Kỹ thuật lập trình

Chương 4: Khái quát về cấu trúc dữ liệu

10/16/2007

Nội dung chương 4

4.1 Giới thiệu chung

- Phần lớn các bài toán trong thực tế liên quan tới các dữ liệu phức hợp, những kiểu dữ liệu cơ bản trong ngôn ngữ lập trình không đủ biểu diễn
- Ví dụ:
  - Dữ liệu sinh viên: Họ tên, ngày sinh, quê quán, mã số SV,...
  - Mô hình hàm truyền: Đa thức tử số, đa thức mẫu số
  - Mô hình trạng thái: Các ma trận A, B, C, D
  - Dữ liệu quá trình: Tên đại lượng, dải đo, giá trị, đơn vị, thời gian, cấp sai số, ngưỡng giá trị,...
  - Đối tượng đồ họa: Kích thước, màu sắc, đường nét, phông chữ, ...
- Phương pháp biểu diễn dữ liệu: định nghĩa kiểu dữ liệu mới sử dụng cấu trúc (struct, class, union, ...)

Vấn đề: Biểu diễn tập hợp dữ liệu

- Đa số những dữ liệu thuộc một ứng dụng có liên quan với nhau => cần biểu diễn trong một tập hợp có cấu trúc, ví dụ:
  - Danh sách sinh viên: Các dữ liệu sinh viên được sắp xếp theo thứ tự Alphabet
  - Mộ hình tổng thể cho hệ thống điều khiển: Bao gồm nhiều thành phần tương tác
  - Dữ liệu quá trình: Một tập dữ liệu có thể mang giá trị của một đại lượng vào các thời điểm gián đoạn, các dữ liệu đầu vào liên quan tới dữ liệu đầu ra
  - Đối tượng đồ họa: Một cửa sổ bao gồm nhiều đối tượng đồ họa, một bản vẽ cũng bao gồm nhiều đối tượng đồ họa
- Thông thường, các dữ liệu trong một tập hợp có cùng kiểu, hoặc ít ra là tương thích kiểu với nhau
- Kiểu mảng không phải bao giờ cũng phù hợp!

Vấn đề: Quản lý (tập hợp) dữ liệu

- Sử dụng kết hợp một cách khéo léo kiểu cấu trúc và kiểu mảng đủ để biểu diễn các tập hợp dữ liệu bất kỳ
- Các giải thuật (hàm) thao tác với dữ liệu, nhằm quản lý dữ liệu một cách hiệu quả:
  - Bổ sung một mục dữ liệu mới vào một danh sách, một bảng, một tập hợp, ...
  - Xóa một mục dữ liệu trong một danh sách, bảng, tập hợp,..
  - Tìm một mục dữ liệu trong một danh sách, bảng tập hợp,... theo một tiêu chuẩn cụ thể
  - Sắp xếp một danh sách theo một tiêu chuẩn nào đó
  - ....

Quản lý DL thế nào là hiệu quả?

- Tiết kiệm bộ nhớ: Phần "overhead" không đáng kể so với phần dữ liệu thực
- Truy nhập nhanh, thuận tiện: Thời gian cần cho bổ sung, tìm kiếm và xóa bỏ các mục dữ liệu phải ngắn
- Linh hoạt: Số lượng các mục dữ liệu không (hoặc ít) bị hạn chế cố định, không cần biết trước khi tạo cấu trúc, phù hợp với cả bài toán nhỏ và lớn
- Hiệu quả quản lý dữ liệu phụ thuộc vào
  - Cấu trúc dữ liệu được sử dụng
  - Giải thuật được áp dụng cho bổ sung, tìm kiếm, sắp xếp, xóa bỏ

Các cấu trúc dữ liệu thông dụng

- Mảng (nghĩa rộng): Tập hợp các dữ liệu có thể truy nhập tùy ý theo chỉ số
- Danh sách (list): Tập hợp các dữ liệu được móc nối đôi một với nhau và có thể truy nhập tuần tự
- Cây (tree): Tập hợp các dữ liệu được móc nối với nhau theo cấu trúc cây, có thể truy nhập tuần tự từ gốc
  - Nếu mỗi nút có tối đa hai nhánh: cây nhị phân (binary tree)
- Bìa, bảng (map): Tập hợp các dữ liệu có sắp xếp, có thể truy nhập rất nhanh theo mã khóa (key)
- Hàng đợi (queue): Tập hợp các dữ liệu có sắp xếp tuần tự, chỉ bổ sung vào từ một đầu và lấy ra từ đầu còn lại

