Prac 1(A)

Aim: Insert an element at a specific position in an array.

Code:

```
#include <iostream>
using namespace std;
int main() {
int arr[100], n, pos, value;
// Input the number of elements
cout << "Enter the number of elements in the array: ";
cin >> n;
// Input the elements
cout << "Enter " << n << " elements:\n";
for(int i = 0; i < n; i++) {
cin >> arr[i];
}
// Input the position and value to insert
cout << "Enter the position to insert (1 to " << n+1 << "): ";
cin >> pos;
cout << "Enter the value to insert: ";
cin >> value;
// Check for valid position
if(pos < 1 || pos > n+1) {
cout << "Invalid position!" << endl;
return 1;
}
// Shift elements to the right to make space
for(int i = n; i >= pos; i--) {
arr[i] = arr[i - 1];
}
// Insert the new element
arr[pos - 1] = value;
n++; // Increase the size of the array
// Output the updated array
cout << "Array after insertion:\n";
for(int i = 0; i < n; i++) {
cout << arr[i] << " ";
}
cout << endl;
return 0;
}
```

Practical 1 (B)

Aim: Delete an element from a specific position in an array.

Code:
```
#include <iostream>
using namespace std;
int main() {
int arr[100], n, pos;
// Input the number of elements
cout << "Enter the number of elements in the array: ";
cin >> n;
// Input the elements
cout << "Enter " << n << " elements:\n";
for(int i = 0; i < n; i++) {
cin >> arr[i];
}
// Input the position to delete
cout << "Enter the position to delete (1 to " << n << "): ";
cin >> pos;
// Check for valid position
if(pos < 1 || pos > n) {
cout << "Invalid position!" << endl;
return 1;
}
// Shift elements to the left to delete the element
for(int i = pos - 1; i < n - 1; i++) {
arr[i] = arr[i + 1];
}
n--; // Decrease the size of the array
// Output the updated array
cout << "Array after deletion:\n";
for(int i = 0; i < n; i++) {
cout << arr[i] << " ";
}
cout << endl;
return 0;
}
```

Practical No: 2 -A

Linked List Manipulation: Write a program to:

Aim: Create a singly linked list

```
#include <iostream>
using namespace std;
struct Node {
int data;
Node* next;
Node(int val) : data(val), next(NULL) {} // This is a constructor, a special function that runs
when you create a new Node object.
};
void insertAtEnd(Node*& head, int val) { Node* newNode = new Node(val);
if (!head) head = newNode;
else {
Node* temp = head
while (temp->next) temp = temp->next;
temp->next = newNode; //temp points to the last node.
}
}
void display(Node* head) {
while (head) {
cout << head->data << " -> "; //Access the data member of the current node and print it
head = head->next;
}
cout << "NULL\n";
}
int main() {
Node* head = NULL;
int n, val;
cout << "Enter number of nodes: ";
cin >> n;
for (int i = 1; i <= n; ++i) {
cout << "Enter value for node " << i << ": ";
cin >> val;
insertAtEnd(head, val); }
cout << "Linked list: ";
display(head);
return 0;
}
```

Practical No 2-B

Aim: Insert a node at the beginning, end, and at a given position in a linked list.

Code:

```
#include <iostream>
using namespace std;
struct Node {
int data;
Node* next;
Node(int val) : data(val), next(NULL) {}
};
void insertAtBeginning(Node*& head, int val) {
Node* newNode = new Node(val);
```

```
newNode->next = head;
head = newNode;
}
void insertAtEnd(Node*& head, int val) {
Node* newNode = new Node(val);
if (!head) head = newNode;
else {
Node* temp = head;
while (temp->next) temp = temp->next;
temp->next = newNode;
}
}
void insertAtPosition(Node*& head, int val, int pos) {
if (pos <= 1) return insertAtBeginning(head, val);
Node* temp = head;
for (int i = 1; i < pos - 1 && temp; i++) temp = temp->next;
if (!temp) return;
Node* newNode = new Node(val);
newNode->next = temp->next;
temp->next = newNode;
}
void display(Node* head) {
while (head) {
cout << head->data << " -> ";
head = head->next;
}
cout << "NULL\n";
}
int main() {
Node* head = NULL;
insertAtEnd(head, 10);
insertAtEnd(head, 20);
insertAtEnd(head, 30);
cout << "Initial list: ";
display(head);
insertAtBeginning(head, 5);
cout << "After inserting 5 at beginning: ";
display(head);
insertAtEnd(head, 40);
cout << "After inserting 40 at end: ";
display(head);
insertAtPosition(head, 15, 3);
cout << "After inserting 15 at position 3: ";
display(head);
```

```
return 0;
}
```

