Assignment – 1

1. **Design the logic to remove the duplicate elements from an Array and after the deletion the array should contain the unique elements.**
2. Create an auxiliary array temp[] to store unique elements.
3. Traverse input array and one by one copy unique elements of arr[] to temp[]. Also keep track of count of unique elements. Let this count be **j**.
4. Copy **j** elements from temp[] to arr[] and return j
5. **Implement the logic to a) Reverse the elements of an array, b) Find the matrix multiplication c) Find the Transpose of a Matrix.**
6. **Reverse the elements of an array**

Initialize i := 0, when i < quotient of n/2, update (increase i by 1), do:

* temp := arr[i]
* arr[i] := arr[n - i - 1]
* arr[n - i - 1] := temp

1. **Matrix multiplication**
2. Enter the value of m and n (or) order of the first matrix.
3. Enter the value of p and q (or) order of the second matrix.
4. Create a matrix of size a[m][n] and b[p][q].
5. Enter the element of matrices row wise using loops.
6. If a number of columns of first matrix is not equal to the number of rows of second matrix, print matrix multiplication is not possible and exit. If not, proceed to next step.
7. Create a third matrix, c of size m x q to store the product.
8. Set a loop from i=0 to i=m.
9. Set an inner loop for the above loop from j=0 to j=q.
10. Initialise the value of element (i, j) of new matrix to 0.
11. Set an inner loop inside the above loop from k=0 to k=p.
12. Using the add and assign operator (+=) store the value of a[i][k] \* b[k][j] in the third matrix, c[i][j].
13. Print the third matrix.
    1. **Transpose of matrix**

Transpose of a matrix is obtained by changing rows to columns and columns to rows. In other words, transpose of A[][] is obtained by changing A[i][j] to A[j][i].

void transpose(int A[][N], int B[][M])

{

    int i, j;

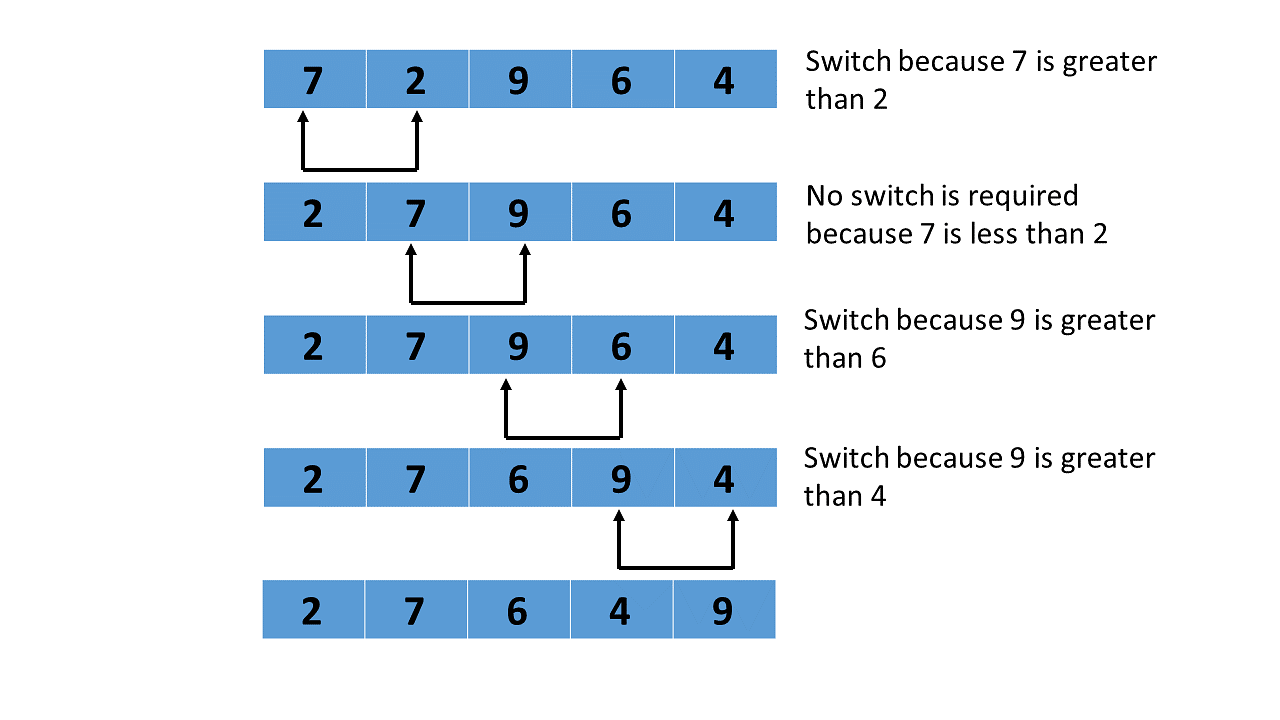
    for (i = 0; i < N; i++)

        for (j = 0; j < M; j++)

