Data Structure

3. Linked List

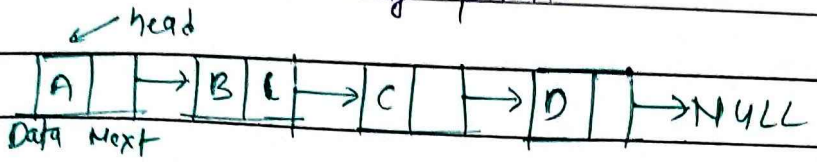
Handwritten by pankaj kumar

Linked List

Date
Page No. 145.

① * Introduction :-

like array, linked list is a linear data structure. unlike array, linked list elements are not stored at contigyour location; the element are linked using pointer



*Why linked list? /Array dis Advantages

Array can be used to store linear data of similar type, but array have following limitations.

① size of array is Fixed

② Insertion at begning or middle is expensive but using linked list we can do this by only traversing at the specific position

③ Deletion is also expensive : after deletion of element one have to shift all the right element by one to left.

* Advantages over array:

① Dynamic size

② Ease of insertion/deletion

* Drawbacks

① Random access is not allowed, we cannot do binary search efficiently.

② Extra memory space for a pointer is required with each element of the list

③ Not cache friendly.

* Representation

- A LL is represented by a pointer to the First node of LL. the First node is called the head,
- Each node in a list consist of atleast two parts.
  ① data (Int, string, any type)
  ② pointer (or Reference) to the next node

(2) Singly linked list using Function:

```
#include <bits/stdc++.h>
using namespace std;
class Node{
    public:
    int data;
    Node * next;
    Node(int data){
        this->data = data;
        this->next = NULL;
    }
    ~Node(){
        int value = this->data;
        if(this->next != NULL){
            delete next;
            this->next = NULL;
        }
        cout<<"memory is free for node with data"<<value<<endl;
    }
};
```

(1) `void insertAtHead(Node* &head, int d){`

```
    Node* temp = new Node(d);
    temp->next = head;
    head = temp;
}
```

(2) `void insertAtTail(Node *&tail, int d){`

```
    Node* temp = new Node(d);
    tail->next = temp;
    tail = temp;
}
```

(3)
```
void print (Node* &head){
    if (head == NULL) {
        cout<<"List is empty"<<endl;
        return;
    }
    Node *temp = head;
    while (temp != NULL){
        cout<<temp->data<<" ";
        temp = temp->next;
    }
    cout<<endl;
}
```

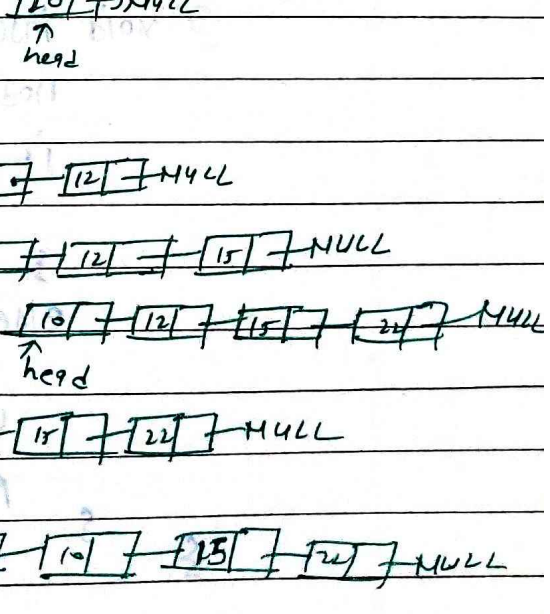
(4)
```
void insertAtPosition(Node* &tail, Node* &head, int position, int d){
    if (position == 1) {
        insertAtHead(head, d);
        return;
    }
    Node *temp = head;
    int cnt = 1;
    while (cnt < position-1){
        temp = temp->next;
        cnt++;
    }
    if (temp->next == NULL){
        insertAtTail(tail, d);
        return;
    }
    Node * nodeToInsert = new Node(d);
    nodeToInsert->next = temp->next;
    temp->next = nodeToInsert;
}
```

(5) void deleteNode(int position, Node* &head){

```
    if(position==1){
        Node *temp=head;
        head=head->next;
        temp->next=NULL;
        delete temp;
    }
    else{
        Node *curr=head;
        Node *prev=NULL;
        int cnt=1;
        while(cnt<position){
            prev=curr;
            curr=curr->next;
            cnt++;
        }
        prev->next=curr->next;
        curr->next=NULL
        delete curr;
    }
}
```

⇒ int main(){

```
Node *node1=new Node(10); //
Node *head=node1;
Node *tail=node1;
insertAtTail(tail,12) //
insertAtTail(tail,15) //
insertAtPosition(tail,head,4,22); //
deleteNode(2,head); //
InsertAtHead(head,5); //
return 0;
}
```



(3) Singly linked list using class:-

```
class node{
    public:
        int data;
        node *next;
        node(){
            next = NULL;
        }
        node(int data){
            this->data = data;
            next = NULL;
        }
};

class LinkedList{
        node *head;
        public:
            LinkedList(){ head=NULL;};
        ① void insert_at_beginning(int v){
            node *temp = new node(v);
            temp->next = head;
            head = temp;
        }
        ② void insert_at_end(int v){
            node *temp = new node(v);
            if (head == NULL){
                head = temp;
            }
            else{
                node *ptr = head;
                while(ptr->next != NULL){ptr = ptr->next;}
                ptr->next = temp;
            }
        }
```

③ void insert_at_given_position(int v, int p){
node *temp = new node(v);
if(p==0){
temp->next = head;
head = temp;
}
else{
node *ptr = head;
while(p>1){
ptr = ptr->next;
--p;
}
temp->next = ptr->next
ptr->next = temp;
}
}

