Bài 1. Cây Splay

1.1. Cây nhị phân tìm kiếm tự cân bằng

Để tăng tính hiệu quả của các thao tác cơ bản trên BST, cách chung nhất là cố gắng giảm độ sâu của các nút trên cây. Với một BST gồm n nút, dĩ nhiên giải pháp lý tưởng là ta xây dựng được cây nhị phân tìm kiếm tối ưu, tuy nhiên chúng ta sẽ vấp phải các khó khăn sau:

- Trong đa số trường hợp chúng ta không biết được tần suất tìm kiếm mỗi khóa trên BST.
- Ngay cả trong trường hợp biết được tần suất tìm kiếm mỗi khóa (chẳng hạn như có thể giả định xác suất tìm kiếm các khóa phân phối đều trên tập các giá trị khóa), các thuật toán xây dựng cây nhị phân tìm kiếm tối ưu hiện rất chậm và làm ảnh hưởng nhiều tới thời gian thực hiện giải thuật. Điều này xảy ra trong trường hợp cây BST luôn biến động về cấu trúc sau các phép chèn/xóa.

Giả định rằng xác suất tìm kiếm các khóa là như nhau, người ta nhận thấy rằng muốn thuật toán tìm kiếm thực hiện nhanh thì phải cố gắng giữ được độ sâu của các nút được truy cập càng nhỏ càng tốt. Cho đến nay có ba cách tiếp cận chính để thực hiện điều này:

Cách thứ nhất là không để ý đến cấu trúc cây BST tổng thể, chỉ quan tâm tới các nút được truy cập. Cấu trúc BST tại một thời điểm nhất định có thể rất "xấu", nhưng nó lại có xu hướng đẩy những nút truy cập thường xuyên lên gần gốc, đảm bảo rằng một dãy nhiều thao tác tìm kiếm, chèn, xóa được thực hiện nhanh dù rằng một thao tác riêng lẻ trong dãy có thể thực hiện chậm. Cây splay và các biến thể của nó là những ví dụ điển hình cho cách tiếp cận này.

Cách thứ hai là dựa vào quan sát: khi các khóa được chèn vào trong BST theo thứ tự ngẫu nhiên thì trung bình độ cao của BST là một đại lượng $O(\log n)$*. Vì vậy, cho dù thứ tự chèn xóa trên BST như thế nào, người ta tìm cách hiệu chỉnh sao cho cấu trúc của BST giống như ta chèn các khóa vào theo một trật tự ngẫu nhiên, điều này khiến xác suất xảy ra trường hợp xấu rất nhỏ và hầu như không gặp phải trên thực tế. Các dạng cây nhị phân tìm kiếm ngẫu nhiên, treap là những ví dụ cho cách tiếp cận này.

Cách thứ ba triệt để hơn, dựa vào các kỹ thuật cân bằng cây để luôn giữ cho chiều cao cây luôn là một đại lượng $O(\log n)$. Đây thực ra là những phát kiến đầu tiên cho cấu trúc BST tự cân bằng như cây AVL, cây đỏ đen,…

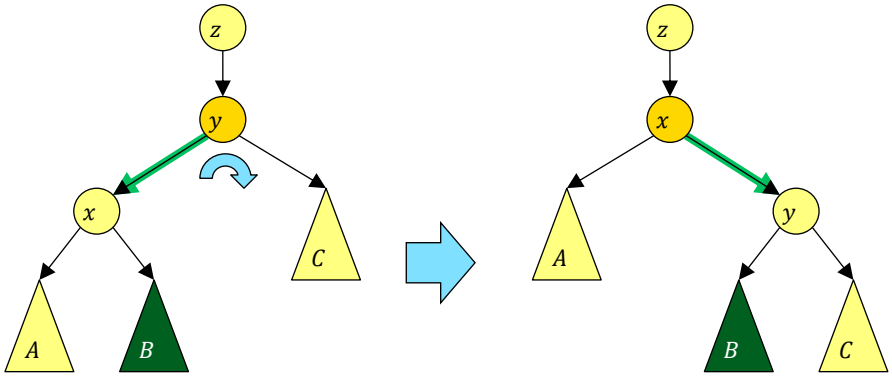
1.2. Phép quay cây

Phép quay cây là thao tác cơ bản để hiệu chỉnh cấu trúc của hầu hết các dạng BST tự cân bằng. Phép quay cây có hai loại: quay phải (*right rotation*) và quay trái (*left rotation*).

