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 Sort**:

- 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 Sort**:

- 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() {
```

```
if (root == NULL)

return true;

else

}

return false;

void insert(tree* n) {

if (isEmpty()) {

root = n; cout << endl;

cout << "VALUE INSERTED AS ROOT NODE!" << endl;

}

else {

tree *temp = root; while (temp != NULL) {

if (n->data == temp->data)

cout << "No Duplicates Allowed!" << endl;

else if (n->data < temp->data && temp->left == NULL) { temp->left = n;

break;

}

else if (n->data < temp->data) temp = temp->left;

else if (n->data > temp->data && temp->right == NULL) { temp->right = n;

break;

}

else

temp = temp->right;

}

}

}

void printPreOrder(tree *r) { if (r == NULL)

return;

else {

}

}
```

```
cout << r->data; cout << " ";

printPreOrder(r->left); printPreOrder(r->right);

void printInOrder(tree *r) { if (r == NULL)

return;

else {

printInOrder(r->left); cout << r->data; cout << " ";

printInOrder(r->right);

}

}

void printPostOrder(tree *r) { if (r == NULL)

return;

else {

}

}

printPostOrder(r->left); printPostOrder(r->right); cout << r->data;

cout << " ";

int height(tree* r){

int lheight,rheight; if(r==NULL)

return -1;

else{

lheight=height(r->left); rheight=height(r->right); if(lheight>rheight)

return (lheight+1);

else

}

}

return (rheight+1);

tree* iterativeSearch(int value) { if (root == NULL)

return root;

else {

tree* temp = root; while (temp != NULL) {

if (temp->data == value)
return temp;

else if (value > temp->data) temp = temp->right;

else

temp = temp->left;

}

}

return NULL;

}

};

int main() {

BST b1;

int option, val, searchVal; do {

tree *n1 = new tree();

cout << "What operation do you want to perform?" << endl; cout << "Select operation no. Enter 0 to exit" << endl;

cout << "1. Insert Node" << endl; cout << "2. Search Node" << endl; cout << "3. Delete Node" << endl;

cout << "4. Print Pre-Order BST values" << endl; cout << "5.Print In-Order BST values" << endl; cout << "6.Print Post-Order values" << endl; cout << "7.Search for a value" << endl; cout<<"8.Print the height of the tree"<<endl; cout << "9. Clear Screen" << endl;

cout << "0. Exit Program" << endl; cin >> option;

switch (option) {

case 0:

case 1:

{

}

break; case 2:

case 3:

case 4:

break;

cout << "Enter the value you want to insert in binary tree" << endl; cin >> val;

n1->data = val; b1.insert(n1); cout << endl;

cout << "Value Inserted!" << endl;

break; break;
```

```
cout << endl; b1.printPreOrder(b1.root); cout << endl;

case 5:

case 6:

break;

cout << endl; b1.printInOrder(b1.root); cout << endl;

break;

cout << endl; b1.printPostOrder(b1.root); cout << endl;

break;

case 7:

cout << "Enter value to search for in the tree" << endl; cin >> searchVal;

cout << endl;

n1 = b1.iterativeSearch(searchVal); if (n1 != NULL)

cout << "Data Element Found in BST!" << endl << endl;

case 8:

else break;

cout << "Data Element not found in BST!" << endl << endl;

cout<<endl;

cout<<"The height of your BST is "<<b1.height(b1.root); cout<<endl;

break; case 9:

system("clear"); break;

default:

cout << "Please Enter Valid Option Number" << endl; break;

}

} while (option != 0);

return 0;

}
```

Eg. Output:

What operation do you want to perform? Select operation no. Enter 0 to exit

1. Insert Node
2. Search Node
3. Delete Node
4. Print Pre-Order BST values 5.Print In-Order BST values 6.Print Post-Order values 7.Search for a value

5. Print the height of the tree
6. Clear Screen
7. Exit Program 1

Enter the value you want to insert in binary tree 12

VALUE INSERTED AS ROOT NODE!

Creation of Graph Data Structure:

```
#include <iostream> #include <vector>

#include <unordered_map> #include <algorithm>

class Graph { private:

std::unordered_map<int, std::vector<int>> adjList;

public:

void insertVertex(int vertex) {

if (adjList.find(vertex) == adjList.end()) { adjList[vertex] = std::vector<int>();

std::cout << "Vertex " << vertex << " inserted successfully." << std::endl;

} else {

std::cout << "Vertex " << vertex << " already exists in the graph." << std::endl;

}

}

void insertEdge(int src, int dest) {

if (adjList.find(src) != adjList.end() && adjList.find(dest) != adjList.end()) { adjList[src].push_back(dest);

adjList[dest].push_back(src);

std::cout << "Edge between " << src << " and " << dest << " inserted successfully." <<

std::endl;

} else {

std::cout << "Vertices " << src << " and/or " << dest << " do not exist in the graph." <<

std::endl;

}

}

void deleteVertex(int vertex) {

if (adjList.find(vertex) != adjList.end()) { adjList.erase(vertex);

for (auto& pair : adjList) {
```

```
auto& neighbors = pair.second;

neighbors.erase(std::remove(neighbors.begin(), neighbors.end(), vertex),

neighbors.end());

}

std::cout << "Vertex " << vertex << " deleted successfully." << std::endl;

} else {

std::cout << "Vertex " << vertex << " does not exist in the graph." << std::endl;

}

}

void deleteEdge(int src, int dest) {

if (adjList.find(src) != adjList.end() && adjList.find(dest) != adjList.end()) { auto& srcNeighbors = adjList[src];

auto& destNeighbors = adjList[dest]; srcNeighbors.erase(std::remove(srcNeighbors.begin(), srcNeighbors.end(), dest),

srcNeighbors.end());

destNeighbors.erase(std::remove(destNeighbors.begin(), destNeighbors.end(), src), destNeighbors.end());

std::cout << "Edge between " << src << " and " << dest << " deleted successfully." <<

std::endl;

} else {

std::cout << "Vertices " << src << " and/or " << dest << " do not exist in the graph." <<

std::endl;

}

}

bool hasVertex(int vertex) {

return adjList.find(vertex) != adjList.end();

}

bool hasEdge(int src, int dest) {

if (adjList.find(src) != adjList.end() && adjList.find(dest) != adjList.end()) { auto& neighbors = adjList[src];

return std::find(neighbors.begin(), neighbors.end(), dest) != neighbors.end();

}

return false;

}

void display() {
```

```
std::cout << "Graph:" << std::endl; for (const auto& pair : adjList) {

std::cout << pair.first << " -> "; for (int neighbor : pair.second) {

std::cout << neighbor << " ";

}

std::cout << std::endl;

}

}

};

int main() { Graph graph;

graph.insertVertex(1); graph.insertVertex(2); graph.insertVertex(3); graph.insertVertex(4);

graph.insertEdge(1, 2);

graph.insertEdge(1, 3);

graph.insertEdge(2, 3);

graph.insertEdge(3, 4); graph.display();

std::cout << "Does vertex 2 exist? " << (graph.hasVertex(2) ? "Yes" : "No") << std::endl; std::cout << "Does edge (1, 4) exist? " << (graph.hasEdge(1, 4) ? "Yes" : "No") << std::endl;

graph.deleteVertex(3); graph.deleteEdge(1, 2);

graph.display(); return 0;

}
```

Eg. Output:

```
Vertex 1 inserted successfully. Vertex 2 inserted successfully. Vertex 3 inserted successfully. Vertex 4 inserted successfully.

Edge between 1 and 2 inserted successfully. Edge between 1 and 3 inserted successfully. Edge between 2 and 3 inserted successfully. Edge between 3 and 4 inserted successfully. Graph:

4 -> 3

3 -> 1 2 4

2 -> 1 3

1 -> 2 3

Does vertex 2 exist? Yes Does edge (1, 4) exist? No Vertex 3 deleted successfully.

