Name: Diotangshu Dey

Roll No: 20CS8018

Assignment 5

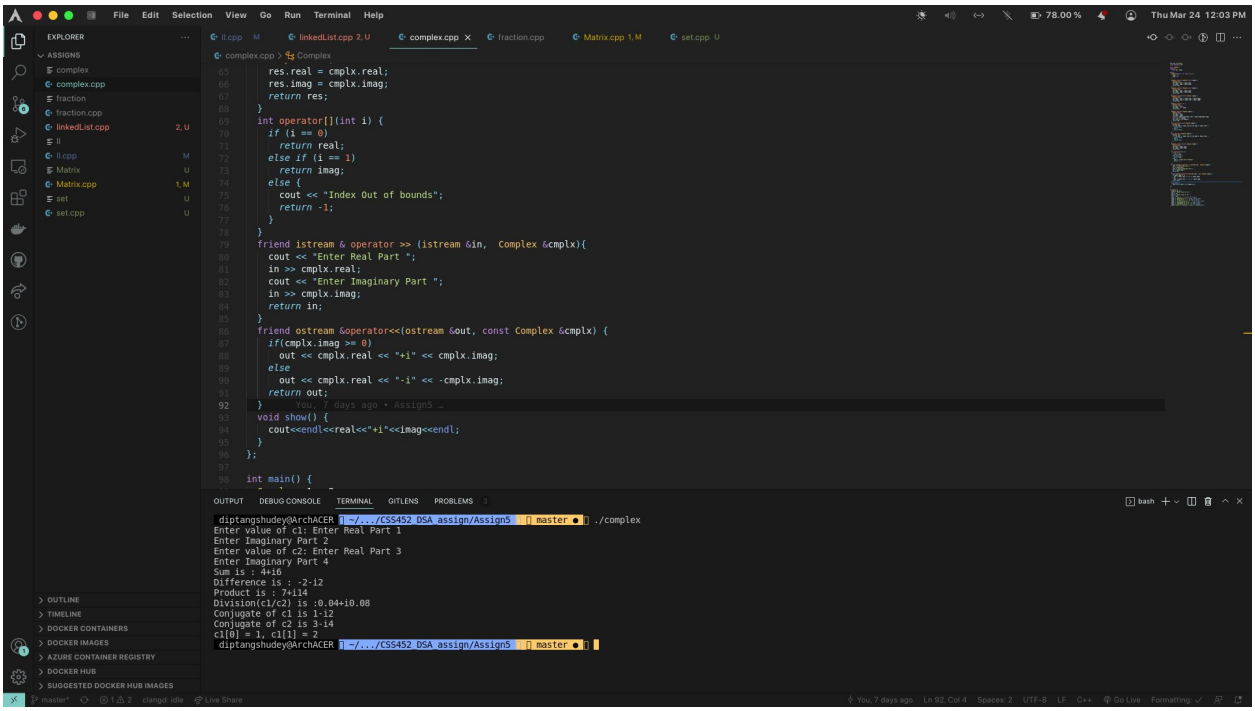
1. Code:

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

class Complex {
private:
  float real, imag;

public:
  Complex(float r = 0, float i = 0) {
    real = r;
    imag = i;
  }

  Complex operator+(Complex const &cmplx) {
    Complex res;
    res.real = real + cmplx.real;
    res.imag = imag + cmplx.imag;
    return res;
  }
  Complex operator-(Complex const &cmplx) {
    Complex res;
    res.real = real - cmplx.real;
    res.imag = imag - cmplx.imag;
    return res;
  }
  Complex operator*(const Complex &cmplx) {
    Complex res;
    res.real = real * (cmplx.real + cmplx.imag);
    res.imag = imag * (cmplx.real + cmplx.imag);
    return res;
  }
  Complex operator !() {
    Complex res;
    res.real = real;
    res.imag = -1 * imag;
    return res;
  }
  Complex operator/(Complex cmplx) {
    Complex res;
    res.real = real;
    res.imag = imag;
    float d = cmplx.real*cmplx.real + cmplx.imag*cmplx.imag;
    res = res * (!cmplx);
    res.real/=d; res.imag/=d;
```

