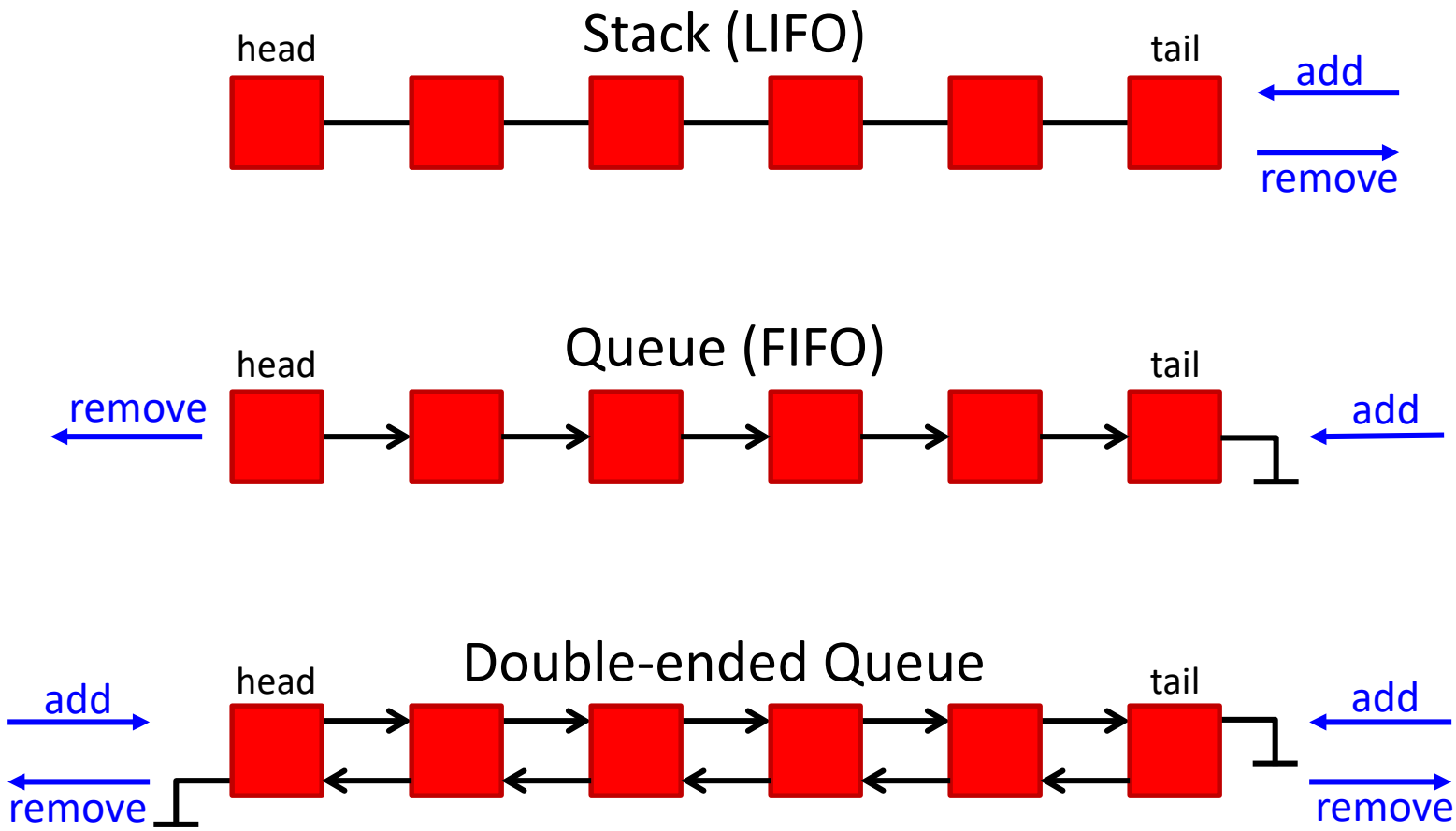
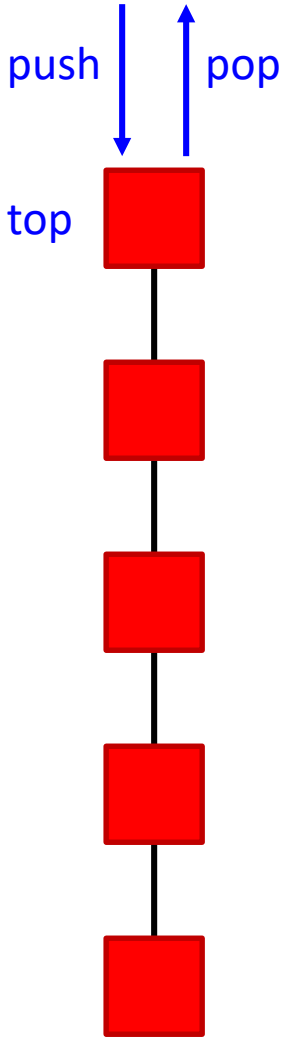
Linear Containers

Jordi Cortadella and Jordi Petit
Department of Computer Science

Linear containers



Stack



- Elements are removed in reversed order of insertion (Last-In-First-Out)
- A stack can be simply implemented with an array, vector or list (adding/removing elements to/from the last location)
- Typical applications:
  - Check balanced parenthesis
  - Backtracking
  - Activation records (function calls)
  - Store actions to "undo" them later

Evaluation of postfix expressions

- This is an infix expression. What's his value? 42 or 144?

  $8 * 3 + 10 + 2 * 4$

- It depends on the operator precedence. For scientific calculators, * has precedence over +.

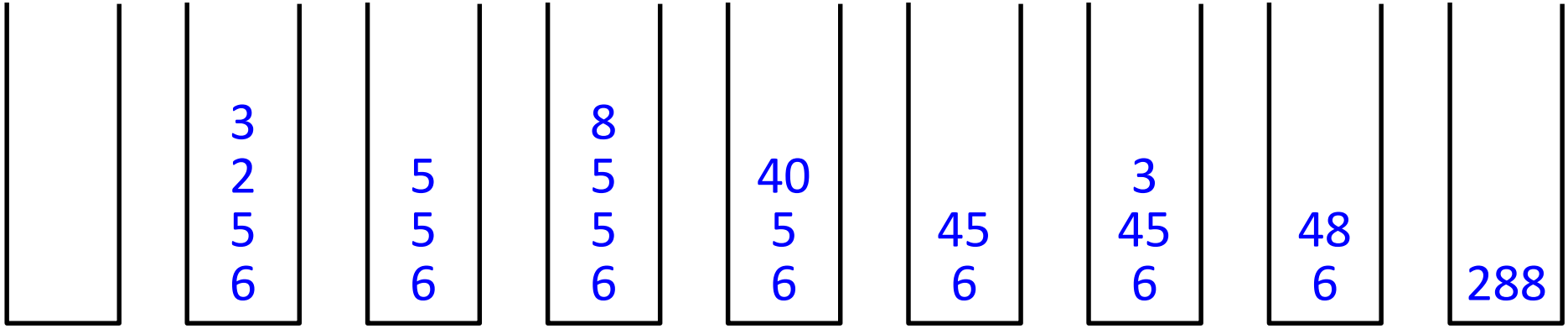
- Postfix (reverse Polish notation) has no ambiguity:

  $8\ 3 *\ 10\ +\ 2\ 4 *\ +$

- Postfix expressions can be evaluated using a stack:
  - each time an operand is read, it is pushed on the stack
  - each time an operator is read, the two top values are popped and operated. The result is push onto the stack

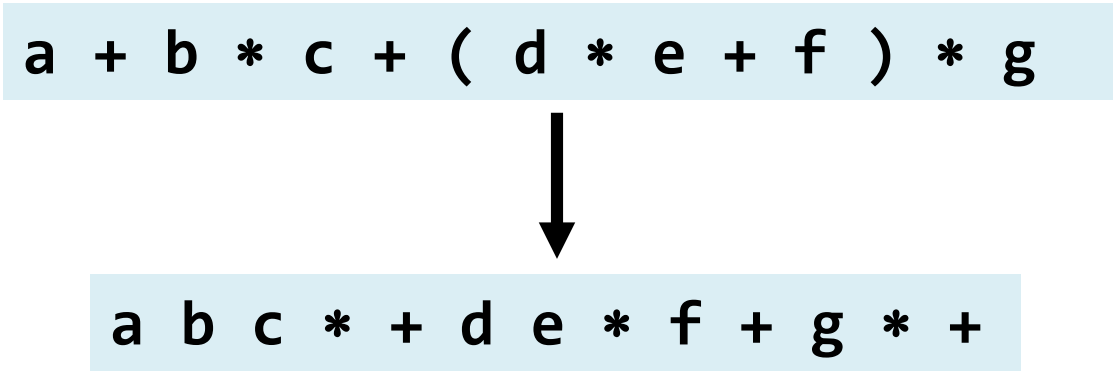
Evaluation of postfix expressions: example

