HỒ SĨ ĐÀM (Chủ biên)
ĐỖ ĐỨC ĐÔNG – LÊ MINH HOÀNG – NGUYỄN THANH HÙNG

# TÀI LIỆU GIÁO KHOA

# CHUYÊN TIN

## QUYỂN 2

NHÀ XUẤT BẢN GIÁO DỤC VIỆT NAM

Công ty Cổ phần dịch vụ xuất bản Giáo dục Hà Nội - Nhà xuất bản Giáo dục Việt Nam giữ quyền công bố tác phẩm.

349-2009/CXB/43-644/GD Mã số : 8I746H9

# LỜI NÓI ĐẦU

*Bộ Giáo dục và Đào tạo đã ban hành chương trình chuyên tin học cho các lớp chuyên 10, 11, 12. Dựa theo các chuyên đề chuyên sâu trong chương trình nói trên, các tác giả biên soạn bộ sách chuyên tin học, bao gồm các vấn đề cơ bản nhất về cấu trúc dữ liệu, thuật toán và cài đặt chương trình.*

*Bộ sách gồm ba quyển, quyển 1, 2 và 3. Cấu trúc mỗi quyển bao gồm: phần lí thuyết, giới thiệu các khái niệm cơ bản, cần thiết trực tiếp, thường dùng nhất; phần áp dụng, trình bày các bài toán thường gặp, cách giải và cài đặt chương trình; cuối cùng là các bài tập. Các chuyên đề trong bộ sách được lựa chọn mang tính hệ thống từ cơ bản đến chuyên sâu.*

*Với trải nghiệm nhiều năm tham gia giảng dạy, bồi dưỡng học sinh chuyên tin học của các trường chuyên có truyền thống và uy tín, các tác giả đã lựa chọn, biên soạn các nội dung cơ bản, thiết yếu nhất mà mình đã sử dụng để dạy học với mong muốn bộ sách phục vụ không chỉ cho giáo viên và học sinh chuyên PTTH mà cả cho giáo viên, học sinh chuyên tin học THCS làm tài liệu tham khảo cho việc dạy và học của mình.*

*Với kinh nghiệm nhiều năm tham gia bồi dưỡng học sinh, sinh viên tham gia các kì thi học sinh giỏi Quốc gia, Quốc tế Hội thi Tin học trẻ Toàn quốc, Olympiad Sinh viên Tin học Toàn quốc, Kì thi lập trình viên Quốc tế khu vực Đông Nam Á, các tác giả đã lựa chọn giới thiệu các bài tập, lời giải có định hướng phục vụ cho không chỉ học sinh mà cả sinh viên làm tài liệu tham khảo khi tham gia các kì thi trên.*

*Lần đầu tập sách được biên soạn, thời gian và trình độ có hạn chế nên chắc chắn còn nhiều thiếu sót, các tác giả mong nhận được ý kiến đóng góp của bạn đọc, các đồng nghiệp, sinh viên và học sinh để bộ sách được ngày càng hoàn thiện hơn .*

**Các tác giả**

Chuyên đề 6

# KIỂU DỮ LIỆU TRỪU TƯỢNG VÀ CẤU TRÚC DỮ LIỆU

Kiểu dữ liệu trừu tượng là một mô hình toán học với những thao tác định nghĩa trên mô hình đó. Kiểu dữ liệu trừu tượng có thể không tồn tại trong ngôn ngữ lập trình mà chỉ dùng để tổng quát hóa hoặc tóm lược những thao tác sẽ được thực hiện trên dữ liệu. Kiểu dữ liệu trừu tượng được cài đặt trên máy tính bằng các cấu trúc dữ liệu: Trong kỹ thuật lập trình cấu trúc (Structural Programming), cấu trúc dữ liệu là các biến cùng với các thủ tục và hàm thao tác trên các biến đó. Trong kỹ thuật lập trình hướng đối tượng (Object-Oriented Programming), cấu trúc dữ liệu là kiến trúc thứ bậc của các lớp, các thuộc tính và phương thức tác động lên chính đối tượng hay một vài thuộc tính của đối tượng.

Trong chương này, chúng ta sẽ khảo sát một vài kiểu dữ liệu trừu tượng cũng như cách cài đặt chúng bằng các cấu trúc dữ liệu. Những kiểu dữ liệu trừu tượng phức tạp hơn sẽ được mô tả chi tiết trong từng thuật toán mỗi khi thấy cần thiết.

## 1. Danh sách

### *1.1. Khái niệm danh sách*

Danh sách là một tập sắp thứ tự các phần tử cùng một kiểu. Đối với danh sách, người ta có một số thao tác: Tìm một phần tử trong danh sách, chèn một phần tử vào danh sách, xóa một phần tử khỏi danh sách, sắp xếp lại các phần tử trong danh sách theo một trật tự nào đó v.v…

Việc cài đặt một danh sách trong máy tính tức là tìm một cấu trúc dữ liệu cụ thể mà máy tính hiểu được để lưu các phần tử của danh sách đồng thời viết các đoạn chương trình con mô tả các thao tác cần thiết đối với danh sách.

Vì danh sách là một tập sắp thứ tự các phần tử cùng kiểu, ta ký hiệu $TElement$ là kiểu dữ liệu của các phần tử trong danh sách, khi cài đặt cụ thể, $TElement$ có thể là bất cứ kiểu dữ liệu nào được chương trình dịch chấp nhận (Số nguyên, số thực, ký tự, …).

## *1.2. Biểu diễn danh sách bằng mảng*

Khi cài đặt danh sách bằng mảng một chiều , ta cần có một biến nguyên $n$ lưu số phần tử hiện có trong danh sách. Nếu mảng được đánh số bắt đầu từ 1 thì các phần tử trong danh sách được cất giữ trong mảng bằng các phần tử được đánh số từ 1 tới $n$: $A = a[1 \dots n]$

### a) Truy cập phần tử trong mảng

Việc truy cập một phần tử ở vị trí $p$ trong mảng có thể thực hiện rất dễ dàng qua phần tử $a_p$. Vì các phần tử của mảng có kích thước bằng nhau và được lưu trữ liên tục trong bộ nhớ, việc truy cập một phần tử được thực hiện bằng một phép toán tính địa chỉ phần tử có thời gian tính toán là hằng số. Vì vậy nếu cài đặt bằng mảng, việc truy cập một phần tử trong danh sách ở vị trí bất kỳ có độ phức tạp là $\Theta(1)$.

### b) Chèn phần tử vào mảng

Để chèn một phần tử $v$ vào mảng tại vị trí $p$, trước hết ta dồn tất cả các phần tử từ vị trí $p$ tới tới vị trí $n$ về sau một vị trí (tạo ra "chỗ trống" tại vị trí $p$), đặt giá trị $v$ vào vị trí $p$, và tăng số phần tử của mảng lên 1.

```
procedure Insert(p: Integer; const v: TElement);
//Thủ tục chèn phần tử v vào vị trí p
var i: Integer;
begin
  for i := n downto p do a[i + 1] := a[i];
  a[p] := v;
  n := n + 1;
end;
```

Trường hợp tốt nhất, vị trí chèn nằm sau phần tử cuối cùng của danh sách ($p = n + 1$), khi đó thời gian thực hiện của phép chèn là $\Theta(1)$. Trường hợp xấu nhất, ta cần chèn tại vị trí 1, khi đó thời gian thực hiện của phép chèn là $\Theta(n)$.

Cũng dễ dàng chứng minh được rằng thời gian thực hiện trung bình của phép chèn là $\Theta(n)$.

### c) Xóa phần tử khỏi mảng

Để xóa một phần tử tại vị trí $p$ của mảng mà vẫn giữ nguyên thứ tự các phần tử còn lại: Trước hết ta phải dồn tất cả các phần tử từ vị trí $p+1$ tới $n$ lên trước một vị trí (thông tin của phần tử thứ $p$ bị ghi đè), sau đó giảm số phần tử của mảng ($n$) đi .

```
procedure Delete(p: Integer); //Thủ tục xóa phần tử tại vị trí p
var i: Integer;
begin
  for i := p to n - 1 do a[i] := a[i + 1];
  n := n - 1;
end;
```

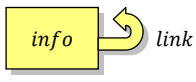
Trường hợp tốt nhất, vị trí xóa nằm cuối danh sách ($p = n$), khi đó thời gian thực hiện của phép xóa là $\Theta(1)$. Trường hợp xấu nhất, ta cần xóa tại vị trí 1, khi đó thời gian thực hiện của phép xóa là $\Theta(n)$. Cũng dễ dàng chứng minh được rằng thời gian thực hiện trung bình của phép xóa là $\Theta(n)$.

Trong trường hợp cần xóa một phần tử mà không cần duy trì thứ tự của các phần tử khác, ta chỉ cần đưa giá trị phần tử cuối cùng vào vị trí cần xóa rồi giảm số phần tử của mảng ($n$) đi 1. Khi đó thời gian thực hiện của phép xóa chỉ là $\Theta(1)$.

## *1.3. Biểu diễn danh sách bằng danh sách nối đơn*

Danh sách nối đơn (Singly-linked list) gồm các nút được nối với nhau theo một chiều. Mỗi nút là một bản ghi (record) gồm hai trường:

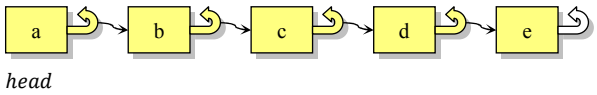
- Trường $info$ chứa giá trị lưu trong nút đó
- Trường $link$ chứa liên kết (con trỏ) tới nút kế tiếp, tức là chứa một thông tin đủ để biết nút kế tiếp nút đó trong danh sách là nút nào, trong trường hợp là nút cuối cùng (không có nút kế tiếp), trường liên kết này được gán một giá trị đặc biệt, chẳng hạn con trỏ $nil$.




```
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record; //Kiểu biến động chứa thông tin trong một nút
```

```
    info: TElement;
    link: PNode;
end;
```

Nút đầu tiên trong danh sách ($head$) đóng vai trò quan trọng trong danh sách nối đơn. Để duyệt danh sách nối đơn, ta bắt đầu từ nút đầu tiên, dựa vào trường liên kết để đi sang nút kế tiếp, đến khi gặp giá trị đặc biệt (duyệt qua nút cuối) thì dừng lại



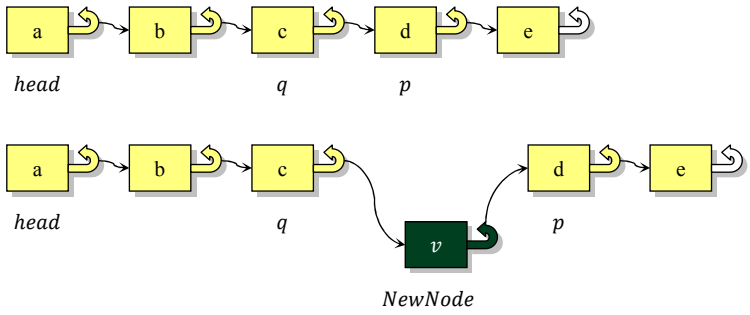

***Hình 1.1. Danh sách nối đơn***

### a) Truy cập phần tử trong danh sách nối đơn

Bản thân danh sách nối đơn đã là một kiểu dữ liệu trừu tượng. Để cài đặt kiểu dữ liệu trừu tượng này, chúng ta có thể dùng mảng các nút (trường $link$ chứa chỉ số của nút kế tiếp) hoặc biến cấp phát động (trường $link$ chứa con trỏ tới nút kế tiếp). Tuy nhiên vì cấu trúc nối đơn, việc xác định phần tử đứng thứ $p$ trong danh sách bắt buộc phải duyệt từ đầu danh sách qua $p$ nút, việc này mất thời gian trung bình $\Theta(n)$, và tỏ ra không hiệu quả như thao tác trên mảng. Nói cách khác, danh sách nối đơn tiện lợi cho việc truy cập tuần tự nhưng không hiệu quả nếu chúng ta thực hiện nhiều phép truy cập ngẫu nhiên.

### b) Chèn phần tử vào danh sách nối đơn

Để chèn thêm một nút chứa giá trị $v$ vào vị trí của nút $p$ trong danh sách nối đơn, trước hết ta tạo ra một nút mới $NewNode$ chứa giá trị $v$ và cho nút này liên kết tới $p$. Nếu $p$ đang là nút đầu tiên của danh sách ($head$) thì cập nhật lại $head$ bằng $NewNode$, còn nếu $p$ không phải nút đầu tiên của danh sách, ta tìm nút $q$ là nút đứng liền trước nút $p$ và chỉnh lại liên kết: $q$ liên kết tới $NewNode$ thay vì liên kết tới thẳng $p$ (h.1.2).




***Hình 1.2. Chèn phần tử vào danh sách nối đơn***

```
procedure Insert(p: PNode; const v: TElement);
//Thủ tục chèn phần tử v vào vị trí nút p
var NewNode, q: PNode;
begin
  New(NewNode);
  NewNode^.info := v;
  NewNode^.link := p;
  if head = p then head := NewNode
  else
    begin
      q := head;
      while q^.link ≠ p do q := q^.link;
      q^.link := NewNode;
    end;
end;
```

Việc chỉnh lại liên kết trong phép chèn phần tử vào danh sách nối đơn mất thời gian $\Theta(1)$, tuy nhiên việc tìm nút đứng liền trước nút $p$ yêu cầu phải duyệt từ đầu danh sách, việc này mất thời gian trung bình $\Theta(n)$. Vậy phép chèn một phần tử vào danh sách nối đơn mất thời gian trung bình $\Theta(n)$ để thực hiện.

### c) Xóa phần tử khỏi danh sách nối đơn:

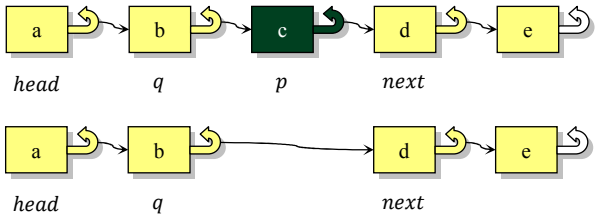
Để xóa nút $p$ khỏi danh sách nối đơn, gọi $next$ là nút đứng liền sau $p$ trong danh sách. Xét hai trường hợp:

- Nếu $p$ là nút đầu tiên trong danh sách $head = p$ thì ta đặt lại $head$ bằng $next$.

- Nếu $p$ không phải nút đầu tiên trong danh sách, tìm nút $q$ là nút đứng liền trước nút $p$ và chỉnh lại liên kết: $q$ liên kết tới $next$ thay vì liên kết tới $p$ (h.1.3)

Việc cuối cùng là huỷ nút $p$.

```
procedure Delete(p: PNode); //Thủ tục xóa nút p của danh sách nối đơn
var next, q: PNode;
begin
  next := p^.link;
  if p = head then head := next
  else
    begin
      q := head;
      while q^.link <> p do q := q^.link;
      q^.link := next;
    end;
  Dispose(p);
end;
```




*Hình 1.3. Xóa phần tử khỏi danh sách nối đơn*

Cũng giống như phép chèn, phép xóa một phần tử khỏi danh sách nối đơn cũng mất thời gian trung bình $\Theta(n)$ để thực hiện.

Trên đây mô tả các thao tác với danh sách biểu diễn dưới dạng danh sách nối đơn các biến động. Chúng ta có thể cài đặt danh sách nối đơn bằng một mảng, mỗi nút chứa trong một phần tử của mảng và trường liên kết $link$ chính là chỉ số của nút kế tiếp. Khi đó mọi thao tác chèn/xóa phần tử cũng được thực hiện tương tự như trên:

```
const max = ...; //Số phần tử cực đại
type
  TNode = record
```

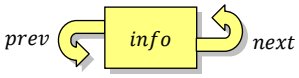
```
    info: TElement;
    link: Integer;
  end;
  TList = array[1..max] of TNode;
var
  Nodes: TList;
  head: Integer;
```

### *1.4. Biểu diễn danh sách bằng danh sách nối kép*

Việc xác định nút đứng liền trước một nút $p$ trong danh sách nối đơn bắt buộc phải duyệt từ đầu danh sách, thao tác này mất thời gian trung bình $\Theta(n)$ để thực hiện và ảnh hưởng trực tiếp tới thời gian thực hiện thao tác chèn/xóa phần tử. Để khắc phục nhược điểm này, người ta sử dụng danh sách nối kép.

Danh sách nối kép gồm các nút được nối với nhau theo hai chiều. Mỗi nút là một bản ghi (record) gồm ba trường:

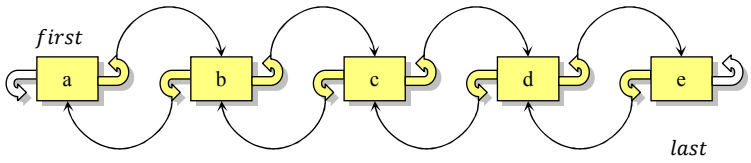
- Trường *info* chứa giá trị lưu trong nút đó.
- Trường $next$ chứa liên kết (con trỏ) tới nút kế tiếp, tức là chứa một thông tin đủ để biết nút kế tiếp nút đó là nút nào, trong trường hợp nút đứng cuối cùng trong danh sách (không có nút kế tiếp), trường liên kết này được gán một giá trị đặc biệt (chẳng hạn con trỏ $nil$)
- Trường $prev$ chứa liên kết (con trỏ) tới nút liền trước, tức là chứa một thông tin đủ để biết nút liền trước nút đó là nút nào, trong trường hợp nút đứng đầu tiên trong danh sách (không có nút liền trước), trường liên kết này được gán một giá trị đặc biệt (chẳng hạn con trỏ $nil$)




```
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record; //Kiểu biến động chứa thông tin trong một nút
    info: TElement;
    next, prev: PNode;
  end;
```

Khác với danh sách nối đơn, trong danh sách nối kép ta quan tâm tới hai nút: Nút đầu tiên ($first$) và phần tử cuối cùng ($last$). Có hai cách duyệt danh sách nối kép: Hoặc bắt đầu từ $first$, dựa vào liên kết $next$ để đi sang nút kế tiếp, đến

khi gặp giá trị đặc biệt (duyệt qua $last$) thì dừng lại. Hoặc bắt đầu từ $last$, dựa vào liên kết $prev$ để đi sang nút liền trước, đến khi gặp giá trị đặc biệt (duyệt qua $first$) thì dừng lại

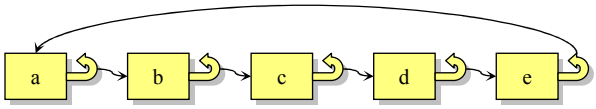



***Hình 1.4. Danh sách nối kép***

Giống như danh sách nối đơn, việc chèn/xóa nút trong danh sách nối kép cũng đơn giản chỉ là kỹ thuật chỉnh lại các mối liên kết giữa các nút cho hợp lý. Tuy nhiên ta có thể xác định được dễ dàng nút đứng liền trước/liền sau của một nút trong thời gian $\Theta(1)$, nên các thao tác chèn/xóa trên danh sách nối kép chỉ mất thời gian $\Theta(1)$, tốt hơn so với cài đặt bằng mảng hay danh sách nối đơn.

## *1.5. Biểu diễn danh sách bằng danh sách nối vòng đơn*

Trong danh sách nối đơn, phần tử cuối cùng trong danh sách có trường liên kết được gán một giá trị đặc biệt (thường sử dụng nhất là giá trị $nil$). Nếu ta cho trường liên kết của phần tử cuối cùng trỏ thẳng về phần tử đầu tiên của danh sách thì ta sẽ được một kiểu danh sách mới gọi là danh sách nối vòng đơn.



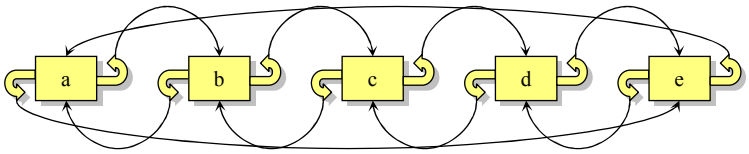

***Hình 1.5. Danh sách nối vòng đơn***

Đối với danh sách nối vòng đơn, ta chỉ cần biết một nút bất kỳ của danh sách là ta có thể duyệt được hết các nút trong danh sách bằng cách đi theo hướng liên kết. Chính vì lý do này, khi chèn/xóa vào danh sách nối vòng đơn, ta không phải xử lý các trường hợp riêng khi nút đứng đầu danh sách. Mặc dù vậy, danh sách nối vòng đơn vẫn cần thời gian trung bình $\Theta(n)$ để thực hiện thao tác chèn/xóa vì việc xác định nút đứng liền trước một nút cho trước cũng gặp trở ngại như với danh sách nối đơn.

## *1.6. Biểu diễn danh sách bằng danh sách nối vòng kép*

Danh sách nối vòng đơn chỉ cho ta duyệt các nút của danh sách theo một chiều, nếu cài đặt bằng danh sách nối vòng kép thì ta có thể duyệt các nút của danh sách cả theo chiều ngược lại nữa. Danh sách nối vòng kép có thể tạo thành từ danh sách nối kép nếu ta cho trường $prev$ của nút $first$ trỏ tới nút $Last$ còn trường $next$ của nút $last$ thì trỏ tới nút $first$.

Tương tự như danh sách nối kép, danh sách nối vòng kép cho phép thao tác chèn/xóa phần tử có thể thực hiện trong thời gian $\Theta(1)$.




***Hình 1.6. Danh sách nối vòng kép***

## *1.7. Biểu diễn danh sách bằng cây*

Có nhiều thao tác trên danh sách, nhưng những thao tác phổ biến nhất là truy cập phần tử, chèn và xóa phần tử. Ta đã khảo sát cách cài đặt danh sách bằng mảng hoặc danh sách liên kết, nếu như mảng cho phép thao tác truy cập ngẫu nhiên tốt hơn danh sách liên kết, thì thao tác chèn/xóa phần tử trên mảng lại mất khá nhiều thời gian.

Dưới đây là bảng so sánh thời gian thực hiện các thao tác trên danh sách.

| Phương pháp | Truy cập ngẫu nhiên | Chèn | Xóa |
|---|---|---|---|
| Mảng | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ |
| Danh sách nối đơn | $\Theta(n)$ | $\Theta(n)$ | $\Theta(n)$ |
| Danh sách nối kép | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| Danh sách nối vòng đơn | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ |
| Danh sách nối vòng kép | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |

Cây là một kiểu dữ liệu trừu tượng mà trong một số trường hợp có thể gián tiếp dùng để biểu diễn danh sách. Với một cách đánh số thứ tự cho các nút của cây (duyệt theo thứ tự giữa), mỗi phép truy cập ngẫu nhiên, chèn, xóa phần tử trên

danh sách có thể thực hiện trong thời gian $O(\log n)$. Chúng ta sẽ tiếp tục chủ đề này trong một bài riêng.

## Bài tập

**1.1.** Viết chương trình thực hiện các phép chèn, xóa, và tìm kiếm một phần tử trong danh sách các số nguyên đã sắp xếp theo thứ tự tăng dần biểu diễn bởi:

- Mảng
- Danh sách nối đơn
- Danh sách nối kép

**1.2.** Viết chương trình nối hai danh sách số nguyên đã sắp xếp, tổng quát hơn, viết chương trình nối $k$ danh sách số nguyên đã sắp xếp để được một danh sách gồm tất cả các phần tử.

**1.3.** Giả sử chúng ta biểu diễn một đa thức $p(x) = a_1x^{b_1} + a_2x^{b_2} + \cdots + a_nx^{b_n}$, trong đó $b_1 > b_2 > \cdots > b_n$ dưới dạng một danh sách nối đơn mà nút thứ $i$ của danh sách chứa hệ số $a_i$, số mũ $b_i$ và con trỏ tới nút kế tiếp (nút $i + 1$). Hãy tìm thuật toán cộng và nhân hai đa thức theo các biểu diễn này.

**1.4.** Một số nhị phân $a_n a_{n-1} \ldots a_0$, trong đó $a_i \in \{0,1\}$ có giá trị bằng $\sum_{i=0}^{n} a_i 2^i$. Người ta biểu diễn số nhị phân này bằng một danh sách nối đơn gồm $n$ nút, có nút đầu danh sách chứa giá trị $a_n$, mỗi nút trong danh sách chứa một chữ số nhị phân $a_i$ và con trỏ tới nút kế tiếp là nút chứa chữ số nhị phân $a_{i-1}$. Hãy lập chương trình thực hiện phép toán "cộng 1" trên số nhị phân đã cho và đưa ra biểu diễn nhị phân của kết quả.

*Gợi ý: Sử dụng phép đệ quy*

## 2. Ngăn xếp và hàng đợi

Ngăn xếp và hàng đợi là hai kiểu dữ liệu trừu tượng rất quan trọng và được sử dụng nhiều trong thiết kế thuật toán. Về bản chất, ngăn xếp và hàng đợi là danh sách tức là một tập hợp các phần tử cùng kiểu có tính thứ tự.

Trong phần này chúng ta sẽ tìm hiểu hoạt động của ngăn xếp và hàng đợi và cách cài đặt chúng bằng các cấu trúc dữ liệu. Tương tự như danh sách, ta gọi kiểu dữ liệu của các phần tử sẽ chứa trong ngăn xếp và hàng đợi là $TElement$. Khi cài đặt chương trình cụ thể, kiểu $TElement$ có thể là kiểu số nguyên, số thực, ký tự, hay bất kỳ kiểu dữ liệu nào được chương trình dịch chấp nhận.

## *2.1. Ngăn xếp*

*Ngăn xếp* (*Stack*) là một kiểu danh sách mà việc bổ sung một phần tử và loại bỏ một phần tử được thực hiện ở cuối danh sách.

Có thể hình dung ngăn xếp như một chồng đĩa, đĩa nào được đặt vào chồng sau cùng sẽ nằm trên tất cả các đĩa khác và sẽ được lấy ra đầu tiên. Vì nguyên tắc "vào sau ra trước", ngăn xếp còn có tên gọi là *danh sách kiểu LIFO* (*Last In First Out*). Vị trí cuối danh sách được gọi là *đỉnh* (*top*) của ngăn xếp.

Đối với ngăn xếp có sáu thao tác cơ bản:

- $Init$: Khởi tạo một ngăn xếp rỗng
- $IsEmpty$: Cho biến ngăn xếp có rỗng không?
- $IsFull$: Cho biết ngăn xếp có đầy không?
- $Get$: Đọc giá trị phần tử ở đỉnh ngăn xếp
- $Push$: Đẩy một phần tử vào ngăn xếp
- $Pop$: Lấy ra một phần tử từ ngăn xếp

### a) Biểu diễn ngăn xếp bằng mảng

Cách biểu diễn ngăn xếp bằng mảng cần có một mảng $Items$ để lưu các phần tử trong ngăn xếp và một biến nguyên $top$ để lưu chỉ số của phần tử tại đỉnh ngăn xếp. Ví dụ:

```
const max = ...; //Dung lượng cực đại của ngăn xếp
type
  TStack = record
    items: array[1..max] of TElement;
    top: Integer;
  end;
var Stack: TStack;
```

Sáu thao tác cơ bản của ngăn xếp có thể viết như sau:

```
//Khởi tạo ngăn xếp rỗng
procedure Init;
begin
  Stack.top := 0;
end;
//Hàm kiểm tra ngăn xếp có rỗng không?
function IsEmpty: Boolean;
begin
  Result := Stack.top = 0;
end;
//Hàm kiểm tra ngăn xếp có đầy không?
function IsFull: Boolean;
begin
  Result := Stack.top = max;
end;
//Đọc giá trị phần tử ở đỉnh ngăn xếp
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Stack is Empty" //Báo lỗi ngăn xếp rỗng
  else
    with Stack do Result := items[top];
//Trả về phần tử ở đỉnh ngăn xếp
end;
//Đẩy một phần tử x vào ngăn xếp
procedure Push(const x: TElement);
begin
  if IsFull then
    Error ← "Stack is Full" //Báo lỗi ngăn xếp đầy
  else
    with Stack do
      begin
        top := top + 1; //Tăng chỉ số đỉnh Stack
        items[top] := x; //Đặt x vào vị trí đỉnh Stack
      end;
end;
//Lấy một phần tử ra khỏi ngăn xếp
function Pop: TElement;
begin
```

```
  if IsEmpty then
    Error ← "Stack is Empty" //Báo lỗi ngăn xếp rỗng
  else
    with Stack do
      begin
        Result := items[top]; //Trả về phần tử ở đỉnh ngăn xếp
        top := top - 1; //Giảm chỉ số đỉnh ngăn xếp
      end;
end;
```

### b) Biểu diễn ngăn xếp bằng danh sách nối đơn kiểu LIFO

Ta sẽ trình bày cách cài đặt ngăn xếp bằng danh sách nối đơn các biến động và con trỏ. Trong cách cài đặt này, ngăn xếp sẽ bị đầy nếu như vùng không gian nhớ dùng cho các biến động không còn đủ để thêm một phần tử mới. Tuy nhiên, việc kiểm tra điều này phụ thuộc vào máy tính, chương trình dịch và ngôn ngữ lập trình. Mặt khác, không gian bộ nhớ dùng cho các biến động thường rất lớn nên ta sẽ không viết mã cho hàm $IsFull$: Kiểm tra ngăn xếp tràn.

Các khai báo dữ liệu:

```
type
  PNode = ^TNode; //Kiểu con trỏ liên kết giữa các nút
  TNode = record //Kiểu dữ liệu cho một nút
    info: TElement;
    link: PNode;
  end;
var top: PNode; //Con trỏ tới phần tử đỉnh ngăn xếp
```

Các thao tác trên ngăn xếp:

```
//Khởi tạo ngăn xếp rỗng
procedure Init;
begin
  top := nil;
end;
//Kiểm tra ngăn xếp có rỗng không
function IsEmpty: Boolean;
begin
  Result := top = nil;
end;
```

```
//Đọc giá trị phần tử ở đỉnh ngăn xếp
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Stack is Empty" //Báo lỗi ngăn xếp rỗng
  else
    Result := top^.info;
end;
//Đẩy một phần tử x vào ngăn xếp
procedure Push(const x: TElement);
var p: PNode;
begin
  New(p); //Tạo nút mới
  p^.info := x;
  p^.link := top; //Nối vào danh sách liên kết
  top := p; //Dịch con trỏ đỉnh ngăn xếp
end;
//Lấy một phần tử khỏi ngăn xếp
function Pop: TElement;
var p: PNode;
begin
  if IsEmpty then
    Error ← "Stack is Empty" //Báo lỗi ngăn xếp rỗng
  else
    begin
      Result := top^.info; //Lấy phần tử tại con trỏ top
      p := top^.link;
      Dispose(top); //Giải phóng bộ nhớ
      top := P; //Dịch con trỏ đỉnh ngăn xếp
    end;
end;
```

## *2.2. Hàng đợi*

*Hàng đợi* (*Queue*) là một kiểu danh sách mà việc bổ sung một phần tử được thực hiện ở cuối danh sách và việc loại bỏ một phần tử được thực hiện ở đầu danh sách.

Khi cài đặt hàng đợi, có hai vị trí quan trọng là vị trí đầu danh sách ($front$), nơi các phần tử được lấy ra, và vị trí cuối danh sách ($rear$), nơi phần tử cuối cùng được đưa vào.

Có thể hình dung hàng đợi như một đoàn người xếp hàng mua vé: Người nào xếp hàng trước sẽ được mua vé trước. Vì nguyên tắc "vào trước ra trước", hàng đợi còn có tên gọi là *danh sách kiểu FIFO* (*First In First Out*).

Tương tự như ngăn xếp, có sáu thao tác cơ bản trên hàng đợi:

- $Init$: Khởi tạo một hàng đợi rỗng
- $IsEmpty$: Cho biến hàng đợi có rỗng không?
- $IsFull$: Cho biết hàng đợi có đầy không?
- $Get$: Đọc giá trị phần tử ở đầu hàng đợi
- $Push$: Đẩy một phần tử vào hàng đợi
- $Pop$: Lấy ra một phần tử từ hàng đợi

### a) Biểu diễn hàng đợi bằng mảng

Ta có thể biểu diễn hàng đợi bằng một mảng $items$ để lưu các phần tử trong hàng đợi, một biến nguyêf$ront$ để lưu chỉ số phần tử đầu hàng đợi và một biến nguyên $rear$ để lưu chỉ số phần tử cuối hàng đợi. Chỉ một phần của mảng $items$ từ vị trí $front$ tới $rear$ được sử dụng lưu trữ các phần tử trong hàng đợi. Ví dụ:

```
const max = ...; //Dung lượng cực đại
type
  TQueue = record
    items: array[1..max] of TElement;
    front, rear: Integer;
  end;
var Queue: TQueue;
```

Sáu thao tác cơ bản trên hàng đợi có thể viết như sau:

```
//Khởi tạo hàng đợi rỗng
procedure Init;
begin
  Queue.front := 1;
  Queue.rear := 0;
end;
//Kiểm tra hàng đợi có rỗng không
```

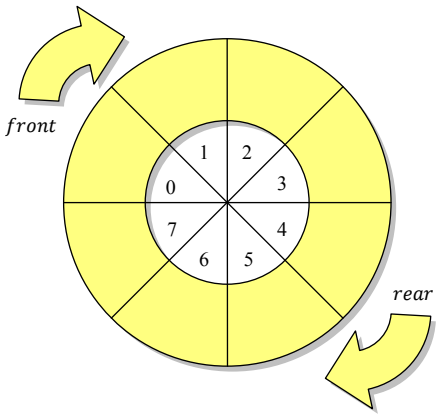
```
function IsEmpty: Boolean;
begin
  Result := Queue.front > Queue.rear;
end;
//Kiểm tra hàng đợi có đầy không
function IsFull: Boolean;
begin
  Result := Queue.rear = max;
end;
//Đọc giá trị phần tử đầu hàng đợi
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    with Queue do Result := items[front];
end;
//Đẩy một phần tử x vào hàng đợi
procedure Push(const x: TElement);
begin
  if IsFull then
    Error ← "Queue is Full" //Báo lỗi hàng đợi đầy
  else
    with Queue do
      begin
        rear := rear + 1;
        items[rear] := x;
      end;
end;
//Lấy một phần tử khỏi hàng đợi
function Pop: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    with Queue do
      begin
        Result := items[front];
```

```
        front := front + 1;
      end;
end;
```

### b) Biểu diễn hàng đợi bằng danh sách vòng

Xét việc biểu diễn ngăn xếp và hàng đợi bằng mảng, giả sử mảng có tối đa 10 phần tử, ta thấy rằng nếu như làm 6 lần thao tác $Push$, rồi 4 lần thao tác $Pop$, rồi tiếp tục 8 lần thao tác $Push$ nữa thì không có vấn đề gì xảy ra cả. Lý do là vì chỉ số $top$ lưu đỉnh của ngăn xếp sẽ được tăng lên 6000, rồi giảm về 2000, sau đó lại tăng trở lại lên 10000 (chưa vượt quá chỉ số mảng). Nhưng nếu ta thực hiện các thao tác đó đối với cách cài đặt hàng đợi như trên thì sẽ gặp thông báo lỗi tràn mảng, bởi mỗi lần đẩy phần tử vào ngăn xếp, chỉ số cuối hàng đợi $rear$ luôn tăng lên và không bao giờ bị giảm đi cả. Đó chính là nhược điểm mà ta nói tới khi cài đặt: Chỉ có các phần tử từ vị trí $front$ tới $rear$ là thuộc hàng đợi, các phần tử từ vị trí 1 tới $front - 1$ là vô nghĩa.

Để khắc phục điều này, ta có thể biểu diễn hàng đợi bằng một danh sách vòng (dùng mảng hoặc danh sách nối vòng đơn): coi như các phần tử của hàng đợi được xếp quanh vòng tròn theo một chiều nào đó (chẳng hạn chiều kim đồng hồ). Các phần tử nằm trên phần cung tròn từ vị trí $front$ tới vị trí $rear$ là các phần tử của hàng đợi. Có thêm một biến $n$ lưu số phần tử trong hàng đợi. Việc đẩy thêm một phần tử vào hàng đợi tương đương với việc ta dịch chỉ số $fear$ theo chiều vòng một vị trí rồi đặt giá trị mới vào đó. Việc lấy ra một phần tử trong hàng đợi tương đương với việc lấy ra phần tử tại vị trí $front$ rồi dịch chỉ số $front$ theo chiều vòng. (h.1.7)




***Hình 1.7. Dùng danh sách vòng mô tả hàng đợi***

Để tiện cho việc dịch chỉ số theo vòng, khi cài đặt danh sách vòng bằng mảng, người ta thường dùng cách đánh chỉ số từ 0 để tiện sử dụng phép chia lấy dư (modulus - mod).

```
const max = ...; //Dung lượng cực đại
type
  TQueue = record
    items: array[0..max - 1] of TElement;
    n, front, rear: Integer;
  end;
var Queue: TQueue;
```

Sáu thao tác cơ bản trên hàng đợi cài đặt trên danh sách vòng được viết dưới dạng giả mã như sau:

```
//Khởi tạo hàng đợi rỗng
procedure Init;
begin
  with Queue do
    begin
      front := 0;
      rear := max - 1;
      n := 0;
    end;
end;
//Kiểm tra hàng đợi có rỗng không
function IsEmpty: Boolean;
```

```
begin
  Result := Queue.n = 0;
end;
//Kiểm tra hàng đợi có đầy không
function IsFull: Boolean;
begin
  Result := Queue.n = max;
end;
//Đọc giá trị phần tử đầu hàng đợi
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    with Queue do Result := items[front];
end;
//Đẩy một phần tử vào hàng đợi
procedure Push(const x: TElement);
begin
  if IsFull then
    Error ← "Queue is Full" //Báo lỗi hàng đợi đầy
  else
    with Queue do
      begin
        rear := (rear + 1) mod max;
        items[rear] := x;
        Inc(n);
      end;
end;
//Lấy một phần tử ra khỏi hàng đợi
function Pop: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    with Queue do
      begin
        Result := items[front];
```

```
        front := (front + 1) mod max;
        Dec(n);
      end;
end;
```

### c) Biểu diễn hàng đợi bằng danh sách nối đơn kiểu FIFO

Tương tự như cài đặt ngăn xếp bằng biến động và con trỏ trong một danh sách nối đơn, ta cũng không viết hàm *IsFull* để kiểm tra hàng đợi đầy.

Các khai báo dữ liệu:

```
type
  PNode = ^TNode; //Kiểu con trỏ liên kết giữa các nút
  TNode = record //Kiểu dữ liệu cho một nút
    info: TElement;
    link: PNode;
  end;
var front, rear: PNode; //Con trỏ tới phần tử đầu và cuối hàng đợi
```

Các thao tác trên hàng đợi:

```
//Khởi tạo hàng đợi rỗng
procedure Init;
begin
  front := nil;
end;
//Kiểm tra hàng đợi có rỗng không
function IsEmpty: Boolean;
begin
  Result := front = nil;
end;
//Đọc giá trị phần tử đầu hàng đợi
function Get: TElement;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    Result := front^.info;
end;
//Đẩy một phần tử x vào hàng đợi
procedure Push(const x: TElement);
```

```
var p: PNode;
begin
  New(p); //Tạo một nút mới
  p^.info := x;
  p^.link := nil;
  //Nối nút đó vào danh sách
  if front = nil then front := p
  else rear^.link := p;
  rear := p; //Dịch con trỏ rear
end;
//Lấy một phần tử ra khỏi hàng đợi
function Pop: TElement;
var P: PNode;
begin
  if IsEmpty then
    Error ← "Queue is Empty" //Báo lỗi hàng đợi rỗng
  else
    begin
      Result := front^.info; //Lấy phần tử tại con trỏ front
      P := front^.link;
      Dispose(front); //Giải phóng bộ nhớ
      front := p; //Dịch con trỏ front
    end;
end;
```

## *2.3. Một số chú ý về kỹ thuật cài đặt*

Ngăn xếp và hàng đợi là hai kiểu dữ liệu trừu tượng tương đối dễ cài đặt, các thủ tục và hàm mô phỏng các thao tác có thể viết rất ngắn. Tuy vậy trong các chương trình dài, các thao tác vẫn nên được tách biệt ra thành chương trình con để dễ dàng gỡ rối hoặc thay đổi cách cài đặt (ví dụ đổi từ cài đặt bằng mảng sang cài đặt bằng danh sách nối đơn). Điều này còn giúp ích cho lập trình viên trong trường hợp muốn biểu diễn các kiểu dữ liệu trừu tượng bằng các lớp và đối tượng. Nếu có băn khoăn rằng việc gọi thực hiện chương trình con sẽ làm chương trình chạy chậm hơn việc viết trực tiếp, bạn có thể đặt các thao tác đó dưới dạng inline functions.

## Bài tập

**1.5.** Hàng đợi hai đầu (*doubled-ended queue*) là một danh sách được trang bị bốn thao tác:

$PushF(v)$: Đẩy phần tử $v$ vào đầu danh sách

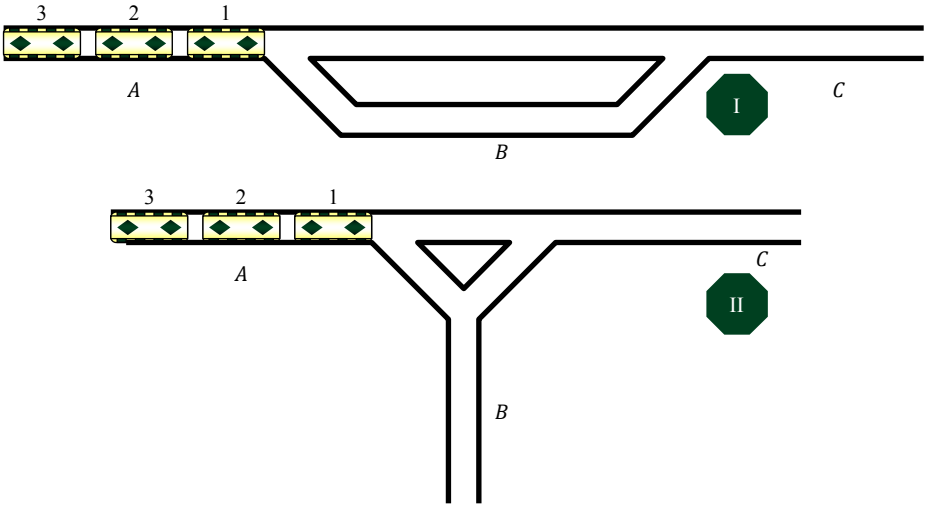
$PushR(v)$: Đẩy phần tử $v$ vào cuối danh sách

$PopF$: Loại bỏ phần tử đầu danh sách

$PopR$: Loại bỏ phần tử cuối danh sách

Hãy tìm cấu trúc dữ liệu thích hợp để cài đặt kiểu dữ liệu trừu tượng hàng đợi hai đầu.

**1.6.** Có hai sơ đồ đường ray xe lửa bố trí như hình sau:




Ban đầu có $n$ toa tàu xếp theo thứ tự từ 1 tới $n$ từ phải qua trái trên đường ray $A$. Người ta muốn xếp lại các toa tàu theo thứ tự mới từ phải qua trái $(p_1, p_2, \ldots, p_n)$ lên đường ray $C$ theo nguyên tắc: Các toa tàu không được "vượt nhau" trên ray, mỗi lần chỉ được chuyển một toa tàu từ $A \to B$, $A \to B$ hoặc $A \to C$. Hãy cho biết điều đó có thể thực hiện được trên sơ đồ đường ray nào trong hai sơ đồ trên.

## 3. Cây

### *3.1. Định nghĩa*

Cây là một kiểu dữ liệu trừu tượng gồm một tập hữu hạn các nút, giữa các nút có một quan hệ phân cấp gọi là quan hệ "cha-con". Có một nút đặc biệt gọi là gốc (*root*).

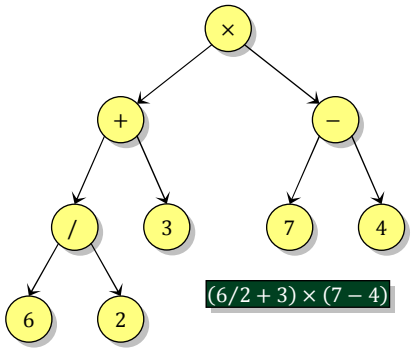
Có thể định nghĩa cây bằng cách đệ quy như sau:

- Một nút là một cây, nút đó cũng là gốc của cây ấy
- Nếu $r$ là một nút và $r_1, r_2, \dots, r_k$ lần lượt là gốc của các cây $T_1, T_2, \dots, T_k$, thì ta có thể xây dựng một cây mới $T$ bằng cách cho nút $r$ trở thành cha của các nút $r_1, r_2, \dots, r_k$. Cây $T$ này nút gốc là $r$ còn các cây $T_1, T_2, \dots, T_k$ trở thành các cây con hay nhánh con (subtree) của nút gốc.

Để tiện, người ta còn cho phép tồn tại một cây không có nút nào mà ta gọi là cây rỗng (null tree), ký hiệu Λ.

Một vài hình ảnh của cấu trúc cây:

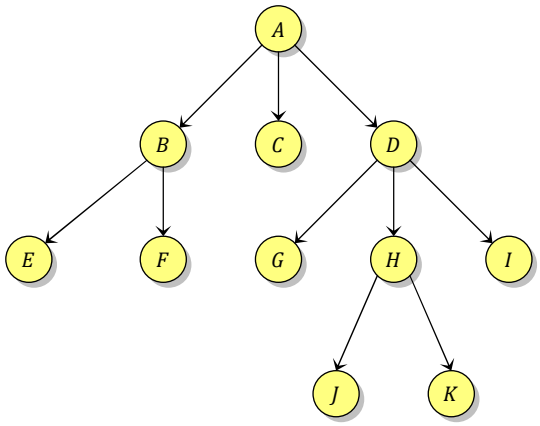
- Mục lục của một cuốn sách với phần, chương, bài, mục v.v… có cấu trúc của cây
- Cấu trúc thư mục trên đĩa cũng có cấu trúc cây, thư mục gốc có thể coi là gốc của cây đó với các cây con là các thư mục con (sub-directories) và tệp (files) nằm trên thư mục gốc.
- Gia phả của một họ tộc cũng có cấu trúc cây.
- Một biểu thức số học gồm các phép toán cộng, trừ, nhân, chia cũng có thể lưu trữ trong một cây mà các toán hạng được lưu trữ ở các nút lá, các toán tử được lưu trữ ở các nút nhánh, mỗi nhánh là một biểu thức con (h.1.8).




***Hình 1.8. Cây biểu diễn biểu thức***

## *3.2. Các khái niệm cơ bản*

Nếu $(r_1, r_2, \dots, r_k)$ là dãy các nút trên cây sao cho $r_i$ là nút cha của nút $r_{i+1}$ với $\forall i: 1 \leq i < k$, thì dãy này được gọi là một *đường đi* (*path*) từ $r_1$ tới $r_k$. Chiều dài của đường đi bằng số nút trên đường đi trừ đi 1. Quy ước rằng có đường đi độ dài 0 từ một nút đến chính nó. Như cây ở hình 1.9, $(A, B, F)$ là đường đi độ dài 2, $(A, D, H, K)$ là đường đi độ dài 3.




***Hình 1.9. Cây***

Nếu có một đường đi độ dài khác 0 từ nút $a$ tới nút $d$ thì nút $a$ gọi là *tiền bối* (ancestor) của nút $d$ và nút $d$ được gọi là *hậu duệ* (descendant) của nút $a$. Như cây ở hình 1.9, nút $A$ là tiền bối của tất cả các nút trên cây, nút $H$ là hậu duệ của nút $A$ và nút $D$. Một số quy ước còn cho phép một nút là tiền bối cũng như hậu duệ của chính nút đó, trong trường hợp này, người ta có thêm khái niệm *tiền bối*

*thực sự* (*proper ancestor*) và *hậu duệ đích thực* (*proper descendant*) trùng với khái niệm tiền bối và hậu duệ mà ta đã định nghĩa.

Trong cây, chỉ duy nhất một nút không có tiền bối là nút gốc. Một nút không có hậu duệ gọi là *nút lá* (*leaf*) của cây, các nút không phải lá được gọi là *nút nhánh* (branch). Như cây ở hình 1.9, các nút $C, E, F, G, I, J, K$ là các nút lá.

*Độ cao* của một nút là độ dài đường đi dài nhất từ nút đó tới một nút lá hậu duệ của nó, độ cao của nút gốc gọi là *chiều cao* (*height*) của cây. Như cây ở hình 1.9, cây có chiều cao là 3 (đường đi $(A, D, H, J)$).

*Độ sâu* (*depth*) của một nút là độ dài đường đi duy nhất từ nút gốc tới nút đó. Như cây ở hình 1.9, nút $A$ có độ sâu là 0, nút $B, C, D$ có độ sâu là 1, nút $E, F, G, H, I$ có độ sâu là 2, và nút $J, K$ có độ sâu là 3. Có thể định nghĩa chiều cao của cây là độ sâu lớn nhất của các nút trong cây.

Một tập hợp các cây đôi một không có nút chung được gọi là *rừng* (*forest*), có thể coi tập các cây con của một nút là một rừng.

Những nút con của cùng một nút được gọi là *anh em* (*sibling*). Với một cây, nếu chúng ta có tính đến thứ tự anh em thì cây đó gọi là *cây có thứ tự* (*ordered tree*), còn nếu chúng ta không quan tâm tới thứ tự anh em thì cây đó gọi là *cây không có thứ tự* (*unordered tree*).

## *3.3. Biểu diễn cây tổng quát*

Trong thực tế, có một số ứng dụng đòi hỏi một cấu trúc dữ liệu dạng cây nhưng không có ràng buộc gì về số con của một nút trên cây, ví dụ như cấu trúc thư mục trên đĩa hay hệ thống đề mục của một cuốn sách. Khi đó, ta phải tìm cách mô tả một cách khoa học cấu trúc dữ liệu dạng cây tổng quát. Giả sử $TElement$ là kiểu dữ liệu của các phần tử chứa trong mỗi nút của cây, khi đó ta có thể biểu diễn cây bằng một trong các cấu trúc dữ liệu sau:

### a) Biểu diễn bằng liên kết tới nút cha

Với $T$ là một cây, trong đó các nút được đánh số từ 1 tới $n$, khi đó ta có thể gán cho mỗi nút $i$ một nhãn $parent[i]$ là số hiệu nút cha của nút $i$. Nếu nút $i$ là nút gốc, thì $parent[i]$ được gán giá trị 0. Cách biểu diễn này có thể cài đặt bằng một

mảng các nút, mỗi nút là một bản ghi bên trong chứa giá trị lưu tại nút ($info$) và nhãn $parent$.

```
const max = ...; //Dung lượng cực đại
type
  TNode = record
    info: TElement;
    parent: Integer;
  end;
  TTree = array[1..max] of TNode;
var Tree: TTree;
```

Trong cách biểu diễn này, nếu chúng ta cần biết nút cha của một nút thì chỉ cần truy xuất trường *parent* của nút đó. Tuy nhiên nếu ta cần liệt kê tất cả các nút con của một nút thì không có cách nào khác là phải duyệt toàn bộ danh sách nút và kiểm tra trường *parent*. Thực hiện việc này mất thời gian $\Theta(n)$ với mỗi nút.

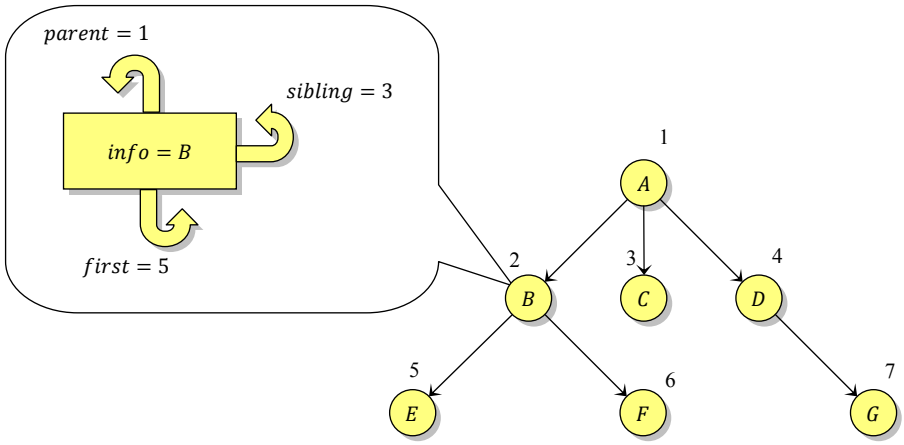
### b) Biểu diễn bằng cấu trúc liên kết

Trong cách biểu diễn này, ta sắp xếp các nút con của mỗi nút theo một thứ tự nào đó. Mỗi nút của cây là một bản ghi gồm 4 trường:

- Trường $info$: Chứa giá trị lưu trong nút
- Trường $parent$: Chứa con trỏ liên kết tới nút cha, tức là chứa một thông tin đủ để biết nút cha của nút đang xét là nút nào. Trong trường hợp nút đang xét là gốc (không có nút cha), trường $parent$ được gán một giá trị đặc biệt ($nil$).
- Trường $first$: Chứa liên kết (con trỏ) tới nút con đầu tiên (con cả) của nút đang xét, trong trường hợp nút đang xét là nút lá (không có nút con), trường này được gán một giá trị đặc biệt ($nil$).
- Trường $sibling$: Chứa liên kết (con trỏ) tới nút em kế cận (nút cùng cha với nút đang xét, khi sắp thứ tự các nút con thì nút $sibling$ đứng liền sau nút đang xét). Trong trường hợp nút đang xét không có nút em, trường này được gán một giá trị đặc biệt ($nil$).

```
type
  PNode = ^TNode;
  TNode = record
    info: TElement;
```

```
    parent, first, sibling: PNode;
end;
```




***Hình 1.10. Cấu trúc nút của cây tổng quát***

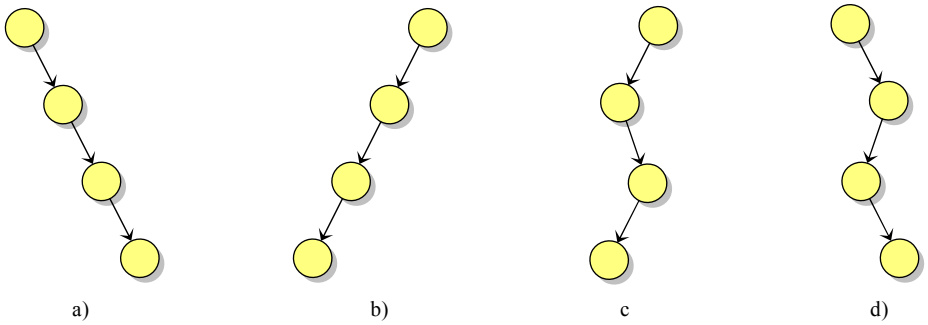
Trong các biểu diễn này, từ một nút $r$ bất kỳ, ta có thể đi theo liên kết $first$ để đến nút con đầu tiên, nút này chính là chốt của một danh sách nối đơn các nút con: Từ nút $first$, đi theo liên kết $sibling$, ta có thể duyệt tất cả các nút con của nút $r$.

Trong trường hợp phải thực hiện nhiều lần phép chèn/xóa một cây con, người ta có thể biểu diễn danh sách các nút con của một nút dưới dạng danh sách móc nối kép để việc chèn/xóa được thực hiện hiệu quả hơn, khi đó thay vì trường liên kết đơn $sibling$, mỗi nút sẽ có hai trường $prev$ và $next$ chứa liên kết tới nút anh liền trước và em liền sau của một nút.

## *3.4. Cây nhị phân*

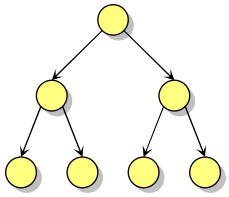
*Cây nhị phân* (*binary tree*) là một dạng quan trọng của cấu trúc cây. Nó có đặc điểm là mọi nút trên cây chỉ có tối đa hai nhánh con. Với một nút thì người ta cũng phân biệt cây con trái và cây con phải của nút đó, tức là cây nhị phân là cây có thứ tự.

## a) Một số dạng đặc biệt của cây nhị phân



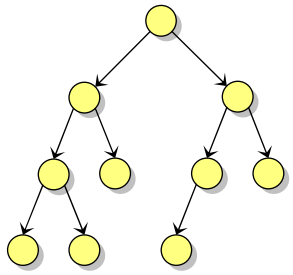

***Hình 1.11. Cây nhị phân suy biến***

Các cây nhị phân trong hình 1.11 được gọi là *cây nhị phân suy biến* (degenerate binary tree), trong cây nhị phân suy biến, các nút không phải lá chỉ có đúng một cây con. Cây a) được gọi là cây lệch phải, cây b) được gọi là cây lệch trái, cây c) và d) được gọi là cây zíc-zắc.



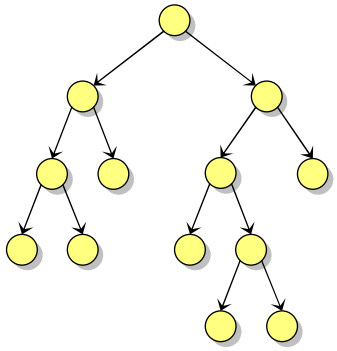
***Hình 1.11. Cây nhị phân hoàn chỉnh***

Cây trong hình 1.11 được gọi là *cây nhị phân hoàn chỉnh* (*complete binary tree*). Cây nhị phân hoàn chỉnh có mọi nút lá nằm ở cùng một độ sâu và mọi nút nhánh đều có hai nhánh con. Số nút ở độ sâu $h$ của cây nhị phân hoàn chỉnh là $2^h$. Tổng số nút của cây nhị phân hoàn chỉnh độ cao $h$ là $2^{h+1} - 1$



***Hình 1.12. Cây nhị phân gần hoàn chỉnh***

Cây trong hình 1.12 được gọi là *cây nhị phân gần hoàn chỉnh* (*nearly complete binary tree*). Một cây nhị phân độ cao $h$ được gọi là cây nhị phân gần hoàn chỉnh nếu ta bỏ đi mọi nút ở độ sâu $h$ thì được một cây nhị phân hoàn chỉnh. Cây nhị phân hoàn chỉnh hiển nhiên là cây nhị phân gần hoàn chỉnh.



***Hình 1.13. Cây nhị phân đầy đủ***

Cây trong hình 1.13 được gọi là *cây nhị phân đầy đủ* (*full binary tree*). Cây nhị phân đầy đủ là cây nhị phân mà mọi nút nhánh của nó đều có hai nút con.

Dễ dàng chứng minh được những tính chất sau:

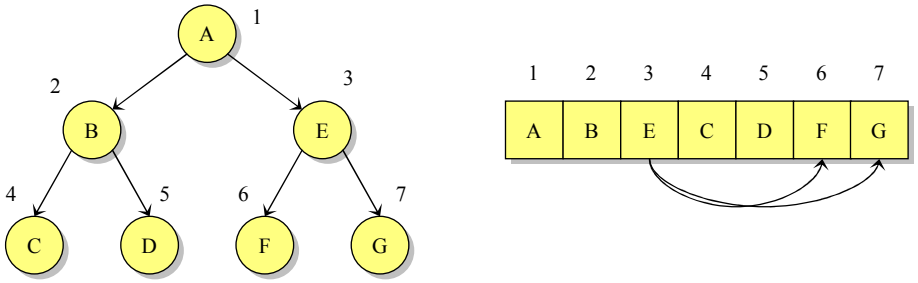
- Trong các cây nhị phân có cùng số lượng nút như nhau thì cây nhị phân suy biến có chiều cao lớn nhất, còn cây nhị phân gần hoàn chỉnh thì có chiều cao nhỏ nhất.
- Số lượng tối đa các nút ở độ sâu $d$ của cây nhị phân là $2^d$
- Số lượng tối đa các nút trên một cây nhị phân có chiều cao $h$ là $2^{h+1} - 1$
- Cây nhị phân gần hoàn chỉnh có $n$ nút thì chiều cao của nó là $\lfloor \lg n \rfloor$.

### b) Biểu diễn cây nhị phân

Rõ ràng có thể sử dụng các cách biểu diễn cây tổng quát để biểu diễn cây nhị phân, nhưng dựa vào những đặc điểm riêng của cây nhị phân, chúng ta có thể có những cách biểu diễn hiệu quả hơn. Trong phần này chúng ta xét một số cách biểu diễn đặc thù cho cây nhị phân, tương tự như với đối với cây tổng quát, ta gọi $TElement$ là kiểu dữ liệu của các phần tử chứa trong các nút của cây nhị phân.

❑ ***Biểu diễn bằng mảng***

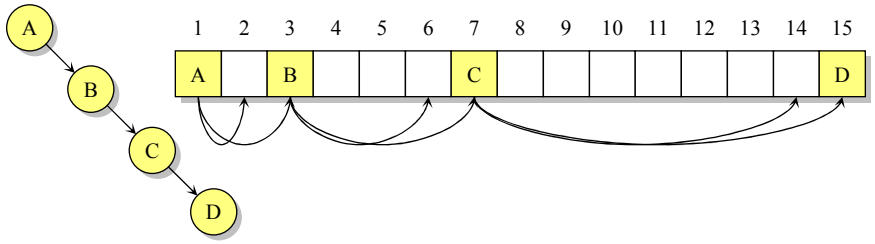
Một cây nhị phân hoàn chỉnh có $n$ nút thì có chiều cao là $\lfloor \lg n \rfloor$, tức là các nút sẽ nằm ở các độ sâu từ 0 tới $\lfloor \lg n \rfloor$, Khi đó ta có thể liệt kê tất cả các nút từ độ sâu 0 (nút gốc) tới độ sâu $\lfloor \lg n \rfloor$, sao cho với các nút cùng độ sâu thì thứ tự liệt kê là từ trái qua phải. Thứ tự liệt kê cho phép ta đánh số các nút từ 1 tới $n$ (h.1.14).




***Hình 1.14. Đánh số các nút của cây nhị phân hoàn chỉnh để biểu diễn bằng mảng***

Với cách đánh số này, hai con của nút thứ $i$ sẽ là các nút thứ $2i$ và $2i + 1$. Cha của nút thứ $i$ là nút thứ $\lfloor i/2 \rfloor$. Ta có thể lưu trữ cây bằng một mảng $info$ trong đó phần tử chứa trong nút thứ $i$ của cây được lưu trữ trong mảng bởi $info[i]$. Với cây nhị phân ở hình 1.14, ta có thể dùng mảng $info = (A, B, E, C, D, E, G)$ để chứa các giá trị trên cây.

Trong trường hợp cây nhị phân không hoàn chỉnh, ta có thể thêm vào một số nút giả để được cây nhị phân hoàn chỉnh. Khi biểu diễn bằng mảng thì những phần tử tương ứng với các nút giả sẽ được gán một giá trị đặc biệt. Chính vì lý do này nên việc biểu diễn cây nhị phân không hoàn chỉnh bằng mảng sẽ rất lãng phí bộ nhớ trong trường hợp phải thêm vào nhiều nút giả. Ví dụ ta cần tới một mảng 15 phần tử để lưu trữ cây nhị phân lệch phải chỉ gồm 4 nút (h.1.15).

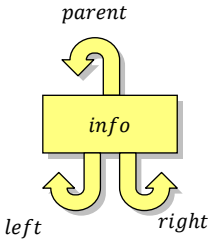



*Hình 1.15. Nhược điểm của phương pháp biểu diễn cây nhị phân bằng mảng*

### ❑ *Biểu diễn bằng cấu trúc liên kết.*

Khi biểu diễn cây nhị phân bằng cấu trúc liên kết, mỗi nút của cây là một bản ghi (record) gồm 4 trường:

- Trường $info$: Chứa giá trị lưu tại nút đó
- Trường $parent$: Chứa liên kết (con trỏ) tới nút cha, tức là chứa một thông tin đủ để biết nút cha của nút đó là nút nào, đối với nút gốc, trường này được gán một giá trị đặc biệt ($nil$).
- Trường $left$: Chứa liên kết (con trỏ) tới nút con trái, tức là chứa một thông tin đủ để biết nút con trái của nút đó là nút nào, trong trường hợp không có nút con trái, trường này được gán một giá trị đặc biệt ($nil$).
- Trường $right$: Chứa liên kết (con trỏ) tới nút con phải, tức là chứa một thông tin đủ để biết nút con phải của nút đó là nút nào, trong trường hợp không có nút con phải, trường này được gán một giá trị đặc biệt ($nil$).

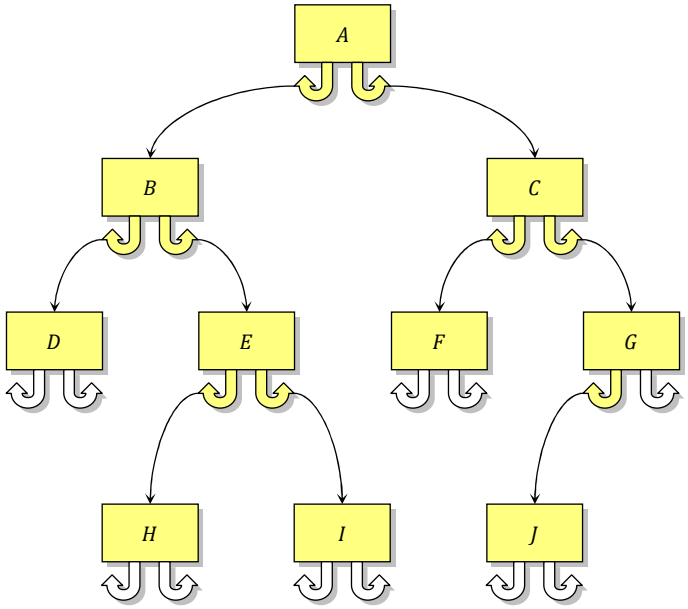



Đối với cây ta chỉ cần phải quan tâm giữ lại nút gốc ($root$), bởi từ nút gốc, đi theo các hướng liên kết $left$, $right$ ta có thể duyệt mọi nút khác.

```
type
  PNode = ^TNode;  //Kiểu con trỏ tới một nút
  TNode = record  //Cấu trúc biến động chứa thông tin trong một nút
    info: TElement;
    left, right: PNode
  end;
var root: PNode;  //Con trỏ tới nút gốc
```

Trong trường hợp biết rõ giới hạn về số nút của cây, ta có thể lưu trữ các nút trong một mảng, và dùng chỉ số mảng như liên kết tới một nút:

```
const max = ...; //Dung lượng cực đại
type
  TNode = record //Cấu trúc biến động chứa thông tin trong một nút
    info: TElement;
    left, right: Integer; //Chỉ số của nút con trái và nút con phải
  end;
  TTree = array[1..max] of TNode;
var
  Tree: TTree;
  root: Integer; //Chỉ số của nút gốc
```




*Hình 1.16. Biểu diễn cây nhị phân bằng cấu trúc liên kết*

### c) Phép duyệt cây nhị phân

Phép xử lý các nút trên cây mà ta gọi chung là phép *thăm* (*visit*) các nút một cách hệ thống sao cho mỗi nút chỉ được thăm một lần gọi là phép duyệt cây.

