

- ❖ Nếu $k[\text{median}] < X$ thì có nghĩa là đoạn từ $k[\text{inf}]$ tới $k[\text{median}]$ chỉ chứa toàn khoá $< X$, ta tiếp tục hành tìm kiếm tiếp với đoạn từ $k[\text{median}+1]$ tới $k[\text{sup}]$.
- ❖ Nếu $k[\text{median}] > X$ thì có nghĩa là đoạn từ $k[\text{median}]$ tới $k[\text{sup}]$ chỉ chứa toàn khoá $> X$, ta tiếp tục hành tìm kiếm tiếp với đoạn từ $k[\text{inf}]$ tới $k[\text{median}-1]$.
- ❖ Nếu $k[\text{median}] = X$ thì việc tìm kiếm thành công (kết thúc quá trình tìm kiếm).

Quá trình tìm kiếm sẽ thất bại nếu đến một bước nào đó, đoạn tìm kiếm là rỗng ($\text{inf} > \text{sup}$).

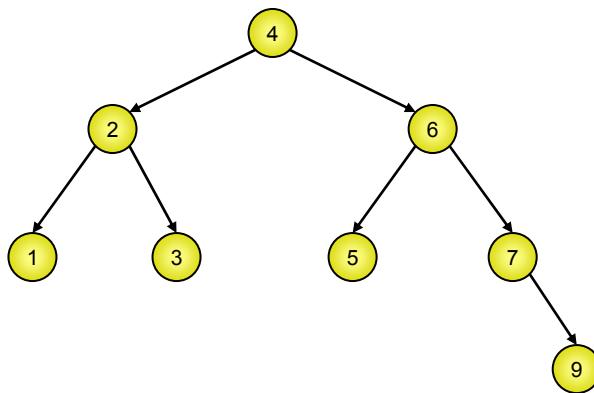
```
{Tìm kiếm nhị phân trên dãy khoá k[1] ≤ k[2] ≤ ... ≤ k[n]; hàm này thử tìm xem trong dãy có khoá nào = X không, nếu thấy nó
trả về chỉ số của khoá ấy, nếu không thấy nó trả về 0}
function BinarySearch(X: TKey): Integer;
var
  inf, sup, median: Integer;
begin
  inf := 1; sup := n;
  while inf ≤ sup do
    begin
      median := (inf + sup) div 2;
      if k[median] = X then
        begin
          BinarySearch := median;
          Exit;
        end;
      if k[median] < X then inf := median + 1
      else sup := median - 1;
    end;
  BinarySearch := 0;
end;
```

Người ta đã chứng minh được độ phức tạp tính toán của thuật toán tìm kiếm nhị phân trong trường hợp tốt nhất là $O(1)$, trong trường hợp xấu nhất là $O(\lg n)$ và trong trường hợp trung bình là $O(\lg n)$. Tuy nhiên, ta không nên quên rằng trước khi sử dụng tìm kiếm nhị phân, dãy khoá phải được sắp xếp rồi, tức là thời gian chi phí cho việc sắp xếp cũng phải tính đến. Nếu dãy khoá luôn luôn biến động bởi phép bổ sung hay loại bỏ đi thì lúc đó chi phí cho sắp xếp lại nổi lên rất rõ làm bộc lộ nhược điểm của phương pháp này.

9.4. CÂY NHỊ PHÂN TÌM KIẾM (BINARY SEARCH TREE - BST)

Cho n khoá $k[1..n]$, trên các khoá có quan hệ thứ tự toàn phần. Cây nhị phân tìm kiếm ứng với dãy khoá đó là một cây nhị phân mà mỗi nút chứa giá trị một khoá trong n khoá đã cho, hai giá trị chứa trong hai nút bất kỳ là khác nhau. Đối với mọi nút trên cây, tính chất sau luôn được thoả mãn:

- ❖ Mọi khoá nằm trong cây con trái của nút đó đều nhỏ hơn khoá ứng với nút đó.
- ❖ Mọi khoá nằm trong cây con phải của nút đó đều lớn hơn khoá ứng với nút đó

**Hình 38: Cây nhị phân tìm kiếm**

Thuật toán tìm kiếm trên cây có thể mô tả chung như sau:

Trước hết, khoá tìm kiếm X được so sánh với khoá ở gốc cây, và 4 tình huống có thể xảy ra:

- ❖ Không có gốc (cây rỗng): X không có trên cây, phép tìm kiếm thất bại
- ❖ X trùng với khoá ở gốc: Phép tìm kiếm thành công
- ❖ X nhỏ hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con trái của gốc với cách làm tương tự
- ❖ X lớn hơn khoá ở gốc, phép tìm kiếm được tiếp tục trong cây con phải của gốc với cách làm tương tự

Giả sử cấu trúc một nút của cây được mô tả dưới dạng:

```

type
  PNode = ^TNode; {Con trỏ chứa liên kết tới một nút}
  TNode = record {Cấu trúc nút}
    Info: TKey; {Trường chứa khoá}
    Left, Right: PNode; {con trỏ tới nút con trái và phải, trỏ tới nil nếu không có nút con trái (phải)}
  end;
  Gốc của cây được lưu trong con trỏ Root. Cây rỗng thì Root = nil
  
```

Thuật toán tìm kiếm trên cây nhị phân tìm kiếm có thể viết như sau:

```
{Hàm tìm kiếm trên BST, nó trả về nút chứa khoá tìm kiếm X nếu tìm thấy, trả về nil nếu không tìm thấy}
function BSTSearch(X: TKey): PNode;
```

```

var
  p: PNode;
begin
  p := Root; {Bắt đầu với nút gốc}
  while p ≠ nil do
    if X = p^.Info then Break;
    else
      if X < p^.Info then p := p^.Left
      else p := p^.Right;
  BSTSearch := p;
end;
```

Thuật toán dựng cây nhị phân tìm kiếm từ dãy khoá k[1..n] cũng được làm gần giống quá trình tìm kiếm. Ta chèn lần lượt các khoá vào cây, trước khi chèn, ta tìm xem khoá đó đã có trong cây hay chưa, nếu đã có rồi thì bỏ qua, nếu nó chưa có thì ta thêm nút mới chứa khoá cần chèn và nối nút đó vào cây nhị phân tìm kiếm tại mỗi liên kết vừa rẽ sang khiến quá trình tìm kiếm thất bại.

```

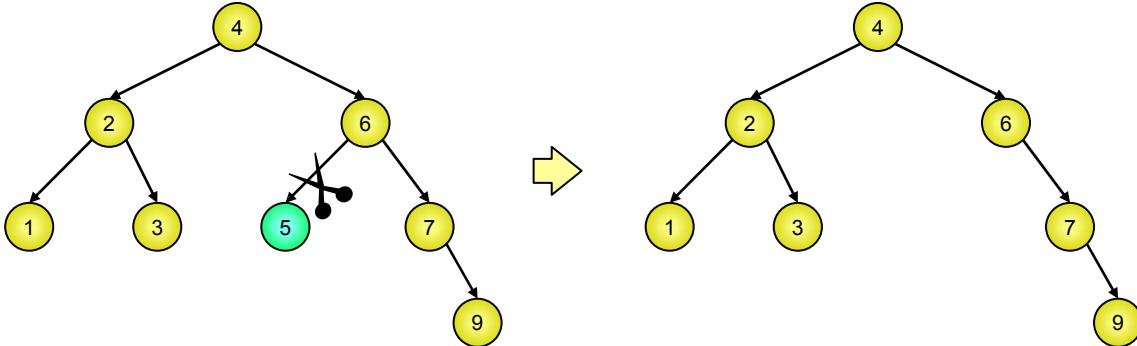
{Thủ tục chèn khoá X vào BST}
procedure BSTInsert(X);
var
  p, q: PNode;
begin
  q := nil; p := Root; {Bắt đầu với p = nút gốc; q là con trỏ chạy đuôi theo sau}
  while p ≠ nil do
    begin
      q := p;
      if X = p^.Info then Break;
      else {X ≠ p^.Info thì cho p chạy sang nút con, q^ luôn giữ vai trò là cha của p^}
        if X < p^.Info then p := p^.Left
        else p := p^.Right;
    end;
  if p = nil then {Khoá X chưa có trong BST}
    begin
      New(p); {Tạo nút mới}
      p^.Info := X; {Đưa giá trị X vào nút mới tạo ra}
      p^.Left := nil; p^.Right := nil; {Nút mới khi chèn vào BST sẽ trở thành nút lá}
      if Root = nil then Root := NewNode {BST đang rỗng, đặt Root là nút mới tạo}
      else {Móp NewNode^ vào nút cha q^}
        if X < q^.Info then q^.Left := NewNode
        else q^.Right := NewNode;
    end;
  end;
end;

```

Phép loại bỏ trên cây nhị phân tìm kiếm không đơn giản như phép bổ sung hay phép tìm kiếm.

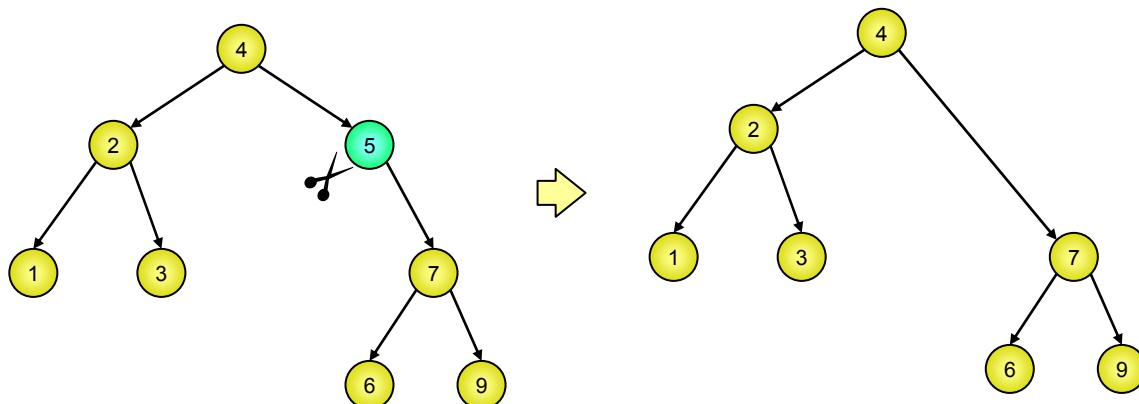
Muốn xoá một giá trị trong cây nhị phân tìm kiếm (Tức là dựng lại cây mới chứa tất cả những giá trị còn lại), trước hết ta tìm xem giá trị cần xoá nằm ở nút D nào, có ba khả năng xảy ra:

- ❖ Nút D là nút lá, trường hợp này ta chỉ việc đem mối nối cũ trỏ tới nút D (từ nút cha của D) thay bởi nil, và giải phóng bộ nhớ cấp cho nút D (Hình 39).



Hình 39: Xóa nút lá ở cây BST

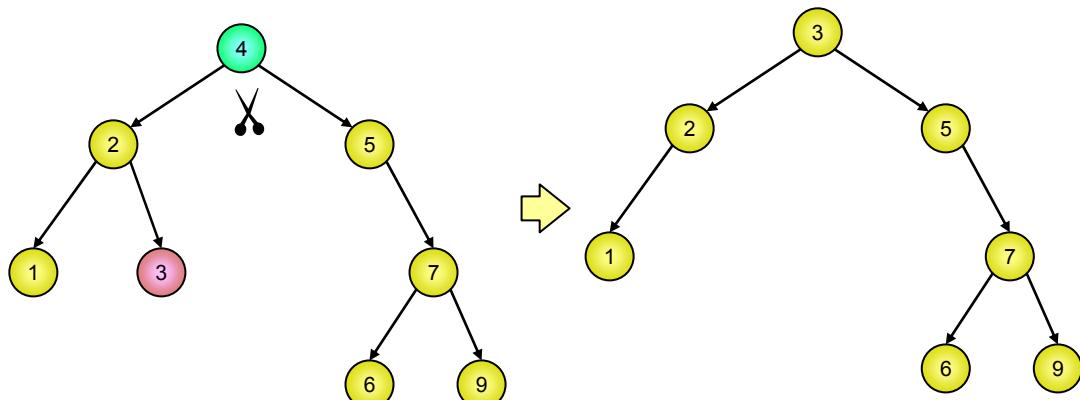
- ❖ Nút D chỉ có một nhánh con, khi đó ta đem nút gốc của nhánh con đó thế vào chỗ nút D, tức là chỉnh lại mối nối: Từ nút cha của nút D không nối tới nút D nữa mà nối tới nhánh con duy nhất của nút D. Cuối cùng, ta giải phóng bộ nhớ đã cấp cho nút D (Hình 40)



Hình 40. Xóa nút chỉ có một nhánh con trên cây BST

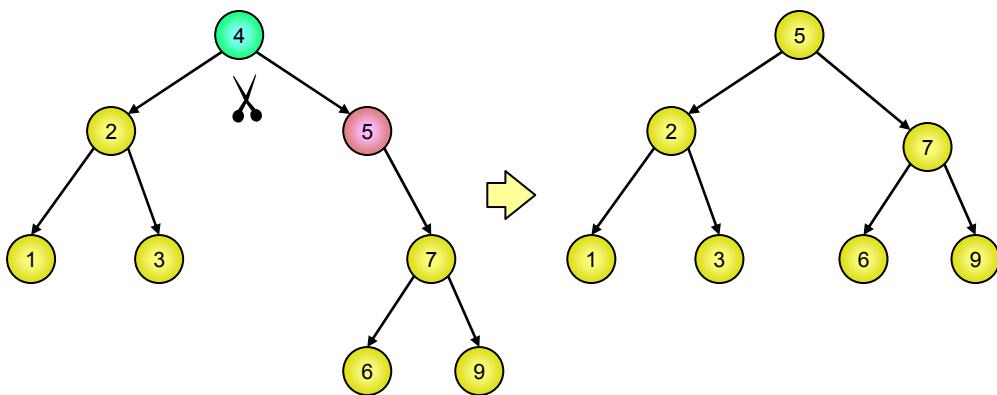
❖ Nút D có cả hai nhánh con trái và phải, khi đó có hai cách làm đều hợp lý cả:

Hoặc tìm nút chứa khoá lớn nhất trong cây con trái, đưa giá trị chứa trong đó sang nút D, rồi xoá nút này. Do tính chất của cây BST, nút chứa khoá lớn nhất trong cây con trái chính là nút cực phải của cây con trái nên nó không thể có hai con được, việc xoá đưa về hai trường hợp trên (Hình 41)



Hình 41: Xóa nút có cả hai nhánh con trên cây BST thay bằng nút cực phải của cây con trái

Hoặc tìm nút chứa khoá nhỏ nhất trong cây con phải, đưa giá trị chứa trong đó sang nút D, rồi xoá nút này. Do tính chất của cây BST, nút chứa khoá nhỏ nhất trong cây con phải chính là nút cực trái của cây con phải nên nó không thể có hai con được, việc xoá đưa về hai trường hợp trên.



Hình 42: Xóa nút có cả hai nhánh con trên cây BST thay bằng nút cực trái của cây con phải

```

{Thủ tục xoá khoá X khỏi BST}
procedure BSTDelete(X: TKey);
var
  p, q, Node, Child: PNode;
begin
  p := Root; q := nil; {Về sau, khi p trỏ sang nút khác, ta luôn giữ cho q^ luôп là cha của p^}
  while p ≠ nil do {Tìm xem trong cây có khoá X không?}
    begin
      if p^.Info = X then Break; {Tìm thấy}
      q := p;
      if X < p^.Info then p := p^.Left
      else p := p^.Right;
    end;
  if p = nil then Exit; {X không tồn tại trong BST nên không xoá được}
  if (p^.Left ≠ nil) and (p^.Right ≠ nil) then {p^ có cả con trái và con phải}
    begin
      Node := p; {Giữ lại nút chứa khoá X}
      q := p; p := p^.Left; {Chuyển sang nhánh con trái để tìm nút cực phải}
      while p^.Right ≠ nil do
        begin
          q := p; p := p^.Right;
        end;
      Node^.Info := p^.Info; {Chuyển giá trị từ nút cực phải trong nhánh con trái lên Node^}
    end;
  {Nút bị xoá giờ đây là nút p^, nó chỉ có nhiều nhất một con}
  {Nếu p^ có một nút con thì đem Child trỏ tới nút con đó, nếu không có thì Child = nil}
  if p^.Left ≠ nil then Child := p^.Left
  else Child := p^.Right;
  if p = Root then Root := Child; {Nút p^ bị xoá là gốc cây}
  else {Nút bị xoá p^ không phải gốc cây thì lấy mồi nối từ cha của nó là q^ nối thẳng tới Child}
    if q^.Left = p then q^.Left := Child
    else q^.Right := Child;
  Dispose(p);
end;

```

Trường hợp trung bình, thì các thao tác tìm kiếm, chèn, xoá trên BST có độ phức tạp là $O(\lg n)$. Còn trong trường hợp xấu nhất, cây nhị phân tìm kiếm bị suy biến thì các thao tác đó đều có độ phức tạp là $O(n)$, với n là số nút trên cây BST.

Nếu ta mở rộng hơn khái niệm cây nhị phân tìm kiếm như sau: Giá trị lưu trong một nút lớn hơn **hoặc bằng** các giá trị lưu trong cây con trái và nhỏ hơn các giá trị lưu trong cây con phải. Thì chỉ cần sửa đổi thủ tục BSTInsert một chút, khi chèn lần lượt vào cây n giá trị, cây BST sẽ có n nút (có thể có hai nút chứa cùng một giá trị). Khi đó nếu ta duyệt các nút của cây theo

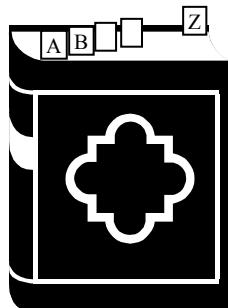
kiểu trung thứ tự (inorder traversal), ta sẽ liệt kê được các giá trị lưu trong cây theo thứ tự tăng dần. Phương pháp sắp xếp này người ta gọi là Tree Sort. Độ phức tạp tính toán trung bình của Tree Sort là $O(n \lg n)$.

Phép tìm kiếm trên cây BST sẽ kém hiệu quả nếu như cây bị suy biến, người ta có nhiều cách xoay xở để tránh trường hợp này. Đó là phép quay cây để dựng cây nhị phân cân đối AVL, hay kỹ thuật dựng cây nhị phân tìm kiếm tối ưu. Những kỹ thuật này ta có thể tham khảo trong các tài liệu khác về cấu trúc dữ liệu và giải thuật.

9.5. PHÉP BĂM (HASH)

Tư tưởng của phép băm là dựa vào giá trị các khoá $k[1..n]$, chia các khoá đó ra thành các nhóm. **Những khoá thuộc cùng một nhóm có một đặc điểm chung** và đặc điểm này không có trong các nhóm khác. Khi có một khoá tìm kiếm X, trước hết ta xác định xem X thuộc vào dãy khoá đã cho thì nó phải thuộc nhóm nào và tiến hành tìm kiếm trên nhóm đó.

Một ví dụ là trong cuốn từ điển, các bạn sinh viên thường dán vào 26 mảnh giấy nhỏ vào các trang để đánh dấu trang nào là trang khởi đầu của một đoạn chứa các từ có cùng chữ cái đầu. Để khi tra từ chỉ cần tìm trong các trang chứa những từ có cùng chữ cái đầu với từ cần tìm.



Một ví dụ khác là trên dãy các khoá số tự nhiên, ta có thể chia nó là làm m nhóm, mỗi nhóm gồm các khoá đồng dư theo mô-đun m.

Có nhiều cách cài đặt phép băm:

- ❖ Cách thứ nhất là chia dãy khoá làm các đoạn, mỗi đoạn chứa những khoá thuộc cùng một nhóm và ghi nhận lại vị trí các đoạn đó. Để khi có khoá tìm kiếm, có thể xác định được ngay cần phải tìm khoá đó trong đoạn nào.
- ❖ Cách thứ hai là chia dãy khoá làm m nhóm, Mỗi nhóm là một danh sách nối đơn chứa các giá trị khoá và ghi nhận lại chốt của mỗi danh sách nối đơn. Với một khoá tìm kiếm, ta xác định được phải tìm khoá đó trong danh sách nối đơn nào và tiến hành tìm kiếm tuần tự trên danh sách nối đơn đó. Với cách lưu trữ này, việc bổ sung cũng như loại bỏ một giá trị khỏi tập hợp khoá dễ dàng hơn rất nhiều phương pháp trên.
- ❖ Cách thứ ba là nếu chia dãy khoá làm m nhóm, mỗi nhóm được lưu trữ dưới dạng cây nhị phân tìm kiếm và ghi nhận lại gốc của các cây nhị phân tìm kiếm đó, phương pháp này có thể nói là tốt hơn hai phương pháp trên, tuy nhiên dãy khoá phải có quan hệ thứ tự toàn phần thì mới làm được.

Cây quản lý phạm vi

Lê Minh Hoàng (ĐHSPHN)

Chuyên đề này giới thiệu cây quản lý phạm vi (range trees). Đây là một cấu trúc dữ liệu quan trọng, có nhiều ứng dụng trong các thuật toán hình học, truy vấn cơ sở dữ liệu và xử lý tín hiệu. Cây quản lý phạm vi được đề cập nhiều trong các kỳ thi olympic tin học trong khoảng 10 năm trở lại đây.

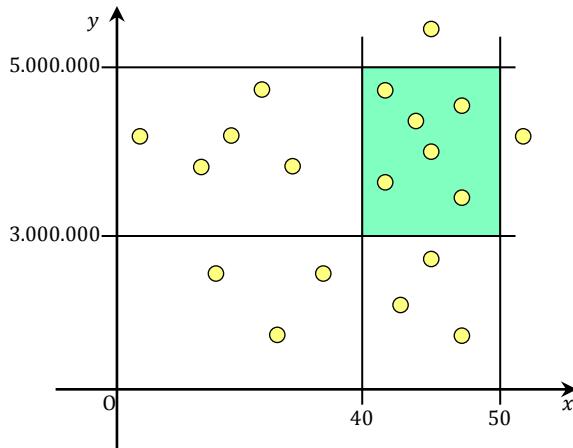
1. Giới thiệu

Giả sử ta có một cơ sở dữ liệu chứa hồ sơ cá nhân của các nhân viên trong một công ty, thông tin về mỗi nhân viên là một bản ghi gồm nhiều trường: tên, địa chỉ, ngày sinh, lương, cấp bậc, v.v... Cơ sở dữ liệu này luôn biến động bởi các lệnh thêm/bớt/cập nhật bản ghi, đi kèm với truy vấn dạng như: “Đếm số người ở trong độ tuổi từ 40 tới 50 và có mức lương tháng từ 3 triệu tới 5 triệu” hay “Tính tổng lương của những người có thâm niên công tác từ 10 năm trở lên và có từ 2 tới 4 con”...