---

* Cụ thể là $\lim_{n\to\infty} \frac{H_n}{\lg n} = c \approx 4.311$ với H_n là độ cao trung bình của BST với n nút [1].

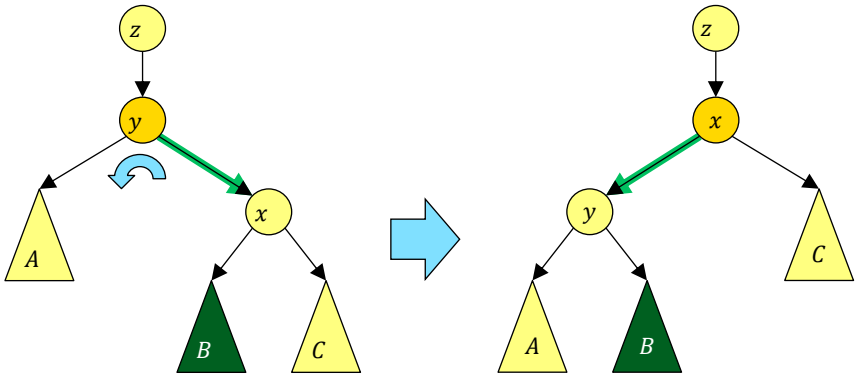
Nếu x là con trái của y, phép quay phải theo liên kết $y \rightarrow x$ cắt nhánh con phải của x làm con trái của y sau đó cho y làm con phải của x. Nút x được đẩy lên làm gốc nhánh thay cho nút y (Hình 1-1).



Thứ tự giữa được bảo tồn: A, x, B, y, C

Hình 1-1. Phép quay phải theo liên kết $y \rightarrow x$: Cho B làm con trái y và y làm con phải x

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. Nút x được đẩy lên làm nút nhánh thay cho nút y (Hình 1-2).



Thứ tự giữa được bảo tồn: A, y, B, x, C

Hình 1-2. Phép quay trái theo liên kết $y \rightarrow x$: Cho B làm con phải y và y làm con trái x

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

Ta sẽ viết một hàn UpTree(x) tổng quát hơn: Với $x \neq root$, và $y = x->P$, phép UpTree(x) sẽ quay theo liên kết $y \longrightarrow x$ để đẩy nút x lên phía gốc cây (độ sâu của x giảm 1) và kéo nút y xuống sâu hơn một mức làm con nút x.

```
void UpTree(PNode x) //Đẩy x lên 1 bước về phía gốc cây
{
    PNode y = x->P; //y là cha của x
    PNode z = y->P; //z là cha của y
    if (x == y->L) //Quay phải
    {
        SetL(y, x->R); //Cho con phải x làm con trái y
        SetR(x, y); //Cho y làm con phải x
    }
    else
    {
        SetR(y, x->L); //Cho con trái x làm con phải y
        SetL(x, y); //Cho y làm con trái x
    }
    SetLink(z, x, z->R == y); //Cho x làm con z thế chỗ cho y
}
```

