BÁO CÁO ASSIGNMENT.3

Bùi Huỳnh Tây

MSSV: 24521589

29/04/2025

Cấu trúc dữ liệu và giải thuật
`IT003.P21.CTTN`

GV HDTH: Trần Đình Khang

Nội dung báo cáo Assignment.3

Mục lục

1 Basic problems

1.1 Kiểm tra biểu thức toán học (LaTeX)

1.1.1 Ý tưởng

Để giải quyết bài toán kiểm tra tính hợp lệ của biểu thức toán học trong LaTeX, chúng ta cần kiểm tra tính cân bằng của các dấu ngoặc. Một cách tiếp cận hiệu quả là sử dụng stack (ngăn xếp). Cụ thể, khi gặp dấu mở ngoặc, ta đẩy nó vào ngăn xếp; khi gặp dấu đóng ngoặc, ta kiểm tra xem nó có khớp với dấu mở ngoặc gần nhất trên ngăn xếp không.

1.1.2 Code mẫu

```
string s;
cin >> s;
/// "{"  , "}" , "(", ")" ,  "[", "]"

vector<char> type = {'(', ')', '{', '}', '[', ']'};

vector<char> ar;
for(int c : s){
    if(c == '(' || c == ')' || c == '{' || c == '}' || c == '[' || c
       == ']'){
        ar.pb(c);
    }
}


stack<char> st;
for(char c : ar){
    if(c == '(' || c == '{' || c == '['){
        st.push(c);
    }else{
        if(st.empty()){
            cout << 0;
            return 0;
        }
        char top = st.top();
        if((top == '(' && c == ')') || (top == '{' && c == '}') ||
           (top == '[' && c == ']')){
            st.pop();
        }else{
            cout << 0;
            return 0;
        }
    }
}

if(!st.empty()){
    cout << 0;
    return 0;
}

cout << 1;
```

1.2 LinkedList-Insertion

1.2.1 Ý tưởng

1. Khởi tạo con trỏ `res` để giữ địa chỉ node đầu của danh sách.
2. Khởi tạo con trỏ `pre_head` để lưu node trước vị trí cần chèn.
3. Duyệt danh sách liên kết từ đầu:
   - Nếu giá trị của node hiện tại nhỏ hơn `x`, tiếp tục đi tới node tiếp theo.
   - Đồng thời cập nhật `pre_head` là node trước đó.
4. Sau khi duyệt:
   - Tạo node mới chứa giá trị `x`.
   - Trỏ `newNode->next` tới `head` (node lớn hơn hoặc bằng `x`).
   - Nếu `pre_head == NULL` (tức cần chèn đầu danh sách) thì gán `res = newNode`.
   - Ngược lại, gán `pre_head->next = newNode`.
5. Trả về con trỏ `res` (node đầu tiên của danh sách mới).

Trả về res (đầu danh sách mới).

1.2.2 Code mẫu

```
SinglyLinkedListNode* res = head;
SinglyLinkedListNode* pre_head = nullptr;

while(head != NULL && head->data < x) {
    pre_head = head;
    head = head->next;
}

SinglyLinkedListNode* newNode = new SinglyLinkedListNode(x);
newNode->next = head;

if(pre_head == NULL) res = newNode;
else pre_head->next = newNode;

return res;
```

1.3 LinkedList-NhapDaThuc

1.3.1 Ý tưởng

1. Nhập số lượng đơn thức n và n cặp hệ số - số mũ theo thứ tự giảm dần của số mũ.
2. Duyệt qua từng đơn thức để xây dựng biểu diễn dạng chuẩn của đa thức:
   - Bỏ qua những đơn thức có hệ số bằng 0.
   - Với đơn thức đầu tiên:

- Nếu hệ số là âm ⇒ in dấu `-`.
- Nếu hệ số là dương ⇒ in trực tiếp (không thêm dấu `+`).

- Với các đơn thức tiếp theo:
  - Nếu hệ số dương ⇒ in dấu `+`.
  - Nếu hệ số âm ⇒ in dấu `-`.
- Xử lý riêng với hệ số bằng 1 hoặc -1:
  - Nếu mũ $= 0$: in hệ số đầy đủ.
  - Nếu mũ $= 1$: in `x` (bỏ mũ).
  - Nếu mũ > 1: in dạng `x^k`.

