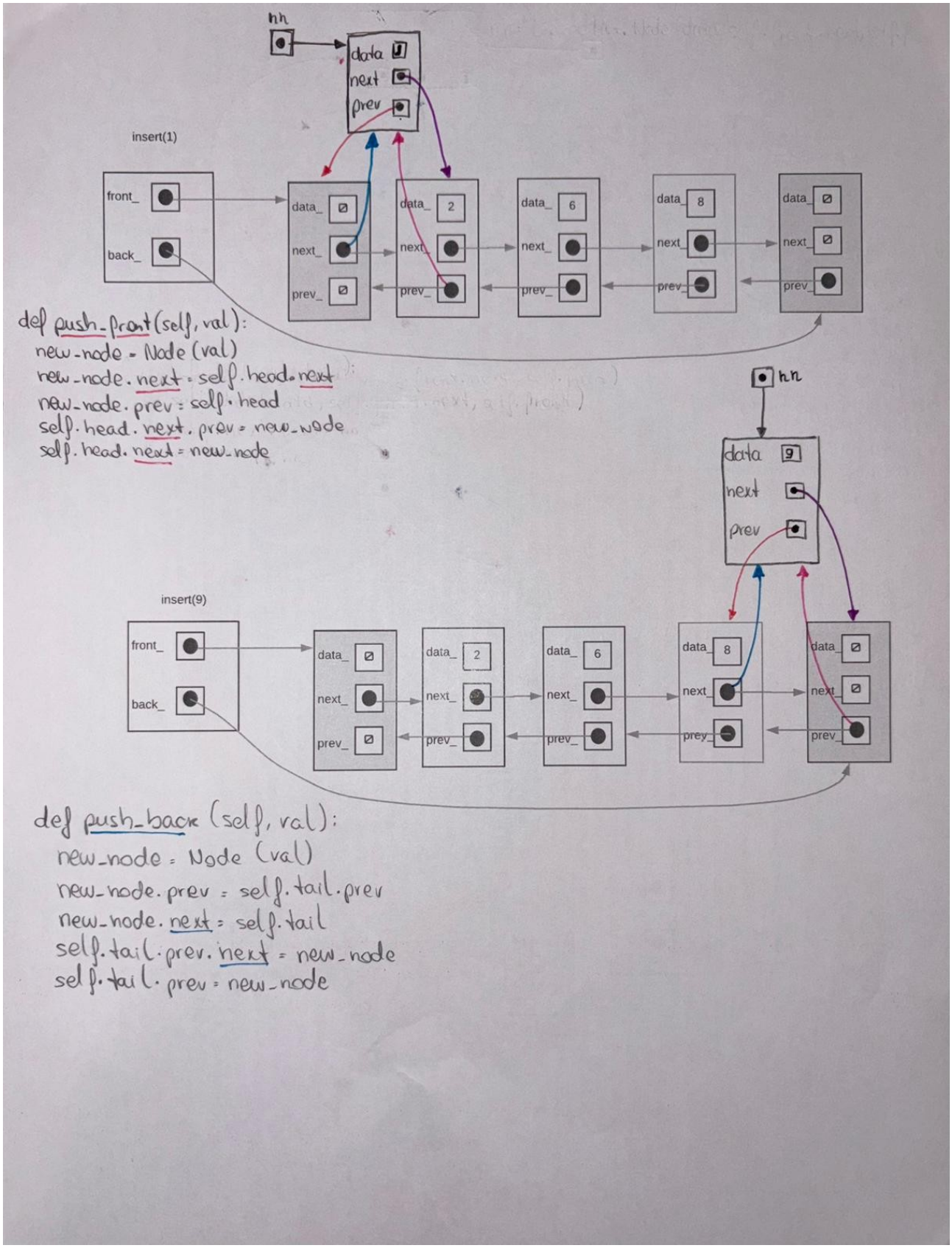
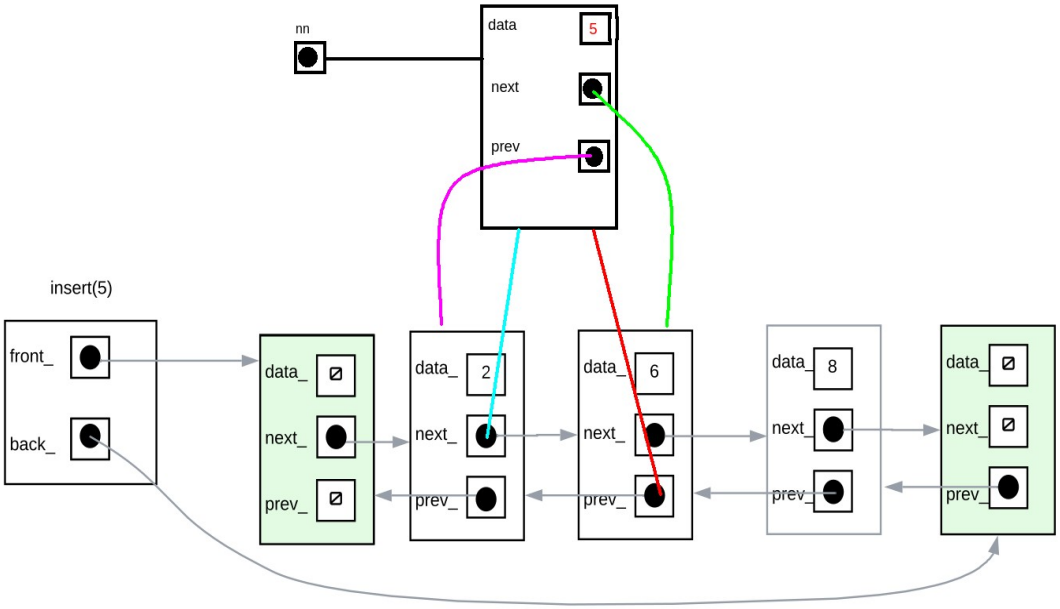
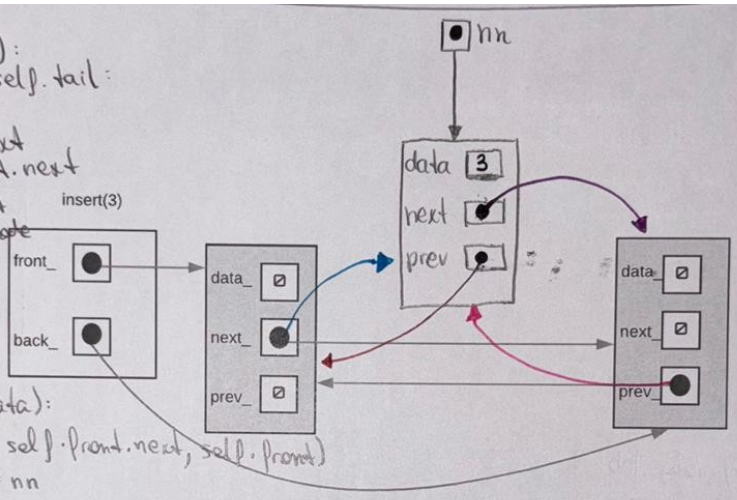

