Algorithm to create Doubly Linked list

```
Begin:
    alloc (head)
    If (head == NULL) then
        write ('Unable to allocate memory')
    End if
    Else then
        read (data)
        head.data ← data;
        head.prev ← NULL;
        head.next ← NULL;
        last ← head;
        write ('List created successfully')
    End else
End
```

Algorithm to traverse Doubly Linked list from beginning

Input : head {Pointer to the first node of the list}

```
Begin:
    If (head == NULL) then
        write ('List is empty')
    End if
    Else then
        temp ← head;
        While (temp != NULL) do
            write ('Data = ', temp.data)
            temp ← temp.next;
        End while
    End else
End
```

Algorithm to traverse Doubly Linked list from end

Input : last {Pointer to the last node of the list}

```
Begin:
    If (last == NULL) then
        write ('List is empty')
    End if
    Else then
        temp ← last;
        While (temp != NULL) do
            write ('Data = ', temp.data)
            temp ← temp.prev;
        End while
    End else
End
```

Program to create and display Doubly linked list

```
#include <stdio.h>

#include <stdlib.h>

struct node {

    int data;

    struct node * prev;

    struct node * next;

}*head, *last;

void createList(int n);

void displayListFromFirst();

void displayListFromEnd();

int main()

{

    int n, choice;

    head = NULL;

    last = NULL;

    printf("Enter the number of nodes you want to create: ");

    scanf("%d", &n);


    createList(n);


    printf("\nPress 1 to display list from First");

    printf("\nPress 2 to display list from End : ");

    scanf("%d", &choice);

    if(choice==1)
    {
        displayListFromFirst();
    }
    else if(choice == 2)
    {
        displayListFromEnd();
    }
    return 0;
}
```

void createList(int n)

```
{
    int i, data;
    struct node *newNode;

    if(n >= 1)
    {
        head = (struct node *)malloc(sizeof(struct node));

        if(head != NULL)
        {
            printf("Enter data of 1 node: ");
            scanf("%d", &data);
            head->data = data;
            head->prev = NULL;
            head->next = NULL;
            last = head;
            for(i=2; i<=n; i++)
            {
                newNode = (struct node *)malloc(sizeof(struct node));
                if(newNode != NULL)
                {
                    printf("Enter data of %d node: ", i);
                    scanf("%d", &data);
                    newNode->data = data;
                    newNode->prev = last;
                    newNode->next = NULL;
                    last->next = newNode;
                    last = newNode;
                }
                else
                {
                    printf("Unable to allocate memory.");
                    break;
                }
            }
            printf("\nDOUBLY LINKED LIST CREATED SUCCESSFULLY\n");
        }
        else
        {
            printf("Unable to allocate memory");
        }
    }
}
```

```
void displayListFromFirst()
{
    struct node * temp;
    int n = 1;
    if(head == NULL)
    {
        printf("List is empty.");
    }
    else
    {
        temp = head;
        printf("\n\nDATA IN THE LIST:\n");
        while(temp != NULL)
        {
            printf("DATA of %d node = %d\n", n, temp->data);
            n++;
            temp = temp->next;
        }
    }
}
void displayListFromEnd()
{
    struct node * temp;
    int n = 0;
    if(last == NULL)
    {
        printf("List is empty.");
    }
    else
    {
        temp = last;
        printf("\n\nDATA IN THE LIST:\n");
        while(temp != NULL)
        {
            printf("DATA of last-%d node = %d\n", n, temp->data);
            n++;
            temp = temp->prev;
        }
    }
}
```

Algorithm to insert a node at the beginning of a Doubly linked list

Input : head {A pointer pointing to the first node of the list}

```
Begin:
    alloc (newNode)
    If (newNode == NULL) then
        write ('Unable to allocate memory')
    End if
    Else then
        read (data)
        newNode.data ← data;
        newNode.prev ← NULL;
        newNode.next ← head;

        head.prev ← newNode;
        head ← newNode;
        write('Node added successfully at the beginning of List')
    End else
End
```

