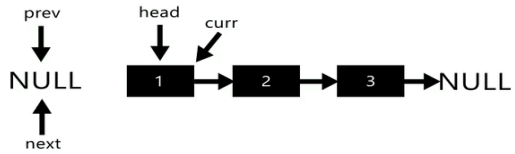
Linked List 👍

Reversal:- Initialize three pointers prev as NULL, curr as head, and next VInitialize three pointers prev as NULL, curr as head, and next as NULLas NULL.




```
while (current != NULL)
     {
          next  = current->next;
          current->next = prev;
          prev = current;
          current = next;
     }
     *head_ref = prev;
```

```cpp
#include <iostream>
using namespace std;

struct Node {
    int data;
    struct Node* next;
    //struct Node* prev;
};

struct Node* head = NULL;

// Insert
void insert(int new_data) {
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    if (head == NULL) {
        head = new_node;
    } else {
        struct Node* temp;
        temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    }
}

void delete_node() {
    struct Node* temp;
    temp = head;
    if (temp == NULL) {
```

```
        cout << "List is empty";
    } else {
        while (temp->next->next != NULL) {
            temp = temp->next;
            cout << temp->data;
        }
        temp->next = NULL;
        free(temp->next);
    }
}

struct Node* current;
struct Node* previous;
struct Node* nextEle;

void reversal() {
    current = head;
    previous = NULL;
    while (current != NULL) {
        nextEle = current->next;
        current->next = previous;
        previous = current;
        current = nextEle;
    }
    head = previous;
}

void display() {
    struct Node* ptr;
    ptr = head;
    while (ptr != NULL) {
        cout << ptr->data << " ";
        ptr = ptr->next;
    }
}

int main() {
    insert(1);
    insert(2);
    insert(5);
    display();

    reversal();
    display();
    delete_node();
    cout<<"After deletion elements are :"<<" ";
    display();
    return 0;
}
```

Doubly Linked List:

Memory leak-Dangling pointer

https://www.ritambhara.in/memory-leaks-and-dangling-pointers/#:~:text=int%20*%20ptr%20%3D%20new%20int%3B,NULL%20after%20deleting%20a%20pointer.&text=In%20the%20above%

Insertion :

```
#include <iostream>
using namespace std;
struct node
{
  struct node *prev;
  int data;
  struct node *next;
};

struct node *head = NULL;

void
InsertAtBegin (int element)
{
  cout << " Insertion started : " << " " << endl;
  struct node *ptr = (struct node *) malloc (sizeof (struct node));
  if (ptr == NULL)
    {
      cout << "Overflow" << " " << endl;
    }

  else
    {
      ptr->data = element;
      if (head == NULL)
        {

          ptr->next = NULL;
          ptr->prev = NULL;
          head = ptr;
        }
      else
        {
          ptr->next = head;
          head->prev = ptr;
          ptr->prev = NULL;
          head = ptr;

        }

    }

}

void
InsertAtEnd (int element)
{
```

```
  struct node *temp = (struct node *) malloc (sizeof (struct node));
  if (temp == NULL)
    {
      cout << "Overflow" << " " << endl;

    }
  else
    {

      if (head == NULL)
        {
          temp->next = NULL;
          temp->prev = NULL;
          temp = head;

        }
      else
        {
          temp->data = element;
          struct node *ptr = head;
          while (ptr->next != NULL)
            {
              ptr = ptr->next;
            }
          ptr->next = temp;
          temp->prev = ptr;
          temp->next = NULL;
        }

    }
}

void
InsertAtPosition (int element, int position)
{
  if (position < 1)
    {
      cout << "Invalid position" << endl;
      return;
    }

  struct node *newNode = (struct node *) malloc (sizeof (struct node));
  newNode->data = element;

  if (position == 1)
    {
      // Insert at the beginning
      newNode->prev = NULL;
      newNode->next = head;
      if (head != NULL)
        {
          head->prev = newNode;
        }
      head = newNode;
    }
  else
    {
      struct node *temp = head;
      int count = 1;
```

```
      // Traverse to the node at the specified position or the end of the list
      while (temp != NULL && count < position - 1)
        {
          temp = temp->next;
          count++;
        }

      if (temp == NULL)
        {
          cout << "Position out of range" << endl;
          free (newNode);
          return;
        }

      newNode->prev = temp;
      newNode->next = temp->next;

      if (temp->next != NULL)
        {
          temp->next->prev = newNode;
        }

      temp->next = newNode;
    }
}


void deletion(int position) {
    if (position < 1) {
        cout << "Invalid position" << endl;
        return;
    }

    if (position == 1) {
        // Delete the first node
        if (head == NULL) {
            cout << "List is already empty" << endl;
        } else {
            struct node* temp = head;
            head = head->next;
            if (head != NULL) {
                head->prev = NULL;
            }
            free(temp);
        }
    } else {
        struct node* current = head;
        int count = 1;

        // Traverse to the node at the specified position
        while (current != NULL && count < position) {
            current = current->next;
            count++;
        }

        if (current == NULL) {
            cout << "Position out of range" << endl;
        } else {
```

```
            // Adjust the pointers to remove the node
            current->prev->next = current->next;
            if (current->next != NULL) {
                current->next->prev = current->prev;
            }
            free(current);
        }
    }
}




void traversal ()
{
  cout << " Traversal Started: " << " " << endl;
  struct node *temp = head;
  temp = head;
  while (temp != NULL)
    {
      cout << temp->data << " ";
      temp = temp->next;

    }
}

int main ()
{

  InsertAtBegin (1);
  InsertAtBegin (2);
  InsertAtEnd (9);
  InsertAtPosition (1, 1);
  InsertAtPosition(91,3);
  InsertAtPosition(21,2);

  //deletion(1);
  traversal();
  deletion(3);

  traversal ();
  return 0;
}
```

