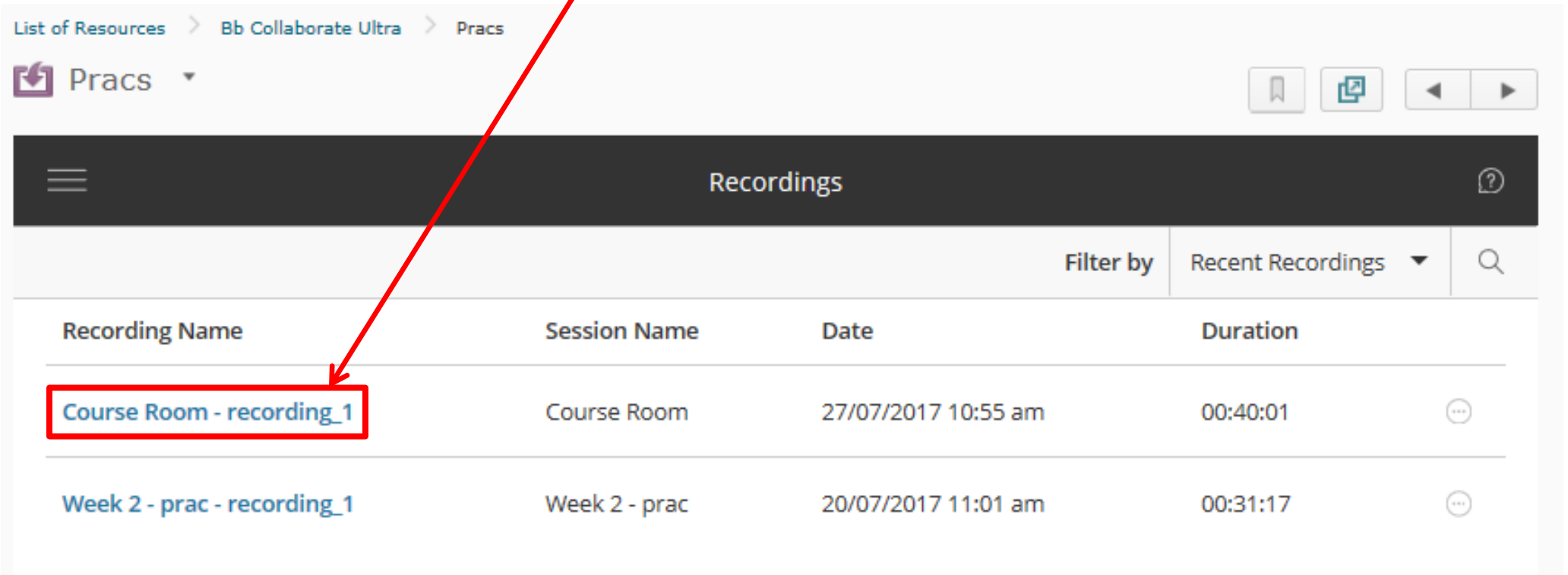
SIT221: Data Structures and Algorithms

Lecture 4: Stacks and Queues

Updates

- Prac Recordings for week 3: available on Bb

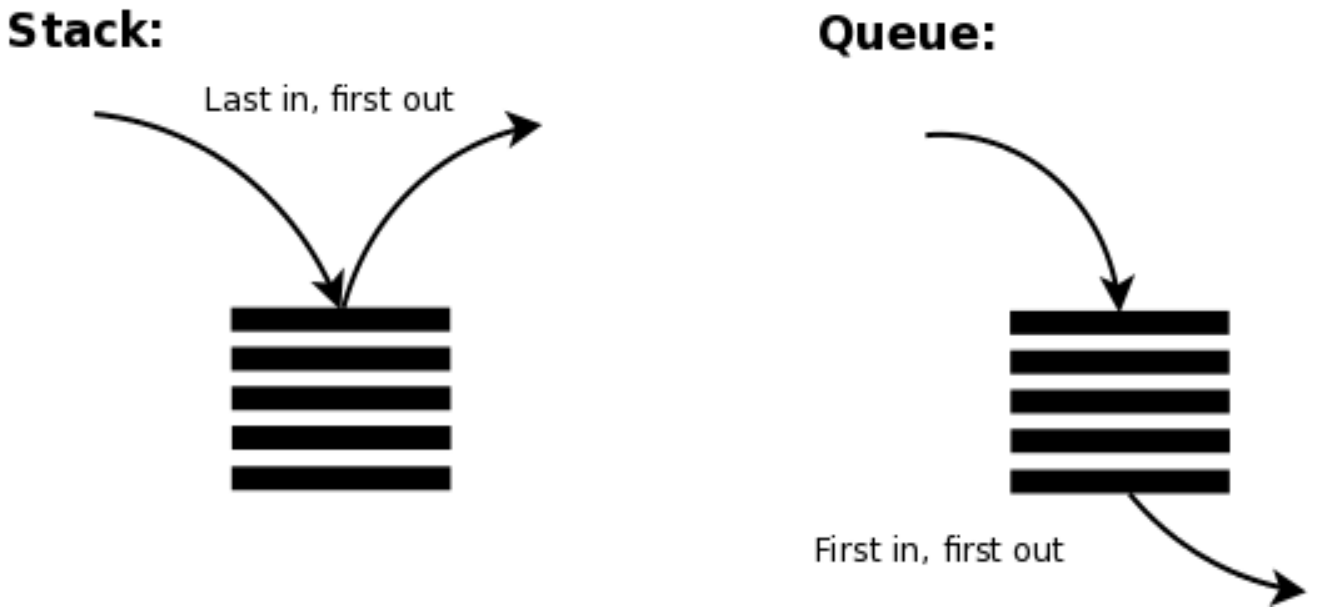


Lecture 3 Recap

- Linked Lists
  - Node/Head
  - Traversal/insert/delete/add
  - Singly/Double/Circular
  - Pros & Cons

Stacks & Queues

- Last-in First-out (LIFO) = Stack = Reverse = Backtrack