```
  return res;
}
bool operator==(const Complex &cmplx) {
  Complex res;
  if (res.real == cmplx.real && res.imag == cmplx.real) {
    return true;
  } else {
    return false;
  }
}
bool operator!=(Complex cmplx) {
  Complex res;
  if (res.real == cmplx.real && res.imag == cmplx.real) {
    return false;
  } else {
    return true;
  }
}
Complex operator=(const Complex &cmplx) {
  Complex res;
  res.real = cmplx.real;
  res.imag = cmplx.imag;
  return res;
}
int operator[](int i) {
  if (i == 0)
    return real;
  else if (i == 1)
    return imag;
  else {
    cout << "Index Out of bounds";
    return -1;
  }
}
friend istream & operator >> (istream &in,  Complex &cmplx){
  cout << "Enter Real Part ";
  in >> cmplx.real;
  cout << "Enter Imaginary Part ";
  in >> cmplx.imag;
  return in;
}
friend ostream &operator<<(ostream &out, const Complex &cmplx) {
  if(cmplx.imag >= 0)
    out << cmplx.real << "+i" << cmplx.imag;
  else
    out << cmplx.real << "-i" << -cmplx.imag;
  return out;
}
void show() {
  cout<<endl<<real<<"+i"<<imag<<endl;
```

```
  }
};

int main() {
  Complex c1, c2;
  cout << "Enter value of c1: ";
  cin >> c1;
  cout << "Enter value of c2: ";
  cin >> c2;
  cout << "Sum is : " << c1 + c2 << endl;
  cout << "Difference is : " << c1 - c2 << endl;
  cout << "Product is : " << c1 * c2 << endl;
  cout << "Division(c1/c2) is : " << c1 / c2 << endl;
  cout << "Conjugate of c1 is " << !c1 << endl;
  cout << "Conjugate of c2 is " << !c2 << endl;
  cout<<"c1[0] = "<<c1[0]<<", c1[1] = "<<c1[1]<<endl;
}
```

Output:



2. Code:

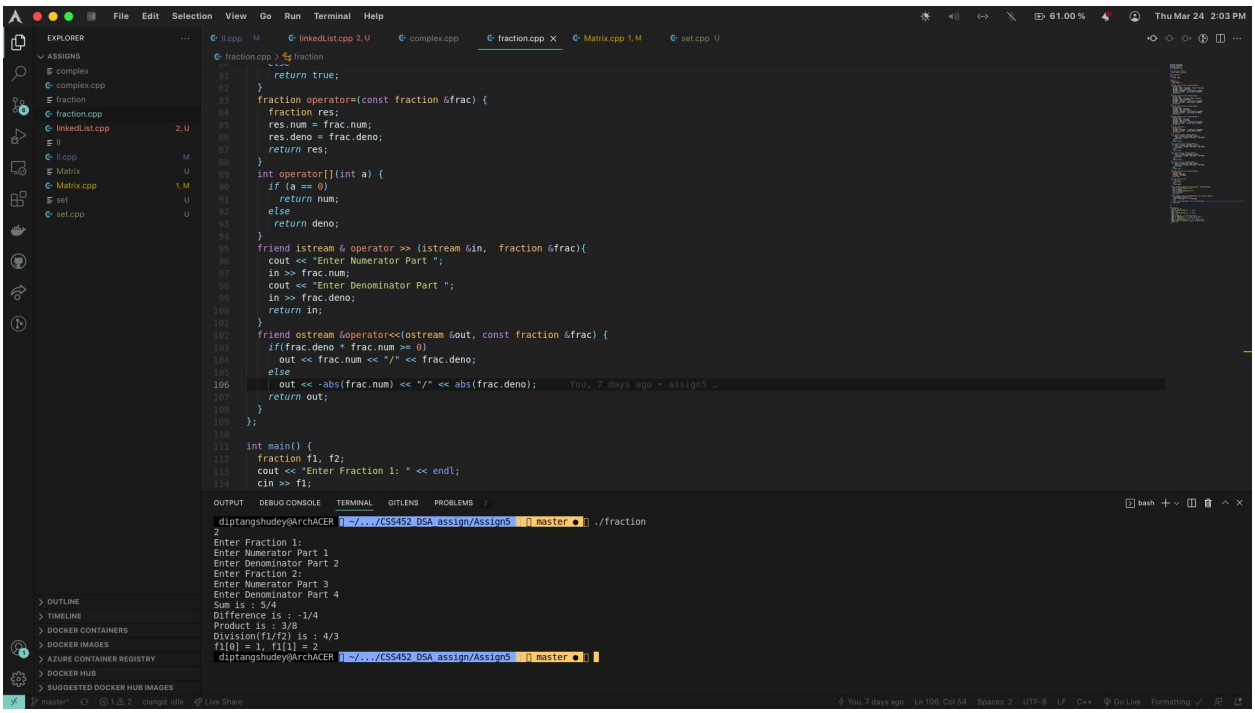
```
#include <iostream>
#include <algorithm>
#include<cstdlib>
using namespace std;