3. Nhập giá trị x, tính giá trị $f(x)$ theo công thức:

$$f(x) = \sum_{i=1}^{n} a_i \cdot x^{k_i}$$

1.3.2 Code mẫu

```
void Nhap(DATHUC& B, double heso, int somu) {
    DONTHUC* p = new DONTHUC(heso, somu);
    Node* q = new Node(p);
    if (B.head == nullptr) {
        B.head = q;
        B.tail = q;
    } else {
        B.tail->next = q;
        B.tail = q;
    }
}

double TinhDaThuc(DATHUC B, double x) {
    double res = 0;
    Node* p = B.head;
    while (p != nullptr) {
        res += p->data->heso * pow(x, p->data->somu);
        p = p->next;
    }
    return res;
}

void Xuat(DATHUC B) {
    Node* p = B.head;
    bool first = true;

    while (p != nullptr) {
        double heso = p->data->heso;
        int somu = p->data->somu;

        if (heso != 0) {
            if (!first) {
                if (heso > 0) cout << "+";
            }

            if (heso == -1 && somu != 0) {
```

```
                cout << "-";
            } else if (heso != 1 || somu == 0) {
                cout << heso;
            }

            if (somu > 0) {
                cout << "x";
                if (somu > 1) {
                    cout << "^" << somu;
                }
            }

            first = false;
        }

        p = p->next;
    }

    if (first) {
        cout << "0";
    }
}
```

1.4 LinkedList-Reverse

1.4.1 Ý tưởng

1. Duyệt qua danh sách liên kết và thực hiện đảo ngược bằng kỹ thuật con trỏ ba bước:
    - **prev**: con trỏ lưu node trước đó (ban đầu là `nullptr`).
    - **current**: con trỏ hiện tại, bắt đầu từ `head`.
    - **next**: con trỏ tạm thời để lưu node tiếp theo.
2. Tại mỗi bước:
    - Lưu node tiếp theo: `next = current->next`.
    - Đảo chiều liên kết: `current->next = prev`.
    - Cập nhật prev và current cho bước tiếp theo.
3. Khi kết thúc vòng lặp, prev chính là node mới đứng đầu danh sách. Gán `llist->head = prev`.
4. In danh sách bằng cách duyệt từ `head` đến cuối danh sách, mỗi phần tử cách nhau bởi dấu cách.

1.4.2 Code mẫu

```
void reverseLinkedList(SinglyLinkedList* llist) {
    SinglyLinkedListNode* prev = nullptr;
    SinglyLinkedListNode* current = llist->head;
    SinglyLinkedListNode* next = nullptr;
```

```
    while (current != nullptr) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    llist->head = prev;
}

void printLinkedList(SinglyLinkedList* llist) {
    SinglyLinkedListNode* current = llist->head;
    while (current != nullptr) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

void insert_node(SinglyLinkedList* llist, int node_data) {
    SinglyLinkedListNode* node = new SinglyLinkedListNode(node_data);

    if (!llist->head) {
        llist->head = node;
    } else {
        llist->tail->next = node;
    }

    llist->tail = node;
}
```

1.5 Dec to Bin

1.5.1 Ý tưởng

1. Áp dụng phương pháp chia lấy dư để chuyển đổi số ở hệ cơ số 10 sang hệ cơ số 2:
   - Trong khi $x > 0$:
     - Lấy phần dư khi chia x cho 2 và lưu lại vào chuỗi kết quả.
     - Cập nhật $x = \lfloor x/2 \rfloor$.
   - Các phần dư được lưu theo thứ tự ngược (phần dư đầu là bit thấp nhất), do đó sau khi thu được kết quả cần đảo ngược chuỗi.
2. Độ phức tạp thuật toán là $O(\log_2 x)$, vì mỗi lần chia đôi số x.

