```
def TanpaNone(arr):
    isi = []
    for i in arr:
        if i != None:
            isi.append(i)
    return isi

class ArrayStack:
    """LIFO Stack implementation using a Python list as underlying
storage."""

    def __init__ (self):
        """Create an empty stack."""
        self._data = [] # nonpublic list instance

    def __len__(self):
        """Return the number of elements in the stack."""
        return len(self._data)

    def is_empty(self):
        """Return True if the stack is empty."""
        return len(self._data) == 0

    def push(self, e):
        """Add element e to the top of the stack."""
        self._data.append(e) # new item stored at end of list

    def top(self):
        """Return (but do not remove) the element at the top of the
stack.
        Raise Empty exception if the stack is empty."""
        if self.is_empty():
            raise Empty("Stack is empty")
        return self._data[-1] # the last item in the list

    def pop(self):
        """Remove and return the element from the top of the stack
(i.e., LIFO).
        Raise Empty exception if the stack is empty."""
        if self.is_empty( ):
            raise Empty("Stack is empty")
        return self._data.pop( ) # remove last item from list

S = ArrayStack()
print(f'Stack Saat ini: {S._data}')
S.push(5), print(f'S.push(5) | Stack Saat ini: {S._data}')
S.push(3), print(f'S.push(3) | Stack Saat ini: {S._data}')
S.pop(), print(f'S.pop     | Stack Saat ini: {S._data}')
S.push(2), print(f'S.push(2) | Stack Saat ini: {S._data}')
S.push(8), print(f'S.push(8) | Stack Saat ini: {S._data}')
```

```
S.pop(), print(f'S.pop     | Stack Saat ini: {S._data}')
S.pop(), print(f'S.pop     | Stack Saat ini: {S._data}')
S.push(9), print(f'S.push(9) | Stack Saat ini: {S._data}')
S.push(1), print(f'S.push(1) | Stack Saat ini: {S._data}')
S.pop(), print(f'S.pop     | Stack Saat ini: {S._data}')
S.push(7), print(f'S.push(7) | Stack Saat ini: {S._data}')
S.push(6), print(f'S.push(6) | Stack Saat ini: {S._data}')
S.pop(), print(f'S.pop     | Stack Saat ini: {S._data}')
S.pop(), print(f'S.pop     | Stack Saat ini: {S._data}')
S.push(4), print(f'S.push(4) | Stack Saat ini: {S._data}')
S.pop(), print(f'S.pop     | Stack Saat ini: {S._data}')
S.pop(), print(f'S.pop     | Stack Saat ini: {S._data}')
S._data
```

```
Stack Saat ini: []
S.push(5) | Stack Saat ini: [5]
S.push(3) | Stack Saat ini: [5, 3]
S.pop     | Stack Saat ini: [5]
S.push(2) | Stack Saat ini: [5, 2]
S.push(8) | Stack Saat ini: [5, 2, 8]
S.pop     | Stack Saat ini: [5, 2]
S.pop     | Stack Saat ini: [5]
S.push(9) | Stack Saat ini: [5, 9]
S.push(1) | Stack Saat ini: [5, 9, 1]
S.pop     | Stack Saat ini: [5, 9]
S.push(7) | Stack Saat ini: [5, 9, 7]
S.push(6) | Stack Saat ini: [5, 9, 7, 6]
S.pop     | Stack Saat ini: [5, 9, 7]
S.pop     | Stack Saat ini: [5, 9]
S.push(4) | Stack Saat ini: [5, 9, 4]
S.pop     | Stack Saat ini: [5, 9]
S.pop     | Stack Saat ini: [5]

[5]
```

