

DANH SÁCH LIÊN KẾT ĐƠN (SINGLY LINKED LIST)

DATA STRUCTURES & ALGORITHMS

ThS Nguyễn Thị Ngọc Diễm
diemntn@uit.edu.vn



- Kiểu dữ liệu trừu tượng (abstract data type – ADT)
 - Là một tập hợp các đối tượng đi kèm với một tập các thao tác (trên các đối tượng đó).
 - Là một dạng trừu tượng hóa toán học (mathematical abstractions).
 - **Abstraction** in [mathematics](#) is the process of extracting the underlying [essence](#) of a mathematical concept, removing any dependence on real world objects with which it might originally have been connected, and generalizing it so that it has wider applications or matching among other abstract descriptions of equivalent [phenomena](#).
- ADT không đề cập đến cách cài đặt các thao tác bằng ngôn ngữ lập trình cụ thể nào.

Kiểu dữ liệu trừu tượng List (ADT List)



- List: $A_0, A_1, A_2, \dots, A_{N-1}$
 - N được gọi là KÍCH THƯỚC (size) của list
 - List có kích thước bằng 0 được gọi là list rỗng
 - A_i theo sau A_{i-1}
 - A_{i-1} đứng trước A_i
- Các thao tác trên list:
 - Xuất danh sách (printList)
 - Xóa danh sách (makeEmpty)
 - Tìm phần tử thứ K (findKth)
 - Tìm kiếm một phần tử (find)
 - Chèn một phần tử (insert)
 - Xóa một phần tử (remove)

Cài đặt ADT List bằng C++

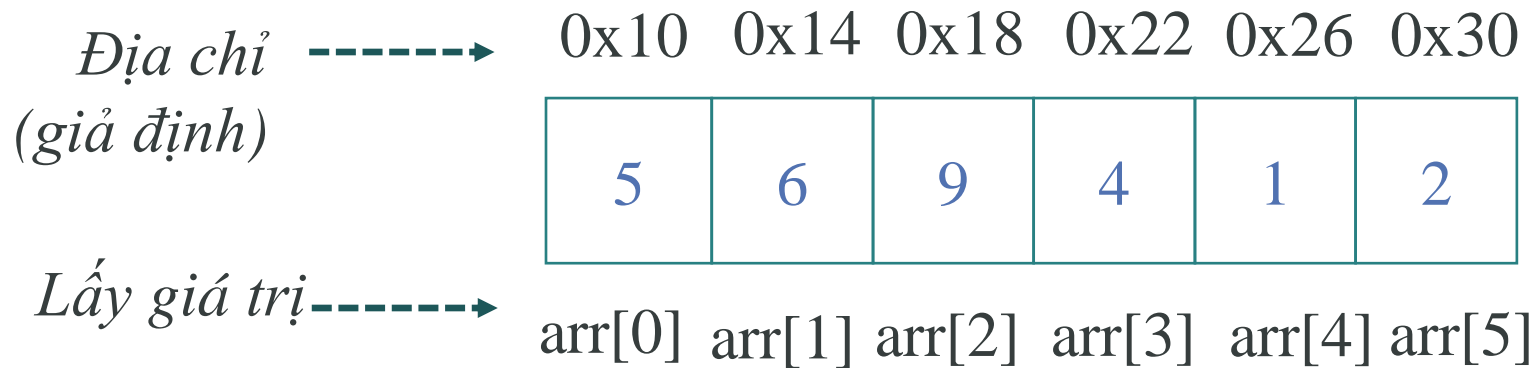


- Có 2 cách cài đặt ADT List bằng C++
 - Sử dụng mảng (array)
 - **Sử dụng danh sách liên kết**
 - Sinh viên sẽ tự cài đặt thử nghiệm
- Cài đặt sẵn trong thư viện STL (standard template library)
 - vector
 - list
 - Sinh viên tự học cách sử dụng thư viện này (không nằm trong môn học nào)
- Lưu ý: **tên hàm và số lượng hàm (thao tác) có thể khác trong ADT List nhưng đều đáp ứng các thao tác cơ bản.**

Nhắc lại: Lưu trữ Mảng 1 chiều



- Cho mảng 1 chiều: `int arr[6] = {5, 6, 9, 4, 1, 2};`



Memory Layout

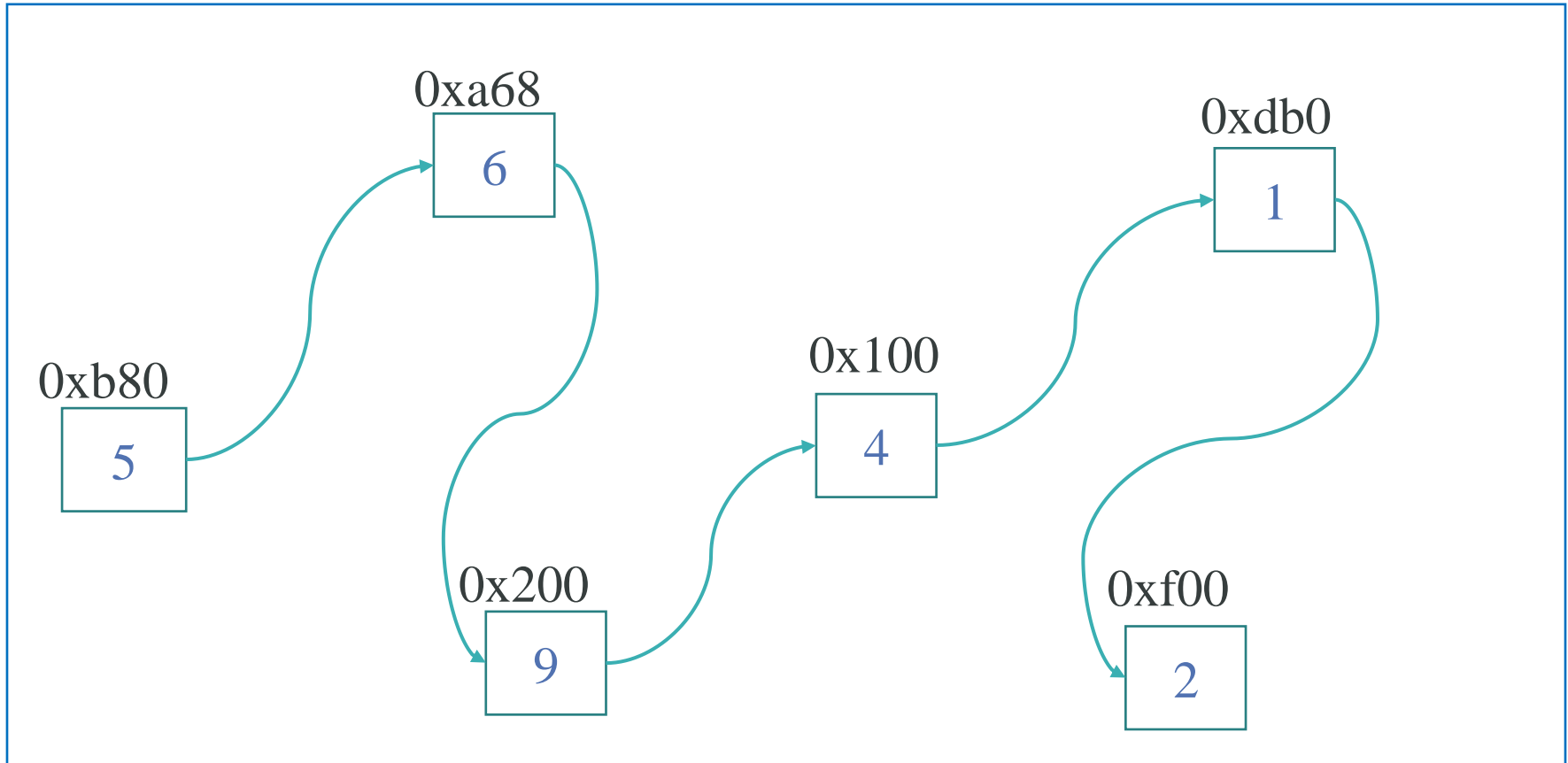


- Mỗi phần tử liên kết với phần tử đứng liền sau trong danh sách.
- Mỗi phần tử trong danh sách liên kết đơn là một cấu trúc có hai thành phần
 - **Thành phần dữ liệu:** Lưu trữ thông tin về bản thân phần tử
 - **Thành phần liên kết:** Lưu địa chỉ phần tử đứng sau trong danh sách hoặc bằng NULL nếu là phần tử cuối danh sách.

Tổ chức của DSLK Đơn



- Danh sách liên kết đơn lưu các giá trị {5, 6, 9, 4, 1, 2}

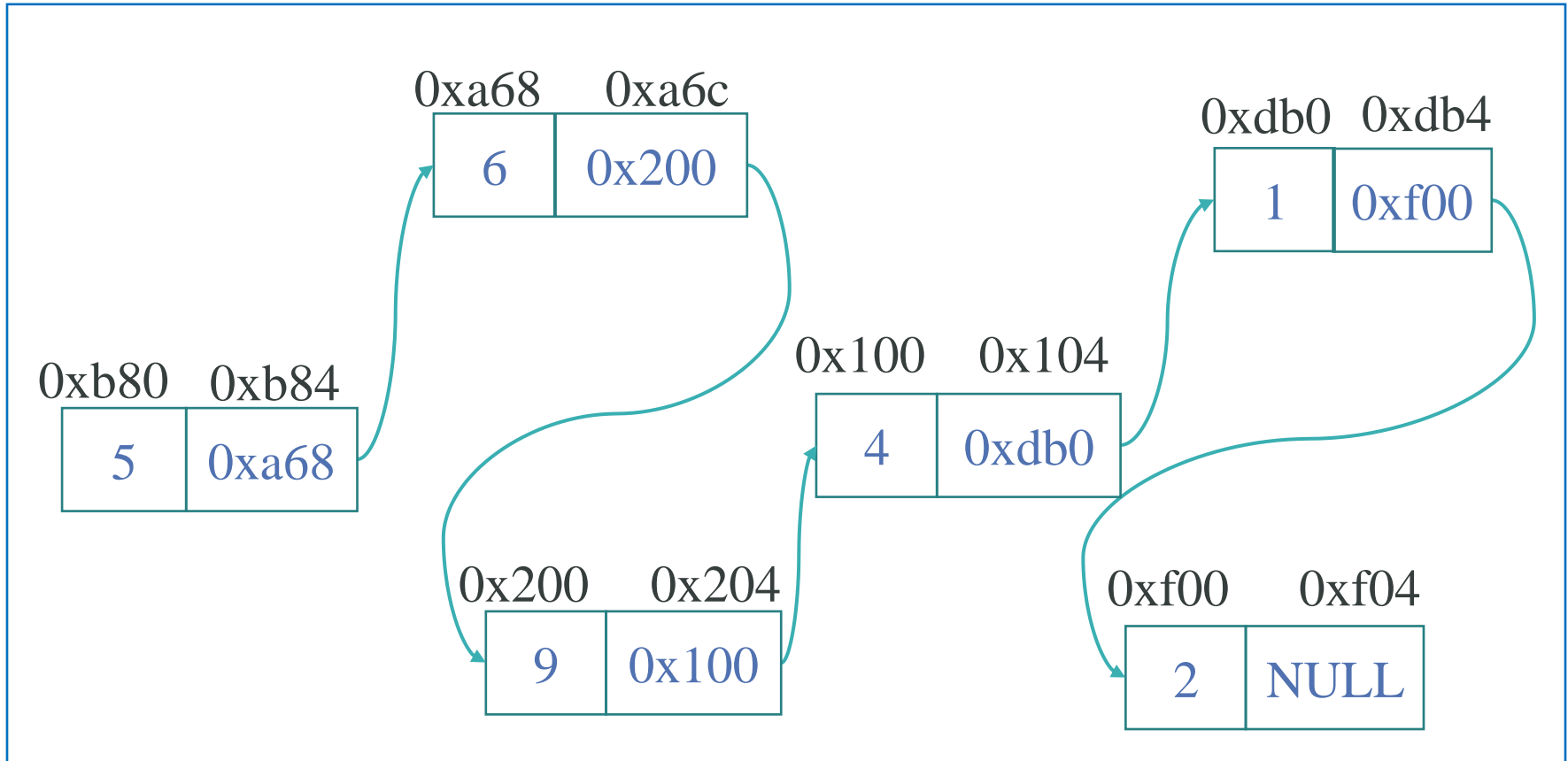


Memory Layout

Tổ chức của DSLK đơn



- Danh sách liên kết đơn lưu các giá trị {5, 6, 9, 4, 1, 2}



Memory Layout



- Cấu trúc dữ liệu của 1 node trong List đơn

```
struct NODE {  
    Data info;  
    NODE* pNext;  
};
```

- Cấu trúc dữ liệu của DSLK đơn

```
struct LIST {  
    NODE* pHead;  
    NODE* pTail;  
};
```



- Tạo danh sách liên kết đơn lưu các giá trị số tự nhiên.

// Cấu trúc của một node

```
struct NODE {  
    int info;  
    NODE* pNext;  
};
```

// Cấu trúc của một DSLK

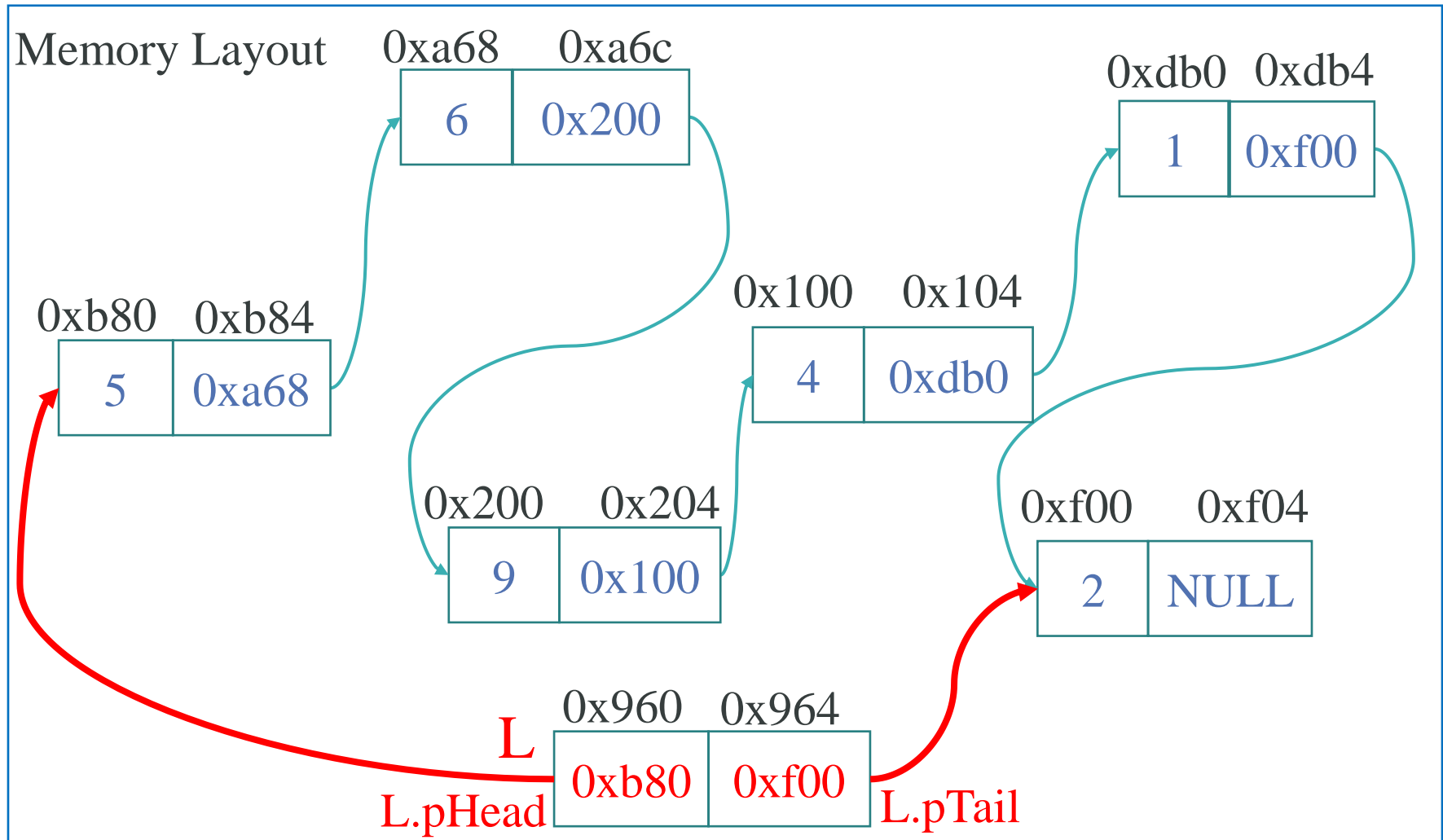
```
struct LIST {  
    NODE* pHead;  
    NODE* pTail;  
};
```

```
LIST L;
```

Ví dụ:



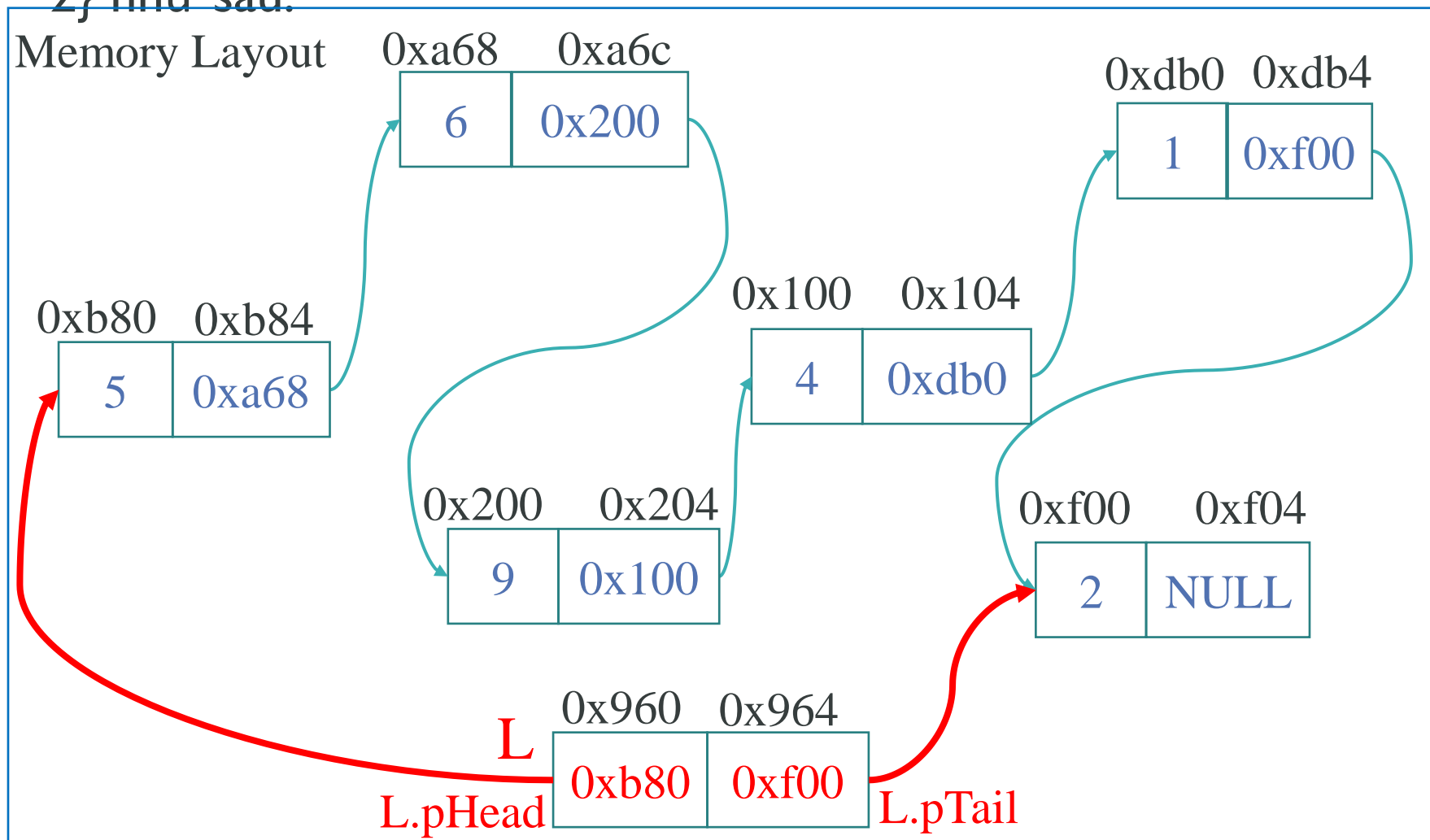
- o Danh sách liên kết đơn lưu các giá trị {5, 6, 9, 4, 1, 2}



Ví dụ:



- Quá trình tạo danh sách liên kết đơn lưu các giá trị {5, 6, 9, 4, 1, 2} như sau:





- Tạo 1 danh sách liên kết đơn rỗng
- Tạo 1 nút có trường **info** bằng x
- Thêm một phần tử có khóa x vào danh sách
- Duyệt danh sách
- Tìm một phần tử có **info** bằng x
- Hủy một phần tử trong danh sách
- Sắp xếp danh sách liên kết đơn
- ...

Khởi tạo danh sách liên kết RỖNG



- Địa chỉ của nút đầu tiên, địa chỉ của nút cuối cùng đều không có

```
void CreateEmptyList(LIST &l) {  
    l.pHead = NULL;  
    l.pTail = NULL;  
}
```



- Hàm trả về địa chỉ phần tử mới tạo

```
NODE* CreateNode(int x) {  
    NODE* p;  
    p = new NODE;  
    if (p == NULL)  
        exit(1);  
    p->info = x;  
    p->pNext = NULL;  
    return p;  
}
```



- **Nguyên tắc thêm:** Khi thêm 1 phần tử vào List thì có làm cho pHead, pTail thay đổi.
- **Các vị trí cần thêm 1 phần tử vào List:**
 - Thêm vào đầu List đơn
 - Thêm vào cuối List
 - Thêm vào sau 1 phần tử Q trong List



- Thêm nút p vào đầu danh sách liên kết đơn

Bắt đầu:

Nếu List rỗng thì

+ pHead = p;

+ pTail = pHead;

Ngược lại

+ p->pNext = pHead;

+ pHead = p



```
void AddHead(LIST &L, NODE* p) {  
    if (L.pHead == NULL) {  
        L.pHead = p;  
        L.pTail = L.pHead;  
    }  
    else {  
        p->pNext = L.pHead;  
        L.pHead = p;  
    }  
}
```



```
void AddHead(LIST &L, int x) {  
    NODE* p= CreateNode(x);  
    if (L.pHead == NULL) {  
        L.pHead = p;  
        L.pTail = L.pHead;  
    }  
    else {  
        p->pNext = L.pHead;  
        L.pHead = p;  
    }  
}
```

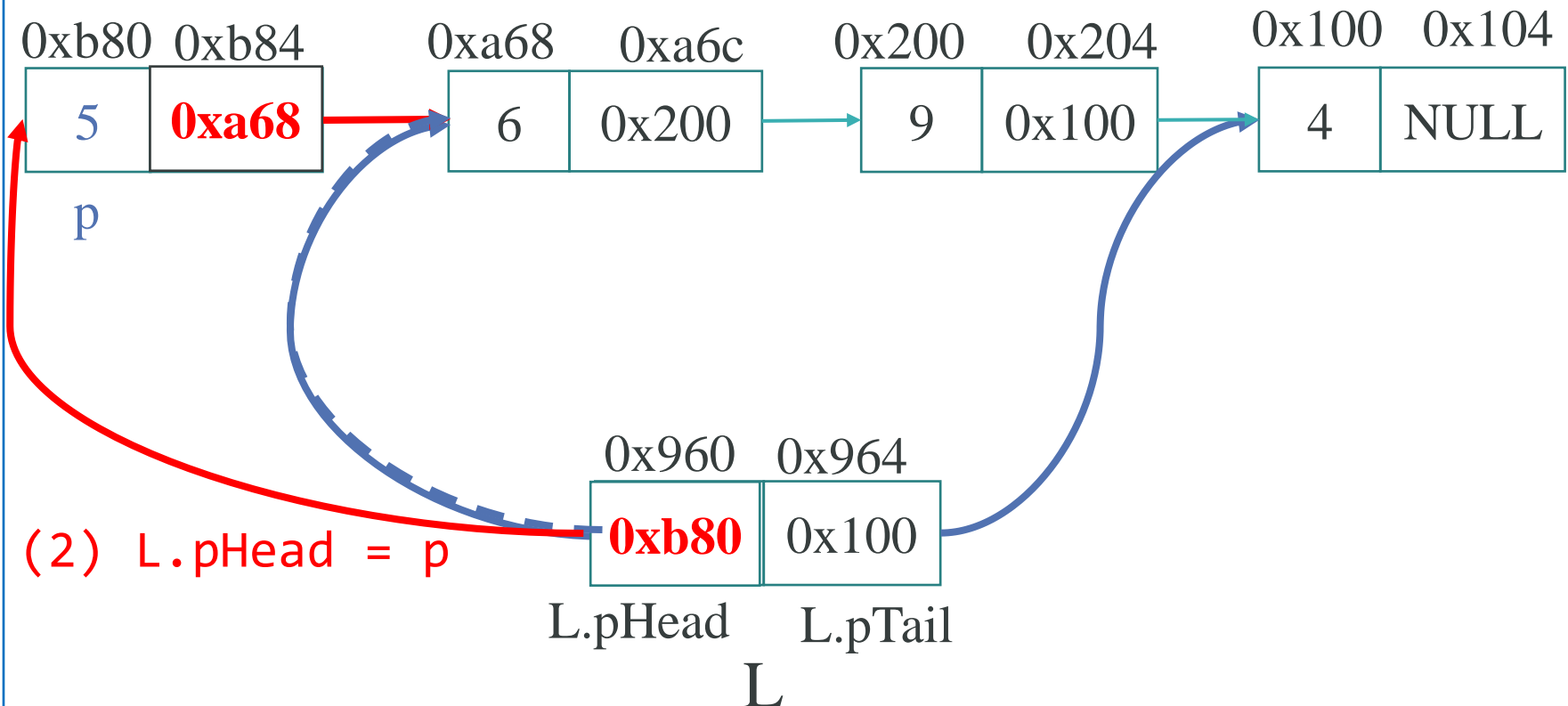


Thêm 1 phần tử vào đầu DSLK: Minh họa (tt)

- Thêm 5 vào đầu danh sách có thứ tự như sau: {6, 9, 4}

Memory Layout

(1) $p \rightarrow pNext = L.pHead$



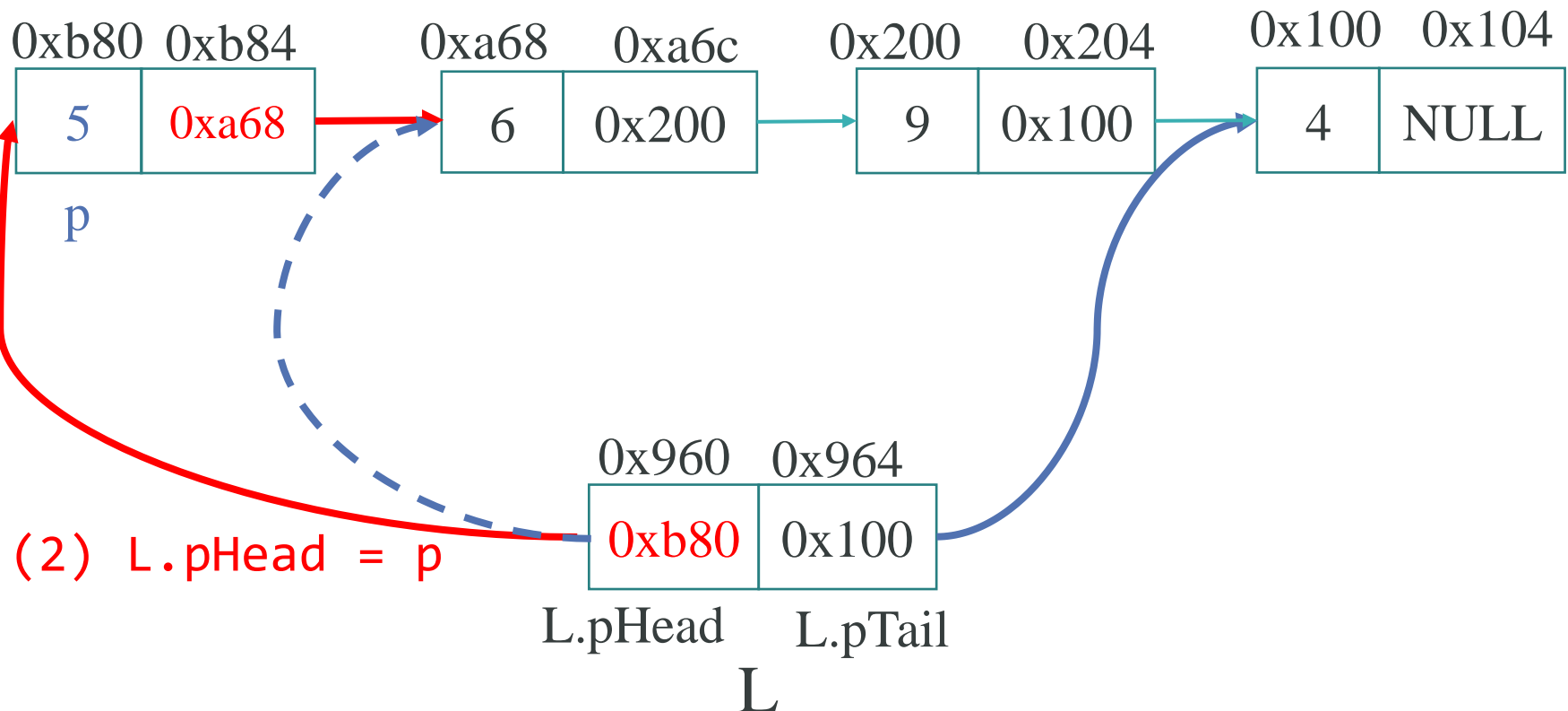


Thêm 1 phần tử vào đầu DSLK: Minh họa (tt)

- Thêm 5 vào đầu danh sách có thứ tự như sau: {6, 9, 4}

Memory Layout

(1) $p \rightarrow pNext = L.pHead$





➤ Ta cần thêm nút p vào cuối list đơn

Bắt đầu:

Nếu List rỗng thì

+ pHead = p;

+ pTail = pHead;

Ngược lại

+ pTail->pNext=p;

+ pTail=p



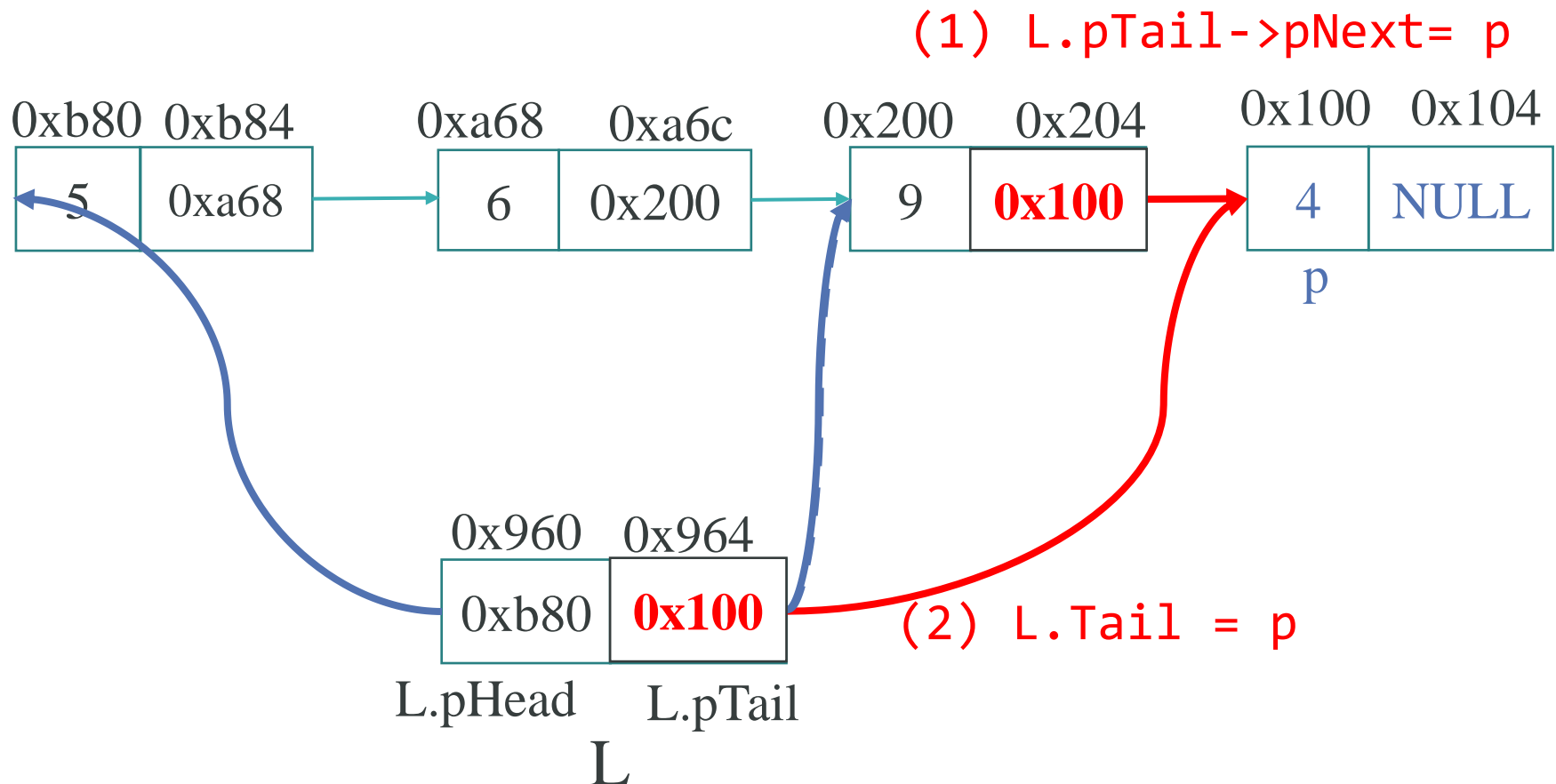
```
void AddTail(LIST &L, NODE* p) {  
    if (L.pHead == NULL) {  
        L.pHead = p;  
        L.pTail = L.pHead;  
    }  
    else {  
        L.pTail->pNext = p;  
        L.pTail = p;  
    }  
}
```



