Chương 1. Cây nhị phân tìm kiếm

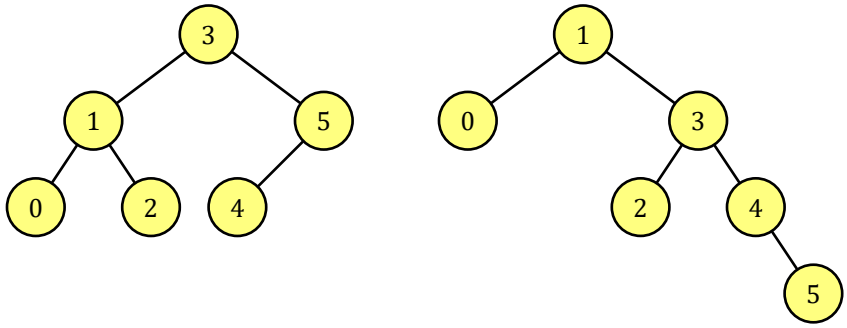
Rất nhiều vấn đề thực tế đòi hỏi phải duy trì một tập hợp có thứ tự các phần tử, kèm theo các thao tác trên tập hợp đó như: Bổ sung phần tử, loại bỏ phần tử, xác định sự tồn tại của một phần tử trong tập hợp, tìm phần tử lớn nhất/nhỏ nhất… Cây nhị phân tìm kiếm là một cấu trúc dữ liệu hiệu quả để cài đặt những thao tác đó.

Các phần tử trong tập hợp được gọi là *khóa* (*keys*), kèm theo một quan hệ thứ tự yếu ngặt "<". Khái niệm về khóa và các loại quan hệ thứ tự sẽ được trình bày rõ hơn trong phần sắp xếp (NEEDREF), trong bài này ta có thể hiểu các quan hệ "≤", "≥", "<", ">", "=", "≠"giống như phép so sánh tương ứng trong toán học.

1.1. Cấu trúc cây nhị phân tìm kiếm

Cây nhị phân tìm kiếm (*binary search trees - BST*) là một dạng cây nhị phân, trong đó mỗi nút chứa một phần tử (khóa). Khóa chứa trong mỗi nút phải thỏa mãn hai điều kiện:

- Lớn hơn mọi khóa trong nhánh con trái
- Nhỏ hơn mọi khóa trong nhánh con phải



Hình 1-1. Cây nhị phân tìm kiếm

Có thể có nhiều cây nhị phân tìm kiếm biểu diễn cùng một bộ khóa. Hình 1-1 là ví dụ về hai cây nhị phân tìm kiếm biểu diễn cùng một bộ khóa (0,1,2,3,4,5).

1.2. Các thao tác trên cây nhị phân tìm kiếm

1.2.1. Cấu trúc nút

BST trong bài này được biểu diễn bằng một cấu trúc các nút và con trỏ liên kết. Mỗi nút trên BST gồm các trường:

- Trường key: Chứa khóa lưu trong nút.
- Trường P: Chứa con trỏ tới nút cha, nếu là nút gốc (không có nút cha) thì trường P được đặt bằng một con trỏ đặc biệt (nil).
- Trường L: Chứa liên kết (con trỏ) tới nút con trái, nếu nút không có nhánh con trái thì trường L được đặt bằng nil.
- Trường R: Chứa liên kết (con trỏ) tới nút con phải, nếu nút không có nhánh con phải thì trường R được đặt bằng nil.

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

```
 1 | struct TNode //Cấu trúc nút của BST
 2 | {
 3 |     TKey key; //Khóa chứa trong nút
 4 |     TNode* P; //Con trỏ tới nút cha
 5 |     TNode* L; //Con trỏ tới nút con trái
 6 |     TNode* R; //Con trỏ tới nút con phải
 7 | };
 8 | using PNode = TNode*;
 9 | PNode root; //Gốc của BST
10 | TNode Sentinel; //Nút lính canh, các trường vô nghĩa
11 | PNode nil = &Sentinel; //Con trỏ nil
```

Ta đồng nhất khái niệm nút với con trỏ tới chính nút đó cho tiện trình bày. Tuy nhiên trong một vài trường hợp vẫn cần định rõ ràng khái niệm nút và con trỏ tới nút, vì vậy ta định nghĩa thêm kiểu `PNode` là kiểu con trỏ tới một nút. Biến `root` là con trỏ tới nút gốc của BST. Cây được quản lý qua nút gốc, vì vậy trong các thao tác trên BST, ta coi nút gốc là đại diện cho cây và có thể đồng nhất khái niệm cây/nhánh với khái niệm nút gốc của cây/nhánh đó.

Thông thường người ta dùng con trỏ `nullptr` để gán cho những liên kết không tồn tại. Tuy vậy việc sử dụng con trỏ `nullptr` sẽ dẫn đến một số phiền toái: Muốn truy cập các trường của nút `x` sẽ phải kiểm tra `x ≠ nullptr` trước, kéo theo rất nhiều lệnh `if`… phải thêm vào trong các thao tác xử lý cây.

Ở đây ta dùng con trỏ `nil` để gán cho những liên kết không tồn tại. Khác với con trỏ `nullptr`, con trỏ `nil` chứa địa chỉ của một nút có thực: `sentinel`, quy ước rằng nút này cũng như các trường của nó là vô nghĩa. Mục đích của con trỏ `nil` cũng như nút lính

canh `sentinel` là để ta có thể thoải mái viết `nil->R`, `nil->L`, `nil->R`, mà không bị gặp lỗi như đối với con trỏ `nullptr`.

1.2.2. Khởi tạo cây rỗng

Ta quy ước nếu cây rỗng thì con trỏ tới nút gốc ($root$) được gán bằng 1, vì vậy phép khởi tạo cây rỗng đơn giản chỉ là:

```
1 | void MakeEmpty() //Khởi tạo cây rỗng
2 | {
3 |     root = nil; //Cây rỗng có gốc bằng nil
4 | }
```

1.2.3. Duyệt cây theo thứ tự giữa

Phép duyệt cây theo thứ tự giữa được thực hiện bằng hàm `Visit(root)`. Hàm `Visit(x)` là một hàm đệ quy có nhiệm vụ duyệt nhánh cây gốc `x` theo trình tự:

- Duyệt con trái của `x` (`Visit(x->L)`)
- Đưa ra khóa trong nút x (`x->key`)
- Duyệt con phải của `x` (`Visit(x->R)`)

```
1 | void Visit(PNode x) //Duyệt nhánh cây gốc x theo thứ tự giữa
2 | {
3 |     if (x == nil) return; //Nhánh gốc x không tồn tại, thoát
4 |     Visit(x->L); //Đệ quy duyệt nhánh con trái theo thứ tự giữa
5 |     Output ← x->key; //Đưa ra khóa trong nút x
6 |     Visit(x->R); //Đệ quy duyệt nhánh con phải theo thứ tự giữa
7 | }
```

Bổ đề 1-1

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

Chứng minh

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

1.2.4. Tìm khóa lớn nhất và nhỏ nhất

Theo cấu trúc của BST, khóa nhỏ nhất của BST nằm ở nút cực trái: Bắt đầu từ gốc ta đi sang con trái chừng nào vẫn còn đi được, quá trình dừng lại ở nút cực trái là nút chứa khóa nhỏ nhất.

```
1 | //Tìm nút cực trái trong cây gốc x, giả thiết x ≠ nil
2 | PNode Minimum(PNode x)
3 | {
4 |     while (x->L ≠ nil) //Chừng nào x có con trái
5 |         x = x->L; //Đi sang con trái
6 |     return x; //Khi không đi được nữa, x là nút cực trái
7 | }
```

Vấn đề tương tự với việc tìm khóa lớn nhất: Khóa nằm ở nút cực phải của BST.

```
1 | //Tìm nút cực phải trong cây gốc x, giả thiết x ≠ nil
2 | PNode Maximum(PNode x)
3 | {
4 |     while (x->R ≠ nil) //Chừng nào x có con phải
5 |         x = x->R; //Đi sang con phải
6 |     return x; //Khi không đi được nữa, x là nút cực phải
7 | }
```

