DEPARTMENT OF

Discover. Learn. Empower.

Day 4 _____

Student Name: Sweta singh **UID: 22BCS10664**

Branch: CSE 3rdyear **Section/Group:** KPIT-901 A

Q1. **1 Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.**

Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

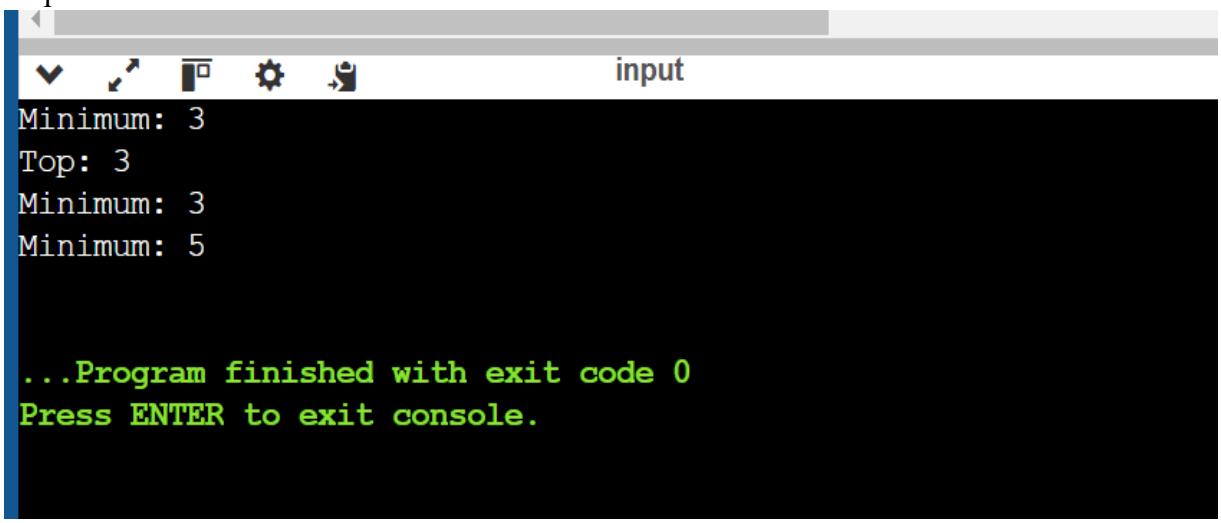
You must implement a solution with O(1) time complexity for each function.

Code :

```
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100
typedef struct {
    int stack[MAX_SIZE];
    int min_stack[MAX_SIZE];
    int top;
    int min_top;
} MinStack;
void minStackInit(MinStack *obj) {
    obj->top = -1;
    obj->min_top = -1;
}
void minStackPush(MinStack *obj, int val) {
    obj->stack[++(obj->top)] = val;
    if (obj->min_top == -1 || val <= obj->min_stack[obj->min_top]) {
        obj->min_stack[++(obj->min_top)] = val;
}
void minStackPop(MinStack *obj) {
    if (obj->stack[obj->top] == obj->min_stack[obj->min_top]) {
```

```
        obj->min_top--;
    }
    obj->top--;
int minStackTop(MinStack *obj) {
    return obj->stack[obj->top];
}
int minStackGetMin(MinStack *obj) {
    return obj->min_stack[obj->min_top];
}
int main() {
    MinStack minStack;
    minStackInit(&minStack);

    minStackPush(&minStack, 5);
    minStackPush(&minStack, 3);
    minStackPush(&minStack, 7);

    printf("Minimum: %d\n", minStackGetMin(&minStack));
    minStackPop(&minStack);
    printf("Top: %d\n", minStackTop(&minStack));
    printf("Minimum: %d\n", minStackGetMin(&minStack));

    minStackPop(&minStack);
    printf("Minimum: %d\n", minStackGetMin(&minStack));

    return 0;
}
```

Output :

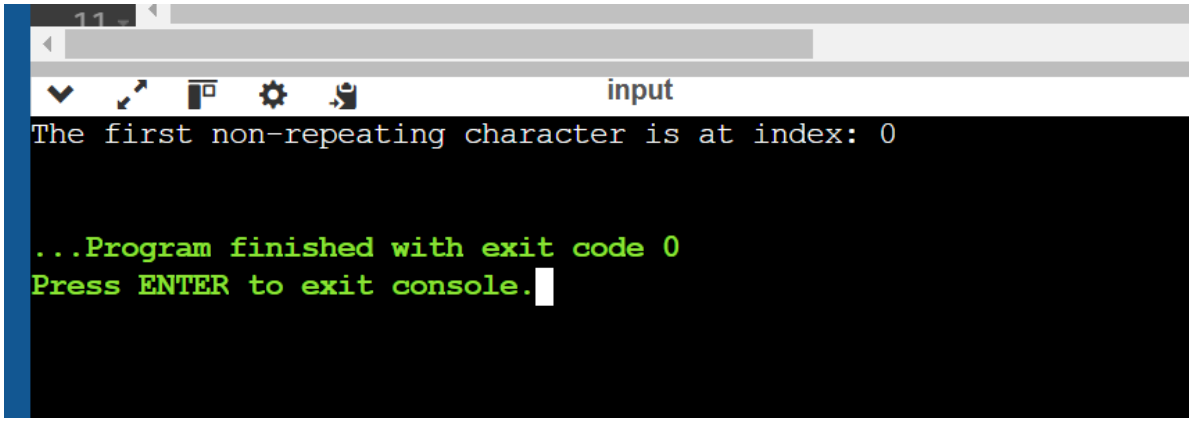


Q2. **Given a string s, find the first non-repeating character in it and return its index. If it does not exist, return -1.**

Code:

```
#include <stdio.h>
#include <string.h>
int firstUniqChar(char *s) {
    int count[256] = {0};
    for (int i = 0; s[i] != '\0'; i++) {
        count[(unsigned char)s[i]]++;
    }
    for (int i = 0; s[i] != '\0'; i++) {
        if (count[(unsigned char)s[i]] == 1) {
            return i;         }
    }
return -1;
}
int main() {
    char s[] = "sweta";
    int index = firstUniqChar(s);
   if (index != -1) {
        printf("The first non-repeating character is at index: %d\n", index);
    } else {
        printf("No non-repeating character found.\n");
    }
return 0;
}
```

Output :



Q3. Implement a simple text editor. The editor initially contains an empty string, S.Perform Q operations of the following 4 types:

append(W) - Append string W to the end of S.

delete (k)- Delete the last k characters of S.

print (k)- Print the k^th character of S.

undo() - Undo the last (not previously undone) operation of type 1 or 2, reverting S to the state it was in prior to that operation.

Code:

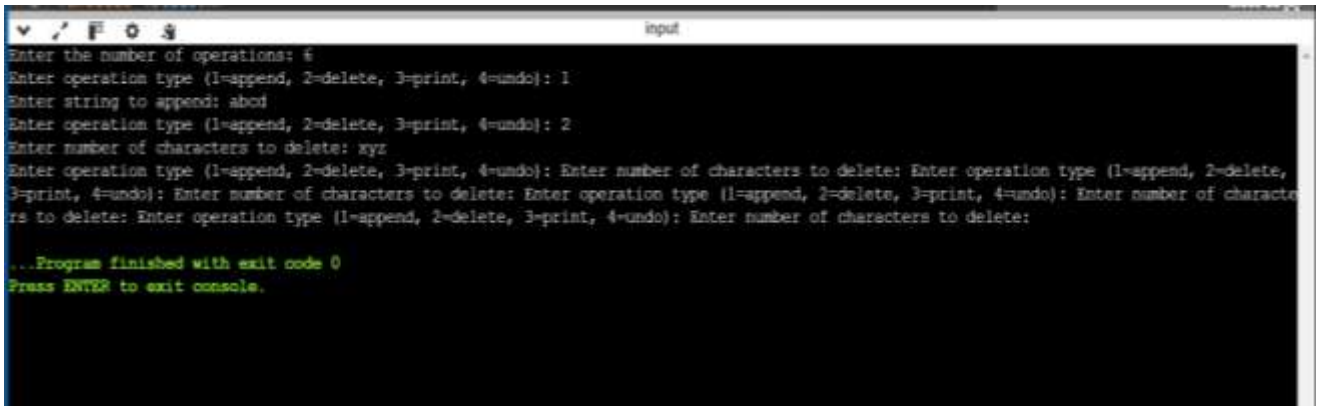
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_LEN 1000
#define MAX_OPERATIONS 100

typedef struct {
    char *state;
} Operation;
```

```
void append(char **S, const char *W, Operation *history, int *undo_index);
void delete (char **S, int k, Operation *history, int *undo_index);
void print(const char *S, int k);
void undo(char **S, Operation *history, int *undo_index);
int main() {
    char *S = (char *)malloc(MAX_LEN * sizeof(char));
    S[0] = '\0';
    Operation history[MAX_OPERATIONS];
    int undo_index = -1;
    int Q;
    printf("Enter the number of operations: ");
    scanf("%d", &Q);
    while (Q--) {
        int type;
        printf("Enter operation type (1=append, 2=delete, 3=print, 4=undo): ");
        scanf("%d", &type);
        if (type == 1) {
            char W[MAX_LEN];
            printf("Enter string to append: ");
            scanf("%s", W);
            append(&S, W, history, &undo_index);
        } else if (type == 2) {
            int k;
            printf("Enter number of characters to delete: ");
            scanf("%d", &k);
            delete (&S, k, history, &undo_index);
        } else if (type == 3) {
            int k;
            printf("Enter character position to print: ");
            scanf("%d", &k);
            print(S, k);
        } else if (type == 4) {
```