1.3. Cây Splay

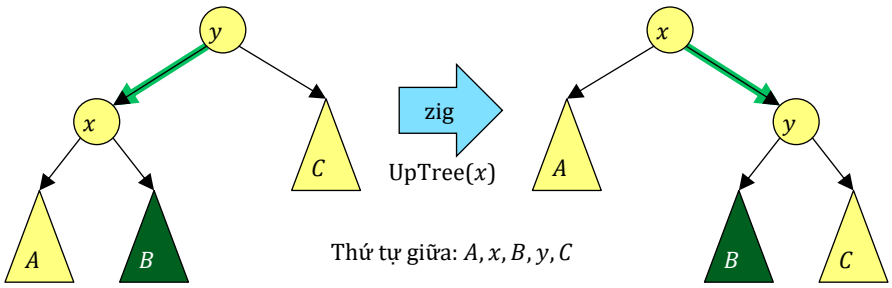
Cây Splay là một trong những ý tưởng thú vị nhất về cây nhị phân tìm kiếm tự cân bằng [1]. Thao tác làm "bẹp" cây (splay) được thực hiện ngay khi có lệnh truy cập nút, làm giảm độ sâu của những nút truy cập thường xuyên. Trong trường hợp tần suất thực hiện phép tìm kiếm trên một khóa hay một cụm khóa cao hơn hẳn so với những khóa khác, cây Splay sẽ phát huy được ưu thế về mặt tốc độ.

1.4. Các thao tác trên cây Splay

1.4.1. Thao tác Splay

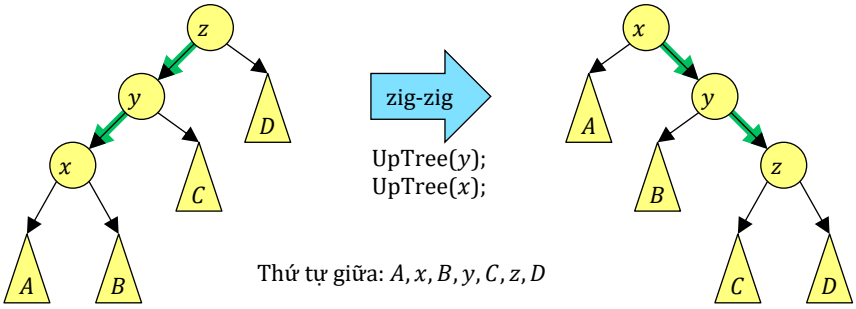
Thao tác quan trọng nhất của cây splay là thao tác $Splay(x)$: nhận vào một nút x và đẩy x lên làm gốc cây. Chừng nào x chưa phải gốc cây, gọi y là nút cha của x ($y = x->P$) và z là nút cha của y ($z = y->P$), x sẽ được đẩy dần lên gốc cây theo quy trình sau:

- Nếu x là con của nút gốc ($z = nil$), một phép $UpTree(x)$ sẽ được thực hiện và x sẽ trở thành gốc cây. Thao tác này gọi là phép *zig*.

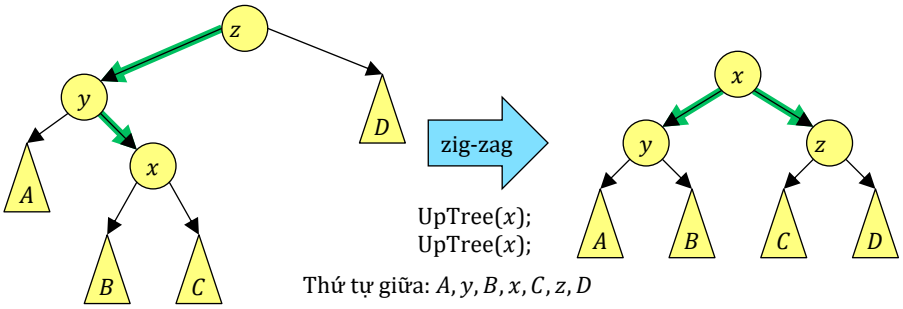


- Nếu x và y cùng là con trái hoặc cùng là con phải, một phép $UpTree(y)$ sẽ được thực hiện để đẩy y lên làm nút cha của z, tiếp theo là một phép $UpTree(x)$ để đẩy x lên làm nút cha

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*.

