Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

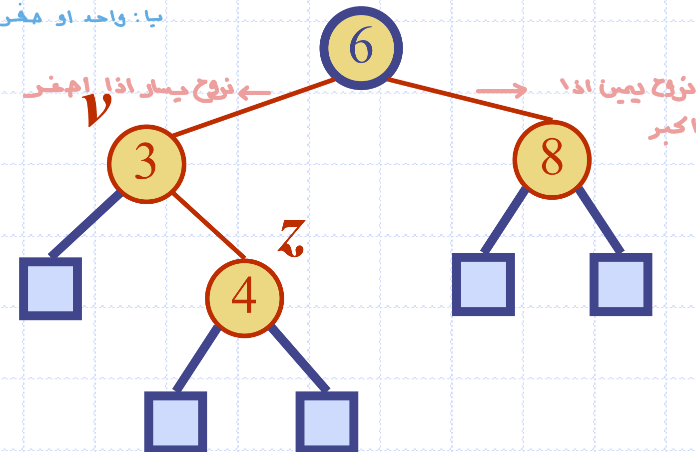
AVL Trees

→ binary search tree + balanced

متوازنه ←

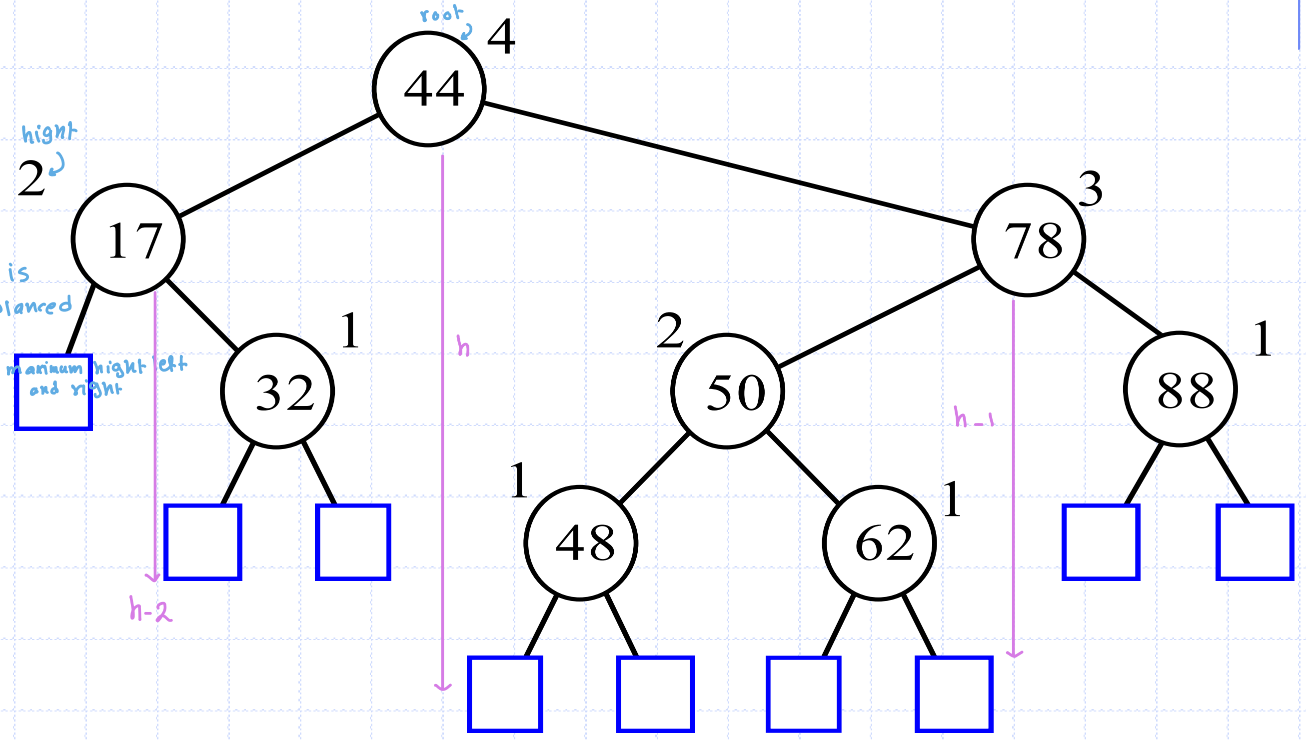
وفرق الطول منطقي

يا: واحد او صفر او سالب واحد



AVL Tree Definition

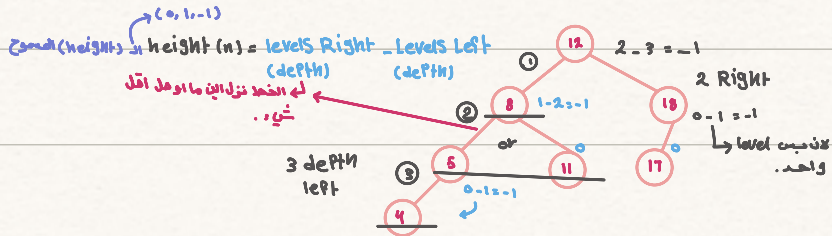
- AVL trees are balanced
- An AVL Tree is a **binary search tree** such that for every internal node v of T, the **heights of the children of v can differ by at most 1**