int lcm(int a, int b) {
  return (a*b/__gcd(a,b));
}
```

```
class fraction {
private:
  int num, deno;

public:
  fraction() {
    num = deno = 1;
  }
  fraction operator+(const fraction &frac) {
    fraction res;
    res.num = (num * frac.deno) + (deno * frac.num);
    res.deno = deno * frac.deno;
    res.deno = res.deno / __gcd(res.num, res.deno);
    res.num = res.num / __gcd(res.num, res.deno);
    return res;
  }
  fraction operator-(const fraction &frac) {
    fraction res;
    res.num = num * frac.deno - deno * frac.num;
    res.deno = deno * frac.deno;
    res.deno = res.deno / __gcd(res.num, res.deno);
    res.num = res.num / __gcd(res.num, res.deno);
    return res;
  }
  fraction operator*(const fraction &frac) {
    fraction res;
    res.num = num * frac.num;
    res.deno = deno * frac.deno;
    res.deno = res.deno / __gcd(res.num, res.deno);
    res.num = res.num / __gcd(res.num, res.deno);
    return res;
  }
  fraction operator/(const fraction &frac) {
    fraction res;
    res.num = num * frac.deno;
    res.deno = deno * frac.num;
    res.deno = res.deno / __gcd(res.num, res.deno);
    res.num = res.num / __gcd(res.num, res.deno);
    return res;
  }
  fraction operator*() {
    fraction res;
    res.deno = res.deno / __gcd(res.num, res.deno);
    res.num = res.num / __gcd(res.num, res.deno);
    return res;
  }
  bool operator==(const fraction &frac) {
    if (num * (lcm(deno, frac.deno) / deno) ==
        frac.num * (lcm(deno, frac.deno) / frac.deno))
      return true;
```

```
  else
   return false;
}
bool operator!=(const fraction &frac) {
  if (num * (lcm(deno, frac.deno) / deno) ==
      frac.num * (lcm(deno, frac.deno) / frac.deno))
    return false;
  else
   return true;
}
bool operator>(const fraction &frac) {
  if (num * (lcm(deno, frac.deno) / deno) >
      frac.num * (lcm(deno, frac.deno) / frac.deno))
    return true;
  else
   return false;
}
bool operator<(const fraction &frac) {
  if (num * (lcm(deno, frac.deno) / deno) >
      frac.num * (lcm(deno, frac.deno) / frac.deno))
    return false;
  else
   return true;
}
fraction operator=(const fraction &frac) {
  fraction res;
  res.num = frac.num;
  res.deno = frac.deno;
  return res;
}
int operator[](int a) {
  if (a == 0)
    return num;
  else
   return deno;
}
friend istream & operator >> (istream &in,  fraction &frac){
  cout << "Enter Numerator Part ";
  in >> frac.num;
  cout << "Enter Denominator Part ";
  in >> frac.deno;
  return in;
}
friend ostream &operator<<(ostream &out, const fraction &frac) {
  if(frac.deno * frac.num >= 0)
    out << frac.num << "/" << frac.deno;
  else
    out << -abs(frac.num) << "/" << abs(frac.deno);
  return out;
}
```

