UNIT-2

SYLLABUS

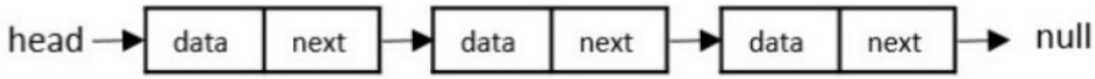
Linked List: Representation and Implementation of Singly Linked Lists, Two-way Header List, Traversing and Searching of Linked List, Overflow and Underflow, Insertion and deletion to/from Linked Lists, Insertion and deletion Algorithms, doubly linked list, Linked List in Array, Polynomial representation, and addition.

Linked List:

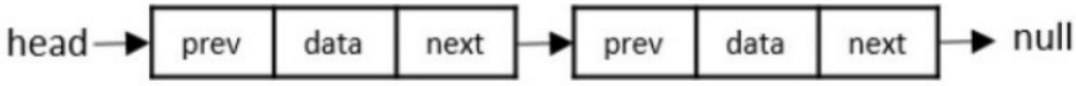
- A linked list is a linear data structure that is used to maintain a list-like structure in the computer memory.
- It is a group of nodes that are not stored at contiguous locations.
- Each node of the list is linked to its adjacent node with the help of pointers.
- Each node is divided into two parts:
  1. The first part contains the information of the element,
  2. The second part, called the link field or next pointer field, contains the address of the next node in the list.

Types of Linked List:

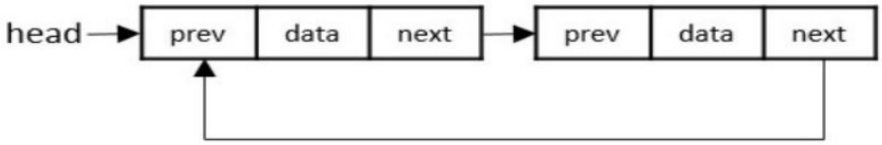
1) **Singly Linked List:**
   - Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list.
   - Traversals can be done in one direction only as there is only a single link between two nodes of the same list.



2) **Doubly Linked List/Two Ways Header List:**
   - Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list.
   - The list is traversed twice as the nodes in the list are connected to each other from both sides.



3) **Circular Linked List:**
   - Circular linked lists can exist in both singly linked list and doubly linked list.
   - The last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.



Array	Linked list
An array is a collection of elements of a similar data type.	A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address.
Array elements store in a contiguous memory location.	Linked list elements can be stored anywhere in the memory or randomly stored.
Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at the run time according to our requirements.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list takes less time while performing any operation like insertion, deletion, etc.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time.	In the case of a linked list, memory is allocated at run time.
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement.

SINGLY LINKED LIST

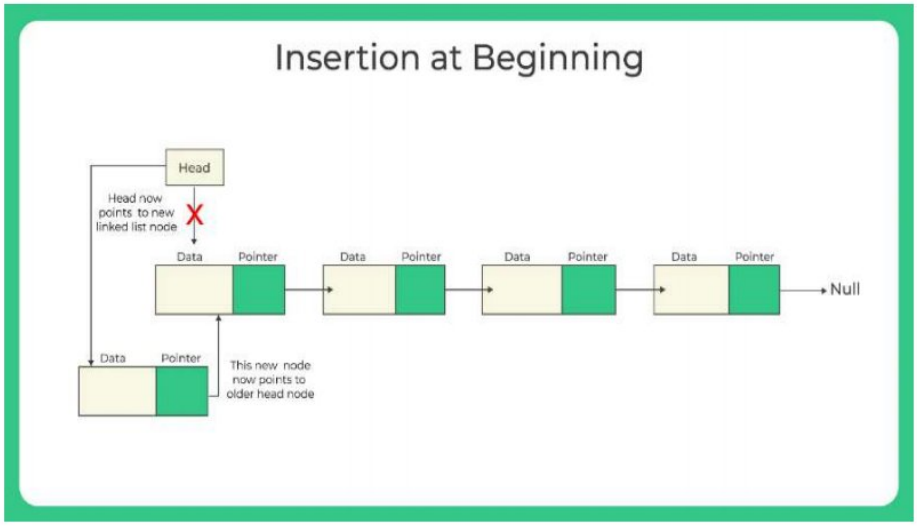
Basic Operations in the Singly Linked Lists:

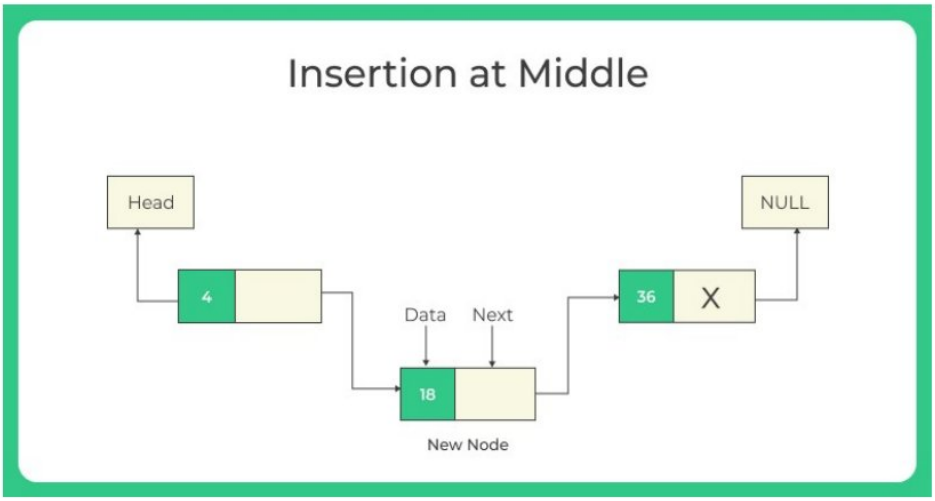
1) **Insertion:** Adds an element at the beginning of the list.
2) **Deletion:** Deletes an element at the beginning of the list.
3) **Traversal:** Visiting each node of the list
4) **Search:** Searches an element using the given key.

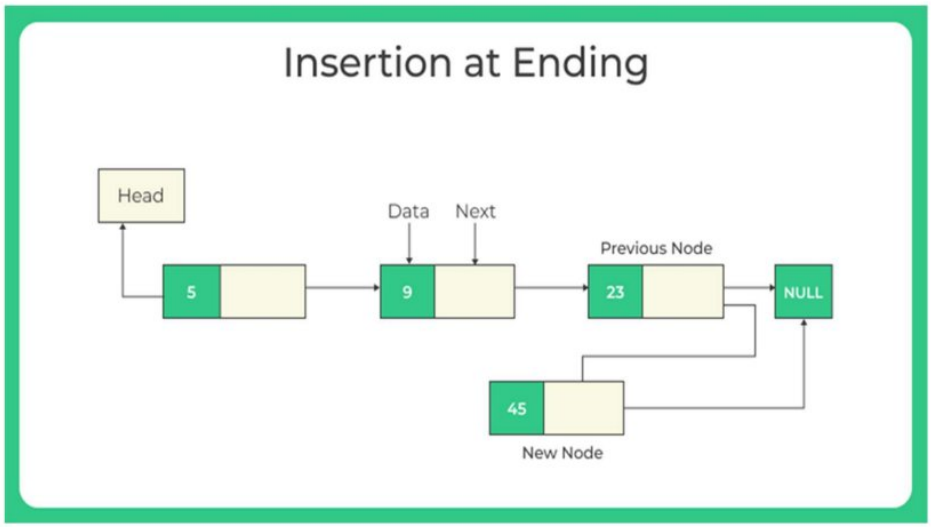
Insertion Operation:

There are three possible positions where we can enter a new node in singly linked list:

1) Insertion at Beginning:
2) Insertion after N^{th} Position:
3) Insertion at End