Các cấu trúc dữ liệu thông dụng (tiếp)

- Tập hợp (set): Tập hợp các dữ liệu được sắp xếp tùy ý nhưng có thể truy nhập một cách hiệu quả
- Ngăn xếp (stack): Tập hợp các dữ liệu được sắp xếp tuần tự, chỉ truy nhập được từ một đầu
- Bảng hash (hash table): Tập hợp các dữ liệu được sắp xếp dựa theo một mã số nguyên tạo ra từ một hàm đặc biệt
- Bộ nhớ vòng (ring buffer): Tương tự như hàng đợi, nhưng dung lượng có hạn, nếu hết chỗ sẽ được ghi quay vòng
- Trong toán học và trong điều khiển: vector, ma trận, đa thức, phân thức, hàm truyền, ...

4.2 Mảng và quản lý bộ nhớ động

- Mảng cho phép biểu diễn và quản lý dữ liệu một cách khá hiệu quả:
  - Đọc và ghi dữ liệu rất nhanh qua chỉ số hoặc qua địa chỉ
  - Tiết kiệm bộ nhớ
- Các vấn đề của mảng tĩnh:

```
VD: Student student_list[100];
```

  - Số phần tử phải là hằng số (biết trước khi biên dịch, người sử dụng không thể nhập số phần tử, không thể cho số phần từ là một biến) => kém linh hoạt
  - Chiếm chỗ cứng trong ngăn xếp (đối với biến cục bộ) hoặc trong bộ nhớ dữ liệu chương trình (đối với biến toàn cục) => sử dụng bộ nhớ kém hiệu quả, kém linh hoạt

Mảng động

- Mảng động là một mảng được cấp phát bộ nhớ theo yêu cầu, trong khi chương trình chạy

```
#include <stdlib.h>     /* C */
int n = 50;
...
float*  p1= (float*) malloc(n*sizeof(float)); /* C */
double* p2= new double[n];     // C++
```

- Sử dụng con trỏ để quản lý mảng động: Cách sử dụng không khác so với mảng tĩnh

```
p1[0] = 1.0f;
p2[0] = 2.0;
```

- Sau khi sử dụng xong => giải phóng bộ nhớ:

```
free(p1); /* C */
delete [] p2;    // C++
```

Cấp phát và giải phóng bộ nhớ động

- C:
  - Hàm **`malloc()`** yêu cầu tham số là **số byte**, trả về con trỏ không kiểu **`(void*)`** mang địa chỉ vùng nhớ mới được cấp phát (nằm trong heap), trả về 0 nếu không thành công.
  - Hàm **`free()`** yêu cầu tham số là con trỏ không kiểu (**`void*`**), giải phóng vùng nhớ có địa chỉ đưa vào
- C++:
  - Toán tử **`new`** chấp nhận kiểu dữ liệu phần tử kèm theo số lượng phần tử của mảng cần cấp phát bộ nhớ (trong vùng heap), trả về con trỏ có kiểu, trả về 0 nếu không thành công.
  - Toán tử **`delete[]`** yêu cầu tham số là con trỏ có kiểu.
  - Toán tử **`new`** và **`delete`** còn có thể áp dụng cho cấp phát và giải phóng bộ nhớ cho một biến đơn, một đối tượng chứ không nhất thiết phải một mảng.