            B[i][j] = A[j][i];

}

1. **Implement the Binary search algorithm regarded as a fast search algorithm with run-time complexity of Ο(log n) in comparison to the Linear Search.**
2. Find **middle index** using **(first+last)/2**
3. If element at **middle index** = **number**, print middle index
4. If element at **middle index** > **number**, **(last) = (middle index) - 1**. Go to first step
5. If element at **middle index** < **number**, **(first) = (middle index) + 1**. Go to first step
6. Continue until second step gets evaluated or middle index becomes zero
7. Print number not found, if middle index becomes zero
8. **Design the Logic for Bubble Sort and Insertion sort algorithm.**



**void bubbleSort(int arr[], int n)**

**{**

**int i, j;**

**bool swapped;**

**for (i = 0; i < n - 1; i++) {**

**swapped = false;**

**for (j = 0; j < n - i - 1; j++) {**

**if (arr[j] > arr[j + 1]) {**

**swap(arr[j], arr[j + 1]);**

**swapped = true;**

**}**

**}**

**// If no two elements were swapped by inner loop, then break**

**if (swapped == false)**

**break;**

**}**

**}**

// Function to print an array

void printArray(int arr[], int size)

{

    int i;

    for (i = 0; i < size; i++)

        cout << " " << arr[i];

}

// Driver program to test above functions

int main()

{

    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };

    int N = sizeof(arr) / sizeof(arr[0]);

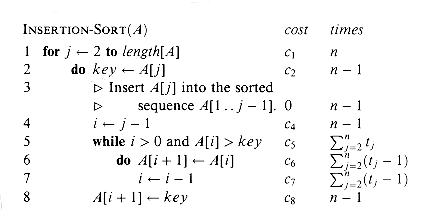
    bubbleSort(arr, N);

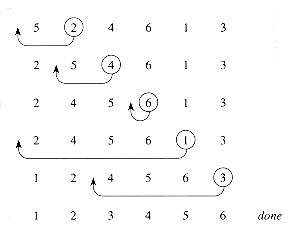
    cout << "Sorted array: \n";

    printArray(arr, N);

    return 0;

}





1. **Design the Logic to Find a Missing Number in a Sorted Array.**

**Input :** arr[] = [1, 2, 3, 4, 6, 7, 8]  
**Output :** 5

**Input :** arr[] = [1, 2, 3, 4, 5, 6, 8, 9]  
**Output :** 7

**Efficient approach** is based on the divide and conquer algorithm that we have seen in binary search, the concept behind this solution is that the elements appearing before the missing element will have ar[i] – i = 1 and those appearing after the missing element will have ar[i] – i = 2.

int search(int ar[], int size)

{

    // Extreme cases

    if (ar[0] != 1)

        return 1;

    if (ar[size - 1] != (size + 1))

        return size + 1;

    int a = 0, b = size - 1;

    int mid;

    while ((b - a) > 1) {

        mid = (a + b) / 2;

        if ((ar[a] - a) != (ar[mid] - mid))

            b = mid;

        else if ((ar[b] - b) != (ar[mid] - mid))

            a = mid;

    }

    return (ar[a] + 1);

}

int main()

{

    int ar[] = { 1, 2, 3, 4, 5, 6, 8 };

    int size = sizeof(ar) / sizeof(ar[0]);

    cout << "Missing number:" << search(ar, size);

}

1. **Write a program to find sum of every row and every column in a two-dimensional array.**

void row\_sum(int arr[m][n])

{

    for (i = 0; i < m; ++i) {

        for (j = 0; j < n; ++j) {

             sum = sum + arr[i][j];

        }

         cout<< "Sum of the row "<< i << " = " << sum<< endl;

        sum = 0;

    }}

void column\_sum(int arr[m][n])

{

         for (i = 0; i < m; ++i) {

         for (j = 0; j < n; ++j) {

             sum = sum + arr[j][i];

