Bubble sort

```
#include <stdio.h>
int main(){
    int n;
    scanf("%d",&n);
    int arr[n];
    for(int i=0;i<n;i++){
        scanf("%d,",&arr[i]);
    }
    int p;
    int s;
    for(int i=0;i<n-1;i++){
        for(int j=0;j<n-i-1;j++){
            p++;
            if(arr[j]>arr[j+1]){
                int temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
                s++;
            }
        }
    }
    for(int i=0;i<n;i++){
        if(i<n-1){
            printf("%d, ",arr[i]);
        }else{
            printf("%d",arr[i]);
        }

    }

    printf("\n");
    printf("%d\n",p);
    printf("%d\n",s);
    return 0;
}
```

Insertion sort

```
#include <stdio.h>
int main(){
    int n;
    scanf("%d",&n);
    int arr[n];
    for(int i=0;i<n;i++){
        scanf("%d,",&arr[i]);
    }
    int p;
    int s;
    for(int i=1;i<n;i++){
        int key=arr[i];
        int j=i-1;
        while(j>=0 && arr[j]>key){
            p++;
            arr[j+1]=arr[j];
            s++;
            j--;
        }
        arr[j+1]=key;
        s++;
    }
    for(int i=0;i<n;i++){
        if(i<n-1){
            printf("%d, ",arr[i]);
        }else{
            printf("%d",arr[i]);
        }

    }

    printf("\n");
    printf("%d\n",p);
    printf("%d\n",s);

    return 0;

}
```

Modified bubble sort

```
#include <stdio.h>
int main(){
    int n;

    scanf("%d",&n);

    int arr[n];

    int c=0,s=0;

    for(int i=0;i<n;i++){

        scanf("%d,",&arr[i]);

    }

    for(int j=0; j<n-1;j++){

        int swapped=0;

        for(int i=0;i<n-j-1;i++){

            c++;

            if(arr[i]>arr[i+1]){

                int temp=arr[i];

                arr[i]=arr[i+1];

                arr[i+1]=temp;

                s++;

                swapped=1;

            }

        }

        if (swapped ==0){

            break;

        }

    }

    for(int i=0;i<n;i++){

        if(i<n-1){

            printf("%d, ",arr[i]);}

        else{

            printf("%d",arr[i]);}

    }

    printf("\n");

    printf("%d\n",c);

    printf("%d\n",s);


    return 0;

}
```

Selction sort

```
#include <stdio.h>

int main(){
    int n;
    scanf("%d",&n);
    int arr[n];
    for(int i=0;i<n;i++){
        scanf("%d,",&arr[i]);
    }
    int p=0;
    int s=0;
    for(int i=0;i<n-1;i++){
        int m=i;
        for(int j=i;j<n;j++){
            p++;
            if(arr[j]<arr[m]){
                m=j;
                s++;
            }
        }
        if(m!=i){
            int temp=arr[i];
            arr[i]=arr[m];
            arr[m]=temp;
            s++;
        }
    }
    for(int i=0;i<n-1;i++){
        printf("%d, ",arr[i]);


    }
    printf("%d ",arr[n-1]);



    printf("\n");
    printf("%d\n",p);
    printf("%d\n",s);
    return 0;
}
```

