BÁO CÁO TRIỂN KHAI DANH SÁCH LIÊN KẾT

# I. Mục đích

Danh sách liên kết đơn là một cấu trúc dữ liệu động được sử dụng để lưu trữ và quản lý các phần tử không liên tục trong bộ nhớ. Báo cáo này trình bày việc triển khai hai bài tập sử dụng danh sách liên kết đơn trong C++:  
  
Bài 1: Quản lý danh sách số nguyên với các thao tác cơ bản.  
Bài 2: Quản lý danh sách sinh viên với thông tin chi tiết và sắp xếp.  
  
Mục tiêu là cài đặt các cấu trúc dữ liệu (CTDL) và chương trình hoàn chỉnh theo yêu cầu, đồng thời kiểm tra tính đúng đắn của các chức năng.

# II. Bài 1: Danh sách liên kết số nguyên

## 1. Cấu trúc dữ liệu và các tác vụ

CTDL ListInt:  
- Sử dụng danh sách liên kết đơn với mỗi node chứa một số nguyên (data) và con trỏ tới node tiếp theo (next).  
Các phép toán:  
- Khởi tạo (ListInt{ }): Tạo danh sách rỗng bằng cách đặt head về nullptr.  
- Hủy (~ListInt{ }): Xóa tất cả node và giải phóng bộ nhớ.  
- Thêm phần tử (add): Thêm một số nguyên vào cuối danh sách.  
- Xóa phần tử (remove): Xóa phần tử đầu tiên có giá trị được chỉ định.  
- Thêm một danh sách (appendList): Nối một danh sách khác vào cuối danh sách hiện tại.  
- Xuất danh sách (print): In các phần tử trong danh sách ra màn hình.

## 2. Chương trình

Quy trình:  
- Nhập 10 số nguyên từ người dùng và thêm vào danh sách list1.  
- In danh sách ban đầu.  
- Nhập số k và xóa k khỏi danh sách.  
- In danh sách sau khi xóa.  
- Nhập 5 số nguyên vào danh sách thứ hai list2.  
- Nối list2 vào list1.  
- In danh sách list1 sau khi nối.  
  
Ví dụ đầu vào:  
10 số: 5 2 9 1 7 3 8 4 6 10  
Số cần xóa: 9  
5 số cho danh sách thứ hai: 11 12 13 14 15

#include <iostream>

struct Node

{

    int data;

    Node \*next;

    Node(int value) : data(value), next(nullptr) {}

};

class ListInt

{

private:

    Node \*head;

public:

    ListInt() : head(nullptr) {}

    ~ListInt()

    {

        while (head != nullptr)

        {

            Node \*temp = head;

            head = head->next;

            delete temp;

        }

    }

    void add(int value)

    {

        Node \*newNode = new Node(value);

        if (head == nullptr)

        {

            head = newNode;

            return;

        }

        Node \*current = head;

        while (current->next != nullptr)

        {

            current = current->next;

        }

        current->next = newNode;

    }

    void remove(int value)

    {

        if (head == nullptr)

            return;

        if (head->data == value)

        {

            Node \*temp = head;

            head = head->next;

            delete temp;

            return;

        }

        Node \*current = head;

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

        {

            current = current->next;

        }

        if (current->next != nullptr)

        {

            Node \*temp = current->next;

            current->next = temp->next;

            delete temp;

        }

    }

    void appendList(ListInt &other)

    {

        if (other.head == nullptr)

            return;

        Node \*newHead = new Node(other.head->data);

        Node \*current = newHead;

        Node \*otherCurrent = other.head->next;

        while (otherCurrent != nullptr)

        {

            current->next = new Node(otherCurrent->data);

            current = current->next;

            otherCurrent = otherCurrent->next;

        }

        if (head == nullptr)

        {

            head = newHead;

        }

        else

        {

            current = head;

            while (current->next != nullptr)

            {

                current = current->next;

            }

            current->next = newHead;

        }

    }

    void print()

    {

        Node \*current = head;

        while (current != nullptr)

        {

            std::cout << current->data << " ";

            current = current->next;

        }

        std::cout << "\n";

    }

};

int main()

{

    ListInt list1;

    for (int i = 0; i < 10; i++)

    {

        int value;

        std::cin >> value;

        list1.add(value);

    }

    std::cout << "Initial list: ";

    list1.print();

    int k;

    std::cout << "Enter number to remove: ";

    std::cin >> k;

    list1.remove(k);

    std::cout << "List after removing " << k << ": ";

    list1.print();

    ListInt list2;

    for (int i = 0; i < 5; i++)

    {

        int value;

        std::cin >> value;

        list2.add(value);

    }

    list1.appendList(list2);

    std::cout << "List after appending second list: ";

    list1.print();

    return 0;

}

## 3. Nhận xét

Ưu điểm:  
- Code đơn giản, dễ hiểu, đáp ứng đầy đủ các yêu cầu cơ bản.  
  
