

# linked list

pepcoding




## Linked list introduction


■■■■■

LEVEL-1

**LINKEDLISTS**  
**INTRODUCTION**

✕





✕

[www.nados.pepcoding.com](http://www.nados.pepcoding.com)

For better experience visit

# Add Last In Linked List

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of addLast function. This function is supposed to add an element to the end of LinkedList. You are required to update head, tail and size as required.
3. Input and Output is managed for you. Just update the code in addLast function.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addLast 10
addLast 20
addLast 30
addLast 40
addLast 50
quit
```

### Sample Output

```
10
20
30
40
50
5
50
```

```

#include <iostream>

using namespace std;

class node
{
public :
    int data;
    node* next;
};

class linked_list
{
public:
    node* head, *tail;
    int size = 0;

public:
    linked_list()
    {
        head = NULL;
        tail = NULL;
    }

    void addLast(int n)
    {
        // Write your code here
        node* temp = new node;
        temp->data = n;
        temp->next = NULL;

        if (size == 0) {
            head = temp;
            tail = temp;
            // don't delete the temp plz don't
        } else { // for size = 1 also this will work
            tail->next = temp;
            tail = tail->next;
        }
        size++;
    }

    void display() {
        for (node* tmp = head; tmp != NULL; tmp = tmp->next) {
            cout << tmp->data << " ";
        }
    }
}

```

```

void testList() {
    for (node* temp = head; temp != NULL; temp = temp->next) {
        cout << temp->data << endl;
    }
    cout<<size<< endl;

    if (size > 0) {
        cout <<tail->data << endl;
    }
}
};

```

```

int main() {

    string str;
    linked_list l;
    while (true) {
        getline(cin, str);
        if (str[0] == 'q') {
            break;
        }
        if (str[0] == 'a') {
            string ss = str.substr(8, 2);
            int n = stoi(ss);
            l.addLast( n);
        }

    }
    l.testList();
    return 0;
}

```

## Display A Linkedlist

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
3. You are required to complete the body of display function and size function
  - 3.1. display - Should print the elements of linked list from front to end in a single line. Elements should be separated by space.
  - 3.2. size - Should return the number of elements in the linked list.
4. Input and Output is managed for you.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addLast 10
addLast 20
addLast 30
display
size
addLast 40
addLast 50
display
size
quit
```

### Sample Output

```
10 20 30
3
10 20 30 40 50
5
```

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

```
class node
{
public :
    int data;
    node* next;
};
```

```
class linked_list
{
public:
    node* head, *tail;
    int size = 0;
```

```
public:
    linked_list()
    {
        head = NULL;
        tail = NULL;
    }
```

```

void addLast(int n)
{
    node* tmp = new node;
    tmp->data = n;
    tmp->next = NULL;

    if (head == NULL)
    {
        head = tmp;
        tail = tmp;
    }
    else
    {
        tail->next = tmp;
        tail = tail->next;
    }
    size++;
}

int length(){
    return this->size;
}

void display() {
    // write your code here
    node* temp = new node;
    temp = head;
    while(temp != NULL) {
        cout<<temp->data<<" ";
        temp = temp->next;
    }
    if(size>0){
        cout<<endl;
    }
}

void testList() {
    for (node* temp = head; temp != NULL; temp = temp->next) {
        cout << temp->data << endl;
    }
    cout << size << endl;

    if (size > 0) {
        cout << tail->data << endl;
    }
}
};

```

```

int main() {

    string str;
    linked_list l;
    while (true) {
        getline(cin, str);
        if (str[0] == 'q') {
            break;
        }
        else if (str[0] == 'a') {
            string ss = str.substr(8, 2);
            int n = stoi(ss);
            l.addLast( n);

        }
        else if (str[0] == 's') {
            cout << l.length() << endl;
        }
        else if (str[0] == 'd') {
            l.display();
        }

    }

    return 0;
}

```

## Remove First In Linkedlist

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line. All elements are separated by space
  - 2.3. size - Returns the number of elements in the linked list.
3. You are required to complete the body of removeFirst function
  - 3.1. removeFirst - This function is required to remove the first element from Linked List. Also, if there is only one element, this should set head and tail to null. If there are no elements, this should print "List is empty".
4. Input and Output is managed for you.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addLast 10
addLast 20
addLast 30
display
removeFirst
size
addLast 40
addLast 50
removeFirst
display
size
removeFirst
removeFirst
removeFirst
removeFirst
quit
```

### Sample Output

```
10 20 30
2
30 40 50
3
List is empty
```

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

```
class node
{
public :
    int data;
    node* next;
};
```



```

class linked_list
{
public:
    node* head, *tail;
    int size = 0;

public:
    linked_list()
    {
        head = NULL;
        tail = NULL;
    }

    void addLast(int n)
    {
        node* tmp = new node;
        tmp->data = n;
        tmp->next = NULL;

        if (head == NULL)
        {
            head = tmp;
            tail = tmp;
        }
        else
        {
            tail->next = tmp;
            tail = tail->next;
        }
        size++;
    }

    void display() {
        for (node* tmp = head; tmp != NULL; tmp = tmp->next) {
            cout << tmp->data << " ";
        }
        cout << endl;
    }

    void removeFirst() {
        //write your code here
        if(size == 0){
            cout<<"List is empty"<<endl;
        }else if(size == 1){
            // node* t = new node;
            node* t = head;
            head = NULL;
            tail = NULL;
            size--;
            delete t;
        }else{

```

```

// node* t = new node;
node* t = head;
head = head->next;
size--;
delete t;
}
}

```

```

void testList() {
    for (node* temp = head; temp != NULL; temp = temp->next) {
        cout << temp->data << endl;
    }
    cout << size << endl;

    if (size > 0) {
        cout << tail->data << endl;
    }
}
};

```

```

int main() {

    string str;
    linked_list l;
    while (true) {
        getline(cin, str);
        if (str[0] == 'q') {
            break;
        }
        else if (str[0] == 'a') {
            string ss = str.substr(8, 2);
            int n = stoi(ss);
            l.addLast( n);

        }
        else if (str[0] == 's') {
            cout << l.size << endl;
        }
        else if (str[0] == 'd') {
            l.display();
        }
        else if (str[0] == 'r') {
            l.removeFirst();
        }
    }
    return 0;
}

```

# Get Value In Linked List

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line.  
All elements are separated by space.
  - 2.3. size - Returns the number of elements in the linked list.
  - 2.4. removeFirst - Removes the first element from Linked List.
3. You are required to complete the body of getFirst, getLast and getAt function
  - 3.1. getFirst - Should return the data of first element. If empty should return -1 and print "List is empty".
  - 3.2. getLast - Should return the data of last element. If empty should return -1 and print "List is empty".
  - 3.3. getAt - Should return the data of element available at the index passed. If empty should return -1 and print "List is empty". If invalid index is passed, should return -1 and print "Invalid arguments".
4. Input and Output is managed for you.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addLast 10
getFirst
addLast 20
addLast 30
getFirst
getLast
getAt 1
addLast 40
getLast
addLast 50
removeFirst
getFirst
removeFirst
removeFirst
getAt 3
removeFirst
removeFirst
getFirst
quit
```

### Sample Output

```
10
10
30
20
40
20
Invalid arguments
List is empty
```

```

#include <iostream>
using namespace std;
class node{
public:
    int val;
    node* next;

};

class LinkedList {
public: node* head=nullptr;
    node* tail=nullptr;
    int size_=0;

void insert_at_tail(int val){
    if(head==NULL){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        head=newnode;

    }
    else{
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        node *temp = head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next = newnode;
    }

}

void print (){
    node *temp =head;
    if(head==NULL){
        cout << "0" << endl;
        return;
    }
    while(temp!=NULL){
        cout<<temp->val<<" ";
        temp=temp->next;
    }
    cout << endl;
}

void deletion_at_head(){
    if(head==NULL) return;
    node *temp=head;
    head=head->next;
}

```

```

        delete temp;
    }

    int size(){
        int cnt=0;
        node* temp=head;
        while(temp!=NULL){
            temp=temp->next;
            cnt++;
        }
        return cnt;
    }

    void getFirst(){
        //write your code here
        cout<<head->val<<endl;
    }

    void getLast(){
        //write your code here
        node * t = head;
        while(t->next != NULL){
            t = t->next;
        }
        cout<<t->val<<endl;
    }

    void getAt(int p){
        //write your code here
        node * t = head;
        for(int i {}; i < p ;i++) {
            t = t->next;
        }
        cout<<t->val<<endl;
    }
};

int main() {
    LinkedList l1;
    string s;
    cin >> s;
    while(s!="quit"){
        if(s=="addLast"){
            int data;
            cin>> data;
            l1.insert_at_tail(data);
        }
        else if(s=="getFirst"){
            if(l1.head!=NULL){
                l1.getFirst();
            }else{
                cout << "List is empty" << endl;
            }
        }
    }
}

```

```

    }
}
else if(s=="getLast"){
    if(l1.head!=NULL){
        l1.getLast();
    }
    else
    {
        cout<<"List is empty";
    }
}
else if(s=="removeFirst"){
    if(l1.head!=NULL){
        l1.deletion_at_head();
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s=="getAt"){
    if(l1.head!=NULL){
        int i;
        cin >> i;
        if(i>=l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.getAt(i);
        }
    }
    else
    {
        cout<<"List is empty";
    }
}
else if(s=="display"){
    if(l1.head!=NULL){
        l1.print();
    }
    else{
        cout << endl;
    }
}
else if(s=="size"){
    if(l1.head!=NULL){
        cout << l1.size() << endl;
    }
}
cin>>s;
}
}

```

# Add First In Linked List

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line.  
All elements are separated by space.
  - 2.3. size - Returns the number of elements in the linked list.
  - 2.4. removeFirst - Removes the first element from Linked List.
  - 2.5. getFirst - Returns the data of first element.
  - 2.6. getLast - Returns the data of last element.
  - 2.7. getAt - Returns the data of element available at the index passed.
3. You are required to complete the body of addFirst function. This function should add the element to the beginning of the linkedlist and appropriately set the head, tail and size data-members.
4. Input and Output is managed for you.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addFirst 10
getFirst
addFirst 20
getFirst
getLast
display
size
addLast 40
getLast
addLast 50
addFirst 30
removeFirst
getFirst
removeFirst
removeFirst
getAt 3
display
size
removeFirst
removeFirst
getFirst
quit
```

### Sample Output

```
10
20
10
```

```
20 10
2
40
20
Invalid arguments
40 50
2
```

List is empty

```
#include <iostream>
using namespace std;
class node{
public:
    int val;
    node* next;

};
class LinkedList {
public: node* head=NULLptr;
    node* tail=NULLptr;
    int size_=0;

void insert_at_tail(int val){
    if(head==NULL){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        head=newnode;

    }
    else{
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        node *temp = head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next = newnode;
    }

}

void print (){
    node *temp =head;
    if(head==NULL){
        cout << "0" << endl;
        return;
    }
    while(temp!=NULL){
        cout<<temp->val<<" ";
        temp=temp->next;
    }
}
```



```

    }
    cout << endl;
}
void deletion_at_head(){
    if(head==NULL) return;
    node *temp=head;
    head=head->next;
    delete temp;
}

int size(){
    int cnt=0;
    node* temp=head;
    while(temp!=NULL){
        temp=temp->next;
        cnt++;
    }
    return cnt;
}

void getFirst(){
    cout<<head->val<<endl;
}

void getLast(){
    node* temp=head;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    cout << temp->val << endl;
}

node* getAt(int p){
    int cnt=0;
    node* temp=head;
    while(cnt < p){
        cnt++;
        temp=temp->next;
    }
    cout << temp->val << endl;
    return temp;
}

void addFirst(int val){
    //write your code here
    node * t = new node;
    t->val = val;
    if(size() == 0){
        head = t;
        tail = t;
        head->next = NULL;
    }
}

```

```

        size_++;
    }else{
        t->next = head;
        head = t;
        size_++;
    }
}

};

int main() {
    LinkedList l1;
    string s;
    cin >> s;
    while(s!="quit"){
        if(s=="addLast"){
            int data;
            cin>> data;
            l1.insert_at_tail(data);
        }
        else if(s=="addFirst"){
            int data;
            cin>>data;
            l1.addFirst(data);
        }
        else if(s=="getFirst"){
            if(l1.head!=NULL){
                l1.getFirst();
            }else{
                cout << "List is empty" << endl;
            }
        }
        else if(s=="getLast"){
            if(l1.head!=NULL){
                l1.getLast();
            }
            else
            {
                cout<<"List is empty";
            }
        }
        else if(s=="removeFirst"){
            if(l1.head!=NULL){
                l1.deletion_at_head();
            }
            else{
                cout << "List is empty" << endl;
            }
        }
        else if(s=="getAt"){
            if(l1.head!=NULL){
                int i;

```

```

        cin >> i;
        if(i>=l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.getAt(i);
        }
    }
    else
    {
        cout<<"List is empty";
    }
}
else if(s=="display"){
    if(l1.head!=NULL){
        l1.print();
    }
    else{
        cout << endl;
    }
}
else if(s=="size"){
    if(l1.head!=NULL){
        cout << l1.size() << endl;
    }
}
cin>>s;
}
}

```

## Add At Index In Linked List

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line. All elements are separated by space
  - 2.3. size - Returns the number of elements in the linked list.
  - 2.4. removeFirst - Removes the first element from Linked List.
  - 2.5. getFirst - Returns the data of first element.
  - 2.6. getLast - Returns the data of last element.
  - 2.7. getAt - Returns the data of element available at the index passed.
  - 2.8. addFirst - adds a new element with given value in front of linked list.
3. You are required to complete the body of addAt function. This function should add the element at the index mentioned as parameter. If the idx is inappropriate print "Invalid arguments".
4. Input and Output is managed for you.

### Constraints

None

### Format

## Input

Input is managed for you

## Output

Output is managed for you

## Example

### Sample Input

```
addFirst 10
getFirst
addAt 0 20
getFirst
getLast
display
size
addAt 2 40
getLast
addAt 1 50
addFirst 30
removeFirst
getFirst
removeFirst
removeFirst
addAt 2 60
display
size
removeFirst
removeFirst
getFirst
quit
```

### Sample Output

```
10
20
10
20 10
2
40
20
10 40 60
3
60
```

```
#include <iostream>
using namespace std;
class node{
public:
    int val;
    node* next;

};
class LinkedList {
public: node* head=nullptr;
    node* tail=nullptr;
    int size_=0;
```

```

void insert_at_tail(int val){
    if(head==NULL){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        head=newnode;
    }
    else{
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        node *temp = head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next = newnode;
    }
}

```

```

void print (){
    node *temp =head;
    if(head==NULL){
        cout << "0" << endl;
        return;
    }
    while(temp!=NULL){
        cout<<temp->val<<" ";
        temp=temp->next;
    }
    cout << endl;
}

```

```

void deletion_at_head(){
    if(head==NULL) return;
    node *temp=head;
    head=head->next;
    delete temp;
}

```

```

int size(){
    int cnt=0;
    node* temp=head;
    while(temp!=NULL){
        temp=temp->next;
        cnt++;
    }
    return cnt;
}

```

```

void getFirst(){

```

```

        cout<<head->val<<endl;
    }

    void getLast(){
        node* temp=head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        cout << temp->val << endl;
    }

    node* getAt(int p){
        int cnt=0;
        node* temp=head;
        while(cnt < p){
            cnt++;
            temp=temp->next;
        }
        cout << temp->val << endl;
        return temp;
    }

    void addFirst(int val){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        if(head==NULL){
            head=newnode;
        }
        else{
            newnode ->next =head;
            head = newnode;
        }
        size_++;
    }

    void addAt(int pos,int data){
        //write your code here
        if (pos == 0) {
            addFirst(data);
        }else if(pos == size()){
            node *newnode = new node;
            newnode->val=data;
            newnode->next=NULL;
            node * c = head; // tail is not maintained in the above
code i think so manually go to end
            while (c->next != NULL){
                c = c->next;
            }
            c->next = newnode;
            tail = newnode;
        }
    }

```

```

        size_++;
        // insert_at_tail(data);
    }else{
        node *newnode = new node;
        newnode->val=data;

        node * c = head;
        for(int i{}; i < pos-1 ; i++) {
            c = c->next;
        }
        newnode->next = c->next;
        c->next = newnode;
        size_++;
    }
}

