CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

DANH SÁCH LIÊN KẾT

LaTeX, bởi Ngô Hoàng Tùng

I. Mục tiêu bài học

1. Danh sách liên kết đơn

a. Một số lý thuyết.

b. Hiểu được cấu trúc của danh sách liên kết.

c. Biết cách khai báo và sử dụng danh sách liên kết.

d. Biết cách thêm, xóa, duyệt danh sách liên kết.

2. Danh sách liên kết kép

a. Một số lý thuyết.

b. Hiểu được cấu trúc của danh sách liên kết kép.

c. Biết cách khai báo và sử dụng danh sách liên kết kép.

d. Biết cách thêm, xóa, duyệt danh sách liên kết kép.

II. Nội dung bài học

1. Danh sách liên kết đơn

a. Một số lý thuyết:
– Danh sách liên kết đơn là một cấu trúc dữ liệu tuyến tính, trong đó mỗi phần tử (gọi là nút) chứa dữ liệu và một con trỏ đến nút tiếp theo trong danh sách. Danh sách liên kết đơn cho phép thêm, xóa và duyệt các phần tử một cách linh hoạt mà không cần phải di chuyển các phần tử khác.
– Cấu trúc của danh sách liên kết đơn bao gồm một nút đầu (head) và một con trỏ đến nút tiếp theo (next) của mỗi nút. Danh sách liên kết đơn có thể được sử dụng để lưu trữ các phần tử theo thứ tự, cho phép truy cập nhanh đến các phần tử đầu tiên và cuối cùng.
– Danh sách liên kết đơn có thể được sử dụng để giải quyết các bài toán như quản lý danh sách sinh viên, danh sách sản phẩm, hoặc bất kỳ bài toán nào yêu cầu lưu trữ và truy cập dữ liệu theo thứ tự.

b. Cấu trúc của danh sách liên kết đơn:

```
struct Node {
    int data;
    Node* next;
};
struct LinkedList {
    Node* head;
    int size;
    LinkedList() : head(nullptr), tail(nullptr), size(0) {}
};
```

c. Khai báo và sử dụng danh sách liên kết đơn:

```
int main() {
    LinkedList list;
    Node* newNode = new Node();
    newNode->data = 10;
    newNode->next = nullptr;
    list.head = newNode;
    list.size++;
    return 0;
}
```

– Khai báo danh sách liên kết đơn bằng cách tạo một đối tượng của cấu trúc 'LinkedList'.
– Tạo một nút mới bằng cách cấp phát bộ nhớ động và gán giá trị cho nó.
– Gán nút mới vào đầu danh sách liên kết và tăng kích thước của danh sách.

d. Thêm, xóa, duyệt danh sách liên kết đơn: – Thêm nút vào đầu danh sách:

```
void addToHead(LinkedList& list, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = list.head;
    list.head = newNode;
    list.size++;
}
```

– Thêm nút vào cuối danh sách:

```
void addToTail(LinkedList& list, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
    if (list.head == nullptr) {
        list.head = newNode;
    } else {
        Node* current = list.head;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newNode;
    }
    list.size++;
}
```

– Xóa nút đầu danh sách:

```
void removeHead(LinkedList& list) {
    if (list.head == nullptr) return;
    Node* temp = list.head;
    list.head = list.head->next;
    delete temp;
    list.size--;
}
```

– Xóa nút cuối danh sách:

```
void removeTail(LinkedList& list) {
    if (list.head == nullptr) return;
    if (list.head->next == nullptr) {
        delete list.head;
        list.head = nullptr;
    } else {
        Node* current = list.head;
        while (current->next->next != nullptr) {
            current = current->next;
        }
        delete current->next;
        current->next = nullptr;
    }
    list.size--;
}
```

– Duyệt danh sách liên kết đơn:

```
void traverse(LinkedList& list) {
    Node* current = list.head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
```

