



BÀI GIẢNG CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

CHƯƠNG 3. MỘT SỐ CẤU TRÚC DỮ LIỆU CƠ BẢN

TS. NGUYỄN ĐÌNH DƯƠNG
BỘ MÔN KHMT - KHOA CÔNG NGHỆ THÔNG TIN

Email: duongnd@utc.edu.vn

Ngày 13/06/2022



Nội dung

Vectors

- 1.1 Một số khái niệm cơ bản
- 1.2 Các hàm của Vectors trong C +
- 1.3 Cài đặt demo class Vector

Danh sách liên kết

- 2.1 Giới thiệu
- 2.2 Khai báo
- 2.3 Các thao tác cơ bản
- 2.4 Bài tập
- 2.5 Sắp xếp trên DSLK đơn
- 2.6 Danh sách liên kết đôi (Double Linked List)
- 2.7 Danh sách liên kết vòng (Circular Linked List)

Trao đổi



Nội dung

Vectors

- 1.1 Một số khái niệm cơ bản
- 1.2 Các hàm của Vectors trong C +
- 1.3 Cài đặt demo class Vector

Danh sách liên kết

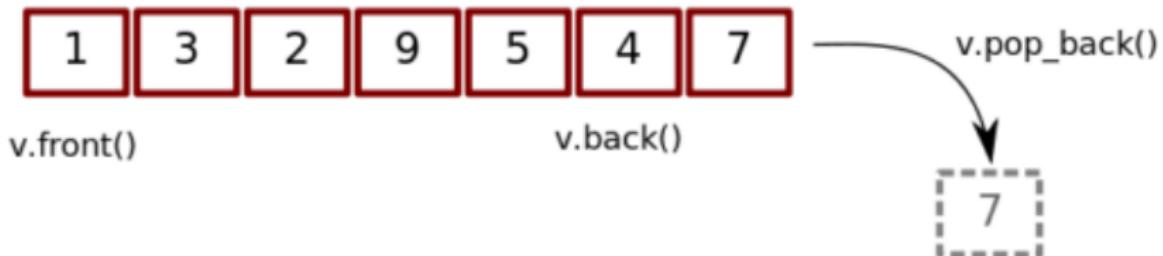
Trao đổi



1. Vectors

1. 1. Một số khái niệm cơ bản

- Array (mảng): chỉ một số giá trị nhất định có thể được lưu trữ dưới một tên biến duy nhất.
- Vector trong C++ giống dynamic array (mảng động) nhưng có khả năng tự động thay đổi kích thước khi một phần tử được chèn hoặc xóa.
- Các phần tử vector được đặt trong continuous storage (bộ nhớ liền kề) cho phép truy cập và duyệt bằng cách sử dụng iterator.





1. Vectors

1. 1. Một số khái niệm cơ bản

- **Cú pháp:**

```
#include <vector>
//...
vector<object_type> variable_name;
```

- **Ví dụ:**

```
#include <vector>
int main()
{
    std::vector<int> my_vector;
    std::vector<int> your_vector = {1,2,3,4,5};
    std::vector<int> my_vector = your_vectors;
}
```



1. Vectors

1. 1. Một số khái niệm cơ bản

- Vector tự ghi nhớ độ dài của mình thông qua hàm size():

```
#include <iostream>
#include <vector>
void printLength(const std::vector<int>& array)
{
    std::cout << "The length is: " << array.size() << '\n';
}
int main()
{
    std::vector<int> array{ 9, 7, 5, 3, 1};
    printLength(array);
    return 0;
}
```



1. Vectors

1. 2. Các hàm của Vectors trong C +

Vector trong STL cung cấp cho chúng ta nhiều chức năng hữu ích khác nhau.

- ① Modifiers
- ② Iterators
- ③ Capacity
- ④ Element access



1. Vectors

1. 2. Các hàm của Vectors trong C +

Modifiers

- ① **push_back()**: đẩy một phần tử vào vị trí sau cùng của vector. Nếu kiểu của đối tượng được truyền dưới dạng tham số trong push_back() không giống với kiểu của vector thì sẽ bị ném ra.
`ten-vector.push_back(ten-cua-phan-tu);`
- ② **assign()**: gán một giá trị mới cho các phần tử vector bằng cách thay thế các giá trị cũ.
`ten-vector.assign(int size, int value);`
- ③ **pop_back()**: xóa đi phần tử cuối cùng một vector.
`ten-vector.pop_back();`
- ④ **insert()**: chèn phần tử mới vào trước phần tử tại vị trí cho trước.
`ten-vector.insert(position, value);`
- ⑤ **reserve(start-position, end-position)**: đảo ngược thứ tự từ vị trí start tới end.
`ten-vector.reserve(start-position, end-position);`



1. Vectors

1. 2. Các hàm của Vectors trong C +

Modifiers

- ⑥ **erase()**: xóa các phần tử tùy theo vị trí vùng chứa
`ten-vector.erase(position);`
`ten-vector.erase(start-position, end-position);`
- ⑦ **emplace()**: mở rộng vùng chứa bằng cách chèn phần tử mới vào
`ten-vector.emplace(ten-vector.position, element);`
- ⑧ **emplace_back()**: chèn một phần tử mới vào vùng chứa vector,
phần tử mới sẽ được thêm vào cuối vector.
`ten-vector.emplace_back(value);`
- ⑨ **swap()**: hoán đổi nội dung của một vector này với một vector khác
cùng kiểu (kích thước có thể khác nhau).
`ten-vector-1.swap(ten-vector-2);`
- ⑩ **clear()**: loại bỏ tất cả các phần tử của vùng chứa vector.
`ten-vector.clear();`



1. Vectors

1. 2. Các hàm của Vectors trong C +

Iterators

- ① **begin()**: đặt iterator đến phần tử đầu tiên trong vector.
`ten-vector.begin();`
- ② **end()**: đặt iterator đến sau phần tử cuối cùng trong vector.
`ten-vector.end();`
- ③ **rbegin()**: đặt reverse iterator (trình lặp đảo) đến phần tử cuối cùng trong vector (reverse begin).
`ten-vector.rbegin();`
- ④ **rend()**: đặt reverse iterator (trình lặp đảo) đến phần tử đầu tiên trong vector (reverse end)
`ten-vector.rend();`



1. Vectors

Capacity

- ① **size()**: trả về số lượng phần tử đang được sử dụng trong vector,
`ten-vector.size();`
- ② **max_size()**: trả về số phần tử tối đa mà vector có thể chứa.
`ten-vector.max_size();`
- ③ **capacity()**: trả về số phần tử được cấp phát cho vector nằm trong bộ nhớ,
`ten-vector.capacity();`
- ④ **resize(n)**: thay đổi kích thước vùng chứa để nó chứa đủ n phần tử. Nếu kích
thước hiện tại của vector lớn hơn n thì các phần tử phía sau sẽ bị xóa khỏi vector
và ngược lại nếu kích thước hiện tại nhỏ hơn n thì các phần tử bổ sung sẽ được
chèn vào phía sau vector
`ten-vector.resize(int n, int value);`
- ⑤ **empty()**: kiểm tra vùng chứa có trống hay không, nếu trống thì trả về True, nếu có
phần tử thì trả về False
`ten-vector.empty();`
- ⑥ **shrink_to_fit()**: Giảm dung lượng của vùng chứa để phù hợp với kích thước của
nó và hủy tất cả các phần tử vượt quá dung lượng
`ten-vector.shrink_to_fit();`



1. Vectors

1. 2. Các hàm của Vectors trong C +

Element access

- ① **at(g):** Trả về một tham chiếu đến phần tử ở vị trí 'g' trong vector,
`ten-vector.at(position);`
- ② **data():** Trả về một con trỏ trực tiếp đến (memory array) bộ nhớ
mảng được vector sử dụng bên trong để lưu trữ các phần tử thuộc
sở hữu của nó.
`ten-vector.data();`
- ③ **front():** lấy ra phần tử đầu tiên của vector, `ten-vector.front();`
- ④ **back():** lấy ra phần tử cuối cùng của vector, `ten-vector.back();`



