Program 1:

Write a C++ program that uses functions to perform the following:

- **Create a singly linked list of integers.**
- **Delete a given integer from the above linked list.**
- **Display the contents of the above list after deletion.**

Source Code:

```
#include<iostream>
using namespace std;
class Node
{
public:
   int data;
   Node* next;
   List() // default constructor
   {
      data=NULL;
      next=NULL;
   }
};
class singly
{
public:
   void create(Node** head_ref, int new_data);
   void delet(Node **head_ref, int position);
   void display(Node *node);
};
void singly:: create(Node** head_ref, int new_data)
{
      Node* new_node = new Node();
   new_node->data = new_data;
   new_node->next = (*head_ref);
```

```
    (*head_ref) = new_node;
}
 void singly:: delet(Node **head_ref, int position)
 {
     if (*head_ref == NULL)
        return;
    Node* temp = *head_ref;
    if (position == 0)
    {
        *head_ref = temp->next;
        free(temp);
        return;
    }
    for(int i = 0; temp != NULL && i < position - 1; i++)
        temp = temp->next;
    if (temp == NULL || temp->next == NULL)
        return;
     Node *next1 = temp->next->next;
    free(temp->next);
    temp->next = next1;
 }
 void singly:: display(Node *node)
 {
     while (node != NULL)
    {
        cout << node->data << " ";
        node = node->next;
    }
 }
int main()
{
```

```
    singly s ;
     Node* head = NULL;
    s.create(&head,1);
    s.create(&head,2);
    s.create(&head,3);
    s.create(&head,4);
  cout << "Created Linked List: ";
  s.display(head);
      s.delet(&head, 1);
 cout << "\nLinked List after Deletion at position 1: ";
    s.display(head);
    return 0;
}
```

Output:

Created Linked List: 4 3 2 1

Linked List after Deletion at position 1: 4 2 1

Program 3:

Write a C++ program that uses stack operations to convert a given infix expression into its postfix equivalent, Implement the stack using an array.

Source Code:

```
#include <iostream>
#include <stack>
using namespace std;
class pre
{
public:int priority (char alpha);
string convert(string infix);
};
int pre:: priority (char alpha){
if(alpha == '+' || alpha =='-')
return 1;
```

```
if(alpha == '*' || alpha =='/')
return 2;
if(alpha == '^')
return 3;
return 0;
}
string pre::convert(string infix)
{
int i = 0;
string postfix = "";
stack <int>s; // using inbuilt stack< > from C++ stack library
while(infix[i]!='\0')
{
if(infix[i]>='a' && infix[i]<='z'|| infix[i]>='A'&& infix[i]<='Z')
{
postfix += infix[i];
i++;
}
else if(infix[i]=='(') // if opening bracket then push the stack
{
s.push(infix[i]);
i++;
}

else if(infix[i]==')') // if closing bracket encounted then keep popping from stack until
// closing a pair opening bracket is not encountered
{
while(s.top()!='('){
postfix += s.top();
s.pop();
}
s.pop();
i++;
}
else
{
while (!s.empty() && priority(infix[i]) <= priority(s.top())){
postfix += s.top();
s.pop();
}
```

```
s.push(infix[i]);
i++;
}
}
while(!s.empty()){
postfix += s.top();
s.pop();
}
cout << "Postfix is : " << postfix; //it will print postfix conversion
return postfix;
}
int main()
{
pre p;
string infix = "((a+b)*c)";
string postfix;
postfix = p.convert(infix);
return 0;

}
```

Output:

Postfix is : ab+c*

Program 4:

Write a C++ program to implement a double ended queue ADT using an array,

Source Code:

```
#include<iostream>
using namespace std;
#define SIZE 5
class dequeue
{
   int a[10],front,rear;
   public:
      dequeue();
      void add_at_beg(int);
      void add_at_end(int);
```

```
        void delete_fr_front();
        void delete_fr_rear();
        void display();
};
dequeue::dequeue()
{
   front=-1;
   rear=-1;
}
void dequeue::add_at_end(int item)
{
   if(rear>=SIZE-1)
   {
      cout<<"\n insertion is not posibble,overflow!!!!";
   }
   else
   {
      if(front==-1)
      {
         front++;
         rear++;
      }
      else
      {
         rear=rear+1;
      }
   a[rear]=item;
   cout<<"\nInserted item is"<<a[rear];
   }
}
void dequeue::add_at_beg(int item)
{

      if(front==-1)
      {
         front=0;
         a[++rear]=item;
         cout<<"\n inserted element is"<<item;
      }
      else if(front!=0)
```

```
        {
           a[--front]=item;
           cout<<"\n inserted element is"<<item;
        }
        else
        {
             cout<<"\n insertion is not posibble,overflow!!!";
        }
}
void dequeue::display()
{
   if(front==-1)
   {
      cout<<"Dequeue is empty";
   }
   else
   {
      for(int i=front;i<=rear;i++)
      {
         cout<<a[i]<<" ";
      }
   }
}
void dequeue::delete_fr_front()
{
        if(front==-1)
        {
           cout<<"deletion is not possible::dequeue is empty";
           return;
        }
        else
        {
           cout<<"the deleted element is"<<a[front];
           if(front==rear)
           {
              front=rear=-1;
              return;
           }
           else
              front=front+1;
```

```
        }
}
void dequeue::delete_fr_rear()
{
        if(front==-1)
        {
           cout<<"deletion is not possible::dequeue is empty";
           return;
        }
        else
        {
           cout<<"the deleted element is"<<a[rear];
           if(front==rear)
           {
              front=rear=-1;
           }
           else
              rear=rear-1;
        }
}
int main()
{
   int c,item;
   dequeue d1;
   do
   {
      cout<<"\n\n***DEQUEUE OPERATION***\n";
      cout<<"\n 1_insert at beginning";
      cout<<"\n 2_insert at end";
      cout<<"\n 3_display";
      cout<<"\n 4_deletion from front";
      cout<<"\n 5_deletion from rear";
      cout<<"\n 6_exit";
      cout<<"\n enter your choice";
      cin>>c;
      switch(c)
      {
         case 1:cout<<"enter the element to be inserted";
                cin>>item;
                d1.add_at_beg(item);
```

```
                break;
        case 2:cout<<"enter the element to be inserted";
                cin>>item;
                d1.add_at_end(item);
                break;
        case 3:d1.display();
                break;
        case 4:d1.delete_fr_front();
                break;
        case 5:d1.delete_fr_rear();
               break;
        case 6:exit(1);
                break;
        csdefault:cout<<"invalid choice";
              break;
      }
   }
   while(c!=7);
}
```

Output:

```
***DEQUEUE OPERATION***

 1_insert at beginning
 2_insert at end
 3_display
 4_deletion from front
 5_deletion from rear
 6_exit
 enter your choice1
enter the element to be inserted20

 inserted element is20

***DEQUEUE OPERATION***

 1_insert at beginning
 2_insert at end
 3_display
 4_deletion from front
 5_deletion from rear
 6_exit
 enter your choice2
```

enter the element to be inserted50

Inserted item is50

DEQUEUE OPERATION

1_insert at beginning
2_insert at end
3_display
4_deletion from front
5_deletion from rear
6_exit
enter your choice2
enter the element to be inserted99

Inserted item is99

DEQUEUE OPERATION

1_insert at beginning
2_insert at end
3_display
4_deletion from front
5_deletion from rear
6_exit
enter your choice3
20 50 99

DEQUEUE OPERATION

1_insert at beginning
2_insert at end
3_display
4_deletion from front
5_deletion from rear
6_exit
enter your choice4
the deleted element is20

DEQUEUE OPERATION

1_insert at beginning
2_insert at end
3_display
4_deletion from front
5_deletion from rear

6_exit
enter your choice20

DEQUEUE OPERATION

1_insert at beginning
2_insert at end
3_display
4_deletion from front
5_deletion from rear
6_exit
enter your choice3
50 99

DEQUEUE OPERATION

1_insert at beginning
2_insert at end
3_display
4_deletion from front
5_deletion from rear
6_exit
enter your choice5
the deleted element is99

DEQUEUE OPERATION

1_insert at beginning
2_insert at end
3_display
4_deletion from front
5_deletion from rear
6_exit
enter your choice99

DEQUEUE OPERATION

1_insert at beginning
2_insert at end
3_display
4_deletion from front
5_deletion from rear
6_exit
enter your choice3

Program 6:

Write a C++ program that implements Insertion sort algorithm to arrange a list of integers in ascending order.