Giả sử rằng cấu trúc một nút của cây được đặc tả như sau:

```
type
```

```
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record //Cấu trúc biến động chứa thông tin trong một nút
    info: TElement;
    left, right: PNode
  end;
var root: PNode; //Con trỏ tới nút gốc
```

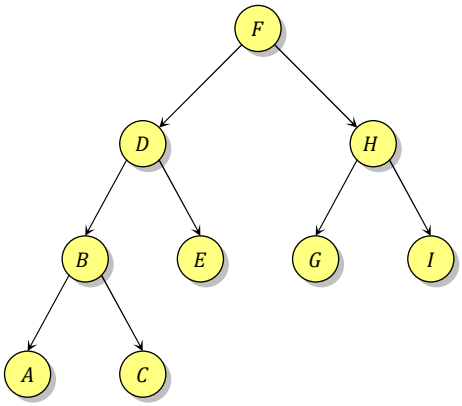
Quy ước rằng nếu như một nút không có nút con trái (hoặc nút con phải) thì liên kết $left$ ($right$) của nút đó được liên kết thẳng tới một nút đặc biệt mà ta gọi là $nil$, nếu cây rỗng thì nút gốc của cây đó cũng được gán bằng $nil$. Khi đó có ba cách duyệt cây hay được sử dụng:

❑ ***Duyệt theo thứ tự trước***

Trong phép duyệt *theo thứ tự trước* (*preorder traversal*) thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê trước tất cả các giá trị lưu trong hai nhánh con của nó, có thể mô tả bằng thủ tục đệ quy sau:

```
procedure Visit(node: PNode); //Duyệt nhánh cây gốc node^
begin
  if node ≠ nil then
    begin
      Output ← node^.info;
      Visit(node^.left);
      Visit(node^.right);
    end;
end;
```

Quá trình duyệt theo thứ tự trước bắt đầu bằng lời gọi $Visit(Root)$.




***Hình 1.17***

Hình 1.17 là một cây nhị phân có 9 nút. Nếu ta duyệt cây này theo thứ tự trước thì quá trình duyệt theo thứ tự trước sẽ lần lượt liệt kê các giá trị:

$$F, D, B, A, C, E, H, G, I$$

### ❑ *Duyệt theo thứ tự giữa*

Trong phép duyệt theo thứ tự giữa (*inorder traversal*) thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê sau tất cả các giá trị lưu ở nút con trái và được liệt kê trước tất cả các giá trị lưu ở nút con phải của nút đó, có thể mô tả bằng thủ tục đệ quy sau:

```
procedure Visit(node: PNode); //Duyệt nhánh cây gốc node^
begin
  if node ≠ nil then
    begin
      Visit(node^.left);
      Output ← node^.info;
      Visit(node^.right);
    end;
end;
```

Quá trình duyệt theo thứ tự giữa cũng bắt đầu bằng lời gọi $Visit(Root)$.

Nếu ta duyệt cây ở hình 1.17 theo thứ tự giữa thì quá trình duyệt sẽ liệt kê lần lượt các giá trị:

$$A, B, C, D, E, F, G, H, I$$

❑ ***Duyệt theo thứ tự sau***

Trong phép duyệt theo thứ tự sau thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê sau tất cả các giá trị lưu trong hai nhánh con của nó, có thể mô tả bằng thủ tục đệ quy sau:

```
procedure Visit(node: PNode); //Duyệt nhánh cây gốc node^
begin
  if node ≠ nil then
    begin
      Visit(node^.left);
      Visit(node^.right);
      Output ← node^.info;
    end;
end;
```

Quá trình duyệt theo thứ tự sau cũng bắt đầu bằng lời gọi $Visit(Root)$.

Cũng với cây ở hình 1.17, nếu ta duyệt theo thứ tự sau thì các giá trị sẽ lần lượt được liệt kê theo thứ tự:

$$A, C, B, E, D, G, I, H, F$$

## *3.5. Cây k-phân*

Cây $k$-phân là một dạng cấu trúc cây mà mỗi nút trên cây có tối đa $k$ nút con (có tính đến thứ tự của các nút con).

Cũng tương tự như việc biểu diễn cây nhị phân, người ta có thể thêm vào cây $k$-phân một số nút giả để cho mỗi nút nhánh của cây $k$-phân đều có đúng $k$ nút con, các nút con được xếp thứ tự từ nút con thứ nhất tới nút con thứ $k$, sau đó đánh số các nút trên cây $k$-phân bắt đầu từ 0 trở đi, bắt đầu từ mức 1, hết mức này đến mức khác và từ "trái qua phải" ở mỗi mức.

Theo cách đánh số này, nút con thứ $j$ của nút $i$ sẽ là: $ki + j$. Nếu $i$ không phải là nút gốc ($i > 0$) thì nút cha của nút i là nút $\lfloor (i-1)/k \rfloor$. Ta có thể dùng một mảng $Info$ đánh số từ 0 để lưu các giá trị trên các nút: Giá trị tại nút thứ $i$ được lưu trữ ở phần tử $Info[i]$. Đây là cơ chế biểu diễn cây $k$-phân bằng mảng.

Cây $k$-phân cũng có thể biểu diễn bằng cấu trúc liên kết với cấu trúc dữ liệu cho mỗi nút của cây là một bản ghi (record) gồm 3 trường:

- Trường $info$: Chứa giá trị lưu trong nút đó.
- Trường $parent$: Chứa liên kết (con trỏ) tới nút cha, tức là chứa một thông tin đủ để biết nút cha của nút đó là nút nào, đối với nút gốc, trường này được gán một giá trị đặc biệt ($nil$).
- Trường $links$: Là một mảng gồm $k$ phần tử, phần tử thứ $i$ chứa liên kết (con trỏ) tới nút con thứ $i$, trong trường hợp không có nút con thứ $i$ thì $links[i]$ được gán một giá trị đặc biệt ($nil$).

Đối với cây $k$-phân, ta cũng chỉ cần giữ lại nút gốc, bởi từ nút gốc, đi theo các hướng liên kết có thể đi tới mọi nút khác.

## Bài tập

**1.7.** Xét hai nút $x, y$ trên một cây nhị phân, ta nói nút $x$ nằm bên trái nút $y$ (nút $y$ nằm bên phải nút $x$) nếu:

- Hoặc nút $x$ nằm trong nhánh con trái của nút $y$
- Hoặc nút $y$ nằm trong nhánh con phải của nút $x$
- Hoặc tồn tại một nút $z$ sao cho $x$ nằm trong nhánh con trái và $y$ nằm trong nhánh con phải của nút $z$

Chỉ ra rằng với hai nút $x, y$ bất kỳ trên một cây nhị phân ($x \neq y$) chỉ có đúng một trong bốn mệnh đề sau là đúng:

- $x$ nằm bên trái $y$
- $x$ nằm bên phải $y$
- $x$ là tiền bối thực sự của $y$
- $y$ là tiền bối thực sự của $x$

**1.8.** Với mỗi nút $x$ trên cây nhị phân $T$, giả sử rằng ta biết được các giá trị $Preorder[x]$, $Inorder[x]$ và $Postorder[x]$ lần lượt là thứ tự duyệt trước, giữa, sau của $x$. Tìm cách chỉ dựa vào các giá trị này để kiểm tra hai nút có quan hệ tiền bối-hậu duệ hay không.

**1.9.** Bậc (degree) của một nút là số nút con của nó. Chứng minh rằng trên cây nhị phân, số lá nhiều hơn số nút bậc 2 đúng một nút.

**1.10.** Chỉ ra rằng cấu trúc của một cây nhị phân có thể khôi phục một cách đơn định nếu ta biết được thứ tự duyệt trước và giữa của các nút. Tương tự như vậy, cấu trúc cây có thể khôi phục nếu ta biết được thứ tự duyệt sau và giữa của các nút.

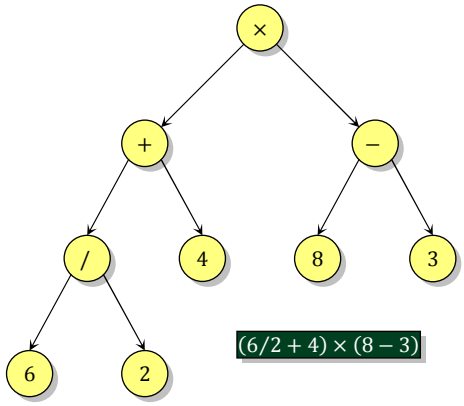
**1.11.** Tìm ví dụ về hai cây nhị phân khác nhau nhưng có thứ tự trước của các nút giống nhau và thứ tự sau của các nút cũng giống nhau trên hai cây.

## 4. Ký pháp tiền tố, trung tố và hậu tố

Để kết thúc chương này, chúng ta nói tới một ứng dụng của ngăn xếp và cây nhị phân: Bài toán phân tích và tính giá trị biểu thức.

### *4.1. Biểu thức dưới dạng cây nhị phân*

Chúng ta có thể biểu diễn các biểu thức số học gồm các phép toán cộng, trừ, nhân, chia bằng một cây nhị phân đầy đủ, trong đó các nút lá biểu thị các toán hạng (hằng, biến), các nút không phải là lá biểu thị các toán tử (phép toán số học chẳng hạn). Mỗi phép toán trong một nút sẽ tác động lên hai biểu thức con nằm ở cây con bên trái và cây con bên phải của nút đó. Ví dụ: Biểu thức $(6/2 + 4) \times (8 - 3)$ được biểu diễn trong cây ở hình 1.18.




***Hình 1.18. Cây biểu diễn biểu thức***

### *4.2. Các ký pháp cho cùng một biểu thức*

Với cây nhị phân biểu diễn biểu thức trong hình 4.1,

- Nếu duyệt theo thứ tự trước, ta sẽ được $\times\ +\ /\ 6\ 2\ 4\ -\ 8\ 3$, đây là *dạng tiền tố* (*prefix*) của biểu thức. Trong ký pháp này, toán tử được viết trước hai toán hạng tương ứng, người ta còn gọi ký pháp này là ký pháp Ba lan.
- Nếu duyệt theo thứ tự giữa, ta sẽ được $6\ /\ 2\ +\ 4\ \times\ 8\ -\ 3$. Ký pháp này bị nhập nhằng vì thiếu dấu ngoặc. Nếu thêm vào thủ tục duyệt một cơ chế bổ sung các cặp dấu ngoặc vào mỗi biểu thức con, ta sẽ thu được biểu thức $\big((6/2)+4\big)\times(8-3)\big)$. Ký pháp này gọi là *dạng trung tố* (*infix*) của một biểu thức (Thực ra chỉ cần thêm các dấu ngoặc đủ để tránh sự mập mờ mà thôi, không nhất thiết phải thêm vào đầy đủ các cặp dấu ngoặc).
- Nếu duyệt theo thứ tự sau, ta sẽ được $6\ 2\ /\ 4\ +\ 8\ 3\ -\ \times$, đây là *dạng hậu tố* (*postfix*) của biểu thức. Trong ký pháp này toán tử được viết sau hai toán hạng, người ta còn gọi ký pháp này là *ký pháp nghịch đảo Balan* (*Reverse Polish Notation - RPN*)

Chỉ có dạng trung tố mới cần có dấu ngoặc, dạng tiền tố và hậu tố không cần phải có dấu ngoặc. Chúng ta sẽ thảo luận về tính đơn định của dạng tiền tố và hậu tố trong phần sau.

### *4.3. Cách tính giá trị biểu thức*

Có một vấn đề cần lưu ý là khi máy tính giá trị một biểu thức số học gồm các toán tử hai ngôi (toán tử gồm hai toán hạng như $+,-,\times,/$) thì máy chỉ thực hiện được phép toán đó với hai toán hạng. Nếu biểu thức phức tạp thì máy phải chia nhỏ và tính riêng từng biểu thức trung gian, sau đó mới lấy giá trị tìm được để tính tiếp*. Ví dụ như biểu thức $1+2+4$ máy sẽ phải tính $1+2$ trước được kết quả là 3 sau đó mới đem 3 cộng với 4 chứ không thể thực hiện phép cộng một lúc ba số được.

Khi lưu trữ biểu thức dưới dạng cây nhị phân thì ta có thể coi mỗi nhánh con của cây đó biểu diễn một biểu thức trung gian mà máy cần tính trước khi tính biểu thức lớn. Như ví dụ trên, máy sẽ phải tính hai biểu thức $6/2+4$ và $8-3$ trước

* Thực ra đây là việc của trình dịch ngôn ngữ bậc cao, còn máy chỉ tính các phép toán với hai toán hạng theo trình tự của các lệnh được phân tích ra.

khi làm phép tính nhân cuối cùng. Để tính biểu thức $6/2 + 4$ thì máy lại phải tính biểu thức $6/2$ trước khi đem cộng với 4.

Vậy để tính một biểu thức lưu trữ trong một nhánh cây nhị phân gốc $r$, máy sẽ làm giống như hàm đệ quy sau:

```
function Calculate(r: Nút): Giá trị;
//Tính biểu thức con trong nhánh cây gốc r
begin
  if «nút r chứa một toán hạng» then
    Result := «Giá trị chứa trong nút r»
  else //Nút r chứa một toán tử ♦
    begin
      x := Calculate(nút con trái của r);
      y := Calculate(nút con phải của r);
      Result := x ♦ y;
    end;
end;
```

(Trong trường hợp lập trình trên các hệ thống song song, việc tính giá trị biểu thức ở cây con trái và cây con phải có thể tiến hành đồng thời làm giảm đáng kể thời gian tính toán biểu thức).

### *4.4. Tính giá trị biểu thức hậu tố*

Để ý rằng khi tính toán biểu thức, máy sẽ phải quan tâm tới việc tính biểu thức ở hai nhánh con trước, rồi mới xét đến toán tử ở nút gốc. Điều đó làm ta nghĩ tới phép duyệt cây theo thứ tự sau và ký pháp hậu tố. Năm 1920, nhà lô-gic học người Balan Jan Łukasiewicz đã chứng minh rằng biểu thức hậu tố không cần phải có dấu ngoặc vẫn có thể tính được một cách đúng đắn bằng cách đọc lần lượt biểu thức từ trái qua phải và dùng một Stack để lưu các kết quả trung gian:

- Bước 1: Khởi tạo một ngăn xếp rỗng
- Bước 2: Đọc lần lượt các phần tử của biểu thức RPN từ trái qua phải (phần tử này có thể là hằng, biến hay toán tử) với mỗi phần tử đó:
  - Nếu phần tử này là một toán hạng thì đẩy giá trị của nó vào ngăn xếp.
  - Nếu phần tử này là một toán tử ♦, ta lấy từ ngăn xếp ra hai giá trị ($y$ và $x$) sau đó áp dụng toán tử ♦ đó vào hai giá trị vừa lấy ra, đẩy kết quả tìm được ($x$♦$y$) vào ngăn xếp (ra hai vào một).

- Bước 3: Sau khi kết thúc bước 2 thì toàn bộ biểu thức đã được đọc xong, trong ngăn xếp chỉ còn duy nhất một phần tử, phần tử đó chính là giá trị của biểu thức.

Ví dụ: Tính biểu thức $6\,2\,/\,4\,+\,8\,3\,-\,\times$ tương ứng với biểu thức trung tố $(6/2+4)\times(8-3)$

| Đọc | Xử lý | Ngăn xếp |
|---|---|---|
| 6 | Đẩy vào 6 | 6 |
| 2 | Đẩy vào 2 | 6, 2 |
| / | Lấy ra 2 và 6, đẩy vào $6/2=3$ | 3 |
| 4 | Đẩy vào 4 | 3, 4 |
| + | Lấy ra 4 và 3, đẩy vào $3+4=7$ | 7 |
| 8 | Đẩy vào 8 | 7, 8 |
| 3 | Đẩy vào 3 | 7, 8, 3 |
| - | Lấy ra 3 và 8, đẩy vào $8-3=5$ | 7, 5 |
| × | Lấy ra 5 và 7, đẩy vào $7\times5=35$ | 35 |

Ta được kết quả là 35

Dưới đây ta sẽ viết một chương trình đơn giản tính giá trị biểu thức RPN.

**Input**

Biểu thức số học RPN, hai toán hạng liền nhau được phân tách bởi dấu cách. Các toán hạng là số thực, các toán tử là +, -, * hoặc /.

**Output**

Kết quả biểu thức

| Sample Input | Sample Output |
|---|---|
| 6 2 / 4 + 8 3 - * | 35.0000 |

Để đơn giản, chương trình không kiểm tra lỗi viết sai biểu thức RPN, việc đó chỉ là thao tác tỉ mỉ chứ không phức tạp lắm, chỉ cần xem lại thuật toán và cài thêm các lệnh bắt lỗi tại mỗi bước.

💻 RPNCALC.PAS ✓ Tính giá trị biểu thức RPN

```
{$MODE OBJFPC}
program CalculatingRPNExpression;
type
```

```
  TStackNode = record
  // Ngăn xếp được cài đặt bằng danh sách móc nối kiểu LIFO
    value: Real;
    link: Pointer;
  end;
  PStackNode = ^TStackNode;
var
  RPN: AnsiString;
  top: PStackNode;
procedure Push(const v: Real);
//Đẩy một toán hạng là số thực v vào ngăn xếp
var p: PStackNode;
begin
  New(p);
  p^.value := v;
  p^.link := top;
  top := p;
end;
function Pop: Real; //Lấy một toán hạng ra khỏi ngăn xếp
var p: PStackNode;
begin
  Result := top^.value;
  p := top^.link;
  Dispose(top);
  top := p;
end;
procedure ProcessToken(const token: AnsiString);
//Xử lý một phần tử trong biểu thức RPN
var
  x, y: Real;
  err: Integer;
begin
  if token[1] in ['+', '-', '*', '/'] then
  //Nếu phần tử token là toán tử
    begin //Lấy ra hai phần tử khỏi ngăn xếp, thực hiện toán tử và đẩy giá trị vào ngăn xếp
      y := Pop;
      x := Pop;
      case token[1] of
        '+': Push(x + y);
```

```
        '-': Push(x - y);
        '*': Push(x * y);
        '/': Push(x / y);
      end;
    end
  else //Nếu phần tử token là toán hạng thì đẩy giá trị của nó vào ngăn xếp
    begin
      Val(token, x, err);
      Push(x);
    end;
end;
procedure Parsing; //Xử lý biểu thức RPN
var i, j: Integer;
begin
  j := 0; //j là vị trí đã xử lý xong
  for i := 1 to Length(RPN) do //Quét biểu thức từ trái sang phải
    if RPN[i] in [' ', '+', '-', '*', '/'] then
    //Nếu gặp toán tử hoặc dấu phân cách toán hạng
      begin
        if i > j + 1 then //Trước vị trí i có một toán hạng chưa xử lý
          ProcessToken(Copy(RPN, j + 1, i - j - 1));
          //Xử lý toán hạng đó
        if RPN[i] in ['+', '-', '*', '/'] then
        //Nếu vị trí i chứa toán tử
          ProcessToken(RPN[i]); //Xử lý toán tử đó
        j := i; //Đã xử lý xong đến vị trí i
      end;
  if j < Length(RPN) then
  //Trường hợp có một toán hạng còn sót lại (biểu thức chỉ có 1 toán hạng)
    ProcessToken(Copy(RPN, j + 1, Length(RPN) - j));
    //Xử lý nốt
end;
begin
  ReadLn(RPN); //Đọc biểu thức
  top := nil; //Khởi tạo ngăn xếp rỗng,
  Parsing; //Xử lý
  Write(Pop:0:4); //Lấy ra phần tử duy nhất còn lại trong ngăn xếp và in ra kết quả.
end.
```

### *4.5. Chuyển từ dạng trung tố sang hậu tố*

Có thể nói rằng việc tính toán biểu thức viết bằng ký pháp nghịch đảo Balan là khoa học hơn, máy móc và đơn giản hơn việc tính toán biểu thức viết bằng ký pháp trung tố. Chỉ riêng việc không phải xử lý dấu ngoặc đã cho ta thấy ưu điểm của ký pháp RPN. Chính vì lý do này, các chương trình dịch vẫn cho phép lập trình viên viết biểu thức trên ký pháp trung tố theo thói quen, nhưng trước khi dịch ra các lệnh máy thì tất cả các biểu thức đều được chuyển về dạng RPN. Vấn đề đặt ra là phải có một thuật toán chuyển biểu thức dưới dạng trung tố về dạng RPN một cách hiệu quả, dưới đây ta trình bày thuật toán đó:

Thuật toán sử dụng một ngăn xếp $Stack$ để chứa các toán tử và dấu ngoặc mở. Thủ tục $Push(v)$ để đẩy một phần tử vào $Stack$, hàm $Pop$ để lấy ra một phần tử từ $Stack$, hàm $Get$ để đọc giá trị phần tử nằm ở đỉnh $Stack$ mà không lấy phần tử đó ra. Ngoài ra mức độ ưu tiên của các toán tử được quy định bằng hàm $Priority$: Ưu tiên cao nhất là dấu nhân (*) và dấu chia (/) với mức ưu tiên là 2, tiếp theo là dấu cộng (+) dấu (-) với mức ưu tiên là 1, ưu tiên thấp nhất là dấu ngoặc mở với mức ưu tiên là 0.

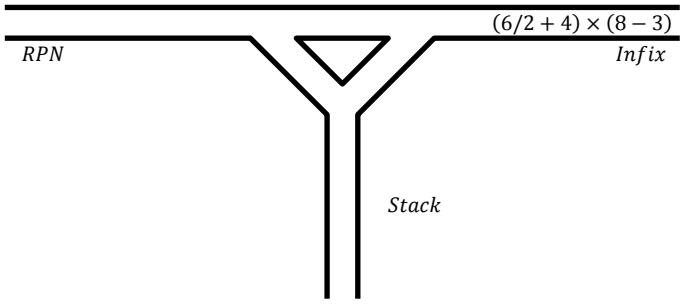
```
Stack := Ø;
for «phần tử token đọc được từ biểu thức trung tố» do
  case token of
//token có thể là toán hạng, toán tử, hoặc dấu ngoặc được đọc lần lượt theo thứ tự từ trái qua phải
    '(': Push(token); //Gặp dấu ngoặc mở thì đẩy vào ngăn xếp
    ')':
//Gặp dấu ngoặc đóng thì lấy ra và hiển thị các phần tử trong ngăn xếp cho tới khi lấy tới dấu ngoặc mở
      repeat
        x := Pop;
        if x ≠ '(' then Output ← x;
      until x = '(';
    '+', '-', '*', '/': //Gặp toán tử
     begin
//Chừng nào đỉnh ngăn xếp có phần tử với mức ưu tiên lớn hơn hay bằng token, lấy phần tử đó ra và hiển thị
       while (Stack ≠ Ø)
             and (Priority(token) ≤ Priority(Get)) do
         Output ← Pop;
       Push(token); //Đẩy toán tử token vào ngăn xếp
```

```
        end;
      else //Gặp toán hạng thì hiển thị luôn
        Output ← token;
    end;
//Khi đọc xong biểu thức, lấy ra và hiển thị tất cả các phần tử còn lại trong ngăn xếp
while Stack ≠ Ø do
  Output ← Pop
```

Ví dụ với biểu thức trung tố $(6/2+4)\times(8-3)$

| Đọc | Xử lý | Ngăn xếp | Output | Chú thích |
|---|---|---|---|---|
| ( | Đẩy "(" vào ngăn xếp | ( | | |
| 6 | Hiển thị | ( | 6 | |
| / | Đẩy "/" vào ngăn xếp | (/ | 6 | "/" > "(" |
| 2 | Hiển thị | (/ | 6 2 | |
| + | Lấy "/" khỏi ngăn xếp và hiển thị, đẩy "+" vào ngăn xếp | (+ | 6 2 / | "(" < "+" < "/" |
| 4 | Hiển thị | (+ | 6 2 / 4 | |
| ) | Lấy "+" và "(" khỏi ngăn xếp, hiển thị "+" | Ø | 6 2 / 4 + | |
| × | Đẩy "×" vào ngăn xếp | × | 6 2 / 4 + | |
| ( | Đẩy "(" vào ngăn xếp | ×( | 6 2 / 4 + | |
| 8 | Hiển thị | ×( | 6 2 / 4 + 8 | |
| - | Đẩy "-" vào ngăn xếp | ×(- | 6 2 / 4 + 8 | "-" > ")" |
| 3 | Hiển thị | ×(- | 6 2 / 4 + 8 3 | |
| ) | Lấy "-" và "(" khỏi ngăn xếp, hiển thị "-" | × | 6 2 / 4 + 8 3 - | |
| Hết | Lấy nốt "×" ra và hiển thị | Ø | 6 2 / 4 + 8 3 - × | |

Thuật toán này có tên là thuật toán "xếp toa tàu" (*shunting yards*) do Edsger Dijkstra đề xuất năm 1960. Tên gọi này xuất phát từ mô hình đường ray tàu hỏa:




***Hình 1.19. Mô hình "xếp toa tàu" của thuật toán chuyển từ dạng trung tố sang hậu tố***

Trong hình 1.19, mỗi toa tàu tương ứng với một phần tử trong biểu thức trung tố nằm ở "đường ray" $Infix$. Có ba phép chuyển toa tàu: Từ đường ray $Infix$ sang thẳng đường ray $RPN$, từ đường ray $Infix$ xuống đường ray $Stack$, hoặc từ đường ray $Stack$ lên đường ray $RPN$. Thuật toán chỉ đơn thuần dựa trên các luật chuyển mà theo các luật đó ta sẽ chuyển được tất cả các toa tàu sang đường ray $RPN$ để được một thứ tự tương ứng với biểu thức hậu tố[*] (ví dụ toán hạng ở $Infix$ sẽ được chuyển thẳng sang $RPN$ hay dấu "(" ở $Infix$ sẽ được chuyển thẳng xuống $Stack$).

Dưới đây là chương trình chuyển biểu thức viết ở dạng trung tố sang dạng RPN:

**Input**

Biểu thức trung tố

**Output**

Biểu thức hậu tố

| Sample Input | Sample Output |
|---|---|
| (6 / 2 + 4) * (8 - 3) | 6 2 / 4 + 8 3 - * |

💻 INFIX2RPN.PAS ✓ Chuyển từ dạng trung tố sang hậu tố

```
{$MODE OBJFPC}
```

[*] Thực ra thì còn thao tác loại bỏ các dấu ngoặc trong biểu thức RPN nữa, nhưng điều này không quan trọng, dấu ngoặc là thừa trong biểu thức RPN vì như đã nói về phương pháp tính: biểu thức RPN có thể tính đơn định mà không cần các dấu ngoặc.

```
program ConvertInfixToRPN;
type
  TStackNode = record
  // Ngăn xếp được cài đặt bằng danh sách móc nối kiểu LIFO
    value: AnsiChar;
    link: Pointer;
  end;
  PStackNode = ^TStackNode;
var
  Infix: AnsiString;
  top: PStackNode;
procedure Push(const c: AnsiChar); //Đẩy một phần tử v vào ngăn xếp
var p: PStackNode;
begin
  New(p);
  p^.value := c;
  p^.link := top;
  top := p;
end;
function Pop: AnsiChar; //Lấy một phần tử ra khỏi ngăn xếp
var p: PStackNode;
begin
  Result := top^.value;
  p := top^.link;
  Dispose(top);
  top := p;
end;
function Get: AnsiChar; //Đọc phần tử ở đỉnh ngăn xếp
begin
  Result := top^.value;
end;
function Priority(c: Char): Integer; //Mức ưu tiên của các toán tử
begin
  case c of
    '*', '/': Result := 2;
    '+', '-': Result := 1;
    '(': Result := 0;
  end;
```

```
end;
//Xử lý một phần tử đọc được từ biểu thức trung tố
procedure ProcessToken(const token: AnsiString);
var
  x: AnsiChar;
  Opt: AnsiChar;
begin
  Opt := token[1];
  case Opt of
    '(': Push(Opt); //token là dấu ngoặc mở
    ')': //token là dấu ngoặc đóng
      repeat
        x := Pop;
        if x <> '(' then Write(x, ' ')
        else Break;
      until False;
    '+', '-', '*', '/': //token là toán tử
      begin
        while (top <> nil)
              and (Priority(Opt) <= Priority(Get)) do
          Write(Pop, ' ');
        Push(Opt);
      end;
    else //token là toán hạng
      Write(token, ' ');
  end;
end;
procedure Parsing;
const Operators = ['(', ')', '+', '-', '*', '/'];
var i, j: Integer;
begin
  j := 0; //j là vị trí đã xử lý xong
  for i := 1 to Length(Infix) do
    if Infix[i] in Operators + [' '] then
    //Nếu gặp dấu ngoặc, toán tử hoặc dấu cách
      begin
        if i > j + 1 then //Trước vị trí i có một toán hạng chưa xử lý
          ProcessToken(Copy(Infix, j + 1, i - j - 1));
            //xử lý toán hạng đó
```

```
        if Infix[i] in Operators then
         //Nếu vị trí i chứa toán tử hoặc dấu ngoặc
          ProcessToken(Infix[i]); //Xử lý ký tự đó
        j := i; //Cập nhật, đã xử lý xong đến vị trí i
      end;
  if j < Length(Infix) then //Xử lý nốt toán hạng còn sót lại
   ProcessToken(Copy(Infix, j + 1, Length(Infix) - j));
   //Đọc hết biểu thức trung tố, lấy nốt các phần tử trong ngăn xếp ra và hiển thị
  while top <> nil do
    Write(Pop, ' ');
  WriteLn;
end;
begin
  ReadLn(Infix); //Nhập dữ liệu
  top := nil; //Khởi tạo ngăn xếp rỗng
  Parsing; //Đọc biểu thức trung tố và chuyển thành dạng RPN
end.
```

## *4.6. Xây dựng cây nhị phân biểu diễn biểu thức*

Ngay trong phần đầu tiên, chúng ta đã biết rằng các dạng biểu thức trung tố, tiền tố và hậu tố đều có thể được hình thành bằng cách duyệt cây nhị phân biểu diễn biểu thức đó theo các trật tự khác nhau. Vậy tại sao không xây dựng ngay cây nhị phân biểu diễn biểu thức đó rồi thực hiện các công việc tính toán ngay trên cây?. Khó khăn gặp phải chính là thuật toán xây dựng cây nhị phân trực tiếp từ dạng trung tố có thể kém hiệu quả, trong khi đó từ dạng hậu tố lại có thể khôi phục lại cây nhị phân biểu diễn biểu thức một cách rất đơn giản, gần giống như quá trình tính toán biểu thức hậu tố:

- Bước 1: Khởi tạo một ngăn xếp rỗng dùng để chứa các nút trên cây
- Bước 2: Đọc lần lượt các phần tử của biểu thức RPN từ trái qua phải (phần tử này có thể là hằng, biến hay toán tử) với mỗi phần tử đó:
  - Tạo ra một nút mới $z$ chứa phần tử mới đọc được
  - Nếu phần tử này là một toán tử, lấy từ ngăn xếp ra hai nút (theo thứ tự là $y$ và $x$), cho $x$ trở thành con trái và $y$ trở thành con phải của nút $z$
  - Đẩy nút $z$ vào ngăn xếp

- Bước 3: Sau khi kết thúc bước 2 thì toàn bộ biểu thức đã được đọc xong, trong ngăn xếp chỉ còn duy nhất một phần tử, phần tử đó chính là gốc của cây nhị phân biểu diễn biểu thức.

## Bài tập

**1.12.** Biểu thức có thể có dạng phức tạp hơn, chẳng hạn biểu thức bao gồm cả phép lấy số đối ($-x$ ), phép tính lũy thừa ($x^y$), hàm số với một hay nhiều biến số. Chúng ta có thể biểu diễn những biểu thức dạng này bằng một cây tổng quát và từ đó có thể chuyển biểu thức về dạng RPN để thực hiện tính toán. Hãy xây dựng thuật toán để chuyển biểu thức số học (dạng phức tạp) về dạng RPN và thuật toán tính giá trị biểu thức đó.

**1.13.** Viết chương trình chuyển biểu thức logic dạng trung tố sang dạng RPN. Ví dụ chuyển: $a$ and $b$ or $c$ and $d$ thành: $a\ b$ and $c\ d$ and or.

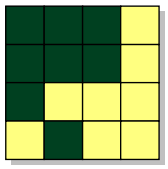
**1.14.** Chuyển các biểu thức sau đây ra dạng RPN

a) $A \times (B + C)$
b) $A + (B/C) + D$
c) $A \times (B + -C)$
d) $A - (B + C)^{D/E}$
e) $(A$ or $B)$ and $\big(C$ or $(D$ or not $E)\big)$
f) $(A = B)$ or $(C = D)$

**1.15.** Với một ảnh đen trắng hình vuông kích thước $2^n \times 2^n$, người ta dùng phương pháp sau để mã hóa ảnh:

- Nếu ảnh chỉ gồm toàn điểm đen thì ảnh đó có thể được mã hóa bằng xâu chỉ gồm một ký tự 'B'
- Nếu ảnh chỉ gồm toàn điểm trắng thì ảnh đó có thể được mã hóa bằng xâu chỉ gồm một ký tự 'W'
- Nếu $P, Q, R, S$ lần lượt là xâu mã hóa của bốn ảnh vuông kích thước bằng nhau thì $\&PQRS$ là xâu mã hóa của ảnh vuông tạo thành bằng cách đặt 4 ảnh vuông ban đầu theo sơ đồ:

$$\begin{matrix} P & Q \\ S & R \end{matrix}$$



Ví dụ "&B&BWWBW&BWBW" và "&&BBBB&BWWBW&BWBW" là hai xâu mã hóa của cùng một ảnh bên:

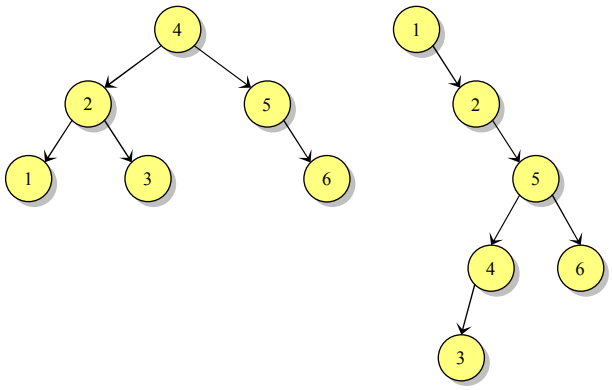
Bài toán đặt ra là cho số nguyên dương $n$ và hai xâu mã hóa của hai ảnh kích thước $2^n \times 2^n$. Hãy cho biết hai ảnh đó có khác nhau không và nếu chúng khác nhau hãy chỉ ra một vị trí có màu khác nhau trên hai ảnh.

# 5. Cây nhị phân tìm kiếm

## *5.1. Cấu trúc chung của cây nhị phân tìm kiếm*

*Cây nhị phân tìm kiếm* (*binary search tree-BST*) là một cây nhị phân, trong đó mỗi nút chứa một phần tử (khóa). Khóa chứa trong mỗi nút phải lớn hơn hay bằng mọi khóa trong nhánh con trái và nhỏ hơn hay bằng mọi khóa trong nhánh con phải.

Ở đây chúng ta giả sử rằng các khóa lưu trữ trong cây được lấy từ một tập hợp $S$ có quan hệ thứ tự toàn phần.




***Hình 1.20. Cây nhị phân tìm kiếm***

Có thể có nhiều cây nhị phân tìm kiếm biểu diễn cùng một bộ khóa. Hình 1.20 là ví dụ về hai cây nhị phân tìm kiếm biểu diễn cùng một bộ khóa $(1,2,3,4,5,6)$.

***Định lý 5.1***

Nếu duyệt cây nhị phân tìm kiếm theo thứ tự giữa, các khóa trên cây sẽ được liệt kê theo thứ tự không giảm (tăng dần).

***Chứng minh***

Ta chứng minh định lý bằng quy nạp: Rõ ràng định lý đúng với BST chỉ có một nút. Giả sử định lý đúng với mọi BST có ít hơn $n$ nút, xét một BST bất kỳ gồm $n$ nút, và ở nút gốc chứa khóa $k$, thuật toán duyệt cây theo thứ tự giữa trước hết sẽ liệt kê tất cả các khóa trong nhánh con trái theo thứ tự không giảm (giả thiết quy nạp), các khóa này đều $\leq k$ (tính chất của cây nhị phân tìm kiếm). Tiếp theo thuật toán sẽ liệt kê khóa $k$ của nút gốc, cuối cùng, lại theo giả thiết quy nạp, thuật toán sẽ liệt kê tất cả các khóa trong nhánh con phải theo thứ tự không giảm, tương tự như trên, các khóa trong nhánh con phải đều $\geq k$. Vậy tất cả $n$ khóa trên BST sẽ được liệt kê theo thứ tự không giảm, định lý đúng với mọi BST gồm $n$ nút. ĐPCM.

## *5.2. Các thao tác trên cây nhị phân tìm kiếm*

### a) Cấu trúc nút

Chúng ta sẽ biểu diễn BST bằng một cấu trúc liên kết các nút động và con trỏ liên kết. Mỗi nút trên BST sẽ là một bản ghi gồm 3 trường:

- Trường $key$: Chứa khóa lưu trong nút.
- Trường $parent$: Chứa liên kết (con trỏ) tới nút cha, nếu là nút gốc (không có nút cha) thì trường $parent$ được đặt bằng một con trỏ đặc biệt, ký hiệu $nilT$.
- Trường $left$: Chứa liên kết (con trỏ) tới nút con trái, nếu nút không có nhánh con trái thì trường $left$ được đặt bằng $nilT$.
- Trường $right$: Chứa liên kết (con trỏ) tới nút con phải, nếu nút không có nhánh con phải thì trường $right$ được đặt bằng $nilT$.

Nếu các khóa chứa trong nút có kiểu $TKey$ thì cấu trúc nút của BST có thể được khai báo như sau:

```
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record
    key: TKey;
    parent, left, right: PNode;
  end;
var
```

```
  sentinel: TNode;
  nilT: PNode; //Con trỏ tới nút đặt biệt
  root: PNode; //Con trỏ tới nút gốc
begin
  nilT := @sentinel;
  ...
end.
```

Các ngôn ngữ lập trình bậc cao thường cung cấp hằng con trỏ $nil$ (hay $null$) để gán cho các liên kết không tồn tại trong cấu trúc dữ liệu. Hằng con trỏ $nil$ chỉ được sử dụng để so sánh với các con trỏ khác, không được phép truy cập biến động $nil$^.

Trong cài đặt BST, chúng ta sử dụng con trỏ $nilT$ có công cụng tương tự như con trỏ $nil$: gán cho những liên kết không có thực. Chỉ có khác là con trỏ $nilT$ trỏ tới một biến $sentinel$ **có thực**, chỉ có điều các trường của $nilT$^ là **vô nghĩa** mà thôi. Chúng ta hy sinh một ô nhớ cho biến $sentinel = nilT$^ để đơn giản hóa các thao tác trên BST*.

### b) Khởi tạo cây rỗng

Trong cấu trúc BST khai báo ở trên, ta quy ước một cây rỗng là cây có gốc $Root = nilT$, phép khởi tạo một BST rỗng chỉ đơn giản là:

```
procedure MakeNull;
begin
  root := nilT;
end;
```

### c) Tìm khóa lớn nhất và nhỏ nhất

Theo Định lí 5.1, khóa nhỏ nhất trên BST nằm trong nút được thăm đầu tiên và khóa lớn nhất của BST nằm trong nút được thăm cuối cùng nếu ta duyệt cây theo thứ tự giữa. Như vậy nút chứa khóa nhỏ nhất (lớn nhất) của BST chính là nút cực trái (cực phải) của BST. Hàm $Minimum$ và $Maximum$ dưới đây lần

* Mục đích của biến này là để bớt đi thao tác kiểm tra con trỏ $p \neq nil$ trước khi truy cập nút $p$^.

lượt trả về nút chứa khóa nhỏ nhất và lớn nhất trong nhánh cây BST gốc $x$ (ở đây ta giả thiết rằng nhánh BST gốc $x$ khác rỗng: $x \neq nilT$)

```
function Minimum(x: PNode): PNode; //Khóa nhỏ nhất nằm ở nút cực trái
begin
  while x^.left ≠ nilT do //Đi sang nút con trái chừng nào vẫn còn đi được
    x := x^.left;
  Result := x;
end;
function Maximum(x: PNode): PNode; //Khóa lớn nhất nằm ở nút cực phải
begin
  while x^.right ≠ nilT do //Đi sang nút con phải chừng nào vẫn còn đi được
    x := x^.right;
  Result := x;
end;
```

### d) Tìm nút liền trước và nút liền sau

Đôi khi chúng ta phải tìm nút đứng liền trước và liền sau của một nút $x$ nếu duyệt cây BST theo thứ tự giữa. Trước hết ta xét viết hàm $Predecessor(x)$ trả về nút đứng liền trước nút $x$, xét hai trường hợp:

- Nếu $x$ có nhánh con trái thì trả về nút cực phải của nhánh con trái: $Result \coloneqq Maximum(x\text{^}.Left)$ .
- Nếu $x$ không có nhánh con trái thì từ $x$, ta đi dần lên phía gốc cây cho tới khi gặp một nút chứa $x$ trong nhánh con phải thì dừng lại và trả về nút đó.

```
function Predecessor(x: PNode): PNode;
begin
  if x^.left ≠ nilT then //x có nhánh con trái
    Result := Maximum(x^.left) //Trả về nút cực phải của cây con trái
  else
    repeat
      Result := x^.parent;
      //Nếu x là gốc hoặc x là nhánh con phải thì thoát ngay
      if (Result = nilT)
          or (x = Result^.right) then Break;
      x := Result; //Nếu không thì đi tiếp lên phía gốc
    until False;
end;
```

Hàm $Successor(x)$ trả về nút liền sau nút $x$ có cách làm tương tự nếu ta đổi vai trò $Left$ và $Right$, $Minimum$ và $Maximum$:

```
function Successor(x: PNode): PNode;
begin
  if x^.right ≠ nilT then //x có nhánh con phải
    Result := Minimum(x^.right) //Trả về nút cực trái của cây con phải
  else
    repeat
      Result := x^.parent;
      //Nếu x là gốc hoặc x là nhánh con trái thì thoát ngay
      if (Result = nilT)
          or (x = Result^.left) then Break;
      x := Result; //Đi tiếp lên phía gốc
    until False;
end;
```

### e) Tìm kiếm

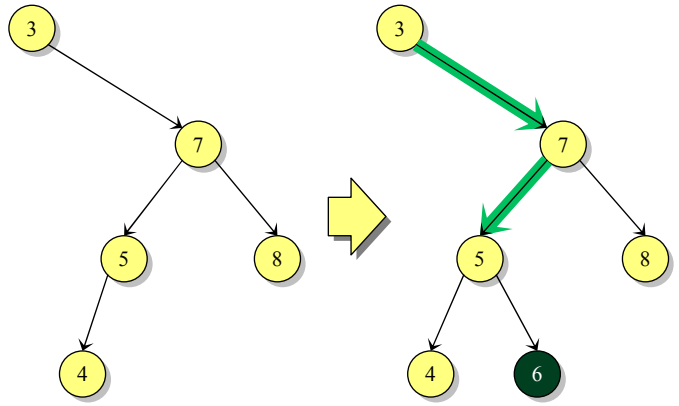
Phép tìm kiếm nhận vào một nút $x$ và một khóa $k$. Nếu khóa $k$ có trong nhánh BST gốc $x$ thì trả về một nút chứa khóa $k$, nếu không trả về $nilT$.

Phép tìm kiếm trên BST có thể cài đặt bằng hàm $Search$, hàm này được xây dựng dựa trên nguyên lý chia để trị: Nếu nút $x$ chứa khóa $= k$ thì hàm đơn giản trả về nút $x$, nếu không thì việc tìm kiếm sẽ được tiến hành tương tự trên cây con trái hoặc cây con phải tùy theo nút $x$ chứa khóa nhỏ hơn hay lớn hơn $k$:

```
//Hàm Search trả về nút chứa khóa k, trả về nilT nếu không tìm thấy khóa k trong nhánh gốc x
function Search(x: PNode; const k: TKey): PNode;
begin
  while (x ≠ nilT) and (x^.key ≠ k) do //Chừng nào chưa tìm thấy
    if k < x^.key then x := x^.left
    //k chắc chắn không nằm trong cây con phải, tìm trong cây con trái
    else x := x^.right;
    //k chắc chắn không nằm trong cây con trái, tìm trong cây con phải
  Result := x;
end;
```

### f) Chèn

Chèn một khóa $k$ vào BST tức là thêm một nút mới chứa khóa $k$ trên BST và móc nối nút đó vào BST sao cho vẫn đảm bảo cấu trúc của một BST. Phép chèn cũng được thực hiện dựa trên nguyên lý chia để trị: Bài toán chèn $k$ vào cây BST sẽ được quy về bài toán chèn $k$ vào cây con trái hay cây con phải, tùy theo khóa $k$ nhỏ hơn hay lớn hơn hoặc bằng khóa chứa trong nút gốc. Trường hợp cơ sở là $k$ được chèn vào một nhánh cây rỗng, khi đó ta chỉ việc tạo nút mới, móc nối nút mới vào nhánh rỗng này và đặt khóa $k$ vào nút mới đó.




***Hình 1.11. Cây nhị phân tìm kiếm trước và sau khi chèn khóa $v = 6$***

Trước tiên ta viết một thủ tục $SetLink(ParentNode, ChildNode, InLeft)$ để chỉnh lại các liên kết sao cho nút $ChildNode$ trở thành nút con của nút $ParentNode$:

```
procedure SetLink(ParentNode, ChildNode: PNode;
                                  InLeft: Boolean);
begin
  ChildNode^.parent := ParentNode;
  if InLeft then ParentNode^.left := ChildNode
  //InLeft = True: Cho ChildNode thành nút con trái của ParentNode
  else ParentNode^.right := ChildNode;
  //InLeft = False: Cho ChildNode thành nút con phải của ParentNode
end;
```

Khi đó thủ tục chèn một nút $NewNode^$ vào BST có thể viết như sau

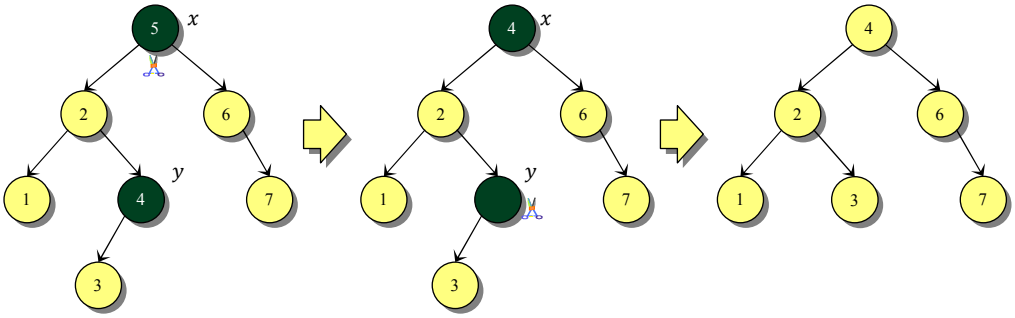
```
//Chèn k vào BST, trả về nút mới chứa k
```

```
function Insert(k: TKey): PNode;
var x, y: PNode;
begin
  y := nilT;
  x := root; //Bắt đầu từ gốc
  while x ≠ nilT do
    begin
      y := x;
      if k < x^.key then x := x^.left //Chèn vào nhánh trái
      else x := x^.right; //Chèn vào nhánh phải
    end;
  New(x); //Tạo nút mới chứa k
  x^.key := k;
  x^.left := nilT;
  x^.right := nilT;
  SetLink(y, x, k < y^.key); //Móc nối vào BST
  if root = nilT then root := x;
  //Cập nhật lại gốc nếu là nút đầu tiên được chèn vào
  Result := x;
end;
```

### g) Xóa

Việc xóa một nút $x$ trong BST thực chất là xóa đi khóa chứa trong nút $x$. Phép xóa được thực hiện như sau:

- Nếu $x$ có ít hơn hai nhánh con, ta lấy nút con (nếu có) của $x$ lên thay cho $x$ và xóa nút $x$.
- Nếu $x$ có hai nhánh con, ta xác định nút $y$ là nút cực phải của nhánh con trái (hoặc nút cực trái của cây con phải), đưa khóa chứa trong nút $y$ lên nút $x$ rồi xóa nút $y$. Chú ý rằng nút $y$ chắc chắn không có đủ hai nút con, việc xóa quy về trường hợp trên. (h.1.22)




*Hình 1.22. Xóa nút khỏi BST*

```
procedure Delete(x: PNode);
var y, z: PNode;
begin
  if (x^.left ≠ nilT) and (x^.right ≠ nilT) then
  //x có hai nhánh con
    begin
      y := Maximum(x^.left); //Tìm nút cực phải của cây con trái
      x^.key := y^.key; //Đưa khóa của nút y lên nút x
      x := y;
    end;
 //Vấn đề bây giờ là xóa nút x có nhiều nhất một nhánh con, xác định y là nút con (nếu có) của x
  if x^.left ≠ nilT then y := x^.left
  else y := x^.right;
  z := x^.parent; //z là cha của x
  //cho y làm con của z thay cho x
  SetLink(z, y, z^.left = x);
  if x = root then root := y;
  //Trường hợp nút x bị hủy là gốc thì cập nhật lại gốc là y
  Dispose(x); //Giải phóng bộ nhớ cấp cho nút x
end;
```

## h) Phép quay cây

Phép quay cây là một phép chỉnh lại cấu trúc liên kết trên BST, có hai loại: quay trái (*left rotation*) và quay phải (*right rotation*).

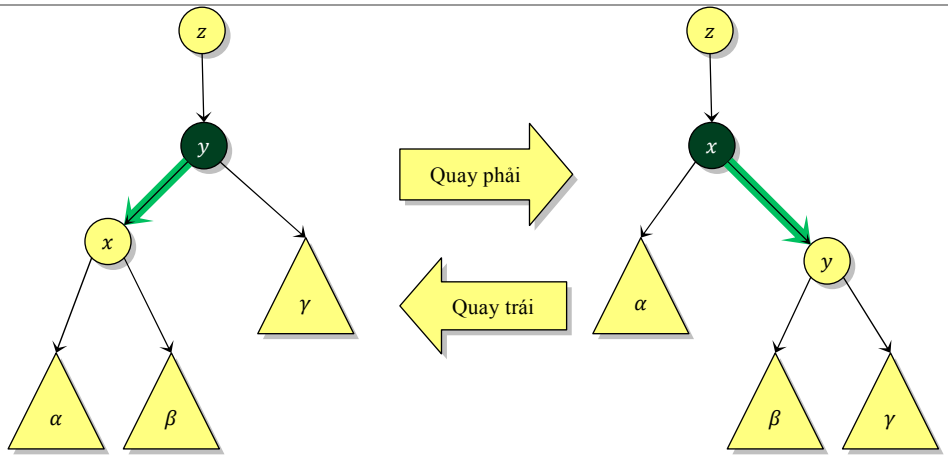
Khi ta thực hiện phép quay trái trên nút $x$, chúng ta giả thiết rằng nút con phải của $x$ là $y \neq nil$. Trên cây con gốc $x$, phép quay trái sẽ đưa $y$ lên làm gốc mới, $x$

trở thành nút con trái của $y$ và nút con trái của $y$ trở thành nút con phải của $x$. Phép quay này còn gọi là *quay theo liên kết* $x \xrightarrow{R} y$.

Ngược lại, phép quay phải thực hiện trên những cây con gốc $y$ mà nút con trái của $y$ là $x \neq nilT$. Sau phép quay phải, $x$ sẽ được đưa lên làm gốc nhánh, $y$ trở thành nút con phải của $x$ và nút con phải của $y$ trở thành nút con trái của $x$. Phép quay này còn gọi là *quay theo liên kết* $y \xrightarrow{L} x$.

Dễ thấy rằng sau phép quay, ràng buộc về quan hệ thứ tự của các khóa chứa trong cây vẫn đảm bảo cho cây mới là một BST.

Hình 5.4 mô tả hai phép quay trên cây nhị phân tìm kiếm.




***Hình 1.23. Phép quay cây***

Các thuật toán quay trái và quay phải có thể viết như sau:

```
procedure RotateLeft(x: PNode);
var y, z, branch: PNode;
begin
  y := x^.right;
  z := x^.parent;
  branch := y^.left;
  SetLink(x, branch, False); //Cho branch trở thành con phải của x
  SetLink(y, x, True); //Cho x trở thành con trái của y
  SetLink(z, y, (z^.left = x)); //Móc nối y vào làm con của z thay cho x
  if root = x then root := y; //Cập nhật lại gốc cây nếu trước đây x là gốc
end;
```

```
procedure RotateRight(y: PNode);
var x, z, branch: PNode;
begin
  x := y^.left;
  z := y^.parent;
  branch := x^.right;
  SetLink(y, branch, True); //Cho branch trở thành con trái của y
  SetLink(x, y, False); //Cho y trở thành con phải của x
  SetLink(z, x, z^.left = y); //Móc nối x vào làm con của z thay cho y
  if root = y then root := x; //Cập nhật lại gốc cây nếu trước đây y là gốc
end;
```

Dễ thấy rằng một BST sau phép quay vẫn là BST. Chúng ta có thể viết một thao tác $UpTree(x)$ tổng quát hơn: Với $x \neq root$, và $y = x^{\wedge}.parent$, phép $UpTree(x)$ sẽ quay theo liên kết $y \longrightarrow x$ để đẩy nút $x$ lên phía gốc cây (độ sâu của $x$ giảm 1) và kéo nút $y$ xuống sâu hơn một mức làm con nút $x$.

```
procedure UpTree(x: PNode);
var y, z, branch: PNode;
begin
  y := x^.parent; //y^ là nút cha của x^
  z := y^.parent; //z^ là nút cha của y^
  if x = y^.left then //Quay phải
    begin
      branch := x^.right;
      SetLink(y, branch, True);
      //Chuyển nhánh gốc branch^ của x^ sang làm con trái y^
      SetLink(x, y, False); //Cho y^ làm con phải x^
    end
  else //Quay trái
    begin
      branch := x^.left;
      SetLink(y, branch, False);
       //Chuyển nhánh gốc branch^ của x^ sang làm con phải y^
      SetLink(x, y, True); //Cho y^ làm con trái x^
    end;
  SetLink(z, x, z^.left = y); //Móc nối x^ vào làm con z^ thay cho y^
  if root = y then root := x;
  //Cập nhật lại gốc BST nếu trước đât y^ là gốc
```

```
end;
```

### *5.3. Hiệu lực của các thao tác trên cây nhị phân tìm kiếm*

Có thể chứng minh được rằng các thao tác $Minimum$, $Maximum$, $Predecessor$, $Successor$, $Search$, $Insert$ đều có thời gian thực hiện $O(h)$ với $h$ là chiều cao của cây nhị phân tìm kiếm. Hơn nữa, trong trường hợp xấu nhất, các thao tác này đều có thời gian thực hiện $\Theta(h)$.

Vậy khi lưu trữ $n$ khóa bằng cây nhị phân tìm kiếm thì cấu trúc BST tốt nhất là cấu trúc cây nhị phân gần hoàn chỉnh (có chiều cao thấp nhất: $h = \lfloor \lg n \rfloor$) còn cấu trúc BST tồi nhất để biểu diễn là cấu trúc cây nhị phân suy biến (có chiều cao $h = n - 1$).

### *5.4. Cây nhị phân tìm kiếm tự cân bằng*

#### a) Tính cân bằng

Để tăng tính hiệu quả của các thao tác cơ bản trên BST, cách chung nhất là cố gắng giảm chiều cao của cây. Với một BST gồm $n$ nút, dĩ nhiên giải pháp lý tưởng là giảm được chiều cao xuống còn $\lfloor \lg n \rfloor$ (cây nhị phân gần hoàn chỉnh) nhưng điều này thường làm ảnh hưởng nhiều tới thời gian thực hiện giải thuật. Người ta nhận thấy rằng muốn một cây nhị phân thấp thì phải cố gắng giữ được sự cân bằng (về chiều cao và số nút) giữa hai nhánh con của một nút bất kỳ. Chính vì vậy những ý tưởng ban đầu để giảm chiều cao của BST xuất phát từ những kỹ thuật cân bằng cây, từ đó người ta xây dựng các cấu trúc dữ liệu cây nhị phân tìm kiếm có khả năng *tự cân bằng* (*self-balancing binary search tree*) với mong muốn giữ được chiều cao của BST luôn là một đại lượng $O(\lg n)$.

#### b) Một số dạng BST tự cân bằng

❑ ***Cây AVL***

Một trong những phát kiến đầu tiên về cấu trúc BST tự cân bằng là cây AVL [**1**]. Trong mỗi nhánh cây AVL, chiều cao của nhánh con trái và nhánh con phải hơn kém nhau không quá 1. Mỗi nút của cây AVL chứa thêm một thông tin về hệ số cân bằng (độ lệch chiều cao giữa nhánh con trái và nhánh con phải). Ngay sau mỗi phép chèn/xóa, hệ số cân bằng của một số nút được cập nhật lại và nếu

phát hiện một nút có hệ số cân bằng $\geq 2$, phép quay cây sẽ được thực hiện để cân bằng độ cao giữa hai nhánh con của nút đó.

Một cây AVL có độ cao $h$ thì có không ít hơn $f(h+3)-1$ nút. Ở đây $f(h+3)$ là số fibonacci thứ $h+3$. Các tác giả cũng chứng minh được rằng chiều cao của cây AVL có $n$ nút trong là một đại lượng $\lesssim 1.4404\lg(n+2)-0.328$.

❑ ***Cây đỏ đen***

Một dạng khác của BST tự cân bằng là cây đỏ đen (Red-Black tree) [**4**]. Mỗi nút của cây đỏ đen chứa thêm một bit màu (đỏ hoặc đen). Ngoài các tính chất của BST, cây đỏ đen thỏa mãn 5 tính chất sau đây:

- Mọi nút đều được tô màu (đỏ hoặc đen)
- Nút gốc $root\hat{}$ có màu đen
- Nút $nilT\hat{}$ có màu đen
- Nếu một nút được tô màu đỏ thì cả hai nút con của nó phải được tô màu đen
- Với mỗi nút, tất cả các đường đi từ nút đó đến các nút lá hậu duệ có cùng một số lượng nút đen.

Tương tự như cây AVL, đi kèm với các phép chèn/xóa trên cây đỏ đen là những thao tác tô màu và cân bằng cây. Người ta chứng minh được rằng chiều cao của cây đỏ đen có $n$ nút trong không vượt quá $2\lg(n+1)$. Trên thực tế cây đỏ đen nhanh hơn cây AVL ở phép chèn và xóa nhưng chậm hơn ở phép tìm kiếm.

❑ ***Cây Splay***

Còn rất nhiều dạng BST tự cân bằng khác nhưng một trong những ý tưởng thú vị nhất là cây Splay [**35**]. Cây Splay duy trì sự cân bằng mà không cần thêm một thông tin phụ trợ nào ở mỗi nút. Phép "làm bẹp" cây được thực hiện mỗi khi có lệnh truy cập, những nút thường xuyên được truy cập sẽ được đẩy dần lên gần gốc cây để có tốc độ truy cập nhanh hơn. Các phép tìm kiếm, chèn và xóa trên cây Splay cũng được thực hiện trong thời gian $O(\lg n)$ (đánh giá bù trừ). Trong trường hợp tần suất thực hiện phép tìm kiếm trên một khóa hay một cụm khóa cao hơn hẳn so với những khóa khác, cây splay sẽ phát huy được ưu thế về mặt tốc độ.

### c) Vấn đề chứng minh lý thuyết và cài đặt

Cây AVL và cây đỏ đen thường được đưa vào giảng dạy trong các giáo trình cấu trúc dữ liệu vì các tính chất của hai cấu trúc dữ liệu khá dễ dàng trong chứng minh lý thuyết. Cây Splay thường được sử dụng trong các phần mềm ứng dụng vì tốc độ nhanh và tính chất "nhanh hơn nếu truy cập lại" (quick to access again). Ba loại cây này khá phổ biến trong các thư viện hỗ trợ của các môi trường phát triển cao cấp để viết các phần mềm ứng dụng. Lập trình viên có thể tùy chọn loại cây thích hợp nhất để cài đặt giải quyết vấn đề của mình.

Trong trường hợp bạn lập trình trong thời gian hạn hẹp mà không có thư viện hỗ trợ (chẳng hạn trong các kỳ thi lập trình), phải nói rằng việc cài đặt các loại cây kể trên không hề đơn giản và dễ nhầm lẫn (bạn có thể tham khảo trong các tài liệu khác về cơ sở lý thuyết và kỹ thuật cài đặt các loại cây kể trên). Nếu bạn cần sử dụng các cấu trúc dữ liệu này trong phần mềm, theo tôi các bạn nên sử dụng các thư viện sẵn có hoặc viết một thư viện lớp mẫu (class template) thật cẩn thận để sử dụng lại.

Tôi sẽ không đi vào chi tiết các loại cây này. Điều bạn phải nhớ chỉ là có tồn tại những cấu trúc như vậy, để khi đánh giá một thuật toán có sử dụng BST, có thể coi các thao tác cơ bản trên BST được thực hiện trong thời gian $O(\lg n)$.

Trong bài sau tôi sẽ giới thiệu một cấu trúc BST khác dễ cài đặt hơn, dễ tùy biến hơn, và tốc độ cũng không hề thua kém trên thực tế: Cấu trúc Treap.

## Bài tập

**1.16.** Quá trình tìm kiếm trên BST có thể coi như một đường đi xuất phát từ nút gốc. Giáo sư X phát hiện ra một tính chất thú vị: Nếu đường đi trong quá trình tìm kiếm kết thúc ở một nút lá, ký hiệu $L$ là tập các giá trị chứa trong các nút nằm bên trái đường đi và $R$ là tập các giá trị chứa trong các nút nằm bên phải đường đi. Khi đó $\forall x \in L, y \in R$, ta có $x \leq y$. Chứng minh phát hiện của giáo sư X là đúng hoặc chỉ ra một phản ví dụ.

**1.17.** Cho BST gồm $n$ nút, bắt đầu từ nút $Minimum\hat{}$, người ta gọi hàm $Successor$ để đi sang nút liền sau cho tới khi duyệt qua nút $Maximum\hat{}$.

Chứng minh rằng thời gian thực hiện giải thuật này có thời gian thực hiện $O(n)$.

*Gợi ý:* Thực hiện $n$ lời gọi $Successor$ lên tiếp đơn giản chỉ là duyệt qua các liên kết cha/con trên BST mỗi liên kết tối đa 2 lần. Điều tương tự có thể chứng minh được nếu ta bắt đầu từ nút $Maximum$^ và gọi liên tiếp hàm $Predecessor$ để đi sang nút liền trước.

**1.18.** Cho BST có chiều cao $h$, bắt đầu từ một nút $p_1$, người ta tìm nút $p_2$ là nút liền sau $p_1$: $p_2 \coloneqq Successor(p_1)$, tiếp theo lại tìm nút $p_3$ là nút liền sau $p_2$,… Chứng minh rằng thời gian thực hiện $k$ lần phép $Successor$ như vậy chỉ mất thời gian $O(k + h)$.

**1.19.** (tree Sort) Người ta có thể thực hiện việc sắp xếp một dãy khóa bằng cây nhị phân tìm kiếm: Chèn lần lượt các giá trị khóa vào một cây nhị phân tìm kiếm sau đó duyệt cây theo thứ tự giữa. Đánh giá thời gian thực hiện giải thuật trong trường hợp tốt nhất, xấu nhất và trung bình. Cài đặt thuật toán tree Sort.

**1.20.** Viết thuật toán $SearchLE(k)$ để tìm nút chứa khóa lớn nhất $\leq k$ trong BST.

**1.21.** Viết thuật toán $SearchGE(k)$ để tìm nút chứa khóa nhỏ nhất $\geq k$ trong BST.

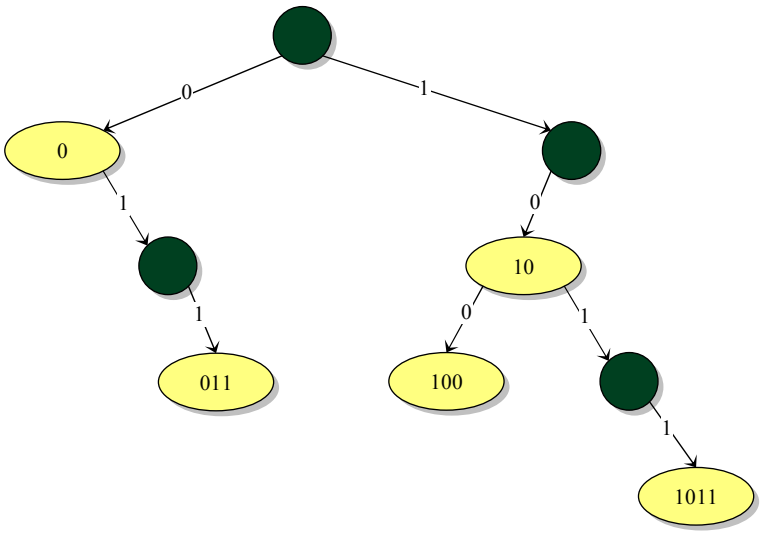
**1.22.** Viết thủ tục $MovetoRoot(p)$ nhận vào nút $p$ và dùng các phép quay để chuyển nút $p$ thành gốc của cây BST.

**1.23.** Viết thủ tục $MovetoLeaf(p)$ nhận vào nút $p$ và dùng các phép quay để chuyển nút $p$ thành một nút lá của cây BST.