Một số điều cần lưu ý

- Con trỏ có vai trò quản lý mảng (động), chứ con trỏ không phải là mảng (động)
- Cấp phát bộ nhớ và giải phóng bộ nhớ chứ không phải cấp phát con trỏ và giải phóng con trỏ
- Chỉ giải phóng bộ nhớ một lần

```
int* p;
p[0] = 1;                    // never do it
new(p);                      // access violation!
p = new int[100];            // OK
p[0] = 1;                    // OK
int* p2=p;          // OK
delete[] p2;                 // OK
p[0] = 1;                    // access violation!
delete[] p;                  // very bad!
p = new int[50];             // OK, new array
…
```

Cấp phát bộ nhớ động cho biến đơn

- **Ý nghĩa**: Các đối tượng có thể được tạo ra động, trong khi chương trình chạy (bổ sung sinh viên vào danh sách, vẽ thêm một hình trong bản vẽ, bổ sung một khâu trong hệ thống,...)
- Cú pháp

```
int* p = new int;
*p = 1;
p[0]= 2;                    // the same as above
p[1]= 1;                    // access violation!
int* p2 = new int(1);       // with initialization
delete p;
delete p2;
Student* ps = new Student;
ps->code = 1000;
...
delete ps;
```

- Một biến đơn **khác** với mảng một phần tử!

Ý nghĩa của sử dụng bộ nhớ động

- Hiệu suất:
  - Bộ nhớ được cấp phát đủ dung lượng theo yêu cầu và khi được yêu cầu trong khi chương trình đã chạy
  - Bộ nhớ được cấp phát nằm trong vùng nhớ tự do còn lại của máy tính (heap), chỉ phụ thuộc vào dung lượng bộ nhớ của máy tính
  - Bộ nhớ có thể được giải phóng khi không sử dụng tiếp.
- Linh hoạt:
  - Thời gian "sống" của bộ nhớ được cấp phát động có thể kéo dài hơn thời gian "sống" của thực thể cấp phát nó.
  - Có thể một hàm gọi lệnh cấp phát bộ nhớ, nhưng một hàm khác giải phóng bộ nhớ.
  - Sự linh hoạt cũng dễ dẫn đến những lỗi "rò rỉ bộ nhớ".

Ví dụ sử dụng bộ nhớ động trong hàm

```
Date* createDateList(int n) {
   Date* p = new Date[n];
   return p;
}
void main() {
   int n;
   cout << "Enter the number of your national holidays:";
   cin >> n;
   Date* date_list = createDateList(n);
   for (int i=0; i < n; ++i) {
        ...
   }
   for (....) { cout << ....}
   delete [] date_list;
}
```

Tham số đầu ra là con trỏ?

```
void createDateList(int n, Date* p) {
   p = new Date[n];
}
void main() {
   int n;
   cout << "Enter the number of your national holidays:";
   cin >> n;
   Date* date_list;
   createDateList(n, date_list);
   for (int i=0; i < n; ++i) {
        ...
   }
   for (....) { cout << ....}
   delete [] date_list;
}
```

4.3 Xây dựng cấu trúc Vector

- Vấn đề: Biểu diễn một vector toán học trong C/C++?
- Giải pháp chân phương: mảng động thông thường, nhưng...
  - Sử dụng không thuận tiện: Người sử dụng tự gọi các lệnh cấp phát và giải phóng bộ nhớ, trong các hàm luôn phải đưa tham số là số chiều.
  - Sử dụng không an toàn: Nhầm lẫn nhỏ dẫn đến hậu quả nghiêm trọng

```
int n = 10;
double *v1,*v2, d;
v1 = (double*) malloc(n*sizeof(double));
v2 = (double*) malloc(n*sizeof(double));
d = scalarProd(v1,v2,n); // scalar_prod đã có
d = v1 * v2;             // OOPS!
v1.data[10] = 0;         // OOPS!
free(v1);
free(v2);
```

Định nghĩa cấu trúc Vector

- Tên file: **vector.h**
- Cấu trúc dữ liệu:

```
struct Vector {
   double *data;
   int     nelem;
};
```

- Khai báo các hàm cơ bản:

```
Vector createVector(int n, double init);
void   destroyVector(Vector);
double getElem(Vector, int i);
void   putElem(Vector, int i, double d);
Vector addVector(Vector, Vector);
Vector subVector(Vector, Vector);
double scalarProd(Vector, Vector);
…
```