        }

        cout<< "Sum of the column "<< i << " = " << sum<< endl;

        sum = 0;

   }}

1. **Space required to store any two-dimensional array is r X c. Assuming array is used to store elements of the following matrices, implement an efficient way that reduces the space requirement.**

**(a) Diagonal Matrix.**

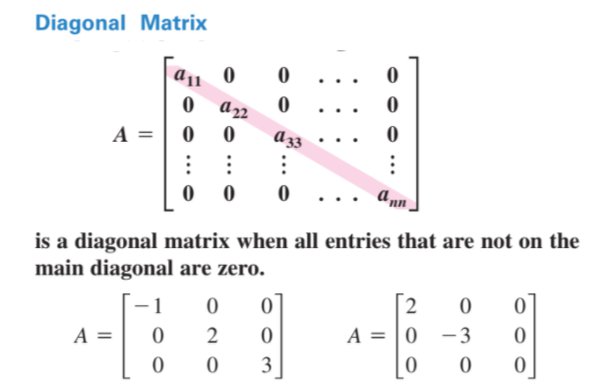
**(b) Tri-diagonal Matrix.**

**(c) Lower triangular Matrix.**

**(d) Upper triangular Matrix.**

**(e) Symmetric Matrix**

**(a) Diagonal Matrix.**



cout<<"Diagonal Matrix (Left To Right) => "<<endl;

for (i=0;i<nr;i++)

{

for(j=0;j<nc;j++)

{

if(i==j)

cout<<A[i][j]<<"\t";

else

cout<<0<<"\t";

}

cout<<endl;

}

cout<<"Diagonal Matrix (Right To Left) => "<<endl;

for (i=0;i<nr;i++)

{

for(j=0;j<nc;j++)

{

if(i+j==nr-1)

cout<<A[i][j]<<"\t";

else

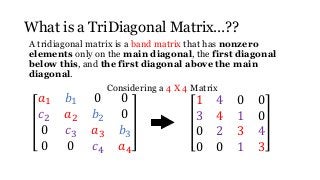
cout<<0<<"\t";

}

cout<<endl;

}

**Tri-diagonal Matrix.**



cout<<"Tridiagonal Matrix => "<<endl;

for (i=0;i<nr;i++)

{

for(j=0;j<nc;j++)

{

if(i-j==-1||i-j==0||i-j==1)

cout<<A[i][j]<<"\t";

else

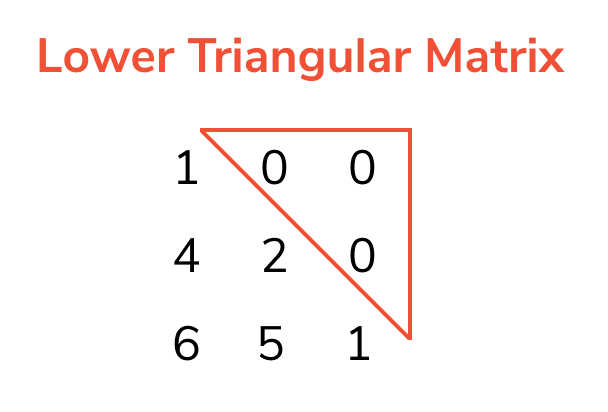
cout<<0<<"\t";

}

cout<<endl;

}

**Lower triangular Matrix.**



cout<<"Lower Triangular Matrix =>"<<endl;

for(i=0;i<nr;i++)

{

for(j=0;j<nc;j++)

{

if(i>=j)

cout<<A[i][j]<<"\t";

else

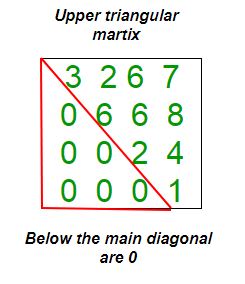
cout<<0<<"\t";

}

cout<<endl;

}

**Upper triangular Matrix.**



cout<<"Upper Triangular Matrix =>"<<endl;

for(i=0;i<nr;i++)

{

for(j=0;j<nc;j++)

{

if(i<=j)

cout<<A[i][j]<<"\t";

else

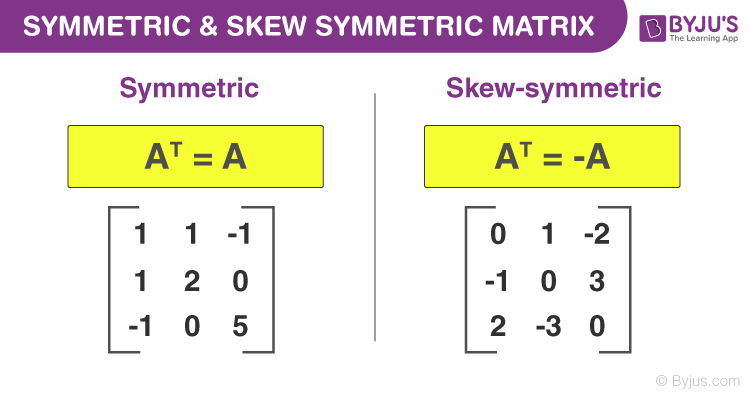
cout<<0<<"\t";

}

cout<<endl;

}

**Symmetric Matrix**



1. **Write a program to implement the following operations on a Sparse Matrix, assuming the matrix is represented using a triplet.**

**(a) Transpose of a matrix.**

**(b) Addition of two matrices.**

**(c) Multiplication of two matrices.**

/\*checking sparse of matrix\*/

   for(i = 0; i < row; i++){

      for(j = 0; j < col; j++){

         if(a[i][j] == 0)

            count++;

      }

   }

   if(count > ((row \* col)/2))

      printf("Matrix is a sparse matrix \n");

   else

      printf("Matrix is not sparse matrix\n");

1. **Write a program to find a saddle point in a two-dimensional array. A saddle point in a numerical array is a number that is larger than or equal to every number in its column, and smaller than or equal to every number in its row.**

*Traverse all rows one by one and do the following for every row i.*

1. *Find the minimum element of the current row and store the column index of the minimum element.*
2. *Check if the row minimum element is also maximum in its column. We use the stored column index here.*
3. *If yes, then saddle point else continues till the end of the matrix.*

int A[10][10], i, j, saddle, p, large, f=1, nr, nc;

for(i=0;i<nr;i++)

{

p=0; saddle=A[i][0];

for(j=0;j<nc;j++)

{

if(saddle>=A[i][j])

{ saddle=A[i][j]; p=j;

}

}

large=0;

for (j=0;j<nc;j++)

{

if (large<=A[j][p])

{ large=A[j][p]; }

}

if(saddle==large)

{ cout<<endl<<"Saddle Point => "<<saddle<<endl;

f=0;

}

}

if (f>0)

{ cout<<endl<<"No Saddle Point."<<endl; }

1. **https://www.interviewbit.com/problems/spiral-order-matrix-i/**

Given a matrix of m \* n elements (m rows, n columns), return all elements of the matrix in spiral order.

**Example:**

Given the following matrix:

[ 1, 2, 3 ],

[ 4, 5, 6 ],

[ 7, 8, 9 ]

You should return [1, 2, 3, 6, 9, 8, 7, 4, 5]

1. **https://www.interviewbit.com/problems/spiral-order-matrix-ii/**

Given an integer **A**, generate a square matrix filled with elements from **1** to **A2** in **spiral order**.

**Input Format:**

The first and the only argument contains an integer, A.

**Output Format:**

Return a 2-d matrix of size A x A satisfying the spiral order.

**Constraints:** 1 <= A <= 1000

**Examples:** Input 1: A = 3

Output 1: [ 1, 2, 3 ],

[ 8, 9, 4 ],

[ 7, 6, 5 ]

**Single Linked List**

**1. Develop a menu driven program for the following operations of on a Singly Linked List.**

**(a) Insertion at the beginning.**

**(b) Insertion at the end.**

**(c) Insertion in between (before or after a node having a specific value, say 'Insert a new Node 35 before/after the Node 30').**

**(d) Deletion from the beginning.**

**(e) Deletion from the end.**

**(f) Deletion of a specific node, say 'Delete Node 60').**

**(g) Search for a node and display its position from head.**

**(h) Display all the node values.**

class Node {

public:

    int data;

    Node\* next;

};

void printList(Node\* n)

{

    while (n != NULL) {

        cout << n->data << " ";

        n = n->next;

    }

}

int main()

{

    Node\* head = NULL;

    Node\* second = NULL;

    Node\* third = NULL;

    // allocate 3 nodes in the heap

    head = new Node();

    second = new Node();

    third = new Node();

    head->data = 1; // assign data in first node

    head->next = second; // Link first node with second

    second->data = 2; // assign data to second node

    second->next = third;

    third->data = 3; // assign data to third node

    third->next = NULL;

    printList(head);    return 0; }

struct node

{

int data;

struct node\* next;

};

struct node \*head = NULL;

node\* create\_node(int);

void insert\_begin();

void insert\_pos();

void insert\_end();

void delete\_pos();

void search();

void display();

int main()

{

create\_node(5); create\_node(8);

int choice1,choice2;

while (1)

{

cout<<"Operations To Be Performed"<<endl;

cout<<"1.Insert Element"<<endl;

cout<<"2.Delete Element"<<endl;

cout<<"3.Search Element"<<endl;

cout<<"4.Display Linked List"<<endl;

cout<<"5.Exit "<<endl;

cout<<endl;

cout<<"Enter Your Choice => ";

cin>>choice1;

cout<<endl;

switch(choice1) {}

}

Outside main () function

node\* create\_node(int num)

{

struct node\* temp;

temp = new node;

if (temp == NULL)

{

cout<<"Memory Not Allocated "<<endl;

return 0;

}

else

{

temp->data = num;

temp->next = NULL;

return temp;

}

}

void display()

{

struct node\* temp;

if (head == NULL)

{ cout<<"The Linked List Is Empty"<<endl; return;

}

temp = head;

cout<<"The Linked List => "<<endl;

while (temp != NULL)

{

cout<<temp->data<<"\t";

temp = temp->next;

}

cout<<endl;

}

void search()

{

int num, pos = 0;

bool flag = false;

if (head == NULL)

{

cout<<"The Linked List Is Empty"<<endl; return;

}

cout<<"Enter The Element To Be Searched => ";

cin>>num;

struct node \*s;

s = head;

while (s != NULL)

{

pos++;

if (s->data == num)

{

flag = true;

cout<<"The Element Is Found At Position "<<pos<<endl;

}

s = s->next;

}

if (!flag)

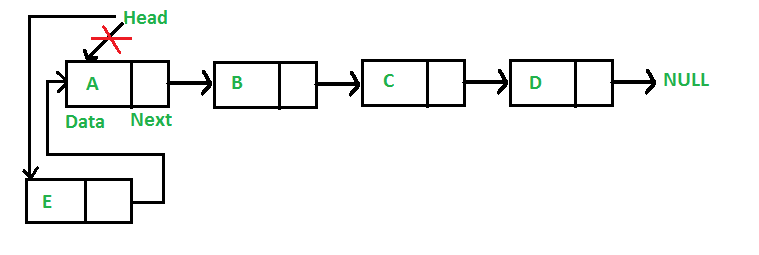
{ cout<<"The Element Is Not In The List"<<endl;

}

}

To insert a node at the start/beginning/front of a Linked List,

* Make the first node of Linked List linked to the new node
* Remove the head from the original first node of Linked List
* Make the new node as the Head of the Linked List.



void push(Node\*\* head\_ref, int new\_data)

{

   Node\* new\_node = new Node();

new\_node->data = new\_data;

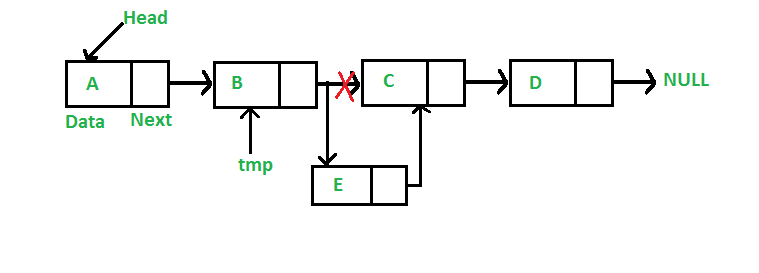
   new\_node->next = (\*head\_ref);

   (\*head\_ref) = new\_node;

}

To insert a node after a given node in a Linked List,

* Check if the given node exists or not.
  + If it do not exists,
    - terminate the process.
  + If the given node exists,
    - Make the element to be inserted as a new node
    - Change the next pointer of given node to the new node
    - Now shift the original next pointer of given node to the next pointer of new node



void insertAfter(Node\* prev\_node, int new\_data)

{

    if (prev\_node == NULL) {

        cout << "The given previous node cannot be NULL";

        return;

    }

     Node\* new\_node = new Node();

     new\_node->data = new\_data;

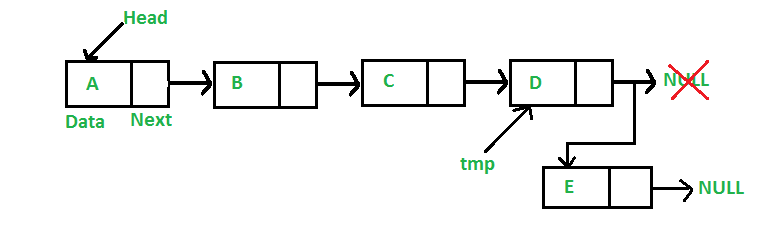
     new\_node->next = prev\_node->next;

     prev\_node->next = new\_node;

}

To insert a node at the end of a Linked List,

* Go to the last node of the Linked List
* Change the next pointer of last node from NULL to the new node
* Make the next pointer of new node as NULL to show the end of Linked List

 void append(Node\*\* head\_ref, int new\_data)

{

    Node\* new\_node = new Node();

    Node\* last = \*head\_ref;

    new\_node->data = new\_data;

    new\_node->next = NULL;

     if (\*head\_ref == NULL) {

        \*head\_ref = new\_node;

        return;

    }

     while (last->next != NULL) {

        last = last->next;

    }

   last->next = new\_node;

    return;

}

To delete a node from front of a Linked List,

Node\* removeFirstNode(struct Node\* head)

{

    if (head == NULL)

        return NULL;

    // Move the head pointer to the next node

    Node\* temp = head;

    head = head->next;

    delete temp;

    return head;

}

**2. Write a program to count the number of occurrences of a given key in a singly linked list and then delete all the occurrences. For example, if given linked list is 1->2->1->2->1->3->1 and given key is 1, then output should be 4. After deletion of all the occurrences of 1, the linked list is 2->2->3.**

Step 1: Start

Step 2: Create A Function Of A Linked List, Pass A Number As Arguments And Provide The Count Of The Number To The Function.

Step 3: Initialize Count Equal To 0.

Step 4: Traverse In Linked List Until Equal Number Found.

Step 5: If Found A Number Equal To Update Count By 1.

Step 6: After Reaching The End Of The Linked List Return Count.

Step 7: Call The Function.

Step 8: Prints The Number Of Int Occurrences.

Step 9: Stop.

/\* Link list node \*/

struct Node {

    int data;

    struct Node\* next;

};

// global variable for counting frequency of

// given element k

int frequency = 0;

/\* Given a reference (pointer to pointer) to the head

of a list and an int, push a new node on the front

of the list. \*/

void push(struct Node\*\* head\_ref, int new\_data)

{

    /\* allocate node \*/

    struct Node\* new\_node

        = (struct Node\*)malloc(sizeof(struct Node));

    /\* put in the data \*/

    new\_node->data = new\_data;

    /\* link the old list of the new node \*/

    new\_node->next = (\*head\_ref);

    /\* move the head to point to the new node \*/

    (\*head\_ref) = new\_node;

}

/\* Counts the no. of occurrences of a node

(search\_for) in a linked list (head)\*/

int count(struct Node\* head, int key)

{

    if (head == NULL)

        return frequency;

    if (head->data == key)

        frequency++;

    return count(head->next, key);

}

/\* Driver program to test count function\*/

int main()

{

    /\* Start with the empty list \*/

    struct Node\* head = NULL;

    /\* Use push() to construct below list

     1->2->1->3->1  \*/

    push(&head, 1);

    push(&head, 3);

    push(&head, 1);

    push(&head, 2);

    push(&head, 1);

    /\* Check the count function \*/

    cout << "count of 1 is " << count(head, 1);

    return 0;

}

**3. Write a program to find the middle of a linked list.**

**4. Write a program to reverse a linked list.**

The idea is to use three pointers **curr**, **prev,** and **next**to keep track of nodes to update reverse links.

Follow the steps below to solve the problem:

* Initialize three pointers **prev** as NULL, **curr** as **head**, and **next** as NULL.
* Iterate through the linked list. In a loop, do the following:
  + Before changing the **next** of **curr**, store the **next** node
    - next = curr -> next
  + Now update the **next** pointer of **curr** to the **prev**
    - curr -> next = prev
  + Update **prev** as **curr** and **curr** as **next**
    - prev = curr
    - curr = next

