21CSC201J
DATA STRUCTURES AND ALGORITHMS

UNIT-2
Topic : Singly Linked List

LINKED LIST

DEFINITION

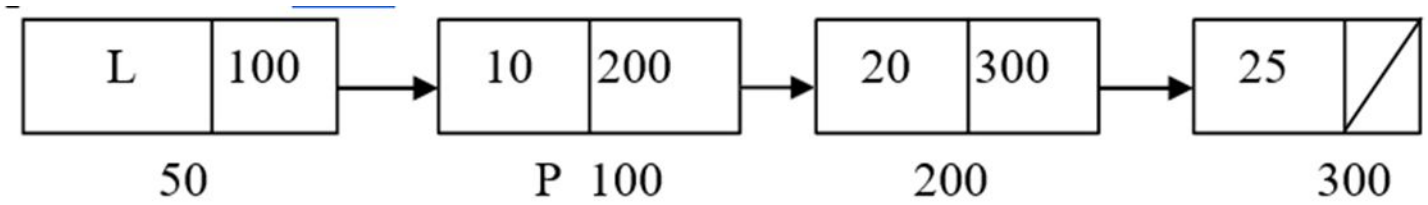
- Linked list is the most commonly used data structure used to store similar type of data in memory.
- The elements of a linked list are not stored in adjacent memory locations as in arrays
- It is the linear collection of data elements, called nodes, where the linear order is implemented by means of pointer.
- Each node contains two fields,

DATA	NEXT
ELEMENT	POINTER

Data: It holds a element

Next: It stores a link to next node in the list

EXAMPLE – SINGLE LINKED LIST



The structure definition for a single linked list is implemented as follows

```
Struct node
{
    int data;
    struct node *next
};
```

STRUCTURE AND VARIABLES FOR THE LIST

```
struct Node
{
    int data;
    struct Node *Next;
};
typedef struct Node *PtrToNode;
typedef PtrToNode LIST;
typedef PtrToNode POSITION;
```

OPERATIONS ON SINGLY LINKED LIST

The following are the possible operations on a Singly Linked List

- Create
- Insertion
- Is Empty
- Is Last
- Find
- Find Previous
- Find Next
- Deletion

ROUTINE TO CREATE A LIST

```
LIST createlist()
{
    LIST L;
    L=(struct Node *)malloc(sizeof(struct Node));
    if(L==NULL)
        printf("fatal error");
    else
    {
        L->data=-1;
        L->Next=NULL;
    }
    return L;
}
```

INSERTION IN A LIST

- Insertion can be done either at the beginning, middle or end of linked list.
  - ***Insert at beginning***
    - Allocate memory for new node
    - Store data
    - Change next of new node to point to head
    - Change head to point to recently created node
  - ***Insert at end***
    - Allocate memory for new node
    - Store data
    - Traverse to last node
    - Change next of last node to recently created node
  - ***Insert at middle (Or in a given Position)***
    - Allocate memory and store data for new node
    - Traverse to node just before the required position of new node
    - Change next pointers to include new node in between

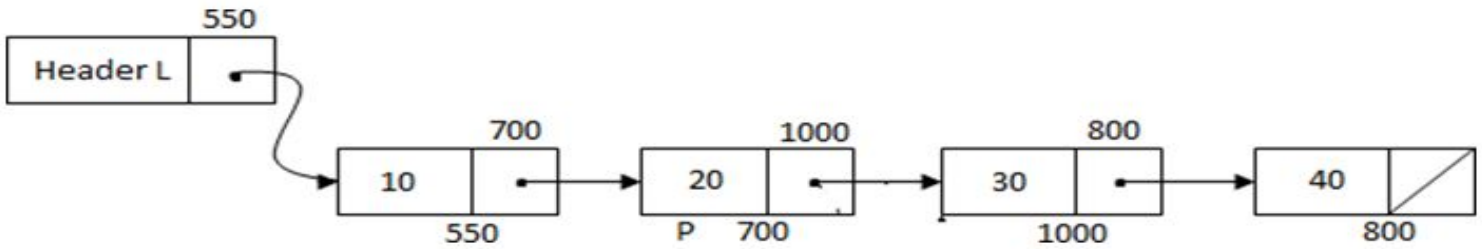
ROUTINE TO INSERT IN A GIVEN POSITION

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

```
void insert (int X ,int L ,position P)
{
    position Newnode;
    Newnode =(struct Node *)malloc(sizeof(struct Node)) ;
    if (Newnode ==NULL)
        printf("Fatal Error");
    else
    {
        Newnode ->Element =X;
        Newnode ->Next =P ->Next;
        P ->Next =Newnode;
    }
}
```

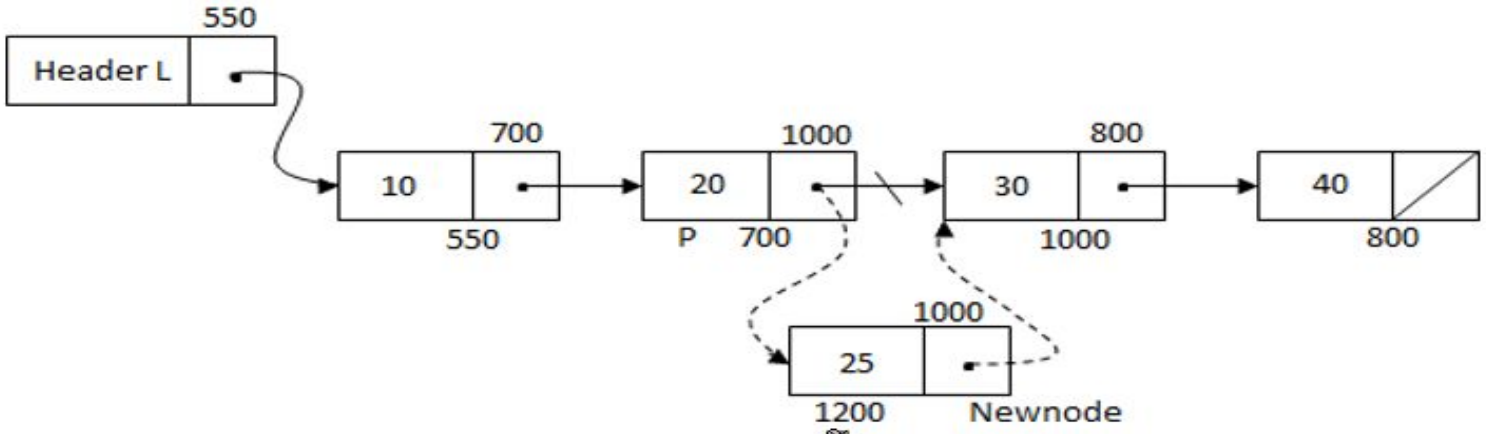
EXAMPLE - INSERTION

In the following Singly linked list perform insert (25, P, L)



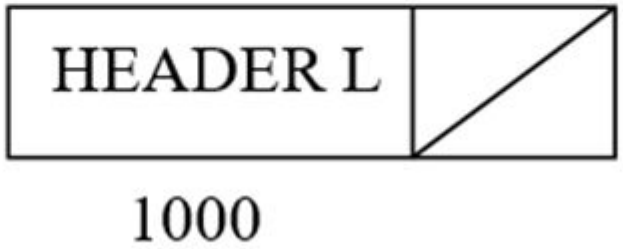
Solution : Linked list after inserting element 25.

INSERT (25, P, L)



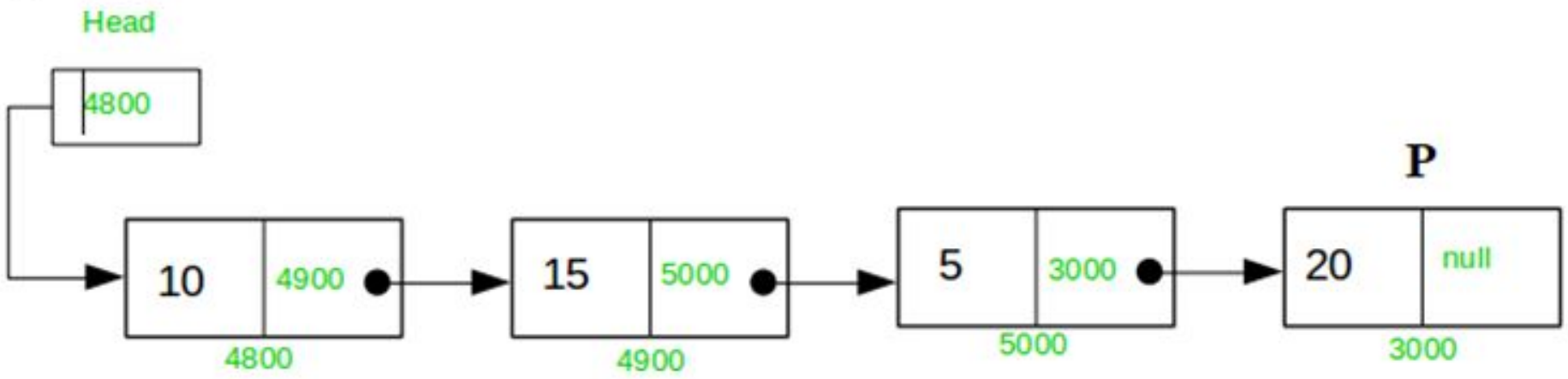
ROUTINE - IS EMPTY

```
int IsEmpty (List L )
{
if (L ->Next==Null)
return(1)

}
```



ROUTINE – LAST POSITION

```
int IsLast (position P, List L)
{
        if (P -> Next==NULL)
        return (1);
}
```

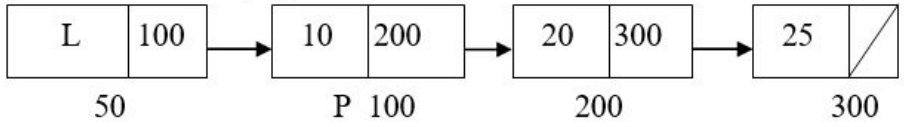


ROUTINE - FIND

Find operation is used to find the position of the given element.

```
position Find (int X, List L)
{
   position P;
   P = L -> Next;
   While (p!=NULL && P -> Element !=X)
         P =P -> Next;
   return P;
}
```

Example: Perform find (10) in this linked list

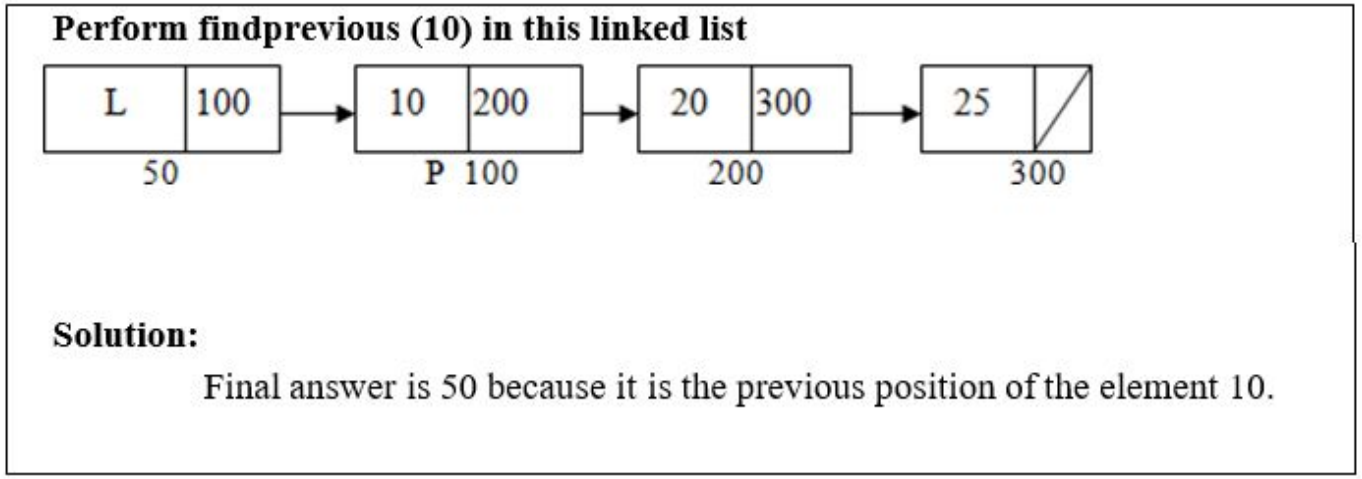


Solution:

Final answer is 100 because it is the position of the element 10.

ROUTINE – FIND PREVIOUS

Find Previous operation is used to find the previous position of the given element.



