Because + >= -, then pop Stack, and add to Postfix, because Stack is empty, then push – to stack
  - Stacks: -
  - Postfix: abc*+
- Scan d
  - Postfix: abc*+d
- Since it’s running out, pop the ToS stack into Postfix
  - Postfix: abc*+d-

Postfix Evaluator

- Scan the Postfix string from the left to the right
- Prepare an empty stack
- If the input is an operand, add it to the stack. If it is an operator, then there will be at least 2 operands on the stack
  - Pop the stack twice, the first pop is stored in y, and the second pop is stored in x. Then evaluate x <operator> y. Save the result and push it onto the stack again.
- Repeat until all inputs have been scanned
- If everything is done, the last element on the stack is the result.
- If there is more than one element, it means an error occurs!

Example: 325*+


postfix
3 2 5 * +
stack
*
*
5 * 2 = 10
+
5
2
3
5
2
3
Y
X
10
3
postfix
3 2 5 * +
stack
+
3 + 10 = 13
13
10
3
Y
X
13

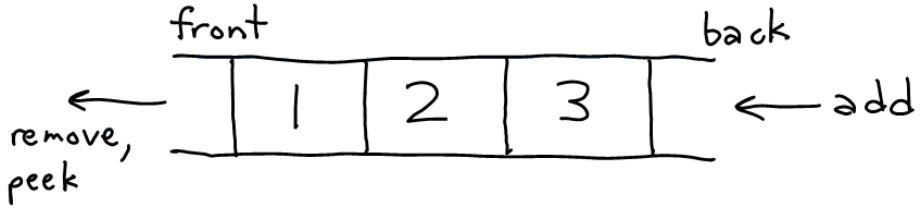
Queue

- What's the difference between Stack and Queue?
  - Stack – a container that allows push and pop
  - Queue – a container that allows enqueue and dequeue

Queue: The Definition & Operation

- Queue: A list with the restriction that insertions are done at one end and deletions are done at the other
  - First-In, First-Out (FIFO)
  - Elements are stored in order of insertion but don't have indexes.
  - Client can only add to the end of the queue, and can only examine/remove the front of the queue.

- Basic queue operations:
  - Add (enqueue): Add an element to the back.
  - Remove (dequeue): Remove the front element.
  - Peek: Examine the elements at the front.



Queue: The Applications

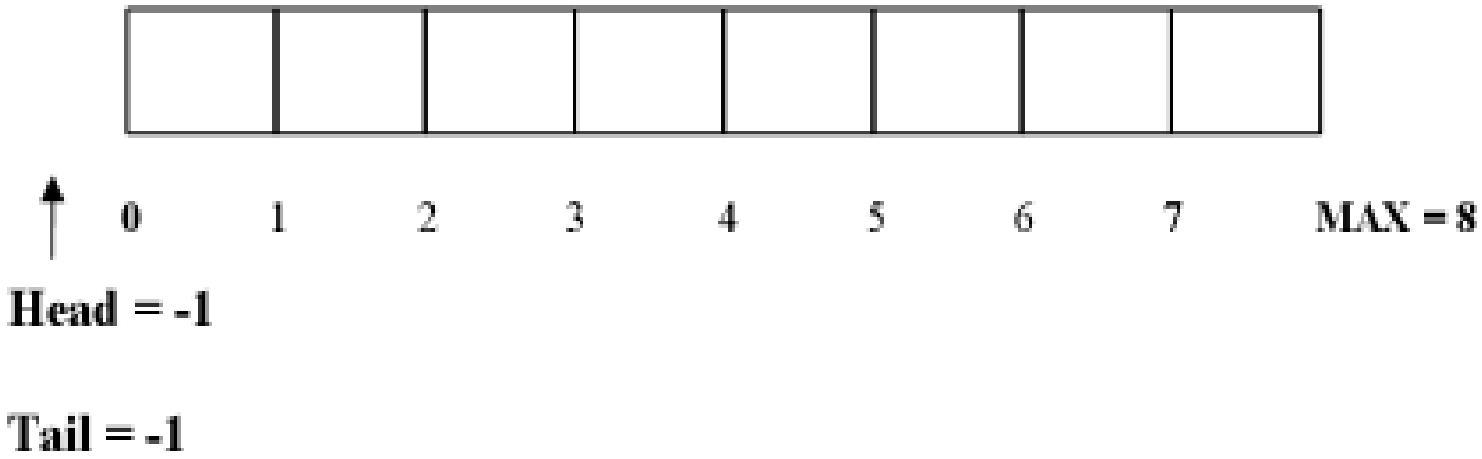
- Real-life examples
  - Waiting in line
  - Waiting on hold for tech support
- Applications related to Computer Science
  - Threads
  - Job scheduling (e.g. Round-Robin algorithm for CPU allocation)

Queue: In Computer Science

- Operating systems:
  - Queue of print jobs to send to the printer
  - Queue of programs/processes to be run
  - Queue of network data packets to send
- Programming:
  - Modelling a line of customers or clients
  - Storing a queue of computations to be performed in order
- Real-world examples:
  - People on an escalator or waiting in a line
  - Cars at a gas station (or on an assembly line)