6 5 2 3 + 8 * + 3 + *



push(6)
push(5)
push(2)
push(3)
+
push(8)
*
+
push(3)
+
*

From infix to postfix



Algorithm:

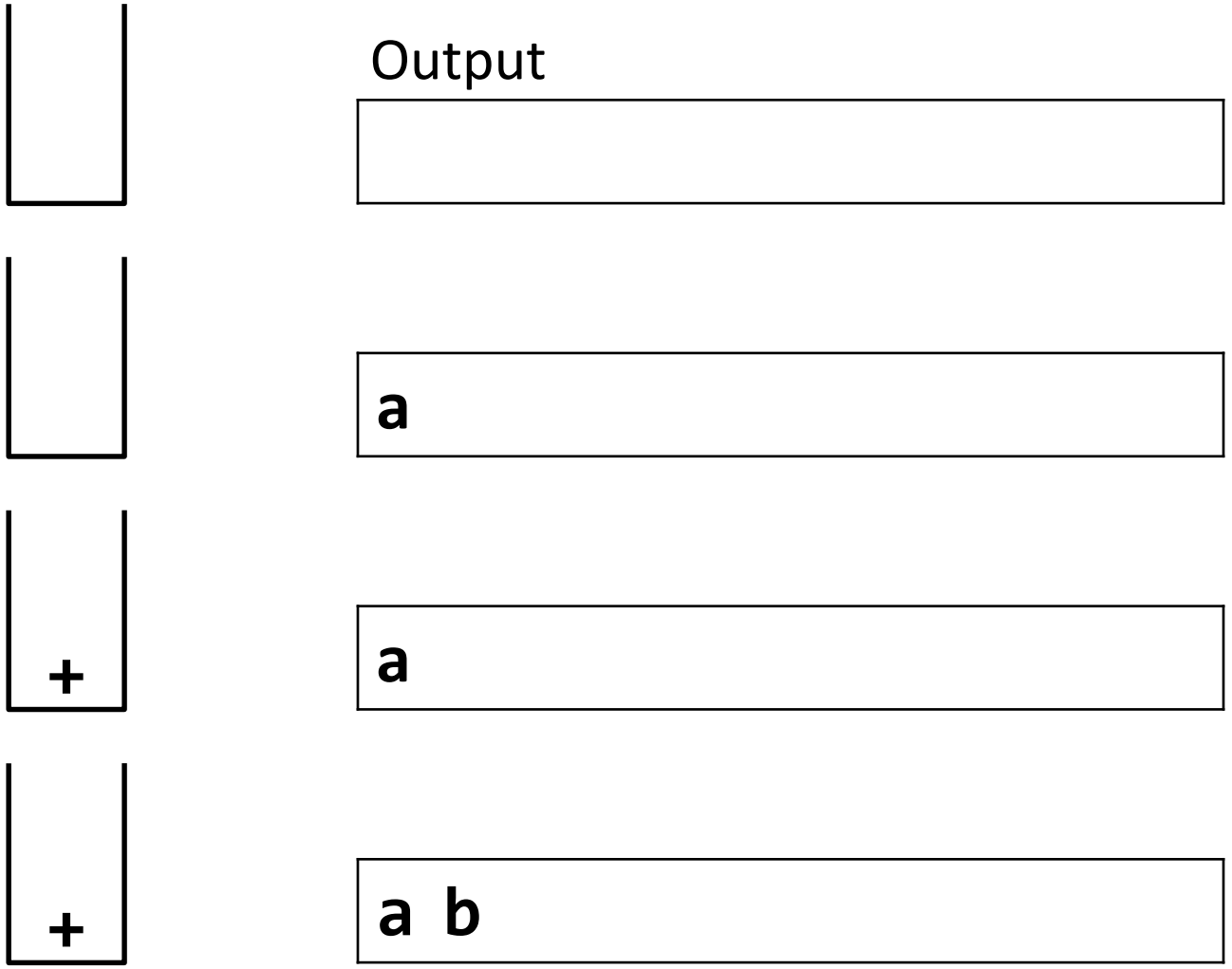
- When an operand is read, write it to the output.
- If we read a right parenthesis, pop the stack writing symbols until we encounter the left parenthesis.
- For any other symbol, i.e., **+ * (**, pop entries and write them until we find an entry with lower priority. After popping, push the symbol onto the stack. Exception: **(** can only be removed when finding a **)**.
- When the end of the input is reached, all symbols in the stack are popped and written onto the output.