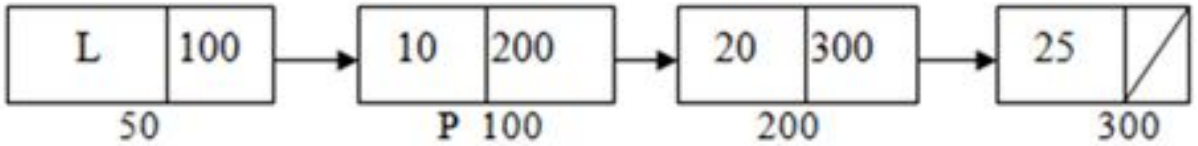

```
Position FindPrevious (int X, List L)
{
    position P;
    P=L;
    While (P -> Next != Null && P -> Next -> Element !=X)
        P=P -> Next;
    return p;
}
```

ROUTINE – FIND NEXT

Find Next operation is used to find the next position of the given element.

```
Position FindNext (int X, List L)
{
    P=L -> Next;
        While (P -> Next != Null && P -> Element !=X)
            P=P -> Next;
    return P -> Next;
}
```

Perform findnext (10) in this linked list



Solution:

Final answer is 200 because it is the next position of the element 10.

ROUTINE - DELETION

You can delete either from beginning, end or from a particular position.

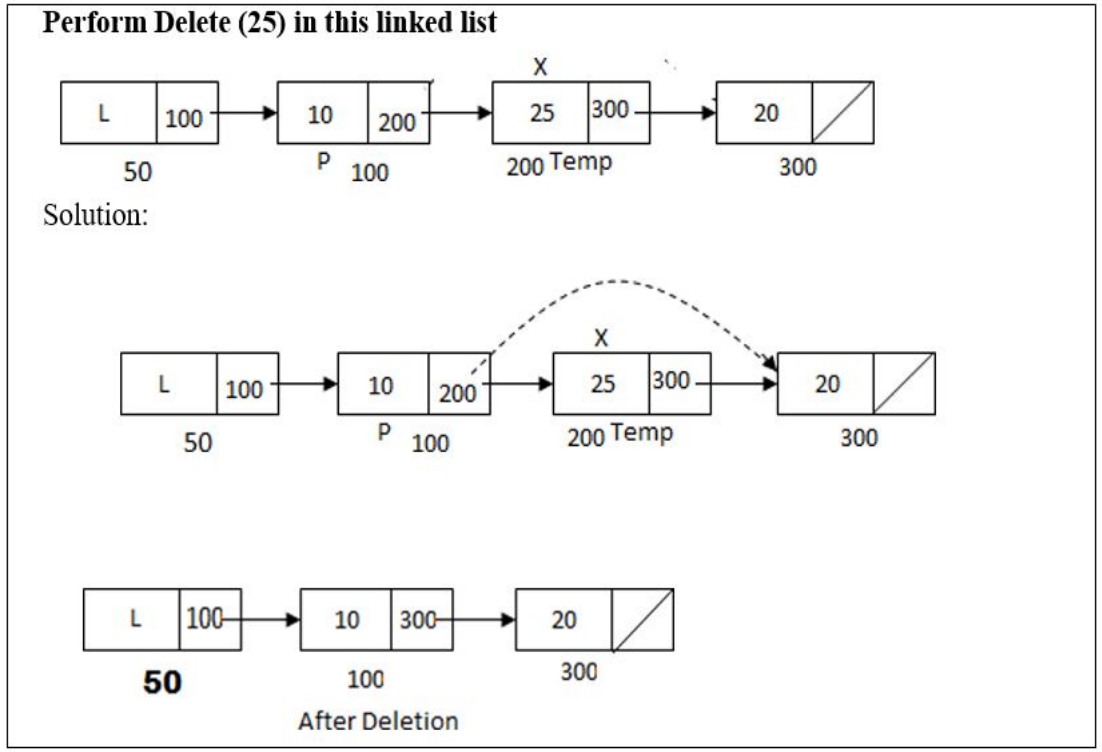
- ***Delete from beginning***
  - Point head to the second node

    head = head->next;

- ***Delete from end***
  - Traverse to second last element
  - Change its next pointer to null
- ***Delete from middle***
  - Traverse to element before the element to be deleted
  - Change next pointers to exclude the node from the chain

ROUTINE TO DELETE AN ELEMENT FR THE LIST

```
Void Delete ( int X, List L)
{
    Position P, Temp;
    P = Findprevious (X,L);
    If(!IsLast(P,L))
    {
        Temp = P□ Next;
        P□Next = Temp□Next;
        Free (Temp);
    }
}
```



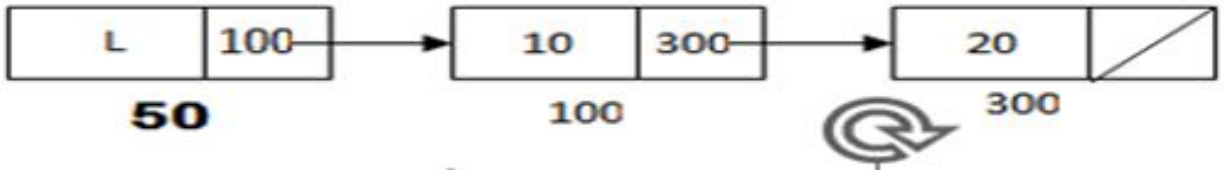
ROUTINE TO DELETE A LIST

- It is used to delete the whole list and make it as empty list.

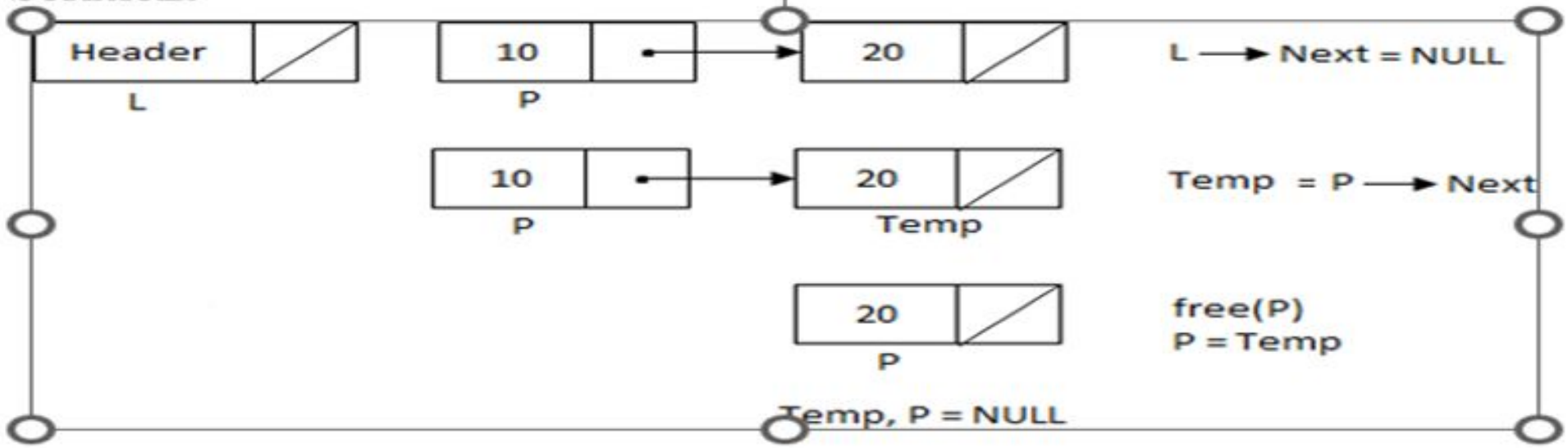
```
Void DeleteList(List L)
{
    Position P,Temp;
    P = L□Next;
    L□Next = NULL;
    While(P!= NULL)
    {
        Temp = P□Next;
        Free(P);
        P = Temp;
    }
}
```

EXAMPLE

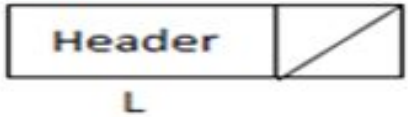
Delete the following linked list



Solution:



Finally there will be header node alone



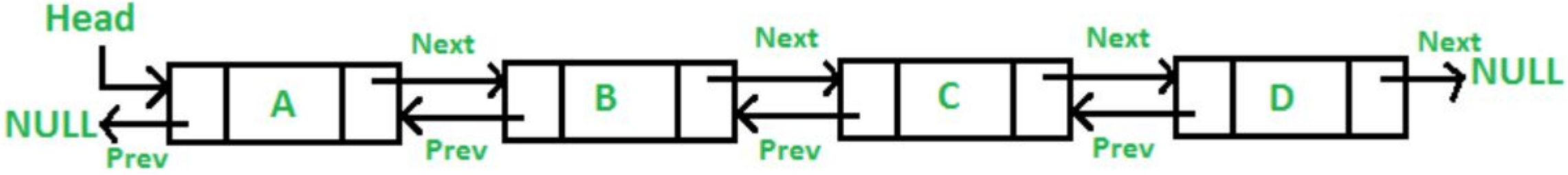
THANK YOU

21CSC201J
DATA STRUCTURES AND ALGORITHMS

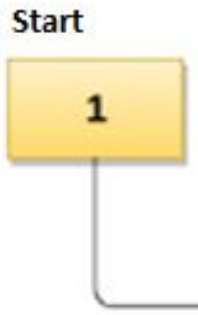
UNIT-2
Topic :Doubly Linked List

DOUBLY LINKED LIST

A doubly linked list (DLL) is a special type of linked list in which each node contains a



Memory Representation



	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Prons and Cons

Advantages of Doubly Linked List over the singly linked list:

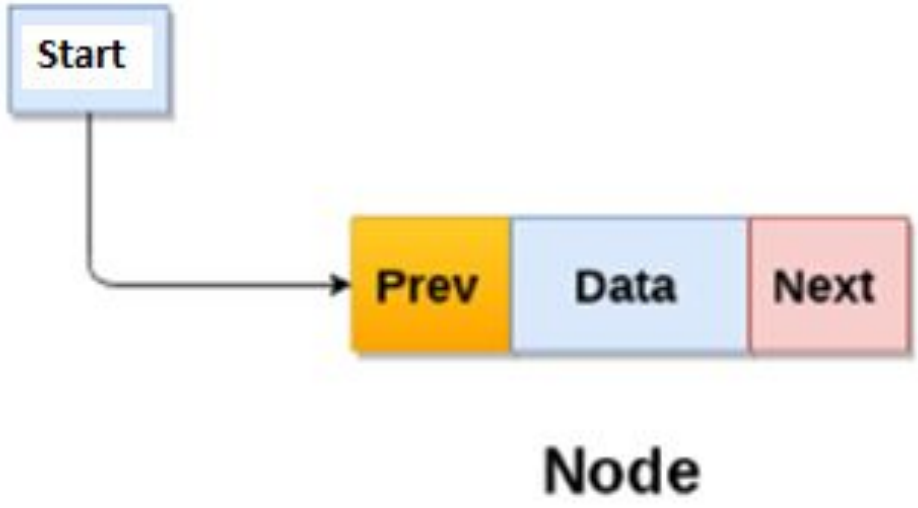
- A DLL can be traversed in both forward and backward directions.
- The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

Disadvantages of Doubly Linked List over the singly linked list:

- Every node of DLL Requires extra space for a previous pointer. It is possible to implement DLL with a single pointer though (See this and this).
- All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with the next pointers. For example in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

Creating a Node in DLL

```
// Linked List Node
struct node {
    int info;
    struct node *prev, *next;
};
struct node* start = NULL;
```

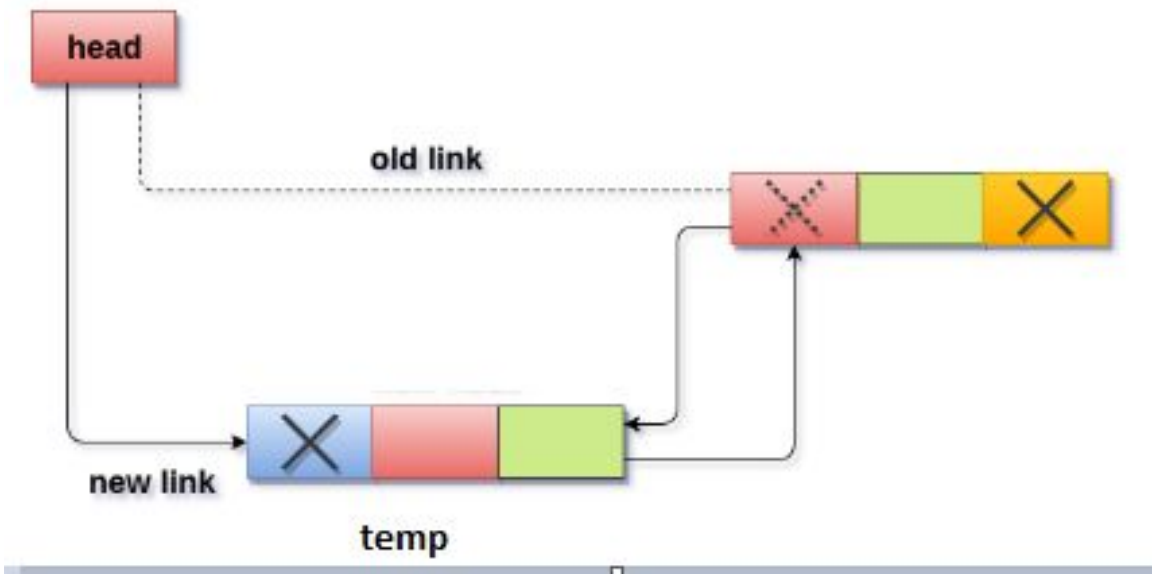


Function to traverse the linked list

```
void traverse()
{
    // List is empty
    if (start == NULL) {
        printf("\nList is empty\n");
        return;
    }
    // Else print the Data

    struct node* temp;
    temp = start;
    while (temp != NULL) {
        printf("Data = %d\n", temp->info);
        temp = temp->next;
    }
}
```