};

int main() {
    LinkedList l1;
    string s;
    cin >> s;
    while(s!="quit"){
        if(s=="addLast"){
            int data;
            cin>> data;
            l1.insert_at_tail(data);
        }
        else if(s=="addFirst"){
            int data;
            cin>>data;
            l1.addFirst(data);
        }
        else if(s=="getFirst"){
            if(l1.head!=NULL){
                l1.getFirst();
            }else{
                cout << "List is empty" << endl;
            }
        }
        else if(s=="getLast"){
            if(l1.head!=NULL){
                l1.getLast();
            }
            else
            {
                cout<<"List is empty";
            }
        }
        else if(s=="removeFirst"){
            if(l1.head!=NULL){
                l1.deletion_at_head();
            }
        }
    }
}

```

```

    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s=="addAt"){
    int val,i;
    cin>>i>>val;
    if(i>l1.size()){
        cout<<"Invalid arguments"<<endl;
    }
    else{
        l1.addAt(i,val);
    }
}
else if(s=="getAt"){
    if(l1.head!=NULL){
        int i;
        cin >> i;
        if(i>=l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.getAt(i);
        }
    }
    else
    {
        cout<<"List is empty";
    }
}
else if(s=="display"){
    if(l1.head!=NULL){
        l1.print();
    }
    else{
        cout << endl;
    }
}
else if(s=="size"){
    if(l1.head!=NULL){
        cout << l1.size() << endl;
    }
}
cin>>s;
}
}

```



# Remove Last In Linked List

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line.  
All elements are separated by space
  - 2.3. size - Returns the number of elements in the linked list.
  - 2.4. removeFirst - Removes the first element from Linked List.
  - 2.5. getFirst - Returns the data of first element.
  - 2.6. getLast - Returns the data of last element.
  - 2.7. getAt - Returns the data of element available at the index passed.
  - 2.8. addFirst - adds a new element with given value in front of linked list.
  - 2.9. addAt - adds a new element at a given index.
3. You are required to complete the body of removeLast function. This function should remove the last element and update appropriate data members. If the size is 0, should print "List is empty". If the size is 1, should set both head and tail to null.
4. Input and Output is managed for you.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addFirst 10
getFirst
addAt 0 20
getFirst
getLast
display
size
addAt 2 40
getLast
addAt 1 50
addFirst 30
removeFirst
getFirst
removeLast
removeLast
addAt 2 60
display
size
removeFirst
removeLast
getFirst
quit
```

### Sample Output

10  
20  
10  
20 10

2  
40  
20  
20 50 60  
3  
50

```
#include <iostream>
using namespace std;
class node{
public:
    int val;
    node* next;

};

class LinkedList {
public: node* head=nullptr;
    node* tail=nullptr;
    int size_=0;

void insert_at_tail(int val){
    if(head==NULL){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        head=newnode;

    }
    else{
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        node *temp = head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next = newnode;
    }

}

void print (){
    node *temp =head;
    if(head==NULL){
        cout << "0" << endl;
        return;
    }
    while(temp!=NULL){
```

```

        cout<<temp->val<<" ";
        temp=temp->next;
    }
    cout << endl;
}
void deletion_at_head(){
    if(head==NULL) return;
    node *temp=head;
    head=head->next;
    delete temp;
}

int size(){
    int cnt=0;
    node* temp=head;
    while(temp!=NULL){
        temp=temp->next;
        cnt++;
    }
    return cnt;
}

void getFirst(){
    cout<<head->val<<endl;
}

void getLast(){
    node* temp=head;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    cout << temp->val << endl;
}

node* getAt(int p){
    int cnt=0;
    node* temp=head;
    while(cnt < p){
        cnt++;
        temp=temp->next;
    }
    cout << temp->val << endl;
    return temp;
}

void addFirst(int val){
    node *newnode = new node;
    newnode->val=val;
    newnode->next=NULL;
    if(head==NULL){
        head=newnode;
    }
}

```

```

    }
    else{
        newnode ->next =head;
        head = newnode;
    }
}

void addAt(int pos,int data){
    if(pos==0){
        addFirst(data);
        return;
    }
    node* newnode=new node;
    newnode->val=data;
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    newnode->next=temp->next;
    temp->next=newnode;
}

void removeLast(){
    //write your code here
    //size == 0 case is handled while calling the function but
    adding here also -> formality
    if(size() == 0 ) {
        cout<<"List is empty"<<endl;
    }else if(size() == 1){
        node* last = head;
        head = NULL;
        tail = NULL;
        delete last;
        size_--;
    }else{
        node* last = tail;//for deleting the memory on the heap
        node* c = head;
        for(int i {};i < size()-2;i++) {
            c = c->next;
        }
        c->next = NULL;
        size_--;
        delete last;
    }
}

};

int main() {
    LinkedList l1;

```

```

string s;
cin >> s;
while(s!="quit"){
    if(s=="addLast"){
        int data;
        cin>> data;
        l1.insert_at_tail(data);
    }
    else if(s=="addFirst"){
        int data;
        cin>>data;
        l1.addFirst(data);
    }
    else if(s=="getFirst"){
        if(l1.head!=NULL){
            l1.getFirst();
        }else{
            cout << "List is empty" << endl;
        }
    }
    else if(s=="getLast"){
        if(l1.head!=NULL){
            l1.getLast();
        }
        else
        {
            cout<<"List is empty";
        }
    }
    else if(s=="removeFirst"){
        if(l1.head!=NULL){
            l1.deletion_at_head();
        }
        else{
            cout << "List is empty" << endl;
        }
    }
    else if(s=="removeLast"){
        if(l1.head!=NULL){
            l1.removeLast();
        }
        else{
            cout<<"List is empty"<<endl;
        }
    }
    else if(s=="addAt"){
        int val,i;
        cin>>i>>val;
        if(i>l1.size()){
            cout<<"Invalid arguments"<<endl;
        }
    }
}

```

```

        else{
            l1.addAt(i,val);
        }
    }
    else if(s=="getAt"){
        if(l1.head!=NULL){
            int i;
            cin >> i;
            if(i>=l1.size()){
                cout << "Invalid arguments" << endl;
            }
            else{
                l1.getAt(i);
            }
        }
        else
        {
            cout<<"List is empty";
        }
    }
    else if(s=="display"){
        if(l1.head!=NULL){
            l1.print();
        }
        else{
            cout << endl;
        }
    }
    else if(s=="size"){
        if(l1.head!=NULL){
            cout << l1.size() << endl;
        }
    }
    cin>>s;
}
}

```

## Remove At Index In Linked List

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line. All elements are separated by space
  - 2.3. size - Returns the number of elements in the linked list.
  - 2.4. removeFirst - Removes the first element from Linked List.
  - 2.5. getFirst - Returns the data of first element.
  - 2.6. getLast - Returns the data of last element.
  - 2.7. getAt - Returns the data of element available at the index passed.
  - 2.8. addFirst - adds a new element with given value in front of linked list.

2.9. addAt - adds a new element at a given index.

2.10. removeLast - removes the last element of linked list.

3. You are required to complete the body of removeAt function. The function should remove the element available at the index passed as parameter. If the size is 0, should print "List is empty". If the index is inappropriate print "Invalid arguments". Also consider the case when list has a single element.

4. Input and Output is managed for you.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addFirst 10
getFirst
addAt 0 20
getFirst
getLast
display
size
addAt 2 40
getLast
addAt 1 50
addFirst 30
removeAt 2
getFirst
removeAt 0
removeAt 1
addAt 2 60
display
size
removeAt 0
removeAt 1
getFirst
quit
```

### Sample Output

```
10
20
10
20 10
2
40
30
20 40 60
3
40
```

```

#include <iostream>
using namespace std;
class node{
public:
    int val;
    node* next;
};
class LinkedList {
public: node* head=nullptr;
    node* tail=nullptr;
    int size_=0;

void insert_at_tail(int val){
    if(head==NULL){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        head=newnode;

    }
    else{
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        node *temp = head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next = newnode;
    }

}

void insertion_at_head(int val){
    node *newnode = new node;
    newnode->val=val;
    newnode->next=NULL;
    if(head==NULL){
        head=newnode;
    }
    else{
        newnode ->next =head;
        head = newnode;
    }

}

void print (){
    node *temp =head;
    if(head==NULL){
        cout << "0" << endl;
        return;
    }
    while(temp!=NULL){

```



```

        cout<<temp->val<<" ";
        temp=temp->next;
    }
    cout << endl;
}
void deletion_at_head(){
    if(head==NULL) return;
    node *temp=head;
    head=head->next;
    delete temp;
}
void deletion_at_tail(){
    if(head==NULL) return;
    node* previous=NULL;
    node* temp=head;
    while(temp->next!=NULL){
        previous=temp;
        temp=temp->next;
    }
    previous->next = NULL;
    size--;
    delete temp;
}
void last(){
    node* temp=head;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    cout << temp->val << endl;
}
int size(){
    int cnt=0;
    node* temp=head;
    while(temp!=NULL){
        temp=temp->next;
        cnt++;
    }
    return cnt;
}
void first(){
    cout << head->val << endl;
}

node* getAt(int p){
    int cnt=0;
    node* temp=head;
    while(cnt < p){
        cnt++;
        temp=temp->next;
    }
    cout << temp->val << endl;
}

```

```

        return temp;
    }

    void addAt(int pos, int data){
        if(pos==0){
            insertion_at_head(data);
            return;
        }
        node* newnode=new node;
        newnode->val=data;
        int cnt=0;
        node* temp=head;
        while(cnt<pos-1){
            cnt++;
            temp=temp->next;
        }
        newnode->next=temp->next;
        temp->next=newnode;
    }

    void removeAt(int pos){
        //write your code here
        if (pos == 0 ){
            if (head == NULL){
                cout<<"List is empty";
                return;
            }
            node * todel = head;
            head = head->next;
            size--;
            delete todel;
        }else if( pos == size()) {
            deletion_at_tail();
        }else{
            node * c = head;
            for(int i {};i < pos-1;i++){
                c = c->next;
            }
            node * todel = c->next;
            c->next = todel->next;
            size--;
            delete todel;
        }
    }

};

int main() {
    LinkedList l1;
    string s;
    cin >> s;

```

```

while(s!="quit"){
    if(s=="addLast"){
        int data;
        cin>> data;
        l1.insert_at_tail(data);
    }
    else if(s=="addFirst"){
        int data;
        cin>> data;
        l1.insertion_at_head(data);
    }
    else if(s=="getFirst"){
        if(l1.head!=NULL){
            l1.first();
        }else{
            cout << "List is empty" << endl;
        }
    }
    else if(s=="getLast"){
        if(l1.head!=NULL){
            l1.last();
        }
        else
        {
            cout<<"List is empty";
        }
    }
    else if(s=="removeFirst"){
        if(l1.head!=NULL){
            l1.deletion_at_head();
        }
        else{
            cout << "List is empty" << endl;
        }
    }
    else if(s=="getAt"){
        if(l1.head!=NULL){
            int i;
            cin >> i;
            if(i>=l1.size()){
                cout << "Invalid arguments" << endl;
            }
            else{
                l1.getAt(i);
            }
        }
        else
        {
            cout<<"List is empty";
        }
    }
}

```

```

    }
    else if(s=="display"){
        if(l1.head!=NULL){
            l1.print();
        }
        else{
            cout << endl;
        }
    }
    else if(s=="size"){
        if(l1.head!=NULL){
            cout << l1.size() << endl;
        }
    }
    else if(s=="addAt"){
        int val,i;
        cin >> i >> val;
        if(i>l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.addAt(i,val);
        }
    }
    else if(s == "removeLast"){
        if(l1.head!=NULL){
            l1.deletion_at_tail();
        }
        else{
            cout << "List is empty" << endl;
        }
    }
    else if(s == "removeAt"){
        if(l1.head!=NULL){
            int i;
            cin >> i;
            if(i>=l1.size()){
                cout << "Invalid arguments" << endl;
            }
            else{
                l1.removeAt(i);
            }
        }
        else{
            cout << "List is empty" << endl;
        }
    }
    cin >> s;
}
}

```

# Reverse A Linked List (data Iterative)

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line.  
All elements are separated by space
  - 2.3. size - Returns the number of elements in the linked list.
  - 2.4. removeFirst - Removes the first element from Linked List.
  - 2.5. getFirst - Returns the data of first element.
  - 2.6. getLast - Returns the data of last element.
  - 2.7. getAt - Returns the data of element available at the index passed.
  - 2.8. addFirst - adds a new element with given value in front of linked list.
  - 2.9. addAt - adds a new element at a given index.
  - 2.10. removeLast - removes the last element of linked list.
  - 2.11. removeAt - remove an element at a given index.
3. You are required to complete the body of reverseDI function. The function should be an iterative function and should reverse the contents of linked list by changing the "data" property of nodes.
4. Input and Output is managed for you.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addFirst 10
addFirst 20
addLast 30
addLast 40
addLast 50
addFirst 60
removeAt 2
display
reverseDI
display
quit
```

### Sample Output

```
60 20 30 40 50
50 40 30 20 60
```

```
#include <iostream>
using namespace std;
class node{
public:
    int val;
    node* next;
```

```
};
```

```

class LinkedList {
    public: node* head=nullptr;
    node* tail=nullptr;
    int size_=0;

    void insert_at_tail(int val){
        if(head==NULL){
            node *newnode = new node;
            newnode->val=val;
            newnode->next=NULL;
            head=newnode;

        }
        else{
            node *newnode = new node;
            newnode->val=val;
            newnode->next=NULL;
            node *temp = head;
            while(temp->next!=NULL){
                temp=temp->next;
            }
            temp->next = newnode;
        }
    }

    void insertion_at_head(int val){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        if(head==NULL){
            head=newnode;
        }
        else{
            newnode ->next =head;
            head = newnode;
        }
    }

    void print (){
        node *temp =head;
        if(head==NULL){
            cout << "0" << endl;
            return;
        }
        while(temp!=NULL){
            cout<<temp->val<<" ";
            temp=temp->next;
        }
        cout << endl;
    }

    void deletion_at_head(){
        if(head==NULL) return;
    }

```

```

        node *temp=head;
        head=head->next;
        delete temp;
    }
    void deletion_at_tail(){
        if(head==NULL) return;
        node* previous=NULL;
        node* temp=head;
        while(temp->next!=NULL){
            previous=temp;
            temp=temp->next;
        }
        previous->next = NULL;
        delete temp;
    }

    void last(){
        node* temp=head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        cout << temp->val << endl;
    }
    int size(){
        int cnt=0;
        node* temp=head;
        while(temp!=NULL){
            temp=temp->next;
            cnt++;
        }
        return cnt;
    }
    void first(){
        cout << head->val << endl;
    }

    node* getAt(int p){
        int cnt=0;
        node* temp=head;
        while(cnt < p){
            cnt++;
            temp=temp->next;
        }
        return temp;
    }

    void addAt(int pos,int data){
        if(pos==0){
            insertion_at_head(data);
            return;
        }
    }

```

```

    }
    node* newnode=new node;
    newnode->val=data;
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    newnode->next=temp->next;
    temp->next=newnode;
}

void removeAt(int pos){
    if(pos==0){
        deletion_at_head();
        return;
    }
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    temp->next = temp->next->next;
}

void reverse_di(){
    //write your code here
    if (size() == 0 ){
        cout<<"List is empty"<<endl;
    }else{
        node * c = head ;
        int n = size()-1;
        for(int i {};i < size()/2;i++) {
            node * tochangewith = head;
            for(int j{};j < n;j++){
                tochangewith = tochangewith->next;
            }
            int a = c->val;
            c-> val = tochangewith->val;
            tochangewith->val = a;
            c = c->next;
            n--;
        }
    }
}

};

```



```

int main() {
    LinkedList l1;
    string s;
    cin >> s;
    while(s!="quit"){
        if(s=="addLast"){
            int data;
            cin>> data;
            l1.insert_at_tail(data);
        }
        else if(s=="addFirst"){
            int data;
            cin>> data;
            l1.insertion_at_head(data);
        }
        else if(s=="getFirst"){
            if(l1.head!=NULL){
                l1.first();
            }else{
                cout << "List is empty" << endl;
            }
        }
        else if(s=="getLast"){
            if(l1.head!=NULL){
                l1.last();
            }
            else
            {
                cout<<"List is empty";
            }
        }
        else if(s=="removeFirst"){
            if(l1.head!=NULL){
                l1.deletion_at_head();
            }
            else{
                cout << "List is empty" << endl;
            }
        }
        else if(s=="getAt"){
            if(l1.head!=NULL){
                int i;
                cin >> i;
                if(i>=l1.size()){
                    cout << "Invalid arguments" << endl;
                }
                else{
                    l1.getAt(i);
                }
            }
            else

```

```

        {
            cout<<"List is empty";
        }
    }
    else if(s=="display"){
        if(l1.head!=NULL){
            l1.print();
        }
        else{
            cout << endl;
        }
    }
    else if(s=="size"){
        if(l1.head!=NULL){
            cout << l1.size() << endl;
        }
    }
    else if(s=="addAt"){
        int val,i;
        cin >> i >> val;
        if(i>l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.addAt(i,val);
        }
    }
    else if(s == "removeLast"){
        if(l1.head!=NULL){
            l1.deletion_at_tail();
        }
        else{
            cout << "List is empty" << endl;
        }
    }
    else if(s == "removeAt"){
        if(l1.head!=NULL){
            int i;
            cin >> i;
            if(i>l1.size()){
                cout << "Invalid arguments" << endl;
            }
            else{
                l1.removeAt(i);
            }
        }
        else{
            cout << "List is empty" << endl;
        }
    }
    else if(s == "reverseDI"){
        if(l1.head!=NULL){

```

```

        l1.reverse_di();
    }
    else{
        cout << "List is empty" <<endl;
    }
}
cin >> s;
}

}

```

## Reverse Linked List (pointer Iterative)

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line.  
All elements are separated by space
  - 2.3. size - Returns the number of elements in the linked list.
  - 2.4. removeFirst - Removes the first element from Linked List.
  - 2.5. getFirst - Returns the data of first element.
  - 2.6. getLast - Returns the data of last element.
  - 2.7. getAt - Returns the data of element available at the index passed.
  - 2.8. addFirst - adds a new element with given value in front of linked list.
  - 2.9. addAt - adds a new element at a given index.
  - 2.10. removeLast - removes the last element of linked list.
  - 2.11. removeAt - remove an element at a given index
3. You are required to complete the body of reversePI function. The function should be an iterative function and should reverse the contents of linked list by changing the "next" property of nodes.
4. Input and Output is managed for you.

