LAB - 4

* WAP to convert infix expressions to postfix expressions.

```
#include <bits/stdc++.h>
using namespace std;

int prec(char c) {
  if (c == '^') { return 3; }
  else if (c == '/' || c == '*') { return 2; }
  else if (c == '+' || c == '-') { return 1; }
  else { return -1; }
}

char associativity(char c) {
  if (c == '^') { return 'R'; }
  return 'L';
}

void infixToPostfix(string s) {
  stack<char> st;
  string result;

  for (int i = 0; i < s.length(); i++) {
    char c = s[i];
```

```
if((c >= 'a' && c <= 'z') || (c >= 'A' &&
    c <= 'Z') || (c >= '0' && c <= '9')){
        result += c;
} else if (c == '(') { st.push('(');
} else if (c == ')') {
    while (st.top() != '(') {
        result += st.top();
        st.pop();
    }
    st.pop();
} else {
    while(!st.empty() && prec(s[i]) <
        prec(st.top()) || !st.empty() &&
        prec(s[i]) == prec(st.top()) &&
        associativity(s[i]) == 'L') {
            result += st.top();
            st.pop();
    }
    cout << result << endl;
}

int main() {
    string expression;
    cout << "Enter expression: " << endl;
    cin >> expression;

    infixToPostfix(expression);

    return 0;
}
```

LAB - 5

WAP to implement following Queue operations.
1] Insert 2] ~~Update~~ 2] delete
3] Display

```
#include <iostream>

using namespace std;

class Queue {
private:
    int front, rear, size;
    int * arr;

public:
  Queue(int size) : size(size), front(-1),
     rear(-1), arr(new int[size]) {}

  ~Queue() {
       delete[] arr;
  }
```

```
Void queueInsertion() {
  if (rear == size-1) {
    cout << "Queue IS FULL!!" << endl;
    return;
  }
  if (front == -1) {
    front = rear = 0;
    cout << "Enter element: " << endl;
    cin >> arr[rear];
  } else {
    rear++;
    cout << "Enter element: " << endl;
    cin >> arr[rear];
  }
}

Void queueDelete() {
  if (front == -1) {
    cout << "QUEUE IS EMPTY!!" << endl;
    return;
  }
  if (rear == front) {
    cout << arr[front] << ": is
      deleted from Queue" << endl;
    front = rear = -1;
  } else {
    cout << arr[front] << ": is deleted
      from Queue" << endl;
    front++;
  }
}
```

```
    void display() {
        if (front == -1 || front > rear) {
            cout << "QUEUE IS EMPTY!!" << endl;
            return;
        }
        cout << "Queue elements: ";
        for (int i = front; i <= rear; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    cout << "Enter size of queue: " << endl;
    int size;
    cin >> size;
    Queue que(size);

    int choice;
    do {
        cout << "1. Insert" << endl;
        cout << "2. Delete" << endl;
        cout << "3. Display" << endl;
        cin >> choice;
```

```
switch (choice) {
    case 1: que.queueInsertion();
            break;
    case 2: que.queueDeletion();
            break;
    case 3: que.queue.display();
            break;
    case 4: exit(0);
            break;
    default: cout << "Enter valid Choice!"
                  << endl;
    }
} while (choice != 4);
return 0;
}
```