Practical No: 2-C

Aim: Delete a node from a given position in a linked list.

Code:

```
#include <iostream>
using namespace std;
struct Node {
int data;
Node* next;
Node(int val) : data(val), next(NULL) {}
};
void insertAtEnd(Node*& head, int val) {
Node* newNode = new Node(val);
if (!head) head = newNode;
else {
Node* temp = head;
while (temp->next) temp = temp->next;
temp->next = newNode;
}
}
void display(Node* head) {
while (head) {
cout << head->data << " -> ";
head = head->next;
}
cout << "NULL\n";
}
void deleteAtPosition(Node*& head, int pos) {
if (!head || pos <= 0) return;
if (pos == 1) {
Node* del = head;
head = head->next;
delete del;
return;
}
Node* temp = head;
for (int i = 1; i < pos - 1 && temp; ++i)
temp = temp->next;
if (!temp || !temp->next) return;
Node* del = temp->next;
temp->next = del->next;
delete del;
}
int main() {
Node* head = NULL;
insertAtEnd(head, 10);
insertAtEnd(head, 20);
insertAtEnd(head, 30);
insertAtEnd(head, 40);
insertAtEnd(head, 50);
cout << "Initial list: ";
display(head);
deleteAtPosition(head, 3);
cout << "After deleting node at position 3: ";
display(head);
deleteAtPosition(head, 1);
cout << "After deleting head: ";
display(head);
deleteAtPosition(head, 10);
cout << "After trying to delete position 10: ";
display(head);
return 0;
}
```

Practical No 3(A)

Aim: Implement a stack using an array

```
#include <iostream>
using namespace std;
#define MAX 100 // Maximum size of the stack
class Stack {
private:
int arr[MAX]; // Array to store stack elements
int top; // Index of the top element
public:
// Constructor
Stack() {
top = -1; // Stack is initially empty
}
// Push operation
void push(int value) {
if (top >= MAX - 1) {
cout << "Stack Overflow! Cannot push " << value << endl;
return;
}
arr[++top] = value;
```

```
cout << value << " pushed to stack." << endl;
}
// Pop operation
void pop() {
if (top < 0) {
cout << "Stack Underflow! Nothing to pop." << endl;
return;
}
cout << arr[top--] << " popped from stack." << endl;
}
// Peek operation
int peek() {
if (top < 0) {
cout << "Stack is empty!" << endl;
return -1;
}
return arr[top];
}
// Display all elements
void display() {
if (top < 0) {
cout << "Stack is empty!" << endl;
return;
}
cout << "Stack elements (top to bottom): ";
for (int i = top; i >= 0; i--) {
cout << arr[i] << " ";
}
cout << endl;
}
// Check if stack is empty
bool isEmpty() {
return top == -1;
}
// Check if stack is full
bool isFull() {
return top == MAX - 1;
}
};
// Main function to demonstrate stack operations
int main() {
Stack s;
int choice, value;
do {
```

```
cout << "\n--- Stack Menu ---\n";
cout << "1. Push\n2. Pop\n3. Peek\n4. Display\n5. Exit\n”;
cout << "Enter your choice: ";
cin >> choice;
switch (choice) {
case 1:
cout << "Enter value to push: ";
cin >> value;
s.push(value);
break;
case 2:
s.pop();
break;
case 3:
cout << "Top element: " << s.peek() << endl;
break;
case 4:
s.display();
break;
case 5:
cout << "Exiting..." << endl;
break;
default:
cout << "Invalid choice!" << endl;
}
} while (choice != 5);
return 0;
}
```

