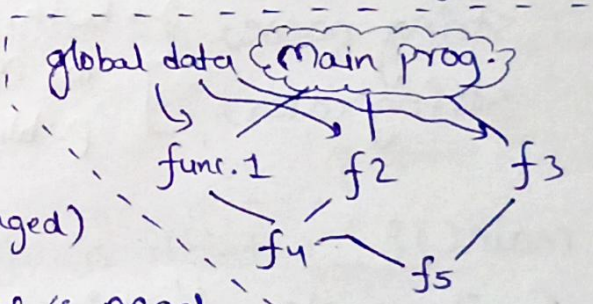
OOPS

Procedure Oriented Programming:

(→) Conventional programming paradigm - use high level lang.
↳ ex. C, FORTRAN, etc.

(→) Programs divided into tasks → group into functions → (main focus)

(→) Global data accessible from entire program.



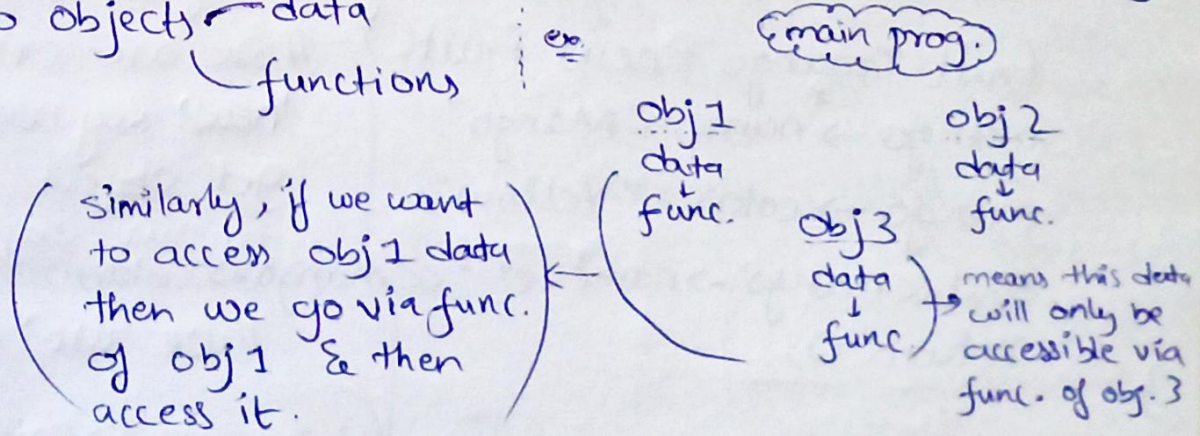
(→) Demerits: → difficult to track which func. changed data (if global value changed)
↳ if data str. is changed, func.'s need to be revised.

Object Oriented Programming:

(→) main focus on data
(→) program → divided into objects ← data
↳ functions

(→) we bind the data to the func. using it

ex.



(→) Class: • fundamental unit of OOP • has user defined data types
We have datatypes like: int, char, bool → built-in datatypes

ex: We have → "Mango, Apple, Grapes" & we need to define this in our program. (just like int, char, strings..)

Now these are of type 'Fruit'. (user-defined datatype)

So we can make a class called → Fruit
- colour
- taste

(where this datatype will have some properties)

& if we were to make variables, then:

Mango Apple Grapes } → are variables

- we can define some data/properties & methods/functions

→ **Objects:**
- Objects are variables of type class. (ex. Mango, Apple, Grapes)
- ex: class Student
  - name
  - roll_no.
  - get_marks() // func.

  (and all objects will have this property)

Now: variable of datatype 'student' = object

ex: Code:

```
class Fruit{
public:
    string name;
    string color;
};
int main(){
    Fruit apple; //object
    apple.name="Apple";
    apple.color="Red";
    cout<<apple.name<<" "<<apple.color;
    cout<<endl; //another way to define an object
    Fruit *mango = new Fruit(); //we are creating an object using
    mango->name="Mango";        'new' keyword. And here we won't
    mango->color="Yellow";      get object but a pointer variable
    cout<<mango->name<<" "<<mango->color;   to our object
    return 0;
}                               //we use '->' operator to access here.
```

(both are private, so to access it in main func., we need to write 'public')

↳ o/p: Apple Red
Mango Yellow

∴ If we create an object using 'new' operator, then we will get a pointer variable which is pointing to that object.

→ **Constructor:**
- are used to initialise an object.
- this is func. which is called when an obj. is created.
- same name as class_name.
- types → default constructor
  - → parameterised constructor
  - → copy constructor

ex: code:

```
class Rectangle{
public:
    int l, b;
    Rectangle(){ //default constructor
        l=0;     no arguments passed
        b=0;
    }
    Rectangle(int x, int y){ //parameterised constr.
        l=x;                  arguments passed
        b=y;
    }
    Rectangle(Rectangle &r){ //copy constr.
        l=r.l;               initialise an obj. by
        b=r.b;               another existing obj.
    }                        (r2 value copied & passed here)
};
```

```
int main(){
    Rectangle r1;
    cout<<r1.l<<" "<<r2.b;
    cout<<endl;
    // value passed
    Rectangle r2(3,4);
    cout<<r2.l<<" "<<r2.b;
    cout<<endl;

    Rectangle r3=r2;
    cout<<r3.l<<" "<<r3.b;
    cout<<endl;
    return 0;
}
```

↳ r1: 0 0
3 4
3 4

ex: Suppose we want copy of r2 in another new object r3 → copy constr. (of same class)

→ **Destructor:**
- func. is called when obj. is deleted.
- cannot pass any arguments.
- name → ~(class_name)

ex: code: (related to above)

```
class Rectangle{
public:
    ⋮
    ~Rectangle(){ //destructor
        cout<<"Destructor called";
        cout<<endl;
    }
};
int main(){
    ⋮
    cout<<r3.l<<" "<<r3.b;
    return 0;
}
```

o/p:
0 0
3 4
3 4
Destructor called
Destructor called
Destructor called

(thrice, as we are deleting 3 rectangle objects.)

if we declare via pointer in main:

```
int main(){
    Rectangle * r1 = new Rectangle();
    cout<<r1->l<<" "<<r1->b<<endl;
    delete r1; //here destructor func. called
    |
    cout<<r3.l<<" "<<r3.b<<endl;
    return 0;
}
```

// 0 0
Destructor called
3 4
3 4
Destructor called
Destructor called.

this keyword only used on a pointer variable

If we did Rectangle r3=r1 (after deleting) → error

⇒ **Encapsulation:**
- It's property of oops which ensures binding of methods & variables together into a single unit → [Class]
- We ensure by seeing iff data is only accessible from the class methods.
- also leads to data abstraction / hiding

(that's why Class = Abstract Data type (ADT))

ex: Code:

```
class ABC{                          (private ele.)
    int x; //suppose we want to extract 'x'
    public:                          so we create func.s
    void set(int n){
        x=n;
    }
    int get(){
        return x;
    }
};
```

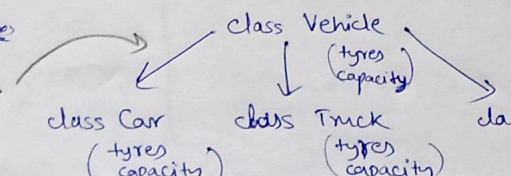
```
int main(){
    ABC obj1;
    obj1.set(3);
    cout<<obj1.get();
    return 0;
}
//if we write obj1.x
   ↳ we will get error.
```

⇒ **Abstraction:**
- enables us to display only essential info. while hiding implementation/unnessary details.
- ex: pow(x,y) → x^y is returned only.