1. Vectors

1. 2. Các hàm của Vectors trong C +

Sort, search

- ① **sort(first, last)**: sắp xếp từ vị trí first tới last

```
sort(vt.begin(), vt.end()); // sắp xếp cả mảng
```

```
sort(vt.begin() + 2, vt.begin() + 5); //sắp xếp các phần tử  
từ vị trí 2 tới 5
```

- ② **find**:

- ③ **seach**:



1. Vectors

1. 2. Các hàm của Vectors trong C +

So sánh mảng và vector

Mảng	Vector
Mảng tĩnh thừa hướng từ ngôn ngữ C	Mảng động được thêm từ C++
Có kích thước cố định	Tự động thay đổi kích thước
Hiệu năng tốt hơn do kích thước cố định trong bộ nhớ	Hiệu năng kém hơn do con trỏ phải tìm để lấy dữ liệu trong các ngăn xếp
Tốc độ nhanh hơn nhưng xử lý bất tiện	Xử lý linh hoạt hơn nhưng tốc độ chậm hơn một chút



1. Vectors

1. 3. Cài đặt demo class Vector

```
#include <bits/stdc++.h>
using namespace std;
template <typename T>
class Vector{
private:
    int size; // bien luu do dai cua vector
    int capacity; // bien luu dung luong cua vector
    T *array; // tao con tro chua mang
    void expand(int newCapacity); // ham tang dung luong
vector
public:
    Vector(int initCapacity = 100) // ham tao vector
    ~Vector() // ham huy vector
    Vector & operator=(Vector & rhs); // ham nap chong toan tu
    gan
    int Size(); // ham tra ve kich thuoc
```



1. Vectors

1. 3. Cài đặt demo class Vector

```
T & operator[](int index); // ham nap chong toan tu truy  
cap chi so vector  
void push_back(T newElement); // ham day phan tu vao cuoi  
vector  
void pop_back(); //ham xoa phan tu cuoi vector  
void insert(int pos, T newElement); // ham them phan tu  
vao vi tri pos  
void erase(int pos); //ham xoa phan tu tai vi tri pos  
void clear(); // ham xoa tat ca cac phan tu vector  
};
```



1. Vectors

1. 3. Cài đặt demo class Vector

```
void expand(int newCapacity){  
    if (newCapacity <= size)  
        return;  
    T * old = array; // old tro toi mang cu  
    array = new T[newCapacity]; // array tro toi mang moi  
    for (int i = 0; i < size; i++)  
        array[i] = old[i]; // sao chep phan tu tu mang cu sang  
    mang moi  
    delete[] old; // xoa mang cu  
    capacity = newCapacity; // dat dung luong moi  
}
```



1. Vectors

1. 3. Cài đặt demo class Vector

- **Cấu tử:**

```
Vector(int initCapacity = 100){  
    size = 0;  
    capacity = initCapacity;  
    array = new T[capacity];  
}
```

- **Huỷ tử:**

```
~Vector(){  
    delete[] array;  
}
```



1. Vectors

1. 3. Cài đặt demo class Vector

- Hàm nạp chồng toán tử gán một ngôi:

```
Vector & operator=(Vector & rhs){  
    if (this != &rhs){ // ngan can tu sao chep  
        delete[] array; // xoa mang hien tai  
        size = rhs.size; // dat kich thuoc moi  
        capacity = rhs.capacity; // dat dung luong moi  
        array = new T[capacity]; // tao mang moi  
        // Sao chep cac phan tu tu phai sang trai  
        for (int i = 0; i < size; i++)  
            array[i] = rhs.array[i];  
    }  
    return *this; // tra ve vector ve trai sau khi gan  
    xong  
}
```



1. Vectors

1. 3. Cài đặt demo class Vector

- **Hàm trả về độ dài của vector:**

```
int Size() {  
    return size;  
}
```

- **Hàm kiểm tra vector rỗng:**

```
bool empty() {  
    return (size == 0);  
}
```

- **Hàm nạp chồng toán tử truy cập chỉ số của vector:**

```
T & operator[](int index){  
    return array[index];  
}
```



1. Vectors

1. 3. Cài đặt demo class Vector

- **Hàm thêm một giá trị vào cuối vector:**

```
void push_back(T newElement){  
    // Gap doi dung luong neu vector day  
    if (size == capacity)  
        expand(2 * size);  
    // Chen phan tu moi vao ngay sau phan tu cuoi cung  
    array[size] = newElement;  
    // Tang kich thuoc  
    size++;  
}
```

- **Hàm xóa phần tử cuối cùng của vector:**

```
void popBack(){  
    size-;  
}
```



1. Vectors

1. 3. Cài đặt demo class Vector

- **Hàm thêm phần tử vào vị trí pos với giá trị là newElement:**

```
void insert(int pos, T newElement){  
    // Gap doi dung luong neu vector day  
    if (size == capacity)  
        expand(2 * size);  
    // Dich cac phan tu sang phai de tao cho trong cho  
    // viec chen  
    for (int i = size; i > pos; i-)  
        array[i] = array[i - 1];  
    // Dat phan tu moi vao vi tri chen  
    array[pos] = newElement;  
    // Tang kich thuoc  
    size++;  
}
```



1. Vectors

1. 3. Cài đặt demo class Vector

- **Hàm xóa phần tử tại vị trí pos:**

```
void erase(int pos){  
    // Dich cac phan tu sang trai de lap day cho trong de  
    // lai do vao  
    for (int i = pos; i < size - 1; i++)  
        array[i] = array[i + 1];  
    // Giam kich thuoc  
    size-;  
}
```

- **Hàm xóa tất cả phần tử của vector:**

```
void clear(){  
    size = 0;  
}
```



1. Vectors

1. 3. Cài đặt demo class Vector

```
int main(){  
    vector<int> a;  
    a.push_back(1);  
    a.push_back(2);  
    a.pop_back();  
    a.insert(0,2);  
    a.clear();  
    return 0;  
}
```



Nội dung

Vectors

Danh sách liên kết

- 2.1 Giới thiệu
- 2.2 Khai báo
- 2.3 Các thao tác cơ bản
- 2.4 Bài tập
- 2.5 Sắp xếp trên DSLK đơn
- 2.6 Danh sách liên kết đôi (Double Linked List)
- 2.7 Danh sách liên kết vòng (Circular Linked List)

Trao đổi



2. Danh sách liên kết

2. 1. Giới thiệu

Kiểu dữ liệu tĩnh

- Kiểu dữ liệu tĩnh: Một số đối tượng dữ liệu không thay đổi được kích thước, cấu trúc, ... trong suốt quá trình tồn tại → kiểu dữ liệu tĩnh.
- Một số kiểu dữ liệu tĩnh: các cấu trúc dữ liệu được xây dựng từ các kiểu cơ sở như kiểu thực, kiểu nguyên, kiểu ký tự ... hoặc từ các cấu trúc đơn giản như mảng tin, tập hợp, mảng ...
- Các đối tượng dữ liệu thuộc kiểu này thường cứng ngắt, gò bó → khó diễn tả được thực tế vốn sinh động, phong phú.
- Dữ liệu tĩnh sẽ chiếm vùng nhớ đã dành cho chúng suốt quá trình thực thi chương trình → sử dụng bộ nhớ kém hiệu quả.