```
            undo(&S, history, &undo_index);
        }
    }
    free(S);
    for (int i = 0; i <= undo_index; i++) {
        free(history[i].state);
    }
    return 0;
}
void append(char **S, const char *W, Operation *history, int *undo_index) {
    history[++(*undo_index)].state = strdup(*S);
     strcat(*S, W);}
void delete (char **S, int k, Operation *history, int *undo_index) {
    history[++(*undo_index)].state = strdup(*S);
    int len = strlen(*S);
    if (k <= len) {
        (*S)[len - k] = '\0';
    }
}
void print(const char *S, int k) {
    int len = strlen(S);
    if (k > 0 && k <= len) {
        printf("Character at position %d: %c\n", k, S[k - 1]);
    } else {
        printf("Invalid position.\n");
    }
}
void undo(char **S, Operation *history, int *undo_index) {
    if (*undo_index >= 0) {

        strcpy(*S, history[(*undo_index)--].state);
    } else {
```

```
printf("No operations to undo.\n");
    }
}
```

Output:



Q4. Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).
Implement the MyQueue class:
void push(int x) Pushes element x to the back of the queue.
int pop() Removes the element from the front of the queue and returns it.
int peek() Returns the element at the front of the queue.
boolean empty() Returns true if the queue is empty, false otherwise.

Code:

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

#define MAX_SIZE 100
typedef struct {
    int data[MAX_SIZE];
    int top;
} Stack;
typedef struct {
    Stack stack1;
    Stack stack2;
```

```
MyQueue;
void stackPush(Stack *stack, int x) {
    if (stack->top < MAX_SIZE - 1) {
        stack->data[++stack->top] = x;
    }
}
int stackPop(Stack *stack) {
    if (stack->top >= 0) {
        return stack->data[stack->top--];
    }
    return -1;
}int stackPeek(Stack *stack) {
    if (stack->top >= 0) {
        return stack->data[stack->top];
    }
    return -1;
}

bool stackEmpty(Stack *stack) {
    return stack->top == -1;
}
MyQueue* myQueueCreate() {
    MyQueue *queue = (MyQueue *)malloc(sizeof(MyQueue));
    queue->stack1.top = -1;
    queue->stack2.top = -1;
    return queue;
}
void myQueuePush(MyQueue *queue, int x) {
    stackPush(&queue->stack1, x);
}

int myQueuePop(MyQueue *queue) {
```

```
    if (stackEmpty(&queue->stack2)) {

        while (!stackEmpty(&queue->stack1)) {
            stackPush(&queue->stack2, stackPop(&queue->stack1));
        }
    }
    return stackPop(&queue->stack2);
}

int myQueuePeek(MyQueue *queue) {
    if (stackEmpty(&queue->stack2)) {
        // Transfer elements from stack1 to stack2
        while (!stackEmpty(&queue->stack1)) {
            stackPush(&queue->stack2, stackPop(&queue->stack1));
        }
    }
    return stackPeek(&queue->stack2);
}

bool myQueueEmpty(MyQueue *queue) {
    return stackEmpty(&queue->stack1) && stackEmpty(&queue->stack2);
}

void myQueueFree(MyQueue *queue) {
    free(queue);
}
int main() {
    MyQueue *queue = myQueueCreate();

    myQueuePush(queue, 1);
    myQueuePush(queue, 2);
    printf("Front element: %d\n", myQueuePeek(queue));
```

```
 printf("Popped element: %d\n", myQueuePop(queue));
    printf("Queue empty: %s\n", myQueueEmpty(queue) ? "true" : "false");


    myQueueFree(queue);
    return 0;
}
```

Output :

Q5. You are given an array of strings tokens that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return an integer that represents the value of the expression.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define MAX_SIZE 100

typedef struct {
    int data[MAX_SIZE];
    int top;
} Stack;
void stackPush(Stack *stack, int value) {
```

```
    if (stack->top < MAX_SIZE - 1) {
        stack->data[++stack->top] = value;
    }
}
int stackPop(Stack *stack) {
    if (stack->top >= 0) {
        return stack->data[stack->top--];
    }
    return 0;
}
int evaluateRPN(char **tokens, int tokensSize) {
    Stack stack = {.top = -1};

    for (int i = 0; i < tokensSize; i++) {
        char *token = tokens[i];

        if (isdigit(token[0]) || (token[0] == '-' && isdigit(token[1]))) {

            stackPush(&stack, atoi(token));
        } else {
                int b = stackPop(&stack);
            int a = stackPop(&stack);
            int result = 0;

            if (strcmp(token, "+") == 0) {
                result = a + b;
            } else if (strcmp(token, "-") == 0) {
                result = a - b;
            } else if (strcmp(token, "*") == 0) {
                result = a * b;
            } else if (strcmp(token, "/") == 0) {
                result = a / b;
```

```
            stackPush(&stack, result);
        }
    }
return stackPop(&stack);
}int main() {
    char *tokens[] = {"2", "1", "+", "3", "*"};
    int tokensSize = sizeof(tokens) / sizeof(tokens[0]);

    int result = evaluateRPN(tokens, tokensSize);
    printf("Result: %d\n", result); // Output: 9

    return 0;
}
```

Output :