

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 lý 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

KIẾU DỮ LIỆU TRÙU TƯỢNG VÀ CẤU TRÚC DỮ LIỆU

Kiểu dữ liệu trùu tượng là một mô hình toán học với những thao tác định nghĩa trên mô hình đó. Kiểu dữ liệu trùu 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 trùu 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 trùu tượng cũng như cách cài đặt chúng bằng các cấu trúc dữ liệu. Những kiểu dữ liệu trùu 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 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;
```

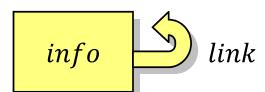
Trường hợp tốt nhất, vị trí xóa nằm cuối danh sách ($p = n$), khi đó thời gian thực hiện của phép xóa là $\Theta(1)$. Trường hợp xấu nhất, ta cần xóa tại vị trí 1, khi đó thời gian thực hiện của phép xóa là $\Theta(n)$. Cũng dễ dàng chứng minh được rằng thời gian thực hiện trung bình của phép xóa là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

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*.



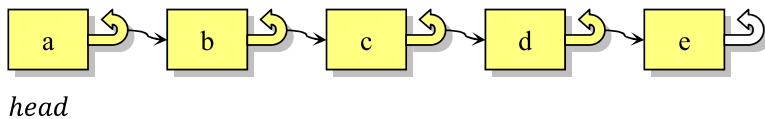
```
type
PNode = ^TNode; //Kiểu con trỏ tới một nút
TNode = record; //Kiểu biến động chứa thông tin trong một nút
```

```

    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



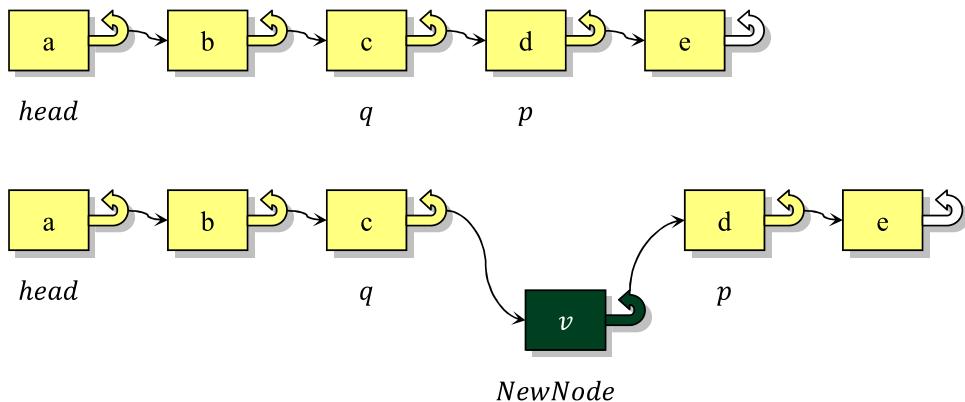
Hình 1.1. Danh sách nối đơn

a) Truy cập phần tử trong danh sách nối đơn

Bản thân danh sách nối đơn đã là một kiểu dữ liệu trùu tượng. Để cài đặt kiểu dữ liệu trùu 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 *thắng 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 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:

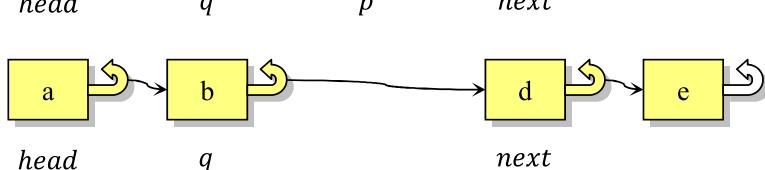
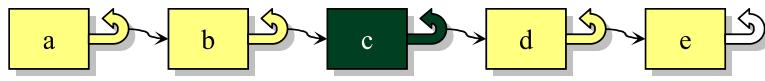
Để xóa nút p khỏi danh sách nối đơn, gọi $next$ là nút đứng liền sau p trong danh sách. Xét hai trường hợp:

- 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

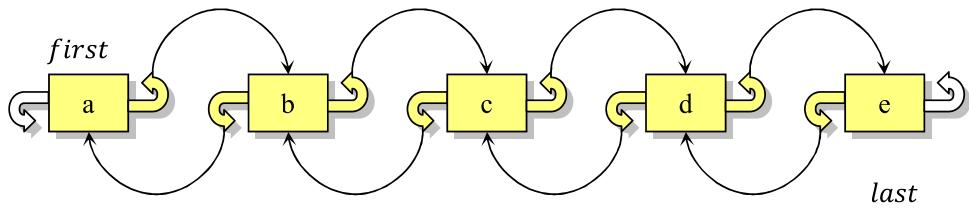
```

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

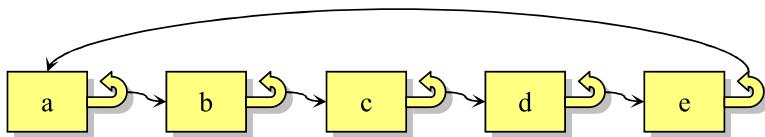


Hình 1.4. Danh sách nối kép

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 trở thăng về phần tử đầu tiên của danh sách thì ta sẽ được một kiểu danh sách mới gọi là danh sách nối vòng đơn.



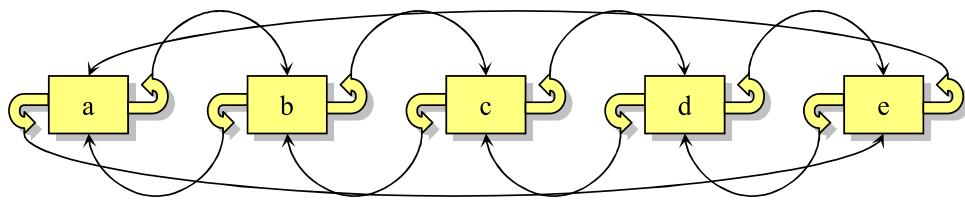
Hình 1.5. Danh sách nối vòng đơn

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 $p(x) = a_1x^{b_1} + a_2x^{b_2} + \dots + a_nx^{b_n}$, trong đó $b_1 > b_2 > \dots > b_n$ 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 $a_n a_{n-1} \dots a_0$, 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 biết 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 biết 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 nguyên *front* 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];
            end;

```

```

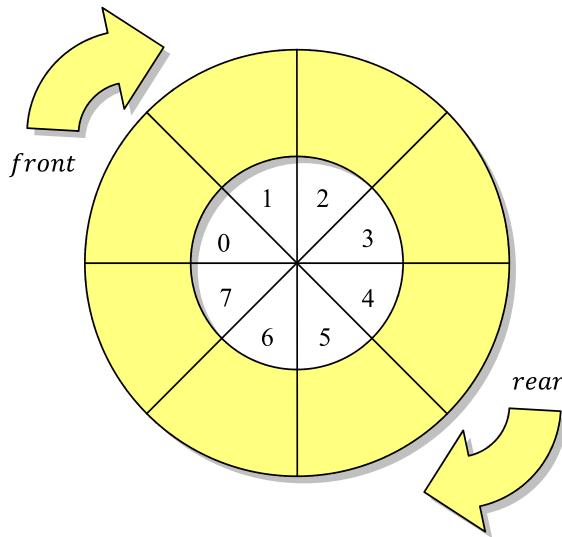
    front := front + 1;
end;
end;

```

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 đầy 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];
            end;

```

