CSIT881
Programming and Data Structures

Stack and Queue

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Dr. Joseph Tonien

Objectives

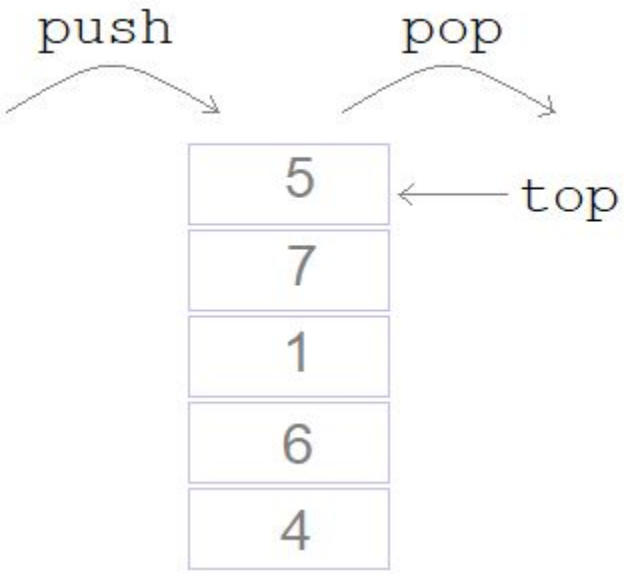
- Stack data structure
- Queue data structure
- Some problem solving with Stack and Queue

(ADT) Stack

A stack is an abstract data type with the following operations:

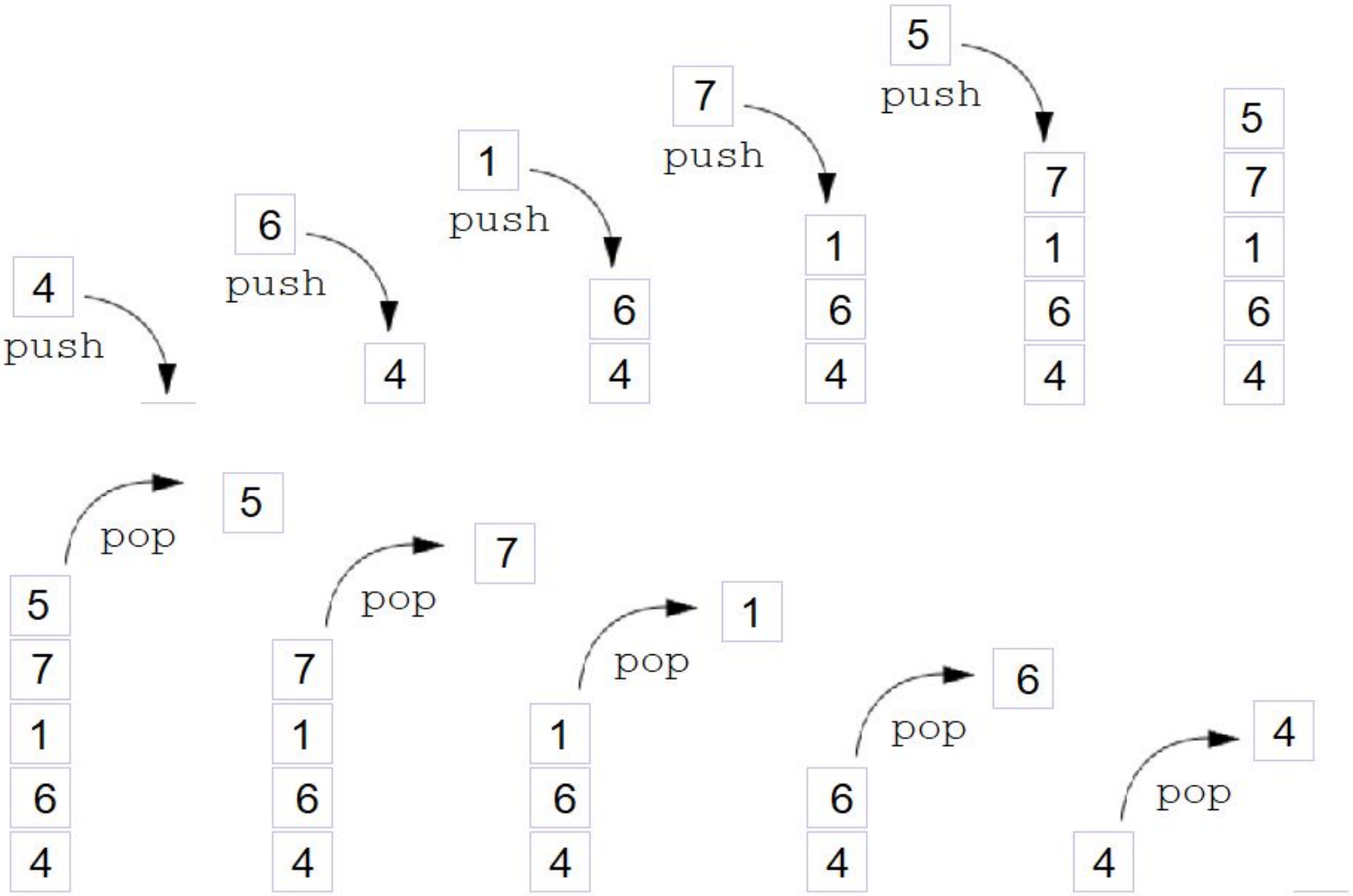
- `push(item)`: add an item onto the top of the stack;
- `pop()`: remove the item from the top of the stack and return it;
- `top()`: look at the item at the top of the stack, but do not remove it. This operation is optional because it can be achieved by `pop` the top item and then `push` it back to the stack.

