

Course Code	:	MCS-021
Course Title	:	Data and File Structures
Assignment Number	:	BCA(III)/021/Assignment/2024-25
Maximum Marks	:	100
Weightage	:	30%
Last Dates for Submission	:	31stOctober,2024(For July Session)
	:	30thApril,20245(For January Session)

This assignment has 16 questions of 5 Marks each, answer all questions. Rest 20 marks are for viva voce. Please go through the guidelines regarding assignments given in the Programme Guide for the format of presentation.

- Q1.** Write a program in C to accepts two polynomials as input and prints the resultant polynomial due to the multiplication of input polynomials.
- Q2.** Write a program in 'C' to create a single linked list and perform the following operations on it:
- (i) Insert a new node at the beginning, in the middle or at the end of the linked list.
 - (ii) Delete a node from the linked list
 - (iii) Display the linked list in reverse order
 - (iv) Sort and display data of the linked list in ascending order.
 - (v) Count the number of items stored in a single linked list
- Q3.** Write a program in 'C' to create a doubly linked list to store integer values and perform the following operations on it:
- (i) Insert a new node at the beginning, in the middle or at the end of the linked list.
 - (ii) Delete a node from the linked list
 - (iii) Sort and display data of the doubly linked list in ascending order.
 - (iv) Count the number of items stored in a single linked list
 - (v) Calculate the sum of all even integer numbers, stored in the doubly linked list.
- Q4.** What is a Dequeue? Write algorithm to perform insert and delete operations in a Dequeue.
- Q5.** Draw the binary tree for which the traversal sequences are given as follows:
- (i) Pre order: A B D E F C G H I J K
In order: B E D F A C I H K J G
 - (ii) Post order: I J H D K E C L M G F B A
In order: I H J D C K E A F L G M B
- Q6.** Write a program in 'C' to implement a binary search tree (BST). Traverse and display the binary search tree in the Inorder, Preorder and Post order form.

- Q7.** Define AVL tree. Create an AVL tree for the following list of data if the data are inserted in the order in an empty AVL tree.

12, 5, 15, 20, 35, 8, 2, 40, 14, 24, 27, 45, 50, 3, 4

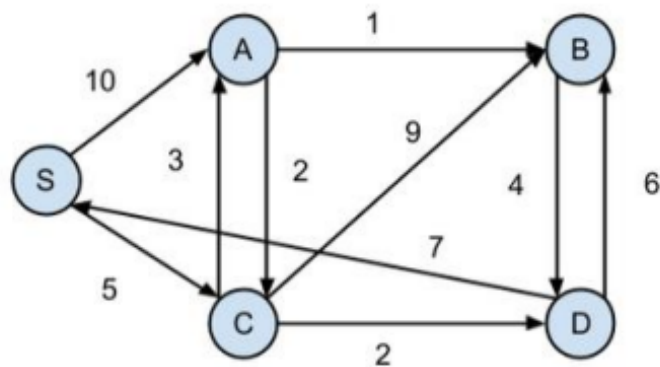
Further delete 2, 4, 5 and 12 from the above AVL tree.

- Q8.** Define a B-tree and its properties. Create a B-tree of order-5, if the data items are inserted into an empty B-tree in the following sequence:

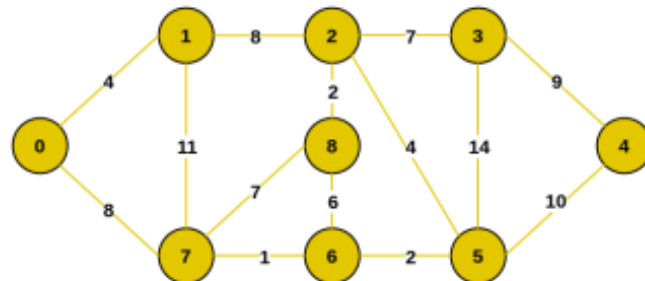
12, 5, 15, 20, 60, 45, 35, 40, 25, 8, 7, 55, 50, 66, 65, 80

Further, delete the items 5, 12, 8, and 20 from the B-tree.

- Q9.** Apply Dijkstra's algorithm to find the shortest path from the vertex 'S' to all other vertices for the following graph:



- Q10.** Apply Prim's Algorithm to find the minimum spanning tree for the following graph.



Example of a Graph

- Q11.** Apply Insertion and Selection sorting algorithms to sort the following list of items. So, all the intermediate steps. Also, analyze their best, worst and average case time complexity.

12, 5, 2, 15, 25, 30, 45, 8, 17, 50, 3, 7

Q1. Write a program in C to accepts two polynomials as input and prints the resultant polynomial due to the multiplication of input polynomials.

ANS

C Program:

C code

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

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

```
// Structure to represent a polynomial term
```

```
struct Term {
```

```
    int coeff; // Coefficient of the term
```

```
    int exp;   // Exponent of the term
```

```
};
```

```
// Function to print the polynomial
```

```
void printPolynomial(struct Term poly[], int n) {
```

```
    for (int i = 0; i < n; i++) {
```

```
        printf("%dx^%d", poly[i].coeff, poly[i].exp);
```

```
        if (i != n - 1)
```

```
            printf(" + ");
```

```
    }
```

```
    printf("\n");
```

```
}
```

```
// Function to multiply two polynomials
```

```
void multiplyPolynomials(struct Term poly1[], int n1, struct Term poly2[], int n2, struct Term result[], int *nResult) {
```

```
    // Initialize the result array with 0
```

```
    for (int i = 0; i < n1 * n2; i++) {
```

```

    result[i].coeff = 0;

    result[i].exp = 0;
}

*nResult = 0; // Initial count of terms in result

// Multiply each term of poly1 with each term of poly2
for (int i = 0; i < n1; i++) {
    for (int j = 0; j < n2; j++) {
        result[*nResult].coeff = poly1[i].coeff * poly2[j].coeff;
        result[*nResult].exp = poly1[i].exp + poly2[j].exp;
        (*nResult)++;
    }
}

// Combine terms with the same exponent
for (int i = 0; i < *nResult; i++) {
    for (int j = i + 1; j < *nResult; j++) {
        if (result[i].exp == result[j].exp) {
            result[i].coeff += result[j].coeff;

            // Remove the duplicate term
            for (int k = j; k < *nResult - 1; k++) {
                result[k] = result[k + 1];
            }

            (*nResult)--; // Reduce the number of terms
            j--; // Re-check the current position
        }
    }
}

```

```
}  
}
```

```
int main() {  
    int n1, n2, nResult;  
  
    // Input first polynomial  
    printf("Enter the number of terms in the first polynomial: ");  
    scanf("%d", &n1);  
    struct Term poly1[n1];  
  
    printf("Enter the terms of the first polynomial (coeff exponent): \n");  
    for (int i = 0; i < n1; i++) {  
        scanf("%d%d", &poly1[i].coeff, &poly1[i].exp);  
    }  
  
    // Input second polynomial  
    printf("Enter the number of terms in the second polynomial: ");  
    scanf("%d", &n2);  
    struct Term poly2[n2];  
  
    printf("Enter the terms of the second polynomial (coeff exponent): \n");  
    for (int i = 0; i < n2; i++) {  
        scanf("%d%d", &poly2[i].coeff, &poly2[i].exp);  
    }  
  
    // Resultant polynomial can have at most  $n1 * n2$  terms  
    struct Term result[n1 * n2];
```

```

// Multiply the polynomials

multiplyPolynomials(poly1, n1, poly2, n2, result, &nResult);

// Print the resultant polynomial

printf("The resultant polynomial after multiplication is: \n");

printPolynomial(result, nResult);

return 0;
}

```

Explanation:

- The program uses a struct Term to represent a term of a polynomial.
- Two polynomials are taken as input, and their multiplication is performed term by term.
- After multiplying, the program combines like terms (terms with the same exponent).
- Finally, the resultant polynomial is printed.

Sample Input and Output:

```

PS D:\.vscode> cd "d:\.vscode\c language\c++\" ; if ($?) { g++ BCSL-032_Q2.cpp -o BCSL-032_Q2 } ; if ($?) { .\BCSL-032_Q2 }
Enter the number of terms in the first polynomial: 2
Enter the terms of the first polynomial (coeff exponent):
3 2
5 1
Enter the number of terms in the second polynomial: 2
Enter the terms of the second polynomial (coeff exponent):
4 1
2 0
The resultant polynomial after multiplication is:
12x^3 + 26x^2 + 10x^1
PS D:\.vscode\c language\c++> 

```

Q2. Write a program in 'C' to create a single linked list and perform the following operations on it:

- Insert a new node at the beginning, in the middle or at the end of the linked list.**
- Delete a node from the linked list**

- (iii) **Display the linked list in reverse order**
- (iv) **Sort and display data of the linked list in ascending order.**
- (v) **Count the number of items stored in a single linked list.**

Ans

C Program: code

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

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

```
// Structure for a node
```

```
struct Node {
```

```
    int data;
```

```
    struct Node* next;
```

```
};
```

```
// Function to create a new node
```

```
struct Node* createNode(int data) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = data;
```

```
    newNode->next = NULL;
```

```
    return newNode;
```

```
}
```

```
// Function to insert a node at the beginning
```

```
void insertAtBeginning(struct Node** head, int data) {
```

```
    struct Node* newNode = createNode(data);
```

```
    newNode->next = *head;
```

```
    *head = newNode;
```

```
}
```

```

// Function to insert a node at the end

void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

```

```

// Function to insert a node in the middle

void insertInMiddle(struct Node** head, int data, int pos) {
    struct Node* newNode = createNode(data);
    if (pos == 1) {
        newNode->next = *head;
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    for (int i = 1; i < pos - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {

```



```
    printf("Position out of bounds\n");  
    return;  
}  
newNode->next = temp->next;  
temp->next = newNode;  
}
```

// Function to delete a node

```
void deleteNode(struct Node** head, int key) {  
    struct Node* temp = *head;  
    struct Node* prev = NULL;  
  
    // If the head node itself holds the key  
    if (temp != NULL && temp->data == key) {  
        *head = temp->next; // Changed head  
        free(temp); // Free old head  
        return;  
    }
```

// Search for the key to be deleted

```
while (temp != NULL && temp->data != key) {  
    prev = temp;  
    temp = temp->next;  
}
```

// If the key was not present

```
if (temp == NULL) return;
```

```

// Unlink the node

prev->next = temp->next;


free(temp); // Free the memory of the deleted node
}


// Function to display the linked list
void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}


// Helper function to display the list in reverse (recursive)
void displayReverseHelper(struct Node* head) {
    if (head == NULL)
        return;
    displayReverseHelper(head->next);
    printf("%d -> ", head->data);
}


// Function to display the list in reverse
void displayReverse(struct Node* head) {
    displayReverseHelper(head);
    printf("NULL\n");
}

```

```
}
```

```
// Function to sort the linked list
```

```
void sortList(struct Node* head) {
```

```
    struct Node* i = head;
```

```
    struct Node* j = NULL;
```

```
    int temp;
```

```
    if (head == NULL) {
```

```
        return;
```

```
    }
```

```
// Bubble sort algorithm
```

```
for (i = head; i != NULL; i = i->next) {
```

```
    for (j = i->next; j != NULL; j = j->next) {
```

```
        if (i->data > j->data) {
```

```
            temp = i->data;
```

```
            i->data = j->data;
```

```
            j->data = temp;
```

```
        }
```

```
    }
```

```
}
```

```
}
```

```
// Function to count the number of nodes
```

```
int countNodes(struct Node* head) {
```

```
    int count = 0;
```

```
    struct Node* temp = head;
```

```
while (temp != NULL) {  
    count++;  
    temp = temp->next;  
}  
return count;  
}
```

// Main function to demonstrate the operations

```
int main() {  
    struct Node* head = NULL;  
    int choice, value, pos;  
  
    while (1) {  
        printf("\nMenu:\n");  
        printf("1. Insert at beginning\n");  
        printf("2. Insert at end\n");  
        printf("3. Insert in middle\n");  
        printf("4. Delete node\n");  
        printf("5. Display list\n");  
        printf("6. Display list in reverse\n");  
        printf("7. Sort list\n");  
        printf("8. Count nodes\n");  
        printf("9. Exit\n");  
        printf("Enter your choice: ");  
        scanf("%d", &choice);  
  
        switch (choice) {  
            case 1:
```

```
printf("Enter value to insert at beginning: ");  
scanf("%d", &value);  
insertAtBeginning(&head, value);  
break;
```

case 2:

```
printf("Enter value to insert at end: ");  
scanf("%d", &value);  
insertAtEnd(&head, value);  
break;
```

case 3:

```
printf("Enter value to insert in middle: ");  
scanf("%d", &value);  
printf("Enter position to insert: ");  
scanf("%d", &pos);  
insertInMiddle(&head, value, pos);  
break;
```

case 4:

```
printf("Enter value to delete: ");  
scanf("%d", &value);  
deleteNode(&head, value);  
break;
```

case 5:

```
printf("Linked list: ");  
displayList(head);  
break;
```

case 6:

```
printf("Linked list in reverse: ");  
displayReverse(head);
```

```

        break;
    case 7:
        sortList(head);
        printf("Sorted linked list: ");
        displayList(head);
        break;
    case 8:
        printf("Number of nodes: %d\n", countNodes(head));
        break;
    case 9:
        exit(0);
    default:
        printf("Invalid choice!\n");
    }
}
return 0;
}

```

Explanation:

Insertion: You can insert a node at the beginning, middle, or end of the list.