Function to insert at the front of the linked list

```
void insertAtFront()
{
    int data;
    struct node* temp;
temp = (struct node*)malloc(sizeof(struct node));
    printf("\nEnter number to be inserted: ");
    scanf("%d", &data);
    temp->info = data;
    temp->prev = NULL;
// Pointer of temp will be  assigned to start
    temp->next = start;
    start = temp;
}
```

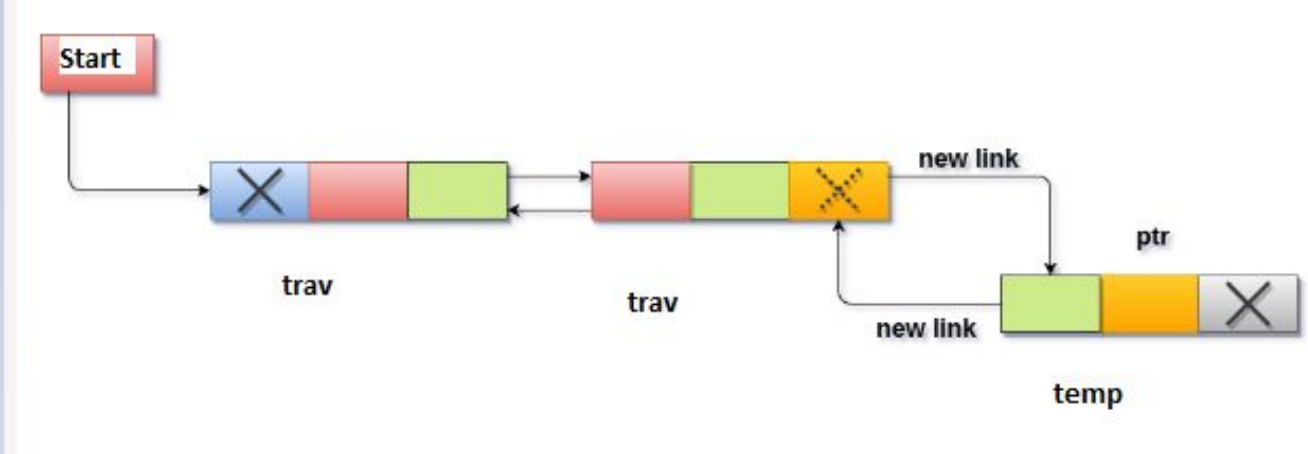


Function to insert at the end of the linked

```
void insertAtEnd()
{
    int data;
    struct node *temp, *trav;
temp = (struct node*)malloc(sizeof(struct node));
    temp->prev = NULL;
    temp->next = NULL;
    printf("\nEnter number to be inserted: ");
    scanf("%d", &data);
    temp->info = data;
    temp->next = NULL;
    trav = start;
```

```
// If start is NULL
  if (start == NULL) {

      start = temp;
  }
```

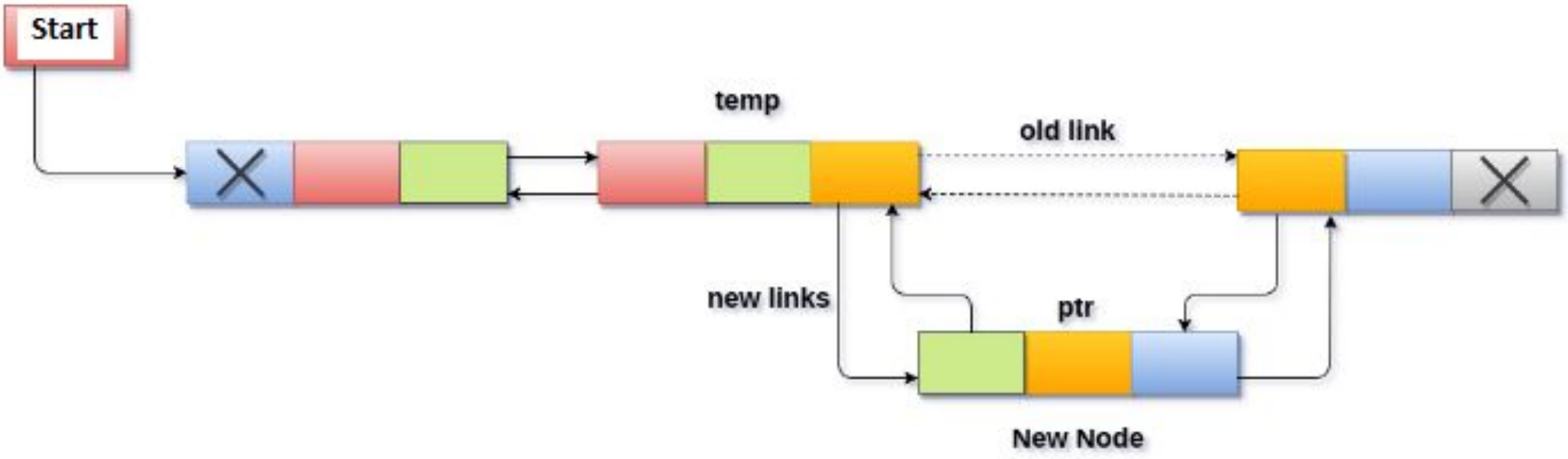
```
// Changes Links
   else {
      while (trav->next != NULL)
            trav = trav->next;
      temp->prev = trav;
      trav->next = temp;
   }
}
```



Function to insert at any specified position in the linked list

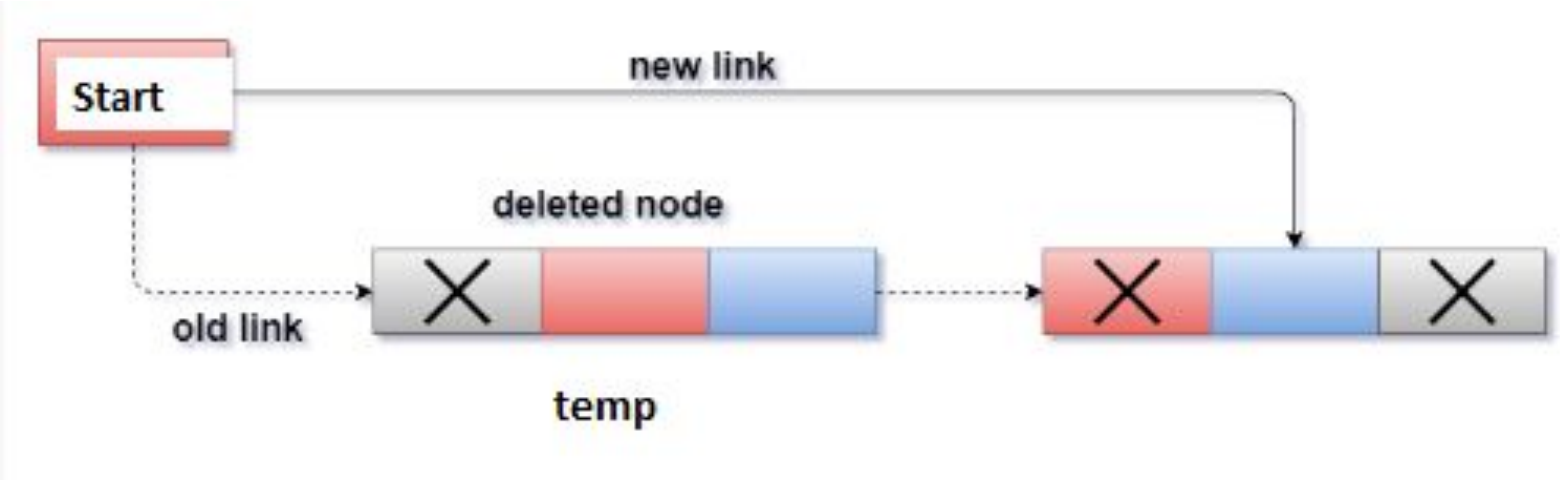
```
void insertAtPosition()
{
   int data, pos, i = 1;
   struct node *temp, *newnode;
   newnode = malloc(sizeof(struct node));
   newnode->next = NULL;
   newnode->prev = NULL;
    // Enter the position and data
   printf("\nEnter position : ");
   scanf("%d", &pos);
       if (start == NULL)
{
      start = newnode;
      newnode->prev = NULL;
      newnode->next = NULL;
   }
else if (pos == 1)
            insertAtFront();
      else {
  printf("\nEnter number to be inserted: ");
 scanf("%d", &data);
 newnode->info = data;
 temp = start;
    while (i < pos - 1) {
       temp = temp->next;
       i++;
                          }
    newnode->next = temp->next;
    newnode->prev = temp;
    temp->next = newnode;
    temp->next->prev = newnode;
 }
}
```

Function to insert at any specified position in the linked list



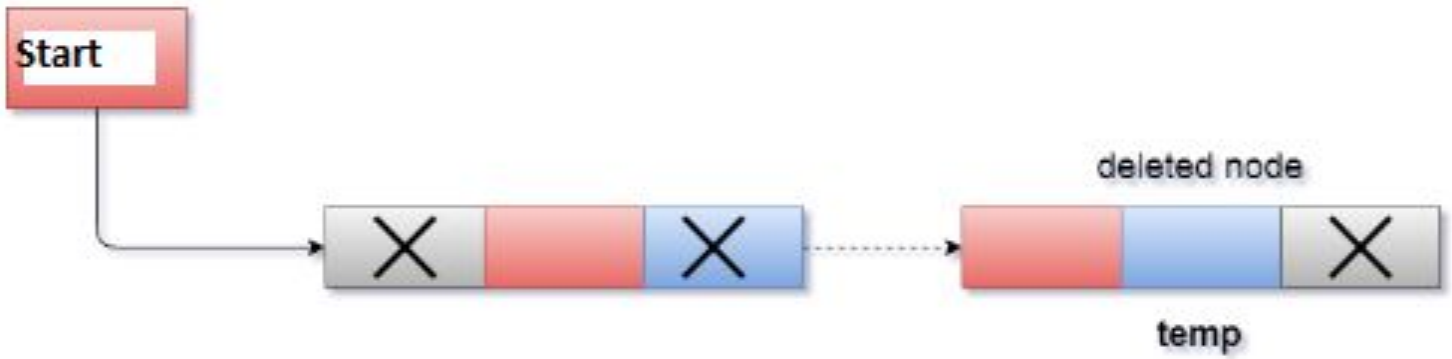
Function to delete from the front of the linked list

```
void deleteFirst()
{
    struct node* temp;
    if (start == NULL)
        printf("\nList is empty\n")
    else {
        temp = start;
        start = start->next;
        if (start != NULL)
            start->prev = NULL;
        free(temp);
    }
}
```



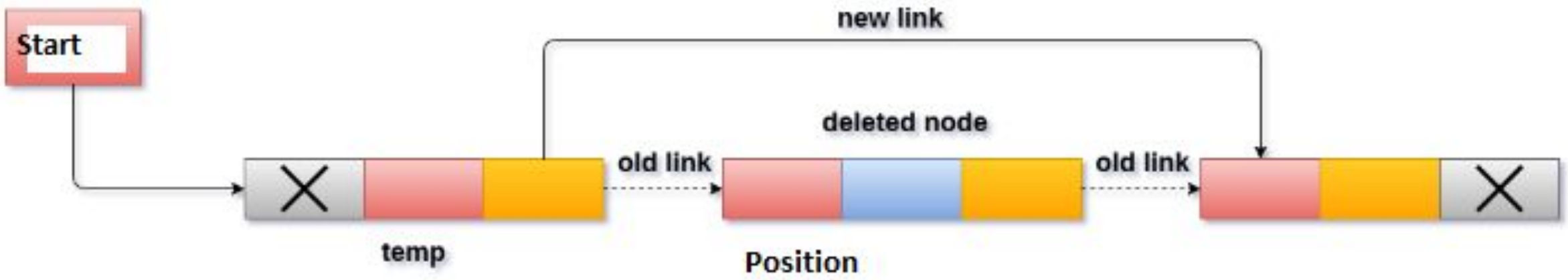
Function to delete from the end of the linked list

```
void deleteEnd()
{
    struct node* temp;
    if (start == NULL)
        printf("\nList is empty\n");
    temp = start;
    while (temp->next != NULL)
        temp = temp->next;
    if (start->next == NULL)
        start = NULL;
    else {
        temp->prev->next = NULL;
        free(temp);
    }
}
```