(ADT) Stack



Last-In-First-Out (LIFO) structure

(ADT) Stack



Last-In-First-Out (LIFO) structure

(ADT) Stack

A stack is a Last-In-First-Out (LIFO) structure:
the last item put in is *the first item got out* of a stack.

Therefore, in a stack, only the top element is accessible.

Alternative terminology:

- `push(item) = add(item)`
- `pop() = remove()`
- `top() = peek()`

(ADT) Stack

Some disadvantage:

- only the top element is accessible;
- no access to random item in a stack;
- no looping through a stack (unless by popping all the items);
- no searching through a stack.

Some programming language may add additional functionalities to their stack implementation to have some of the above lacking behaviours.

(ADT) Stack

```
Java:
https://docs.oracle.com/javase/10/docs/api/
java/util/Stack.html

package java.util
class Stack<E>
```

```
Python:
https://docs.python.org/3/library/queue.html

from queue import LifoQueue

Push operation: put
Pop operation: get
Peek operation: not implemented
```

Stack in python

```
from queue import LifoQueue

# creating a stack
stack = LifoQueue()

# put() add item to the stack
stack.put("would")
stack.put("you")
stack.put("like")
stack.put("green")
stack.put("eggs")
stack.put("and")
stack.put("ham")

# get() remove item from stack in LIFO order
print(stack.get())
print(stack.get())
print(stack.get())
print(stack.get())
print(stack.get())
print(stack.get())
print(stack.get())
```

What is the output of this program?

Example: Parenthesis checking using Stack

In mathematics, we use different types of parenthesis, such as (,), {, }, [,], to write an expression

```
4 * {z - [(a+b) * c]}
```

Similarly, in programming, we use several types of parenthesis in our code.

```
public static void main(String[] args){
  System.out.println("Hello");
}
```

We can use Stack to check the validity of these expressions and codes, to make sure every open parentheses matches with a closed parentheses.

Example: Parenthesis checking using Stack

pseudocode

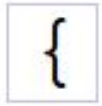
```
INPUT: A math expression, or a programming code
OUTPUT: Returns true for valid parenthesis
        Returns false for invalid parenthesis

Initialize an empty stack
FOR each character c of the input
    IF c is an open symbol
        Push c into the stack
    ELSE IF c is a closed symbol
        x = Pop the stack
        IF x is not the open symbol matching with c
            RETURN false
END FOR

IF the stack is not empty
    RETURN false
ELSE
    RETURN true
```

Example: Parenthesis checking using Stack

```
4 * {z - [(a+b) * c]}
    ^
```

encounter open symbol

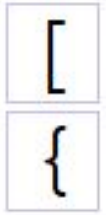
push it in Stack




```
4 * {z - [(a+b) * c]}
        ^
```

encounter open symbol

push it in Stack




```
4 * {z - [(a+b) * c]}
          ^
```

encounter open symbol

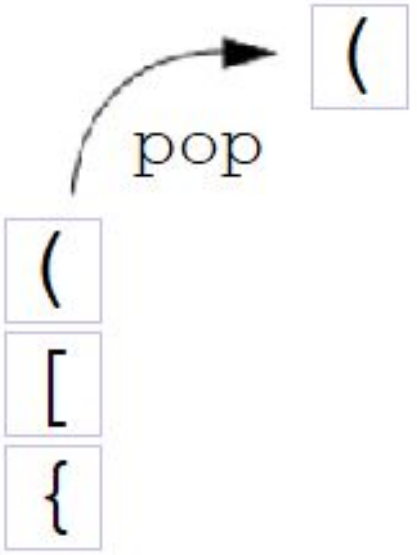
push it in Stack

Example: Parenthesis checking using Stack

```
4 * {z - [(a+b) * c]}
```

^ encounter closed symbol

pop the Stack and compare



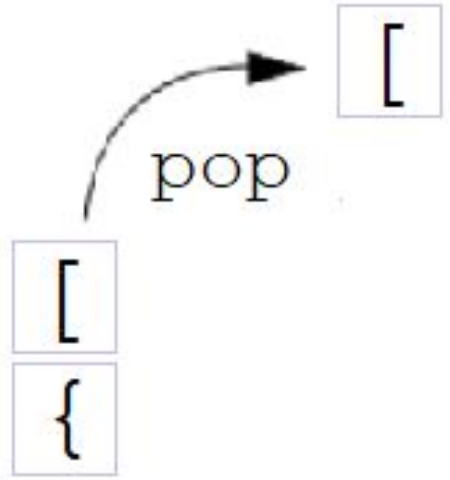
Yes it matches!

Example: Parenthesis checking using Stack

```
4 * {z - [(a+b) * c]}
                   ^
```

encounter closed symbol

pop the Stack and compare



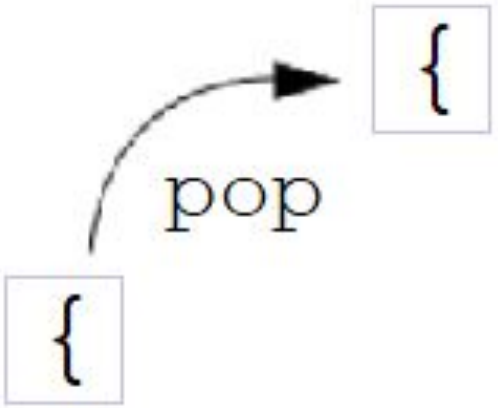
Yes it matches!

