MCS - 021 Assignment solution 3rd semester

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:
This program assumes that the input polynomials are represented as arrays, where the index of the array represents the power of the variable, and the value at each index represents the coefficient for that power.

```c
#include

#define MAX 100 // Maximum degree of polynomial

// Function to multiply two polynomials
void multiplyPolynomials(int poly1[], int poly2[], int result[], int deg1, int deg2) {
// Initialize the result polynomial with 0
for (int i = 0; i <= deg1 + deg2; i++) {
result[i] = 0;
}

// Multiply two polynomials term by term
for (int i = 0; i <= deg1; i++) {
for (int j = 0; j <= deg2; j++) {
result[i + j] += poly1[i] * poly2[j];
}
}
}

// Function to print a polynomial
void printPolynomial(int poly[], int degree) {
for (int i = 0; i <= degree; i++) {
if (poly[i] != 0) {
if (i != 0 && poly[i] > 0) {
printf(" + ");
}
printf("%d", poly[i]);
if (i > 0) {
printf("x^%d", i);
}
}
}
printf("\n");
}

int main() {
int deg1, deg2;

// Read degree of the first polynomial
printf("Enter the degree of the first polynomial: ");
scanf("%d", °1);

// Declare arrays to store coefficients of the polynomials
int poly1[MAX], poly2[MAX], result[MAX];

// Read coefficients of the first polynomial
printf("Enter the coefficients of the first polynomial:\n");
for (int i = 0; i <= deg1; i++) {
printf("Coefficient of x^%d: ", i);
scanf("%d", &poly1[i]);
}

// Read degree of the second polynomial
printf("Enter the degree of the second polynomial: ");
scanf("%d", °2);

// Read coefficients of the second polynomial
printf("Enter the coefficients of the second polynomial:\n");
for (int i = 0; i <= deg2; i++) {
printf("Coefficient of x^%d: ", i);
scanf("%d", &poly2[i]);
}

// Multiply the two polynomials
multiplyPolynomials(poly1, poly2, result, deg1, deg2);

// Print the result
printf("The resultant polynomial after multiplication is:\n");
printPolynomial(result, deg1 + deg2);

return 0;
}
```

### Explanation:
1. **Input**: The program accepts the degree and coefficients of two polynomials.
2. **Multiplication**: It multiplies the two polynomials using a nested loop and stores the result in a third array.
3. **Output**: The resulting polynomial is printed in the standard form.

### Sample Input:
```
Enter the degree of the first polynomial: 2
Enter the coefficients of the first polynomial:
Coefficient of x^0: 3
Coefficient of x^1: 2
Coefficient of x^2: 1

Enter the degree of the second polynomial: 1
Enter the coefficients of the second polynomial:
Coefficient of x^0: 1
Coefficient of x^1: 2
```

### Sample Output:
```
The resultant polynomial after multiplication is:
3 + 8x^1 + 7x^2 + 2x^3
```
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

Ans:-
C program that creates a singly linked list and performs the operations as requested:

1. Insert a new node at the beginning, middle, or end.
2. Delete a node from the linked list.
3. Display the linked list in reverse order.
4. Sort and display the data in ascending order.
5. Count the number of nodes in the linked list.

```c
#include
#include

// Node structure definition
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 at the beginning of the list
void insertAtBeginning(struct Node** head, int data) {
struct Node* newNode = createNode(data);
newNode->next = *head;
*head = newNode;
}

// Function to insert at the end of the list
void insertAtEnd(struct Node** head, int data) {
struct Node* newNode = createNode(data);
if (*head == NULL) {
*head = newNode;
} else {
struct Node* temp = *head;
while (temp->next != NULL) {
temp = temp->next;
}
temp->next = newNode;
}
}

// Function to insert at a specific position in the list
void insertAtPosition(struct Node** head, int data, int pos) {
struct Node* newNode = createNode(data);
if (pos == 1) {
insertAtBeginning(head, data);
return;
}

struct Node* temp = *head;
```

```c
for (int i = 1; i < pos - 1 && temp != NULL; i++) {
temp = temp->next;
}

if (temp == NULL) {
printf("Position out of bounds\n");
} else {
newNode->next = temp->next;
temp->next = newNode;
}
}

// Function to delete a node by value
void deleteNode(struct Node** head, int key) {
struct Node* temp = *head;
struct Node* prev = NULL;

if (temp != NULL && temp->data == key) {
*head = temp->next;
free(temp);
return;
}

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

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

prev->next = temp->next;
free(temp);
}

// Function to count the number of nodes
int countNodes(struct Node* head) {
int count = 0;
while (head != NULL) {
count++;
head = head->next;
}
return count;
}

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

// Function to sort the linked list in ascending order
void sortList(struct Node** head) {
struct Node *i, *j;
```

```
int temp;

if (*head == NULL) {
return;
}

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

int main() {
struct Node* head = NULL;
int choice, data, position;

do {
printf("\nLinked List Operations Menu:\n");
printf("1. Insert at Beginning\n");
printf("2. Insert at End\n");
printf("3. Insert at Position\n");
printf("4. Delete Node\n");
printf("5. Display in Reverse Order\n");
printf("6. Sort and Display\n");
printf("7. Count Nodes\n");
printf("8. Display List\n");
printf("9. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
case 1:
printf("Enter data to insert at the beginning: ");
scanf("%d", &data);
insertAtBeginning(&head, data);
break;

case 2:
printf("Enter data to insert at the end: ");
scanf("%d", &data);
insertAtEnd(&head, data);
break;

case 3:
printf("Enter data to insert: ");
scanf("%d", &data);
printf("Enter position: ");
scanf("%d", &position);
insertAtPosition(&head, data, position);
break;

case 4:
printf("Enter the value to delete: ");
scanf("%d", &data);
deleteNode(&head, data);
break;

case 5:
printf("Linked List in reverse order: ");
displayReverse(head);
printf("NULL\n");
break;

case 6:
printf("Sorted Linked List: ");
sortList(&head);
displayList(head);
break;

case 7:
printf("Total number of nodes: %d\n", countNodes(head));
break;

case 8:
printf("Linked List: ");
displayList(head);
break;

case 9:
printf("Exiting...\n");
break;

default:
printf("Invalid choice! Please try again.\n");
}
} while (choice != 9);

return 0;
}
```

### Explanation:
1. **Node Structure**: A `Node` struct contains an integer `data` and a pointer to the next node.
2. **Insert Operations**:
- Insert at the beginning, end, or at a specific position.
3. **Delete Operation**: Deletes a node with the specified value.
4. **Display in Reverse**: Recursively displays the list in reverse.
5. **Sort and Display**: Sorts the list in ascending order and then displays it.
6. **Count Nodes**: Counts the total number of nodes in the list.

