
Problem Set 1

Name: Moaaz Elsayed

Problem 1-1.

(a)

$$f_1 = \log n^n = O(n)$$

$$f_2 = (\log n)^n = O((\log n)^n)$$

$$f_3 = \log n^{6006} = O(\log n)$$

$$f_4 = (\log n)^{6006} = O((\log n)^c)$$

$$f_5 = \log \log 6006n = O(\log \log n)$$

solution:

$$(f_5, f_3, f_4, f_1, f_2)$$

(b) solution:

$$(f_1, f_2, f_5, f_4, f_3)$$

(c) solution:

$$(\{f_2, f_5\}, f_4, f_1, f_3)$$

(d) solution:

$$(f_5, f_2, f_1, f_3, f_4)$$

Problem 1-2.

- (a) iterate over the given data structure starting from index i to $(i + k - 1) // 2$, swap i with $(i + k - 1)$ using two variables $x1$ and $x2$ to store the returned values from `delete_at(i)` and `delete_at(i + k - 1)` then use `insert_at(i, x2)` and `insert_at(i + k - 1, x1)` to complete the swap. This recursive procedure makes no more than $k/2 = O(k)$ recursive calls before reaching a base case, doing $O(\log n)$ work per call, so the algorithm runs in $O(k \log n)$ time.

```

1 def reverse(D, i, k):
2     if k < 2:
3         return
4
5     x1 = D.delete_at(i);
6     x2 = D.delete_at(i + k - 1);
7     D.insert_at(i, x2);
8     D.insert_at(i + k - 1, x1);
9     reverse(D, i + 1, k - 2)

```

- (b) we can solve it recursively by setting the base case that is $k == 0$ then do nothing and recursively use `delete_at(i + k - 1)` and store the returned value in $x1$ then we check if j is bigger than i we do `insert_at(j - 1, x1)` as deleting an element make a shift by one in the desired index to insert at else if not $j > i$ we `insert_at(j, x1)`, then call `move(D, i, k - 1, j)` This recursive procedure makes no more than $k = O(k)$ recursive calls before reaching a base case, doing $O(\log n)$ work per call, so the algorithm runs in $O(k \log n)$ time.

```

1 def move(D, i, k, j):
2     if(k == 0):
3         return
4
5     x1 = D.delete_at(i + k - 1);
6
7     if j > i:
8         D.insert_at(j - 1, x1)
9     else:
10        D.insert_at(j, x1)
11
12
13    move(D, i, k - 1, j)

```

Problem 1-3.**Problem 1-4.**

- (a) To support `insert_first(x)` in $O(1)$ time we can use `L.head` pointer by make `x.next` equals the head pointer then make `L.head` points to the node `x`. This algorithm does constant-time operations in so it runs in $O(1)$.

To support `insert_last(x)` in $O(1)$ time we can use `L.tail` pointer by make `L.tail.next` points to the Node `x` then make `L.tail` points to the node `x`. This algorithm does constant-time operations in so it runs in $O(1)$.

To support `delete_first()` in $O(1)$ time we make `L.head` points to `L.head.next`. (assuming that `L.head` exists) This algorithm does constant-time operations in so it runs in $O(1)$.

To support `delete_last()` in $O(1)$ time we make `L.tail` points to `L.tail.prev`. (assuming that `L.tail` exists) This algorithm does constant-time operations in so it runs in $O(1)$.

- (b)
- make the head pointer of `L2` points to `x1`.
 - make the tail pointer of `L2` points to `x2`.
 - if `x1` is the head
 - make `x1.prev` equals `x2.next`
 - if `x1` is **NOT** the head
 - make `x1.prev.next` equals `x2.next`
 - if `x2` is the tail
 - make the tail equals `x1.prev`
 - if `x1` is **NOT** the tail
 - make `x2.next.prev` equals `x1.prev`
 - set `x1.prev` and `x2.next` to `None`
 - return `L2`
- (c)
- make `L2.head.prev` equals to `x`.
 - save `x.next` in `xn` variable.
 - make `x.next` equals `L2.head`.
 - make `L2.tail.next` equals `xn`.
 - if `x` is the tail of `L1`
 - make `L1.tail` equals `L2.tail`.

- if x is **not** the tail of $L1$
 - make $xn.prev$ equals $L2.tail$.

(d)

```

2 class Doubly_Linked_List_Node:
3     def __init__(self, x):
4         self.item = x
5         self.prev = None
6         self.next = None
7
8     def later_node(self, i):
9         if i == 0: return self
10        assert self.next
11        return self.next.later_node(i - 1)
12
13 class Doubly_Linked_List_Seq:
14     def __init__(self):
15         self.head = None
16         self.tail = None
17
18     def __iter__(self):
19         node = self.head
20         while node:
21             yield node.item
22             node = node.next
23
24     def __str__(self):
25         return '-'.join([('(%s)' % x) for x in self])
26
27     def build(self, X):
28         for a in X:
29             self.insert_last(a)
30
31     def get_at(self, i):
32         node = self.head.later_node(i)
33         return node.item
34
35     def set_at(self, i, x):
36         node = self.head.later_node(i)
37         node.item = x
38
39     def insert_first(self, x):
40         node = Doubly_Linked_List_Node(x)
41         if self.head is None:
42             self.head = node
43             self.tail = node
44         else:
45             node.next = self.head
46             self.head.prev = node
47             self.head = node

```

```
48
49
50
51     def insert_last(self, x):
52         node = Doubly_Linked_List_Node(x)
53         if self.head is None:
54             self.head = node
55             self.tail = node
56         else:
57             self.tail.next = node
58             node.prev = self.tail
59             self.tail = node
60
61
62
63     def delete_first(self):
64         x = self.head.item
65         self.head = self.head.next
66         # if we the first ans last elements are the same node
67         if self.head is None:
68             self.tail = None
69         else:
70             self.head.prev = None
71
72         return x
73
74     def delete_last(self):
75         x = self.tail.item
76         self.tail = self.tail.prev
77         # if we the first ans last elements are the same node
78         if self.tail is None:
79             self.head = None
80         else:
81             self.tail.next = None
82
83         return x
84
85     def remove(self, x1, x2):
86         L2 = Doubly_Linked_List_Seq()
87         L2.head = x1
88         L2.tail = x2
89         if x1 == self.head:
90             x1.prev = x2.next
91         else:
92             x1.prev.next = x2.next
93
94         if x2 == self.tail:
95             self.tail = x1.prev
96         else:
97             x2.next.prev = x1.prev
98
```

```
99         x1.prev = None
100         x2.next = None
101         return L2
102
103     def splice(self, x, L2):
104         L2.head.prev = x
105         xn = x.next
106         x.next = L2.head
107         L2.tail.next = xn
108         if x == self.tail:
109             self.tail = L2.tail
110         else:
111             xn.prev = L2.tail
```