Example: Parenthesis checking using Stack

```
4 * {z - [(a+b) * c]}
                    ^
```

encounter closed symbol

pop the Stack and compare



Yes it matches!

The stack is empty.

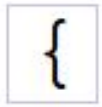
Checking parenthesis DONE!

Example: Parenthesis checking using Stack

```
y + {w * [(2+k)) - z]}
    ^
```

encounter open symbol

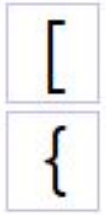
push it in Stack




```
y + {w * [(2+k)) - z]}
        ^
```

encounter open symbol

push it in Stack




```
y + {w * [(2+k)) - z]}
          ^
```

encounter open symbol

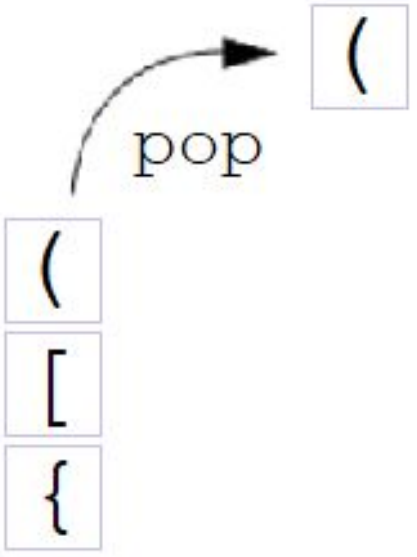
push it in Stack

Example: Parenthesis checking using Stack

```
y + {w * [(2+k)) - z]}
          ^
```

encounter closed symbol

pop the Stack and compare



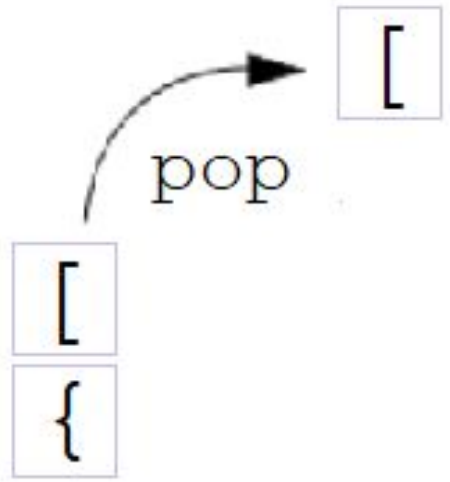
Yes it matches!

Example: Parenthesis checking using Stack

```
y + {w * [(2+k)) - z]}
```

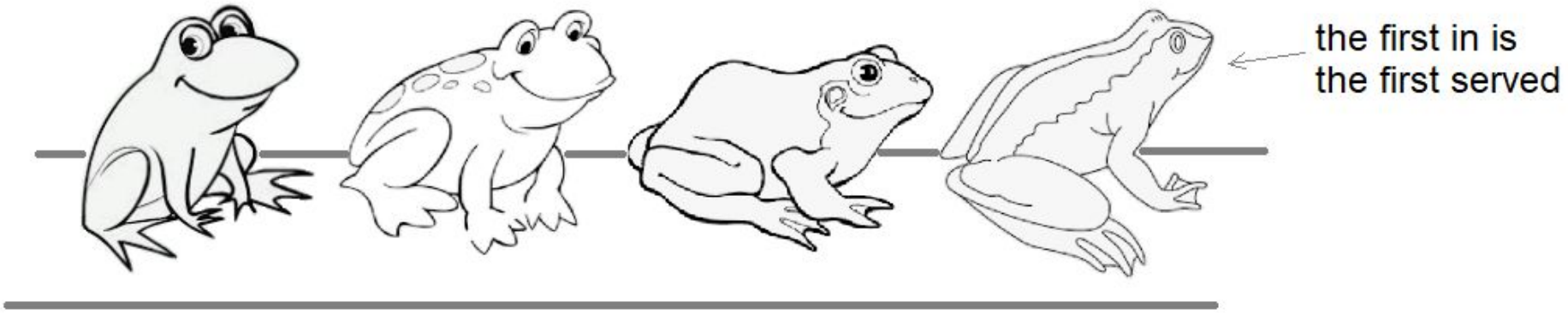
^ encounter closed symbol

pop the Stack and compare



No, it does NOT match 😢

(ADT) Queue



(ADT) Queue

A queue is an abstract data type with the following operations:

- `enqueue(item)`: add an item to the back of the queue;
- `dequeue()`: remove the item from the front of the queue and return it;
- `front()`: look at the item at the front of the queue, but do not remove it.

A queue is a *first-in-first-out* (FIFO) structure:

the first item put in is *the first item got out* of a queue.

(ADT) Queue

Some disadvantage:

- only the front element is accessible;
- no access to random item in a queue;
- no looping through a queue (unless by dequeuing all the items);
- no searching through a queue.

Some programming language may add additional functionalities to their queue implementation to have some of the above lacking behaviours.