⇒ **Inheritance:**
- a class inherits properties of another class.
- A (Parent class / Super-class)
  ↓
  B (Child class / Sub-class)

Suppose we create 3 classes; Car, Truck, Scooter with few properties. Now, instead of rewriting, we can directly inherit it from class Vehicle.

ex:



- **Access Specifiers & Modes of Inheritance:**

⇒ Public: Data & func.'s can be accessed from anywhere in the code.

ex:
```
class ABC{
    public:
        [ ]
};
```

⇒ Private: accessible only in own class

⇒ Protected: → accessible in own class, parent class & derived class.

ex:
```
class ABC{
    protected:
        [ ]
};
```

ex:
```
class Parent{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};
int main(){
    Parent p;
    p.x; //only 'x' can be accessed.
         but 'y' & 'z' cannot be accs.
}
```

(modes of inheritance)

```
class Child1 : public Parent{
    //x → public
    //y → protected
};  //z → not accessible
class Child2 : private Parent{
    //x → private
    //y → private
};  //z → not accessible
class Child3 : protected Parent{
    //x → protected
    //y → protected
};  //z → not accessible
```

- **Types of Inheritance:**

⇒ **Single Inheritance:** ⇒ Class A
↓
Class B

ex:
```
class Parent{
    public:                    → constructor
        Parent(){
            cout<<"Parent class\n";
        }
};
class Child : public Parent{
    public:
        Child(){
            cout<<"Child class";
        }
};
int main(){
    Child C;
    return 0;
}
```

parent class will be called first, then child class.

// Parent class
Child class

⇒ **Multi-level Inheritance:** ⇒ Parent class derived from another class.
⇒ Class A
↓
Class B
↓
Class C

⇒ ex: //copy from above ex.)

```
};
class Grandchild: public Child{
public:
    Grandchild(){
        cout<<"Grandchild class";
    }
};
int main(){
    Grandchild gc;
    return 0;
}
```

o/p: Parent class
Child class
Grandchild class

⇒ Multiple-Inheritance: (⇒) Inherits from multiple classes

(⇒) Class A, Class B → Class C

(⇒) ex:

```
class Parent1{
public: Parent1(){ cout<<"Parent1 class\n";}
};
class Parent2{
public: Parent2(){ cout<<"Parent2 class\n";}
};
class Child: public Parent1, public Parent2{
public:
    Child(){ cout<<"Child class";};
};
int main(){
    Child c;
    return 0;
}
```

cvc can inherits many classes & properties while defining a class.

o/p:
Parent1 class
Parent2 class
Child class

⇒ Hierarchical Inheritance: (⇒) Class A → Class B, Class C

(⇒) One parent class with multiple child classes.

(⇒) ex: //copy from above ex.)

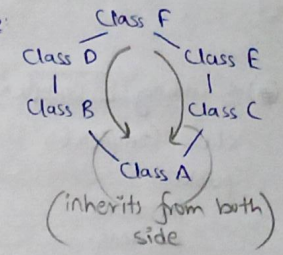
```
class Child2 : public Parent1{
public:
    Child2(){
        cout<<"Child2 class";
    }
};
```

⇒ Hybrid Inheritance: (⇒) Combination of >1 inheritance types.

(⇒) e.g.: Class A → Class B, Class C → Class D

• Diamond Problem: ⇒ base class has multiple parent classes having a common ancestor class

⇒ ex:



(⇒) Code:

```
class Parent{
public: Parent(){cout<<"Parent\n";}
};
class Child1 : public Parent{
public: Child1(){cout<<"Child1\n";}
};
class Child2 : public Parent{
public: Child2(){cout<<"Child2\n";}
};
class Grandchild: public Child1, public Child2{
public: Grandchild(){cout<<"Grandchild";}
};
int main(){
    Grandchild gh;
    return 0;
}
```

o/p:
Parent
Child1
Parent
Child2
Grandchild

(⇒) Polymorphism: - ability of objects/methods to take diff. forms. It depends on compiler/run-time to take which form.

• Compile time polymorphism:

⇒ Function Overloading: (#) define a no. of func.s with same func. name.

(#) they perform differently acc. to the arguments passed. (no. of arg. / type of arg.)

(#) ex:

```
class Sum{
public:
    void add(int x, int y){
        int sum = x+y;
        cout<<sum;
    }
    void add(int x, int y, int z){
        int sum = x+y+z;
        cout<<sum;
    }
    void add(float x, float y){
        float sum = x+y;
        cout<<sum;
    }
};
int main(){
    Sum s;
    s.add(2,3);
    s.add(2,3,4);
    s.add(float(2.3), float(3.7));
    return 0;
}
```

→ it depends on compile time for which func. to call when!

o/p: 5
9
6

⇒ **Operator Overloading**: (#) ex: '+' → 2+3
↳ "cat" + "woman"
∴ Depending on our context which operator we are using!

(#) ex: adding 2 complex no.s:

```
class Complex{
public:
    int real;
    int img;
    Complex(int x, int y){ //constructor for initialising
        real = x;
        img = y;
    }
    operator overload
    Complex operator+ (Complex &c){
        Complex ans(0,0); //complex obj.
        ans.real = real + c.real;
        ans.img = img + c.img;
        return ans;
    }
};
```

(whatever operator we want to use)

(refers to c1)

(c2 is passed i.e. after '+' sign)

(c2)

```
int main(){
    Complex c1(1,2);
    Complex c2(1,3);

    Complex c3 = c1 + c2;
    cout << c3.real << endl;
    cout << "i" << c3.img;
    return 0;
}
```

- **Run time Polymorphism**: ⇒ resolved at runtime, using func. overriding.
(happens when child class defines a func. of parent class.)

#ex:
```
class Parent{
public:
    virtual void print(){
        cout << "Parent" << endl;
    }
    void show(){
        cout << "Parent" << endl;
    }
};
class Child: public Parent{
public:
    void print(){
        cout << "child" << endl;
    }
    void show(){
        cout << "child";
    }
};
```

(need to use this keyword for func. overriding)

```
int main(){
    Parent *p;
    Child c;
    p = &c;
    p->print();
    p->show();
    return 0;
}
```

(means my obj. variable overrided the runtime.)

⇒ o/p: child
parent

(showing since we used 'virtual' keyword.)

