Bài 3. Danh sách liên kết

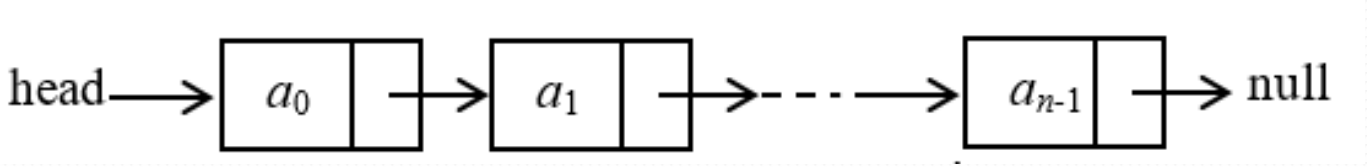
Nội dung

- Danh sách liên kết đơn
- Danh sách liên kết vòng
- Danh sách hai liên kết
- Bài tập

Danh sách liên kết đơn (Single Linked List)

Giới thiệu Danh sách liên kết đơn

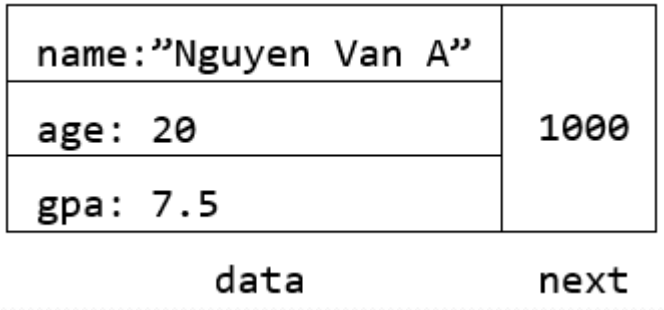
- Mục đích:
  - Lưu trữ và xử lý danh sách
  - Khắc phục hạn chế của mảng
- Cơ chế:
  - Dùng ô nhớ cấp phát động
  - Các ô nhớ không liên tiếp nhau
  - Cần lưu địa chỉ để liên kết các phần tử và xác định thứ tự



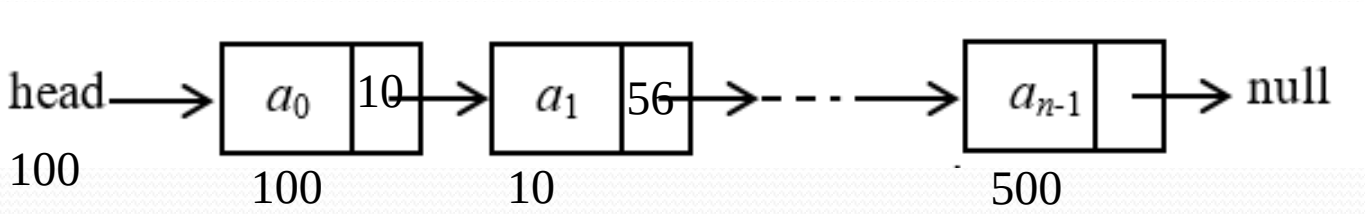
Tổ chức dữ liệu

- data: lưu dữa liệu 1 phần tử
- next: lưu địa chỉ phần tử tiếp theo
- Để quản lý danh sách cần biết địa chỉ phần tử đầu head

```
struct Student
{
 string name;
 int  age;
 double gpa;
};
struct SLLStudents{
 Student data;
 SLLStudents *next;
};
```




```
struct SLLNode{
 ElementType data;
 SLLNode     *next;
};
```



Các thao tác

- Khởi tạo: gán phần tử đầu rỗng.

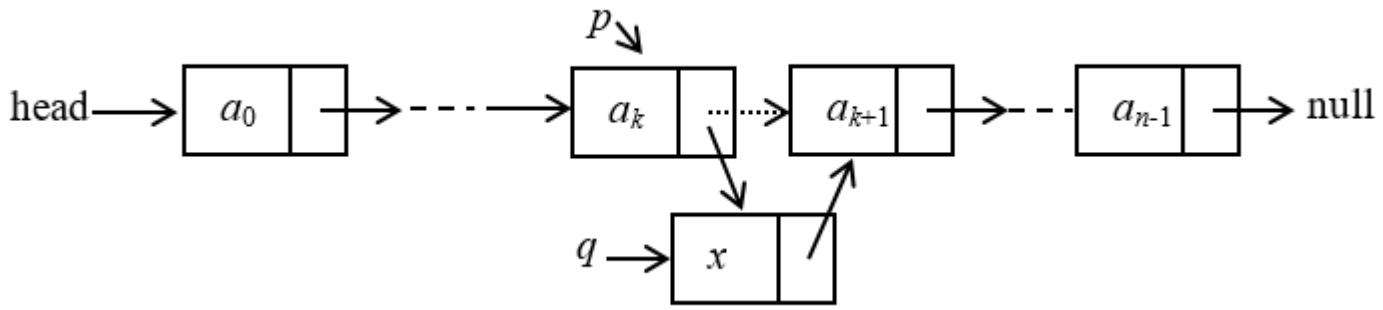
```
void initialize(SLLNode* &head)
{
   head = nullptr;
}
```

Thêm một phần tử

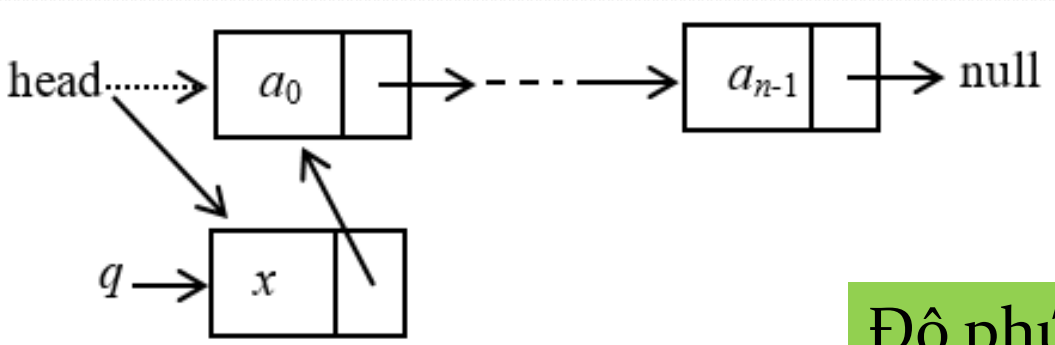
Input: Phần tử cần thêm x, vị trí p
Output: Danh sách sau khi thêm
Action:
+ Tạo phần tử q của danh sách liên kết đơn.
+ Đưa x vào dữ liệu của q.
+ Cho q liên kết đến phần tử sau p.
+ Cho p liên kết đến q.

```
void insert(SLLNode *p, ElementType x)
{
   SLLNode *q = new SLLNode;
   q->data = x;
   q->next = p->next;
   p->next = q;
}
```

Độ phức tạp thuật toán O(1)