(ADT) Queue

```
Java:
https://docs.oracle.com/javase/10/docs/api/
java/util/Queue.html

package java.util
interface Queue<E>
```

```
Python:
https://docs.python.org/3/library/queue.html

from queue import Queue

enqueue operation: put
dequeue operation: get
front operation: not implemented
```

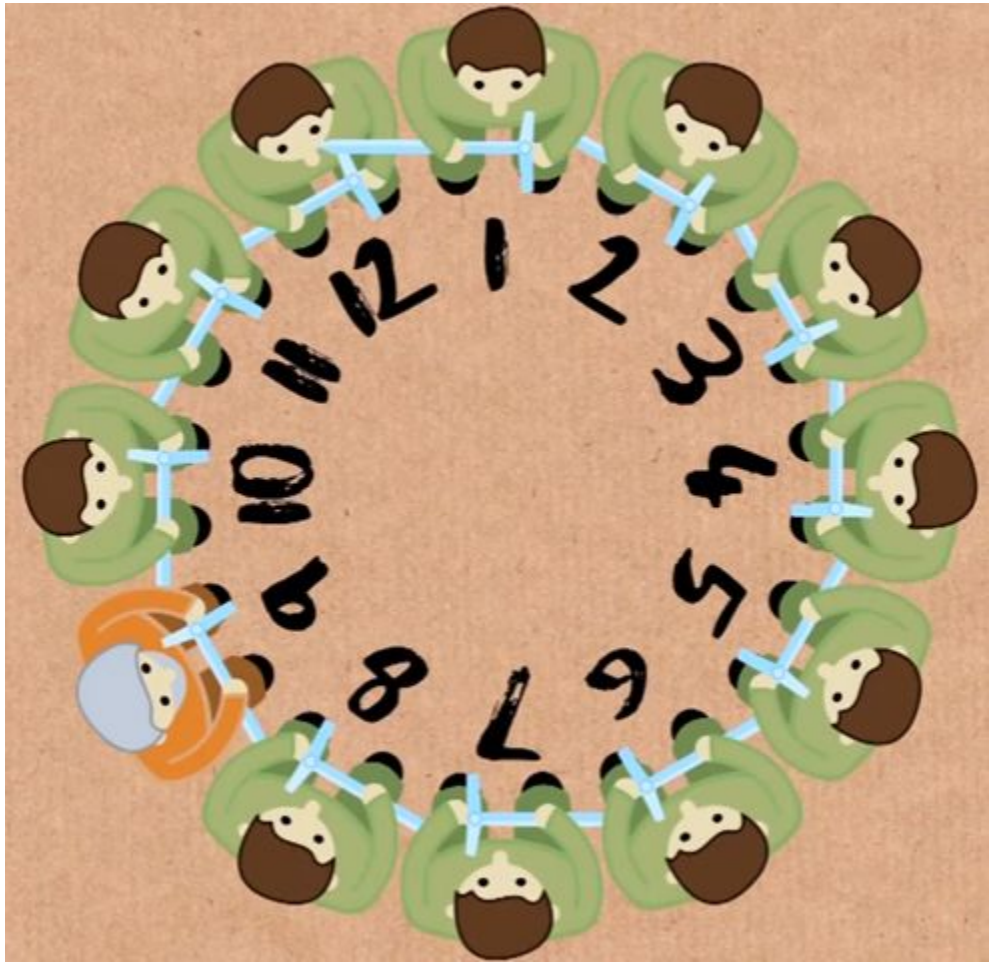
Queue in Python

```
from queue import Queue

# creating a queue
queue = Queue()

# put() add item to the queue
queue.put("would")
queue.put("you")
queue.put("like")
queue.put("green")
queue.put("eggs")
queue.put("and")
queue.put("ham")

# get() remove item from queue in FIFO order
print(queue.get())
print(queue.get())
print(queue.get())
print(queue.get())
print(queue.get())
print(queue.get())
print(queue.get())
```

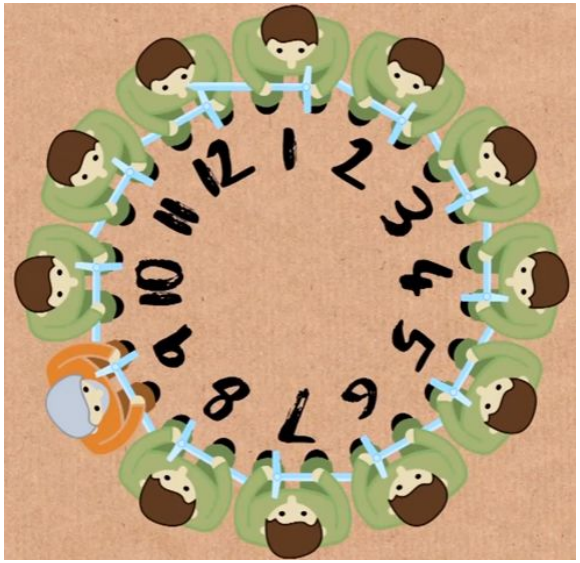
What is the output of this program?