```

        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

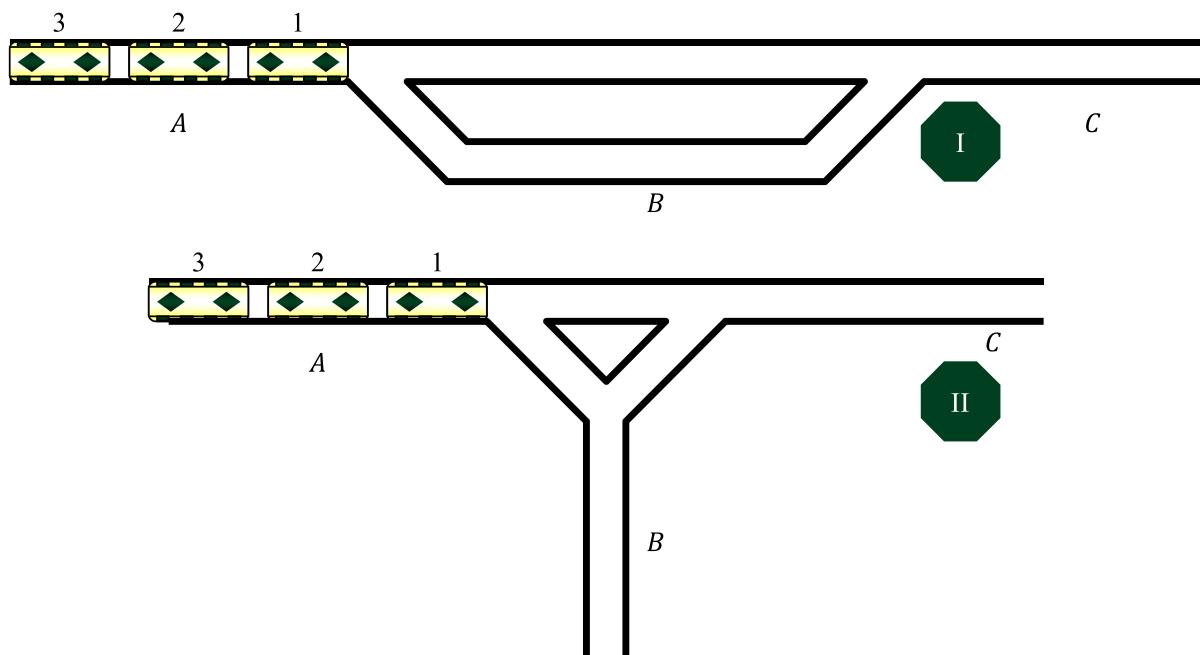
$PushR(v)$: Đẩy phần tử v vào cuối danh sách

$PopF$: 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, \dots, 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 $A \rightarrow B$, $A \rightarrow B$ hoặc $A \rightarrow C$. 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*).

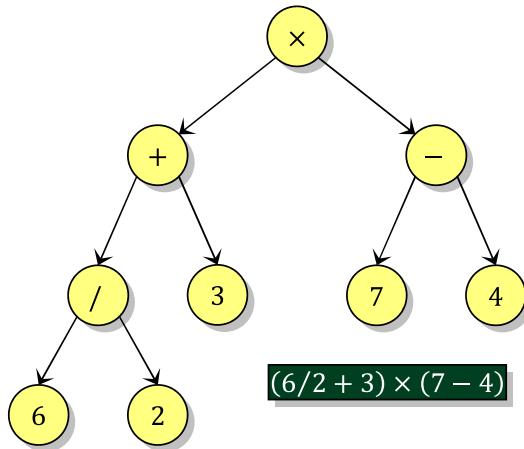
Có thể định nghĩa cây bằng cách đê quy như sau:

- 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:

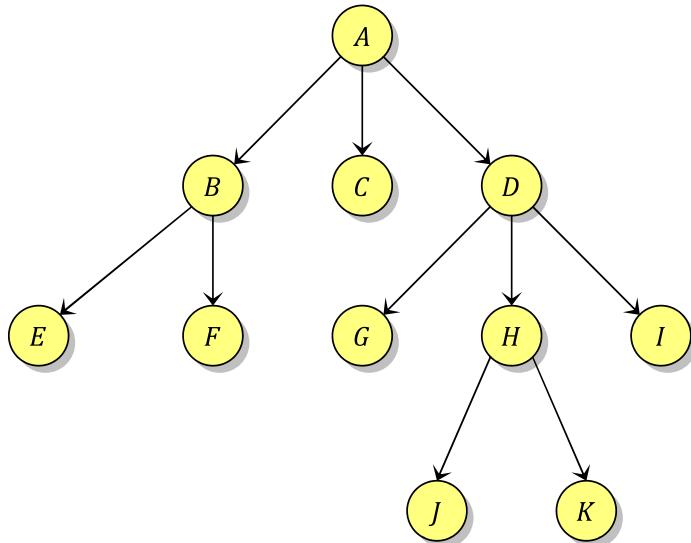
- Mục lục của một cuốn sách với phần, chương, bài, mục v.v... có cấu trúc của cây
- Cấu trúc thư mục trên đĩa cũng có cấu trúc cây, thư mục gốc có thể coi là gốc của cây đó với các cây con là các thư mục con (sub-directories) và tệp (files) nằm trên thư mục gốc.
- Gia phả của một họ tộc cũng có cấu trúc 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 (height) 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:

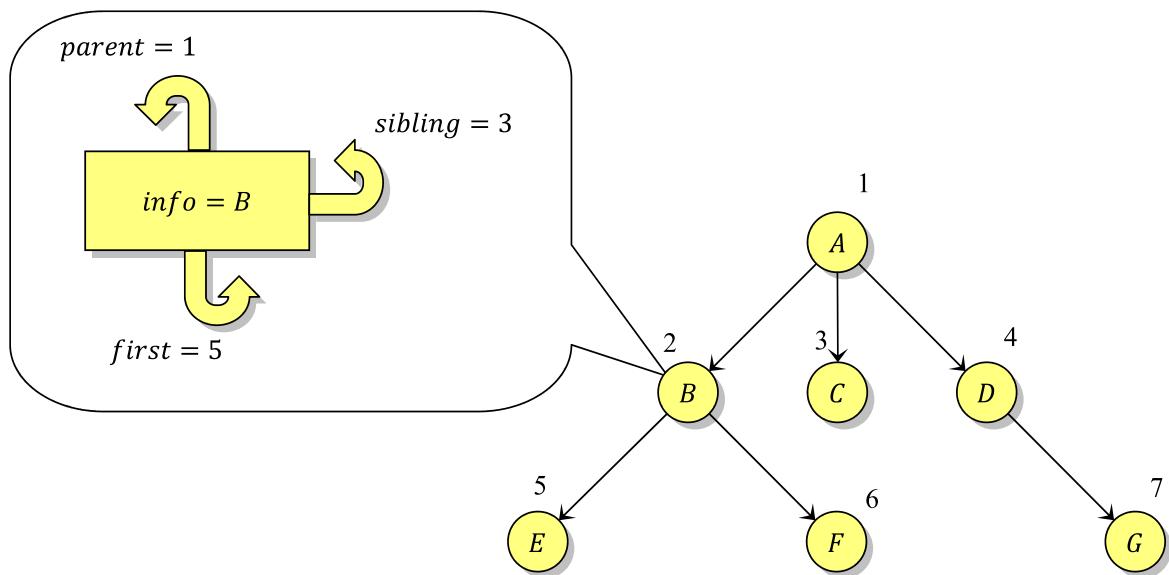
- Trường *info*: Chứa giá trị lưu trong nút
- Trường *parent*: Chứa con trỏ liên kết tới nút cha, tức là chứa một thông tin đủ để biết nút cha của nút đang xét là nút nào. Trong trường hợp nút đang xét là gốc (không có nút cha), trường *parent* được gán một giá trị đặc biệt (*nil*).
- Trường *first*: 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

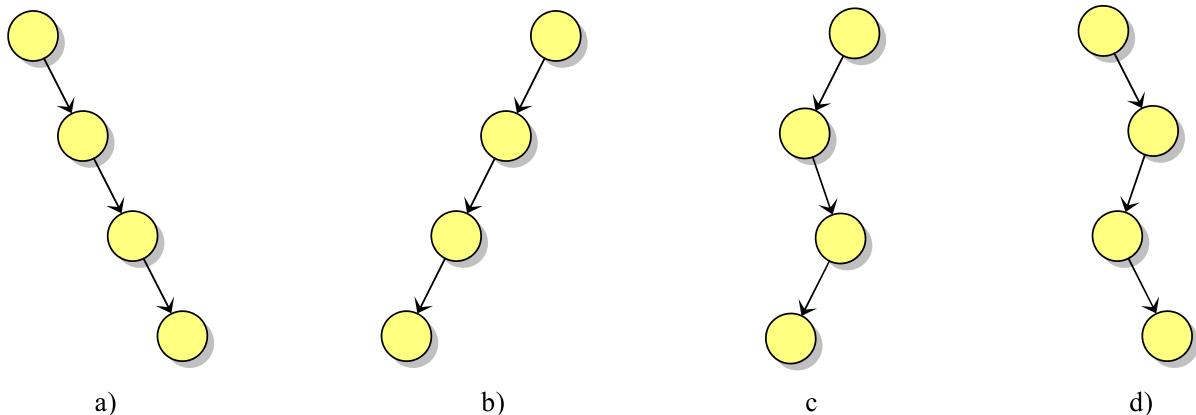
Trong các biểu diễn này, từ một nút r bất kỳ, ta có thể đi theo liên kết $first$ để đến nút con đầu tiên, nút này chính là chót của một danh sách nối đơn các nút con: Từ nút $first$, đi theo liên kết $sibling$, ta có thể duyệt tất cả các nút con của nút r .

