First there were linked lists…then…

Stacks and Queues

Stacks and Queues

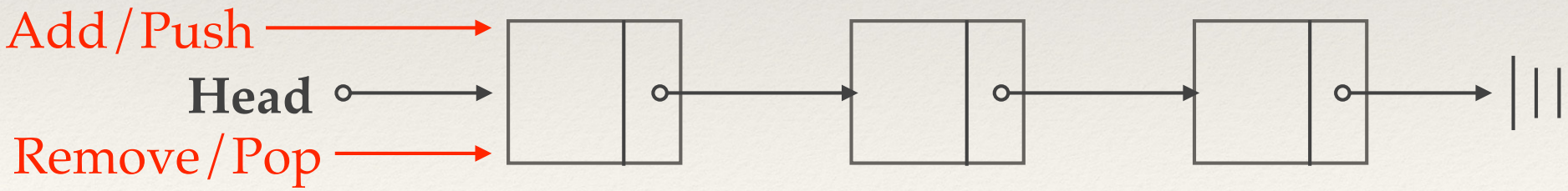
- Stacks and queues are both **linear** data structures
  - All components are arranged in a straight line, *i.e.* **sequential**.
- You can describe their behaviour through **Growth** and **Decay** Rules.
- These structures obey behavioural rules.
  - Knowledge about how they are implemented is **not** required to understand their uses.

Stacks

Please get one and bring it here

Stacks: Growth/Decay Rules

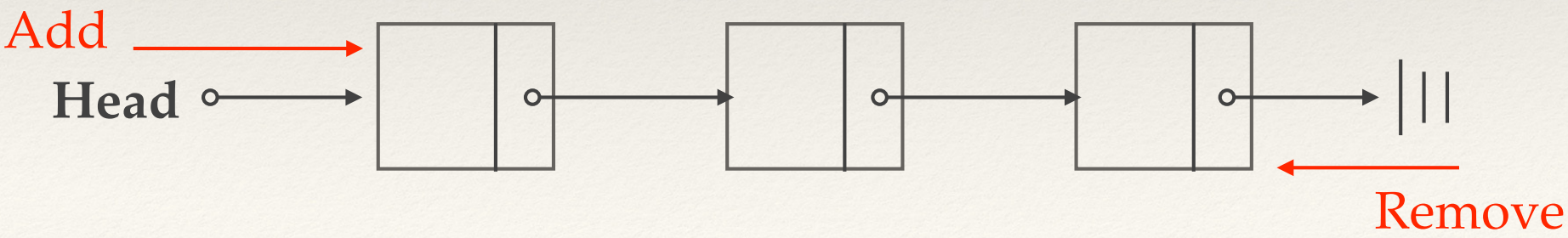
- Add and remove components from the same end.
- Adding is called a **Push**.
- Removing is called a **Pop**.
- First added, last removed (**LIFO** - last in first out)



Queues

Queues: Growth/Decay Rules

- Add and remove components from the **opposite** ends.
- Adding is called **enqueue**.
- Removing is called **dequeue**
- First added, first removed (FIFO - first in first out)



Implementations

Stacks and queues can be built many different ways

- linked lists
- arrays (i.e. sequential memory)

