C++ [Part 2]

Date
Page

{Lecture 50}

(Linked List in C++ (Part 1))

⇒ What is a linked list? → Collection of Nodes [Info | link]
linear data structure used to store a list of values

Array

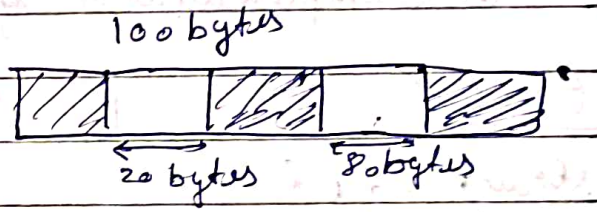
a single Memory Block with partitions

Linked list

Memory Blocks linked to each other

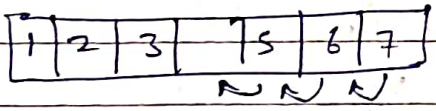
⇒ Challenges of array
→ static size

→ Contiguous Memory allocation



But No ~~element~~ array can be inserted

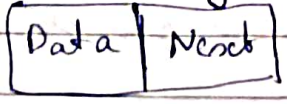
→ Inserting and Deleting is costly (O(n))

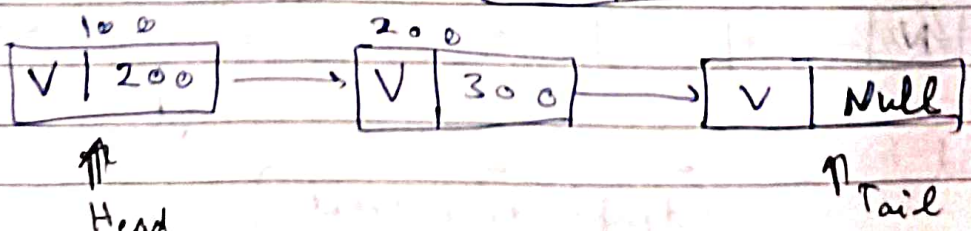


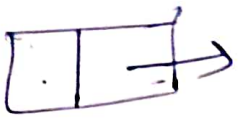
⇒ Advantages of a linked list over an array
→ Dynamic Size
→ Non contiguous Memory allocation
→ Insertion and Deletion is not expensive.

⇒ Listnode

blocks of memory → node







Date
Page

⇒ Types of linked lists

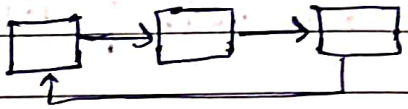
→ Singly linked list

every node points to its successor node

→ Doubly linked list

every node is connected to its previous & next node.

→ Circular linked list



the last node point to head node.

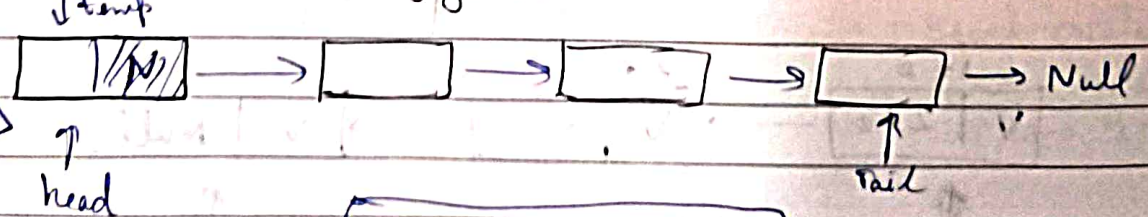
⇒ Implementation of a listnode in a singly linked list.

For C

```
struct node {
    int value;
    struct node* next;
};
```

```
Class Node { public:
    int value;
    Node* next;

    Node (int data) {
        value = data;
        next = null;
    }
};
int main() {
    Node* n = new Node(1)
    cout << n->value << " " << n->next << endl;
    return 0;
}
```

⇒ Traversal in a Singly linked list



Creat a new pointer temp to traverse the linked list

temp = temp->next

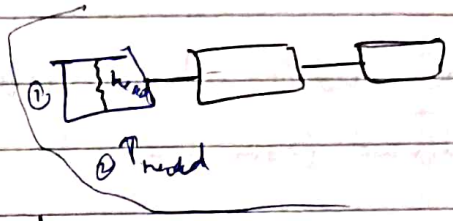
FREEMIND

Date

Page

⇒ Insertion at k^{th} position in a Singly linked list.

- Add a node at the start



//Continued to previous code

① 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;
}

int main () {
Node* head = Null
InsertAtHead (head, 2)
display (head)
InsertAtHead (head, 1)
display (Head)
return 0;
}

Output

2 -> Null

1 -> 2 -> Null

② Void insertAtTail (Node* &head, int Val) {
Node* new_node = new Node (Val)
Node* temp = Head;
while (temp -> next != Null) {
temp = temp -> next;
}
temp -> next = new_Node;
}

Addd node to end

int main() {
InsertAtTail (Head, 3);
display (head);
return 0;
}

1 -> 2 -> 3 -> Null

FREEMIND

Date

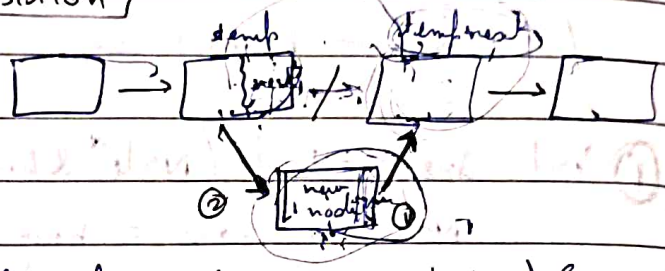
Page

(A) Time Complexity :- O(n)

O(1) {when tail is given}

Add a node at an arbitrary position

// Continued from previous



(3) Void InsertAtPosition (Node* &head, int val, int pos) {

```
if (pos == 0){
        InsertAtHead (head, value)
        return;
}
Node* new_node = new Node (val);
Node* temp = head;
int current_position = 0;
while (current_position != pos-1) {
        temp = temp -> next;
        current_pos++;
}
// temp is pointing to node at pos-1
// new_node -> next = temp -> next;
    temp -> next = new_node;
}
```

```
int main(){
InsertAtPosition
(head, 4, 1);
display (head)
```