### Sample Menu Interaction:

```
Linked List Operations Menu:
```

```
1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete Node
5. Display in Reverse Order
6. Sort and Display
7. Count Nodes
8. Display List
9. Exit
Enter your choice: 1
Enter data to insert at the beginning: 10
Linked List Operations Menu:
...
```

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
#include
#include

struct Node {
int data;
struct Node* prev;
struct Node* next;
};

struct Node* head = NULL;

// Function to create a new node
struct Node* createNode(int value) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->prev = NULL;
newNode->next = NULL;
return newNode;
}

// Insert a new node at the beginning
void insertAtBeginning(int value) {
struct Node* newNode = createNode(value);
if (head == NULL) {
head = newNode;
} else {
newNode->next = head;
head->prev = newNode;
head = newNode;
}
}

// Insert a new node at the end
void insertAtEnd(int value) {
struct Node* newNode = createNode(value);
```

```
if (head == NULL) {
head = newNode;
} else {
struct Node* temp = head;
while (temp->next != NULL) {
temp = temp->next;
}
temp->next = newNode;
newNode->prev = temp;
}
}

// Insert a new node in the middle
void insertAtMiddle(int value, int position) {
struct Node* newNode = createNode(value);
if (head == NULL) {
head = newNode;
} else {
struct Node* temp = head;
for (int i = 1; temp != NULL && i < position - 1; i++) {
temp = temp->next;
}
if (temp != NULL) {
newNode->next = temp->next;
if (temp->next != NULL) {
temp->next->prev = newNode;
}
newNode->prev = temp;
temp->next = newNode;
} else {
printf("Position out of bounds!\n");
}
}
}

// Delete a node from the list
void deleteNode(int value) {
if (head == NULL) {
printf("List is empty!\n");
return;
}

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

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

if (temp->prev != NULL) {
temp->prev->next = temp->next;
} else {
head = temp->next;
}

if (temp->next != NULL) {
```

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

free(temp);
}

// Sort the linked list in ascending order
void sortList() {
if (head == NULL) {
printf("List is empty!\n");
return;
}

struct Node* i = head;
struct Node* j = NULL;
int temp;

while (i != NULL) {
j = i->next;
while (j != NULL) {
if (i->data > j->data) {
temp = i->data;
i->data = j->data;
j->data = temp;
}
j = j->next;
}
i = i->next;
}
}

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

// Count the number of nodes in the list
int countNodes() {
int count = 0;
struct Node* temp = head;
while (temp != NULL) {
count++;
temp = temp->next;
}
return count;
}

// Calculate the sum of even integers in the list
int sumEvenNumbers() {
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
int main() {
int choice, value, position;

do {
printf("\nMenu:\n");
printf("1. Insert at Beginning\n");
printf("2. Insert at End\n");
printf("3. Insert at 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(value);
break;
case 2:
printf("Enter value to insert at end: ");
scanf("%d", &value);
insertAtEnd(value);
break;
case 3:
printf("Enter value to insert in middle: ");
scanf("%d", &value);
printf("Enter position: ");
scanf("%d", &position);
insertAtMiddle(value, position);
break;
case 4:
printf("Enter value to delete: ");
scanf("%d", &value);
deleteNode(value);
break;
case 5:
displayList();
break;
case 6:
sortList();
printf("List sorted in ascending order.\n");
break;
case 7:
printf("Number of nodes: %d\n", countNodes());
break;
case 8:
printf("Sum of even numbers: %d\n", sumEvenNumbers());
```

```
break;
case 9:
printf("Exiting...\n");
break;
default:
printf("Invalid choice!\n");
}
} while (choice != 9);

return 0;
}
```

### Explanation:
1. **Insert a new node**:
- At the beginning: `insertAtBeginning()`.
- At the end: `insertAtEnd()`.
- In the middle: `insertAtMiddle()` (insert at a given position).

2. **Delete a node**: The `deleteNode()` function finds the node by value and removes it.

3. **Sort and display**: The `sortList()` function sorts the list in ascending order using bubble sort, and `displayList()` prints the values.

4. **Count nodes**: `countNodes()` traverses the list and counts the number of nodes.

5. **Sum of even numbers**: `sumEvenNumbers()` adds up the even integers in the list.

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

Ans:-
### **What is a Dequeue?**

A **Dequeue** (also spelled **Deque**) stands for **Double-Ended Queue**, which is a linear data structure that allows insertion and deletion of elements from both ends: the front and the rear. Unlike a regular queue (FIFO structure), where elements are added at the rear and removed from the front, in a dequeue, you can:
- Insert elements at the front or rear.
- Delete elements from the front or rear.

#### **Types of Dequeues**:
1. **Input-Restricted Dequeue**: Insertion is restricted to one end, but deletion can be done from both ends.
2. **Output-Restricted Dequeue**: Deletion is restricted to one end, but insertion can be done at both ends.

### **Algorithm for Insertion and Deletion Operations in a Dequeue**

#### **1. Insert at the Front of the Dequeue**

Algorithm:
1. Check if the dequeue is full.
- If full, print "Overflow" and exit.
2. If the dequeue is empty (i.e., front = -1 and rear = -1):
- Set both `front` and `rear` to 0.
3. Else, if `front == 0`:
- Set `front` to `n - 1` (wrap around to the last index if the dequeue is implemented circularly).
4. Else:
- Decrement `front` by 1.
5. Insert the element at the position `front`.

#### **2. Insert at the Rear of the Dequeue**

Algorithm:
1. Check if the dequeue is full.
- If full, print "Overflow" and exit.
2. If the dequeue is empty (i.e., front = -1 and rear = -1):
- Set both `front` and `rear` to 0.
3. Else, if `rear == n - 1`:
- Set `rear` to 0 (wrap around to the first index if the dequeue is implemented circularly).
4. Else:
- Increment `rear` by 1.
5. Insert the element at the position `rear`.

#### **3. Delete from the Front of the Dequeue**

Algorithm:
1. Check if the dequeue is empty.
- If empty, print "Underflow" and exit.
2. If `front == rear` (i.e., only one element left):
- Set both `front` and `rear` to -1 (dequeue becomes empty).
3. Else, if `front == n - 1`:
- Set `front` to 0 (wrap around to the first index).
4. Else:
- Increment `front` by 1.
5. Remove the element from the front.

#### **4. Delete from the Rear of the Dequeue**

Algorithm:
1. Check if the dequeue is empty.
- If empty, print "Underflow" and exit.
2. If `front == rear` (i.e., only one element left):
- Set both `front` and `rear` to -1 (dequeue becomes empty).
3. Else, if `rear == 0`:
- Set `rear` to `n - 1` (wrap around to the last index).
4. Else:
- Decrement `rear` by 1.
5. Remove the element from the rear.

### **Example: Insert and Delete Operations on Dequeue**

Consider a dequeue implemented in an array of size `n` with two pointers: `front` and `rear`.

1. **Insert 10 at the rear**: Rear is updated to hold the position, and 10 is stored.
2. **Insert 20 at the rear**: The value 20 is stored at the new rear position.
3. **Delete from the front**: The element at the front is removed.
4. **Insert 30 at the front**: Front pointer is updated, and 30 is stored at the front.

This algorithm allows flexible operations on both ends of the 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

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

To construct the binary tree from the given traversals:

1. **Pre-order (Root, Left, Right)**: A is the root (first element of pre-order).
2. **In-order**: Locate A in the in-order sequence, splitting the left subtree (elements before A) and right subtree (elements after A).

- Left subtree (In-order): B E D F
- Right subtree (In-order): C I H K J G

3. For the left subtree, take the next element in pre-order: **B** is the root of the left subtree.

4. Locate B in the in-order left subtree (B E D F). Everything after B is its right subtree.

- Left of B: None (so B is a leaf)
- Right of B: E D F

5. **D** becomes the root of the right subtree of B.
- Left of D: E
- Right of D: F

This completes the left subtree.