Practical No 3(B)

Aim: Convert an infix expression to postfix notation using a stack

Code:

```
#include <iostream>
#include <stack>
#include <cctype> // Provides isalnum() to check if a character is alphanumeric
using namespace std;
// Function to return precedence of operators
int precedence(char op) {
if (op == '^')
return 3;
else if (op == '*' || op == '/')
return 2;
else if (op == '+' || op == '-')
```

```
return 1;
else
return -1;
}
// Function to check associativity
bool isRightAssociative(char op) {
return op == '^';
}
// Function to convert infix to postfix
string infixToPostfix(string infix) {
stack<char> st; //stack to store operators and parentheses
string postfix = "";
for (int i = 0; i < infix.length(); i++) {
char ch = infix[i];
if (isalnum(ch)) {
postfix += ch;
}
else if (ch == '(') {
st.push(ch);
}
else if (ch == ')') {
while (!st.empty() && st.top() != '(') {
postfix += st.top();
st.pop();
}
st.pop(); // remove '('
}
else {
while (!st.empty() &&
((precedence(ch) < precedence(st.top())) ||
(precedence(ch) == precedence(st.top()) && ch != '^')) &&
st.top() != '(') {
postfix += st.top(); //Take the top operator from the stack, add it to postfix
st.pop(); // after adding remove from the Stack
}
st.push(ch);
}
}
while (!st.empty()) {
postfix += st.top();
st.pop();
}
return postfix;
}
```

```
// Main function
int main() {
string infix;
cout << "Enter infix expression: ";
cin >> infix;
string postfix = infixToPostfix(infix);
cout << "Postfix expression: " << postfix << endl;
return 0;
}
```

Practical No 4 (A)

Aim: Implement a queue using an array.

Code:

```
#include <iostream>
using namespace std;
#define SIZE 100
class Queue {
private:
int arr[SIZE];
int front, rear;
public:
Queue() {
front = -1;
rear = -1;
}
// Add element to the queue
void enqueue(int value) {
if (rear == SIZE - 1) {
cout << "Queue Overflow (Full)" << endl;
return;
}
if (front == -1) front = 0; //If it’s the first element, set front to 0.
rear++;
arr[rear] = value; //Adds element to queue
cout << value << " enqueued to queue." << endl;
}
// Remove element from the queue
void dequeue() {
if (front == -1 || front > rear) {
cout << "Queue Underflow (Empty)" << endl;
return;
}
cout << arr[front] << " dequeued from queue." << endl;
Front++; //Move front forward by 1.
```

```
}
// Display front element
void peek() {
if (front == -1 || front > rear) {
cout << "Queue is empty." << endl;
return;
}
cout << "Front element: " << arr[front] << endl;
}
// Display all elements
void display() {
if (front == -1 || front > rear) {
cout << "Queue is empty." << endl;
return;
}
cout << "Queue elements: ";
for (int i = front; i <= rear; i++) {
// loops from front to rear and prints each value in the queue.
cout << arr[i] << " ";
}
cout << endl;
}
};
int main() {
Queue q;
int choice, value;
do {
cout << "\nQueue Menu:\n";
cout << "1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\n";
cout << "Enter your choice: ";
cin >> choice;
switch (choice) {
case 1:
cout << "Enter value to enqueue: ";
cin >> value;
q.enqueue(value);
break;
case 2:
q.dequeue();
break;
case 3:
q.peek();
break;
case 4:
```

```
q.display();
break;
case 5:
cout << "Exiting program." << endl;
break;
default:
cout << "Invalid choice. Try again!" << endl;
}
} while (choice != 5);
return 0;
}
```