From infix to postfix

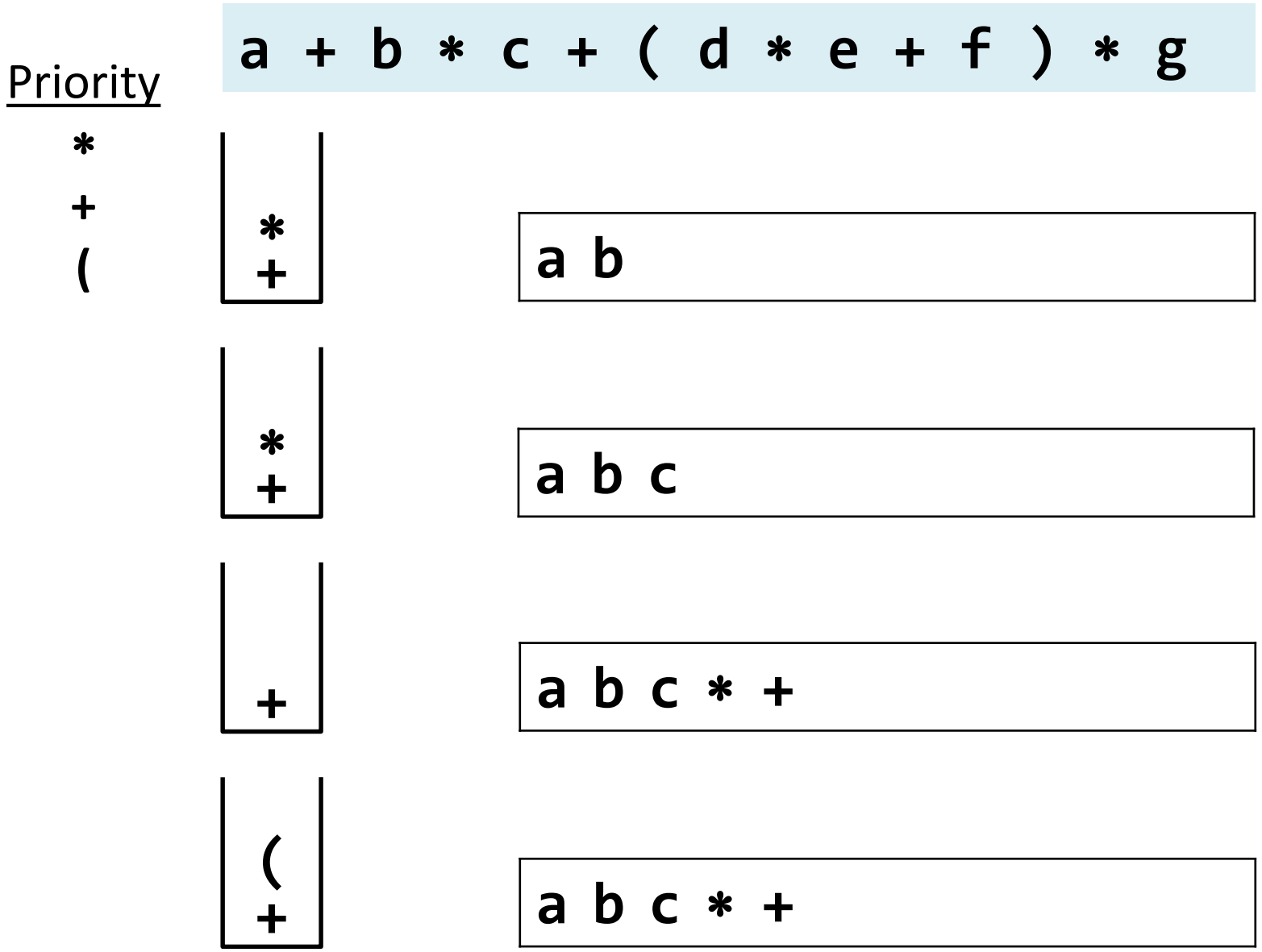
a + b * c + (d * e + f) * g

Priority

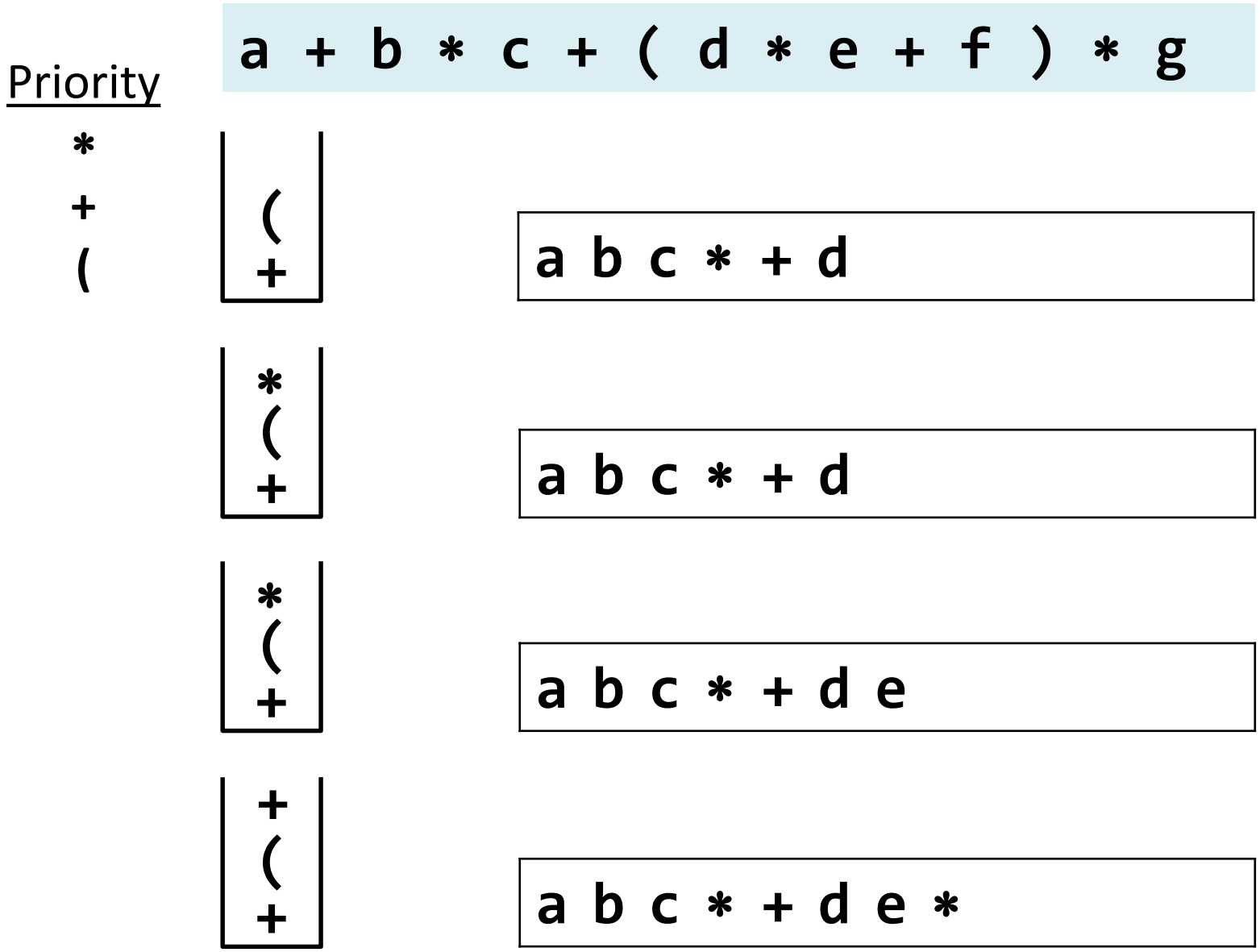
*
+
(



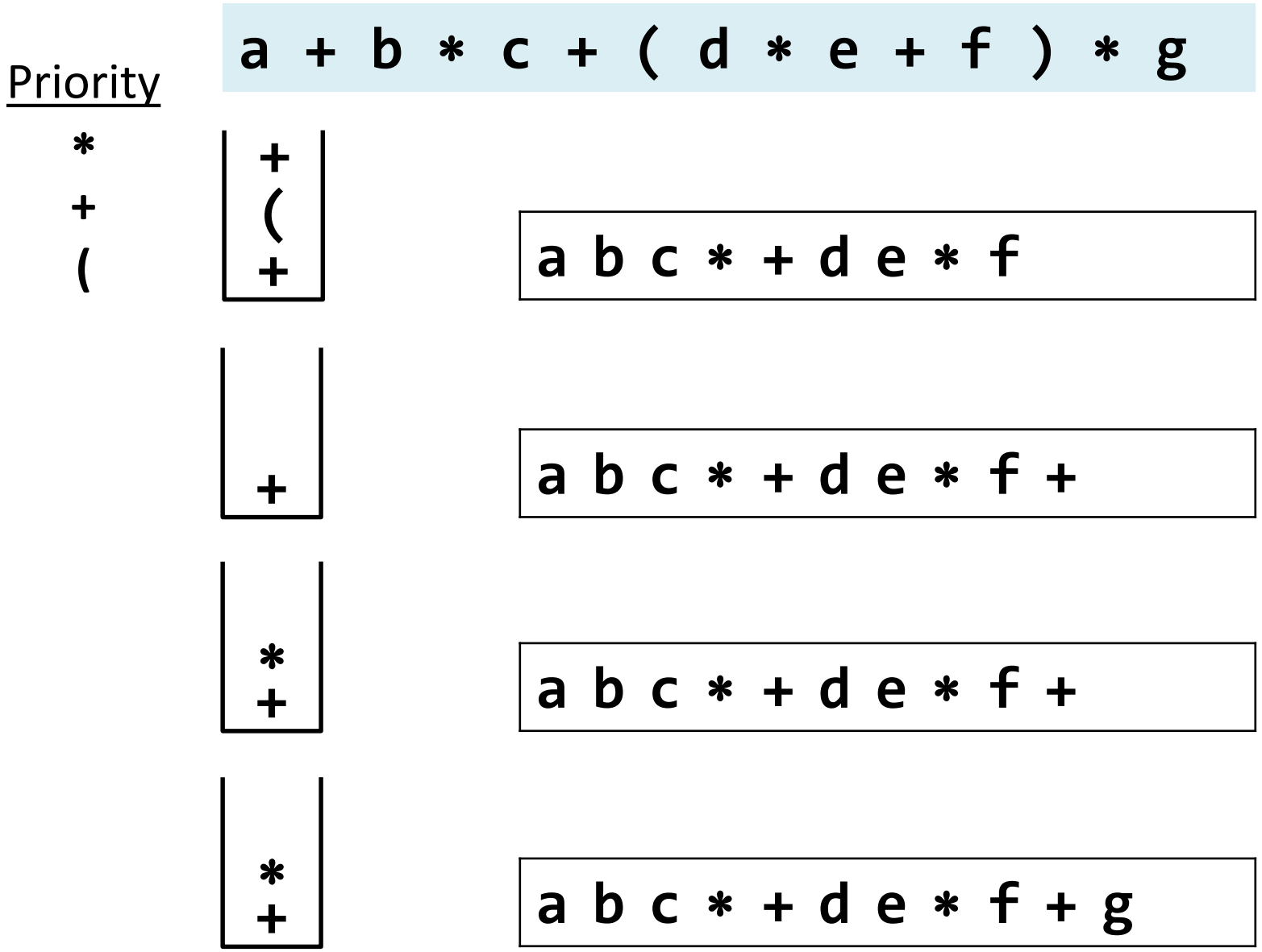
From infix to postfix



From infix to postfix



From infix to postfix



From infix to postfix

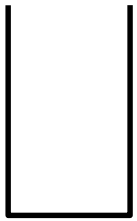
a + b * c + (d * e + f) * g

Priority

*

+

(



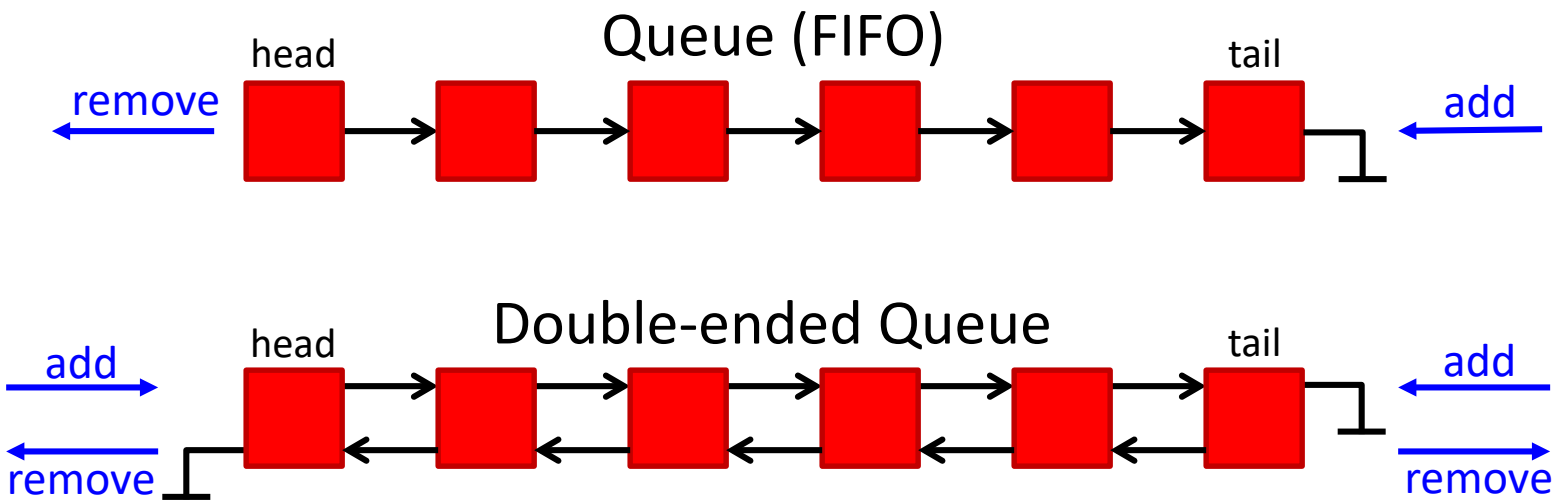
a b c * + d e * f + g * +

Complexity: $O(n)$

Suggested exercise:

- Add substraction (same priority as addition) and division (same priority as multiplication).

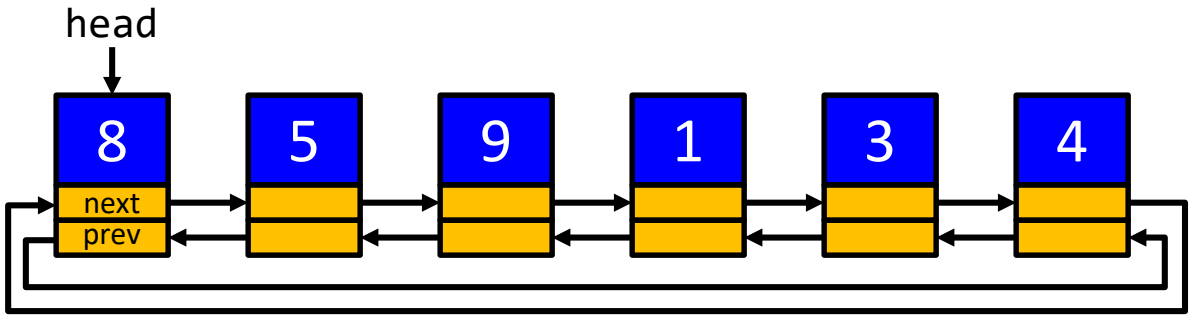
Queues



Queues are usually implemented using references to objects (also called pointers in C/C++). These references allow moving left/right and iterating over the queue.

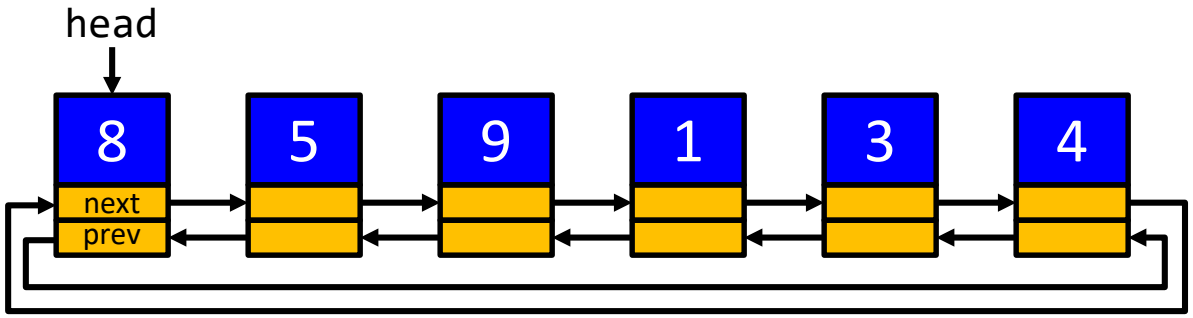