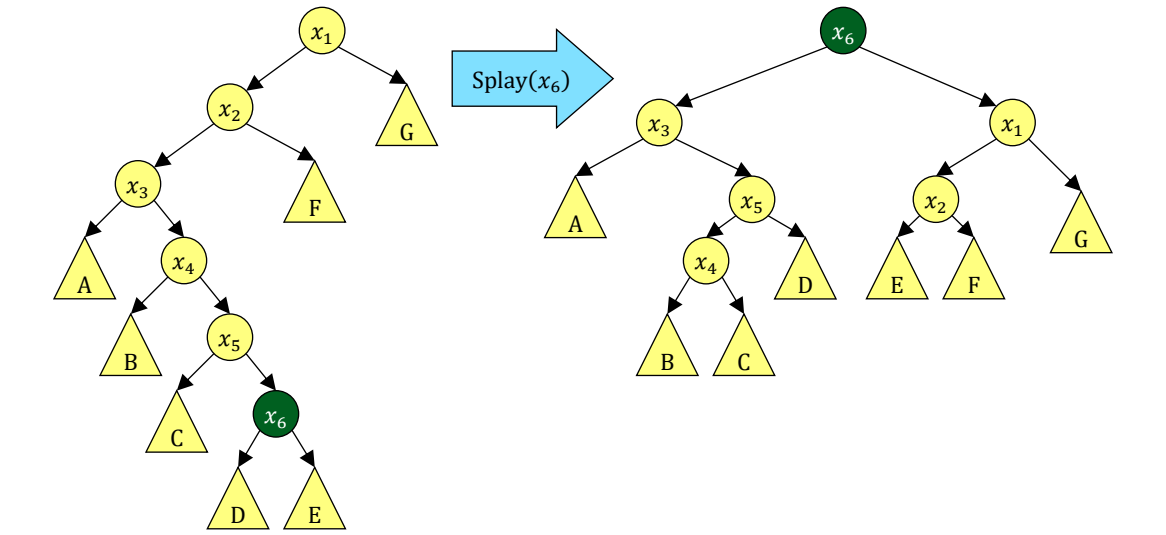


Thao tác $Splay(x)$ cần sử dụng tối đa một phép zig.

```
void Splay(PNode x)
{
    PNode y, z;
    while ((y = x->P)!= nil) //Chừng nào x chưa phải gốc, y là cha x
    {
        z = y->P; //z là cha y
        if (z != nil) //zig-zig hoặc zig-zac. Cần một phép UpTree(.) trước UpTree(x)
            (x == y->L) == (y == z->L) ? UpTree(y) : UpTree(x);
        UpTree(x);
    }
}
```

Tại sao lại làm phức tạp hóa 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 gốc tới x. 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 phía gốc.

Hình 1-3 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.



Hình 1-3. Ví dụ về thao tác Splay

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:

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

1.4.2. Tìm kiếm

Thao tác tìm kiếm *Search* được thực hiện tương tự như trên BST, chỉ có điều bất kể quá trình tìm kiếm thành công hay thất bại, nút đi qua cuối cùng trong quá trình tìm kiếm sẽ được đẩy lên thành gốc cây bằng thao tác *Splay*:

```
//Trả về nút chứa khóa = k, trả về nil nếu không tìm thấy
PNode Search(const TKey& k)
{
    PNode x = root;
    while (x != nil && x->key != k) //Nút x chứa khóa khác k
    {
        root = x; //Lưu lại vị trí nút trước khi đi xuống nút con
        x = k < x->key ? x->L : x->R; //Đi xuống nhánh con…
    }
    if (x != nil) root = x; //Nếu tìm thấy, đặt root = x
    if (root != nil) Splay(root); //Bất kể tìm thấy hay không, root được đẩy lên trở lại làm gốc
    return x;
}
```

1.4.3. Tách

Phép tách (*Split*) một nút x tách cây chứa x thành hai cây A và B. Mọi nút đứng trước x theo thứ tự giữa được chuyển sang cây A, trong khi đó nút x và những nút đứng sau nó theo thứ tự giữa được chuyển sang cây B. Quy ước rằng nếu $x = nil$ thì $A = T$ và $B = nil$.

Điều này thực hiện đơn giản bằng cách: Thực hiện phép $Splay(x)$ để đẩy x lên gốc, khi đó nhánh con trái của x chính là cây A, x cùng với nhánh con phải của nó tạo thành cây B.

```
//Tách cây T thành hai cây A, B. Điểm cắt là nút x
void Split(PNode T, PNode x, PNode& A, PNode& B)
{
    if (x == nil)
    {
        A = T;
        B = nil;
    }
    else
    {
        Splay(x); //Đẩy x lên gốc
        A = x->L; A->P = nil; //Cắt nhánh con trái của x ra làm cây A
        x->L = nil; B = x; //x và nhánh con phải tạo thành cây B
    }
}
```

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

Pl

1.4.4. Ghép

Với hai cây A và B trong đó mọi khóa trong A đều nhỏ hơn hoặc bằng mọi khóa trong B, phép ghép ($Join$) sẽ tạo cây mới chứa các nút của cả hai cây A, B.

- Nếu $B = nil$, phép ghép đơn giản trả về cây A.
- Nếu $B \neq nil$, phép ghép tìm nút cực trái của cây B, gọi là nút x ($x = Minimum(B)$), thực hiện phép $Splay(x)$ để đẩy x lên thành gốc cây. Khi đó gốc x chắc chắn không có nhánh con trái. Việc cuối cùng là cho A làm nhánh con trái của x và trả về x là gốc cây sau khi ghép.

```
//Ghép hai cây A, B thành một cây, trả về trong kết quả hàm
PNode Join(PNode& A, PNode& B)
{
    if (B == nil) return A; //Cây B rỗng, trả về A
    while (B->L != nil) B = B->L; //Tìm nút cực trái cây B
    Splay(B); //Đẩy nút cực trái cây B lên gốc, gốc này không có nhánh con trái
    SetL(B, A); //Cho A làm nhánh con trái cây B
    return B;
}
```

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

1.4.5. Chèn và xóa

Phép chèn và xóa trên cây Splay được thực hiện dễ dàng qua các thao tác tách và ghép:

- Để chèn một khóa k vào cây T, ta tìm nút x đầu tiên theo thứ tự giữa chứa khóa $> k$, cây T được tách thành hai cây: Các nút đứng trước x theo thứ tự giữa được đưa vào cây A, nút x và các nút đứng sau nó theo thứ tự giữa được đưa vào cây B. Có thể thấy rằng các khóa trong A đều $\leq k$ và các khóa trong B đều $> k$. Tiếp theo ta tạo nút x chứa khóa k và cho A, B lần lượt làm nhánh con trái và con phải của x. Thu được cây mới gốc x.

- Để xóa một nút x khỏi cây T, ta đẩy x lên gốc cây và thực hiện phép ghép nhánh con trái với nhánh con phải của x, được cây mới sau khi đã xóa nút x

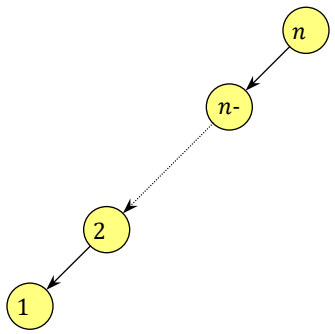
```
//Chèn khóa k vào cây
void Insert(const TKey& k)
{
    PNode x = nil, y = root, A, B;
    while (y != nil) //Bắt đầu từ gốc, tìm nut đầu tiên theo thứ tự giữa chứa khóa > k
        if (y->key <= k) y = y->R; //y chứa khóa ≤ k, tìm tiếp trong nhánh con phải
        else x = y, y = y->L; //y chứa khóa > k, ghi nhận lại y, tìm tiếp trong nhánh con trái
    Split(root, x, A, B); //Dùng nút x, tách thành 2 cây A, B
    root = new TNode; //Tạo nút gốc mới
    root->P = nil;
    root->key = k; //Đưa khóa k vào gốc mới
    SetL(root, A); //Cho A làm con trái gốc mới
    SetR(root, B); //Cho B làm con phải gốc mới
}

//Xóa nút x
void Delete(PNode x)
{
    Splay(x); //Đẩy x lên gốc
    x->L->P = x->R->P = nil; //Cắt rời nhánh trái và nhánh phải của x
    root = Join(x->L, x->R); //Nối nhánh trái với nhánh phải thành cây mới
    delete x; //Hủy nút x
}
```

1.4.6. Một số chú ý khi cài đặt

Có thể chỉ ra nhiều ví dụ mà cây Splay bị suy biến (mỗi nút trong chỉ có đúng một nút con), chẳng hạn khi ta chèn các khóa vào cây Splay theo thứ tự tăng dần hoặc giảm dần. Cây Splay chỉ có thể hiệu chỉnh lại cấu trúc khi ta thực hiện các thao tác $Splay$, (chẳng hạn khi gọi hàm $Search$). Chính vì vậy, luôn nhớ thực hiện phép $Splay(x)$ sau mỗi lần thực hiện xong phép di chuyển từ nút gốc xuống nút x.

Có thể lấy ví dụ: Với số nguyên n tương đối lớn, chèn lần lượt các khóa 1,2, … , n vào cây Splay. Cây tạo thành sẽ là cây nhị phân suy biến lệch trái với nút gốc là nút n (Hình 1-4)



Hình 1-4. Cây Splay khi chèn lần lượt các khóa theo thứ tự tăng dần

Như vậy nếu ta thực hiện n lần gọi hàm $Minimum(root)$ với cách cài đặt hàm như trên BST thông thường, thời gian thực hiện của mỗi lời gọi hàm là $\Theta(n)$, tổng thời gian thực hiện n lần $Minimum(root)$ là $\Theta(n^2)$.

Nhưng đối với cây Splay, sau khi thực hiện lời gọi hàm $Mininum(root)$ thứ nhất, nút cực trái trên cây sẽ được đẩy lên làm gốc và gốc $root$ mới này chắc chắn không có con trái. Điều này làm cho tất cả các lời gọi hàm $Minimum(root)$ tiếp theo có thời gian thực hiện $\Theta(1)$. Tổng thời gian thực hiện n lời gọi $Minimum(root)$ là $\Theta(n)$.

1.5. Hiệu lực của cây Splay

Thời gian thực hiện các phép tách, ghép, chèn, xóa có thể đánh giá qua thời gian thực hiện thao tác $Splay$.

Trong trường hợp xấu nhất, cây suy biến, thời gian thực hiện thao tác $Splay(x)$ là $\Theta(n)$ với n là số nút trên cây. Tuy nhiên thao tác $Splay$ được gọi thực hiện nhiều lần trong quá trình xây dựng cây cũng như trong quá trình tìm kiếm, nên ta sẽ đánh giá thời gian thực hiện giải thuật của một dãy các thao tác $Splay$.

Với cây Splay có n nút, các nút được đánh số từ 1 tới n theo thứ tự giữa. Xét dãy S gồm m phép $Splay$ tác động lần lượt lên các nút $x_1, x_2, \dots, x_m$ $(1 \le x_i \le n, \forall i)$

Định lý 1-1

Thời gian thực hiện của dãy thao tác S là $O\big((m+n)\log n + m\big)$

Định lý 1-2

Trong dãy thao tác S, gọi q_i là số lần nút thứ i được "splay", giả thiết rằng các $q_i \ge 1$. Khi đó thời gian thực hiện dãy thao tác S là $O\left(m + \sum_{i=1}^{n} q_i \log\frac{m}{q_i}\right)$.

Khi thực hiện phép $Splay$ thứ i tác động lên nút x_i, ta xét phép $Splay$ gần nhất trước đó cũng tác động lên nút x_i (tính từ đầu dãy thao tác nếu đây là lần đầu tiên ta thực hiện $Splay(x_i)$), ký hiệu t_i là số nút khác nhau được "splay" giữa hai thời điểm này.

Định lý 1-3

Tổng thời gian thực hiện dãy thao tác S là $O(m + n\log n + \sum_{i=1}^{m}\log(t_i+1))$

Việc chứng minh các định lý kể trên khá dài dòng, chúng ta sẽ chỉ diễn giải nội dung để hiểu bản chất của chúng.