Algorithm to insert a node at the end of Doubly linked list

Input : last {Pointer to the last node of doubly linked list}

```
Begin:
    alloc (newNode)
    If (newNode == NULL) then
        write ('Unable to allocate memory')
    End if
    Else then
        read (data)
        newNode.data ← data;
        newNode.next ← NULL;
        newNode.prev ← last;

        last.next ← newNode;
        last ← newNode;
        write ('Node added successfully at the end of List')
    End else
End
```

Algorithm to insert node at any position of doubly linked list

Input : head {Pointer to the first node of doubly linked list}

: last {Pointer to the last node of doubly linked list}

: N {Position where node is to be inserted}

```
Begin:
    temp ← head
    For i←1 to N-1 do
        If (temp == NULL) then
            break
        End if
        temp ← temp.next;
    End for
    If (N == 1) then
        insertAtBeginning()
    End if
    Else If (temp == last) then
        insertAtEnd()
    End if
    Else If (temp != NULL) then
        alloc (newNode)
        read (data)

        newNode.data ← data;
        newNode.next ← temp.next
        newNode.prev ← temp
        If (temp.next != NULL) then
            temp.next.prev ← newNode;
        End if
        temp.next ← newNode;
        write('Node added successfully')
    End if
```

End

C program to insert a node in Doubly linked list

```
#include <stdio.h>

#include <stdlib.h>

struct node {

    int data;

    struct node * prev;

    struct node * next;

}*head, *last;


void createList(int n);

void displayList();

void insertAtBeginning(int data);

void insertAtEnd(int data);

void insertAtN(int data, int position);

int main()

{

    int n, data, choice=1;


    head = NULL;

    last = NULL;

    while(choice != 0)

    {

        printf("============================================\n");

        printf("DOUBLY LINKED LIST PROGRAM\n");

        printf("============================================\n");

        printf("1. Create List\n");
printf("2. Insert node - at beginning\n");
printf("3. Insert node - at end\n");
printf("4. Insert node - at N\n");
printf("5. Display list\n");
printf("0. Exit\n");
printf("--------------------------------------------\n");
printf("Enter your choice : ");
scanf("%d", &choice);
switch(choice)
{
   case 1:
      printf("Enter the total number of nodes in list: ");
      scanf("%d", &n);


      createList(n);
      break;
   case 2:
      printf("Enter data of first node : ");
      scanf("%d", &data);


      insertAtBeginning(data);
      break;
   case 3:
      printf("Enter data of last node : ");
      scanf("%d", &data);
      insertAtEnd(data);
      break;
```

```
            case 4:
                printf("Enter the position where you want to insert new node: ");
                scanf("%d", &n);
                printf("Enter data of %d node : ", n);
                scanf("%d", &data);
                insertAtN(data, n);
                break;
            case 5:
                displayList();
                break;
            case 0:
                break;
            default:
                printf("Error! Invalid choice. Please choose between 0-5");
        }
        printf("\n\n\n\n\n");
    }
    return 0;
}

void createList(int n)
{
    int i, data;
    struct node *newNode;
    if(n >= 1)
    {
        head = (struct node *)malloc(sizeof(struct node));
        printf("Enter data of 1 node: ");
        scanf("%d", &data);
        head->data = data;
        head->prev = NULL;
        head->next = NULL;
        last = head;
        for(i=2; i<=n; i++)
        {
            newNode = (struct node *)malloc(sizeof(struct node));

            printf("Enter data of %d node: ", i);
            scanf("%d", &data);
            newNode->data = data;
            newNode->prev = last;
            newNode->next = NULL;
            last->next = newNode;
            last = newNode;
        }
        printf("\nDOUBLY LINKED LIST CREATED SUCCESSFULLY\n");
    }
}
void displayList()
{
    struct node * temp;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        temp = head;
        printf("DATA IN THE LIST:\n");
        while(temp != NULL)
        {
            printf("DATA of %d node = %d\n", n, temp->data);

            n++;
            temp = temp->next;
        }
    }
}
void insertAtBeginning(int data)
{
    struct node * newNode;

    if(head == NULL)
    {
        printf("Error, List is Empty!\n");
    }
    else
    {
        newNode = (struct node *)malloc(sizeof(struct node));
        newNode->data = data;
        newNode->next = head;
        newNode->prev = NULL;
        head->prev = newNode;
        head = newNode;
        printf("NODE INSERTED SUCCESSFULLY AT THE BEGINNING OF THE LIST\n");
    }
}
void insertAtEnd(int data)
{
    struct node * newNode;
    if(last == NULL)
    {
        printf("Error, List is empty!\n");
    }
    else
    {
        newNode = (struct node *)malloc(sizeof(struct node));
        newNode->data = data;
        newNode->next = NULL;
        newNode->prev = last;
        last->next = newNode;
        last = newNode;
        printf("NODE INSERTED SUCCESSFULLY AT THE END OF LIST\n");
    }
}
void insertAtN(int data, int position)
```

```
{
    int i;
    struct node * newNode, *temp;
    if(head == NULL)
    {
        printf("Error, List is empty!\n");
    }
    else
    {
        temp = head;
        i=1;

        while(i<position-1 && temp!=NULL)
        {
            temp = temp->next;
            i++;
        }
        if(position == 1)
        {
            insertAtBeginning(data);
        }
        else if(temp == last)
        {
            insertAtEnd(data);
        }
        else if(temp!=NULL)
        {
            newNode = (struct node *)malloc(sizeof(struct node));
            newNode->data = data;
            newNode->next = temp->next;
            newNode->prev = temp;
            if(temp->next != NULL)
            {
                temp->next->prev = newNode;
            }
             temp->next = newNode;

            printf("NODE INSERTED SUCCESSFULLY AT %d POSITION\n", position);
        }
        else
        {
            printf("Error, Invalid position\n");
        }
    }
}
```