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 mốc 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

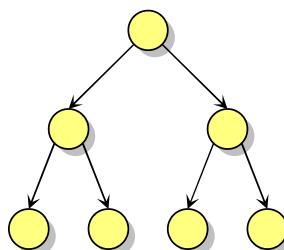
Cây nhị phân (*binary tree*) là một dạng quan trọng của cấu trúc cây. Nó có đặc điểm là mọi nút trên cây chỉ có tối đa hai nhánh con. Với một nút thì người ta cũng phân biệt cây con trái và cây con phải của nút đó, tức là cây nhị phân là cây có thứ tự.

a) Một số dạng đặc biệt của cây nhị phân



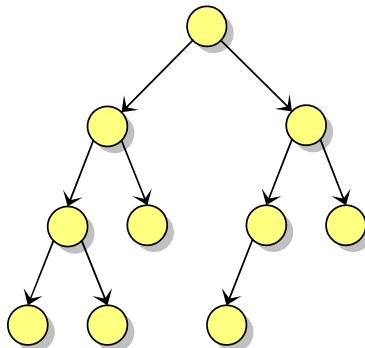
Hình 1.11. Cây nhị phân suy biến

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.



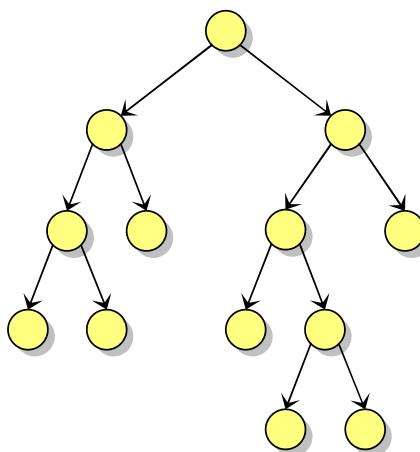
Hình 1.11. Cây nhị phân hoàn chỉnh

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:

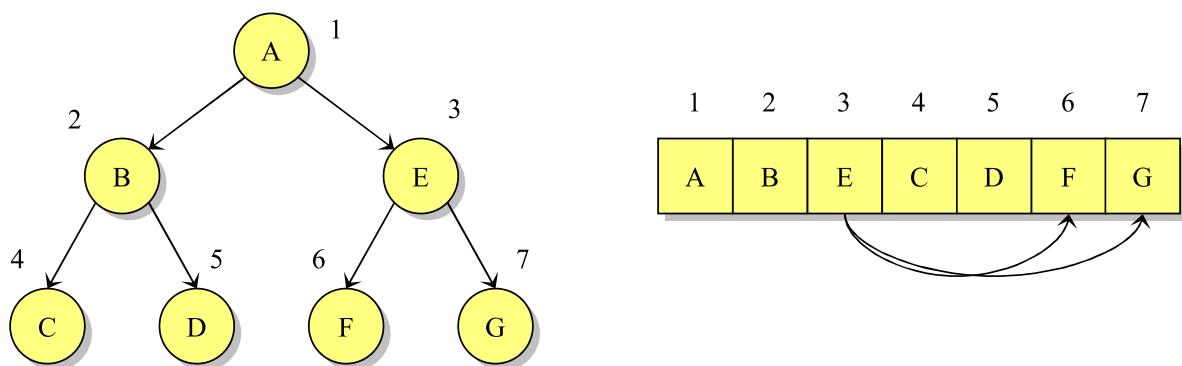
- Trong các cây nhị phân có cùng số lượng nút như nhau thì cây nhị phân suy biến có chiều cao lớn nhất, còn cây nhị phân gần hoàn chỉnh thì có chiều cao nhỏ nhất.
- Số lượng tối đa các nút ở độ sâu d của cây nhị phân là 2^d
- Số lượng tối đa các nút trên một cây nhị phân có chiều cao h là $2^{h+1} -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

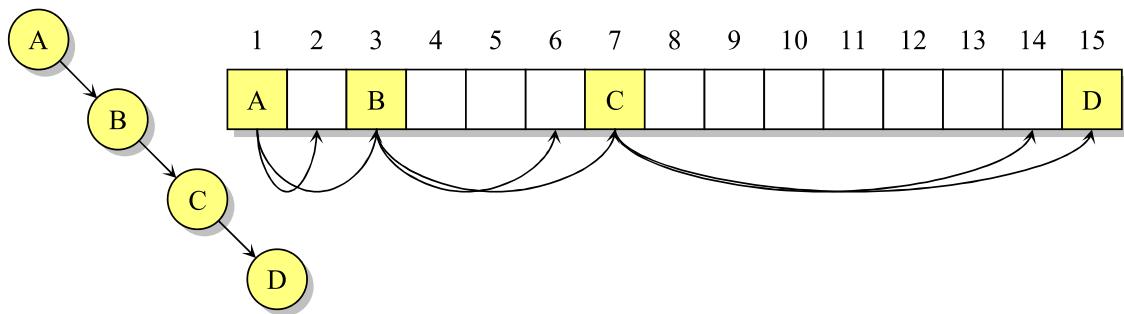
Một cây nhị phân hoàn chỉnh có n nút thì có chiều cao là $\lfloor \lg n \rfloor$, tức là các nút sẽ nằm ở các độ sâu từ 0 tới $\lfloor \lg n \rfloor$. Khi đó ta có thể liệt kê tất cả các nút từ độ sâu 0 (nút gốc) tới độ sâu $\lfloor \lg n \rfloor$, sao cho với các nút cùng độ sâu thì thứ tự liệt kê là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



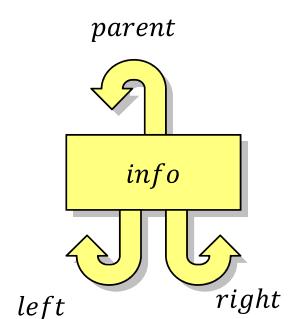
Hình 1.15. Nhược điểm của phương pháp biểu diễn cây nhị phân bằng mảng