1.5.2 Code mẫu

```
void input(){
    cin >> n;
}

void solve(){
    input();
```

```
    while(n){
        s += char(n % 2 + '0');
        n /= 2;
    }
    reverse(s.begin(), s.end());
    cout << s;
}
```

1.6 LinkedList: MergetwoSortedLinkedList

1.6.1 Ý tưởng

1. Khởi tạo một node giả `dummy` để làm node đầu của danh sách kết quả.
2. Duyệt song song hai danh sách liên kết `head_list1` và `head_list2`:
   - So sánh giá trị tại hai node hiện tại.
   - Gán node có giá trị nhỏ hơn vào danh sách kết quả (thông qua `tail`), sau đó di chuyển con trỏ tương ứng sang node kế tiếp.
3. Khi một trong hai danh sách hết phần tử:
   - Nối phần còn lại của danh sách kia vào cuối danh sách kết quả.
4. Trả về danh sách kết quả bắt đầu từ `dummy.next`.

Độ phức tạp: $O(n+m)$ với n và m là độ dài hai danh sách đầu vào.

1.6.2 Code mẫu

```
SinglyLinkedListNode dummy(0);
SinglyLinkedListNode* tail = &dummy;


while (head_list1 != nullptr && head_list2 != nullptr) {
    if (head_list1->data <= head_list2->data) {
        tail->next = head_list1;
        head_list1 = head_list1->next;
    } else {
        tail->next = head_list2;
        head_list2 = head_list2->next;
    }
    tail = tail->next;
}


if (head_list1 != nullptr) {
    tail->next = head_list1;
} else {
    tail->next = head_list2;
}

return dummy.next;
```

1.7 Tree: Preorder Traversal (NLR) - Duyệt cây BST theo NLR

1.7.1 Ý tưởng

1. Bắt đầu từ node gốc (`root`).
2. In giá trị của node hiện tại (N - Node).
3. Đệ quy duyệt cây con bên trái (L - Left).
4. Đệ quy duyệt cây con bên phải (R - Right).

Đặc điểm: thứ tự in ra các node là: gốc → trái → phải.
Độ phức tạp: $O(n)$ với n là số node trong cây (vì mỗi node được thăm đúng một lần).

1.7.2 Code mẫu

```
Node* cur = root;

cout << cur->data << ' ';

if(cur->left){
    preOrder(cur->left);
}
if(cur->right){
    preOrder(cur->right);
}
return;
```

1.8 Tree: Preorder Traversal (NLR) II - Duyệt cây BST theo NLR không đệ quy

1.8.1 Ý tưởng

1. Sử dụng một **ngăn xếp (stack)** để mô phỏng hoạt động đệ quy.
2. Ban đầu, đẩy node gốc `root` vào stack.
3. Trong khi stack còn phần tử:
   - Lấy phần tử trên cùng của stack ra (node hiện tại) và in giá trị của node.
   - Nếu node hiện tại có cây con phải, đẩy node phải vào stack.
   - Nếu node hiện tại có cây con trái, đẩy node trái vào stack.
4. **Lưu ý:** đẩy cây con phải trước, sau đó mới đẩy cây con trái để đảm bảo thứ tự NLR khi pop.

Thứ tự duyệt: Node → Left → Right.
Độ phức tạp: $O(n)$ với n là số node trong cây.

1.8.2 Code mẫu

```
stack <Node*> st;

st.push(root);
Node* node = root;

while(st.size()){
    node = st.top();
    st.pop();

    cout << node->data << ' ';

    if(node->right) st.push(node->right);
    if(node->left) st.push(node->left);
}
```

1.9 LinkedList: Tìm nút chung của 2 danh sách liên kết đơn

1.9.1 Ý tưởng

1. **Tính độ dài của hai danh sách liên kết**:
   - Duyệt qua từng nút của danh sách liên kết thứ nhất (`head1`) để đếm số lượng nút, lưu vào biến `len1`.
   - Duyệt qua từng nút của danh sách liên kết thứ hai (`head2`) để đếm số lượng nút, lưu vào biến `len2`.
2. **Điều chỉnh vị trí bắt đầu của danh sách dài hơn**:
   - Nếu `len1` > `len2`, di chuyển con trỏ `temp1` (ban đầu trỏ đến `head1`) về phía trước `diff = len1 - len2` nút. Điều này đảm bảo rằng cả hai con trỏ sẽ bắt đầu duyệt từ vị trí cách đều nút cuối cùng.
   - Ngược lại, nếu `len2` > `len1`, di chuyển con trỏ `temp2` (ban đầu trỏ đến `head2`) về phía trước `diff = len2 - len1` nút.
3. **Duyệt đồng thời và so sánh**:
   - Duyệt đồng thời từ vị trí hiện tại của `temp1` và `temp2`.
   - Tại mỗi bước, so sánh địa chỉ của hai nút hiện tại:
     - Nếu `temp1 == temp2`, trả về nút này vì đây là nút chung.
     - Nếu không, di chuyển cả hai con trỏ đến nút tiếp theo.
   - Nếu duyệt hết danh sách mà không tìm thấy nút chung, trả về `nullptr`.