Queue: Linear Array

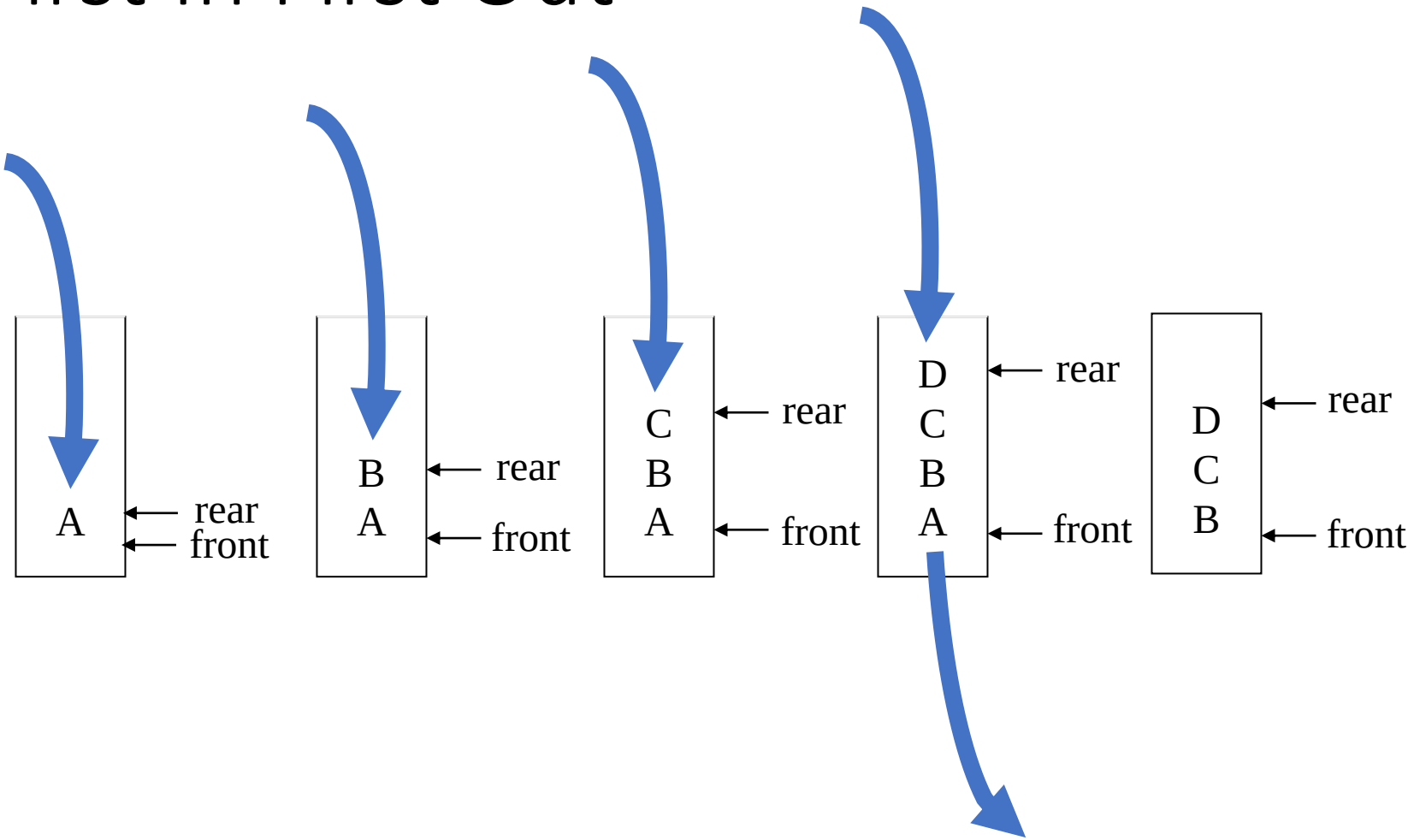
- There is one entrance at one end and one exit at the other end
- So it requires 2 variables: Head and Tail



Queue: Using Array

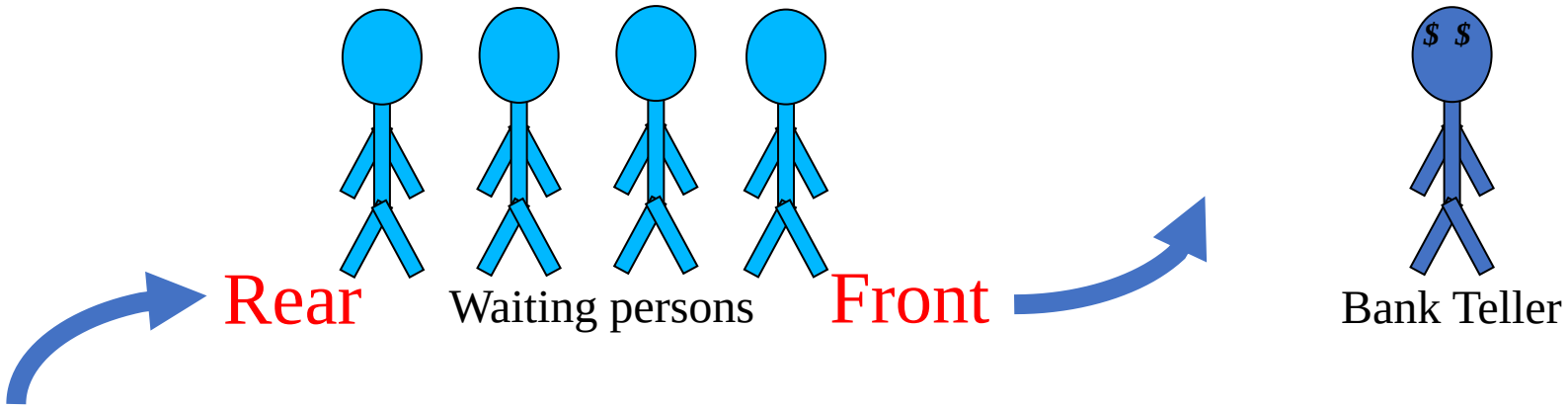
- FIFO (First In First Out)
- The element that enters the queue **first** will be the **first to exit**
- DEQUEUE is removing one element from a Queue
- Queues can be created using: Linear Array and Circular Array

FIFO: First In First Out



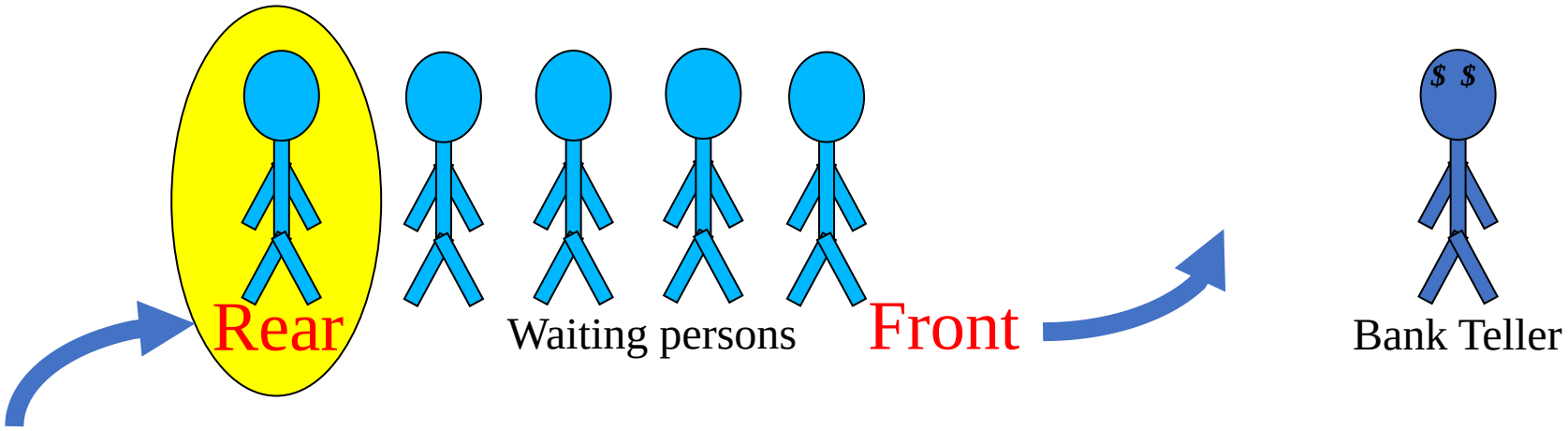
Queue: The Operations

- A queue is like a line of persons waiting for some bank's services by a bank teller
- The queue has a **front** and a **rear**