□ 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

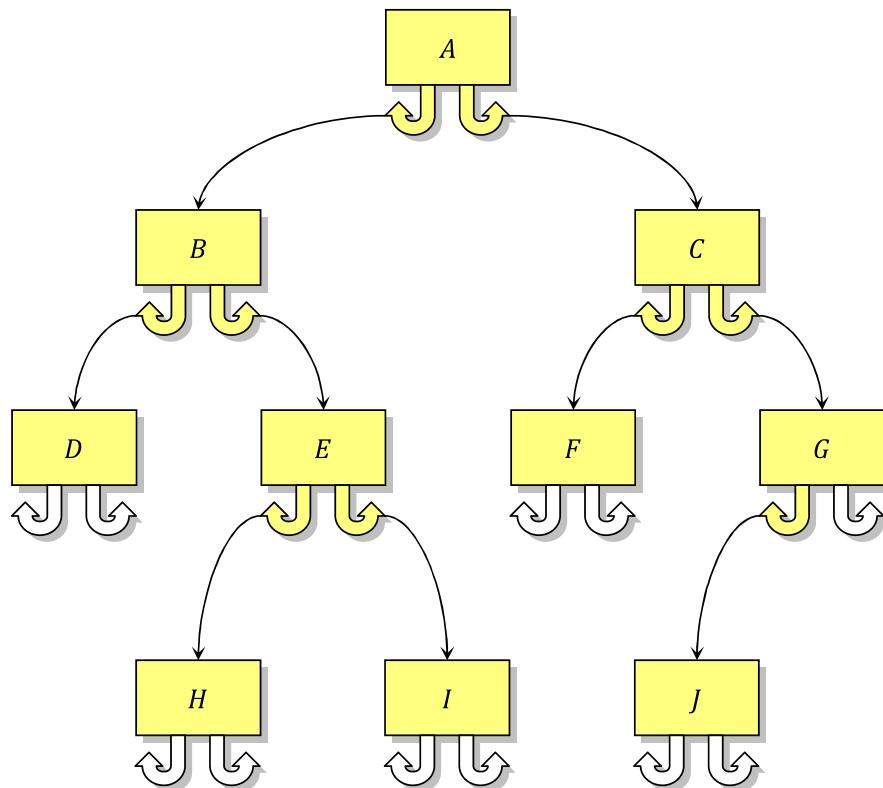
```

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

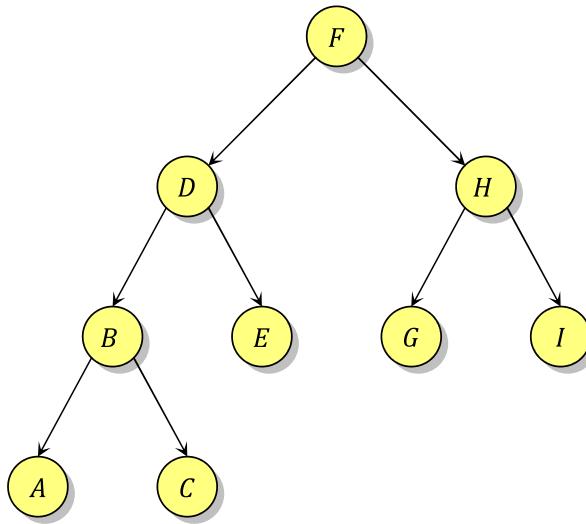
Trong phép duyệt *theo thứ tự trước* (*preorder traversal*) thì giá trị trong mỗi nút bất kỳ sẽ được liệt kê trước tất cả các giá trị lưu trong hai nhánh con của nó, có thể mô tả bằng thủ tục đệ quy sau:

```

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. Độ (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.

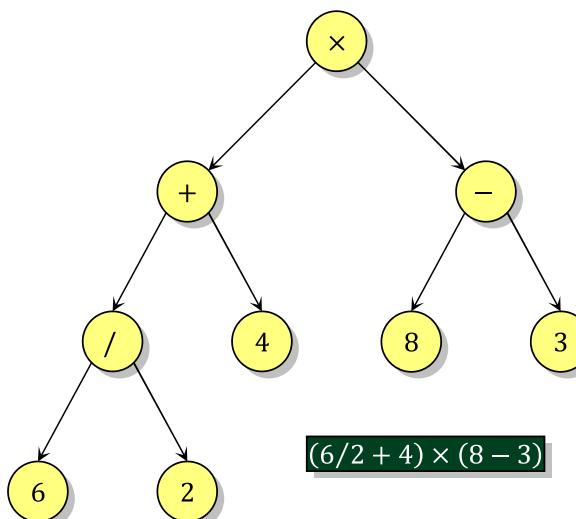
- 1.10.** Chỉ ra rằng cấu trúc của một cây nhị phân có thể khôi phục một cách đơn định nếu ta biết được thứ tự duyệt trước và giữa của các nút. Tương tự như vậy, cấu trúc cây có thể khôi phục nếu ta biết được thứ tự duyệt sau và giữa của các nút.
- 1.11.** Tìm ví dụ về hai cây nhị phân khác nhau nhưng 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 $((6/2) + 4) \times (8 - 3)$. 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
\times	Lấy ra 5 và 7, đẩy vào $7 \times 5 = 35$	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.

💻 RPN CALC.PAS ✓ Tính giá trị biểu thức RPN

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

```

TStackNode = record
    // Ngăn xếp được cài đặt bằng danh sách mốc 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 chưa 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
```

```

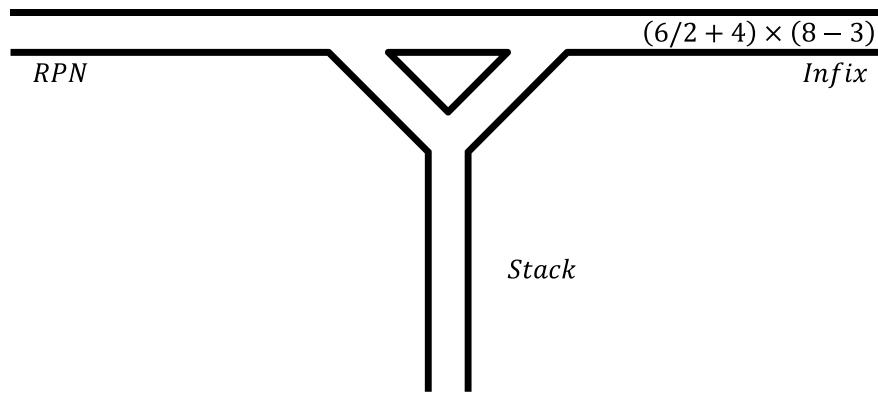
end;
else //Gặp toán hạng thì hiển thị luôn
    Output ← token;
end;
//Khi đọc xong biểu thức, lấy ra và hiển thị tất cả các phần tử còn lại trong ngăn xếp
while Stack ≠ Ø do
    Output ← Pop

```

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 "x" 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 "x" 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 mốc 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
      end;
  
```

```

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: $a\ b\ \text{and}\ c\ d\ \text{and}\ \text{or}$.

1.14. Chuyển các biểu thức sau đây ra dạng RPN

- $A \times (B + C)$
- $A + (B/C) + D$
- $A \times (B + -C)$
- $A - (B + C)^{D/E}$
- $(A \text{ or } B) \text{ and } (C \text{ or } (D \text{ or not } E))$
- $(A = B) \text{ or } (C = D)$

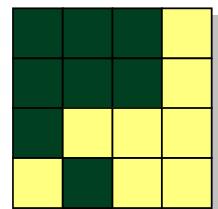
1.15. Với một ảnh đen trắng hình vuông kích thước $2^n \times 2^n$, người ta dùng phương pháp sau để mã hóa ảnh:

- 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:



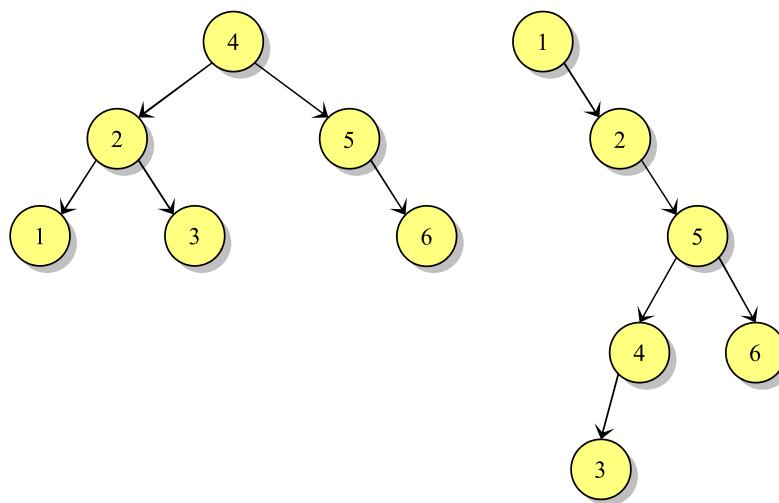
Bài toán đặt ra là cho số nguyên dương n và hai xâu mã hóa của hai ảnh kích thước $2^n \times 2^n$. Hãy cho biết hai ảnh đó có khác nhau không và nếu chúng khác nhau hãy chỉ ra một vị trí có màu khác nhau trên hai ảnh.