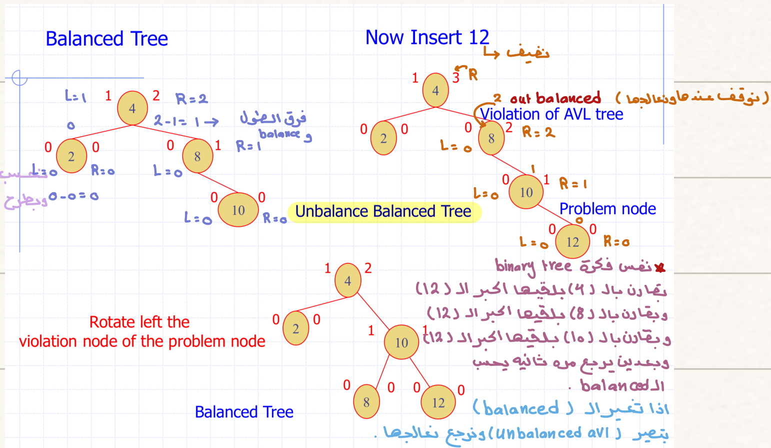


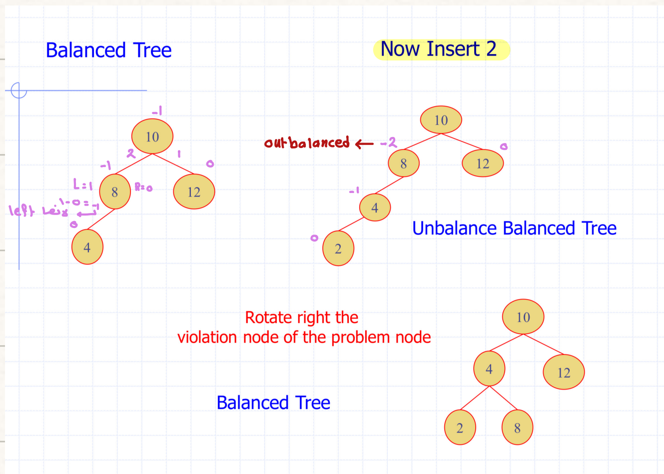
An example of an AVL tree where the heights are shown next to the nodes

height AVL Tree

لـ فرق الطول الـ (right subtree) والـ (left subtree)




Balanced Tree
Now Insert 12
نضيف ←
L=1 R=2
2-1= 1 → فرق الطول و balance
(left and right) hight نحسب الـ ونطرح منهم الفرق
0-0=0
R=1
L=0 R=0
2 out balanced (نوقف عندها ونعالجها)
Violation of AVL tree
R=2
Unbalance Balanced Tree
R=1
Problem node
L=0 R=0
نفس فكرة binary tree
يقارن باد (4) بلقيها الكبر الـ (12)
ويقارن باد (8) بلقيها الكبر الـ (12)
ويقارن باد (10) بلقيها الكبر الـ (12)
وبعدين يرجع مره ثانيه يحسب الـ balanced.
Rotate left the
violation node of the problem node
Balanced Tree
اذا تغير الـ (balanced) بتصير (unbalanced avl) ونرجع نعالجها.

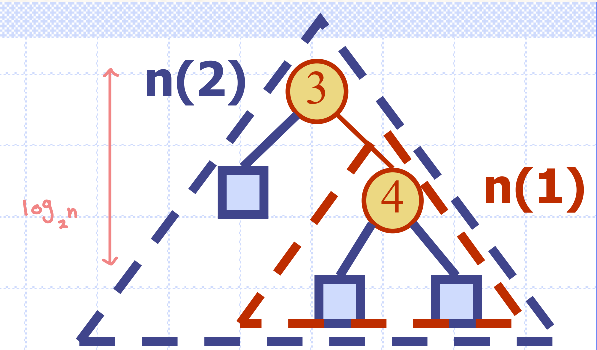

Balanced Tree
Now Insert 2
L=1 R=0
left
out balanced ← -2
Unbalance Balanced Tree
Rotate right the
violation node of the problem node
Balanced Tree

the heights of the children of v can differ by at most 1.

(الفرق بالطول بين الاولاد مايكون اكثر من واحد)

balanced : تتسمى

Height of an AVL Tree



Fact: The height of an AVL tree storing n keys is O(log n).

Proof (by induction): Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.

- We easily see that $n(1) = 1$ and $n(2) = 2$
- For $n > 2$, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.
- That is, $n(h) = 1 + n(h-1) + n(h-2)$
- Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$. So
  $n(h) > 2n(h-2)$, $n(h) > 4n(h-4)$, $n(h) > 8n(n-6)$, … (by induction),
  $n(h) > 2^i n(h-2i)$
- Solving the base case we get: $n(h) > 2^{h/2-1}$
- Taking logarithms: $h < 2\log n(h) + 2$
- Thus the height of an AVL tree is O(log n)