**1.24.** Radix tree (cây tìm kiếm cơ số) là một cây nhị phân trong đó mỗi nút có thể chứa hoặc không chứa giá trị khóa, (người ta thường dùng một giá trị đặc biệt tương ứng với nút không chứa giá trị khóa hoặc sử dụng thêm một bit đánh dấu những nút không chứa giá trị khóa)

Các giá trị khóa lưu trữ trên Radix tree là các dãy nhị phân, hay tổng quát hơn là một kiểu dữ liệu nào đó có thể mã hóa bằng các dãy nhị phân. Phép chèn một khóa vào Radix tree được thực hiện như sau: Bắt đầu từ nút gốc ta duyệt biểu diễn nhị phân của khóa, gặp bit 0 đi sang nút con trái và gặp

bit 1 đi sang nhánh con phải, mỗi khi không đi được nữa (đi vào liên kết $nilT$), ta tạo ra một nút và nối nó vào cây ở chỗ liên kết $nilT$ vừa rẽ sang rồi đi tiếp. Cuối cùng ta đặt khóa vào nút cuối cùng trên đường đi. Hình dưới đây là Radix tree sau khi chèn các giá trị 1011, 10, 100, 0, 011. Các nút tô đậm không chứa khóa




Gọi $S$ là tập chứa các khóa là các dãy nhị phân, tổng độ dài các dãy nhị phân trong $S$ là $n$. Chỉ ra rằng chúng ta chỉ cần mất thời gian $\Theta(n)$ để xây dựng Radix tree chứa các phần tử của $S$, mất thời gian $\Theta(n)$ để duyệt Radix tree theo thứ tự giữa và liệt kê các phần tử của $S$ theo thứ tự từ điển.

**1.25.** Cho BST tạo thành từ $n$ khóa được chèn vào theo một trật tự ngẫu nhiên, gọi Gọi $X$ là biến ngẫu nhiên cho chiều cao của BST. Chứng minh rằng kỳ vọng $E[X] = \mathrm{O}(\lg n)$.

**1.26.** Cho BST tạo thành từ $n$ khóa được chèn vào theo một trật tự ngẫu nhiên, gọi Gọi $X$ là biến ngẫu nhiên cho độ sâu của một nút. Chứng minh rằng kỳ vọng $E[X] = \mathrm{O}(\lg n)$.

**1.27.** Gọi $b(n)$ là số lượng các cây nhị phân tìm kiếm chứa $n$ khóa hoàn toàn phân biệt.

- Chứng minh rằng $b(0) = 1$ và $b(n) = \sum_{k=0}^{n-1} b_k b_{n-1-k}$

- Chứng minh rằng $b(n) = \frac{1}{n+1}\binom{2n}{n}$ (số catalan thứ $n$). Từ đó suy ra xác suất để BST là cây nhị phân gần hoàn chỉnh (hoặc cây nhị phân suy biến) nếu $n$ khóa được chèn vào theo thứ tự ngẫu nhiên.
- Chứng minh công thức xấp xỉ $b(n) = \frac{4^n}{\sqrt{\pi}n^{3/2}}\left(1 + O(1/n)\right)$

## 6. Cây nhị phân tìm kiếm ngẫu nhiên

### *6.1. Độ cao trung bình của BST*

Trong bài trước ta đã biết rằng các thao tác cơ bản của BST được thực hiện trong thời gian $O(h)$ với $h$ là chiều cao của cây. Nếu $n$ khóa được chèn vào một BST rỗng, ta sẽ được một BST gồm $n$ nút. Chiều cao của BST có thể là một số nguyên nào đó nằm trong phạm vi từ $\lfloor lgn \rfloor$ tới $n - 1$. Nếu thay đổi thứ tự chèn $n$ khóa vào cây, ta có thể thu được một cấu trúc BST khác.

Điều chúng ta muốn biết là nếu chèn $n$ khóa vào BST theo các trật tự khác nhau thì độ cao trung bình của BST thu được là bao nhiêu. Hay nói chính xác hơn, chúng ta cần biết giá trị kỳ vọng của độ cao một BST khi chèn $n$ khóa vào theo một trật tự ngẫu nhiên.

Thực ra trong các thao tác cơ bản của BST, độ sâu trung bình của các nút mới là yếu tố quyết định hiệu suất chứ không phải độ cao của cây. Độ sâu của nút $i$ chính là số phép so sánh cần thực hiện để chèn nút $i$ vào BST. Tổng số phép so sánh để chèn toàn bộ $n$ nút vào BST có thể đánh giá tương tự như QuickSort, bằng $O(n \lg n)$. Vậy độ sâu trung bình của mỗi nút là $\frac{1}{n}O(n \lg n) = O(\lg n)$.

Người ta còn chứng minh được một kết quả mạnh hơn: Độ cao trung bình của BST là một đại lượng $O(\lg n)$. Cụ thể là $E[h] \leq 3 \lg n + O(1)$ với $E[h]$ là giá trị kỳ vọng của độ cao và $n$ là số nút trong BST. Chứng minh này khá phức tạp, bạn có thể tham khảo trong các tài liệu khác .

## *6.2. Treap*

Chúng ta có thể tránh trường hợp suy biến của BST bằng cách chèn các nút vào cây theo một trật tự ngẫu nhiên[*]. Tuy nhiên trên thực tế rất ít khi chúng ta đảm bảo được các nút được chèn/xóa trên BST theo trật tự ngẫu nhiên, bởi các thao tác trên BST thường do một tiến trình khác thực hiện và thứ tự chèn/xóa hoàn toàn do tiến trình đó quyết định.

Trong mục này chúng ta quan tâm tới một dạng BST mà cấu trúc của nó không phụ thuộc vào thứ tự chèn/xóa: Treap.

Cho mỗi nút của BST thêm một thông tin $priority$ gọi là "độ ưu tiên". Độ ưu tiên của mỗi nút là một số dương. Khi đó Treap[†] [**33**] được định nghĩa là một BST thỏa mãn tính chất của Heap. Cụ thể là:

- Nếu nút $y$ nằm trong nhánh con trái của nút $x$ thì $y.key \leq x.key$.
- Nếu nút $y$ nằm trong nhánh con phải của nút $x$ thì $y.key \geq x.key$.
- Nếu nút $y$ là hậu duệ của nút $x$ thì $y.priority \leq x.priority$

Hai tính chất đầu tiên là tính chất của BST, tính chất thứ ba là tính chất của Heap. Nút gốc của Treap có độ ưu tiên lớn nhất. Để tiện trong cài đặt, ta quy định nút giả $nilT$^ có độ ưu tiên bằng 0.

***Định lý 6-1***

Xét một tập các nút, mỗi nút chứa khóa và độ ưu tiên, khi đó tồn tại cấu trúc Treap chứa các nút trên.

> ***Chứng minh***
>
> Khởi tạo một BST rỗng và chèn lần lượt các nút vào BST theo thứ tự từ nút ưu tiên cao nhất tới nút ưu tiên thấp nhất. Hai ràng buộc đầu tiên được thỏa mãn vì ta sử dụng phép chèn của BST. Hơn nữa phép chèn của BST luôn chèn nút mới vào thành nút lá nên sau mỗi bước chèn, nút lá mới chèn vào không thể mang độ ưu tiên lớn hơn các nút tiền bối của nó được. Điều này chỉ ra rằng BST tạo thành là một Treap.

---

[*] Từ "tránh" ở đây không chính xác, trên thực tế phương pháp này không tránh được trường hợp xấu. Có điều là xác suất xảy ra trường hợp xấu quá nhỏ và rất khó để "cố tình" chỉ ra cụ thể trường hợp xấu (giống như Randomized QuickSort).

[†] Tên gọi Treap là ghép của hai từ: "tree" và "Heap"

***Định lý 6-2***

Xét một tập các nút, mỗi nút chứa khóa và độ ưu tiên. Nếu các khóa cũng như độ ưu tiên của các nút hoàn toàn phân biệt thì tồn tại duy nhất cấu trúc Treap chứa các nút trên.

> ***Chứng minh***
>
> Sự tồn tại của cấu trúc Treap đã được chỉ ra trong chứng minh trong Định lí 6.1. Tính duy nhất của cấu trúc Treap này có thể chứng minh bằng quy nạp: Rõ ràng định lý đúng với tập gồm 0 nút (Treap rỗng). Xét tập gồm $\geq 1$ nút, khi đó nút có độ ưu tiên lớn nhất chắc chắn sẽ phải là gốc Treap, những nút mang khóa nhỏ hơn khóa của nút gốc phải nằm trong nhánh con trái và những nút mang khóa lớn hơn khóa của nút gốc phải nằm trong nhánh con phải. Sự duy nhất về cấu trúc của nhánh con trái và nhánh con phải được suy ra từ giả thiết quy nạp. ĐPCM.

Trong cài đặt thông thường của Treap, độ ưu tiên $priority$ của mỗi nút thường được gán bằng một **số ngẫu nhiên** để vô hiệu hóa những tiến trình "cố tình" làm cây suy biến: Cho dù các nút được chèn/xóa trên Treap theo thứ tự nào, cấu trúc của Treap sẽ luôn giống như khi chúng ta chèn các nút còn lại vào theo thứ tự giảm dần của $Priority$ (tức là thứ tự ngẫu nhiên). Hơn nữa nếu biết trước được tập các nút sẽ chèn vào Treap, ta còn có thể gán độ ưu tiên $priority$ cho các nút một cách hợp lý để "ép" Treap thành cây nhị phân gần hoàn chỉnh (trung vị của tập các khóa sẽ được gán độ ưu tiên cao nhất để trở thành gốc cây, tương tự với nhánh trái và nhánh phải…). Ngoài ra nếu biết trước tần suất truy cập nút ta có thể gán độ ưu tiên của mỗi nút bằng tần suất này để các nút bị truy cập thường xuyên sẽ ở gần gốc cây, đạt tốc độ truy cập nhanh hơn.

## *6.3.Các thao tác trên Treap*

### a) Cấu trúc nút

Tương tự như BST, cấu trúc nút của Treap chỉ có thêm một trường $priority$ để lưu độ ưu tiên của nút

```
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record
    key: TKey;
    parent, left, right: PNode;
    priority: Integer;
```

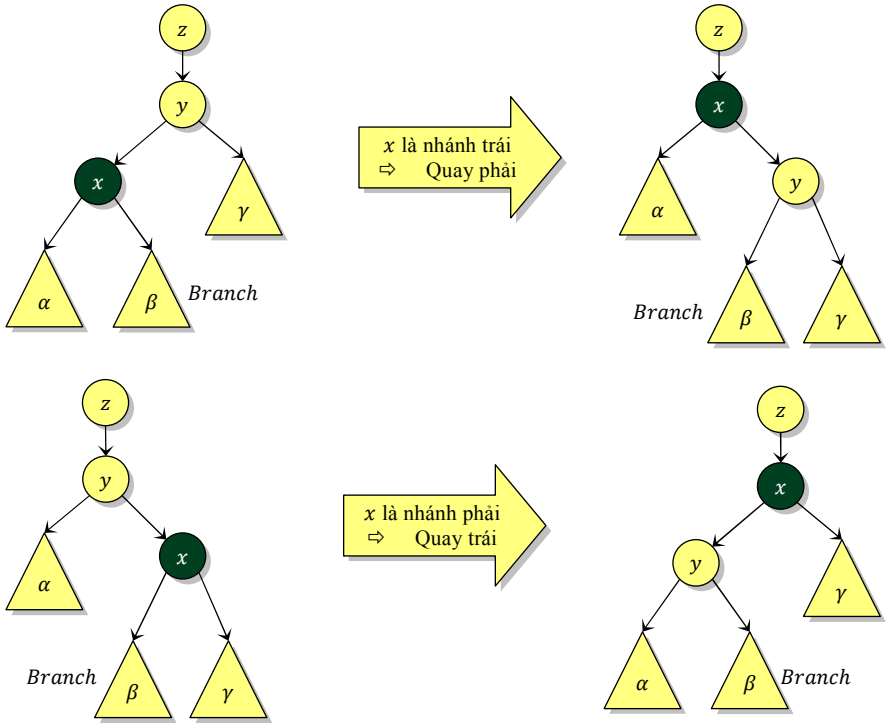
```
  end;
var
  sentinel: TNode;
  nilT: PNode; //Con trỏ tới nút đặt biệt
  root: PNode; //Con trỏ tới nút gốc
begin
  sentinel.priority := 0;
  nilT := @sentinel;
  ...
end.
```

Trên lý thuyết người ta thường cho các giá trị $Priority$ là số thực ngẫu nhiên, khi cài đặt ta có thể cho $Priority$ số nguyên dương lấy ngẫu nhiên trong một phạm vi đủ rộng. Ký hiệu $RP$ là hàm trả về một số dương ngẫu nhiên, bạn có thể cài đặt hàm này bằng bất kỳ một thuật toán tạo số ngẫu nhiên nào. Ví dụ:

```
function RP: Integer;
begin
  Result := 1 + Random(MaxInt - 1);
  //Lấy ngẫu nhiên từ 1 tới MaxInt - 1
end;
```

Các phép khởi tạo cây rỗng, tìm phần tử lớn nhất, nhỏ nhất, tìm phần tử liền trước, liền sau trên Treap không khác gì so với trên BST thông thường. Phép quay không được thực hiện tùy tiện trên Treap vì nó sẽ phá vỡ ràng buộc thứ tự Heap, thay vào đó chỉ có thao tác UpTree được nhúng vào trong mỗi phép chèn (Insert) và xóa (Delete) để hiệu chỉnh cấu trúc Treap.

Nhắc lại về thao tác $UpTree(x)$




*Hình 1.24. Thao tác* $UpTree(x)$

```
procedure UpTree(x: PNode);
var y, z, branch: PNode;
begin
  y := x^.parent; //y^ là nút cha của x^
  z := y^.parent; //z^ là nút cha của y^
  if x = y^.left then //Quay phải
    begin
      branch := x^.right;
      SetLink(y, branch, True);
      SetLink(x, y, False);
    end
  else //Quay trái
    begin
      branch := x^.left;
      SetLink(y, branch, False);
      SetLink(x, y, True);
    end;
```

```
  SetLink(z, x, z^.left = y); //Móc nối x^ vào làm con z^ thay cho y^
  if root = y then root := x;
  //Cập nhật lại gốc BST nếu trước đât y^ là gốc
end;
```

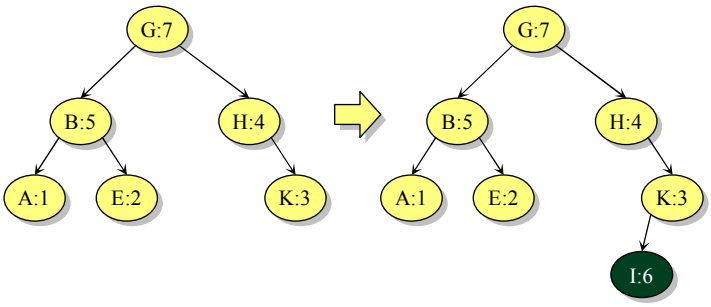
## b) Chèn

Phép chèn trên Treap trước hết thực hiện như phép chèn trên BST để chèn khóa vào một nút lá. Nút lá $x$^ mới chèn vào sẽ được gán một độ ưu tiên ngẫu nhiên. Tiếp theo là phép hiệu chỉnh Treap: Gọi $y$^ là nút cha của $x$^, chừng nào thấy $x$^ mang độ ưu tiên lớn hơn $y$^ (vi phạm thứ tự Heap) ta thực hiện lệnh $UpTree(x)$ để đẩy nút $x$^ lên làm cha nút $y$^ và kéo nút $y$^ xuống làm con nút $x$^.

```
//Chèn khóa k vào Treap, trả về con trỏ tới nút chứa k
function Insert(k: TKey): PNode;
var x, y: PNode;
begin
 //Thực hiện phép chèn như trên BST
  y := nilT;
  x := root;
  while x ≠ nilT do
    begin
      y := x;
      if k < x^.key then x := x^.left
      else x := x^.right;
    end;
  New(x);
  x^.key := k;
  x^.left := nilT;
  x^.right := nilT;
  SetLink(y, x, k < y^.key);
  if root = nilT then root := x;
  //Chỉnh Treap
  x^.priority := RP; //Gán độ ưu tiên ngẫu nhiên
  repeat
    y := x^.parent;
    if (y ≠ nilT)
        and (x^.priority > y^.priority) then UpTree(x)
```

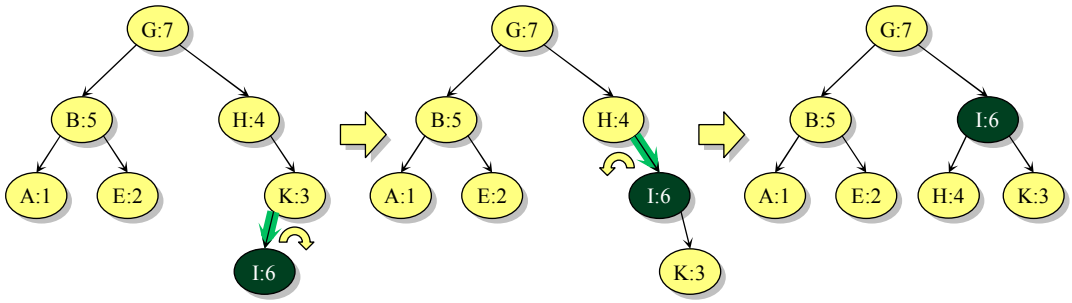
```
    else Break;
  until False;
  Result := x;
end;
```

Ví dụ chúng ta có một Treap chứa các khóa A, B, E, G, H, K với độ ưu tiên là A:1, B:5, E:2, G:7, H:4, K:3 và chèn một nút khóa chứa khóa I và độ ưu tiên 6 vào Treap, trước hết thuật toán chèn trên BST được thực hiện như trong hình 1.25.




***Hình 1.15. Phép chèn trên Treap trước hết thực hiện như trong BST***

Tiếp theo là hai phép $UpTree$ để chuyển nút I:6 về vị trí đúng trên Treap (h.1.26)




***Hình 1.26. Sau phép chèn BST là các phép UpTree để chỉnh lại Treap***

Số phép $UpTree$ cần thực hiện phụ thuộc vị trí và độ ưu tiên của nút mới chèn vào (Có thể là số nào đó từ 0 tới $h$ với $h$ là độ cao của Treap), nhưng người ta đã chứng minh được định lý sau:

***Định lý 6.3***

Trung bình số phép $UpTree$ cần thực hiện trong phép chèn $Insert$ là 2.

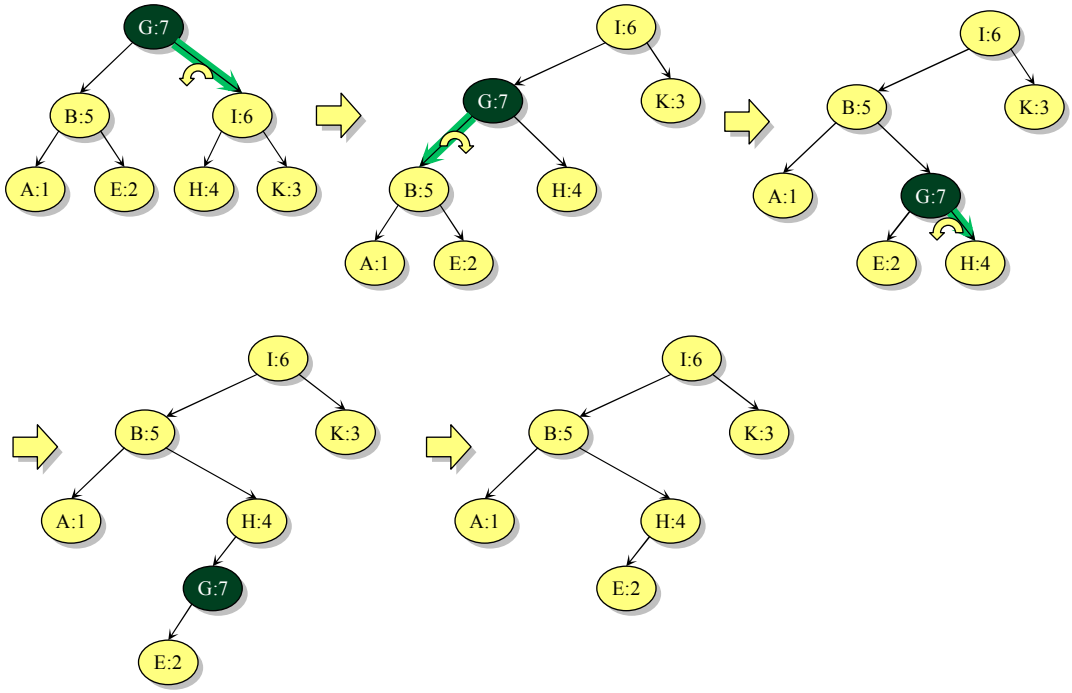
### c) Xóa

Phép xóa nút $x$^ trên Treap được thực hiện như sau:

- Nếu $x$^ có ít hơn hai nhánh con, ta lấy nút con (nếu có) của $x$^ lên thay cho $x$^ và xóa nút $x$^.
- Nếu $x$^ có đúng hai nhánh con, gọi $y$^ là nút con mang độ ưu tiên lớn hơn trong hai nút con, thực hiện phép $UpTree(y)$ để kéo nút $x$^ xuống sâu phía dưới lá và lặp lại cho tới khi $x$^ chỉ còn một nút con. Việc xóa quy về trường hợp trên

```
procedure Delete(x: PNode);
var y, z: PNode;
begin
  while (x^.left ≠ nilT) and (x^.right ≠ nilT) do
  //Chừng nào x^ có 2 nút con
    begin
      //Xác định y^ là nút con mang độ ưu tiên lớn hơn
      y := x^.left;
      if y^.priority < x^.right^.priority then
         y := x^.right;
      UpTree(y); //Đẩy y lên phía gốc, kéo x xuống phía lá
    end;
  //Bây giờ x^ chỉ có tối đa một nút con, xác định y^ là nút con (nếu có) của x
  if x^.left <> nilT then y := x^.left
  else y := x^.right;
  z := x^.parent; //z^ là nút cha của x^
  SetLink(z, y, z^.left = x); //Cho y^ làm con của z^ thay cho x^
  if x = root then root := y; //Cập nhật lại gốc
  Dispose(x); //Giải phóng bộ nhớ
end;
```

Ví dụ chúng ta có một Treap chứa các khóa A, B, E, G, H, I, K với độ ưu tiên là A:1, B:5, E:2, G:7, H:4, I:6, K:3 và xóa nút chứa khóa G. Ba phép $UpTree$ (quay) sẽ được thực hiện trước khi xóa nút chứa khóa G




***Hình 1.27. Thực hiện các phép quay đẩy dần nút cần xóa xuống dưới lá, khi nút cần xóa còn ít hơn 1 nhánh con thì xóa trực tiếp.***

Tương tự như phép chèn, số phép $UpTree$ cần thực hiện phụ thuộc vị trí và độ ưu tiên của nút bị xóa, nhưng người ta đã chứng minh được định lý sau đây.

***Định lý 6-4***

Trung bình số phép $UpTree$ cần thực hiện trong phép xóa $Delete$ là 2.

## Bài tập

**1.28.** Thứ tự thống kê: Cho một Treap, hãy xây dựng thuật toán tìm khóa đứng thứ $p$ khi sắp thứ tự. Ngược lại cho một nút, hãy tìm số thứ tự của nút đó khi duyệt Treap theo thứ tự giữa.

**1.29.** Treap biểu diễn tập hợp

Khi dùng Treap $T$ biểu diễn tập hợp các giá trị khóa, (tức là các khóa trong Treap hoàn toàn phân biệt), phép thử $k \in T$ có thể được thực hiện thông

qua hàm $Search$. Việc thêm một phần tử vào tập hợp có thể thực hiện thông qua một sửa đổi của hàm $Insert$ (Chỉ chèn nếu khóa chưa có trong Treap). Việc xóa một phần tử khỏi tập hợp cũng được thực hiện thông qua việc sửa đổi thủ tục $Delete$ (tìm phần tử trong Treap, nếu tìm thấy thì thực hiện phép xóa). Ngoài ra còn có nhiều thao tác khác được thực hiện rất hiệu quả với cấu trúc Treap, hãy cài đặt các thao tác sau đây trên Treap:

**Phép tách (Split)**: Với một giá trị $k_0$, tách các khóa $< k_0$ và các khóa $> k_0$ ra hai Treap biểu diễn hai tập hợp riêng rẽ.

*Gợi ý:* Tìm nút chứa phần tử $k_0$ trong Treap, nếu không thấy thì chèn $k_0$ vào một nút mới. Đặt độ ưu tiên của nút này bằng $+\infty$. Theo nguyên lý của cấu trúc Treap, nút này sẽ được đẩy lên thành gốc cây. Ngoài ra theo nguyên lý của cấu trúc BST, nhánh con trái của gốc cây sẽ chứa tất cả các khóa $< k_0$ và nhánh con phải của gốc cây sẽ chứa tất cả các khóa $> k_0$.

**Phép hợp (Union)**: Cho hai Treap chứa hai tập khóa, xây dựng Treap mới chứa tất cả các khóa của hai Treap ban đầu

**Phép giao (Intersection)**: Cho hai Treap chứa hai tập khóa, xây dựng Treap mới chứa tất cả các khóa có mặt trong cả hai Treap ban đầu

**Phép lấy hiệu (Difference)**: Cho hai Treap $A, B$ chứa hai tập khóa, xây dựng Treap mới chứa các khóa thuộc $A$ nhưng không thuộc $B$.

## 7. Một số ứng dụng của cây nhị phân tìm kiếm

Ngoài ứng dụng để biểu diễn tập hợp (Bài tập 6.2), cây nhị phân tìm kiếm còn có nhiều ứng dụng quan trọng khác nữa. Trong bài này chúng ta sẽ khảo sát một vài ứng dụng khác của cấu trúc BST.

Cấu trúc BST thông thường có thể dùng để cài đặt chương trình giải quyết những vấn đề trong bài, tuy nhiên bạn nên sử dụng một dạng BST tự cân bằng hoặc Treap để tránh trường hợp xấu của BST.

### *7.1. Cây biểu diễn danh sách*

Chúng ta đã biết những cách cơ bản để biểu diễn danh sách là sử dụng mảng hoặc danh sách móc nối. Sử dụng mảng có tốc độ tốt với phép truy cập ngẫu nhiên nhưng sẽ bị chậm nếu danh sách luôn bị biến động bởi các phép chèn/xóa.

Trong khi đó, sử dụng danh sách móc nối có thể thuận tiện hơn trong các phép chèn/xóa thì lại gặp nhược điểm trong phép truy cập ngẫu nhiên.

Trong mục này chúng ta sẽ trình bày một phương pháp biểu diễn danh sách bằng cây nhị phân mà các trên đó, phép truy cập ngẫu nhiên, chèn, xóa đều được thực hiện trong thời gian $O(\lg n)$. Ta sẽ phát biểu một bài toán cụ thể và cài đặt chương trình giải bài toán đó.

### a) Bài toán

Cho một danh sách $L$ để chứa các số nguyên. Ký hiệu $Length(L)$ là số phần tử trong danh sách. Xét các thao tác căn bản trên danh sách:

- Phép chèn $Insert(v, i)$: Nếu $1 \leq i \leq Length(L) + 1$, thao tác này chèn một số $v$ vào vị trí $i$ của danh sách, nếu không thao tác này không có hiệu lực. (Trường hợp $i = Length(L) + 1$ thì $Value$ sẽ được thêm vào cuối danh sách).
- Phép xóa $Delete(i)$: Nếu $1 \leq i \leq Length(L)$, thao tác này xóa phần tử thứ $i$ trong danh sách, nếu không thao tác này không có hiệu lực.

Cho danh sách $L = \emptyset$ và $n$ thao tác thuộc một trong hai loại, hãy in ra các phần tử theo đúng thứ tự trong danh sách cuối cùng.

**Input**

- Dòng 1 chứa số nguyên dương $n \leq 10^5$
- $n$ dòng tiếp, mỗi dòng cho thông tin về một thao tác. Mỗi dòng bắt đầu bởi một ký tự $\in \{I, D\}$. Nếu ký tự đầu dòng là "I" thì tiếp theo là hai số nguyên $v, i$ tương ứng với phép chèn $Insert(v, i)$, nếu ký tự đầu dòng là $D$ thì tiếp theo là số nguyên $i$ tương ứng với phép xóa $Delete(i)$. Các giá trị $v, i$ là số nguyên Integer.

**Output**

Các phần tử trong danh sách cuối cùng theo đúng thứ tự.

| Sample Input | Sample Output |
|---|---|
| 8<br>I 5 1<br>I 6 1<br>I 7 1<br>I 8 3<br>I 1 2<br>D 4<br>I 9 2<br>D 1 | 9 1 6 5 |

### b) Giải thuật và tổ chức dữ liệu

Chúng ta sẽ lưu trữ các phần tử của danh sách *L* trong một cấu trúc Treap sao cho nếu duyệt Treap theo thứ tự giữa thì các phần tử của *L* sẽ được liệt kê theo đúng thứ tự trong danh sách.

❑ ***Nút cầm canh cuối danh sách***

Độ ưu tiên của các nút là một số nguyên dương ngẫu nhiên nhỏ hơn hằng số MaxInt. Để tiện cài đặt, ta thêm vào danh sách *L* một phần tử giả đứng cuối danh sách và gán độ ưu tiên MaxInt để phần tử này trở thành nút gốc của Treap. Phần tử giả này có hai công dụng:

- Mọi phép chèn hiệu lực đều có một phần tử của danh sách nằm tại vị trí chèn, ta bớt đi được các phép xử lý trường hợp riêng khi chèn vào cuối danh sách.
- Nút gốc $root\hat{}$ của Treap không bao giờ bị thay đổi, ta không cần phải kiểm tra và cập nhật lại gốc sau các phép chèn hoặc xóa.

Theo cách xây dựng Treap như vậy, toàn bộ các phần tử của danh sách *L* sẽ nằm trong nhánh con trái của nút gốc Treap. Ta chỉ cần duyệt nhánh con trái của nút gốc Treap theo thứ tự giữa là liệt kê được tất cả các phần tử theo đúng thứ tự.

❑ ***Quản lý số nút***

Trong mỗi nút $r\hat{}$ của Treap, ta lưu trữ $r\hat{}.size$ là số lượng nút nằm trong nhánh Treap gốc $r\hat{}$. Trường $size$ của các nút sẽ được cập nhật mỗi khi có sự thay đổi

cấu trúc Treap. Công dụng của trường $size$ là để quản lý số nút trong một nhánh Treap, phục vụ cho phép truy cập ngẫu nhiên.

❑ ***Truy cập ngẫu nhiên***

Cả hai phép chèn và xóa đều có một tham số vị trí $i$. Việc chèn/xóa trên danh sách trừu tượng $L$ sẽ quy về việc chèn/xóa trên Treap sao cho duy trì được sự thống nhất giữa Treap và danh sách $L$ như đã định. Vậy việc đầu tiên chính là xác định nút tương ứng với vị trí $i$ là nút nào trong Treap. Theo nguyên lý của phép duyệt cây theo thứ tự giữa (duyệt nhánh trái, duyệt nút gốc, sau đó duyệt nhánh phải), thuật toán xác định nút tương ứng với vị trí $i$ có thể diễn tả như sau: Xét bài toán tìm nút thứ $i$ trong nhánh Treap gốc $r$^:

- Nếu $i = r\text{^}.left\text{^}.size + 1$ thì nút cần tìm chính là nút $r$.
- Nếu $i < r\text{^}.left\text{^}.size + 1$ thì quy về tìm nút thứ $i$ trong nhánh con trái của $r$.
- Nếu $i > r\text{^}.left\text{^}.size + 1$ thì quy về tìm nút thứ $i - r\text{^}.left\text{^}.size - 1$ trong nhánh con phải của $r$^.

Số bước lặp để tìm nút tương ứng với vị trí $i$ có thể tính bằng độ sâu của nút kết quả (cộng thêm 1). Phép truy cập ngẫu nhiên được cài đặt bằng hàm $NodeAt(i)$: Nhận vào một số nguyên $i$ và trả về nút tương ứng với vị trí đó trên Treap.

❑ ***Chèn***

Để chèn một giá trị $v$ vào vị trí $i$, trước hết ta tạo nút $x$^ chứa giá trị $v$, xác định nút $y$^ là nút hiện đang đứng thứ $i$. Nếu $y$^ không có nhánh trái thì móc nối $x$^ vào thành nút con trái của $y$^. Nếu không ta đi sang nhánh trái của $y$^ và móc nối $x$^ vào thành nút cực phải của nhánh trái này.

Tiếp theo là phải cập nhật số nút, nút $x$^ chèn vào sẽ trở thành nút lá và có $x\text{^}.size = 1$, trường $size$ trong tất cả các nút tiền bối của $x$^ cũng được tăng lên 1 để giữ tính đồng bộ.

Cuối cùng, ta gán cho $x\text{^}.priority$ một độ ưu tiên ngẫu nhiên và thực hiện các phép $UpTree(x)$ để đẩy $x$^ lên vị trí đúng. Chú ý là trong phép $UpTree$, ngoài những thao tác xử lý cơ bản trên Treap, ta phải cập nhật lại trường $size$ của hai nút chịu ảnh hưởng qua phép quay.

❑ *Xóa*

Để xóa phần tử tại vị trí $i$, ta xác định nút $x$^ nằm tại vị trí $i$ và tiến hành xóa nút $x$^. Phép xóa được thực hiện như trên Treap: Chừng nào $x$^ còn hai nút con, ta xác định $y$^ là nút con mang độ ưu tiên lớn hơn và thực hiện $UpTree(y)$ để kéo $x$^ sâu xuống dưới lá. Khi $x$^ còn ít hơn hai nút con, ta đưa nhánh con gốc $y$^ (nếu có) của $x$^ vào thế chỗ và xóa nút $x$^. Sau khi xóa thì toàn bộ trường $Size$ trong các nút tiền bối của $x$^ phải giảm đi 1 để giữ tính đồng bộ.

### c) Cài đặt

💻 DYNLIST.PAS ✓ Cây biểu diễn danh sách

```
{$MODE OBJFPC}
program DynamicList;
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record //Kiểu nút Treap
    value: Integer;
    priority: Integer;
    size: Integer;
    left, right, parent: PNode;
  end;
var
  sentinel: TNode; //Lính canh (= nilT^)
  nilT, root: PNode;
  n: Integer; //Số thao tác
function NewNode: PNode; //Hàm tạo nút mới, trả về con trỏ tới nút mới
begin
  New(Result); //Cấp phát bộ nhớ
  with Result^ do //Khởi tạo các trường trong nút mới tạo ra
    begin
      priority := Random(MaxInt - 1) + 1;
      //Gán độ ưu tiên ngẫu nhiên
      size := 1; //Nút đứng đơn độc, size = 1
      parent := nilT;
      left := nilT;
      right := nilT; //Các trường liên kết được gán = nilT
    end;
```

```
end;
procedure InitTreap; //Khởi tạo Treap
begin
  sentinel.priority := 0;
  sentinel.size := 0;
  nilT := @sentinel; //Đem con trỏ nilT trỏ tới sentinel
  root := NewNode;
  root^.priority := MaxInt; //root^ được gán độ ưu tiên cực đại
end;
//Móc nối ChildNode thành con của ParentNode
procedure SetLink(ParentNode, ChildNode: PNode;
                                  InLeft: Boolean);
begin
  ChildNode^.parent := ParentNode;
  if InLeft then ParentNode^.left := ChildNode
  else ParentNode^.right := ChildNode;
end;
function NodeAt(i: Integer): PNode; //Truy cập ngẫu nhiên
begin
  Result := root; //Bắt đầu từ gốc Treap
  repeat
    if i = Result^.left^.size + 1 then Break;
    //Nếu nút này đứng thứ i thì dừng
    if i <= Result^.left^.size then //Lặp lại, tìm trong nhánh con trái
      Result := Result^.left
    else //Lặp lại, tìm trong nhánh con phải
      begin
        Dec(i, Result^.left^.size + 1);
        Result := Result^.right;
      end;
  until False;
end;
procedure UpTree(x: PNode); //Đẩy x^ lên phía gốc Treap bằng phép quay
var y, z, branch: PNode;
begin
  y := x^.parent;
  z := y^.parent;
  if x = y^.left then //Quay phải
```

```
    begin
      branch := x^.right;
      SetLink(y, branch, True);
      SetLink(x, y, False);
    end
  else //Quay trái
    begin
      branch := x^.left;
      SetLink(y, branch, False);
      SetLink(x, y, True);
    end;
  SetLink(z, x, z^.left = y);
  //Cẩn thận, phải cập nhật y^.size trước khi cập nhật x^.size
  with y^ do size := left^.size + right^.size + 1;
  with x^ do size := left^.size + right^.size + 1;
end;
procedure Insert(v, i: Integer); //Chèn
var x, y: PNode;
begin
  if (i < 1) or (i > root^.size) then Exit;
  //Phép chèn vô hiệu, bỏ qua
  x := NewNode;
  x^.value := v; //Tạo nút x^ chứa value
  y := NodeAt(i); //Xác định nút y^ cần chèn x^ vào trước
  if y^.left = nilT then SetLink(y, x, True)
  //y^ không có nhánh trái, cho x^ làm nhánh trái
  else
    begin
      y := y^.left; //Đi sang nhánh trái
      while y^.right <> nilT do y := y^.right;
      //Tới nút cực phải
      SetLink(y, x, False); //Móc nối x^ vào làm nút cực phải
    end;
    //y = x^.parent, cập nhật trường size của các nút từ y lên gốc
  while y <> nilT do
    begin
      Inc(y^.size);
      y := y^.parent;
    end;
```

```
    //Chỉnh Treap bằng phép UpTree
  while x^.priority > x^.parent^.priority do
  //Chừng nào x^ ưu tiên hơn nút cha
    UpTree(x); //Đẩy x^ lên phía gốc
end;
procedure Delete(i: Integer); //Xóa
var x, y, z: PNode;
begin
  if (i < 1) or (i >= root^.size) then Exit;
  //Phép xóa vô hiệu, bỏ qua
  x := NodeAt(i); //Xác định nút cần xóa x^
  while (x^.left <> nilT) and (x^.right <> nilT) do
  //x^ có hai nút con
    begin //Xác định y^ là nút con mang độ ưu tiên lớn hơn
      if x^.left^.priority > x^.right^.priority then
         y := x^.left
      else y := x^.right;
      UpTree(y); //Kéo x^ xuống sâu phía dưới lá
    end;
  //x^ chỉ còn tối đa 1 nút con, xác định y^ là nút con nếu có của x^
  if x^.left <> nilT then y := x^.left
  else y := x^.right;
  z := x^.parent; //z^ là cha của x^
  SetLink(z, y, z^.left = x); //Cho y^ vào làm con z^ thay cho x^
  Dispose(x); //Giải phóng bộ nhớ
  while z <> nilT do //Cập nhật trường size của các nút từ z^ lên gốc
    begin
      Dec(z^.size);
      z := z^.parent;
    end;
end;
procedure ReadOperators;
//Đọc dữ liệu, gặp thao tác nào thực hiện ngay thao tác đó
var
  k, v, i: Integer;
  op: AnsiChar;
begin
  ReadLn(n);
  for k := 1 to n do
```

```
    begin
      Read(op);
      case op of
        'I': begin
               ReadLn(v, i);
               Insert(v, i);
             end;
        'D': begin
               ReadLn(i);
               Delete(i);
             end;
      end;
    end;
end;
procedure PrintResult; //In kết quả
  procedure InOrderTraversal(x: PNode); //Duyệt cây theo thứ tự giữa
  begin
    if x = nilT then Exit;
    InOrderTraversal(x^.left); //Duyệt nhánh trái
    Write(x^.value, ' '); //In ra giá trị trong nút
    InOrderTraversal(x^.right); //Duyệt nhánh phải
    Dispose(x); //Duyệt xong thì giải phóng bộ nhớ luôn
  end;
begin
  InOrderTraversal(root^.left);
  //Toàn bộ danh sách trừu tượng L nằm trong nhánh trái của gốc
  Dispose(root); //Giải phóng luôn nút gốc
  WriteLn;
end;
begin
  InitTreap;
  ReadOperators;
  PrintResult;
end.
```

### c) Hoán vị Josephus

❑ ***Bài toán tìm hoán vị Josephus***

Bài toán lấy tên của Flavius Josephus, một sử gia Do Thái vào thế kỷ thứ nhất. Tương truyền rằng Josephus và 40 chiến sĩ bị người La Mã bao vây trong một hang động. Họ quyết định tự vẫn chứ không chịu bị bắt. 41 chiến sĩ đứng thành vòng tròn và bắt đầu đếm theo một chiều vòng tròn, cứ người nào đếm đến 3 thì phải tự vẫn và người kế tiếp bắt đầu đếm lại từ 1. Josephus không muốn chết và đã chọn được một vị trí mà ông ta cũng với một người nữa là hai người sống sót cuối cùng theo luật này. Hai người sống sót sau đó đã đầu hàng và gia nhập quân La Mã (Josephus sau đó chỉ nói rằng đó là sự may mắn, hay "bàn tay của Chúa" mới giúp ông và người kia sống sót).

Có rất nhiều truyền thuyết và tên gọi khác nhau về bài toán Josephus, trong toán học người ta phát biểu bài toán dưới dạng một trò chơi: Cho $n$ người đứng quanh vòng tròn theo chiều kim đồng hồ đánh số từ 1 tới $n$. Họ bắt đầu đếm từ người thứ nhất theo chiều kim đồng hồ, người nào đếm đến $m$ thì bị loại khỏi vòng và người kế tiếp bắt đầu đếm lại từ 1. Trò chơi tiếp diễn cho tới khi vòng tròn không còn lại người nào. Nếu ta xếp số hiệu của $n$ người theo thứ tự họ bị loại khỏi vòng thì sẽ được một hoán vị $(j_1, j_2, \dots, j_n)$ của dãy số $(1,2, \dots, n)$ gọi là hoán vị Josephus $(n, m)$. Ví dụ với $n = 7, m = 3$, hoán vị Josephus sẽ là $(3,6,2,7,5,1,4)$. Bài toán đặt ra là cho trước hai số $n, m$ hãy xác định hoán vị Josephus $(n, m)$.

❑ ***Thuật toán***

Bài toán tìm hoán vị Josephus $(n, m)$ có thể giải quyết dễ dàng nếu sử dụng danh sách động (Mục 0): Danh sách được xây dựng có $n$ phần tử tương ứng với $n$ người. Việc xác định người sẽ phải ra khỏi vòng sau đó xóa người đó khỏi danh sách đơn giản chỉ là phép truy cập ngẫu nhiên và xóa một phần tử khỏi danh sách động.

**Nhận xét:** Nếu sau một lượt nào đó, người vừa bị loại là người thứ $p$ và danh sách còn lại $k$ người. Khi đó người kế tiếp bị loại là người đứng thứ: $(p + m - 2) \bmod k + 1$ trong danh sách.

Tuy bài toán khá đơn giản nhưng liên quan tới một kỹ thuật cài đặt quan trọng nên ta sẽ cài đặt cụ thể chương trình tìm hoán vị Josephus $(n, m)$.

**Input**

Hai số nguyên dương $n, m \leq 10^5$

**Output**

Hoán vị Josephus $(n, m)$

| Sample Input | Sample Output |
|---|---|
| 7 3 | 3 6 2 7 5 1 4 |

Xây dựng danh sách gồm $n$ phần tử, ban đầu các phần tử đều chưa đánh dấu (chưa bị xóa). Thuật toán sẽ tiến hành $n$ bước, mỗi bước sẽ đánh dấu một phần tử tương ứng với một người bị loại.

Có thể quan sát rằng nếu biểu diễn danh sách này bằng cây nhị phân gồm $n$ nút, thì chúng ta chỉ cần cài đặt hai thao tác:

- Truy cập ngẫu nhiên: Nhận vào một số thứ tự $p$ và trả về nút đứng thứ $p$ trong số các nút chưa đánh dấu (theo thứ tự giữa)
- Đánh dấu: Đánh dấu một nút tương ứng với một người bị loại

Vậy có thể biểu diễn danh sách bằng một cây nhị phân gần hoàn chỉnh **dựng sẵn**. Cụ thể là chúng ta tổ chức dữ liệu trong các mảng sau:

- Mảng $Tree[1 \dots n]$ để biểu diễn cây nhị phân gồm $n$ nút có gốc là nút 1, ta quy định nút thứ $i$ có nút con trái là $2i$ và nút con phải là $2i + 1$, nút cha của nút $j$ là nút $\lfloor j/2 \rfloor$. Cây này ban đầu sẽ được duyệt theo thứ tự giữa và các phần tử $1,2, \dots, n$ sẽ được điền lần lượt vào cây (mảng $Tree$) theo thứ tự giữa.
- Mảng $Marked[1 \dots n]$ để đánh dấu, trong đó $Marked[i] = True$ nếu nút thứ $i$ đã bị đánh dấu, ban đầu mảng $Marked[1 \dots n]$ được khởi tạo bằng toàn giá trị $False$.
- Mảng $Size[1 \dots n]$ trong đó $Size[i]$ là số nút chưa bị đánh dấu trong nhánh cây gốc $i$. Mảng $Size[1 \dots n]$ cũng được khởi tạo ngay trong quá trình dựng cây.

Phép truy cập ngẫu nhiên - nhận vào một số thứ tự $p$ và trả về chỉ số nút đứng thứ $p$ chưa bị đánh dấu theo thứ tự giữa - sẽ được thực hiện như sau: Bắt đầu từ nút gốc $x = 1$, gọi $LeftSize$ là số nút chưa đánh dấu trong cây con trái của $x$. Nếu nút $x$ chưa bị đánh dấu và $p = LeftSize + 1$ thì trả về ngay nút $x$ và dừng ngay, nếu không thì quá trình tìm kiếm sẽ tiếp tục trên cây con trái ($p \leq LeftSize$) hoặc cây con phải của $x$ ($p > LeftSize$).

Phép đánh dấu một nút $x$ chỉ đơn thuần gán $Marked[x] \coloneqq True$, sau đó đi ngược từ $x$ lên gốc cây, đi qua nút $y$ nào thì giảm $Size[y]$ đi 1 để giữ tính đồng bộ.

❑ ***Cài đặt***

💻 JOSEPHUS.PAS ✓ Tìm hoán vị Josephus

```
{$MODE OBJFPC}
program JosephusPermutation;
const max = 100000;
var
  n, m: Integer;
  tree: array[1..max] of Integer;
  Marked: array[1..max] of Boolean;
  size: array[1..max] of Integer;
procedure BuildTree; //Dựng sẵn cây nhị phân gần hoàn chỉnh gồm n nút
var Person: Integer;
  procedure InOrderTraversal(Node: Integer);
  //Duyệt cây theo thứ tự giữa
  begin
    if Node > n then Exit;
    InOrderTraversal(Node * 2); //Duyệt nhánh trái
    Inc(Person);
    tree[Node] := Person; //Điền phần tử kế tiếp vào nút
    InOrderTraversal(Node * 2 + 1); //Duyệt nhánh phải
    //Xây dựng xong nhánh trái và nhánh phải thì bắt đầu tính trường size
    size[Node] := 1;
    if Node * 2 <= n then
       Inc(size[Node], size[Node * 2]);
    if Node * 2 + 1 <= n then
       Inc(size[Node], size[Node * 2 + 1]);
```

```
  end;
begin
  Person := 0;
  InOrderTraversal(1);
  FillChar(Marked, SizeOf(Marked), False);
  //Tất cả các nút đều chưa đánh dấu
end;
//Truy cập ngẫu nhiên, nhận vào số thứ tự p, trả về nút đứng thứ p trong số các nút chưa đánh dấu
function NodeAt(p: Integer): Integer;
var LeftSize: Integer;
begin
  Result := 1; //Bắt đầu từ gốc
  repeat
    //Tính số nút trong nhánh con trái
    if Result * 2 <= n then
       LeftSize := size[Result * 2]
    else LeftSize := 0;
    if not Marked[Result] and (LeftSize + 1 = p) then
       Break; //Nút Result chính là nút thứ p, dừng
    if LeftSize >= p then
      Result := Result * 2 //Tìm tiếp trong nhánh trái
    else
      begin
        Dec(p, LeftSize);
        //Trước hết tính lại số thứ tự tương ứng trong nhánh phải
        if not Marked[Result] then Dec(p);
        Result := Result * 2 + 1; //Tìm tiếp trong nhánh phải
      end;
  until False;
end;
procedure SetMark(Node: Integer); //Đánh dấu một nút
begin
  Marked[Node] := True; //Đánh dấu
  while Node > 0 do //Đồng bộ hóa trường size của các nút tiền bối
    begin
      Dec(size[Node]);
      Node := Node div 2; //Đi lên nút cha
    end;
```

```
end;
procedure FindJosephusPermutation;
var Node, p, k: Integer;
begin
  p := 1;
  for k := n downto 1 do
    begin //Danh sách có k người
      p := (p + m - 2) mod k + 1; //Xác định số thứ tự của người bị loại
      Node := NodeAt(p); //Tìm nút chứa người bị loại
      Write(tree[Node], ' '); //In ra số hiệu người bị loại
      SetMark(Node); //Đánh dấu nút tương ứng trên cây
    end;
end;
begin
  Readln(n, m);
  BuildTree;
  FindJosephusPermutation;
  WriteLn;
end.
```

### ❑ *Tìm người cuối cùng còn lại ★*

Một bài toán khác liên quan tới bài toán Josephus là cho trước hai số nguyên dương $n, m$, hãy tìm người cuối cùng bị loại. Ta có thể sử dụng thuật toán tìm hoán vị Josephus và in ra phần tử cuối cùng trong hoán vị. Tuy nhiên có thuật toán quy hoạch động hiệu quả hơn để tìm người cuối cùng bị loại. Thuật toán dựa trên công thức truy hồi sau:

$$f(n) = \begin{cases} 1, \text{nếu } n = 1 \\ (f(n-1) - 1 + m) \bmod n + 1, \text{nếu } n > 1 \end{cases} \quad (7.1)$$

Trong đó $f(n)$ là chỉ số người bị loại cuối cùng trong trò chơi với trò chơi gồm $n$ người. Công thức truy hồi (7.2) có thể giải trong thời gian $\Theta(n)$ bằng một đoạn chương trình đơn giản.

```
Input → n, m;
f := 1;
for i := 2 to n do
  f := (f - 1 + m) mod i + 1;
```

```
Output ← f;
```

### *7.2. Thứ tự thống kê động*

**Nhắc lại:** Bài toán thứ tự thống kê: Cho một tập $S$ gồm $n$ đối tượng, mỗi đối tượng có một khóa sắp xếp. Hãy cho biết nếu sắp xếp $n$ đối tượng theo thứ tự tăng dần của khóa thì đối tượng thứ $k$ là đối tượng nào?.

Chúng ta đã biết thuật toán tìm thứ tự thống kê trong thời gian $O(n)$. Tuy nhiên trong trường hợp tập $S$ liên tục có những sự thay đổi phần tử (thêm vào hay bớt đi một đối tượng), đồng thời có rất nhiều truy vấn về thứ tự thống kê đi kèm với những sự thay đổi đó thì thuật toán này tỏ ra không hiệu quả. Chúng ta cần có phương pháp tốt hơn đối với bài toán thứ tự thống kê động (Dynamic Order Statistics).

Cây tìm kiếm nhị phân là một cách để giải quyết hiệu quả vấn đề này. Chúng ta lưu trữ các đối tượng của $S$ trong một BST, mỗi nút chứa một đối tượng với khóa so sánh chính là khóa của đối tượng chứa trong.

Mỗi đối tượng $i$ được gắn với một con trỏ $ptr[i]$ tới nút tương ứng trên BST, con trỏ này được cập nhật mỗi khi có phép thêm/bớt đối tượng. Tại mỗi nút ta duy trì số nút trong nhánh con đó bằng trường $size$, trường này được cập nhật mỗi khi cấu trúc BST bị thay đổi (chèn/xóa/quay). Khi đó phép thêm và bớt đối tượng được thực hiện tự nhiên bằng phép chèn và xóa trên BST ($O(\lg n)$. Dựa vào trường $size$ mỗi nút, mỗi truy vấn về thứ tự thống kê được trả lời trong thời gian $O(\lg n)$ (Xem lại mục 0 về cách sử dụng trường $Size$).

### *7.3. Interval tree*

Cây chứa khoảng (Interval tree) là một cấu trúc dữ liệu để lưu trữ một tập các khoảng trên trục số.

Có ba loại khoảng trên trục số: khoảng đóng (closed interval), khoảng mở (open interval) và khoảng nửa mở.

$$[s, f] = \{x \in \mathbb{R}: s \leq x \leq f\}$$
$$(s, f) = \{x \in \mathbb{R}: s < x < f\}$$
$$[s, f) = \{x \in \mathbb{R}: s \leq x < f\}$$
$$(s, f] = \{x \in \mathbb{R}: s < x \leq f\}$$

Khoảng đóng còn có tên gọi là đoạn. Ở những vấn đề trong phần này chúng ta quan tâm tới khoảng đóng, nếu muốn làm việc với khoảng mở hoặc khoảng nửa mở cần có một số sửa đổi nhỏ. Chúng ta quy định thêm là với một khoảng đóng $[s, f]$ bất kỳ thì $s \leq f$.

***Định lý 7-1***

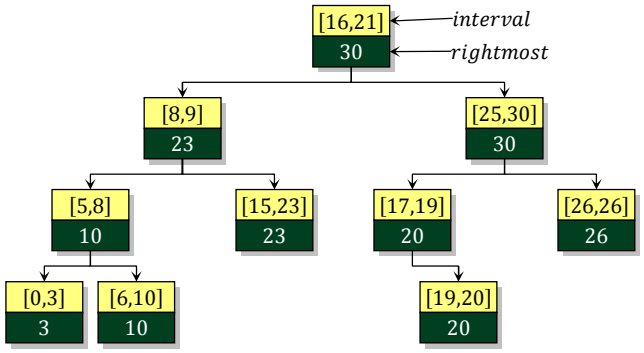
Với hai đoạn $i_1 = [s_1, f_1]$ và $i_2 = [s_2, f_2]$, đúng một trong ba mệnh đề dưới đây thỏa mãn (interval trichotomy):

- $i_1$ và $i_2$ gối nhau (overlap), tức là hai đoạn $i_1$ và $i_2$ có điểm chung
- $i_1$ nằm bên trái $i_2$: $f_1 < s_2$
- $i_1$ nằm bên phải $i_2$: $f_2 < s_1$

Interval tree bản chất là một BST, mỗi nút chứa một đoạn và khóa so sánh là đầu mút trái của mỗi đoạn. Tức là nếu duyệt cây theo thứ tự giữa ta sẽ liệt kê được tất cả các đoạn theo thứ tự tăng dần của đầu mút trái.

Tại mỗi nút $x$, ta lưu trữ thêm một trường $rightmost$: Giá trị lớn nhất của các đầu mút phải của các đoạn nằm trong nhánh cây gốc $x$. Nút giả $nilT$^ có trường $rightmost = -\infty$. Hình 1.28 là ví dụ về cây chứa 10 đoạn:

[16,21]; [8,9]; [25,30]; [5,8]; [15,23]; [17,19]; [26,26]; [0,3]; [6,10]; [19,20]




***Hình 1.28. Interval tree***

Cấu trúc nút của Interval tree có thể đặc tả như sau:

```
type
  PNode = ^TNode;
  TNode = record
    s, f: Real; //s: đầu mút trái, f: đầu mút phải
```

```
    rightmost: Real; //Thông tin phụ trợ
    parent, left, right: PNode;
  end;
var
  sentinel: TNode;
  nilT, root: PNode;
begin
  sentinel.rightmost := -∞;
  nilT := @sentinel;
  root := nilT;
  ...
end.
```

Trường $rightmost$ của mỗi nút $x^$ được tính theo công thức truy hồi:

$$x^.rightmost := \max\begin{cases} x^.left^.rightmost \\ x^.f \\ x^.right^.rightmost \end{cases} \quad (7.2)$$

Khi một nút $x^$ được chèn vào thành một nút lá hay bị xóa khỏi BST, tất cả các trường $rightmost$ trong $x^$ và các nút tiền bối của $x^$ phải được cập nhật lại. Nếu bạn cài đặt Interval tree bằng Treap hay một dạng cây nhị phân tìm kiếm tự cân bằng, cần chú ý cập nhật lại trường $rightmost$ sau phép quay cây ($UpTree$).

### a) Tìm đoạn có giao với một đoạn cho trước

Bài toán đặt ra là cho một tập $S$ gồm $n$ đoạn. Cho một đoạn $[a, b]$, hãy chỉ ra một đoạn của $S$ có giao với (hay gối lên) đoạn $[a, b]$. Một dạng truy vấn cụ thể hơn là hãy chỉ ra một đoạn của $S$ chứa một điểm $x$ cho trước. Bài toán này có thể quy về bài toán tổng quát với $[a, b] = [x, x]$

Dĩ nhiên ta có thể trả lời truy vấn này trong thời gian $O(n)$: Duyệt tất cả các đoạn của $S$ và sử dụng hàm $Overlapped$ dưới đây để tìm cũng như liệt kê các đoạn gối lên đoạn $it$. Hàm $Overlapped$ nhận vào 2 đoạn $[s_1, f_1], [s_2, f_2]$ và trả về giá trị $True$ nếu hai đoạn gối nhau:

```
function Overlapped(s1, f1, s2, f2: Real): Boolean;
begin
  Result := (s1 ≤ f2) and (s2 ≤ f1);
```

```
end;
```

Tuy vậy nếu tập $S$ liên tục có sự biến động (thêm/bớt) các đoạn thì phương pháp này tỏ ra không hiệu quả. Sử dụng Interval tree cho phép thực hiện thêm/bớt đoạn và trả lời truy vấn này hiệu quả hơn.

Xây dựng Interval tree chứa tất cả các đoạn của tập $S$. Bắt đầu từ nút $x\hat{} = root\hat{}$, nếu đoạn trong $x\hat{}$ có giao với $[a, b]$ thì xong. Ngược lại, nếu $x\hat{}.left\hat{}.rightmost \geq a$, ta quy về tìm trong nhánh con trái của $x\hat{}$, nếu không ta quy về tìm trong nhánh con phải của $x\hat{}$:

```
//Trả về nút chứa đoạn giao với [a, b], trả về nilT nếu không thấy
function IntervalSearch(a, b: Real): PNode;
begin
  Result := root; //Bắt đầu từ gốc
  while (Result ≠ nilT) and
        not Overlapped(Result^.s, Result^.f, a, b) do
        //Result chứa đoạn không giao với [a, b]
    if (Result^.left^.rightmost ≥ a) then
       Result := Result^.left //Sang trái
    else Result := Result^.right; //Sang phải
end;
```

Tính đúng đắn của thuật toán được chỉ ra trong hai nhận xét sau:

- Nếu $x\hat{}.left\hat{}.rightmost \geq a$ thì chỉ cần tìm trong nhánh con trái của $x\hat{}$ là đủ, bởi nếu tìm trong nhánh con trái của $x\hat{}$ không thấy thì chắc chắn tìm trong nhánh con phải cũng thất bại.
- Nếu $x\hat{}.left\hat{}.rightmost < a$ thì nhánh con trái của $x\hat{}$ chắc chắn không chứa đoạn nào có giao với $[a, b]$.

Thời gian thực hiện giải thuật $IntervalSearch$ là $O(h)$ với $h$ là chiều cao của Interval tree. Các phép thêm/bớt đoạn trong tập $S$ được xử lý như trên BST, sau đó cập nhật các trường $rightmost$, cũng có thời gian thực hiện $O(h)$ ($= O(\lg n)$ nếu sử dụng một dạng BST tự cân bằng).

### b) Tìm đoạn đầu tiên có giao với một đoạn cho trước

Trong một số trường hợp chúng ta cần tìm nút đầu tiên trên Interval tree (theo thứ tự giữa) chứa đoạn có giao với đoạn $[a, b]$. Điều này được thực hiện dựa trên một hàm đệ quy $FirstIntervalSearch$ như sau:

```
//Tìm đoạn đầu tiên giao với [a, b] trong nhánh cây gốc x^
function FirstIntervalSearch(x: PNode;
                             a, b: Real): PNode;
begin
  Result := nilT;
  if x = nilT then Exit; //Nhánh rỗng thì trả về nilT
  if x^.left^.rightmost ≥ a then //Nếu có thì phải có trong nhánh trái
    Result := FirstIntervalSearch(x^.left, a, b);
  else //Nhánh trái chắc chắn không có
    if Overlapped(x^.s, x^.f, a, b) then
    //Đoạn trong x^ có giao với [a, b]
      Result := x
    else //Nếu có thì chỉ có trong nhánh phải
      Result := FirstIntervalSearch(x^.right, a, b);
end;
```

### c) Liệt kê các đoạn có giao với một đoạn cho trước

Bài toán đặt ra là cho tập $S$ gồm $n$ đoạn, hãy liệt kê các đoạn có giao với đoạn $[a, b]$ cho trước. Dĩ nhiên chúng ta có thể duyệt tất cả các đoạn của $S$ và dùng hàm $Overlapped$ để liệt kê các đoạn thỏa mãn trong thời gian $\Theta(n)$, trên thực tế không có thuật toán nào tốt hơn trong trường hợp xấu nhất: Tất cả các đoạn của $S$ đều có giao với $[a, b]$.

Tuy nhiên chúng ta có thể tìm một thuật toán khác mà thời gian thực hiện giải thuật phụ thuộc vào số đoạn được liệt kê và ít phụ thuộc vào giá trị của $n$.

Trước hết ta xây dựng Interval tree chứa các đoạn của $S$. Sau đó sử dụng thủ tục $ListIntervals(Root, a, b)$ để liệt kê. Thủ tục $ListIntervals$ được cài đặt như sau:

```
//Liệt kê các đoạn có giao với [a, b] trong nhánh cây gốc x^
procedure ListIntervals(x: PNode; a, b: Real);
begin
  if x = nilT then Exit;
  if x^.left^.rightmost >= a then //Trong nhánh con trái có thể có
    ListIntervals(x^.left); //Liệt kê trong nhánh con trái
  if Overlapped(x^.s, x^.f, a, b) then //Đoạn chứa trong x^ có giao
    Output ← [x^.s, x^.f]; //Liệt kê
  if x^.s <= b then //Trong nhánh con phải có thể có
```

```
    ListIntervals(x^.right); //Liệt kê trong nhánh con phải
end;
```

Thời gian thực hiện giải thuật là $O(\lg n + m)$ với $m$ là số nút được liệt kê

## *7.4. Tóm tắt về kỹ thuật cài đặt*

Còn rất nhiều ứng dụng khác liên quan tới cấu trúc cây nhị phân, nhưng chỉ với một số ứng dụng kể trên, ta có thể thấy rằng cây nhị phân là một cấu trúc dữ liệu tốt để biểu diễn danh sách: Bằng cơ chế đánh số nút theo thứ tự giữa, chúng ta có hình ảnh một danh sách với các nút được sắp thứ tự, qua đó có thể cài đặt các phép chèn/xóa và truy cập ngẫu nhiên rất hiệu quả.

Tại sao lại là thứ tự giữa mà không phải thứ tự trước hay thứ tự sau? Mặc dù các thao tác cơ bản này vẫn có thể cài đặt nếu các nút của cây được đánh số theo thứ tự trước (hoặc sau), nhưng chúng ta sẽ gặp phải khó khăn khi thực hiện thao tác cân bằng cây. Hiện tại hầu hết các kỹ thuật cân bằng cây nhị phân (trên cây AVL, cây đỏ đen, cây Splay hay Treap) đều dựa vào phép quay, mà phép quay thì không bảo toàn thứ tự trước và thứ tự sau của các nút. Nếu như chúng ta không cần sử dụng phép quay (như bài toán tìm hoán vị Josephus) thì hoàn toàn có thể đánh số các nút trên cây theo thứ tự trước hoặc thứ tự sau.

Một chú ý quan trọng nữa là cơ chế lưu trữ và đồng bộ hóa thông tin phụ trợ. Thông thường đối với các bài toán sử dụng cây nhị phân, mỗi nút sẽ có chứa một thông tin để hỗ trợ quá trình tìm kiếm trên cây (tại mỗi bước thì đi tiếp sang nhánh trái hay nhánh phải). Như ở ví dụ cây biểu diễn danh sách chúng ta sử dụng trường $size$ chứa số nút trong một nhánh cây, hay ở ví dụ Interval tree, chúng ta sử dụng trường $rightmost$ để chứa đầu mút phải lớn nhất của một đoạn nằm trong nhánh cây. Thông tin phụ trợ sẽ được cập nhật mỗi khi có sự thay đổi cấu trúc để giữ tính đồng bộ. Có hai nguyên lý chọn thông tin phụ trợ: Thứ nhất, thông tin phụ trợ ở mỗi nút phải là thông tin **tổng hợp từ tất cả các nút** trong nhánh đó, thứ hai, tuy là sự tổng hợp thông tin từ tất cả các nút trong nhánh nhưng thông tin phụ trợ có thể **tính được chỉ bằng thông tin ở gốc nhánh và hai nút con**.

Những ràng buộc như vậy đảm bảo cho quá trình đồng bộ thông tin phụ trợ không làm tăng cấp phức tạp của thời gian thực hiện giải thuật. Việc thay đổi

thông tin phụ trợ ở một nút chỉ kéo theo việc cập nhật lại thông tin phụ trợ ở những nút tiền bối mà thôi[*].

Chú ý cuối cùng là nếu như biết trước cây nhị phân sẽ chỉ chứa các phần tử trong một tập hữu hạn $S$, đồng thời có cách nào đó tránh không phải cài đặt phép chèn và xóa thì có thể dựng sẵn một cây nhị phân gần hoàn chỉnh gồm $|S|$ nút. Khi đó chúng ta loại bỏ được các con trỏ liên kết và không cần thực hiện các phép cân bằng cây – hai thứ dễ gây nhầm lẫn nhất trong việc cài đặt cây nhị phân.

Ta xét một ví dụ cuối cùng trước khi kết thúc bài.

## *7.5. Điểm giao nhiều nhất*

### a) Trường hợp một chiều

Bài toán tìm điểm giao nhiều nhất (point of Maximum Overlap – POM) (một chiều) phát biểu như sau: Cho $n$ khoảng đóng trên trục số, khoảng đóng thứ $i$ là $[s_i, f_i]$ ($s_i \leq f_i$), hãy tìm một điểm trên trục số thuộc nhiều khoảng nhất trong số $n$ khoảng đã cho.

