Bài giảng môn học

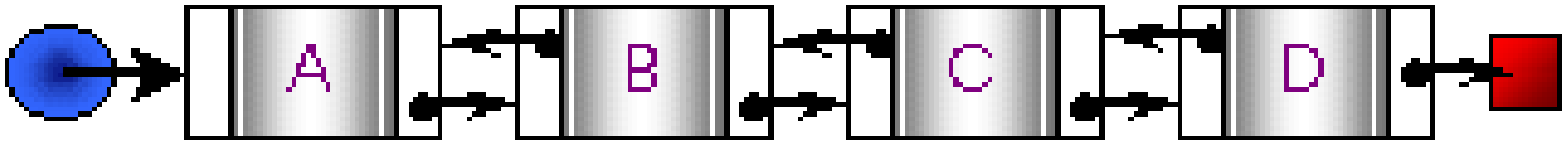
Chương 2:
Cấu trúc dữ liệu động

ThS. Nguyen Minh Phuc
Software Technology Department - Faculty of Information Technology – Lac Hong University

Cấu trúc dữ liệu và giải thuật

Danh sách liên kết kép

- Danh sách liên kết kép là danh sách mà mỗi phần tử trong danh sách có kết nối với 1 phần tử đứng trước và 1 phần tử đứng sau nó.



Danh sách liên kết kép

- Khai báo: Các khai báo sau định nghiã một danh sách liên kết kép đơn giản trong đó ta dùng hai con trỏ: pPrev liên kết với phần tử đứng trước và pNext như thường lệ, liên kết với phần tử đứng sau:

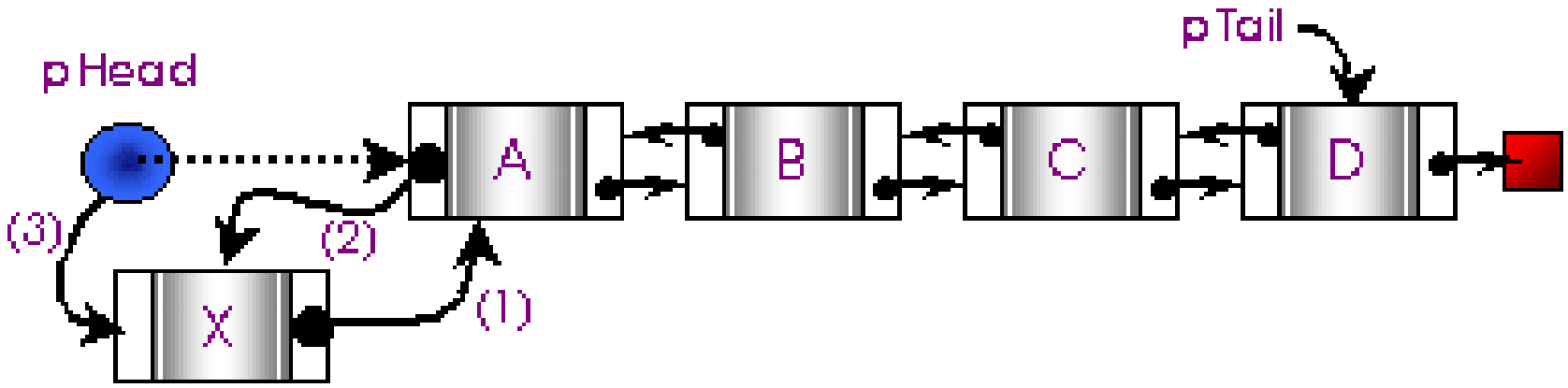
```
typedef struct  tagDNode
{
    Data            Info;
    struct tagDNode* pPre;          // trỏ đến phần tử đứng trước
    struct tagDNode* pNext;         // trỏ đến phần tử đứng sau
} DNODE;
typedef struct  tagDList
{
    DNODE* pHead;       // trỏ đến phần tử đầu danh sách
    DNODE* pTail;       // trỏ đến phần tử cuối danh sách
} DLIST;
```

- Khi đó, thủ tục khởi tạo một phần tử cho danh sách liên kết kép được viết lại như sau :

```
DNODE*          GetNode(Data x)
{
        DNODE *p;
          // Cấp phát vùng nhớ cho phần tử
        p = new DNODE;
        if ( p==NULL) {
                printf("Không đủ bộ nhớ");
                exit(1);
        }
// Gán thông tin cho phần tử p
        p ->Info = x;
        p->pPrev = NULL;
        p->pNext = NULL;
        return p;
}
```

