1. Write a C program to remove duplicate element from sorted Linked List.

Input:

2 -> 3 -> 3 -> 4

Output:

2 -> 3 -> 4

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

// Node structure for a linked list
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to remove duplicate elements from a sorted linked list
void removeDuplicates(struct Node* head) {
    struct Node* current = head;

    // Traverse the list
    while (current != NULL && current->next != NULL) {
        // Check if the next node has the same data
        if (current->data == current->next->data) {
            // Duplicate found, remove the next node
            struct Node* temp = current->next;
            current->next = current->next->next;
            free(temp);
        } else {
            // Move to the next node
            current = current->next;
        }
    }
}

// Function to print the linked list
void printList(struct Node* head) {
    struct Node* current = head;
```

```c
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
// Main function
int main() {
    // Sample input: 2 -> 3 -> 3 -> 4
    struct Node* head = createNode(2);
    head->next = createNode(3);
    head->next->next = createNode(3);
    head->next->next->next = createNode(4);

    printf("Input: ");
    printList(head);

    // Remove duplicates
    removeDuplicates(head);

    printf("Output: ");
    printList(head);

    // Free the memory
    while (head != NULL) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
    }
    return 0;
}
```

2.Write a C program to rotate a doubly linked list by N nodes.

Input: (When N=2)

a b c d e

Output:

c d e a b

Input: (When N=4)

a b c d e f g h

Output:

e f g h a b c d

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

struct Node
{ char data;
  struct Node*prev;
  struct Node*next;
};

struct Node* createNode(char d)
{ struct Node* newNode= (struct Node*)malloc(sizeof(struct Node));
  newNode->data=d;
  newNode->prev=NULL;
  newNode->next=NULL;
  return newNode;}

void printList(struct Node* head)
{ while(head!=NULL)
    { printf("%c",head->data);
      head=head->next;}
    printf("\n");}

struct Node* rotateByN(struct Node* head,int N)
{ if(head==NULL||N==0){return head;}
  struct Node* current = head;
  int count =1;

  //Fetch to the Nth Node
  while(count<N&&current!=NULL)
  { current=current->next; count++;}
  if(current == NULL){//If N is greater than list length, do nothing
  return head;}
  //Save Nth Nodse as head
  struct Node* newHead=current->next;

  //updating the links
  current->next=NULL;
  current->prev=NULL;

  //Traverse to list end
  while(current->next!=NULL)
  {current->next=head;
  head->prev=current;
  return newHead;}}

int main()
{ struct Node* head = createNode('a');
    head->next = createNode('b');
```

```
    head->next->prev = head;
    head->next->next = createNode('c');
    head->next->next->prev = head->next;
    head->next->next->next = createNode('d');
    head->next->next->next->prev = head->next->next;
    head->next->next->next->next = createNode('e');
    head->next->next->next->next->prev = head->next->next->next;

    // Print the original list
    printf("Original List: ");
    printList(head);

    // Rotate the list by N nodes (N=2)
    head = rotateByN(head, 2);

    // Print the rotated list
    printf("Rotated List (N=2): ");

    printList(head);

    // Rotate the list by N nodes (N=4)
    head = rotateByN(head, 4);

    // Print the rotated list
    printf("Rotated List (N=4): ");

    printList(head);

// Free allocated memory
    while (head != NULL) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
    }

    return 0;}
```

3. Write a C program to sort the elements of a queue in ascending order.

Input

4 2 7 5 1

Output

1 2 4 5 7

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

```c
// Node structure for the queue
struct Node {
    int data;
    struct Node* next;
};

// Structure for the queue
struct Queue {
    struct Node* front;
    struct Node* rear;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

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

// Function to check if the queue is empty
int isEmpty(struct Queue* queue) {
    return queue->front == NULL;
}

// Function to enqueue a node
void enqueue(struct Queue* queue, int data) {
    struct Node* newNode = createNode(data);
    if (isEmpty(queue)) {
        queue->front = queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}

// Function to dequeue a node
int dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        return -1; // Return a sentinel value for an empty queue
```

```
    }

    struct Node* temp = queue->front;
    int data = temp->data;

    queue->front = temp->next;
    free(temp);

    if (queue->front == NULL) {
        queue->rear = NULL; // Update rear if the last element is dequeued
    }

    return data;
}

// Function to sort the elements of the queue in ascending order
void sortQueue(struct Queue* queue) {
    int temp;
    struct Node* current;

    while (!isEmpty(queue)) {
        temp = dequeue(queue);

        // Find the correct position to insert the current element
        for (current = queue->front; current != NULL && current->data < temp;
current = current->next);

        // Insert the element at the correct position
        if (current == NULL) {
            enqueue(queue, temp); // Insert at the end
        } else {
            struct Node* newNode = createNode(temp);
            newNode->next = current->next;
            current->next = newNode;
        }
    }
}

// Function to print the elements of the queue
void printQueue(struct Queue* queue) {
    struct Node* current = queue->front;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}
```

```c
int main() {
    // Create a queue and enqueue elements
    struct Queue* queue = createQueue();
    enqueue(queue, 4);
    enqueue(queue, 2);
    enqueue(queue, 7);
    enqueue(queue, 5);
    enqueue(queue, 1);

    // Print the original queue
    printf("Original Queue: ");
    printQueue(queue);

    // Sort the elements of the queue in ascending order
    sortQueue(queue);

    // Print the sorted queue
    printf("Sorted Queue: ");
    printQueue(queue);

    // Free allocated memory
    while (!isEmpty(queue)) {
        dequeue(queue);
    }
    free(queue);

    return 0;
}
```

