Unit-2

Prepared By

Dr. V.R. Balasaraswathi

Dr. K. Arthi

Dr. K. Deepa Thilak

Mrs. V. Lavanya

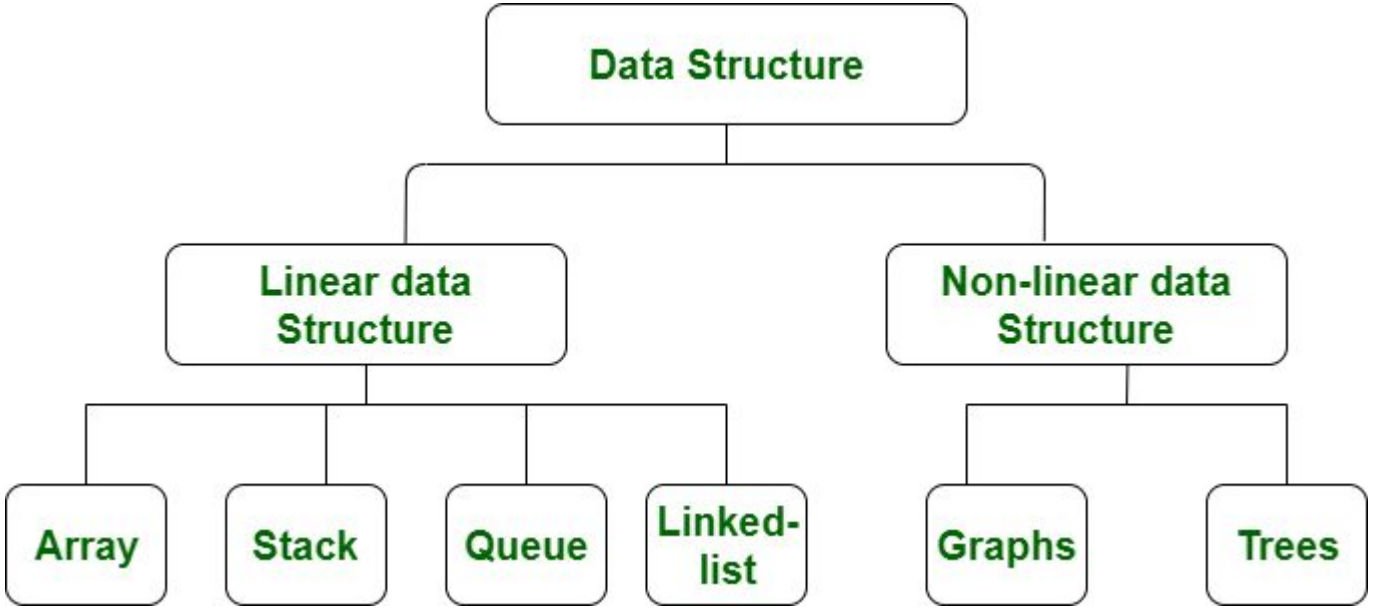
Dr. T. Rajashree

Syllabus

- Arrays
- Operations on Arrays-Insertion and Deletion
- Applications on Array
- Multidimensional Arrays- Sparse Matrix
- Linked List Implementation – Insertion, Deletion and Search
- Applications of Linked List
- Polynomial Arithmetic
- Cursor Based Implementation
- Cursor Based Implementation – Methodology
- Circular Linked List
- Circular Linked List-Implementation
- Applications of Circular List -Joseph Problem
- Doubly Linked List
- Doubly Linked List –Insertion
- Doubly Linked List Insertion variations
- Doubly Linked List –Deletion
- Doubly Linked List - Search

Arrays

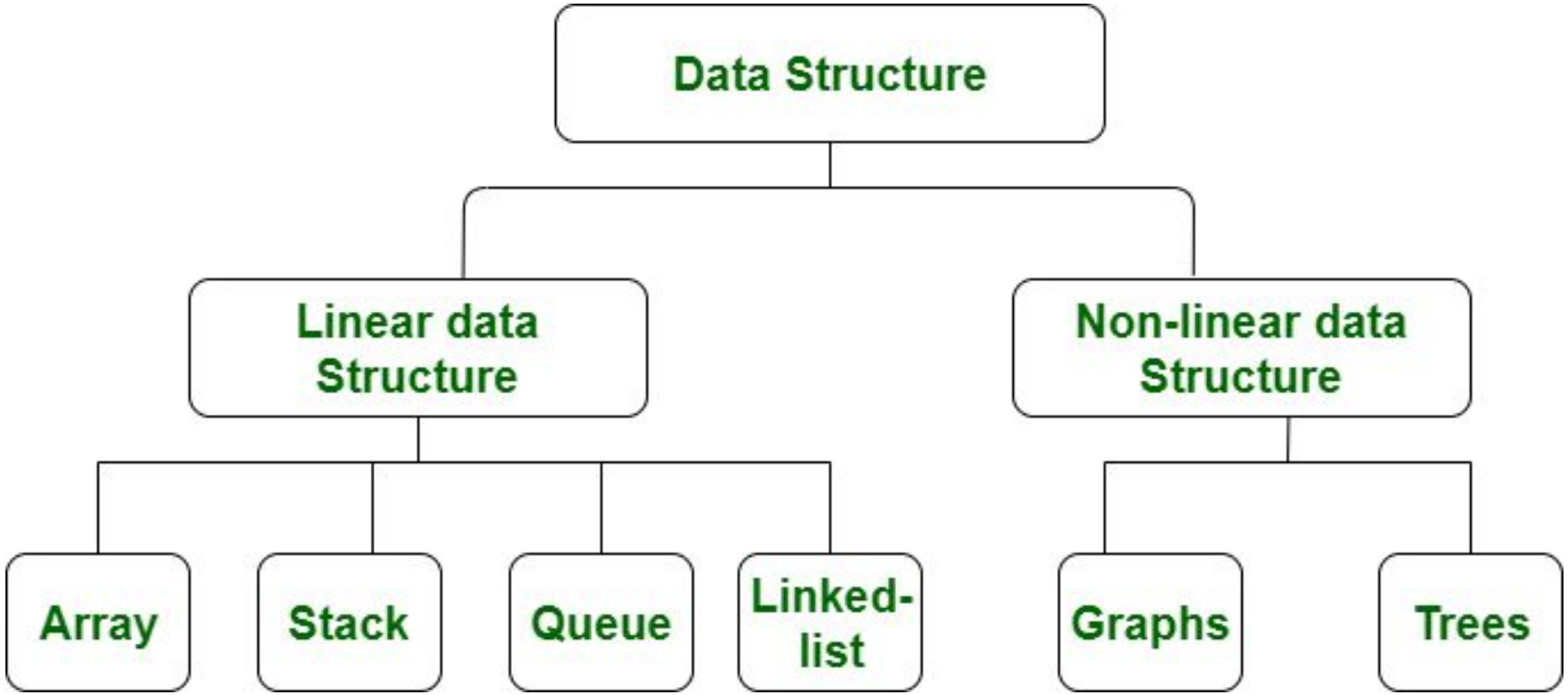
Data Structure-Introduction



Data Structure-Introduction

- DS is classified into
- **Linear DS**
    - Data elements are arranged sequentially or linearly
    - Single level is involved. Therefore, we can traverse all the elements in single run only.
    - Easy to implement because computer memory is arranged in a linear way.
    - Examples are Array, Stack, Queue, Linked List.
- **Nonlinear DS**
    - Data elements are not arranged sequentially or linearly
    - Single level is not involved. Therefore, we can't traverse all the elements in single run only.
    - Not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure.
    - Examples are Trees and Graphs

Data Structure-Introduction…



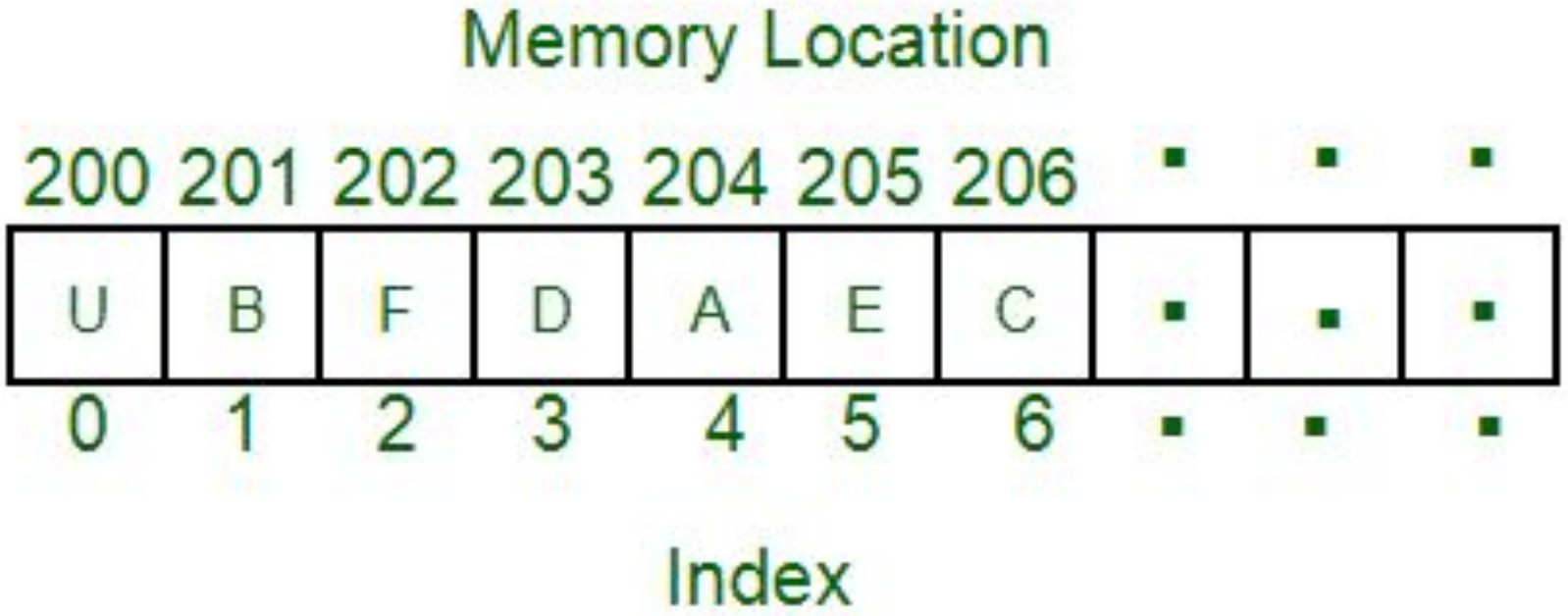
Linear Vs Nonlinear Data Structures

S.No	Linear	Nonlinear
1	Data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	Data elements are attached in hierarchically manner.
2	Single level is involved.	Multiple levels are involved.
3	Its implementation is easy	Its implementation is complex
4	Data elements can be traversed in a single run only.	Data elements can’t be traversed in a single run only.
5	Memory is not utilized in an efficient way.	Memory is utilized in an efficient way.
6	Examples are: array, stack, queue, linked list, etc.	Examples are: trees and graphs.
7	Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.
8	Time complexity often increases with increase in size.	Time complexity often remain with increase in size.

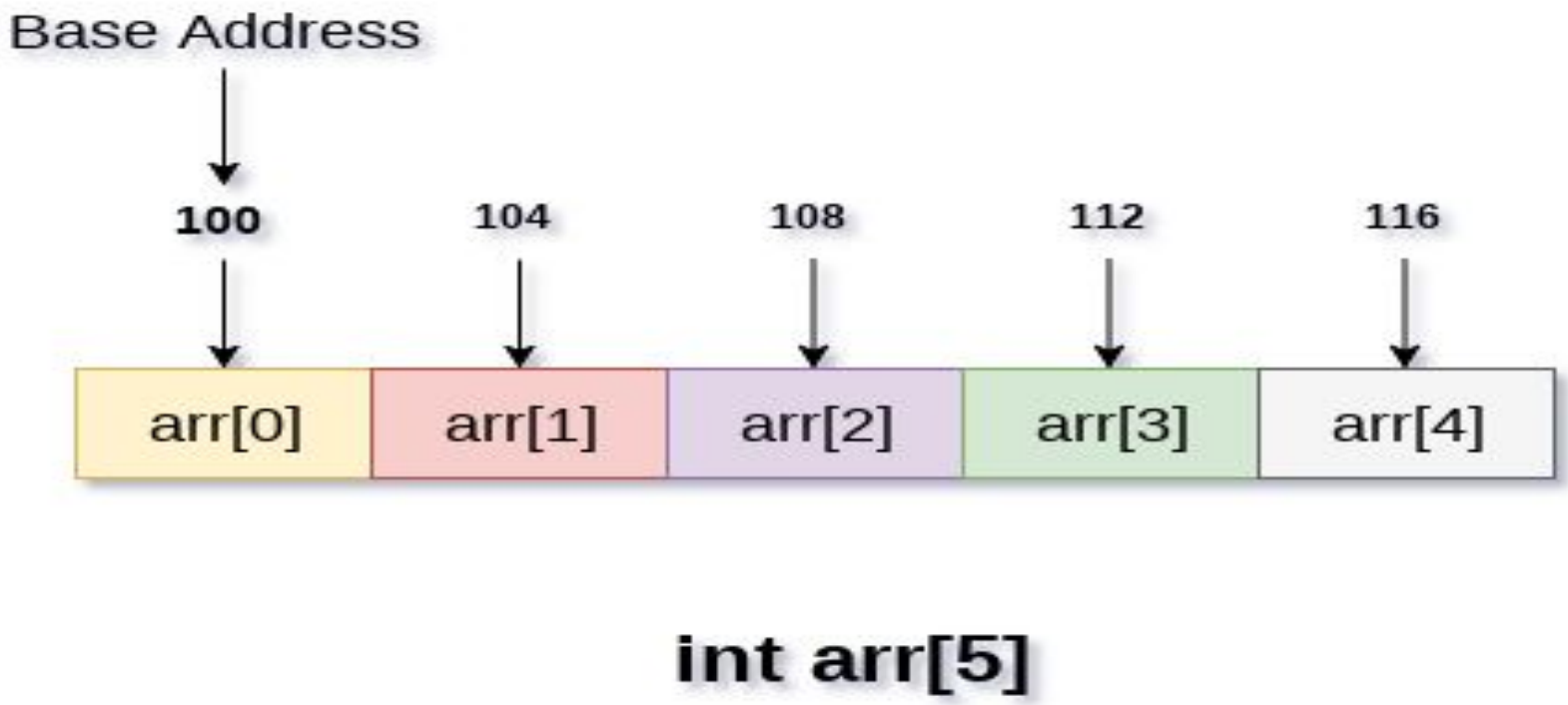
Linear Data Structure-Array

- An array is a collection of items stored at contiguous memory locations.
- The idea is to store multiple items of the same type together.
- This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array

Array…


Memory Location
200 201 202 203 204 205 206
U B F D A E C
0 1 2 3 4 5 6
Index

Array…


Base Address
100
104
108
112
116
arr[0]
arr[1]
arr[2]
arr[3]
arr[4]
int arr[5]

Linear Structure-Operations

- Traversal : Processing each element in the list
- Search : Finding the location of the element with a given value (key)
- Insertion : Adding a new element to the list
- Deletion : Removing an element from the list
- Sorting : Arranging the elements in some type of order
- Merging : Combining two lists into a single list

Linear Arrays

- List of a finite number n of homogeneous data elements , such that
  - Elements are referenced by an index
  - Elements are stored in successive memory locations.