Thêm 1 phần tử vào cuối DSLK: Minh họa (tt)

- Thêm 4 vào cuối danh sách ban đầu có thứ tự như sau: {5, 6, 9}

Memory Layout

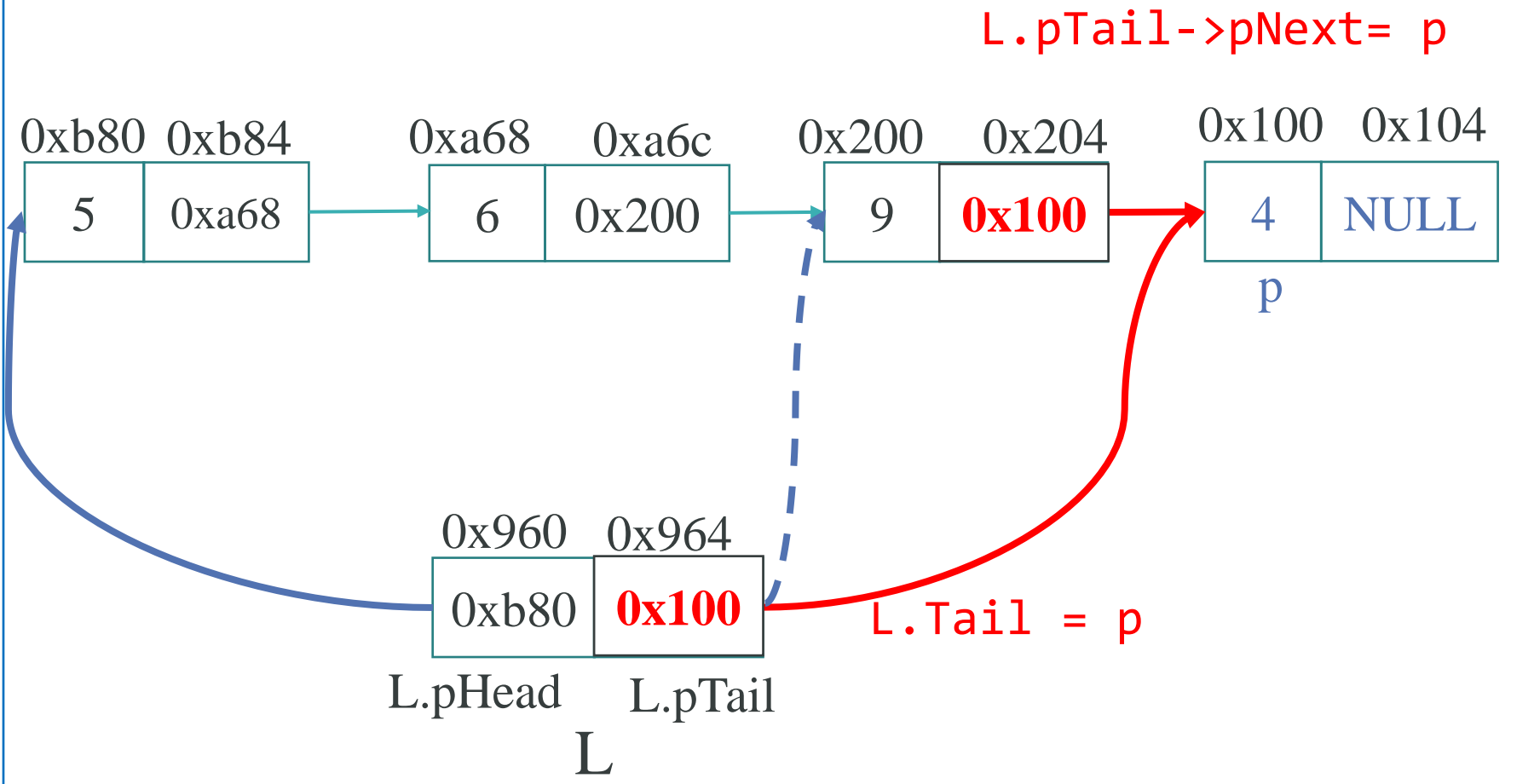




Thêm 1 phần tử vào cuối DSLK: Minh họa (tt)

- Thêm 4 vào cuối danh sách ban đầu có thứ tự như sau: {5, 6, 9}

Memory Layout



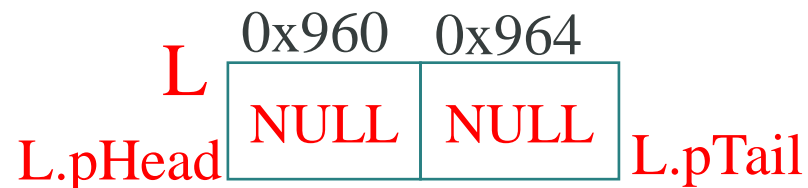


Ví dụ: Quá trình Thêm từng phần tử vào danh sách

○ Tạo danh sách RỖNG.

Memory Layout

```
int main() {  
    LIST L;  
    CreateEmptyList(L);  
}
```



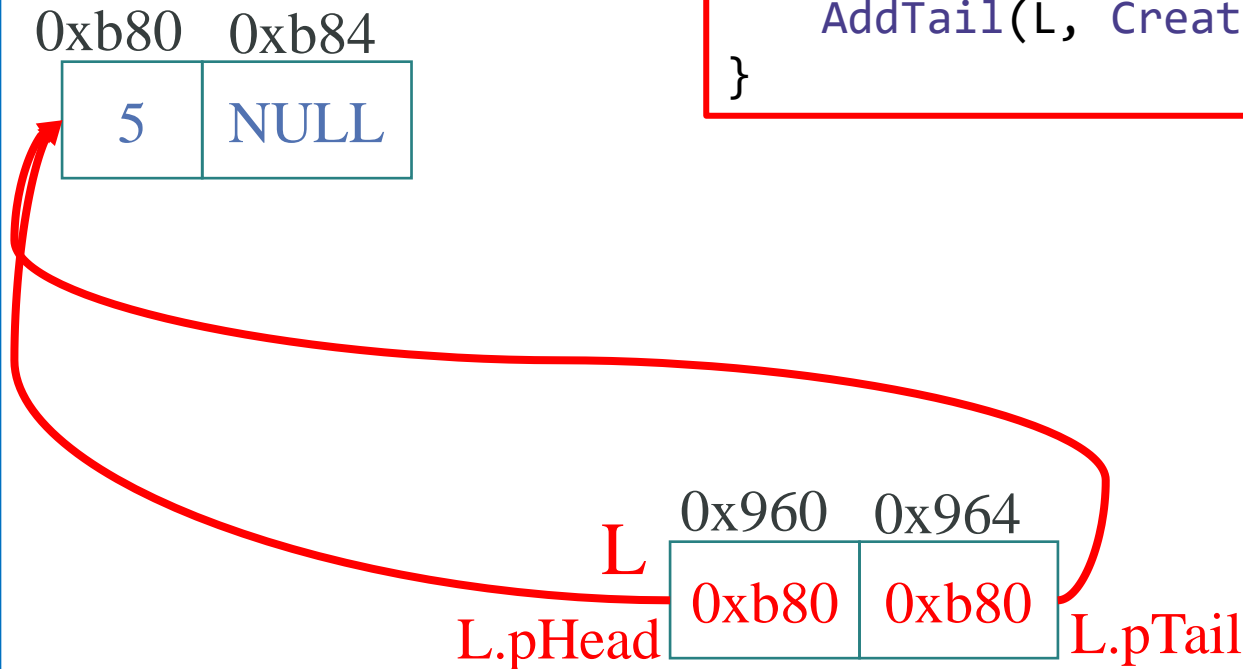


Ví dụ: Quá trình Thêm từng phần tử vào danh sách (tt)

- Thêm 5 vào danh sách.

Memory Layout

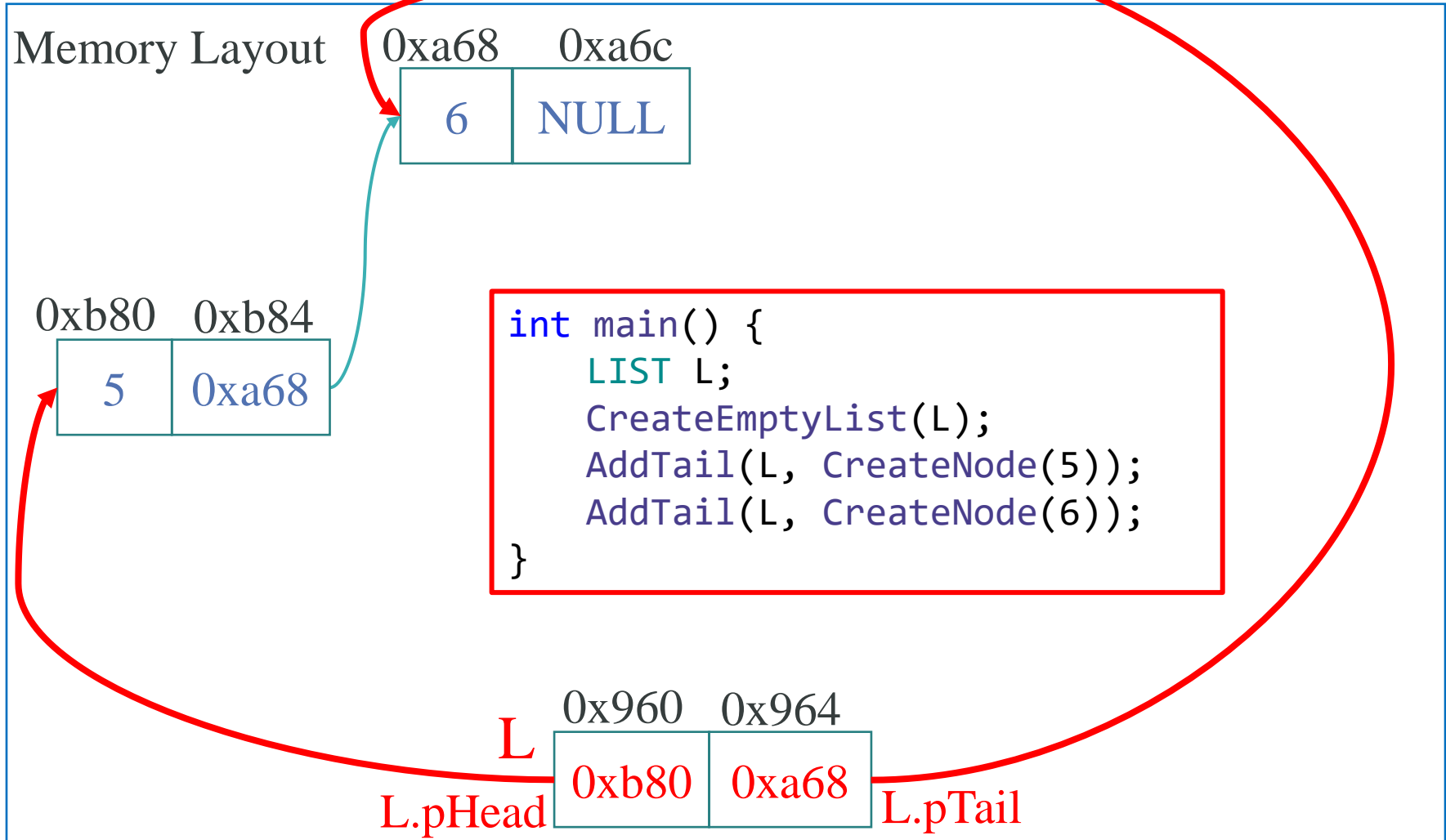
```
int main() {  
    LIST L;  
    CreateEmptyList(L);  
    AddTail(L, CreateNode(5));  
}
```





Ví dụ: Quá trình Thêm từng phần tử vào danh sách (tt)

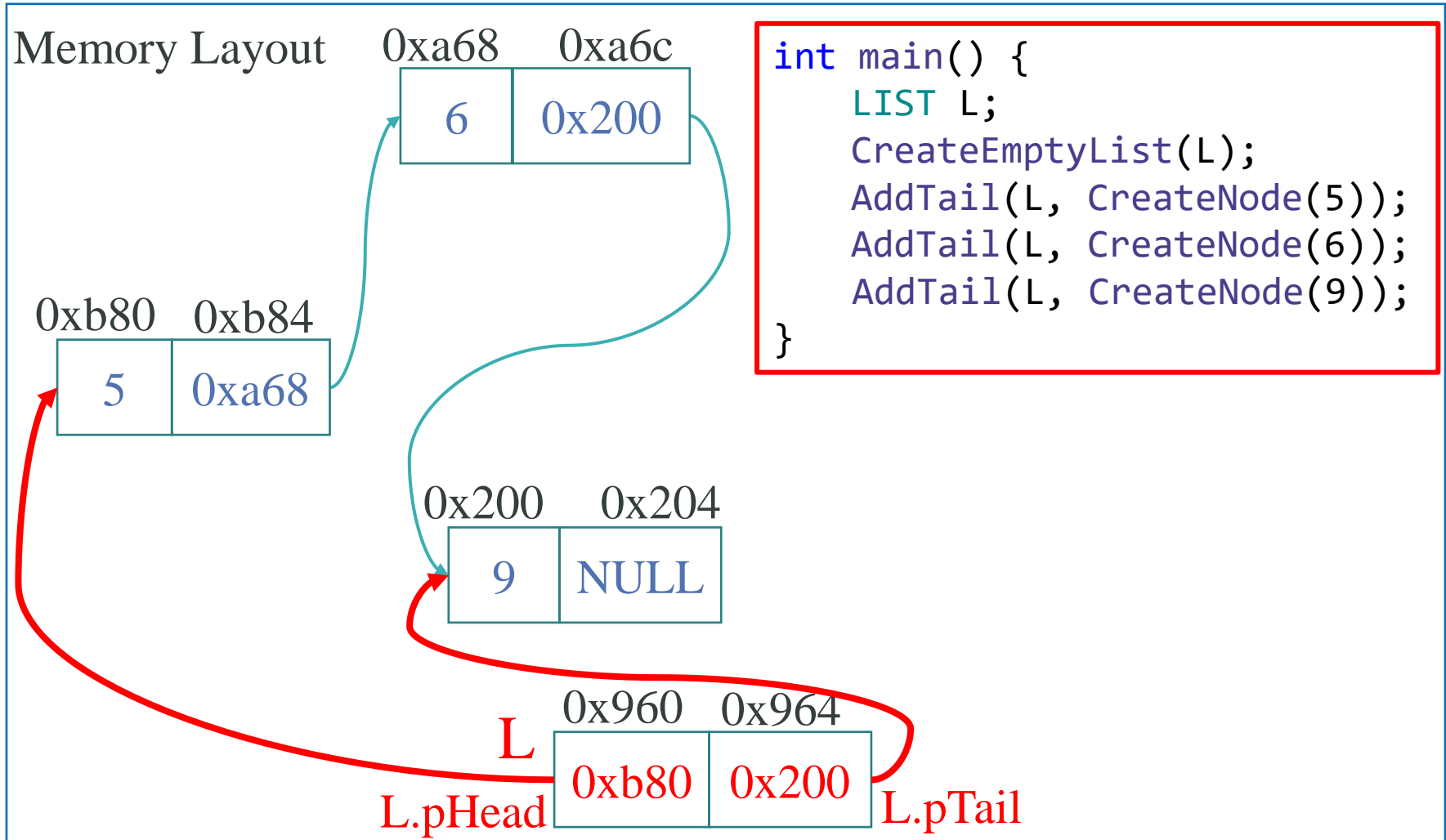
◦ Thêm 6 vào danh sách





Ví dụ: Quá trình Thêm từng phần tử vào danh sách (tt)

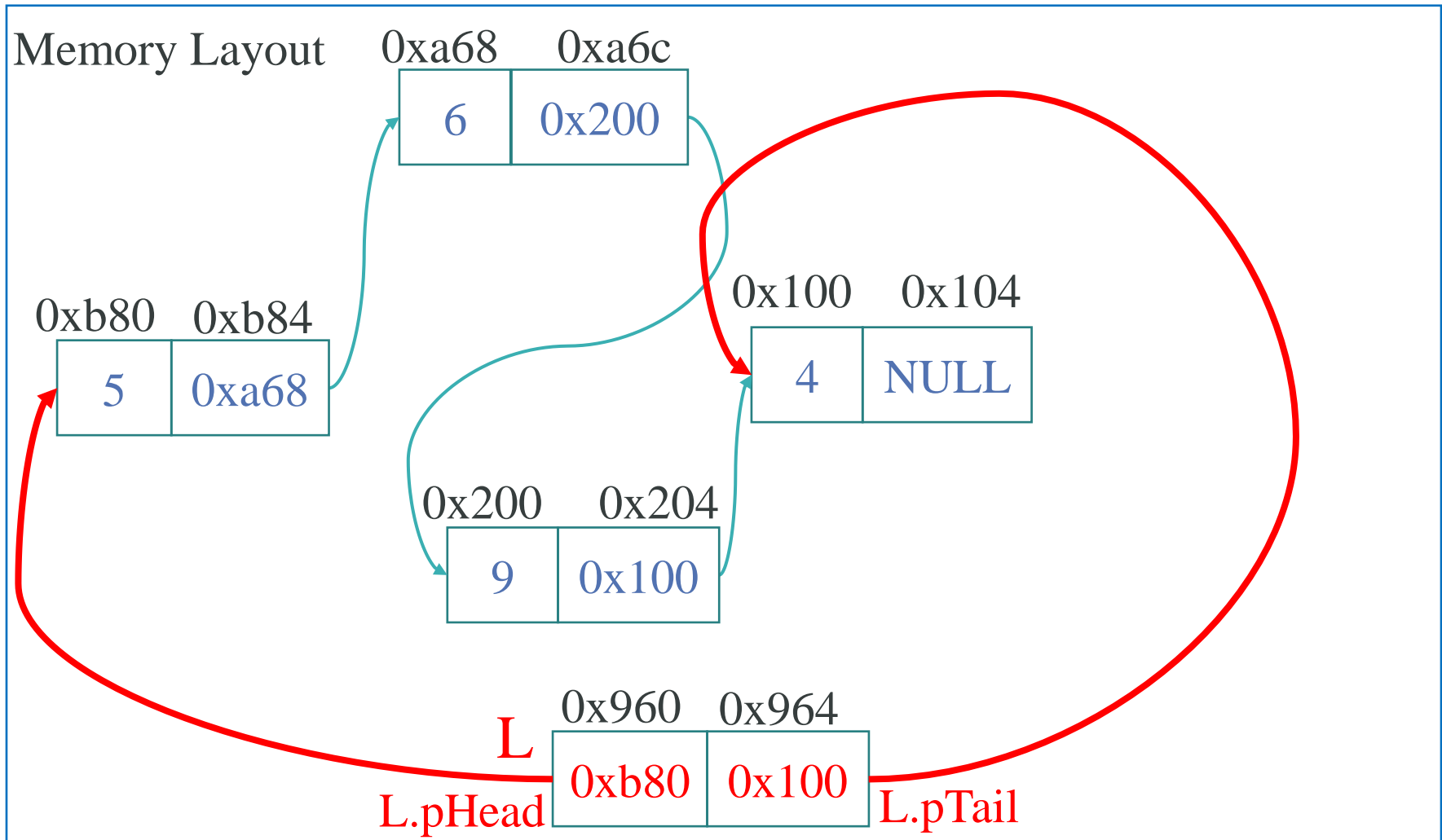
- Thêm 9 vào danh sách.