Các thao tác cơ bản

- Chèn vào đầu danh sách

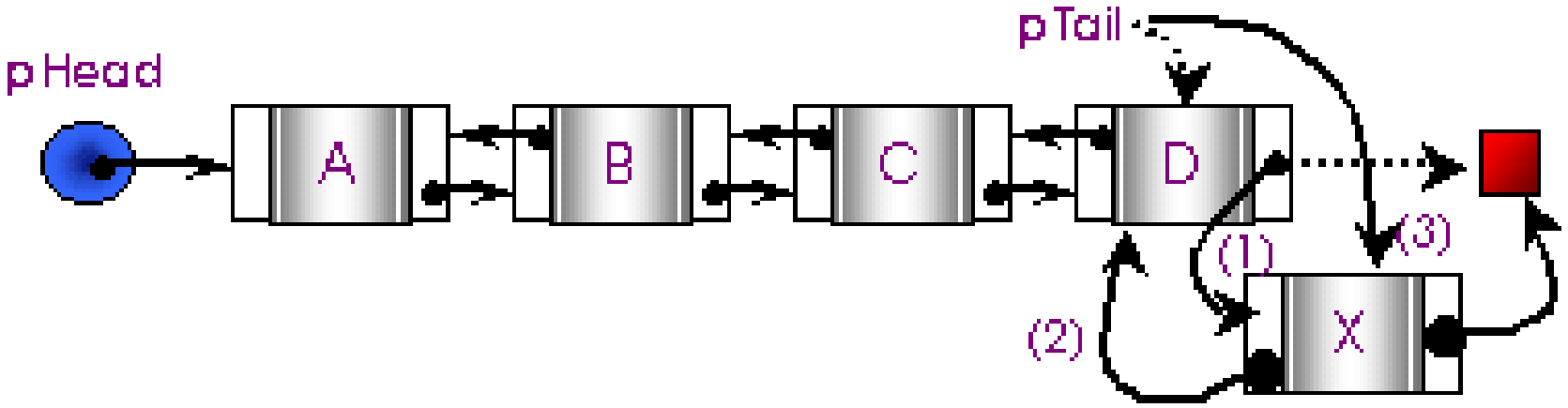


Các thao tác cơ bản


```
void AddFirst(DLIST &l, DNODE* new_ele)
{
if (l.pHead==NULL)  //Xâu rỗng
{
        l.pHead = new_ele;
        l.pTail = l.pHead;
}
else
{
        new_ele->pNext = l.pHead;   // (1)
        l.pHead ->pPrev = new_ele;  // (2)
        l.pHead = new_ele;          // (3)
}
}
NODE* InsertHead(DLIST &l, Data x)
{
        NODE* new_ele = GetNode(x);
if (new_ele ==NULL)  return NULL;
if (l.pHead==NULL)
{
        l.pHead = new_ele;
         l.pTail = l.pHead;
}
else
{
        new_ele->pNext = l.pHead;   // (1)
        l.pHead ->pPrev = new_ele;  // (2)
        l.pHead = new_ele;          // (3)
}
return new_ele;
}
```