Edge between 1 and 2 deleted successfully.
```

Creation of Linked List:

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

```
class node { public:

int data;

node* next = NULL; node(int val) {

data = val; next = NULL;

}

};

void displayList(node* head) { while (head->next != NULL) {

cout << head->data << "->"; head = head->next;

}

cout << head->data;

}

void insertNode(node* head, int newnode, int val) { node* curr = head;

node* fixed = head; node* tmp;

for (int i = 0; i < newnode; i++) { curr = curr->next;

}

tmp = curr->next;

node* n = new node(val); curr->next = n;

n->next = tmp; displayList(fixed);

}

int countFinalNodes(node* head) { int count = 0;

while (head->next != NULL) { count++;

head = head->next;

}

count = count + 1; return count;

}

void searchNode(node* head, int val) { int count = 0;

while (head->data != val && head->next != NULL) { head = head->next;

count++;

}

if (head->data != val && head->next == NULL) count = -1;

if (count >= 0)
```

```
cout << "The element that you are searching for in the linked list is at node/index: " << count << endl;

else

cout << "Sorry, the element that you are searching for does not exist in the linked list " <<

endl;

}

void deleteNode(node* head, int delnode) { node* curr = head;

node* curr2 = head; node* fixed = head;

for (int i = 1; i < delnode - 1; i++) { curr = curr->next;

}

for (int i = 1; i <= delnode; i++) { curr2 = curr2->next;

}

curr->next = curr2; displayList(fixed);

}

int main() {

int num, x, newnode, delnode;

cout << "Enter the length of linked list" << endl; cin >> num;

cout << "Enter elements of the linked list" << endl; cin >> x;

node* n = new node(x);

node* temp = n; node* head = n;

for (int i = 1; i < num; i++) { cin >> x;

node* n = new node(x); temp->next = n;

temp = temp->next;

}

cout << "The elements of your linked list are as follows" << endl; displayList(head);

cout << endl; char ch = 'y'; cout << endl;

while (ch == 'y' && ch != 'n') { cout << endl;

cout << "Choose an operation:" << endl; cout << "1. Insertion" << endl;

cout << "2. Search" << endl; cout << "3. Delete" << endl; cout << "4. Exit" << endl;

int choice;

cin >> choice; switch (choice) {

case 1:
```

```
cout << "Enter the index of node after which you want to insert a new node in the linked list (indexing starts from 0)" << endl;

cin >> newnode;

cout << "Enter the data element of the new node" << endl; cin >> x;

cout << endl;

cout << "The list after you inserted an element in it is as such: " << endl; cout << endl;

insertNode(head, newnode, x); cout << endl;

break; case 2:

cout << "Enter the data element you want to search for " << endl; cin >> x;

cout << endl; searchNode(head, x); cout << endl;

break; case 3:

endl;

}

}

cout << "Enter the position of the node within the list which you want to delete: " <<

cin >> delnode; cout << endl;

cout << "The list after you deleted a node from within it is as such: " << endl; deleteNode(head, delnode);

cout << endl; break;

case 4:

ch = 'n'; break;

default:

cout << "Invalid choice. Please try again." << endl;

return 0;

}
```

Eg. Output:

```
Enter the length of linked list 5

Enter elements of the linked list 10 20 30 40 50

The elements of your linked list are as follows 10->20->30->40->50

Choose an operation:

1. Insertion
2. Search
3. Delete
4. Exit 1
```

```
Enter the index of node after which you want to insert a new node in the linked list (indexing starts from 0)