Deletion: Allows deletion of a node based on the key (value of the node).

Display: Displays the list in normal order and reverse order (using recursion).

Sorting: Uses bubble sort to sort the list in ascending order.

Count: Counts the number of nodes in the list.

Sample Run:

```

PS D:\.vscode> cd "d:\.vscode\c language\c++\" ; if ($?) { g++ BCSL-032_Q2.cpp -o BCSL-032_Q2 } ; if ($?) { .\BCSL-032_Q2 }

Menu:
1. Insert at beginning
2. Insert at end
3. Insert in middle
4. Delete node
5. Display list
6. Display list in reverse
7. Sort list
8. Count nodes
9. Exit
Enter your choice: 1
Enter value to insert at beginning: 10

Menu:
1. Insert at beginning
2. Insert at end
3. Insert in middle
4. Delete node
5. Display list
6. Display list in reverse
7. Sort list
8. Count nodes
9. Exit
Enter your choice: 5
Linked list: 10 -> NULL

```

Q3. Write a program in ‘C’ to create a doubly linked list to store integer values and perform the following operations on it:

- (i) Insert a new node at the beginning, in the middle or at the end of the linked list.**
- (ii) Delete a node from the linked list**
- (iii) Sort and display data of the doubly linked list in ascending order.**
- (iv) Count the number of items stored in a single linked list**
- (v) Calculate the sum of all even integer numbers, stored in the doubly linked list.**

ANS

C Program:

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

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

```
// Structure for a doubly linked list node
```

```
struct Node {
```

```
    int data;
```

```
    struct Node* next;
```

```
    struct Node* prev;  
};
```

```
// Function to create a new node
```

```
struct Node* createNode(int data) {  
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));  
    newNode->data = data;  
    newNode->next = NULL;  
    newNode->prev = NULL;  
    return newNode;  
}
```

```
// Function to insert a node at the beginning
```

```
void insertAtBeginning(struct Node** head, int data) {  
    struct Node* newNode = createNode(data);  
    if (*head != NULL) {  
        newNode->next = *head;  
        (*head)->prev = newNode;  
    }  
    *head = newNode;  
}
```

```
// Function to insert a node at the end
```

```
void insertAtEnd(struct Node** head, int data) {  
    struct Node* newNode = createNode(data);  
    if (*head == NULL) {  
        *head = newNode;  
        return;  
    }
```



```

}

struct Node* temp = *head;
while (temp->next != NULL) {
    temp = temp->next;
}

temp->next = newNode;
newNode->prev = temp;
}

```

// Function to insert a node in the middle

```

void insertInMiddle(struct Node** head, int data, int pos) {
    struct Node* newNode = createNode(data);
    if (pos == 1) {
        newNode->next = *head;
        if (*head != NULL)
            (*head)->prev = newNode;
        *head = newNode;
        return;
    }

```

```

    struct Node* temp = *head;
    for (int i = 1; i < pos - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

```

```

    if (temp == NULL) {
        printf("Position out of bounds\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
```

```
void deleteNode(struct Node** head, int key) {  
    struct Node* temp = *head;  
  
    // If the head node holds the key  
    if (temp != NULL && temp->data == key) {  
        *head = temp->next;  
        if (*head != NULL)  
            (*head)->prev = NULL;  
        free(temp);  
        return;  
    }  
}
```

```
// Search for the node with the key
```

```
while (temp != NULL && temp->data != key) {  
    temp = temp->next;  
}
```

```
// If the key is not found
```

```

if (temp == NULL) {
    printf("Node not found!\n");
    return;
}

// Unlink the node
if (temp->next != NULL)
    temp->next->prev = temp->prev;
if (temp->prev != NULL)
    temp->prev->next = temp->next;

free(temp);
}

// Function to display the doubly linked list
void displayList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Function to sort the doubly linked list
void sortList(struct Node* head) {
    struct Node* i = head;
    struct Node* j = NULL;

```

```
int temp;
```

```
if (head == NULL)
```

```
    return;
```

```
// Bubble sort algorithm
```

```
for (i = head; i->next != NULL; i = i->next) {
```

```
    for (j = i->next; j != NULL; j = j->next) {
```

```
        if (i->data > j->data) {
```

```
            temp = i->data;
```

```
            i->data = j->data;
```

```
            j->data = temp;
```

```
        }
```

```
    }
```

```
}
```

```
}
```

```
// Function to count the number of nodes
```

```
int countNodes(struct Node* head) {
```

```
    int count = 0;
```

```
    struct Node* temp = head;
```

```
    while (temp != NULL) {
```

```
        count++;
```

```
        temp = temp->next;
```

```
    }
```

```
    return count;
```

```
}
```

```
// Function to calculate the sum of even numbers in the list
```

```
int sumEvenNumbers(struct Node* head) {  
    int sum = 0;  
    struct Node* temp = head;  
    while (temp != NULL) {  
        if (temp->data % 2 == 0) {  
            sum += temp->data;  
        }  
        temp = temp->next;  
    }  
    return sum;  
}
```

```
// Main function to demonstrate the operations
```

```
int main() {  
    struct Node* head = NULL;  
    int choice, value, pos;  
  
    while (1) {  
        printf("\nMenu:\n");  
        printf("1. Insert at beginning\n");  
        printf("2. Insert at end\n");  
        printf("3. Insert in middle\n");  
        printf("4. Delete node\n");  
        printf("5. Display list\n");  
        printf("6. Sort list\n");  
        printf("7. Count nodes\n");  
        printf("8. Sum of even numbers\n");
```

```
printf("9. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

    case 1:

        printf("Enter value to insert at beginning: ");

        scanf("%d", &value);

        insertAtBeginning(&head, value);

        break;

    case 2:

        printf("Enter value to insert at end: ");

        scanf("%d", &value);

        insertAtEnd(&head, value);

        break;

    case 3:

        printf("Enter value to insert in middle: ");

        scanf("%d", &value);

        printf("Enter position to insert: ");

        scanf("%d", &pos);

        insertInMiddle(&head, value, pos);

        break;

    case 4:

        printf("Enter value to delete: ");

        scanf("%d", &value);

        deleteNode(&head, value);

        break;

    case 5:
```

```

        printf("Doubly linked list: ");
        displayList(head);
        break;
    case 6:
        sortList(head);
        printf("Sorted doubly linked list: ");
        displayList(head);
        break;
    case 7:
        printf("Number of nodes: %d\n", countNodes(head));
        break;
    case 8:
        printf("Sum of even numbers: %d\n", sumEvenNumbers(head));
        break;
    case 9:
        exit(0);
    default:
        printf("Invalid choice!\n");
    }
}

return 0;
}

```

Explanation:

- **Insertions:** Insert a node at the beginning, middle, or end of the list.
- **Deletion:** Delete a node by value.
- **Sorting:** Uses bubble sort to sort the list in ascending order.
- **Counting Nodes:** Count the number of nodes in the list.
- **Sum of Even Numbers:** Calculate the sum of all even integers in the list.

Sample Run:

```
PS D:\vscode> cd "d:\vscode\c language\c++\" ; if ($?) { g++ BCSL-032_Q2.cpp -o BCSL-032_Q2 } ; if ($?) { .\BCSL-032_Q2 }

Menu:
1. Insert at beginning
2. Insert at end
3. Insert in middle
4. Delete node
5. Display list
6. Display list in reverse
7. Sort list
8. Count nodes
9. Exit
Enter your choice: 1
Enter value to insert at beginning: 12

Menu:
1. Insert at beginning
2. Insert at end
3. Insert in middle
4. Delete node
5. Display list
6. Display list in reverse
7. Sort list
8. Count nodes
9. Exit
Enter your choice: 5
Linked list: 12 -> NULL
```

Q4. What is a Dequeue? Write algorithm to perform insert and delete operations in a Dequeue.

Ans

What is a Dequeue?

A Dequeue (pronounced "deck") or Double-ended Queue is a linear data structure that allows insertion and deletion of elements from both ends: the front and the rear. Unlike a standard queue (FIFO - First In, First Out), a dequeue provides more flexibility as elements can be added or removed from either end.

Key Characteristics:

Insertions and Deletions: Can be performed from both the front and the rear.

Dynamic: Similar to queues, dequeues are dynamic and can grow or shrink in size as required.

Types:

Input-restricted deque: Allows insertion only at one end but deletion at both ends.

Output-restricted deque: Allows deletion at one end but insertion at both ends.

Basic Operations in a Dequeue:

Insert at the front.

Insert at the rear.

Delete from the front.

Delete from the rear.

Algorithms to Perform Insert and Delete Operations in a Dequeue

Assuming that a dequeue is implemented using an array (circular or not), the operations are performed based on two pointers: front and rear.

1. Insert at the Front:

Algorithm:

Check if the dequeue is full (circular array implementation).

If full, report overflow and exit.

Otherwise, adjust the front pointer:

If front is at the first position, wrap around to the last index of the array.

Otherwise, decrement front.

Insert the element at the front.

Algorithm:

arduino

Copy code

Algorithm InsertAtFront(dequeue, item)

1. If ((front == 0 and rear == size - 1) or (rear == front - 1)):

 Output "Overflow"

 Return

2. If (front == -1):

 front = rear = 0

3. Else If (front == 0):

 front = size - 1

4. Else:

front = front - 1

5. dequeue[front] = item

2. Insert at the Rear:

Algorithm:

Check if the dequeue is full.

If full, report overflow and exit.

Otherwise, adjust the rear pointer:

If rear is at the last index, wrap around to the first index of the array.

Otherwise, increment rear.

Insert the element at the rear.

Algorithm:

arduino

Copy code

Algorithm InsertAtRear(dequeue, item)

1. If ((front == 0 and rear == size - 1) or (rear == front - 1)):

Output "Overflow"

Return

2. If (rear == -1):

front = rear = 0

3. Else If (rear == size - 1):

rear = 0

4. Else:

rear = rear + 1

5. dequeue[rear] = item

3. Delete from the Front:

Algorithm:

Check if the dequeue is empty.

If empty, report underflow and exit.

Otherwise, retrieve the element from front.

Adjust the front pointer:

If $\text{front} == \text{rear}$, the dequeue has only one element. Set front and rear to -1 (empty condition).

If front is at the last index, wrap around to the first index.

Otherwise, increment front.

Algorithm:

mathematica

Copy code

Algorithm DeleteFromFront(dequeue)

1. If ($\text{front} == -1$):

 Output "Underflow"

 Return

2. Output $\text{dequeue}[\text{front}]$

3. If ($\text{front} == \text{rear}$):

$\text{front} = \text{rear} = -1$

4. Else If ($\text{front} == \text{size} - 1$):

$\text{front} = 0$

5. Else:

$\text{front} = \text{front} + 1$

4. Delete from the Rear:

Algorithm:

Check if the dequeue is empty.

If empty, report underflow and exit.

Otherwise, retrieve the element from rear.

Adjust the rear pointer:

If $\text{front} == \text{rear}$, the dequeue has only one element. Set front and rear to -1 (empty condition).

If rear is at the first index, wrap around to the last index.