6. Moving to the right subtree of A, the next element in the pre-order is **C**, which is the root of the right subtree.
- Left of C (in in-order): I H K J
- Right of C (in in-order): G

7. For C's left subtree, the root is **G** (next in pre-order).
- Left of G: H
- Right of G: None

H is the next root, which has **I** and **K J** as its children based on the in-order sequence.

The tree can be visualized as:

```
A
/ \
B C
\ \
D G
/ \ \
E F H
/ \
I K
\
J
```

### (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

1. **Post-order (Left, Right, Root)**: A is the root (last element of post-order).
2. **In-order**: Locate A in the in-order sequence.

- Left subtree (In-order): I H J D C K E
- Right subtree (In-order): F L G M B

3. For the left subtree, the next root (from post-order) is **B**.

4. Recursively apply the same logic as above to build the tree:

The tree structure becomes:

```
A
/ \
B F
/ / \
C L G
/ \ \
D E M
/ \ /
I H K
\
J
```

These trees are constructed based on recursive placement of roots and subtrees derived from the traversal sequences.

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:

```c
#include
#include

// Define a node structure
struct Node {
int data;
struct Node* left;
struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->left = NULL;
newNode->right = NULL;
return newNode;
}

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

```
struct Node* insert(struct Node* root, int data) {
if (root == NULL) {
return createNode(data);
}
if (data < root->data) {
root->left = insert(root->left, data);
} else if (data > root->data) {
root->right = insert(root->right, data);
}
return root;
}

// Function for In-order traversal (Left, Root, Right)
void inorder(struct Node* root) {
if (root != NULL) {
inorder(root->left);
printf("%d ", root->data);
inorder(root->right);
}
}

// Function for Pre-order traversal (Root, Left, Right)
void preorder(struct Node* root) {
if (root != NULL) {
printf("%d ", root->data);
preorder(root->left);
preorder(root->right);
}
}

// Function for Post-order traversal (Left, Right, Root)
void postorder(struct Node* root) {
if (root != NULL) {
postorder(root->left);
postorder(root->right);
printf("%d ", root->data);
}
}

// Main function
int main() {
struct Node* root = NULL;
int n, value;

printf("Enter the number of nodes to insert: ");
scanf("%d", &n);

printf("Enter %d values to insert in the BST:\n", n);
for (int i = 0; i < n; i++) {
scanf("%d", &value);
root = insert(root, value);
}

printf("\nIn-order Traversal: ");
inorder(root);

printf("\nPre-order Traversal: ");
preorder(root);
```

printf("\nPost-order Traversal: ");
postorder(root);

return 0;
}
```

### Explanation:

- **Node Structure**: Each node of the BST has a data field and two pointers (`left` and `right`) to its left and right children.
- **Insert Function**: Inserts nodes into the BST. If the data is less than the current node's data, it is inserted into the left subtree, otherwise, into the right subtree.
- **Traversal Functions**:
- **In-order**: Recursively visits the left subtree, the root node, and then the right subtree.
- **Pre-order**: Visits the root node first, then recursively visits the left subtree and the right subtree.
- **Post-order**: Recursively visits the left subtree, the right subtree, and then the root node.
- **Main Function**: Takes user input to insert nodes into the BST and then prints the tree traversals.

### Example Output:

```
Enter the number of nodes to insert: 5
Enter 5 values to insert in the BST:
50 30 20 40 70

In-order Traversal: 20 30 40 50 70
Pre-order Traversal: 50 30 20 40 70
Post-order Traversal: 20 40 30 70 50
```

This program implements a basic binary search tree and displays it in all three traversal forms: **In-order**, **Pre-order**, and **Post-order**.

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:-
### Definition of an AVL Tree:
An **AVL tree** (named after its inventors Adelson-Velsky and Landis) is a **self-balancing binary search tree**. In an AVL tree, the heights of the two child subtrees of any node differ by at most one. If at any time the height difference becomes greater than one, rebalancing is done to restore this property by rotating the tree.

The **height-balance property** ensures that the height of the tree remains logarithmic with respect to the number of nodes, resulting in **O(log n)** time complexity for insertion, deletion, and search operations.

### Rotation Operations in an AVL Tree:
To maintain the balance, the following rotations are used:
1. **Left Rotation (LL Rotation)**
2. **Right Rotation (RR Rotation)**
3. **Left-Right Rotation (LR Rotation)**
4. **Right-Left Rotation (RL Rotation)**

### Insertion into AVL Tree:
Let's build the AVL tree step by step using the following sequence:
```

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

**Step 1: Insert 12**
- 12 is inserted as the root.

```
12
```

Step 2: Insert 5
- 5 becomes the left child of 12.

```
12
/
5
```

Step 3: Insert 15
- 15 becomes the right child of 12 (No balancing needed).

```
12
/ \
5 15
```

Step 4: Insert 20
- 20 becomes the right child of 15.

```
12
/ \
5 15
\
20
```

Step 5: Insert 35
- 35 becomes the right child of 20. The tree becomes unbalanced. A **Left Rotation** is required around node 15.

After balancing:
```
12
/ \
5 20
/ \
15 35
```

Step 6: Insert 8
- 8 becomes the right child of 5. No balancing is needed.

```
12
/ \
5 20

\ / \
8 15 35
```

Step 7: Insert 2
- 2 becomes the left child of 5. No balancing is needed.

```
12
/ \
5 20
/ \ / \
2 8 15 35
```

Step 8: Insert 40
- 40 becomes the right child of 35. No balancing is needed.

```
12
/ \
5 20
/ \ / \
2 8 15 35
\
40
```

Step 9: Insert 14
- 14 becomes the left child of 15. No balancing is needed.

```
12
/ \
5 20
/ \ / \
2 8 15 35
/ \ \
14 40
```

Step 10: Insert 24
- 24 becomes the left child of 35. A **Right-Left Rotation** is required to rebalance around node 35.

After balancing:
```
12
/ \
5 20
/ \ / \
2 8 15 35
/ \ / \
14 24 40
```

Step 11: Insert 27
- 27 becomes the right child of 24. No balancing is needed.