(A) Time complexity => O(pos)

O(n) (Worst case)

Date

Page

⇒ Updation at k^{th} position in a Singly linked list

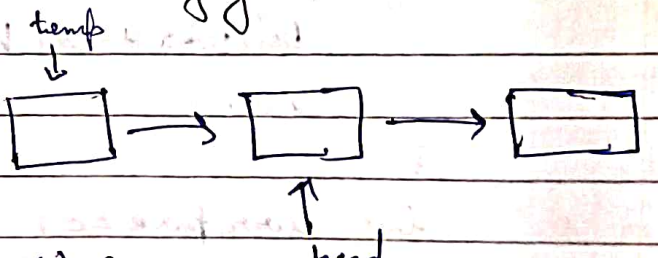
(Continued)

```
Void UpdateAtPos(Node* &head, int k, int Val){

    Node* temp = head
    int currpos = 0
    while(currpos != k){
        temp = temp->next
        currpos++
    }
    // temp is pointing to node at k
    temp->value = Value;
}
int main(){
    UpdateAtPosition(head, 2, 5)
    display(head)
}
```

---

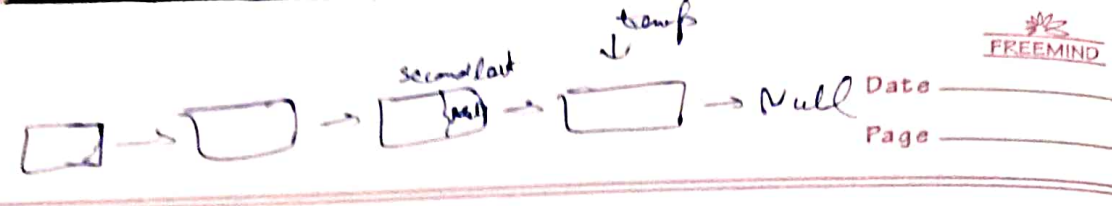
⑧ Deletion at k^{th} Position in a singly linked list.

→ Delete a node at the start




```
Void DeleteAtHead(Node* &head){

    Node* temp = head;
    head = head->next;
    free(temp);
}
int main(){
    DeleteAtHead(head)
    Display(head)
```

Time complexity → O(1)




```
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)
}
```

Time complexity

O(n)

```
// Deletion at Arbitrary Position.
  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 is pointing to node at pos-1
      Node* temp = prev -> next;
          prev -> next = prev -> next -> next;
          free (temp);
     }
```

Time Complexity - O(n)

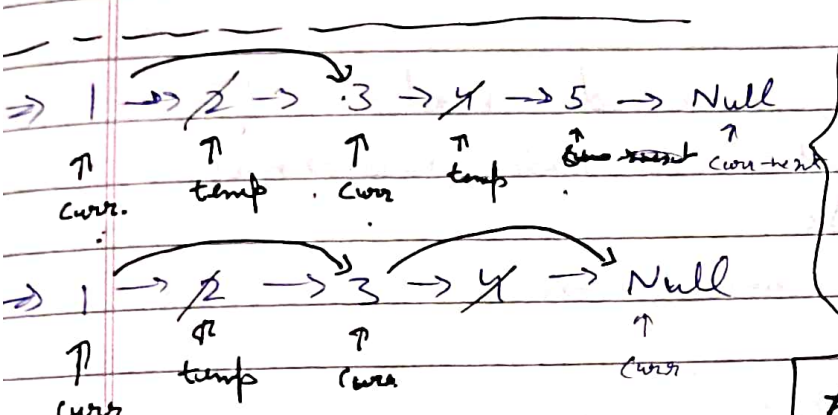
```
int main
DeleteAtPos
   (head, 2)
Display (head)
}
```

FREEMIND

Date

Page

Q: Given the head of a linked list, delete every alternate element from the list starting from the second element.




```
temp = curr->next
curr->next = curr->next->next
free(temp)
curr = curr->next
```

★ Stopping condition
1. curr->next == NULL (odd)
2. curr == NULL (Even)

(Continue from Prev. codes)

Sol:

```
Class Node { }
Class LinkedList {
    public:
        Node* head;
        LinkedList () {
            head = NULL
        }
        Void InsertAtTail (int Val.)
        Node* new_node = new Node(Val);
        if (head == NULL) { //linked list is empty
            head = new_node;
            return;
        }
        Node* temp = head
        while (temp->next != NULL) {
            temp = temp->next;
        }
            temp->next = new_node;
    } }
} Void display { Same prev }
```

```
Void DeleteAlternateNodes(Node* &head){
    Node* currnode = head;
    while(curr_node != NULL && curr_node->next != NULL){
    ~~temp~~
    * Node *temp = curr_node->next;
      Curr_node->next = Curr=node->next->next;
      free(temp);
      Curr_node = Curr_node->reset
    }
}
int main(){
    Linkedlist ll;
    ll.InsertAtTail(1);
    ll.InsertAtTail(2);
    ll.InsertAtTail(3);
    ll.InsertAtTail(4);
    ll.InsertAtTail(5);
    ll.display();
    deleteAlternateNodes(ll.head)
    ll.display();
    return 0;
}
```

Q-21 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.

```
while(curr != NULL){
    while(Curr->next != Null && Curr->value == Curr->next->value){
        // remove curr->next Node as code of Delete Alternate nodes

    } Curr = Curr->next
}
```

1→2→2→3→3→3

1→2→3.

Continued from prev. Ques.

FREEMIND

Date

Page

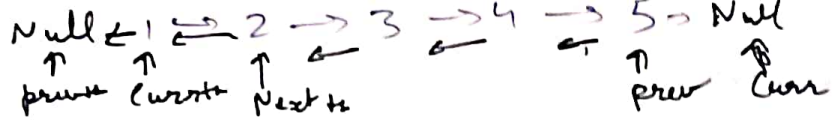
```
Void DeleteDuplicateNodes (Node * & head)
    Node* curr_node = head;
    while (curr_node) {
        while (curr_node->next && curr_node->value == curr_node->next->value)

            Node* temp = curr_node->next;
            curr_node->next = curr_node->next->next;
            free (temp);
        }
        // this loop ends when current node and next node values are different or linked list ends
        curr_node = curr_node->next;
    }
}
int main () {
    ll. _ _
    ll _ _
    DeleteDuplicateNodes (ll.head)
    ll.display ();
    return 0;
}
```

Time complexity O(n)

Q3) Given the head of a singly linked list, and print the reversed list.