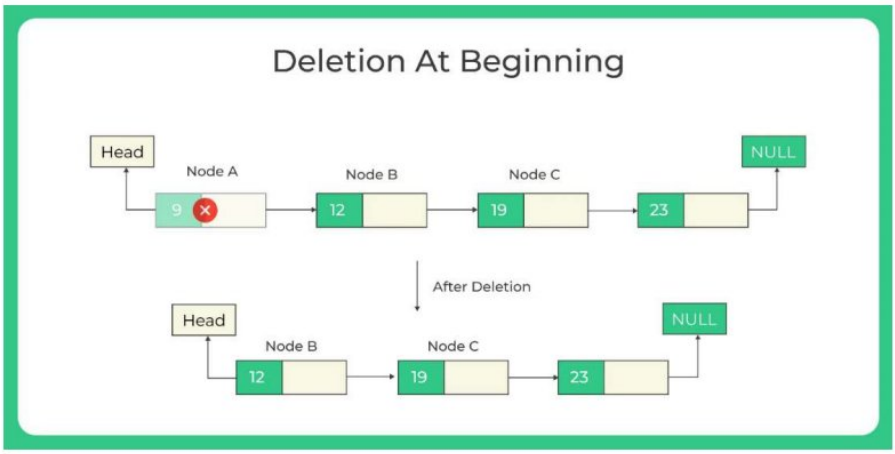


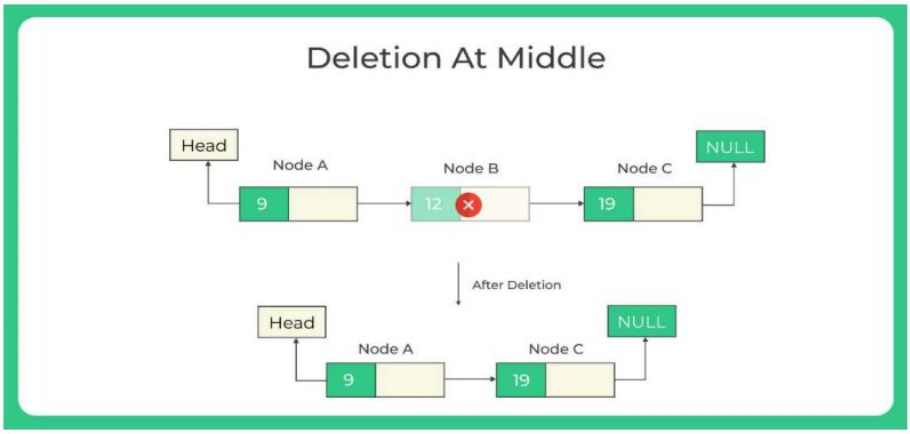
INSERTION ALGORITHMS			
Steps	**AT BEGINNING**	**AT Nth POSITION**	**AT END**
1	**IF ptr = null** write OVERFLOW go to step 6 [end of if]	**IF ptr = null** write OVERFLOW go to step 10 [end of if]	**IF ptr = null** write OVERFLOW go to step 9 [end of if]
2	Set new_node = ptr	Set new_node = ptr	Set new_node = ptr
3	Set ptr → data = val	Set ptr → data = val	Set ptr → data = val
4	Set ptr → next = head	**IF head = = null** Set ptr → next = head Set head = ptr go to step 10 [end of if]	**IF head = null** Set ptr → next = null Set head =ptr go to step 9 [end of if]
5	Set head = ptr	Set temp = head	Set temp = head
6	Exit	**For (i=0; i < loc; i++)** Set temp= temp→next	Repeat Step 6 **while** **temp → next != null** Set temp = temp → next [end of loop]
7		**IF temp = null** Write NO INSERT go to step 10 [end of if and for]	Set temp → next=ptr
8		Set ptr→next=temp→next	Set ptr → next = null
9		Set temp →next = ptr	Exit
10		Exit	

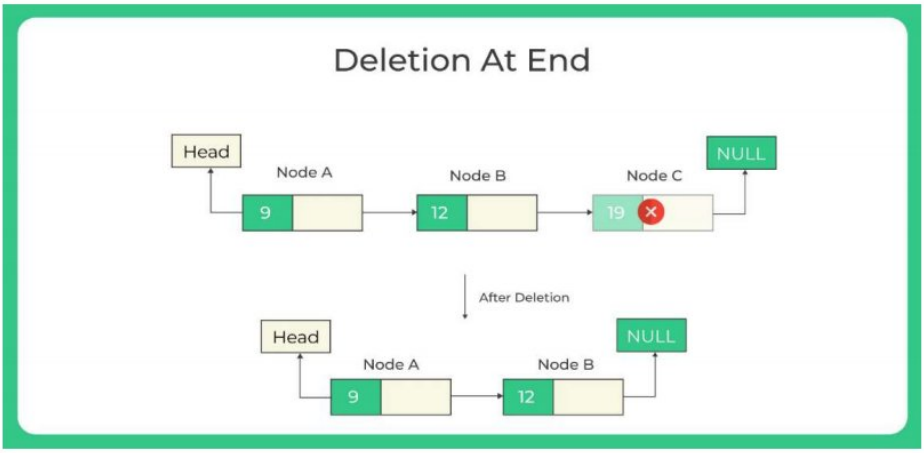
Deletion Operation:

There are three possible positions from where we can delete a node in singly linked list:

1) Deletion at Beginning:
2) Deletion after N^{th} Position:
3) Deletion at End:







DELETION ALGORITHMS			
Steps	**AT BEGINNING**	**AT Nth POSITION**	**AT END**
1	**IF head = null** write UNDERFLOW go to step 5 [end of if]	**IF head = null** write UNDERFLOW go to step 7 [end of if]	**IF head = null** write UNDERFLOW go to step 7 [end of if]
2	Set ptr = head	Set ptr = head	Set ptr = head
3	Set head = ptr → next	**For (i=0; i < loc; i++)** ptr1 = ptr ptr = ptr → next	**IF head→next==null** Head = null Free head go to step 7 [end of if]
4	Set free ptr	**IF ptr = = null** Write NO LOCATION FOUND [end of if and for]	Repeat Step 4 **while** **ptr → next != null** Set ptr1 = ptr Set ptr = ptr → next [end of loop]
5	Exit	Set ptr1→next=ptr→next	Set ptr1→ next = null
6		Set free ptr	Set free ptr
7		Exit	Exit

Traversing:

- Traversing means visiting each node of the list once in order to perform some operation on that. This will be done by using the following algorithm.

Searching:

- Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element, then the location of the element is returned from the function.

Steps	Traversal Algorithm	Searching Algorithm
1	Set ptr = head	Set ptr = head
2	**IF ptr == NULL** write EMPTY LIST go to step 4 [end of if]	Set i=0
3	Repeat Step 3 **while ptr!=NULL** Set ptr→ data Set ptr = ptr → next [end of loop]	**IF ptr = NULL** write EMPTY LIST go to step 8 [end of if]
4	Exit	Repeat Step 5 to 7 **while ptr!=NULL**
5		**IF ptr→ data = value** write i +1 [end of if]
6		i= i +1
7		ptr = ptr → next; [end of loop]
8		Exit

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* head = NULL;
// Function to insert a new node at the beginning of the linked list
void insertAtBeginning(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}
// Function to insert a new node at the end of the linked list
void insertAtEnd(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;   }
    temp->next = newNode;    }
// Function to insert a new node at a specific position in the linked list
void insertAtPosition(int value, int position) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
```

```
    if (position == 1) {
        newNode->next = head;
        head = newNode;
        return;
    }
    struct Node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Invalid position\n");
        return;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}
// Function to delete a node from the beginning of the linked list
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty\n");
        return;    }
    struct Node* temp = head;
    head = head->next;
    free(temp);
}
// Function to delete a node from the end of the linked list
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    if (head->next == NULL) {
```

```
        free(head);
        head = NULL;
        return;
    }
    struct Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}
// Function to delete a node from a specific position in the linked list
void deleteFromPosition(int position) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* temp = head;
    if (position == 1) {
        head = head->next;
        free(temp);
        return;
    }
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL || temp->next == NULL) {
        printf("Invalid position\n");
        return;
    }
    struct Node* deletedNode = temp->next;
    temp->next = temp->next->next;
```

```
    free(deletedNode);
}
// Function to search for a value in the linked list
void search(int value) {
    struct Node* temp = head;
    int position = 1;
    while (temp != NULL) {
        if (temp->data == value) {
            printf("Value %d found at position %d\n", value, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("Value %d not found in the list\n", value);
}
// Function to display the linked list
void display() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    int choice, value, position;
    do {
        // Menu for user choices
        printf("\nLinked List Operations:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
```