Thuật toán để giải quyết bài toán POM một chiều khá đơn giản: Với một khoảng đóng $[s_i, f_i]$, ta gọi $s_i$ là đầu mút mở và $f_i$ là đầu mút đóng. Sắp xếp $2n$ đầu mút của các khoảng đã cho từ trái qua phải (từ nhỏ đến lớn), nếu nhiều đầu mút ở cùng tọa độ thì tất cả đầu mút mở tại vị trí đó được xếp các đầu mút đóng. Khởi tạo một biến đếm bằng 0 và duyệt các đầu mút theo thứ tự đã sắp xếp, gặp đầu mút mở thì biến đếm tăng 1 còn gặp đầu mút đóng thì biến đếm giảm 1. Quá trình kết thúc, biến đếm trở lại thành 0, nơi biến đếm đạt cực đại chính là điểm cần tìm. Giá trị cực đại của bộ đếm đạt được chính là số khoảng đóng phủ qua điểm đó.

Tuy vậy, thuật toán trên không thực sự hiệu quả nếu tập các khoảng đóng liên tục biến động đi kèm với những truy vấn về POM. Chúng ta cần xây dựng cấu

[*] Có những bài toán cụ thể sử dụng cây nhị phân mà thông tin phụ trợ không tuân theo hai nguyên lý này, nhưng cần đến một cơ chế đồng bộ hóa đặc biệt.

trúc dữ liệu để hỗ trợ các phép thêm/bớt khoảng và trả lời các truy vấn về điểm giao nhiều nhất hiệu quả hơn.

Giải pháp là ta lưu trữ các đầu mút trong một BST sao cho nếu duyệt BST theo thứ tự giữa thì ta sẽ được thứ tự sắp xếp nói trên (đầu mút nhỏ hơn xếp trước và đầu mút mở được xếp trước đầu mút đóng nếu ở cùng vị trí trên trục số).

Thông tin phụ trợ thứ nhất trong mỗi nút $x\hat{}$ là nhãn $sign$. Ở đây $x\hat{}.sign = +1$ nếu nút chứa đầu mút mở và $x\hat{}.sign = -1$ nếu nút chứa đầu mút đóng.

Thông tin phụ trợ thứ hai trong mỗi nút $x\hat{}$ là trường $sum$: Tổng của tất cả các nhãn trong các nút nằm trong nhánh cây gốc $x\hat{}$. Trường $sum$ được tính theo công thức:

$$x\hat{}.sum \coloneqq x\hat{}.left\hat{}.sum + x\hat{}.sign + x\hat{}.right\hat{}.sum \tag{7.3}$$

Thông tin phụ trợ thứ ba trong mỗi nút $x\hat{}$ là trường $maxsum$: Cho biết nếu ta duyệt nhánh cây gốc $x\hat{}$ theo thứ tự giữa và cộng dồn lần lượt các trường $sign$ ở mỗi nút thì giá trị lớn nhất đạt được trong quá trình cộng là bao nhiêu. Trường $maxsum$ được tính theo công thức:

$$x\hat{}.maxsum \coloneqq \max\begin{cases} x\hat{}.left\hat{}.maxsum \\ x\hat{}.left\hat{}.sum + x\hat{}.sign \\ x\hat{}.Left\hat{}.sum + x\hat{}.sign + x\hat{}.right\hat{}.maxsum \end{cases} \tag{7.4}$$

Công thức truy hồi (7.3) khá dễ hiểu, ta sẽ phân tích tính đúng đắn của công thức truy hồi (7.4). Rõ ràng trong quá trình cộng dồn các trường $sign$ ở mỗi nút vào một biến đếm, giá trị cực đại của biến đếm này có thể bằng:

- Giá trị lớn nhất đạt được khi cộng xong nhánh con trái, khi đó $x\hat{}.maxsum = x\hat{}.left\hat{}maxsum$.
- Giá trị đạt được khi cộng tới nút $x\hat{}$, khi đó $x\hat{}.maxsum = x\hat{}.left\hat{}.sum + x\hat{}.sign$
- Giá trị đạt được khi cộng tới một nút nào đó ở nhánh con phải, khi đó $x\hat{}.maxsum = x\hat{}.left\hat{}.sum + x\hat{}.sign + x\hat{}.right\hat{}.maxsum$

Vậy ta có thể lấy giá trị lớn nhất trong ba khả năng này để gán cho $x\hat{}.maxsum$.

Để tiện hơn trong việc trả lời truy vấn POM, tại mỗi nút ta lưu trữ một trường $POM$ chứa điểm mà tại đó quá trình cộng dồn các trường $sign$ đạt cực đại (bằng $maxsum$). Phép cập nhật $POM$ được thực hiện song song với quá trình tính $maxsum$: Tùy theo $maxsum$ đạt tại nhánh trái, chính nút $x$^ hay nhánh phải, ta sẽ cập nhật lại trường $POM$ của $x$^.

Có thể nhận thấy rằng trường $sum$ và $maxsum$ tuy là thông tin tổng hợp từ tất cả các nút trong một nhánh, nhưng có thể tính được chỉ dựa vào thông tin trong nút gốc và hai nút con. Tức là ta có thể duy trì và đồng bộ thông tin phụ trợ trong mỗi nút sau mỗi phép chèn/xóa mà không làm tăng cấp phức tạp của thời gian thực hiện giải thuật.

Vậy thì nếu $n$ là số nút trên BST,

- Chèn một đoạn $[s, f]$ vào tập trừu tượng các khoảng đóng tương ứng với chèn một đầu mút $s$ với $sign = 1$ vào BST và chèn một đầu mút $f$ với $sign = -1$ vào BST. Thời gian $O(\lg n)$.
- Xóa một đoạn $[s, f]$ tương ứng với xóa đi đầu mút $s$ và đầu mút $f$ khỏi BST. Thời gian $O(\lg n)$.
- Trả lời truy vấn POM, chỉ cần truy xuất $root$^.$POM$. Thời gian $O(1)$.

### b) Trường hợp hai chiều

Bài toán tìm điểm giao nhiều nhất (hai chiều) được phát biểu như sau: Cho $n$ hình chữ nhật đánh số từ 1 tới $n$ trong mặt phẳng trực giao $Oxy$. Các hình chữ nhật có cạnh song song với các trục tọa độ. Mỗi hình chữ nhật được cho bởi 4 tọa độ $x_1, y_1, x_2, y_2$ trong đó $(x_1, y_1)$ là tọa độ góc trái dưới và $(x_2, y_2)$ là tọa độ góc phải trên $(x_1 < x_2, y_1 < y_2)$. Hãy tìm một điểm trên mặt phẳng thuộc nhiều hình chữ nhật nhất trong số các hình chữ nhật đã cho (điểm nằm trên cạnh một hình chữ nhật vẫn tính là thuộc hình chữ nhật đó).

Bài toán POM hai chiều là một trong những ví dụ hay về kỹ thuật cài đặt, chúng ta sẽ viết chương trình đầy đủ giải bài toán POM hai chiều với khuôn dạng Input/Output như sau.

**Input**

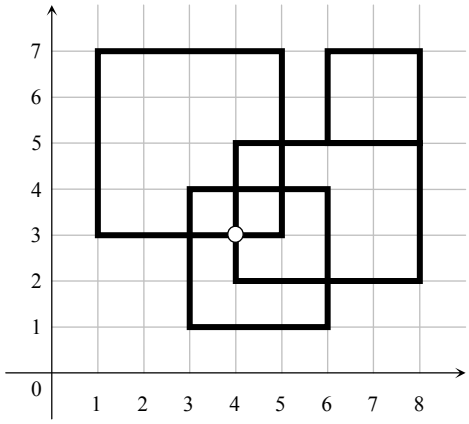
- Dòng 1 chứa số nguyên dương $n \leq 10^6$.

- $n$ dòng tiếp theo, dòng thứ $i$ chứa 4 số nguyên dương $x_1, y_1, x_2, y_2$. $(-10^9 \leq x_1 < x_2 \leq 10^9; -10^9 \leq y_1 < y_2 \leq 10^9)$.

**Output**

Điểm giao nhiều nhất và số hình chữ nhật chứa điểm giao nhiều nhất.

| Sample Input | Sample Output |
|---|---|
| 4<br>1 3 5 7<br>3 1 6 4<br>4 2 8 5<br>6 5 8 7 | point of Maximum Overlap: (4, 3)<br>Number of Rectangles: 3 |



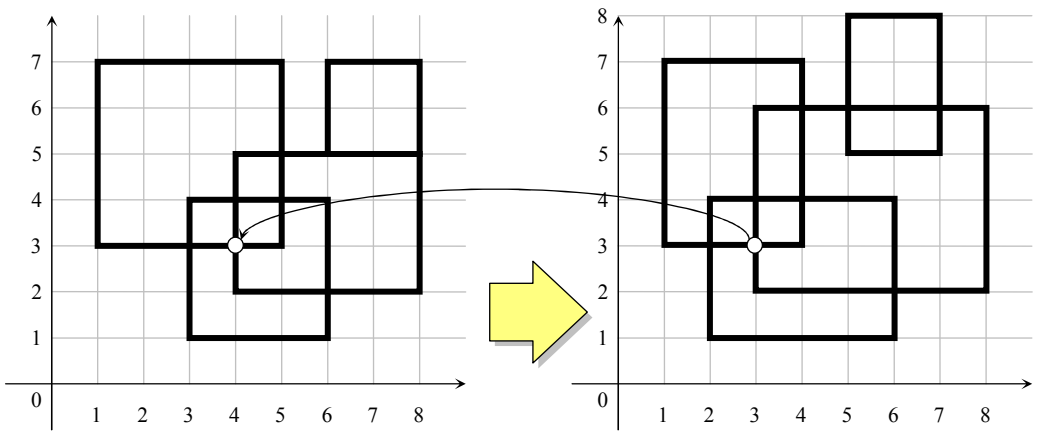

❑ ***Biến đổi tọa độ***

Mỗi hình chữ nhật tương ứng với hai cạnh ngang (cạnh đáy và cạnh đỉnh) và hai cạnh dọc (cạnh trái và cạnh phải). Như vậy có tất cả $2n$ cạnh ngang và $2n$ cạnh dọc. Sắp xếp hai dãy tọa độ này theo quy tắc sau:

- Dãy $2n$ cạnh dọc được xếp theo thứ tự tăng dần theo hoành độ (trái qua phải), nếu nhiều cạnh dọc ở cùng hoành độ thì những cạnh trái được xếp trước những cạnh phải.
- Dãy $2n$ cạnh ngang được xếp theo thứ tự tăng dần theo tung độ (dưới lên trên), nếu nhiều cạnh ngang ở cùng tung độ thì những cạnh đáy được xếp trước những cạnh đỉnh.

Sau đó ta ánh xạ hoành độ mỗi cạnh ngang cũng như tung độ mỗi cạnh dọc thành chỉ số của nó trong hai dãy đã sắp xếp.

Tọa độ những hình chữ nhật ban đầu sẽ bị biến đổi qua ánh xạ này, tuy nhiên ta có thể tìm POM trong tập các hình chữ nhật mới và ánh xạ ngược tọa độ POM thành tọa độ ban đầu. Ví dụ:




***Hình 1.19. Phép ánh xạ tọa độ***

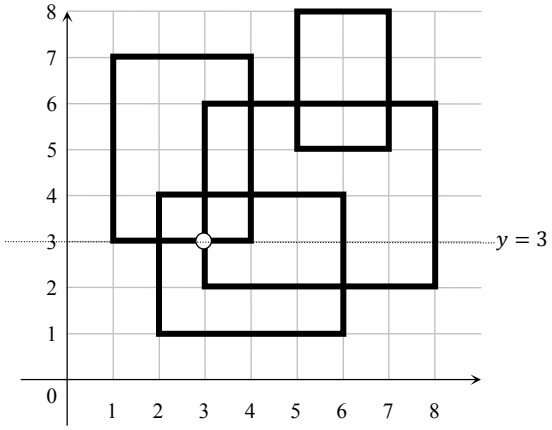
Phép biến đổi này có những công dụng:

- Tất cả các hoành độ của cạnh dọc cũng như tất cả các tung độ của cạnh ngang trở thành $2n$ số nguyên *hoàn toàn phân biệt*.
- Cho dù tọa độ của các hình chữ nhật ban đầu có thể rất lớn, hoặc là số thực, qua ánh xạ này chúng ta sẽ *chỉ xử lý các tọa độ nguyên* $1,2,\dots,2n$.
- Tọa độ POM có thể *ánh xạ ngược lại dễ dàng*. Bởi khi xác định được điểm giao nhiều nhất nằm trên đường ngang thứ mấy và đường dọc thứ mấy, ta có thể chiếu vào hai dãy tọa độ ban đầu để tìm tọa độ trước khi ánh xạ.

Bạn có thể thắc mắc rằng POM có thể không nằm trên đường ngang cũng như đường dọc nào (như ví dụ trên có thể ta tìm được POM là $(3.5,3.5)$ trên bản đồ ánh xạ). Khi đó ta có thể xác định POM nằm giữa hai đường ngang liên tiếp nào và nằm giữa hai đường dọc liên tiếp nào, sau đó ánh xạ ngược lại. Tuy nhiên không cần phải rắc rối như vậy, thuật toán mà chúng ta sẽ trình bày luôn tìm được POM nằm trên giao của một đường ngang và một đường dọc.

❑ ***Đường quét ngang***

G điểm giao nhiều nhất có tọa độ $(a, b)$, khi đó nếu ta xét đường thẳng $y = b$, nó sẽ cắt ngang qua một số hình chữ nhật. Giao của các hình chữ nhật với đường thẳng $y = b$ tạo thành các khoảng đóng trên đường thẳng đó. Khi đó tọa độ $a$ chính là điểm giao nhiều nhất của các khoảng đóng (một chiều).




Ví dụ như với bản đồ trên, ta xét đường thẳng $y = 3$, nó sẽ cắt ngang qua 3 hình chữ nhật tạo thành 3 khoảng đóng: [1,4]; [2,6]; [3,8]. Điểm giao nhiều nhất của 3 khoảng đóng này là điểm $x = 3$ (hay bất cứ điểm nào nằm trong đoạn [3,4]). Vậy POM tương ứng có tọa độ $(3,3)$.

Thuật toán tìm POM hai chiều có thể trình bày như sau: Xét tất cả các đường thẳng nằm ngang có phương trình $y = b$, với mỗi đường đó ta xét các giao với các hình chữ nhật đã cho và tìm điểm giao nhiều nhất, ghi nhận lại điểm giao nhiều nhất trên tất cả các giá trị $b$ đã thử.

Chúng ta sẽ phải thử với bao nhiêu đường thẳng dạng $y = b$?, có thể nhận thấy rằng chỉ cần thử với lần lượt các giá trị $b = 1,2, \dots ,2n$ là đủ.

Rõ ràng với $b = 0$, tập các khoảng đóng tạo ra trên đường thẳng $y = 0$ là rỗng. Sau khi xét xong mỗi đường thẳng $y = b$, xét tiếp đến đường thẳng $y = b + 1$, có hai khả năng xảy ra:

- Nếu đường $y = b + 1$ là một cạnh đáy của hình chữ nhật $(x_1, y_1, x_2, y_2)$ $(y_1 = b + 1)$, ta bổ sung $[x_1, x_2]$ vào tập các khoảng đóng (thời gian $O(\lg n)$) và tìm POM $(O(1))$.

- Nếu đường $y = b + 1$ là một cạnh đỉnh của hình chữ nhật $(x_1, y_1, x_2, y_2)$ $(y_2 = b + 1)$, ta loại bỏ $[x_1, x_2]$ khỏi tập các khoảng đóng (thời gian $O(\lg n)$).

❑ ***Dựng sẵn BST***

Nhắc lại trong kỹ thuật cài đặt bài toán tìm POM một chiều, ta sử dụng BST chứa các đầu mút của các khoảng đóng. Tuy nhiên do các đầu mút là các số nguyên hoàn toàn phân biệt trong phạm vi $1,2,\ldots,2n$, nên ta có thể dựng sẵn cây nhị phân chứa $2n$ nút, mỗi nút sẽ chứa một đầu mút với nhãn $Sign = +1$ nếu đó là đầu mút mở, $Sign = -1$ nếu đó là đầu mút đóng và $Sign = 0$ để đánh dấu đầu mút đó không có hiệu lực (do khoảng đóng tương ứng chưa được xét đến hoặc đã bị loại bỏ). Có thể thấy rằng cách gán giá trị $Sign = 0$ này không làm ảnh hưởng đến tính đúng đắn của công thức (7.3) và (7.4).

Cây nhị phân dựng sẵn được biểu diễn bởi một mảng $tree[0 \ldots 2n]$, mỗi phần tử là một bản ghi chứa các trường $point$: tọa độ điểm, $sign$: Nhãn đầu mút $\in \{-1,0,+1\}$, $sum$, $maxsum$ và $POM$. Ý nghĩa của các trường được giải thích như trên. Nút gốc của cây là $tree[1]$. Nút $i$ có con trái là $2i$ và con phải là $2i + 1$. Hai hàm $left$ và $right$ dưới đây trả về nút con trái và con phải của một nút, (trả về 0 trong trường hợp nút $i$ không có con trái hoặc con phải)

```
function valid(x: Integer): Boolean;
begin
  Result := x <= 2 * n;
end;
function left(x: Integer): Integer;
begin
  if IsNode(x * 2) then Result := x * 2
  else Result := 0;
end;
function right(x: Integer): Integer;
begin
  if IsNode(x * 2 + 1) then Result := x * 2 + 1
  else Result := 0;
end;
```

Vì nếu nút không có con trái (phải) thì hàm $left$ ($right$) trả về 0, ta sẽ gán các trường $sum$ và $maxsum$ của $tree[0]$ bằng 0 cho tiện cài đặt.

❑ *Cài đặt*

POM.PAS ✓ Điểm giao nhiều nhất

```
{$MODE OBJFPC}
program PointOfMaximumOverlap;
const max = 100000;
type
  TEndPointType = (eptOpen, eptClose);
  TEndPoint = record //Kiểu đầu mút của khoảng đóng
    value: Integer; //Tọa độ
    ept: TEndPointType; //Loại: đầu mút mở hay đầu mút đóng
    rid: Integer; //Chỉ số hình chữ nhật tương ứng
  end;
  TRect = record //Kiểu hình chữ nhật
    x1, y1, x2, y2: Integer; //(x1, y1): Trái Dưới, (x2, y2): Phải Trên
  end;
  TEndPointArray = array[1..2 * max] of TEndPoint;
  //Kiểu danh sách các đầu mút
  TNode = record //Thông tin nút của cây nhị phân
    point: Integer;
    sign: Integer;
    sum, maxsum: Integer;
    POM: Integer;
  end;
var
  x, y: TEndPointArray;
  r: array[1..max] of TRect;
  tree: array[0..2 * max] of TNode;
  ptr: array[1..2 * max] of Integer;
  n, ResX, ResY, m: Integer;
procedure Enter; //Nhập dữ liệu
var i, j: Integer;
begin
  ReadLn(n);
  for i := 1 to n do //Đọc 2n tọa độ x và 2n tọa độ y
    begin
      j := 2 * n + 1 - i;
      Read(x[i].value);
```

```
      x[i].Ept := eptOpen;
      x[i].rid := i;
      Read(y[i].value);
      y[i].Ept := eptOpen;
      y[i].rid := i;
      Read(x[j].value);
      x[j].Ept := eptClose;
      x[j].rid := i;
      Read(y[j].value);
      y[j].Ept := eptClose;
      y[j].rid := i;
    end;
end;
operator < (const p, q: TEndPoint): Boolean;
//p sẽ được xếp trước q nếu...
begin
  Result := (p.value < q.value) or
            (p.value = q.value) and (p.Ept < q.Ept);
             //Open < Close
end;
procedure Sort(var k: TEndPointArray);
//Sắp xếp danh sách các đầu mút
  procedure Partition(L, H: Integer);
  var
    i, j: Integer;
    Pivot: TEndPoint;
  begin
    if L >= H then Exit;
    i := L + Random(H - L + 1);
    Pivot := k[i];
    k[i] := k[L];
    i := L;
    j := H;
    repeat
      while (Pivot < k[j]) and (i < j) do Dec(j);
      if i < j then
        begin
          k[i] := k[j];
          Inc(i);
```

```
        end
      else Break;
      while (k[i] < Pivot) and (i < j) do Inc(i);
      if i < j then
        begin
          k[j] := k[i];
          Dec(j);
        end
      else Break;
    until i = j;
    k[i] := Pivot;
    Partition(L, i - 1);
    Partition(i + 1, H);
  end;
begin
  Partition(1, 2 * n);
end;
procedure RefineRects; //Ánh xạ tọa độ
var i: Integer;
begin
  for i := 1 to 2 * n do
    begin
      with x[i] do
        if ept = eptOpen then r[rid].x1 := i
        else r[rid].x2 := i;
      with y[i] do
        if ept = eptOpen then r[rid].y1 := i
        else r[rid].y2 := i;
    end;
end;
function valid(x: Integer): Boolean; //Nút có hợp lệ không
begin
  Result := x <= 2 * n;
end;
function left(x: Integer): Integer; //Tìm nút con trái của Node
begin
  if valid(x * 2) then Result := x * 2 //Nút con trái hợp lệ
  else Result := 0; //Nếu không trả về 0
```

```
end;
function right(x: Integer): Integer; //Tìm nút con phải của Node
begin
  if valid(x * 2 + 1) then Result := x * 2 + 1
//Nút con phải hợp lệ
  else Result := 0; //Nếu không trả về 0
end;
procedure BuildTree; //Dựng sẵn BST gồm 2n nút
var
  i: Integer;
  procedure InOrderTraversal(x: Integer);
  //Duyệt cây theo thứ tự giữa
  begin
    if not valid(x) then Exit;
    InOrderTraversal(x * 2);
    Inc(i);
    tree[x].point := i; //Đưa điểm i vào nút Node
    ptr[i] := x;   //ptr[i]: Nút chứa điểm tọa độ i trong BST
    InOrderTraversal(x * 2 + 1);
  end;
begin
  FillByte(tree, SizeOf(tree), 0); //sum, maxsum := 0
  i := 0;
  InOrderTraversal(1);
end;
//target := Max(a, b, c)
function ChooseMax3(var target: Integer;
                    a, b, c: Integer): Integer;
begin
  target := a;
  Result := 1; //Trả về 1 nếu Max đạt tại a
  if target < b then
    begin
      target := b;
      Result := 2; //Trả về 2 nếu Max đạt tại b
    end;
  if target < c then
    begin
      target := c;
```

```
      Result := 3; //Trả về 3 nếu Max đạt tại c
    end;
end;
//Đặt nhãn sign = s cho nút mang tọa độ p trong BST
procedure SetPoint(p: Integer; s: Integer);
var
  Node, L, r, Choice: Integer;
begin
  Node := ptr[p]; //Xác định nút chứa điểm p
  tree[Node].sign := s; //Đặt nhãn sign
  repeat //Cập nhật thông tin phụ trợ từ Node lên gốc 1
    L := left(Node);
    r := right(Node); //L: Con trái; r: Con phải
    //Tính trường sum
    tree[Node].sum := tree[Node].sign + tree[L].sum
                                      + tree[r].sum;
    //Tính trường maxsum
    Choice := ChooseMax3(tree[Node].maxsum,
                         tree[L].maxsum,
                         tree[L].sum + tree[Node].sign,
                         tree[L].sum + tree[Node].sign
                                     + tree[r].maxsum);
    case Choice of //Tính POM tùy theo maxsum đạt tại đâu
      1: tree[Node].POM := tree[L].POM; //Đạt tại nhánh trái
      2: tree[Node].POM := tree[Node].point;
         //Đạt tại chính Node
      3: tree[Node].POM := tree[r].POM; //Đạt tại nhánh phải
    end;
    Node := Node div 2; //Đi lên nút cha
  until Node = 0;
end;
//Thêm một đoạn [x1, x2] vào tập cần tìm POM
procedure InsertInterval(x1, x2: Integer);
begin
  SetPoint(x1, +1); //Đặt nhãn đầu mút mở là +1
  SetPoint(x2, -1); //Đặt nhãn đầu mút đóng là -1
end;
//Loại một đoạn [x1, x2] khỏi tập cần tìm POM (đặt nhãn của hai đầu mút thành 0)
procedure DeleteInterval(x1, x2: Integer);
```

```
begin
  SetPoint(x1, 0);
  SetPoint(x2, 0);
end;
procedure Sweep; //Sử dụng các dòng quét ngang để tìm POM
var
  sweepY, POM: Integer;
begin
  BuildTree;
  m := 0;
  for sweepY := 1 to 2 * n do //Xét các đường quét y = 1, 2, ..., 2n
    with y[sweepY] do
      if ept = eptOpen then //Quét vào một cạnh đáy
        begin
          InsertInterval(r[rid].x1, r[rid].x2);
          //Thêm một đoạn vào tập tìm POM
          if tree[1].maxsum > m then
          //POM mới là giao của nhiều hình chữ nhật hơn POM cũ
            begin
              m := tree[1].maxsum;
              POM := tree[1].POM;
              //Ánh xạ ngược lại, tìm tọa độ
              ResX := x[POM].value;
              ResY := y[sweepY].value;
            end;
        end
      else //Quét vào một cạnh đỉnh
        DeleteInterval(r[rid].x1, r[rid].x2);
end;
procedure PrintResult;
begin
  WriteLn('Point of Maximum Overlap:
           (', ResX, ', ', ResY, ')');
  WriteLn('Number of Rectangles: ', m);
end;
begin
  Enter; //Nhập dữ liệu
  Sort(x);
```

```
  Sort(y); //Sắp xếp hai dãy tọa độ
  RefineRects; //Ánh xạ sang tọa độ mới
  Sweep; //Sử dụng đường quét ngang để tìm POM
  PrintResult; //In kết quả
end.
```

Thời gian thực hiện giải thuật tìm điểm giao nhiều nhất của $n$ hình chữ nhật là $O(n \lg n)$.

## Bài tập

**1.30.** Cho một BST chứa $n$ khóa, và hai khóa $a, b$. Người ta muốn liệt kê tất cả các khóa $k$ của BST thỏa mãn $a \leq k \leq b$. Hãy tìm thuật toán $O(m + \lg n)$ để trả lời truy vấn này ($m$ là số khóa được liệt kê).

**1.31.** Cho $n$ là một số nguyên dương và $x = (x_1, x_2, \dots, x_n)$ là một hoán vị của dãy số $(1,2, \dots, n)$. Với $\forall i: 1 \leq i \leq n$, gọi $t_i$ là số phần tử đứng trước giá trị $i$ mà lớn hơn $i$ trong dãy $x$. Khi đó dãy $t = (t_1, t_2, \dots, t_n)$ được gọi là dãy nghịch thế của dãy $x = (x_1, x_2, \dots, x_n)$.

Ví dụ với $n = 6$, dãy $x = (3,2,1,6,4,5)$ thì dãy nghịch thế của nó là $t = (2,1,0,1,1,0)$

Xây dựng thuật toán $O(n \lg n)$ tìm dãy nghịch thế từ dãy hoán vị cho trước và thuật toán $O(n \lg n)$ để tìm dãy hoán vị từ dãy nghịch thế cho trước.

**1.32.** Trên mặt phẳng với hệ tọa độ trực giao $Oxy$ cho $n$ hình chữ nhật có cạnh song song với các trục tọa độ. Hãy tìm thuật toán $O(n \lg n)$ để tính diện tích phần mặt phẳng bị $n$ hình chữ nhật đó chiếm chỗ.

**1.33.** Cho $n$ dây cung của một hình tròn, không có hai dây cung nào chung đầu mút. Tìm thuật toán $O(n \lg n)$ xác định số cặp dây cung cắt nhau bên trong hình tròn. (Ví dụ nếu $n$ dây cung đều là đường kính thì số cặp là $\binom{n}{2}$).

**1.35.** Trên trục số cho $n$ đoạn đóng, đoạn thứ $i$ là $[a_i, b_i]$, $(a_i, b_i \in \mathbb{N})$. Hãy chọn trên trục số một số ít nhất các điểm nguyên phân biệt sao cho có ít nhất $c_i$ điểm được chọn thuộc vào đoạn thứ $i$. $(1 \leq n \leq 10^5; 0 \leq a_i, b_i, c_i \leq 10^5)$.

**1.36.** Bản đồ một khu đất hình chữ nhật kích thước $m \times n$ được chia thành lưới ô vuông đơn vị. Trên đó có đánh dấu $k$ ô trồng cây ($m, n, k \leq 10^5$). Người ta muốn giải phóng một mặt bằng nằm trong khu đất này. Bản đồ mặt bằng có các ràng buộc sau:

- Cạnh mặt bằng là số nguyên
- Mặt bằng chiếm trọn một số ô trên bản đồ
- Cạnh mặt bằng song song với cạnh bản đồ

Hãy trả lời hai câu hỏi:

- Nếu muốn xây dựng mặt bằng với cạnh là $D$ thì phải giải phóng ít nhất bao nhiêu ô trồng cây?
- Nếu không muốn giải phóng ô trồng cây nào thì có thể xây dựng được mặt bằng với cạnh lớn nhất là bao nhiêu?

**1.37.** Một bộ $n \leq 10^5$ lá bài được xếp thành tập và mỗi lá bài được ghi số thứ tự ban đầu của lá bài đó trong tập bài (vị trí các lá bài được đánh số từ 1 tới $n$ từ trên xuống dưới).

Xét phép tráo ký hiệu bởi $S(i, j)$: rút ra lá bài thứ $i$ và chèn lên trên lá bài thứ $j$ trong số $n - 1$ lá bài còn lại ($1 \leq i, j \leq n$), quy ước rằng nếu $j = n$ thì lá bài thứ $i$ sẽ được đặt vào vị trí dưới cùng của tập bài.

Ví dụ với $n = 6$

$$(1, \boxed{2}, 3,4,5,6) \xrightarrow{S(2,3)} (1,3, \boxed{2}, 4,5,6)$$

$$(\boxed{1}, 3,2,4,5,6) \xrightarrow{S(1,2)} (3, \boxed{1}, 2,4,5,6)$$

$$(3,1,2, \boxed{4}, 5,6) \xrightarrow{S(4,5)} (3,1,2,5, \boxed{4}, 6)$$

$$(\boxed{3}, 1,2,5,4,6) \xrightarrow{S(1,6)} (1,2,5,4,6, \boxed{3})$$

Người ta tráo bộ bài bằng $x$ phép tráo ($x \leq 10^5$). Bạn được cho biết $x$ phép tráo đó, hãy sử dụng thêm ít nhất các phép tráo nữa để đưa các lá bài về vị trí ban đầu. Như ở ví dụ trên, chúng ta cần sử dụng thêm 2 phép tráo $S(6,3)$ và $S(5,4)$.

Chuyên đề 7

# ĐỒ THỊ

## 1. Đường đi ngắn nhất

### *1.1. Đồ thị có trọng số*

Trong các ứng dụng thực tế, chẳng hạn trong mạng lưới giao thông, người ta không chỉ quan tâm đến việc tìm đường đi giữa hai địa điểm mà còn phải lựa chọn một hành trình tiết kiệm nhất (theo tiêu chuẩn không gian, thời gian hay một đại lượng mà chúng ta cần giảm thiểu theo hành trình). Khi đó người ta gán cho mỗi cạnh của đồ thị một giá trị phản ánh chi phí đi qua cạnh đó và cố gắng tìm một con đường mà tổng chi phí các cạnh đi qua là nhỏ nhất.

Đồ thị có trọng số là một bộ ba $G = (V, E, w)$ trong đó $G = (V, E)$ là một đồ thị, $w$ là hàm trọng số:

$$w: E \longrightarrow \mathbb{R}$$
$$e \longmapsto w(e)$$

Hàm trọng số gán cho mỗi cạnh $e$ của đồ thị một số thực $w(e)$ gọi là trọng số (weight) của cạnh. Nếu cạnh $e = (u, v)$ thì ta cũng ký hiệu $w(u, v) = w(e)$.

Tương tự như đồ thị không trọng số, có nhiều cách biểu diễn đồ thị có trọng số trong máy tính. Nếu ta sử dụng danh sách cạnh, danh sách kề hay danh sách liên thuộc, mỗi phần tử của danh sách sẽ chứa thêm một thông tin về trọng số của cạnh tương ứng. Trường hợp biểu diễn đơn đồ thị gồm $n$ đỉnh, ta còn có thể sử dụng ma trận trọng số $W = \{w_{uv}\}_{n \times n}$ trong đó $w_{uv}$ là trọng số của cạnh $(u, v)$. Trong trường hợp $(u, v) \notin E$ thì tùy bài toán cụ thể, $w_{uv}$ sẽ được gán một giá trị đặc biệt để nhận biết $(u, v)$ không phải là cạnh (chẳng hạn có thể gán bằng $+\infty$, 0 hay $-\infty$).

Đường đi, chu trình trong đồ thị có trọng số cũng được định nghĩa giống như trong trường hợp không trọng số, chỉ có khác là độ dài đường đi không tính

bằng số cạnh đi qua, mà được tính bằng tổng trọng số của các cạnh đi qua. Độ dài của một đường đi $P$ được ký hiệu là $w(P)$.

### *1.2. Đường đi ngắn nhất xuất phát từ một đỉnh*

Bài toán tìm đường đi ngắn nhất xuất phát từ một đỉnh (*single-source shortest path*) được phát biểu như sau: Cho đồ thị có trọng số $G = (V, E, w)$, hãy tìm các đường đi ngắn nhất từ đỉnh xuất phát $s \in V$ đến tất cả các đỉnh còn lại của đồ thị. Độ dài của đường đi từ đỉnh $s$ tới đỉnh $t$, ký hiệu $\delta(s, t)$, gọi là *khoảng cách* (distance) từ $s$ đến $t$. Nếu như không tồn tại đường đi từ $s$ tới $t$ thì ta sẽ đặt khoảng cách đó bằng $+\infty$. Có một vài biến đổi khác của bài toán tìm đường đi ngắn nhất xuất phát từ một đỉnh:

- Tìm các con đường ngắn nhất từ mọi đỉnh tới một đỉnh $t$ cho trước. Bằng cách đảo chiều các cung của đồ thị, chúng ta có thể quy về bài toán tìm đường đi ngắn nhất xuất phát từ $t$.
- Tìm đường đi ngắn nhất từ đỉnh $s$ tới đỉnh $t$ cho trước. Dĩ nhiên nếu ta tìm được đường đi ngắn nhất từ $s$ tới mọi đỉnh khác thì bài toán tìm đường đi ngắn nhất từ $s$ tới $t$ cũng sẽ được giải quyết. Hơn nữa, vẫn chưa có một thuật toán nào tìm đường đi ngắn nhất từ $s$ tới $t$ mà không cần quy về bài toán tìm đường đi ngắn nhất từ $s$ tới mọi đỉnh khác.
- Tìm đường đi ngắn nhất giữa mọi cặp đỉnh của đồ thị: Mặc dù có những thuật toán đơn giản và hiệu quả để tìm đường đi ngắn nhất giữa mọi cặp đỉnh, chúng ta vẫn có thể giải quyết bằng cách thực hiện thuật toán tìm đường đi ngắn nhất xuất phát từ một đỉnh với mọi cách chọn đỉnh xuất phát.

#### a) Cấu trúc bài toán con tối ưu

Các thuật toán tìm đường đi ngắn nhất mà chúng ta sẽ khảo sát đều dựa vào một đặc tính chung: Mỗi đoạn đường trên đường đi ngắn nhất phải là một đường đi ngắn nhất.

***Định lý 1-1***

Cho đồ thị có trọng số $G = (V, E, w)$, gọi $P = \langle v_1, v_2, \dots, v_k \rangle$ là một đường đi ngắn nhất từ $v_1$ tới $v_k$, khi đó với mọi $i, j: 1 \le i \le j \le k$, đoạn đường $P_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ là một đường đi ngắn nhất từ $v_i$ tới $v_j$.

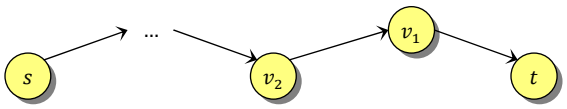
Chúng ta sẽ thấy rằng hầu hết các thuật toán tìm đường đi ngắn nhất đều là thuật toán quy hoạch động (ví dụ thuật toán Floyd) hoặc tham lam (ví dụ thuật toán Dijkstra) bởi tính chất bài toán con tối ưu nêu ra trong Định lý 1-1.

Nếu như đồ thị có chu trình âm (chu trình với độ dài âm) thì khoảng cách giữa một số cặp đỉnh nào đó có thể không xác định, bởi vì bằng cách đi vòng theo chu trình này một số lần đủ lớn, ta có thể chỉ ra đường đi giữa hai đỉnh nào đó trong chu trình này nhỏ hơn bất kỳ một số cho trước nào. Trong trường hợp như vậy, có thể đặt vấn đề tìm đường đi đơn ngắn nhất. Vấn đề đó lại là một bài toán NP-đầy đủ, hiện chưa ai chứng minh được sự tồn tại hay không một thuật toán đa thức tìm đường đi đơn ngắn nhất trên đồ thị có chu trình âm.

### b) Quy về bài toán đo khoảng cách

Nếu như đồ thị không có chu trình âm thì có thể chứng minh được rằng một trong những đường đi ngắn nhất là đường đi đơn. Khi đó chỉ cần biết được khoảng cách từ $s$ tới tất cả những đỉnh khác thì đường đi ngắn nhất từ $s$ tới $t$ có thể tìm được một cách dễ dàng qua thuật toán sau:

Trước tiên ta tìm đỉnh $v_1 \neq t$ để $\delta(s,t) = \delta(s,v_1) + c(v_1,t)$. Dễ thấy rằng luôn tồn tại đỉnh $v_1$ như vậy và đỉnh đó sẽ là đỉnh đứng liền trước $t$ trên đường đi ngắn nhất từ $s$ tới $t$. Nếu $v_1 = s$ thì đường đi ngắn nhất là đường đi trực tiếp theo cung $(s,t)$. Nếu không thì vấn đề trở thành tìm đường đi ngắn nhất từ $s$ tới $v_1$ . Và ta lại tìm được một đỉnh $v_2 \notin \{t, v_1\}$ để $\delta(s,v_1) = \delta(s,v_2) + c(v_2,t)$…Cứ tiếp tục như vậy sau một số hữu hạn bước cho tới khi xét tới đỉnh $v_k = s$, Ta có dãy $t = v_0, v_1, v_2, \dots v_k = s$ không chứa đỉnh lặp lại. Lật ngược thứ tự dãy cho ta đường đi ngắn nhất từ $s$ tới $t$.




### c) Nhãn khoảng cách và phép co

Tất cả những thuật toán chúng ta sẽ khảo sát để tìm đường đi ngắn nhất xuất phát từ một đỉnh đều sử dụng kỹ thuật gán nhãn khoảng cách: Với mỗi đỉnh $v \in V$, nhãn khoảng cách $d[v]$ là độ dài một đường đi nào đó từ $s$ tới $v$. Trong

trường hợp chúng ta chưa xác định được đường đi nào từ $s$ tới $v$, nhãn $d[v]$ được gán giá trị $+\infty$.

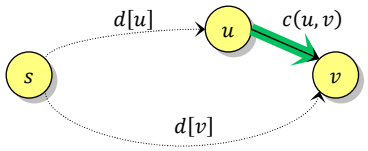
Ban đầu chúng ta chưa xác định được bất kỳ đường đi nào từ $s$ tới các đỉnh khác nên các $d[v]$ được gán giá trị khởi tạo là:

$$d[v] = \begin{cases} 0, \text{nếu } v = s \\ +\infty, \text{nếu } v \neq s \end{cases} \quad (v = 1,2,\dots,n) \qquad (1.1)$$

```
procedure Init;
begin
  for ∀v ∈ V do d[v] := +∞;
  d[s] := 0;
end;
```

Do tính chất của nhãn khoảng cách, ta có $d[v] \geq \delta(s,v), \forall v \in V$. Các thuật toán tìm đường đi ngắn nhất sẽ cực tiểu hóa dần các nhãn $d[.]$ cho tới khi $d[v] = \delta(s,v), \forall v \in V$. Trong các thuật toán mà chúng ta sẽ khảo sát, việc cực tiểu hóa các nhãn khoảng cách được thực hiện bởi các *phép co*.

Phép co theo cạnh $(u,v) \in E$, gọi tắt là phép co $(u,v)$ được thực hiện như sau: Giả sử chúng ta đã xác định được $d[u]$ là độ dài một đường đi từ $s$ tới $u$, ta nối thêm cạnh $(u,v)$ để được một đường đi từ $s$ tới $v$ với độ dài $d[u] + w(u,v)$. Nếu đường đi này có độ dài ngắn hơn $d[v]$, ta ghi nhận lại $d[v]$ bằng $d[u] + w(u,v)$. Điều này có nghĩa là nếu $s \rightsquigarrow u$ nối thêm cạnh $(u,v)$ lại ngắn hơn đường đi $s \rightsquigarrow v$ đang có, thì ta hủy bỏ đường đi $s \rightsquigarrow v$ hiện tại và ghi nhận lại đường đi $s \rightsquigarrow v$ mới là đường đi $s \rightsquigarrow u \to v$.




***Hình 2.1. Phép co***

Có thể hình dung hoạt động của phép co như sau: Căng một đoạn dây đàn hồi dọc theo đường đi $s$ tới $v$, đoạn dây sẽ dãn ra tới độ dài $d[v]$. Tiếp theo ta thử lấy đoạn dây đó căng dọc theo đường đi từ $s$ tới $u$ rồi nối tiếp đến $v$. Nếu đoạn dây bị chùng xuống (co lại) hơn so với cách căng cũ, ta ghi nhận đường đi tương

ứng với cách căng mới, nếu đoạn dây không chùng xuống (hoặc căng thêm) thì ta vẫn giữ đoạn dây đó căng theo đường cũ. Chính vì phép co không làm "dài" thêm $d[v]$, ta nói rằng $d[v]$ bị cực tiểu hóa qua phép co $(u, v)$.

Phép co $(u, v)$ được thực hiện bởi hàm $Relax$, hàm nhận vào cạnh $(u, v)$ và trả về True nếu nhãn $d[v]$ bị giảm đi qua phép co $(u, v)$:

```
function Relax(e = (u,v)∈E): Boolean;
begin
  Result := d[v] > d[u] + w(e);
  if Result then
    begin
      d[v] := d[u] + w(e); //cực tiểu hóa nhãn d[v]
      trace[v] := u; //Lưu vết đường đi
    end
end;
```

Mỗi khi $d[v]$ bị giảm xuống sau phép co $(u, v)$, ta lưu lại vết $trace[v] := u$ với ý nghĩa đường đi ngắn nhất từ $s$ tới $v$ cho tới thời điểm được ghi nhận sẽ là đường đi qua $u$ trước rồi đi tiếp theo cung $(u, v)$, vết này được sử dụng để truy vết tìm đường đi khi thuật toán kết thúc.

### d) Một số tính chất và quy ước

Các tính chất sau đây tuy đơn giản nhưng quan trọng để chứng minh tính đúng đắn của các thuật toán trong bài:

- Bất đẳng thức tam giác (triangle inequality): Với một cạnh $(u, v) \in E$, ta có $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
- Cận dưới (lower bound) và sự hội tụ (convergence): Các $d[v]$ sau một loạt phép co sẽ giảm dần nhưng không bao giờ nhỏ hơn khoảng cách $\delta(s, v)$. Tức là khi $d[v] = \delta(s, v)$ ($d[v]$ đạt cận dưới) thì không một phép co nào làm giảm $d[v]$ đi được nữa.
- Cây đường đi ngắn nhất (shortest-path tree): Nếu ta khởi tạo các $d[v]$ và thực hiện các phép co cho tới khi $d[v]$ bằng khoảng cách từ $s$ tới $v$ ($\forall v \in V$) thì chúng ta cũng xây dựng được một cây gốc $s$ trong đó nút $v$ là con của nút $Trace[v]$. Đường đi trên cây từ nút gốc $s$ tới một nút $v$ chính là đường đi ngắn nhất.

- Sự không tồn tại đường đi (no path): Nếu không tồn tại đường đi từ $s$ tới $t$ thì cho dù chúng ta co như thế nào chăng nữa, $d[t]$ luôn bằng $+\infty$.

Để tiện trong trình bày thuật toán, ta đưa vào một quy ước khi cộng giá trị $\infty$ với hằng số $C$. Bởi trong máy tính không có khái niệm $\infty$, các chương trình cài đặt thường sử dụng một hằng số đặc biệt để thay thế, nhưng có thể phải đi kèm với vài sửa đổi để hợp lý hóa các phép toán:

$$C + (+\infty) = (+\infty) + C = +\infty$$
$$C + (-\infty) = (-\infty) + C = -\infty$$

Dưới đây ta sẽ xét một số thuật toán tìm đường đi ngắn nhất từ đỉnh $s$ tới đỉnh $t$ trên đồ thị có hướng $G = (V, E)$ có $n$ đỉnh và $m$ cung, các đỉnh được đánh số từ 1 tới $n$. Trong trường hợp đồ thị vô hướng với trọng số không âm, bài toán tìm đường đi ngắn nhất có thể quy dẫn về bài toán tìm đường đi ngắn nhất trên phiên bản có hướng của đồ thị.

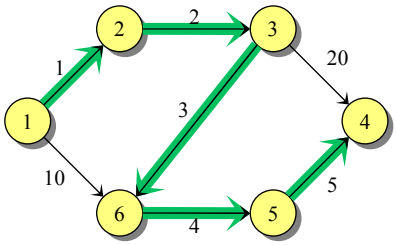
**Input**

- Dòng 1 chứa số đỉnh $n \leq 10^4$, số cung $m \leq 10^5$ của đồ thị, đỉnh xuất phát $s$, đỉnh đích $t$.
- $m$ dòng tiếp theo, mỗi dòng có dạng ba số $u, v, w$, cho biết $(u, v)$ là một cung $\in E$ và trọng số của cung đó là $w$ ($w$ là số nguyên có giá trị tuyệt đối $\leq 10^5$)

**Output**

Đường đi ngắn nhất từ $s$ tới $t$ và độ dài đường đi đó.

| Sample Input | Sample Output |
|---|---|
| 6 7 1 4<br>1 2 1<br>1 6 10<br>2 3 2<br>3 4 20<br>3 6 3<br>5 4 5<br>6 5 4 | Distance from 1 to 4: 15<br>4<-5<-6<-3<-2<-1 |




### e) Thuật toán Bellman-Ford

❑ ***Thuật toán***

Thuật toán Bellman-Ford[**5**][**14**] có thể sử dụng để tìm đường đi ngắn nhất xuất phát từ một đỉnh $s \in V$ **trong trường hợp đồ thị $G = (V, E, w)$ không có chu trình âm**. Thuật toán này khá đơn giản: Khởi tạo các nhãn khoảng cách $d[s] := 0$ và $d[v] := +\infty, \forall v \neq s$, sau đó thực hiện phép co theo mọi cạnh của đồ thị. Cứ lặp lại như vậy đến khi không thể cực tiểu hóa thêm bất kỳ một nhãn $d[v]$ nào nữa.

```
Init;
repeat
  Stop := True;
  for ∀e ∈E do
    if Relax(e) then Stop := False;
until Stop;
```

❑ ***Tính đúng và tính dừng***

Gọi $\delta_k(s, v)$ là độ dài ngắn nhất của một đường đi từ $s$ tới $v$ qua đúng $k$ cạnh, nếu không tồn tại đường đi từ $s$ tới $v$ qua $k$ cạnh thì $\delta_k(s, v) = +\infty$.

Ta chứng minh rằng sau mỗi lần lặp thứ $k$ của vòng lặp repeat…until thì

$$d[v] \leq \delta_k(s, v), \forall v \in V \qquad (1.2)$$

Tại bước khởi tạo, rõ ràng $d[v] = \delta_0(s, v)$. Giả sử bất đẳng thức (1.2) đúng trước lần lặp thứ $k$, ta chứng minh rằng bất đẳng thức vẫn đúng sau lần lặp thứ $k$. Thật vậy, đường đi ngắn nhất từ $s$ tới $v$ qua $k$ cạnh sẽ phải thành lập bằng

cách lấy một đường đi ngắn nhất từ $s$ tới một đỉnh $u$ nào đó qua $k-1$ cạnh rồi đi tiếp tới $v$ bằng cung $(u,v)$: $s \rightsquigarrow u \rightarrow v$. Vì thế $\delta^k(s,v)$ có thể được tính bằng công thức truy hồi:

$$\begin{aligned}\delta_k(s,v) &= \min_{\substack{u\in V\\(u,v)\in E}} \{\delta_{k-1}(s,u) + w(u,v)\}\\ &\geq \min_{\substack{u\in V\\(u,v)\in E}} \{d[u] + w(u,v)\} \qquad (1.3)\\ &\geq d[v]\end{aligned}$$

Bất đẳng thức thứ nhất đúng do giả thiết quy nạp và các phép co không bao giờ làm tăng $d[u]$. Bất đẳng thức thứ hai đúng vì sau khi căng theo tất cả các cạnh $(\dots, v)$ thì không thể tồn tại $u$ để $d[v] > d[u] + w(u,v)$ được nữa.

Trong các đường đi ngắn nhất từ $s$ tới $v$, sẽ có một đường đi đơn (qua không quá $n-1$ cạnh). Tức là $\delta(s,v) = \min_{k:1\leq k\leq n-1} \delta_k(s,v)$. Sau $n-1$ bước lặp của vòng lặp repeat…until, ta thu được các $d[v]$ thỏa mãn $d[v] \leq \delta_k(s,v)$ với mọi $v \in V$ và mọi số $k = 1,2,\dots,n-1$. Điều này chỉ ra rằng: $d[v] \leq \delta(s,v), \forall v \in V$. Mặt khác $d[v] \geq \delta(s,v)$ (tính bị chặn dưới), vậy $\forall v \in V: d[v] = \delta(s,v)$ sau $n-1$ bước lặp repeat…until, điều này cũng cho thấy thuật toán Bellman-Ford sẽ kết thúc sau không quá $n-1$ bước lặp repeat…until.

❑ ***Cài đặt***

Cách biểu diễn đồ thị tốt nhất để cài đặt thuật toán Bellman-Ford là sử dụng danh sách cạnh. Danh sách cạnh của đồ thị được lưu trữ trong mảng $e[1\dots m]$, mỗi phần tử của mảng là một bản ghi chứa chỉ số hai đỉnh đầu mút $u, v$ và trọng số $w$ tương ứng với một cạnh

💻 BELLMANFORD.PAS ✓ Thuật toán Bellman-Ford

```
{$MODE OBJFPC}
program BellmanFordShortestPath;
const
  maxN = 10000;
  maxM = 100000;
  maxW = 100000;
  maxD = maxN * maxW;
type
```

```
  TEdge = record //Cấu trúc biểu diễn cung
    x, y: Integer; //Đỉnh đầu và đỉnh cuối
    w: Integer; //Trọng số
  end;
var
  e: array[1..maxM] of TEdge; //Danh sách cung
  d: array[1..maxN] of Integer; //Nhãn trọng số
  trace: array[1..maxN] of Integer; //Vết
  n, m, s, t: Integer;
procedure Enter; //Nhập dữ liệu
var i: Integer;
begin
  ReadLn(n, m, s, t);
  for i := 1 to m do
    with e[i] do ReadLn(x, y, w);
end;
procedure Init; //Khởi tạo
var v: Integer;
begin
  for v := 1 to n do d[v] := MaxD; //Các nhãn d[v] := +∞
  d[s] := 0; //Ngoại trừ d[s] = 0
end;
function Relax(const e: TEdge): Boolean; //Phép co theo cạnh e
begin
  with e do
    begin
      Result := (d[x] < maxD) and (d[y] > d[x] + w);
      if Result then  //Co được
        begin
          d[y] := d[x] + w; //Cực tiểu hóa nhãn d[v]
          trace[y] := x; //Lưu vết
        end;
    end;
end;
procedure BellmanFord;
var
  Stop: Boolean;
  i, CountLoop: Integer;
```

```
begin
  for CountLoop := 1 to n - 1 do //Lặp tối đa n - 1 lần
    begin
      Stop := True; //Chưa có sự thay đổi nhãn nào
      for i := 1 to m do
        if Relax(e[i]) then Stop := False;
      if Stop then Break; //Không nhãn nào thay đổi, dừng
    end;
end;
procedure PrintResult; //In kết quả
begin
  if d[t] = maxD then //d[t] = +∞, không có đường
    WriteLn('There is no path from ', s, ' to ', t)
  else
    begin
      WriteLn('Distance from ', s, ' to ', t, ':
                                              ', d[t]);
      while t <> s do //Truy vết từ t
        begin
          Write(t, '<-');
          t := trace[t];
        end;
      WriteLn(s);
    end;
end;
begin
  Enter;
  Init;
  BellmanFord;
  PrintResult;
end.
```

Dễ thấy rằng thời gian thực hiện giải thuật Bellman-Ford trên đồ thị $G(V, E, w)$ là $O(|V||E|)$.

### f) Thuật toán Dijkstra

❑ ***Thuật toán***

Trong **trường hợp đồ thị $G = (V, E, w)$ có trọng số trên các cung không âm**, thuật toán do Dijkstra [**8**] đề xuất dưới đây hoạt động hiệu quả hơn nhiều so với thuật toán Bellman-Ford bởi quá trình sửa nhãn sẽ *chỉ xét mỗi cạnh tối đa một lần.* Tại mỗi bước, thuật toán đi tìm đỉnh $u$ mà nhãn $d[u]$ đã đạt cận dưới bằng $\delta(s, u)$. Nhãn $d[u]$ chắc chắn không thể cực tiểu hóa được nữa, khi đó thuật toán mới tiến hành cực tiểu hóa các nhãn $d[v]$ khác bằng các phép sửa nhãn theo cạnh $(u, v)$. Các bước cụ thể được tiến hành như sau:

**Bước 1: Khởi tạo**

Gọi thủ tục Init để khởi tạo các nhãn khoảng cách $d[s] := 0$ và $d[v] := +\infty, \forall v \neq s$. Một nhãn $d[v]$ gọi là **cố định** nếu ta biết chắc $d[v] = \delta(s, v)$ và không thể cực tiểu hóa $d[v]$ thêm nữa bằng phép co, ngược lại nhãn $d[v]$ gọi là **tự do**. Ta sẽ đánh dấu trạng thái nhãn bằng mảng $avail[1..n]$ trong đó $avail[v] =$ True nếu nhãn $d[v]$ còn tự do. Ban đầu tất cả các nhãn đều tự do.

**Bước 2: Lặp, bước lặp gồm có hai thao tác:**

- Cố định nhãn: Chọn trong các đỉnh có nhãn tự do, lấy ra đỉnh $u$ là đỉnh có $d[u]$ nhỏ nhất, đánh dấu cố định nhãn đỉnh $u$ ($avail[u] :=$ False).
- Sửa nhãn: Dùng đỉnh $u$, xét tất cả những đỉnh $v$ nối từ $u$ và thực hiện phép co theo cung $(u, v)$ để cực tiểu hóa nhãn $d[v]$.

Bước lặp sẽ kết thúc khi mà đỉnh đích $t$ được cố định nhãn (tìm được đường đi ngắn nhất từ $s$ tới $t$); hoặc tại thao tác cố định nhãn, tất cả các đỉnh tự do đều có nhãn là $+\infty$ (không tồn tại đường đi).

Tại lần lặp đầu tiên, đỉnh $s$ có $d[s]$ nhỏ nhất (bằng 0) sẽ được cố định nhãn. Có thể đặt câu hỏi tại sao đỉnh $u$ có nhãn tự do nhỏ nhất được cố định nhãn tại từng bước, giả sử $d[u]$ còn có thể làm nhỏ hơn nữa thì tất phải có một đỉnh $x$ mang nhãn tự do sao cho $d[u] > d[x] + w(x, u)$. Do trọng số $w(x, u)$ không âm nên $d[u] > d[x]$, trái với cách chọn $d[u]$ nhỏ nhất.

**Bước 3: Truy vết**

Kết hợp với việc lưu vết đường đi trên từng bước sửa nhãn, thông báo đường đi ngắn nhất tìm được hoặc cho biết không tồn tại đường đi.

❑ ***Cài đặt***

Thuật toán Dijkstra hoạt động tốt nhất nếu đồ thị được biểu diễn bằng danh sách kề dạng forward star. Cấu trúc danh sách kề được khai báo như sau:

```
type
  TAdjNode = record //Cấu trúc nút của danh sách kề
    v: Integer;   //Đỉnh kề
    w: Integer; //Trọng số cạnh tương ứng
  end;
var
  adj: array[1..maxM] of TAdjNode; //Mảng các nút
  head: array[1..maxN] of Integer;
  //head[u]: Chỉ số nút đầu tiên của danh sách kề u
  link: array[1..maxM] of Integer;
  //link[i]: Chỉ số nút kế tiếp nút adj[i] trong cùng một danh sách kề
```

Mỗi đỉnh $u$ sẽ tương ứng với một danh sách các nút, mỗi nút chứa một đỉnh $v$ và trọng số $w$ của một cung $(u, v)$. Tất cả các nút được lưu trữ trong mảng $adj[1 \dots m]$ và mỗi nút sẽ thuộc đúng một danh sách kề. Các nút thuộc danh sách kề của đỉnh $u$ là $adj[i_1], adj[i_2], adj[i_3]$, trong đó $i_1 = head[u]; i_2 = link[i_1]; i_3 = link[i_2]$ ... Đây chính là cấu trúc dữ liệu biểu diễn danh sách móc nối đơn, nhưng khác với những phép cài đặt truyền thống, ta sử dụng mảng các nút thay cho cơ chế cấp phát biến động và sử dụng chỉ số với vai trò như con trỏ.

💻 DIJKSTRA.PAS ✓

```
{$MODE OBJFPC}
program DijkstraShortestPath;
const
  maxN = 10000;
  maxM = 100000;
  maxW = 100000;
  maxD = maxN * maxW;
type
  TAdjNode = record //Cấu trúc nút của danh sách kề
    v: Integer; //Đỉnh kề
    w: Integer; //Trọng số cung tương ứng
```

```
    link: Integer; //Chỉ số nút kế tiếp trong cùng danh sách kề
  end;
var
  adj: array[1..maxM] of TAdjNode; //Mảng chứa tất cả các nút
  head: array[1..maxN] of Integer;
  //head[u]: Chỉ số nút đứng đầu danh sách kề của u
  d: array[0..maxN] of Integer; //Nhãn khoảng cách
  avail: array[1..maxN] of Boolean; //Đánh dấu tự do/cố định
  trace: array[1..maxN] of Integer; //Vết đường đi
  n, m, s, t: Integer;
procedure Enter; //Nhập dữ liệu
var i, u: Integer;
begin
  ReadLn(n, m, s, t);
  FillChar(head[1], n * SizeOf(head[1]), 0);
  //Khởi tạo các danh sách kề rỗng
  for i := 1 to m do
    begin
      ReadLn(u, adj[i].v, adj[i].w);
      //Đọc một cung (u, v) trọng số w, đưa v và w vào trong nút adj[i]
      adj[i].link := head[u]; //Chèn nút adj[i] vào đầu danh sách kề của u
      head[u] := i; //Cập nhật chỉ số nút đứng đầu danh sách kề của u
    end;
end;
procedure Init; //Khởi tạo
var v: Integer;
begin
  for v := 0 to n do d[v] := MaxD;
  //Các nhãn d[v] := +∞, d[0]: phần tử cầm canh
  d[s] := 0; //Ngoại trừ d[s] = 0
  FillChar(avail[1], n * SizeOf(avail[1]), True);
  //Các nhãn đều tự do
end;
procedure Relax(u, v: Integer; w: Integer);
//Phép co theo cung (u, v) trọng số w
begin
  if d[v] > d[u] + w then
    begin
      d[v] := d[u] + w;
```

```
      trace[v] := u;
    end;
end;
procedure Dijkstra; //Thuật toán Dijkstra
var u, v, i: Integer;
begin
  repeat
    //Tìm đỉnh u có nhãn tự do nhỏ nhất
    u := 0;
    for v := 1 to n do
      if avail[v] and (d[v] < d[u]) then
        u := v;
    if (u = 0) or (u = t) then Break;
    //u = 0: không tồn tại đường đi, u = t, xong
    avail[u] := False; //Cố định nhãn đỉnh u
    //Co theo các cung nối từ u
    i := head[u]; //Duyệt từ đầu danh sách kề
    while i <> 0 do
      begin
        Relax(u, adj[i].v, adj[i].w); //Thực hiện phép co
        i := adj[i].link; //Chuyển sang nút kế tiếp trong danh sách kề
      end;
  until False;
end;
procedure PrintResult; //In kết quả
begin
  if d[t] = maxD then
    WriteLn('There is no path from ', s, ' to ', t)
  else
    begin
      WriteLn('Distance from ', s, ' to ', t, ':
                                              ', d[t]);
      while t <> s do
        begin
          Write(t, '<-');
          t := trace[t];
        end;
      WriteLn(s);
```

```
    end;
end;
begin
  Enter;
  Init;
  Dijkstra;
  PrintResult;
end.
```

Mỗi lượt của vòng lặp repeat…until sẽ có một đỉnh mang nhãn tự do bị cố định nhãn, suy ra số lượt lặp của vòng lặp repeat…until là $O(|V|)$. Việc tìm đỉnh $u$ có nhãn tự do nhỏ nhất được thực hiện trong thời gian $O(|V|)$, vậy nên xét trên toàn thuật toán, tổng thời gian thực hiện của các pha cố định nhãn là $O(|V|^2)$.

Mỗi lượt của vòng lặp repeat…until khi cố định nhãn đỉnh $u$ sẽ phải duyệt danh sách các đỉnh nối từ $u$ để thực hiện pha sửa nhãn. Vì vậy xét trên toàn thuật toán, tổng thời gian thực hiện của các pha sửa nhãn là $O(\sum_{u \in V} \deg^+(u)) = O(|E|)$.

Vậy thủ tục Dijkstra thực hiện trong thời gian $O(|V|^2 + |E|)$.

❑ ***Kết hợp với hàng đợi ưu tiên***

Để thuật toán Dijkstra làm việc hiệu quả hơn, người ta thường kết hợp với một cấu trúc dữ liệu hàng đợi ưu tiên chứa các đỉnh tự do có nhãn $\neq +\infty$ để thuận tiện trong việc lấy ra đỉnh có nhãn nhỏ nhất cũng như cập nhật lại nhãn của các đỉnh. Hàng đợi ưu tiên cần hỗ trợ các thao tác sau:

- $Extract$: Lấy ra một đỉnh ưu tiên nhất (đỉnh $u$ có $d[u]$ nhỏ nhất) khỏi hàng đợi ưu tiên.
- $Update(v)$: Thao tác này báo cho hàng đợi ưu tiên biết rằng nhãn $d[v]$ đã bị giảm đi, cần tổ chức lại (thêm $v$ vào hàng đợi ưu tiên nếu $v$ đang nằm ngoài).

Khi đó thuật toán Dijkstra có thể viết theo mô hình mới sử dụng hàng đợi ưu tiên:

```
Init;
PQ := (s); //Hàng đợi ưu tiên được khởi tạo chỉ gồm đỉnh xuất phát
repeat
  u := Extract; //Lấy ra đỉnh u có d[u] nhỏ nhất
```

```
  if u = t then Break;
  for ∀(u, v) ∈ E do
    if Relax(u, v) then Update(v);
until PQ = ∅;
if d[t] = +∞ then
  Output ← Không có đường
else
  «Truy vết tìm đường đi từ s tới t»
```

Vòng lặp repeat…until mỗi lần sẽ lấy một đỉnh khỏi hàng đợi ưu tiên và đỉnh lấy ra sẽ không bao giờ bị đẩy vào hàng đợi ưu tiên lại nữa. Vòng lặp for bên trong xét trong tổng thể cả chương trình sẽ duyệt qua tất cả các cung $(u, v)$ của đồ thị. Vậy thuật toán Dijkstra cần thực hiện không quá $n$ phép $Extract$ và $m$ phép $Update$ với $n$ là số đỉnh và $m$ là số cung của đồ thị.

Trong chương trình cài đặt thuật toán Dijkstra dưới đây tôi sử dụng Binary Heap để biểu diễn hàng đợi ưu tiên, khi đó thời gian thực hiện giải thuật sẽ là $O(n \lg n + m \lg n)$. Ngoài cấu trúc Binary Heap, người ta cũng đã nghiên cứu nhiều cấu trúc dữ liệu hiệu quả hơn để biểu diễn hàng đợi ưu tiên, chẳng hạn Fibonacci Heap[**17**], Relaxed Heap[**11**], 2-3 Heap[**39**], v.v… Những cấu trúc dữ liệu này cho phép cài đặt thuật toán Dijkstra chạy trong thời gian $O(n \lg n + m)$. Tuy nhiên việc cài đặt các cấu trúc dữ liệu này khá phức tạp, các bạn có thể tham khảo trong những tài liệu khác.