Otherwise, decrement rear.

Algorithm:

mathematica

Copy code

Algorithm DeleteFromRear(dequeue)

1. If ($\text{front} == -1$):

 Output "Underflow"

 Return

2. Output dequeue[rear]

3. If ($\text{front} == \text{rear}$):

$\text{front} = \text{rear} = -1$

4. Else If ($\text{rear} == 0$):

$\text{rear} = \text{size} - 1$

5. Else:

$\text{rear} = \text{rear} - 1$

Dequeue Implementation Example (Circular Array-based):

c

Copy code

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

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

```
#define MAX 5
```

```
struct Dequeue {  
    int arr[MAX];  
    int front;  
    int rear;  
};
```

```
// Function to initialize the Dequeue
```

```
void initialize(struct Dequeue* dq) {  
    dq->front = -1;  
    dq->rear = -1;  
}
```

```
// Function to check if the Dequeue is full
```

```
int isFull(struct Dequeue* dq) {  
    return (dq->front == 0 && dq->rear == MAX - 1) || (dq->rear == (dq->front - 1) % (MAX - 1));  
}
```

```
// Function to check if the Dequeue is empty
```

```
int isEmpty(struct Dequeue* dq) {  
    return dq->front == -1;  
}
```

```
// Function to insert an element at the front
```

```
void insertFront(struct Dequeue* dq, int item) {  
    if (isFull(dq)) {
```

```

    printf("Dequeue is full!\n");
    return;
}
if (dq->front == -1) { // First element insertion
    dq->front = dq->rear = 0;
} else if (dq->front == 0) {
    dq->front = MAX - 1;
} else {
    dq->front--;
}
dq->arr[dq->front] = item;
}

```

```

// Function to insert an element at the rear
void insertRear(struct Dequeue* dq, int item) {
    if (isFull(dq)) {
        printf("Dequeue is full!\n");
        return;
    }
    if (dq->rear == -1) { // First element insertion
        dq->front = dq->rear = 0;
    } else if (dq->rear == MAX - 1) {
        dq->rear = 0;
    } else {
        dq->rear++;
    }
    dq->arr[dq->rear] = item;
}

```

// Function to delete an element from the front

```
void deleteFront(struct Dequeue* dq) {  
    if (isEmpty(dq)) {  
        printf("Dequeue is empty!\n");  
        return;  
    }  
    printf("Deleted from front: %d\n", dq->arr[dq->front]);  
    if (dq->front == dq->rear) { // Only one element  
        dq->front = dq->rear = -1;  
    } else if (dq->front == MAX - 1) {  
        dq->front = 0;  
    } else {  
        dq->front++;  
    }  
}
```

// Function to delete an element from the rear

```
void deleteRear(struct Dequeue* dq) {  
    if (isEmpty(dq)) {  
        printf("Dequeue is empty!\n");  
        return;  
    }  
    printf("Deleted from rear: %d\n", dq->arr[dq->rear]);  
    if (dq->front == dq->rear) { // Only one element  
        dq->front = dq->rear = -1;  
    } else if (dq->rear == 0) {  
        dq->rear = MAX - 1;  
    }  
}
```

```
    } else {  
        dq->rear--;  
    }  
}
```

// Function to display the Dequeue

```
void display(struct Dequeue* dq) {  
    if (isEmpty(dq)) {  
        printf("Dequeue is empty!\n");  
        return;  
    }  
    printf("Dequeue elements: ");  
    int i = dq->front;  
    while (1) {  
        printf("%d ", dq->arr[i]);  
        if (i == dq->rear) break;  
        i = (i + 1) % MAX;  
    }  
    printf("\n");  
}
```

// Main function

```
int main() {  
    struct Dequeue dq;  
    initialize(&dq);  
  
    insertRear(&dq, 5);  
    insertRear(&dq, 10);
```



```

insertFront(&dq, 15);

display(&dq);


deleteFront(&dq);

display(&dq);


deleteRear(&dq);

display(&dq);


return 0;
}

```

Output:

```

PS D:\.vscode> cd "d:\.vscode\c language\c++\" ; if ($?) { g++ BCSL-032_Q2.cpp -o BCSL-032_Q2 } ; if ($?) { .\BCSL-032_Q2 }
Dequeue elements: 15 5 10
Deleted from front: 15
Dequeue elements: 5 10
Deleted from rear: 10
Dequeue elements: 5
PS D:\.vscode\c language\c++>

```

In this example, we implemented a circular array-based deque and performed various operations such as insertion and deletion from both ends.

Q5. Draw the binary tree for which the traversal sequences are given as follows:

- (i) **Pre order: A B D E F C G H I J K In order: B E D F A C I H K J G**
(ii) **Post order: I J H D K E C L M G F B A In order: I H J D C K E A F L G M B**

Ans

(i) Pre-order: A B D E F C G H I J K

In-order: B E D F A C I H K J G

Steps to Construct the Tree:

- Pre-order** traversal gives the root of the tree as the first element.
 - The root is A.
- Find the root A in the **In-order** sequence.

- In-order sequence: B E D F [A] C I H K J G
- Elements to the left of A (B E D F) are part of the left subtree.
- Elements to the right of A (C I H K J G) are part of the right subtree.

3. Recursively apply the same logic for left and right subtrees.

Left Subtree of A:

- Pre-order: B D E F
- In-order: B E D F
 - The root of the left subtree is B.
 - In the In-order sequence, elements to the right of B (E D F) are in the right subtree of B.

Right Subtree of B:

- Pre-order: D E F
- In-order: E D F
 - The root is D, and E is in the left subtree of D, while F is in the right subtree.

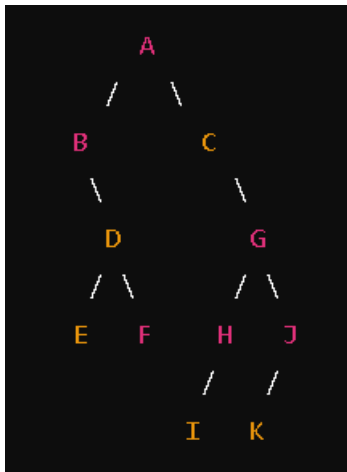
Right Subtree of A:

- Pre-order: C G H I J K
- In-order: C I H K J G
 - The root is C.
 - In the In-order sequence, elements to the right of C (I H K J G) are in the right subtree of C.

Right Subtree of C:

- Pre-order: G H I J K
- In-order: I H K J G
 - The root is G.
 - I H forms the left subtree of G and K J forms the right subtree.

The binary tree for the first case can be constructed as:



(ii) Post-order: I J H D K E C L M G F B A

In-order: I H J D C K E A F L G M B

Steps to Construct the Tree:

1. **Post-order** traversal gives the root of the tree as the last element.
 - The root is A.
2. Find the root A in the **In-order** sequence.
 - In-order sequence: I H J D C K E [A] F L G M B
 - Elements to the left of A (I H J D C K E) are part of the left subtree.
 - Elements to the right of A (F L G M B) are part of the right subtree.
3. Recursively apply the same logic for left and right subtrees.

Left Subtree of A:

- Post-order: I J H D K E C
- In-order: I H J D C K E
 - The root of the left subtree is C (last element in Post-order).
 - Elements to the left of C (I H J D) are part of the left subtree of C.
 - Elements to the right of C (K E) are part of the right subtree of C.

Left Subtree of C:

- Post-order: I J H D

- In-order: I H J D
 - The root is D.
 - I H forms the left subtree of D, and J is the right child of D.

Right Subtree of C:

- Post-order: K E
- In-order: K E
 - The root is E, and K is the left child of E.

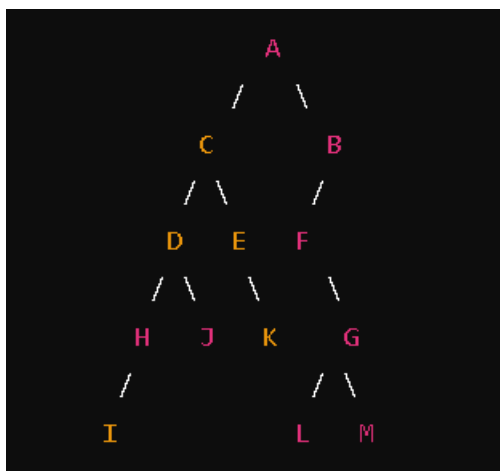
Right Subtree of A:

- Post-order: L M G F B
- In-order: F L G M B
 - The root is B (last element in Post-order).
 - Elements to the left of B (F L G M) are part of the left subtree of B.

Left Subtree of B:

- Post-order: L M G F
- In-order: F L G M
 - The root is F.
 - L G M forms the right subtree of F, with G as the root of that subtree.

The binary tree for the second case can be constructed as:



Q6. Write a program in 'C' to implement a binary search tree (BST). Traverse and display the binary search tree in the Inorder, Preorder and Post order form.

Ans

C Program to Implement a Binary Search Tree (BST) and Traversals

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

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

```
// Define the structure of the binary search tree node
```

```
struct Node {
```

```
    int data;
```

```
    struct Node* left;
```

```
    struct Node* right;
```

```
};
```

```
// Function to create a new node in the BST
```

```
struct Node* createNode(int value) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    newNode->data = value;
```

```
    newNode->left = NULL;
```

```
    newNode->right = NULL;
```

```
    return newNode;
```

```
}
```

```
// Function to insert a node in the BST
```

```
struct Node* insert(struct Node* root, int value) {
```

```
    if (root == NULL) {
```

```
        return createNode(value);
```

```
    }
```

```
if (value < root->data) {  
    root->left = insert(root->left, value);  
} else if (value > root->data) {  
    root->right = insert(root->right, value);  
}  
return root;  
}
```

// In-order Traversal (Left, Root, Right)

```
void inorderTraversal(struct Node* root) {  
    if (root == NULL) return;  
    inorderTraversal(root->left);  
    printf("%d ", root->data);  
    inorderTraversal(root->right);  
}
```

// Pre-order Traversal (Root, Left, Right)

```
void preorderTraversal(struct Node* root) {  
    if (root == NULL) return;  
    printf("%d ", root->data);  
    preorderTraversal(root->left);  
    preorderTraversal(root->right);  
}
```

// Post-order Traversal (Left, Right, Root)

```
void postorderTraversal(struct Node* root) {  
    if (root == NULL) return;  
    postorderTraversal(root->left);
```

```
    postorderTraversal(root->right);  
    printf("%d ", root->data);  
}
```

```
// Main function
```

```
int main() {  
    struct Node* root = NULL;  
    int choice, value;  
  
    while (1) {  
        printf("\n\nBinary Search Tree Operations:\n");  
        printf("1. Insert a node\n");  
        printf("2. In-order Traversal\n");  
        printf("3. Pre-order Traversal\n");  
        printf("4. Post-order Traversal\n");  
        printf("5. Exit\n");  
        printf("Enter your choice: ");  
        scanf("%d", &choice);  
  
        switch (choice) {  
            case 1:  
                printf("Enter the value to insert: ");  
                scanf("%d", &value);  
                root = insert(root, value);  
                break;  
  
            case 2:  
                printf("In-order Traversal: ");
```

```
inorderTraversal(root);
```

```
printf("\n");
```

```
break;
```

```
case 3:
```

```
printf("Pre-order Traversal: ");
```

```
preorderTraversal(root);
```

```
printf("\n");
```

```
break;
```

```
case 4:
```

```
printf("Post-order Traversal: ");
```

```
postorderTraversal(root);
```

```
printf("\n");
```

```
break;
```

```
case 5:
```

```
exit(0);
```

```
default:
```

```
printf("Invalid choice! Please try again.\n");
```

```
}
```

```
}
```

```
return 0;
```

```
}
```

Explanation of the Program:

Node Structure:

A struct Node represents a single node in the BST, containing an integer data, and pointers to the left and right children (left, right).

Creating a Node:

createNode(int value) allocates memory for a new node, assigns the value to it, and initializes its children to NULL.

Insertion in BST:

The insert() function inserts a new value in the correct position in the BST by recursively traversing the tree based on the value:

If the value is smaller than the current node, it goes to the left subtree.

If the value is greater, it goes to the right subtree.

Traversals:

In-order Traversal: Visits the left subtree, root, and then the right subtree (Left -> Root -> Right).