We will explain a toy implementation of a double-ended queue (deque), with the basic functionality to add/remove elements and iterate over them.

Dequeu



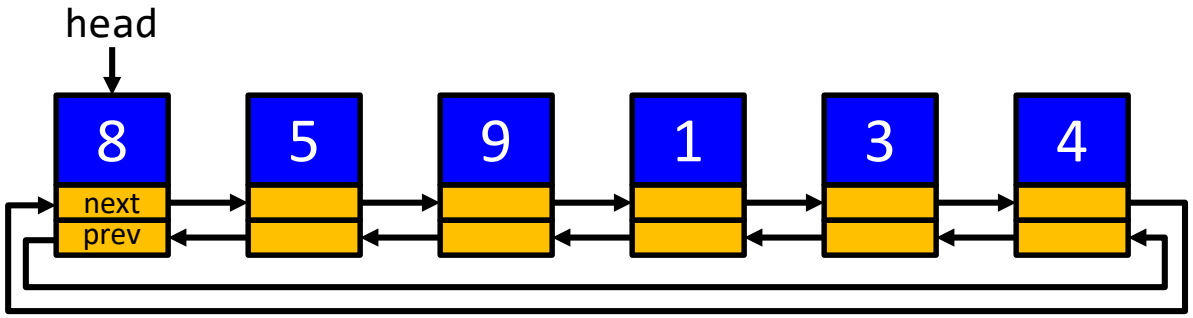
- Implemented as a circular queue with a reference to the head
- Elements can be appended/removed to/from the head or tail
- Operations:
  - **`len(q), q.append(x), q.appendleft(x), q.pop(), q.popleft()`**
  - Access to the i-th element (**`q[0], q[1],q[-1], q[-2],…`**)
  - Iterators: **`for x in q:`**

Dequeu node



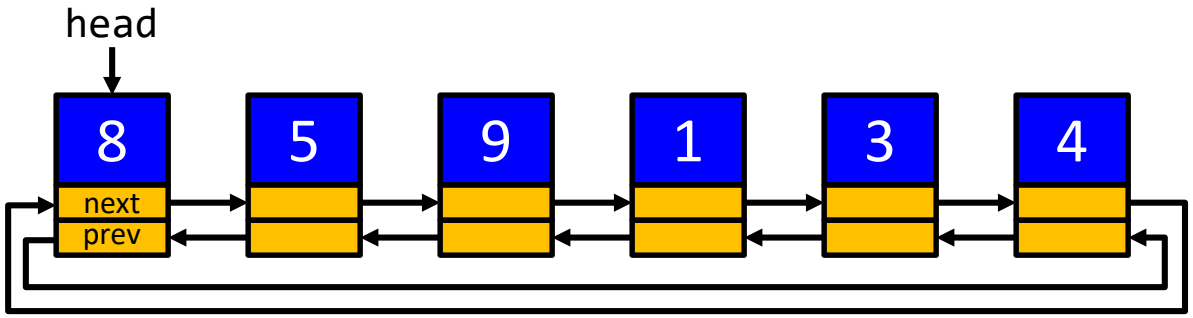

```
from dataclasses import dataclass, field
from typing import TypeVar, Generic, Iterable

T = TypeVar('T')  # Generic type for the deque

@dataclass
class Node(Generic[T]):
    """Internal node of the deque"""
    data: T  # information stored in the node
    next: 'Node[T]' = field(init=False)  # next in the queue
    prev: 'Node[T]' = field(init=False)  # previous in the queue
```