④ void delete_at_begining(){
if(head==NULL){
cout<<"List is Empty"<<endl;
}
else if(head->next==NULL){
cout<<"Element Deleted:"<<head->data<<endl;
delete(head);
head = NULL;
}
else{
node *temp = head;
cout<<"deleted:"<<head->data<<endl;
head = head->next;
delete(temp);
}
}

⑤ void delete_at_end(){
if(head==NULL){
cout<<"List is Empty"<<endl;
}

```
else if (head->next == NULL){
        cout<<"Element Deleted: " << head->data <<endl;
        delete (head);
        head = NULL;
}
else{
        node *temp = head;
        while (temp->next->next != NULL){
                temp = temp->next;
        }
        cout<<"Element Deleted:" <<temp->next->data <<endl;
        delete (temp->next);
        temp->next = NULL;
}
}
```

(6) void delete_at_given_position(int p){

```
        if (head == NULL)
                cout<<"List is empty"<<endl;
        else{
                node *temp, *ptr;
                if(p==0){
                        cout<<"Element Deleted:" <<head->data <<endl;
                        ptr = head;
                        head = head->next;
                        delete (ptr);
                }
                else{
                        temp = ptr = head;
                        while (p>0){
                                --p;
                                temp = ptr;
                                ptr = ptr->next;
                        }
                        cout<<"Element Deleted:" <<ptr->data <<endl;
```

```
        temp->next = ptr->next;
        free(ptr);
    }
}
```

(7) void print(){
    if (head == NULL){
        cout<<"List is empty"<<endl;
    }
    else{
        node *temp = head;
        cout<<"Linked List: ";
        while (temp != NULL){
            cout<<temp->data<<"->";
            temp = temp->next;
        }
        cout<<"NULL"<<endl;
    }
};

⇒ int main(){
    LinkedList ll; // head -> NULL
    ll.insert_at_beginning(5); // [5] -> NULL (head)
    ll.insert_at_end(6); // [5] -> [6] -> NULL (head)
    ll.insert_at_given_position(~~1~~, 1); // [5] -> [7] -> [6] -> NULL (head)
    ll.print();
    ll.delete_at_beginning(); // [7] -> [6] -> NULL (head)
    ll.delete_at_end(); // [7] -> NULL (head)

    return 0;
}

⑨ Doubly linked list :-

```
#include<bits/stdc++.h>
using namespace std;
class Node{
    public:
    int data;
    Node *prev;
    Node * next;
    Node (int d){
        this->data = d;
        this -> prev = NULL;
        this -> next = NULL;
    }
    ~Node(){
        int val = this->data;
        if (next != NULL){
            delete next;
            next=NULL;
        }
        cout<<"memory free for node with data "<<val<<endl;
    }
};

① void print(Node* head){
    //    Node* temp = head;
        while (temp != NULL){
            cout<<temp->data<<" ";
            temp=temp->next;
        }
        cout<<endl;
}
```

```
(2) int getLength(Node* head){
      int len=0;
      Node* temp=head;
      while(temp!=NULL){
        len++;
        temp=temp->next;
      }
      return len;
    }
(3) void insertAtHead(Node* &tail, Node* &head, int d){
      if(head==NULL){
        Node* temp=new Node(d);
        head=temp;
        tail=temp;
      }
      else{
        Node* temp=new Node(d);
        temp->next=head;
        head->prev=temp;
        head=temp;
      }
    }
(4) void insertAtTail(Node* &tail, Node* &head, int d){
      if(tail==NULL){
        Node* temp=new Node(d);
        tail=temp;
        head=temp;
      }
      else{
        Node* temp=new Node(d);
        tail->next=temp;
        temp->prev=tail;
        tail=temp;
      }
    }
```