```
printf("3. Insert at Position\n");
printf("4. Delete from Beginning\n");
printf("5. Delete from End\n");
printf("6. Delete from Position\n");
printf("7. Search\n");
printf("8. Display\n");
printf("0. Exit\n");
// User input for choice
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
   case 1:
      printf("Enter value to insert at the beginning: ");
      scanf("%d", &value);
      insertAtBeginning(value);
      break;
   case 2:
      printf("Enter value to insert at the end: ");
      scanf("%d", &value);
      insertAtEnd(value);
      break;
   case 3:
      printf("Enter value to insert: ");
      scanf("%d", &value);
      printf("Enter position to insert at: ");
      scanf("%d", &position);
      insertAtPosition(value, position);
      break;
   case 4:
      deleteFromBeginning();
      break;
   case 5:
```

```
            deleteFromEnd();
            break;
        case 6:
            printf("Enter position to delete from: ");
            scanf("%d", &position);
            deleteFromPosition(position);
            break;
        case 7:
            printf("Enter value to search: ");
            scanf("%d", &value);
            search(value);
            break;
        case 8:
            display();
            break;
        case 0:
            printf("Exiting program.\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }
  } while (choice != 0);
  return 0;
}
```

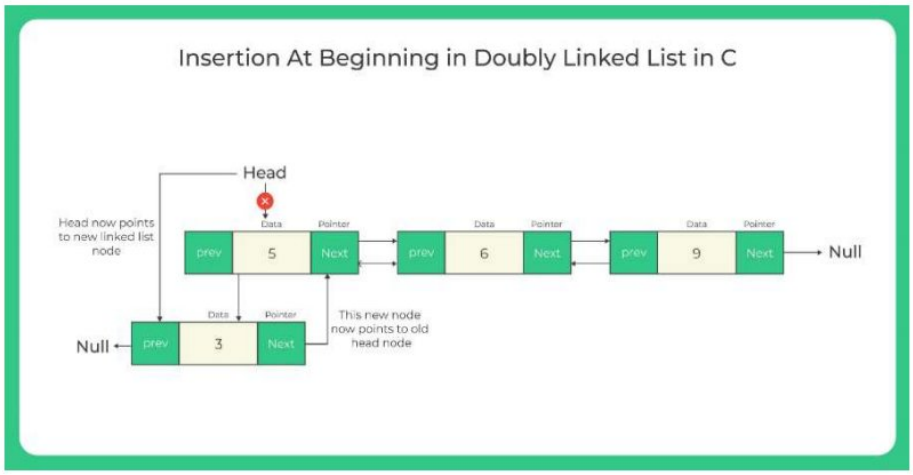
DOUBLY LINKED LIST

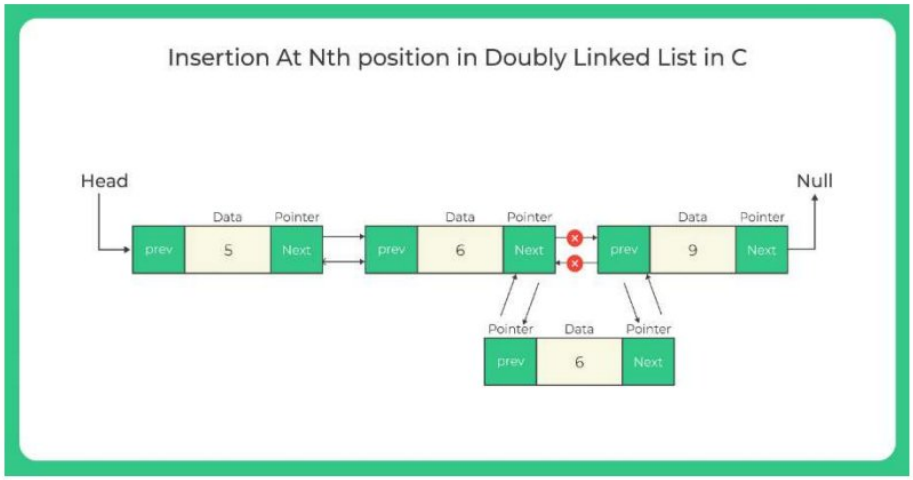
Basic Operations in the Doubly Linked Lists:

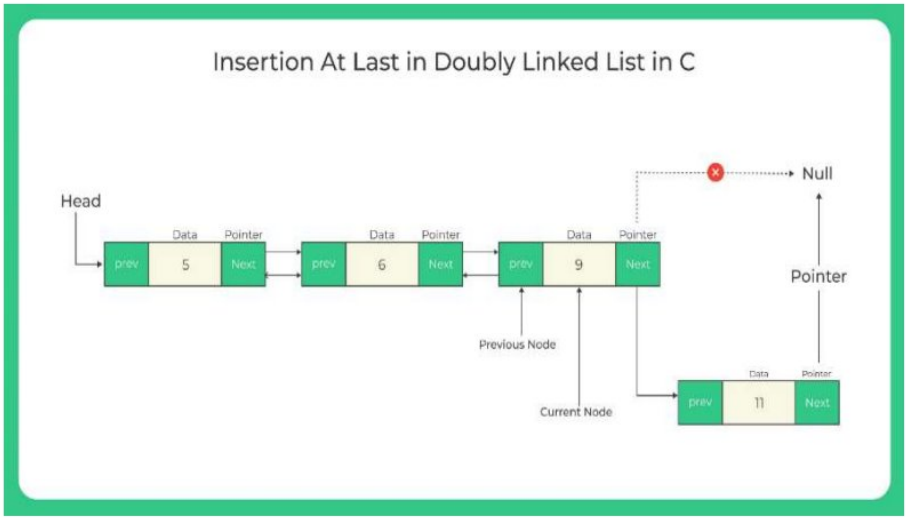
1) **Insertion:** Adds an element at the beginning of the list.
2) **Deletion:** Deletes an element at the beginning of the list.
3) **Traversal:** Visiting each node of the list
4) **Search:** Searches an element using the given key.

Insertion Operation: There are three possible positions where we can enter a new node in doubly linked list:

Insertion at Beginning, After N^{th} Position, and Insertion at End





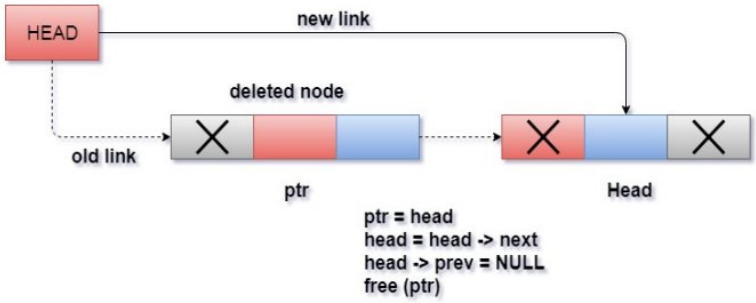


INSERTION ALGORITHMS			
Steps	**AT BEGINNING**	**AT Nth POSITION**	**AT END**
1	**IF ptr = null** write OVERFLOW go to step 9 [end of if]	**IF ptr = null** write OVERFLOW go to step 10 [end of if]	**IF ptr = null** write OVERFLOW go to step 10 [end of if]
2	Set new_node = ptr	Set new_node = ptr	Set new_node = ptr
3	Set ptr → data = val	Set ptr → data = val	Set ptr → data = val
4	**IF head = null** Set ptr → next = null Set ptr → prev = null go to step 8 [end of if]	**IF head = null** Set ptr → next = null Set ptr → prev = null Set head =ptr go to step 10 [end of if]	**IF head = null** Set ptr → next = null Set ptr → prev = null Set head =ptr go to step 10 [end of if]
5	Set ptr → prev = null	Set temp = head **For (i=0; i < loc; i++)** Set temp= temp→next	Set temp = head
6	Set ptr → next= head	**IF temp = null** Write NO INSERT go to step 10 [end of if and for]	Repeat Step 6 **while** **temp → next != null** Set temp = temp → next [end of loop]
7	Set head →prev = ptr	Set ptr→next=temp→next Set ptr→prev = temp	Set temp → next=ptr
8	Set head = ptr	Set temp →next = ptr	Set ptr →prev = temp
9	Exit	Set temp →next → prev= ptr	Set ptr → next = null
10		Exit	Exit

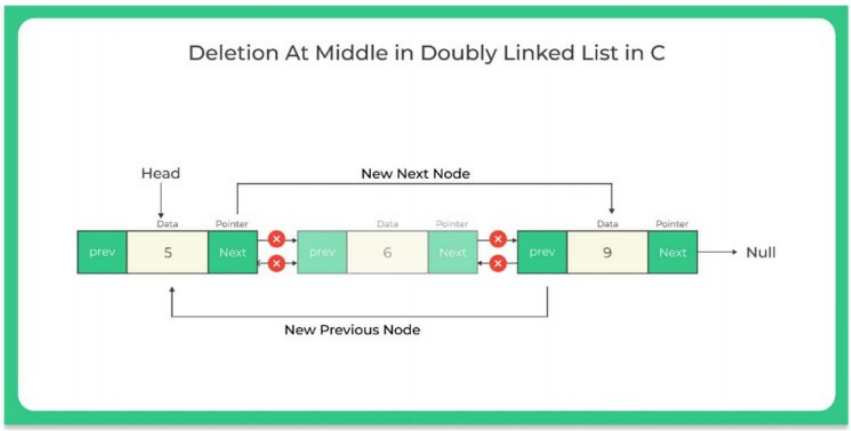
Deletion Operation:

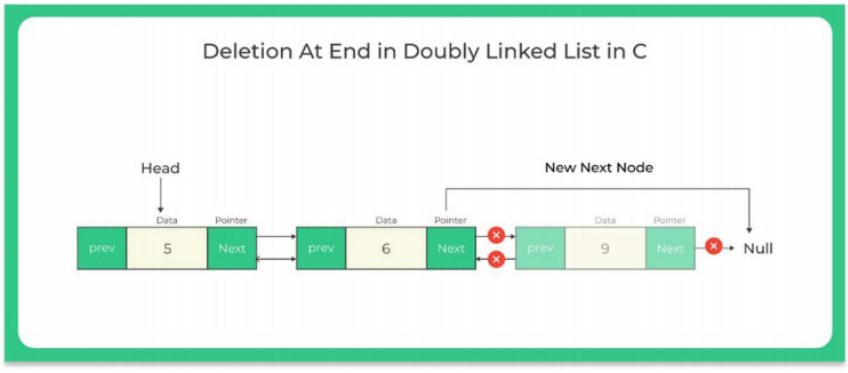
There are three possible positions from where we can delete a node in doubly linked list:

5) Deletion at Beginning:
6) Deletion after N^{th} Position:
7) Deletion at End:



Deletion in doubly linked list from beginning





DELETION ALGORITHMS			
Steps	**AT BEGINNING**	**AT Nth POSITION**	**AT END**
1	**IF head = null** Write UNDERFLOW go to step 7 [end of if]	**IF head = null** write UNDERFLOW go to step 8 [end of if]	**IF head = null** write UNDERFLOW go to step 7 [end of if]
2	**Else If** head→next=null Go to step 6 [end of else if]	Set temp = head	**Else If** head→next=null Go to step 6 [end of else if]
3	Set ptr = head	**If head→ data = val** Set ptr = head Set head=head→next Set head→prev = null Then Go to step 8 **Else go to step 4**	Set ptr = head
4	Set head=head→next	Repeat Step 4 **While** **temp →next→ data != val** temp = temp -> next [end of loop]	Repeat Step 4 **While** **ptr → next != null** Set ptr = ptr → next [end of loop]
5	Set head→prev = null	Set ptr = temp → next	Set ptr→prev→next= null
6	Set free ptr	Set temp→next = ptr → next	Set free ptr
7	Exit	Set ptr → next→prev = temp	Exit
8		Set free ptr	
9		Exit	

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
// Node structure for doubly linked list
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};
struct Node* head = NULL;
// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
// Function to insert a new node at the beginning of the doubly linked list
void insertAtBeginning(int value) {
    struct Node* newNode = createNode(value);
    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
}
// Function to insert a new node at the end of the doubly linked list
void insertAtEnd(int value) {
    struct Node* newNode = createNode(value);
```

```
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
}
// Function to insert a new node at a specific position in the doubly linked list
void insertAtPosition(int value, int position) {
    struct Node* newNode = createNode(value);
    if (position == 1) {
        if (head != NULL) {
            newNode->next = head;
            head->prev = newNode;
        }
        head = newNode;
        return;
    }
    struct Node* temp = head;
    int i;
    for (i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Invalid position\n");
        return;
    }
    newNode->next = temp->next;
```

```
    if (temp->next != NULL) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
// Function to delete a node from the beginning of the doubly linked list
void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    if (head != NULL) {
        head->prev = NULL;
    }
    free(temp);
}
// Function to delete a node from the end of the doubly linked list
void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    if (temp->prev != NULL) {
        temp->prev->next = NULL;
    } else {
```

```
        head = NULL;
    }
    free(temp);
}
// Function to delete a node from a specific position in the doubly linked list
void deleteFromPosition(int position) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* temp = head;
    if (position == 1) {
        head = head->next;
        if (head != NULL) {
            head->prev = NULL;
        }
        free(temp);
        return;
    }
    int i;
    for (i = 1; i < position && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Invalid position\n");
        return;
    }
    temp->prev->next = temp->next;
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    free(temp);
```

```
}
// Function to search for a value in the doubly linked list
void search(int value) {
    struct Node* temp = head;
    int position = 1;
    while (temp != NULL) {
        if (temp->data == value) {
            printf("Value %d found at position %d\n", value, position);
            return;
        }
        temp = temp->next;
        position++;
    }
    printf("Value %d not found in the list\n", value);
}
// Function to display the doubly linked list
void display() {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
int main() {
    int choice, value, position;
    do {
        // Menu for user choices
        printf("\nDoubly Linked List Operations:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
```

```
printf("4. Delete from Beginning\n");
printf("5. Delete from End\n");
printf("6. Delete from Position\n");
printf("7. Search\n");
printf("8. Display\n");
printf("0. Exit\n");
// User input for choice
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
   case 1:
      printf("Enter value to insert at the beginning: ");
      scanf("%d", &value);
      insertAtBeginning(value);
      break;
   case 2:
      printf("Enter value to insert at the end: ");
      scanf("%d", &value);
      insertAtEnd(value);
      break;
   case 3:
      printf("Enter value to insert: ");
      scanf("%d", &value);
      printf("Enter position to insert at: ");
      scanf("%d", &position);
      insertAtPosition(value, position);
      break;
   case 4:
      deleteFromBeginning();
      break;
   case 5:
      deleteFromEnd();
```

```
            break;
        case 6:
            printf("Enter position to delete from: ");
            scanf("%d", &position);
            deleteFromPosition(position);
            break;
        case 7:
            printf("Enter value to search: ");
            scanf("%d", &value);
            search(value);
            break;
        case 8:
            display();
            break;
        case 0:
            printf("Exiting program.\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
    }
  } while (choice != 0 && getchar() != '0');
  return 0;
}
```

