Singly Linked List :

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

struct Node{
    public:
        int data;
        Node* next;

    public:
        Node(int val, Node* next1){
            data = val;
            next = next1;
        }
        Node(int val){
            data = val;
            next = nullptr;
        }
```

```
    class Node {
        public:
        int data;
        Node* next;

        public:
        Node(int data1, Node* next1) {
            data = data1;
            next = next1;
        }

        public:
        Node(int data1) {
            data = data1;
            next = nullptr;
        }
    };
```

```
Node* converArrToLL(vector<int> &arr){
    Node* head = new Node(arr[0]);
    Node* mover = head;

    for(int i = 1; i < arr.size(); i++){
        Node* temp = new Node(arr[i]);
        mover->next = temp;
        mover = mover->next;
    }
    return head;
}
int lengthOfLL(Node* head){
    int counter = 0;
    Node* temp = head;
    while(temp){
        temp = temp->next;
        counter++;
    }
    return counter;
}

void print(Node* head){
    Node* temp = head;
    while(temp){
        cout << temp->data << " ";
        temp = temp->next;
    }
}

Node* removeTail(Node* head){
    Node* temp = head;
    if(head == NULL || head->next == NULL) return NULL;

    while(temp->next->next){
```

```
        temp = temp->next;
    }
    delete temp->next;
    temp->next = nullptr;
    return head;
}

Node* removeEl(Node* head, int element){
    if(head->next == NULL) return head;
    Node* temp = head;
    Node* prev = nullptr;

    if(head->data == element){
        head = head->next;
        delete temp;
    }

    while(temp){
        if(temp->data == element){
            prev->next = prev->next->next;
            delete temp;
            break;
        }
        prev = temp;
        temp = temp->next;
    }
    return head;
}

Node* insertHead(Node* head, int val){
    if(head == NULL) return new Node(val);
    Node* temp = new Node(val);
    temp->next = head;
    return temp;
}
```

```
Node* insertTail(Node* head, int val){
    if(head == NULL) return new Node(val);

    Node* temp = head;
    while(temp->next){
        temp = temp->next;
    }
    Node* newNode = new Node(val);
    temp->next = newNode;
    return head;
}

Node* insertK(Node* head, int val, int k){
     if(head == NULL) return new Node(val);

     if(k == 1){
         Node* temp = new Node(val);
        temp->next = head;
        return temp;
     }

     int counter = 0;
     Node* temp = head;
     Node* prev = NULL;
     Node* newNode = new Node(val);
     while(temp){
         counter++;
         if(counter == k){
             newNode->next = prev->next;
             prev->next = newNode;
             break;
         }
         prev = temp;
         temp = temp->next;
     }
```

```
    if((counter+1) == k){
      newNode->next = prev->next;
      prev->next = newNode;
    }
    return head;
}

Node* insertVal(Node* head, int val, int before){
    if(head == NULL) return NULL;

    if(before == head->data){
      Node* temp = new Node(val);
      temp->next = head;
      return temp;
    }

    Node* temp = head;
    Node* newNode = new Node(val);
    while(temp->next){
       if(before == temp->next->data){
          newNode->next = temp->next;
          temp->next = newNode;
          break;
       }
       temp = temp->next;
    }
    return head; }

int main() {
   vector<int> arr = {1,3,2,8};
   Node* head = converArrToLL(arr);
      head = insertVal(head, 10, 1);

   print(head);
   return 0; }
```

Doubly Linked List :

```
#include <iostream>
#include <vector>
using namespace std;
struct Node{
    public:
    int data;
    Node* next;
    Node* back;

    public:
    Node(int val, Node* next1, Node* back1){
        data = val;
        next = next1;
        back = back1;
    }
    Node(int val){
        data = val;
        next = nullptr;
        back = nullptr;
    }
};

Node* convertArrToDLL(vector<int> &arr){
    Node* head = new Node(arr[0]);
    Node* prev = head;

    for(int i = 1; i < arr.size(); i++){
        Node* temp = new Node(arr[i] , nullptr, prev);
        prev->next = temp;
        prev = prev->next;
    }
```

```
    return head;
}

void print(Node* head){
    Node* temp = head;

    while(temp){
        cout << temp->data << " ";
        temp = temp->next;
    }
}

Node* deleteHead(Node* head){
    Node* temp = head;
    temp = temp->next;
    temp->back = nullptr;
    head->next = nullptr;
    delete head;
    return temp;
}

Node* deleteTail(Node* head){
    Node* tail = head;
    while(tail->next){
        tail = tail->next;
    }
    Node* newTail = tail->back;
    newTail->next = nullptr;
    tail->back = nullptr;
    delete tail;
    return head;
}
Node* deleteK(Node* head, int k){
    Node* prev = head;
```

