Section: A

1. **20 friends put their wallets in a row. The first wallet contains 20 dollars, the second has 30 dollars, the third has 40 dollars, and so on, with each wallet having 10 dollars more than the previous one. Since the data is already sorted in ascending order, no sorting is required. But if you are given a chance to sort the wallets, which sorting technique would be best to apply? Write a C++ program to implement your chosen sorting approach.**

   Ans:

   Theory:
   The numbers are already sorted. Best is to check if sorted and not sort again. If sorting is required, Insertion Sort is best because it is very fast for sorted data.

   Code:

```
#include <iostream>
using namespace std;

  int main()
 { int a[20];
   for (int i = 0; i < 20; i++) a[i] = 20 + i*10;

   bool sorted = true;
   for (int i = 1; i < 20; i++) {
      if (a[i] < a[i-1]) { sorted = false;
   break; } }

   if (sorted) {
      cout << "Array is already
   sorted\n"; } else {
      for (int i = 1; i < 20; i++)
         { int key = a[i];
         int j = i-1;
         while (j >= 0 && a[j] > key)
            { a[j+1] = a[j];
            j--;
```

```
        }
        a[j+1] =
    key; }
    cout << "Sorted array: ";
    for (int i = 0; i < 20; i++) cout << a[i] << "
"; }
  return
0; }
```

Explanation:

We first check if array is sorted. If yes, we do nothing. If no, we use insertion sort.

1. Array Initialization

- An integer array named a is declared with a size of 20.
- A for loop iterates from i = 0 to 19.
- In each iteration, the element a[i] is assigned the value 20 + i * 10. This will result in the array being initialized with the following values:
    - a[0] = 20 + 0 * 10 = 20
    - a[1] = 20 + 1 * 10 = 30
    - ...
    - a[19] = 20 + 19 * 10 = 210

2. Checking if Sorted

- A boolean variable sorted is initialized to true. This variable will track whether the array is currently sorted.
- Another for loop iterates from i = 1 to 19. This loop compares each element with its preceding element.
- Inside the loop, if (a[i] < a[i-1]) checks if the current element is smaller than the previous one.
    - If this condition is met at any point, it means the array is **not sorted**. The sorted variable is set to false, and the break statement immediately exits this loop, as further checks are unnecessary.

3. Sorting (if needed)

- An if (sorted) block checks the value of the sorted variable.
    - If sorted is true (meaning the array was already sorted), a message "Array is already sorted" is printed.
    - If sorted is false (meaning the array needs sorting), the code proceeds to the else block.

Insertion Sort Algorithm

- The else block implements the **insertion sort** algorithm.

- The outer for loop iterates from i = 1 to 19. This loop considers each element as a potential "key" to be inserted into its correct position within the already sorted portion of the array (elements from index 0 to i-1).
- int key = a[i]; stores the current element to be inserted.
- int j = i - 1; initializes an index j to the last element of the sorted portion.
- The while (j >= 0 && a[j] > key) loop performs the insertion:
    - It continues as long as j is a valid index (non-negative) AND the element at a[j] is greater than the key.
    - a[j+1] = a[j]; shifts the element a[j] one position to the right to make space for the key.
    - j--; moves j one step to the left to compare the key with the next preceding element.
- a[j+1] = key; once the while loop finishes, j+1 is the correct position for the key, and it's inserted there.

4. Output

- After the sorting process (if it occurred), the code prints "Sorted array: ".
- A final for loop iterates through the now-sorted array and prints each element followed by a space.

2. In a park, 10 friends were discussing a game based on sorting. They placed their wallets in a row. The maximum money in any wallet is $6. Among them, 3 wallets contain exactly $2, 2 wallets contain exactly $3, 2 wallets are empty ($0), 1 wallet contains $1, and 1 wallet contains $4. Which sorting technique would you apply to sort the wallets on the basis of the money they contain? Write a program to implement your chosen sorting technique.

Ans:

Theory:
When numbers are small and repeated, Counting Sort is best. It counts how many times each number appears and then prints them in order.

Code:

```
#include <iostream>
using namespace std;

int main() {
   int a[10] = {2,0,3,2,0,1,3,4,0,0};
   int maxVal = 6;
   int count[7] = {0};
```

```
for (int i = 0; i < 10; i++) count[a[i]]++;

cout << "Sorted wallets: ";
for (int i = 0; i <= maxVal; i++) {
```

```
        for (int j = 0; j < count[i]; j++)
            { cout << i << " ";
        }
    }
    return
0; }
```

Explanation:

We count each number and then print them. This gives sorted result quickly.

Initialization

- int a[10] = {2, 0, 3, 2, 0, 1, 3, 4, 0, 0};: This declares and initializes an integer array a of size 10 with the given unsorted values.
- int maxVal = 6;: This variable stores the maximum possible value in the input array. Although the highest value in the given array a is 4, the code uses 6, which indicates the maximum *range* of possible values.
- int count[7] = {0};: A new integer array count of size 7 is declared and initialized with all zeros. This array will be used to store the frequency of each number from 0 to 6 (a size of maxVal + 1 is needed).

2. Counting Frequencies

- for (int i = 0; i < 10; i++) count[a[i]]++;: This loop iterates through the unsorted array a.
    - For each element a[i], it increments the value at the corresponding index in the count array. For example, when a[i] is 2, count[2] is incremented.
    - After this loop, the count array will be {4, 1, 2, 2, 1, 0, 0}.
    - count[0] is 4 (there are four 0s in a).
    - count[1] is 1 (one 1).
    - count[2] is 2 (two 2s).
    - count[3] is 2 (two 3s).
    - count[4] is 1 (one 4).
    - count[5] and count[6] are 0.

3. Printing the Sorted Array

- cout << "Sorted wallets: ";: Prints a descriptive string.
- for (int i = 0; i <= maxVal; i++) { ... }: This outer loop iterates from i = 0 to i = 6 (the range of possible values).
- for (int j = 0; j < count[i]; j++) { ... }: This inner loop runs count[i] times.
    - The value of i is printed count[i] times. For example, when i is 0, the inner loop runs 4

times (since count[0] is 4), printing "0 0 0 0 ".

- When i is 1, the inner loop runs once, printing "1 ".
- When i is 2, the inner loop runs twice, printing "2 2 ".