Ví dụ: Quá trình Thêm từng phần tử vào danh sách (tt)

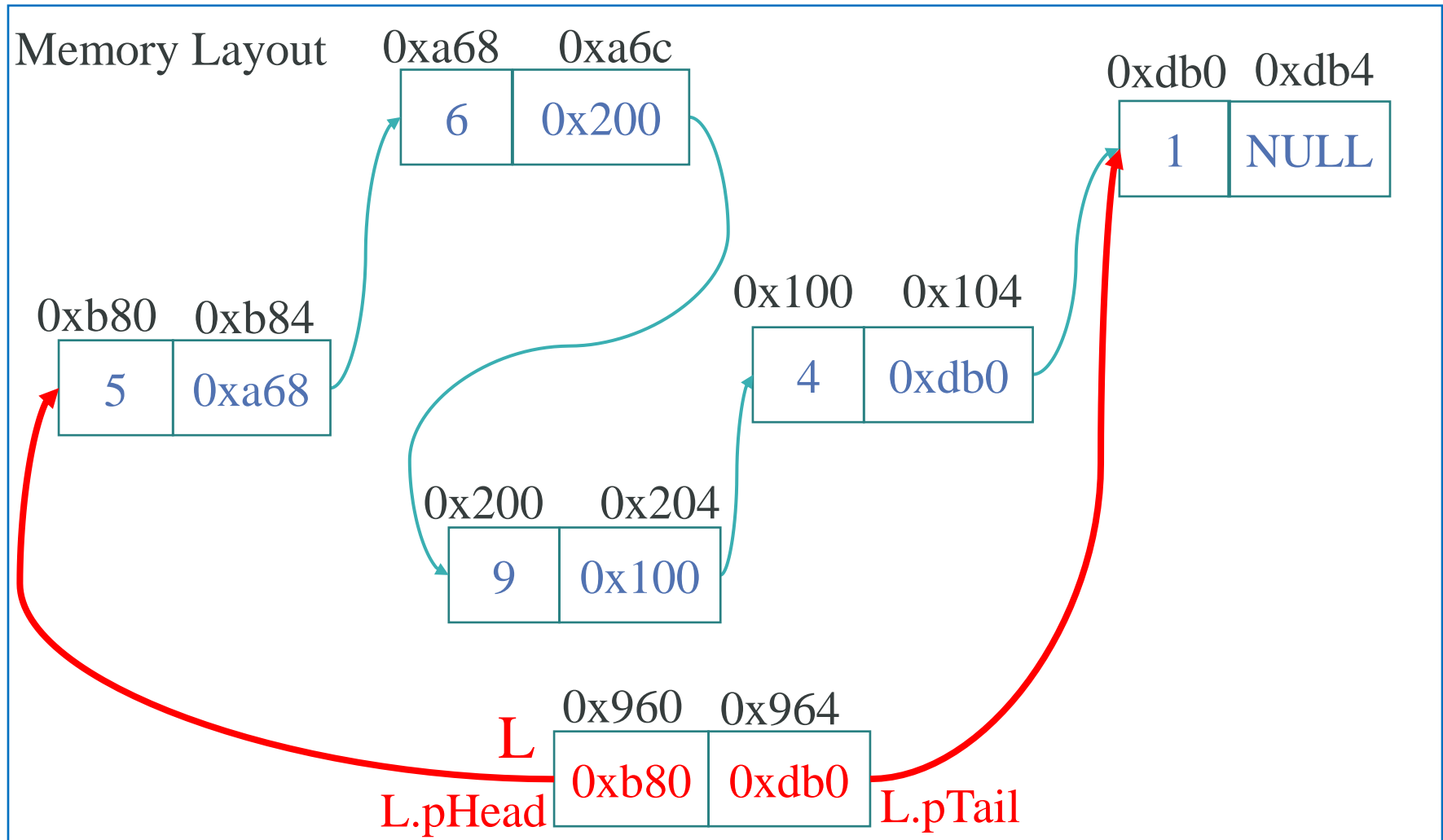
- Thêm 4 vào danh sách.





Ví dụ: Quá trình Thêm từng phần tử vào danh sách (tt)

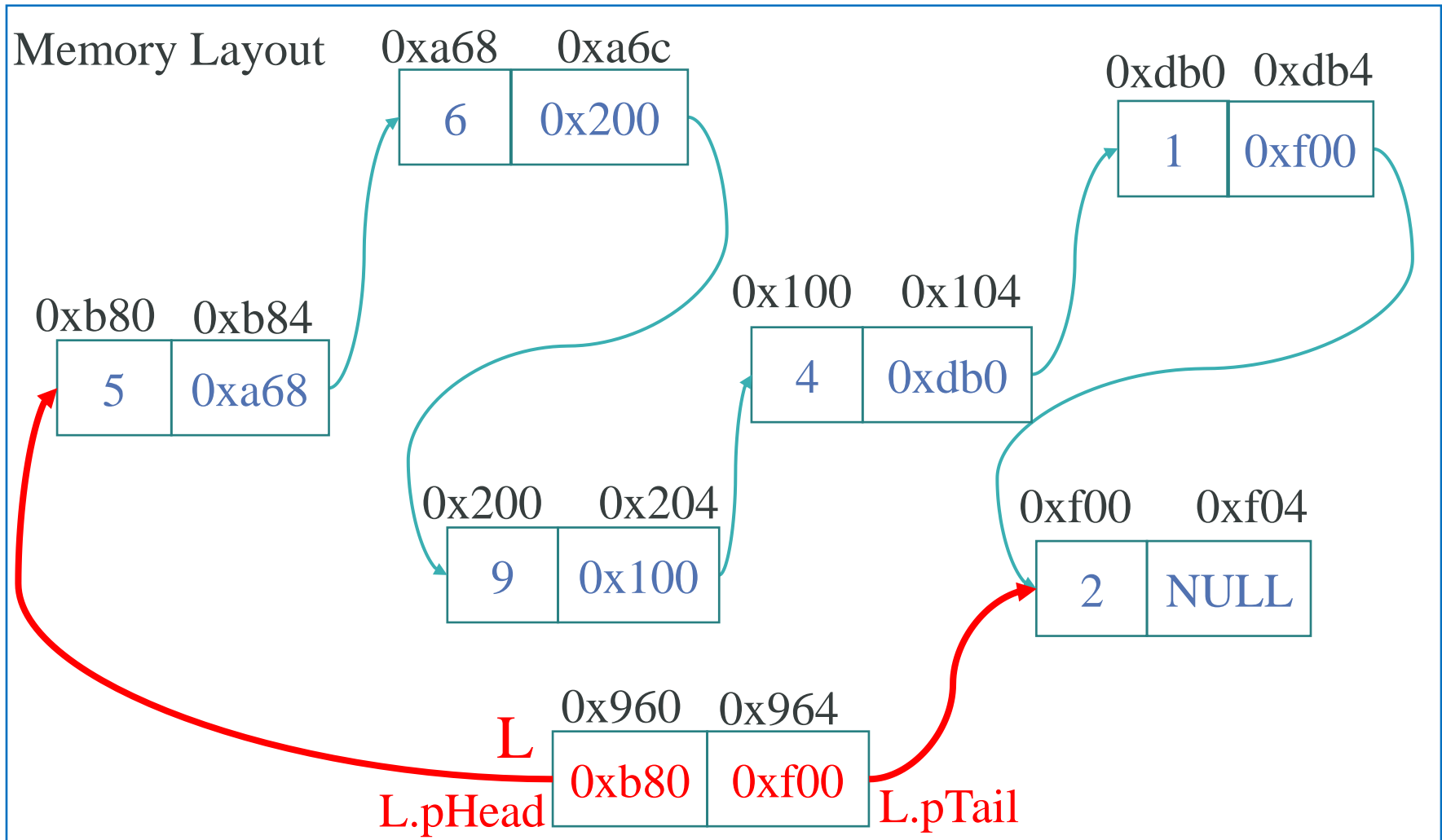
○Thêm 1 vào danh sách.





Ví dụ: Quá trình Thêm từng phần tử vào danh sách (tt)

○Thêm 2 vào danh sách



Thuật toán: Thêm phần tử p vào sau phần tử



- Ta cần thêm nút p vào sau nút Q trong list đơn

Bắt đầu:

Nếu ($Q \neq \text{NULL}$) thì

B1: $p \rightarrow \text{pNext} = Q \rightarrow \text{pNext}$

B2:

+ $Q \rightarrow \text{pNext} = p$

+ Nếu $Q = \text{pTail}$ thì

$\text{pTail} = p$

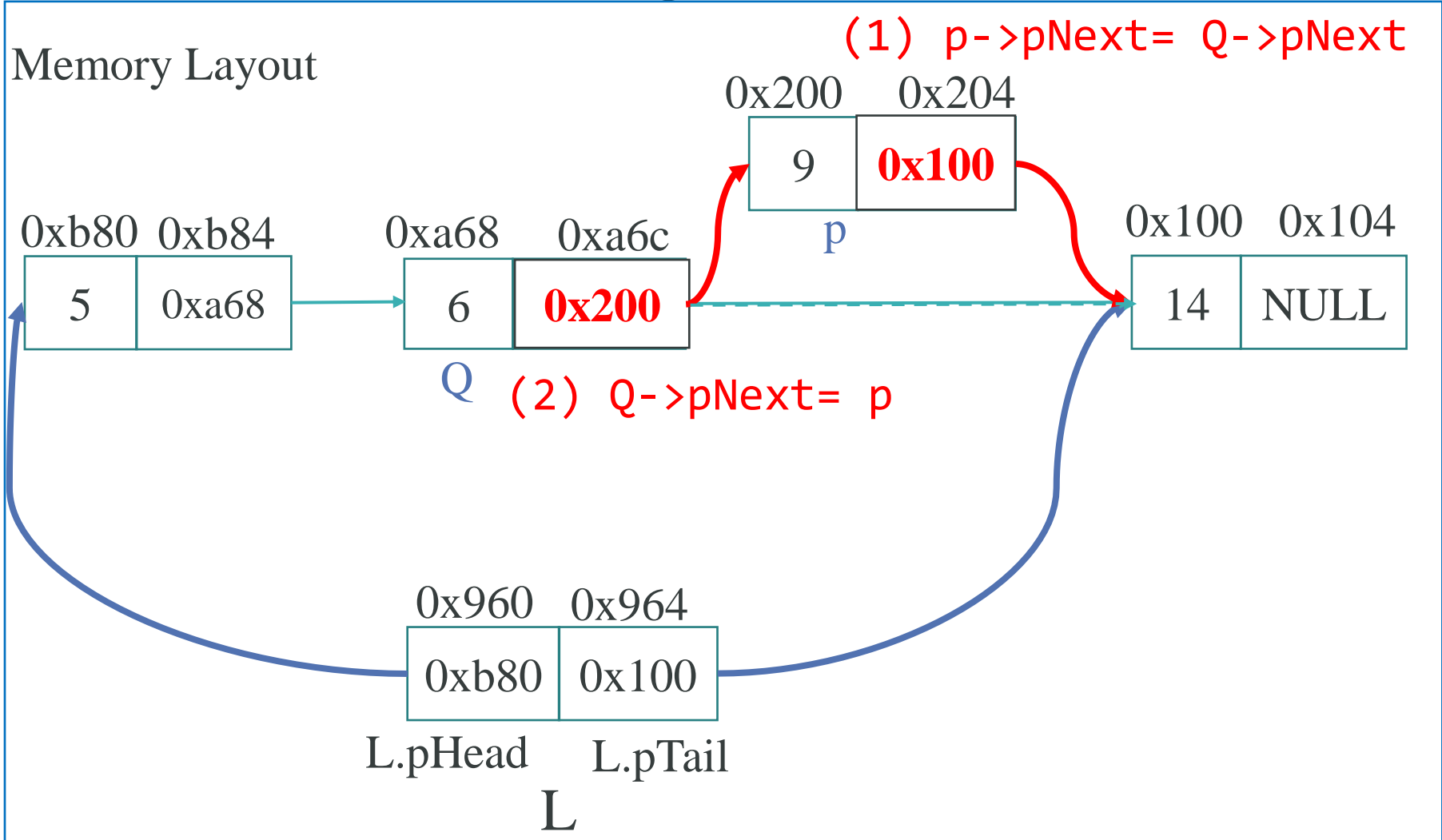


```
void AddAfterQ(LIST &L, NODE* p, NODE* Q) {  
    if (Q != NULL) {  
        p->pNext = Q->pNext;  
        Q->pNext = p;  
        if (L.pTail == Q)  
            L.pTail = p;  
    }  
    else  
        AddHead(L, p); // thêm q vào đầu list  
}
```



- Thêm 9 vào sau node Q có giá trị 6: {5, 6, 14}

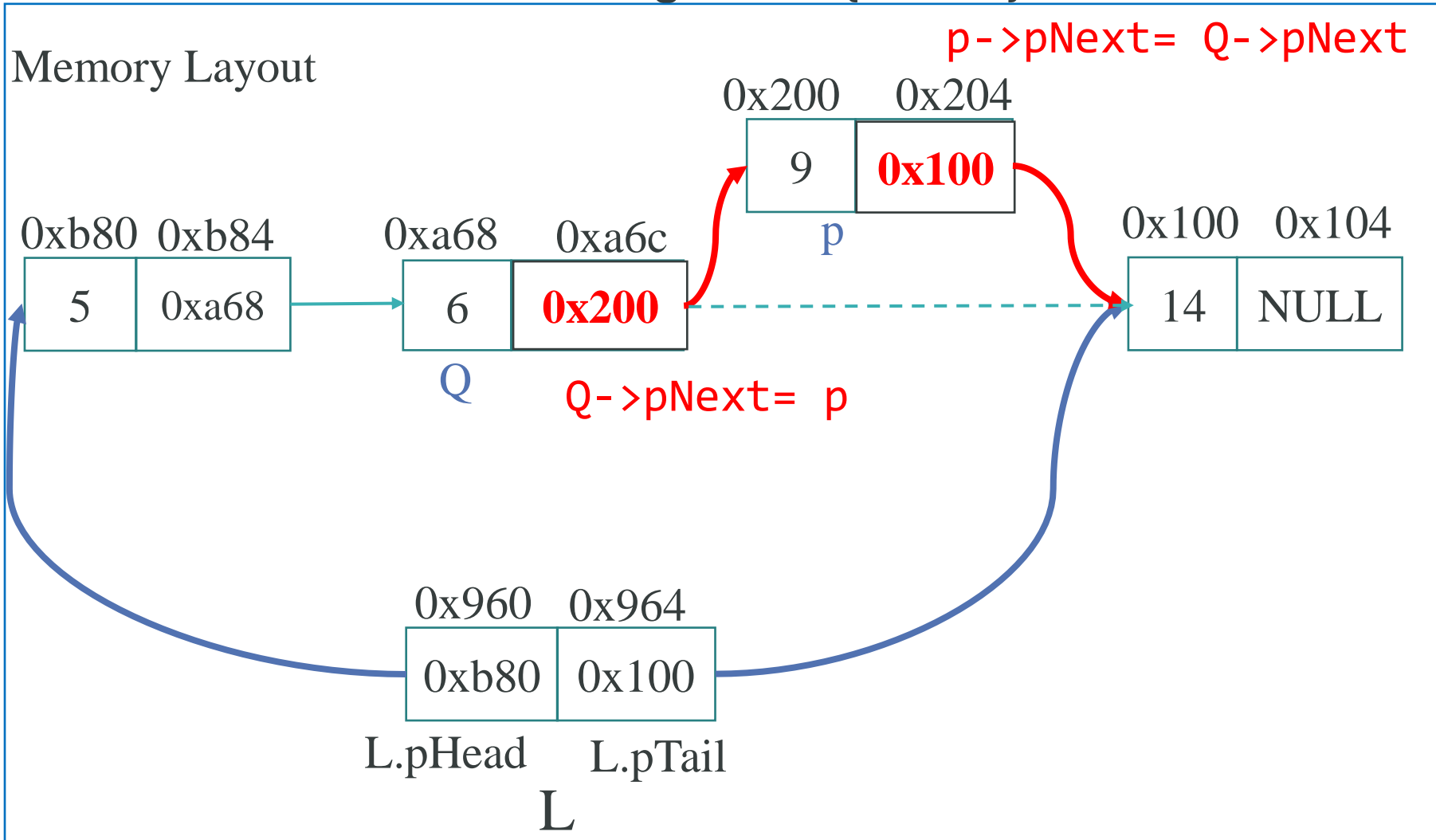
Memory Layout





Minh họa (tt)

- Thêm 9 vào sau node Q có giá trị 6: {5, 6, 14}



Hủy phần tử trong DSLK đơn



- **Nguyên tắc:** Phải cô lập phần tử cần hủy trước hủy.
- Các vị trí cần hủy
 - Hủy phần tử đứng đầu List
 - Hủy phần tử có khoá bằng x
 - Hủy phần tử đứng sau q trong danh sách liên kết đơn
- Ở phần trên, các phần tử trong DSLK đơn được cấp phát vùng nhớ động bằng hàm new, thì sẽ được giải phóng vùng nhớ bằng hàm delete.



