Computing Foundations of Data Science

Arrays, Linked Lists, Stacks, and Queues

Lecture 9

Hyung-Sin Kim

SNU Graduate School of Data Science

Review

- Time Complexity
- Big O
- Merge Sort
- Recursion

Contents

- Data structure
- Array
- Linked list
  - Single Linked list
  - Sentinel
- Queue
- Stack

Data Structure

Computing Foundations of Data Science

Data Structures So Far

- Various data structures with different characteristics

Collection	Mutable?	Ordered?	Use When…
list	Yes	Yes	You want to keep track of an ordered sequence that you want update
tuple	No	Yes	You want to build an ordered sequence that you know won’t change or that you want to use as a key in a dictionary or as a value in a set
set	Yes	No	You want to keep track of values, but order doesn’t matter, and you don’t want duplicates. The values must be immutable.
dictionary	Yes	No	You want to keep a mapping of keys to values. The keys must be immutable.

Data Structures So Far

- Each data structure has its methods for our convenience, which we have used their methods without knowing how they are implemented
    - Append
    - Pop
    - Insert
    - Remove
    - Get
    - Size

From now, let’s dive into their implementation and learn more data structures!

Array

Computing Foundations of Data Science

Looking into Lists – Arrays

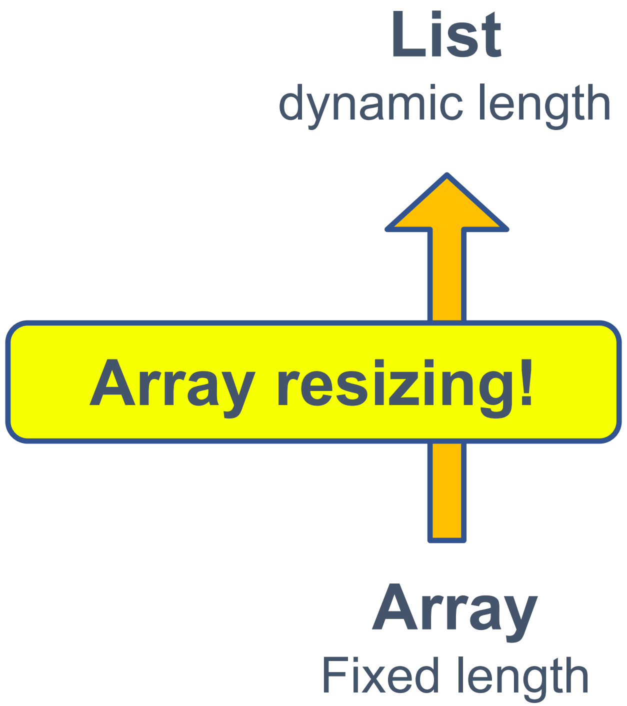
- We need to declare memory boxes for storing information
  - Ex.) “A = 1” declares a memory box to store an integer object A

- An **array** is an object comprising a **numbered sequence** of memory boxes
  - This is a more fundamental data structure (no method at all) that Python lists are built on
  - This is why we can easily access the i-th element of list A by using **A[i]**

- An array comprises
  - **Fixed** integer length (N) – should be set when initializing it
  - A **sequence** of N memory boxes (numbered 0 through N-1)

Wait… **Fixed** length?

We have inserted, appended, popped, and removed freely using lists! Its length must be **dynamic**!



List

dynamic length

Array resizing!

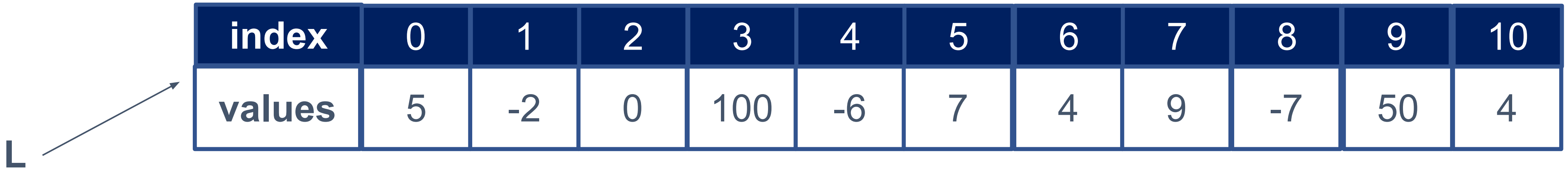
Array

Fixed length

Array Resizing

- Two problems of an array due to its fixed length
  - Memory wastage: If it contains only n(<<N) valid elements
  - Memory shortage: If it wants to contain more than N elements

- Array resizing: create another larger array and copy all the elements
  - L.append(3) when the current array is **full**



index	0	1	2	3	4	5	6	7	8	9	10
values	5	-2	0	100	-6	7	4	9	-7	50	4

Array Resizing

- Two problems of an array due to its fixed length
  - Memory wastage: If it contains only n(<<N) valid elements
  - Memory shortage: If it wants to contain more than N elements

- Array resizing: create another larger array and copy all the elements
  - L.append(3) when the current array is **full**

L

index	0	1	2	3	4	5	6	7	8	9	10
values	5	-2	0	100	-6	7	4	9	-7	50	4

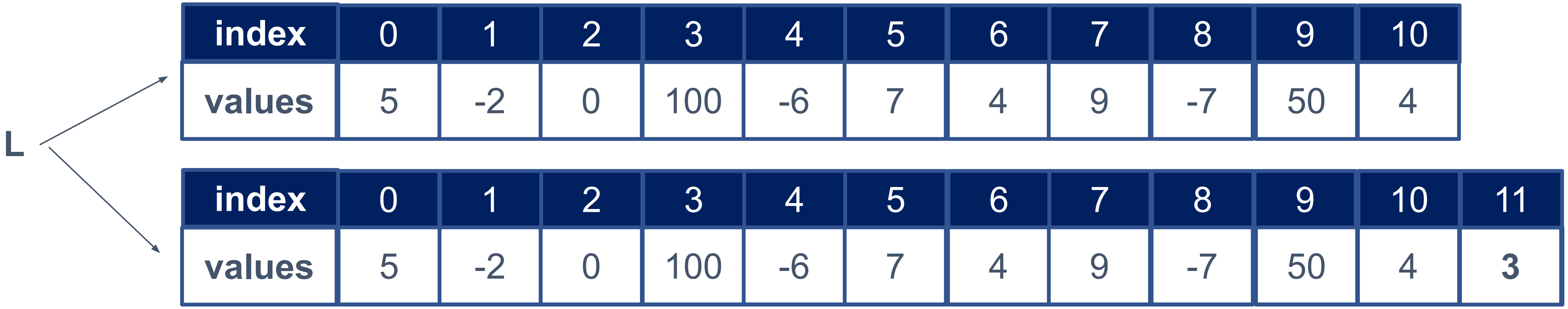
index	0	1	2	3	4	5	6	7	8	9	10	11
values	0	0	0	0	0	0	0	0	0	0	0	0

Create a new longer array

Array Resizing

- Two problems of an array due to its fixed length
  - Memory wastage: If it contains only n(<<N) valid elements
  - Memory shortage: If it wants to contain more than N elements

- Array resizing: create another larger array and copy all the elements
  - L.append(3) when the current array is **full**



index	0	1	2	3	4	5	6	7	8	9	10
values	5	-2	0	100	-6	7	4	9	-7	50	4

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	**3**

Create a new longer array Copy all elements and switch **L** Add a new element

Array Resizing

- Array resizing is expensive: new memory boxes and copy operation
  - Increasing size by one every time is not efficient (too many resizing)
  - Increasing size too much at once is not efficient either (memory wastage)

- To resize fewer, Python list size grows as 0, 4, 8, 16, 25, 35, 46, 58, …
  - Mild over-allocation proportional to the current size

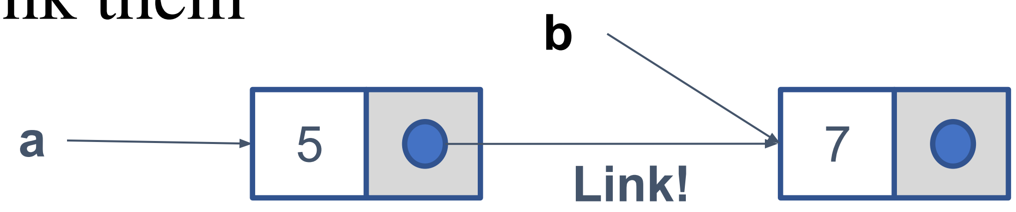
- Anyway… is there another way of organizing a collection of data to support append and pop easily?

Linked list

- **Single Linked list**
- **Sentinel**

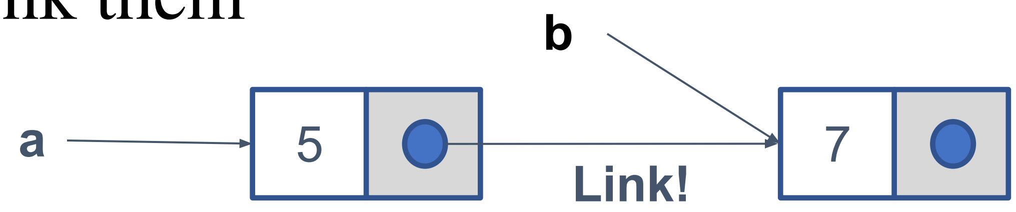
Basis

- Let's define a class that contains a single integer value as below:
  - class LinkedNode():
  - def __init__(self, x):
  - self.val = x
  - self.next = None # A special variable for **linking** to another node
- Let's create two LinkedNodes and link them
  - a = LinkedNode(5)
  - b = LinkedNode(7)
  - a.next = b



Basis

- Let’s define a class that contains a single integer value as below:
  - class LinkedNode():
  - def __init__(self, x):
  - self.val = x
  - self.next = None # A special variable for **linking** to another node
- Let’s create two LinkedNodes and link them
  - a = LinkedNode(5)
  - b = LinkedNode(7)
  - a.next = b

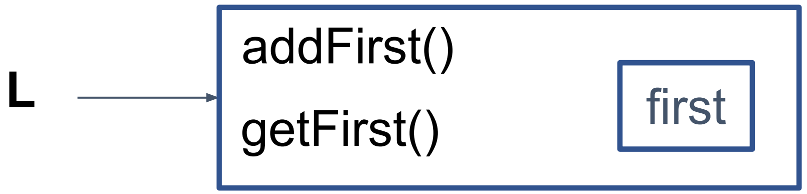


- Now we can access LinkedNode **b** through LinkedNode **a** because they are **linked!**
  - b.val
  - a.next.val

Single Linked Lists

- A linked list whose node has a single link as we've just seen
  - Every node can be access through the **first** node

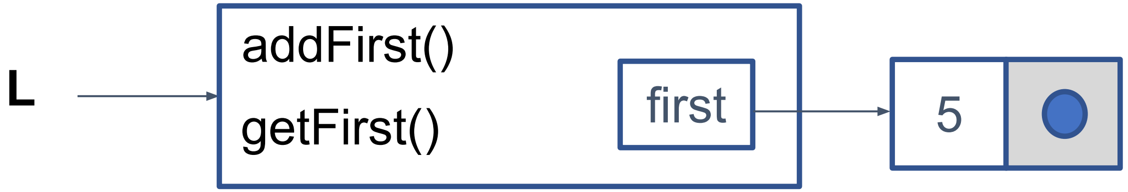
- An example of a SLList consisting of two basic methods and one variable
  - L = SLList()



Single Linked Lists

- A linked list whose node has a single link as we’ve just seen
  - Every node can be access through the **first** node

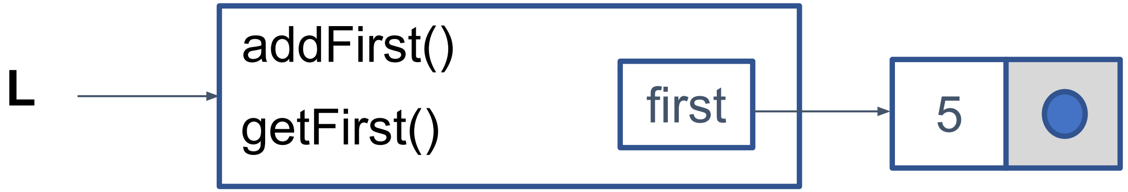
- An example of a SLList consisting of two basic methods and one variable
  - L = SLList()
  - L.addFirst(5)



Single Linked Lists

- A linked list whose node has a single link as we've just seen
    - Every node can be access through the **first** node

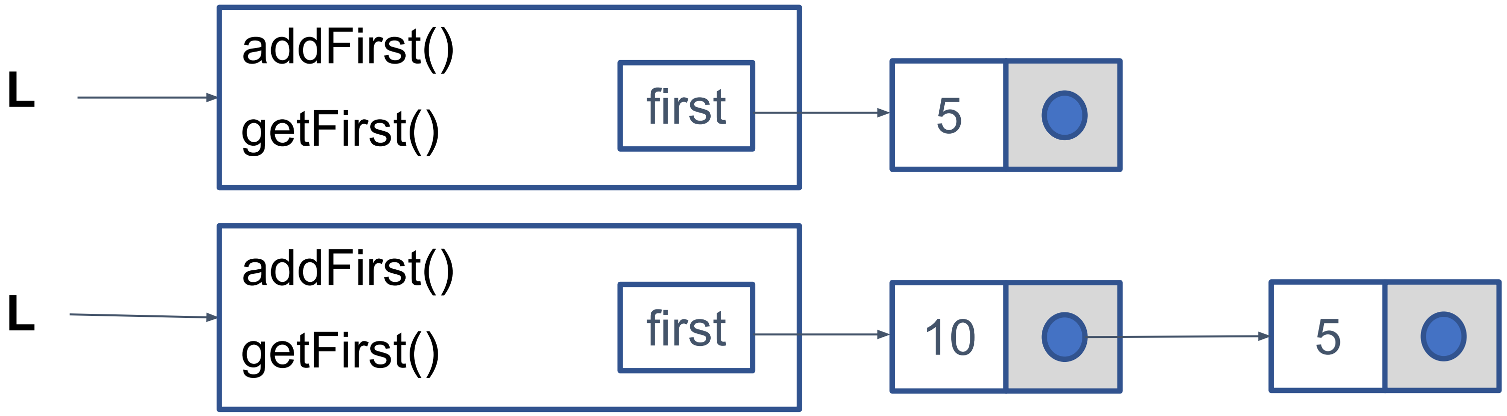
- An example of a SLList consisting of two basic methods and one variable
    - L = SLList()
    - L.addFirst(5)
    - L.getFirst()
        - 5