Pre-order Traversal: Visits the root, left subtree, and then the right subtree (Root -> Left -> Right).

Post-order Traversal: Visits the left subtree, right subtree, and then the root (Left -> Right -> Root).

Main Function:

The user is presented with a menu to insert nodes or perform any of the three traversals.

The program continues until the user chooses to exit.

Sample Output:

Binary Search Tree Operations:

1. Insert a node
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Exit

Enter your choice:

1

Enter the value to insert: 50

Binary Search Tree Operations:

1. Insert a node
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Exit

Enter your choice: 1

Enter the value to insert: 70

Binary Search Tree Operations:

1. Insert a node
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Exit

Enter your choice: 1

Enter the value to insert: 30

```
Binary Search Tree Operations:
```

1. Insert a node
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Exit

```
Enter your choice: 2
```

```
In-order Traversal: 30 50 70
```

```
Binary Search Tree Operations:
```

1. Insert a node
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Exit

```
Enter your choice: 4
```

```
Post-order Traversal: 30 70 50
```

```
Binary Search Tree Operations:
```

1. Insert a node
2. In-order Traversal
3. Pre-order Traversal
4. Post-order Traversal
5. Exit

```
Enter your choice: █
```

This program demonstrates how to create a Binary Search Tree (BST) and perform different types of tree traversals.

Q7. Define AVL tree. Create an AVL tree for the following list of data if the data are inserted in the order in an empty AVL tree. 12, 5, 15, 20, 35, 8, 2, 40, 14, 24, 27, 45, 50, 3, 4 Further delete 2, 4, 5 and 12 from the above AVL tree.

Ans

Steps to Create and Modify the AVL Tree

1. Insertion:

- Insert the elements in the given order: 12, 5, 15, 20, 35, 8, 2, 40, 14, 24, 27, 45, 50, 3, 4.
- After each insertion, perform rotations to maintain the balance factor.

2. Deletion:

- Delete the elements 2, 4, 5, and 12.
- After each deletion, perform rotations to maintain the balance factor.

AVL Tree Construction

Date _____

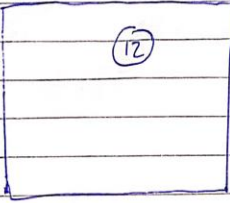
Expt. No. _____ Page No. _____

Q(1)
Ans Balance an AVL tree :-

Give Data :-

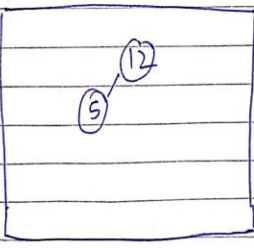
12, 5, 15, 20, 35, 8, 2, 40, 14, 29, 27, 45, 50
3, 4

(1) Insert 12 :-



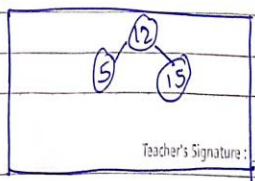
```
graph TD; 12((12))
```

(2) Insert 5 :-



```
graph TD; 12((12)) --- 5((5))
```

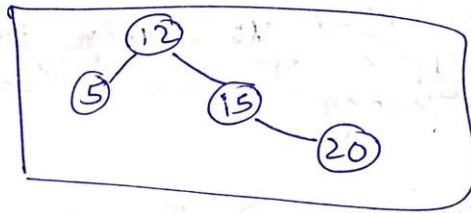
(3) Insert 15 :-



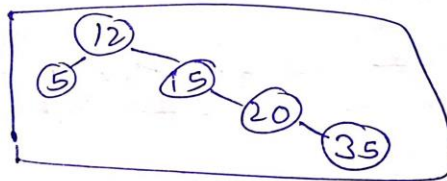
```
graph TD; 12((12)) --- 5((5)); 12 --- 15((15))
```

Teacher's Signature : _____

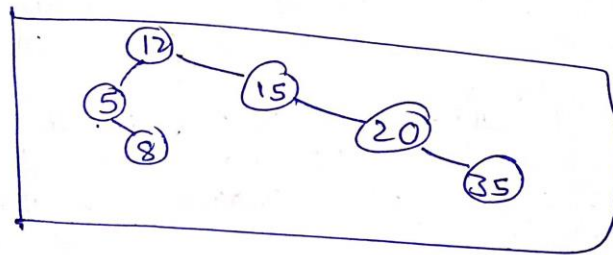
④ Insert 20 :-



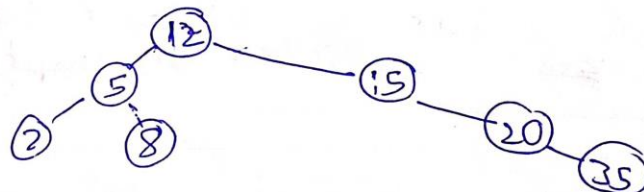
⑤ Insert 35 :-



⑥ Insert 8 :-

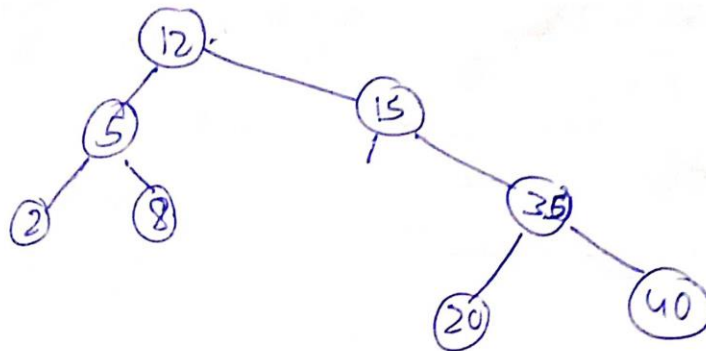


⑦ Insert 2 :-

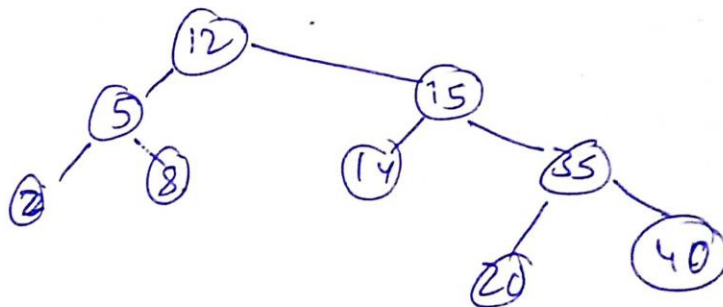


⑧ Insert 40 :-

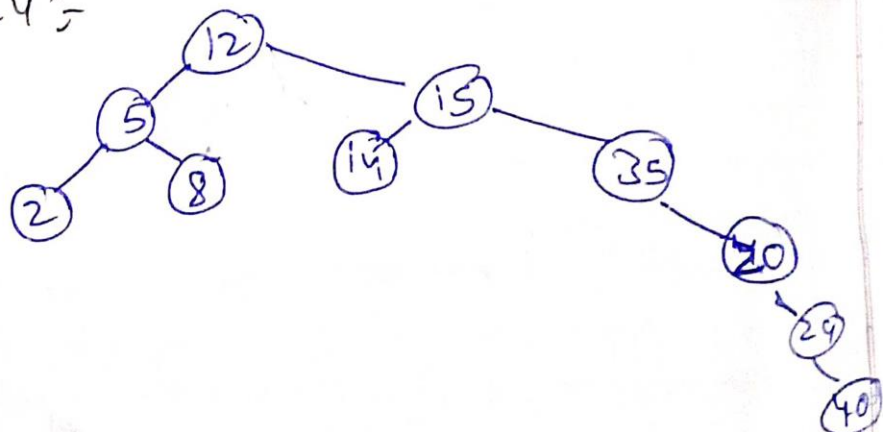
- 40 is inserted to right of 35. Balance factor of 20 becomes -2 (right heavy) Perform a left rotation on 20.



