SN	DSA
1	**Illustrate a c function to add two polynomials. Show the linked list representation of two polynomials and its addition.**

Each polynomial term is represented as a node in a linked list containing:

1. **Coefficient**
2. **Exponent**
3. **Pointer to the next node**

Example:

- Polynomial (x) = 3x^3 + 5x^2 + 2:
  Head → (3,3) → (5,2) → (2,0)
- Polynomial (x) = 4x^3 + 2x + 7:
  Head → (4,3) → (2,1) → (7,0)

C Function for Adding Two Polynomials

This function takes two linked lists representing polynomials and returns a new linked list for the resultant polynomial

```c
// Function to add two polynomials
NODE addPolynomials(NODE poly1, NODE poly2) {
    NODE result = NULL;
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exp == poly2->exp) {
            result = insertTerm(result, poly1->coeff + poly2->coeff, poly1->exp);
            poly1 = poly1->next;
            poly2 = poly2->next;
        } else if (poly1->exp > poly2->exp) {
            result = insertTerm(result, poly1->coeff, poly1->exp);
            poly1 = poly1->next;
        } else {
            result = insertTerm(result, poly2->coeff, poly2->exp);
            poly2 = poly2->next;
        }
    }

    // Add remaining terms of poly1 or poly2
    while (poly1 != NULL) {
        result = insertTerm(result, poly1->coeff, poly1->exp);
        poly1 = poly1->next;
    }

    while (poly2 != NULL) {
        result = insertTerm(result, poly2->coeff, poly2->exp);
        poly2 = poly2->next;
    }

    return result;
}
```

Example: Adding Two Polynomials
Input:
- Polynomial 1: P(x) = 3x^3 + 5x^2 + 2
- Polynomial 2: Q(x) = 4x^3 + 2x + 7

Output:
- Resultant Polynomial: R(x) = 7x^3 + 5x^2 + 2x + 9

Linked List Representation:
- P(x): Head → (3,3) → (5,2) → (2,0)
- Q(x): Head → (4,3) → (2,1) → (7,0)
- R(x): Head → (7,3) → (5,2) → (2,1) → (9,0)

3. **Describe double linked list with advantages and disadvantages. Write a C functions to delete the node from circular double linked list.**

A **doubly linked list (DLL)** is a type of linked list where each node contains three components:
1. **Data**: Stores the value of the node.
2. **Pointer to the next node**: Points to the next node in the list.
3. **Pointer to the previous node**: Points to the previous node in the list.

Representation
NULL <- [prev | data | next] <-> [prev | data | next] <-> [prev | data | next] -> NULL
In a **circular doubly linked list**, the next pointer of the last node points to the first node, and the prev pointer of the first node points to the last node, forming a circular structure.

Advantages of Doubly Linked List
1. **Bidirectional Traversal**: Can traverse the list both forwards and backwards.
2. **Efficient Insertion/Deletion**:
   - Nodes can be inserted or deleted without traversing the list to find the previous node.
3. **Reversible Operations**: Can reverse the list easily due to the presence of prev pointers.

Disadvantages of Doubly Linked List
1. **Memory Overhead**: Each node requires extra memory to store the prev pointer.
2. **Complex Implementation**: Maintaining two pointers (next and prev) increases implementation complexity.
3. **Extra Operations**: More updates are required during insertion and deletion to maintain both pointers.

Structure of a Node
```
struct Node {
    int data;                   // Data value of the node
    struct Node* next;          // Pointer to the next node
    struct Node* prev;          // Pointer to the previous node
};
typedef struct Node* NODE;     // Defining NODE as a pointer to struct Node
```

C Function to Delete a Node from a Circular Doubly Linked List

```c
NODE deleteNode(NODE head, int key) {
    if (head == NULL) {
        printf("List is empty\n");
        return NULL;
    }

    NODE cur = head;

    // Search for the node with the given key
    do {
        if (cur->data == key) {
            // Case: Only one node in the list
            if (cur->next == cur && cur->prev == cur) {
                free(cur);
                return NULL;
            }

            // Update pointers of adjacent nodes
            cur->prev->next = cur->next;
            cur->next->prev = cur->prev;

            // Update the head if necessary
            if (cur == head) {
                head = cur->next;
            }

            free(cur);
            printf("Node with data %d deleted\n", key);
            return head;
        }
        cur = cur->next;
    } while (cur != head);

    printf("Key %d not found in the list\n", key);
    return head;
}
```