Các thao tác cơ bản

- Chèn vào cuối danh sách

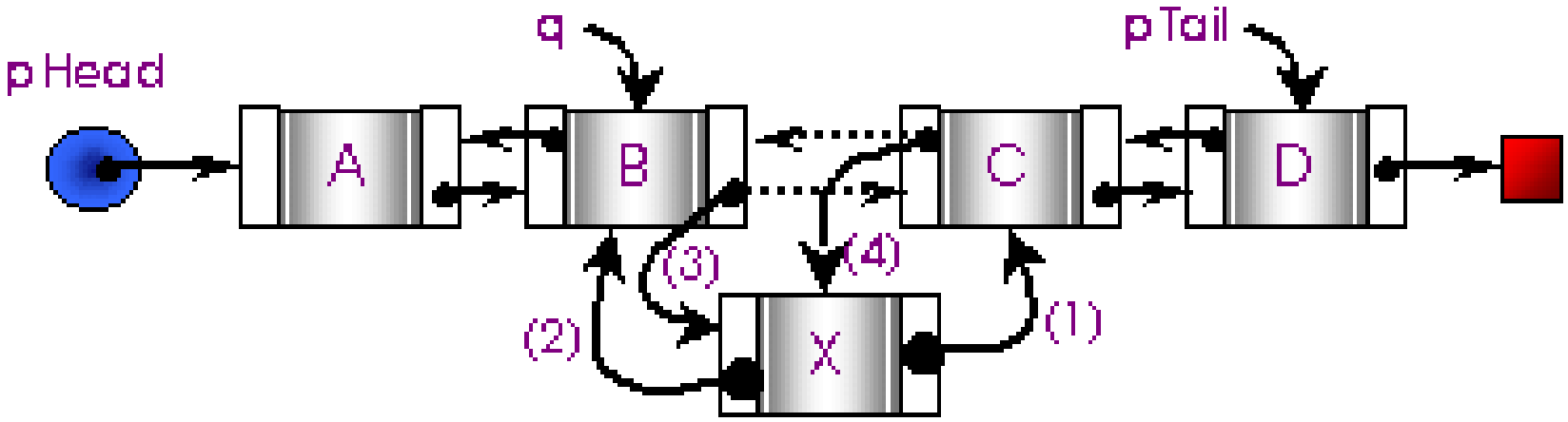


Các thao tác cơ bản


```
void AddTail(DLIST &l, DNODE *new_ele)
{
if (l.pHead==NULL)
{
        l.pHead = new_ele;
        l.pTail = l.pHead;
}
else
{
        l.pTail->Next = new_ele;      // (1)
        new_ele ->pPrev = l.pTail;    // (2)
        l.pTail = new_ele;   // (3)
}
}
NODE* InsertTail(DLIST &l, Data x)
{
        NODE* new_ele = GetNode(x);
if (new_ele ==NULL)  return NULL;
if (l.pHead==NULL)
{
        l.pHead = new_ele;
        l.pTail = l.pHead;
}
else
{
        l.pTail->Next = new_ele;      // (1)
        new_ele ->pPrev = l.pTail;    // (2)
        l.pTail = new_ele;   // (3)
}
return new_ele;
}
```