nn
data 1
next
prev
insert(1)
front_
back_
data_
next_
prev_
2
6
8
def push_front(self, val):
new_node = Node(val)
new_node.next = self.head.next
new_node.prev = self.head
self.head.next.prev = new_node
self.head.next = new_node
nn
data 9
next
prev
insert(9)
def push_back(self, val):
new_node = Node(val)
new_node.prev = self.tail.prev
new_node.next = self.tail
self.tail.prev.next = new_node
self.tail.prev = new_node

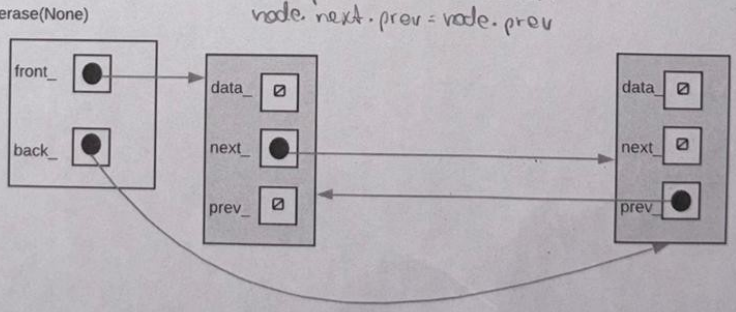


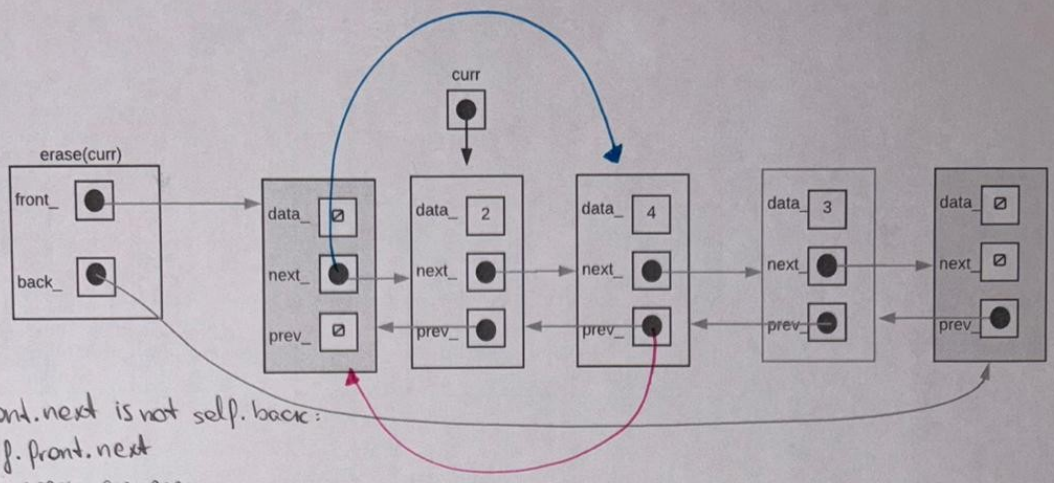

```
current = self.head
for _ in range(position):
    if current.next == self.tail:
        break
    current = current.next
new_node.next = current.next
new_node.prev = current
current.next.prev = newnode
current.next = new_node.
```



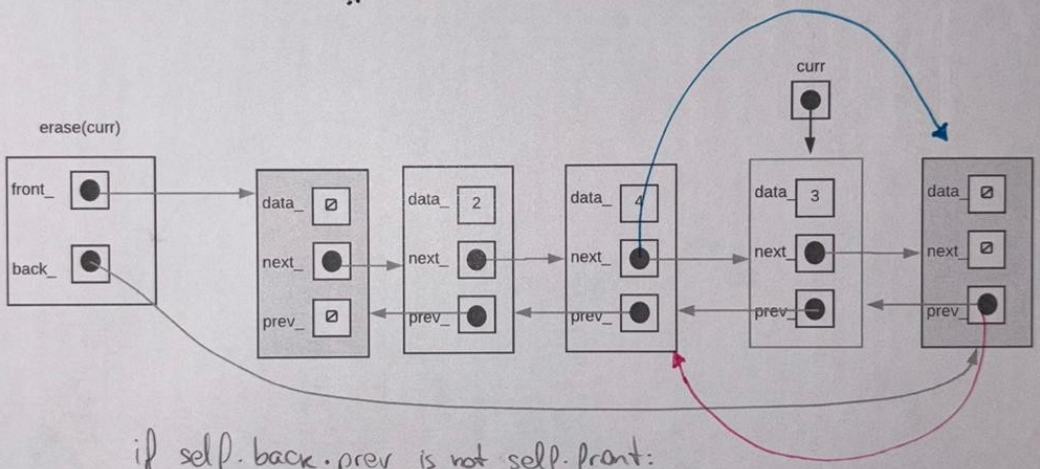

```
def push_front(self, data):
    nn = self.Node(data, self.front.next, self.front)
    self.front.next.prev = nn
    self.front.next = nn
```

```
def erase(self, node = None):
    if self.head.next == self.tail:
        print("The list are empty")
        return
    if node is None:
        print("No node specification")
        return
    node.prev.next = node.next
    node.next.prev = node.prev
```