```
    while(prev){
        prev = prev->next;
    }
    return head;
}

int main() {
    vector<int> arr = {1,4,3,8};
    Node* head = new Node(arr[0]);
    head = convertArrToDLL(arr);

    print(head);

    return 0;
}
```

LInked List Problems :

1) Middle element :

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        int length = 0;
        ListNode* temp = head;
        while(temp){
            length++;
            temp = temp->next;
        }
        int middle = length / 2;
        int counter = 0;
        temp = head;
        for(int i = 0; i < middle; i++) {
            temp = temp->next;
        }
        return temp; } };
```

2) Binary num in LL to Int :

```
#include <bits/stdc++.h>
class Solution {
public:
    int getDecimalValue(ListNode* head) {
        if(!head) return 0;
        if(!head->next) return head->val * 1;

        ListNode* temp = head;
        int length = 0;
        while(temp){
            length++;
            temp = temp->next;
        }
        int integerVal = 0;
        temp = head;
        int mul = length - 1;
        while(temp){
            int data = temp->val;
            integerVal += pow(2, mul) * data;
            mul--;
            temp = temp->next;
        }
        return integerVal; } };
```

3) Delete array elements in a LL :

```
class Solution {
public:
    ListNode* modifiedList(vector<int>& nums, ListNode* head) {
        int n = nums.size();
        unordered_set<int> sett(nums.begin(), nums.end());

        ListNode* temp = head;
        ListNode* dummy = new ListNode(-1);
```

```
        ListNode* prev = dummy;
        dummy->next = head;

        while(temp){
            if(sett.find(temp->val) != sett.end()){
                prev->next = temp->next;
            }
            else{
                prev = temp;
            }
            temp = temp->next;
        }

        return dummy->next; } };
```

4) Merge 2 sorted LL :

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode* temp1 = list1;
        ListNode* temp2 = list2;
        ListNode* dummy = new ListNode(-1);
        ListNode* prev = dummy;

        while(list1 && list2){
            if(list1->val <= list2->val){
                prev->next = list1;
                list1 = list1->next;
            }
            else{
                prev->next = list2;
                list2 = list2->next;
            }
            prev = prev->next;
        }
```

```
        if(list1) prev->next = list1;
        if(list2) prev->next = list2;

        return dummy->next; } };
```

5) Merge nodes in between zeroes :

```
class Solution {
public:
ListNode* mergeNodes(ListNode* head) {
        vector<int> op;
        int sum = 0;

        ListNode* dummy = new ListNode(-1);
        ListNode* tail = dummy;
        dummy->next = head;
        ListNode* temp = head->next;

        while(temp){
            if(temp->val == 0){
                ListNode* newNode = new ListNode(sum);
                sum = 0;
                tail->next = newNode;
                tail = tail->next;
            }
            else{
                sum += temp->val;
            }
            temp = temp->next;
        }
        return dummy->next; } };
```

More Efficient :

```
class Solution {
public:
ListNode* mergeNodes(ListNode* head) {
```

```
        ListNode* modify = head->next;
        ListNode* mover = modify;

        while(mover){
            int sum = 0;
            while(mover->val != 0){
                sum += mover->val;
                mover = mover->next;
            }
            modify->val = sum;
            mover = mover->next;
            modify->next = mover;
            modify = modify->next;
        }
        return head->next; } };
```

6) Insert GCD between nodes :

```
class Solution {
public:
    int gcd(int a, int b){
        if(b == 0) return a;
        return gcd(b, a % b);
    }

    ListNode* insertGreatestCommonDivisors(ListNode* head) {
        if(!head) return nullptr;
        ListNode* temp = head;
        ListNode* second = head->next;

        while(temp->next){
            int num = gcd(temp->val, second->val);
            ListNode* newNode = new ListNode(num, second);
            temp->next = newNode;
            temp = temp->next->next;
```

```
            second = second->next;
        }
        return head; } };
```

7) Remove nodes from LL :

```
class Solution {
public:
ListNode* reverseList(ListNode* head){
    ListNode* curr = head;
    ListNode* prev = nullptr;
    ListNode* nextNode = nullptr;

    while(curr){
        nextNode = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextNode;
    }
    return prev;
}
    ListNode* removeNodes(ListNode* head) {
        head = reverseList(head);

        int maxi = head->val;
        ListNode* temp = head;
        ListNode* newHead = head;

        while(temp->next){
            if(temp->next->val >= maxi){
                maxi = temp->next->val;
                temp = temp->next;
            }
            else{
                temp->next = temp->next->next;
            }
```