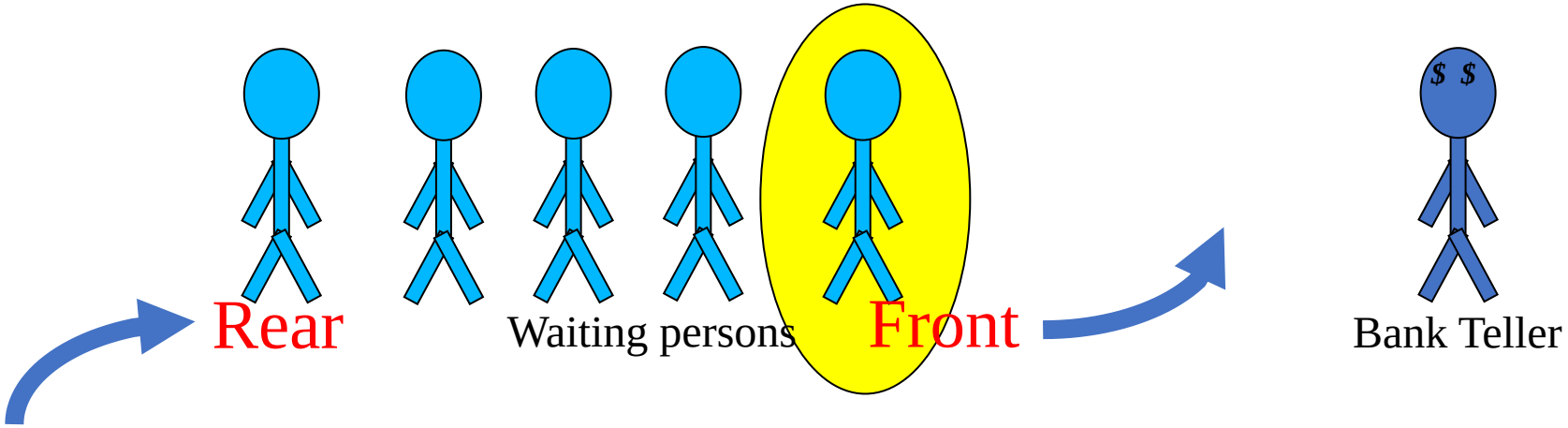
Queue: The Operations (continued)

- An incoming person must enter the queue at the rear
- It's usually called an **enqueue** operation



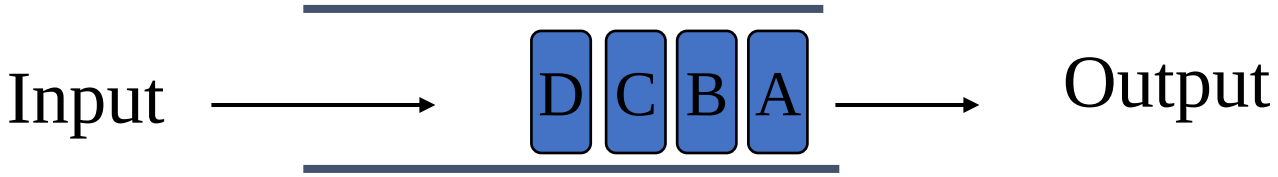
Queue: The Operations (continued)

- When an item is taken from the queue, it always comes from the front
- It's usually called a **dequeue** operation



Queue: The Examples

- Queue: First In First Out (FIFO)



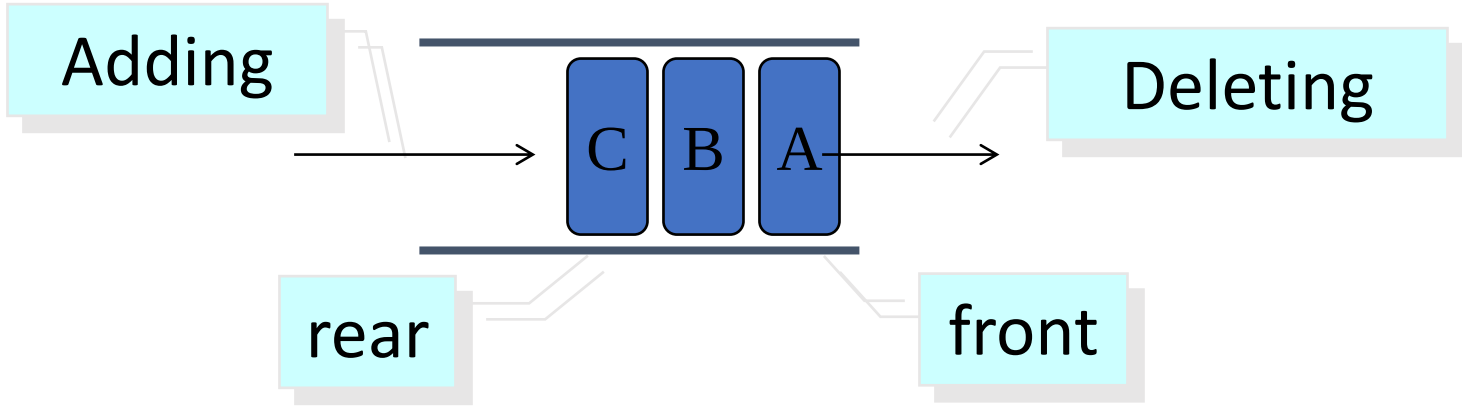
- Toll Stations
  - Car comes, pays, leaves
- Check-out at Big Y market
  - Customer comes, checks out and leaves
- More examples: Printer, Office Hours, …

Queue: More Examples

- In our daily life
  - Airport Security Check
  - Cinema Ticket Office
  - Banks, ATMs
  - Anything else?

