Data Structures - Assignment 2

Name - Gaurang Shukla
Rollno - CS2415
Course - Mtech CS

Q1. Design a stack to support an additional operation that returns the min element from the stack in constant time. The stack should continue supporting all other operations like PUSH, POP, TOP, EMPTY etc. with no degradation in these operations' performance.

Stack Design

1. **Primary Stack (`stack`)**: This stack will store all the elements.
2. **Min Stack (`minStack`)**: This stack will store the minimum values.

Operations

1. **Push Operation (`push(item)`)**:
   - Push the element `x` onto the `stack`.
   - If `minStack` is empty or an `item` is less than or equal to the top element of `minStack`, push the `item` onto `minStack`.
2. **Pop Operation (`pop()`)**:
   - Pop the element from the top of the `stack`.
   - If the popped element is equal to the top element of `minStack`, pop the top element from `minStack`.
3. **Min Operation (`min()`)**:
   - Return the top element of `minStack`, which is the current minimum.
4. **Empty Operation (`empty()`)**:
   - Check if the `stack` is empty. If it is, return `true`; otherwise, return `false`.

Implementation in C

```c
#include<stdio.h>
#include<stdbool.h>
#define MAX_SIZE 1000

int stack[MAX_SIZE];

int minStack[MAX_SIZE];
```

```
int top=-1;
int minTop = -1;

bool empty() {
    return top == -1;
}

void push(int item) {

    if (top == MAX_SIZE-1) {

        printf("Stack Overflow\n");
        return;

    }

    if(empty() == true || minStack[minTop] >= item)
        minStack[++minTop]=item;

    stack[++top] = item;

}

int pop() {
    if (top == -1) {

        printf("Stack Underflow\n");
        return -1;

    }

    if(stack[top] == minStack[minTop]) {

        minStack[minTop--];
    }

    return stack[top--];
}


int min_ele() {


    if (top == -1) {

        printf("Stack Underflow\n");
```

```
        return -1;

    }

    return minStack[minTop];


}



void print() {

    for(int i=0;i<=top;i++) {
        printf("%d ",stack[i]);
    }
}


int main() {




    printf("1.Push\n2.Pop\n3.Minimum element\n4.Print stack\n5.exit\n");



    while(true) {

        int oper;


        scanf("%d",&oper);

        if(oper == 5) {
            break;
        } else if (oper == 1)
        {
            int item;
            scanf("%d",&item);

            push(item);
        } else if (oper == 2)
        {
            int ele = pop();
```

```
            printf("%d\n",ele);
        } else if (oper == 3)
        {
            int minimum = min_ele();
            printf("%d\n",minimum);
        }else if (oper == 4)
        {
            print();
            printf("\n");
        }
         else {
            printf("\n Invalid Input\n");
        }



    }


    return 0;
}
```

Q2. Given a single linked list x1 → x2 → · · · → x2n containing 2n nodes, rearrange the nodes to be x1 → x2n → x2 → x2n−1 · · · .

Algorithm

- Find the middle of the linked list and break the linked list into two parts 1…n and n+1…..2n.
- Reverse the second half of the linked list starting from the n+1.
- Merge the first half and the reversed second half in an interleaved fashion.

Time Complexity

We have to linearly traverse all the elements of the linked list to rearrange the nodes in the given manner.

Time Complexity = Reversing n nodes + Traversing 2n nodes to rearrange.

= O(n) + O(2n) = O(n)