Reverse a DLL:

```
void reversal(){
    struct node* temp = NULL;
    struct node* current = head;
    while(current!=NULL){
        temp = current->prev;
        current->prev = current->next;
        current->next = temp;
        current = current->prev;
    }

    if(temp!=NULL){
        head=temp->prev;
    }


}
```

CLL:
Xor LL:

Stacks:

Push()
Pop()
Display()

Size()
Peek()

Using Arrays

```
#include <iostream>
using namespace std;
int stack[100];
int top = -1;
int n;

void
push ()
{
  int val;
  if (top == n)
    {
      cout << "Stack is full" << " " << endl;

    }
  else
    {
      cout << "Enter the value: " << " " << endl;
      cin >> val;
      top = top + 1;
      stack[top] = val;

    }
}

int
peek ()
{
  int x = stack[top];
  return x;

}
```

```
void
pop ()
{
  if (top == -1)
    {
      cout << "Stack underflow" << " ";
    }
  else
    {
      top = top - 1;
    }


}


void
display ()
{

  for (int i = top; i >= 0; i--)
    {
      cout << stack[i] << " ";
    }
  if (top == -1)
    {
      cout << "Stack is empty" << " ";
    }



}


int
main ()
{

  cout << " Enter the number of total elements:" << " " << endl;
```

```
  cin >> n;
  for (int i = 0; i < n; i++)
    {
      push ();
    }



  display ();
  //pop();
  cout << " Peek element is :" << " ";
  //display();
  int p = peek ();
  cout << p;
  return 0;
}
```

Using Linked List:

Get min at POP
Difference between exit and return

```
#include <iostream>
using namespace std;

class Node
{
public:
  int data;
  Node *link;

    Node (int n)
  {
    this->data = n;
    this->link = NULL;
  }
};
```

```
class Stack
{
  Node *top;

public:
    Stack ()
  {
    top = NULL;
  }

  void push (int data)
  {
    Node *temp = new Node (data);
    if (!temp)
      {
      cout << "\nstack is full . ";
      }

    temp->data = data;
    temp->link = top;
    top = temp;


  }


  bool isempty ()
  {
    return top == NULL;

  }

  int peek ()
  {
    if (!isempty ())
      {
      return top->data;
      }
```

```
  else
    {
    exit (1);
    }

}

void Pop ()
{
  Node *temp;
  if (top == NULL)
    {
    cout << "\nstack underflow" << endl;

    }
  else
    {
    temp = top;
    top = top->link;
    free (temp);



    }



}




void display ()
{
  Node *temp;
  if (top == NULL)
    {
```

```
      cout << "Stack underflow";
      exit (1);
       }
     else
       {
      temp = top;
      while (temp != NULL)
        {
          cout << temp->data;
          temp = temp->link;
          if (temp != NULL)
            {
            cout << "->";
             }
        }


       }



  }


};

int
main ()
{
  Stack s;
  s.push (1);
  s.push (11);
  s.push (22);
  s.push (33);
  s.push (44);

  s.display ();
  cout << " Delete top" << endl;
  s.Pop ();
```

```
  s.display ();
  cout << "After peek:" << endl;
  int p = s.peek ();
  cout << p << endl;
  s.display ();


  return 0;
}
```

Queue :
Implementation using array

```
#include <iostream>
using namespace std;

//take two variables front and rear both of which will be
initialized to 0 which means the queue is currently empty.
//front
//rear
//Enqueue
//Dequeue
//front
//Display
int queue[100] , n=100 , front = -1 , rear = -1;
void Insert(){
    int val;
    if(rear==n-1){
        cout<<"Queue Overflow"<<endl;
```

```
    }
    else{
        if(front == -1){
            front=0;
        }
        cout<<"Insert the element in Queue : "<<endl;
        cin>>val;
        rear++;
        queue[rear]=val;

    }
}
void Delete(){
    if(front==-1){
        cout<<"Queue is empty"<<endl;
    }
    else{
        cout<<"deleted element is : "<<queue[front]<<endl;
        front++;
    }

}
void Display(){

    if(front ==-1){
        cout<<"Queue is empty "<<endl;

    }
    else{
        cout<<"Queue elememnts are :";
        for(int i=front;i<=rear;i++){
            cout<<queue[i]<<" ";
        }
```

```
    }
}
int main()
{
   int ch;

   cout<<"1) Insert element to queue"<<endl;
   cout<<"2) Delete element from queue"<<endl;
   cout<<"3) Display all the elements of queue"<<endl;
   cout<<"4) Exit"<<endl;
  do {
      cout<<"Enter your choice : "<<endl;
      cin>>ch;
      switch (ch) {
         case 1: Insert();
         break;
         case 2: Delete();
         break;
         case 3: Display();
         break;
         case 4: cout<<"Exit"<<endl;
         break;
         default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=4);
    return 0;
}
```