1.9.2 Code mẫu

```
int len1 = 0;
SinglyLinkedListNode* temp1 = head1;
while (temp1 != nullptr) {
    len1++;
```

```
    temp1 = temp1->next;
}


int len2 = 0;
SinglyLinkedListNode* temp2 = head2;
while (temp2 != nullptr) {
    len2++;
    temp2 = temp2->next;
}


temp1 = head1;
temp2 = head2;
if (len1 > len2) {
    int diff = len1 - len2;
    while (diff--) {
        temp1 = temp1->next;
    }
} else {
    int diff = len2 - len1;
    while (diff--) {
        temp2 = temp2->next;
    }
}


while (temp1 != nullptr && temp2 != nullptr) {
    if (temp1 == temp2) {
        return temp1;
    }
    temp1 = temp1->next;
    temp2 = temp2->next;
}

return nullptr;
```

1.10 Binary Search Tree: Nút tổ tiên thấp nhất (tiếng Việt)

1.10.1 Ý tưởng

- **Tính chất BST**: Mọi nút ở cây con trái có giá trị nhỏ hơn nút hiện tại, và mọi nút ở cây con phải có giá trị lớn hơn nút hiện tại.
- **Điều kiện LCA**:
  - Nếu cả v_1 và v_2 nhỏ hơn giá trị nút hiện tại, LCA nằm ở cây con trái.
  - Nếu cả v_1 và v_2 lớn hơn giá trị nút hiện tại, LCA nằm ở cây con phải.
  - Nếu v_1 và v_2 nằm về hai phía khác nhau của nút hiện tại (một nhỏ hơn, một lớn hơn hoặc bằng), nút hiện tại chính là LCA.
- **Thuật toán**:
  1. Duyệt từ gốc (`root`) của BST.

2. Di chuyển sang trái nếu cả v_1 và v_2 nhỏ hơn `root->data`.
3. Di chuyển sang phải nếu cả v_1 và v_2 lớn hơn `root->data`.
4. Nếu không thỏa mãn hai điều kiện trên, dừng lại và trả về `root` vì đây là LCA.

- **Độ phức tạp**:
  - Thời gian: $O(h)$, với h là chiều cao của BST (tối đa $O(n)$ nếu cây lệch).
  - Không gian: $O(1)$, không sử dụng đệ quy hoặc cấu trúc dữ liệu phụ.

Công thức kiểm tra LCA:

$$(v_1 - \texttt{root->data}) \times (v_2 - \texttt{root->data}) \leq 0$$

1.10.2 Code mẫu

```
while(root){
    while(v1 < root->data && v2 < root->data) root = root->left;
    while(v1 > root->data && v2 > root->data) root = root->right;
    if((v1 - root->data) * (v2 - root->data) <= 0) return root;
}
```

2 Advance problems

2.1 Kiểm Kê 2

2.1.1 Ý tưởng

1. **Sắp xếp đầu vào**:
   - Dùng QuickSort tùy chỉnh sắp xếp các mã hàng
   - Ưu tiên độ dài chuỗi tăng dần trước
   - Nếu cùng độ dài thì so sánh từ điển
2. **Đếm tần suất**:
   - Duyệt mảng đã sắp xếp
   - Đếm số lần xuất hiện liên tiếp của mỗi mã
3. **Sắp xếp kết quả**:
   - Sắp xếp theo số lượng giảm dần
   - Nếu cùng số lượng: ưu tiên mã ngắn hơn
   - Nếu cùng độ dài: sắp xếp từ điển tăng dần