Compile Time	Runtime
⇒ through func. overloading & operator overloading	⇒ through func. overriding
⇒ func. name should be same, but parameters can be diff.	⇒ func. name & parameters should be same.
⇒ faster execution time.	⇒ slower execution time.
⇒ more memory efficient.	⇒ less memory efficient.

(⇒) **Friend Function**: • non-member func. which can access private member of the class.

• ex:
```
class A{
    friend F1();
};
```

function F1 → (suppose we want this func. to get data of class A, so we introduce friend F1() in class A, to get all info.)

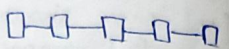
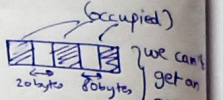
• Code:
```
class A{ ——— (1)
    int x; ——— (2)
public:
    A(int y){
        x = y;
    }
    friend void print(A &obj); ——— (7) → func. name.
};
void print(A &obj){ //passing an obj. of class A ——— (3)
    cout << obj.x; ——— (4)
}
int main(){
    A obj(5); ——— (5)
    // cout << obj.x; ——— (6)
    print(obj); ——— (8)
    return 0;
}
```

(without any value)
If only 1, 2, 7, 3, 4, (5), 8 are there, then our code would run but it will give some random value since we didn't initialise x. (6 → commented)

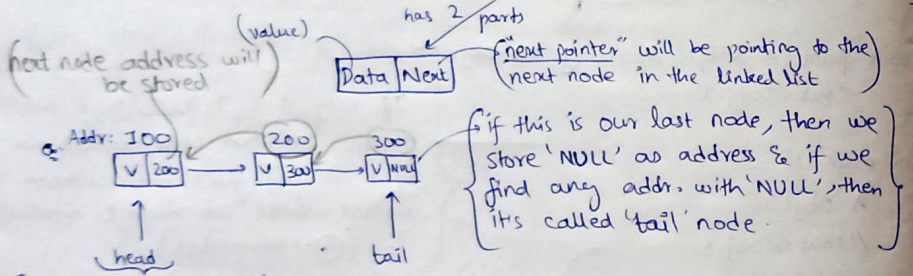
(without any value)
If only 1, 2, 3, 4, (5), 6 are there in code, then it would throw me an error, since 'x' is private

(This func. is independant func., but it can also be of some other class.)

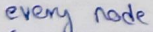
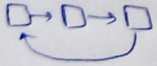
LINKED LIST IN C++ - PART 1

- Linked list is linear data str. used to store a list of values.
- Array:  ; Linked list: 
  - Array: ↳ single memory block with partitions
  - Linked list: ↳ memory blocks linked to each other
- Challenges of array:
  - → static size: if we get extra (or) less i/p wrt size, then according array memory will reach it's limit (or) get wasted.
  - → Contiguous memory allocation: e.g. 100 bytes needed ↳ but: 

  - → Inserting & deleting is costly (O(n))
- Advantages of linked list over an array:
  - → dynamic size → non-contiguous memory allocation
  - → insertion & deletion is non-expensive.
- Listnode:
  - → blocks of memory is called listnode/node. {Linked List = LL}



(head pointer points to first node of my linked list) → we can pass head func., to use linked list

- Types of linked lists:
  - → Singly linked lists: every node points to its successor node ↳ 
  - → Doubly linked lists: every node is connected to it's previous & next node. ↳ 
  - → Circular linked lists: the last node points to head node. ↳ 

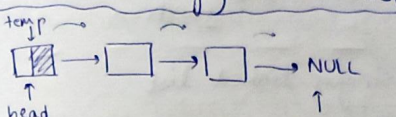
- Implementation of a listnode in a singly linked lists:

```
class Node{
    public:
    int val;
    Node* next;
    Node(int data){
        val=data;
        next=NULL;
    }
};
```

```
int main(){
    Node* n = new Node(1);   // means 'n' named a new node is made whose initializing value=1
    cout<< n->val << " " << n->next;   // & address = NULL
    return 0;
}
```

↳ 1 0x0 (NULL address)

- Traversal in singly linked list:



We make a new pointer 'temp' to traverse our array & we do. "temp = temp → next" to move our 'temp' pointer.

Clearing Basic Concepts: ↴

Constructors ⇒ are automatically invoked when an obj is created using the class's constructor syntax.
eg: 'MyClass obj;' (or) 'MyClass obj(params);'.

Normal Member functions ⇒ they're invoked using dot operator on an obj. like: 'obj.memberFunction(params)'.

Constructors are primarily responsible for initializing the obj. when it's created, while normal member functions perform various operations on the objects data members.

Using Pointers

→ Creating objects with Pointers: ⇒ We can create objects of a class dynamically using pointers.

⇒ `MyClass *ptrObj = new MyClass;` // creating an obj using dynamic memory allocation

→ Accessing Members via Pointers: ⇒ Once you have a pointer to an object, you can access it's members (both variables & member functions) using the arrow (->) operator.

⇒
```
ptrObj -> memberVariable = value;   // accessing member variable
ptrObj -> memberFunction(params);   // calling a member func.
```

→ Constructors with Pointers: ⇒ Constructors are still used to initialize objects, even when creating objects via pointers. (Syntax ↓)

⇒ `MyClass *ptrObj = new MyClass(params);` // creating & initializing with constructor

→ Destructors & deleting objects: ⇒ When we allocate memory using 'new', we are responsible for releasing that memory using 'delete' to avoid memory leaks.

⇒ delete ptrObj; // deallocate memory & call the destructor

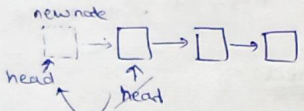
→ Accessing Member Functions via Pointers: ⇒ Calling member func.'s through pointer works the same way as accessing member variables.

⇒ ptrObj -> memberFunction(params); // Calling a member function

---

• Insertion at k^{th} position in a singly linked list:

⇒ Adding node at start:



$T_c = O(1)$, as we need not traverse the array ele.'s

code:
```
class Node{
    public:
    int val;
    Node* next;
    Node(int data){
        val = data;
        next = NULL;
    }
};

int main(){
    Node* head = NULL;
    insertAtHead(head, 2);
    display(head);
    insertAtHead(head, 1);
    display(head);
    return 0;
}
```

(constructor called)

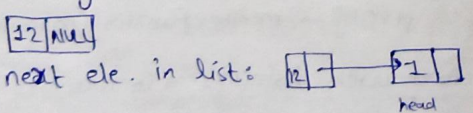
```
void insertAtHead(Node* &head, int val){
    Node* new_node = new Node(val);
    new_node -> next = head;
    head = new_node;
}

void display(Node* head){
    Node* temp = head;
    while(temp != NULL){
        cout << temp -> val << " -> ";
        temp = temp -> next;
    }
    cout << "NULL" << endl;
}
```

① Node* new_node = new Node(val); — named node created whose initial value is val (keyword)

② display function

exp:
2 → NULL
1 → 2 → NULL

Exp!: Here we are initialising node with NULL head.



Now it points to next ele. in list:

Now we shift head of node to it's new prev. ele.