Source Code:

```
#include<iostream>
using namespace std;
class insertion
{
public: void insertionSort(int arr[], int n);//parameterized constructor
        void printArray(int arr[], int n);
};
void insertion::insertionSort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}
void insertion:: printArray(int arr[], int n)
{
   int i;
   for (i = 0; i < n; i++)
      cout << arr[i] << " ";
   cout << endl;
```

```
}
int main()
{
    insertion numbers;
    int arr[] = { 12, 11, 13, 5, 6 };
    int N = sizeof(arr) / sizeof(arr[0]);
cout << "Array before Sorting: ";
   numbers.printArray(arr, N);

cout << "Array after Sorting: ";
    numbers.insertionSort(arr, N);
    numbers.printArray(arr, N);


    return 0;
}
```

Output:

Array before Sorting: 12 11 13 5 6

Array after Sorting: 5 6 11 12 13

Program 7:

Write a template-based C++ program that implements selection sort algorithm to arrange a list of elements in descending order.

Source Code:

```
//Selection sort with template List
#include <iostream>
#include <vector>
using namespace std;

template <typename T>
class List
{
```

```
private:
    std::vector<T> data;
public:
    // Constructor
    List(const std::vector<T>& input) : data(input) {} // Constructor


    void selectionSort()     // Member function for selection sort
     {
        int n = data.size();

        for (int i = 0; i < n - 1; ++i) // Find the minimum element in the unsorted part of the array
            {
            int max = i;
            for (int j = i + 1; j < n; ++j) {
                if (data[max] < data[j]) {
                    max = j;
                }
            }

            std::swap(data[i], data[max]); // Swap the found minimum element with the first element
        }
    }

    void display() const      // Member function to display the elements of the list
     {
        for (T value : data) {
            std::cout << value << " ";
        }
        std::cout << std::endl;
    }
};

int main() {

    std::vector<int> numbers = {64, 25, 12, 22, 11};
    std::vector<string> names = {"kholi", "rohith", "dhoni", "sachin", "ganguly"};


    List<int> numbersList(numbers);
    List<string> namesList(names);

    std::cout << "Original list: "<<endl;
    numbersList.display();
    namesList.display();
```

```
    numbersList.selectionSort();
    namesList.selectionSort();

    std::cout << "Sorted list: "<<endl;
    numbersList.display();
    namesList.display();
    return 0;
}
```

Output:

Original list:

64 25 12 22 11

kholi rohith dhoni sachin ganguly

Sorted list:

64 25 22 12 11

sachin rohith kholi ganguly dhoni

Program 8:

Write a template-based C++ program that implements Quick sort algorithm to arrange a list of elements in ascending order.

Source Code:

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

template <typename T>
int partition(std::vector<T>& arr, int low, int high) {
    T pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            std::swap(arr[i], arr[j]);
```

```
        }
    }
    std::swap(arr[i + 1], arr[high]);
    return i + 1;
}

template <typename T>
void quickSort(std::vector<T>& arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

template <typename T>
void quickSort(std::vector<T>& arr) {
    int n = arr.size();
    quickSort(arr, 0, n - 1);
}
int main() {
    std::vector<int> arr = {12, 7, 11, 13, 5, 6};
    std::cout << "Original array: ";
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    quickSort(arr);

    std::cout << "Sorted array: ";
```

```
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output:

Original array: 12 7 11 13 5 6

Sorted array: 5 6 7 11 12 13

Program 9:

Write a C++ program that implements Heap sort algorithm for sorting a list of integers in ascending order.