Function to delete from any specified position from the linked list

```
void deletePosition()
{
    int pos, i = 1;
    struct node *temp, *position;
    temp = start;
      if (start == NULL)
       printf("\nList is empty\n");
       else {
       printf("\nEnter position : ");
       scanf("%d", &pos);
       if (pos == 1) {
           deleteFirst();
      if (start != NULL)
 {
        start->prev = NULL;
           }
           free(position);
           return;
        }
 while (i < pos - 1)
{
           temp = temp->next;
           i++;
        }
 position = temp->next;
if (position->next != NULL)
  position->next->prev = temp;
    temp->next = position->next;

      free(position);
  }
 }
```

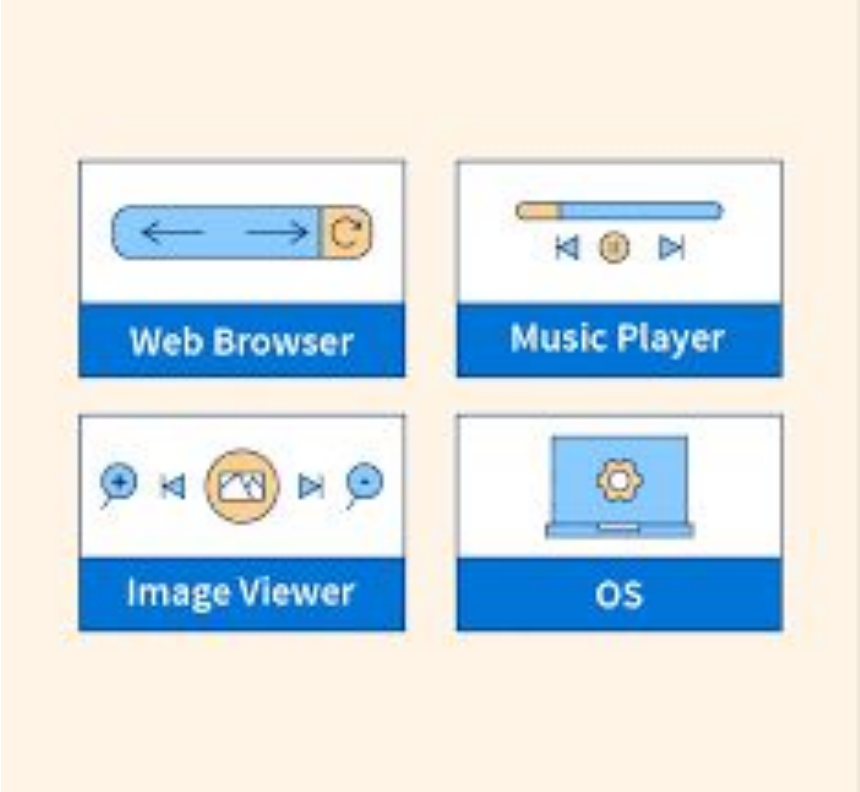

Start
new link
deleted node
old link
old link
temp
Position

Applications of DLL

It is used in the navigation systems where front and back navigation is required.

It is used by the browser to implement backward and forward navigation of visited web pages that is a back and forward button.

It is also used to represent a classic game deck of cards.



Applications of List Data Structure

Applications of List Data Structure

1. Sparse Matrix
2. Polynomial
3. Joseph Problem

1. Sparse Matrix

- a matrix can be defined with a 2-dimensional array
- Any array with 'm' columns and 'n' rows represent a m X n matrix.
- There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

sparse … many elements are zero

dense … few elements are zero

Representation of Sparse Matrix

- Representing a sparse matrix by a 2D array **leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the ca**

Example

- Consider a matrix of size 100 X 100 containing only 10 non-zero elements.
- In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero.
- Totally we allocate 100 X 100 X 2 = 20000 bytes of space to store this integer matrix.
- To access these 10 non-zero elements we have to make scanning for 10000 times.

Representation of Sparse Matrix contd.,

- A sparse matrix can be effectively represented by using TWO representations, those are as follows...
  - Triplet Representation (Array Representation)
  - Linked Representation

Triplet Representation of Sparse Matrix

- In this representation, only non-zero values along with their row and column index values are considered. (Triplet: Row, Column, Value)
- In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.
- **Example:** consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

Triplet Representation of Sparse Matrix

- *a[0].row*: number of rows of the matrix
- *a[0].col*: number of columns of the matrix
- *a[0].value*: number of nonzero entries
- The triples are ordered by row and within rows by columns.

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Triplet Representation of Sparse Matrix (Using Array)

```
int main()
{
   // Assume 4x5 sparse matrix
   int sparseMatrix[4][5] =
   {
      {0 , 0 , 3 , 0 , 4 },
      {0 , 0 , 5 , 7 , 0 },
      {0 , 0 , 0 , 0 , 0 },
      {0 , 2 , 6 , 0 , 0 }
   };

   int size = 0;
   for (int i = 0; i < 4; i++)
      for (int j = 0; j < 5; j++)
         if (sparseMatrix[i][j] != 0)
            size++;

   // number of columns in compactMatrix (size) must
      be
   // equal to number of non - zero elements in
   // sparseMatrix
   int compactMatrix[3][size];
```

```
// Making of new matrix
   int k = 0;
   for (int i = 0; i < 4; i++)
      for (int j = 0; j < 5; j++)
         if (sparseMatrix[i][j] != 0)
         {
            compactMatrix[0][k] = i;
            compactMatrix[1][k] = j;
            compactMatrix[2][k] = sparseMatrix[i][j];
            k++;
         }
    for (int i=0; i<3; i++)
   {
      for (int j=0; j<size; j++)
         printf("%d ", compactMatrix[i][j]);

      printf("\n");
   }
   return 0;
}
```

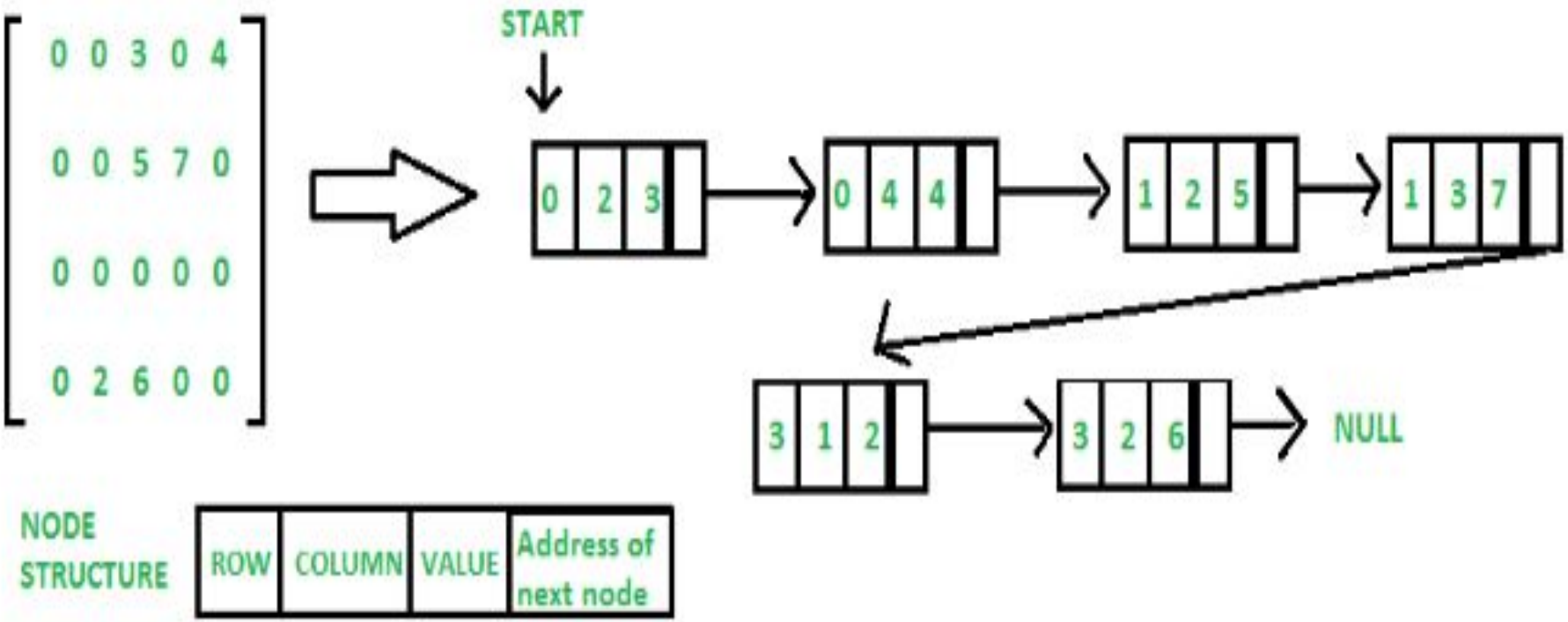
Output

```
0 0 1 1 3 3
2 4 2 3 1 2
3 4 5 7 2 6
```

Linked List Representation of Sparse Matrix

- In linked list, each node has four fields. These four fields are defined as:
  - **Row:** Index of row, where non-zero element is located
  - **Column:** Index of column, where non-zero element is located
  - **Value:** Value of the non zero element located at index – (row,column)
  - **Next node:** Address of the next node

Linked List Representation of Sparse Matrix



Sparse Matrix Operations

- Transpose of a sparse matrix.
- What is the transpose of a matrix?

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

transpose →

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

Sparse Matrix Operations -Transpose

(1) for each row i

take element <i, j, value> and store it

in element <j, i, value> of the transpose.

difficulty: where to put <j, i, value>?

(0, 0, 15) ====> (0, 0, 15)

(0, 3, 22) ====> (3, 0, 22)

(0, 5, -15) ====> (5, 0, -15)

(1, 1, 11) ====> (1, 1, 11)

Move elements down very often.

(2) For all elements in column j,

place element <i, j, value> in element <j, i, value>

Sparse Matrix Operations -Transpose

- Using column indices to determine placement of elements

```
for all elements in column j
    place element <i, j, value> in element <j, i, value>
```

- Algorithms

```
n = a[0].value;
...
if (n > 0) {
    currentb = 1;
    for (i = 0; i < a[0].col; i++)   /* for all columns */
        for (j = 1; j <= n; j++)     /* for all nonzero elements */
            if (a[j].col == i) {     /* in current column */
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
}
```

Sparse Matrix Operations -Transpose

Analysis of Transpose Algorithm

- The total time for the nested **for** loops is **`columns*elements`**
- Asymptotic time complexity is **`O(columns*elements)`**
- When # of elements is order of **`columns*rows`**, **`O(columns*elements)`** becomes **`O(columns`**^{2}**`*rows)`**
- A simple form algorithm has time complexity **`O(columns*rows)`**

```
for (j = 0; j < columns; j++)
    for (i = 0; i < rows; i++)
        b[j][i] = a[i][j];
```

Sparse Matrix Operations - Addition

To **Add** the matrices, simply traverse through both matrices element by element and insert the smaller element (one with smaller row and col value) into the resultant matrix. If we come across an element with the same row and column value, simply add their values and insert the added data into the resultant matrix.

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 9 & 0 & 4 & 0 \\ 0 & 8 & 0 & 0 & 0 & 7 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

$$+$$

$$B = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 18 & 0 & 0 & 0 \\ 0 & 3 & 0 & 24 & 0 & 17 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \end{bmatrix} \end{array}$$

A

0	1	2	3	4	5	6
5	1	2	2	3	3	4
6	5	3	5	2	6	1
6	3	9	4	8	7	2

B

0	1	2	3	4	5	6
5	1	2	3	3	3	5
6	1	3	2	4	6	4
6	2	18	3	24	17	6

Sparse Matrix Operations - Addition

A

0	1	2	3	4	5	6
5	1	2	2	3	3	4
6	5	3	5	2	6	1
6	3	9	4	8	7	2

+

B

0	1	2	3	4	5	6
5	1	2	3	3	3	5
6	1	3	2	4	6	4
6	2	18	3	24	17	6

C

0	1	2	3	4	5	6	7	8	9
5	1	1	2	2	3	3	3	4	5
6	1	5	3	5	2	4	6	1	4
9	2	3	27	4	11	24	24	2	6

$$C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 2 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 27 & 0 & 4 & 0 \\ 0 & 11 & 0 & 24 & 0 & 24 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \end{bmatrix} \end{array}$$

Note:

The max.size of the resultant structure will be = no. of non-zero elements in (Matrix A + Matrix B) , if we don't have any common entries in both the tables A & B.

