29-07-2024-day 4

Infix to postfix

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Function to return precedence of operators

int prec(char c) {

    if (c == '^')

        return 3;

    else if (c == '/' || c == '*')

        return 2;

    else if (c == '+' || c == '-')

        return 1;

    else

        return -1;

}

// Function to return associativity of operators

char associativity(char c) {

    if (c == '^')

        return 'R';

    return 'L'; // Default to left-associative

}

// The main function to convert infix expression to postfix expression

void infixToPostfix(char s[]) {

    char result[1000];

    int resultIndex = 0;

    int len = strlen(s);
```

```
char stack[1000];
int stackIndex = -1;

for (int i = 0; i < len; i++) {
    char c = s[i];

    // If the scanned character is an operand, add it to the output string.
    if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9')) {
        result[resultIndex++] = c;
    }
    // If the scanned character is an '(', push it to the stack.
    else if (c == '(') {
        stack[++stackIndex] = c;
    }
    // If the scanned character is an ')', pop and add to the output string from the stack
    // until an '(' is encountered.
    else if (c == ')') {
        while (stackIndex >= 0 && stack[stackIndex] != '(') {
            result[resultIndex++] = stack[stackIndex--];
        }
        stackIndex--; // Pop '('
    }
    // If an operator is scanned
    else {
        while (stackIndex >= 0 && (prec(s[i]) < prec(stack[stackIndex]) ||
                                   prec(s[i]) == prec(stack[stackIndex]) &&
                                       associativity(s[i]) == 'L')) {
            result[resultIndex++] = stack[stackIndex--];
        }
        stack[++stackIndex] = c;
    }
    }

    // Pop all the remaining elements from the stack
    while (stackIndex >= 0) {
        result[resultIndex++] = stack[stackIndex--];
    }

    result[resultIndex] = '\0';
    printf("%s\n", result);
}

// Driver code
int main() {
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";

    // Function call
    infixToPostfix(exp);

    return 0;
}
```

Output: abcd^e-fgh*+^*+i-

2.Array implementation in an queue

```
// C program for array implementation of queue
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a queue
struct Queue {
            int front, rear, size;
            unsigned capacity;
```

```
        int* array;
};

// function to create a queue
// of given capacity.
// It initializes size of queue as 0
struct Queue* createQueue(unsigned capacity)
{
        struct Queue* queue = (struct Queue*)malloc(
                sizeof(struct Queue));
        queue->capacity = capacity;
        queue->front = queue->size = 0;

        // This is important, see the enqueue
        queue->rear = capacity - 1;
        queue->array = (int*)malloc(
                queue->capacity * sizeof(int));
        return queue;
}

// Queue is full when size becomes
// equal to the capacity
int isFull(struct Queue* queue)
{
        return (queue->size == queue->capacity);
}

// Queue is empty when size is 0
int isEmpty(struct Queue* queue)
{
        return (queue->size == 0);
```

```
}

// Function to add an item to the queue.
// It changes rear and size
void enqueue(struct Queue* queue, int item)
{
	if (isFull(queue))
		return;
	queue->rear = (queue->rear + 1)
					% queue->capacity;
	queue->array[queue->rear] = item;
	queue->size = queue->size + 1;
	printf("%d enqueued to queue\n", item);
}

// Function to remove an item from queue.
// It changes front and size
int dequeue(struct Queue* queue)
{
	if (isEmpty(queue))
		return INT_MIN;
	int item = queue->array[queue->front];
	queue->front = (queue->front + 1)
					% queue->capacity;
	queue->size = queue->size - 1;
	return item;
}

// Function to get front of queue
int front(struct Queue* queue)
{
        if (isEmpty(queue))
                return INT_MIN;
        return queue->array[queue->front];
}

// Function to get rear of queue
int rear(struct Queue* queue)
{
        if (isEmpty(queue))
                return INT_MIN;
        return queue->array[queue->rear];
}

// Driver program to test above functions./
int main()
{
        struct Queue* queue = createQueue(1000);

        enqueue(queue, 10);
        enqueue(queue, 20);
        enqueue(queue, 30);
        enqueue(queue, 40);

        printf("%d dequeued from queue\n\n",
                dequeue(queue));

        printf("Front item is %d\n", front(queue));
        printf("Rear item is %d\n", rear(queue));

        return 0;
}
```

Output:

```
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue

Front item is 20
Rear item is 40
```

3.Linked list implementation in queue:

```
// A C program to demonstrate linked list based
// implementation of queue
#include <stdio.h>
#include <stdlib.h>

// A linked list (LL) node to store a queue entry
struct QNode {
	int key;
	struct QNode* next;
};

// The queue, front stores the front node of LL and rear
// stores the last node of LL
struct Queue {
	struct QNode *front, *rear;
};

// A utility function to create a new linked list node.
struct QNode* newNode(int k)
{
	struct QNode* temp
```

```
                = (struct QNode*)malloc(sizeof(struct QNode));
        temp->key = k;
        temp->next = NULL;
        return temp;
}

// A utility function to create an empty queue
struct Queue* createQueue()
{
        struct Queue* q
                = (struct Queue*)malloc(sizeof(struct Queue));
        q->front = q->rear = NULL;
        return q;
}

// The function to add a key k to q
void enQueue(struct Queue* q, int k)
{
        // Create a new LL node
        struct QNode* temp = newNode(k);

        // If queue is empty, then new node is front and rear
        // both
        if (q->rear == NULL) {
                q->front = q->rear = temp;
                return;
        }

        // Add the new node at the end of queue and change rear
        q->rear->next = temp;
        q->rear = temp;
```

```
}

// Function to remove a key from given queue q
void deQueue(struct Queue* q)
{
        // If queue is empty, return NULL.
        if (q->front == NULL)
                return;

        // Store previous front and move front one node ahead
        struct QNode* temp = q->front;

        q->front = q->front->next;

        // If front becomes NULL, then change rear also as NULL
        if (q->front == NULL)
                q->rear = NULL;

        free(temp);
}

// Driver code
int main()
{
        struct Queue* q = createQueue();
        enQueue(q, 10);
        enQueue(q, 20);
        deQueue(q);
        deQueue(q);
        enQueue(q, 30);
        enQueue(q, 40);
        enQueue(q, 50);
        deQueue(q);
        printf("Queue Front : %d \n", ((q->front != NULL) ? (q->front)->key : -1));
        printf("Queue Rear : %d", ((q->rear != NULL) ? (q->rear)->key : -1));
        return 0;
}
```

Output:

```
Queue Front : 40
Queue Rear : 50
```