```
        }

        return reverseList(newHead); } };
```

8) Reverse SLL :

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* curr = head;
        ListNode* prev = nullptr;
        ListNode* nextNode = nullptr;

        while(curr){
            nextNode = curr->next;
            curr->next = prev;
            prev = curr;
            curr = nextNode;
        }
        return prev; } };
```

9) Palindrome :

```
class Solution {
public:

    ListNode* reverseList(ListNode* head){
        ListNode* prev = nullptr;
        ListNode* curr = head;
        ListNode* nextNode = nullptr;

        while(curr){
            nextNode = curr->next;
            curr->next = prev;
            prev = curr;
            curr = nextNode;
        }
        return prev;
```

```
    }
    bool isPalindrome(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;
        while(fast && fast->next){
            slow = slow->next;
            fast = fast->next->next;
        }

        ListNode* secondHalf = reverseList(slow);
        ListNode* temp2 = secondHalf;
        ListNode* firstHalf = head;

        while(temp2){
            if(temp2->val != firstHalf->val) return false;
            temp2 = temp2->next;
            firstHalf = firstHalf->next;
        }
        return true; } };
```

10) Remove nth node from back :

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        if(!head || !head->next) return nullptr;

        ListNode* temp = head;
        ListNode* nextNode = head;
        ListNode* dummy = new ListNode(-1);
        dummy->next = head;
        ListNode* prev = dummy;

        for(int i = 0; i < n; i++){
            nextNode = nextNode->next;
```

```
        }

        while(nextNode){
            nextNode = nextNode->next;
            prev = temp;
            temp = temp->next;
        }
        prev->next = temp->next;
        temp->next = nullptr;
        return dummy->next;
    }
};
```

11) Odd Even LL :

```
class Solution {
public:
    ListNode* oddEvenList(ListNode* head) {
        if(!head || !head->next) return head;

        ListNode* odd = head;
        ListNode* even = head->next;
        ListNode* firstEven = head->next;
        int counter = 1;

        while(even && even->next){
            odd->next = even->next;
            odd = odd->next;

            even->next = odd->next;
            even = even->next;
        }
        odd->next = firstEven;
        return head; } };
```

12) Delete middle node in LL :

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if(!head || !head->next) return nullptr;
        if(!head->next->next){
            head->next = nullptr;
            return head;
        }
        ListNode* temp = head;
        int length = 0;
        while(temp){
            temp = temp->next;
            length++;
        }
        int middle = length / 2;

        temp = head;
        for(int i = 0; i < middle - 1; i++){
            temp = temp->next;
        }
        if(temp->next->next) temp->next = temp->next->next;
        return head;
    }
};
```

Optimized (two pointers) :

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if(!head) return nullptr;
        ListNode* prev = new ListNode(0);
        prev->next = head;
        ListNode* slow = prev;
```

```
        ListNode* fast = head;
        while(fast && fast->next){
            slow = slow->next;
            fast = fast->next->next;
        }

        slow->next = slow->next->next;
        return prev->next;
    }
};
```

13) Reverse LL recursively :

```
class Solution {
public:

    ListNode* reverseList(ListNode* head) {
        if(!head || !head->next){
            return head;
        }
        ListNode* newHead = reverseList(head->next);
        ListNode* front = head->next;
        front->next = head;
        head->next = nullptr;
        return newHead;
    }
};
```

14) Detect Loop :

Hashap :

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        unordered_map<ListNode*, int> mpp;
        ListNode* temp = head;

        while(temp){
```

```
            if(mpp.find(temp) != mpp.end()){
                return true;
            }
            else mpp[temp] = 1;
            temp = temp->next;
        }
        return false;
    }
};
```

Floyd's Cycle Detection (Tortoise and Hare Algorithm)

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        if(!head || !head->next) return false;
        ListNode* slow = head;
        ListNode* fast = head->next;

        while(fast && fast->next){
            if(slow == fast){
                return true;
            }
            slow = slow->next;
            fast = fast->next->next;
        }
        return false;
    }
};
```

15) Detect starting point :

```
class Solution {
public:

    ListNode *detectCycle(ListNode *head) {
        if(!head || !head->next) return nullptr;
```

```
        ListNode* slow = head;
        ListNode* fast = head;

        while(fast && fast->next){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast){
                slow = head;
                while(slow != fast){
                    slow = slow->next;
                    fast = fast->next;
                }
                return slow;
            }
        }
        return nullptr;
    }
};
```

16) Length of the Loop :

```
class Solution {
  public:

    int countNodesinLoop(Node *head) {
        if(!head || !head->next) return 0;
        Node* slow = head;
        Node* fast = head;

        while(fast && fast->next){
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast){
                fast = fast->next;
                int counter = 1;
```

```
            while(slow != fast){
                counter++;
                fast = fast->next;
            }
            return counter;
        }
    }
    return 0;

  }
};
```