Báo cáo Assignment 03.01

Võ Lê Ngọc Thịnh

Ngày 4 tháng 4 năm 2025

Mục lục

1 Bài tập 04: So sánh ưu nhược điểm các Linked List

1.1 So sánh Linked List có và không có con trỏ Tail

- **Linked List có con trỏ Tail**

  Ưu điểm:

  - **Truy cập nhanh đến node cuối:** Nhờ con trỏ *Tail* luôn trỏ đến node cuối, việc truy xuất phần tử cuối cùng chỉ cần một bước thay vì phải duyệt toàn bộ danh sách.
  - **Thêm phần tử vào cuối danh sách hiệu quả:** Khi thêm một node mới, chỉ cần cập nhật con trỏ *next* của *Tail* để liên kết với node mới và di chuyển *Tail* đến node đó, không cần duyệt qua danh sách.
  - **Tối ưu cho các thao tác ở cuối danh sách:** Các thao tác thường dùng như thêm phần tử vào cuối, nối hai danh sách hoặc xóa node cuối đều nhanh hơn vì luôn có sẵn con trỏ Tail để xử lý chính xác node cuối.

  Khuyết điểm:

  - **Tốn thêm bộ nhớ:** Mỗi danh sách liên kết có *Tail* phải cấp phát bộ nhớ cho thêm một con trỏ để lưu địa chỉ node cuối cùng.
  - **Khó quản lý đồng bộ:** Việc duy trì tính nhất quán của con trỏ *Tail* đòi hỏi sự cẩn thận khi thực hiện các thao tác thay đổi cuối danh sách. Nếu quên cập nhật *Tail* khi xóa hoặc thêm node, con trỏ sẽ trỏ đến vùng nhớ không hợp lệ, dẫn đến lỗi truy cập bộ nhớ hoặc rò rỉ dữ liệu.

- **Linked List không có con trỏ Tail**

  Ưu điểm:

  - **Tiết kiệm bộ nhớ:** Danh sách liên kết không có con trỏ *Tail* chỉ cần lưu trữ con trỏ *Head*, giúp tiết kiệm không gian bộ nhớ khi triển khai nhiều danh sách đồng thời.
  - **Cài đặt đơn giản:** Không cần việc cập nhật đồng bộ con trỏ *Tail* khi thực hiện thao tác thêm/xóa ở cuối danh sách, giúp giảm thiểu bug tiềm ẩn và phức tạp trong code.

  Khuyết điểm:

  - **Truy cập node cuối chậm:** Để truy cập được node cuối của danh sách, ta cần phải duyệt qua tất cả các node của danh sách. Điều này làm giảm hiệu suất chương trình khi danh sách có nhiều phần tử.
  - **Thao tác thêm/xóa ở cuối danh sách kém hiệu quả:** Mỗi khi thêm/xóa một node ở cuối danh sách không có con trỏ *Tail*, ta cần phải duyệt qua toàn bộ danh sách để tìm phần tử cuối trước khi thực hiện thao tác thêm/xóa.

1.2 So sánh Linked List có và không có Header Node

- **Linked List có Header Node**

  Ưu điểm:

  - **Đơn giản hóa thao tác thêm/xóa:** Header Node tạo ra một điểm bắt đầu cố định, giúp thao tác thêm/xóa ở đầu không cần phải xử lý trường hợp đặc biệt khi danh sách rỗng.
  - **Dễ quản lý thông tin danh sách:** Header Node là nơi thuận tiện để lưu trữ metadata như số lượng phân tử, kích thước hoặc các thông tin liên quan mà không cần phải lưu trữ riêng biệt.

  Khuyết điểm:

  - **Tốn thêm bộ nhớ:** Mỗi danh sách phải cấp phát thêm bộ nhớ cho không chứa dữ liệu thật, điều này là đáng kể khi số lượng danh sách là rất lớn.
  - **Tăng độ phức tạp của cấu trúc:** Việc thêm một node đặc biệt làm tăng sự phức tạp của cấu trúc dữ liệu, đòi hỏi logic để xử lý node này khác biệt so với những node còn lại trong danh sách.

- **Linked List không có Header Node**

  Ưu điểm:

  - **Tiết kiệm bộ nhớ:** Không cần cấp phát thêm bộ nhớ cho Header Node, mỗi node trong danh sách đều chứa dữ liệu thực, tối ưu hóa việc sử dụng bộ nhớ.
  - **Dễ triển khai:** Cấu trúc cơ bản hơn và dễ triển khai.