2. Danh sách liên kết kép

a. Một số lý thuyết:
– Danh sách liên kết kép là một cấu trúc dữ liệu tuyến tính, trong đó mỗi phần tử (gọi là nút) chứa dữ liệu, một con trỏ đến nút tiếp theo (next) và một con trỏ đến nút trước đó (prev). Danh sách liên kết kép cho phép thêm, xóa và duyệt các phần tử một cách linh hoạt hơn so với danh sách liên kết đơn.
– Cấu trúc của danh sách liên kết kép bao gồm một nút đầu (head), một nút cuối (tail) và các con trỏ next và prev của mỗi nút. Danh sách liên kết kép cho phép truy cập nhanh đến các phần tử đầu tiên, cuối cùng và các phần tử ở giữa.
– Danh sách liên kết kép có thể được sử dụng để giải quyết các bài toán như quản lý danh sách sinh viên, danh sách sản phẩm, hoặc bất kỳ bài toán nào yêu cầu lưu trữ và truy cập dữ liệu theo thứ tự với khả năng duyệt ngược lại.

b. Cấu trúc của danh sách liên kết kép:

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

```
    Node* tail;
    int size;
    LinkedList() : head(nullptr), tail(nullptr), size(0) {}
};
```

c. Khai báo và sử dụng danh sách liên kết kép:

```
int main() {
    LinkedList list;
    Node* newNode = new Node();
    newNode->data = 10;
    newNode->next = nullptr;
    newNode->prev = nullptr;
    list.head = newNode;
    list.tail = newNode;
    list.size++;
    return 0;
}
```

– Khai báo danh sách liên kết kép bằng cách tạo một đối tượng của cấu trúc 'LinkedList'.
– Tạo một nút mới bằng cách cấp phát bộ nhớ động và gán giá trị cho nó.
– Gán nút mới vào đầu và cuối danh sách liên kết kép và tăng kích thước của danh sách.

d. Thêm, xóa, duyệt danh sách liên kết kép: – Thêm nút vào đầu danh sách:

```
void addToHead(LinkedList& list, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = list.head;
    newNode->prev = nullptr;
    if (list.head != nullptr) {
        list.head->prev = newNode;
    } else {
        list.tail = newNode;
    }
    list.head = newNode;
    list.size++;
}
```

– Thêm nút vào cuối danh sách:

```
void addToTail(LinkedList& list, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
    newNode->prev = list.tail;
    if (list.tail != nullptr) {
        list.tail->next = newNode;
    } else {
        list.head = newNode;
    }
    list.tail = newNode;
```

```
        list.size++;
    }
```

– Xóa nút đầu danh sách:

```
    void removeHead(LinkedList& list) {
        if (list.head == nullptr) return;
        Node* temp = list.head;
        list.head = list.head->next;
        if (list.head != nullptr) {
            list.head->prev = nullptr;
        } else {
            list.tail = nullptr;
        }
        delete temp;
        list.size--;
    }
```

– Xóa nút cuối danh sách:

```
    void removeTail(LinkedList& list) {
        if (list.tail == nullptr) return;
        Node* temp = list.tail;
        list.tail = list.tail->prev;
        if (list.tail != nullptr) {
            list.tail->next = nullptr;
        } else {
            list.head = nullptr;
        }
        delete temp;
        list.size--;
    }
```

– Duyệt danh sách liên kết kép:

```
    void traverse(LinkedList& list) {
        Node* current = list.head;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->next;
        }
        std::cout << std::endl;
    }
    void traverseReverse(LinkedList& list) {
        Node* current = list.tail;
        while (current != nullptr) {
            std::cout << current->data << " ";
            current = current->prev;
        }
        std::cout << std::endl;
    }
```

III. Code mẫu và giải thích

1. Danh sách liên kết đơn

```
#include <iostream>
struct Node {
    int data;
    Node* next;
};
struct LinkedList {
    Node* head;
    int size;
    LinkedList() : head(nullptr), size(0) {}
};
void addToHead(LinkedList& list, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = list.head;
    list.head = newNode;
    list.size++;
}
void addToTail(LinkedList& list, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
    if (list.head == nullptr) {
        list.head = newNode;
    } else {
        Node* current = list.head;
        while (current->next != nullptr) {
            current = current->next;
        }
        current->next = newNode;
    }
    list.size++;
}
void removeHead(LinkedList& list) {
    if (list.head == nullptr) return;
    Node* temp = list.head;
    list.head = list.head->next;
    delete temp;
    list.size--;
}
void removeTail(LinkedList& list) {
    if (list.head == nullptr) return;
    if (list.head->next == nullptr) {
        delete list.head;
        list.head = nullptr;
    } else {
```

```
        Node* current = list.head;
        while (current->next->next != nullptr) {
            current = current->next;
        }
        delete current->next;
        current->next = nullptr;
    }
    list.size--;
}
void traverse(LinkedList& list) {
    Node* current = list.head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
int main() {
    LinkedList list;
    addToHead(list, 10);
    addToTail(list, 20);
    addToTail(list, 30);
    traverse(list); // Output: 10 20 30
    removeHead(list);
    traverse(list); // Output: 20 30
    removeTail(list);
    traverse(list); // Output: 20
    return 0;
}
```