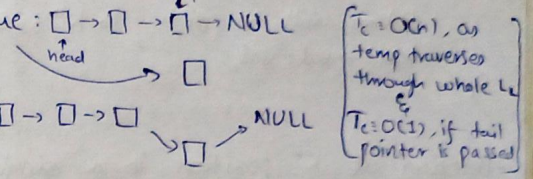
---

① "Node* &head" represents a reference to a pointer to a 'Node'.
→ Node*: represents a pointer to a 'Node'
→ &head: represents address of the pointer variable 'head'.

② We use 'temp' pointer to traverse through linked list.
temp = temp -> next;
↳ points to next ele. address in linked list

⇒ Adding node at the end: → (until temp -> next != NULL)

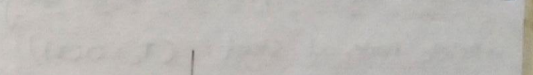
Here we initialise value:



$T_c = O(n)$, as temp traverses through whole LL

$T_c = O(1)$, if tail pointer is passed

Then we relate it:

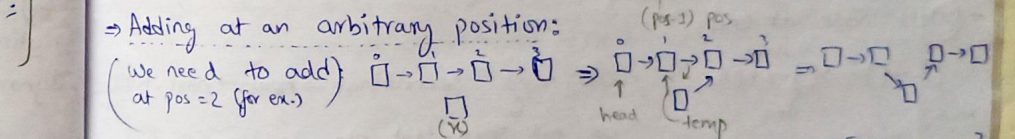


Code (related to prev.)
```
void insertAtTail(Node* &head, int val){
    Node* new_node = new Node(val);
    Node* temp = head;
    while(temp -> next != NULL){
        temp = temp -> next;
    }
    // temp has reached last node
    temp -> next = new_node;
}

int main(){
    ...
    insertAtTail(head, 3);
    display(head);
    ...
}
```

exp:
1 → 2 → 3 → NULL

⇒ Adding at an arbitrary position:

(We need to add at pos = 2 (for ex.))



Code: (related to prev)
```
void insertAtPosition(Node* &head, int val, int pos){
    if(pos == 0){
        insertAtHead(head, val);
        return;
    }
    Node* new_node = new Node(val);
    Node* temp = head;
    int current_pos = 0;
    while(current_pos != pos-1){
        temp = temp -> next;
        current_pos++;
    }
    // temp is pointing to node at pos-1
    new_node -> next = temp -> next;
    temp -> next = new_node;
}
```

→ establishing relation btw 'x' & '2' (see above ex.)

→ establishing relation btw '1' & 'x' (see above ex.)

$T_c = O(position)$

```
int main(){
    insertAtPosition(head, 3, 1);
    display(head);
}
```

exp: 1 → 4 → 2 → 3 → NULL

⇒ Updation at K^{th} position:

$T_c = O(K^{th}$ position$)$

Code:

```
void updateAtPosition(Node* &head, int k, int val){
    Node * temp = head;
    int curr_pos = 0;
    while(curr_pos != k){
        temp = temp -> next;
        curr_pos++;
    }
    //temp will be pointing to k^th node
    temp -> val = val;
}
```

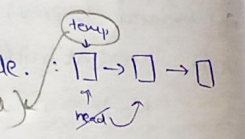
```
int main(){
    ...
    updateAtPosition(head,2,5);
    display(head);
}
```

o/p: 1→4→5→3→NULL

• Deletion at K^{th} position in a singly linked list:

⇒ Delete node at start: ($T_c = O(1)$)

We move our head pointer to next ele. (points to node to be deleted)



Code: (related to prev)

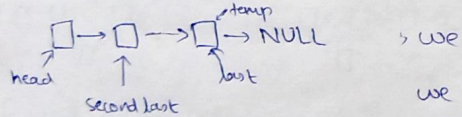
```
void deleteAtHead(Node* &head){
    Node*temp = head; //node to be deleted
    head = head->next;
    free(temp);
}
```

(free (keyword) → used to free up the space taken by first node)

```
int main(){
    deleteAtHead(head);
    display(head);
}
```

o/p: 4→5→3→NULL

⇒ Delete node at end:



we need: secondlast → next → next = NULL

& we will store last ele. in temp so that we can free up it later.

$[T_c = O(n)]$

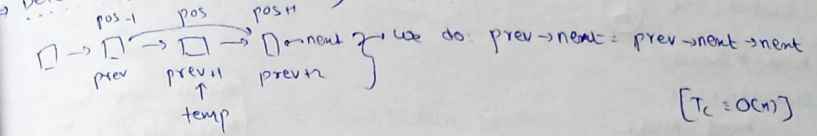
Code (related to prev.)

```
void deleteAtTail(Node* &head){
    Node* second_last = head;
    while(second_last->next->next != NULL){
        second_last = second_last->next;
    }
    //now second last points to second last node
    Node* temp = second_last->next; //node to be deleted
    second_last->next = NULL;
    free(temp);
}
```

```
int main(){
    deleteAtTail(head);
    display(head);
}
```

o/p: 4→5→NULL

⇒ Delete node at arbitrary position:



we do: prev->next = prev->next->next

$[T_c = O(n)]$

Code: (related to prev.)

```
void deleteAtPosition(Node* &head, int pos){
    if(pos==0){
        deleteAtHead(head);
        return;
    }
    int curr_pos = 0;
    Node*prev = head;
    while(curr_pos != pos-1){
        prev = prev->next;
        curr_pos++;
    }
    //prev pointing to Node at pos-1
    Node* temp = prev->next; //node to be deleted
    prev->next = prev->next->next;
    free(temp);
}
```

```
int main(){
    ...
    deleteAtPosition(head,1);
    display(head);
}
```

o/p: 1→5→3→NULL (above 2 del → 1 1)

Q) Given the head of a linked list, delete every alternate element from the list starting from the 2nd element

A) ex: 1 → 2 → 3 → 4 → 5 (curr→next, curr→next→next) & we need: 1→3→5

(curr) (temp)

∴ 1) we make current & temp → (to delete it)

2) ∴ temp = curr→next

curr→next = curr→next→next

3) free(temp) → 1 → 3 → 4 → 5 (curr)

4) curr = curr→next

& process repeats

Stopping condn → a) curr→next == NULL (odd)

b) curr == NULL (even)

Code:

```
class Node{
public:
    int val;
    Node* next;
    Node(int data){
        val = data;
        next = NULL;
    }
};
```

(a)

```
class LinkedList{
public:
    Node* head;
    LinkedList(){
        head = NULL;
    }
    void insertAtTail(int value){
        Node* new_node = new Node(value);
        if(head == NULL){
            head = new_node;
            return;
        }
        Node* temp = head;
        while(temp->next != NULL){
            temp = temp->next;
        }
        temp->next = new_node;
    }
    void display(){
        Node* temp = head;
        while(temp != NULL){
            cout<<temp->val<<"->";
            temp = temp->next;
        }cout<<"NULL"<<endl;
    }
};
```

(b)

main code, before this, all codes used to make & display linked list.

```
void deleteAlternateNodes(Node* &head){
    Node* curr_node = head;
    while(curr_node != NULL and
          curr_node->next != NULL){
        Node* temp = curr_node->next;
        // this node is to be deleted
        curr_node->next = curr_node->next->next;
        free(temp);
        curr_node = curr_node->next;
    }
}
```

```
int main(){
    LinkedList ll;
    ll.insertAtTail(1);
    ll.insertAtTail(2);
    ll.insertAtTail(3);
    :
    ll.insertAtTail(6);
    ll.display();
    deleteAlternateNodes(ll.head);
    ll.display();
    return 0;
}
```

(c)

ip: 1->2->3->4->5->6->NULL

op: 1->3->5->NULL

Always check if code working for odd as well as even ips.