© 2014 Goodrich, Tamassia, Goldwasser

✱ Beast casue: $O(\log n)$

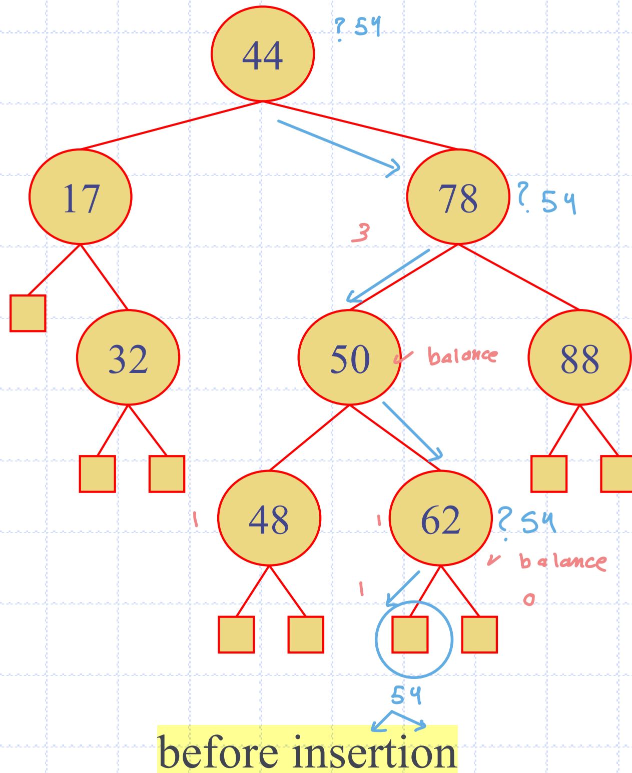
✱ Worst casue: $O(n)$

Insertion

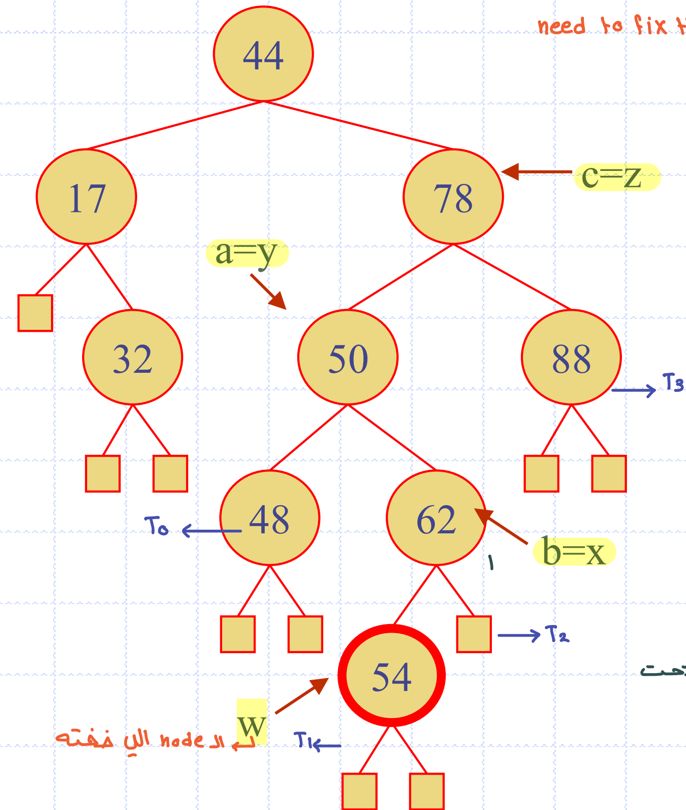
- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example: insert (54) root

✱ Height balanced of AVL tree is violated and need to fix that!

hight $> \log_2 n$



before insertion



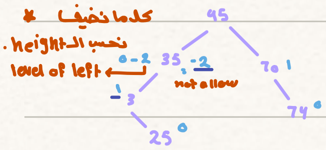
after insertion

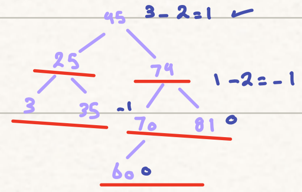
نبدا من الـ node اللي تحت

اللي فوق .

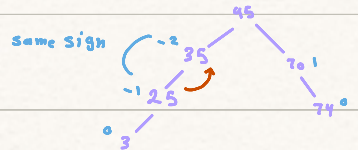
creat an AVL tree by inserting the values:

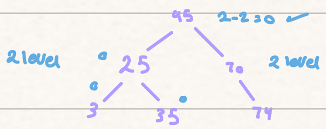
45 , 70 , 35 , 3 , 74 , 25 , 81 , 60

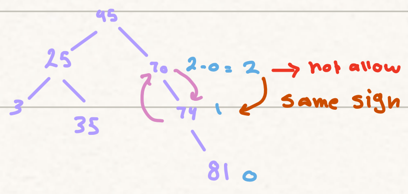




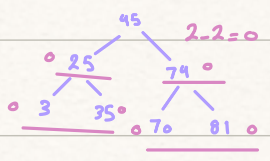
oppo sit sign (doble rotations)







Single rotations



Complexity of Binary Search Tree

Operation	Average	Worst
Insert	O(log(n))	O(n)
Delete	O(log(n))	O(n)
Search	O(log(n))	O(n)

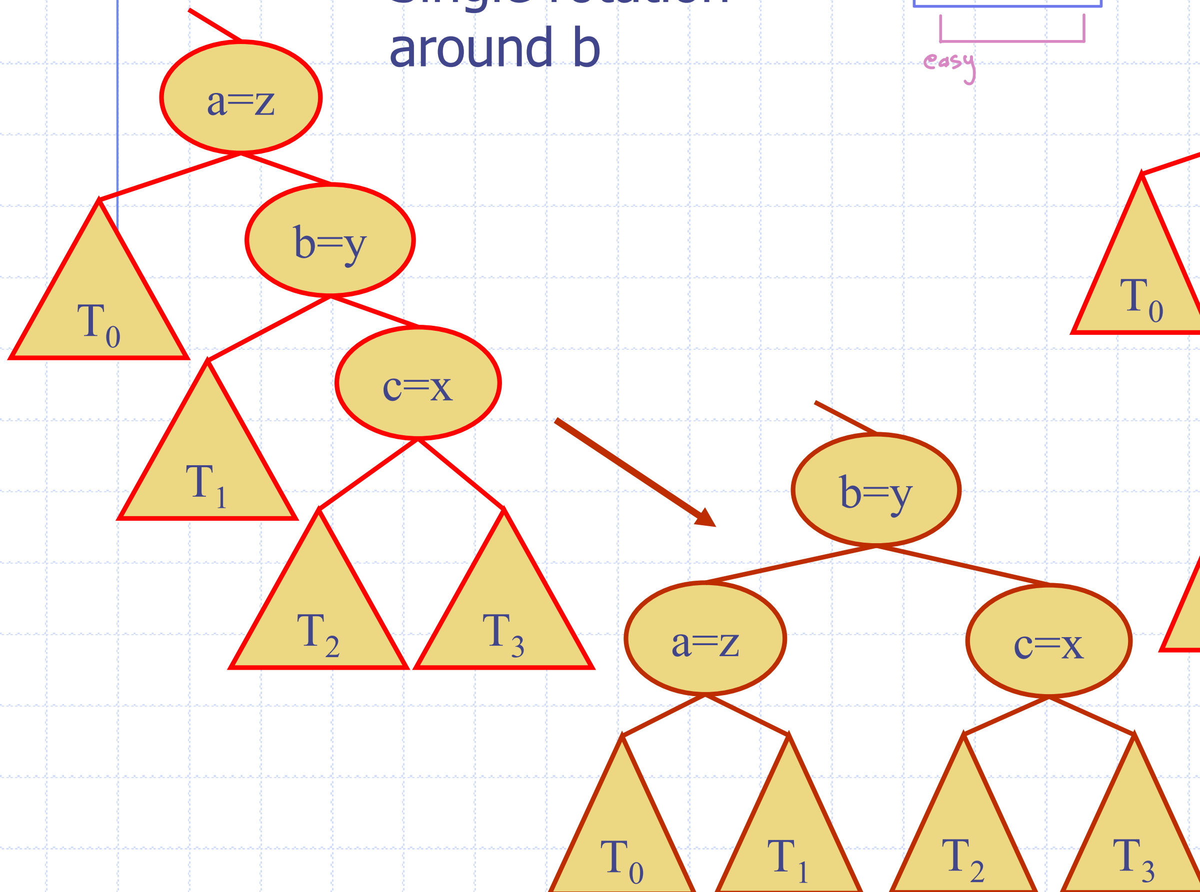
Complexity of Balanced Binary Search Tree → لأنها مرتبه