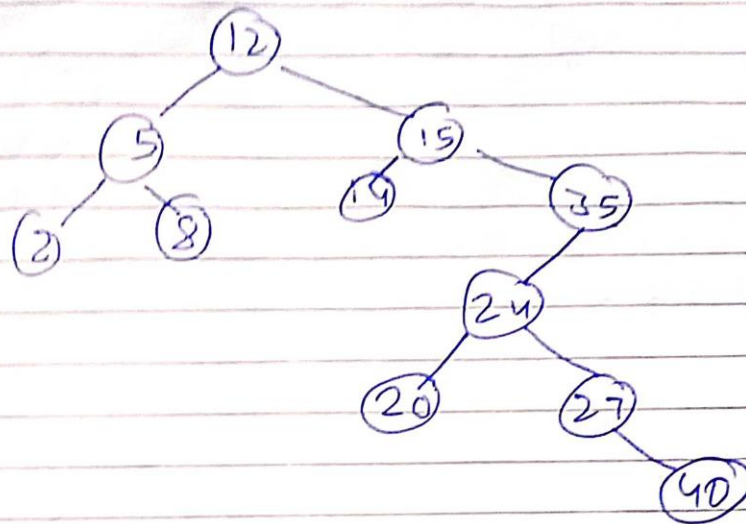
⑨ Insert 14 :-



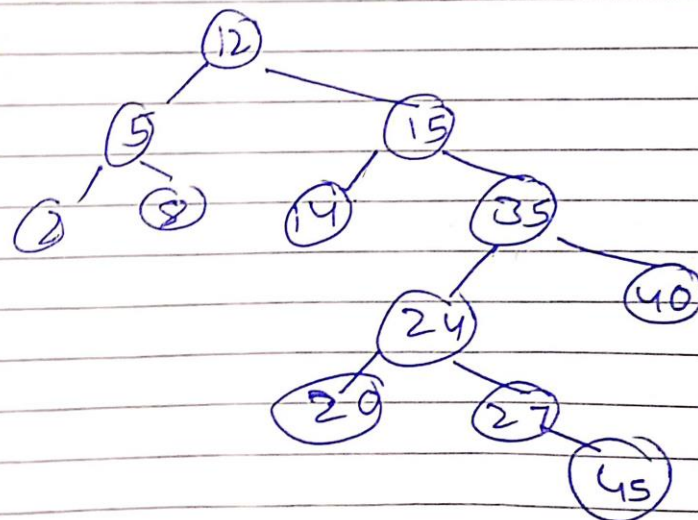
⑩ Insert 24 :-



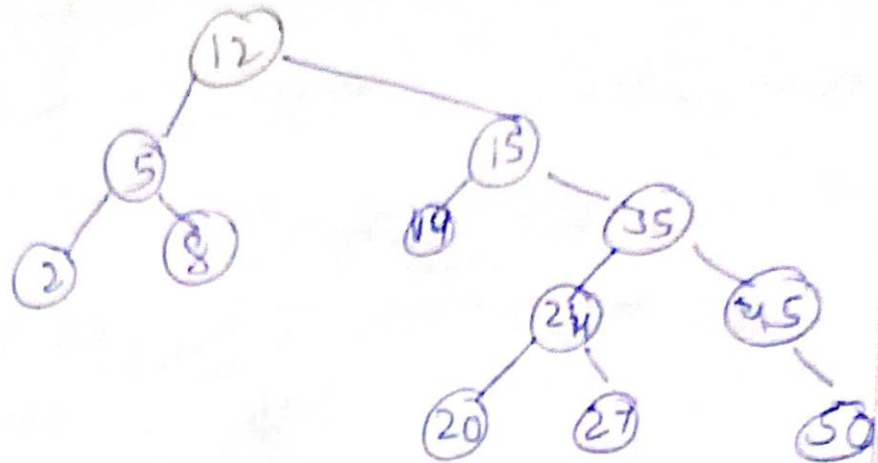
(11) Insert 27



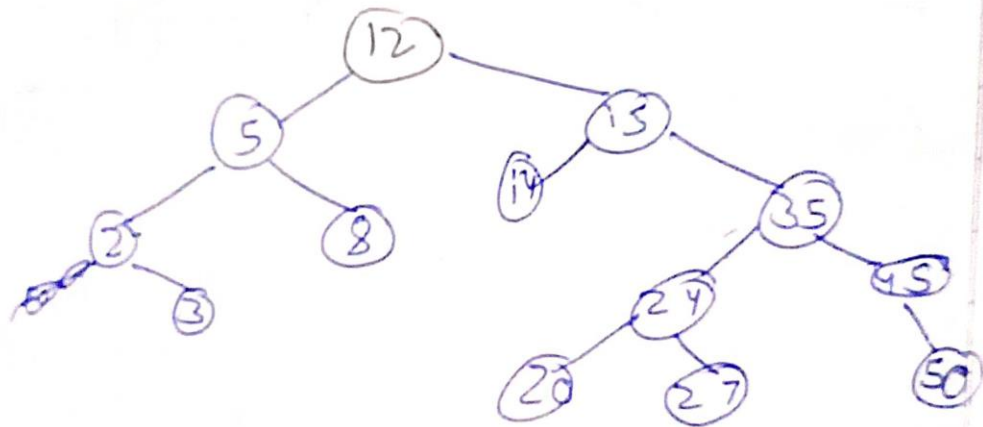
(12) Insert 45 :-



(13) Insert 50 :-

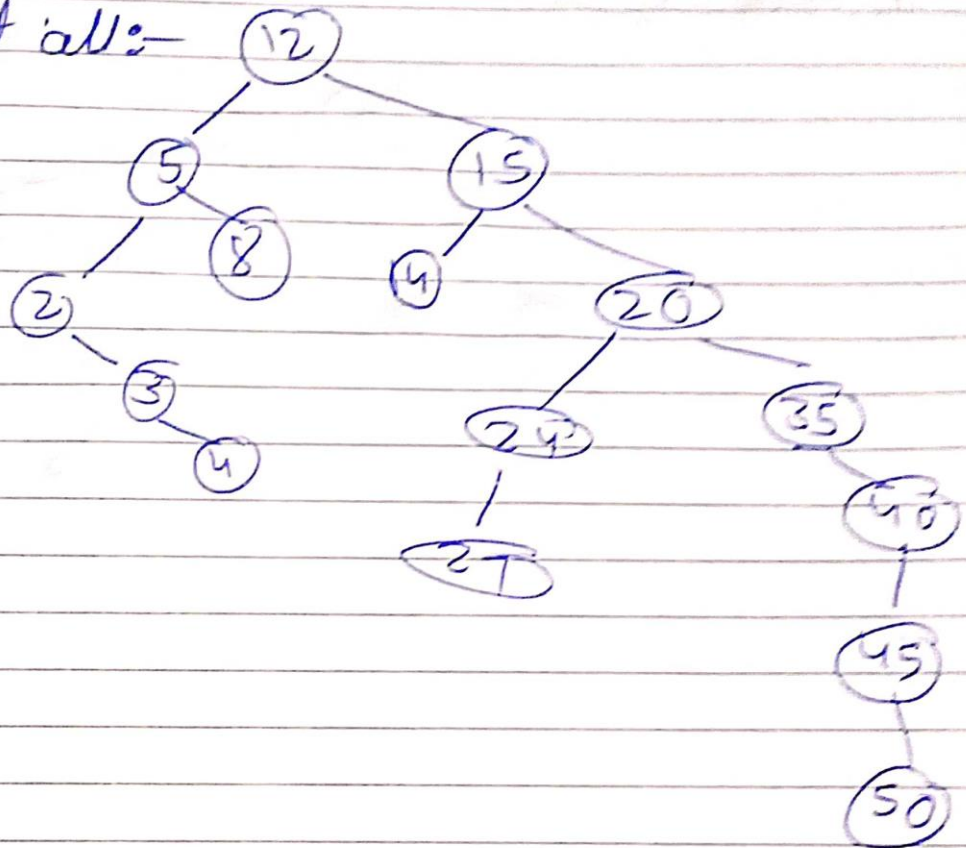


(14) insert 3 :-



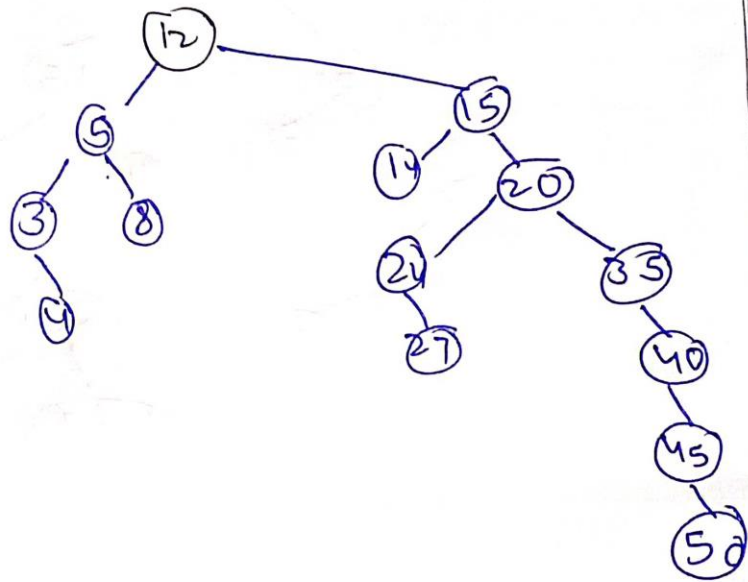
Empu

(15) Insert all:-

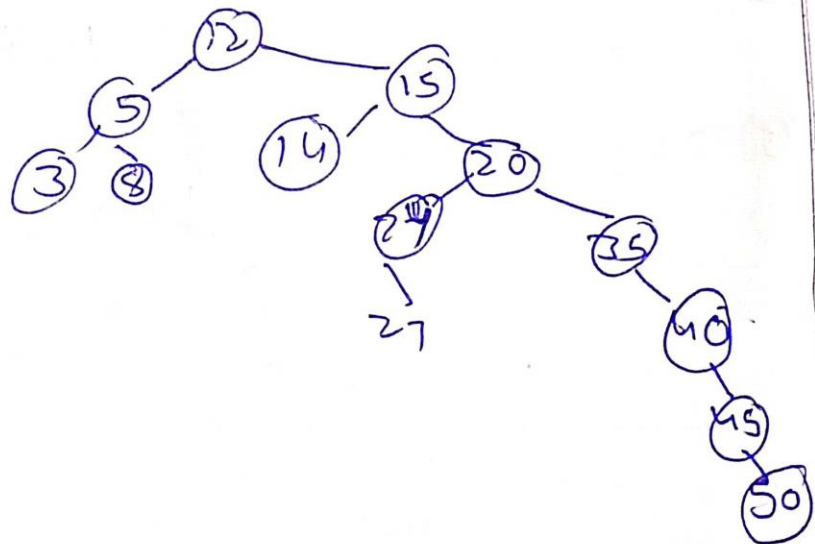


Deleting of Nodes 24, 5 and 12

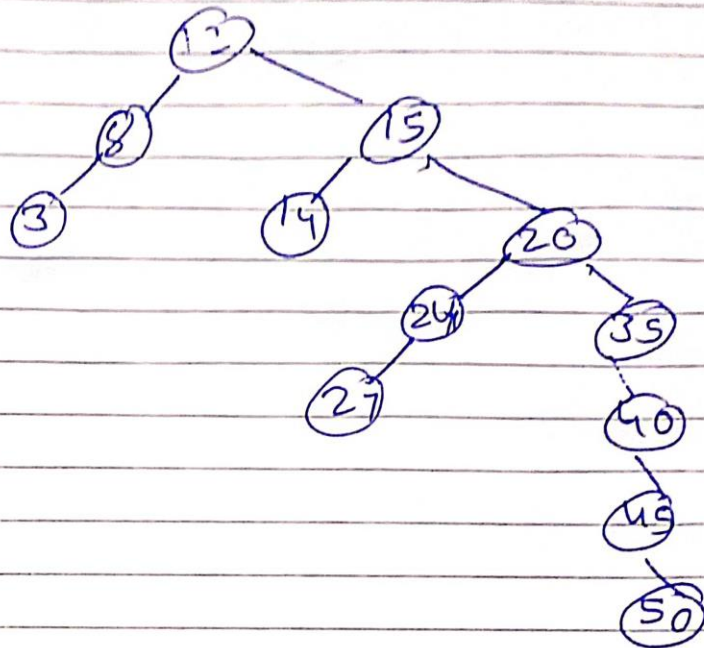
(1) Deleting 2 :-



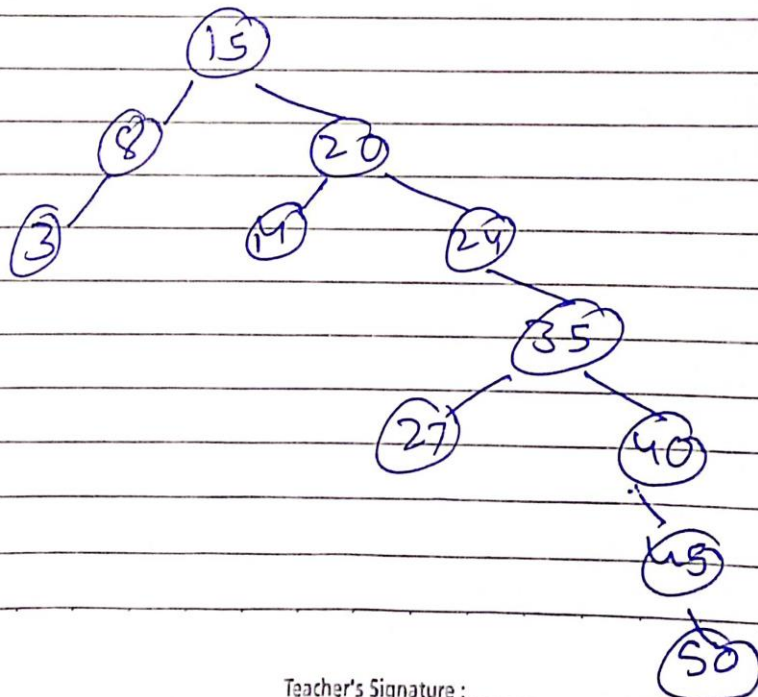
(2) Deleting 4 :-



③ Deleting 5 :-



④ Deleting :-



Teacher's Signature : _____

Q8. Define a B-tree and its properties. Create a B-tree of order-5, if the data items are inserted into an empty B-tree in the following sequence: 12, 5, 15, 20, 60, 45, 35, 40, 25, 8, 7, 55, 50, 66, 65, 80 Further, delete the items 5, 12, 8, and 20 from the B-tree.

Ans

Properties of a B-tree:

1. **Order:** A B-tree of order (m) (also known as a B-tree of degree (m)) has the following properties:
 - Every node has at most (m) children.
 - Every node, except for the root and the leaves, has at least ($\lceil m/2 \rceil$) children.
 - The root has at least two children if it is not a leaf.
 - All leaves appear on the same level.
 - A non-leaf node with (k) children contains ($k-1$) keys.
 - Keys in each node are sorted in increasing order.

Creating a B-tree of Order-5

Let's insert the given sequence of data items into an empty B-tree of order-5:

Sequence: 12, 5, 15, 20, 60, 45, 35, 40, 25, 8, 7, 55, 50, 66, 65, 80.

