UNIT II

18CSC201J – DATA STRUCTURES AND ALGORITHMS

Topics to be covered:

- ARRAYS
- SPARSE MATRIX
- LINKED LIST –SLL
- DOUBLY LINKED LIST
- CIRCULAR LINKED LIST
- CURSOR BASED IMPLEMENTATION OF LINKED LIST
- APPLICATION OF LINKED LIST – POLYNOMIAL ARITHEMETIC

ARRAYS

Declaration of Arrays

- Syntax:

```
<type> <arrayName>[<array_size>]
```

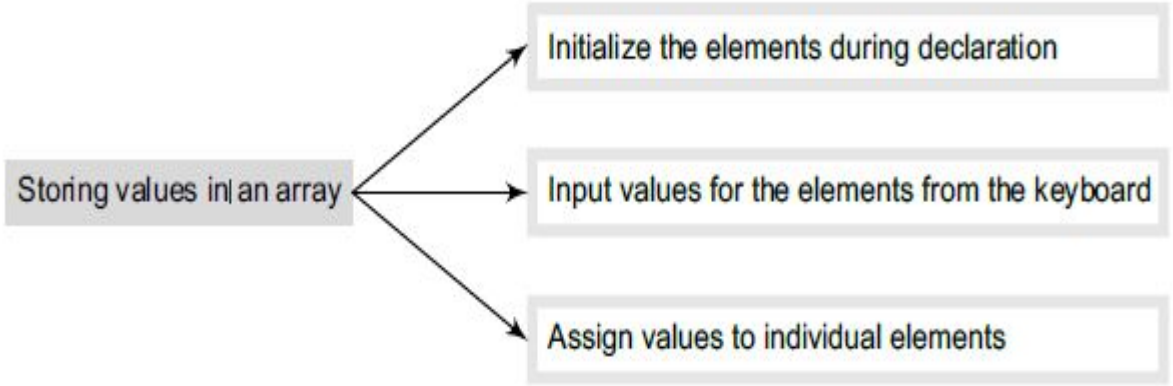
```
Ex. int Ar[10];
```

- The array elements are all values of the type **<type>.**
- The size of the array is indicated by **<array_size>**, the number of elements in the array.
- **<array_size>** must be an **int** constant or a constant expression. Note that an array can have multiple dimensions.

Accessing array elements

To access all the elements, we must use a loop. That is, we can access all the elements of an array by varying the value of the subscript into the array. But note that the subscript must be an integral value or an expression that evaluates to an integral value.

```
// Set each element of the array to -1
int i, marks[10];
for(i=0;i<10;i++)
        marks[i] = -1;
```



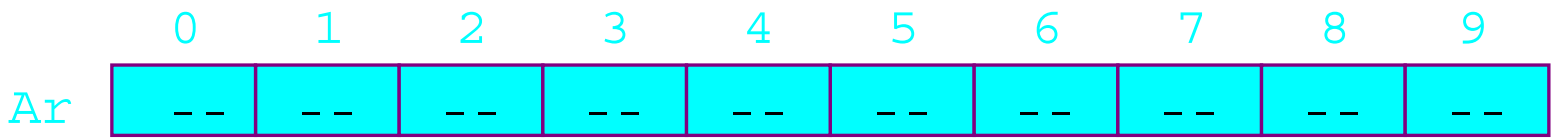
Example of array declaration

```
int Ar[10];
```

- To access an individual element we must apply a subscript to array named **`Ar`**.
  - A subscript is a bracketed expression.
    - The expression in the brackets is known as the index.
  - First element of array has index 0.

    `Ar[0]`
  - Second element of array has index 1, and so on.

    `Ar[1], Ar[2], Ar[3],`…



Array applications

- Given a list of test scores, determine the maximum and minimum scores.
- Read in a list of student names and rearrange them in alphabetical order (sorting).
- Given the height measurements of students in a class, output the names of those students who are taller than average.

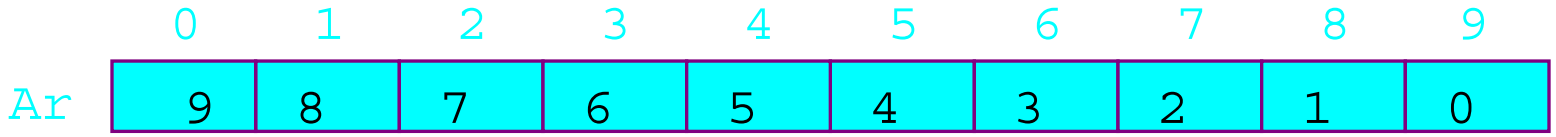
Operations on array elements

- Consider

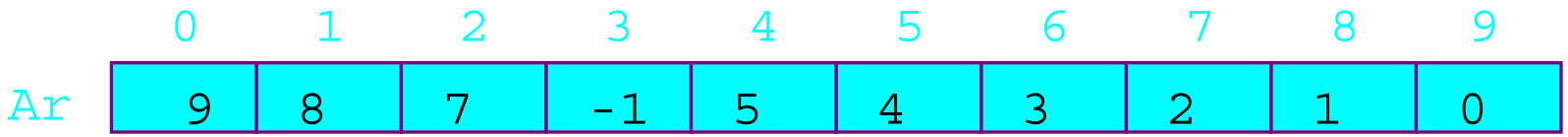
```
int Ar[10], i = 7, j = 2, k = 4;
Ar[0] = 1;
Ar[i] = 5;
Ar[j] = Ar[i] + 3;
Ar[j+1] = Ar[i] + Ar[0];
Ar[Ar[j]] = 12;
cin >> Ar[k];
```