Operation	Average	Worst
Insert	O(log(n))	O(log(n))
Delete	O(log(n))	O(log(n))
Search	O(log(n))	O(log(n))

Trinode Restructuring

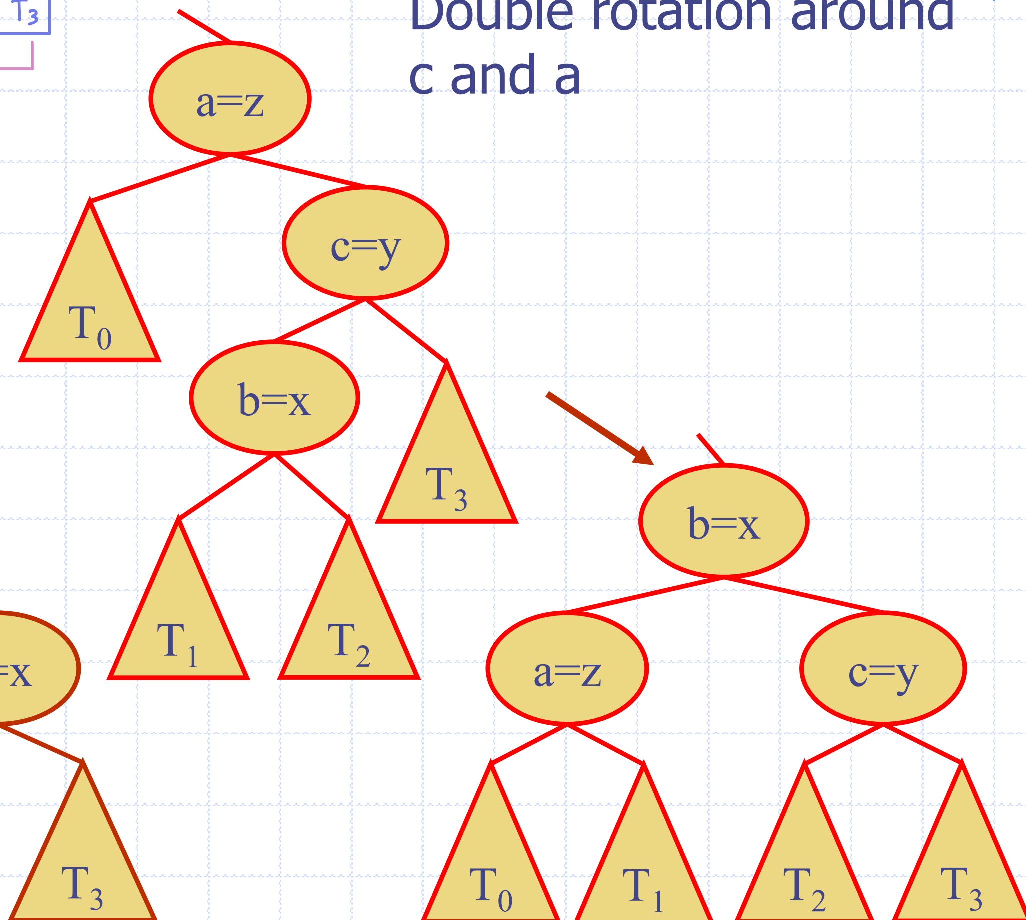
working with 3 node.

- Let (a,b,c) be the inorder listing of x, y, z
- Perform the rotations needed to make b the topmost node of the three



