```
#ifndef Sequence_h
#define Sequence_h
#include <iostream>
#include <string>
using namespace std;

using ItemType = unsigned long;
class Sequence
{
public:
    //new Sequence();
    Sequence();
    // Create an empty sequence
(i.e., one with no items)

    ~Sequence(); //Destructor
    Sequence(const Sequence&
other); //copy constructor
    Sequence& operator=(const
Sequence& rhs); //assigment
operator


private:
    struct Node
    {
        ItemType m_value;
        Node* next;
        Node* prev;
    };
    Node *head, *tail;
    int m_size;
};
int subsequence(const Sequence&
seq1, const Sequence& seq2);
void interleave(const Sequence&
seq1, const Sequence& seq2,
Sequence& result);
#endif



int subsequence(const Sequence& seq1,
const Sequence& seq2)
{
    if (seq1.size() == 0 || seq2.size() == 0 ||
seq1.size() == 1 || seq2.size() == 1)
    {
        return -1;
    }
    ItemType value1, value2, value1_s,
value2_s;
    for (int i = 0; i < seq2.size()-1; ++i)
    {
        seq2.get(i, value2);
        seq2.get(i+1, value2_s);
        for (int j = 0; j < seq1.size()-1; ++j)
        {
            seq1.get(j, value1);
            seq1.get(j+1, value1_s);
            if (value1 == value2 &&
                value1_s == value2_s)
            {
                return j;
            }
        }
    }
    return -1;
}

void interleave(const Sequence& seq1, const
Sequence& seq2, Sequence& result)
{
    Sequence r1;
    int p1 = 0;
    int p2 = 0;
    ItemType value1, value2;

    if (seq1.size() == 0) //If seq1 is empty return
seq2
        r1 = seq2;
    else if (seq2.size() == 0) //If seq2 is empty return
seq1
        r1 = seq1;
    else
    {
        while(p1 < seq1.size() || p2 < seq2.size())
        {
            if (p2 < seq2.size())
            {
                seq2.get(p2, value2);
                r1.insert(p2, value2);
                p2++;
            }
            if (p1 < seq1.size())
            {
                seq1.get(p1, value1);
                r1.insert(p1, value1);
                p1++;
            }
        }
    }

    result = r1;
}
```

```
#include "Sequence.h"

//new Constructor
Sequence::Sequence()
{
    head = nullptr;
    tail = nullptr;
    m_size = 0;
}
//Destructor
Sequence::~Sequence()
{
    Node *p = head;
    while(p != nullptr)
    {
        Node *n = p->next;
        delete p;
        p = n;
    }
}

//Copy Constructor
Sequence::Sequence(const Sequence& other)
{
    m_size = other.m_size;
    if (other.head == nullptr)
    {
        head = nullptr;
        head-> prev = nullptr;
    }
    else
    {
        head = new Node; //creates the head Node
        head ->m_value = other.head-> m_value; //sets value
        head -> prev = nullptr; //makes prev point to null
    }

    Node *p = head;
    Node *n = other.head->next;

    while(n != nullptr)
    {
        p -> next = new Node; //creates new Node to store
the next item
        p -> next -> prev = p; //links the next Node's prev to
current Node
        p = p-> next; //now pointing to the newly made Node
        p->m_value = n->m_value; //Sets the value
        n = n->next;// advances other
    }
    tail = p; //makes tail point to last Node
    p->next = nullptr; //marks end of list
}

//Assignment Operator
Sequence& Sequence::operator=(const Sequence& rhs)
{
    if (this != &rhs) {
        Sequence temp(rhs);
        swap(temp);
    }
    return *this; }
```

```
int Sequence::insert(int pos, const ItemType&
value)
{
    if (pos == 0)
    {
        Node* p = new Node; //creates the inserted
item
        p-> m_value = value;
        p-> prev = nullptr;
        p-> next = head;
        head = p;
        m_size++;
        return pos;
    }
    else if (pos > 0 && pos < size() )
    {
        Node* p = head;
        for (int i = 0; i < pos; i++)
        {
            p = p->next;
        }

        Node* n = new Node; //creates the inserted
item
        n-> m_value = value;
        n-> prev = p;
        n-> next = p->next;
        m_size++;
        return pos;
    }
    else if (pos == size())
    {
        Node* p = new Node; //creates the inserted
item
        p-> m_value = value;
        p-> prev = tail;
        p-> next = nullptr;
        tail->next = p;
        tail = p;
        m_size++;
        return pos;
    }
    else
        return -1;
}

int Sequence::insert(const ItemType& value)
{
    int pos=0;
    Node* p = head;
    while( pos < size() )
    {
        if (p -> m_value <= value)
        {
            break;
        }
        pos++;
        p = p->next;
    }

    return insert(pos, value);
}

bool Sequence::erase(int pos)
{
    if ( pos == 0 && head -> next == nullptr)
    {
        head = nullptr;
        m_size--;
        return true;
    }
```

```
else if (pos == 0)
    {
        Node *killMe = head;
        head = killMe -> next;
        head -> prev = nullptr;
        delete killMe;
        m_size--;
        return true;
    }
    else if (pos >0 && pos < size()-1 )
    {
        Node *killMe = head;
        for (int i = 0; i < pos; ++i)