1.2.5. Tìm nút liền trước và nút liền sau

Đôi khi chúng ta phải tìm nút đứng liền trước và liền sau của một nút x nếu duyệt cây BST theo thứ tự giữa. Hàm Predecessor(x) trả về nút đứng liền trước nút x được tiến hành như sau:

- Nếu x có con trái thì trả về nút cực phải của nhánh con trái.
- Nếu x không có con trái thì từ x, ta đi dần lên phía gốc cây cho tới khi gặp một nút chứa x trong nhánh con phải thì dừng lại và trả về nút đó.

```
1 | //Tìm nút liền trước nút x theo thứ tự giữa, trả về nil nếu không tìm thấy
2 | PNode Predecessor(PNode x)
3 | {
4 |     if (x->L ≠ nil) //x có con trái
5 |         return Maximum(x->L); //Trả về nút cực phải trong nhánh con trái
6 |     while (x->P ≠ nil && x->P->R ≠ x) //x không phải gốc và x không là con phải
7 |         x = x->P; //Đi lên cha x
8 |     return x->P; //Trả về cha x, mặc nhiên trả về nil nếu x đã là gốc
9 | }
```

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

```
1 | //Tìm nút liền sau nút x theo thứ tự giữa, trả về nil nếu không tìm thấy
2 | PNode Successor(PNode x)
3 | {
4 |     if (x->R ≠ nil) //x có con phải
5 |         return Minimum(x->R); //Trả về nút cực trái trong nhánh con phải
6 |     while (x->P ≠ nil && x->P->L ≠ x) //x không phải gốc và x không là con trái
7 |         x = x->P; //Đi lên cha x
8 |     return x->P; //Trả về cha x, mặc nhiên trả về nil nếu x đã là gốc
9 | }
```

1.2.6. Tìm kiếm

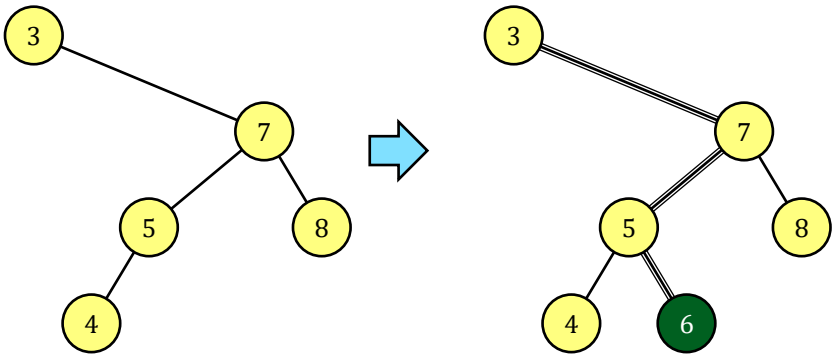
Phép tìm kiếm nhận vào một khóa k. Nếu khóa k có trong BST thì trả về một nút chứa khóa k, nếu không trả về nil.

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

```
1 | //Tìm nút chứa khóa k, trả về nil nếu không tìm thấy
2 | PNode Find(TKey k)
3 | {
4 |     PNode x = root; //Bắt đầu từ gốc
5 |     while (x ≠ nil && x->key ≠ k) //Chừng nào x tồn tại và chứa khóa ≠ k
6 |         x = k < x->key ? x->L : x->R; //Đi xuống con của x
7 |     return x;
8 | }
```

1.2.7. Chèn

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



Hình 1-2. BST trước và sau khi chèn khóa 6

Để các đoạn code sau này được ngắn gọn, ta viết hàm:

```
SetLink(parent, child, InRight);
```

Có nhiệm vụ cho nút `child` làm con của nút `parent` (làm con trái hay con phải tùy theo `InRight` là `false` hay `true`)

```
1 | void SetLink(PNode parent, PNode child, bool InRight)
2 | {
3 |     child->P = parent;
4 |     if (InRight) parent->R = child;
5 |     else parent->L = child;
6 | }
```

(Có thể thấy công dụng của việc dùng con trỏ `nil` thay cho `nullptr`: Ta thoải mái truy cập `child->P`, `parent->R`, `parent->L`, mà không lo ngại sinh lỗi).

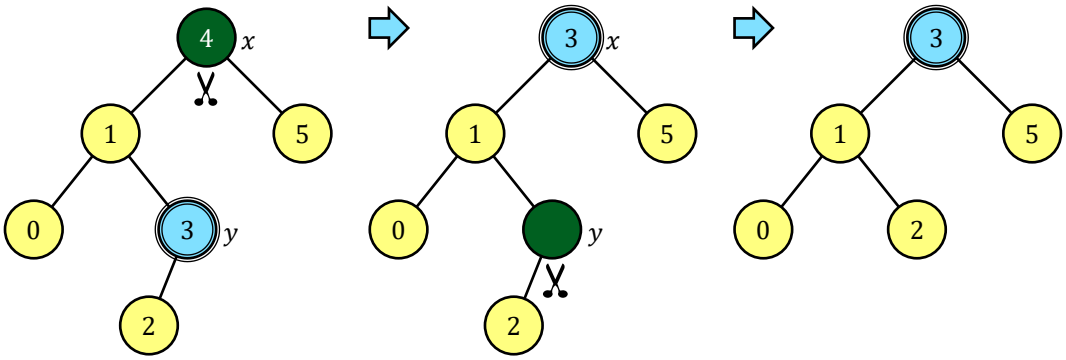
Hàm chèn khóa *k* vào BST có thể viết như sau:

```
 1 | //Chèn khóa k vào BST
 2 | void Insert(TKey k)
 3 | {
 4 |     PNode y = nil;
 5 |     PNode x = root; //Bắt đầu từ gốc
 6 |     while (x ≠ nil) //Chưa chạm tới nút rỗng
 7 |     {
 8 |         if (k == x->key) return; //Khóa k đã có trên BST
 9 |         y = x; //Giữ lại vị trí x cũ
10 |         x = k < x->key ? x->L : x->R; //x chuyển xuống con trái hoặc con phải
11 |     }
12 |     //Tạo nút x mới chứa khóa k
13 |     x = new TNode{.key = k, .P = nil, .L = nil, .R = nil};
14 |     SetLink(y, x, k > y->key);
15 |     if (root == nil) //Nếu gốc đang là nil
16 |         root = x; //thì cập nhật gốc mới là x
17 | }
```

1.2.8. Xóa

Việc xóa một nút *x* trong BST có thể thực hiện bằng cách xóa đi khóa chứa trong nút *x*. Phép xóa được thực hiện như sau:

- Nếu x có ít hơn 2 con, lấy nút con (nếu có) của x lên thay cho x và hủy nút x.
- Nếu x có hai con, xác định nút y là nút cực phải của nhánh con trái, đưa khóa chứa trong nút y lên nút x rồi xóa nút y. Chú ý rằng nút y chắc chắn không có nhánh con phải, việc xóa quy về trường hợp trên (Hình 1-4)



