Circular Queue using Array

Case Queue is not Full

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

Case Element cannot be inserted

- When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- Condition is : front== rear + 1;

Algorithm

Step 1: IF (REAR+1) % MAX = FRONT
Print " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

-

Program:

```
#include <stdio.h>
# define max 6
int queue[max];  // array declaration
int front=-1;
int rear=-1;
                                        // function to insert an element in a circular queue
void enqueue(int x)
{
   if(front==-1 && rear==-1)            // condition to check queue is empty
   {
      front=0;
      rear=0;
      queue[rear]=x;
   }
   else if((rear+1)%max==front)         // condition to check queue is full
   {
      printf("Queue is overflow..");
   }
   else
   {
      rear=(rear+1)%max;                // rear is incremented
      queue[rear]=x;                    // assigning a value to the queue at the rear position.
   }
}

                                        // function to delete the element from the queue
int dequeue()
{
   if((front==-1) && (rear==-1))        // condition to check queue is empty
   {
      printf("\nQueue is underflow..");
   }
 else if(front==rear)
{
  printf("\nThe dequeued element is %d", queue[front]);
  front=-1;
  rear=-1;
}
```

```
else
{
   printf("\nThe dequeued element is %d", queue[front]);
  front=(front+1)%max;
}
}
                                        // function to display the elements of a queue
void display()
{
   int i=front;
   if(front==-1 && rear==-1)
   {
      printf("\n Queue is empty..");
   }
   else
   {
      printf("\nElements in a Queue are :");
      while(i<=rear)
      {
         printf("%d,", queue[i]);
         i=(i+1)%max;
      }
   }
}
int main()
{
   int choice=1,x;   // variables declaration

   while(choice<4 && choice!=0)   // while loop
   {
   printf("\n Press 1: Insert an element");
   printf("\nPress 2: Delete an element");
   printf("\nPress 3: Display the element");
   printf("\nEnter your choice");
   scanf("%d", &choice);

   switch(choice)
   {
```

```
        case 1:

       printf("Enter the element which is to be inserted");
       scanf("%d", &x);
       enqueue(x);
       break;
       case 2:
       dequeue();
       break;
       case 3:
       display();

   }}
   return 0;
}
```

Priority Queues

```
# include<stdio.h>
# include<malloc.h>
  typedef struct node
    {
    int priority;
    int data;
    struct node *next;
}NODE;
NODE *front = NULL;

                                                    // insert method
void insert(int data,int priority)  {
    NODE *temp,*q;

    temp = (NODE *)malloc(sizeof(NODE));
    temp->data = data;
    temp->priority = priority;
```

```
        // condition to check whether the first element is empty or the element to be inserted has more priority than the first element

    if( front == NULL || priority < front->priority )
    {
        temp->next = front;
        front = temp;
    }
    else
    {
        q = front;
        while( q->next != NULL && q->next->priority <= priority )
            q=q->next;
        temp->next = q->next;
        q->next = temp;
    }
}
  // delete method

void del()
{
    NODE *temp;
                                        // condition to check whether the Queue is empty or not
    if(front == NULL)
        printf("Queue Underflow\n");
    else
    {
        temp = front;
        printf("Deleted item is %d\n", temp->data);
        front = front->next;
        free(temp);
    }
                                                    // display method
void display()  {
    NODE *ptr;
    ptr = front;
    if(front == NULL)
```

```
        printf("Queue is empty\n");
    else
    {
        printf("Queue is :\n");
        printf("Priority       Item\n");
        while(ptr != NULL)
        {
            printf("%5d        %5d\n",ptr->priority,ptr->data);
            ptr = ptr->next;
        }
    }
}
                                                        /*End of display*/

                                                       // main method
int main()
{
    int choice, data, priority;
    do
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter the data which is to be added in the queue : ");
                scanf("%d",&data);
                printf("Enter its priority : ");
                scanf("%d",&priority);
                insert(data,priority);
                break;
            case 2:
                del();
                break;
```

```
        case 3:
            display();
            break;
        case 4:
        break;
            default :
            printf("Wrong choice\n");
      }
   }while(choice!=4);

   return 0;
}
```

STACK APPLICATIONS

a) Infix To Postfix

Algorithm to convert Infix To Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1) Examine the next element in the input.
2) If it is **operand**, output it.
3) If it is **opening parenthesis**, push it on stack.
4) If it is an **operator**, then
   i) If stack is empty, push operator on stack.
   ii) If the top of stack is opening parenthesis, push operator on stack
   iii) If it has higher priority than the top of stack, push operator on stack.
   iv) Else pop the operator from the stack and output it, repeat step 4
5) If it is a ***closing parenthesis***, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
6) If there is **more input** go to step 1
7) If there is **no more input, pop** the remaining operators to output.

Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form,

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

So, the Postfix Expression is 23*21-/53*+

Program

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

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

#define SIZE 100



char stack[SIZE];
int top = -1;

void push(char item)
{
        if(top >= SIZE-1)
        {
                printf("\nStack Overflow.");
        }
        else
        {
                top = top+1;
                stack[top] = item;
        }
}


char pop()
{
        char item ;

        if(top <0)
        {
                printf("stack under flow: invalid infix expression");
                getchar();

                exit(1);
        }
        else
        {
                item = stack[top];
```

```
                top = top-1;
                return(item);
        }
}

int is_operator(char symbol)
{
        if(symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || symbol =='-')
        {
                return 1;
        }
        else
        {
        return 0;
        }
}



int precedence(char symbol)
{
        if(symbol == '^')       /* exponent operator, highest precedence*/
        {
                return(3);
        }
        else if(symbol == '*' || symbol == '/')
        {
                return(2);
        }
        else if(symbol == '+' || symbol == '-')         /* lowest precedence */
        {
                return(1);
        }
        else
        {
                return(0);
        }
}
```

```
void InfixToPostfix(char infix_exp[], char postfix_exp[])
{
	int i, j;
	char item;
	char x;

	push('(');
	strcat(infix_exp,")");

	i=0;
	j=0;
	item=infix_exp[i];

	while(item != '\0')
	{
		if(item == '(')
		{
			push(item);
		}
		else if( isdigit(item) || isalpha(item))
		{
			postfix_exp[j] = item;
			j++;
		}
		else if(is_operator(item) == 1)
		{
			x=pop();
			while(is_operator(x) == 1 && precedence(x)>= precedence(item))
			{
				postfix_exp[j] = x;
				j++;
				x = pop();
			}
			push(x);


			push(item);
```

```
        }
        else if(item == ')')
        {
                x = pop();
                while(x != '(')
                {
                        postfix_exp[j] = x;
                        j++;
                        x = pop();
                }
        }
        else
        {
                printf("\nInvalid infix Expression.\n");
                getchar();
                exit(1);
        }
        i++;


        item = infix_exp[i];
}
if(top>0)
{
        printf("\nInvalid infix Expression.\n");
        getchar();
        exit(1);
}
if(top>0)
{
        printf("\nInvalid infix Expression.\n");
        getchar();
        exit(1);
}


postfix_exp[j] = '\0';
```

```
}


int main()
{
        char infix[SIZE], postfix[SIZE];


        printf("Infix -Postfix.\n");
        printf("\nEnter Infix expression : ");
        gets(infix);

        InfixToPostfix(infix,postfix);
        printf("Postfix Expression: ");
        puts(postfix);

        return 0;
}
```

b) Evaluation of Postfix Expression

```
#include<stdio.h>
int stack[20];
int top = -1;

void push(int x)
{
   stack[++top] = x;
}

int pop()
{
   return stack[top--];
}

int main()
```

```
{
   char exp[20];
   char *e;
   int n1,n2,n3,num;
   printf("Enter the expression :: ");
   scanf("%s",exp);
   e = exp;
   while(*e != '\0')
   {
      if(isdigit(*e))
      {
         num = *e - 48;
         push(num);
      }
      else
      {
         n1 = pop();
         n2 = pop();
         switch(*e)
         {
         case '+':
         {
            n3 = n1 + n2;
            break;
         }
         case '-':
         {
            n3 = n2 - n1;
            break;
         }
         case '*':
         {
            n3 = n1 * n2;
            break;
         }
         case '/':
         {
            n3 = n2 / n1;
```

```
            break;
        }
        }
        push(n3);
      }
      e++;
   }
   printf("\nThe result of expression %s  =  %d\n\n",exp,pop());
   return 0;
}
```

C) Balancing Parenthesis

Program:

```
#include <stdio.h>
#include <stdlib.h>
#define bool int


struct sNode {
        char data;
        struct sNode* next;
};


void push(struct sNode** top_ref, int new_data);


int pop(struct sNode** top_ref);


bool isMatchingPair(char character1, char character2)
{
        if (character1 == '(' && character2 == ')')
                return 1;
```

```
        else if (character1 == '{' && character2 == '}')
                return 1;
        else if (character1 == '[' && character2 == ']')
                return 1;
        else
                return 0;
}


bool areBracketsBalanced(char exp[])
{
        int i = 0;


        struct sNode* stack = NULL;


        while (exp[i])
        {

                if (exp[i] == '{' || exp[i] == '(' || exp[i] == '[')
                        push(&stack, exp[i]);


                if (exp[i] == '}' || exp[i] == ')'
                        || exp[i] == ']') {


                        if (stack == NULL)
                                return 0;


                        else if (!isMatchingPair(pop(&stack), exp[i]))
                                return 0;
                }
                i++;
        }
```

```
        if (stack == NULL)
                return 1;
        else
                return 0;
}


int main()
{
        char exp[100] = "{()}[]";


        if (areBracketsBalanced(exp))
                printf("Balanced \n");
        else
                printf("Not Balanced \n");
        return 0;
}

void push(struct sNode** top_ref, int new_data)
{

        struct sNode* new_node
                = (struct sNode*)malloc(sizeof(struct sNode));

        if (new_node == NULL) {
                printf("Stack overflow n");
                getchar();
                exit(0);
        }


        new_node->data = new_data;


        new_node->next = (*top_ref);
```

```
        (*top_ref) = new_node;
}


int pop(struct sNode** top_ref)
{
        char x;
        struct sNode* top;


        if (*top_ref == NULL) {
                printf("Stack overflow n");
                getchar();
                exit(0);
        }
        else {
                top = *top_ref;
                x = top->data;
                *top_ref = top->next;
                free(top);
                return x;
        }
}
```