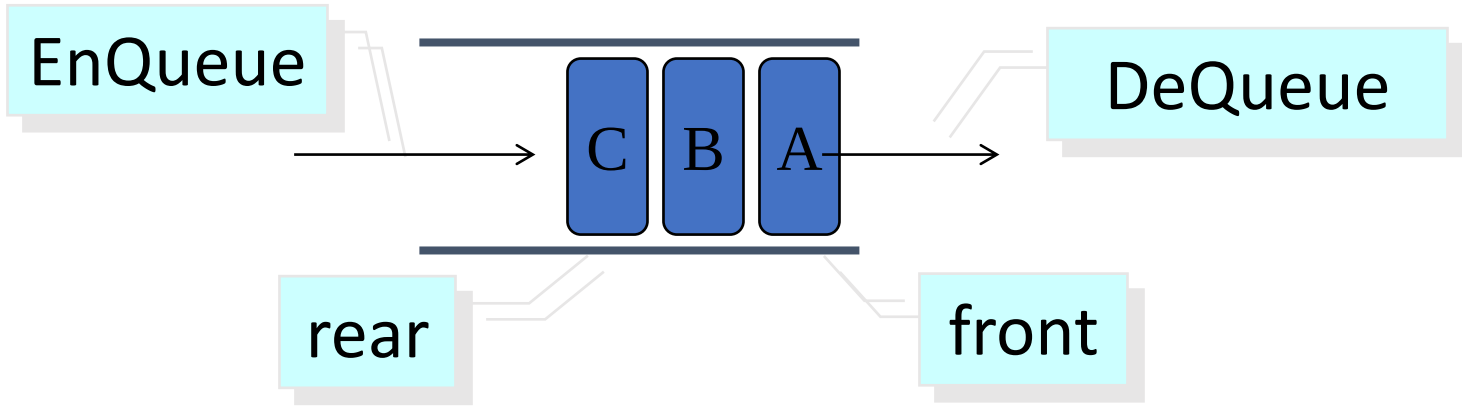
Queue: What is All About?

- Queue is an Abstract Data Type (ADT)
- Adding an entry at the **rear**
- Deleting an entry at the **front**



Queue: Abstract Data Type (ADT)

- Queues
- Operating on both ends
- Operations: EnQueue(**in**), DeQueue(**out**)



Queue: The Mechanism

- Queue is FIFO (First-In First-Out)
- A queue is open at **two ends**
  - You can only add entry (**EnQueue**) at the **rear**, and delete entry (**DeQueue**) at the **front**.
- Note:
  - You cannot **add/extract** entries in the **middle** of the queue!

Queue: Other Applications

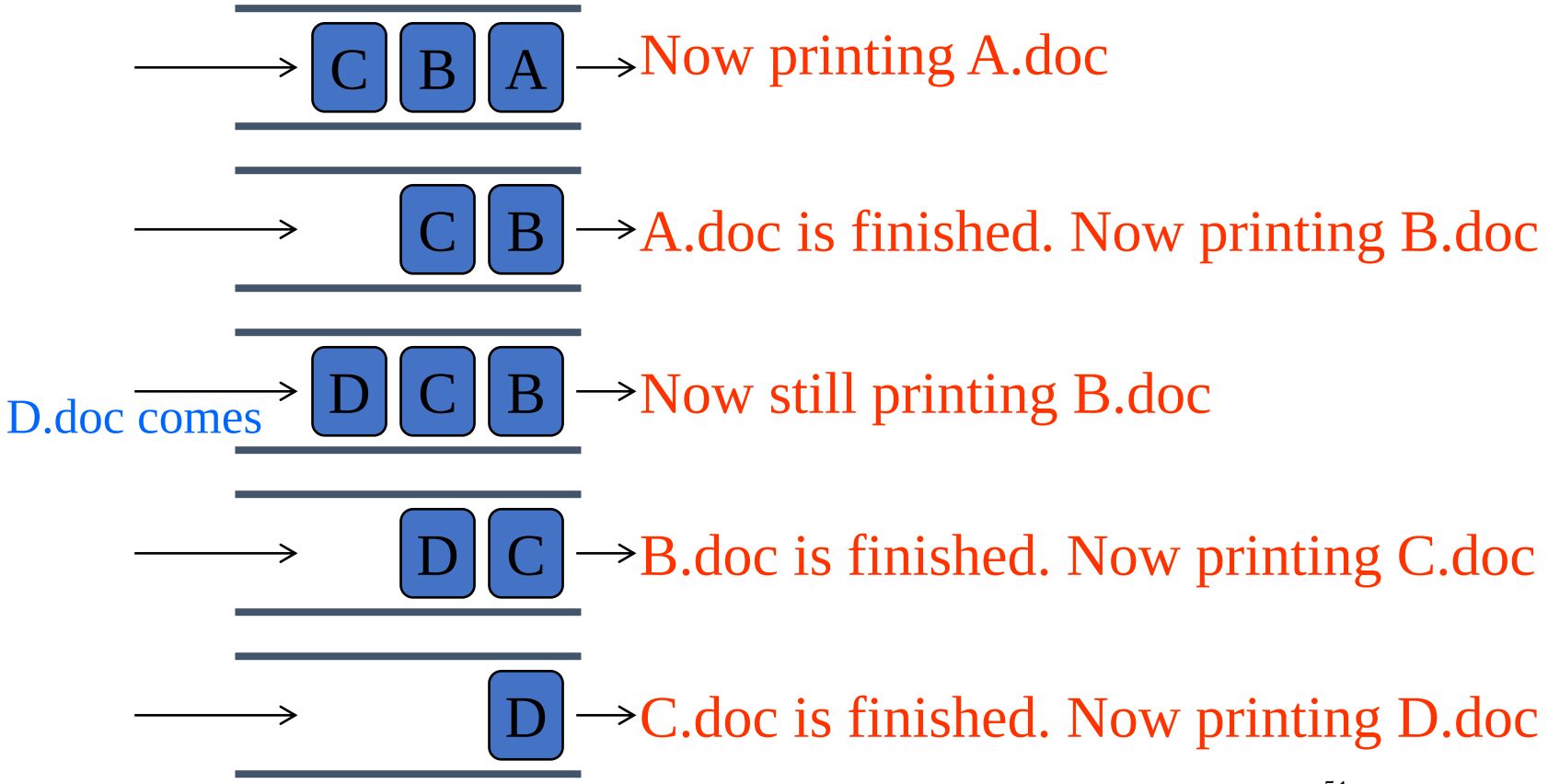
- Printing Job Management
- Packet Forwarding in Routers
- Message queue in Windows
- I/O buffers