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 đặc 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 dụng 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* ≠ *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: $Result := Maximum(x^.Left)$.
- 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 di 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

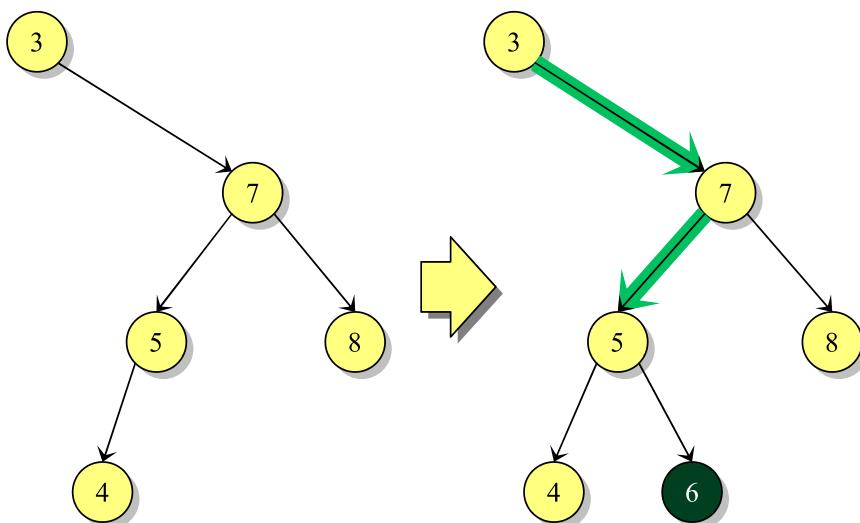
Phép tìm kiếm nhận vào một nút x và một khóa k . Nếu khóa k có trong nhánh BST gốc x thì trả về một nút chứa khóa k , nếu không trả về $nilT$.

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
```

```

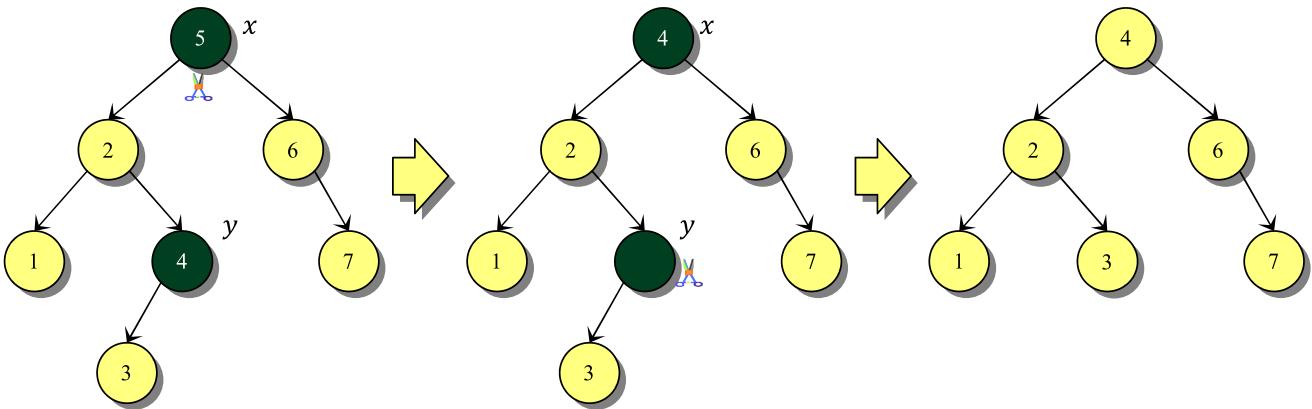
function Insert(k: TKey) : PNode;
var x, y: PNode;
begin
    y := nilT;
    x := root; //Bắt đầu từ gốc
    while x ≠ nilT do
        begin
            y := x;
            if k < x^.key then x := x^.left //Chèn vào nhánh trái
            else x := x^.right; //Chèn vào nhánh phải
        end;
    New(x); //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

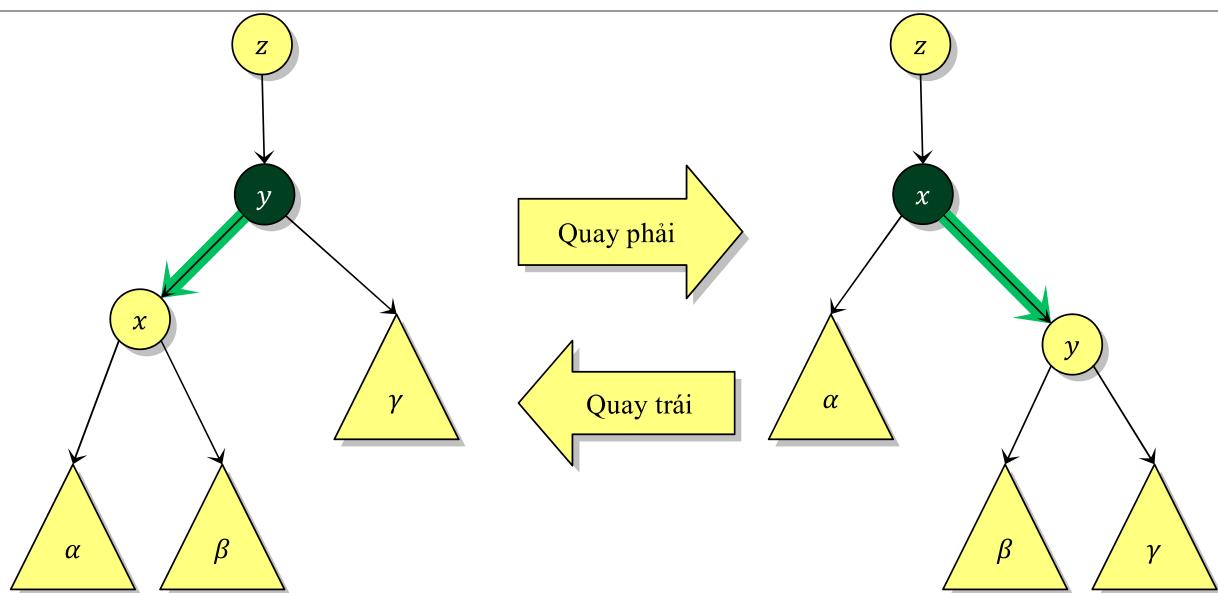
trở thành nút con trái của y và nút con trái của y trở thành nút con phải của x .

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 chưa 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 nốt 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 $x \neq \text{root}$, và $y = x^.parent$, phép *UpTree(x)* sẽ quay theo liên kết $y \rightarrow x$ 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 đây 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 ≥ 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 $root^{\wedge}$ có màu đen
- Nút $nilT^{\wedge}$ 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 bếp” 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 đỗ 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 $Minimum^{\wedge}$, người ta gọi hàm *Successor* để đi sang nút liền sau cho tới khi duyệt qua nút $Maximum^{\wedge}$.

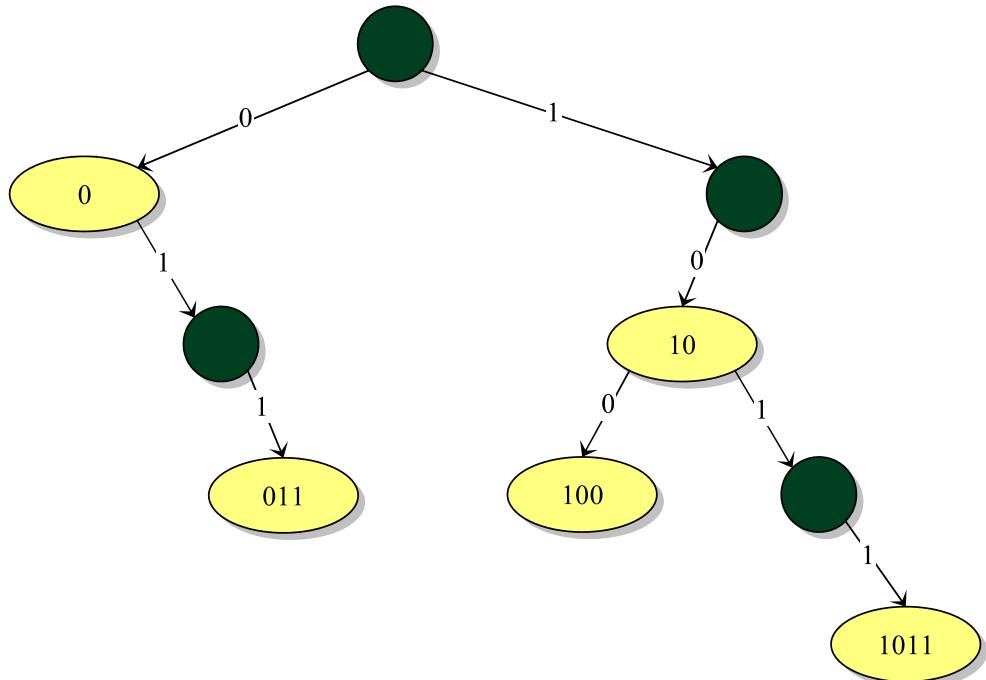
Chứng minh rằng thời gian thực hiện giải thuật này có thời gian thực hiện $O(n)$.