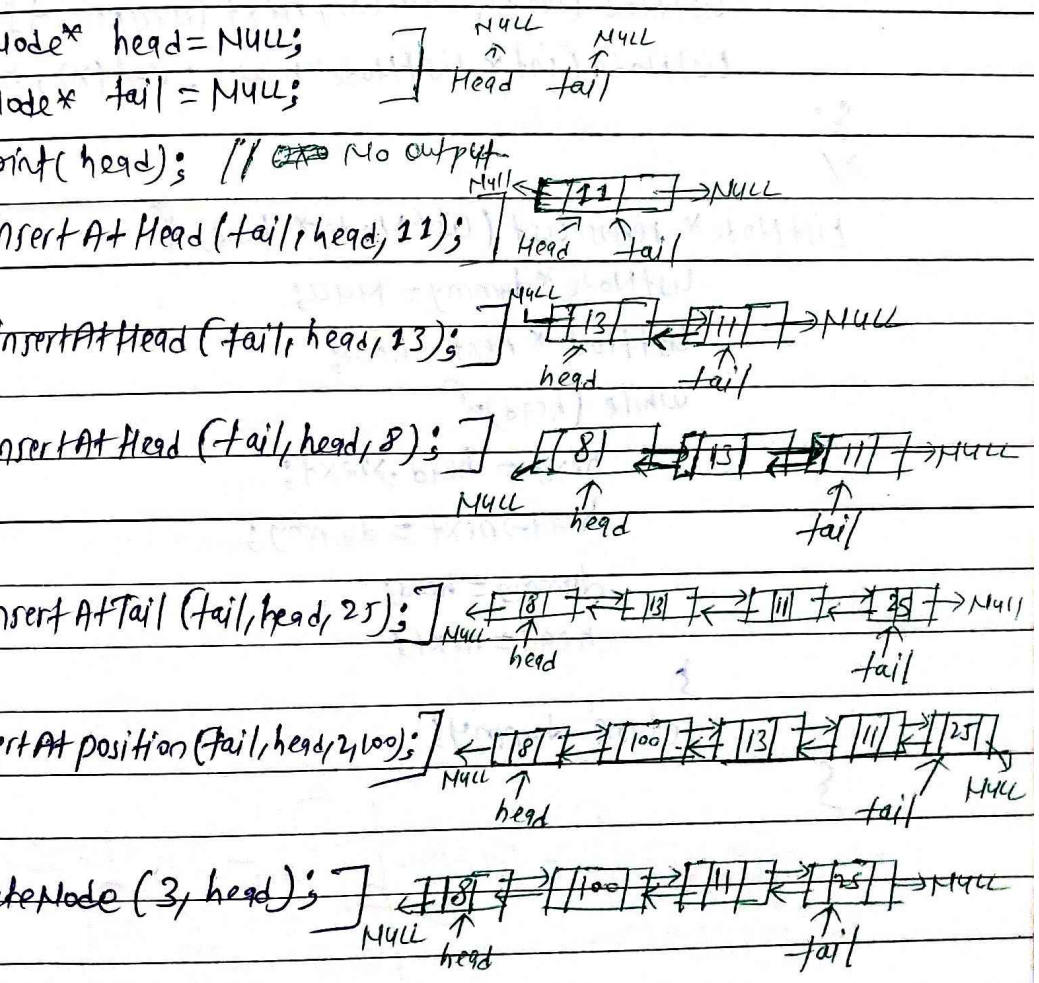
(5) void insertAtPosition (Node* &tail, Node* &head, int position, int d){
if(position == 1){
insertAtHead (tail, head, d);
return;
}
Node *temp = head;
int cnt = 1;
while (cnt < position-1){
temp = temp->next;
cnt++;
}
if (temp->next == NULL){
insertAtTail (tail, head, d);
return;
}
Node* nodeToInsert = new Node (d);
nodeToInsert->next = temp->next;
temp->next->prev = nodeToInsert;
temp->next = nodeToInsert;
nodeToInsert->prev = temp;
}

(6) void deleteNode (int position, Node* &head){
if (position == 1){
Node* temp = head;
temp->next->prev = NULL;
head = temp->next;
temp->next = NULL;
delete temp;
}
else{
Node* curr = head;
Node *prev = NULL;

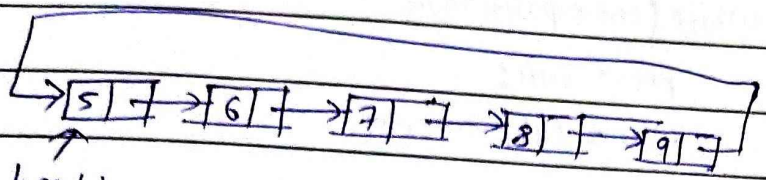
```
int cnt=1;
while(cnt<position){
    prev=curr;
    curr=curr->next;
    cnt++;
}
curr->prev=NULL;
prev->next=curr->next;
curr->next=NULL;
delete curr;
}
}
```

⇒

```
int main(){
    Node* head=NULL;
    Node* tail=NULL;
    print(head); // No output
    insertAtHead(tail,head,11);
    insertAtHead(tail,head,13);
    insertAtHead(tail,head,8);
    insertAtTail(tail,head,25);
    insertAtPosition(tail,head,2,100);
    deleteNode(3,head);
    return 0;
}
```



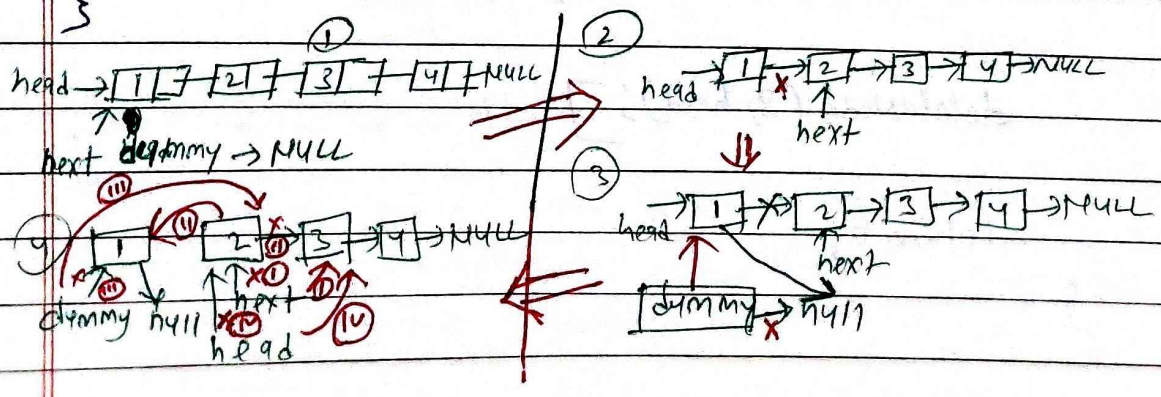
(5) Circular Linked list :-



• in circular list ^last node is point to the first node.