Linear Arrays…

- Number of elements in a array is called as length of the array
- Elements of an array A is denoted by **subscript notation** or
  - $A_1, A_2, A_3, \ldots, A_n$
- By **bracket notation**
  - A[1], A[2], A[3], …, A[N]
- A[1] is called as subscript or index

Linear Arrays…

- Array can be processed using a loop
- Each programming language has its own rules for declaring arrays
- Declaration can be given implicitly or explicitly with 3 items of information
  - 1) Name of the array
  - 2) Data type of the array
  - 3) Index set of the array

Types of indexing in an array

- **0 (zero-based indexing):** The first element of the array is indexed by a subscript of 0.
- **1 (one-based indexing):** The first element of the array is indexed by the subscript of 1.
- **n (n-based indexing):** The base index of an array can be freely chosen. Programming languages allowing n-based indexing also allow negative index values, and other scalar data types like enumerations, or characters may be used as an array index.

Linear Array…

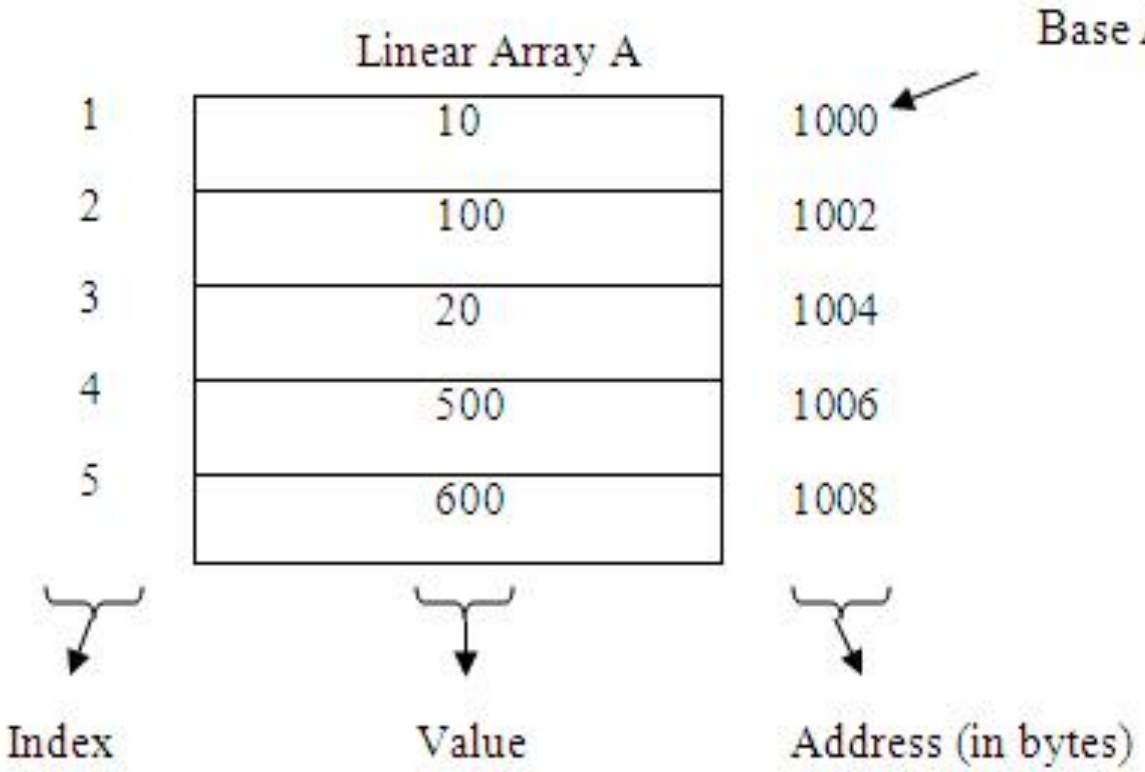
- Some Programming languages(FORTRAN and PASCAL) allocate memory space for arrays **statically**, during program compilation; hence size of the array is fixed during program execution

- Some programming languages(C,C++) allow one to read an integer n and then declare an array with n elements; said to allocate memory **dynamically.**

Arrays as ADT

- Array is a fundamental ADT
- Eg. Object(A,N). An array A is created with set of N elements in it
- A[i] is an item in the array A stored in its i^{th} position
- Arrays are named as A[1], A[2],…., A[N]
- In computers, data is mostly stored in the form of arrays.

Representation of Linear Arrays in Memory



Representation of Linear Arrays in Memory…

- A is Linear Array
- 1, 2, 3, 4 are index values
- 1000 is bases address(starting address of the array) in bytes
- 10, 100, 20, 500, 600 are elements of the array

Operations on arrays

- Array traversal
- Insertion
- Deletion
- Sorting
- Searching

Array Traversal -Eg

```
#include <stdio.h>
main()
{
 int Array[] = {1,3,5,7,8};
 int item = 10, k = 3, n = 5;
int i = 0, j = n;
 printf("The original array elements are :\n");
 for(i = 0; i<n; i++)
   {
     printf("Array[%d] = %d \n", i, Array[i]);
   }
}
```

Output:

```
The original array elements are :
Array[0] = 1
Array[1] = 3
Array[2] = 5
Array[3] = 7
Array[4] = 8
```

Insertion

- Insertion refers to addition of element to the array
- A new element can be added at the beginning, end, or any given index of array.
- Inserting an element at the end of the array can be easily done provided memory space allocated is large enough to accommodate the element.
- Inserting an element in the middle of the array is somewhat difficult
  - Half of the element is must be moved downward to new loactions to accommodate the new element to keep the order of other elements.

Insertion…

```
#include <stdio.h>
main()
 {
  int LA[] = {1,3,5,7,8};
  int item = 10, k = 3, n = 5;
   int i = 0, j = n;
    printf("The original array elements are :\n");
  for(i = 0; i<n; i++)
        printf("LA[%d] = %d \n", i, LA[i]);
       n = n + 1;
   while( j >= k)
 {     LA[j+1] = LA[j];      j = j - 1; }
   LA[k] = item;
printf("The array elements after insertion :\n");
   for(i = 0; i<n; i++)
      printf("LA[%d] = %d \n", i, LA[i]);  }
```

OUTPUT:

The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8

Insertion…

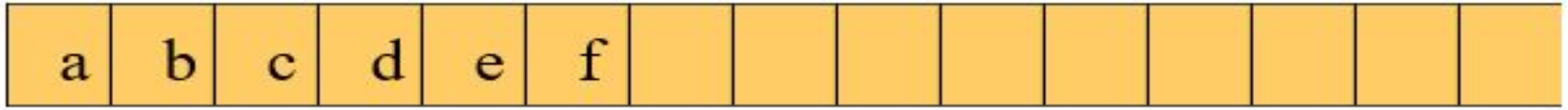
size = 5

add(6,f)

```
add(k,val)
for(i=size-1,i>=k,i--)
    x[i+1]=x[i];
```

x[k]=val; size++;

size = 6



Deletion

- Deletion refers to removing element from the array
- Deleting an element at the end of the array presents no difficulties but deleting an element in the middle requires that each subsequent element be moved one location upward

Deletion…

```
#include <stdio.h>
void main()
{
 int LA[] = {1,3,5,7,8};
  int k = 3, n = 5;
  int i, j;
    printf("The original array elements are :\n");
   for(i = 0; i<n; i++)
        printf("LA[%d] = %d \n", i, LA[i]);
   j = k;
 while( j < n)
 {      LA[j-1] = LA[j];    j = j + 1;     }
   n = n -1;
printf("The array elements after deletion :\n");
   for(i = 0; i<n; i++)
         printf("LA[%d] = %d \n", i, LA[i]);
}
```

Output:

The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8

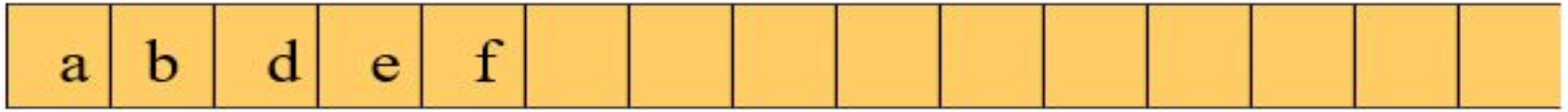
Deletion…

size = 6

remove(2)

```
remove(k)
val=x[k];
for(i=k,i<size,i++)
    x[i]=x[i+1];
size--; return(val);
```

size = 5



Searching

- Search for an array element based on its value or its index.

Algorithm

1.Start

2.Set j=0

3.Repeat steps 4 and 5 while J<N

4.If LA[J] is equal to item then goto step 6

5.Set J=J+1

6.Print J, item

7.Stop

Searching…

```
#include <stdio.h>
void main()
 {
 int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
  int i = 0, j = 0;
  printf("The original array elements are :\n");
   for(i = 0; i<n; i++)
        printf("LA[%d] = %d \n", i, LA[i]);
while( j < n)
{
     if( LA[j] == item )
        break;
      j = j + 1;
   }
   printf("Found element %d at position %d\n", item, j+1);
}
```

Output:

The original array elements are
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3

Applications of arrays

Applications

Arrays have a number of useful applications. These applications are

- Arrays are widely used to implement mathematical vectors, matrices, and other kinds of rectangular tables.
- Many databases include one-dimensional arrays whose elements are records.
- Arrays are also used to implement other data structures such as strings, stacks, queues, heaps, and hash tables.
- Arrays can be used for sorting elements in ascending or descending order

Multidimensional Arrays

- Dynamically allocating a 2-d Array
- Malloc() can be used to allocate a block of memory which can simulate an array.
- Similarly we can do the same to simulate multidimensional arrays.
- If we are not sure of the number of columns that the array will have, then we will first allocate memory for each row by calling malloc.
- Each row will then be represented by a pointer.

Multidimensional Arrays…

Example

```
#include #include <stdio.h
int main()
{ int **arr, i, j, ROWS, COLS;
printf("\n Enter the number of rows and columns in the array: ");
scanf("%d %d",ROWS,COLS);
arr = (int **)malloc(ROWS * sizeof(int *));
 if(arr == NULL)
{
printf("\n Memory could not be allocated");
exit(-1);
 }
```

Multidimensional Arrays…

```
for(i=0; i< ROWS; i++)
   for(j = 0; j < COLS; j++)
    scanf("%d",&arr[i][j]);
printf("\n The array is as follows: ");
 for(i = 0; i < ROWS; i++)
    for(j = 0; j < COLS; j++)
       printf("%d",arr[i][j]);
for(i = 0; i < ROWS; i++)
free(arr[i]);
 free(arr);
return 0;
 }
```

Sparse Matrix

What is Sparse Matrix?

- A matrix can be defined with a 2-dimensional array.
- Any array with 'm' rows and 'n' columns 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 matrix is a matrix which contains very few non-zero elements.

Sparse Matrix…

- The 2d array can be used to represent a sparse matrix in which there are three rows named as:
- Row: It is an index of a row where a non-zero element is located.
- Column: It is an index of the column where a non-zero element is located.
- Value: The value of the non-zero ele
- 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 cases**.
- So, instead of storing zeroes with non-zero elements, we only store non-zero elements. T
- This means storing non-zero elements with triples- (Row, Column, value).

Sparse Matrix…

- Harry Markowitz coined the term Sparse Matrix
- Sparse matrices can be useful for computing large-scale applications that dense matrices cannot handle.
- One such application involves **solving partial differential equations by using the finite element method**.

Sparse matrix…

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.

Sparse matrix…

Sparse Matrix is ….