Practical No. 4(B)

Aim: Simulate a simple queuing system (e.g., customer service queue).

```
#include <iostream>
#include <string>
using namespace std;
#define SIZE 100
class CustomerQueue {
private:
string queue[SIZE];
int front, rear;
public:
CustomerQueue() {
front = -1;
rear = -1;
}
// Add a new customer to the queue
void arrive(string name) {
if (rear == SIZE - 1) {
cout << "Queue is full! Cannot add more customers.\n";
return;
}
if (front == -1) front = 0;
rear++;
queue[rear] = name;
cout << "Customer '" << name << "' arrived.\n";
}
// Serve the next customer
void serve() {
if (front == -1 || front > rear) {
cout << "Queue is empty! No customers to serve.\n";
return;
```

```
}
cout << "Customer '" << queue[front] << "' is being served.\n";
front++;
}
// Show current customers in the queue
void showQueue() {
if (front == -1 || front > rear) {
cout << "Queue is empty.\n";
return;
}
cout << "Customers in queue:\n";
for (int i = front; i <= rear; i++) {
cout << (i - front + 1) << ". " << queue[i] << endl;
}
}
};
int main() {
CustomerQueue q;
int choice;
string name;
do {
cout << "\n--- Customer Service Queue ---\n";
cout << "1. New Customer Arrives\n";
cout << "2. Serve Next Customer\n";
cout << "3. Show Queue\n";
cout << "4. Exit\n";
cout << "Enter your choice: ";
cin >> choice;
switch (choice) {
case 1:
cout << "Enter customer name: ";
cin >> name;
q.arrive(name);
break;
case 2:
q.serve();
break;
case 3:
q.showQueue();
break;
case 4:
cout << "Exiting system.\n";
break;
default:
```

```
cout << "Invalid choice. Try again.\n";
}
} while (choice != 4);
return 0;
}
```

Practical No.5 (A,B)

Aim: a) Create a binary search tree.
b) Insert nodes into a binary search tree.

Aim: Insert nodes into a binary search tree.

```
#include <iostream>
using namespace std;
// Define the structure for a tree node
struct Node {
int data;
Node* left;
Node* right;
// Constructor to create a new node
Node(int value) {
data = value;
left = right = NULL;
}
};
// Function to insert a node into the BST
Node* insert(Node* root, int value) {
if (root == NULL) {
return new Node(value); // Create a new node if tree is empty
}
if (value < root->data) {
root->left = insert(root->left, value); // Insert in left subtree
} else if (value > root->data) {
root->right = insert(root->right, value); // Insert in right subtree
}
// If value == root->data, we ignore it (no duplicates in BST)
return root;
}
// Function to print the tree in Inorder (Left -> Root -> Right)
void inorder(Node* root) {
if (root != NULL) {
inorder(root->left);
cout << root->data << " ";
inorder(root->right);
}
}
int main() {
Node* root = NULL; // Start with an empty tree
int n, value;
cout << "Enter number of nodes to insert: ";
cin >> n;
cout << "Enter " << n << " node values:\n";
for (int i = 0; i < n; i++) {
cin >> value;
root = insert(root, value);
}
cout << "Inorder traversal of BST: ";
inorder(root);
cout << endl;
return 0;
}
```