Gợi ý: Thực hiện n lời gọi *Successor* liên 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 : $p_2 := \text{Successor}(p_1)$, tiếp theo lại tìm nút p_3 là nút liền sau p_2, \dots . 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 sở) 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, gấp bit 0 đi sang nút con trái và gấp

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 X là biến ngẫu nhiên cho chiều cao của BST. Chứng minh rằng kỳ vọng $E[X] = O(\lg n)$.
- 1.26.** Cho BST tạo thành từ n khóa được chèn vào theo một trật tự ngẫu nhiên, gọi X là biến ngẫu nhiên cho độ sâu của một nút. Chứng minh rằng kỳ vọng $E[X] = O(\lg n)$.
- 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 $b(n) = \frac{4^n}{\sqrt{\pi} n^{3/2}} (1 + O(1/n))$

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 $\lfloor \lg n \rfloor$ 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 $nilT^{\wedge}$ 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 ≥ 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 đặc 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:

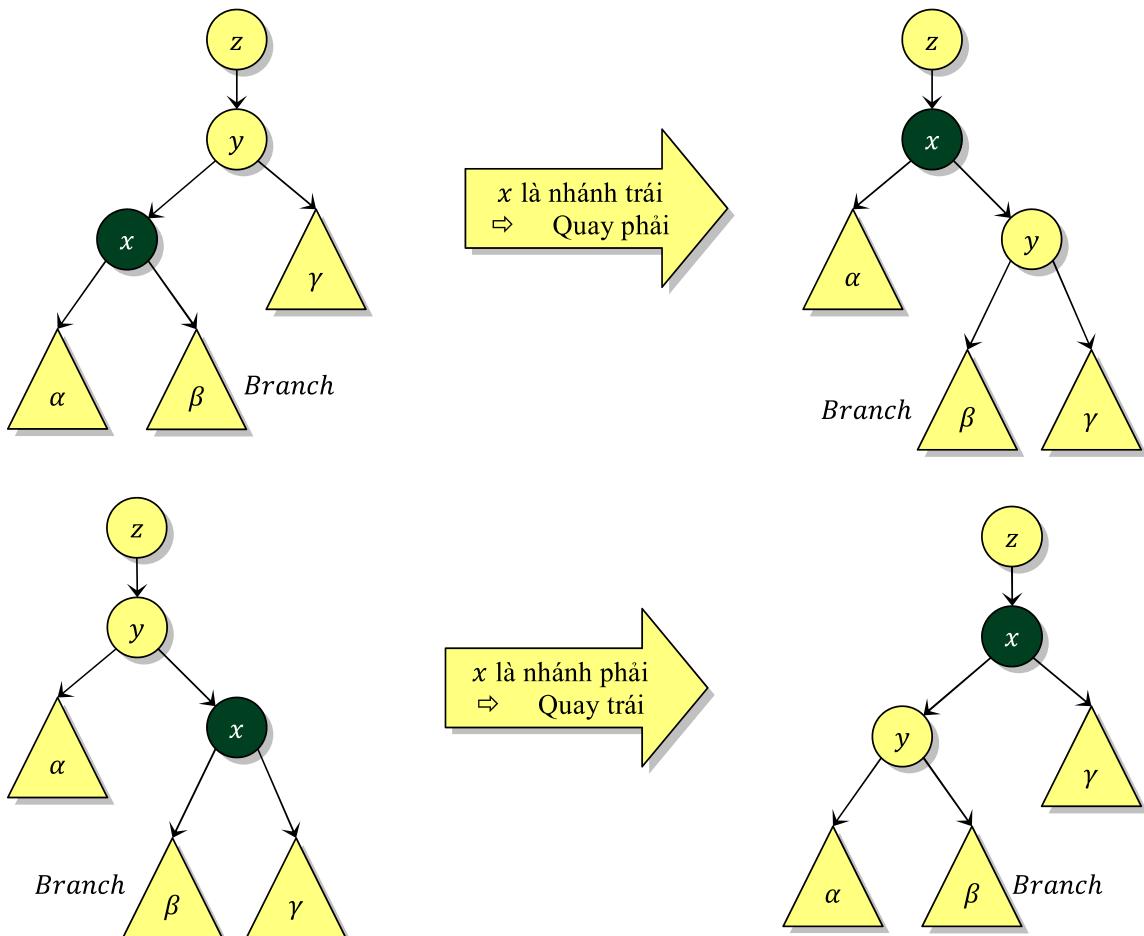
```

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); //Mởc nối x^ vào làm con z^ thay cho y^
if root = y then root := x;
//Cập nhật lại gốc BST nếu trước đây 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)

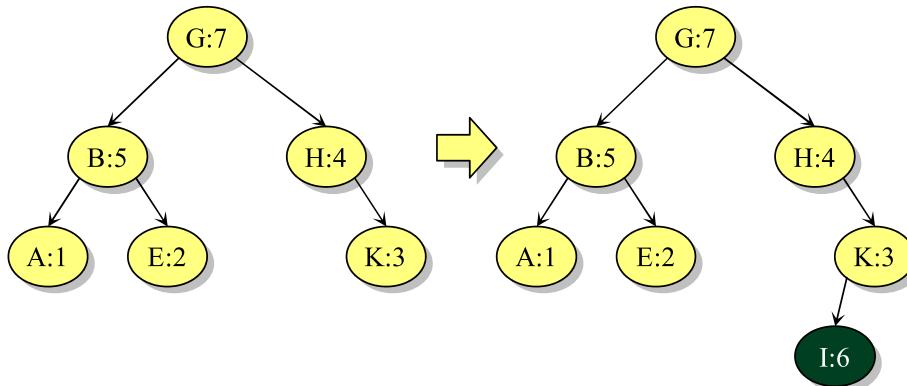
```

```

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

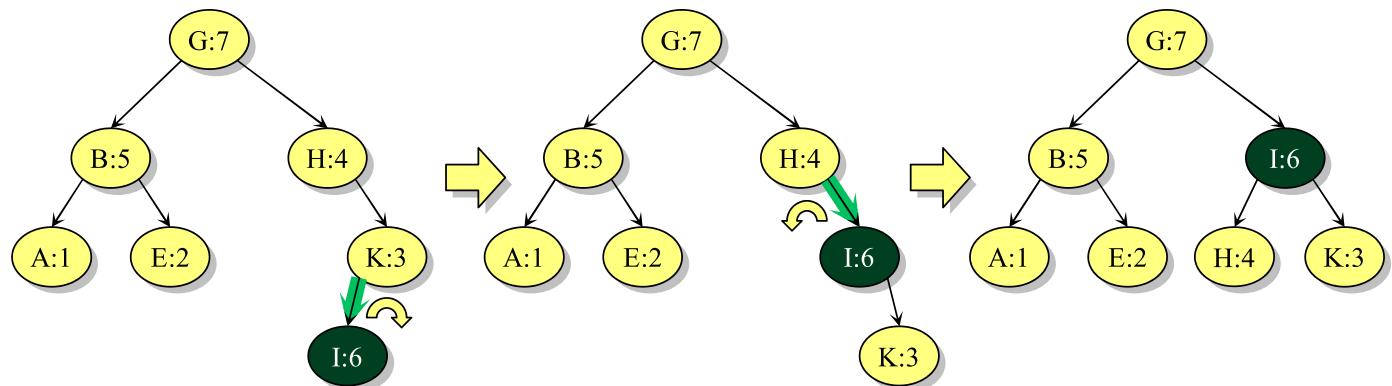
```

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.