	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

Sparse Matrix Representation

Sparse Matrix Representations

- Triplet Representation (Array Representation)
- Linked Representation

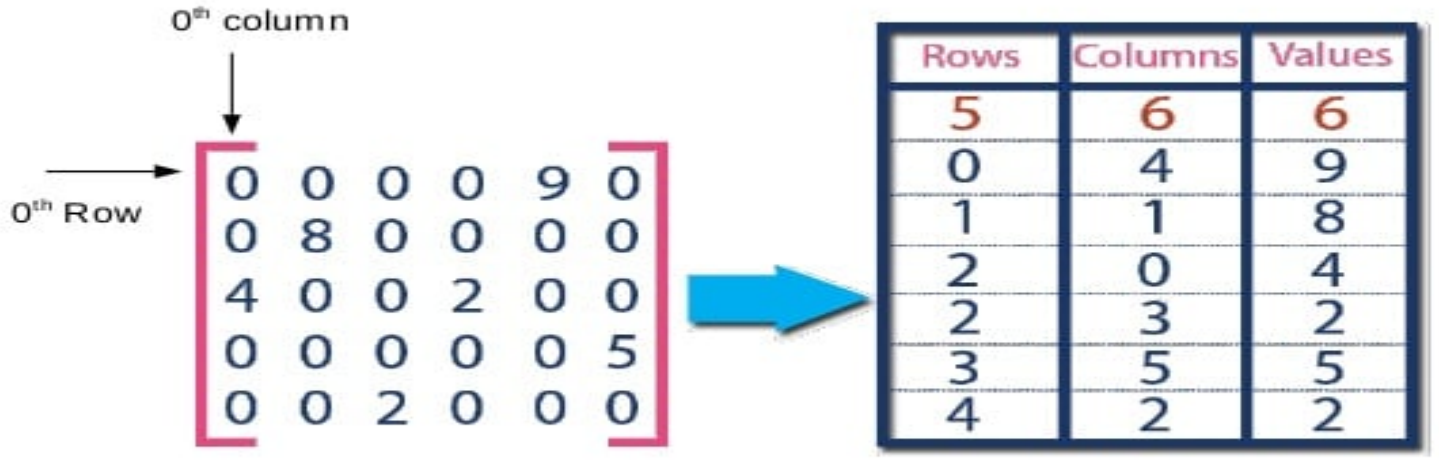
Sparse Matrix Representation…

Triplet Representation (Array Representation)

- In this representation, we consider only non-zero values along with their row and column index values.
- 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.
- For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values.

Sparse Matrix Representation…

Example



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

Sparse Matrix Representation…

Sparse Matrix Representation

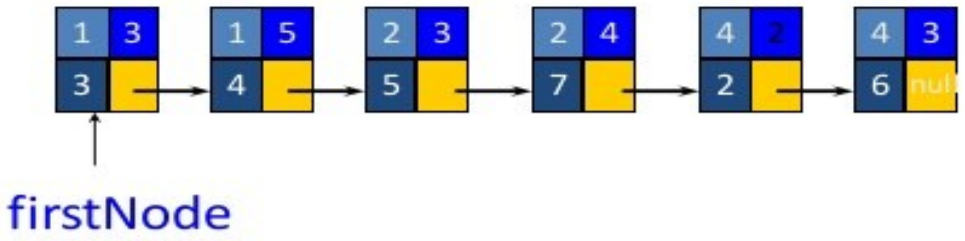
- *a[0].row*: number of rows of the matrix
- *a[0].col*: number of columns of the matrix
- *a[0].value*: number of nonzero entries
- The triples are ordered by row and within rows by columns.

	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

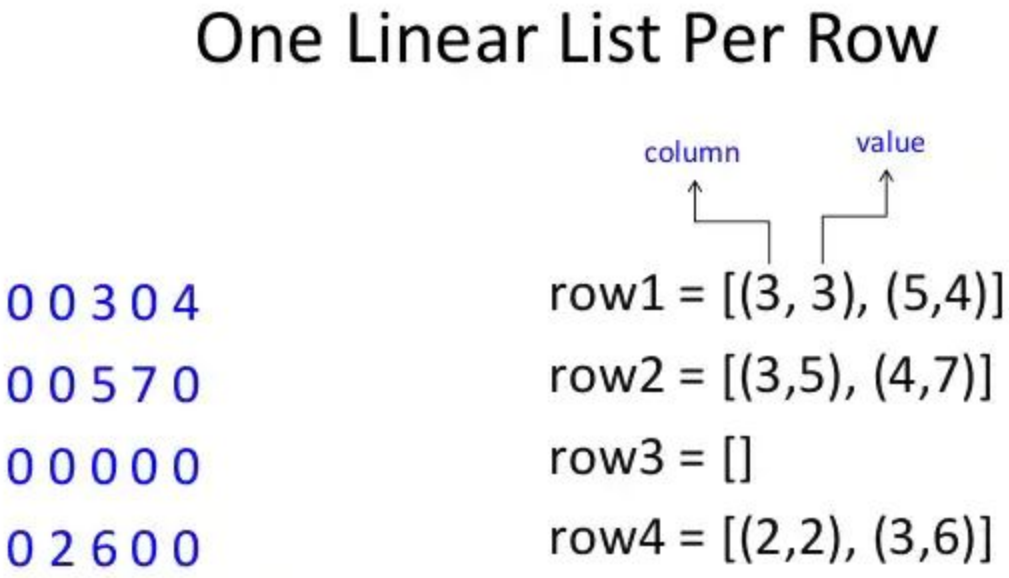
Sparse Matrix Representation…

Single Chain

$$\text{list} = \begin{matrix} \text{row} \\ \text{column} \\ \text{value} \end{matrix} \begin{bmatrix} 1 & 1 & 2 & 2 & 4 & 4 \\ 3 & 5 & 3 & 4 & 2 & 3 \\ 3 & 4 & 5 & 7 & 2 & 6 \end{bmatrix}$$



Sparse Matrix Representation…



Sparse Matrix Representation…

Transposing Matrix

- Interchange the rows and columns
- Each element **a[i][j]** in the original matrix becomes element **b[j][i]** in the transpose matrix

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

Sparse Matrix Example

Transposing a Matrix

- Bad approach: why?

  for each row i
    take element <i, j, value> and
      store it as 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.

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

Matrix Transposition

Transposing a Matrix

- 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++;
            }
}
```

Matrix Transposition…

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];
```

Matrix 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 $\langle i, j \rangle$ element is:

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

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

- Example

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

Matrix 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)
```

Multiplication of Two Sparse Matrices

Multiply two sparse matrices

- D = A×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

Multiplication of Two Sparse Matrices…

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

Multiplication of Two Sparse Matrices…

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;
        }
```

Multiplication of Two Sparse Matrices…

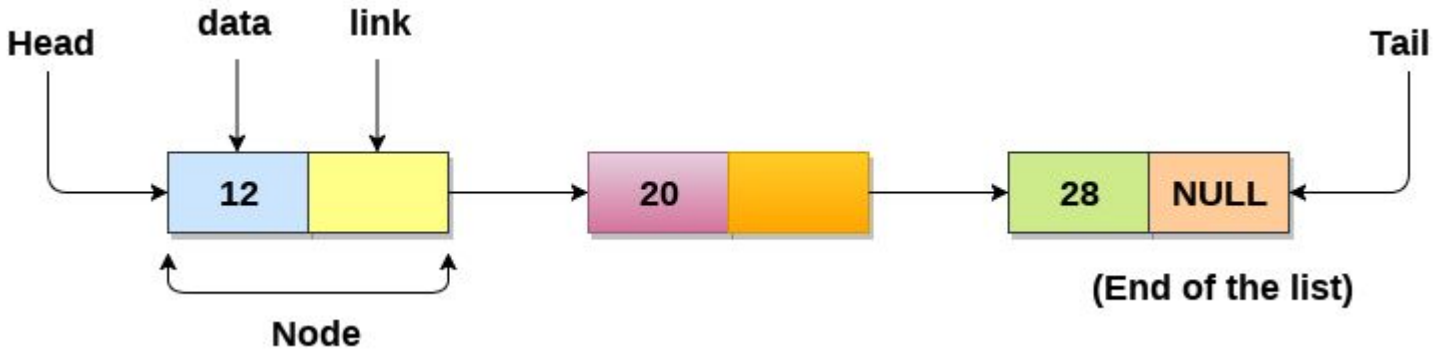
mmult [2]

```
    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;
```

Linked List

Introduction

- Linked List is a linear data structure.
- A list can be defined as an ordered collection
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



- A linked list can change during execution.
  - Successive elements are connected by pointers.
  - Last element points to `NULL`.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.

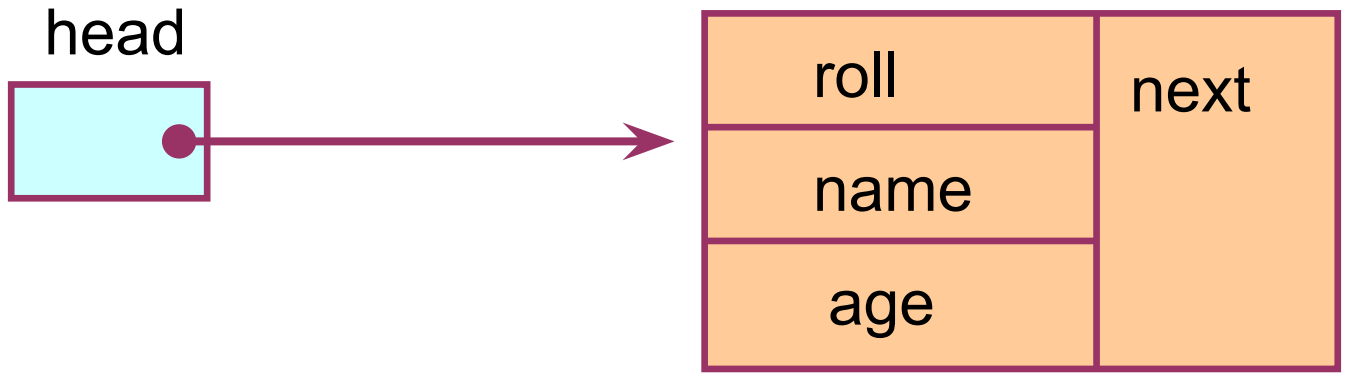
Basic operations of Linked List

- **Creation**- Creates a linked list
- **Insertion** – Adds an element to the list.
- **Deletion** – Deletes an element to the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Creating a Linked List

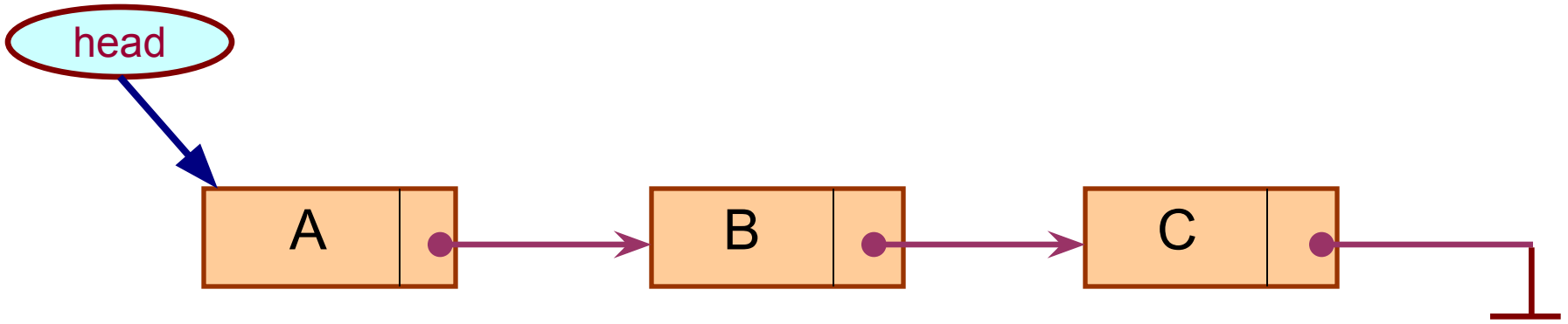
- To start with, we have to create a node (the first node), and make `head` point to it.

- struct node
- {
- **int** data;
- struct node *next;
- };
- struct node *head, *ptr;
- ptr = (struct node *)malloc(sizeof(struct node *));



Contd.

- If there are n number of nodes in the initial linked list:
  - Allocate n records, one by one.
  - Read in the fields of the records.
  - Modify the links of the records so that the chain is formed.

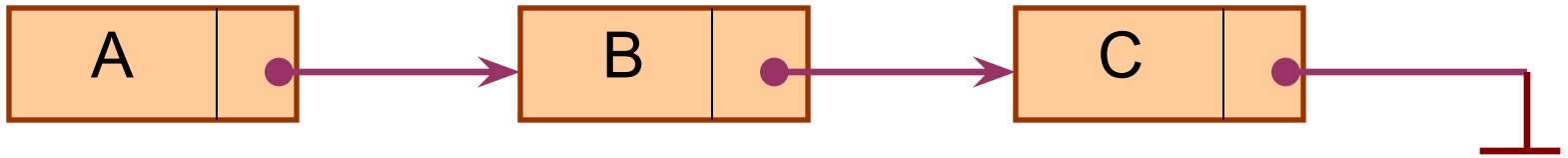


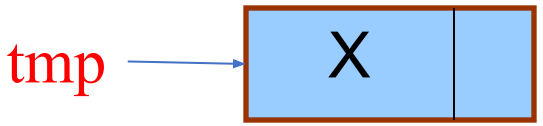
Inserting a Node in a List

- The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

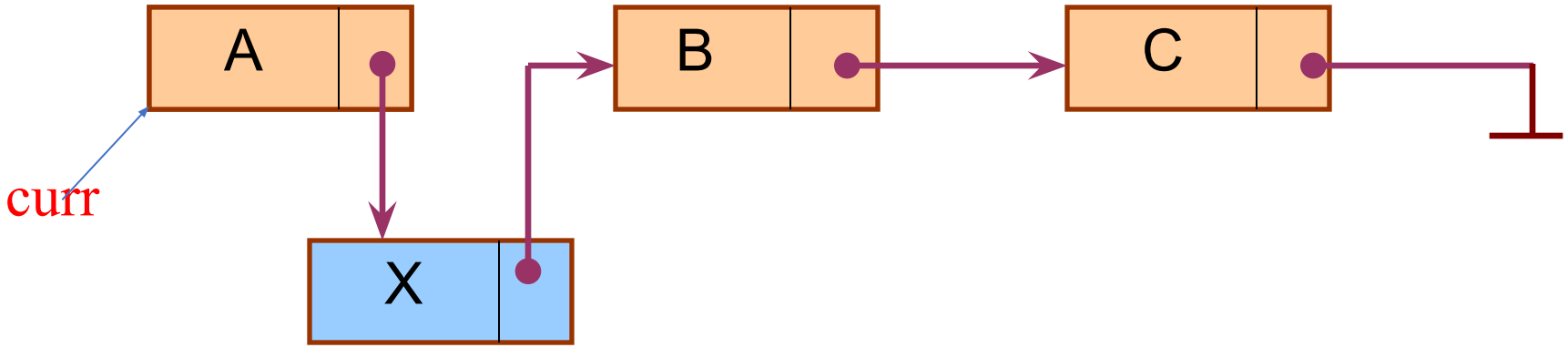
SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

Illustration: Insertion





Item to be inserted



Pseudo-code for insertion

```
typedef struct nd {
   struct item data;
   struct nd * next;
   } node;

void insert(node *curr)
{
node * tmp;

tmp=(node *) malloc(sizeof(node));
tmp->next=curr->next;
curr->next=tmp;
}
```

Contd.