Q) Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

A) 1->2->2->3->3->3 (free) 1->2->3->3->1->2->3->NULL

curr, curr->next, temp

∴ we can say: while (curr != NULL){

    while (curr->val == curr->next->val && curr->next != NULL){

        //remove curr->next node

    } curr = curr->next;

Code:

(a)

(b)

```
void deleteDuplicateNodes(Node* &head){
    Node* curr_node = head;
    while(curr_node != NULL){
        while(curr_node->next != NULL && curr_node->val == curr_node->next->val)
        { //delete curr->next
            Node* temp = curr_node->next;
            curr_node->next = curr_node->next->next;
            free(temp);
        } // loop ends when curr.node & next node values are different or linked list ends
        curr_node = curr_node->next;
    }
}
```

```
int main(){
    LinkedList ll;
    ll.insertAtTail(1);
    ll.insertAtTail(2);
    ll.insertAtTail(2);
    ll.insertAtTail(3);
    ll.insertAtTail(3);
    ll.insertAtTail(3);
    deleteDuplicateNodes(ll.head);
    ll.display();
}
```

(c)

ip: 1->2->2->3->3->3->NULL

op: 1->2->3->NULL

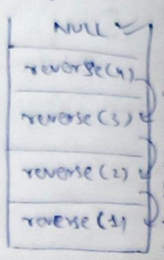
$T_C = O(n)$ → no. of nodes

↳ as we are traversing through our entire array.

Q) Given the head of a singly linked list & print the reversed list.

A) ex: 1 → 2 → 3 → 4 → NULL (head) — we can solve this using recursion.

```
reverse(head){
    if(head==NULL) return; //base case
    reverse(head->next); //calling func.
    cout<<head->val<<endl;
}
```



Code:

```
a)
b)
void reversePrint(Node*&head){
    if(head==NULL){ //base case
        return;
    }
    reversePrint(head->next); //recursive case
    cout<<head->val<<" ";
}
```

```
int main(){
    (3);
    (2);
    (3);
    (4);
    reversePrint(ll.head);
    return 0;
}
```

↳ rp: 1→2→3→4→NULL

o/p: 4 3 2 1

Q) Given the head of a singly linked list, reverse the list, & return the reversed list (NEW CONCEPT)

A) ex: NULL 1 → 2 → 3 → 4 → 5

Pointers: (prev) (current) (next) → this pointer there to move current, once current & prev are related

current->next = prev : NULL ← 1 2→3→4→5

↑ prev (curr) next

++ (all)

⇓

NULL ← 1 ← 2 3→4→5 .. →NULL←1←2←3←4

↑ prev ↑ curr ↑ next

(prev) (current) NULL

Code:

```
Node* reverseLL(Node*&head){
    Node* prevptr=NULL;
    Node* currptr=head;
    while(currptr!=NULL){
        Node* nextptr=currptr->next;
        currptr->next=prevptr;
        prevptr=currptr;
        currptr=nextptr;
    }
    //when this loop ends, prevptr pointing to my last node which is new head.
    Node* new_head=prevptr;
    return new_head;
}
```

moving 3 ptrs by 1 step ahead

```
int main(){
    ll.head=reverseLL(ll.head);
    ll.display();
}
```

1→2→3→4→NULL

4→3→2→1→NULL

Now we can do recursively also!! since we are moving all ptrs by 1 repeatedly

Method 2:

1 → (2→3→4→5) → If 'n' nodes are there, then using recursion we will get 'n-1' nodes reversed

↑ head

1 → (2←3←4←5)

head head->next new_head

Now we do: head->next->next=head (relation ✓ btw 1 & 2 reversed)

1←2←3←4←5

↑ head

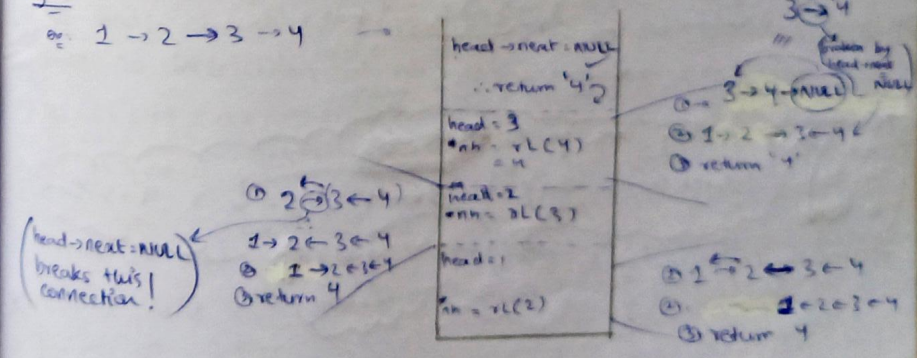
Now: head->next=NULL ∴ NULL←1←2←3←4←5

∴ Code:

```
Node* reverseLLRecursion(Node*&head){
    //base case
    if(head==NULL || head->next==NULL){
        return head;
    }
    //recursive case
    Node* new_head = reverseLLRecursion(head->next); // holds the new head of reversed list
    head->next->next=head;
    head->next=NULL; //head is now pointing to last node in reversed ll.
    return new_head;
}
int main(){
    ll.head=reverseLLRecursion(ll.head);
    ll.display();
}
```

dry run:

ex: 1 → 2 → 3 → 4



Q) Given head of a linked list, reverse the nodes of the list 'k' at a time, & return the modified list?

