#include<iostream>

using namespace std;

template <class T>

class DLinkedList {

public:

    class Node; // Forward declaration

protected:

    Node\* head;

    Node\* tail;

    int count;

public:

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

    ~DLinkedList(){};

    void    add(const T &e);

    void    add(int index, const T &e);

    int     size();

    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;

}

int main(){

    DLinkedList<int> list;

    int size = 10;

    for(int idx=0; idx < size; idx++){

    list.add(idx);

    }

    cout << list.toString()<<endl;;

//! [0,1,2,3,4,5,6,7,8,9]

    DLinkedList<int> list1;

    int size1 = 10;

    for(int idx=0; idx < size; idx++){

    list1.add(0, idx);

    }

    cout << list1.toString();

 //!   [9,8,7,6,5,4,3,2,1,0]

}

#include<iostream>

using namespace std;

template <class T>

class DLinkedList {

public:

    class Node; // Forward declaration

protected:

    Node\* head;

    Node\* tail;

    int count;

public:

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

    ~DLinkedList(){};

    void    add(const T &e);

    void    add(int index, const T &e);

    int     size();

    string  toString();

    bool    empty();

    T       get(int index);

    void    set(int index, const T &e);

    int     indexOf(const T &item);

    bool    contains(const T &item);

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;

}

int main(){

    DLinkedList<int> list;

    int size = 10;

    for(int idx=0; idx < size; idx++){

    list.add(idx);

    }

    for(int idx=0; idx < size; idx++){

    cout << list.get(idx) << " |";

    }

//! 0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |

    DLinkedList<int> list1;

    int size1 = 10;

    int value1[] = {2,5,6,3,67,332,43,1,0,9};

    for(int idx=0; idx < size1; idx++){

    list1.add(idx);

    }

    for(int idx=0; idx < size1; idx++){

    list1.set(idx, value1[idx]);

    }

    cout << list1.toString();

//! [0,1,2,3,4,5,6,7,8,9]

}

#include<iostream>

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();

    Iterator begin()

    {

        return Iterator(this, true);

    }

    Iterator end()

    {

        return Iterator(this, false);

    }

public:

    class Node

    {

    private:

        T data;

        Node \*next;

        friend class DLinkedList<T>;

    public:

        Node()

        {

            next = 0;

        }

        Node(Node \*next)

        {

            this->next = next;

        }

        Node(T data, Node \*next = NULL)

        {

            this->data = data;

            this->next = next;

        }

    };

    class Iterator

    {

    private:

        DLinkedList<T> \*pList;

        Node \*current;

        int index; // is the index of current in pList

    public:

        Iterator(DLinkedList<T> \*pList, bool begin);

        Iterator &operator=(const Iterator &iterator);

        void set(const T &e);

        T &operator\*();

        bool operator!=(const Iterator &iterator);

        void remove();

        // Prefix ++ overload

        Iterator &operator++();

        // Postfix ++ overload

        Iterator operator++(int);

    };

};

   template <class T>

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

    if(count == 0){

        head = tail = new Node(e);

    }

    else{

        Node\* tmp = new Node(e);

        tail->next = tmp;

        tail = tmp;

    }

    count++;

}

//! hiện thực các hàm như ở câu 1

template<class T>

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

    if(index < 0 || index > count) return;

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

    else if(index == 0){

        Node\* tmp = new Node(e);

        tmp->next = head;

        head = tmp;

        count ++;

    }

    else{

        index --;

        Node\* tmp = head;

        Node\* New = new Node(e);

        while(index){

            tmp = tmp->next;

            index--;

        }

        New->next = tmp->next;

        tmp->next = New;

        count ++;

    }

}

template<class T>

int DLinkedList<T>::size() {

   return count;

}

template<class T>

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

//! ném ra ngoại lệ

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

    Node\* tmp = head;

    while(index){

        index --;

        tmp = tmp->next;

    }

    return tmp->data;

}

template <class T>

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

//! ném ra ngoại lệ

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

    Node\* tmp = head;

    while(index){

        index --;

        tmp = tmp->next;

    }

    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 \*//

    int i = 0;

    Node\* tmp = head;

    while(tmp){

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

        i ++;

        tmp = tmp->next;

    }

    return -1;

}

template<class T>

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

    Node\* tmp = head;

//! duyệt vòng while

    while(tmp){

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

        tmp = tmp->next;

    }

    return false;

}

//! hiện thực hàm toString

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>

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

{

    T result;

//! nếu đúng thì chương trình chạy sai thì dừng chương trình

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

//! nếu chỉ có 1 phần tử đầu tiên thì reset head = tail = nullptr

    else if(count == 1){

        result = head->data;

        delete head;

        head =tail = nullptr;

    }

//! nếu xóa ở vị trí 0 thì

    else if(index == 0){

//! lưu data của node đầu tiên lại

        result = head->data;

        Node\* tmp = head;

        head = head->next;

//! xóa vùng nhớ

        delete tmp;

    }

    else{

//! giảm index xuống 1 đơn vị

        index --;

        Node\* tmp = head;

//! dùng vòng lặp đến vị trí cần tìm

        while(index){

            tmp = tmp->next;

            index --;

        }

        if(tmp->next->next){

            Node\* d = tmp->next;

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

            result = d->data;

            delete d;

        }

        else{

//! nếu không thỏa if thì phải lưu tail lại vì phần tử đang xóa là phần tử ở vị trí cuối cùng

            tail = tmp;

            Node\* d = tmp->next;

            tmp->next = nullptr;

            result = d->data;

            delete d;

        }

    }

    count --;

    return result;

}

//!

template <class T>

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

{

//! xóa node có giá trị bằng phần tử ở vị trí đầu tiên của danh sách nếu có phần tử thì xóa rồi trả về true không thì ngược lại

   int index = indexOf(item);

   if(index == -1 ) return false;

   removeAt(index);

   return true;

}

//! giải phóng các node

template<class T>

void DLinkedList<T>::clear(){

    Node\* tmp = head;

    while(tmp){

        tmp = tmp->next;

        delete head;

        head = tmp;

    }

    tail = head = NULL;

    count = 0;

}

template <class T>

DLinkedList<T>::Iterator::Iterator(DLinkedList<T>\* pList , bool begin)

{

    /\*

        Constructor of iterator

        \* Set pList to pList

        \* begin = true:

        \* \* Set current (index = 0) to pList's head if pList is not NULL

        \* \* Otherwise set to NULL (index = -1)

        \* begin = false:

        \* \* Always set current to NULL

        \* \* Set index to pList's size if pList is not NULL, otherwise 0

    \*/

   this->pList = pList;

   if(pList == NULL){index = -1;}

   else if(begin){

        if(this->pList->size() == 0){this->current = NULL;index = -1;}

        else{ this->current = pList->head;index = 0;}

   }

   else{

        this->current = NULL;

        if(this->pList->size() == 0) index = this->pList->size();

        else index = pList->size();

   }

}

template <class T>

typename DLinkedList<T>::Iterator& DLinkedList<T>::Iterator::operator=(const Iterator& iterator)

{

    /\*

        Assignment operator

        \* Set this current, index, pList to iterator corresponding elements.

    \*/

    // cout << this->index;

    this->pList = iterator.pList;

    this->current = iterator.current;

    this->index = iterator.index;

    return \*this;

}

template <class T>

void DLinkedList<T>::Iterator::remove()

{

    /\*

        Remove a node which is pointed by current

        \* After remove current points to the previous node of this position (or node with index - 1)

        \* If remove at front, current points to previous "node" of head (current = NULL, index = -1)

        \* Exception: throw std::out\_of\_range("Segmentation fault!") if remove when current is NULL

    \*/

   if(this->current == NULL) throw out\_of\_range("Segmentation fault!");

    int index = this->pList->indexOf(this->current->data);

    if(index == 0){

        this->pList->removeAt(index);

        this->index = -1;

        current = NULL;

    }

    else{

        T e = this->pList->removeAt(index-1);

        this->index = index - 1;

        current->data = e;

}

}

template <class T>

void DLinkedList<T>::Iterator::set(const T& e)

{

    /\*

        Set the new value for current node

        \* Exception: throw std::out\_of\_range("Segmentation fault!") if current is NULL

    \*/

    if(current == NULL) throw out\_of\_range("Segmentation fault!");

    current->data = e;

}

template <class T>

T& DLinkedList<T>::Iterator::operator\*()

{

    /\*

        Get data stored in current node

        \* Exception: throw std::out\_of\_range("Segmentation fault!") if current is NULL

    \*/

   if(current == NULL) throw out\_of\_range("Segmentation fault!");

   //cout << current->data;

   return current->data;

}

template <class T>

bool DLinkedList<T>::Iterator::operator!=(const Iterator& iterator)

{

    /\*

        Operator not equals

        \* Returns false if two iterators points the same node and index

    \*/

   return !(iterator.index == this->index || iterator.current == this->current);

}

// Prefix ++ overload

template <class T>

typename DLinkedList<T>::Iterator& DLinkedList<T>::Iterator::operator++()

{

    /\*

        Prefix ++ overload

        \* Set current to the next node

        \* If iterator corresponds to the previous "node" of head, set it to head

        \* Exception: throw std::out\_of\_range("Segmentation fault!") if iterator corresponds to the end

    \*/

    if(current == NULL){

        current = pList->head;

        index = 0;

    }

    else{

        current = current->next;

        index++;

    }

    return \*this;

}

// Postfix ++ overload

template <class T>

typename DLinkedList<T>::Iterator DLinkedList<T>::Iterator::operator++(int)

{

    /\*

        Postfix ++ overload

        \* Set current to the next node

        \* If iterator corresponds to the previous "node" of head, set it to head

        \* Exception: throw std::out\_of\_range("Segmentation fault!") if iterator corresponds to the end

    \*/

    DLinkedList<T>::Iterator t = \*this;

    if(current == NULL){

        current = pList->head;

        index = 0;

    }

    else{

        current = current->next;

        index++;

    }

    return t;

}

int main(){

    DLinkedList<int> list;

    int size = 10;

    for(int idx=0; idx < size; idx++){

        list.add(idx);

    }

    DLinkedList<int>::Iterator it = list.begin();

    for(; it != list.end(); it++)

    {

        cout << \*it << " |";

    }

    //! 0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |

    DLinkedList<int> list1;

    int size1 = 10;

    for (int idx = 0; idx < size1; idx++)

    {

        list1.add(idx);

    }

    DLinkedList<int>::Iterator it1 = list1.begin();

    for(; it1 != list1.end(); it1++)

    {

        it1.remove();

    }

    cout<<endl;

    cout << list1.toString();

//![]

}

#include<iostream>

using namespace std;

template <class T>

class DLinkedList {

public:

    class Node; // Forward declaration

protected:

    Node\* head;

    Node\* tail;

    int count;

public:

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

    ~DLinkedList(){};

    void    add(const T &e);

    void    add(int index, const T &e);

    int     size();

    string  toString();

    bool    empty();

    T       get(int index);

    void    set(int index, const T &e);

    int     indexOf(const T &item);

    bool    contains(const T &item);

    T       removeAt(int index);

    bool    removeItem(const T &item);

    void    clear();

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;

}

int main(){

    DLinkedList<int> list;

    int size = 10;

    int value[] = {2,5,6,3,67,332,43,1,0,9};

    for(int idx=0; idx < size; idx++){

    list.add(value[idx]);

    }

    list.removeAt(0);

    cout << list.toString();

//! [5,6,3,67,332,43,1,0,9]

}

#include<iostream>

#include <list>

using namespace std;

class DataLog

{

private:

    list<int> logList;

    list<int>::iterator currentState;

public:

    DataLog();

    DataLog(const int &data);

    void addCurrentState(int number);

    void subtractCurrentState(int number);

    void save();

    void undo();

    void redo();

    int getCurrentStateData()

    {

        return \*currentState;

    }

    void printLog()

    {

        for (auto i = logList.begin(); i != logList.end(); i++) {

            if(i == currentState) cout << "Current state: ";

            cout << "[ " << \*i << " ] => ";

        }

        cout << "END\_LOG";

    }

};

DataLog::DataLog()

{

    /\*

     \* TODO:  add the first state with 0

     \*/

    logList.push\_front(0);

    currentState = logList.begin();

}

DataLog::DataLog(const int &data)

{

    /\*

     \* TODO:  add the first state with data

     \*/

    logList.push\_front(data);

    currentState = logList.begin();

}

void DataLog::addCurrentState(int number)

{

    \*currentState += number;

}

void DataLog::subtractCurrentState(int number)

{

    /\*

     \* TODO: Decrease the value of current state by number

     \*/

    \*currentState -= number;

}

void DataLog::save()

{

    /\*

     \* TODO: This function will create a new state, copy the data of the currentState

     \*       and move the currentState Iterator to this new state. If there are other states behind the

     \*       currentState Iterator, we delete them all before creating a new state.

     \*/

 //! Di chuyển con trỏ currentState lên một bước để trỏ tới phần tử tiếp theo trong danh sách.

     currentState++;

//! Kiểm tra nếu currentState không trỏ tới cuối danh sách.

     if(currentState != logList.end()){

         list<int>::iterator curr = currentState;

         list<int>::iterator prev = currentState;

        while(curr != logList.end()){

            curr++;

            logList.erase(prev);

            prev = curr;

        }

        currentState = logList.end();

    }

    currentState--;

    logList.push\_back(\*currentState);

    currentState++;

     for (auto i = logList.begin(); i != logList.end(); i++) {

            cout << "[ " << \*i << " ] => ";

        }

        cout<<endl;

}

void DataLog::undo()

{

    /\*

     \* TODO: Switch to the previous state of the data

     \*       If this is the oldest state in the log, nothing changes

     \*/

    if(currentState != logList.begin())

        --currentState;

}

void DataLog::redo()