### Constraints

None

### Format

#### Input

Input is managed for you

#### Output

Output is managed for you

### Example

#### Sample Input

```

addFirst 10
addFirst 20
addLast 30
addLast 40
addLast 50
addFirst 60
removeAt 2
display
reversePI
display
quit

```

### Sample Output

```
60 20 30 40 50
50 40 30 20 60
```

```
#include <iostream>
using namespace std;
class node{
    public:
    int val;
    node* next;

};
class LinkedList {
    public: node* head=nullptr;
    node* tail=nullptr;
    int size_=0;

void insert_at_tail(int val){
    if(head==NULL){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        head=newnode;

    }
    else{
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        node *temp = head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next = newnode;
    }

}

void insertion_at_head(int val){
    node *newnode = new node;
    newnode->val=val;
    newnode->next=NULL;
    if(head==NULL){
        head=newnode;
    }
    else{
        newnode ->next =head;
        head = newnode;
    }

}

void print (){
    node *temp =head;
```

```

    if(head==NULL){
        cout << "0" << endl;
        return;
    }
    while(temp!=NULL){
        cout<<temp->val<<" ";
        temp=temp->next;
    }
    cout << endl;
}
void deletion_at_head(){
    if(head==NULL) return;
    node *temp=head;
    head=head->next;
    delete temp;
}
void deletion_at_tail(){
    if(head==NULL) return;
    node* previous=NULL;
    node* temp=head;
    while(temp->next!=NULL){
        previous=temp;
        temp=temp->next;
    }
    previous->next = NULL;
    delete temp;
}

void last(){
    node* temp=head;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    cout << temp->val << endl;
}
int size(){
    int cnt=0;
    node* temp=head;
    while(temp!=NULL){
        temp=temp->next;
        cnt++;
    }
    return cnt;
}
void first(){
    cout << head->val << endl;
}

node* getAt(int p){
    int cnt=0;

```

```

    node* temp=head;
    while(cnt < p){
        cnt++;
        temp=temp->next;
    }
    cout << temp->val << endl;
    return temp;
}

void addAt(int pos,int data){
    if(pos==0){
        insertion_at_head(data);
        return;
    }
    node* newnode=new node;
    newnode->val=data;
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    newnode->next=temp->next;
    temp->next=newnode;
}

void removeAt(int pos){
    if(pos==0){
        deletion_at_head();
        return;
    }
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    temp->next = temp->next->next;
}

void reverse_di(){
    int left = 0;
    int right = size() - 1;
    while(left < right){
        node* templ = getAt( left);
        node* tempr = getAt( right);

        int temp = templ->val;
        templ->val = tempr->val;
        tempr->val = temp;
    }
}

```

```

        left++;
        right--;
    }
}

//this is the part of the answer
private:
node* get_node_at(int indx){
    node* c = head;
    for(int i {}; i<indx; i++){
        c = c->next;
    }
    return c;
}
public:
void reverse_pi(){

    if(size() == 0) {
        cout<<"List is empty"<<endl;
    }else{
        // int i = size()-1;
        // node *for_end = get_node_at(size() - 1);
        // while(i != 0){
        //     node* at_i = get_node_at(i); //get_node_at() is
written just above this function
        //     node* prev_i = get_node_at(i - 1);
        //     at_i->next = prev_i;

        //     i--;
        // }
        // node* at_i = get_node_at(0);
        // at_i->next = NULL;
        // tail = head;

        // head = for_end; //here i can't use get_node_at() because
        // //now all the link have been reverse so
cant travell from head to last element

```

```

//OR LITTLE SIMPLE
node* for_tail = head;
while (for_tail->next != NULL) {
    for_tail = for_tail->next;
}

```

```

node *c = head;
node *p = NULL;
while (c != NULL) {
    node* nxt = c->next;
    c->next = p;

```

```

        p = c;
        c = nxt;
    }
    node* temp = head;
    head = for_tail;
    tail = temp;

}

};

int main() {
    LinkedList l1;
    string s;
    cin >> s;
    while(s!="quit"){
        if(s=="addLast"){
            int data;
            cin>> data;
            l1.insert_at_tail(data);
        }
        else if(s=="addFirst"){
            int data;
            cin>> data;
            l1.insertion_at_head(data);
        }
        else if(s=="getFirst"){
            if(l1.head!=NULL){
                l1.first();
            }else{
                cout << "List is empty" << endl;
            }
        }
        else if(s=="getLast"){
            if(l1.head!=NULL){
                l1.last();
            }
            else
            {
                cout<<"List is empty";
            }
        }
        else if(s=="removeFirst"){
            if(l1.head!=NULL){
                l1.deletion_at_head();
            }
            else{
                cout << "List is empty" << endl;
            }
        }
    }
}

```



```

else if(s=="getAt"){
    if(l1.head!=NULL){
        int i;
        cin >> i;
        if(i>=l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.getAt(i);
        }
    }
    else
    {
        cout<<"List is empty";
    }
}
else if(s=="display"){
    if(l1.head!=NULL){
        l1.print();
    }
    else{
        cout << endl;
    }
}
else if(s=="size"){
    if(l1.head!=NULL){
        cout << l1.size() << endl;
    }
}
else if(s=="addAt"){
    int val,i;
    cin >> i >> val;
    if(i>l1.size()){
        cout << "Invalid arguments" << endl;
    }
    else{
        l1.addAt(i,val);
    }
}
else if(s == "removeLast"){
    if(l1.head!=NULL){
        l1.deletion_at_tail();
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s == "removeAt"){
    if(l1.head!=NULL){
        int i;

```

```

        cin >> i;
        if(i>l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.removeAt(i);
        }
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s == "reverseDI"){
    if(l1.head!=NULL){
        l1.reverse_di();
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s == "reversePI"){
    if(l1.head!=NULL){
        l1.reverse_pi();
    }
    else{
        cout << "List is empty" << endl;
    }
}
}
cin >> s;
}
}

```

## Linked List To Stack Adapter

Easy

1. You are required to complete the code of our LLToStackAdapter class.
2. As data members, you've a linkedlist available in the class.
3. Here is the list of functions that you are supposed to complete
  - 3.1. push -> Should accept new data in LIFO manner
  - 3.2. pop -> Should remove and return data in LIFO manner. If not available, print "Stack underflow" and return -1.
  - 3.3. top -> Should return data in LIFO manner. If not available, print "Stack underflow" and return -1.
  - 3.4. size -> Should return the number of elements available in the stack
4. Input and Output is managed for you.

Note -> The intention is to use linked list functions to achieve the purpose of a stack. All the functions should work in constant time.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
push 10
push 20
push 5
push 8
push 2
push 4
push 11
top
size
pop
top
size
pop
top
size
pop
top
size
pop
top
size
pop
top
size
pop
top
size
pop
quit
```

### Sample Output

```
11
7
11
4
6
4
2
5
2
8
4
8
5
3
```

```

5
20
2
20
10
1
10
#include <iostream>
using namespace std;
class node{
    public:
    int val;
    node* next;

};

class LinkedList {
    public:
    node* head=NULLptr;
    node* tail=NULLptr;
    int size_=0;

    void insert_at_tail(int val){
        if(head==NULL){
            node *newnode = new node;
            newnode->val=val;
            newnode->next=NULL;
            head=newnode;

        }
        else{
            node *newnode = new node;
            newnode->val=val;
            newnode->next=NULL;
            node *temp = head;
            while(temp->next!=NULL){
                temp=temp->next;
            }
            temp->next = newnode;
        }
    }

    void insertion_at_head(int val){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        if(head==NULL){
            head=newnode;
        }
        else{

```

```

        newnode ->next =head;
        head = newnode;
    }

}

void print (){
    node *temp =head;
    if(head==NULL){
        cout << "0" << endl;
        return;
    }
    while(temp!=NULL){
        cout<<temp->val<<" ";
        temp=temp->next;
    }
    cout << endl;
}

void deletion_at_head(){
    if(head==NULL) return;
    node *temp=head;
    head=head->next;
    delete temp;
}

void deletion_at_tail(){
    if(head==NULL) return;
    node* previous=NULL;
    node* temp=head;
    while(temp->next!=NULL){
        previous=temp;
        temp=temp->next;
    }
    previous->next = NULL;
    delete temp;
}

void last(){
    node* temp=head;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    cout << temp->val << endl;
}

int size(){
    int cnt=0;
    node* temp=head;
    while(temp!=NULL){
        temp=temp->next;
        cnt++;
    }
    return cnt;
}

```

```

}
int first(){
    return head->val ;
}

node* getAt(int p){
    int cnt=0;
    node* temp=head;
    while(cnt < p){
        cnt++;
        temp=temp->next;
    }
    cout << temp->val << endl;
    return temp;
}

void addAt(int pos,int data){
    if(pos==0){
        insertion_at_head(data);
        return;
    }
    node* newnode=new node;
    newnode->val=data;
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    newnode->next=temp->next;
    temp->next=newnode;

}

void removeAt(int pos){
    if(pos==0){
        deletion_at_head();
        return;
    }
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    temp->next = temp->next->next;
}

void reverse_di(){
    int left = 0;
    int right = size() - 1;

```

```

while(left < right){
    node* templ = getAt( left);
    node* tempr = getAt( right);

    int temp = templ->val;
    templ->val = tempr->val;
    tempr->val = temp;
    left++;
    right--;
}
}
void reverse_pi(){
    if(size()<=1){
        return;
    }

    node* t=head;
    while(t->next!=NULL){
        t=t->next;
    }
    tail=t;

    node* prev=nullptr;
    node* curr=head;
    while(curr !=nullptr){
        node* next= curr->next;
        curr->next=prev;
        prev=curr;
        curr=next;
    }
    node* temp=head;

    head =tail;
    tail=temp;
}
};

```

```

class LLToStackAdapter{
public:
    LinkedList l1;

    int size1(){
        // write your code here
        return l1.size();
    }
    void push(int val){
        // write your code here
        l1.insertion_at_head(val);
    }
    int pop(){
        // write your code here
    }
}

```

```

    if(l1.size() == 0) {
        cout<<"Stack underflow"<<endl;
        return -1;
    }else{
        int v = (l1.head)->val; // we can write these function for
the end but its complexity will be O(n) cause the implementaion of
the function we wrote earliar
        l1.deletion_at_head();
        return v;
    }
}
int top(){
    // write your code here
    if(l1.size() == 0) {
        cout<<"Stack underflow"<<endl;
        return -1;
    }else{
        int v = l1.first(); // we can write these function for the
end but its complexity will be O(n) cause the implementaion of the
function we wrote earliar
        return v;
    }
}
};
int main() {
    LLToStackAdapter l1;
    string s;
    cin >> s;
    while(s!="quit"){
        if(s == "push"){
            int val;
            cin >> val;
            l1.push(val);
        }
        else if(s == "pop"){
            int val= l1.pop();
            if (val != -1){
                cout << val << endl;
            }
        }
        else if(s== "top"){
            int val= l1.top();
            if(val != -1){
                cout << val << endl;
            }
        }
        }else if(s== "size"){
            cout << l1.size1() << endl;
        }
        cin >> s;
    }
}

```



# Linked List To Queue Adapter

Easy

1. You are required to complete the code of our LLToQueueAdapter class.
2. As data members, you've a linkedlist available in the class.
3. Here is the list of functions that you are supposed to complete
  - 3.1. add -> Should accept new data in FIFO manner
  - 3.2. remove -> Should remove and return data in FIFO manner. If not available, print "Queue underflow" and return -1.
  - 3.3. peek -> Should return data in FIFO manner. If not available, print "Queue underflow" and return -1.
  - 3.4. size -> Should return the number of elements available in the queue
4. Input and Output is managed for you.

Note -> The intention is to use linked list functions to achieve the purpose of a queue. All the functions should work in constant time.

## Constraints

None

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
add 10
add 20
add 30
add 40
add 50
add 60
peek
remove
peek
remove
peek
remove
peek
remove
peek
remove
peek
remove
quit
```

### Sample Output

```
10
10
20
20
30
```

```

30
40
40
50
50
60
60
#include <iostream>
using namespace std;
class node{
    public:
    int val;
    node* next;

};

class LinkedList {
    public:
    node* head=NULLptr;
    node* tail=NULLptr;
    int size_=0;

void insert_at_tail(int val){
    if(head==NULL){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        head=newnode;

    }
    else{
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        node *temp = head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next = newnode;
    }

}

void insertion_at_head(int val){
    node *newnode = new node;
    newnode->val=val;
    newnode->next=NULL;
    if(head==NULL){
        head=newnode;
    }
    else{

```

```

        newnode ->next =head;
        head = newnode;
    }
}
void print (){
    node *temp =head;
    if(head==NULL){
        cout << "0" << endl;
        return;
    }
    while(temp!=NULL){
        cout<<temp->val<<" ";
        temp=temp->next;
    }
    cout << endl;
}
int deletion_at_head(){
    if(head==NULL) return -1;
    int tr=head->val;
    node *temp=head;
    head=head->next;

    delete temp;
    return tr;
}
void deletion_at_tail(){
    if(head==NULL) return;
    node* previous=NULL;
    node* temp=head;
    while(temp->next!=NULL){
        previous=temp;
        temp=temp->next;
    }
    previous->next = NULL;
    delete temp;
}

void last(){
    node* temp=head;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    cout << temp->val << endl;
}
int size(){
    int cnt=0;
    node* temp=head;
    while(temp!=NULL){
        temp=temp->next;
    }
}

```

```

        cnt++;
    }
    return cnt;
}
int first(){
    return head->val ;
}

node* getAt(int p){
    int cnt=0;
    node* temp=head;
    while(cnt < p){
        cnt++;
        temp=temp->next;
    }
    cout << temp->val << endl;
    return temp;
}

void addAt(int pos,int data){
    if(pos==0){
        insertion_at_head(data);
        return;
    }
    node* newnode=new node;
    newnode->val=data;
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    newnode->next=temp->next;
    temp->next=newnode;
}

void removeAt(int pos){
    if(pos==0){
        deletion_at_head();
        return;
    }
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    temp->next = temp->next->next;
}

void reverse_di(){