Định nghĩa các hàm cơ bản

- Tên file: **vector.cpp**

```
#include <stdlib.h>
#include "vector.h"
Vector createVector(int n, double init) {
   Vector v;
   v.nelem = n;
   v.data = (double*) malloc(n*sizeof(double));
   while (n--) v.data[n] = init;
   return v;
}
void destroyVector(Vector v) {
   free(v.data);
}
double getElem(Vector v, int i) {
   if (i < v.nelem && i >= 0) return v.data[i];
   return 0;
}
```

```
void putElem(Vector v, int i, double d) {
   if (i >=0 && i < v.nelem) v.data[i] = d;
}
Vector addVector(Vector a, Vector b) {
   Vector c = {0,0};
   if (a.nelem == b.nelem) {
        c = createVector(a.nelem,0.0);
        for (int i=0; i < a.nelem; ++i)
          c.data[i] = a.data[i] + b.data[i];
   }
   return c;
}
Vector subVector(Vector a, Vector b) {
   Vector c = {0,0};
   ...
   return c;
}
```

Ví dụ sử dụng

```
#include "vector.h"
void main() {
   int n = 10;
   Vector a,b,c;
   a = createVector(10,1.0);
   b = createVector(10,2.0);
   c = addVector(a,b);
   //...
   destroyVector(a);
   destroyVector(b);
   destroyVector(c);
}
```

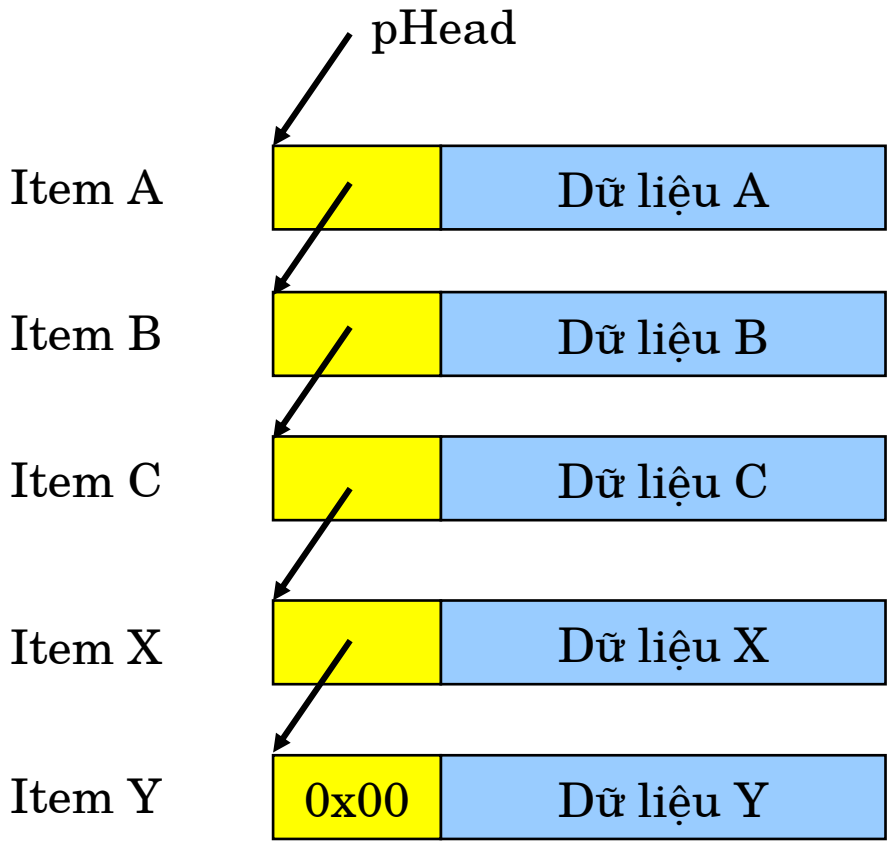
4.4 Xây dựng cấu trúc List

- Vấn đề: Xây dựng một cấu trúc để quản lý một cách hiệu quả và linh hoạt các dữ liệu động, ví dụ:
  - Hộp thư điện tử
  - Danh sách những việc cần làm
  - Các đối tượng đồ họa trên hình vẽ
  - Các khâu động học trong sơ đồ mô phỏng hệ thống (tương tự trong SIMULINK)
- Các yêu cầu đặc thù:
  - Số lượng mục dữ liệu trong danh sách có thể thay đổi thường xuyên
  - Các thao tác bổ sung hoặc xóa dữ liệu cần được thực hiện nhanh, đơn giản
  - Sử dụng tiết kiệm bộ nhớ