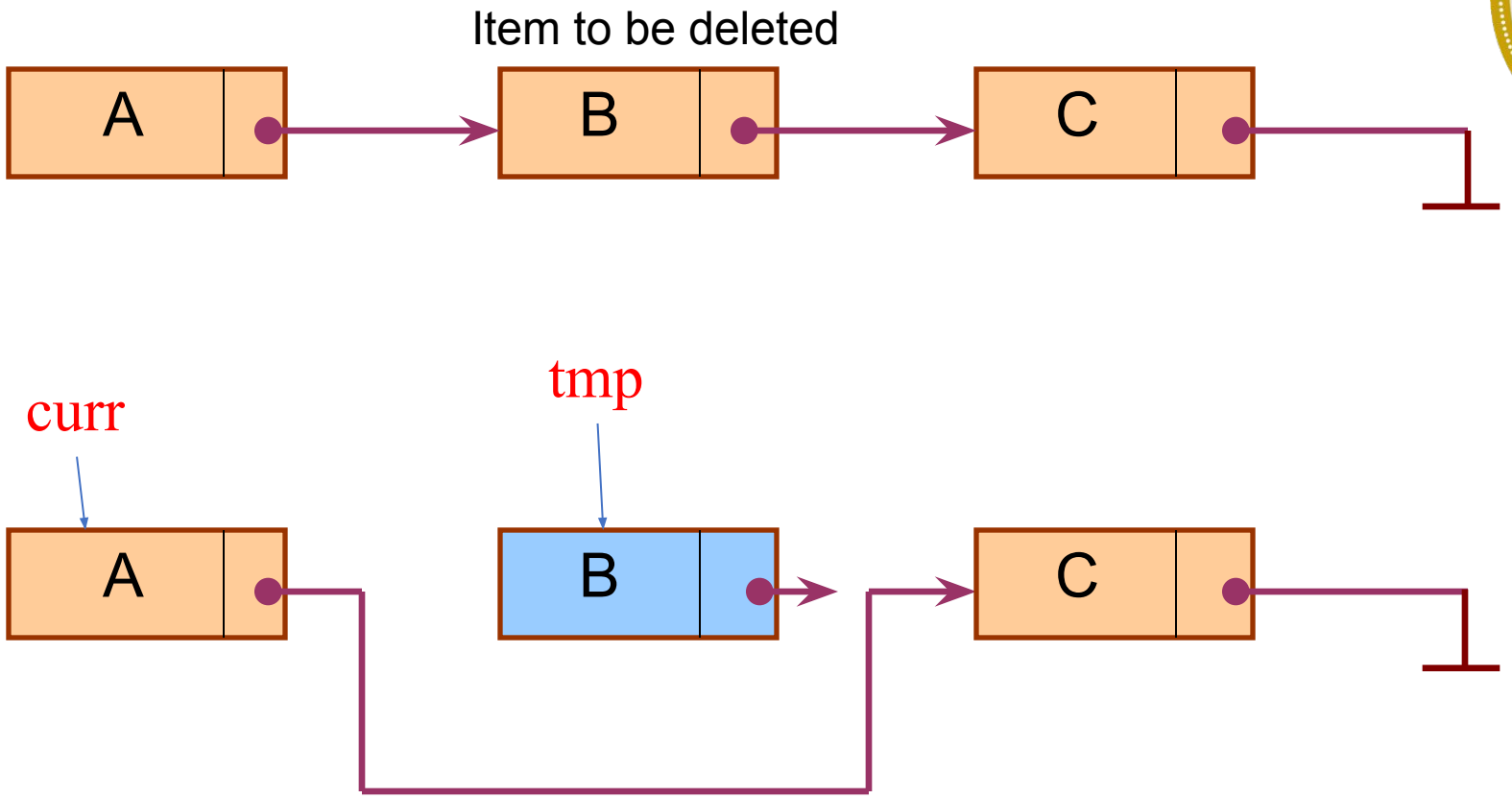
- When a node is added at the beginning,
  - Only one next pointer needs to be modified.
    - *head* is made to point to the new node.
    - New node points to the previously first element.
- When a node is added at the end,
  - Two next pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL.
- When a node is added in the middle,
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.

Deleting a node from the list

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.

Illustration: Deletion



Pseudo-code for deletion

```
typedef struct nd {
   struct item data;
   struct nd * next;
   } node;

void delete(node *curr)
{
node * tmp;
 tmp=curr->next;
curr->next=tmp->next;
free(tmp);
}
```

Application of Linked List

Applications of linked list in computer science

- Implementation of stacks and queues
- Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
- Dynamic memory allocation : We use linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list
- representing sparse matrices

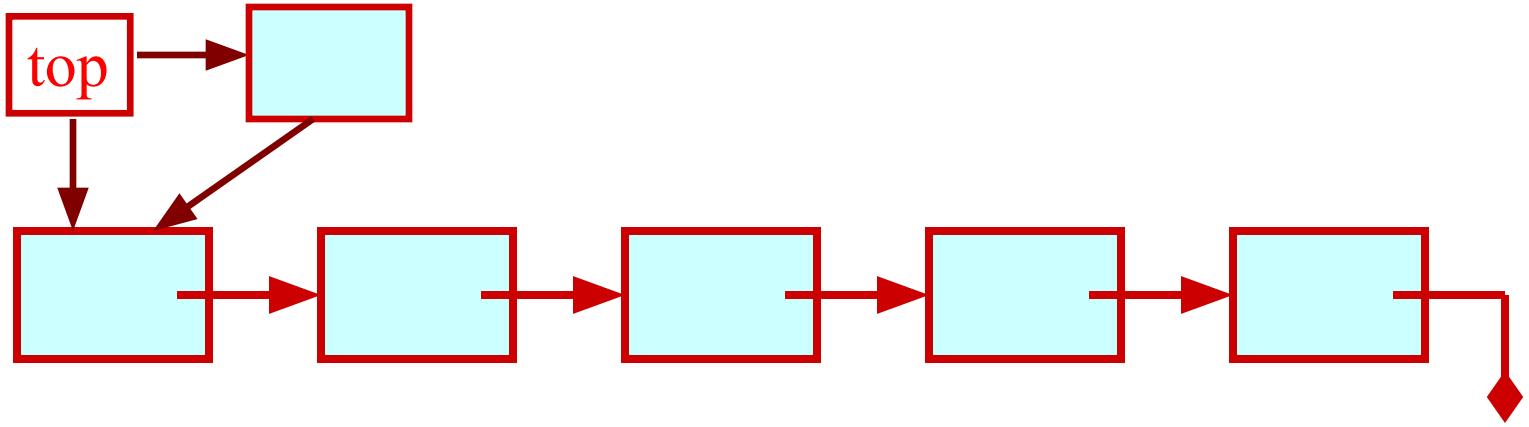
Applications of linked list in real world

- *Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.
- *Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- *Music Player* – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

Stack Implementations: Using Array and Linked List

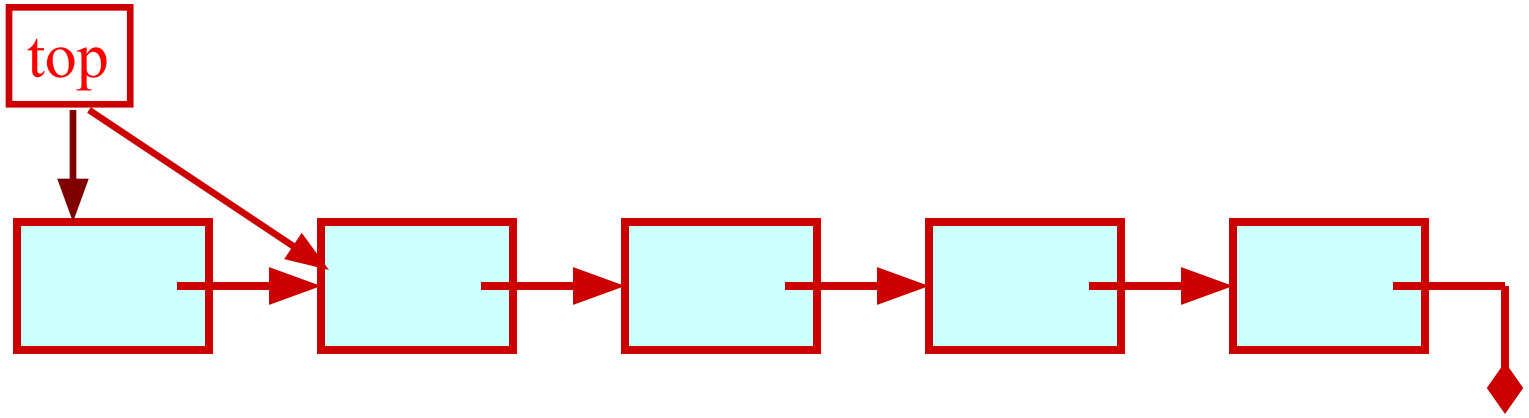
Stack: Linked List Structure

PUSH OPERATION



Stack: Linked List Structure

POP
OPERATION



Basic Idea

- We would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the “top” of the stack.
    - Contains the array index of the “top” element.

Basic Idea

- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable `top` points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Declaration


```
struct lifo
{
    int value;
    struct lifo *next;
};
typedef struct lifo
                  stack;

stack *top;
```

Stack Creation

```
void create (stack **top)
{
   *top = NULL;

   /* top points to NULL,
      indicating empty
      stack                */
}
```

Pushing an element into the stack

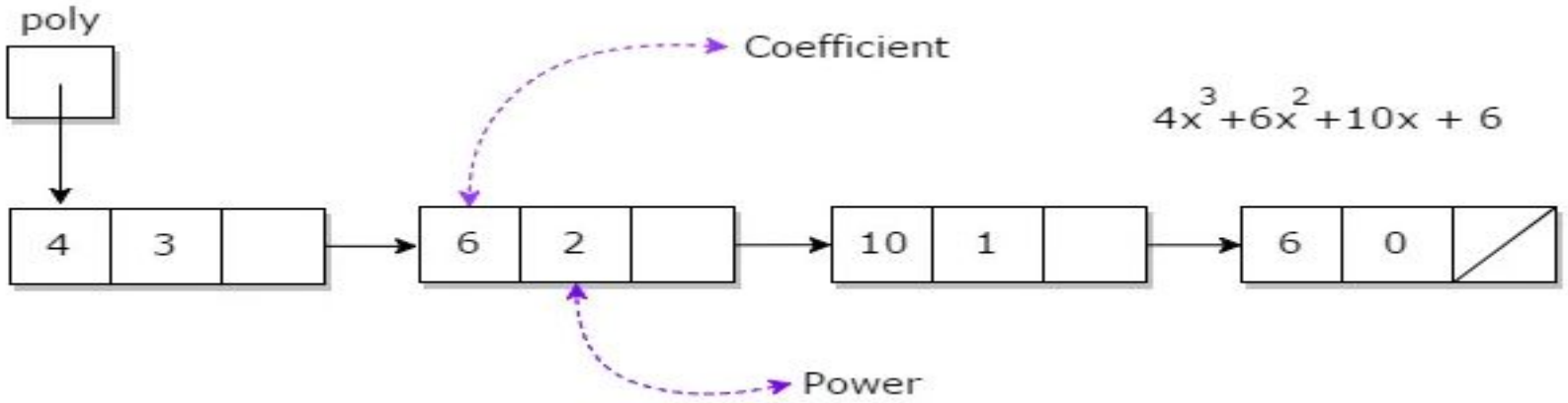
```
void push (stack **top, int element)
{
    stack *new;

    new = (stack *) malloc(sizeof(stack));
    if (new == NULL)
    {
       printf ("\n Stack is full");
       exit(-1);
    }

    new->value = element;
    new->next = *top;
    *top = new;
}
```

Popping an element from the stack

```
int pop (stack **top)
{
    int t;
    stack *p;

    if (*top == NULL)
    {
        printf ("\n Stack is empty");
        exit(-1);
    }
    else
    {
        t = (*top)->value;
        p = *top;
        *top = (*top)->next;
        free (p);
        return t;
    }
}
```