Sol:

```
Void reverseprint (Node * head) {
    if (head == NULL) {          // base case
        return;
    }
    reverseprint (head->next)    // recursive food. case
    Cout << head->value << " ";
}
int main () {
    Same
```



FREEMIND

Date

Page

⭐ Reversing a linked list:-

Q: Given the head of a singly linked list, reverse the list, and return the reversed list.

Sol:

```
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;
}
int main () {
    ll.head = reverseLL (ll.head);
    ll.display ();
    return 0;
}
```

Recursively

```
head -> next -> next = head
head -> next = NULL
```

1 → (2 → 3 → 4 → 5)

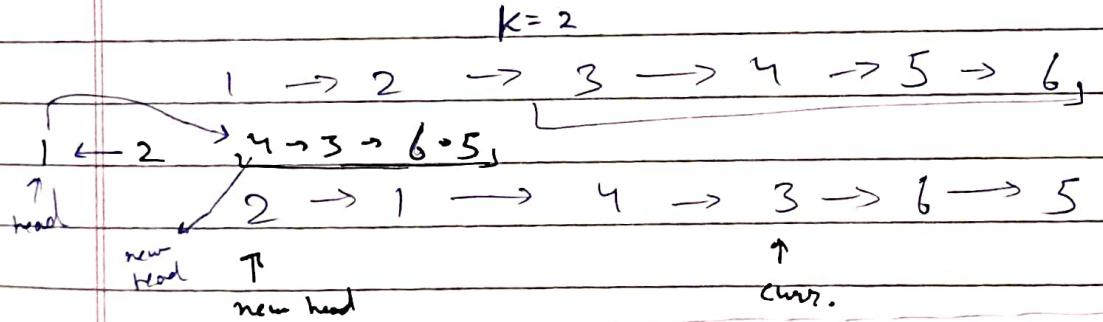
↑ head

NULL ← 1 ← [2 ← 3 ← 4 ← 5]

↑ head ↑ head → next ↑ new_head

```
Node* reverseLLRecursion(Node* &head){
    if(head==NULL || head->next==NULL){
        return head;
    }

    Node* newhead = reverseLLRecursion(head->next);
    head->next->next = head;
    head->next = NULL;
    return newhead;
}
```

Q> Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list.



---

sol→

```
Node* reverseKLL(Node* &head, int k){
    Node* prevptr = NULL;
    Node* currptr = head;
    int counter = 0;                    // Counting 1st k nodes
    while(currptr!=NULL && counter<k){  // reversing 1st k nodes
        Node* nextptr = currptr->next
        currptr->next = prevptr
        prevptr = currptr;
        currptr = nextptr;
        counter++;
    }
    if
```

```
// currptr will give us (k+1)th node
    if (currptr != NULL){
        Node * newhead = reverseKLL(currptr, k); //recursive call
        head -> next = newhead;
    }
    return prevptr; //will give the new head of connected linkedlist
}
int main(){

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

Lecture - 51

P(Linked List (Part-2))

FREEMIND

Date

Page

① Pattern 1 : 2 Pointers

Q-s Given 2 linked lists, Tell if they are equal or not. Two linked lists are equal if they have same data and arrangement of data is also the same.

Sol.

```
bool checkEqualLinkedlist(Node* head1, Node* head2)

    Node* ptr1 = head1;
    Node* ptr2 = head2;

    while(ptr1 != NULL && 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 && ptr2 == NULL);
}
int main(){
    Linked ll1;
    ll1.InsertAtTail(1)
                    (2)
                    (3)
    Linkedlist ll2;
    ll2.InsertAtTail(1)
                    (2)
                    (3)
    ll1.display(); ll2.display();
    cout << checkEqualLinkedList(ll1.head(), ll2.head())
         << endl;
    return 0;
}
```

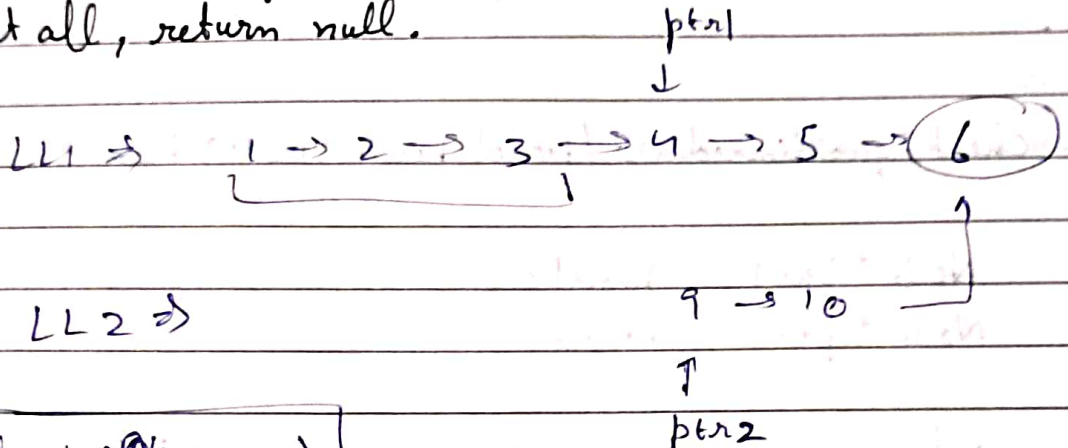
FREEMIND

Date

Page

Q → Given the head of two singly linked-lists head A and head B, return the node at which the two lists intersect. If the two linked lists have no intersection at all, return null.

Sol-



Time complexity → O(2n+m)

```
Node* getIntersection(Node* head1, Node* head2){

    Node* ptr1 = head1;
    Node* ptr2 = head2;
int getLength(Node* head){

    Node* temp = head;
    int length = 0;
    while (temp != NULL)
        length++;
        temp = temp->next;
    }
    return length;
}
Void moveHeadByk(Node* head, int* k){
    Node* ptr = head
    while (k--){
        ptr = ptr->next;
    }
    return ptr;
}
```

Date

Page