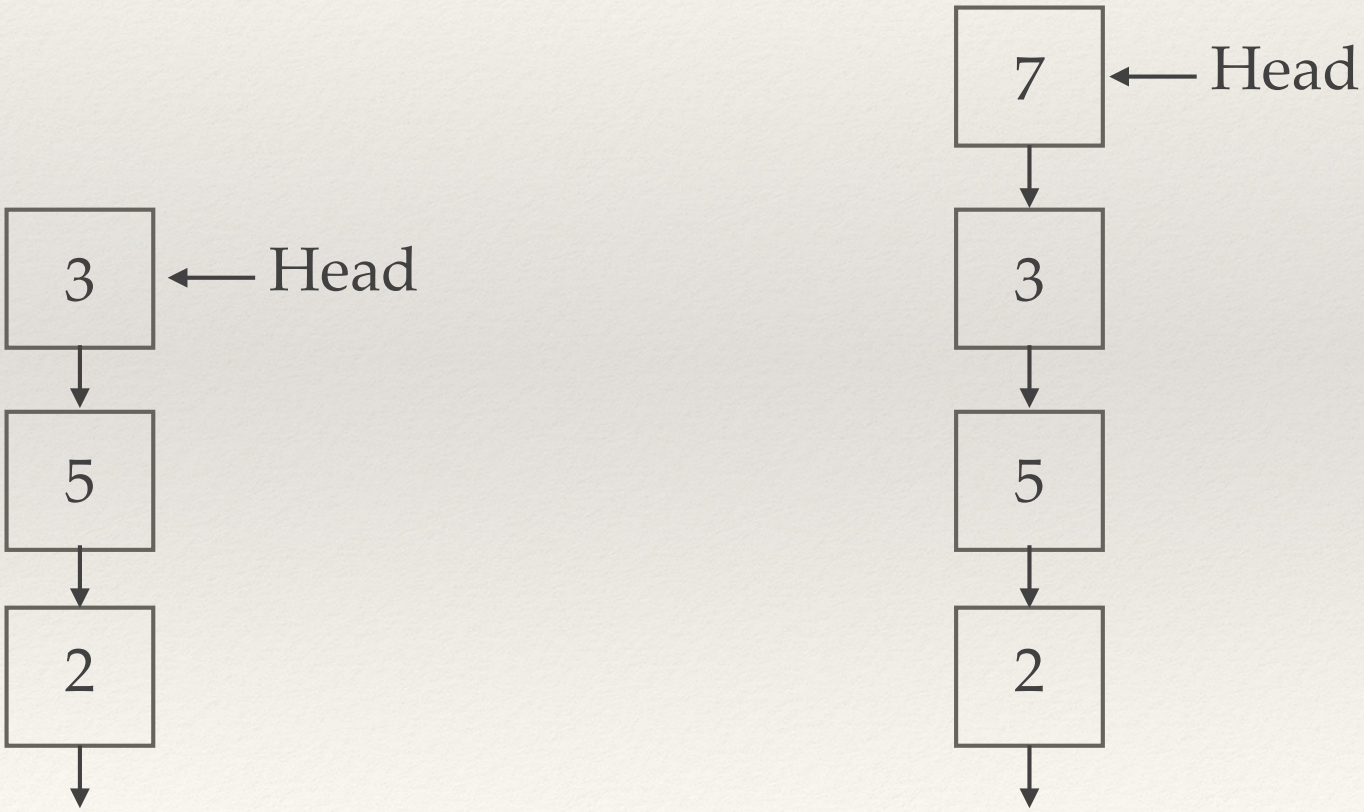
Linked List Implementation: Stacks

- The list representation requires dynamically allocated structures to be added/removed.

Linked List Implementation: Stacks

- push: adding an element onto a stack

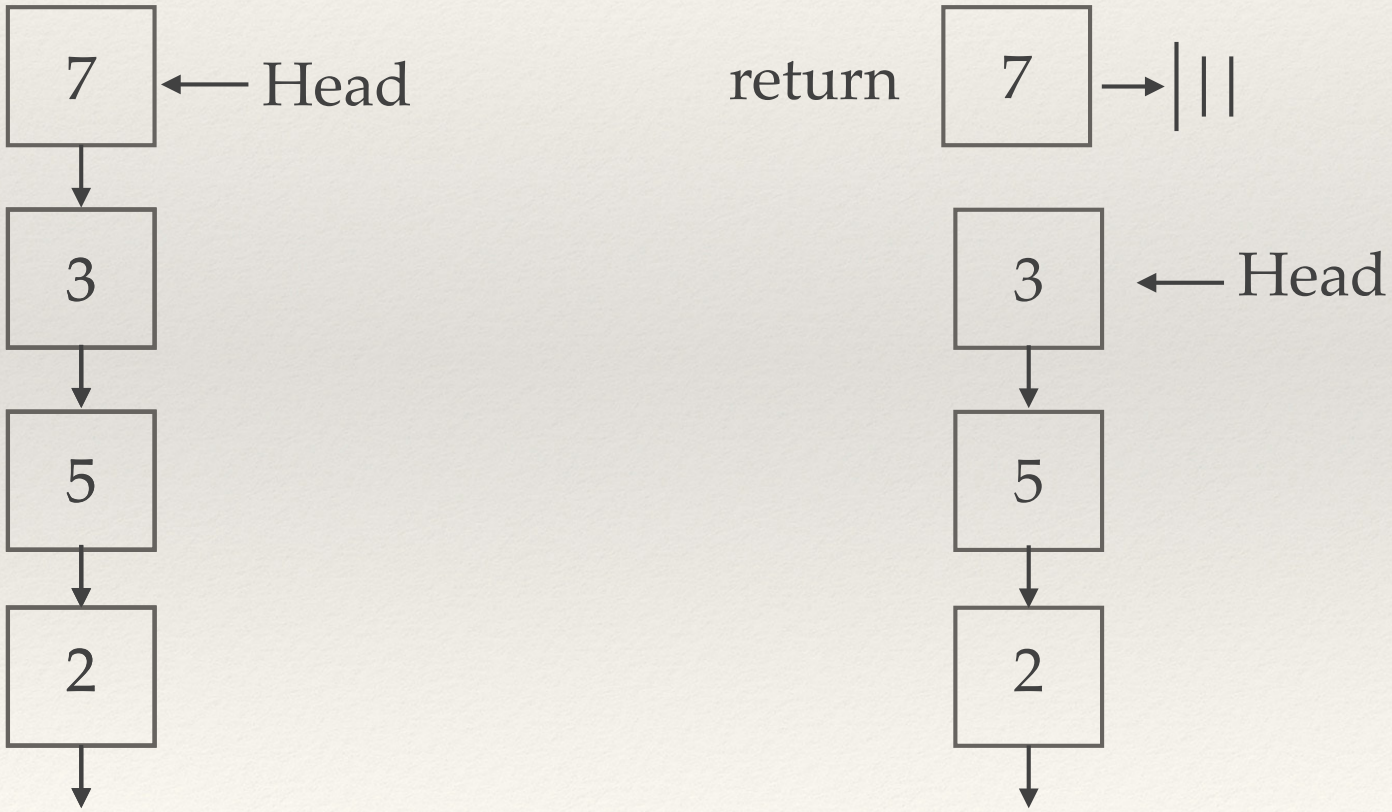
  Stacks grow from the head



Linked List Implementation: Stacks

- pop: removing a stack element

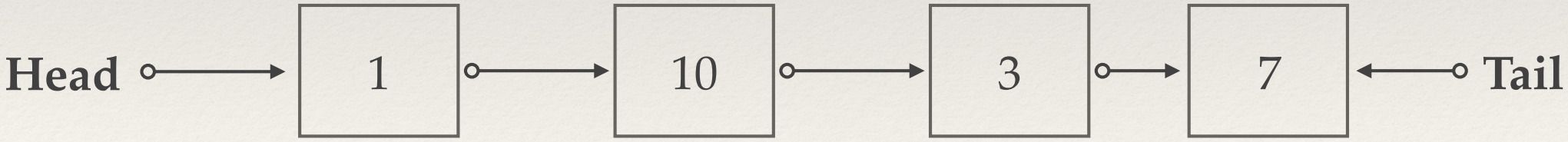
  Stacks also shrink from the head



Linked List Implementation: Queues

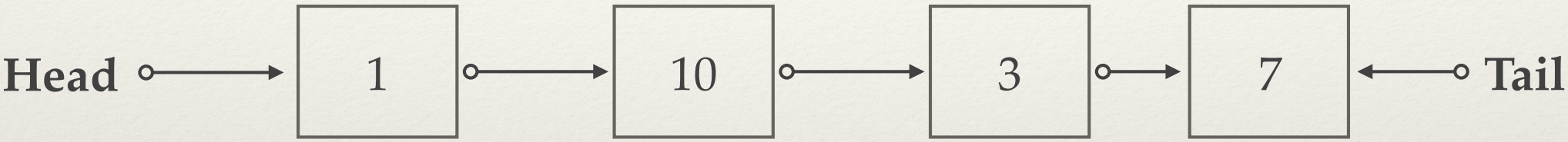
Adding to a queue (enqueuing)

- Add 7 to the queue - *append at tail*



Linked List Implementation: Queues

Removing an element from a queue (dequeuing)



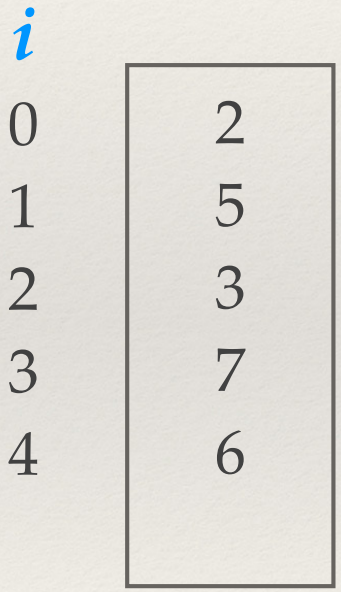
- *take a node from the head and return it*

Array Implementation

- With the array implementation
  we use array indices instead of pointers.
- i.e. the implementation is just an array of contents,
  e.g. structs, ints, *etc*.

Array Implementation: Stack

- The stack is easy to represent as an array.
- There is no defined representation, just a “stack” of values.



Head -> 6 -> 7 -> 3 -> 5 -> 2

Head = 4

Array Implementation: Stack

- Add element 4 by: writing element to index $i = 5$ and setting Head to 5.

i	
0	2
1	5
2	3
3	7
4	6
5	4

Head -> 4 -> 6 -> 7 -> 3 -> 5 -> 2

Head = 5

Array Implementation: Stack

- Remove element 4 by setting Head to 4.

i	
0	2
1	5
2	3
3	7
4	6
5	4

Head -> 4 -> 6 -> 7 -> 3 -> 5 -> 2

Head = 5

Array Implementation: Stack

- Remove element 4 by setting Head to 4.

i	
0	2
1	5
2	3
3	7
4	6
5	4

Head = 4

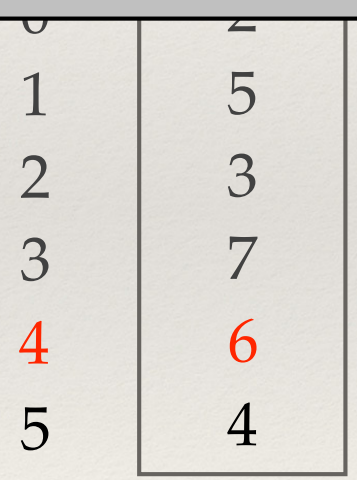
Head -> 6 -> 7 -> 3 -> 5 -> 2

Array Implementation: Stack

- Remove ele

Array vs Linked-List

Advantage:	Faster (no malloc) Less memory (no pointers)
Disadvantage:	Has a maximum size

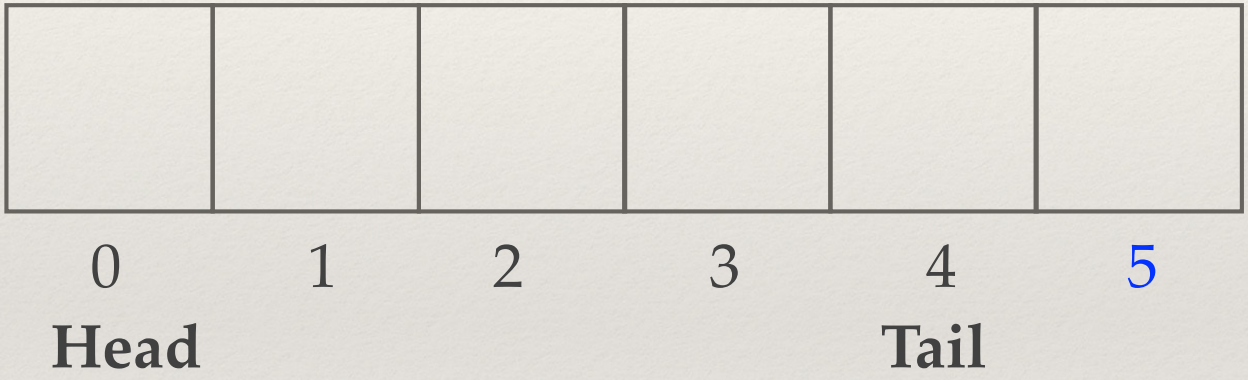


Head = 4

> 7 -> 3 -> 5 -> 2

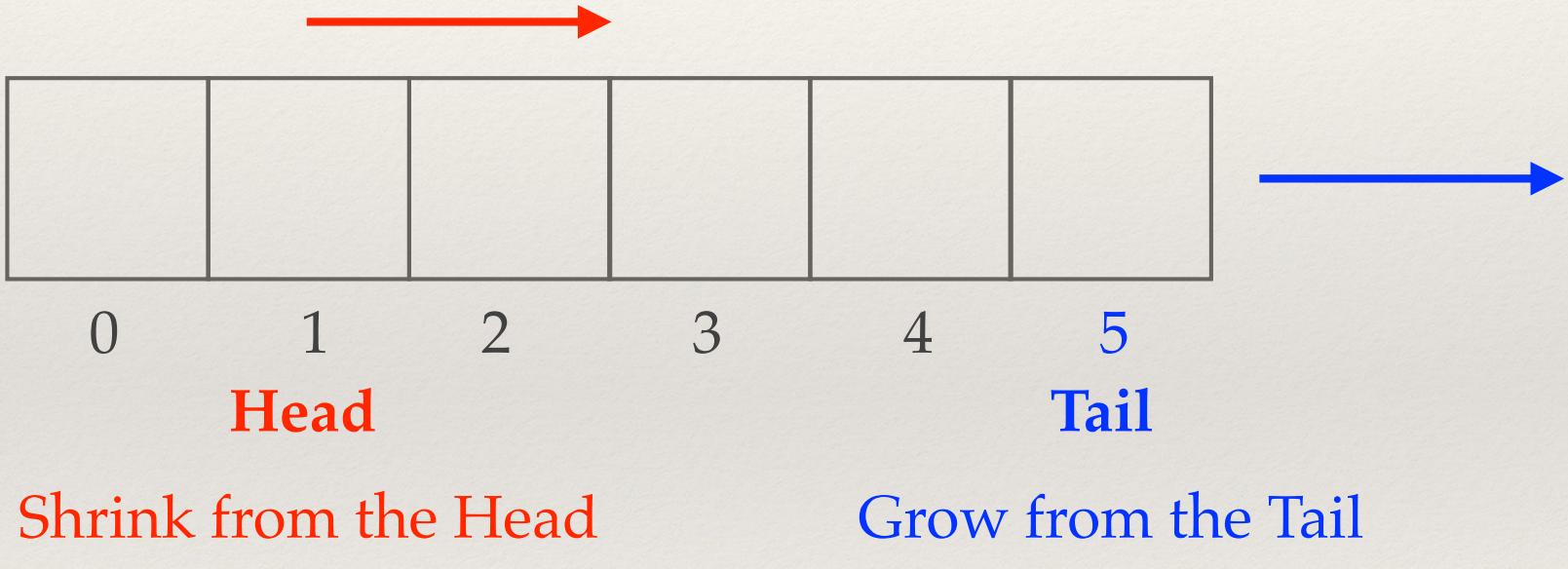
Array Implementation: Queue

- A queue is harder to implement than a stack.



Array Implementation: Queue

- A queue is harder to implement than a stack.



- It continues to grow in one direction
  and will eventually reach the end of the array.

Stack ADT

```
typedef Element double;

typedef struct {
    /* implementation dependent */
} Stack;

Stack * create_stack ( );
```

or whatever data type required

- returns a pointer to the head of an empty stack

Stack ADT

```
int push ( Stack stack, Element item );
```

- pushes the value in item onto the stack
- returns SUCCESS/FAILURE (i.e. 1/0)

```
int pop ( Stack stack, Element * item );
```

- pop value from the stack and place it in item
- returns SUCCESS/FAILURE (i.e. 1/0)

Stack ADT

```
int size ( );
```

- returns the number of elements on the stack

```
int is_empty ( );
```

- returns TRUE if the queue is empty,
- else returns FALSE

Stack ADT optional

```
int peek (Element * item );
```

- stores the top value from the stack into item without removing it
- returns SUCCESS/FAILURE (i.e. 1/0)

```
void print_stack (Stack mystack);
```

- print the elements of the stack

Queue ADT

```
typedef Element char *;

typedef struct {
    /* implementation dependent */
} Queue;

Stack * create_queue ( );
```

- returns a pointer to the head of an empty queue

Queue ADT

```
int enqueue ( Element item );
```

- adds the value in item to the end of the queue
- returns SUCCESS/FAILURE (i.e. 1/0)

```
int dequeue ( Element * item );
```

- removes the element from the front of the queue and places it in item
- returns SUCCESS/FAILURE (i.e. 1/0)

Queue ADT


```
int enqueue ( char * item );
```

- adds the value in item to the end of the queue
- returns SUCCESS/FAILURE (i.e. 1/0)

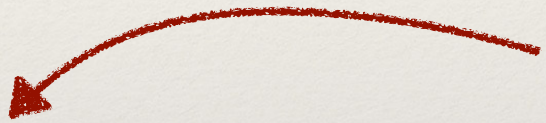
```
int dequeue ( char ** item );
```

- removes the element from the front of the queue and places it in item
- returns SUCCESS/FAILURE (i.e. 1/0)

Queue ADT

```
int enqueue ( char * item );
```

- adds the value in item to the end of the queue
- returns SUCCESS/FAILURE (i.e. 1/0)




```
int dequeue ( char ** item );
```

- removes the element from the front of the queue and places it in item
- returns SUCCESS/FAILURE (i.e. 1/0)

Queue ADT

```
int enqueue ( Element item );
```

- adds the value in item to the end of the queue
- returns SUCCESS/FAILURE (i.e. 1/0)

```
int dequeue ( Element * item );
```

- removes the element from the front of the queue and places it in item
- returns SUCCESS/FAILURE (i.e. 1/0)

Queue ADT

```
int size ( );
```

- returns the number of elements in the queue

```
int is_empty ( );
```

- returns TRUE if the queue is empty,
- else returns FALSE