6.7

```
class ArrayQueue:
    """FIFO queue implementation using a Python list as underlying
storage."""
    DEFAULT_CAPACITY = 10 # moderate capacity for all new queues

    def __init__(self):
        """Create an empty queue."""
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0

    def __len__(self):
        """Return the number of elements in the queue."""
```

```
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the
queue.
        Raise Empty exception if the queue is empty."""
        if self.is_empty():
            raise Empty('Queue is empty')
        return self._data[self._front]

    def dequeue(self):
        """Remove and return the first element of the queue (i.e.,
FIFO).
        Raise Empty exception if the queue is empty."""
        if self.is_empty():
            raise Empty('Queue is empty')
        answer = self._data[self._front]
        self._data[self._front] = None # help garbage collection
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        return answer

    def enqueue(self, e):
        """Add an element to the back of queue."""
        if self._size == len(self._data):
            self._resize(2*len(self.data)) # double the array size
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1

    def _resize(self, cap): # we assume cap >= len(self)
        """Resize to a new list of capacity >= len(self)."""
        old = self._data # keep track of existing list
        self._data = [None]*cap # allocate list with new capacity
        walk = self._front
        for k in range(self._size): # only consider existing elements
            self._data[k] = old[walk] # intentionally shift indices
            walk = (1 + walk) % len(old) # use old size as modulus
        self._front = 0 # front has been realigned

AQ = ArrayQueue()
print(f'isi Queue saat ini: {AQ._data} | {TanpaNone(AQ._data)}')
AQ.enqueue(5), print(f'AQ.enqueue(5) | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.enqueue(3), print(f'AQ.enqueue(3) | isi Queue saat ini:
{TanpaNone(AQ._data)}')
```

```
AQ.dequeue(), print(f'AQ.dequeue()  | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.enqueue(2), print(f'AQ.enqueue(2) | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.enqueue(8), print(f'AQ.enqueue(8) | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.dequeue(), print(f'AQ.dequeue()  | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.dequeue(), print(f'AQ.dequeue()  | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.enqueue(9), print(f'AQ.enqueue(9) | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.enqueue(1), print(f'AQ.enqueue(1) | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.dequeue(), print(f'AQ.dequeue()  | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.enqueue(7), print(f'AQ.enqueue(7) | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.enqueue(6), print(f'AQ.enqueue(6) | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.dequeue(), print(f'AQ.dequeue()  | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.dequeue(), print(f'AQ.dequeue()  | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.enqueue(4), print(f'AQ.enqueue(4) | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.dequeue(), print(f'AQ.dequeue()  | isi Queue saat ini:
{TanpaNone(AQ._data)}')
AQ.dequeue(), print(f'AQ.dequeue()  | isi Queue saat ini:
{TanpaNone(AQ._data)}')
print(f'hasil akhir queue: {AQ._data} | {TanpaNone(AQ._data)}')
```

```
isi Queue saat ini: [None, None, None, None, None, None, None, None,
None, None] | []
AQ.enqueue(5) | isi Queue saat ini: [5]
AQ.enqueue(3) | isi Queue saat ini: [5, 3]
AQ.dequeue()  | isi Queue saat ini: [3]
AQ.enqueue(2) | isi Queue saat ini: [3, 2]
AQ.enqueue(8) | isi Queue saat ini: [3, 2, 8]
AQ.dequeue()  | isi Queue saat ini: [2, 8]
AQ.dequeue()  | isi Queue saat ini: [8]
AQ.enqueue(9) | isi Queue saat ini: [8, 9]
AQ.enqueue(1) | isi Queue saat ini: [8, 9, 1]
AQ.dequeue()  | isi Queue saat ini: [9, 1]
AQ.enqueue(7) | isi Queue saat ini: [9, 1, 7]
AQ.enqueue(6) | isi Queue saat ini: [9, 1, 7, 6]
AQ.dequeue()  | isi Queue saat ini: [1, 7, 6]
AQ.dequeue()  | isi Queue saat ini: [7, 6]
AQ.enqueue(4) | isi Queue saat ini: [7, 6, 4]
AQ.dequeue()  | isi Queue saat ini: [6, 4]
```

```
AQ.dequeue()  | isi Queue saat ini: [4]
hasil akhir queue: [None, None, None, None, None, None, None, None, 4,
None] | [4]
```

6.12

```
class ArrayDeque():
    """Deque implementation using python list"""

    def __init__(self):
        """Create an empty list"""
        self._data = []

    def __len__(self):
        """Return the number elements of deque"""
        return len(self._data)

    def is_empty(self):
        """Return True if the stack is empty."""
        return len(self._data) == 0

    def add_first(self, e):
        """Add an element to the front of deque"""
        self._data.insert(0, e)

    def add_last(self, e):
        """Add an element to the back of deque"""
        self._data.append(e)

    def delete_first(self):
        """Remove an element at the front"""
        if self.is_empty():
            raise Empty('Deque is empty')
        del self._data[0]

    def delete_last(self):
        """Remove an element at the back"""
        if self.is_empty():
            raise Empty('Deque is empty')
        return self._data.pop()

    def first(self):
        """Return (but do not remove) the element at the front of the
deque.
        Raise Empty exception if the queue is empty."""
        if self.is_empty():
            raise Empty('Deque is empty')
        return self._data[0]

    def last(self):
```

```
        """Return (but do not remove) the element at the back of the
deque.
        Raise Empty exception if the queue is empty."""
        if self.is_empty():
            raise Empty('Deque is empty')
        return self._data[-1]

Deku = ArrayDeque()
print(f'isi deque saat ini: {Deku._data}')
Deku.add_first(4), print(f'Deku.add_first(4)   | isi deque saat ini:
{Deku._data}')
Deku.add_last(8), print(f'Deku.add_last(4)    | isi deque saat ini:
{Deku._data}')
Deku.add_last(9), print(f'Deku.add_last(4)    | isi deque saat ini:
{Deku._data}')
Deku.add_first(5), print(f'Deku.add_first(5)   | isi deque saat ini:
{Deku._data}')
#Deku.back()
Deku.last(), print(f'Deku.last()         | isi deque saat ini:
{Deku._data}')
Deku.delete_first(), print(f'Deku.delete_first() | isi deque saat ini:
{Deku._data}')
Deku.delete_last(), print(f'Deku.delete_last()  | isi deque saat ini:
{Deku._data}')
Deku.add_last(7), print(f'Deku.add_last(7)    | isi deque saat ini:
{Deku._data}')
Deku.first(), print(f'Deku.delete_first() | isi deque saat ini:
{Deku._data}')
Deku.last(), print(f'Deku.last()         | isi deque saat ini:
{Deku._data}')
Deku.add_last(6), print(f'Deku.add_last(6)    | isi deque saat ini:
{Deku._data}')
Deku.delete_first(), print(f'Deku.delete_first() | isi deque saat ini:
{Deku._data}')
Deku.delete_first(), print(f'Deku.delete_first() | isi deque saat ini:
{Deku._data}')
Deku._data
```

```
isi deque saat ini: []
Deku.add_first(4)   | isi deque saat ini: [4]
Deku.add_last(4)    | isi deque saat ini: [4, 8]
Deku.add_last(4)    | isi deque saat ini: [4, 8, 9]
Deku.add_first(5)   | isi deque saat ini: [5, 4, 8, 9]
Deku.last()         | isi deque saat ini: [5, 4, 8, 9]
Deku.delete_first() | isi deque saat ini: [4, 8, 9]
Deku.delete_last()  | isi deque saat ini: [4, 8]
Deku.add_last(7)    | isi deque saat ini: [4, 8, 7]
Deku.delete_first() | isi deque saat ini: [4, 8, 7]
Deku.last()         | isi deque saat ini: [4, 8, 7]
Deku.add_last(6)    | isi deque saat ini: [4, 8, 7, 6]
```

```
Deku.delete_first() | isi deque saat ini: [8, 7, 6]
Deku.delete_first() | isi deque saat ini: [7, 6]

[7, 6]
```