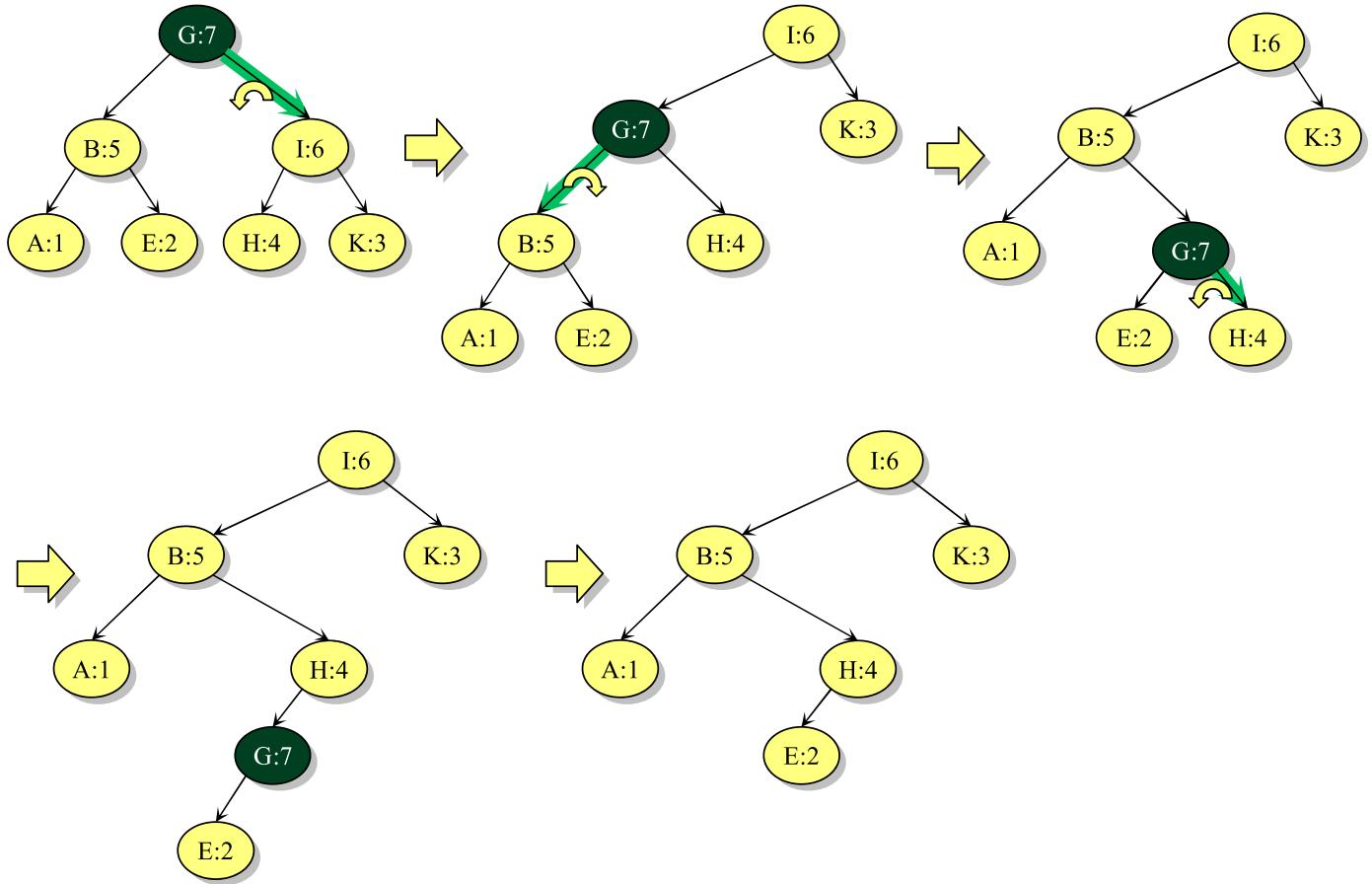
c) Xóa

Phép xóa nút x^{\wedge} trên Treap được thực hiện như sau:

- Nếu x^{\wedge} có ít hơn hai nhánh con, ta lấy nút con (nếu có) của x^{\wedge} lên thay cho x^{\wedge} và xóa nút x^{\wedge} .
- Nếu x^{\wedge} có đúng hai nhánh con, gọi y^{\wedge} là nút con mang độ ưu tiên lớn hơn trong hai nút con, thực hiện phép $UpTree(y)$ để kéo nút x^{\wedge} xuống sâu phía dưới lá và lặp lại cho tới khi x^{\wedge} 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  $x^{\wedge}$  có 2 nút con  
        begin  
            //Xác định  $y^{\wedge}$  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 gốc, kéo x xuống lá  
        end;  
        //Bây giờ  $x^{\wedge}$  chỉ có tối đa một nút con, xác định  $y^{\wedge}$  là nút con (nếu có) của x  
        if x^.left <> nilT then y := x^.left  
        else y := x^.right;  
        z := x^.parent; //z^{\wedge} là nút cha của  $x^{\wedge}$   
        SetLink (z, y, z^.left = x); //Cho  $y^{\wedge}$  làm con của  $z^{\wedge}$  thay cho  $x^{\wedge}$   
        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 mốc 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 mốc 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	9 1 6 5
I 5 1	
I 6 1	
I 7 1	
I 8 3	
I 1 2	
D 4	
I 9 2	
D 1	

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 $root^{\wedge}$ 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^{\wedge} của Treap, ta lưu trữ $r^{\wedge}.size$ là số lượng nút nằm trong nhánh Treap gốc r^{\wedge} . 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^{\wedge} :

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

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^{\wedge} chứa giá trị v , xác định nút y^{\wedge} là nút hiện đang đứng thứ i . Nếu y^{\wedge} không có nhánh trái thì mốc nối x^{\wedge} vào thành nút con trái của y^{\wedge} . Nếu không ta đi sang nhánh trái của y^{\wedge} và mốc nối x^{\wedge} vào thành nút cực phải của nhánh trái này.

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

Cuối cùng, ta gán cho $x^{\wedge}.priority$ một độ ưu tiên ngẫu nhiên và thực hiện các phép $UpTree(x)$ để đẩy x^{\wedge} 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^{\wedge} nằm tại vị trí i và tiến hành xóa nút x^{\wedge} . Phép xóa được thực hiện như trên Treap: Chứng nào x^{\wedge} còn hai nút con, ta xác định y^{\wedge} là nút con mang độ ưu tiên lớn hơn và thực hiện $UpTree(y)$ để kéo x^{\wedge} sâu xuống dưới lá. Khi x^{\wedge} còn ít hơn hai nút con, ta đưa nhánh con gốc y^{\wedge} (nếu có) của x^{\wedge} vào thế chỗ và xóa nút x^{\wedge} . Sau khi xóa thì toàn bộ trường $Size$ trong các nút tiền bối của x^{\wedge} 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;
```

```

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; //Di 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, gấp 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;
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 trùu 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 **dụng 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 $Marked[x] := True$, 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;
```

```

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; //Di 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); //Tim nút chứa người bị loại
            Write(tree[Node], ' ');
            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. Thú 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 trực số.

Có ba loại khoảng trên trực 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

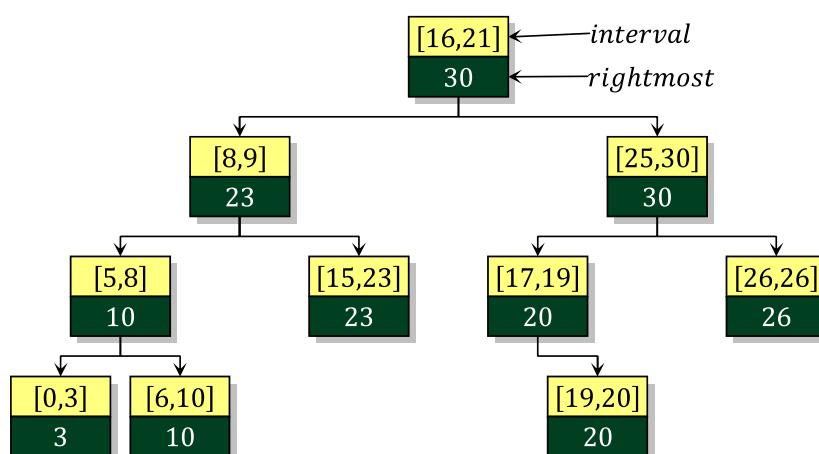
Với hai đoạn $i_1 = [s_1, f_1]$ và $i_2 = [s_2, f_2]$, đúng một trong ba mệnh đề dưới đây thỏa mãn (interval trichotomy):