Practical no 5-C

Aim: Search for a node in a binary search tree.

```
#include <iostream>
using namespace std;
struct Node {
int data;
Node* left;
Node* right;
//constructor that sets the node’s value and initializes both child pointers to nullptr.
Node(int value) {
data = value;
left = right = NULL;
}
};
// Function to insert a new node into the BST
Node* insert(Node* root, int value) {
if (root == NULL)
return new Node(value);
if (value < root->data)
root->left = insert(root->left, value);
else if (value > root->data)
root->right = insert(root->right, value);
return root;
}
// Function to search a node in BST
bool search(Node* root, int key) {
if (root == NULL)
return false;
if (root->data == key)
return true;
//Search left BST if the key is smaller otherwise search right BST if it’s larger.
else if (key < root->data)
return search(root->left, key);
else
return search(root->right, key);
}
// Inorder traversal to display the tree in ascending
void inorder(Node* root) {
if (root != NULL) {
inorder(root->left);
cout << root->data << " ";
inorder(root->right);
}
}
// Main function
int main() {
Node* root = NULL;
int n, value, key;
cout << "Enter number of nodes to insert: ";
cin >> n;
cout << "Enter " << n << " values:\n";
for (int i = 0; i < n; i++) {
cin >> value;
root = insert(root, value);
}
cout << "Inorder traversal of BST: ";
inorder(root);
cout << endl;
cout << "Enter value to search: ";
cin >> key;
if (search(root, key))
cout << key << " is found in the BST." << endl;
else
cout << key << " is NOT found in the BST." << endl;
return 0;
}
```

Practical No:6

Tree Traversal: Write a program to:

a. Implement pre-order,

b. in-order,

c. Post-order traversal of a binary tree.

```
#include <iostream>
using namespace std;
// Node structure of BST
struct Node {
 int data;
 Node* left;
 Node* right;
};
// Function to create a new node
Node* createNode(int value) {
 Node* newNode = new Node(); //dynamically allocates memory for the node.
 newNode->data = value;
 newNode->left = NULL;
 newNode->right = NULL;
 return newNode;
}
// Pre-order Traversal (Root ? Left ? Right)
void preorder(Node* root) {
 if (root != NULL) {
 cout << root->data << " "; // Visit root
 preorder(root->left); // Visit left subtree
 preorder(root->right); // Visit right subtree
 }
}
// In-order Traversal (Left ? Root ? Right)
void inorder(Node* root) {
 if (root != NULL) {
 inorder(root->left); // Visit left subtree
 cout << root->data << " "; // Visit root
 inorder(root->right); // Visit right subtree
 }
}
// Post-order Traversal (Left ? Right ? Root)
void postorder(Node* root) {
 if (root != NULL) {
 postorder(root->left); // Visit left subtree
 postorder(root->right); // Visit right subtree
 cout << root->data << " "; // Visit root
 }
}
int main() {
 // Manually creating a binary tree
 /*
 1
 / \
 2 3
 / \ \
 4 5 6
 */
 Node* root = createNode(1);
 root->left = createNode(2);
 root->right = createNode(3);
 root->left->left = createNode(4);
 root->left->right = createNode(5);
 root->right->right = createNode(6);
 // Display traversals
 cout << "Pre-order Traversal: ";
 preorder(root);
 cout << endl;
 cout << "In-order Traversal: ";
 inorder(root);
 cout << endl;
 cout << "Post-order Traversal: ";
 postorder(root);
 cout << endl;
 return 0;
}
```

Practical No.7

Hash Table: Write a program to:

a. Implement a hash table with separate chaining for collision handling.

b. Store and retrieve data from the hash table.

```
#include <iostream>
#include <list>
using namespace std;
class HashTable {
int size; // Number of buckets
list<int>* table; // Array of linked lists
public:
 // Constructor uns when you create the hash table
HashTable(int s) {
size = s; //saves how many buckets you want.
table = new list<int>[size]; // Create array of lists
}
// Hash function
int hashFunction(int key) {
 return key % size; //The hash function maps any integer key to a bucket index.
}
// Insert key
void insert(int key) {
int index = hashFunction(key); //decide which bucket the key belongs to.
table[index].push_back(key); //appends the key to the end of that list
}
// Search key (Retrieve) and return bucket index if found
int search(int key) {
int index = hashFunction(key);
//This loop goes through each element in the bucket at that index.
for (list<int>::iterator it = table[index].begin(); it != table[index].end(); ++it) {
if (*it == key) {
return index; // return bucket index
}
}
return -1; // key not found
}
// Display hash table
void display() {
for (int i = 0; i < size; i++) {
cout << i << " --> ";
//loop goes through every element stored in one bucket of the hash table.
for (list<int>::iterator it = table[i].begin(); it != table[i].end(); ++it) {
cout << *it << " ";
}
cout << endl;
}
}
};
int main() {
int size, n, key;
cout << "Enter size of hash table: ";
cin >> size;
HashTable ht(size);
cout << "Enter number of keys to insert: ";
cin >> n;
cout << "Enter " << n << " keys:\n";
for (int i = 0; i < n; i++) {
cin >> key;
ht.insert(key); // Store key
}
cout << "\nHash Table:\n";
ht.display();
```

```
// Search for a key
cout << "\nEnter key to search: ";
cin >> key;
int bucketIndex = ht.search(key);
if (bucketIndex != -1) {
cout << key << " is found in bucket index " << bucketIndex << ".\n";
} else {
cout << key << " is NOT found in the hash table.\n";
}
system("pause");
return 0;
}
```