Array Initialization

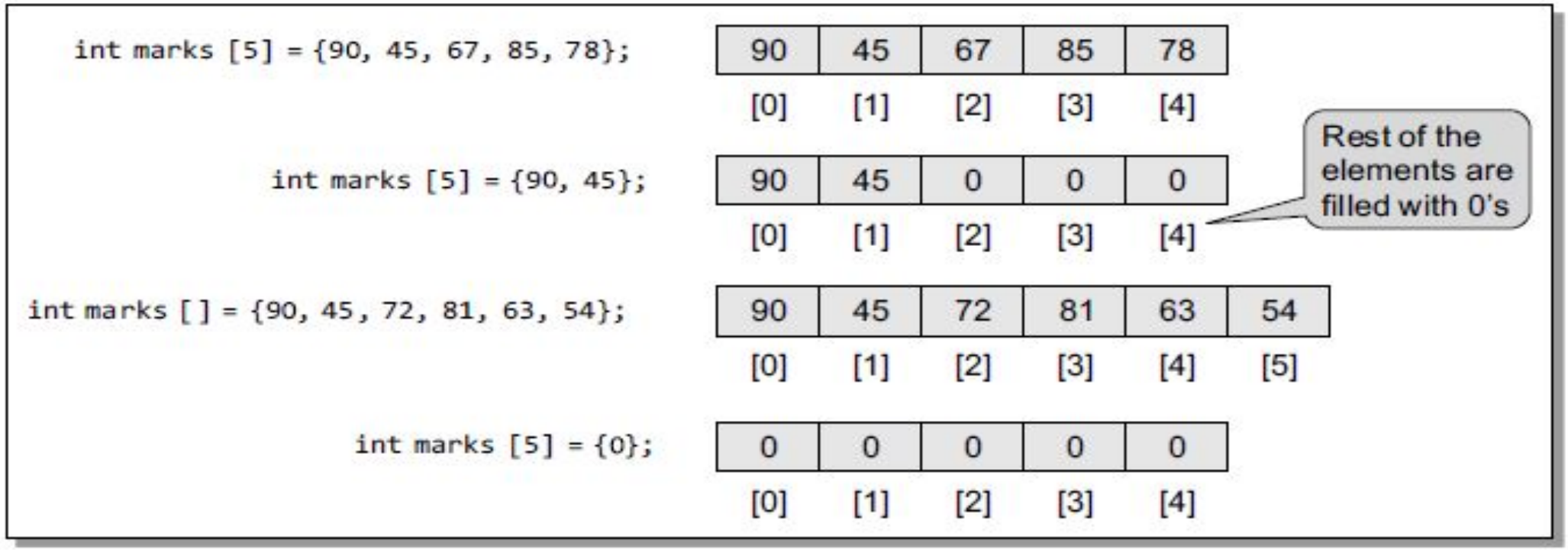
```
int Ar[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```




```
Ar[3] = -1;
```



Array Initialization



OPERATIONS ON ARRAYS

There are a number of operations that can be preformed on arrays. These operations include:

- Traversing an array
- Inserting an element in an array
- Searching an element in an array
- Deleting an element from an array
- Merging two arrays
- Sorting an array in ascending or descending order

Algorithm for array traversal

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:      Apply Process to A[I]
Step 4:      SET  I = I + 1
       [END OF LOOP]
Step 5: EXIT
```

Traversing the data elements of an array A can include printing every element, counting the total number of elements, or performing any process on these elements. Since, array is a linear data structure (because all its elements form a sequence), traversing its elements is very simple and straightforward.

Inserting an Element in an Array

If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple. We just have to add 1 to the `upper_bound` and assign the value. Here, we assume that the memory space allocated for the array is still available.

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

Inserting an Element in an Array

- ***Algorithm to Insert an Element in the Middle of an Array***
- The algorithm INSERT will be declared as INSERT (A, N, POS, VAL). The arguments are
- (a) A, the array in which the element has to be inserted
- (b) N, the number of elements in the array
- (c) POS, the position at which the element has to be inserted
- (d) VAL, the value that has to be inserted

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:         SET A[I + 1] = A[I]
Step 4:         SET I = I - 1
        [END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Deleting an Element from an Array

- If the element has to be deleted from the end of the existing array, then the task of deletion is quite simple. We just have to subtract 1 from the upper_bound.