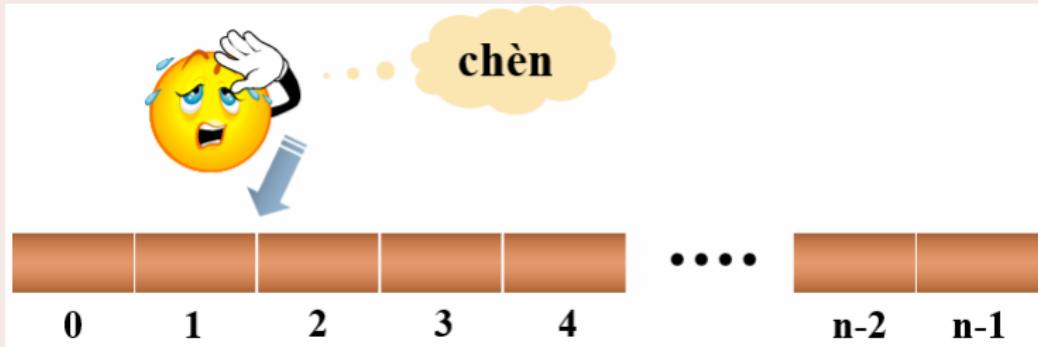
2. Danh sách liên kết

2. 1. Giới thiệu

Ví dụ 2.1

Cấu trúc dữ liệu tinh: Mảng 1 chiều

- Kích thước cố định (fixed size)
- Chèn 1 phần tử vào mảng rất khó
- Các phần tử tuân tự theo chỉ số $0 \rightarrow n - 1$
- Truy cập ngẫu nhiên (random access)



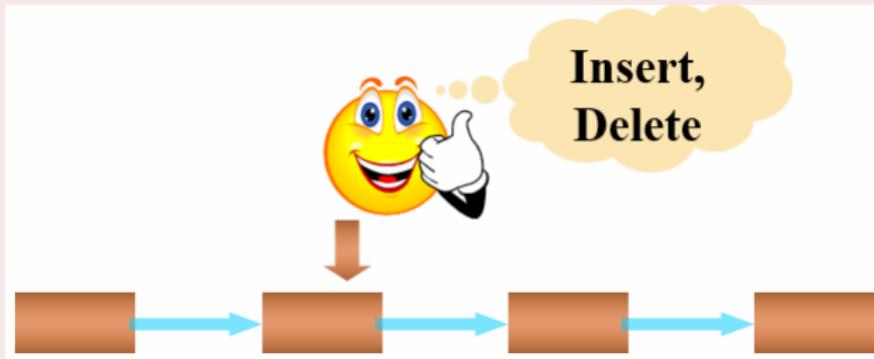


2. Danh sách liên kết

2. 1. Giới thiệu

Cấu trúc dữ liệu động: danh sách liên kết, cây

- Cấp phát động lúc chạy chương trình
- Các phần tử nằm rải rác ở nhiều nơi trong bộ nhớ
- Kích thước danh sách chỉ bị giới hạn do RAM
- Thao tác thêm xoá đơn giản





2. Danh sách liên kết

2. 1. Giới thiệu

So sánh Mảng và Danh sách liên kết

Nội dung	Mảng	Danh sách liên kết
Kích thước	- Kích thước cố định - Cần chỉ rõ kích thước trong khi khai báo	- Kích thước thay đổi trong quá trình thêm/xóa phần tử, chuyên cần - Kích thước tối đa phụ thuộc vào bộ nhớ
Cấp phát bộ nhớ	Tĩnh: Bộ nhớ được cấp phát trong quá trình biên dịch	Động: Bộ nhớ được cấp phát trong quá trình chạy
Thứ tự & sắp xếp	Được lưu trữ trên một dãy ô nhớ liên tục	Được lưu trữ trên các ô nhớ ngẫu nhiên
Truy cập	Truy cập tới phần tử ngẫu nhiên trực tiếp bằng cách sử dụng chỉ số mảng: $O(1)$	Truy cập tới phần tử ngẫu nhiên cần phải duyệt từ đầu/cuối đến phần tử đó: $O(n)$
Tìm kiếm	Tìm kiếm tuyến tính hoặc tìm kiếm nhị phân	Chỉ có thể tìm kiếm tuyến tính

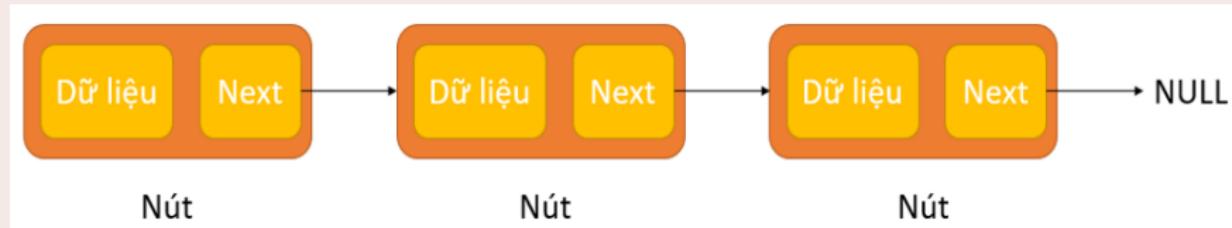


2. Danh sách liên kết

2. 1. Giới thiệu

Danh sách liên kết:

- Mỗi phần tử của danh sách gọi là node (nút)
- Mỗi node có 2 thành phần: *phần dữ liệu* và *phần liên kết* chứa địa chỉ của node kế tiếp hay node trước nó



- Các thao tác cơ bản trên danh sách liên kết:

- Thêm một phần tử mới
- Xóa một phần tử
- Tìm kiếm
- ...



2. Danh sách liên kết

2. 1. Giới thiệu

Có một số cách tổ chức liên kết giữa các phần tử trong danh sách như:

- Danh sách liên kết đơn
- Danh sách liên kết kép
- Danh sách liên kết vòng



2. Danh sách liên kết

2. 1. Giới thiệu

- **Danh sách liên kết đơn:** mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách.





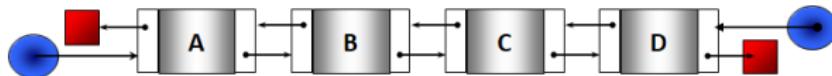
2. Danh sách liên kết

2. 1. Giới thiệu

- **Danh sách liên kết đơn:** mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách.



- **Danh sách liên kết đôi:** mỗi phần tử liên kết với các phần tử đứng trước và sau nó trong danh sách.





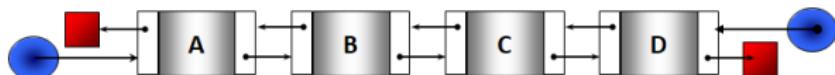
2. Danh sách liên kết

2. 1. Giới thiệu

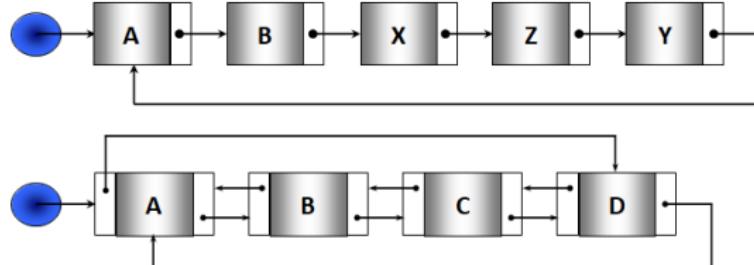
- **Danh sách liên kết đơn:** mỗi phần tử liên kết với phần tử đứng sau nó trong danh sách.



- **Danh sách liên kết đôi:** mỗi phần tử liên kết với các phần tử đứng trước và sau nó trong danh sách.



- **Danh sách liên kết vòng:** phần tử cuối danh sách liên kết với phần tử đầu danh sách.