2.1.2 Code mẫu

```
bool cmp(string a, string b){
    if(a.size() == b.size()){
        return a < b;
    }
    return a.size() < b.size();
}
void quick_string(vector<string> &s, int left, int right){
    if(left >= right) return;
    string pivot = s[(left + right) / 2];
    int i = left, j = right;
    while(i <= j){
        while(cmp(s[i], pivot)) i++;
        while(cmp(pivot, s[j])) j--;
        if(i <= j){
            swap(s[i], s[j]);
            i++;
            j--;
        }
    }
    quick_string(s, left, j);
    quick_string(s, i, right);
}


bool cmp2(pair<int, string> a, pair<int, string> b){
    if(a.fi == b.fi){
        if(a.se.size() == b.se.size()){
            return a.se < b.se;
        }
        return a.se.size() < b.se.size();
    }
    return a.fi > b.fi;
}
void quick_string2(vector< pair<int, string> > &s, int left, int
   right){
    if(left >= right) return;
    pair<int, string> pivot = s[(left + right) / 2];
    int i = left, j = right;
    while(i <= j){
        while(cmp2(s[i], pivot)) i++;
        while(cmp2(pivot, s[j])) j--;
        if(i <= j){
            swap(s[i], s[j]);
            i++;
            j--;
        }
    }
    quick_string2(s, left, j);
    quick_string2(s, i, right);
}

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    file("file");
```

```
56
57      int n;
58      cin >> n;
59      vector<string> s(n);
60      for(int i = 0; i < n; i++){
61          cin >> s[i];
62      }
63
64      quick_string(s, 0, n - 1);
65
66
67      vector< pair<int, string> > a;
68      for(int i = 0; i < n; i++){
69          if(i == 0 || s[i] != s[i - 1]){
70              a.push_back(std::make_pair(1, s[i]));
71          }else{
72              a.back().fi++;
73          }
74      }
75
76
77      quick_string2(a, 0, a.size() - 1);
78
79      for(int i = 0; i < a.size(); i++){
80          cout << a[i].se << " " << a[i].fi << "\n";
81      }
82      cout << "\n";
83      return 0;
84  }
```

2.2 Binary Search 2

2.2.1 Ý tưởng

- **Ý tưởng chính:**
  - Bài toán yêu cầu tìm vị trí đầu/cuối của giá trị y trong mảng A với số lượng truy vấn lớn ($Q \leq 5 \times 10^5$).
  - Cần sử dụng phương pháp **tìm kiếm nhị phân (Binary Search)** để đảm bảo hiệu suất.
- **Các bước thực hiện:**
  a. **Chuẩn bị dữ liệu:**
    * Tạo mảng S gồm các cặp (giá trị, vị trí) từ mảng A ban đầu.
    * Sắp xếp mảng S theo giá trị tăng dần bằng thuật toán **QuickSort** tùy chỉnh.

  b. **Xử lý truy vấn:**
    * Với mỗi truy vấn ($type$, x, y):
    * Nếu $type = 1$: Tìm **vị trí đầu tiên** của y trong mảng đã sắp xếp.
    * Khi tìm thấy y, tiếp tục tìm kiếm phía **bên trái** để kiểm tra còn xuất hiện trước đó không.

    - Nếu $type = 2$: Tìm **vị trí cuối cùng** của y trong mảng đã sắp xếp.
    - Khi tìm thấy y, tiếp tục tìm kiếm phía **bên phải** để kiểm tra còn xuất hiện sau đó không.
    - Nếu không tìm thấy y, trả về -1.

- **Tối ưu hóa:**
  - Sắp xếp trước một lần với độ phức $O(N \log N)$.
  - Mỗi truy vấn được xử lý trong $O(\log N)$ nhờ tìm kiếm nhị phân.
  - Tắt đồng bộ hóa I/O để tăng tốc độ nhập xuất dữ liệu.

- **Độ phức tạp:**
  - Tiền xử lý: $O(N \log N)$ cho việc sắp xếp.
  - Xử lý truy vấn: $O(Q \log N)$ cho Q truy vấn.
  - Tổng cộng: $O((N + Q) \log N)$ - đủ hiệu quả với giới hạn đề bài.

