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	:	**31st October,2024(For July Session)**
	:	**30th April,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

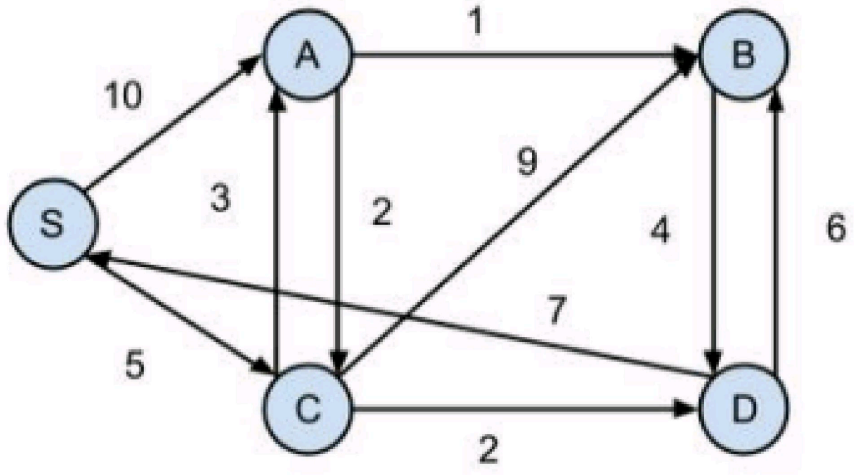
Further delete 2, 4, 5 and 12 from the above AVL tree.

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:

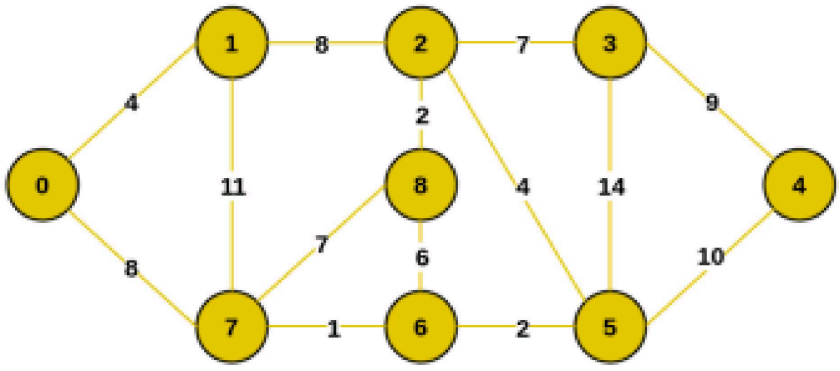
12, 5, 15, 20, 60, 45, 35, 40, 25, 8, 7, 55, 50, 66, 65, 80

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

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

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

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

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

Q16. Explain Direct File and Indexed Sequential File Organization.

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

```
#include <stdio.h>

struct Term {
    int coeff;
    int exp;
};

void multiplyPolynomials(struct Term poly1[], int terms1, struct Term
poly2[], int terms2, struct Term result[], int *termsResult) {
    *termsResult = 0;
    for (int i = 0; i < terms1; i++) {
        for (int j = 0; j < terms2; j++) {
            result[*termsResult].coeff = poly1[i].coeff * poly2[j].coeff;
            result[*termsResult].exp = poly1[i].exp + poly2[j].exp;
            (*termsResult)++;
        }
    }
}

void displayPolynomial(struct Term poly[], int terms) {
    for (int i = 0; i < terms; i++) {
        printf("%dx^%d", poly[i].coeff, poly[i].exp);
        if (i != terms - 1) {
            printf(" + ");
        }
    }
    printf("\n");
}

int main() {
    struct Term poly1[2] = {{1, 1}, {2, 0}};
    struct Term poly2[2] = {{1, 1}, {3, 0}};
    struct Term result[4];
    int termsResult;

    multiplyPolynomials(poly1, 2, poly2, 2, result, &termsResult);

    printf("Resultant polynomial after multiplication:\n");
    displayPolynomial(result, termsResult);

    return 0;
}
```

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

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

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

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

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

int main() {
    struct Node* head = NULL;

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);

    printf("List in reverse order: ");
    displayReverse(head);
    printf("\n");

    return 0;
}
```

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

⟶

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

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

// Insert at the end of the doubly linked list
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
    }

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

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

```
int main() {
    struct Node* head = NULL;

    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);

    printf("Doubly Linked List: ");
    displayList(head);

    return 0;
}
```

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

Dequeue (Double-Ended Queue):

A **dequeue**, also known as a double-ended queue, is a data structure where elements can be inserted or removed from both the front and rear ends. This provides greater flexibility compared to a standard queue, where insertion happens at the rear and deletion occurs at the front. Dequeues are useful in applications where elements need to be added or removed from both ends, such as job scheduling or caching systems.

Types of Dequeues:

1. **Input-restricted Dequeue**: Insertion happens only at one end, while deletion can occur from both ends.
2. **Output-restricted Dequeue**: Deletion occurs only at one end, while insertion can occur from both ends.

Insert Operation in Dequeue:

1. Check if the Dequeue is full.
2. If not, increment the front or rear pointer depending on where the insertion is to occur.
3. Insert the element.

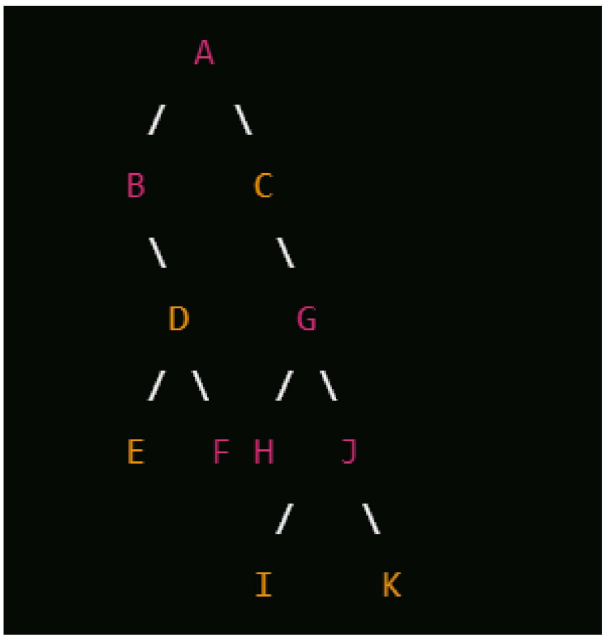
Delete Operation in Dequeue:

1. Check if the Dequeue is empty.
2. If not, increment/decrement the front or rear pointer to delete the element.
3. Adjust the pointers accordingly.

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

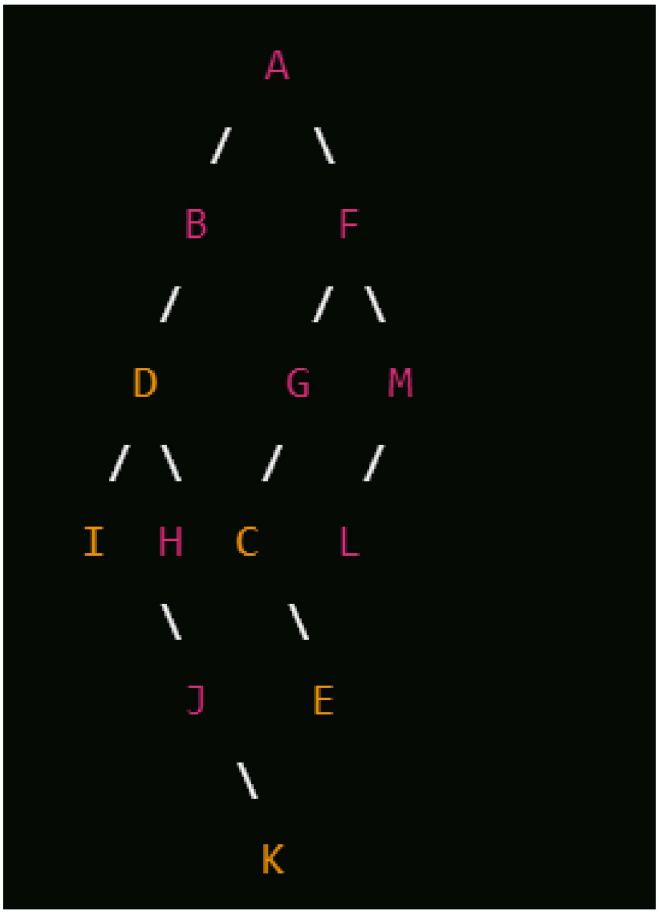
(i) **Preorder:** A B D E F C G H I J K

Inorder: B E D F A C I H K J G



(ii) **Postorder:** I J H D K E C L M G F B A

Inorder: 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 Postorder form.

→

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

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 = newNode->right = NULL;
    return newNode;
}

// Function to insert a new node in the BST
struct Node* insert(struct Node* node, int data) {
    if (node == NULL) return createNode(data);

    if (data < node->data)
        node->left = insert(node->left, data);
    else if (data > node->data)
        node->right = insert(node->right, data);

    return node;
}

// Inorder Traversal (Left, Root, Right)
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

```
// Preorder Traversal (Root, Left, Right)
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

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

int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");

    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

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.

Answer:

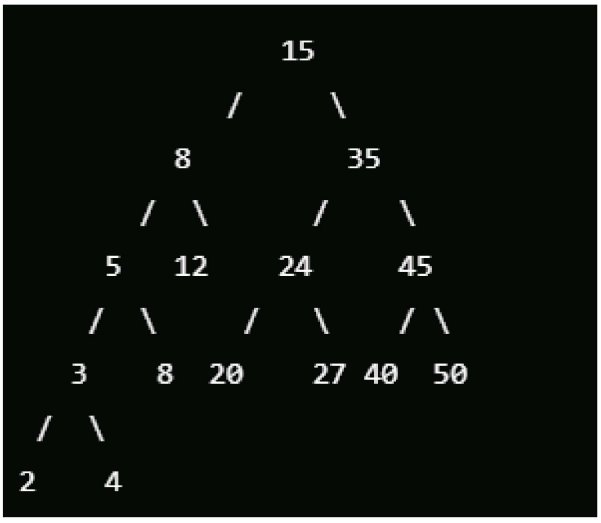
An **AVL tree** is a self-balancing binary search tree where the difference between the heights of the left and right subtrees (the balance factor) of any node is at most 1. If this balance factor is violated, rebalancing is performed using rotations (left rotation, right rotation, or a combination).

To construct the AVL tree and perform deletions, we follow the following steps:

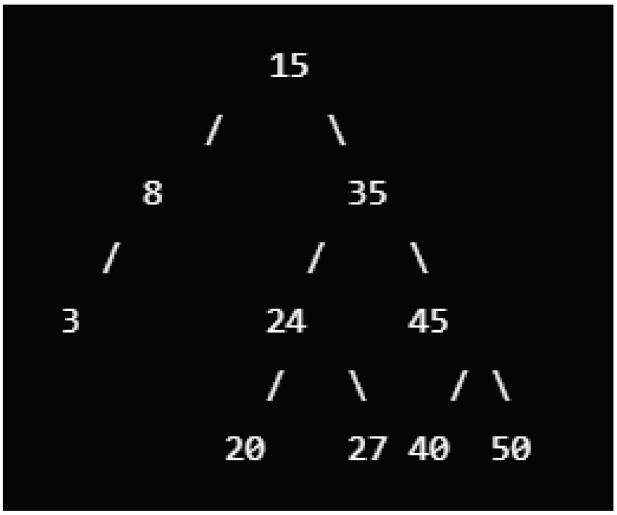
Step 1: Insert the elements in the given order:

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

After inserting these elements, the AVL tree looks like this (with necessary rotations applied during insertion):



Step 2: Perform deletions of nodes `2, 4, 5, and 12`. After performing the deletions, the AVL tree rebalances itself and will look like this:



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.

Answer:

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. A B-tree of order `m` has the following properties:

1. Each node can contain a maximum of `m-1` keys and a minimum of `⌈m/2⌉ - 1` keys.
2. Each internal node can have at most `m` children.
3. All leaves appear on the same level.
4. The root can have as few as one key.

Steps to Construct a B-tree of Order 5: We insert the following elements in the sequence provided:
`12, 5, 15, 20, 60, 45, 35, 40, 25, 8, 7, 55, 50, 66, 65, 80`

Initial Insertions:

1. The first five elements (`12, 5, 15, 20, 60`) fill the root node.
2. On inserting the sixth element (`45`), the root node splits, and the median value `20` becomes the new root.
3. The process continues, and after all elements are inserted, the B-tree will look like this:

markdown Copy code

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

Step 1: Deleting `5`:

- `5` is a leaf node in the leftmost child of the root, so it can be directly removed.

After deletion:

```markdown
            20
          /    \
    8, 12, 15   35, 45, 60
   /     |     /   |   |   \
  7      8    25  40  50  55, 66, 65, 80
```

Step 2: Deleting `12`:

- `12` is in the left child of the root. As it is a leaf, it is deleted, and the remaining values are adjusted.

After deletion:

```markdown
            20
          /    \
     8, 15      35, 45, 60
   /   |      /   |   |   \
  7    8     25  40  50  55, 66, 65, 80
```

Step 3: Deleting `8`:

- `8` is in the left child of the root and is also a leaf node. It is removed, and the keys are shifted accordingly.

After deletion:

```markdown
            20
          /    \
     7, 15      35, 45, 60
       |      /   |   |   \
       8     25  40  50  55, 66, 65, 80
```

Step 4: Deleting `20`:

- `20` is the root. After deleting it, the tree reorganizes itself, promoting `25` from the left subtree.

After deletion and restructuring:

markdown Copy code

```
            25
          /    \
     7, 15      35, 45, 60
       |       /   |   |   \
       8      40  50  55, 66, 65, 80
```

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

Applying Dijkstra's Algorithm to the Given Graph:

- Start at vertex `S`. Initialize the distances:
  - Distance to `S` = 0, all others = ∞.
- **Step-by-step Calculation:**
  - From `S`, visit adjacent vertices:
    - To `A`: Distance is `3` (0 + 3)
    - To `C`: Distance is `5` (0 + 5)
  - Mark `S` as visited.
  - Now pick the next unvisited vertex with the smallest distance (`A` with distance 3).
    - To `B`: Distance is `3 + 1 = 4`
    - To `D`: Distance is `3 + 7 = 10`
  - Mark `A` as visited.
  - Next, pick vertex `B` (distance 4).
    - To `D`: Distance is `4 + 4 = 8` (this is shorter than the previous 10)
  - Mark `B` as visited.
  - Next, pick vertex `C` (distance 5).
    - To `D`: Distance is `5 + 2 = 7` (this is shorter than the previous 8)
  - Mark `C` as visited.
  - Finally, pick `D` (distance 7).

Final Shortest Paths:

- `S -> S`: Distance = 0
- `S -> A`: Distance = 3
- `S -> B`: Distance = 4
- `S -> C`: Distance = 5
- `S -> D`: Distance = 7

Q10. **Apply Prim's Algorithm to find the minimum spanning tree (MST) for the given graph.**

Applying Prim's Algorithm to the Given Graph:

- Step-by-step Calculation:
  - Start at vertex `0` (arbitrary choice).
    - Pick the smallest edge: `0 - 1` (weight 4).
  - From `1`, pick the smallest edge to an unvisited vertex: `1 - 2` (weight 8).
  - From `2`, pick the smallest edge: `2 - 5` (weight 7).
  - From `5`, pick the smallest edge: `5 - 6` (weight 2).
  - From `6`, pick the smallest edge: `6 - 7` (weight 1).
  - From `7`, pick the smallest edge: `7 - 8` (weight 7).
  - From `8`, pick the smallest edge: `8 - 3` (weight 14).
  - From `3`, pick the smallest edge: `3 - 4` (weight 9).

Final MST:

- The edges included in the MST are:
  - `0 - 1` (weight 4)
  - `1 - 2` (weight 8)
  - `2 - 5` (weight 7)
  - `5 - 6` (weight 2)
  - `6 - 7` (weight 1)
  - `7 - 8` (weight 7)
  - `8 - 3` (weight 14)
  - `3 - 4` (weight 9)

Total Weight of the MST:

4 + 8 + 7 + 2 + 1 + 7 + 14 + 9 = 52

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

List to sort:

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

Insertion Sort:

Insertion sort builds the final sorted array one item at a time. It is much less efficient on large lists than other sorting algorithms.

Steps:

1. Start with the second element (5) and compare it with the first element (12). Since 5 is smaller, swap them.
2. Move to the third element (2), compare with the sorted part of the array (5, 12), and insert it in its correct position.
3. Continue this process for the rest of the list.

Sorted list after insertion sort: `2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 45, 50`

Time Complexity:

- Best case: O(n) (Already sorted)
- Worst case: O(n^2) (Reversed order)
- Average case: O(n^2)

Selection Sort:

Selection sort repeatedly finds the minimum element from the unsorted part and moves it to the beginning.

Steps:

1. Find the smallest element in the list (2) and swap it with the first element (12).
2. Then, find the second smallest element (3) and swap it with the second element (5).
3. Continue this process until the entire list is sorted.

Sorted list after selection sort: `2, 3, 5, 7, 8, 12, 15, 17, 25, 30, 45, 50`

Time Complexity:

- Best case: O(n^2)
- Worst case: O(n^2)
- Average case: 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 heap sort with the help of the thus created heap tree.

Answer:

Heap Tree: A **heap** is a special tree-based data structure that satisfies the heap property:

- **Max Heap**: Every parent node has a value greater than or equal to its child nodes.
- **Min Heap**: Every parent node has a value smaller than or equal to its child nodes.

In a **max heap**, the largest element is always at the root, and each node has a greater value than its children.

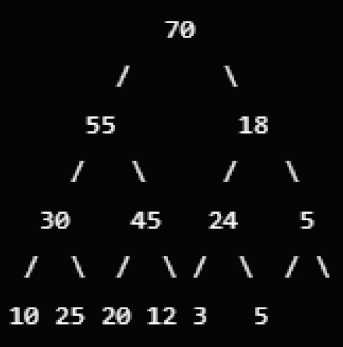
Steps to Build a Max Heap:

Given list: `10, 20, 5, 25, 30, 18, 3, 70, 55, 45, 12, 24`

1. Insert `10` → Heap: `[10]`
2. Insert `20` → Heap: `[20, 10]` (Swap 20 and 10)
3. Insert `5` → Heap: `[20, 10, 5]`
4. Insert `25` → Heap: `[25, 20, 5, 10]` (Swap 25 with 20)
5. Insert `30` → Heap: `[30, 25, 5, 10, 20]` (Swap 30 with 25)
6. Insert `18` → Heap: `[30, 25, 18, 10, 20, 5]`
7. Insert `3` → Heap: `[30, 25, 18, 10, 20, 5, 3]`
8. Insert `70` → Heap: `[70, 30, 18, 25, 20, 5, 3, 10]` (Swap 70 with 30)
9. Insert `55` → Heap: `[70, 55, 18, 30, 20, 5, 3, 10, 25]` (Swap 55 with 30)
10. Insert `45` → Heap: `[70, 55, 18, 30, 45, 5, 3, 10, 25, 20]` (Swap 45 with 20)
11. Insert `12` → Heap: `[70, 55, 18, 30, 45, 12, 3, 10, 25, 20, 5]`
12. Insert `24` → Heap: `[70, 55, 18, 30, 45, 24, 3, 10, 25, 20, 5, 12]`

Max Heap Representation:

markdown Copy code



Heap Sort:

Heap sort involves the following steps:

1. Build a max heap from the given array.
2. Swap the root (largest element) with the last element of the heap and reduce the size of the heap by 1.
3. Restore the max heap property by heapifying the root.
4. Repeat the process until the heap size becomes 1.

Time Complexity:

- Building the heap: O(n)
- Heapifying: O(log n) per element
- **Overall Time Complexity:** O(n log n)

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

```
#include <stdio.h>

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

```
void printArray(int A[], int size) {
    for (int i = 0; i < size; i++)
        printf("%d ", A[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;
}
```

Q14. What is Splay Tree? Explain the Zig-Zag and Zag-Zig rotations in Splay Tree with the help of a suitable example.

Answer:

A **Splay Tree** is a self-adjusting binary search tree that performs splaying – a process where the recently accessed node is moved to the root through a series of tree rotations. This ensures that frequently accessed elements are quick to reach.

Zig-Zag and Zag-Zig Rotations:

1. Zig-Zag Rotation:
    - Occurs when the node is a right child of a left child or a left child of a right child.
    - A double rotation is performed to move the node to the top.

    Example: Consider a splay tree where node X is the right child of the left child of its grandparent:

```
    G
   /
  P
   \
    X
```

After Zig-Zag rotation:

```
  X
 / \
P   G
```

2. **Zag-Zig Rotation:**
   - Occurs when the node is a left child of a right child or a right child of a left child.
   - Similar to Zig-Zag, a double rotation is performed.

Example: Consider a splay tree where node X is the left child of the right child of its grandparent:

```css
  G
   \
    P
   /
  X
```

After Zag-Zig rotation:

```css
    X
   / \
  G   P
```

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

Answer:

A **Red-Black Tree** is a self-balancing binary search tree where each node has an extra bit that represents "color" (either red or black). The tree maintains balance through certain properties and rotations during insertions and deletions.

Properties of a Red-Black Tree:

1. Each node is either red or black.
2. The root is always black.
3. Every leaf (NIL) is black.
4. Red nodes cannot have red children (no two consecutive red nodes).
5. Every path from a node to its descendant NIL nodes contains the same number of black nodes.

Insertion in Red-Black Tree:

- A new node is always inserted as a red node.
- If the parent is black, no further action is needed.
- If the parent is red, the tree is restructured using rotations (left or right) and recoloring to maintain the Red-Black properties.

Example: Insert the following elements: `10, 20, 30, 40, 50`.

1. Insert `10` as the root (black).
2. Insert `20` (red).
3. Insert `30` – causes a violation (two consecutive red nodes). A left rotation is performed at `20`, and recoloring is done.

The final tree after insertions will be balanced.

Deletion in Red-Black Tree:

When deleting a black node, it can lead to violations of the Red-Black properties. To handle this, we use a series of recoloring and rotations to restore the balance of the tree.

Steps for Deletion:

1. **Case 1: Node is Red**
   If the node to be deleted is red, it is simply removed without affecting the tree's balance.
2. **Case 2: Node is Black**
   If the node is black, its removal might violate the black-height property. Several cases are handled:
   - **Sibling Case:** If the deleted node's sibling is red, we perform a rotation and recolor to balance the tree.
   - **Black Sibling with Black Children:** We recolor the sibling and move the issue up the tree.
   - **Black Sibling with at Least One Red Child:** We perform appropriate rotations and recoloring to restore balance.

Example: Let's consider a Red-Black Tree with nodes: `30, 20, 40, 10, 25`.

1. Insertions:

```scss
        30(B)
      /     \
   20(R)    40(B)
  /   \
10(B) 25(B)
```

2. Delete Node `20`:
   - When `20` (red) is removed, the tree remains balanced without any additional steps.

After deletion:

```css
        30(B)
      /     \
   25(B)    40(B)
  /
10(B)
```

Q16. Explain Direct File and Indexed Sequential File Organization.

Answer:

1. Direct File Organization:

Direct file organization, also known as **hash file organization**, allows records to be retrieved directly using a hash function. In this organization, a key field is passed through a hashing algorithm, and the result of the algorithm gives the address where the record is stored.

- **Advantages:**
  - Fast access: Records can be retrieved quickly without needing to search sequentially.
  - Efficient for applications that require frequent data lookups.
- **Disadvantages:**
  - If the hash function generates the same address for multiple keys (collision), techniques like chaining are used to handle them, which can slow down retrieval.
  - Not ideal for sequential data access.

Example: If we have a key `123`, and our hash function is `key mod 10`, the record with key `123` will be stored at address `3`.

2. Indexed Sequential File Organization:

Indexed sequential file organization combines both sequential and direct access. Records are stored sequentially in the file, but an index is maintained that allows direct access to blocks of records. This approach provides the benefits of both sequential and random access.

- **Advantages:**
  - Fast search using the index, as it allows both direct and sequential access.
  - Useful for applications where data is frequently accessed in both random and sequential order.
- **Disadvantages:**
  - Requires additional space for storing the index.
  - More complex to maintain, especially when records are added or deleted, as the index must be updated.

Example: Consider a file containing employee records sorted by employee ID. An index might store the starting address of each block of 100 records. To find employee ID 1500, we look up the index to find the block where the employee's record is stored and then search sequentially within that block.