```

```

int left = 0;
int right = size() - 1;
while(left < right){
    node* templ = getAt( left);
    node* tempr = getAt( right);

    int temp = templ->val;
    templ->val = tempr->val;
    tempr->val = temp;
    left++;
    right--;
}
}
void reverse_pi(){
    if(size()<=1){
        return;
    }

    node* t=head;
    while(t->next!=NULL){
        t=t->next;
    }
    tail=t;

    node* prev=nullptr;
    node* curr=head;
    while(curr !=nullptr){
        node* next= curr->next;
        curr->next=prev;
        prev=curr;
        curr=next;
    }
    node* temp=head;

    head =tail;
    tail=temp;
}
};

```

```

class LLToQueueAdapter{
public:
    LinkedList l1;

    int size1(){
        // write your code here
        return l1.size();
    }
    void add(int val){
        // write your code here
        l1.insert_at_tail(val);
    }
}

```

```

}
int Remove(){
    // write your code here
    if(l1.size() == 0 ) {
        cout<<"Queue underflow"<<endl;
        return -1;
    }else{
        int v = l1.first();
        l1.deletion_at_head();
        return v;
    }
}
int peek(){
    // write your code here
    if(l1.size() == 0 ) {
        cout<<"Queue underflow"<<endl;
        return -1;
    }else{
        int v = l1.first();
        return v;
    }
}
};
int main() {
    LLToQueueAdapter l1;
    string s;
    cin >> s;
    while(s!="quit"){
        if(s == "add"){
            int val;
            cin >> val;
            l1.add(val);
        }
        else if(s == "remove"){
            int val= l1.Remove();
            if (val != -1){
                cout << val << endl;
            }
        }
        else if(s== "peek"){
            int val= l1.peek();
            if(val != -1){
                cout << val << endl;
            }
        }else if(s== "size"){
            cout << l1.size1() << endl;
        }
        cin >> s;
    }
}

```

# Kth Node From End Of Linked List

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line.  
All elements are separated by space.
  - 2.3. size - Returns the number of elements in the linked list.
  - 2.4. removeFirst - Removes the first element from Linked List.
  - 2.5. getFirst - Returns the data of first element.
  - 2.6. getLast - Returns the data of last element.
  - 2.7. getAt - Returns the data of element available at the index passed.
  - 2.8. addFirst - adds a new element with given value in front of linked list.
  - 2.9. addAt - adds a new element at a given index.
  - 2.10. removeLast - removes the last element of linked list.
  - 2.11. removeAt - remove an element at a given index
3. You are required to complete the body of kthFromLast function. The function should be an iterative function and should return the kth node from end of linked list. Also, make sure to not use size data member directly or indirectly (by calculating size via making a traversal). k is a 0-based index. Assume that valid values of k will be passed.
4. Input and Output is managed for you.

## Constraints

1. Size property should not be used directly or indirectly
2. Constant time, single traversal is expected
3. Iterative solution, (not recursion) is expected

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addLast 10
getFirst
addLast 20
addLast 30
getFirst
getLast
getAt 1
addLast 40
kthFromEnd 3
getLast
addLast 50
removeFirst
getFirst
removeFirst
removeFirst
kthFromEnd 0
```

```
removeFirst  
removeFirst  
getFirst  
quit
```

### Sample Output

```
10  
10  
30  
20  
10  
40  
20  
50
```

```
List is empty
```

```
#include <iostream>  
using namespace std;  
class node{  
    public:  
    int val;  
    node* next;  
  
};  
class LinkedList {  
    public: node* head=nullptr;  
    node* tail=nullptr;  
    int size_=0;  
  
    void insert_at_tail(int val){  
        if(head==NULL){  
            node *newnode = new node;  
            newnode->val=val;  
            newnode->next=NULL;  
            head=newnode;  
  
        }  
        else{  
            node *newnode = new node;  
            newnode->val=val;  
            newnode->next=NULL;  
            node *temp = head;  
            while(temp->next!=NULL){  
                temp=temp->next;  
            }  
            temp->next = newnode;  
        }  
    }  
  
    void insertion_at_head(int val){  
        node *newnode = new node;  
        newnode->val=val;  
        newnode->next=NULL;
```



```

        if(head==NULL){
            head=newnode;
        }
        else{
            newnode ->next =head;
            head = newnode;
        }
    }
    void print (){
        node *temp =head;
        if(head==NULL){
            cout << 0 << endl;
            return;
        }
        while(temp!=NULL){
            cout<<temp->val<<" ";
            temp=temp->next;
        }
        cout << endl;
    }
    void deletion_at_head(){
        if(head==NULL) return;
        node *temp=head;
        head=head->next;
        delete temp;
    }
    void deletion_at_tail(){
        if(head==NULL) return;
        node* previous=NULL;
        node* temp=head;
        while(temp->next!=NULL){
            previous=temp;
            temp=temp->next;
        }
        previous->next = NULL;
        delete temp;
    }

    void last(){
        node* temp=head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        cout << temp->val << endl;
    }
    int size(){
        int cnt=0;
        node* temp=head;
        while(temp!=NULL){

```

```

        temp=temp->next;
        cnt++;
    }
    return cnt;
}
void first(){
    cout << head->val << endl;
}

node* getAt(int p){
    int cnt=0;
    node* temp=head;
    while(cnt < p){
        cnt++;
        temp=temp->next;
    }
    // cout << temp->val << endl;
    return temp;
}

void addAt(int pos,int data){
    if(pos==0){
        insertion_at_head(data);
        return;
    }
    node* newnode=new node;
    newnode->val=data;
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    newnode->next=temp->next;
    temp->next=newnode;
}

void removeAt(int pos){
    if(pos==0){
        deletion_at_head();
        return;
    }
    int cnt=0;
    node* temp=head;
    while(cnt<pos-1){
        cnt++;
        temp=temp->next;
    }
    temp->next = temp->next->next;
}

```

```

void reverse_di(){
    int left = 0;
    int right = size() - 1;
    while(left < right){
        node* templ = getAt( left);
        node* tempr = getAt( right);

        int temp = templ->val;
        templ->val = tempr->val;
        tempr->val = temp;
        left++;
        right--;
    }
}

int kthFromEnd(int k){
    // write your code here
    // int i_from_starting =
    node* c1 = head;
    for(int i{}; i < k;i++) {
        c1 = c1->next;
    }
    node* c2 = head;
    while(c1->next != NULL){
        c1 = c1->next;
        c2 = c2->next;
    }

    return c2->val;
}
};

int main() {
    LinkedList l1;
    string s;
    cin >> s;
    while(s!="quit"){
        if(s=="addLast"){
            int data;
            cin>> data;
            l1.insert_at_tail(data);
        }
        else if(s=="addFirst"){
            int data;
            cin>> data;
            l1.insertion_at_head(data);
        }
        else if(s=="getFirst"){
            if(l1.head!=NULL){
                l1.first();
            }else{
                cout << "List is empty" << endl;
            }
        }
    }
}

```

```

else if(s=="getLast"){
    if(l1.head!=NULL){
        l1.last();
    }
    else
    {
        cout<<"List is empty";
    }
}
else if(s=="removeFirst"){
    if(l1.head!=NULL){
        l1.deletion_at_head();
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s=="getAt"){
    if(l1.head!=NULL){
        int i;
        cin >> i;
        if(i>=l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            cout<<(l1.getAt(i))->val<<endl;
        }
    }
    else
    {
        cout<<"List is empty";
    }
}
else if(s=="display"){
    if(l1.head!=NULL){
        l1.print();
    }
    else{
        cout << endl;
    }
}
else if(s=="size"){
    if(l1.head!=NULL){
        cout << l1.size() << endl;
    }
}
else if(s=="addAt"){
    int val,i;
    cin >> i >> val;
    if(i>l1.size()){

```

```

        cout << "Invalid arguments" << endl;
    }
    else{
        l1.addAt(i,val);
    }
}
else if(s == "removeLast"){
    if(l1.head!=NULL){
        l1.deletion_at_tail();
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s == "removeAt"){
    if(l1.head!=NULL){
        int i;
        cin >> i;
        if(i>l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.removeAt(i);
        }
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s == "reverseDI"){
    if(l1.head!=NULL){
        l1.reverse_di();
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s=="kthFromEnd"){
    int i;
    cin>>i;

    if(i>=l1.size()){
        cout << "Invalid arguments" << endl;
    }
    else{
        cout<<l1.kthFromEnd(i)<< endl;
    }
}
cin >> s;
}
}

```

# Mid Of Linked List

Easy

1. You are given a partially written LinkedList class.
2. Here is a list of existing functions:
  - 2.1 addLast - adds a new element with given value to the end of Linked List
  - 2.2. display - Prints the elements of linked list from front to end in a single line.  
All elements are separated by space
  - 2.3. size - Returns the number of elements in the linked list.
  - 2.4. removeFirst - Removes the first element from Linked List.
  - 2.5. getFirst - Returns the data of first element.
  - 2.6. getLast - Returns the data of last element.
  - 2.7. getAt - Returns the data of element available at the index passed.
  - 2.8. addFirst - adds a new element with given value in front of linked list.
  - 2.9. addAt - adds a new element at a given index.
  - 2.10. removeLast - removes the last element of linked list.
  - 2.11. removeAt - remove an element at a given index
  - 2.12 kthFromLast - return kth node from end of linked list.
3. You are required to complete the body of mid function. The function should be an iterative function and should return the mid of linked list. Also, make sure to not use size data member directly or indirectly (by calculating size via making a traversal). In linked list of odd size, mid is unambiguous. In linked list of even size, consider end of first half as mid.
4. Input and Output is managed for you.

## Constraints

1. Size property should not be used directly or indirectly
2. Constant time, single traversal is expected
3. Iterative solution, (not recursion) is expected.

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
addLast 10
getFirst
addLast 20
addLast 30
getFirst
getLast
getAt 1
addLast 40
mid
getLast
addLast 50
removeFirst
getFirst
removeFirst
removeFirst
```

```
mid
removeFirst
removeFirst
getFirst
quit
```

### Sample Output

```
10
10
30
20
20
40
20
40
```

```
List is empty
```

```
#include <iostream>
using namespace std;
class node{
public:
    int val;
    node* next;

};
class LinkedList {
public: node* head=NULLptr;
    node* tail=NULLptr;
    int size_=0;

void insert_at_tail(int val){
    if(head==NULL){
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        head=newnode;

    }
    else{
        node *newnode = new node;
        newnode->val=val;
        newnode->next=NULL;
        node *temp = head;
        while(temp->next!=NULL){
            temp=temp->next;
        }
        temp->next = newnode;
    }

}

void insertion_at_head(int val){
    node *newnode = new node;
    newnode->val=val;
```

```

newnode->next=NULL;
if(head==NULL){
    head=newnode;
}
else{
    newnode ->next =head;
    head = newnode;
}

}
void print (){
    node *temp =head;
    if(head==NULL){
        cout << 0 << endl;
        return;
    }
    while(temp!=NULL){
        cout<<temp->val<<" ";
        temp=temp->next;
    }
    cout << endl;
}
void deletion_at_head(){
    if(head==NULL) return;
    node *temp=head;
    head=head->next;
    delete temp;
}
void deletion_at_tail(){
    if(head==NULL) return;
    node* previous=NULL;
    node* temp=head;
    while(temp->next!=NULL){
        previous=temp;
        temp=temp->next;
    }
    previous->next = NULL;
    delete temp;
}

void last(){
    node* temp=head;
    while(temp->next!=NULL){
        temp=temp->next;
    }
    cout << temp->val << endl;
}
int size(){
    int cnt=0;
    node* temp=head;

```



```

        while(temp!=NULL){
            temp=temp->next;
            cnt++;
        }
        return cnt;
    }
    void first(){
        cout << head->val << endl;
    }

    node* getAt(int p){
        int cnt=0;
        node* temp=head;
        while(cnt < p){
            cnt++;
            temp=temp->next;
        }
        cout << temp->val << endl;
        return temp;
    }

    void addAt(int pos,int data){
        if(pos==0){
            insertion_at_head(data);
            return;
        }
        node* newnode=new node;
        newnode->val=data;
        int cnt=0;
        node* temp=head;
        while(cnt<pos-1){
            cnt++;
            temp=temp->next;
        }
        newnode->next=temp->next;
        temp->next=newnode;
    }

    void removeAt(int pos){
        if(pos==0){
            deletion_at_head();
            return;
        }
        int cnt=0;
        node* temp=head;
        while(cnt<pos-1){
            cnt++;
            temp=temp->next;
        }
        temp->next = temp->next->next;
    }

```

```

void reverse_di(){
    int left = 0;
    int right = size() - 1;
    while(left < right){
        node* templ = getAt( left);
        node* tempr = getAt( right);

        int temp = templ->val;
        templ->val = tempr->val;
        tempr->val = temp;
        left++;
        right--;
    }
}

int kthFromEnd(int k){
    if(head==nullptr)
    {
        cout<<"List is empty";
        return -1;
    }
    node *temp1=head;
    node *temp2=head;
    for(int i=0;i<k;i++){
        temp2=temp2->next;
    }

    while(temp2->next!=nullptr){
        temp2=temp2->next;
        temp1=temp1->next;
    }

    return temp1->val;
}

void mid(){
    // write your code here
    node * f = head->next;
    node * s = head;
    while(f != NULL && f->next != NULL ) {
        f = f->next;
        f = f->next;
        s = s->next;
    }
    cout<<s->val<<endl;
    return ;
}

};

int main() {
    LinkedList l1;
    string s;
    cin >> s;
    while(s!="quit"){

```

```

if(s=="addLast"){
    int data;
    cin>> data;
    l1.insert_at_tail(data);
}
else if(s=="addFirst"){
    int data;
    cin>> data;
    l1.insertion_at_head(data);
}
else if(s=="getFirst"){
    if(l1.head!=NULL){
        l1.first();
    }else{
        cout << "List is empty" << endl;
    }
}
else if(s=="getLast"){
    if(l1.head!=NULL){
        l1.last();
    }
    else
    {
        cout<<"List is empty";
    }
}
else if(s=="removeFirst"){
    if(l1.head!=NULL){
        l1.deletion_at_head();
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s=="getAt"){
    if(l1.head!=NULL){
        int i;
        cin >> i;
        if(i>=l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.getAt(i);
        }
    }
    else
    {
        cout<<"List is empty";
    }
}
}

```

```

else if(s=="display"){
    if(l1.head!=NULL){
        l1.print();
    }
    else{
        cout << endl;
    }
}
else if(s=="size"){
    if(l1.head!=NULL){
        cout << l1.size() << endl;
    }
}
else if(s=="addAt"){
    int val,i;
    cin >> i >> val;
    if(i>l1.size()){
        cout << "Invalid arguments" << endl;
    }
    else{
        l1.addAt(i,val);
    }
}
else if(s == "removeLast"){
    if(l1.head!=NULL){
        l1.deletion_at_tail();
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s == "removeAt"){
    if(l1.head!=NULL){
        int i;
        cin >> i;
        if(i>l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            l1.removeAt(i);
        }
    }
    else{
        cout << "List is empty" << endl;
    }
}
else if(s == "reverseDI"){
    if(l1.head!=NULL){
        l1.reverse_di();
    }
    else{
        cout << "List is empty" << endl;
    }
}

```

```

    }
    else if(s== "kthFromEnd"){
        int i;
        cin>>i;

        if(i>=l1.size()){
            cout << "Invalid arguments" << endl;
        }
        else{
            cout<<l1.kthFromEnd(i)<< endl;
        }
    }
    else if(s=="mid"){
        l1.mid();
    }
    cin >> s;
}
}

```

## Merge Two Sorted Linked Lists

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of mergeTwoSortedLists function. The function is static and is passed two lists which are sorted. The function is expected to return a new sorted list containing elements of both lists. Original lists must stay as they were.
3. Input and Output is managed for you.

### Constraints

1.  $O(n)$  time complexity and constant space complexity expected.