Stack- balanced not balanced

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

```
#include<malloc.h>
#include <sys/types.h>
#include <stdbool.h>

#define CHAR_MIN '\0'
typedef struct STACK
{
    char *array;
    int max_size;
    int top;
}mySTACK;

mySTACK* init_stack(int max_size);
void push(mySTACK *s, char x);
void show_stack(mySTACK *s);
char pop(mySTACK *s);
char get_top(mySTACK *s);
void delete_stack(mySTACK *s);
bool check_stack_overflow(mySTACK *s);
bool check_stack_underflow(mySTACK *s);
bool isBalanced(const char* str, ssize_t max_size);
ssize_t my_getline(char **lineptr, size_t *n, FILE *stream);
int main()
{
    bool flag_balanced;
    char *line = NULL;
    size_t len = 0;
    ssize_t nread;
    nread = my_getline(&line, &len, stdin);
    if (nread != -1) {
    } else {
        printf("Error or end of file.\n");
    }
    flag_balanced = isBalanced(line, nread);
    if(flag_balanced)
    {
        printf("Balanced\n");
    }
    else
```

```
        printf("NOT Balanced\n");
    free(line);
    return 0;
}
mySTACK* init_stack(int max_size) {
    mySTACK *s = (mySTACK *)malloc(sizeof(mySTACK));
    if (!s) return NULL;
    s->array = (char *)malloc(sizeof(char) * max_size);
    if (!s->array) {
        free(s);
        return NULL;
    }
    s->max_size = max_size;
    s->top = -1;  // empty stack
    return s;
}
bool check_stack_overflow(mySTACK *s) {
    return s->top == s->max_size - 1;
}
bool check_stack_underflow(mySTACK *s) {
    return s->top == -1;
}
void push(mySTACK *s, char x) {
    if (check_stack_overflow(s)) {
        return;
    }
    s->array[++s->top] = x;
}

char pop(mySTACK *s) {
    if (check_stack_underflow(s)) {
        return CHAR_MIN;  // Return null char on underflow
    }
    return s->array[s->top--];
}
char get_top(mySTACK *s) {
    if (check_stack_underflow(s)) {
        return CHAR_MIN;
    }
    return s->array[s->top];
}
void delete_stack(mySTACK *s) {
    if (s) {
        if (s->array) free(s->array);
        free(s);
    }
}
bool isBalanced(const char* str,  ssize_t max_size)
{
    if (!str) return true;
    mySTACK *s = init_stack((int)max_size);
    if (!s) {
        return false;
    }
    for (ssize_t i = 0; i < max_size; i++) {
        char c = str[i];
        if (c == '\0' || c == '\n') break;

        if (c == '(' || c == '[' || c == '{') {
            push(s, c);
        } else if (c == ')' || c == ']' || c == '}') {
            if (check_stack_underflow(s)) {
                delete_stack(s);
                return false;
            }
            char top_char = pop(s);
            if ((c == ')' && top_char != '(') ||
                (c == ']' && top_char != '[') ||
                (c == '}' && top_char != '{')) {
                delete_stack(s);
                return false;
            }
        }
    }
    bool balanced = check_stack_underflow(s);
    delete_stack(s);
    return balanced;
}
```

```
ssize_t my_getline(char **lineptr, size_t *n, FILE *stream)
{
    if (lineptr == NULL || n == NULL || stream == NULL)
    {
        return -1;
    }

    if (*lineptr == NULL)
    {
        *n = 0; // Initialize size to 0
        *lineptr = (char *)malloc(128);
        if (*lineptr == NULL) {
            return -1; // Allocation failed
        }
        *n = 128;
    }
    size_t pos = 0;
    int c;
    while ((c = fgetc(stream)) != EOF)
    {
        if (pos >= *n - 1)
        {
            size_t new_size = *n * 2;
            char *new_lineptr = (char *)realloc(*lineptr, new_size);
            if (new_lineptr == NULL) {
                return -1;
            }
            *lineptr = new_lineptr;
            *n = new_size;
        }
        (*lineptr)[pos++] = (char)c;
        if (c == '\n')
        {
            break;
        }
    }
    if (pos == 0 && c == EOF)
    {
        return -1;
    }
    (*lineptr)[pos] = '\0';
    return (ssize_t)pos;
}
```

Stack- infix

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include<malloc.h>
#include <sys/types.h> // for ssize_t
#include <stdbool.h>

#define CHAR_MIN_T '\0'
typedef struct STACK
{
    char *array;
    int max_size;
    int top;
}mySTACK;
mySTACK* init_stack(int max_size);
void push(mySTACK *s, char x);
void show_stack(mySTACK *s);
char pop(mySTACK *s);
char get_top(mySTACK *s);
void delete_stack(mySTACK *s);
bool check_stack_overflow(mySTACK *s);
bool check_stack_underflow(mySTACK *s);

bool isBalanced(const char* str, ssize_t max_size);

ssize_t my_getline(char **lineptr, size_t *n, FILE *stream);

int precedence(char op);
void infixToPostfix(const char* infix, char* postfix, ssize_t max_size);

int main() {
    char *infix=NULL;
    char *postfix=NULL;
    size_t len = 0;
    ssize_t nread;
    nread = my_getline(&infix, &len, stdin);

    if (nread != -1) {
    } else {
        printf("Error or end of file.\n");
    }

    postfix = malloc(sizeof(char)* (nread+1));
    infixToPostfix(infix, postfix, nread);
    printf("%s\n", postfix);
    return 0;
}
int precedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return 0;
    }
}
void infixToPostfix(const char* infix, char* postfix, ssize_t max_size)
{
    mySTACK *stack = init_stack((int)max_size);
    if (!stack) {
        postfix[0] = '\0';
        return;
    }
    int k = 0;
    for (ssize_t i = 0; i < max_size; i++) {
        char c = infix[i];
        if (c == '\0' || c == '\n') break;
        if (c == ' ' || c == '\t') {
            continue;
        }

        if (isalnum(c)) {
            postfix[k++] = c;
        }
        else if (c == '(') {
            push(stack, c);
        }
        else if (c == ')') {
            bool found_open = false;
            while (!check_stack_underflow(stack)) {
                char top = pop(stack);
                if (top == '(') {
                    found_open = true;
                    break;
                }
                postfix[k++] = top;
            }
            if (!found_open) {