Dequeu attributes



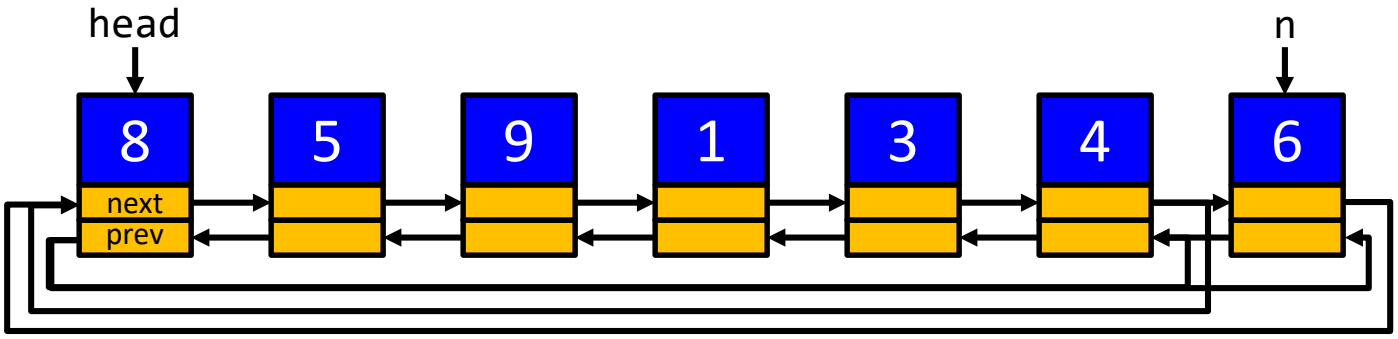

```
class Deque(Generic[T]):
    """Class to represent a double-ended queue"""

    # Attributes of the class
    _head: Node[T]  # reference to the head of the queue
    _n: int  # number of elements in the queue
```

Dequeu: __init__ and __len__



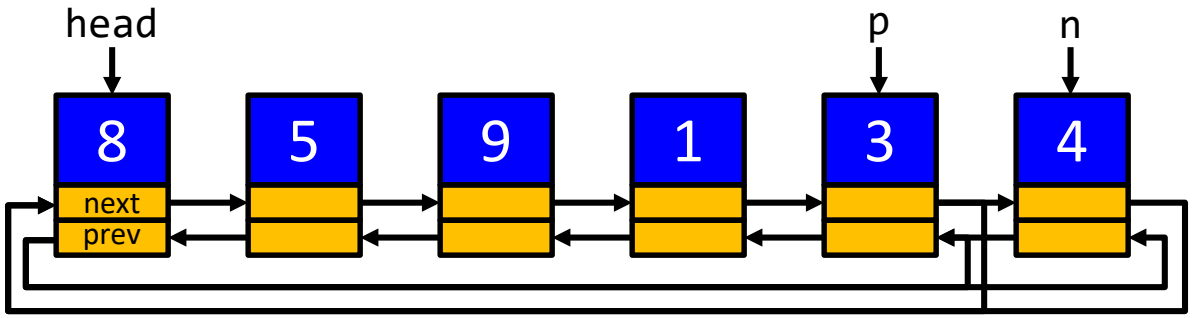

```
class Deque(Generic[T]):

                                    ⋮

    def __init__(self, it: Iterable[T] = list()) -> None:
        """Constructor: initialize from iterable"""
        self._n = 0
        for x in it:
            self.append(x)

    def __len__(self) -> int:
        """Length of the queue"""
        return self._n
```

Dequeu: append



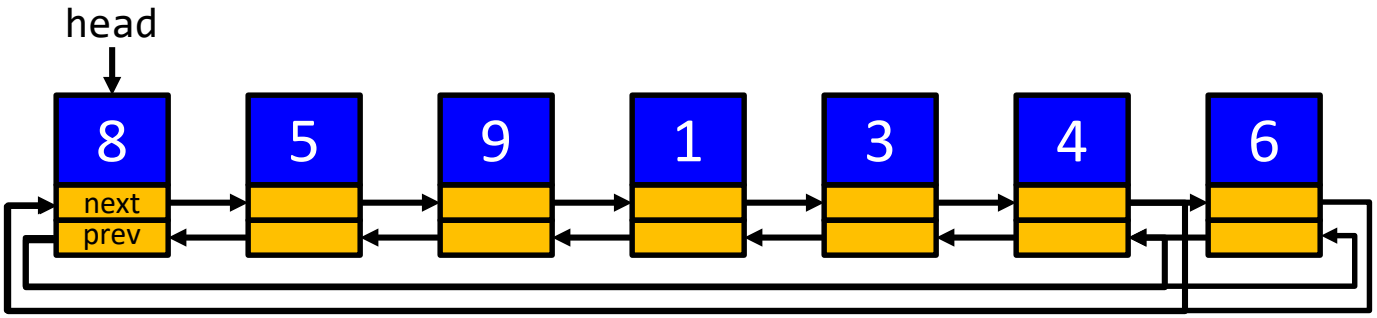

```
class Deque(Generic[T]):

                        ⋮

    def append(self, x: T) -> None:
        """Add element x to the tail of the queue"""
        n = Node(x)
        if self._n == 0:
            self._head = n.next = n.prev = n
        else:
            n.next = self._head
            n.prev = self._head.prev
            n.prev.next = self._head.prev = n
        self._n += 1
```

Dequeu: pop



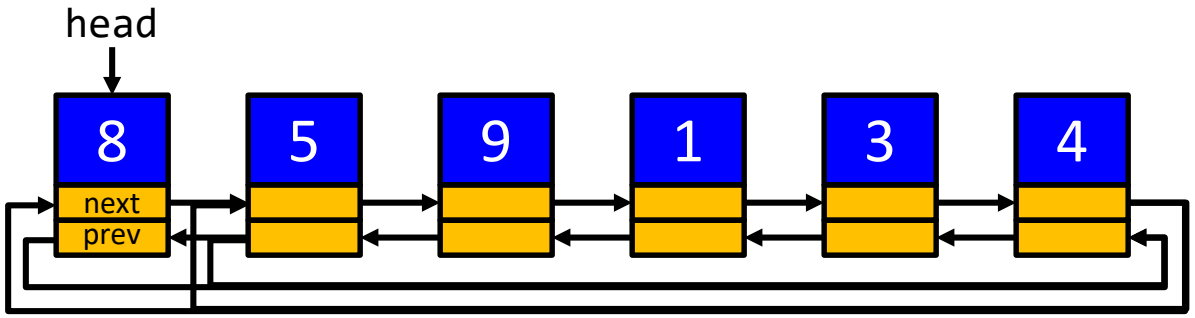

```
class Deque(Generic[T]):

                        ⋮

    def pop(self) -> T:
        """Returns and removes the last element of the queue"""
        if len(self) == 0:
            raise IndexError
        self._n -= 1
        n = self._head.prev
        if self._n > 0:
            p = n.prev
            p.next = self._head
            self._head.prev = p
        return n.data
```

Dequeu: appendleft



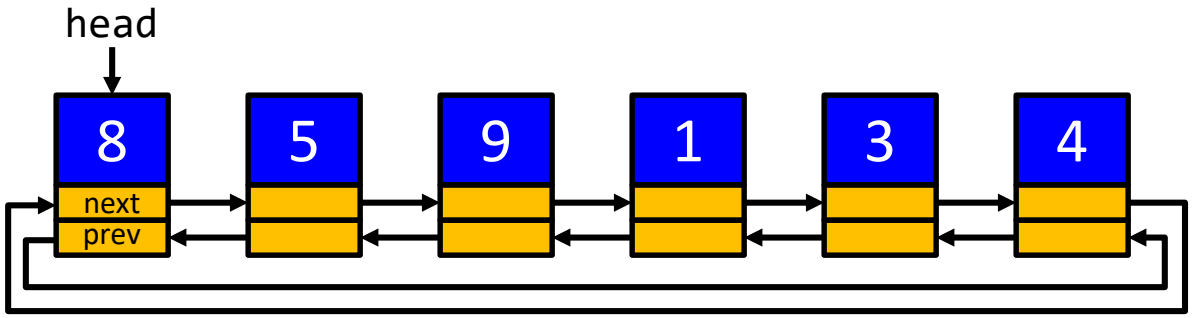

```
class Deque(Generic[T]):

                        ⋮

    def appendleft(self, x: T) -> None:
        """Add element x to the head of the queue"""
        self.append(x)
        self._head = self._head.prev
```

Dequeu: popleft




```
class Deque(Generic[T]):
                                ⋮


    def popleft(self) -> T:
        """Returns and removes the head of the queue"""
        self._head = self._head.next
        return self.pop()
```

Dequeu: __getitem__




```
class Deque(Generic[T]):

                        ⋮
    def __getitem__(self, i: int) -> T:
        """Returns the i-th element (i can be negative)"""
        if i < 0:  # Convert the index into positive
            i = len(self) + i
        if i < 0 or i >= len(self):  # Check if out of bounds
            raise IndexError

        p = self._head
        for _ in range(i):
            p = p.next
        return p.data
```

Can we make it more efficient? How about **q[-1]** in a queue with 10^6 elements?