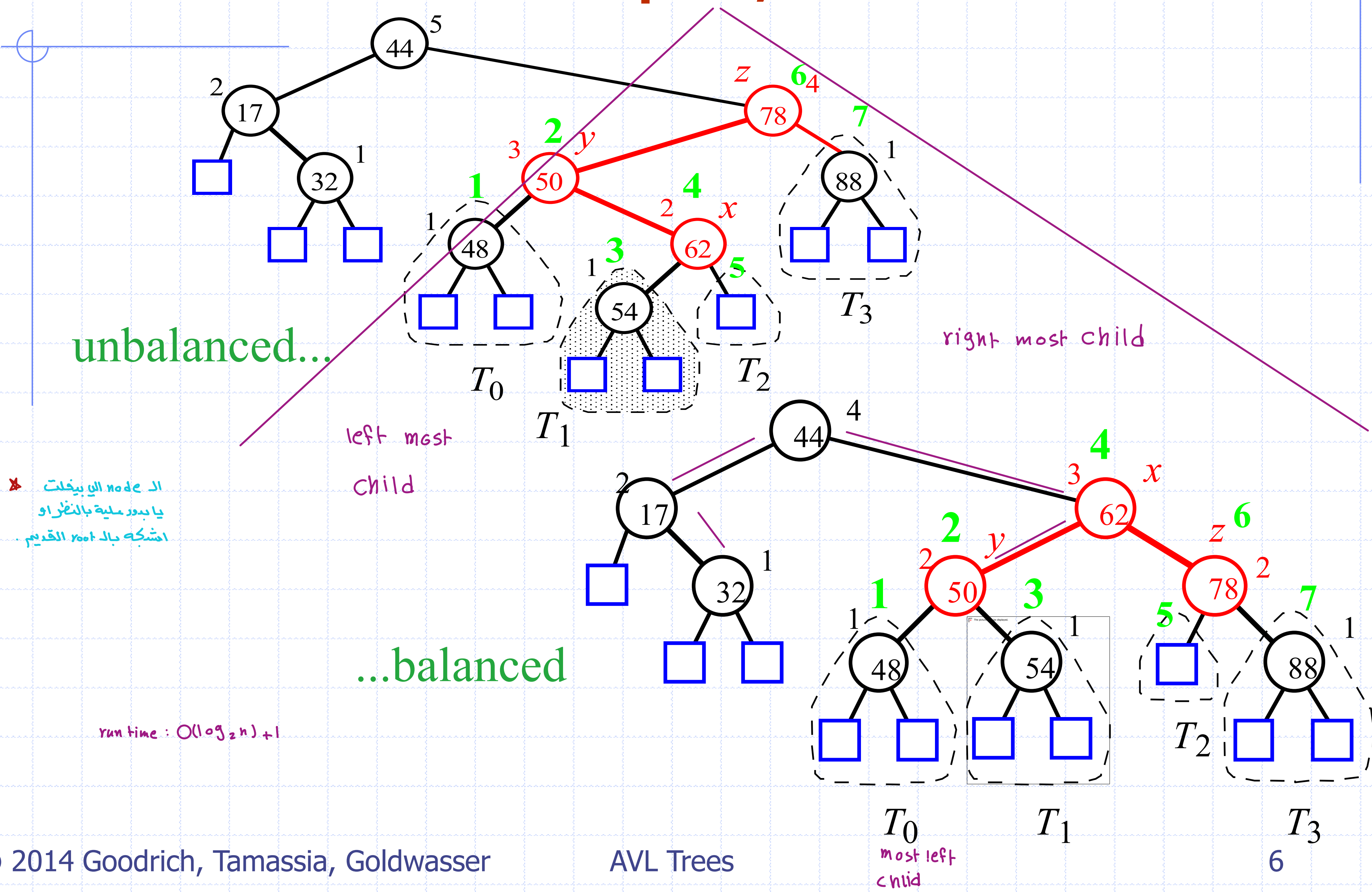
T0 T1 T2 T3

easy



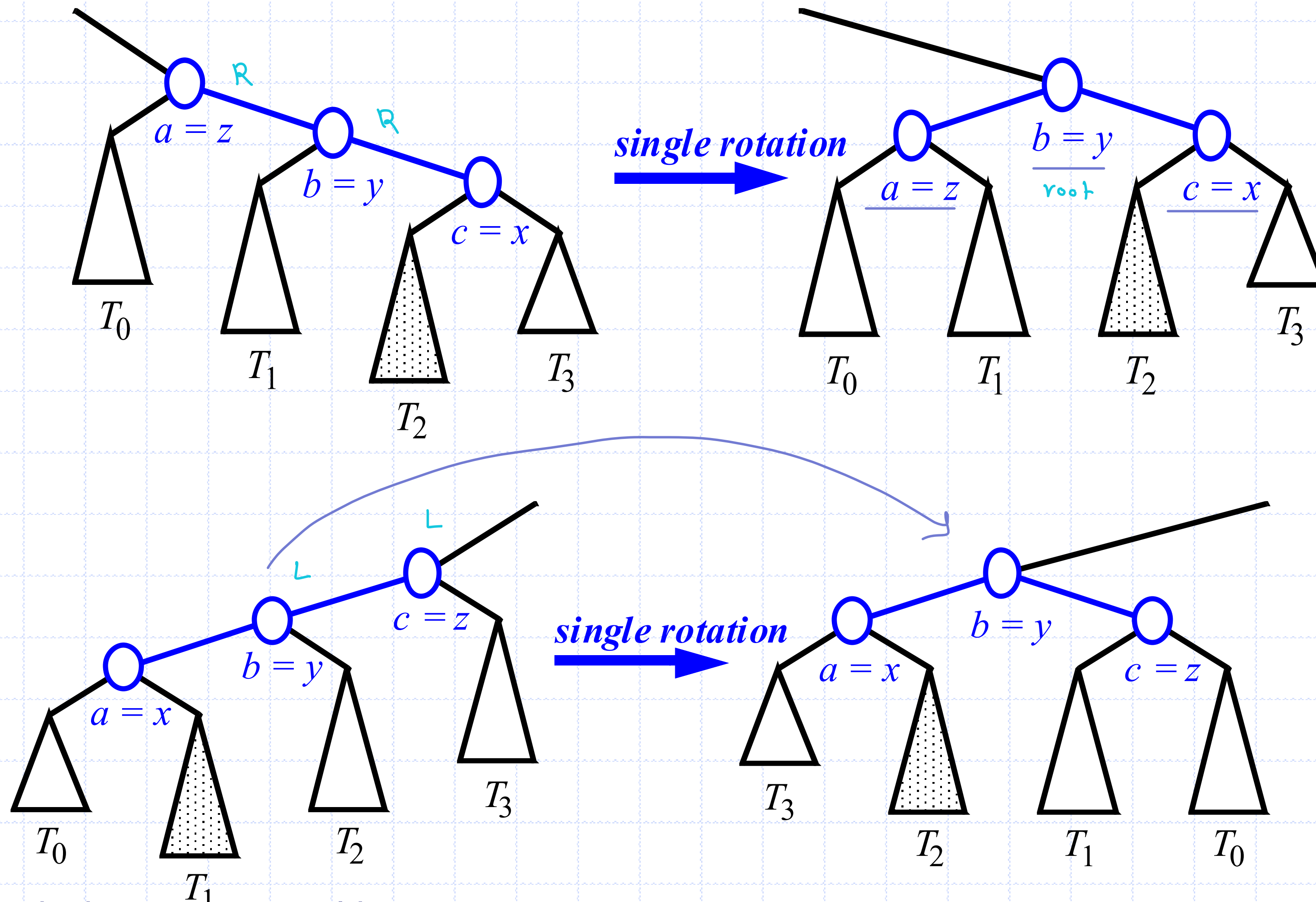
© 2014 Goodrich, Tamassia, Goldwasser

Insertion Example, continued



Restructuring (as Single Rotations)

- Single Rotations:

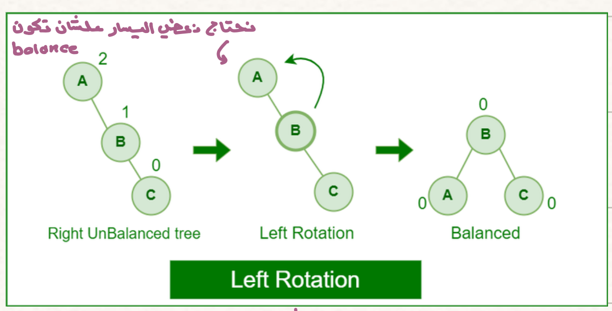


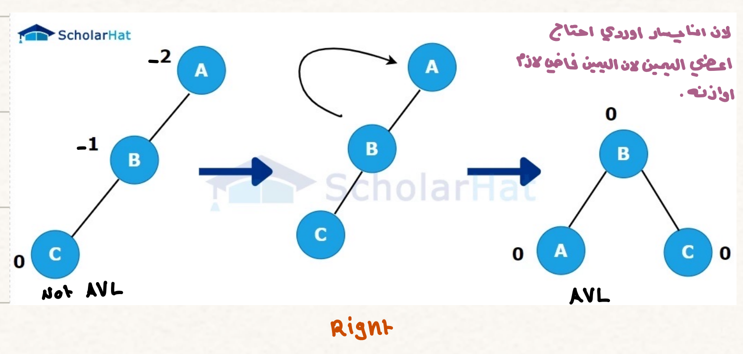
single
1- rotation (node مشكله) + (قبلها) same sign.