```

```
12
/ \
5 20
/ \ / \
2 8 15 35
/ \ / \
14 24 40
\
27
```

Step 12: Insert 45
- 45 becomes the right child of 40. No balancing is needed.

```
12
/ \
5 20
/ \ / \
2 8 15 35
/ \ / \
14 24 40
\ \
27 45
```

Step 13: Insert 50
- 50 becomes the right child of 45. No balancing is needed.

```
12
/ \
5 20
/ \ / \
2 8 15 35
/ \ / \
14 24 40
\ \
27 45
\
50
```

Step 14: Insert 3
- 3 becomes the right child of 2. No balancing is needed.

```
12
/ \
5 20
/ \ / \
2 8 15 35
\ / \ / \
3 14 24 40
\ \
27 45
\
50
```

Step 15: Insert 4
- 4 becomes the right child of 3. No balancing is needed.

```
12
/ \
5 20
/ \ / \
2 8 15 35
\ / \ / \
3 14 24 40
\ \ \
4 27 45
\
50
```

### Deletion Steps:
Now we delete 2, 4, 5, and 12 from the tree.

Delete 2:
- Node 2 is removed. Its child 3 becomes the left child of 5. No balancing is needed.

```
12
/ \
5 20
/ \ / \
3 8 15 35
\ / \ / \
4 14 24 40
\ \
27 45
\
50
```

Delete 4:
- Node 4 is removed. No further action is needed.

```
12
/ \
5 20
/ \ / \
3 8 15 35
/ \ / \
14 24 40
\ \
27 45
\
50
```

Delete 5:
- Node 5 is removed. Its right child 8 becomes the new left child of 12.

```
```

12
/ \
8 20
/ / \
3 15 35
/ \ / \
14 24 40
\ \
27 45
\
50
```

**Delete 12**:
- Node 12 is removed. Node 14 (in-order successor of 12) replaces 12, and the tree is rebalanced.

**Final Tree**:
```
14
/ \
8 20
/ / \
3 15 35
\ / \
24 40
\ \
27 45
\
50
```

This is the final AVL tree after all the deletions.

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:-
### Definition of a B-tree:
A **B-tree** is a self-balancing tree data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations. It is commonly used in databases and file systems where data is read and written in large blocks.

### Properties of a B-tree:
1. A B-tree of order **m** is a tree where each node has at most **m children**.
2. Each internal node (except for the root) has at least **⌈m/2⌉ children**.
3. All leaves are at the same level.
4. An internal node with **k** children has **k−1** keys.
5. The keys within a node are always stored in a sorted order.
6. Insertion and deletion of data are done in such a way that the tree remains balanced after each operation.

### Creating a B-tree of Order 5:
The order of a B-tree defines the maximum number of children a node can have. For a **B-tree of order 5**, each node can have at most **4 keys** and **5 children**.

We will build the B-tree by inserting the following sequence of keys into an empty B-tree:
```
12, 5, 15, 20, 60, 45, 35, 40, 25, 8, 7, 55, 50, 66, 65, 80

```

### Step-by-Step Insertion:

**1. Insert 12:**
- 12 is inserted into an empty tree.

```
[12]
```

**2. Insert 5:**
- 5 is inserted, and the keys are sorted.

```
[ 5, 12 ]
```

3. Insert 15:
- 15 is inserted and sorted.

```
[ 5, 12, 15 ]
```

4. Insert 20:
- 20 is inserted and sorted.

```
[ 5, 12, 15, 20 ]
```

5. Insert 60:
- 60 is inserted. Now the node has 5 keys, which exceeds the maximum allowed in a B-tree of order 5, so a **split** occurs. The middle key (15) is promoted to a new root, and the remaining keys are split into two nodes.

```
[ 15 ]
/ \
[ 5, 12 ] [ 20, 60 ]
```

6. Insert 45:
- 45 is inserted into the right child and sorted.

```
[ 15 ]
/ \
[ 5, 12 ] [ 20, 45, 60 ]
```

7. Insert 35:
- 35 is inserted and sorted in the right child.

```
[ 15 ]
/ \
[ 5, 12 ] [ 20, 35, 45, 60 ]

```

8. Insert 40:
- 40 is inserted and sorted in the right child. A **split** occurs because there are more than 4 keys. The middle key (45) is promoted to the root.

```
[ 15, 45 ]
/ | \
[ 5, 12 ] [ 20, 35 ] [ 60 ]
```

9. Insert 25:
- 25 is inserted and sorted in the middle child.

```
[ 15, 45 ]
/ | \
[ 5, 12 ] [ 20, 25, 35 ] [ 60 ]
```

10. Insert 8:
- 8 is inserted into the left child and sorted.

```
[ 15, 45 ]
/ | \
[ 5, 8, 12 ] [ 20, 25, 35 ] [ 60 ]
```

11. Insert 7:
- 7 is inserted into the left child and sorted.

```
[ 15, 45 ]
/ | \
[ 5, 7, 8, 12 ] [ 20, 25, 35 ] [ 60 ]
```

12. Insert 55:
- 55 is inserted into the right child and sorted.

```
[ 15, 45 ]
/ | \
[ 5, 7, 8, 12 ] [ 20, 25, 35 ] [ 55, 60 ]
```

13. Insert 50:
- 50 is inserted and sorted in the right child.

```
[ 15, 45 ]
/ | \
[ 5, 7, 8, 12 ] [ 20, 25, 35 ] [ 50, 55, 60 ]
```

14. Insert 66:
- 66 is inserted into the right child and sorted.
```

```
[ 15, 45 ]
/ | \
[ 5, 7, 8, 12 ] [ 20, 25, 35 ] [ 50, 55, 60, 66 ]
```

15. Insert 65:
- 65 is inserted, causing the right child to split. The middle key (60) is promoted to the root.

```
[ 15, 45, 60 ]
/ | | \
[ 5, 7, 8, 12 ] [ 20, 25, 35 ] [ 50, 55 ] [ 65, 66 ]
```

16. Insert 80:
- 80 is inserted into the rightmost child.

```
[ 15, 45, 60 ]
/ | | \
[ 5, 7, 8, 12 ] [ 20, 25, 35 ] [ 50, 55 ] [ 65, 66, 80 ]
```

### Deletion Steps:
Now we delete 5, 12, 8, and 20 from the tree.

1. Delete 5:
- Node 5 is removed from the left child.

```
[ 15, 45, 60 ]
/ | | \
[ 7, 8, 12 ] [ 20, 25, 35 ] [ 50, 55 ] [ 65, 66, 80 ]
```

2. Delete 12:
- Node 12 is removed from the left child.

```
[ 15, 45, 60 ]
/ | | \
[ 7, 8 ] [ 20, 25, 35 ] [ 50, 55 ] [ 65, 66, 80 ]
```

3. Delete 8:
- Node 8 is removed from the left child.

```
[ 15, 45, 60 ]
/ | | \
[ 7 ] [ 20, 25, 35 ] [ 50, 55 ] [ 65, 66, 80 ]
```

4. Delete 20:
- Node 20 is removed from the middle child.

```
```

[15, 45, 60]
/ | | \
[7] [25, 35] [50, 55] [65, 66, 80]
```

### Final B-tree after Deletions:

```
[15, 45, 60]
/ | | \
[7] [25, 35] [50, 55] [65, 66, 80]
```
This is the final B-tree after the insertions and deletions.

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

Ans:-

In order to apply **Dijkstra's algorithm** to find the shortest path from vertex **S** to all other vertices in a given graph, you will need the following steps:

### Steps of Dijkstra's Algorithm:
1. **Initialize**:
- Set the distance of the source vertex (S) to 0.
- Set the distance of all other vertices to infinity (∞).
- Mark all vertices as unvisited.

2. **Select the Current Node**:
- Choose the unvisited vertex with the smallest known distance from the source vertex.

3. **Calculate Tentative Distances**:
- For each neighbor of the current node, calculate the tentative distance from the source.
- If the calculated distance for a vertex is smaller than the current known distance, update the shortest distance.

4. **Mark the Current Node as Visited**:
- Once all neighbors are considered, mark the current node as visited. A visited node will not be checked again.

5. **Repeat**:
- Repeat steps 2 to 4 until all vertices are visited.

6. **Output**:
- Once all nodes have been visited, the shortest path to each vertex from the source will be determined.

### Example:

#### Given Graph:

Let's assume you have a graph with vertices **S, A, B, C, D, E**, and the following weighted edges:

```
S --(4)--> A
S --(1)--> B
A --(3)--> C
A --(1)--> D
B --(2)--> A
B --(5)--> D
```
```

C --(6)--> E
D --(1)--> C
D --(4)--> E
```

### Step-by-Step Application of Dijkstra's Algorithm:

1. **Initialization**:
- Distance from **S** to itself = 0.
- Distances from **S** to all other vertices = ∞.