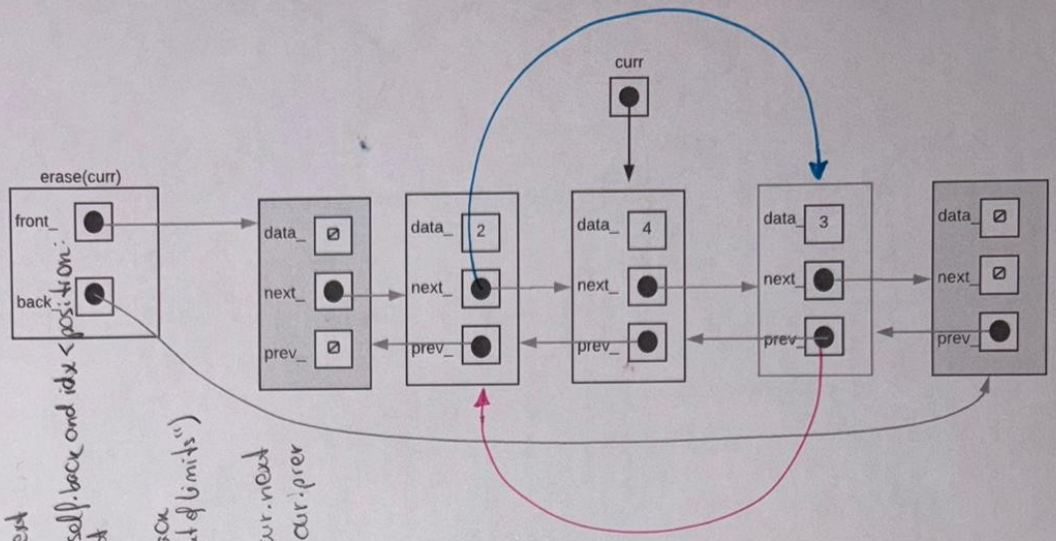


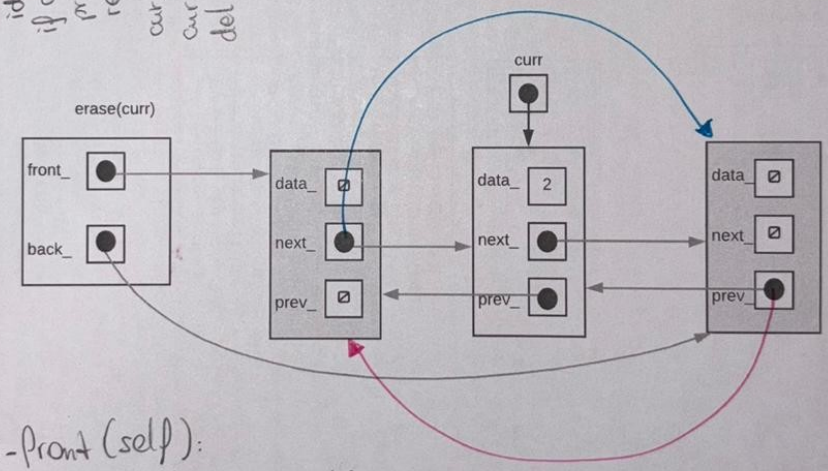

```
if self.front.next is not self.back:
    rm = self.front.next
    rm.next.prev = rm.prev
    rm.prev.next = rm.next
    del rm
```




```
if self.back.prev is not self.front:
    rm = self.back.prev
    rm.prev.next = rm.next
    rm.next.prev = rm.prev
    del rm
```

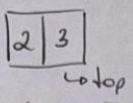
```
def erase_at_position(self, position):
    if self.front.next is self.back:
        print("Empty List")
        return
    cur = self.front.next
    idx = 0
    while cur is not self.back and idx < position:
        cur = cur.next
        idx += 1
    if cur is self.back:
        print("Position out of limits")
        return
    cur.prev.next = cur.next
    cur.next.prev = cur.prev
    del cur
```



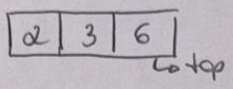



```
def pop_front(self):
    if self.front.next is not self.back:
        rm = self.front.next
        rm.next.prev = rm.prev
        rm.prev.next = rm.next
        del rm
```

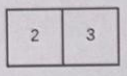
Stack: In the diagrams below list what data members you need to track and what their values are in its initial state and their state after each of the operations are applied to the diagram. If the array needs to be resized, draw the new array with the correct capacity

initial state 

stack.push(6) 

stack.push(6) result 

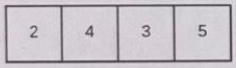
3 is at top of stack

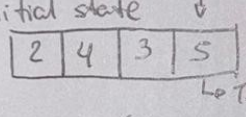


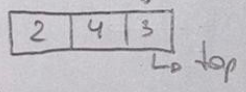
---

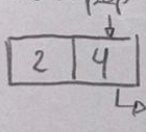
stack.pop()
stack.pop()
stack.push(6)