Printing Job Management

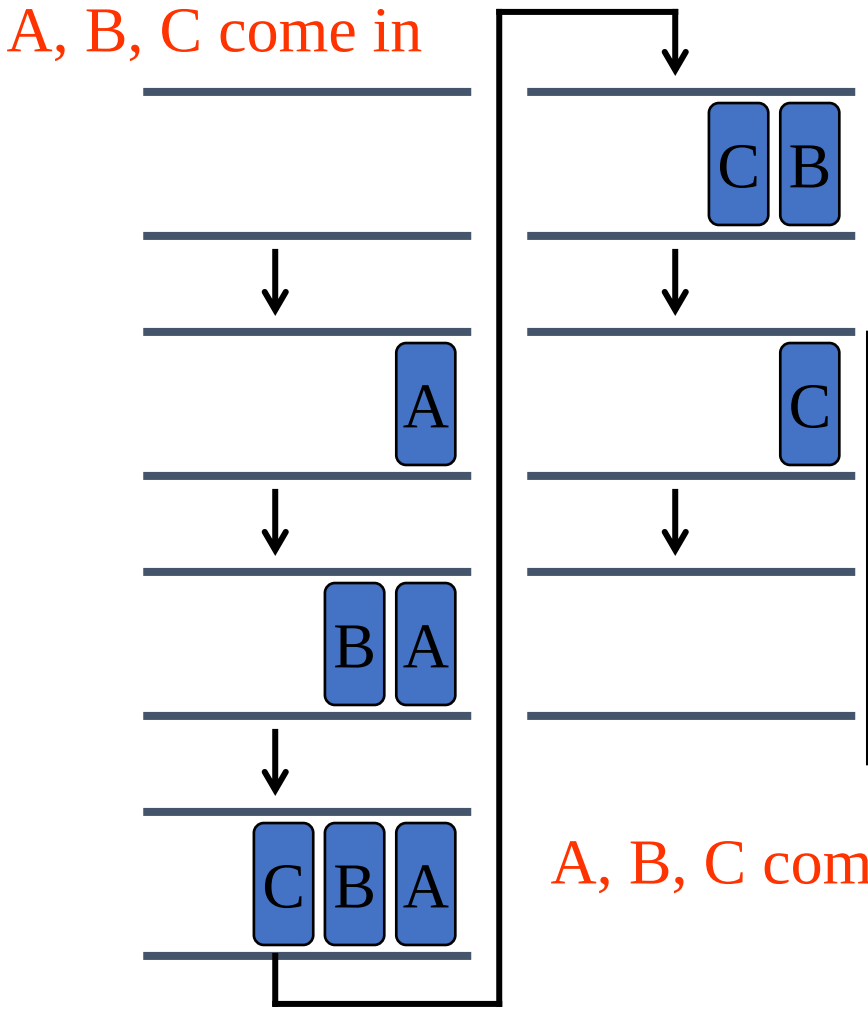
- Many users send their printing jobs to a public printer
- The printer will put them into a queue according to the arrival time and print the jobs one by one
- These printing documents are A.doc, B.doc, C.doc and D.doc

Printing Queue

- A.doc, B.doc, and C.doc arrive at the printer



Printing Queue: FIFO

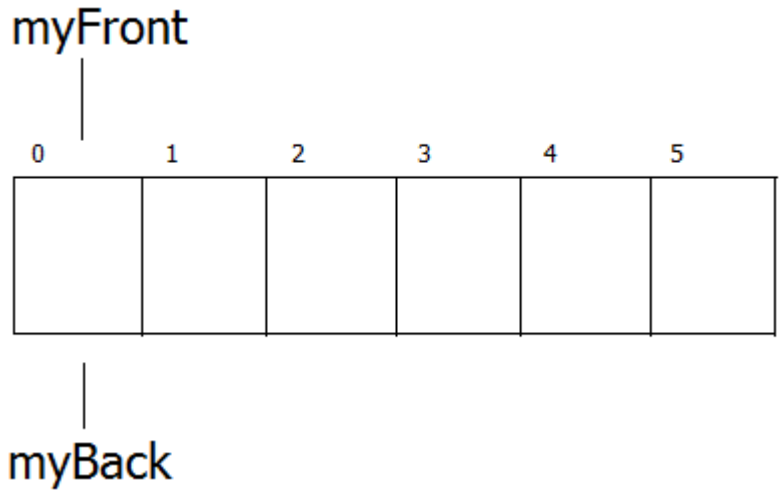


The first one **enqueued** is the first one **dequeued** (FIFO)

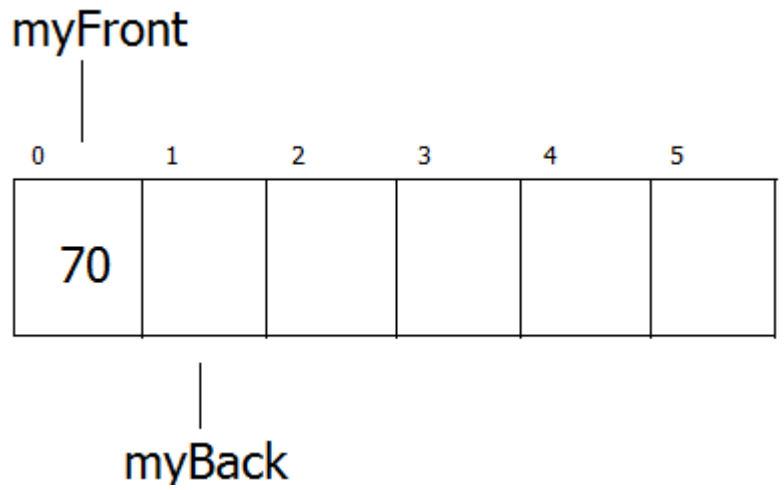
When we enqueue entries in the queue and then dequeue them one by one, we will get the **items** in the **same order**

Queue: The Operation Example

- Empty Queue

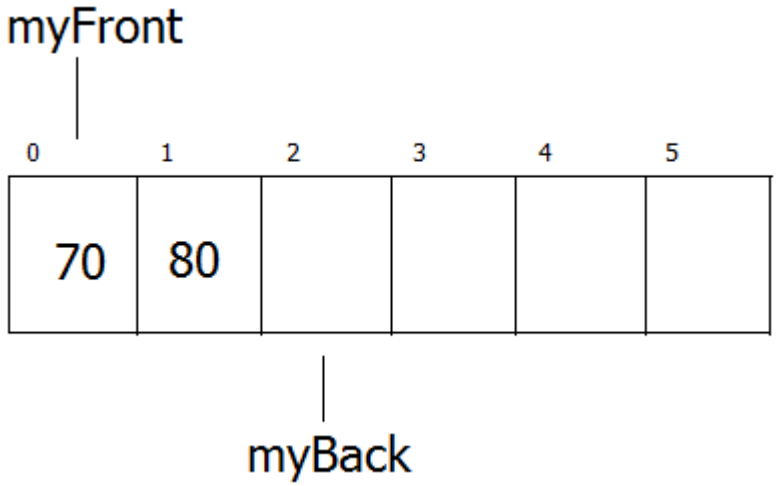


- Enqueue(70)