```
Distance(S) = 0
Distance(A) = ∞
Distance(B) = ∞
Distance(C) = ∞
Distance(D) = ∞
Distance(E) = ∞
```

2. **Choose Vertex S** (smallest distance = 0):
- From **S**, we can go to **A** with cost 4 and to **B** with cost 1.
- Update distances for **A** and **B**.

```
Distance(S) = 0
Distance(A) = 4 (via S)
Distance(B) = 1 (via S)
Distance(C) = ∞
Distance(D) = ∞
Distance(E) = ∞
```

3. **Choose Vertex B** (smallest distance = 1):
- From **B**, we can go to **A** with cost 2 (total = 1 + 2 = 3) and to **D** with cost 5 (total = 1 + 5 = 6).
- Update distances for **A** and **D**.

```
Distance(S) = 0
Distance(A) = 3 (via B)
Distance(B) = 1
Distance(C) = ∞
Distance(D) = 6 (via B)
Distance(E) = ∞
```

4. **Choose Vertex A** (smallest distance = 3):
- From **A**, we can go to **C** with cost 3 (total = 3 + 3 = 6) and to **D** with cost 1 (total = 3 + 1 = 4).
- Update distances for **C** and **D**.

```
Distance(S) = 0
Distance(A) = 3
Distance(B) = 1
Distance(C) = 6 (via A)
Distance(D) = 4 (via A)
Distance(E) = ∞
```
```

5. **Choose Vertex D** (smallest distance = 4):
- From **D**, we can go to **C** with cost 1 (total = 4 + 1 = 5) and to **E** with cost 4 (total = 4 + 4 = 8).
- Update distances for **C** and **E**.

```
Distance(S) = 0
Distance(A) = 3
Distance(B) = 1
Distance(C) = 5 (via D)
Distance(D) = 4
Distance(E) = 8 (via D)
```

6. **Choose Vertex C** (smallest distance = 5):
- From **C**, we can go to **E** with cost 6, but the current shortest path to **E** is 8, so no update is made.

```
Distance(S) = 0
Distance(A) = 3
Distance(B) = 1
Distance(C) = 5
Distance(D) = 4
Distance(E) = 8
```

7. **Choose Vertex E** (smallest distance = 8):
- No further updates are needed.

### Final Shortest Distances from Vertex S:

```
Distance(S) = 0
Distance(A) = 3
Distance(B) = 1
Distance(C) = 5
Distance(D) = 4
Distance(E) = 8
```

### Shortest Paths:
- **S → B**: Distance = 1
- **S → A**: Distance = 3 (via B)
- **S → D**: Distance = 4 (via A)
- **S → C**: Distance = 5 (via D)
- **S → E**: Distance = 8 (via D)

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

Ans:-
To find the minimum spanning tree (MST) using **Prim's Algorithm**, follow these steps:

### Step 1: Start with any node. Let's start with **node 1**.

- **Node 1** has the following edges:
- 1 - 2: weight 8

- 1 - 7: weight 4

The smallest edge is **1 - 7** with weight **4**, so we add this edge to the MST.

### Step 2: Include **node 7** in the MST. Now the available edges are:
- 7 - 1: weight 4 (already included)
- 7 - 8: weight 11
- 7 - 6: weight 7

The smallest edge is **7 - 6** with weight **7**, so we add this edge to the MST.

### Step 3: Include **node 6** in the MST. Now the available edges are:
- 6 - 5: weight 2
- 6 - 7: weight 7 (already included)
- 6 - 8: weight 6

The smallest edge is **6 - 5** with weight **2**, so we add this edge to the MST.

### Step 4: Include **node 5** in the MST. Now the available edges are:
- 5 - 4: weight 10
- 5 - 6: weight 2 (already included)

The smallest edge is **5 - 4** with weight **10**, so we add this edge to the MST.

### Step 5: Include **node 4** in the MST. Now the available edges are:
- 4 - 3: weight 9

Add **4 - 3** to the MST.

### Step 6: Include **node 3** in the MST. Now the available edges are:
- 3 - 2: weight 7

Add **3 - 2** to the MST.

### Step 7: Include **node 2** in the MST. Now the available edges are:
- 2 - 1: weight 8 (already included)

### Final MST:
- Edges: **1 - 7**, **7 - 6**, **6 - 5**, **5 - 4**, **4 - 3**, **3 - 2**
- Weights: 4, 7, 2, 10, 9, 7

### Total weight of the MST: **39**

So, the minimum spanning tree includes the edges with the weights **4, 7, 2, 10, 9, and 7**, and the total weight is **39**.

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:-
### Insertion Sort

List to be sorted:
`12, 5, 2, 15, 25, 30, 45, 8, 17, 50, 3, 7`

#### Steps for Insertion Sort:

1. **Initial List**:

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

2. **Pass 1 (i = 1)**:
Compare `5` with `12`, shift `12` to the right, and insert `5` at the beginning.
List: `5, 12, 2, 15, 25, 30, 45, 8, 17, 50, 3, 7`

3. **Pass 2 (i = 2)**:
Compare `2` with `12` and `5`, shift both to the right, and insert `2`.
List: `2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7`

4. **Pass 3 (i = 3)**:
`15` is already in the correct position.
List: `2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7`

5. **Pass 4 (i = 4)**:
`25` is already in the correct position.
List: `2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7`

6. **Pass 5 (i = 5)**:
`30` is already in the correct position.
List: `2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7`

7. **Pass 6 (i = 6)**:
`45` is already in the correct position.
List: `2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7`

8. **Pass 7 (i = 7)**:
Compare `8` with `45, 30, 25, 15, 12`, shift them to the right, and insert `8`.
List: `2, 5, 8, 12, 15, 25, 30, 45, 17, 50, 3, 7`

9. **Pass 8 (i = 8)**:
Compare `17` with `45, 30, 25`, shift them to the right, and insert `17`.
List: `2, 5, 8, 12, 15, 17, 25, 30, 45, 50, 3, 7`

10. **Pass 9 (i = 9)**:
`50` is already in the correct position.
List: `2, 5, 8, 12, 15, 17, 25, 30, 45, 50, 3, 7`

11. **Pass 10 (i = 10)**:
Compare `3` with all previous elements, shift them to the right, and insert `3`.
List: `2, 3, 5, 8, 12, 15, 17, 25, 30, 45, 50, 7`

12. **Pass 11 (i = 11)**:
Compare `7` with all previous elements, shift them to the right, and insert `7`.
List: `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`

### Selection Sort

#### Steps for Selection Sort:

1. **Initial List**:
`12, 5, 2, 15, 25, 30, 45, 8, 17, 50, 3, 7`

2. **Pass 1**:
Find the smallest element (`2`) and swap with the first element (`12`).
List: `2, 5, 12, 15, 25, 30, 45, 8, 17, 50, 3, 7`