- **Lưu ý:**
  - Cần lưu cả **vị trí gốc** khi sắp xếp để trả về kết quả đúng.
  - So sánh giá trị dạng **số nguyên** để đảm bảo thứ tự đúng.
  - Xuất kết quả bằng '\n' thay vì `endl` để tránh quá thời gian.

2.2.2 Code mẫu

```
template<class X> bool cmp(X a, X b){

    if(a.fi == b.fi)
        return a.se < b.se;
    return a.fi < b.fi;
}

template<class X> void quick_string(vector<X> &s, int left, int
   right, bool (*cmp)(X, X)){
    if(left >= right) return;
    X pivot = s[(left + right) / 2];
    int i = left, j = right;
    while(i <= j){
        while(cmp(s[i], pivot)) i++;
        while(cmp(pivot, s[j])) j--;
        if(i <= j){
            swap(s[i], s[j]);
            i++;
            j--;
        }
    }
    quick_string(s, left, j, cmp);
    quick_string(s, i, right, cmp);
}
```

```cpp

int main(){
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    file("file");

    int n, q;
    cin >> n >> q;

    vector< pair<int,int> > s(n);
    for(int i = 0; i < n; i++){
        cin >> s[i].fi;
        s[i].se = i + 1;
    }

    quick_string(s, 0, n - 1, cmp);


    for(int i = 1; i <= q; i++){
        string type;
        int x;
        int y;
        cin >> type >> x >> y;


        int l = 0, r = n - 1;
        int ans = -1;
        while(l <= r){
            int mid = (l + r) / 2;
            if(s[mid].fi == y){
                ans = s[mid].se;

                if(x == 1) r = mid - 1;
                else l = mid + 1;
            }
            else if(s[mid].fi < y){
                l = mid + 1;
            }
            else{
                r = mid - 1;
            }
        }

        cout << ans << '\n';
    }
    return 0;
}
```

2.3 khangtd.XepHang và khangtd.XepHang2

2.3.1 Ý tưởng

- Sử dụng **Danh sách liên kết đôi (Doubly Linked List)** để quản lý dãy số:
  - **Thêm số**: Thêm vào cuối danh sách (nếu số chưa tồn tại, tạo node mới).

- **Xóa số**: Tìm và xóa node chứa số đó, xử lý các trường hợp (đầu, cuối, giữa danh sách).
- **Xử lý truy vấn**: Với mỗi truy vấn, xóa số x khỏi danh sách, rồi thêm x vào cuối, in ra số ở đầu danh sách.
- **Tối ưu**: Dùng mảng `save` để lưu trữ node đã tạo, tránh tạo lại node cho cùng giá trị.

- **Độ phức tạp:**
  - Thêm/xóa: $O(1)$ mỗi thao tác.
  - Tổng: $O(N + M)$ cho N số ban đầu và M truy vấn.

2.3.2 Code mẫu

```
struct Node {
    int data;
    Node* prev;
    Node* next;
};

Node* save[1000000];

struct DoublyLinkedList {
    Node* head;
    Node* tail;

    DoublyLinkedList() {
        head = tail = nullptr;
    }


    Node* createNode(int value) {
        Node* newNode = new Node;
        newNode->data = value;
        newNode->prev = nullptr;
        newNode->next = nullptr;
        return newNode;
    }


    void add(int value) {

        if(save[value] == nullptr) {
            save[value] = createNode(value);
        }

        Node* newNode = save[value];

        if (head == nullptr) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
```

```
    }


    void deleteNode(int value) {
        Node* curr = save[value];

        if (curr == nullptr) return;


        if (curr == head) {
            head = head->next;
            if (head != nullptr) head->prev = nullptr;
            else tail = nullptr;
        }

        else if (curr == tail) {
            tail = tail->prev;
            tail->next = nullptr;
        }

        else {
            curr->prev->next = curr->next;
            curr->next->prev = curr->prev;
        }
        delete curr;
        save[value] = nullptr;
    }


    void print() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " <-> ";
            temp = temp->next;
        }
        cout << "NULL\n";
    }


    void clear() {
        while (head != nullptr) {
            Node* temp = head;
            head = head->next;
            delete temp;
        }
        tail = nullptr;
    }

    ~DoublyLinkedList() {
        clear();
    }
};


int main() {
    DoublyLinkedList list;

    int n, m;
```

```
    cin >> n >> m;
    for(int i = n; i >= 1; i--){
        list.add(i);
    }


    for(int i = 1; i <= m; i++){
        int x;
        cin >> x;
        list.deleteNode(x);
        list.add(x);


        cout << list.head->data << ' ';
    }

    return 0;
}
```