Dạng bài toán quản lý đối tượng và trả lời các dạng truy vấn kể trên có thể mô hình hóa dưới dạng hình học: Chẳng hạn ta có thể biểu diễn mỗi nhân viên bởi một điểm trên mặt phẳng tọa độ Oxy với trục tọa độ Ox là tuổi và trục tọa độ Oy là mức lương tháng. Khi đó truy vấn “đếm số người trong độ tuổi từ 40 tới 50 và có mức lương tháng từ 3 tới 5 triệu” có thể diễn giải thành: đếm số điểm có tọa độ

$$(x, y) \in [40,50] \times [3.000.000,5.000.000]$$

(Xem Hình 1)



Hình 1

Bài toán truy vấn phạm vi có thể phát biểu hình thức như sau: Ta cần quản lý một tập S các đối tượng biểu diễn bởi các điểm trong không gian d chiều, mỗi điểm ngoài tọa độ của nó còn có thể chứa thêm một số thông tin khác*. Tập S liên tục biến động bởi phép thêm/bớt điểm đi kèm với các truy vấn. Mỗi truy vấn, cho bởi phạm vi là một siêu hộp $C = [L_1, H_1] \times [L_2, H_2] \times \dots \times [L_d, H_d]$, yêu cầu trả lời về thông tin tổng hợp từ tất cả các điểm nằm trong siêu hộp C . Trong chuyên đề này, ta quan tâm tới các dạng thông tin tổng hợp có tính chất “chia để trị”, tức là nếu A và B là hai tập điểm rời nhau thì thông tin tổng hợp từ các điểm $\in A \cup B$ có thể dễ dàng tính được từ thông tin tổng hợp trên A và thông tin tổng hợp trên B . Như ví dụ về cơ sở dữ liệu nhân sự, ta có thể dễ dàng tính được:

- Số người trong tập $A \cup B$ bằng cách lấy số người trong A cộng số người trong B .
- Lương của người có lương cao nhất trong tập $A \cup B$, $\max_S(A \cup B)$, có thể xác định bằng công thức $\max(\max_S(A), \max_S(B))$.

Sẽ không có gì đáng nói nếu tập S chỉ gồm ít điểm, khi đó ta có thể lọc tất cả những điểm nằm trong phạm vi truy vấn và tổng hợp thông tin từ những điểm đó. Tuy nhiên khi số điểm lớn, mặc dù phép thêm bớt điểm có thể thực hiện

* Trong CSDL quan hệ, một số cột sẽ được xác định làm tọa độ và những cột còn lại sẽ được coi là thông tin cần truy vấn tùy theo từng ứng dụng cụ thể.

nhanh (thời gian $O(1)$ nếu sử dụng cấu trúc dữ liệu danh sách thông thường), nhưng thời gian truy vấn trở nên rất chậm (mất $\Theta(nd)$ thời gian để trả lời mỗi truy vấn với n là số điểm trong S và d là số chiều của không gian).

Cây quản lý phạm vi là một cấu trúc dữ liệu hiệu quả để giải quyết bài toán trên, nó cho phép thực hiện mỗi phép thêm, bớt, cập nhật, truy vấn trong thời gian $O(\log^d n)$, thích hợp với xử lý dữ liệu lớn vì đại lượng $O(\log^d n)$ tăng rất chậm so với đại lượng $\Theta(nd)$ khi $n \rightarrow \infty$.

Trong các phần tiếp theo, phần 2 khảo sát một số cấu trúc dữ liệu và thuật toán giải quyết bài toán truy vấn phạm vi 1 chiều. Phần 3 mở rộng các cấu trúc dữ liệu cho truy vấn phạm vi nhiều chiều. Phần 4 là một số bài toán ứng dụng, mở rộng của cây quản lý phạm vi, cuối cùng là kết luận và danh mục tài liệu tham khảo.

2. Truy vấn phạm vi một chiều

2.1. Cây nhị phân tìm kiếm

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 và truy vấn đều được thực hiện trong thời gian $O(\log n)$. Nội dung phương pháp này được trình bày qua một bài toán cụ thể:

Bài toán (Range Query):

Cho một danh sách L để chứa các số nguyên. Ký hiệu $Length(L)$ là số phần tử trong danh sách. Các phần tử trong danh sách được đánh số liên tiếp bắt đầu từ 1.

Bắt đầu với một danh sách rỗng, xét các phép biến đổi danh sách sau đây:

- Phép chèn $Insert(i, v)$: Chèn một số v vào vị trí i của danh sách. Trường hợp $i > Length(L)$ thì v sẽ được thêm vào cuối danh sách.
- Phép xóa $Delete(i)$: Xóa phần tử thứ i trong danh sách.
- Phép cập nhật $Update(i, v)$: Đặt phần tử thứ i bằng v .
- Phép truy vấn $Query(i, j)$: Trả về tổng các phần tử nằm trong phạm vi từ i đến j

Yêu cầu: Cho dãy k phép biến đổi được thực hiện tuần tự, hãy trả lời tất cả các truy vấn $Query$

Input

- Dòng 1 chứa số nguyên dương $k \leq 10^5$
- k dòng tiếp, mỗi dòng cho thông tin về một phép biến đổi. Mỗi dòng bắt đầu bởi một ký tự $\in \{I, D, U, Q\}$
 - Nếu ký tự đầu dòng là “I” thì tiếp theo là hai số nguyên v, i tương ứng với phép chèn $Insert(v, i)$ ($v \leq 10^9$)
 - Nếu ký tự đầu dòng là “D” thì tiếp theo là số nguyên i tương ứng với phép xóa $Delete(i)$.
 - Nếu ký tự đầu dòng là “U” thì tiếp theo là hai số nguyên i, v tương ứng với phép cập nhật $Update(i, v)$ ($v \leq 10^9$)
 - Nếu ký tự đầu dòng là “Q” thì tiếp theo là hai số nguyên i, j tương ứng với phép truy vấn $Query(i, j)$ ($i < j$)

Output

Trả lời tất cả các truy vấn $Query$, với mỗi truy vấn in ra câu trả lời trên 1 dòng

Sample Input	Sample Output
11	13
I 1 8 {8}	18
I 1 3 {3 8}	25
I 3 6 {3 8 6}	
I 3 2 {3 8 2 6}	
Q 1 3	
U 3 4 {3 8 4 6}	
I 2 5 {3 5 8 4 6}	
D 1 {5 8 4 6}	
Q 2 4	
U 1 7 {7 8 4 6}	
Q 1 4	

2.1.1. Biểu diễn danh sách

Chúng ta sẽ lưu trữ các phần tử của danh sách L trong một cấu trúc cây nhị phân sao cho nếu duyệt cây theo *thứ tự giữa* thì các phần tử của L sẽ được liệt kê theo đúng thứ tự trong danh sách. Cây nhị phân này được cài đặt bởi một cấu trúc liên kết các nút động và con trỏ, ý tưởng chính của cách tiếp cận này là *đồng bộ thứ tự của danh sách trùu tượng L với thứ tự giữa của cây, quy việc chèn xóa trên L về việc chèn xóa trên cây nhị phân*.

Có một số điểm tương đồng giữa cấu trúc dữ liệu này và cây nhị phân tìm kiếm (*binary search trees – BST*):

- Nếu danh sách trùu tượng L gồm các phần tử đã sắp xếp tăng dần thì tự nhiên cây biểu diễn danh sách trở thành cây nhị phân tìm kiếm.
- Nếu duyệt cây nhị phân tìm kiếm theo thứ tự giữa, ta được dãy tăng dần các khóa tìm kiếm, còn nếu duyệt cây biểu diễn danh sách theo thứ tự giữa ta sẽ được danh sách L .
- Có thể có nhiều cấu trúc cây nhị phân tìm kiếm ứng với một tập khóa tìm kiếm, tương tự như vậy có nhiều cấu trúc cây biểu diễn danh sách ứng với danh sách L .
- Các phương pháp cân bằng cây biểu diễn danh sách có thể kế thừa từ cây nhị phân tìm kiếm bởi các kỹ thuật cân bằng cây nhị phân tìm kiếm luôn bảo tồn thứ tự giữa của các nút (tương ứng với thứ tự tăng dần của các khóa tìm kiếm). Đây cũng là lý do ta đồng bộ thứ tự danh sách với thứ tự giữa của các nút trên cây chứ không phải là thứ tự trước hay thứ tự sau.

Chính vì có nhiều điểm tương đồng giữa cấu trúc cây biểu diễn danh sách và cây nhị phân tìm kiếm, ta cũng dùng tên cây nhị phân tìm kiếm – *binary search trees (BST)* cho cấu trúc dữ liệu này.

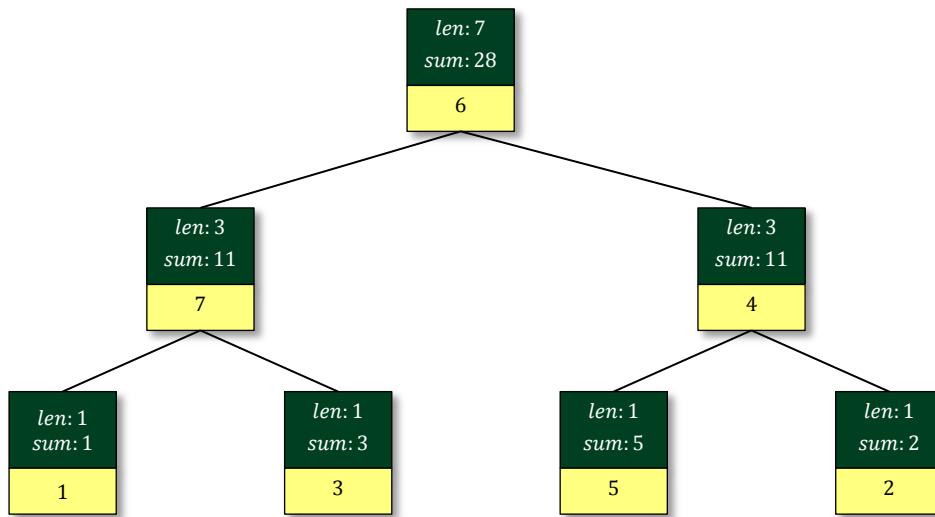
2.1.2. Cấu trúc nút

Có thể thấy rằng khi duyệt cây theo thứ tự giữa thì các nút một nhánh cây sẽ được liệt kê trong một đoạn tiếp, không có nút ở nhánh khác xen vào. Nói cách khác, bản chất mỗi nhánh cây là một đoạn các phần tử liên tiếp trong danh sách trùu tượng L . Điều đó gợi ý cho việc lưu trữ thêm những thông tin tổng hợp về đoạn liên tiếp này tại những nút trên cây, trong trường hợp

cụ thể này, mỗi nút x sẽ chứa thêm 2 thông tin: Số nút trong nhánh gốc x (dùng để thực hiện phép truy cập ngẫu nhiên) và tổng các phần tử trong nhánh gốc x (dùng để trả lời truy vấn tổng).

Tóm lại, mỗi nút trên cây là một bản ghi gồm các trường:

- Trường $value$: Chứa phần tử lưu trong nút.
- Trường P : Chứa liên kết (con trỏ) tới nút cha, nếu là nút gốc (không có nút cha) thì trường P được đặt bằng một con trỏ đặc biệt, ký hiệu $nilT$.
- Trường L : Chứa liên kết (con trỏ) tới nút con trái, nếu nút không có nhánh con trái thì trường L được đặt bằng $nilT$.
- Trường R : Chứa liên kết (con trỏ) tới nút con phải, nếu nút không có nhánh con phải thì trường R được đặt bằng $nilT$.
- Trường len chứa số nút trong nhánh gốc x
- Trường sum chứa tổng phần tử chứa trong nhánh gốc x



Hình 2. Cây biểu diễn danh sách gồm 7 phần tử $(1, 7, 3, 6, 5, 4, 2)$

```

type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record
    key: TKey;
    P, L, R: PNode;
    len: Integer;
    sum: Int64;
  end;
var
  sentinel: TNode;
  nilT: PNode; //Con trỏ tới nút đặc biệt
  root: PNode; //Con trỏ tới nút gốc
begin
  nilT := @sentinel;
  nilT^.size := 0;
  nilT^.sum := 0;
  ...
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 cây nhị phân, chúng ta sử dụng con trỏ *nilT* gán cho những liên kết không có thực (công dụng tương tự như con trỏ *nil*). Chỉ có khác là con trỏ *nilT* trỏ tới một biến **sentinel có thực** mặc dù các trường của *nilT*[^] là **vô nghĩa**. Chúng ta hy sinh một ô nhớ cho biến *sentinel* = *nilT*[^] để đơn giản hóa các thao tác trên cây*.

Thủ tục *Update(x)* sau đây tổng hợp các thông tin trong *x* từ những thông tin phụ trợ trong hai nút con:

```

procedure Update(x: PNode);
begin
  x^.len := x^.L^.len + x^.R^.len + 1; //Tính số nút trong nhánh x
  x^.sum := x^.L^.sum + x^.R^.sum + x^.value; //Tính tổng các phần tử trong nhánh x
end;

```

Ở đây ta quy ước *nilT*[^].*len* = 0 và *nilT*[^].*sum* = 0.

Để dễ trình bày các thao tác, ta viết sẵn các thủ tục mốc nối các nút: Thủ tục *SetL(x,y)* cho *y* làm con trái *x*, thủ tục *SetR(x,y)* cho *y* làm con phải *x*

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

```

procedure SetL(x, y: PNode);
begin
  y^.P := x; x^.L := y;
end;

procedure SetR(x, y: PNode);
begin
  y^.P := x; x^.R := y;
end;

```

2.1.3. Truy cập ngẫu nhiên

Cả các phép biến đổi danh sách đều có một tham số vị trí, vậy việc đầu tiên là phải xác định nút tương ứng với vị trí i trong danh sách trừu tượng là nút nào trong cây. Theo nguyên lý của phép duyệt cây theo thứ tự giữa (duyệt nhánh trái, thăm nút gốc, sau đó duyệt nhánh phải), thuật toán xác định nút tương ứng với vị trí i có thể diễn tả như sau:

Để tìm nút thứ i trong nhánh cây gốc x , trước hết ta xác định ord là số thứ tự của nút gốc x (theo thứ tự giữa): $ord = x^.L^.size + 1$, sau đó

- Nếu $i = ord$ thì nút cần tìm chính là nút x .
- Nếu $i < ord$ thì quy về tìm nút thứ i trong nhánh con trái của x .
- Nếu $i > ord$ thì quy về tìm nút thứ $i - ord$ trong nhánh con phải của x .

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 $NodeAt(x, i)$: Nhận vào nút x và một số nguyên i , trả về nút tương ứng với vị trí đó trên nhánh cây gốc x .

```

function NodeAt(x: PNode; i: Integer): PNode;
var
  ord: Integer;
begin
  repeat
    ord := x^.L^.len + 1; //Xác định số thứ tự nút x
    if ord = i then Break; //Nút cần tìm chính là x
    if i < ord then //Quy về tìm nút thứ i trong nhánh con trái
      x := x^.L
    else //Quy về tìm nút thứ i - ord trong nhánh con phải
      begin
        i := i - ord;
        x := x^.R;
      end;
  until False;
  Result := x;
end;

```

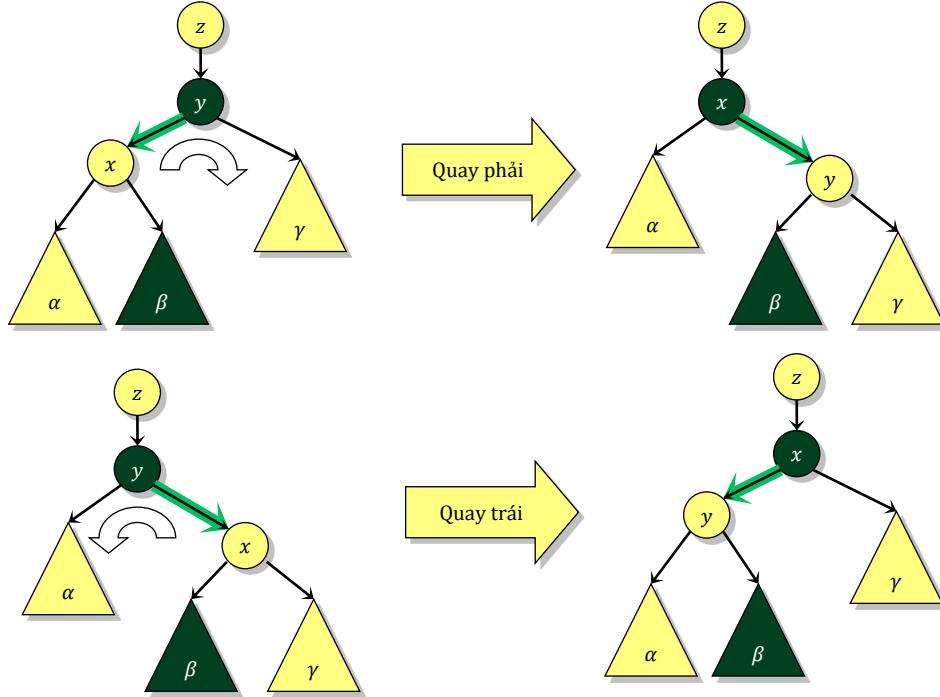
2.1.4. Vấn đề cân bằng cây

Thao tác truy cập ngẫu nhiên thực hiện một phép di chuyển từ gốc xuống một nút mang số thứ tự đã định, nếu cây nhị phân suy biến, phép truy cập ngẫu nhiên không khác gì phép truy cập tuần tự trên danh sách mốc nối đơn và trở nên kém hiệu quả. Rất may mắn là ta có những phương pháp cân bằng cây. Những phương pháp này hoặc giữ cho chiều cao cây là một đại lượng $O(\log n)$ với n là số nút trên cây, hoặc đảm bảo cho dãy m thao tác truy cập ngẫu nhiên được thực hiện trong thời gian $O((m + n) \log n)$. Những phương pháp cân bằng cây được kể thừa từ các cấu trúc dữ liệu cây nhị phân tìm kiếm tự cân bằng (*self-balancing binary search trees*) như cây AVL, cây đỏ đen, cây Splay, Treaps, cây nhị phân tìm kiếm ngẫu nhiên v.v....

Hầu hết các kỹ thuật cân bằng cây được thực hiện bởi phép quay cây, có hai loại: quay phải (*right rotation*) và quay trái (*left rotation*).

- Nếu x là con trái của y , phép quay phải theo liên kết $y \rightarrow x$ cắt nhánh con phải của x làm con trái của y sau đó cho y làm con phải của x . Nút x được đẩy lên làm gốc nhánh thay cho nút y . Như ví dụ ở Hình 3, có thể thấy rằng phép quay phải bảo tồn thứ tự giữa $\alpha - x - \beta - y - \gamma$.

- Ngược lại, nếu x là con phải của y , phép quay trái theo liên kết $y \rightarrow x$ với x cắt nhánh con trái của x làm con phải của y sau đó cho y làm con trái của x . Tương tự như phép quay phải, phép quay trái đẩy x lên làm nút nhánh thay cho nút y . Như ví dụ ở Hình 3, cũng có thể thấy rằng phép quay trái bảo tồn thứ tự giữa $\alpha - y - \beta - x - \gamma$.



Hình 3. Phép quay cây

Ta viết một thao tác $UpTree(x)$ tổng quát hơn: Với $x \neq root$, và $y = x^{\wedge}.P$, 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 . Chú ý là sau phép $UpTree(x)$, ta chỉ cần cập nhật thông tin phụ trợ của nút y và sau đó cập nhật thông tin phụ trợ của nút x (trường len và trường sum), thông tin phụ trợ trong các nút khác không thay đổi.

```

procedure UpTree(x: PNode);
var
  y, z: PNode;
begin
  y := x^.P; //y^ là nút cha của x^
  z := y^.P; //z^ là nút cha của y^
  if x = y^.L then //Quay phải
    begin
      SetL(y, x^.R); //Chuyển nhánh con phải của x^ sang làm con trái y^
      SetR(x, y); //Cho y^ làm con phải x^
    end
  else //Quay trái
    begin
      SetR(y, x^.L); //Chuyển nhánh con trái của x^ sang làm con phải y^
      SetL(x, y); //Cho y^ làm con trái x^
    end;
  //Mởc nối x^ vào làm con z^ thay cho y^
  if z^.L = y then SetL(z, x) else SetR(z, x);
  Update(y); Update(x)
end;

```

2.1.5. Cây Splay

Trong ví dụ cụ thể này, cây Splay được sử dụng làm cấu trúc dữ liệu cây biểu diễn danh sách. Đây là một trong những ý tưởng thú vị nhất về cây nhị phân tìm kiếm tự cân bằng [8]. Thao tác làm “bẹp” cây (splay) được thực hiện ngay khi có lệnh truy cập nút, làm giảm độ sâu của những nút truy cập thường xuyên. 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.

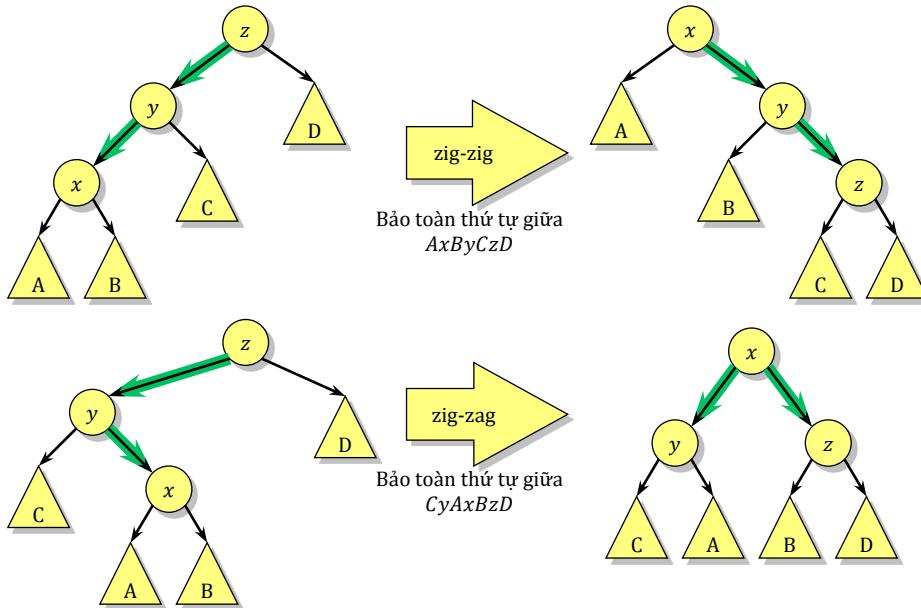
❑ Thao tác Splay

Thao tác quan trọng nhất của cây splay là thao tác *Splay(x)*: nhận vào một nút x và đẩy x lên làm gốc cây: Nếu x chưa phải gốc cây, gọi y là nút cha của x ($y := x^.P$) và z là nút cha của y ($z := y^.P$), x sẽ được đẩy dần lên gốc cây theo quy trình sau:

- Nếu x là con của nút gốc ($z = nilT$), một phép $UpTree(x)$ sẽ được thực hiện và x sẽ trở thành gốc cây. Thao tác này gọi là phép *zig*.
- Nếu x và y cùng là con trái hoặc cùng là con phải, một phép $UpTree(y)$ sẽ được thực hiện để đẩy y lên làm nút cha của z , tiếp theo là một phép $UpTree(x)$ để đẩy x lên làm nút cha của y . Thao tác này gọi là phép *zig-zig*.

- Nếu x và y có một nút là con trái và một nút là con phải, hai phép $UpTree(x)$ sẽ được thực hiện để đẩy x lên làm nút cha của cả y và z . Thao tác này gọi là phép zig-zag.

Thao tác $Splay(x)$ cần sử dụng tối đa một phép zig.



Hình 4. Phép zig-zig và zig-zag.