Hình 1-3. Xóa nút khỏi BST

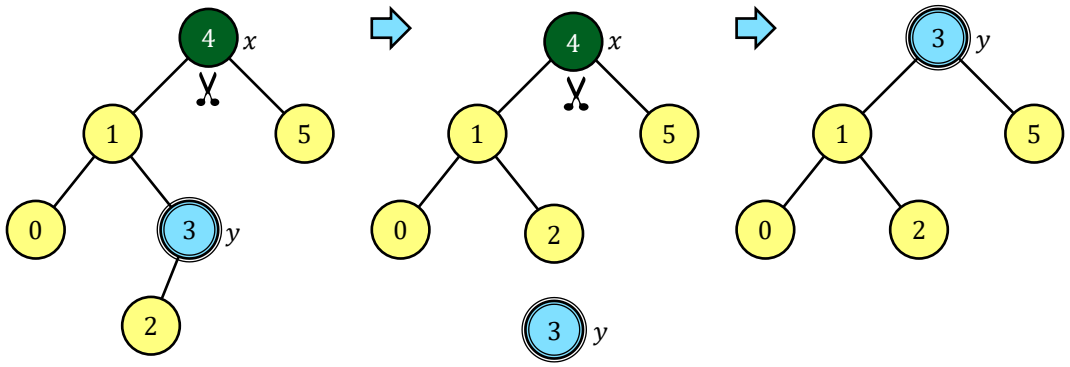
```
1  | //Xóa nút x khỏi BST
2  | void Erase(PNode x)
3  | {
4  |     PNode y;
5  |     if (x->L ≠ nil && x->R ≠ nil) //x có 2 con
6  |     {
7  |         y = Maximum(x->L); //y = nút cực phải của nhánh con trái
8  |         x->key = y->key; //Đưa khóa trong y sang x
9  |         x = y; //Quy về xóa nút x không có con phải
10 |     }
11 |     //x có ít hơn 2 nhánh con, cắt x khỏi cây
12 |     y = x->L != nil ? x->L : x->R; //y = nút con của x (nếu có)
13 |     SetLink(x->P, y, x->P->R == x); //Cho y làm con của x->P thay cho x
14 |     if (x == root) //x đang là gốc
15 |         root = y;  //thì y sẽ là gốc mới
16 |     delete x; //Hủy nút x
17 | }
```

Mặc dù đây là phương pháp được trình bày trong hầu hết các sách thuật toán để xóa nút trên BST. Cách làm này thực ra rất nguy hiểm và không bao giờ nên sử dụng. Hai nhược điểm lớn nhất của nó là:

- Thuật toán cần di chuyển dữ liệu khóa giữa các nút, khi khóa thuộc kiểu dữ liệu có cấu trúc phức tạp, việc di chuyển dữ liệu khóa khá tốn thời gian và kéo theo nhiều rắc rối.
- Thuật toán có nhiệm vụ xóa nút, nhưng trên thực tế nó chỉ loại bỏ khóa trong nút đó ra khỏi BST, nút bị xóa có thể là một nút khác. Nếu con trỏ tới một nút đang lưu lại ở đâu đó trong chương trình cho nhiệm vụ khác, khi đó sẽ có tình trạng con trỏ tới nút "đã bị xóa" nhưng thực ra nút đó vẫn tồn tại, trong khi con trỏ tới nút không hề bị xóa nhưng thực ra nút đó không còn tồn tại nữa. Những lỗi truy cập bộ nhớ trái phép có cơ hội phát sinh và không dễ để kiểm soát.

Vì vậy để xóa nút x một cách chính quy hơn, ta sẽ sử dụng cách cài đặt khác:

- Nếu x có ít hơn 2 con, ta lấy nút con (nếu có) của x lên thay cho x và hủy nút x.
- Nếu x có cả hai con, xác định nút y là nút cực phải của nhánh con trái, y chắc chắn không có nhánh con phải, ta cắt rời y khỏi cây bằng cách đưa nhánh con trái của nó (nếu có) vào thế chỗ y. Cuối cùng là cho nút y thế chỗ nút x trên cây và xóa nút x. Tất cả các thao tác được thực hiện trên con trỏ liên kết mà không động chạm vào khóa trong nút (Hình 1-4).



Hình 1-4. Xóa nút khỏi BST (cách an toàn)

```
//Xóa nút x khỏi BST
void Erase(PNode x)
{
    PNode y;
    if (x->L == nil || x->R == nil) //x có ít hơn 2 con
    {
        y = x->L != nil ? x->L : x->R; //y = nút con của x nếu có
        SetLink(x->P, y, x->P->R == x); //Cho y thế chỗ x làm con của x->P
    }
    else //x có 2 con
    {
        y = Maximum(x->L); //y = nút cực phải của nhánh con trái
        SetLink(y->P, y->L, y->P->R == y); //Cho con trái y thế chỗ y, cắt y khỏi cây
        //Cho y thế chỗ x
        SetLink(y, x->L, false); //y nhận con trái của x
        SetLink(y, x->R, true);  //y nhận con phải của x
        SetLink(x->P, y, x->P->R == x); //y nhận cha của x
    }
    if (x == root) //Gốc x bị xóa
        root = y; //Gốc mới là y thay thế
    delete x; //x đã tách rời khỏi cây, hủy nút x
}
```

1.3. Một số chú ý khi cài đặt cây nhị phân tìm kiếm

1.3.1. Hiệu lực của các thao tác trên cây nhị phân tìm kiếm

Không khó để chứng minh được rằng các thao tác Minimum, Maximum, Predcessor, Successor, Search, Insert, Delete, đều có thời gian thực hiện $O(h)$ với h là chiều cao của cây nhị phân tìm kiếm. Tuy nhiên đó là nói một thao tác riêng lẻ, Bài tập 1-1 yêu cầu chứng minh rằng nếu bắt đầu từ nút cực trái của cây và cứ đi sang nút liền sau cho tới khi duyệt hết các nút thì chỉ mất thời gian $O(n)$ với n là số nút trên cây, độ phức tạp tương đương với phép duyệt cây theo thứ tự giữa.

Nếu như xác suất tìm kiếm được phân phối đều trên tập gồm n giá trị khóa thì cấu trúc BST tốt nhất là cấu trúc cây nhị phân gần hoàn chỉnh (có chiều cao thấp nhất: $h = \lfloor \lg n \rfloor$) còn cấu trúc BST tồi nhất để biểu diễn là cấu trúc cây nhị phân suy biến (có chiều cao $h = n - 1$).

Định lý 1-2

Cây nhị phân tìm kiếm xây dựng bằng cách chèn n khóa phân biệt vào cây theo một thứ tự ngẫu nhiên có chiều cao trung bình $O(\log n)$

Chứng minh ★

Ta có thể coi BST được xây dựng bằng cách chèn các khóa số nguyên $\{1,2,\dots,n\}$ vào theo một thứ tự ngẫu nhiên. Gọi X_n là biến ngẫu nhiên biểu thị chiều cao của BST, ta cần chứng minh giá trị kỳ vọng $E[X_n] = O(\log n)$. Định nghĩa $Y_n = 2^{X_n}$. Ý tưởng chính là ta sẽ chứng minh kỳ vọng của Y_n là cấp O lớn của một hàm đa thức và từ đó chứng minh kỳ vọng của X_n bằng $O(\log n)$.

Khi $n = 1$, ta có chiều cao của BST luôn bằng 0, tức là $X_1 = 0$ và $Y_1 = 1$, để tiện trình bày, ta quy ước rằng $Y_0 = 0$.