Polynomial Arithmetic using Linked List

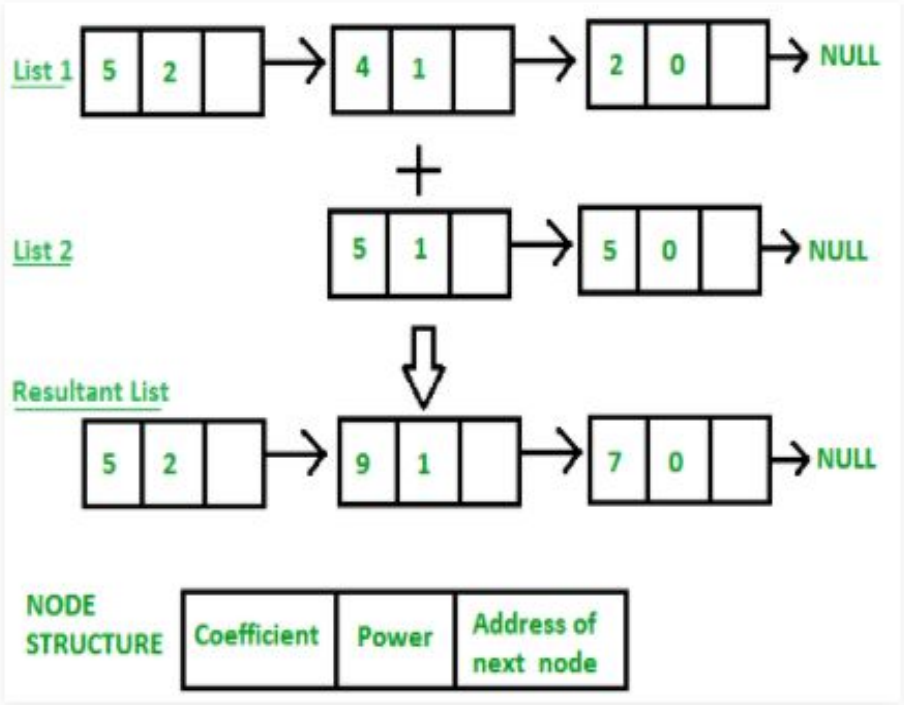
- A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + ... + jx + k$), where a, b, c ...., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.
- A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:
  - The exponent part
  - The coefficient part



Adding two polynomials using Linked List

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 = $-5x^1 + 5x^0$



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

Cursor Based Implementation

Methodology

Motivation

- Some programming languages doesn’t support pointers.
- But we need linked list representation to store data.
- Solution:
  - **Cursor implementation** – Linked list implementation using arrays.

What is Cursor List?

- A CursorList is an array version of a Linked List.
- It is an array of list nodes but instead of each node containing a pointer to the next item in the linked list, each node element in the array contains the index for the next node element.

data	5	3	2	11	9				
next	4	0	1	-1	3				

Methodology

- Convert a linked list based implementation to an array based implementation.
- Instead of pointers, array index could be used to keep track of node.
- Convert the node structures (collection of data and next pointer) to a global array of structures.
- Need to find a way to perform dynamic memory allocation performed using malloc and free functions.

Methodology: 1. Cursor Representation Model

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

Cursor Model:

- Global array of structures
- Each structure has two members
  1. Element
  2. Next
- Figure 2.7.1 has 11 array of structures.

Initial configuration of cursor:

- It contains empty list with 0 as header.
- 0□1□2□3□4□5□6□7□8□9□10
- Node 10 is the last node of the list with next pointer to ‘ 0 ’.

Figure 2.7.1 Array representation of the linked list

Inserting an element in the linked list

- Steps:
  1. Allocate a node from the cursor model.
  2. Place the value in the element field of the node
  3. Adjust the next pointer.

Methodology: 1. Insert an element in a list

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

Figure 2.7.2 Initial Configuration

1. To Allocate a node to store an element

- Find the free space in the cursor
- Select the node pointed by the next field of the 0^{th} node.
- In the figure 2.7.2 0 → next is 1
- Find the location of node in the list after which the new element to be inserted

2. Place the element in the node

- 1→element = 10

3. Adjust the pointers

- 0→next=1→next
- 1→next=0
- 2→next=1

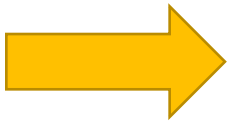
Updated List is 2→1

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

After inserting 20 and 30

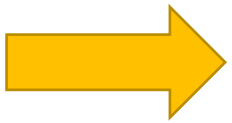
Slot	Element	Next
0	header	3
1	10	0
2	header	1
3	-	4
4	-	5
5	-	6
6	-	8
7	header	0
8	-	9
9	-	10
10	-	0

Insert(20,L)



Slot	Element	Next
0	header	4
1	10	3
2	header	1
3	20	0
4	-	5
5	-	6
6	-	8
7	header	0
8	-	9
9	-	10
10	-	0

Insert(30,L)



Slot	Element	Next
0	header	5
1	10	3
2	header	1
3	20	4
4	30	0
5	-	6
6	-	8
7	header	0
8	-	9
9	-	10
10	-	0

Methodology: 2. Finding an element from the list

- Start from the header of the list.
- Traverse the list by comparing the elements in each node
  - (p□element == x)
- If matching element is found return the slot number else return not found.

Methodology: 2. Finding an element from the list

Slot	Element	Next
0	header	5
1	10	3
2	header	1
3	20	4
4	30	0
5	-	6
6	-	8
7	header	0
8	-	9
9	-	10
10	-	0

find(20,L)

Slot	Element	Next
0	header	5
1	10 No match	3
2	header	1
3	20 match	4
4	30	0
5	-	6
6	-	8
7	header	0
8	-	9
9	-	10
10	-	0

Returns 3

Methodology: 2. Delete an element from a list

- Find the address of the element to be deleted.
- Get the address of the previous element(p) of the deletion element(q)
- Adjust the next element of the previous node.
  - p□next = q□next
- Add the freed node to the empty list
  - q□next=0□next
  - 0□next = q

Methodology: 2. Deleting a node from the list

Slot	Element	Next
0	header	5
1	10	3
2	header	1
3	20	4
4	30	0
5	-	6
6	-	8
7	header	0
8	-	9
9	-	10
10	-	0

Delet(20,L)

Slot	Element	Next
0	header	3
1	10	4
2	header	1
3	-	5
4	30	0
5	-	6
6	-	8
7	header	0
8	-	9
9	-	10
10	-	0

Cursor Based Implementation

1. Convert the node structures (collection of data and next pointer) to a global array of structures.

Declarations for cursor implementation of linked lists

```
Struct Node
{
  ElementType Element;
  Position Next;
};

struct Node CursorSpace[SpaceSize];
typedef NodePtr List;
typedef NodePtr Position;
```

index	Element	Next
0	20	4

2. Need to find a way to perform dynamic memory allocation performed using malloc and free functions.

CursorAlloc()

```
Position CursorAlloc( void )
{
    Position P;

   P = CursorSpace[0].Next;
   CursorSpace[0].Next = CursorSpace[P].Next;

   return P;
}
```

CursorFree()

```
void CursorFree( Position P)
{
   CursorSpace[P].Next=CursorSpace[0].Next;
   CursorSpace[0].Next = P;
}
```

Initializing Cursor Space

- Cursor_space[list] =0;

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

Cursor implementation of linked lists - Example

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

"-" in Element column represents empty cell.

"0" in Next column represents end of the list.

- If L= 5, then L represents list (A, B, E)
- If M= 3, then M represents list (C, D, F)
- Empty list(Always starts from 0): 0→6→4

IsEmpty() and IsLast()

```
/* Return true if L is empty */
int IsEmpty(List L)
{
    return CursorSpace[L].Next == 0;
}

/* Return true if P is the last node in the linked list */
/* Parameter L is unused in this implementation */

int IsLast(Position P, List L)
{
    return CursorSpace[P].Next == 0;
}
```

Insert Routine

```
/* Insert (after legal position P) */
/* Header implementation assumed */
/* Parameter L is unused in this implementation */

void Insert( ElementType x, List L, Position P )
{
  Position TmpCell;

  TmpCell = CursorAlloc( )
  if( TmpCell ==0 )
     fatal_error("Out of space!!!");

  CursorSpace[TmpCell].Element = x;
  CursorSpace[TmpCell]. Next = CursorSpace[P]. Next;
  CursorSpace[P]. Next =TmpCell;
}
```

Find Routine

```
/* Return Position of X in L; 0 if not found */
/* Uses a header node */

Position Find( ElementType x, List L)
{
  Position P;

  P = CursorSpace[L].Next;
  while( P && CursorSpace[P].Element != x )
  P = CursorSpace[P].Next;

  return P;
}
```

Deletion Routine

```
/* Delete first occurrence of x from a list */
/* Assume use of a header node */

void Delete( ElementType x, List L )
{
  Position P, TmpCell;

  P = FindPrevious( x, L );

  if( !IsLast( P, L) )  /* x is found; delete it */
  {
      TmpCell = CursorSpace[P].Next;
      CursorSpace[P].Next = CursorSpace[TmpCell].Next;
      CursorFree( TmpCell );
  }
}
```

Practice Exercises

1. Find the 3rd node from the end of a linked list given in the following cursor representation.
2. Sort the element of the linked list .
3. Find the 1st largest element in the linked list.
4. Find the first 3 smallest element from the linked list.
5. Remove duplicates from the given linked list.

Slot	Element	Next
0	header	6
1	24	3
2	21	1
3	12	4
4	5	9
5	7	0
6	-	8
7	header	2
8	-	10
9	15	5
10	-	0

Review Questions

1. Why we need cursor implementation? **Implement linked list in programming language that do not support pointers**
2. The cursor implementation is _____. **Linked list implementation using arrays.**
3. How to create linked list using cursor implementation? **Global Arrays of structures**
4. Default list in cursor implementation is called _____. **Free list**
5. The header of the free list is at location_____. **0**
6. Node to allocate and released node always occur at _____. **Value at next field of 0**
7. Condition to find the last node of the list_____. **if(cursor[p].next == 0)**
8. Condition to check whether the list is empty or not_____. **if(cursor[L].next == 0)**
9. Code to assign element to the node _____. **cursor[p].Element = x**
10. Code to set a node as last node of a list _____. **cursor[p].next = 0**

ARRAYS AND LISTS

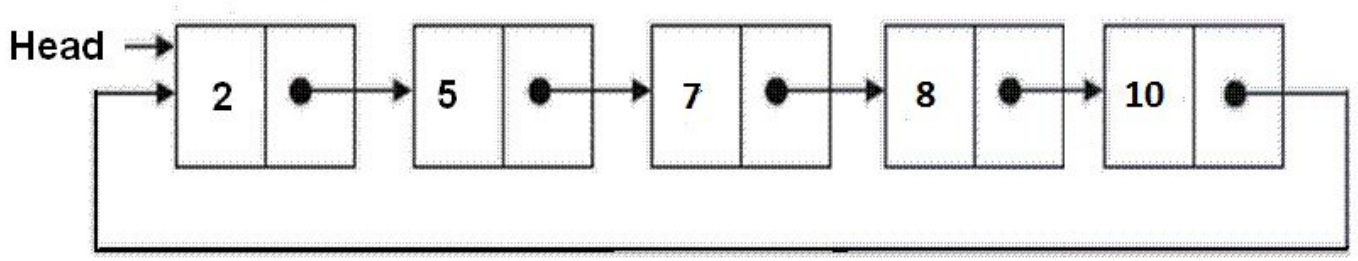
1. Circular Linked List
2. Circular Linked List - Implementation

Circular Linked List

- **Circular linked list** is a linked list where all nodes are connected to form a circle.

- There is no NULL at the end.

- A circular linked list can be a singly circular linked list or doubly circular linked list.



Advantages of Circular Linked Lists

- Any node can be a starting point.
- We can traverse the whole list by starting from any point.
- We just need to stop when the first visited node is visited again.
- Useful for implementation of queue.
- For example, when multiple applications are running on a PC

OPERATIONS ON CIRCULARLY LINKED LIST

- Insertion
- Deletion
- Display

Circular linked list – Creation

```
temp=head;
temp->next = new;
new -> next = head;
```

Insertion in a Circular Linked List

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

Steps to **insert a new node at beginning** of the circular linked list.

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is**Empty (head == NULL)**

Step 3 - If it is **Empty** then, set **head = newNode** and **newNode→next = head**

Step 4 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **'head'**

Step 5 - Keep moving the **'temp'** to its next node until it reaches to the last node (until **'temp → next == head'**).

Step 6 - Set '**newNode → next =head**', '**head = newNode**' and '**temp → next = head**'.

Inserting At the End of the list

- Steps to insert a new node at end of the circular linked list.

Step 1 - Create a **newNode** with given value.

Step 2 - Check whether list is **Empty (head == NULL)**

Step 3 - If it is **Empty** then, set **head = newNode** and **newNode → next = head**

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**

Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next == head**)

Step 6 - Set **temp → next = newNode** and **newNode → next = head**

Inserting At Specific location in the list (After a Node)

Step 1 - Create a **newNode** with given value

Step 2 - Check whether list is **Empty (head == NULL)**

Step 3 - If it is **Empty** then, set **head = newNode** and **newNode → next = head**.

Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**

Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).

Inserting At Specific location in the list (After a Node)

Step 6 - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!!**

Insertion not possible!!!' and terminate the function.

Otherwise move the **temp** to next node.

Step 7 - If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).

Step 8 - If **temp** is last node then set **temp → next = newNode** and

newNode → next = head

Step 9 - If **temp** is not last node then set **newNode → next = temp → next**

and **temp → next = newNode**.

Circular Linked List Insertion (ROUTINE)

Insertion at first:

```
new->next = head
head =new;
temp->next= head;
```

Insertion at middle:

```
n1->next = new;
new->next = n2;
n2->next = head;
```

Insertion at last:

```
n2->next=new;
new->next = head;
```

Circular Linked List
Deletion Operation

- In a circular linked list, the deletion operation can be performed in three ways those are as follows.