```

procedure Splay(x: PNode);
var
  y, z: PNode;
begin
  repeat
    y := x^.P;
    if y = nilT then Break; //x là gốc thì dừng ngay
    z := y^.P;
    if z ≠ nilT then //zig-zig hoặc zig-zag, cần 1 phép UpTree trước phép UpTree(x)
      if (y = z^.L) = (x = y^.L) then UpTree(y) //x và y cùng là con trái hoặc cùng là con phải
      else UpTree(x); //x và y có một nút con trái và một nút con phải
    UpTree(x);
  until False;
end;

```

Tại sao lại làm phức tạp vấn đề khi mà ta có thể thực hiện liên tiếp một loạt các thao tác $UpTree(x)$ để đẩy x lên gốc cây?. Câu trả lời là phép $Splay(x)$ ngoài việc đẩy x lên thành gốc cây, nó còn có xu hướng làm giảm độ sâu của các nút nằm trên đường đi từ x lên gốc. Đây là đặc điểm nổi bật của

cây Splay: Những nút truy cập thường xuyên sẽ được làm giảm độ sâu và phân bố gần gốc.

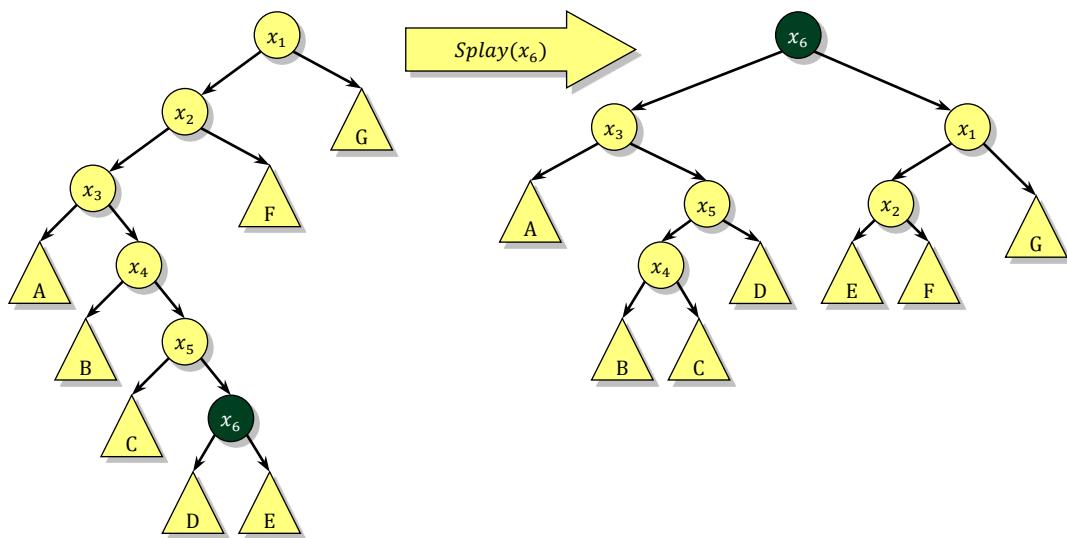
Hình 5 là ví dụ về cây trước và sau khi thực hiện thao tác $Splay(x_6)$. Thao tác $Splay(x_6)$ bao gồm một phép zig-zig, một phép zig-zag và một phép zig.

Trước khi $Splay(x_6)$, x_1 ở độ sâu 0, x_2 ở độ sâu 1, ...Tổng độ sâu của các nút $x_{1...6}$ là:

$$0 + 1 + 2 + 3 + 4 + 5 = 15$$

Sau khi $Splay(x_6)$, x_6 ở độ sâu 0, x_1 và x_3 ở độ sâu 1, x_2 và x_5 ở độ sâu 2, x_4 ở độ sâu 3. Tổng độ sâu của các nút $x_{1...6}$ là:

$$0 + 1 + 1 + 2 + 2 + 3 = 9$$



Hình 5. Ví dụ về thao tác Splay

□ Tách

Phép tách (*Split*) nhận vào một danh sách trừu tượng L (biểu diễn bởi cây T) và một chỉ số i , sau đó tách danh sách L thành hai danh sách: Danh sách L_1 (biểu diễn bởi cây T_1) gồm i phần tử đầu của L và danh sách L_2 (biểu diễn bởi cây T_2) gồm các phần tử còn lại trong L . Thuật toán tách có thể diễn giải như sau:

- Nếu $i = 0$, ta gán $T_1 := nilT$ và $T_2 := T$ bởi trong trường hợp này danh sách $L_1 = \emptyset$ và $L_2 = L$.

- Nếu $i > 0$, xác định nút T_1 chứa phần tử đứng thứ i bằng hàm $NodeAt(T, i)$, đẩy T_1 lên gốc cây bằng lệnh $Splay(T_1)$, và xác định T_2 là nút con phải của T_1 . Vì thứ tự giữa được bảo tồn, nút T_1 vẫn là nút đứng thứ i theo thứ tự giữa. Việc cuối cùng chỉ là cắt bỏ liên kết giữa T_1 và nhánh con phải T_2 của nó để tạo thành hai cây.

Chú ý rằng sau phép tách cây T thành hai cây T_1 và T_2 , ta phải coi như cây T đã bị hủy và không được sử dụng nữa.

```

procedure Split(T: PNode; i: Integer; var T1, T2: PNode);
begin
  if i = 0 then
    begin
      T1 := nilT; T2 := T;
    end
  else
    begin
      T1 := NodeAt(T, i); //T1: nút đứng thứ i theo thứ tự giữa
      Splay(T1); //Đẩy T1 lên làm gốc cây
      T2 := T1^.R; //T2 = con phải T1
      T1^.R := nilT; T2^.P := nilT; //Cắt liên kết T1 - T2
      Update(T1); //Tính lại thông tin phụ trợ trong T1 sau khi cắt liên kết
    end;
end;

```

□ Ghép

Việc ghép nối tiếp danh sách L_1 (biểu diễn bởi cây T_1) với danh sách L_2 (biểu diễn bởi cây T_2) được thực hiện như sau:

- Nếu $T_1 = nilT$, danh sách L_1 rỗng, đơn giản ta trả về T_2 .
- Nếu $T_1 \neq nilT$, ta xác định nút cực phải của T_1 (nút chứa phần tử đứng cuối danh sách L_1), tạm gọi là nút x , đẩy x lên gốc cây bằng phép $Splay(x)$ khi đó x chắc chắn không có nhánh con phải. Việc cuối cùng chỉ là cho T_2 làm con phải của x và trả về cây gốc x .

```

function Join(T1, T2: PNode): PNode;
begin
  if T1 = nilT then Result := T2 //T1 rỗng thì trả về T2
  else
    begin
      while T1^.R <> nilT do T1 := T1^.R; //Đi xuống nút cực phải T1
      Splay(T1); //Đẩy lên gốc cây
      SetR(T1, T2); //Cho T2 làm con phải T1
      Update(T1); //Cập nhật lại thông tin phụ trợ trong T1
      Result := T1; //Trả về cây biểu diễn danh sách đã ghép
    end;
end;

```

Chú ý rằng sau phép ghép cây T_1 và T_2 thành một cây T , cây T_1 và T_2 phải coi như đã bị hủy và không được sử dụng nữa.

❑ Chèn, xóa, cập nhật và truy vấn

Đến đây ta có thể dễ dàng cài đặt các phép biến đổi đã cho trên danh sách dựa trên cơ chế tách và ghép.

Để chèn phần tử v vào vị trí i , ta tách cây biểu diễn danh sách thành hai cây T_1, T_2 . Cây T_1 biểu diễn danh sách gồm $i - 1$ phần tử đầu tiên và cây T_2 biểu diễn danh sách gồm các phần tử còn lại. Tạo nút x chứa phần tử v và cho T_1, T_2 lần lượt là con trái và con phải nút x . Việc cuối cùng là cập nhật lại thông tin phụ trợ trong nút x và trả về cây gốc x biểu diễn danh sách mới sau khi đã chèn phần tử v .

Để xóa phần tử tại vị trí i , ta xác định nút x chứa phần tử đứng thứ i , thực hiện $Splay(x)$ để đẩy x lên gốc cây, cắt rời nhánh con trái T_1 và nhánh con phải T_2 của x sau đó thực hiện ghép T_1 với T_2 được cây biểu diễn danh sách mới sau khi đã xóa phần tử thứ i . Việc cuối cùng là giải phóng bộ nhớ cấp cho nút x .

Để cập nhật giá trị phần tử tại vị trí i , ta xác định nút x chứa phần tử đứng thứ i , đẩy x lên gốc cây bằng lệnh $Splay(x)$, sau đó cập nhật giá trị phần tử lưu trong x đồng thời cập nhật lại các thông tin phụ trợ của nút x .

Để tính tổng các phần tử từ vị trí i tới vị trí j , ta tách cây biểu diễn danh sách thành ba cây: Cây T_1 biểu diễn danh sách gồm $i - 1$ phần tử đầu dãy, cây T_2 biểu diễn danh sách gồm các phần tử nằm trong phạm vi truy vấn từ i tới j và cây T_3 biểu diễn danh sách từ phần tử thứ $j + 1$ đến cuối. Cuối cùng ta trả về giá trị $T_2^.sum$ và tiến hành ghép 3 cây lại như cũ. Thực ra phép truy vấn này

có thể trả lời mà không cần dùng thao tác tách ghép, nhưng ta tận dụng các thao tác tách ghép để đơn giản hóa việc cài đặt.

2.1.6. Cài đặt



LISTQUERY.PAS ✓ Cây biểu diễn danh sách

```
{$MODE OBJFPC}
program SumQueries;
type
  PNode = ^TNode;
  TNode = record //Cấu trúc nút
    value: Integer;
    P, L, R: PNode;
    len: Integer;
    sum: Int64;
  end;
var
  sentinel: TNode;
  root, nilT: PNode;

procedure Init;
begin
  nilT := @sentinel;
  nilT^.len := 0;
  nilT^.sum := 0;
  root := nilT;
end;

procedure Update(x: PNode); //Cập nhật thông tin phụ trợ trong nút x từ hai nút con
begin
  x^.len := x^.L^.len + x^.R^.len + 1;
  x^.sum := x^.L^.sum + x^.R^.sum + x^.value;
end;

procedure SetL(x, y: PNode); //Đặt y làm con trái x
begin
  y^.P := x; x^.L := y;
end;

procedure SetR(x, y: PNode); //Đặt y làm con phải x
begin
  y^.P := x; x^.R := y;
end;

//Tim nút thứ i trong cây gốc x
function NodeAt(x: PNode; i: Integer): PNode;
var
  ord: Integer;
begin
  begin
    repeat
      ord := x^.L^.len + 1;
      if ord = i then Break;
      if i < ord then
        x := x^.L
    until Break;
  end;
  result := x;
end;
```

```

    else
      begin
        i := i - ord;
        x := x^.R;
      end;
    until False;
  Result := x;
end;

procedure UpTree(x: PNode); //Đẩy x lên cao hơn 1 mức bằng phép quay
var
  y, z: PNode;
begin
  y := x^.P;
  z := y^.P;
  if x = y^.L then //Quay phải
    begin
      SetL(y, x^.R);
      SetR(x, y);
    end
  else //Quay trái
    begin
      SetR(y, x^.L);
      SetL(x, y);
    end;
  if z^.L = y then SetL(z, x) else SetR(z, x);
  Update(y); Update(x) //Cập nhật thông tin phụ trợ trong x và y
end;

procedure Splay(x: PNode); //Đẩy x lên thành gốc cây
var
  y, z: PNode;
begin
  repeat
    y := x^.P;
    if y = nilT then Break;
    z := y^.P;
    if z <> nilT then
      if (y = z^.L) = (x = y^.L) then UpTree(y)
      else UpTree(x);
    UpTree(x);
  until False;
end;

//Tách cây T thành 2 cây T1, T2, cây T1 gồm i phần tử đầu, cây T2 gồm các phần tử còn lại
procedure Split(T: PNode; i: Integer; var T1, T2: PNode);
begin
  if i = 0 then //i=0, cây T1 rỗng
    begin
      T1 := nilT; T2 := T;
    end
  else
    begin
      T1 := NodeAt(T, i); //Nhảy đến nút thứ i
      Splay(T1); //Đẩy lên gốc
      T2 := T1^.R; //Cắt nhánh con phải ra làm T2
      T1^.R := nilT; T2^.P := nilT;
    end;
end;

```

```

        Update(T1); //Cập nhật thông tin phụ trợ trong T1
    end;
end;

function Join(T1, T2: PNode): PNode; //Ghép nối tiếp hai danh sách T1, T2
begin
    if T1 = nilT then Result := T2
    else
        begin
            while T1^.R <> nilT do T1 := T1^.R; //Nhảy đến nút cực phải của T1
            Splay(T1); //Đẩy lên gốc
            SetR(T1, T2); //Cho T2 làm con phải
            Update(T1); //Cập nhật thông tin phụ trợ
            Result := T1;
        end;
end;

//Các thao tác chính trên danh sách
procedure Insert(i: Integer; v: Integer); //Chèn v vào vị trí i
var
    T1, T2: PNode;
begin
    if i > root^.len then i := root^.len + 1;
    Split(root, i - 1, T1, T2); //Tách thành hai danh sách con tại vị trí chèn
    New(root); //Tạo gốc mới chứa phần tử chèn vào
    root^.value := v;
    root^.P := nilT;
    SetL(root, T1); SetR(root, T2); //Cho hai danh sách con vào hai bên
    Update(root); //Cập nhật thông tin phụ trợ
end;

procedure Delete(i: Integer); //Xóa phần tử thứ i
var
    x, T1, T2: PNode;
begin
    x := NodeAt(root, i); //Nhảy tới nút chứa phần tử cần xóa
    Splay(x); //Đẩy lên gốc
    T1 := x^.L; T1^.P := nilT;
    T2 := x^.R; T2^.P := nilT;
    Dispose(x); //Giải phóng bộ nhớ
    root := Join(T1, T2); //Ghép nối tiếp hai danh sách hai bên vị trí xóa
end;

procedure Update(i: Integer; v: Integer); //Cập nhật giá trị phần tử thứ i
begin
    root := NodeAt(root, i); //Nhảy tới nút chứa phần tử thứ i
    Splay(root); //Đẩy lên gốc
    root^.value := v; //Đặt giá trị mới
    Update(root); //Cập nhật lại thông tin phụ trợ (sum)
end;

function Query(i, j: Integer): Int64; //Tính tổng các phần tử từ i tới j
var
    T1, T2, T3: PNode;
begin
    Split(root, j, T2, T3); Split(T2, i - 1, T1, T2); //Tách thành 3 danh sách nối tiếp T1, T2, T3

```

```

Result := T2^.sum; //Trả về tổng các phần tử trong T2
Root := Join(Join(T1, T2), T3); //Ghép lại
end;

procedure Solve;
var
  c: Char;
  k, p: Integer;
  i, j: Integer;
  v: Integer;
begin
  ReadLn(k);
  for p := 1 to k do
    begin
      Read(c);
      case c of
        'I':
        begin
          ReadLn(i, v);
          Insert(i, v);
        end;
        'D':
        begin
          ReadLn(i);
          Delete(i);
        end;
        'U':
        begin
          ReadLn(i, v);
          Update(i, v);
        end;
        'Q':
        begin
          ReadLn(i, j);
          WriteLn(Query(i, j));
        end;
      end;
    end;
  end;

procedure FreeMemory(x: PNode);
begin
  if x = nilT then Exit;
  FreeMemory(x^.L);
  FreeMemory(x^.R);
  Dispose(x);
end;

begin
  Init;
  Solve;
  FreeMemory(root);
end.

```

Thời gian thực hiện giải thuật có thể đánh giá qua số lần thực hiện thao tác *Splay* và số lần gọi hàm *NodeAt*. Nhận xét rằng mỗi khi phép truy cập phần tử

NodeAt đi từ gốc xuống một nút thì ngay sau đó là một lệnh *Splay* chuyển nút đó lên gốc. Chính vì vậy thời gian thực hiện giải thuật có thể đánh giá qua dãy các thao tác *Splay*, số thao tác *Splay* cần thực hiện là một đại lượng $O(k)$.

Các nghiên cứu lý thuyết đã chỉ ra rằng một phép *Splay* thực hiện riêng rẽ có thể mất thời gian $\Theta(n)$ với n là số nút trên cây. Tuy vậy, thời gian thực hiện cả dãy gồm k thao tác *Splay* tính tổng lại chỉ là $O((k + n) \log n + k)$. Tức là ta có thể coi mỗi phép cập nhật/truy vấn được thực hiện trong thời gian $O(\log n)$ (đánh giá bù trừ - *amortized analysis*) khi $k > n$.

Người ta cũng đã chứng minh được rằng cây *Splay* có thời gian thực hiện giải thuật trên một dãy thao tác chèn/xóa/tìm kiếm tương đương với cây nhị phân tìm kiếm tối ưu khi đánh giá bằng ký pháp O lớn (dĩ nhiên là sai khác một hằng số ẩn trong ký pháp O).

2.2. Cây quản lý đoạn

Segment trees [2] là một cấu trúc dữ liệu ban đầu được thiết kế cho các ứng dụng hình học. Cấu trúc này khá phức tạp và được sử dụng để trả lời nhiều loại truy vấn khó. *Segment trees* thường được so sánh với *interval trees* là một dạng cấu trúc dữ liệu khác cung cấp các chức năng tương đương.

Trong mục này, ta đơn giản hóa cấu trúc *segment trees* để giải quyết bài toán truy vấn phạm vi. Điều này làm cho cây quản lý đoạn chỉ giống với *segment trees* ở hình ảnh biểu diễn còn các thuộc tính và phương thức trở nên đơn giản và “yếu” hơn nhiều so với cấu trúc nguyên thủy, theo nghĩa không trả lời được những truy vấn khó như cấu trúc nguyên thủy.

Trên các diễn đàn thảo luận về thuật toán trong nước và thế giới, đôi khi tên gọi *interval trees* hoặc *segment trees* vẫn được dùng để gọi tên cấu trúc này, tuy nhiên tôi không thấy định nghĩa về nó trong các cuốn sách thuật toán phổ biến [3] [1] [4]. Các bài giảng thuật toán cũng chỉ dùng tên gọi *interval trees* và *segment trees* để đề cập tới hai cấu trúc dữ liệu trong hình học tính toán. Cấu trúc mà mục này nói đến, tôi tạm gọi là cây quản lý đoạn, chỉ là một hạn chế của *interval trees* hay *segment trees* trong trường hợp cụ thể.

Mọi bài toán giải quyết được bằng cây quản lý đoạn đều có thể giải được bằng cấu trúc dữ liệu đã trình bày ở mục 2.1, tuy nhiên điều ngược lại không đúng.

Mặc dù vậy, cây quản lý đoạn cung cấp một cách quản lý mới thông qua các đoạn sơ cấp, ngoài ra việc cài đặt dễ dàng cũng là một ưu điểm của cấu trúc dữ liệu này. Ta giới hạn lại bài toán Range Query trong trường hợp đặc biệt: không có phép chèn và xóa phần tử để khảo sát cấu trúc dữ liệu cây quản lý đoạn.

Bài toán: (Range Query 2).

Cho một dãy gồm n số nguyên $A = (a_1, a_2, \dots, a_n)$. Xét hai phép biến đổi:

- Phép cập nhật $Update(i, v)$: Đặt $a_i := v$
- Phép truy vấn $Query(i, j)$: Trả về tổng các phần tử từ a_i tới a_j

Yêu cầu: Cho dãy k thao tác thực hiện tuần tự, hãy trả lời tất cả các truy vấn $Query$

Input

- Dòng 1 chứa 2 số nguyên dương $n, k \leq 10^5$
- Dòng 2 chứa n số nguyên a_1, a_2, \dots, a_n
- n dòng tiếp, mỗi dòng cho thông tin về một phép biến đổi. Mỗi dòng bắt đầu bởi một ký tự $\in \{U,Q\}$
 - Nếu ký tự đầu dòng là “U” thì tiếp theo là hai số nguyên i, v tương ứng với phép cập nhật $Update(i, v)$ ($v \leq 10^5$)
 - Nếu ký tự đầu dòng là “Q” thì tiếp theo là hai số nguyên i, j tương ứng với phép truy vấn $Query(i, j)$ ($i < j$)

Output

Trả lời tất cả các truy vấn $Query$, với mỗi truy vấn in ra câu trả lời trên 1 dòng

Sample Input	Sample Output
<pre>9 5 1 2 3 4 5 6 7 8 9 U 1 5 //5 2 3 4 5 6 7 8 9 U 3 6 //5 2 6 4 5 6 7 8 9 Q 1 4 U 8 1 //5 2 6 4 5 6 7 1 9 Q 1 9</pre>	<pre>17 45</pre>

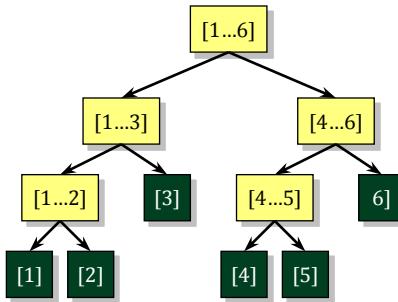
2.2.1. Cấu trúc

Cây quản lý đoạn là một cây nhị phân đầy đủ (*full binary trees*) có cấu trúc như sau:

- Mỗi nút quản lý một dãy các đối tượng liên tiếp, trong nút chứa thông tin tổng hợp từ các đối tượng mà nó quản lý (trong trường hợp cụ thể này, mỗi nút chứa tổng của các phần tử trong phạm vi mà nó quản lý).
- Nút gốc quản lý các đối tượng từ 1 tới n .
- Nếu một nút quản lý dãy các đối tượng từ l tới h ($l < h$) thì nút con trái của nó quản lý các đối tượng từ l tới m và nút con phải của nó quản lý các đối tượng từ $m + 1$ tới h . Ở đây $m = \lfloor(l + h)/2\rfloor$.
- Nếu một nút chỉ quản lý một đối tượng thì nó sẽ là nút lá và không có nút con. Trong một số trường hợp cần tăng tốc thuật toán, mỗi đối tượng i sẽ được gắn với một con trỏ $leaf[i]$ trỏ tới nút lá quản lý trực tiếp đối tượng i .

Để thuận tiện cho trình bày, với mỗi nút x ta lưu thêm hai chỉ số l_x và h_x cho biết x quản lý các phần tử từ trong mảng A từ chỉ số l_x tới chỉ số h_x .

Hình 6 là ví dụ về cây quản lý đoạn gồm 6 đối tượng



Hình 6. Cây quản lý đoạn

Bố đề 1

Với mọi số nguyên dương n , độ sâu của các nút lá trên cây quản lý đoạn gồm n đối tượng không vượt quá $\lg(2n - 1)$

Chứng minh

Gọi $d_{max}(n)$ là độ sâu lớn nhất của một nút lá trên cây quản lý đoạn gồm n đối tượng. Theo cơ chế xây dựng cây: nút con trái của nút gốc quản lý $[n/2]$

đối tượng và nút con phải của nút gốc quản lý $[n/2]$ đối tượng. Tức là nếu xét về độ sâu:

$$dmax(n) = \begin{cases} 0, & \text{nếu } n = 1 \\ dmax([n/2]) + 1, & \text{nếu } n > 1 \end{cases} \quad (1)$$

Rõ ràng khi $n = 1$, ta có $dmax(1) = 0$, bổ đề đúng.

Khi $n \geq 2$, ta chứng minh bất đẳng thức mạnh hơn: $dmax(n) \leq \lg(2n - 2)$.

Khi $n = 2$ ta có $dmax(2) = 1 = \lg(2n - 2)$, bất đẳng thức đúng.

Trường hợp $n > 2$, bất đẳng thức sẽ được chứng minh bằng quy nạp theo n :

$$\begin{aligned} dmax(n) &= dmax([n/2]) + 1 \\ &\leq \lg(2[n/2] - 2) + 1 \\ &\leq \lg(n + 1 - 2) + 1 \\ &= \lg(n - 1) + 1 \\ &= \lg(2n - 2) \end{aligned} \quad (2)$$

Bổ đề 2

Với mọi số nguyên dương n , độ sâu của các nút lá trên cây quản lý đoạn gồm n đối tượng không nhỏ hơn $\lg(n + 1) - 1$

Chứng minh

Gọi $dmin(n)$ là độ sâu nhỏ nhất của một nút lá trên cây quản lý đoạn gồm n đối tượng. Cũng theo cơ chế xây dựng cây, ta có:

$$dmin(n) = \begin{cases} 0, & \text{nếu } n = 1 \\ dmin([n/2]) + 1, & \text{nếu } n > 1 \end{cases} \quad (3)$$

Rõ ràng khi $n = 1$, ta có $dmin(1) = 0$, bổ đề đúng.

Khi $n > 1$, bổ đề được chứng minh bằng quy nạp theo n :

$$\begin{aligned} dmin(n) &= dmin([n/2]) + 1 \\ &\geq \lg([n/2] + 1) - 1 + 1 \\ &= \lg([n/2] + 1) \\ &\geq \lg((n - 1)/2 + 1) \\ &= \lg((n + 1)/2) \\ &= \lg(n + 1) - 1 \end{aligned} \quad (4)$$

Bổ đề 1 và Bổ đề 2 chỉ ra rằng độ sâu của các nút lá cũng như độ cao của cây là một đại lượng $\Theta(\lg n)$. Đó là cơ sở cho phép biểu diễn cây cũng như các phép đánh giá thời gian thực hiện giải thuật.

2.2.2. Biểu diễn

Ta sẽ sử dụng mảng để biểu diễn cây quản lý đoạn theo cơ chế quen thuộc: Nút gốc được đánh số 1, nút x có nút con trái là nút $2x$, nút con phải là nút $2x + 1$ và nút cha nút x là $\lfloor x/2 \rfloor$. Muốn vậy ta cần ước lượng kích thước mảng cần khai báo.

Xét một cây quản lý đoạn độ cao h , ta thêm cho nút cực trái của cây một nút lá bên trái, gọi là nút u . Do phép xây dựng cây, nút lá u này chắc chắn có độ sâu lớn hơn tất cả các nút khác trong cây và do đó nó sẽ được đánh số lớn nhất. Theo cơ chế đánh chỉ số, nút lá u sẽ có chỉ số 2^{h+1} (do nó là lá cực trái và nằm ở độ sâu $h + 1$).

Bổ đề 1 đã chỉ ra rằng cây u quản lý danh sách n phần tử có chiều cao $h \leq \lg(2n - 1)$. Do đó chỉ số của nút lá u không bao giờ vượt quá $2^{\lg(2n-1)+1} = 4n - 2$. Điều đó cho thấy chỉ số các nút còn lại không vượt quá $4n - 3$.

Để dễ nhớ người ta thường khai báo mảng với các phần tử được đánh số từ 1 tới $4n$.

Thủ tục $Build(x, low, high)$ dưới đây khởi tạo nhánh cây gốc x quản lý các đối tượng a_{low} tới a_{high} :