– Đoạn code trên định nghĩa một danh sách liên kết đơn với các chức năng thêm, xóa và duyệt danh sách.
– Hàm 'addToHead' thêm một nút mới vào đầu danh sách, 'addToTail' thêm vào cuối danh sách, 'removeHead' xóa nút đầu, 'removeTail' xóa nút cuối, và 'traverse' duyệt danh sách để in ra các giá trị.
– Trong hàm 'main', ta tạo một danh sách liên kết, thêm các nút vào đầu và cuối, duyệt và in ra danh sách, sau đó xóa các nút đầu và cuối và in lại danh sách.
– Kết quả in ra sẽ là các giá trị của các nút trong danh sách liên kết.
– Lưu ý rằng việc quản lý bộ nhớ là rất quan trọng trong danh sách liên kết, vì vậy cần phải giải phóng bộ nhớ đã cấp phát khi không còn sử dụng nữa.

2. Danh sách liên kết kép

```
#include <iostream>
struct Node {
    int data;
    Node* next;
    Node* prev;
};
struct LinkedList {
    Node* head;
    Node* tail;
```

```
    int size;
    LinkedList() : head(nullptr), tail(nullptr), size(0) {}
};
void addToHead(LinkedList& list, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = list.head;
    newNode->prev = nullptr;
    if (list.head != nullptr) {
        list.head->prev = newNode;
    } else {
        list.tail = newNode;
    }
    list.head = newNode;
    list.size++;
}
void addToTail(LinkedList& list, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
    newNode->prev = list.tail;
    if (list.tail != nullptr) {
        list.tail->next = newNode;
    } else {
        list.head = newNode;
    }
    list.tail = newNode;
    list.size++;
}
void removeHead(LinkedList& list) {
    if (list.head == nullptr) return;
    Node* temp = list.head;
    list.head = list.head->next;
    if (list.head != nullptr) {
        list.head->prev = nullptr;
    } else {
        list.tail = nullptr;
    }
    delete temp;
    list.size--;
}
void removeTail(LinkedList& list) {
    if (list.tail == nullptr) return;
    Node* temp = list.tail;
    list.tail = list.tail->prev;
    if (list.tail != nullptr) {
        list.tail->next = nullptr;
    } else {
        list.head = nullptr;
```

```
    }
    delete temp;
    list.size--;
}
void traverse(LinkedList& list) {
    Node* current = list.head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }
    std::cout << std::endl;
}
void traverseReverse(LinkedList& list) {
    Node* current = list.tail;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->prev;
    }
    std::cout << std::endl;
}
int main() {
    LinkedList list;
    addToHead(list, 10);
    addToTail(list, 20);
    addToTail(list, 30);
    traverse(list); // Output: 10 20 30
    traverseReverse(list); // Output: 30 20 10
    removeHead(list);
    traverse(list); // Output: 20 30
    removeTail(list);
    traverse(list); // Output: 20
    return 0;
}
```

– Đoạn code trên định nghĩa một danh sách liên kết kép với các chức năng thêm, xóa và duyệt danh sách.
– Hàm ‘addToHead‘ thêm một nút mới vào đầu danh sách, ‘addToTail‘ thêm vào cuối danh sách, ‘removeHead‘ xóa nút đầu, ‘removeTail‘ xóa nút cuối, ‘traverse‘ duyệt danh sách để in ra các giá trị từ đầu đến cuối, và ‘traverseReverse‘ duyệt ngược lại từ cuối đến đầu.
– Trong hàm ‘main‘, ta tạo một danh sách liên kết kép, thêm các nút vào đầu và cuối, duyệt và in ra danh sách theo cả hai chiều, sau đó xóa các nút đầu và cuối và in lại danh sách.
– Kết quả in ra sẽ là các giá trị của các nút trong danh sách liên kết kép.
– Lưu ý rằng việc quản lý bộ nhớ là rất quan trọng trong danh sách liên kết kép, vì vậy cần phải giải phóng bộ nhớ đã cấp phát khi không còn sử dụng nữa.