Implementation in C

```c
#include<stdio.h>
#include<stdbool.h>

struct Node
{
    int val;
    struct Node* next;
};

struct Node* createNode(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->val = val;
    newNode->next = NULL;
    return newNode;
}

void appendNode(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Function to print the linked list
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->val);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to free the linked list
void freeList(struct Node* head) {
    struct Node* temp;
    while (head != NULL) {
        temp = head;
```

```
        head = head->next;
        free(temp);
    }
}

struct Node* reverse(struct Node* head) {

    struct Node *temp = head;
    struct Node *next = NULL;
    struct Node *prev = NULL;

    while(temp!=NULL) {
        next = temp->next;

        temp->next = prev;

        prev = temp;

        temp =next;

    }

    return prev;
}


struct Node* rearrange(struct Node* head)
{

    struct Node *slow = head;
    struct Node *fast = head;

    struct Node *prev = head;

    while(slow!=NULL && fast!=NULL && fast->next!=NULL ) {

        prev = slow;
        slow = slow->next;

        fast = fast->next->next;

    }

    prev->next = NULL;

    struct Node *Node2 = reverse(slow);
```

```
    struct Node *Node1 = head;
    struct Node *next1 = NULL;
    struct Node *next2 = NULL;

    while(Node2!=NULL) {

        next1 = Node1->next;
        next2 = Node2->next;

        Node1->next = Node2;

        Node2->next = next1;

        Node1 = next1;
        Node2 = next2;

    }

    return head;
}


int main() {

    struct Node* head = NULL;
    int data, count = 0;

    printf("Enter nodes for the linked list (input an even number of
nodes):\n");
    while (1) {
        printf("Enter data for node %d: ", count + 1);
        scanf("%d", &data);
        appendNode(&head, data);
        count++;

        if (count % 2 == 0) {
            char choice;
            printf("You have entered %d nodes. Do you want to add more nodes?
(y/n): ", count);
            scanf(" %c", &choice);
            if (choice == 'n' || choice == 'N') {
                break;
            }
        }
    }
```

```
    printf("The linked list is: ");
    printList(head);

    rearrange(head);
    printf("The Rearranged linked list is: ");
    printList(head);

    // Free the allocated memory for the list
    freeList(head);


    return 0;
}
```

Q3. Prove that the permutation of 1, 2, · · · , n can be generated by a stack if and only if it has no forbidden triple (a, b, c) s.t. a < b < c with ’c’ appears before ’a’ and ’a’ appears before ’b’.

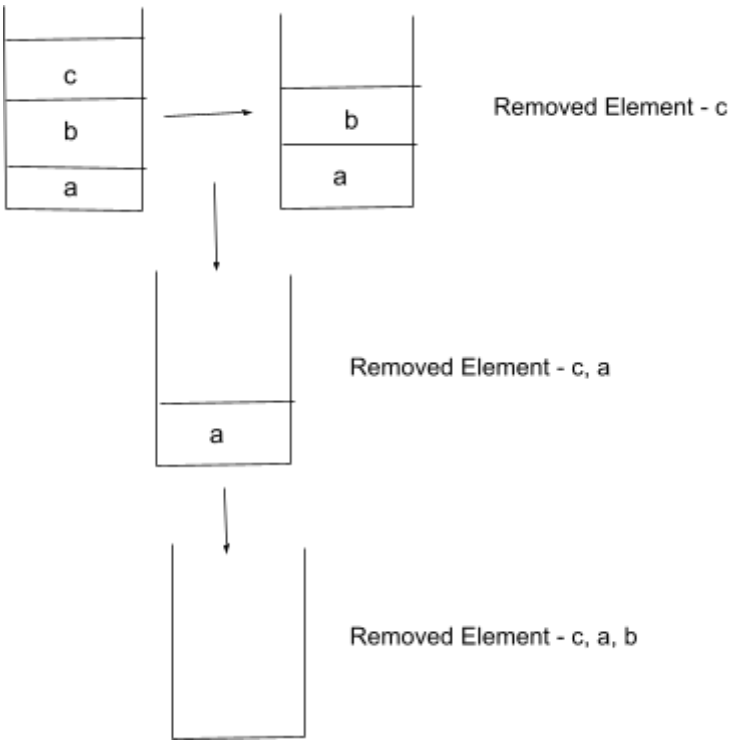
- Consider a permutation of 1,2,3,.....,n that is generated by a stack, where n>=3. Let a,b,c be three numbers in the permutation such that a<b<c.

- Let us also consider that the permutation formed by stack has a sequence in which **c** appears before **a** and **a** appears before **b**.

- We have inserted the elements in the stack in the order **a b c**. As **c** appears the first before the two elements in the permutation. This can only happen after pushing **a**, **b** and **c** into the stack and then removing **c** from the top of the stack.

- After removing the element **c**, **a** appears before **b** in the stack, then **b** will always be removed before **a**.

- So, the contradiction arises, The condition **a<b<c** with **c** before **b** and **a** before **b** cannot be achieved with stack operations because **b** would need to be removed before **a**.


c
b
a
b
a
Removed Element - c
a
Removed Element - c, a
Removed Element - c, a, b