Thêm phần tử đầu

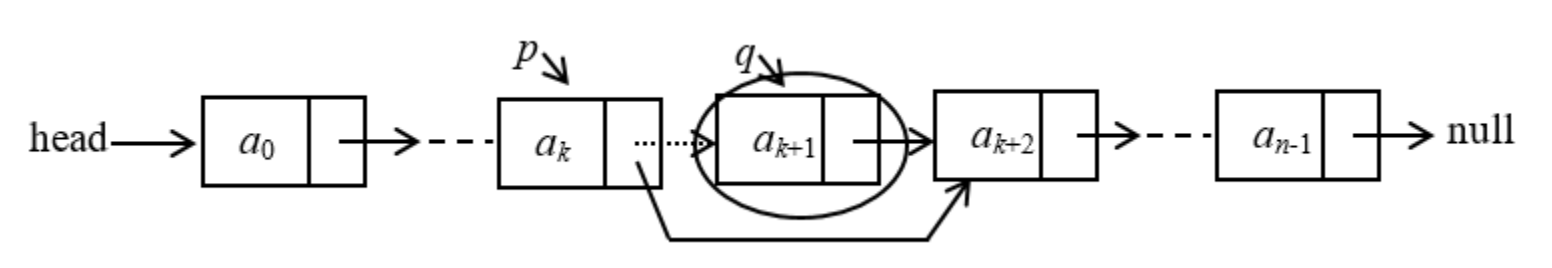


Độ phức tạp thuật toán O(1)

Input: Phần tử đầu danh sách cần thêm, phần cần thêm x
Output: Danh sách sau khi thêm
Action:
+ Tạo phần tử q của danh sách liên kết đơn.
+ Đưa x vào dữ liệu của q.
+ Cho q liên kết đến phần tử đầu.
+ Cho phần tử đầu chuyển đến q.

```
void   insertFirst(SLLNode*   &head,
ElementType x)
{
   SLLNode *q = new SLLNode;
   q->data = x;
   q->next = head;
   head = q;
}
```

Xóa một phần tử



Input: Vị trí p ngay trước phần tử cần xóa

Output: Danh sách sau khi xóa

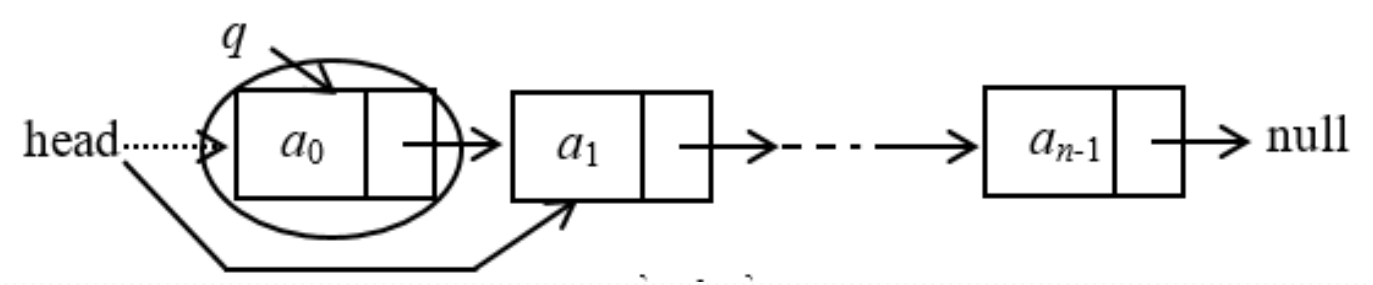
Action:

+ Gọi q là phần tử ngay sau p
+ Cho p liên kết đến phần tử sau q.
+ Thu hồi ô nhớ q.

```
void delete(SLLNode *p)
{
   SLLNode *q;
   q = p->next;
   p->next = q->next;
   delete q;
}
```

Độ phức tạp thuật toán O(1)

Xóa phần tử đầu

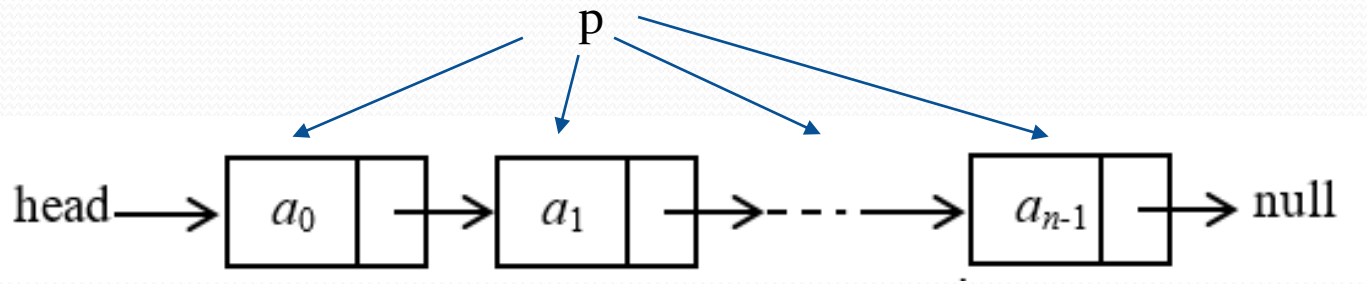


Input: Danh sách cần xóa
Output: Danh sách sau khi xóa
Action
+ Cho q trỏ vào phần tử đầu danh sách
+ Chuyển phần tử đầu danh sách ra phần tử ngay sau nó.
+ Thu hồi ô nhớ q.

```
void deleteFirst(SLLNode* &head)
{
   SLLNode *q;
   q = head;
   head = q->next;
   delete q;
}
```

Độ phức tạp thuật toán O(1)

Duyệt danh sách liên kết đơn



Input: Danh sách cần duyệt
Output: Thứ tự các phần tử được duyệt
Action:
+ Xuất phát p là phần tử đầu danh sách
+ Lặp khi phần tử p còn khác rỗng:
Thực hiện các thao tác trên phần tử p
p chuyển sang phần tử ngay sau nó

```
p = head;
while (p != nullptr)
{
   Thao tác trên p;
   p = p->next;
}
```

Tìm trong danh sách liên kết đơn

- Các phần tử không truy xuất trực tiếp nên tìm tuần tự

```
SLLNode* search(SLLNode *head, KeyType x)
{
   SLLNode *p;
   p = head;
   while (p != nullptr)
   {
      if(p->data.key == x)
      return p;
      else
      p = p->next;
   }
   return nullptr;
}
```