Các thao tác cơ bản

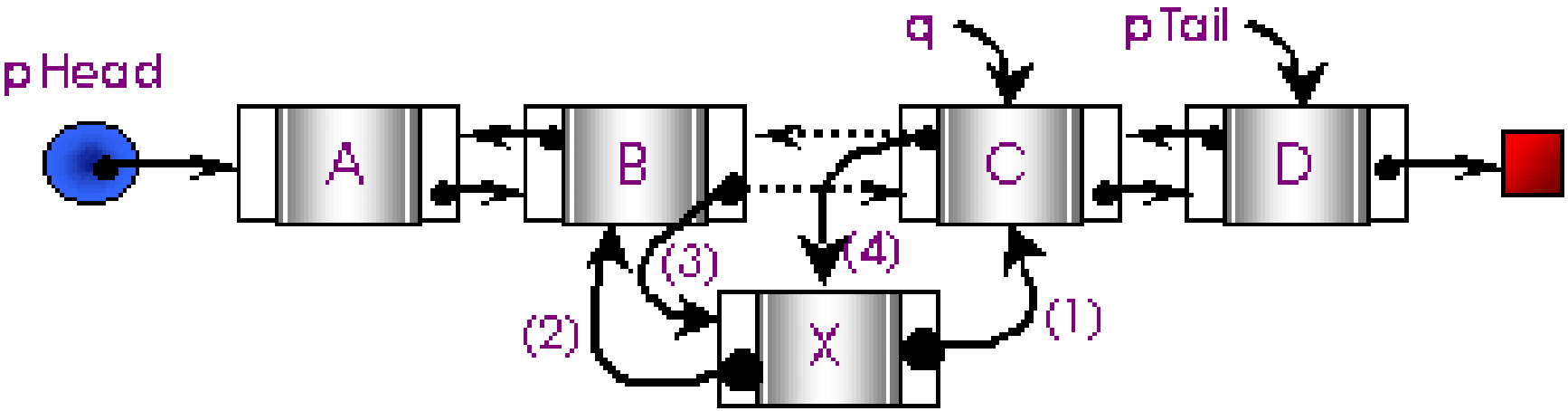
- Chèn vào danh sách sau một phần tử q



Các thao tác cơ bản


```
void AddAfter(DLIST &l, DNODE* q,DNODE*
new_ele)
{               DNODE* p = q->pNext;
if ( q!=NULL)
{
                new_ele->pNext = p;     //(1)
                new_ele->pPrev = q;     //(2)
                q->pNext = new_ele;     //(3)
if(p != NULL)
                p->pPrev = new_ele;     //(4)
                if(q == l.pTail)
                                l.pTail = new_ele;
}
                else  //chèn vào đầu danh sách
                                AddFirst(l, new_ele);
}
void InsertAfter(DLIST &l, DNODE *q, Data x)
{
                DNODE* p = q->pNext;
                                NODE* new_ele = GetNode(x);
 if (new_ele ==NULL)  return NULL;
 if ( q!=NULL)
{
                new_ele->pNext = p;     //(1)
                new_ele->pPrev = q;     //(2)
                q->pNext = new_ele;     //(3)
if(p != NULL)
                p->pPrev = new_ele;     //(4)
                if(q == l.pTail)
                                l.pTail = new_ele;
}
else  //chèn vào đầu danh sách
                AddFirst(l, new_ele);
}
```

- Chèn vào danh sách trước một phần tử q



```
void AddBefore(DLIST &l, DNODE q, DNODE* new_ele)
{
        DNODE* p = q->pPrev;
if ( q!=NULL)
{
        new_ele->pNext = q;     //(1)
        new_ele->pPrev = p;     //(2)
        q->pPrev = new_ele;     //(3)
if(p != NULL)
        p->pNext = new_ele;     //(4)
if(q == l.pHead)
        l.pHead = new_ele;
}
else  //chèn vào đầu danh sách
        AddTail(l, new_ele);
}
void InsertBefore(DLIST &l, DNODE q, Data x)
{
        DNODE* p = q->pPrev;
NODE* new_ele = GetNode(x);
if (new_ele ==NULL)
        return NULL;
  if ( q!=NULL)
{
        new_ele->pNext = q;     //(1)
        new_ele->pPrev = p;     //(2)
        q->pPrev = new_ele;     //(3)
if(p != NULL)
        p->pNext = new_ele;     //(4)
if(q == l.pHead)
        l.pHead = new_ele;
}
else  //chèn vào đầu danh sách
        AddTail(l, new_ele);
}
```

- Hủy một phần tử khỏi danh sách: Có 5 loại thao tác thông dụng hủy một phần tử ra khỏi xâu. Chúng ta sẽ lần lượt khảo sát chúng

- **Hủy phần tử đầu danh sách**

```
Data RemoveHead(DLIST &l)
{
        DNODE      *p;
        Data   x = NULLDATA;
  if ( l.pHead != NULL)
{
        p = l.pHead;
        x = p->Info;
        l.pHead = l.pHead->pNext;
        l.pHead->pPrev = NULL;
        delete p;
if(l.pHead == NULL)
        l.pTail = NULL;
else
        l.pHead->pPrev = NULL;
}
return x;
}
```

- **Hủy phần tử cuối danh sách**

```
Data RemoveTail (DLIST &l)
{
        DNODE *p;
        Data  x = NULLDATA;
if ( l.pTail != NULL)
{
        p = l.pTail;
        x = p->Info;
        l.pTail = l.pTail->pPrev;
        l.pTail->pNext = NULL;
delete p;
if (l.pHead == NULL)
        l.pTail = NULL;
else
        l.pHead->pPrev = NULL;
}
return x;
}
```

- **Hủy một phần tử đứng sau phần tử q:**

```
void RemoveAfter (DLIST &l, DNODE *q)
{
        DNODE *p;
if ( q != NULL)
{
        p = q ->pNext ;
if ( p != NULL)
{
        q->pNext = p->pNext;
if(p == l.pTail)
        l.pTail = q;
else
        p->pNext->pPrev = q;
delete p;
}
}
else
        RemoveHead(l);
}
```

- **Hủy một phần tử đứng trước phần tử q**

```
void RemoveAfter (DLIST &l, DNODE *q)
{
        DNODE       *p;
if ( q != NULL)
{
        p = q ->pPrev;
if ( p != NULL)
{
q->pPrev = p->pPrev;
if(p == l.pHead)
        l.pHead = q;
else
        p->pPrev->pNext = q;
delete p;
}
}
else
        RemoveTail(l);
}
```