Khi xây dựng BST, đầu tiên ta chọn ngẫu nhiên một khóa $i \in \{1,2, \dots, n\}$ và chèn vào cây, khóa này sẽ luôn nằm ở gốc cây. Cây con trái sẽ là một BST chứa $i - 1$ khóa $\{1,2, \dots, i-1\}$ và cây con phải sẽ là một BST chứa $n - i$ khóa $\{i+1, i+2, \dots, n\}$. Vậy thì chiều cao của cây sẽ là:

$$X_n = 1 + \max(X_{i-1}, X_{n-i})$$

Từ đó:

$$Y_n = 2.\max(Y_{i-1}, Y_{n-i})$$

Xác suất để chọn ngẫu nhiên một khóa $i \in \{1,2, \dots, n\}$ là $1/n$, vì thế giá trị kỳ vọng của Y_n có thể viết thành:

$$\begin{aligned} E[Y_n] &= \frac{1}{n}\sum_{i=1}^{n} E[2.\max(Y_{i-1}, Y_{n-i})] \\ &= \frac{2}{n}\sum_{i=1}^{n} E[\max(Y_{i-1}, Y_{n-i})] \\ &\leq \frac{2}{n}\sum_{i=1}^{n} (E[Y_{i-1}] + E[Y_{n-i}]) \end{aligned} \tag{1.1}$$

Vì mỗi hạng tử $E[Y_0], E[Y_1], \dots, E[Y_{n-1}]$ xuất hiện 2 lần trong tổng cuối cùng của công thức (1.1) nên suy ra:

$$E[Y_n] \leq \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i] \tag{1.2}$$

Ta sẽ chứng minh rằng $E[Y_n] \leq \frac{1}{4}C_{n+3}^3$ bằng quy nạp. Ở trường hợp cơ sở:

Khi $n = 0, E[Y_0] = Y_0 = 0 < \frac{1}{4} = \frac{1}{4}C_3^3$

Khi $n = 1, E[Y_1] = Y_1 = 1 = \frac{1}{4}C_4^3$

Khi $n > 1$:

$$
\begin{aligned}
E[Y_n] &\le \frac{4}{n}\sum_{i=0}^{n-1} E[Y_i] \le \frac{4}{n}\sum_{i=0}^{n-1}\frac{1}{4}C_{i+3}^{3} \text{ (theo giả thiết quy nạp)} \\
&\le \frac{1}{n}\sum_{i=0}^{n-1} C_{i+3}^{3} \\
&= \frac{1}{n}C_{n+3}^{4}
\end{aligned} \tag{1.3}
$$

Đẳng thức cuối cùng có thể chứng minh thuần túy bằng đại số, nhưng có thể giải thích dễ hiểu như sau: Xét C_{n+3}^{4} số cách chọn 4 số nguyên $a < b < c < d$ từ tập $\{1,2,\dots,n+3\}$. Dĩ nhiên trong mọi cách chọn thì $4 \le d \le n+3$. Với mỗi giá trị d từ 4 tới $n+3$, số cách chọn ứng với mỗi giá trị d đó tương đương với số cách chọn 3 số nguyên $a < b < c$ từ tập $\{1,2,\dots,d-1\}$, tức là bằng C_{d-1}^{3}. Suy ra: $C_{n+3}^{4} = C_3^3 + C_4^3 + \cdots + C_{n+2}^{3} = \sum_{i=0}^{n-1} C_{i+3}^{3}$

Công thức (1.3) viết tiếp thành:

$$
E[Y_n] \le \frac{1}{n}C_{n+3}^{4} = \frac{1}{n}\frac{(n+3)!}{4!\,(n-1)!} = \frac{1}{4}\frac{(n+3)!}{3!\,n!} = \frac{1}{4}C_{n+3}^{3} \tag{1.4}
$$

Đến đây ta có $E[Y_n]$ là một hàm đa thức, cụ thể là $\frac{n^3+6n^2+11n+6}{24}$. Mặt khác hàm 2^x là hàm lồi, bất đẳng thức Jensen cho ta điều phải chứng minh:

$$
2^{E[X_n]} \le E[2^{X_n}] = E[Y_n] \le \frac{n^3+6n^2+11n+6}{24}
$$

Lấy $\log_2(.)$ cả hai vế, ta có $E[X_n] = \mathrm{O}(\log n)$.

Hệ quả

Cây nhị phân tìm kiếm xây dựng bằng cách chèn n khóa phân biệt vào cây theo một thứ tự ngẫu nhiên có độ sâu trung bình của các nút là $\mathrm{O}(\log n)$

Cấu trúc BST bị phụ thuộc vào thứ tự chèn/xóa, điều đó làm cho BST rất ít khi được sử dụng ngoài mục đích học tập vì không có gì đảm bảo cho cây BST không bị suy biến. Ở trong các ứng dụng thực tế, BST luôn được sử dụng kèm với các kỹ thuật cân bằng cây mà ta sẽ trình bày trong bài sau.

1.3.2. Việc sử con trỏ tới nút cha

Trong các thao tác đã trình bày mỗi nút trên BST có trường *P* là con trỏ tới nút cha. Có hai điểm cần lưu ý:

- Đối với những ứng dụng cụ thể mà ta không cần dùng đến trường *P*, có thể loại bỏ trường này khỏi cấu trúc nút cho tiết kiệm bộ nhớ. Việc loại bỏ trường *P* khỏi cấu trúc nút thường được áp dụng khi chúng ta viết các thao tác trên BST bằng đệ quy.
- Việc truy cập nút thường diễn ra sau một quá trình di chuyển từ nút gốc tới nút đang xét. Vì vậy khi thay đổi cấu trúc BST, chúng ta không nhất thiết phải cập nhật các con trỏ tới nút cha. Mỗi khi có phép di chuyển từ nút gốc tới *x*, ta có thể khéo léo lồng vào quá trình di chuyển việc cập nhật trường *P* của tất cả các nút dọc đường đi để sử dụng trong trường hợp chúng ta cần đi ngược lên gốc. Kỹ thuật này cần được **sử dụng cẩn thận** trong lập trình vì nó có thể gây khó khăn cho việc kiểm soát.

1.3.3. Việc sử dụng giải thuật đệ quy

Hầu hết các thao tác căn bản của BST đều có thể dễ dàng cài đặt bằng đệ quy. Ví dụ phép tìm kiếm khóa *k* có thể thực hiện bằng hàm đệ quy `RecursiveFind(root, k)` như dưới đây:

```
1 | PNode RecursiveFind(PNode x, TKey k) //Tìm nút chứa khóa k trong nhánh cây gốc x
2 | {
3 |     if (x == nil || x->key == k) //x không tồn tại hoặc chứa khóa bằng k
4 |         return x; //Trả về chính x
5 |     return k < x->key ? //Tìm tiếp trong nhánh con bằng đệ quy
6 |         RecursiveFind(x->L, k) : RecursiveFind(x->R, k);
7 | }
```

Phép chèn khóa `k` vào BST cũng có thể viết bằng một hàm đệ quy `RecursiveInsert(root, k)` như sau:

```
//Chèn khóa k vào BST bằng hàm RecursiveInsert(root, k)
void RecursiveInsert(PNode& x, TKey k)
{
    if (x == nil) //nhánh x rỗng, cho x thành nút mới chứa khóa k
        x = new TNode{.key = k, .P = nil, .L = nil, .R = nil};
    else
    {
        if (k < x->key) //Chèn k vào nhánh con trái của x
        {
            RecursiveInsert(x->L, k);
            x->L->P = x;
        }
        if (k > x->key) //Chèn k vào nhánh con phải của x
        {
            RecursiveInsert(x->R, k);
            x->R->P = x;
        }
    }
}
```

Cách viết đệ quy thường ngắn gọn và dễ hiểu nhưng chương trình chậm hơn cách viết bằng giải thuật lặp. Vì vậy chỉ nên sử dụng đệ quy với những thao tác thực sự khó khăn với giải thuật lặp.

1.3.4. Vấn đề các khóa bằng nhau

BST hoàn toàn có thể sửa đổi để quản lý tập các khóa trong đó có những khóa bằng nhau. Chỉ cần sửa đổi lại phép chèn khóa k vào BST: Bắt đầu từ gốc cây, nếu k nhỏ hơn khóa ở gốc, ta lặp lại việc chèn trên cây con trái, nếu không ta lặp lại việc chèn trên cây con phải…

```
//Chèn khóa k vào BST
void Insert(TKey k)
{
    PNode y = nil;
    PNode x = root; //Bắt đầu từ gốc
    while (x ≠ nil) //Chưa chạm tới nút rỗng
    {
        y = x; //Giữ lại vị trí x cũ
        x = k < x->key ? x->L : x->R; //x chuyển xuống con trái hoặc con phải
    }
    //Tạo nút x mới chứa khóa k
    x = new TNode{.key = k, .P = nil, .L = nil, .R = nil};
    SetLink(y, x, k > y->key); //cho x làm con y
    if (root == nil) //Nếu gốc đang là nil
        root = x; //thì cập nhật gốc mới là x
}
```

1.4. Lớp mẫu set và map trong C++ STL

Cũng tương tự như các lớp mẫu khác của C++STL, phần này chỉ trình bày vắn tắt công dụng của các lớp mẫu và cách sử dụng chúng trong chương trình. Người đọc cần tham khảo thêm các tài liệu chuẩn về C++ về các lớp mẫu này.