Explanation

1. **Structure**:
   - The Node structure contains data and two pointers, next (to the next node) and prev (to the previous node).
2. **Input Parameters**:
   - head: Pointer to the head node of the circular doubly linked list.
   - key: The data value of the node to delete.
3. **Steps in the Function**:
   - If the list is empty (head == NULL), return with a message.
   - Traverse the list to find the node with the given key.
   - If the node is found:
     - If it's the only node, free it and return NULL.
     - Otherwise, update the next pointer of the previous node and the prev pointer of the next node.
     - Update the head if the deleted node is the current head.
   - If the key is not found, return the unchanged list with a message.
4. **Special Cases**:
   - List is empty.
   - List has only one node.
   - Node to delete is the head.

4 **Define a tree? With the suitable example, define:**
a) Binary tree b) Degree of binary tree c) Level of binary tree d) Complete binary tree

Tree Data Structure

A **tree** is a hierarchical, non-linear data structure where elements are organized in the form of nodes connected by edges. It is defined as:

1. A set of nodes, where one node is the **root**.
2. The rest of the nodes are partitioned into disjoint subsets, each being a **subtree** of the root.

Binary Tree

A **binary tree** is a tree where each node has at most two children, often referred to as the **left child** and **right child**.

Example of a Binary Tree

```
       A
     /   \
    B     C
   / \   / \
  D   E F   G
```

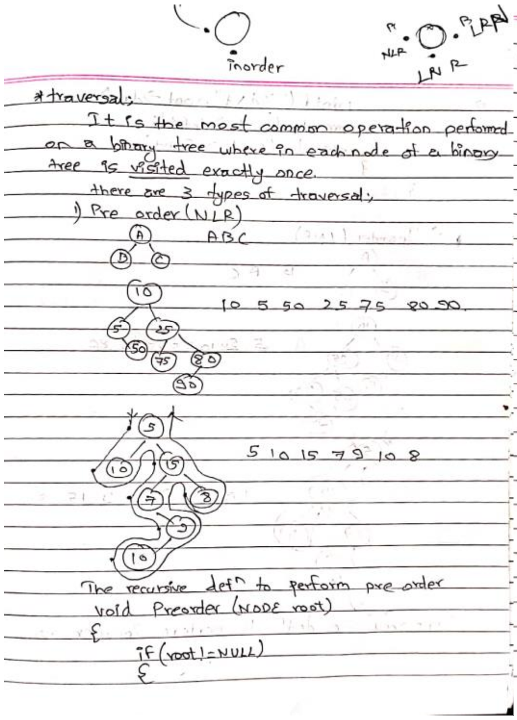
Key Concepts of a Binary Tree

1. **Degree of a Binary Tree**:
   The degree of a node is the number of children it has. In a binary tree:
   - A node's degree is at most 2.
   - The **degree of a tree** is the maximum degree of any node in the tree.
     Example: In the tree above:
   - Degree of `A` is 2 (two children: `B` and `C`).
   - Degree of `D`, `E`, `F`, and `G` is 0 (no children).
2. **Level of a Binary Tree**:
   The **level** of a node is its depth in the tree. The root is at level 0, its children are at level 1, and so on.
   Example:
   - Node `A` is at level 0.
   - Nodes `B` and `C` are at level 1.
   - Nodes `D`, `E`, `F`, and `G` are at level 2.
3. **Complete Binary Tree**:
   A binary tree is **complete** if:
   - All levels except possibly the last level are completely filled.
   - Nodes in the last level are as far left as possible.
     Example of a Complete Binary Tree:

```
4.           1
5.         /   \
6.        2      3
7.      / \    /
8.     4   5  6
```

This content aligns with the details from your uploaded PDF. Let me know if further elaboration is needed on any topic!