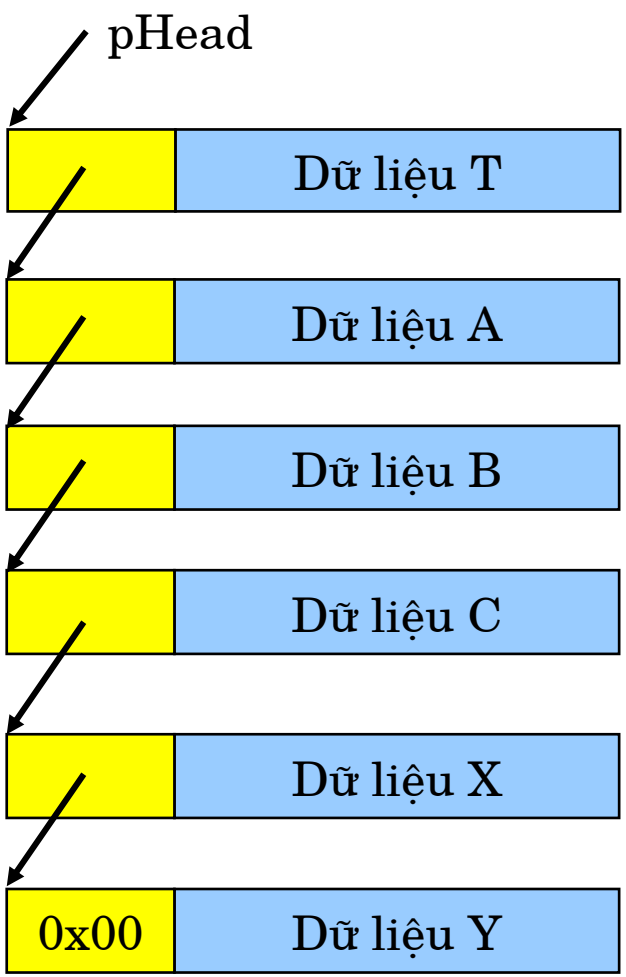
Sử dụng kiểu mảng?

- Số phần tử trong một mảng thực chất không bao giờ thay đổi được. Dung lượng bộ nhớ vào thời điểm cấp phát phải biết trước, không thực sự co giãn được.
- Nếu không thực sự sử dụng hết dung lượng đã cấp phát => lãng phí bộ nhớ
- Nếu đã sử dụng hết dung lượng và muốn bổ sung phần tử thì phải cấp phát lại và sao chép toàn bộ dữ liệu sang mảng mới => cần nhiều thời gian nếu số phần tử lớn
- Nếu muốn chèn một phần tử/xóa một phần tử ở đầu hoặc giữa mảng thì phải sao chép và dịch toàn bộ phần dữ liệu còn lại => rất mất thời gian

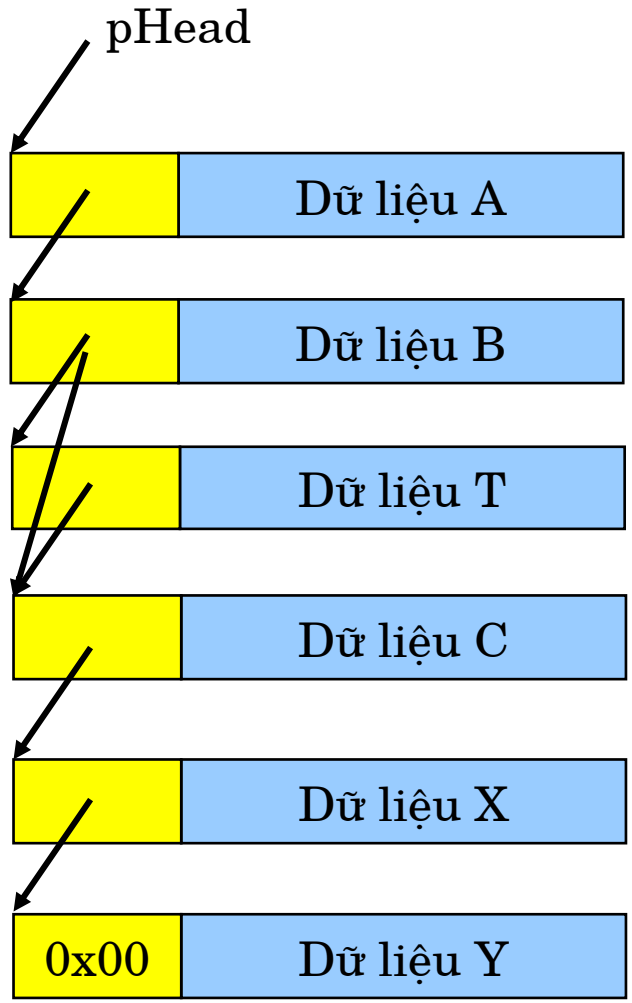
Danh sách móc nối (linked list)