1.4.1. Lớp mẫu std::set

Lớp mẫu `std::set<T>` là tập hợp các phần tử kiểu `T`, trên các phần tử kiểu `T` phải có quan hệ thứ tự yếu ngặt "<". Trong trường hợp quan hệ thứ tự yếu chưa được định nghĩa hoặc muốn định nghĩa lại quan hệ thứ tự yếu ngặt trên `T`, cần khai báo `std::set<T>` kèm `functor` có toán tử `()` định nghĩa quan hệ thứ tự yếu ngặt.

Quan hệ thứ tự yếu ngặt quyết định thứ tự các phần tử trong `std::set<T>`, phần tử "đứng trước" luôn "nhỏ hơn" phần tử đứng sau. Lớp mẫu `std::set<T>` không cho phép hai phần tử "bằng nhau" cùng có mặt trong tập hợp.

Lớp mẫu `std::set<T>` còn cung cấp các `iterator` cho phép duyệt các phần tử của tập hợp theo thứ tự từ nhỏ tới lớn hoặc từ lớn đến nhỏ. Đây là loại `iterator` cho phép chạy hai chiều (`bidirectional`), hỗ trợ các phép toán `++` và `--` để dịch chuyển tiến/lùi. Loại `iterator` này không cho phép truy cập ngẫu nhiên.

Một số phương thức cơ bản được `std::set<T>` cung cấp có thể liệt kê ra là:

- `bool empty()`: Kiểm tra tập hợp có rỗng không
- `size_type size()`: Trả về số phần tử trong tập hợp
- `void clear()`: Làm rỗng tập hợp
- `const_iterator begin()`: Trả về `iterator` tới phần tử đầu tiên (nhỏ nhất) của tập hợp
- `const_iterator end()`: Trả về `iterator` tới phần tử đứng sau tập hợp (có thể coi như lớp mẫu duy trì một phần tử đứng sau phần tử lớn nhất với địa chỉ được cho bởi `*this->end()`)
- `const_iterator find(T key)`: Trả về `iterator` tới phần tử bằng `key`, nếu không tìm thấy trả về `*this->end()`.
- `const_iterator lower_bound(T key)`: Trả về `iterator` tới phần tử nhỏ nhất lớn hơn hay bằng `key`, nếu không tìm thấy trả về `*this->end()`.
- `const_iterator upper_bound(T key)`: Trả về `iterator` tới phần tử nhỏ nhất lớn hơn `key`, nếu không tìm thấy trả về `*this->end()`.
- `pair<iterator,bool> insert(T key)`: Chèn `key` vào tập hợp.
  - Nếu `key` đã có trong tập, trường `first` của kết quả là `iterator` tới phần tử đã có trong tập và trường `second` của kết quả bằng `false` báo hiệu không chèn được.
  - Nếu `key` chưa có trong tập, nó sẽ được chèn vào tập, trường `first` của kết quả là `iterator` tới phần tử vừa chèn và trường `second` của kết quả bằng `true` báo hiệu chèn thành công.
- `iterator erase(iterator pos)`: Xóa phần tử ở vị trí chỉ ra bởi `pos`, hàm trả về `iterator` tới phần tử kế tiếp (theo thứ tự). Nếu phần tử bị xóa là phần tử lớn nhất, trả về `*this->end()`.

- `size_type erase(T key)`: Xóa phần tử `key` khỏi tập hợp, trả về số phần tử xóa được (trả về 0 nếu phần tử `key` không tồn tại).

Về bản chất, `std::set<T>` được cài đặt bằng một dạng cây nhị phân tìm kiếm tự cân bằng. Nó dùng các kỹ thuật điều chỉnh cấu trúc để luôn giữ cho độ cao của BST gồm n nút luôn là một đại lượng $O(\log n)$. Điều này đảm bảo cho tất cả các thao tác tìm kiếm, chèn, xóa có độ phức tạp $O(\log n)$ (ta sẽ tìm hiểu các cơ chế tự cân bằng trong bài sau).

1.4.2. Lớp mẫu std::multiset

Lớp mẫu `std::multiset<T>` cũng tương tự như `std::set<T>`, tuy nhiên nó cho phép các phần tử của tập hợp có thể bằng nhau (phần tử đứng trước "nhỏ hơn" hoặc "bằng" phần tử đứng sau). Các phương thức của `std::multiset<T>` cũng tương tự như `std::set<T>` chỉ có một vài sự thay đổi nhỏ:

- `iterator insert(T key)`: Chèn `key` vào tập hợp. Khóa `key` sẽ được chèn vào trước phần tử đầu tiên trong tập hợp lớn hơn nó và sau phần tử cuối cùng trong tập hợp nhỏ hơn hay bằng nó. Phép chèn trên `std::multiset` luôn thành công, trả về `iterator` tới phần tử mới chèn.
- `size_type erase(T key)`: Xóa tất cả các phần tử bằng `key` khỏi tập hợp, trả về số phần tử xóa được.
- `size_type count(T key)`: Trả về số phần tử bằng `key` trong tập hợp. Độ phức tạp bằng $O(\log n + m)$ trong đó n là số phần tử trong tập hợp và m là số phần tử bằng key. Thực ra về mặt thuật toán, thao tác này có thể xử lý trong thời gian $O(\log n)$, nhưng đáng tiếc là chuẩn C++ chỉ quy định như vậy.

1.4.3. Lớp mẫu std::map

Lớp mẫu `std::map<TKey, TValue>` là tập hợp các phần tử, mỗi phần tử là một cặp `(key, value)` trong đó `key` thuộc kiểu `TKey` và `value` thuộc kiểu `TValue`. Ở đây `key` được gọi là "khóa" còn `value` được gọi là "giá trị". Mỗi cặp `(key, value)` cho biết khóa `key` được "ánh xạ" thành `value`.

Trên kiểu `TKey` phải có quan hệ thứ tự yếu ngặt "<". Trong trường hợp quan hệ thứ tự yếu ngặt chưa được định nghĩa hoặc muốn định nghĩa lại quan hệ thứ tự yếu ngặt trên `TKey`, cần khai báo `std::map<TKey, TValue>` kèm `functor` có toán tử `()` định nghĩa quan hệ thứ tự yếu ngặt trên kiểu `TKey`.

Quan hệ thứ tự yếu ngặt trên `TKey` quy định luôn quan hệ thứ tự yếu ngặt trên các cặp `(key, value)`:

$$(key_1, value_1) < (key_2, value_2) \Leftrightarrow key_1 < key_2$$

nên có thể coi như `std::map<TKey, TValue>` tổ chức dữ liệu của nó giống như lớp mẫu `std::set<pair<TKey, TValue>>` với quan hệ thứ tự yếu trên kiểu `pair<TKey, TValue>`

được dẫn xuất từ quan hệ thứ tự yếu trên `TKey`. Đôi khi ta cũng dùng tên "tập hợp" cho kiểu dữ liệu `std::map`.

Lớp mẫu `std::map<TKey, TValue>` cũng cung cấp các `iterator` cho phép duyệt các phần tử theo thứ tự từ nhỏ tới lớn hoặc từ lớn đến nhỏ. Đây là loại `iterator` cho phép chạy hai chiều (`bidirectional`), hỗ trợ các phép toán `++` và `--` để dịch chuyển tiến/lùi. Lưu ý rằng khi dùng `iterator` để lấy phần tử từ `std::map<TKey, TValue>`, phần tử này có kiểu `pair<TKey, TValue>` (trường `.first` kiểu `TKey` và trường `.second` kiểu `TValue`).

