MECHTRON 2MD3 | Assignment 2

Question 1

> Describe in detail how to swap two nodes `x` and `y` (and not just their contents) in a singly linked list L given references only to `x` and `y`. You can assume that you also have a reference to the `head` of the singly linked list, and to the header and trailer for the doubly linked list. No assumptions can be made about the relative location of nodes `x` and `y` in the list. Repeat this exercise for the case when L is a doubly linked list. Which algorithm takes more time?

```
//using the structure
struct Node {
    int data;
    Node* next;
};
```

To swap two nodes in a singly-linked list, you would do the following steps:

1. Lets say that you are given the following structure in the function call:

```
int main(){
    swapNodes(head, x, y) //where head, x, and y are addresses
    return 0;
}
```

2. Then in the swap function you would create a while loop that goes through the entire linked list to find the positions of x and y starting from head.
3. With the positions of x and y, to swap the two nodes you would have to change 4 things:
    - `x->next` value
    - `x->prev->next` value
    - `y->next` value
    - `y->prev->next` value
4. What these values would be is:
    - `x->next = y->next`
    - `x->prev->next = y`
    - `y->next = x->next`
    - `y->prev->next = x`

5. To do this you would have to check if x and y are the same address, if they are then you would do nothing.
6. First you would use a while loop to search for an address that is the same as the address of x

```
//the following is pseudocode
Node *prevX = NULL; //the address of prevX
Node *currX = *head;
while address of currX != x {
    prevX = currX;
    currX = currX->next;
}
//when this while loop stops we are left with curr=x and prevX = x->prev

//the same code above would be repeated for node y
Node *prevY = NULL; //the address of prevX
Node *currY = *head;
while address of currY != y {
    prevY = currY;
    currY = currY->next;
}
```

7. Now we are left with the addresses of `prevX` , `x` , `prevY` , and `y`
8. Now assume that x and y exist and that they are not the head of the list. You would simply just:

```
prevX->next = currY;

prevY->next = currX;

Node *temp = currX->next;
currX->next = currY->next;
currY->next = temp;
```

9. Else if x or y is the head, you would:

```
// if x is the head of the list, instead of prevX->next = currY
*head = currY;
// if y is the head of the list, instead of prevY->next = currX
*head = currX;
```

For a doubly-linked list, the process would be simpler as you do not have to save the value of the prev pointer. You would first check if x or y is the head of the list.

- If x is the head of the list or the tail then the you would set y as the head or the tail respectively. The vice-versa also applies to if y is the head or the tail.

Then what you would do is:

```
//asumming that you checked if x and y exist
Node *temp = NULL;
temp = x->next;
x->next = y->next;
y->next = temp;
//then assuming that x->next and y->next is not NULL due to being tail
x->next->prev = x; //since y->next->prev would still be y
y->next->prev = y; //since x->next->prev would still be x
//then to swap the previous
temp = x->prev;
x->prev = y->prev;
y->prev = temp;
//and finally if x->prev and y->prev is not NULL due to being head
x->prev->next = x //since x->prev->next would still be y
y->prev->next = y
```

The above steps would result in swapping two nodes in a doubly linked list. Now in terms of the time usage by both algorithms the swapping nodes for the singly linked list would take more time as you would have to use a while loop to find the `prev` values while in a doubly-linked list you would just take the points using `x->prev`

Question 2

> Draw the recursion trace for the execution of function ReverseArray(A,0,4) (Code Fragment 3.39 in Goodrich text) on array A={4,3,6,2,5}

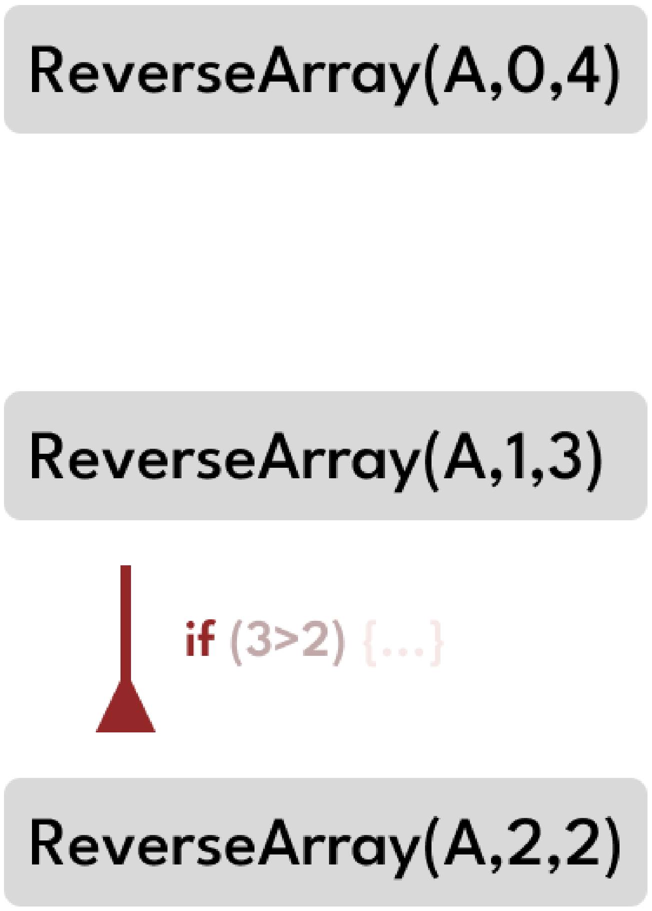
```
Algorithm ReverseArray(A, i, j):
    Input: An array A and nonnegative integer indices i and j
    Output: The reversal of the elements in A starting at index i and ending at j
    if i < j then:
        Swap A[i] and A[ j]
        ReverseArray(A, i+1, j-1)
    return
```

```
#include <iostream>

int ReverseArray(int* A, int i, int j){
    if(i < j) {
        int temp = A[i];
        A[i] = A[j];
        A[j] = temp;
        ReverseArray(A, i+1, j-1);
    } else {
        return 0;
    }
}

int main(){
    int array[] = {4,3,6,2,5};
    ReverseArray(array, 0, 4);
    for(int i =0; i < sizeof(array)/sizeof(int); i++){
        std::cout << array[i] << std::endl;
    }
}
```

Utilizing this code as a reference, the recursion trace for the recursion trace of the function `ReverseArray` for index `A(0,4)` would be:


ReverseArray(A,0,4)
ReverseArray(A,1,3)
if (3>2) {...}
ReverseArray(A,2,2)