(6) Reverse Linked list :-

```
/**
  Struct ListNode{
        int val;
        ListNode *next;
        ListNode(): val(0), next(nullptr){}
        ListNode(int x): val(x), next(nullptr){}
        ListNode(int x, ListNode *next): val(x), next(next){}
};
*/
ListNode* reverseList(ListNode* head){
        ListNode *dummy = NULL;
        ListNode* next = head;
        while (head){
              next = head->next;
              head->next = dummy;
              dummy = head;
              head = next;
        }
        return dummy;
}
```



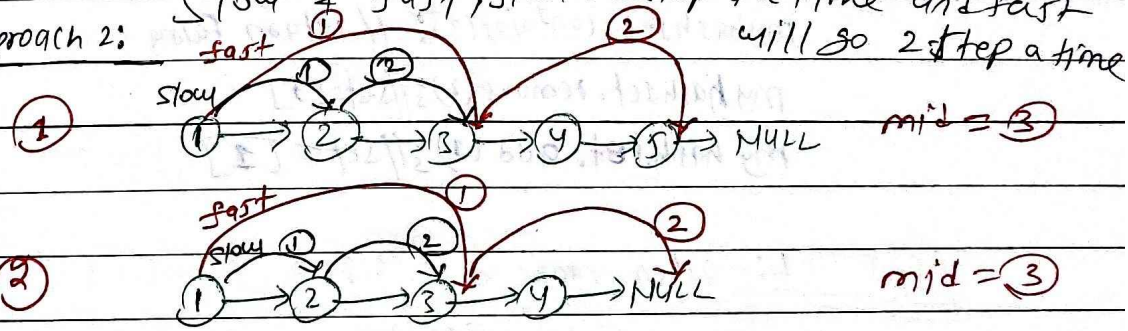
7) Middle of the linked list:-

Note: the node is defined as same as the previous question

Approach 1: Find the total length and again go to the half of the length. (in two iteration)

```
ListNode* middleNode(ListNode* head){
    int n=0;
    ListNode* temp=head;
    while(temp!=NULL){
        n++;
        temp=temp->next;
    }
    int half=n/2;
    while(half--){ temp=temp->next; }
    return temp;
}
```

Approach 2: Slow & fast, slow one step at a time and fast will go 2 step a time




```
ListNode* middleNode(ListNode* head){
    int n=0;
    ListNode *slow=head, *fast=head;
    while(fast!=NULL && fast->next!=NULL){
        slow=slow->next;
        fast=fast->next->next;
    }
    return slow;
}
```

- When fast will reach to the end slow will be in the middle.

(8) Design a Hashset without using builtin hashtable:

Implement MyHashSet class:

- void add(key) insert the value [key] into the HashSet.
- bool contains(key) Returns whether the value [key] exist in HashSet or not.
- void remove(key) Removes the value [key] in the HashSet. If [key] does not exist in the HashSet, do nothing.

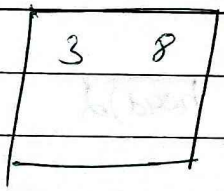
Input:

["MyHashSet", "add", "add", "contains", "contains", "remove", "add"]
[[], [1], [2], [1], [3], [2], [1]]

O/p:- [null, null, null, true, False, null, null]

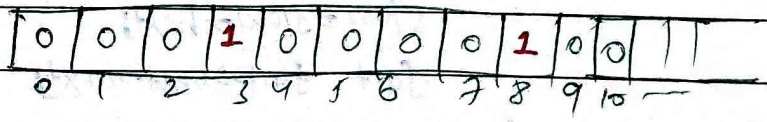
```
MyHashSet myhashset = new MyHashSet();
myhashset.add(1); //set=[1]
myhashset.add(2); //set=[1,2]
myhashset.contains(2); //return true
myhashset.contains(3); // return False
myhashset.remove(2); //set=[1]
myhashset.add(1); //set=[1], bcz set contain unique element
```

Approach 1:- when range $< 10^7$ or 10^6



add(3);
add(8);

maintain a vector and set the all value to 0 when a key will be add change their index to 1

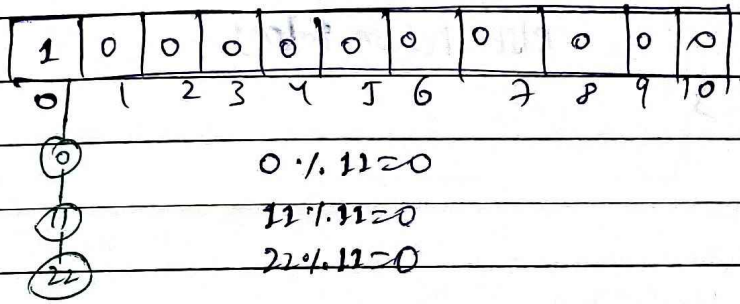


code:-

```
class MyHashSet{
    public:
        vector<int>m;
        int siz;
```

```
MyHashSet(){
    size = 1e6+1;
    m.resize(siz);
}
void add(int key) { m[key]=1;}
void remove(int key) {m[key]=0;}
bool contains(int key) { return m[key];}
};
```

Approach 2: using hash function (if range > 10^7 or, 10^6)
if there is collison we use doubly linked for that:
ex:- add(0), add(11), add(22)



Code:-

```
class MyHashSet{
    public:                          // doubly LL
        vector<list<int>>m;
        int siz;
        MyHashSet(){
            siz=1e6+1;               // we can change the size with 100, 1000, etc.
            m.resize(siz);
        }
        int hash(int key){           // hash function
            return key%siz;
        }
        list<int> ::iterator search(int key){
            int i = hash(key);
            return find(m[i].begin(), m[i].end(), key);
        }
```

search the element is present or not in LL

```
void add (int key){
    if (contains(key)) return;
    int i=hash(key);
    m[i].push_back(key);
}
void remove(int key){
    if(!contains(key)) return;
    int i=hash(key);
    m[i].erase(search(key));
}
bool contains(int key){
    int i= hash(key);
    if(search(key)!=m[i].end()) return true;
    else return false;
}
};
```

(9.) Design HashMap without using any built in hash table.