Một số phương thức cơ bản được `std::map<TKey, TValue>` cung cấp có thể liệt kê ra là:

- `bool empty()`: Kiểm tra tập hợp có rỗng không (có cặp `(key, value)` nào chứa trong hay không)
- `size_type size()`: Trả về số phần tử trong tập hợp
- `void clear()`: Làm rỗng tập hợp
- `const_iterator begin()`: Trả về `iterator` tới phần tử đầu tiên (nhỏ nhất) của tập hợp
- `const_iterator end()`: Trả về `iterator` tới phần tử đứng sau tập hợp (có thể coi như lớp mẫu duy trì một phần tử đứng sau phần tử lớn nhất với địa chỉ được cho bởi `*this->end()`)
- `const_iterator find(TKey key)`: Trả về `iterator` tới phần tử có khóa bằng `key`, nếu không tìm thấy trả về `*this->end()`.
- `const_iterator lower_bound(TKey key)`: Trả về `iterator` tới phần tử có khóa nhỏ nhất lớn hơn hay bằng `key`, nếu không tìm thấy trả về `*this->end()`.
- `const_iterator upper_bound(TKey key)`: Trả về `iterator` tới phần tử nhỏ nhất có khóa lớn hơn `key`, nếu không tìm thấy trả về `*this->end()`.
- `pair<iterator,bool> insert(pair<TKey, TValue> x)`: Chèn `x` vào tập hợp.
  - Nếu trong tập đã có phần tử mang khóa bằng `x.first`, trường `first` của kết quả là `iterator` tới phần tử đã có trong tập và trường `second` của kết quả bằng `false` báo hiệu không chèn được.
  - Nếu trong tập chưa có phần tử mang khóa bằng `x.first`, `x` sẽ được chèn vào tập, trường `first` của kết quả là `iterator` tới phần tử vừa chèn và trường `second` của kết quả bằng `true` báo hiệu chèn thành công.
- `iterator erase(iterator pos)`: Xóa phần tử ở vị trí chỉ ra bởi `pos`, hàm trả về `iterator` tới phần tử kế tiếp (theo thứ tự). Nếu phần tử bị xóa là phần tử mang khóa lớn nhất (đứng sau cùng theo thứ tự), trả về `*this->end()`.
- `size_type erase(TKey key)`: Xóa phần tử mang khóa `key` khỏi tập hợp, trả về số phần tử xóa được (trả về 0 nếu phần tử `key` không tồn tại).

Lớp mẫu `std::map<TKey, TValue>` cũng thường được cài đặt bằng một dạng cây nhị phân tìm kiếm tự cân bằng. Chuẩn C++ đảm bảo rằng tất cả các thao tác tìm kiếm, chèn, xóa có độ phức tạp $O(\log n)$.

Mỗi cặp `(key, value)` có thể coi như một phép cho tương ứng mỗi phần tử `key ∈ TKey` với một và chỉ một phần tử `value ∈ TValue`, tên gọi "map" chính là "ánh xạ". Lớp mẫu `std::map<TKey, TValue>` cung cấp toán tử `[]` để truy cập giá trị được ánh xạ từ khóa tương ứng:

`TValue& operator[](key)`: Nếu `f` là đối tượng kiểu `std::map<TKey, TValue>` thì `f[key]` trả về tham chiếu tới trường `value` của phần tử `(key, value)` thuộc `f`. Cơ chế thực hiện như sau:

- Nếu trong `f` chưa có phần tử mang khóa `key`, một cặp `(key, value)` được chèn vào `f` với trường `value` được khởi tạo bằng `TValue()`, toán tử trả về tham chiếu tới `value` của cặp `(key, value)` vừa chèn vào.
  - Nếu `TValue` là một kiểu dữ liệu đơn giản (*Plain Old Datatype – POD*), `value` được khởi tạo bằng 0
  - Nếu `TValue` là một class, `value` được khởi tạo bằng `constructor` mặc định (thực ra điều này không hoàn toàn đúng từ C++11 trở đi, xem thêm về cơ chế *value-initialized* của C++ cho rõ hơn).
- Nếu trong `f` đã có phần tử mang khóa `key`, toán tử trả về tham chiếu tới `value` của cặp `(key, value)` sẵn có trong `f`.

1.4.4. Lớp mẫu std::multimap

Lớp mẫu `std::multimap<TKey, TValue>` cũng tương tự như `std::map<TKey, TValue>`, tuy nhiên nó cho phép các phần tử của tập hợp có thể mang khóa bằng nhau (khóa của phần tử đứng trước "nhỏ hơn" hoặc "bằng" khóa của phần tử đứng sau). Các phương thức của `std::multiset<TKey, TValue>` cũng tương tự như `std::set<TKey, TValue>` chỉ có một vài sự thay đổi nhỏ:

- `iterator insert(pair<TKey, TValue> x)`: Chèn cặp `x` vào tập hợp. Phần tử `x` sẽ được chèn vào trước phần tử đầu tiên trong tập hợp mang khóa lớn hơn `x.first` và sau phần tử cuối cùng trong tập hợp mang khóa nhỏ hơn hay bằng `x.first`. Phép chèn trên `std::multimap` luôn thành công, trả về `iterator` tới phần tử mới chèn.
- `size_type erase(TKey key)`: Xóa tất cả các phần tử có khóa bằng `key` khỏi tập hợp, trả về số phần tử xóa được.

- `size_type count(TKey key)`: Trả về số phần tử có khóa bằng `key` trong tập hợp. Độ phức tạp bằng $O(\log n + m)$ trong đó n là số phần tử trong tập hợp và m là số phần tử bằng key. Thực ra về mặt thuật toán, thao tác này có thể xử lý trong thời gian $O(\log n)$, nhưng đáng tiếc là chuẩn C++ chỉ quy định như vậy.
- Chú ý rằng `std::multimap` không được trang bị toán tử `[]` điều này dễ hiểu vì một khóa `key` có thể có nhiều giá trị `value` cặp với nó.

1.5. Một số bài toán ví dụ

1.5.1. Đóng gói

Một người đi siêu thị mua n món hàng đánh số từ 0 tới $n-1$, món hàng thứ i có trọng lượng là w_i. Anh ta có những cái túi giống nhau với số lượng không hạn chế. Mỗi túi có thể chứa được trọng lượng tối đa là q ($q \geq w_i, \forall i: 0 \leq i < n$).

Để tránh mua thừa hoặc sót, các món hàng cần được mua theo đúng thứ tự từ 1 tới n. Ngoài ra để tiết kiệm túi, khi mua xong một món hàng, anh ta sẽ đưa ngay nó vào trong túi nặng nhất trong số những túi có thể cho thêm món hàng đó mà không làm trọng lượng túi vượt quá q. Trong trường hợp tất cả các túi đang dùng đều không thể chứa thêm món hàng mới mua, món hàng đó sẽ được đưa vào một túi mới.

Chú ý rằng khi một món đồ đã đưa vào túi, nó sẽ không bị lấy ra để chuyển sang túi khác vì việc này rất mất thời gian.