```
Node* getIntersection(Node* head1, int k, Node* head2){

    //Step1: Calculate lengths of both linked lists.
    int L1 = getLength(head1)
    int L2 = getLength(head2)

//Find diff. k b/w linked-list and move longer linkedlist ptr by k

    if (L1 > L2){
        int k = L1-L2;
        ptr1 = movehead by k (head1, k);
        ptr2 = head2;
    } else {
        int k = L2-L1;
        ptr1 = head1
        ptr2 = movehead by k (head2, k);
    }
// Step 3: compare ptr1 and ptr2 nodes.
    while (ptr1) {
        if (ptr1 == ptr2) {
            return ptr1;
        }
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }
    return NULL;
}.
int main() {
    linkedlist ll1
    ll1.insertAtTail(1); (2); (3); (4); (5)
    linkedlist ll2
    ll2.insertAtTail(6); (7)
    ll2.head->next->next = ll1.head->next->next->next;
```

Date

Page

```
Node* intersection = get intersection (ll1.head, ll2.head);
if (intersection){
        cout << intersection -> val << endl;
}
else {
    cout << "-1" << endl;
}

return 0;
}
```

Q:1 Given the head of a linked list, remove the k-th node from the end of the list and return its head.

Sol:1

```
Void removeKthNodeFromEnd (Node* head, int k)

    Node * ptr1 = head
    Node * ptr2 = head
    //move ptr2 by k steps ahead
    int count = k
    while (count --) {
        ptr2 = ptr2 -> next;
    }
. if (ptr2 = NULL) { // k is equal to length of linked list.
        //we have to delete head node
        Node * temp = head;
        head = head -> next;
        free (temp);
        return;
    }
    //now ptr2 is k steps ahead of ptr1
    //when ptr2 is at Null (end of list), ptr1 will be at
                                        node to be deleted
```

```
while (ptr2->next != NULL) {
    ptr1 = ptr1->next;
    ptr2 = ptr2->next;
}
Node // Now ptr1 is pointing to the node before kth node from end
// Node to be deleted is ptr1->next
Node* temp = ptr1->next;
    ptr1->next = ptr1->next->next;
    free(temp);

}
int main() {
    ll. —
    ll. — —
    removeKthNode fromEnd (ll.head, 3);
    ll.display();
    return 0;
}
```

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

Sol:-

```
Node* mergeLinkedList (Node* &head1, Node* &head2) {
    Node* DummyHeadNode = new Node(-1);
    Node* ptr1 = head1
    Node* ptr2 = head2
    Node* ptr3 = DummyHeadNode

    while (ptr1 && ptr2) {
        if (ptr1->val < ptr2->val) {
            ptr3->next = ptr1
        } else ptr1 = ptr1->next
```

{ Time Complexity → O(n+m)

```
        else {
            ptr3->next = ptr2
            ptr2 = ptr2->next
        }
    }
    if (ptr1 != Null) {
        ptr3->next = ptr1
    }else{
        ptr3->next = ptr2
    }
        return dummyHeadNode->next;
}
int main(){
    ll1 ~~ ll2 ~~
    LinkedList ll3.
    ll3.head = mergeLinkedLists(ll1.head, ll2.head);
    ll3.display();
    return 0;
};
```

Pattern → Merging Multiple Linked list

Q1 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 and return it.

Concept ⇒ 2 Phle 2 lists ko merge kro and then unse jo list banegi use 3rd k sath merge kara

⇒ Merge 2 lists Code from prev. Question

```
Node* MergeKLinkedLists (Vector<Node*> &lists) {

    if (lists.size() == 0) {
        return NULL;
    }
    while (lists.size()>1) {

        Node* mergehead = mergeLinkedLists (lists[0], lists[1]);
        lists.push_back (mergedhead);
        lists.erase (lists.begin());
        lists.erase (lists.begin());

    }
    return lists[0];
}
int main () {
    LinkedList ll1;
    Linked list ll2;
    Linked list ll3;
    ——  ——  ——

    Vector<Node *> lists
    lists.push_back (ll1.head);
    lists.push_back (ll2.head);
    lists.push_back (ll3.head);

    LinkedList mergedLL;
    mergedLL.head = mergeKLinkedList (lists)
    mergedLL.display();

    return 0;
}
```

FREEMIND

Date

Page

Pattern: Slow Fast Pointer

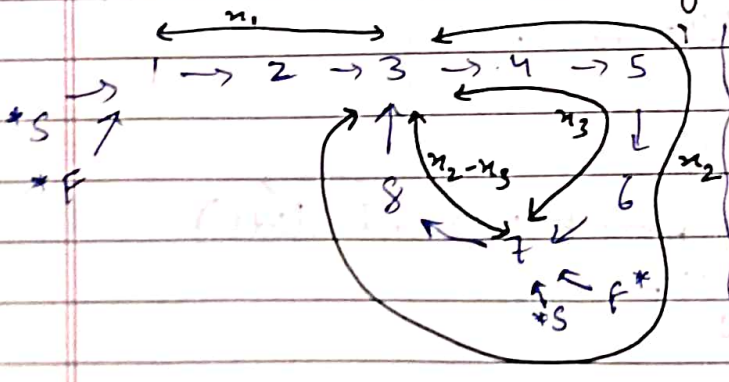
Q1 Find the middle element of the given linked list.

slow ⇒ ptr = ptr → next
fast ⇒ ptr = ptr → next → next

Sol:-

```
Node* findMiddleElement(Node* &head){
    Node* slow = head;
    Node* fast = head;

    while(fast != NULL && fast->next != NULL)
        slow = slow->next
        fast = fast->next->next;
    }
    return slow;
}
int main(){
    Node* middleNode = findMiddleNode(ll.head);
    cout<<middleNode->val<<endl;
    return 0;
}
```

Q2 Given the head of a linked list, determine if the linked list has a cycle in it.



If slow/fast pointer meets at a same node then linkedlist has a cycle in it.

s = s->next
f = f->next->next

By Proving $x_1 = x_2 - x_3$; means, cycle ka starting point head se or pointers ke meeting point se equidistant hai.

Print node val.

FREEMIND

Date

Page