Single Linked Lists

- A linked list whose node has a single link as we've just seen
  - Every node can be access through the **first** node

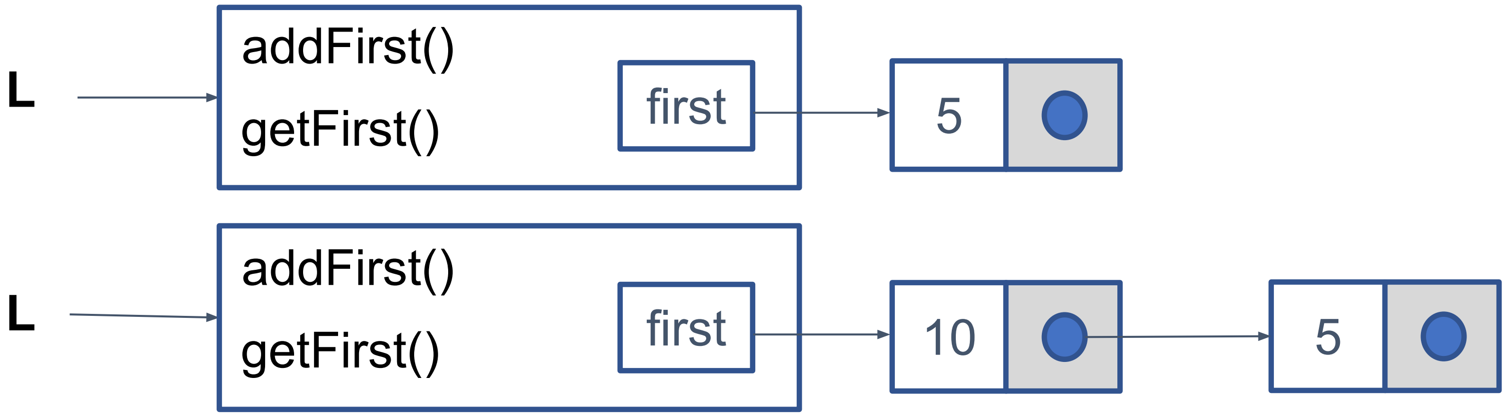
- An example of a SLList consisting of two basic methods and one variable
  - L = SLList()
  - L.addFirst(5)
  - L.getFirst()
    - 5
  - L.addFirst(10)



Single Linked Lists

- A linked list whose node has a single link as we've just seen
  - Every node can be access through the **first** node

- An example of a SLList consisting of two basic methods and one variable
  - L = SLList()
  - L.addFirst(5)
  - L.getFirst()
    - 5
  - L.addFirst(10)
  - L.getFirst()
    - 10

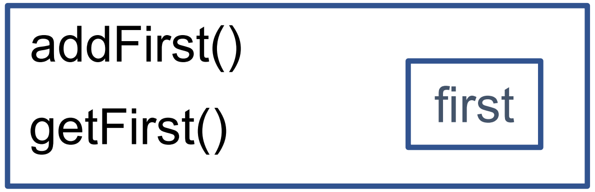


Single Linked Lists – addFirst and getFirst

```
class SLList():
    def __init__(self) -> None:
        self.first = None

    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.first
        self.first = newFirst

    def getFirst(self) -> int:
        if self.first:
            return self.first.val
```



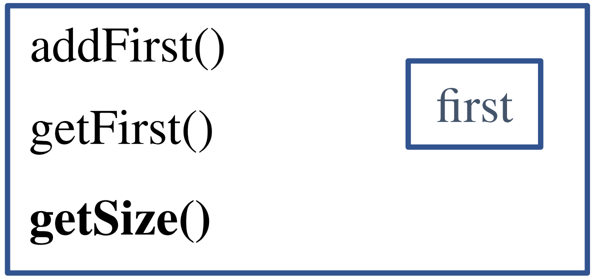
Let’s add more functionality to make our linked list useful!

Single Linked Lists – size

```
class SLList():
    def __init__(self) -> None:
        self.first = None

    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.first
        self.first = newFirst

    def getFirst(self) -> int:
        if self.first:
            return self.first.val
```

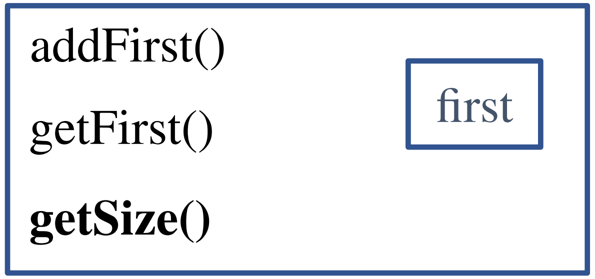


Single Linked Lists – size

```
class SLList():
    def __init__(self) -> None:
        self.first = None

    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.first
        self.first = newFirst

    def getFirst(self) -> int:
        if self.first:
            return self.first.val
```



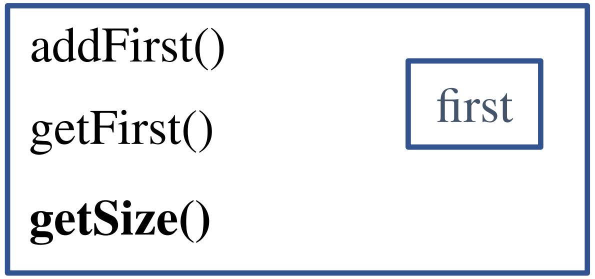
```
    def getSize(self) -> int:
        curNode = self.first
        size = 0
        while curNode != None: #Navigate the whole list
            size += 1
            curNode = curNode.next
        return size
```

Single Linked Lists – size

```
class SLList():
    def __init__(self) -> None:
        self.first = None

    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.first
        self.first = newFirst

    def getFirst(self) -> int:
        if self.first:
            return self.first.val
```



addFirst()

getFirst()

getSize()

first

```
def getSize(self) -> int:
    curNode = self.first
    size = 0
    while curNode != None: #Navigate the whole list
        size += 1
```

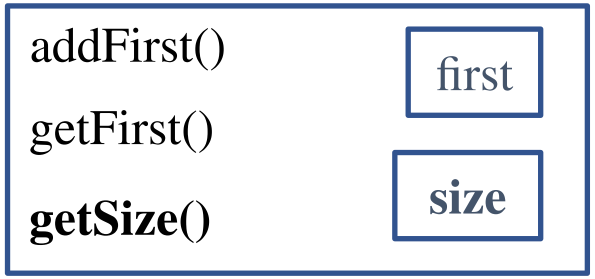
But it takes **O(N)** time…
How to reduce the time cost?

Single Linked Lists – size and size variable

```
class SLList():
        def __init__(self) -> None:
                self.first = None

        def addFirst(self, x: int) -> None:
                newFirst = LinkedNode(x)
                newFirst.next = self.first
                self.first = newFirst


         def getFirst(self) -> int:
                if self.first:
```



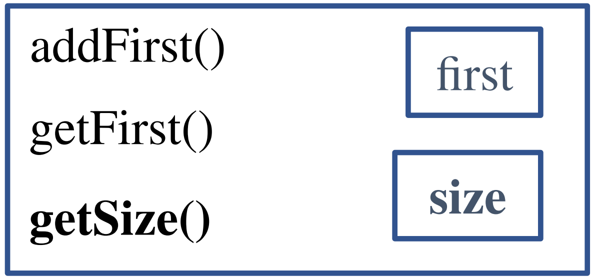
A special variable that caches the size information!
Then getSize() implementation becomes very simple

Single Linked Lists – size and size variable

```
class SLList():
    def __init__(self) -> None:
        self.first = None

    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.first
        self.first = newFirst

    def getFirst(self) -> int:
        if self.first:
```


addFirst()

getFirst()

getSize()

first

size

A special variable that caches the size information!
Then getSize() implementation becomes very simple

```
def getSize(self) -> int:
    return self.size   # O(1)!
```

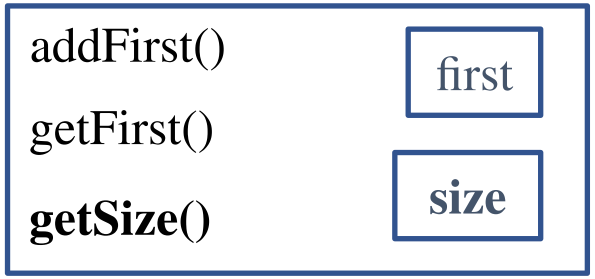
Now we need to manage the size variable properly.
+1 operation in each add function call

Single Linked Lists – size and size variable

```
class SLList():
    def __init__(self) -> None:
        self.first = None
        self.size = 0

    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.first
        self.first = newFirst
        self.size += 1

    def getFirst(self) -> int:
```



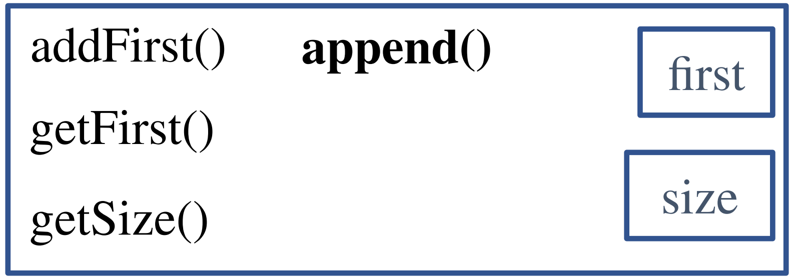
A special variable that caches the size information!
Then getSize() implementation becomes very simple

```
def getSize(self) -> int:
    return self.size   # O(1)!
```

Now we need to manage the size variable properly.
+1 operation in each add function call

Single Linked Lists – append

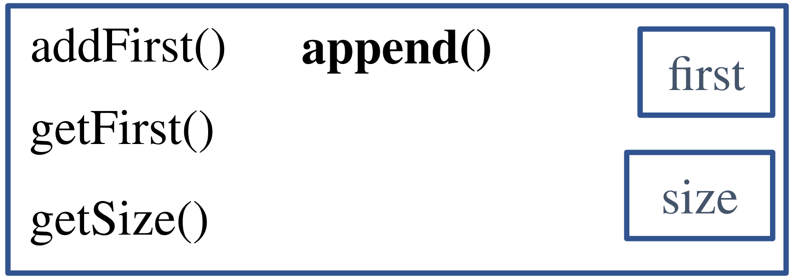
```
class SLList():
    def __init__(self) -> None:
        self.first = None
        self.size = 0
    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.first
        self.first = newFirst
        self.size += 1
     def getFirst(self) -> int:
        if self.first:
```



Now we want to **append** a new node at the **end** of a linked list

Single Linked Lists – append

```
class SLList():
    def __init__(self) -> None:
        self.first = None
        self.size = 0
    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.first
        self.first = newFirst
        self.size += 1
    def getFirst(self) -> int:
        if self.first:
```

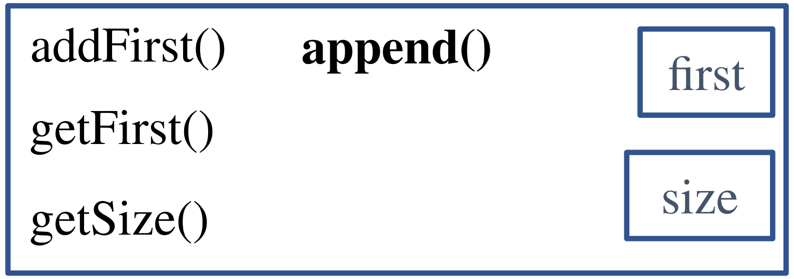


Now we want to **append** a new node at the **end** of a linked list

```
def append(self, x: int) -> None:
    self.size += 1
    curNode = self.first
    while(curNode.next != None):
        curNode = curNode.next
    curNode.next = LinkedNode(x)
```

Single Linked Lists – append

```
class SLList():
    def __init__(self) -> None:
        self.first = None
        self.size = 0
    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.first
        self.first = newFirst
        self.size += 1
    def getFirst(self) -> int:
        if self.first:
```



Now we want to **append** a new node at the **end** of a linked list

```
def append(self, x: int) -> None:
    self.size += 1
    curNode = self.first
    while(curNode.next != None):
        curNode = curNode.next
    curNode.next = LinkedNode(x)
```

Is anything **wrong** with it?

What if SLList is **empty** (first == None)?

