#include<iostream>

#include <cassert>

using namespace std;

template <class T>

class DLinkedList

{

public:

    class Iterator; //forward declaration

    class Node;     //forward declaration

protected:

    Node \*head;

    Node \*tail;

    int count;

public:

    DLinkedList() : head(NULL), tail(NULL), count(0){};

    ~DLinkedList(){};

    void add(const T &e);

    void add(int index, const T &e);

    T removeAt(int index);

    bool removeItem(const T &item);

    bool empty();

    int size();

    void clear();

    T get(int index);

    void set(int index, const T &e);

    int indexOf(const T &item);

    bool contains(const T &item);

    string  toString();

    public:

    class Node

    {

    private:

        T data;

        Node \*next;

        Node \*previous;

        friend class DLinkedList<T>;

    public:

        Node()

        {

            this->previous = NULL;

            this->next = NULL;

        }

        Node(const T &data)

        {

            this->data = data;

            this->previous = NULL;

            this->next = NULL;

        }

    };

};

    template<class T>

string DLinkedList<T>::toString() {

    string s = "[";

    Node\* current = head;

    while (current) {

        s+= to\_string(current->data);

        if (current->next) {

            s+= ",";

        }

        current = current->next;

    }

   s+= "]";

    return s;

}

template <class T>

void DLinkedList<T>::add(const T& e) {

    /\* //\*Insert an element into the end of the list. \*/

//! nếu chưa có phần tử nào thì thêm vào đầu danh sách

    if(count == 0){

        tail = head = new Node(e);

    }

    else{

        Node\* tmp = new Node(e);

//! xử lý phần con trỏ để kết nói 2 chiều

        tail->next = tmp;

        tmp->previous = tail;

        tail = tmp;

    }

//! cộng số lượng danh sách

    count ++;

}

template<class T>

void DLinkedList<T>::add(int index, const T& e) {

//! kiểm tra đầu vào có hợp lệ hay không nếu không hợp lệ thì dừng chương trình

    if(index < 0 && index > count) throw out\_of\_range("");

//! nếu chưa có phần tử nào và thêm vào đầu danh sách

    else if(count == 0 || index == count ) add(e);

    else if(index == 0){

//! xử lý con trỏ

        Node\* tmp = new Node(e);

        tmp->next = head;

        head->previous = tmp;

        head = tmp;

        count  ++;

    }

    else{

//! tạo 2 node curr và prev lưu node ở vị trí chèn và node trước vị trí chèn

        Node\* curr = head->next,\* prev = head;

        index --;

        while(index){

            index--;

            prev = curr;

            curr= curr->next;

        }

//! thực hiện chèn node

        Node\* tmp = new Node(e);

        prev->next = tmp;

        tmp->previous = prev;

        tmp->next = curr;

        curr->previous = tmp;

        count ++;

    }

}

template<class T>

int DLinkedList<T>::size() {

    return count;

}

template<class T>

T DLinkedList<T>::get(int index) {

//! kiểm tra đầu vào có hợp lệ hay không

    if(index < 0 && index >= count) throw out\_of\_range("");

    Node\* tmp = head;

    while(index){

        tmp = tmp->next;

        index --;

    }

    return tmp->data;

}

template <class T>

void DLinkedList<T>::set(int index, const T& e) {

//! kiểm tra đầu vào có hợp lệ hay không

    if(index < 0 && index >= count) throw out\_of\_range("");

    Node\* tmp = head;

    while(index){

        tmp = tmp->next;

        index --;

    }

    tmp->data = e;

}

template<class T>

bool DLinkedList<T>::empty() {

    return count == 0;

}

template<class T>

int DLinkedList<T>::indexOf(const T& item) {

    /\* //\*Return the first index wheter item appears in list, otherwise return -1 \*/

    Node\* tmp = head;

    int index = 0;

    while(tmp){

        if(tmp->data == item) return index;

        tmp = tmp->next;

        index ++;

    }

    return -1;

}

template<class T>

bool DLinkedList<T>::contains(const T& item) {

    /\* //\*Check if item appears in the list \*/

    Node\* tmp = head;

    while(tmp){

        if(tmp->data == item) return true;

        tmp = tmp->next;

    }

    return false;

}

template <class T>

T DLinkedList<T>::removeAt(int index)

{

    T result;