Sparse Matrix Operations - Multiplication

Matrix Multiplication

- Definition

  Given A and B where A is $m \times n$ and B is $n \times p$, the product matrix D has dimension $m \times p$. Its $<i, j>$ element is:

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

  for $0 \leq i < m$ and $0 \leq j < p$.

- Example

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2 & 4 \\ 3 & 3 & 6 \end{bmatrix}$$

Sparse Matrix Operations - Multiplication

Classic multiplication algorithm

- Algorithm:

```
for (i = 0; i < rows_a; i++)
    for (j = 0; j < cols_b; j++) {
        sum = 0;
        for (k = 0; k < cols_a; k++)
            sum += (a[i][k] * b[k][j]);
            d[i][j] = sum;
    }
```

- The time complexity is

```
O(rows_a*cols_a*cols_b)
```

Sparse Matrix Operations - Multiplication

Multiply two sparse matrices

- $D = A \times B$
- Matrices are represented as ordered lists
- Pick a row of A and find all elements in column j of B for j = 0, 1, 2, …, cols_b – 1
- Have to scan all of B to find all the elements in column j
- Compute the transpose of B first
  - This put all column elements of B in consecutive order
- Then, just do a merge operation

Sparse Matrix Operations - Multiplication

Example

	col 0	col 1	col 2	col 3	col 4	col 5
row 0	15	0	0	22	0	-15
row 1	0	11	3	0	0	0
row 2	0	0	0	-6	0	0
row 3	0	0	0	0	0	0
row 4	91	0	0	0	0	0
row 5	0	0	28	0	0	0

Matrix B

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

Transpose of B

Sparse Matrix Operations - Multiplication

mmult [1]

```
int rows_a = a[0].row, cols_a = a[0].col, totala = a[0].value,...
int row_begin = 1, row = a[1].row, sum = 0;
fast_transpose(b, new_b);
a[totala+1].row = rows_a; new_b[totalb+1].row = cols_b;
new_b[totalb+1].col = 0;
for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for (; new_b[j].row == column; j++);
            column = new_b[j].row;
        }
```

Sparse Matrix Operations - Multiplication

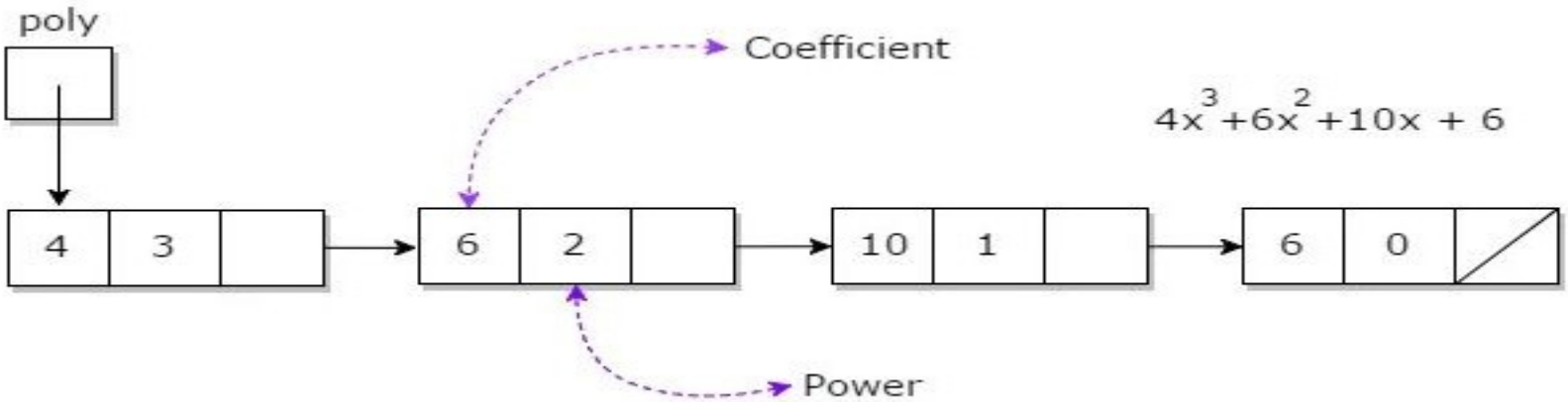
mmult [2]

```
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[j],col)) {
            case -1: i++;  break;
            // a[i].col < new_b[j],col,  go to next term in a
            case 0: sum += (a[i++].value * new_b[j++].value);  break;
            /* add terms, advance i and j*/
            case 1: j++;     /* advance to next term in b */
            }
        }
    for (; a[i].row; == row; i++) ;
    row_begin = i; row = a[i].row;
}
d[0].row = rows_a; d[0].col = cols_b; d[0].value = totald;
```

2. Polynomial Arithmetic using Linked List

- A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots + jx + k$), where a, b, c…., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of Polynomial.

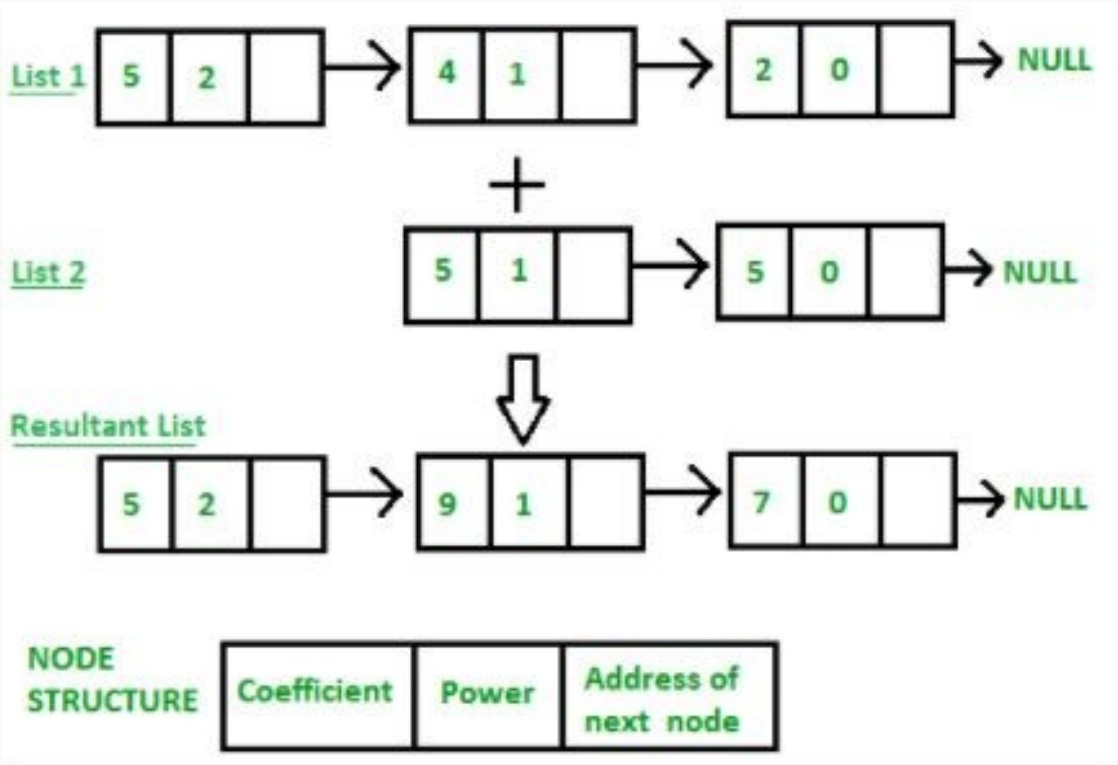
- A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tupple which holds two pieces of information:
  - The exponent part



2. Polynomial Arithmetic using Linked List -Addition

- Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

  - **1st number** = $5x^2 + 4x^1 + 2x^0$
  - **2nd number** = $-5x^1 + 5x^0$



3. Josephus Circle using circular linked list

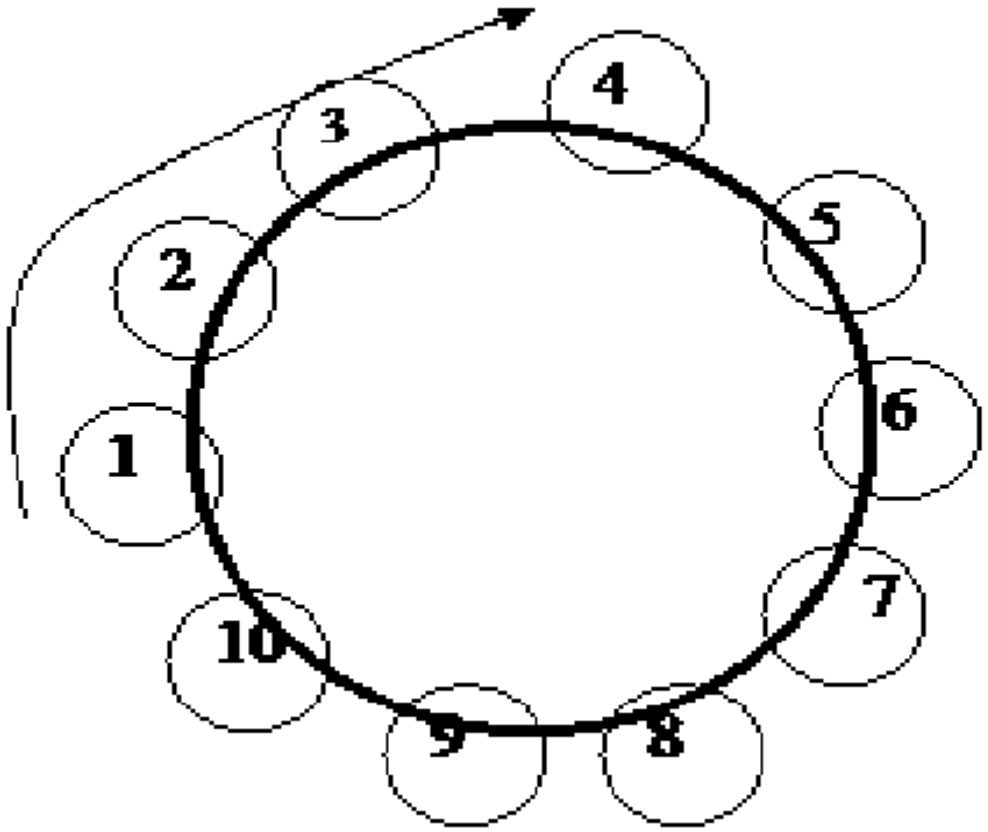
- There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last person remains, who is given freedom. Given the total number of persons n and a number m which indicates that m-1 persons are skipped and m-th person is killed in circle. The task is to choose the place in the initial

Examples :

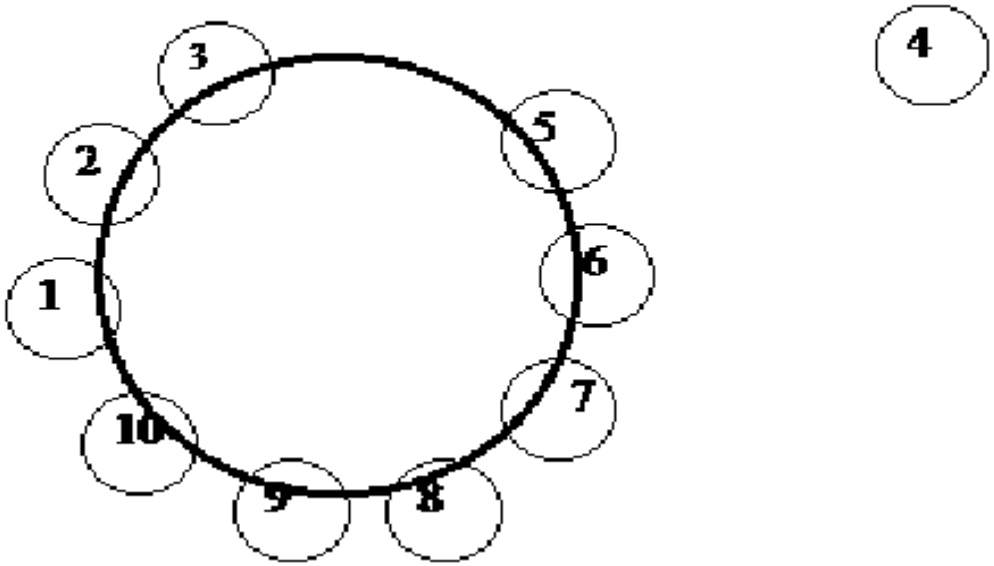
```
Input : Length of circle : n = 4
        Count to choose next : m = 2
Output : 1

Input : n = 5
        m = 3
Output : 4
```