➤ Bắt đầu:

Nếu ($pHead \neq NULL$) thì

- B1: $p = pHead$

- B2:

 - + $pHead = pHead \rightarrow pNext$

 - + delete (p)

- B3:

Nếu $pHead == NULL$ thì $pTail = NULL$



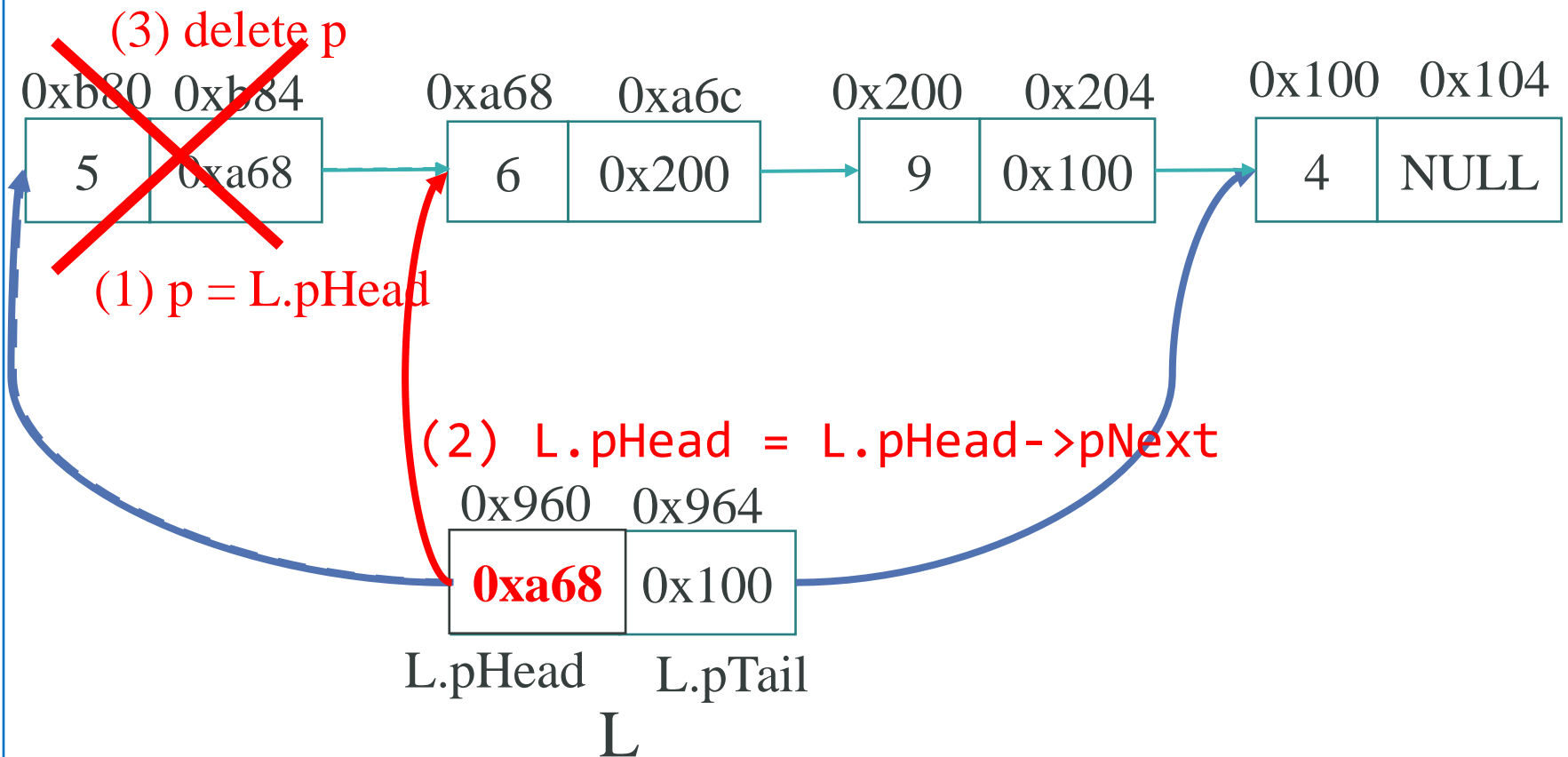
```
bool RemoveHead(LIST &L, int &x) {  
    NODE *p;  
    if (L.pHead != NULL) {  
        x = L.pHead->info;  
        p = L.pHead; // Bước 1  
        L.pHead = L.pHead->pNext; // Bước 2  
        if (L.pHead == NULL)  
            L.pTail = NULL;  
        delete p; // Bước 3  
        return 1; // Đã xóa phần tử  
    }  
    return 0; // List rỗng không có phần tử để xóa  
}
```



Hủy phần tử đầu trong DSLK: Minh họa

- Hủy phần tử đầu danh sách liên kết có thứ tự như sau: {5, 6, 9, 4}

Memory Layout

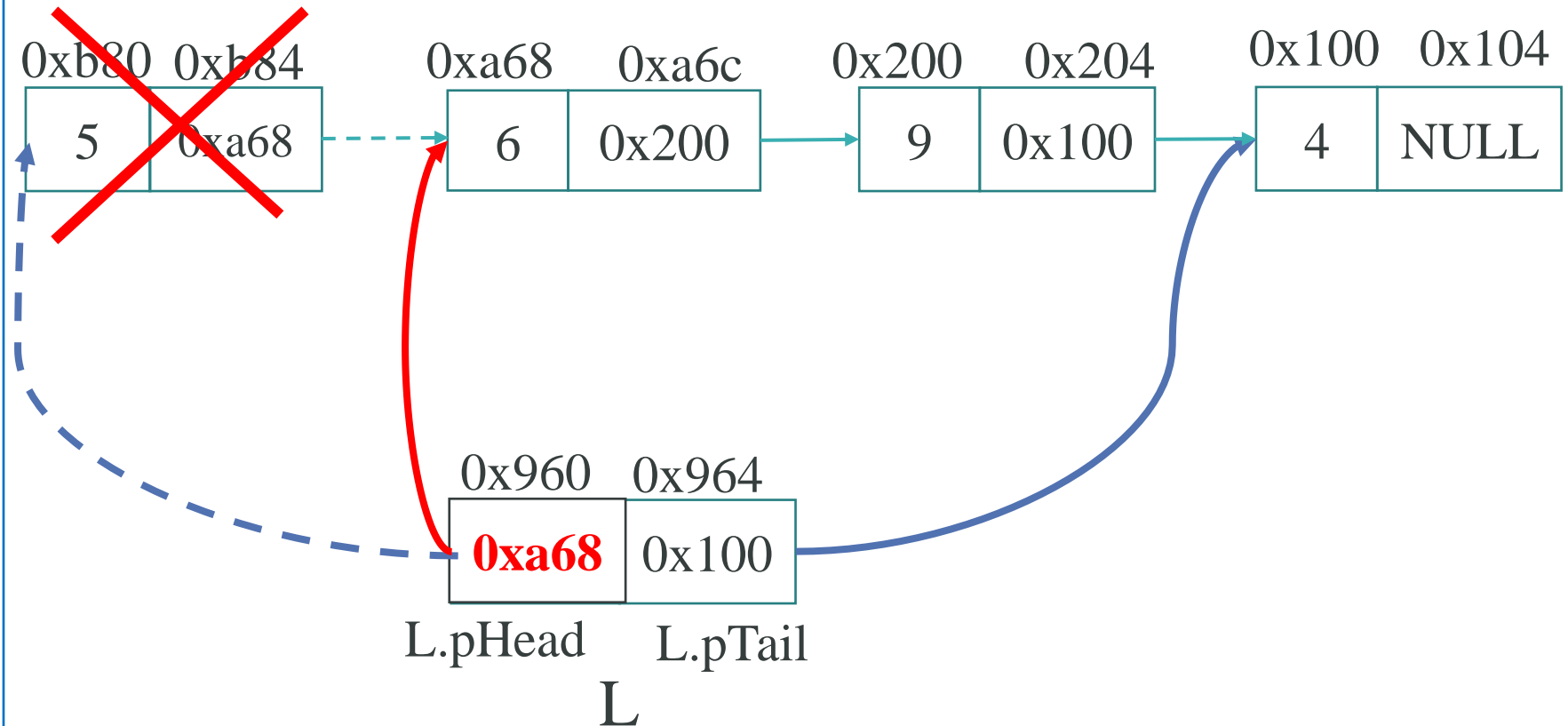




Hủy phần tử đầu trong DSLK: Minh họa (tt)

- Hủy phần tử đầu danh sách liên kết có thứ tự như sau: {5, 6, 9, 4}

Memory Layout



Thuật toán hủy phần tử đứng sau phần tử Q



➤ Bắt đầu

Nếu ($Q \neq \text{NULL}$) thì: // Q tồn tại trong List

- B1: $p = Q \rightarrow \text{pNext}$; // p là phần tử cần hủy
- B2: Nếu ($p \neq \text{NULL}$) thì // Q không là phần tử cuối
 - + $Q \rightarrow \text{pNext} = p \rightarrow \text{pNext}$; // tách p ra khỏi xâu
 - + Nếu ($p == \text{pTail}$) // nút cần hủy là nút cuối
 $\text{pTail} = Q$;
 - + delete p; // hủy p



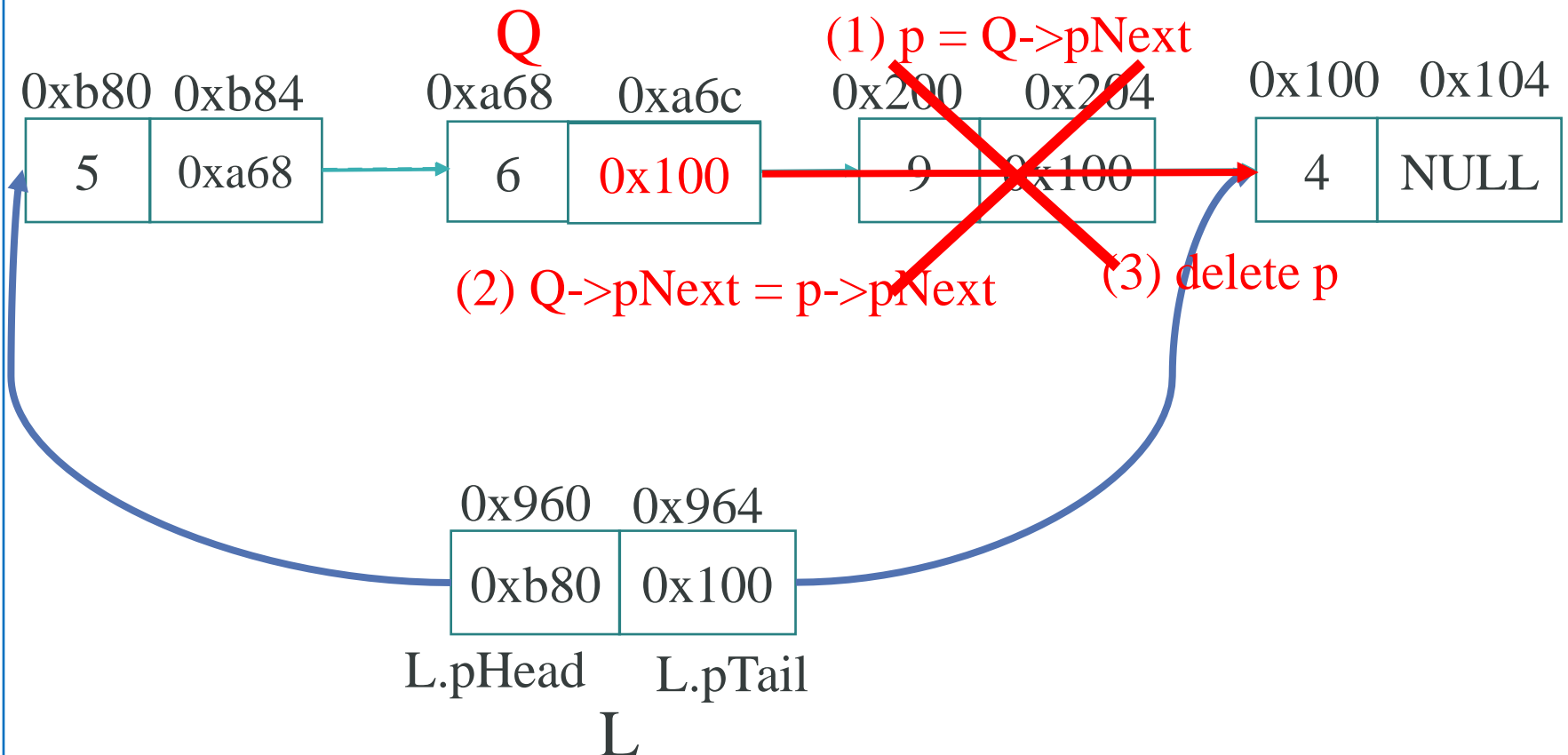
```
bool RemoveAfterQ(LIST &l, NODE *Q, int &x) {  
    NODE *p;  
    if (Q != NULL) {  
        p = Q->pNext;    // p là nút cần xóa  
        if (p != NULL) {    // q không phải là nút cuối  
            if (p == l.pTail) // nút cần xóa là nút cuối cùng  
                l.pTail = Q;    // cập nhật lại pTail  
            Q->pNext = p->pNext;  
            x = p->info;  
            delete p;  
        }  
        return 1;  
    }  
    return 0;  
}
```

Thuật toán hủy phần tử đứng sau phần tử Q: Minh họa



- Hủy phần tử đứng sau node có giá trị 6 trong danh sách: {5, 6, 9, 4}

Memory Layout

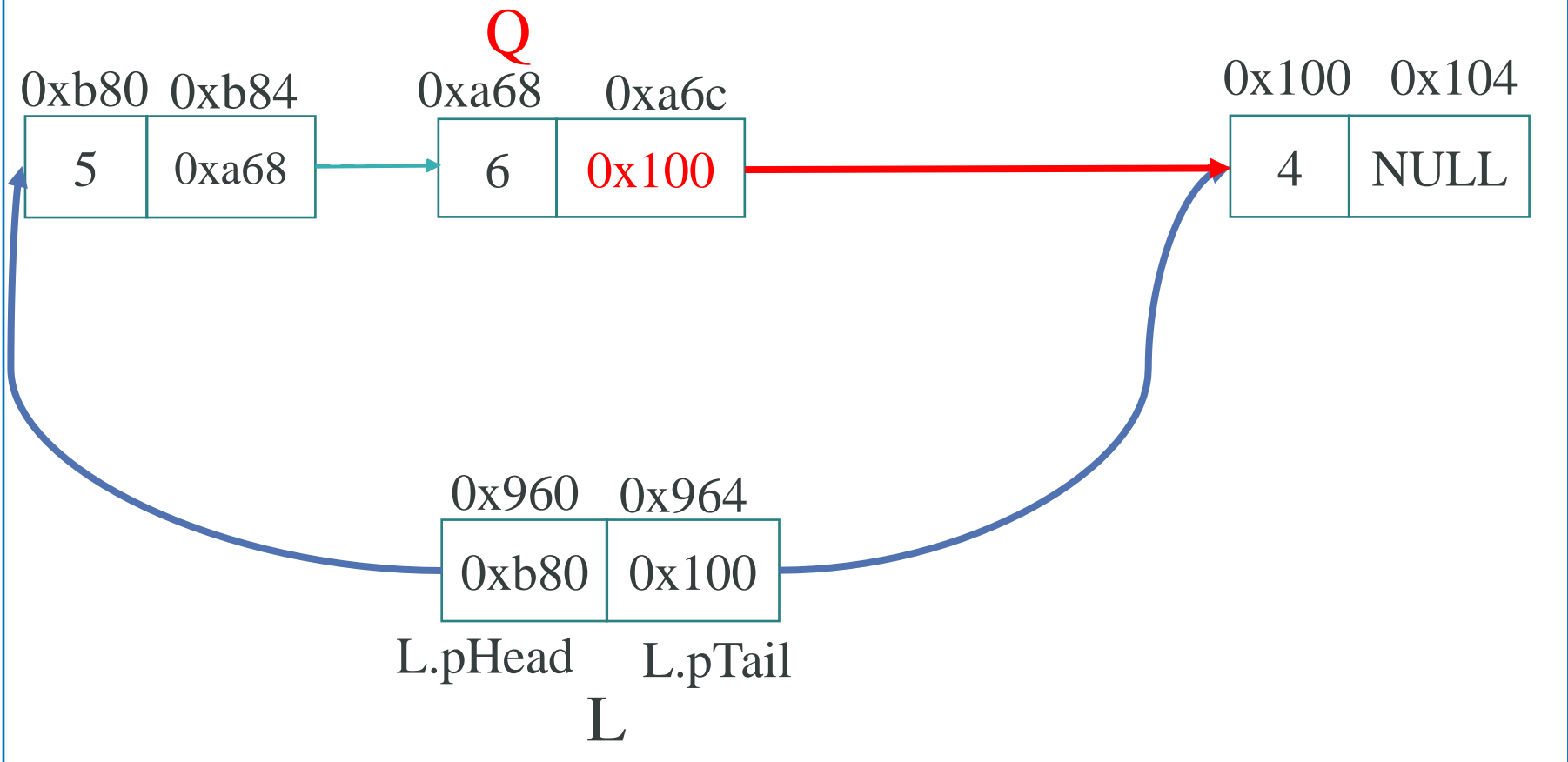


Thuật toán hủy phần tử đứng sau phần tử Q: Minh họa



- Hủy phần tử đứng sau node có giá trị 6 trong danh sách: {5, 6, 9, 4}

Memory Layout



Thuật toán hủy phần tử có khoá x



Bước 1:

Tìm phần tử p có khoá bằng x, và Q đứng trước p

Bước 2:

Nếu (p!=NULL) thì //tìm thấy phần tử có khoá bằng x

Hủy p ra khỏi List bằng cách hủy phần tử đứng sau Q

Ngược lại: Báo không tìm thấy phần tử có khoá x



```
bool RemoveX(LIST &L, int x) {  
    NODE *Q, *p;  
    Q = NULL;  
    p = L.pHead;  
    while (p!=NULL && p->info!=x) {  
        Q = p;  
        p = p->pNext;  
    }  
  
    if (p == NULL) return 0; // Tìm không thấy x  
    if (Q != NULL) RemoveAfterQ(L, Q, x);  
    else RemoveHead(L, x);  
    return 1;  
}
```

Tìm 1 phần tử trong DSLK đơn



- Tìm tuần tự (hàm trả về), các bước của thuật toán tìm nút có info bằng x trong list đơn.

Bước 1: `p=pHead`; // địa chỉ của phần tử đầu trong list đơn

Bước 2:

Trong khi `p!=NULL` và `p->info!=x` thì

`p=p->pNext`; // xét phần tử kế

Bước 3:

- + Nếu `p!=NULL` thì p lưu địa chỉ của nút có `info = x`
- + Ngược lại: Không có phần tử cần tìm



// Hàm trả về NULL: Không tìm thấy

// Hàm trả về khác NULL: Tìm thấy

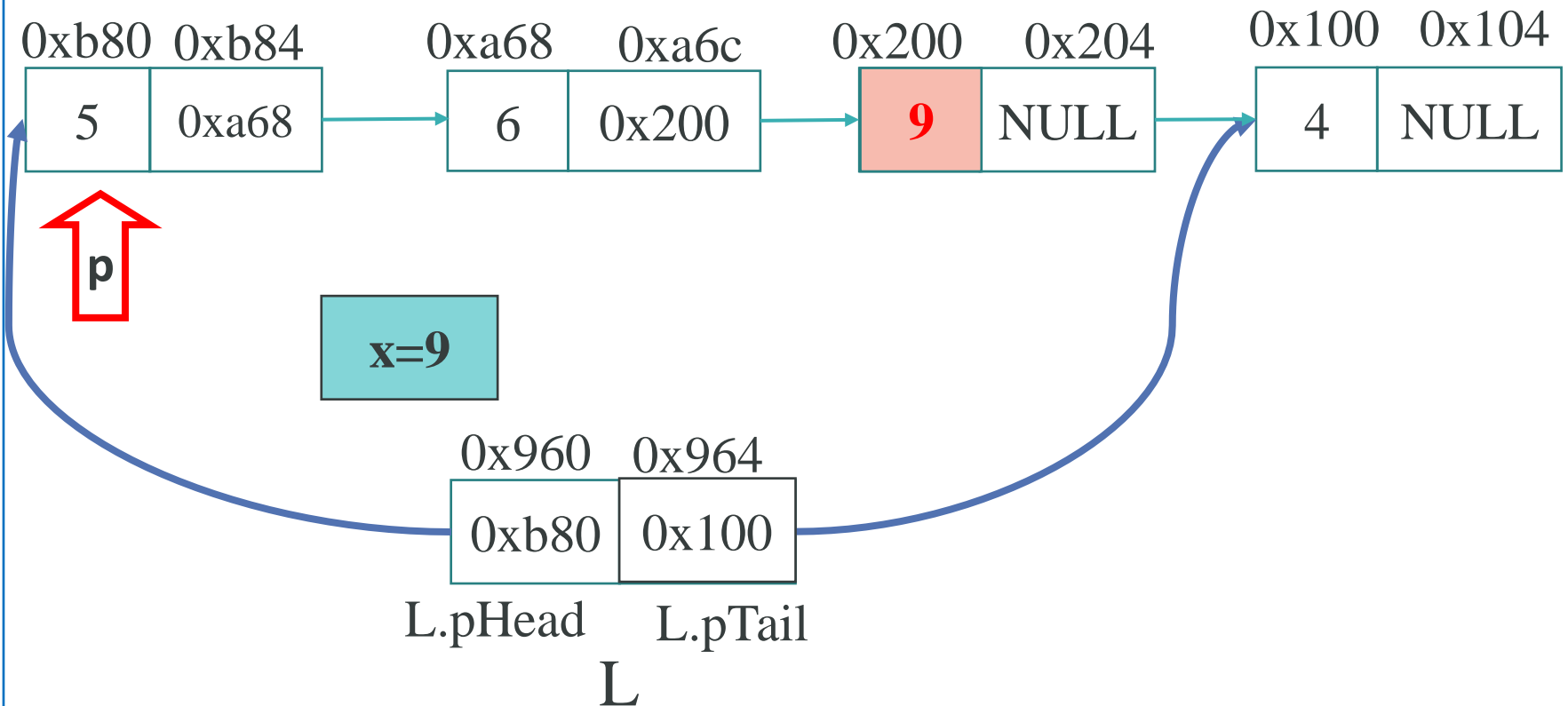
```
NODE* SearchNode(LIST L, int x) {  
    NODE* p = L.pHead;  
    while (p != NULL && p->info != x)  
        p = p->pNext;  
    return p;  
}
```



Tìm 1 phần tử trong DSLK đơn: Minh họa

- Tìm phần tử có giá trị bằng 9 trong DSLK gồm {5, 6, 9, 4}

Memory Layout





- Duyệt danh sách là thao tác thường được thực hiện khi có nhu cầu cần xử lý các phần tử trong danh sách như:
 - Đếm các phần tử trong danh sách
 - Tìm tất cả các phần tử trong danh sách thỏa điều kiện
 - Hủy toàn bộ danh sách



- Bước 1:

`p = pHead;` // p lưu địa chỉ của phần tử đầu trong List

- Bước 2:

Trong khi (danh sách chưa hết) thực hiện

- + Xử lý phần tử p

- + `p=p->pNext;` // qua phần tử kế

Cài đặt in các phần tử trong List



```
void PrintList(LIST L) {  
    NODE* p;  
    if (L.pHead == NULL)  
        cout << "\nDSLK rong."  
    else {  
        p = L.pHead;  
        while (p) {  
            cout << p->info << "\t";  
            p = p->pNext;  
        }  
    }  
}
```



▪ Bước 1:

Trong khi việc duyệt danh sách chưa kết thúc, thực hiện:

• BI.1:

$p = pHead;$

$pHead = pHead \rightarrow pNext;$ // cập nhật pHead

• BI.2:

Hủy p

▪ Bước 2:

$pTail = NULL;$ // bảo toàn tính nhất quán khi xâu rỗng



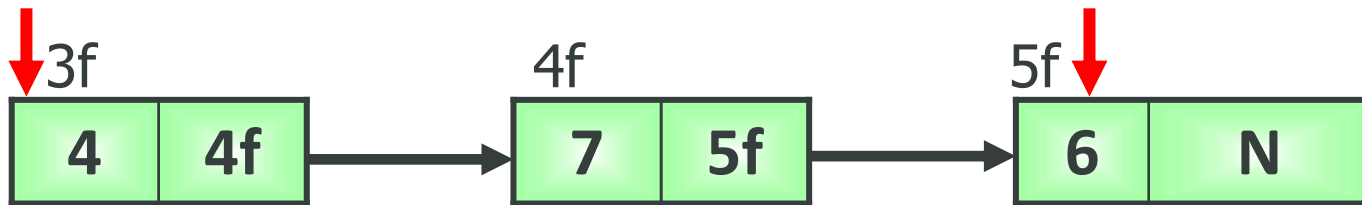
```
void RemoveList(LIST &L) {  
    NODE * p;  
    while (L.pHead != NULL) {  
        p = L.pHead;  
        L.pHead = p->pNext;  
        delete p;  
    }  
    L.pTail = NULL;  
}
```

Sắp xếp danh sách

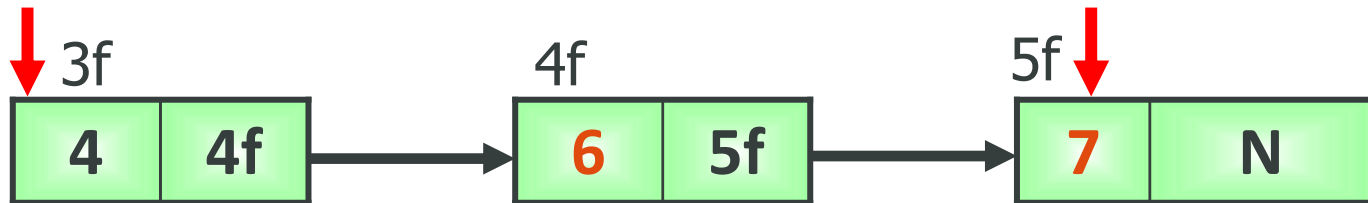


- Có hai cách tiếp cận
- **Cách I:** Thay đổi thành phần Info

pHead



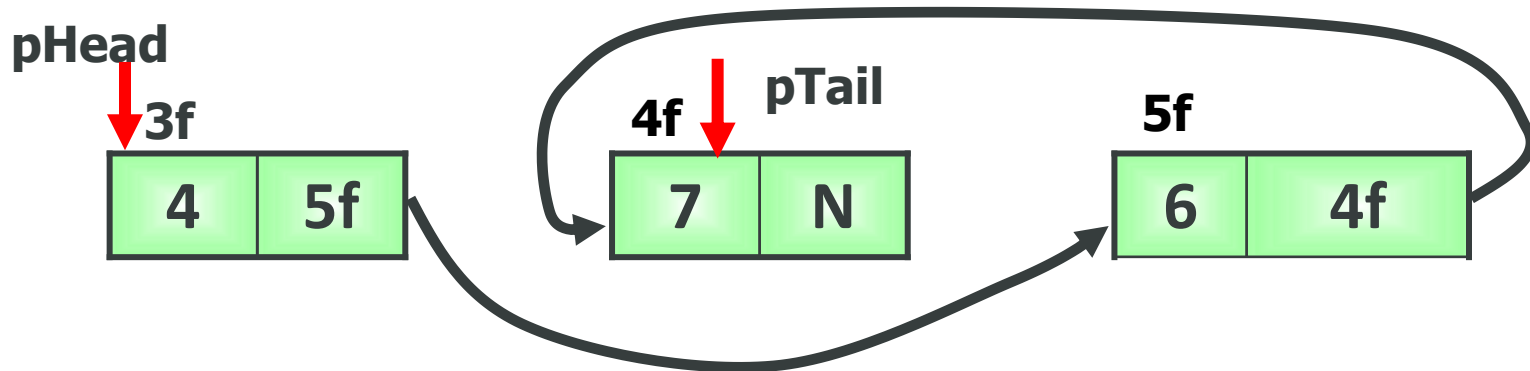
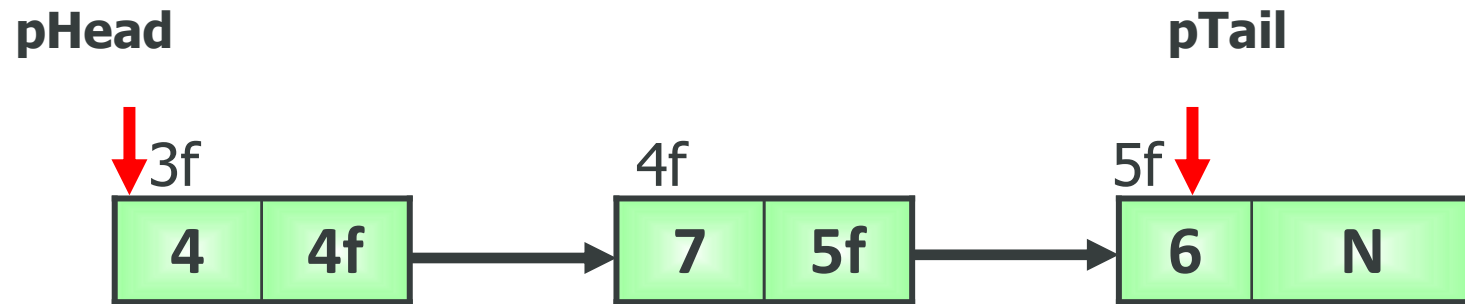
pHead



Sắp xếp danh sách



- **Cách 2:** Thay đổi thành phần pNext (thay đổi trình tự móc nối của các phần tử sao cho tạo lập nên được thứ tự mong muốn)



Ưu, nhược điểm của 2 cách tiếp cận



- Thay đổi thành phần info (dữ liệu)
 - Ưu: Cài đặt đơn giản, tương tự như sắp xếp mảng
 - Nhược:
 - Đòi hỏi thêm vùng nhớ khi hoán vị nội dung của 2 phần tử -> chỉ phù hợp với những cấu trúc có kích thước info nhỏ
 - Khi kích thước info (dữ liệu) lớn chi phí cho việc hoán vị thành phần info lớn
 - ✓ Làm cho thao tác sắp xếp chậm
- Thay đổi thành phần pNext
 - Ưu:
 - Kích thước của trường này không thay đổi, do đó không phụ thuộc vào kích thước bản chất dữ liệu lưu tại mỗi nút.
 - ✓ Thao tác sắp xếp nhanh
 - Nhược: Cài đặt phức tạp

Selection Sort: Code C/C++



```
void SelectionSort(LIST &L) {  
    NODE *p, *Q, *min;  
    p = L.pHead;  
    while (p != L.pTail) {  
        min = p;  
        Q = p->pNext;  
  
        while (Q != NULL) {  
            if (Q->info < min->info)  
                min = Q;  
            Q = Q->pNext;  
        }  
  
        if (min != p) Swap(min->info, p->info);  
  
        p = p->pNext;  
    }  
}
```



- Các thuật toán sắp xếp xâu (List) bằng các thay đổi thành phần pNext (thành phần liên kết) có hiệu quả cao như:
 - Thuật toán sắp xếp Quick Sort
 - Thuật toán sắp xếp Merge Sort
 - Thuật toán sắp xếp Radix Sort



- Bước 1:

Chọn X là phần tử đầu xâu L làm phần tử cầm canh

Loại X ra khỏi L

- Bước 2:

Tách xâu L ra làm 2 xâu L_1 (gồm các phần tử nhỏ hơn hoặc bằng x) và L_2 (gồm các phần tử lớn hơn X)

- Bước 3: Nếu ($L_1 \neq \text{NULL}$) thì QuickSort(L_1)

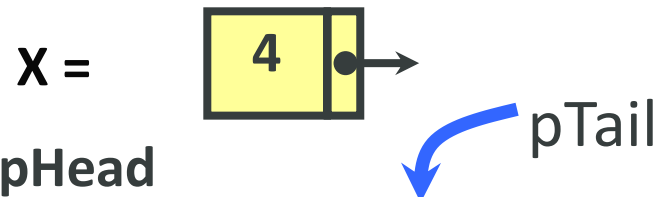
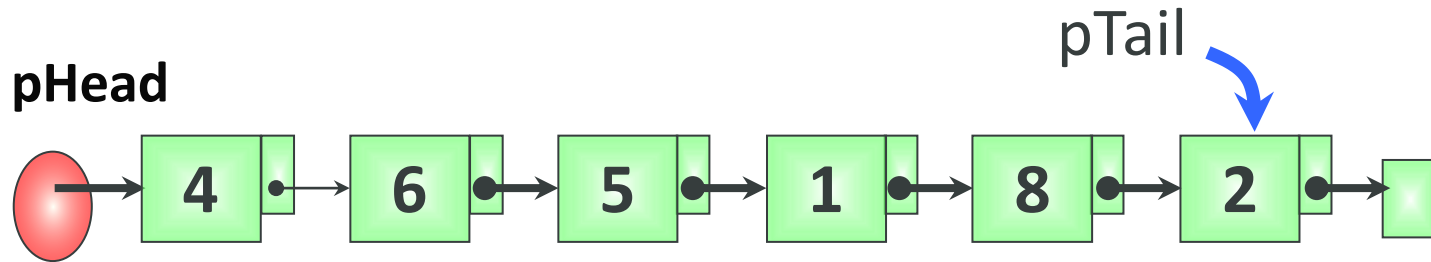
- Bước 4: Nếu ($L_2 \neq \text{NULL}$) thì QuickSort(L_2)