(1) insert 12 :-

[12]

(2) insert 5 :-

[5, 12]

(3) insert 15 :-

[5, 12, 15]

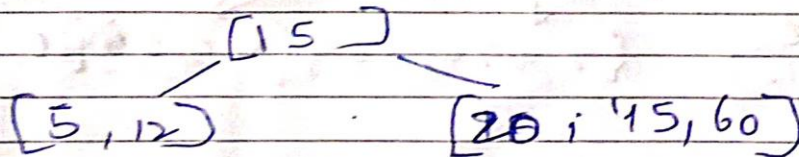
(4) insert 20 :-

[5, 12, 15, 20]

(5) insert 60 :-

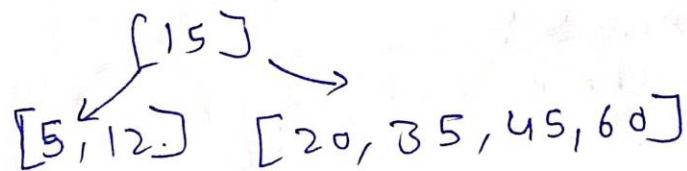
[5, 12, 15, 20, 60]

6 insert 45 (Node splits as it exceeds the order) :-

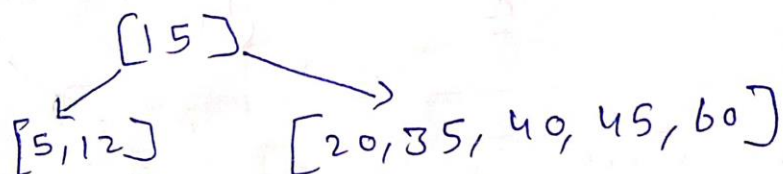


Teacher's Signature : _____

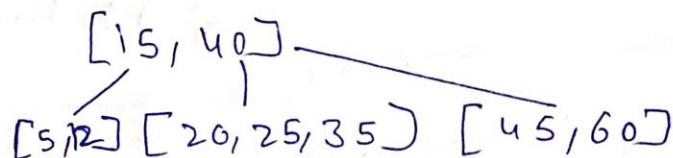
(7) insert 35 :-



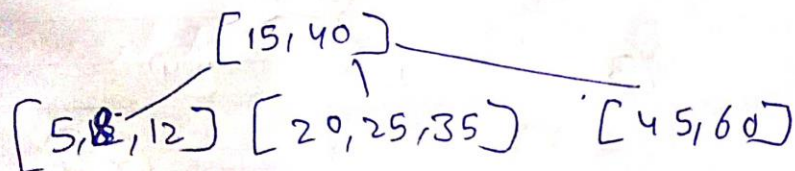
(8) insert 40 :-



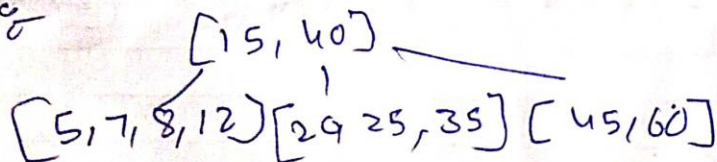
(9) insert 25 (Node splits as it exceeds



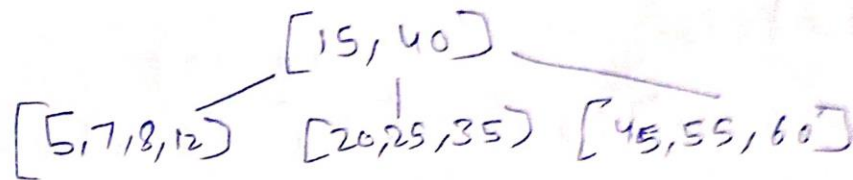
(10) insert 8 :-



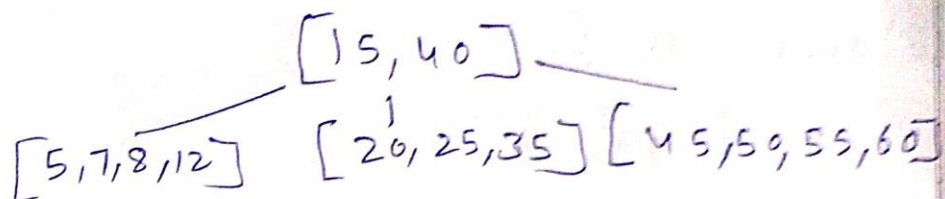
(11) insert 7 :-



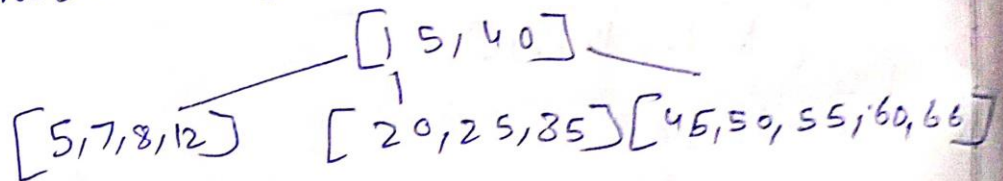
(12) Insert 55 :-



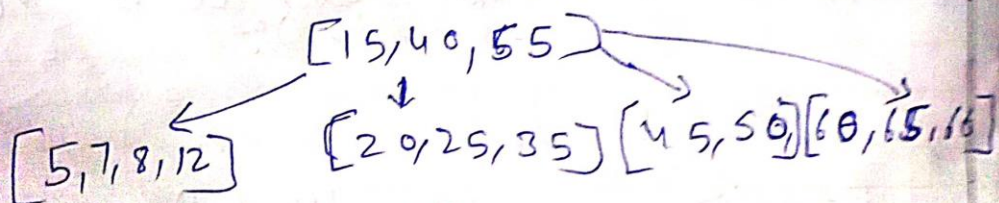
(13) Insert 50 :-



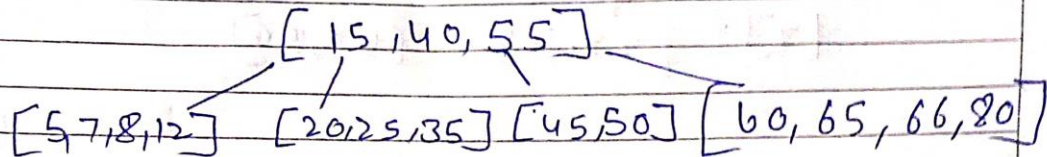
(14) insert 66 :-



(15) insert 65 :- (Node splits as it exceeds the order) :-

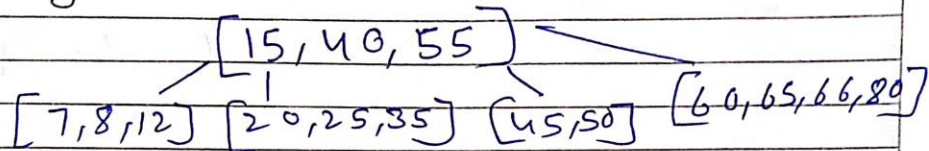


16 insert 80:-

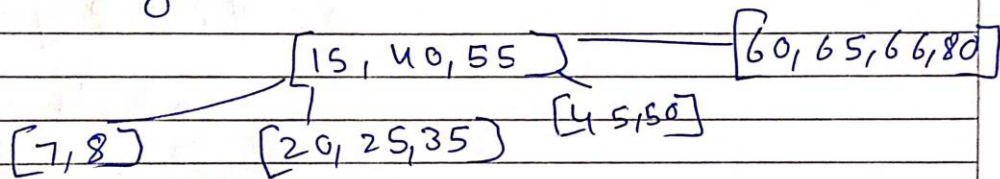


Deleting items from the B-tree:-

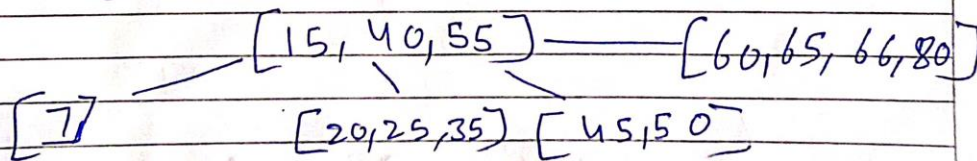
(1) Deleting 5:-



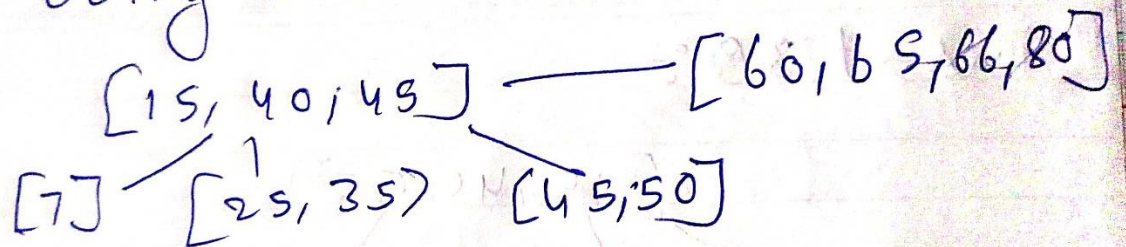
(2) Deleting 12:-



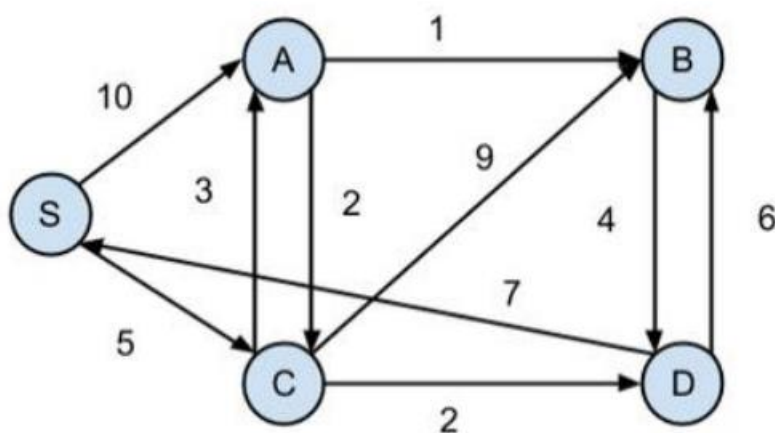
(3) Deleting 8:-



(4) Deleting 200



Q9. Apply Dijkstra's algorithm to find the shortest path from the vertex 'S' to all other vertices for the following graph:



Ans :-

Applying Dijkstra's Algorithm

To find the shortest path from vertex 'S' to all other vertices in the given graph, follow these steps:

1. Initialization:

- Set the distance to the source vertex 'S' as 0.
- Set the distance to all other vertices as infinity.
- Mark all vertices as unvisited.
- Start from the source vertex 'S'.

2. Step-by-Step Calculation:

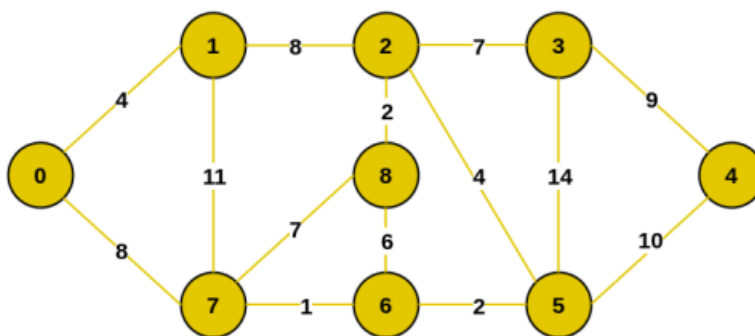
- **Step 1:** Start from vertex 'S'.
 - Distances: $S = 0$, $A = \infty$, $B = \infty$, $C = \infty$, $D = \infty$.
 - Visit adjacent vertices:
 - Update A: $0 + 10 = 10$
 - Update B: $0 + \infty = \infty$ (not directly connected)
 - Update C: $0 + 5 = 5$
 - Updated distances: $S = 0$, $A = 10$, $B = \infty$, $C = 5$, $D = \infty$.
- **Step 2:** Move to the vertex with the smallest tentative distance, which is 'C'.
 - Distances: $S = 0$, $A = 10$, $B = \infty$, $C = 5$, $D = \infty$.
 - Visit adjacent vertices:
 - Update A: $5 + 3 = 8$ (Smaller than previous 10)
 - Update D: $5 + 2 = 7$
 - Updated distances: $S = 0$, $A = 8$, $B = \infty$, $C = 5$, $D = 7$.
- **Step 3:** Move to the next smallest vertex, 'A'.
 - Distances: $S = 0$, $A = 8$, $B = \infty$, $C = 5$, $D = 7$.
 - Visit adjacent vertices:
 - Update B: $8 + 1 = 9$
 - Updated distances: $S = 0$, $A = 8$, $B = 9$, $C = 5$, $D = 7$.
- **Step 4:** Move to vertex 'D'.

- Distances: $S = 0$, $A = 8$, $B = 9$, $C = 5$, $D = 7$.
- Visit adjacent vertices:
 - Update B: $7 + 4 = 11$ (But 9 is smaller, so no update)
- Updated distances: $S = 0$, $A = 8$, $B = 9$, $C = 5$, $D = 7$.
- **Step 5:** Move to the last vertex, 'B'.
 - All vertices visited.

3. Final Shortest Distances:

- S to A: 8
- S to B: 9
- S to C: 5
- S to D: 7

Q10. Apply Prim's Algorithm to find the minimum spanning tree for the following graph.