Implement the MyHashMap class:

- MyHashMap() initializes the object with an empty map.
- void put(int key, int value) inserts a (key, value) pair into the HashMap. If the key already exists in the map, update the corresponding value.
- int get(int key) returns the value to which the specified key is mapped, or -1, if the no mapping of key.
- void remove(key) removes the key and its corresponding value if the map contains the mapping for the key.

Ex- ["myHashmap", "put", "put", "get", "get", "put", "get", "remove", "get"]

[[], [1,1], [2,2], [1], [3], [2,1], [2], [2], [2]]

Op [null, null, null, 1, -1, null, 1, null, -1]

Code :-

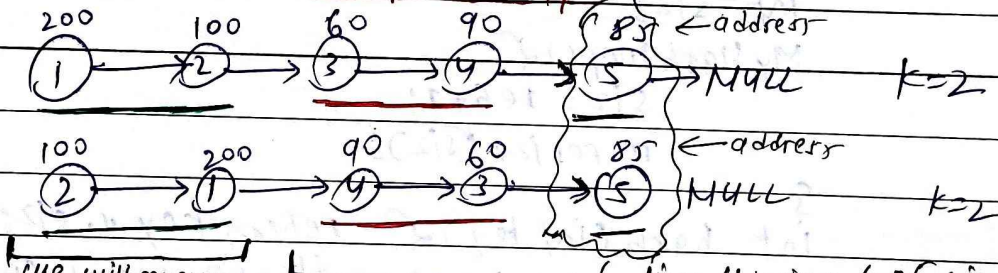
```
class MyHashMap {
    public:
        vector<int> m;
        int siz;
        MyHashMap() {
            siz = 1e6+1;
            m.resize(siz);
        } fill(m.begin(), m.end(), -1);
        void put(int key, int value) {
            m[key] = value;
        }
        int get(int key) { return m[key]; }
        void remove(int key) { m[key] = -1; }
};
```

Approach 1 :-

```
class MyHashMap {
    public:
        vector<list<pair<int, int>>> m;
        int siz;
        MyHashMap() {
            siz = 1e6+1;
            m.resize(siz);
        }
        int hash(int key) { return key % siz; }
        list<pair<int, int>>::iterator search(int key) {
            int i = hash(key);
            list<pair<int,int>>::iterator it = m[i].begin()
            while (it != m[i].end()) {
                if (it->first == key) return it;
                it++;
            }
            return it;
        }
```

```
void put (int key, int value) {
    int i=hash(key);
    remove(key);
    m[i].push_back({key, value});
}
int get(int key){
    int i = hash(key);
    list<pair<int,int>>::iterator it=search(key);
    if(it==m[i].end()) return -1;
    else return it->second;
}

void remove(int key){
    int i= hash(key);
    list<pair<int,int>>::iterator it = search(key);
    if(it!=m[i].end()) m[i].erase(it);
}
};
```

(10.) Reverse Nodes in k-Groups



Recursion Approach:-

we reverse first k-node and then we call the recursion for next node (it will return the remaining list in k-group reverse) then we connect both.

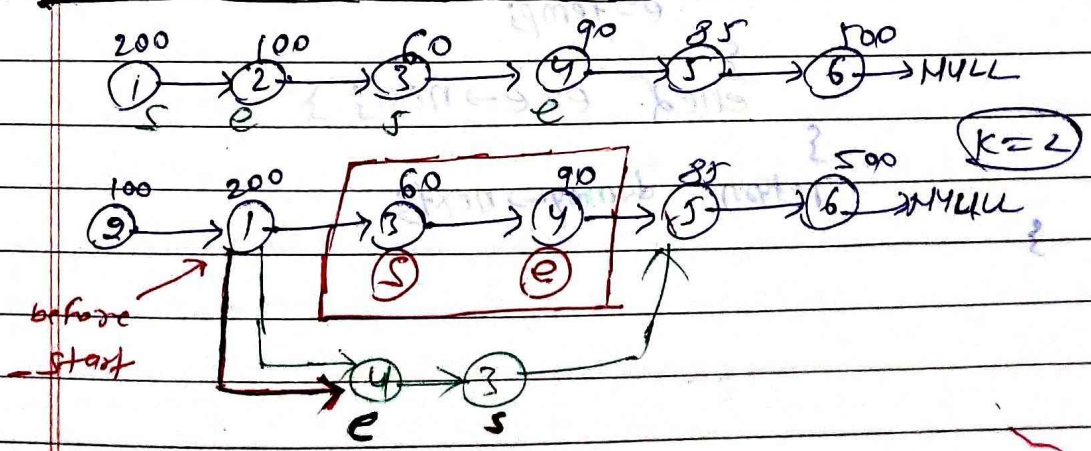
definition of node

```
/* struct ListNode{
    int val;
    ListNode *next;
    ListNode(): val(0), next(nullptr){}
    ListNode(int x): val(x), next(nullptr){}
};
```

```
class Solution{
public:
    void reverse (ListNode *s, ListNode *e){
        ListNode *p=NULL, *c=s, *n=s->next;
        while (p!=e){
            c->next=p;
            p=c;
            c=n;
            if (n!=NULL) n=n->next;
        }
    }
    ListNode* reverseKGroup (ListNode* head, int k){
        if (head==NULL || head->next==NULL || k==1) return head;
        ListNode *s=head, *e=head;
        int inc=k-1;
        while (inc--){
            e=e->next;
            if (e==NULL) return head;
        }
        ListNode* nextHead=reverseKGroup (e->next, k);
        reverse (s, e);
        s->next=nextHead;
        return e;
    }
};
```

