Exercise 2.1*

[10 points]

The List method `add_all(i, c)` inserts all elements of the Collection `c` into the list at position `i`. (The `add(i, x)` method is a special case where `c = {x}`.)

Explain why, for the data structures in Chapter 2, it is not efficient to implement `add_all(i,c)` by repeated calls to `add(i, x)`. Design a more efficient implementation.

Solution: In chapter 2, all the data structures are array based. This means that the length of data structure can neither be enlarged nor be shortened dynamically. In order to increase or decrease the array-size, we use a *backing array concept* on the back-end. In some cases resize() operation is called on array. If we wish to transfer all the elements of Collection 'c' into an array 'a' at an index 'i', it would cost a lot. The add() function will make internally shift n-i elements to the right to make a space for a single element at index i. The process will be repeated unless all the elements in the collection are transferred to *a*. The estimated time of this operation will be O(n$\hat{2}$). We can always do this job in O(n) hence it is not efficient to use this approach for the given job. Following is more efficient way to perform this task.

Algorithm 0.1: EfficientArray(a, i, c)

comment: Transfers all the elements from c to a at index i.

Assuming that 'n' is the number of elements in a 'm' is the number of elements in c.

```
procedure TRANSFER$(a, c, i)$
 if $len(a) - n \le m + 1$ then RESIZE(a)
   else for $range \leftarrow 0$ to $m - 1$
      do { $lastValue \leftarrow a[n-1]$
           for $j \leftarrow n-1$ to $i$
             do $a[j+m] \leftarrow a[j]$. //Shifts all the elements after ith postion to (i+m)th position
           $a[i] \leftarrow lastValue$
           for $k \leftarrow 0$ to $m-1$ do a[i+k]$\leftarrow c[k]$
```

Exercise 2.4*

[5 points]

Design a method `rotate(a,r)` that "rotates" the array `a` so that `a[i]` moves to `a[(i + r) mod length(a)]`, `for all i` $\in$ {`0, ... , length(a)`}.

Solution:

Algorithm 0.2: ARRAY(a, r)

```
comment: Rotates the array 'a' towards right w.r.t. 'r'.
Assuming that 'n' is the length of $a$

procedure ROTATE$(a, r)$
 r=r%n
 for $range \leftarrow 0$ to $r - 1$
    do { $lastValue \leftarrow a[n-1]$
         for $i \leftarrow n - 1$ to $0$
           do $a[i] \leftarrow a[i-1]$
         $a[0] \leftarrow lastValue$
```

Exercise 2.5*

[5 points]

Design a method `rotate(r)` that "rotates" a List so that list item i becomes list item $(i + r) \bmod n$. When run on an ArrayDeque, or a DualArrayDeque, `rotate(r)` should run in $O(1 + \min r, n - r)$ time.

Solution:

Algorithm 0.3: ArrayDeque(a, r)

comment: Rotates the array 'a' towards right w.r.t. 'r'.

Assuming that add_first(x), add_last(x), remove_first() & remove_last exist in an ArrayDeque, a.

```
procedure ROTATE$(a, r)$
 r=r%n
 if $r \leq n/2$
   then { for $i \leftarrow 1$ to $r$
            do $add\_first(remove\_last)$
   else { for $j \leftarrow 1$ to $(n - r)$
            do $add\_last(remove\_first)$
```

Exercise 2.8* [10 points]

Design a variant of ArrayDeque that does not do any modular arithmetic at all. Instead, all the data sits in a consecutive block, in order, inside an array. When the data overruns the beginning or the end of this array, a modified `rebuild()` operation is performed. The amortized cost of all operations should be the same as in an ArrayDeque.

Hint: Getting this to work is really all about how you implement the `rebuild()` operation. You would like `rebuild()` to put the data structure into a state where the data cannot run off either end until at least $n/2$ operations have been performed.

Solution:

Algorithm 0.4: ArrayDeque(*array*)

We start off with rebuilding the list. The rebuild(arr) method increases the size of the list by two times. The 'n' elements placed in the middle while a room of n/2 elements left on either side. Due to space constraint (20 lines) rebuild(arr) has only been implemented for an increasing size; however it works in the similar way for decreasing size. We assume that initially we have an array in its rebuilt version i.e. no. of elements = n & size of array = 2n.

```
procedure INITIALISE($arr$)
  $j \leftarrow n/2, k \leftarrow 3n/2$ //global variables.

procedure ADD_FIRST($x$)
  if $j = 0$ then rebuild(arr, true) else arr[j]$\leftarrow x, j \leftarrow j - 1$

procedure ADD_LAST($x$)
  if $k = (2n - 1)$ then rebuild(arr, true) else arr[j]$\leftarrow x, k \leftarrow k + 1$

procedure REMOVE_FIRST()
  if $n \leq len(arr)$
    then $rebuild(arr, false)$
    else $j \leftarrow j + 1$

procedure REMOVE_LAST()
  if $n \leq len(arr)$
    then $rebuild(arr, false)$
    else $k \leftarrow k - 1$

procedure REBUILD($arr, boolean$)
  if $boolean = true$
    then { $n \leftarrow len(arr), newArray \leftarrow \underline{array}[2n]$ //$\underline{\text{array}}$ here means initializing a new array.
         { for $i \leftarrow 0$ to $(3n/2)$
         {   do if $i < n/2$ then continue else newArray[i]$\leftarrow arr[i - n/2]$
    else { $n \leftarrow len(arr), newArray \leftarrow \underline{array}[n/2]$ //$\underline{\text{array}}$ here means initializing a new array.
         { for $i \leftarrow 0$ to $(2n)$
         {   do if $newArray[i]$ is empty then continue else newArray[i]$\leftarrow arr[i]$
```

Exercise 3.1 [10 points]

Why is it not possible to use a dummy node in an SLList to avoid all the special cases that occur in the operations `push(x), pop(), add(x)`, and `remove()`?