A) Code:

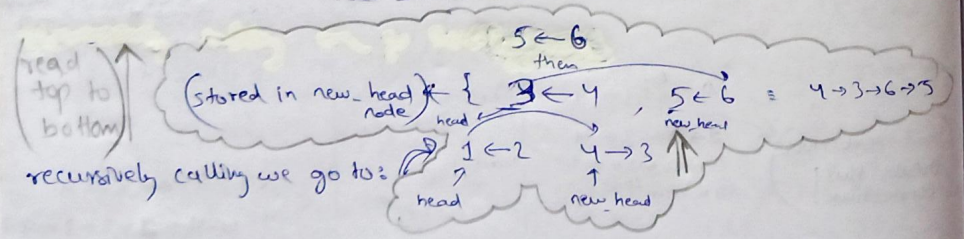
```
Node * reverseKLL (Node * &head, int k){
    Node * prevptr = NULL;
    Node * currptr = head;

    int count = 0; //for counting first 'k' nodes

    while (currptr != NULL and counter<k){ //reversing first k nodes
        Node * nextptr = currptr -> next;
        currptr -> next = prevptr;
        prevptr = currptr;
        currptr = nextptr;
        counter++;
    }
    //currptr will give us (k+1)th node
    if (currptr != NULL){
        Node * new_head = reverseKLL (currptr, k); //recursive call
        head -> next = new_head;
    }
    return prevptr; //prevptr will give us the new_head of connected linked lists.
}

int main(){
    ll.head = reverseKLL (ll.head, 2);
    ll.display();
}
```

o/p: 1→2→3→4→5→6→NULL
↳ 2→1→4→3→6→5→NULL

Now explanation:

ex: 1→2→3→4→5→6 ; k=2

✓ 2→1→4→3→6→5



LINKED LIST PART 2

Q) Given 2 linked lists, Tell if they are equal or not. Two linked lists are equal if they have the same data & the arrangement of data is also the same?

(Pattern : 2 pointers) ↗

A) We can use 2 pointers which traverse through linked list.
i) Nodes have equal value. ii) Length should be same.

ex: 1→2→3→NULL (ptr1)
1→2→3→NULL (ptr2)

call by value as we are just traversing through LL & not making any changes

Code:

```
bool checkEqualLL(Node* head1, Node * head2){
    Node * ptr1 = head1;
    Node * ptr2 = head2;
    while(ptr1 != NULL and ptr2 != NULL){
        if(ptr1->val != ptr2->val){
            return false;
        }
        ptr1 = ptr1 -> next;
        ptr2 = ptr2 -> next;
    }
    //at this point either ptr1 is null, or ptr2 is null, or both are null.
    return (ptr1 == NULL and ptr2 == NULL);
}

int main(){
    LinkedList ll1;
    ll1.insertAtTail(1);
    ll1.insertAtTail(2);
    ll1.insertAtTail(3);
    LinkedList ll2;
    ll2.insertAtTail(1);
    ll2.insertAtTail(2);
    ll2.insertAtTail(3);
    ll1.display();
    ll2.display();
    cout<<checkEqualLL(ll1.head, ll2.head);
    return 0;
}
```

o/p: 1→2→3→NULL
1→2→3→NULL
1

Q) Given heads of 2 singly linked-lists headA & headB, return the node at which 2 lists intersect. If the 2 linked lists have no intersection at all, return null.

A) ex: LL1 ⇒ 1→2→3→4→5→6 (6-3=skipped) ; Length: 6
LL2 ⇒ 9→10→4→5→6 ; 3
(if ptr1=ptr2)

Now we need to set starting point of ptr1 so we substract length of both & skip those many part in longer LL

∴ T: O(n+m+n-m+m) = O(2n+m) ~ O(n+m)
(n for length, n-m for skipping, m checking ptr1=ptr2)

Code:

```
int getLength(Node* head){
    Node* temp = head;
    int length = 0;
    while(temp != NULL){
        length++;
        temp = temp->next;
    }
    return length;
}

Node* moveHeadByK(Node* head1, int k){
    Node* ptr = head1;
    while(k--){
        ptr = ptr->next;
    }
    return ptr;
}

Node* intersection(Node* head1, Node* head2){
    //S1: calculate length of both linked lists
    int l1 = getLength(head1);
    int l2 = getLength(head2);
    //S2: find diff. k between linkedlist & move longer ll ptr by k steps
    Node* ptr1; Node* ptr2;
    if(l1 > l2){ //LL1 is longer
        int k = l1 - l2;
        ptr1 = moveHeadByK(head1, k);
        ptr2 = head2;
    }
    else{ //LL2 is longer
        int k = l2 - l1;
        ptr1 = head1;
        ptr2 = moveHeadByK(head2, k);
    } //S3: compare ptr1 & ptr2 nodes
    while(ptr){
        if(ptr1 == ptr2){
            return ptr1;
        }
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }
    return NULL;
}

int main(){
    LinkedList ll1;
        (1);
        (2);
        (3);
        (4);
        (5);
    ll1.display();
    LinkedList ll2;
        (1);
        (3);
    ll2.head->next->next = ll1.head->next->next->next;
    ll2.display();
    Node* intersect = intersection(ll1.head, ll2.head);
    if(intersect) cout << intersect->val;
    else cout << "-1";
    return 0;
}
```

Op:

```
1->2->3->4->5->NULL
1->3->4->5->NULL
4
```

Q) Given the head of a linked list, remove the k^{th}-node from the end of the list & return its head.

A) Method 1: Delete 'k' from last = delete 'n-k+1' from first.

Here we will traverse the array twice ↳ one for length, ↳ two for deleting.

But if we wanted to solve in 1 traversal:

Method 2: (head), k=3, (temp)

ex. 1 → 2 → 3 → 4 → 5 → 6 → NULL

2 ptrs: ptr1, ptr2 — k steps — ptr1, ptr2 — ptr1, ptr2

∴ When ptr2 = NULL, then ptr1 points at node which needs to be deleted.

Code: // assuming k <= length of linked list

```
void removeKthFromEnd(Node* &head, int k){
    Node* ptr1 = head;
    Node* ptr2 = head;
    int count = k; //move ptr2 by k steps;
    while(count--) ptr2 = ptr2->next;
    if(ptr2 == NULL){ // k = length of linked list
        Node* temp = head;          ↓ ∴ have to delete
        head = head->next;          head node
        free(temp);
        return;
    }
    //now ptr2 is k steps ahead of ptr1.
    //when ptr2 at null (end of list), ptr1 will
    while(ptr2->next != NULL){      be at node to
        ptr1 = ptr1->next;          be deleted.
        ptr2 = ptr2->next;
    }
    //now ptr1 is pointing to the node before kth node from end, so node to be deleted is;
    Node* temp = ptr1->next;        ptr1->next
    ptr2->next = ptr1->next->next;
    free(temp);
}

int main(){
    LinkedList ll;
        (1);
        (2);
        (3);
        (4);
        (5);
        (6);
    ll.display();
    removeKthFromEnd(ll.head, 3);
    ll.display();
}
```

Op:

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

Q) Given 2 sorted linked lists, merge them into 1 singly linked list such that the resulting list is also sorted.