Example of a Graph

Ans

Applying Prim's Algorithm

To find the minimum spanning tree (MST) for the given graph using Prim's Algorithm:

1. Initialization:

- Start with vertex 0.
- Initialize the MST as empty.
- Maintain a priority queue for the edges connected to the MST.

2. Step-by-Step Calculation:

- **Step 1:** Start at vertex 0.
 - Add edge (0, 1) with weight 8 to the priority queue.
 - Add edge (0, 7) with weight 8 to the priority queue.
 - Select edge (0, 1) since it has the minimum weight.
- **Step 2:** Add vertex 1 to the MST.
 - Add edge (1, 2) with weight 8 to the priority queue.
 - Add edge (1, 7) with weight 11 to the priority queue.
 - Select edge (0, 7) (same weight as before).
- **Step 3:** Add vertex 7 to the MST.
 - Add edge (7, 6) with weight 1 to the priority queue.
 - Add edge (7, 8) with weight 7 to the priority queue.
 - Select edge (7, 6).
- **Step 4:** Add vertex 6 to the MST.
 - Add edge (6, 5) with weight 2 to the priority queue.
 - Select edge (6, 5).
- **Step 5:** Add vertex 5 to the MST.
 - Add edge (5, 2) with weight 4 to the priority queue.
 - Add edge (5, 3) with weight 14 to the priority queue.
 - Select edge (5, 2).
- **Step 6:** Add vertex 2 to the MST.
 - Add edge (2, 3) with weight 7 to the priority queue.
 - Add edge (2, 8) with weight 2 to the priority queue.
 - Select edge (2, 8).
- **Step 7:** Add vertex 8 to the MST.
 - Select edge (2, 3).
- **Step 8:** Add vertex 3 to the MST.
 - Add edge (3, 4) with weight 9 to the priority queue.

- Select edge (3, 4).
- **Step 9:** Add vertex 4 to the MST.
 - All vertices are now part of the MST.

3. Minimum Spanning Tree Edges:

- (0, 1): 8
- (0, 7): 8
- (7, 6): 1
- (6, 5): 2
- (5, 2): 4
- (2, 8): 2
- (2, 3): 7
- (3, 4): 9

Total weight of the MST: $8 + 8 + 1 + 2 + 4 + 2 + 7 + 9 = 41$

Q11. Apply Insertion and Selection sorting algorithms to sort the following list of items. So, all the intermediate steps. Also, analyze their best, worst and average case time complexity. 12, 5, 2, 15, 25, 30, 45, 8, 17, 50, 3, 7.

Ans

Let's apply **Insertion Sort** and **Selection Sort** to the list of items:

[12, 5, 2, 15, 25, 30, 45, 8, 17, 50, 3, 7].

1. Insertion Sort

Insertion Sort works by building the sorted list one element at a time, by repeatedly picking the next element and inserting it in the correct position.

Initial list:

[12, 5, 2, 15, 25, 30, 45, 8, 17, 50, 3, 7]

Steps:

1. The first element 12 is already sorted, so we move to the second element.
2. **Insert 5** into the sorted sublist [12]. Compare 5 with 12, shift 12 to the right, and insert 5 before 12.

[5, 12, 2, 15, 25, 30, 45, 8, 17, 50, 3, 7]

3. **Insert 2** into the sorted sublist [5, 12]. Compare 2 with 12 and 5, shift them to the right, and insert 2 at the beginning.
[2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7]
4. **Insert 15** into [2, 5, 12]. No shifts are needed as 15 is greater than 12.
[2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7]
5. **Insert 25** into [2, 5, 12, 15]. No shifts are needed.
[2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7]
6. **Insert 30** into [2, 5, 12, 15, 25]. No shifts are needed.
[2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7]
7. **Insert 45** into [2, 5, 12, 15, 25, 30]. No shifts are needed.
[2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7]
8. **Insert 8** into [2, 5, 12, 15, 25, 30, 45]. Compare 8 with 45, 30, 25, 15, and 12, shift all these elements to the right, and insert 8.
[2, 5, 8, 12, 15, 25, 30, 45, 17, 50, 3, 7]
9. **Insert 17** into [2, 5, 8, 12, 15, 25, 30, 45]. Compare 17 with 45, 30, and 25, shift them to the right, and insert 17.
[2, 5, 8, 12, 15, 17, 25, 30, 45, 50, 3, 7]
10. **Insert 50** into [2, 5, 8, 12, 15, 17, 25, 30, 45]. No shifts are needed.
[2, 5, 8, 12, 15, 17, 25, 30, 45, 50, 3, 7]
11. **Insert 3** into [2, 5, 8, 12, 15, 17, 25, 30, 45, 50]. Compare 3 with all elements, shift them to the right, and insert 3 after 2.
[2, 3, 5, 8, 12, 15, 17, 25, 30, 45, 50, 7]
12. **Insert 7** into [2, 3, 5, 8, 12, 15, 17, 25, 30, 45, 50]. Compare 7 with 50, 45, 30, 25, 17, 15, 12, and 8, and shift them to the right. Insert 7 after 5.
[2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 45, 50]

Sorted list (Insertion Sort):

[2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 45, 50]

Time Complexity of Insertion Sort:

- **Best case (already sorted):** $O(n)$
- **Average case:** $O(n^2)$
- **Worst case (reverse sorted):** $O(n^2)$

2. Selection Sort

Selection Sort repeatedly finds the minimum element from the unsorted part and swaps it with the first unsorted element.

Initial list:

[12, 5, 2, 15, 25, 30, 45, 8, 17, 50, 3, 7]

Steps:

1. **Find the minimum element** from the list. It's 2. Swap it with 12.
[2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7]
2. **Find the next minimum element** from [5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7]. It's 3. Swap it with 5.
[2, 3, 12, 15, 25, 30, 45, 8, 17, 50, 5, 7]
3. **Find the next minimum element** from [12, 15, 25, 30, 45, 8, 17, 50, 5, 7]. It's 5. Swap it with 12.
[2, 3, 5, 15, 25, 30, 45, 8, 17, 50, 12, 7]
4. **Find the next minimum element** from [15, 25, 30, 45, 8, 17, 50, 12, 7]. It's 7. Swap it with 15.
[2, 3, 5, 7, 25, 30, 45, 8, 17, 50, 12, 15]
5. **Find the next minimum element** from [25, 30, 45, 8, 17, 50, 12, 15]. It's 8. Swap it with 25.
[2, 3, 5, 7, 8, 30, 45, 25, 17, 50, 12, 15]
6. **Find the next minimum element** from [30, 45, 25, 17, 50, 12, 15]. It's 12. Swap it with 30.
[2, 3, 5, 7, 8, 12, 45, 25, 17, 50, 30, 15]
7. **Find the next minimum element** from [45, 25, 17, 50, 30, 15]. It's 15. Swap it with 45.
[2, 3, 5, 7, 8, 12, 15, 25, 17, 50, 30, 45]
8. **Find the next minimum element** from [25, 17, 50, 30, 45]. It's 17. Swap it with 25.
[2, 3, 5, 7, 8, 12, 15, 17, 25, 50, 30, 45]
9. **Find the next minimum element** from [25, 50, 30, 45]. It's 25. No swap needed.
[2, 3, 5, 7, 8, 12, 15, 17, 25, 50, 30, 45]
10. **Find the next minimum element** from [50, 30, 45]. It's 30. Swap it with 50.
[2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 50, 45]
11. **Find the next minimum element** from [50, 45]. It's 45. Swap it with 50.
[2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 45, 50]

12. The last element is already sorted.

Sorted list (Selection Sort):

[2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 45, 50]

Time Complexity of Selection Sort:

- **Best case:** $O(n^2)$
- **Average case:** $O(n^2)$
- **Worst case:** $O(n^2)$

Time Complexity Comparison:

- **Insertion Sort** has a better best-case time complexity of $O(n)$ when the list is already sorted.
- **Selection Sort** always takes $O(n^2)$, regardless of the list's order.

Q12. What is a heap tree? Create a max heap tree for the following list of items inserted in the order. Also, explain the heap sort with the help of thus created heap tree. 10, 20, 5, 25, 30, 18, 3, 70, 55, 45, 12, 24.

Ans

Heap Tree

A **heap tree** is a specialized binary tree that satisfies the **heap property**:

1. **Max Heap:** Every parent node is greater than or equal to its children.
2. **Min Heap:** Every parent node is smaller than or equal to its children.

In a **Max Heap**, the largest element is always at the root.

Steps to Create a Max Heap

To build a **Max Heap**, we insert elements one by one and adjust (heapify) the tree after each insertion to maintain the heap property.

The given list is:

[10, 20, 5, 25, 30, 18, 3, 70, 55, 45, 12, 24]

Let's insert these elements step-by-step into a max heap.

Step-by-Step Max Heap Creation:

1. **Insert 10:**
[10]

2. **Insert 20:**

Heapify: 20 is larger than 10, so they are swapped.

[20, 10]

3. **Insert 5:**

No need to heapify as 5 is smaller than 20.

[20, 10, 5]

4. **Insert 25:**

Heapify: 25 is larger than 10, so they are swapped. 25 is now the child of 20, no further heapify needed.

[20, 25, 5, 10]

Swap 25 with 20 to maintain the max heap property.

[25, 20, 5, 10]

5. **Insert 30:**

Heapify: 30 is larger than 20, so they are swapped. Then, 30 is swapped with 25 (parent).

[30, 25, 5, 10, 20]

6. **Insert 18:**

No need to heapify as 18 is smaller than 30.

[30, 25, 18, 10, 20, 5]

7. **Insert 3:**

No need to heapify as 3 is smaller than 18.

[30, 25, 18, 10, 20, 5, 3]

8. **Insert 70:**

Heapify: 70 is larger than 10, so swap them. Then, swap 70 with 25 and finally with 30 to maintain the max heap property.

[70, 30, 18, 25, 20, 5, 3, 10]

9. **Insert 55:**

Heapify: 55 is larger than 25, so swap them. Then swap 55 with 30.

[70, 55, 18, 30, 20, 5, 3, 10, 25]

10. **Insert 45:**

Heapify: 45 is larger than 30, so swap them. No further heapify needed.

[70, 55, 18, 45, 20, 5, 3, 10, 25, 30]

11. **Insert 12:**

No need to heapify as 12 is smaller than 45.

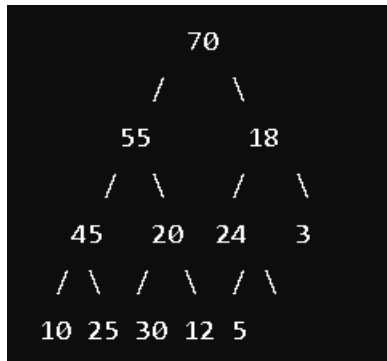
[70, 55, 18, 45, 20, 5, 3, 10, 25, 30, 12]

12. Insert 24:

Heapify: 24 is larger than 5, so swap them. No further heapify needed.

[70, 55, 18, 45, 20, 24, 3, 10, 25, 30, 12, 5]

Final Max Heap:



Heap Sort

Heap sort uses a max heap to sort elements in ascending order. It involves the following steps:

1. **Build a max heap** from the unsorted array.
2. **Extract the root** (maximum element) from the heap and swap it with the last element of the heap.
3. **Heapify** the remaining heap to maintain the max heap property.
4. **Repeat** until all elements are sorted.

Heap Sort Steps:

1. **Initial Max Heap:**
[70, 55, 18, 45, 20, 24, 3, 10, 25, 30, 12, 5]
2. **Swap 70 (root) with 5 (last element):**
[5, 55, 18, 45, 20, 24, 3, 10, 25, 30, 12, 70]
Heapify to restore max heap property.
3. **Heapify** after removing 70:
[55, 45, 18, 30, 20, 24, 3, 10, 25, 5, 12]
4. **Swap 55 (root) with 12 (last element):**
[12, 45, 18, 30, 20, 24, 3, 10, 25, 5, 55, 70]
Heapify.
5. **Heapify** after removing 55:
[45, 30, 18, 25, 20, 24, 3, 10, 12, 5]

6. **Swap 45 with 5:**

[5, 30, 18, 25, 20, 24, 3, 10, 12, 45, 55, 70]

Heapify.