- Bước 5: Nối L_1, X, L_2 lại theo thứ tự ta có xâu L đã được sắp xếp

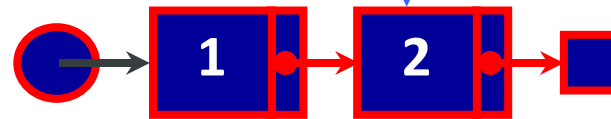
Quick Sort: Minh họa



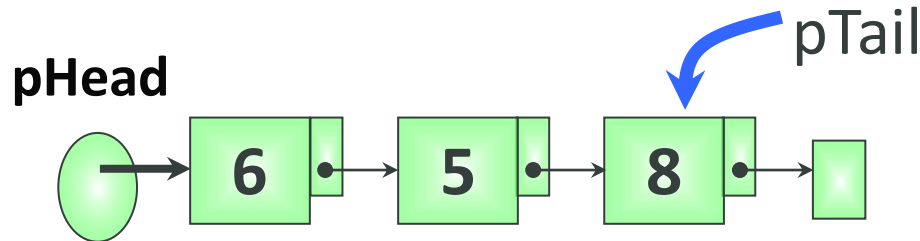
➤ Cho danh sách liên kết gồm các phần tử sau:



$L_1 (\leq X)$



$L_2 (> X)$



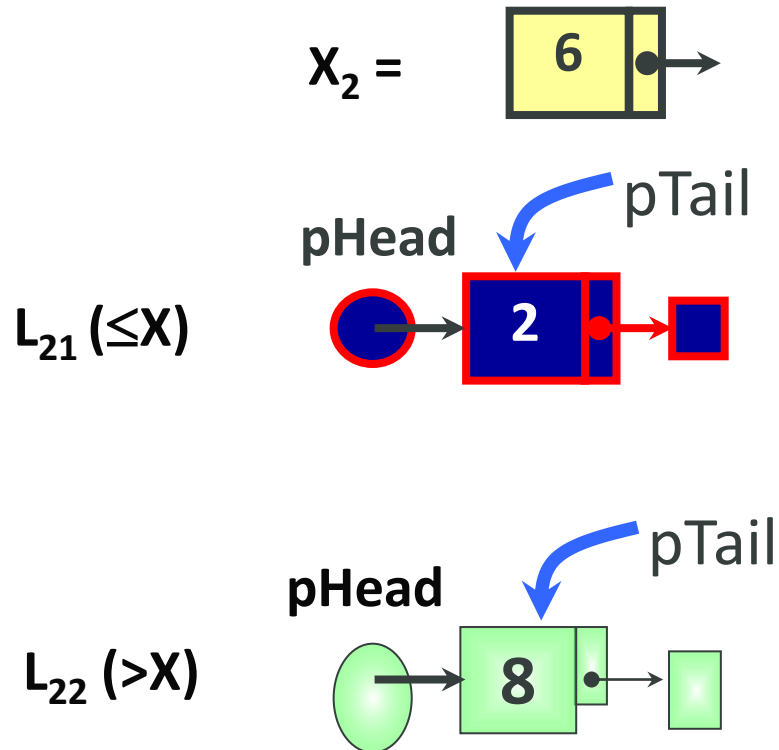
Quick Sort: Minh họa



➤ Sắp xếp L_1

➤ Sắp xếp L_2

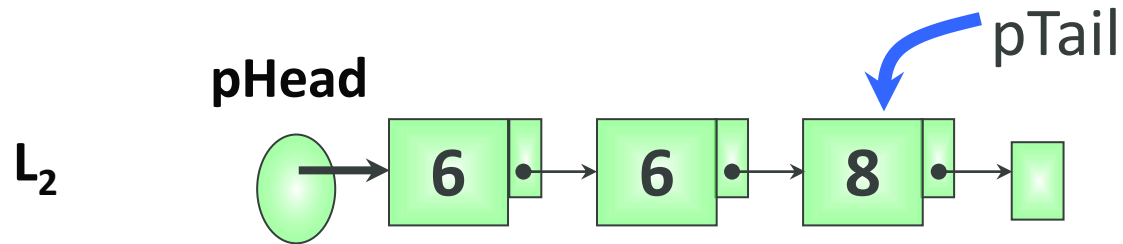
- Chọn $x=6$ làm chốt, và tách L_2 thành L_{21} và L_{22}



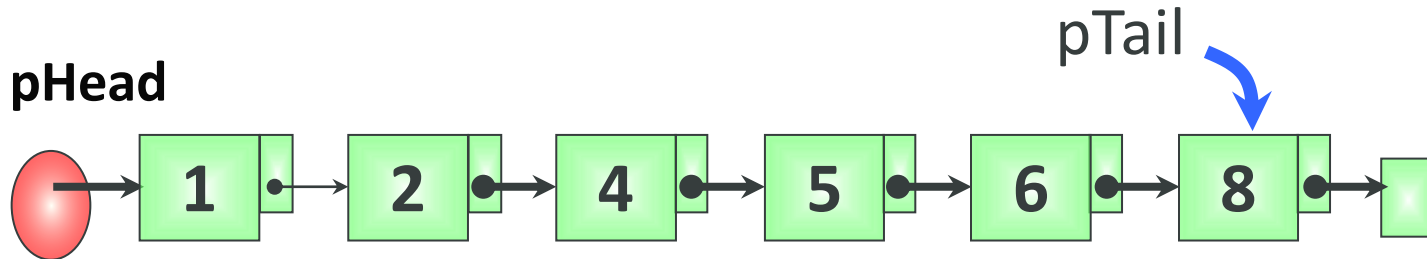
Quick Sort: Minh họa



➤ Nối L_{21} , X_2 , L_{22} thành L_2



➤ Nối L_1 , X , L_2 thành L





```
void QuickSort(LIST &L) {  
    LIST L1, L2;  
    CreateEmptyList(L1);  
    CreateEmptyList(L2);  
  
    NODE *pivot;  
    Partition(L, L1, pivot, L2);  
  
    if (L1.pHead != L1.pTail)  
        QuickSort(L1);  
    if (L2.pHead != L2.pTail)  
        QuickSort(L2);  
    Join(L, L1, pivot, L2);  
}
```



```
void Partition(LIST &L, LIST &L1, NODE *&pivot, LIST &L2) {  
    NODE *p;  
  
    if (L.pHead == NULL) return;  
  
    pivot = SeparateHead(L);  
  
    while (L.pHead != NULL) {  
        p = SeparateHead(L);  
        if (p->info <= pivot->info)  
            AddTail(L1, p);  
        else  
            AddTail(L2, p);  
    }  
}
```

Thuật toán: Code C/C++ (tt)



```
void Join(LIST &L1, LIST &L2, NODE *pivot, LIST &L3) {  
    NODE *p;  
  
    while (L2.pHead != NULL) {  
        p = SeparateHead(L2);  
        AddTail(L1, p);  
    }  
  
    AddTail(L1, pivot);  
  
    while (L3.pHead != NULL) {  
        p = SeparateHead(L3);  
        AddTail(L1, p);  
    }  
}
```



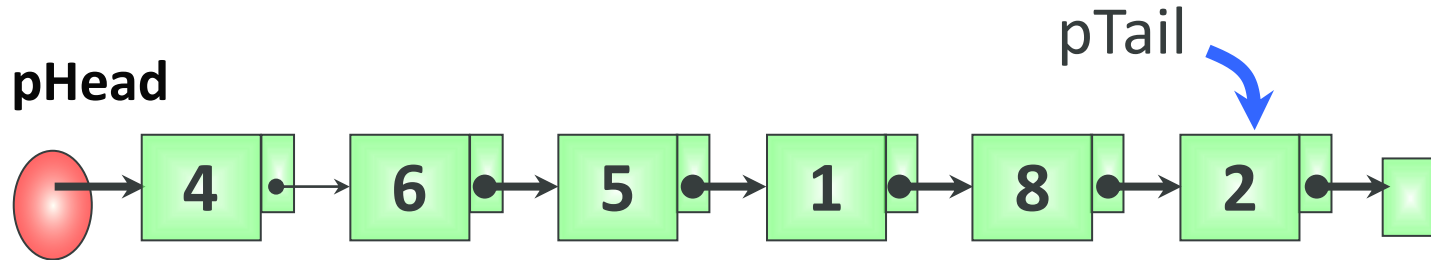
```
NODE* SeparateHead(LIST &L) {  
    NODE*p = L.pHead;  
    if (p == NULL) return NULL;  
  
    L.pHead = L.pHead->pNext;  
    if (L.pHead == NULL) L.pTail = NULL;  
  
    p->pNext = NULL;  
    return p;  
}
```



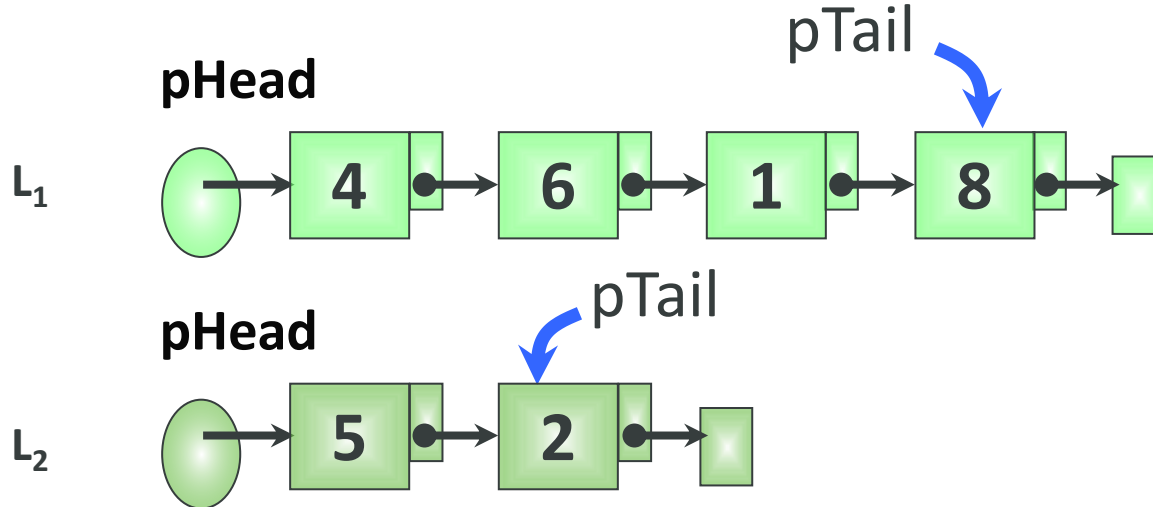
- Bước 1: Phân phối luân phiên từng đường chạy của xâu L vào 2 xâu con L_1 và L_2 .
- Bước 2: Nếu $L_1 \neq \text{NULL}$ thì Merge Sort (L_1).
- Bước 3: Nếu $L_2 \neq \text{NULL}$ thì Merge Sort (L_2).
- Bước 4: Trộn L_1 và L_2 đã sắp xếp lại ta có xâu L đã được sắp xếp.
- Không tốn thêm không gian lưu trữ cho các dãy phụ



- Cho danh sách liên kết gồm các phần tử sau:



- Phân phối các đường chạy của L vào L_1 , L_2

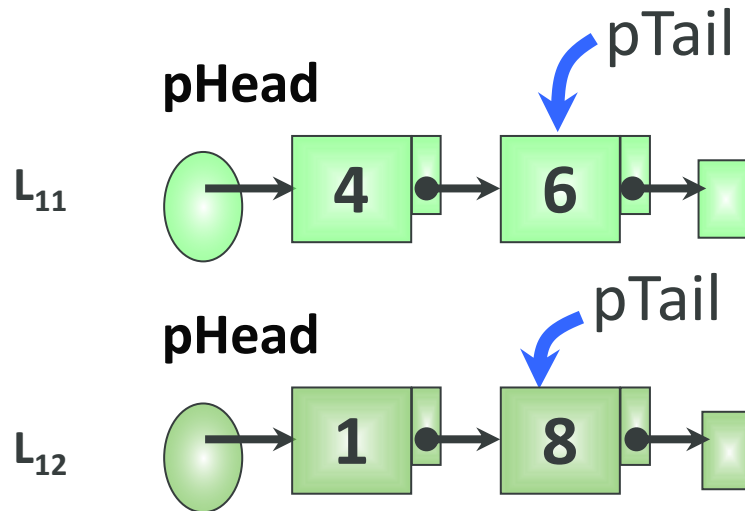


Minh họa thuật toán (tt)

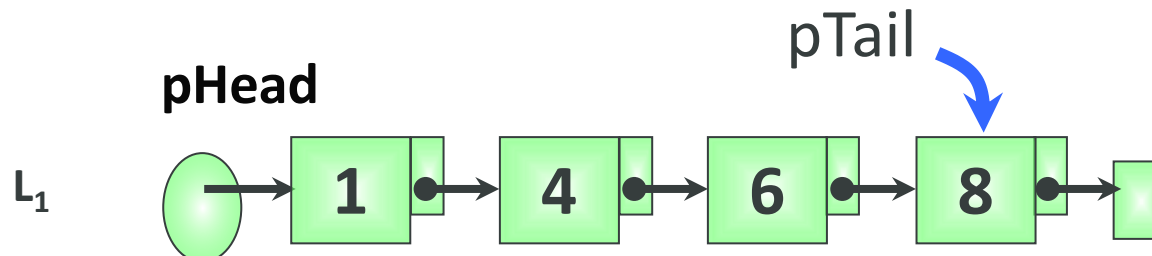


➤ Sắp xếp L_1

- Phân phối các đường chạy L_1 vào L_{11}, L_{12}



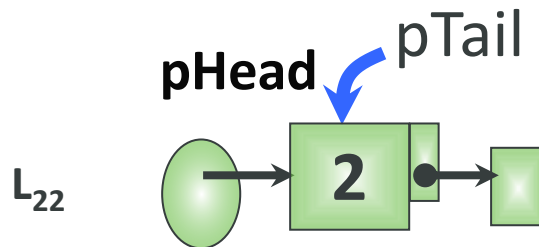
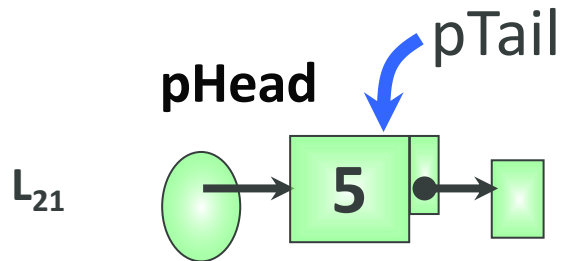
- Trộn L_{11} và L_{12} vào L_1



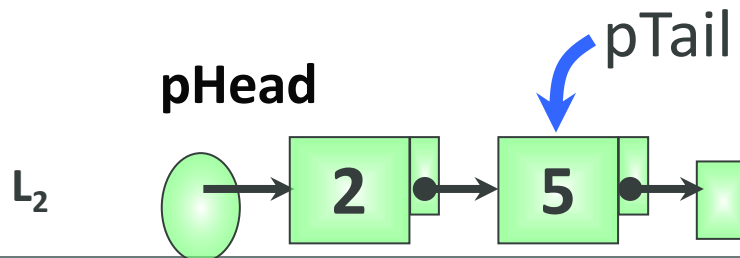


➤ Sắp xếp L_2

- Phân phối các đường chạy của L_2 vào L_{21}, L_{22}

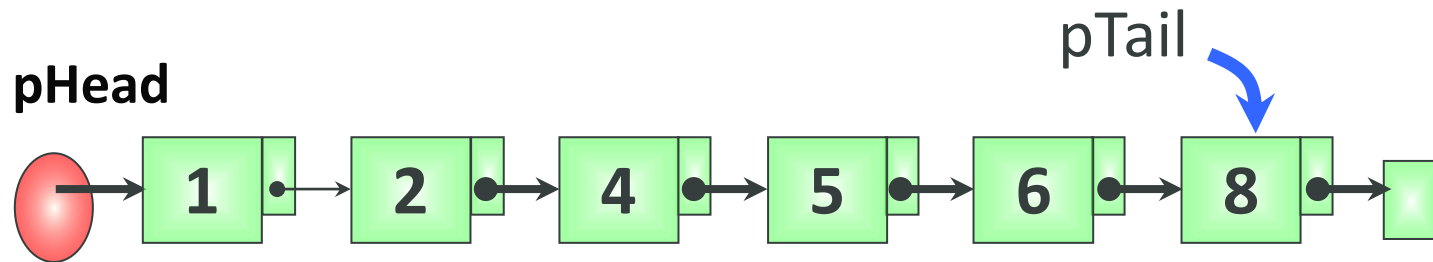


➤ Trộn L_{21}, L_{22} thành L_2





➤ Trộn L_1, L_2 thành L



Merge Sort: Code C/C++



```
void MergeSort(LIST &L) {  
    if (L.pHead == L.pTail) return;  
    LIST L1, L2;  
    MergeSplit(L, L1, L2);  
    MergeSort(L1);  
    MergeSort(L2);  
    Merge(L, L1, L2);  
}
```

Merge Sort: Code C/C++ (tt)



```
void MergeSplit(LIST &L, LIST &L1, LIST& L2) {  
    NODE *p;  
    CreateEmptyList(L1);  
    CreateEmptyList(L2);  
    while (L.pHead) {  
        p = SeparateHead(L);  
        AddTail(L1, p);  
        if (L.pHead) {  
            p = SeparateHead(L);  
            AddTail(L2, p);  
        }  
    }  
}
```

Merge Sort: Code C/C++ (tt)



```
void Merge(LIST &L, LIST &L1, LIST& L2) {  
    NODE *p;  
    CreateEmptyList(L);  
    while (L1.pHead!=NULL || L2.pTail!=NULL) {  
        if ( L2.pHead == NULL || (L1.pHead != NULL && L1.pHead->info  
        < L2.pHead->info) ) {  
            p = SeparateHead(L1);  
            AddTail(L, p);  
        }  
        else {  
            p = SeparateHead(L2);  
            AddTail(L, p);  
        }  
    }  
}
```

Sử dụng danh sách trong hàm main?



```
int main() {  
    LIST L;  
    CreateEmptyList(L);  
  
    AddTail(L, CreateNode(5));  
    AddTail(L, CreateNode(6));  
    AddTail(L, CreateNode(9));  
    AddTail(L, CreateNode(4));  
    AddTail(L, CreateNode(1));  
    AddTail(L, CreateNode(2));  
  
    cout << endl << "Linked list: "; PrintList(L);  
  
    QuickSort(L);  
    cout << endl << "After sorting: "; PrintList(L);  
  
    RemoveList(L);  
    cout << endl << "After removing all: "; PrintList(L);  
  
    cout << endl;  
    system("pause");  
}
```

```
Select C:\Users\nguye\onedrive\documents\visual studio 2015\Projects\Test CTDL\Debug\Te...  
Linked list: 5 6 9 4 1 2  
After sorting: 1 2 4 5 6 9  
After removing all: Empty List.  
Press any key to continue . . .
```

Cài đặt hàm main()



Yêu cầu:Viết chương trình thành lập 1 cây đơn, trong đó thành phần dữ liệu của mỗi nút là 1 số nguyên dương.