```
};

int main() {
  fraction f1, f2;
  cout << "Enter Fraction 1: " << endl;
  cin >> f1;
  cout << "Enter Fraction 2: " << endl;
  cin >> f2;
  cout << "Sum is : " << f1 + f2 << endl;
  cout << "Difference is : " << f1 - f2 << endl;
  cout << "Product is : " << f1 * f2 << endl;
  cout << "Division(f1/f2) is : " << f1 / f2 << endl;
  cout<<"f1[0] = "<<f1[0]<<", f1[1] = "<<f1[1]<<endl;
  return 0;
}
```

Output:



3. Code:

```
#include "bits/stdc++.h"
using namespace std;
class Matrix
{
  int **a;
  int r, c, t;

public:
  Matrix()
  {
```

```
    r = 0;
    c = 0;
    t = -1;
    a = new int *[r];
}
Matrix(int R, int C)
{
    r = R;
    c = C;
    t = -1;
    a = new int *[r];
    for (int i = 0; i < r; i++)
    {
        a[i] = new int[c];
    }
}
Matrix(const Matrix &M)
{
    r = M.r;
    c = M.c;
    t = -1;
    a = new int *[r];
    for (int i = 0; i < r; i++)
    {
        a[i] = new int[c];
        for (int j = 0; j < c; j++)
        {
            a[i][j] = M.a[i][j];
        }
    }
}
bool checkEqualOrder(const Matrix &M)
{
    if (M.c == c && M.r == r)
        return true;
    return false;
}
Matrix operator+(const Matrix &M)
{
    if (checkEqualOrder(M))
    {
        for (int i = 0; i < r; i++)
        {
            for (int j = 0; j < c; j++)
            {
                a[i][j] += M.a[i][j];
            }
        }
    }
    return *this;
```

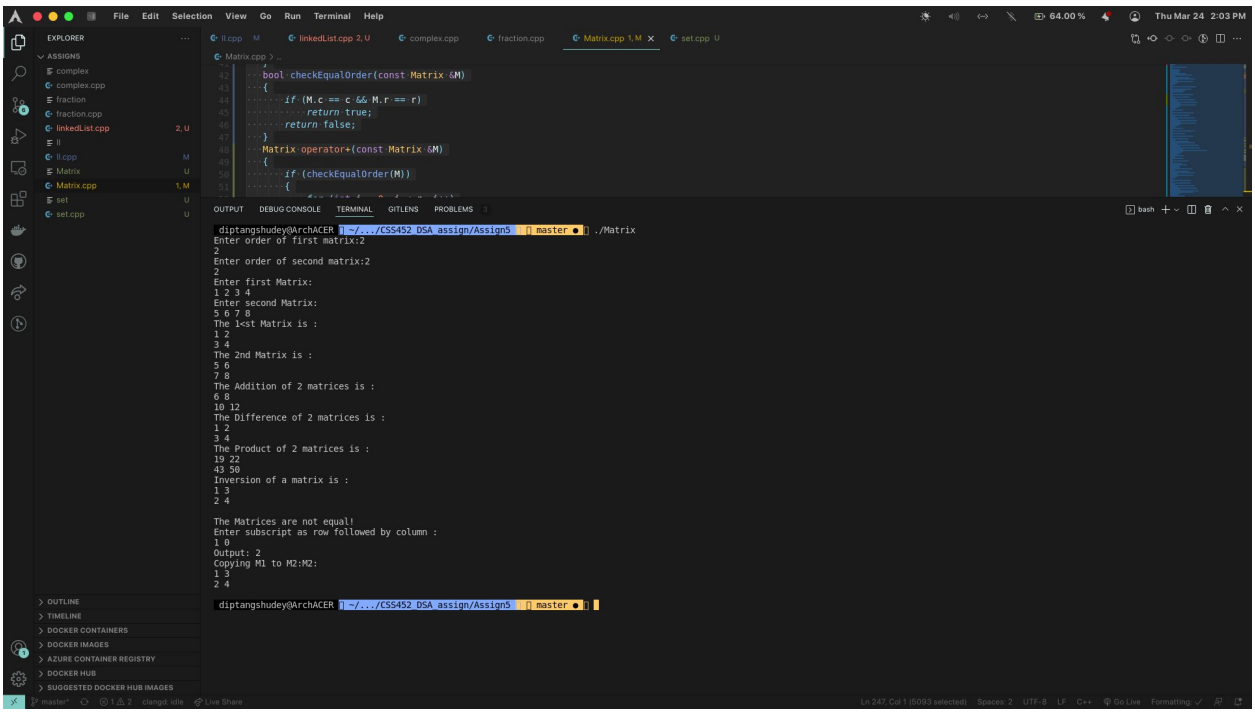
```
}
Matrix operator-(const Matrix &M)
{
   if (checkEqualOrder(M))
   {
      for (int i = 0; i < r; i++)
      {
         for (int j = 0; j < c; j++)
         {
            a[i][j] -= M.a[i][j];
         }
      }
   }
   return *this;
}
bool checkMulOrder(const Matrix &M)
{
   return c == M.r;
}
Matrix operator*(const Matrix &M)
{
   if (checkMulOrder(M))
   {
      Matrix M2(r, M.c);
      for (int i = 0; i < r; i++)
      {
         for (int j = 0; j < M.c; j++)
         {
            for (int k = 0; k < c; k++)
            {
               M2.a[i][j] += a[i][k] * M.a[k][j];
            }
         }
      }
      return M2;
   }
   return *this;
}
void Copy(const Matrix &M)
{
   delete[] a;
   a = new int *[M.r];
   r = M.r;
   c = M.c;
   for (int i = 0; i < r; i++)
   {
      a[i] = new int[c];
      for (int j = 0; j < c; j++)
      {
         a[i][j] = M.a[i][j];
```