The Josephus Problem

https://www.youtube.com/watch?v=uCsD3ZGzMgE



The Josephus Problem



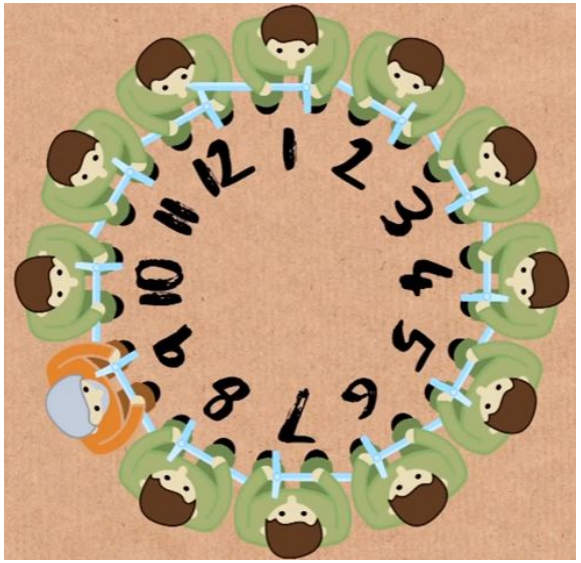
Use a Queue to represent the current status

This is the initial status

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

and it is number 1's turn

The Josephus Problem



This is the initial status

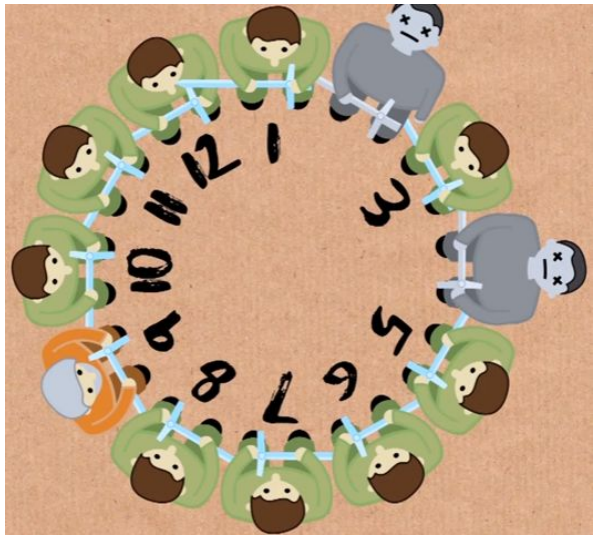
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

After 1 killed 2, the status is:

```
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1]
```

and it is number 3's turn

The Josephus Problem



[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

After 1 killed 2

[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1]

After 3 killed 4

[5, 6, 7, 8, 9, 10, 11, 12, 1, 3]

[7, 8, 9, 10, 11, 12, 1, 3, 5]

[9, 10, 11, 12, 1, 3, 5, 7]

[11, 12, 1, 3, 5, 7, 9]

[1, 3, 5, 7, 9, 11]

[5, 7, 9, 11, 1]

[9, 11, 1, 5]

[1, 5, 9]

[9, 1]

[9] ← *winning seat*

The Josephus Problem

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

After 1 killed 2

```
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1]
```

person at the front of the queue has the turn to kill the next person

Here is the algorithm to update the queue at each step:

pseudocode

```
person_get_turn = queue.dequeue()
person_get_killed = queue.dequeue()

IF queue is empty
    person_get_turn is the last person

// append person_get_turn to the end of the Queue
queue.enqueue(person_get_turn)
```

The Josephus Problem

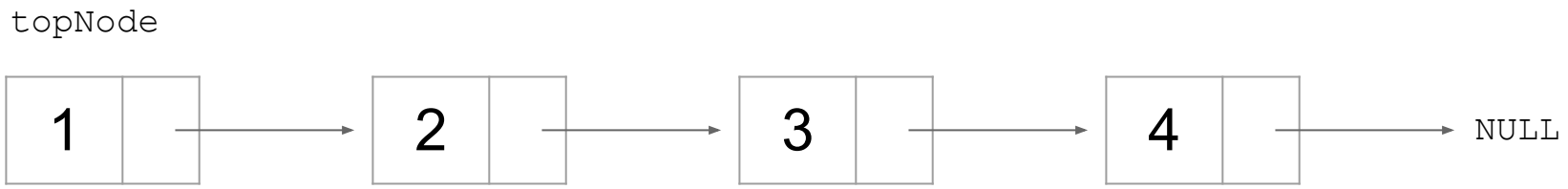
Python implementation

```
from queue import Queue

count = int(input("Enter person count: "))
# initial the queue to hold [1, 2, 3, ..., count]
queue = Queue()
for i in range(1, count+1):
   queue.put(i)

while True:
    # person at the front of the queue has the turn to kill the next person
    person_get_turn = queue.get()
    person_get_killed = queue.get()

    print("{0} kills {1}".format(person_get_turn, person_get_killed))

    if queue.empty():
        print("{0} is the last person".format(person_get_turn))
        break

    # append person_get_turn to the end of the Queue
    queue.put(person_get_turn)
```

Implementation of Stack and Queue

based on Linked List

Implementation of Stack



We will implement a Stack as a linked list that connect

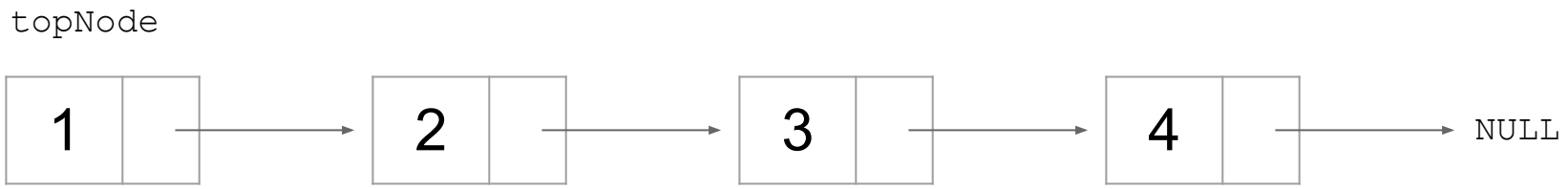
topNode -> node -> node -> …. -> NULL

pseudocode

```
record Node
{
    datum           // the data being stored in the node
    Node next       // a reference to the next node
}
```

```
record Stack
{
    Node topNode  // top node of the stack (null for empty stack)
}
```