Linked list

- Single Linked list
- **Sentinel**

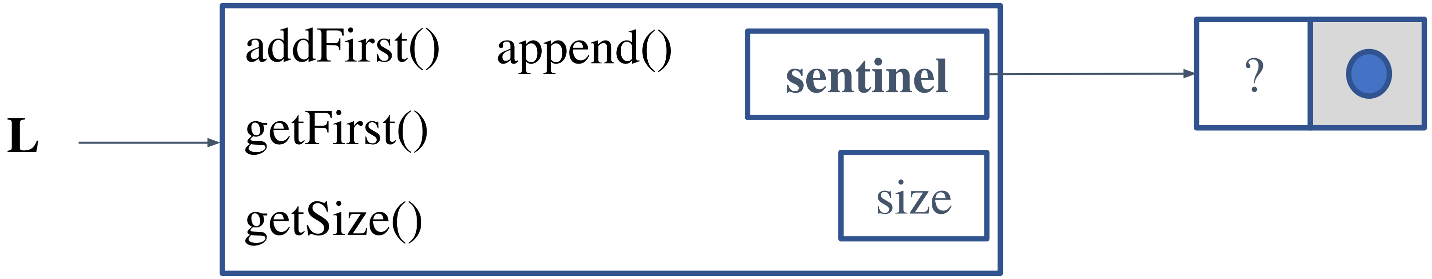
Single Linked Lists – Sentinel Node

- We now replace **first** with **sentinel**, which is a **dummy node**

Single Linked Lists – Sentinel Node

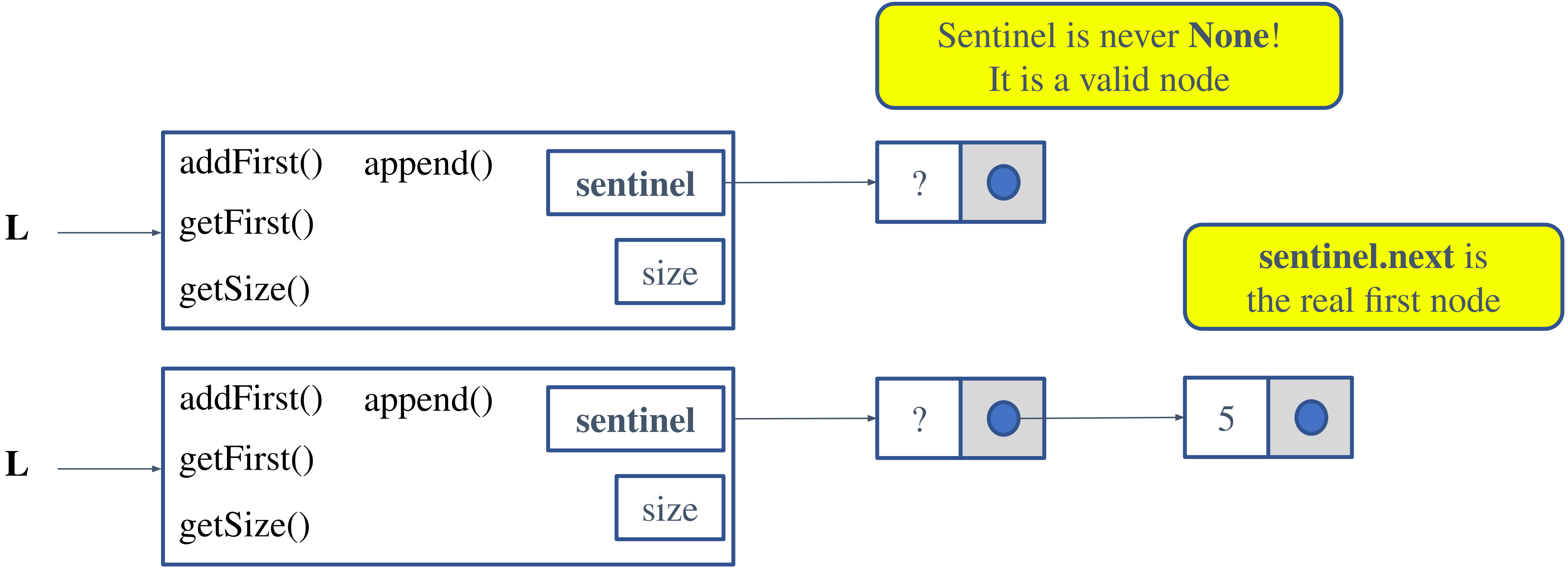
- We now replace **first** with **sentinel**, which is a **dummy node**
    - L = SLList()

Sentinel is never **None**!
It is a valid node



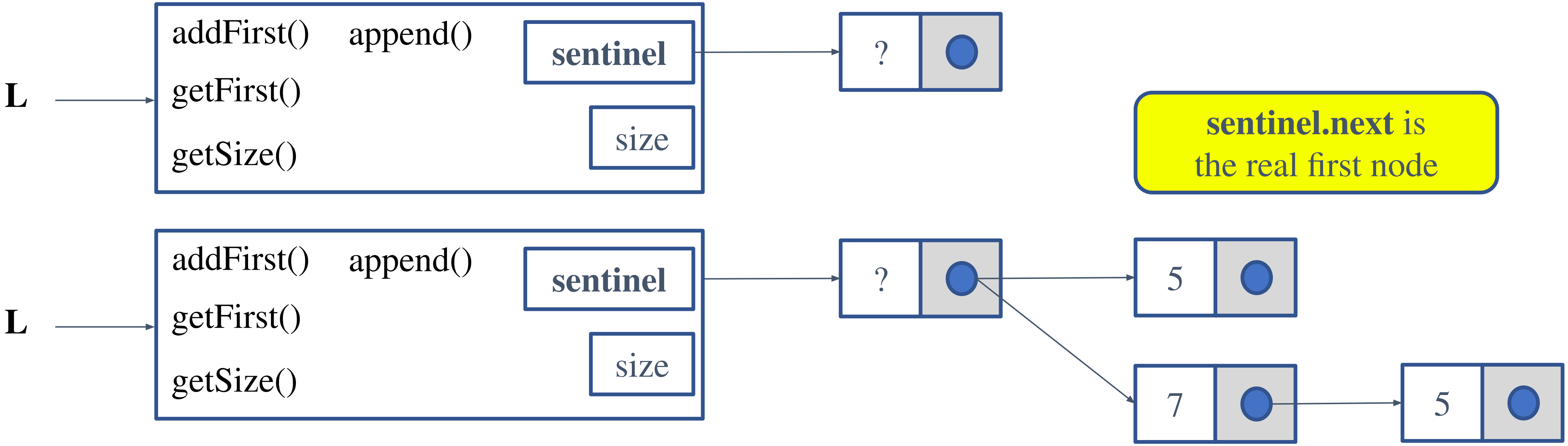
Single Linked Lists – Sentinel Node

- We now replace **first** with **sentinel**, which is a **dummy node**
  - L = SLList()
  - L.addfirst(5)



Single Linked Lists – Sentinel Node

- We now replace **first** with **sentinel**, which is a **dummy node**
  - L = SLList()
  - L.addfirst(5)
  - L.addfirst(7)

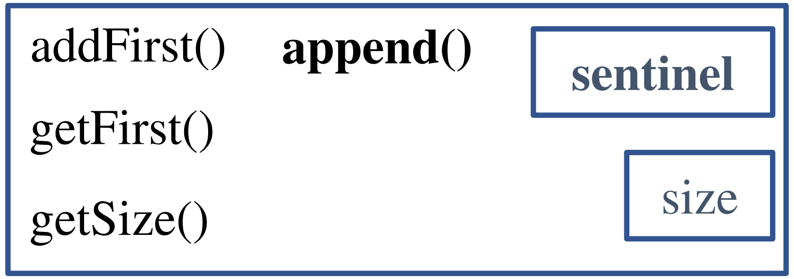


Sentinel is never **None**!
It is a valid node

sentinel.next is
the real first node

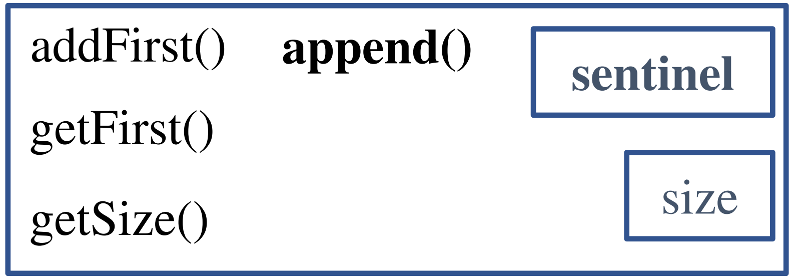
Single Linked Lists – Modification with Sentinel

```
class SLList():
    def __init__(self) -> None:
        self.sentinel = LinkedNode(0)
        self.size = 0
    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.sentinel.next
        self.sentinel.next = newFirst
        self.size += 1
    def getFirst(self) -> int:
        if self.sentinel.next:
            return self.sentinel.next.val
        return None
```



Single Linked Lists – Append with Sentinel

```
class SLList():
    def __init__(self) -> None:
        self.sentinel = LinkedNode(0)
        self.size = 0
    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.sentinel.next
        self.sentinel.next = newFirst
        self.size += 1
    def getFirst(self) -> int:
        if self.sentinel.next:
            return self.sentinel.next.val
        return None
```



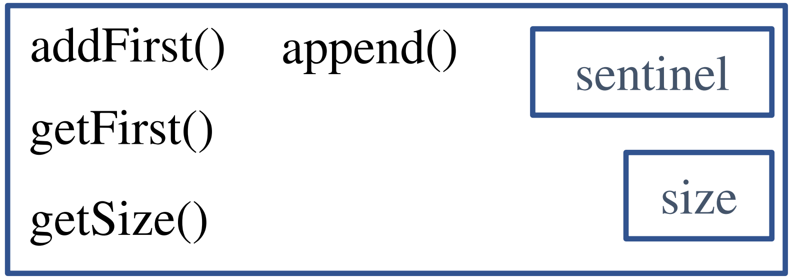
```
def append(self, x: int) -> None:
    self.size += 1
    curNode = self.sentinel
    while curNode.next != None:
        curNode = curNode.next
```

Now we don’t have any special case ☺

Single Linked Lists – Summary

```
class SLList():
    def __init__(self) -> None:
        self.sentinel = LinkedNode(0)
        self.size = 0
    def addFirst(self, x: int) -> None:
        newFirst = LinkedNode(x)
        newFirst.next = self.sentinel.next
        self.sentinel.next = newFirst
        self.size += 1
    def getFirst(self) -> int:
        if self.sentinel.next:
            return self.sentinel.next.val
        return None
```

```
def append(self, x: int) -> None:  # Improved with sentinel!
    self.size += 1
    curNode = self.sentinel
    while curNode.next != None:
        curNode = curNode.next
    curNode.next = LinkedNode(x)
```



Looking Forward …

- **Problem**: append() is still much lower than addFirst()

- **Solution**: Doubly linked list (DLList)
  - Add **another sentinel** at the back
  - Each node has not only **next** but also **prev** (pointing at the previous node)

Don't **panic**!
This is out of scope of this course ☺

Queue

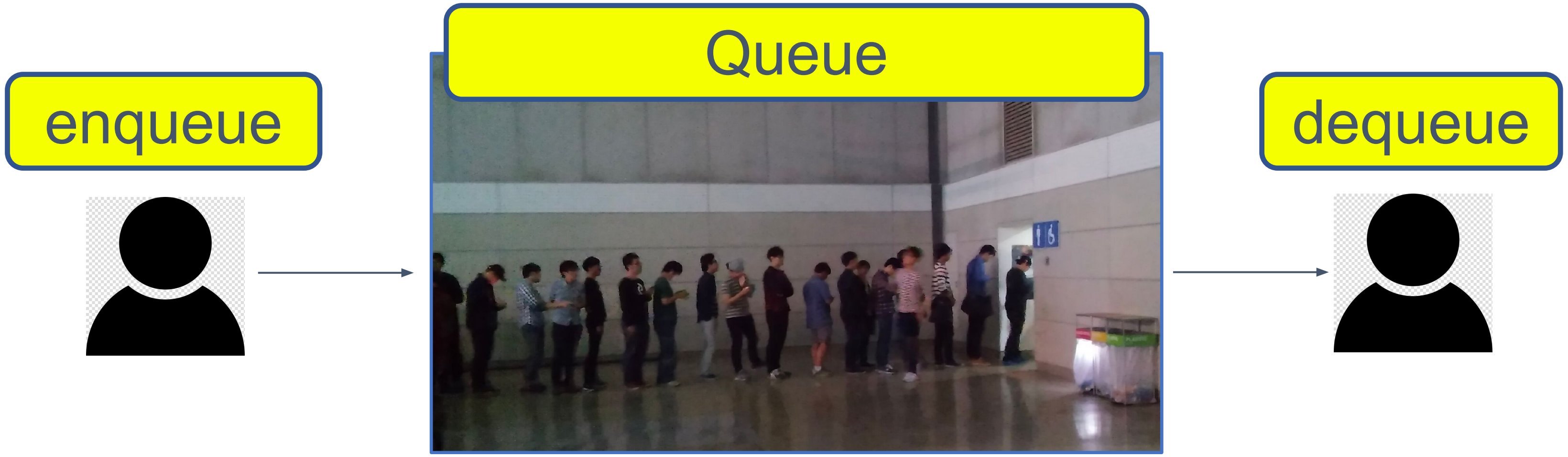
Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element from the queue

enqueue()

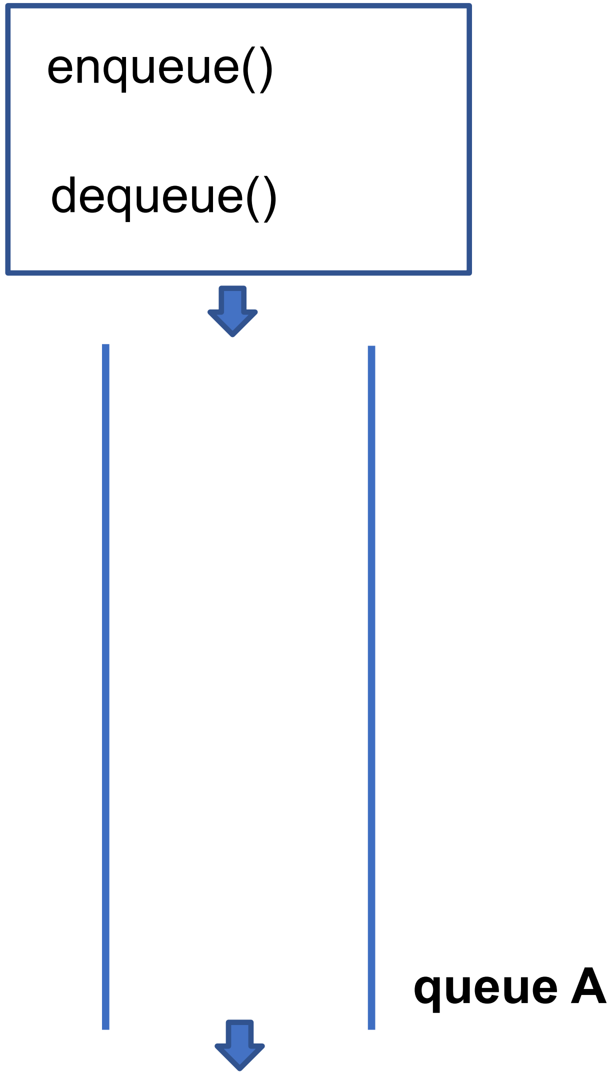
dequeue()



Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



enqueue()

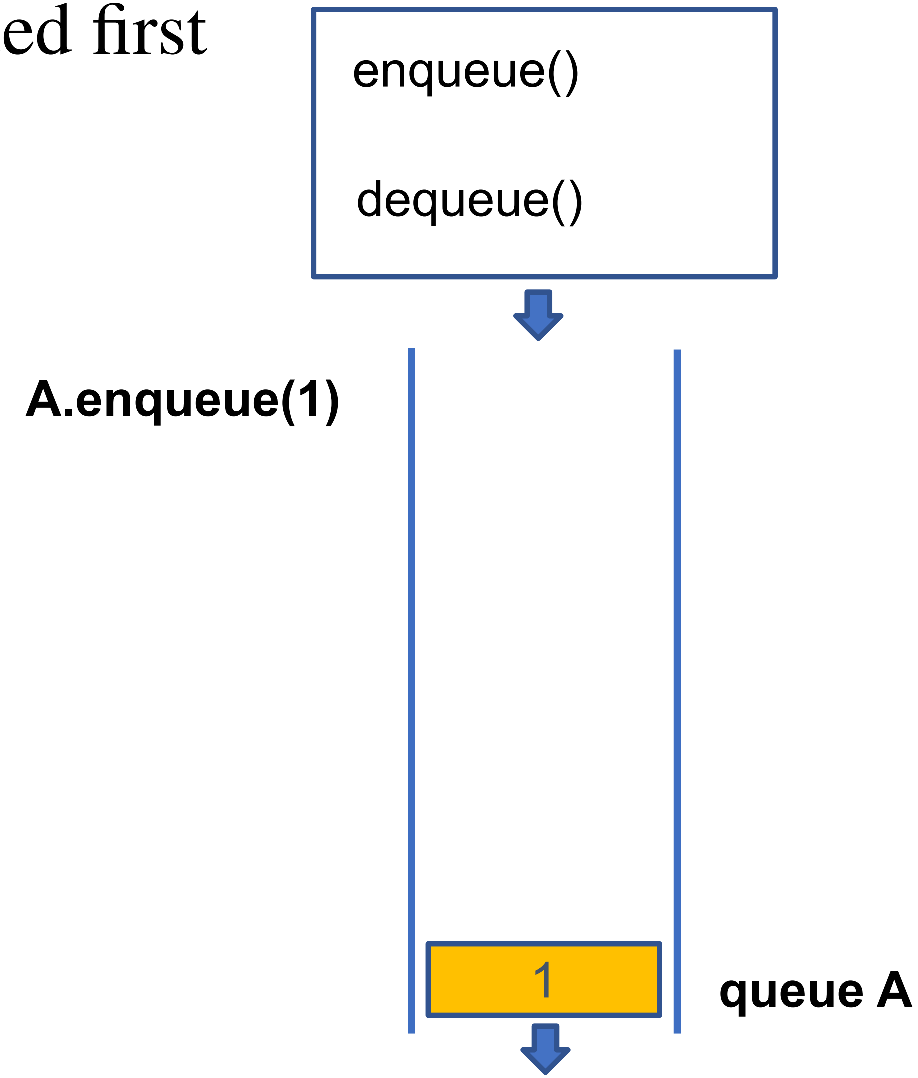
dequeue()

queue A

Queue – a First In and First Out Data Structure

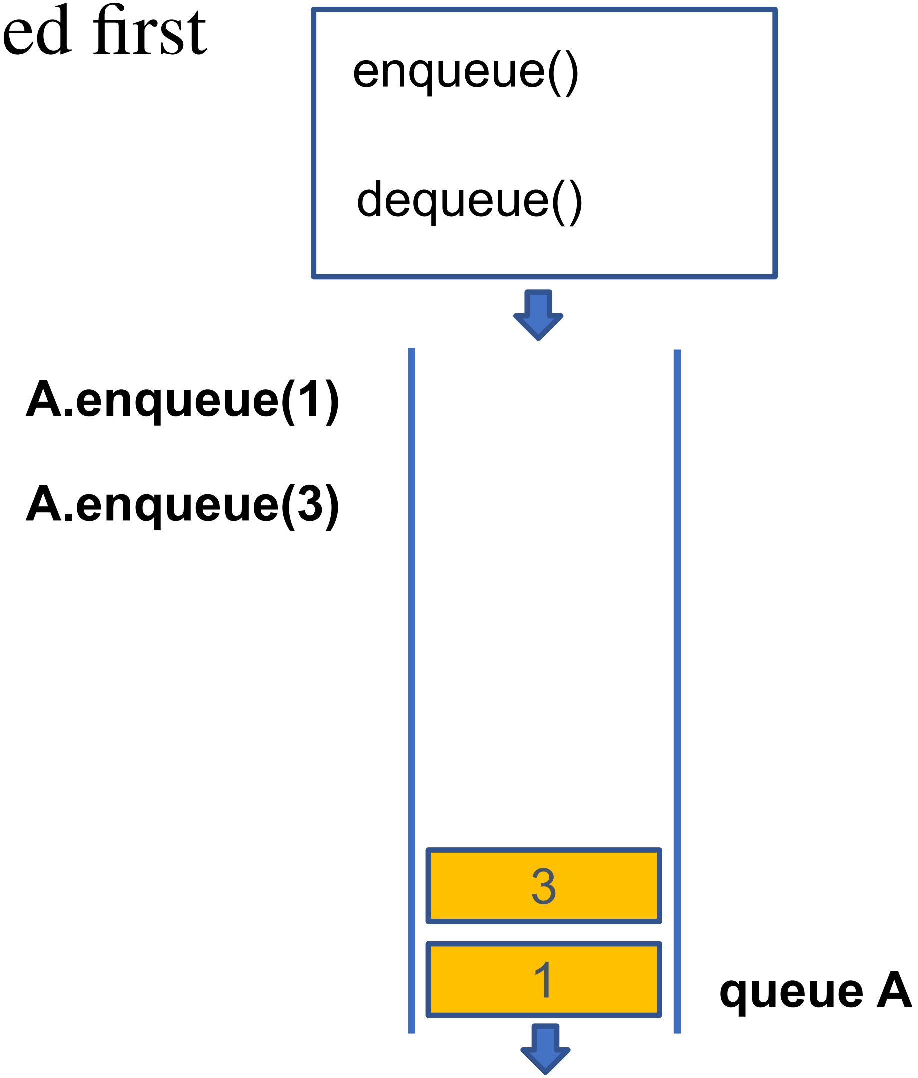
- FIFO – First enqueued element is dequeued first

- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



Queue – a First In and First Out Data Structure

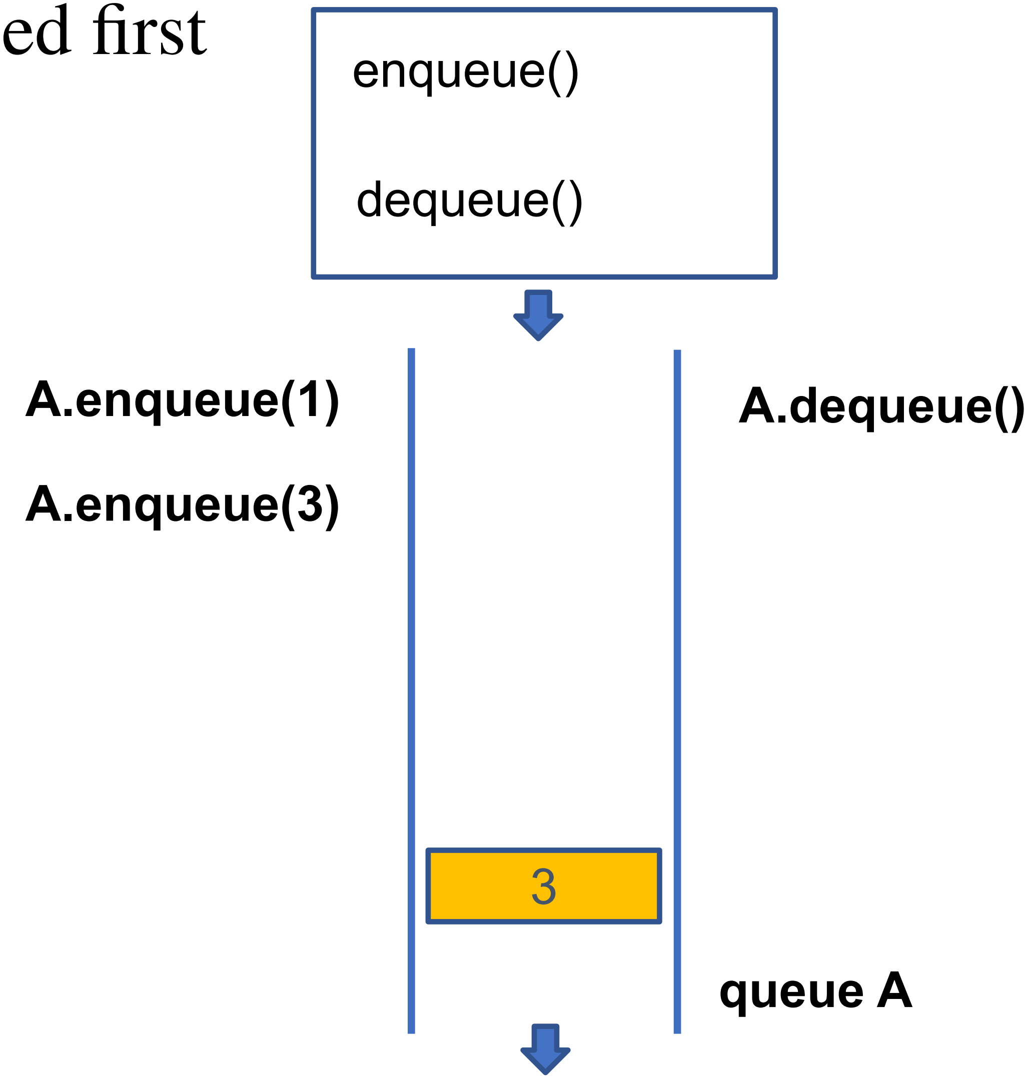
- FIFO – First enqueued element is dequeued first
- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

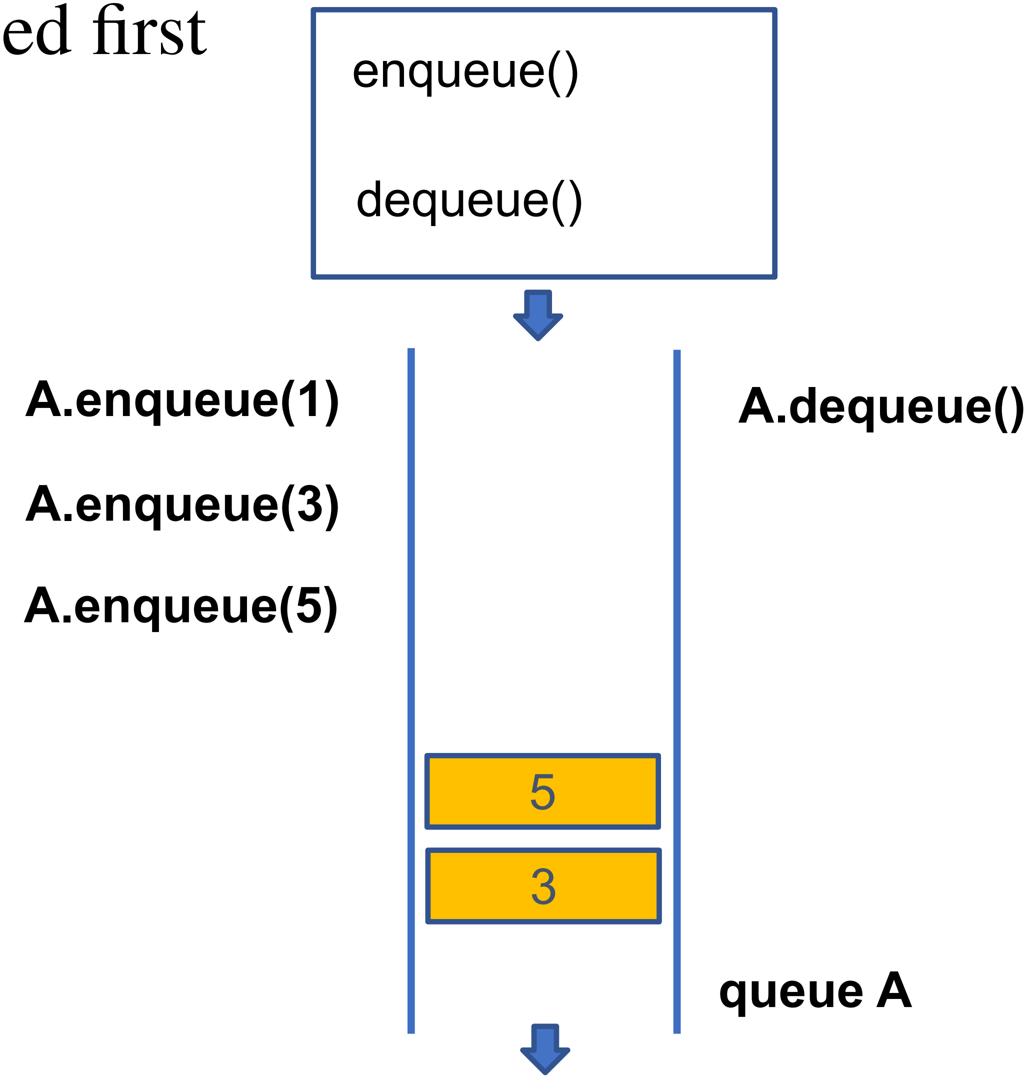
- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