  Khuyết điểm:

  - **Khó xử lý các trường hợp đặc biệt:** Cần code xử lý riêng cho trường hợp danh sách rỗng hoặc thêm/xóa ở đầu danh sách, làm tăng độ phức tạp và khả năng xảy ra bug.

2 Bài tập 05: Chuyển danh sách sinh viên từ mảng động

Khai báo cấu trúc SinhVien và mảng động lưu các đối tượng SinhVien:

```
struct SinhVien{
    string HoTen;
    string MSSV;
    int Tuoi;
    SinhVien (string _HoTen="", string _MSSV="",int _Tuoi=0){
        HoTen = _HoTen;
        MSSV = _MSSV;
        Tuoi = _Tuoi;
    }
};

int size = 100;
SinhVien* arr = new SinhVien[size];
```

2.1 Chuyển sang danh sách liên kết đơn không có con trỏ cuối và ngược lại

Khai báo cấu trúc danh sách:

```
struct SLL_Node{
    SinhVien data;
    SLL_Node* next;

    SLL_Node (SinhVien data){
        this->data = data;
        this->next = nullptr;
    }
};

struct SLL{
    SLL_Node* pHead;

    SLL (){
        this->pHead = nullptr;
    }
};
```

Chuyển từ mảng động sang danh sách liên kết đơn:

```
SLL* ArrayToSLL(SinhVien* arr, int size){
    SLL* newList = new SLL();
    if (size <= 0)
        return newList;

    newList->pHead = new SLL_Node(arr[0])
```

```
    SLL_Node* current = newList->pHead;
    for (int i=1;i< size;i++ ){
        current->next = new SLL_Node(arr[i]);
        current = newNode;
    }

    return newList;
}
```

Chuyển từ danh sách liên kết đơn sang mảng động:

```
SinhVien* SLLToArray(SLL* list){
    int size = 0;
    SLL_Node* current = list->pHead;
    while (current != nullptr){
        size++;
        current = current->next;
    }

    SinhVien* newArr = new SinhVien[size];
    SLL_Node* current = list->pHead;
    for (int i=0;i< size;i++ ){
        newArr[i] = current->data;
        current = current->next;
    }
    return newArr;
}
```

2.2 Chuyển sang danh sách liên kết đơn luôn có con trỏ cuối và ngược lại

Khai báo cấu trúc danh sách:

```
struct SLL{
    SLL_Node* pHead;
    SLL_Node* pTail;

    SLL (){
        this->pHead = nullptr;
        this->pTail = nullptr;
    }
};
```

Chuyển từ mảng động sang danh sách liên kết đơn:

```
SLL* ArrayToSLL(SinhVien* arr, int size){
    SLL* newList = new SLL();

    for (int i=0;i< size;i++ ){
        SLL_Node* newNode = new SLL_Node(arr[i]);
        if (newList->pHead == nullptr)
            newList->pHead = newList->pTail = newNode;
```

```
        else{
            newList->pTail->next = newNode;
            newList->pTail = newNode;
        }
    }

    return newList;
}
```

Chuyển từ danh sách liên kết đơn sang mảng động:

```
SinhVien* SLLToArray(SLL* list){
    int size = 0;
    SLL_Node* current = list->pHead;
    while (current != nullptr){
        size++;
        current = current->next;
    }

    SinhVien* newArr = new SinhVien[size];
    current = list->pHead;
    for (int i=0;i< size;i++ ){
        newArr[i] = current->data;
        current = current->next;
    }
    return newArr;
}
```

2.3 Chuyển sang danh sách liên kết kép và ngược lại

Khai báo cấu trúc danh sách:

```
struct DLL_Node{
    SinhVien data;
    DLL_Node* prev;
    DLL_Node* next;

    DLL_Node (SinhVien data){
        this->data = data;
        this->prev = nullptr;
        this->next = nullptr;
    }
};

struct DLL{
    DLL_Node* pHead;
    DLL_Node* pTail;

    DLL (){
        this->pHead = nullptr;
        this->pTail = nullptr;
    }
```

```
};
```

Chuyển từ mảng động sang danh sách liên kết kép:

```
DLL* ArrayToDLL(SinhVien* arr, int size){
    DLL* newList = new DLL();
    for (int i=0;i< size;i++ ){
        DLL_Node* newNode = new DLL_Node(arr[i]);
        if (newList->pHead == nullptr)
            newList->pHead = newList->pTail = newNode;
        else{
            newList->pTail->next = newNode;
            newNode->prev = newList->pTail;
            newList->pTail = newNode;
        }
    }
    return newList;
}
```