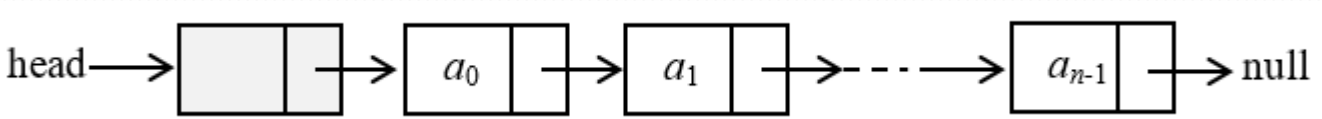
Sắp xếp

Input: Danh sách cần sắp xếp
Output: Danh sách đã sắp xếp
Action:
+ Nếu danh sách rỗng thì dừng
+ Gán biến last rỗng
+ Thực hiện lặp:
    + Gán biến p là phần tử đầu danh sách
    + Lặp khi phần tử tiếp theo của p khác last:
        Gán q là phần tử ngay sau p
        Nếu khóa phần tử tại p lớn hơn khóa phần tử tại q thì
        Đổi dữ liệu hai phần tử p và q
        Gán last là p
        Cho p chuyển ra ngay sau p
+ Cho đến khi last là phần tử ngay sau đầu hoặc last rỗng

```
void BubbleSort(SLLNode *head)
{
 SLLNode *p, *q, *last;
 if (head == nullptr) return;
 last = nullptr;
 do{
 p = head;
 while (p->next != last)
 {
    q = p->next;
    if (p->data.key > q->data.key)
    {
       ElementType tmp = p->data;
       p->data = q->data;
       q->data = tmp;
       last = q;
    }
    p = p->next ;
 }
} while (last==nullptr||last==head->next);
}
```

Danh sách liên kết đơn với header

- Thêm phần tử đặc biệt header vào đầu danh sách liên kết đơn
- Header không chứa dữ liệu, chỉ đánh dấu vị trí xuất phát của danh sách



Thao tác thêm/xóa đầu danh sách đưa về thêm/xóa sau header

Khởi tạo:

```
void initialize(SLLNode* &head)
{
 head = new SLLNode;
 head->next = nullptr;
}
```

Thao tác: bỏ qua header
Ví dụ thao tác duyệt

```
p = head->next;
while (p != nullptr)
{
 Thao tác trên p;
 p = p->next;
}
```

Nhận xét

- Ưu điểm:
  - Khắc phục lãng phí bộ nhớ.
  - Thao tác thêm, xóa thực hiện đơn giản.
- Nhược điểm:
  - Thao tác với các phần tử là tuần tự.
  - Tốn bộ nhớ cho việc lưu liên kết.

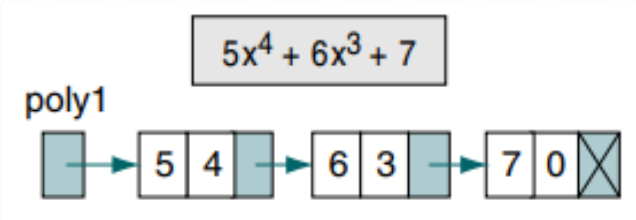
Luyện tập

- Đếm số phần tử của danh sách liên kết đơn
- Xác định phần tử giữa danh sách liên kết đơn

Ứng dụng: Đa thức

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + .. + a_1 x + a_0$$

Tổ chức dữ liệu:
+ Danh sách liên kết đơn mỗi phần tử lưu hệ số (coefficient) và số mũ (exponent).
+ Chỉ lưu các hạng tử với hệ số khác o
+ Lưu theo số mũ giảm




```
struct Node {
    int coefficient;
    int exponent;
    Node* next;
};
```

```
//Thêm vào đầu
void insert(Node** head, int coef, int expo) {
    Node* newNode = new Node;
    newNode->coefficient = coef;
    newNode->exponent = expo;
    newNode->next = (*head);
    (*head) = newNode;
}
```

```
// In đa thức
void print(Node* head) {
  cout << "The polynomial is: ";
  while (head != NULL) {
    if(head->coefficient < 0)
      cout << "-" << abs(head->coefficient);
    else
      cout << "+" << head->coefficient;
    cout << "x^" << head->exponent;
    head = head->next;
    }
   cout << endl;
}
```

Thuật toán tính giá trị đa thức

DL vào: Đa thức p, giá trị x_0

DL ra: giá trị đa thức p tại $x = x_0$

Thao tác:

gt = 0

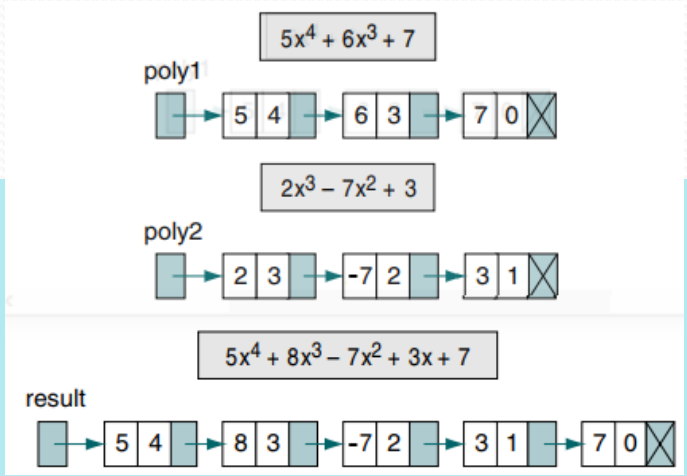
Duyệt đa thức p từ đầu đến cuối

gt = gt + p.coefficient * x_0^p.exponent

Trả về gt

```
// Function to evaluate the polynomial at a given value
double evaluate(Node* head, double x) {
    double result = 0;
    while (head != NULL) {
        result += head->coefficient * pow(x, head->exponent);
        head = head->next;
    }
    return result;
}
```

Cộng hai đa thức:



DL vào: 2 đa thức p, q

DL ra: đa thức p + q

Thao tác:

Tạo đa thức kq rỗng

Duyệt từng phần tử của đa thức p và q khi còn khác rỗng

Nếu phần tử của đa thức p và q cùng số mũ thì

Nếu tổng hệ số của hai phần tử tại p và q khác 0 thì

Thêm vào cuối đa thức kq hạng tử có hệ số là tổng hệ số của hai hạng tử của p và q, số mũ là số mũ của hạng tử đa thức p hoặc q

Chuyển ra phần tử tiếp theo của đa thức p và q

Ngược lại nếu số mũ của đa thức nào lớn hơn thì đưa hệ số và số mũ của đa thức đó vào đầu đa thức kq rồi chuyển sang phần tử tiếp theo