Thời gian thực hiện một thao tác $Splay$ có thể rất tồi tệ trong trường hợp xấu nhất, nhưng Định lý 1-1 cho biết rằng cây Splay cũng "tốt" như cây nhị phân gần hoàn chỉnh nếu chúng ta cần thực hiện không ít hơn n phép tìm kiếm.

Lý thuyết cũng đã chứng minh rằng với mọi cấu trúc cây nhị phân tìm kiếm tĩnh (không có sự thay đổi cấu trúc cây trong quá trình tìm kiếm), thời gian thực hiện m phép tìm kiếm trong trường hợp xấu nhất là $\Omega\left(m + \sum_{i=1}^{n} q_i \log\frac{m}{q_i}\right)$ kể cả với cây nhị phân tìm kiếm tối ưu, và như vậy, Định lý 1-2 cho thấy cây Splay cũng "tốt" như cây nhị phân tìm kiếm tối ưu.

Định lý 1-3 cho thấy sự ảnh hưởng của độ lớn các t_i lên thời gian thực hiện dãy thao tác S: các t_i càng nhỏ, cây Splay càng hiệu quả. Nói cách khác, phép $Splay(x)$ sẽ thực hiện nhanh nếu như gần trước đó cũng đã có phép $Splay(x)$ trong dãy thao tác S. Đây là đặc điểm nổi bật của cây Splay: Những khóa nào mới được tìm kiếm sẽ được tìm lại nhanh hơn các khóa khác.

Ngoài ra người ta cũng chứng minh được các kết quả sau đối với cây Splay:

Định lý 1-4

Gọi x^* là một nút bất kỳ trong dãy $x_1, x_2, \dots, x_m$, khi đó thời gian thực hiện dãy thao tác S là $O(n \log n + m + \sum_{i=1}^{m} \log(|x_i - x^*| + 1))$.

Định lý 1-5

Nếu ta thực hiện phép $Splay$ trên các nút lần lượt theo thứ tự 1, 2, …, n thì thời gian thực hiện toàn bộ n thao tác $Splay$ là $\Theta(n)$, không phụ thuộc vào cấu trúc ban đầu của cây Splay.

Cần phải nói rõ về từ "tốt" trong các diễn giải ở trên, ví dụ "cây Splay tốt như cây nhị phân gần hoàn chỉnh". Điều này cần hiểu là: Trong trường hợp xấu nhất, nếu dãy thao tác thực hiện trên cây nhị phân gần hoàn chỉnh mất thời gian T thì dãy thao tác đó thực hiện trên cây Splay sẽ chỉ mất thời gian $O(T)$.

Những kết quả lý thuyết trên cho thấy những ưu điểm và nhược điểm của cây Splay:

Ưu điểm:

- Nếu bỏ qua hằng số nhân vào thời gian thực hiện giải thuật khi đánh giá bằng ký pháp O, cây Splay ít ra là không tồi hơn các cấu trúc cây nhị phân tìm kiếm tự cân bằng khác.
- Khi có một nhóm giá trị khóa được tìm kiếm với tần suất cao hơn các giá trị khóa khác, cây Splay tỏ ra rất hiệu quả.
- Cây Splay cần ít bộ nhớ khi cài đặt vì nó không cần lưu trữ thông tin về mức độ cân bằng tại các nút.
- Các thao tác cơ bản: chèn, xóa, tìm kiếm, tách, ghép… khá đơn giản và dễ dàng cài đặt.

Nhược điểm:

- Cấu trúc của cây luôn phải hiệu chỉnh lại sau mỗi phép tìm kiếm
- Một thao tác cơ bản thực hiện riêng rẽ có thể thực hiện chậm, điều này không thích hợp trong các ứng dụng yêu cầu xử lý trực tuyến (*online*) với tốc độ ổn định.

Các thực nghiệm cũng cho thấy khi xác suất tìm kiếm phân phối đều trên tập các giá trị khóa, cây Splay chậm hơn các dạng cây nhị phân tìm kiếm tự cân bằng khác. Người ta cũng đề xuất nhiều phương pháp để cải tiến tốc độ của cây Splay nhưng cũng chỉ khắc phục được phần nào nhược điểm chủ yếu của nó: Phải hiệu chỉnh lại cấu trúc của cây sau mỗi phép tìm kiếm.