IV. Bài tập thực hành

1. Danh sách liên kết đơn

Đề:

Cho n là một số nguyên dương, và 1 mảng chứa n phần tử. Thực hiện các truy vấn có dạng sau:
1 id – Thêm phần tử có vị trí id vào đầu danh sách liên kết.
2 value – Thêm phần tử có giá trị value vào cuối danh sách liên kết.(Luôn đảm bảo giá trị value có trong mảng)
3 id – Xóa phần tử thứ id trong danh sách khỏi danh sách liên kết.
4 val – Xóa phần tử có giá trị bằng value khỏi danh sách liên kết.(Luôn dảmd bảo có giá trị value trong danh sách)
5 – In ra các phần tử trong danh sách liên kết.

Input:

– Dòng đầu tiên chứa số nguyên dương n.
– Dòng thứ hai chứa n số nguyên, là các phần tử của mảng. $(a_1, a_2, ...a_n)$
– Dòng thứ ba chứa số nguyên dương q, là số lượng truy vấn.
– Các dòng tiếp theo chứa các truy vấn theo định dạng đã nêu ở trên.

Output:

– Với mỗi truy vấn in ra kết quả tương ứng. Nếu không có phần tử nào thì in ra "Empty".

Input	Output
5 1 2 3 4 5 5 1 3 2 2 1 1 2 5 5	1 3 2 5

– Giải thích:
+ Truy vấn loại 1 thêm phần tử có vị trí id vào đầu danh sách liên kết. Ở đây ta thêm a[3] = 3 vào đầu danh sách ta được (3)
+ Truy vấn loại 2 thêm phần tử có giá trị value vào cuối danh sách liên kết. Ở đây ta thêm value = 2 vào cuối danh sách ta được (3, 2)
+ Sau đó ta vẫn thêm a[1] = 1 vào đầu danh sách ta được (1, 3, 2)
+ Cuối cùng ta thêm value = 5 vào cuối danh sách ta được (1, 3, 2, 5)

2. Danh sách liên kết kép

Đề:

Cho n là một số nguyên dương, và 1 mảng chứa n phần tử. Thực hiện các truy vấn có dạng sau:
1 id – Thêm phần tử có vị trí id vào đầu danh sách liên kết.
2 value – Thêm phần tử có giá trị value vào cuối danh sách liên kết.(Luôn đảm bảo giá trị value có trong mảng)

3 id – Xóa phần tử thứ id trong danh sách khỏi danh sách liên kết.
4 val – Xóa phần tử có giá trị bằng value khỏi danh sách liên kết.(Luôn dảmd bảo có giá trị value trong danh sách)
5 – In ra các phần tử trong danh sách liên kết theo thứ tự từ trái qua phải.
6 – In ra các phần tử trong danh sách liên kết theo thứ tự từ phải qua trái.

Input:

– Dòng đầu tiên chứa số nguyên dương n.
– Dòng thứ hai chứa n số nguyên, là các phần tử của mảng. $(a_1, a_2, ...a_n)$
– Dòng thứ ba chứa số nguyên dương q, là số lượng truy vấn.
– Các dòng tiếp theo chứa các truy vấn theo định dạng đã nêu ở trên.

Output:

– Với mỗi truy vấn in ra kết quả tương ứng. Nếu không có phần tử nào thì in ra "Empty".
– Giải thích:

Input	Output
5 1 2 3 4 5 5 1 3 2 2 1 1 2 5 5 6	1 3 2 5 5 2 3 1

+ Truy vấn loại 1 thêm phần tử có vị trí id vào đầu danh sách liên kết. Ở đây ta thêm a[3] = 3 vào đầu danh sách ta được (3)
+ Truy vấn loại 2 thêm phần tử có giá trị value vào cuối danh sách liên kết. Ở đây ta thêm value = 2 vào cuối danh sách ta được (3, 2)
+ Sau đó ta vẫn thêm a[1] = 1 vào đầu danh sách ta được (1, 3, 2)
+ Cuối cùng ta thêm value = 5 vào cuối danh sách ta được (1, 3, 2, 5)
+ Truy vấn loại 5 in ra các phần tử trong danh sách liên kết theo thứ tự từ trái qua phải. Ta được (1, 3, 2, 5)
+ Truy vấn loại 6 in ra các phần tử trong danh sách liên kết theo thứ tự từ phải qua trái. Ta được (5, 2, 3, 1)