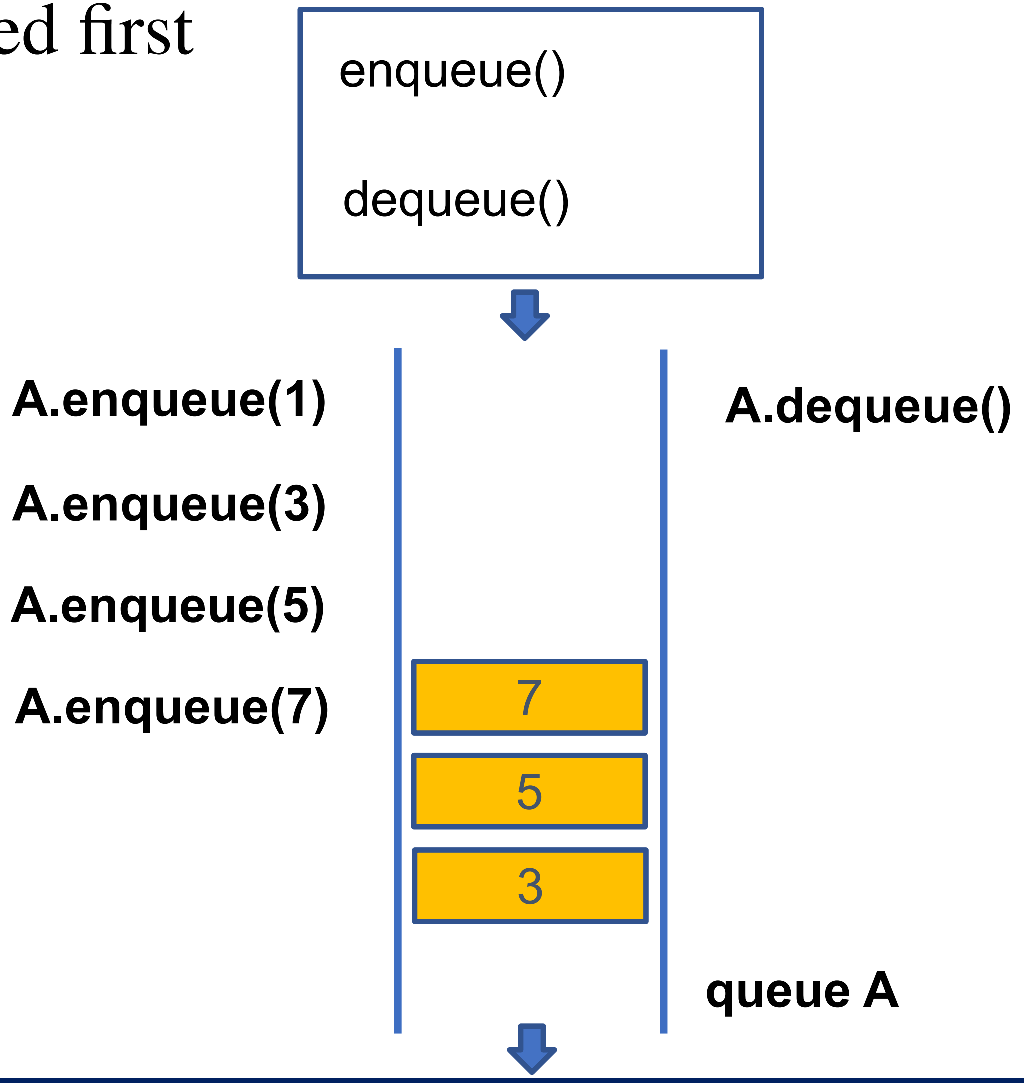
- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

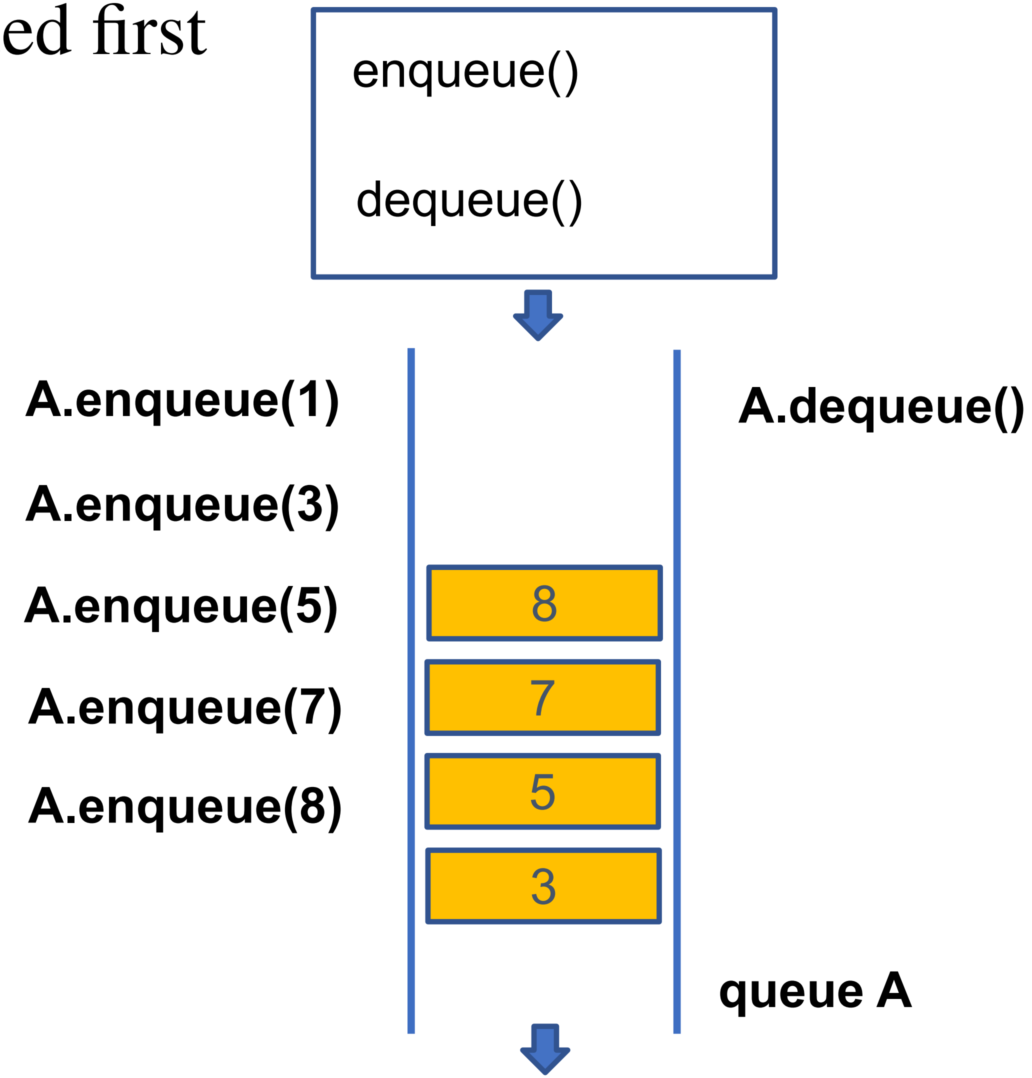
- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

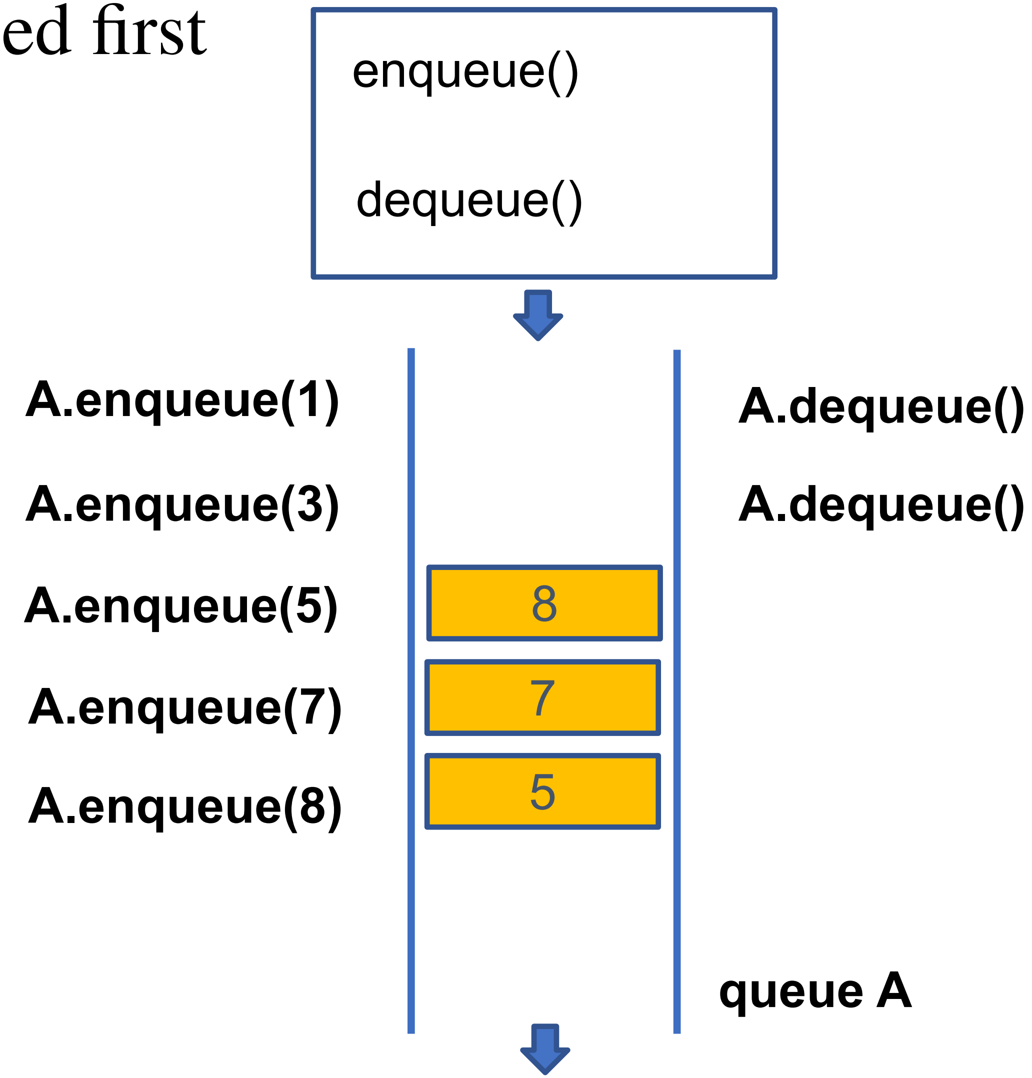
- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

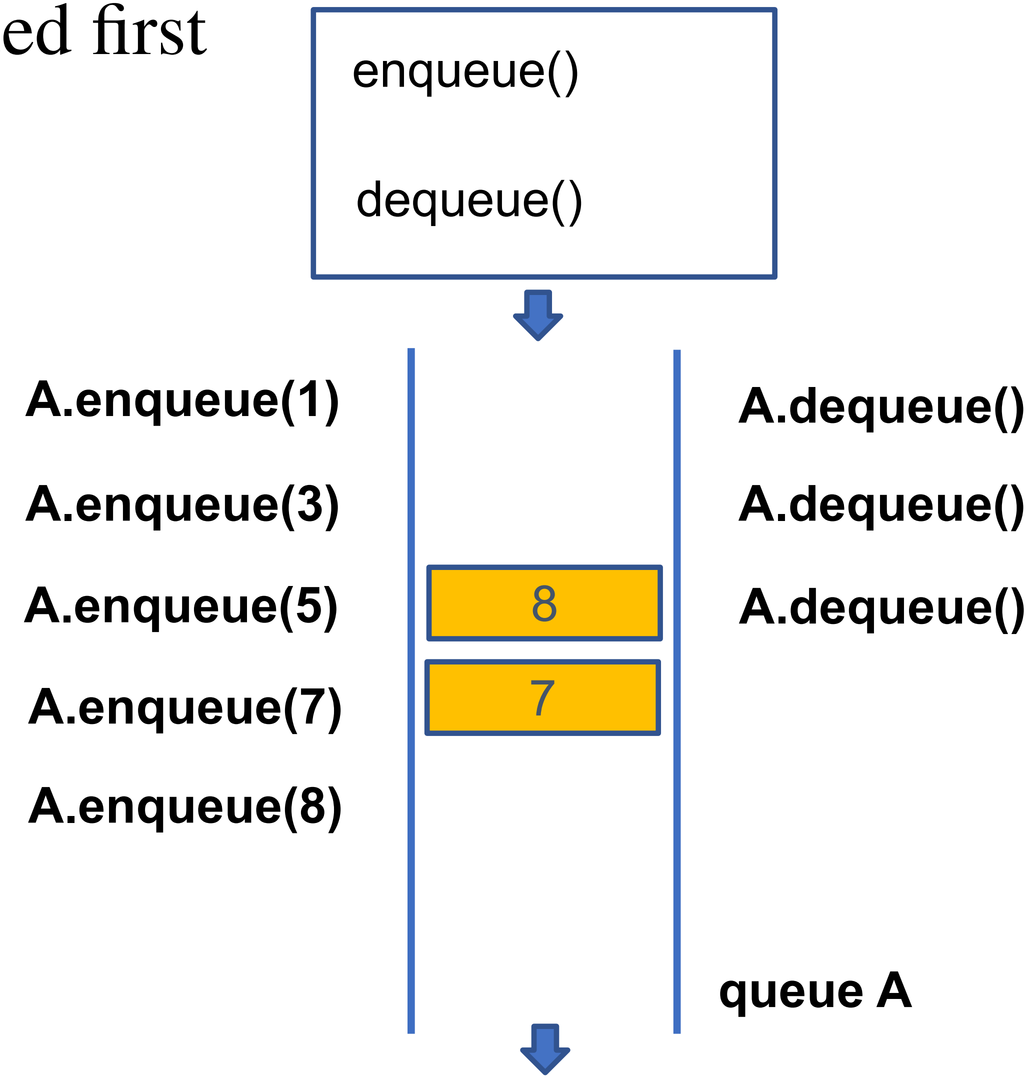
- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



enqueue()

dequeue()

