Problem Set 1

Name: Vivek Verma

Collaborators: None

Problem 1-1.

(a) $(f_5, f_3, f_4, f_1, f2)$

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

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

(d) $(f_5, f_3, f_4, f_2, f_1)$ WRONG
$(f_5, f_2, f_1, f_3, f_4)$ From solution

Problem 1-2.

(a) Solving recursively:
Base case: If k is $<= 1$, then nothing needs to be done. Hence return.
Recursive case: If k is > 1,
Swap the elements at index i and $i + k - 1$
Two elements are reversed, the remaining elements to reverse are $k-2$ and the starting index now will be $i + 1$.
Increment i by 1 and k by 2.
Call the function recursively with new i and k
The correctness of this algorithm can be proved by induction.
Swapping the elements take O(log(n)) time because constant number of insertion and deletion are used for swapping.
And k is reduced by 2 in each recursive call, So there will be atmost k/2 recursion calls. Therefore the time complexity is O(klog(n))

```
def reverse(D, i, k):
    if k <= 1:
        return
    right = D.delete_at(i+k-1)
    left = D.delete_at(i)
    D.insert_at(i, right)
    D.insert_at(i+k-1, left)

    reverse(D, i+1, k-2)
```

(b) Solving recursively:
Base case: If k is 0, nothing needs to be done. Hence return.
Recursive case: If k is greater than 0,
Delete the element at index i. and subtract 1 from k.

There are two cases:
1. $j > i$
After deletion of element at index i, all the elements after i will be shifted left by 1 and there indices are reduced by 1.
Insert the element at at index $j - 1$
Call the function recursively.

2. Otherwise,
Insert the element at index j.
The next element to be moved is now at index i+1 and the next index to which it will be moved is at index j + 1.
Increment i by 1.
Increment j by 1.
Call the function recursively.

Insertion and deletion takes O(log(n)) time and k is reduced by 1 with each recursive call, So maximum recursive calls will be atmost k. And hence, the time complexity is O(klog(n)).

```
def move(D, i, k, j):
    x = D.delete_at(i)
    k -= 1
    if j > i:
        D.insert_at(j-1)
        move(D, i, k, j)
    else:
        D.insert_at(j)
        i += 1
        j += 1
        move(D, i, k, j)
```

Problem 1-3. *— After looking at solution —*
Before placing the bookmarks, the pages can be stored in a static array of size n.
$read\_page(i)$ operation takes O(1) time to return a page at index i.

To place the bookmarks, three new dynamic arrays $A_1, A_2\ and\ A_3$ will be created:
A_1 contains the pages before bookmark A
A_2 contains the pages between bookmarks A and B.
A_3 contains the pages after bookmark B.

Building the dynamic arrays takes linear time. Therefore, $place\_mark(i, m)$ operation will take O(n) in worst case.

Also initialize 4 indices variables $a_1, a_2, b_1\ and\ b_2$.
a_1 points to the end of A_1
a_2 points to the start of A_2
b_1 points to the end of A_2
b_2 points to the start of A_3

A_1 supports following operations:

- insert_ last()
- delete_ last()

A_2 supports following operations:

- insert_ last()
- delete_ last()
- insert_ first()
- delete_ first()

A_3 supports following operations:

- insert_ first()
- delete_ first()

* $read\_page(i)$:

```
if i < a_2:
    return A_1[i]
else if i < b_2:
    return A_2[i - a_2]
else:
    return A_3[i - b_2]
```

$read\_page(i)$ takes O(1) time in worst case.

* $shift\_mark(m, d)$:
Moving a page from index (a_1, a_2, b_1, b_2) to the index $(a_2 - 1, a_1 + 1, b_2 - 1, b_1 + 1)$ respectively requires one deletion from a array and one insertion to another array and updating of the indices. An array can be filled completely. In that case, the array can be rebuilt in O(n) time.

Therefore, this operation will take amortized O(1) time.

* $move\_page(m)$:
Moving the page at (a_1, b_1) to the index $(b_1 + 1, a_1 + 1)$ requires one deletion, one insertion and updating of the indices.
Array can be filled completely. In that case, rebuild operation will be required which takes O(n) time.

This operation will also take amortized O(1) time.

Problem 1-4.

(a) $insert\_first(x)$
Create a Node containing the item x.
Check the head of the list to see if list is empty.
If list is empty, set the Node as head and tail of the list and return.
If list is non-empty, get the head of the list called $first$, Connect $first$ as next element of x and x as previous element of $first$.
Set the head of the list to x.