Source Code:

```
#include <iostream>
using namespace std;
class heap{
public:
    void heaplargest(int arr[], int n, int i);//parameterized constructor
    void heapSort(int arr[], int n);//parameterized constructor
    void printArray(int arr[], int n);//parameterized constructor
};
void  heap::heaplargest(int arr[], int n, int i) {
  // Find largest among root, left child and right child
  int largest = i;
  int left = 2 * i + 1;
  int right = 2 * i + 2;
```

```
  if (left < n && arr[left] > arr[largest])
    largest = left;

  if (right < n && arr[right] > arr[largest])
    largest = right;
  // Swap and continue heaplarge if root is not largest
  if (largest != i) {
    swap(arr[i], arr[largest]);
    heaplargest(arr, n, largest);
  }
}
// main function to do heap sort
void heap::heapSort(int arr[], int n) {
  //  max heap
  for (int i = n / 2 - 1; i >= 0; i--)
   heaplargest(arr, n, i);
  // Heap sort
  for (int i = n - 1; i >= 0; i--) {
    swap(arr[0], arr[i]);

    // Heapify root element to get highest element at root again
    heaplargest(arr, i, 0);
  }
}
// Print an array
void heap:: printArray(int arr[], int n) {
  for (int i = 0; i < n; ++i)
    cout << arr[i] << " ";
  cout << "\n";
}
// Driver code
```

```
int main() {
    heap s;
  int arr[] = {1, 12, 9, 5, 6, 10};
  int n = sizeof(arr) / sizeof(arr[0]);
  s.heapSort(arr, n);

  cout << "Sorted array is \n";
  s.printArray(arr, n);
 }
```

Output:

Sorted array is

1 5 6 9 10 12

Program 10:

Write a C++ program that implements Radix sort algorithm for sorting a list of integers in ascending order.

Source Code:

```
#include<iostream>
using namespace std;
class radix
{
  public:
  int getMax(int array[], int n);
  void countSort(int array[], int size, int place);
  void radixsort(int array[], int size);
  void display(int array[], int size);
};
int radix::getMax(int array[], int n)//Function to get the largest element from an array
{
 int max = array[0];
```

```
  for (int i = 1; i < n; i++) if (array[i] > max)
      max = array[i];
  return max;
}
void radix::countSort(int array[], int size, int place)
{
  const int max = 10;
  int output[size];
  int count[max];

  for (int i = 0; i < max; ++i)
    count[i] = 0;

  for (int i = 0; i < size; i++)  //Calculate count of elements
    count[(array[i] / place) % 10]++;

  for (int i = 1; i < max; i++) //Calculating cumulative count

    count[i] += count[i - 1];
  for (int i = size - 1; i >= 0; i--)//Placing the elements in sorted order
  {
    output[count[(array[i] / place) % 10] - 1] = array[i];
    count[(array[i] / place) % 10]--;
  }

  for (int i = 0; i < size; i++)
    array[i] = output[i];
}

void radix::radixsort(int array[], int size)//Main function to implement radix sort
{
```

```
  int max = getMax(array, size); //Getting maximum element

  //Applying counting sort to sort elements based on place value.
  for (int place = 1; max / place > 0; place *= 10)
    countSort(array, size, place);
}
void radix::display(int array[], int size)//Printing an array
{
  int i;
  for (i = 0; i < size; i++)
    cout << array[i] << "\t";
  cout << endl;
}
int main()
{
    radix r;
  int array[] = {170, 45, 75, 90, 802, 24, 2, 66};
  int n = sizeof(array) / sizeof(array[0]);
  cout<<"Before sorting: \n";
  r.display(array, n);
  r.radixsort(array, n);
  cout<<"After sorting :\n";
  r.display(array, n);
}
```

Output:

```
Before sorting:
170     45      75      90      802     24      2       66
After sorting :
2       24      45      66      75      90      170     802
```

Program 11:

Write a C++ program that uses functions to perform the following:

- **Create a binary search tree of integers.**
- **Traverse the above Binary search tree non recursively in order.**

Source Code:

```
#include <bits/stdc++.h>
using namespace std;
// A binary tree Node has data, pointer to left child
// and a pointer to right child
struct Node
{
   int data;
   struct Node* left;
   struct Node* right;
   Node(int data)
   {
      this->data = data;
      left = right = NULL;
   }
};
class binary
{
public:
   void inOrder(struct Node* root);
};
void binary::inOrder(struct Node* root)
{
   stack<Node*> s;
   Node* curr = root;

   while (curr != NULL || s.empty() == false) {
```

```
        // Reach the left most Node of the
        // curr Node
        while (curr != NULL) {
            s.push(curr);
            curr = curr->left;
        }
        curr = s.top();
        s.pop();
        cout << curr->data << " ";
        curr = curr->right;
    }
}

int main()
{
 binary b;
    struct Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);

    b.inOrder(root);
    return 0;
}
```

Output:

4 2 5 1 3

Program 12:

Write a C++ program that uses functions to perform the following:

- Create a binary search tree of integers.