Yêu cầu: Tính số túi cần sử dụng nếu mua n món hàng theo quy tắc trên

Input

- Dòng 1 chứa số hai nguyên dương $n \leq 10^5, q \leq 10^9$
- Dòng 2 chứa n số nguyên dương $w_0, w_1, \dots, w_{n-1}$ ($\forall i: w_i \leq q$)

Output

Ghi ra số túi cần sử dụng

Ví dụ

Sample Input	Sample Output
6 14 8 4 6 1 7 2	Number of bags to use: 3

Giải thích
Theo cách làm của người mua:
Trọng lượng túi thứ nhất: 8 + 4 + 1 = 13
Trọng lượng túi thứ hai: 6 + 7 = 13
Trọng lượng túi thứ ba: 2

❋ Thuật toán

Thuật toán cho bài toán này là làm đúng như những gì đề bài mô tả. Vấn đề duy nhất phải xử lý là làm giảm độ phức tạp tính toán. Muốn vậy ta cần phải xử lý tốt hai thao tác sau đây đối với một món hàng vừa mua:

- Xác định túi để đưa món hàng vào theo quy tắc đặt ra
- Cập nhật tình trạng túi khi có thêm một món hàng đó

Gọi m là số túi cần sử dụng. Khi đó:

Nếu như biểu diễn các túi bằng các cấu trúc dữ liệu danh sách. Để xác định túi nặng nhất mà có thể cho thêm một món hàng, ta phải duyệt danh sách vào mất thời gian $O(m)$, việc cập nhật tình trạng trọng lượng của một túi sau khi cho một món hàng vào có thể thực hiện trong thời gian $O(1)$. Vì ta phải xử lý lần lượt cả n món hàng, thời gian thực hiện sẽ là $O(m \times n)$. Vấn đề là trong trường hợp xấu nhất, thuật toán thực hiện trong thời gian $\Omega(m \times n)$. Cũng có thể chứng minh được rằng nếu xác suất phải lấy thêm túi phân phối đều khi mua n mặt hàng thì thời gian thực hiện giải thuật trung bình cũng là $\Omega(m \times n)$. Thuật toán chỉ thích hợp với những bộ dữ liệu có số mặt hàng (n) hoặc số túi thực dụng (m) không quá lớn.

Nếu như biểu diễn các túi cũng bằng cấu trúc dữ liệu danh sách nhưng luôn duy trì danh sách các túi được sắp xếp tăng dần hoặc giảm dần theo trọng lượng, việc xác định túi để đưa một mặt hàng vào có thể sử dụng thuật toán tìm kiếm nhị phân $(O(\log m))$ nhưng sau khi cập nhật trọng lượng túi, sẽ phải di chuyển nó trong danh sách để bảo tồn thứ tự sắp xếp, việc này mất thời gian $O(m)$. Cách thứ hai này không tốt hơn cách thứ nhất, nhưng nó là gợi ý cho ta tìm cấu trúc dữ liệu thích hợp hơn.

Cấu trúc dữ liệu phù hợp để duy trì một tập hợp các phần tử sắp thứ tự đi kèm với thao tác thêm/bớt phần tử chính là cây BST.

Thuật toán:

Gọi dư lượng của một túi là trọng lượng tối đa có thể thêm vào túi đó mà không làm trọng lượng của túi vượt quá giới hạn q. Duy trì tập hợp S các túi đang sử dụng, sắp xếp tăng dần theo dư lượng. Ban đầu $S = \emptyset$.

Khi mua xong mặt hàng thứ i (có trọng lượng w_i), ta tìm túi có dư lượng nhỏ nhất $\geq w_i$:

- Nếu tìm thấy, đưa mặt hàng thứ i vào túi, dư lượng của túi giảm đi w_i, cập nhật S theo dư lượng mới của túi (điều này tương đương với xóa túi cũ khỏi S và đẩy vào S túi mới)
- Nếu không tìm thấy, lấy một túi mới để đưa mặt hàng thứ i vào, đưa túi mới vào S với dư lượng $q - w_i$.

Tập S có thể biểu diễn bằng `std::multiset<int>`, việc tìm túi có dư lượng nhỏ nhất $\geq w_i$ được thực hiện bằng phương thức `lower_bound(…)`, việc xóa một túi hay thêm vào một túi có thể thực hiện bằng cặp phương thức `erase(…)` và `insert(…)`.

❋ Cài đặt

PACKAGES.cpp 🗎 Bài toán đóng gói

```
#include <iostream>
#include <set>
using namespace std;

int main()
{
    multiset<int> S; //Tập chứa dư lượng các túi
    int n, q;
    cin >> n >> q;
    while (n-- > 0) //Lặp n lần mua hàng
    {
        int w;
        cin >> w; //Mua 1 món hàng với trọng lượng w
        auto it = S.lower_bound(w); //Tìm túi có dư lượng nhỏ nhất >= w
        if (it == S.end()) //Không tìm thấy, mọi túi có dư lượng < w
            S.insert(q - w); //Lấy một túi mới cho món hàng vào
        else //Tìm thấy
        {
            int cap = *it; //cap = Dư lượng cũ của túi
            S.erase(it); //Xóa túi cũ khỏi S
            S.insert(cap - w); //Thêm vào S túi đã cập nhật dư lượng
        }
    }
    cout << "Number of bags to use: " << S.size();
}
```

❋ Nói thêm về bài toán đóng gói

Cần nhận định rằng cách làm của người mua tiết kiệm số túi theo cảm tính của anh ta, không phải cách tối ưu dùng ít túi nhất. Như trong ví dụ này ta chỉ cần 2 túi với trọng lượng mỗi túi là 14 ($14 = 8 + 4 + 2 = 6 + 1 + 7$).

Vấn đề tìm phương án tối ưu (theo tiêu chuẩn dùng ít túi nhất) đã được chứng minh là bài toán NP-khó, hiện chưa có thuật toán với độ phức tạp đa thức để giải quyết.

Một phương pháp xấp xỉ cho bài toán tối ưu là sắp xếp trước các món hàng theo thứ tự từ nặng tới nhẹ và thực hiện cách làm như người mua trong mục này. Phương pháp này xuất phát từ cách làm thực tế: Món hàng nặng hơn sẽ được ưu tiên cho vào túi trước, món hàng nhẹ hơn sẽ cho vào túi sau, bởi vì món hàng nhẹ hơn sẽ dễ tận dụng những túi đang dùng hơn là món hàng nặng hơn. Như ở ví dụ mẫu, giới hạn trọng lượng túi $q = 14$, có 6 món hàng với trọng lượng (8,4,6,1,7,2), khi sắp xếp lại ta sẽ được (8,7,6,4,1,2). Theo thứ tự này ta sẽ phương án tối ưu dùng 2 túi: $14 = 8 + 6 = 7 + 4 + 1 + 2$.

Mặc dù phương pháp xấp xỉ cho kết quả khá tốt, nó cũng không phải là thuật toán cho phương án tối ưu. Ví dụ với 6 món hàng với trọng lượng là (6,5,3,2,2,2) và giới hạn trọng lượng mỗi túi là 10. Phương pháp xấp xỉ sẽ cho kết quả là 3 túi, trong khi phương án tối ưu chỉ cần 2 túi mà thôi.

1.5.2. Cặp phần tử bằng nhau

Cho dãy số nguyên $A = (a_0, a_1, \dots, a_{n-1})$. Hãy cho biết trong dãy A có bao nhiêu cặp phần tử bằng nhau, tức là đếm số cặp chỉ số (i, j) thỏa mãn:

$$\begin{cases} 0 \le i < j < n \\ a_i = a_j \end{cases}$$

Input

- Dòng 1 chứa số nguyên dương $n \le 10^6$
- Dòng 2 chứa n số nguyên $a_0, a_1, \dots, a_{n-1}$ $(\forall i: |a_i| \le 10^9)$

Output

Số cặp phần tử bằng nhau trong dãy A

Sample Input	Sample Output
9 1 2 3 1 2 1 3 1 2	There are 10 pairs of equal numbers

Thuật toán

Thuật toán tầm thường là quét mọi cặp chỉ số $(i < j)$ và kiểm tra điều kiện $a_i = a_j$.

```
Result = 0;
for (j = 0; j < n; ++j)
{
    x = a[j];
    //Đếm c là số phần tử đứng trước vị trí j và giá trị bằng x
    c = 0;
    for (i = 0; i < j; ++i)
        if (a[i] == x) ++c;
    Result += c; //Cộng c vào kết quả
}
Output ← Result;
```

Thuật toán thực hiện trong thời gian $\Theta(n^2)$ và không thích hợp để xử lý dữ liệu lớn. Tuy nhiên nó lại gợi ý cho ta một hướng cải tiến: Làm thế nào để đếm nhanh được số phần tử đứng trước vị trí j mà có giá trị bằng a_j.