```

procedure Build(x: Integer; low, high: Integer);
var
  middle: Integer;
begin
  l[x] := low; h[x] := high;
  if low = high then //Nút x là một lá quản lý duy nhất một đối tượng a[low]
    begin
      sum[x] := a[low];
      leaf[low] := x;
    end
  else //Nút x là nút nhánh và sẽ có đúng 2 nút con
    begin
      middle := (low + high) div 2;
      Build(x * 2, low, middle); //Khởi tạo nhánh con trái quản lý các đối tượng low...middle
      Build(x * 2 + 1, middle + 1, high); //Khởi tạo nhánh con phải quản lý các đối tượng middle + 1...high
      sum[x] := sum[2 * x] + sum[2 * x + 1]; //Tổng hợp thông tin tổng từ 2 nút con lên nút x
    end;
end;

```

Thủ tục khởi tạo cây quản lý đoạn sẽ được thực hiện bằng lời gọi *Build(1,1,n)*. Việc khởi tạo cây được thực hiện trong thời gian $\Theta(n)$.

2.2.3. Cập nhật

Thao tác đầu tiên trên cây quản lý đoạn là phải cập nhật lại cây mỗi khi có sự thay đổi giá trị phần tử. Khi một phần tử a_i bị thay đổi, ta nhảy tới nút $x = \text{leaf}[i]$ là nút lá trực tiếp quản lý a_i và đi từ nút x lên gốc, cứ đi qua nút nào thì cập nhật lại thông tin tổng $\text{sum}[\cdot]$. Sau quá trình này, tất cả các nút chứa phần tử a_i trong phạm vi quản lý sẽ được cập nhật lại thông tin tổng $\text{sum}[\cdot]$.

Thủ tục *Update(i,v)* dưới đây hiệu chỉnh lại cây quản lý đoạn khi có sự thay đổi $a_i := v$

```

procedure Update(i: Integer; v: Integer);
var
  x: Integer;
begin
  x := leaf[i]; //Nhảy tới nút lá quản lý trực tiếp đối tượng i
  sum[x] := v; //Cập nhật thông tin tại lá x
  while x > 1 do //Chừng nào x chưa phải là gốc
    begin
      x := x div 2; //Nhảy lên nút cha của x rồi cập nhật thông tin tổng
      sum[x] := sum[2 * x] + sum[2 * x + 1];
    end;
end;

```

Có thể hình dung cơ chế cập nhật thông qua mô hình quản lý nhân sự: Giả sử một cơ quan cần quản lý n nhân viên và có cơ cấu tổ chức theo sơ đồ phân cấp dạng cây: Các nút lá tương ứng với các nhân viên và mỗi nút nhánh là một vị lãnh đạo, mỗi lãnh đạo sẽ quản lý trực tiếp đúng 2 nút con của mình (có thể là nhân viên hoặc vị lãnh đạo cấp thấp hơn). Khi mỗi nhân viên thay đổi thông tin, anh ta cần báo cáo sự thay đổi đó cho vị lãnh đạo quản lý trực tiếp mình để vị lãnh đạo này cập nhật và báo cáo lên cấp cao hơn... cho tới khi thông tin trên nút gốc – lãnh đạo cấp cao nhất được cập nhật.

Thời gian thực hiện của thủ tục *Update* là $\Theta(\lg n)$ vì vòng lặp while bên trong có số lần lặp bằng độ sâu của nút $leaf[i]$.

2.2.4. Truy vấn

Phép truy vấn nhận vào hai chỉ số $i \leq j$ và trả về tổng các phần tử từ a_i tới a_j

Trước tiên, ta viết một hàm *Request(x)* nhận vào một nút x và trả về tổng các phần tử do x quản lý mà nằm trong phạm vi truy vấn $[i, j]$ (các phần tử có chỉ số vừa nằm trong phạm vi $l_x \dots h_x$, vừa nằm trong phạm vi $i \dots j$):

- Trường hợp 1: $[l_x, h_x] \cap [i, j] = \emptyset$, nút x không quản lý đối tượng nào trong phạm vi truy vấn, hàm *Request* trả về 0.
- Trường hợp 2: $[l_x, h_x] \subseteq [i, j]$, tất cả các đối tượng do x quản lý đều nằm trong phạm vi truy vấn, hàm *Request* trả về $sum[x]$
- Ngoài hai trường hợp trên, hàm *Request* gọi đệ quy để “hỏi” hai nút con về tổng các phần tử truy vấn thuộc phạm vi quản lý của nhánh con trái và tổng các phần tử truy vấn thuộc phạm vi quản lý của nhánh con phải, sau đó cộng hai kết quả lại thành kết quả hàm.

Phép truy vấn về tổng các phần tử từ a_i tới a_j được thực hiện bởi một lời gọi hàm *Request(1)*.

```

function Query(i, j: Integer): Int64;

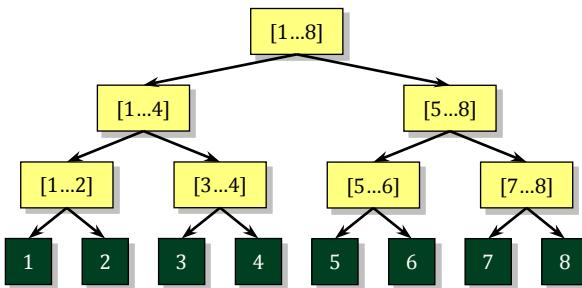
function Request(x: Integer): Int64;
begin
  if (j < l[x]) or (h[x] < i) then //l[x]...h[x] không có giao với i...j
    Exit(0);
  if (i ≤ l[x]) and (j ≥ h[x]) then //l[x]...h[x] nằm hoàn toàn trong i...j
    Exit(sum[x]);
  //Truy vấn trên hai nút con rồi tổng hợp lại
  Result := Request(2 * x) + Request(2 * x + 1);
end;

begin
  Result := Request(1);
end;

```

Thời gian thực hiện giải thuật của phép truy vấn có thể đánh giá qua số lần gọi hàm *Request*. Nhìn vào mô hình cài đặt, mỗi phép “+” sẽ kéo theo 2 lần gọi hàm *Request*, như vậy số lần gọi hàm *Request* bằng $2 \times$ số phép “+” cộng thêm 1. Tức là thời gian thực hiện giải thuật có thể đánh giá qua số lần thực hiện phép “+”.

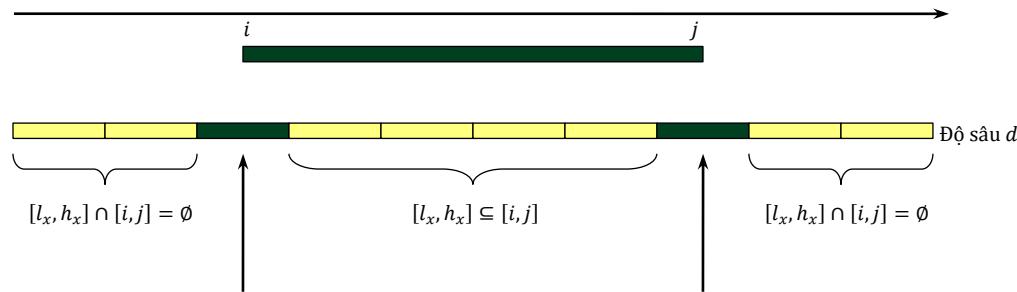
Trong hàm *Request*(x), phép “+” chỉ được thực hiện khi mà đoạn $[l_x, h_x]$ có giao khác rỗng với đoạn $[i, j]$ nhưng đoạn $[l_x, h_x]$ không nằm trong đoạn $[i, j]$. Với một độ sâu d cố định, ta khảo sát xem có bao nhiêu phép “+” được thực hiện trong các hàm *Request*(x) với x là một nút có độ sâu d . Có thể thấy rằng các phạm vi quản lý của các nút ở độ sâu d đôi một không giao nhau. Như ví dụ ở Hình 7 là cây quản lý đoạn gồm 8 phần tử, ở độ sâu 2, có 4 nút quản lý các phạm vi rời nhau $[1,2], [3,4], [5,6]$ và $[7,8]$.



Hình 7. Cây quản lý đoạn gồm 8 phần tử

Nhận xét trên cho ta thấy rằng ở một độ sâu d , có không quá 2 nút mà phạm vi quản lý mỗi nút có giao khác rỗng với đoạn $[i, j]$ nhưng không chứa trong

đoạn $[i, j]$: Một nút chứa i và một nút chứa j trong phạm vi quản lý của chúng (Hình 8)



Hình 8. Khảo sát phạm vi quản lý của các nút tại độ sâu d .

Vậy thì, ở độ sâu d , có không quá hai phép “+” được thực hiện, điều này chỉ ra rằng tổng số phép “+” không vượt quá $2h$ với h chiều cao của cây. Mặt khác, độ cao của cây là một đại lượng $O(\log n)$. Suy ra thời gian thực hiện giải thuật của phép *Query* là $O(\log n)$.

2.2.5. Cài đặt

LISTQUERY2.PAS✓ Cây quản lý đoạn

```
{$MODE OBJFPC}
program ListQuery2;
const
  maxN = Round(1E5);
var
  a: array[1..maxN] of Integer;
  n, k: Integer;
  sum: array[1..4 * maxN] of Int64;
  l, h: array[1..4 * maxN] of Integer;
  leaf: array[1..maxN] of Integer;

procedure Enter;
var
  i: Integer;
begin
  ReadLn(n, k);
  for i := 1 to n do Read(a[i]);
  ReadLn;
end;

procedure Build(x: Integer; low, high: Integer); //Dựng cây quản lý đoạn
var
  middle: Integer;
begin
  l[x] := low; h[x] := high;
  if low = high then
    begin
      sum[x] := a[low];
    end
    else
      begin
        middle := (low + high) shr 1;
        Build(x * 2, low, middle);
        Build(x * 2 + 1, middle + 1, high);
        sum[x] := sum[x * 2] + sum[x * 2 + 1];
      end;
end;

procedure Query(x, low, high: Integer; var res: Int64);
var
  middle: Integer;
begin
  if low <= l[x] and high >= h[x] then
    res := sum[x]
  else
    begin
      if low <= h[x] then
        Query(x * 2, low, high, res);
      if high >= l[x] then
        Query(x * 2 + 1, low, high, res);
      res := res + sum[x];
    end;
end;

begin
  Enter;
  Build(1, 1, maxN);
  for i := 1 to k do
    begin
      ReadLn;
      if ReadLn = '+' then
        begin
          ReadLn;
          ReadLn;
          Query(1, a[i], a[i + 1], res);
          WriteLn(res);
        end
        else
          begin
            ReadLn;
            ReadLn;
            l[a[i]] := a[i];
            h[a[i]] := a[i + 1];
            sum[a[i]] := a[i];
          end;
    end;
end.
```

```

        leaf[low] := x;
    end
else
begin
    middle := (low + high) div 2;
    Build(2 * x, low, middle);
    Build(2 * x + 1, middle + 1, high);
    sum[x] := sum[2 * x] + sum[2 * x + 1];
end;
end;

procedure Update(i: Integer; v: Integer); //Đặt phần tử thứ i bằng v
var
    x: Integer;
begin
    x := leaf[i]; //Nhảy tới lá quản lý a[i]
    sum[x] := v; //Cập nhật sum[,] từ lá lên gốc
    while x > 1 do
begin
    begin
        x := x div 2;
        sum[x] := sum[2 * x] + sum[2 * x + 1];
    end;
end;
end;

function Query(i, j: Integer): Int64; //Tính tổng các phần tử a[i..j]

function Request(x: Integer): Int64; //Trả về tổng các phần tử truy vấn nằm trong phạm vi l[x]...h[x]
begin
    if (l[x] > j) or (h[x] < i) then Exit(0);
    if (i <= l[x]) and (h[x] <= j) then Exit(sum[x]);
    Result := Request(2 * x) + Request(2 * x + 1);
end;

begin
    Result := Request(1);
end;

procedure Solve;
var
    c: Char;
    p: Integer;
    i, j: Integer;
    v: Integer;
begin
    for p := 1 to k do
begin
    Read(c);
    case c of
        'U':
        begin
            ReadLn(i, v);
            Update(i, v);
        end;
        'Q':
        begin
            ReadLn(i, j);
            WriteLn(Query(i, j));
        end;
    end;
end;

```

```

        end;
    end;
end;
end;

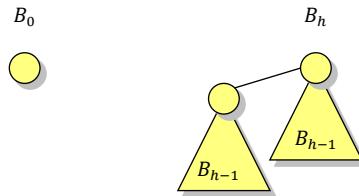
begin
Enter;
Build(1, 1, n);
Solve;
end.

```

2.3. Cây chỉ số nhị phân

Cây nhị thức (*binomial trees*) B_h là cây có thứ tự (có phân biệt thứ tự các nhánh con) được định nghĩa quy nạp như sau:

- Cây nhị thức bậc 0, ký hiệu B_0 , là cây chỉ gồm một nút.
- Với $h > 0$, cây nhị thức bậc h , ký hiệu B_h , được tạo thành từ hai cây nhị thức B_{h-1} bằng cách cho gốc một cây nhị thức B_{h-1} làm con trái nhất của gốc cây kia (Hình 9).



Hình 9. Minh họa định nghĩa của cây nhị thức

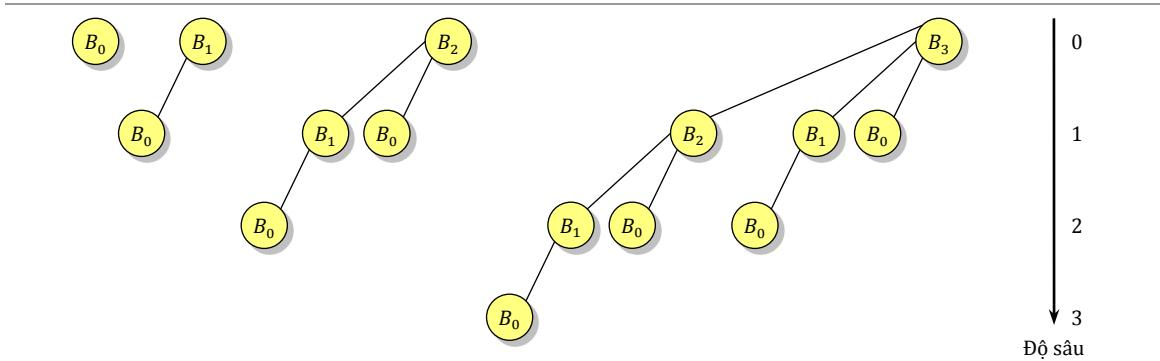
Định lý 3

Cây nhị thức B_h có:

- Đúng 2^h nút
- Chiều cao là h
- Đúng $\binom{h}{d}$ nút ở độ sâu d ($d = 0, 1, \dots, h$)
- Nút gốc có h nút con, là nút có số nút con (bậc) lớn nhất. Nếu ta ta đánh số các nút con của nút gốc theo thứ tự từ trái qua phải bởi các số nguyên $h - 1, h - 2, \dots, 0$ thì nút con thứ i là gốc của một cây nhị thức B_i .

Định lý 3 có thể dễ dàng chứng minh bằng quy nạp. Tên gọi cây nhị thức xuất phát từ tính chất thứ ba. $\binom{h}{d}$ là hệ số nhị thức (số tổ hợp chập d của tập gồm h phần tử).

Cây nhị thức là một cây có tính thứ tự (các nút con của một nút được liệt kê theo thứ tự từ trái qua phải theo thứ tự giảm dần của bậc).



Hình 10. Một số ví dụ về cây nhị thức

Cây nhị thức thường được sử dụng để cài đặt *binomial heaps* [9] và *fibonacci heaps* [6]. Dưới đây ta sẽ trình bày một phương pháp đánh số các nút trên cây nhị thức để giải quyết bài toán truy vấn phạm vi [5]. Do phương pháp này sử dụng biểu diễn nhị phân để quản lý chỉ số, tác giả đã đặt tên cho nó là *cây chỉ số nhị phân* (*binary indexed trees - BIT*) một số tài liệu còn gọi cấu trúc này là *Fenwick trees* theo tên của tác giả.

Mọi bài toán giải quyết được bằng BIT đều có thể giải được bằng cây quản lý đoạn (mục 2.2) và tất nhiên có thể giải được bằng BST (mục 2.1), tuy nhiên điều ngược lại không đúng. Mặc dù bị hạn chế về tầm ứng dụng hơn rất nhiều so với hai cấu trúc dữ liệu trước, BIT cung cấp một cách cài đặt cực kỳ hiệu quả kể cả về kích thước mã lệnh lẫn tốc độ thực thi.

Ta cũng lấy bài toán truy vấn tổng trong (mục 2.2) làm ví dụ

2.3.1. Cơ chế biểu diễn số nguyên có dấu

Khi biểu diễn một số nguyên bằng dãy h bit: $b_{h-1}b_{h-2} \dots b_1b_0$ thì các bit được đánh số từ 0 tới $h - 1$ theo thứ tự từ phải qua trái, gọi là thứ tự từ bit thấp nhất (b_0) tới bit cao nhất (b_{h-1}). Mỗi giá trị thuộc một kiểu số nguyên được biểu diễn trong máy tính bằng một dãy bit chiều dài cố định, nhưng tùy theo kiểu số nguyên, việc giải mã dãy bit ra giá trị số có sự khác nhau.

Cơ chế biểu diễn số nguyên không dấu (Byte, Word, LongWord, QWord) khá dễ hiểu: Giá trị số đúng bằng giá trị nhị phân của dãy bit. Tức là giá trị của số nguyên không dấu biểu diễn bởi dãy bit $b_{h-1}b_{h-2} \dots b_1b_0$ bằng $\sum_{i=0}^{h-1} b_i 2^i$.

Cơ chế biểu diễn số nguyên có dấu (ShortInt, SmallInt, LongInt, Int64) thì phức tạp hơn:

- Nếu bit cao nhất là 0 thì dãy bit này biểu diễn một số không âm đúng bằng giá trị không dấu
- Nếu bit cao nhất là 1 thì dãy bit này biểu diễn một số âm bằng giá trị không dấu trừ đi 2^h với h là kích thước tính bằng bit của kiểu số nguyên tương ứng.

Ví dụ xét hai kiểu số nguyên 8 bit: Byte và ShortInt, ta có giá trị của một số dãy nhị phân trong hai kiểu số nguyên này là:

Dãy bit	Giá trị Byte: $0 \dots 2^8 - 1$	Giá trị ShortInt: $-2^7 \dots 2^7 - 1$
00000000	0	0
00000001	1	1
00011001	25	25
01111111	127	127
10000000	128	-128
10000001	129	-127
11001101	205	-51
11111111	255	-1

Việc thực hiện các phép tính số học được thực hiện trực tiếp trên dãy bit. Ví dụ:

$$\begin{array}{r}
 11001101 \\
 + 00011001 \\
 \hline
 11100110
 \end{array}$$

Biểu thức này có thể diễn giải ra thành $205 + 25 = 230$ hoặc $-51 + 25 = -26$ tùy theo kiểu số nguyên trên từng toán hạng Byte (không dấu) hay ShortInt (có dấu)*.

Một số công thức biến đổi bit:

□ Phép cộng 1

Để tăng một số nguyên lên 1, máy sẽ xét dãy bit biểu diễn số nguyên đó, tìm bit 0 thấp nhất thay bởi bit 1 và đặt tất cả các bit đứng sau thành 0.

Ví dụ khi cộng 1 vào số $43 = \%101011$, máy xác định bit 0 cuối cùng (bit số 2) thay bởi 1 và đặt các bit đứng sau (bit số 0, 1) thành 0, được dãy $\%101100$ là biểu diễn nhị phân của số 44

□ Phép lấy số đối

Trong các kiểu số nguyên có dấu, giá trị -1 được biểu diễn bởi dãy gồm toàn bit 1. Chính vì thế phép toán $x + \text{not } x$ sẽ luôn cho giá trị -1 . Từ $x + \text{not } x = -1$, ta suy ra:

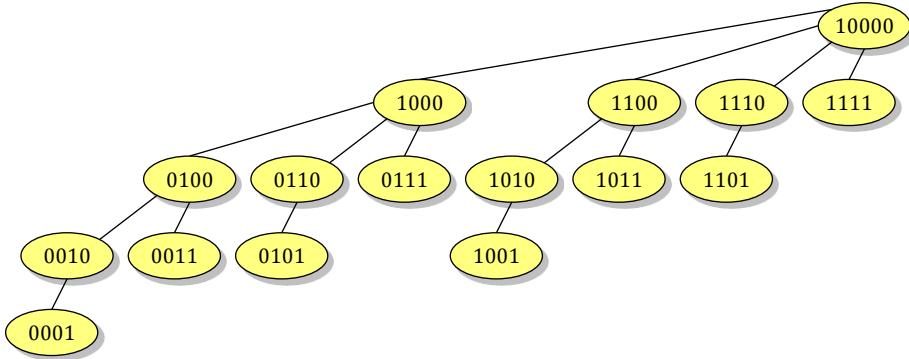
$$-x = \text{not } x + 1$$

Tức là để tính giá trị $-x$, máy sẽ đảo tất cả các bit trong x , sau đó cộng thêm 1 vào dãy bit này.

2.3.2. Đánh số các nút

Xét một cây nhị thức có chiều cao h và số nút $n = 2^h$. Ta duyệt cây theo thứ tự sau (post-order traversal) và đánh số các nút trên cây từ 1 tới n bởi các số nguyên kiểu có dấu và đồng nhất mỗi số thứ tự với dãy bit biểu diễn số thứ tự đó. Hình 11 là số thứ tự của các nút trên cây B_4 theo cách đánh số này.

* Để hiểu bản chất phép cộng cần phân tích cơ chế xử lý trên các thanh ghi tích lũy, cờ nhớ (carry) và cờ tràn (overflow) (dây là những thành phần của ALU), ngoài ra còn phải nói về cơ chế ép kiểu toán hạng khi xử lý biểu thức số học trong ngôn ngữ lập trình bậc cao. Ta bỏ qua các chi tiết này cho dễ hiểu.



Hình 11. Thứ tự sau của các nút

□ Tính số nút trong một nhánh

Nhận xét: Giả sử nút x có bit 1 cuối cùng nằm ở vị trí i thì nút này là gốc của cây B_i và nó sẽ quản lý 2^i nút nằm trong nhánh đó.

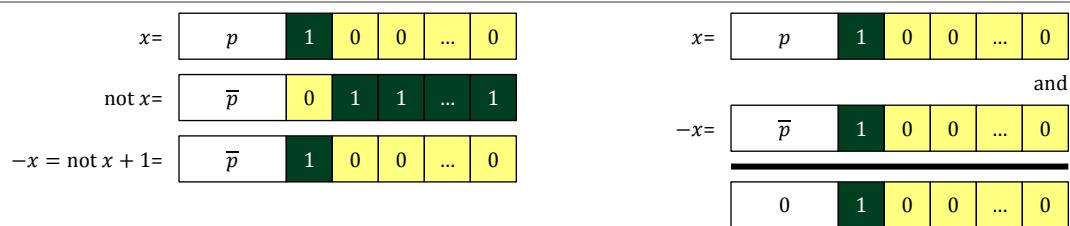
Như ví dụ trên cây B_4

- Các nút 1, 3, 5, 7, 9, 11, 13, 15 ứng với các dãy bit 0001, 0011, 0101, 0111, 1001, 1011, 1101, 1111 là gốc của nhánh cây B_0
- Các nút 2, 6, 10, 14 ứng với các dãy bit 0010, 0110, 1010, 1110 là gốc của nhánh cây B_1
- Các nút 4, 12 ứng với các dãy bit 0100, 1100 là gốc của nhánh cây B_2
- Nút 8 ứng với dãy bit 1000 là gốc của cây B_3
- Nút 16 ứng với dãy bit 10000 là gốc của cây B_4

Tức là nếu ta xác định được bit 1 cuối cùng của x thì có thể tính ra được số nút trong nhánh gốc x . Công thức xác định số nút trong nhánh gốc x là

$$x \text{ and } -x$$

Công thức này được giải thích qua Hình 12



Hình 12

□ Xác định nút cha của một nút

Nếu nút x là gốc của một cây B_k thì các nút em của nó sẽ là gốc của các cây $B_{k-1}, B_{k-2}, \dots, B_0$. Theo nguyên lý đánh số theo thứ tự sau, nút cha của x sẽ có số thứ tự là:

$$x + 2^{k-1} + 2^{k-2} + \cdots + 2^0 + 1 = x + 2^k$$

Chính vì vậy, chỉ số nút cha của x sẽ được xác định bằng cách lấy x cộng với số nút trong nhánh gốc x :

$$\text{parent}(x) = x + (x \text{ and } -x)$$

□ Xác định nút đứng liền trước một nhánh

Nếu x là gốc của một cây B_k gồm 2^k nút thì 2^k nút này sẽ được xếp liên tiếp theo thứ tự sau với x là nút đứng cuối cùng. Vì vậy nút đứng liền trước nhánh cây B_k này sẽ mang số thứ tự $x - 2^k$, tức là lấy x trừ đi số nút trong nhánh gốc x

$$\text{prev}(x) = x - (x \text{ and } -x)$$

Việc lấy x trừ đi số nút trong nhánh gốc x đơn thuần là thay bit 1 thấp nhất trong x bởi bit 0, công thức này có thể tính cách khác nhanh hơn

$$\text{prev}(x) = x \text{ and } (x - 1)$$

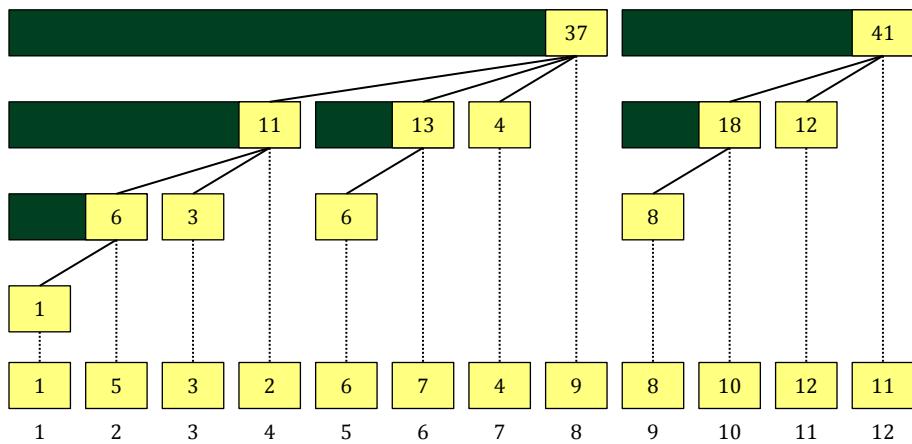
2.3.3. Cấu trúc cây chỉ số nhị phân

Từ kỹ thuật đánh số nút, ta có thể biểu diễn một danh sách các phần tử $A = (a_1, a_2, \dots, a_n)$ bởi một rurgeon các cây nhị thức, gọi là cây chỉ số nhị phân, theo cách sau:

- Mỗi nút quản lý trực tiếp một phần tử trong A , nút i quản lý trực tiếp phần tử a_i
- Nút x là gốc của nhánh cây chứa $(x \text{ and } -x)$ nút và có nút cha là nút $\text{parent}(x) = x + (x \text{ and } -x)$
- Mỗi nút x còn chứa thông tin tổng hợp từ tất cả các nút nằm trong nhánh cây gốc x . Trong trường hợp bài toán truy vấn tổng, mỗi nút x sẽ chứa thông tin về tổng của các phần tử từ $a_{\text{prev}(x)+1}$ tới a_x

Hình 13 là cây chỉ số nhị phân quản lý dãy 12 phần tử

$$1, 5, 3, 2, 6, 7, 4, 9, 8, 10, 12, 11$$



Hình 13. Cây chỉ số nhị phân

Cây chỉ số nhị phân trong bài toán truy vấn tổng được biểu diễn bởi mảng $sum[1 \dots n]$ trong đó $sum[x]$ là tổng các phần tử nằm trong nhánh cây gốc x .

2.3.4. Dụng cây

Thuật toán dụng cây từ dãy $A = (a_1, a_2, \dots, a_n)$ được thực hiện từ dưới lên. Để tính $sum[x]$, ta lấy tổng các giá trị $sum[y]$ với y là nút con của x cộng thêm với $a[x]$:

```

procedure BuildTree;
var
  x, y, low: Integer;
begin
  for x := 1 to n do
    begin
      low := x and pred(x); //cây gốc x quản lý các phần tử a[low + 1...x]
      sum[x] := a[x];
      y := pred(x); //y = nút con út của x
      while y > low do
        begin
          Inc(sum[x], sum[y]); //Cộng sum[y] vào sum[x]
          y := y and pred(y); //Nhảy tới nút con anh liền kề của y
        end;
    end;
end;

```

Thuật toán dụng cây có thời gian thực hiện $O(n)$

2.3.5. Cập nhật

Để tăng giá trị của một phần tử a_x lên Δ , ta nhảy tới nút x quản lý trực tiếp a_x , dọc trên đường đi từ x lên gốc, đi qua nút nào ta tăng $sum[.]$ của nút đó lên Δ :

```
procedure Increase(x: Integer; Delta: Integer);
begin
  while x <= n do
    begin
      sum[x] := sum[x] + Delta; //Cập nhật thông tin phụ trợ trong x
      Inc(x, x and -x); //Nhảy lên nút cha của x
    end;
end;
```

Phép cập nhật có thời gian thực hiện tỉ lệ thuận với độ sâu của nút x tức là mất thời gian $O(\log n)$.

Phép thay đổi giá trị của một phần tử có thể thực hiện bằng thủ tục *Increase* vì thủ tục này chấp nhận giá trị Δ âm hoặc dương.

2.3.6. Truy vấn

Để tính tổng các phần tử từ a_1 tới a_x , giá trị $sum[x]$ dĩ nhiên có mặt trong tổng, ta nhảy từ x sang nút x' đứng liền trước nhánh cây gốc x và cộng thêm vào tổng giá trị $sum[x']$, tiếp theo ta lại nhảy sang nút x'' đứng liền trước nhánh cây gốc x' ... cho tới khi nhảy về 0.

```
function Query1(x: Integer): Int64; //Tính tổng a[1...x]
begin
  Result := 0;
  while x > 0 do
    begin
      Result := Result + sum[x]; //Cộng sum[x] vào kết quả
      x := x and Pred(x); //Nhảy sang nút đứng liền trước nhánh cây gốc x
    end;
end;
```

Hàm *Query1* thực hiện trong thời gian $O(\log n)$, nó có thể sử dụng để tính tổng các phần tử từ a_i tới a_j bằng công thức $Query1(j) - Query1(i - 1)$.

2.3.7. Cài đặt

💻 RANGEQUERY3.PAS✓ Cây chỉ số nhị phân

```
{$MODE OBJFPC}
program ListQuery2;
const
```

```

maxN = Round(1E5);
var
  a: array[1..maxN] of Integer;
  sum: array[1..maxN] of Int64;
  n, k: Integer;

procedure Enter;
var
  i: Integer;
begin
  ReadLn(n, k);
  for i := 1 to n do Read(a[i]);
  ReadLn;
end;

procedure BuildTree; //Dựng cây BIT ứng với dãy A
var
  x, y, low: Integer;
begin
  for x := 1 to n do
    begin
      low := x and pred(x);
      sum[x] := a[x];
      y := pred(x);
      while y > low do
        begin
          Inc(sum[x], sum[y]);
          y := y and pred(y);
        end;
    end;
end;

procedure Increase(x: Integer; Delta: Integer); //Cập nhật khi tăng a[x] lên Delta
begin
  while x <= n do
    begin
      Inc(sum[x], Delta);
      Inc(x, x and -x);
    end;
end;

function Query1(x: Integer): Int64; //Tính tổng a[1..x]
begin
  Result := 0;
  while x > 0 do
    begin
      Result := Result + sum[x];
      x := x and Pred(x);
    end;
end;

procedure Solve;
var
  c: Char;
  p: Integer;
  i, j: Integer;
  v: Integer;

```

```

begin
for p := 1 to k do
begin
  Read(c);
  case c of
    'U':
    begin
      ReadLn(i, v);
      Increase(i, v - a[i]);
      a[i] := v;
    end;
    'Q':
    begin
      ReadLn(i, j);
      WriteLn(Query1(j) - Query1(i - 1));
    end;
  end;
end;

begin
  Enter;
  BuildTree;
  Solve;
end.

```

2.4. Bàn luận

Trong bài toán truy vấn phạm vi một chiều, các cấu trúc dữ liệu mà ta đã khảo sát đều là cấu trúc cây mà mỗi nút chứa thông tin tổng hợp từ một dãy các đối tượng liên tiếp mà nó quản lý. Các dãy các đối tượng liên tiếp do các nút quản lý hình thành nên các “khoảng sơ cấp”. Mỗi phần tử trong danh sách ban đầu có thể thuộc phạm vi quản lý của $O(\log n)$ khoảng sơ cấp.

Nguyên lý cập nhật: Mỗi khi một phần tử bị thay đổi, tất cả các khoảng sơ cấp quản lý nó cần phải cập nhật lại.

Nguyên lý truy vấn: Khi cần truy vấn thông tin tổng hợp từ một dãy các phần tử từ i tới j , ta tách khoảng $[i, j]$ thành $O(\log n)$ khoảng sơ cấp liên tiếp rời nhau và tổng hợp thông tin từ những khoảng sơ cấp thay vì phải truy cập riêng rẽ từng phần tử.

Trong ba cấu trúc dữ liệu đã giới thiệu, cây tìm kiếm nhị phân là tổng quát nhất và cũng khó cài đặt nhất, cây chỉ số nhị phân dễ cài đặt nhất nhưng phạm vi ứng dụng cũng hạn chế nhất. Việc chọn cấu trúc dễ cài đặt nhất cho một bài

toán cụ thể là kỹ năng hết sức cần thiết khi tham gia các kỳ thi, hoặc phải lập trình trong thời gian có hạn.

Thông thường với một bài toán quản lý danh sách, cây tìm kiếm nhị phân là cấu trúc dữ liệu nên nghĩ tới đầu tiên vì nó là cấu trúc tổng quát nhất để biểu diễn và quản lý danh sách. Sau khi đã hình thành thuật toán, lập trình viên có thể đưa vào một số biến đổi để quy dẫn về cây quản lý đoạn hay cây chỉ số nhị phân cho dễ cài đặt hơn.

Chính vì lý do trên mà việc giảng dạy bài toán truy vấn phạm vi 1 chiều nên đề cập tới cây tìm kiếm nhị phân trước, cho dù rất khó cài đặt. Việc giảng dạy cây quản lý đoạn và cây chỉ số nhị phân mà không có kiến thức nền tảng về cây BST biểu diễn danh sách sẽ dẫn tới việc học sinh luôn luôn nghĩ cách biến đổi phương án tiếp cận để có thể cài đặt được trên những cấu trúc dữ liệu đặc thù. Điều này ảnh hưởng tới tư duy thiết kế thuật toán tổng thể.

Trong trường hợp bắt buộc phải dùng cây tìm kiếm nhị phân, theo tôi giáo viên cần cho học sinh có sự chuẩn bị trước về các kỹ năng sau:

- Cần cho học sinh có kiến thức đúng và sử dụng thành thạo con trỏ.
- Bước đầu có thể viết trước một số hàm và thủ tục, học sinh chỉ việc hoàn thiện nốt phần việc còn lại
- Sử dụng con trỏ và bộ nhớ cấp phát động là một kỹ thuật quan trọng, tuy nhiên các chương trình thường trỏ nên cồng kềnh và khó gỡ rối. Vì vậy cần chuẩn hóa kỹ năng cài đặt (tìm cách cài đặt tốt rồi học thuộc), có thể tích hợp các công cụ giúp trực quan hóa hình ảnh cây, danh sách khi gỡ rối.
- Khuyến khích học sinh sử dụng lớp và đối tượng trong lập trình OOP thay vì dùng biến động và con trỏ.

3. Truy vấn phạm vi nhiều chiều

3.1. Trường hợp hai chiều

Trong bài toán truy vấn phạm vi một chiều, các cấu trúc dữ liệu mà ta đã khảo sát đều là cấu trúc cây mà mỗi nút chứa thông tin tổng hợp từ một dãy các đối tượng liên tiếp mà nó quản lý.

Chuyển sang trường hợp hai chiều, vấn đề trở nên phức tạp hơn. Ý tưởng đầu tiên có thể áp dụng là sửa đổi cấu trúc cây biểu diễn/quản lý danh sách từ cây nhị phân sang cây tứ phân (*quad trees*): Chẳng hạn như với cây quản lý đoạn thay vì chia đôi phạm vi quản lý của mỗi nút nhánh cho hai nút con, ta chia 4 phần hình chữ nhật trong mặt phẳng thuộc phạm vi quản lý của mỗi nút nhánh cho 4 nút con... Cách làm này tuy có cải thiện một chút về mặt tốc độ nhưng không nhiều, ngoài ra trong trường hợp xấu nhất, việc trả lời mỗi truy vấn vẫn mất thời gian $\Omega(n)$ với n là số điểm trên mặt phẳng.

Ta xét bài toán truy vấn trên mặt phẳng:

Trên mặt phẳng trực chuẩn Oxy xét hai lệnh:

- *Set(x, y)*: Chấm một điểm tại tọa độ (x, y)
- *Query(x_1, y_1, x_2, y_2)*: Trả về số chấm trong hình chữ nhật $[x_1, x_2] \times [y_1, y_2]$.

Cho k lệnh thực hiện tuần tự, hãy trả lời tất cả những lệnh *Query*.

3.1.1. Cây quản lý phạm vi hai chiều

Cây quản lý phạm vi 2 chiều (2D-range trees) có cấu trúc sơ cấp là một cây tìm kiếm nhị phân chứa các điểm với khóa tìm kiếm là hoành độ của điểm. Nói cách khác nó là một cây biểu diễn danh sách các điểm theo thứ tự tăng dần của hoành độ, tạm thời ta chưa quan tâm tới tung độ.

Để đảm bảo tính tổng quát, ở đây ta dùng cây nhị phân tìm kiếm, trong trường hợp cụ thể có thể biến đổi về cây quản lý đoạn hay cây chỉ số nhị phân cho dễ cài đặt hơn.

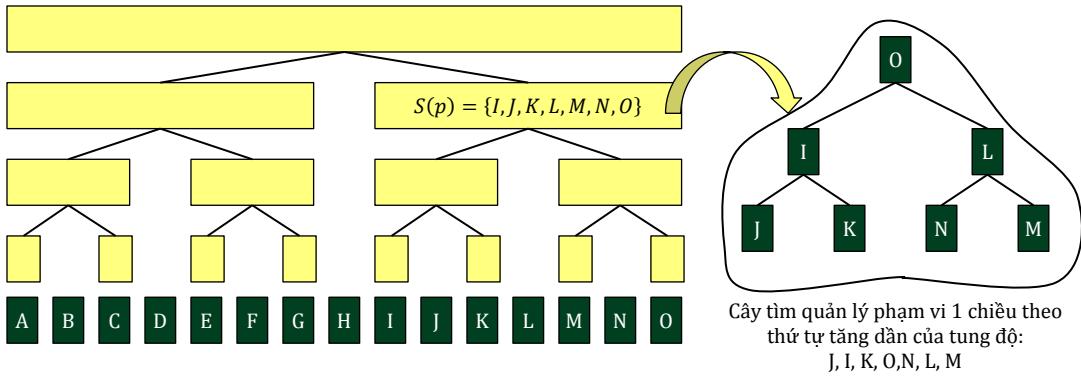
Mỗi nút của cây quản lý tất cả các điểm nằm trong nhánh gốc đó. Ta gọi $S(p)$ là tập các điểm nằm trong nhánh cây gốc p . Cấu trúc thứ cấp của cây quản lý phạm vi hai chiều nằm ở cách tổ chức các $S(p)$: Mỗi $S(p)$ lại là một cây quản lý phạm vi một chiều theo tung độ của các điểm thuộc $S(p)$.

Hình 14 là cây quản lý phạm vi chứa 15 điểm trên mặt phẳng theo thứ tự tăng dần của hoành độ là:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O

Giả sử các các điểm này nếu xếp tăng dần theo tung độ thì thứ tự sẽ là

A, J, B, I, C, K, D, O, E, N, F, L, G, M, H



Hình 14. Cây quản lý phạm vi hai chiều

3.1.2. Bổ sung điểm

Muốn bổ sung một điểm $Z = (x, y)$ ta chèn điểm này vào cấu trúc sơ cấp của cây quản lý phạm vi 2 chiều, việc chèn được giống như trên cây nhị phân tìm kiếm thông thường. Giả sử nút chứa điểm mới chèn vào là nút p , đọc trên đường đi từ p lên gốc, đi qua điểm q nào ta lại chèn điểm Z vào cấu trúc thứ cấp $S(q)$. Như vậy điểm Z sẽ có mặt trong $depth(p) + 1$ cấu trúc thứ cấp với $depth(p)$ là độ sâu của nút p .

Với n là số điểm cần quản lý, những dạng cây nhị phân tìm kiếm tự cân bằng có thể giữ được chiều cao cây ở mức $O(\log n)$, như vậy về mặt bộ nhớ ta mất không gian $O(n \log n)$ để cài đặt cây quản lý phạm vi 2 chiều. Ngoài ra chi phí thời gian thực hiện thao tác bổ sung điểm khi đó sẽ là $O(\log^2 n)$.

3.1.3. Xóa điểm

Quá trình xóa một điểm $Z = (x, y)$ diễn ra ngược lại với quá trình bổ sung điểm. Trước hết ta tìm nút p chứa điểm Z trong cấu trúc sơ cấp (tìm theo x). Đọc trên đường đi từ p lên gốc, đi qua nút q nào ta lại xóa Z trong cấu trúc thứ cấp $S(q)$. Thời gian thực hiện giải thuật cho mỗi phép xóa là $O(\log^2 n)$ nếu sử dụng các dạng cây nhị phân tìm kiếm tự cân bằng.

3.1.4. Truy vấn

Để trả lời về truy vấn thông tin tổng hợp của các điểm trong hình chữ nhật $[x_1, x_2] \times [y_1, y_2]$, (trong trường hợp này là số điểm). Trước hết ta truy vấn khoảng $[x_1, x_2]$ trên cấu trúc sơ cấp để tìm ra một danh sách các nút mà phạm

vì quản lý hoành độ của chúng đúng bằng phạm vi truy vấn (xem lại nguyên lý truy vấn 1 chiều). Trong những nút p vừa tìm được, ta lại dùng khoảng $[y_1, y_2]$ truy vấn trên cấu trúc thứ cấp để đếm số lượng điểm. Thời gian thực hiện giải thuật cho mỗi truy vấn là $O(\log^2 n)$ nếu cấu trúc sơ cấp và thứ cấp là cây BST tự cân bằng, có chiều cao $O(\log n)$.

3.2. Trường hợp nhiều chiều

Tương tự như trường hợp hai chiều, ta có thể mở rộng cho trường hợp d chiều bằng cấu trúc dữ liệu đệ quy:

- Nếu $d = 1$, đơn thuần đây là cây quản lý phạm vi một chiều ta đã trình bày trong mục 2.
- Nếu $d \geq 2$, lấy một trục tọa độ làm trục chính, gọi là trục x , xây dựng cấu trúc sơ cấp là cây tìm kiếm nhị phân của các điểm với khóa tìm kiếm là tọa độ x . Mỗi nút của cấu trúc sơ cấp chứa tập các điểm nằm trong nhánh cây đó, gốc đó. Ta gọi $S(p)$ là tập các điểm nằm trong nhánh cây gốc p . Cấu trúc thứ cấp $S(p)$, là một cây quản lý phạm vi $d - 1$ chiều của các điểm nằm trong $S(p)$.

Trong trường hợp sử dụng các dạng cây nhị phân tìm kiếm tự cân bằng, chi phí bộ nhớ cho cây quản lý phạm vi d chiều là $O(n \log^{d-1} n)$. Mỗi phép cập nhật, bổ sung, loại bỏ điểm mất thời gian $O(\log^d n)$

Chú ý là trong trường hợp một chiều, ta có thể sử dụng cây splay biểu diễn danh sách vì lý do cài đặt đơn giản. Tuy nhiên cây splay sẽ thể hiện nhược điểm với số chiều lớn hơn. Mặc dù thời gian thực hiện một dãy k phép biến đổi chèn/xóa trên cây splay được đảm bảo ở giới hạn thời gian $O((k + n) \log n + k)$, việc truy cập (access) tuần tự các nút sẽ khiến cho cây splay suy biến và có chiều cao $n - 1$. Vấn đề này khiến cho mỗi điểm có thể có mặt trong n cấu trúc thứ cấp và làm tăng lượng bộ nhớ cần huy động lên $\Omega(n^2)$. Chính vì vậy khi cài đặt cây quản lý phạm vi nhiều chiều và không có cách nào sử dụng cấu trúc tĩnh như cây quản lý đoạn hay cây chỉ số nhị phân, người ta thường dùng Treaps, cây đỏ đen, hoặc cây AVL để thay thế cho cây splay.

4. Một số bài toán ví dụ

4.1. Một số bài toán để hiểu rõ cấu trúc dữ liệu

Đối với bài toán truy vấn tổng 1 chiều, ta xét trường hợp dãy các số cần quản lý là dãy số nguyên dương.

4.1.1. Tìm chỉ số

Yêu cầu bổ sung một loại truy vấn: $First(i, v)$: Tìm chỉ số i nhỏ nhất mà tổng các phần tử từ 1 tới i vừa đủ $\geq v$.

Ý tưởng: Thiết kế thuật toán tương tự như thuật toán tìm kiếm nhị phân.

4.1.2. Tìm giá trị

Giả sử rằng với các cấu trúc biểu diễn cây quản lý phạm vi 1 chiều, ta không dùng các thông tin khác ngoài cấu trúc cây, tức là không dùng mảng A , mảng $leaf \dots$. Với một chỉ số i hãy tìm giá trị của phần tử thứ i trong danh sách cần quản lý

4.1.3. Bài toán Range-Minimum Query (RMQ)

Tương tự như bài toán truy vấn tổng, nhưng mỗi truy vấn $Query(i, j)$ cần trả về giá trị nhỏ nhất trong các phần tử $a_i \dots a_j$.

Câu hỏi: Việc sử dụng cây chỉ số nhị phân có khó khăn gì trong trường hợp này?

4.1.4. Cộng/nhân cả dãy

Tương tự như bài toán truy vấn tổng, nhưng bổ sung thêm phép toán:

$Add(\Delta)$: Cộng tất cả các phần tử của dãy lên Δ

$Mul(\Pi)$: Nhân tất cả các phần tử của dãy với Π

4.2. Một số bài toán luyện tập

4.2.1. Uống rượu

(Nguồn bài: thầy Nguyễn Đức Nghĩa, 2002)

Bờm thắng phú ông trong một cuộc đánh cược và buộc phú ông phải đái rượu. Phú ông bèn bày ra một dãy n chai chứa đầy rượu, và nói với Bờm rằng có thể

uống bao nhiêu tuỳ ý, nhưng đã chọn chai nào thì phải uống hết và không được uống ở k chai liền nhau bởi đó là điều xui xẻo.

Bạn hãy chỉ cho Bòm cách uống được nhiều rượu nhất.

Dữ liệu: Vào từ file văn bản BOTTLES.INP

- Dòng 1 chứa hai số nguyên $1 \leq n \leq 10^5; 2 \leq k \leq 10^5$
- Dòng 2 chứa các số nguyên dương ($\leq 10^9$) là dung tích của các chai rượu phú ông bày ra, theo thứ tự liệt kê từ chai thứ nhất tới chai thứ n

Kết quả: Ghi ra file văn bản BOTTLES.OUT

- Dòng 1 ghi số chai được chọn và lượng rượu tối đa có thể uống.
- Dòng 2 ghi chỉ số của các chai được chọn theo thứ tự tăng dần.

Các số trên một dòng của Input/Output files được/phải ghi cách nhau ít nhất một dấu cách

Ví dụ

BOTTLES.INP	BOTTLES.OUT
6 3	4 40
6 10 10 13 10 10	2 3 5 6

Gợi ý lời giải:

Thuật toán quy hoạch động $\Theta(n \cdot k)$ tính $f[i]$ là lượng rượu lớn nhất có thể uống từ chai thứ 1 tới chai thứ k không phải là khó, tuy nhiên khi n, k khá lớn, ta cần có sự hỗ trợ của cấu trúc dữ liệu để làm giảm thời gian thực hiện giải thuật.

Đặt lại vấn đề: Thay vì chọn rượu để uống, ta cần chọn một số chai rượu để bỏ sao cho tổng lượng rượu phải bỏ là nhỏ nhất.

Gọi $f[i]$ là lượng rượu ít nhất phải bỏ trong trường hợp được chọn trong các chai từ 1 tới i , không chọn k chai liền nhau, và chắc chắn có bỏ chai thứ i . Ta quan tâm tới $f[n + 1]$: lượng rượu ít nhất phải bỏ khi chọn trong n chai.

Công thức truy hồi tính $f[i]$

$$f[i] = \begin{cases} a[i], & \text{nếu } i \leq k \\ \min_{i-k \leq j \leq i-1} \{f[j]\} + a[i], & \text{nếu } i > k \end{cases} \quad (5)$$

Lý do vì khi đã bỏ chai i , chai liền trước đó phải bỏ phải là chai j nằm trong phạm vi từ $i - k$ tới $i - 1$.

Phép tính $\min_{i-k \leq j \leq i-1} \{f[j]\} + a[i]$ có thể tính được nhanh nhờ cây quản lý phạm vi 1 chiều, do đó là truy vấn giá trị nhỏ nhất trong mảng f từ vị trí $i - k$ tới vị trí $i - 1$.

Thời gian thực hiện giải thuật $O(n \log n)$.

Chương trình: BOTTLES.PAS

Cách khác: Có thể sử dụng Heap để tính min.

4.2.2. Kế hoạch thuê nhân công

(Nguồn bài: thầy Nguyễn Đức Nghĩa, 2001)

Một dự án phần mềm cần triển khai trong n tháng đánh số từ 1 tới n . Biết rằng:

- Bắt đầu vào một tháng, dự án có quyền thuê thêm nhân công. Để thuê mỗi nhân công cần một khoản chi phí H (trả cho nhà tuyển dụng).
- Mỗi nhân công được thuê sẽ được trả một khoản lương S mỗi tháng kể cả khi không làm việc.
- Kết thúc một tháng, dự án có quyền sa thải nhân công. Để sa thải mỗi nhân công cần trả một khoản chi phí D .
- Không có nhân công nào trước khi dự án bắt đầu. Mỗi tháng i cần tối thiểu a_i nhân công. Kết thúc tháng thứ n , toàn bộ nhân công phải bị sa thải.

Yêu cầu: Hãy giúp ông giám đốc dự án xây dựng kế hoạch thuê nhân công để dự án được hoàn thành với chi phí thuê nhân công ít nhất có thể.

Dữ liệu: Vào từ file văn bản PROJECT.INP

- Dòng 1 chứa số tháng n ($1 \leq n \leq 10^5$)
- Dòng 2 chứa ba số nguyên dương H, S, D ($H, S, D \leq 10^6$)
- Dòng 3 chứa n số nguyên dương a_1, a_2, \dots, a_n ($\forall i: a_i \leq 10^6$)

Kết quả: Ghi ra file văn bản PROJECT.OUT

- Dòng 1: Ghi chi phí tối thiểu tìm được
- Dòng 2: Ghi n số, số thứ i là số nhân công làm trong dự án tại tháng thứ i

Ví dụ

PROJECT.INP	PROJECT.OUT
3	265
4 5 6	10 10 11
10 9 11	

Gợi ý lời giải:

Lời giải dựa trên thuật toán tham lam:

Ở tháng 1 ta sẽ thuê đúng a_1 , giả sử trong phương án tối ưu ta có k_i nhân công ở tháng thứ i , ta sẽ quyết định xem ta sẽ thuê thêm hay sa thải bớt nhân công khi kết thúc tháng i .

Gọi $T = \left\lceil \frac{H+D}{S} \right\rceil$, T gọi là ngưỡng sa thải, tức là nếu một nhân công không làm việc nhiều hơn T tháng thì sa thải nhân công đó rồi sau này thuê lại sẽ có chi phí thấp hơn. Vậy từ khi kết thúc tháng i , ta xét xa hơn về sau $T + 1$ tháng: từ tháng $i + 1$ tới tháng $i + T + 1$, nếu nhu cầu nhân công tháng cao nhất trong phạm vi đó là M thì:

Nếu $M \geq k_i$, ta sẽ giữ nguyên k_i nhân công trong tháng i

Nếu $M < k_i$, ta sa thải bớt $k_i - M$ nhân công vì những nhân công này sẽ ngồi không nhiều hơn T tháng.

Việc tính nhu cầu nhân công tháng cao nhất trong phạm vi từ tháng $i + 1$ tới tháng $i + T + 1$ có thể thực hiện hiệu quả bởi cây quản lý phạm vi một chiều.

Thời gian thực hiện giải thuật $O(n \log n)$.

Chương trình: PROJECT.PAS

Cách khác: Có thể sử dụng Heap để tính max, nhưng cài đặt sẽ khó hơn.

4.2.3. Hoán vị Josephus

(Nguồn bài: cô Hồ Cẩm Hà, 2000)

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) :

Dữ liệu: Vào từ file văn bản JOSEPHUS.INP gồm một dòng chứa hai số nguyên dương $n, m \leq 10^5$.

Kết quả: Ghi ra file văn bản JOSEPHUS.OUT trên một dòng các số j_1, j_2, \dots, j_n tương ứng với hoán vị Josephus tìm được.

Các số trên một dòng của Input/Output files được/phải ghi cách nhau ít nhất một dấu cách

Ví dụ

JOSEPHUS.INP	JOSEPHUS.OUT
7 3	3 6 2 7 5 1 4

Gợi ý lời giải

Trước hết ta xây dựng phép ánh xạ chỉ số vòng: Nếu danh sách có k người, ta coi người ở vị trí $k + 1$ tương ứng với người 1, người ở vị trí $k + 2$ tương ứng với người 2... Cụ thể là nếu ta quan tâm tới vị trí p (có thể $p > k$) thì vị trí thật sự tương ứng là $(p - 1) \bmod k + 1$. Theo cách ánh xạ chỉ số như vậy, xét một lượt chơi còn k người:

- Nếu người đếm 1 là người thứ p thì người đếm m sẽ là người thứ $p' = p + m - 1$. Theo cách ánh xạ chỉ số vòng, chỉ số p' có giá trị thật sự bằng $(p + m - 2) \bmod k + 1$

- Sau khi người thứ p' bị loại, danh sách còn lại $k - 1$ người được đánh số lại từ 1. Khi đó người thứ p' lại là người đếm 1.

Với những nhận xét trên, bài toán tìm hoán vị Josephus (n, m) có thể giải quyết hiệu quả nếu sử dụng BST. 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.