Queue using linked list

```
#include <iostream>
```

```
using namespace std;

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


struct node *front;
struct node *rear;

void insert(){
    struct node *ptr;
    int item;

    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr==NULL){

        cout<<"Overflow"<<" "<<endl;
        return;
    }
    else{
        cout<<"Enter value "<<" "<<endl;
        cin>>item;
        ptr->data=item;
        if(front==NULL){
            front=ptr;
            rear=ptr;
            front->next=NULL;
            rear->next=NULL;
        }
        else{
            rear->next=ptr;
```

```
            rear=ptr;
            rear->next=NULL;
        }
    }

}


void Delete(){
    struct node *ptr;
    if(front==NULL){
        cout<<"Underflow"<<" "<<endl;
        return;
    }

    else{
          ptr=front;
          front=front->next;
          free(ptr);


        }

        }

void display(){
    struct node *ptr;
    ptr=front;
    if(front == NULL){
        cout<<" Empty queue :"<<" "<<endl;
    }
    else{
        cout<<" printing values : "<<" "<<endl;
```

```
        while(ptr!=NULL){
            cout<<ptr->data<<" "<<endl;
            ptr=ptr->next;
        }
    }

}

int main(){
int ch;

   cout<<"1) Insert element to queue"<<endl;
   cout<<"2) Delete element from queue"<<endl;
   cout<<"3) Display all the elements of queue"<<endl;
   cout<<"4) Exit"<<endl;
  do {
      cout<<"Enter your choice : "<<endl;
      cin>>ch;
      switch (ch) {
         case 1: insert();
         break;
         case 2: Delete();
         break;
         case 3: display();
         break;
         case 4: cout<<"Exit"<<endl;
         break;
         default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=4);
    return 0;

}
```

Implementation of Stack using Queue:

```
#include<bits/stdc++.h>
using namespace std;
 class Stack {
queue < int > q;
 public:
void Push(int x) {
int s = q.size();
 q.push(x);
 for (int i = 0; i < s; i++)
 { q.push(q.front()); q.pop(); }
}
int Pop()
 {
 int n = q.front();
 q.pop();
return n;
 }
int Top()
 {
return q.front();
}
 int Size()
 { return q.size();
 } };
 int main()
 { Stack s;
s.Push(3);
s.Push(2);
 s.Push(4);
 s.Push(1);
cout << "Top of the stack: " << s.Top() << endl;
 cout << "Size of the stack before removing element: " <<
```

```
s.Size() << endl; cout << "The deleted element is: " <<
s.Pop() << endl; cout << "Top of the stack after removing
element: " << s.Top() << endl; cout << "Size of the stack
after removing element: " << s.Size();
}
```

Remove Outer Parentheses:

A valid parentheses string is either empty "", "(" + A + ")", or A + B, where A and B are valid parentheses strings, and + represents string concatenation.

For example, "", "()", "(())()", and "(()(()))" are all valid parentheses strings.
A valid parentheses string s is primitive if it is nonempty, and there does not exist a way to split it into s = A + B, with A and B nonempty valid parentheses strings.

Given a valid parentheses string s, consider its primitive decomposition: s = P1 + P2 + ... + Pk, where Pi are primitive valid parentheses strings.

Return s after removing the outermost parentheses of every primitive string in the primitive decomposition of s.

Example 1:

Input: s = "(()())(())"
Output: "()()()"
Explanation:
The input string is "(()())(())", with primitive decomposition "(()())" + "(())".
After removing outer parentheses of each part, this is "()()" + "()" = "()()()".

```
class Solution {
public:
    string removeOuterParentheses(string s) {
        int count=0;
        bool flag=true;
        string ans="";
        for(int i=0;i<s.length();i++){
            if(s[i]=='('){
```

```
                count++;
            }
            if(s[i]==')'){
                count--;
            }
            if(count==1 && flag==true){
                flag=false;
                continue;
            }
            if(count==0 && flag==false){
                flag=true;
                continue;
            }

            ans=ans+s[i];
        }

        return ans;

    }
};
```

Snake Pattern:

```
class Solution
{
    public:
    //Function to return list of integers visited in snake pattern in matrix.
    vector<int> snakePattern(vector<vector<int> > matrix)
    {    int n =matrix.size();
         vector<int> ans;
        for(int i = 0; i<n;i++){
        if(i%2==0){
            for(int j = 0;j<n;j++){
                ans.push_back(matrix[i][j]);
            }
        }
        else{
            for(int j =n-1 ; j>=0;j--){
                ans.push_back(matrix[i][j]);
            }

        }
```

```
    }
    return ans;
    }
};
```

Strings - solve strivers problem
Recursion