2. Danh sách liên kết

2. 1. Giới thiệu

Biến tĩnh

- Được khai báo tường minh
- Tồn tại khi vào phạm vi khai báo và chỉ mất khi ra khỏi phạm vi này
- Được cấp phát vùng nhớ trong vùng dữ liệu (Data segment) hoặc là Stack (các biến cục bộ)
- Kích thước không thay đổi trong suốt quá trình tồn tại
- Do được khai báo tường minh, các biến tĩnh có một định danh đã được kết nối với địa chỉ vùng nhớ lưu trữ biến và được truy xuất trực tiếp thông qua định danh đó.

```
int a;      // a, b la cac bien tinh
char b[10];
```



2. Danh sách liên kết

2. 1. Giới thiệu

Biến động

- Trong nhiều trường hợp, tại thời điểm biên dịch không thể xác định trước kích thước chính xác của một số đối tượng dữ liệu do sự tồn tại và tăng trưởng của chúng phụ thuộc vào ngữ cảnh của việc thực hiện chương trình.
- Các đối tượng dữ liệu có đặc điểm kể trên nên được khai báo như biến động:
 - Biến không được khai báo tường minh
 - Có thể được cấp phát hoặc giải phóng bộ nhớ khi người sử dụng yêu cầu
 - Các biến này không theo qui tắc phạm vi
 - Vùng nhớ của biến được cấp phát trong Heap
 - Kích thước có thể thay đổi trong quá trình tồn tại



2. Danh sách liên kết

2. 1. Giới thiệu

Biến động (tiếp ...)

- Do không được khai báo tường minh nên các biến động không có một định danh được kết buộc với địa chỉ vùng nhớ cấp phát cho nó, do đó gặp khó khăn khi truy xuất đến một biến động.
- Để giải quyết vấn đề, *biến con trỏ* (là biến không động) được sử dụng để trỏ đến biến động.
- Khi tạo ra một biến động, phải dùng một con trỏ để lưu địa chỉ của biến này và sau đó, truy xuất đến biến động thông qua biến con trỏ đã biết định danh.
- Hai thao tác cơ bản trên biến động là **tạo** và **hủy** một biến động do biến con trỏ '*p*' trỏ đến:
 - Tạo ra một biến động và cho con trỏ *p* chỉ đến nó
 - Hủy một biến động do *p* chỉ đến



2. Danh sách liên kết

2. 1. Giới thiệu

Biến động (tiếp ...)

- Tạo ra một biến động và cho con trỏ '*p*' chỉ đến nó

```
1     void* malloc(size);      // tra ve con tro chi den vung
     nho size byte vua duoc cap phat.
2     void* calloc(n,size); ); // tra ve con tro chi den vung
     nho vua duoc cap phat gom n phan tu,
     // moi phan tu co kich thuoc size
     // byte
3
4     new                      // toan tu cap phat bo nho trong
     C++
```

- Hàm **free**(*p*) huỷ vùng nhớ cấp phát bởi hàm malloc hoặc calloc do *p* trả tới
- Toán tử **delete** *p* huỷ vùng nhớ cấp phát bởi toán tử **new** do *p* trả tới



2. Danh sách liên kết

2. 1. Giới thiệu

Ví dụ 2.2

```
int *p1, *p2; // cap phat vung nho cho 1 bien dong kieu int
p1 = (int*)malloc(sizeof(int)); // dat gia tri 5 cho bien dong
    dang duoc p1 quan ly
// cap phat bien dong kieu mang gom 10 phan tu kieu int
*p1 = 5;
p2 = (int*)calloc(10, sizeof(int));
*(p2+3) = 0; // dat gia tri 0 cho phan tu thu 4 cua mang p2
free(p1);
free(p2);
```

- Kiểu con trỏ là kiểu dữ liệu cơ sở dùng lưu địa chỉ của một đối tượng dữ liệu khác.
- Biến thuộc kiểu con trỏ T là biến mà giá trị của nó là địa chỉ của một vùng nhớ ứng với một biến kiểu T, hoặc là giá trị NULL.

`int i *p`

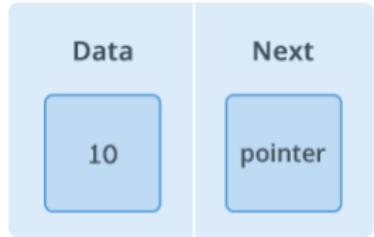


2. Danh sách liên kết

2. 2. Khai báo

Là danh sách các node mà mỗi node có 2 thành phần:

- Thành phần *dữ liệu*: lưu trữ các thông tin về bản thân phần tử
- Thành phần *mỗi liên kết*: lưu trữ địa chỉ của phần tử kế tiếp trong danh sách, hoặc lưu trữ giá trị NULL nếu là phần tử cuối danh sách
- Khai báo node



```
1 struct Node{  
2     DataType data;    // DataType là kiểu dữ định nghĩa trước  
3     Node *pNext;      // con trỏ chỉ đến cấu trúc Node  
4 };
```



2. Danh sách liên kết

2. 2. Khai báo

Ví dụ 2.3

Khai báo node lưu số nguyên:

```
1 struct Node{  
2     int data;  
3     Node *pNext;  
4 };
```

Ví dụ 2.4

Định nghĩa một phần tử trong danh sách lưu trữ hồ sơ sinh viên:

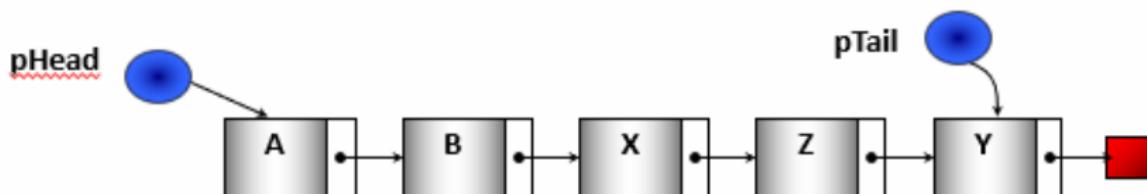
```
1 struct SinhVien{  
2     char Ten[30];  
3     int MaSV;  
4 };  
5 struct SVNode {  
6     SinhVien data;  
7     SVNode *pNext;  
8 };
```



2. Danh sách liên kết

2. 2. Khai báo

- Để quản lý một DSLK đơn chỉ cần biết địa chỉ phần tử đầu danh sách
- Con trỏ **pHead** sẽ được dùng để lưu trữ địa chỉ phần tử đầu danh sách. Ta có khai báo:
Node *pHead;
- Để tiện lợi, có thể sử dụng thêm một con trỏ **pTail** giữ địa chỉ phần tử cuối danh sách. Khai báo **pTail** như sau:
Node *pTail;





2. Danh sách liên kết

2. 2. Khai báo

Ví dụ 2.5

Khai báo cấu trúc 1 DSLK đơn chứa số nguyên

```
struct Node{    // kieu cua mot phan tu trong danh sach
    int    data;
    Node*  pNext;
};

struct List{    // kieu danh sach lien ket
    Node* pHead;
    Node* pTail;
};
List l;          //Khai bao bien kieu danh sach
```

2. Danh sách liên kết

2. 2. Khai báo

- Thủ tục **getNode** để tạo ra một nút cho danh sách với thông tin chứa trong x

```
1 Node* getNode (DataType x){  
2     Node *p;  
3     p = new Node; // Cap phat vung nho cho node  
4     if (p==NULL){  
5         cout<<"Khong du bo nho!"; return NULL;  
6     }  
7     p->data = x; // Gan du lieu cho phan tu p  
8     p->pNext = NULL;  
9     return p;  
10 }
```

- Để tạo một phần tử mới cho danh sách, cần thực hiện câu lệnh:
 $\text{new_node} = \text{getNode}(x);$
→ new_node sẽ quản lý địa chỉ của phần tử mới được tạo.



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

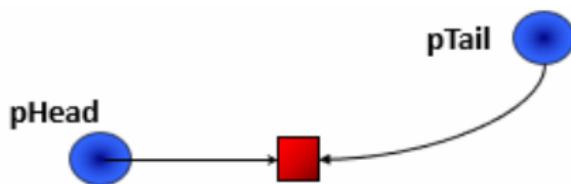
- Tạo danh sách rỗng
- Thêm một phần tử vào danh sách
- Duyệt danh sách
- Tìm kiếm một giá trị trên danh sách
- Xóa một phần tử ra khỏi danh sách
- Hủy toàn bộ danh sách
- ...



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

- Tạo danh sách rỗng



```
1 void Init(List &l){  
2     l.pHead = l.pTail = NULL;  
3 }
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

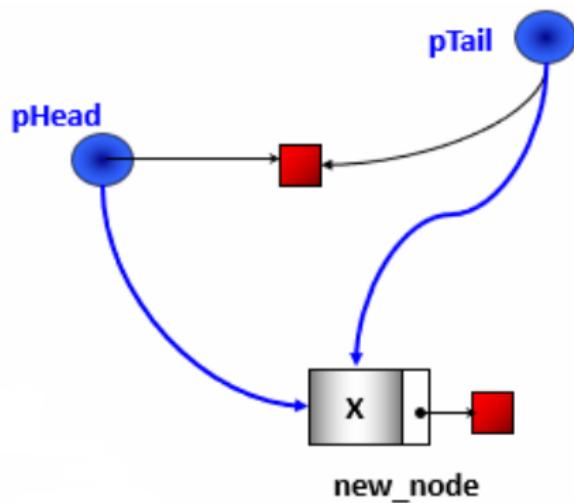
- **Thêm một phần tử vào danh sách:** có 3 vị trí thêm
 - Gắn vào đầu danh sách
 - Gắn vào cuối danh sách
 - Chèn vào sau nút q trong danh sách
- Chú ý trường hợp danh sách ban đầu rỗng



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

- Nếu danh sách ban đầu rỗng



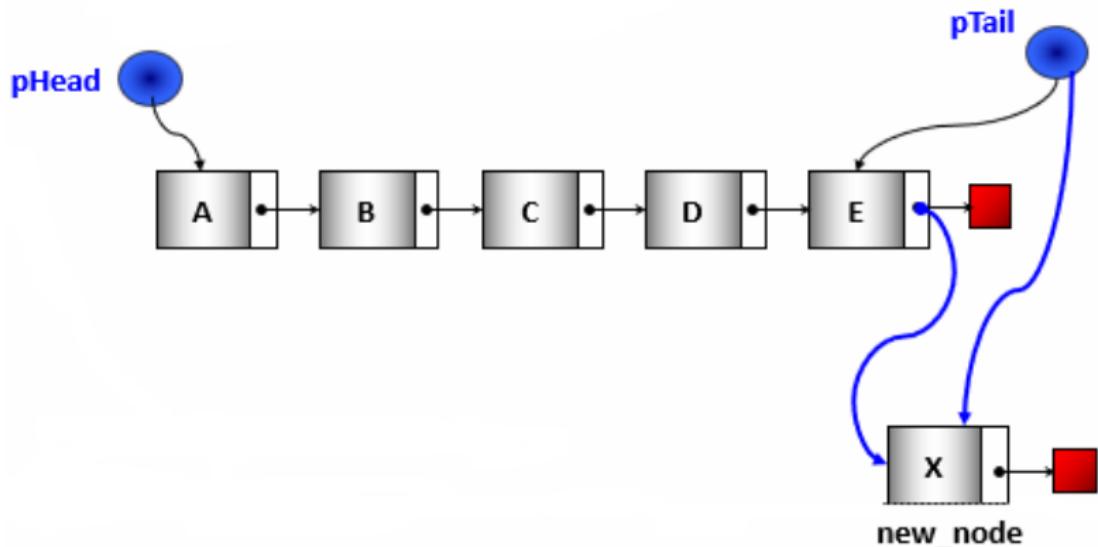
```
1 pHad = pTail = new_node;
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

- Nếu danh sách ban đầu không rỗng: gắn node vào cuối danh sách



```
1 pTail->pNext = new_node;  
2 pTail = new_node;
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

```
void addTail(List &l, Node *new_node){  
    if (l.pHead == NULL){  
        l.pHead = l.pTail = new_node;  
    }  
    else{  
        l.pTail->pNext = new_node;  
        l.pTail = new_node ;  
    }  
}
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

Ví dụ 2.6

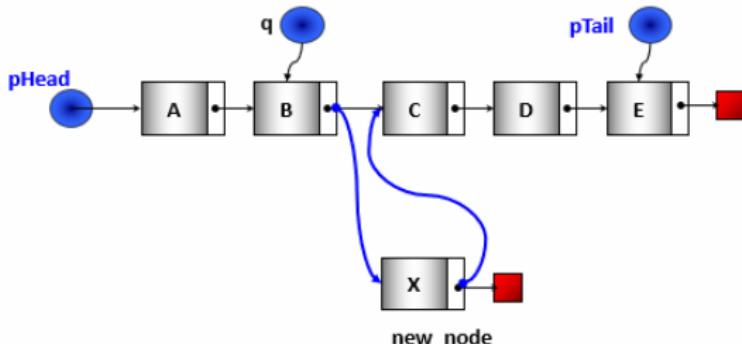
Thêm một số nguyên vào cuối danh sách

```
int x;
cout<<"Nhập x=";    // Nhập dữ liệu cho x
cin>>x;
Node* p = getNode(x); // Tạo nút mới
if (p != NULL)
    addTail(l, p);    // Gán nút vào cuối ds
```



2. Danh sách liên kết

- Chèn một phần tử sau q



```
1 void addAfter (List &l, Node *q, Node* new_node){  
2     if (q!=NULL){  
3         new_node->pNext = q->pNext;  
4         q->pNext = new_node;  
5         if(q == l.pTail)  
6             l.pTail = new_node;  
7     }  
8 }
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

- **Duyệt danh sách:** là thao tác thường được thực hiện khi có nhu cầu xử lý các phần tử của danh sách theo cùng một cách thức hoặc khi cần lấy thông tin tổng hợp từ các phần tử của danh sách như:
 - Đếm các phần tử của danh sách
 - Tìm tất cả các phần tử thỏa điều kiện
 - Hủy toàn bộ danh sách (và giải phóng bộ nhớ)
 - ...



```
void processList (List l){  
    Node *p = l.pHead; // Cho p tro den phan tu dau danh sach  
    while (p!= NULL){  
        // xu ly cu the p tuy ung dung  
        p = p->pNext; // Cho p tro toi phan tu ke  
    }  
}
```

Ví dụ 2.7 (In các phần tử trong danh sách)

```
void Output (List l){  
    Node* p=l.pHead;  
    while (p!=NULL) {  
        cout<<p->data<<''\t'';  
        p=p ->pNext;  
    }  
    cout<<endl;  
}
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

- Tìm kiếm một phần tử có khóa x

```
1 Node* Search (List l, int x){  
2     Node* p = l.pHead;  
3     while (p!=NULL){  
4         if (p->data==x)  
5             return p;  
6         p=p->pNext;  
7     }  
8     return NULL;  
9 }
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

- Xóa một node của danh sách
 - Xóa node đầu của danh sách
 - Xóa node sau node q trong danh sách
 - Xóa node có khoá k

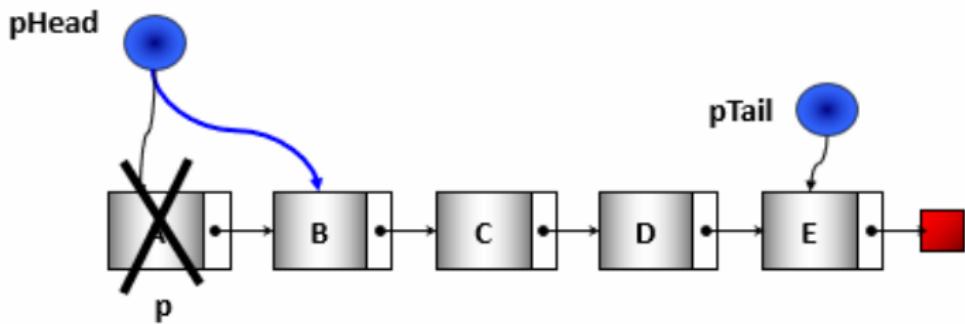


2. Danh sách liên kết

2. 3. Các thao tác cơ bản

- Xóa node đầu của danh sách

- Gọi p là node đầu của danh sách (pHead)
- Cho pHead trỏ vào node sau node p (là p->pNext)
- Nếu danh sách trở thành rỗng thì pTail = NULL
- Giải phóng vùng nhớ mà p trỏ tới





2. Danh sách liên kết

2. 3. Các thao tác cơ bản