Example

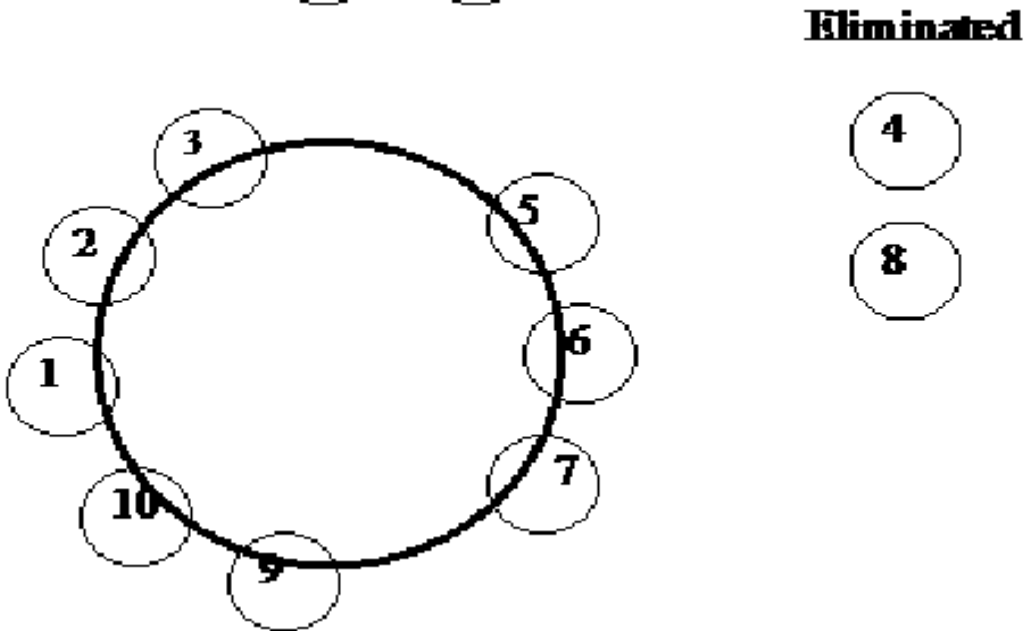
N=10, M=3



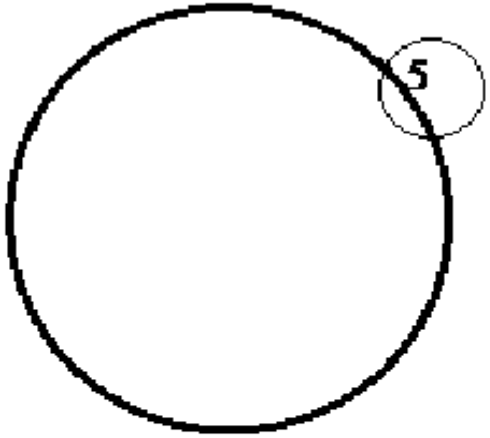
N=10, M=3



N=10, M=3



N=10, M=3



Eliminated

Josephus Problem - Snippet

```
int josephus(int m, node *head)
{
    node *f;
    int c=1;
    while(head->next!=head)
    {
        c=1;
        while(c!=m)
        {
            f=head;
            head=head->next;
            c++;
        }
        f->next=head->next;
//sequence in which nodes getting deleted

        printf("%d->",head->data);
 head=f->next;
}
printf("\n");
printf("Winner is:%d\n",head->data);
return;

}
```

Thank You…

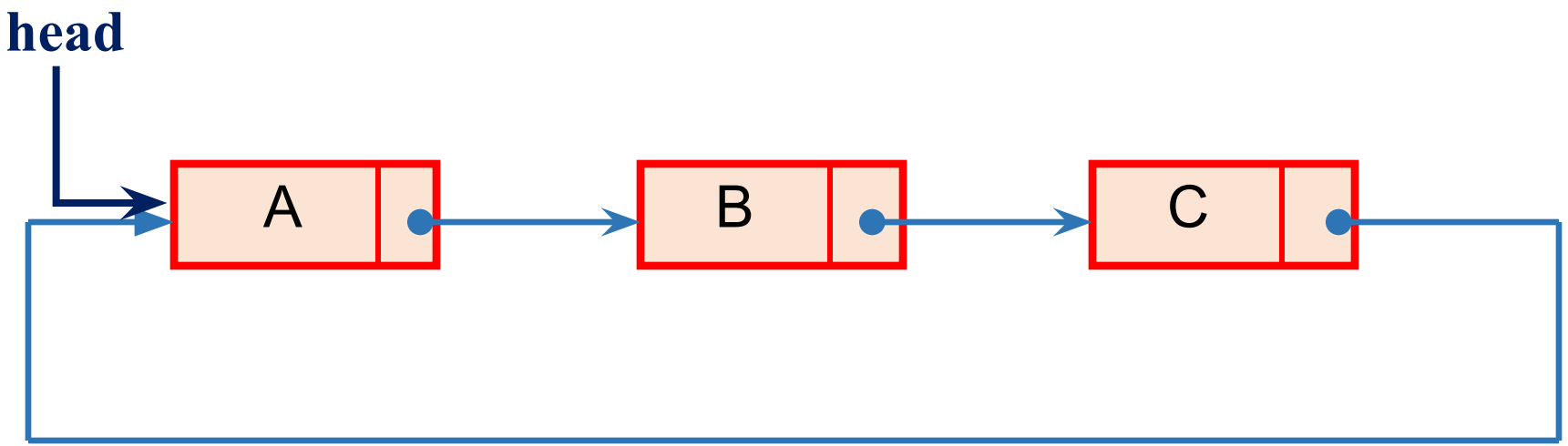
21CSC201J
DATA STRUCTURES AND ALGORITHMS

UNIT-2
Topic : Circular linked list

Circular Linked List

Circular linked list

- The pointer from the last element in the list points back to the first element.

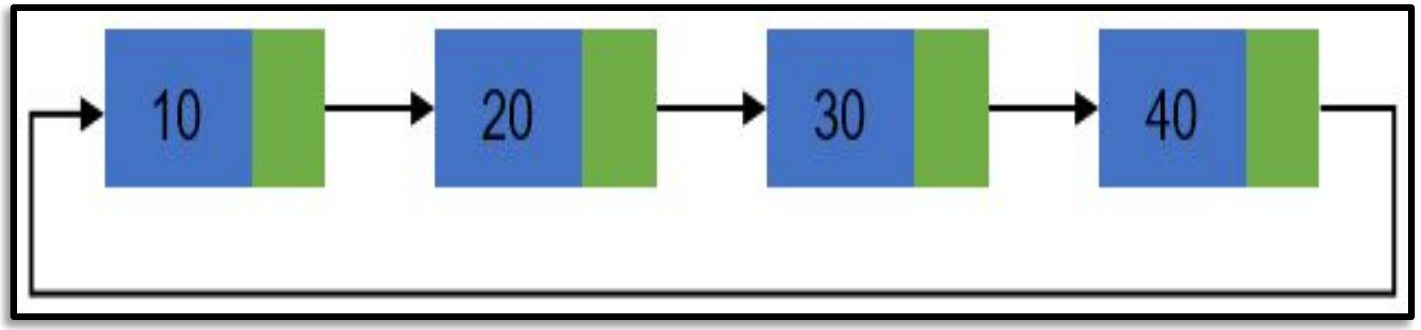


Circular Linked List

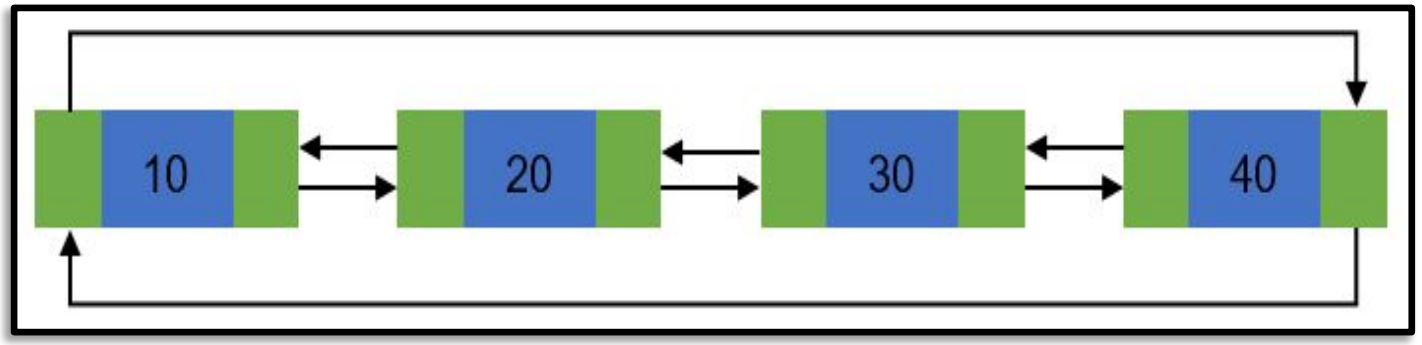
- A circular linked list is basically a linear linked list that may be single- or double-linked.
- The only difference is that there is no any NULL value terminating the list.
- In fact in the list every node points to the next node and last node points to the first node, thus forming a circle. Since it forms a circle with no end to stop it is called as **circular linked list**.
- In circular linked list there can be no starting or ending node, whole node can be traversed from any node.
- In order to traverse the circular linked list, only once we need to traverse entire list until the starting node is not traversed again.
- A circular linked list can be implemented using both singly linked list and doubly linked list.

Circular Linked List

Basic structure of singly circular linked list:



Doubly circular linked list:



Circular Linked List

Advantages of a Circular linked list

- Entire list can be traversed from any node.
- Circular lists are the required data structure when we want a list to be accessed in a circle or loop.
- Despite of being singly circular linked list we can easily traverse to its previous node, which is not possible in singly linked list.

Disadvantages of Circular linked list

- Circular list are complex as compared to singly linked lists.
- Reversing of circular list is a complex as compared to singly or doubly lists.
- If not traversed carefully, then we could end up in an infinite loop.
- Like singly and doubly lists circular linked lists also doesn't supports direct accessing of elements.

Operations on Circular Linked List

- Creation of list
- Traversal of list
- Insertion of node
    - At the beginning of list
    - At any position in the list
- Deletion of node
    - Deletion of first node
    - Deletion of node from middle of the list
    - Deletion of last node
- Counting total number of nodes
- Reversing of list

Creation and Traversal of a Circular List

```
#include <stdio.h>
#include <stdlib.h>

/* Basic structure of Node */

struct node {
    int data;
    struct node * next;
}*head;

int main()
{
    int n, data;
    head = NULL;

    printf("Enter the total number of nodes in list: ");
    scanf("%d", &n);
    createList(n);                  // function to create circular linked list
    displayList();                  // function to display the list

    return 0;
}
```

Creation of a Circular List

```
void createList(int n)
{
    int i, data;
    struct node *prevNode, *newNode;
    if(n >= 1){                          /* Creates and links the head node */
        head = (struct node *)malloc(sizeof(struct node));

        printf("Enter data of 1 node: ");
        scanf("%d", &data);

        head->data = data;
        head->next = NULL;
        prevNode = head;

        for(i=2; i<=n; i++){        /* Creates and links rest of the n-1 nodes */
            newNode = (struct node *)malloc(sizeof(struct node));

            printf("Enter data of %d node: ", i);
            scanf("%d", &data);

            newNode->data = data;
            newNode->next = NULL;
            prevNode->next = newNode; //Links the previous node with newly created node
            prevNode = newNode;       //Moves the previous node ahead
        }
            prevNode->next = head; //Links the last node with first node
           printf("\nCIRCULAR LINKED LIST CREATED SUCCESSFULLY\n");
    }
}
```

Traversal of a Circular List

```
void displayList()
{
    struct node *current;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        current = head;
        printf("DATA IN THE LIST:\n");

        do {
            printf("Data %d = %d\n", n, current->data);

            current = current->next;
            n++;
        }while(current != head);
    }
}
```

Few Exercises to Try Out

- **For circular linked list write a function to:**

- Insert a node at any position of the list and delete from the beginning of the list.
  - `insert_position(data,position);`
  - `delete_front();`

- Reverse the given circular linked link.