💻 DIJKSTRAHEAP.PAS ✓ Thuật toán Dijkstra và cấu trúc Heap

```
{$MODE OBJFPC}
program DijkstraShortestPathUsingHeap;
const
  maxN = 10000;
  maxM = 100000;
  maxW = 100000;
  maxD = maxN * maxW;
type
  TAdjNode = record //Cấu trúc nút của danh sách kề
    v: Integer; //Đỉnh kề
    w: Integer; //Trọng số cung tương ứng
    link: Integer; //Chỉ số nút kế tiếp trong cùng danh sách kề
```

```
  end;
  THeap = record //Cấu trúc Heap
    Items: array[1..maxN] of Integer; //Các phần tử chứa trong
    nItems: Integer; //Số phần tử chứa trong
    Pos: array[1..maxN] of Integer;
    //Pos[v] = vị trí của đỉnh v trong Heap
  end;
var
  adj: array[1..maxM] of TAdjNode; //Mảng chứa tất cả các nút
  head: array[1..maxN] of Integer;
  //head[u]: Chỉ số nút đứng đầu danh sách kề của u
  d: array[1..maxN] of Integer;
  trace: array[1..maxN] of Integer;
  n, m, s, t: Integer;
  Heap: THeap;
procedure Enter; //Nhập dữ liệu
var i, u: Integer;
begin
  ReadLn(n, m, s, t);
  FillChar(head[1], n * SizeOf(head[1]), 0);
  //Khởi tạo các danh sách kề rỗng
  for i := 1 to m do
    begin
      ReadLn(u, adj[i].v, adj[i].w);
      //Đọc một cung (u, v) trọng số w, đưa v và w vào trong nút adj[i]
      adj[i].link := head[u]; //Chèn nút adj[i] vào đầu danh sách kề của u
      head[u] := i; //Cập nhật chỉ số nút đứng đầu danh sách kề của u
    end;
end;
procedure Init;
var v: Integer;
begin
  for v := 1 to n do d[v] := MaxD;
  d[s] := 0;
  with Heap do //Khởi tạo Heap chỉ chứa mỗi phần tử s
    begin
      FillChar(Pos[1], n * SizeOf(Pos[1]), 0);
      Items[1] := s;
      Pos[s] := 1;
```

```
      nItems := 1;
    end;
end;
function Extract: Integer; //Lấy đỉnh u có nhãn d[u] nhỏ nhất ra khỏi Heap
var p, c, v: Integer;
begin
  with Heap do
    begin
      Result := Items[1]; //Trả về đỉnh ở gốc Heap
      v := Items[nItems]; //Vun lại Heap bằng phép Down-Heap
      Dec(nItems);
      p := 1; //Bắt đầu từ gốc
      repeat
      //Tìm c là nút con chứa đỉnh mang nhãn khoảng cách nhỏ hơn trong hai nút con
        c := p * 2;
        if (c < nItems)
            and (d[Items[c + 1]] < d[Items[c]]) then
          Inc(c);
        if (c > nItems)
            or (d[v] <= d[Items[c]]) then Break;
        Items[p] := Items[c];  //Chuyển đỉnh từ c lên p
        Pos[Items[p]] := p; //Cập nhật vị trí
        p := c; //Đi xuống nút con
      until False;
      Items[p] := v; //Đặt đỉnh v vào nút p của Heap
      Pos[v] := p; //Cập nhật vị trí
    end;
end;
procedure Update(v: Integer); //d[v] vừa bị cực tiểu hóa, tổ chức lại Heap
var p, c: Integer;
begin
  with Heap do
    begin
      c := Pos[v]; //c là vị trí của đỉnh v trong Heap
      if c = 0 then //Nếu v chưa có trong Heap
        begin
          Inc(nItems);
          c := nItems; //Cho v vào Heap ở vị trí một nút lá
```

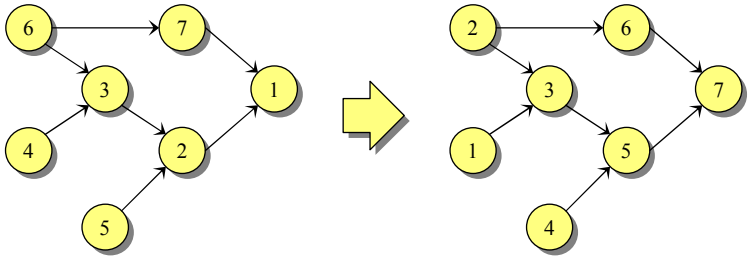
```
        end;
      repeat //Thực hiện Up-Heap
        p := c div 2; //Xét nút cha của c
        if (p = 0) or (d[Items[p]] <= d[v]) then Break;
        //Dừng nếu đã xét lên gốc hoặc gặp vị trí đúng
        Items[c] := Items[p]; //Kéo đỉnh từ nút cha xuống nút con
        Pos[Items[c]] := c; //Cập nhật vị trí
        c := p; //Đi lên nút cha
      until False;
      Items[c] := v; //Đặt v vào nút c
      Pos[v] := c; //Cập nhật vị trí
    end;
end;
function Relax(u, v: Integer; w: Integer): Boolean;
//Phép co theo cạnh (u, v) trọng số w
begin
  Result := d[v] > d[u] + w;
  if Result then
    begin
      d[v] := d[u] + w;
      trace[v] := u;
    end;
end;
procedure Dijkstra; //Thuật toán Dijkstra
var u, i: Integer;
begin
  repeat
    u := Extract; //Lấy ra đỉnh u có d[u] nhỏ nhất
    if (u = 0) or (u = t) then Break;
    i := head[u]; //Duyệt từ đầu danh sách kề của u
    while i <> 0 do
      begin
        if Relax(u, adj[i].v, adj[i].w) then
        //Nếu thực hiện được phép co
          Update(adj[i].v); //Tổ chức lại Heap
        i := adj[i].link; //Chuyển sang nút kế tiếp trong danh sách kề
      end;
  until Heap.nItems = 0;
end;
```

```
procedure PrintResult; //In kết quả
begin
  if d[t] = maxD then
    WriteLn('There is no path from ', s, ' to ', t)
  else
    begin
      WriteLn('Distance from ', s, ' to ', t, ':
                                              ', d[t]);
      while t <> s do
        begin
          Write(t, '<-');
          t := trace[t];
        end;
      WriteLn(s);
    end;
end;
begin
  Enter;
  Init;
  Dijkstra;
  PrintResult;
end.
```

### g) Đường đi ngắn nhất trên đồ thị không có chu trình

❑ ***Thuật toán***

Xét **trường hợp đồ thị có hướng, không có chu trình** (Directed Acyclic Graph - DAG), có một thuật toán hiệu quả để tìm đường đi ngắn nhất dựa trên kỹ thuật sắp xếp Tô pô (Topological Sorting), cơ sở của thuật toán dựa vào định lý: Nếu $G = (V, E)$ là một DAG thì các đỉnh của nó có thể đánh số sao cho mỗi cung của $G$ chỉ nối từ đỉnh có chỉ số nhỏ hơn đến đỉnh có chỉ số lớn hơn.




*Hình 2.2. Phép đánh chỉ số lại theo thứ tự tô pô*

Phép đánh số theo thứ tự tô pô đã được trình bày trong thuật toán Kosaraju-Sharir (**Error! Reference source not found.**, Mục **Error! Reference source not found.**): Dùng thuật toán tìm kiếm theo chiều sâu trên đồ thị đảo chiều và đánh số các đỉnh theo thứ tự duyệt xong (Finish).

```
procedure TopoSort;
  procedure DFSVisit(v∈V);
  //Thuật toán tìm kiếm theo chiều sâu từ đỉnh v trên đồ thị đảo chiều
  begin
    avail[v] := False; //avail[v] = False ⇔ v đã thăm
    for ∀u ∈V:(u, v)∈E do //Duyệt mọi đỉnh v chưa thăm nối đến u
      if avail[u] then
        DFSVisit(u); //Gọi đệ quy để tìm kiếm theo chiều sâu từ đỉnh u
    «Đánh số v»; //Đánh số v theo thứ tự duyệt xong
  end;
begin
  for ∀v ∈V do avail[v] := True; //Đánh dấu mọi đỉnh đều chưa thăm
  for ∀v ∈V do
    if avail[v] then DFSVisit(v);
end.
```

Một cách khác để đánh số theo thứ tự tô pô là sử dụng thuật toán tìm kiếm theo chiều rộng: Với mỗi đỉnh $v$ ta tính và lưu trữ $\deg^-[v]$ là bán bậc vào của nó. Sử dụng một hàng đợi chứa các đỉnh có bán bậc vào bằng 0 (đỉnh không có cung đi vào). Sau đó cứ lấy một đỉnh $u$ khỏi hàng đợi, đánh số cho đỉnh $u$ đó và xóa đỉnh $u$ khỏi đồ thị. Việc xóa đỉnh $u$ khỏi đồ thị tương đương với việc giảm tất cả các $\deg^-[v]$ của những đỉnh $v$ nối từ $u$ đi 1, nếu $\deg^-[v]$ bị giảm về 0 thì đẩy $v$ vào hàng đợi để chờ…Quá trình đánh số sẽ kết thúc khi hàng đợi rỗng (tất cả

các đỉnh đều đã được đánh số thứ tự mới). Phương pháp này tuy cài đặt có dài dòng hơn và chậm hơn một chút nhưng không cần sử dụng đệ quy.

Nếu các đỉnh được đánh số sao cho mỗi cung phải nối từ một đỉnh tới một đỉnh khác mang chỉ số lớn hơn thì thuật toán tìm đường đi ngắn nhất có thể thực hiện khá đơn giản:

```
Init;
for v := s + 1 to t do
  for ∀u:(u, v)∈E do Relax(u, v); //Co theo các cung nối tới v
```

Có thể thấy rằng sau mỗi bước lặp với đỉnh $v$, nhãn $d[v]$ không thể co thêm được nữa ($d[u] = \delta(s,u)$).

❑ ***Cài đặt***

Vì mục tiêu của chúng ta chỉ cần tìm đường đi từ $s$ tới $t$, vì thế chúng ta chỉ cần đánh số thứ tự tô pô cho những đỉnh đến được $t$ bằng lời gọi $DFSVisit(t)$, những đỉnh đến được từ $t$ (có thứ tự tô pô lớn hơn $t$) sẽ bị bỏ qua.

Cách cài đặt thông thường nhất để tìm đường đi ngắn nhất trên đồ thị không có chu trình là tách biệt hai pha: Sắp xếp tô pô và tối ưu nhãn. Pha sắp xếp tô pô trước hết thực hiện tìm kiếm theo chiều sâu trên đồ thị đảo chiều bằng lời gọi $DFSVisit(t)$. Mỗi khi một đỉnh được duyệt xong (đánh số thứ tự tô pô), nó sẽ được đẩy vào một hàng đợi. Pha tối ưu nhãn lần lượt lấy các đỉnh ra khỏi hàng đợi (theo đúng thứ tự tô pô) và thực hiện phép co theo tất cả các cung nối tới đỉnh vừa lấy ra. Cách cài đặt này có ưu điểm là khá sáng sủa, trong trường hợp mà ta bỏ qua được khâu sắp xếp tô pô hoặc có thể xác định thứ tự tô pô bằng một cách đơn giản hơn, chương trình trở nên rất gọn.

Tuy nhiên nếu ta phải giải quyết bài toán tổng quát bao gồm cả hai pha sắp xếp tô pô và tối ưu nhãn thì có thể khéo léo lồng pha tối ưu nhãn vào pha sắp xếp tô pô: Trước khi đỉnh $v$ được đánh số (duyệt xong), ta thực hiện tất cả các phép co theo các cung $(u,v)$ với mọi đỉnh $u$ nối đến $v$.

💻 SHORTESTPATHDAG.PAS ✓ Đường đi ngắn nhất trên DAG

```
{$MODE OBJFPC}
program DAGShortestPath;
const
  maxN = 10000;
```

```
  maxM = 100000;
  maxW = 100000;
  maxD = maxN * maxW;
type
  TAdjNode = record //Cấu trúc nút của danh sách kề dạng reverse star
    u: Integer; //Đỉnh kề
    w: Integer; //Trọng số cung tương ứng
    link: Integer; //Chỉ số nút kế tiếp trong cùng danh sách kề
  end;
var
  adj: array[1..maxM] of TAdjNode; //Mảng chứa tất cả các nút
  head: array[1..maxN] of Integer;
  //head[u]: Chỉ số nút đứng đầu danh sách kề của u
  d: array[1..maxN] of Integer; //Nhãn khoảng cách
  trace: array[1..maxN] of Integer; //Vết
  avail: array[1..maxN] of Boolean;
  n, m, s, t: Integer;
procedure Enter; //Nhập dữ liệu
var i, v: Integer;
begin
  ReadLn(n, m, s, t);
  FillChar(head[1], n * SizeOf(head[1]), 0);
  for i := 1 to m do
    begin
      ReadLn(adj[i].u, v, adj[i].w);
      //Đọc một cung (u, v) trọng số w, đưa u và w vào trong nút adj[i]
      adj[i].link := head[v]; //Chèn nút adj[i] vào đầu danh sách kề của v
      head[v] := i; //Cập nhật chỉ số nút đứng đầu danh sách kề của v
    end;
end;
procedure Init;
var v: Integer;
begin
  FillChar(avail[1], n * SizeOf(avail[1]), True);
  for v := 1 to n do d[v] := maxD;
  d[s] := 0;
end;
procedure Relax(u, v: Integer; w: Integer);
```

```
begin
  if (d[u] < maxD) and (d[v] > d[u] + w) then
    begin
      d[v] := d[u] + w;
      trace[v] := u;
    end;
end;
procedure DFSVisit(v: Integer); //DFS trên đồ thị đảo chiều
var i: Integer;
begin
  avail[v] := False;
  i := head[v];
  while i <> 0 do //Duyệt danh sách các đỉnh nối đến v
    begin
      if avail[adj[i].u] then
      //adj[i].u là một đỉnh nối đến v, nếu adj[i].u chưa thăm thì đi thăm,
        DFSVisit(adj[i].u); //sau lời gọi này d[adj[i].u] sẽ bằng δ(s, adj[i].u)
      Relax(adj[i].u, v, adj[i].w);
      //Thực hiện luôn phép co (adj[i].u, v)
      i := adj[i].link; //Nhảy sang nút kế tiếp trong danh sách kề
    end;
end; //Khi thủ tục kết thúc, d[v] sẽ bằng δ(s,v)
procedure PrintResult;  //In kết quả
begin
  if d[t] = maxD then
    WriteLn('There is no path from ', s, ' to ', t)
  else
    begin
      WriteLn('Distance from ', s, ' to ', t, ':
                                              ', d[t]);
      while t <> s do
        begin
          Write(t, '<-');
          t := trace[t];
        end;
      WriteLn(s);
    end;
end;
```

```
begin
  Enter;
  Init;
  DFSVisit(t);
  PrintResult;
end.
```

Thời gian thực hiện giải có thể đánh giá qua thời gian thực hiện giải thuật DFS, tức là bằng $O(|E|)$ khi đồ thị được biểu diễn bởi danh sách kề.

## *1.3. Đường đi ngắn nhất giữa mọi cặp đỉnh*

Trong một số ứng dụng thực tế, đôi khi người ta ta có nhu cầu tính sẵn *đường đi ngắn nhất giữa mọi cặp đỉnh* của đồ thị (*all-pairs shortest paths*) để trả lời nhanh những truy vấn tìm đường đi ngắn nhất mà không cần thực hiện lại thuật toán. Rõ ràng ta có thể áp dụng thuật toán tìm đường đi ngắn nhất xuất phát từ một đỉnh với $n$ lượt chọn đỉnh xuất phát, nhưng những thuật toán trong mục này có thể thực hiện nhanh hơn và đơn giản hơn nhiều.

### a) Thuật toán Floyd

Cho đơn đồ thị có hướng, có trọng số $G = (V, E)$ với $n$ đỉnh và $m$ cung. Thuật toán Floyd tính tất cả các phần tử của ma trận khoảng cách $D = \{d_{uv}\}_{n\times n}$, trong đó $d[u, v]$ là khoảng cách từ $u$ tới $v$. Cách làm tương tự như thuật toán Warshall để tìm bao đóng đồ thị: từ ma trận trọng số $W = \{w[u, v]\}_{n\times n}$, trong đó $w[v, v] = 0, \forall v \in V$, thuật toán Floyd tính lại các $w[u, v]$ thành độ dài đường đi ngắn nhất từ $u$ tới $v$ theo cách sau: Với $\forall k \in V$ được xét theo thứ tự từ 1 tới $n$, thuật toán xét mọi cặp đỉnh $u, v$ và cực tiểu hóa $w[u, v]$ theo công thức:

$$w[u, v]_{\text{mi}} \coloneqq \min\{w[u, v]_{\text{cũ}}, w[u, k] + w[k, v]\} \qquad (1.4)$$

Tức là nếu như đường đi từ $u$ tới $v$ đang có lại dài hơn đường đi từ $u$ tới $k$ cộng với đường đi từ $k$ tới $v$ thì ta huỷ bỏ đường đi từ $u$ tới $v$ hiện thời và coi đường đi từ $u$ tới $v$ sẽ là nối của hai đường đi từ $u$ tới $k$ rồi từ $k$ tới $v$:

```
for k := 1 to n do
  for u := 1 to n do
    for v := 1 to n do
      w[u, v] := min(w[u, v], w[u, k] + w[k, v]);
```

❑ ***Tính đúng của thuật toán***

Gọi $\delta_k(u,v)$ là độ dài đường đi ngắn nhất từ $u$ tới $v$ mà chỉ đi qua các đỉnh trung gian thuộc tập $\{1,2,\dots,k\}$. Rõ ràng khi $k=0$ thì $\delta_0(u,v)=w[u,v]$ (đường đi ngắn nhất là đường đi trực tiếp không qua đỉnh trung gian nào).

Nếu đường đi ngắn nhất từ $u$ tới $v$ mà chỉ qua các đỉnh trung gian thuộc tập $\{1,2,\dots,k\}$ lại:

- Không đi qua đỉnh $k$, tức là chỉ qua các đỉnh trung gian thuộc tập $\{1,2,\dots,k-1\}$ thì $\delta_k(u,v)=\delta_{k-1}(u,v)$.
- Có đi qua đỉnh $k$, thì đường đi đó sẽ là nối của một đường đi ngắn nhất từ $u$ tới $k$ và một đường đi ngắn nhất từ $k$ tới $v$, hai đường đi này chỉ đi qua các đỉnh trung gian thuộc tập $\{1,2,\dots,k-1\}$, vậy $\delta_k(u,v)=\delta_{k-1}(u,k)+\delta_{k-1}(k,v)$.

Vì ta muốn $\delta_k(u,v)$ nhỏ nhất nên suy ra:

$$\delta_k(u,v)=\min\{\delta_{k-1}(u,v),\delta_{k-1}(u,k)+\delta_{k-1}(k,v)\} \qquad (1.5)$$

Cuối cùng ta quan tâm tới các $\delta_n(u,v)$: Độ dài đường đi ngắn nhất từ $u$ tới $v$ mà chỉ đi qua các đỉnh trung gian thuộc tập $\{1,2,\dots,n\}$, tức là khoảng cách giữa $u$ và $v$: $\delta(u,v)$.

Ta sẽ chứng minh rằng sau mỗi bước lặp của vòng lặp "for k…", thì:

$$w[u,v]\le\delta_k(u,v) \qquad (1.6)$$

Phép chứng minh được thực hiện quy nạp theo $k$. Ký hiệu $w_k[u,v]$ là giá trị $w[u,v]$ sau vòng lặp thứ $k$, khi $k=0$ thì như đã chỉ ra ở trên, $w_0[u,v]=\delta_0(u,v)$. Giả sử bất đẳng thức đúng với $k-1$, trước hết dễ thấy rằng các $w[u,v]$ sẽ được tối ưu hóa giảm dần theo từng bước. Từ công thức (1.5), ta có:

$$\delta_k(u,v)=\min\left\{\underbrace{\delta_{k-1}(u,v)}_{\ge w_{k-1}(u,v)},\underbrace{\delta_{k-1}(u,k)}_{\ge w_{k-1}(u,k)}+\underbrace{\delta_{k-1}(k,v)}_{\ge w_{k-1}(k,v)}\right\}$$
$$\ge w_k[u,v]$$

Mặt khác có thể thấy thuật toán Floyd tìm được $w_n[u,v]$ là độ dài của một đường đi từ $u$ tới $v$. Tức là $w_n[u,v]\ge\delta_n(u,v)$. Từ những kết quả trên suy ra

khi kết thúc thuật toán, $w_n(u,v) = \delta_n(u,v) = \delta(u,v)$ là độ dài đường đi ngắn nhất từ $u$ tới $v$.

❑ ***Cài đặt***

Ta sẽ cài đặt thuật toán Floyd trên đồ thị có hướng gồm $n$ đỉnh, $m$ cung với khuôn dạng Input/Output như sau:

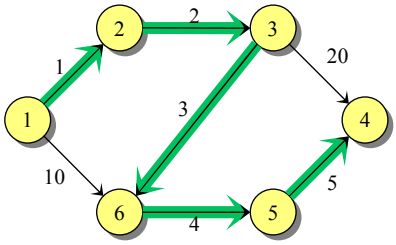
**Input**

- Dòng 1 chứa số đỉnh $n \leq 10^3$, số cung $m \leq 10^6$ của đồ thị, đỉnh xuất phát $s$, đỉnh cần đến $t$.
- $m$ dòng tiếp theo, mỗi dòng có dạng ba số $u, v, w$, cho biết $(u,v)$ là một cung $\in E$ và trọng số của cung đó là $w$ ($w$ là số nguyên có giá trị tuyệt đối $\leq 10^5$)

**Output**

Đường đi ngắn nhất từ $s$ tới $t$ và độ dài đường đi đó.

| Sample Input | Sample Output |
|---|---|
| 6 7 1 4<br>1 2 1<br>1 6 10<br>2 3 2<br>3 4 20<br>3 6 3<br>5 4 5<br>6 5 4 | Distance from 1 to 4: 15<br>1->2->3->6->5->4 |




Thuật toán Floyd chỉ cần thực hiện một lần trên đồ thị và khi cần tìm đường đi ngắn nhất giữa một cặp đỉnh khác, ta chỉ cần dò đường dựa trên ma trận khoảng cách mà thôi. Trên thực tế người ta thường kết hợp với một cơ chế lưu vết để trả lời nhanh nhiều truy vấn về đường đi ngắn nhất: Gọi $trace[u,v]$ là đỉnh đứng liền sau $u$ trên đường đi ngắn nhất từ $u$ tới $v$. Sau mỗi phép cực tiểu hóa $c[u,v] := c[u,k] + c[k,v]$ (đường đi ngắn nhất từ $u$ tới $v$ phải đi vòng qua $k$), ta cập nhật lại vết $trace[u,v] := trace[u,k]$.

## FLOYD.PAS ✓ Thuật toán Floyd

```
{$MODE OBJFPC}
program FloydAllPairsShortestPaths;
const
  maxN = 1000;
  maxW = 1000;
  maxD = maxN * maxW;
var
  w: array[1..maxN, 1..maxN] of Integer; //Ma trận trọng số
  trace: array[1..maxN, 1..maxN] of Integer; //Vết
  n, m, s, t: Integer;
procedure Enter; //Nhập dữ liệu, các cạnh không có được gán trọng số +∞
var
  i, u, v, weight: Integer;
begin
  ReadLn(n, m, s, t);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then w[u, v] := 0
      else w[u, v] := maxD;
  for i := 1 to m do
    begin
      ReadLn(u, v, weight);
      if w[u, v] > weight then
      //Phòng trường hợp nhiều cung nối từ u tới v (đa đồ thị)
        w[u, v] := weight; //Chỉ ghi nhận cung trọng số nhỏ nhất
    end;
end;
procedure Floyd;
var k, u, v: Integer;
begin
  for u := 1 to n do
    for v := 1 to n do trace[u, v] := v;
    //Khởi tạo đường đi ngắn nhất là đường đi trực tiếp
  for k := 1 to n do
    for u := 1 to n do
      if w[u, k] < maxD then
        for v := 1 to n do
```

```
            if (w[k, v] < maxD)
                and (w[u, v] > w[u, k] + w[k, v]) then
            begin //Cực tiểu hóa c[u, v]
              w[u, v] := w[u, k] + w[k, v];
              //Ghi nhận đường đi vòng qua k
              trace[u, v] := trace[u, k]; //Lưu vết
            end;
end;
procedure PrintResult; //In kết quả
begin
  if w[s, t] = maxD then
    WriteLn('There is no path from ', s, ' to ', t)
  else
    begin
      WriteLn('Distance from ', s, ' to ', t, ':
                                          ', w[s, t]);
      while s <> t do
        begin
          Write(s, '->');
          s := trace[s, t];
        end;
      WriteLn(t);
    end;
end;
begin
  Enter;
  Floyd;
  PrintResult;
end.
```

Dễ thấy rằng thời gian thực hiện giải thuật Floyd là $\Theta(n^3)$ và chi phí bộ nhớ là $\Theta(n^2)$.

### b) Thuật toán Johnson

Trong trường hợp $G = (V, E)$ là đồ thị thưa gồm $n$ đỉnh và $m$ cạnh: $m \ll n^2$, thuật toán Johnson tìm đường đi ngắn nhất giữa mọi cặp đỉnh hoạt động hiệu quả hơn thuật toán Floyd. Bản chất của thuật toán Johnson là thực hiện thuật

toán Bellman-Ford để *gán lại trọng số* và thực hiện tiếp thuật toán Dijkstra để tìm đường đi ngắn nhất.

Nếu đồ thị không có cạnh trọng số âm, ta có thể thực hiện thuật toán Dijkstra $n$ lần với cách chọn lần lượt $n$ đỉnh làm đỉnh xuất phát. Bằng cách kết hợp với một hàng đợi ưu tiên được tổ chức dưới dạng Fibonacci Heap, thời gian thực hiện một lần thuật toán Dijkstra là $O(n \lg n + m)$, và như vậy ta có thể tìm đường đi ngắn nhất giữa mọi cặp đỉnh trong thời gian $O(n^2 \lg n + mn)$.

Nếu đồ thị có cạnh trọng số âm nhưng không có chu trình âm, thuật toán Johnson thực hiện một kỹ thuật gọi là *gán lại trọng số (re-weighting)*. Tức là trọng số $w: E \to \mathbb{R}$ sẽ được biến đổi thành trọng số $\widehat{w}: E \to \mathbb{R}$ thỏa mãn hai điều kiện sau đây:

- Với mọi cặp đỉnh $u, v \in V$, đường đi $P$ là đường đi ngắn nhất từ $u$ tới $v$ ứng với trọng số $w$ nếu và chỉ nếu $P$ cũng là đường đi ngắn nhất từ $u$ tới $v$ ứng với trọng số $\widehat{w}$
- Với mọi cạnh $e \in E$, trọng số $\widehat{w}(e)$ là một số không âm

***Định lý 1-2***

Cho đồ thị $G = (V, E)$ với trọng số $w: E \to \mathbb{R}$. Gọi $h: V \to \mathbb{R}$ là một hàm gán cho mỗi đỉnh $v$ một số thực $h(v)$. Xét trọng số $\widehat{w}: E \to \mathbb{R}$ định nghĩa bởi:

$$\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$$

Xét $p = \langle v_0, v_1, \dots, v_k \rangle$ là một đường đi từ $v_0$ tới $v_k$, khi đó $p$ là đường đi ngắn nhất từ $v_0$ tới $v_k$ với trọng số $w$ nếu và chỉ nếu $p$ là đường đi ngắn nhất từ $v_0$ tới $v_k$ với trọng số $\widehat{w}$. Hơn nữa $G$ có chu trình âm tương ứng với trọng số $w$ nếu và chỉ nếu $G$ có chu trình âm với trọng số $\widehat{w}$.

***Chứng minh***

Ký hiệu $w(p)$ và $\widehat{w}(p)$ lần lượt là độ dài đường đi $p$ với trọng số $w$ và $\widehat{w}$. Ta có

$$\begin{aligned}\widehat{w}(p) &= \sum_{i=1}^{k} \widehat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} \big(w(v_{i-1}, v_i) + h(v_{i-1}) - h(v)\big)\end{aligned}$$

$$= \left(\sum_{i=1}^{k} w(v_{i-1}, v_i)\right) + h(v_0) - h(v_k)$$
$$= w(p) + h(v_0) - h(v_k)$$

Vậy qua phép biến đổi trọng số, độ dài mọi đường đi từ $v_0$ tới $v_k$ sẽ được cộng thêm một lượng $h(v_0) - h(v_k)$. Hay nói cách khác, đường đi nào ngắn nhất với trọng số $w$ cũng là đường đi ngắn nhất với trọng số $\widehat{w}$.

Nếu $p$ là một chu trình, $v_0 = v_k$, ta suy ra $\widehat{w}(p) = w(p)$. Độ dài của chu trình được bảo toàn qua phép biến đổi trọng số, tức là $G$ có chu trình âm với trọng số $w$ nếu và chỉ nếu $G$ có chu trình âm với trọng số $\widehat{w}$.

Mục tiêu tiếp theo là chỉ ra một phép gán trọng số mới cho đồ thị $G$ để đảm bảo các trọng số không âm. Nếu $G$ không có chu trình âm ta thêm vào đồ thị một đỉnh giả $s$ và các cung nối từ đỉnh $s$ tới tất cả các đỉnh còn lại của đồ thị, trọng số của các cung này được đặt bằng 0. Do đỉnh $s$ không có cung đi vào, đồ thị mới tạo thành cũng không có chu trình âm. Thực hiện thuật toán Bellman-Ford trên đồ thị mới với đỉnh xuất phát $s$ để xác định các $h[v] = \delta(s, v)$ là độ dài đường đi ngắn nhất từ $s$ tới $v$ tương ứng với trọng số $c$.

***Định lý 1-3***

Trọng số $\widehat{w}: E \to \mathbb{R}$ xác định bởi $\widehat{w}(u, v) = w(u, v) + h[u] - h[v]$ là một hàm trọng số không âm.

***Chứng minh***

Khi thuật toán Bellman-Ford kết thúc, sẽ không tồn tại một cạnh $(u, v)$ nào mà

$$h(v) > h(u) + w(u, v)$$

hay nói cách khác, với mọi cạnh $(u, v)$ của đồ thị, ta có

$$w(u, v) + h(u) - h(v) \geq 0$$

Các nhận xét kể trên, đặc biệt là Định lý 1-2 và Định lý 1-3, cho ta mô hình cài đặt thuật toán Johnson:

- Thêm vào đồ thị một đỉnh $s$ và các cung trọng số 0 nối từ $s$ tới tất cả các đỉnh khác, dùng thuật toán Bellman-Ford tính các $h[v]$ là độ dài đường đi ngắn nhất từ $s$ tới $v$. Thời gian thực hiện giải thuật $O(nm)$
- Loại bỏ $s$ và các cung mới thêm vào khỏi đồ thị, gán lại trọng số cạnh $\widehat{w}(u, v) \coloneqq w(u, v) + h[u] - h[v]$ với mọi cạnh $(u, v) \in E$. Thời gian thực hiện giải thuật $\Theta(m)$

- Lần lượt lấy các đỉnh $v \in V$ làm đỉnh xuất phát, thực hiện thuật toán Dijkstra để tìm đường đi ngắn nhất từ $v$ tới tất cả các đỉnh khác. Thời gian thực hiện giải thuật $O(n^2 \lg n + nm)$ nếu sử dụng Fibonacci Heap

Vậy thuật toán Johnson tìm đường đi ngắn nhất giữa mọi cặp đỉnh có thể thực hiện trong thời gian $O(n^2 \lg n + nm)$. Thuật toán dựa trên hai thuật toán đã biết để tìm đường đi ngắn nhất xuất phát từ một đỉnh, việc cài đặt xin dành cho bạn đọc.

### *1.4. Một số chú ý*

Ở một số chương trình trong bài, đôi khi ta sử dụng ma trận trọng số và đem trọng số $+\infty$ gán cho những cạnh không có trong đồ thị ban đầu, hay khi khởi tạo các nhãn khoảng cách, chúng ta thường gán $d[v] := +\infty$ và cực tiểu hóa dần các nhãn đó. Trên máy tính thì không có khái niệm trừu tượng $+\infty$ nên ta sẽ phải chọn một số dương $maxD$ đủ lớn để thay. Như thế nào là đủ lớn?. Số đó phải đủ lớn hơn tất cả trọng số của các đường đi đơn để cho dù đường đi thật có tồi tệ đến đâu vẫn tốt hơn đường đi trực tiếp theo cạnh tưởng tượng ra đó.

Trong trường hợp đồ thị có cạnh trọng số âm, cần cẩn thận với phép cộng trọng số: Nếu một trong hai hạng tử là $maxD$, ta coi như tổng bằng $maxD$ ($C + \infty = \infty$) và không cần cộng nữa. Lý do thứ nhất là để hạn chế lỗi tràn số khi hằng số $maxD$ trong bài toán cụ thể quá lớn, lý do thứ hai là để không bị tính sai khi cộng $maxD$ với một số âm được kết quả $< maxD$, khi đó rất có thể $d[t] < maxD$ mặc dù không tồn tại đường đi từ $s$ tới $t$.

Những thuật toán tìm đường đi ngắn nhất bộc lộ rất rõ ưu, nhược điểm trong từng trường hợp cụ thể (Ví dụ như số đỉnh của đồ thị quá lớn làm cho không thể biểu diễn bằng ma trận trọng số thì thuật toán Floyd sẽ gặp khó khăn, hay thuật toán Ford-Bellman làm việc khá chậm). Vì vậy cần phải hiểu bản chất và thành thạo trong việc cài đặt tất cả các thuật toán trên để có thể sử dụng chúng một cách uyển chuyển trong từng bài toán thực tế.

## Bài tập

**2.1.** Cho đồ thị có hướng $G = (V, E)$ không có chu trình âm, hãy tìm thuật toán $O(|V||E|)$ để tính tất cả các $\delta^*(u) = \min_{v \in V}\{\delta(u, v)\}$

**2.2.** Cho đồ thị có hướng $G = (V, E)$ gồm $n$ đỉnh và $m$ cung, hãy tìm thuật toán $O(nm)$ để xác định đồ thị có chu trình âm hay không và chỉ ra một chu trình âm nếu có.

*Gợi ý:* Thực hiện thuật toán Bellman-Ford (tối đa $n - 1$ lần quét danh sách cạnh và thực hiện phép co), sau đó ta quét lại danh sách cạnh xem có thể thực hiện được phép co nào nữa hay không. Nếu còn có thể co theo cạnh $(u, v)$ nào đó, ta kết luận đồ thị có chu trình âm và thực hiện phép co này, sau đó lần ngược vết đường đi từ đỉnh $v$, nếu quá trình lần vết đi lặp lại một đỉnh $x$ nào đó, ta có một chu trình bắt đầu và kết thúc ở đỉnh $x$.

**2.3.** Hệ ràng buộc: Cho $x_1, x_2, \dots, x_n$ là các biến số, cho $m$ ràng buộc, mỗi ràng buộc có dạng:

$$x_j - x_i \leq w_{ij}, (w_{ij} \in \mathbb{R})$$

Vấn đề đặt ra là hãy tìm cách gán giá trị cho các biến $x_1, x_2, \dots, x_n$ thỏa mãn tất cả các ràng buộc đã cho.

*Gợi ý:* Có nhiều ví dụ về hệ ràng buộc trên thực tế, chẳng hạn một công trình xây dựng có $n$ công đoạn. Vì lý do kỹ thuật, một công đoạn $x_j$ không được bắt đầu muộn hơn $w_{ij}$ thời gian so với công đoạn $x_i$. Ràng buộc này có thể viết dưới dạng $x_j - x_i \leq w_{ij}$. Một dạng ràng buộc khác là công đoạn $x_j$ phải bắt đầu sau khi công đoạn $x_i$ bắt đầu được ít nhất $w_{ij}$ thời gian, ràng buộc này cũng có thể viết dưới dạng $x_j - x_i \geq w_{ij}$ hay $x_i - x_j \leq -w_{ij}$.

Coi mỗi biến là một đỉnh của đồ thị, mỗi ràng buộc $x_j - x_i \leq w_{ij}$ cho tương ứng với một cạnh $(x_i, x_j)$ có trọng số $w_{ij}$. Khi đó nếu đồ thị có chu trình âm thì không tồn tại giải pháp. Thật vậy, giả sử tồn tại chu trình âm, không giảm tính tổng quát, giả sử chu trình âm đó là $C = \langle x_1, x_2, \dots, x_k, x_1 \rangle$, ta có

$$x_2 - x_1 \leq w_{12}$$

$$x_3 - x_2 \le w_{23}$$
$$\dots$$
$$x_1 - x_k \le w_{k_1}$$

Tổng vế trái của các bất đẳng thức trên bằng 0 và tổng vế phải chính là trọng số của chu trình (âm), bất đẳng thức $0 \le w(c) < 0$ không thể được thỏa mãn.

Nếu đồ thị không có chu trình âm, ta thêm vào một đỉnh $s$ và các cung nối trọng số 0 từ $s$ tới mọi đỉnh khác. Dùng thuật toán Bellman-Ford để tìm đường đi ngắn nhất từ $s$, khi đó các nhãn $d[x_i] = \delta(s, x_i)$ chính là một cách gán giá trị thỏa mãn tất cả các ràng buộc. Hơn nữa cách này còn làm khoảng giá trị gán cho các biến là hẹp nhất:

$$\max_{1 \le i \le n}\{x_i\} - \min_{1 \le j \le n}\{x_j\} \to \min$$

**2.4.** (Cải tiến của Yen cho thuật toán Bellman-Ford) Với đồ thị có hướng $G = (V, E)$ gồm $n$ đỉnh, ta đánh số các đỉnh từ 1 tới $n$, chia tập cạnh $E$ làm hai tập con: $E_1$ gồm các cung nối từ đỉnh có chỉ số nhỏ tới đỉnh có chỉ số lớn và $E_2$ gồm các cung nối từ đỉnh có chỉ số lớn tới đỉnh có chỉ số nhỏ. Đặt $G_1 = (V, E_1)$ và $G_2 = (V, E_2)$, hai đồ thị này là đồ thị có hướng không có chu trình được biểu diễn bằng danh sách kề.

Thuật toán Bellman-Ford sau đó được thực hiện như sau:

```
Init;
repeat
  Stop := True;
  for u := 1 to n - 1 do
    for ∀v: (u, v)∈E1 do
      if Relax(u, v) then Stop := False;
  for u := n downto 2 do
    for ∀v: (u, v)∈E2 do
      if Relax(u, v) then Stop := False;
until Stop;
```

Bên trong vòng lặp repeat…until là hai pha tối ưu nhãn: pha thứ nhất xét các đỉnh theo thứ tự tăng dần còn pha thứ hai xét các đỉnh theo thứ tự giảm dần của chỉ số. Mỗi khi một đỉnh được xét và thực hiện phép co theo tất cả

các cung đi ra khỏi $u$, mỗi pha thực hiện tương tự như thuật toán tìm đường đi ngắn nhất trên đồ thị không có chu trình.

Chỉ ra rằng vòng lặp repeat…until trong cải tiến của Yen lặp không quá $\lceil n/2 \rceil$ lần. Cài đặt thuật toán và so sánh với cách cài đặt chuẩn của thuật toán Bellman-Ford.

**2.5.** Arbitrage là một cách sử dụng sự bất hợp lý trong hối đoái tiền tệ để kiếm lời. Ví dụ nếu 1$ mua được 0.7£, 1£ mua được 190¥, 1¥ mua được 0.009$ thì từ 1$, ta có thể đổi sang 0.7£, sau đó sang 0.7x190=133¥, rồi đổi lại sang 133x0.009=1.197$. Kiếm được 0.197$ lãi.

Giả sử rằng có $n$ loại tiền tệ đánh số từ 1 tới $n$. Bảng $R = \{r_{ij}\}_{n \times n}$ cho biết tỉ lệ hối đoái: một đơn vị tiền $i$ đổi được $r_{ij}$ đơn vị tiền $j$. Hãy tìm thuật toán để xác định xem có thể kiếm lời từ bảng tỉ giá hối đoái này bằng phương pháp arbitrage hay không? Nếu có thể sử dụng arbitrage, hãy chỉ ra một cách kiếm lời.

**2.6.** (Thuật toán Karp tìm chu trình có trung bình trọng số nhỏ nhất) Cho đồ thị có hướng $G = (V, E)$ gồm $n$ đỉnh, hàm trọng số $w: E \to \mathbb{R}$. Ta định nghĩa trung bình trọng số của một chu trình $C$ gồm các cạnh $\langle e_1, e_2, \dots, e_k \rangle$ là:

$$\mu(C) = \frac{1}{k} \sum_{i=1}^{k} w(e_i)$$

Đặt

$$\mu^* = \min_C \{\mu(C)\}$$

Khi đó chu trình $C$ có $\mu(C) = \mu^*$ gọi là chu trình có trung bình trọng số nhỏ nhất (*minimum mean-weight cycle)*. Chu trình có trung bình trọng số nhỏ nhất có nhiều ý nghĩa trong các thuật toán tìm luồng với chi phí cực tiểu.

Không giảm tính tổng quát, giả sử mọi đỉnh $v \in V$ đều đến được từ một đỉnh $s \in V$ (Ta có thể thêm một đỉnh giả $s$ và cung trọng số 0 nối từ $s$ tới mọi đỉnh khác, $s$ không nằm trên chu trình đơn nào nên không ảnh hưởng tới tính đúng đắn của thuật toán). Đặt $\delta(v)$ là độ dài đường đi ngắn nhất từ $s$ tới $v$. Đặt $\delta_k(v)$ là độ dài đường đi ngắn nhất trong số các đường đi từ $s$

tới $v$ qua đúng $k$ cạnh (ta có thể thêm vào các cung trọng số đủ lớn để với mọi cặp đỉnh $u, v$ luôn tồn tại cung $(u, v)$ và $(v, u)$, việc tìm chu trình trung bình trọng số nhỏ nhất không bị ảnh hưởng bởi những cung thêm vào và $\delta_k(v)$ luôn là giá trị hữu hạn).

a) Chứng minh rằng nếu $\mu^* = 0$, đồ thị $G$ không có chu trình âm và:

$$\delta(v) = \min_{0 \le k \le n-1} \{\delta_k(v)\}, \forall v \in V$$

b) Chứng minh rằng nếu $\mu^* = 0$ thì

$$\max_{0 \le k \le n-1} \frac{\delta_n(v) - \delta_k(v)}{n-k} \ge 0, \forall v \in V$$

(Gợi ý: Sử dụng kết quả câu a)

c) Gọi $C$ là một chu trình trọng số 0, $u, v$ là hai đỉnh nằm trên $C$, giả sử $\mu^* = 0$ và $x$ là độ dài đường đi từ $u$ tới $v$ dọc theo chu trình $C$. Chứng minh rằng

$$\delta(v) = \delta(u) + x$$

d) Chứng minh rằng nếu $\mu^* = 0$ thì trên mỗi chu trình trọng số 0 sẽ tồn tại một đỉnh $v$ sao cho:

$$\max_{0 \le k \le n-1} \frac{\delta_n(v) - \delta_k(v)}{n-k} = 0$$

e) Chứng minh rằng nếu $\mu^* = 0$ thì

$$\min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(v) - \delta_k(v)}{n-k} = 0$$

f) Chỉ ra rằng nếu chúng ta cộng thêm một hằng số $\Delta$ vào tất cả các trọng số cạnh thì $\mu^*$ tăng lên $\Delta$. Sử dụng tính chất này để chứng minh rằng

$$\mu^* = \min_{v \in V} \max_{0 \le k \le n-1} \frac{\delta_n(v) - \delta_k(v)}{n-k}$$

g) Tìm thuật toán $O(||V||E|)$ và lập chương trình để tính $\mu^*$ và chỉ ra một chu trình có trung bình trọng số nhỏ nhất.

**2.7.** Trên mặt phẳng cho $n$ đường tròn, đường tròn thứ $i$ được cho bởi bộ ba số thực $(x_i, y_i, r_i)$, $(x_i, y_i)$ là toạ độ tâm và $r_i$ là bán kính. Chi phí di chuyển trên mỗi đường tròn bằng 0. Chi phí di chuyển giữa hai đường tròn bằng

khoảng cách giữa chúng. Hãy tìm phương án di chuyển giữa hai đường tròn $s, t$ cho trước với chi phí ít nhất.

**2.8.** Thuật toán Dijkstra có thể sai nếu đồ thị có cạnh trọng số âm, hãy chỉ ra một ví dụ.

**2.9.** Cho đồ thị vô hướng $G = (V, E)$ có $n$ đỉnh và $m$ cạnh, các cạnh có trọng số là số nguyên trong phạm vi từ 0 tới $k$. Hãy thay đổi thuật toán Dijkstra để được thuật toán $O(kn + m)$ tìm đường đi ngắn nhất xuất phát từ một đỉnh $s \in V$.

*Gợi ý:* Để ý rằng nếu một nhãn $d[v] < +\infty$ thì nhãn này phải là số nguyên nằm trong khoảng $[0 \dots (n-1) \times k]$, đồng thời nếu xét nhãn khoảng cách của các đỉnh lấy ra khỏi hàng đợi ưu tiên thì các nhãn khoảng cách này được sắp xếp theo thứ tự không giảm. Ta tổ chức hàng đợi ưu tiên dưới dạng bảng băm: $q[0 \dots (n-1) \times k]$ trong đó $q[x]$ là chốt của một danh sách móc nối chứa các đỉnh $v$ mà $d[v] = x$. Khi đó các phép chèn, cập nhật trên hàng đợi ưu tiên chỉ mất thời gian $O(1)$. Phép lấy ra một phần tử trong hàng đợi ưu tiên tính tổng thể mất thời gian $O(kn)$.

**2.10.** Tương tự như Bài tập 2.9 nhưng hãy tìm một thuật toán $O\big((m+n)\log k\big)$

*Gợi ý*: Để ý rằng tại mỗi bước của thuật toán Dijkstra, có tối đa $k + 2$ giá trị khác nhau của các nhãn $d[v]$ trong hàng đợi ưu tiên. Mỗi giá trị $x$ sẽ cho tương ứng với một danh sách móc các nút $v$ mà $d[v] = x$, các chốt của danh sách móc nối được lưu trữ trong một Binary Heap, khi đó các phép đẩy vào, lấy ra, co nhãn khoảng cách được thực hiện trong thời gian $O(\log k)$.

**2.11.** (Thuật toán Gabow) Xét đồ thị $G = (V, E)$ có các trọng số cạnh là số tự nhiên: $w: E \to \mathbb{N}$. Giả sử rằng từ đỉnh xuất phát $s$ có đường đi tới mọi đỉnh khác. Gọi $k$ là trọng số lớn nhất của các cạnh trong $E$. Gọi $z = \lceil \lg(k+1) \rceil$, khi đó trọng số mỗi cạnh có thể được biểu diễn bằng một dãy $z$ bit. Với mỗi cạnh $e \in E$ mang trọng số $w(e)$, ta ký hiệu $w_i(e)$ là số tạo thành bằng $i$ bit đầu tiên của $w(e)$, tức là:

$$w_i(e) = w(e) \text{ div } 2^{z-i}, (\forall i = 1,2, \dots, z).$$

Ví dụ $z = 5$ và $w(e) = 11 = 01011_{(2)}$. Ta có:

$$w_1(e) = 0_{(2)} = 0$$
$$w_2(e) = 01_{(2)} = 1$$
$$w_3(e) = 010_{(2)} = 2$$
$$w_4(e) = 0101_{(2)} = 5$$
$$w_5(e) = 01011_{(2)} = 11$$

Định nghĩa $\delta_i(s,v)$ là độ dài đường đi ngắn nhất từ $s$ tới $v$ trên đồ thị $G$ với hàm trọng số $w_i$. Rõ ràng $w_z(e) = w(e)$ nên $\delta_z(s,v) = \delta(s,v)$ với $\forall v \in V$.

a) Giả sử rằng $\delta(s,v) \leq |E|$ với mọi đỉnh $v$ có đường đi từ $s$, tìm thuật toán $O(|E|)$ để xác định tất cả các $\delta(s,v)$. (*Gợi ý:* Sử dụng hàng đợi ưu tiên như trong ***Error! Reference source not found.***).

b) Chứng minh rằng các $\delta_1(s,v)$ có thể tính được trong thời gian $O(|E|)$. (*Gợi ý:* Chú ý rằng các trọng số $w_1(e) \in \{0,1\}$).

c) Chỉ ra rằng với mọi $i = 2,3,\dots,z$: $w_i(e) = 2.w_{i-1}(e)$ hoặc $w_i(e) = 2.w_{i-1}(e) + 1$. Từ đó chứng minh rằng:

$$2.\delta_{i-1}(s,v) \leq \delta_i(s,v) < 2.\delta_{i-1}(s,v) + |V|$$

(*Gợi ý: Độ dài đường đi ngắn nhất từ s tới v sẽ nhân đôi nếu ta nhân đôi các trọng số cạnh*)

d) Với mọi cạnh $e = (u,v) \in E$, định nghĩa:

$$\widehat{w}_i(e) = \widehat{w}_i(u,v) = w_i(u,v) + 2.\delta_{i-1}(s,u) - 2.\delta_{i-1}(s,v)$$

Chứng minh rằng với mọi đường đi $p: u \rightsquigarrow v$, ta có:

$$\widehat{w}_i(p) = w_i(p) + 2.\delta_{i-1}(s,u) - 2.\delta_{i-1}(s,v)$$

e) Định nghĩa $\hat{\delta}_i(s,v)$ là độ dài đường đi ngắn nhất từ $s$ tới $v$ trên đồ thị $G$ với hàm trọng số $\widehat{w}_i$. Chứng minh rằng với $i = 2,3,\dots,z$ và $\forall v \in V$:

$$\hat{\delta}_i(s,v) = \delta_i(s,v) - 2.\delta_{i-1}(s,v) \leq |E|$$

f) Tìm thuật toán tính các $\delta_i(s,v)$ từ các $\delta_{i-1}(s,v)$ trong thời gian $O(|E|)$. Từ đó chứng minh rằng có thể tìm đường đi ngắn nhất trên đồ thị $G$ trong thời gian $O(|E|.z) = O(|E| \lg k)$.

**2.12.** Cho một bảng các số tự nhiên kích thước $m \times n$. Từ một ô có thể di chuyển sang một ô kề cạnh với nó. Hãy tìm một cách đi từ ô $(x, y)$ ra một ô biên sao cho tổng các số ghi trên các ô đi qua là nhỏ nhất.

**2.13.** Cho một dãy số nguyên $A = (a_1, a_2, \dots, a_n)$. Hãy tìm một dãy con gồm nhiều nhất các phần tử của dãy đã cho mà tổng của hai phần tử liên tiếp là số nguyên tố.

**2.14.** Một công trình lớn được chia làm $n$ công đoạn. Công đoạn $i$ phải thực hiện mất thời gian $t_i$. Quan hệ giữa các công đoạn được cho bởi bảng $A = \{a_{ij}\}_{n \times n}$ trong đó $a_{ij} = 1$ nếu công đoạn $j$ chỉ được bắt đầu khi mà công đoạn $i$ đã hoàn thành và $a_{ij} = 0$ trong trường hợp ngược lại. Mỗi công đoạn khi bắt đầu cần thực hiện liên tục cho tới khi hoàn thành, hai công đoạn độc lập nhau có thể tiến hành song song, hãy bố trí lịch thực hiện các công đoạn sao cho thời gian hoàn thành cả công trình là sớm nhất, cho biết thời gian sớm nhất đó.

*Gợi ý:* Dựng đồ thị có hướng $G = (V, E)$, mỗi đỉnh tương ứng với một công đoạn, đỉnh $u$ có cung nối tới đỉnh $v$ nếu công đoạn $u$ phải hoàn thành trước khi công đoạn $v$ bắt đầu. Thêm vào $G$ một đỉnh $s$ và cung nối từ $s$ tới tất cả các đỉnh còn lại. Gán trọng số mỗi cung $(u, v)$ của đồ thị bằng $t_v$.

Nếu đồ thị có chu trình, không thể có cách xếp lịch, nếu đồ thị không có chu trình (DAG) tìm đường đi dài nhất xuất phát từ $s$ tới tất cả các đỉnh của đồ thị, khi đó nhãn khoảng cách $d[v]$ chính là thời điểm hoàn thành công đoạn $v$, ta chỉ cần xếp lịch để công đoạn $v$ được bắt đầu vào thời điểm $d[v] - t_v$ là xong.

**2.15.** Cho đồ thị $G = (V, E)$, các cạnh được gán trọng số không âm. Tìm thuật toán và viết chương trình tìm một chu trình có độ dài ngắn nhất trên $G$.

## 2. Cây khung nhỏ nhất

Cho $G = (V, E, w)$ là đồ thị vô hướng liên thông có trọng số. Với một cây khung $T$ của $G$, ta gọi trọng số của cây $T$, ký hiệu $w(T)$, là tổng trọng số các cạnh trong $T$. Bài toán đặt ra là trong số các cây khung của $G$, chỉ ra cây khung có trọng số nhỏ nhất, cây khung như vậy được gọi là cây khung nhỏ nhất (*minimum*

*spanning tree*) của đồ thị. Sau đây ta sẽ xét hai thuật toán thông dụng để giải bài toán cây khung nhỏ nhất của đơn đồ thị vô hướng có trọng số, cả hai thuật toán này đều là thuật toán tham lam.

### *2.1. Phương pháp chung*

Xét đồ thị vô hướng liên thông có trọng số $G = (V, E, w)$. Cả hai thuật toán để tìm cây khung ngắn nhất đều dựa trên một cách làm chung: Nở dần cây khung. Cách làm này được mô tả như sau: Thuật toán quản lý một tập các cạnh $A \subseteq E$ và cố gắng duy trì tính chất sau (tính bất biến vòng lặp):

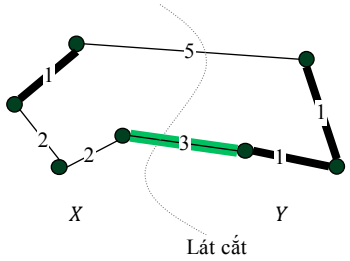
***A* luôn nằm trong tập cạnh của một cây khung nhỏ nhất.**

Tại mỗi bước lặp, thuật toán tìm một cạnh $(u, v)$ để thêm vào tập $A$ sao cho tính bất biến vòng lặp được duy trì, tức là $A \cup (u, v)$ phải nằm trong tập cạnh của một cây khung nhỏ nhất. Ta nói những cạnh $(u, v)$ như vậy là *an toàn* (*safe*) đối với tập $A$.

```
procedure FindMST; //Tìm cây khung ngắn nhất
begin
  A := ∅;
  while «A chưa phải cây khung» do
    begin
      «Tìm cạnh an toàn (u, v) đối với A»;
      A := A ∪ {(u, v)}; //Bổ sung (u, v) vào A
    end;
end;
```

Vấn đề còn lại là tìm một thuật toán hiệu quả để tìm cạnh an toàn đối với tập $A$. Chúng ta cần một số khái niệm để giải thích tính đúng đắn của những thuật toán sau này.

Một lát cắt (*cut*) trên đồ thị là một cách phân hoạch tập đỉnh $V$ thành hai tập rời nhau $X, Y$: $X \cup Y = V; X \cap Y = \emptyset$. Ta nói một lát cắt $V = X \cup Y$ tương thích với tập $A$ nếu không có cạnh nào của $A$ nối giữa một đỉnh thuộc $X$ và một đỉnh thuộc $Y$. Trong những cạnh nối $X$ với $Y$, ta gọi những cạnh có trọng số nhỏ nhất là những cạnh nhẹ (*light edge*) của lát cắt $V = X \cup Y$.




***Hình 2.3. Lát cắt và cạnh nhẹ***

### *Định lý 2-1*

Cho đồ thị vô hướng liên thông có trọng số $G = (V, E, w)$. Gọi $A$ là một tập con của tập cạnh của một cây khung nhỏ nhất và $V = X \cup Y$ là một lát cắt tương thích với $A$. Khi đó mỗi cạnh nhẹ của lát cắt $V = X \cup Y$ đều là cạnh an toàn đối với $A$.

***Chứng minh***

Gọi $(u, v)$ là một cạnh nhẹ của lát cắt $V = X \cup Y$, gọi $T$ là cây khung nhỏ nhất chứa tất cả các cạnh của $A$. Nếu $T$ chứa cạnh $(u, v)$, ta có điều phải chứng minh. Nếu $T$ không chứa cạnh $(u, v)$, ta thêm cạnh $(u, v)$ vào $T$ sẽ được một chu trình, trên chu trình này có đỉnh thuộc $X$ và cũng có đỉnh thuộc $Y$, vì vậy sẽ phải có ít nhất hai cạnh trên chu trình nối $X$ với $Y$. Ngoài cạnh $(u, v)$ nối $X$ với $Y$, ta gọi $(u', v')$ là một cạnh khác nối $X$ với $Y$ trên chu trình, theo giả thiết $(u, v)$ là cạnh nhẹ nên $w(u, v) \leq w(u', v')$. Ngoài ra do lát cắt $V = X \cup Y$ tương thích với $A$ nên $(u', v') \notin A$.

Cắt bỏ cạnh $(u', v')$ khỏi cây $T$, cây sẽ bị tách rời làm hai thành phần liên thông, sau đó thêm cạnh $(u, v)$ vào cây nối lại hai thành phần liên thông đó để được cây $T'$. Ta có

$$\begin{aligned} w(T') &= w(T) - w(u', v') + w(u, v) \\ &\leq w(T) \end{aligned}$$

Do $T$ là cây khung nhỏ nhất, $T'$ cũng phải là cây khung nhỏ nhất. Ngoài ra cây $T'$ chứa cạnh $(u, v)$ và tất cả các cạnh của $A$. Ta có điều phải chứng minh.

### *Hệ quả*

Cho đồ thị vô hướng liên thông có trọng số $G = (V, E, w)$. Gọi $A$ là một tập con của tập cạnh của một cây khung nhỏ nhất. Gọi $C$ là tập các đỉnh của một thành phần liên thông trên đồ thị $G_A = (V, A)$. Khi đó nếu $(u, v) \in E$ là cạnh trọng số

nhỏ nhất nối từ $C$ tới một thành phần liên thông khác thì $(u, v)$ là cạnh an toàn đối với $A$.

***Chứng minh***

Xét lát cắt $V = C \cup (V - C)$, lát cắt này tương thích với $A$ và cạnh $(u, v)$ là cạnh nhẹ của lát cắt này. Theo Định lý 3-17, $(u, v)$ an toàn đối với $A$.

Chúng ta sẽ trình bày hai thuật toán tìm cây khung nhỏ nhất trên đơn đồ thị vô hướng và cài đặt chương trình với khuôn dạng Input/Output như sau:

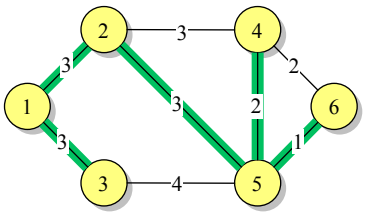
**Input**

- Dòng 1 chứa số đỉnh $n \leq 1000$ và số cạnh $m$ của đồ thị
- $m$ dòng tiếp theo, mỗi dòng chứa chỉ số hai đỉnh đầu mút và trọng số của một cạnh. Trọng số cạnh là số nguyên có giá trị tuyệt đối không quá 1000.

**Output**

Cây khung nhỏ nhất của đồ thị

| Sample Input | Sample Output |
|---|---|
| 6 8<br>1 2 3<br>1 3 3<br>2 4 3<br>2 5 3<br>3 5 4<br>4 5 2<br>4 6 2<br>5 6 1 | Minimum Spanning Tree:<br>(5, 6) = 1<br>(4, 5) = 2<br>(1, 2) = 3<br>(2, 5) = 3<br>(1, 3) = 3<br>Weight = 12 |




## *2.2. Thuật toán Kruskal*

Thuật toán Kruskal [**27**] dựa trên mô hình xây dựng cây khung bằng thuật toán hợp nhất, chỉ có điều thuật toán không phải xét các cạnh với thứ tự tuỳ ý mà xét các cạnh theo thứ tự đã sắp xếp: Để tìm cây khung ngắn nhất của đồ thị $G = (V, E, w)$, thuật toán khởi tạo cây $T$ ban đầu không có cạnh nào. Duyệt danh sách cạnh của đồ thị từ cạnh có trọng số nhỏ đến cạnh có trọng số lớn, mỗi khi xét tới một cạnh và việc thêm cạnh đó vào $T$ không tạo thành chu trình đơn trong $T$ thì kết nạp thêm cạnh đó vào $T$… Cứ làm như vậy cho tới khi:

- Hoặc đã kết nạp được $|V|-1$ cạnh vào trong $T$ thì ta được $T$ là cây khung nhỏ nhất
- Hoặc khi duyệt hết danh sách cạnh mà vẫn chưa kết nạp đủ $|V|-1$ cạnh. Trong trường hợp này đồ thị $G$ là không liên thông, việc tìm kiếm cây khung thất bại.

Như vậy cần làm rõ hai thao tác sau khi cài đặt thuật toán Kruskal:

- Làm thế nào để xét được các cạnh từ cạnh có trọng số nhỏ tới cạnh có trọng số lớn.
- Làm thế nào kiểm tra xem việc thêm một cạnh có tạo thành chu trình đơn trong T hay không.

### a) Duyệt danh sách cạnh

Vì các cạnh của đồ thị phải được xét từ cạnh có trọng số nhỏ tới cạnh có trọng số lớn. Ta có thể thực hiện một thuật toán sắp xếp danh sách cạnh rồi sau đó duyệt lại danh sách đã sắp xếp. Tuy nhiên khi cài đặt cụ thể, ta có thể khéo léo lồng thuật toán Kruskal vào QuickSort hoặc HeapSort để đạt hiệu quả cao hơn. Chẳng hạn với QuickSort, ý tưởng là sau khi phân đoạn danh sách cạnh bằng một cạnh chốt $Pivot$, ta được ba phân đoạn: Đoạn đầu gồm các cạnh có trọng số $\leq w(Pivot)$, tiếp theo là cạnh $Pivot$, đoạn sau gồm các cạnh có trọng số $\geq w(Pivot)$. Ta gọi đệ quy để sắp xếp và xử lý các cạnh thuộc đoạn đầu, tiếp theo xử lý cạnh $Pivot$, cuối cùng lại gọi đệ quy để sắp xếp và xử lý các cạnh thuộc đoạn sau. Dễ thấy rằng thứ tự xử lý các cạnh như vậy đúng theo thứ tự tăng dần của trọng số. Ngoài ra khi thấy đã có đủ $n-1$ cạnh được kết nạp vào cây khung, ta có thể ngưng ngay QuickSort mà không cần xử lý tiếp nữa.

### b) Kết nạp cạnh và hợp cây

Trong quá trình xây dựng cây khung, các cạnh trong $T$ ở các bước sẽ tạo thành một rừng (đồ thị không có chu trình đơn), mỗi thành phần liên thông của rừng này là một cây khung. Muốn thêm một cạnh $(u,v)$ vào $T$ mà không tạo thành chu trình đơn thì $(u,v)$ phải nối hai cây khác nhau của rừng $T$. Điều này làm chúng ta nghĩ đến cấu trúc dữ liệu biểu diễn các tập rời nhau: Ban đầu ta khởi tạo $n$ tập $S_1, S_2, \dots, S_n$, mỗi tập chứa đúng một đỉnh của đồ thị. Khi xét tới cạnh

$(u, v)$, nếu $u$ và $v$ thuộc hai tập khác nhau $S_u, S_v$ thì ta hợp nhất $S_u, S_v$ lại thành một tập.

Vậy có hai thao tác cần phải cài đặt hiệu quả trong thuật toán Kruskal: phép kiểm tra hai đỉnh có thuộc hai tập khác nhau hay không và phép hợp nhất hai tập. Một trong những cấu trúc dữ liệu hiệu quả để cài đặt những thao tác này là *rừng các tập rời nhau* (*disjoint-set forest*). Cấu trúc dữ liệu này được cài đặt như sau:

Mỗi tập $S[.]$ được biểu diễn bởi một cây, trong đó mỗi đỉnh trong tập tương ứng với một nút trên cây. Cây được biểu diễn bởi mảng con trỏ tới nút cha: $lab[v]$ là nút cha của nút $v$. Trong trường hợp $v$ là nút gốc của cây, ta đặt:

$$lab[v] := -\text{hạng của cây}$$

Hạng (*rank*) của một cây là một số nguyên không nhỏ hơn độ cao của cây. Ban đầu mỗi tập $S[.]$ chỉ gồm một đỉnh, nên họ các tập $S_1, S_2, \dots, S_n$ được khởi tạo với các nhãn $lab[v] := 0, \forall v \in V$ tương ứng với một rừng gồm $n$ cây độ cao 0.

Để xác định hai đỉnh $u, v$ có thuộc 2 tập khác nhau hay không, ta chỉ cần xác định xem gốc của cây chứa $u$ và gốc của cây chứa $v$ có khác nhau hay không. Việc xác định gốc của cây chứa $u$ được thực hiện bởi hàm $FindSet(u)$: Đi từ $u$ lên nút cha, đến khi gặp nút gốc (nút $r$ có $lab[r] \leq 0$) thì dừng lại. Đi kèm với hàm $FindSet(u)$ là phép nén đường (*path compression*): Dọc trên đường đi từ $u$ tới nút gốc $r$, đi qua đỉnh nào ta cho luôn đỉnh đó làm con của $r$:

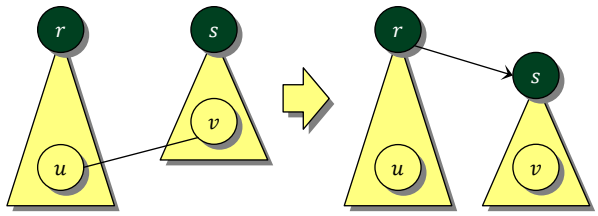
```
function FindSet(u: Integer): Integer;
//Xác định gốc cây chứa đỉnh u
begin
  if lab[u] <= 0 then Result := u
   //u là gốc của một tập S[.] nào đó, trả về chính u
  else //u không phải gốc
    begin
      Result := FindSet(lab[u]); //Gọi đệ quy tìm gốc
      lab[u] := Result; //Nén đường, cho u làm con của nút gốc luôn
    end;
end;
```

Việc hợp nhất hai tập tức là xây dựng cây mới chứa tất cả các phần tử trong hai cây ban đầu. Giả sử $r$ và $s$ là gốc của hai cây tương ứng với hai tập cần hợp nhất. Khi đó:

- Nếu cây gốc $r$ có hạng cao hơn cây gốc $s$, ta cho $s$ làm con của $r$, hạng của cây gốc $r$ không thay đổi, tương tự cho trường hợp cây gốc $r$ thấp hơn cây gốc $s$.
- Nếu hai cây ban đầu có cùng hạng, ta cho cây gốc $r$ làm con của gốc $s$ khi đó cây gốc $s$ có thể sẽ bị tăng độ cao, do vậy để hợp lý hóa ta tăng hạng của $s$ lên 1, tương đương với việc giảm $lab[s]$ đi 1.

```
procedure Union(r, s: Integer); //Hợp nhất hai tập r và s
begin
  if lab[r] < lab[s] then //hạng của r lớn hơn
    lab[s] := r //cho s làm con của r
  else
    begin
      if lab[r] = lab[s] then Dec(lab[s]);
      //Nếu hai tập bằng hạng, tăng hạng của s
      lab[r] := s; //Cho r làm con của s
    end;
end;
```

Hình 2.4 mô tả hai cây biểu diễn hai tập rời nhau, sau khi xét tới một cạnh $(u, v)$ nối giữa hai tập, hai cây được hợp nhất lại bằng cách cho một cây làm cây con của gốc cây kia.