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

DELETING FROM BEGINNING OF THE LIST

- The following steps to delete a node from beginning of the circular linked list.

Step 1 - Check whether list is **Empty (head == NULL)**

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize both **'temp1'** and **'temp2'** with **head**

Step 4 - Check whether list is having only one node **(temp1 → next == head)**

Step 5 - If it is **TRUE** then set **head = NULL** and delete **temp1** (Setting **Empty** list conditions)

Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next == head**)

Step 7 - Then set **head = temp2 → next**, **temp1 → next = head** and delete **temp2**

DELETING FROM END OF THE LIST

- The following steps to delete a node from end of the circular linked list.

Step 1 - Check whether list is **Empty (head == NULL)**

Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with **head**

Step 4 - Check whether list has only one Node **(temp1 → next == head)**

Step 5 - If it is **TRUE**. Then, set **head = NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)

Step 6 - If it is **FALSE**. Then, set **'temp2 = temp1 '** and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next == head**)

Step 7 - Set **temp2 → next = head** and delete **temp1**.

DELETING A SPECIFIC NODE FROM THE LIST

- The following steps to delete a specific node from the circular linked list...

  Step 1 - Check whether list is **Empty (head == NULL)**

  Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function

  Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with **head**

  Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set **'temp2 = temp1'** before moving the **'temp1'** to its next node.

  Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'.** And terminate the function.

  Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node **(temp1 → next == head)**

DELETING A SPECIFIC NODE FROM THE LIST

Step 7 - If list has only one node and that is the node to be deleted then set **head = NULL** and delete **temp1 (free(temp1))**.

Step 8 - If list contains multiple nodes then check whether **temp1** is the first node in the list **(temp1 == head)**.

Step 9 - If **temp1** is the first node then set **temp2 = head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next, temp2 → next = head** and delete **temp1**.

Step 10 - If **temp1** is not first node then check whether it is last node in the list **(temp1 → next == head)**.

Step 11- If **temp1** is last node then set **temp2 → next = head** and delete **temp1 (free(temp1))**.

Step 12 - If **temp1** is not first node and not last node then set **temp2 → next = temp1 → next** and delete **temp1 (free(temp1))**.

Circular Linked List - Deletion Operation


```
//delete first item

struct node * deleteFirst() {

//save reference to first link

struct node *tempLink = head;
if(head->next == head){
head = NULL;
return tempLink;
}

//mark next to first link as first

head = head->next;

//return the deleted link

return tempLink;
}
```

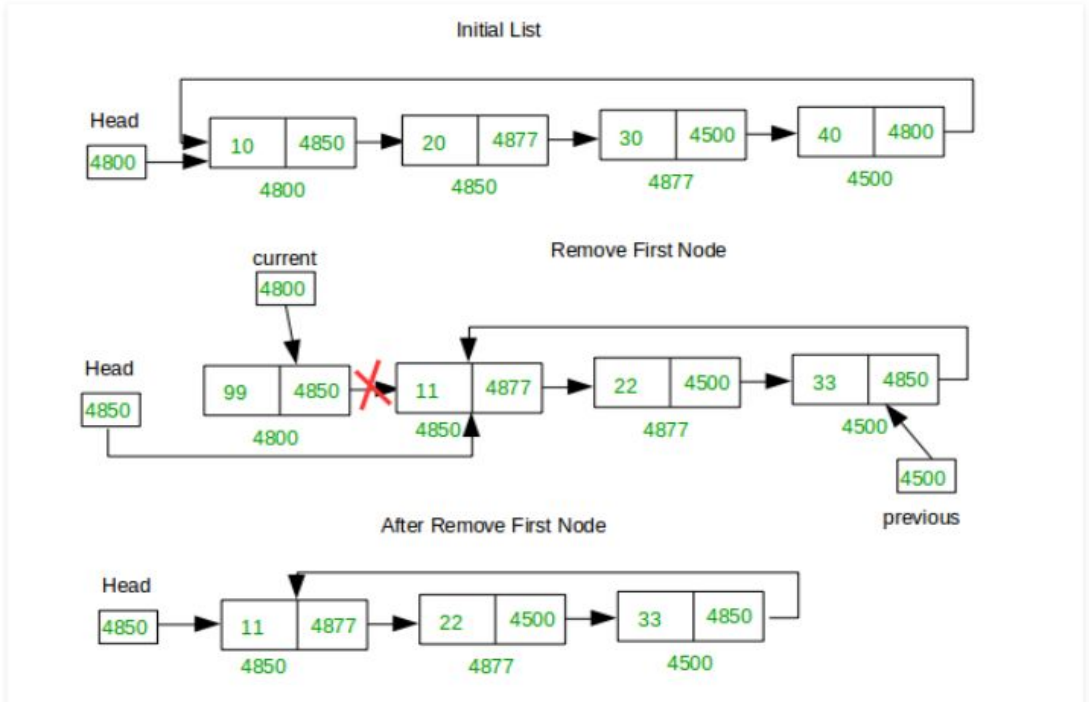
Deleting first node from Singly Circular Linked List

- Given a Circular Linked List. The task is to write programs to delete nodes from this list present at:

- First position.
- Last Position.
- At any given position i.

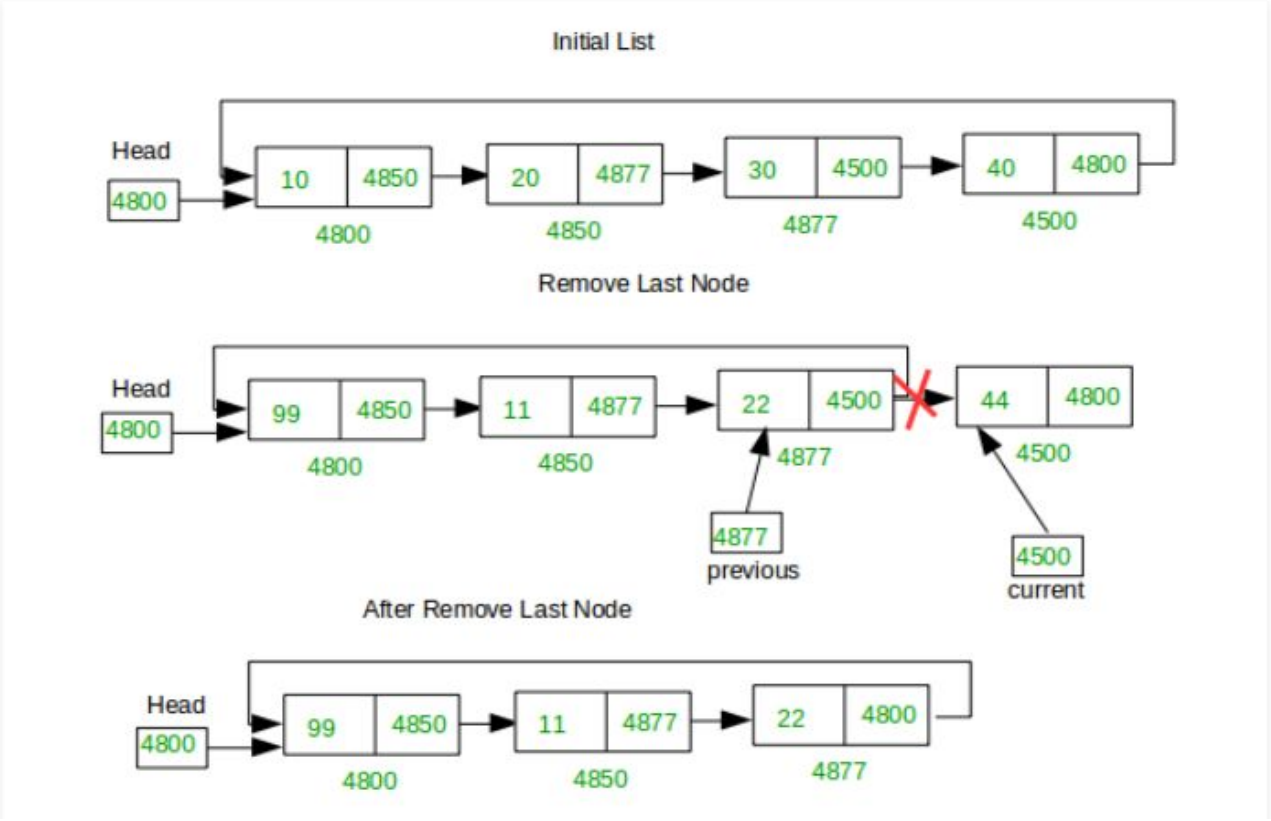
Examples:

```
Input : 99->11->22->33->44->55->66
Output : 11->22->33->44->55->66

Input : 11->22->33->44->55->66
Output : 22->33->44->55->66
```



Deleting the last node of the Circular Linked List

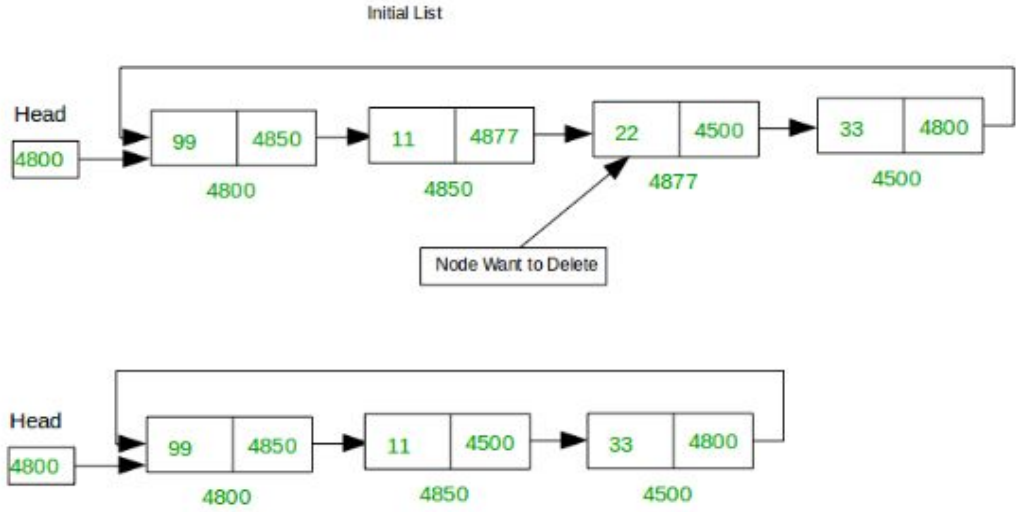

```
Input : 99->11->22->33->44->55->66
Output : 99->11->22->33->44->55

Input : 99->11->22->33->44->55
Output : 99->11->22->33->44
```



Deleting nodes at given index in the Circular linked list


```
Input : 99->11->22->33->44->55->66
        Index= 4
Output : 99->11->22->33->55->66

Input : 99->11->22->33->44->55->66
        Index= 2
Output : 99->11->33->44->55->66
```

Note: 0-based indexing is considered for the list.



Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.

  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.

  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.

1. Applications of Circular List -Joseph Problem
2. Doubly Linked List

Josephus Circle using circular linked list

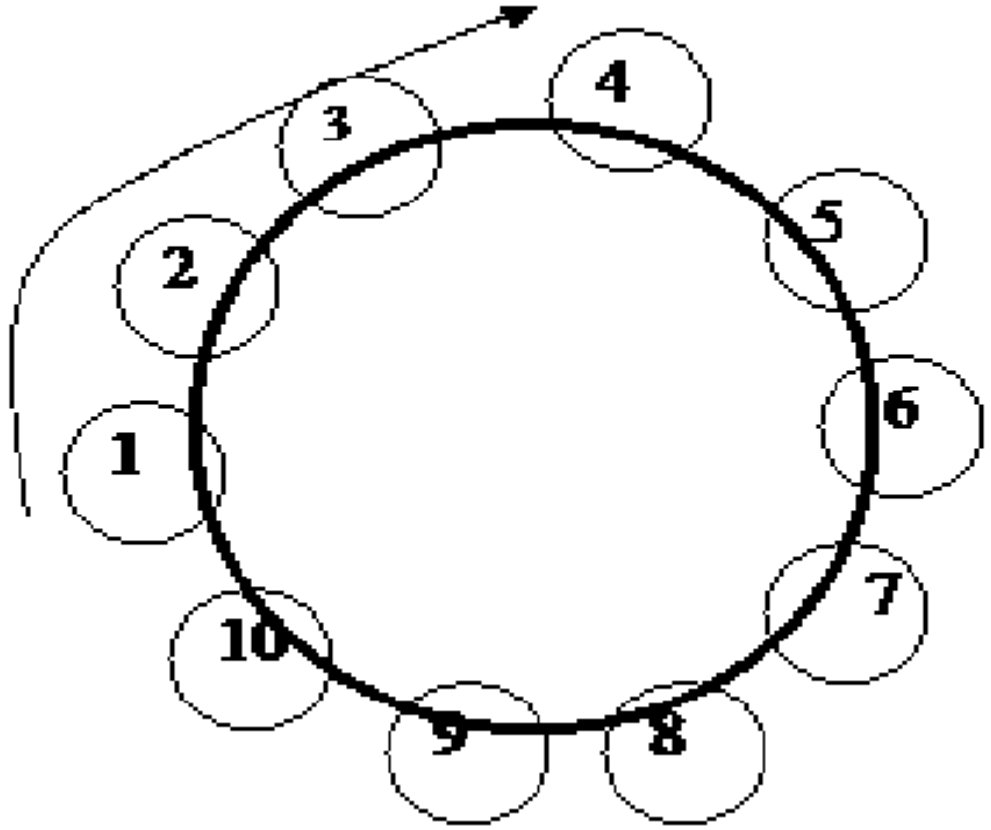
- There are n people standing in a circle waiting to be executed. The counting out begins at some point in the circle and proceeds around the circle in a fixed direction. In each step, a certain number of people are skipped and the next person is executed. The elimination proceeds around the circle (which 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 circle so that you are the last one remaining and so survive.