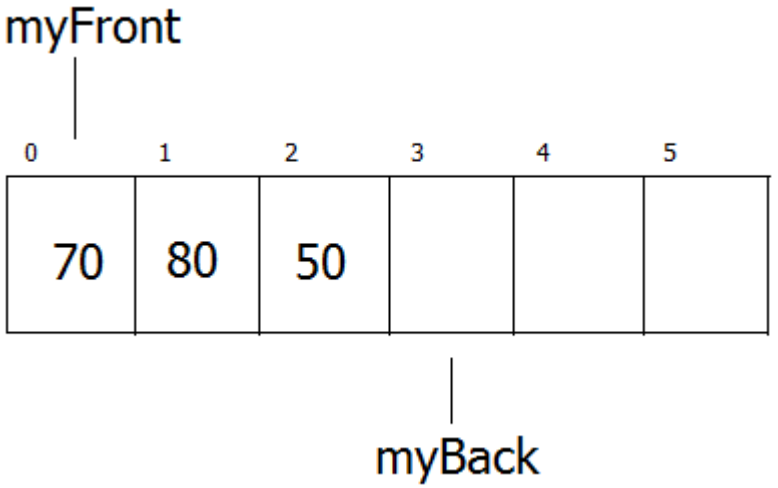


Queue: The Operation Example (continued)

- Enqueue(80)

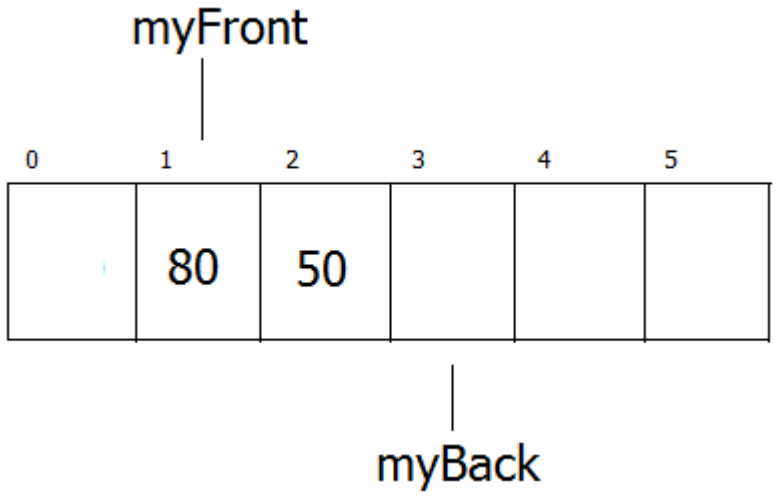


- Enqueue(50)

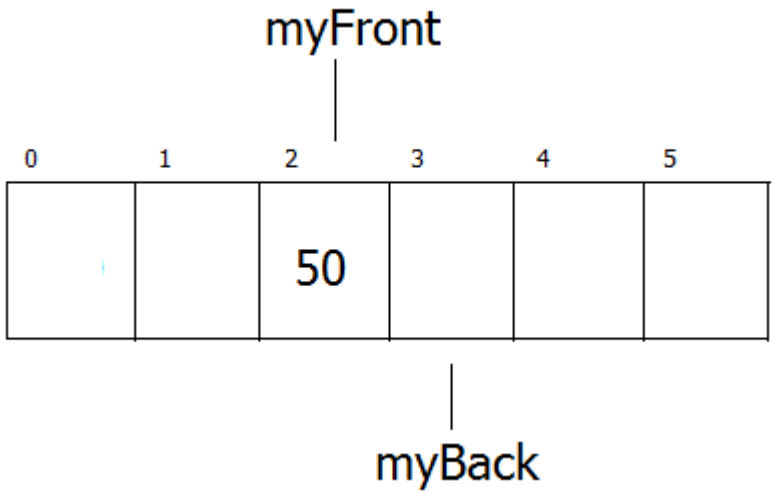


Queue: The Operation Example (continued)

- Dequeue()

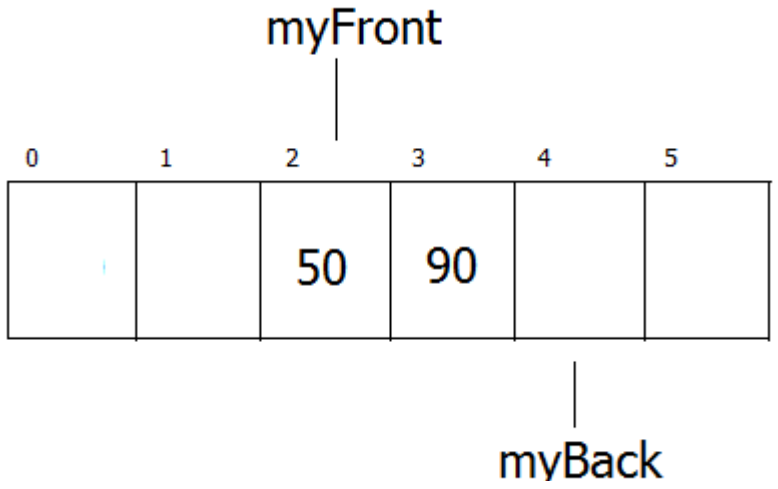


- Dequeue()

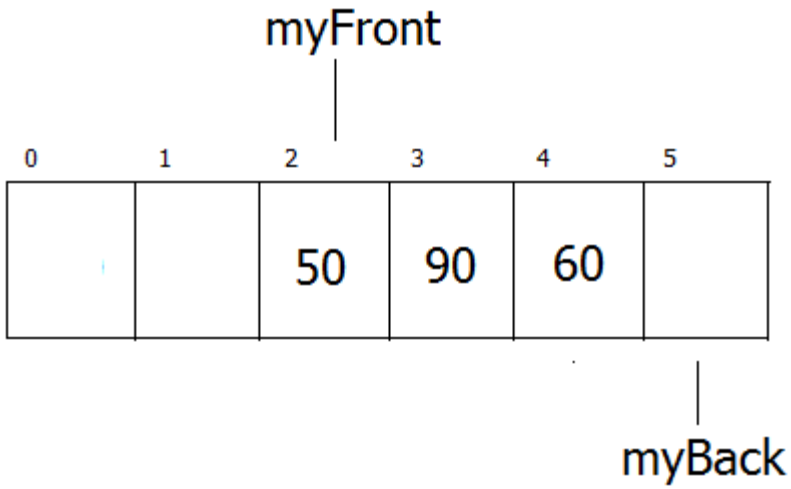


Queue: The Operation Example (continued)

