Stacks

- A stack is an example of a linear data structure
- We can implement a stack using other linear data structures such as:
  1) Arrays
  2) Linked Lists

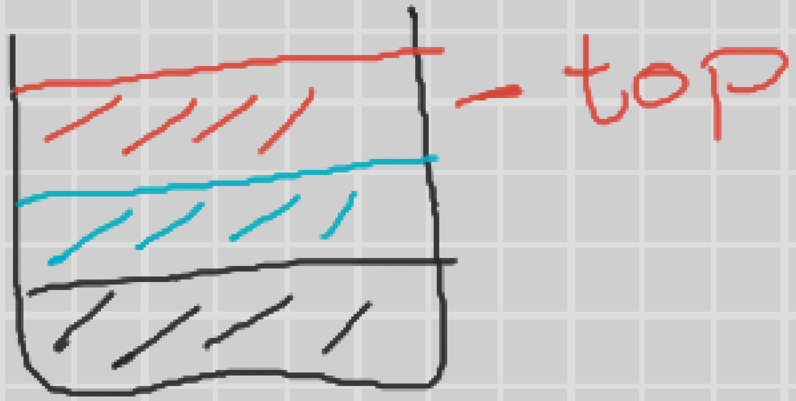
- Objects can be inserted at the top of a stack and removed from the top. LIFO data structure.

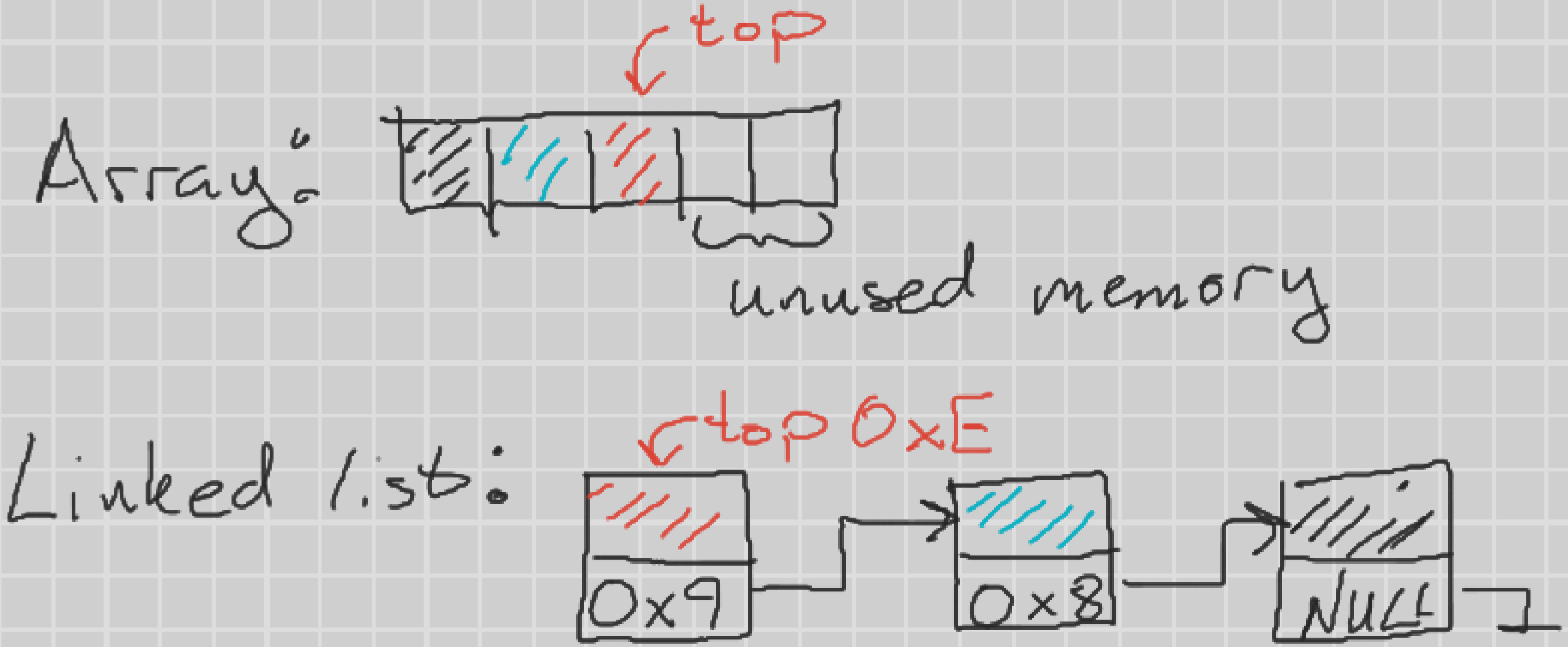
The operations on a stack are:

1) PUSH(x) - inserts an element at the top of a stack
2) POP() - removes an element from the top of the stack
3) PEEK() - returns the element at the top of the stack
4) isEMPTY() - checks if the stack is empty - return true if it is, false otherwise
5) new STACK - creates stack

Stack uses

- Stacks are commonly used for the verification of balanced pairs.
  ex: When your compiler/text editor alerts you of unbalanced parentheses/braces/etc.
- Stacks are also used for editor action history
  ex: CTRL+Z and CTRL+R to undo or redo any action





- Using an array, one can run out of memory because they are fixed in size (C++ for example). You could end up with stack overflow using an array implemetation
- Using a linked list means that we can create as many nodes as needed without worrying too much about memory usage, it can grow or shrink on demand.

Stack operation pseudocode using arrays

Stack PUSH(x): There are three approaches
1) Stop accepting new elements
2) Extend & copy elements into a larger array
3) Do nothing

1) PUSH(x) - we will use a variable called TOP to track where the top of the stack is and it will be initialized with -1 to signify an emtpy stack. When we push to a stack, we assume that memory has already been allocated for an array.

```
function PUSH(x)
    if top = length(A)−1 then
        print("Stack overflow")
        return
    end if
    top ← top + 1
    A[top] ← x
end function
```

This version of PUSH has a time complexity of THETA(1), because all instructions take constant time.

Stack PUSH(x): There are three approaches

1) Stop accepting new elements
2) Extend & copy elements into a larger array
3) Do nothing

2) PUSH(x) - extend and copy needs to create a larger array whenever it runs out of room to add new elements at the top of a stack

```
function PUSH(x)
    if top = SIZE(A) -1
        Extend and copy // usually double the size of the array
    top <-- top + 1
    A[top] <-- x
```

This version of PUSH has a worst case time complexity of THETA(N), because every element needs to copied into the larger array.

Stack POP()

```
function POP()
    if top = −1 then
        print("Empty stack")
        return
    end if
    top ← top − 1
end function
```

NOTE: We are not deleting the element from the array when it gets popped, we only decrease the value of top by 1. Think undo/redo example.

Each instruction takes constant time, so the time complexity is THETA(1)

Stack PEEK()

```
function PEEK()
    if top = −1 then
        print("Empty stack")
        return
    end if
    return A[top]
end function
```

Each instruction inside the PEEK function takes contstant time, therefore the time complexity is THETA(1)

Stack isEMPTY()

```
function ISEMPTY()
    if top = −1 then
        return TRUE
    end if
    return FALSE
end function
```

Each instruction inside the isEMPTY function takes contstant time, therefore the time complexity is THETA(1)

Stack operations pseudocode using linked lists

Stack PUSH(x)

```
function PUSH(data)
    newNode ← new Node(data)
    newNode.next ← top
    top ← newNode
end function
```

Each instruction inside the linked list PUSH function takes contstant time, therefore the time complexity is THETA(1)

```
function POP()
    if top = NULL then
        print("Empty list")
        return
    end if
    top ← top.next
end function
```

```
function PEEK()
    if top = NULL then
        print("Empty list")
        return
    end if
    return top.data
end function
```

```
function ISEMPTY()
    if top = NULL then
        return TRUE
    end if
    return FALSE
end function
```

Every stack operation implemented using a linked list takes constant time. They all have time complexity of THETA(1)

Queues

- A queue is an example of a linear data structure
- We can implement a queue using other linear data structures such as:
    1) Arrays
    2) Linked Lists

- Objects can be inserted at the top of a stack and removed from the top. FIFO data structure.

The operations on a stack are:

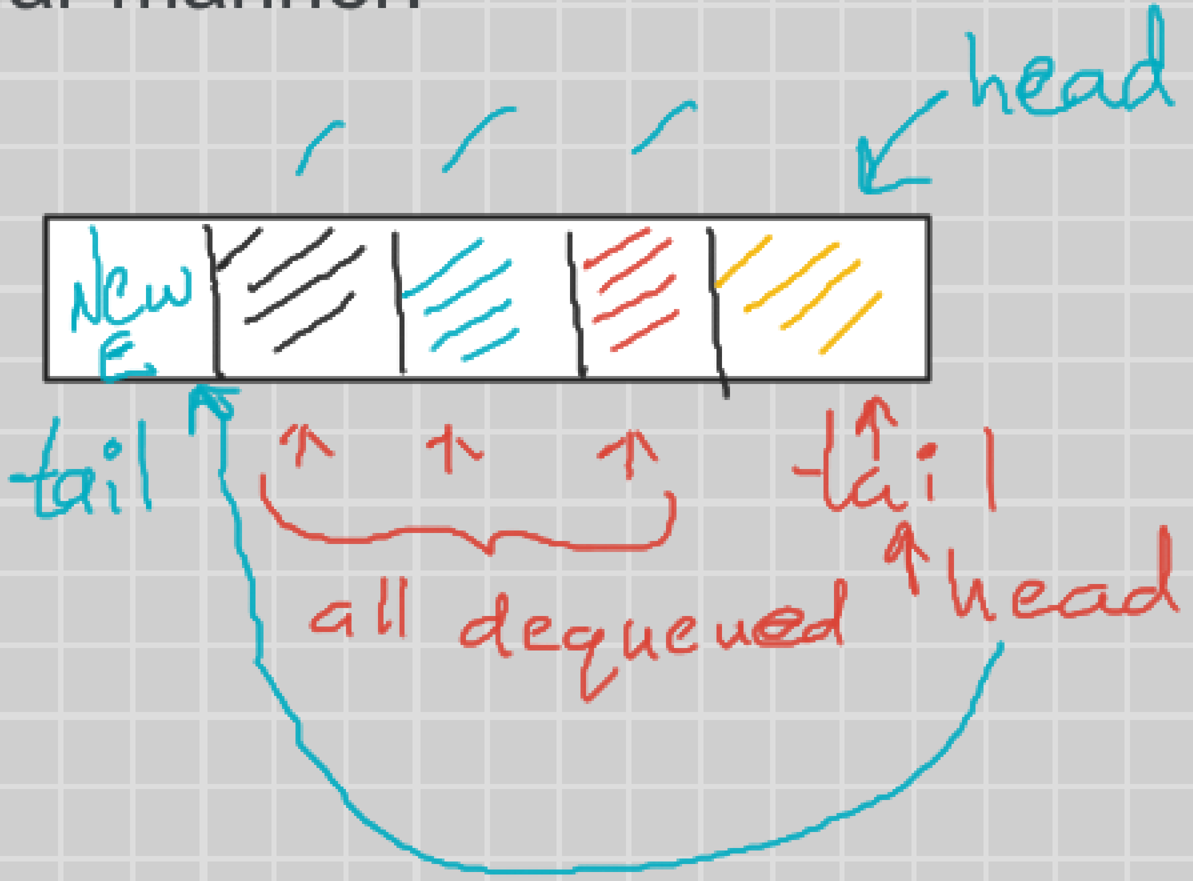
1) ENQUEUE(x) - inserts an element at the tail of a queue
2) DEQUEUE() - removes an element from the front (head) of a queue
3) PEEK() - returns the element at the front (head) of a queue
4) isEMPTY() - checks if the queue is empty - return true if it is, false otherwise

Queues pseudocode using arrays*

```
function ENQUEUE(A, x)
    if (tail + 1)%N = head then
        print("Queue is full")
        return
    end if
    if ISEMPTY() then
        head ← 0
        tail ← 0
    else
        tail ← (tail + 1)%N
    end if
    A[tail] ← x
end function
```

```
function DEQUEUE(A)
    if ISEMPTY() then
        print("Queue is empty")
        return
    end if
    if head = tail then
        head ← −1
        tail ← −1
    else
        head ← (head + 1)%N
    end if
end function
```

We use (tail+1) % N so that we can traverse the array in a circular manner.



```
function PEEK(A)
    if head = −1 then
        print("Queue is empty")
        return
    end if
    return A[head]
end function
```

All of the operations for a QUEUE implemented using an array have time complexity of THETA(1).

```
function ISEMPTY(A)
    if head = −1 then
        return TRUE
    end if
    return FALSE
end function
```

Queues pseudocode using linked lists

```
function ENQUEUE(Q, x)
    newNode ← new Node(x)
    if head = NULL and tail = NULL then
        Q.head ← newNode
    else
        Q.tail.next ← newNode
    end if
    Q.tail ← newNode
end function
```

```
function PEEK()
    if Q.head = NULL and Q.tail = NULL then
        print("Empty queue")
        return
    end if
    return head.data
end function
```

```
function DEQUEUE(Q)
    if Q.head = NULL and Q.tail = NULL then
        print("Queue is empty")
        return
    end if
    if Q.head = Q.tail then
        Q.head ← NULL
        Q.tail ← NULL
    else
        Q.head ← Q.head.next
    end if
end function
```

```
function ISEMPTY()
    if Q.head = NULL and Q.tail = NULL then
        return TRUE
    end if
    return FALSE
end function
```