Examples :

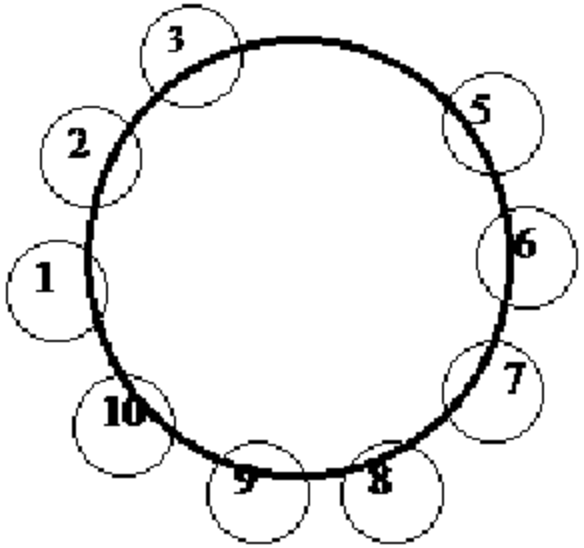
```
Input : Length of circle : n = 4
        Count to choose next : m = 2
Output : 1

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

Example

N=10, M=3



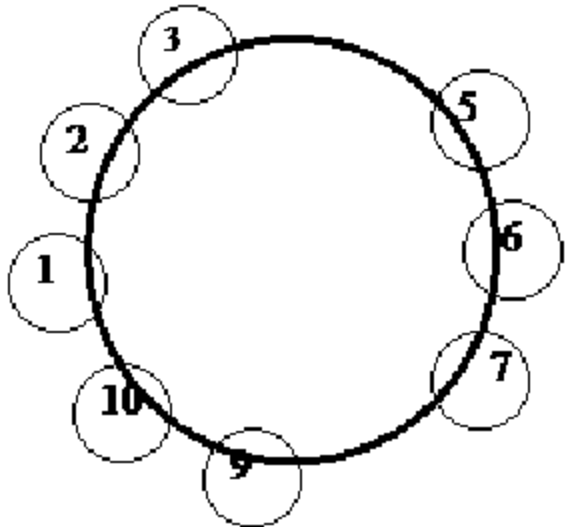
N=10, M=3



4

N=10, M=3

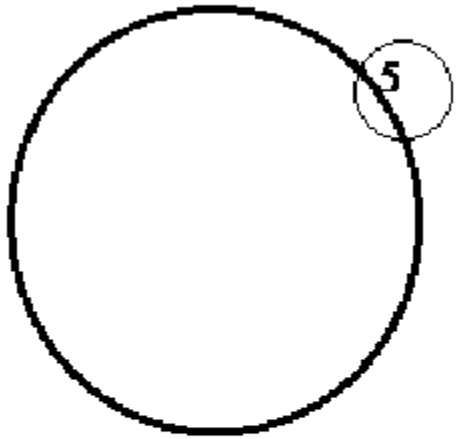
Eliminated



4

8

N=10, M=3



Eliminated

4
8
2
7
3
10
9
1
6

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;
}
```

Exercise

- Write a C program to implement the following operations using Circular Linked List
  - Insertion in all the positions like start , middle and end
  - Deletion in all the positions like start , middle and end
  - Display

Quiz

1. Find the concept which is right for the below condition.“Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again”.
   - Circular Linked List
   - Singly Linked List
   - Stack
   - Queue

2. What is the right statement (condition) to check the last node in the circular linked list. (Assume temp is initialized as temp=head)
   - temp->next=NULL
   - temp->next=head
   - temp=NULL
   - temp=head

Quiz

3. Find the code used to count the nodes in the circular linked list.

```
public int length(Node head)
{
        int length = 0;
        if( head == null)

                return 0;
        Node temp =
      head.getNext();
      while(temp!=head )
      { length++; }
```

```
public int length(Node head)
{
        int length = 0;
        if( head == null)
                return 0;
        Node temp = head.getNext();
        while(temp !=NULL)
        {length++;}
```

```
public int length(Node head)
{
        int length = 0;
        if( head == null)

                return 0;
        Node temp = head.getNext();
        while(temp.getnext()!=NULL)
        {
                length++;
                temp=temp.getNext();
        }
```

```
public int length(Node head)
{
        int length = 0;
        if( head == null)
                return 0;
        Node temp = head.getNext();
        while(temp!=head)
        {
                length++;
                temp=temp.getNext();
        }
```

Doubly Linked List

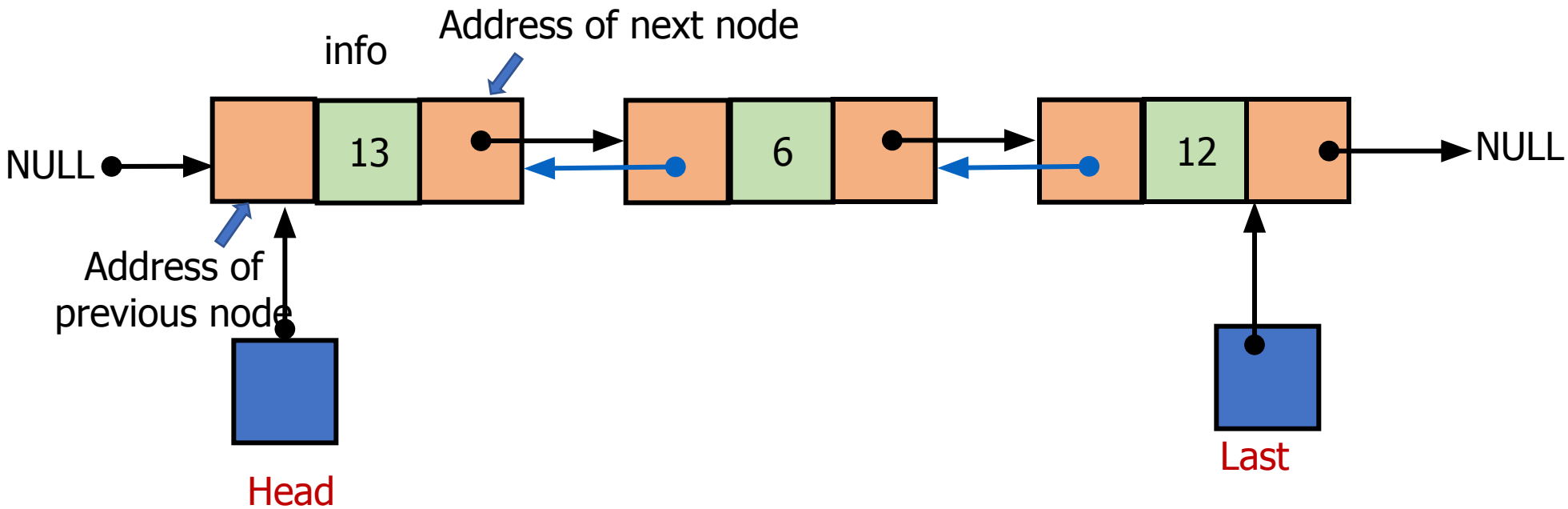
Doubly Linked list - Motivation

- Doubly linked lists are useful for playing video and sound files with “rewind” and “instant replay”

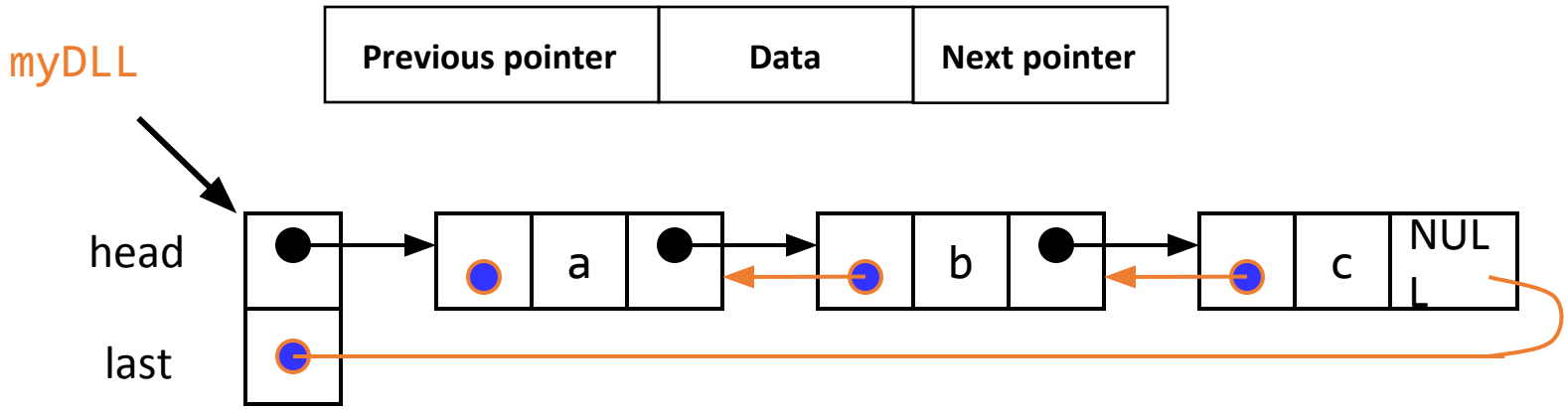
- They are also useful for other linked data which require “rewind” and “fast forward” of the data

Doubly Linked Lists

- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two `NULL`: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



Doubly-linked lists



- Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

Operations - Doubly Linked List

- Create a Node
- Insertion at beginning of a list
- Insertion at end of the list
- Insertion after a given position
- Delete a node from beginning
- Delete a node from end
- Delete a node from any location

Creating a Node

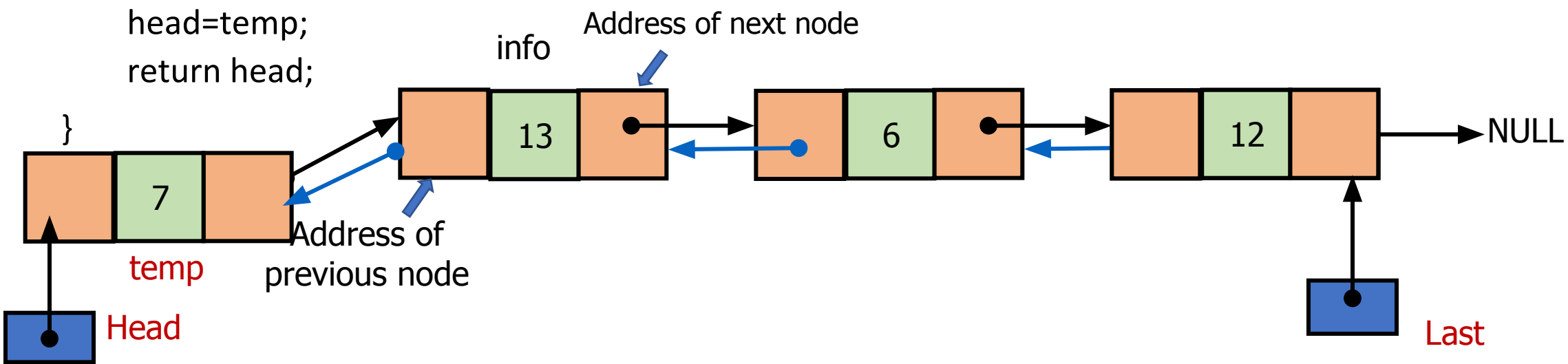
```
struct node
{
      Struct node *previous;
      Int info;
      Struct node *next;
}
void display(struct node *head)
{
Struct node *P;
If (head==NULL)
{
printf(List is empty \n”);
return;
}
P=head;
printf(“List is: \t”);
While(P!=NULL)
{ printf(“%d”, P->info);   P=P->next;  }
}
```

- info: the user's data
- next, previous: the address of the next and previous node in the list

Insertion at Beginning

- Time Complexity –O(1)

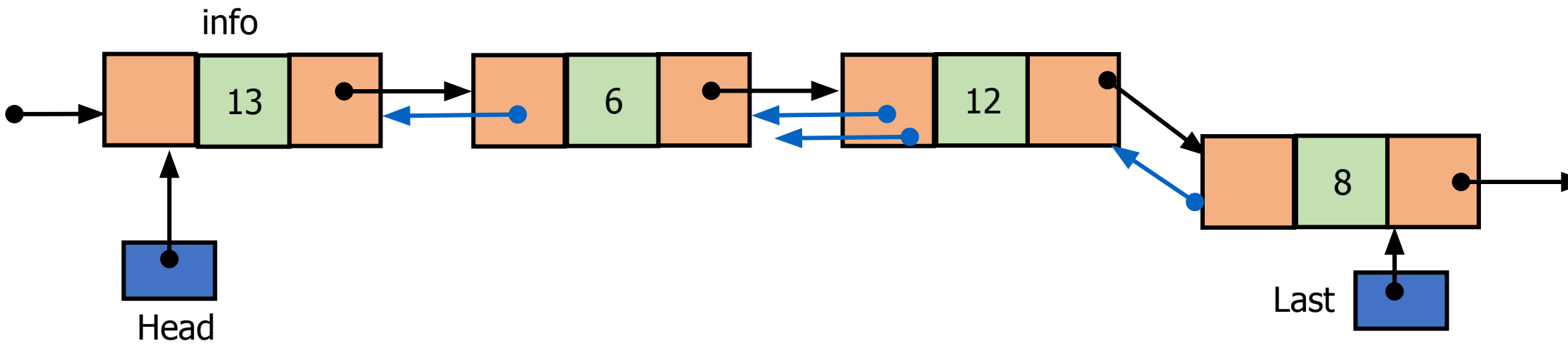
```
Struct node *insertAtBeginning (Struct node *head, int data)
{
Struct node *temp;
temp=(Struct node*)malloc(sizeof (Struct node));
temp->info=data;
temp->previous=NULL;
temp->next=head; //starts pointing to head of list
If(head)
        head->previous=temp;
        head=temp;
        return head;
}
```