5 **State a C routine to traverse the binary tree**



inorder
NLR
LRN
LNR

* traversal:

It is the most common operation performed on a binary tree where in each node of a binary tree is visited exactly once.

there are 3 types of traversal:

1) Pre order (NLR)

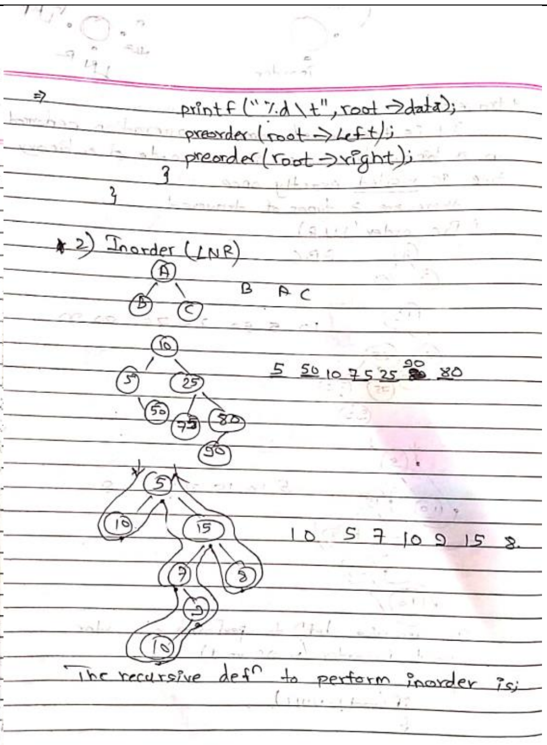
ABC

10 5 50 25 75 80 90.

5 10 15 7 9 10 8

The recursive defn to perform pre order

```
void Preorder (NODE root)
{
    if (root != NULL)
    {
```



```
=>      printf("%d\t", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

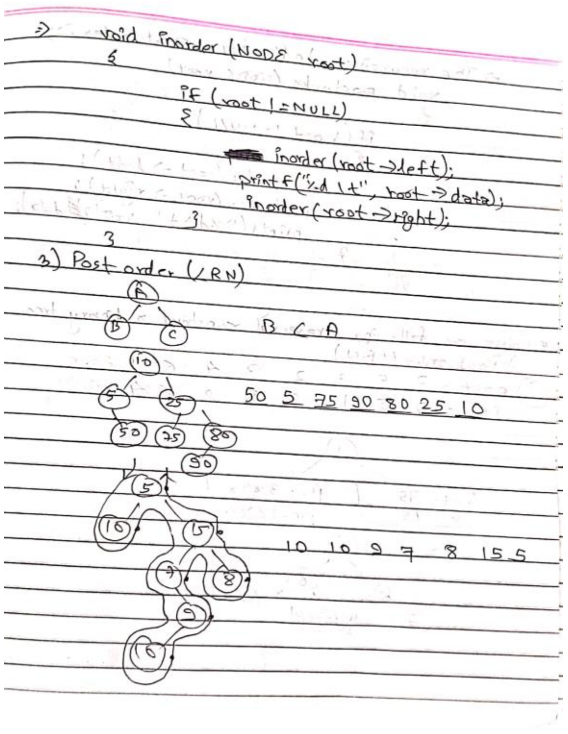
*2) Inorder (LNR)

B A C

5 50 10 75 25 90 80

10 5 7 10 9 15 8.

The recursive defn to perform inorder is:



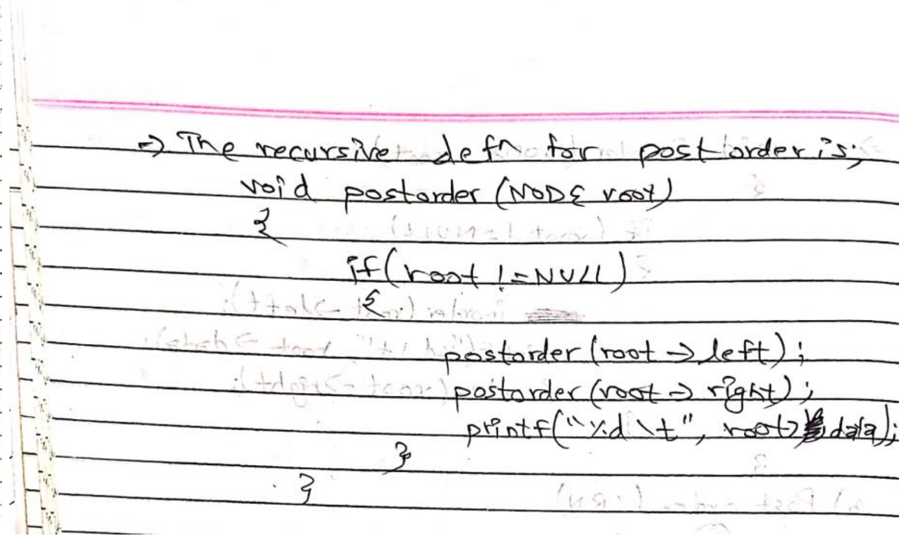
```
=> void inorder (NODE root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d\t", root->data);
        inorder(root->right);
    }
}
```

3) Post order (LRN)

B C A

50 5 75 90 80 25 10

10 10 9 7 8 15 5



=> The recursive defn for post order is;