//! kiểm tra đầu vào

    if(index < 0 && index >= count) throw out\_of\_range("");

//! nếu chỉ có 1 phần tử

    if(count == 1){

        result = head->data;

        delete head;

        head = tail = NULL;

    }

//! nhiều hơn 1 phần tử và ở vị trí đầu tiên

    else if(index == 0){

        Node\* tmp = head;

        head = head->next;

        result = tmp->data;

        delete tmp;

        head->previous = NULL;

    }

    else{

        Node\* tmp = head;

//! tìm vị trí trước vị trí cần xóa

        index --;

        while(index){

            tmp = tmp->next;

            index --;

        }

//! lưu vị giá trị xóa đi

        result = tmp->next->data;

//! nếu nó là vị trí cuối cùng

        if(tmp->next == tail){

            delete tail;

            tail = tmp;

            tail->next = NULL;

        }

        else{

            Node\* h = tmp->next;

            tmp->next = tmp->next->next;

            tmp->next->previous = tmp;

            delete h;

        }

    }

    count --;

    return result;

}

template <class T>

bool DLinkedList<T>::removeItem(const T& item)

{

    int index = indexOf(item);

    if(index == -1) return false;

    removeAt(index);

    return true;

}

template<class T>

void DLinkedList<T>::clear(){

    Node\* tmp = head;

    while(head){

        head = head->next;

        delete tmp;

        tmp = head;

    }

    head = tail = NULL;

    count = 0;

}

template<class T>

class Stack {

protected:

    DLinkedList<T> list;

    public:

    Stack() {}

    void push(T item) ;

    T pop() ;

    T top() ;

    bool empty() ;

    int size() ;

    void clear() ;

};

template<class T>

    void  Stack<T>::push(T item) {

        list.add(0,item);

    }

template<class T>

    T  Stack<T>::pop() {

        // TODO: Remove an element on top of the stack

        T tmp = list.get(0);

        list.removeAt(0);

        return tmp;

    }

template<class T>

    T   Stack<T>::top() {

        // TODO: Get value of the element on top of the stack

        return list.get(0);

    }

template<class T>

    bool   Stack<T>::empty() {

        // TODO: Determine if the stack is empty

        return list.empty();

    }

template<class T>

    int Stack<T>::size() {

        // TODO: Get the size of the stack

        return list.size();

    }

template<class T>

    void   Stack<T>::clear() {

        // TODO: Clear all elements of the stack

        list.clear();

    }

int main(){

  Stack<int> stack;

    int item[] = { 3, 1, 4, 5, 2, 8, 10, 12 };

    for (int idx = 0; idx < 8; idx++) stack.push(item[idx]);

    assert(stack.top() == 12);

    stack.pop();

    stack.pop();

    cout << stack.top();

}

    #include<iostream>

#include <cassert>

using namespace std;

template <class T>

class DLinkedList

{

public:

    class Iterator; //forward declaration

    class Node;     //forward declaration

protected:

    Node \*head;

    Node \*tail;

    int count;

public:

    DLinkedList() : head(NULL), tail(NULL), count(0){};

    ~DLinkedList(){};

    void add(const T &e);

    void add(int index, const T &e);

    T removeAt(int index);

    bool removeItem(const T &item);

    bool empty();

    int size();

    void clear();

    T get(int index);

    void set(int index, const T &e);

    int indexOf(const T &item);

    bool contains(const T &item);

    string  toString();

    public:

    class Node

    {

    private:

        T data;

        Node \*next;

        Node \*previous;

        friend class DLinkedList<T>;

    public:

        Node()

        {

            this->previous = NULL;

            this->next = NULL;

        }

        Node(const T &data)

        {

            this->data = data;

            this->previous = NULL;

            this->next = NULL;

        }

    };

};

    template<class T>

string DLinkedList<T>::toString() {

    string s = "[";

    Node\* current = head;

    while (current) {

        s+= to\_string(current->data);

        if (current->next) {

            s+= ",";

        }

        current = current->next;

    }

   s+= "]";

    return s;

}

template <class T>

void DLinkedList<T>::add(const T& e) {

    /\* //\*Insert an element into the end of the list. \*/

//! nếu chưa có phần tử nào thì thêm vào đầu danh sách

    if(count == 0){

        tail = head = new Node(e);

    }

    else{

        Node\* tmp = new Node(e);