لم تدويره وحدة

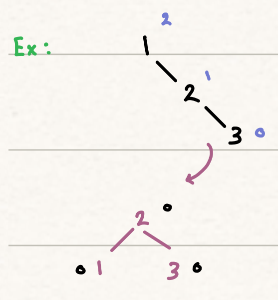
يكون الـ right ← نفسه والـ قبله نفس الإشارة.

← الي قبل فيها المشكله تدفها.

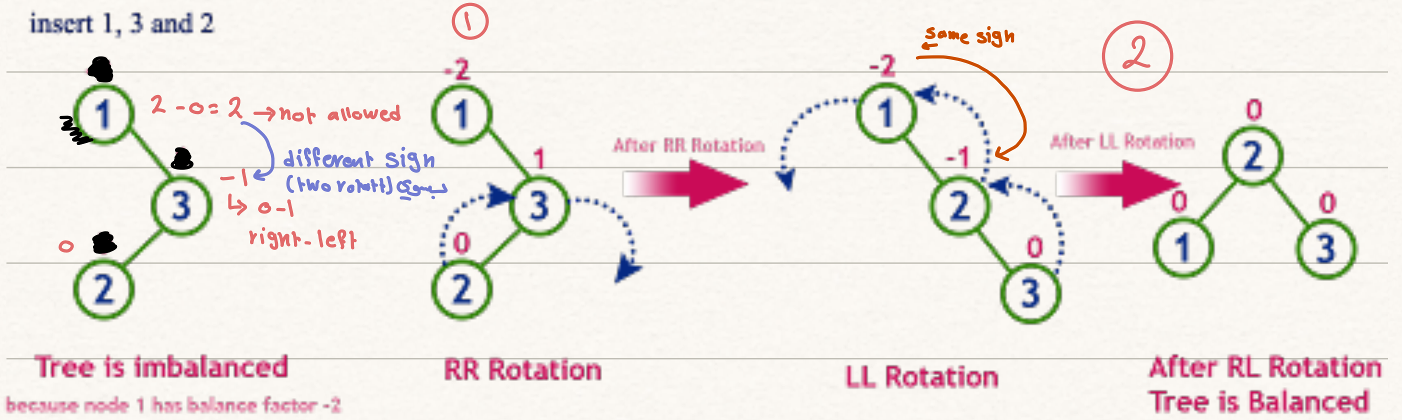


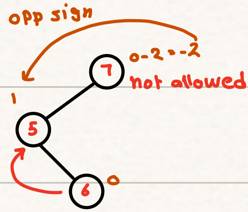


بنخلي الي قبل الي فية مشكله تدفها لتحت.

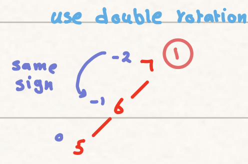


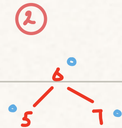
2- rotations (node مشكله) + (قبلها) opp. sign
↳ عكس ها





use double rotation

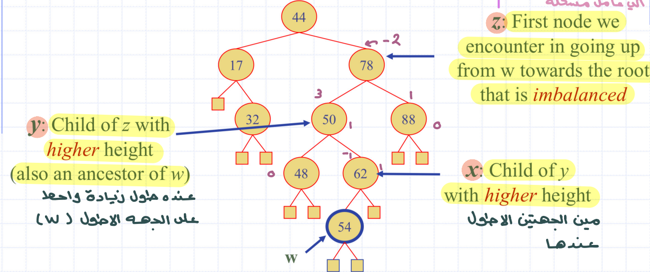




١- الي فيها مشكله نشخب عليها ونركز على الثانين ونعاملها معامله (own rotation)

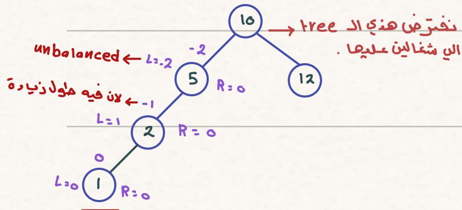
عندي مشكله باليسار بلحظا يمين (الي تحت بتدف الى فوق ، الي فوق بتنزل)

٢- الحين امسح الى شخبته المشكه واشوف .



✱ اول شي ء حددنا الـ node اليعلية مشكله وسمينا الـ node الي بعدة .

كيف نعمل الـ balancing : لمن يصير عندي node ← unbalanced



1_ Insert element (1).

2_ نرجع نحسب الـ balance بعد القيمة الي اضفتها .

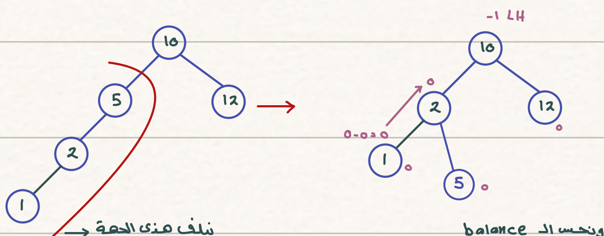
3_ حددنا الـ unbalanced بعدة علشان نشوف ايش نوعها لازم نحدد وين الطول الزايد اي جهة .

4_ حددنا الطول → الطول الزيادة بالـ left والي هو الاب (5) الـ عامل مشكله .

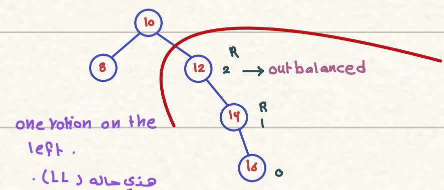
5_ اذا كانت المشكله في LL or RR
نسوي (one rotion)

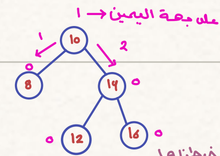
one rotion : عندها طول زيادة في الجهة اليسار ، فا بنلفها بحيث نقلل من الطول الزيادة .

فا بدال من الـ (5) هي الـ root بنزلها ونخلي الـ 2 هي الـ root .



نوع ثاني الـ mirror

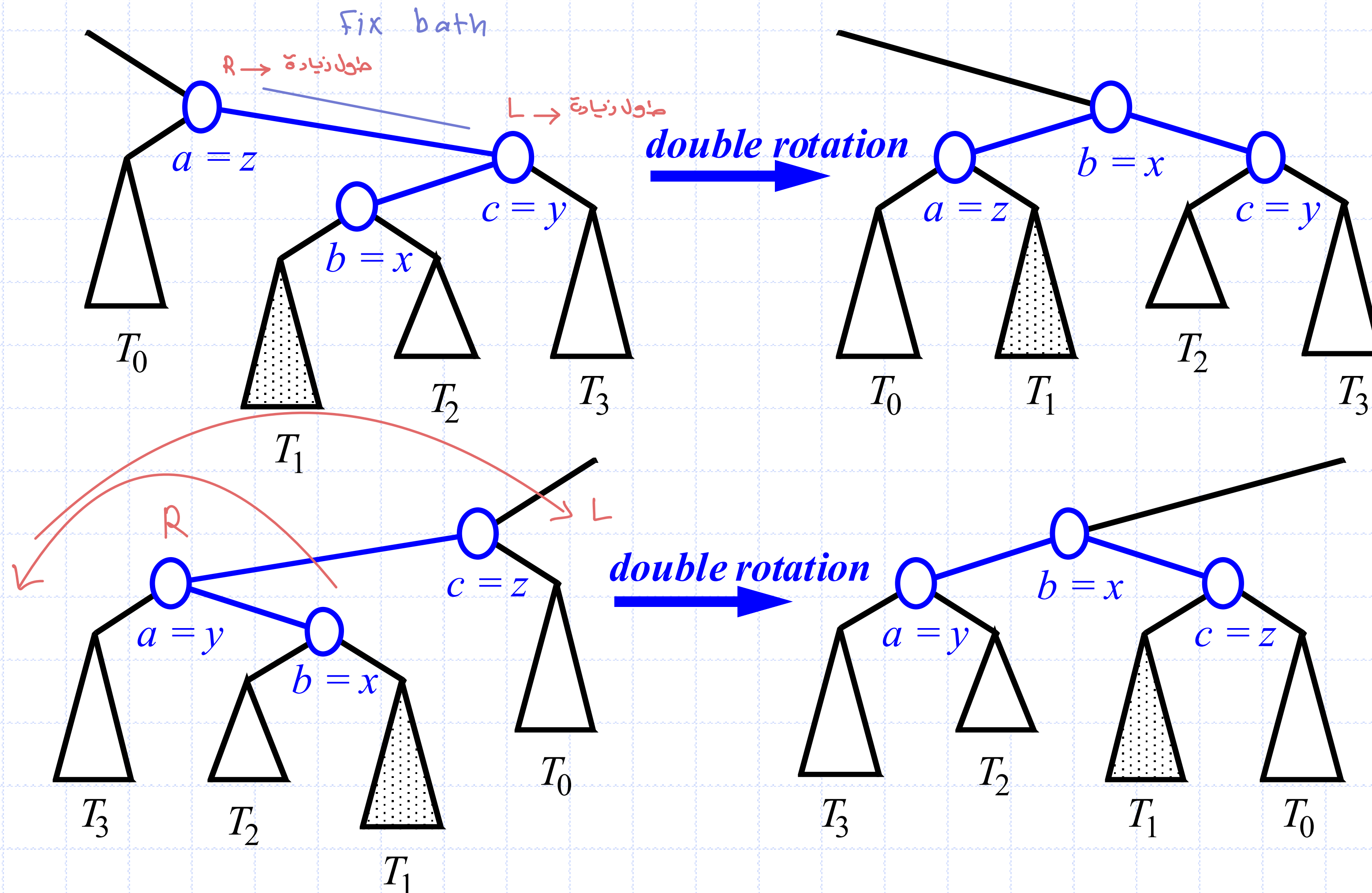




ونحسب الـ balanced بعدما ظبطناها
هذي حاله (RR)

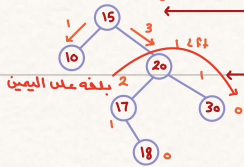
Restructuring (as Double Rotations)

- double rotations:



(as Double Rotations)

unbalanced ← 2 R hight



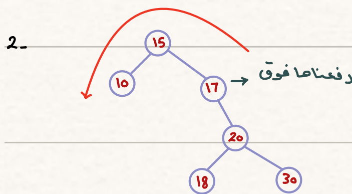
بلففه على اليمين

RH_LH

اول شيء نبدا
من الابن عنده
طول زيادة على
الجهة اليسار

1_ نحسب الـ balance

2_

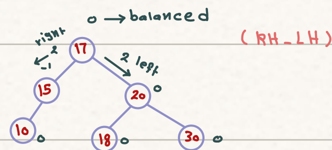


دفعناها فوق

لما انفكت بندور على المكان
المناسب واحطها فيه .

3_ وبنلفها مرة ثانية

0 → balanced



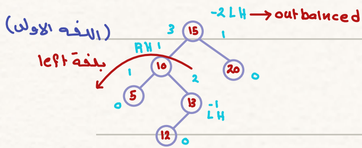
(RH_LH)

هي اصلا : كان عندها مشكلة
في الجهة الـ left وحلت

لفيتها right فصارت right
وحنا لفيناها من حالة الابن left
والاب right : بنلفها للحالة الاولى (RR) .

(LH_RH)

-2 LH → out balanced

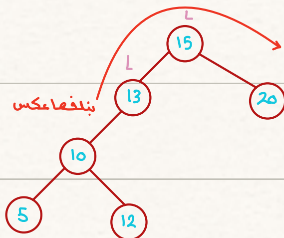


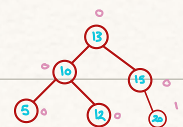
1_ نحسب الـ balanced

2_ عندها مشكلة بالـ left
بتروح للـ left وتشوف ايش
مشكلة بيطلع (right)

Double rotition : اذا اختلف النوعين معناته

دائماً ابدا بالابن بعدين
الاب .