Thank You

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

21CSC201J
DATA STRUCTURES AND ALGORITHMS

UNIT-2
Topic : Implementation of List ADT-Array, Cursor based & linked

Implementing an ADT

- To implement an ADT, you need to choose:
  - A data representation
    - must be able to represent all necessary values of the ADT
    - should be private
  - An algorithm for each of the necessary operation:
    - must be consistent with the chosen representation
    - all auxiliary (helper) operations that are not in the contract should be private
- Remember: Once other people are using it
  - It's easy to add functionality

List - Implementation

- Lists can be implemented using:
  - Arrays
  - Linked List
  - Cursor [Linked List using Arrays]

Arrays

- Array is a static data structure that represents a collection of fixed number of homogeneous data items <u>or</u>
- A fixed-size indexed sequence of elements, all of the same type.
- The individual elements are typically stored in consecutive memory locations.
- The length of the array is determined when the array is created, and cannot be changed.

Arrays (Contd..)

- Any component of the array can be inspected or updated by using its index.
  - This is an efficient operation
  - O(1) = constant time
- The array indices may be integers (C, Java) or other discrete data types (Pascal, Ada).
- The lower bound may be zero (C, Java), one (Fortran), or chosen by the programmer (Pascal, Ada)

Different types of Arrays

- One-dimensional array: only one index is used
- Multi-dimensional array: array involving more than one index
- Static array: the compiler determines how memory will be allocated for the array
- Dynamic array: memory allocation takes place during execution

Insertion into Array

What happens if you want to insert an item at a specified position in an existing array?

1. Write over the current contents at the given index (which might not be appropriate), or
2. The item originally at the given index must be moved up one position, and all the items after that index shuffled up.

Removal from Arrays

What happens if you want to remove an item from a specified position in an existing array?

1. Leave gaps in the array, i.e. indices that contain no elements, which in practice, means that the array element has to be given a special value to indicate that it is “empty”, or
2. All the items after the (removal items) index must be shuffled down

- **Good** things:
  - Fast, random access of elements
  - Very memory efficient, very little memory is required other than that needed to store the contents (but see bellow)
- **Bad** things:
  - Slow deletion and insertion of elements
  - Size must be known when the array is created and is fixed (static)

Implementation of List ADT using Linked List

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct Node
{   int data;
    struct Node *Next;
};
typedef struct Node *PtrToNode;
typedef PtrToNode LIST;
typedef PtrToNode POSITION;
int IsEmpty(LIST L)
{
 return L->Next==NULL;
}
```

```
LIST createlist()
{
    LIST L;
    L=(struct Node *)malloc(sizeof(struct Node));
    if(L==NULL)
   printf("fatal error");
    else
    {   L->data=-1;
     L->Next=NULL;
    }
    return L;
}
```

Title

```
void Insert(int x,POSITION P)
{
    PtrToNode Temp;
    Temp=(struct Node*)malloc(sizeof(struct Node)
    if(Temp==NULL)
   printf("fatal error");
    else
    {
   Temp->data=x;
   Temp->Next=P->Next;
   P->Next=Temp;
    }
```

```
POSITION FindPrevious(int x,LIST L)
{
    POSITION P;
    P=L;
    while(P->Next !=NULL && P->Next->data!=x)
    P=P->Next;
  return P;
}
int IsLast(POSITION P)
{
  return P->Next==NULL;
}
```

Title

```
void Delete(int x,LIST L)
{
    POSITION P, Tempcell;
    P=FindPrevious(x,L);
    if(!IsLast(P))
    {
   Tempcell=P->Next;
   P->Next=Tempcell->Next;
   free(Tempcell);
    }
}
```

```
void MakeEmpty(LIST L)
{
    if(L==NULL)
    printf("list is not created");
    else
    {
   while(!IsEmpty(L))
   Delete(L->Next->data,L);
    }
}
```

Title

```
POSITION Find(int x,LIST L)
{
    POSITION Temp;
    Temp=L;
    while(Temp!=NULL)
    {
   if(Temp->data==x)
   return Temp;
   Temp=Temp->Next;
    }
    return Temp;
}
```

```
void Display(LIST L)
{
    L=L->Next;
    while(L!=NULL)
    {
   printf("\n%d",L->data);
   L=L->Next;
    }
}
LIST Deletelist(LIST L)
{
MakeEmpty(L);
free(L);
L=NULL;
return L;
}
```

List Implemented Using Array

The List ADT

- The List is an
  - Ordered sequence of data items called elements
  - $A_1, A_2, A_3, \ldots, A_N$ is a list of size N
  - size of an empty list is 0
  - A_{i+1} succeeds A_i
  - A_{i-1} preceeds A_i
  - Position of A_i is i
  - First element is A_1 called “head”
  - Last element is A_N called “tail”

Operations on List

- MakeEmpty
- PrintList
- Find
- FindKth
- Insert
- Delete
- Next
- Previous

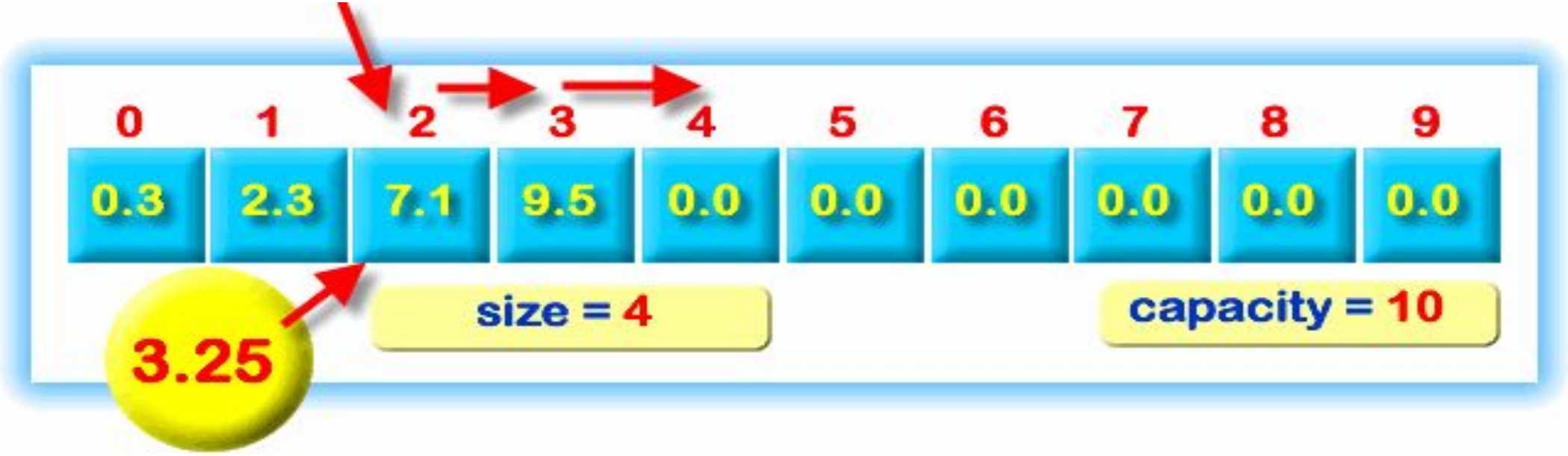
List – An Example

- The elements of a list are 34, 12, 52, 16, 12
  - Find (52) -> 3
  - Insert (20, 4) -> 34, 12, 52, 20, 16, 12
  - Delete (52) -> 34, 12, 20, 16, 12
  - FindKth (3) -> 20

Operations on List

- We'll consider only few operations and not all operations on Lists
- Let us consider Insert
- There are two possibilities:
  - Ordered List
  - Unordered List

Insertion into an ordered list

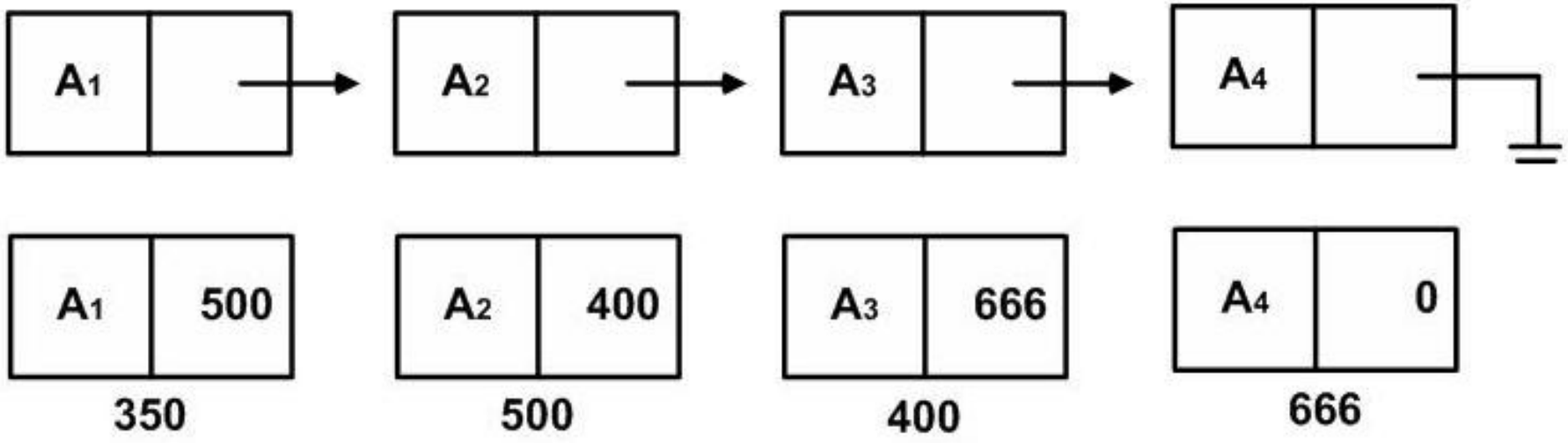


Disadvantages of using Arrays

- Need to define a size for array
  - High overestimate (waste of space)
- insertion and deletion is very slow
  - need to move elements of the list
- redundant memory space
  - it is difficult to estimate the size of array

Linked List

- Series of nodes
  - not adjacent in memory
  - contain the element and a pointer to a node containing its succesor
- Avoids the linear cost of insertion and deletion!



Advantages of using Linked Lists

- Need to know where the first node is
  - the rest of the nodes can be accessed
- No need to move the elements in the list for insertion and deletion operations
- No memory waste

Cursor Implementation

Problems with linked list implementation:

- Same language do not support pointers!
  - Then how can you use linked lists ?
- **`new`** and **`free`** operations are slow
  - Actually not constant time
- SOLUTION: Implement linked list on an array - called CURSOR

Cursor Implementation using ADT

- It is nothing but the linked list representation using array.

Operations:

1. Initialization of array
2. Insertion: Insert new element at position pointed by header (ZERO) and assign new position which is null to header.
3. Deletion: Delete an element and assign that position to header (ZERO)
4. Traversal: Display the entire array

Title

```
#include <iostream.h>
#include <conio.h>

#define SIZE 11

struct node
{
        char element;
        int nextpos;
};
typedef struct node  cursor;

cursor cursorspace[SIZE];
```

```
int main()
{
        int choice,pos;  char ch;
        intialize();
        do
        {
                cout<<"\nMenu";
                cout<<"\n1.> Insert ";
                cout<<"\n2.> Delete ";
                cout<<"\n3.> Display";
                cout<<"\n4.> Exit";
                cout<<"\nEnter Your choice: ";
                cin>>choice;

                switch(choice)
                {
                        case 1:
                                cout<<"\nEnter Character to enter: ";
                                cin>>ch;
                                insert(ch);
                                break;
                        case 2:
                                cout<<"\nEnter Character to Delete: ";
                                cin>>ch;
                                del(ch);
                                break;
                        case 3:
                                display();
                                break;
                        case 4:
                                break;
                }
        }while(choice!=4);
        getch();
        return 0;
}
```

Title

```
void intialize()
{
        for(int i=0;i<SIZE-1;i++)
        {       cursorspace[i].element = 'x';
                cursorspace[i].nextpos = i+1;
        }
        cursorspace[SIZE-1].element = 'x';
        cursorspace[SIZE-1].nextpos = 0;
}
void display()
{
        cout<<"\nCursor Implementation ";
        cout<<"\n-----------------------------------";
        cout<<"\n"<<setw(6)<<"Slot"<<setw(12)<<"Element"<<setw(16)
                                        <<"Next Position";
        for(int i=0;i<SIZE;i++)
        {       cout<<endl<<setw(4)<<i<<"       ";
                cout<<cursorspace[i].element;
                cout<<setw(15)<<cursorspace[i].nextpos;
                cout<<endl;
        }
}
```

Cursor Implementation

Slot	Element	Next Position
0	x	1
1	x	2
2	x	3
3	x	4
4	x	5
5	x	6
6	x	7
7	x	8
8	x	9
9	x	10
10	x	0

Title

```
void insert(char x)
{
        int tmp;

        tmp = cursoralloc();

        if(tmp==0)
                cout<<"\nError-Outof space.";
        else
                cursorspace[tmp].element = x;