//! xử lý phần con trỏ để kết nói 2 chiều

        tail->next = tmp;

        tmp->previous = tail;

        tail = tmp;

    }

//! cộng số lượng danh sách

    count ++;

}

template<class T>

void DLinkedList<T>::add(int index, const T& e) {

//! kiểm tra đầu vào có hợp lệ hay không nếu không hợp lệ thì dừng chương trình

    if(index < 0 && index > count) throw out\_of\_range("");

//! nếu chưa có phần tử nào và thêm vào đầu danh sách

    else if(count == 0 || index == count ) add(e);

    else if(index == 0){

//! xử lý con trỏ

        Node\* tmp = new Node(e);

        tmp->next = head;

        head->previous = tmp;

        head = tmp;

        count  ++;

    }

    else{

//! tạo 2 node curr và prev lưu node ở vị trí chèn và node trước vị trí chèn

        Node\* curr = head->next,\* prev = head;

        index --;

        while(index){

            index--;

            prev = curr;

            curr= curr->next;

        }

//! thực hiện chèn node

        Node\* tmp = new Node(e);

        prev->next = tmp;

        tmp->previous = prev;

        tmp->next = curr;

        curr->previous = tmp;

        count ++;

    }

}

template<class T>

int DLinkedList<T>::size() {

    return count;

}

template<class T>

T DLinkedList<T>::get(int index) {

//! kiểm tra đầu vào có hợp lệ hay không

    if(index < 0 && index >= count) throw out\_of\_range("");

    Node\* tmp = head;

    while(index){

        tmp = tmp->next;

        index --;

    }

    return tmp->data;

}

template <class T>

void DLinkedList<T>::set(int index, const T& e) {

//! kiểm tra đầu vào có hợp lệ hay không

    if(index < 0 && index >= count) throw out\_of\_range("");

    Node\* tmp = head;

    while(index){

        tmp = tmp->next;

        index --;

    }

    tmp->data = e;

}

template<class T>

bool DLinkedList<T>::empty() {

    return count == 0;

}

template<class T>

int DLinkedList<T>::indexOf(const T& item) {

    /\* //\*Return the first index wheter item appears in list, otherwise return -1 \*/

    Node\* tmp = head;

    int index = 0;

    while(tmp){

        if(tmp->data == item) return index;

        tmp = tmp->next;

        index ++;

    }

    return -1;

}

template<class T>

bool DLinkedList<T>::contains(const T& item) {

    /\* //\*Check if item appears in the list \*/

    Node\* tmp = head;

    while(tmp){

        if(tmp->data == item) return true;

        tmp = tmp->next;

    }

    return false;

}

template <class T>

T DLinkedList<T>::removeAt(int index)

{

    T result;

//! kiểm tra đầu vào

    if(index < 0 && index >= count) throw out\_of\_range("");

//! nếu chỉ có 1 phần tử

    if(count == 1){

        result = head->data;

        delete head;

        head = tail = NULL;

    }

//! nhiều hơn 1 phần tử và ở vị trí đầu tiên

    else if(index == 0){

        Node\* tmp = head;

        head = head->next;

        result = tmp->data;

        delete tmp;

        head->previous = NULL;

    }

    else{

        Node\* tmp = head;

//! tìm vị trí trước vị trí cần xóa

        index --;

        while(index){

            tmp = tmp->next;

            index --;

        }

//! lưu vị giá trị xóa đi

        result = tmp->next->data;

//! nếu nó là vị trí cuối cùng

        if(tmp->next == tail){

            delete tail;

            tail = tmp;

            tail->next = NULL;

        }

        else{

            Node\* h = tmp->next;

            tmp->next = tmp->next->next;

            tmp->next->previous = tmp;

            delete h;

        }

    }

    count --;

    return result;

}

template <class T>

bool DLinkedList<T>::removeItem(const T& item)

{

    int index = indexOf(item);

    if(index == -1) return false;

    removeAt(index);

    return true;

}

template<class T>

void DLinkedList<T>::clear(){

    Node\* tmp = head;

    while(head){

        head = head->next;

        delete tmp;

        tmp = head;

    }

    head = tail = NULL;

    count = 0;

}

template<class T>

class Queue {

protected:

    DLinkedList<T> list;

    public:

    Queue() {}

    void push(T item) ;

    T pop() ;