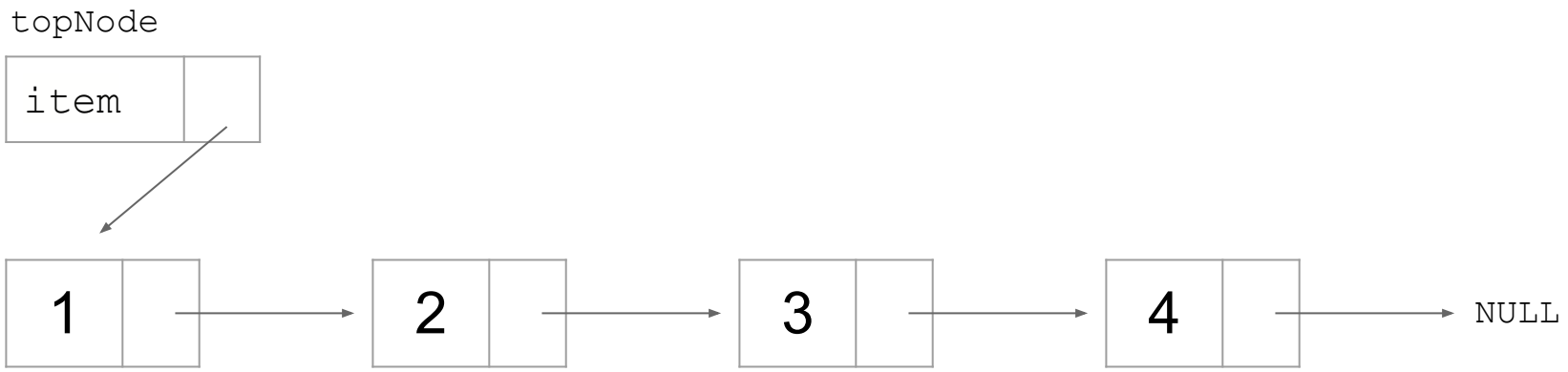
Implementation of Stack



pseudocode

```
Function constructor()
// construct an empty stack
{
    topNode = NULL
}

Function top()
// returns the item stored at the top of the stack
// but do not remove it. If the stack is empty then returns NULL
{
    IF topNode is NULL
        RETURN NULL
    ELSE
        RETURN topNode.datum
}
```

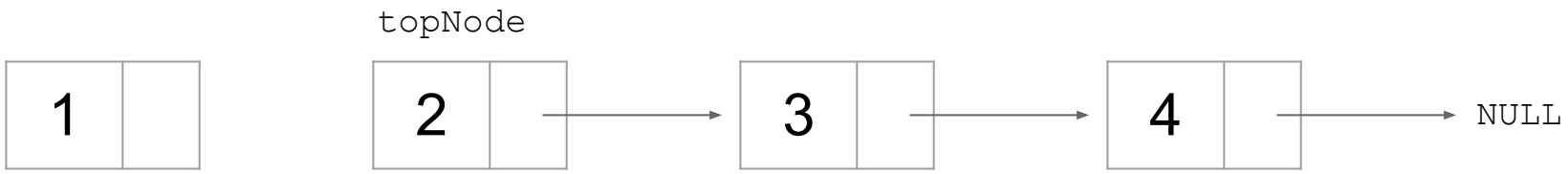
Implementation of Stack



pseudocode

```
Function push(item)
// adds an item to the top of the stack
{
    // create a new node to be put at the top of the stack
    newNode.datum = item
    newNode.next = NULL

    IF topNode is NULL
        topNode = newNode
    ELSE
        newNode.next = topNode
        topNode = newNode
}
```

Implementation of Stack



pseudocode

```
Function pop()
// Removes the item stored at the top of the stack and returns it.
// If the stack is empty then returns NULL.
{
    IF topNode is NULL
        RETURN NULL

    // get the item stored at the top node
    item = topNode.datum

    // reset the top node of this stack
    topNode = topNode.next

    RETURN item
}
```

Implementation of Stack

Python implementation

```
class Node:
#{
    """
    Representing a node consisting of
    - datum: the datum stored at the node
    - next: reference to the next node
    """

    def __init__(self, datum, next):
    #{
        self.datum = datum
        self.next = next
    #}

#}
```

Implementation of Stack

Python implementation

```
class MyStack:
#{
    """
    Implementation of a LIFO Stack as a linked list
    that connect topNode -> node -> node -> … -> NULL
    """

    def __init__(self):
    #{
        """
        Constructs an empty stack
        """
        self.topNode = None
    #}
```

Implementation of Stack

Python implementation

```
class MyStack:
    ...
    def top(self):
    #{
        """
        Returns the item stored at the top of the stack,
        but do not remove it.
        If the stack is empty then returns None.
        """
        if self.topNode == None:
            return None

        return self.topNode.datum
    #}
```