A) ex: ptr1 → 1 → 4 → 5; ptr2 → 2 → 3; compare; ptr1++

dummy head node (its next val = actual head): 1 → 2 → 3 → 4 → 5; ptr3++

Code:

```
Node* mergeLinkedList(Node* &head1, Node* &head2){
    Node* dummyHeadNode = new Node(-1);
    Node* ptr1 = head1;
    Node* ptr2 = head2;
    Node* ptr3 = dummyHeadNode;
    while(ptr1 and ptr2){          if anyone is NULL, loop ends
        if(ptr1->val < ptr2->val){
            ptr3->next = ptr1;
            ptr1 = ptr1->next;
        }
        else{
            ptr3->next = ptr2;
            ptr2 = ptr2->next;
        }
        ptr3 = ptr3->next;
    }
    if(ptr1) ptr3->next = ptr1;    means if(ptr1 != NULL)
    else ptr3->next = ptr2;
    return dummyHeadNode->next;
}

int main(){
    LinkedList ll1;
        1;
        4;
        5;
    ll1.display();
    LinkedList ll2;
        2;
        3;
    ll2.display();
    LinkedList ll3;
    ll3.head = mergeLinkedList(ll1.head, ll2.head);
    ll3.display();
}
```

true because we are linking to the address of remaining address

Can also be written

```
while(ptr1){
    ptr3->next = ptr1;
    ptr1 = ptr1->next;
}
while(ptr2){
    ptr3->next = ptr2;
    ptr2 = ptr2->next;
}
```

o/p: 1→4→5→NULL
2→3→NULL

1→2→3→4→5→NULL

Q) You are given an array of 'k' linked lists, each linked-list is sorted in ascending order. Merge all the linked-lists into one sorted linked-list & return it.

A) Similar to prev. que. i.e. we have 3 LL → L_1, L_2, L_3

We can merge L_1, L_2 = L_4 & now merge L_4 & L_3 → ans.

Code: (related to prev.)

```
Node* mergeKLinkedLists(vector<Node*> &Lists){
    if(lists.size()==0) return NULL;
    while(lists.size()>1){
        Node* mergedHead = mergeLinkedLists(lists[0], lists[1]);   (from prev.)
        lists.push_back(mergedHead);
        lists.erase(lists.begin()); //erase lists[0]
        lists.erase(lists.begin()); //erase lists[1]
    }
    return lists[0];
}

int main(){
    LinkedList ll1;
        (1);
        (3);
        (4);
    display();
    LinkedList ll2;
        (2);
        (4);
        (5);
    display();
    LinkedList ll3;
        (5);
        (6);
    display();

    vector<Node*> lists; //made vector called → lists
    lists.push_back(ll1.head);
    lists.push_back(ll2.head);
    lists.push_back(ll3.head);
    LinkedList mergedLL;
    mergedLL.head = mergeKLinkedLists(lists);
    mergedLL.display();
    return 0;
}
```

ex: vector → head1 head2 head3 head4

1) head1 and head2 will be merged

vector → head3 head4 mergedhead1_2

2) head3 and head4 will be merged

vector → mergedhead1_2 mergedhead3_4

3) mergedhead1_2 and mergedhead3_4 will be merged

vector → mergedhead1_2_3_4

→ indicates a pointer to Node object.

(1) It declares a vector named 'lists' that can hold pointers to objects of type 'Node'.

(Slow Fast Pointer) → Pattern

Q) Find the middle element of the given linked list?

A) slow = ptr → next
fast = ptr → next → next

ex: odd. p_1 → p_2 → p_3, d_1, d_2

(If we want to get ans. in 1 traverse)

ex: 1 → 2 → 3 → 4 → 5 → NULL (for odd)

Code:

```
Node* middleEle(Node* &head){
    Node* slow = head;
    Node* fast = head;
    while(fast!=NULL and fast→next!=NULL){
        slow = slow→next;
        fast = fast→next→next;
    }
    return slow;
}

int main(){
    LinkedList ll;
        (1);
        (2);
        (3);
        (4);
        (5);
        (6);
    display();
    Node* middleNode = middleEle(ll.head);
    cout<< middleNode→val;
}
```

($T_C = O(n)$, n: no. of nodes)

o/p: 1→2→3→4→5→NULL
3

Q) Given head, the head of a linked list, determine if the linked list has a cycle in it?

A) A → n
B → 2n

If 'A' rotates / traverses through circle 'n' points, then B will meet A surely at some point

(slow) (fast)

ex:

∴ Code:

```
bool detectCycle(Node* head){
    if(!head){
        return false;
    }
    Node* slow = head;
    Node* fast = head;
    while(fast and fast->next){
        slow = slow->next;
        fast = fast->next->next;
        if(slow == fast){
            cout<<slow->val<<endl;
            return true;
        }
    }
    return false;
}
```

```
int main(){
    1
    2
    3
    4
    5
    6
    7
    8
display;
ll.head->next->next = ll.head->next->next;
        (x4)
cout<<detectCycle(ll.head);
}
```

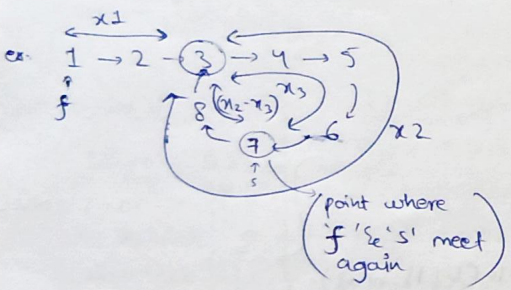
op:

1→2→3→4→5→6→7→8→NULL

7

1

Now suppose we need to remove the cycle;



We use Floyd's Algo.:

Distance from head of linked list to start of cycle is equal to distance between start of cycle & point where f & s pointer meet in cycle.

(i.e. → $x_1 = x_2 - x_3$)

code: assume LL has a cycle

```
void removeCycle(Node* &head){
    Node* slow = head;
    Node* fast = head;
    do{
        slow = slow->next;
        fast = fast->next->next;
    }while(slow != fast)
```

→ first time we don't want to check condn., so we are not using while loop. → (slow != fast)

```
    fast = head;
    while(slow->next != fast->next){
        slow = slow->next;
        fast = fast->next;
    } // now we got cycle starting point
    slow->next = NULL;
}
```

```
int main(){
    ; //above one - copy
    removeCycle(head);
    cout<<detectCycle(ll.head)
    cout<<endl;
    ll.display;
}
```

op:

7

1

0 // no cycle detected

1→2→3→4→5→6→7→8→NULL

Q) Given head, the head of a linked list, determine if the linked list is a palindrome or not

A) eg: 1→2→3→3→2→1 ; process: 1) Find middle element (use s,f ptr)
2) Break linked list into 2
3) Reverse second half of LL
4) Compare the 2 parts of LL

(code)

```
bool isPalindrome(Node* &head){
    // 1) find middle element
    Node* slow = head;
    Node* fast = head;
                    !=NULL
    while(fast and fast->next){
        slow = slow->next;
        fast = fast->next->next;
    } // now slow ptr is pointing to middle ele.
    2) Break LL in the middle
    Node* curr = slow->next;
    Node* prev = slow;
    slow->next = NULL;
    // 3) reverse 2nd half of LL
    while(curr){
        a) Node* nextNode = curr->next;
        b) curr->next = prev;
        c) prev = curr;
        d) curr = nextNode;
    }
    4) check if 2 LL's are equal
    Node* head1 = head;
    Node* head2 = prev;
    while(head2){
        if(head1->val != head2->val){
            return false;
        }
        head1 = head1->next;
        head2 = head2->next;
    }
    return true;
}
```

your LL: 1→2→3→3 2→1→NULL

NULL, prev, curr, next

b) 3→3←2 1→NULL ⇒ 3←2←1 NULL

prev curr prev curr

```
int main(){
    1
    2
    3
    3
    2
    1
    ll.display();
    cout<<isPalindrome(ll.head);
}
```

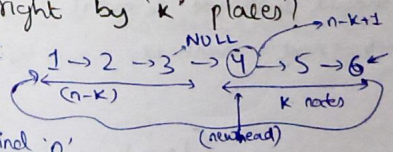
op:

1→2→3→3→2→1→NULL

1

Q) Given the head of a linked list, rotate the list to the right by 'k' places?