Bổ sung dữ liệu

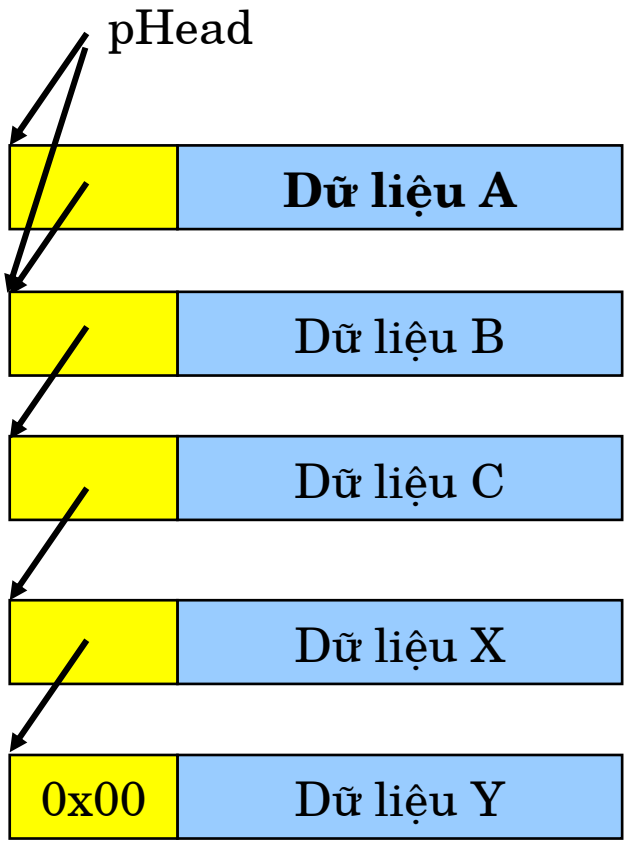


Bổ sung vào đầu danh sách

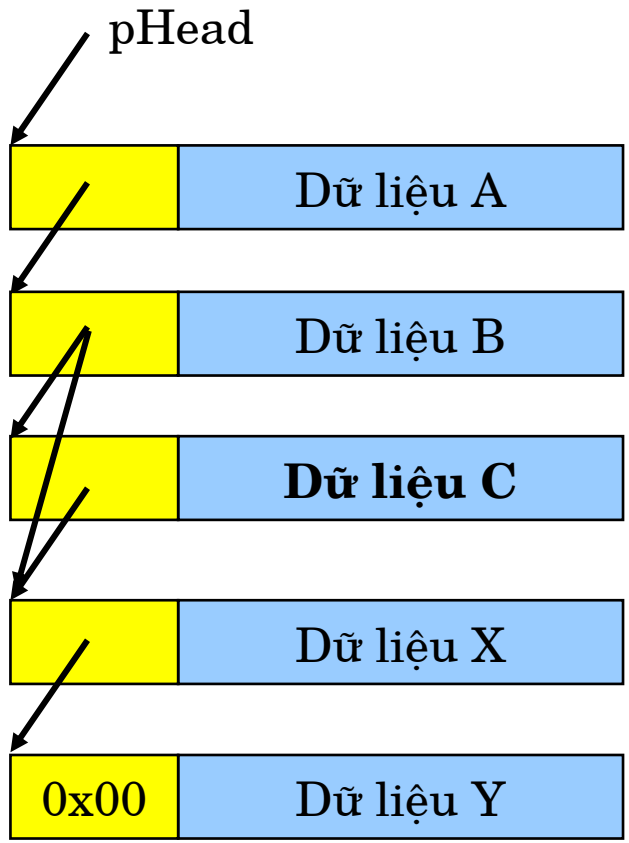


Bổ sung vào giữa danh sách

Xóa bớt dữ liệu



Xóa dữ liệu đầu danh sách



Xóa dữ liệu giữa danh sách

Các đặc điểm chính

- Ưu điểm:
  - Sử dụng rất linh hoạt, cấp phát bộ nhớ khi cần và xóa khi không cần
  - Bổ sung và xóa bỏ một dữ liệu được thực hiện thông qua chuyển con trỏ, thời gian thực hiện là hằng số, không phụ thuộc vào chiều dài và vị trí
  - Có thể truy nhập và duyệt các phần tử theo kiểu tuần tự
- Nhược điểm:
  - Mỗi dữ liệu bổ sung mới đều phải được cấp phát bộ nhớ động
  - Mỗi dữ liệu xóa bỏ đi đều phải được giải phóng bộ nhớ tương ứng
  - Nếu kiểu dữ liệu không lớn thì phần overhead chiếm tỉ lệ lớn
  - Tìm kiếm dữ liệu theo kiểu tuyến tính, mất thời gian

Ví dụ: Danh sách thông báo (hộp thư)

```
#include <string>
using namespace std;

struct MessageItem {
   string subject;
   string content;
   MessageItem* pNext;
};
struct MessageList {
   MessageItem* pHead;
};
void initMessageList(MessageList& l);
void addMessage(MessageList&, const string& sj,
                const string& ct);
bool removeMessageBySubject(MessageList& l,
                            const string& sj);
void removeAllMessages(MessageList&);
```

```
#include "List.h"
void initMessageList(MessageList& l) {
   l.pHead = 0;
}
void addMessage(MessageList& l, const string& sj,
                const string& ct) {
   MessageItem* pItem = new MessageItem;
   pItem->content = ct;
   pItem->subject = sj;
   pItem->pNext = l.pHead;
   l.pHead = pItem;
}
void removeAllMessages(MessageList& l) {