1. Liệt kê tất thành phần dữ liệu của tất cả các nút trong cây
2. Tìm 1 phần tử có khoá bằng x trong cây.
3. Xoá 1 phần tử đầu cây
4. Xoá 1 phần tử có khoá bằng x trong cây
5. Sắp xếp cây tăng dần theo thành phần dữ liệu (info)
6. Chèn 1 phần tử vào cây, sao cho sau khi chèn cây vẫn tăng dần theo trường dữ liệu

v.v..



- Dùng xâu đơn để lưu trữ danh sách các học viên trong lớp học
- Dùng xâu đơn để quản lý danh sách nhân viên trong một công ty, trong cơ quan
- Dùng xâu đơn để quản lý danh sách các cuốn sách trong thư viện
- Dùng xâu đơn để quản lý các băng đĩa trong tiệm cho thuê đĩa.
- V..V.

Dùng cấu trúc dữ liệu để quản lý lớp học



- Yêu cầu: Thông tin của một sinh viên gồm, mã số sinh viên, tên sinh viên, điểm trung bình.
- 1. Hãy khai báo cấu trúc dữ liệu dạng danh sách liên kết để lưu danh sách sinh viên nói trên.
- 2. Nhập danh sách các sinh viên, và thêm từng sinh viên vào đầu danh sách (việc nhập kết thúc khi tên của một sinh viên bằng rỗng)
- 3. Tìm một sinh viên có trong lớp học hay không
- 4. Xoá một sinh viên có mã số bằng x (x nhập từ bàn phím)
- 5. Liệt kê thông tin của các sinh viên có điểm trung bình lớn hơn hay bằng 5.



6. Xếp loại và in ra thông tin của từng sinh viên, biết rằng cách xếp loại như sau:

$\text{ĐTB} \leq 3.6$: Loại yếu

$\text{ĐTB} \geq 5.0$ và $\text{ĐTB} < 6.5$: Loại trung bình

$\text{ĐTB} \geq 6.5$ và $\text{ĐTB} < 7.0$: Loại trung bình khá

$\text{ĐTB} \geq 7.0$ và $\text{ĐTB} < 8.0$: Loại khá

$\text{ĐTB} \geq 8.0$ và $\text{ĐTB} < 9.0$: Loại giỏi

$\text{ĐTB} \geq 9.0$: Loại xuất sắc

7. Sắp xếp và in ra danh sách sinh viên tăng theo điểm trung bình.

8. Chèn một sinh viên vào danh sách sinh viên tăng theo điểm trung bình nói trên, sao cho sau khi chèn danh sách sinh viên vẫn tăng theo điểm trung bình

..vv



- Cấu trúc dữ liệu của một sinh viên

```
typedef struct {
    char ten[40];
    char Maso[40];
    float ĐTB;
} SV;
```
- Cấu trúc dữ liệu của 1 nút trong chuỗi

```
typedef struct tagNode {
    SV info;
    struct tagNode *pNext;
} Node;
```



1. Nêu các bước để thêm một nút vào đầu, giữa và cuối danh sách liên kết đơn.
2. Nêu các bước để xóa một nút ở đầu, giữa và cuối danh sách liên kết đơn.
3. Viết thủ tục để in ra tất cả các phần tử của 1 danh sách liên kết đơn.
4. Viết chương trình thực hiện việc sắp xếp 1 danh sách liên kết đơn bao gồm các phần tử là số nguyên.
5. Viết chương trình cộng 2 đa thức được biểu diễn thông qua danh sách liên kết đơn.



- Stack (ngăn xếp): Là 1 vật chứa các đối tượng làm việc theo cơ chế LIFO (Last In First Out), tức việc thêm 1 đối tượng vào Stack hoặc lấy 1 đối tượng ra khỏi Stack được thực hiện theo cơ chế “vào sau ra trước”
- Queue (hàng đợi): Là 1 vật chứa các đối tượng làm việc theo cơ chế FIFO (First In First Out), tức việc thêm 1 đối tượng vào hàng đợi hay lấy 1 đối tượng ra khỏi hàng đợi thực hiện theo cơ chế “vào trước ra trước”.



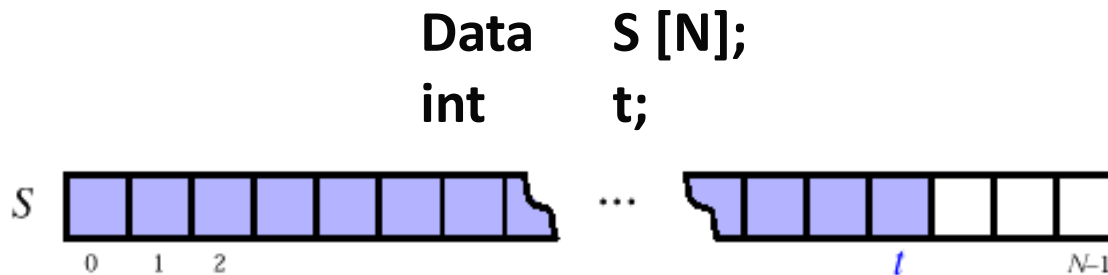
- Stack:
 - Trình biên dịch
 - Khử đệ qui đuôi
 - Lưu vết các quá trình quay lui, vết cạn
- Queue:
 - Tổ chức lưu vết các quá trình tìm kiếm theo chiều rộng, và quay lui vết cạn
 - Tổ chức quản lý và phân phối tiến trình trong các hệ điều hành.
 - Tổ chức bộ đệm bàn phím, ...



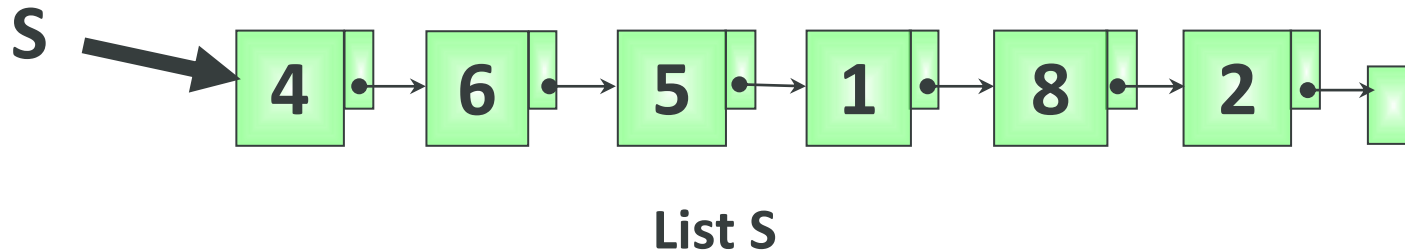
- `Push(o)`: Thêm đối tượng `o` vào Stack
- `Pop()`: Lấy đối tượng từ Stack
- `isEmpty()`: Kiểm tra Stack có rỗng hay không
- `isFull()`: Kiểm tra Stack có đầy hay không
- `Top()`: Trả về giá trị của phần tử nằm đầu Stack mà không hủy nó khỏi Stack.



- Dùng mảng 1 chiều



- Dùng danh sách liên kết đơn



- Chú ý: Thêm và hủy cùng phía



- Cấu trúc dữ liệu của Stack

```
struct Stack {  
    int a[MAX];  
    int t;  
};
```

- Khởi tạo Stack

```
void CreateStack(Stack &s) {  
    s.t = -1;  
}
```


Kiểm tra tính rỗng và đầy của Stack



```
bool isEmpty(Stack s) { //Stack có rỗng hay không
    if (s.t == -1)
        return 1;
    return 0;
}
```

```
bool isFull(Stack s) { //Kiểm tra Stack có đầy hay không
    if (s.t >= MAX)
        return 1;
    return 0;
}
```

Thêm 1 phần tử vào Stack



```
bool Push(Stack &s, int x) {  
    if (isFull(s) == 0)  
        s.a[++s.t] = x;  
    return 1;  
}  
return 0;  
}
```

Lấy 1 phần tử từ Stack



```
int Pop(Stack &s, int &x) {  
    if (isEmpty(s) == 0) {  
        x = s.a[s.t--];  
        return 1;  
    }  
    return 0;  
}
```



- Kiểm tra tính rỗng của Stack

```
int isEmpty(LIST &s) {  
    if (s.pHead == NULL) // Stack rỗng  
        return 1;  
    return 0;  
}
```

Thêm 1 phần tử vào Stack



```
void Push(LIST &s, NODE *p) { // AddHead
    if (s.pHead == NULL) {
        s.pHead = p;
        s.pTail = p;
    }
    else {
        p->pNext = s.pHead;
        s.pHead = p;
    }
}
```

Lấy 1 phần tử từ Stack



```
bool Pop(LIST &s, int &x) {  
    NODE *p;  
    if (isEmpty(s) != 1) {  
        if (s.pHead != NULL) {  
            p = s.pHead;  
            x = p->info;  
            s.pHead = s.pHead->pNext;  
            if (s.pHead == NULL)  
                s.pTail = NULL;  
            return 1;  
        }  
    }  
    return 0;  
}
```

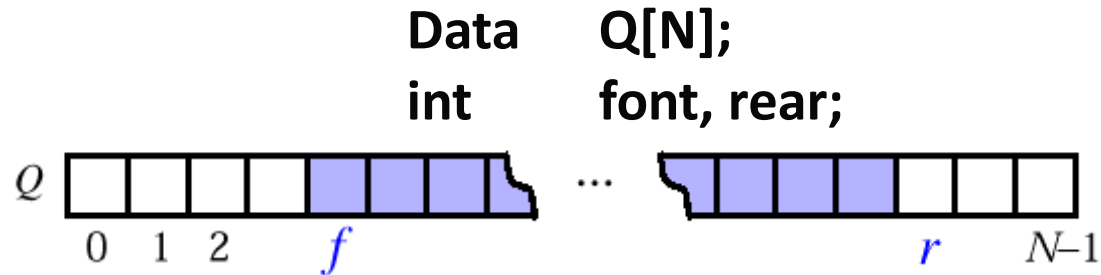


- `EnQueue(O)`: Thêm đối tượng `O` vào cuối hàng đợi.
- `DeQueue()`: Lấy đối tượng ở đầu hàng đợi
- `isEmpty()`: Kiểm tra xem hàng đợi có rỗng hay không?
- `Front()`: Trả về giá trị của phần tử nằm đầu hàng đợi mà không hủy nó.

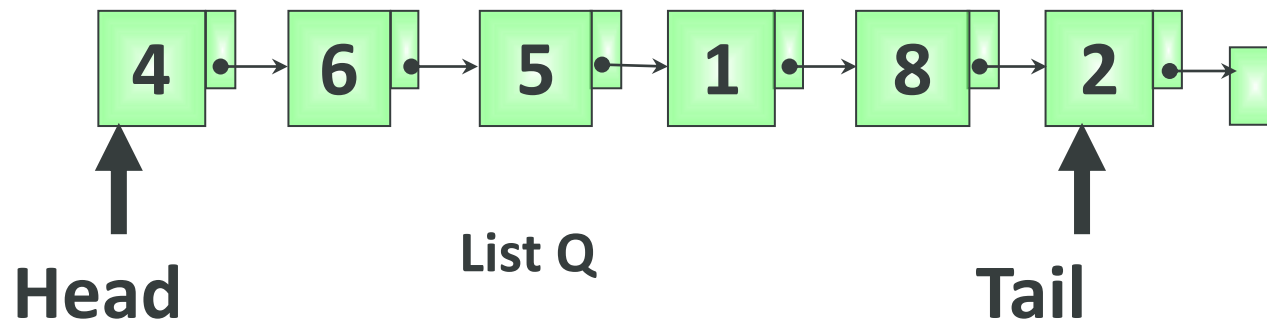
Cài đặt Queue



- Dùng mảng 1 chiều



- Dùng danh sách liên kết đơn



- Chú ý: **Thêm và hủy Khác phía**

Cài đặt Queue bằng mảng 1 chiều



- Cấu trúc dữ liệu:

```
typedef struct tagQueue {  
    int a[MAX];  
    int Front; //chỉ số của phần tử đầu trong Queue  
    int Rear; //chỉ số của phần tử cuối trong Queue  
} Queue;
```

- Khởi tạo Queue rỗng

```
void CreateQueue(Queue &q) {  
    q.Front = -1;  
    q.Rear = -1;  
}
```

Kiểm tra tính rỗng và đầy của Queue



```
bool isEmpty(Queue q) { // Queue có rỗng?  
    if (q.Front == -1)  
        return 1;  
    return 0;  
}
```

```
bool isFull(Queue q) { // Kiểm tra Queue có đầy?  
    if (q.Rear - q.Front + 1 == MAX)  
        return 1;  
    return 0;  
}
```

Thêm 1 phần tử vào Queue



```
void EnQueue(Queue &q, int x) {
    int f, r;
    if (isFull(q)) //queue bị đầy => không thêm được nữa
        printf("queue day roi khong the them vao duoc nua");
    else {
        if (q.Front == -1) {
            q.Front = 0;
            q.Rear = -1;
        }
        if (q.Rear == MAX - 1) { //Queue đầy ảo
            f = q.Front;
            r = q.Rear;
            for (int i = f; i <= r; i++)
                q.a[i - f] = q.a[i];
            q.Front = 0;
            q.Rear = r - f;
        }
        q.Rear++;
        q.a[q.Rear] = x;
    }
}
```

Lấy 1 phần tử từ Queue



```
bool DeQueue(Queue &q, int &x) {  
    if (isEmpty(q)==0) { //queue không rỗng  
        x = q.a[q.Front];  
        q.Front++;  
        if (q.Front>q.Rear) { //trường hợp có một phần tử  
            q.Front = -1;  
            q.Rear = -1;  
        }  
        return 1;  
    }  
    //queue trống  
    printf("Queue rỗng");  
    return 0;  
}
```

Cài đặt Queue bằng List



- Kiểm tra Queue có rỗng?

```
bool isEmpty(LIST &Q) {  
    if (Q.pHead == NULL) //Queue rỗng  
        return 1;  
    return 0;  
}
```

Thêm 1 phần tử vào Queue



```
void EnQueue(LIST &Q, NODE *p) {  
    if (Q.pHead == NULL) {  
        Q.pHead = p;  
        Q.pTail = p;  
    }  
    else {  
        Q.pTail->pNext = p;  
        Q.pTail = p;  
    }  
}
```

Lấy 1 phần tử từ Queue



```
int DeQueue(LIST &Q, int &x) {  
    NODE *p;  
    if (isEmpty(Q) != 1) {  
        if (Q.pHead != NULL) {  
            p = Q.pHead;  
            x = p->info;  
            Q.pHead = Q.pHead->pNext;  
            if (Q.pHead == NULL)  
                Q.pTail = NULL;  
            return 1;  
        }  
    }  
    return 0;  
}
```



1. Hãy nêu ưu điểm và hạn chế của CTDL tĩnh.
2. Hãy nêu ưu điểm và hạn chế của CTDL động.
3. Danh sách là gì? Cho ví dụ.
4. Hãy nêu ưu điểm và nhược điểm của danh sách liên kết ngầm và danh sách liên kết tường minh.



Chúc các em học tốt!