```
int removeHead (List &l){  
    if (l.pHead == NULL) return 0;  
    Node* p=l.pHead;  
    l.pHead = p->pNext;  
    if (l.pHead == NULL) l.pTail=NULL; //Neu danh sach rong  
    delete p;  
    return 1;  
}
```

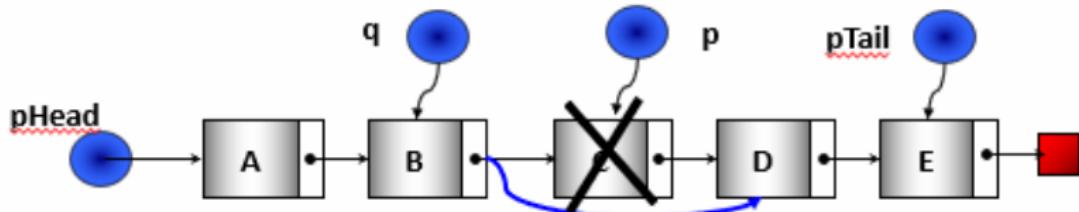


2. Danh sách liên kết

2. 3. Các thao tác cơ bản

Xóa node sau node q trong danh sách

- Điều kiện để có thể xóa được node sau q là:
 - q phải khác NULL ($q \neq \text{NULL}$)
 - Node sau q phải khác NULL ($q->pNext \neq \text{NULL}$)
- Các thao tác:
 - Gọi p là node sau q
 - Cho vùng pNext của q trỏ vào node đứng sau p
 - Nếu p là phần tử cuối thì pTail trỏ vào q
 - Giải phóng vùng nhớ mà p trỏ tới





2. Danh sách liên kết

2. 3. Các thao tác cơ bản

```
int removeAfter (List &l, Node *q ){
    if (q !=NULL && q->pNext !=NULL){
        Node* p = q->pNext;
        q->pNext = p->pNext;
        if (p==l.pTail) l.pTail = q;
        delete p;
        return 1;
    }
    else return 0;
}
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

Xoá node chứa khoá k trong danh sách

- Tìm phần tử p có khóa k và phần tử q đứng trước nó
- Ngược lại, thông báo không có k

```
int removeNode(List &l, int k){  
    Node *p=l.pHead; Node *q=NULL;  
    while (p != NULL){  
        if (p->data == k) break;  
        q = p;  
        p = p->pNext;  
    }  
    if (p == NULL){  
        cout<< "Khong tim thay k"; return 0;  
    }  
    else if (q == NULL)  
        // thuc hien xoa phan tu dau ds la p  
    else  
        // thuc hien xoa phan tu p sau q
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

Hủy toàn bộ danh sách

- Để hủy toàn bộ danh sách, thao tác xử lý bao gồm hành động giải phóng một phần tử, do vậy phải cập nhật các liên kết liên quan
- Các thao tác:
 - Trong khi danh sách chưa hết thực hiện:
 $p = pHead;$ $pHead = pHead ->pNext;$ Hủy $p;$
 - $pTail = NULL;$

```
void RemoveList (List &l){  
    Node *p;  
    while (l.pHead != NULL){  
        p = l.pHead;  
        l.pHead = p->pNext;  
        delete p;  
    }  
    l.pTail = NULL;  
}
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

- Đếm số nút trong danh sách:

```
1 int CountNodes (List &l){  
2     int count = 0;  
3     Node *p = l.pHead;  
4     while (p!=NULL) {  
5         count++;  
6         p = p->pNext;  
7     }  
8     return count;  
9 }
```



2. Danh sách liên kết

2. 3. Các thao tác cơ bản

- Trích phần tử đầu danh sách:

```
1 Node* PickHead (List &l){  
2     Node *p = NULL;  
3     if (l.pHead != NULL){  
4         p = l.pHead;  
5         l.pHead = l.pHead->pNext;  
6         p->pNext = NULL;  
7         if (l.pHead == NULL) l.pTail = NULL;  
8     }  
9     return p;  
10 }
```



2. Danh sách liên kết

2. 4. Bài tập

Write a program for building single linked list (Display menu)

- a) Add one node at first
- b) Add one node at last
- c) Add many node at first
- d) Add many node at last
- e) Add one node after select node
- f) Display List
- g) Find one node
- h) Select and display n(th) node
- i) Display node count
- j) Remove one node
- k) Remove List
- l) Get sum of all nodes
- m) Insert a new node in a sorted list



2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Cách tiếp cận:

- Phương án 1: Hoán vị nội dung các phần tử trong danh sách (thao tác trên vùng data).
 - Cài đặt lại trên danh sách liên kết một trong những thuật toán sắp xếp đã biết trên mảng
 - Điểm khác biệt duy nhất là cách thức truy xuất đến các phần tử trên danh sách liên kết thông qua liên kết thay vì chỉ số như trên mảng.
 - Khi kích thước của trường data lớn, việc hoán vị giá trị của hai phân tử sẽ chiếm chi phí đáng kể (do đòi hỏi sử dụng thêm vùng nhớ trung gian).
- Phương án 2: Thay đổi các mối liên kết (thao tác trên vùng link)
 - Thay vì hoán đổi giá trị, ta sẽ tìm cách thay đổi trình tự móc nối của các phần tử sao cho tạo lập nên được thứ tự mong muốn → chỉ thao tác trên các móc nối (link), không phụ thuộc kích thước trường data.
 - Thao tác trên các móc nối thường phức tạp hơn thao tác trực tiếp trên dữ liệu.

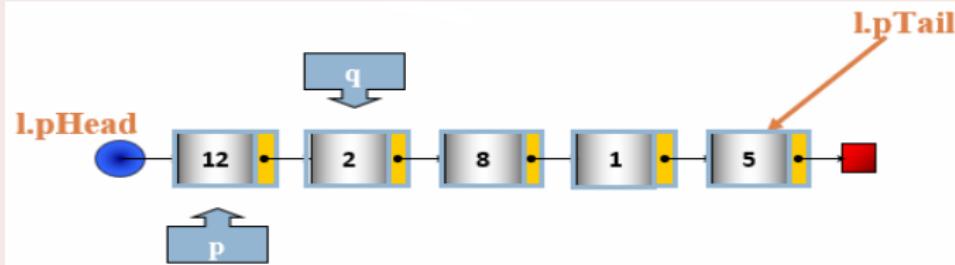


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.8 (Interchange Sort)

```
void SLL_InterChangeSort ( List &l ){
    for ( Node* p=l.pHead ; p!=l.pTail; p=p->pNext )
        for ( Node* q=p->pNext; q!=NULL ; q=q->pNext )
            if ( p->data > q->data )
                Swap( p->data , q->data );
}
```



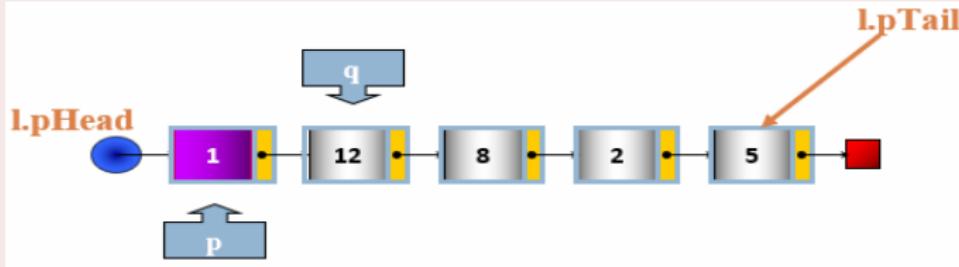


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.8 (Interchange Sort)

```
void SLL_InterChangeSort ( List &l ){
    for ( Node* p=l.pHead ; p!=l.pTail; p=p->pNext )
        for ( Node* q=p->pNext; q!=NULL ; q=q->pNext )
            if ( p->data > q->data )
                Swap( p->data , q->data );
}
```