- Search for an integer key in the above binary search tree non recursively.
- Search for an integer key in the above binary search tree recursively.

1) Search for an integer key in the above binary search tree non recursively.

Source Code:

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

class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

class BST {
private:
    TreeNode* root;

public:
    BST() : root(nullptr) {}

    // Function to insert data into the BST
    void insert(int value) {
        TreeNode* newNode = new TreeNode(value);
        if (root == nullptr) {
            root = newNode;
```

```
        return;
    }

    TreeNode* current = root;
    while (true) {
        if (value <= current->data) {
            if (current->left == nullptr) {
                current->left = newNode;
                return;
            }
            current = current->left;
        } else {
            if (current->right == nullptr) {
                current->right = newNode;
                return;
            }
            current = current->right;
        }
    }
}

// Function to display in-order traversal non-recursively
void displayInOrder() {
    if (root == nullptr) {
        cout << "Tree is empty" << endl;
        return;
    }

    stack<TreeNode*> stk;
    TreeNode* current = root;
```

```
    cout << "In-order traversal: ";
    while (current != nullptr || !stk.empty()) {
        while (current != nullptr) {
            stk.push(current);
            current = current->left;
        }

        current = stk.top();
        stk.pop();
        cout << current->data << " ";

        current = current->right;
    }
    cout << endl;
}

// Function to search for a key non-recursively
bool search(int key) {
    TreeNode* current = root;
    while (current != nullptr) {
        if (key == current->data) {
            return true; // Key found
        } else if (key < current->data) {
            current = current->left;
        } else {
            current = current->right;
        }
    }
    return false; // Key not found
```

```
    }
};

int main() {
    BST bst;

    // Insert some data
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);

    // Display in-order traversal
    bst.displayInOrder();

    // Search for a key
    int key = 40;
    if (bst.search(key)) {
        cout << key << " found in the tree." << endl;
    } else {
        cout << key << " not found in the tree." << endl;
    }

    return 0;
}
```

Output:

In-order traversal: 20 30 40 50 60 70 80

40 found in the tree.

2) Search for an integer key in the above binary search tree recursively.

Source Code:

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

class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

class BST {
private:
    TreeNode* root;

    // Helper function to insert data recursively
    TreeNode* insert(TreeNode* node, int value) {
        if (node == nullptr) {
            return new TreeNode(value);
        }

        if (value <= node->data) {
            node->left = insert(node->left, value);
```

```
    } else {
        node->right = insert(node->right, value);
    }
    return node;
}

// Helper function to perform in-order traversal recursively
void inOrderTraversal(TreeNode* node) {
    if (node == nullptr) {
        return;
    }
    inOrderTraversal(node->left);
    cout << node->data << " ";
    inOrderTraversal(node->right);
}

// Helper function to search for key data recursively
bool search(TreeNode* node, int key) {
    if (node == nullptr) {
        return false;
    }

    if (node->data == key) {
        return true;
    } else if (key < node->data) {
        return search(node->left, key);
    } else {
        return search(node->right, key);
    }
}
```

```
public:
    BST() : root(nullptr) {}

    // Function to insert data into the BST
    void insert(int value) {
        root = insert(root, value);
    }

    // Function to display in-order traversal
    void displayInOrder() {
        cout << "In-order traversal: ";
        inOrderTraversal(root);
        cout << endl;
    }

    // Function to search for a key
    bool search(int key) {
        return search(root, key);
    }
};

int main() {
    BST bst;

    // Insert some data
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
```

```
    bst.insert(60);
    bst.insert(80);

    // Display in-order traversal
    bst.displayInOrder();

    // Search for a key
    int key = 40;
    if (bst.search(key)) {
        cout << key << " found in the tree." << endl;
    } else {
        cout << key << " not found in the tree." << endl;
    }

    return 0;
}
```

Output:

In-order traversal: 20 30 40 50 60 70 80

40 found in the tree.