3

Enter the data element of the new node 32

The list after you inserted an element in it is as such: 10->20->30->40->32->50
```

6. Comparison of all above Data Structures:

Implementation:

- Stack: Can be implemented using arrays or linked lists.
- Binary Search Tree: Implemented using nodes where each node has at most two children.
- Graph: Implemented using adjacency lists, adjacency matrices, or edge lists.
- Linked List: Implemented using nodes where each node contains a data field and a reference (or pointer) to the next node.

Operations:

-
    - Stack: Supports operations like push (insert), pop (remove), top (retrieve top element), and isEmpty (check if the stack is empty).
        - Binary Search Tree: Supports operations like insert, delete, search, traversal (inorder, preorder, postorder), and findMin/findMax.
        - Graph: Operations include adding vertices and edges, removing vertices and edges, checking if vertices or edges exist, and traversing the graph (DFS, BFS).
    - Linked List: Supports operations like insertion, deletion, searching, and traversal (forward and backward).

Memory Management:

-
    - Stack: Memory management is usually straightforward as it uses a fixed-size array or dynamic allocation in a linked list.
- Binary Search Tree: Memory management can be complex due to the dynamic nature of the tree. Balancing operations might be needed to maintain the efficiency of operations.
- Graph: Memory management depends on the representation used (adjacency list, matrix, etc.). Adjacency lists are memory-efficient for sparse graphs, while matrices are more suitable for dense graphs.

- Linked List: Memory management involves dynamic memory allocation for each node.

Memory can be dynamically allocated and deallocated as needed.

Complexity Analysis:

- Stack: Operations like push and pop have a time complexity of O(1).
- Binary Search Tree: Search, insert, and delete operations have an average time complexity of O(log n) for a balanced tree. However, in the worst-case scenario (unbalanced tree), it can degrade to O(n).

- Graph: Time complexity varies based on the operation and the representation used. In general, basic operations like adding or removing vertices and edges are O(1) with adjacency lists, while searching for an edge or vertex can be O(V) or O(E), depending on the representation and the algorithm used.

- Linked List: Insertion and deletion at the beginning or end of the list are O(1). Searching for an element, however, is O(n) as it requires traversing the list.

Use Cases:

-
    - Stack: Useful for implementing algorithms like backtracking, expression evaluation, function call management (call stack), etc.
    - Binary Search Tree: Used in applications requiring fast search, insertion, and deletion, such as database indexing, sorting, and certain searching algorithms.
    - Graph: Suitable for modeling relationships between entities in various domains like social networks, transportation networks, computer networks, etc.
    - Linked List: Commonly used in implementing other data structures like stacks, queues, and hash tables. Also used in scenarios where frequent insertion and deletion operations are required.

7.Creation of Standard Shopping Cart System using Linked List Data Structure implemented in C++:

```
#include <iostream> #include <string> #include <iomanip> #include <limits>

using namespace std;

class CartItem { public:

string name; double price; CartItem* next;

CartItem(string itemName, double itemPrice) { name = itemName;

price = itemPrice; next = nullptr;

}

};

class ShoppingCart { private:

CartItem* head; int itemCount; double totalCost;

public:

ShoppingCart() { head = nullptr; itemCount = 0;

totalCost = 0.0;

}

void addItem(string itemName, double itemPrice) { CartItem* newItem = new CartItem(itemName, itemPrice); if (head == nullptr) {

head = newItem;

} else {

CartItem* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}
```

```
temp->next = newItem;

}

itemCount++; totalCost += itemPrice;

cout << itemName << " added to the cart." << endl;

}

void removeItem(int index) {

if (index <= 0 || index > itemCount) { cout << "Invalid index." << endl; return;

}

CartItem* temp = head; CartItem* prev = nullptr;

for (int i = 1; i < index; i++) { prev = temp;

temp = temp->next;

}

if (prev == nullptr) { head = temp->next;

} else {

prev->next = temp->next;

}

totalCost -= temp->price;

cout << temp->name << " removed from the cart." << endl; delete temp;

itemCount--;

}

void displayCart() {

if (itemCount == 0) {

cout << "The shopping cart is empty." << endl;

