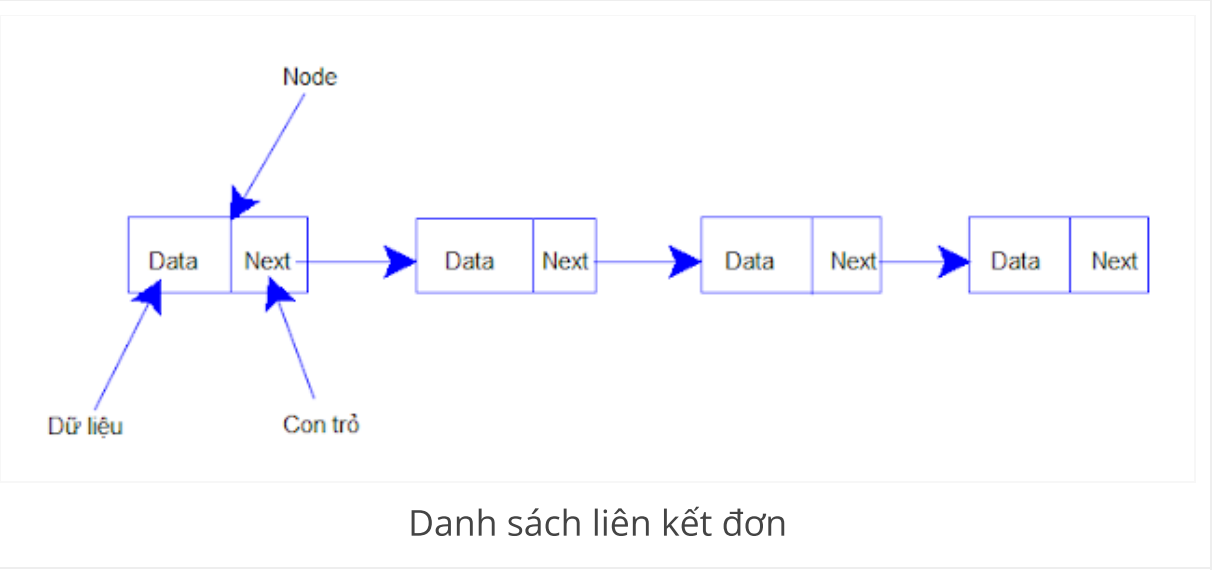
Danh Sách Liên Kết Đơn - Link List

Cường Công 07:30 Cấu trúc dữ liệu và thuật toán

CÁC PHÉP TOÁN TRÊN DANH SÁCH LIÊN KẾT ĐƠN

A, Mô tả, tính chất của danh sách liên kết đơn



Danh sách liên kết đơn

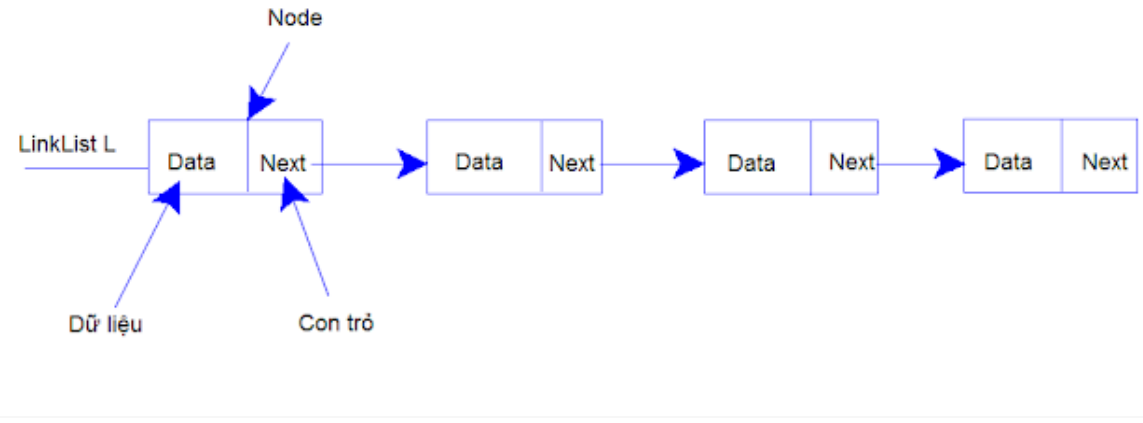
- Bên trên chính là hình ảnh mô tả một danh sách liên kết đơn. Trong hình, mỗi một ô hình chữ nhật được gọi là một "**Node**". Trong một "**Node**" chứa 2 thuộc tính:
+, Data (dữ liệu): có thể là số nguyên, số thực hay là sinh viên, nhân viên...
+, Next: là một con trỏ để lưu địa chỉ của "Node" tiếp theo. Nói đơn giản là nó trỏ đến phần tử kế tiếp như trong hình. Cũng có thể hiểu nó như một sợi dây buộc vào Node tiếp theo
- Tính chất **danh sách liên kết đơn**:
+, Là một danh sách sử dụng con trỏ và động.
+, **Danh sách liên kết** có thể thêm phần tử vào thoải mái đến khi nào hết bộ nhớ máy tính thì thôi.
+, Đối với mảng các phần tử trong mảng phải liên tiếp nhau, còn với Danh Sách Liên Kết thì khác, các "Node" có thể nằm ở bất kì đâu trong bộ nhớ. Chỉ cần chúng ta liên kết chúng lại với nhau thì sẽ tạo thành một danh sách.

+, Danh Sách Liên Kết đơn chỉ được đi tiến chứ không được đi lùi.
Ví dụ: Ta cần truy xuất phần tử thứ 2 trong Danh Sách Liên Kết Đơn, ta phải duyệt từ đầu đến phần tử thứ 2 chứ không thể duyệt ngược lại từ cuối.

B, Cài đặt và các phép toán trên Danh Sách Liên Kết Đơn

1, Dạng cài đặt

+, Tạo một cấu trúc struct Node gồm 2 thuộc tính: Data (dữ liệu) và con trỏ Next kiểu dữ liệu là Node luôn, vì con trỏ Next trỏ đến một Node khác.
Ví dụ: bạn khai báo một con trỏ P trỏ đến một bến kiểu nguyên (int) thì là: int P. Thì tương tự như vậy Next trỏ đến "Node" thì phải là kiểu Node.
+, Khai báo thêm một kiểu cấu trúc struct LinkList để tượng trưng cho cả một danh sách (gồm nhiều Node).



Danh sách liên kết đơn

Khai báo một danh sách L. L luôn phải cầm đầu(ở Node đầu tiên), nếu L mất hoặc di chuyển chúng ta sẽ mất danh sách.
Ở đây mình demo bằng danh sách số nguyên

```
typedef struct Node {
 int Data;
 Node* Next;
};

typedef struct Node* LinkList;
```

2, Khởi tạo (InIt)

+, Để khởi tạo một danh sách rỗng ta chỉ cần gán L = NULL
+, Chú ý: L ở đây phải tham trị vì ta cần lưu lại giá trị của L khi thay đổi.

```
void InIt(LinkList& L) { // tham trị
 L = NULL;
}
```

3, Kiểm tra rỗng (IsEmpty)

Danh sách rỗng khi L = NULL

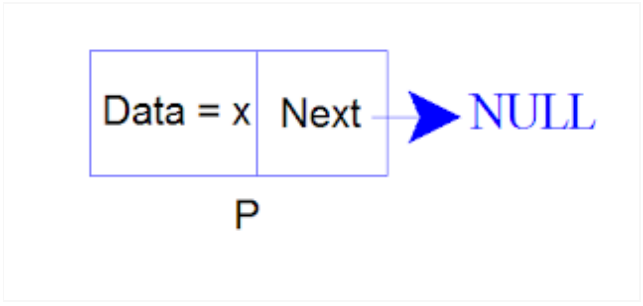
```
bool IsEmpty(LinkList L) {
 if (L == NULL) return true;
 return false;
}
```

hoặc

```
bool IsEmpty(LinkList L) {
 return (L == NULL)
}
```

4, Tạo một Node (AddNode)

Viết hàm tạo ra một node để áp dụng vào phép toán thêm phần tử vào danh sách



Tạo Node

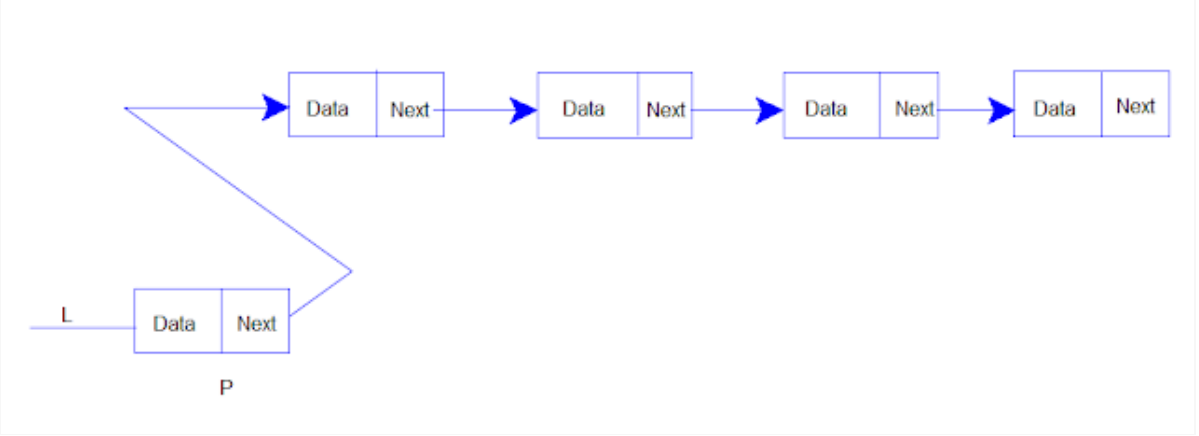
```
Node* AddNode(int x) { // tạo môt Node có giá trị x
 Node* P; // khai báo con trỏ P kiểu Node
 P = (Node*)malloc(sizeof(Node));  // cấp phát
 P->Data = x;  // gán Data của Node P (hay con trỏ P) là x
 P->Next = NULL; // tạo ra một Node nó chưa liên kết với ai cả lên Next
của P là NULL
 return P;  // trả về
}
```

5, Thêm Node vào Danh sách

+, Thêm vào Đầu danh sách(AddHead)

Vì L chính là Node đầu tiên vì thế ta chỉ cần cập nhật lại L là xong

Cho Next của P (như là sợi dây) trỏ hay buộc vào L. Lúc này P chính là Node đầu tiên thì cập nhật lại L chính là P;



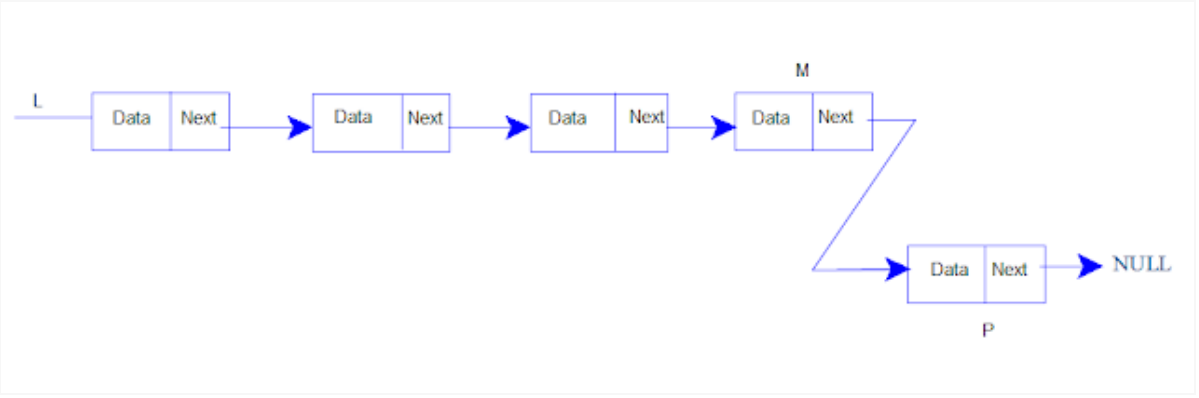
Thêm Node vào đầu sanh sách

```
void AddHead(LinkList& L, int x) {
 Node* P;  // khai báo con trỏ P
 P = AddNode(x);  // gọi hàm tạo Node P với giá trị là x
 if (IsEmpty(L) == true) L = P;  // nếu danh sách rỗng thì P chính là L
vì chỉ có 1 Node
 else {
  P->Next = L;  // đem Next của Node P vừa tạo trỏ đến L hiện tại
  L = P; // cập nhật L
 }
}
```

+, Thêm vào Cuối danh sách (AddTail)

Duyệt từ Node đầu tiên (tức L) đến Node cuối cùng trong danh sach hiện tại, để xác định được Node cuối cùng. Khi xác định được thì chỉ cần dùng Next của Node cuối đó trỏ vào Node mới

Tạo con trỏ M bằng Node đầu tiên là L rồi cho M tăng đến Node cuối cùng thì dừng. Node cuối đó chính là M.



Thêm Node vào cuối danh sách

```
void AddTail(LinkList& L, int x) {
 Node* P;
 P = AddNode(x);  // tạo node P với giá trị x
```

```
 if (IsEmpty(L) == true) L = P;  // nếu danh sách rỗng thì P chính là L luôn
 else {
  Node* M;  // khai báo node M
  M = L; // gán M bằng L đầu danh sách
  while (M->Next != NULL) {  // M tăng dần khi M->Next = NULL thì dừng
   M = M->Next;
  }
  M->Next = P; // xác định được node cuối rồi thì nối vào P
  P->Next = NULL; // P là node cuối cùng nên Next trỏ đến NULL
 }
}
```

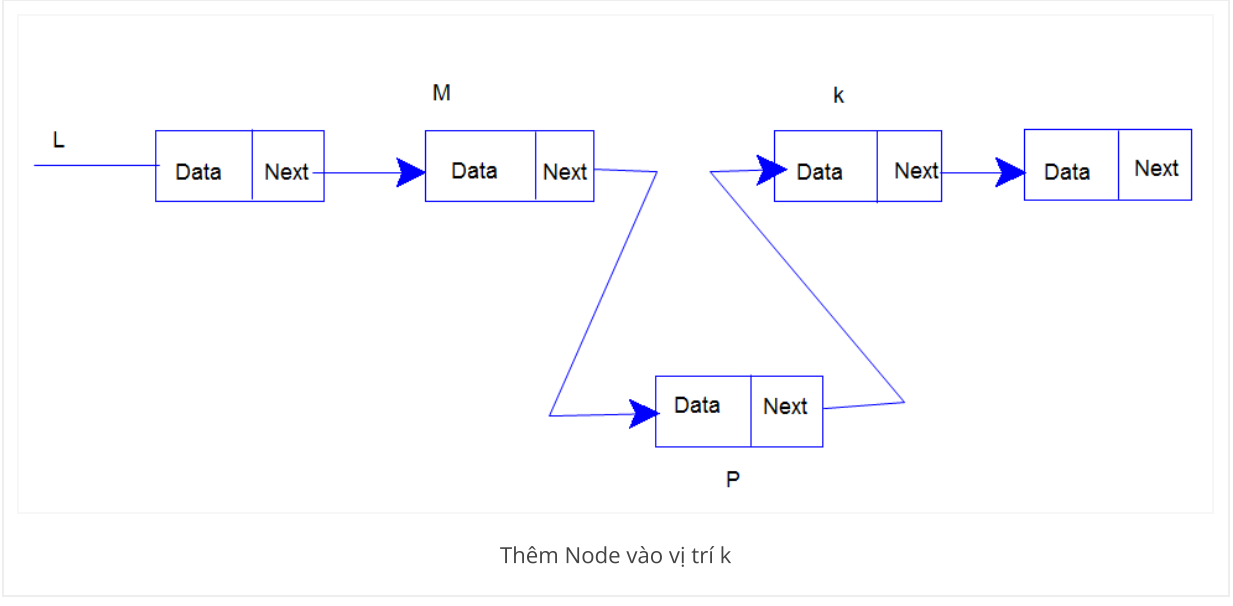
+, Đếm số lượng Node hay phần tử trong danh sách (Quantity)

Tạo node M cho nó chạy từ đầu danh sách, mỗi lần tăng lên thì số lượng tăng thêm 1 đơn vị

```
int Quantity(LinkList L) {
 int temp = 0;  // khởi tạo ban đầu bằng 0
 for (Node* M = L; M != NULL; M = M->Next) { // M tăng lên bằng NULL thì dừng
  temp++; // mỗi lần lặp temp tăng lên
 }
 return temp; // trả về temp
}
```

+, Thêm vào vị trí k bất kỳ(AddAny)

Kiểm tra vị trí k có thỏa mãn hay không? Nếu thỏa mãn thì xác định Node M trước vị trí k. Sau đó cho Next của P trỏ vào Node mà Next của M đang trỏ (P->Next = M->Next). Nối M với P (M->Next = P)



Thêm Node vào vị trí k

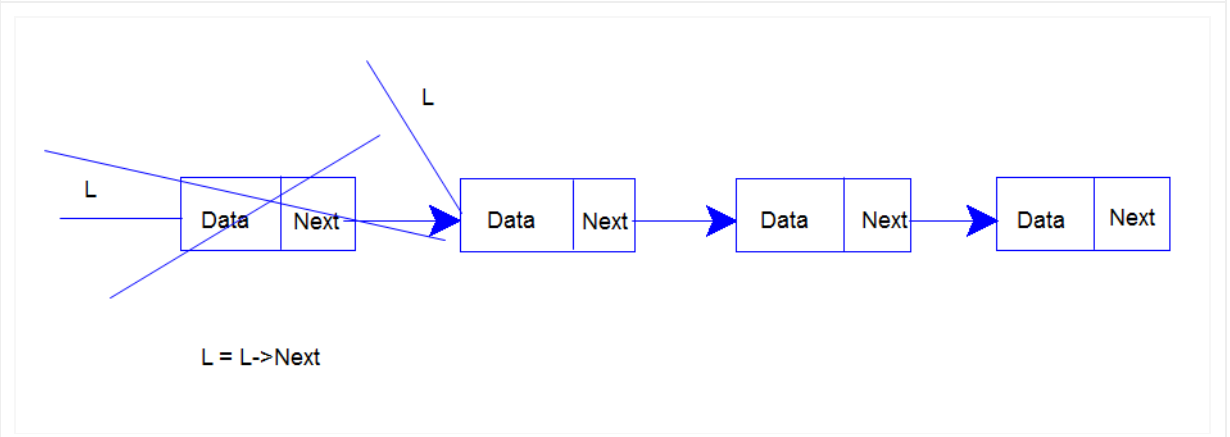
```
void AddAny(LinkList& L, int k, int x) {
 if (k < 1 || k > Quantity(L)) cout << "Not Add. " << endl; // kiểm tra vị trí k có thỏa mãn hay không?
 else{
  if (k == 1) AddHead(L, x); // nếu k = 1 thì chính là thêm vào phần tử đầu tiên
```

```
   else {
    Node* P;
    P = AddNode(x); // tạo node P với giá trị x
    Node* M = L; // tạo node M bằng L
    int temp = 1; // biến temp để biết điều kiện dừng
    while (temp != k - 1 && M != NULL) { //
     temp++;
     M = M->Next; // M tăng dần lên
    }
    P->Next = M->Next; // P->Next bằng chính Node mà M->Next đến
    M->Next = P;  // M->Next là P;
   }
  }
}<span style="font-size: large;">
</span>
```

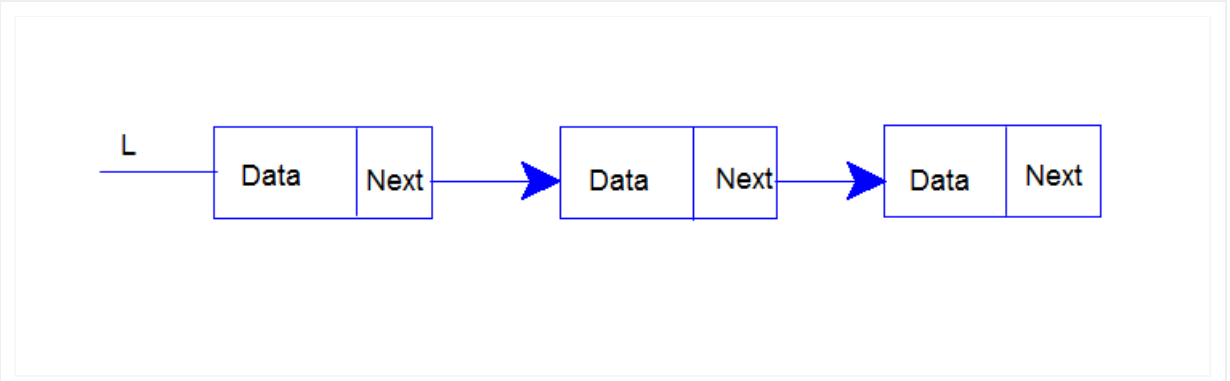
6, Xóa Node khỏi danh sách

+, Xóa đầu (DeleteHead)

Cho L chạy tăng lên Node tiếp theo thì Node đầu sẽ mất



Xóa đầu

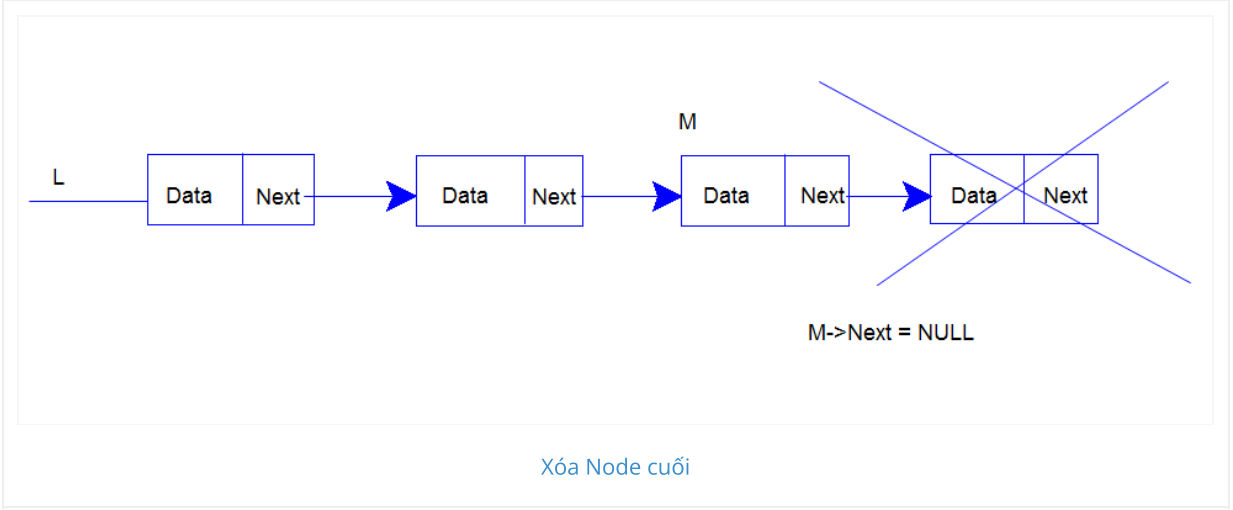


Sau khi xóa

```
void DeleteHead(LinkList& L) {
 L = L->Next; // Tăng L
}
```

+, Xóa cuối (DeleteTail)

Xác định Node sau Node cuối rồi cho Node đó trỏ đến NULL

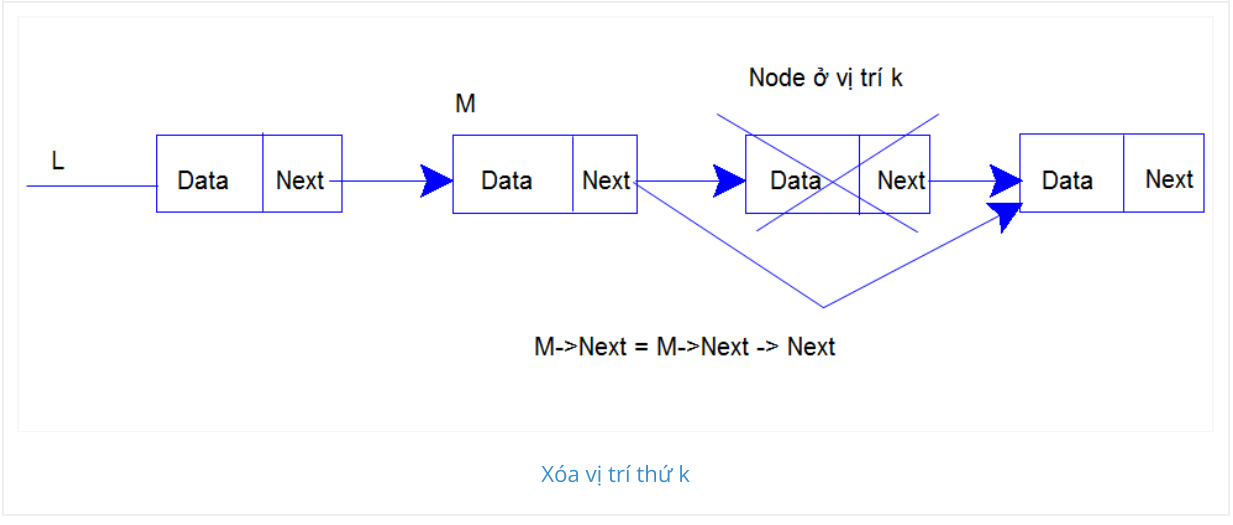


Xóa Node cuối

```
void DeleteTail(LinkList &L){
Node* M = L;
 while(M->Next->Next != NULL) {
  M = M->Next;
 }
 M->Next = NULL;
}
```

+, Xóa vị trí thứ k (DeleteAny)

Xác định Node trước vị trí k rồi cho Node đó trỏ trực tiếp đến Node sau k.



Xóa vị trí thứ k

```
void DeleteAny(LinkList& L, int k) {
 if (k < 1 || k> Quantity(L)) cout << "Not Delete." << endl; // kiểm
tra k
 else {
  if (k == 1) DeleteHead(L);
  else {
   Node* M = L;
   int temp = 1;
   while (temp != k - 1 && M != NULL) {
    temp++;
```

```
     M = M->Next;
    }
    M->Next = M->Next->Next;
   }
  }
 }
```

C, Danh sách số nguyên toàn bộ chức năng trên

```
#include iostream
using namespace std;

typedef struct Node {
 int Data;
 Node* Next;
};

typedef struct Node* LinkList;

void InIt(LinkList& L) {
 L = NULL;
}

bool IsEmpty(LinkList L) {
 if (L == NULL) return true;
 return false;
}

int Quantity(LinkList L) {
 int temp = 0;
 for (Node* M = L; M != NULL; M = M->Next) {
  temp++;
 }
 return temp;
}
Node* AddNode(int x) {
 Node* P;
 P = (Node*)malloc(sizeof(Node));
 P->Data = x;
 P->Next = NULL;
 return P;
}

void AddTail(LinkList& L, int x) {
 Node* P;
 P = AddNode(x);
 if (IsEmpty(L) == true) L = P;
 else {
  Node* M;
  M = L;
  while (M->Next != NULL) {
   M = M->Next;
  }
  M->Next = P;
  P->Next = NULL;
 }
}

void AddHead(LinkList& L, int x) {
 Node* P;
```

```
 P = AddNode(x);
 if (IsEmpty(L) == true) L = P;
 else {
  P->Next = L;
  L = P;
 }
}

void AddAny(LinkList& L, int k, int x) {
 if (k < 1 || k > Quantity(L)) cout << "Not Add. " << endl;
 else{
  if (k == 1) AddHead(L, x);
  else {
   Node* P;
   P = AddNode(x);
   Node* M = L;
   int temp = 1;
   while (temp != k - 1 && M != NULL) {
    temp++;
    M = M->Next;
   }
   P->Next = M->Next;
   M->Next = P;
  }
 }
}

void DeleteHead(LinkList& L) {
 L = L->Next;
}
void DeleteTail(LinkList& L) {
 Node* M = L;
 while(M->Next->Next != NULL) {
  M = M->Next;
 }
 M->Next =  NULL;
}

void DeleteAny(LinkList& L, int k) {
 if (k < 1 || k> Quantity(L)) cout << "Not Delete." << endl;
 else {
  if (k == 1) DeleteHead(L);
  else {
   Node* M = L;
   int temp = 1;
   while (temp != k - 1 && M != NULL) {
    temp++;
    M = M->Next;
   }
   M->Next = M->Next->Next;
  }
 }
}
void InPut(LinkList& L, int n) {
 InIt(L);
 int x;
 for (int i = 0; i < n; i++) {
  cout << "Value " << i << endl;
  cin >> x;
  AddTail(L, x);
 }
}

void OutPut(LinkList L) {
```

```
 for (Node* M = L; M != NULL; M = M->Next) {
  cout << M->Data << " ";
 }
}
int main() {
 LinkList L;
 int n, select;
 cout << "Quantity: ";
 cin >> n;
 InPut(L, n);
 do
 {
  system("cls");
  cout << "1. Add Head. " << endl;
  cout << "2. Add Tail. " << endl;
  cout << "3. Add Any. " << endl;
  cout << "4. Delete Head. " << endl;
  cout << "5. Delete Tail. " << endl;
  cout << "6. Delete Any. " << endl;
  cout << "9. OutPut. " << endl;
  cin >> select;
  switch (select)
  {
  case 1: {
   int x;
   cout << "Value: ";
   cin >> x;
   AddHead(L, x);
   break;
  }
  case 2: {
   int x;
   cout << "Value: ";
   cin >> x;
   AddTail(L, x);
   break;
  }
  case 3: {
   int x, k;
   cout << "Possity: ";
   cin >> k;
   cout << "Value: ";
   cin >> x;
   AddAny(L,k, x);
   break;
  }
  case 9: {
   OutPut(L);
   system("pause");
   break;
  }
  case 4:{
   DeleteHead(L);
   break;
    }
  case 5: {
   DeleteTail(L);
   break;
  }
  case 6: {
   int k;
   cout << "Possity Delete: ";
   cin >> k;
   DeleteAny(L, k);
```

```
180       break;
181      }
182      }
183    } while (select != 0);
184  }
```

Share this :