Chuyển từ danh sách liên kết kép sang mảng động:

```
SinhVien* DLLToArray(DLL* list){
    int size = 0;
    DLL_Node* current = list->pHead;
    while (current != nullptr){
        size++;
        current = current->next;
    }
    SinhVien* newArr = new SinhVien[size];
    current = list->pHead;
    for (int i=0;i< size;i++ ){
        newArr[i] = current->data;
        current = current->next;
    }
    return newArr;
}
```

2.4 Chuyển sang danh sách liên kết vòng và ngược lại:

Khai báo cấu trúc danh sách:

```
struct CLL_Node{
    SinhVien data;
    CLL_Node* next;
    CLL_Node (SinhVien data){
        this->data = data;
        this->next = nullptr;
    }
};

struct CLL{
    CLL_Node* pHead;
    CLL_Node* pTail;
```

```
    CLL (){
        this->pHead = nullptr;
        this->pTail = nullptr;
    }
};
```

Chuyển từ mảng động sang danh sách liên kết vòng:

```
CLL* ArrayToCLL(SinhVien* arr, int size){
    CLL* newList = new CLL();
    for (int i=0;i< size;i++ ){
        CLL_Node* newNode = new CLL_Node(arr[i]);
        if (newList->pHead == nullptr){
            newNode->next = newNode;
            newList->pHead = newList->pTail = newNode;
        }
        else{
            newList->pTail->next = newNode;
            newList->pTail = newNode;
            newList->pTail = newList->pHead;
        }
    }
    return newList;
}
```

Chuyển từ danh sách liên kết vòng sang mảng động:

```
SinhVien* CLLToArray(CLL* list){
    int size = 0;
    CLL_Node* current = list->pHead;
    do{
        if (current == nullptr) break;
        size++;
        current = current->next;
    } while (current != list->pHead);

    SinhVien* newArr = new SinhVien[size];
    for (int i=0;i< size;i++ ){
        newArr[i] = current->data;
        current = current->next;
    }
    return newArr;
}
```

2.5 Chuyển sang danh sách liên kết kép vòng và ngược lại:

Khai báo cấu trúc danh sách:

```
struct CDLL_Node{
    SinhVien data;
    CDLL_Node* next;
    CDLL_Node* prev;
    CDLL_Node (SinhVien data){
```

```
        this->data = data;
        this->next = nullptr;
        this->prev = nullptr;
    }
};

struct CDLL{
    CDLL_Node* pHead;
    CDLL_Node* pTail;
    CDLL (){
        this->pHead = nullptr;
        this->pTail = nullptr;
    }
};
```

Chuyển từ mảng động sang danh sách liên kết kép vòng:

```
CDLL* ArrayToCDLL(SinhVien* arr, int size){
    CDLL* newList = new CDLL();
    if (size == 0)  return newList;

    newList->pHead = new CDLL_Node(arr[0]);
    newList->pHead->next = newList->pHead;
    newList->pHead->prev = newList->pHead;
    newList->pTail = newList->pHead;

    for (int i=0;i< size;i++ ){
        CDLL_Node* newNode = new CDLL_Node(arr[i]);
        newNode->next = newList->pHead;
        newNode->prev = newList->pTail;

        newList->pTail->next = newNode;
        newList->pHead->prev = newNode;

        newList->pTail = newNode;
    }
    return newList;
}
```

Chuyển từ danh sách liên kết kép vòng sang mảng động:

```
SinhVien* CDLLToArray(CDLL* list){
    int size = 0;
    CDLL_Node* current = list->pHead;
    do{
        if (current == nullptr) break;
        size++;
        current = current->next;
    } while (current != list->pHead);

    SinhVien* newArr = new SinhVien[size];
    for (int i=0;i< size;i++ ){
        newArr[i] = current->data;
```

```
        current = current->next;
    }
    return newArr;
}
```

3 Bài tập 06: Viết hàm thêm một phần tử vào các cấu trúc

3.1 Mảng động đang có thứ tự

```
struct Array{
    int* items;
    int capacity, n;
};

bool ExtendArray(Array &arr, int INC=101){
    for (;INC > 0;INC -= 10){
        if (realloc(arr.items, (arr.capacity + INC)*sizeof(arr.
           items[0])))
            break;
    }
    if (INC <= 0)   return false;
    arr.capacity += INC;
    return true;
}

bool InsertElement(Array &arr, int x){
    if (arr.n == arr.capacity)
        if (!ExtendArray(arr))
            return false;
    arr.items[arr.n] = x;
    for (int i=arr.n;i> 0;i-- )
        if (arr.items[i] < arr.items[i - 1])
            swap(arr.items[i], arr.items[i - 1]);
        else break;
    arr.n++;
    return true;
}
```