*Hình 2.4. Hai tập rời nhau được hợp nhất lại khi xét tới một cạnh nối một đỉnh của tập này với một đỉnh của tập kia*

## KRUSKAL.PAS ✓ Thuật toán Kruskal

```
{$MODE OBJFPC}
program MinimumSpanningTree;
```

```
const
  maxN = 1000;
  maxM = maxN * (maxN - 1) div 2;
type
  TEdge = record //Cấu trúc cạnh
    x, y: Integer; //Hai đỉnh đầu mút
    w: Integer; //Trọng số
    Selected: Boolean; //Đánh dấu chọn/không chọn vào cây khung
  end;
var
  lab: array[1..maxN] of Integer; //Nhãn của disjoint set forest
  e: array[1..maxM] of TEdge; //Danh sách cạnh
  n, m, k: Integer;
procedure Enter; //Nhập dữ liệu
var i: Integer;
begin
  ReadLn(n, m);
  for i := 1 to m do
    with e[i] do
      begin
        ReadLn(x, y, w);
        Selected := False; //Chưa chọn cạnh nào
      end;
  for i := 1 to n do lab[i] := 0;
  //Khởi tạo n tập rời nhau hạng của mỗi tập bằng 0
  k := 0; //Biến đếm số cạnh được kết nạp vào cây khung
end;
function FindSet(u: Integer): Integer; //Xác định tập chứa đỉnh u
begin
  if lab[u] <= 0 then Result := u //u là gốc của một tập S[.] nào đó
  else //u không phải gốc
    begin
      Result := FindSet(lab[u]); //Gọi đệ quy tìm gốc
      lab[u] := Result; //Nén đường
    end;
end;
procedure Union(r, s: Integer); //Hợp nhất hai tập r và s
begin
```

```
  if lab[r] < lab[s] then //hạng của r lớn hơn
    lab[s] := r //cho s làm con của r
  else
    begin
      if lab[r] = lab[s] then Dec(lab[s]);
      //Nếu hai tập bằng hạng, tăng hạng của s
      lab[r] := s; //Cho r làm con của s
    end;
end;
procedure ProcessEdge(var e: TEdge); //Xử lý một cạnh e
var r, s: Integer;
begin
  with e do
    begin
      r := FindSet(x);
      s := FindSet(y); //Xác định 2 tập tương ứng với 2 đầu mút
      if r <> s then //Hai đầu mút thuộc hai tập khác nhau
        begin
          Selected := True; //Cạnh e sẽ được chọn vào cây khung nhỏ nhất
          Inc(k); //Tăng biến đếm số cạnh được kết nạp
          Union(r, s); //Hợp nhất hai tập thành một
        end;
    end;
end;
procedure QuickSort(L, H: Integer); //Xử lý danh sách cạnh e[L...H]
var
  i, j: Integer;
  pivot: TEdge;
begin
  //Nếu cây đã có đủ k cạnh hoặc danh sách cạnh rỗng thì thoát luôn
  if (L > H) or (k = n - 1) then Exit;
  //Chú ý L > H, không phải L ≥ H như trong QuickSort
  i := L + Random(H - L + 1);
  pivot := e[i];
  e[i] := e[L];
  i := L;
  j := H;
  repeat
```

```
    while (e[j].w > pivot.w) and (i < j) do Dec(j);
    if i < j then
      begin
        e[i] := e[j];
        Inc(i);
      end
    else Break;
    while (e[i].w < pivot.w) and (i < j) do Inc(i);
    if i < j then
      begin
        e[j] := e[i];
        Dec(j);
      end
    else Break;
  until i = j;
  QuickSort(L, i - 1);
  //Các cạnh e[L...i – 1] có trọng số ≤ Pivot.w, gọi đệ quy xử lý trước
  e[i] := Pivot;
  if k < n - 1 then ProcessEdge(e[i]); //Xử lý tiếp cạnh e[i] = Pivot
  QuickSort(i + 1, H);
  //Các cạnh e[i + 1...H] có trọng số ≥ Pivot.w, gọi đệ quy xử lý sau
end;
procedure PrintResult;
var
  i, Weight: Integer;
begin
  if k < n - 1 then //Không kết nạp đủ n – 1 cạnh, đồ thị không liên thông
    WriteLn('Graph is not connected!')
  else //In ra cây khung nhỏ nhất
    begin
      WriteLn('Minimum Spanning Tree:');
      Weight := 0;
      for i := 1 to m do
        with e[i] do
          if Selected then //In ra cách cạnh được đánh dấu chọn
            begin
              WriteLn('(', x, ', ', y, ') = ', w);
              Inc(Weight, w);
```

```
            end;
      WriteLn('Weight = ', Weight);
    end;
end;
begin
  Enter;
  QuickSort(1, m); //Lồng thuật toán Kruskal vào QuickSort
  PrintResult;
end.
```

Tính đúng đắn của thuật toán Kruskal được suy ra từ Định lý 3-17: Để ý rằng các cạnh được kết nạp vào cây khung sau mỗi bước sẽ tạo thành một rừng (đồ thị không có chu trình đơn). Mỗi khi cạnh $(u, v)$ được xét đến, nó sẽ chỉ được kết nạp vào cây khung nếu như $u$ và $v$ thuộc hai cây (hai thành phần liên thông) $T_u, T_v$ khác nhau. Ký hiệu $A$ là tập cạnh của $T_u$, khi đó lát cắt $V = T_u \cup (V - T_u)$ là tương thích với tập $A$, $(u, v)$ là cạnh nhẹ của lát cắt nên $(u, v)$ cũng phải là một cạnh trên một cây khung nhỏ nhất.

### c) Thời gian thực hiện giải thuật

Với hai số tự nhiên $m, n$, hàm Ackermann $A(m, n)$ được định nghĩa như sau:

$$A(m,n) = \begin{cases} n+1, \text{nếu } m = 0 \\ A(m-1,1), \text{nếu } m > 0 \text{ và } n = 0 \\ A\big(m-1, A(m,n-1)\big), \text{nếu } m > 0 \text{ và } n > 0 \end{cases}$$

Dưới đây là bảng một số giá trị hàm Ackermann:

| $m$: \ $n$: | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 2 | 3 | 5 | 7 | 9 | 11 |
| 3 | 5 | 13 | 29 | 61 | 125 |
| 4 | 13 | 65533 | $2^{65536}-3$ | $2^{2^{65536}}-3$ | $2^{2^{2^{65536}}}-3$ |

Hàm $A(m,n)$ là một hàm tăng rất nhanh theo đối số $n$. Có thể chứng minh được

$$A(0,n) = n+1$$
$$A(1,n) = n+2$$
$$A(2,n) = 2n+3$$
$$A(3,n) = 2^{n+3}-3$$
$$A(4,n) = \underbrace{2^{2^{\cdots^{2}}}}_{n+3 \text{ lũy thừa}} - 3$$

Chẳng hạn $A(4,2)$ là một số có 19729 chữ số, $A(4,4)$ là một số mà số chữ số của nó lớn hơn cả số nguyên tử trong phần vũ trụ mà con người biết đến.

Khi $n > 0$, xét hàm $\alpha(m,n)$, gọi là nghịch đảo của hàm Ackerman, định nghĩa như sau:

$$\alpha(m,n) = \min\left\{k \geq 1 : A\left(k, \left\lfloor \frac{m}{n} \right\rfloor\right) \geq \lg n\right\}$$

Người ta đã chứng minh được rằng với cấu trúc dữ liệu rừng các tập rời nhau, việc thực hiện $m$ thao tác $FindSet$ và $Union$ mất thời gian $O(m\alpha(m,n))$. Ở đây $\alpha(m,n)$ là một hằng số rất nhỏ (trên tất cả các dữ liệu thực thế, không bao giờ $\alpha(m,n)$ vượt quá 4). Điều đó chỉ ra rằng ngoại trừ việc sắp xếp danh sách cạnh, thuật toán Kruskal ở trên có thời gian thực hiện $O(|E|\alpha(|E|,|V|))$.

Tuy nhiên nếu phải thực hiện sắp xếp danh sách cạnh, chúng ta cần cộng thêm thời gian thực hiện giải thuật sắp xếp $O(|E|\lg|E|)$ nữa.

## *2.3. Thuật toán Prim*

### a) Tư tưởng của thuật toán

Trong trường hợp đồ thị dày (có nhiều cạnh), có một thuật toán hiệu quả hơn để tìm cây khung ngắn nhất là thuật toán Prim [**32**]. Với một cây khung $T$ và một đỉnh $v \notin T$, ta gọi khoảng cách từ $v$ tới $T$, ký hiệu $d[v]$, là trọng số nhỏ nhất của một cạnh nối $v$ với một đỉnh nằm trong $T$:

$$d[v] = \min_{u \in T}\{w(u, v)\}$$

Tư tưởng của thuật toán có thể trình bày như sau: Ban đầu khởi tạo một cây $T$ chỉ gồm 1 đỉnh bất kỳ của đồ thị, sau đó ta cứ tìm đỉnh gần $T$ nhất (có khoảng cách tới $T$ ngắn nhất) kết nạp vào $T$ và kết nạp luôn cạnh tạo ra khoảng cách gần nhất đó, cứ làm như vậy cho tới khi:

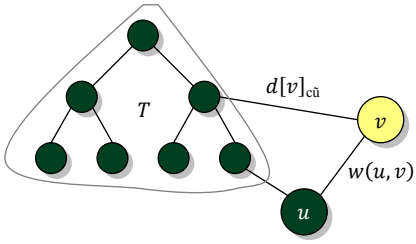
- Hoặc đã kết nạp đủ $n$ đỉnh vào $T$, ta có một cây khung ngắn nhất.
- Hoặc chưa kết nạp đủ $n$ đỉnh nhưng không còn cạnh nào nối một đỉnh trong $T$ với một đỉnh ngoài $T$. Ta kết luận đồ thị không liên thông và không thể tồn tại cây khung.

### b) Kỹ thuật cài đặt

Khi cài đặt thuật toán Prim, ta sử dụng các nhãn khoảng cách $d[v]$ để lưu khoảng cách từ $v$ tới $T$ tại mỗi bước. Mỗi khi cây $T$ bổ sung thêm một đỉnh $u$, ta tính lại các nhãn khoảng cách theo công thức sau:

$$d[v]_{\text{mới}} := \min\{d[v]_{\text{cũ}}, w(u, v)\}$$

Tính đúng đắn của công thức có thể hình dung như sau: $d[v]$ là khoảng cách từ $v$ tới cây $T$, theo định nghĩa là trọng số nhỏ nhất trong số các cạnh nối $v$ với một đỉnh nằm trong $T$. Khi cây $T$ "nở" ra thêm đỉnh $u$ nữa mà đỉnh $u$ này lại gần $v$ hơn tất cả các đỉnh khác trong $T$, ta ghi nhận khoảng cách mới $d[v]$ là trọng số cạnh $(u, v)$, nếu không ta vẫn giữ khoảng cách cũ.




***Hình 2.5. Cơ chế cập nhật nhãn khoảng cách***

Tại mỗi bước, đỉnh ngoài cây có nhãn khoảng cách nhỏ nhất sẽ được kết nạp vào cây, sau đó các nhãn khoảng cách được cập nhật và lặp lại. Mô hình cài đặt của thuật toán có thể viết như sau:

```
u := «Một đỉnh bất kỳ»;
T := {u};
for ∀v ∉ T do d[v] := +∞;
//Các đỉnh ngoài T được khởi tạo nhãn khoảng cách +∞
for i := 1 to n - 1 do //Làm n – 1 lần
   begin
     for (∀v ∉ T:(u, v) ∈ E) do
       d[v] := min{d[v], w(u, v)};
       //Cập nhật nhãn khoảng cách của các đỉnh kề u nằm ngoài T
     u := arg min{d[v]:v ∉ T};
     //Chọn u là đỉnh có nhãn khoảng cách nhỏ nhất trong số các đỉnh nằm ngoài T
     if d[u] = +∞ then //Đồ thị không liên thông
       begin
         Output ← "Không tồn tại cây khung";
         Break;
       end;
     T := T ∪ {u}; //Bổ sung u vào T
   end;
Output ← T;
```

Cài đặt dưới đây sử dụng ma trận trọng số để biểu diễn đồ thị. Kỹ thuật đánh dấu được sử dụng để biết một đỉnh $v$ đang nằm ngoài ($outside[v] = True$) hay nằm trong cây $T$ ($outside[v] = False$). Ngoài ra để tiện lợi hơn trong việc chỉ ra cây khung nhỏ nhất, với mỗi đỉnh $v$ nằm ngoài $T$ ta lưu lại $trace[v]$ là đỉnh $u$ nằm trong $T$ mà cạnh $(u, v)$ tạo ra khoảng cách gần nhất từ $v$ tới $T$: $w(u, v) =$

$d[v]$. Khi thuật toán kết thúc, các cạnh trong cây khung là những cạnh $(trace[v], v)$.

💻 PRIM.PAS ✓ Thuật toán Prim

```
{$MODE OBJFPC}
program MinimumSpanningTree;
const
  maxN = 1000;
  maxW = 1000;
var
  w: array[1..maxN, 1..maxN] of Integer; //Ma trận trọng số
  d: array[1..maxN] of Integer; //Các nhãn khoảng cách
  outside: array[1..maxN] of Boolean; //Đánh dấu các đỉnh ngoài cây
  trace: array[1..maxN] of Integer; //Vết
  n: Integer;
procedure Enter;
var
  i, m, u, v: Integer;
begin
  ReadLn(n, m);
  for u := 1 to n do
    for v := 1 to n do w[u, v] := maxW + 1;
    //Khởi tạo ma trận trọng số với các phần tử +∞
  for i := 1 to m do
    begin
      ReadLn(u, v, w[u, v]);
      //Chú ý: Đơn đồ thị mới có thể đọc dữ liệu thế này
      w[v, u] := w[u, v]; //Đồ thị vô hướng
    end;
end;
function Prim: Boolean; //Thuật toán Prim, trả về True nếu tìm được cây khung
var
  u, v, dmin, i: Integer;
begin
  u := 1; //Cây ban đầu chỉ gồm đỉnh 1
  for v := 2 to n do d[v] := maxW + 1;
  //Nhãn khoảng cách cho các đỉnh ngoài cây khởi tạo bằng +∞
  FillChar(outside[2],
```

```
            (n - 1) * SizeOf(outside[2]),
            True); //Đánh dấu các đỉnh 2...n nằm ngoài cây
  outside[1] := False; //Đỉnh 1 nằm trong cây
  for i := 1 to n - 1 do //Làm n – 1 lần
    begin
      //Trước hết tính lại các nhãn khoảng cách
      for v := 1 to n do
        if outside[v] and (d[v] > w[u, v]) then
        //Cạnh (u, v) tạo khoảng cách ngắn hơn khoảng cách cũ
          begin
            d[v] := w[u, v]; //Cập nhật nhãn khoảng cách
            trace[v] := u; //Lưu vết
          end;
      //Tìm đỉnh u ngoài cây có nhãn khoảng cách nhỏ nhất
      dmin := maxW + 1;
      u := 0;
      for v := 1 to n do
        if outside[v] and (d[v] < dmin) then
          begin
            dmin := d[v];
            u := v;
          end;
      if u = 0 then Exit(False);
      //Cây không có cạnh nào nối ra ngoài, đồ thị không liên thông, thoát
      outside[u] := False; //Kết nạp u vào cây
    end;
  Result := True;
end;
procedure PrintResult; //In kết quả trong trường hợp tìm ra cây khung nhỏ nhất
var
  v, Weight: Integer;
begin
  WriteLn('Minimum Spanning Tree:');
  Weight := 0;
  for v := 2 to n do
    begin
      WriteLn('(', trace[v], ', ', v, ') = ',
                 w[trace[v], v]);
      Inc(Weight, w[trace[v], v]);
```

```
    end;
  WriteLn('Weight = ', Weight);
end;
begin
  Enter;
  if Prim then PrintResult
  else WriteLn('Graph is not connected!');
end.
```

Tính đúng đắn của thuật toán Prim cũng dễ dàng suy ra được từ Định lý 3-17: Gọi $A$ là tập cạnh của cây $T$ tại mỗi bước, xét lát cắt tách tập đỉnh $V$ làm 2 tập rời nhau, một tập gồm các đỉnh $\in T$ và tập còn lại gồm các đỉnh $\notin T$. Đỉnh $v$ được kết nạp vào cây $T$ tại mỗi bước tương ứng với cạnh $(Trace[v], v)$ là cạnh nhẹ của lát cắt nên nó là an toàn với tập $A$, việc bổ sung cạnh này vào $A$ vẫn đảm bảo $A$ là tập con của tập cạnh một cây khung ngắn nhất.

### c) Thời gian thực hiện giải thuật

Chương trình cài đặt thuật toán Prim ở trên có thời gian thực hiện $\Theta(n^2)$, hiệu quả hơn thuật toán Kruskal trong trường hợp đồ thị dày nhưng lại kém hơn nếu đồ thị thưa. Trong trường hợp đồ thị thưa, ta có thể cải tiến mô hình cài đặt thuật toán Prim bằng cách kết hợp với một hàng đợi ưu tiên chứa các đỉnh ngoài cây có nhãn khoảng cách $< +\infty$. Hàng đợi ưu tiên cần hỗ trợ các thao tác sau:

- $Extract$: Lấy ra một đỉnh ưu tiên nhất (đỉnh $u$ có $d[u]$ nhỏ nhất) khỏi hàng đợi ưu tiên.
- $Update(v)$: Thao tác này báo cho hàng đợi ưu tiên biết rằng nhãn $d[v]$ đã bị giảm đi, cần tổ chức lại (thêm $v$ vào hàng đợi ưu tiên nếu $v$ đang nằm ngoài).

Khi đó thuật toán Prim có thể viết theo mô hình mới:

```
T := {Một đỉnh bất kỳ u};
for ∀v ∉ T do d[v] := +∞;
//Các đỉnh ngoài T được khởi tạo nhãn khoảng cách +∞
PQ := ∅; //Hàng đợi ưu tiên được khởi tạo rỗng
for i := 1 to |V| - 1 do //Làm n - 1 lần
   begin
     for (∀v ∈ V:(u, v)∈ E) do //Cập nhật nhãn các đỉnh ngoài cây kề với u
```

```
        if (v ∉ T) and (d[v] > w(u, v)) then
           begin
             d[v] := w(u, v);
             Update(v);
           end;
      if PQ = ∅ then //Đồ thị không liên thông
        begin
          Output ← "Không tồn tại cây khung";
          Break;
        end;
      u := Extract;
      //Chọn u là đỉnh có nhãn khoảng cách nhỏ nhất trong số các đỉnh nằm ngoài T
      T := T ∪ {u}; //Bổ sung u vào T
    end;
```

Thời gian thực hiện giải thuật có thể ước lượng theo số lời gọi $Update$ và $Extract$. Tương tự như thuật toán Dijkstra, có thể thấy rằng chúng ta cần sử dụng không quá $n-1$ lời gọi $Extract$ và không quá $m$ lời gọi $Update$ với $n$ là số đỉnh và $m$ là số cạnh của đồ thị.

Một số cấu trúc dữ liệu biểu diễn hàng đợi ưu tiên có thể sử dụng để cải thiện tốc độ thuật toán Prim trong trường hợp đồ thị thưa. Chẳng hạn nếu sử dụng Fibonacci Heap làm hàng đợi ưu tiên, thời gian thực hiện giải thuật là $O(|V|\lg|V| + |E|)$. Mặc dù vậy cần nhấn mạnh rằng trong các ứng dụng thực tế mà danh sách cạnh có thể được sắp xếp trong thời gian $O(|E|)$ (chẳng hạn dùng các thuật toán sắp xếp cơ số hoặc đếm phân phối), thuật toán Kruskal luôn là sự lựa chọn hợp lý hơn cả vì nó có thể tìm được cây khun trong thời gian $O\big(|E|\alpha(|E|,|V|)\big)$.

## Bài tập

**2.16.** Cho $T$ là cây khung nhỏ nhất của đồ thị $G$ và $(u, v)$ là một cạnh trong $T$. Chứng minh rằng nếu ta trừ trọng số cạnh $(u, v)$ đi một số dương thì $T$ vẫn là cây khung nhỏ nhất của đồ thị $G$.

**2.17.** Cho $G$ là một đồ thị vô hướng liên thông, $C$ là một chu trình trên $G$ và $e$ là cạnh trọng số lớn nhất của $C$. Chứng minh rằng nếu ta loại bỏ cạnh $e$ khỏi đồ thị thì không ảnh hưởng tới trọng số của cây khung nhỏ nhất.

**2.18.** Chứng minh rằng đồ thị có duy nhất một cây khung nhỏ nhất nếu với mọi lát cắt của đồ thị, có duy nhất một cạnh nhẹ nối hai tập của lát cắt. Cho một ví dụ để chỉ ra rằng điều ngược lại không đúng.

**2.19.** Gọi $T$ là một cây khung nhỏ nhất của đồ thị vô hướng liên thông $G$, ta giảm trọng số của một cạnh không nằm trong cây $T$, hãy tìm một thuật toán đơn giản để tìm cây khung của đồ thị mới.

*Gợi ý: Gọi cạnh bị giảm trọng số là $(u, v)$, thêm $(u, v)$ vào $T$ ta sẽ được đúng một chu trình đơn, loại bỏ cạnh trọng số lớn nhất trên chu trình đơn này sẽ được cây khung nhỏ nhất của đồ thị mới.*

**2.20.** Giả sử rằng đồ thị vô hướng liên thông $G$ có cây khung nhỏ nhất $T$, người ta thêm vào đồ thị một đỉnh mới và một số cạnh liên thuộc với đỉnh đó. Tìm thuật toán xác định cây khung nhỏ nhất của đồ thị mới.

**2.21.** Giáo sư X đề xuất một thuật toán tìm cây khung ngắn nhất dựa trên ý tưởng chia để trị: Với đồ thị vô hướng liên thông $G = (V, E)$, phân hoạch tập đỉnh $V$ làm hai tập rời nhau $V_1, V_2$ mà lực lượng của hai tập này hơn kém nhau không quá 1. Gọi $E_1$ là tập các cạnh chỉ liên thuộc với các đỉnh $\in V_1$ và $E_2$ là tập các cạnh chỉ liên thuộc với các đỉnh $\in V_2$. Tìm cây khung nhỏ nhất trên đồ thị $G_1 = (V_1, E_1)$ và $G_2 = (V_2, E_2)$ bằng thuật toán đệ quy, sau đó chọn cạnh trọng số nhỏ nhất nối $V_1$ với $V_2$ để nối hai cây khung tìm được thành một cây. Chứng minh tính đúng đắn của thuật toán hoặc chỉ ra một phản ví dụ cho thấy thuật toán sai.

**2.22.** (Cây khung nhỏ thứ nhì) Cho $G = (V, E, w)$ là đồ thị vô hướng liên thông có trọng số, giả sử rằng $|E| \geq |V|$ và các trọng số cạnh là hoàn toàn phân biệt ($w$ là đơn ánh). Gọi $\mathcal{T}$ là tập tất cả các cây khung của $G$ và $A$ là cây khung nhỏ nhất của $G$, khi đó cây khung nhỏ thứ nhì được định nghĩa là cây khung $B \in \mathcal{T}$ thỏa mãn:

$$w(B) = \min_{T \in \mathcal{T} - \{A\}} \{w(T)\}$$

- Chỉ ra rằng đồ thị $G$ có duy nhất một cây khung nhỏ nhất là $A$, nhưng có thể có nhiều cây khung nhỏ thứ nhì.
- Chứng minh rằng luôn tồn tại một cạnh $(u, v) \in A$ và $(x, y) \notin A$ để nếu ta loại bỏ cạnh $(u, v)$ khỏi $A$ rồi thêm cạnh $(x, y)$ vào $A$ thì sẽ được cây khung nhỏ thứ nhì.
- Với $\forall u, v \in V$, gọi $f[u, v]$ là cạnh mang trọng số lớn nhất trên đường đi duy nhất từ $u$ tới $v$ trên cây $A$. Tìm thuật toán $O(|V|^2)$ để tính tất cả các $f[u, v]$, $\forall u, v \in V$.
- Tìm thuật toán hiệu quả để tìm cây khung nhỏ thứ nhì của đồ thị.

**2.23.** Cho $s$ và $t$ là hai đỉnh của một đồ thị vô hướng có trọng số $G = (V, E, w)$. Tìm một đường đi từ $s$ tới $t$ thỏa mãn: Trọng số cạnh lớn nhất đi qua trên đường đi là nhỏ nhất có thể.

*Gợi ý: Có rất nhiều cách làm: Kết hợp một thuật toán tìm kiếm trên đồ thị với thuật toán tìm kiếm nhị phân, hoặc sửa đổi thuật toán Dijkstra, hoặc sử dụng thuật toán tìm cây khung ngắn nhất.*

**2.24.** (Euclidean Minimum Spanning Tree) Trong trường hợp các đỉnh của đồ thị đầy đủ được đặt trên mặt phẳng trực chuẩn và trọng số cạnh nối giữa hai đỉnh chính là khoảng cách hình học giữa chúng. Người ta có một phép tiền xử lý để giảm bớt số cạnh của đồ thị bằng thuật toán tam giác phân Delaunay ($O(n \lg n)$), đồ thị sau phép tam giác phân Delaunay sẽ còn không quá $3n$ cạnh, do đó sẽ làm các thuật toán tìm cây khung nhỏ nhất hoạt động hiệu quả hơn. Hãy tự tìm hiểu về phép tam giác phân Delaunay và cài đặt chương trình để tìm cây khung nhỏ nhất.

**2.25.** Trên một nền phẳng với hệ toạ độ trực chuẩn đặt $n$ máy tính, máy tính thứ $i$ được đặt ở toạ độ $(x_i, y_i)$. Đã có sẵn một số dây cáp mạng nối giữa một số cặp máy tính. Cho phép nối thêm các dây cáp mạng nối giữa từng cặp máy tính. Chi phí nối một dây cáp mạng tỉ lệ thuận với khoảng cách giữa hai máy cần nối. Hãy tìm cách nối thêm các dây cáp mạng để cho các máy tính trong toàn mạng là liên thông và chi phí nối mạng là nhỏ nhất.

**2.26.** Hệ thống điện trong thành phố được cho bởi $n$ trạm biến thế và các đường dây điện nối giữa các cặp trạm biến thế. Mỗi đường dây điện $e$ có độ an toàn là $p(e) \in (0,1]$. Độ an toàn của cả lưới điện là tích độ an toàn trên các

đường dây. Hãy tìm cách bỏ đi một số dây điện để cho các trạm biến thế vẫn liên thông và độ an toàn của mạng là lớn nhất có thể.

*Gợi ý:* Bằng kỹ thuật lấy logarithm, độ an toàn trên lưới điện trở thành tổng độ an toàn trên các đường dây.

## 3. Luồng cực đại trên mạng

### *3.1. Các khái niệm và bài toán*

#### a) Mạng

*Mạng* (*flow network*) là một bộ năm $G = (V, E, c, s, t)$, trong đó:

$V$ và $E$ lần lượt là tập đỉnh và tập cung của một đồ thị có hướng không có khuyên (cung nối từ một đỉnh đến chính nó).

$s$ và $t$ là hai đỉnh phân biệt thuộc $V$, $s$ gọi là *đỉnh phát* (*source*) và $t$ gọi là *đỉnh thu* (*sink*).

$c$ là một hàm xác định trên tập cung $E$:

$$c: E \longrightarrow [0, +\infty)$$
$$e \longmapsto c(e)$$

gán cho mỗi cung $e \in E$ một số không âm gọi là *sức chứa (capacity)*[*] $c(e) \geq 0$.

Bằng cách thêm vào mạng một số cung có sức chứa 0, ta có thể giả thiết rằng mỗi cung $e = (u, v) \in E$ luôn có tương ứng duy nhất một cung ngược chiều, ký hiệu $-e = (v, u) \in E$, gọi là *cung đối* của cung $e$, ta cũng coi $e$ là cung đối của cung $-e$ (tức là $-(-e) = e$). Có thể thấy rằng số cung cần thêm vào mạng là một đại lượng $O(E)$.

Chú ý rằng **mạng là một đa đồ thị**, tức là giữa hai đỉnh có thể có nhiều cung nối.

Để thuận tiện cho việc trình bày, ta quy ước các ký hiệu sau:

[*] Từ này còn có thể dịch là "khả năng thông qua" hay "lưu lượng"

Với $X, Y$ là hai tập con của tập đỉnh $V$ và $f: E \to \mathbb{R}$ là một hàm xác định trên tập cạnh $E$:

Ký hiệu $\{X \to Y\}$ là tập các cung nối một từ một đỉnh thuộc $X$ tới một đỉnh thuộc $Y$:

$$\{X \to Y\} = \{e = (u, v) \in E: u \in X, y \in Y\}$$

Ký hiệu $f(X, Y)$ là tổng các giá trị hàm $f$ trên các cung $e \in \{X \to Y\}$:

$$f(X, Y) = \sum_{e \in X \to Y} f(e)$$

### b) Luồng

*Luồng* (*flow*) trên mạng $G$ là một hàm:

$$f: E \longrightarrow \mathbb{R}$$
$$e \longmapsto f(e)$$

gán cho mỗi cung $e$ một số thực $f(e)$, gọi là luồng trên cung $e$, thỏa mãn ba ràng buộc sau đây:

- Ràng buộc về sức chứa (*Capacity constraint*): Luồng trên mỗi cung không được vượt quá sức chứa của cung đó: $\forall e \in E: f(e) \leq c(e)$.
- Ràng buộc về tính đối xứng lệch (*Skew symmetry*): Với $\forall e \in E$, luồng trên cung $e$ và luồng trên cung đối $-e$ có cùng giá trị tuyệt đối nhưng trái dấu nhau: $\forall e \in E: f(e) = -f(-e)$.
- Ràng buộc về tính bảo tồn (*Flow conservation*): Với mỗi đỉnh $v$ không phải đỉnh phát và cũng không phải đỉnh thu, tổng luồng trên các cung đi ra khỏi $v$ bằng 0: $\forall v \in V - \{s, t\}: f(\{v\}, V) = 0$.

Từ ràng buộc về tính đối xứng lệch và tính bảo tồn, ta suy ra được: Với mọi đỉnh $v \in V - \{s, t\}$, tổng luồng trên các cung đi vào $v$ bằng 0: $f(V, \{v\}) = 0$.

Giá trị của luồng $f$ trên mạng $G$ được định nghĩa bằng tổng luồng trên các cung đi ra khỏi đỉnh phát:

$$|f| = f(\{s\}, V) \qquad (3.1)$$

*Bài toán luồng cực đại trên mạng* (*maximum-flow problem*): Cho một mạng $G$ với đỉnh phát $s$ và đỉnh thu $t$, hàm sức chứa $c$, hãy tìm một luồng có giá trị lớn nhất trên mạng $G$.

### c) Luồng dương

*Luồng dương* (*positive flow*) trên mạng $G$ là một hàm

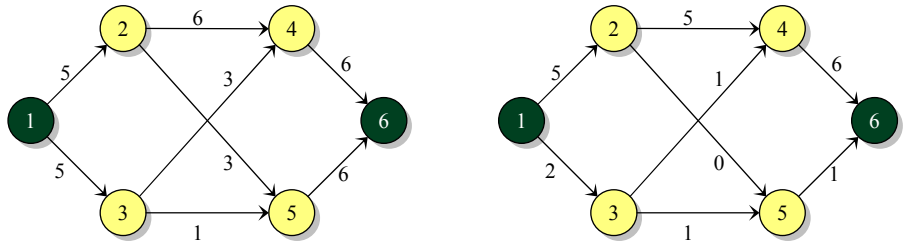
$$\varphi: E \longrightarrow [0, +\infty)$$
$$e \longmapsto \varphi(e)$$

gán cho mỗi cung $e$ một số thực không âm $\varphi(e)$ gọi là luồng dương trên cung $e$ thỏa mãn hai ràng buộc sau đây:

- Ràng buộc về sức chứa (*Capacity constraint*): Luồng dương trên mỗi cung không được vượt quá sức chứa của cung đó: $\forall e \in E: 0 \leq \varphi(e) \leq c(e)$.
- Ràng buộc về tính bảo tồn (*Flow conservation*): Với mỗi đỉnh $v$ không phải đỉnh phát và cũng không phải đỉnh thu, tổng luồng dương trên các cung đi vào $v$ bằng tổng luồng dương trên các cung đi ra khỏi $v$: $\forall v \in V - \{s, t\}$: $\varphi(V, \{v\}) = \varphi(\{v\}, V)$.

Giá trị của một luồng dương được định nghĩa bằng tổng luồng dương trên các cung đi ra khỏi đỉnh phát trừ đi tổng luồng dương trên các cung đi vào đỉnh phát*:

$$|\varphi| = \varphi(\{s\}, V) - \varphi(V, \{s\}) \qquad (3.2)$$




***Hình 2.6. Mạng với các sức chứa trên cung (1 phát, 6 thu) và một luồng dương với giá trị 7***

* Một số tài liệu khác đưa vào thêm ràng buộc: đỉnh phát $s$ không có cung đi vào và đỉnh thu $t$ không có cung đi ra. Khi đó giá trị luồng dương bằng tổng luồng dương trên các cung đi ra khỏi đỉnh phát. Cách hiểu này có thể quy về một trường hợp riêng của định nghĩa.

## d) Mối quan hệ giữa luồng và luồng dương

***Bổ đề 3-1***

Cho $\varphi: E \to \mathbb{R}$ là một luồng dương trên mạng $G = (V, E, c, s, t)$. Khi đó hàm

$$f: E \longrightarrow \mathbb{R}$$
$$e \longmapsto f(e) = \varphi(e) - \varphi(-e)$$

là một luồng trên mạng $G$ và $|f| = |\varphi|$

***Chứng minh***

Trước hết ta chứng minh $f$ thỏa mãn tất cả các ràng buộc về luồng:

Ràng buộc về sức chứa: Với $\forall e \in E$:

$$f(e) = \varphi(e) - \underbrace{\varphi(-e)}_{\geq 0} \leq \varphi(e) \leq c(u, v)$$

Ràng buộc về tính đối xứng lệch: Với $\forall e \in E$

$$\begin{aligned} f(e) &= \varphi(e) - \varphi(-e) \\ &= -\big(\varphi(-e) - \varphi(e)\big) \\ &= -f(-e) \end{aligned}$$

Ràng buộc về tính bảo tồn: $\forall v \in V$, ta có:

$$\begin{aligned} f(\{v\}, V) &= \sum_{e \in \{\{v\} \to V\}} \big(\varphi(e) - \varphi(-e)\big) \\ &= \left(\sum_{e \in \{\{v\} \to V\}} \varphi(e)\right) - \left(\sum_{e \in \{\{v\} \to V\}} \varphi(-e)\right) \\ &= \left(\sum_{e \in \{\{v\} \to V\}} \varphi(e)\right) - \left(\sum_{-e \in \{V \to \{v\}\}} \varphi(-e)\right) \\ &= \varphi(\{v\}, V) - \varphi(V, \{v\}) \end{aligned}$$

Nếu $v \neq s$ và $v \neq t$, ta có:

$$f(\{v\}, V) = \varphi(\{v\}, V) - \varphi(V, \{v\}) = 0$$

(Do tổng luồng dương đi ra khỏi $v$ bằng tổng luồng dương đi vào $v$)

Nếu $v = s$, xét giá trị luồng $f$

$$|f| = f(\{s\}, V) = \varphi((\{s\}, V) - \varphi(V, \{s\}) = |\varphi|$$

***Bổ đề 3-2***

Cho $f: E \to \mathbb{R}$ là một luồng trên mạng $G = (V, E, c, s, t)$. Khi đó hàm:

$$\varphi: E \longrightarrow [0, \infty)$$
$$e \longmapsto \varphi(e) = \max\{f(e), 0\} = \begin{cases} f(e), \text{nếu } f(e) \geq 0 \\ 0, \text{nếu } f(e) < 0 \end{cases}$$

là một luồng dương trên mạng và $|\varphi| = |f|$

***Chứng minh***

Trước hết ta chứng minh $\varphi$ thỏa mãn các ràng buộc về luồng dương:

Ràng buộc về sức chứa: $\forall e \in E$, rõ ràng $\varphi(e)$ là số không âm và $\varphi(e) = \max\{f(e), 0\} \leq c(e)$.

Ràng buộc về tính bảo tồn: $\forall v \in V$, tổng luồng dương ra khỏi $v$ trừ tổng luồng dương đi vào $v$ bằng:

$$\varphi(\{v\}, V) - \varphi\{V, \{v\}) = \sum_{\substack{e \in \{\{v\} \to V\} \\ f(e) \geq 0}} f(e) - \sum_{\substack{e \in \{V \to \{v\}\} \\ f(e) > 0}} f(e)$$
$$= \sum_{\substack{e \in \{\{v\} \to V\} \\ f(e) \geq 0}} f(e) + \sum_{\substack{-e \in \{\{v\} \to V\} \\ f(-e) < 0}} f(-e) = f(\{v\}, V)$$

Nếu $v \neq s$ và $v \neq t$, $f(\{v\}, V) = 0$ nên luồng dương đi vào $v$ ($\varphi(\{v\}, V)$) được bảo tồn khi đi ra khỏi $v$ ($\varphi(V, \{v\})$).

Nếu $v = s$, ta có:

$$|\varphi| = \varphi(\{s\}, V) - \varphi(V, \{s\}) = f(\{s\}, V) = |f|$$

Bổ đề 3-1 và Bổ đề 3-2 cho ta một mối tương quan giữa luồng và luồng dương. Khái niệm về luồng dương dễ hình dung hơn so với khái niệm luồng, tuy nhiên những định nghĩa về luồng tổng quát lại thích hợp hơn cho việc trình bày và chứng minh các thuật toán trong bài. Ta sẽ **sử dụng luồng dương trong các hình vẽ và output** (chỉ quan tâm tới các giá trị luồng dương $\varphi(e)$), còn các khái niệm về luồng sẽ được dùng để diễn giải các thuật toán trong bài.

Trong quá trình cài đặt thuật toán, các hàm $c$ và $f$ sẽ được xác định bởi tập các giá trị $\{c[e]\}_{e \in E}$ và $\{f[e]\}_{e \in E}$ nên ta có thể dùng lẫn các ký hiệu $c(e), f(e)$ (nếu muốn đề cập tới giá trị hàm) hoặc $c[e], f[e]$ (nếu muốn đề cập tới các biến số).

### e) Một số tính chất cơ bản

Cho mạng $G = (V, E, c, s, t)$ và một luồng $f$ trên $G$. Gọi $c(X, Y)$ là *lưu lượng* từ $X$ sang $Y$ và $f(X, Y)$ là giá trị luồng từ $X$ sang $Y$.

***Định lý 3-3***

Cho $f$ là một luồng trên mạng $G = (V, E, c, s, t)$, khi đó:

a) $\forall X \subseteq V$, ta có $f(X, X) = 0$.

b) $\forall X, Y \subseteq V$, ta có $f(X, Y) = -f(Y, X)$.

c) $\forall X, Y, Z \subseteq V$ và $X \cap Y = \emptyset$, ta có $f(X, Z) + f(Y, Z) = F(X \cup Y, Z)$.

d) $\forall X \subseteq V - \{s, t\}$, ta có $f(X, V) = 0$.

***Chứng minh***

a) $\forall X \subseteq V$, ta có:

$$f(X, X) = \sum_{e \in \{X \to X\}} f(e)$$

như vậy $f(e)$ xuất hiện trong tổng nếu và chỉ nếu $f(-e)$ cũng xuất hiện trong tổng. Theo tính đối xứng lệch của luồng: $f(e) = -f(-e)$, ta có $f(X, X) = 0$.

b) $\forall X, Y \subseteq V$, ta có :

$$f(X, Y) = \sum_{e \in \{X \to Y\}} f(e) = - \sum_{-e \in \{Y \to X\}} f(-e) = -f(Y, X)$$

c) $\forall X, Y, Z \subseteq V$ và $X \cap Y = \emptyset$, ta có:

$$f(X \cup Y, Z) = \sum_{e \in \{X \cup Y \to Z\}} f(e)$$

$$= \underbrace{\sum_{e \in \{X \to Z\}} f(e)}_{f(X,Z)} + \underbrace{\sum_{e \in \{Y \to Z\}} f(e)}_{f(Y,Z)}$$

d) $\forall X \subseteq V - \{s, t\}$, do

$$X = \bigcup_{u \in X} \{u\}$$

Nên theo chứng minh phần c):

$$f(X, V) = \sum_{u \in X} f(\{u\}, V)$$

Mỗi hạng tử của tổng: $f(\{u\}, V)$ chính là tổng luồng trên các cung đi ra khỏi đỉnh $u$, do tính bảo tồn luồng và $u$ không phải đỉnh phát cũng không phải đỉnh thu, hạng tử này phải bằng 0, suy ra $f(X, V) = 0$. Từ chứng minh phần b), ta còn suy ra $f(V, X) = 0$ nữa.

***Định lý 3-4***

Giá trị luồng trên mạng bằng tổng luồng trên các cung đi vào đỉnh thu

***Chứng minh***

Giả sử $f$ là một luồng trên mạng $G = (V, E, c, s, t)$, ta có:

$$\begin{aligned} |f| &= f(\{s\}, V) \\ &= f(V, V) - f(V - \{s\}, V) \\ &= -f(V - \{s\}, V) \\ &= f(V, V - \{s\}) \\ &= f(V, \{t\}) + f(V, V - \{s, t\}) \\ &= f(V, \{t\}) \end{aligned}$$

***Hệ quả***

Giá trị luồng dương trên mạng bằng tổng luồng dương đi vào đỉnh thu trừ tổng luồng dương ra khỏi đỉnh thu.

### f) Mạng thặng dư

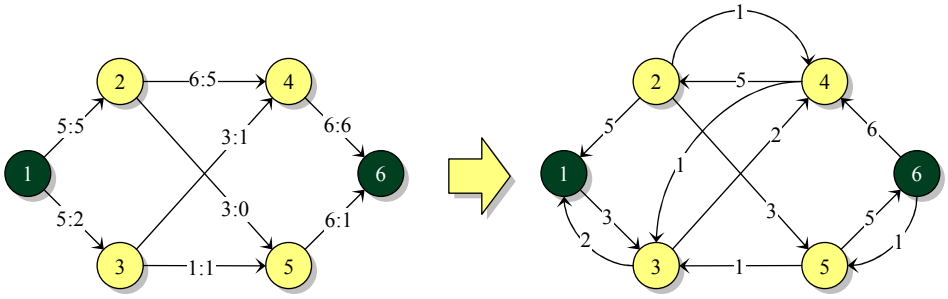
Với $f$ là một luồng trên mạng $G = (V, E, c, s, t)$. Ta xét mạng $G_f$ cũng là mạng $G$ nhưng với hàm sức chứa mới cho bởi:

$$\begin{aligned} c_f &: E \longrightarrow [0, +\infty) \\ e &\longmapsto c_f(e) = c(e) - f(e) \end{aligned} \tag{3.3}$$

Mạng $G_f$ xây dựng như vậy được gọi là *mạng thặng dư* (*residual network*) của mạng $G$ sinh ra bởi luồng $f$. Sức chứa của cung $(e)$ trên $G_f$ thực chất là lượng luồng tối đa chúng ta có thể đẩy thêm vào luồng $f(e)$ mà không làm vượt quá sức chứa $c(e)$.

Một cung trên $G$ gọi là *cung bão hòa (saturated edge)* nếu luồng trên cung đó đúng bằng sức chứa, ngược lại cung đó gọi là *cung thặng dư (residual edge)*. Ký hiệu $E_f$ là tập các cung thặng dư trên mạng thặng dư $G_f$. Một đường đi chỉ qua các cung thặng dư trên $G_f$ gọi là *đường thặng dư (residual path)*.

Cung bão hòa của mạng $G$ trên mạng thặng dư sẽ có sức chứa 0, cung này ít có ý nghĩa trong thuật toán nên chúng ta sẽ chỉ vẽ các cung thặng dư ($\in E_f$) trong các hình vẽ.




***Hình 2.7. Một luồng trên mạng (số ghi trên các cung là: sức chứa:luồng dương) và mạng thặng dư tương ứng.***

Hình 2.7 là một ví dụ về mạng thặng dư. Như đã quy ước, chúng ta chỉ vẽ các luồng dương. Đồ thị có cung $(2,4)$ với sức chứa $c(2,4) = 6$, tức là phải có cung đối $(4,2)$ với $c(4,2) = 0$. Luồng dương trên cung $(2,4)$ là $\varphi(2,4) = f(2,4) = 5$, điều này cũng cho biết luồng trên cung $(4,2)$ là $f(4,2) = -5$ theo tính đối xứng lệch. Vậy trên mạng thặng dư, ta có cung $(2,4)$ với sức chứa $c(2,4) - f(2,4) = 6 - 5 = 1$ đồng thời có cung $(4,2)$ với sức chứa $c(4,2) - f(4,2) = 0 - (-5) = 5$.

***Định lý 3-5***

Cho $f$ là một luồng trên mạng $G = (V, E, c, s, t)$. Khi đó nếu $f'$ là một luồng trên $G_f$ thì hàm:

$$f + f' : E \longrightarrow \mathbb{R}$$
$$e \longmapsto (f + f')(e) = f(e) + f'(e)$$

là một luồng trên mạng $G$ với giá trị luồng $|f + f'| = |f| + |f'|$.

***Chứng minh***

Ta chứng minh $(f + f')$ thỏa mãn ba tính chất của luồng:

Ràng buộc về sức chứa: Với $\forall e \in E$:

$$\begin{aligned}(f + f')(e) &= f(e) + f'(e) \\ &\leq f(e) + \big(c(e) - f(e)\big) \\ &= c(e)\end{aligned}$$

Tính đối xứng lệch: Với $\forall e \in E$:

$$\begin{aligned}(f + f')(e) &= f(e) + f'(e) \\ &= -f(-e) - f'(-e)\end{aligned}$$

$$= -\big(f(-e) + f'(-e)\big)$$
$$= -(f + f')(-e)$$

Tính bảo tồn: Với $\forall u \in V$, tổng luồng $f + f'$ đi ra khỏi $u$ bằng:

$$(f + f')(\{u\}, V) = \sum_{e \in \{\{u\} \to V\}} \big(f(e) + f'(e)\big)$$
$$= \sum_{e \in \{\{u\} \to V\}} f(e) + \sum_{e \in \{\{u\} \to V\}} f'(e)$$
$$= f(\{u\}, V) + f'(\{u\}, V)$$

Nếu $u \neq s$ và $u \neq t$, ta có $(f + f')(\{u\}, V) = f(\{u\}, V) + f'(\{u\}, V) = 0$.

Thay $u = s$, ta có

$$|f + f'| = (f + f')(\{s\}, V) = f(\{s\}, V) + f'(\{s\}, V) = |f| + |f'|$$

***Định lý 3-6***

Cho $f$ và $f'$ là hai luồng trên mạng $G = (V, E, c, s, t)$ khi đó hàm:

$$f' - f : E \longrightarrow \mathbb{R}$$
$$e \longmapsto (f' - f)(e) = f'(e) - f(e)$$

là một luồng trên mạng thặng dư $G_f$ với giá trị luồng $|f' - f| = |f'| - |f|$.

***Chứng minh***

Ta chứng minh rằng $f' - f$ thỏa mãn ba tính chất của luồng

Ràng buộc về sức chứa: Với $\forall e \in E$:

$$(f' - f)(e) = f'(e) - f(e)$$
$$\leq c(e) - f(e)$$
$$= c_f(e)$$

Tính đối xứng lệch: Với $\forall e \in E$:

$$(f' - f)(e) = f'(e) - f(e)$$
$$= -\big(f'(-e) - f(-e)\big)$$
$$= -(f' - f)(-e)$$

Tính bảo tồn: Với $\forall v \in V$

$$(f' - f)(\{v\}, V) = \sum_{e \in \{\{v\} \to V\}} \big(f'(e) - f(e)\big)$$
$$= \sum_{e \in \{\{v\} \to V\}} f'(e) - \sum_{e \in \{\{v\} \to V\}} f(e)$$
$$= f'(\{v\}, V) - f(\{v\}, V)$$

Nếu $u \neq s$ và $u \neq t$, ta có $(f' - f)(\{v\}, V) = f'(\{v\}, V) - f(\{v\}, V) = 0$.

Thay $u = s$, ta có

$$|f' - f| = (f' - f)(\{s\}, V) = f'(\{s\}, V) - f(\{s\}, V) = |f'| - |f|$$

## *3.2. Thuật toán Ford-Fulkerson*

### a) Đường tăng luồng

Với $f$ là một luồng trên mạng $G = (V, E, c, s, t)$. Gọi $P$ là một đường đi đơn từ $s$ tới $t$ trên mạng thặng dư $G_f$. *Giá trị thặng dư* (*residual capacity*) của đường $P$, ký hiệu $\Delta_P$, được định nghĩa bằng sức chứa nhỏ nhất của các cung dọc trên đường $P$ (xét trên $G_f$):

$$\Delta_P = \min\{c_f(e) : (e) \text{ nằm trên } P\}$$

Vì các sức chứa $c_f(e)$ là số không âm nên $\Delta_P$ luôn là số không âm. Nếu $\Delta_P > 0$ tức là đường đi $P$ là một đường thặng dư, khi đó đường đi $P$ gọi là một *đường tăng luồng (augmenting path)* tương ứng với luồng $f$.

***Định lý 3-7***

Cho $f$ là một luồng trên mạng $G = (V, E, c, s, t)$, $P$ là một đường tăng luồng trên $G_f$. Khi đó hàm $f_P : E \to \mathbb{R}$ định nghĩa như sau:

$$f_P(e) = \begin{cases} +\Delta_P, \text{ nếu } e \in P \\ -\Delta_p, \text{ nếu } -e \in P \\ 0, \text{ trường hợp khác} \end{cases} \qquad (3.4)$$

là một luồng trên $G_f$ với giá trị luồng $|f_P| = \Delta_P > 0$.

***Chứng minh***

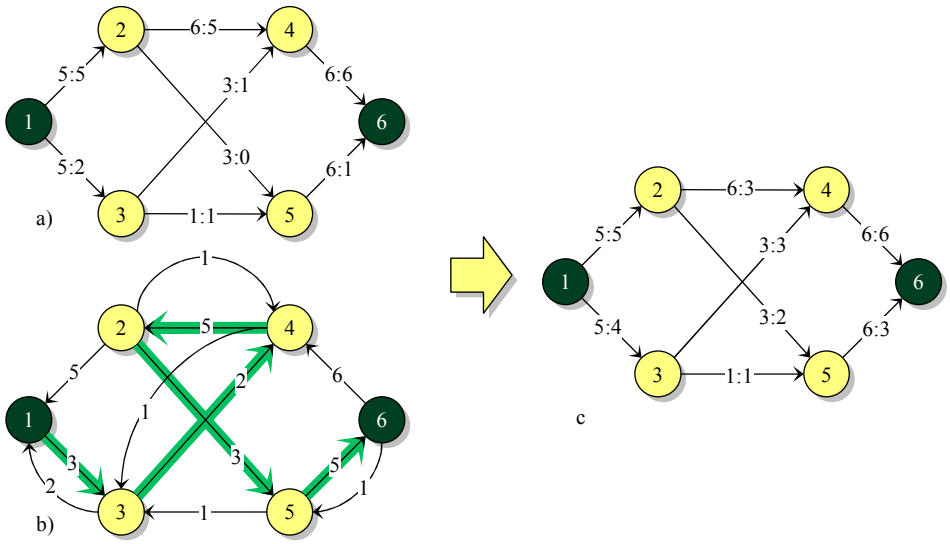
Chúng ta sẽ không chứng minh cụ thể vì việc kiểm chứng $f_P$ thỏa mãn ba tính chất của luồng khá dễ dàng. Bản chất của luồng $f_P$ là đẩy một giá trị luồng $\Delta_P$ từ $s$ tới $t$ dọc theo các cung trên đường $P$, đồng thời kéo một giá trị luồng $-\Delta_P$ từ $t$ về $s$ theo hướng ngược lại*.

* Có thể hình dung cơ chế này như một quá trình điện phân: Bao nhiêu ion dương (cation) chuyển đến cực âm (catot) $t$ thì cũng phải có bấy nhiêu ion âm (anion) chuyển đến cực dương (anot) $s$.

Định lý 3-5 và Định lý 3-7 cho ta một hệ quả sau:

***Hệ quả 3-8***

Cho $f$ là một luồng trên mạng $G = (V, E, c, s, t)$ và $P$ là một đường tăng luồng trên $G_f$, gọi $f_P$ là luồng trên $G_f$ định nghĩa như trong công thức (3.4). Khi đó $f + f_P$ là một luồng mới trên $G$ với giá trị $|f + f_P| = |f| + |f_P| = |f| + \Delta_P$.




***Hình 2.8. Tăng luồng dọc đường tăng luồng.***

Hình 2.8 là ví dụ về cơ chế tăng luồng trên mạng với đỉnh phát 1, đỉnh thu 6 và luồng $f$ giá trị 7 (hình a) (chú ý rằng ta chỉ vẽ các luồng dương cho đỡ rối). Với mạng thặng dư $G_f$ (hình b), giả sử ta chọn đường đi $P = \langle 1,3,4,2,5,6 \rangle$ làm đường tăng luồng, giá trị thặng dư của $P$ bằng $\Delta_P = 2$ (sức chứa của cung $(3,4)$). Luồng $f_P$ trên $G_f$ sẽ có các giá trị sau:

$$f_P(1,3) = f_P(3,4) = f_P(4,2) = f_P(2,5) = f_P(5,6) = 2$$
$$f_P(3,1) = f_P(4,3) = f_P(2,4) = f_P(5,2) = f_P(6,5) = -2$$

Cộng các giá trị này vào luồng $f$ đang có, ta sẽ được một luồng mới trên $G$ với giá trị 9 (hình c).

Cơ chế cộng luồng $f_P$ vào luồng $f$ hiện có gọi là *tăng luồng dọc theo đường tăng luồng* $P$.

### b) Thuật toán Ford-Fulkerson

Thuật toán Ford-Fulkerson [**14**] để tìm luồng cực đại trên mạng dựa trên cơ chế tăng luồng dọc theo đường tăng luồng. Bắt đầu từ một luồng $f$ bất kỳ trên mạng (chẳng hạn luồng trên mọi cung đều bằng 0), thuật toán tìm đường tăng luồng $P$ trên mạng thặng dư, gán $f := f + f_P$ để tăng giá trị luồng $f$ và lặp lại cho tới khi không tìm được đường tăng luồng nữa.

```
f := «Một luồng bất kỳ»;
while «Tìm được đường tăng luồng P» do
   f := f + fP;
Output ← f;
```

### c) Cài đặt

Chúng ta sẽ cái đặt thuật toán Ford-Fulkerson để tìm luồng cực đại trên mạng với khuôn dạng Input/Output như sau:

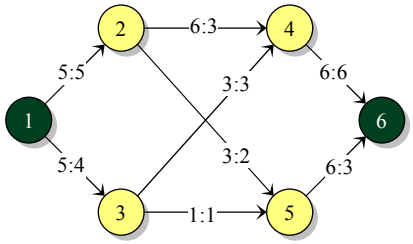
**Input**

- Dòng 1 chứa số đỉnh $n \leq 10^3$, số cung $m \leq 10^5$ của mạng, đỉnh phát $s$, đỉnh thu $t$.
- $m$ dòng tiếp theo, mỗi dòng chứa ba số nguyên dương $u, v, c$ tương ứng với một cung nối từ $u$ tới $v$ với sức chứa $c \leq 10^4$.

**Output**

Luồng cực đại trên mạng (như đã quy ước, chỉ đưa ra các luồng dương trên các cung).

| Sample Input | Sample Output |
|---|---|
| 6 8 1 6<br>5 6 6<br>4 6 6<br>3 5 1<br>3 4 3<br>2 5 3<br>2 4 6<br>1 3 5<br>1 2 5 | Maximum flow:<br>e[1] = (5, 6): c = 6, f = 3<br>e[2] = (4, 6): c = 6, f = 6<br>e[3] = (3, 5): c = 1, f = 1<br>e[4] = (3, 4): c = 3, f = 3<br>e[5] = (2, 5): c = 3, f = 2<br>e[6] = (2, 4): c = 6, f = 3<br>e[7] = (1, 3): c = 5, f = 4<br>e[8] = (1, 2): c = 5, f = 5<br>Value of flow: 9 |




Để cài đặt thuật toán được hiệu quả cần có một cơ chế tổ chức dữ liệu hợp lý. Chúng ta cần lưu trữ luồng $f$ trên các cung, tìm đường tăng luồng $P$ trên $G_f$ và cộng luồng $f_P$ vào luồng $f$ hiện có. Việc tìm đường tăng luồng $P$ trên $G_f$ sẽ được thực hiện bằng một thuật toán tìm kiếm trên đồ thị còn việc tăng luồng dọc trên đường $P$ đòi hỏi phải tăng giá trị luồng trên các cung dọc trên đường đi đồng thời giảm giá trị luồng trên các cung đối. Vậy cấu trúc dữ liệu cần tổ chức để tạo điều kiện thuận lợi cho thuật toán tìm đường tăng luồng cũng như dễ dàng chỉ ra cung đối của một cung cho trước.

Đồ thị được biểu diễn bởi danh sách liên thuộc. Tất cả $m$ cung của mạng được chứa trong danh sách $e[1 \ldots m]$. Ngoài ra ta thêm $m$ cung đối của chúng với sức chứa 0. Các cung đối này được lưu trữ trong danh sách $e[-m \ldots -1]$, cung đối của cung $e[i]$ là cung $e[-i]$, cung $e[0]$ được sử dụng với vai trò phần tử cầm canh và không được tính đến.

Mỗi phần tử của danh sách $e$ là một bản ghi gồm 4 trường $(x, y, c, f)$ trong đó $x$, $y$ là đỉnh đầu và đỉnh cuối của cung, $c$ là sức chứa và $f$ là luồng trên cung. Danh

sách liên thuộc được xây dựng bởi hai mảng $head[1 \ldots n]$ và $link[-m \ldots m]$, trong đó:

- $head[u]$ là chỉ số cung đầu tiên trong danh sách liên thuộc các cung đi ra khỏi $u$, trường hợp $u$ không có cung đi ra, $head[u]$ được gán bằng 0.
- $link[i]$ là chỉ số cung kế tiếp cung $e[i]$ trong cùng danh sách liên thuộc các cung đi ra khỏi một đỉnh. Trường hợp $e[i]$ là cung cuối cùng của một danh sách liên thuộc, $link[i]$ được gán bằng 0

Việc duyệt các cung đi ra khỏi đỉnh $u$ sẽ được thực hiện theo cách sau:

```
i := head[u]; //i là chỉ số cung đầu tiên trong danh sách liên thuộc các cung ra khỏi u
while i ≠ 0 do //Chừng nào chưa duyệt qua cung cuối danh sách liên thuộc
  begin
    «Xử lý cung e[i]»;
    i := link[i]; //Nhảy sang xét cung kế tiếp trong danh sách liên thuộc
  end;
```

Tại mỗi bước, ta dùng thuật toán BFS để tìm đường đi từ $s$ tới $t$ trên $G_f$, mỗi đỉnh $v$ trên đường đi được lưu vết $trace[v]$ là chỉ số cung đi vào $v$ trên đường đi $P$ tìm được. Dựa vào vết này, ta sẽ liệt kê được tất cả các cung trên đường đi, tăng luồng trên các cung này lên $\Delta_P$ đồng thời giảm luồng trên các cung đối đi $\Delta_P$.

Edmonds và Karp [**12**] đã đề xuất mô hình cài đặt thuật toán Ford-Fulkerson trong đó thuật toán BFS được sử dụng để tìm đường tăng luồng nên người ta còn gọi thuật toán Ford-Fulkerson với kỹ thuật sử dụng BFS tìm đường tăng luồng là thuật toán Edmonds-Karp.

💻 EDMONDSKARP.PAS ✓ Thuật toán Edmonds-Karp

```
{$MODE OBJFPC}
program MaximumFlow;
const
  maxN = 1000;
  maxM = 100000;
  maxC = 10000;
type
  TEdge = record //Cấu trúc một cung
    x, y: Integer; //Hai đỉnh đầu mút
```

```
    c, f: Integer; //Sức chứa và luồng
  end;
  TQueue = record //Hàng đợi dùng cho BFS
    items: array[1..maxN] of Integer;
    front, rear: Integer;
  end;
var
  e: array[-maxM..maxM] of TEdge; //Danh sách các cung
  link: array[-maxM..maxM] of Integer;
  //Móc nối trong danh sách liên thuộc
  head: array[1..maxN] of Integer;
  //head[u]: Chỉ số cung đầu tiên trong danh sách liên thuộc các cung ra khỏi u
  trace: array[1..maxN] of Integer; //Vết đường đi
  n, m, s, t: Integer;
  FlowValue: Integer;
  Queue: TQueue;
procedure Enter; //Nhập dữ liệu
var i, u, v, capacity: Integer;
begin
  ReadLn(n, m, s, t);
  FillChar(head[1], n * SizeOf(head[1]), 0);
  for i := 1 to m do
    begin
      ReadLn(u, v, capacity);
      with e[i] do //Thêm cung e[i] = (u, v) vào danh sách liên thuộc của u
        begin
          x := u;
          y := v;
          c := capacity;
          link[i] := head[u];
          head[u] := i;
        end;
      with e[-i] do //Thêm cung e[-i] = (v, u) vào danh sách liên thuộc của v
        begin
          x := v;
          y := u;
          c := 0;
          link[-i] := head[v];
```

```
          head[v] := -i;
        end;
    end;
end;
procedure InitZeroFlow; //Khởi tạo luồng 0
var i: Integer;
begin
  for i := -m to m do e[i].f := 0;
  FlowValue := 0;
end;
function FindPath: Boolean; //Tìm đường tăng luồng bằng BFS
var u, v, i: Integer;
begin
  FillChar(trace[1], n * SizeOf(trace[1]), 0);
  trace[s] := 1; //trace[s] ≠ 0: đỉnh đã thăm, có thể dùng bất cứ hằng số nào khác 0
  with Queue do
    begin
      items[1] := s;
      front := 1;
      rear := 1; //Hàng đợi chỉ gồm đỉnh s
      repeat
        u := items[front];
        Inc(front); //Lấy một đỉnh u khỏi hàng đợi
        i := head[u];
        while i <> 0 do //Duyệt danh sách liên thuộc của u
          begin
            v := e[i].y; //nút e[i] chứe một cung đi từ u tới v
            if (trace[v] = 0)
                and (e[i].f < e[i].c) then
            //v chưa thăm và e[i] là cung thặng dư
              begin
                trace[v] := i; //Lưu vết
                if v = t then Exit(True);
                //Tìm thấy đường tăng luồng, thoát
                Inc(rear);
                items[rear] := v; //Đẩy v vào hàng đợi
              end;
            i := link[i]; //Nhảy sang nút kế tiếp trong danh sách liên thuộc
```

```
            end;
        until front > rear;
        Result := False; //Không tìm thấy đường tăng luồng
      end;
end;
procedure AugmentFlow; //Tăng luồng dọc đường một tăng luồng
var Delta, v, i: Integer;
begin
  //Trước hết xác định Delta bằng sức chứa nhỏ nhất của các cung trên đường tăng luồng
  v := t; //Bắt đầu từ t
  Delta := maxC;
  repeat
    i := trace[v];
    // e[i] là một cung trên đường tăng luồng với sức chứa e[i].c - e[i].f
    if e[i].c - e[i].f < Delta then
      Delta := e[i].c - e[i].f;
    v := e[i].x; //Đi dần về s
  until v = s;
  //Tăng luồng thêm Delta
  v := t; //Bắt đầu từ t
  repeat
    i := trace[v]; // e[i] là một cung trên đường tăng luồng
    Inc(e[i].f, Delta); //Tăng luồng trên e[i] lên Delta
    Dec(e[-i].f, Delta); //Giảm luồng trên cung đối tương ứng đi Delta
    v := e[i].x; //Đi dần về s
  until v = s;
  Inc(FlowValue, Delta); //Giá trị luồng f được tăng lên Delta
end;
procedure PrintResult; //In kết quả
var i: Integer;
begin
  WriteLn('Maximum flow: ');
  for i := 1 to m do
    with e[i] do
      if f > 0 then //Chỉ cần in ra các cung có luồng > 0
        WriteLn('e[', i, '] = (', x, ', ', y, '): c
                              = ', c, ', f = ', f);
  WriteLn('Value of flow: ', FlowValue);
end;
```

```
begin
  Enter; //Nhập dữ liệu
  InitZeroFlow; //Khởi tạo luồng 0
  while FindPath do //Thuật toán Ford-Fulkerson
    AugmentFlow;
  PrintResult; //In kết quả
end.
```

### d) Tính đúng của thuật toán

Trước hết dễ thấy rằng thuật toán Ford-Fulkerson trả về một luồng, tức là kết quả mà thuật toán trả về thỏa mãn các tính chất của luồng. Việc chứng minh luồng đó là cực đại đã xây dựng một định lý quan trọng về mối quan hệ giữa luồng cực đại và lát cắt hẹp nhất.

Ta gọi một lát cắt $(X, Y)$ là một cách phân hoạch tập đỉnh $V$ làm hai tập rời nhau: $X \cap Y = \emptyset$ và $X \cup Y = V$. Lát cắt có $s \in X$ và $t \in Y$ được gọi là một lát cắt $s - t$.

Lưu lượng từ $X$ sang $Y$ $(c(X, Y))$ và luồng từ $X$ sang $Y$ $(f(X, Y))$ được gọi là lưu lượng và luồng thông qua lát cắt.

***Bổ đề 3-9***

Với $f$ là một luồng trên mạng $G = (V, E, c, s, t)$. Khi đó luồng thông qua một lát cắt $s - t$ bất kỳ bằng $|f|$.

***Chứng minh***

Với $V = X \cup Y$ là một lát cắt $s - t$ bất kỳ, theo Định lý 3-3

$$f(X, Y) = f(X, V) - f(X, V - Y) = f(X, V) - f(X, X) = f(X, V)$$

Cũng theo định lý này ta có:

$$f(X, V) = f(s, V) + \underbrace{f(X - \{s\}, V)}_{0} = f(s, V) = |f|$$

***Bổ đề 3-10***

Với $f$ là một luồng trên mạng $G = (V, E, c, s, t)$. Khi đó luồng thông qua một lát cắt $s - t$ bất kỳ không vượt quá lưu lượng của lát cắt đó.