Implementation of Stack

Python implementation

```
class MyStack:
    ...
    def push(self, item):
    #{
        """
        Adds an item to the top of the stack
        """

        # create a new node to be put at the top of the stack
        newNode = Node(datum=item, next= None)

        if self.topNode == None:
            # stack is empty
            self.topNode = newNode
        else:
            newNode.next =  self.topNode
            self.topNode = newNode
    #}
```

Implementation of Stack

Python implementation

```
class MyStack:
    ...
    def pop(self):
    #{
        """
        Removes the item stored at the top of the stack and returns it.
        If the stack is empty then returns None.
        """
        # if the stack is empty
        if (self.topNode == None):
            return None

        # get the item stored at the top node
        item = self.topNode.datum

        # reset the top node of this stack
        self.topNode = self.topNode.next

        return item
    #}
```

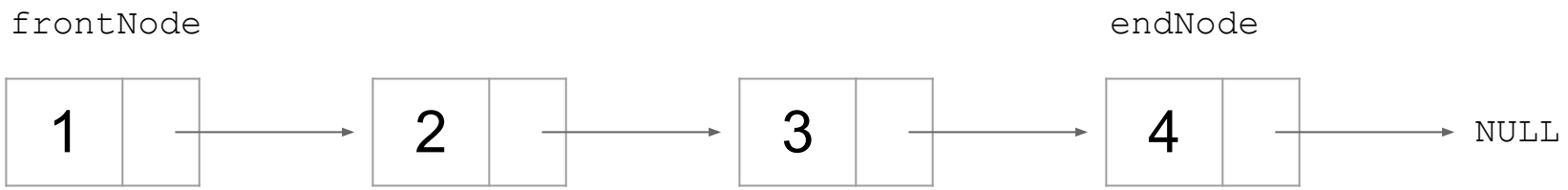
Implementation of Stack

Python implementation

```
# testing stack
stackObj = MyStack()

# push() add item to the stack
stackObj.push("dog")
stackObj.push("cat")
stackObj.push("frog")

# top() get top item of the stack, but do not remove it
print("using top():")
print(stackObj.top())
print(stackObj.top())
print(stackObj.top())

# pop() remove item from stack in LIFO order
print("using pop():")
print(stackObj.pop())
print(stackObj.pop())
print(stackObj.pop())
print(stackObj.pop())
```

```
using top():
frog
frog
frog

using pop():
frog
cat
dog
None
```

Implementation of Queue



We will implement a Queue as a linked list that connect

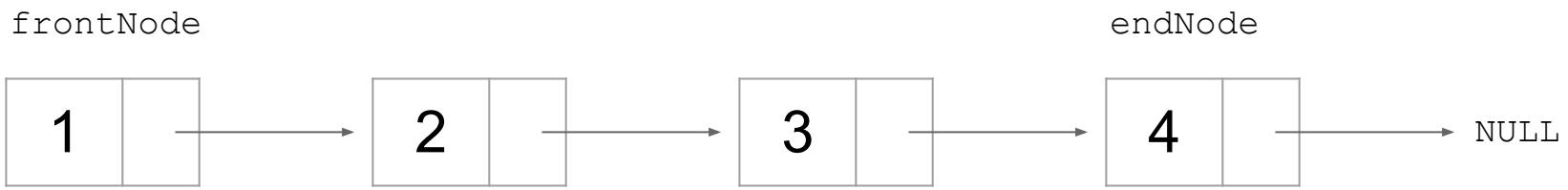
frontNode -> node -> node -> …. -> endNode -> NULL

pseudocode

```
record Node
{
    datum           // the data being stored in the node
    Node next       // a reference to the next node
}
```

```
record Queue
{
    Node frontNode //front node of the queue (null for empty queue)
    Node endNode   //end node of the queue (null for empty queue)
}
```