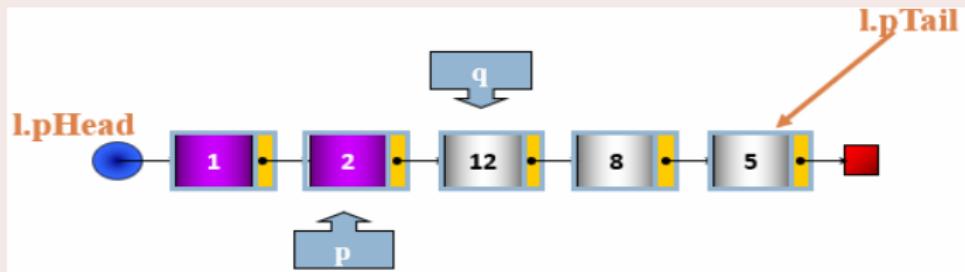


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.8 (Interchange Sort)

```
void SLL_InterChangeSort ( List &l ){
    for ( Node* p=l.pHead ; p!=l.pTail; p=p->pNext )
        for ( Node* q=p->pNext; q!=NULL ; q=q->pNext )
            if ( p->data > q->data )
                Swap( p->data , q->data );
}
```



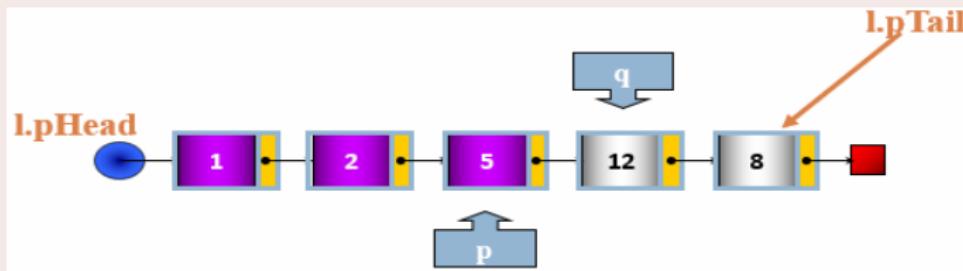


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.8 (Interchange Sort)

```
void SLL_InterChangeSort ( List &l ){
    for ( Node* p=l.pHead ; p!=l.pTail; p=p->pNext )
        for ( Node* q=p->pNext; q!=NULL ; q=q->pNext )
            if ( p->data > q->data )
                Swap( p->data , q->data );
}
```



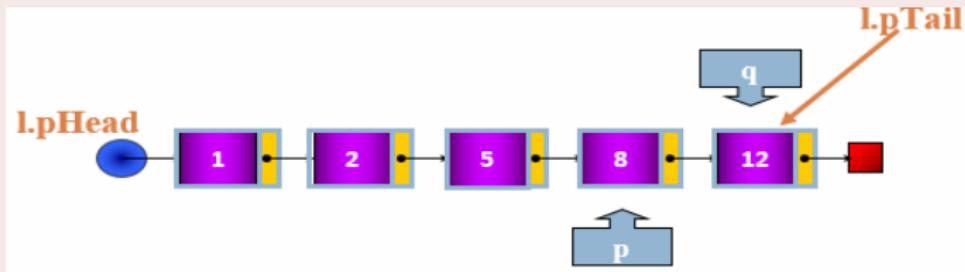


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.8 (Interchange Sort)

```
void SLL_InterChangeSort ( List &l ){
    for ( Node* p=l.pHead ; p!=l.pTail; p=p->pNext )
        for ( Node* q=p->pNext; q!=NULL ; q=q->pNext )
            if ( p->data > q->data )
                Swap( p->data , q->data );
}
```



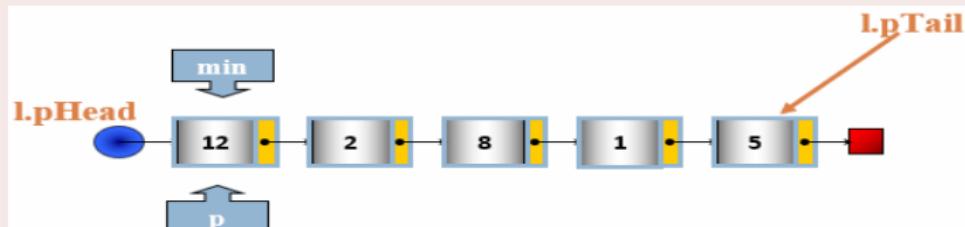


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.9 (SelectionSort)

```
void SLL_SelectionSort (List &l){
    for ( Node* p = l.pHead ; p != l.pTail ; p = p->pNext ){
        Node* min = p;
        for ( Node* q = p->pNext ; q != NULL ; q = q->pNext )
            if ( min->data > q->data ){
                min = q ;
                Swap(min->data, p->data);
            }
    }
}
```



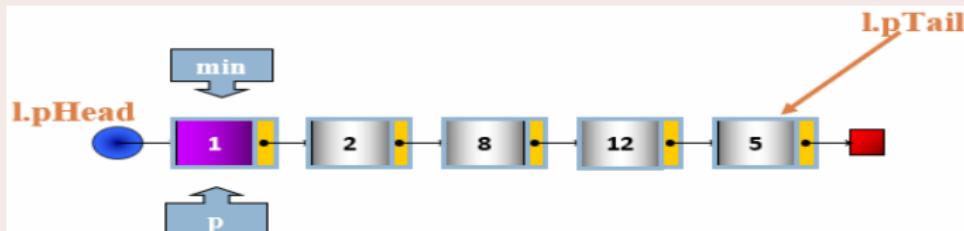


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.9 (SelectionSort)

```
void SLL_SelectionSort (List &l){  
    for ( Node* p = l.pHead ; p != l.pTail ; p = p->pNext ){  
        Node* min = p;  
        for ( Node* q = p->pNext ; q != NULL ; q = q->pNext )  
            if ( min->data > q->data ){  
                min = q ;  
                Swap(min->data, p->data);  
            }  
    }  
}
```



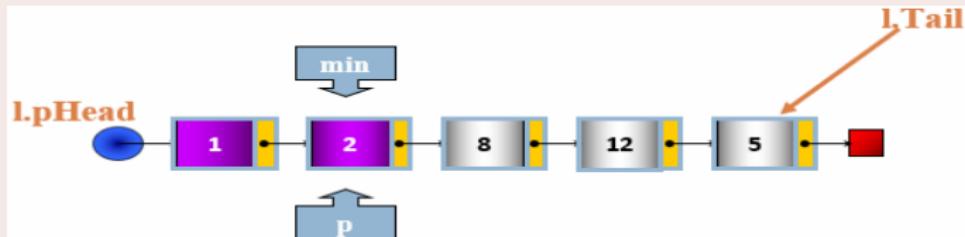


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.9 (SelectionSort)

```
void SLL_SelectionSort (List &l){  
    for ( Node* p = l.pHead ; p != l.pTail ; p = p->pNext ){  
        Node* min = p;  
        for ( Node* q = p->pNext ; q != NULL ; q = q->pNext )  
            if ( min->data > q->data ){  
                min = q ;  
                Swap(min->data, p->data);  
            }  
    }  
}
```