} else {

cout << "Shopping Cart:" << endl; CartItem* temp = head;

int index = 1;

while (temp != nullptr) {

cout << setw(5) << index << ". ";

cout << setw(20) << left << temp->name << " $" << fixed << setprecision(2) << temp->price << endl;

temp = temp->next; index++;

}
```

```
cout << "Total Items: " << itemCount << endl;

cout << "Total Cost: $" << fixed << setprecision(2) << totalCost << endl;

}

}

void clearCart() {

while (head != nullptr) { CartItem* temp = head; head = head->next; delete temp;

}

itemCount = 0;

totalCost = 0.0;

cout << "Shopping cart cleared." << endl;

}

~ShoppingCart() { clearCart();

}

};

void displayProductList() {

cout << "Available Products:" << endl;

cout << "1. Laptop $1000.00" << endl; cout << "2. Smartphone $500.00" << endl; cout << "3. Headphones $100.00" << endl;
cout << "4. T-shirt $20.00" << endl;

cout << "5. Jeans $50.00" << endl;

cout << "6. Shoes $80.00" << endl; cout << "7. Watch $200.00" << endl; cout << "8. Sunglasses $30.00" << endl; cout << "9.
Backpack $40.00" << endl; cout << "10. Wallet $15.00" << endl;

}

int main() { ShoppingCart cart; int choice;

string itemName;

double itemPrice;

cout << "Welcome to the Shopping Cart Program!" << endl; do {

cout << "\nMenu:\n";

cout << "1. Add item to cart\n";

cout << "2. Remove item from cart\n"; cout << "3. Display cart\n";

cout << "4. Clear cart\n"; cout << "5. Exit\n";

cout << "Enter your choice: "; cin >> choice;

switch (choice) { case 1:
```

```
displayProductList();
cout << "Enter the number of the product you want to add: "; int productChoice;
cin >> productChoice; switch (productChoice) {
case 1:
itemName = "Laptop"; itemPrice = 1000.00; break;
case 2:
itemName = "Smartphone"; itemPrice = 500.00;
break; case 3:
itemName = "Headphones"; itemPrice = 100.00;
break; case 4:
itemName = "T-shirt"; itemPrice = 20.00; break;
case 5:
itemName = "Jeans"; itemPrice = 50.00; break;
case 6:
itemName = "Shoes"; itemPrice = 80.00;
break; case 7:
itemName = "Watch"; itemPrice = 200.00; break;
case 8:
itemName = "Sunglasses"; itemPrice = 30.00;
break; case 9:
itemName = "Backpack"; itemPrice = 40.00; break;
case 10:
itemName = "Wallet"; itemPrice = 15.00; break;
default:
cout << "Invalid choice. Please try again." << endl; continue;
}
cart.addItem(itemName, itemPrice); break;
case 2:
int index;
cout << "Enter the index of the item you want to remove: "; cin >> index;
cart.removeItem(index); break;
```

```
case 3:

cart.displayCart(); break;

case 4:

cart.clearCart(); break;

case 5:

cout << "Exiting program...\n"; break;

default:

cout << "Invalid choice. Please try again.\n"; cin.clear(); cin.ignore(numeric_limits<streamsize>::max(), '\n');

}

} while (choice != 5);

return 0;

}
```

Eg. Output of Shopping Cart System:

```
Welcome to the Shopping Cart Program! Menu:

  1. Add item to cart
  2. Remove item from cart
  3. Display cart
  4. Clear cart
  5. Exit

Enter your choice: 1 Available Products:

 1. Laptop      $1000.00
 2. Smartphone  $500.00
 3. Headphones  $100.00
 4. T-shirt     $20.00
 5. Jeans       $50.00
 6. Shoes       $80.00
 7. Watch       $200.00
 8. Sunglasses  $30.00
 9. Backpack    $40.00
 10. Wallet     $15.00

Enter the number of the product you want to add: 1 Laptop added to the cart.
```