6.10

KASUS 1: ketika melakukan queuing melebihi jumlah wadah yang tersedia dan melakukan Dequeing sebelumnya

- Pada loop normal (sebelum perubahan)

```
def _resize(self, cap): # we assume cap >= len(self)
        """Resize to a new list of capacity >= len(self)."""
        old = self._data # keep track of existing list
        self._data = [None]*cap # allocate list with new capacity
        walk = self._front
        for k in range(self._size): # only consider existing elements
            self._data[k] = old[walk] # intentionally shift indices
            walk = (1 + walk) % len(old) # use old size as modulus
        self._front = 0 # front has been realigned
```

Ketika kita ingin memasukan elemen baru dan wadah masih berisi None di sebelah kanan elemen terakhir yang bukan None, elemen akan ditaruh sesuai dengan wadah yang kosong tersebut. Namun ketika semua wadah di sebelah kanan elemen terakhir yang bukan None sudah terisi semua, elemen yang akan dimasukkan akan diletakkan di None sebelah kiri elemen2 yang bukan none. Selanjutnya, jika kedua kejadian tadi sudah terjadi dan kita ingin memasukkan elemen lagi, element dengan indeks front akan dipindahkan ke indeks ke-0 dan semua selanjutnya akan disusun di sebelah kanan elemen tadi, kemudian panjang dari array akan dikali 2 lalu elemen tadi yang ingin kita masukkan akan diletakkan di indeks ke-11, terakhir variabel front diupdate ulang menjadi 0.

- Namun pada loop yang sudah dilakukan perubahan

```
def _resize(self, cap): # we assume cap >= len(self)
        """Resize to a new list of capacity >= len(self)."""
        old = self._data # keep track of existing list
        self._data = [None]*cap # allocate list with new capacity
        walk = self._front
        for k in range(self._size): # only consider existing elements
            self._data[k] = old[k] # intentionally shift indices
            walk = (1 + walk) % len(old) # use old size as modulus
        self._front = 0 # front has been realigned
```

Ketika kita ingin memasukan elemen baru dan wadah masih berisi None di sebelah kanan elemen terakhir yang bukan None, elemen akan ditaruh sesuai dengan wadah yang kosong tersebut. Namun ketika semua wadah di sebelah kanan elemen terakhir yang bukan None sudah terisi semua, elemen yang akan dimasukkan akan diletakkan di None sebelah kiri elemen2 yang bukan none. Selanjutnya, jika kedua kejadian tadi sudah terjadi dan kita ingin memasukkan elemen lagi, yang terjadi hanya kita memperpanjang array menjadi 2x lipat kemudian

memasukkan elemen yang baru tersebut ke indeks ke-11 tanpa mengubah urutan array sebelumnya, selanjutnya variabel front diupdate menjadi 0. hal ini akan menyebabkan nilai front salah, karena array pada indeks ke-0 sampai ke-9 tidak berubah namun front diubah.

KASUS 2: Ketika kita akan melakukan resize secara manual:

- List baru hanya memotong list lama menjadi sepanjang input(cap), kemudian merubah self._front menjadi 0.