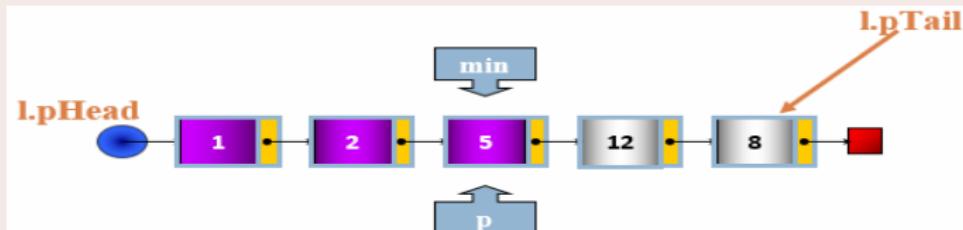


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.9 (SelectionSort)

```
void SLL_SelectionSort (List &l){
    for ( Node* p = l.pHead ; p != l.pTail ; p = p->pNext ){
        Node* min = p;
        for ( Node* q = p->pNext ; q != NULL ; q = q->pNext )
            if ( min->data > q->data ){
                min = q ;
                Swap(min->data, p->data);
            }
    }
}
```



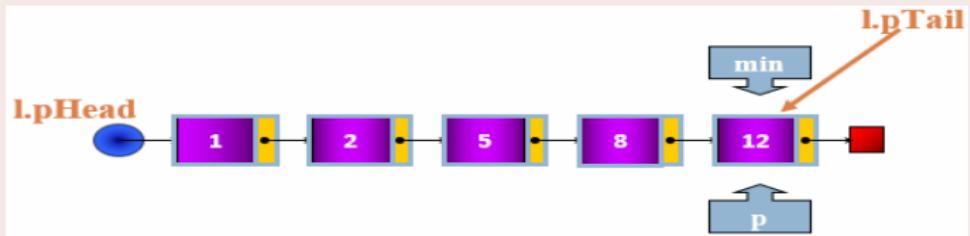


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.9 (SelectionSort)

```
void SLL_SelectionSort (List &l){  
    for ( Node* p = l.pHead ; p != l.pTail ; p = p->pNext ){  
        Node* min = p;  
        for ( Node* q = p->pNext ; q != NULL ; q = q->pNext )  
            if ( min->data > q->data ){  
                min = q ;  
                Swap(min->data, p->data);  
            }  
    }  
}
```



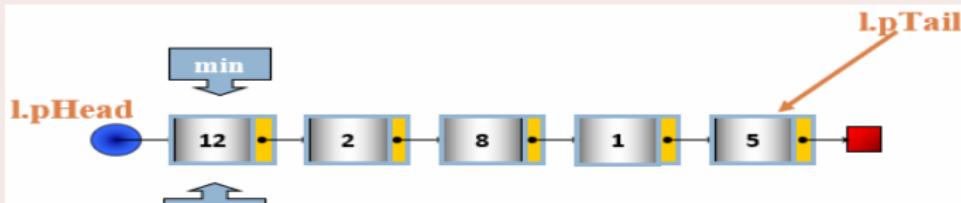


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.10 (BubbleSort)

```
void SLL_BubbleSort (List &l){  
    Node* t = l.pTail ;  
    for ( Node* p = l.pHead ; p != NULL ; p = p->pNext){  
        Node* t1;  
        for ( Node* q=l.pHead ; p!=t ; q=q->pNext){  
            if( q->data > q->pNext->data )  
                Swap( q->data , q->pNext->data );  
            t1 = q ;  
        }  
        t = t1;  
    }  
}
```



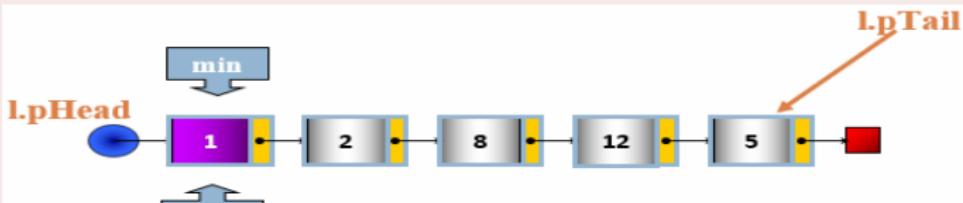


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.10 (BubbleSort)

```
void SLL_BubbleSort (List &l){  
    Node* t = l.pTail ;  
    for ( Node* p = l.pHead ; p != NULL ; p = p->pNext){  
        Node* t1;  
        for ( Node* q=l.pHead ; p!=t ; q=q->pNext){  
            if( q->data > q->pNext->data )  
                Swap( q->data , q->pNext->data );  
            t1 = q ;  
        }  
        t = t1;  
    }  
}
```



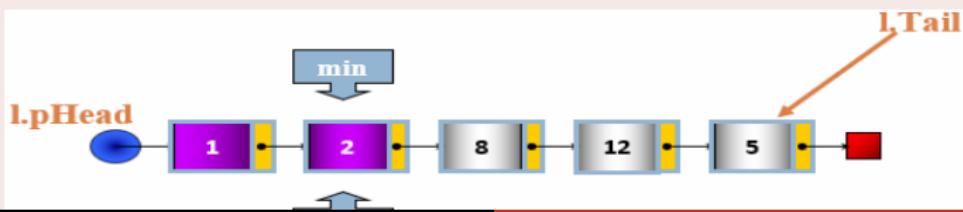


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.10 (BubbleSort)

```
void SLL_BubbleSort (List &l){  
    Node* t = l.pTail ;  
    for ( Node* p = l.pHead ; p != NULL ; p = p->pNext){  
        Node* t1;  
        for ( Node* q=l.pHead ; p!=t ; q=q->pNext){  
            if( q->data > q->pNext->data )  
                Swap( q->data , q->pNext->data );  
            t1 = q ;  
        }  
        t = t1;  
    }  
}
```



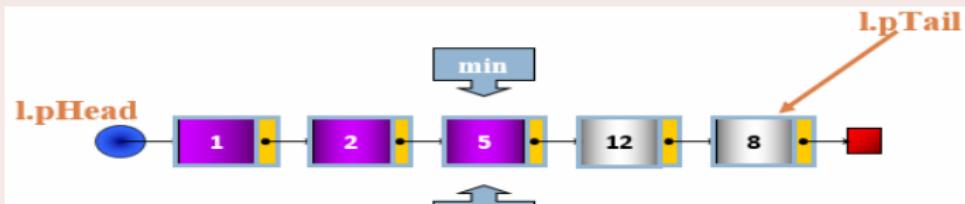


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.10 (BubbleSort)

```
void SLL_BubbleSort (List &l){  
    Node* t = l.pTail ;  
    for ( Node* p = l.pHead ; p != NULL ; p = p->pNext){  
        Node* t1;  
        for ( Node* q=l.pHead ; p!=t ; q=q->pNext){  
            if( q->data > q->pNext->data )  
                Swap( q->data , q->pNext->data );  
            t1 = q ;  
        }  
        t = t1;  
    }  
}
```



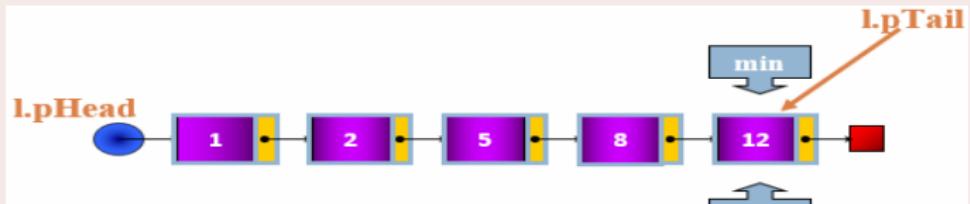


2. Danh sách liên kết

2. 5. Sắp xếp trên DSLK đơn

Ví dụ 2.10 (BubbleSort)

```
void SLL_BubbleSort (List &l){  
    Node* t = l.pTail ;  
    for ( Node* p = l.pHead ; p != NULL ; p = p->pNext){  
        Node* t1;  
        for ( Node* q=l.pHead ; p!=t ; q=q->pNext){  
            if( q->data > q->pNext->data )  
                Swap( q->data , q->pNext->data );  
            t1 = q ;  
        }  
        t = t1;  
    }  
}
```





Nội dung

Vectors

Danh sách liên kết

Trao đổi



TRAO ĐỔI