```
void postorder (NODE root)
{
    if(root != NULL)
    {
        postorder(root->left);
        postorder(root->right);
        printf("%d\t", root->data);
    }
}
```

7	With The Snippet of Code Explain insertion techniques to Threaded Binary Tree a) Insertion of New Node as Left child b) Insertion of New Node as Right Child c) Insertion of New Node as Root

Here are the code snippets for each of the three insertion techniques (left child, right child, and root node) in a **Threaded Binary Tree**, followed by explanations for each technique.

1. Insertion of New Node as Left Child

This code snippet demonstrates how to insert a new node as the **left child** in a threaded binary tree.

```
void insertLeft(threadedPointer s, threadedPointer r) {
    /* Insert r as the left child of s */
    threadedPointer temp;

    // If the left child of s is NULL, make r the left child
    r->leftChild = s->leftChild;
    r->leftThread = s->leftThread;
    r->rightChild = s;
    r->rightThread = TRUE;
    s->leftChild = r;
    s->leftThread = FALSE;

    // If r's left child is not a thread, update the inorder predecessor
    if (!r->leftThread) {
        temp = insucc(r);  // Find the inorder successor
        temp->rightChild = r;  // Set it to point to r
    }
}
```

Explanation:

- **Step 1**: We check if the left child of node `s` is null. If it is, we can directly insert `r` as the left child of `s`.
- **Step 2**: Set the left child of `r` to the current left child of `s`, and set `r`'s `leftThread` to the previous `leftThread` of `s`.
- **Step 3**: Set `r` as the left child of `s`, and update `s`'s `leftThread` to `FALSE` to indicate that `r` is now a real child (not a thread).
- **Step 4**: If the left child of `r` was not a thread, we find its in-order successor using the `insucc` function and update the successor's right child to point to `r`.

---

2. Insertion of New Node as Right Child

This code snippet demonstrates how to insert a new node as the **right child** in a threaded binary tree.

```
void insertRight(threadedPointer s, threadedPointer r) {
    /* Insert r as the right child of s */
    threadedPointer temp;

    // If the right child of s is NULL, make r the right child
    r->rightChild = s->rightChild;
    r->rightThread = s->rightThread;
    r->leftChild = s;
    r->leftThread = TRUE;
    s->rightChild = r;
    s->rightThread = FALSE;

    // If r's right child is not a thread, update the inorder successor
    if (!r->rightThread) {
        temp = insucc(r);  // Find the inorder successor
        temp->leftChild = r;  // Set it to point to r
    }
}
```

Explanation:

- **Step 1**: If the right child of `s` is null, we insert `r` as the right child of `s`.
- **Step 2**: We update `r`'s right child to the current right child of `s`, and `r`'s `rightThread` is set to `s`'s `rightThread`.
- **Step 3**: Set `r` as the right child of `s`, and set `s`'s `rightThread` to `FALSE`.
- **Step 4**: If the right child of `r` is not a thread (i.e., it's a real child), we find the in-order successor of `r` and update the successor's left child to point to `r`.

---

3. Insertion of New Node as Root

This code snippet demonstrates how to insert a new node as the **root** of a threaded binary tree.