- This process continues for all values from 0 to 6, effectively printing the sorted numbers based on their counts.

3. **During a college fest, 12 students participated in a gaming competition. Each student's score was recorded as follows: 45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32, 76 The organizers want to arrange the scores in ascending order to decide the ranking of the players. Since the data set is unsorted and contains numbers spread across a wide range, the most efficient technique to apply here is Quick Sort. Write a C++ program to implement Quick Sort to arrange the scores in ascending order.**

Ans:

Theory:
The numbers are random. Quick Sort is best because it works fast on such data.

Code:

```
#include <iostream>

using namespace std;


int partitionArr(int a[], int low, int high) {
   int pivot = a[high];
   int i = low - 1;
   for (int j = low; j < high; j++) {
```

```
        if (a[j] <= pivot) {
            i++;
            int temp = a[i]; a[i] = a[j]; a[j] = temp;
        }
    }
    int temp = a[i+1]; a[i+1] = a[high]; a[high] = temp;
    return i+1;
}

void quickSort(int a[], int low, int high) {
    if (low < high) {
        int pi = partitionArr(a, low, high);
        quickSort(a, low, pi-1);
        quickSort(a, pi+1, high);
    }
}

int main() {
    int scores[12] = {45,12,78,34,23,89,67,11,90,54,32,76};
    quickSort(scores, 0, 11);
    cout << "Sorted scores: ";
    for (int i = 0; i < 12; i++) cout << scores[i] << " ";
    return 0;
}
```

Explanation:

Quick sort selects a pivot, puts smaller numbers on left and larger on right, and repeats.

main Function

- **Array Initialization:** An integer array scores of size 12 is declared and initialized with the following unsorted values: {45, 12, 78, 34, 23, 89, 67, 11, 90, 54, 32, 76}.
- **Calling quickSort:** The quickSort function is called with the scores array, the starting index 0 (low), and the ending index 11 (high). This initiates the sorting process.

- **Printing Sorted Array:** After quickSort completes, a loop iterates through the now-sorted scores array and prints each element to the console, preceded by the text "Sorted scores: ".

2. quickSort Function

This is the recursive function that performs the Quicksort.

- **Base Case:** if (low < high): This condition checks if the low index is less than the high index. If low is not less than high, it means the sub-array has 0 or 1 element, which is considered already sorted, and the function returns.
- **Partitioning:** int pi = partitionArr(a, low, high);: The partitionArr function is called to rearrange the elements in the sub-array defined by low and high. It selects a **pivot** element, places it at its correct sorted position, and arranges all elements smaller than the pivot to its left and all elements greater than the pivot to its right. The function returns the index (pi) of the pivot after partitioning.
- **Recursive Calls:**
    - quickSort(a, low, pi - 1);: Recursively calls quickSort on the sub-array to the **left** of the pivot (elements smaller than the pivot).
    - quickSort(a, pi + 1, high);: Recursively calls quickSort on the sub-array to the **right** of the pivot (elements larger than the pivot).

3. partitionArr Function

This function is the core of Quicksort, responsible for rearranging the array around a pivot.

- **Pivot Selection:** int pivot = a[high];: The last element of the current sub-array (a[high]) is chosen as the **pivot**.
- **Index Initialization:** int i = low - 1;: An index i is initialized to low - 1. This index will keep track of the boundary between elements smaller than or equal to the pivot and elements greater than the pivot.
- **Iterating and Swapping:**
    - for (int j = low; j < high; j++): This loop iterates through the sub-array from the low index up to (but not including) the high index (the pivot).
    - if (a[j] <= pivot): If the current element a[j] is less than or equal to the pivot:
        - i++;: The i index is incremented.
        - **Swap:** The elements a[i] and a[j] are **swapped**. This moves the smaller element a[j] to the left side of the partition (at index i).
- **Placing the Pivot:**
    - int temp = a[i + 1]; a[i + 1] = a[high]; a[high] = temp;: After the loop finishes, the pivot element (originally at a[high]) is swapped with the element at a[i + 1]. This places the pivot in its correct sorted position.
- **Return Pivot Index:** return i + 1;: The function returns the index of the pivot element (i + 1).

4. **A software company is tracking project deadlines (in days remaining to submit). The deadlines are: 25, 12, 45, 7, 30, 18, 40, 22, 10, 35. The manager wants to arrange the deadlines in ascending order to prioritize the projects with the least remaining time. For efficiency, the project manager hints to the team to apply a divide-and-conquer technique that divides the array into unequal parts. Write a C++ program to sort the project deadlines using the above sorting technique.**

Ans:

Theory:
Divide-and-conquer with unequal parts is Quick Sort. It divides array into two parts and sorts.

Code:

```
#include <iostream>
using namespace std;

int partitionArr(int a[], int low, int high) {
   int pivot = a[high];
   int i = low - 1;
   for (int j = low; j < high; j++) {
      if (a[j] <= pivot) {
         i++;
         int temp = a[i]; a[i] = a[j]; a[j] = temp;
      }
   }
   int temp = a[i+1]; a[i+1] = a[high]; a[high] = temp;
   return i+1;
```

```
}

void quickSort(int a[], int low, int high) {
   if (low < high) {
      int pi = partitionArr(a, low, high);
      quickSort(a, low, pi-1);
      quickSort(a, pi+1, high);
   }
}

int main() {
   int deadlines[10] = {25,12,45,7,30,18,40,22,10,35};
   quickSort(deadlines, 0, 9);
   cout << "Sorted deadlines: ";
   for (int i = 0; i < 10; i++) cout << deadlines[i] << " ";
   return 0;
}
```

This C++ code implements the **Quick Sort** algorithm to sort an array of integers in ascending order. Here's a step-by-step explanation of how it works:

1. main Function: Initialization and Triggering the Sort

- int deadlines[10] = {25, 12, 45, 7, 30, 18, 40, 22, 10, 35};: An integer array named deadlines is declared and initialized with 10 unsorted values.
- quickSort(deadlines, 0, 9);: This is the main call that initiates the sorting process. It tells the quickSort function to sort the deadlines array from index 0 to index 9 (inclusive).
- The code then prints "Sorted deadlines:" to the console.
- The subsequent for loop iterates through the now sorted deadlines array and prints each element, followed by a space.

2. quickSort Function: The Recursive Sorting Logic

- void quickSort(int a[], int low, int high): This function takes the array a and the low and high indices defining the current subarray to be sorted.
- if (low < high): This is the **base case** for the recursion. If low is not less than high, it means the subarray has 0 or 1 element, which is already considered sorted, so the function returns.
- int pi = partitionArr(a, low, high);: This is the core of Quick Sort. It calls the partitionArr function to:

    - **Choose a pivot element** (in this implementation, it's the last element of the subarray).
    - **Rearrange the subarray** such that all elements less than or equal to the pivot are placed before it, and all elements greater than the pivot are placed after it.
    - The partitionArr function returns the **pivot's final index** (pi).
- quickSort(a, low, pi - 1);: This line recursively calls quickSort on the **left subarray**, which includes all elements from the low index up to (but not including) the pivot's index (pi - 1).
- quickSort(a, pi + 1, high);: This line recursively calls quickSort on the **right subarray**, which includes all elements from the index immediately after the pivot (pi + 1) up to the high index.

3. partitionArr Function: Rearranging Elements Around a Pivot

- int partitionArr(int a[], int low, int high): This function takes the array a and the low and high indices of the subarray.
- int pivot = a[high];: The last element of the subarray (a[high]) is chosen as the **pivot**.
- int i = low - 1;: An index i is initialized to low - 1. This i will keep track of the **rightmost boundary of elements smaller than or equal to the pivot**.
- for (int j = low; j < high; j++): This loop iterates through the subarray from low up to (but not including) high.
    - if (a[j] <= pivot): If the current element a[j] is less than or equal to the pivot:
        - i++;: Increment i.
        - The elements at a[i] and a[j] are **swapped**. This moves the smaller element (a[j]) to the left side of the partition (where elements less than or equal to the pivot belong).
- **Final Pivot Placement**: After the loop finishes, all elements from low to i are less than or equal to the pivot. The pivot itself is still at a[high].
    - int temp = a[i + 1]; a[i + 1] = a[high]; a[high] = temp;: The pivot element (a[high]) is swapped with the element at a[i + 1]. This places the pivot in its correct sorted position.
- return i + 1;: The function returns the index where the pivot was placed (i + 1).

Explanation:
Quick sort is used. After sorting, the nearest deadlines will appear first.

5. Suppose there is a square named SQ-1. By connecting the midpoints of SQ-1, we create another square named SQ-2. Repeating this process, we create a total of 50 squares {SQ 1, SQ-2, …, SQ-50}. The areas of these squares are stored in an array. Your task is to search whether a given area is present in the array or not. What would be the best searching approach? Write a C++ program to implement

this approach.

Ans:

Theory:
If data is sorted, the best search is Binary Search. It checks the middle, then goes left or right. It is faster than linear search.

Code:

```
#include <iostream>
using namespace std;

int binarySearch(double a[], int n, double key) {
    int l = 0, r = n - 1;
    while (l <= r) {
        int mid = (l + r) / 2;
        if (a[mid] == key) return mid;
        else if (a[mid] < key) l = mid + 1;
        else r = mid - 1;
    }
    return -1;
}

int main() {
    double areas[5] = {100, 80, 60, 40, 20}; // example values
    int n = 5;

    double key;
    cout << "Enter area to search: ";
    cin >> key;
```

```
    int pos = binarySearch(areas, n, key);
    if (pos != -1) cout << "Area found at position " << pos << endl;
    else cout << "Area not found" << endl;

    return 0;
}
```

Explanation:
We use binary search on the array of areas. It checks the middle element and reduces the search size each step.

Initialization and Input

- double areas[5] = {100, 80, 60, 40, 20};: An array named areas is declared and initialized with five double values. **Crucially, binary search requires the array to be sorted.** In this example, the array is sorted in descending order.
- int n = 5;: The variable n stores the number of elements in the areas array.
- double key;: A variable key is declared to store the value the user wants to search for.
- cout << "Enter an area to search:"; cin >> key;: The program prompts the user to enter an area and stores their input in the key variable.

2. The binarySearch Function

This function performs the core logic of the binary search.

- int binarySearch(double a[], int n, double key):
    - a[]: The array to search within.
    - n: The number of elements in the array.
    - key: The value to find.
- int l = 0, r = n - 1;: Two index variables, l (left) and r (right), are initialized. l points to the beginning of the array, and r points to the end.
- while (l <= r): The loop continues as long as the left index is less than or equal to the right index. This means there's still a portion of the array to search.
- int mid = (l + r) / 2;: The mid index is calculated as the average of l and r. This effectively divides the search space in half.
- if (a[mid] == key) return mid;: If the element at the mid index is equal to the key, the function has found the value and returns the mid index.
- else if (a[mid] < key) l = mid + 1;: If the element at mid is *less than* the key, it means the key (if it exists) must be in the right half of the current search space (because the array is sorted in descending order). So, the l index is updated to mid + 1.
- else r = mid - 1;: If the element at mid is *greater than* the key, it means the key (if it exists) must be in the left half of the current search space. So, the r index is updated to mid - 1.

- return -1;: If the while loop finishes without finding the key (i.e., l becomes greater than r), it means the key is not present in the array, and the function returns -1.

3. Output

- int pos = binarySearch(areas, n, key);: The binarySearch function is called with the areas array, its size n, and the user-provided key. The returned index is stored in pos.
- if (pos != -1) cout << "Area found at position" << pos << endl;: If pos is not -1 (meaning the key was found), the program prints the position where it was found.
- else cout << "Area not found" << endl;: If pos is -1 (meaning the key was not found), the program prints a "not found" message.

6. Before a match, the chief guest wants to meet all the players. The head coach introduces the first player, then that player introduces the next player, and so on, until all players are introduced. The chief guest moves forward with each introduction, meeting the players one at a time. How would you implement the above activity using a Linked List? Write a C++ program to implement the logic.

Ans:

Theory:

A singly linked list is best. Each node stores a player name and a pointer to the next player.

Code:

```
#include <iostream>

using namespace std;


struct Node {
   string name;
   Node* next;
```

```
    Node(string s) { name = s; next = NULL; }
};

int main() {
    string players[5] =
{"Player1","Player2","Player3","Player4","Player5"};

    Node* head = NULL;
    Node* tail = NULL;

    for (int i = 0; i < 5; i++) {
        Node* node = new Node(players[i]);
        if (head == NULL) head = tail = node;
        else {
            tail->next = node;
            tail = node;
        }
    }

    cout << "Chief guest meets players:\n";
    Node* cur = head;
    while (cur != NULL) {
        cout << cur->name << endl;
        cur = cur->next;
    }
    return 0;
}
```

This C++ code implements a **singly linked list** to store player names and then iterates through the list to print them. Here's the procedure:

1. Setup and Initialization

- **Include Headers:** The code starts by including <iostream> for input/output operations and using

namespace std; to avoid repeatedly typing std::.

- **Node Structure:**
    - A struct Node is defined to represent an element in the linked list.
    - Each Node contains:
        - string name: Stores the player's name.
        - Node* next: A pointer to the next Node in the list.
    - A constructor Node(string s) is provided to easily create a new node with a given name, initializing next to NULL.
- **Main Function:** The main function is the entry point of the program.
- **Player Names:** string players[5] = {"Player1", "Player2", "Player3", "Player4", "Player5"}; declares an array holding the names of five players.
- **List Pointers:**
    - Node* head = NULL;: A pointer head is initialized to NULL, indicating that the list is initially empty. This pointer will point to the first node.
    - Node* tail = NULL;: A pointer tail is also initialized to NULL. This pointer will point to the last node in the list, making insertions at the end efficient.

2. Building the Linked List

- **Loop through Players:** A for loop iterates five times (from i = 0 to 4) to process each player name from the players array.
- **Create New Node:** Inside the loop, Node* node = newNode(players[i]); creates a new Node object using the current player's name. (Note: The code provided has newNode but it's not defined. Assuming it's a typo and should be new Node(players[i]) as per the struct definition.)
- **Handle Empty List:**
    - if (head == NULL): This condition checks if the list is currently empty.
    - If it is empty, both head and tail are set to point to the newly created node, making it the first and only node in the list.
- **Append to Non-Empty List:**
    - else: If the list is not empty (meaning head is not NULL), this block executes.
    - tail->next = node;: The next pointer of the current last node (tail) is updated to point to the new node. This links the new node to the end of the list.
    - tail = node;: The tail pointer is then updated to point to the new node, as it is now the last node in the list.

3. Traversing and Printing

- **Output Header:** cout << "Chiefguest meets players:\n"; prints a message to the console.
- **Initialize Current Pointer:** Node* cur = head; creates a temporary pointer cur and initializes it to head. This pointer will be used to walk through the list.

- **Iterate and Print:**
    - while (cur != NULL): This loop continues as long as cur is not NULL, meaning we haven't reached the end of the list.
    - cout << cur->name << endl;: The name stored in the current node is printed to the console, followed by a newline character.
    - cur = cur->next;: The cur pointer is moved to the next node in the list by following the next pointer.

4. Program Termination

- return 0;: Indicates that the program executed successfully.

7. **A college bus travels from stop A → stop B → stop C → stop D and then returns in reverse order D → C → B → A. Model this journey using a doubly linked list. Write a program to:**
   • Store bus stops in a doubly linked list.
   • Traverse forward to show the onward journey.
   • Traverse backward to show the return journey.

   Theory:
   A doubly linked list allows forward and backward travel. Each node has next and prev pointers.

   Code:

```
#include <iostream>
using namespace std;

struct Node
   { string stop;
   Node* next;
   Node* prev;
   Node(string s) { stop = s; next = prev =
NULL; } };

int main() {
   string stops[4] = {"A","B","C","D"};
   Node* head = NULL;
   Node* tail = NULL;

   for (int i = 0; i < 4; i++) {
      Node* node = new Node(stops[i]);
      if (head == NULL) head = tail = node;
      else {
         tail->next = node;
         node->prev = tail;
         tail = node;
      }
   }

   cout << "Onward journey: ";
   Node* cur = head;
   while (cur != NULL) {
```

```
        cout << cur->stop << " ";
        cur = cur->next;
    }
    cout << endl;

    cout << "Return journey: ";
    cur = tail;
    while (cur != NULL)
        {   cout << cur->stop << "
        "; cur = cur->prev;
    }
    cout << endl;

    return
0; }
```

This C++ code implements and demonstrates a **doubly linked list**, specifically for managing a sequence of "stops" (like bus stops or train stations). Here's a procedure of how it works:

1. Node Structure Definition

- struct Node: This defines the building block of our linked list. Each Node contains:
    - string stop: Stores the data for this node (the name of a stop, e.g., "A", "B").
    - Node* next: A pointer to the *next* node in the sequence.
    - Node* prev: A pointer to the *previous* node in the sequence.

- Node(string s): This is the constructor for the Node structure. When a new Node is created, it's initialized with a given string s for its stop value, and both next and prev pointers are set to NULL, indicating it's not yet connected to other nodes.

2. Initialization

- string stops[4] = {"A", "B", "C", "D"};: An array of strings is created to hold the names of the stops that will be added to the linked list.
- Node* head = NULL;: A pointer named head is initialized to NULL. This pointer will eventually point to the first node in the list.
- Node* tail = NULL;: A pointer named tail is initialized to NULL. This pointer will eventually point to the last node in the list.

3. Building the Doubly Linked List

- for (int i = 0; i < 4; i++): This loop iterates through the stops array.
    - Node* node = newNode(stops[i]);: In each iteration, a new Node is created using the newNode function (which is likely a typo and should be new Node(stops[i]) as per standard C++ syntax for dynamic memory allocation of structures/classes) with the current stop name.
    - **First Node (head == NULL):** If head is NULL, it means the list is currently empty. So, the newly created node becomes both the head and the tail of the list.
    - **Subsequent Nodes:** If the list is not empty (head is not NULL):
        - tail->next = node;: The next pointer of the current tail node is made to point to the new node.
        - node->prev = tail;: The prev pointer of the new node is made to point back to the current tail. This establishes the backward link.
        - tail = node;: The tail pointer is updated to point to the newly added node, as it's now the last node in the list.

After this loop, a doubly linked list will be formed with head pointing to "A" and tail pointing to "D". Each node will be correctly linked to its predecessor and successor.

4. Traversing Forward (Onward Journey)

- cout << "Onward journey: ";: Prints a label for the output.
- Node* cur = head;: A temporary pointer cur is initialized to point to the head of the list.
- while (cur != NULL): This loop continues as long as cur is not NULL (meaning we haven't reached the end of the list).
    - cout << cur->stop << " ";: The stop value of the current node is printed.
    - cur = cur->next;: The cur pointer is moved to the next node in the list.
- This section prints the stops in forward order: "A B C D ".

5. Traversing Backward (Return Journey)

- cout << "Return journey: ";: Prints a label for the output.

- cur = tail;: The cur pointer is reset to point to the tail of the list.
- while (cur != NULL): This loop continues as long as cur is not NULL (meaning we haven't reached the beginning of the list).
    - cout << cur->stop << " ";: The stop value of the current node is printed.
    - cur = cur->prev;: The cur pointer is moved to the *previous* node in the list using the prev pointer.
- This section prints the stops in reverse order: "D C B A ".

6. Program Termination

- return 0;: Indicates that the program executed successfully.

8. There are two teams named Dalta Gang and Malta Gang. Dalta Gang has 4 members, and each member has 2 Gullaks (piggy banks) with some money stored in them. Malta Gang has 2 members, and each member has 3 Gullaks. Both gangs store their Gullak money values in a 2D array. Write a C++ program to:
• Display the stored data in matrix form.
• To multiply Dalta Gang matrix with Malta Gang Matrix

Theory:
Matrix multiplication: (4×2) × (2×3) = (4×3). Each cell is row × column multiplication.

Code:

```
#include <iostream>
using namespace std;

int main() {
   int Dalta[4][2] = {{10,20},{5,15},{12,8},{7,9}};
   int Malta[2][3] = {{2,3,4},{1,0,5}};
   int result[4][3] = {0};

   for (int i = 0; i < 4; i++)
      { for (int j = 0; j < 3; j++)
      {
         result[i][j] = 0;
```

```
            for (int k = 0; k < 2; k++) {
                result[i][j] += Dalta[i][k] *
            Malta[k][j]; }
        }
    }

    cout << "Result matrix (4x3):\n";
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 3; j++) cout << result[i][j] << " ";
        cout << endl;
    }
    return
0; }
```

Explanation:

We multiply each row of Dalta with each column of Malta to form a new 4×3 matrix.

Initialization:

- int Dalta[4][2] = {{10, 20}, {5, 15}, {12, 8}, {7, 9}};: A 4x2 matrix named Dalta is declared and initialized with specific integer values.
- int Malta[2][3] = {{2, 3, 4}, {1, 0, 5}};: A 2x3 matrix named Malta is declared and initialized.
- int result[4][3] = {0};: A 4x3 matrix named result is declared and initialized with all elements set to 0. This matrix will store the outcome of the multiplication.

2. **Matrix Multiplication Logic:**

- The code uses three nested loops to compute the product of Dalta and Malta.
- **Outer Loop (i):** This loop iterates through the rows of the result matrix (and Dalta matrix), from i = 0 to 3.
- **Middle Loop (j):** This loop iterates through the columns of the result matrix (and Malta matrix), from j = 0 to 2.
- **Inner Loop (k):** This loop is the core of the multiplication. It iterates through the columns of Dalta and the rows of Malta, from k = 0 to 1.
    - Inside the inner loop, the operation result[i][j] += Dalta[i][k] * Malta[k][j]; is performed. This means:
        - For each element result[i][j], it takes the element from the i-th row and k-th column of Dalta and multiplies it by the element from the k-th row and j-th column of Malta.
        - This product is then added to the current value of result[i][j]. This process is repeated for all values of k, effectively summing up the products of corresponding elements.
- **Compatibility Check:** For matrix multiplication to be valid, the number of columns in the

first matrix (Dalta has 2 columns) must equal the number of rows in the second matrix (Malta has 2 rows). This condition is met. The resulting matrix will have the number of rows from the first matrix (4) and the number of columns from the second matrix (3).

3. **Output:**

    - cout << "Result matrix(4x3):\n";: Prints a header indicating the output.
    - The code then uses two nested loops to iterate through the result matrix.
    - cout << result[i][j] << " ";: Each element of the result matrix is printed, followed by a space.
    - cout << endl;: After each row is printed, a newline character is printed to format the output as a matrix.

The program will output the 4x3 result matrix, which is the product of Dalta and Malta.

Section: B

1. **To store the names of family members, an expert suggests organizing the data in a way that allows efficient searching, traversal, and insertion of new members. For this purpose, use a Binary Search Tree (BST) to store the names of family members, starting with the letters: Write a C++ program to Create a Binary Search Tree (BST) using the given names and find and display the successor of the family member whose name starts with M.**

Ans:

Theory:

In BST, successor means the next bigger element in in-order traversal. If node has right child → leftmost of right subtree. Else → go up.

Code:

```
#include <iostream>
```

```
using namespace std;

struct Node {
    string key;
    Node* left;
    Node* right;
    Node(string k) { key = k; left = right = NULL; }
};

Node* insert(Node* root, string k) {
    if (root == NULL) return new Node(k);
    if (k < root->key) root->left = insert(root->left, k);
    else root->right = insert(root->right, k);
    return root;
}

Node* minValue(Node* node) {
    while (node->left != NULL) node = node->left;
    return node;
}

Node* inorderSuccessor(Node* root, Node* n) {
    if (n->right != NULL) return minValue(n->right);
    Node* succ = NULL;
    while (root != NULL) {
        if (n->key < root->key) {
            succ = root;
```

```
            root = root->left;
        } else if (n->key > root->key) {
            root = root->right;
        } else break;
    }
    return succ;
}

Node* search(Node* root, string k) {
    if (root == NULL || root->key == k) return root;
    if (k < root->key) return search(root->left, k);
    else return search(root->right, k);
}

int main() {
    string arr[9] = {"Q","S","R","T","M","A","B","P","N"};
    Node* root = NULL;
    for (int i = 0; i < 9; i++) root = insert(root, arr[i]);

    Node* mnode = search(root, "M");
    Node* succ = inorderSuccessor(root, mnode);

    if (succ != NULL) cout << "Successor of M is: " << succ->key <<
endl;
    else cout << "No successor found" << endl;

    return 0;
}
```

This C++ code constructs a **Binary Search Tree (BST)** and then finds the **in-order successor** of a specific node. Here's a breakdown of its procedure:

1. Setting up the BST Structure

- **struct Node**: This defines the basic building block of the BST. Each Node contains:
    - string key: The data stored in the node.
    - Node* left: A pointer to the left child node.
    - Node* right: A pointer to the right child node.
    - Node(string k): A constructor to easily create a new node with a given key and initialize its children to NULL.

2. Building the BST

- **Node* insert(Node* root, string k)**: This function inserts a new key k into the BST.
    - **Base Case**: If the root is NULL (meaning the tree is empty or we've reached an empty spot), it creates a new node with key k and returns it.
    - **Recursive Insertion**:
        - If k is less than the current root->key, it recursively calls insert on the root->left subtree.
        - If k is greater than or equal to the current root->key, it recursively calls insert on the root->right subtree.
    - It returns the (potentially updated) root of the subtree.

3. Finding the Minimum Value in a Subtree

- **Node* minValue(Node* node)**: This helper function finds the node with the smallest key in a given subtree.
    - It repeatedly traverses to the left child until it reaches a node whose left child is NULL. This leftmost node contains the minimum value.

4. Finding the In-order Successor

- **Node* inorderSuccessor(Node* root, Node* n)**: This is the core function that finds the in-order successor of a given node n. The in-order successor is the node that comes immediately after n in an in-order traversal of the BST.
    - **Case 1: Node n has a right child**: If n->right is not NULL, the successor is the node with the **minimum value** in n's right subtree. This is found by calling minValue(n->right).
    - **Case 2: Node n does not have a right child**: In this case, the successor is one of n's ancestors. We need to traverse up from the root of the entire tree:
        - We search for node n starting from the root.
        - If n->key is less than the current root->key, it means the current root *could* be the successor, so we store it in succ and move to the root->left.
        - If n->key is greater than the current root->key, the successor must be further down the right subtree, so we move to root->right.
        - If n->key is equal to root->key, we've found n, and the loop breaks.

        - The succ variable will hold the lowest ancestor for which n is in its left subtree.

5. Searching for a Node

- **Node* search(Node* root, string k)**: This function searches for a node with key k in the BST.
    - **Base Cases**:
        - If root is NULL (tree is empty or the node isn't found) or root->key matches k, it returns the root.
    - **Recursive Search**:
        - If k is less than root->key, it recursively searches the left subtree.
        - Otherwise, it recursively searches the right subtree.

6. Main Function Execution

- **int main()**:
    - An array arr of strings is initialized: {"Q", "S", "R", "T", "M", "A", "B", "P", "N"}.
    - A Node* root is initialized to NULL.
    - The for loop iterates through arr, inserting each string into the BST using the insert function. The root pointer is updated after each insertion.
    - Node* mnode = search(root, "M");: The search function is called to find the node containing the key "M".
    - Node* succ = inorderSuccessor(root, mnode);: The inorderSuccessor function is called with the root of the tree and the pointer to the node "M" (mnode) to find its successor.
    - Finally, it checks if a successor was found (succ != NULL).
        - If found, it prints "Successor of M is: " followed by the successor's key.
        - If not found (meaning "M" was the largest element in the tree), it prints "No successor found".

In essence, the code first builds a BST from the given array, then locates the node with the key "M", and finally determines and prints the node that would immediately follow "M" if the tree's elements were listed in ascending alphabetical order.

2. **Implement the In-Order, Pre- Order and Post-Order traversal of Binary search tree with help of C++ Program.**

Ans:

Theory:

- Inorder → Left, Root, Right (gives sorted order)
- Preorder → Root, Left, Right
- Postorder → Left, Right, Root

Code:

```
#include <iostream>
using namespace std;

struct Node
   { int val;
   Node* left;
   Node* right;
   Node(int v) { val = v; left = right =
NULL; } };

Node* insert(Node* root, int v) {
   if (root == NULL) return new Node(v);
   if (v < root->val) root->left = insert(root->left, v);
   else root->right = insert(root->right, v);
   return
root; }

void inorder(Node* root)
   { if (root == NULL)
   return;
   inorder(root->left);
   cout << root->val << " ";
   inorder(root->right);
}

void preorder(Node* root)
   { if (root == NULL)
   return; cout << root->val
   << " ";
   preorder(root->left);
   preorder(root->right);
```

```
}

void postorder(Node* root)
   { if (root == NULL)
   return;
   postorder(root->left);
   postorder(root->right);
   cout << root->val << " ";
}

int main() {
   int vals[7] = {50,30,70,20,40,60,80};
   Node* root = NULL;
   for (int i = 0; i < 7; i++) root = insert(root, vals[i]);

   cout << "Inorder: "; inorder(root); cout << endl;
   cout << "Preorder: "; preorder(root); cout << endl;
   cout << "Postorder: "; postorder(root); cout << endl;

   return
0; }
```

This C++ code constructs a **Binary Search Tree (BST)** and then performs three types of tree traversals: **inorder, preorder, and postorder**.

1. Node Structure

- The Node structure defines the basic building block of the BST. Each node contains:
    - val: An integer value stored in the node.
    - left: A pointer to the left child node.
    - right: A pointer to the right child node.
    - The constructor Node(int v) initializes a new node with a given value and sets its left and right children to NULL.

2. Inserting into the BST

- The insert function recursively adds new values to the BST, maintaining the BST property:
    - If the root is NULL (the tree is empty), a new node with the given value v is created and returned as the new root.
    - If v is less than the current root->val, it's inserted into the left subtree by recursively calling insert on root->left.
    - Otherwise (if v is greater than or equal to root->val), it's inserted into the right subtree by recursively calling insert on root->right.

3. Tree Traversal Functions

- **Inorder Traversal (inorder)**:

    - This function visits the left subtree, then the current node, and finally the right subtree.
    - For a BST, inorder traversal visits the nodes in **ascending order**.
    - The process is recursive: if the root is not NULL, it calls inorder on the left child, prints the root->val, and then calls inorder on the right child.
- **Preorder Traversal (preorder)**:
    - This function visits the current node first, then the left subtree, and finally the right subtree.
    - The process is recursive: it prints the root->val, then calls preorder on the left child, and then calls preorder on the right child.
- **Postorder Traversal (postorder)**:
    - This function visits the left subtree, then the right subtree, and finally the current node.
    - The process is recursive: it calls postorder on the left child, then calls postorder on the right child, and finally prints the root->val.

4. Main Function (main)

- int vals[7] = {50, 30, 70, 20, 40, 60, 80};: An array of integers is defined.
- Node* root = NULL;: A pointer root is initialized to NULL, indicating an empty tree.
- The for loop iterates through the vals array and calls the insert function for each value to build the BST.
- Finally, it calls inorder, preorder, and postorder functions to print the values of the BST in their respective traversal orders, followed by a newline character for formatting.

3. Write a C++ program to search an element in a given binary search Tree.

Ans:

Theory:

In BST, to search:

- If value = root → found.
- If value < root → go left.
- If value > root → go right.

This repeats until found or tree ends.

Code:

```
#include <iostream>

using namespace std;
```

```
struct Node {
    int val;
    Node* left;
    Node* right;
    Node(int v) { val = v; left = right = NULL; }
};

Node* insert(Node* root, int v) {
    if (root == NULL) return new Node(v);
    if (v < root->val) root->left = insert(root->left, v);
    else root->right = insert(root->right, v);
    return root;
}

bool search(Node* root, int key) {
    if (root == NULL) return false;
    if (root->val == key) return true;
    if (key < root->val) return search(root->left, key);
    else return search(root->right, key);
}

int main() {
    int vals[10] = {45,12,78,34,23,89,67,11,90,54};
    Node* root = NULL;
    for (int i = 0; i < 10; i++) root = insert(root, vals[i]);
```

```
    int key;
    cout << "Enter value to search: ";
    cin >> key;

    if (search(root, key)) cout << key << " found\n";
    else cout << key << " not found\n";

    return 0;
}
```

Explanation:
We insert values into BST. Search checks left or right until the number is found.

The struct Node defines the building block of the BST. Each Node contains:

- val: An integer value stored in the node.
- left: A pointer to the left child node.
- right: A pointer to the right child node.

- The constructor Node(int v) initializes a new node with the given value v and sets its left and right pointers to NULL, indicating it has no children initially.

2. Insertion (insert function)

- This function recursively inserts a new value v into the BST.
- **Base Case:** If the root is NULL (meaning the tree or subtree is empty), it creates a new Node with value v and returns it as the new root.
- **Recursive Step:**
  - If v is **less than** the current root->val, it recursively calls insert on the **left subtree** (root->left). The result of this recursive call becomes the new left child of the current node.
  - If v is **greater than or equal to** the current root->val, it recursively calls insert on the **right subtree** (root->right). The result becomes the new right child.
- The function returns the (potentially modified) root of the subtree.

3. Search (search function)

- This function recursively searches for a given key within the BST.
- **Base Case 1:** If the root is NULL (meaning the tree or subtree is empty and the key wasn't found), it returns false.

- **Base Case 2:** If the root->val is equal to the key, the value is found, and the function returns true.
- **Recursive Step:**
    - If the key is **less than** root->val, it recursively calls search on the **left subtree** (root->left).
    - If the key is **greater than** root->val, it recursively calls search on the **right subtree** (root->right).
- The result of the recursive call is returned.

4. Main Function (main)

- An integer array vals of size 10 is initialized with sample values.
- A Node* root pointer is initialized to NULL, signifying an empty BST.
- A for loop iterates through the vals array, and for each value, it calls the insert function to add it to the BST. The root pointer is updated with the result of each insertion.
- The program prompts the user to "Enter value to search:".
- The entered key is read using cin.
- The search function is called with the root of the BST and the key.
- Based on the boolean result from search:
    - If true, it prints "key found".
    - If false, it prints "key not found".
- The program returns 0, indicating successful execution.

In essence: The code first builds a BST where smaller values are placed to the left of a node and larger values to the right. Then, when searching, it efficiently navigates through the tree, discarding half of the remaining nodes at each step, similar to a binary search on a sorted array.

4. In a university, the roll numbers of newly admitted students are: 45, 12, 78, 34, 23, 89, 67, 11, 90, 54 The administration wants to store these roll numbers in a way that allows fast searching, insertion, and retrieval in ascending order. For efficiency, they decide to apply a Binary Search Tree (BST). Write a C++ program to construct a Binary Search Tree using the above roll numbers and perform an in-order traversal to display them in ascending order.

Ans:

Theory:
In-order traversal of BST prints values in ascending order.

Code:

```
#include <iostream>
using namespace std;
```

```
struct Node {
    int val;
    Node* left;
    Node* right;
    Node(int v) { val = v; left = right = NULL; }
};

Node* insert(Node* root, int v) {
    if (root == NULL) return new Node(v);
    if (v < root->val) root->left = insert(root->left, v);
    else root->right = insert(root->right, v);
    return root;
}

void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->val << " ";
    inorder(root->right);
}

int main() {
    int rolls[10] = {45,12,78,34,23,89,67,11,90,54};
    Node* root = NULL;
    for (int i = 0; i < 10; i++) root = insert(root, rolls[i]);

    cout << "In-order (ascending): ";
```

```
    inorder(root);
    cout << endl;


    return 0;
}
```

Node Structure: A struct named Node is defined to represent a node in the BST. Each node contains an integer val and two pointers, left and right, to its child nodes. The constructor initializes a new node with a given value and sets the child pointers to NULL.

1. **insert Function**: This function recursively inserts a new value (v) into the BST.
    - **Base Case**: If the current root is NULL, a new Node is created and returned. This is where the new node is actually added.
    - **Recursive Step**: If the tree is not empty, the function compares the new value v with the root's value.
        - If v is less than root->val, the function calls itself on the left subtree (root->left = insert(root->left, v)).
        - Otherwise (v is greater than or equal to root->val), it calls itself on the right subtree (root->right = insert(root->right, v)).
2. **inorder Function**: This function performs an **in-order traversal** of the BST. This type of traversal visits nodes in a specific order: left child, parent, and then right child.
    - **Base Case**: If the current root is NULL, the function returns, ending the traversal for that branch.
    - **Traversal**: It recursively calls inorder on the left child (inorder(root->left)), then prints the current node's value (cout << root->val), and finally calls inorder on the right child (inorder(root->right)).
3. **main Function**:
    - An integer array rolls is initialized with 10 unsorted values.
    - A Node pointer root is initialized to NULL to represent an empty tree.
    - A for loop iterates through the rolls array. In each iteration, it calls the insert function to add the current roll number into the BST.
    - After building the BST, it prints "In-order (ascending):" to the console.
    - It then calls the inorder function with the root of the tree. Due to the properties of a BST and the in-order traversal, the values will be printed in **ascending order**.
    - The program concludes by printing a newline character.

Result: The code will build a BST and then print the elements in the following sorted order: 11 12 23 34 45 54 67 78 89 90.

Explanation:

BST is built from roll numbers. In-order prints them in increasing order.

5. **In a university database, student roll numbers are stored using a Binary Search Tree (BST) to allow efficient searching, insertion, and deletion. The roll numbers are: 50, 30, 70, 20, 40, 60, 80. The administrator now wants to delete a student record from the BST. Write a C++ program to delete a node (student roll number) entered by the user.**

Ans:

Theory:

To delete from BST:

1. If node has no child → remove it.
2. If one child → replace with child.
3. If two children → replace with inorder successor, then delete successor.

Code:

```
#include <iostream>

using namespace std;


struct Node {
```

```
    int val;
    Node* left;
    Node* right;
    Node(int v) { val = v; left = right = NULL; }
};

Node* insert(Node* root, int v) {
    if (root == NULL) return new Node(v);
    if (v < root->val) root->left = insert(root->left, v);
    else root->right = insert(root->right, v);
    return root;
}

Node* minValue(Node* node) {
    while (node->left != NULL) node = node->left;
    return node;
}

Node* deleteNode(Node* root, int key) {
    if (root == NULL) return root;
    if (key < root->val) root->left = deleteNode(root->left, key);
    else if (key > root->val) root->right = deleteNode(root->right, key);
    else {
        if (root->left == NULL) {
            Node* temp = root->right;
            delete root;
            return temp;
```

```
        } else if (root->right == NULL) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* succ = minValue(root->right);
        root->val = succ->val;
        root->right = deleteNode(root->right, succ->val);
    }
    return root;
}

void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->val << " ";
    inorder(root->right);
}

int main() {
    int arr[7] = {50,30,70,20,40,60,80};
    Node* root = NULL;
    for (int i = 0; i < 7; i++) root = insert(root, arr[i]);

    cout << "In-order before deletion: ";
    inorder(root); cout << endl;
```

```
    int key;
    cout << "Enter roll number to delete: ";
    cin >> key;

    root = deleteNode(root, key);

    cout << "In-order after deletion: ";
    inorder(root); cout << endl;

    return 0;
}
```

Explanation:
We handle all three delete cases. After deletion, we print BST again using in-order.

This C++ program demonstrates the creation and deletion of a node in a **Binary Search Tree (BST)**. Here's how the code works step-by-step:

1. Building the BST

- A Node struct is defined, representing a node in the tree. Each node stores an integer val and pointers to its left and right children.
- The main function initializes an integer array arr with values to be inserted into the BST.
- A root pointer is initialized to NULL, signifying an empty tree.
- A for loop iterates through the arr array, and the insert function is called for each value.
- The insert function recursively places each new value in the correct position. If the new value is less than the current node's value, it's placed in the left subtree; otherwise, it's placed in the right subtree.

2. Deleting a Node

- After building the tree, the program prompts the user to enter a value (key) to delete.
- The deleteNode function is called, which handles three cases for a node to be deleted:
    - **Case 1: Node has no children (a leaf node)**: The node is simply removed, and its memory is freed.
    - **Case 2: Node has one child**: The node is replaced by its single child. The memory of the deleted node is freed.
    - **Case 3: Node has two children**: The node is replaced by its **in-order successor** (the smallest value in its right subtree). The minValue function finds this successor. The

successor's value is copied to the node being deleted, and then the successor node is removed from its original position.

3. Traversal

- The inorder function is used to print the contents of the tree.
- This function recursively traverses the tree in the order of left subtree, current node, and then right subtree.
- Because of the BST property, this traversal prints the nodes in **ascending order**.
- The main function calls inorder twice: once before deletion to show the original tree and again after deletion to show the modified tree.

6. Design and implement a family tree hierarchy using a Binary Search Tree (BST). The family tree should allow efficient storage, retrieval, and manipulation of information related to individuals and their relationships within the family. Write a C++ program to:
1. Insert family members into the BST (based on their names).
2. Perform in-order, pre-order, and post-order traversals to display the hierarchy.
3. Search for a particular family member by name.

Ans:

Theory:

We can store family members in BST by their names. Traversals give different views, and search finds a member quickly.

Code:

```
#include <iostream>
using namespace std;
```

```
struct Node
   { string
   name;
   Node* left;
   Node* right;
   Node(string s) { name = s; left = right =
NULL; } };

Node* insert(Node* root, string name) {
   if (root == NULL) return new Node(name);
   if (name < root->name) root->left = insert(root->left, name);
   else root->right = insert(root->right, name);
   return
root; }

void inorder(Node* root)
   { if (root == NULL)
   return;
   inorder(root->left);
   cout << root->name << " ";
   inorder(root->right);
}

void preorder(Node* root)
   {  if (root == NULL)
   return; cout << root->name
   << " "; preorder(root->left);
   preorder(root->right);
}

void postorder(Node* root)
   { if (root == NULL) return;
   postorder(root->left);
   postorder(root->right);
   cout << root->name << " ";
}

bool search(Node* root, string key)
   { if (root == NULL) return false;
   if (root->name == key) return true;
   if (key < root->name) return search(root->left, key);
```

```
    else return search(root->right,
key); }

int main() {
    string family[7] =
{"John","Alice","Bob","Mary","David","Zara","Peter"};
    Node* root = NULL;
    for (int i = 0; i < 7; i++) root = insert(root, family[i]);

    cout << "In-order: "; inorder(root); cout << endl;
    cout << "Pre-order: "; preorder(root); cout << endl;
    cout << "Post-order: "; postorder(root); cout << endl;

    string q;
    cout << "Enter name to search: ";
    cin >> q;
    if (search(root, q)) cout << q << " found\n";
    else cout << q << " not found\n";

    return
0; }
```

Explanation:
We store family names in BST. We display tree in 3 traversals and check if a given name is in the tree.