                break;
            }
        }
        else {
            while (!check_stack_underflow(stack) && precedence(get_top(stack)) >= precedence(c)) {
                postfix[k++] = pop(stack);
            }
            push(stack, c);
        }
    }
    while (!check_stack_underflow(stack)) {
        char top = pop(stack);
        if (top == '(') {
            continue;
        }
        postfix[k++] = top;
    }
    postfix[k] = '\0';
    delete_stack(stack);
}
mySTACK* init_stack(int max_size) {
    mySTACK *s = (mySTACK *)malloc(sizeof(mySTACK));
    if (!s) return NULL;
    s->array = (char *)malloc(sizeof(char) * max_size);
    if (!s->array) {
        free(s);
        return NULL;
    }
    s->max_size = max_size;
    s->top = -1;
    return s;
}
bool check_stack_overflow(mySTACK *s) {
    return s->top == s->max_size - 1;
}
bool check_stack_underflow(mySTACK *s) {
    return s->top == -1;
}
void push(mySTACK *s, char x) {
    if (check_stack_overflow(s)) {

        return;
    }
    s->array[++s->top] = x;
}

char pop(mySTACK *s) {
    if (check_stack_underflow(s)) {
        return CHAR_MIN_T;
    }
    return s->array[s->top--];
}

char get_top(mySTACK *s) {
    if (check_stack_underflow(s)) {
        return CHAR_MIN_T;
    }
    return s->array[s->top];
}
void delete_stack(mySTACK *s) {
    if (s) {
        if (s->array) free(s->array);
        free(s);
    }
}
bool isBalanced(const char* str, ssize_t max_size) {
    // Not used in this program, can be empty or implemented as needed
    return true;
}
ssize_t my_getline(char **lineptr, size_t *n, FILE *stream)
{
    if (lineptr == NULL || n == NULL || stream == NULL)
    {
        return -1;  // Invalid arguments
    }
    if (*lineptr == NULL)
    {
        *n = 0; // Initialize size to 0
        *lineptr = (char *)malloc(128);
        if (*lineptr == NULL) {
            return -1;
        }
        *n = 128;
    }
    size_t pos = 0;
    int c;
    while ((c = fgetc(stream)) != EOF)
    {
        if (pos >= *n - 1)
        {
            size_t new_size = *n * 2;
            char *new_lineptr = (char *)realloc(*lineptr, new_size);
            if (new_lineptr == NULL) {
                return -1;
            }
            *lineptr = new_lineptr;
            *n = new_size;
        }

        (*lineptr)[pos++] = (char)c;
        if (c == '\n')
        {
            break;
        }
    }
    if (pos == 0 && c == EOF)
    {
        return -1;
    }
    (*lineptr)[pos] = '\0';
    return (ssize_t)pos;
}
```

Stack q3

```
#include<stdio.h>
#include<malloc.h>
#include <stdbool.h>
#define INT_MIN -9999
typedef struct STACK
{
    int *array;
    int max_size;
    int top;
}mySTACK;
mySTACK* init_stack(int max_size);
void push(mySTACK *s, int x);
void show_stack(mySTACK *s);
int pop(mySTACK *s);
int get_top(mySTACK *s);
void delete_stack(mySTACK *s);
bool check_stack_overflow(mySTACK *s);
bool check_stack_underflow(mySTACK *s);

int main()
```

```
{
    int op_code, max_size;
    int new_element, poped_element;
    bool flag_empty, flag_full;

    scanf("%d", &max_size);
    mySTACK *s1= NULL;
    while(1)
    {
        scanf("%d", &op_code);
        //printf("Op_code =%d\n", op_code);
        if(op_code ==0)
        {
            s1 = init_stack(max_size);
        }
        else if( op_code==1)
        {
            scanf("%d", &new_element);
            push(s1, new_element);
        }
        else if(op_code ==2)
        {
            poped_element = pop(s1);
            printf("%d\n", poped_element);
        }
        else if(op_code == 3)
        {
            flag_empty = check_stack_underflow(s1);
            printf("%d\n", flag_empty);
        }
        else if(op_code ==4)
        {
            show_stack(s1);
        }
        else if(op_code ==5)
        {
            flag_full = check_stack_overflow(s1);
            printf("%d\n", flag_full);
        }
        else if(op_code==6)
        {
            int top_element = get_top(s1);
            printf("%d\n", top_element);
        }
        else if(op_code ==9)
            break;
    }
    return 0;
}
mySTACK* init_stack(int max_size)
{
    mySTACK *s = (mySTACK*)malloc(sizeof(mySTACK));
    s->array = (int*)malloc(max_size * sizeof(int));
    s->max_size = max_size;
    s->top = -1;
    return s;
}
void push(mySTACK *s, int x)
{
    if(check_stack_overflow(s))
    {
        printf("STACK overflow\n");
        return;
    }
    s->array[++(s->top)] = x;
}

void show_stack(mySTACK *s)
{
    if(check_stack_underflow(s))
    {
        printf("STACK Underflow\n");
        return;
    }
    for(int i = 0; i <= s->top; i++)
    {
        printf("%d", s->array[i]);
        if(i < s->top)
```