A.enqueue(1)

A.enqueue(3)

A.enqueue(5)

A.enqueue(7)

A.enqueue(8)

8

7

A.dequeue()

A.dequeue()

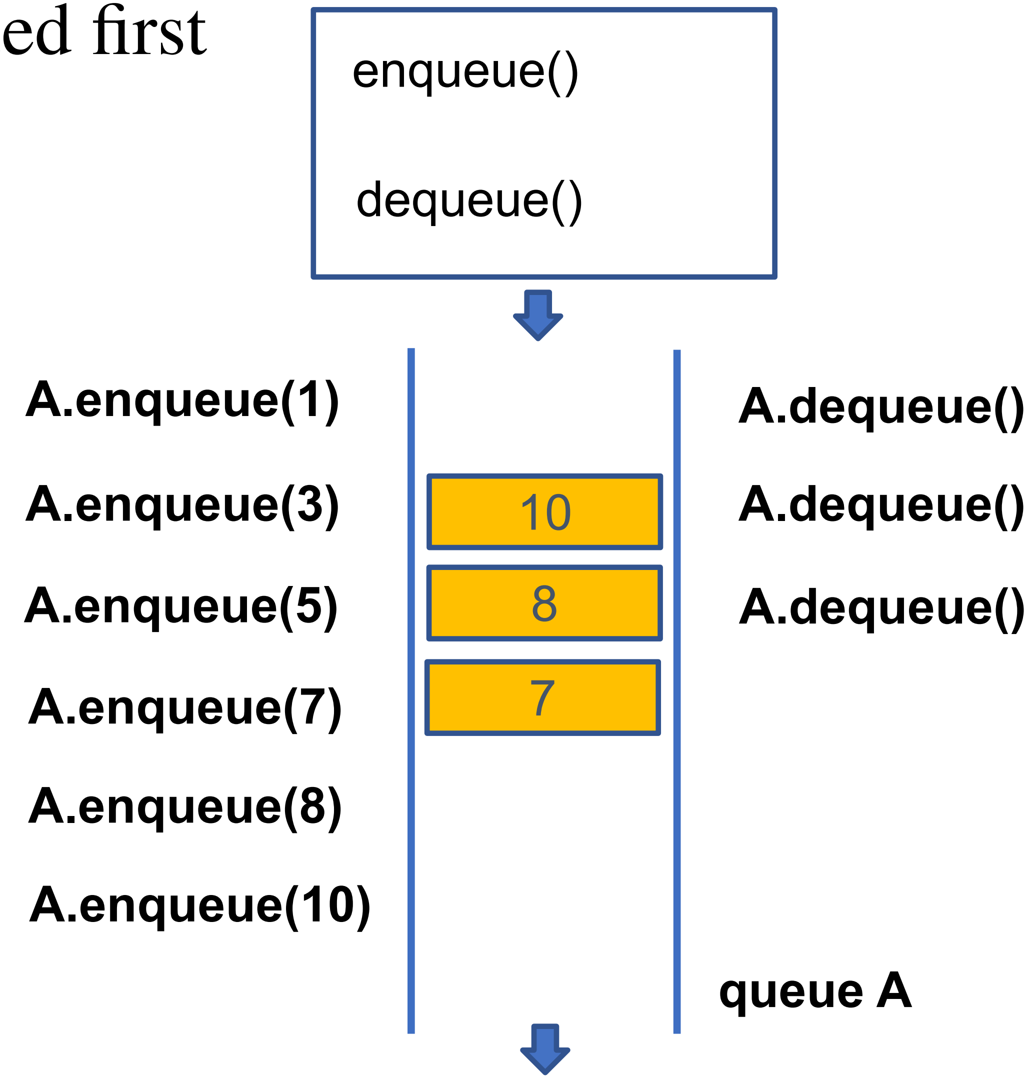
A.dequeue()

queue A

Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

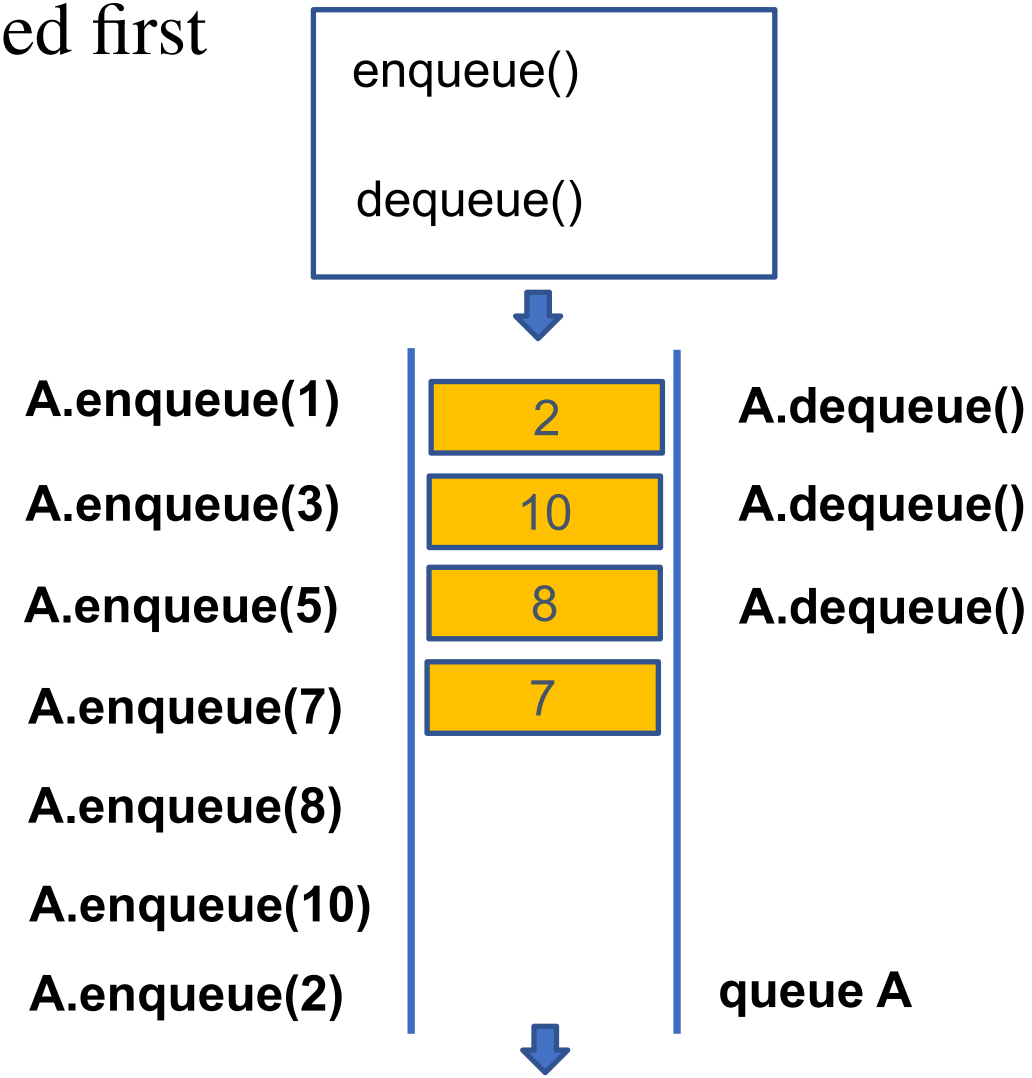
- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



Queue – a First In and First Out Data Structure

- FIFO – First enqueued element is dequeued first

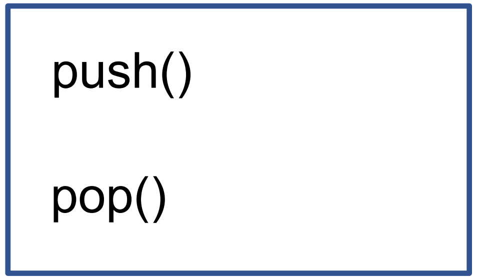
- Queue has two methods
  - enqueue(): add an element to the queue
  - dequeue(): remove the oldest element



Stack

Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first



push()

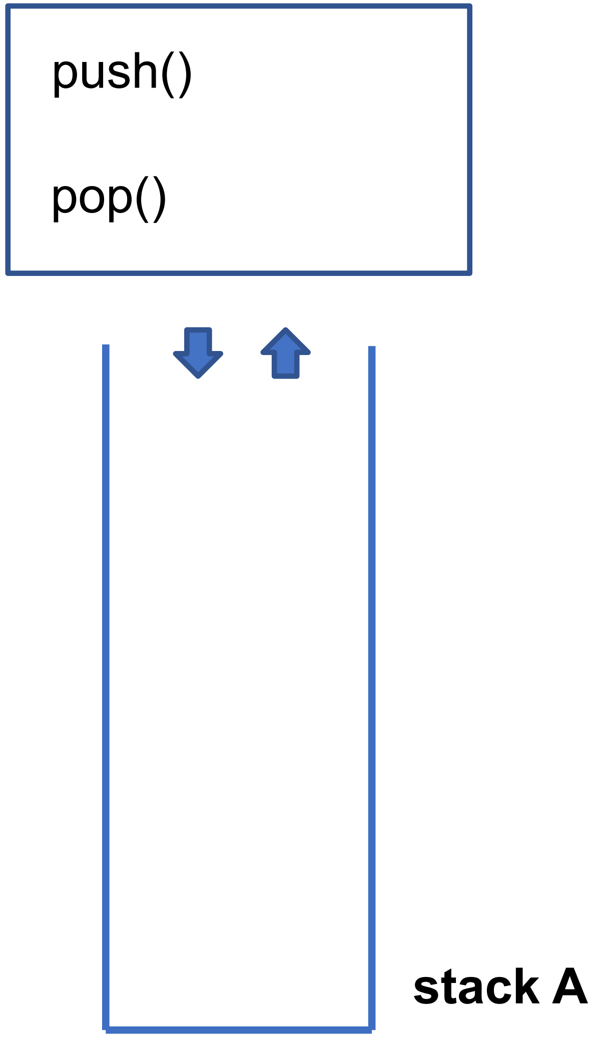
pop()

- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack

Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

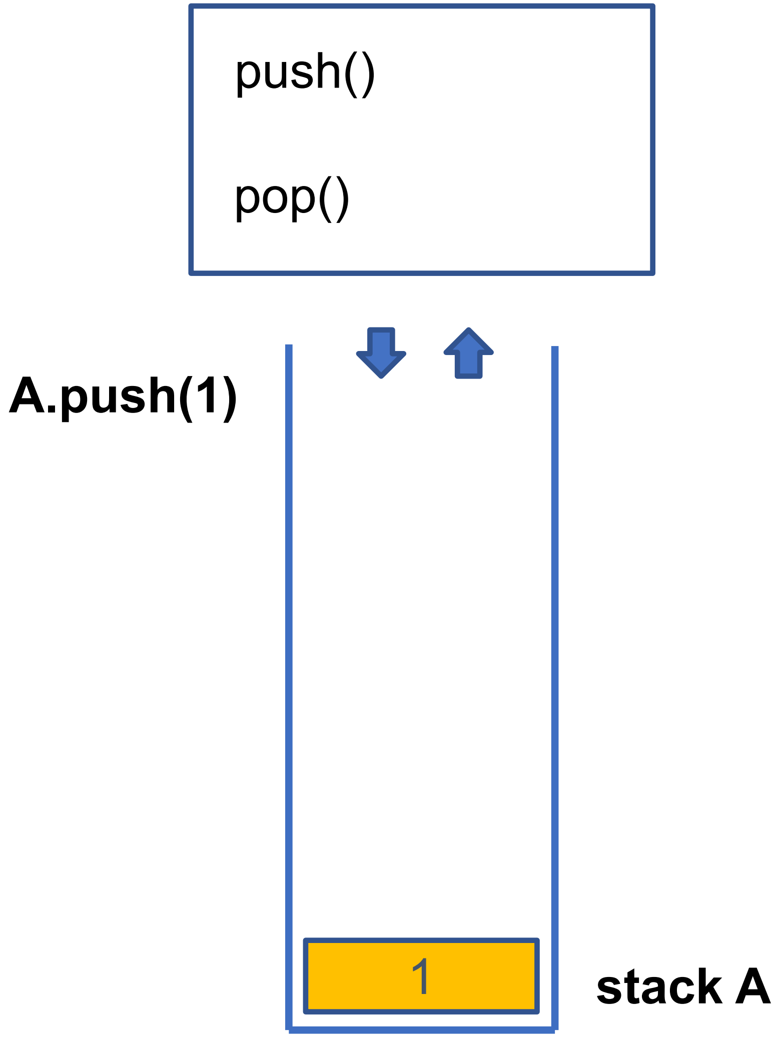
- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



