Stack

LIFO (LAST IN FIRST OUT) nghĩa là phần tử thêm vào cuối cùng thì sẽ là phần tử đầu tiên bị xóa

VD:



trong mảng có [1,2,3,4]

push(5)

mảng : [1,2,3,4,5]

pop()

mảng : [1,2,3,4]

add = push()

remove = pop()

Peek (xem phần tử cuối) thường là xem phần tử bị xóa đầu tiên

Search (tìm phần tử trong mảng stack)

KHI NÀO SỬ DỤNG ?

* hoàn tác/làm lại trong trình chỉnh sửa
* di chuyển lùi/tiến qua lịch sử trình duyệt
* backtracking algorithms (maze, file directions)
  + - * (thuật toán quay lui (mê cung, hướng dẫn tệp))
* calling functions (call stacks)

QUEUE

Linear data structure

FIFO (FIRST-IN FIRST-OUT) nghĩa là phần tử thêm vào đầu tiên sẽ là phần tử đầu tiên bị xóa

VD:



trong mảng có [1,2,3,4]

offer(5)

mảng : [1,2,3,4,5]

poll()

mảng : [2,3,4,5]

add = enqueue, offer()

remove = dequeue, poll()

Peek, Element (xem phần tử đầu) thường là xem phần tử bị xóa đầu tiên

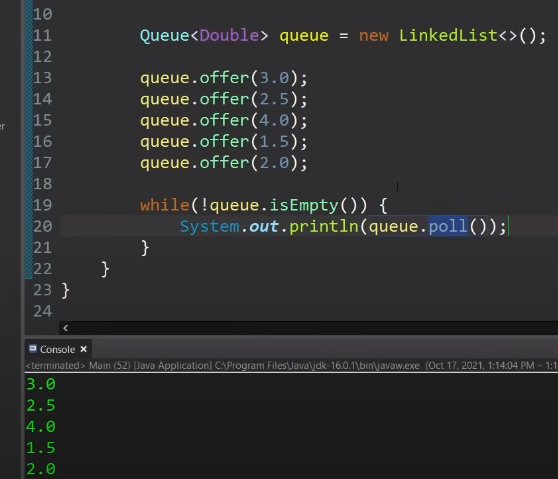
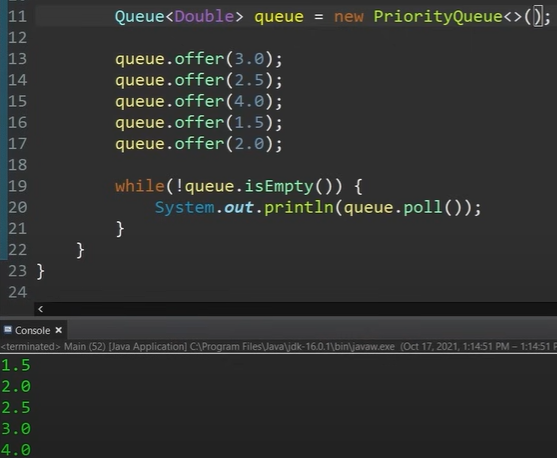
KHI NÀO SỬ DỤNG ?

* Keyboard buffer (các chữ cái sẽ xuất hiện trên màn hình theo thứ tự chúng được nhấn)
* Printer Queue (print jobs should be completed in order)
* Used in Linked Lists, **Priority Queues**, Breadth-first search

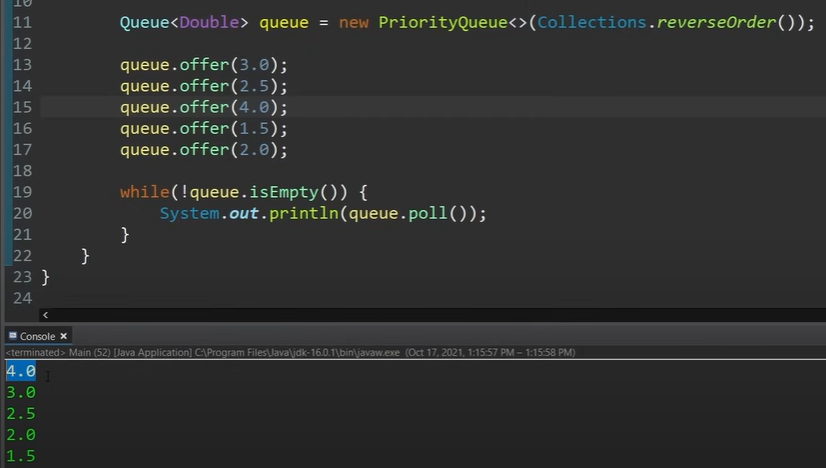
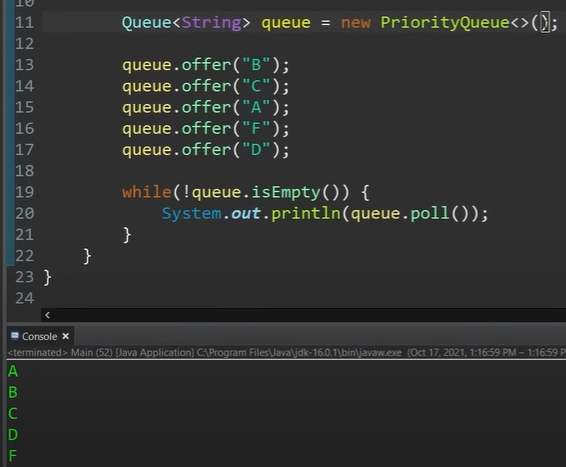
Priority Queues

A FIFO data structure that serves elements with the highest priorities first before elements with lower priorities **(ưu tiên cao -> ưu tiên thấp)**

Vd: khác nhau giữa queue và **Priority Queues**

Hoặc

Linked Lists

Cấu trúc liên kết là tập hợp các nút lưu trữ dữ liệu và liên kết đến các nút khác

Danh sách liên kết là cấu trúc dữ liệu tuyến tính bao gồm các nút, mỗi nút chứa một số thông tin và tham chiếu đến nút khác trong danh sách

Sumary:

LinkedList = stores Nodes in 2 parts (dafta + address)

Nodes are in non-consecutive memony locations

Elements are linked using pointers

* Types of linked lists:
  + Singly-Linked List
  + Doubly-Linked List

A/ Singly-linked lists

Node Node Node

[data | address] -> [data | address] -> [data | address]

B/ Doubly-linked lists

Node Node

[address | data | address] <-> [address | data | address]

Advantages?

1. Dynamic Data Structure (allocates needed memory while running)

2. Insertion and Deletion of Nodes is easy. O(1)

3. No/Low memorny waste

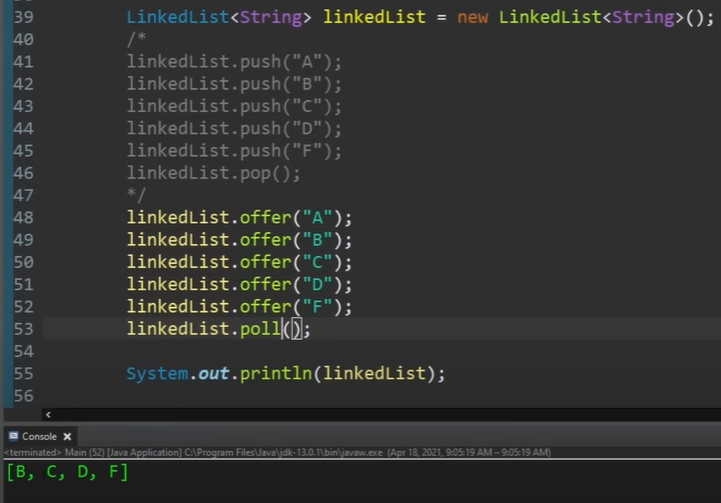
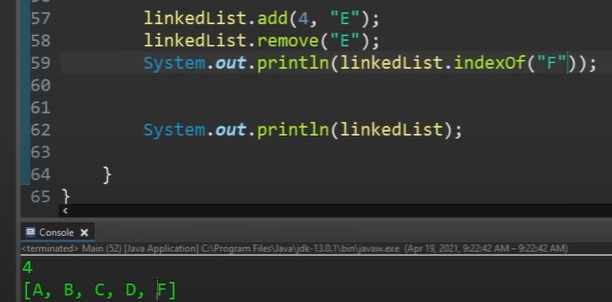
Disadvantages?

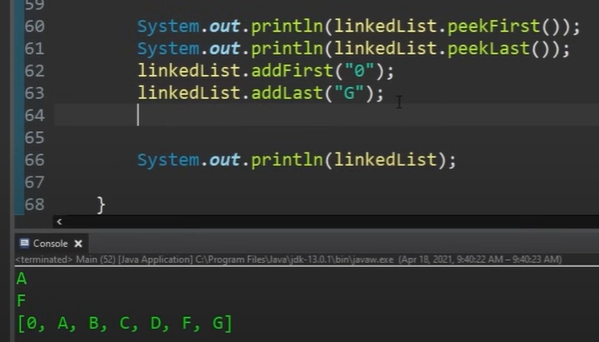
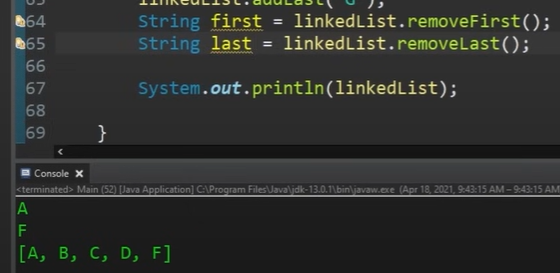
1. Greater memory usage (additional pointer)
2. No random access of elements (no index[i])
3. Accessing/searching elements is more time consuming, O(n)

KHI NÀO SỬ DỤNG ?

1. Implement Stacks/Queues
2. GPS navigation
3. Music playlist

VD:

Recursion (đệ quy)

* Là kĩ thuật giải bài toán bằng bài toán khác
* Bản chất:
  + Chuyển về dạng cơ bản có thể giải trực tiếp
  + Là phép lặp
* **Đệ quy tuyến tính** (linear recursion): chỉ 1 đệ quy được gọi (cho chính nó) trong hàm đệ quy
* **Đệ quy nhị phân** (binary recursion): có chính xác 2 đệ quy được gọi (cho chính nó) trong hàm đệ quy (vd: fibo number)
* **Nhiều đệ quy** (Mutiple recursion): có từ 3 hoặc nhiều hơn đệ quy được gọi (cho chính nó) trong hàm đệ quy (VD: tháp hà nội)
* **Đệ quy gián tiếp**: đệ quy gọi 1 đệ quy khác
* **Đệ quy lồng nhau**: gọi đệ quy lồng với 1 đệ quy khác
* **Đệ quy đuôi** (tail recursion): chỉ có 1 đệ quy được gọi trong cuối của việc thực hiện phương pháp
* **Đệ quy không đuôi** (Non-tail recursion): Cuộc gọi đệ quy không ở cuối phương thức thực hiện.

TREES

**Height**: từ node đến root (đo từ dưới lên)

**Depth**: từ root đến node (đo từ trên xuống)

Ordered Tree

Ordered Tree: nếu đc sắp xếp có một thứ tự tuyến tính có ý nghĩa giữa các nút con của mỗi node

(nghĩa là: xác định các nút con của một node là nút đầu tiên, thứ hai, thứ ba, v.v.) (LEFT -> RIGHT)

Binary Search Tree

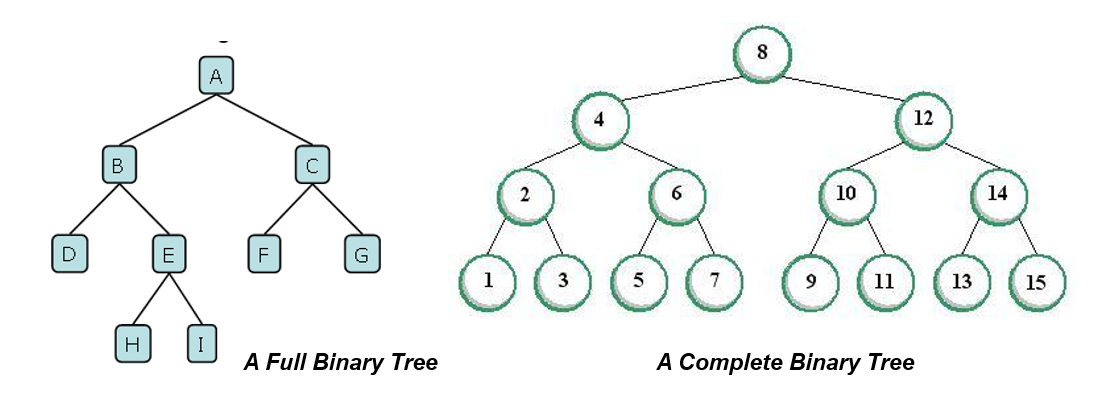
**Binary Search Tree** is a node-based binary tree data structure

(là cấu trúc dữ liệu cây nhị phân dựa trên nút)

* Cây con bên trái của node chỉ chứa các node có khóa nhỏ hơn khóa của node đó.
* Cây con bên phải của node chỉ chứa các node có khóa lớn hơn khóa của node đó.
* Mỗi cây con trái và phải cũng phải là cây tìm kiếm nhị phân.
* **Pre-order**: CHA (ROOT) -> CON CẢ (LEFT) -> CON THỨ (RIGHT)
* **In-order**: CON CẢ (LEFT) -> CHA (ROOT) -> CON THỨ (RIGHT)
* **Post-order**: CON CẢ (LEFT) -> CON THỨ (RIGHT) -> CHA (ROOT)
* **Level-order (Depth-First Search):**
* Từ trên xuống (root -> node)
* Từ trái qua phải (Right -> Left)

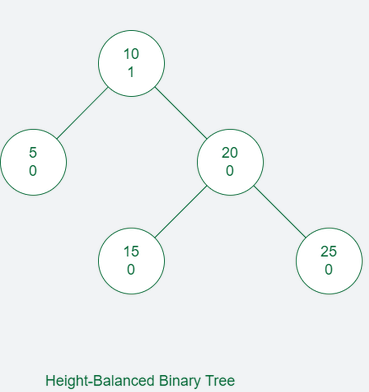
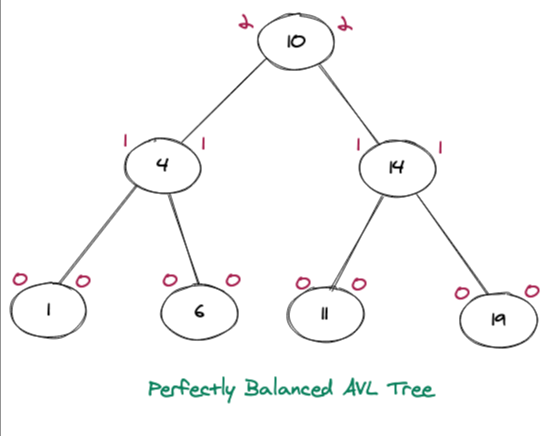
**Type of Binary tree:**

* **Full binary tree: 2h -1**
* **Proper binary tree** (đôi khi là cây nhị phân đầy đủ hoặc 2 cây): mọi node trừ các lá đều có hai nút con.
  + **CT: n is odd (n is node)**
* **Complete binary tree**: all **non-terminal nodes** (là các nút không phải lá của cây phân tích cú pháp) đều có cả hai nút con và tất cả các lá đều có cùng cấp độ.
  + **CT: n = 2h+1 – 1 (h is height in tree)**



Balanced Binary Tree

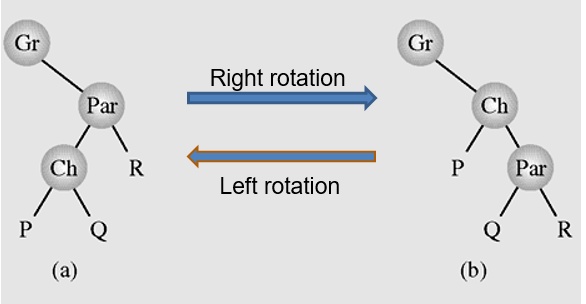
* **Height-balanced**: nếu với mỗi node bên trong p của T, chiều cao của các nút con của p khác nhau nhiều nhất là 1.
* **Perfectly balanced**: nếu nó có chiều cao cân đối và tất cả các lá đều nằm ở một hoặc hai tầng

Rotations on Binary Search Tree (Phép quay)

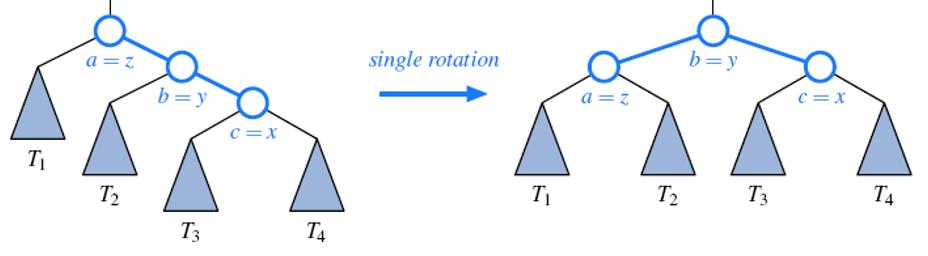
**Right/Left rotation**:

VD: đổi Ch lên thì Par xuống bên phải (Par lớn hơn Ch) và ngược lại (áp dụng quy tắc cây nhỏ trái lớn phải)



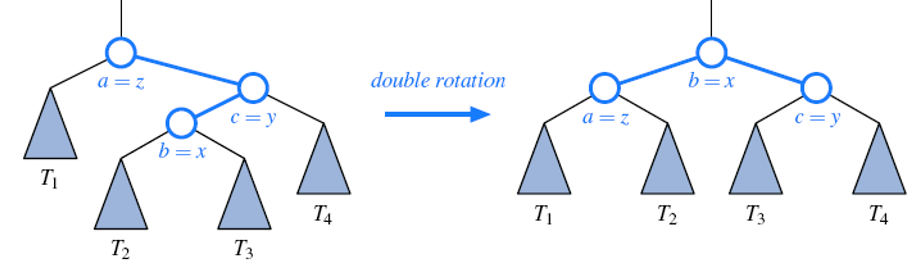
**Single rotations**: lấy thằng ở giữa làm gốc và cân bằng cây

VD: b là giá trị trung bình lấy làm gốc và làm theo quy tắc cây cân bằng



**Double rotations**: lấy thằng có 2 lá làm gốc và cân bằng cây

VD: b ở đây có 2 lá T2 và T3 nên lấy làm gốc sau đó cân bằng cây theo quy tắc



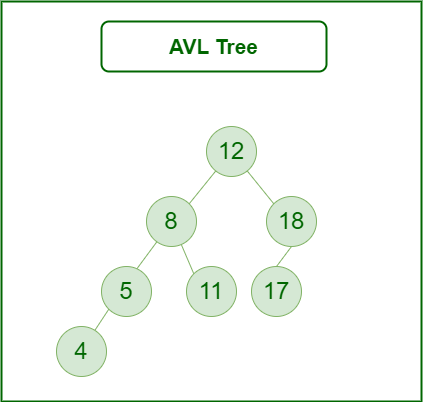
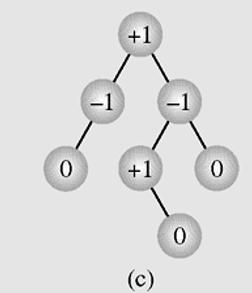
AVL Tree

An AVL tree (by Adelson Velskii, Landis) is a height-balanced binary search tree. (độ cao của cây con trái và cây con phải của mỗi nút đều nhỏ hơn hoặc bằng 1)

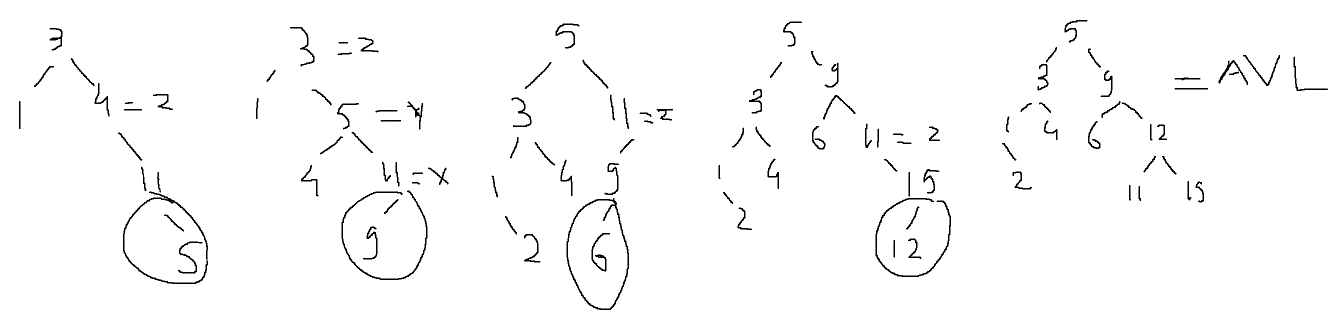
**Công thức tính Balance factor: Height(Right) – Height(Left)**

**Lưu ý: AVL có thể tự cân bằng mỗi khi thêm 1 node vào cây**

Example of AVL Trees:

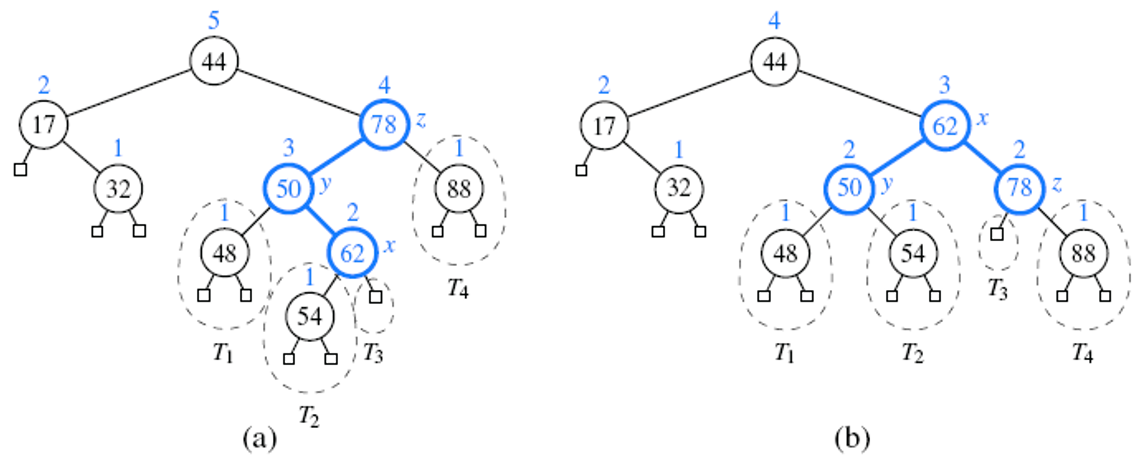
**VD:** **create the AVL tree:** 4, 3, 1, 11, 5, 9, 2, 6, 15, 12. **Give the preorder traversals of the created binary trees.**

****

**Insertion algorithm** in AVL tree:

* Chèn một nút như trong cây tìm kiếm nhị phân.
* Tính toán lại hệ số cân bằng của các nút từ nút được chèn trở lại nút gốc.
  + Nếu không có nút nào không cân bằng (có nghĩa là không có hệ số cân bằng -2 hoặc 2) trên cây thì dừng lại.
  + Nếu tìm thấy nút đầu tiên p có hệ số cân bằng là 2 hoặc -2 thì chúng ta nên xoay p quanh con của nó. Nếu hệ số cân bằng của p và của con của p cùng dấu thì chỉ thực hiện một phép quay duy nhất. Nếu dấu của p và q khác nhau thì nên thực hiện double rotations: đầu tiên q được quay, sau đó là p.
    - Rotation rule: for left unbalane the right rotation should be used and vice versa (Ngược lại).
  + Chỉ nên thực hiện tối đa một vòng quay (đơn hoặc đôi).

VD: Thêm nút 54 vào



Bắt đầu đếm từ nút 54 -> 62 -> 50 (Height left = 3), Height right = 1 (88) => balance factor nút 78 = 1 - 3 = -2

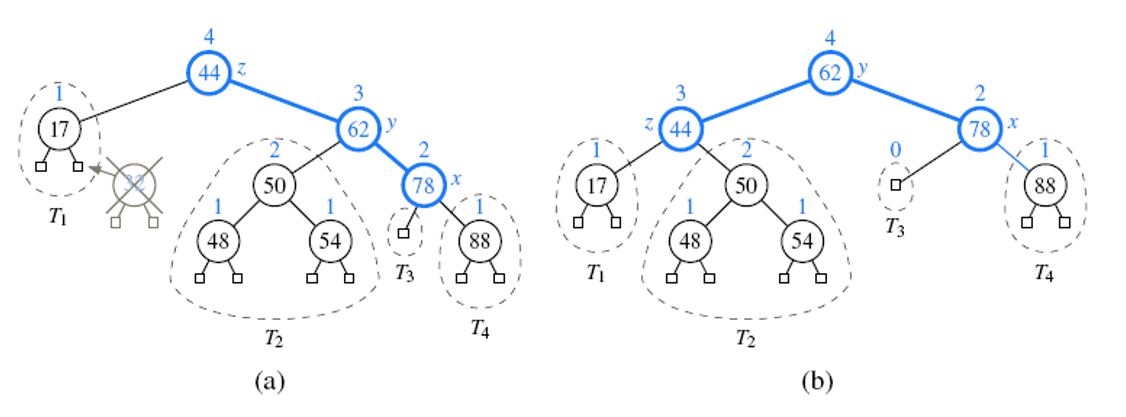
Để cân bằng thì Height right cần = 2 => đẩy 1 nút về bên phải

Áp dụng double rotations cho bên Height left => nút 62 làm gốc rồi áp dụng quy tắc cây ta ra được hình b cân bằng

**Delete algorithm** in AVL tree

Áp dụng như trên chỉ là thay vì thêm thì thành delete

Vd: Xóa nút 32 ở hình a đi



balance factor nút 44 = Height right (88 -> 78 -> 62) – height left (17) = 2

xài double rotation (78 -> 62 -> 44) => 62 làm gốc

áp dụng quy tắc cây ta được hình b cân bằng