2.4 khangtd.DetectVirusin2D

2.4.1 Ý tưởng

- Sử dụng **Rolling Hash** để kiểm tra sự tồn tại của xâu:
  - Tính **hash** cho tất cả xâu con độ dài từ 2 trở lên trong lưới $N \times M$ (theo hàng và cột) bằng cách dùng hai modulo (`MOD1`, `MOD2`) để giảm xung đột.
  - Lưu các cặp hash vào `unordered_set` với hàm băm tùy chỉnh.
  - Với mỗi truy vấn, tính hash của xâu truy vấn và kiểm tra xem có trong tập hash không, trả về '1' nếu có, '0' nếu không.
- **Độ phức tạp:**
  - Tiền xử lý: $O(NM \cdot \min(N, M))$ để tính hash.
  - Truy vấn: $O(Q \cdot K)$ với K là độ dài xâu truy vấn.

2.4.2 Code mẫu

```
const int MOD1 = 1e9 + 7;
const int MOD2 = 1e9 + 9;
const int BASE = 37;

int n, m, q;


struct pair_hash {
    template <class T1, class T2>
    size_t operator()(const pair<T1, T2>& p) const {
        auto hash1 = hash<T1>{}(p.first);
        auto hash2 = hash<T2>{}(p.second);
        return hash1 ^ (hash2 << 1);
    }
};
```

```cpp

void addHashes(vector<vector<char>> &a, unordered_set<pair<long long, long long>, pair_hash> &hashSet) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            long long hash1 = 0, hash2 = 0, power1 = 1, power2 = 1;
            for (int r = j; r < j + 10 && r <= m; r++) {
                hash1 = (hash1 * BASE + (a[i][r] - 'a' + 1)) % MOD1;
                hash2 = (hash2 * BASE + (a[i][r] - 'a' + 1)) % MOD2;
                if (r - j + 1 >= 2) {
                    hashSet.insert({hash1, hash2});
                }
                power1 = (power1 * BASE) % MOD1;
                power2 = (power2 * BASE) % MOD2;
            }
        }
    }

    for (int j = 1; j <= m; j++) {
        for (int i = 1; i <= n; i++) {
            long long hash1 = 0, hash2 = 0, power1 = 1, power2 = 1;
            for (int r = i; r < i + 10 && r <= n; r++) {
                hash1 = (hash1 * BASE + (a[r][j] - 'a' + 1)) % MOD1;
                hash2 = (hash2 * BASE + (a[r][j] - 'a' + 1)) % MOD2;
                if (r - i + 1 >= 2) {
                    hashSet.insert({hash1, hash2});
                }
                power1 = (power1 * BASE) % MOD1;
                power2 = (power2 * BASE) % MOD2;
            }
        }
    }
}

pair<long long, long long> computeHash(const string &s) {
    long long hash1 = 0, hash2 = 0;
    for (char c : s) {
        hash1 = (hash1 * BASE + (c - 'a' + 1)) % MOD1;
        hash2 = (hash2 * BASE + (c - 'a' + 1)) % MOD2;
    }
    return {hash1, hash2};
}

int main() {
    ios_base::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    file("file");

    cin >> n >> m >> q;

    vector<vector<char>> a(n + 1, vector<char>(m + 1));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            cin >> a[i][j];
        }
    }

```

```
    unordered_set<pair<long long, long long>, pair_hash> hashSet;
    addHashes(a, hashSet);

    string result = "";
    for (int i = 0; i < q; i++) {
        string x;
        cin >> x;
        auto queryHash = computeHash(x);
        result += (hashSet.count(queryHash) ? '1' : '0');
    }
    cout << result;
    return 0;
}
```