        {
            killMe = killMe -> next;
        }
        killMe->prev = killMe -> next;
        delete killMe;
        m_size--;
        return true;
    }
    else if (pos == size()-1)
    {
        Node *killMe = tail;
        tail = killMe -> prev;
        tail -> next = nullptr;
        delete killMe;
        m_size--;
        return true;
    }
    else
        return false;
}

int Sequence::remove(const ItemType& value)
{
    int count = 0;
    int pos=0;
    Node *p = head;
    while(p-> next != nullptr)
    {
        if (p->m_value == value)
        {
            erase(pos);
            pos--;
            count++;
        }
        pos++;
    }

    return count;
}
    //swap Names of Linked Lists
    Node* p = head;
    Node* n = tail;

    head = other.head;
    tail = other.tail;
    other.head = p;
    other.tail = n;

    //swap m_size;
    int temp_size = m_size;
    m_size = other.m_size;
    other.m_size = temp_size;
}
```

```
#ifndef Sequence_h
#define Sequence_h

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

using ItemType = unsigned long;
class Sequence
{
public:
    Sequence();    // Create an empty sequence (i.e., one with no items)
    static const int DEFAULT_MAX_ITEMS = 250;
   int insert(const ItemType& value);

    bool erase(int pos);

    int remove(const ItemType& value);
    bool get(int pos, ItemType& value) const;

    bool set(int pos, const ItemType& value);

    int find(const ItemType& value) const;
    void swap(Sequence& other);

private:
    int m_size;
    ItemType m_sequence[DEFAULT_MAX_ITEMS];
}; #endif


#include "newSequence.h"

//new Constructor
Sequence::Sequence(int max)
{
    if (max <0) {
        exit(1);
    }
    m_size =0;
    max_size=max;
    m_sequence = new ItemType[max_size];
}
//Destructor
Sequence::~Sequence()
{
    delete[] m_sequence;
}

//Copy Constructor
Sequence::Sequence(const Sequence& other)
{
    m_size= other.m_size;
    max_size= other.max_size;
    m_sequence= other.m_sequence;
    for (int i=0; i<m_size; i++) {
        m_sequence[i] = other.m_sequence[i];
    }
}

//Assignment Operator
Sequence& Sequence::operator=(const Sequence& rhs)
{
    if (this != &rhs) {
        Sequence temp(rhs);
        swap(temp);
    }
    return *this;
}
void Sequence::swap(Sequence& other)
{
    //swap m_sequence
    ItemType *temp_sequence = m_sequence;
    m_sequence = other.m_sequence;
    other.m_sequence = temp_sequence;

    //swap m_size;
    int temp_size = m_size;
    m_size = other.m_size;
    other.m_size = temp_size;

    //swap max_size
    int temp_max = max_size;
    max_size = other.max_size;
    other.max_size = temp_max;

}
```

```
#include "Sequence.h"

Sequence::Sequence(){
    m_size =0;
}

bool Sequence::empty()
{
    if (size()==0) {
        return true;
    }
    return false;
}

int Sequence::size() const
{
    return m_size;
}

int Sequence::insert(int pos, const ItemType& value)
{
    if (pos >= 0 && pos <=size() && size()<
DEFAULT_MAX_ITEMS) {
        for (int i=size(); i>pos; i--) {
            m_sequence[i] = m_sequence[i-1];
        }
        m_sequence[pos]=value;
        m_size++;
        return pos; //must check fixed-size array
    }
    else
        return -1;
}

int Sequence::insert(const ItemType& value)
{
    int p = size();

    for (int i=0; i<size(); i++) {
        if (m_sequence[i] >= value) {
            p=i;
            break;
        }
    }

    return insert(p, value);
}
```

```
    bool Sequence::erase(int pos)
    {
        if (pos >=0 && pos < size() ) {
            for (int i=pos; i< size()-1; i++) {
                m_sequence[i]=m_sequence[i+1];
            }
            m_size--;
            return true;
        }
        else
            return false;
    }

    int Sequence::remove(const ItemType& value)
    {
        int count = 0;
        for (int i=0; i<size(); i++) {
            if (m_sequence[i] == value) {
                erase(i);
                count++;
            }
        }
        return count;
    }

    bool Sequence::get(int pos, ItemType& value)
    const
    {
        if (pos >=0 && pos < size() ) {
            value = m_sequence[pos];
            return true;
        }
        else
            return false;
    }

    bool Sequence::set(int pos, const ItemType&
    value)
    {
        if (pos >=0 && pos < size() ) {
            m_sequence[pos] = value;
            return true;
        }
        else
            return false;
    }

    int Sequence::find(const ItemType& value) const
    {
     int p=-1;
    for (int i=0; i<size(); i++) {
        if (m_sequence[i] == value) {
            p = i;
            break;
        }
    }
    return p;
}

void Sequence::swap(Sequence& other)
{
    int temp_size = m_size;
    ItemType temp_array[DEFAULT_MAX_ITEMS];
    m_size = other.m_size;
    other.m_size = temp_size;

    for (int i = 0; i < DEFAULT_MAX_ITEMS; i++)
    {
        temp_array[i] = m_sequence[i];
        m_sequence[i] = other.m_sequence[i];
        other.m_sequence[i] = temp_array[i];
    }
}
```