A) eg:



; Let K = 3

1) Find 'n'

2) find tail node, tail->next = head;

3) Traverse 'n-k' nodes, (n-k)th node ->next = NULL

4) (n-k+1)th → new head.

Code:

```
Node* rotateByk(Node* &head, int k){
    1) Find length of LL
    int n = 0;
    2) Find tail node
    Node* tail = head;
    while(tail->next){ // != NULL
        n++; //finding length
        tail = tail->next; //finding last node
    }
    n++; //for including last node

    k = k%n;
    if(k==0) return head;

    tail->next = head;
    3) Traverse (n-k) nodes
    Node* temp = head;
    for(int i=1; i<n-k; i++){
        temp = temp->next;
    }
    4) Temp is now pointing to (n-k)th node
    Node* newhead = temp->next;
    temp->next = NULL;
    return newhead;
}
```

```
int main(){
    ll;
    (1);
    (2);
    (3);
    (4);
    (5);
    (6);
    ll.display;
    ll.head = rotateByk(ll.head,3);
    ll.display();
    return 0;
}
```

o/p:

1→2→3→4→5→6→NULL

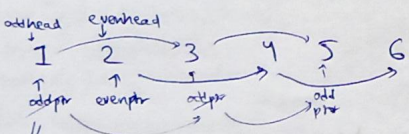
4→5→6→1→2→3→NULL

$\therefore T_c = O(n+n-k) = O(2n-k) \sim O(n)$

(for length) (n-k nodes)

Q) Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, & return the reordered list.

A) ex:



∴ odd->next = odd->next->next

Similarly for evenptr

when even->next = NULL in even=NULL we should link odd & even!

∴ oddptr->next = evenhead.

And here we are traversing length only once.

∴ $T_c = O(n)$ (length of linked list)

Code:

```
Node* oddEvenLL(Node* &head){
    if(!head) return head; // if head=empty
    Node* evenhead = head->next;
    Node* oddPtr = head;
    Node* evenPtr = evenhead;

    while(evenPtr and evenPtr->next){
        oddPtr->next = oddPtr->next->next;
        evenPtr->next = evenPtr->next->next;
        oddPtr = oddPtr->next;
        evenPtr = evenPtr->next;
    }
    oddPtr->next = evenhead;
    return head;
}
```

```
int main(){
    (1);
    (2);
    (3);
    (4);
    (5);
    (6);
    display();
    ll.head = oddEvenLL(ll.head);
    ll.display();
}
```

o/p:

1→2→3→4→5→6→NULL

1→3→5→2→4→6→NULL

Q) You are given the head of a singly linked-list. The list can be represented as: L0→L1→ ⋯ →Ln-1→Ln

Reorder the list to be on the following form:

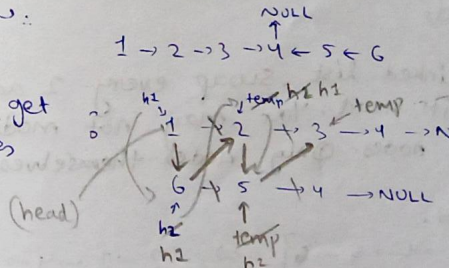
L0→Ln→L1→Ln-1→L2→Ln-2→ ⋯

A) ex:

1→2→3→4→5→6→NULL

Now we can traverse array/list from last; so we do it like palindrome.

∴



Code:

```
Node* reorderLL(Node* &head);
int main(){
    1;
    2;
    3;
    4;
    5;
    6;
    display();
    ll.head = reorderLL(ll.head);
    ll.display();
}
```

o/p: 1→2→3→4→5→6→NULL

1→6→2→5→3→4→NULL

```
Node * reorderLL(Node* &head){
    //can check if LL has atleast 3 nodes
    //1) finding the middle element
    Node* slow = head;
    Node* fast = head;
    while(fast and fast->next){
        slow = slow->next;
        fast = fast->next->next;
    }
    //now slow pointing to middle ele.
    //2) seperate the linked list & reverse second half.
    Node* curr = slow->next;
    slow->next = NULL;
    Node* prev = slow;
    while(curr){
        Node* nextPtr = curr->next;
        curr->next = prev;
        prev = curr;
        curr = nextPtr;
    }
    //3) merging 2 halves of the linked list
    Node* ptr1 = head;
    Node* ptr2 = prev;
    while(ptr1 != ptr2){
        Node* temp = ptr1->next;
        ptr1->next = ptr2;
        ptr1 = ptr2;
        ptr2 = temp;
    }
    return head;
}
```

Q) Given a linked list, swap every 2 adjacent nodes & return its head. You may not modify the values in the list's nodes. Only nodes themselves may be changed.

Ex. (head) (second_node)
$1 \to 2 \to [3 \to 4 \to 5 \to 6] \Rightarrow 2 \to 1 \to 4 \to 3 \to 5 \to 6$
(second node) (head) (newhead)

Here we will use recursion, to swap the no.s!

base case: if head==NULL & head->next==NULL{
    return head;
}

code:

```
Node* swapAdjacent(Node* &head){
    if(head==NULL or head->next==NULL) return head; //base case
    //recursive case
    Node* secondNode = head->next;
    head->next = swapAdjacent(secondNode->next);
    secondNode->next = head;
    return secondNode;
}

int main(){
    (1);
    (2);
    (3);
    (4);
    (5);
    display();
    ll.head = swapAdjacent(ll.head);
    ll.display();
}
```

o/p: 1→2→3→4→5→NULL
2→1→4→3→5→NULL

To be continued...

IMP POINTS

* floor(): returns the largest integer that is smaller than (or) equal to the value passed.

ex: floor(2.3) = 2

ceil(): returns the smallest integer that is greater than (or) equal to the value passed.

ex: ceil(2.3) = 3

COMPLETE GUIDE TO OOPS CONCEPT

- "Class" is basically a type or a blueprint from which you go & create objects
- Objects are instances of class → {type template}

ex:

```
Cycle c1 = new Cycle();
```

(obj. that gets created) — c1
(calling constructor) — Cycle()
(class) — Cycle

```
c1.brand = "Hero";
c1.name = "Fluffy";
```

} setting the value