

Warm-up

Problem 1. Suppose we implement a stack using a singly linked list. What would be the complexity of the push and pop operations? Try to be as efficient as possible.

Problem 2. Suppose we implement a queue using a singly linked list. What would be the complexity of the enqueue and dequeue operations? Try to be as efficient as possible.

Problem solving

Problem 3. We want to extend the queue that we saw during the lectures with an operation `GETAVERAGE()` that returns the average value of all elements stored in the queue. This operation should run in $O(1)$ time and the running time of the other queue operations should remain the same as those of a regular queue.

- Design the `GETAVERAGE()` operation. Also describe any changes you make to the other operations, if any.
- Briefly argue the correctness of your operation(s).
- Analyse the running time of your operation(s).

Problem 4. Given a singly linked list, we would like to traverse the elements of the list in reverse order.

- Design an algorithm that uses $O(1)$ extra space. What is the best time complexity you can get?
- Design an algorithm that uses $O(\sqrt{n})$ extra space. What is the best time complexity you can get?

You are not allowed to modify the list, but you are allowed to use position/cursors to move around the list.

Problem 5. Using only two stacks, provide an implementation of a queue. Analyze the time complexity of enqueue and dequeue operations.

Problem 6. Consider the problem of given an integer n , generating all possible permutations of the numbers $\{1, 2, \dots, n\}$. Provide a recursive algorithm for this problem.

Problem 7. Consider the problem of given an integer n , generating all possible permutations of the numbers $\{1, 2, \dots, n\}$. Provide a non-recursive algorithm for this problem using a stack.

Problem 3. We want to extend the queue that we saw during the lectures with an operation `GETAVERAGE()` that returns the average value of all elements stored in the queue. This operation should run in $O(1)$ time and the running time of the other queue operations should remain the same as those of a regular queue.

- Design the `GETAVERAGE()` operation. Also describe any changes you make to the other operations, if any.
- Briefly argue the correctness of your operation(s).
- Analyse the running time of your operation(s).

$$\begin{array}{l} \text{(a)} \quad \text{total}() \\ \text{(b)} \quad \hline \\ \text{(c)} \quad \text{size}() \end{array}$$

Queue: FIFO

- We should create a new method, which functionality is:
 - ① After enqueue, it adds the `first()`
 - ② when deque: it minus the element of `dequeue()`
- I prefer to call this function "`total()`"
- since both `total()` and `size()` just return value which complexity is $O(1)$, therefore $\frac{\text{total}()}{\text{size}()}$ is $O(1)$

Problem 1. Suppose we implement a stack using a singly linked list. What would be the complexity of the push and pop operations? Try to be as efficient as possible.

- Pop and push are stack's function
- They simply just manipulate the 1st element
- For Push(), we can
 - ① Create a new node x with element e
 - ② $x.\text{next}$ connect to original node
 - ③ Change head pointer to node x
- As for pop(), we can
 - ① Change the head pointer to $x.\text{next}$,
since there is no pointer connect to
 x , x will be recycled
- Both operations have $O(1)$ complexity in the worst case

Head \rightarrow [1] \rightarrow [2] \rightarrow [3]

1st Deg

Problem 2. Suppose we implement a queue using a singly linked list. What would be the complexity of the enqueue and dequeue operations? Try to be as efficient as possible.

- For enqueue():

X

① Similar to push, so the complexity is $O(1)$

- For dequeue():

X

since we need to traverse $n-1$ elements to find the predecessor of last node (due to singly linked node), therefore the complexity is $O(n)$

FI FO

For enqueue :

$O(1)$

- ① we firstly record the address of last node
- ② create new node
- ③ last.next to new node
- ④ record new last node address

For dequeue

- ① just remove the 1st node

$O(1)$ since it's the 1st element be enqueued

$O(1)$ extra space refers to an algorithm that requires a constant amount of additional memory space, regardless of the size of the input data.