### Format

#### Input

Input is managed for you

#### Output

Output is managed for you

### Example

#### Sample Input

```

5
10 20 30 40 50
10
7 9 12 15 37 43 44 48 52 56

```

#### Sample Output

```

7 9 10 12 15 20 30 37 40 43 44 48 50 52 56
10 20 30 40 50
7 9 12 15 37 43 44 48 52 56

```

```

#include <iostream>
#include <string>

```

```

using namespace std;
class linkedlist
{

public:

    class Node
    {
    public:
        int data = 0;
        Node *next = nullptr;

        Node(int data)
        {
            this->data = data;
        }
    };

    Node *head = nullptr;
    Node *tail = nullptr;
    int size = 0;

    //basic->=====

    int size_()
    {
        return this->size;
    }

    bool isEmpty()
    {
        return this->size == 0;
    }

    string toString()
    {
        Node *curr = this->head;
        string sb = "";

        while (curr != nullptr)
        {
            sb += to_string(curr->data);
            if (curr->next != nullptr)
                sb += " ";
            curr = curr->next;
        }
        return sb;
    }

    //add->=====

```

```

private:
    void addFirstNode(Node *node)
    {
        if (this->head == nullptr)
        {
            this->head = node;
            this->tail = node;
        }
        else
        {
            node->next = this->head;
            this->head = node;
        }

        this->size++;
    }

public:
    void addFirst(int val)
    {
        Node *node = new Node(val);
        addFirstNode(node);
    }

public:
    void addLastNode(Node *node)
    {
        if (this->head == nullptr)
        {
            this->head = node;
            this->tail = node;
        }
        else
        {
            this->tail->next = node;
            this->tail = node;
        }

        this->size++;
    }

    void addLast(int val)
    {
        Node *node = new Node(val);
        addLastNode(node);
    }

    void addNodeAt(Node *node, int idx)
    {
        if (idx == 0)
            addFirstNode(node);
        else if (idx == this->size)

```

```

        addLastNode(node);
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = node;
        curr->next = node;

        this->size++;
    }
}

void addAt(int data, int idx)
{
    if (idx < 0 || idx > this->size)
    {
        throw("invalidLocation: " + to_string(idx));
    }

    Node *node = new Node(data);
    addNodeAt(node, idx);
}

//remove->=====
Node *removeFirstNode()
{
    Node *node = this->head;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        this->head = this->head->next;
        node->next = nullptr;
    }

    this->size--;
    return node;
}

int removeFirst(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = removeFirstNode();
    int rv = node->data;

```



```

        delete node;
        return rv;
    }

Node *removeLastNode()
{
    Node *node = this->tail;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        Node *prev = getNodeAt(this->size - 2);
        this->tail = prev;
        prev->next = nullptr;
    }

    this->size--;
    return node;
}

int removeLast(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = new Node(val);
    int rv = node->data;
    delete node;
    return rv;
}

Node *removeNodeAt(int idx)
{
    if (idx == 0)
        return removeFirstNode();
    else if (idx == this->size - 1)
        return removeLastNode();
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = curr->next;
        curr->next = nullptr;

        this->size--;
    }
}

```

```

        return curr;
    }
}

int removeAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = removeNodeAt(idx);
    int rv = node->data;
    delete node;
    return rv;
}

//get->=====
Node *getFirstNode()
{
    return this->head;
}

int getFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = getFirstNode();
    return node->data;
}

Node *getLastNode()
{
    return this->tail;
}

int getLast()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException");
    }

    Node *node = getLastNode();
    return node->data;
}

Node *getNodeAt(int idx)
{

```

```

Node *curr = this->head;

while (idx-- > 0)
{
    curr = curr->next;
}

return curr;
}

int getAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = getNodeAt(idx);
    return node->data;
}

//merge two sorted linkedlist
void mergeTwoSortedLists(linkedlist l1, linkedlist l2) {
    // write your code here
    Node* c1 = l1.head;
    Node* c2 = l2.head;

    while(c1 != NULL && c2 != NULL) {
        if((c1->data) < (c2->data)) {
            Node *nn = new Node(c1->data);
            c1 = c1->next;
            addLastNode(nn);
        }else{
            Node *nn = new Node(c2->data);
            c2 = c2->next;
            addLastNode(nn);
        }
    }
    while(c1 != NULL ) {
        Node *nn = new Node(c1->data);
        c1 = c1->next;
        addLastNode(nn);
    }
    while(c2 != NULL ) {
        Node *nn = new Node(c2->data);
        c2 = c2->next;
        addLastNode(nn);
    }
}

```

```

    }
};

int main()
{
    linkedlist l1;
    linkedlist l2;

    int n1;
    cin>>n1;

    for (int i = 0; i <n1; i++)
    {
        int val;
        cin>>val;
        l1.addLast(val);
    }

    int n2;
    cin>>n2;
    for (int i = 0; i<n2; i++)
    {
        int val;
        cin>>val;
        l2.addLast(val);
    }

    linkedlist merged;
    merged.mergeTwoSortedLists(l1, l2);

    cout << merged.toString()<<" " << endl;
    cout << l1.toString()<<" " << endl;
    cout << l2.toString() <<" " << endl;

    return 0;
}

```

## Merge Sort A Linked List

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of mergeSort function. The function is static and is passed the head and tail of an unsorted list. The function is expected to return a new sorted list. The original list must not change.
3. Input and Output is managed for you.

Note - Watch the question video for theory of merge sort.

## Constraints

1.  $O(n \log n)$  time complexity required.

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
6
10 2 19 22 3 7
```

### Sample Output

```
2 3 7 10 19 22
10 2 19 22 3 7
```

```
#include <iostream>
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Node
{
public:
    int data = 0;
    Node *next = nullptr;

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

```
class linkedlist
{
```

```
public:
```

```
    Node *head = nullptr;
    Node *tail = nullptr;
    int size = 0;
```

```
//basic->=====
```

```
    int size_()
    {
        return this->size;
    }
```

```
    bool isEmpty()
    {
```

```

        return this->size == 0;
    }

    string toString()
    {
        Node *curr = this->head;
        string sb = "";

        while (curr != nullptr)
        {
            sb += to_string(curr->data);
            if (curr->next != nullptr)
                sb += " ";
            curr = curr->next;
        }
        sb += " "; //me don't know
        return sb;
    }

    //add->=====
private:
    void addFirstNode(Node *node)
    {
        if (this->head == nullptr)
        {
            this->head = node;
            this->tail = node;
        }
        else
        {
            node->next = this->head;
            this->head = node;
        }

        this->size++;
    }

public:
    void addFirst(int val)
    {
        Node *node = new Node(val);
        addFirstNode(node);
    }

public:
    void addLastNode(Node *node)
    {
        if (this->head == nullptr)
        {
            this->head = node;
            this->tail = node;
        }
    }

```

```

        else
        {
            this->tail->next = node;
            this->tail = node;
        }

        this->size++;
    }

    void addLast(int val)
    {
        Node *node = new Node(val);
        addLastNode(node);
    }

    void addNodeAt(Node *node, int idx)
    {
        if (idx == 0)
            addFirstNode(node);
        else if (idx == this->size)
            addLastNode(node);
        else
        {
            Node *prev = getNodeAt(idx - 1);
            Node *curr = prev->next;

            prev->next = node;
            curr->next = node;

            this->size++;
        }
    }

    void addAt(int data, int idx)
    {
        if (idx < 0 || idx > this->size)
        {
            throw("invalidLocation: " + to_string(idx));
        }

        Node *node = new Node(data);
        addNodeAt(node, idx);
    }

    //remove->=====
    Node *removeFirstNode()
    {
        Node *node = this->head;
        if (this->size == 1)
        {
            this->head = nullptr;
            this->tail = nullptr;
        }
    }

```

```

    }
    else
    {
        this->head = this->head->next;
        node->next = nullptr;
    }

    this->size--;
    return node;
}

int removeFirst(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = removeFirstNode();
    int rv = node->data;
    delete node;
    return rv;
}

Node *removeLastNode()
{
    Node *node = this->tail;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        Node *prev = getNodeAt(this->size - 2);
        this->tail = prev;
        prev->next = nullptr;
    }

    this->size--;
    return node;
}

int removeLast(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = new Node(val);
    int rv = node->data;

```



```

        delete node;
        return rv;
    }

Node *removeNodeAt(int idx)
{
    if (idx == 0)
        return removeFirstNode();
    else if (idx == this->size - 1)
        return removeLastNode();
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = curr->next;
        curr->next = nullptr;

        this->size--;
        return curr;
    }
}

int removeAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = removeNodeAt(idx);
    int rv = node->data;
    delete node;
    return rv;
}

//get->=====
Node *getFirstNode()
{
    return this->head;
}

int getFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = getFirstNode();
    return node->data;
}

```

```

}

Node *getLastNode()
{
    return this->tail;
}

int getLast()
{
    if (this->size == 0)
    {
        throw("nullPointerException");
    }

    Node *node = getLastNode();
    return node->data;
}

Node *getNodeAt(int idx)
{
    Node *curr = this->head;

    while (idx-- > 0)
    {
        curr = curr->next;
    }

    return curr;
}

int getAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = getNodeAt(idx);
    return node->data;
}
};

Node* getMid(Node* head, Node* tail){
    Node* slow = head, *fast = head;
    while(fast->next != tail && fast->next->next != tail){
        fast = fast->next->next;
        slow = slow->next;
    }

    return slow;
}

```

```

//merge two sorted linkedlist
linkedlist mergeTwoSortedLists(linkedlist l1, linkedlist l2) {
    linkedlist ans;
    Node* one = l1.head;
    Node* two = l2.head;

    while(one != nullptr && two != nullptr){
        if(one->data < two->data){
            ans.addLast(one->data);
            one = one->next;
        }else{
            ans.addLast(two->data);
            two = two->next;
        }
    }
    while(one!=nullptr){
        ans.addLast(one->data);
        one = one->next;
    }
    while(two !=nullptr){
        ans.addLast(two->data);
        two = two->next;
    }

    return ans;
}

linkedlist mergeSort(Node* head,Node* tail ){
    //write your code here
    int s{0};
    Node *c = head;
    while(c != tail){
        c = c->next;
        s++;
    }
    // c = c->next;
    s++;
    if (s == 1) {
        linkedlist a;
        a.addFirst(head->data);
        return a;
    }
    // cout<<"size "<<s<<endl;

    Node * t1 = head;
    Node * h2 = head;
    for(int i {} ; i < (s/2)-1 ;i++) {
        t1 = t1->next;
        h2 = h2->next;
    }
    h2 = h2->next;
}

```

```

        // cout<<t1->data <<endl;
        linkedlist x = mergeSort(head,t1);
        linkedlist y = mergeSort(h2,tail);
        linkedlist ans = mergeTwoSortedLists(x,y);

        return ans;

    }

linkedlist makeList() {
    linkedlist l;
    int n1;
    cin >> n1;

    for (int i = 0; i < n1; i++) {
        int val;
        cin >> val;
        l.addLast(val);
    }
    return l;
}

int main()
{
    linkedlist l = makeList();

    linkedlist sorted = mergeSort(l.head, l.tail);

    cout << sorted.toString() << endl;
    cout << l.toString() << endl;

    return 0;
}

```

## Remove Duplicates In A Sorted Linked List

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of removeDuplicates function. The function is called on a sorted list. The function must remove all duplicates from the list in linear time and constant space
3. Input and Output is managed for you.

### Constraints

1. Time complexity ->  $O(n)$
2. Space complexity -> constant

### Format

#### Input

Input is managed for you

## Output

Output is managed for you

## Example

### Sample Input

```
10
2 2 2 3 3 5 5 5 5 5
```

### Sample Output

```
2 2 2 3 3 5 5 5 5 5
2 3 5
```

```
#include <iostream>
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Node
{
public:
    int data = 0;
    Node* next = nullptr;

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

```
class linkedlist
{
```

```
public:
```

```
Node* head = nullptr;
Node* tail = nullptr;
int size = 0;
```

```
//basic->=====
```

```
int size_()
{
    return this->size;
}
```

```
bool isEmpty()
{
    return this->size == 0;
}
```

```
string toString()
{
```

```

Node* curr = this->head;
string sb = "";

while (curr != nullptr)
{
    sb += to_string(curr->data);
    if (curr->next != nullptr)
        sb += " ";
    curr = curr->next;
}
sb += " "; //me
return sb;
}

//add->=====
private:
void addFirstNode(Node* node)
{
    if (this->head == nullptr)
    {
        this->head = node;
        this->tail = node;
    }
    else
    {
        node->next = this->head;
        this->head = node;
    }

    this->size++;
}

public:
void addFirst(int val)
{
    Node* node = new Node(val);
    addFirstNode(node);
}

public:
void addLastNode(Node* node)
{
    if (this->head == nullptr)
    {
        this->head = node;
        this->tail = node;
    }
    else
    {
        this->tail->next = node;
        this->tail = node;
    }
}

```

```

        this->size++;
    }

    void addLast(int val)
    {
        Node* node = new Node(val);
        addLastNode(node);
    }

    void addNodeAt(Node* node, int idx)
    {
        if (idx == 0)
            addFirstNode(node);
        else if (idx == this->size)
            addLastNode(node);
        else
        {
            Node* prev = getNodeAt(idx - 1);
            Node* curr = prev->next;

            prev->next = node;
            curr->next = node;

            this->size++;
        }
    }

    void addAt(int data, int idx)
    {
        if (idx < 0 || idx > this->size)
        {
            throw ("invalidLocation: " + to_string(idx));
        }

        Node* node = new Node(data);
        addNodeAt(node, idx);
    }

    //remove->=====
    Node* removeFirstNode()
    {
        Node* node = this->head;
        if (this->size == 1)
        {
            this->head = nullptr;
            this->tail = nullptr;
        }
        else
        {
            this->head = this->head->next;
            node->next = nullptr;
        }
    }

```

```

    }

    this->size--;
    return node;
}

void removeFirst()
{
    if (this->size == 0)
    {
        throw ("nullptrPointerException: -1");
    }

    Node* node = removeFirstNode();
    int rv = node->data;
    delete node;
}

Node* removeLastNode()
{
    Node* node = this->tail;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        Node* prev = getNodeAt(this->size - 2);
        this->tail = prev;
        prev->next = nullptr;
    }

    this->size--;
    return node;
}

int removeLast(int val)
{
    if (this->size == 0)
    {
        throw ("nullptrPointerException: -1");
    }

    Node* node = new Node(val);
    int rv = node->data;
    delete node;
    return rv;
}

Node* removeNodeAt(int idx)

```



```

{
    if (idx == 0)
        return removeFirstNode();
    else if (idx == this->size - 1)
        return removeLastNode();
    else
    {
        Node* prev = getNodeAt(idx - 1);
        Node* curr = prev->next;

        prev->next = curr->next;
        curr->next = nullptr;

        this->size--;
        return curr;
    }
}

int removeAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw ("invalidLocation: " + idx);
    }

    Node* node = removeNodeAt(idx);
    int rv = node->data;
    delete node;
    return rv;
}

//get->=====
Node* getFirstNode()
{
    return this->head;
}

int getFirst()
{
    if (this->size == 0)
    {
        throw ("nullptrPointerException: -1");
    }

    Node* node = getFirstNode();
    return node->data;
}

Node* getLastNode()
{
    return this->tail;
}

```

```

}

int getLast()
{
    if (this->size == 0)
    {
        throw ("NullPointerException");
    }

    Node* node = getLastNode();
    return node->data;
}

Node* getNodeAt(int idx)
{
    Node* curr = this->head;

    while (idx-- > 0)
    {
        curr = curr->next;
    }

    return curr;
}

int getAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw ("invalidLocation: " + idx);
    }

    Node* node = getNodeAt(idx);
    return node->data;
}

public:
void removeDuplicates() {
    //write your code here

    //sir solution BUT for this -> tail should be managed properly

    // linkedlist *ans = new linkedlist{};
    // while(this->size_() > 0) {
    //     int a = this->getFirst();
    //     removeFirst();
    //     if(ans->size_() == 0 || ans->tail->data != a) {
    //         ans->addLast(a);
    //     }
    // }
    // // delete a;
    // this->head = ans->head;
    // this->tail = ans->tail;

```

```

//    this->size = ans->size;