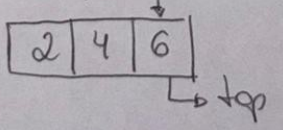
initially 5 is at top of stack



initial state 

stack.pop() 

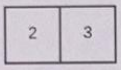
stack.pop() 

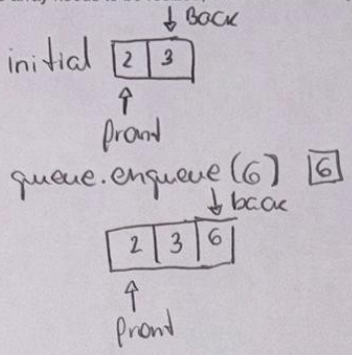
stack.push(6) [6] 

Queues: In the diagrams below list what data members you need to track and what their values are in its initial state and their state after each of the operations are applied to the diagram. If the array needs to be resized, draw the new array with the correct capacity

queue.enqueue(6)

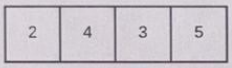
2 is at front of queue, 3 is at back

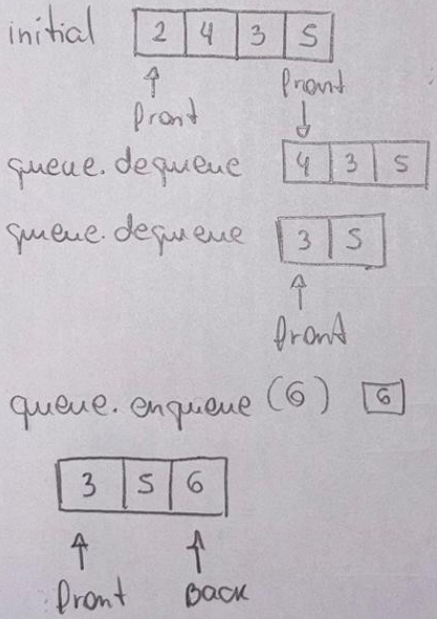




queue.dequeue()
queue.dequeue()
queue.enqueue(6)

initially 2 is at front of queue, 5 is at back





Deques: In the diagrams below list what data members you need to track and what their values are in its initial state and their state after each of the operations are applied to the diagram. If the array needs to be resized, draw the new array with the correct capacity

initial | 2 | 3 |
F B

deque.push_front(6)

2 is at front of Deque, 3 is at back

| 2 | 3 |

deque.push_front(6) 6

| 6 | 2 | 3 |
F B

deque.push_back(6)

2 is at front of Deque, 3 is at back

| 2 | 3 |

initial | 2 | 3 |
F B

deque.push_back(6) 6

| 2 | 3 | 6 |
F B

deque.pop_back()
deque.push_front(6)

initially 2 is at front of deque, 5 is at back

| 2 | 4 | 3 | 5 |

initial | 2 | 4 | 3 | 5 |
F B

deque.pop_back()

| 2 | 4 | 3 |
F B

deque.push_front(6) 5

| 6 | 2 | 4 | 3 |
F B

deque.pop_front()
deque.push_back(6)
deque.pop_front()
deque.push_back(7)

initially 2 is at front of deque, 5 is at back

| 2 | 4 | 3 | 5 |

initial | 2 | 4 | 3 | 5 |
F B

deque.pop_front() | 4 | 3 | 5 |
F B

deque.push_back(6) 6 | 4 | 3 | 5 | 6 |
F B

deque.pop_front() | 3 | 5 | 6 |
F B

deque.push_back(7) 7 | 3 | 5 | 6 | 7 |
F B

F = front
B = back

overflow(grid,the_queue) - apply the overflow function to the gride below and show all the grids the function would add to the queue. Number the grid in the order they are added to the queue. Also state the return value. Note that some grids may remain empty

-2	1	-3	-3	0
2	0	3	2	0
0	0	-3	0	0
0	0	1	0	0

1

0	-3	-1	-1	-1
-3	0	-4	-3	0
0	0	-3	0	0
0	0	1	0	0

2

-2	0	-1	-1	-1
0	-3	0	-2	0
-1	0	-2	0	0
0	0	1	0	0

3

0	-1	-1	-1	-1
-1	-3	0	-2	0
-1	0	-2	0	0
0	0	1	0	0

return 3