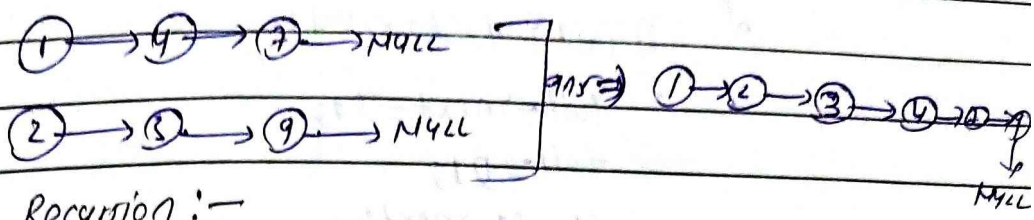
Time: O(n) | Space: O(N/k) (for of Recursion)

*Space Optimize method:-

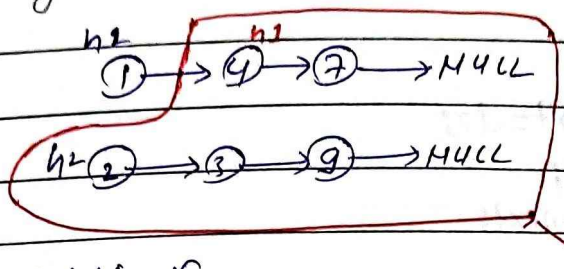


```
class Solution{
    public:
        void reverse(ListNode *s, ListNode *e){
            ListNode *p=NULL, *c=s, *n=s->next;
            while(p!=e){
                c->next=p;
                p=c;
                c=n;
                if(n!=null) n=n->next;
            }
        }

        ListNode* reverseKGroup(ListNode* head, int k){
            if(head==NULL || head->next==NULL || k==1) return head;
            ListNode* dummy=new ListNode(-1);
            dummy->next=head;
            ListNode *beforestart=dummy, *e=head;
            int i=0;
            while(e!=NULL){
                i++;
                if(i%k==0)
                {
                    ListNode *s=beforestart->next, *temp=e->next;
                    reverse(s,e);
                    beforestart->next=e;
                    s->next=temp;
                    beforestart=s;
                    e=temp;
                }
                else{ e=e->next; }
            }
            return dummy->next;
        }
```

(11) Merge two sorted lists



* Using Recursion :-



h1->val < h2->val

ans
1 → 2

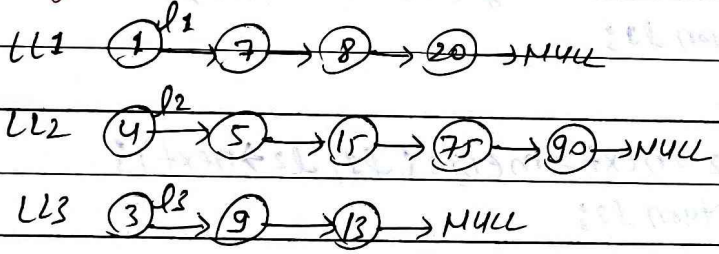
```
Class Solution {
  public:
    ListNode* merge (ListNode* l1, ListNode* l2)
    {
        if (l1==NULL) return l2;
        if (l2==NULL) return l1;
        if (l1->val < l2->val)
        {
            l1->next=merge(l1->next, l2);
            return l1;
        }
        else{
            l2->next = merge (l1, l2->next);
            return l2;
        }
    }
    ListNode* mergeTwoLists (ListNode* l1, ListNode * l2)
    {
        return merge(l1,l2);
    }
};
```

* Iterative :-

```
ListNode * mergeTwoList (ListNode * l1, ListNode * l2)
{
    if (l1==NULL) return l2;
    if (l2==NULL) return l1;
    ListNode* ans=new ListNode(-1);
    ListNode * tail=ans;
```

```
while (l1 != NULL && l2 != NULL)
{
    if (l1->val < l2->val)
    {
        tail->next = l1;
        tail = l1;
        l1 = l1->next;
    }
    else
    {
        tail->next = l2;
        tail = l2;
        l2 = l2->next;
    }
}
if (l1 == NULL) tail->next = l2;
else tail->next = l1;
return ans->next;
}
};
```

(12) Merge k Sorted Linked list.



Approach 1:- insert all the nodes in a vector and sort them again make the linke list

Approach 2:- pick up two list and merge them using merge two sorted list process and again pickup two list and merge them untill one list remains.

Approach 3:- compare all the link list First element and pick up them and add in sorted link list increase the one node (next node for that pick up list) and again start same process. (k comparison for each iteration)

Approach 4: (using Heap):-

It will take log k

take First k nodes in Heap take (min or max) from heap again add the next node in heap start same process till all nodes (list) are processed

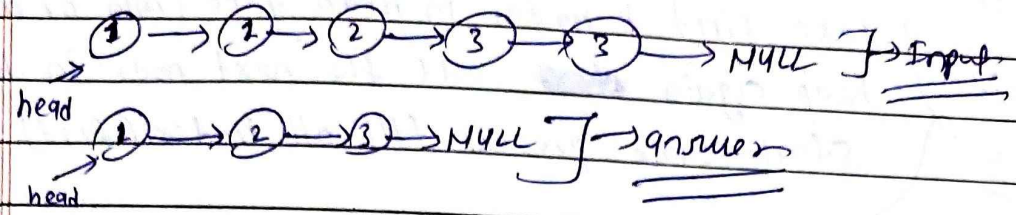
code:-

```
class cmp{public:
    bool operator() (ListNode* a, ListNode* b){
        return a->val > b->val;
    }
};
Class Solution{
    public:
    ListNode* mergeKLists (vector<ListNode*>& lists){
        priority_queue<ListNode*, vector<ListNode*>, cmp> q;
        For(int i=0; i<lists.size(); i++)
        {
            if(list[i]!=NULL) q.push(lists[i]);
        }
        ListNode* dummy = new ListNode(-1);
        ListNode* tail = dummy;
        while (q.size())
        {
            ListNode* temp = q.top();
            tail->next = temp;
            tail = temp
            q.pop();
            if(temp->next!=NULL) q.push(temp->next);
        }
        return dummy->next;
    }
};
```

Date

Page No. 170.