//my solution
Node * c = head;
Node * p = NULL;
while(c->next != NULL) {
    p = c;
    c = c->next;
    if(p->data == c->data){
        p->next = c->next;
        Node * tdel = c;
        c = p;
        delete tdel;
    }
}
};

linkedList makeList() {
    linkedlist l;
    int n1;
    cin >> n1;

    for (int i = 0; i < n1; i++) {
        int val;
        cin >> val;
        l.addLast(val);
    }
    return l;
}

int main()
{
    linkedlist l = makeList();

    cout << l.toString() << endl;
    l.removeDuplicates();
    cout << l.toString() << endl;

    return 0;
}

```

## Odd Even Linked List

Medium

1. You are given a partially written LinkedList class.
2. You are required to complete the body of oddEven function. The function is expected to tweak the list such that all odd values are followed by all even values. The relative order of elements should not change. Also, take care of the cases when there are no odd or no even elements. Make sure to properly set head, tail and size as the function will be tested by calling addFirst and addLast.
3. Input and Output is managed for you.

## Constraints

1. Time complexity  $\rightarrow O(n)$
2. Space complexity  $\rightarrow$  constant

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
7
2 8 9 1 5 4 3
10
100
```

### Sample Output

```
2 8 9 1 5 4 3
9 1 5 3 2 8 4
10 9 1 5 3 2 8 4 100
```

```
#include <iostream>
// #include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Node
{
public:
    int data = 0;
    Node *next = nullptr;

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

```
class linkedlist
{
```

```
public:
```

```
    Node *head = nullptr;
    Node *tail = nullptr;
    int size = 0;
```

```
//basic->=====
```

```
int size_()
{
    return this->size;
}

bool isEmpty()
{
    return this->size == 0;
}

string toString()
{
    Node *curr = this->head;
    string sb = "";

    while (curr != nullptr)
    {
        sb += to_string(curr->data);
        if (curr->next != nullptr)
            sb += " ";
        curr = curr->next;
    }
    sb += " ";
    return sb;
}
```

```
//add->=====
```

```
private:
void addFirstNode(Node *node)
{
    if (this->head == nullptr)
    {
        this->head = node;
        this->tail = node;
    }
    else
    {
        node->next = this->head;
        this->head = node;
    }

    this->size++;
}
```

```
public:
void addFirst(int val)
{
    Node *node = new Node(val);
    addFirstNode(node);
}
```

```
}
```

```
public:
```

```
void addLastNode(Node *node)
```

```
{
```

```
    if (this->head == nullptr)
```

```
    {
```

```
        this->head = node;
```

```
        this->tail = node;
```

```
    }
```

```
    else
```

```
    {
```

```
        this->tail->next = node;
```

```
        this->tail = node;
```

```
    }
```

```
    this->size++;
```

```
}
```

```
void addLast(int val)
```

```
{
```

```
    Node *node = new Node(val);
```

```
    addLastNode(node);
```

```
}
```

```
void addNodeAt(Node *node, int idx)
```

```
{
```

```
    if (idx == 0)
```

```
        addFirstNode(node);
```

```
    else if (idx == this->size)
```

```
        addLastNode(node);
```

```
    else
```

```
    {
```

```
        Node *prev = getNodeAt(idx - 1);
```

```
        Node *curr = prev->next;
```

```
        prev->next = node;
```

```
        curr->next = node;
```

```
        this->size++;
```

```
    }
```

```
}
```

```
void addAt(int data, int idx)
```

```
{
```

```
    if (idx < 0 || idx > this->size)
```

```
    {
```

```
        throw("invalidLocation: " + to_string(idx));
```

```
    }
```

```
    Node *node = new Node(data);
```

```
    addNodeAt(node, idx);
```

```

}

//remove->=====
Node *removeFirstNode()
{
    Node *node = this->head;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        this->head = this->head->next;
        node->next = nullptr;
    }

    this->size--;
    return node;
}

void removeFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = removeFirstNode();
    int rv = node->data;
    delete node;
}

Node *removeLastNode()
{
    Node *node = this->tail;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        Node *prev = getNodeAt(this->size - 2);
        this->tail = prev;
        prev->next = nullptr;
    }

    this->size--;
    return node;
}

```

```

int removeLast(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = new Node(val);
    int rv = node->data;
    delete node;
    return rv;
}

Node *removeNodeAt(int idx)
{
    if (idx == 0)
        return removeFirstNode();
    else if (idx == this->size - 1)
        return removeLastNode();
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = curr->next;
        curr->next = nullptr;

        this->size--;
        return curr;
    }
}

int removeAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = removeNodeAt(idx);
    int rv = node->data;
    delete node;
    return rv;
}

//get->=====
Node *getFirstNode()
{
    return this->head;
}

```



```

int getFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = getFirstNode();
    return node->data;
}

Node *getLastNode()
{
    return this->tail;
}

int getLast()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException");
    }

    Node *node = getLastNode();
    return node->data;
}

Node *getNodeAt(int idx)
{
    Node *curr = this->head;

    while (idx-- > 0)
    {
        curr = curr->next;
    }

    return curr;
}

int getAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = getNodeAt(idx);
    return node->data;
}

public:

```

```

void oddEven(){
    //write your code here

    linkedlist * odd = new linkedlist{};
    linkedlist * even = new linkedlist{};

    while(size_() > 0){
        int a = getFirst();
        removeFirst();
        if(a%2 != 0) {
            odd->addLast(a);
        }else{
            even->addLast(a);
        }
    }

    if(even->size_() > 0 && odd->size_() > 0) {
        Node * c= odd->head;
        while(c->next != NULL){
            c = c->next;
        } //could have simply used tail of odd but...
        c->next = even->head;
        this->head = odd->head;
        this->tail = even->tail;
        this->size = odd->size + even->size;

    }else if (odd->size_() > 0) {
        this->head = odd->head;
        this->tail = odd->tail;
        this->size = odd->size;
    }else if (even->size_() > 0) {
        this->head = even->head;
        this->tail = even->tail;
        this->size = even->size;
    }

}

// int n = size_();
// Node * c= head;
// Node * p = NULL;
// linkedlist * ans = new linkedlist{};
// for(int i{};i < n;i++){
//     int a = c->data;
//     if(a % 2 != 0){
//         if(c == head){
//             c = c->next;

```

```

        //          this->removeFirst(); //if c== head then if
first element is removed than it's memory will also be get
deallocate
        //          ans->addLast(a);
        //          continue;
        //      }else{
        //          Node * tdel = c;
        //          p->next = c->next;
        //          size--;
        //          c = p;
        //          delete tdel;
        //      }
        //      ans->addLast(a);
        // // cout<<"size: "<<size_()<<endl;
        // }
        //      p = c;
        //      c = c->next;
        // }

        // n = size_();
        // for(int i{} ;i < n ;i++) {
        //     int a = getFirst();
        //     removeFirst();
        //     ans->addLast(a);
        // }
        // this->head = ans->head;
        // this->tail = ans->tail;
        // this->size = ans->size;

    }
};

```

```

linkedlist makeList() {
    linkedlist l;
    int n1;
    cin >> n1;

    for (int i = 0; i < n1; i++) {
        int val;
        cin >> val;
        l.addLast(val);
    }
    return l;
}

```

```

int main()
{
    linkedlist l = makeList();
    int a;
    cin >>a;
    int b;

```

```

        cin >> b;

        cout<<l.toString()<< endl;
        l.oddEven();
        cout<<l.toString()<< endl;
        l.addLast(b);
        l.addFirst(a);
        cout<<l.toString()<< endl;

        return 0;
}

```

## K Reverse In Linked List

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of kReverse function. The function is expected to tweak the list such that all groups of k elements in the list get reversed and linked. If the last set has less than k elements, leave it as it is (don't reverse).
3. Input and Output is managed for you.

### Constraints

1. Time complexity  $\rightarrow O(n)$
2. Space complexity  $\rightarrow \text{constant}$

### Format

#### Input

Input is managed for you

#### Output

Output is managed for you

### Example

#### Sample Input

```

11
1 2 3 4 5 6 7 8 9 10 11
3
100
200

```

#### Sample Output

```

1 2 3 4 5 6 7 8 9 10 11
3 2 1 6 5 4 9 8 7 10 11
100 3 2 1 6 5 4 9 8 7 10 11 200

```

```

#include <iostream>
#include <string>

```

```

// #include <bits/stdc++.h>

```

```

using namespace std;

```

```

class Node
{
public:
    int data = 0;

```

```

    Node *next = nullptr;

    Node(int data)
    {
        this->data = data;
    }
};

```

```

class linkedlist
{

```

```

public:

```

```

    Node *head = nullptr;
    Node *tail = nullptr;
    int size = 0;

```

```

//basic->=====

```

```

    int size_()
    {
        return this->size;
    }

```

```

    bool isEmpty()
    {
        return this->size == 0;
    }

```

```

    string toString()
    {
        Node *curr = this->head;
        string sb = "";

        while (curr != nullptr)
        {
            sb += to_string(curr->data);
            if (curr->next != nullptr)
                sb += " ";
            curr = curr->next;
        }
        sb += " ";
        return sb;
    }

```

```

//add->=====

```

```

private:

```

```

    void addFirstNode(Node *node)
    {
        if (this->head == nullptr)

```

```

        {
            this->head = node;
            this->tail = node;
        }
        else
        {
            node->next = this->head;
            this->head = node;
        }

        this->size++;
    }

public:
    void addFirst(int val)
    {
        Node *node = new Node(val);
        addFirstNode(node);
    }

public:
    void addLastNode(Node *node)
    {
        if (this->head == nullptr)
        {
            this->head = node;
            this->tail = node;
        }
        else
        {
            this->tail->next = node;
            this->tail = node;
        }

        this->size++;
    }

    void addLast(int val)
    {
        Node *node = new Node(val);
        addLastNode(node);
    }

    void addNodeAt(Node *node, int idx)
    {
        if (idx == 0)
            addFirstNode(node);
        else if (idx == this->size)
            addLastNode(node);
        else
        {
            Node *prev = getNodeAt(idx - 1);

```

```

        Node *curr = prev->next;

        prev->next = node;
        curr->next = node;

        this->size++;
    }
}

void addAt(int data, int idx)
{
    if (idx < 0 || idx > this->size)
    {
        throw("invalidLocation: " + to_string(idx));
    }

    Node *node = new Node(data);
    addNodeAt(node, idx);
}

//remove->=====
Node *removeFirstNode()
{
    Node *node = this->head;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        this->head = this->head->next;
        node->next = nullptr;
    }

    this->size--;
    return node;
}

int removeFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = removeFirstNode();
    int rv = node->data;
    delete node;
    return rv;
}

```

```

Node *removeLastNode()
{
    Node *node = this->tail;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        Node *prev = getNodeAt(this->size - 2);
        this->tail = prev;
        prev->next = nullptr;
    }

    this->size--;
    return node;
}

int removeLast(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = new Node(val);
    int rv = node->data;
    delete node;
    return rv;
}

Node *removeNodeAt(int idx)
{
    if (idx == 0)
        return removeFirstNode();
    else if (idx == this->size - 1)
        return removeLastNode();
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = curr->next;
        curr->next = nullptr;

        this->size--;
        return curr;
    }
}

```



```

int removeAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = removeNodeAt(idx);
    int rv = node->data;
    delete node;
    return rv;
}

//get->=====
Node *getFirstNode()
{
    return this->head;
}

int getFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = getFirstNode();
    return node->data;
}

Node *getLastNode()
{
    return this->tail;
}

int getLast()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException");
    }

    Node *node = getLastNode();
    return node->data;
}

Node *getNodeAt(int idx)
{
    Node *curr = this->head;

    while (idx-- > 0)
    {

```

```

        curr = curr->next;
    }

    return curr;
}

int getAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = getNodeAt(idx);
    return node->data;
}
//ANS
// linkedlist * kR(int k){
//     if(k > )
// }

public:
void kReverse(int k){
    //write your code here
    linkedlist * ans = new linkedlist {};
    if(size_() >= k){
        for(int i{}; i<k; i++){
            int a = getFirst();
            removeFirst();
            ans->addFirst(a);
        }
        kReverse(k);
        Node * c = ans->head;
        while(c->next != NULL){
            c = c->next;
        }
        c->next = this->head;
        this->head = ans->head;

        if(this->tail == NULL){
            this->tail = c;
        }

        return;
    }else{
        return;
    }
}

};

```

```

linkedlist makeList() {

```

```

    linkedlist l;
    int n1;
    cin >> n1;

    for (int i = 0; i < n1; i++) {
        int val;
        cin >> val;
        l.addLast(val);
    }
    return l;
}

int main()
{
    linkedlist l = makeList();
    int k;
    cin >> k;
    int a;
    cin >> a;
    int b;
    cin >> b;

    cout<<l.toString()<< endl;
    l.kReverse(k);
    cout<<l.toString()<< endl;
    l.addLast(b);
    l.addFirst(a);
    cout<<l.toString()<< endl;

    return 0;
}

```

## Display Reverse (recursive) - Linked List

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of displayReverse and displayReverseHelper functions. The function are expected to print in reverse the linked list without actually reversing it.
3. Input and Output is managed for you.

Note -> The online judge can't force you to write recursive function. But that is what the expectation is, the intention in to help you learn.

### Constraints

1. Time complexity ->  $O(n)$
2. Space complexity ->  $O(n)$

### Format

#### Input

Input is managed for you

#### Output

Output is managed for you

## Example

### Sample Input

```
11
1 2 3 4 5 6 7 8 9 10 11
100
200
```

### Sample Output

```
1 2 3 4 5 6 7 8 9 10 11
11 10 9 8 7 6 5 4 3 2 1
200 1 2 3 4 5 6 7 8 9 10 11 100
```

```
#include <iostream>
```

```
// #include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Node
{
public:
    int data = 0;
    Node *next = nullptr;

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

```
class linkedlist
{
```

```
public:
```

```
    Node *head = nullptr;
    Node *tail = nullptr;
    int size = 0;
```

```
    //basic->=====
```

```
    int size_()
    {
        return this->size;
    }
```

```
    bool isEmpty()
    {
        return this->size == 0;
    }
```

```
    string toString()
    {
```

```

Node *curr = this->head;
string sb = "";

while (curr != nullptr)
{
    sb += to_string(curr->data);
    if (curr->next != nullptr)
        sb += " ";
    curr = curr->next;
}
sb += " ";
return sb;
}

//add->=====
private:
void addFirstNode(Node *node)
{
    if (this->head == nullptr)
    {
        this->head = node;
        this->tail = node;
    }
    else
    {
        node->next = this->head;
        this->head = node;
    }

    this->size++;
}

public:
void addFirst(int val)
{
    Node *node = new Node(val);
    addFirstNode(node);
}

public:
void addLastNode(Node *node)
{
    if (this->head == nullptr)
    {
        this->head = node;
        this->tail = node;
    }
    else
    {
        this->tail->next = node;
        this->tail = node;
    }
}

```

```

        this->size++;
    }

    void addLast(int val)
    {
        Node *node = new Node(val);
        addLastNode(node);
    }

    void addNodeAt(Node *node, int idx)
    {
        if (idx == 0)
            addFirstNode(node);
        else if (idx == this->size)
            addLastNode(node);
        else
        {
            Node *prev = getNodeAt(idx - 1);
            Node *curr = prev->next;

            prev->next = node;
            curr->next = node;

            this->size++;
        }
    }

    void addAt(int data, int idx)
    {
        if (idx < 0 || idx > this->size)
        {
            throw("invalidLocation: " + to_string(idx));
        }

        Node *node = new Node(data);
        addNodeAt(node, idx);
    }

    //remove->=====
    Node *removeFirstNode()
    {
        Node *node = this->head;
        if (this->size == 1)
        {
            this->head = nullptr;
            this->tail = nullptr;
        }
        else
        {
            this->head = this->head->next;
            node->next = nullptr;
        }
    }

```

```

    }

    this->size--;
    return node;
}

int removeFirst(int val)
{
    if (this->size == 0)
    {
        throw("NullPointerException: -1");
    }

    Node *node = removeFirstNode();
    int rv = node->data;
    delete node;
    return rv;
}

Node *removeLastNode()
{
    Node *node = this->tail;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        Node *prev = getNodeAt(this->size - 2);
        this->tail = prev;
        prev->next = nullptr;
    }

    this->size--;
    return node;
}

int removeLast(int val)
{
    if (this->size == 0)
    {
        throw("NullPointerException: -1");
    }

    Node *node = new Node(val);
    int rv = node->data;
    delete node;
    return rv;
}

Node *removeNodeAt(int idx)

```

```

{
    if (idx == 0)
        return removeFirstNode();
    else if (idx == this->size - 1)
        return removeLastNode();
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = curr->next;
        curr->next = nullptr;

        this->size--;
        return curr;
    }
}

int removeAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = removeNodeAt(idx);
    int rv = node->data;
    delete node;
    return rv;
}

//get->=====
Node *getFirstNode()
{
    return this->head;
}

int getFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = getFirstNode();
    return node->data;
}

Node *getLastNode()
{
    return this->tail;
}

```



```

}

int getLast()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException");
    }

    Node *node = getLastNode();
    return node->data;
}

Node *getNodeAt(int idx)
{
    Node *curr = this->head;

    while (idx-- > 0)
    {
        curr = curr->next;
    }

    return curr;
}

int getAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = getNodeAt(idx);
    return node->data;
}

private:
    void displayReverseRecursiveHelper(Node* node){
        //write your code here
        if(node == NULL){
            return;
        }
        // int a = node->data;
        // node = node->next;
        displayReverseRecursiveHelper(node->next);
        cout<< node->data <<" ";
    }

public:
    void displayReverseRecursive(){
        //write your code here
        displayReverseRecursiveHelper(this->head);
    }

```

```

        cout<<endl;
    }
};

linkedlist makeList() {
    linkedlist l;
    int n1;
    cin >> n1;

    for (int i = 0; i < n1; i++) {
        int val;
        cin >> val;
        l.addLast(val);
    }
    return l;
}

int main()
{
    linkedlist l = makeList();
    int a;
    cin >>a;
    int b;
    cin >>b;

    cout<<l.toString()<< endl;
    l.displayReverseRecursive();
    l.addLast(a);
    l.addFirst(b);
    cout<<l.toString()<< endl;

    return 0;
}

```

## Reverse Linked List (pointer - Recursive)

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of reversePR and reversePRHelper functions. The functions are expected to reverse the linked list by using recursion and changing the "next" data member of nodes.
3. Input and Output is managed for you.