3. **Pass 2**:
Find the next smallest element (`3`) and swap with the second element (`5`).
List: `2, 3, 12, 15, 25, 30, 45, 8, 17, 50, 5, 7`

4. **Pass 3**:
Find the next smallest element (`5`) and swap with the third element (`12`).
List: `2, 3, 5, 15, 25, 30, 45, 8, 17, 50, 12, 7`

5. **Pass 4**:
Find the next smallest element (`7`) and swap with the fourth element (`15`).
List: `2, 3, 5, 7, 25, 30, 45, 8, 17, 50, 12, 15`

6. **Pass 5**:
Find the next smallest element (`8`) and swap with the fifth element (`25`).
List: `2, 3, 5, 7, 8, 30, 45, 25, 17, 50, 12, 15`

7. **Pass 6**:
Find the next smallest element (`12`) and swap with the sixth element (`30`).
List: `2, 3, 5, 7, 8, 12, 45, 25, 17, 50, 30, 15`

8. **Pass 7**:
Find the next smallest element (`15`) and swap with the seventh element (`45`).
List: `2, 3, 5, 7, 8, 12, 15, 25, 17, 50, 30, 45`

9. **Pass 8**:
Find the next smallest element (`17`) and swap with the eighth element (`25`).
List: `2, 3, 5, 7, 8, 12, 15, 17, 25, 50, 30, 45`

10. **Pass 9**:
Find the next smallest element (`25`) and swap with the ninth element (`25`). No change.
List: `2, 3, 5, 7, 8, 12, 15, 17, 25, 50, 30, 45`

11. **Pass 10**:
Find the next smallest element (`30`) and swap with the tenth element (`50`).
List: `2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 50, 45`

12. **Pass 11**:
Find the next smallest element (`45`) and swap with the eleventh element (`50`).
List: `2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 45, 50`

Sorted List (Selection Sort):
`2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 45, 50`

### Time Complexity Analysis

- **Insertion Sort:**
- **Best Case (already sorted)**: O(n)
- **Worst Case (reverse order)**: O(n^2)
- **Average Case**: O(n^2)

- **Selection Sort:**
- **Best Case**: O(n^2)
- **Worst Case**: O(n^2)
- **Average Case**: O(n^2)

Conclusion: Insertion sort performs better than selection sort in the best-case scenario when the list is already sorted, with a time complexity of O(n). However, in most other cases, both algorithms have a time complexity of O(n^2).

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:-
### What is a Heap Tree?

A **heap tree** is a special binary tree that satisfies the **heap property**:
- In a **max heap**, for each node, the value of the parent is greater than or equal to the values of its children.
- In a **min heap**, for each node, the value of the parent is less than or equal to the values of its children.

A heap is usually implemented as a binary heap, which is a complete binary tree where every level, except possibly the last, is fully filled, and all nodes are as far left as possible.

### Steps to Create a Max Heap Tree

The given list of items is:
`10, 20, 5, 25, 30, 18, 3, 70, 55, 45, 12, 24`

We will insert these elements one by one and maintain the max heap property.

#### Insert Elements and Build Max Heap:

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

2. **Insert 20**:
Swap 20 with 10 to maintain the max heap property.
Heap: `[20, 10]`

3. **Insert 5**:
Heap: `[20, 10, 5]`

4. **Insert 25**:
Insert 25 as a child of 10, then swap 25 with 20 to maintain the heap property.
Heap: `[25, 20, 5, 10]`

5. **Insert 30**:
Insert 30 as a child of 20, then swap 30 with 25.
Heap: `[30, 25, 5, 10, 20]`

6. **Insert 18**:
Heap: `[30, 25, 18, 10, 20, 5]`

7. **Insert 3**:
Heap: `[30, 25, 18, 10, 20, 5, 3]`

8. **Insert 70**:
Insert 70, swap with 25, then with 30 to maintain the heap property.
Heap: `[70, 30, 18, 25, 20, 5, 3, 10]`

9. **Insert 55**:
Insert 55, swap with 25, then with 30.
Heap: `[70, 55, 18, 30, 20, 5, 3, 10, 25]`

10. **Insert 45**:
Insert 45, swap with 30.
Heap: `[70, 55, 18, 45, 20, 5, 3, 10, 25, 30]`

11. **Insert 12**:
Heap: `[70, 55, 18, 45, 20, 5, 3, 10, 25, 30, 12]`

12. **Insert 24**:
Heap: `[70, 55, 18, 45, 20, 24, 3, 10, 25, 30, 12, 5]`

Final Max Heap Tree:
```
70
/ \
55 18
/ \ / \
45 20 24 3
/ \ / \ / \
10 25 30 12 5
```

### Heap Sort Algorithm

Heap sort uses the heap data structure to sort elements. The steps are:

1. **Build a Max Heap**: Arrange the elements into a max heap, as shown above.

2. **Extract Max Element**: The largest element (root) is at the top of the heap. Swap it with the last element in the heap and reduce the heap size by 1. Heapify the remaining heap.

3. **Heapify**: After each extraction, restore the heap property by reheapifying the root to maintain the max heap structure.

4. **Repeat**: Continue this process of extracting the max element and heapifying until the heap is empty.

#### Example of Heap Sort Using the Above Max Heap:

1. **Initial Heap**:
`[70, 55, 18, 45, 20, 24, 3, 10, 25, 30, 12, 5]`

2. **Extract 70** (largest element):
Swap 70 with 5, reduce heap size, and heapify:
Heap after swap: `[5, 55, 18, 45, 20, 24, 3, 10, 25, 30, 12]`
Heapify: `[55, 45, 18, 30, 20, 24, 3, 10, 25, 5, 12]`

3. **Extract 55**:
Swap 55 with 12, reduce heap size, and heapify:
Heap after swap: `[12, 45, 18, 30, 20, 24, 3, 10, 25, 5]`
Heapify: `[45, 30, 18, 25, 20, 24, 3, 10, 12, 5]`

4. **Extract 45**:
Swap 45 with 5, reduce heap size, and heapify:
Heap after swap: `[5, 30, 18, 25, 20, 24, 3, 10, 12]`
Heapify: `[30, 25, 18, 12, 20, 24, 3, 10, 5]`

5. **Extract 30**:
Swap 30 with 5, reduce heap size, and heapify:
Heap after swap: `[5, 25, 18, 12, 20, 24, 3, 10]`
Heapify: `[25, 20, 18, 12, 5, 24, 3, 10]`

6. **Extract 25**:
Swap 25 with 10, reduce heap size, and heapify:

Heap after swap: `[10, 20, 18, 12, 5, 24, 3]`
Heapify: `[20, 12, 18, 10, 5, 24, 3]`

7. **Extract 20**:
Swap 20 with 3, reduce heap size, and heapify:
Heap after swap: `[3, 12, 18, 10, 5, 24]`
Heapify: `[18, 12, 3, 10, 5, 24]`

8. **Extract 18**:
Swap 18 with 24, reduce heap size, and heapify:
Heap after swap: `[24, 12, 3, 10, 5]`
Heapify: `[24, 12, 3, 10, 5]`

9. **Extract 24**:
Swap 24 with 5, reduce heap size, and heapify:
Heap after swap: `[5, 12, 3, 10]`
Heapify: `[12, 10, 3, 5]`

10. **Extract 12**:
Swap 12 with 5, reduce heap size, and heapify:
Heap after swap: `[5, 10, 3]`
Heapify: `[10, 5, 3]`

11. **Extract 10**:
Swap 10 with 3, reduce heap size, and heapify:
Heap after swap: `[3, 5]`
Heapify: `[5, 3]`

12. **Extract 5**:
Swap 5 with 3, reduce heap size, and heapify:
Heap after swap: `[3]`

13. **Extract 3**:
Heap is empty.

