Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Instructors: Erik Demaine, Jason Ku, and Justin Solomon

Name:
Problem Set 1

Problem Set 1

Asem Ashraf

Problem 1-1.

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

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

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

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

Problem 1-2.

(a) We go through deleting and storing the $(i+m)^{th}$ and $(i+k-m-1)^{th}$ item in two separate variables, Where m starts from 0 and goes up to $k/2-1$. Then reverse their order in the sequence when inserting them back in. This used 4 O(logn) operations at most $k/2$ times which makes the total complexity O(k.logn).
This is the procedural version of solving this algorithm. There is also a recursive solution.

```
for m in range (k/2):
    Front_var = D.delete_at(i+m) #O(logn)
    End_var = D.delete_at(i+k-m-1) #O(logn)
    D.insert_at(i,End_var)    #O(logn)
    D.insert_at(i+k-1-m,Front_var) #O(logn)
```

(b) i dont know how to do this So, apparently you overwrite the items in front of j to move the k items in front of i. We go through this by deleting the item at index i, and storing it in j+1 m = 0
delete the item at i+m and save it
delete the item at j+m and save it
swap them
add 1 to m
then repeat

Problem 1-3. this can only be implemented in a dynamic array because we want to move_ page(m) in O(1) which can only be done in amortized time. We did not use linked list however because we want the reading time to be O(1) and that cant be achieved even amortized in a linked list. the question is, how are we gonna insert in the middle of an array in a O(1) amortized time?

Problem 1-4.

(a)

- insert_first(x)
  - create a new doubly linked node storing x.
  - if the doubly linked list is empty, then link both the head and tail to point to the new node.
  - Otherwise, assign the head of the list to the next-node pointer in the that new node.
  - assign the new head to previous-node pointer in the old-head node.
  - update the list head pointer to point to the new head.
- insert_last(x):
  - create a new doubly linked node storing x.
  - if the doubly linked list is empty, then link both the head and tail to point to the new node.
  - Otherwise, assign the tail of the list to the previous-node pointer in the that new node.
  - assign the new tail to the next-node pointer in the old-tail node .
  - update the list tail pointer to point to the new tail.
- delete_first():
  - if the next-node pointer of the head node is set to None, then there is no other nodes in the list, hence the tail node is assigned None
  - if there is more than 1 node in the list, then assign the next-node pointer as the list's head node.
  - assign the previous-node pointer of the new head node to None
- delete_last():
  - if the previous-node pointer of the tail node is set to None, then there is no other nodes in the list, hence the head node is assigned None
  - if there is more than 1 node in the list, then assign the next-node pointer to the list's tail node.
  - assign the next-node pointer of the new tail node to None

(b)

- Create a new doubly linked list in O(1) time, and assign x_1 to its head and x_2 to its tail.
- If x_1 is the head of the first list, assign the node after x_2 to be the head of the original list. And set the previous-node pointer in that node to be None.
- Otherwise, assign x_2 to the next-node pointer in node preceding node x_1.
- If x_2 is the tail of the first list, assign the node before x_1 to be the tail of the original list.
- Otherwise, assign the node preceding x_1 to the previous-node pointer in node succeeding node x_2.
- Assign the previous-node pointer of x_1 to null.
- Assign the next-node pointer of x_2 to null.
- Return the new doubly linked list.

(c)

- Save the x.next pointer in a variable x_ next = x.next.
- Assign the previous-node pointer of the head pointer of L_2 to the x.next pointer.
- Assign x.next previous-node pointer to L_2 tail next-node pointer.
- Set L_2 tail pointer to null.
- Set L_2 head pointer to null.

(d)

```
class Doubly_Linked_List_Node:
   def __init__(self, x):
      self.item = x
      self.prev = None
      self.next = None

   def later_node(self, i):
      if i == 0: return self
      assert self.next
      return self.next.later_node(i - 1)

class Doubly_Linked_List_Seq:
   def __init__(self):
      self.head = None
      self.tail = None

   def __iter__(self):
      node = self.head
      while node:
         yield node.item
         node = node.next

   def __str__(self):
      return '-'.join([('(%s)' % x) for x in self])

```

```python
    def build(self, X):
        for a in X:
            self.insert_last(a)

    def get_at(self, i):
        node = self.head.later_node(i)
        return node.item

    def set_at(self, i, x):
        node = self.head.later_node(i)
        node.item = x

    def insert_first(self, x):
        newnode = Doubly_Linked_List_Node(x)
        if self.head==None:
            self.head=newnode
            self.tail=newnode
            return
        newnode.next=self.head
        self.head.prev=newnode
        self.head=newnode

    def insert_last(self, x):
        newnode = Doubly_Linked_List_Node(x)
        if self.tail==None:
            self.head=newnode
            self.tail=newnode
            return

        newnode.prev = self.tail
        self.tail.next = newnode
        self.tail=newnode
    def delete_first(self):
        if self.head.next == None:
            ans=self.head
            self.tail=None
            self.head=None
            return ans
        ans=self.head
        self.head=self.head.next
        self.head.prev=None
        return ans.item

    def delete_last(self):
        if self.tail.prev == None:
            ans=self.tail
            self.tail = None
            self.head = None
            return ans
        ans = self.tail
        self.tail = self.tail.prev
```

```python
        self.tail.next = None
        return ans.item

    def remove(self, x1, x2):
        L2 = Doubly_Linked_List_Seq()
        L2.head=x1
        L2.tail=x2
        if self.head==x1:
            self.head=x2.next
            x2.next.prev=None
        else:
            x1.prev.next=x2.next #this assumes that x2 is not the tail of the list
        if self.tail==x2:
            self.tail=x1.prev
            x1.next.prev=None
        else:
            x2.next.prev=x1.prev #this assumes that x1 is not the head of the list
        x1.prev=None
        x2.next=None
        return L2

    def splice(self, x, L2):
        if L2.head==None and L2.tail==None: # if L2 is empty
            return
        if x.next==None: # if x is the last node in self
            L2.head.prev=self.tail
            self.tail.next=L2.head
            self.tail=L2.tail
            L2.tail=None
            L2.head=None
            return
        x.next.prev=L2.tail
        L2.tail.next = x.next
        x.next = L2.head
        L2.head.prev = x
        L2.head=None
        L2.tail=None
```

Test passed: 5 out of 5 tests - $225ms$