Algorithm to delete node from beginning

Input: head {Pointer to first node of the linked list}

```
Begin:
    If (head == NULL) then
        write ('Can't delete from an empty list')
    End if
    Else then
        toDelete ← head;
        head ← head.next;
        head.prev ← NULL;
        unalloc (toDelete)
        write ('Successfully deleted first node from the list')
    End if
End
```

Algorithm to delete node from end

Input: last {Pointer to last node of the linked list}

```
Begin:
    If (last == NULL) then
        write ('Can't delete from an empty list')
    End if
    Else then
        toDelete ← last;
        last ← last.prev;
        last.next ← NULL;
        unalloc (toDelete)
        write ('Successfully deleted last node from the list')
    End if
End
```

Algorithm to delete node from any position

Input : head {Pointer to the first node of the list}

last {Pointer to the last node of the list}

N {Position to be deleted from list}

Begin:

```
    current ← head;
    For i ← 1 to N and current != NULL do
        current ← current.next;
    End for
    If (N == 1) then
        deleteFromBeginning()
    End if
    Else if (current == last) then
        deleteFromEnd()
    End if
    Else if (current != NULL) then
        current.prev.next ← current.next
        If (current.next != NULL) then
            current.next.prev ← current.prev;
        End if
        unalloc (current)
        write ('Node deleted successfully from ', N, ' position')
    End if
    Else then
        write ('Invalid position')
    End if
End
```