```
            printf(", ");
    }
    printf("\n");
}
int pop(mySTACK *s)
{
    if(check_stack_underflow(s))
    {
        printf("STACK Underflow\n");
        return INT_MIN;
    }
    return s->array[(s->top)--];
}
int get_top(mySTACK *s)
{
    if(check_stack_underflow(s))
        return INT_MIN;
    return s->array[s->top];
}

bool check_stack_overflow(mySTACK *s)
{
    return (s->top == s->max_size - 1);
}
bool check_stack_underflow(mySTACK *s)
{
    return (s->top == -1);
}
void delete_stack(mySTACK *s)
{
    if(s)
    {
        if(s->array)
            free(s->array);
        free(s);
    }
}
```

Stack q4

```
#include<stdio.h>
#include<malloc.h>
#include<stdbool.h>
#define INT_MIN -9999
typedef struct STACK
{
    int *array;
    int max_size;
    int top;
} mySTACK;
mySTACK* init_stack(int max_size);
void push(mySTACK *s, int x);
void show_stack(mySTACK *s);
int pop(mySTACK *s);
int get_top(mySTACK *s);
void delete_stack(mySTACK *s);
bool check_stack_overflow(mySTACK *s);
bool check_stack_underflow(mySTACK *s);
int main()
{
    int op_code, max_size;
    int new_element, poped_element;
    bool flag_empty, flag_full;
    scanf("%d", &max_size);
    mySTACK *s1 = NULL;

    while(1)
    {
        scanf("%d", &op_code);

        if(op_code == 0)
        {
            s1 = init_stack(max_size);
        }
        else if(op_code == 1)
        {
            scanf("%d", &new_element);
            push(s1, new_element);
        }
        else if(op_code == 2)
        {
            poped_element = pop(s1);
            printf("%d\n", poped_element);
        }
        else if(op_code == 3)
        {
            flag_empty = check_stack_underflow(s1);
            printf("%d\n", flag_empty);
        }
        else if(op_code == 4)
        {
            show_stack(s1);
        }
        else if(op_code == 5)
        {
            flag_full = check_stack_overflow(s1);
            printf("%d\n", flag_full);
        }
        else if(op_code == 6)
        {
            int top_element = get_top(s1);
            printf("%d\n", top_element);
        }
        else if(op_code == 9)
            break;
    }
    return 0;
}
mySTACK* init_stack(int max_size)
{
    mySTACK *s = malloc(sizeof(mySTACK));
    if(s == NULL) return NULL;

    s->array = malloc(sizeof(int) * max_size);
    if(s->array == NULL)
    {
        free(s);
        return NULL;
    }
    s->max_size = max_size;
    s->top = -1;
    return s;
}
void push(mySTACK *s, int x)
{
    if(check_stack_overflow(s)) {
        printf("STACK overflow\n");
        return;
    }
    s->array[++(s->top)] = x;
}
void show_stack(mySTACK *s)
{
    if(check_stack_underflow(s))
        return;

    for(int i = 0; i <= s->top; i++)
    {
        if(i > 0) printf(", ");
        printf("%d", s->array[i]);
    }
    printf("\n");
}
int pop(mySTACK *s)
{
    if(check_stack_underflow(s)) {
        printf("STACK Underflow\n");
        return INT_MIN;
    }
    return s->array[(s->top)--];
}
int get_top(mySTACK *s)
{
    if(check_stack_underflow(s))
        return INT_MIN;
    return s->array[s->top];
}
```