(13) Remove duplicate from sorted list.



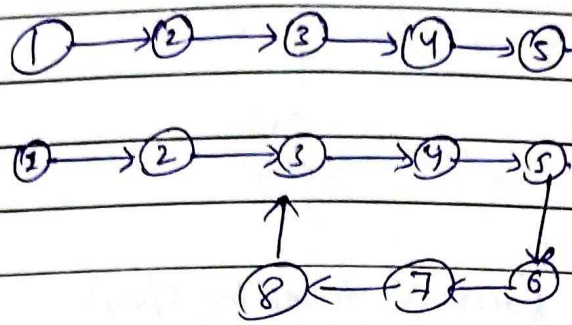
Approach 1: (using Recursion) :-

```
class Solution{
public:
    ListNode* deleteDuplicates(ListNode* head)
    {
        if(head==NULL || head->next==NULL) return head;
        ListNode* newHead=deleteDuplicates(head->next);
        if(head->val==newHead->val) return newHead;
        else{
            head->next=newHead;
            return head;
        }
    }
};
```

Approach 2: (Iterative) :-

```
class Solution{
public:
    ListNode* deleteDuplicates(ListNode *head)
    {
        if(head==NULL||head->next==NULL) return head;
        ListNode* temp=head;
        while(temp->next!=NULL)
        {  if(temp->val==temp->next->val)
            {  ListNode* del=temp->next;
               temp->next=del->next;
               delete del;
            }
            else{
               temp=temp->next;
            }
        }
        return head;
    }
};
```

(14) Cycle detection in Linked List

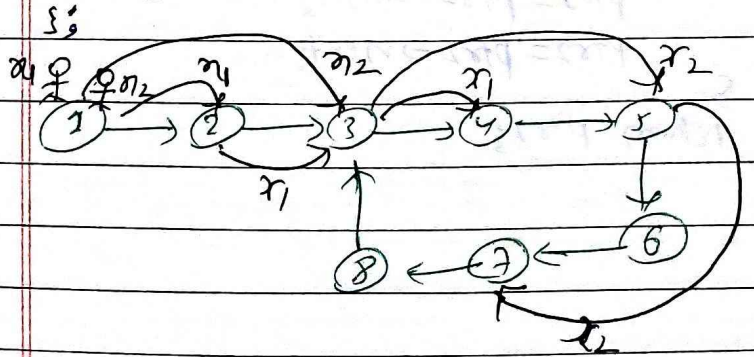


Approach 1:- (using set):
Create a set, iterate over the list if the node is present in set then return that node if not present the add the node in set.

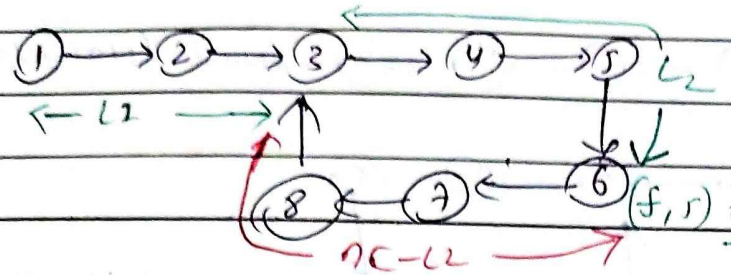
Time:- O(n), space O(n)

Approach 2:- (Slow & Fast):

```
class Solution{
  public:
    bool hasCycle(ListNode *head)
    {
        ListNode * slow=head, *fast=head;
        while(fast!=NULL && fast->next!=NULL){
            slow=slow->next;
            fast=fast->next->next;
            if(fast==slow) return true;
        }
        return false;
    }
```



(15) Start of a linked list cycle



$s \rightarrow L1 + L2$ (distance travel by slow)

$f \rightarrow L1 + L2 + nC$ (n = circular length & C = no. of round by fast)

$\therefore 2(L1+L2) = L1+L2+nC$

$L1+L2 = nC$

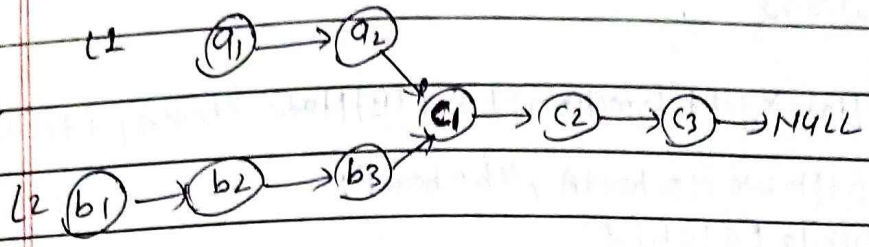
$L1 = (nC - L2)$

Note:- if no cycle return null

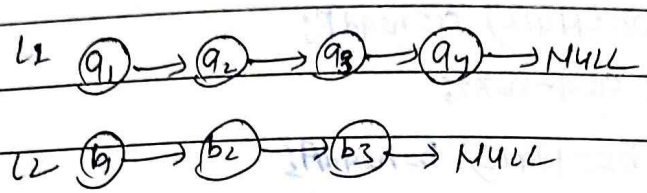
```
Class Solution {
    public:
        ListNode *detectCycle (ListNode *head) {
            ListNode * slow=head, *fast=head;
            while (fast!=NULL && fast->next!=NULL){
                    slow=slow->next;
                    fast=fast->next->next;
                    if(fast==slow) break;
            }
            if(fast==NULL||fast->next==NULL) return NULL;
            ListNode* ptr1=head,*ptr2=slow;
            while (ptr1!=ptr2){
                    ptr1= ptr1->next;
                    ptr2= ptr2->next;
            }
            return ptr1;
        }
};
```