Linked List in Array using C

```
#include <stdio.h>
#include <stdlib.h>
// Define a Node structure
struct Node {
  int data;
  struct Node* next;  };
// Function to create a new node
struct Node* createNode(int value) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
// Function to create a linked list from an array
struct Node* createLinkedList(int arr[], int n) {
    if (n == 0) {
        return NULL;
    }
    // Create the head node
    struct Node* head = createNode(arr[0]);
    struct Node* current = head;
    // Create and add nodes for the remaining elements
    for (int i = 1; i < n; ++i) {
        current->next = createNode(arr[i]);
        current = current->next;
    }
    return head;
}
// Function to print the linked list
void printLinkedList(struct Node* head) {
    while (head != NULL) {
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}
int main() {
    int n;
    // Get the size of the array from the user
    printf("Enter the size of the array: ");
```

```
scanf("%d", &n);
int arr[n];
// Get array elements from the user
printf("Enter the elements of the array:\n");
for (int i = 0; i < n; ++i) {
   scanf("%d", &arr[i]);
}
// Create a linked list from the array
struct Node* head = createLinkedList(arr, n);
// Print the linked list
printf("Linked List: ");
printLinkedList(head);
return 0; }
```

Polynomial:

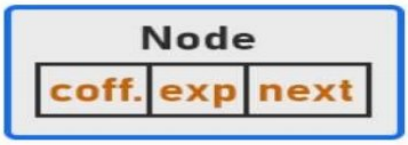
- A polynomial is a collection of different terms, each comprising coefficients, and exponents.
- It can be represented using a linked list. This representation makes polynomial manipulation efficient.
- While representing a polynomial using a linked list, each polynomial term represents a node in the linked list.
- To get better efficiency in processing, we assume that the term of every polynomial is stored within the linked list in the order of decreasing exponents. Also, no two terms have the same exponent, and no term has a zero coefficient and without coefficients. The coefficient takes a value of 1.