C program to delete a node from Doubly linked list

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

```
struct node {
    int data;
    struct node * prev;
    struct node * next;
}*head, *last;



void createList(int n);
void displayList();
void deleteFromBeginning();
void deleteFromEnd();
void deleteFromN(int position);
int main()
{
    int n, data, choice=1;
    head = NULL;
    last = NULL;
    while(choice != 0)
    {
        printf("============================================\n");
        printf("DOUBLY LINKED LIST PROGRAM\n");
        printf("============================================\n");
        printf("1. Create List\n");
        printf("2. Delete node - from beginning\n");
        printf("3. Delete node - from end\n");
        printf("4. Delete node - from N\n");
        printf("5. Display list\n");
        printf("0. Exit\n");
        printf("--------------------------------------------\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter the total number of nodes in list: ");
                scanf("%d", &n);
                createList(n);
                break;
            case 2:
                deleteFromBeginning();
                break;
            case 3:
                deleteFromEnd();
                break;
            case 4:
                printf("Enter the node position which you want to delete: ");
                scanf("%d", &n);
                deleteFromN(n);
                break;
            case 5:
                displayList();
                break;
            case 0:
                break;
```

```
            default:
                printf("Error! Invalid choice. Please choose between 0-5");
        }

        printf("\n\n\n\n\n");
    }
    return 0;
}

void createList(int n)
{
    int i, data;
    struct node *newNode;
    if(n >= 1)
    {
        head = (struct node *)malloc(sizeof(struct node));

        printf("Enter data of 1 node: ");
        scanf("%d", &data);
        head->data = data;
        head->prev = NULL;
        head->next = NULL;
        last = head;
        for(i=2; i<=n; i++)
        {
            newNode = (struct node *)malloc(sizeof(struct node));
            printf("Enter data of %d node: ", i);
            scanf("%d", &data);

            newNode->data = data;
            newNode->prev = last; // Link new node with the previous node
            newNode->next = NULL;
            last->next = newNode; // Link previous node with the new node
            last = newNode; // Make new node as last node
        }
        printf("\nDOUBLY LINKED LIST CREATED SUCCESSFULLY\n");
    }
}
void displayList()
{
    struct node * temp;
    int n = 1;
    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        temp = head;
        printf("DATA IN THE LIST:\n");
        while(temp != NULL)
        {
            printf("DATA of %d node = %d\n", n, temp->data);
            n++;
            temp = temp->next;
        }
    }
}

void deleteFromBeginning()
{
    struct node * toDelete;
    if(head == NULL)
    {
        printf("Unable to delete. List is empty.\n");
    }
    else
    {
        toDelete = head;
        head = head->next;

        if (head != NULL)
            head->prev = NULL;

        free(toDelete);
        printf("SUCCESSFULLY DELETED NODE FROM BEGINNING OF THE LIST.\n");
    }
}

void deleteFromEnd()
{
    struct node * toDelete;
    if(last == NULL)
    {
        printf("Unable to delete. List is empty.\n");
    }
    else
    {
        toDelete = last;
        last = last->prev;
        if (last != NULL)
            last->next = NULL;

        free(toDelete);       // Delete the last node
        printf("SUCCESSFULLY DELETED NODE FROM END OF THE LIST.\n");
    }
}
void deleteFromN(int position)
{
    struct node *current;
    int i;

    current = head;
    for(i=1; i<position && current!=NULL; i++)
    {
        current = current->next;
    }
    if(position == 1)
    {
        deleteFromBeginning();
    }
    else if(current == last)
    {
        deleteFromEnd();
    }
    else if(current != NULL)
    {
        current->prev->next = current->next;
        current->next->prev = current->prev;

        free(current); // Delete the n node

        printf("SUCCESSFULLY DELETED NODE FROM %d POSITION.\n", position);
    }
    else
    {
        printf("Invalid position!\n");
    }
}
```

Stack

Stack is a **LIFO (Last In First Out)** data structure. It allows us to insert and remove an element in special order. Stack allows element addition and removal from the top of stack.

Operations performed on Stack

Basic operations performed on stack.

- Push
- Pop

Stack node structure.

```
// Stack node structure

struct stack

{

    int data;

    struct stack *next;

} *top;



int size = 0;
```

Push elements in stack using linked list

- Insertion of new element to stack is known as *push operation* in stack.
- push elements at top of stack.