```
bool detectCycle(Node* head)

    if (!head){
        return false;
    }
    Node * slow = head
    Node * fast = head;
    while (fast && fast->next) {

        slow = slow -> next
        Fast = Fast -> next -> next;
        if (slow == fast) { cout << slow -> val << endl;
            return true;
        }
    }
    return false;
}
int main() {
    LinkedList ll;
    ll.InsertAtTail(1);----------(8);
    ll.display();
    ll.head->next->next->next->next->next->next->next->next
                    = ll.head->next->next.
    cout << detectCycle(ll.head) << endl;
    return 0;
}
```

(*) Jab bhi hame Question diya jaega cycle remove karne k toh hum. Slow/Fast me se ek ko head par le jayenge and then dono ko ek ek step se aage badhayenge. Jis point par dono k next me same value hogi vha slow ke next me null point kar denge.

FREEMIND

Date

Page

```
Void removeCycle(Node *&head) {

    Node * slow = head;
    Node * Fast = head;

    do {
    slow = slow -> next;
    Fast = fast -> next -> next;
    } while (slow != Fast);

    fast = head
    while (slow->next != Fast->next) {
        slow = slow->next;
        fast = Fast->next;
    }
        slow->next = NULL;
}
int main() {
    // Continued from prev.
    removeCycle(ll.head);
    cout << "After removing cycle:" << endl;
    cout << detectCycle(ll.head) << endl;
    ll.display();

    return 0;
}
```

Q => Given the head of a linked list, determine if the ll is palindrome or not. 1 -> 2 -> 3 -> 3 <- 2 <- 1 Null

Steps =>
1) Finde Middle element -> slow/Fast Pointer
2) Break linked list into 2.
3) Reverse 2nd half of ll. -> (prev, curr, next)
4) Compare the 2 parts of ll. (head1 (start) { head2 (end) } Compare

Date
Page

```
bool isPalindrome (Node* head) {

① //Find middle element
    Node* slow = head;
    Node* fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    } //now slow is pointing to middle element
② // break the linked list in the middle.
    Node* curr = slow->next; Node* prev = slow;
        slow->next = NULL;
③ // reverse the 2nd half of ll.
    while (curr) {
    Node* nextNode = curr->next;
    curr->next = prev;
    prev = curr
    curr = nextNode
    }
④ //check if the 2 ll. 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;
}
int main() {
    cout << isPalindrome (ll.head) << endl;
}
```

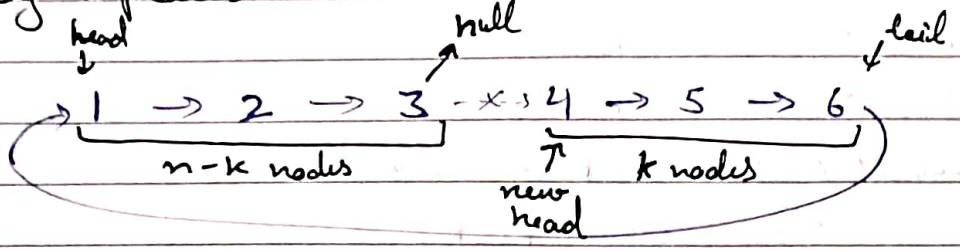
FREEMIND

Date

Page

⊛ Pattern :→ Rearrangement of Nodes in a list.

Q→ Given the head of a ll., Rotate the list to the right by k places.



Steps:→ ① Find n
② Find tail node,
tail → next = head
③ Traverse n-k nodes, $(n-k)^{th}$ node → next = NULL
$(n-k+1)^{th}$ = new head

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

    K = k % n;
    if (k==0) {
        return head;
    }
    tail->next = head
    Node * temp = head;  //③ traverse n-k nodes
    for (i=0; i<n-k; i++) {
        temp = temp->next ;
    }
```

//temp is now pointing to $(n-k)^{th}$ node

```
Node* newhead = temp->next;
    temp->next = NULL;
return newhead
}
int main() {
    LinkedList ll;
    ll.InsertAtTail(1); ---------(6),
    ll.display();
    ll.head = rotatebyk(ll.head, 3)
    ll.display();
    return 0;
}
```

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

Sol:-

```
Node* oddEvenLinkedList(Node*& head) {
    if (!head) {
        return head;
    }
    Node* evenhead = head->next;
    Node* oddptr = head;
    Node* evenptr = evenhead;
    while (evenptr && evenptr->next) {
        oddptr->next = oddptr->next->next;
        evenptr->next = evenptr->next->next;
        oddptr = oddptr->next;
        evenptr = evenptr->next;
    }
```

Date

Page

// temp is now pointing to $(n-k)^{th}$ node

```
    Node* newhead = temp->next;
        temp->next = NULL;
    return newhead
}
int main() {
    LinkedList ll;
    ll.InsertAtTail(1); ---------- (6),
    ll.display();
    ll.head = rotatebyk(ll.head, 3)
    ll.display();
    return 0;
}
```

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

Sol:-

```
Node* oddEvenLinkedList(Node*& head) {

    if (!head) {
        return head;
    }
    Node* evenhead = head->next;
    Node* oddptr = head;
    Node* evenptr = evenhead;
    while (evenptr && evenptr->next) {
        oddptr->next = oddptr->next->next;
        evenptr->next = evenptr->next->next;
        oddptr = oddptr->next;
        evenptr = evenptr->next;
    }
```

Time Complexity = O(n)

Date

Page

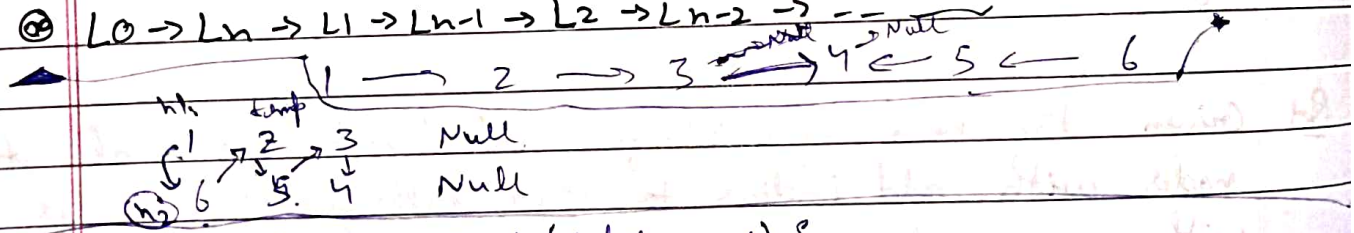
```
    oddptr->next = evenhead;
    return head;
}

int main() {
    ll.head = oddevenlinkedlist(ll.head)
    ll.display
    return 0;
}
```

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 → ...