```
        }
    }
}
bool Compare(const Matrix &M)
{
    if (r == M.r && c == M.c)
    {
        for (int i = 0; i < r; i++)
        {
            for (int j = 0; j < c; j++)
            {
                if (a[i][j] != M.a[i][j])
                    return false;
            }
        }
        return true;
    }
    else
        return false;
}
Matrix operator!()
{
    for (int i = 0; i < r; i++)
    {
        for (int j = i; j < c; j++)
        {
            int t = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = t;
        }
    }
    return *this;
}
int *operator[](int x)
{
    if (x >= r)
    {
        t = -1;
        return NULL;
    }
    t = x;
    return a[x];
}
int operator[](long x)
{
    if (x > c || t == -1)
    {
        cout << "Out of Bounds access!" << endl;
        exit(1);
        return 2e-5;
```

```
        }
        return a[t][x];
    }
    bool operator==(const Matrix &M)
    {
        return Compare(M);
    }
    bool operator!=(const Matrix &M)
    {
        return !Compare(M);
    }
    void operator=(const Matrix &M)
    {
        Copy(M);
    }
    friend ostream &operator<<(ostream &x, const Matrix &M)
    {
        for (int i = 0; i < M.r; i++)
        {
            for (int j = 0; j < M.c; j++)
            {
                x << M.a[i][j] << " ";
            }
            x << endl;
        }
        return x;
    }
    friend istream &operator>>(istream &x, Matrix &M)
    {
        for (int i = 0; i < M.r; i++)
        {
            for (int j = 0; j < M.c; j++)
            {
                x >> M.a[i][j];
            }
        }
        return x;
    }
};
int main()
{
    int r1, r2, c1, c2;
    cout << "Enter order of first matrix:";
    cin >> r1 >> c1;
    cout << "Enter order of second matrix:";
    cin >> r2 >> c2;
    Matrix M1(r1, c1), M2(r2, c2);
    cout << "Enter first Matrix:\n";
    cin >> M1;
    cout << "Enter second Matrix:\n";
```

