TRƯỜNG ĐẠI HỌC ĐÀ LẠT

KHOA TOÁN - TIN HỌC

TIỂU LUẬN MÔN HỌC

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT 1 (TN2201D)

CẤU TRÚC CÂY NHỊ PHÂN & CÂY AVL (Adelson-Velsky & Landis)

GIẢNG VIÊN HƯỚNG DẪN:
TS. DƯƠNG VĂN HẢI

SINH VIÊN THỰC HIỆN:
TRẦN DUY THANH - 2015830

12, 2021

TÓM TẮT

Lưu trữ (store) và **truy vấn** (query) là 2 thao tác cơ bản và cũng là phổ biến trong các tác vụ máy tính ở mọi cấp độ kể từ xử lý các *tập lệnh* (instruction) ở tầng CPU, xử lý các *tiểu trình* (process) ở *tầng hệ điều hành* (operation system), và xử lý các *thủ tục* (procedure) ở tầng *ứng dụng* (application). Muốn truy vấn hiệu quả, ta cần lưu trữ có cấu trúc phù hợp với loại truy vấn mà ta cần sau đó.

Các giải thuật sắp xếp quen thuộc như: *sắp xếp chèn* (insert sort), *sắp xếp lựa chọn* (selection sort), *sắp xếp nổi bọt* (bubble sort), *sắp xếp trộn* (merge sort), *sắp xếp nhanh* (quick sort) dù có nhiều điểm khác nhau cơ bản thì vẫn đặt trên cơ sở tạo ra 1 cấu trúc dữ liệu đã sắp xếp theo 1 chiều duy nhất (một dãy giá trị). Ta bắt đầu thấy thấp thoáng bóng dáng của cấu trúc cây trong giải thuật *sắp xếp đống* (heap sort). Theo đó, trong phạm vi tiểu luận này, tôi xin trình bày chuỗi các ý tưởng cải tiến từng bước từ cấu trúc cây đến *cây tìm kiếm nhị phân* (binary search tree - BST) cùng với 1 trong số các biến thể cải tiến của nó: *cây AVL* (AVL tree)

Trong tiểu luận này, tôi thực thi giải thuật bằng ngôn ngữ python phiên bản 3.8.12. Độc giả quan tâm cũng có thể tải về toàn bộ mã nguồn của thuật toán này từ tài khoản github của tôi (xem phần phụ lục cuối tiểu luận)

Mục lục

Chương 1

Cây tìm kiếm nhị phân (binary search tree)

1.1 Ý tưởng

Đâu là cách sắp xếp chỉ dẫn cho ta biết **hướng** tìm kiếm phần tử mong muốn cho trước?

Với câu hỏi này, ta dễ dàng liên tưởng đến một mạng lưới, một thứ mà toán học gọi là *đồ thị* (graph).

Đồ thị là cấu trúc toán học được xây dựng trên 2 khái niệm là *nút* (node) gắn với một giá trị và *cạnh* (edge)là đường nối biểu diễn quan hệ theo một tiêu chí nào đó giữa 2 nút.

Trong đồ thị không có chu trình, nếu trong số các nút chỉ có 1 cạnh, ta chỉ định 1 nút gọi là *nút gốc* (root node) thì ta sẽ có cấu trúc mới gọi là **cây** (tree). Theo đó, các nút có 1 cạnh còn lại gọi là *nút lá* (leaf node). Các nút có từ 2 cạnh trở lên gọi là *nút trong* (internal node). Xét theo chiều đi từ nút gốc đến nút lá, nút nằm liền sau sẽ là *nút con* (child node) của nút liền trước nó, và nút liền trước đó sẽ được gọi là *nút cha* (parent node). Ngoại trừ nút gốc, mỗi nút chỉ có duy nhất 1 nút cha. Mỗi nút cha có thể có nhiều nút con. Ngoài ra, theo thói quen,

một cạnh trong cấu trúc cây còn được gọi là *nhánh* (branch).

Một cách tương đối, một nút bất kỳ đều có thể xem như là nút gốc của tất cả các nút con cháu của nó, và tính từ nút đó ta có một cấu trúc gọi là **cây con**. Giả sử nút đang xét là nút A, ta sẽ gọi cây con đó là cây con A hoặc gọi tắt là cây A.

Trong trường hợp ta ràng buộc nút cha chỉ có tối đa 2 nút con thì cây này được gọi là *cây nhị phân* (binary tree). Khi biểu diễn cây trên giấy theo chiều dọc, ta thường quen gọi 2 nút con này theo cảm quan trực tiếp là *nút trái* (left node) và *nút phải* (right node).

Nếu ta bổ sung thêm ràng buộc rằng nút trái phải có giá trị nhỏ hơn nút cha, nút phải phải có giá trị lớn hơn nút cha thì cây trở thành *cây tìm kiếm nhị phân* (binary search tree). Chữ "tìm kiếm" ở đây chính là mang ngụ ý "chỉ dẫn" trả lời cho câu hỏi ý tưởng xuất phát đầu tiên của chúng ta. Và ta liên tưởng ngay được rằng, ràng buộc ta vừa thêm vào, chính là tiêu chí để chọn "hướng" - chính là chọn nhánh con.

Ta thực thi định nghĩa một nút trong python như sau:

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

Giải thuật 1.1: Định nghĩa nút

Ngoài ra để phục vụ cho các *luận lý* (logic) phức tạp hơn, ta sẽ bổ sung một số *phương thức* (method) kiểm tra nút con trái/phải như sau:

```
class Node:
    ...

    def has_no_child(self):
        return (self.left is None) and (self.right is None)

    def has_only_left(self):
        return (self.left is not None) and (self.right is None)

```

```
    def has_only_right(self):
        return (self.left is None) and (self.right is not None)
```

Giải thuật 1.2: Bổ sung các *phương thức* (method) kiểm tra nút con

Định nghĩa một *cây* (tree)

```
class Tree:
    def __init__(self, root = None):
        self.root = root
```

Giải thuật 1.3: Định nghĩa cây

1.2 Tìm kiếm nút con

Ý tưởng:

- Nếu cây không có nút nào, dĩ nhiên ta k tìm thấy nút.
- Đi từ nút gốc, so sánh giá trị cần tìm với giá trị của nút hiện tại, nếu nhỏ hơn (hoặc lớn hơn) thì ta sẽ tiếp tục tìm kiếm ở nút trái (hoặc nút phải)

```
class Tree:
        ...

    def exist_recursive(self, x):
        return self.__exist_recursive(self.root, x)

    def __exist_recursive(self, node, x):
        if node.val == x:
            return True
        if node.val < x:
            if node.right:
                return self.__exist_recursive(node.right, x)
            return False
        if node.left:
            return self.__exist_recursive(node.left, x)
        return False
```

Giải thuật 1.4: Tìm kiếm nút con sử dụng đệ quy

```
class Tree:
        ...

    def exist_loop(self, x):
        node = self.root
        while True:
            if not node:
                return False
            if node.val == x:
                return True
            if node.val < x:
                node = node.right
                continue
            node = node.left
```

Giải thuật 1.5: Tìm kiếm nút con sử dụng vòng lặp

1.3 Chèn nút

Ý tưởng:

- Nếu cây chưa có nút nào, thì nút chèn vào sẽ làm nút gốc.
- Nếu một nút nào đó trong cây đã chứa giá trị cần chèn này thì ta không tạo thêm nút mới cho giá trị đó nữa.
- Thực thi giải thuật tìm kiếm nút bắt đầu từ nút gốc, nếu không tìm thấy thì thực hiện chèn nút mới làm nút con của nút lá đang xét.
- Nếu nút thêm vào có giá trị nhỏ hơn (hoặc lớn hơn) nút đang xét thì tạo nút con trái (hoặc nút con phải) cho nó.

```python
class Tree:
    ...

    def insert_recursive(self, x):
        if self.root is None:
            self.root = Node(x)
            return
        self.__insert_recursive(self.root, x)

    def __insert_recursive(self, node, x):
        if node.val == x:
            return
        if node.val < x:
            if node.right is None:
                node.right = Node(x)
                return
            self.__insert_recursive(node.right, x)
            return
        # insert to left
        if node.left is None:
            node.left = Node(x)
        self.__insert_recursive(node.left, x)
```

Giải thuật 1.6: Chèn nút vào cây sử dụng đệ quy

```python
class Tree:
    ...

    def insert_loop(self, x):
        if self.root is None:
            self.root = Node(x)
            return

        node = self.root
        while True:
            if node.val > x:
                if node.left:
                    node = node.left
                    continue
                node.left = Node(x)
                return

```

```
            if node.right:
                node = node.right
                continue
            node.right = Node(x)
            return

```

Giải thuật 1.7: Chèn nút con vào cây sử dụng vòng lặp

1.4 Loại bỏ nút

Ý tưởng

- Nếu nút cần loại bỏ không tồn tại trong cây, ta không cần phải làm gì cả.
- Nếu cây chỉ có 1 mình nút gốc, và đó là nút cần loại bỏ, thì ta gán trường *root* của cây bằng *None*.
- Nếu nút cần loại bỏ không có nút con, ta chỉ cần gắn giá trị *None* cho tham chiếu $left$ hoặc *right* của nút cha tương ứng với nút cần loại bỏ.
- Nếu nút cần loại bỏ chỉ có 1 nút con là nút trái (hoặc nút phải) thì chỉ cần thay thế nút con đó bằng nút trái (hoặc nút phải) của nó
- Nếu nút cần loại bỏ E có 2 nút con, thì ta cần tìm nút có giá trị nhỏ nhất trong toàn bộ cây con E, giả sử đó là nút I thì ta tiến hành gán giá trị của nút E bằng giá trị của nút I, sau đó đệ quy thực hiện tìm xóa nút I trong cây con E này.

```
class Tree:
        ...

    def get_min_of_node(self, node):
        while node.left is not None:
            node = node.left
        return node.val
```

Giải thuật 1.8: Tìm kiếm nút có giá trị bé nhất trong cây con của nút

```python
class Tree:
    ...

    def delete(self, x):
        self.root = self.__delete(self.root, x)

    def __delete(self, node, x):
        if node is None:
            return None
        if node.val < x:
            node.right = self.__delete(node.right, x)
            return node
        if node.val > x:
            node.left = self.__delete(node.left, x)
            return node
        if node.val == x:
            if node.has_no_child():
                return None
            # Handle case: node has a single child
            if node.has_only_left():
                return node.left
            if node.has_only_right():
                return node.right
            # handle case: node has 2 children
            #          ____C___
            #         /         \
            #        B           E <---- delete this node
            #                   /   \
            #                  D     K
            #                      /   \
            #                     I     L
            #                      \
            #                       J
            # step 1: replace E by I
            # step 2: delete I
            min_value = self.get_min_of_node(node.right)
            node.val = min_value
            node.right = self.__delete(node.right, min_value)
            return node
        return node
```

Giải thuật 1.9: Loại bỏ nút sử dụng đệ quy

1.5 Cây cân bằng

Để tiện diễn giải, trước hết, ta sẽ đưa ra khái niệm *độ cao của nút*. **Độ cao** của nút là số nút tối đa trên đường đi từ nút đang xét đến *nút lá* (leaf node) của nó. **Độ cao của cây** là độ cao của nút gốc.

Một bổ sung khác, đi kèm vs khái niệm độ cao, sẽ là khái niệm hệ số cân bằng của nút. **Hệ số cân bằng** b là hiệu số độ cao giữa độ cao nút con nhánh trái h_l và độ cao nút con phải h_r của nút đang xét. Tức là

$$b = h_l - h_r$$

Hệ số cân bằng của cây là hệ số cân bằng của nút gốc

Giả sử, với giải thuật trên, ta thêm nút vào cây theo thứ tự $\{3, 4, 5, 7, 9, 11\}$, ta thu được cây chỉ gồm các nút có 1 nhánh duy nhất liên tục về phải như sau:

```
#
#           3
#            \
#             4
#              \
#               5
#                \
#                 7
#                  \
#                   9
#                    \
#                     11
#
```

Tương ứng, độ cao h và hệ số cân bằng b của từng nút như sau:

```
#
#          3 (h=5, b=-5)
#           \
#            4 (h=4, b=-4)
#             \
#              5 (h=3, b=-3)
#               \
#                7 (h=2, b=-2)
#                 \
#                  9 (h=1, b=-1)
#                   \
#                    11 (h=0, b=0)
#
```

Trong trường hợp này, cấu trúc cây không khác gì cấu trúc dãy 1 chiều. Khi đó, độ phức tạp của thuật toán tìm kiếm nút trong cây rơi vào trường hợp tệ nhất là $O(n)$

Từ đây, ta nghĩ ngay đến việc cải tiến cây này sao cho chia đều các nút về 2 bên nhằm giảm độ cao h của cây. Tương ứng ta cũng sẽ tính được độ ca và hệ số cân bằng tương ứng cho từng nút như sau:

```
#
#                ____________5__   (h=3, b=-1)
#              /                  \
#            4 (h=1, b=1)          7 (h=2, b=-2)
#          /                        \
#         3 (h=0, b=0)               9 (h=1, b=-1)
#                                     \
#                                      11 (h=0, b=0)
#
```

Để ý một chút, ta thấy nhánh bên phải vẫn còn có thể chia tiếp được

```
#
#                ____________5__ (h=3, b=0)
#              /                  \
#            4 (h=1, b=1)        __9_________ (h=1, b=0)
#          /                   /              \
#         3 (h=0, b=0)       7 (h=0, b=0)      11 (h=0, b=0)
#
```

1.6 Một vài nhận xét rút ra

Nút lá luôn có $h = 0, b = 0$

Một nút A nào đó có $|b| > 1$, ta luôn có thể tái cấu trúc cây con A để giảm độ cao của cây con A nói chung. Từ đây ta bổ sung thêm định nghĩa: **nút cân bằng** là nút có $|b| \leq 1$.

Ý tưởng: cây có mọi nút đều là nút cân bằng thì cây sẽ là cây cân bằng. Và thực vậy, đây chính là tiêu chuẩn cân bằng theo Adelson-Velsky & Landis mà ta sẽ thấy trong cây AVL trong chương tiếp theo.

Chương 2

Cây AVL - Adelson-Velsky & Landis

2.1 Ưu thế của cây tìm kiếm nhị phân cân bằng

Cây AVL là cây tìm kiếm nhị phân có mọi nút đều là nút cân bằng.

Một cách không chuẩn tắc, ta có thể hiểu khái niệm các *tầng* trong cây thông qua ví dụ sau:

```
#
#          ____C___                (tầng 1)
#         /         \
#        B           E             (tầng 2)
#                  /   \
#                 D     K          (tầng 3)
#                     /   \
#                    I     L       (tầng 4)
#                     \
#                      J           (tầng 5)
#
```

Để có được cây cân bằng theo tiêu chuẩn AVL, mỗi khi thêm vào 1 nút con, thuật toán có xu hướng tái cấu trúc cây đã có sao cho cố gắng lấp đầy từng tầng trước

khi bổ sung thêm tầng nút con mới. Theo đó, giả sử với cây có độ cao h thì:

Tầng 1 có tối đa $2^0 = 1$ nút - là nút gốc.

Tầng 2 có tối đa $2^1 = 2$ nút,

Tầng 3 có tối đa $2^2 = 4$ nút.

Cây cân bằng có độ cao h thì sẽ có h tầng, và ở tầng dưới cùng, sẽ có tối đa 2^h nút, tối thiểu 1 nút.

Vậy số nút tối đa mà cây cân bằng có độ cao h có thể có là:

$$\begin{aligned} n_{max} &= 2^0 + 2^1 + ... + 2^{h-1} + 2^h \\ \Leftrightarrow 2n_{max} &= 2^1 + 2^2 + ... + 2^h + 2^{h+1} \\ \Leftrightarrow 2n_{max} &= 2^0 + 2^1 + 2^2 + ... + 2^h + 2^{h+1} - 2^0 \\ \Leftrightarrow 2n_{max} &= n_{max} + 2^{h+1} - 2^0 \\ \Leftrightarrow n_{max} &= 2^{h+1} - 1 \end{aligned}$$

Số nút tối thiểu mà cây cân bằng có độ cao h có thể có là:

$$\begin{aligned} n_{min} &= 2^0 + 2^1 + ... + 2^{h-1} + 1 \\ \Leftrightarrow 2n_{min} &= 2^1 + 2^2 + ... + 2^h + 2 \\ \Leftrightarrow 2n_{min} &= 2^0 + 2^1 + 2^2 + ... + 2^h + 2 - 2^0 \\ \Leftrightarrow 2n_{min} &= 2^0 + 2^1 + 2^2 + ... + 2^h + 1 \\ \Leftrightarrow 2n_{min} &= n_{min} + 2^h \\ \Leftrightarrow n_{min} &= 2^h \end{aligned}$$

Số nút thực tế n mà cây có thể có thỏa:

$$\begin{aligned} n_{min} \leq & n \leq n_{max} \\ 2^h \leq & n \leq 2^{h+1} - 1 \end{aligned}$$

Từ đó suy ra:

$$log_2 n \geq h \geq log_2(n+1) - 1$$

Điều tuyệt vời nhất rút ra ở đây, đó là, với cây cân bằng AVL, khi số nút tăng tuyến tính thì độ cao của cây chỉ tăng theo hàm log_2. Tức là độ phức tạp của thuật toán tìm kiếm nút trên cây chỉ là $O(log_2 n)$

Định nghĩa nút và cây trong trường hợp cây AVL

```
class Node():
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1
```

Giải thuật 2.1: Định nghĩa lớp Node cho cây AVL

```
class AVLTree():
    def __init__(self, root=None):
        self.root = root
```

Giải thuật 2.2: Định nghĩa lớp AVLTree

Sau khi đã thử qua nhiều cách thực thi, để gọn và thuận tiện cho việt cân bằng cây sau này, tôi sẽ viết các phương thức tính độ cao, hệ số cân bằng của nút trong AVLTreen thay vì trong Node. Cụ thể

```
class AVLTree():

    ...

    def height_of(self, node):
        if node is None:
            return 0
        return node.height


    def cal_balance(self, node):
        if node is None:
            return 0
```

```
        left_height = self.height_of(node.left)
        right_height = self.height_of(node.right)
        return left_height - right_height


    def find_min_value_node(self, node):
        if node is None or node.left is None:
            return node
        return self.find_min_value_node(node.left)
```

Giải thuật 2.3: Bổ sung một vài phương thức tiện dụng sau này

Điểm mấu chốt trong giải thuật cây AVL chính là nó thường xuyên kiểm tra lại hệ số cân bằng của nút có nút con vừa bị loại bỏ hoặc chèn thêm để tái cân bằng cây.

2.2 Tái cân bằng cây

Ta dừng lại quan sát một chút để cùng xây dựng ý tưởng.

- Quá trình xây dựng BST đảm bảo mọi cây con cũng là một cây BST hoàn chỉnh. Một BST cân bằng thì các cây con của nó cũng đều là các BST cân bằng. Vì lý do này, nên ta mới có thể áp dụng đệ quy các phương thức của lớp Tree như đã thấy trong chương 1.
- Khi thêm hoặc loại bỏ nút thì cây con nhỏ nhất chứa nút vừa được thêm hay loại bỏ sẽ là cây con mất cân bằng trước tiên. Việc tái cân bằng cây con này sẽ ảnh hưởng lên độ cao của nó, từ đó có thể gián tiếp ảnh hưởng lên hệ số cân bằng của nút cha gắn với nó, và cứ thế, ta cần thực hiện một chuỗi các thao tác tái cân bằng đi ngược lên cây con của nút cha, nút ông,... cây con của các nút "*tổ tiên*" (ancestor).

Điều này gợi ý cho chúng ta xây dựng giải thuật xử lý các tình huống phát sinh ở cây con đơn giản, sau đó thêm một vài luận lý để xử lý một vài trường hợp biên và có thể áp dụng đệ quy để truy ngược và tái cân bằng cây con của các nút tổ tiên, tiến tới tái cân bằng toàn bộ cây.

Từ công thức tính hệ số cân bằng $h = h_{left} - h_{right}$ như ta đã biết từ chương 1, ta dễ dàng nghĩ ngay được rằng, muốn tái cân bằng, ta phải san bớt nút ở cây con trái sang cây con phải hoặc ngược lại để điều chỉnh h sao cho $|h| \leq 1$

Quan sát bằng mắt thường, ta sẽ thấy quá trình san bớt nút từ cây con trái sang cây con phải sẽ giống như quá trình **xoay phải** (right rotation) và ngược lại sẽ là quá trình **xoay trái**. Ta xem minh họa dưới đây

```
#
#       Bước 1: ta tách nút              Bước 2: Gán T2 làm
#       con phải T2 ra,                  nút con trái của z
#       chuyển vai trò nút
#       gốc từ z về cho y
#                             xoay phải
#         |------(Bước 1)-------------------(Bước 2)-->>|
#
#          z                    y ----- z                  y
#         / \                  /         \               / \
#        y   T3              T1           T3            T1  z
#       / \                                               / \
#      T1  T2                       T2                   T2  T3
#
#                             xoay trái
#         |<<-----------------------------------------|
#
#       Bước 2: Gắn T2 làm               Bước 1: Tách nút con
#       nút con phải của y               trái T2 của z ra,
#                                        chuyển vai trò nút
#                                        gốc từ y về cho z.
#
```

Trong đó:

- $T1$, $T2$ là cây con của cây y
- $T3$ là cây con của cây z

Vì đây đều là các BST, vậy nên ta luôn có tính chất sau

$$values(T1) < value(y) < values(T2) < value(z) < values(T3)$$

Với $value(y)$ chính là giá trị của nút y. $values(T1)$ chính là các giá trị của tất cả nút trong cây $T1$. Tương tự với $value(z)$, $values(T2)$, $values(T3)$.

Ta bổ sung 2 phương thức thực hiện xoay trái, xoay phải ở trên như sau:

```
    class AVLTree():
        ...
        def rotate_left(self, z):
            z = y.right
            T2 = z.left
            z.left = y
            y.right = T2

        def rotate_right(self, z):
            y = z.left
            T2 = y.right
            y.right = z
            z.left = T2
```

Giải thuật 2.4: Xoay trái, xoay phải

Giải thuật trên chỉ mới thực hiện phép xoay như ý tưởng đã trình bày. Để có thể áp dụng đệ quy sau này, ta sẽ cần thực hiện 2 thao tác phụ nữa chính là:

- Tính toán lại độ cao của nút gốc cũ và nút gốc mới trong cây con đang xét
- Trả về nút gốc mới của cây con đang xét

```
    class AVLTree():
        ...
        def rotate_left(self, z):
            z = y.right
            T2 = z.left
            z.left = y
            y.right = T2

            y.height = 1 + max(
                self.height_of(y.left),
                self.height_of(y.right)
            )

```

```
            z.height = 1 + max(
                self.height_of(z.left),
                self.height_of(z.right)
            )
            return z

        def rotate_right(self, z):
            y = z.left
            T2 = y.right
            y.right = z
            z.left = T2

            z.height = 1 + max(
                self.height_of(z.left),
                self.height_of(z.right)
            )
            y.height = 1 + max(
                self.height_of(y.left),
                self.height_of(y.right)
            )
            return y
```

Giải thuật 2.5: Phiên bản thực thi xoay phải, xoay trái đầy đủ

Sử dụng 2 phương thức nòng cốt này, ta xét tiếp 4 tình huống mất cân bằng có thể xảy ra khi chèn hoặc loại bỏ một nút ra khỏi cây (hoặc cây con)

2.2.1 Tái cân bằng trái-trái

```
#
#          z                                       y
#         / \                                    /   \
#        y   T4      xoay phải cây z            x      z
#       / \          ---------------->         /  \    /  \
#      x   T3                                 T1  T2  T3  T4
#     / \
#   T1   T2
#
```

Cây z bị lệch ở nút con trái x của nút con trái y (nên mới gọi là *trái-trái* (left-left)).

Thực hiện xoay phải cây z, để giảm dần độ cao bên trái và tăng độ cao bên phải. Thường thì ta chỉ cần xoay 1 lần bởi ta sẽ luôn kiểm tra để tái cân bằng ngay sau khi thêm hoặc bớt 1 nút con chứ không phải để tích lũy chênh lệch nhiều.

2.2.2 Tái cân bằng trái-phải

```
#
#       z                                   z                                   x
#      / \                                 /   \                               /  \
#     y   T4                              x     T4                            y      z
#    / \       xoay trái y               /  \       xoay phải z              / \    / \
# T1   x      ------------> y            T3   ------------> T1   T2 T3   T4
#     / \                  / \
#    T2  T3              T1   T2
#
```

Cây z bị lệch ở nút con phải x của nút con trái y (nên mới gọi là *trái-phải* (left-right)).

Vẫn quay lại với cách tư duy quen thuộc khi giải toán, ta cố gắng đưa trường hợp này về trường hợp đã biết rồi áp dụng giải thuật đã có. Và ở đây, ta thực hiện xoay trái y để đưa cây z về trường hợp lệch trái-trái, sau đó áp dụng xoay phải như đã biết.

2.2.3 Tái cân bằng phải-phải

```
#
#     z                                 y
#    /  \                              /   \
#   T1   y       xoay trái z          z      x
#       /  \     ------------>       / \    / \
#      T2   x                       T1  T2 T3  T4
#          / \
#         T3  T4
#
```

Cây z bị lệch ở nút con phải x của nút con phải y (nên mới gọi là *phải-phải* (right-right)).

Ngược lại với trường hợp trái trái, ở đây ta chỉ cần xoay trái để giảm dần độ cao bên trái và tăng độ cao bên phải. Thường thì ta cũng chỉ cần xoay 1 lần.

2.2.4 Tái cân bằng phải-trái

```
#
#     z                          z                           x
#    / \                        / \                        /  \
# T1   y     xoay phải y     T1   x     xoay trái z      z      y
#     / \    ---------->        /  \    ---------->     / \    / \
#    x   T4                    T2   y                 T1  T2  T3  T4
#   / \                           /  \
# T2   T3                        T3   T4
#
```

Cây z bị lệch ở nút con phải x của nút con phải y (nên mới gọi là *phải-phải* (right-right)).

Vâng, ở đây ta thực hiện xoy phải y để về trường hợp lệch phải-phải rồi sau đó xoay trái z như đã biết.

```
        # Left Left
        if balance > 1 and val < node.left.val:
            return self.rotate_right(node)
        # Right Right
        if balance < -1 and val > node.right.val:
            return self.rotate_left(node)
        # Left Right
        if balance > 1 and val > node.left.val:
            node.left = self.rotate_left(node.left)
            return self.rotate_right(node)
        # Right Left
        if balance < -1 and val < node.right.val:
            node.right = self.rotate_right(node.right)
            return self.rotate_left(node)
```

Giải thuật 2.6: Xử lý các tình huống cây con bị lệch

Vì 2 quá trình chèn và loại bỏ nút có một số điểm khác nhau trong thao tác thực hiện nên ta sẽ tùy biến đoạn mã trên chút xíu để nhét vào phương thức chèn và phương thức loại bỏ nút tương ứng.

```
class AVLTree():
    ...
    def insert(self, val):
        self.root = self.__insert(self.root, val)

    def __insert(self, node, val):
        if node is None:
            return Node(val)
        elif val == node.val:
            return node
        elif val < node.val:
            node.left = self.__insert(node.left, val)
        else:
            node.right = self.__insert(node.right, val)
        node.height = 1 + max(
            self.height_of(node.left),
            self.height_of(node.right)
        )
        balance = self.cal_balance(node)

        # Left Left
        if balance > 1 and val < node.left.val:
            return self.rotate_right(node)
        # Right Right
        if balance < -1 and val > node.right.val:
            return self.rotate_left(node)
        # Left Right
        if balance > 1 and val > node.left.val:
            node.left = self.rotate_left(node.left)
            return self.rotate_right(node)
        # Right Left
        if balance < -1 and val < node.right.val:
            node.right = self.rotate_right(node.right)
            return self.rotate_left(node)

        return node
```

Giải thuật 2.7: Chèn nút vào cây AVL

```
    class AVLTree():

        ...

    def delete(self, val):
        self.root = self.__delete(self.root, val)

    def __delete(self, node, val):

        if node is None:
            return node
        elif val < node.val:
            node.left = self.__delete(node.left, val)
        elif val > node.val:
            node.right = self.__delete(node.right, val)
        else:
            if node.left is None:
                temp = node.right
                node = None
                return temp

            elif node.right is None:
                temp = node.left
                node = None
                return temp

            temp = self.find_min_value_node(node.right)
            node.val = temp.val
            node.right = self.__delete(node.right, temp.val)

        if node is None:
            return node

        node.height = 1 + max(
            self.height_of(node.left),
            self.height_of(node.right)
        )

        balance = self.cal_balance(node)


```

```
        # Left Left
        if balance > 1 and self.cal_balance(node.left) >= 0:
            return self.rotate_right(node)

        # Right Right
        if balance < -1 and self.cal_balance(node.right) <= 0:
            return self.rotate_left(node)

        # Left Right
        if balance > 1 and self.cal_balance(node.left) < 0:
            node.left = self.rotate_left(node.left)
            return self.rotate_right(node)

        # Right Left
        if balance < -1 and self.cal_balance(node.right) > 0:
            node.right = self.rotate_right(node.right)
            return self.rotate_left(node)

        return node
```

Giải thuật 2.8: Loại bỏ nút khỏi cây AVL

2.3 Kết luận và hướng phát triển

Giải thuật trên đây yêu cầu mỗi nút phải có một trường lưu trữ độ cao của nút đó, và thường là kiểu số nguyên 8 bit (tức là chiều cao tối đa là 127 hoặc 255, quá nhiều cho các ứng dụng thực tế). Ta có thể cải tiến giải thuật trên sao cho chỉ dùng đúng 1 bit để đánh dấu nút có cân bằng hay không. Và đó là *cây đỏ-đen* (red-black tree) mà ta sẽ đề cập trong một dịp khác

Chương 3

Phụ lục

3.1 Tham khảo

1. Giáo trình Cấu trúc dữ liệu và giải thuật 1, trường đại học Đà Lạt, Trương Chí Tín.
2. Thực hành cấu trúc dữ liệu và giải thuật 1, trường đại học Đà Lạt, Tạ Thị Thu Phượng
3. https://www.giaithuatlaptrinh.com/?p=1954

3.2 Mã nguồn

- Giải thuật BST:
  https://github.com/tranduythanh/algorithm-in-python/src/bst.py
- Giải thuật cây AVL:
  https://github.com/tranduythanh/algorithm-in-python/src/avl.py
- Bài giải cho một số bài tập trong bài thực hành 10 trong 3.1 nguồn số 2
  https://github.com/tranduythanh/algorithm-in-python/src/probs