   MessageItem *pItem = l.pHead;
   while (pItem != 0) {
        MessageItem* pItemNext = pItem->pNext;
        delete pItem;
        pItem = pItemNext;
   }
   l.pHead = 0;
}
```

```
bool removeMessageBySubject(MessageList& l,
                            const string& sj) {
   MessageItem* pItem = l.pHead;
   MessageItem* pItemBefore;
   while (pItem != 0 && pItem->subject != sj) {
        pItemBefore = pItem;
        pItem = pItem->pNext;
   }
   if (pItem != 0) {
        if (pItem == l.pHead)
               l.pHead = 0;
        else
               pItemBefore->pNext = pItem->pNext;
        delete pItem;
   }
   return pItem != 0;
}
```

Chương trình minh họa

```
#include <iostream>
#include "list.h"
using namespace std;
void main() {
   MessageList myMailBox;
   initMessageList(myMailBox);
   addMessage(myMailBox,"Hi","Welcome, my friend!");
   addMessage(myMailBox,"Test","Test my mailbox");
   addMessage(myMailBox,"Lecture Notes","Programming Techniques");
   removeMessageBySubject(myMailBox,"Test");
   MessageItem* pItem = myMailBox.pHead;
   while (pItem != 0) {
        cout << pItem->subject << ":" << pItem->content << '\n';
        pItem = pItem->pNext;
   }
   char c;
   cin >> c;
   removeAllMessages(myMailBox);
}
```

Bài tập về nhà

- Xây dựng kiểu danh sách móc nối chứa các ngày lễ trong năm và ý nghĩa của mỗi ngày (string), cho phép:
  - Bổ sung một ngày lễ vào đầu danh sách
  - Tìm ý nghĩa của một ngày (đưa ngày tháng là tham số)
  - Xóa bỏ đi một ngày lễ ở đầu danh sách
  - Xóa bỏ đi một ngày lễ ở giữa danh sách (đưa ngày tháng là tham số)
  - Xóa bỏ đi toàn bộ danh sách
- Viết chương trình minh họa cách sử dụng