}
int cursoralloc()
{
        int p;
        p = cursorspace[0].nextpos;
        cursorspace[0].nextpos = cursorspace[p].nextpos;
        return p;
}
```

Cursor Implementation

Slot	Element	Next Position
0	x	2
1	A	2
2	x	3
3	x	4
4	x	5
5	x	6
6	x	7
7	x	8
8	x	9
9	x	10
10	x	0

Title

Cursor Implementation

Slot	Element	Next
Position		
0	x	2
1	A	2
2	x	3
3	x	4
4	x	5
5	x	6
6	x	7
7	x	8
8	x	9
9	x	10
10	x	0

→

Cursor Implementation

Slot	Element	Next
Position		
0	x	3
1	A	2
2	B	3
3	x	4
4	x	5
5	x	6
6	x	7
7	x	8
8	x	9
9	x	10
10	x	0

→

Cursor Implementation

Slot	Element	Next
Position		
0	x	6
1	A	2
2	B	3
3	C	4
4	D	5
5	E	6
6	x	7
7	x	8
8	x	9
9	x	10
10	x	0

Title

```
void del(char x)
{
        int p,tmp;

        p = findprevious(x);

        if(p==0)
        {
                tmp=1;
                cursorfree(tmp);
        }
        else
        {
                tmp = cursorspace[p].nextpos;
                cursorspace[p].nextpos = cursorspace[tmp].nextpos;
                cursorfree(tmp);
        }
}
```

```
int findprevious(char x)
{
        int p=0;
        if(cursorspace[1].element==x)
        {
                return 0;
        }

        for(int i=0;x!=cursorspace[i].element;i++)
        {
                p = cursorspace[i].nextpos;
        }//gives index of element to be deleted

        for(i=0;cursorspace[i].nextpos!=p;i++);

        p=i;//gives index of previous element
        return p;
}
```

Title

```
void cursorfree(int p)
{
        for(int i=cursorspace[p].nextpos;i<SIZE-1;i++)
        {
                if(cursorspace[i].nextpos==cursorspace[0].nextpos)
                {
                        cursorspace[i].nextpos=p;
                        break;
                }
        }
        cursorspace[p].nextpos = cursorspace[0].nextpos;
        cursorspace[p].element = 'x';
        cursorspace[0].nextpos = p;

}
```

Title

Cursor Implementation

Slot	Element	Next Position
0	x	6
1	A	2
2	B	3
3	C	4
4	D	5
5	E	6
6	x	7
7	x	8
8	x	9
9	x	10
10	x	0

Delete D

→

Cursor Implementation

Slot	Element	Next Position
0	x	4
1	A	2
2	B	3
3	C	5
4	x	6
5	E	4
6	x	7
7	x	8
8	x	9
9	x	10
10	x	0

Title

Cursor Implementation

Slot	Element	Next Position
0	x	4
1	A	2
2	B	3
3	C	5
4	x	6
5	E	4
6	x	7
7	x	8
8	x	9
9	x	10
10	x	0

Delete C

→

Cursor Implementation

Slot	Element	Next Position
0	x	3
1	A	2
2	B	5
3	x	4
4	x	6
5	E	3
6	x	7
7	x	8
8	x	9
9	x	10
10	x	0

Cursor Implementation - Diagram

Slot	Element	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		8
8		9
9		10
10		0

Slot	Element	Next
0	-	6
1	B	9
2	F	0
3	Header	7
4	-	0
5	Header	10
6	-	4
7	C	8
8	D	2
9	E	0
10	A	1

If L = 5, then L represents list (A, B, E)

If M = 3, then M represents list (C, D, F)

21CSC201J
DATA STRUCTURES AND ALGORITHMS

UNIT-2
Topic : Introduction to List

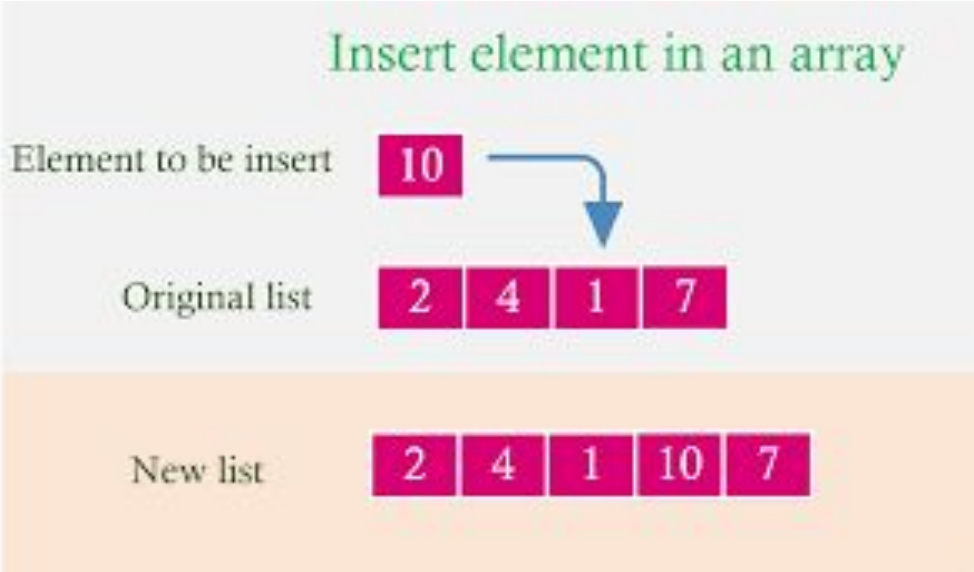
List ADT - Operations

- Insertion
- Deletion
- Searching
- Sorting
- Merging
- Traversal

Insertion

- Insert the elements either at starting position or at the middle or at last or anywhere in the array.
- After inserting the element in the array, the positions or index location is increased

Insertion

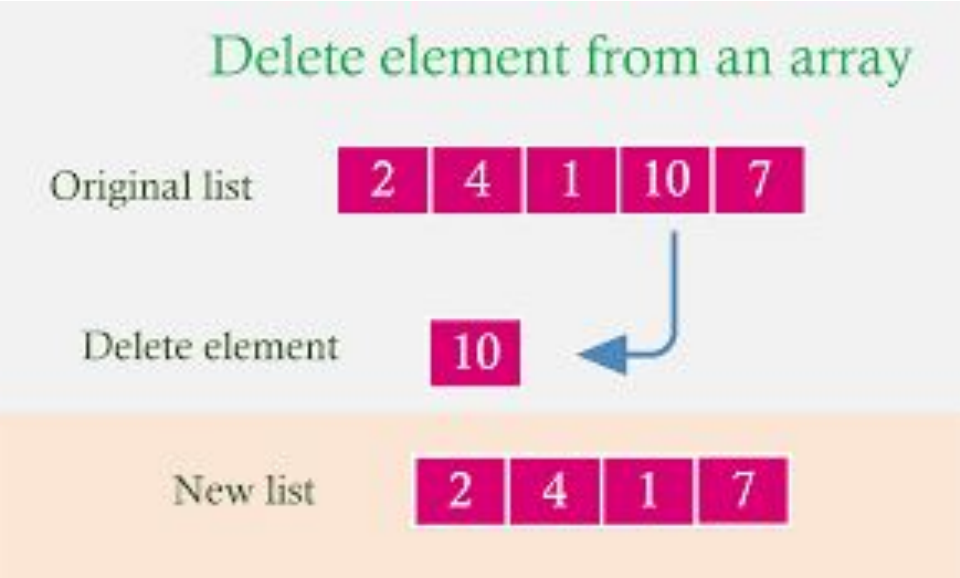



```
for(i=size-1;i>=pos-1;i--)
    student[i+1]=student[i];
    student[pos-1]= value;
```

Deletion

- Input the size of the array arr[] using num, and then declare the pos variable to define the position, and i represent the counter value.
- Input the position of the particular element to delete from an array.
- Remove the particular element and shift the rest elements position to the left side in an array.
- Display the resultant array after deletion or removal of the element from an array.

Deletion




```
for (i = pos - 1; i < num -1; i++)
{
    arr[i] = arr[i+1]; // assign arr[i+1] to arr[i]
}
```

Searching

- One of the basic operations to be performed on an array is searching.
- Searching an array means to find a particular element in the array. The search can be used to return the position of the element or check if it exists in the array.
- The simplest search to be done on an array is the linear search. This search starts from one end of the array and keeps iterating until the element is found, or there are no more elements left (which means that the element does not exist).

Searching

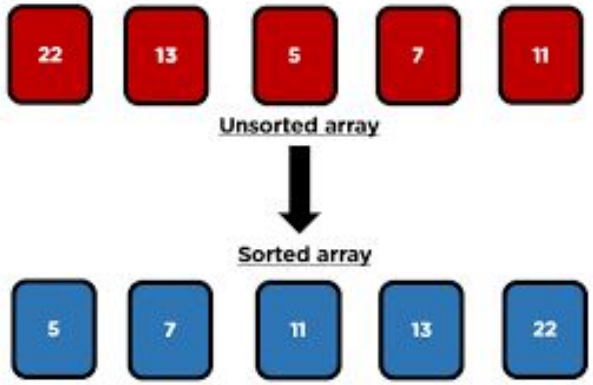
Linear Search

```
int linearsearch (int a [ ], int first, int last, int key)
{
        for (int i = first; i < = last; i ++)
        {
                if (key = = a [i])
                {
                return i;        // successfully found the
                }        // key and return location
        }
        return – 1;              // failed to find key element
}
```

Sorting

- Create an array of fixed size
- Take n, a variable which stores the number of elements of the array, less than maximum capacity of array.
- The array elements are in unsorted fashion
- In the nested loop, the each element will be compared to all the elements below it
- In case the element is greater than the element present below it, then they are interchanged

Sorting

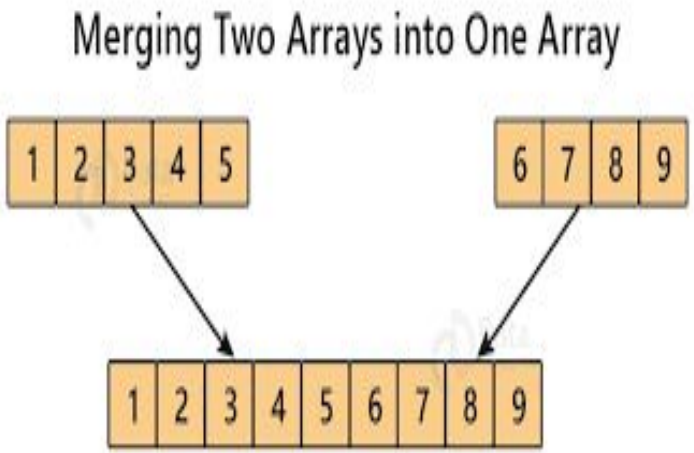



```
for (i = 0; i < n; ++i){
   for (j = i + 1; j < n; ++j){
      if (num[i] > num[j]){
         a = num[i];
         num[i] = num[j];
         num[j] = a;
      }
   }
}
```

Merging

- Input the length of both the arrays.
- Input the arrays elements from user.
- Copy the elements of the first array to the merged array when initializing it.
- Copy the elements of the second array to the merged array while initializing the second array.
- Display the merged array

Merging

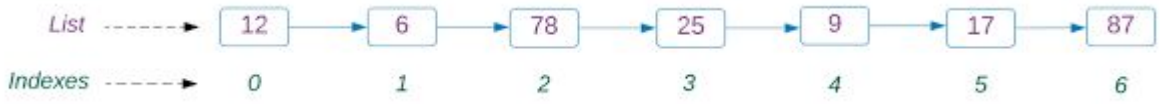



```
for(i=0; i<size1; i++)
{
    scanf("%d", &arr1[i]);
    merge[i] = arr1[i];
}
k = i;
printf("\nEnter Array 2 Size: ");
scanf("%d", &size2);
printf("Enter Array 2 Elements: ");
for(i=0; i<size2; i++)
{
    scanf("%d", &arr2[i]);
    merge[k] = arr2[i];
    k++;
}
```

Traversal

- Traversal means accessing each array element for a specific purpose, either to perform an operation on them , counting the total number of elements or else using those values to calculate some other result.

- Since array elements is a linear data structure meaning that all elements are placed in consecutive blocks of memory it is easy to traverse them.

Traversal

```c
#include <stdio.h>
#include <stdlib.h>

void main()
{
    int list[5],i;
    /* Getting Array Elements From User*/
    printf("Enter array elements: \n");
    for(i=0;i<5;i++){

        scanf("%d",&list[i]);
        printf("\n");
    }
    /* Traversing Array and Displaying them */
    for(i=0;i<5;i++){
        printf("\n List[%d] = %d",i,list[i]);
    }
    getch();
}
```