LAB - 6

WAP to implement following Circulex Queue operation
1] Insert 2] Delete 3] Display

```
#include <iostream>

using namespace std;

class CircularQueue {
private:
    int front, rear, size;
    int *arr;

public:
    CircularQueue(int size) : size(size),
        front(-1), rear(-1), arr(new int[size])
    {}

    void queueInsertion() {
        if ((front == rear+1) || ((rear == size-1)
            && (front == 0))) {
            cout << "QUEUE IS FULL!!" << endl;
            return;
        } else if (front == -1) {
            cout << "Enter the Element: " << endl;
            front = rear = 0;
```

```
        cin >> arr[rear];
    } else if(rear == size-1) {
        rear = 0;
        cout << "Enter the element: " <<endl;
        cin >> arr[rear];
    } else {
        cout << "Enter the Element: " <<endl;
        rear++;
        cin >> arr[rear];
    }
}

void queueDelete() {
    if(rear == -1) {
        cout << "QUEUE IS EMPTY!!" <<endl;
        return;
    } else if(rear == front) {
        int temp = arr[front];
        cout << temp << ": is deleted from
                Queue" << endl;
        rear = front = -1;
    } else if(front == size-1) {
        cout << arr[front] << ": is deleted
            from Queue" << endl;
        front = 0;
    } else {
        cout << arr[front] << ": is deleted
            from Queue" << endl;
        front++;
    }
}
```

```
void display() {
  if (rear == -1) {
    cout << "QUEUE IS EMPTY!!"
         << endl;
    return;
  }
  cout << "Queue elements: ";
  if (front > rear) {
    for (int i = front; i <= size-1; i++){
      cout << arr[i] << " ";
    }
    for (int i=0; i <= rear; i++) {
      cout << arr[i] << " ";
    }
  } else {
    for (int i=front; i <= rear; i++){
      cout << arr[i] << " ";
    }
  }
}
};
```

```
int main() {
    cout << "Enter size of queue: " << endl;
    int size;
    cin >> size;
    CircularQueue que(size);

    int choice;
    do {
        cout << "\n1. Insert" << endl;
        cout << "2. Delete" << endl;
        cout << "3. Display" << endl;
        cout << "4. Exit" << endl;
        cout << "\n Enter choice: ";
        cin >> choice;

        switch(choice) {
            case 1: que.queueInsertion(); break;
            case 2: que.queueDelete(); break;
            case 3: que.display(); break;
            case 4: exit(0); break;
            default: cout << "Enter valid choice!"
                << endl; break;
        }
    } while(choice != 4);
    return 0;
}
```