Polynomial Representation:

Each node of a linked list representing polynomial constitute three parts:

1) The first part contains the value of the coefficient of the term.
2) The second part contains the value of the exponent.
3) The third part, LINK points to the next term (next node).

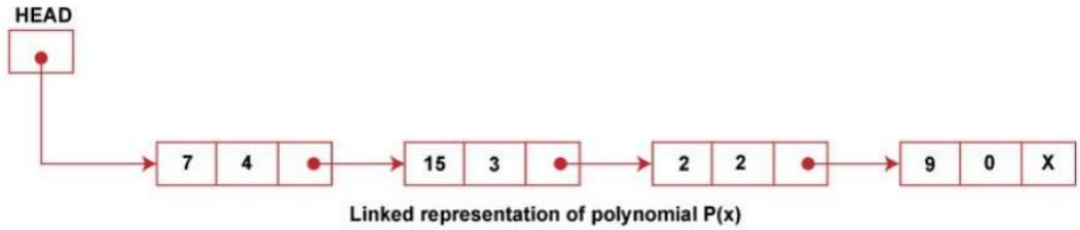
Node Structure:



Consider a polynomial $P(x) = 7x^4 + 15x^3 + 2x^2 + 9$.

Here 7, 15, 2, and 9 are the coefficients, and 4,3,2,0 are the exponents of the terms in the polynomial.

Representing this Polynomial using a Linked List:



Linked representation of polynomial P(x)

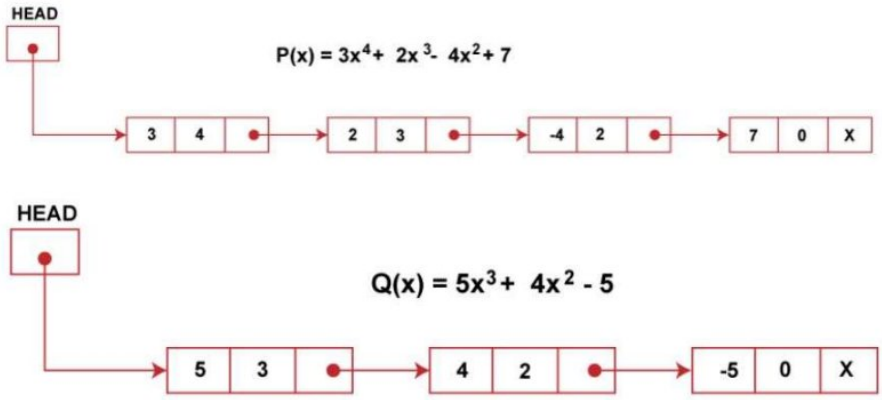
Addition of Polynomials:

Example:

$$\mathbf{P(x)} = 3x^4 + 2x^3 - 4x^2 + 7$$

$$\mathbf{Q(x)} = 5x^3 + 4x^2 - 5$$

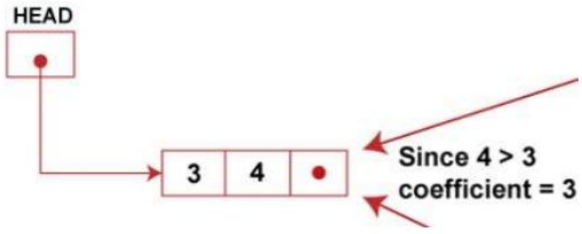
Solution:



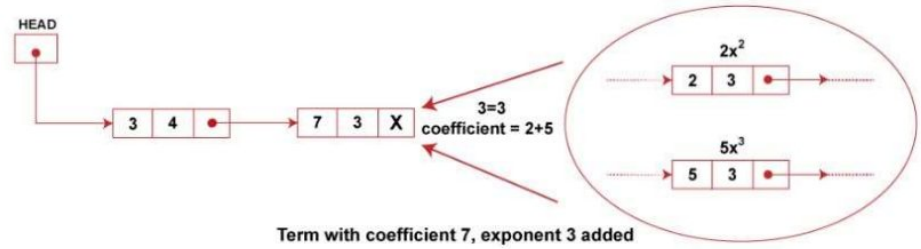
Steps for Addition of given polynomials P(x) and Q(x):

1) Traverse the two lists P and Q and examine all the nodes.
2) We compare the exponents of the corresponding terms of two polynomials.
3) The first term of polynomials P and Q contain exponents 4 and 3, respectively. Since the exponent of the first term of the polynomial P is greater than the other polynomial Q, the term having a larger exponent is inserted into the new list.

   The new list initially looks as:

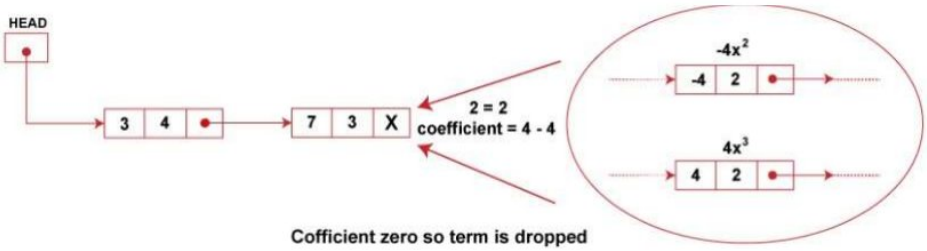


4) Compare the exponent of the next term of the list P with the exponents of the present term of list Q. Since the two exponents are equal, so their coefficients are added and appended to the new list as



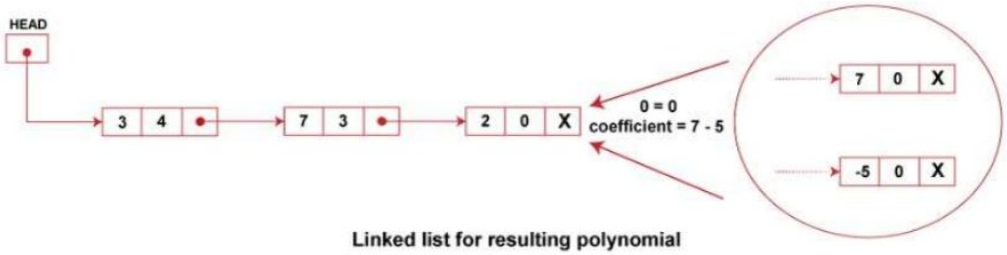
Term with coefficient 7, exponent 3 added

5) We move to the next term of P and Q lists and compare their exponents. Since exponents of both these terms are equal and after addition of their coefficients, we get 0, so the term is dropped, and no node is appended to the new list after this



Cofficient zero so term is dropped

6) Moving to the next term of the two lists, P and Q, we find that the corresponding terms have the same exponents equal to 0. We add their coefficients and append them to the new list for the resulting polynomial as