V. Chia sẻ kinh nghiệm

1. Danh sách liên kết đơn

- Nên sử dụng danh sách liên kết đơn khi cần quản lý một tập hợp các phần tử mà không cần truy cập ngược lại.
- Cần chú ý đến việc quản lý bộ nhớ, đặc biệt là khi xóa các nút để tránh rò rỉ bộ nhớ.
- Việc duyệt danh sách liên kết đơn có thể thực hiện dễ dàng bằng cách sử dụng vòng lặp while.
- Nên sử dụng danh sách liên kết đơn khi cần thực hiện các thao tác thêm, xóa nhanh chóng mà không cần phải di chuyển các phần tử khác.

- Cần chú ý đến việc kiểm tra điều kiện rỗng của danh sách trước khi thực hiện các thao tác thêm, xóa để tránh lỗi.
- Nên sử dụng danh sách liên kết đơn khi cần lưu trữ dữ liệu có kích thước thay đổi thường xuyên, vì nó cho phép thêm và xóa phần tử một cách linh hoạt.
- Cần chú ý đến việc cập nhật kích thước của danh sách sau mỗi thao tác thêm hoặc xóa để đảm bảo tính chính xác của thông tin.
- Nên sử dụng danh sách liên kết đơn khi cần lưu trữ dữ liệu theo thứ tự, vì nó cho phép truy cập nhanh đến các phần tử đầu tiên và cuối cùng.
- Cần chú ý đến việc sử dụng con trỏ để liên kết các nút với nhau, vì nếu không quản lý đúng, có thể dẫn đến lỗi truy cập bộ nhớ.
- Nên sử dụng danh sách liên kết đơn khi cần thực hiện các thao tác trên danh sách mà không cần phải di chuyển các phần tử khác, vì nó cho phép thêm và xóa phần tử một cách linh hoạt.
- Cần chú ý đến việc giải phóng bộ nhớ đã cấp phát cho các nút khi không còn sử dụng nữa để tránh rò rỉ bộ nhớ.

2. Danh sách liên kết kép

- Nên sử dụng danh sách liên kết kép khi cần quản lý một tập hợp các phần tử mà cần truy cập ngược lại.
- Cần chú ý đến việc quản lý bộ nhớ, đặc biệt là khi xóa các nút để tránh rò rỉ bộ nhớ.
- Việc duyệt danh sách liên kết kép có thể thực hiện dễ dàng bằng cách sử dụng vòng lặp while hoặc for.
- Nên sử dụng danh sách liên kết kép khi cần thực hiện các thao tác thêm, xóa nhanh chóng mà không cần phải di chuyển các phần tử khác.
- Cần chú ý đến việc kiểm tra điều kiện rỗng của danh sách trước khi thực hiện các thao tác thêm, xóa để tránh lỗi.
- Nên sử dụng danh sách liên kết kép khi cần lưu trữ dữ liệu có kích thước thay đổi thường xuyên, vì nó cho phép thêm và xóa phần tử một cách linh hoạt.
- Cần chú ý đến việc cập nhật kích thước của danh sách sau mỗi thao tác thêm hoặc xóa để đảm bảo tính chính xác của thông tin.
- Nên sử dụng danh sách liên kết kép khi cần lưu trữ dữ liệu theo thứ tự, vì nó cho phép truy cập nhanh đến các phần tử đầu tiên, cuối cùng và các phần tử ở giữa.
- Cần chú ý đến việc sử dụng con trỏ để liên kết các nút với nhau, vì nếu không quản lý đúng, có thể dẫn đến lỗi truy cập bộ nhớ.
- Nên sử dụng danh sách liên kết kép khi cần thực hiện các thao tác trên danh sách mà không cần phải di chuyển các phần tử khác, vì nó cho phép thêm và xóa phần tử một cách linh hoạt.
- Cần chú ý đến việc giải phóng bộ nhớ đã cấp phát cho các nút khi không còn sử dụng nữa để tránh rò rỉ bộ nhớ.

HẾT