Ý tưởng cho thuật toán tốt hơn: Cùng với quá trình duyệt qua từng vị trí `j`, ta duy trì các giá trị `c[x]` để thống kê số phần tử bằng `x` trong dãy `A` tính đến trước vị trí `j`:

- Khi duyệt tới vị trí `j`, `Result` được tăng lên `c[a[j]]`, bởi vì phần tử `a[j]` có `c[a[j]]` phần tử đứng trước nó và bằng nó.
- Sau khi duyệt qua vị trí `j`, `c[a[j]]` được tăng lên 1 để chuẩn bị xét tới vị trí `j + 1`.

Trong trường hợp các giá trị trong mảng A là số nguyên trong phạm vi hẹp, chẳng hạn từ 0 tới 9, ta có thể sử dụng mảng `c[0 … 9]` cho mục đích thống kê. Tuy nhiên nếu các giá trị trong mảng A trải ra trong phạm vi rất lớn, vì không thể khai báo mảng `c`, ta có thể dùng `map<int, int>` để thay thế.

Cài đặt

EQUALPAIRS.cpp Cặp số bằng nhau

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map<int, int> c;
    int n, x, Result = 0;
    cin >> n;
    while (n-- > 0)
    {
        cin >> x;
        Result += c[x]++;
    }
    cout << "There are " << Result << " pairs of equal numbers";
}
```

Chú ý dòng lệnh thứ 13: `Result += c[x]++;` Khi đọc được một phần tử bằng `x` ta có `c[x]` là số phần tử đã đọc được và bằng `x`, tiếp theo `Result` được tăng lên `c[x]` và `c[x]` được tăng lên 1 để thống kê phần tử `x` vừa xử lý xong. Thời gian thực hiện giải thuật của lệnh này là $O(\log n)$ do mất một lần tìm kiếm phần tử có khóa bằng x trong std::map.

Mặc dù không có sự khác biệt về độ phức tạp tính toán, cách viết như sau sẽ chậm hơn đáng kể:

```
Result = Result + c[x];
c[x] = c[x] + 1;
```

Bởi vì việc truy cập phần tử `c[x]` mất thời gian $O(\log n)$.

Bài tập 1-1

Cho một BST khác rỗng gồm n nút.

Chứng minh rằng thuật toán sau in ra các khóa trên BST theo thứ tự tăng dần và thời gian duyệt qua tất cả các nút của BST là $O(n)$

```
for (PNode x = Minimum(root); x != nil; x = Successor(x))
    Output ← x->key;
```

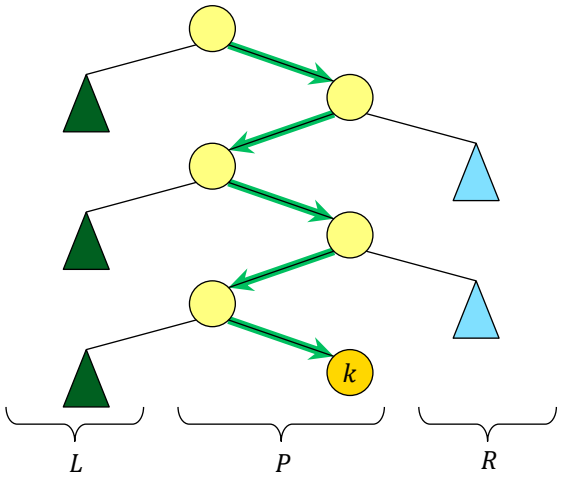
Chứng minh rằng thuật toán sau in ra các khóa trên BST theo thứ tự giảm dần và thời gian duyệt qua tất cả các nút của BST cũng là $O(n)$

```
for (PNode x = Maximum(root); x != nil; x = Predecessor(x))
    Output  ← x->key;
```

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

Bài tập 1-2.

Quá trình tìm kiếm trên BST có thể coi như một đường đi P xuất phát từ nút gốc. Nếu đường đi trong quá trình tìm kiếm kết thúc ở một nút lá chứa khóa k, ký hiệu L là tập các giá trị chứa trong các nút nằm bên trái đường đi và R là tập các giá trị chứa trong các nút nằm bên phải đường đi.



a) Giáo sư X phát hiện ra một tính chất: Với $\forall x \in L, y \in R$, ta có $x \leq k \leq y$. Chứng minh phát hiện của giáo sư X là đúng hoặc chỉ ra một phản ví dụ.

b) Giáo sư Y phát hiện ra một tính chất: Với $\forall x \in L, y \in R, q \in P$, ta có $x \leq q \leq y$. Chứng minh phát hiện của giáo sư Y là đúng hoặc chỉ ra một phản ví dụ.

Bài tập 1-3.

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

Bài tập 1-4.

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

Bài tập 1-5.

Viết thuật toán để tìm nút chứa khóa lớn nhất $\leq k$ trong BST.

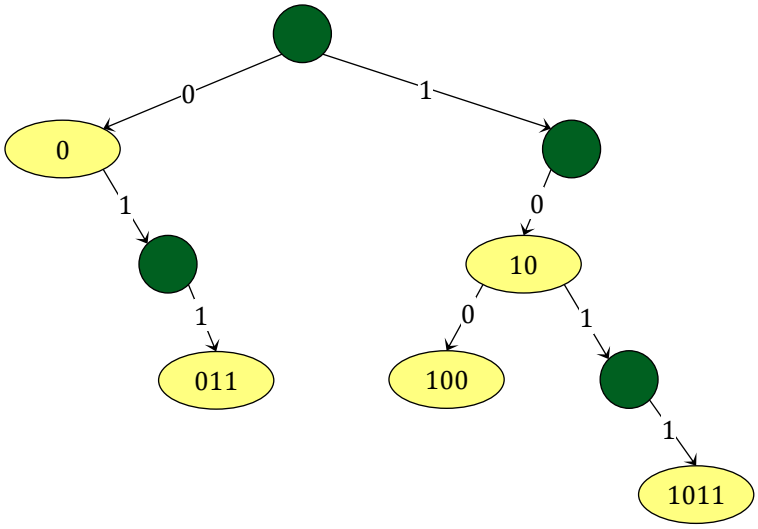
Bài tập 1-6.

Viết thuật toán để tìm nút chứa khóa nhỏ nhất $\geq k$ trong BST.

Bài tập 1-7.

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

Các giá trị khóa lưu trữ trên Radix Tree là các dãy nhị phân, hay tổng quát hơn là một kiểu dữ liệu nào đó có thể mã hóa bằng các dãy nhị phân. Phép chèn một khóa vào Radix Tree được thực hiện như sau: Bắt đầu từ nút gốc ta duyệt biểu diễn nhị phân của khóa, gặp bit 0 đi sang nút con trái và gặp bit 1 đi sang nhánh con phải, mỗi khi không đi được nữa (đi vào liên kết `nil`), ta tạo ra một nút và nối nó vào cây ở chỗ liên kết `nil` vừa rẽ sang rồi đi tiếp. Cuối cùng ta đặt khóa vào nút cuối cùng trên đường đi. Hình dưới đây là Radix Tree sau khi chèn các giá trị 1011, 10, 100, 0, 011. Các nút tô đậm không chứa khóa



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

Bài tập 1-8.

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

- Chứng minh rằng $b(0) = 1$ và $b(n) = \sum_{k=1}^{n} b(k)b(n-k)$
- Chứng minh rằng $b(n) = \frac{1}{n+1}\binom{2n}{n}$ (số catalan thứ n). Từ đó suy ra xác suất để BST là cây nhị phân gần hoàn chỉnh (hoặc cây nhị phân suy biến) nếu n khóa được chèn vào theo thứ tự ngẫu nhiên.
- Chứng minh công thức xấp xỉ $b(n) = \frac{4^n}{\sqrt{\pi}n^{3/2}}\left(1 + O(1/n)\right)$