(16) Intersection of two linked list

case 1



case 2

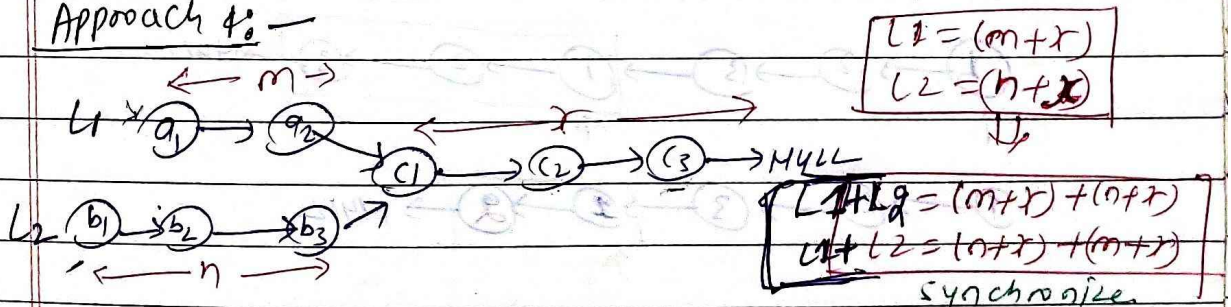


Approach 1: (using two loop):- Choose every node from one list and check if this is present or not in second list by traversing whole 2nd list. Time: O(m*n)

Approach 2: (using set):- insert every node from first list, and then iterate over second list and check if it is present or not in set. Time: O(m+n), space: O(max(m,n))

Approach 3: (synchronize length):- Find the length of both list if the length of one list is more than other then first move in that list which is greater in length to the abs(len1-len2) and then start comparing from both list by moving both list together.

Approach 4:-



move L1 & L2 together if L1 reach at the NULL then move L1 to the b1 and if L2 reach at the NULL then move L2 to the a1 Now the length will be same & if there is intersection it will definetly meet otherwise it will reach at null together

code:-

```
class Solution{
  public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB)
    {
      ListNode* a=headA, *b=headB;
      while(a!=b){
          if(a==NULL) a=headB;
          else a=a->next;
          if(b==NULL) b=headA;
          else b=b->next;
      }
      return a;
    }
};
```

(17) Palindrome linked list

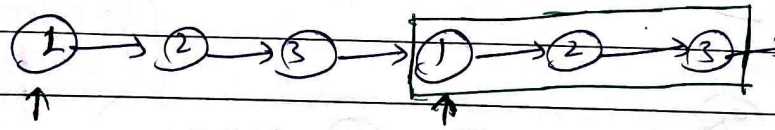
Q. 1: 1 → 2 → 3 → 3 → 2 → 1 → NULL

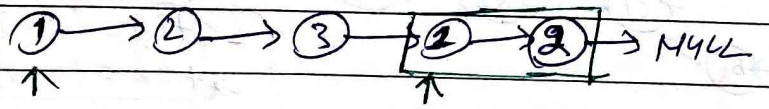
Q. 2: 1 → 2 → 3 → 2 → 1 → NULL

Note:- there is no two linked for single question, 9 have taken for two test case

Approach 1: store all the data in vector and other check palindrome. Space complexity will be more in this case.

Approach 2: Reverse last half of the list and then check from beginning and half of the list by traversing each node, that value is equal or not.





Note:- since the structure of the list will be change so we have to again reverse the last half node

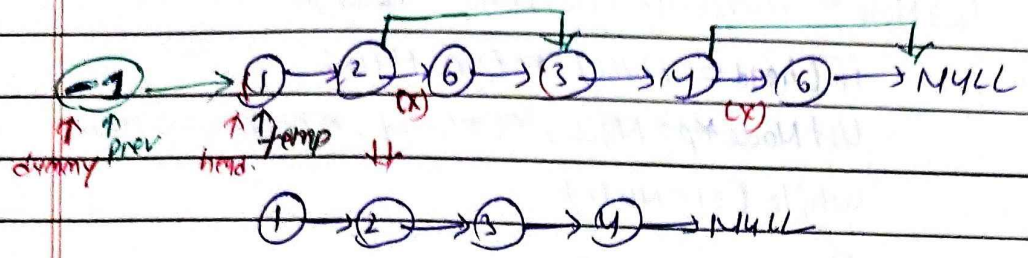
```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(head==NULL) return NULL;
        ListNode *p=NULL, *c=head, *n=head->next;
        while(c!=NULL)
        {
            c->next=p;
            p=c
            c=n
            if(n!=NULL) n=n->next;
        }
        return p;
    }

bool isPalindrome(ListNode* head){
        ListNode *slow=head, *fast=head;
        while(fast->next!=NULL && fast->next->next!=NULL)
        { slow=slow->next;
          fast=fast->next->next;
        }
        slow->next=reverseList(slow->next);
        ListNode *start=head, *mid=slow->next;
        while(mid!=NULL){
            if(mid->val!=start->val) return false;
            start=start->next;
            mid=mid->next;
        }
        slow->next=reverseList(slow->next);
        return true;
    }
};
```

Time:- $\frac{n}{1} + \frac{n}{2} + \frac{n}{2} + \frac{n}{2} = 2n = O(n)$

space = $O(1)$

(18) Remove Linked List Elements

val = 6



iterate temp and ~~dummy~~ prev over the list dummy is prev of the head if temp value is equal to val then add prev next to the temp next (i.e delete the temp). and at last return dummy->next