Step by step descriptive logic to push elements in stack.

a. Check stack overflow,
    if (size >= CAPACITY), then print "Stack overflow" error message.
    Otherwise move to below step.

b. Create a new stack node using dynamic memory allocation
    struct stack * newNode = (struct stack *) malloc(sizeof(struct stack));.

c. Assign data to the newly created node using

    newNode->data = data;

d. Link new node with the current stack top most element.
    newNode->next = top; and
    Increment size count by 1.

e. Finally make sure the top of stack should always be the new node i.e.
    top = new Node;.

Pop

Removal of top most element from stack is known as *pop operation* in stack.

Step by step descriptive logic to pop elements from stack.

a. If size <= 0
   then throw "Stack is Empty" error,
   otherwise move to below step.
b. Assign the top most element reference to some temporary variable,
   say struct stack *topNode = top;.
c. Similarly copy data of stack top element to some variable
   int data = top->data;.
d. Make second element of stack as top element i.e.
   top = top->next;.
e. Delete the top most element from memory using
   free(topNode);.
f. Decrement stack size by one and return data.

Stack program in C

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>     // For INT_MIN

#define CAPACITY 10000  // Stack maximum capacity

// Define stack node structure

struct stack

{

   int data;

   struct stack *next;

} *top;
```

```
// Stack size
int size = 0;
void push(int element);
int  pop();
int main()
{
    int choice, data;

    while(1)
    {
        /* Menu */
        printf("------------------------------------\n");
        printf("    STACK IMPLEMENTATION PROGRAM    \n");
        printf("------------------------------------\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Size\n");
        printf("4. Exit\n");
        printf("------------------------------------\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter data to push into stack: ");
                scanf("%d", &data);

                push(data);

                break;


            case 2:

                data = pop();

                if (data != INT_MIN)

                    printf("Data => %d\n", data);

                break;


            case 3:

                printf("Stack size: %d\n", size);

                break;


            case 4:

                printf("Exiting from app.\n");

                exit(0);

                break;

            default:

                printf("Invalid choice, please try again.\n");

        }

        printf("\n\n");

    }   return 0;

}

void push(int element)
```

```
{
    if (size >= CAPACITY)
    {
        printf("Stack Overflow, can't add more element to stack.\n");
        return;
    }
    struct stack * newNode = (struct stack *) malloc(sizeof(struct stack));
    newNode->data = element;
    newNode->next = top;

        top = newNode;
        size++;
    printf("Data pushed to stack.\n");
}
int pop()
{
    int data = 0;
    struct stack * topNode;
    if (size <= 0 || !top)
    {
        printf("Stack is empty.\n");
        return INT_MIN;
    }
    topNode = top;
        data = top->data;
```

```
   top = top->next;

      free(topNode);

   size--;

   return data;

}
```

Queue:

Queue is a linear data structure where elements are ordered in special fashion i.e. **FIFO** (**First In First Out**) that is the element inserted first to the queue will be removed first from the queue.

Examples: queue of persons at ticket counter.

Operations performed on Queue

queue two basic operations.

1. Enqueue (Insertion)
2. Dequeue (Removal)

Queue structure

```
typedef struct node

{

   int data;

   struct node * next;

} Queue;
```

Note: typedef is used to create an alias for our new type. In this program Queue is used instead of struct node.

```
Unsigned int size = 0;    // Size of queue

Queue *rear, *front;      // Reference of rear and front node in queue
```

Enqueue an element in Queue using linked list

- Insertion of new element in queue is known as *enqueue*.
- Enqueue a new element at rear of the queue, if its capacity is not full.

Step by step descriptive logic to enqueue an element in queue.

a. If queue size is more than capacity, then throw out of capacity error. Otherwise continue to next step.

b. Allocate memory for node of Queue type using malloc().

```
Queue *newNode = (Queue *) malloc (sizeof(Queue));
```

c. Make sure that the newly created node points to nothing i.e.

```
newNode->next = NULL;
```

d. Assign data to the new node, may be user input.

e. If queue is not empty then link rear node to newNode.