```
  cin >> M2;
  cout << "The 1<st Matrix is : \n";
  cout << M1;
  cout << "The 2nd Matrix is : \n";
  cout << M2;
  Matrix M3(r1, c1);
  M3 = M1 + M2;
  cout << "The Addition of 2 matrices is : \n";
  cout << M3;
  Matrix M4(r1, c1);
  M4 = M1 - M2;
  cout << "The Difference of 2 matrices is : \n";
  cout << M4;
  Matrix M5(r1, c1);
  M5 = M1 * M2;
  cout << "The Product of 2 matrices is : \n";
  cout << M5;
  cout << "Inversion of a matrix is : \n"
      << (!M1) << endl;
  if (M1 == M2)
  {
     cout << "The Matrices are equal!" << endl;
  }
  else
  {
     cout << "The Matrices are not equal!" << endl;
  }
  cout << "Enter subscript as row followed by column :\n";
  int x, y;
  cin >> x >> y;
  cout << "Output: " << M1[x][y] << endl;
  cout << "Copying M1 to M2:";
  M2 = M1;
  cout << "M2: \n"<< M2 << endl;
  return 0;
}
```

Output:



4. Code:

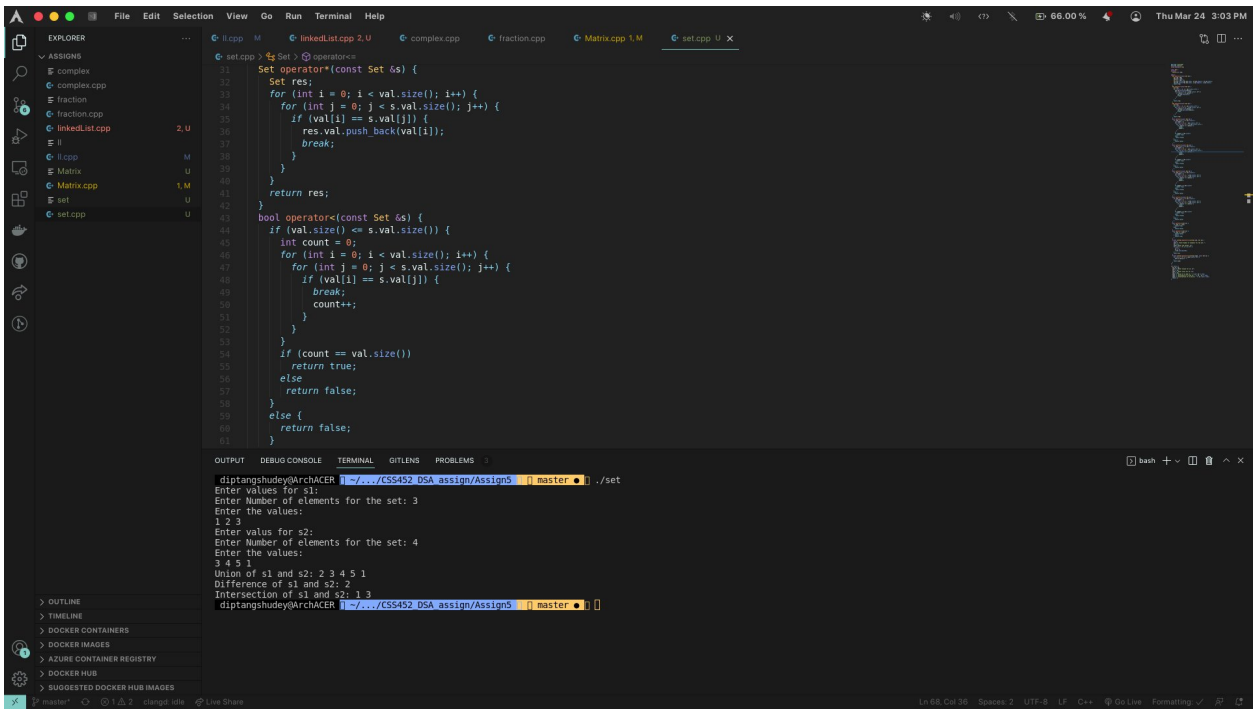
```
#include <iostream>
#include <vector>
using namespace std;

class Set {
private:
  vector<int> val;

public:
  Set operator+(const Set &s) {
    Set res, tmp;
    tmp.val = val;
    Set t1 = tmp - s;
    res.val.insert(res.val.end(), t1.val.begin(), t1.val.end());
    res.val.insert(res.val.end(), s.val.begin(), s.val.end());
    return res;
  }
  Set operator-(const Set &s) {
    Set res;
    for (int i = 0; i < val.size(); i++) {
      res.val.push_back(val[i]);
      for (int j = 0; j < s.val.size(); j++) {
        if (val[i] == s.val[j]) {
          res.val.pop_back();
          break;
        }
      }
    }
    return res;
```