- First-in First-out (FIFO) = Queue

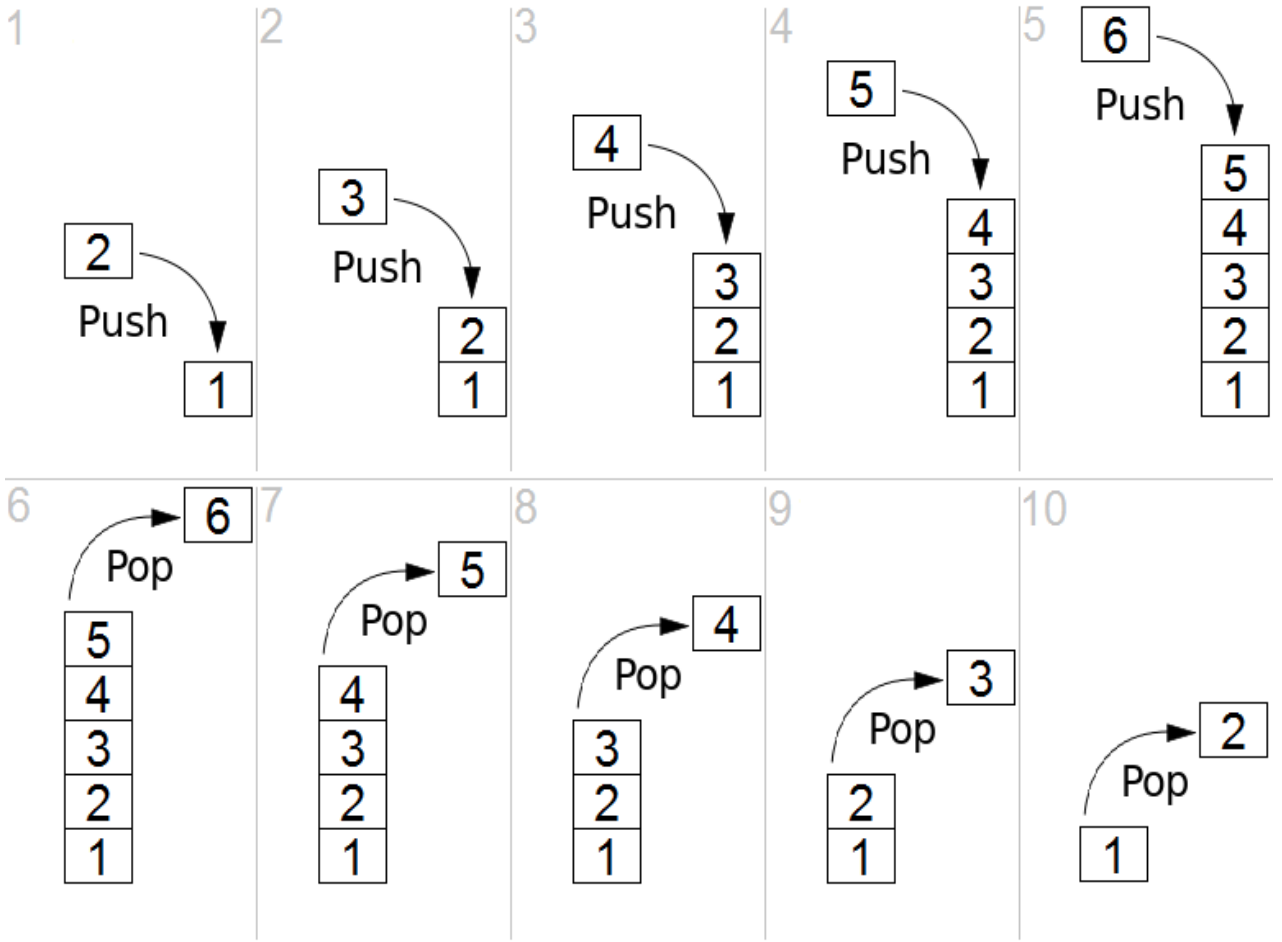


Stack data structure

- When you have items stacked on top of each other, so you can only take an element off the top of the stack.
- Only One end for both add & remove.
- key operations:
  - Push: Input/InsertAt(item, index = 0)
  - Pop: Output/RemoveAt(index = 0)
  - Peek: ElementAt(index = 0)

How stacks work?