Sorted List:
`[3, 5, 10, 12, 18, 20, 24, 25, 30, 45, 55, 70]`

### Time Complexity of Heap Sort

- **Building the Max Heap**: O(n)
- **Heapifying (in-place sort)**: O(n log n)
Overall time complexity: **O(n log n)**

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

Ans:-

### C Program for 2-Way Merge Sort

```c
#include

// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right) {
int n1 = mid - left + 1;
int n2 = right - mid;

// Temporary arrays to store subarrays
int L[n1], R[n2];

// Copy data to temporary arrays L[] and R[]
for (int i = 0; i < n1; i++)
L[i] = arr[left + i];
for (int j = 0; j < n2; j++)
R[j] = arr[mid + 1 + j];

// Merge the temporary arrays back into arr[left..right]
int i = 0, j = 0, k = left;

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

// Copy remaining elements of L[] if any
while (i < n1) {
arr[k] = L[i];
i++;
k++;
}

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

// Recursively sort first and second halves
mergeSort(arr, left, mid);
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");
}

// Main function to test the program
```

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

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

mergeSort(arr, 0, size - 1);

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

return 0;
}
```

### Explanation:

1. **`merge` function**:
- It takes three arguments: the array, the left index, and the right index. It merges two subarrays (from `left` to `mid` and from `mid + 1` to `right`) into a single sorted array.
- This function compares elements from two temporary arrays `L[]` and `R[]` and merges them into the original array `arr[]`.

2. **`mergeSort` function**:
- This function recursively splits the array into two halves, sorts each half, and merges them using the `merge` function. It divides the array until each subarray has only one element (which is trivially sorted).

3. **`printArray` function**:
- This function prints the array elements.

4. **Main function**:
- The `main` function defines an array, calls the `mergeSort` function to sort the array, and prints the sorted array.

### Output Example:
```
Given array is:
12 11 13 5 6 7

Sorted array is:
5 6 7 11 12 13
```

### Time Complexity:
- **Time complexity**: O(n log n) for both the worst-case and average-case scenarios, where `n` is the number of elements.
- **Space complexity**: O(n) due to the auxiliary arrays used for merging.

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 type of **self-adjusting binary search tree** where recently accessed elements are moved to the root through a process called **splaying**. This helps to keep the frequently accessed nodes near the top of the tree, improving the average access time for these nodes.

Key operations like insertion, deletion, and search are followed by a splay operation, where the accessed node is moved to the root of the tree through a series of tree rotations.

### Operations in a Splay Tree

The tree adjusts itself by performing a series of rotations (either **Zig**, **Zig-Zag**, **Zig-Zig**, etc.) to bring the accessed node to the root. This process helps improve the time complexity for future operations on recently accessed nodes.

### Types of Rotations in Splay Tree

1. **Zig (Single Rotation)**:
- When the node being splayed is a direct child of the root. It involves a single rotation.

2. **Zig-Zag (Double Rotation)**:
- When the node being splayed is a right child of its parent, and its parent is a left child of its grandparent, or vice versa. It involves two rotations.

3. **Zig-Zig (Double Rotation)**:
- When both the node being splayed and its parent are either both left children or both right children. This also involves two rotations.

Let's focus on **Zig-Zag** and **Zag-Zig** rotations with examples.

### Zig-Zag Rotation (Left-Right Rotation)

A **Zig-Zag** rotation occurs when:
- The node is a **right child** of its parent, and the parent is a **left child** of the grandparent.

#### Example:
Consider the following tree:

```
20
/
10
\
15
```

We want to splay node 15 (the right child of 10, and 10 is the left child of 20). This requires a Zig-Zag (left-right) rotation.

#### Steps for Zig-Zag Rotation:
1. **Left Rotation** on node 10 (around 15):
- Resulting tree after left rotation:
```
20
/
15
/
10
```

2. **Right Rotation** on node 20 (around 15):
- Resulting tree after right rotation:
```
```

15
/ \
10 20
```

Now, node 15 has been splayed to the root of the tree.

---

### Zag-Zig Rotation (Right-Left Rotation)

A **Zag-Zig** rotation occurs when:
- The node is a **left child** of its parent, and the parent is a **right child** of the grandparent.

#### Example:
Consider the following tree:

```
20
\
30
/
25
```

We want to splay node 25 (the left child of 30, and 30 is the right child of 20). This requires a Zag-Zig (right-left) rotation.

#### Steps for Zag-Zig Rotation:
1. **Right Rotation** on node 30 (around 25):
- Resulting tree after right rotation:
```
20
\
25
\
30
```

2. **Left Rotation** on node 20 (around 25):
- Resulting tree after left rotation:
```
25
/ \
20 30
```

Now, node 25 has been splayed to the root of the tree.

### Summary

- **Zig-Zag** occurs when the node is a **right child** of its parent, and the parent is a **left child** of its grandparent.
- **Zag-Zig** occurs when the node is a **left child** of its parent, and the parent is a **right child** of its grandparent.

These rotations help maintain the balance of the tree by moving frequently accessed nodes closer to the root,
```

thus optimizing future operations.

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 type of **self-balancing binary search tree** where each node stores an extra bit representing "color" (either red or black). Red-Black Trees are used to ensure the tree remains balanced after insertion or deletion operations, guaranteeing efficient search, insertion, and deletion operations with a time complexity of **O(log n)**.

### Properties of a Red-Black Tree:

1. **Every node is either red or black**.
2. **The root is always black**.
3. **Every leaf (NIL node) is black** (leaf nodes are sometimes represented by `NULL` or NIL pointers).
4. **If a node is red, then both its children must be black** (no two consecutive red nodes on any path).
5. **Every path from a node to its descendant NIL nodes has the same number of black nodes** (this is called the black-height property).

These properties ensure that the tree remains approximately balanced, preventing it from becoming skewed like a regular binary search tree.

### Insertion in a Red-Black Tree

When inserting a new node into a Red-Black Tree, the following steps are followed to maintain the Red-Black properties:

#### Insertion Steps:

1. **Insert the node** like a regular binary search tree (BST) node. Initially, color the new node **red**.

2. **Fix any Red-Black Tree property violations** caused by the insertion. These violations can occur if the parent of the inserted node is also red (which violates the property of no consecutive red nodes). There are three cases that need to be handled (usually with rotations and color changes).

### Example of Red-Black Tree Insertion

Let’s insert the following sequence of numbers into an empty Red-Black Tree:
`10, 20, 30, 15`

#### Step 1: Insert 10
- Insert 10 as the root node and color it **black** (since the root is always black).

```
10 (black)
```

#### Step 2: Insert 20
- Insert 20 as a right child of 10 and color it **red** (new nodes are always red).