```
void insertRoot(threadedPointer *root, threadedPointer r) {
    /* Insert r as the root of the tree */
    threadedPointer temp;

    // Set the left and right child of the new root
    r->leftChild = (*root)->leftChild;
    r->rightChild = (*root)->rightChild;
    r->leftThread = (*root)->leftThread;
    r->rightThread = (*root)->rightThread;

    // Update the header node's left child to point to the new root
    (*root)->leftChild = r;
    (*root)->leftThread = FALSE;

    // Update the in-order successor if necessary
    if (!r->rightThread) {
        temp = insucc(r);
        temp->leftChild = r;
    }
}
```

Explanation:

- **Step 1**: When inserting a new node as the root, we set the new root `r` to inherit the left and right child, and their corresponding threads from the old root.
- **Step 2**: Update the header node (pointed by `root`) to point to `r` as the left child, and set `leftThread` to `FALSE` to indicate that `r` is now the real root (not a thread).
- **Step 3**: If `r`'s right child is not a thread, we update the in-order successor of `r` to point to `r` as the predecessor.

Summary of the Insertion Techniques:

1. **Insertion as Left Child**:
   - Insert the new node `r` as the left child of node `s`, and handle threading if necessary.
   - If the left child is null, the insertion is straightforward. Otherwise, adjust the threading.
2. **Insertion as Right Child**:
   - Insert the new node `r` as the right child of node `s`, updating the right child and handling threading.
   - If the right child is null, the insertion is direct. Otherwise, adjust the threading.
3. **Insertion as Root**:
   - Insert the new node `r` as the root by updating the root node's pointers to point to the new node.
   - The previous root becomes a child of the new root, and threading is adjusted to maintain the tree structure.

These methods ensure that the threaded binary tree remains properly structured, with all the necessary threads set to allow for efficient in-order traversal without requiring additional data structures like stacks.

10. **Discuss collision? What are the methods to resolve collision? Explain linear probing with an example**

Collision in Hashing:

A **collision** occurs in a hash table when two different keys are hashed to the same location. In other words, when multiple keys map to the same index (or bucket) in the hash table, a collision happens. This can lead to inefficiencies in both the search and insertion processes.

Methods to Resolve Collision:

There are several methods to handle collisions in a hash table. The most common ones are:

1. **Linear Probing (Open Addressing)**:
   - This method involves searching for the next available position in the hash table linearly (sequentially) after a collision.
   - When a collision occurs at a position `h`, the algorithm checks positions `h+1`, `h+2`, `h+3`, etc., wrapping around the table if needed (circular probing).
2. **Quadratic Probing**:

    - This is a variation of linear probing, where the step size increases quadratically instead of linearly. If a collision occurs at position `h`, the next positions checked are `h+1^2`, `h+2^2`, `h+3^2`, and so on.
3. **Double Hashing**:
    - This method uses two hash functions. The first function computes the initial index, and if a collision occurs, the second hash function provides a step size to find the next available index.
4. **Chaining**:
    - This technique uses linked lists or other data structures to store multiple elements that hash to the same location. Each bucket points to a linked list, and all colliding keys are added to this list.
5. **Rehashing**:
    - Rehashing is performed when the load factor of the hash table exceeds a certain threshold. The table size is doubled, and all existing keys are rehashed and inserted into the new table.

Explanation of Linear Probing with Example:

Linear probing resolves collisions by finding the next available slot in the table after a collision. If the hash function `H(k) = k % m` results in a collision, the next available slot is checked at `H(k) + 1`, `H(k) + 2`, and so on, wrapping around the table if necessary.

Example:

Let's use linear probing to insert keys into a hash table of size 10 with the hash function `H(k) = k % 10`. We want to insert the following keys: 72, 27, 36, 24, 63, 81, 92, 101.

Step-by-step insertion:

- **Insert 72**: `H(72) = 72 % 10 = 2`. The table is empty at position 2. Insert 72 at index 2.
- **Insert 27**: `H(27) = 27 % 10 = 7`. The table is empty at position 7. Insert 27 at index 7.
- **Insert 36**: `H(36) = 36 % 10 = 6`. The table is empty at position 6. Insert 36 at index 6.
- **Insert 24**: `H(24) = 24 % 10 = 4`. The table is empty at position 4. Insert 24 at index 4.
- **Insert 63**: `H(63) = 63 % 10 = 3`. The table is empty at position 3. Insert 63 at index 3.
- **Insert 81**: `H(81) = 81 % 10 = 1`. The table is empty at position 1. Insert 81 at index 1.
- **Insert 92**: `H(92) = 92 % 10 = 2`. A collision occurs at index 2 because it is already occupied by 72. The next available position is 3, but it's already filled by 63. So, we check position 4, which is filled by 24. Next, check position 5, which is empty. Insert 92 at index 5.
- **Insert 101**: `H(101) = 101 % 10 = 1`. A collision occurs at index 1 because it is filled by 81. The next available position is 2, but it's filled by 72. Check positions 3 and 4, which are also occupied. Finally, position 8 is available, so insert 101 at index 8.

Final Hash Table:

```
Index:  0   1    2    3    4    5    6    7    8    9
Values: -  81   72   63   24   92   36   27   101  -
```

Key Summary:

- **Key 72** was inserted at index 2.

- **Key 27** was inserted at index 7.
- **Key 36** was inserted at index 6.
- **Key 24** was inserted at index 4.
- **Key 63** was inserted at index 3.
- **Key 81** was inserted at index 1.
- **Key 92** was inserted at index 5 after linear probing.
- **Key 101** was inserted at index 8 after linear probing.

Average Search Length:

- The average search length is calculated as the average number of probes needed to insert all keys, which is 1+1+1+1+4+1+1+88=2.25\frac{1+1+1+1+4+1+1+8}{8} = 2.25.

This shows how **linear probing** helps to resolve collisions by systematically searching for the next available position in the hash table.

11. Describe hashing? Explain any three hash functions.

Hashing

Hashing is a technique used to map data of arbitrary size to fixed-size values, called hash values, using a hash function. The hash value serves as an index to store and retrieve data efficiently in a hash table. The primary goal of hashing is to minimize collisions (when two keys map to the same index) and ensure a uniform distribution of data across the hash table.

Three Common Hash Functions

1. **Division Method**
   - **Explanation**: This method divides the key xx by the size of the hash table MM and uses the remainder as the hash value. h(x)=xmod Mh(x) = x \mod M
   - **Advantages**:
     - Simple and fast.
     - Works well if MM (table size) is chosen as a prime number to reduce clustering.
   - **Example**:
     Key = 25, Table Size M=7M = 7: h(25)=25mod 7=4h(25) = 25 \mod 7 = 4 Key 25 is mapped to index 4.

---

2. **Mid-Square Method**
   - **Explanation**: The key is squared, and specific middle digits are extracted as the hash value.
     - This reduces the chance of clustering since the square operation spreads the values more uniformly.
   - **Steps**:

Square the key KK.

Extract the middle ll digits from the result.

- **Example**:
  Key = 1234: K2=1522756⇒H(1234)=27 (middle two digits).K^2 = 1522756 \quad \Rightarrow \quad H(1234) = 27 \; \text{(middle two digits)}.

---

3. **Folding Method**
   - **Explanation**: This method splits the key into equal parts, adds them together, and ignores the carry, if any.
   - **Steps**:
     1. Divide the key into parts (e.g., 123456 into 123, 456).
     2. Add the parts together.
     3. Use the sum, modulo the table size, as the hash value.
   - **Example**:
     Key = 123456, Table Size = 100:
     - Divide: 123,456123, 456
     - Add: 123+456=579123 + 456 = 579
     - Hash Value: 579mod 100=79579 \mod 100 = 79.

These functions ensure uniform distribution and help avoid clustering, which is crucial for efficient hash table performance.

.	