- Enqueue(90)



- Enqueue(60)



Queue: The Code

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 10

typedef struct myQueue {
    int head;
    int tail;
    int data[10];
} QUEUE;
QUEUE queue;

void create() {
    queue.head = queue.tail = -1;
}

int isFull() {
    return (queue.tail == MAX - 1) ? 1 : 0;
}

int isEmpty() {
    return (queue.tail == -1) ? 1 : 0;
}

void enQueue(int data) {
    if (isFull()) {
        printf("Queue is full\n");
    }
    if (isEmpty()) {
        queue.head = 0;
    }
    queue.data[++queue.tail] = data;
}

int deQueue() {
    int tmp;
    if (!isEmpty()) {
        tmp = queue.data[queue.head];
        for (int i = queue.head; i <= queue.tail - 1; i++) {
            queue.data[i] = queue.data[i + 1];
        }
        return tmp;
    }
}

void printQueue() {
    if (!isEmpty()) {
        for (int i = queue.head; i < queue.tail; i++) {
            printf("Queue data #%d is %d\n", i, queue.data[i]);
        }
    }
}

void clearQueue() {
    queue.head = queue.tail = -1;
    printf("Clearing the queue\n");
}

void fillData() {
    int data;
    while (!isFull()) {
        printf("Enter the data: ");
        scanf("%d", &data);
        enQueue(data);
    }
}

void main() {
    create();
    fillData();
    printf("DeQueue %d\n", deQueue());
    printQueue();
}
```

```
Enter the data: 1
Enter the data: 2
Enter the data: 3
Enter the data: 4
Enter the data: 5
Enter the data: 6
Enter the data: 7
Enter the data: 8
Enter the data: 9
Enter the data: 10
DeQueue 1
Queue data #0 is 2
Queue data #1 is 3
Queue data #2 is 4
Queue data #3 is 5
Queue data #4 is 6
Queue data #5 is 7
Queue data #6 is 8
Queue data #7 is 9
Queue data #8 is 10
```

Queue: The Code Explanation

`create()`

- To create and initialize the **queue**
- By making **head** and **tail** = -1

```
 4
 5  typedef struct myQueue {
 6      int head;
 7      int tail;
 8      int data[10];
 9  } QUEUE;
10  QUEUE queue;
11
12  void create() {
13      queue.head = queue.tail = -1;
14  }
15
```

Queue: The Code Explanation (continued)

- The first **queue** condition

head = -1

tail = -1

```
11
12  void create() {
13      queue.head = queue.tail = -1;
14  }
15
```

MAX = 8

Queue: The Code Explanation (continued)

`isEmpty()`

- To check whether the **queue** is **empty** or not
- By checking the **tail** value, if **tail** = -1 then it is **empty**
- We do not check the **head**, because **head** is an indicator for the head of the queue (the **first** element in the queue) which will not change
- Movement in the **queue** occurs by adding queue elements **backwards**, i.e., by using the **tail** value

Queue: The Code Explanation (continued)

- The **queue** is empty, because the **tail** = -1

head = -1

tail = -1

```
19
20  int isEmpty() {
21      return (queue.tail == -1) ? 1 : 0;
22  }
23
```

MAX = 8

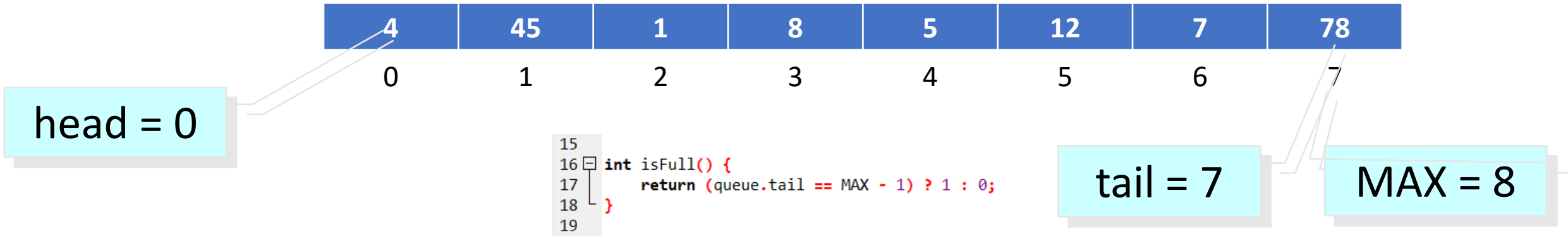
Queue: The Code Explanation (continued)

`isFull()`

- To check whether the **queue** is **full** or not
- By checking the value of **tail**, if **tail** >= MAX-1 (because MAX-1 is the limit of array elements in C) it means it is **full**

Queue: The Code Explanation (continued)

- The **queue** is **full**, because the **tail** = MAX-1




```
15
16  int isFull() {
17      return (queue.tail == MAX - 1) ? 1 : 0;
18  }
19
```

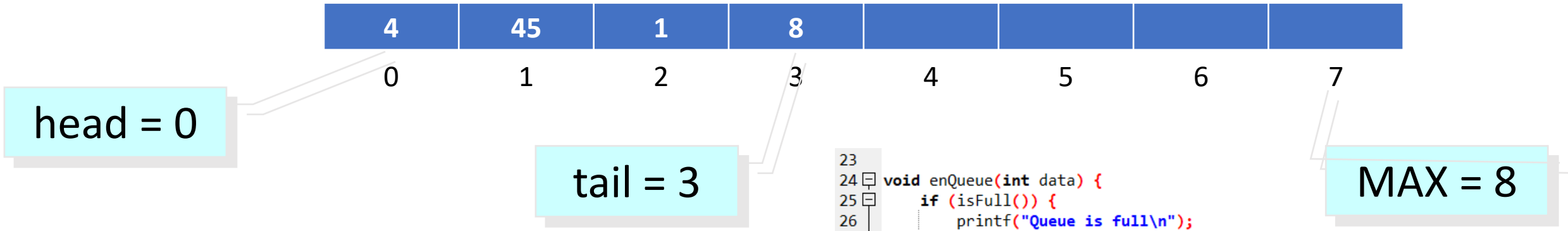
Queue: The Code Explanation (continued)

`enQueue()`

- To **add** an element to the **queue**, the added element is always added to the **last** element
- Adding elements always moves the **tail** variable by incrementing the tail **counter** first

Queue: The Code Explanation (continued)

- The **queue** after `enQueue(8)` has executed