Practical No.8a

Sorting Algorithms: Write programs to implement and compare the following sorting algorithms:

a. Bubble sort

Code:

```
#include <iostream>
using namespace std;
int main() {
int n;
cout << "Enter number of elements: ";
cin >> n;
int arr[50]; // Maximum 50 elements
cout << "Enter elements:\n";
for(int i = 0; i < n; i++)
cin >> arr[i];
// Bubble Sort
for(int i = 0; i < n-1; i++) { //number of passes.
for(int j = 0; j < n-i-1; j++) { //compare adjacent pairs arr[j] and arr[j+1].
if(arr[j] > arr[j+1]) { // Swap if in wrong order
int temp = arr[j];
arr[j] = arr[j+1];
arr[j+1] = temp;
}
}
}
// Display sorted array
cout << "Sorted array: ";
for(int i = 0; i < n; i++)
cout << arr[i] << " ";
cout << endl;
return 0;
}
```

Practical No.8b

Sorting Algorithms: Write programs to implement and compare the following sorting algorithms:
a
b. Insertion sort
Code:

```
#include <iostream>
using namespace std;
int main() {
int n;
cout << "Enter number of elements: ";
cin >> n;
int arr[50]; // maximum 50 elements
cout << "Enter elements:\n";
for(int i = 0; i < n; i++)
cin >> arr[i];
// Insertion Sort
for(int i = 1; i < n; i++) {
int key = arr[i]; // element to insert
int j = i - 1; //Start comparing with the last element of the already sorted part.
// loop shifts elements to the right if they are greater than the key
while(j >= 0 && arr[j] > key) {
arr[j + 1] = arr[j];
J--; //Move one step left and continue checking.
}
arr[j + 1] = key; // insert key at correct position
}
// Display sorted array
cout << "Sorted array: ";
for(int i = 0; i < n; i++)
cout << arr[i] << " ";
cout << endl;
system("pause");
return 0;
}
```

Practical No.8c

Sorting Algorithms: Write programs to implement and compare the following sorting algorithms:
c. Selection sort
Code:

```
#include <iostream>
using namespace std;
int main() {
```

```
int n;
cout << "Enter number of elements: ";
cin >> n;
int arr[50]; // maximum 50 elements
cout << "Enter elements:\n";
for(int i = 0; i < n; i++)
cin >> arr[i];
// Selection Sort
for(int i = 0; i < n-1; i++) { //Pick a position
int minIndex = i; // assume current element is minimum
for(int j = i+1; j < n; j++) { //Find the smallest element in the rest of the array.
if(arr[j] < arr[minIndex])
minIndex = j; // update index of minimum
}
// Swap minimum element with first element of unsorted part
int temp = arr[i];
arr[i] = arr[minIndex];
arr[minIndex] = temp;
}
// Display sorted array
cout << "Sorted array: ";
for(int i = 0; i < n; i++)
cout << arr[i] << " ";
cout << endl;
system("pause");
return 0;
}
```