Sol.:-

```
Node* reorderedlinkedlist(Node* &head) {
    //1. Finding the middle element
    Node* slow = head;
    Node* Fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    // now slow is pointing to the middle element
    //2. seperate the linked list and reverse
       the 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; //LL 1st half.
    Node* ptr2 = prev; //LL 2nd half.

    while( ptr1 != ptr2) {
        Node* temp = ptr1->next
            ptr1->next = ptr2;
            ptr1 = ptr2;
            ptr2 = temp;
    }
    return head;
}
int main() {
    ll.head = reorderedLinkedList(ll.head)
    ll.display();
    return 0;
}
```

Q> Given a linked list, swap every two adjacent nodes and return its head. You may not modify the values in the list nodes. only nodes themselves may be changed.

head → 1 → 2 → [3 → 4 → 5 → 6] Recursion

second Node

2 → 1 → 4 → 3 → 6 → 5

new head

Date

Page

Sol:

```
Node* swapAdjacent(Node* &head){

    //base case
    if (head == NULL || head->next == NULL){
        return head;
    }
    //recursive case
    Node* secondNode = head->next
    head->next = swapAdjacent(secondNode->next);
    secondNode->next = head; //reversing the link
                             b/w 1st and 2nd Node;
    return secondNode;

}
int main(){
    ll.head = swapAdjacent(ll.head);
    ll.display();
    return 0;
}
```

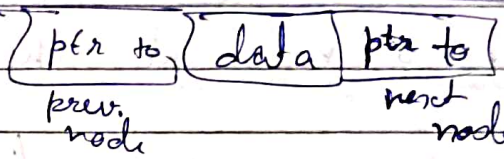
Lecture - 52

FREEMIND

Date

```
struct node {
    int info;
    struct node *prev;
    struct node *next;
}
```

⇒ What is a Doubly linked list?



⇒ **Advantages**

→ traversal → both ways

→ Insertion / Deletion becomes more efficient

⇒ **Disadvantage**

→ Extra space for previous ptr.

#Code

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

class Node {
    public:
    int val;
    Node* prev;
    Node* next;

    Node (int data) {
        val = data;
        prev = null;
        next = null;
    }
};
class doublylinkedlist {
    public:
        Node* head;
        Node* tail;
        doublylinkedList () {
            head = NULL
            tail = NULL
        }
};
```

```
int main () {
    Node* new_node = new Node(3)

    doublylinkedlist dll;
    dll.head = new_node;
    dll.tail = new_node;
    cout << dll.head->val << endl;

    return 0;
}
void display () {
    same
}
```

FREEMIND

Date

Page

⊛ Traversal in doubly linked list

⇒ Insertion at k^{th} position in a Doubly linked list.

⊙ Add a node at the start

```
Void insertAtStart (int Val)

    Node* new_node = new Node(Val)
    if (head == NULL)
        head = new_node;
        tail = new_node
        return;
    }
    new_node -> next = head
    head -> prev = new_node
    head = new_node;
    return;
}
```

```
int main
    dll.insertatstart()   (2)
    ;                     (3)
    {
    dll display();
}
```

⊙ Add a node at end

```
Void insertAtEnd (int val) {
    Node* new_node = new Node(val);
    if (tail == NULL) {
        head = new_node;
        tail = new_node;
        return;
    }
    tail -> next = new_node;
    new_node -> prev = tail;
    tail = new_node;
    return;
}
```

FREEMIND

Date

Page

O(n) - O(k)

⊙ Add a node at an arbitrary position

```
Void insertatPosition(int val, int k)
    // assuming k is less or equal to length of dll
    Node * temp = head;
    int count = 1;
    while (count < (k-1)){
        temp = temp -> next;
        count++;
    }

    Node * new_node = new Node(val);
    new_node -> next = temp -> next;
    temp -> next = new_node;

    new_node -> prev = temp;
    new_node -> next -> prev = new_node;
    return
}
```

```
int main(){
dll.insertatPosition(33
dll.display();
return 0;
}
```

⇒ Deletion at k^{th} Position in a doubly linked list.

```
⊙ Void deleteAt_Start(){
    if (head == NULL){
        return;
    }
    Node * temp = head;
    head = head -> next
    if (head != Null) {
        head -> prev = NULL};
    }
    free(temp);
    return;
}
```

```
if(head == NULL){
    tail = NULL;
}else{
    head -> prev = NULL
}
```

Date

Page

⑤ Delete At End

```
Void deleteAt_End ()
    if (head == NULL) {
        return;
    }
    Node* temp = tail;
    tail = tail -> prev;
    if (tail == NULL) {
        head = NULL;
    } else {
        tail -> next = NULL;
    }
    free (temp);
    return;
}
```

⑥ Add a node at Arbitrary position O(k)

```
Void deleteAtPosition (int k) {
    // k <= length of ll.
    Node * temp = head;
    int count = 1;
    while (count < k) {
        temp = temp -> next;
        count ++;
    }
    //Now temp is pointing to node at kth position.
    temp -> prev -> next = temp -> next;
    temp -> next -> prev = temp -> prev;
    free (temp);
    return;
}
```

FREEMIND

Date

Page

Q1) Reverse a doubly linked list.

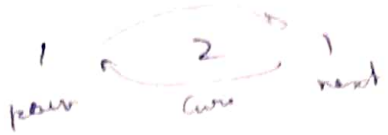
Sol:

```
Void reverseLL(Node*&head, Node*&tail) {

    Node * Currptr = head;
    while (Currptr) {
        Node* nextptr = Currptr->next;
        Currptr->next = Currptr->prev;
        Currptr->prev = nextptr;
        Currptr = nextptr;
    }
    // Swapping head and tail pointers
    Node* newhead = tail;
    tail = head;
    head = newhead;