- **Hủy phần tử có khoá k**

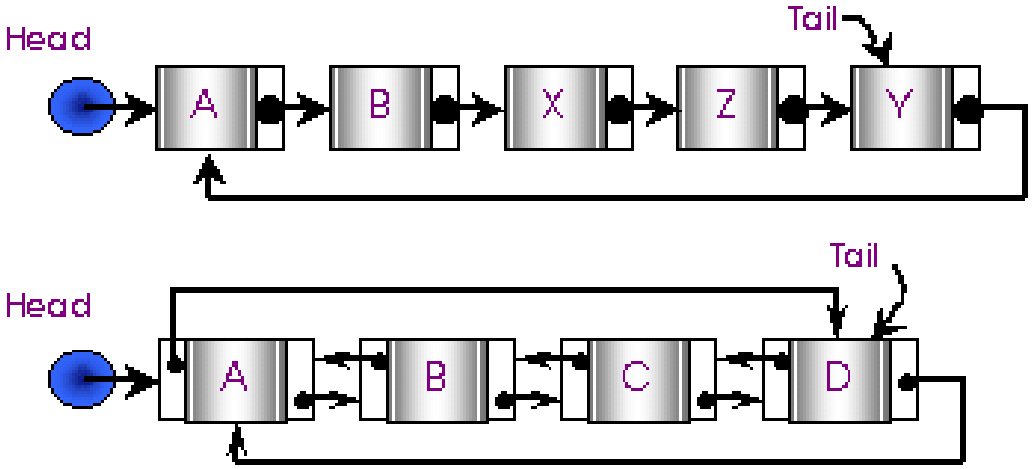
```
int RemoveNode(DLIST &l, Data k)
{
        DNODE   *p = l.pHead;
        NODE    *q;
while( p != NULL)
{
        if(p->Info == k) break;
                p = p->pNext;
}
if(p == NULL) return 0; //Không tìm thấy k
q = p->pPrev;
if ( q != NULL)
{
        p = q ->pNext ;
if ( p != NULL)
{
        q->pNext = p->pNext;
if(p == l.pTail)
        l.pTail = q;
else
        p->pNext->pPrev = q;
}
}
else //p là phần tử đầu xâu
{
l.pHead = p->pNext;
if(l.pHead == NULL)
        l.pTail = NULL;
else
l.pHead->pPrev = NULL;
}
delete p;
return 1;
}
```

Nhận xét

- Danh sách liên kết kép về mặt cơ bản có tính chất giống như Danh sách liên kết đơn. Tuy nhiên nó có một số tính chất khác xâu đơn như sau:
- @ Danh sách liên kết kép có mối liên kết hai chiều nên từ một phần tử bất kỳ có thể truy xuất một phần tử bất kỳ khác. Trong khi trên Danh sách liên kết đơn ta chỉ có thể truy xuất đến các phần tử đứng sau một phần tử cho trước. Điều này dẫn đến việc ta có thể dễ dàng hủy phần tử cuối Danh sách liên kết kép, còn trên Danh sách liên kết đơn thao tác này tốn chi phí O(n).
- @ Bù lại, Danh sách liên kết kép tốn chi phí gấp đôi so với Danh sách liên kết đơn cho việc lưu trữ các mối liên kết. Điều này khiến việc cập nhật cũng nặng nề hơn trong một số trường hợp. Như vậy ta cần cân nhắc lựa chọn CTDL hợp lý khi cài đặt cho một ứng dụng cụ thể.

Danh sách liên kết vòng

- Danh sách liên kết vòng (xâu vòng) là một danh sách đơn (hoặc kép) mà phần tử cuối danh sách thay vì mang giá trị NULL, trỏ tới phần tử đầu danh sách. Để biểu diễn, ta có thể xử dụng các kỹ thuật biểu diễn như danh sách đơn (hoặc kép).



Khai báo

- Ta có thể khai báo xâu vòng như khai báo xâu đơn (hoặc kép).

Các thao tác

- **Tìm phần tử trên danh sách vòng:** Danh sách vòng không có phần tử đầu danh sách rõ rệt, nhưng ta có thể đánh dấu một phần tử bất kỳ trên danh sách xem như phân tử đầu xâu để kiểm tra việc duyệt đã qua hết các phần tử của danh sách hay chưa.

code


```
NODE* Search(LIST &l, Data  x)
{
NODE    *p;
p = l.pHead;
do {
if ( p->Info == x)
return p;
    p = p->pNext;
} while  (p != l.pHead); // chưa đi giáp
vòng
```

Thêm phần tử mới

- Thêm phần tử đầu xâu

```
void   AddHead(LIST &l, NODE *new_ele)
{
       if(l.pHead == NULL)  //Xâu rỗng
       {
              l.pHead = l.pTail = new_ele;
              l.pTail->pNext = l.pHead;
       }
       else
       {
              new_ele->pNext = l.pHead;
              l.pTail->pNext = new_ele;
              l.pHead = new_ele;
       }
}
```