```
bool check_stack_overflow(mySTACK *s)
{
    return (s->top == s->max_size - 1);
}
bool check_stack_underflow(mySTACK *s)
{
    return (s->top == -1);
}
void delete_stack(mySTACK *s)
{
    if(s)
    {
        if(s->array)
            free(s->array);
        free(s);
    }
}
```

Circular Queue

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#define INT_MIN -9999
typedef struct CircularQUEUE {
    int *array;
    int front;
    int rear;
    int size;
    int max_size;
} myQueue;
myQueue* init_Queue(int max_size);
void enQueue(myQueue *q, int x);
int deQueue(myQueue *q);
void show_queue(myQueue *q);
bool isQueueOverflow(myQueue *q);
bool isQueueUnderflow(myQueue *q);

int main() {
    myQueue *q = NULL;
    int choice, x, max_size;
    scanf("%d", &max_size);
    while(1) {
        scanf("%d", &choice);

        if(choice == 0) {
            if(q == NULL)
                q = init_Queue(max_size);
        }
        else if(choice == 1) {
            scanf("%d", &x);
            enQueue(q, x);
        }
        else if(choice == 2) {
            x = deQueue(q);
            printf("%d\n", x);
        }
        else if(choice == 3) {
            printf("%d\n", isQueueOverflow(q));
        }
        else if(choice == 4) {
            show_queue(q);
        }
        else if(choice == 5) {
            printf("%d\n", isQueueUnderflow(q));
        }
        else if(choice == 9)
            break;
        else
            printf("Invalid choice\n");
    }
    return 0;
}
myQueue* init_Queue(int max_size) {
    myQueue *q = (myQueue*)malloc(sizeof(myQueue));
    q->array = (int*)malloc(max_size * sizeof(int));
    q->front = -1;
    q->rear = -1;
    q->size = 0;
    q->max_size = max_size;
    return q;
}
void enQueue(myQueue *q, int x) {
    if(isQueueOverflow(q)) {
        printf("Queue Overflow\n");
        return;
    }
    if(q->front == -1)
        q->front = 0;
    q->rear = (q->rear + 1) % q->max_size;
    q->array[q->rear] = x;
    q->size++;
}
int deQueue(myQueue *q) {
    if(isQueueUnderflow(q)) {
        return INT_MIN;  // Will print -9999 in main
    }
    int val = q->array[q->front];

    if(q->front == q->rear) {
        q->front = -1;
        q->rear = -1;
    }
    else {
        q->front = (q->front + 1) % q->max_size;
    }
    q->size--;
    return val;
}
void show_queue(myQueue *q) {
    if(isQueueUnderflow(q)) return;

    int count = q->size;
    int i = q->front;
    while(count--) {
        printf("%d", q->array[i]);
        if(count) printf(", ");
        i = (i + 1) % q->max_size;
    }
    printf("\n");
}
bool isQueueOverflow(myQueue *q) {
    return (q->size == q->max_size);
}
bool isQueueUnderflow(myQueue *q) {
    return (q->size == 0);
}
```

Linear queue

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#define INT_MIN -9999
typedef struct QUEUE {
    int *array;
    int rear;
    int front;
    int size;
    int max_size;
} myQueue;
myQueue* init_Queue(int max_size);
void enQueue(myQueue * q, int x);
int deQueue(myQueue *q);
void show_queue(myQueue *q);
void printQueue(myQueue *Q);
bool isQueueOverflow(myQueue *q);
bool isQueueUnderflow(myQueue *q);
int main() {
    myQueue *q;
    int choice=0, x, max_size;
    scanf("%d", &max_size);
    while(1) {
        scanf("%d", &choice);
        if(choice==0) {
            q = init_Queue(max_size);
        }
        else if(choice==1) {
            scanf("%d", &x);
            enQueue(q, x);
        }
        else if(choice==2) {
            x = deQueue(q);
            if(x!= INT_MIN)
                printf("%d\n", x);
        }
        else if(choice ==3) {
            printf("%d\n", isQueueOverflow(q));
        }
        else if(choice==4) {
            show_queue(q);
        }
        else if(choice ==5) {
            printf("%d\n", isQueueUnderflow(q));
        }
        else if(choice ==9)
            break;
        else
            printf("Invalid choice\n");
    }
    return 0;
}
myQueue* init_Queue(int max_size) {
    myQueue *Q = (myQueue*)malloc(sizeof(myQueue));
    Q->array = (int*)malloc(max_size * sizeof(int));
    Q->front = 0;
    Q->rear = -1;
    Q->size = 0;
    Q->max_size = max_size;
    return Q;
}
void enQueue(myQueue *Q, int x) {
    if(isQueueOverflow(Q)) {
        printf("Queue Overflow\n");
        return;
    }
    Q->rear++;
    Q->array[Q->rear] = x;
    Q->size++;
```

```
}
int deQueue(myQueue *Q) {
    if(isQueueUnderflow(Q)) {
        printf("%d\n", INT_MIN);
        return INT_MIN;
    }
    int val = Q->array[Q->front];
    Q->front++;
    Q->size--;
    return val;
}
void show_queue(myQueue *Q) {
    if(isQueueUnderflow(Q)) return;
    for(int i = Q->front; i <= Q->rear; i++) {
        if(i == Q->rear)
            printf("%d", Q->array[i]);
        else
            printf("%d, ", Q->array[i]);
    }
    printf("\n");
}
bool isQueueOverflow(myQueue *Q) {
    return (Q->rear == Q->max_size - 1);
}
bool isQueueUnderflow(myQueue *Q) {
    return (Q->front > Q->rear);
}
void printQueue(myQueue *Q) {
    printf("\n Queue structure details: \n");
    printf("\t Queue Adress: %p\n", Q);
    printf("\t Q->array:%p\n", Q->array);
    printf("\t Q->max_size:%d\n", Q->max_size);
    printf("Q->front: %d, Q->rear: %d \n", Q->front, Q->rear);
    printf("\n");
}
```

Fill the code of of following link list function.

i) insertion at beginning: insertAtBeg(ListNode **head, int x)

ii) insertion at end: insertAtLast(ListNode **head, int x)

iii) printing the list : printList(ListNode *head)

iv) delete the item from link list at beginning - if link list is empty return -9999

v) delete the item from link list from the last - if link list is empty return -9999