```

p := 1;
for k := n downto 1 do
begin
  p := (p + m - 2) mod k + 1; //Chỉ số của người bị loại tại lượt chơi còn k người
  q := «Người đứng thứ p trong danh sách»; //Truy cập ngẫu nhiên
  Output ← q;
  «Xóa người thứ p trong danh sách»;
end;

```

Tuy vậy, lời giải bài toán này có thể cài đặt đơn giản hơn bằng cây quản lý đoạn: Mỗi người trong danh sách sẽ có một trong hai trạng thái: chưa bị loại/đã bị loại. Sau đó ta cần cài đặt một thủ tục làm hai nhiệm vụ:

- Định vị: nhận vào chỉ số p , xác định người đứng thứ p trong số những người chưa bị loại là người mang số hiệu q nào trong danh sách ban đầu.
- Loại bỏ: Đánh dấu loại bỏ người mang số hiệu q .

Thời gian thực hiện giải thuật $O(n \log n)$.

Chương trình: JOSEPHUS.PAS

4.2.4. Dãy con tăng

Cho dãy số nguyên dương $A = (a_1, a_2, \dots, a_n)$, phần tử a_i có trọng số là w_i . Mỗi dãy $(a_{i_1}, a_{i_2}, \dots, a_{i_k})$ thỏa mãn:

$$\begin{cases} 1 \leq i_1 < i_2 < \dots < i_k \leq n \\ a_{i_1} < a_{i_2} < \dots < a_{i_k} \end{cases}$$

được gọi là một dãy con tăng của dãy A . Chú ý rằng dãy chỉ gồm duy nhất một phần tử của A cũng được gọi là một dãy con tăng của dãy A .

Yêu cầu: Trong số các dãy con tăng của dãy A hãy chỉ ra một dãy có tổng trọng số các phần tử là lớn nhất có thể.

Dữ liệu: Vào từ file văn bản IS.INP

- Dòng 1 chứa số nguyên dương $n \leq 10^5$
- Dòng 2 chứa n số nguyên dương a_1, a_2, \dots, a_n theo đúng thứ tự đó ($\forall i: a_i \leq 10^5$)
- Dòng 3 chứa n số nguyên dương w_1, w_2, \dots, w_n theo đúng thứ tự đó ($\forall i: w_i \leq 10^9$)

Kết quả: Ghi ra file văn bản IS.OUT

- Dòng 1 ghi số phần tử trong dãy con tăng tìm được (m)
- Dòng 2 ghi m chỉ số của các phần tử được chọn theo thứ tự tăng dần

Các số trên một dòng của Input/Output files được/phải ghi cách nhau ít nhất một dấu cách

Ví dụ

IS.INP	IS.OUT
10	6
1 2 3 6 4 5 9 6 7 8	1 2 3 5 6 7
11 22 33 66 44 55 999 66 77 88	

Gợi ý lời giải

Khi các trọng số đều bằng 1, ta quy về bài toán tìm dãy con tăng dài nhất. Đây là bài toán tổng quát hơn.

Trước hết ta xây dựng công thức truy hồi: $f[i]$ là trọng số lớn nhất của dãy con tăng kết thúc tại $a[i]$

$$f[i] = \max_{j < i, a_j < a_i} \{f[j]\} + w_i$$

Khởi tạo một mảng $g[1 \dots 10^5]$ bằng 0. Ta sẽ triển khai tính toán từ $f[1]$ tới $f[n]$. Mỗi khi tính xong một $f[i]$, ta lưu lại $g[a[i]] := f[i]$ nếu $f[i]$ lớn hơn giá trị hiện có của $g[a[i]]$. Việc tính $\max_{j < i, a_j < a_i} \{f[j]\}$ quy về việc truy vấn giá trị lớn nhất trong phạm vi từ g_1 tới g_{a_i-1} . Điều này có thể thực hiện dễ dàng bằng cây chỉ số nhị phân

Thời gian thực hiện giải thuật $O(n \log n)$.

Chương trình: IS.PAS

Nhận xét: Kích thước a_i có thể cho lớn tùy ý, ta có thể gán lại giá trị cho dãy A từ 1 tới n mà không ảnh hưởng đến kết quả, tuy nhiên sẽ mất thêm thao tác sắp xếp và thống kê.

4.2.5. Dãy nghịch thế

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 $t = (2, 1, 0, 1, 1, 0)$

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

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

Vấn đề đặt ra là :

- Cho trước một dãy hoán vị x , hãy tìm dãy nghịch thế của x
- Cho trước một dãy nghịch thế t , hãy tìm dãy hoán vị nhận t làm dãy nghịch thế.

Dữ liệu: Vào từ file văn bản IVECTOR.INP gồm 3 dòng:

- Dòng 1: Chứa số nguyên dương $n \leq 10^5$.
- Dòng 2: Chứa dãy hoán vị x gồm n số x_1, x_2, \dots, x_n
- Dòng 3: Chứa dãy nghịch thế t : gồm n số t_1, t_2, \dots, t_n

Kết quả: Ghi ra file văn bản IVECTOR.OUT gồm 2 dòng:

- Dòng 1: Ghi lần lượt từng phần tử của dãy nghịch thế của x
- Dòng 2: Ghi lần lượt từng phần tử của dãy hoán vị của t

Các số trên một dòng của Input/Output files được/phải ghi cách nhau ít nhất một dấu cách

IVECTOR.INP	IVECTOR.OUT
6	0 0 0 0 0 0
1 2 3 4 5 6	3 2 1 6 4 5
2 1 0 1 1 0	

Gợi ý lời giải:

Từ dãy hoán vị X tìm dãy nghịch thế T :

Khởi tạo lại dãy X toàn bằng 0, điền lần lượt các số từ 1 tới n trở lại dãy X . Mỗi khi điền số i vào vị trí j , ta đếm số lượng số 0 đứng trước vị trí j , số lượng đó

chính là t_i . Việc đếm số lượng số 0 đứng trước vị trí j có thể thực hiện bằng cây quản lý phạm vi 1 chiều.

Từ dãy nghịch thế T khôi phục dãy hoán vị X :

Khởi tạo dãy X toàn bằng 0, xét lần lượt $t_1 \dots t_n$, khi xét tới t_i , thay số 0 thứ $t_i + 1$ trong dãy X bởi số i . Việc xác định số 0 thứ $t_i + 1$ có thể thực hiện bằng cây quản lý phạm vi 1 chiều.

Thời gian thực hiện giải thuật $O(n \log n)$.

Chương trình: IVECTOR.PAS

4.2.6. Xóa chữ số

Cho một xâu ký tự S gồm n chữ số 1, các ký tự trong xâu S được đánh số từ 1 tới n theo thứ tự từ trái qua phải. Xét lệnh $\text{Fill}(i, j, c)$: Trong đó i, j là các số nguyên dương, $1 \leq i \leq j \leq n$ và c là một chữ số $\in \{1, 2, \dots, 9\}$: Điền ký tự c vào xâu S bắt đầu từ vị trí i tới vị trí j . Các chữ số mới điền vào sẽ đè lên các chữ số đang có trong xâu S .

Ví dụ với $n = 6$

$$\begin{array}{rcl} 111111 & \xrightarrow{\text{Fill}(4,6,5)} & 111555 \\ 111555 & \xrightarrow{\text{Fill}(1,3,2)} & 222555 \\ 222555 & \xrightarrow{\text{Fill}(3,4,9)} & 229955 \end{array}$$

Cho biết trước m lệnh Fill và thứ tự thực hiện của chúng. Với một số nguyên dương $k < n$, hãy xóa đi k ký tự trong xâu S (sau m lệnh Fill đã cho) để được một xâu T gồm $n - k$ ký tự là biểu diễn thập phân của một số lớn nhất có thể.

Dữ liệu: Vào từ file văn bản FILLCHAR.INP

- Dòng 1 chứa ba số nguyên dương n, m, k ($k < n \leq 10^6; m \leq 10^5$)
- m dòng tiếp theo, dòng thứ p chứa ba số nguyên i_p, j_p, c_p cho biết lệnh Fill thứ p là $\text{Fill}(i_p, j_p, c_p)$ ($1 \leq i_p \leq j_p \leq n; 1 \leq c_p \leq 9$).

Các số trên một dòng của Input file được ghi cách nhau ít nhất một dấu cách

Kết quả: Ghi ra file văn bản FILLCHAR.OUT xâu T tìm được

Ví dụ

FILLCHAR.INP	FILLCHAR.OUT
6 3 2	9955
4 6 5	
1 3 2	
3 4 9	

Gợi ý lời giải: Nếu xác định được dãy chữ số cuối, thuật toán xóa chữ số là thuật toán quen thuộc sử dụng stack. Để xác định dãy chữ số cuối, ta thực hiện các lệnh Fill theo thứ tự ngược lại, nhưng chỉ điền lại những chữ số chưa được điền trước đó. Có thể dùng BST (điền xong vị trí nào xóa luôn vị trí đó khỏi BST) hoặc dùng cây quản lý đoạn (chỉ điền vào đoạn sơ cấp nếu đoạn đó chưa được điền).

Thời gian thực hiện giải thuật $O(m \log n)$

Cách giải khác: Có thể dùng cấu trúc dữ liệu rùng các tập rời nhau để hợp nhất các khoảng đã điền, thời gian thực hiện giải thuật tốt hơn $O(m\alpha(n))$.

Chương trình FILLCHAR.PAS

4.2.7. Nuôi cấy vi khuẩn

Phòng thí nghiệm XYZ thực hiện nuôi cấy một loại vi khuẩn trên một bảng ô vuông kích thước $n \times n$ với các dòng và các cột đánh số từ 1 tới n . Số cá thể vi khuẩn trong một ô vuông ban đầu là 0 và người ta có thể tiến hành cấy thêm một số cá thể vi khuẩn trong mỗi ô.

Có hai loại chỉ thị:

- Chỉ thị S $x y a$: Cấy thêm a cá thể vi khuẩn vào ô (x, y) ($1 \leq a \leq 10^5$)
- Chỉ thị Q $x_1 y_1 x_2 y_2$: Yêu cầu cho biết tổng số cá thể vi khuẩn trong các ô (x, y) nằm trong phạm vi $x_1 \leq x \leq x_2, y_1 \leq y \leq y_2$

Yêu cầu: Hãy nhận vào một dãy m chỉ thị thực hiện tuần tự và trả lời tất cả các chỉ thị loại Q.

Dữ liệu: Vào từ file văn bản BACTER.INP

- Dòng 1 chứa hai số nguyên dương $n, m \leq 10^5$
- m dòng tiếp theo, mỗi dòng chứa một chỉ thị theo thứ tự thực hiện

Kết quả: Ghi ra file văn bản BACTER.OUT ứng với mỗi chỉ thị Q, ghi ra một dòng câu trả lời là số lượng cá thể vi khuẩn tương ứng tính được.

Ví dụ

BACTER. INP	BACTER. OUT
4 7	5
S 2 2 2	16
S 4 3 3	7
Q 2 2 4 4	
S 1 2 5	
S 4 3 6	
Q 1 1 4 4	
Q 1 1 2 2	

Gợi ý lời giải:

Đây là bài toán thuần túy truy vấn phạm vi 2 chiều, mở rộng của bài thi IOI 2001: Mobile

Theo đáp án của BTC IOI 2001, cấu trúc sơ cấp và cấu trúc thứ cấp đều là cây chỉ số nhị phân vì kích thước m, n tương đối nhỏ. Đối với bài toán này ta chỉ đưa vào một sửa đổi: Cấu trúc sơ cấp là cây chỉ số nhị phân còn cấu trúc thứ cấp là cây splay

Thời gian thực hiện giải thuật $O(n \log^2 n)$.

Chương trình: BACTER.PAS

4.2.8. Tham quan

Một hướng dẫn viên đưa vợ chồng giáo sư X đi du lịch bằng ô tô. Ngoại trừ điểm xuất phát và điểm kết thúc, đường đi phải qua n thành phố đánh số từ 1 tới n theo đúng thứ tự trên hành trình. Thành phố i có a_i di tích lịch sử và b_i trung tâm mua sắm.

Vợ chồng giáo sư X muốn tham quan một số thành phố trên đường đi (những thành phố khác chỉ đi qua mà không dừng lại). Mỗi khi tham quan một thành phố, giáo sư X yêu cầu điểm tham quan tiếp theo (nếu có) phải có nhiều di tích lịch sử hơn, trong khi bà vợ ông ta lại muốn điểm tham quan tiếp theo phải có nhiều trung tâm mua sắm hơn thành phố hiện tại.

Để có được một chuyến đi thú vị, hãy giúp người hướng dẫn viên chọn ra một số nhiều nhất các thành phố để tham quan sao cho thỏa mãn được yêu cầu của cả hai vợ chồng giáo sư X. Cụ thể là bạn cần chọn số m lớn nhất và dãy chỉ số $1 \leq i_1 < i_2 < \dots < i_m \leq n$ sao cho:

$$\begin{cases} a_{i_1} < a_{i_2} < \dots < a_{i_m} \\ b_{i_1} < b_{i_2} < \dots < b_{i_m} \end{cases}$$

Dữ liệu: Vào từ file văn bản GUIDE.INP

- Dòng 1 chứa số nguyên dương $n \leq 10^5$
- Dòng 2 chứa n số nguyên a_1, a_2, \dots, a_n ($\forall i: 0 \leq a_i \leq 10^9$)
- Dòng 3 chứa n số tự nhiên b_1, b_2, \dots, b_n ($\forall i: 0 \leq b_i \leq 10^9$)

Kết quả: Ghi ra file văn bản GUIDE.OUT

- Dòng 1 ghi số thành phố được chọn (m)
- Dòng 2 ghi chỉ số của m thành phố được chọn theo thứ tự tăng dần

Các số trên một dòng của Input/Output files được/phải ghi cách nhau ít nhất một dấu cách

Ví dụ

GUIDE.INP	GUIDE.OUT
9	4
1 2 3 7 5 4 8 6 9	1 4 7 9
1 7 9 2 4 3 5 6 8	

Gợi ý lời giải:

Coi mỗi cặp (a_i, b_i) là một điểm trên mặt phẳng, gọi f_i là số thành phố nhiều nhất thăm được khi đi tới thành phố i . Mỗi khi tính xong f_i ta gán cho điểm (a_i, b_i) nhãn f_i nếu f_i lớn hơn nhãn cũ. Khi tính lần lượt các giá trị f_1, \dots, f_n , giá trị f_i sẽ được tính bằng cách lấy nhãn lớn nhất của các điểm đã có nằm trái dưới điểm (a_i, b_i) . Điều này có thể thực hiện bằng cây quản lý phạm vi hai chiều.

Một cách khác là sử dụng n cây quản lý phạm vi một chiều, phương pháp này giảm chi phí bộ nhớ xuống còn $O(n)$ nhưng thời gian thực hiện giải thuật vẫn là $O(n \log^2 n)$.

Chương trình: GUIDE.PAS

5. Kết luận

Cây quản lý phạm vi một cấu trúc dữ liệu quan trọng và có nhiều ứng dụng. Đối với những học viên môn cấu trúc dữ liệu và giải thuật, cây quản lý phạm vi là một ví dụ điển hình cho loại cấu trúc dữ liệu đệ quy, chia để trị. Việc cài đặt các loại cấu trúc dữ liệu này cũng là những bài tập tốt để rèn luyện kỹ thuật lập trình.

Trong phạm vi một chuyên đề, chúng tôi chỉ có thể giới thiệu sơ lược những nét chính và đặc trưng của cấu trúc dữ liệu cây quản lý phạm vi cùng một số ví dụ đơn giản, những vấn đề chi tiết và hệ thống bài tập luyện tập có thể tìm thấy trong danh mục tài liệu tham khảo

Tài liệu tham khảo

- 1 Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D. *Data Structures and Algorithms*. Addison Wesley, 1983.
- 2 Bentley, J.L. *Solution to Klee's rectangle problems*. Carnegie-Mellon university, Pittsburgh, PA, 1977.
- 3 Cormen, Thomas H., E., Leiserson Charles, L., Rivest Ronald, and Clifford, Stein. *Introduction to Algorithms*. MIT Press, 2001.
- 4 de Berg, Mark, Cheong, Otfried, van Kreveld, Marc, and Overmars, Mark. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.
- 5 Fenwick, Peter M. A New Data Structure for Cumulative Frequency Tables. *Software: Practice and Experience*, 24 (1994), 327-336.
- 6 Fredman, Michael Lawrence and Tarjan, Robert Endre. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34, 3 (1987), 596-615.
- 7 Seidel, Raimund G. and Aragon, Cecilia R. Randomized search trees. *Algorithmica*, 16 (1996), 464-497.
- 8 Sleator, Daniel Dominic and Tarjan, Robert Endre. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 32, 3 (1985), 652-686.
- 9 Vuillemin, Jean. A data structure for manipulating priority queues. *Communications of the ACM*, 21, 4 (1978), 309-314.

Phụ lục 1: Cây nhị phân tìm kiếm ngẫu nhiên

Phụ lục này cung cấp một lựa chọn khác trong việc cài đặt cây nhị phân tìm kiếm. Cấu trúc dữ liệu Treaps qua các thử nghiệm đạt tốc độ tốt nhất khi so sánh với cây đỏ đen, cây AVL và cây Splay khi dữ liệu phân bố ngẫu nhiên. Việc cài đặt Treaps cũng khá đơn giản, không phức tạp hơn cây splay là mấy và dễ hơn nhiều so với cây đỏ đen và cây AVL.

1. Độ cao trung bình của BST

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 nếu xác suất tìm kiếm phân phối đều trên tập các giá trị khóa thì 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.

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.

Treap là một dạng BST mà cấu trúc của nó không phụ thuộc vào thứ tự chèn/xóa. Nói rõ hơn là cho dù chúng ta chèn/xóa các khóa vào Treap theo thứ tự nào, cấu trúc của Treap vẫn sẽ như một BST khi chúng ta chèn các khóa vào theo trật tự ngẫu nhiên. Các thực nghiệm cũng cho thấy Treap có tốc độ tốt nhất khi so sánh với cây AVL, cây đẻ đơn hay cây Splay.

Cho mỗi nút của BST thêm một trường *priority* gọi là “độ ưu tiên”. Độ ưu tiên của mỗi nút là một số dương. Khi đó Treap[†] [7] được định nghĩa là một BST thỏa mãn tính chất của Heap. Cụ thể là với x và y là con trỏ tới hai nút trên Treap:

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

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 4

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

* 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”

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.

Định lý 5

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 lý 4. Tính duy nhất của cấu trúc Treap này có thể chứng minh bằng quy nạp theo số nút: 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 độ ưu tiên (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.

3. Các thao tác trên Treap

3.1. 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

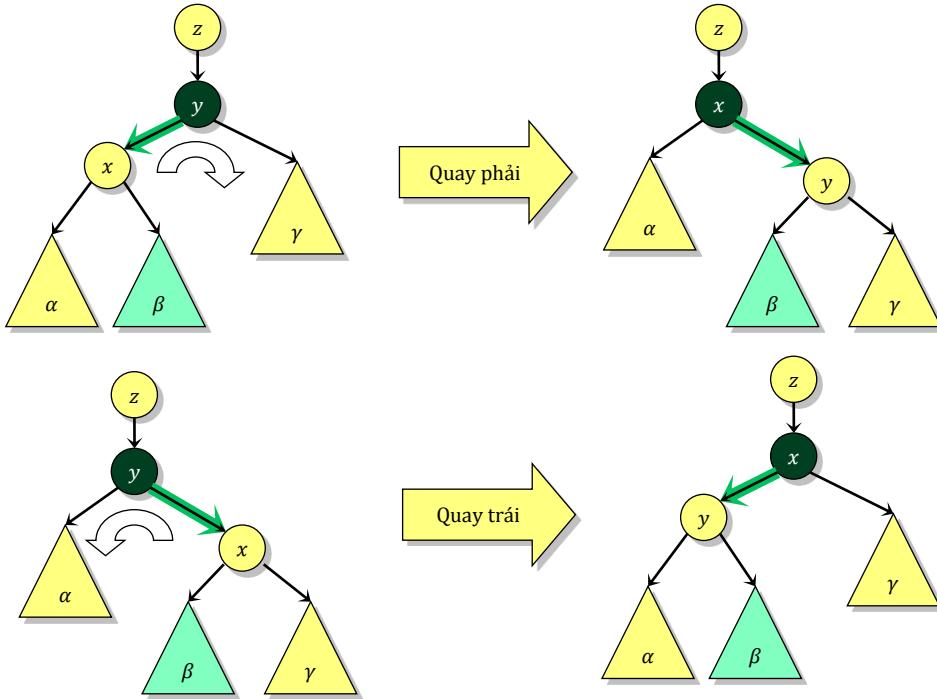
```
type
  PNode = ^TNode; //Kiểu con trỏ tới một nút
  TNode = record
    key: TKey;
    P, L, R: PNode;
    priority: Integer;
  end;
var
  sentinel: TNode;
  nilT: PNode; //Con trỏ tới nút đặc biệt
  root: PNode; //Con trỏ tới nút gốc
begin
  nilT := @sentinel;
  nilT^.priority := 0; //nilT có độ ưu tiên bằng 0
  ...
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); //Lấy ngẫu nhiên từ 1 tới MaxInt
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.