3.2 Xâu đơn có thứ tự và có hoặc không có con trỏ cuối

```
struct Node {
    int data;
    Node* next;
    Node(int data=0){
        this->data = data;
        this->next = nullptr;
    }
```

```
};

struct LinkedList{
    Node* head;
    LinkedList(){
        this->head = nullptr;
    }
};


bool InsertElement_LinkedList(LinkedList &list, int value){
    Node* newNode = new Node(value);
    if (list.head == nullptr || value < list.head->data) {
        newNode->next = list.head;
        list.head = newNode;
        return true;
    }
    Node* current = list.head;
    while (current->next != nullptr && current->next->data <
       value)
        current = current->next;

    newNode->next = current->next;
    current->next = newNode;
    return true;
}


struct LinkedListTail{
    Node* head;
    Node* tail;
    LinkedListTail(){
        this->head = nullptr;
        this->tail = nullptr;
    }
};

bool InsertSorted_LinkedListTail(LinkedListTail &list, int value)
   {
    Node* newNode = new Node(value);
    if (list.head == nullptr) {
        list.head = list.tail = newNode;
        return true;
    }

    if (value < list.head->data) {
        newNode->next = list.head;
        list.head = newNode;
        return true;
    }

```

```
    if (value >= list.tail->data) {
        list.tail->next = newNode;
        list.tail = newNode;
        return true;
    }

    Node* current = list.head;
    while (current->next != nullptr && current->next->data <
       value)
        current = current->next;

    newNode->next = current->next;
    current->next = newNode;
    return true;
}
```

3.3 Xâu đơn có thứ tự, có HeaderNode và có hoặc không có con trỏ cuối

```
struct LinkedListWithHeader {
    Node* header;
    LinkedListWithHeader(){
        this->header = new Node();
    }
};

bool InsertSorted_LinkedListWithHeader(LinkedListWithHeader &list
   , int value) {
    Node* newNode = new Node(value);
    Node* current = list.header;

    while (current->next != nullptr && current->next->data <
       value)
        current = current->next;

    newNode->next = current->next;
    current->next = newNode;
    return true;
}

struct LinkedListWithHeaderAndTail {
    Node* header;
    Node* tail;

    LinkedListWithHeaderAndTail() {
        this->header = new Node();
        this->tail = nullptr;
    }
};

```

```
bool InsertSorted_LinkedListWithHeaderAndTail(
   LinkedListWithHeaderAndTail &list, int value) {
    Node* newNode = new Node(value);

    if (list.header->next == nullptr) {
        list.header->next = newNode;
        list.tail = newNode;
        return true;
    }

    if (value >= list.tail->data) {
        list.tail->next = newNode;
        list.tail = newNode;
        return true;
    }

    Node* current = list.header;
    while (current->next != nullptr && current->next->data <
       value) {
        current = current->next;
    }

    newNode->next = current->next;
    current->next = newNode;
    return true;
}
```

3.4 Xâu đơn có thứ tự, có HeaderNode và luôn có địa chỉ node cuối nằm trong các byte đầu của trường dữ liệu HeaderNode

```
struct HeaderNode {
    Node* tailAddress;
    Node* next;
    HeaderNode() {
        tailAddress = nullptr;
        next = nullptr;
    }
};

struct LinkedList{
    HeaderNode* header;
    LinkedList() {
        header = new HeaderNode();
    }
};

bool InsertSorted(LinkedList &list, int value){
    Node* newNode = new Node(value);

    if (list.header->next == nullptr) {
```

```
21          list.header->next = newNode;
22          list.header->tailAddress = newNode;
23          return true;
24      }
25
26      if (value < list.header->next->data) {
27          newNode->next = list.header->next;
28          list.header->next = newNode;
29          return true;
30      }
31
32      if (value >= list.header->tailAddress->data) {
33          list.header->tailAddress->next = newNode;
34          list.header->tailAddress = newNode;
35          return true;
36      }
37
38      Node* current = list.header->next;
39      while (current->next != nullptr && current->next->data <
            value)
40          current = current->next;
41
42      newNode->next = current->next;
43      current->next = newNode;
44
45      return true;
46  }
```