Thêm phần tử mới

- Thêm phần tử cuối xâu

```
void   AddTail(LIST &l, NODE *new_ele)
{
       if(l.pHead == NULL)  //Xâu rỗng
       {
              l.pHead = l.pTail = new_ele;
              l.pTail->pNext = l.pHead;
       }
       else
       {
              new_ele->pNext = l.pHead;
              l.pTail->pNext = new_ele;
              l.pTail = new_ele;
}
}
```

Thêm phần tử mới

- Thêm phần tử sau nút q

```
void    AddAfter(LIST &l, NODE *q, NODE *new_ele)
{
        if(l.pHead == NULL)  //Xâu rỗng
        {
                l.pHead = l.pTail = new_ele;
                l.pTail->pNext = l.pHead;
        }
        else
        {
                new_ele->pNext = q->pNext;
                q->pNext = new_ele;
                if(q == l.pTail)
                l.pTail = new_ele;
        }
}
```

Hủy phần tử đầu xâu


```
void RemoveHead(LIST &l)
{
	NODE  *p = l.pHead;
	if(p == NULL) return;
	if (l.pHead = l.pTail)
		l.pHead = l.pTail = NULL;
	else
	{
		l.pHead = p->Next;
		if(p == l.pTail)
			l.pTail->pNext = l.pHead;
	}
delete p;
}
```

Hủy phần tử đứng sau nút q


```
void RemoveAfter(LIST &l, NODE *q)
{
        NODE    *p;
        if(q != NULL)
        {
                p = q ->Next ;
                if ( p == q)
                        l.pHead = l.pTail = NULL;
                else
                {
                        q->Next = p->Next;
                        if(p == l.pTail)
                                l.pTail = q;
                }
        delete p;
        }
}
```

Nhật xét

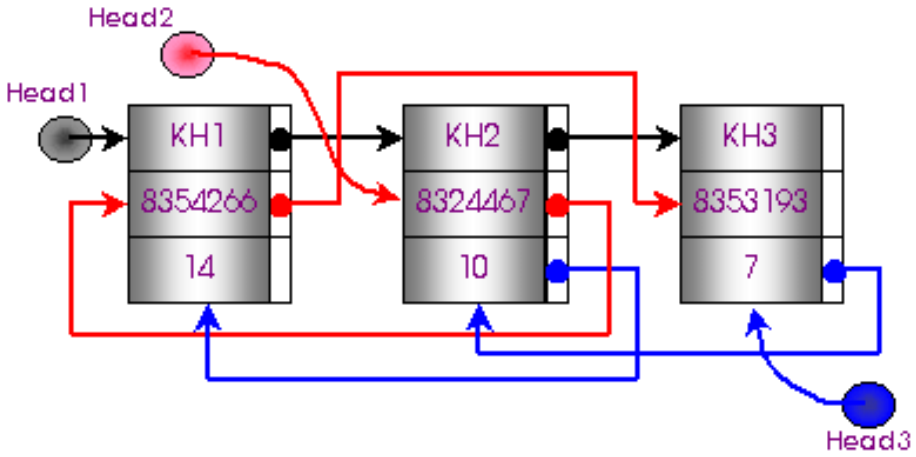
- Đối với danh sách vòng, có thể xuất phát từ một phần tử bất kỳ để duyệt toàn bộ danh sách

Danh sách có nhiều mối liên kết

- Danh sách có nhiều mối liên kết là danh sách mà mỗi phần tử có nhiều khoá và chúng được liên kết với nhau theo từng loại khoá.
- Danh sách có nhiều mối liên kết thường được xử dụng trong các ứng dụng quản lý một cơ sở dữ liệu lớn với những nhu cầu tìm kiếm dữ liệu theo những khoá khác nhau.