Duyệt từng phần tử còn lại của đa thức p và q thêm vào cuối đa thức kq

Trả về kq

```
Node* add(Node* poly1, Node* poly2) {
  Node *result = NULL, *last = NULL, *temp;
  while (poly1 && poly2) {
   if (poly1->exponent > poly2->exponent) {
     last = insertNext(last, poly1->coefficient,poly1->exponent);
     poly1 = poly1->next;
   }
   else if (poly2->exponent > poly1->exponent) {
     last=insertNext(last,poly2->coefficient,poly2->exponent);
     poly2 = poly2->next;
   }
     else {
       int coef = poly1->coefficient + poly2->coefficient;
       if (coef != 0)
         last = insertNext(last, coef, poly1->exponent);
       poly1 = poly1->next;
       poly2 = poly2->next;
      }
   if (!result)
    result = last;
 }
```

```
while (poly1) {
        last = insertNext(last, poly1->coefficient, poly1->exponent);
        poly1 = poly1->next;
    }

 while (poly2) {
        last = insertNext(last, poly2->coefficient, poly2->exponent);
        poly2 = poly2->next;
    }

    return result;
}
```

```
Node* insertNext(Node* node, int coef, int expo) {
    Node* newNode = new Node;
    newNode->coefficient = coef;
    newNode->exponent = expo;
    newNode->next = NULL;
    if (node) node->next = newNode;
    return newNode;
}
```

```
int main() {
    // Creating the first polynomial
    Node* poly1 = NULL;
    insert(&poly1, 7, 0);
    insert(&poly1, 6, 3);
    insert(&poly1, 5, 4);

    // Creating the second polynomial
    Node* poly2 = NULL;
    insert(&poly2, 3, 1);
    insert(&poly2, -7, 2);
    insert(&poly2, 2, 3);

    // Print two polynomials
    print(poly1);
    print(poly2);
```

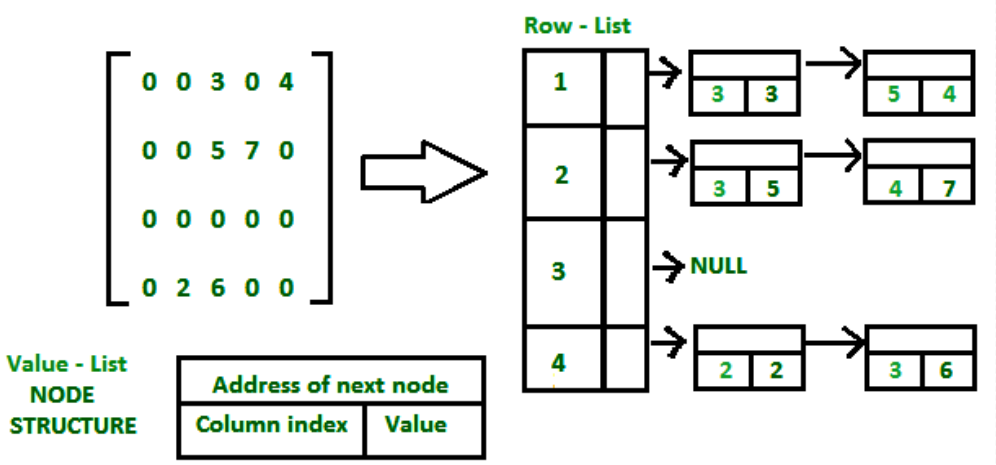
```
// Evaluating the first polynomial at x = 2
    double x = 2;
    cout << evaluate(poly1, x) << endl;

    // Adding the two polynomials
    Node* result = add(poly1, poly2);

    // Displaying the result
    print(result);
    return 0;
}
```

Ma trận thưa

- Tổ chức dữ liệu cho ma trận thưa m dòng, n cột:
  - Dùng mảng rowList gồm m phần tử
  - Mỗi phần tử là một danh sách liên kết đơn lưu các số khác 0 gồm chỉ số cột và giá trị

```
struct Node{
  int colIndex;
  float value;
  Node* next;
};
```

```
struct SparseMatrix{
  int numRow, numCol;
  Node* rowList[MAX];
};
```



Bài tập

Bài 3.1 Trình bày thuật toán và cài đặt các thao tác sau trên danh sách sinh viên tổ chức dưới dạng danh sách liên kết đơn.

a) Đếm số sinh viên của danh sách sinh viên.

b) Thêm một sinh viên vào cuối danh sách

c) Tạo danh sách mới chứa những sinh viên có đtb>=8.

d) Xóa 1 sinh viên có họ tên x

e) Thêm sinh viên vào vị trí k, với k là số nguyên, phần tử đầu tiên có thứ tự là 0.

f) Xóa sinh viên tại vị trí k, với k là số nguyên.

g) Chèn một danh sách sinh viên l2 vào danh sách sinh viên l1 ngay sau sinh viên có tên t.

Bài 3.2 Cho một danh sách liên kết đơn, p là một phần tử trong danh sách. Hãy viết hàm đổi hai phần tử kề nhau ngay sau phần tử p, chỉ được đổi liên kết mà không được đổi dữ liệu.

Bài 3.3 Cho một danh sách liên kết đơn L và một mảng P chứa các số nguyên được sắp theo thứ tự tăng. Viết hàm printLots(L, P) in những phần tử của L ở các vị trí được xác định trong mảng P. Ví dụ: nếu P = 1, 3, 4, 6 thì in các phần tử của L ở các vị trí 1, 3, 4, 6. Giả sử vị trí phần tử đầu tiên là 0. Cho biết độ phức tạp thời gian của thuật toán cài đặt thao tác trên.

Bài 3.4 Cho p là một con trỏ đến một nút trong danh sách liên kết đơn, nút p trỏ đến không phải là nút cuối cùng trong danh sách. Giả sử không có con trỏ đến một nút nào khác (trừ các liên kết giữa các nút). Trình bày thuật toán có độ phức tạp O(1) xóa giá trị lưu tại nút p khỏi danh sách, các phần tử còn lại vẫn giữ nguyên thứ tự.

Bài 3.5 Cài đặt các thuật toán sau trên danh sách liên kết đơn các số nguyên:

a) Đảo ngược danh sách

b) Thuật toán Quick sort.

Danh sách liên kết vòng (Circular Linked List)

Giới thiệu

- Là danh sách liên kết đơn, phần tử cuối liên kết đến phần tử đầu.
- Thường dùng trong danh sách không quan tâm đến thứ tự
- Các thao tác:
  - Duyệt danh sách liên kết vòng
  - Thêm một phần tử vào đầu danh sách liên kết vòng
  - Thêm một phần tử vào cuối danh sách liên kết vòng
  - Xóa phần tử đầu danh sách liên kết vòng
  - Xóa phần tử cuối danh sách liên kết vòng.