```
#include <stdio.h>
#include <stdlib.h>
typedef struct ListNode {
    int data;
    struct ListNode *next;
} ListNode;
void init_link_list(ListNode **head);
void insertAtBeg(ListNode **head, int x);
void insertAtLast(ListNode **head, int x);
void printList(ListNode *head);
int deleteFromBeg(ListNode **head);
int deleteFromLast(ListNode **head);
int main() {
    ListNode *head;
    int choice = 0, x;
    init_link_list(&head);
    while (1) {
        scanf("%d", &choice);
        if (choice == 0) {
            init_link_list(&head);
        } else if (choice == 1) {
            scanf("%d", &x);
            insertAtBeg(&head, x);
        } else if (choice == 2) {
            scanf("%d", &x);
            insertAtLast(&head, x);
        } else if (choice == 3) {
            printList(head);
        } else if (choice == 5) {
            printf("%d\n", deleteFromBeg(&head));
        } else if (choice == 6) {
            printf("%d\n", deleteFromLast(&head));
        } else {
            break;
        }
    }
    return 0;
```

```
}
void init_link_list(ListNode **head) {
    (*head) = NULL;
}
void insertAtBeg(ListNode **head, int x) {
    ListNode *newNode = (ListNode *)malloc(sizeof(ListNode));
    newNode->data = x;
    newNode->next = *head;
    *head = newNode;
}
void insertAtLast(ListNode **head, int x) {
    ListNode *newNode = (ListNode *)malloc(sizeof(ListNode));
    newNode->data = x;
    newNode->next = NULL;
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    ListNode *temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}
void printList(ListNode *head) {
    ListNode *temp = head;
    while (temp != NULL) {
        printf("%d", temp->data);
        if (temp->next != NULL) printf(", ");
        temp = temp->next;
    }
    printf("\n");
}
int deleteFromBeg(ListNode **head) {
    if (*head == NULL) return -999;
    ListNode *temp = *head;
    int val = temp->data;
    *head = (*head)->next;
    free(temp);
    return val;
}
int deleteFromLast(ListNode **head) {
    if (*head == NULL) return -999;
    ListNode *temp = *head;
    if (temp->next == NULL) {
        int val = temp->data;
        free(temp);
        *head = NULL;
        return val;
    }
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    int val = temp->next->data;
    free(temp->next);
    temp->next = NULL;
    return val;
}
```

Circular linked list

```
#include <stdio.h>
#include <stdlib.h>
typedef struct ListNode {
    int data;
    struct ListNode *next;
} ListNode;
void init_link_list(ListNode **head);
void insertAtBeg(ListNode **head, int x);
void insertAtLast(ListNode **head, int x);
void printList(ListNode *head);
int main() {
    ListNode *head;
    int choice = 0, x;
    init_link_list(&head);
    while (1) {
        scanf("%d", &choice);
        if (choice == 0) {
            init_link_list(&head);
        }
        else if (choice == 1) {
            scanf("%d", &x);
            insertAtBeg(&head, x);
        }
        else if (choice == 2) {
            scanf("%d", &x);
            insertAtLast(&head, x);
        }
        else if (choice == 3) {
            printList(head);
        }
        else {
            break;
        }
    }
    return 0;
}
void init_link_list(ListNode **head) {
    *head = NULL;
}
void insertAtBeg(ListNode **head, int x) {
    ListNode *newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->data = x;
    if (*head == NULL) {
        newNode->next = newNode;
        *head = newNode;
    } else {
        ListNode *temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        newNode->next = *head;
        temp->next = newNode;
        *head = newNode;
    }
}
void insertAtLast(ListNode **head, int x) {
    ListNode *newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->data = x;
    if (*head == NULL) {
        newNode->next = newNode;
        *head = newNode;
    } else {
        ListNode *temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = *head;
    }
}
void printList(ListNode *head) {
    if (head == NULL) {
        return;
    }
    ListNode *temp = head;
    do {
        printf("%d", temp->data);
        temp = temp->next;
        if (temp != head) {
            printf(", ");
        }
    } while (temp != head);
    printf("\n");
}
```

The menu driven main function is already implemented. The menu for main function are as follows:

1 - to insert element at beginning

2 - to insert element at end

3 - to print element of link list separated by a comma and space

4 - to exit the main function