```
23
24  void enQueue(int data) {
25      if (isFull()) {
26          printf("Queue is full\n");
27      }
28      if (isEmpty()) {
29          queue.head = 0;
30      }
31      queue.data[++queue.tail] = data;
32  }
33
```

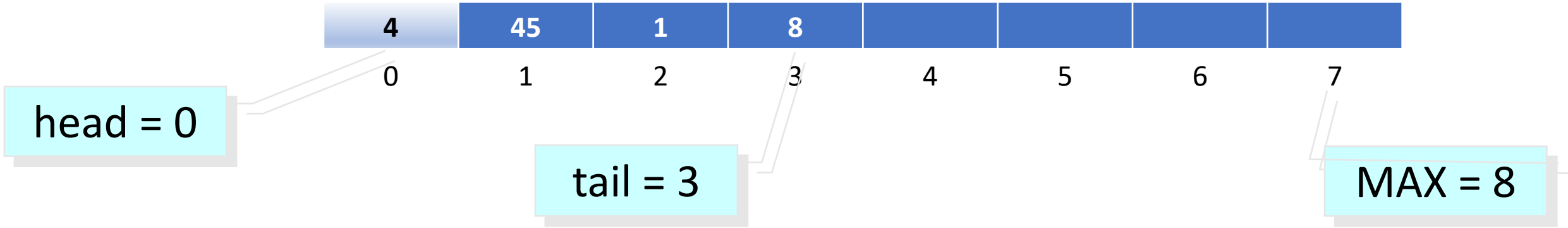
Queue: The Code Explanation (continued)

`deQueue()`

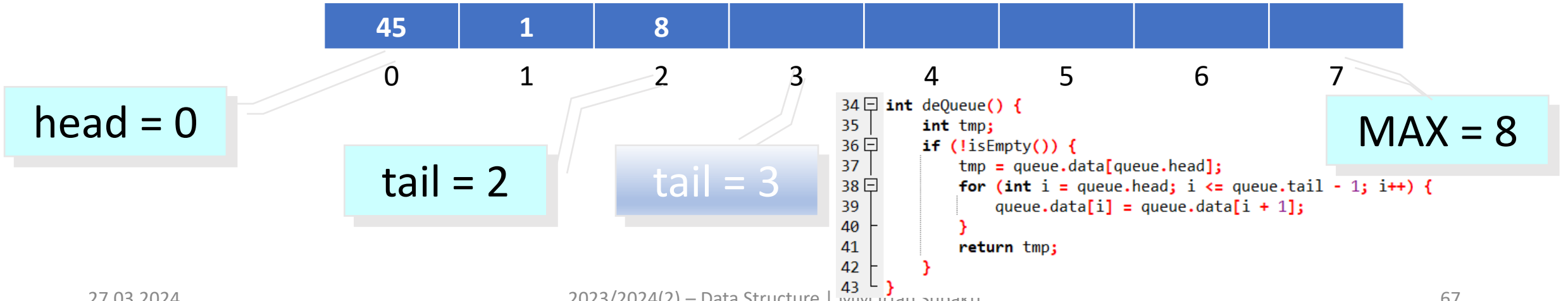
- Used to **delete** the leading/**first** element (**head**) from the **queue**
- By shifting all queue elements forward and reducing the **tail** with 1
- Shifting is done by using **looping**

Queue: The Code Explanation (continued)

- The **queue** after `deQueue()` has executed



- Then, all of element are shifted to the left (move **forward**)




```
34  int deQueue() {
35      int tmp;
36      if (!isEmpty()) {
37          tmp = queue.data[queue.head];
38          for (int i = queue.head; i <= queue.tail - 1; i++) {
39              queue.data[i] = queue.data[i + 1];
40          }
41          return tmp;
42      }
43  }
```

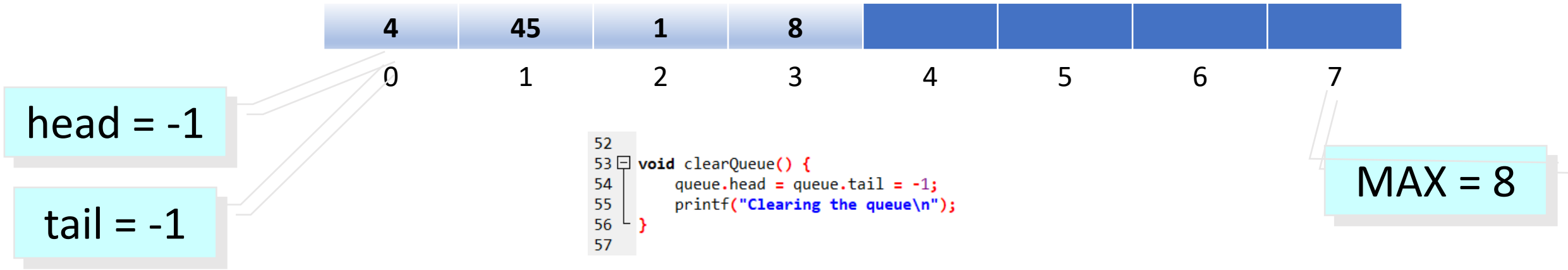
Queue: The Code Explanation (continued)

`clearQueue()`

- To delete the **queue** elements by making **tail** and **head** = -1
- Deleting the queue **elements** does not delete the array, but only sets the **access index** to **-1** so that the queue **elements** are no longer readable

Queue: The Code Explanation (continued)

- The **queue** after `clearQueue()` has executed




```
52
53  void clearQueue() {
54      queue.head = queue.tail = -1;
55      printf("Clearing the queue\n");
56  }
57
```

Queue: The Code Explanation (continued)

`printQueue()`

- To display the queue **element values**
- Using looping from the **head** to the **tail**

```
44
45  void printQueue() {
46      if (!isEmpty()) {
47          for (int i = queue.head; i < queue.tail; i++) {
48              printf("Queue data #%d is %d\n", i, queue.data[i]);
49          }
50      }
51  }
52
```

Exercise

- Add a function to **search** for an **element** in the **queue** & **stack**
- Add a function to **edit an element** in the **queue** & **stack**
- Find the **total**, **average**, **greatest** and **smallest** values of the queue **elements** in a separate function

NEXT: Introduction to **pointers** and **functions by reference**