7. **Heapify** after removing 45:

[30, 25, 18, 10, 20, 24, 3, 5, 12]

8. **Swap 30 with 12:**

[12, 25, 18, 10, 20, 24, 3, 5, 30, 45, 55, 70]

Heapify.

9. **Heapify** after removing 30:

[25, 20, 18, 10, 12, 24, 3, 5]

10. **Swap 25 with 5:**

[5, 20, 18, 10, 12, 24, 3, 25, 30, 45, 55, 70]

Heapify.

11. **Heapify** after removing 25:

[20, 12, 18, 10, 5, 24, 3]

12. Continue extracting and heapifying until the list is fully sorted.

Final Sorted Array (Heap Sort):

[3, 5, 10, 12, 18, 20, 24, 25, 30, 45, 55, 70]

Time Complexity of Heap Sort:

- **Best case:** $O(n \log n)$
- **Average case:** $O(n \log n)$
- **Worst case:** $O(n \log n)$

Heap sort is efficient for both average and worst cases, making it useful for large datasets.

Q13. Write a program in 'C' language for 2-way merge sort.

Ans

Here's a C program for 2-way Merge Sort:

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

```
// Function to merge two halves into a sorted array
```

```

void merge(int arr[], int left, int mid, int right) {

    int i, j, k;

    int n1 = mid - left + 1;

    int n2 = right - mid;

    // Temporary arrays

    int L[n1], R[n2];

    // Copy data to temporary arrays L[] and R[]

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr[left..right]

    i = 0; // Initial index of the first subarray
    j = 0; // Initial index of the second subarray
    k = left; // Initial index of the merged subarray

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
}

```

```
}
```

```
// Copy the remaining elements of L[], if any
```

```
while (i < n1) {
```

```
    arr[k] = L[i];
```

```
    i++;
```

```
    k++;
```

```
}
```

```
// Copy the remaining elements of R[], if any
```

```
while (j < n2) {
```

```
    arr[k] = R[j];
```

```
    j++;
```

```
    k++;
```

```
}
```

```
}
```

```
// Function to implement Merge Sort
```

```
void mergeSort(int arr[], int left, int right) {
```

```
    if (left < right) {
```

```
        // Find the middle point
```

```
        int mid = left + (right - left) / 2;
```

```
        // Sort the first half
```

```
        mergeSort(arr, left, mid);
```

```
        // Sort the second half
```

```
        mergeSort(arr, mid + 1, right);
```

```

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);

    return 0;
}

```

Explanation:

- **mergeSort():** Recursively divides the array into halves.
- **merge():** Merges two sorted halves.
- The array is divided until each element is a single unit, then merged in a sorted order.

Output:

```

PS D:\.vscode> cd "d:\.vscode\c language\c++\" ; if ($?) { g++ BCSL-032_Q2.cpp -o BCSL-032_Q2 } ; if ($?) { .\BCSL-032_Q2 }
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
PS D:\.vscode\c language\c++>

```

This is the implementation of 2-way merge sort in C.

Q14. What is Splay tree? Explain the Zig zag and Zag zig rotations in Splay tree with the help of a suitable example.

Ans

What is a Splay Tree?

A **Splay Tree** is a self-adjusting binary search tree where recently accessed elements are moved to the root through a series of tree rotations. The primary purpose of a splay tree is to keep frequently accessed nodes closer to the root, reducing the time required for future accesses.

The basic operations like search, insert, and delete in a splay tree involve bringing the accessed node to the root using **splaying**, which involves a sequence of rotations. This makes recently accessed elements quicker to access.

Types of Rotations in a Splay Tree:

There are three main types of rotations used during splaying:

1. **Zig (Single Rotation)**
2. **Zig-Zig (Double Rotation)**
3. **Zig-Zag (Double Rotation)**

We'll focus on **Zig-Zag** and **Zag-Zig** rotations here.

Zig-Zag and Zag-Zig Rotations

1. Zig-Zag Rotation:

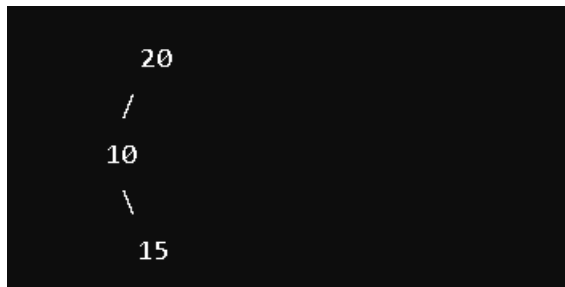
- Occurs when the node we are accessing is a **right child of a left child** or a **left child of a right child**.
- Two rotations are performed: first, a rotation is performed on the parent and then on the grandparent.

2. Zag-Zig Rotation:

- It's the symmetric case of Zig-Zag, where the node is a **left child of a right child**.

Example for Zig-Zag Rotation

Consider a tree where we access node 15 in the following structure:

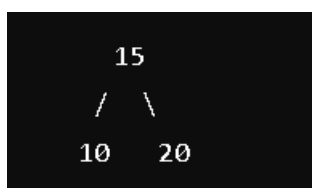


Here, node 15 is a **right child** of 10, and node 10 is a **left child** of 20, making it a Zig-Zag case.

Steps:

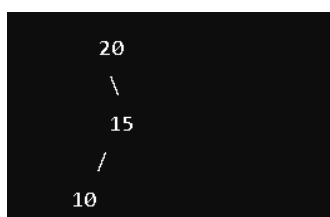
1. First, we perform a **left rotation** on the parent (10).
2. Then, perform a **right rotation** on the grandparent (20).

The tree will look like this after the rotations:



Example for Zag-Zig Rotation

Consider a tree where we access node 10 in the following structure:

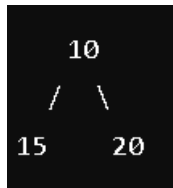


Here, node 10 is a **left child** of 15, and node 15 is a **right child** of 20, making it a Zag-Zig case.

Steps:

1. First, perform a **right rotation** on the parent (15).
2. Then, perform a **left rotation** on the grandparent (20).

The tree will look like this after the rotations:



Conclusion:

Zig-Zag and Zag-Zig rotations are essential operations in splay trees, ensuring efficient reorganization of the tree by bringing accessed nodes closer to the root. These rotations play a vital role in maintaining the splay tree's overall performance, especially in cases where nodes are accessed frequently.

Q15. What is Red-Black tree? Explain insertion and deletion operations in a Red-Black tree with the help of a suitable example.

Ans

What is a Red-Black Tree?

A **Red-Black Tree** is a balanced binary search tree with an extra bit of data per node to keep track of the color of the node: either **red** or **black**. The primary goal of the Red-Black Tree is to maintain balance, ensuring that the height of the tree remains logarithmic relative to the number of nodes, thereby providing efficient search, insertion, and deletion operations.

Properties of a Red-Black Tree:

1. **Each node is either red or black.**
2. **The root node is always black.**
3. **All leaf nodes (NIL or NULL nodes) are black.**
4. **If a node is red, then both its children must be black** (no two consecutive red nodes).
5. **Every path from a given node to its descendant NIL nodes must contain the same number of black nodes** (black height).

Operations in a Red-Black Tree:

1. Insertion in a Red-Black Tree:

When a new node is inserted, it's always colored **red** initially. After insertion, the Red-Black Tree may violate its properties, so the tree must be adjusted by recoloring nodes and performing rotations to restore these properties.

Steps of Insertion:

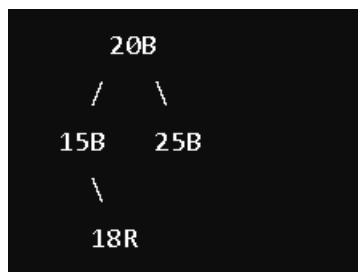
1. **Insert the node** just like in a binary search tree.
2. **Color the newly inserted node red.**
3. If the parent of the newly inserted node is also red, the tree violates the Red-Black Tree properties, and we need to adjust the tree. We use **rotations** and **recoloring** based on the position of the newly inserted node.

Example of Insertion:

Consider inserting node 18 into the following Red-Black Tree:



- Insert 18 as a red node under 15. Now, the tree looks like this:



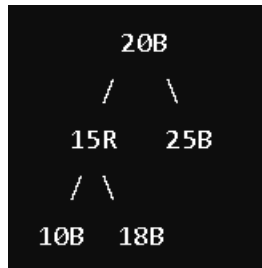
Since the parent (15) is black, no violation occurs, and the tree remains balanced.

Now, if we insert 10 into this tree, we get:



Here, both 10 and 18 are red, violating the rule that no two consecutive red nodes can exist. To fix this:

- **Recoloring:** 15 is recolored red, and both its children, 10 and 18, are recolored black.



If further violations occur, rotations (left or right) are used to maintain balance.

2. Deletion in a Red-Black Tree:

When deleting a node from a Red-Black Tree, the structure needs to be adjusted to maintain the Red-Black properties. This process is more complex than insertion and requires several cases to be handled carefully.

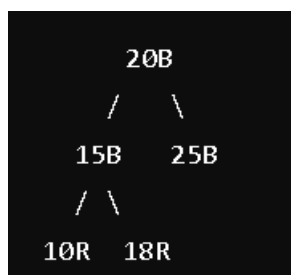
Steps of Deletion:

1. **Perform the standard BST deletion.**
2. If the deleted node is **red**, there is no issue since red nodes can be removed without violating the properties.
3. If the deleted node is **black**, the tree may need to be adjusted to maintain the same number of black nodes on all paths from the root to the leaves.

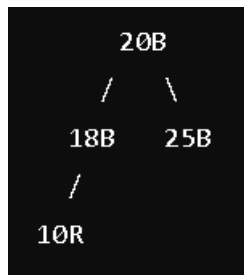
Key Idea: After deleting a black node, the tree will need "fix-up" procedures, such as **recoloring** and **rotations** (similar to insertion) to maintain balance.

Example of Deletion:

Consider deleting node 15 from the following Red-Black Tree:



- After deleting node 15, we need to replace it with its successor (18), resulting in:



In this case, the tree remains balanced, but if a black node is removed, the tree would require further adjustments (recoloring and/or rotations).

Conclusion:

The Red-Black Tree ensures that all operations (search, insertion, deletion) take **$O(\log n)$** time, even in the worst case. By carefully recoloring and performing rotations after insertion and deletion, Red-Black Trees maintain balance while efficiently managing data.

Q16. Explain Direct File and Indexed Sequential File Organization.

Ans

Direct File Organization and **Indexed Sequential File Organization** are two different methods of organizing data for efficient retrieval and storage in computer systems. Here's an explanation of each:

1. Direct File Organization:

In **direct file organization** (also known as **hash file organization**), records are stored at a location determined by a hashing algorithm. This method allows for direct access to data without the need to sequentially search through a file, making it efficient for large databases where frequent random access is needed.

- **Storage:** A hashing function generates a unique address or location for each record based on its key (e.g., student ID, employee number).
- **Access Time:** Very fast as the record can be accessed directly without searching sequentially.
- **Advantages:**
 - Quick data retrieval due to direct access.
 - Efficient for applications with a high volume of random access.
- **Disadvantages:**

- Collisions (two records hashed to the same location) can occur, requiring additional methods like open addressing or chaining to handle them.
- Difficult to manage with frequent insertions and deletions.

2. Indexed Sequential File Organization:

Indexed sequential file organization combines the features of sequential and direct access. Records are stored in a sorted (sequential) order, and an index is maintained to provide direct access to these records.

- **Storage:** The data is stored sequentially, and an index table is built to map keys to their locations. The index can point to specific blocks or records within the file.
- **Access Time:** Faster than sequential access but slower than direct file organization. First, the index is searched to locate the block, and then the data is fetched.
- **Advantages:**
 - Supports both sequential and random access.
 - Good for systems that require both types of access, such as transaction systems or large databases.
 - Efficient for range queries (e.g., retrieving a group of records based on key range).
- **Disadvantages:**
 - Requires additional storage for the index.
 - Overhead for maintaining the index when records are added or deleted.
 - More complex than direct file organization.

Comparison:

- **Direct File Organization** is optimal when random access is frequent and retrieval time is critical.
- **Indexed Sequential File Organization** is better suited for applications that require both random and sequential access, as it allows for flexibility but comes with overhead for maintaining the index.