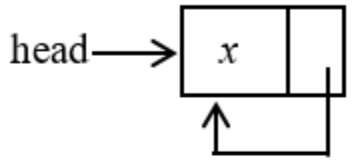
Duyệt

```
if (head != nullptr)
{
   p = head;
   do
   {
      visit(p);
      p = p->next;
   }
   while (p == head);
}
```

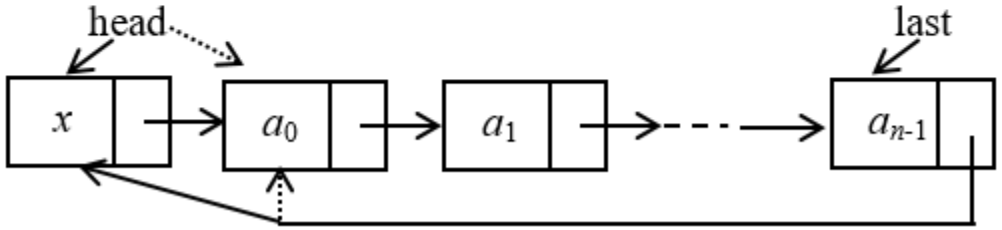
Thêm vào đầu

Input: Danh sách cần thêm, phần tử cần thêm
Output: Danh sách sau khi thêm
Action:
Cấp phát một ô nhớ q kiểu danh sách liên kết
Đưa phần tử cần thêm vào dữ liệu của q
Nếu danh sách rỗng thì:
    Cho q liên kết đến q
    Gán phần tử q cho đầu danh sách
Ngược lại:
    Xuất phát last từ phần tử đầu
    Lặp khi phần tử sau last khác đầu
        Chuyển last sang phần tử ngay sau nó
Gán last liên kết đến q
Cho q liên kết đến phần tử đầu
Gán q cho phần tử đầu

Thêm vào danh sách rỗng

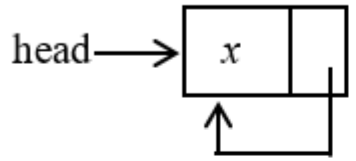


Thêm vào danh sách khác rỗng

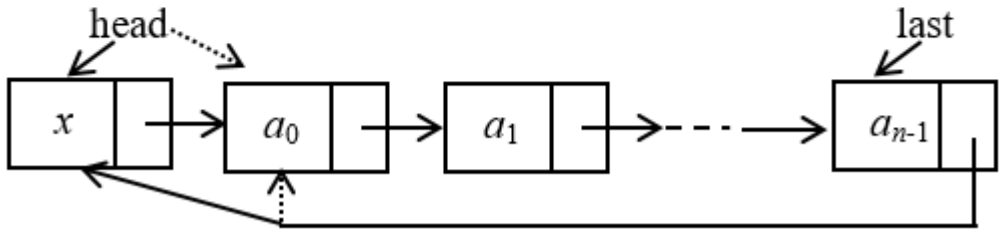


```
void insertFirst(SLLNode* &head, ElementType x)
{
   SLLNode *q, *last;
   q = new SLLNode;
   q->data = x;
   if (head == nullptr)
   {
      q->next = q;
      head = q;
   }
   else
   {
      last = head;
      while (last->next != head) last = last->next;
      last->next = q;
      q->next = head;
      head = q;
   }
}
```

Thêm vào danh sách rỗng



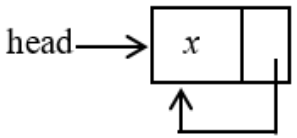
Thêm vào danh sách khác rỗng



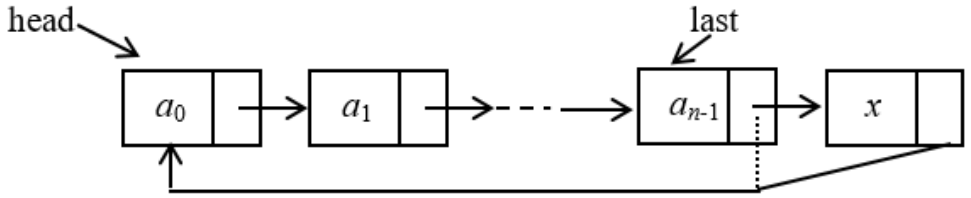
Độ phức tạp O(n)

Thêm vào cuối

Thêm vào danh sách rỗng



Thêm vào danh sách khác rỗng



Độ phức tạp O(n)

```
void insertLast(SLLNode* &head, ElementType x)
{
   SLLNode *q, *last;
   q = new SLLNode;
   q->data = x;
   if (head == nullptr)
   { head = q; q->next = q; }
   else
   {
      last = head;
      while (last->next != head)
         last = last->next;
      last->next = q;
      q->next = head;
   }
}
```

Xóa đầu

Input: Danh sách cần xóa
Output: Danh sách sau khi xóa
Action:
Nếu danh sách rỗng thì dừng
Nếu danh sách có 1 phần tử thì:
- Thu hồi ô nhớ đầu
- Gán phần tử đầu là null

Ngược lại :
- Tìm phần tử last cuối danh sách
- Cho last liên kết sau phần tử đầu
- Thu hồi ô nhớ đầu
- Gán phần tử sau last cho phần tử đầu

Độ phức tạp O(n)

```
void deleteFirst(SLLNode* &head)
{
   if (head == nullptr) return;
   if (head->next == head)
   {
      delete head;
      head = nullptr;
   }
   else
   {
      last = head;
      while (last->next != head) last = last->next;
      last->next = head->next;
      delete head;
      head = last->next;
   }
}
```

Xóa cuối

Input: Danh sách cần xóa
Output: Danh sách sau khi xóa
Action:
Nếu danh sách rỗng thì dừng
Nếu danh sách có 1 phần tử thì:
    Thu hồi ô nhớ đầu
    Gán phần tử đầu là null
Ngược lại :
    Tìm phần tử plast ngay trước phần tử cuối danh sách
    Gán phần tử ngay sau plast cho biến q
    Cho plast liên kết đến phần tử đầu
    Thu hồi ô nhớ q

Độ phức tạp O(n)

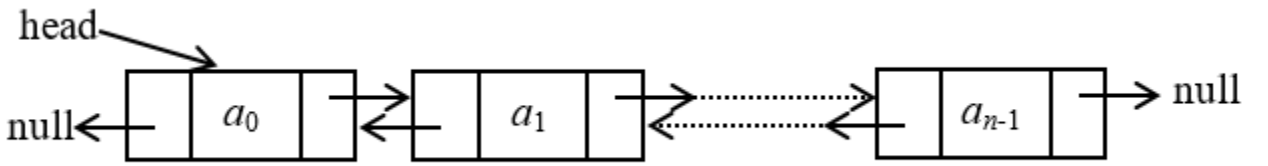
```
void deleteLast(SLLNode* &head)
{
   if (head == nullptr) return;
   if (head->next == head)
   {
      delete head;
      head = nullptr;
   }
   else
   {
      SLLNode *plast = head;
      while (plast->next->next != head)
        plast = plast->next;
      SLLNode *q = plast->next;
      plast->next = head;
      delete q;
   }
}
```

Danh sách hai liên kết (Doubly Linked List)

Giới thiệu

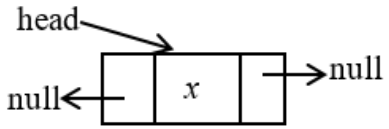
- Mỗi phần tử gồm hai liên kết đến phần tử trước và sau nó.
- Thuận lợi cho các thao tác
- Tổ chức dữ liệu:

```
struct DLLNode
{
   ElementType data;
   DLLNode *prev, *next;
};
```