***Chứng minh***

Với $V = X \cup Y$ là một lát cắt $s - t$ bất kỳ ta có

$$f(X,Y) = \sum_{e\in\{X\to Y\}} f(e) \le \sum_{e\in\{X\to Y\}} c(e) = c(X,Y)$$

***Định lý 3-11 (mối quan hệ giữa luồng cực đại, đường tăng luồng và lát cắt hẹp nhất)***

Nếu $f$ là một luồng trên mạng $G = (V, E, c, s, t)$, khi đó ba mệnh đề sau là tương đương:

a) $f$ là luồng cực đại trên mạng $G$.

b) Mạng thặng dư $G_f$ không có đường tăng luồng.

c) Tồn tại $V = X \cup Y$ là một lát cắt $s - t$ để $f(X,Y) = c(X,Y)$

***Chứng minh***

"a⇒b"

Giả sử phản chứng rằng mạng thặng dư $G_f$ có đường tăng luồng $P$ thì $f + f_P$ cũng là một luồng trên $G$ với giá trị luồng lớn hơn $f$, trái giả thiết $f$ là luồng cực đại trên mạng.

"b⇒c"

Nếu $G_f$ không tồn tại đường tăng luồng thì ta đặt $X$ là tập các đỉnh đến được từ $s$ bằng một đường thặng dư và $Y$ là tập các đỉnh còn lại:

$$X = \{v: \exists \text{ đường thặng dư } s \rightsquigarrow v\}; Y = V - X$$

Rõ ràng $X \cap Y = \emptyset$, $X \cup Y = V$ và $s \in X$, $t \in Y$ ($t$ không thể đến được từ $s$ bởi một đường thặng dư bởi nếu không thì đường đi đó sẽ là một đường tăng luồng).

Các cung $e \in \{X \to Y\}$ chắc chắn phải là cung bão hòa, bởi nếu có cung thặng dư $e = (u, v) \in \{X \to Y\}$ thì từ $s$ sẽ tới được $v$ bằng một đường thặng dư. Tức là $v \in X$, trái với cách xây dựng lát cắt. Từ $f(e) = c(e)$ với $\forall e \in \{X \to Y\}$, ta có

$$f(X,Y) = \sum_{e\in\{X\to Y\}} f(e) = \sum_{e\in\{X\to Y\}} c(e) = c(X,Y)$$

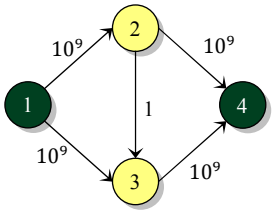
"c⇒a"

Bổ đề 16.9 và Định lí 16.17 cho thấy giá trị của một luồng trên mạng không thể vượt quá lưu lượng của một lát cắt $s - t$ bất kỳ. Nếu tồn tại một lát cắt $s - t$ mà luồng thông qua lát cắt đúng bằng lưu lượng thì luồng đó chắc chắn phải là luồng cực đại.

Lát cắt $s - t$ có lưu lượng nhỏ nhất (bằng giá trị luồng cực đại trên mạng) gọi là *Lát cắt hẹp nhất* của mạng $G$.

### e) Tính dừng của thuật toán

Thuật toán Ford-Fulkerson có thời gian thực hiện phụ thuộc vào thuật toán tìm đường tăng luồng tại mỗi bước. Có thể chỉ ra được ví dụ mà nếu dùng DFS để tìm đường tăng luồng thì thời gian thực hiện giải thuật không bị chặn bởi một hàm đa thức của số đỉnh và số cạnh. Thêm nữa, nếu sức chứa của các cung là số thực, người ta còn chỉ ra được ví dụ mà với thuật toán tìm đường tăng luồng không tốt, giá trị luồng sau mỗi bước vẫn tăng nhưng **không bao giờ đạt luồng cực đại**. Tức là nếu có thể cài đặt chương trình tính toán số thực với độ chính xác tuyệt đối, thuật toán sẽ chạy mãi không dừng.




***Hình 2.8. Mạng với 4 đỉnh (1 phát, 4 thu), thuật toán Ford-Fulkerson có thể mất 2 tỉ lần tìm đường tăng luồng nếu luân phiên chọn hai đường $\langle 1,2,3,4\rangle$ và $\langle 1,3,2,4\rangle$ làm đường tăng luồng, mỗi lần tăng giá trị luồng lên 1 đơn vị.***

Chính vì vậy nên trong một số tài liệu người ta gọi là "phương pháp Ford-Fulkerson" để chỉ một cách tiếp cận chung, còn từ "thuật toán" được dùng để chỉ một cách cài đặt phương pháp Ford-Fulkerson trên một cấu trúc dữ liệu cụ thể, với một thuật toán tìm đường tăng luồng cụ thể. Ví dụ phương pháp Ford-Fulkerson cài đặt với thuật toán tìm đường tăng luồng bằng BFS như trên được gọi là thuật toán Edmonds-Karp. Tính dừng của thuật toán Edmonds-Karp sẽ được chỉ ra khi chúng ta đánh giá thời gian thực hiện giải thuật.

Xét $G_f$ là mạng thặng dư của một mạng $G$ ứng với luồng $f$ nào đó, ta gán trọng số 1 cho các cung thặng dư của $G_f$ và gán trọng số $+\infty$ cho các cung bão hòa của $G_f$. Dễ thấy rằng thuật toán tìm đường tăng luồng bằng BFS sẽ trả về một đường đi ngắn nhất từ $s$ tới $t$ tương ứng với hàm trọng số đã cho. Ký hiệu $\delta_f(u,v)$ là độ dài đường đi ngắn nhất từ $u$ tới $v$ (khoảng cách từ $u$ tới $v$) trên mạng thặng dư.

***Bổ đề 3-12***

Nếu ta khởi tạo luồng 0 và thực hiện thuật toán Edmonds-Karp trên mạng $G = (V, E)$ có đỉnh phát $s$ và đỉnh thu $t$. Khi đó với mọi đỉnh $v \in V$, khoảng cách từ $s$ tới $v$ trên mạng thặng dư không giảm sau mỗi bước tăng luồng.

***Chứng minh***

Khi $v = s$, rõ ràng khoảng cách từ $s$ tới chính nó luôn bằng 0 từ khi bắt đầu tới khi kết thúc thuật toán. Ta chỉ cần chứng minh bổ đề đúng với những đỉnh $v \neq s$.

Giả sử phản chứng rằng tồn tại một đỉnh $v \in V - \{s\}$ mà khi thuật toán Edmonds-Karp tăng luồng $f$ lên thành $f'$ sẽ làm cho $\delta_{f'}(s, v)$ nhỏ hơn $\delta_f(s, v)$. Nếu có nhiều đỉnh $v$ như vậy ta chọn đỉnh $v$ có $\delta_{f'}(s, v)$ nhỏ nhất. Gọi $P = s \rightsquigarrow u \rightarrow v$ là đường đi ngắn nhất từ $s$ tới $v$ trên $G_{f'}$, ta có $(u, v)$ là cung thặng dư trên $G_{f'}$ và

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$$

Bởi cách chọn đỉnh $v$, độ dài đường đi ngắn nhất từ $s$ tới $u$ không thể bị giảm đi sau phép tăng luồng, tức là

$$\delta_{f'}(s, u) \geq \delta_f(s, u)$$

Ta chứng minh rằng $(u, v)$ phải là cung bão hòa trên $G_f$. Thật vậy, nếu $(u, v)$ là cung thặng dư (có trọng số 1) trên $G_f$ thì:

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 \text{ (bất đẳng thức tam giác)} \\ &\leq \delta_{f'}(s, u) + 1 \text{ (khoảng cách từ } s \text{ tới } u \text{ không giảm)} \\ &= \delta_{f'}(s, v) \end{aligned}$$

Trái với giả thiết rằng khoảng cách từ $s$ tới $v$ phải giảm đi sau phép tăng luồng.

Làm thế nào để $(u, v)$ là cung bão hòa trên $G_f$ nhưng lại là cung thặng dư trên $G_{f'}$? Câu trả lời duy nhất là do phép tăng luồng từ $f$ lên $f'$ làm giảm luồng trên cung $(u, v)$, tức là cung đối $(v, u)$ phải là một cung trên đường tăng luồng tìm được. Vì đường tăng luồng tại mỗi bước luôn là đường đi ngắn nhất nên $(v, u)$ phải là cung cuối cùng trên đường đi ngắn nhất từ $s$ tới $u$ của $G_f$. Từ đó suy ra:

$$\begin{aligned} \delta_f(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 \text{ (khoảng cách từ } s \text{ tới } u \text{ không giảm)} \\ &= \delta_{f'}(s, v) - 2 \text{ (theo cách chọn } u \text{ và } v) \end{aligned}$$

Mâu thuẫn với giả thuyết khoảng cách từ $s$ tới $v$ phải giảm đi sau khi tăng luồng. Ta có điều phải chứng minh: Với $\forall v \in V$, khoảng cách từ $s$ tới $v$ trên mạng thặng dư không giảm sau mỗi bước tăng luồng.

***Bổ đề 3-13***

Nếu thuật toán Edmonds-Karp thực hiện trên mạng $G = (V, E, c, s, t)$ với luồng khởi tạo là luồng 0 thì số lượt tăng luồng được sử dụng trong thuật toán là $O(|V||E|)$.

***Chứng minh***

Ta chia quá trình thực hiện thuật toán Edmonds-Karp thành các pha. Mỗi pha tìm một đường tăng luồng $P$ và tăng luồng thêm một giá trị thặng dư $\Delta_P$. Giá trị thặng dư này theo định nghĩa sẽ phải bằng sức chứa của một cung thặng dư $e$ nào đó trên đường $P$:

$$\exists e \in P: \Delta_P = c(e) - f(e)$$

Khi tăng luồng dọc trên được $P$ thì cung $e$ sẽ trở thành bão hòa. Những cung thặng dư trở nên bão hòa sau khi tăng luồng gọi là *cung tới hạn* (*critical edge*) tại mỗi pha. Mỗi pha có ít nhất một cung tới hạn.

Ta đánh giá xem mỗi cung của mạng có thể trở thành cung tới hạn bao nhiêu lần. Với một cung $e = (u, v)$, ta xét pha $A$ đầu tiên làm $e$ trở thành cung tới hạn và $f_A$ là luồng khi bắt đầu pha $A$. Do $e$ nằm trên đường tăng luồng ngắn nhất trên $G_{f_A}$ nên khi pha này bắt đầu:

$$\delta_{f_A}(s, u) + 1 = \delta_{f_A}(s, v)$$

Pha $A$ sau khi tăng luồng sẽ làm cung $e$ sẽ trở nên bão hòa.

Để $e$ có thể trở thành cung tới hạn một lần nữa thì tiếp theo pha $A$ phải có một pha $B$ giảm luồng trên cung $e$ để biến $e$ thành cung thặng dư, tức là cung $-e = (v, u)$ phải là một cung trên đường tăng luồng của pha $B$. Gọi $f_B$ là luồng khi pha $B$ bắt đầu, cũng vì tính chất của đường đi ngắn nhất, ta có

$$\delta_{f_B}(s, v) + 1 = \delta_{f_B}(s, u)$$

Bổ đề 16.12 đã chứng minh rằng khoảng cách từ $s$ tới $v$ trên mạng thặng dư không giảm đi sau mỗi pha, nên $\delta_{f_B}(s, v) \geq \delta_{f_A}(s, v)$. Suy ra:

$$\begin{aligned} \delta_{f_B}(s, u) &= \delta_{f_B}(s, v) + 1 \\ &\geq \delta_{f_A}(s, v) + 1 \\ &= \delta_{f_A}(s, u) + 2 \end{aligned}$$

Như vậy nếu một cung $(u, v)$ là cung tới hạn trong $k$ pha thì khi pha thứ $k$ bắt đầu, khoảng cách từ $s$ tới $u$ trên mạng thặng dư đã tăng lên ít nhất $2(k-1)$ đơn vị so với thời điểm trước pha thứ nhất. Khoảng cách $\delta_f(s, u)$ ban đầu là số không âm và chừng nào còn đường thặng dư đi từ $s$ tới $u$, khoảng cách $\delta_f(s, u)$ không thể vượt quá $|V| - 1$. Điều đó cho thấy $k \leq \frac{|V|+1}{2} = O(|V|)$.

Tổng hợp lại, ta có:

- Mạng có tất cả $|E|$ cung.
- Mỗi pha có ít nhất một cung tới hạn
- Một cung có thể trở thành tới hạn trong $O(|V|)$ pha

Vậy tổng số pha được thực hiện trong thuật toán Edmonds-Karp là một đại lượng $O(|V||E|)$

***Định lý 3-14***

Có thể cài đặt thuật toán Edmonds-Karp để tìm luồng cực đại trên mạng $G = (V, E, c, s, t)$ trong thời gian $O(|V||E|^2)$.

***Chứng minh***

Bổ đề 3-15 đã chứng minh rằng thuật toán Edmonds-Karp cần thực hiện $O(|V||E|)$ lượt tăng luồng. Tại mỗi lượt thuật toán tìm đường tăng luồng bằng BFS và tăng luồng dọc đường này có thời gian thực hiện $O(|E|)$. Suy ra thời gian thực hiện giải thuật Edmonds-Karp là $O(|V||E|^2)$.

Nếu khả năng thông qua trên các cung của mạng là số nguyên thì còn có một cách đánh giá khác dựa trên giá trị luồng cực đại, nếu ta khởi tạo luồng 0 thì sau mỗi lượt tăng luồng, giá trị luồng được tăng lên ít nhất 1 đơn vị. Suy ra thời gian thực hiện giải thuật khi đó là $O(|f||E|)$ với $|f|$ là giá trị luồng cực đại.

## *3.3. Thuật toán đẩy tiền luồng*

Thuật toán Ford-Fulkerson không những là một cách tiếp cận thông minh mà việc chứng minh tính đúng đắn của nó cho ta nhiều kết quả thú vị về mối liên hệ giữa luồng cực đại và lát cắt hẹp nhất. Tuy vậy với những đồ thị kích thước rất lớn thì tốc độ của chương trình tương đối chậm.

Trong phần này ta sẽ trình bày một lớp các thuật toán nhanh nhất cho tới nay để giải bài toán luồng cực đại, tên chung của các thuật toán này là thuật toán *đẩy tiền luồng* (*preflow-push*).

Hãy hình dung mạng như một hệ thống đường ống dẫn nước từ với điểm phát $s$ tới điểm thu $t$, các cung là các đường ống, sức chứa là lưu lượng đường ống có thể tải. Nước chảy theo nguyên tắc từ chỗ cao về chỗ thấp. Với một lượng nước lớn phát ra từ $s$ tới một đỉnh $v$, nếu có cách chuyển lượng nước đó sang địa điểm khác thì không có vấn đề gì, nếu không thì có hiện tượng "tràn" xảy ra tại $v$, ta "dâng cao" điểm $v$ để lượng nước đó đổ sang điểm khác (có thể đổ ngược về $s$).

Cứ tiếp tục quá trình như vậy cho tới khi không còn hiện tượng tràn ở bất cứ điểm nào. Cách tiếp cận này hoàn toàn khác với thuật toán Ford-Fulkerson: thuật toán Ford-Fulkerson cố gắng tìm một dòng chảy phụ từ $s$ tới $t$ và thêm dòng chảy này vào luồng hiện có đến khi không còn dòng chảy phụ nữa.

### a) Tiền luồng

Cho một mạng $G = (V, E, c, s, t)$. Một *tiền luồng (preflow)* trên $G$ là một hàm:

$$f: E \longrightarrow \mathbb{R}$$
$$e \longmapsto f(e)$$

gán cho mỗi cung $e \in E$ một số thực $f(e)$ thỏa mãn ba ràng buộc:

- Ràng buộc về sức chứa (capacity constraint): tiền luồng trên mỗi cung không được vượt quá sức chứa của cung đó: $\forall e \in E: f(e) \leq c(e)$.
- Ràng buộc về tính đối xứng lệch (skew symmetry): Với $\forall e \in E$, tiền luồng trên cung $e$ và cung đối $-e$ có cùng giá trị tuyệt đối nhưng trái dấu nhau: $f(e) = -f(-e)$.
- Ràng buộc về tính dư: Với mọi đỉnh không phải đỉnh phát, tổng tiền luồng trên các cung đi vào đỉnh đó là số không âm: $\forall v \in V - \{s\}: f(V, \{v\}) = \sum_{e \in \{V \to \{v\}\}} f(e) \geq 0$.

Với $\forall v \in V$, ta gọi lượng tràn tại $v$, ký hiệu $excess[v]$, là tổng tiền luồng trên các cung đi vào đỉnh $v$:

$$excess[v] = f(V, \{v\}) = \sum_{e \in \{V \to \{v\}\}} f(e)$$

Đỉnh $v \in V - \{s, t\}$ gọi là *đỉnh tràn (overflowing vertex)* nếu $excess[v] > 0$. Khái niệm đỉnh tràn chỉ có nghĩa với các đỉnh không phải đỉnh phát cũng không phải đỉnh thu.

```
function Overflow(v∈V): Boolean;
begin
  Result := (v ≠ s) and (v ≠ t) and (excess[u] > 0);
end;
```

Định nghĩa về tiền luồng tương tự như định nghĩa luồng, chỉ khác nhau ở ràng buộc thứ ba. Vì vậy chúng ta cũng có khái niệm mạng thặng dư, cung thặng dư, đường thặng dư… ứng với tiền luồng tương tự như đối với luồng.

### b) Khởi tạo

Cho $f$ là một tiền luồng trên mạng $G = (V, E, c, s, t)$. Ta gọi $h: V \to \mathbb{N}$ là một hàm độ cao ứng với $f$ nếu $h$ gán cho mỗi đỉnh $v \in V$ một số tự nhiên $h(v)$ thỏa mãn ba điều kiện:

- $h(s) = |V|$.
- $h(t) = 0$.
- $h(u) \leq h(v) + 1$ với mọi cung thặng dư $(u, v)$.

Những ràng buộc này gọi là **ràng buộc độ cao**.

Hàm độ cao $h$ khi cài đặt sẽ được xác định bởi tập các giá trị $\{h[v]\}_{v \in V}$ nên tùy theo từng trường hợp, ta có thể sử dụng ký hiệu $h(v)$ (nếu muốn nói tới giá trị hàm) hoặc $h[v]$ (nếu muốn nói tới một biến số).

Thao tác khởi tạo $Init$ chịu trách nhiệm khởi tạo một tiền luồng và một hàm độ cao tương ứng. Một cách khởi tạo là đặt tiền luồng trên mỗi cung $e$ đi ra khỏi $s$ đúng bằng sức chứa $c(e)$ của cung đó (dĩ nhiên sẽ phải đặt cả tiền luồng trên cung đối $-e$ bằng $-c(e)$ để thỏa mãn tính đối xứng lệch), còn tiền luồng trên các cung khác bằng 0. Khi đó tất cả các cung đi ra khỏi $s$ là bão hòa.

$$f(e) = \begin{cases} c(e), \text{nếu } e \in E^+(s) \\ -c(e), \text{nếu } -e \in E^+(s) \\ 0, \text{trường hợp khác} \end{cases}$$

Ta khởi tạo hàm độ cao $h: V \to \mathbb{N}$ như sau:

$$h(v) = \begin{cases} |V|, \text{nếu } v = s \\ 0, \text{nếu } v = t \\ 1, \text{nếu } v \neq \{s, t\} \end{cases}$$

Rõ ràng mọi cung thặng dư $(u, v)$ không thể là cung đi ra khỏi $s$ ($u \neq s$) nên ta có $h(u) \leq 1 \leq h(v) + 1$. Hàm độ cao trên là thích ứng với tiền luồng $f$.

Việc cuối cùng là khởi tạo các giá trị $excess[.]$ ứng với tiền luồng $f$.

```
procedure Init;
```

```
begin
  //Khởi tạo tiền luồng
  for ∀e ∈E do f[e] := 0;
  for ∀v ∈V do excess[v] := 0;
  for ∀e =(s,v)∈E+(s) do
    begin
      f[e] := c(e);
      f[-e] := -c(e);
      excess[v] := excess[v] + c(e);
    end;
  //Khởi tạo hàm độ cao
  for ∀v ∈V do h[v] := 1;
  h[s] := |V|;
  h[t] := 0;
end;
```

### c) Phép đẩy luồng

Phép đẩy luồng $Push(e)$ có thể thực hiện trên cung $e = (u, v)$ nếu các điều kiện sau được thỏa mãn:

- $u$ là đỉnh tràn: $u \in V - \{s, t\}$ và $excess[u] > 0$
- $e$ là cung thặng dư trên $G_f$: $c_f(e) = c(e) - f(e) > 0$
- $u$ cao hơn $v$: $h(u) > h(v)$

Ràng buộc $h(u) > h(v)$ kết hợp với ràng buộc độ cao: $h(u) \leq h(v) + 1$ có thể viết thành $h(u) = h(v) + 1$.

Phép $Push\big(e = (u, v)\big)$ sẽ tính lượng luồng tối đa có thể thêm vào theo cung $e$: $\Delta = \min\{excess[u], c_f(e)\}$, thêm lượng luồng này vào cung $e$ và bớt một lượng luồng $\Delta$ từ $v$ về $u$ theo cung $-e$ để giữ tính đối xứng lệch của tiền luồng. Việc cuối cùng là cập nhật lại $excess[u]$ và $excess[v]$ theo tiền luồng mới. Bản chất của phép $Push\big(e = (u, v)\big)$ là chuyển một lượng luồng tràn $\Delta$ từ đỉnh $u$ sang đỉnh $v$. Dễ thấy rằng các tính chất của tiền luồng vẫn được duy trì sau phép $Push$:

```
procedure Push(e = (u,v));
begin
  Δ := min(excess[u], cf(u, v)); //Tính lượng luồng tối đa có thể đẩy
```

```
  f[e] := f[e] + Δ; f[-e] := f[-e] - Δ; //Đẩy luồng
  excess[u] := excess[u] - Δ;
  excess[v] := excess[v] + Δ; //Cập nhật mức tràn
end;
```

Phép $Push$ bảo tồn tính chất của hàm độ cao. Thật vậy, khi thao tác $Push\big(e = (u,v)\big)$ được thực hiện, nó chỉ có thể sinh ra thêm một cung thặng dư $-e = (v,u)$ mà thôi. Phép $Push$ không làm thay đổi các độ cao, tức là trước khi $Push$, $h[u] > h[v]$ thì sau khi $Push$, $h[v]$ vẫn nhỏ hơn $h[u]$, tức là ràng buộc độ cao $h[v] \leq h[u] + 1$ vẫn được duy trì trên cung thặng dư $-e = (v,u)$.

Phép $Push\big(e = (u,v)\big)$ đẩy một lượng luồng $\Delta = \min\{excess[u], c_f(e)\}$ tràn từ $u$ sang $v$. Nếu $\Delta$ đúng bằng $c_f(e) = c(e) - f(e)$ có nghĩa là khi phép $Push$ tăng $f(e)$ lên $\Delta$ thì cung $e$ sẽ bão hòa và không còn là cung thặng dư trên $G_f$ nữa, ta gọi phép đẩy luồng này là *đẩy bão hòa (saturating push)*, ngược lại phép đẩy luồng đó gọi là *đẩy không bão hòa (non-saturating push)*, sau phép đẩy không bão hòa thì $excess[u] = 0$, tức là $u$ không còn là đỉnh tràn nữa.

### d) Phép nâng

Phép nâng $Lift(u)$ thực hiện trên đỉnh $u$ nếu các điều kiện sau được thỏa mãn:

- $u$ là đỉnh tràn: $(u \neq s)$, $(u \neq t)$ và $excess[u] > 0$.
- $u$ không chuyển được luồng xuống nơi nào thấp hơn: Với mọi cung thặng dư $e = (u,v) \in E_f$: $h(u) \leq h(v)$.

Khi đó phép $Lift(u)$ nâng đỉnh $u$ lên bằng cách đặt $h[u]$ bằng độ cao thấp nhất của một đỉnh $v$ nó có thể chuyển tải sang cộng thêm 1:

$$h[u] := \min\{h[v] : \exists (u,v) \in E_f\} + 1$$

```
procedure Lift(u∈V);
begin
  minH := +∞;
  for ∀v:(u,v)∈E_f do
    if h[v] < minH then minH := h[v];
  h[u] := minH + 1;
end;
```

Nếu $u$ là đỉnh tràn thì ít nhất phải có một cung thặng dư đi ra khỏi $u$, điều này đảm bảo cho phép lấy $\min\{h[v]: (u,v) \in E_f\}$ được thực hiện trên một tập khác rỗng. Thật vậy, do $u$ là đỉnh tràn, ta có $excess[u] = \sum_{e \in \{V \to \{u\}\}} f(e) > 0$ tức là ít nhất có một cung $e \in \{V \to \{u\}\}$ để $f(e) > 0$. Cung đối $-e$ chắc chắn là một cung thặng dư đi ra khỏi $u$ bởi:

$$c_f(-e) = c(-e) - f(-e) = c(-e) + f(e) > 0$$

Phép $Lift$ không động chạm gì đến tiền luồng $f$. Ngoài ra phép $Lift$ **chỉ tăng độ cao của một đỉnh** và bảo tồn ràng buộc độ cao: Với một cung thặng dư $(v,u)$ đi vào $u$, ràng buộc độ cao $h(v) \leq h(u) + 1$ không bị vi phạm nếu ta nâng độ cao $h(u)$ của đỉnh $u$. Mặt khác, với một cung thặng dư $(u,v)$ đi ra khỏi $u$ thì việc đặt $h[u] := \min\{h[v]: \exists (u,v) \in E_f\} + 1$ cũng đảm bảo rằng $h(u) \leq h(v) + 1$.

### e) Mô hình chung và thuật toán FIFO Preflow-Push

❑ ***Mô hình chung***

Thuật toán đẩy tiền luồng có mô hình cài đặt chung khá đơn giản: Khởi tạo tiền luồng $f$ và hàm độ cao, sau đó nếu thấy phép nâng ($Lift$) hay đẩy luồng ($Push$) nào thực hiện được thì thực hiện nga… Cho tới khi không còn phép nâng hay đẩy nào có thể thực hiện được nữa thì tiền luồng $f$ sẽ trở thành luồng cực đại trên mạng.

Chính vì thứ tự các phép $Push$ và $Lift$ được thực hiện không ảnh hưởng tới tính đúng đắng của thuật toán nên người ta đã đề xuất rất nhiều cơ chế chọn thứ tự thực hiện nhằm giảm thời gian thực hiện giải thuật.

***Bổ đề 3-15***

Cho mạng $G = (V, E, c, s, t)$ có tiền luồng $f$ và hàm độ cao $h$. Với một đỉnh tràn $u$, luôn có thể thực hiện được thao tác $Push(e)$ trên một cung $e$ đi ra khỏi $u$ hoặc thực hiện được thao tác $Lift(u)$

***Chứng minh***

Nếu thao tác $Push$ không thể áp dụng được cho cung thặng dư nào đi ra khỏi $u$ tức là với mọi cung thặng dư $(u,v) \in E_f$, $h(u)$ không cao hơn $h(v)$, điều đó chính là điều kiện hợp lệ để thực hiện thao tác $Lift(u)$.

❑ ***Thuật toán FIFO Preflow-Push***

Định lý 3-17 là cơ sở cho thuật toán FIFO Preflow-Push. Thuật toán được Goldberg đề xuất [**18**] dựa trên cơ chế xử lý đỉnh tràn lấy ra từ một hàng đợi.

Tại thao tác khởi tạo, các đỉnh tràn sẽ được lưu trữ trong một hàng đợi $Queue$ hỗ trợ hai thao tác: $PushToQueue(v)$ để đẩy một đỉnh tràn $v$ vào hàng đợi và $PopFromQueue$ để lấy một đỉnh tràn khỏi hàng đợi. Thuật toán sẽ xử lý từng đỉnh tràn $z$ lấy ra khỏi hàng đợi theo cách sau: Trước hết cố gắng đẩy luồng trên các cung thặng dư đi ra khỏi $z$ bằng phép $Push$. Nếu đẩy được hết lượng tràn ($excess[z] = 0$) thì xong, nếu không ta dâng cao đỉnh $z$ bằng phép $Lift(z)$ và đẩy lại $z$ vào hàng đợi chờ xử lý sau. Thuật toán sẽ tiếp tục với đỉnh tràn tiếp theo trong hàng đợi và kết thúc khi hàng đợi rỗng, bởi khi mạng không còn đỉnh tràn thì không còn thao tác $Push$ hay $Lift$ nào có thể thực hiện được nữa.

Giả sử rằng chúng ta có một đỉnh tràn $u$ và một cung $e = (u, v)$ không thể đẩy luồng được, tức là ít nhất một trong hai điều kiện sau đây được thỏa mãn:

- $(u, v)$ là cung bão hòa $c(e) = f(e)$.
- $u$ không cao hơn $v$: $h(u) \leq h(v)$.

Khi đó:

- Sau bất kỳ phép $Push$ nào, chúng ta vẫn không thể đẩy luồng được trên cung $e = (u, v)$. Thật vậy, nếu $u$ không cao hơn $v$, phép $Push$ không làm thay đổi hàm độ cao nên sau phép $Push$ thì $u$ vẫn không cao hơn $v$. Nếu $u$ cao hơn $v$ thì $e$ phải là cung bão hòa, lệnh $Push$ duy nhất có thể biến nó thành cung thặng dư là lệnh $Push(-e)$ làm giảm $f(e)$. Nhưng lệnh $Push(-e)$ không thể thực hiện được vì cung $-e = (v, u)$ có $v$ thấp hơn $u$.
- Sau bất kỳ phép $Lift$ nào ngoại trừ $Lift(u)$, chúng ta cũng không thể đẩy luồng được trên cung $e = (u, v)$. Bởi phép $Lift$ không làm thay đổi tiền luồng trên các cung, tính bão hòa hay thặng dư của các cung được giữ nguyên. Như vậy nếu $(u, v)$ đang bão hòa thì sau phép $Lift$ nó vẫn bão hòa và không thể đẩy luồng được. Nếu $(u, v)$ là cung thặng dư thì $u$ đang không cao hơn $v$, lệnh $Lift$ duy nhất có thể khiến $u$ cao hơn $v$ là lệnh $Lift(u)$.

Hai nhận định trên cho phép ta xây dựng một cấu trúc dữ liệu hiệu quả để cài đặt thuật toán:

Tương tự như chương trình cài đặt thuật toán Edmonds-Karp, ta sử dụng mảng $e[-m \ldots m]$ chứa các cung, mảng $link[m \ldots m]$ chứa móc nối trong danh sách liên thuộc và mảng $head[1 \ldots n]$ chứa chỉ số cung đầu tiên của các danh sách liên thuộc. Ngoài ra thuật toán duy trì một mảng chỉ số $current[1 \ldots n]$, ở đây $current[v]$ là chỉ số của một cung nào đó trong danh sách liên thuộc các cung đi ra khỏi $v$, ban đầu $current[v]$ được gán bằng $head[v]$ với mọi đỉnh $v \in V$.

```
type
  TEdge = record //Cấu trúc một cung
    x, y: Integer; //Hai đỉnh đầu mút
    c, f: Integer; //Sức chứa và luồng
  end;
var
  e: array[-maxM..maxM] of TEdge; //Danh sách các cung
  link: array[-maxM..maxM] of Integer;
  //Móc nối trong danh sách liên thuộc
  head, current: array[1..maxN] of Integer;
```

Trên cấu trúc dữ liệu này, danh sách móc nối các nút chứa các cung đi ra khỏi $z$ là:

$$e[i_1], e[i_2], e[i_3], \ldots$$

Trong đó $i_1 = head[z]$, $i_2 = link[i_1]$, $i_3 = link[i_2]$,…

Thuật toán FIFO Preflow-Push sẽ xử lý lần lượt từng đỉnh tràn lấy ra khỏi hàng đợi. Với mỗi đỉnh tràn $z$ lấy khỏi hàng đợi, cung $e\big[current[z]\big]$ là một cung đi ra khỏi $z$, giả sử cung đó là $(z, v)$. Nếu phép đẩy luồng ($Push$) trên cung đó có thể thực hiện được thì thực hiện ngay, đồng thời đẩy $v$ vào hàng đợi nếu $v$ chưa có trong hàng đợi. Nếu phép đẩy luồng này làm $z$ hết tràn thì chuyển sang xử lý đỉnh tràn kế tiếp trong hàng đợi, ngược lại nếu $z$ vẫn còn là đỉnh tràn (tức là không thể đẩy luồng trên cung $e\big[current[z]\big]$ nữa), ta dịch chỉ số $current[z]$ sang cung kế tiếp trong danh sách liên thuộc ($current[z] := link\big[current[z]\big]$) để chuyển sang xét một cung khác…Khi dịch chỉ số $current[z]$ đến hết danh sách liên thuộc mà $z$ vẫn tràn, đỉnh $z$ sẽ được nâng lên bằng phép $Lift(z)$, chỉ số $current[x]$ được đặt trở lại bằng $head[z]$ để nó trỏ lại về đầu danh sách liên thuộc. Đỉnh $z$ sau đó được đẩy lại vào hàng đợi chờ xử lý sau…

Tính hợp lý của thuật toán nằm ở chỗ : khi đỉnh tràn $z$ bắt đầu được xử lý, tất cả những cung đứng trước cung $e[current[z]]$ đều không thể đẩy luồng được. Tức là nếu muốn đẩy luồng ra khỏi $z$ thì chỉ cần xét các cung từ $e[current[z]]$ trở đi là đủ, không cần duyệt từ đầu danh sách liên thuộc.

```
procedure FIFOPreflowPush;
begin
  Init; //Khởi tạo tiền luồng, độ cao, hàng đợi Queue chứa các đỉnh tràn
  while Queue ≠ Ø do
    begin
      z := PopFromQueue; //Xử lý đỉnh tràn x lấy ra từ hàng đợi
      while current[z] <> 0 do //Cố gắng đẩy luồng khỏi z
        begin //Xét cung (z, v) chứa trong nút e[current[z]]
          v := e[current[z]].y;
          if «Có thể đẩy luồng trên cung (z, v)» then
            begin
              NeedQueue := (v ≠ s) and (v ≠ t)
                                   and (excess[v] = 0);
              Push(z, v); //Đẩy luồng
              if NeedQueue then
              //Sau phép đẩy, v đang không tràn trở thành tràn
                PushToQueue(v); //Đẩy v vào hàng đợi chờ xử lý
              if excess[z] = 0 then Break;
              //Sau phép đẩy mà z hết tràn thì dừng đẩy
            end;
          current[z] := link[current[z]];
          //z chưa hết tràn, chuyển sang xét cung liên thuộc tiếp theo
        end;
      if excess[z] > 0 then //Duyệt hết danh sách liên thuộc mà x vẫn tràn
        begin
          Lift(z); //Dâng cao z
          current[z] := head[z];
          //Đặt lại chỉ số current[z] về nút đầu danh sách liên thuộc
          PushToQueue(z); //Đẩy z vào hàng đợi chờ xử lý sau
        end;
    end;
end;
```

Từ nhận xét trên, có thể nhận thấy rằng những phép $Push$ và $Lift$ trong mô hình cài đặt đảm bảo được gọi tại những thời điểm mà những điều kiện cần để thực thi chúng được thỏa mãn.

### f) Tính đúng của thuật toán

Sau mỗi bước của vòng lặp chính, hàng đợi $Queue$ luôn chứa danh sách các đỉnh tràn và thuật toán sẽ kết thúc khi không còn đỉnh tràn nào trên mạng. Với $\forall v \in V - \{s,t\}$, ta có:

$$f(\{v\},V) = -f(V,\{v\}) = -excess[v] = 0$$

Tức là với $\forall v \in V - \{s,t\}$ thì tổng luồng trên các cung đi ra khỏi $v$ bằng 0, điều này chỉ ra rằng khi thuật toán kết thúc, tiền luồng chúng ta duy trì trên mạng trở thành một luồng.

***Định lý 3-16***

Cho $f$ là một tiền luồng trên mạng $G = (V,E,c,s,t)$, nếu tồn tại một hàm độ cao $h: V \to \mathbb{N}$ ứng với $f$ thì mạng thặng dư $G_f$ không có đường tăng luồng.

> ***Chứng minh***
>
> Nhắc lại về ràng buộc độ cao: $h(s) = |V|$, $h(t) = 0$ và với mọi cung thặng dư $(u,v)$ thì $h(u) \le h(v) + 1$. Giả sử phản chứng rằng có đường tăng luồng $\langle s = v_0, v_1, \dots, v_k = t\rangle$ trên mạng thặng dư $G_f$ đi qua $k$ cung thặng dư. Khi đó:
>
> $$h(v_0) \le h(v_1) + 1 \le h(v_2) + 2 \le \cdots \le h(v_k) + k$$
>
> hay
>
> $$\underbrace{h(s)}_{|V|} \le \underbrace{h(t)}_{0} + k$$
>
> Ta có $|V| \le k$, nhưng đường tăng luồng phải là đường đi đơn, tức là qua không quá $|V| - 1$ cạnh, vậy $k \le |V| - 1$. Điều này mâu thuẫn, nghĩa là không thể tồn tại đường tăng luồng trên $G_f$.

Định lý 3-17 và Định lý 3-11 (mối quan hệ giữa luồng cực đại, đường tăng luồng và lát cắt hẹp nhất) chỉ ra rằng: thuật toán đẩy tiền luồng trả về một luồng và một hàm độ cao ứng với luồng đó nên luồng trả về chắc chắn là luồng cực đại.

### g) Tính dừng của thuật toán

Tính dừng của thuật toán đẩy tiền luồng ở trên sẽ được suy ra khi chúng ta phân tích thời gian thực hiện giải thuật. Tương tự như thuật toán Ford-Fulkerson, chúng ta sẽ không phân tích thời gian thực hiện trên mô hình tổng quát mà chỉ phân tích thời gian thực hiện giải thuật FIFO Preflow-Push mà thôi.

***Định lý 3-17***

Cho $f$ là một tiền luồng trên mạng $G = (V, E, c, s, t)$, khi đó với mọi đỉnh tràn $u$, tồn tại một đường thặng dư đi từ $u$ tới $s$.

***Chứng minh***

Với một đỉnh tràn $u$ bất kỳ, xét tập $X$ là tập các đỉnh có thể đến được từ $u$ bằng một đường thặng dư. Đặt $Y = V - X$ là tập những đỉnh nằm ngoài $X$. Trước hết ta chỉ ra rằng tiền luồng trên các cung thuộc $\{Y \to X\}$ không thể là số dương. Thật vậy nếu có $e \in \{Y \to X\}$ mà $f(e) > 0$ thì $-e \in \{X \to Y\}$ và $f(-e) < 0$. Suy ra có cung thặng dư $-e$ nối một đỉnh thuộc $X$ với một đỉnh $y$ nào đó thuộc $Y$. Theo cách xây dựng tập $X$, $y$ sẽ phải là đỉnh thuộc $X$. Mâu thuẫn.

Tiền luồng trên các cung thuộc $\{Y \to X\}$ không thể là số dương thì $f(Y, X) \leq 0$. Ta xét tổng mức tràn của các đỉnh $\in X$:

$$excess(X) = f(V, X) = \underbrace{f(X, X)}_{0} + \underbrace{f(Y, X)}_{\leq 0} \leq 0$$

Lượng tràn tại mỗi đỉnh không phải đỉnh phát đều là số không âm, ngoài ra $u$ là đỉnh tràn $\in X$ nên $excess[u] > 0$, điều này cho thấy chắc chắn đỉnh phát $s$ phải thuộc $X$ để $excess(X) \leq 0$. Nói cách khác từ $u$ đến được $s$ bằng một đường thặng dư.

***Hệ quả***

Cho mạng $G = (V, E, c, s, t)$. Giả sử chúng ta thực hiện thuật toán đẩy tiền luồng với hàm độ cao $h: V \to \mathbb{N}$ thì độ cao của các đỉnh trong quá trình thực hiện giải thuật không vượt quá $2|V| - 1$.

***Chứng minh***

Mạng phải có ít nhất một đỉnh phát và một đỉnh thu nên $|V| \geq 2$. Ban đầu, $h(s) = |V|$, $h(t) = 0$ và $h(v) = 1, \forall v \notin \{s, t\}$ nên độ cao của các đỉnh đều nhỏ hơn $2|V| - 1$.

Độ cao của $s$ và $t$ không bao giờ bị thay đổi và với mỗi đỉnh $u \in V - \{s, t\}$ thì chỉ phép $Lift(u)$ có thể làm tăng độ cao của đỉnh $u$. Điều kiện để thực hiện phép

$Lift(u)$ là $u$ phải là đỉnh tràn. Phép $Lift$ không thay đổi tiền luồng nên sau phép $Lift(u)$ thì $u$ vẫn tràn. Áp dụng kết quả của Định lý 3-17, tồn tại đường đi đơn từ $u$ tới $s$ $\langle u = v_0, v_1, \dots, v_k = s\rangle$ chỉ đi qua $k$ cung thặng dư $(v_0, v_1)$,$(v_1, v_2)$,…,$(v_{k-1}, v_k)$. Từ ràng buộc độ cao ta có:

$$h(u) = h(v_0) \le h(v_1) + 1 \le h(v_2) + 2 \le \cdots \le h(v_k) + k \le |V| + k$$

Đường đi đơn thì không qua nhiều hơn $|V| - 1$ cạnh nên ta có $k \le |V| - 1$, kết hợp lại có $h(u) \le 2|V| - 1$. ĐPCM.

***Định lý 3-18 (thời gian thực hiện giải thuật FIFO Preflow-Push)***

Có thể cài đặt giải thuật FIFO Preflow-Push để tìm luồng cực đại trên mạng $G = (V, E, c, s, t)$ trong thời gian $O(|V|^3 + |V||E|)$.

***Chứng minh***

Ta sẽ chứng minh mô hình cài đặt thuật toán FIFO Preflow-Push ở trên có thời gian thực hiện là $O(|V|^3 + |V||E|)$. Vòng lặp chính của thuật toán mỗi lượt lấy một đỉnh tràn $z$ khỏi hàng đợi và cố gắng tháo luồng cho đỉnh $z$ bằng các phép $Push$ theo các cung đi ra khỏi $z$. Nếu $z$ chưa hết tràn thì thực hiện phép $Lift(z)$ và đẩy lại $z$ vào hàng đợi. Như vậy thuật toán FIFO Preflow-Push sẽ thực hiện một dãy các phép $Lift$ và $Push$:

$$Lift(.), Push(.), Push(.) \dots, Push(.), Lift(.), Push(.), \dots$$

Trước hết ta chứng minh rằng số phép $Lift$ trong dãy thao tác trên là $O(|V|^2)$ và tổng thời gian thực hiện chúng là $O(|V||E|)$. Thật vậy, Mỗi phép $Lift$ sẽ nâng độ cao của một đỉnh lên ít nhất 1, ngoài ra độ cao của mỗi đỉnh không vượt quá $2|V| - 1$ (theo hệ quả của định lý Định lý 3-17). Cứ cho là mọi đỉnh $\in V - \{s, t\}$ khi kết thúc thuật toán đều có độ cao $2|V| - 1$ đi nữa thì do chúng được khởi tạo bằng 1, tổng số phép $Lift$ cần thực hiện cũng không vượt quá:

$$(|V| - 2)(2|V| - 2) = O(|V|^2)$$

Mỗi cung $(u, v)$ sẽ được xét đến đúng một lần trong phép $Lift(u)$, phép $Lift(u)$ lại được gọi không quá $2|V| - 2$ lần. Vậy tổng cộng trong tất cả các phép $Lift$ thì mỗi cung sẽ được xét không quá $2|V| - 2$ lần, mạng có $|E|$ cung suy ra tổng thời gian thực hiện của các phép $Lift$ trong giải thuật là $|E|(2|V| - 2) = O(|V||E|)$.

Tiếp theo ta chứng minh rằng số phép đẩy bão hòa cũng như tổng thời gian thực hiện chúng là $O(|V||E|)$. Sau phép đẩy bão hòa $Push(e)$, nếu muốn thực hiện tiếp phép đẩy $Push(e)$ nữa thì trước đó chắc chắn phải có phép đẩy $Push(-e)$ để làm giảm $f(e)$ và biến $e$ trở lại thành cung thặng dư. Giả sử $e = (u, v)$ và $-e = (v, u)$ thì để thực hiện phép $Push(e)$, ta phải có $h(u) > h(v)$. Để thực hiện $Push(-e)$, ta phải có $h(v) > h(u)$ và để thực hiện tiếp $Push(e)$ nữa ta lại phải có $h(u) >$

$h(v)$. Bởi độ cao của các đỉnh không bao giờ giảm đi nên sau phép $Push(e)$ thứ hai, độ cao $h(u)$ lớn hơn ít nhất 2 đơn vị so với $h(u)$ ở phép $Push(e)$ thứ nhất. Vậy nếu một cung $e = (u, v)$ của mạng được đẩy bão hòa $k$ lần thì độ cao của đỉnh $u$ sẽ tăng lên ít nhất là $2(k-1)$. Vì độ cao của các đỉnh không vượt quá $2|V|-1$ nên số phép đẩy bão hòa trên mỗi cung $e$ là $k \leq |V|$. Mạng có $|E|$ cung và thời gian thực hiện phép $Push$ là $O(1)$ nên số phép đẩy bão hòa là $O(|V||E|)$ và thời gian thực hiện chúng cũng là $O(|V||E|)$.

Đối với các phép đẩy không bão hòa, việc đánh giá thời gian thực hiện giải thuật được thực hiện bằng *hàm tiềm năng* (*potential function*). Định nghĩa hàm tiềm năng $\Phi$ là độ cao lớn nhất của các đỉnh tràn:

$$\Phi = \max\{h[v]: v \text{ là đỉnh tràn}\}$$

Trong trường hợp mạng không còn đỉnh tràn thì ta quy ước $\Phi = 0$. Vậy $\Phi \leq 1$ khi khởi tạo tiền luồng và trở lại bằng 0 khi thuật toán kết thúc.

Chia dãy các thao tác $Lift$ và $Push$ làm các pha liên tiếp. Pha thứ nhất bắt đầu khi hàng đợi được khởi tạo gồm các đỉnh tràn và kết thúc khi tất cả các đỉnh đó (và chỉ những đỉnh đó thôi) đã được lấy ra khỏi hàng đợi và xử lý. Pha thứ hai tiếp tục với hàng đợi gồm những đỉnh được đẩy vào trong pha thứ nhất và kết thúc khi tất cả các đỉnh này được lấy ra khỏi hàng đợi và xử lý, pha thứ ba, thứ tư… được chia ra theo cách tương tự như vậy.

Nhận xét rằng phép $Push$ chỉ đẩy luồng từ đỉnh cao xuống đỉnh thấp, vậy nên những đỉnh được đẩy vào hàng đợi sau phép $Push$ luôn thấp hơn đỉnh đang xét vừa lấy ra khỏi hàng đợi. Suy ra nếu một pha chỉ chứa phép $Push$ thì giá trị hàm tiềm năng $\Phi$ sau pha đó giảm đi ít nhất 1 đơn vị.

Giá trị hàm tiềm năng $\Phi$ chỉ **có thể** tăng lên sau một pha nếu pha đó có chứa phép $Lift$ và giá trị $\Phi$ tăng lên phải bằng một độ cao của một đỉnh $v$ nào đó sau phép $Lift(v)$ trong pha. Xét mức tăng của $\Phi$ sau pha đang xét:

$$\Phi_{\text{mới}} - \Phi_{\text{cũ}} = h(v)_{\text{mới}} - \Phi_{\text{cũ}} \leq h(v)_{\text{mới}} - h(v)_{\text{cũ}}$$

Tức là sau mỗi pha làm $\Phi$ tăng lên, luôn tồn tại một đỉnh $v$ mà mức tăng độ cao của $v$ lớn hơn mức tăng của $\Phi$. Xét trên toàn bộ giải thuật, độ cao của mỗi đỉnh $v \in V - \{s, t\}$ được khởi tạo bằng 0 và được nâng lên tối đa bằng $2|V|-1$ nên tổng toàn bộ mức tăng của các đỉnh không vượt quá $(|V|-2)(2|V|-1) = O(|V|^2)$.

Vậy nếu ta xét các pha làm $\Phi$ tăng thì tổng mức tăng của $\Phi$ trên các pha này là $O(|V|^2)$, tức là số các pha làm $\Phi$ giảm cũng phải là $O(|V|^2)$. Nói cách khác, sẽ chỉ có $O(|V|^2)$ pha có chứa phép $Lift$ và $O(|V|^2)$ pha không chứa phép $Lift$. Cộng lại ta có số pha cần thực hiện trong toàn bộ giải thuật là $O(|V|^2)$.

Một pha sẽ phải lấy khỏi hàng đợi tối đa $|V|-2$ đỉnh để xử lý. Với mỗi đỉnh lấy từ hàng đợi, việc tháo luồng sẽ chỉ sử dụng tối đa 1 phép đẩy không bão hòa vì sau phép đẩy này thì đỉnh sẽ hết tràn và quá trình xử lý sẽ chuyển sang đỉnh tiếp theo trong hàng đợi. Vậy trong mỗi pha có không quá $|V|-2$ phép đẩy không bão hòa. Vì tổng số pha là $O(|V|^2)$, ta có số phép đẩy không bão hòa trong cả giải thuật là $O(|V|^3)$ và tổng thời gian thực hiện chúng cũng là $O(|V|^3)$.

Cuối cùng, ta đánh giá thời gian thực hiện những thao tác duyệt danh sách liên thuộc bằng các chỉ số $current[.]$ trong thuật toán FIFO Preflow-Push. Với mỗi đỉnh $z$, chỉ số $current[z]$ ban đầu sẽ ứng với nút đầu danh sách liên thuộc và chuyển dần đến hết danh sách gồm $\deg^+(z)$ nút. Khi duyệt hết danh sách liên thuộc mà $z$ vẫn tràn thì sẽ có một phép $Lift(z)$ và con trỏ $current[z]$ được đặt lại về đầu danh sách liên thuộc. Số phép $Lift(z)$ trong toàn bộ giải thuật không vượt quá $2|V|-1$, nên số lượt dịch chỉ số $current[z]$ không vượt quá $2|V|\deg^+(z)$. Suy ra nếu xét tổng thể, số phép dịch các chỉ số $current[.]$ trên tất cả các danh sách liên thuộc phải nhỏ hơn:

$$(2|V|)\sum_{z\in V}\deg^+(z) = 2|V||E| = O(|V||E|)$$

Kết luận:

Tổng thời gian thực hiện các phép nâng: $O(|V||E|)$.

Tổng thời gian thực hiện các phép đẩy bão hòa: $O(|V||E|)$.

Tổng thời gian thực hiện các phép đẩy không bão hòa: $O(|V|^3)$.

Tổng thời gian thực hiện các phép duyệt danh sách liên thuộc bằng chỉ số $current[.]$: $O(|V||E|)$.

Thời gian thực hiện giải thuật FIFO Preflow-Push: $O(|V|^3+|V||E|)$.

### h) Một số kỹ thuật tăng tốc độ giải thuật

Ta đã chứng minh rằng thuật toán Edmonds-Karp có thời gian thực hiện $O(|V||E|^2)$ và thuật toán FIFO Preflow-Push có thời gian thực hiện $O(|V|^3+|V||E|)$. Những đại lượng này thoạt nhìn làm chúng ta có cảm giác như thuật toán FIFO Preflow-Push thực hiện nhanh hơn thuật toán Edmonds-Karp, đặc biệt trong trường hợp đồ thị dày ($|E| \gg |V|$).

Tuy vậy, những đánh giá này chỉ là cận trên của thời gian thực hiện giải thuật trong trường hợp xấu nhất. Hiện tại chưa có các đánh giá chặt về cận trên và cận dưới trong trường hợp trung bình. Những thử nghiệm bằng chương trình cụ thể cũng cho thấy rằng các thuật toán đẩy tiền luồng như FIFO Preflow-Push, Lift-

to-Front Preflow-Push, Highest-Label Preflow-Push… không có có cải thiện gì về tốc độ so với thuật toán Edmonds-Karp (thậm chí còn chậm hơn) nếu không sử dụng những mẹo cài đặt (*heuristics*).

Chưa có đánh giá lý thuyết chặt chẽ nào về tác động của những mẹo cài đặt lên thời gian thực hiện giải thuật nhưng hầu hết các thử nghiệm đều cho thấy việc sử dụng những mẹo cài đặt trên thực tế gần như là bắt buộc đối với các thuật toán đẩy tiền luồng.

❑ ***Bản chất của hàm độ cao***

Nhắc lại về ràng buộc độ cao: Xét một tiền luồng $f$ trên mạng $G = (V, E, c, s, t)$, hàm độ cao $h: V \to \mathbb{N}$ gọi là tương ứng với tiền luồng $f$ nếu $h(s) = |V|$; $h(t) = 0$; và với mọi cung thặng dư $(u, v)$ thì $h(u) \leq h(v) + 1$.

Nếu ta gán trọng số cho các cung của mạng thặng dư $G_f$ theo quy tắc: Cung thặng dư có trọng số 1 và cung bão hòa có trọng số $+\infty$. Ký hiệu $\delta_f(u, v)$ là độ dài đường đi ngắn nhất từ $u$ tới $v$ trên $G_f$ với cách gán trọng số này. Khi đó không khó khăn kiểm chứng được rằng với $\forall v \in V - \{s, t\}$:

- $h(v) \leq \delta_f(v, t)$, tức là $h(v)$ luôn là cận dưới của độ dài đường đi ngắn nhất từ $v$ tới đỉnh thu.
- Trong trường hợp $h(v) > |V|$, từ $v$ chắc chắn không có đường thặng dư đi tới $t$ và $h(v) - |V| \leq \delta_f(v, s)$, tức là $h(v) - |V|$ trong trường hợp này là cận dưới của độ dài đường đi ngắn nhất từ $v$ về đỉnh phát.

Những mẹo cài đặt dưới đây nhằm đẩy nhanh các độ cao $h(v)$ trong tiến trình thực hiện giải thuật dựa vào những nhận xét trên.

❑ ***Gán nhãn lại toàn bộ***

Nội dung của phương pháp gán nhãn lại toàn bộ (*global relabeling heuristic*) được tóm tắt như sau: Xét lát cắt chia tập $V$ làm hai tập rời nhau $X$ và $Y$: Tập $Y$ gồm những đỉnh đến được $t$ bằng một đường thặng dư và tập $X$ gồm những đỉnh còn lại. Chắc chắn không có cung thặng dư nối từ $X$ sang $Y$, ta có $s \in X, t \in Y$. Phép gán nhãn lại toàn bộ sẽ đặt:

- Với $\forall v \in Y$, ta gán lại độ cao $h[v] \coloneqq \delta_f(v, t)$.
- Với $\forall u \in X$ và $\delta_f(u, s) < +\infty$, ta gán lại độ cao $h[u] \coloneqq |V| + \delta_f(u, s)$

- Với $\forall u \in X$ và $\delta_f(u,s) = +\infty$, ta gán lại độ cao $h[u] \coloneqq 2|V| - 1$

Không khó khăn để kiểm chứng tính hợp lý của hàm độ cao mới. Có thể thấy rằng các độ cao mới ít ra là không thấp hơn các độ cao cũ.

Các giá trị $\delta_f(v,t)$ cũng như $\delta_f(u,s)$ có thể được xác định bằng hai lượt thực hiện thuật toán BFS từ $t$ và $s$. Bởi ta cần thời gian $O(|E|)$ cho hai lượt BFS và gán lại các độ cao, nên phép gán nhãn lại toàn bộ thường được gọi thực hiện sau một loạt chỉ thị sơ cấp để không làm ảnh hưởng tới đánh giá O lớn của thời gian thực hiện giải thuật (chẳng hạn sau mỗi $|V|$ phép $Lift$). Chú ý là khi nâng độ cao $h[z]$ của một đỉnh $z$ nào đó, cần cập nhật lại $current[z] \coloneqq head[z]$.

❑ ***Đẩy nhãn theo khe***

Phép đẩy nhãn theo khe (*gap heuristic*) được thực hiện nhờ quan sát sau:

Giả sử ta có một số nguyên $0 < gap < |V|$ mà không đỉnh nào có độ cao $gap$ (số nguyên $gap$ này được gọi là "khe"), khi đó mọi đỉnh $z$ có $h[z] > gap$ đều không có đường thặng dư đi đến $t$.

Nhận định trên có thể chứng minh bằng phản chứng: Giả sử từ $z$ có đường thặng dư đi đến $t$, với một cung $(u,v)$ trên đường đi ta có $h(u) \leq h(v) + 1$, tức là trên đường đi này, từ một đỉnh $u$ ta chỉ có thể đi sang một đỉnh $v$ không thấp hơn hoặc thấp hơn $u$ đúng một đơn vị. Từ $h(z) > gap > 0$ và $h(t) = 0$, chắc chắn trên đường thặng dư từ $x$ tới $t$ phải có một đỉnh độ cao $gap$. Mâu thuẫn với giả thuyết phản chứng.

Phép đẩy nhãn theo khe nếu phát hiện khe $0 < gap < |V|$ sẽ xét tất cả những đỉnh $z \in V - \{s\}$ có $gap < h(z) \leq |V|$ và đặt lại $h[z] \coloneqq |V| + 1$.

Ta sẽ chỉ ra rằng phép đẩy độ cao này vẫn đảm bảo ràng buộc độ cao của hàm $h$. Độ cao của đỉnh phát và đỉnh thu không bị động chạm đến, tức là $h[s] = |V|$ và $h[t] = 0$. Trước khi thực hiện phép đẩy theo khe, ta chia tập đỉnh $V$ thành hai tập rời nhau: Tập $X$ gồm những đỉnh cao hơn $gap$ và tập $Y$ gồm những đỉnh thấp hơn $gap$. Do ràng buộc độ cao $h(u) \leq h(v) + 1$ với mọi cung thặng dư $(u,v)$, không tồn tại cung thặng dư nối từ $X$ tới $Y$. Phép đẩy theo khe chỉ tăng độ cao của một vài đỉnh $x \in X$ và như vậy ràng buộc độ cao nếu bị vi phạm thì chỉ bị vi phạm trên những cung thặng dư đi ra khỏi $x$. Như lập luận trên, cung thặng dư đi

ra khỏi $x$ chắc chắn phải đi vào một đỉnh $x' \in X$ có độ cao ít nhất là $|V|$ sau phép đẩy theo khe. Từ $h(x) = |V| + 1$ ta có $h(x) \leq h(x') + 1$.

Phép đẩy nhãn theo khe sử dụng mảng $count[0 \dots 2|V| - 1]$ để đếm $count[k]$ là số đỉnh có độ cao $k$. Mỗi khi có sự thay đổi độ cao, ta phải đồng bộ lại mảng $count$ theo tình trạng hàm độ cao mới. Sau mỗi phép $Lift(u)$, độ cao cũ của đỉnh $u$ được lưu trữ lại trong biến $OldH$ và phép $Lift$ thực hiện như bình thường. Sau đó nếu $0 < OldH < |V|$ và $count[OldH] = 0$, phép đẩy theo khe $OldH$ sẽ được gọi và thực hiện trong thời gian $O(|V|)$. Bởi số phép $Lift$ cần thực hiện trong toàn bộ giải thuật là $O(|V|^2)$, tổng thời gian thực hiện các phép đẩy theo khe sẽ là $O(|V|^3)$ nên không ảnh hưởng tới đánh giá O-lớn của thời gian thực hiện giải thuật FIFO Preflow-Push.

Dưới đây là bảng so sánh tốc độ của các chương trình cài đặt cụ thể trên một số bộ dữ liệu. Với một cặp số $n, m$, 100 đồ thị với $n$ đỉnh, $m$ cung được sinh ngẫu nhiên với sức chứa là số nguyên trong khoảng từ 0 tới $10^4$. Có 4 chương trình được thử nghiệm: A: Thuật toán Edmonds-Karp, B: thuật toán FIFO Preflow-Push, C: thuật toán FIFO Preflow-Push với phép gán nhãn lại toàn bộ và D: thuật toán FIFO Preflow-Push với phép đẩy nhãn theo khe. Mỗi chương trình được thử trên cả 100 đồ thị và đo thời gian thực hiện trung bình (tính bằng giây):

| | $n = 100$<br>$m = 10000$ | $n = 200$<br>$m = 30000$ | $n = 500$<br>$m = 40000$ | $n = 800$<br>$m = 90000$ | $n = 1000$<br>$m = 100000$ |
|---|---|---|---|---|---|
| A | 0.0688 | 0.5925 | 0.6598 | 1.7158 | 2.7629 |
| B | 0.0983 | 0.7395 | 3.4377 | 9.9014 | 25.0723 |
| C | 0.0313 | 0.0624 | 0.0857 | 0.1809 | 0.2433 |
| D | 0.0282 | 0.0577 | 0.0828 | 0.1575 | 0.1889 |

❑ ***Cài đặt***

Dưới đây là chương trình cài đặt thuật toán FIFO Preflow-Push kết hợp với kỹ thuật đẩy nhãn theo khe, việc cài đặt và đánh giá hiệu suất của phép gán nhãn lại toàn bộ chúng ta coi như bài tập. Các bạn có thể thử cài đặt kết hợp cả hai kỹ thuật tăng tốc này để xác định xem việc đó có thực sự cần thiết không.

Input/Output có khuôn dạng giống như ở chương trình cài đặt thuật toán Edmonds-Karp. Hàng đợi chứa các đỉnh tràn được tổ chức dưới dạng danh sách vòng: Các chỉ số đầu/cuối hàng đợi sẽ chạy xuôi trong một mảng và khi chạy đến hết mảng sẽ tự động quay về đầu mảng.

💻 FIFOPREFLOWPUSH.PAS ✓ Thuật toán FIFO Preflow-Push

```
{$MODE OBJFPC}
program MaximumFlow;
const
  maxN = 1000;
  maxM = 100000;
  maxC = 10000;
type
  TEdge = record //Cấu trúc một cung
    x, y: Integer; //Hai đỉnh đầu mút
    c, f: Integer; //Sức chứa và luồng
  end;
  TQueue = record //Cấu trúc hàng đợi
    items: array[0..maxN - 1] of Integer; //Danh sách vòng
    front, rear, nItems: Integer;
  end;
var
  e: array[-maxM..maxM] of TEdge; //Mảng chứa các cung
  link: array[-maxM..maxM] of Integer;
  //Móc nối trong danh sách liên thuộc
  head, current: array[1..maxN] of Integer;
  //con trỏ tới đầu và vị trí hiện tại của danh sách liên thuộc
  excess: array[1..maxN] of Integer; //mức tràn của các đỉnh
  h: array[1..maxN] of Integer; //hàm độ cao
  count: array[0..2 * maxN - 1] of Integer;
  //count[k] = số đỉnh có độ cao k
  Queue: TQueue; //Hàng đợi chứa các đỉnh tràn
  n, m, s, t: Integer;
  FlowValue: Integer;
procedure Enter; //Nhập dữ liệu
var
  i, u, v, capacity: Integer;
begin
```

```
  ReadLn(n, m, s, t);
  FillChar(head[1], n * SizeOf(head[1]), 0);
  for i := 1 to m do
    begin
      ReadLn(u, v, capacity);
      with e[i] do //Thêm cung e[i] = (u, v) vào danh sách liên thuộc của u
        begin
          x := u;
          y := v;
          c := capacity;
          link[i] := head[u];
          head[u] := i;
        end;
      with e[-i] do //Thêm cung e[-i] = (v, u) vào danh sách liên thuộc của v
        begin
          x := v;
          y := u;
          c := 0;
          link[-i] := head[v];
          head[v] := -i;
        end;
    end;
  for v := 1 to n do current[v] := head[v];
end;
procedure PushToQueue(v: Integer); //Đẩy một đỉnh v vào hàng đợi
begin
  with Queue do
    begin
      rear := (rear + 1) mod maxN;
       //Dịch chỉ số cuối hàng đợi, rear = maxN - 1 sẽ trở lại thành 0
      items[rear] := v; //Đặt v vào vị trí cuối hàng đợi
      Inc(nItems); //Tăng biến đếm số phần tử trong hàng đợi
    end;
end;
function PopFromQueue: Integer; //Lấy một đỉnh khỏi hàng đợi
begin
  with Queue do
    begin
```

```
      Result := items[front]; //Trả về phần tử ở đầu hàng đợi
      front := (front + 1) mod maxN;
      //Dịch chỉ số đầu hàng đợi, front = maxN - 1 sẽ trở lại thành 0
      Dec(nItems); //Giảm biến đếm số phần tử trong hàng đợi
    end;
end;
procedure Init; //Khởi tạo
var v, sf, i: Integer;
begin
  //Khởi tạo tiền luồng
  for i := -m to m do e[i].f := 0;
  FillChar(excess[1], n * SizeOf(excess[1]), 0);
  i := head[s];
  while i <> 0 do
//Duyệt các cung đi ra khỏi đỉnh phát và đẩy bão hòa các cung đó, cập nhật các mức tràn excess[.]
    begin
      sf := e[i].c;
      e[i].f := sf;
      e[-i].f := -sf;
      Inc(excess[e[i].y], sf);
      Dec(excess[s], sf);
      i := link[i];
    end;
  //Khởi tạo hàm độ cao
  for v := 1 to n do h[v] := 1;
  h[s] := n;
  h[t] := 0;
  //Khởi tạo các biến đếm: count[k] là số đỉnh có độ cao k
  FillChar(count[0], (2 * n) * SizeOf(count[0]), 0);
  count[n] := 1;
  count[0] := 1;
  count[1] := n - 2;
  //Khởi tạo hàng đợi chứa các đỉnh tràn
  Queue.front := 0;
  Queue.rear := -1;
  Queue.nItems := 0; //Hàng đợi rỗng
  for v := 1 to n do //Duyệt tập đỉnh
    if (v <> s) and (v <> t) and (excess[v] > 0) then
    //v tràn
```

```
      PushToQueue(v); //đẩy v vào hàng đợi
end;
procedure Push(i: Integer); //Phép đẩy luồng theo cung e[i]
var Delta: Integer;
begin
  with e[i] do
    if excess[x] < c - f then Delta := excess[x]
    else Delta := c - f;
  Inc(e[i].f, Delta);
  Dec(e[-i].f, Delta);
  with e[i] do
    begin
      Dec(excess[x], Delta);
      Inc(excess[y], Delta);
    end;
end;
procedure SetH(u: Integer; NewH: Integer);
//Đặt độ cao của u thành NewH, đồng bộ hóa mảng count
begin
  Dec(count[h[u]]);
  h[u] := NewH;
  Inc(count[NewH]);
end;
procedure PerformGapHeuristic(gap: Integer);
//Đẩy nhãn theo khe gap
var v: Integer;
begin
  if (0 < gap) and (gap < n) and (count[gap] = 0) then
   //gap đúng là khe thật
    for v := 1 to n do
      if (v <> s) and (gap < h[v]) and (h[v] <= n) then
        begin
          SetH(v, n + 1);
          current[v] := head[v];
          //Nâng độ cao của v cần phải cập nhật lại con trỏ current[v]
        end;
end;
procedure Lift(u: Integer); //Phép nâng đỉnh u
var minH, OldH, i: Integer;
```

```
begin
  minH := 2 * maxN;
  i := head[u];
  while i <> 0 do //Duyệt các cung đi ra khỏi u
    begin
      with e[i] do
        if (c > f) and (h[y] < minH) then
        //Gặp cung thặng dư (u, v), ghi nhận đỉnh v thấp nhất
          minH := h[y];
      i := link[i];
    end;
  OldH := h[u]; //Nhớ lại h[u] cũ
  SetH(u, minH + 1); //nâng cao đỉnh u
  PerformGapHeuristic(OldH); //Có thể tạo ra khe OldH, đẩy nhãn theo khe
end;
procedure FIFOPreflowPush; //Thuật toán FIFO Preflow-Push
var
  NeedQueue: Boolean;
  z: Integer;
begin
  while Queue.nItems > 0 do //Chừng nào hàng đợi vẫn còn đỉnh tràn
    begin
      z := PopFromQueue; //Lấy một đỉnh tràn x khỏi hàng đợi
      while current[z] <> 0 do //Xét một cung đi ra khỏi x
        begin
          with e[current[z]] do
            begin
              if (c > f) and (h[x] > h[y]) then
              //Nếu có thể đẩy luồng được theo cung (u, v)
                begin
                  NeedQueue := (y <> s) and (y <> t)
                                 and (excess[y] = 0);
                  Push(current[z]); //Đẩy luồng luôn
                  if NeedQueue then
                  //v đang không tràn sau phép đẩy trở thành tràn
                    PushToQueue(y); //Đẩy v vào hàng đợi
                  if excess[z] = 0 then Break;
                  //x hết tràn thì chuyển qua xét đỉnh khác ngay
                end;
```

```
            end;
          current[z] := link[current[z]];
          //x chưa hết tràn thì chuyển sang xét cung liên thuộc tiếp theo
        end;
      if excess[z] > 0 then //Duyệt hết danh sách liên thuộc mà x vẫn tràn
        begin
          Lift(z); //Nâng cao x
          current[z] := head[z];
          //Đặt con trỏ current[x] trỏ lại về đầu danh sách liên thuộc
          PushToQueue(z); //Đẩy lại x vào hàng đợi chờ xử lý sau
        end;
    end;
  FlowValue := excess[t];
  //Thuật toán kết thúc, giá trị luồng bằng tổng luồng đi vào đỉnh thu (= - excess[s])
end;
procedure PrintResult; //In kết quả
var i: Integer;
begin
  WriteLn('Maximum flow: ');
  for i := 1 to m do
    with e[i] do
      if f > 0 then //Chỉ cần in ra các cung có luồng > 0
        WriteLn('e[', i, '] = (', x, ', ', y, '): c
                              = ', c, ', f = ', f);
  WriteLn('Value of flow: ', FlowValue);
end;
begin
  Enter; //Nhập dữ liệu
  Init; //Khởi tạo
  FIFOPreflowPush; //Thực hiện thuật toán đẩy tiền luồng
  PrintResult; //In kết quả
end.
```

***Định lý 3-19 (định lý về tính nguyên)***

Nếu tất cả các sức chứa là số nguyên thì thuật toán Ford-Fulkerson cũng như thuật toán đẩy tiền luồng luôn tìm được luồng cực đại với luồng trên cung là các số nguyên.