Stack – a Last In and First Out Data Structure

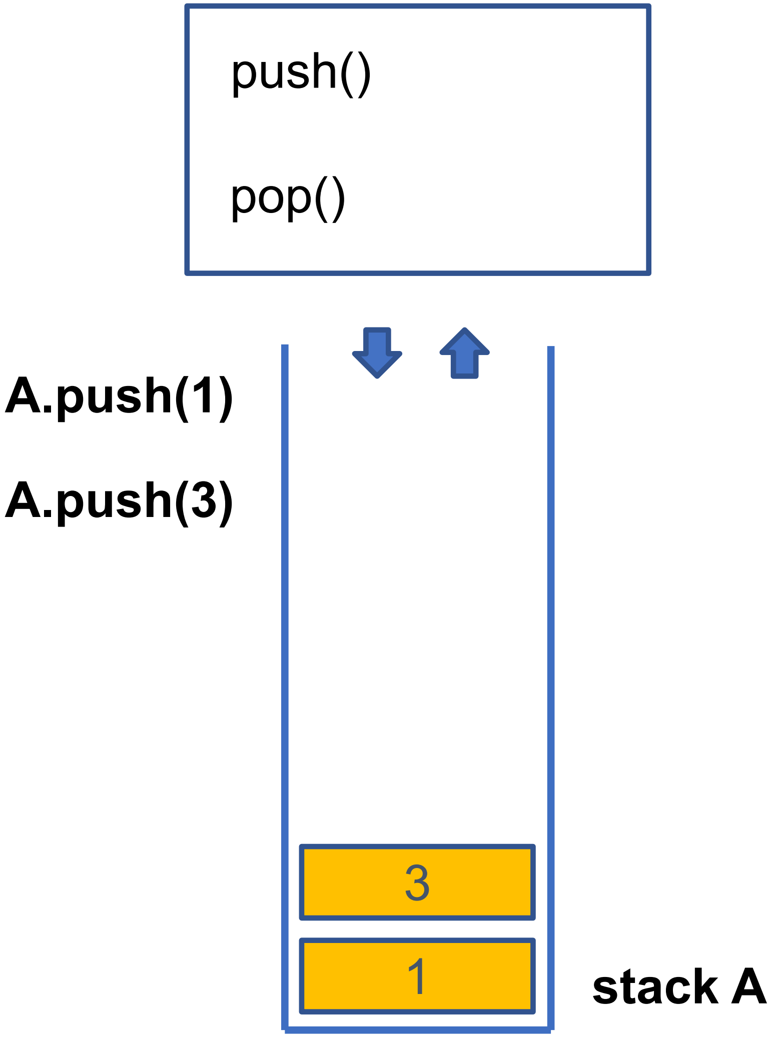
- LIFO – Last pushed element is popped first

- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



Stack – a Last In and First Out Data Structure

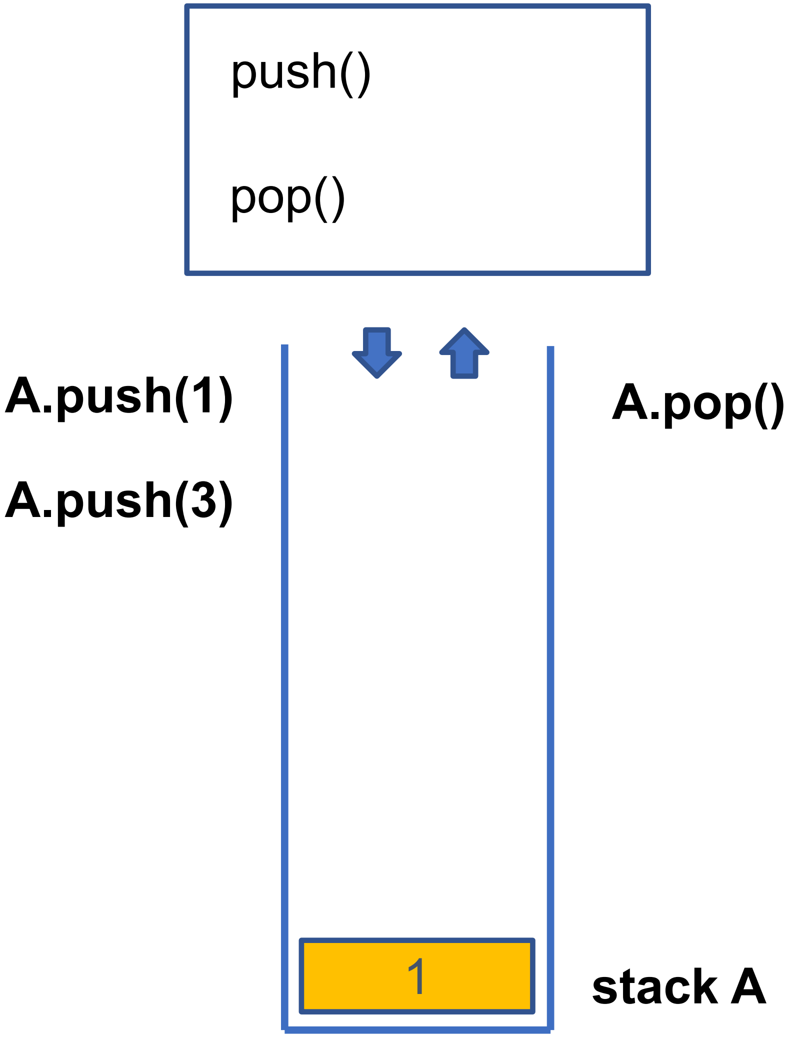
- LIFO – Last pushed element is popped first
- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

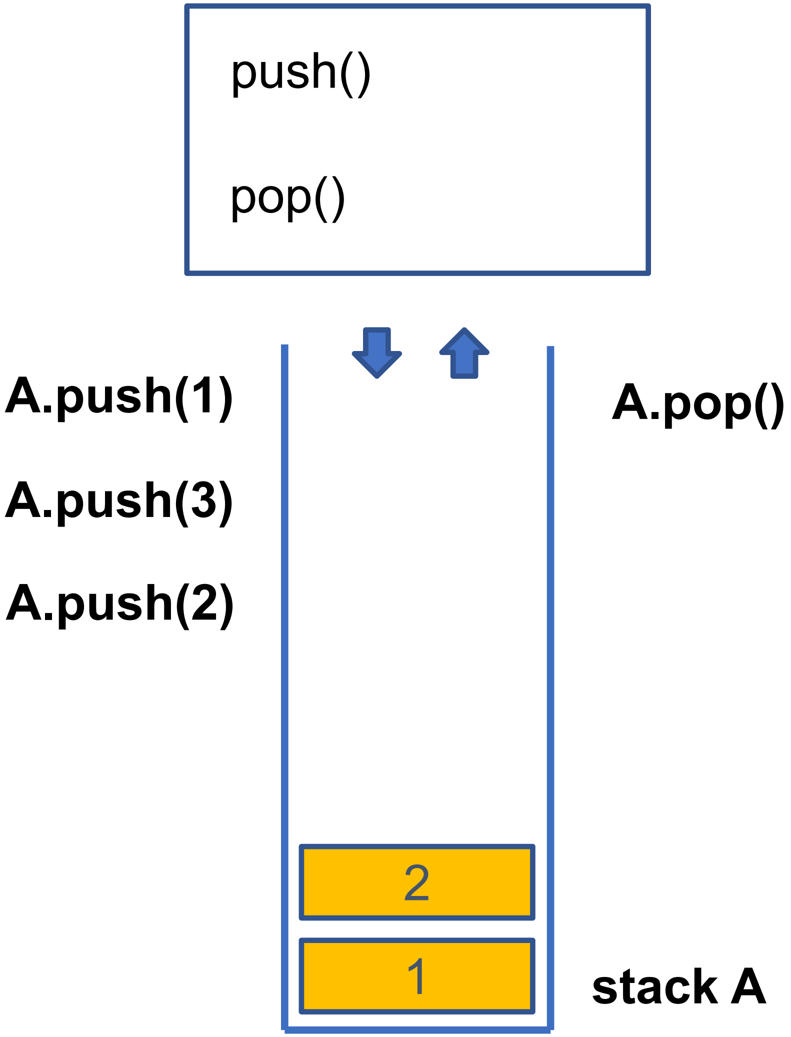
- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

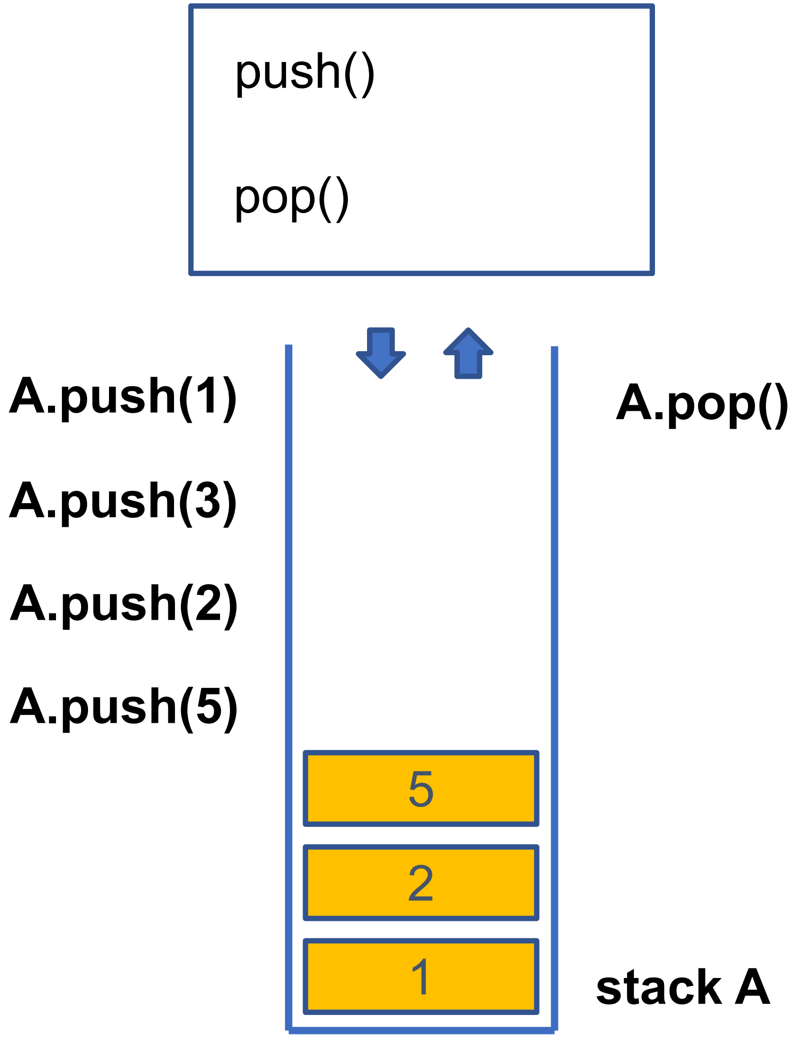
- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

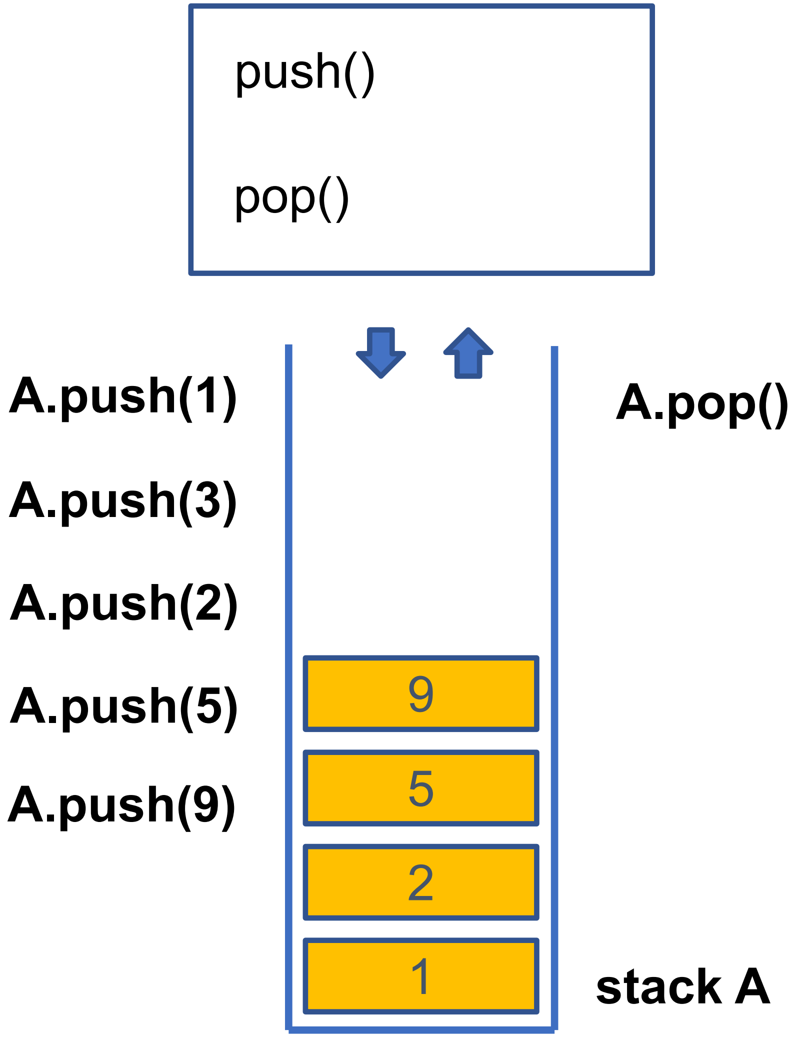
- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