Note -> The online judge can't force you to write recursive function, nor can it check if you changed the "next" data member or not. But that is what the expectation is, the intention in to help you learn.

## Constraints

1. Time complexity  $\rightarrow O(n)$
2. Space complexity  $\rightarrow O(n)$

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
11
1 2 3 4 5 6 7 8 9 10 11
100
200
```

### Sample Output

```
1 2 3 4 5 6 7 8 9 10 11
200 11 10 9 8 7 6 5 4 3 2 1 100
```

```
#include <iostream>
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Node
{
public:
    int data = 0;
    Node *next = nullptr;

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

```
class linkedlist
{
```

```
public:
```

```
    Node *head = nullptr;
    Node *tail = nullptr;
    int size = 0;
```

```
//basic->=====
```

```
    int size_()
    {
        return this->size;
    }
```

```

bool isEmpty()
{
    return this->size == 0;
}

string toString()
{
    Node *curr = this->head;
    string sb = "";

    while (curr != nullptr)
    {
        sb += to_string(curr->data);
        if (curr->next != nullptr)
            sb += " ";
        curr = curr->next;
    }
    sb += " ";
    return sb;
}

//add->=====
private:
void addFirstNode(Node *node)
{
    if (this->head == nullptr)
    {
        this->head = node;
        this->tail = node;
    }
    else
    {
        node->next = this->head;
        this->head = node;
    }

    this->size++;
}

public:
void addFirst(int val)
{
    Node *node = new Node(val);
    addFirstNode(node);
}

public:
void addLastNode(Node *node)
{
    if (this->head == nullptr)
    {

```

```

        this->head = node;
        this->tail = node;
    }
    else
    {
        this->tail->next = node;
        this->tail = node;
    }

    this->size++;
}

void addLast(int val)
{
    Node *node = new Node(val);
    addLastNode(node);
}

void addNodeAt(Node *node, int idx)
{
    if (idx == 0)
        addFirstNode(node);
    else if (idx == this->size)
        addLastNode(node);
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = node;
        curr->next = node;

        this->size++;
    }
}

void addAt(int data, int idx)
{
    if (idx < 0 || idx > this->size)
    {
        throw("invalidLocation: " + to_string(idx));
    }

    Node *node = new Node(data);
    addNodeAt(node, idx);
}

//remove->=====
Node *removeFirstNode()
{
    Node *node = this->head;
    if (this->size == 1)

```

```

    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        this->head = this->head->next;
        node->next = nullptr;
    }

    this->size--;
    return node;
}

int removeFirst(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = removeFirstNode();
    int rv = node->data;
    delete node;
    return rv;
}

Node *removeLastNode()
{
    Node *node = this->tail;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        Node *prev = getNodeAt(this->size - 2);
        this->tail = prev;
        prev->next = nullptr;
    }

    this->size--;
    return node;
}

int removeLast(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }
}

```

```

        Node *node = new Node(val);
        int rv = node->data;
        delete node;
        return rv;
    }

Node *removeNodeAt(int idx)
{
    if (idx == 0)
        return removeFirstNode();
    else if (idx == this->size - 1)
        return removeLastNode();
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = curr->next;
        curr->next = nullptr;

        this->size--;
        return curr;
    }
}

int removeAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = removeNodeAt(idx);
    int rv = node->data;
    delete node;
    return rv;
}

//get->=====
Node *getFirstNode()
{
    return this->head;
}

int getFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }
}

```

```

        Node *node = getFirstNode();
        return node->data;
    }

Node *getLastNode()
{
    return this->tail;
}

int getLast()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException");
    }

    Node *node = getLastNode();
    return node->data;
}

Node *getNodeAt(int idx)
{
    Node *curr = this->head;

    while (idx-- > 0)
    {
        curr = curr->next;
    }

    return curr;
}

int getAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = getNodeAt(idx);
    return node->data;
}

private:
void displayReversePointerRecHelper(Node* node){
    //write your code here
    if(node->next == NULL){
        return ;
    }
    displayReversePointerRecHelper(node->next);
    node->next->next = node;
}

```



```

    }

    public:
        void displayReversePointerRec(){
            //write your code here
            Node * c = head;
            while(c->next != NULL){
                c = c->next;
            }
            displayReversePointerRecHelper(head);
            Node *temp = head;
            head = c;//could have used tail
            tail = temp;
        }
};

```

```

linkedlist makeList() {
    linkedlist l;
    int n1;
    cin >> n1;

    for (int i = 0; i < n1; i++) {
        int val;
        cin >> val;
        l.addLast(val);
    }
    return l;
}

```

```

int main()
{
    linkedlist l = makeList();
    int a;
    cin >>a;
    int b;
    cin >>b;

    cout<<l.toString()<< endl;
    l.displayReversePointerRec();
    l.addLast(a);
    l.addFirst(b);
    cout<<l.toString()<< endl;

    return 0;
}

```

# Is Linked List A Palindrome?

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of IsPalindrome function. The function is expected to check if the linked list is a palindrome or not and return true or false accordingly.
3. Input and Output is managed for you.

## Constraints

1. Time complexity  $\rightarrow O(n)$
2. Space complexity  $\rightarrow$  Recursion space,  $O(n)$

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
5
1 2 3 2 1
```

### Sample Output

```
true
```

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

```
class node
{
    public :
    int data;
    node *next;
};
```

```
class linked_list
{
    public:
    node *head,*tail;
    int size=0;
```

```
    public:
    linked_list()
    {
        head = NULL;
        tail = NULL;
    }
```

```
    void add_node(int n)
    {
        node *tmp = new node;
```

```

tmp->data = n;
tmp->next = NULL;

if(head == NULL)
{
    head = tmp;
    tail = tmp;
}
else
{
    tail->next = tmp;
    tail = tail->next;
}
size++;

}

void display(){
    for(node* tmp = head; tmp != NULL; tmp = tmp->next){
        cout<<tmp->data<<" ";
    }
}

int isPalindromehelper(node * right){
    if(right == NULL){
        return 1;
    }
    int to_return = isPalindromehelper(right->next );
    if(to_return == 0){
        return to_return;
    }else{
        if(right->data != left->data){
            return --to_return;
        }else{
            left = left->next;
            return to_return;
        }
    }
}
}

```

//WATCH THE SOL VIDEO ON YOUTUBE OF REVERSE DATA RECURSIVE TO UNDERSTAND BELOW PART

//maybe when we don't have the size we can make a size function that

//calculates size in linear time and set a new data member size which

//we will use in helper (similar to the left node) now it will be in constant

// time (I may be wrong )

```

node* left;

int isPalindrome(){
// write your code here
    left = head;
    int a = isPalindromehelper(head);
    return a;
}
};

int main()
{
    int b ;
    cin>>b;
    linked_list a;
    vector<int> arr(b,0);
    for(int i=0;i<b;i++)
    {

        cin>>arr[i];
        a.add_node(arr[i]);
    }

    int res = a.isPalindrome();
    if(res==1)
    {
        cout<<"true";
    }
    else
    {
        cout<<"false";
    }
    return 0;
}

```

## Fold A Linked List

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of fold function. The function is expected to place last element after 1st element, 2nd last element after 2nd element and so on. For more insight check the example

Example 1

1->2->3->4->5

will fold as

1->5->2->4->3

Example 2

1->2->3->4->5->6

1->6->2->5->3->4

## Constraints

1. Time complexity  $\rightarrow O(n)$
2. Space complexity  $\rightarrow$  Recursion space,  $O(n)$

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
5
1 2 3 4 5
10
20
```

### Sample Output

```
1 2 3 4 5
1 5 2 4 3
10 1 5 2 4 3 20
```

```
#include <iostream>
#include <vector>
```

```
using namespace std;
```

```
class node
{
```

```
    public :
    int data;
    node *next;
```

```
};
```

```
class linked_list
{
```

```
private:
    node *head, *tail;
    int size=0;
```

```
public:
    linked_list()
    {
        head = NULL;
        tail = NULL;
    }
```

```
void addFirst(int val) {
    node* temp = new node();
    temp->data = val;
    temp->next = head;
    head = temp;
```

```

    if (size == 0) {
        tail = tmp;
    }

    size++;
}

void add_node(int n)
{
    node *tmp = new node;
    tmp->data = n;
    tmp->next = NULL;

    if(head == NULL)
    {
        head = tmp;
        tail = tmp;
    }
    else
    {
        tail->next = tmp;
        tail = tail->next;
    }
    size++;
}

}

void display(){
    for(node* tmp = head; tmp != NULL; tmp = tmp->next){
        cout<<tmp->data<<" ";
    }
    cout<<endl;
}

}

node* pleft;
int IsPalindromeHelper(node* right){
    if(right == NULL){
        return 1;
    }

    int rres = IsPalindromeHelper(right->next);
    if(rres == 0){
        return 0;
    } else if(pleft->data != right->data){
        return 0;
    } else {
        pleft = pleft->next;
        return 1;
    }
}
}

```

```

int isPalindrome(){
    pleft = head;
    return IsPalindromeHelper(head);
}

```

```

void foldHelper(node *right,int floor) {
    if(right == NULL){
        return;
    }
    foldHelper(right->next,floor+1);
    if(floor > sz/2){
        right->next = fleft->next;
        node* temp = fleft;
        fleft = fleft->next;
        temp->next = right;
    }else if(floor == sz/2){
        this->tail = right;
        right->next = NULL;
    }
}

```

```

}
//WATCH THE SOL VIDEO ON YOUTUBE OF REVERSE DATA RECURSIVE TO
UNDERSTAND BELOW PART
//maybe when we don't have the size we can make a size
function that
//calculates size in linear time and set a new data member
size which
//we will use in helper (similar to the left node) now it will
be in constant
// time (I may be wrong )

```

```

int sz{};
void siz(){
    node* c = head;
    while(c != NULL){
        c = c->next;
        sz++;
    }
}
node* fleft;
void fold() {
    // write your code here
    fleft = head;
    siz();
    foldHelper(head, 0);
}

```

```

}

```

```

;

int main()
{

    int b ;
    cin>>b;
    linked_list a;
    vector<int> arr(b,0);
    for(int i=0;i<b;i++)
    {

        cin>>arr[i];
        a.add_node(arr[i]);
    }
    int c;
    cin>>c;
    int d;
    cin>>d;
    a.display();
    a.fold();
    a.display();
    a.addFirst(c);
    a.add_node(d);
    a.display();

};

```

## Add Two Linked Lists

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of addLinkedLists function. The function is passed two linked lists which represent two numbers - the first element is the most significant digit and the last element is the least significant digit. The function is expected to add the two linked list and return a new linked list.

The following approaches are not allowed :

1. Don't reverse the linked lists in order to access them from least significant digit to most significant.
2. Don't use arrays or explicit extra memory.
3. Don't convert linked lists to number, add them up and convert the result back to a linked list.

Hint - You are expected to take help of recursion to access digits from least significant to most significant. You have to tackle the challenge of unequal size of lists and manage carry where required.

3. Input and Output is managed for you.



Note-> Make sure to set head and tail appropriately because addFirst and addLast has been used to test their values in the input-output code.

## Constraints

1. Time complexity ->  $O(n)$
2. Space complexity -> Recursion space,  $O(n)$

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
1
1
3
9 9 9
10
20
```

### Sample Output

```
1
9 9 9
1 0 0 0
10 1 0 0 0 20
```

```
#include <iostream>
#include <vector>
#include <string>
#include <cstdlib>
```

```
using namespace std;
```

```
class Node
{
public:
    int data = 0;
    Node *next = nullptr;

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

```
class linkedlist
{
```

```
public:
```

```
    Node *head = nullptr;
    Node *tail = nullptr;
```

```

int size = 0;

//basic->=====

int size_()
{
    return this->size;
}

bool isEmpty()
{
    return this->size == 0;
}

string toString()
{
    Node *curr = this->head;
    string sb = "";

    while (curr != nullptr)
    {
        sb += to_string(curr->data);
        sb += " ";
        curr = curr->next;
    }
    return sb;
}

//add->=====
private:
void addFirstNode(Node *node)
{
    if (this->head == nullptr)
    {
        this->head = node;
        this->tail = node;
    }
    else
    {
        node->next = this->head;
        this->head = node;
    }

    this->size++;
}

public:
void addFirst(int val)
{
    Node *node = new Node(val);
    addFirstNode(node);
}

```

```

public:
    void addLastNode(Node *node)
    {
        if (this->head == nullptr)
        {
            this->head = node;
            this->tail = node;
        }
        else
        {
            this->tail->next = node;
            this->tail = node;
        }

        this->size++;
    }

    void addLast(int val)
    {
        Node *node = new Node(val);
        addLastNode(node);
    }

    void addNodeAt(Node *node, int idx)
    {
        if (idx == 0)
            addFirstNode(node);
        else if (idx == this->size)
            addLastNode(node);
        else
        {
            Node *prev = getNodeAt(idx - 1);
            Node *curr = prev->next;

            prev->next = node;
            curr->next = node;

            this->size++;
        }
    }

    void addAt(int data, int idx)
    {
        if (idx < 0 || idx > this->size)
        {
            throw("invalidLocation: " + to_string(idx));
        }

        Node *node = new Node(data);
        addNodeAt(node, idx);
    }

```

```

//remove->=====
Node *removeFirstNode()
{
    Node *node = this->head;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        this->head = this->head->next;
        node->next = nullptr;
    }

    this->size--;
    return node;
}

int removeFirst(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = removeFirstNode();
    int rv = node->data;
    delete node;
    return rv;
}

Node *removeLastNode()
{
    Node *node = this->tail;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        Node *prev = getNodeAt(this->size - 2);
        this->tail = prev;
        prev->next = nullptr;
    }

    this->size--;
    return node;
}

```

```

int removeLast(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = new Node(val);
    int rv = node->data;
    delete node;
    return rv;
}

Node *removeNodeAt(int idx)
{
    if (idx == 0)
        return removeFirstNode();
    else if (idx == this->size - 1)
        return removeLastNode();
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = curr->next;
        curr->next = nullptr;

        this->size--;
        return curr;
    }
}

int removeAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = removeNodeAt(idx);
    int rv = node->data;
    delete node;
    return rv;
}

//get->=====
Node *getFirstNode()
{
    return this->head;
}

```

```

int getFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = getFirstNode();
    return node->data;
}

Node *getLastNode()
{
    return this->tail;
}

int getLast()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException");
    }

    Node *node = getLastNode();
    return node->data;
}

Node *getNodeAt(int idx)
{
    Node *curr = this->head;

    while (idx-- > 0)
    {
        curr = curr->next;
    }

    return curr;
}

int getAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = getNodeAt(idx);
    return node->data;
}
};