***Chứng minh***

Đối với thuật toán Ford-Fulkerson, ban đầu ta khởi tạo luồng 0 thì luồng trên các cung là nguyên. Mỗi lần tăng luồng dọc theo đường tăng luồng $P$, luồng trên mỗi cung hoặc giữ nguyên, hoặc tăng/giảm một lượng $\Delta_P$ cũng là số nguyên. Vậy nên cuối cùng luồng cực đại phải có giá trị nguyên trên tất cả các cung.

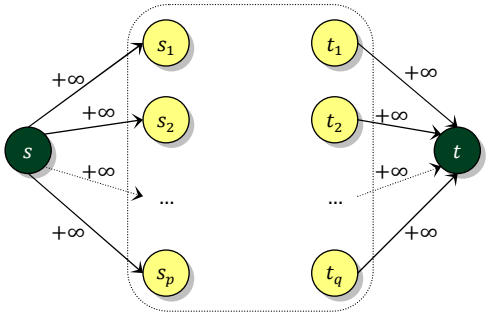
Đối với thuật toán đẩy tiền luồng, ban đầu ta khởi tạo một tiền luồng trên các cung là số nguyên. Phép $Lift$ và $Push$ không làm thay đổi tính nguyên của tiền luồng trên các cung. Vậy nên khi thuật toán kết thúc, tiền luồng trở thành luồng cực đại với giá trị luồng trên các cung là số nguyên.

## *3.4. Một số mở rộng và ứng dụng của luồng*

### a) Mạng với nhiều đỉnh phát và nhiều đỉnh thu

Ta mở rộng khái niệm mạng bằng cách cho phép mạng $G$ có $p$ đỉnh phát: $s_1, s_2, \dots, s_p$ và $q$ đỉnh thu $t_1, t_2, \dots, t_q$, các đỉnh phát và các đỉnh thu hoàn toàn phân biệt. Hàm sức chứa và luồng trên mạng được định nghĩa tương tự như trong trường hợp mạng có một đỉnh phát và một đỉnh thu. Giá trị của luồng được định nghĩa bằng tổng luồng trên các cung đi ra khỏi các đỉnh phát. Bài toán đặt ra là tìm luồng cực đại trên mạng có nhiều đỉnh phát và nhiều đỉnh thu.

Thêm vào mạng hai đỉnh: một siêu đỉnh phát $s$ và siêu đỉnh thu $t$. Thêm các cung nối từ $s$ tới các đỉnh $s_i$ có sức chứa $+\infty$, thêm các cung nối từ các đỉnh $t_j$ tới $t$ với sức chứa $+\infty$. Ta được một mạng mới $G' = (V, E')$ (h.2.9).




***Hình 2.9. Mạng với nhiều đỉnh phát và nhiều đỉnh thu***

Có thể thấy rằng nếu $f$ là một luồng cực đại trên $G'$, thì $f$ hạn chế trên $G$ cũng là luồng cực đại trên $G$. Vậy để tìm luồng cực đại trên $G$, ta sẽ tìm luồng cực đại

trên $G'$ rồi loại bỏ siêu đỉnh phát $s$, siêu đỉnh thu $t$ và tất cả những cung giả mới thêm vào.

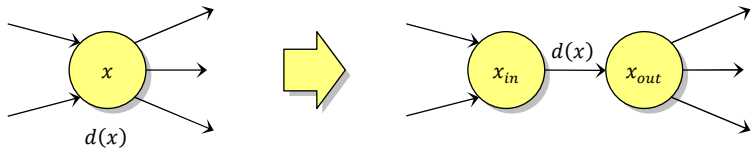
Một cách khác có thể thực hiện để tìm luồng trên mạng có nhiều đỉnh phát và nhiều đỉnh thu là loại bỏ tất các các cung đi vào các đỉnh phát cũng như các cung đi ra khỏi các đỉnh thu. Chập tất cả các đỉnh phát thành một siêu đỉnh $s$ và chập tất cả các đỉnh thu lại thành một siêu đỉnh thu $t$, mạng không còn các đỉnh $s_1, s_2, .., s_p$ và $t_1, t_2, \dots, t_q$ nữa mà chỉ có thêm đỉnh phát $s$ và đỉnh thu $t$. Trên mạng ban đầu, mỗi cung đi vào/ra $s_i$ được chỉnh lại đầu mút để nó đi vào/ra đỉnh $s$, mỗi cung đi vào/ra $t_j$ cũng được chỉnh lại đầu mút để nó đi vào/ra đỉnh $t$, ta được một mạng mới $G''$.

Khi đó ta ta có thể tìm $f$ là một luồng cực đại trên $G''$ và khôi phục lại đầu mút của các cung như cũ để $f$ trở thành luồng cực đại trên $G$.

### b) Mạng với sức chứa trên cả các đỉnh và các cung

Cho mạng $G = (V, E, c, s, t)$, mỗi đỉnh $v \in V - \{s, t\}$ được gán một số không âm $d(v)$ gọi là sức chứa của đỉnh đó. Luồng dương $\varphi$ trên mạng này được định nghĩa với tất cả các ràng buộc của luồng dương và thêm một điều kiện: Tổng luồng dương trên các cung đi vào mỗi đỉnh $v \in V - \{s, t\}$ không được vượt quá $d(v)$: $\sum_{e \in E^-(v)} \varphi(e) \leq d(v)$. Bài toán đặt ra là tìm luồng dương cực đại trên mạng có ràng buộc sức chứa trên cả các đỉnh và các cung.

Tách mỗi đỉnh $x \in V - \{s, t\}$ thành 2 đỉnh mới $x_{in}, x_{out}$ và một cung $(x_{in}, x_{out})$ với sức chứa $d(x)$. Các cung đi vào $x$ được chỉnh lại đầu mút để đi vào $x_{in}$ và các cung đi ra khỏi $x$ được chỉnh lại đầu mút để đi ra khỏi $x_{out}$ (h.2.10). Ta xây dựng được mạng $G' = (V', E')$ với đỉnh phát $s$ và đỉnh thu $t$.




***Hình 2.10. Tách đỉnh***

Khi đó việc tìm luồng dương cực đại trên mạng $G$ có thể thực hiện bằng cách tìm luồng dương cực đại trên mạng $G'$, sau đó chập tất cả các cặp $(x_{in}, x_{out})$ trở lại thành đỉnh $x$ ($\forall x \in V - \{s,t\}$) để khôi phục lại mạng $G$ ban đầu.

### c) Mạng với ràng buộc luồng dương bị chặn hai phía

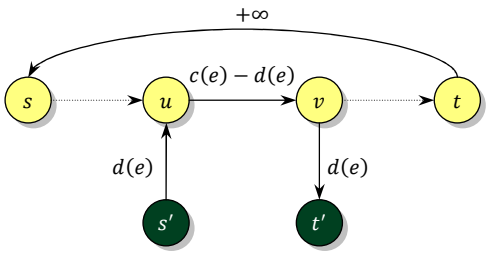
Cho mạng $G = (V, E, c, s, t)$ trong đó mỗi cung $e \in E$ ngoài sức chứa (lưu lượng) tối đa $c(e)$ còn được gán một số không âm $d(e) \leq c(e)$ gọi là lưu lượng tối thiểu. Một *luồng dương tương thích* $\varphi$ trên $G$ được định nghĩa với tất cả các ràng buộc của luồng dương và thêm một điều kiện: Luồng dương trên mỗi cung $e \in E$ không được nhỏ hơn sức chứa tối thiểu của cung đó:

$$d(e) \leq \varphi(e) \leq c(e)$$

Bài toán đặt ra là kiểm chứng sự tồn tại của luồng dương tương thích trên mạng với ràng buộc luồng dương bị chặn hai phía.

Xây dựng một mạng $G' = (V', E')$ từ mạng $G$ theo quy tắc:

- Tập đỉnh $V'$ có được từ tập $V$ thêm vào đỉnh phát giả $s'$ và đỉnh thu giả $t'$: $V' = V + \{s', t'\}$.
- Mỗi cung $e = (u, v) \in E$ sẽ tương ứng với ba cung trên $E'$: cung $e_1 = (u, v)$ có sức chứa $c(e) - d(e)$, cung $e_2 = (s', v)$ và cung $e_3 = (u, t')$ có sức chứa $d(e)$. Ngoài ra thêm vào cung $(t, s) \in E'$ với sức chứa $+\infty$




***Hình 2.11.***

Gọi $D = \sum_{e \in E} d(e)$ là tổng sức chứa tối thiểu của các cung trên mạng $G$. Khi đó trên mạng $G'$, tổng sức chứa các cung đi ra khỏi $s'$ cũng như tổng sức chứa các cung đi vào $t'$ bằng $D$. Vì vậy với mọi luồng dương trên $G'$ thì giá trị luồng đó không thể vượt quá $D$.

Từ đó suy ra rằng nếu tồn tại một luồng dương $\varphi'$ trên $G'$ có giá trị luồng $|\varphi'| = D$ thì $\varphi'$ bắt buộc là luồng dương cực đại trên $G'$.

Bổ đề 3-20 cho phép ta kiểm chứng sự tồn tại luồng dương tương thích trên $G$ bằng việc đo giá trị luồng cực đại trên $G'$.

***Bổ đề 3-20***

Điều kiện cần và đủ để tồn tại luồng dương tương thích $\varphi$ trên mạng $G$ là tồn tại luồng dương cực đại $\varphi'$ trên $G'$ với giá trị luồng $|\varphi'| = D$.

***Chứng minh***

Giả sử luồng dương cực đại $\varphi'$ trên $G'$ có $|\varphi| = D = \sum_{e \in E} d(e)$. Ta xây dựng luồng $\varphi$ trên $G$ bằng cách cộng thêm vào luồng $\varphi'$ trên mỗi cung $e$ một lượng $d(e)$:

$$\varphi: E \to [0, +\infty)$$
$$e \to \varphi(e) = \varphi'(e) + d(e)$$

Khi đó có thể dễ dàng kiểm chứng được $\varphi$ thỏa mãn tất cả các ràng buộc của luồng dương tương thích trên mạng $G$.

Ngược lại nếu $\varphi$ là một luồng dương tương thích trên $G$. Ta xây dựng luồng dương $\varphi'$ trên $G'$ bằng cách trừ luồng $\varphi$ trên mỗi cung $e$ đi một lượng $d(e)$, đồng thời đặt luồng $\varphi'$ trên các cung đi ra khỏi $s'$ cũng như trên các cung đi vào $t'$ đúng bằng sức chứa của cung đó. Khi đó cũng dễ dàng kiểm chứng được $\varphi'$ là luồng dương cực đại và $|\varphi| = D$.

## d) Mạng với sức chứa âm

Cho mạng $G = (V, E, c, w, s, t)$ trong đó ta mở rộng khái niệm sức chứa bằng cách cho phép cả những sức chứa âm trên một số cung. Khái niệm luồng được định nghĩa như bình thường.

Nếu như có thể khởi tạo được một luồng thì thuật toán Ford-Fulkerson vẫn hoạt động đúng để tìm luồng cực đại trên mạng có sức chứa âm. Vấn đề khởi tạo một luồng bất kỳ trên mạng không phải đơn giản vì chúng ta không thể khởi tạo bằng luồng 0, bởi nếu như vậy, ràng buộc sức chứa tối đa sẽ bị vi phạm trên các cung có sức chứa âm.

Giả sử một cung $e \in E$ có sức chứa $c(e) < 0$. Theo tính đối xứng lệch của luồng $f(e) = -f(-e)$ và ràng buộc sức chứa tối đa $f(e) \leq c(e)$, ta có:

$$f(-e) = -f(e) \geq -c(e) > 0 \qquad (3.5)$$

Như vậy ràng buộc sức chứa tối đa $f(e) \leq c(e)$ tương đương với ràng buộc về sức chứa tối thiểu $-c(e)$ trên cung đối $-e$. Việc chỉ ra một luồng bất kỳ trên $G$ có thể thực hiện bằng cách tìm luồng dương trên mạng với ràng buộc luồng dương bị chặn hai phía sau đó biến đổi luồng dương này thành luồng cần tìm.

### e) Lát cắt hẹp nhất

Ta quan tâm tới đồ thị vô hướng liên thông $G = (V, E)$ với hàm trọng số (hay lưu lượng) $c: E \to [0, +\infty)$. Giả sử $|V| \geq 2$, người ta muốn bỏ đi một số cạnh để đồ thị mất tính liên thông và yêu cầu tìm phương án sao cho tổng trọng số các cạnh bị loại bỏ là nhỏ nhất.

Bài toán cũng có thể phát biểu dưới dạng: hãy phân hoạch tập đỉnh $V$ thành hai tập khác rỗng rời nhau $X$ và $Y$ sao cho tổng lưu lượng các cạnh nối giữa $X$ và $Y$ là nhỏ nhất có thể. Cách phân hoạch này gọi là lát cắt tổng quát hẹp nhất của $G$, ký hiệu $MinCut(G)$.

$$c(X, Y) \to \min$$

$$X \neq \emptyset; Y \neq \emptyset; X \cap Y = \emptyset; X \cup Y = V;$$

Một cách tệ nhất có thể thực hiện là thử tất cả các cặp đỉnh $s, t$. Với mỗi lần thử ta cho $s$ làm đỉnh phát và $t$ làm đỉnh thu trên mạng $G$, sau đó tìm luồng cực đại và lát cắt $s - t$ hẹp nhất. Cuối cùng là chọn lát cắt $s - t$ có lưu lượng nhỏ nhất trong tất cả các lần thử. Phương pháp này cần $\binom{n}{2} = \frac{n \times (n-1)}{2}$ lần tìm luồng cực đại, có tốc độ chậm và không khả thi với dữ liệu lớn.

***Bổ đề 3-21***

Với $s$ và $t$ là hai đỉnh bất kỳ. Từ đồ thị $G$, ta xây dựng đồ thị $G_{st}$ bằng cách chập hai đỉnh $s$ và $t$ thành một đỉnh duy nhất, ký hiệu $st$, các cạnh nối $s$ với $t$ bị hủy bỏ, các cạnh liên thuộc với chỉ $s$ hoặc $t$ được chỉnh lại đầu mút để trở thành cạnh liên thuộc với $st$. Khi đó $MinCut(G)$ có thể thu được bằng lấy lát cắt có lưu lượng nhỏ nhất trong hai lát cắt:

- Lát cắt $s-t$ hẹp nhất: Coi $s$ là đỉnh phát và $t$ là đỉnh thu, lát cắt $s-t$ hẹp nhất có thể xác định bằng việc giải quyết bài toán luồng cực đại trên mạng $G$.
- Lát cắt tổng quát hẹp nhất trên $G_{st}$: $MinCut(G_{st})$.

> ***Chứng minh***
>
> Xét lát cắt tổng quát hẹp nhất trên $G$ có thể đưa $s$ và $t$ vào hai thành phần liên thông khác nhau hoặc đưa chúng vào cùng một thành phần liên thông. Trong trường hợp thứ nhất, $MinCut(G)$ là lát cắt $s-t$ hẹp nhất. Trong trường hợp thứ hai, $MinCut(G)$ là $MinCut(G_{st})$.

Bổ đề 3-21 cho phép chúng ta xây dựng một thuật toán tốt hơn: Nếu đồ thị chỉ gồm 2 đỉnh thì chỉ việc cắt rời hai đỉnh vào hai tập. Nếu không, ta chọn hai đỉnh bất kỳ $s,t$ làm đỉnh phát và đỉnh thu, tìm luồng cực đại và ghi nhận lát cắt $s-t$ hẹp nhất. Tiếp theo ta chập hai đỉnh $s,t$ thành một đỉnh $st$ và lặp lại với đồ thị $G_{st}$… Cuối cùng là chỉ ra lát cắt $s-t$ hẹp nhất trong số tất cả các lát cắt được ghi nhận. Phương pháp này đòi hỏi phải thực hiện $|V|-1$ lần tìm luồng cực đại, tuy đã có sự cải thiện về tốc độ nhưng chưa phải thật tốt.

Nhận xét rằng tại mỗi bước của cách giải trên, chúng ta có thể chọn hai đỉnh $s,t$ bất kỳ miễn sao $s \neq t$. Vì vậy người ta muốn tìm một cách chọn cặp đỉnh $s,t$ một cách hợp lý tại mỗi bước để có thể chỉ ra ngay lát cắt $s-t$ hẹp nhất mà không cần tìm luồng cực đại. Thuật toán dưới đây [**37**] là một trong những thuật toán hiệu quả dựa trên ý tưởng đó.

Với $A$ là một tập con của tập đỉnh $V$ và $x$ là một đỉnh không thuộc $A$. Định nghĩa *lực hút* của $A$ đối với $x$ là tổng trọng số các cạnh nối $x$ với các đỉnh thuộc $A$:

$$c(A,\{x\}) = \sum_{\substack{e=(x,y)\in E \\ y\in A}} c(e)$$

***Bổ đề 3-22***

Bắt đầu từ tập $A$ chỉ gồm một đỉnh bất kỳ $a \in V$, ta cứ tìm một đỉnh bị $A$ hút chặt nhất kết nạp thêm vào $A$ cho tới khi $A = V$. Gọi $s$ và $t$ là hai đỉnh được kết nạp cuối cùng theo cách này. Khi đó lát cắt $(V-\{t\},\{t\})$ là lát cắt $s-t$ hẹp nhất.

***Chứng minh***

Xét một lát cắt $s-t$ bất kỳ $\kappa$, ta sẽ chứng minh rằng lưu lượng của lát cắt $(V-\{t\},\{t\})$ không lớn hơn lưu lượng của lát cắt $\kappa$.

Một đỉnh $v$ được gọi là *đỉnh hoạt tính* nếu $v$ và đỉnh được đưa vào $A$ liền trước $v$ bị rơi vào hai phía của lát cắt $\kappa$. Gọi $A_v$ là tập các đỉnh được kết nạp vào $A$ trước đỉnh $v$, $\kappa_v$ là lát cắt $\kappa$ hạn chế trên $A_v \cup \{v\}$ (Lát cắt $\kappa_v$ dùng đúng cách phân hoạch của lát cắt $\kappa$ nhưng chỉ quan tâm tới tập đỉnh $A_v \cup \{v\}$). Gọi $c(\kappa)$ là lưu lượng của lát cắt $\kappa$, $c(\kappa_v)$ là lưu lượng của lát cắt $\kappa_v$.

Trước hết ta sử dụng phép quy nạp để chỉ ra rằng nếu $u$ là đỉnh hoạt tính thì:

$$c(A_u,\{u\}) \le c(\kappa_u) \tag{3.6}$$

Nếu $u$ là đỉnh hoạt tính đầu tiên được kết nạp vào $A$, lát cắt $\kappa_u$ sẽ chia tập $A_u \cup \{u\}$ làm hai tập, một tập là $A_u$ và một tập là $\{u\}$, khi đó ta có $c(A_u,\{u\})$ cũng chính là $c(\kappa_u)$. Giả thiết rằng bất đẳng thức (3.6) đúng với đỉnh hoạt tính $u$, ta sẽ chứng nó cũng đúng với những đỉnh hoạt tính $v$ được kết nạp vào $A$ sau $u$. Thật vậy,

$$c(A_v,\{v\}) = c(A_u,\{v\}) + c(A_v - A_u,\{v\}) \tag{3.7}$$

Do $A_u$ phải hút $u$ mạnh hơn $v$, kết hợp với giả thiết quy nạp, ta có:

$$c(A_u,\{v\}) \le c(A_u,\{u\}) \le c(\kappa_u)$$

Hạng tử $c(A_v - A_u,\{v\})$ là tổng trọng số các cạnh nối giữa $v$ và $A_v - A_u$. Do $u$ và $v$ là hai đỉnh hoạt tính liên tiếp, các cạnh này sẽ nối giữa hai phía của lát cắt $\kappa_v$ và có đóng góp trong phép tính $c(\kappa_v)$, mặt khác do $v \notin A_u \cup \{u\}$ nên những cạnh này không đóng góp trong phép tính $c(\kappa_u)$. Vậy từ công thức (3.7), ta suy ra:

$$\begin{aligned} c(A_v,\{v\}) &= c(A_u,\{v\}) + c(A_v - A_u,\{v\}) \\ &\le c(\kappa_u) + c(A_v - A_u,\{v\}) \\ &\le c(\kappa_v) \end{aligned} \tag{3.8}$$

Vì $\kappa$ là một lát cắt $s-t$ nên chắc chắn $s$ và $t$ nằm ở hai phía khác nhau của lát cắt $\kappa$, hay nói cách khác, $t$ là đỉnh hoạt tính. Bất đẳng thức (3.6) chứng minh ở trên cho ta kết quả:

$$\begin{aligned} c(V-\{t\},\{t\}) &= c(A_t,\{t\}) \\ &\le c(\kappa_t) \\ &= c(\kappa) \end{aligned} \tag{3.9}$$

Ta chứng minh được lát cắt $(V-\{t\},\{t\})$ là lát cắt $s-t$ hẹp nhất.

***Định lý 3-23***

Việc tìm lát cắt tổng quát trên đồ thị vô hướng liên thông với hàm trọng số không âm có thể được thực hiện bằng thuật toán trong thời gian $O(|V|^2 \log|V| + |V||E|)$.

***Chứng minh***

Bắt đầu từ tập $A$ chỉ gồm một đỉnh bất kỳ, ta mở rộng $A$ bằng cách lần lượt kết nạp vào $A$ đỉnh bị hút chặt nhất cho tới khi $A = V$. Việc này được thực hiện với kỹ thuật tương tự như thuật toán Prim: với $\forall v \notin A$, ta ký hiệu nhãn $d[v]$ là lực hút của $A$ đối với đỉnh $v$. khi $A$ được kết nạp thêm một $u$ thì các nhãn lực hút của những đỉnh $v$ khác sẽ được cập nhật lại theo công thức:

$$d[v]_{\text{mới}} := d[v]_{\text{cũ}} + c(u,v), \forall (u,v) \in E$$

Bằng việc tổ chức các đỉnh ngoài $A$ trong một hàng đợi ưu tiên dạng Fibonacci Heap, việc mở rộng tập $A$ cho tới khi $A = V$ được thực hiện trong thời gian $O(|V|\log|V| + |E|)$. Trong quá trình đó, $s$ và $t$ là hai đỉnh cuối cùng được kết nạp vào $A$ cũng được xác định và $MinCut(G)$ được cập nhật theo lát cắt $s - t$ hẹp nhất. Sau đó hai đỉnh $s, t$ được chập vào và thuật toán lặp lại với đồ thị $G_{st}$. Tổng cộng ta có $|V| - 1$ lần lặp, suy ra lát cắt tổng quát hẹp nhất có thể tìm được trong thời gian $O(|V|^2 \log|V| + |V||E|)$.

Mặc dù tính đúng đắng của thuật toán được chứng minh dựa vào lý thuyết về luồng cực đại và lát cắt hẹp nhất, việc cài đặt thuật toán lại khá đơn giản và không động chạm gì đến luồng cực đại.

## Bài tập

**2.27.** Cho $f_1$ và $f_2$ là hai luồng trên mạng $G = (V, E, c, s, t)$ và $\alpha$ là một số thực nằm trong đoạn $[0,1]$. Xét ánh xạ:

$$f_\alpha : E \to \mathbb{R}$$
$$e \mapsto f_\alpha(e) = \alpha f_1(e) + (1-\alpha) f_2(e)$$

Chứng minh rằng $f_\alpha$ cũng là một luồng trên mạng $G$ với giá trị luồng:

$$|f_\alpha| = \alpha|f_1| + (1-\alpha)|f_2|$$

**2.28.** Cho $f$ là một luồng trên mạng $G = (V, E, c, s, t)$, chứng minh rằng với $\forall e \in E$, ta có:

$$c_f(e) + c_f(-e) = c(e) + c(-e)$$

**2.29.** Cho $f$ là luồng cực đại trên mạng $G = (V, E, c, s, t)$, gọi $Y$ là tập các đỉnh đến được $t$ bằng một đường thặng dư trên $G_f$ và $X = V - Y$. Chứng minh rằng $(X, Y)$ là lát cắt $s - t$ hẹp nhất của mạng $G$.

**2.30.** Viết chương trình nhận vào một đồ thị có hướng $G = (V, E)$ với hai đỉnh phân biệt $s$ và $t$ và tìm một tập gồm nhiều đường đi nhất từ $s$ tới $t$ sao cho các đường đi trong tập này đôi một không có cạnh chung.

*Gợi ý*

Coi $s$ là đỉnh phát và $t$ là đỉnh thu, các cung đều có sức chứa 1. Tìm luồng cực đại trên mạng bằng thuật toán Ford-Fulkerson, theo Định lý 3-19 (định lý về tính nguyên), luồng trên các cung chỉ có thể là 0 hoặc 1. Loại bỏ các cung có luồng 0 và chỉ giữ lại các cung có luồng 1. Tiếp theo ta tìm một đường đi từ $s$ tới $t$, chọn đường đi này vào tập hợp, loại bỏ tất cả các cung dọc trên đường đi này khỏi đồ thị và lặp lại…, thuật toán sẽ kết thúc khi đồ thị không còn cạnh nào (không còn đường đi từ $s$ tới $t$).

Về kỹ thuật cài đặt, ta có thể tìm một đường đi từ $s$ tới $t$ trên đồ thị $G$, đảo chiều tất cả các cung trên đường đi này và lặp lại cho tới khi không còn đường đi từ $s$ tới $t$ nữa. Có thể thấy rằng đồ thị $G$ tại mỗi bước chính là đồ thị các cung thặng dư và đường đi tìm được ở mỗi bước chính là đường tăng luồng.

Đồ thị $G$ giờ đây không còn đường đi từ $s$ tới $t$, ta tìm một đường đi từ $t$ về $s$, kết nạp đường đi theo chiều ngược lại (từ $s$ tới $t$) vào tập hợp, xóa bỏ tất cả các cung trên đường đi và cứ tiếp tục như vậy cho tới khi không còn đường đi từ $t$ về $s$ nữa.

**2.31.** Tương tự như Bài tập 2.29 nhưng yêu cầu thực hiện trên đồ thị vô hướng.

**2.32.** (Hệ đại diện phân biệt) Một lớp học có $n$ bạn nam và $n$ bạn nữ. Nhân ngày 8/3, lớp có mua $m$ món quà để các bạn nam tặng các bạn nữ. Mỗi món quà có thể thuộc sở thích của một số bạn trong lớp.

Hãy lập chương trình tìm cách phân công tặng quà thỏa mãn:

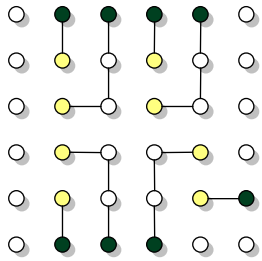
- Mỗi bạn nam phải tặng quà cho đúng một bạn nữ và mỗi bạn nữ phải nhận quà của đúng một bạn nam. Món quà được tặng phải thuộc sở thích của cả hai người.
- Món quà nào đã được một bạn nam chọn để tặng thì bạn nam khác không được chọn nữa.

*Gợi ý*: Xây dựng một mạng trong đó tập đỉnh $V$ gồm 3 lớp đỉnh $S$, $X$ và $T$:

- Lớp đỉnh phát $S = \{s_1, s_2, \dots, s_n\}$, mỗi đỉnh tương ứng với một bạn nam.
- Lớp đỉnh $X = (x_1, x_2, \dots, x_n)$ mỗi đỉnh tương ứng với một món quà.
- Lớp đỉnh thu $T = \{t_1, t_2, \dots, t_n\}$ mỗi đỉnh tương ứng với một bạn nữ.

Nếu bạn nam $i$ thích món quà $k$, ta cho cung nối từ $s_i$ tới $x_k$, nếu bạn nữ $j$ thích món quà $k$, ta cho cung nối từ $x_k$ tới $t_j$. Sức chứa của các cung đặt bằng 1 và sức chứa của các đỉnh $v_1, v_2, \dots, v_n$ cũng đặt bằng 1. Tìm luồng nguyên cực đại trên mạng $G$ có $n$ đỉnh phát, $n$ đỉnh thu, đồng thời có cả ràng buộc sức chứa trên các đỉnh, những cung có luồng 1 sẽ nối giữa một món quà và người tặng/nhận tương ứng.

**2.33.** Cho mạng điện gồm $m \times n$ điểm nằm trên một lưới $m$ hàng, $n$ cột. Một số điểm nằm trên biên của lưới là nguồn điện, một số điểm trên lưới là các thiết bị sử dụng điện. Người ta chỉ cho phép nối dây điện giữa hai điểm nằm cùng hàng hoặc cùng cột. Hãy tìm cách đặt các dây điện nối các thiết bị sử dụng điện với nguồn điện sao cho hai đường dây bất kỳ nối hai thiết bị sử dụng điện với nguồn điện tương ứng của chúng không được có điểm chung.



**2.34.** (Kỹ thuật giãn sức chứa) Cho mạng $G = (V, E, c, w, s, t)$ với sức chứa nguyên: $c: E \to \mathbb{N}$. Gọi $C := \max_{e \in E} c(e)$.

a) Chứng minh rằng lát cắt $s - t$ hẹp nhất của $G$ có lưu lượng không vượt quá $C|E|$

b) Với một số nguyên $k$, tìm thuật toán xác định đường tăng luồng có giá trị thặng dư $\geq k$ trong thời gian $O(|E|)$.

c) Chứng minh rằng thuật toán sau đây tìm được luồng cực đại trên mạng $G$:

```
procedure MaxFlowByScaling;
begin
  f := «Luồng 0»;
  k := C; //k là sức chứa lớn nhất của một cung trong E
  while k ≥ 1 do
    begin
      while «Tìm được đường tăng luồng P
             có giá trị thặng dư ≥ k» do
        «Tăng luồng dọc theo đường P»;
      k := k div 2;
    end;
end;
```

d) Chứng minh rằng khi bước vào mỗi lượt lặp của vòng lặp:

```
while k ≥ 1 do...
```

Lưu lượng của lát cắt hẹp nhất trên mạng thặng dư $G_f$ không vượt quá $2k|E|$.

e) Chứng minh rằng trong mỗi lượt lặp của vòng lặp:

```
while k ≥ 1 do...
```

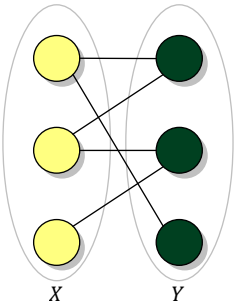
Vòng lặp while bên trong thực hiện $O(E)$ lần với mỗi giá trị của $k$.

f) Chứng minh rằng thuật toán trên (*maximum flow by scaling*) có thể cài đặt để tìm luồng cực đại trên $G$ trong thời gian $O(|E|^2 \log C)$.

## 4. Bộ ghép cực đại trên đồ thị hai phía

### *4.1. Đồ thị hai phía*

Đồ thị vô hướng $G = (V, E)$ được gọi là đồ thị hai phía nếu tập đỉnh $V$ của nó có thể chia làm hai tập con rời nhau: $X$ và $Y$ sao cho mọi cạnh của đồ thị đều nối một đỉnh thuộc $X$ với một đỉnh thuộc $Y$. Khi đó người ta còn ký hiệu $G = (X \cup Y, E)$. Để thuận tiên trong trình bày, ta gọi các đỉnh thuộc $X$ là các $X$_đỉnh và các đỉnh thuộc $Y$ là các $Y$_đỉnh.




***Hình 2.12. Đồ thị hai phía***

Một đồ thị vô hướng là đồ thị hai phía nếu và chỉ nếu từng thành phần liên thông của nó là đồ thị hai phía. Để kiểm tra một đồ thị vô hướng liên thông có phải đồ thị hai phía hay không, ta có thể sử dụng một thuật toán tìm kiếm trên đồ thị (BFS hoặc DFS) bắt đầu từ một đỉnh $s$ bất kỳ. Đặt:

$$X := \{\text{tập các đỉnh đến được từ } s \text{ qua một số chẵn cạnh}\}$$
$$Y := \{\text{tập các đỉnh đến được từ } s \text{ qua một số lẻ cạnh}\}$$

Nếu tồn tại cạnh của đồ thị nối hai đỉnh $\in X$ hoặc hai đỉnh $\in Y$ thì đồ thị đã cho không phải đồ thị hai phía, ngược lại đồ thị đã cho là đồ thị hai phía với cách phân hoạch tập đỉnh thành hai tập $X, Y$ ở trên.

Đồ thị hai phía gặp rất nhiều mô hình trong thực tế. Chẳng hạn quan hệ hôn nhân giữa tập những người đàn ông và tập những người đàn bà, việc sinh viên chọn trường, thầy giáo chọn tiết dạy trong thời khoá biểu v.v…

## *4.2. Bài toán tìm bộ ghép cực đại trên đồ thị hai phía*

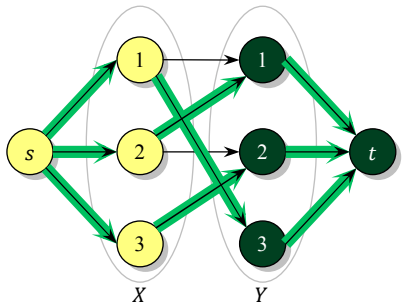
Cho đồ thị hai phía $G = (X \cup Y, E)$. Một *bộ ghép (matching)* của $G$ là một tập các cạnh đôi một không có đỉnh chung. Có thể coi một bộ ghép là một tập $M \subseteq E$ sao cho trên đồ thị $(X \cup Y, M)$, mỗi đỉnh có bậc không quá 1.

Vấn đề đặt ra là tìm *một bộ ghép lớn nhất (maximum matching)* (có nhiều cạnh nhất) trên đồ thị hai phía cho trước.

## *4.3. Mô hình luồng*

Định hướng các cạnh của $G$ thành cung từ $X$ sang $Y$. Thêm vào đỉnh phát giả $s$ và các cung nối từ $s$ tới các $X$_đỉnh, thêm vào đỉnh thu giả $t$ và các cung nối từ

các $Y$_đỉnh tới $t$. Sức chứa của tất cả các cung được đặt bằng 1, ta được mạng $G'$. Xét một luồng trên mạng $G'$ có luồng trên các cung là số nguyên, khi đó có thể thấy rằng những cung có luồng bằng 1 từ $X$ sang $Y$ sẽ tương ứng với một bộ ghép trên $G$. Bài toán tìm bộ ghép cực đại trên $G$ có thể giải quyết bằng cách tìm luồng nguyên cực đại trên $G'$.




***Hình 2.13. Mô hình luồng của bài toán tìm bộ ghép cực đại trên đồ thị hai phía.***

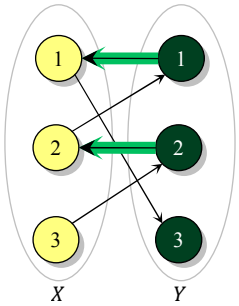
Chúng ta sẽ phân tích một số đặc điểm của đường tăng luồng trong trường hợp này để tìm ra một cách cài đặt đơn giản hơn.

Xét đồ thị hai phía $G = (X \cup Y, E)$ và một bộ ghép $M$ trên $G$.

- Những đỉnh thuộc $M$ gọi là những *đỉnh đã ghép* (*matched vertices*), những đỉnh không thuộc $M$ gọi là những *đỉnh chưa ghép* (*unmached vertices*).
- Những cạnh thuộc $M$ gọi là những cạnh đã ghép, những cạnh không thuộc $M$ được gọi là những cạnh chưa ghép.
- Nếu định hướng lại những cạnh của đồ thị thành cung: Những cạnh chưa ghép định hướng từ $X$ sang $Y$, những cạnh đã ghép định hướng ngược lại từ $Y$ về $X$. Trên đồ thị định hướng đó, một đường đi được gọi là *đường pha (alternating path)* và một đường đi từ một $X$_đỉnh chưa ghép tới một $Y$_đỉnh chưa ghép gọi là một *đường mở* (*augmenting path*).

Dọc trên một đường pha, các cạnh đã ghép và chưa ghép xen kẽ nhau. Đường mở cũng là một đường pha, đi qua một số lẻ cạnh, trong đó số cạnh chưa ghép nhiều hơn số cạnh đã ghép đúng một cạnh.

> Ví dụ với đồ thị hai phía trong hình 2.14 và một bộ ghép $\{(x_1, y_1), (x_2, y_2)\}$. Đường đi $\langle x_3, y_2, x_2, y_1 \rangle$ là một đường pha, đường đi $\langle x_3, y_2, x_2, y_1, x_1, y_3 \rangle$ là một đường mở.




*Hình 2.14. Đồ thị hai phía và các cạnh được định hướng theo một bộ ghép*

Đường mở thực chất là đường tăng luồng với giá trị thặng dư 1 trên mô hình luồng. Định lý 3-11 (mối quan hệ giữa luồng cực đại, đường tăng luồng và lát cắt hẹp nhất) đã chỉ ra rằng điều kiện cần và đủ để một bộ ghép $M$ là bộ ghép cực đại là không tồn tại đường mở ứng với $M$.

Nếu tồn tại đường mở $P$ ứng với bộ ghép $M$, ta mở rộng bộ ghép bằng cách: dọc trên đường $P$ loại bỏ những cạnh đã ghép khỏi $M$ và thêm những cạnh chưa ghép vào $M$. Bộ ghép mới thu được sẽ có lực lượng nhiều hơn bộ ghép cũ đúng một cạnh. Đây thực chất là phép tăng luồng dọc trên đường $P$ trên mô hình luồng.

### *4.4. Thuật toán đường mở*

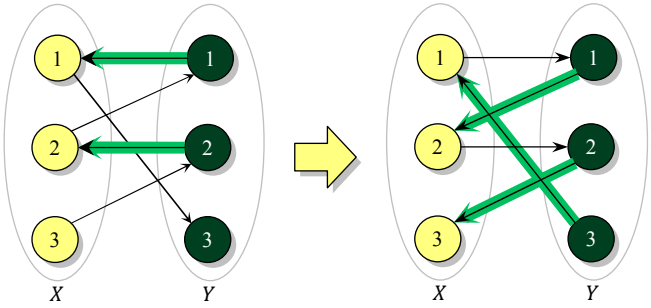
Từ mô hình luồng của bài toán, chúng ta có thể xây dựng được thuật toán tìm bộ ghép cực đại dựa trên cơ chế tìm đường mở và tăng cặp: Thuật toán khởi tạo một bộ ghép bất kỳ trước khi bước vào vòng lặp chính. Tại mỗi bước lặp, đường mở (thực chất là một đường đi từ một $X$_đỉnh chưa ghép tới một $Y$_đỉnh chưa ghép) được tìm bằng BFS hoặc DFS và bộ ghép sẽ được mở rộng dựa trên đường mở tìm được.

```
M := «Một bộ ghép bất kỳ, chẳng hạn: Ø»;
while «Tìm được đường mở P» do
  begin
    «Dọc trên đường P:
       - Loại bỏ những cạnh đã ghép khỏi M
       - Thêm những cạnh chưa ghép vào M
    »
  end;
```

Ví dụ với đồ thị trong hình 2.14 và bộ ghép $M = \{(x_1, y_1), (x_2, y_2)\}$, thuật toán sẽ tìm được đường mở:

$$x_3 \dashrightarrow y_2 \underbrace{\rightarrow}_{\in M} x_2 \dashrightarrow y_1 \underbrace{\rightarrow}_{\in M} x_1 \dashrightarrow y_3$$

Dọc trên đường mở này, ta loại bỏ hai cạnh $(y_2, x_2)$ và $(y_1, x_1)$ khỏi bộ ghép và thêm vào bộ ghép ba cạnh $(x_3, y_2)$, $(x_2, y_1)$, $(x_1, y_3)$, được bộ ghép mới 3 cạnh. Đồ thị với bộ ghép mới không còn đỉnh chưa ghép (không còn đường mở) nên đây chính là bộ ghép cực đại (h.2.15).




***Hình 2.15. Mở rộng bộ ghép***

## *4.5. Cài đặt*

Chúng ta sẽ cài đặt thuật toán tìm bộ ghép cực đại trên đồ thị hai phía $G = (X \cup Y, E)$, trong đó $|X| = p$, $|Y| = q$ và $|E| = m$. Các $X$_đỉnh được đánh số từ 1 tới $p$ và các $Y$_đỉnh được đánh số từ 1 tới $q$. Khuôn dạng Input/Output như sau:

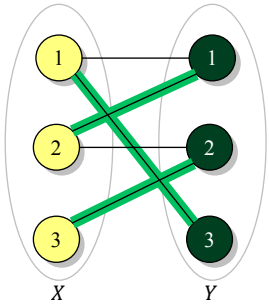
**Input**

- Dòng 1 chứa ba số nguyên dương $p, q, m$ lần lượt là số $X$_đỉnh, số $Y$_đỉnh và số cạnh của đồ thị hai phía. $(p, q \leq 10^4; m \leq 10^6)$.
- $m$ dòng tiếp theo, mỗi dòng chứa hai số nguyên dương $i, j$ tương ứng với một cạnh $(x_i, y_j)$ của đồ thị.

**Output**

Bộ ghép cực đại trên đồ thị.

| Sample Input | Sample Output |
|---|---|
| 3 3 5<br>3 2<br>2 2<br>2 1<br>1 3<br>1 1 | 1: x[2] - y[1]<br>2: x[3] - y[2]<br>3: x[1] - y[3] |




### a) Biểu diễn đồ thị hai phía và bộ ghép

Đồ thị hai phía $G = (X \cup Y, E)$ sẽ được biểu diễn bằng cách danh sách kề của các $X$_đỉnh. Cụ thể là ta sẽ sử dụng mảng $head[1 \dots p]$ với các phần tử ban đầu được khởi tạo bằng 0, mảng $adj[1 \dots m]$ và mảng $link[1 \dots m]$. Danh sách kề được xây dựng ngay trong quá trình đọc danh sách cạnh: mỗi khi đọc một cạnh $e_i = (x, y)$ ta gán $adj[i] := y$, đặt $link[i] := head[x]$ sau đó cập nhật lại $head[x] := i$. Khi đọc xong danh sách cạnh thì danh sách kề cũng được xây dựng xong, khi đó để duyệt các $Y$_đỉnh kề với một đỉnh $x \in X$, ta có thể sử dụng thuật toán sau:

```
i := head[x]; //Từ đầu danh sách móc nối các đỉnh kề x
while i ≠ 0 do
  begin
    «Xử lý đỉnh adj[i]»;
    i := link[i]; //Nhảy sang phần tử kế tiếp trong danh sách móc nối
  end;
```

Bộ ghép trên đồ thị hai phía được biểu diễn bởi mảng $match[1 \dots ny]$, trong đó $match[j]$ là chỉ số của $X$_đỉnh ghép với đỉnh $y_j$. Nếu $y_j$ là đỉnh chưa ghép, ta gán $match[j] := 0$.

### b) Tìm đường mở

Đường mở thực chất là một đường đi từ một $X$_đỉnh chưa ghép tới một $Y$_đỉnh chưa ghép trên đồ thị định hướng. Ta sẽ tìm đường mở tại mỗi bước bằng thuật toán DFS:

Bắt đầu từ một đỉnh $x \in X$ chưa ghép, trước hết ta đánh dấu các $Y$_đỉnh bằng mảng $avail[1 \dots q]$ trong đó $avail[j] =$ True nếu đỉnh $y_j \in Y$ chưa thăm và $avail[j] =$ False nếu đỉnh $y_j \in Y$ đã thăm (chỉ cần đánh dấu các $Y$_đỉnh).

Thuật toán DFS để tìm đường mở xuất phát từ $x$ được thực hiện bằng một thủ tục đệ quy $Visit(x)$, thủ tục này sẽ quét tất cả những đỉnh $y \in Y$ chưa thăm nối từ $X$ (dĩ nhiên qua một cạnh chưa ghép), với mỗi khi xét đến một đỉnh $y \in Y$, trước hết ta đánh dấu thăm $y$. Sau đó:

- Nếu $y$ đã ghép, dựa vào sự kiện từ $y$ chỉ đi đến được $match[y]$ qua một cạnh đã ghép hướng từ $Y$ về $X$, lời gọi đệ quy $Visit(match[y])$ được thực hiện để thăm luôn đỉnh $match[y] \in X$ (thăm liền hai bước).
- Ngược lại nếu $y$ chưa ghép, tức là thuật toán DFS tìm được đường mở kết thúc ở $y$, ta thoát khỏi dây chuyền đệ quy. Quá trình thoát dây chuyền đệ quy thực chất là lần ngược đường mở, ta sẽ lợi dụng quá trình này để mở rộng bộ ghép dựa trên đường mở.

Để thuật toán hoạt động hiệu quả hơn, ta sử dụng liên tiếp các pha xử lý lô: Ký hiệu $X^*$ là tập các $X$_đỉnh chưa ghép, mỗi pha sẽ cố gắng mở rộng bộ ghép dựa trên không chỉ một mà nhiều đường mở không có đỉnh chung xuất phát từ các đỉnh khác nhau thuộc $X^*$. Cụ thể là một pha sẽ khởi tạo mảng đánh dấu $avail[1 \dots q]$ bởi giá trị True, sau đó quét tất cả những đỉnh $x \in X^*$, thử tìm đường mở xuất phát từ $x$ và mở rộng bộ ghép nếu tìm ra đường mở. Trong một pha có thể có nhiều $X$_đỉnh được ghép thêm.

```
procedure Visit(x ∈X); //Thuật toán DFS
begin
  for ∀y: (x, y)∈E do //Quét các Y_đỉnh kề x
    if avail[y] then //y chưa thăm, chú ý (x, y) chắc chắn là cạnh chưa ghép
      begin
        avail[y] := False; //Đánh dấu thăm y
        if match[y] = 0 then Found := True
        //y chưa ghép, dựng cờ báo tìm thấy đường mở
        else Visit(match[y]); //y đã ghép, gọi đệ quy tiếp tục DFS
        if Found then // Ngay khi đường mở được tìm thấy
          begin
            match[y] := x; //Chỉnh lại bộ ghép theo đường mở
```

```
            Exit; //Thoát luôn, lệnh Exit đặt ở đây sẽ thoát cả dây chuyền đệ quy
          end;
      end;
end;
begin //Thuật toán tìm bộ ghép cực đại trên đồ thị hai phía
  «Khởi tạo một bộ ghép bất kỳ, chẳng hạn ∅»;
  X* := «Tập các đỉnh chưa ghép»;
  repeat //Lặp các pha xử lý theo lô
    Old := |X*|; //Lưu số đỉnh chưa ghép khi bắt đầu pha
    for ∀y ∈ Y do avail[y] := True; //Đánh dấu mọi Y_đỉnh chưa thăm
    for ∀x ∈ X* do
      begin
        Found := False; //Cờ báo chưa tìm thấy đường mở
        Visit(x); //Tìm đường mở bằng DFS
        if Found then X* := X* - {x};
        //x đã được ghép, loại bỏ x khỏi X*
      end;
  until |X*| = Old; //Lặp cho tới khi không thể ghép thêm
end;
```

## BMATCH.PAS ✓ Tìm bộ ghép cực đại trên đồ thị hai phía

```
{$MODE OBJFPC}
program MaximumBipartiteMatching;
const
  maxN = 10000;
  maxM = 1000000;
var
  p, q, m: Integer;
  adj: array[1..maxM] of Integer;
  link: array[1..maxM] of Integer;
  head: array[1..maxN + 1] of Integer;
  match: array[1..maxN] of Integer;
  avail: array[1..maxN] of Boolean;
  List: array[1..maxN] of Integer;
  nList: Integer;
procedure Enter; //Nhập dữ liệu
var i, x, y: Integer;
begin
```

```
  ReadLn(p, q, m);
  FillChar(head[1], p * SizeOf(head[1]), 0);
  for i := 1 to m do
    begin
      ReadLn(x, y); //Đọc một cạnh (x, y), đưa y vào danh sách kề của x
      adj[i] := y;
      link[i] := head[x];
      head[x] := i;
    end;
end;
procedure Init; //Khởi tạo bộ ghép rỗng
var i: Integer;
begin
  FillChar(match[1], q * SizeOf(match[1]), 0);
  for i := 1 to p do List[i] := i;
  //Mảng List chứa nList X_đỉnh chưa ghép
  nList := p;
end;
procedure SuccessiveAugmentingPaths;
var
  Found: Boolean;
  Old, i: Integer;
  procedure Visit(x: Integer); //Thuật toán DFS từ x∈X
  var i, y: Integer;
  begin
    i := head[x]; //Từ đầu danh sách kề của x
    while i <> 0 do
      begin
        y := adj[i]; //Xét một đỉnh y∈Y kề x
        if avail[y] then //y chưa thăm, hiển nhiên (x, y) là cạnh chưa ghép
          begin
            avail[y] := False; //Đánh dấu thăm y
            if match[y] = 0 then Found := True
            //y chưa ghép thì báo hiệu tìm thấy đường mở
            else Visit(match[y]);
            //Thăm luôn match[y] ∈X (thăm liền 2 bước)
            if Found then //Tìm thấy đường mở
              begin
                match[y] := x; //Chỉnh lại bộ ghép
```

```
                Exit; //Thoát dây chuyền đệ quy
              end;
          end;
        i := link[i]; //Chuyển sang đỉnh kế tiếp trong danh sách các đỉnh kề x
      end;
  end;
begin
  repeat
    Old := nList; //Lưu lại số X_đỉnh chưa ghép
    FillChar(avail[1], q * SizeOf(avail[1]), True);
    for i := nList downto 1 do
      begin
        Found := False;
        Visit(List[i]); //Cố ghép List[i]
        if Found then //Nếu ghép được
          begin //Xóa List[i] khỏi danh sách các X_đỉnh chưa ghép
            List[i] := List[nList];
            Dec(nList);
          end;
      end;
  until Old = nList; //Không thể ghép thêm X_đỉnh nào nữa
end;
procedure PrintResult; //In kết quả
var j, k: Integer;
begin
  k := 0;
  for j := 1 to q do
    if match[j] <> 0 then
      begin
        Inc(k);
        WriteLn(k, ': x[', match[j], '] - y[', j, ']');
      end;
end;
begin
  Enter;
  Init;
  SuccessiveAugmentingPath;
  PrintResult;
```

```
end.
```

Nếu đồ thị có $n$ đỉnh ($n = p + q$) và $m$ cạnh, do mảng đánh dấu $avail[1 \dots q]$ chỉ được khởi tạo một lần trong pha, thời gian thực hiện của một pha sẽ bằng $O(n + m)$ (suy ra từ thời gian thực hiện giải thuật của DFS).

Các pha sẽ được thực hiện lặp cho tới khi $X^* = \emptyset$ hoặc khi một pha thực hiện xong mà không ghép thêm được đỉnh nào. Thuật toán cần không quá $p$ lần thực hiện pha xử lý lô, nên thời gian thực hiện giải thuật tìm bộ ghép cực đại trên đồ thị hai phía là $O(n^2 + nm)$ trong trường hợp xấu nhất. Còn trong trường hợp tốt nhất, ta có thể tìm được bộ ghép cực đại chỉ qua một lượt thực hiện pha xử lý lô, tức là bằng thời gian thực hiện giải thuật DFS. Cần lưu ý rằng đây chỉ là những đánh giá O lớn về cận trên của thời gian thực hiện. Thuật toán này chạy rất nhanh trên thực tế nhưng hiện tại chưa có đánh giá nào chặt hơn.

Ý tưởng tìm một lúc nhiều đường mở không có đỉnh chung đã được nghiên cứu trong bài toán luồng cực đại bởi Dinic[**10**]. Dựa trên ý tưởng này, Hopcroft và Karp[**21**] đã tìm ra thuật toán tìm bộ ghép cực đại trên đồ thị hai phía trong thời gian $O\left(\sqrt{|V|}|E|\right)$. Thuật toán Hopcroft-Karp trước hết sử dụng BFS để phân lớp các đỉnh theo độ dài đường đi ngắn nhất sau đó mới sử dụng DFS trên rừng các cây BFS để xử lý lô tương tự như cách làm của chúng ta ở trên.

## Bài tập

**2.35.** Có $p$ thợ và $q$ việc. Mỗi thợ cho biết mình có thể làm được những việc nào, và mỗi việc khi giao cho một thợ thực hiện sẽ được hoàn thành xong trong đúng 1 đơn vị thời gian. Tại một thời điểm, mỗi thợ chỉ thực hiện không quá một việc.

Hãy phân công các thợ làm các công việc sao cho:

- Mỗi việc chỉ giao cho đúng một thợ thực hiện.
- Thời gian hoàn thành tất cả các công việc là nhỏ nhất. Chú ý là các thợ có thể thực hiện song song các công việc được giao, việc của ai người nấy làm, không ảnh hưởng tới người khác.

**3.36.** Một bộ ghép $M$ trên đồ thị hai phía gọi là tối đại nếu việc bổ sung thêm bất cứ cạnh nào vào $M$ sẽ làm cho $M$ không còn là bộ ghép nữa.

a) Chỉ ra một ví dụ về bộ ghép tối đại nhưng không là bộ ghép cực đại trên đồ thị hai phía

b) Tìm thuật toán $O(|E|)$ để xác định một bộ ghép tối đại trên đồ thị hai phía

c) Chứng minh rằng nếu $A$ và $B$ là hai bộ ghép tối đại trên cùng một đồ thị hai phía thì $|A| \leq 2|B|$ và $|B| \leq 2|A|$. Từ đó chỉ ra rằng nếu thuật toán đường mở được khởi tạo bằng một bộ ghép tối đại thì số lượt tìm đường mở giảm đi ít nhất một nửa so với việc khởi tạo bằng bộ ghép rỗng.

**3.37.** (Phủ đỉnh – Vertex Cover) Cho đồ thị hai phía $G = (X \cup Y, E)$. Bài toán đặt ra là hãy chọn ra một tập $C$ gồm ít nhất các đỉnh sao cho mọi cạnh $\in E$ đều liên thuộc với ít nhất một đỉnh thuộc $C$.

Bài toán tìm phủ đỉnh nhỏ nhất trên đồ thị tổng quát là NP-đầy đủ, hiện tại chưa có thuật toán đa thức để giải quyết. Tuy vậy trên đồ thị hai phía, phủ đỉnh nhỏ nhất có thể tìm được dựa trên bộ ghép cực đại.

Dựa vào mô hình luồng của bài toán bộ ghép cực đại, giả sử các cung $(X, Y)$ có sức chứa $+\infty$, các cung $(s, X)$ và $(Y, t)$ có sức chứa 1. Gọi $(S, T)$ là lát cắt hẹp nhất của mạng. Đặt $C = \{x \in T\} \cup \{y \in S\}$.

a) Chứng minh rằng $C$ là một phủ đỉnh

b) Chứng minh rằng $C$ là phủ đỉnh nhỏ nhất

c) Giả sử ta tìm được $M$ là bộ ghép cực đại trên đồ thị hai phía, khi đó chắc chắn không còn tồn tại đường mở tương ứng với bộ ghép $M$. Đặt:

$Y^* = \{y \in Y: \exists x \in X$ chưa ghép, $x$ đến được $y$ qua một đường pha$\}$

$X^* = \{x \in X: x$ đã ghép và đỉnh ghép với $x$ không thuộc $Y^*\}$

Chứng minh rằng $(X^*, Y^*)$ là lát cắt hẹp nhất.

d) Xây dựng thuật toán tìm phủ đỉnh nhỏ nhất trên đồ thị hai phía dựa trên thuật toán tìm bộ ghép cực đại.

**3.38.** Cho $M$ là một bộ ghép trên đồ thị hai phía $G = (X \cup Y, E)$. Gọi $k$ là số $X$_đỉnh chưa ghép. Chứng minh rằng ba mệnh đề sau đây là tương đương:

- $M$ là bộ ghép cực đại.
- $G$ không có đường mở tương ứng với bộ ghép $M$.
- Tồn tại một tập con $A$ của $X$ sao cho $|N(A)| = |A| - k$. Ở đây $N(A)$ là tập các $Y$_đỉnh kề với một đỉnh nào đó trong $A$ (*Gợi ý: Chọn A là tập các X_đỉnh đến được từ một X_đỉnh chưa ghép bằng một đường pha*)

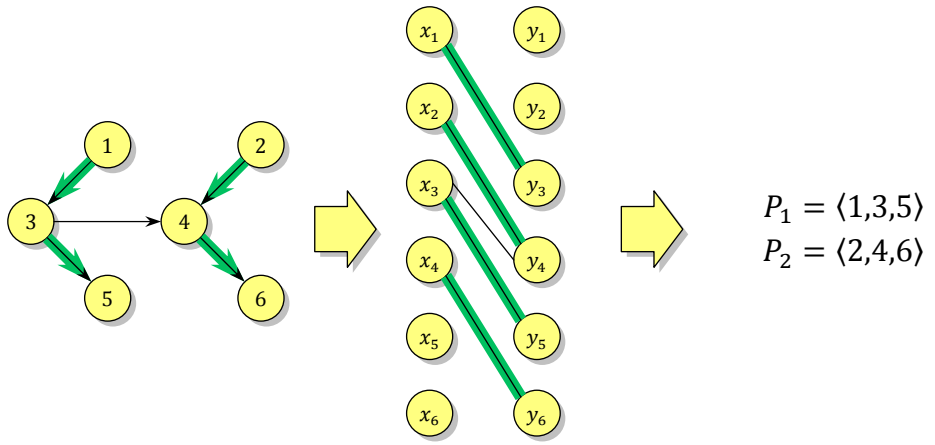
**2.39. (**Định lý Hall) Cho $G = (X \cup Y, E)$ là đồ thị hai phía có $|X| = |Y|$. Chứng minh rằng $G$ có bộ ghép đầy đủ (bộ ghép mà mọi đỉnh đều được ghép) nếu và chỉ nếu $|A| \leq |N(A)|$ với mọi tập $A \subseteq X$.

**2.40.** (Phủ đường tối thiểu) Cho $G = (V, E)$ là đồ thị có hướng không có chu trình. Một *phủ đường* (p*ath cover*) là một tập $P$ các đường đi trên $G$ thỏa mãn: Với mọi đỉnh $v \in V$, tồn tại duy nhất một đường đi trong $P$ chứa $v$. Đường đi có thể bắt đầu và kết thúc ở bất cứ đâu, tính cả đường đi độ dài 0 (chỉ gồm một đỉnh). Bài toán đặt ra là tìm *phủ đường tối thiểu* (*minimum path cover*): Phủ đường gồm ít đường đi nhất.

Gọi $n$ là số đỉnh của đồ thị, ta đánh số các đỉnh thuộc $V$ từ 1 tới $n$. Xây dựng đồ thị hai phía $G' = (X \cup Y, E')$ trong đó:

$$X = \{x_1, x_2, \dots x_n\}$$
$$Y = \{y_1, y_2, \dots y_n\}$$

Tập cạnh $E'$ được xây dựng như sau: Với mỗi cung $(i, j) \in E$, ta thêm vào một cạnh $(x_i, y_j) \in E'$ (h.2.16)




***Hình 2.16. Bài toán tìm phủ đường tối thiểu trên DAG có thể quy về bài toán bộ ghép cực đại trên đồ thị hai phía.***

Gọi $M$ là một bộ ghép trên $G'$. Khởi tạo $P$ là tập $n$ đường đi, mỗi đường đi chỉ gồm một đỉnh trong $G$, khi đó $P$ là một phủ đường. Xét lần lượt các cạnh của bộ ghép, mỗi khi xét tới cạnh $(x_i, y_j)$ ta đặt cạnh $(i, j)$ nối hai đường đi trong $P$ thành một đường…Khi thuật toán kết thúc, $P$ vẫn là một phủ đường.

a) Chứng minh tính bất biến vòng lặp: Tại mỗi bước khi xét tới cạnh $(x_i, y_j) \in M$, cạnh $(i, j) \in E$ chắc chắn sẽ nối hai đường đi trong $P$: một đường đi kết thúc ở $i$ và một đường đi khác bắt đầu ở $j$. Từ đó chỉ ra tính đúng đắn của thuật toán. (*Gợi ý:* mỗi khi xét tới cạnh $(x_i, y_j) \in M$ và đặt cạnh $(i, j)$ nối hai đường đi của $P$ thành một đường thì $|P|$ giảm 1. Vậy khi thuật toán trên kết thúc, $|P| = n - |M|$, tức là muốn $|P| \to \min$ thì $|M| \to \max$).

b) Viết chương trình tìm phủ đường cực tiểu trên đồ thị có hướng không có chu trình.

c) Chỉ ra ví dụ để thấy rằng thuật toán trên không đúng trong trường hợp $G$ có chu trình.

d) Chứng minh rằng nếu tìm được thuật toán giải bài toán tìm phủ đường cực tiểu trên đồ thị tổng quát trong thời gian đa thức thì có thể tìm được đường đi Hamilton trên đồ thị đó (nếu có) trong thời gian đa thức. (Lý

thuyết về độ phức tạp tính toán đã chứng minh được rằng trên đồ thị tổng quát, bài toán tìm đường đi Hamilton là NP-đầy đủ và bài toán tìm phủ đường cực tiểu là NP-khó. Có nghĩa là một thuật toán với độ phức tạp đa thức để giải quyết bài toán phủ đường cực tiểu trên đồ thị tổng quát sẽ là một phát minh lớn và đáng ngạc nhiên).

**2.41.** Tự tìm hiểu về thuật toán Hopcroft-Karp. Cài đặt và so sánh tốc độ thực tế với thuật toán trong bài.

**2.42.** (Bộ ghép cực đại trên đồ thị chính quy hai phía) Một đồ thị vô hướng $G = (V, E)$ gọi là *đồ thị chính quy bậc k* (*k-regular graph*) nếu bậc của mọi đỉnh đều bằng $k$. Đồ thị chính quy bậc 0 là đồ thị không có cạnh nào, đồ thị chính quy bậc 1 thì các cạnh tạo thành bộ ghép đầy đủ, đồ thị chính quy bậc 2 có các thành phần liên thông là các chu trình đơn.

a) Chứng minh rằng đồ thị hai phía $G = (X \cup Y, E)$ là đồ thị chính quy thì $|X| = |Y|$.

b) Chứng minh rằng luôn tồn tại bộ ghép đầy đủ trên đồ thị hai phía chính quy bậc $k$ ($k \geq 1$).

c) Tìm thuật toán $O(|E| \log |E|)$ để tìm một bộ ghép đầy đủ trên đồ thị chính quy bậc $k \geq 1$.