LAB 7 & 8

WAP to implement following operation of singly linked List

1] Insert node at first
2] Insert node at last
3] Inset node such that the linked list is in ascending order.
4] Delete a node at first
5] Delete node before specified position
6] Delete node after specified position

```
#include <iostream>

using namespace std;

class Node {
private
   int data;
   Node* next;

public:
   ~Node() {
     Node * head = this;
     while (head != nullptr) {
       Node* next = head->next;
       delete head;
       head = next;
   }
2. }
```

```
static Node* createLinkedList () {
  Node *head = nullptr, *p = nullptr;
  int size;

  cout << "Enter the number of nodes
       you want to create: ";
  cin >> size;

  head = new Node ();
  p = head;
  head -> next = nullptr;

  cout << "Enter value: " << endl;
  cin >> p -> data;

  for (int i = 1; i < size; i++) {
    p -> next = new Node();
    p = p -> next;

    cout << "Enter Values: " << endl;
    cin >> p -> data;
  }
  p -> next = nullptr;

  return head;
}
```

```
int display() {
  Node * q = this;

  If (q == nullptr()) {
    cout << "LINKED LIST IS EMPTY!!"
         << endl;
  } else {
    cout << "Linked List value: " << endl;
    while(q != nullptr()) {
      cout << "value is: " << q->data
           << endl;
      q = q->next;
    }
  }
  return 0;
}

Node* insertAtFirst() {
  Node* head = this;
  Node* p = new Node();

  cout << "Enter value: " << endl;
  cin >> p->data;

  p->next = head;
  head = p;

  return head;
}
```

```
Node * insertAtLast(){
  Node *head = this;
  Node *p = new Node();
  Node *q = nullptr;
  int value;

  cout << "Enter value: " << endl;
  cin >> value;
  p->data = value;
  p->next = nullptr;

  if(head == nullptr) { head = p; }
  else {
    for(q = head; q->next != nullptr;
        q = q->next) { }
    q->next = p;
  }

  return head;
}

Node* insertSortedNodes(){
  Node *head = this;
  Node *p = new Node();
  int value;

  cout << "Enter element to insert: ";
  cin >> value;
  p->data = value;
  p->next = nullptr;
```

```
if(head == nullptr) {
  cout << "Inserting at first place
    because there no nodes present in
    linked list" << endl;

    head = p;
} else {
  if(p->data < head->data) {
    p->next = head;
    head = p;
  } else {
    for(q = head; (q->next != nullptr) &&
        (value > q->next->data);
        q = q->next) {}

        p->next = q->next;
        q->next = p;
  }
}
return head;
}
Node * deleteAtFirst() {
  Node * head = this, *p = nullptr;

  if(head == nullptr) {
    cout << "LINKED LIST IS EMPTY!!"
      << endl;
  } else {
    p = head;
    head = p->next;
```

```
        cout << p->data << " : is deleted"
             << endl;

        free(p);
    }
    return head;
}

Node * deleteAtSpecifiedlocation_after(){
 Node *head = this;
 Node* p = nullptr, *q = nullptr;
 int location, i = 1;

 if(head == nullptr){
  cout << "LINKED LIST IS EMPTY!!"
       << endl;
 }else{
  cout << "Enter location of Node to
   delete that Node" << endl;
  cin >> location;

 if(location <= 0){ cout << "ENTER
   POSITIVE LOCATION!!" << endl;
  return head; }

 if(head->next == nullptr || location==
      1){
    p = head;
    head = p->next;
```

```
cout << p->data << ": is deleted"
     << endl;

  free(p);
} else {
  for (q = head; q->next->next != NULL && i < location; q = q->next,
    i++) { }

  if ((q->next == NULL) && (i < location)
      ) {
    cout << "INVALID LOCATION!!
         << endl;
  } else {
    p = q->next;
    q->next = p->next;
    cout << p->data << ": is deleted"
         << endl;
    free(p);
  }
}

return head;
}
```

```
Node * deleteAtSpecificlocation_before(){
  Node* head = this;
  Node * p = nullptr;
  Node * q = nullptr;
  int location, i = 1;

  if(head == nullptr){
    cout << "LINKED LIST IS EMPTY!!"
         << endl;
  }else{
    cout << "Enter location of Node
      to delete that Node" << endl;
    cin >> location;

    if(location <= 0){ cout << "
      ENTER POSITIVE LOCATION!!"
      << endl; return head; }

    if(head -> next == nullptr || location
          == 1){
      p = head;
      head = p -> next;
      cout << p -> data << " is
          deleted" << endl;

      free(p);
    }else{
      for(q = head; q -> next -> next !=
        nullptr && i < location-1;
                    q = q -> next, i++);
    }}
```

```
if((q->next==NULL)&&(i<loc))
                    {
    cout<<"INVALID LOCATION!!"
        <<endl;
  }else{
    p=q->next;
    q->next=p->next;
    cout<<p->data<<": is deleted"
        <<endl;

    free(p);
  }
  }
  }

  return head;
}
};

int main(){
  node* head=NULL;
  int choice;

do{
  cout<<endl;
  cout<<"1. Create"<<endl;
  cout<<"2. Display"<<endl;
  cout<<"3. Exit"<<endl;
  cout<<"4. Insert at First"<<endl;
```

```
cout << " 5. Insert at Last " << endl;
cout << " 6. Insert value by number
     at sorted postion " << endl;
cout << " 7. Delete at First " << endl;
cout << " 8. Delete after specified
     location " << endl;
cout << " 9. Delete before specified
     location " << endl;

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

switch (choice){
 case 1: head = Node1; CreateLinkedList
                    (); break;
 case 2: head->display(); break;
 case 3: exit(0); break;
 case 4: head = head -> insertAtFirst();
         break;
 case 5: head = head -> insertAtLast(); break;
 case 6: head = head -> insertSortedNodes();
         break;
 case 7: head = head -> deleteAtFirst();
         break;
 case 8: head = head -> deleteAtSpecifiedLocation
                    _after();
         break;
 case 9: head = head -> deleteAtSpecifiedLocation
                    _before(); break;
```

```
        default: cout << endl << " ENTER
            VALID CHOICE " << endl; break;
    }
}while (choice != 3);

delete head;

return 0;
}
```

LAB - 9 & 10

WAP to implement following operations of doubly linked list.

1] Insert a node at the front
2] Insert a node at the End
3] Delete Node at the End
4] Delete Node at specified location.