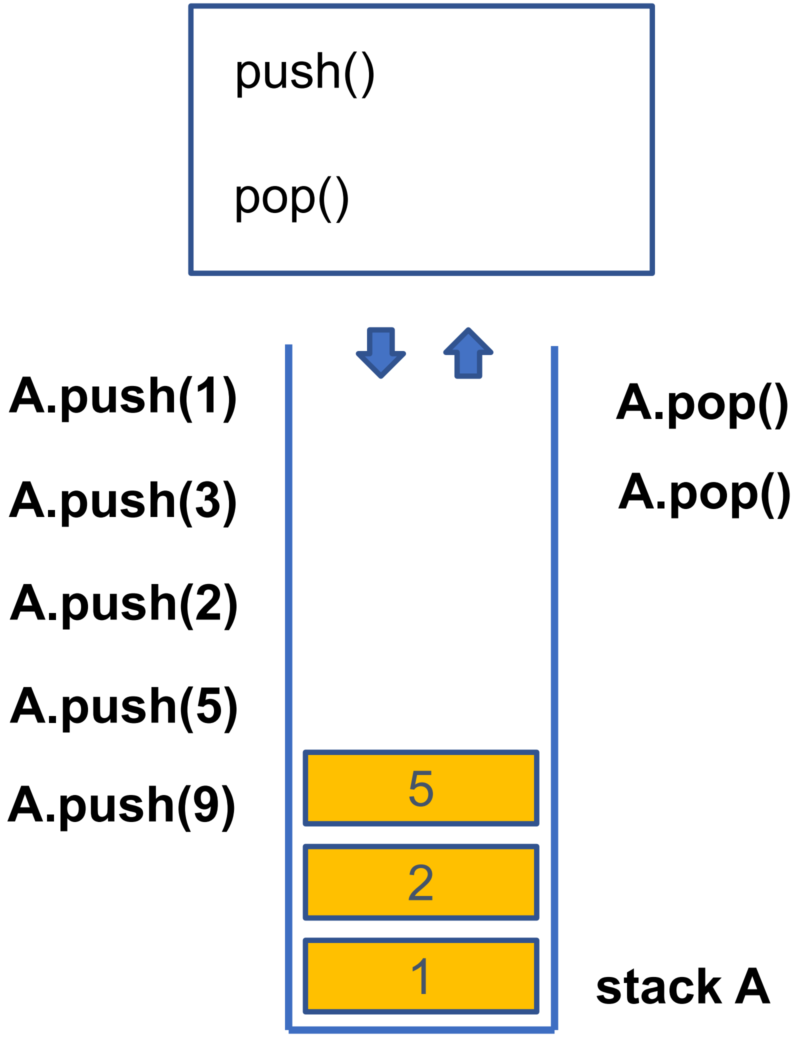
- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

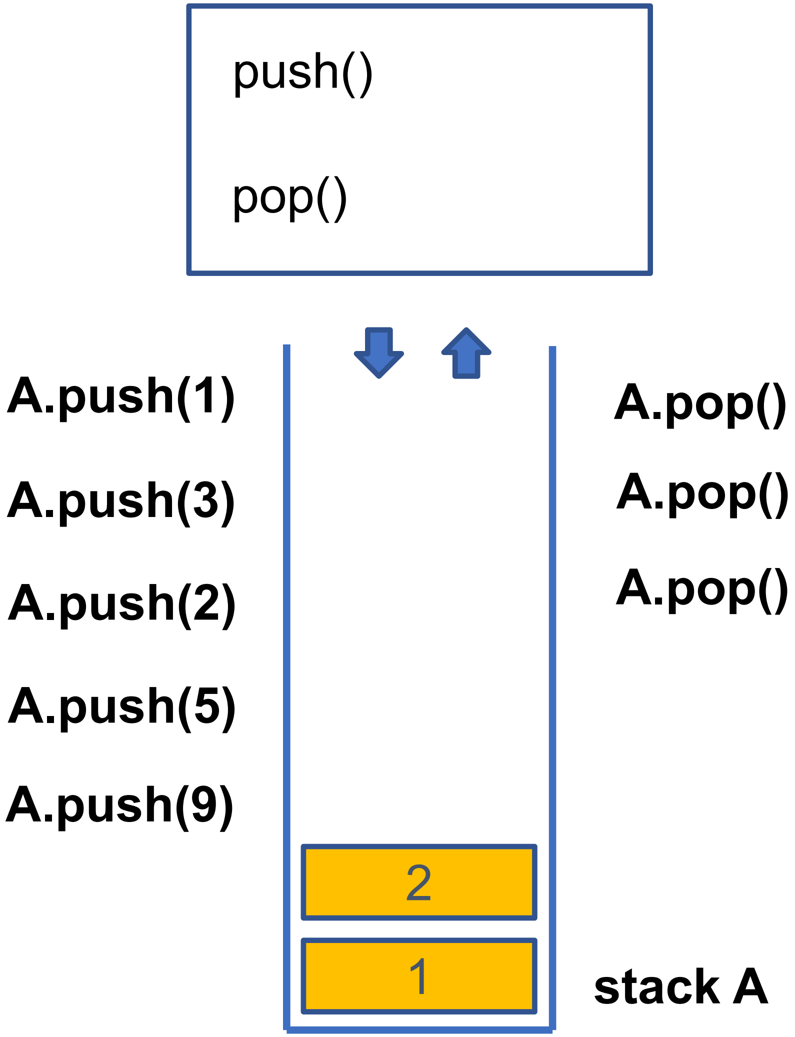
- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

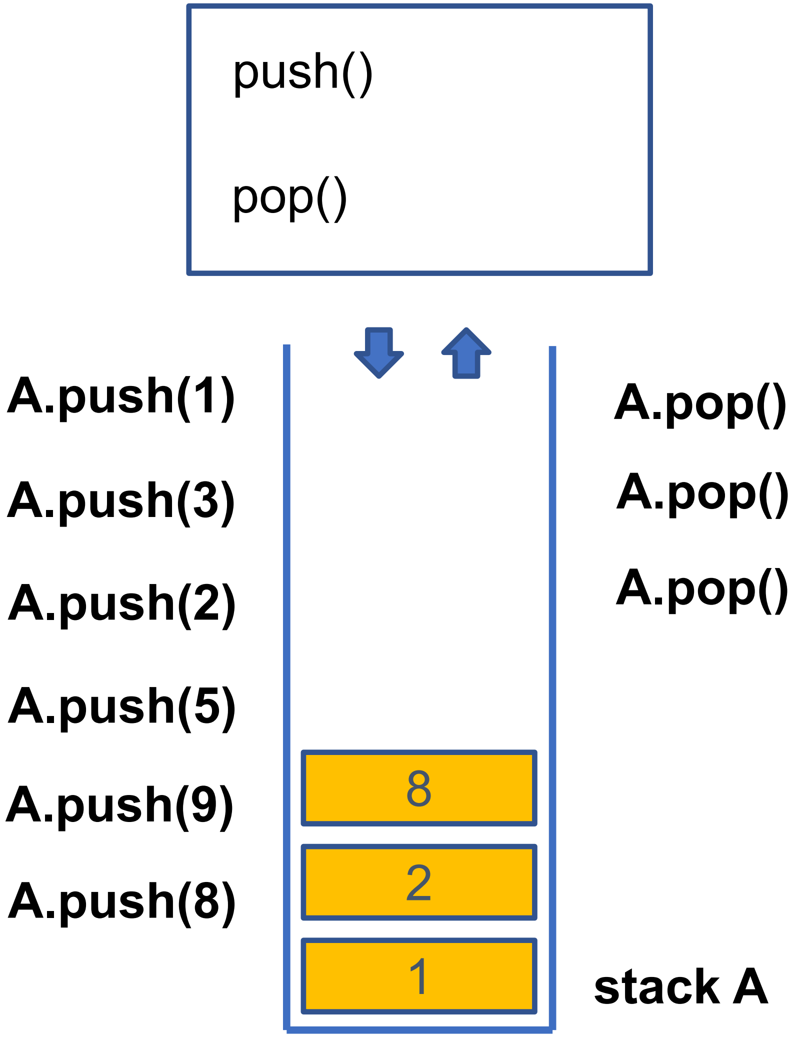
- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack



push()

pop()

A.push(1)

A.push(3)

A.push(2)

A.push(5)

A.push(9)

A.push(8)

A.pop()

A.pop()

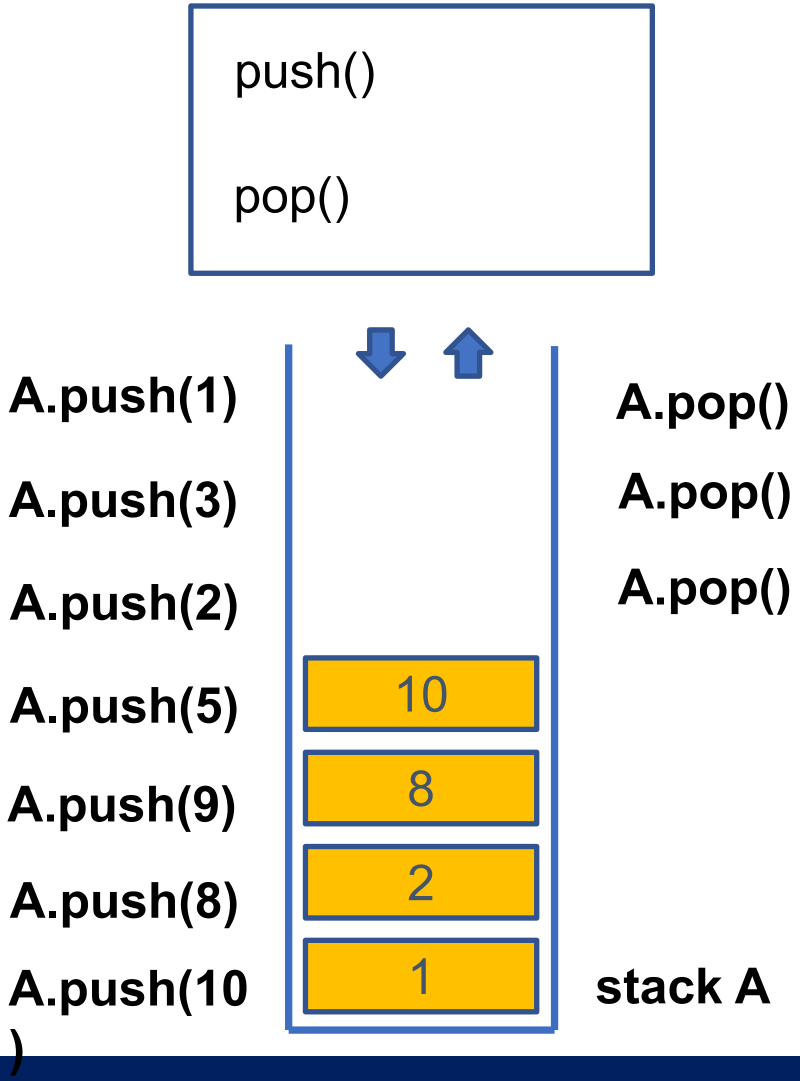
A.pop()

stack A

Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack

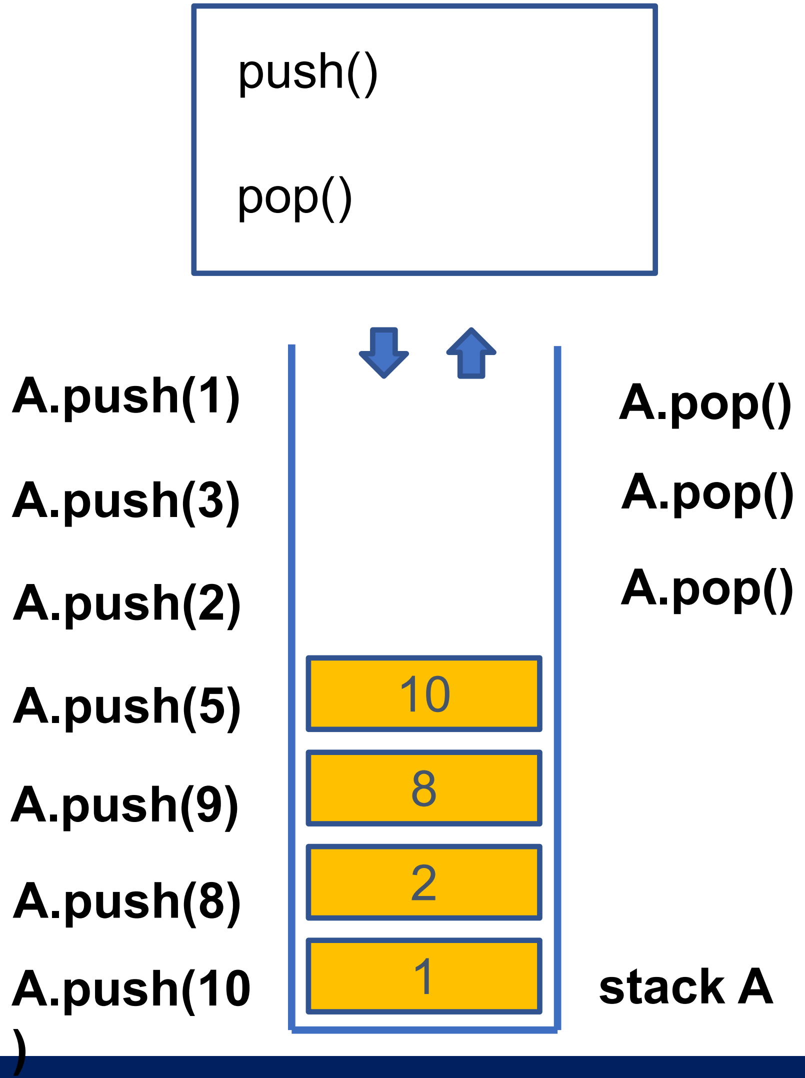


Stack – a Last In and First Out Data Structure

- LIFO – Last pushed element is popped first

- Stack has two methods
  - push(): add an element to the stack
  - pop(): remove the newest element from the stack

- Use cases
  - Undo function: Ctrl + z
  - Parentheses matching: ((){}[])


push()
pop()
A.push(1)
A.push(3)
A.push(2)
A.push(5)
A.push(9)
A.push(8)
A.push(10
)
A.pop()
A.pop()
A.pop()
10
8
2
1
stack A