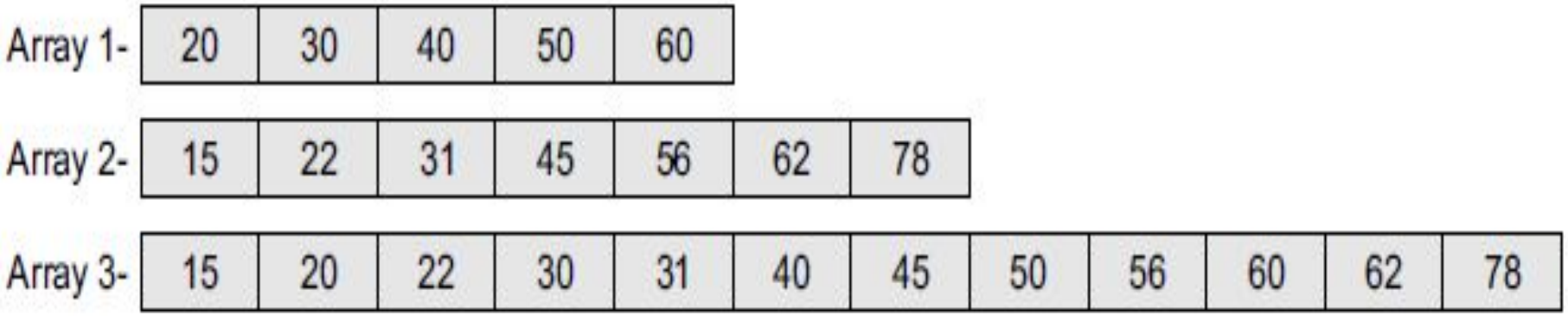
```
Step 1: SET upper_bound = upper_bound - 1
Step 2: EXIT
```

- To delete an element in the middle, we must first find the location from where the element has to be deleted and then move all the elements (having a value greater than that of the element) one position towards left so that the space will be free.

```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3:          SET A[I] = A[I + 1]
Step 4:          SET I = I + 1
        [END OF LOOP]
Step 5: SET N = N - 1
Step 6: EXIT
```

Merging Two Arrays

- Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array. Hence, the merged array contains the contents of the first array followed by the contents of the second array.
- Example:



SPARSE MATRICES

SPARSE MATRICES

- Sparse matrix is a matrix that has large number of elements with a zero value. In order to efficiently utilize the memory, specialized algorithms and data structures that take advantage of the sparse structure should be used.
- If we apply the operations using standard matrix structures and algorithms to sparse matrices, then the execution will slow down and the matrix will consume large amount of memory.
- Sparse data can be easily compressed, which in turn can significantly reduce memory usage.

Types of sparse matrices

- There are two types of sparse matrices.
- In the first type of sparse matrix, all elements above the main diagonal have a zero value. This type of sparse matrix is also called a (*lower*) *triagonal matrix.*
- In an *upper-triangular matrix*, Ai,j = 0 where i > j. An n \ n upper-triangular matrix A has n non-zero elements in the first row, n–1 non-zero elements in the second row and likewise one non-zero element in the *n*th row.

Types of sparse matrices

- There is another variant of a sparse matrix, in which elements with a non-zero value can appear only on the diagonal or immediately above or below the diagonal. This type of matrix is also called a *tri-diagonal matrix*

$$\begin{bmatrix} 1 & & & & \\ 5 & 3 & & & \\ 2 & 7 & -1 & & \\ 3 & 1 & 4 & 2 & \\ -9 & 2 & -8 & 1 & 7 \end{bmatrix}$$

Figure 3.34 Lower-triangular matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ & 3 & 6 & 7 & 8 \\ & & -1 & 9 & 1 \\ & & & 9 & 2 \\ & & & & 7 \end{bmatrix}$$

Figure 3.35 Upper-triangular matrix

$$\begin{bmatrix} 4 & 1 & & & & \\ 5 & 1 & 2 & & & \\ & 9 & 3 & 1 & & \\ & & 4 & 2 & 2 & \\ & & & 5 & 1 & 9 \\ & & & & 8 & 7 \end{bmatrix}$$

Figure 3.36 Tri-diagonal matrix

LINKED LIST

Comparison of Array and Linked list

Arrays	Linked list
Inserting/deleting an element at the end.	Inserting/deleting an element at any place
Randomly accessing any element.	Applications where sequential access is required.
Searching the list for a particular value.	It can be applied where the number of elements cannot be predicted beforehand.
Static memory allocation	Dynamic memory allocation
Data is homogeneous	Data can be heterogeneous

Abstract Data Type (ADT)

- What is an abstract data type?
  - It is a data type defined by the user.
  - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
  - Because details of the implementation are hidden.
  - When you do some operation on the list, say insert an element, you just call a function.
  - Details of how the list is implemented or how the insert function is written is no longer required.

Linked List

- A linked list, in simple terms, is a linear collection of data elements. These data elements are called *nodes*.
- Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.
- Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a *self-referential data type.*

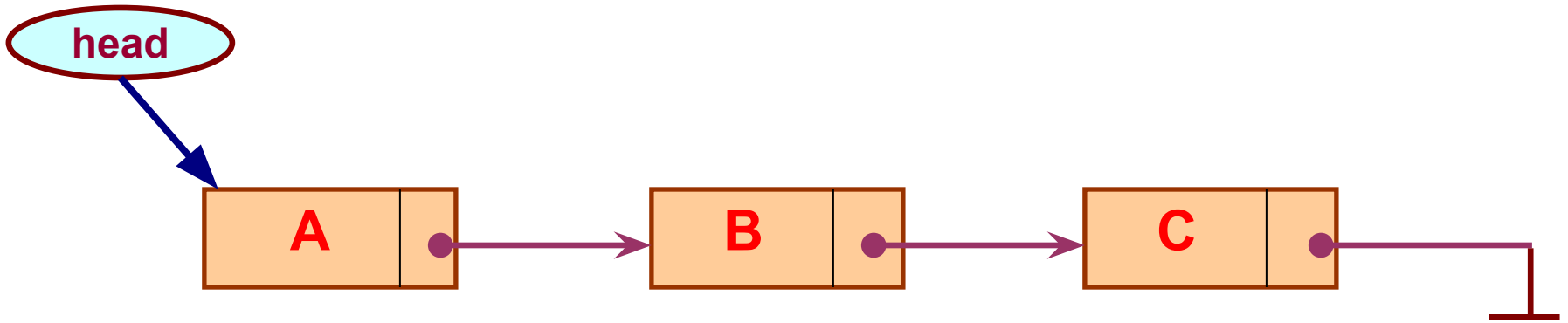
Types of linked list

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

1. Linear singly-linked list (or simply linear list)
2. Circular Linked list
3. Doubly linked list
4. Multi linked list

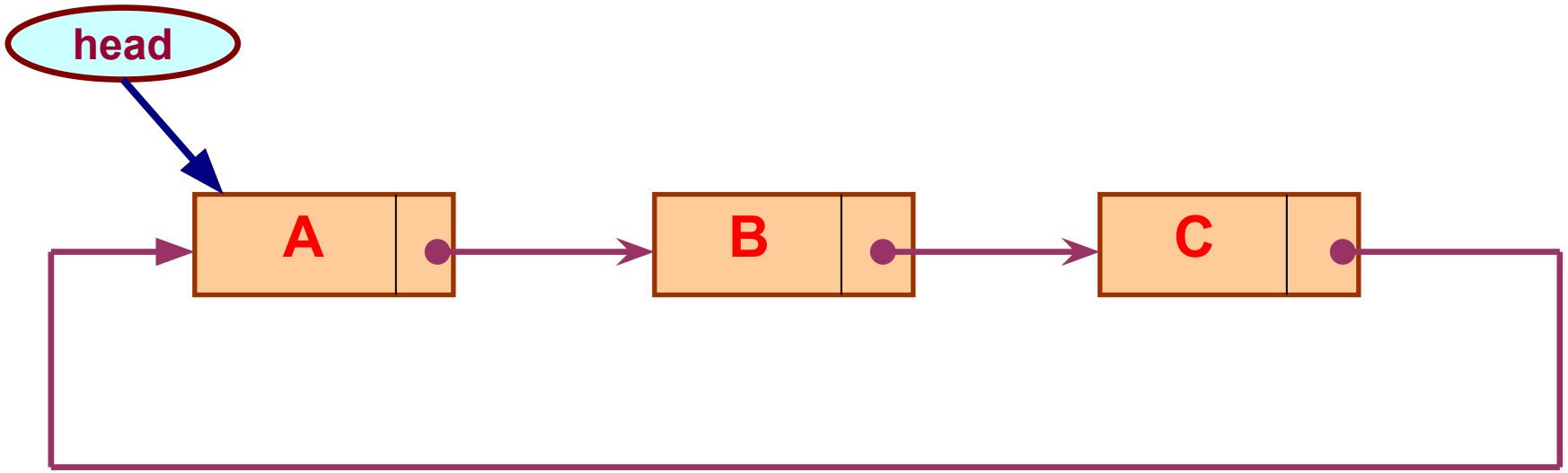
Singly linked list

- A singly linked list is the simplest type of linked list in which every node contains some data and
- a pointer to the next node of the same data type. By saying that the node contains a pointer to the
- next node, we mean that the node stores the address of the next node in sequence.



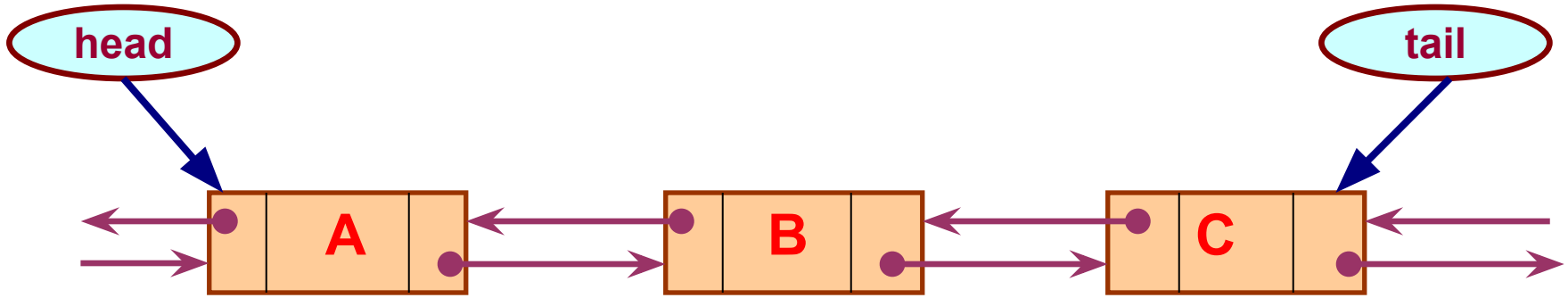
Circular Linked list

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



Doubly linked list

- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two pointers are maintained to keep track of the list, *head* and *tail*.



Singly Linked List (SLL)

Singly Linked List (SLL)

- In C, we can create a node using the following code:

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

Operations on SLL:

1. Traversal
2. Searching
3. Insertion
4. Deletion

Traversing a Linked List

- Traversing a linked list means accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:            Apply Process to PTR->DATA
Step 4:            SET PTR = PTR->NEXT
       [END OF LOOP]
Step 5: EXIT
```

Searching for a Value in a Linked List

Searching a linked list means to find a particular element in the linked list. As already discussed, a linked list consists of nodes which are divided into two parts, the information part and the next part. So searching means finding whether a given value is present in the information part of the node or not.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:      IF VAL = PTR -> DATA
                   SET POS = PTR
                   Go To Step 5
             ELSE
                   SET PTR = PTR -> NEXT
             [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Inserting a New Node in a Linked List

- Case 1: The new node is inserted at the beginning.
- Case 2: The new node is inserted at the end.
- Case 3: The new node is inserted after a given node.
- Case 4: The new node is inserted before a given node.

OVERFLOW:

Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

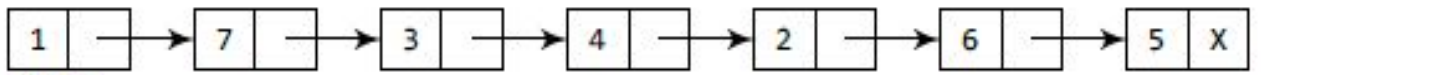
Inserting a Node at the Beginning of a Linked List

Algorithm:

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

Inserting a Node at the Beginning of a Linked List

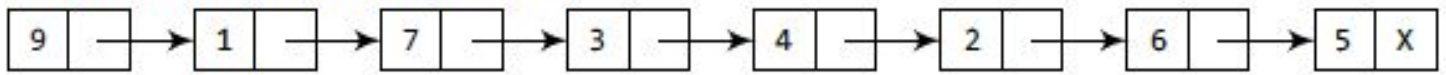
Example:



START

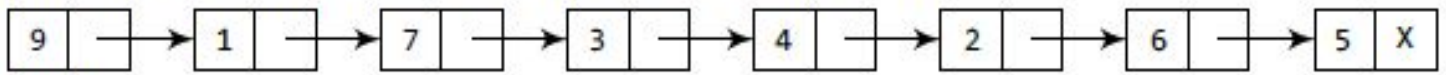
Allocate memory for the new node and initialize its DATA part to 9.

Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.



START

Now make START to point to the first node of the list.



START

Inserting a Node at the End of a Linked List

Algorithm:

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Inserting a Node After a Given Node in a Linked List

Algorithm:

```
Step 1:  IF AVAIL = NULL
             Write OVERFLOW
             Go to Step 12
         [END OF IF]
Step 2:  SET NEW_NODE = AVAIL
Step 3:  SET AVAIL = AVAIL -> NEXT
Step 4:  SET NEW_NODE -> DATA = VAL
Step 5:  SET PTR = START
Step 6:  SET PREPTR = PTR
Step 7:  Repeat Steps 8 and 9 while PREPTR -> DATA
         != NUM
Step 8:      SET PREPTR = PTR
Step 9:      SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Inserting a Node Before a Given Node in a Linked List

- Algorithm:

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR->DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR->NEXT
        [END OF LOOP]
Step 10: PREPTR->NEXT = NEW_NODE
Step 11: SET NEW_NODE->NEXT = PTR
Step 12: EXIT
```

Deleting a Node from a Linked List

Consider three cases and how deletion is done in each case.

- Case 1: The first node is deleted.
- Case 2: The last node is deleted.
- Case 3: The node after a given node is deleted.

Deleting the First Node from a Linked List

Algorithm:

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Deleting the Last Node from a Linked List

- Algorithm:

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:     SET PREPTR = PTR
Step 5:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Deleting the Node After a Given Node in a Linked List

- Algorithm:

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:     SET PREPTR = PTR
Step 6:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

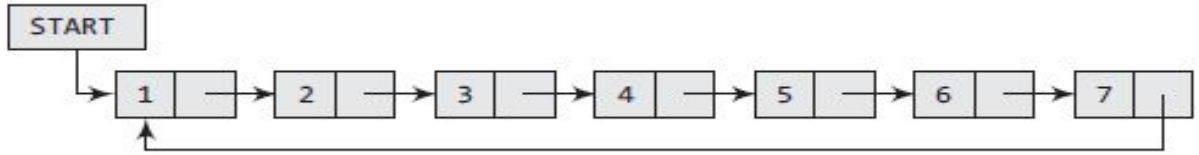
CIRCULAR LINKED LIST

CIRCULAR LINKED LIST

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list.

While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started.

Thus, a circular linked list has no beginning and no ending



Operations on a Circular Linked List

1. Insertion
2. Deletion
3. Traversal
4. Search

Inserting a New Node in a Circular Linked List

- Case 1: The new node is inserted at the beginning of the circular linked list.

Algorithm:

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:      PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Inserting a New Node in a Circular Linked List

- Case 2: The new node is inserted at the end of the circular linked list.

Algorithm:

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Deleting a Node from a Circular Linked List

Case 1: The first node is deleted.

Algorithm:

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:        SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step 7: SET START = PTR -> NEXT
Step 8: EXIT
```

Deleting a Node from a Circular Linked List

- Case 2: The last node is deleted.

```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:          SET PREPTR = PTR
Step 5:          SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

DOUBLY LINKED LISTS

DOUBLY LINKED LISTS

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node.

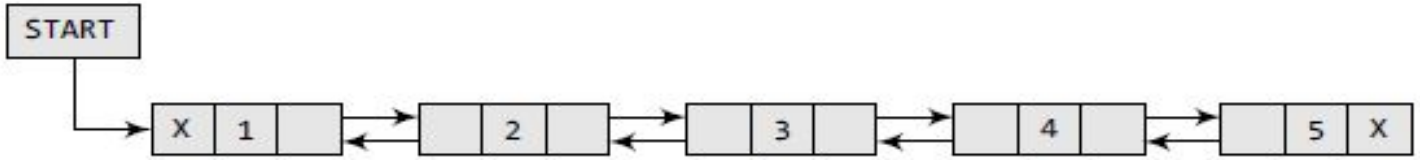


Figure 6.37 Doubly linked list

In C, the structure of a doubly linked list can be given as,

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

Inserting a New Node in a Doubly Linked List

Insertion is done in following cases:

Case 1: The new node is inserted at the beginning

- Case 2: The new node is inserted at the end.
- Case 3: The new node is inserted after a given node.
- Case 4: The new node is inserted before a given node.

Inserting a Node at the Beginning of a Doubly Linked List

Algorithm:

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Inserting a Node at the End of a Doubly Linked List

Algorithm:

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:      SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

Inserting a Node After a Given Node in a Doubly Linked List

Algorithm:

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

Deleting a Node from a Linked List

Deletion can be done in following ways:

- Case 1: The first node is deleted.
- Case 2: The last node is deleted.
- Case 3: The node after a given node is deleted.
- Case 4: The node before a given node is deleted..

Deleting the First Node from a Doubly Linked List

- Algorithm:

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 6
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

Deleting the Last Node from a Doubly Linked List

- Algorithm:

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 7
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

Deleting the Node After a Given Node in a Doubly Linked List

- Algorithm:

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> NEXT
Step 6: SET PTR -> NEXT = TEMP -> NEXT
Step 7: SET TEMP -> NEXT -> PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

Deleting the Node Before a Given Node in a Doubly Linked List

- Algorithm:

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> PREV
Step 6: SET TEMP -> PREV -> NEXT = PTR
Step 7: SET PTR -> PREV = TEMP -> PREV
Step 8: FREE TEMP
Step 9: EXIT
```

Cursor based implementation of a Linked List

Cursor based implementation of a Linked List

Many languages, such as BASIC and FORTRAN, do not support pointers. If linked lists are required and pointers are not available, then an alternate implementation must be used. The alternate method we will describe is known as a **cursor implementation**.

The two important items present in a pointer implementation of linked lists are

1. The data is stored in a collection of structures. Each structure contains the data and a pointer to the next structure.

2. A new structure can be obtained from the system's global memory by a call to malloc and released by a call to free.

Our cursor implementation must be able to simulate this. The logical way to satisfy condition 1 is to have a global array of structures. For any cell in the array, its array index can be used in place of an address.

We must now simulate condition 2 by allowing the equivalent of malloc and free for cells in the CURSOR_SPACE array. To do this, we will keep a list (the freelist) of cells that are not in any list. The list will use cell 0 as a header.

Cursor based implementation of a Linked List

- To perform an malloc, the first element (after the header) is removed from the freelist.

```
typedef unsigned int node_ptr;

struct node
{
   element_type element;
   node_ptr next;
};

typedef node_ptr LIST;
typedef node_ptr position;
struct node CURSOR_SPACE[ SPACE_SIZE ];
```

Cursor based implementation of a Linked List

- An initialized CURSOR_SPACE looks as below:

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 based implementation of a Linked List

- Given this, the cursor implementation of linked lists is straightforward. For consistency, we will implement our lists with a header node. The routine for allocation and free is given below:

```
position cursor_alloc( void )
{
  position p;
  p = CURSOR_SPACE[0].next;
  CURSOR_SPACE[0].next = CURSOR_SPACE[p].next;
  return p;
}

void cursor_free( position p)
{
  CURSOR_SPACE[p].next = CURSOR_SPACE[0].next;
  CURSOR_SPACE[0].next = p;
}
```

Cursor based implementation of a Linked List

- Example of a cursor implementation of linked lists contains two lists.

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

Cursor based implementation of a Linked List

- Function to test whether a linked list is empty--cursor implementation

```
int is_empty( LIST L ) /* using a header node */
{
  return( CURSOR_SPACE[L].next == 0
}
```

- Function to test whether p is last in a linked list--cursor implementation

```
int is_last( position p, LIST L) /* using a header node */
{
  return( CURSOR_SPACE[p].next == 0
}
```

Cursor based implementation of a Linked List

- Insertion routine for linked lists--cursor implementation

```
/* Insert (after legal position p); */
/* header implementation assumed */

void insert( element_type x, LIST L, position p )
{
  position tmp_cell;
  /*1*/ tmp_cell = cursor_alloc( )
  /*2*/ if( tmp_cell ==0 )
  /*3*/ fatal_error("Out of space!!!");
  else
  {
    /*4*/ CURSOR_SPACE[tmp_cell].element = x;
    /*5*/ CURSOR_SPACE[tmp_cell].next = CURSOR_SPACE[p].next;
    /*6*/ CURSOR_SPACE[p].next = tmp_cell;
  }
}
```

Cursor based implementation of a Linked List

- Deletion routine for linked lists--cursor implementation

```
void insert( element_type x, LIST L, position p )
{
  position tmp_cell;
  /*1*/ tmp_cell = cursor_alloc( )
  /*2*/ if( tmp_cell ==0 )
  /*3*/ fatal_error("Out of space!!!");
  else
  {
    /*4*/ CURSOR_SPACE[tmp_cell].element = x;
    /*5*/ CURSOR_SPACE[tmp_cell].next = CURSOR_SPACE[p].next;
    /*6*/ CURSOR_SPACE[p].next = tmp_cell;
  }
}
```

Cursor based implementation of a Linked List

- Find routine--cursor implementation

```
position find( element_type x, LIST L) /* using a header node */
{
  position p;
  /*1*/ p = CURSOR_SPACE[L].next;
  /*2*/ while( p && CURSOR_SPACE[p].element != x )
  /*3*/ p = CURSOR_SPACE[p].next;
  /*4*/ return p;
}
```

- The freelist represents an interesting data structure in its own right. The cell that is removed from the freelist is the one that was most recently placed there by virtue of free.
- Thus, the last cell placed on the **freelist** is the first cell taken off. The data structure that also has this property is known as a stack.

APPLICATIONS OF LINKED LIST

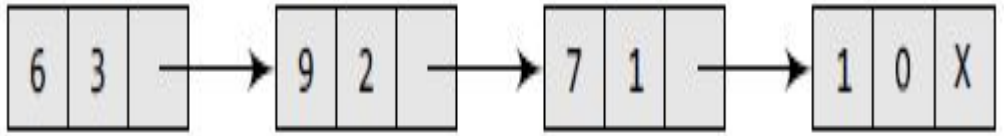
Polynomial arithmetic

APPLICATIONS OF LINKED LISTS

- Linked lists can be used to represent polynomials and the different operations that can be performed on them. In this section, we will see how polynomials are represented in the memory using linked lists.
- **Polynomial Representation:**
- Let us see how a polynomial is represented in the memory using a linked list. Consider a polynomial 6x^3 + 9x^2 + 7x + 1.
- Every individual term in a polynomial consists of two parts, a coefficient and a power.
- Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.

- Every term of a polynomial can be represented as a node of the linked list.
- The following polynomial equation can be represented in the form of linked list as follows:

$$6x^3 + 9x^2 + 7x + 1.$$



- operations on polynomials can also be performed using linked list.
- Structure of a node :

```
struct node
{
        int num;
        int coeff;
        struct node *next;
};
struct node *start1 = NULL;
struct node *start2 = NULL;
struct node *start3 = NULL;
struct node *start4 = NULL;
struct node *last3 = NULL;
```

Create a polynomial linked list

```
struct node *create_poly(struct node *start)
{
        struct node *new_node, *ptr;
        int n, c;
        printf("\n Enter the number : ");
        scanf("%d", &n);
        printf("\t Enter its coefficient : ");
        scanf("%d", &c);
        while(n != -1)
        {
                if(start==NULL)
                {
                        new_node = (struct node *)malloc(sizeof(struct node));
                        new_node -> num = n;
                        new_node -> coeff = c;
                        new_node -> next = NULL;
                        start = new_node;
                }
                else
                {
                        ptr = start;
                        while(ptr -> next != NULL)
                                ptr = ptr -> next;
                        new_node = (struct node *)malloc(sizeof(struct node));
                        new_node -> num = n;
                        new_node -> coeff = c;
                        new_node -> next = NULL;
                        ptr -> next = new_node;
                }
                printf("\n Enter the number : ");
                scanf("%d", &n);
                if(n == -1)
                        break;
                printf("\t Enter its coefficient : ");
                scanf("%d", &c);
        }
        return start;
}
```

Add two polynomials using linked list

```
ptr1 = start1, ptr2 = start2;
while(ptr1 != NULL && ptr2 != NULL)
{
        if(ptr1 -> coeff == ptr2 -> coeff)
        {
                sum_num = ptr1 -> num + ptr2 -> num;
                start3 = add_node(start3, sum_num, ptr1 -> coeff);
                ptr1 = ptr1 -> next;
                ptr2 = ptr2 -> next;
        }
        else if(ptr1 -> coeff > ptr2 -> coeff)
        {
                start3 = add_node(start3, ptr1 -> num, ptr1 -> coeff);
                ptr1 = ptr1 -> next;
        }
        else if(ptr1 -> coeff < ptr2 -> coeff)
        {
                start3 = add_node(start3, ptr2 -> num, ptr2 -> coeff);
                ptr2 = ptr2 -> next;
        }
}
if(ptr1 == NULL)
{
        while(ptr2 != NULL)
        {
                start3 = add_node(start3, ptr2 -> num, ptr2 -> coeff);
                ptr2 = ptr2 -> next;
        }
}
if(ptr2 == NULL)
{
        while(ptr1 != NULL)
        {
                start3 = add_node(start3, ptr1 -> num, ptr1 -> coeff);
                ptr1 = ptr1 -> next;
        }
}
return start3;
```

Difference of two polynomials using Linked list

```
ptr1 = start1, ptr2 = start2;
do
{
if(ptr1 -> coeff == ptr2 -> coeff)
{
sub_num = ptr1 -> num – ptr2 -> num;
start4 = add_node(start4, sub_num, ptr1 -> coeff);
ptr1 = ptr1 -> next; ptr2 = ptr2 -> next;
}
else if(ptr1 -> coeff > ptr2 -> coeff)
{
start4 = add_node(start4, ptr1 -> num, ptr1 -> coeff);
ptr1 = ptr1 -> next;
}
else if(ptr1 -> coeff < ptr2 -> coeff)
{
start4 = add_node(start4, ptr2 -> num, ptr2 -> coeff);
ptr2 = ptr2 -> next;
}
}while(ptr1 != NULL || ptr2 != NULL);
if(ptr1 == NULL)
{
while(ptr2 != NULL)
{
start4 = add_node(start4, ptr2 -> num, ptr2 -> coeff);
ptr2 = ptr2 -> next;
}
}
if(ptr2 == NULL)
{
while(ptr1 != NULL)
{
start4 = add_node(start4, ptr1 -> num, ptr1 -> coeff);
ptr1 = ptr1 -> next;
}
}
```

Displaying a polynomial using linked list

```
struct node *display_poly(struct node *start)
{
        struct node *ptr;
        ptr = start;
        while(ptr != NULL)
        {
                printf("\n%d x %d\t", ptr->num, ptr->coeff);
                ptr = ptr->next;
        }
        return start;
}
```

Add a node to the existing polynomial

```
struct node *add_node(struct node *start, int n, int c)
{
        struct node *ptr, *new_node;
        if(start == NULL)
        {
                new_node = (struct node *)malloc(sizeof(struct node));
                new_node->num = n;
                new_node->coeff = c;
                new_node->next = NULL;
                start = new_node;
        }
        else
        {
                ptr = start;
                while(ptr->next != NULL)
                        ptr = ptr->next;
                new_node = (struct node *)malloc(sizeof(struct node));
                new_node->num = n;
                new_node->coeff = c;
                new_node->next = NULL;
                ptr->next = new_node;
        }
        return start;
}
```