Ta lại cần nhắc lại về thao tác $UpTree(x)$: Đẩy x lên phía gốc cây, giảm độ sâu của x đi 1.



```

procedure SetLink(parent, child: PNode; InRight: Boolean);
begin
  child^.P := parent;
  if Right then parent^.R := child
  else parent^.L := child;
end;

procedure UpTree(x: PNode);
var
  y, z: PNode;
begin
  y := x^.P; //y^ là nút cha của x^
  z := y^.P; //z^ là nút cha của y^
  if x = y^.L then //Quay phải
    begin
      SetLink(y, x^.R, False); //Chuyển nhánh con phải của x^ sang làm con trái y^
      SetLink(x, y, True); //Cho y^ làm con phải x^
    end
  else //Quay trái
    begin
      SetLink(y, x^.L, True); //Chuyển nhánh con trái của x^ sang làm con phải y^
      SetLink(x, y, False); //Cho y^ làm con trái x^
    end;
  SetLink(z, x, z^.R = y); //Móc nối x^ vào làm con z^ thay cho y^
end;

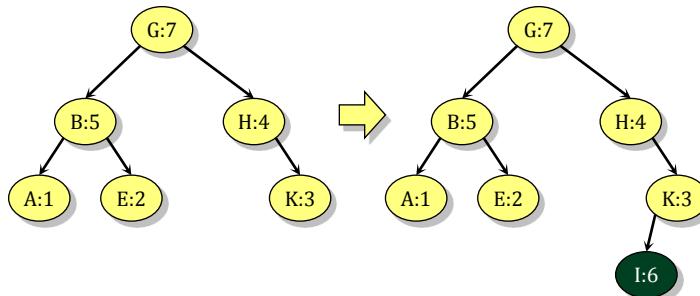
```

3.2. Chèn

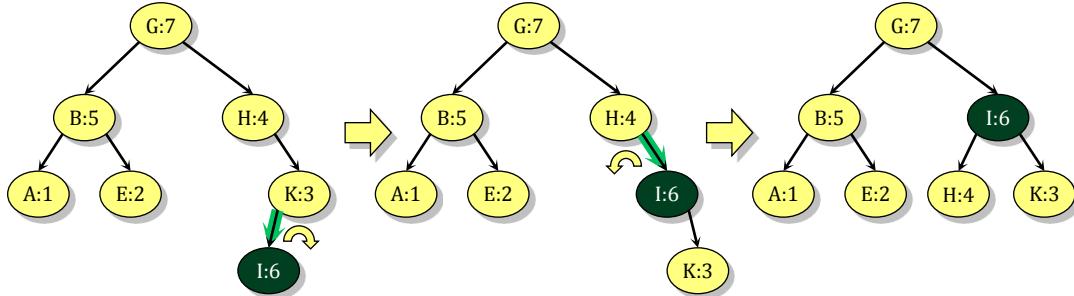
Phép chèn trên Treap trước hết thực hiện như phép chèn trên BST để chèn khóa vào một nút lá. Nút lá x mới chèn vào sẽ được gán một độ ưu tiên ngẫu nhiên. Tiếp theo là phép hiệu chỉnh Treap: chừng nào thấy x mang độ ưu tiên lớn hơn nút cha y của nó (vi phạm thứ tự Heap) ta thực hiện lệnh $UpTree(x)$ để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

```
//Chèn khóa k vào Treap
procedure Insert(k: TKey);
var
  x, y: PNode;
begin
  //Thực hiện phép chèn như trên BST
  y := nilT; x := root; //Bắt đầu từ gốc
  while x ≠ nilT do
    begin
      y := x;
      if k < x^.key then x := x^.L //Chèn vào nhánh trái
      else x := x^.R; //Chèn vào nhánh phải
    end;
  New(x); //Tạo nút mới chứa k
  x^.key := k;
  x^.L := nilT; x^.R := nilT;
  SetLink(y, x, k ≥ y^.key); //Móc nối vào BST
  x^.priority := RP; //Gán độ ưu tiên ngẫu nhiên
  //Chỉnh Treap
  while (x^.P ≠ nilT) and (x^.priority > x^.P^.priority) do
    UpTree(x);
  if x^.P = nilT then root := x; //Cập nhật lại gốc nếu x trở thành gốc
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



Tiếp theo là hai phép $UpTree$ để chuyển nút I:6 về vị trí đúng trên 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 $h - 1$ với h là độ cao của Treap), nhưng người ta đã chứng minh được định lý sau:

Định lý 6

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

3.3. Xóa

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

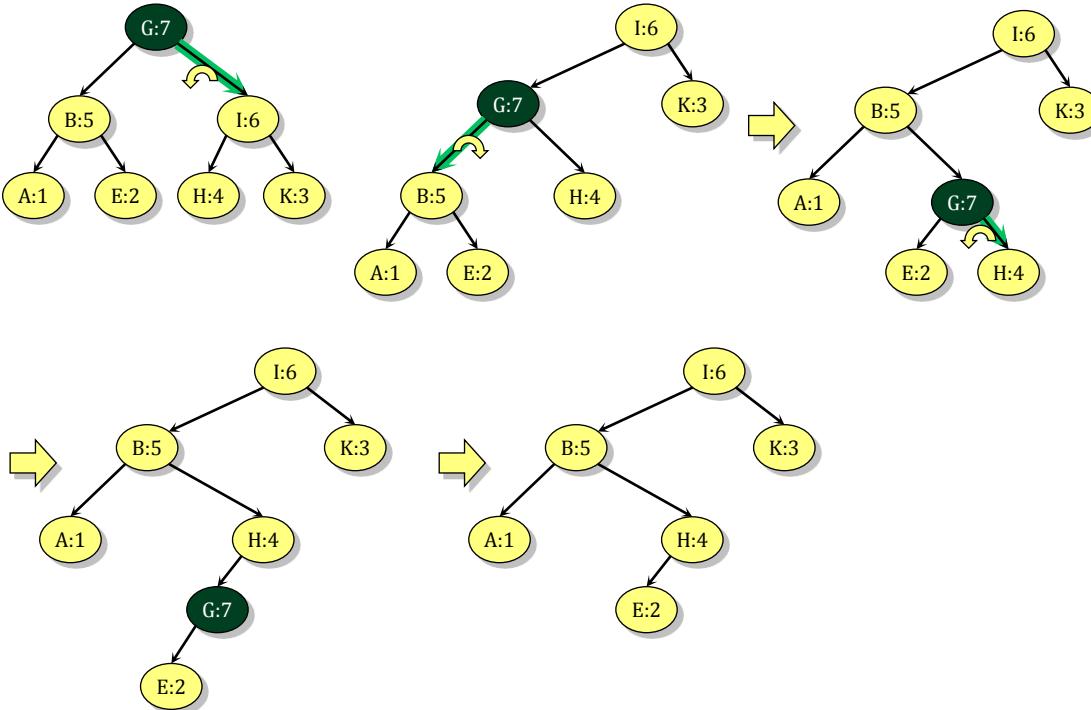
- Nếu x có ít hơn hai nhánh con, ta lấy nút con (nếu có) của x lên thay cho x và xóa nút x .
- Nếu x có đúng hai nhánh con, gọi y là nút con mang độ ưu tiên lớn hơn trong hai nút con, thực hiện phép *UpTree*(y) để kéo nút x xuống sâu phía dưới lá và lặp lại cho tới khi x chỉ còn một nút con. Việc xóa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

```

procedure Delete(x: PNode);
var
  y, z: PNode;
begin
  while (x^.L ≠ nilT) and (x^.R ≠ nilT) do //Chứng nào x có 2 nút con, đẩy nút con mang độ ưu tiên cao hơn lên
    if x^.L^.priority > x^.R^.priority then UpTree(x^.L)
    else UpTree(x^.R);
  while root^.P ≠ nilT do root := root^.P; //Cập nhật lại gốc nếu ban đầu root = x và đã bị kéo xuống
  //Bây giờ x chỉ có tối đa một nút con, xác định y là nút con (nếu có) của x
  if x^.L ≠ nilT then y := x^.L
  else y := x^.R;
  z := x^.P; //z là nút cha của x
  SetLink(z, y, z^.R = x); //Cho y làm con của z thay cho x
  if root = x then root := y; //Cập nhật lại gốc nếu x đang là 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



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 7

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

Phụ lục 2: Các mã nguồn chương trình

BOTTLES.PAS

```
{$MODE OBJFPC}
program Bottles;
const
  InputFile = 'BOTTLES.INP';
  OutputFile = 'BOTTLES.OUT';
  maxN = Round(1E5);
  maxV = Round(1E9);
  infty = maxN * maxV + 1;
type
  THeap = record
    nItems: Integer;
    items: array[1..maxN + 2] of Integer;
  end;
var
  a: array[1..maxN + 1] of Integer;
  f: array[1..maxN + 1] of Int64;
  tree: array[1..4 * maxN] of Int64;
  l, h: array[1..4 * maxN] of Integer;
  leaf: array[1..maxN] of Integer;
  n, k: Integer;
  Sum, Res: Int64;

procedure Enter;
var
  fi: TextFile;
  i: Integer;
begin
  AssignFile(fi, InputFile); Reset(fi);
  try
    ReadLn(fi, n, k);
    Sum := 0;
    a[n + 1] := 0;
    for i := 1 to n do
      begin
        Read(fi, a[i]); Inc(Sum, a[i]);
      end;
  finally
    CloseFile(fi);
  end;
end;

procedure Build(x: Integer; low, high: Integer);
var
  middle: Integer;
begin
  l[x] := low; h[x] := high;
  tree[x] := 0;
  if low = high then
    leaf[low] := x
  else
    begin
```

```

        middle := (low + high) shr 1;
        Build(x * 2, low, middle);
        Build(x * 2 + 1, middle + 1, high);
    end;
end;

function Min(p, q: Int64): Int64;
begin
    if p < q then Result := p else Result := q;
end;

procedure Update(i: Integer; f: Int64);
var
    x: Integer;
begin
    x := leaf[i];
    tree[x] := f;
    while x > 1 do
    begin
        x := x shr 1;
        tree[x] := Min(tree[2 * x], tree[2 * x + 1]);
    end;
end;

function Query(i, j: Integer): Int64;

    function Request(x: Integer): Int64;
    begin
        if (h[x] < i) or (l[x] > j) then Exit(infty);
        if (l[x] >= i) and (h[x] <= j) then Exit(tree[x]);
        Result := Min(Request(2 * x), Request(2 * x + 1));
    end;

begin
    Result := Request(1);
end;

procedure Solve;
var
    i: Integer;
begin
    for i := 1 to n + 1 do
    begin
        if i <= k then f[i] := a[i]
        else f[i] := Query(i - k, i - 1) + a[i];
        if i <= n then
            Update(i, f[i]);
    end;
    res := Sum - f[n + 1];
end;

procedure PrintResult;
var
    fo: TextFile;
    i, nSel: Integer;
    temp: Int64;
begin

```

```

temp := f[n + 1];
nSel := 0;
for i := n downto 1 do
  if f[i] = temp then
    begin
      temp := temp - a[i];
      a[i] := -1;
    end
  else
    Inc(nSel);
AssignFile(fo, OutputFile); Rewrite(fo);
try
  WriteLn(fo, nSel, ' ', res);
  for i := 1 to n do
    if a[i] <> -1 then Write(fo, i, ' ');
finally
  CloseFile(fo);
end;
end;

begin
  Enter;
  Build(1, 1, n);
  Solve;
  PrintResult;
end.

```

PROJECT.PAS

```

{$MODE OBJFPC}
program Employment;
uses Math;
const
  InputFile  = 'PROJECT.INP';
  OutputFile = 'PROJECT.OUT';
  maxN = 100000;
var
  a, b: array[1..maxN + 1] of Integer;
  info: array[1..4 * maxN] of Integer;
  n: Integer;
  H, S, D: Int64;
  Period: Integer;
  TotalCost: Int64;

procedure Enter;
var
  f: TextFile;
  i: Integer;
begin
  AssignFile(f, InputFile); Reset(f);
  try
    ReadLn(f, n);
    ReadLn(f, H, S, D);
    for i := 1 to n do Read(f, a[i]);
  finally
    CloseFile(f);
  
```

```

    end;
end;

procedure CalPeriod;
begin
  Period := (H + D) div S;
end;

procedure Build(x, l, h: Integer);
var
  m: Integer;
begin
  if l = h then
    info[x] := a[l]
  else
    begin
      m := (l + h) div 2;
      Build(x * 2, l, m);
      Build(x * 2 + 1, m + 1, h);
      info[x] := Max(info[x * 2], info[x * 2 + 1]);
    end;
end;

function Query(i, j: Integer): Integer;

function Request(x, l, h: Integer): Integer;
var
  m: Integer;
begin
  if (i > h) or (j < l) then Exit(0);
  if (i <= l) and (j >= h) then Exit(info[x]);
  m := (l + h) div 2;
  Result := Max(Request(x * 2, l, m), Request(x * 2 + 1, m + 1, h));
end;

begin
  if j > n then j := n;
  Result := Request(1, 1, n);
end;

procedure Solve;
var
  i, nRemove, e: Integer;
begin
  b[1] := a[1];
  b[n] := a[n];
  for i := 2 to n - 1 do
    if a[i] >= b[i - 1] then b[i] := a[i]
    else
      begin
        e := Query(i, i + Period);
        if b[i - 1] > e then nRemove := b[i - 1] - e
        else nRemove := 0;
        b[i] := b[i - 1] - nRemove;
      end;
  b[n + 1] := 0;
  TotalCost := b[1] * (H + S);
end;

```

```

for i := 2 to n + 1 do
begin
  if b[i] >= b[i - 1] then
    TotalCost := TotalCost + (b[i] - b[i - 1]) * H
  else
    TotalCost := TotalCost + (b[i - 1] - b[i]) * D;
  TotalCost := TotalCost + b[i] * S;
end;
end;

procedure PrintResult;
var
  f:TextFile;
  i: Integer;
begin
  AssignFile(f, OutputFile); Rewrite(f);
  try
    WriteLn(f, TotalCost);
    for i := 1 to n do Write(f, b[i], ' ');
  finally
    CloseFile(f);
  end;
end;

begin
  Enter;
  CalPeriod;
  Build(1, 1, n);
  Solve;
  PrintResult;
end.