Hạn chế:  
- Việc thêm phần tử vào cuối danh sách có độ phức tạp O(n) do phải duyệt từ đầu đến cuối.

# III. Bài 2: Danh sách liên kết sinh viên

## 1. Cấu trúc dữ liệu và các tác vụ

CTDL SinhVien:  
- Chứa thông tin sinh viên: hoTen (họ tên), diaChi (địa chỉ), lop (lớp), khoa (khóa) sử dụng std::string cho các trường chuỗi và int cho khóa.  
  
Hàm nhập (inputStudent): Nhập thông tin một sinh viên từ người dùng.  
Hàm xuất (outputStudent): In thông tin một sinh viên.  
  
4 hàm so sánh:  
- compareByName: So sánh theo họ tên.  
- compareByAddress: So sánh theo địa chỉ.  
- compareByClass: So sánh theo lớp.  
- compareByYear: So sánh theo khóa.  
  
CTDL ListSV:  
- Sử dụng danh sách liên kết đơn với mỗi node chứa một SinhVien và con trỏ tới node tiếp theo.  
  
Các phép toán:  
- Khởi tạo (ListSV()): Tạo danh sách rỗng.  
- Hủy (~ListSV()): Giải phóng bộ nhớ.  
- Thêm phần tử (add): Thêm sinh viên vào cuối danh sách.  
- Xóa phần tử (removeByName, removeByAddress): Xóa sinh viên theo họ tên hoặc địa chỉ.  
- Thêm một danh sách (appendList): Nối danh sách khác vào cuối.  
- Xuất danh sách (print): In toàn bộ danh sách.  
- Sắp xếp (sort): Sử dụng Selection Sort với con trỏ hàm so sánh.

## 2. Chương trình

Quy trình:  
- Nhập thông tin 10 sinh viên vào danh sách list.  
- In danh sách ban đầu.  
- Xóa sinh viên có tên 'Nguyen Van Teo'.  
- Xóa sinh viên có địa chỉ 'Nguyen Van Cu'.  
- Thêm sinh viên mới: 'Tran Thi Mo', '25 Hong Bang', 'TT0901', 2009.  
- In danh sách sau khi thực hiện các thao tác.  
  
Ví dụ đầu vào (giả lập):  
Sinh viên 1: 'Nguyen Van A', '123 Tran Hung Dao', 'CNTT01', 2020  
Sinh viên 2: 'Nguyen Van Teo', '456 Le Loi', 'CNTT02', 2021  
Sinh viên 3: 'Tran Thi B', 'Nguyen Van Cu', 'CNTT03', 2022  
(7 sinh viên khác tương tự)

#include <iostream>

#include <string>

struct SinhVien

{

    std::string hoTen;

    std::string diaChi;

    std::string lop;

    int khoa;

    SinhVien() {}

    SinhVien(const std::string &ht, const std::string &dc, const std::string &l, int k)

        : hoTen(ht), diaChi(dc), lop(l), khoa(k) {}

};

struct Node

{

    SinhVien data;

    Node \*next;

    Node(SinhVien sv) : data(sv), next(nullptr) {}

};

bool compareByName(SinhVien sv1, SinhVien sv2)

{

    return sv1.hoTen < sv2.hoTen;

}

bool compareByAddress(SinhVien sv1, SinhVien sv2)

{

    return sv1.diaChi < sv2.diaChi;

}

bool compareByClass(SinhVien sv1, SinhVien sv2)

{

    return sv1.lop < sv2.lop;

}

bool compareByYear(SinhVien sv1, SinhVien sv2)

{

    return sv1.khoa < sv2.khoa;

}

void inputStudent(SinhVien &sv)

{

    std::cin.ignore();

    std::cout << "Enter name: ";

    std::getline(std::cin, sv.hoTen);

    std::cout << "Enter address: ";

    std::getline(std::cin, sv.diaChi);

    std::cout << "Enter class: ";

    std::getline(std::cin, sv.lop);

    std::cout << "Enter year: ";

    std::cin >> sv.khoa;

}

void outputStudent(SinhVien sv)

{

    std::cout << "Name: " << sv.hoTen << ", Address: " << sv.diaChi

              << ", Class: " << sv.lop << ", Year: " << sv.khoa << "\n";

}

class ListSV

{

private:

    Node \*head;

public:

    ListSV() : head(nullptr) {}

    ~ListSV()

    {

        while (head != nullptr)

        {

            Node \*temp = head;

            head = head->next;

            delete temp;

        }

    }

    void add(SinhVien sv)

    {

        Node \*newNode = new Node(sv);

        if (head == nullptr)