```
#include <stdio.h>
#include <stdlib.h>
typedef struct ListNode{
    int data;
    struct ListNode *next;
} ListNode;


void init_link_list(ListNode **head);
void insertAtBeg(ListNode **head, int x);
void insertAtLast(ListNode **head, int x);
void printList(ListNode *head);
int main()
{
   ListNode *head;
   int choice=0, x;
   init_link_list(&head);
   while(1)
   {
      scanf("%d", &choice);
      if(choice==0)
      {
         init_link_list(&head);
      }
      else if(choice==1)
      {
         scanf("%d", &x);
         insertAtBeg(&head, x);
      }
      else if(choice==2)
      {
         scanf("%d", &x);
         insertAtLast(&head, x);
      }
      else if(choice==3)
      {
         printList(head);
      }
      else
         break;
   }
   return 0;
}
void init_link_list(ListNode **head){
   (*head) = NULL;
}
void insertAtBeg(ListNode **head, int x)
{
    ListNode *newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->data = x;
    if (*head == NULL) {
        newNode->next = newNode;  // first node points to itself
        *head = newNode;
    } else {
        ListNode *temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        newNode->next = *head;
        temp->next = newNode;
        *head = newNode; // update head
    }
}
void printList(ListNode *head)
{
    if (head == NULL) {
        return;
    }
    ListNode *temp = head;
    do {
        printf("%d", temp->data);
        temp = temp->next;
        if (temp != head) {
            printf(", ");
        }
    } while (temp != head);
    printf("\n");
}
void insertAtLast(ListNode **head, int x)
{
    ListNode *newNode = (ListNode*)malloc(sizeof(ListNode));
    newNode->data = x;
    if (*head == NULL) {
        newNode->next = newNode;  // first node points to itself
        *head = newNode;
    } else {
        ListNode *temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = *head;
    }
}
```

Merge sort

```
#include <stdio.h>
#include <stdlib.h>
void printArrayPart(int arr[], int l, int r)
{
    for(int i = l; i <= r; i++)
    {
        if(i == r)
            printf("%d\n", arr[i]);
        else
            printf("%d, ", arr[i]);
    }
}
void merge(int arr[], int l,int m,int r)
{
    int i,j,k;
    int n1 = m - l + 1;
    int n2 = r-m;
    int L[n1],R[n2];
    for(i=0;i<n1;i++)
    {
        L[i] = arr[l+i];
    }

    for(j=0;j<n2;j++)
    {
        R[j] = arr[m+1+j];
    }

    i=0;
    j=0;
    k=l;
    while(i<n1 && j<n2)
    {
        if(L[i]<=R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while(i<n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while(j<n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
    printArrayPart(arr, l, r);
}
void mergeSort(int arr[],int l,int r,int n)
{
    if(l<r)
    {
        int m =(r+l)/2;
        mergeSort(arr,l,m,n);
        mergeSort(arr,m+1,r,n);

        merge(arr,l,m,r);
    }
}
```

```
int main()
{
    int n;
    scanf("%d",&n);
    int arr[n];
    for(int i=0;i<n;i++)
    {
        scanf("%d, ",&arr[i]);
    }
    mergeSort(arr,0,n-1,n);
    for(int i=0;i<n;i++)
    {
        if(i==n-1)
            printf("%d\n",arr[i]);
        else
            printf("%d, ",arr[i]);
    }
    return 0;
}
```

Quicksort

```
#include<stdio.h>
#include<stdlib.h>
void swap(int*a,int*b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int arr[],int low, int high)
{
    int pivot = arr[low];
    int i = low +1;
    for(int j = low + 1;j<=high;j++)
    {
        if(arr[j]<pivot)
        {
            swap(&arr[i],&arr[j]);
            i++;
        }
    }
    swap(&arr[low],&arr[i-1]);
    return (i-1);
}
void quickSort(int arr[], int low, int high,int n)
{
    if(low<high)
    {
        int pi = partition(arr,low,high);
        quickSort(arr,low,pi-1,n);
        quickSort(arr,pi+1,high,n);
        for(int i=0;i<n;i++)
        {
            if(i==n-1)
                printf("%d\n",arr[i]);
            else
                printf("%d, ",arr[i]);
        }
    }
}
int main()
{
    int n;
    scanf("%d",&n);
    int arr[n];
    for(int i=0;i<n;i++)
    {
        scanf("%d, ",&arr[i]);
    }
    for(int i=0;i<n;i++)
    {
        if(i==n-1)
            printf("%d\n",arr[i]);
        else
            printf("%d, ",arr[i]);
    }
    quickSort(arr,0, n-1,n);

    for(int i=0;i<n;i++)
```

```
  {
    if(i==n-1)
      printf("%d\n",arr[i]);
    else
      printf("%d, ",arr[i]);
  }
  return 0;
}
```

The menu-driven main function is already implemented. The menu for the main function are as follows:

1 - to insert element x into BST

in case of a duplicate element – PRINT – DUPLICATE ELEMENT <VALUE OF X> NOT INSERTED

2 - to print inorder traversal of BST

3- to print preorder traversal of BST

4- to print postorder traversal of BST