Ví dụ

- Để quản lý danh mục điện thoại thuận tiện cho việc in danh mục theo những trình tự khác nhau : tên khách tăng dần, theo số điện thoại tăng dần, thời gian lắp đặt giảm dần, ta có thể tổ chức dữ liệu như hình trên: một danh sách vơi 3 mối liên kết:



Một số cấu trúc tuyến tính đặc biệt

- Ngăn xếp (stack)
- Hàng đợi (queue)

Ngăn xếp - Stack

- Ngăn xếp thường được sử dụng để lưu trữ dữ liệu tạm thời trong quá trình chờ xử lý theo nguyên tắc: vào sau ra trước (Last In First Out - LIFO)

Định nghiã

- Một ngăn xếp các phần tử kiểu T là một chuỗi nối tiếp các phần tử của T, kèm các tác vụ sau:
  - 1. Tạo một đối tượng ngăn xếp rỗng.
  - 2. Đẩy (push) một phần tử mới vào ngăn xếp, giả sử ngăn xếp chưa đầy (phần tử dữ liệu mới luôn được thêm tại đỉnh).
  - 3. Lấy (pop) một phần tử ra khỏi ngăn xếp, giả sử ngăn xếp chưa rỗng (phần tử bị loại là phần tử đang nằm tại đỉnh).
  - 4. Xem phần tử tại đỉnh ngăn xếp (top).

Cài đặt ngăn xếp bằng xâu đơn

- **Khai báo Cấu trúc dữ liệu:**

```
typedef <Định nghiã kiểu T>;
typedef struct Node
{
    T  Data;
    struct Node *Next; // Vùng liên kết
} NodeType;
typedef NodeType *StackType;
```

Cài đặt ngăn xếp bằng xâu đơn

- **Khai báo con trỏ đầu Stack:**

```
StackType  Stack;
```

Cài đặt ngăn xếp bằng xâu đơn

- **Các Thao tác:**
  - **Tạo Stack Rỗng:** `Stack = NULL;`
  - **Kiểm tra Ngăn xếp Rỗng:** `(Stack == NULL);`
  - **Thêm 1 phần tử có nội dung x vào đầu Stack:**

```
void Push(DataType x , StackType &Stack )
{
      NodePtr  P;
      P = CreateNode(x);
      P->Next = Stack; /*InsertFirst(P, Stack);*/
      Stack = P;
}
```

Cài đặt ngăn xếp bằng xâu đơn

- **Lấy Phần tử ở đỉnh ngăn xếp:**

```
KieuT  Pop(StackType &Stack)
{
      DataType  x;
      If (Stack==NULL)
      {
            puts("Stack rong");
            exit(1);
      }
      x = (Stack)->Data; /*Xoa nut dau*/
      P = Stack;
      Stack = P->Next;
      free(P);
return x;
}
```

Cài đặt ngăn xếp bằng mảng và các thao tác

- **Cấu trúc dữ liệu:**

```
#define MaxSize 100  /*Kích thước của ngăn xếp*/
typedef    <Định nghiã kiểu T >
```

- **Khai báo kiểu mảng:**

```
typedef  KiểuT
StackArray[MaxSize];
```

- **Khai báo một Stack:**

```
StackArray Stack;
int top;
```

Cài đặt ngăn xếp bằng mảng

- **Khởi tạo 1 Stack rỗng: top = -1;**
- **Kiểm tra ngăn xếp rỗng: top == -1**
- **Kiểm tra ngăn xếp đầy: top == MaxSize-1;**
- **Thêm 1 phần tử có nội dung x vào đầu Stack:**
- **Thuật toán:**
  - Nếu đầy stack thì kết thúc;
  - Nếu không
  - Tăng đỉnh stack
  - Cất dữ liệu vào đỉnh stack

Cài đặt


```
void Push(KieuT x, StackArray Stack, int
top)
{
    If (*top == MaxSize-1) exit(1);
    top++;
    Stack[top]= x;
}
```

Lấy Phần tử ở đỉnh ngăn xếp

- **Thuật toán:**
  - Nếu stack rỗng thì kết thúc với lỗi
  - Nếu không
  - Lấy dữ liệu tại đỉnh stack
  - Giảm đỉnh stack
- **Cài đặt:**

```
KieuT Pop(StackArray Stack, int top)
{
        KieuT  Item;
        If (top == -1)
                exit(1);
        Item = Stack[top];
        top--;
        return Item;
}
```

Ứng dụng ngăn xếp trong xử lý biểu thức hậu tố:

- **Biểu thức sử dụng dấu ngoặc toàn phần:** Cách viết Biểu thức sử dụng ngoặc toàn phần dựa trên 4 qui tắc sau:
  - Một biến là 1 BTNTP.
  - Nếu x và y là 2 BTNTP, β là phép toán 2 ngôi thì (xβy) là BTNTP.
  - Nếu x là BTNTP, α là phép toán 1 ngôi thì (αx) là BTNTP.
  - Ngoài các biểu thức được tạo thành từ các qui tắc trên không còn các BTSDNTP khác.
- Ví dụ: ((A-B)*C)