Insertion at end

- Time Complexity – O(n)

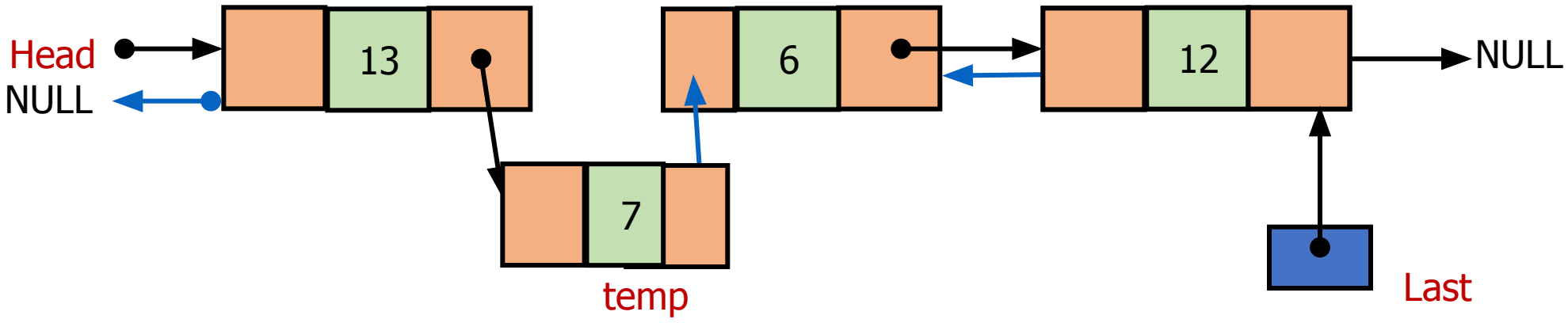
```
struct node *insertAtEnd (struct node *head, int data)
{
struct node *temp, *P;
temp=(struct node*)malloc(sizeof (struct node));
temp->info=data;
P=head;
If(P)
{ while(P->next!=NULL)
{     P = P->next;P->next=temp;  Temp->previous = P;      }
else
      head=temp;
      return head;
}
```



Insertion at a given position

```
Struct node *insertAtPostion(Struct node *head, int data, int item)
{
Struct node *temp, *P;
temp=(Struct node*)malloc(Sizeof(struct node));
temp->info=data;
P=head;
While(P!=NULL)
{
If(P->info==item)
{
        temp->previous=P;
        Temp->next=P->next;
        If(P->next->previous=temp);
        P->next=temp;
        Return head;
}
P= P->next;
}
Printf("%d not present in the list \n \n",
item);
return head;
}
```

- Time Complexity –O(n)

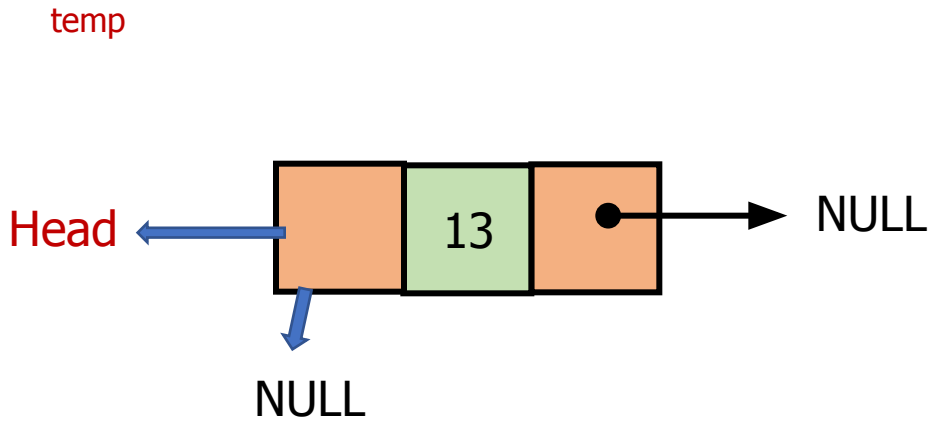


Deleting a node from the list

```
Struct node *delete(Struct node *head, int data)
{
Struct node *temp;
If(head==NULL)
{
printf("List is empty");
return head;
}
If(head->next==NULL)
{
If(head->info==data)
{
temp=head;
head=NULL;
free(temp);
return head;
}
```

```
Else
{
printf("Element %d not found \n", data);
return head;
}
}
```

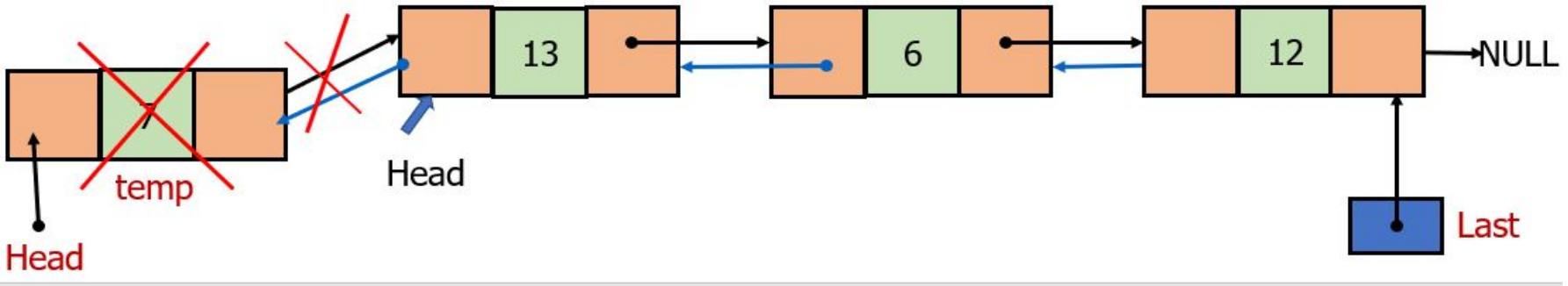
Time Complexity – O(1)



Deletion of first node

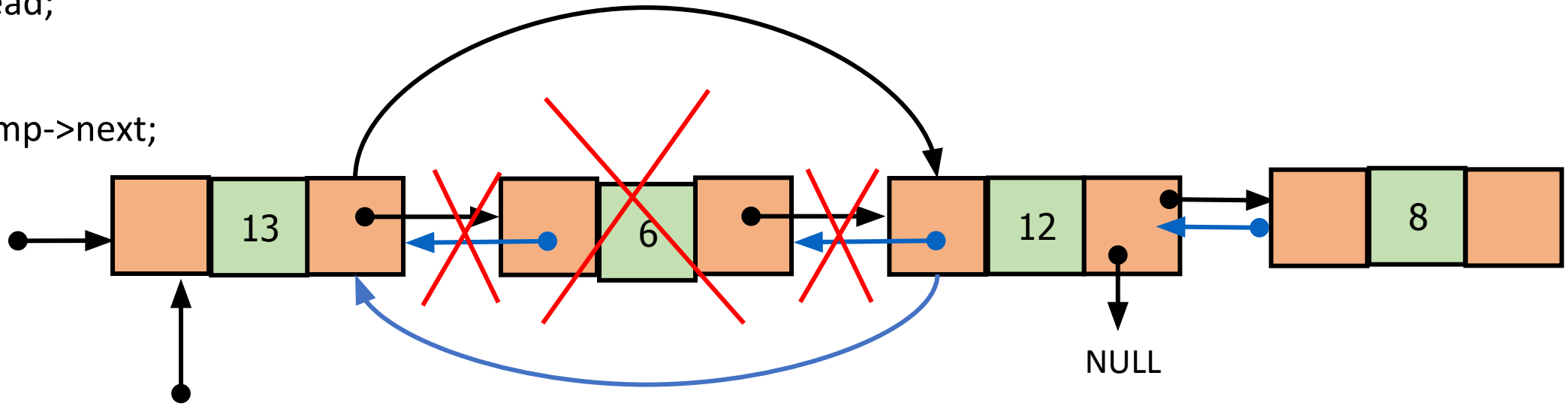
```
Struct node *delete(Struct node *head, int data)
{
Struct node *temp;
If(head==NULL)
{
printf("List is empty");
return head;
}
If(head->info==data)
{
    temp=head;
    head=head->next;
    head->previous=NULL;
    free(temp);
    return head;
}
```

- Time Complexity – O(1)



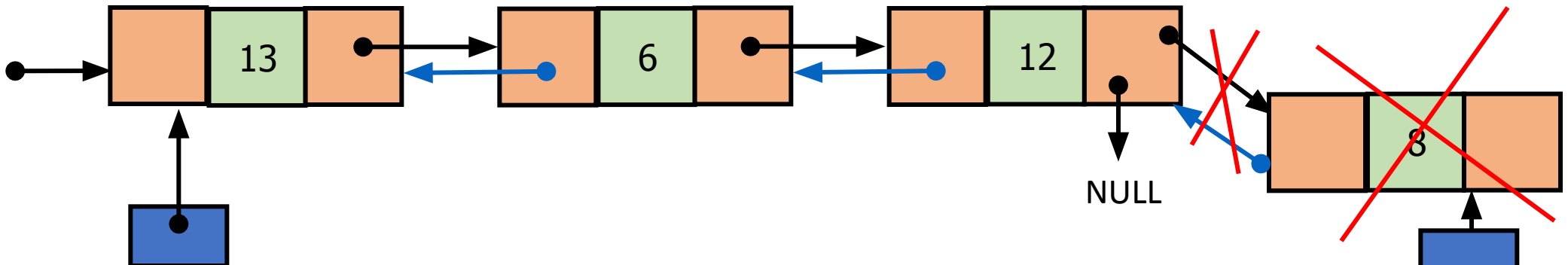
Deletion of a node in the middle

```
temp=head->next;
While(temp->next!=NULL)
{
If(temp->info==data)
{
temp->previous->next=temp->next;
temp->next->previous=temp->previous;
free(temp);
return head;
}
temp=temp->next;
}
```



Deletion of a node at end

```
int deleteFromEnd()
{
       struct node *temp;
        temp = last;
        if(temp->previous==NULL)
        {
                free(temp);
                head=NULL;
                last=NULL;
                        return 0;
        }
        printf("\nData deleted from list is %d \n",last->data);
        last=temp->previous;
      last->next=NULL;
        free(temp);
      return 0;
}
```



Searching a node

```
int searchList(struct Node* head, int search)
{
        struct Node *temp = head;
              int count=0,flag=0,value;
             if(temp == NULL)
                return -1;
        else {
              while(temp->next != head)
         {
                        count++;
                        if(temp->info == search)
                        {
                        flag = 1;
                        count--;
                        break;
                        }
                              temp = temp->next;
         }
```

```
if(temp->info == search)
        {
           count++;
           flag = 1;
        }
   if(flag == 1)
        printf("%d Search node is found \n");
        else
                printf("%d Search node is found \n");
    }
}
```

Advantages and Disadvantages of Doubly Linked List

- **Advantages:**

  1. We can traverse in both directions i.e. from starting to end and as well as from end to starting.
  2. It is easy to reverse the linked list.
  3. If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.

  Disadvantages:

  1. It requires more space per space per node because one extra field is required for pointer to previous node.
  2. Insertion and deletion take more time than linear linked list because more pointer operations are required than linear linked list.

Review Questions

1) How do you calculate the pointer difference in a memory efficient double linked list?

a) head xor tail b) pointer to previous node xor pointer to next node

c) pointer to next node - pointer to previous node

d) pointer to previous node - pointer to next node

2) What is the time complexity of inserting a node in a doubly linked list?

a) O(nlogn) b) O(logn) c) O(n) d) O(1)

3) Consider the following doubly linked list: head-1-2-3-4-5-tail

Node temp = new Node(6, head, head.next()); Node temp1 = new Node(0, tail.getPrev(), tail); head.setNext(temp); temp.getNext().setPrev(temp); tail.setPrev(temp1); temp1.getPrev().setNext(temp1);

a) head-0-1-2-3-4-5-6-tail b) head-1-2-3-4-5-6-tail

c) head-6-1-2-3-4-5-0-tail d) head-0-1-2-3-4-5-tail

Review Questions … Contd.

4) What is a memory efficient double linked list?

a) Each node has only one pointer to traverse the list back and forth.
b) The list has breakpoints for faster traversal
c) An auxiliary singly linked list acts as a helper list to traverse through the doubly linked list
d) None of these

5) Which of the following is false about a doubly linked list?

a) We can navigate in both the directions
b) It requires more space than a singly linked list
c) The insertion and deletion of a node take a bit longer
d) Implementing a doubly linked list is easier than singly linked list

Exercises

- Sort the DLL in ascending order.
- Count the number of nodes in the given DLL.
- Reverse all nodes in doubly linked list
- Swap Kth node from beginning with Kth node from end in a Linked List
- Insert value in sorted way in a sorted doubly linked list
- Remove duplicates from an unsorted doubly linked list
- Count triplets in a sorted doubly linked list whose sum is equal to a given value x
- Check if a doubly linked list of characters is palindrome or not

References

1. Seymour Lipschutz, Data Structures with C, McGraw Hill, 2014
2. Mark Allen Weiss, Data Structures and Algorithm Analysis in C, 2nd ed., Pearson Education, 2015
3. https://cse.iitkgp.ac.in/~palash/Courses/2020PDS/PDS2020.html
4. http://www.btechsmartclass.com/data_structures
5. https://www.geeksforgeeks.org/

THANK YOU