```
```

10 (black)
\
20 (red)
```

#### Step 3: Insert 30
- Insert 30 as a right child of 20 and color it **red**. Now, we have two consecutive red nodes (20 and 30), which violates the Red-Black Tree properties.

```
10 (black)
\
20 (red)
\
30 (red)
```

Fixing the violation:
- Perform a **left rotation** around node 10.
- Recolor nodes 20 and 10. Node 20 becomes **black**, and node 10 becomes **red**.

```
20 (black)
/ \
10 (red) 30 (red)
```

Now the tree satisfies all Red-Black properties.

#### Step 4: Insert 15
- Insert 15 as the left child of 20 and color it **red**.

```
20 (black)
/ \
10 (red) 30 (red)
\
15 (red)
```

Fixing the violation:
- Since we have two consecutive red nodes (10 and 15), we perform a **right rotation** around node 10.
- After the rotation, recolor node 15 **black** and node 10 **red**.

```
20 (black)
/ \
15 (black) 30 (red)
/
10 (red)
```

Now, the Red-Black Tree is balanced and satisfies all properties.

### Deletion in a Red-Black Tree

Deletion in a Red-Black Tree is more complex than insertion because removing a node can potentially violate
```

multiple Red-Black properties. The main challenge is ensuring that the **black-height** property is preserved.

#### Deletion Steps:

1. **Remove the node** like you would in a regular binary search tree. If the node to be deleted has two children, replace it with its in-order successor (the smallest node in the right subtree) or its in-order predecessor (the largest node in the left subtree).

2. If the deleted node or the node that replaces it is **red**, the tree remains valid, and no further fixes are required. If the node is **black**, it could violate the Red-Black properties, and fixing is necessary.

3. **Fix the violations** by applying a series of transformations:
- **Recoloring** nodes.
- **Rotations** (left or right).
- Use the concept of "double black" to balance the tree when a black node is deleted.

### Example of Red-Black Tree Deletion

Let’s delete a node from the following Red-Black Tree:

```
20 (black)
/ \
15 (black) 30 (red)
/
10 (red)
```

#### Step 1: Delete node 10
- Node 10 is **red**, so its deletion does not violate any Red-Black properties.

```
20 (black)
/ \
15 (black) 30 (red)
```

#### Step 2: Delete node 15
- Node 15 is **black**. When deleting a black node, we need to fix the tree to maintain the black-height property.

```
20 (black)
\
30 (red)
```

Fixing the violation:
- Node 30 becomes **black** to maintain the correct black height.

```
20 (black)
\
30 (black)
```

The tree is now balanced and satisfies the Red-Black Tree properties.

### Summary of Red-Black Tree Operations

- **Insertion**: Always insert a new node as **red**. Use rotations and recoloring to fix violations if necessary.
- **Deletion**: If a black node is deleted, fix violations using recoloring and rotations to maintain the tree's balance and properties.

### Time Complexity:
- **Insertion**: O(log n)
- **Deletion**: O(log n)
- **Search**: O(log n)

These complexities ensure that Red-Black Trees are efficient for dynamic sets where insertions and deletions occur frequently.

Q16. Explain Direct File and Indexed Sequential File Organization.

Ans:-
### Direct File Organization

Direct File Organization, also known as **hashing file organization**, is a method of file storage where records are directly accessed using a **hash function**. This method is particularly useful when quick access to records is needed, without having to search through large sets of data sequentially.

#### Characteristics:
1. **Hash Function**: A mathematical function is used to calculate the address or location where the record should be stored. The key (or some part of the data) is passed through this hash function to generate a unique location or address in the file system.

2. **Direct Access**: Once the hash function computes the location, the system directly accesses the record in that location, eliminating the need to search through the file sequentially.

3. **Collisions**: Two different records may have the same hash value, leading to a **collision**. Various strategies such as **open addressing** or **chaining** are used to handle collisions.

4. **Efficient for Random Access**: This method is very efficient when you need to frequently retrieve or update data quickly. It is commonly used in databases where records are accessed using a unique identifier (like student ID or account number).

#### Advantages:
- **Fast Access**: Since the hash function provides direct access to a record, there's no need to perform a sequential search, making it faster for retrieval.
- **Best for Unique Records**: Works best when accessing records using a unique key, such as an account number or employee ID.

#### Disadvantages:
- **Collisions**: Collisions can occur when multiple records are assigned the same hash value, leading to complexity in managing and retrieving those records.
- **Space**: Hashing can result in unused space (called **bucket overflow**), leading to inefficient use of storage.

### Indexed Sequential File Organization

Indexed Sequential File Organization combines the benefits of **sequential access** with **direct access** using an **index**. This file organization method stores records sequentially but also maintains an index that allows for quicker access to records without searching through the entire file.

#### Characteristics:
1. **Sequential Storage**: The records are stored in a sequential order based on a key field (such as customer ID or name). This allows for easy sequential access.

2. **Index Table**: Along with sequential storage, an index table is maintained. This table contains key fields and pointers (addresses) to the corresponding record. The index allows for faster access by providing direct pointers to specific data blocks.

3. **Two Types of Indexing**:
- **Primary Indexing**: The primary index is built on a primary key and contains pointers to blocks of records.
- **Secondary Indexing**: Additional indexes can be created on non-primary key fields to allow for fast access on other fields as well.

4. **Direct and Sequential Access**: Indexed Sequential File Organization supports both **sequential access** (for reading through records) and **direct access** (for quickly locating specific records using the index).

#### Example:
If we have a file containing student records with fields like **Student ID** and **Name**, the records might be stored in sequential order by **Student ID**. An index table is built using the **Student ID** that points to the specific locations of these records.

Index Table	Record Location
1001	Block 1
1005	Block 2
1010	Block 3

When a record with **Student ID 1010** is searched, the index table is first checked to find the block where this record is stored. This allows for faster access compared to searching through the entire list of records sequentially.

#### Advantages:
- **Fast Access**: The index allows for fast direct access to records.
- **Sequential and Direct Access**: The method supports both sequential and direct access to records, making it flexible.
- **Efficient for Large Files**: It is useful for large files where random access is required frequently, but sequential access is also needed for processing large amounts of data.

#### Disadvantages:
- **Index Overhead**: Maintaining an index table requires extra space and memory, which adds overhead.
- **Updating the Index**: When records are inserted or deleted, the index must be updated, which adds complexity.

### Comparison of Direct File and Indexed Sequential File Organization

Aspect	**Direct File Organization**	**Indexed Sequential File Organization**
Access Method	Direct access using a hash function	Direct access using index, sequential access possible
Speed	Very fast for direct access	Fast access for both direct and sequential operations
Structure	Records are stored at hash-calculated addresses	Records are stored sequentially with an index table

Collisions	Requires handling of hash collisions	No collision issues, but index management is needed
Use Case	Best for direct access to records using unique keys	Best for large files where both sequential and direct access is required
Complexity	Simple except for collision handling	More complex due to index maintenance

### Summary:
- **Direct File Organization** is efficient for random access using unique keys through hashing but requires managing collisions.
- **Indexed Sequential File Organization** provides both sequential and direct access to data using an index, making it suitable for large datasets where both types of access are needed. However, maintaining the index adds complexity.