4. List all queue function operations available for manipulation of data elements in c

In C, the basic operations that can be performed on a queue data structure include:

1. **enqueue**: This operation adds an element to the rear (end) of the queue.

   void enqueue(struct Queue* queue, int data);

2. **dequeue**: This operation removes and returns the element from the front of the queue.

   int dequeue(struct Queue* queue);

3. **isEmpty**: This operation checks if the queue is empty.

   int isEmpty(struct Queue* queue);

4. **isFull**: This operation checks if the queue is full (useful in implementations with a fixed-size array).

```
int isFull(struct Queue* queue);
```

5. **front**: This operation returns the element at the front of the queue without removing it.

```
int front(struct Queue* queue);
```

6. **rear**: This operation returns the element at the rear of the queue without removing it.

```
int rear(struct Queue* queue);
```

7. **size**: This operation returns the current number of elements in the queue.

```
int size(struct Queue* queue);
```

5. Reverse the given string using stack

Input: (string)

"LetsLearn"

Output: (string)

"nraeLsteL"

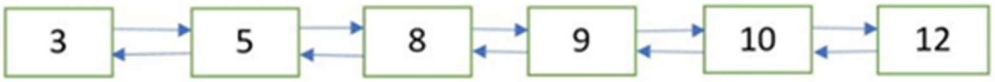
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure for the stack
struct Stack {
    char *arr;
    int top;
    unsigned capacity;
};

// Function to create a new stack
struct Stack* createStack(unsigned capacity) {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->arr = (char*)malloc(stack->capacity * sizeof(char));
    return stack;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
```

```c
    return stack->top == -1;
}

// Function to push a character onto the stack
void push(struct Stack* stack, char ch) {
    stack->arr[++stack->top] = ch;
}

// Function to pop a character from the stack
char pop(struct Stack* stack) {
    return stack->arr[stack->top--];
}

// Function to reverse a string using a stack
void reverseString(char* str) {
    int length = strlen(str);

    // Create a stack with capacity equal to the length of the string
    struct Stack* stack = createStack(length);

    // Push each character onto the stack
    for (int i = 0; i < length; i++) {
        push(stack, str[i]);
    }

    // Pop each character from the stack to reverse the string
    for (int i = 0; i < length; i++) {
        str[i] = pop(stack);
    }

    // Free allocated memory for the stack
    free(stack->arr);
    free(stack);
}

int main() {
    char input[] = "LetsLearn";

    printf("Original String: %s\n", input);

    // Reverse the string using a stack
    reverseString(input);

    printf("Reversed String: %s\n", input);

    return 0;
}
```

6. Insert value in sorted way in a sorted doubly linked list. Given a sorted doubly linked list and a value to insert, write a function to insert the value in sorted way.

Initial doubly linked list

Doubly Linked List after insertion of 9




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

// Node structure for doubly linked list
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a value in a sorted way into a doubly linked list
struct Node* insertSorted(struct Node* head, int value) {
    struct Node* newNode = createNode(value);

    // If the list is empty or the value is smaller than the head, insert at
the beginning
    if (head == NULL || value < head->data) {
        newNode->next = head;
        if (head != NULL) {
            head->prev = newNode;
        }
        return newNode;
    }
```

```c
    // Traverse the list to find the correct position for the new value
    struct Node* current = head;
    while (current->next != NULL && current->next->data < value) {
        current = current->next;
    }

    // Insert the new node at the correct position
    newNode->next = current->next;
    newNode->prev = current;
    if (current->next != NULL) {
        current->next->prev = newNode;
    }
    current->next = newNode;

    return head;
}

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

// Function to free the allocated memory for the doubly linked list
void freeList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        struct Node* temp = current;
        current = current->next;
        free(temp);
    }
}

int main() {
    // Create a sorted doubly linked list
    struct Node* head = createNode(3);
    head->next = createNode(5);
    head->next->prev = head;
    head->next->next = createNode(8);
    head->next->next->prev = head->next;
    head->next->next->next = createNode(10);
    head->next->next->next->prev = head->next->next;
    head->next->next->next->next = createNode(12);
    head->next->next->next->next->prev = head->next->next->next;
```

```
    printf("Original List: ");
    printList(head);

    // Insert a value in a sorted way (e.g., insert 9)
    head = insertSorted(head, 9);

    printf("List after inserting 9: ");
    printList(head);

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

    return 0;
}
```