Những ý tưởng cải tiến cây splay sẽ được nêu ra trong phần bài tập. Trong phần sau, ta sẽ khảo sát kỹ thuật sử dụng số ngẫu nhiên để cài đặt một dạng BST hiệu quả hơn: cấu trúc Treap.

Bài tập 1-1.

Hãy cài đặt cây Splay và thử nghiệm các thao tác chèn/xóa/tìm kiếm trên các khóa được lựa chọn ngẫu nhiên.

Bài tập 1-2.

Một phương pháp khác tốt hơn để chèn/xóa các khóa trên cây Splay như sau:

Để chèn một khóa vào cây Splay, ta thực hiện phép chèn như trên BST để chèn khóa mới vào một nút là x, sau đó thực hiện phép $Splay(x)$ để đẩy x lên thành gốc cây. Để xóa một nút x khỏi BST, ta thực hiện phép xóa như trên BST (quy về trường hợp x có ít hơn 2 nút con), sau đó nút cha của x sẽ được đẩy lên thành gốc cây bằng phép $Splay$.

Hãy sửa đổi chương trình cài đặt cây Splay theo phương pháp mới này và so sánh hiệu suất với cách cài đặt cũ.

Bài tập 1-3. (Top-down Splay)

Phép $Splay(x)$ trong bài còn gọi là "bottom-up splay". Thông thường trước khi ta thực hiện phép $Splay(x)$, ta phải thực hiện một phép di chuyển từ nút gốc xuống nút x, sau đó phép $Splay(x)$ lại phải lần ngược đường đi từ x lên gốc, điều này có thể gây lãng phí thời gian.

"Top-down splay" là một phương pháp cải tiến tốc độ của cây Splay: Ngay trong quá trình đi từ gốc xuống x, ta tách cây Splay T thành hai cây T_L và T_R. T_L chứa các nút đứng trước x và T_R chứa các nút đứng sau x theo thứ tự giữa. Sau đó chỉ việc cho nút x làm gốc với hai nhánh con là T_L và T_R. Ý tưởng này được dựa trên quan sát sau: Trong quá trình đi từ gốc xuống x mà đi qua nút y. Nếu từ y ta đi tiếp sang trái thì y và nhánh con phải của nó được chuyển sang cây T_R, nếu từ y ta đi tiếp sang phải thì y và nhánh con trái của nó sẽ được chuyển sang cây T_L. Vấn đề cần cẩn thận duy nhất là đảm bảo cấu trúc hai cây T_L, T_R trong trường hợp zig-zig và zig-zag.

Hãy cài đặt thử nghiệm cây Splay sử dụng kỹ thuật này. Chú ý: khi cài đặt Top-down splay, không cần sử dụng con trỏ tới nút cha (P).

Bài tập 1-4. (Semi-splay)

Kỹ thuật semi-splay được dùng để cải thiện tốc độ của cây Splay. Các thực nghiệm và đánh giá lý thuyết cho thấy kỹ thuật này có thể giảm thời gian thực hiện giải thuật xuống còn $\sim 2/3$ so với phép $Splay$ thông thường.

Để cài đặt kỹ thuật semi-splay, ta cần cài đặt phép chèn/xóa như ở Bài tập 1-2. Phép semi-splay được sử dụng để thay thế cho phép splay theo cách sau: Trường hợp zig và zig-zag được thực hiện như cũ, chỉ có trường hợp zig-zig được thực hiện theo cách: Giả sử ta đang đứng ở nút x và y là nút cha của x, ta thực hiện UpTree(y) và gán $x = y$ rồi lặp lại (không thực hiện UpTree(x)). Điều này có nghĩa là khi gặp trường hợp zig-zig, ta đẩy nút y lên bằng một phép UpTree rồi tiếp tục quá trình splay từ y. Chú ý là thao tác semi-splay(x) có thể không đẩy x lên thành gốc.

Hãy cài đặt cây splay với kỹ thuật này.