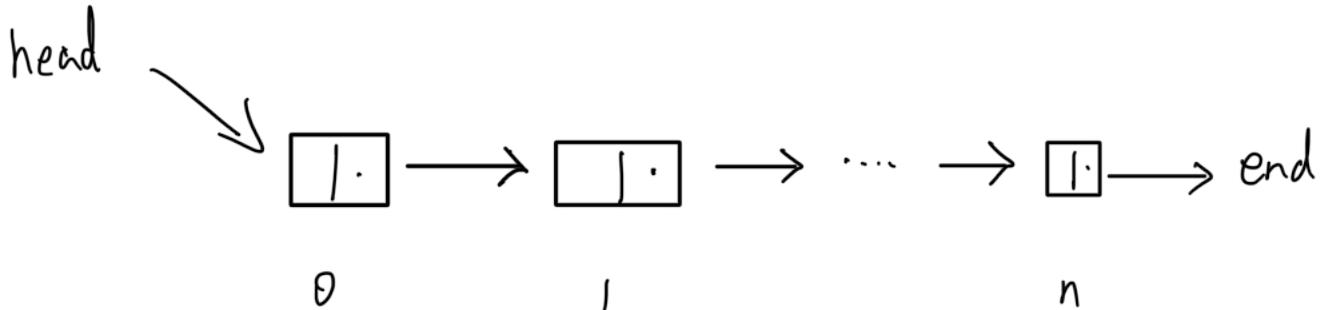


Problem 4. Given a singly linked list, we would like to traverse the elements of the list in reverse order.

- Design an algorithm that uses $O(1)$ extra space. What is the best time complexity you can get?
- Design an algorithm that uses $O(\sqrt{n})$ extra space. What is the best time complexity you can get?

You are not allowed to modify the list, but you are allowed to use position/cursors to move around the list.

$O(1)$ extra space mean?



. Without, the complexity is $O\left(H + \dots + n = \frac{n(n+1)}{2}\right) = O(n^2)$

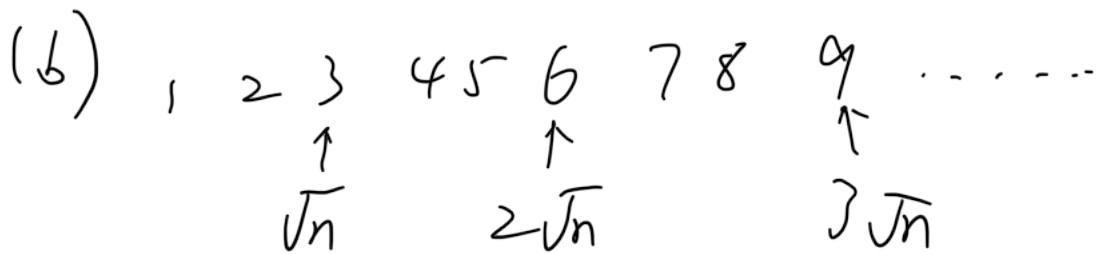
(a)
def reverse(> :

key idea is : we don't have former
node address

How about we create a new linked
list? No, since the space is $O(n)$

① Store the middle node address ?

check answer



如果可以用 stack R:

① 把到 $[1 : \sqrt{n}], \dots, [\sqrt{n} : n]$

② 用 stack 存这些值的 list

③ 最后按顺序 pop 每个 list

Problem 5. Using only two stacks, provide an implementation of a queue. Analyze the time complexity of enqueue and dequeue operations.

Stack: FI LO

Queue: FI FO

① store all enqueue elements in stack1

② whenever enqueue action stops,
we begin to pop all elements
in stack1, and store those
elements in stack2, then we
do deque action for stack2

③ whenever we want to do
any enqueue again, we need
to move all elements in stack2
to stack1 before the enqueue
action

5	5	4
4	4	5
3	3	1
2	2	2
1	1	3
En/push	De	pop

- 更好是在如果 s_2 is empty
就把 s_1 移动到 s_2 (pop 的方式)
- 所有 push 都往 s_1
- pop 时 s_2 有值则 pop s_2
否则先从 s_1 取出到 s_2
~~再 pop s_2~~

Permutation: 扰乱

$n!$

Problem 6. Consider the problem of given an integer n , generating all possible permutations of the numbers $\{1, 2, \dots, n\}$. Provide a recursive algorithm for this problem.

```
return a list
def per(list):
    if list == []:
        return [null]
    if list == 1:
        return [list[0]]
    else:
        for i in list:
            res.append(i + per(list))
        return res
```

有问题，因为 list 应该包含 i

而且这里应该用 For 表示

正确答案：

```
def permutation(lst): # lst = [0, 1, 2, ..., n]
    if len(lst) == 1:
        return [lst]
    else:
        res = []
        for i in lst:
            # 不包含 i 的
            lst = exclude_i_list = [x for x in lst if x != i]
            for p in permutation(exclude_i_list)
                res.append([i] + p)
        return res
```

这里 P 表示不包含 i 的所有 Permutation，在每个

P 前面加上 i 即表示

$$\underbrace{n \cdot (n-1) \cdot \dots \cdot (1)}_{\text{w}} = n!$$

~~X~~ **Problem 7.** Consider the problem of given an integer n , generating all possible permutations of the numbers $\{1, 2, \dots, n\}$. Provide a non-recursive algorithm for this problem using a stack.

x is the list

stack: FILO



just do $\exists x \dots x \ni x_1 \dots x_n$
res = 1
for i in stack:
 res = res \times x_i