Linked list for resulting polynomial

Answer = $3x^4+7x^3+0x^2+0x^1+2x^0 = 3x^4+7x^3+2$

SOURCE CODE

```
#include <stdio.h>
#include <stdlib.h>
// Define a structure for a polynomial term
struct Term {
    int coefficient;
    int exponent;
    struct Term* next;      };
// Function to create a new term
struct Term* createTerm(int coef, int exp) {
    struct Term* newTerm = (struct Term*)malloc(sizeof(struct Term));
    newTerm->coefficient = coef;
    newTerm->exponent = exp;
    newTerm->next = NULL;
    return newTerm;         }
// Function to insert a term at the end of a linked list
void insertTerm(struct Term** poly, int coef, int exp) {
    struct Term* newTerm = createTerm(coef, exp);
    if (*poly == NULL) {
        *poly = newTerm;
    } else {
        struct Term* current = *poly;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newTerm;        }    }
// Function to add two polynomials
struct Term* addPolynomials(struct Term* poly1, struct Term* poly2) {
    struct Term* result = NULL;
    struct Term* current = NULL;
    while (poly1 != NULL || poly2 != NULL) {
        struct Term* newTerm;
```

```
        if (poly1 == NULL) {
            newTerm = createTerm(poly2->coefficient, poly2->exponent);
            poly2 = poly2->next;
        } else if (poly2 == NULL) {
            newTerm = createTerm(poly1->coefficient, poly1->exponent);
            poly1 = poly1->next;
        } else {
            if (poly1->exponent > poly2->exponent) {
                newTerm = createTerm(poly1->coefficient, poly1->exponent);
                poly1 = poly1->next;
            } else if (poly1->exponent < poly2->exponent) {
                newTerm = createTerm(poly2->coefficient, poly2->exponent);
                poly2 = poly2->next;
            } else {
                newTerm = createTerm(poly1->coefficient + poly2->coefficient, poly1->exponent);
                poly1 = poly1->next;
                poly2 = poly2->next;
            }
        }
        if (result == NULL) {
            result = newTerm;
            current = result;
        } else {
            current->next = newTerm;
            current = newTerm;
        }
    }
    return result;
}
// Function to display a polynomial
void displayPolynomial(struct Term* poly) {
    while (poly != NULL) {
```

```
        printf("%dx^%d", poly->coefficient, poly->exponent);
        poly = poly->next;
        if (poly != NULL) {
            printf(" + ");
        }
    }
    printf("\n");
}
// Function to free memory allocated for the linked list
void freeLinkedList(struct Term* poly) {
    struct Term* current = poly;
    struct Term* next;
    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
}
int main() {
    // Create two polynomials
    struct Term* poly1 = NULL;
    struct Term* poly2 = NULL;

    insertTerm(&poly1, 3, 4);
    insertTerm(&poly1, 2, 3);
    insertTerm(&poly1, -4, 2);
    insertTerm(&poly1, 7, 0);

    insertTerm(&poly2, 5, 3);
    insertTerm(&poly2, 4, 2);
    insertTerm(&poly2, -5, 0);
    // Display the original polynomials
```

```
    printf("Polynomial 1: ");
    displayPolynomial(poly1);
    printf("Polynomial 2: ");
    displayPolynomial(poly2);
    // Add the polynomials
    struct Term* result = addPolynomials(poly1, poly2);
    // Display the result
    printf("Sum of Polynomials: ");
    displayPolynomial(result);
    // Free the memory allocated for the linked lists
    freeLinkedList(poly1);
    freeLinkedList(poly2);
    freeLinkedList(result);
    return 0;
}
```

Output:

```
Polynomial 1: 3x^4 + 2x^3 + -4x^2 + 7x^0
Polynomial 2: 5x^3 + 4x^2 + -5x^0
Sum of Polynomials: 3x^4 + 7x^3 + 0x^2 + 2x^0
```

END

UNIT-1

SYLLABUS

Stack: Array Representation and Implementation of stack, Operations on Stacks: Push & Pop, Array Representation of Stack, Linked Representation of Stack, Operations Associated with Stacks, Application of stack: Conversion of Infix to Prefix and Postfix Expressions, Evaluation of postfix expression using stack. Recursion.

Stack:

- A stack is one of the most commonly used data structure.
- A stack, also called Last In First Out (LIFO) system, is a linear list in which insertion and deletion can take place only at one end, called top.
- This structure operates in much the same way as stack of trays.
- If we want to remove a tray from stack of trays it can only be removed from the top only.
- The insertion and deletion operation in stack terminology are known as PUSH and POP operations.
- Following operation can be performed on stack :
    1) **Create stack (s) :** To create an empty stack s.
    2) **PUSH (s, i) :** To push an element i into stack s.
    3) **POP (s) :** To access and remove the top element of the stack s.
    4) **Peek (s) :** To access the top element of stack s without removing it from the stack s.
    5) **Overflow :** To check whether the stack is full.
    6) **Underflow :** To check whether the stack is empty.

PUSH operation : In push operation, we insert an element onto stack. Before inserting, first we increase the top pointer and then insert the element.

Algorithm :

PUSH (STACK, TOP, MAX, DATA)

1) If TOP = MAX – 1 then write "STACK OVERFLOW and STOP"
2) READ DATA
3) TOP ← TOP + 1
4) STACK [TOP] ← DATA
5) STOP

POP operation : In pop operation, we remove an element from stack. After every pop operation top of stack is decremented by 1.

Algorithm :

POP (STACK, TOP, ITEM)

1) If TOP < 0 then write "STACK UNDERFLOW and STOP"
2) STACK [TOP] ←NULL
3) TOP ← TOP – 1
4) STOP

Implementation of PUSH Algorithm in C Language:

```
void push () {
```

```
int val, n;
  if (top == n )
  printf("\n Overflow");
  else  {
     printf("Enter the value?");
     scanf("%d",&val);
     top = top +1;
     stack[top] = val;     }  }
```

Implementation of POP Algorithm in C Language:

```
void pop ()  {
  if(top == -1)
  printf("Underflow");
  else
  top = top -1;  }
```

Implementation of PEEK Algorithm in C Language:

```
int peek()  {
  if (top == -1)    {
     printf("Underflow");
     return 0;   }
  else    {
     return stack [top];  }  }
```

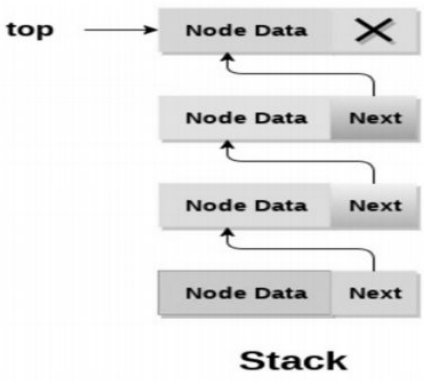
Complete C Program for Array Implementation of Stack:

```
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()  {
  clrscr();
  printf("Enter the number of elements in the stack ");
  scanf("%d",&n);
  while(choice != 4)  {
```

```
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
printf("\n Enter your choice");
        scanf("%d",&choice);
        switch(choice)    {
           case 1:        {
              push();
              break;        }
           case 2:         {
              pop();
              break;     }
           case 3:        {
              show();
              break;        }
           case 4:       {
              printf("Exiting....");
              break;     }
           default:    {
              printf("Please Enter valid choice ");     }      };     } }
void push ()  {
   int val;
   if (top == n )
   printf("\n Overflow");
   else     {
      printf("Enter the value?");
      scanf("%d",&val);
      top = top +1;
      stack[top] = val;      }   }
void pop ()   {
   if(top == -1)
   printf("Underflow");
   else
```

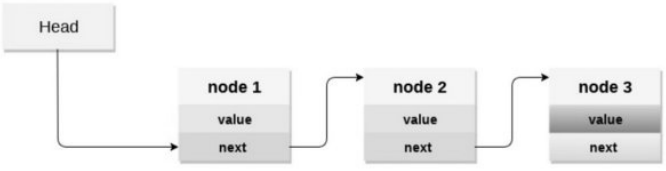
```
    top = top -1;   }
void show()  {
    for (i=top;i>=0;i--)     {
        printf("%d\n",stack[i]);   }
    if(top == -1)   {
        printf("Stack is empty");  }  }
```

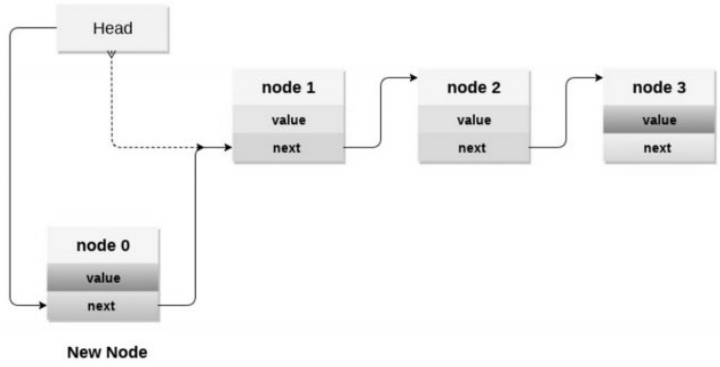
Linked List Implementation of Stack:



Adding a node to the stack (PUSH Operation):

- Create a node first and allocate memory to it.
- If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
- If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.