```
#include <stdio.h>
#include <stdlib.h>
typedef struct BSTNode {
  int data;
  struct BSTNode *left, *right;
} BSTNode;
BSTNode* create_node(int x) {
  BSTNode* new_node = (BSTNode*)malloc(sizeof(BSTNode));
  new_node->data = x;
  new_node->left = new_node->right = NULL;
  return new_node;
}
void add_node(BSTNode **root, int x) {
  if (*root == NULL) {
    *root = create_node(x);
    return;
  }
  if (x < (*root)->data) {
    add_node(&((*root)->left), x);
  } else if (x > (*root)->data) {
    add_node(&((*root)->right), x);
  } else {
    printf("DUPLICATE ELEMENT %d NOT INSERTED\n", x);
  }
}
```

```
void inorder(BSTNode *root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d, ", root->data);
    inorder(root->right);
}
void preorder(BSTNode *root) {
    if (root == NULL) return;
    printf("%d, ", root->data);
    preorder(root->left);
    preorder(root->right);
}
void postorder(BSTNode *root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    printf("%d, ", root->data);
}
int main() {
    BSTNode* root = NULL;
    int choice, x;
    while (scanf("%d", &choice) != EOF) {
        switch (choice) {
            case 1:
                scanf("%d", &x);
                add_node(&root, x);
                break;
            case 2:
                inorder(root);
                printf("\n");
                break;
            case 3:
                preorder(root);
                printf("\n");
                break;
            case 4:
                postorder(root);
                printf("\n");
                break;
            default:
                break;
        }
    }
    return 0;
}
```

the menu-driven main function is already implemented. The menu for the main function are as follows:

1 - to insert element x into BST

in case of a duplicate element – PRINT – DUPLICATE ELEMENT <VALUE OF X> NOT INSERTED

2 - to print inorder traversal of BST

3- to print preorder traversal of BST

4- to print postorder traversal of BST

5 - to search an element x form a BST

6 - to delete an element x from a BST

```
#include <stdio.h>
#include <stdlib.h>
typedef struct BSTNode {
    int data;
    struct BSTNode *left, *right;
} BSTNode;
BSTNode* createNode(int x) {
    BSTNode* newNode = (BSTNode*)malloc(sizeof(BSTNode));
    newNode->data = x;
    newNode->left = newNode->right = NULL;
    return newNode;
}
void add_node(BSTNode **root, int x) {
    if (*root == NULL) {
        *root = createNode(x);
        return;
    }
    if (x < (*root)->data)
        add_node(&((*root)->left), x);
    else if (x > (*root)->data)
        add_node(&((*root)->right), x);
    else
        printf("DUPLICATE ELEMENT %d NOT INSERTED\n", x);
}
void inorder(BSTNode *root) {
    if (root) {
        inorder(root->left);
        printf("%d, ", root->data);
        inorder(root->right);
    }
}
void preorder(BSTNode *root) {
    if (root) {
        printf("%d, ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
void postorder(BSTNode *root) {
    if (root) {
        postorder(root->left);
        postorder(root->right);
        printf("%d, ", root->data);
    }
}
BSTNode* findMin(BSTNode* root) {
    while (root->left) root = root->left;
    return root;
}
int delete_BST(BSTNode **root, int x) {
    if (*root == NULL)
        return -9999;
    if (x < (*root)->data)
        return delete_BST(&((*root)->left), x);
    else if (x > (*root)->data)
        return delete_BST(&((*root)->right), x);
    else {
        if ((*root)->left == NULL) {
            BSTNode* temp = *root;
            *root = (*root)->right;
            int val = temp->data;
            free(temp);
            return val;
        }
        else if ((*root)->right == NULL) {
            BSTNode* temp = *root;
            *root = (*root)->left;
            int val = temp->data;
            free(temp);
            return val;
        }
        else {
            BSTNode* temp = findMin((*root)->right);
            int val = (*root)->data;
            (*root)->data = temp->data;
            delete_BST(&((*root)->right), temp->data);
            return val;
        }
    }
}
int search(BSTNode *root, int x) {
    if (!root) return 0;
    if (x == root->data) return 1;
    if (x < root->data) return search(root->left, x);
    return search(root->right, x);
}
int main() {
    BSTNode *root = NULL;
    int choice, x, res;
    while (1) {
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                scanf("%d", &x);
                add_node(&root, x);
                break;
            case 2:
                inorder(root);
                printf("\n");
                break;
            case 3:
                preorder(root);
                printf("\n");
                break;
            case 4:
                postorder(root);
                printf("\n");
                break;
            case 5:
                scanf("%d", &x);
                if (search(root, x)) printf("%d FOUND\n", x);
                else printf("%d NOT FOUND\n", x);
                break;
            case 6:
                scanf("%d", &x);
                res = delete_BST(&root, x);
                if (res == -9999) printf("%d NOT FOUND\n", x);
                else printf("%d DELETED\n", res);
                break;
            case 9:
                exit(0);
        }
    }
    return 0;
}
```