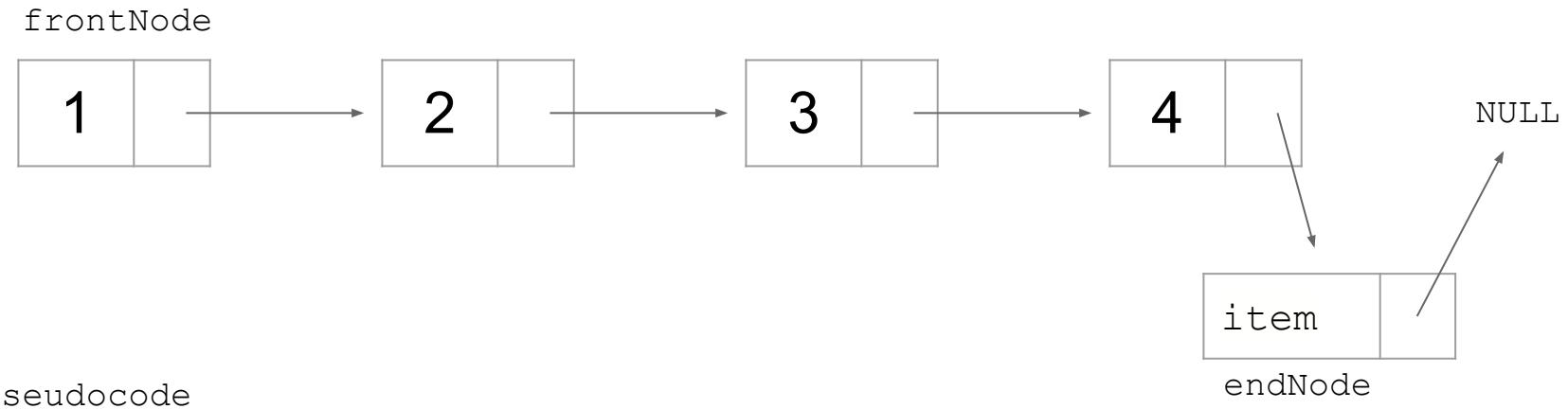
Implementation of Queue



pseudocode

```
Function constructor()
// construct an empty queue
{
    frontNode = NULL
    endNode = NULL
}

Function front()
// returns the item stored at the front of the queue
// but do not remove it. If the queue is empty then returns NULL
{
    IF frontNode is NULL
        RETURN NULL
    ELSE
        RETURN frontNode.datum
}
```

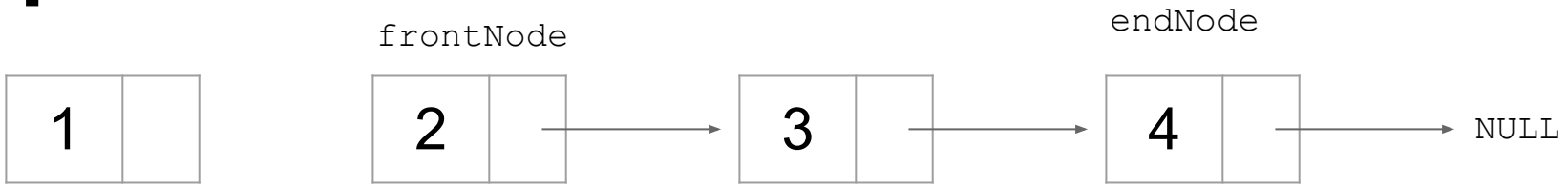
Implementation of Queue



pseudocode

```
Function enqueue(item)
// adds an item to the end of the queue
{
    // create a new node to be put at the end of the queue
    newNode.datum = item
    newNode.next = NULL

    IF frontNode is NULL
        frontNode = newNode
        endNode = newNode
    ELSE
        endNode.next = newNode
        endNode = newNode
}
```

Implementation of Queue



pseudocode

```
Function dequeue()
//Removes the item stored at the front of the queue and returns it
//If the queue is empty then returns NULL.
{
    IF frontNode is NULL
        RETURN NULL

    // get the item stored at the front node
    item = frontNode.datum

    // reset the front node of this queue
    frontNode = frontNode.next

    // if the queue becomes empty
    IF frontNode is NULL
        endNode = NULL

    RETURN item
}
```

Implementation of Queue

Python implementation

```
class Node:
#{
    """
    Representing a node consisting of
    - datum: the datum stored at the node
    - next: reference to the next node
    """

    def __init__(self, datum, next):
    #{
        self.datum = datum
        self.next = next
    #}

#}
```

Implementation of Queue

Python implementation

```
class MyQueue:
#{
    """
    Implementation of a FIFO Queue as a linked list
    that connect frontNode -> node -> node -> ... -> endNode -> NULL
    """

    def __init__(self):
    #{
        """
        Constructs an empty queue
        """
        self.frontNode = None
        self.endNode = None
    #}
```

Implementation of Queue

Python implementation

```
class MyQueue:
    ...

    def front(self):
    #{
        """
        Returns the item stored at the front of the queue,
        but do not remove it.
        If the queue is empty then returns None.
        """
        if self.frontNode == None:
            return None

        return self.frontNode.datum
    #}
```

Implementation of Queue

Python implementation

```
class MyQueue:
    ...
    def enqueue(self, item):
    #{
        """
        Adds an item to the end of the queue
        """
        # create a new node to be put at the end of the queue
        newNode = Node(datum=item, next= None)

        if self.frontNode == None:
            # queue is empty
            self.frontNode = newNode
            self.endNode = newNode
        else:
            self.endNode.next = newNode
            self.endNode = newNode
    #}
```

Implementation of Queue

```
class MyQueue:
    ...
    def dequeue(self):
    #{
        """
        Removes the item stored at the front of the queue and return it.
        If the queue is empty then returns None.
        """
        # if the queue is empty
        if (self.frontNode == None):
            return None

        # get the item stored at the front node
        item = self.frontNode.datum

        # reset the front node of this queue
        self.frontNode = self.frontNode.next

        # if the queue becomes empty
        if (self.frontNode == None):
            self.endNode = None

        return item
    #}
```

Implementation of Queue

```
# testing queue
queueObj = MyQueue()

# enqueue() add item to the queue
queueObj.enqueue("dog")
queueObj.enqueue("cat")
queueObj.enqueue("frog")

# front() get front item of the queue, but do not
remove it
print("using front():")
print(queueObj.front())
print(queueObj.front())
print(queueObj.front())

# dequeue() remove item from queue in FIFO order
print("using dequeue():")
print(queueObj.dequeue())
print(queueObj.dequeue())
print(queueObj.dequeue())
print(queueObj.dequeue())
```

```
using front():
dog
dog
dog

using dequeue():
dog
cat
frog
None
```

References

- Python 3 documentation
  https://docs.python.org/3/

- NumPy Reference
  https://numpy.org/doc/stable/reference/