Deque iterator: how to use it?

```
q: Dequeu[int] = Dequeu([12, 15, 6, -4])

# Visiting the elements of q with an iterator
q_iter: DequeIter[int] = iter(q)
try:
    while True:
        print('', next(q_iter), end='')
except StopIteration:
    print()

# Equivalent code with the same functionality
for x in q:
    print('', x, end='')
print()

# Important: q may have more than one iterator
# Iterators are independent from each other
q_iter1 = iter(q)
q_iter2 = iter(q)
```

Deque iterator: how to use it?

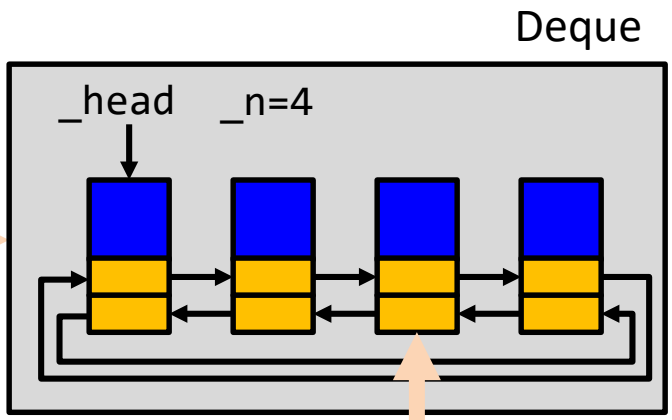
```
q: Deque[int] = Dequeu([12, 15, 6, -4])

q_iter: DequeIter[int] = iter(q)

try:
    while True:
        print('', next(q_iter), end='')
except StopIteration:
    print()
```

q.__iter__()

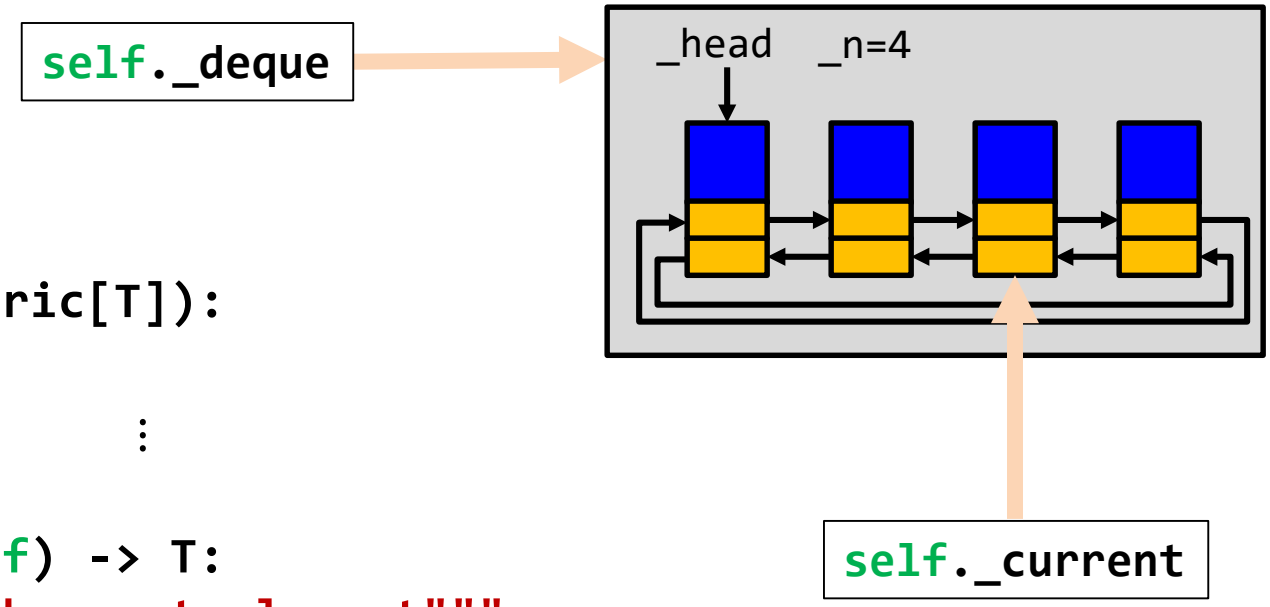
q_iter.__next__()

Deque: iterator

```
class Deque(Generic[T]):
                                    ⋮
    def __iter__(self) -> 'DequeIter[T]':
        """Returns an iterator"""
        return DequeIter(self)


class DequeIter(Generic[T]):
    """Iterator of a deque"""
    _deque: Deque[T]  # The deque
    _current: Node[T] | None  # The current node

    def __init__(self, q: Deque[T]) -> None:
        """Initializes the iterator"""
        self._deque = q
        self._current = q._head if len(q) > 0 else None
```



Deque: iterator




```
class DequeIter(Generic[T]):

                              ⋮

    def __next__(self) -> T:
        """Returns the next element"""
        if self._current is None:
            raise StopIteration
        data = self._current.data
        self._current = self._current.next
        if self._current == self._deque._head:
            self._current = None
        return data
```

Using a deque

```
q: Deque[int] = Deque(range(1,9))  # Initialize with iterable
q.appendleft(0)
print("Number of elements:", len(q))

# Using q[i] (__getitem__). Quadratic cost!
for i in range(len(q)):
    print("  Element", i, " =", q[i])

# Using the iterators (linear cost: much more efficient!)
for i, x in enumerate(q):
    print("  Element", i, " =", x)

print("Removing the last element:", q.pop())
print("Removing the first element:", q.popleft())

print("Remaining elements in the queue:", end='')
for x in q:
    print(' ', x, sep='', end='')
print()
```

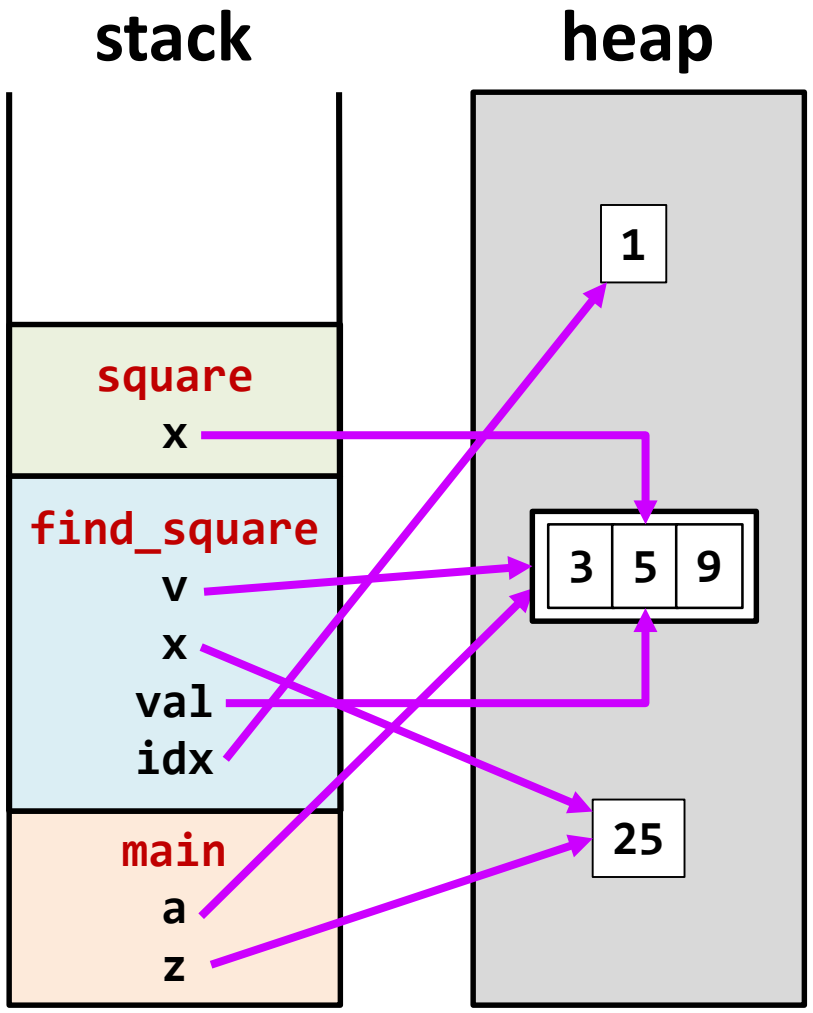
Memory management

- Programming languages have different strategies for memory management

- Typically, there are two structures used for memory management:
  - Stack, for static memory allocation, associated to functions/methods
  - Heap, for dynamic memory allocation