        {

            head = newNode;

            return;

        }

        Node \*current = head;

        while (current->next != nullptr)

        {

            current = current->next;

        }

        current->next = newNode;

    }

    void removeByName(const std::string &name)

    {

        if (head == nullptr)

            return;

        if (head->data.hoTen == name)

        {

            Node \*temp = head;

            head = head->next;

            delete temp;

            return;

        }

        Node \*current = head;

        while (current->next != nullptr && current->next->data.hoTen != name)

        {

            current = current->next;

        }

        if (current->next != nullptr)

        {

            Node \*temp = current->next;

            current->next = temp->next;

            delete temp;

        }

    }

    void removeByAddress(const std::string &address)

    {

        if (head == nullptr)

            return;

        if (head->data.diaChi == address)

        {

            Node \*temp = head;

            head = head->next;

            delete temp;

            return;

        }

        Node \*current = head;

        while (current->next != nullptr && current->next->data.diaChi != address)

        {

            current = current->next;

        }

        if (current->next != nullptr)

        {

            Node \*temp = current->next;

            current->next = temp->next;

            delete temp;

        }

    }

    void appendList(ListSV &other)

    {

        if (other.head == nullptr)

            return;

        Node \*newHead = new Node(other.head->data);

        Node \*current = newHead;

        Node \*otherCurrent = other.head->next;

        while (otherCurrent != nullptr)

        {

            current->next = new Node(otherCurrent->data);

            current = current->next;

            otherCurrent = otherCurrent->next;

        }

        if (head == nullptr)

        {

            head = newHead;

        }

        else

        {

            current = head;

            while (current->next != nullptr)

            {

                current = current->next;

            }

            current->next = newHead;

        }

    }

    void print()

    {

        Node \*current = head;

        while (current != nullptr)

        {

            outputStudent(current->data);

            current = current->next;

        }

    }

    void sort(bool (\*compare)(SinhVien, SinhVien))

    {

        if (head == nullptr || head->next == nullptr)

            return;

        Node \*current = head;

        while (current != nullptr)

        {

            Node \*minNode = current;

            Node \*next = current->next;

            while (next != nullptr)

            {

                if (!compare(minNode->data, next->data))

                {

                    minNode = next;

                }

                next = next->next;

            }

            if (minNode != current)

            {

                SinhVien temp = current->data;

                current->data = minNode->data;

                minNode->data = temp;

            }

            current = current->next;

        }

    }

};

int main()

{

    ListSV list;

    for (int i = 0; i < 10; i++)

    {

        SinhVien sv;

        std::cout << "Enter student " << i + 1 << ":\n";

        inputStudent(sv);

        list.add(sv);

    }

    std::cout << "Initial list:\n";

    list.print();

    list.removeByName("Nguyen Van Teo");

    list.removeByAddress("Nguyen Van Cu");

    std::cout << "List after removing:\n";

    list.print();

    SinhVien newSV("Tran Thi Mo", "25 Hong Bang", "TT0901", 2009);

    list.add(newSV);

    std::cout << "List after adding new student:\n";

    list.print();

    return 0;

}

## 3. Nhận xét

Ưu điểm:  
- Sử dụng std::string giúp linh hoạt trong việc lưu trữ chuỗi, không cần giới hạn độ dài cố định.  
- Hàm so sánh và Selection Sort với con trỏ hàm cho phép sắp xếp theo nhiều tiêu chí khác nhau.  
  
Hạn chế:  
- Selection Sort có độ phức tạp O(n²), không tối ưu cho danh sách lớn.  
- Việc thêm phần tử vào cuối danh sách vẫn là O(n) vì không dùng con trỏ tail.

# IV. Nhận xét chung

Hiệu quả: Cả hai bài đều triển khai thành công các chức năng yêu cầu, phù hợp với danh sách nhỏ (10-15 phần tử).  
Tối ưu hóa: Có thể cải thiện bằng cách:  
- Thêm con trỏ tail để giảm độ phức tạp của thao tác thêm phần tử vào cuối từ O(n) xuống O(1).  
- Sử dụng thuật toán sắp xếp hiệu quả hơn (như Merge Sort) cho bài 2 nếu danh sách lớn.  
  
Ứng dụng: Các CTDL này có thể mở rộng để quản lý dữ liệu thực tế như danh sách học sinh, nhân viên, v.v.

# V. Kết luận

Hai bài tập đã được hoàn thành với việc sử dụng danh sách liên kết đơn trong C++. Code đáp ứng đầy đủ yêu cầu đề bài, từ quản lý số nguyên đơn giản đến thông tin sinh viên phức tạp hơn. Đây là nền tảng tốt để học tập và phát triển thêm các cấu trúc dữ liệu nâng cao như danh sách liên kết đôi hoặc cây.