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

class Node {
private:
    int data;
    Node *next;
    Node * previous;

public:
    ~Node () {
    Node * head = this;
    while (head != nullptr) {
      Node * next = head -> next;
      delete head;
       head = next;
    }
}
```

```
Node* CreateLinkedList() {
  Node *head = nullptr;
  Node *p = nullptr, *q = nullptr;

int size;

cout << "Enter the size of Nodes: ";
cin >> size;

head = new Node();
p = head;
q = head;
p->next = nullptr;
p->previous = nullptr;

cout << "Enter value: " << endl;
cin >> head->data;
p->data = head->data;

for (int i = 1; i < size; i++) {
  p->next = new Node();
  q = p->next;
  q->previous = p;
  p = p->next;

  cout << "Enter values: " << endl;
  cin >> p->data;
}

p->next = nullptr;
return head;
```

```
int display() {
  Node * q = this;

  if (q == nullptr) {
    cout << "Linked List is empty: "
         << endl;
  } else {
    cout << "Linked List values: "
         << endl;
    while (q != nullptr) {
    cout << "value is: " <<
      q->data << endl;
      q = q->next;
    }
  }

  return 0;
}

int reverseDisplay() {
  Node * head = this;
  Node * p = head;

  if (head == nullptr) {
    cout << "Linked List is empty! "
         << endl;
  } else {
    for (p = head; p->next; p = p->next) {}
    cout << "Linked List values: "
         << endl;
```

```
while(p->previous != nullptr) {
    cout << "value is: " << p->data
         << endl;
    p = p->previous;
}
cout << "value is: " << p->data
     << endl;
return 0;
}

Node * insertAtFirst() {
Node * head = this;
Node *p = new Node();

cout << "Enter a value: " << endl;
cin >> p->data;

p->next = nullptr;
p->previous = nullptr;

if(head == nullptr) {
  head = p;
} else {
  p->next = head;
  head->previous = p;
  head = p;
}

return head;
}
```

Last

```
Node * insertAtFirst() {
  Node * head = this;
  Node * p = new Node();
  Node * q = nullptr;

  cout << "Enter value: "
       << endl;
  cin >> p->data;

  p->next = nullptr;
  p->previous = nullptr;

  if(head == nullptr) {
    head = p;
  } else {
    for(q=head; q->next != nullptr;
        q = q->next) {}
    q->next = p;
    p->previous = q;
  }
  return head;
}
```

```
Node *deleteAtSpecifiedLocation() {
  Node *head = this;
  Node *p = nullptr;
  Node *q = nullptr;
  int location, i = 1;

  if (head == nullptr) {
    cout << "LinkedLIST IS EMPTY!!" << endl;
  } else {
    cout << "Enter location: " << endl;
    cin >> location;

    if (location < 0) {
      cout << "Enter POSITIVE LOCATION!!"
           << endl;
      return head;
    }

    if (head->next == nullptr || location == 1)
    {
      p = head;
      head = nullptr;
      cout << p->data << " is deleted"
           << endl;
      free(p);
    } else {
      for (q = head; (q->next != nullptr)
        && (i < location - 1); q = q->next,
        i++) {}
```

```
if((q->next == nullptr) && (j<location)){
  cout << "INVALID LOCATION!!" << endl;
} else {
  p = q->next;
  q->next = p->next;
  p->next->previous = q;

  cout << p->data << ": is deleted"
       << endl;
  free(p);
}
}
}
return head;
}

int main() {
  Node* head = nullptr;
  int choice;

  do {
    cout << endl;
    cout << "1. Create" << endl;
    cout << "2. Display" << endl;
    cout << "3. Exit" << endl;
    cout << "4. Insert at Front" << endl;
    cout << "5. Insert at End" << endl;
    cout << "6. Delete at last" << endl;
```

```
cout << " 7. Delete at specified location"
     << endl;

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

switch (choice) {
  case 1: head = Node::createLinkedList
          ();
          break;
  case 2: head->display();
          break;
  case 3: exit(0); break;
  case 4: head = head->insertAtFirst();
          break;
  case 5: head = head->insertAtEnd();
          break;
  case 6: head = head->deleteAtEnd();
          break;
  case 7: head = head->deleteAtSpecified
          location(); break;

  default:
    cout << endl << "ENTER VALID
    CHOICE" << endl; break;
}
} while (choice != 4);
delete head;
return 0;
```