    T top() ;

    bool empty() ;

    int size() ;

    void clear() ;

};

template<class T>

    void Queue<T> :: push(T item) {

        list.add(item);

    }

template<class T>

    T  Queue<T> :: pop() {

        // TODO: Remove an element on top of the stack

        T tmp = list.get(0);

        list.removeAt(0);

        return tmp;

    }

template<class T>

    T  Queue<T> :: top() {

        // TODO: Get value of the element on top of the stack

        return list.get(0);

    }

template<class T>

    bool  Queue<T> :: empty() {

        // TODO: Determine if the stack is empty

        return list.empty();

    }

template<class T>

    int  Queue<T> :: size() {

        // TODO: Get the size of the stack

        return list.size();

    }

template<class T>

    void  Queue<T> ::clear() {

        // TODO: Clear all elements of the stack

        list.clear();

    }

int main(){

    Queue<int> queue;

    assert(queue.empty());

    assert(queue.size() == 0);

}

#include <iostream>

#include <queue>

#include <stack>

using namespace std;

//! Hàm đảo ngược một queue

void reverseQueue(queue<int> & q) {

//! khai báo stack

    stack<int> s;

    while(!q.empty()){

        s.push(q.front());

        q.pop();

    }

    while(!s.empty()){

        q.push(s.top());

        s.pop();

    }

}

int main(){

    // Khởi tạo một queue

  queue<int> q;

  // Thêm một số phần tử vào queue

  q.push(1);

  q.push(2);

  q.push(3);

  q.push(4);

  q.push(5);

  //! In ra các phần tử của queue ban đầu

 queue<int> q1 = q ;

 int a = q1.size();

   for (int i = 0; i < a; i++) {

    cout << q1.front() << " ";

    q1.pop();

  }

  cout << endl;

  //! Đảo ngược queue

  reverseQueue(q);

  //! In ra các phần tử của queue sau khi đảo ngược

  for (int i = 0; i < a; i++) {

    cout << q.front() << " ";

    q.pop();

  }

  cout << endl;

  return 0;

}

#include <iostream>

#include <queue>

#include <stack>

//! giống câu 3

void reverseStack(stack<int> & st) {

    queue<int> q;

    while(!st.empty()){

        q.push(st.top());

        st.pop();

    }

    while(!q.empty()){

        st.push(q.front());

        q.pop();

    }

}

string deleteDuplicate(string s, int k) {

int i = 0, n = s.length();

int count[n] = {0};

for (int j = 0; j < n; ++j, ++i) {

s[i] = s[j];

count[i] = i > 0 && s[i - 1] == s[j] ? count[i - 1] + 1 : 1;

if (count[i] == k) i -= k;

}

return s.substr(0, i);

}

int sumOfMinSubarray(vector<int>& A) {

// STUDENT ANSWER

if (A.size() == 0)

return 0;

int n = A.size();

int MOD = 1e4 ;

vector<int> left(n), right(n);

// for every i find the Next smaller element to left and right

// Left

stack<int>st;

st.push(0);

left[0] = 1; // distance = 1, left not found, this is distance multiplied with num, so it can't be zero

for(int i=1; i<n; i++)

{

while(!st.empty() && A[i] < A[st.top()])

st.pop();

if(st.empty())

left[i] = i+1; // total distance if less element not found = i+1

else

left[i] = i - st.top(); // distance = i-st.top()

st.push(i);

}

while(st.size())

st.pop();

// Right

st.push(n-1);

right[n-1] = 1; // distance = 1, right not found, this is distance multiplied with num, so it can't be zero

for(int i=n-2; i>=0; i--)

{

while(!st.empty() && A[i] <= A[st.top()])

st.pop();

if(st.empty())

right[i] = n-i; // distance

else

right[i] = st.top()-i;

st.push(i);

}

// total number of subarrays : (Left[i] \* Right[i])

// total contribution in A[i] element in final answer : (Left \* Right) \* A[i]

// for(int i=0; i<n; i++)

// cout << left[i] << " : " << right[i] << endl;

// for each i, contribution is (Left \* Right) \* Element

int res = 0;

for(int i=0; i<n; i++)

{

long long prod = (left[i]\*right[i])%MOD;

prod = (prod\*A[i])%MOD;

res = (res + prod)%MOD;

}

return res%MOD;

}