Deleting a node from the stack (POP Operation):

1) **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2) **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Display the nodes (Traversing):

1) Copy the head pointer into a temporary pointer.
2) Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Implementation of PUSH Algorithm in C Language:

```
void push () {
   int val;
   struct node *ptr = (struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)    {
      printf("not able to push the element");      }
   else      {
      printf("Enter the value");
      scanf("%d",&val);
      if(head==NULL)         {
         ptr->val = val;
         ptr -> next = NULL;
         head=ptr;       }
      else          {
         ptr->val = val;
         ptr->next = head;
         head=ptr;                    }
      printf("Item pushed");            } }
```

Implementation of POP Algorithm in C Language:

```
void pop() {
   int item;
   struct node *ptr;
   if (head == NULL)     {
      printf("Underflow");     }
```

```
    else    {
        item = head->val;
        ptr = head;
        head = head->next;
        free(ptr);
        printf("Item popped");          } }
```

Implementation of TRAVERSING/DISPLAY Algorithm in C Language:

```
void display()  {
    int i;
    struct node *ptr;
    ptr=head;
    if(ptr == NULL)     {
        printf("Stack is empty\n");     }
    else    {
        printf("Printing Stack elements \n");
        while(ptr!=NULL)        {
            printf("%d\n",ptr->val);
            ptr = ptr->next;        }     } }
```

Complete C Program for Linked List Implementation of Stack:

```
#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node   {
int val;
struct node *next;  };
struct node *head;
void main ()  {
    int choice=0;
    clrscr();
    while(choice != 4)    {
```

```
        printf("\n\nChose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)    {
          case 1:    {
             push();
             break;      }
          case 2:  {
             pop();
             break;              }
          case 3:             {
             display();
             break;           }
          case 4:             {
             printf("Exiting....");
             break;              }
          default:            {
             printf("Please Enter valid choice ");          }       }; } }
void push () {
   int val;
   struct node *ptr = (struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)     {
      printf("not able to push the element");       }
   else       {
      printf("Enter the value");
      scanf("%d",&val);
      if(head==NULL)          {
         ptr->val = val;
         ptr -> next = NULL;
         head=ptr;        }
      else          {
```

```
        ptr->val = val;
        ptr->next = head;
        head=ptr;                    }
    printf("Item pushed");          } }
void pop() {
  int item;
  struct node *ptr;
  if (head == NULL)    {
    printf("Underflow");    }
  else    {
    item = head->val;
    ptr = head;
    head = head->next;
    free(ptr);
    printf("Item popped");          } }
void display() {
  int i;
  struct node *ptr;
  ptr=head;
  if(ptr == NULL)    {
    printf("Stack is empty\n");    }
  else    {
    printf("Printing Stack elements \n");
    while(ptr!=NULL)        {
      printf("%d\n",ptr->val);
      ptr = ptr->next;      }    } }
```

Reversion a String using Stack:

```
#include<stdio.h>
#include<string.h>
#define size 20
int top = -1;
char stack[size];
```

```
char push(char ch)
{
 if(top==(size-1))
  printf("Stack is Overflow\n");
 else
  stack[++top]=ch;
  return 0;
}
char pop()
{
 if(top==-1)
  printf("Stack is Underflow\n");
 else
  return stack[top--];
  return 0;
}
void main()
{
 char str[20];
 int i;
 printf("Enter the string : \n" );
 gets(str);
 for(i=0;i<strlen(str);i++)
  {
   push(str[i]);
  }
 for(i=0;i<strlen(str);i++)
  {
     str[i]=pop();
  }
 printf("Reversed string is : ");
```

```
puts(str);
getch();
}
```

Application of Stack:

- The Stack is used as a Temporary Storage Structure for recursive operations.
- Stack is also utilized as Auxiliary Storage Structure for function calls, nested operations, and deferred/postponed functions.
- We can manage function calls using Stacks.
- Stacks are also utilized to evaluate the arithmetic expressions in different programming languages.
- Stacks are also helpful in converting infix expressions to postfix expressions.
- Stacks allow us to check the expression's syntax in the programming environment.
- We can match parenthesis using Stacks.
- Stacks can be used to reverse a String.
- Stacks are helpful in solving problems based on backtracking.
- We can use Stacks in depth-first search in graph and tree traversal.
- Stacks are also used in Operating System functions.
- Stacks are also used in UNDO and REDO functions in an edit.

Operation : Any Expression of algebraic format , Example (A + B) or (5+6)

Operands : A and B or 5 & 6 are operands

Operators : +. -, %, *, / etc are operators

Infix Notation:

- An infix notation is a notation in which an expression is written in a usual or normal format.
- It is a notation in which the operators lie between the operands.
- The examples of infix notation are A + B, A * B, A / B, etc

Prefix Notation/Polish Notation:

- The prefix expression is an expression in which the operator comes before the operands.
- The examples of prefix notation are +AB, *AB, /AB, etc

Postfix Expression

- The postfix expression is an expression in which the operator is written after the operands.
- The examples of postfix notation are AB+, AB*, AB/, etc

Precedence Order:

Operators	Symbols
Parenthesis	{ }, ( ), [ ]
Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

Associativity Order:

Operators	Associativity
^	Right to Left
*, /	Left to Right
+, -	Left to Right

Example:

1 + 2 * 3 + 30 / 5

1) The multiple (*) and divide (/) have the same precedence, so we will apply the associativity rule.
2) The multiple (*) and divide (/) operators have the left to right associativity, so we will scan from the leftmost operator
3) The operator that comes first will be evaluated first.
4) The operator * appears before the / operator, and multiplication would be done first.

1 + (2 * 3) + (30 / 5)

1 + 6 + 6 = 13

Conversion of Infix to Prefix Expressions:

a * b + c **(Infix expression)**

↓

*ab + c

↓

+ * a b c **(Prefix expression)**

Conversion of Infix to Prefix using Stack

Rules for the conversion of infix to prefix expression:

1. First, reverse the infix expression given in the problem.
2. Scan the expression from left to right.
3. Whenever the operands arrive, print them.
4. If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
5. If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
6. If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
7. If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
8. If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
9. When we reach the end of the expression, pop, and print all the operators from the top of the stack.

10. If the operator is ')', then push it into the stack.
11. If the operator is '(', then pop all the operators from the stack till it finds) opening bracket in the stack.
12. If the top of the stack is ')', push the operator on the stack.
13. At the end, reverse the output.

EXAMPLE:

K + L - M * N + (O ^ P) * W / U / V * T + Q

Firstly, we need to reverse the expression

Q + T * V / U / W *) P ^ O (+ N * M - L + K

Created a table that consists of three columns, i.e., input expression, stack, and prefix expression

When we encounter any symbol, we simply add it into the prefix expression.

If we encounter the operator, we will push it into the stack.

Input expression	Stack	Prefix expression
Q		Q
+	+	Q
T	+	Q T
*	+ *	Q T
V	+ *	Q T V
/	+ * /	Q T V
U	+ * /	Q T V U
/	+ * / /	Q T V U
W	+ * / /	Q T V U W
*	+ * / / *	Q T V U W
)	+ * / / *)	Q T V U W
P	+ * / / *)	Q T V U W P
^	+ * / / *) ^	Q T V U W P
O	+ * / / *) ^	Q T V U W P O
(	+ * / / *	Q T V U W P O ^
+	+ +	Q T V U W P O ^ * / / *
N	+ +	Q T V U W P O ^ * / / * N
*	+ + *	Q T V U W P O ^ * / / * N
M	+ + *	Q T V U W P O ^ * / / * N M
-	+ + -	Q T V U W P O ^ * / / * N M *
L	+ + -	Q T V U W P O ^ * / / * N M * L
+	+ + - +	Q T V U W P O ^ * / / * N M * L
K	+ + - +	Q T V U W P O ^ * / / * N M * L K
		Q T V U W P O ^ * / / * N M * L K + - + +

Q T V U W P O ^ * / / * N M * L K + - + +

is not a final expression. We need to reverse this expression to obtain the prefix expression.