```
(*rear)->next = newNode;.
```

f. Make newNode as rear. Since after each enqueue rear gets changed.

g. If its first node in queue then make it as front node

```
*front = *rear;.
```

h. Finally after each successful enqueue, increment size++ by one.

Dequeue an element from Queue using linked list

- Removal of an existing element from queue is known as *dequeue*.
- dequeue from front of the queue, if its not empty.

Step by step descriptive logic to dequeue element from queue using linked list.

a.If queue is empty, then throw empty queue error. Otherwise continue to next step.

b.Get front element from queue, which is our required element to dequeue. Store it in some variable

a. Queue *toDequeue = *front;.
b. Also store its data to some variable as
   int data = toDequeue->data;
c. Move front node ahead *front = (*front)->next;.
d. Decrement size--; by one.
e. Free the dequeued element from memory to save resources,
   free(toDequeue);.
f. Return data, which is our required dequeued element.

Queue implementation using linked list in C.

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

#define CAPACITY 100

typedef struct node
{
   int data;
   struct node * next;
} Queue;

unsigned int size = 0;

int enqueue(Queue ** rear, Queue ** front, int data);
int dequeue(Queue ** front);
int getRear(Queue * rear);
int getFront(Queue * front);
int isEmpty();
int isFull();


int main()
{
   int ch, data;
   Queue *rear, *front;

   rear  = NULL;
   front = NULL;

   while (1)
   {
      printf("--------------------------------------------\n");
      printf("  QUEUE LINKED LIST IMPLEMENTATION PROGRAM  \n");
      printf("--------------------------------------------\n");
      printf("1. Enqueue\n");
      printf("2. Dequeue\n");
      printf("3. Size\n");
      printf("4. Get Rear\n");
      printf("5. Get Front\n");
      printf("0. Exit\n");
      printf("--------------------------------------------\n");
      printf("Select an option: ");
```

```
scanf("%d", &ch);

switch (ch)
{
    case 1:
        printf("\nEnter data to enqueue: ");
        scanf("%d", &data);

        if (enqueue(&rear, &front, data))
            printf("Element added to queue.");
        else
            printf("Queue is full.");

        break;

    case 2:
        data = dequeue(&front);


        if (data == INT_MIN)
            printf("Queue is empty.");
        else
            printf("Data => %d", data);

        break;

    case 3:

        if (isEmpty())
            printf("Queue is empty.");
        else
            printf("Queue size => %d", size);

        break;

    case 4:
        data = getRear(rear);

        if (data == INT_MIN)
            printf("Queue is empty.");
        else
            printf("Rear => %d", data);

        break;

    case 5:
```

```
                data = getFront(front);

                if (data == INT_MIN)
                    printf("Queue is empty.");
                else
                    printf("Front => %d", data);

                break;

            case 0:
                printf("Exiting from app.\n");
                exit(0);

            default:
                printf("Invalid choice, please input number between (0-5).");
                break;
        }

        printf("\n\n");
    }
}


int enqueue(Queue ** rear, Queue ** front, int data)
{
    Queue * newNode = NULL;


    if (isFull())
    {
        return 0;
    }

    newNode = (Queue *) malloc (sizeof(Queue));

    newNode->data = data;

    newNode->next = NULL;


    if ( (*rear) )
    {
        (*rear)->next = newNode;
    }
    *rear = newNode;

    if ( !( *front) )
    {
        *front = *rear;
    }

    size++;

    return 1;
}


int dequeue(Queue ** front)
{
    Queue *toDequque = NULL;
    int data = INT_MIN;

    if (isEmpty())
    {
        return INT_MIN;
    }

    toDequque = *front;
    data = toDequque->data;


    *front = (*front)->next;

    size--;

    free(toDequque);

    return data;
}


int getRear(Queue * rear)
{
    .
    return (isEmpty())    ? INT_MIN        : rear->data;
}



.
```

```
int getFront(Queue * front)
{

    return (isEmpty())     ? INT_MIN     : front->data;
}

int isEmpty()
{
    return (size <= 0);
}


int isFull()
{
    return (size > CAPACITY);
}
```