```
}
Set operator*(const Set &s) {
  Set res;
  for (int i = 0; i < val.size(); i++) {
    for (int j = 0; j < s.val.size(); j++) {
      if (val[i] == s.val[j]) {
        res.val.push_back(val[i]);
        break;
      }
    }
  }
  return res;
}
bool operator<(const Set &s) {
  if (val.size() <= s.val.size()) {
    int count = 0;
    for (int i = 0; i < val.size(); i++) {
      for (int j = 0; j < s.val.size(); j++) {
        if (val[i] == s.val[j]) {
          break;
          count++;
        }
      }
    }
    if (count == val.size())
      return true;
    else
     return false;
  }
  else {
    return false;
  }
}
bool operator<=(const Set &s) {
  if (val.size() <= s.val.size()) {
    int count = 0;
    for (int i = 0; i < val.size(); i++) {
      for (int j = 0; j < s.val.size(); j++) {
        if (val[i] == s.val[j]) {
          break;
          count++;
        }
      }
    }
    if (count == val.size())
      return true;
    else
     return false;
  }
  else {
```

```
    return false;
  }
}
bool operator>(const Set &s) {
  if (val.size() >= s.val.size()) {
    int count = 0;
    for (int i = 0; i < s.val.size(); i++) {
      for (int j = 0; j < val.size(); j++) {
        if (val[j] == s.val[i]) {
          break;
          count++;
        }
      }
    }
    if (count == s.val.size())
      return true;
    else
     return false;
  }
  else {
    return false;
  }
}
bool operator>=(const Set &s) {
  if (val.size() >= s.val.size()) {
    int count = 0;
    for (int i = 0; i < s.val.size(); i++) {
      for (int j = 0; j < val.size(); j++) {
        if (val[j] == s.val[i]) {
          break;
          count++;
        }
      }
    }
    if (count == s.val.size())
      return true;
    else
     return false;
  }
  else {
    return false;
  }
}
bool operator==(Set &s) {
  if (val == s.val)
    return true;
  else
    return false;
}
bool operator!=(Set &s) {
```

```
    if (val == s.val) {
      return false;
    } else {
      return true;
    }
  }
  friend istream &operator>>(istream &in, Set &s) {
    int n;
    cout << "Enter Number of elements for the set: ";
    in >> n;
    cout<<"Enter the values: \n";
    for (int i = 0; i < n; i++) {
      int t;
      in >> t;
      s.val.push_back(t);
    }
    return in;
  }
  friend ostream &operator<<(ostream &out, const Set &s) {
    for (int i = 0; i < s.val.size(); i++) {
      out<<s.val[i]<<" ";
    }
    return out;
  }
};

int main() {
  Set s1, s2;
  cout << "Enter values for s1: \n";
  cin >> s1;
  cout << "Enter valus for s2: \n";
  cin >> s2;
  cout << "Union of s1 and s2: " << s1 + s2 << endl;
  cout << "Difference of s1 and s2: " << s1 - s2 << endl;
  cout << "Intersection of s1 and s2: " << s1 * s2 << endl;
  return 0;
}
```

Output:



5. Code :

```
#include <iostream>
using namespace std;
class Node
{

public:
  int data;
  Node *next_link;
  static Node *avail;
  Node(int, Node *);
  Node(const Node &);
  void *operator new(size_t size);
  void operator delete(void *p);
  void display();
};

Node::Node(int x = 0, Node *nd = NULL)
{

  data = x;
  next_link = nd;
}

Node::Node(const Node &nd)
{
  data = nd.data;
  next_link = nd.next_link;
}
```

```
void *Node::operator new(size_t size)
{
  void *p;
  if (avail == NULL)
  {
     p = ::operator new(size);
  }
  else
  {
     p = avail;
     avail = avail->next_link;
  }
  return p;
}

void Node::operator delete(void *p)
{
  Node *t = (Node *)p;
  t->next_link = avail;
  t->data = 0;
  avail = t;
  return;
}

