BST Code :

```
#include <bits/stdc++.h>
using namespace std;
struct Node{
    int data;
    struct Node *left,*right;
};
struct Node *root=NULL;
Node* insertBST(Node *node,int val){
    if(node==NULL){
        node=new Node();node->data=val;
        return node;
    }
    if(val<node->data){
        node->left=insertBST(node->left,val);
    }
    else if(val>node->data){
        node->right=insertBST(node->right,val);
    }
    return node;
}
//Deletion by inorder successor
Node *minValueNode(Node *node){
    Node *curr=node;
    while(curr!=NULL&&curr->left!=NULL){
        curr=curr->left;
    }
    return curr;
}
Node *deleteBST(Node *node,int val){
    if(node==NULL){
        return node;
    }
    if(val<node->data){
        node->left=deleteBST(node->left,val);
    }
    else if(val>node->data){
        node->right=deleteBST(node->right,val);
    }
    else{
        //Node found
        if(node->left==NULL){
            Node *temp=node->right;
            free(node);
            return temp;
        }
        else if(node->right==NULL){
            Node *temp=node->left;
            free(node);
            return temp;
        }
        //Node with two children
        Node *temp=minValueNode(node->right);
        node->data=temp->data;
        node->right=deleteBST(node->right,temp->data);
    }
    return node;
}
void inorderTraversal(Node *node){
    if(node==NULL){return;}
    inorderTraversal(node->left);
    cout<<node->data<<" ";
    inorderTraversal(node->right);
}
int main(){
    vector<int>tr={11, 6, 4, 5, 8, 10, 19, 17, 43, 31, 49}; // Preorder
traversal
    for(int i=0;i<(int)tr.size();i++){
        if(i==0){
            root=insertBST(root,tr[i]);
        }
        else{
            insertBST(root,tr[i]);
        }
    }
    inorderTraversal(root);cout<<"\n";
    deleteBST(root,11);
    inorderTraversal(root);cout<<"\n";
}
```

Insertion Sort :

****Insertion sort*** is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- We start with the second element of the array as the first element is assumed to be sorted.
- Compare the second element with the first element if the second element is smaller then swap them.
- Move to the third element, compare it with the first two elements, and put it in its correct position
- Repeat until the entire array is sorted.

Selection Sort :

****Selection Sort**** is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the ****smallest (or largest)**** element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and swap it with the second element.
3. We keep doing this until we get all elements moved to correct position.

Bubble Sort :

****Bubble Sort**** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

- We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
- In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.
- In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

Quick Sort :

****QuickSort**** is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

It works on the principle of ****divide and conquer****, breaking down the problem into smaller sub-problems.

There are mainly three steps in the algorithm:

1. ****Choose a Pivot:**** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. ****Partition the Array:**** Re arrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. ****Recursively Call:**** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
4. ****Base Case:**** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

Code :

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

int partition(vector<int>& arr, int low, int high) {

    // choose the pivot
    int pivot = arr[high];

    // undex of smaller element and indicates
    // the right position of pivot found so far
    int i = low - 1;

    // Traverse arr[low..high] and move all smaller
    // elements on left side. Elements from low to
    // i are smaller after every iteration
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    // move pivot after smaller elements and
    // return its position
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// the QuickSort function implementation
void quickSort(vector<int>& arr, int low, int high) {

    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    vector<int> arr = {10, 7, 8, 9, 1, 5};
    int n = arr.size();
    quickSort(arr, 0, n - 1);

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

Merge Sort :

****Merge sort**** is a popular sorting algorithm known for its efficiency and stability. It follows the Divide and Conquer approach. It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array.
Here's a step-by-step explanation of how merge sort works:

1. ****Divide:**** Divide the list or array recursively into two halves until it can no more be divided.
2. ****Conquer:**** Each subarray is sorted individually using the merge sort algorithm.
3. ****Merge:**** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Code :

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

// Merges two subarrays of arr[].
// First subarray is arr[left..mid]
// Second subarray is arr[mid+1..right]
void merge(vector<int>& arr, int left,
                     int mid, int right){

    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temp vectors
    vector<int> L(n1), R(n2);

    // Copy data to temp vectors 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];

    int i = 0, j = 0;
    int k = left;

    // Merge the temp vectors back
    // into arr[left..right]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[],
    // if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy the remaining elements of R[],
    // if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// begin is for left index and end is right index
// of the sub-array of arr to be sorted
void mergeSort(vector<int>& arr, int left, int right){

    if (left >= right)
        return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}

// Driver code
int main(){

    vector<int> arr = {38, 27, 43, 10};
    int n = arr.size();

    mergeSort(arr, 0, n - 1);
    for (int i = 0; i < arr.size(); i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

Soring algorithm and time complexity table :

Sorting Algorithm	Best Case Time	Average Case Time	Worst Case Time	Space Complexity	Stable?	In-place?
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	✅ Yes	✅ Yes

Sorting Algorithm	Best Case Time	Average Case Time	Worst Case Time	Space Complexity	Stable?	In-place?
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	❌ No	✅ Yes
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	✅ Yes	✅ Yes
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	✅ Yes	❌ No
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$	O(log n) (recursive)	❌ No	✅ Yes
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	❌ No	✅ Yes
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)	✅ Yes	❌ No
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)	✅ Yes	❌ No
Bucket Sort	O(n + k)	O(n + k)	$O(n^2)$	O(n)	✅ Yes	❌ No
Shell Sort	O(n log n)	O(n^(3/2))	$O(n^2)$	O(1)	❌ No	✅ Yes
Tim Sort *(Python's sort)*	O(n)	O(n log n)	O(n log n)	O(n)	✅ Yes	❌ No
Tree Sort	O(n log n)	O(n log n)	$O(n^2)$	O(n)	✅ Yes	❌ No

Tower of Hanoi Problem :

To move n disks from **A → C** using **B**:

1. Move **(n−1)** disks from **A → B** (using **C**).
2. Move **the largest disk** (nth disk) from **A → C**. → requires **1 move**
3. Move the **(n−1)** disks from **B → C** (using **A**).

The problem can be solved by recursion:
• Move top n-1 disks from source to auxiliary rod.
• Move the nth (largest) disk from source to destination rod.
• Move the n-1 disks from auxiliary to destination rod.

Code :

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

// Recursive function to solve Tower of Hanoi
void TowerOfHanoi(int n, char source, char auxiliary, char destination) {
    // Base Case: Only one disk
    if (n == 1) {
        cout << "Move disk 1 from " << source << " to " << destination <<
endl;
        return;
    }

    // Step 1: Move (n-1) disks from source to auxiliary
    TowerOfHanoi(n - 1, source, destination, auxiliary);

    // Step 2: Move the largest disk (nth) from source to destination
    cout << "Move disk " << n << " from " << source << " to " << destination
<< endl;

    // Step 3: Move (n-1) disks from auxiliary to destination
    TowerOfHanoi(n - 1, auxiliary, source, destination);
}

int main() {
    int n;

    cout << "Enter number of disks: ";
    cin >> n;

    cout << "\nSequence of moves:\n";
    TowerOfHanoi(n, 'A', 'B', 'C');

    cout << "\nTotal moves required: " << ( (1 << n) - 1 ) << endl;  // 2^n -
1
    return 0;
}
```