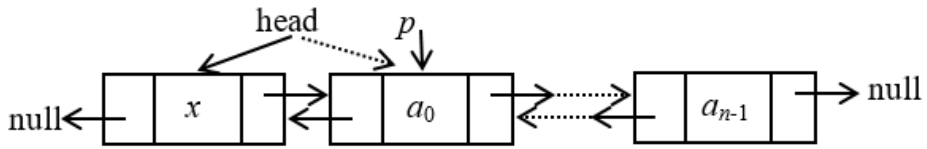


Thêm trước

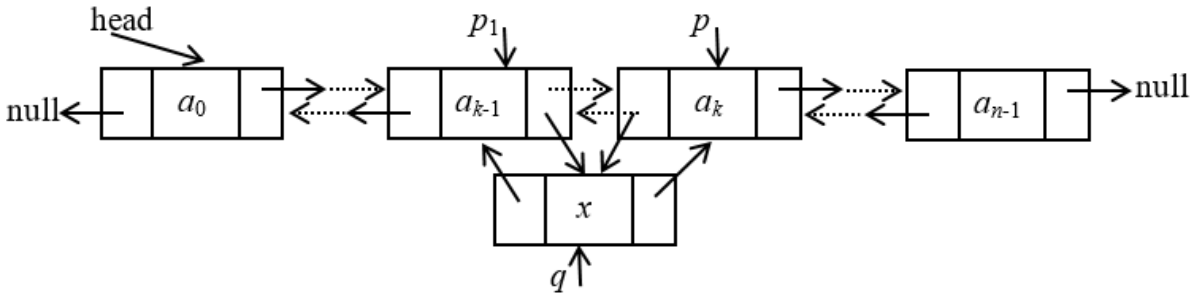
Thêm vào danh sách rỗng



Thêm vào đầu danh sách

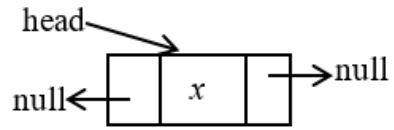


Thêm vào trước phần tử p

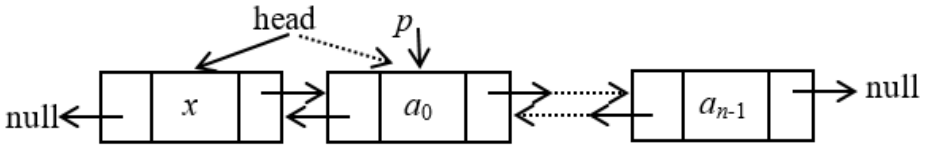


Input: Danh sách cần thêm, phần tử cần thêm x, vị trí thêm p
Output: Danh sách sau khi thêm
Action:
Tạo phần tử q kiểu danh sách hai liên kết
Đưa x vào dữ liệu của q
Nếu danh sách rỗng thì:
    Cho q liên kết trước, liên kết sau rỗng
    Gán q cho phần tử đầu
Ngược lại:
    Nếu p là phần tử đầu thì:
        Cho q liên kết trước đến rỗng
        Cho q liên kết sau đến p
        Cho p liên kết trước đến q
        Gán q cho phần tử đầu
    Ngược lại:
        Gọi p1 là phần tử ngay trước p
        Cho q liên kết trước đến p1
        Cho p1 liên kết sau đến q
        Cho q liên kết sau đến p
        Cho p liên kết trước đến q

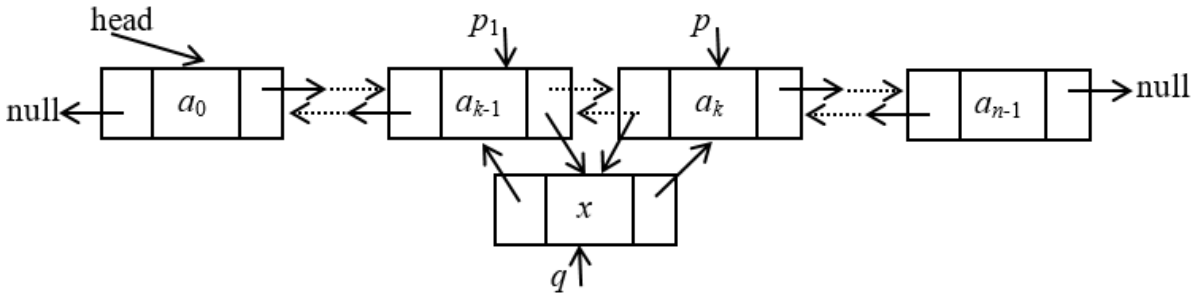
Thêm vào danh sách rỗng



Thêm vào đầu danh sách



Thêm vào trước phần tử p

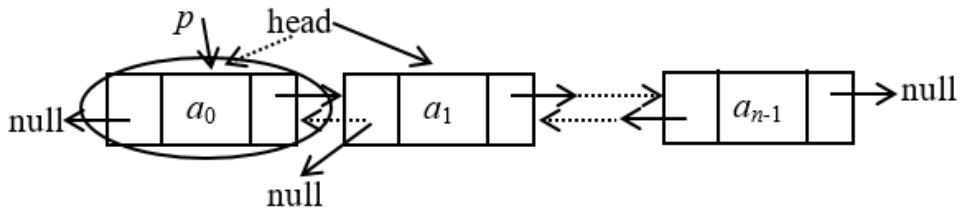


Độ phức tạp O(1)

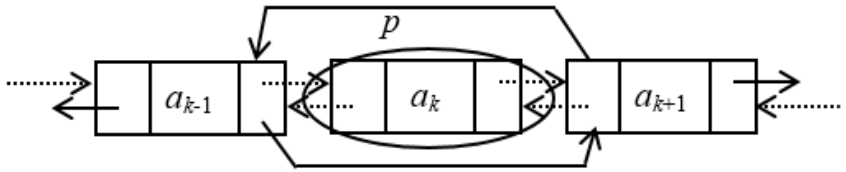
```
void insertBefore(DLLNode* &head, DLLNode *p,
ElementType x)
{
   DLLNode *q,*p1;
   q = new DLLNode;
   q->data = x;
   if (head == nullptr){
      q->prev = nullptr; q->next = nullptr;
      head = q;}
   else{
      if (head == p){
      q->prev = nullptr; q->next = p;
      p->prev = q; head = q;}
      else {
      p1 = p->prev; q->prev = p1;
      p1->next = q; q->next = p;
      p->prev = q;}
   }
}
```