Node *Node::avail = NULL;

void Node::display()
{
  cout << data;
}

class list
{
private:
  Node head;
  static list lstat;

public:
  list()
  {
     head.data = 0;
     head.next_link = NULL;
  }
  list(const list &ll)
  {
     head.data = ll.head.data;
     Node *p = ll.head.next_link;
     if (p == NULL)
     {
        head.next_link = NULL;
```

```
    }
    else
    {
        Node *q = &head;
        while (p != NULL)
        {
            Node *temp = new Node(p->data);
            q->next_link = temp;
            q = q->next_link;
            p = p->next_link;
        }
    }
}
~list(){
    Node *p = head.next_link;
    while (p != NULL)
    {
        Node *temp = p;
        p = p->next_link;
        delete temp;
    }
    head.data = 0;
}

list &operator+(const list &ll){
    static list lstat;
    lstat.~list();
    lstat = *this;
    Node *p = &lstat.head;
    while (p->next_link != NULL)
    {
        p = p->next_link;
    }

    Node *q = ll.head.next_link;
    while (q != NULL)
    {
        p->next_link = new Node(q->data);
        q = q->next_link;
        p = p->next_link;
    }
    lstat.head.data = head.data + ll.head.data;
    return lstat;
}
list &operator!(){
    if (head.data < 2)
    {
        return *this;
    }
    static list lstat;
```

```
    lstat.~list();
    lstat.head.data = head.data;
    Node *p = head.next_link;

    while (p != NULL)
    {
        lstat.head.next_link = new Node(p->data, lstat.head.next_link);
        p = p->next_link;
    }
    return lstat;
}
bool operator==(const list &ll){
    if (this == &ll)
    {
        return true;
    }
    if (head.data != ll.head.data)
    {
        return false;
    }
    Node *p = head.next_link;
    Node *q = ll.head.next_link;
    while (p != NULL)
    {
        if (p->data != q->data)
        {
            return false;
        }
        p = p->next_link;
        q = q->next_link;
    }
    return true;
}
list &operator=(const list &ll){
    if (this == &ll)
    {
        return *this;
    }
    this->~list();
    this->head.data = ll.head.data;
    Node *p = &(this->head);
    Node *q = ll.head.next_link;
    while (q != NULL)
    {
        p->next_link = new Node(q->data);
        p = p->next_link;
        q = q->next_link;
    }
    return *this;
}
```

```
    int operator[](int index){
        if (index >= head.data)
        {
            return -1e5;
        }
        Node *p = head.next_link;
        int i = 0;
        while (p != NULL)
        {
            if (i == index)
            {
                return p->data;
            }
            i++;
            p = p->next_link;
        }
        return 1e-5;
    }

    friend ostream &operator<<(ostream &os, list &ll){
        Node *p = &(ll.head);
        p = p->next_link;
        while (p != NULL)
        {
            p->display();
            os << "-->";
            p = p->next_link;
        }
        os << "NULL\n";
        return os;
    }
    friend istream &operator>>(istream &is, list &ll){
        cout << "Enter no. of nodes: ";
        is >> ll.head.data;
        Node *p = &ll.head;
        while (p->next_link != NULL)
        {
            p = p->next_link;
        }
        for (int i = 0; i < ll.head.data; i++)
        {
            Node *temp = new Node();
            is >> temp->data;
            p->next_link = temp;
            p = p->next_link;
        }
        return is;
    }
};
```

```
int main(void)
{
  list l1;
  cin >> l1;
  list l2;
  cin >> l2;
  cout << "List 1 : " << l1 << endl;
  cout << "List 2 : " << l2 << endl;
  cout << "Concatinating 2 lists :" << l1 + l2 << endl;
  cout << "Reversing the list : " << (!l1) << endl;
  if (l1 == l2)
  {
     cout << "Lists are equal\n";
  }
  else
  {
     cout << "Lists are not equal\n";
  }
  cout << "Index 1 of 1st list is : " << l1[1] << endl;
}
```

Output: