quy định

điểm TB trên 4

điểm lab trên 4

điểm lab =

% lab trên BKel

% bài test (random trong 5 lab + 1 số câu vận dụng)

# REVISION – recursion & arraylist

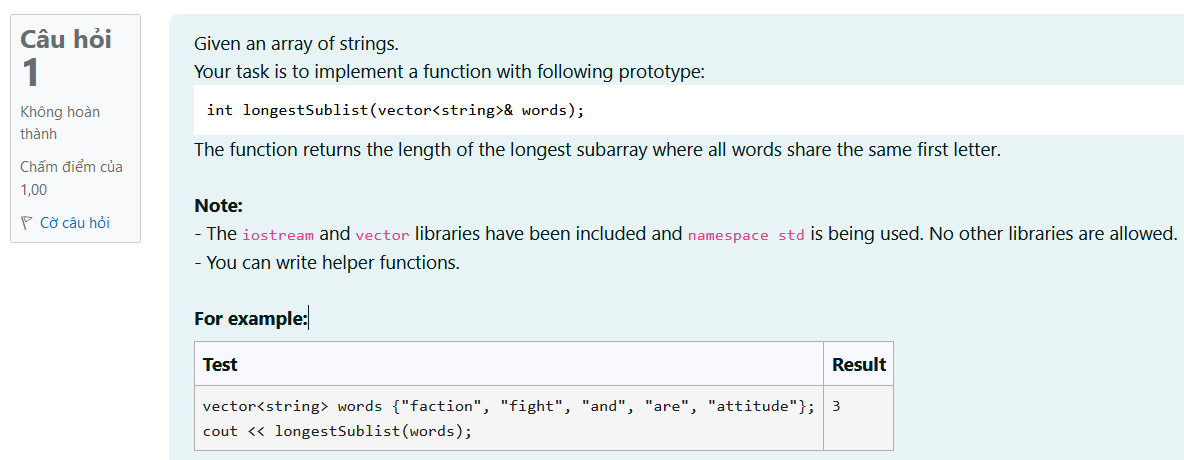
array vs arraylist

-cấp phát tĩnh - where: stack - bộ nhớ tĩnh

-cấp phát động - where: heap (big storage) - bộ nhớ động

arrayList => vector STL

## cau hoi 1



int longestSublist(vector<string>& words) {

// STUDENT ANSWER

}

int longestSublist(vector<*string*>& *words*) {

    // STUDENT ANSWER

    int length[*words*.size()]={1}; // all elements 0

    int k=0;

    for(int i = 0; i < *words*.size()-1; i++){

        if(*words*[i][0] == *words*[i+1][0]){

            length[k]++;

        }

        else{

            k++;

            length[k] = 1;

            continue;

        }

    }

    int result = 1;

    for(int i = 0; i < *words*.size(); i++){

        if(length[i]>= result){

            result = length[i];

        }

    }

    return result;

}

## cau hoi 2

Give a positive integer x, implement recursive function

void printHailstone(int number){}

to print the Hailstone Sequence of a given number upto 1 (no space at the end).

Hailstone Sequences follow these rules:

* If a number is even, divide it by 2
* If a number is odd, multiply it by 3 and add 1.

Example:

If number = 5. 5 is odd number so next number is 5\*3 + 1 = 16. 16 is even number so next number is 16/2 = 8...  
 Finally, we get Hailstone sequence: 5 16 8 4 2 1.

You can find more information at: <https://diendantoanhoc.net/topic/89145-d%C3%A3y-s%E1%BB%91-hailstone/>

**Note:** Please note that you can't using key work for, while, goto (even in variable names, comment).

You can implement other recursive functions if needed.

For this exercise, we have #include <iostream> and using namespace std;

For example:

| **Test** | **Result** |
| --- | --- |
| printHailstone(32); | 32 16 8 4 2 1 |

void printHailstone(int number)

{

/\*

\* STUDENT ANSWER

\*/

}

void printHailstone2(int *number*){

    int result;

    if(*number*%2 == 0){

        result = *number* /2;

    }

    else{

        result = *number*\*3 +1;

    }

    //print

    if(result !=1){

        cout<<result<<' ';

    }

    else{

        cout<<result;

        return;

    }

    return printHailstone2(result);

}

void printHailstone(int *number*)

{

    /\*

     \* STUDENT ANSWER

     \*/

    if(*number* != 1){

        cout<<*number*<<' ';

        return printHailstone2(*number*);

    }

    else{

        cout<<*number*;

    }

}

Version mới

void printHailstone(int number)

{

if(number == 1){

cout<<"1";

return;

}

cout<<number<<' ';

if(number%2 == 0){

printHailstone(number/2);

}

else{

printHailstone(3\*number+1);

}

}

## Cau hoi 3 -redo

In this exercise, you can use implemented functions in *previous question*(if needed) and implement these following functions.

bool containsPoint(const Point point){}

bool containsTriangle(const Point pointA, const Point pointB, const Point pointC){}

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| Point pointO(0, 2);  Point point1(1, 2);  Circle A = Circle(pointO, 2);  cout << A.containsPoint(point1); | 1 |
| Point pointO(0, 0);  Point point1(1, 0), point2(-1, 0), point3(0, 3);  Circle A = Circle(pointO, 3);  cout << A.containsTriangle(point1, point2, point3); | 0 |

class Point

{

/\*

\* STUDENT ANSWER

\* TODO: using code template in previous question

\*/

};

class Circle

{

private:

Point center;

double radius;

public:

Circle()

{

/\*

\* STUDENT ANSWER

\*/

}

Circle(Point center, double radius)

{

/\*

\* STUDENT ANSWER

\*/

}

bool containsPoint(const Point point)

{

/\*

\* STUDENT ANSWER

\* TODO: check if a given point is entirely within the circle (does not count if the point lies on the circle).

If contain, return true.

\*/

}

bool containsTriangle(const Point pointA, const Point pointB, const Point pointC)

{

/\*

\* STUDENT ANSWER

\* TODO: check if a given triangle ABC (A, B, C are not on the same line) is entirely within the circle (does not count if any point of the triangle lies on the circle).

If contain, return true.

\*/

}

};

## Feedback:

Kĩ năng đọc tài liệu: hàm xuất gì, xài sao

Documentation c++, các ví dụ minh hoạ

Char \*: là mảng các kí tự

Kí tự kết thúc ‘\0’

String:

Không dùng ‘\0’

//segmentation fail

struct *Node*{

    int data;

};

*Node* \*node;

node->data =5;//chua cap phat vung nho cho int data

//hoac cac du lieu dau/ cuoi array

int a=5;

auto a=5;

//neu truong hop qua phuc tap, -> xai auto

//normal: không nen xai

Pointer

//pointer

int a =5;

int\* p =&a;

a = 5

&a = 0xA

P = 0xA

&p = 0xP

\*p = 5

# SINGLE LINKED LIST

## Revision

//encapsulation

//mean: mix value and function together

/\*

class Node{

    int data;

    void print(){

        cout<<data;

    }

};

\*/

//member access control: public, private, protected

//constructor, destructor

template <class *T*>

class Node{

private:

*T* data;

*Node*\* next;

public:

    void print(){

        cout<<data;

    }

    Node(*T* *data*){

*this*->data = *data*;

        next = nullptr;

    }

    ~Node(){

        cout<<endl;

    }

};

template<class *T*>

class SLL{

private:

    Node<*T*>\* head;

public:

    SLL(){

        head = nullptr;

    }

    void find(Node<*T*>\* *node*);

    void insert(Node<*T*>\* *node*);

    void delete(Node<*T*>\* *node*);

    //erase vs delete: delete phai giai phong vung nho

};

## Cau hoi 1

Class LLNode is used to store a node in a singly linked list, described on the following:

class LLNode {

public:

int val;

LLNode\* next;

LLNode();

LLNode(int val, LLNode\* next);

}

Where val is the value of node, next is the pointer to the next node.

**Request:** Implement function:

LLNode\* addLinkedList(LLNode\* l0, LLNode\* l1);

Where l0, l1 are two linked lists represented positive integers, each node is a digit, the head is the least significant digit (the value of each node is between 0 and 9, the length of each linked list is between 0 and 100000). This function returns the linked list representing the sum of the two integers.

**Example:**

Given l0 = [2, 3] (representing 32) and l1 = [1, 8] (representing 81). The result would be l0 = [3, 1, 1] (representing 32 + 81 = 113).

**Note:**

In this exercise, the libraries iostream, string, cstring, climits, utility, vector, list, stack, queue, map, unordered\_map, set, unordered\_set, functional, algorithm has been included and namespace std are used. You can write helper functions and classes. Importing other libraries is allowed, but not encouraged, and may result in unexpected errors.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr1[] = {2, 9};  int arr2[] = {0, 5};  LLNode\* head1 = LLNode::createWithIterators(arr1, arr1 + sizeof(arr1) / sizeof(int));  LLNode\* head2 = LLNode::createWithIterators(arr2, arr2 + sizeof(arr2) / sizeof(int));  LLNode\* newhead = addLinkedList(head1, head2);  LLNode::printList(newhead);  head1->clear();  head2->clear();  newhead->clear(); | [2, 4, 1] |

LLNode\* addLinkedList(LLNode\* l0, LLNode\* l1) {

// STUDENT ANSWER

}

*LLNode*\* addLinkedList(*LLNode*\* *l0*, *LLNode*\* *l1*) {

            // STUDENT ANSWER

            if(*l0* == nullptr && *l1* == nullptr) return nullptr;

            //initialize the pointer

            int\* arr = **new** int[100001];

            for(int i = 0; i < 100001; i++){ arr[i] = 0; }

            //input l0 into arr

*LLNode*\* temp1 = *l0*;

            int count0 = 0;

            while(temp1 != nullptr){

                arr[count0] = temp1->val;

                count0++;

                temp1= temp1->next;

            }

            //input l1 into arr

*LLNode*\* temp2 = *l1*;

            int count1 = 0;

            while(temp2 != nullptr){

                arr[count1] += temp2->val;

                count1++;

                temp2= temp2->next;

            }

            //dem so element max trong nLL co the co

            int count;

            if(count0 >= count1) count = count0 + 1;

            else count = count1 + 1;

            //xu ly so du

            for(int i=0; i<count; i++){

                if(arr[i] >= 10) {

                    int temp = arr[i] / 10;

                    arr[i] = arr[i] % 10;

                    arr[i+1] += temp;

                }

            }

            //input array into stack

            stack<int> stack;

            for(int i=0; i<count; i++){

                stack.push(arr[i]);

            }

            if(stack.top() == 0) stack.pop();

            //input stack into nLL

*LLNode*\* head = **new** *LLNode*(stack.top(),nullptr);

            stack.pop();

            while(!stack.empty()){

*LLNode*\* tempNode = **new** *LLNode*(stack.top(), head);

                stack.pop();

                head = tempNode;

            }

            return head;

        }

Code version ôn

*LLNode*\* addLinkedList(*LLNode*\* *l0*, *LLNode*\* *l1*) {

    // STUDENT ANSWER

*LLNode*\* tmp = **new** *LLNode*(0,nullptr);

*LLNode*\* walk = tmp;

*LLNode*\* tmpl0 = *l0*;

*LLNode*\* tmpl1 = *l1*;

    while(tmpl0 != nullptr || tmpl1 != nullptr) {

        if(tmpl0){

            walk->val += tmpl0->val;

            tmpl0 = tmpl0->next;

        }

        if(tmpl1){

            walk->val += tmpl1->val;

            tmpl1 = tmpl1->next;

        }

        if(walk->val > 9){

            walk->val -= 10;

            walk->next = **new** *LLNode*(1,nullptr);

            walk = walk->next;

        }

        else{

            walk->next = **new** *LLNode*(0,nullptr);

            walk = walk->next;

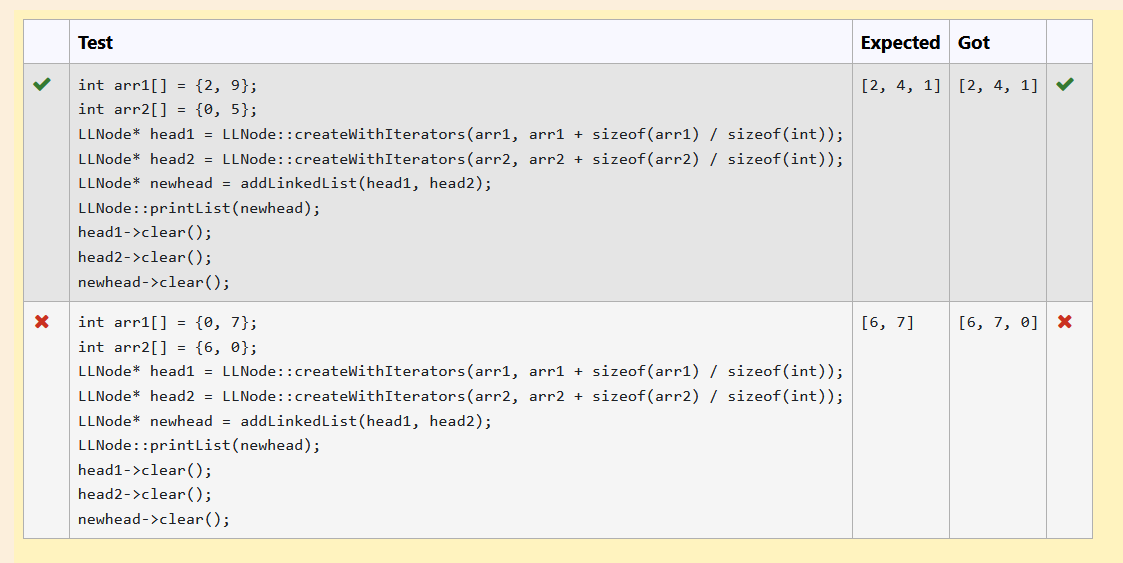
        }

    }

    if(walk->val == 0) walk = nullptr;

    return tmp;

}



## Cau hoi 2

Class LinkedList is used to represent single linked list, described as the following:

class LinkedList {  
    public:   
        class Node;  
    private:  
        Node\* head;  
        Node\* tail;  
        int size;  
    public:   
        class Node {  
            private:   
                int value;  
                Node\* next;  
                friend class LinkedList;  
            public:  
                Node() {  
                    this->next = NULL;  
                }  
                Node(Node\* node) {  
                    this->value = node->value;  
                    this->next = node->next;  
                }  
                Node(int value, Node\* next = NULL) {  
                    this->value = value;  
                    this->next = next;  
                }  
        };  
        LinkedList(): head(NULL), tail(NULL), size(0) {};  
 void partition(int k);  
};

In this class; head, tail and size are the pointers of the first element, the last element and size of linked list.

Request: Implement function

void LinkedList::partition(int k);

To rearrange elements in this linked list by this way: split all elements of this linked list to 3 groups include: group I (the value  of each element is smaller than k), group II (the value of each element is equal to k) and group III (the value of each element is larger than k); after that, move group I to the head  of linked list, group II is after group I and group III is after group II. Note that the orders of the elements of each group are unchanged.

Example:

- The given linked list: 10->55->45->42->50

- k=45

In this case, group I includes elements with value 10 and 42, group II includes elements with 45 and group III includes elements with value 55 and 50. Rearrange by above way,  this linked list will become 10->42->45->55->50 (the orders of th elements in each group is unchanged: 10 before 42 and 55 before 50).

*Note: In this exercise, libraries iostream, sstream and using namespace std; have been used. You can add other functions for your answer, but you are not allowed to add other libraries.*

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| LinkedList\* l1 = new LinkedList();  l1->add(20); l1->add(30); l1->add(10); l1->add(60); l1->add(40); l1->add(45); l1->add(55);  l1->partition(45);  cout << l1->toString() << "\n"; | [20,30,10,40,45,60,55] |

void LinkedList::partition(int k) {

}

void *LinkedList*::partition(int *k*){

    int arr[*this*->size];

    int IdxArr = 0;

    //input elements that is smaller than k

*Node*\* temp = *this*->head;

    while(temp != nullptr){

        if(temp->value < *k*) {

            arr[IdxArr] = temp->value;

            IdxArr++;

        }

        temp = temp->next;

    }

    //input elements that is equal k

    temp = *this*->head;

    while(temp != nullptr){

        if(temp->value == *k*) {

            arr[IdxArr] = temp->value;

            IdxArr++;

        }

        temp = temp->next;

    }

    //input elements that is bigger than k

    temp = *this*->head;

    while(temp != nullptr){

        if(temp->value > *k*) {

            arr[IdxArr] = temp->value;

            IdxArr++;

        }

        temp = temp->next;

    }

    //set new order of Nodes

    temp = *this*->head;

    for(int i=0; i< *this*->size; i++){

        temp->value = arr[i];

        temp = temp->next;

    }

}

## Cau hoi 3

Cho node root của một danh sách liên kết đơn, hiện thực hàm sau đây:

void reduceDuplicate(Node\* root);

để giảm các phần tử trùng lặp liên tiếp trong danh sách. Nếu có các phần tử liên tiếp giống nhau, ta chỉ giữ lại 1 phần tử.

Ví dụ, ta có 1 danh sách 122234452, sau khi thực hiện hàm reduceDuplicate ta thu được danh sách 123452. (số 2 cuối cùng giữ nguyên do nó không liên tiếp với dãy 222 phía trước)

**Lưu ý:** Các bạn có thể include thêm thư viện nếu cần thiết

Cấu trúc của một node được cho bên dưới. Các bạn chỉ cần viết nội dung hàm reduceDuplicate, các cấu trúc khác đã được import sẵn.

class Node

{

int data;

Node\* next;

public:

Node(): data(0), next(nullptr){}

Node(int data, Node\* next)

{

this->data = data;

this->next = next;

}

int getData()

{

return this->data;

}

void setData(int data)

{

this->data = data;

}

Node\* getNext()

{

return this->next;

}

void setNext(Node\* next)

{

this->next = next;

}

};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| Node\* node1 = new Node(1, nullptr);  Node\* node2 = new Node(1, node1);  Node\* node3 = new Node(0, node2);  printList(node3);  reduceDuplicate(node3);  printList(node3); | HEAD -> 0 -> 1 -> 1 -> NULL  HEAD -> 0 -> 1 -> NULL |

void reduceDuplicate(Node\* root)

{

}

void reduceDuplicate(*Node*\* *root*){

        if(*root* == nullptr || *root*->getNext() == nullptr){

            return;

        }

        else{

*Node*\* preNode = *root*;

*Node*\* exNode = *root*->getNext();

            while(exNode != nullptr){

                if(preNode->getData() == exNode->getData()){

                    exNode = exNode->getNext();

*Node*\* tempNode = preNode->getNext();

                    preNode->setNext(preNode->getNext()->getNext());

                    //Error: preNode->getNext() = preNode->getNext()->getNext();

                    //explain: you cannot use {10=20;}

**delete** tempNode;

                }

                else{

                    preNode = preNode->getNext();

                    exNode = exNode->getNext();

                }

            }

        }

    }

## Cau hoi 4

Implement methods**add, size**in template class **SLinkedList (which implements List ADT)**representing the singly linked list with type T with the initialized frame. The description of each method is given in the code.

template <class T>  
class SLinkedList {  
public:  
    class Node; // Forward declaration  
protected:  
    Node\* head;  
    Node\* tail;  
    int count;  
public:  
    SLinkedList();  
    ~SLinkedList();  
 void add(T e);  
 void add(int index, T e);  
 int size();  
public:  
    class Node {  
    private:  
        T data;  
        Node\* next;  
        friend class SLinkedList<T>;  
    public:  
        Node() {  
            next = 0;  
        }  
        Node(Node\* next) {  
            this->next = next;  
        }  
        Node(T data, Node\* next) {  
            this->data = data;  
            this->next = next;  
        }  
    };  
};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| SLinkedList<int> list;  int size = 10;  for(int index = 0; index < size; index++){  list.add(index);  }  cout << list.toString(); | [0,1,2,3,4,5,6,7,8,9] |
| SLinkedList<int> list;  int size = 10;  for(int index = 0; index < size; index++){  list.add(0, index);  }  cout << list.toString(); | [9,8,7,6,5,4,3,2,1,0] |

template <class T>

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

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

}

template<class T>

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

/\* Insert an element into the list at given index. \*/

}

template<class T>

int SLinkedList<T>::size() {

/\* Return the length (size) of list \*/

return 0;

}

template <class *T*>

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

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

    if(count == 0){

        head = **new** Node(e, nullptr);

        tail = head; //NOTE: SLL, when count = 1, head = tail

        //SLL, when count =1, we set tail is nullptr, is so complicated

    }

    else{

        Node\* newNode = **new** Node(e,nullptr);

        tail->next = newNode;

        tail = tail->next;

    }

    count++;

}

template<class *T*>

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

    /\* Insert an element into the list at given index. \*/

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

    if(count == 0){

        add(e);

    }

    else if(index == 0){

        Node\* newNode = **new** Node(e,head);

        head = newNode;

        count++;

    }

    else if(index == count){

        add(e);

    }

    else{

        Node\* temp = head;

        for(int i = 0; i < index - 1; i++){

            temp = temp->next;

        }

        Node\* newNode = **new** Node(e,temp->next);

        temp->next = newNode;

        count++;

    }

}

template<class *T*>

int SLinkedList<*T*>::size() {

    /\* Return the length (size) of list \*/

    return *this*->count;

}

## Cau hoi 5

Implement methods**get, set, empty, indexOf, contains**in template class **SLinkedList (which implements List ADT)**representing the singly linked list with type T with the initialized frame. The description of each method is given in the code.

template <class T>  
class SLinkedList {  
public:  
    class Node; // Forward declaration  
protected:  
    Node\* head;  
    Node\* tail;  
    int count;  
public:  
    SLinkedList(): head(NULL), tail(NULL), count(0);  
    ~SLinkedList() { };  
 void add(T e);  
 void add(int index, T e);  
 int size();  
 bool    empty();  
    T       get(int index);  
    void    set(int index, T e);  
    int     indexOf(T item);  
    bool    contains(T item);  
public:  
    class Node {  
    private:  
        T data;  
        Node\* next;  
        friend class SLinkedList<T>;  
    public:  
        Node() {  
            next = 0;  
        }  
        Node(Node\* next) {  
            this->next = next;  
        }  
        Node(T data, Node\* next = NULL) {  
            this->data = data;  
            this->next = next;  
        }  
    };  
};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| SLinkedList<int> list;  int values[] = {10, 15, 2, 6, 4, 7, 40, 8};  int index[] = {0, 0, 1, 3, 2, 3, 5, 0};  int expvalues[]= {8, 15, 2, 4, 7, 10, 40, 6};  for (int idx = 0; idx < 8; idx++){  list.add(index[idx], values[idx]);  }  assert( list.size() == 8 );    for (int idx = 0; idx < 8; idx++){  assert( list.get(idx) == expvalues[idx] );  }  cout << list.toString(); | [8,15,2,4,7,10,40,6] |
| SLinkedList<int> list;  assert( list.empty() == true );  cout << list.toString(); | [] |

template<class T>

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

/\* Give the data of the element at given index in the list. \*/

}

template <class T>

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

/\* Assign new value for element at given index in the list \*/

}

template<class T>

bool SLinkedList<T>::empty() {

/\* Check if the list is empty or not. \*/

}

template<class T>

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

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

}

template<class T>

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

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

}

template<class *T*>

*T* SLinkedList<*T*>::get(int *index*) {

    /\* Give the data of the element at given index in the list. \*/

    //CHECK whether index is valid, to avoid segmentation fault

    if(*index* < 0 || *index* >= count){

        throw std::*out\_of\_range*(" 1 == 1 ");

    }

*Node*\* temp = head;

    int i = 0;

    while(i != *index*){

        temp = temp->next;

        i++;

    }

    return temp->data;

}

template <class *T*>

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

    /\* Assign new value for element at given index in the list \*/

    //CHECK whether index is valid, to avoid segmentation fault

    if(index < 0 || index >= count){

        throw std::out\_of\_range(" 1 == 1 ");

    }

    Node\* temp = head;

    int i = 0;

    while(i != index){

        temp = temp->next;

        i++;

    }

    temp->data = e;

}

template<class *T*>

bool SLinkedList<*T*>::empty() {

    /\* Check if the list is empty or not. \*/

    return count ==0;

}

template<class *T*>

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

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

    Node\* temp = head;

    int i = 0;

    while(temp != nullptr){

        if(temp->data == item){

            return i;

        }

        else{

            temp = temp->next;

            i++;

        }

    }

    return -1;

}

template<class *T*>

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

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

    //advice: tận dụng các hàm đã viết

    return indexOf(item) != -1;

}

## Cau hoi 6

Implement methods**removeAt, removeItem, clear**in template class **SLinkedList (which implements List ADT)**representing the singly linked list with type T with the initialized frame. The description of each method is given in the code.

template <class T>  
class SLinkedList {  
public:  
    class Node; // Forward declaration  
protected:  
    Node\* head;  
    Node\* tail;  
    int count;  
public:  
    SLinkedList();  
    ~SLinkedList();  
 void add(T e);  
 void add(int index, T e);  
 int size();  
 bool    empty();  
    int     size();  
    void    clear();  
    T       get(int index);  
    void    set(int index, T e);  
    int     indexOf(T item);  
    bool    contains(T item);  
 T       removeAt(int index);  
    bool    removeItem(T item);  
public:  
    class Node {  
    private:  
        T data;  
        Node\* next;  
        friend class SLinkedList<T>;  
    public:  
        Node() {  
            next = 0;  
        }  
        Node(Node\* next) {  
            this->next = next;  
        }  
        Node(T data, Node\* next = NULL) {  
            this->data = data;  
            this->next = next;  
        }  
    };  
};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| SLinkedList<int> list;  for (int i = 0; i < 10; ++i) {  list.add(i);  }  assert(list.get(0) == list.removeAt(0));  cout << list.toString(); | [1,2,3,4,5,6,7,8,9] |
| SLinkedList<int> list;  for (int i = 0; i < 10; ++i) {  list.add(i);  }  assert(list.get(9) == list.removeAt(9));  cout << list.toString(); | [0,1,2,3,4,5,6,7,8] |
| SLinkedList<int> list;  for (int i = 0; i < 10; ++i) {  list.add(i);  }  assert(list.removeItem(9));  cout << list.toString(); | [0,1,2,3,4,5,6,7,8] |

template <class T>

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

{

/\* Remove element at index and return removed value \*/

}

template <class T>

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

{

/\* Remove the first apperance of item in list and return true, otherwise return false \*/

}

template<class T>

void SLinkedList<T>::clear(){

/\* Remove all elements in list \*/

}

template <class *T*>

*T* SLinkedList<*T*>::removeAt(int *index*)

{

    /\* Remove element at index and return removed value \*/

    //CHECK whether index is valid, to avoid segmentation fault

    if(*index* < 0 || *index* >= count){

        throw std::*out\_of\_range*(" 1 == 1 ");

    }

    if(*index* == 0){

*Node*\* temp = head;

        head = head->next;

        temp->next = nullptr;

        //notice:

        count--;

        return temp->data;

    }

    else if(*index* == count-1){

*Node*\* prevTail = head;

        while(prevTail->next != tail){

            prevTail = prevTail->next;

        }

        //From here: prevTail is forward Node of Tail Node

*Node*\* temp = tail;

        tail = prevTail;

        tail->next = nullptr;

        //notice:

        count--;

        return temp->data;

    }

    else{

        int tempIndex;

*Node*\* prevNode = head;

        for(tempIndex=0; tempIndex< *index - 1*; tempIndex++){

            prevNode = prevNode->next;

        }

        //from here: tempIndex = index

*Node*\* nextNode = prevNode->next->next;

*Node*\* tempNode = prevNode->next;

        tempNode->next = nullptr;

        prevNode->next = nextNode;

        //notice:

        count--;

        return tempNode->data;

    }

}

template <class *T*>

bool SLinkedList<*T*>::removeItem(const *T*& *item*)

{

    /\* Remove the first apperance of item in list and return true, otherwise return false \*/

    if(contains(item) == false) return false;

    int index = indexOf(item);

    removeAt(index);

    return true;

}

template<class *T*>

void SLinkedList<*T*>::clear(){

    /\* Remove all elements in list \*/

    if(head == nullptr){return;}

    else{

        while(head != nullptr){

*Node*\* temp = head;

            head = head->next;

            delete temp;

        }

    }

    count = 0;

}

## Cau hoi 7

Class LLNode representing a node of singly linked lists is declared as below:

class LLNode {

public:

    int val;

    LLNode\* next;

    LLNode(); // Constructor: val = 0, next = nullptr

    LLNode(int val, LLNode\* next); // Constructor with customized data

Given a singly linked list head node.

Your task is to implement a function with following prototype:

LLNode\* foldLinkedList(LLNode\* head);

The function returns head node of the folded singly linked list.

**More info:**

A folded singly linked list is achieved by taking the first half of the linked list and folding over the second half and merging the intersecting nodes by taking their sum.

**Note:**

- The iostream library has been used and namespace std is being used. No other libraries are allowed.

- The constructors and methods of class LLNode have been defined fully so you do not need to redefine them.

- You can write helper functions.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {9, 5, 7, 5, 0};  LLNode\* head = LLNode::createWithIterators(arr, arr + sizeof(arr) / sizeof(int));  LLNode::printList(head);  cout << "\n";  LLNode\* newhead = foldLinkedList(head);  LLNode::printList(newhead); | [9, 5, 7, 5, 0]  [7, 10, 9] |
| int arr[] = {8, 4, 6, 8, 5, 3};  LLNode\* head = LLNode::createWithIterators(arr, arr + sizeof(arr) / sizeof(int));  LLNode::printList(head);  cout << "\n";  LLNode\* newhead = foldLinkedList(head);  LLNode::printList(newhead); | [8, 4, 6, 8, 5, 3]  [14, 9, 11] |

LLNode\* foldLinkedList(LLNode\* head) {

// STUDENT ANSWER

}

int findsize(*LLNode*\* *head*) {

*LLNode*\* temp = *head*;

        int count = 0;

        while(temp != nullptr) {

            count++;

            temp = temp->next;

        }

        return count;

    }

*LLNode*\* getNode(int *index*, *LLNode*\* *head*) {

        int count = 0;

*LLNode*\* temp = *head*;

        while(count != *index*){

            temp = temp->next;

            count++;

        }

        return temp;

    }

*LLNode*\* foldLinkedList(*LLNode*\* *head*){

        //find the number of LinkedList

        int count = findsize(*head*);

        //special case: count ==1, ==0

        if(count ==1 || count == 0) {return *head*;}

        //normal case: count >=2

        //case 1: count is even

        if(count % 2 == 0){

            //change value with index [count/2, count -1]

            for(int i = count/2; i < count; i++){

                getNode(i,*head*)->val += getNode(count-1-i,*head*)->val;

            }

        }

        else{

            //case 2: count is odd

            //change value with index [count/2 + 1, count -1]

            for(int i = count/2 + 1; i < count; i++){

                getNode(i,*head*)->val += getNode(count-1-i,*head*)->val;

            }

        }

        //get LL: index [count/2, count -1]

        return getNode(count/2,*head*);

    }

## Cau hoi 8

Class LLNode representing a node of singly linked lists is declared as below:

class LLNode {

public:

    int val;

    LLNode\* next;

    LLNode(); // Constructor: val = 0, next = nullptr

    LLNode(int val, LLNode\* next); // Constructor with customized data

}

Given a singly linked list head node.

Your task is to implement a function with following prototype:

LLNode\* replaceFirstGreater(LLNode\* head);

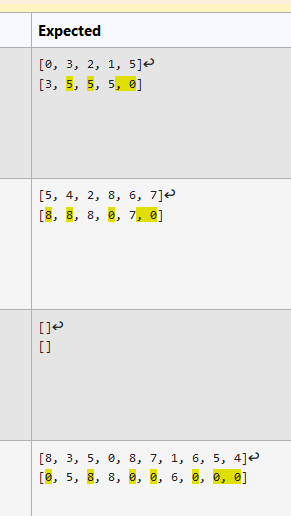
The function returns head node of the singly linked list after replacing every node's value with the first greater node's value to its right. If a node doesn't have a next greater node, set its value to 0.

**Note:**

- The iostream and stack libraries have been included and namespace std is being used. No other libraries are allowed.

- The constructors and methods of class LLNode have been defined fully so you do not need to redefine them.

- You can write helper functions.



LLNode\* replaceFirstGreater(LLNode\* head) {

// STUDENT ANSWER

}

bool firstgreater(*LLNode*\* *head*, int& *revalue*);

*LLNode*\* replaceFirstGreater(*LLNode*\* *head*) {

        // STUDENT ANSWER

        //if(head == nullptr) return head; //do nothing

        LLNode\* temp = head;

        while(temp != nullptr) {

            int revalue;

            if(firstgreater(temp, revalue)){

                temp->val = revalue;

            }

            else {temp->val = 0;}

            temp= temp->next;

            //continue to change value of next node

        }

        return head;

    }

    bool firstgreater(*LLNode*\* *head*, int& *revalue*){

        LLNode\* temp = head->next;

        while(temp != nullptr){

            if(temp->val > head->val){

                revalue = temp->val;

                return true;

            }

            else{

                temp = temp->next;

            }

        }

        return false; //if head -> next == nullptr

    }

## Cau hoi 9

Class LLNode representing a node of singly linked lists is declared as below:

class LLNode {

public:

    int val;

    LLNode\* next;

    LLNode(); // Constructor: val = 0, next = nullptr

    LLNode(int val, LLNode\* next); // Constructor with customized data

}

Given a singly linked list head node.

Your task is to implement a function with following prototype:

LLNode\* reverseLinkedList(LLNode\* head);

The function returns head node of the reversed singly linked list.

**Note:**

- The iostream library has been included and namespace std is being used. No other libraries are allowed.

- The constructors and methods of class LLNode have been defined fully so you do not need to redefine them.

- You can write helper functions.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {13, 88, 60, 7, 192};  LLNode\* head = LLNode::createWithIterators(arr, arr + sizeof(arr) / sizeof(int));  LLNode::printList(head);  cout << "\n";  LLNode\* newhead = reverseLinkedList(head);  LLNode::printList(newhead);  newhead->clear(); | [13, 88, 60, 7, 192]  [192, 7, 60, 88, 13] |

LLNode\* reverseLinkedList(LLNode\* head) {

// STUDENT ANSWER

}

*LLNode*\* reverseLinkedList(*LLNode*\* *head*) {

        // STUDENT ANSWER

*LLNode*\* previous = nullptr;

*LLNode*\* current = *head*;

*LLNode*\* Next = nullptr;

        while(current != nullptr) {

            Next = current->next;

            current->next = previous;

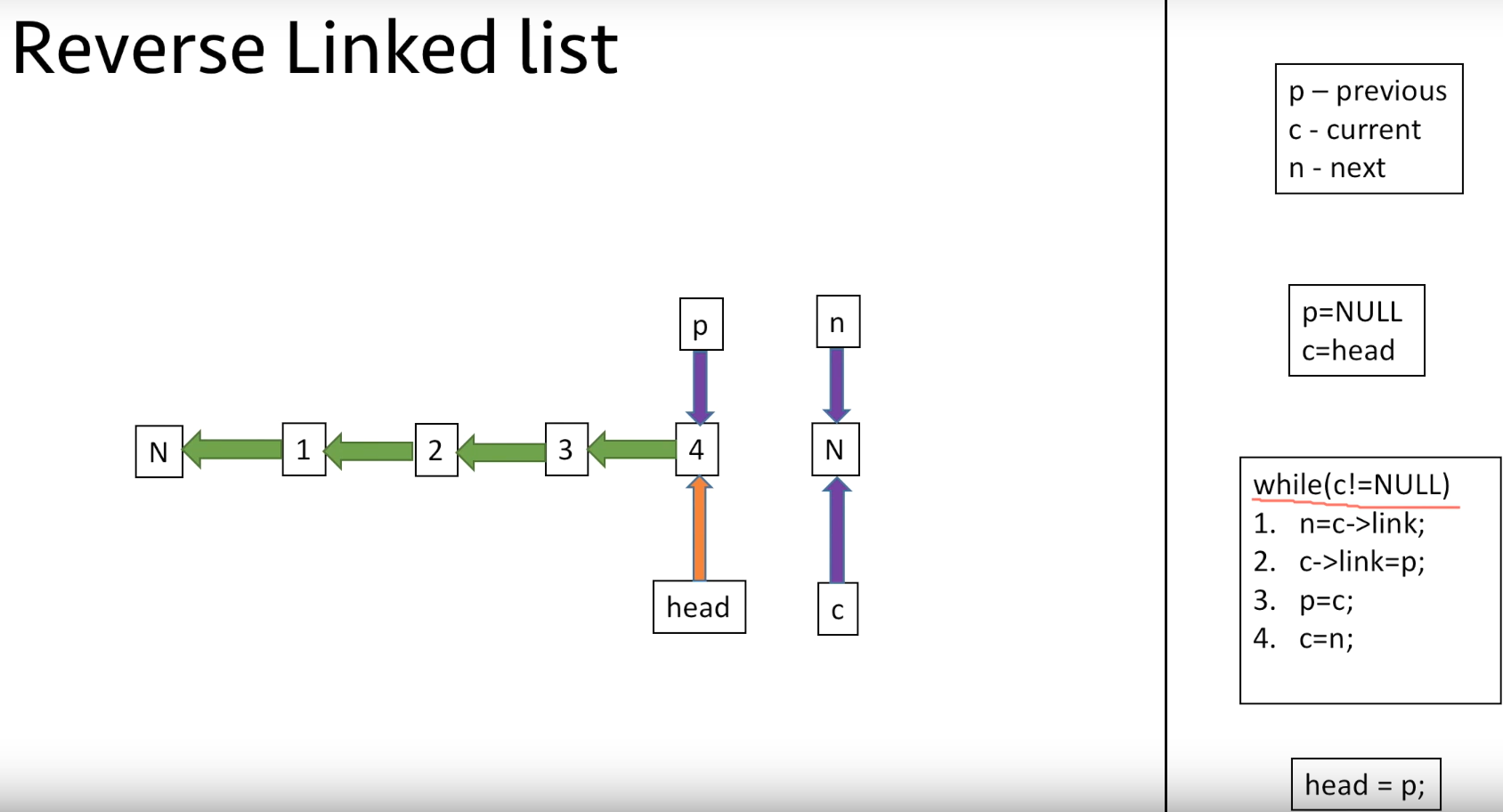
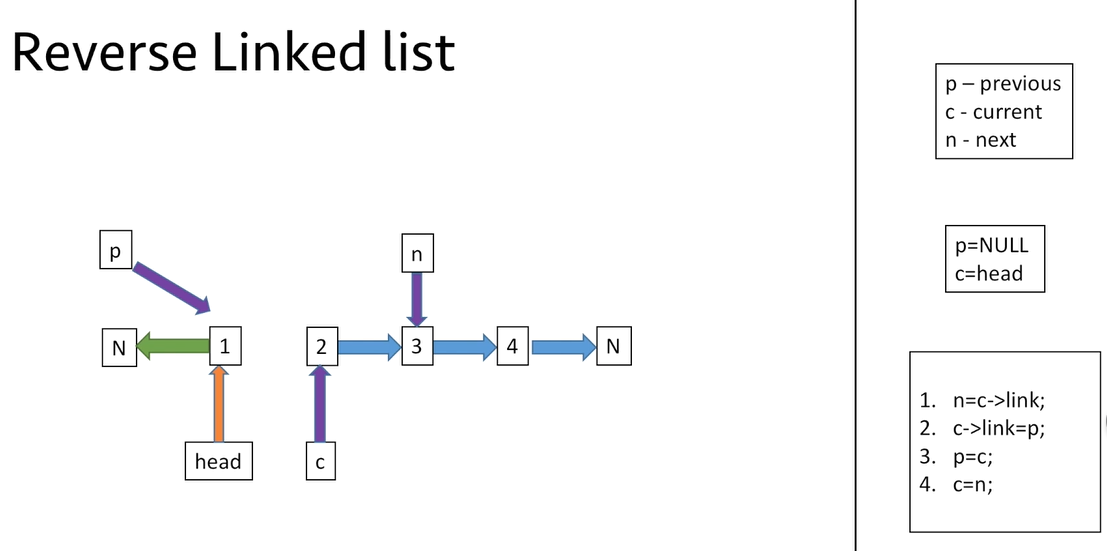
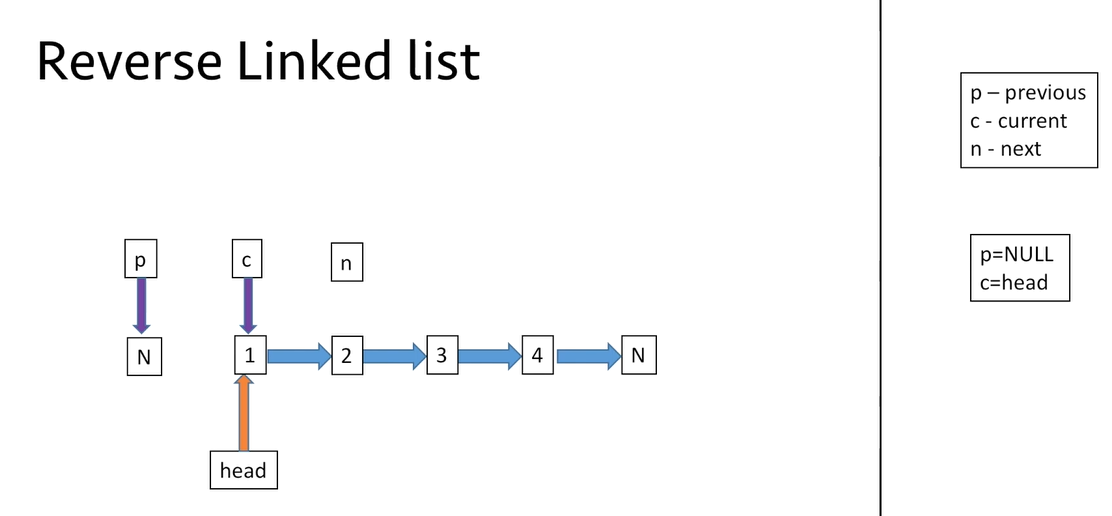
            previous = current;

            current = Next;

        }

return previous;

}



## Cau so 10

Class LLNode representing a node of singly linked lists is declared as below:

class LLNode {

public:

    int val;

    LLNode\* next;

    LLNode(); // Constructor: val = 0, next = nullptr

    LLNode(int val, LLNode\* next); // Constructor with customized data

}

Given a singly linked list head node and a integer k.

Your task is to implement a function with following prototype:

LLNode\* rotateLinkedList(LLNode\* head, int k);

The function returns head node of the rotated singly linked list obtained after rotate the linked list to the right by k places.

**Note:**

- The iostream library has been used and namespace std is being used. No other libraries are allowed.

- The constructors and methods of class LLNode have been defined fully so you do not need to redefine them.

- You can write helper functions.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| int arr[] = {2, 4, 6, 6, 3};  int k = 3;  LLNode\* head = LLNode::createWithIterators(arr, arr + sizeof(arr) / sizeof(int));  LLNode::printList(head);  cout << "\n";  LLNode\* newhead = rotateLinkedList(head, k);  LLNode::printList(newhead); | [2, 4, 6, 6, 3]  [6, 6, 3, 2, 4] |

LLNode\* rotateLinkedList(LLNode\* head, int k) {

// STUDENT ANSWER

}

*LLNode*\* rotateLinkedList(*LLNode*\* *head*, int *k*) {

        // STUDENT ANSWER

        //special case: head is nullptr

        if(*head* == nullptr) return *head*;

        //find count of nodes and tail node

*LLNode*\* node = *head*;

*LLNode*\* tail = nullptr;

        int count = 0;

        while(node != nullptr) {

            count++;

            if(node->next == nullptr) tail = node;

            node = node->next;

        }

        //set circular LL

        tail->next = *head*;

        //move head and tail nodes

        //rotate Right (counterclockwise): need move head node count-dist (steps)

        //rotate Left (clockwise): need move head node dist (steps)

        int dist = *k* % count;

        for(int i = 0; i < (count-dist); i++) {

*head* = *head*->next;

            tail = tail->next;

        }

        tail->next = nullptr;

        return *head*;

    }

## Cau hoi 11 – dung mot phan

- Class LinkedList is used to represent single linked list, described as the following:

class LinkedList {  
    public:   
        class Node;  
    private:  
        Node\* head;  
        Node\* tail;  
        int size;  
    public:   
        class Node {  
            private:   
                int value;  
                Node\* next;  
                friend class LinkedList;  
            public:  
                Node() {  
                    this->next = NULL;  
                }  
                Node(Node\* node) {  
                    this->value = node->value;  
                    this->next = node->next;  
                }  
                Node(int value, Node\* next = NULL) {  
                    this->value = value;  
                    this->next = next;  
                }  
        };  
        LinkedList(): head(NULL), tail(NULL), size(0) {};  
 void replace(LinkedList\* linked\_list, int low, int high);  
};

- In this class; head, tail and size are the pointers of the first element, the last element and size of linked list.  
- Request: Implement function 'replace(LinkedList\* linked\_list, int low, int high)'  (low <= high and high >= 0) to delete all elements from position low to position high in the linked list (head's position is 0). After that, add linked\_list to this linked list in the position after position (low - 1) (if low - 1 < 0, the head of this linked list will become the head of linked\_list.   
(If high >= size, delete all elements from low position to the last position of this linked list. If low < 0, delete all elements from the first position of this linked list to high position)   
- Example:  
+ The given linked list: 10->20->30->40->50  
+ The linked\_list: 55->45->42  
+ low=1, high=3  
In this case, delete all elements from position 1 (value 20) to position 3 (value 40). After that, add linked\_list to this linked list to the position after position 0 (value 10), therefore, this linked list will become: 10->55->45->42->50 .  
Note: In this exercise, library 'iostream' and 'sstream' and 'using namespace std;' has been used. You can add other functions, however, you are not allowed to add other libraries.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| LinkedList\* l1 = new LinkedList();  LinkedList\* l2 = new LinkedList();  l1->add(10); l1->add(20); l1->add(30); l1->add(40); l1->add(50);  l2->add(55); l2->add(45); l2->add(42);  l1->replace(l2, -1, 2);  cout << l1->toString() << "\n"; | [55,45,42,40,50] |

void LinkedList::replace(LinkedList\* linked\_list, int low, int high) {

}

void *LinkedList*::replace(*LinkedList*\* *linked\_list*, int *low*, int *high*) {

    //reset int low and int high

    if(*low* < 0) *low* = 0;

    if(*high* >= *this*->size) *high* = *this*->size - 1;

    //DONT CARE: special case: nLL->size = 0

    //initialize nextHigh of this LL

*Node*\* nextHigh = head;

    int i = 0;

    while(i < *high* + 1){

        nextHigh = nextHigh->next;

        i++;

    }

    //case: low ==0 (there is no preLow Node)

    if(*low* == 0) {

        //special case: size of nLL = 1

        if(*linked\_list*->tail == nullptr) *linked\_list*->head->next = nextHigh;

        else{

            //other cases: nLL size >=2

*linked\_list*->tail->next = nextHigh;

        }

*this*->head = *linked\_list*->head;

    }

    else{

        //case: low <= high

        //initialize preLow of this LL

*Node*\* preLow = head;

        int j = 0;

        while(j < *low* - 1){

            preLow = preLow->next;

            j++;

        }

        preLow->next = *linked\_list*->head;

        //special case: size of nLL = 1

        if(*linked\_list*->tail == nullptr) *linked\_list*->head->next = nextHigh;

        else{

            //other cases: nLL size >=2

*linked\_list*->tail->next = nextHigh;

        }

    }

    //resize

*this*->size += *linked\_list*->size - (*high* - *low* +1);

}

# DOUBLE LINKED LIST

## Cau hoi 1

A double-ended queue or deque (pronounced "deck") is like a queue or a stack but supports adding and removing items at both ends.

A deque stores a collection of items and supports the following methods:

+ getSize(): int => number of items in the deque

+ pushFront(int item): void => add an item to the left end

+ pushBack(int item): void => add an item to the right end

+ popFront(): int => remove and return an item from the left end

+ popBack(): int => remove and return an item from the right end

+ clear(): void => erase all items in the deque

// For checking purposes

+ printDeque(): void => print all items in the deque from left to right, separated by a space, with a new line (i.e '\n') at the end.

+ printDequeReverse(): void => print all items in the deque from right to left, separated by a space, with a new line at the end.

Note: if the deque is empty, every pop method return -1;

class Deque {

private:

class Node {

private:

int value;

Node\* left;

Node\* right;

friend class Deque;

public:

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

};

private:

Node\* head;

Node\* tail;

int curSize;

public:

Deque();

~Deque();

int getSize();

void pushFront(int item);

void pushBack(int item);

int popFront();

int popBack();

void clear();

void printDequeReverse();

void printDeque();

};

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| Deque\* deque = new Deque();  vector<int> arr = {223, 1234, 43, 568, 90, 193, 2109};  for(int i = 0; i < (int)arr.size(); i++) {  if (i < (int)arr.size() / 2)  deque->pushFront(arr[i]);  else {  deque->pushBack(arr[i]);  }  }  deque->printDeque();  cout << deque->getSize();  delete deque; | 43 1234 223 568 90 193 2109  7 |
| Deque\* deque = new Deque();  int size = 20;  for(int i = 0; i < size; i++) {  deque->pushBack(i);  }  for(int i = 0; i < size / 2; i++) {  if (i % 2 == 0)  deque->popBack();  else  deque->popFront();  }  deque->printDequeReverse();  cout << deque->getSize();  delete deque; | 14 13 12 11 10 9 8 7 6 5  10 |

Deque::Deque() {

}

Deque::~Deque() {

}

void Deque::clear() {

}

int Deque::getSize() {

}

void Deque::pushFront(int i) {

}

void Deque::pushBack(int i) {

}

int Deque::popFront() {

}

int Deque::popBack() {

}

void Deque::printQueueReverse() {

}

void Deque::printQueue() {

}

Deque::Deque() {

    head = nullptr;

    tail = nullptr;

    curSize = 0;

}

Deque::~Deque() {

}

void Deque::clear() {

    int tempsize = curSize;

    for(int i = 0; i < tempsize; i++) popFront();

}

int Deque::getSize() {

    return curSize;

}

void Deque::pushFront(int *i*) {

    if (head == nullptr){

        Node\* newNode = **new** Node(i,nullptr,nullptr);

        head = newNode;

        tail = newNode;

    } else {

        Node\* newNode = **new** Node(i,nullptr,head);

        head->left = newNode;

        head = newNode;

    }

    curSize += 1;

}

void Deque::pushBack(int *i*) {

    if (head == nullptr){

        Node\* newNode = **new** Node(i,nullptr,nullptr);

        head = newNode;

        tail = newNode;

    } else {

        Node\* newNode = **new** Node(i,tail,nullptr);

        tail->right = newNode;

        tail = newNode;

    }

    curSize += 1;

}

int Deque::popFront() {

    if(curSize == 0) return -1;

    Node\* popNode = head;

    int popValue = popNode->value;

    if(popNode->right){

        head = head->right;

        head->left = nullptr;

    }

    else{

        head = nullptr;

        tail = nullptr;

    }

    //delete popNode

    delete popNode;

    //return result

    curSize--;

    return popValue;

}

int Deque::popBack() {

    if(curSize == 0) return -1;

    Node\* popNode = tail;

    int popValue = popNode->value;

    if(popNode->left){

        tail = tail->left;

        tail->right=nullptr;

    }

    else{

        tail = nullptr;

        head = nullptr;

    }

    //delete popNode

    delete popNode;

    //return result

    curSize--;

    return popValue;

}

void Deque::printDequeReverse() {

   Node\* temp = tail;

    while(temp != nullptr) {

        if(temp == head) cout<<temp->value<<"\n";

        else cout<<temp->value<<" ";

        temp = temp->left;

    }

}

void Deque::printDeque() {

    Node\*temp=head;

    while(temp!=nullptr){

        cout<<temp->value<<" ";

        temp=temp->right;

    }

    cout<<endl;

}

## Cau hoi 2

Implement methods**add, size**in template class **DLinkedList (which implements List ADT)**representing the doubly linked list with type T with the initialized frame. The description of each method is given in the code.

template <class T>  
class DLinkedList {  
public:  
    class Node; // Forward declaration  
protected:  
    Node\* head;  
    Node\* tail;  
    int count;  
public:  
    DLinkedList();  
    ~DLinkedList();  
 void add(const T &e);  
 void add(int index, const T &e);  
 int size();  
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;

}

};

};

In this exercise, we have include <iostream>, <string>, <sstream> and using namespace std.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| DLinkedList<int> list;  int size = 10;  for(int idx=0; idx < size; idx++){  list.add(idx);  }  cout << list.toString(); | [0,1,2,3,4,5,6,7,8,9] |
| DLinkedList<int> list;  int size = 10;  for(int idx=0; idx < size; idx++){  list.add(0, idx);  }  cout << list.toString(); | [9,8,7,6,5,4,3,2,1,0] |

template <class T>

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

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

}

template<class T>

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

/\* Insert an element into the list at given index. \*/

}

template<class T>

int DLinkedList<T>::size() {

/\* Return the length (size) of list \*/

return 0;

}

template <class *T*>

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

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

    if(head == nullptr){

        head = **new** *Node*(*e*);

        tail = head; //NOTE: SLL, when count = 1, head = tail

        //SLL, when count =1, we set tail is nullptr, is so complicated

        count++;

    }

    else{

*Node*\* newNode = **new** *Node*(*e*);

        newNode->previous = tail;

        tail->next = newNode;

        tail = tail->next;

        count++;

    }

}

template<class *T*>

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

    /\* Insert an element into the list at given index. \*/

    if(*index* < 0 || *index* > count) throw std::*out\_of\_range*("Index out of range");

    if(count == 0){

        add(*e*);

    }

    else if(*index* == 0){

*Node*\* newNode = **new** *Node*(*e*);

        newNode->next = head;

        head->previous = newNode;

        head = newNode;

        count++;

    }

    else if(*index* == count){

        add(*e*);

    }

    else{

*Node*\* prevIndex = head;

        for(int i = 0; i < *index* - 1; i++){

            prevIndex = prevIndex->next;

        }

*Node*\* newNode = **new** *Node*(*e*);

        newNode->next = prevIndex->next;

        newNode->next->previous = newNode;

        prevIndex->next = newNode;

        newNode->previous = prevIndex;

        count++;

    }

}

template<class *T*>

int DLinkedList<*T*>::size() {

    /\* Return the length (size) of list \*/

    return count;

}

## Cau hoi 3

Implement methods**get, set, empty, indexOf, contains**in template class D**LinkedList (which implements List ADT)**representing the [singly linked list](https://e-learning.hcmut.edu.vn/mod/quiz/view.php?id=108613) with type T with the initialized frame. The description of each method is given in the code.

template <class T>  
class DLinkedList {  
public:  
    class Node; // Forward declaration  
protected:  
    Node\* head;  
    Node\* tail;  
    int count;  
public:  
    DLinkedList();  
    ~DLinkedList();  
 void add(const T &e);  
 void add(int index, const T &e);  
 int size();  
 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;

}

};

};

In this exercise, we have include <iostream>, <string>, <sstream> and using namespace std.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| 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> 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(idx);  }  for(int idx=0; idx < size; idx++){  list.set(idx, value[idx]);  }  cout << list.toString(); | [2,5,6,3,67,332,43,1,0,9] |

template<class T>

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

/\* Give the data of the element at given index in the list. \*/

}

template <class T>

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

/\* Assign new value for element at given index in the list \*/

}

template<class T>

bool DLinkedList<T>::empty() {

/\* Check if the list is empty or not. \*/

}

template<class T>

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

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

}

template<class T>

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

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

}

template<class *T*>

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

    /\* Give the data of the element at given index in the list. \*/

    if(*index* < 0 || *index* >= count) throw std::*out\_of\_range*("Index out of range");

*Node*\* temp = head;

    for(int i = 0; i < *index*; i++) temp = temp->next;

    return temp->data;

}

template <class *T*>

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

    /\* Assign new value for element at given index in the list \*/

    if(*index* < 0 || *index* >= count) throw std::*out\_of\_range*("Index out of range");

*Node*\* temp = head;

    for(int i = 0; i < *index*; i++) temp = temp->next;

    temp->data = *e*;

}

template<class *T*>

bool DLinkedList<*T*>::empty() {

    /\* Check if the list is empty or not. \*/

    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*\* temp = head;

    int index = 0;

    while(temp != NULL) {

        if(temp->data == *item*) return index;

        index++;

        temp = temp->next;

    }

    //case: item is not in list

    return -1;

}

template<class *T*>

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

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

*Node*\* temp = head;

    while(temp != NULL) {

        if(temp->data == *item*) return true;

        temp = temp->next;

    }

    return false;

}

## Cau hoi 4

Implement methods**removeAt, removeItem, clear**in template class **SLinkedList (which implements List ADT)**representing the [singly linked list](https://e-learning.hcmut.edu.vn/mod/quiz/view.php?id=108613) with type T with the initialized frame. The description of each method is given in the code.

template <class T>  
class DLinkedList {  
public:  
    class Node; // Forward declaration  
protected:  
    Node\* head;  
    Node\* tail;  
    int count;  
public:  
    DLinkedList();  
    ~DLinkedList();  
 void add(const T &e);  
 void add(int index, const T &e);  
 int size();  
 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;

}

};

};

In this exercise, we have include <iostream>, <string>, <sstream> and using namespace std.

For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| 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] |

template <class T>

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

{

/\* Remove element at index and return removed value \*/

}

template <class T>

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

{

/\* Remove the first apperance of item in list and return true, otherwise return false \*/

}

template<class T>

void DLinkedList<T>::clear(){

/\* Remove all elements in list \*/

}

template <class *T*>

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

{

    /\* Remove element at index and return removed value \*/

    if(head == NULL) throw std::*out\_of\_range*("head is nullptr");

    if(*index* < 0 || *index* >= count) throw std::*out\_of\_range*("Index out of range");

    if(*index* == 0) {

*Node*\* removeNode = head;

        head = head->next;

        head->previous = nullptr;

        count--;

        return removeNode->data;

    }

    if(*index* == count - 1) {

*Node*\* removeNode = tail;

        tail = tail->previous;

        tail->next = nullptr;

        count--;

        return removeNode->data;

    }

    //case: index (0,count-1)

        //find preNode

*Node*\* prevNode = head;

            int i = 0;

            while(i < *index* - 1){

                prevNode = prevNode->next;

                i++;

            }

        //find nextNode

*Node*\* nextNode = tail;

        i = count - 1;

        while(i > *index* + 1){

            nextNode = nextNode->previous;

            i--;

        }

        //link prevNode and nextNode

*Node*\* removeNode = prevNode->next;

        prevNode->next = nextNode;

        nextNode->previous = prevNode;

        count--;

        return removeNode->data;

}

template <class *T*>

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

{

    /\* Remove the first apperance of item in list and return true, otherwise return false \*/

    if(contains(*item*) == false) return false;

    //case: item exists

    int index = indexOf(*item*);

    removeAt(index);

    return true;

}

template<class *T*>

void DLinkedList<*T*>::clear(){

    /\* Remove all elements in list \*/

    Node\* temp = head;

    while(temp!= nullptr){

        Node\* next = temp->next;

        delete temp;

        temp = next;

    }

    head = nullptr;

    tail = nullptr;

    count = 0;

}

## Cau hoi 5

In this exercise, we will use [Standard Template Library List](https://www.geeksforgeeks.org/list-cpp-stl/) (click open in other tab to show more) to implement a Data Log.

This is a simple implementation in applications using undo and redo. For example in Microsoft Word, you must have nodes to store states when Ctrl Z or Ctrl Shift Z to go back or forward.

DataLog has a doubly linked list to store the states of data (an integer) and iterator to mark the current state. Each state is stored in a node, the transition of states is depicted in the figure below.

Your task in this exercise is implement functions marked with /\*  \* TODO   \*/.

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 << " ] => ";

}

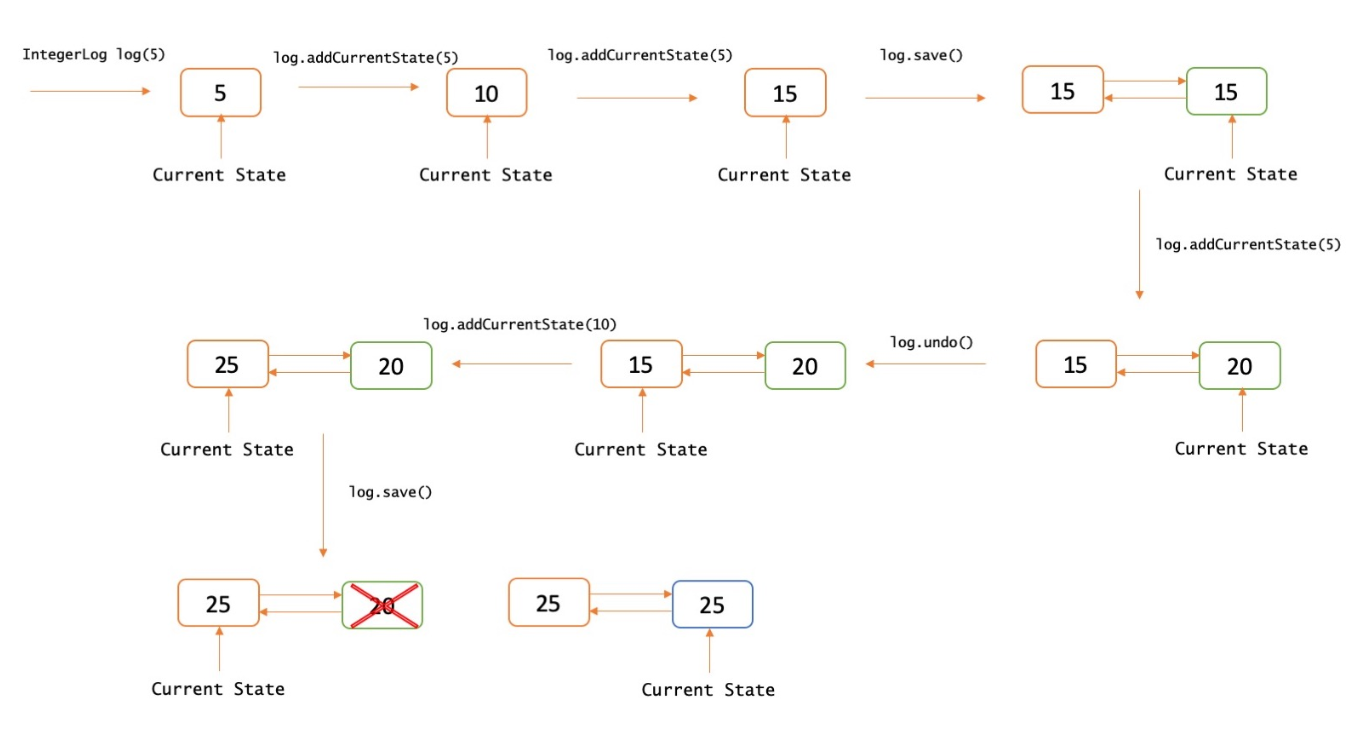
cout << "END\_LOG";

}

};

Note: Normally, when we say a List, we talk about doubly linked list. For implementing a [singly linked list](https://e-learning.hcmut.edu.vn/mod/quiz/view.php?id=108613), we use forward list.

We have include <iostream> <list> and using namespace std;



For example:

|  |  |
| --- | --- |
| **Test** | **Result** |
| DataLog log(10);  log.save();  log.addCurrentState(15);  log.save();  log.addCurrentState(15);  log.undo();  log.printLog(); | [ 10 ] => Current state: [ 25 ] => [ 40 ] => END\_LOG |
| DataLog log(10);  log.save();  log.addCurrentState(15);  log.save();  log.addCurrentState(15);  log.save();  log.subtractCurrentState(5);  log.printLog(); | [ 10 ] => [ 25 ] => [ 40 ] => Current state: [ 35 ] => END\_LOG |

DataLog::DataLog()

{

/\*

\* TODO: add the first state with 0

\*/

}

DataLog::DataLog(const int &data)

{

/\*

\* TODO: add the first state with data

\*/

}

void DataLog::addCurrentState(int number)

{

/\*

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

\*/

}

void DataLog::subtractCurrentState(int number)

{

/\*

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

\*/

}

void DataLog::undo()

{

/\*

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

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

\*/

}

void DataLog::redo()

{

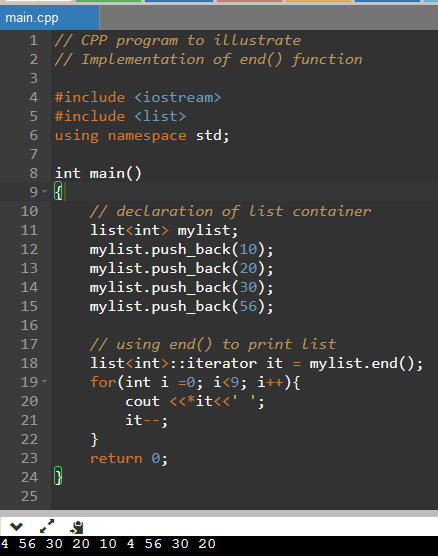
/\*

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

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

\*/

}



Source:

<https://www.geeksforgeeks.org/list-cpp-stl/>

<https://www.geeksforgeeks.org/iterator-library-c-stl/>

*DataLog*::DataLog()

{

    /\*

     \* TODO:  add the first state with 0

     \*/

    logList.push\_back(0);

    currentState = logList.begin();

}

*DataLog*::DataLog(const int &*data*)

{

    /\*

     \* TODO:  add the first state with data

     \*/

    logList.push\_back(*data*);

    currentState = logList.begin();

}

void *DataLog*::addCurrentState(int *number*)

{

    /\*

     \* TODO: Increase the value of current state by 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.

     \*/

    list<int>::*iterator* it = logList.end();

    it--;

    while(logList.size() > 1 && currentState != it){

        it--;

        logList.pop\_back();

    }

    logList.push\_back(\*currentState);

    currentState = std::next(currentState, 1);

}

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 = std::prev(currentState, 1);

    }

}

void *DataLog*::redo()

{

    /\*

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

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

     \*/

    list<int>::*iterator* it = logList.end();

    it--;

    if(currentState != it){

        currentState = std::next(currentState, 1);

    }

}

## Cau hoi 6 – chưa làm

Given the head of a doubly linked list, two positive integer a and b where a <= b. Reverse the nodes of the list from position a to position b and return the reversed list

Note: the position of the first node is 1. It is guaranteed that a and b are valid positions. You MUST NOT change the val attribute in each node.

struct ListNode {

int val;

ListNode \*left;

ListNode \*right;

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

};

Constraint:

1 <= list.length <= 10^5

0 <= node.val <= 5000

1 <= left <= right <= list.length

Example 1:

Input: list = {3, 4, 5, 6, 7} , a = 2, b = 4

Output: 3 6 5 4 7

Example 2:

Input: list = {8, 9, 10}, a = 1, b = 3

Output: 10 9 8

For example:

|  |  |  |
| --- | --- | --- |
| **Test** | **Input** | **Result** |
| 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; | 5  3 4 5 6 7  2 4 | 3 6 5 4 7 |
| 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; | 3  8 9 10  1 3 | 10 9 8 |

/\*

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) {

/To Do

}