```

JOSEPHUS.PAS

```

{$MODE OBJFPC}
program JosephusPermutation;
const
  InputFile = 'JOSEPHUS.INP';
  OutputFile = 'JOSEPHUS.OUT';
  maxN = 100000;
var
  n, m: Integer;
  remain: array[1..4 * maxN - 1] of Integer;
  res: array[1..maxN] of Integer;

procedure Enter;
var
  f: TextFile;
begin
  AssignFile(f, InputFile); Reset(f);
  try
    ReadLn(f, n, m);
  finally
    CloseFile(f);
  end;
end;

```

```

procedure Init(node: Integer; L, H: Integer);
var
  M: Integer;
begin
  if L = H then remain[node] := 1
  else
    begin
      M := (L + H) div 2;
      Init(node * 2, L, M);
      Init(node * 2 + 1, M + 1, H);
      remain[node] := H - L + 1;
    end;
end;

function SetMark(i: Integer): Integer;
var
  L, M, H: Integer;
  node, left: Integer;
begin
  node := 1;
  L := 1; H := n;
  repeat
    M := (L + H) div 2;
    left := node * 2;
    if i <= remain[left] then
      begin
        node := left;
        H := M
      end
    else
      begin
        node := left + 1;
        L := M + 1;
        Dec(i, remain[left]);
      end;
  until L = H;
  Result := L;
  repeat
    Dec(remain[node]);
    node := node div 2;
  until node = 0;
end;

procedure FindJosephusPermutation;
var
  Node, p, k, i: Integer;
begin
  p := 1; k := n;
  for i := 1 to n do
    begin //Danh sách có k người
      p := (p + m - 2) mod k + 1; //Xác định số thứ tự của người bị loại
      res[i] := SetMark(p);
      Dec(k);
    end;
end;

procedure PrintResult;

```

```

var
  f: TextFile;
  i: Integer;
begin
  AssignFile(f, OutputFile); Rewrite(f);
  try
    for i := 1 to n do Write(f, res[i], ' ');
  finally
    CloseFile(f);
  end;
end;

begin
  Enter;
  Init(1, 1, n);
  FindJosephusPermutation;
  PrintResult;
end.

```

IS.PAS

```

{$MODE OBJFPC}
program MaxWeightIncreasingSequence;
const
  InputFile = 'IS.INP';
  OutputFile = 'IS.OUT';
  maxN = Round(1E5);
  maxV = Round(1E5);
var
  a: array[1..maxN] of Integer;
  w: array[1..maxN] of Integer;
  f: array[1..maxN + 1] of Int64;
  bit: array[1..maxV] of Int64;
  n: Integer;
  resSum: Int64;
  nsel: Integer;

procedure Enter;
var
  f: TextFile;
  i: Integer;
begin
  AssignFile(f, InputFile); Reset(f);
  try
    ReadLn(f, n);
    for i := 1 to n do Read(f, a[i]);
    ReadLn(f);
    for i := 1 to n do Read(f, w[i]);
  finally
    CloseFile(f);
  end;
end;

function Max(p, q: Int64): Int64;
begin
  if p > q then Result := p else Result := q;
end;

```

```

procedure Init;
begin
  FillQWord(bit[1], maxV, 0);
end;

procedure Update(x: Integer; value: Int64);
begin
  while x <= maxV do
    begin
      bit[x] := Max(bit[x], value);
      Inc(x, x and -x);
    end;
end;

function Query(x: Integer): Int64;
begin
  Result := 0;
  while x > 0 do
    begin
      Result := Max(Result, bit[x]);
      x := x and Pred(x);
    end;
end;

procedure Solve;
var
  i: Integer;
begin
  for i := 1 to n do
    begin
      f[i] := Query(a[i] - 1) + w[i];
      Update(a[i], f[i]);
    end;
end;

procedure PrintResult;
var
  fo: TextFile;
  temp: Int64;
  tempA: Integer;
  i: Integer;
begin
  temp := Query(maxV);
  tempA := maxV + 1;
  nSel := 0;
  for i := n downto 1 do
    if (f[i] = temp) and (a[i] < tempA) then
      begin
        temp := temp - w[i];
        w[i] := -1;
        tempA := a[i];
        Inc(nSel);
      end;
  AssignFile(fo, OutputFile); Rewrite(fo);
  try
    WriteLn(fo, nSel);
  
```

```

    for i := 1 to n do
      if w[i] = -1 then Write(fo, i, ' ');
  finally
    CloseFile(fo);
  end;
end;

begin
  Enter;
  Init;
  Solve;
  PrintResult;
end.

```

IVECTOR.PAS

```

{$MODE OBJFPC}
program InversionVector;
const
  InputFile  = 'IVECTOR.INP';
  OutputFile = 'IVECTOR.OUT';
  max = 100000;
type
  TArray = array[1..max] of Integer;
var
  bit: array[1..max] of Integer;
  x, t, pos: TArray;
  n: Integer;
  fi, fo: TextFile;

procedure OpenFiles;
begin
  AssignFile(fi, InputFile); Reset(fi);
  AssignFile(fo, OutputFile); Rewrite(fo);
  ReadLn(fi, n);
end;

procedure CloseFiles;
begin
  Close(fi); Close(fo);
end;

procedure BuildTree;
var
  i: Integer;
begin
  for i := 1 to n do
    bit[i] := i and -i;
end;

function CountAndMark(i: Integer): Integer;
var
  x: Integer;
begin
  x := Pred(i);
  Result := 0;
  while x > 0 do

```

```

begin
  Inc(Result, bit[x]);
  x := x and Pred(x);
end;
x := i;
while x <= n do
begin
  Dec(bit[x]);
  Inc(x, x and -x);
end;
end;

procedure FindInversionVector;
var
  i, j: Integer;
begin
  for i := 1 to n do Read(fi, x[i]);
  ReadLn(fi);
  BuildTree;
  for i := 1 to n do pos[x[i]] := i;
  for i := 1 to n do
  begin
    begin
      j := pos[i];
      t[i] := CountAndMark(j);
    end;
    for i := 1 to n do Write(fo, t[i], ' ');
    WriteLn(fo);
  end;
end;

function LocateAndMark(i: Integer): Integer;
var
  x, next, mask: Integer;
begin
  mask := 1;
  while mask shl 1 <= n do mask := mask shl 1;
  x := 0;
  while mask <> 0 do
  begin
    begin
      next := x + mask;
      if next <= n then
        begin
          if i = bit[next] then
            Result := next;
          if i > bit[next] then
            begin
              x := next;
              Dec(i, bit[next]);
            end;
        end;
      mask := mask shr 1;
    end;
  x := Result;
  while x <= n do
  begin
    Dec(bit[x]);
    Inc(x, x and -x);
  end;
end;

```

```

end;

procedure FindPermutationVector;
var
  i, j: Integer;
begin
  for i := 1 to n do Read(fi, t[i]);
  FillChar(x, sizeof(x), 0);
  BuildTree;
  for i := 1 to n do
    begin
      j := LocateAndMark(t[i] + 1);
      x[j] := i;
    end;
  for i := 1 to n do Write(fo, x[i], ' ');
end;

begin
  OpenFiles;
  try
    FindInversionVector;
    FindPermutationVector;
  finally
    CloseFiles;
  end;
end.

```

FILLCHAR.PAS

```

{$MODE OBJFPC}
program MaximumSubNumber;
uses Test143;
const
  InputFile = 'FILLCHAR.INP';
  OutputFile = 'FILLCHAR.OUT';
  maxN = 1000000;
  maxM = 100000;
type
  TFillCommand = record
    a, b: Integer;
    c: Byte;
  end;
  PNode = Integer;
  TTreeNode = record
    Key: Integer;
    Parent, Left, Right: PNode;
  end;
var
  s: array[1..maxN] of Byte;
  cmd: array[1..maxM] of TFillCommand;
  Tree: array[0..maxN + 2] of TTreeNode;
  Root: Integer;
  n, m, k, nNodes: Integer;
  Stack: array[0..maxN] of Integer;
  Top: Integer;

procedure Enter;

```

```

var
  f: TextFile;
  i: Integer;
begin
  AssignFile(f, InputFile); Reset(f);
  SetTextBuf(f, s, $FFFF);
  try
    ReadLn(f, n, m, k);
    for i := 1 to m do
      with cmd[i] do ReadLn(f, a, b, c);
  finally
    CloseFile(f);
  end;
end;

procedure SetLink(ParentNode, ChildNode: Integer; InLeft: Boolean);
begin
  Tree[ChildNode].Parent := ParentNode;
  if InLeft then Tree[ParentNode].Left := ChildNode
  else Tree[ParentNode].Right := ChildNode;
end;

procedure Init;
var
  k: Integer;

procedure Traversal(x: PNode);
begin
  if x shl 1 <= nNodes then
    begin
      Traversal(x shl 1);
      SetLink(x, x shl 1, True);
    end
  else SetLink(x, 0, True);
  Tree[x].Key := k; Inc(k);
  if x shl 1 or 1 <= nNodes then
    begin
      Traversal(x shl 1 or 1);
      SetLink(x, x shl 1 or 1, False);
    end
  else SetLink(x, 0, False);
end;

begin
  nNodes := n + 2;
  k := 0;
  Traversal(1); //Dien cac vi tri 0..n+1 vao n+2 nut
  Root := 1;
  FillChar(s[1], n * SizeOf(s[1]), 0);
end;

//Tim Node chua khoa nho nhat >= Position
function Search(Position: Integer): PNode;

//Tim Node chua khoa nho nhat >= Position trong cay goc x^
function SearchIn(x: PNode): PNode;
begin

```

```

if x = 0 then
begin
  Result := 0; Exit;
end;
if Tree[x].Key < Position then Result := SearchIn(Tree[x].Right)
else //x^.Key >= k
begin
  Result := SearchIn(Tree[x].Left);
  if Result = 0 then Result := x;
end;
end;

begin
  Result := SearchIn(Root);
end;

function Minimum(x: PNode): PNode;
begin
  while Tree[x].Left <> 0 do x := Tree[x].Left;
  Result := x;
end;

function Maximum(x: PNode): PNode;
begin
  while Tree[x].Right <> 0 do x := Tree[x].Right;
  Result := x;
end;

function Successor(x: PNode): PNode;
begin
  if Tree[x].Right <> 0 then Result := Minimum(Tree[x].Right)
  else
  begin
    Result := Tree[x].Parent;
    while Tree[Result].Left <> x do
    begin
      x := Result; Result := Tree[x].Parent;
    end;
  end;
end;
end;

procedure Delete(x: PNode);
var
  y, z: PNode;
begin
  if (Tree[x].Left <> 0) and (Tree[x].Right <> 0) then
  begin
    y := Maximum(Tree[x].Left);
    Tree[x].Key := Tree[y].Key;
    x := y;
  end;
  if Tree[x].Left <> 0 then y := Tree[x].Left
  else y := Tree[x].Right;
  z := Tree[x].Parent;
  SetLink(z, y, Tree[z].Left = x);
  if Root = x then Root := y;
end;

```

```

procedure FillBackward;
var
  i: Integer;
  Node, Next: PNode;
begin
  for i := m downto 1 do
    with Cmd[i] do
      begin
        Node := Search(a);
        while Tree[Node].Key <= b do
          begin
            Next := Successor(Node);
            s[Tree[Node].Key] := c;
            Delete(Node);
            Node := Next;
          end;
        end;
      end;
end;

procedure GetMaxV;
var
  i, Top, Remain: Integer;
begin
  Remain := k;
  Top := 0; Stack[0] := 10;
  for i := 1 to n do
    begin
      while (Remain > 0) and (s[i] > Stack[Top]) do
        begin
          Dec(Top);
          Dec(Remain);
        end;
      Inc(Top);
      Stack[Top] := s[i];
    end;
end;

procedure PrintResult;
var
  f: TextFile;
  i: Integer;
begin
  AssignFile(f, OutputFile); Rewrite(f);
  SetTextBuf(f, s, $FFFF);
  try
    for i := 1 to n - k do
      Write(f, Stack[i]);
  finally
    CloseFile(f);
  end;
end;

begin
  Enter;
  Init;
  FillBackward;

```

```

GetMaxV;
PrintResult;
end.

BACTER.PAS

{$MODE OBJFPC}
{$INLINE ON}
program Bacteria;
const
  InputFile = 'BACTER.INP';
  OutputFile = 'BACTER.OUT';
  maxN = Round(1E5);
  maxM = Round(1E5);
type
  PNode = ^TNode;
  TNode = record
    key: Integer;
    localsize: Int64;
    size: Int64;
    P, L, R: PNode;
  end;
var
  bit: array[1..maxN] of PNode;
  sentinel: TNode;
  nilT: PNode;
  fi, fo: TextFile;
  n, m: Integer;

procedure OpenFiles;
begin
  AssignFile(fi, InputFile); Reset(fi);
  AssignFile(fo, OutputFile); Rewrite(fo);
end;

procedure CloseFiles;
begin
  CloseFile(fi); Closefile(fo);
end;

procedure Enter;
var
  i: Integer;
begin
  nilT := @sentinel;
  nilT^.size := 0;
  nilT^.localsize := 0;
  ReadLn(fi, n, m);
  for i := 1 to n do
    bit[i] := nilT;
end;

procedure SetLink(parent, child: PNode; InRight: Boolean); inline;
begin
  child^.P := parent;
  if InRight then parent^.R := child
  else parent^.L := child;

```

```

end;

procedure Update(x: PNode); inline;
begin
  x^.size := x^.localsize + x^.L^.size + x^.R^.size;
end;

procedure UpTree(x: PNode);
var
  b, y, z: PNode;
begin
  y := x^.P; z := y^.P;
  if x = y^.L then
    begin
      b := x^.R;
      SetLink(y, b, False);
      SetLink(x, y, True);
    end
  else
    begin
      b := x^.L;
      SetLink(y, b, True);
      setLink(x, y, False);
    end;
  SetLink(z, x, z^.R = y);
  Update(y);
  Update(x);
end;

procedure Splay(x: PNode);
var
  y, z: PNode;
begin
  repeat
    y := x^.P;
    if y = nilT then Break;
    z := y^.P;
    if z <> nilT then
      if (x = y^.L) = (y = z^.L) then
        UpTree(y)
      else
        UpTree(x);
    UpTree(x);
  until False;
end;

procedure Split(T: PNode; y: Integer; var TL, TR: PNode);
var
  x, p: PNode;
begin
  x := nilT; p := nilT;
  while T <> nilT do
    begin
      p := T;
      if T^.key > y then
        begin
          x := T;

```

```

        T := T^.L;
    end
else
    T := T^.R;
end;
if x <> nilT then
begin
    Splay(x);
    TL := x^.L; TR := x;
    TR^.L := nilT;
    TL^.P := nilT;
    Update(TR);
end
else
begin
    if p <> nilT then
        Splay(p);
    TL := p; TR := nilT;
end;
end;

function Join(TL, TR: PNode): PNode;
begin
    if TR = nilT then Exit(TL);
    while TR^.L <> nilT do TR := TR^.L;
    Splay(TR);
    SetLink(TR, TL, False);
    Update(TR);
    Result := TR;
end;

procedure IncY(var T: PNode; y: Integer; a : Integer);
var
    TL, TR: PNode;
begin
    Split(T, y, TL, TR);
    New(T);
    T^.P := nilT;
    T^.localsize := a;
    T^.key := y;
    SetLink(T, TL, False);
    SetLink(T, TR, True);
    Update(T);
end;

procedure IncXY(x, y, a: Integer); inline;
begin
    while x <= n do
begin
    IncY(bit[x], y, a);
    Inc(x, x and -x);
end;
end;

function QueryY(var T: PNode; y1, y2: Integer): Int64;
var
    T1, T2, T3: PNode;

```

```

begin
  Split(T, y1 - 1, T1, T2);
  Split(T2, y2, T2, T3);
  Result := T2^.size;
  T := Join(T2, T3);
  T := Join(T1, T);
end;

function QueryXY(x, y1, y2: Integer): Int64; inline;
begin
  Result := 0;
  while x > 0 do
    begin
      Result := Result + QueryY(bit[x], y1, y2);
      x := x and pred(x);
    end;
end;

procedure Solve;
var
  iq: Integer;
  c: Char;
  x1, y1, x2, y2, x, y, a: Integer;
  temp1, temp2: Int64;
begin
  for iq := 1 to m do
    begin
      Read(fi, c);
      case Upcase(c) of
        'S':
        begin
          ReadLn(fi, x, y, a);
          IncXY(x, y, a);
        end;
        'Q':
        begin
          ReadLn(fi, x1, y1, x2, y2);
          temp1 := QueryXY(x1 - 1, y1, y2);
          temp2 := QueryXY(x2, y1, y2);
          WriteLn(fo, temp2 - temp1);
        end;
      end;
    end;
end;

procedure FreeMemory;
var
  i: Integer;

procedure FreeTree(x: PNode);
begin
  if x = nilT then Exit;
  FreeTree(x^.L); FreeTree(x^.R);
  Dispose(x);
end;

begin

```

```

    for i := 1 to n do FreeTree(bit[i]);
end;

begin
  OpenFiles;
  Enter;
  try
    Solve;
    FreeMemory;
  finally
    CloseFiles;
  end;
end.

```

GUIDE.PAS

```

{$MODE OBJFPC}
{$M 10000000}
program TwoLines;
const
  InputFile  = 'GUIDE.INP';
  OutputFile = 'GUIDE.OUT';
  maxN = 100000;
  maxV = 1000000000;
type
  PNode = ^TNode;
  TNode = record
    key: Integer;
    P, L, R: PNode;
  end;
var
  a, b: array[0..maxN] of Integer;
  trace: array[0..maxN] of Integer;
  tree: array[1..maxN + 1] of PNode;
  n, m: Integer;
  nilT: PNode;
  sentinel: TNode;

procedure Enter;
var
  f: TextFile;
  i: Integer;
begin
  AssignFile(f, InputFile); Reset(f);
  try
    ReadLn(f, n);
    for i := 1 to n do Read(f, a[i]);
    ReadLn(f);
    for i := 1 to n do Read(f, b[i]);
  finally
    CloseFile(f);
  end;
end;

procedure Init;
var

```

```

      i: Integer;
begin
  a[0] := -1;
  b[0] := -1;
  m := 0;
  nilT := @sentinel;
end;

procedure SetLink(parent, child: PNode; InLeft: Boolean);
begin
  child^.P := parent;
  if InLeft then parent^.L := child
  else parent^.R := child;
end;

procedure UpTree(x: PNode);
var
  y, z, beta: PNode;
begin
  y := x^.P;
  z := y^.P;
  if x = y^.L then
    begin
      beta := x^.R;
      SetLink(y, beta, True);
      SetLink(x, y, False);
    end
  else
    begin
      beta := x^.L;
      SetLink(y, beta, False);
      SetLink(x, y, True);
    end;
  SetLink(z, x, z^.L = y);
end;

procedure Splay(x: PNode);
var
  y, z: PNode;
begin
  repeat
    y := x^.P;
    if y = nilT then Break;
    z := y^.P;
    if z <> nilT then
      if (y = z^.L) = (x = y^.L) then UpTree(y)
      else UpTree(x);
    UpTree(x);
  until False;
end;

procedure Split(x: PNode; var LTree, Rtree: PNode);
begin
  Splay(x);
  Rtree := x;
  Ltree := x^.L;
  Rtree^.L := nilT;

```

```

Ltree^.P := nilT;
end;

function Join(Ltree, Rtree: PNode): PNode;
begin
  if Ltree = nilT then Exit(Rtree);
  while Ltree^.R <> nilT do Ltree := Ltree^.R;
  Splay(Ltree);
  SetLink(Ltree, Rtree, False);
  Result := Ltree;
end;

procedure DelTree(tree: PNode);
begin
  if tree = nilT then Exit;
  DelTree(tree^.L);
  DelTree(tree^.R);
  Dispose(tree);
end;

{
Tim trong cay tree, nut chua chi so j co a[j] nho nhat > ka
neu khong thay tra ve nilT
cac a[j] trong cay xep tang dan theo thu tu giua
}

function SearchA(var tree: PNode; ka: Integer): PNode;
var
  j: Integer;
  x: PNode;
begin
  if tree = nilT then Exit(nilT);
  Result := nilT;
  x := tree;
  repeat
    tree := x;
    j := x^.key;
    if a[j] <= ka then x := x^.R //Tim trong nhanh phai
    else
      begin
        Result := x; //Ghi nhan
        x := x^.L; //Tim trong nhanh trai
      end;
  until x = nilT;
  Splay(tree);
end;

{
Tim trong cay tree, nut chua chi so j co b[j] lon nhat < kb
neu khong thay tra ve nilT
cac b[j] trong cay xep giam dan theo thu tu giua
}

function SearchB(var tree: PNode; kb: Integer): PNode;
var
  j: Integer;
  x: PNode;

```

```

begin
  if tree = nilT then Exit(nilT);
  Result := nilT;
  x := tree;
  repeat
    tree := x;
    j := x^.key;
    if b[j] >= kb then x := x^.R //Tim trong nhanh phai
    else
      begin
        Result := x;
        x := x^.L;
      end;
  until x = nilT;
  Splay(tree);
end;

procedure Delete(var tree: PNode; x: PNode);
var
  Ltree, Rtree: PNode;
begin
  Splay(x);
  Ltree := x^.L; Rtree := x^.R;
  Ltree^.P := nilT; Rtree^.P := nilT;
  Dispose(x);
  tree := Join(Ltree, Rtree);
end;

//Chen chi so i vao cay tree, trong cay khong co chi so nao tot hon i
procedure Insert(var tree: PNode; i: Integer);
var
  x, temp, ltree, rtree: PNode;
begin
  //x: Nut chua chi so j co a[j] nho nhat > a[i]
  x := SearchA(tree, a[i]);
  //Tach: ltree chua cac j: a[j] <= a[i], rtree chua cac j: a[j] > a[i]
  if x <> nilT then
    Split(x, ltree, rtree)
  else
    begin
      ltree := tree; rtree := nilT;
    end;
  //Loai bo tat ca cac chi so j khong tot hon i: a[j] <= a[i] & b[j] <= b[i]
  //cac chi so nay deu nam trong cay ltree
  if ltree <> nilT then
    begin
      x := SearchB(ltree, b[i] + 1);
      if x <> nilT then
        begin
          Split(x, ltree, temp);
          DelTree(temp);
        end;
    end;
  New(tree);
  tree^.key := i;
  tree^.P := nilT;
  SetLink(tree, ltree, True);
end;

```

```

SetLink(tree, rtree, False);
end;

function BinarySearch(i: Integer): Integer;
var
  Low, Middle, High: Integer;
  x: PNode;
begin
  Low := 1; High := m;
  while Low <= High do
    begin //Low - 1: Success; High + 1: Fail
      Middle := (Low + High) div 2;
      x := SearchA(tree[Middle], a[i]);
      if (x <> nilT) and (b[x^.key] > b[i]) then Low := Middle + 1
      else High := Middle - 1;
    end;
  Result := High;
end;

procedure Solve;
var
  i: Integer;
  len: Integer;
  x: PNode;
begin
  for i := n downto 0 do
    begin
      len := BinarySearch(i);
      if len = 0 then trace[i] := n + 1
      else
        begin
          x := SearchA(tree[len], a[i]);
          trace[i] := x^.key;
        end;
      Inc(len);
      if len > m then
        begin
          Inc(m);
          tree[m] := nilT;
        end;
      Insert(tree[len], i);
    end;
  for i := 1 to m do
    DelTree(tree[i]);
  Dec(m);
end;

procedure PrintResult;
var
  f: TextFile;
  i: Integer;
begin
  AssignFile(f, OutputFile); Rewrite(f);
  try
    WriteLn(f, m);
    i := trace[0];
    repeat

```

```
    Write(f, i, ' ');
    i := trace[i];
  until i = n + 1;
finally
  CloseFile(f);
end;
end;

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