Solution: We know that with the introduction of a dummy node, we remove the head and tail pointers pointing to the first and last nodes of the SLList respectively. The introduction of dummy node can appear in two ways. Either the dummy node gets added at the *tail* position of SLList or it gets added at the *head* position. Let's discuss both of the scenarios one by one.

The Tail-End Scenario:
There is no point in adding the dummy node towards the tail end of a SLList as it will restrict the occurence of any operation of a SLList, namely; push(x), pop(), add(x) & remove(). There will be no way left to navigate through the list because when the program runs, we are at a dummy position of our data structure which does not point towards any other node (in non-circular SLList); hence the navigation gets restricted.

The Head-End Scenario:
If dummy node is introduced at the head-end of SLList, we are stuck for the special case of add() operation. There is no tail for the 'add()' operation to happen and hence if we wish to add towards the tail-end, we will have to traverse through the whole list until we find the node which has a *nullptr* as its next pointer.

On the whole, it is not possible to use a dummy node in a SLList for a bunch of reasons. In special cases, we arrive at certain points where the data struture gets restricted to access extreme-end positions. Hence, a SLList is implemented only with the tail and head pointers and not with a single dummy node replacement.

Exercise 3.2* [5 points]

Design an SLList method, `second_last()`, that returns the second-last element of an SLList. Do this without using the member variable, n, that keeps track of the size of the list.

Solution:

Algorithm 0.5: SLList(L)

comment: Returns the second last element of L.

Assuming that 'head' has been declared as the first element of L and 'tail' as the last element of L. Moreover, each node has an attribute 'front' which points towards the next node in the SLList L and a 'data' attribute which holds the value.

```
procedure SECOND_LAST()
 $temp \leftarrow L.head$
 while $temp.front \neq L.tail$
   do $temp \leftarrow temp.front$
 return ($temp.data$)
```

Exercise 3.4* [5 points]

Design an SLList method, `reverse()` that reverses the order of elements in an SLList. This method should run in $O(n)$ time, should not use recursion, should not use any secondary data structures, and should not create any new nodes.

Solution:

Algorithm 0.6: SLList(L)

comment: Reverses the SLList L.

Assuming that 'head' has been declared as the first element of L and 'tail' as the last element of L. Moreover, each node has an attribute 'front' which points towards the next node in the SLList L. Declaring three variables (pointer-type) as *temp*, *tempPrevious* and *tempNext*.

```
procedure REVERSE()
 $temp \leftarrow L.head$
 $tempNext \leftarrow temp.front$
 $tempPrevious \leftarrow NULL$
 while $temp \neq L.tail$
   do { $tempPrevious \leftarrow temp$
        $temp = tempNext$
        $tempNext = temp.front$
        $temp.front = tempPrevious$
```

Exercise 3.8* [5 points]

Design a method `rotate(r)` that "rotates" a DLList so that list item i becomes list item $(i+r) \bmod n$. This method should run in $O(1+\min\{r, n-r\})$ time and should not modify any nodes in the list.

Solution:

Algorithm 0.7: DLList(L, i, r)

comment: Rotates a DLList L such that every ith element moves to $(i+r)\%len(L)th$ position.

Assuming that there is a dummy node named 'dummy' in the circular DLList L. The number of iterations done (uni-directional) from *dummy* all the way to *dummy* determines the size of the L.

```
procedure ROTATE($r$)
  r=r%n
  if $r \leq n/2 - 1$
    then
      for $i \leftarrow 0$ to $(r-1)$
        do
          $dummy.back = dummy.back.back$
          $dummy.back.front.front = dummy.front$
          $dummy.front.back = dummy.back.front$
          $dummy.front = dummy.back.front$
          $dummy.back.front = dummy$
          $dummy.front.back = dummy$
    else
      for $i \leftarrow 0$ to $(n-r-1)$
        do
          $dummy.front = dummy.front.front$
          $dummy.front.back.back = dummy.back$
          $dummy.back.front = dummy.front.back$
          $dummy.back = dummy.front.back$
          $dummy.front.back = dummy$
          $dummy.back.front = dummy$
```

Exercise 3.19* (challenge) [0 points]

Show using pseudocode how the bitwise exclusive-or operator, ˆ, can be used to swap the values of two `int` variables without using a third variable.

Solution:

Assumptions: We assume that:
1. The two variables containing integer values are not type-sensitive (just like *Python* variables or 'auto' in *C++*).
2. There exists a method decimalToBinary(a) which takes a decimal value as input and returns a binary equivalent of *a*.
3. There exists a method binaryToDecimal(a) which takes a binary value as input and returns a decimal equivalent of *a*.

Algorithm 0.8: SWAP($first, second$)

```
comment: Swaps the values of first and second.
procedure SWAP($first, second$)
  $first \leftarrow decimalToBinary(first)$
  $second \leftarrow decimalToBinary(second)$

  $first \leftarrow first \oplus second$
  $second \leftarrow first \oplus second$
  $first \leftarrow first \oplus second$

  $first \leftarrow binaryToDecimal(first)$
  $second \leftarrow binaryToDecimal(second)$


procedure DECIMALTOBINARY($number$)
  //Checks if the number is decimal. Converts 'number' to binary and returns its binary
  equivalent.


procedure BINARYTODECIMAL($number$)
  //Checks if the number is binary. Converts 'number' to decimal and returns its decimal
  equivalent.
```

* - The question has been modified from the one in the book to exclude implementation. "Design" in a question here means to write pseudocode.