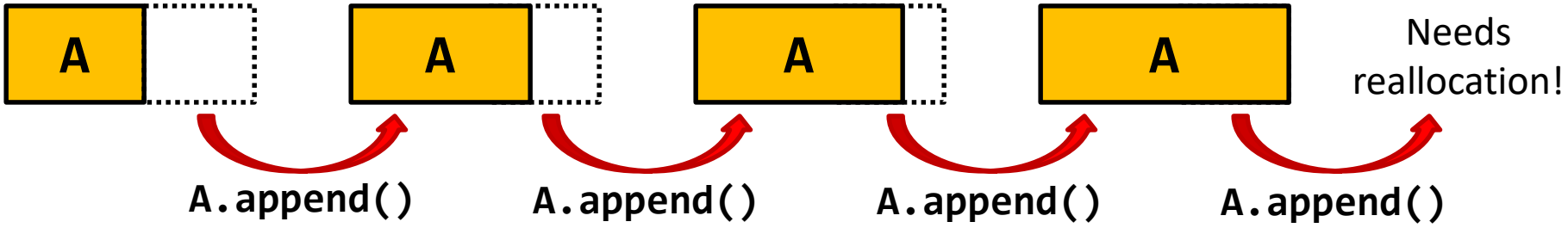
Memory management in Python

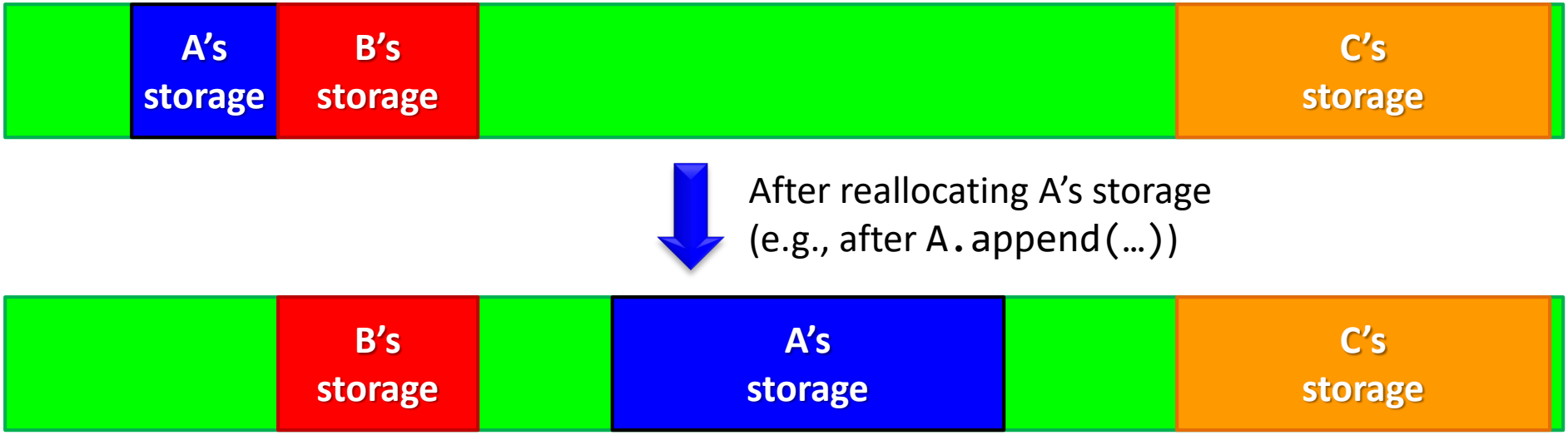
```
def square(x):
    return x*x

def find_square(v, x):
    for idx, val in enumerate(v):
        if x == square(val):
            return idx
    return None

a = [3, 5, 9]
z = 25
print(find_square(a, z))
```



Heap management

Data structures often have some extra allocated memory to avoid frequent reallocations

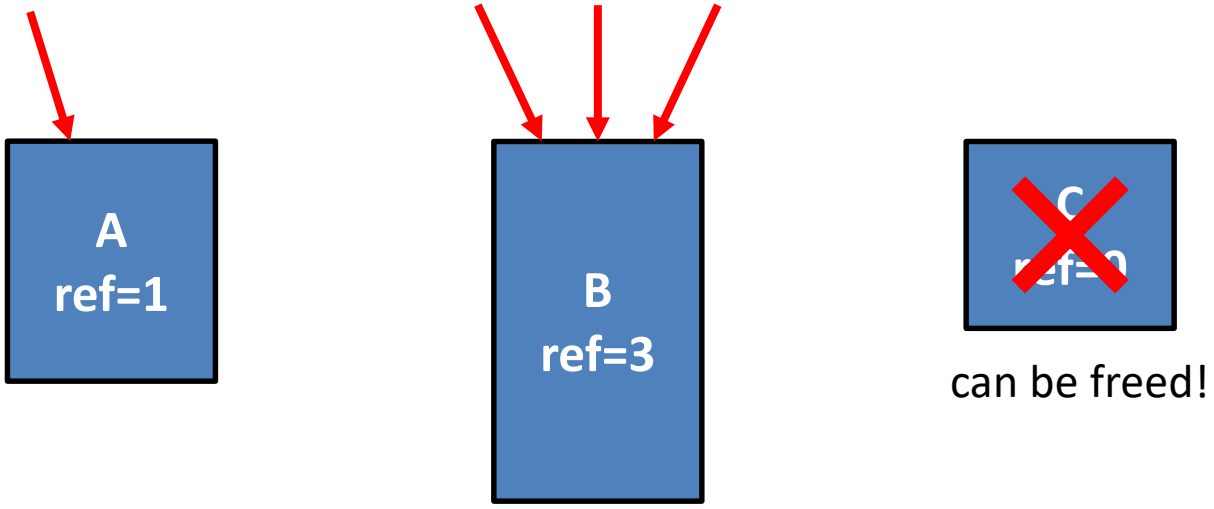




Garbage collection

- Objects may have multiple references. Each object has a reference count.

- Python runs a garbage collector periodically. All objects with zero references are freed.