Xóa tại vị trí p

Xóa phần tử đầu

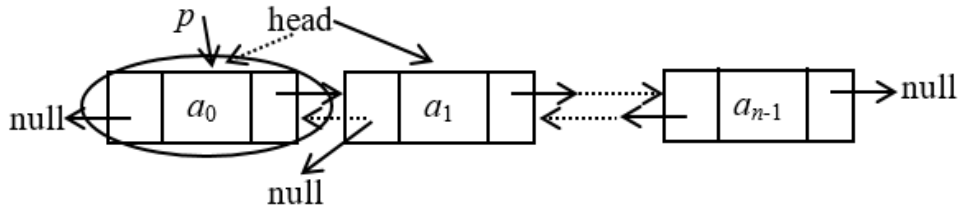


Xóa phần tử p

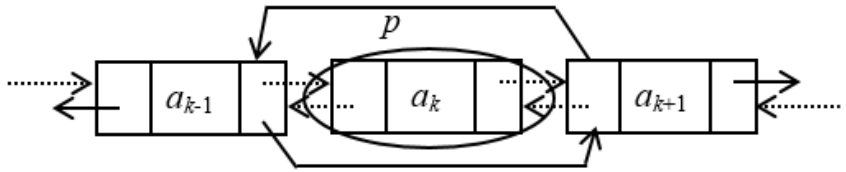


Input: Danh sách cần xóa, vị trí cần xóa p
Output: Danh sách sau khi xóa
Action:
Nếu p là phần tử đầu:
    Cho phần tử đầu chuyển ra phần tử ngay sau
    Nếu phần tử đầu khác rỗng thì :
        Gán liên kết trước của phần tử đầu rỗng
Ngược lại :
    Gán liên kết sau của phần tử trước p là phần tử sau p
    Nếu phần tử sau p khác rỗng thì :
        Gán liên kết trước của phần tử sau p là phần tử trước p
Thu hồi ô nhớ p

Xóa phần tử đầu



Xóa phần tử p



Độ phức tạp O(1)

```
void delete(DLLNode* &head, DLLLNode *p)
{
   if (head == p) // xóa phần tử đầu
   {
      head = p->next;
      if (head != nullptr)
      head->prev = nullptr;
   }
   else
   {
      p->prev->next = p->next;
      if (p->next != nullptr)
      p->next->prev = p->prev;
   }
   delete p;
}
```

Lớp list

Giới thiệu

- Trong C++ lớp list dùng để thao tác với danh sách
- Sử dụng cấu trúc danh sách hai liên kết
- Khai báo: std::list<DataType> variable;
- Ví dụ:
  - std::list<int> listOfInt;
  - std::list<Student> listOfStudents;
- Khởi tạo:
  - `std::list<int> listOfInt = {1, 5, 2, 4, 8, 3};`

Các phướng thức

Phương thức	Chức năng	Độ phức tạp
size()	Số phần tử trong list	O(1)
empty()	Cho biết list có rỗng không	O(1)
clear()	Xóa các phần tử trong list	O(1)
push_back(e)	Thêm một phần tử e vào cuối list	O(1)
push_front(e)	Thêm một phần tử e vào đầu list	O(1)
insert(pos, e)	Thêm một phần tử e vào vị trí pos (iterator)	O(1)
pop_back()	Xóa phần tử cuối cùng của list	O(1)
pop_front()	Xóa phần tử đầu tiên của list	O(1)
erase(pos)	Xóa phần tử tại vị trí pos trong list	O(1)
begin()	Trả về iterator đến phần tử đầu tiên trong list	O(1)
end()	Trả về iterator đến phần tử sau phần tử cuối cùng trong list	O(1)
sort()	Sắp xếp các phần tử theo thứ tự tăng	O(nlogn)

iterator

- Là đối tượng dùng để duyệt các phần tử trong collection.

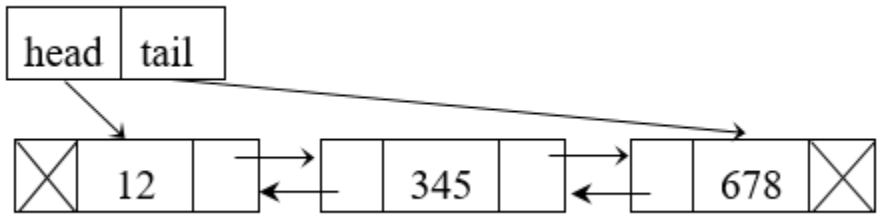
```
#include <list>
#include <iostream>
int main()
{
    std::list<int> myList {1, 2, 3};
    std::cout << "Forward iteration:\n";
   std::list<int>::iterator it;
    for (it = myList.begin(); it != myList.end(); it++)
        std::cout << *it << ' ';
    std::cout << '\n';

    std::cout << "Reverse iteration:\n";
     std::list<int>::reverse_iterator rit;
    for (rit = myList.rbegin(); rit != myList.rend(); rit++)
        std::cout << *rit << ' ';
    std::cout << '\n';
    return 0;
}
```

```cpp
#include <iostream>
#include <list>
#include <algorithm>

int main() {
    std::list<int> myList = {5, 1, 4, 2, 3};
    std::cout << "Before sorting: ";
    for (int element : myList) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

    myList.sort();

    std::cout << "After sorting: ";
    for (int element : myList) {
        std::cout << element << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

```cpp
bool compareDescending(int a, int b) {
    return a > b;
}
int main() {
    std::list<int> myList = {5, 1, 4, 2, 3};
    std::cout << "Before sorting: ";
    for (int element : myList) {
        std::cout << element << " ";
    }
    std::cout << std::endl;
    myList.sort(compareDescending);
    std::cout << "After sorting: ";
    for (int element : myList) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Ứng dụng: Số nguyên lớn

- Để thao tác với các số nguyên lớn, có thể dùng danh sách hai liên kết trong đó dữ liệu tại mỗi nút là một số nguyên không quá 3 chữ số.
- Ví dụ: số nguyên n = 12 345 678 được lưu trong danh sách hai liên kết như sau:




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

```
struct BigInteger {
    Node* head;
    Node* tail;
    int size;
};
```

Chuyển số nguyên thành BigInteger

```
BigInteger intToBigInteger(int n) {
    BigInteger result;
    result.head = result.tail = NULL;
    result.size = 0;
 while (n > 0)
 {
    int num = n % 1000;
    n = n / 1000;

   Node* newNode = new Node();
    newNode->data = num;
    newNode->next = NULL;
    newNode->prev = NULL;
```

```
if (result.head == NULL) {
       result.head = result.tail = newNode;
    } else {
            result.head->prev = newNode;
            newNode->next = result.head;
            result.head = newNode;
        }
        result.size++;
    }
    return result;
}
```

- In BigInteger:
  - In các số trong danh sách 2 liên kết từ đầu đến cuối.
  - Với các số khác số đầu tiên nếu chưa đủ 3 chữ số thì bổ sung các số 0 ở trước cho đủ 3 chữ số.

```
void printBigInteger(BigInteger b) {
    Node* current = b.head;
    while (current != NULL) {
        // print leading zeros
        if (current != b.head) {
            if (current->data < 100) cout << "0";
            if (current->data < 10) cout << "0";
        }
        cout << current->data;
        current = current->next;
    }
    cout << endl;
}
```