$insert\_last(x)$
Create a Node containing the item x.
Check the tail of the list to see if list empty.
If list is empty, set the Node as head and tail of the list and return.
If list is non-empty, get the tail of list called $last$, Connect $last$ as previous element of x and x as next element of $last$.
Set the tail of the list to x.

$delete\_first()$
Get the first two elements as $first$ and $second$.
Set $second$ as the head of the list and set $second.prev$ as None.
Return $first.item$.

$delete\_last()$
Get the last two elements as $last$ and $second\_last$.
Set $second\_last$ as tail of the list and set $second\_last.next$ as None.
Return $last.item$.

(b) Construct a new empty list $L2$.
There are four cases based on the location of x_1 and x_2 in the list:

1. Neither x_1 is head nor x_2 is tail:
   Connect the element before x_1 to the element after x_2.
2. x_1 is the head:
   Set the head of the list to the element after x_2 and set the previous element of head as None.
3. x_2 is the tail:
   Set the tail of the list to the element before and set the next element of tail as None.
4. x_1 is the head and x_2 is the tail:
   Set the head and tail of the list to None.

Independent of which case above was executed,
Set the previous element of x_1 and next element of x_2 as None.
Set the head of $L2$ to x_1 and the tail of $L2$ to x_2.

(c) If $L2$ is an empty list, nothing needs to be done, hence return None.
Otherwise,
Get the element after x in a variable x_next.
Set the next element of x to head of $L2$ and previous element of $L2.head$ to x.

If x_next is not None, Set previous element of x_next to tail of $L2$ and next next element of $L2's$ tail to x_next.
Set the tail of L as tail of $L2$.
Set head and tail of $L2$ to None.

(d)

```python
def insert_first(self, x):
    # create node with item x
    x = Doubly_Linked_List_Node(x)

    # if list is empty, add x as only element
    if self.head is None:
        self.head = x
        self.tail = x
        return

    # get the first element and connect it to x
    first = self.head
    first.prev = x

    # connect x to first
    x.next = first

    # set x as the head
    self.head = x

def insert_last(self, x):
    # create node with item x
    x = Doubly_Linked_List_Node(x)

    # if list is empty add, x as the only element
    if self.tail is None:
        self.tail = x
        self.head = x

    # get the last element and connect it to x
    last = self.tail
    last.next = x

    # connect x to last element
    x.prev = last

    # set x as the tail
    self.tail = x

def delete_first(self):
    # get first two elements
    first = self.head
    second = first.next

    # set second element as head ot the list
    second.prev = None
    self.head = second
    return first.item

def delete_last(self):
    # get last two elements
```

```
        last = self.tail
        second_last = last.prev

        # set second last element as tail of the list
        second_last.next = None
        self.tail = second_last
        return last.item

    def remove(self, x1, x2):
        L2 = Doubly_Linked_List_Seq()

        # Neither x1 nor x2 is the head or tail
        if x1.prev is not None and x2.next is not None:
            # connect the element before x1 to the element after x2
            x1.prev.next = x2.next
            x2.next.prev = x1.prev
        # if x1 is the head but x2 is not tail
        elif x1.prev is None and x2.next is not None:
            # disconnect elements x1 to x2 from the list
            self.head = x2.next
            self.head.prev = None
        # if x2 is the tail but x1 is not head
        elif x2.next is None and x1.prev is not None:
            # disconnect elements from x1 to x2 from the list
            self.tail = x1.prev
            self.tail.next = None
        # x1 is head and x2 is tail
        else:
            self.head = None
            self.tail = None

        # disconnect the links to any element before x1 and any elment
        x1.prev = None
        x2.next = None

        # set the head and tail of L2
        L2.head = x1
        L2.tail = x2
        return L2

    def splice(self, x, L2):
        # if L2 is empty, return None
        if L2.head is None:
            return

        # get the element after x
        x_next = x.next

        # connect x to head of L2
        x.next = L2.head
        L2.head.prev = x
```

```python

        # there was an element after x, connect it to tail of L2
        if x_next is not None:
            x_next.prev = L2.tail
            L2.tail.next = x_next

        # set the tail of current list as tail of L2
        self.tail = L2.tail

        # remove all elements from L2
        L2.head = None
        L2.tail = None
```