- 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 $nilT^{\wedge}$ 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^{\wedge} được tính theo công thức truy hồi:

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

Khi một nút x^{\wedge} được chèn vào thành một nút lá hay bị xóa khỏi BST, tất cả các trường *rightmost* trong x^{\wedge} và các nút tiền bối của x^{\wedge} 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 gói 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 $[a, b] = [x, x]$

Đĩ 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 gói 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 gói 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 $x^\wedge = root^\wedge$, nếu đoạn trong x^\wedge có giao với $[a, b]$ thì xong. Ngược lại, nếu $x^\wedge.left^\wedge.rightmost \geq a$, ta quy về tìm trong nhánh con trái của x^\wedge , nếu không ta quy về tìm trong nhánh con phải của x^\wedge :

```
//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 $x^\wedge.left^\wedge.rightmost \geq a$ thì chỉ cần tìm trong nhánh con trái của x^\wedge là đủ, bởi nếu tìm trong nhánh con trái của x^\wedge không thấy thì chắc chắn tìm trong nhánh con phải cũng thất bại.
- Nếu $x^\wedge.left^\wedge.rightmost < a$ thì nhánh con trái của x^\wedge 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:

```

//Tim 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 trực số, khoảng đóng thứ i là $[s_i, f_i]$ ($s_i \leq f_i$), hãy tìm một điểm trên trực 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, gấp đầu mút mở thì biến đếm tăng 1 còn gấp 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.

trục 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 trực số).

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

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

$$x^{\wedge}.sum := x^{\wedge}.left^{\wedge}.sum + x^{\wedge}.sign + x^{\wedge}.right^{\wedge}.sum \quad (7.3)$$

Thông tin phụ trợ thứ ba trong mỗi nút x^{\wedge} là trường *maxsum*: Cho biết nếu ta duyệt nhánh cây gốc x^{\wedge} 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:

$$\begin{aligned} & x^{\wedge}.maxsum \\ &:= \max \begin{cases} x^{\wedge}.left^{\wedge}.maxsum \\ x^{\wedge}.left^{\wedge}.sum + x^{\wedge}.sign \\ x^{\wedge}.Left^{\wedge}.sum + x^{\wedge}.sign + x^{\wedge}.right^{\wedge}.maxsum \end{cases} \end{aligned} \quad (7.4)$$

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

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

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 trùu tượng các khoảng đóng tương ứng với chèn một đầu mứt s với $sign = 1$ vào BST và chèn một đầu mứt f với $sign = -1$ vào BST. Thời gian $O(\lg n)$.
- Xóa một đoạn $[s, f]$ tương ứng với xóa đi đầu mứt s và đầu mứt 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

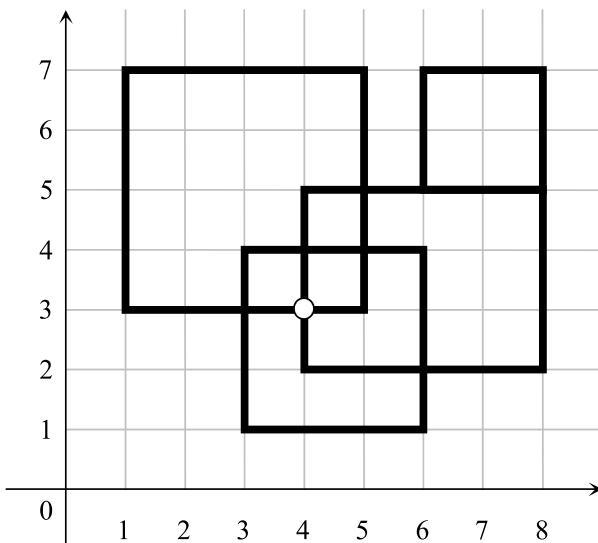
- Dòng 1 chứa số nguyên dương $n \leq 10^6$.

- 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
<pre>4 1 3 5 7 3 1 6 4 4 2 8 5 6 5 8 7</pre>	point of Maximum Overlap: (4, 3) Number of Rectangles: 3



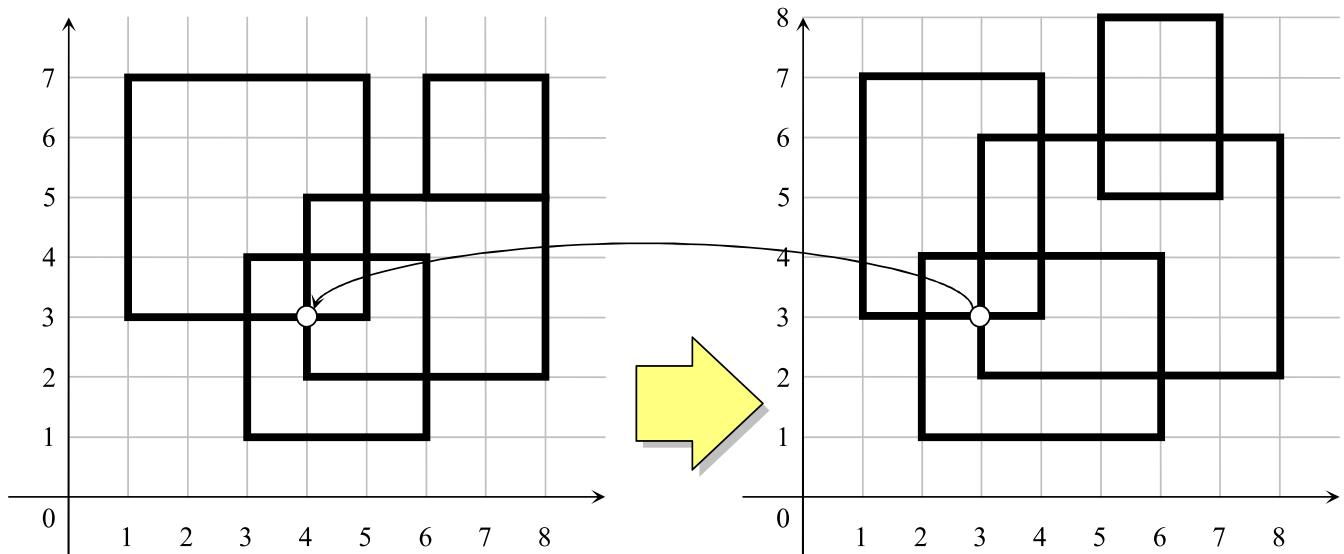
□ Biến đổi tọa độ

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

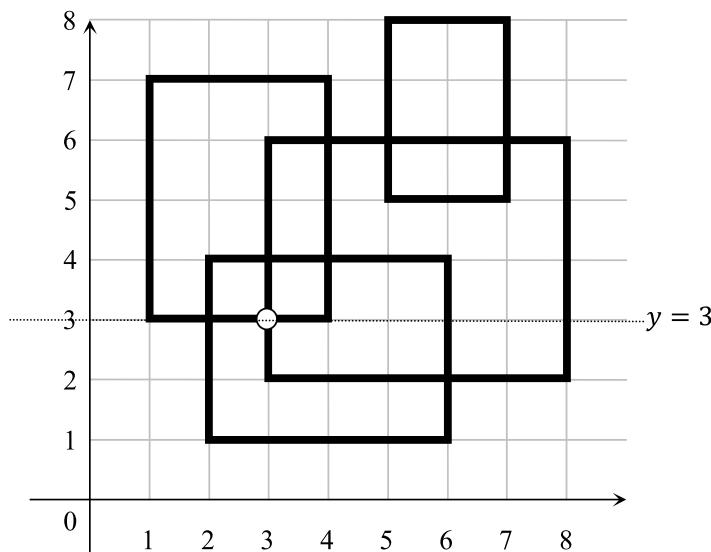
Phép biến đổi này có những công dụng:

- 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 nhau 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 $1, 2, \dots, 2n$, 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 $tree[0 \dots 2n]$, 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; //Tim 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; //Xây dựng 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;
        end;

```

```

    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; //Di 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 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 trống 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 trống cây?
- Nếu không muốn giải phóng ô trống 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 $n = 6$

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

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 $S(5,4)$.