Biểu thức trung tố

- Là biểu thức được viết dưới dạng BTNTP nhưng bỏ bớt các dấu ngoặc dựa trên thứ tự ưu tiên thực hiện của các phép toán.
- Ví dụ: ((((A/B)*C) - (C/(D*E))) -(F - G) ==> A/B*C - C/(D*E) - (F - G)

Biểu thức hậu tố (Ký pháp nghịch đảo BaLan)

- Dựa trên 4 qui tắc:
  - Một *biến* là 1 BTHT.
  - Nếu x và y là 2 BTHT, β là phép toán 2 ngôi thì **xyβ** là BTHT.
  - Nếu x là BTHT, α là phép toán 1 ngôi thì **xα** là BTHT.
  - Ngoài các biểu thức được tạo thành từ các qui tắc trên không còn các BTHT khác.
- Ví dụ: Trung tố: (A-B)+C ==> Hậu tố: AB-C+

Biểu thức tiền tố

- Dựa trên 4 qui tắc:
  - Một *biến* là 1 BTTT.
  - Nếu x và y là 2 BTTT, β là phép toán 2 ngôi thì **βxy** là BTTT.
  - Nếu x là BTTT, α là phép toán 1 ngôi thì **αx** là BTTT.
  - Ngoài các biểu thức được tạo thành từ các qui tắc trên không còn các BTTT khác.
- Ví dụ: Trung tố: (A-B)+C ==> Hậu tố: AB-C+ ==> Tiền tố: +-ABC

Hàng đợi - Queue

- Loại danh sách này có hành vi giống như việc xếp hàng chờ mua vé, với qui tắc **Đến trước - Mua trước. (First in First Out - FIFO)**
- Ví dụ: Bộ đệm bàn phím, tổ chức công viêc chờ in trong Print Manager của Windows.

Khái niệm

- Hàng đợi là một kiểu danh sách đặt biệt trong đó
- Các thao tác chèn đều thực hiện ở cuối danh sách
- Các thao tác lấy dữ liệu được thực hiện ở đầu (head) danh sách.
- Các phép toán cơ bản:
  - Tạo 1 hàng đợi rỗng.
  - Xác định hàng đợi có rỗng hay không
  - Thêm phần tử vào cuối hàng đợi
  - Lấy ra và hủy phần tử ở đầu hàng đợi.

Cài đặt hàng đợi bằng xâu liên kết

- Để tăng nhanh thao tác chèn vào cuối danh sách, ta sử dụng thêm 1 con trỏ phụ: trỏ tới phần tử cuối cùng.

```
NodePtr  Head, Tail;
```

- **Khởi tạo hàng đợi rỗng:**

```
void CreateQ(NodePtr &Head, NodePtr &Last)
{
   Head = NULL;
   Tail = NULL;
}
```

Thuận toán

- Kiểm tra hàng đợi rỗng: Head == NULL
- Chèn dữ liệu X vào cuối hàng đợi:

Thuật toán: Bắt đầu:

```
b1:Tạo nút chứa x: P = CreateNode(x);
b2: Nếu Head = NULL thì chèn đầu(P, đầu); đuôi = đầu;
Nếu Head != NULL thì đuôi^.Next = P; đuôi = đuôi^.Next;
Kết thúc
```

Cài đặt:

```
void AddQ( KieuT x, NodePtr &Head, NodePtr &Tail )
{
        NodePtr P;
        P = CreateNode(x);
        if (Head == NULL){
                Head = P;
                Tail = Head;
        }
        else
        {
                Last->Next = P;
                Tail = P;
        }
}
```

Lấy dữ liệu từ đầu hàng đợi

- **Ý tưởng:**
  - Nếu (Hàng đợi rỗng) kết thúc
  - Nếu (Hàng đợi không rỗng)
  - Lưu dữ liệu đầu hàng đợi vào 1 biến cùng kiểu T.
  - Xóa nút đầu hàng đợi
  - Nếu (hàng đợi rỗng) thì cập nhật Last = NULL

```
KieuT GetQ( NodePtr  &Head, NodePtr &Tail)
{
        NodePtr P; KieuT x;
        if (Head != NULL){          printf("Queue rong"); exit(1); }
        x = Head->Data;
        P = Head;           /* DeleteFirst(Head);*/
        Head = P->Next;
        free(P);
        if (Head == NULL) Tail = NULL;
}
```

Cấu trúc dữ liệu và giải thuật