- So sánh BigInteger
  - So sánh chiều dài
  - So sánh lần lượt các số

```
int compareBigIntegers(BigInteger a, BigInteger b) {
    // check the size of the numbers first
    if (a.size > b.size) return 1;
    if (a.size < b.size) return -1;

    Node* currentA = a.head;
    Node* currentB = b.head;

    // compare each node's data
    while (currentA != NULL) {
        if (currentA->data > currentB->data) return 1;
        if (currentA->data < currentB->data) return -1;
        currentA = currentA->next;
        currentB = currentB->next;
    }
    return 0;
}
```

- Cộng BigInteger:
  - Cộng các số tương ứng ở các nút
  - Thêm vào đầu danh sách kết quả

```
BigInteger addBigIntegers(BigInteger a, BigInteger b) {
    BigInteger result;
    result.head = result.tail = NULL;
    result.size = 0;
    Node* currentA = a.tail;
    Node* currentB = b.tail;
    int carry = 0;
    while (currentA != NULL || currentB != NULL) {
        int sum = carry;
        if (currentA != NULL) {
            sum += currentA->data;
            currentA = currentA->prev;}
        if (currentB != NULL) {
            sum += currentB->data;
            currentB = currentB->prev;}
```

```
carry = sum / 1000;
sum = sum % 1000;
Node* newNode = new Node();
newNode->data = sum;
newNode->next = NULL;
newNode->prev = NULL;

if (result.head == NULL) {
  result.head = result.tail = newNode;
} else {
    result.head->prev = newNode;
    newNode->next = result.head;
    result.head = newNode;
  }
 result.size++;
}
```

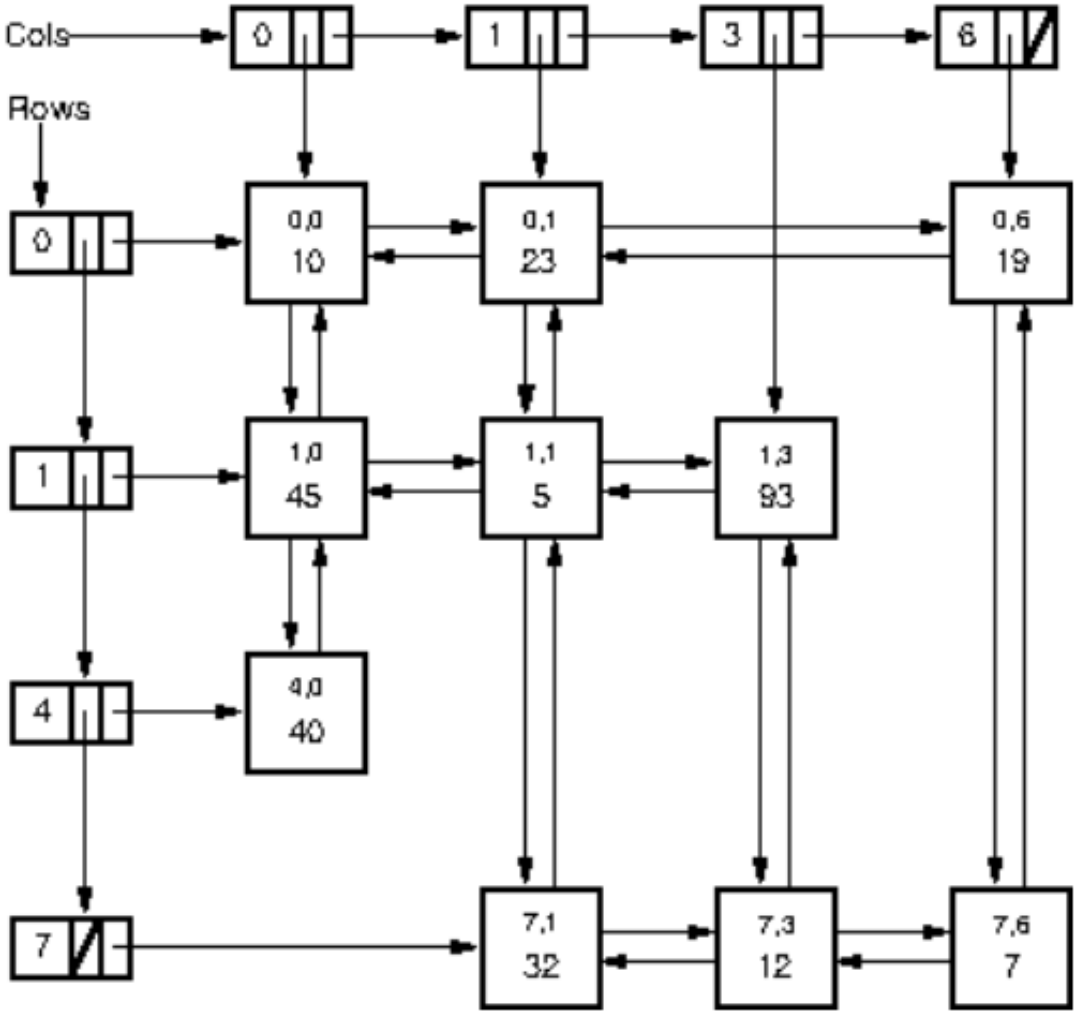
```
if (carry > 0) {
        Node* newNode = new Node();
        newNode->data = carry;
        newNode->next = NULL;
        newNode->prev = NULL;
        result.head->prev = newNode;
        newNode->next = result.head;
        result.head = newNode;
        result.size++;
    }

    return result;
}
```

Bài tập

Bài 3.5 Ma trận thưa

- Một cách tổ chức dữ liệu cho ma trận thưa là dùng danh sách trực giao (orthogonal list). Trong cách tổ chức này, mỗi nút trong ma trận thưa gồm các trường:
- row: chỉ số dòng
- col: chỉ số cột
- value: giá trị của ma trận tại (row, col)
- down: con trỏ đến dòng tiếp theo cùng cột với nút
- right: con trỏ đến cột tiếp theo cùng dòng với nút
- Để quản lý, các cột sử dụng một danh sách liên kết lưu chỉ số và con trỏ đến nút đầu tiên của cột chứa giá trị khác 0. Tương tự cho quản lý các dòng.
- Mỗi ma trận thưa sẽ lưu số dòng, số cột và hai con trỏ đến hai danh sách cột và dòng.

10	23	0	0	0	0	19
45	5	0	93	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
40	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	32	0	12	0	0	7



- Hãy khai báo tổ chức dữ liệu theo mô tả trên và cài đặt các thao tác:

a) Thêm một giá trị v ở dòng r, cột c của ma trận thưa.

b) Lấy giá trị của ma trận thưa tại dòng r, cột c.

c) Tạo một ma trận thưa từ mảng hai chiều m dòng, n cột.

d) Chuyển một ma trận thưa thành mảng hai chiều.

e) In một ma trận thưa dưới dạng danh sách các giá trị khác 0 của ma trận: row, col, value.

f) Tìm vị trí phần tử lớn nhất của ma trận thưa.

g) Tìm ma trận chuyển vị của một ma trận thưa.

h) Cộng hai ma trận thưa cùng kích thước.

i) Nhân hai ma trận thưa có kích thước phù hợp.