1. Comparison of all Sorting Techniques:

1. **Bubble Sort**:

- Simple and easy to implement.
- Not efficient for large datasets, has a time complexity of O(n^2).
- Works well for small datasets or nearly sorted arrays.

1. **Selection Sor**t:

- Similar simplicity to Bubble Sort.
- Still not efficient for large datasets, with a time complexity of O(n^2).
- Doesn't require as much swapping as Bubble Sort.

1. **Insertion Sort**:

- Efficient for small datasets or nearly sorted arrays.
- Also has a time complexity of O(n^2), but performs better than Bubble and Selection Sort in practice.
- Works well with partially sorted arrays.

1. **Merge Sort**:

- Efficient for large datasets, with a stable time complexity of O(n log n).
- Requires additional memory space for the merging process.
- Well-suited for sorting linked lists.

1. **Quick Sor**t:

- Efficient for large datasets, with an average time complexity of O(n log n).
- However, its worst-case time complexity is O(n^2) (rarely encountered).
- In-place sorting algorithm, meaning it doesn't require additional memory.

1. **Heap Sort**:

- Efficient for large datasets, with a time complexity of O(n log n).
- Requires additional space for the heap data structure.
- In-place sorting algorithm, making it more memory-efficient than Merge Sort.

Each sorting algorithm has its own advantages and disadvantages, and the choice of algorithm depends on factors such as the size and distribution of data, memory constraints, and desired time complexity.

Creation of Stack Data Structure:

Here is the C++ Implementation of the Stack Data Structure with operations such as push, pop, and peek:

#include <iostream>

#define MAX_SIZE 100 // Maximum size of the stack class Stack {

private:

int top; // Index of the top element

int stack[MAX_SIZE];// Array to store elements

public:

```
Stack() { // Constructor to initialize the stack top = -1; // Stack is initially empty

}

bool isEmpty() { // Check if the stack is empty return top == -1;

}

bool isFull() { // Check if the stack is full return top == MAX_SIZE - 1;

}

void push(int value) { // Push an element onto the stack if (isFull()) {

std::cout << "Stack overflow! Cannot push element " << value << std::endl; return;

}

stack[++top] = value;

std::cout << "Pushed " << value << " onto the stack." << std::endl;

}

int pop() { // Pop an element from the stack if (isEmpty()) {

std::cout << "Stack underflow! Cannot pop element." << std::endl; return -1; // Return a default value indicating failure

}

return stack[top--];

}

int peek() { // Get the top element of the stack without removing it if (isEmpty()) {

std::cout << "Stack is empty! No element to peek." << std::endl; return -1; // Return a default value indicating failure

}

return stack[top];

}

void display() { // Display all elements of the stack if (isEmpty()) {

std::cout << "Stack is empty!" << std::endl; return;

}

std::cout << "Elements of the stack: "; for (int i = 0; i <= top; ++i) {

std::cout << stack[i] << " ";

}

std::cout << std::endl;

}

};
```

```
int main() { Stack stack;

stack.push(10); // Push 10 stack.push(20); // Push 20 stack.push(30); // Push 30

stack.display(); // Output: Elements of the stack: 10 20 30

std::cout << "Top element: " << stack.peek() << std::endl; // Output: Top element: 30 std::cout << "Popped element: " << stack.pop() << std::endl; // Output: Popped element: 30 stack.display(); // Output: Elements of the stack: 10 20

return 0;

}
```

Eg. Output:

Pushed 10 onto the stack. Pushed 20 onto the stack. Pushed 30 onto the stack. Elements of the stack: 10 20 30

Top element: 30

Popped element: 30

Elements of the stack: 10 20

Creation of Tree Data Structure:

Here is an Implementation of Binary Tree Data Structure using C++ with operations such as

insertion, searching, deletion, in-order, pre-order, post-order search.

```
#include<iostream> #define SPACE 10

using namespace std; class tree {

public:

int data; tree *left; tree *right;

tree() {

}

data = 0; left = NULL;

right = NULL;

tree(int val) {

data = val; left = NULL; right = NULL;

}

};

class BST { public:

tree *root;

bool isEmpty() {
```