```

```

Node* getMid(Node* head, Node* tail){

```

```

Node* slow = head, *fast = head;
while(fast->next != tail && fast->next->next != tail){
    fast = fast->next->next;
    slow = slow->next;
}

return slow;
}

//merge two sorted linkedlist
linkedlist mergeTwoSortedLists(linkedlist l1, linkedlist l2) {
    linkedlist ans;
    Node* one = l1.head;
    Node* two = l2.head;

    while(one != nullptr && two != nullptr){
        if(one->data < two->data){
            ans.addLast(one->data);
            one = one->next;
        }else{
            ans.addLast(two->data);
            two = two->next;
        }
    }
    while(one!=nullptr){
        ans.addLast(one->data);
        one = one->next;
    }
    while(two !=nullptr){
        ans.addLast(two->data);
        two = two->next;
    }

    return ans;
}

linkedlist mergeSort(Node* head,Node* tail ){
    if(head == tail){
        linkedlist base;
        base.addLast(head->data);
        return base;
    }

    Node* mid = getMid(head,tail);
    linkedlist fsh = mergeSort(head, mid);
    linkedlist ssh = mergeSort(mid->next, tail);

    return mergeTwoSortedLists(fsh,ssh);
}

```

// //MY Answer

---

```
//      int carry{0};
//      linkedlist addTwoListsHelper(Node* c1,Node* c2,int d,int
//      floor){
//          if(c1 == NULL && c2 == NULL){
//              linkedlist result;
//              return result;
//          }
//          linkedlist result;
//          if(floor < d){
//              result = addTwoListsHelper(c1->next,c2,d,floor +1);
//              int a = c1->data + carry;
//              carry = a/10;
//              a = a % 10;
//              result.addFirst(a);
//              return result;
//          }else{
//              result = addTwoListsHelper(c1->next,c2-
//>next,d,floor +1);
//              int a = c1->data + c2->data + carry;
//              carry = a/10;
//              a = a % 10;
//              result.addFirst(a);
//              return result;
//          }
//      }
//  }
```

```
//      linkedlist addTwoLists(linkedlist one, linkedlist two) {
//          // write your code here
//          Node* h1 = one.head;
//          Node* h2 = two.head;
//          int diff = abs(one.size - two.size);
//          linkedlist ans;
//          if(one.size > two.size){
//              ans = addTwoListsHelper(h1,h2,diff,0);
//          }else{
//              ans = addTwoListsHelper(h2,h1,diff,0);
//          }
//          if (carry != 0){
//              ans.addFirst(carry);
//              carry = 0;
//          }
//          return ans;
//      }
```



```

//pepcoding answer
// int carry{0};
    int addTwoListsHelper(Node* c1,Node* c2,int d,int
floor,linkedlist &result){
    if(c1 == NULL && c2 == NULL){
        return 0;
    }

    int carry{};
    int a{};
    if(floor < d){
        carry = addTwoListsHelper(c1->next,c2,d,floor
+1,result);
        a = c1->data + carry;
    }else{
        carry = addTwoListsHelper(c1->next,c2->next,d,floor
+1,result);
        a = c1->data + c2->data + carry;
    }
    carry = a/10;
    a = a % 10;
    result.addFirst(a);
    return carry;
}

```

```

linkedlist addTwoLists(linkedlist one, linkedlist two) {
    // write your code here
    Node* h1 = one.head;
    Node* h2 = two.head;
    int diff = abs(one.size - two.size);
    linkedlist ans;
    int carry{};
    if(one.size > two.size){
        carry = addTwoListsHelper(h1,h2,diff,0,ans);
    }else{
        carry = addTwoListsHelper(h2,h1,diff,0,ans);
    }
    if (carry != 0){
        ans.addFirst(carry);
        carry = 0;
    }
    return ans;
}

```

```

linkedlist makeList() {
    linkedlist l;
    int n1;
    cin >> n1;
}

```

```

        for (int i = 0; i < n1; i++) {
            int val;
            cin >> val;
            l.addLast(val);
        }
        return l;
    }

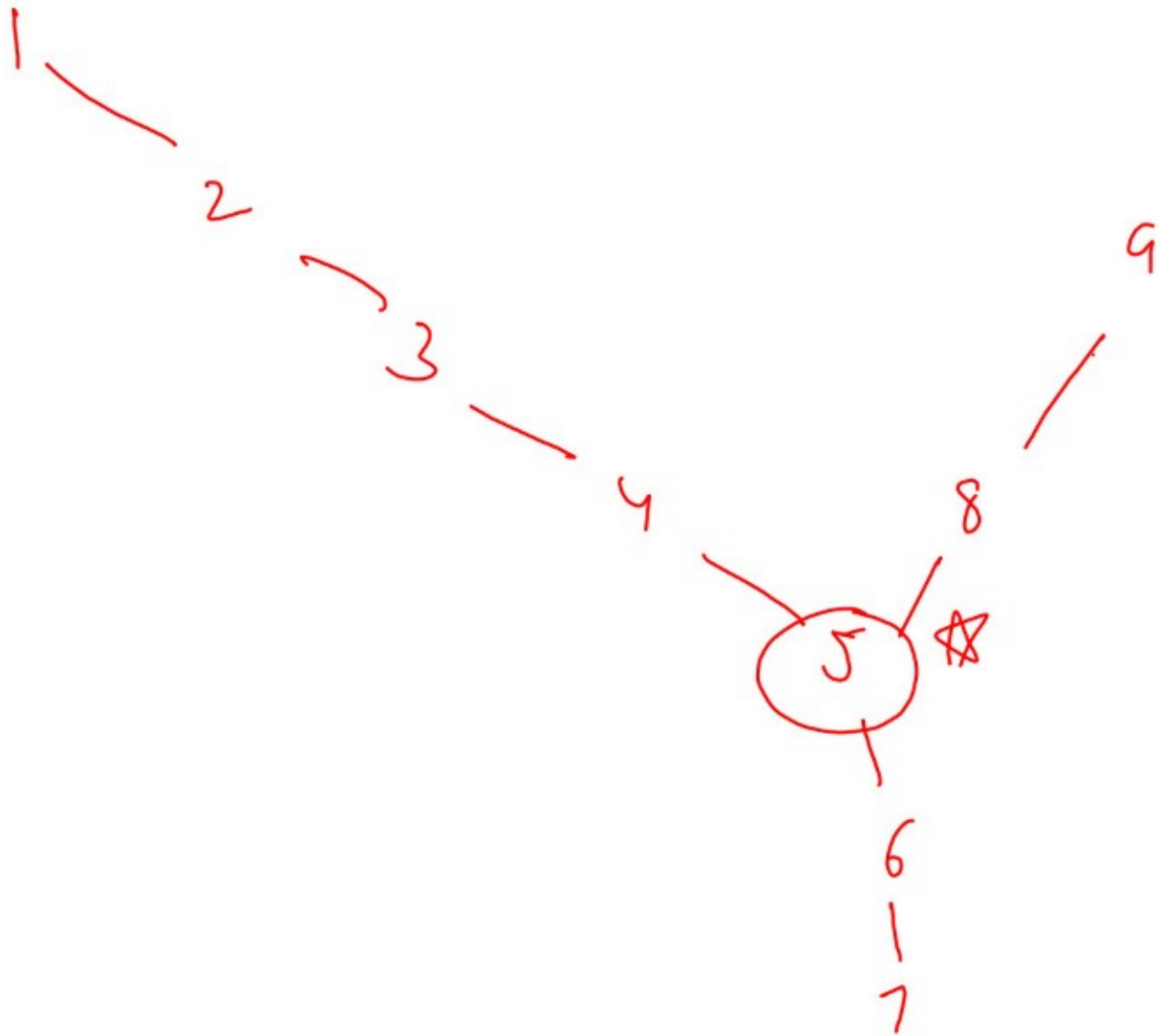
int main()
{
    linkedlist l= makeList();
    linkedlist m= makeList();
    int a;
    cin>>a;
    int b;
    cin>>b;
    linkedlist res= addTwoLists(l,m);
    cout << l.toString() << endl;
    cout << m.toString() << endl;
    cout << res.toString() << endl;
    res.addFirst(a);
    res.addLast(b);
    cout << res.toString() << endl;
    return 0;
}

```

## Intersection Point Of Linked Lists

Easy

1. You are given a partially written LinkedList class.
2. You are required to complete the body of findIntersection function. The function is passed two linked lists which start separately but merge at a node and become common thereafter. The function is expected to find the point where two linked lists merge. You are not allowed to use arrays to solve the problem.
3. Input and Output is managed for you.



## Constraints

1. Time complexity  $\rightarrow O(n)$
2. Space complexity  $\rightarrow$  constant

## Format

### Input

Input is managed for you

### Output

Output is managed for you

## Example

### Sample Input

```
5
1 2 3 4 5
8
11 22 33 44 55 66 77 88
2
3
```

### Sample Output

```
44
#include <iostream>
// #include <bits/stdc++.h>
```

```
using namespace std;
```

```
class Node
{
public:
    int data = 0;
    Node *next = nullptr;

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

```
class linkedlist
{
```

```
public:
```

```
    Node *head = nullptr;
    Node *tail = nullptr;
    int size = 0;
```

```
//basic->=====
```

```
    int size_()
    {
        return this->size;
    }
```

```
    bool isEmpty()
    {
        return this->size == 0;
    }
```

```
    string toString()
    {
        Node *curr = this->head;
        string sb = "";

        while (curr != nullptr)
        {
            sb += to_string(curr->data);
            if (curr->next != nullptr)
                sb += " ";
            curr = curr->next;
        }
        return sb;
    }
```

```

//add->=====
private:
    void addFirstNode(Node *node)
    {
        if (this->head == nullptr)
        {
            this->head = node;
            this->tail = node;
        }
        else
        {
            node->next = this->head;
            this->head = node;
        }

        this->size++;
    }

public:
    void addFirst(int val)
    {
        Node *node = new Node(val);
        addFirstNode(node);
    }

public:
    void addLastNode(Node *node)
    {
        if (this->head == nullptr)
        {
            this->head = node;
            this->tail = node;
        }
        else
        {
            this->tail->next = node;
            this->tail = node;
        }

        this->size++;
    }

    void addLast(int val)
    {
        Node *node = new Node(val);
        addLastNode(node);
    }

    void addNodeAt(Node *node, int idx)
    {
        if (idx == 0)
            addFirstNode(node);
    }

```

```

        else if (idx == this->size)
            addLastNode(node);
        else
        {
            Node *prev = getNodeAt(idx - 1);
            Node *curr = prev->next;

            prev->next = node;
            curr->next = node;

            this->size++;
        }
    }

void addAt(int data, int idx)
{
    if (idx < 0 || idx > this->size)
    {
        throw("invalidLocation: " + to_string(idx));
    }

    Node *node = new Node(data);
    addNodeAt(node, idx);
}

//remove->=====
Node *removeFirstNode()
{
    Node *node = this->head;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        this->head = this->head->next;
        node->next = nullptr;
    }

    this->size--;
    return node;
}

int removeFirst(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = removeFirstNode();

```

```

    int rv = node->data;
    delete node;
    return rv;
}

```

```

Node *removeLastNode()
{
    Node *node = this->tail;
    if (this->size == 1)
    {
        this->head = nullptr;
        this->tail = nullptr;
    }
    else
    {
        Node *prev = getNodeAt(this->size - 2);
        this->tail = prev;
        prev->next = nullptr;
    }

    this->size--;
    return node;
}

```

```

int removeLast(int val)
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = new Node(val);
    int rv = node->data;
    delete node;
    return rv;
}

```

```

Node *removeNodeAt(int idx)
{
    if (idx == 0)
        return removeFirstNode();
    else if (idx == this->size - 1)
        return removeLastNode();
    else
    {
        Node *prev = getNodeAt(idx - 1);
        Node *curr = prev->next;

        prev->next = curr->next;
        curr->next = nullptr;
    }
}

```

```

        this->size--;
        return curr;
    }
}

int removeAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = removeNodeAt(idx);
    int rv = node->data;
    delete node;
    return rv;
}

//get->=====
Node *getFirstNode()
{
    return this->head;
}

int getFirst()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException: -1");
    }

    Node *node = getFirstNode();
    return node->data;
}

Node *getLastNode()
{
    return this->tail;
}

int getLast()
{
    if (this->size == 0)
    {
        throw("nullptrPointerException");
    }

    Node *node = getLastNode();
    return node->data;
}

Node *getNodeAt(int idx)

```



```

{
    Node *curr = this->head;

    while (idx-- > 0)
    {
        curr = curr->next;
    }

    return curr;
}

int getAt(int idx)
{
    if (idx < 0 || idx >= this->size)
    {
        throw("invalidLocation: " + idx);
    }

    Node *node = getNodeAt(idx);
    return node->data;
}
};

```

```

Node* getMid(Node* head, Node* tail){
    Node* slow = head, *fast = head;
    while(fast->next != tail && fast->next->next != tail){
        fast = fast->next->next;
        slow = slow->next;
    }

    return slow;
}

```

```

//merge two sorted linkedlist
linkedlist mergeTwoSortedLists(linkedlist l1, linkedlist l2) {
    linkedlist ans;
    Node* one = l1.head;
    Node* two = l2.head;

    while(one != nullptr && two != nullptr){
        if(one->data < two->data){
            ans.addLast(one->data);
            one = one->next;
        }else{
            ans.addLast(two->data);
            two = two->next;
        }
    }
    while(one!=nullptr){
        ans.addLast(one->data);
        one = one->next;
    }
}

```

```

    }
    while(two != nullptr){
        ans.addLast(two->data);
        two = two->next;
    }

    return ans;
}

linkedlist mergeSort(Node* head, Node* tail ){
    if(head == tail){
        linkedlist base;
        base.addLast(head->data);
        return base;
    }

    Node* mid = getMid(head, tail);
    linkedlist fsh = mergeSort(head, mid);
    linkedlist ssh = mergeSort(mid->next, tail);

    return mergeTwoSortedLists(fsh, ssh);
}

int addTwoLists(Node* on, Node* tn, int pio, int pit,
linkedlist & res){
    if(on == nullptr && tn == nullptr){
        return 0;
    }

    int carry = 0;
    int data = 0;
    if(pio > pit){
        carry = addTwoLists(on->next, tn, pio - 1, pit, res);
        data = carry + on->data;
    } else if(pio < pit){
        carry = addTwoLists(on, tn->next, pio, pit - 1, res);
        data = carry + tn->data;
    } else {
        carry = addTwoLists(on->next, tn->next, pio - 1, pit - 1,
res);
        data = carry + on->data + tn->data;
    }

    carry = data / 10;
    data = data % 10;

    res.addFirst(data);
    return carry;
}

linkedlist addTwoLists(linkedlist one, linkedlist two) {
    linkedlist res ;

```

```

        int carry = addTwoLists(one.head, two.head, one.size,
two.size, res);
        if(carry > 0){
            res.addFirst(carry);
        }

        return res;
    }

```

```

linkedlist makeList() {
    linkedlist l;
    int n1;
    cin >> n1;

    for (int i = 0; i < n1; i++) {
        int val;
        cin >> val;
        l.addLast(val);
    }
    return l;
}

```

```

int findIntersection(linkedlist one, linkedlist two)
{
    // write your code here
    Node * c1 = one.head;
    Node * c2 = two.head;

    if(one.size > two.size){
        for(int i{};i < (one.size - two.size);i++){
            c1 = c1->next;
        }
    }else{
        for(int i{};i < (two.size - one.size);i++){
            c2 = c2->next;
        }
    }
    while(c1 != c2){
        c1 = c1->next;
        c2 = c2->next;
    }
    return c1->data;
}

```

```

int main()
{

    linkedlist l1= makeList();

```

```

linkedlist l2= makeList();
int li;
cin>>li;
int di;
cin>>di;
    if (li == 1) {
        Node * n = l1.getNodeAt(di);

        if (l2.size > 0) {
            l2.tail->next = n;
        } else {
            l2.head = n;
        }

        l2.tail = l1.tail;
        l2.size += l1.size - di;
    } else {
        Node *n = l2.getNodeAt(di);

        if (l1.size > 0) {
            l1.tail->next = n;
        } else {
            l1.head = n;
        }

        l1.tail = l2.tail;
        l1.size += l2.size - di;
    }

    int s =findIntersection(l1,l2);
    cout <<s<< endl;
    return 0;
}

```