7. Write a C program to insert/delete and count the number of elements in a queue.

Expected Output:

Initialize a queue!
Check the queue is empty or not? Yes
Number of elements in queue: 0
Insert some elements into the queue:
Queue elements are: 1 2 3
Number of elements in queue: 3
Delete two elements from the said queue:
Queue elements are: 3
Number of elements in queue: 1
Insert another element into the queue:
Queue elements are: 3 4
Number of elements in the queue: 2

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

// Node structure for the queue
struct Node {
    int data;
    struct Node* next;
};

// Structure for the queue
struct Queue {
    struct Node* front;
```

```c
    struct Node* rear;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

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

// Function to check if the queue is empty
int isEmpty(struct Queue* queue) {
    return queue->front == NULL;
}

// Function to enqueue a node
void enqueue(struct Queue* queue, int data) {
    struct Node* newNode = createNode(data);
    if (isEmpty(queue)) {
        queue->front = queue->rear = newNode;
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
    }
}

// Function to dequeue a node
int dequeue(struct Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        return -1; // Return a sentinel value for an empty queue
    }

    struct Node* temp = queue->front;
    int data = temp->data;

    queue->front = temp->next;
    free(temp);

    if (queue->front == NULL) {
```

```c
        queue->rear = NULL; // Update rear if the last element is dequeued
    }

    return data;
}

// Function to count the number of elements in the queue
int countElements(struct Queue* queue) {
    int count = 0;
    struct Node* current = queue->front;

    while (current != NULL) {
        count++;
        current = current->next;
    }

    return count;
}

// Function to print the elements of the queue
void printQueue(struct Queue* queue) {
    struct Node* current = queue->front;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    // Create an empty queue
    struct Queue* queue = createQueue();

    // Check if the queue is empty
    printf("Check the queue is empty or not? %s\n", isEmpty(queue) ? "Yes" :
"No");

    // Count the number of elements in the queue
    printf("Number of elements in queue: %d\n", countElements(queue));

    // Insert elements into the queue
    printf("Insert some elements into the queue:\n");
    enqueue(queue, 1);
    enqueue(queue, 2);
    enqueue(queue, 3);
    printf("Queue elements are: ");
    printQueue(queue);
    printf("Number of elements in queue: %d\n", countElements(queue));
```

```
    // Delete two elements from the queue
    printf("Delete two elements from the said queue:\n");
    dequeue(queue);
    dequeue(queue);
    printf("Queue elements are: ");
    printQueue(queue);
    printf("Number of elements in queue: %d\n", countElements(queue));

    // Insert another element into the queue
    printf("Insert another element into the queue:\n");
    enqueue(queue, 4);
    printf("Queue elements are: ");
    printQueue(queue);
    printf("Number of elements in the queue: %d\n", countElements(queue));

    // Free allocated memory for the queue
    while (!isEmpty(queue)) {
        dequeue(queue);
    }
    free(queue);

    return 0;
}
```

8. Write a C program to Find whether an array is a subset of another array.

Input:

arr1[] = {11, 1, 13, 21, 3, 7}, arr2[] = {11, 3, 7, 1}

Output:

arr2[] is a subset of arr1[]

Input:

arr1[] = {10, 5, 2, 23, 19}, arr2[] = {19, 5, 3}

Output:

arr2[] is not a subset of arr1[]

```
#include <stdio.h>

// Function to check if arr2[] is a subset of arr1[]
int isSubset(int arr1[], int m, int arr2[], int n) {
    int i, j;

    // Traverse all elements of arr2[]
    for (i = 0; i < n; i++) {
```

```
        // Check if the current element of arr2[] is present in arr1[]
        for (j = 0; j < m; j++) {
            if (arr2[i] == arr1[j])
                break;
        }

        // If the element was not found in arr1[], return false
        if (j == m)
            return 0;
    }

    // If all elements of arr2[] were found in arr1[], return true
    return 1;
}

int main() {
    int arr1[] = {11, 1, 13, 21, 3, 7};
    int arr2[] = {11, 3, 7, 1};
    int m = sizeof(arr1) / sizeof(arr1[0]);
    int n = sizeof(arr2) / sizeof(arr2[0]);

    // Check if arr2[] is a subset of arr1[]
    if (isSubset(arr1, m, arr2, n))
        printf("arr2[] is a subset of arr1[]\n");
    else
        printf("arr2[] is not a subset of arr1[]\n");

    int arr3[] = {10, 5, 2, 23, 19};
    int arr4[] = {19, 5, 3};
    int m2 = sizeof(arr3) / sizeof(arr3[0]);
    int n2 = sizeof(arr4) / sizeof(arr4[0]);

    // Check if arr4[] is a subset of arr3[]
    if (isSubset(arr3, m2, arr4, n2))
        printf("arr4[] is a subset of arr3[]\n");
    else
        printf("arr4[] is not a subset of arr3[]\n");

    return 0;
}
```