{

    /\*

     \* TODO: Switch to the latter state of the data

     \*       If this is the latest state in the log, nothing changes

     \*/

     ++currentState;

    if(currentState == logList.end()) --currentState;

}

int main(){

    DataLog log(10);

    log.save();

    log.addCurrentState(15);

    log.save();

    log.addCurrentState(15);

    log.undo();

    log.printLog();

//! [ 10 ] => Current state: [ 25 ] => [ 40 ] => END\_LOG

cout<<endl;

    DataLog log1(10);

    log1.save();

    log1.addCurrentState(15);

    log1.save();

    log1.addCurrentState(15);

    log1.save();

    log1.subtractCurrentState(5);

    log1.printLog();

// //! [ 10 ] => [ 25 ] => [ 40 ] => Current state: [ 35 ] => END\_LOG

}

#include<iostream>

#include <unordered\_map>

using namespace std;

struct ListNode {

    int val;

    ListNode \*left;

    ListNode \*right;

    ListNode(int x = 0, ListNode \*l = nullptr, ListNode\* r = nullptr) : val(x), left(l), right(r) {}

};

ListNode\* reverse(ListNode\* head, int a, int b) {

//! Khai báo một bảng băm (unordered\_map) để lưu trữ con trỏ tới các ListNode với khóa là chỉ số.

    unordered\_map<int ,ListNode\* > v;

//! Khởi tạo một con trỏ tạm thời để duyệt qua danh sách.

    ListNode\* tmp = head;

//! Khởi tạo biến index để theo dõi chỉ số của các nút trong danh sách.

    int index = 1;

//! Khởi tạo biến i với giá trị b, sẽ được sử dụng để đảo ngược thứ tự của các nút từ b đến a.

    int i = b;

    while(tmp){

        if(index >= a && index <= b){

 //! Lưu trữ con trỏ tới các nút trong khoảng a đến b vào bảng băm v.

            v[i] = tmp;

            i--;

        }

//! Lưu trữ con trỏ tới các nút nằm ngoài khoảng a đến b.

        else v[index] = tmp;

        tmp = tmp->right;

        index ++;

    }

//! Gán con trỏ head bằng con trỏ tới nút đầu tiên sau khi đảo ngược.

    head = v[1];

    head->left = NULL;

    tmp = head;

    for(int i = 2; i < index; i++){

//! thực hiện các thao tác liên kết lại

        tmp->right = v[i];

        v[i]->left = tmp;

        tmp = v[i];

    }

    tmp->right = NULL;

    return head;

}

ListNode\* init(int\* list, int size, unordered\_map<ListNode\*, int>& nodeValue) {

    if (size == 0) {

        return nullptr;

    }

    ListNode\* head = new ListNode(list[0]);

    nodeValue[head] = list[0];

    ListNode\* prev = head;

    for (int i = 1; i < size; i++) {

        ListNode\* newNode = new ListNode(list[i]);

        nodeValue[newNode] = list[i];

        prev->right = newNode;

        newNode->left = prev;

        prev = newNode;

    }

    return head;

}

void printList(ListNode\* head, unordered\_map<ListNode\*, int>& nodeValue) {

    ListNode\* tmp = head;

    while (tmp) {

        cout << tmp->val << " ";

        tmp = tmp->right;

    }

    cout << endl;

}

void freeMem(ListNode\* head) {

    while (head) {

        ListNode\* tmp = head;

        head = head->right;

        delete tmp;

    }

}

int main(){

    int size;

    cin >> size;

    int\* list = new int[size];

    for(int i = 0; i < size; i++) {

        cin >> list[i];

    }

    int a, b;

    cin >> a >> b;

    unordered\_map<ListNode\*, int> nodeValue;

    ListNode\* head = init(list, size, nodeValue);

    ListNode\* reversed = reverse(head, a, b);

    try {

        printList(reversed, nodeValue);

    }

    catch(char const\* err) {

        cout << err << '\n';

    }

    freeMem(head);

    delete[] list;

}

ListNode\* reverse(ListNode\* head, int a, int b) {

ListNode\* stackHead = NULL, \* stackTail = NULL, \*ptr = head;

for(int i = 1; i < a; i++){

ptr = ptr->right;

}

ListNode \*front = ptr->left;

stackTail = ptr;

for(int i = a; i <= b; i++){

ListNode \*tmp = ptr;

ptr = ptr->right;

tmp->left = NULL;

tmp->right = stackHead;

stackHead = tmp;

stackHead->left = ptr;

}

if(front){

front->right = stackHead;

stackHead->left = front;

stackTail->right = ptr;

if(ptr) ptr->left = stackTail;

}

else{

stackTail->right = ptr;

if(ptr) ptr->left = stackTail;

head = stackHead;

stackHead->left = NULL;

}

return head;

}