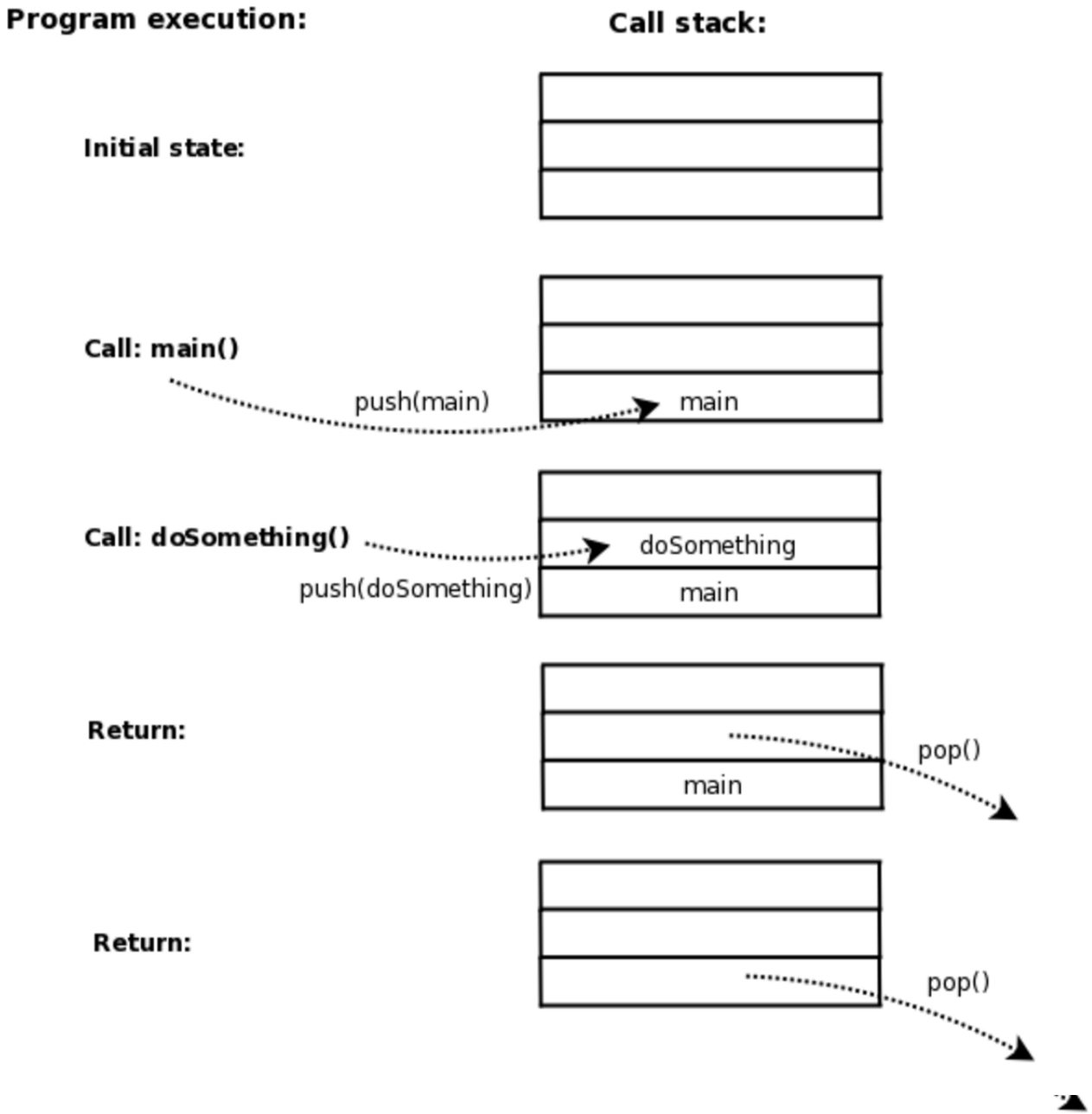
- Stack has 1
- Push(2)
- Push(3)
- Push(4)
- Push(5)
- Push(6)
- Pop() ---> ?
- Pop() --> ?



Stack example (1)

```
procedure main ()
doSomething()

procedure doSomething ()
print "whoo!"
```



Stack example (2)

- Convert 155_{10} to binary…
  - 155 / 2 = 77 Remainder 1
  - 77 / 2 = 38 Remainder 1
  - 38 / 2 = 19 Remainder 0
  - 19 / 2 = 9 Remainder 1
  - 9 / 2 = 4 Remainder 1
  - 4 / 2 = 2 Remainder 0
  - 2 / 2 = 1 Remainder 0
  - 1 / 2 = 0 Remainder 1
- Answer = 10011011

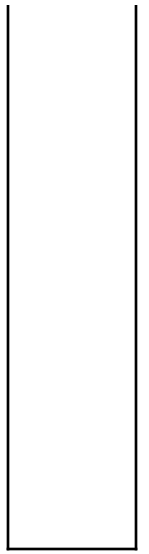
Stack example (3)

- Reverse Polish Notation (RPN)
  https://en.wikipedia.org/wiki/Reverse_Polish_notation
  - Infix notation: 3 * (4 + 5)
  - Postfix notation: 3 4 5 + *

Stack example (3)

- Infix → Postfix
  - Input: a + b * c - d

Expression: a + b * c - d



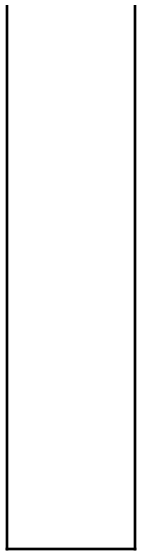
Stack

Postfix:

Stack example (3)

- Infix → Postfix

Expression: **a** + b * c - d



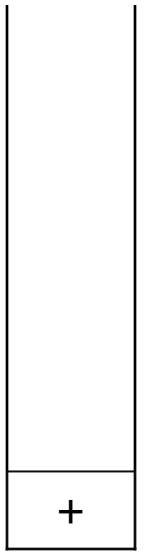
Stack

Postfix: **a**

Stack example (3)

- Infix → Postfix

Expression: a + b * c - d



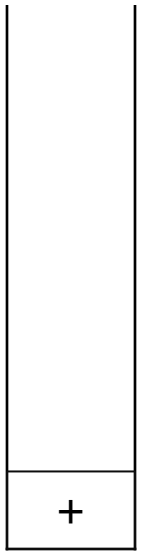
Stack

Postfix: a

Stack example (3)

- Infix → Postfix

Expression: a + **b** * c - d



Stack

Postfix: a **b**

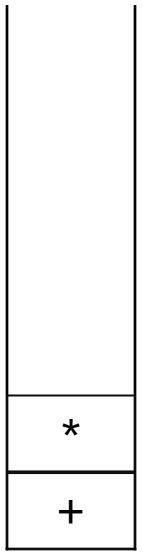
Stack example (3)

- Infix → Postfix

Expression: a + b * c - d

Postfix: a b

Priority: * > +
Thus * can be pushed in

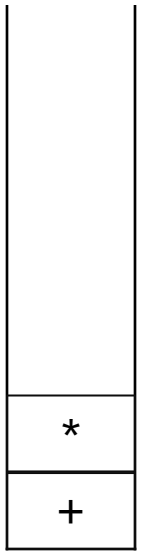


Stack

Stack example (3)

- Infix → Postfix

Expression: a + b * **c** - d



Stack

Postfix: a b **c**

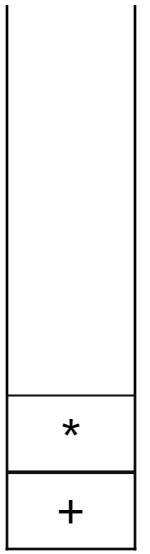
Stack example (3)

- Infix → Postfix

Expression: a + b * c - d

Postfix: a b c

Priority: - < *
Thus - cannot be pushed in,
and * needs to be popped up



Stack

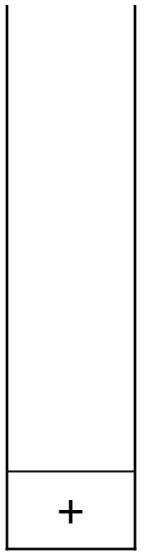
Stack example (3)

- Infix → Postfix

Expression: a + b * c - d

Postfix: a b c *

Priority: - = +
Thus - cannot be pushed in,
and + needs to be popped up



Stack

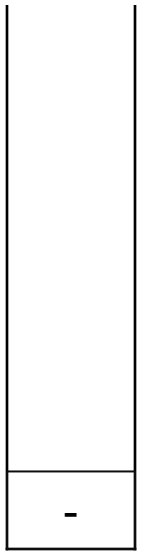
Stack example (3)

- Infix → Postfix

Expression: a + b * c **-** d

Postfix: a b c * **+**

Stack is empty and thus - can be pushed in

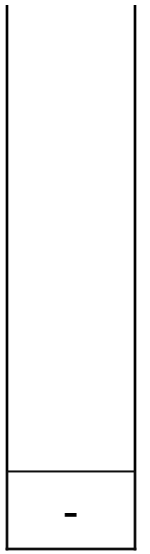


Stack

Stack example (3)

- Infix → Postfix

Expression: a + b * c - **d**



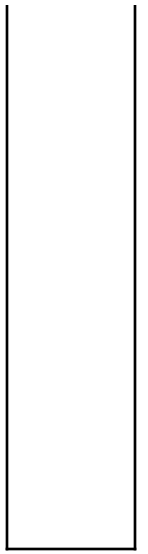
Stack

Postfix: a b c * + **d**

Stack example (3)

- Infix → Postfix

Expression: a + b * c - d



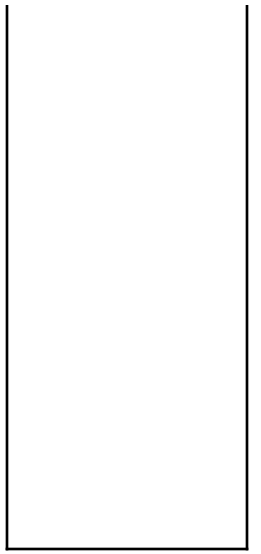
Stack

Postfix: a b c * + d -

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b c * + d -



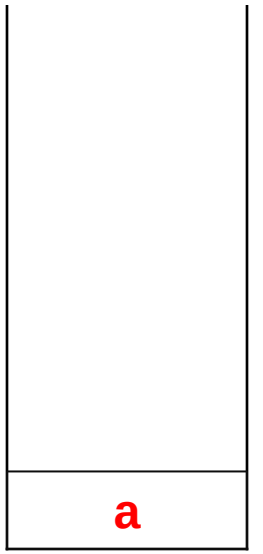
Stack

Infix:

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: **a** b c * + d -



Stack

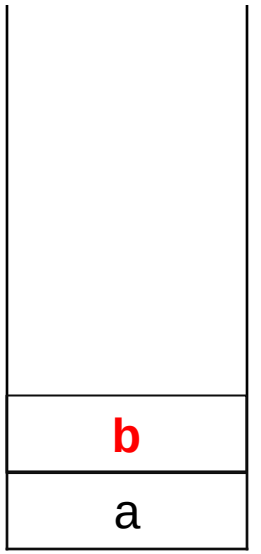
Infix:

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a **b** c * + d -

Infix:

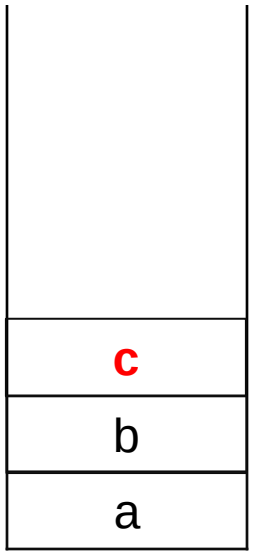


Stack

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b **c** * + d -



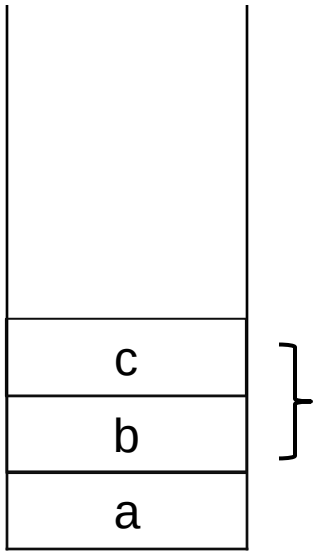
Stack

Infix:

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b c * + d -



Stack

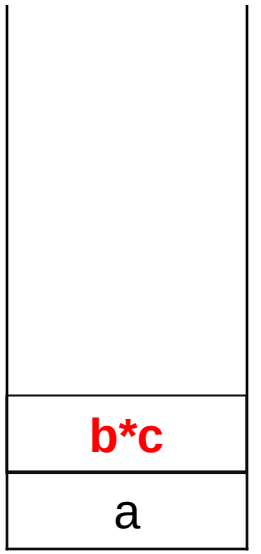
Infix:

- Popping up
- Applying *
- Pushing the result back

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b c * + d -



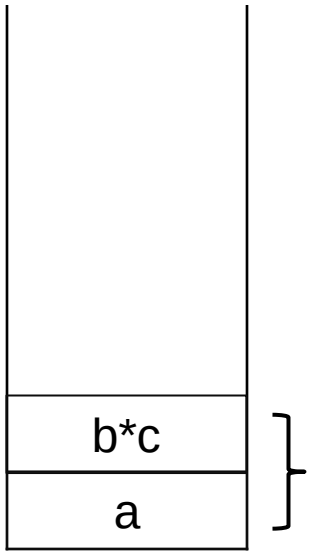
Stack

Infix: b * c

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b c * **+** d -



Stack

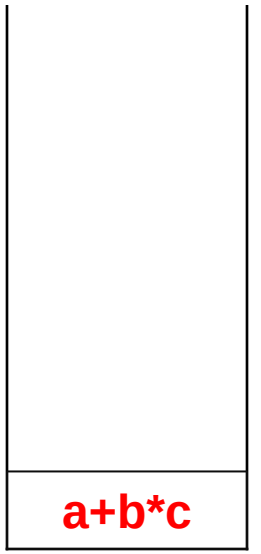
Infix: b * c

- Popping up
- Applying **+**
- Pushing the result back

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b c * **+** d -



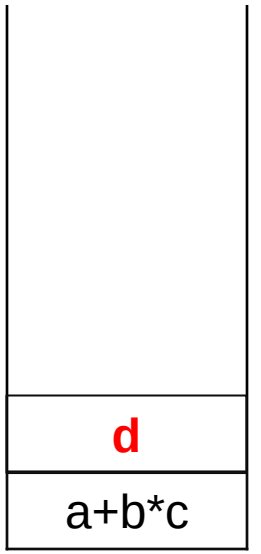
Stack

Infix: a + b * c

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b c * + **d** -



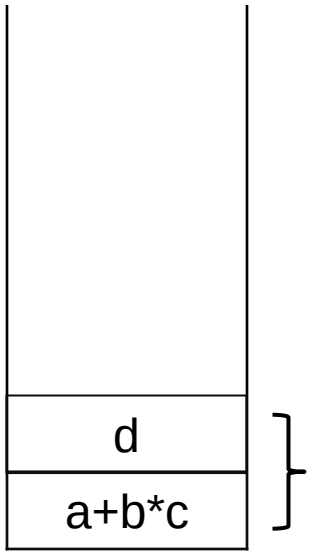
Stack

Infix: a + b * c

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b c * + d -



Stack

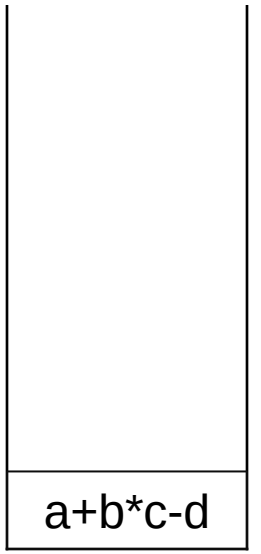
Infix: a + b * c

- Popping up
- Applying -
- Pushing the result back

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b c * + d -



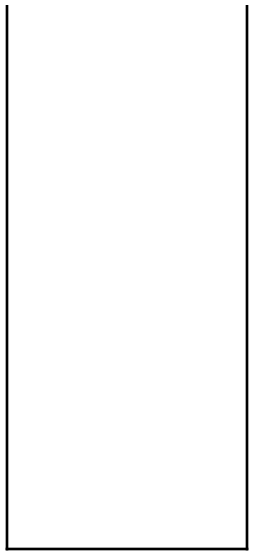
Stack

Infix: a + b * c - d

Stack example (3)

- Postfix → Infix/Calculation
  - Input: a b c * + d -

Expression: a b c * + d -



Stack

Infix: a + b * c – d
Result: (a + (b * c)) - d

More exercise

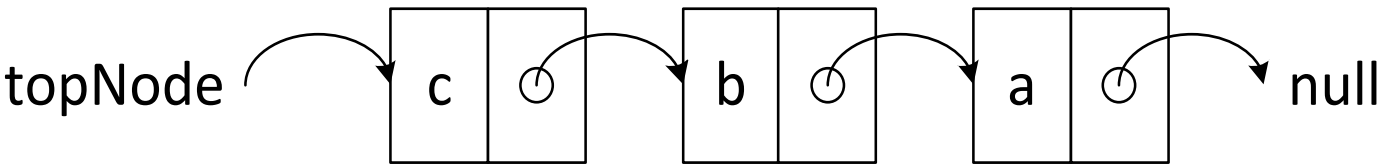
- Fibonacci number
  - F(0) = F(1) = 1
  - For n > 1, F(n) = F(n-1) + F(n-2)
- Using stack to calculate F(n) for a given n

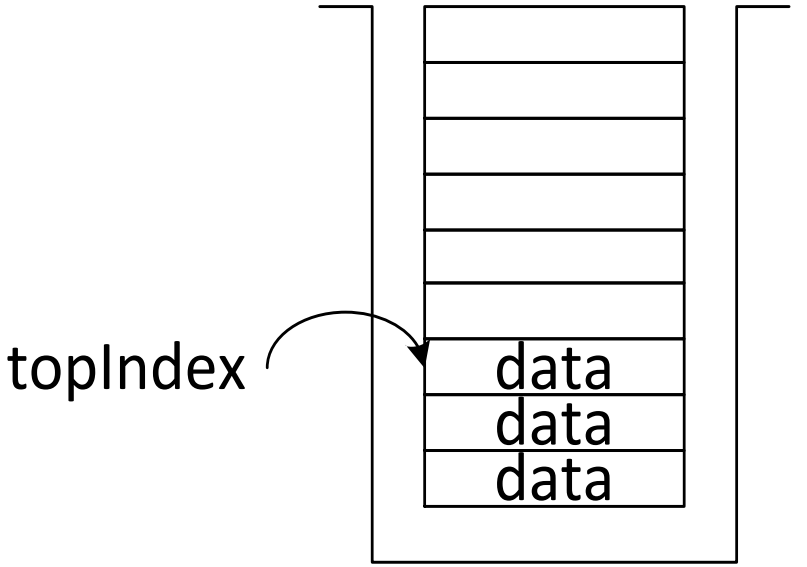
Why Stacks?

- Navigation – back button on a web browser
- Operating Systems
- Undo functionality
- Expression Evaluation
- Backtrackings & Graph Traversal
- …

How to implement a stack class?

- Using Dynamic Arrays
- Using Linked Lists





Stack class

```
Public class StackT>
{
        LinkedList<T>  data; //Can you do it with an array?
        public Stack() { data = new LinkedList<T>(); }
        public void Push(T element);
        public T Pop();
        public T Peek();
        public int Count {get;}
}
```

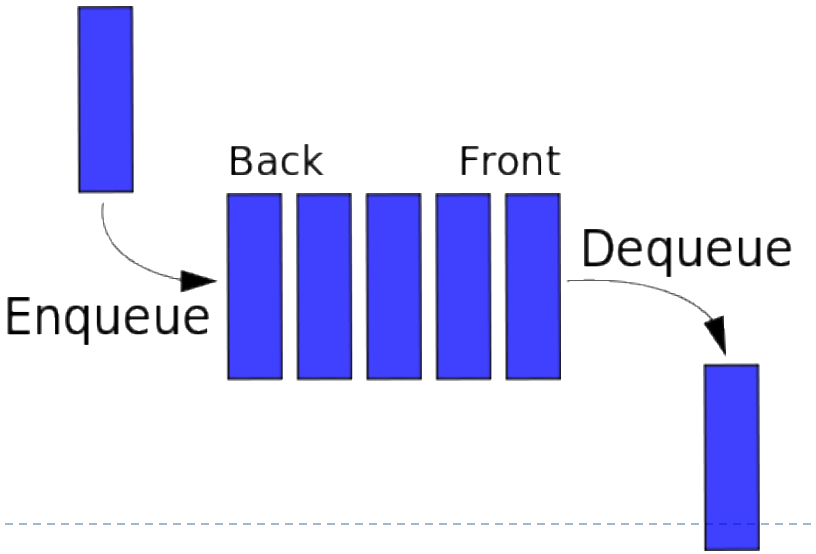
Stack class – Push/Pop/Peek

```
public T Pop() {
        if(data.Count == 0)
                return default(T);
        var item = data[0];
        data.RemoveAt(0);
        return item;
}
public void Push(T item) {
        data.Insert(T, 0);
}
```

```
public T Peek() {
        if(data.Count == 0)
                return default(T);
        var item = data[0];
        return item;
}
```

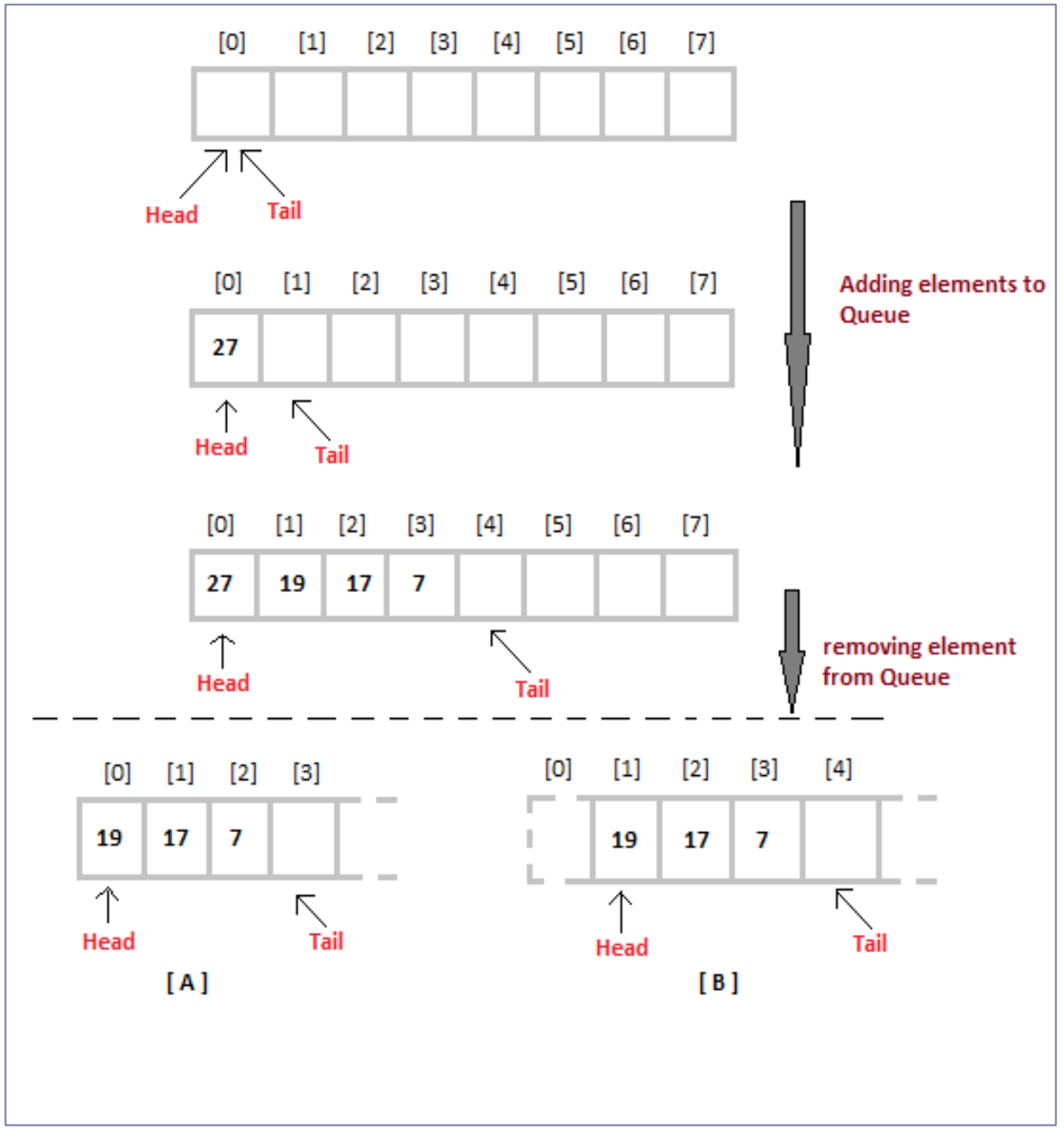
Queues

- First-in First-out (FIFO)
- Key operations:
  - Enqueue => Add (to the back, at the end)
  - Deque => Remove(from the front, at index zero)



How Queues work?

- Start with empty queue.
- Enque(27)
- Enque(19)
- Enque(17)
- Enque(7)
- Deque() --> ?
- Deque() --> ?

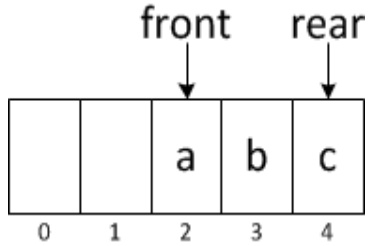


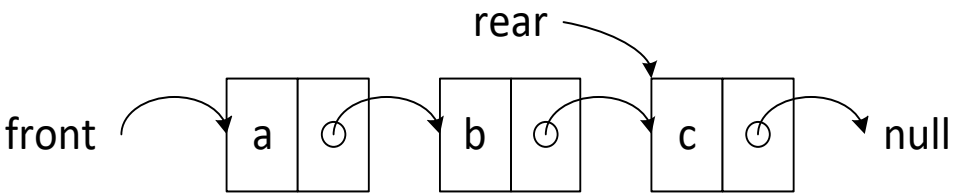
Queue error states

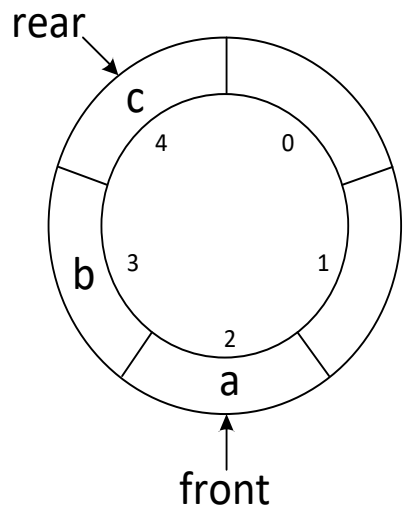
- **Underflow** – an error that occurs when you attempt to dequeue an element when the queue is empty
- **Overflow** – an error that occurs when you attempt to enqueue an element when the queue is full (implementation dependent).

How to implement a queue class?

- Dynamic array
- Linked list







Circular Queue

Queue class

```
Public class Queue<T>
{
        LinkedList<T> data;
        public Queue() { data = new LinkedList<T>(); }
        public void Enque(T element);
        public T Deque();
        public T Peek();
        public int Count {get;}
}
```

Queue – Enque/Dequeue

```
public void Enque(T element)
{
        data.Add(T);
}
public T Deque()
{
        if(data.Count == 0) return default(T);
        var item = data[0];
        data.RemoveAt(0);
        return item;
}
```

Why Queues?

- Queue of people for service at supermarket/bank/cafe
- Queue of people on a hospital waiting list
- Queue of patients to attend
- Queue to board an aeroplane/train/bus/escalator
- Queue of cars at traffic lights or other street corner
- Queue of tasks in a to-do list/assignment
- Queue of print jobs to be processed by a printer
- Queue of requests to a web server
- Queue of task scheduling in Operating Systems

Different types of queues

- Circular Queue
- Double Ended Queues = Deques [add and remove from both ends]
- Priority Queues = Sorted Queue, A queue where elements are sorted by priority