+ + - + K L * M N * / / * ^ O P W U V T Q

Conversion of Prefix to Infix using Stack

Rules for the conversion of prefix to infix expression:

1. First, we will reverse the given prefix expression and read it from left to right.
2. We will use a stack to store the operands. So, if we encounter an operand, we will push it into the stack
3. If we encounter an operator, then we will pop two operands out of the stack and put the operator between them.
4. We will repeat the above steps until we reach the end of the given prefix expression.

Prefix : + + - + K L * M N * / / * ^ O P W U V T Q

Reverse of Prefix : Q T V U W P O ^ * / / * N M * L K + - + +

Input expression	Stack Top	Stack (Infix expression)
Q	Q	Q
T	T	T Q
V	V	V T Q
U	U	U V T Q
W	W	W U V T Q
P	P	P W U V T Q
O	O	O P W U V T Q
^	(O^P)	(O^P) W U V T Q
*	(O^P)*W	(O^P)*W U V T Q
/	(O^P)*W/U	(O^P)*W/U V T Q
/	(O^P)*W/U/V	(O^P)*W/U/V T Q
*	(O^P)*W/U/V*T	(O^P)*W/U/V*T Q
N	N	N (O^P)*W/U/V*T Q
M	M	M N (O^P)*W/U/V*T Q
*	M*N	M*N (O^P)*W/U/V*T Q
L	L	L M*N (O^P)*W/U/V*T Q
K	K	K L M*N (O^P)*W/U/V*T Q
+	K+L	K+L M*N (O^P)*W/U/V*T Q
-	K+L-M*N	K+L-M*N (O^P)*W/U/V*T Q
+	K+L-M*N+(O^P)*W/U/V*T	K+L-M*N+(O^P)*W/U/V*T Q
+	K+L-M*N (O^P)*W/U/V*T+Q	K+L-M*N+(O^P)*W/U/V*T+Q

Infix will be K + L - M * N + (O ^ P) * W / U / V * T + Q

Conversion of Infix to Postfix using Stack

Rules for the conversion from infix to postfix expression:

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack, then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Example: Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Input	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(	+ (	K L + M N *-
O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

Postfix KL+MN*-OP^W*U/V/T*+Q+

Conversion of Postfix to Infix using Stack

Postfix KL+MN*-OP^W*U/V/T*+Q+

Input expression	Stack Top	Stack (Infix expression)
K	K	K
L	L	L K
+	K+L	K+L
M	M	M K+L
N	N	N M K+L
*	M*N	M*N K+L
-	K+L-M*N	K+L-M*N
O	O	O K+L-M*N
P	P	P O K+L-M*N
^	O^P	O^P K+L-M*N
W	W	WO^PK+L-M*N
*	O^P*W	O^P*W K+L-M*N
U	U	U O^P*W K+L-M*N
/	O^P*W/U	O^P*W/U K+L-M*N
V	V	V O^P*W/U K+L-M*N
/	O^P*W/U/V	O^P*W/U/V K+L-M*N
T	T	T O^P*W/U/V K+L-M*N
*	O^P*W/U/V*T	O^P*W/U/V*T K+L-M*N
+	K+L-M*N+O^P*W/U/V*T	K+L-M*N+O^P*W/U/V*T
Q	Q	Q K+L-M*N+O^P*W/U/V*T
+	K+L-M*N+O^P*W/U/V*T+Q	K+L-M*N+O^P*W/U/V*T+Q

Conversion of Prefix to Postfix using Stack

Prefix: + + * + A B C / D + B * A C D

Reverse: D C A * B + D / C B A + * + +

Input expression	Stack Top	Stack (Postfix expression)
D	D	D
C	C	C D
A	A	A C D
*	AC*	AC* D
B	B	B AC* D
+	BAC*+	BAC*+ D
D	D	D BAC*+ D
/	DBAC*+/	DBAC*+/ D
C	C	C DBAC*+/ D
B	B	B C DBAC*+/ D
A	A	A B C DBAC*+/ D
+	AB+	AB+ C DBAC*+/ D
*	AB+C*	AB+C* DBAC*+/ D
+	AB+C*DBAC*+/+	AB+C*DBAC*+/+ D
+	AB+C*DBAC*+/+D+	AB+C*DBAC*+/+D+

AB+C*DBAC*+/+D+

Conversion of Postfix to Prefix using Stack

Postfix: AB+C*DBAC*+/+D+

Input expression	Stack Top	Stack (Postfix expression)
A	A	A
B	B	BA
+	+AB	+AB
C	C	C +AB
*	*+ABC	*+ABC
D	D	D *+ABC
B	B	B D *+ABC
A	A	A B D *+ABC
C	C	C A B D *+ABC
*	*AC	*AC B D *+ABC
+	+B*AC	+B*AC D *+ABC
/	/D+B*AC	/D+B*AC *+ABC
+	+*+ABC/D+B*AC	+*+ABC/D+B*AC
D	D	D +*+ABC/D+B*AC
+	++*+ABC/D+B*ACD	++*+ABC/D+B*ACD

++*+ABC/D+B*ACD

Recursion:

- Recursion is a process of expressing a function that calls itself to perform specific operation.
- Indirect recursion occurs when one function calls another function that then calls the first function.

- Suppose P is a procedure containing either a call statement to itself or a call stateme4nt to a second procedure that may eventually result in a call statement back to the original procedure P.
- Then P is called recursive procedure. So, the program will not continue to run indefinitely.
- A recursive procedure must have the following two properties :
  - **a)** There must be certain criteria, called base criteria, for which the procedure does not call itself.
  - **b)** Each time the procedure does call itself, it must be closer to the criteria.
- A recursive procedure with these two properties is said to be well defined.
- Similarly, a function is said to be recursively defined if the function definition refers to itself.
- Again, in order for the definition not to be circular, it must have the following two properties :
  - **a)** There must be certain arguments, called base values, for which the function does not refer to itself.
  - **b)** Each time the function does refer to itself, the argument of the function must be closer to a base value.

Types of Recursion :

1) **Direct recursion :** A function is directly recursive if it contains an explicit call to itself.
   For example :
```
int foo (int x)
{ if (x <= 0)
return x;
return foo (x – 1);
}
```
2) **Indirect recursion:** A function is indirectly recursive if it contains a call to another function.
   For example :
```
int foo (int x)
{ if (x <= 0)
return x;
return bar (x) ;
}
int bar (int y)
{ return foo (y – 1) ;
}
```
3) **Tail recursion :**
   - Tail recursion (or tail-end recursion) is a special case of recursion in which the last operation of the function, the tail call is a recursive call. Such recursions can be easily transformed to iterations.
   - Replacing recursion with iteration, manually or automatically, can drastically decrease the amount of stack space used and improve efficiency.
   - Converting a call to a branch or jump in such a case is called a tail call optimization.
     For example :
     Consider this recursive definition of the factorial function in C :
```
factorial (n)
{
if( n == 0)
return 1;
return n * factorial (n – 1);
}
```

- This definition is tail recursive since the recursive call to factorial is not the last thing in the function (its result has to be multiplied by n).

```
factorial (n, accumulator)
{
if(n == 0)
return accumulator;
return factorial (n – 1, n * accumulator);
}
factorial (n)
{
return factorial (n – 1) ;
}
```

4) Linear and tree recursive :

- A recursive function is said to be linearly recursive when no pending operation involves another recursive call to the function.
- A recursive function is said to be tree recursive (or non-linearly recursive) when the pending operation does involve another recursive call to the function.
- The Fibonacci function fib provides a classic example of tree recursion. The Fibonacci numbers can be defined by the rule :

```
int fib (int n)
{ /* n >= 0 */
if (n == 0)
return 0;
if (n == 1)
return 1;
return fib (n – 1) + fib (n – 2) ;
}
```

- The pending operation for the recursive call is another call to fib. Therefore, fib is tree recursive.