}
```

Q2) Find it is a palindrome or not. (DLL)

Sol:

```
Bool isPalindrome(Node*head, Node*tail) {
    while (head != tail && tail != head->prev) {
        if (head->val != tail->val) {
            return false;
        }
        head = head->next;
        tail = tail->prev;
    }
    return true;
}
```



Date

Page

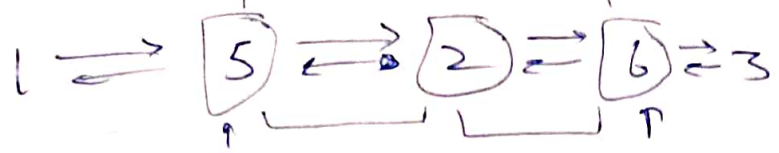
Q3→ Given head (DLL), delete the nodes whose neighbours has the same value. Traverse the list from right to left.

Sol→

```
Void deleteSameNeighbourNode (Node* &head, Node *&tail) {

    Node* currPtr = tail -> prev;

    while (currPtr != head) {
        Node* prevNode = currPtr -> prev;
        Node* nextNode = currPtr -> next;
        if (prevNode == nextNode) {
            //Delete currNode
            prevNode -> next = nextNode;
            nextNode -> prev = prevNode;
            free (currPtr);
        }
        currPtr = prevNode;
    }
    return;
}
```

Q→ A critical point in a linked list is defined as either a local maxima or a local minima. Given a linked list head, return an array of length 2 containing [minDistance, maxDistance] where minDistance is the minimum distance b/w any two distinct ~~co~~ critical points and maxDistance is the maximum distance b/w any two distinct critical points. If there are fewer than two critical points, return [-1, -1].

Note that a node can only be a local maxima/minima if there ~~ex~~ exists both a previous node and a next node.



FREEMIND

Date

Page

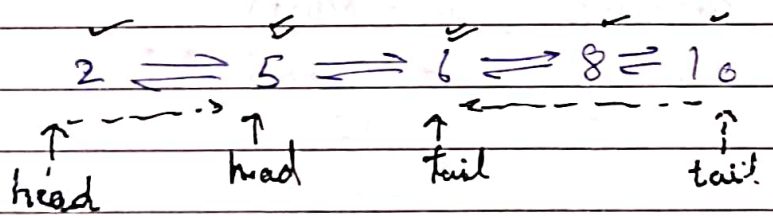
Sol:-

```
bool isCriticalPoint(Node* &currNode){
    if(currNode->prev->val < currNode->val && currNode->next->val < currNode->val)
        return true;
    if(currNode->prev->val > currNode->val && currNode->next->val > currNode->val)
        return true;
}

Vector<int> distanceBwCriticalPoints(Node* head, Node* tail){

    Vector<int> ans(2, INT_MAX);
    Vector
    int firstCP = -1, lastCP = -1;

    Node* currNode = tail->prev;
    if(currNode == NULL){
        ans[0] = ans[1] = -1;
    return ans;
    } int currPos = 0
    while(currNode->prev != NULL){
        if(isCriticalPoint(currNode)){
            if(firstCP == -1){
                firstCP = lastCP = currPos;
            }else{
                ans[0] = min(ans[0], currPos - lastCP);
                ans[1] = currPos - firstCP
                lastCP = currPos;
            }
        }
        currPos++;
        currNode = currNode->prev;
    }
```

```
    if (ans[0] == INF_MAX) {
        ans[0] = ans[1] = -1;
    }
    return ans;
}
int main() {                                        dll.tail
    vector<int> ans = distance b/w critical Points (dll.head, ↑);
    cout << ans[0] << " " << ans[1] << endl;
    return 0;
}
```

Q → Given the head of a dll. The values of the linked list are sorted in non-decreasing order. Find if there exists a pair of distinct nodes such that the sum of their values is x. Return the pair in the form of a vector [l, r], where l and r are the values stored in the two nodes pointed by the pointers. If there are multiple such pairs return any of them. If there is no such pair return [-1, -1].

(x = 11)



Sol →

```
vector<int> pairSumDLL (Node* head, Node* tail, int x) {
    vector<int> ans(2, -1);
    while (head != tail) {
        int sum = head->val + tail->val;
        if (sum == x) {
            ans[0] = head->val;
            ans[1] = tail->val;
            return ans;
```

```
        if (sum > x) { // I need smaller values, i will move tail back.
            tail = tail -> prev;
        } else { // i need bigger values, i will move head forward.
            head = head -> next;
        }
    }
    return ans;
}
int main() {
    vector<int> ans = pairSumDLL(dll.head, dll.tail, 11);
    cout << ans[0] << " " << ans[1] << endl;
    return 0;
}
```

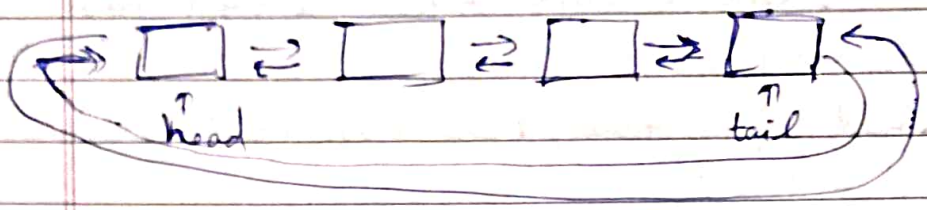
Lecture → 53

(Linked list (Part-4))

Date

Page

Circular Linked list.



tail → next = head

⇒ Advantages

→ We can start traversing from any node to print all nodes untill we reach visited node.

→ When we have to traversed linked list in circular fashion multiple time.

⇒ Implementation of Node → Same

⇒ Traversal

```
temp = head;
while (temp -> next != head) {
temp = temp -> next;
}
```

⇒ Insertion at k^{th} position in a circular linked list.

⊙ Add a node at the start

Use do{ }while(temp!=head) at time of display

```
class Node {
};
class CircularLinkedList {          //Same as Prev
};
Void insertAtStart (int val) {
    Node* new_node = new Node(val);
    if (head == NULL) {
        head = new_node;
        new_node -> next = head; //Circular linked list
        return;
    }
```

These all fxns are passed inside class so, head is not passed in the fxn.

Date

Page

```
Node* tail = head
while (tail->next != head) {
    tail = tail->next;
}
//now tail is pointing to the last node
tail->next = new_node;
new_node->next = head;
head = new_node
}

void printCircular {
    Node* temp = head;
    for (i=0; i<15; i++) {
        cout << temp->val << "->";
        temp = temp->next;
    } cout << endl;
}
int main()
    circularlinkedlist cll;
    cll.insertAtStart(3)---(1)
    cll.printCircular();
    return 0;
}
```

① Add a node at end

Same as Prev.

```
tail->next = new_node;
new_node->next = head;
```

// head will remain same.

② Add Node at Arbitrary Position

Same as Singly LL.

FREEMIND

Date

Page

⇒ Deletion at K^{th} Position

①
```
Void DeleteAtStart ( ) {
    if (head == NULL) {
        return;
    }
    Node * temp = head;
    Node * tail = head;
    while (tail -> next != head) {
        tail = tail -> next;
    }
    head = head -> next;
    tail -> next = head;
    free (temp);
}
```

②
```
Void DeleteAtEnd ( ) {
    if (head == NULL) {
        return
    }
    Node * tail = head;
    while (tail -> next -> next != head) {
        tail = tail -> next
    }
    // here tail is pointing to second last
    Node * temp = tail -> next; // to be deleted
    tail -> next = head;
    free (temp);
}
```

③ Delete at arbitrary Position.

Same as Singly LL.

FREEMIND

Date

Page

⭐ Template Classes :-

→ they allow us to pass data type as parameter

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

template <typename T>
class Node{ // Template class
  public:
    T val;
    Node* next;
  Node(T data){
      val = data
      next = NULL
  }
};
int main(){
    Node<int>* node1 = new Node<int>(3);
    cout << node1 -> val << endl;

    Node<char>* node2 = new Node<char>('a')
    cout << node2 -> val << endl;
    return 0;
}
```

⭐ STL :->

(Standard Template Library)

→ set of template classes for implementing commonly used data structures & functions.

eg: vector<int>
<char>

lists
set
map
queue
stack

◉ 3 major components
→ containers
→ Iterators
→ Algorithms

FREEMIND

Date

Page

⇒ **Containers:** used to hold data of the same type

eg:- Vector:- Class that defines a dynamic array.
list → class for doubly linked list.
map → used to store unique key value pairs
Set ⇒ used to store unique values

⇒ **Iterators:** → used to iterate / traverse the container.
→ pointer like entities.

```
Vector <int> v = {1, 2, 3, 4, 5}
Vector <int> :: iterator itr;

itr. began() = 1
```

⇒ **Algorithms:**
functions that are used to manipulate data in the containers

eg:

```
sort()  ---> sort (v.began(), v.end())
min()   ---> min (a,b)
max()   ---> max (a,b)
```

C++.com for these funct./Templates

Ⓐ What is a List?
→ template class in STL for implementing doubly linked list.

⇒ **Declaration of a List**

```
# include <list>
list <datatype> list name
list <int> rollNo;
list <int> list1 {1, 2, 3, 4}
list <int> list2 {5, 6, 7, 8}
```

FREEMIND

Date

Page

⇒ Advantage of list in C++ STL :-

→ implementation becomes easy

⇒ Iterator functions

list.begin() → iterator for 1st element [1, 2, 3, 4] (↑ 1)

list.end() → iterator for the position after last element. [1, 2, 3, 4] (↑ after 4)

list.rbegin() → iterator for the first element in reverse iteration [1, 2, 3, 4] (↑ 4)

list.rend() → iterator for the position after last element in reverse iteration ↑ [1, 2, 3, 4]

advance(itr, n) → advances the itr by n places.

(itr, 1) → [1, 2, 3, 4] (↑ 1, ↑ 2)

```
#include <iostream>
#include <list>
Using namespace std;
int main(){
    list<int> l1 = {1,2,3,4};
    auto itr = l1.begin();
    cout<<*itr<<endl;                    //1
    auto rev_itr = l1.rbegin();
    cout<<*rev_itr<<endl;                //4
    advance(itr,2);
    cout<<*itr<<endl;                    //3
    //Using Range based loop.
    for(auto num:l1){                    //traversal of list.
        cout<<num<<endl;
    }
//Using iterators  for(auto itr = l1.begin(); itr!=l1.end(); itr++){
        cout<<*itr<<" ";
    }cout<<endl;
```

FREEMIND

Date

Page

```
// reverse traversal
for (auto itr = l1.rbegin(); itr != l1.rend(); itr++) {
    cout << *itr << " ";
}
}
```

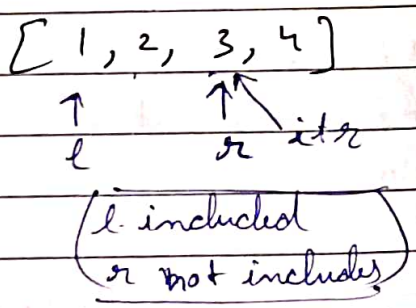
⭐ Inserting elements into a list

list.insert(itr, Value) → insert value before the position of the itr.

list.insert(itr, count, value) → insert value count no. of times before itr.

list.insert(itr, str_itr, end_itr) → insert values from str_itr to end_itr before itr.

eg: //continued from prev. --




```
→ auto itr = l1.begin();
advance(itr, 2);
auto l = l1.begin();
auto r = l1.begin();
advance(r, 2);
l1.insert(itr, l, r);        //[1,2,1,2,3,4]
for (auto itr = l1.begin(); itr != l1.end(); itr++) {
    cout << *itr << " ";
} cout << endl;
```

⭐ Deleting elements from a list.

list.erase(itr) → delete the element pointed to by the itr.

list.erase(str_itr, end_itr) → delete elements from str_itr to end_itr.

⇒ Other member function of a list container.

push_front(Val) → insert val in the front of list
pop_front() → removes val from front of list
push_back(Val) → insert val in the back of the list.
pop_back() → removes val from back of list.