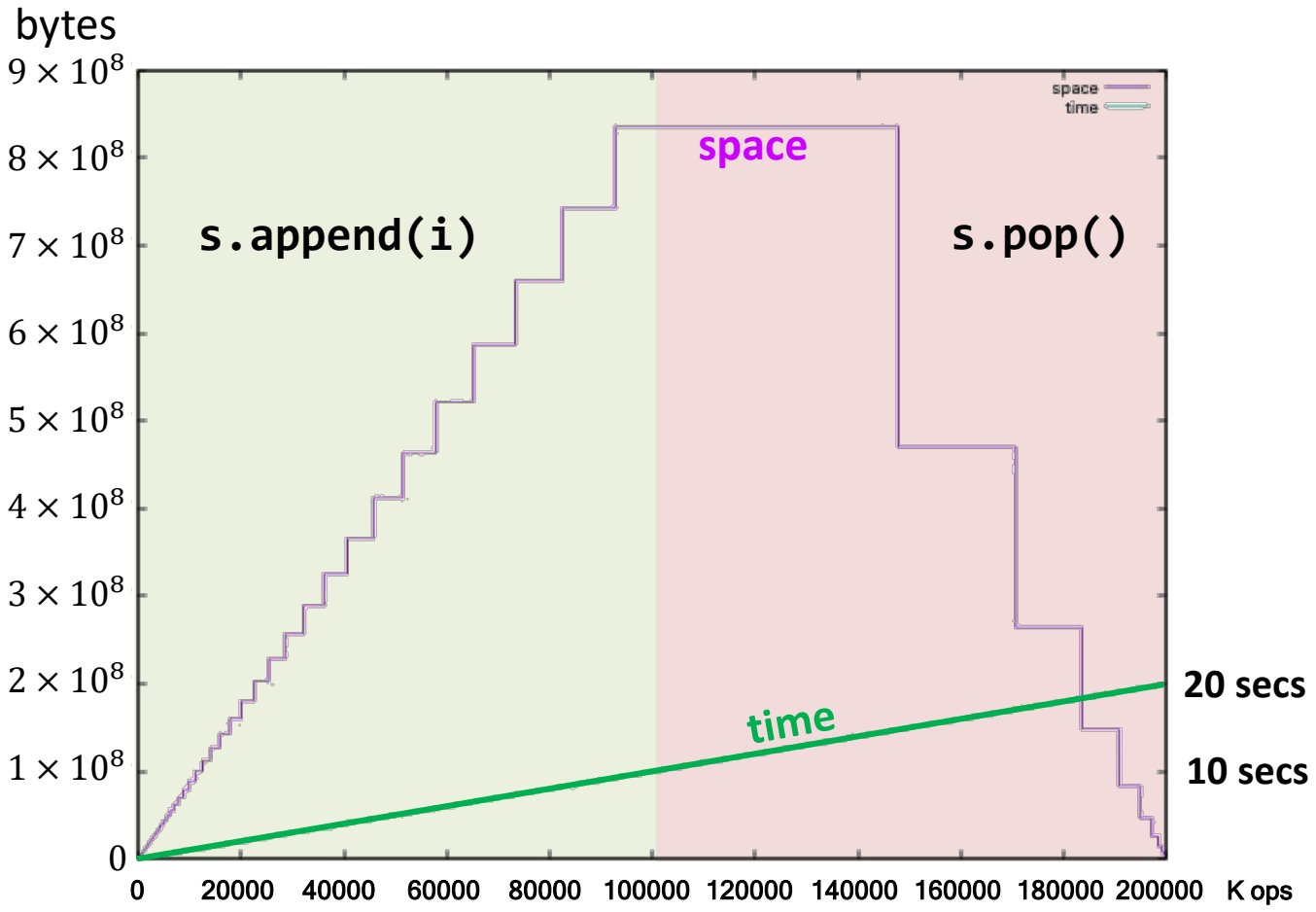
Efficient memory management

```
import sys
import time

t0 = time.time()
n = 100000000
s: list[int] = []

# append n elements
for i in range(n):
    s.append(i)
    if i % 1000 == 0:
        print(i, sys.getsizeof(s), time.time() - t0)

# remove n elements
for i in range(n):
    s.pop()
    if i % 1000 == 0:
        print(n+i, sys.getsizeof(s), time.time() - t0)
```

Efficient memory management

Avoid managing memory at every resizing operation

Use extra memory to amortize the effort in allocating/deallocating memory



EXERCISES

Stack: Interleaved push/pop operations

Suppose that an intermixed sequence of push and pop operations are performed. The pushes push the integers 0 through 9 in order; the pops print out the return value. Which of the following sequences could not occur?

a) 4 3 2 1 0 9 8 7 6 5
b) 4 6 8 7 5 3 2 9 0 1
c) 2 5 6 7 4 8 9 3 1 0
d) 4 3 2 1 0 5 6 7 8 9

Source: Robert Sedgewick, Computer Science 126, Princeton University.

Middle element of a stack

Design the class **`MidStack`** implementing a stack with the following operations:

- Push/pop: the usual operations on a stack.
- FindMiddle: returns the value of the element in the middle.
- DeleteMiddle: deletes the element in the middle.

All the operations must be executed in $O(1)$ time.

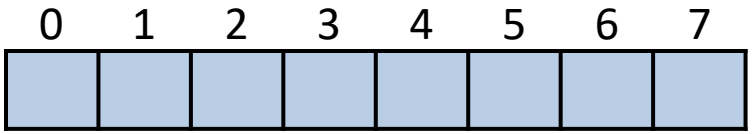
Suggestion: use some container of the STL to implement it.

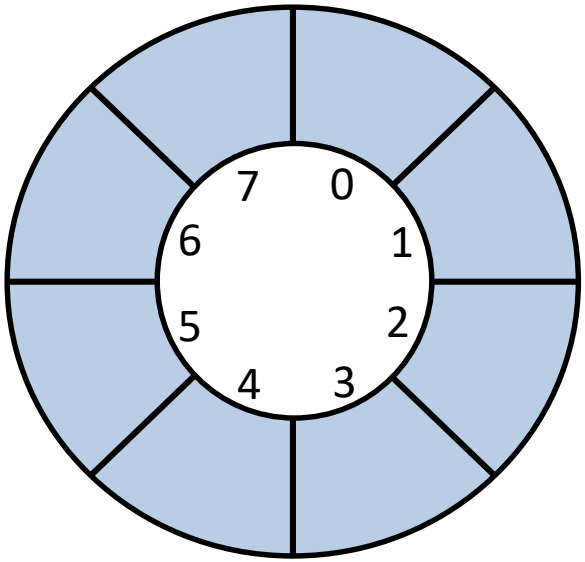
Note: if the stack has n elements at locations $0..n-1$, where 0 is the location at the bottom, the middle element is the one at location $\lfloor (n-1)/2 \rfloor$.

Queues implemented as circular buffers

- Design the class queue implemented with a circular buffer (using a Python list):
  - The add/remove operations should run in constant time.
  - The class should have a constructor with a parameter n that indicates the maximum number of elements in the queue.

- Consider the design of a variable-size queue using a circular buffer. Discuss how the implementation should be modified.





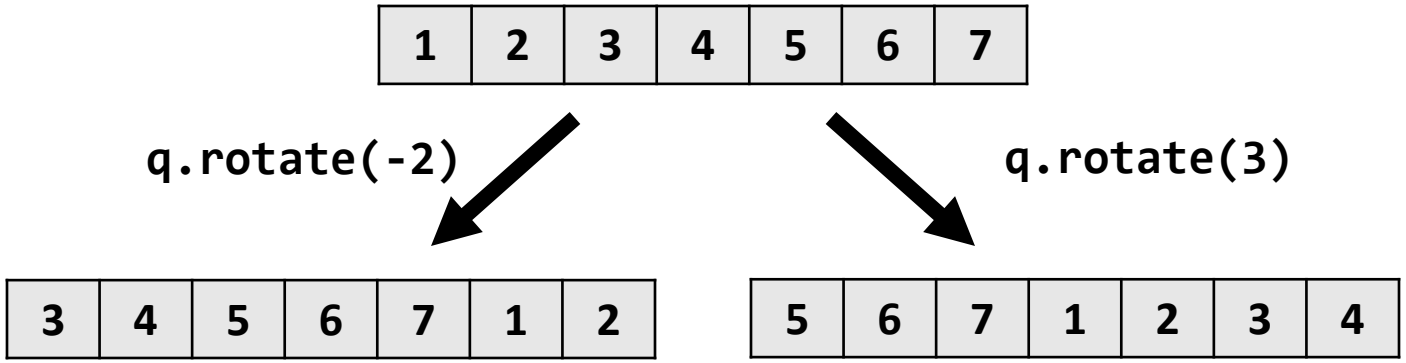
Reverse

Design the method `reverse()` that reverses the contents of a deque:

- No auxiliary data structures should be used.
- No copies of the elements should be performed.

Rotate and Josephus

- Design the method **`q.rotate(n)`** that rotates a deque **`n`** positions to the right (if positive) or **`–n`** positions to the left (if negative). Make the method efficient (e.g., assume that n can be large)



- Solve the Josephus problem, for n people and executing every k-th person, using a deque:

  https://en.wikipedia.org/wiki/Josephus_problem

Merge and quick sort

- Design the method **`q.merge(other: Deque)`** that merges the deque with another deque, assuming that both are sorted. Assume that a pair of elements can be compared with the operator <. After merging, all the elements must have been transferred to **`q`** (**`other`** becomes empty).

- Design the method **`q.sort()`** that sorts the deque according to the < operator. Consider merge sort and quick sort as possible algorithms.