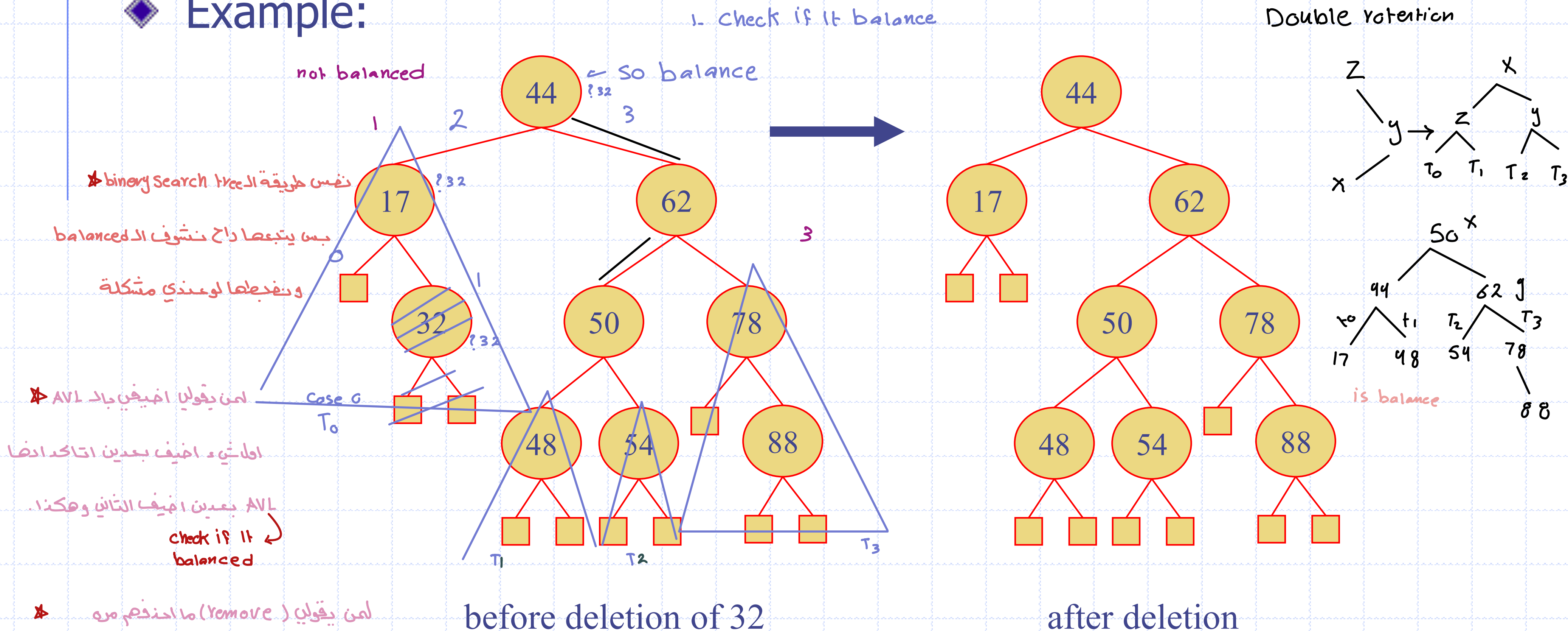


ونحسب الـ balanced
بعد اللفه الثانيه.

بعد ما خذمنا اللفه الاولى بنلفها اللفه الثانيه عكس →

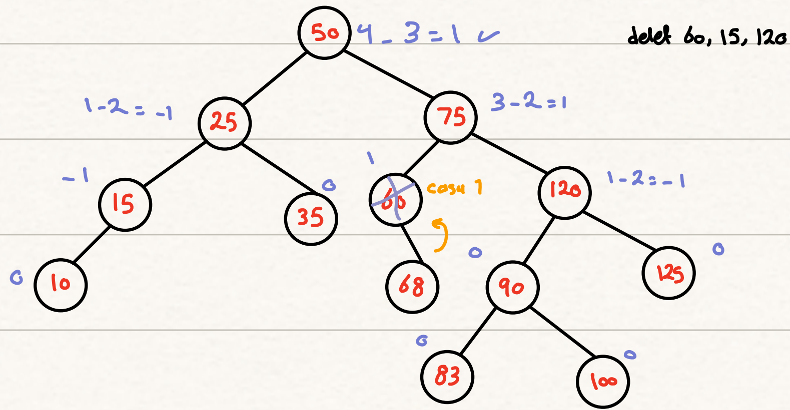
Removal

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:



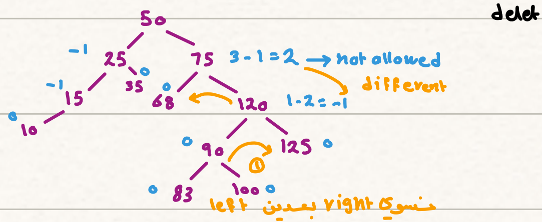
before deletion of 32

after deletion

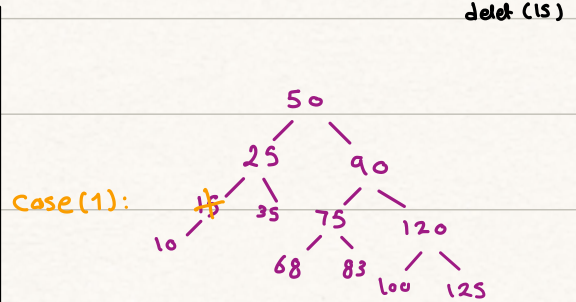


delet 60, 15, 120

delet (60)

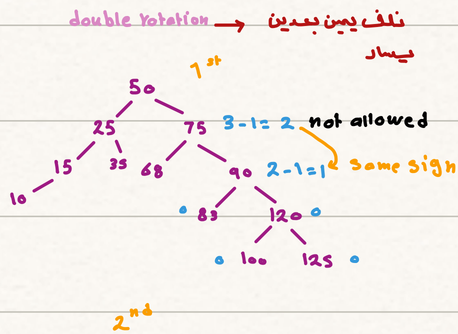


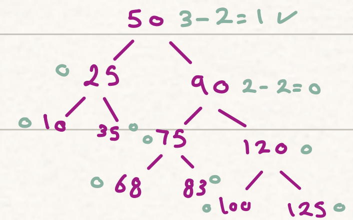
delet (15)



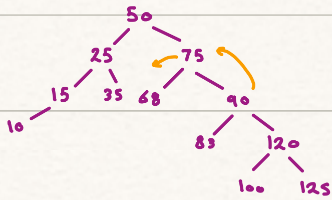
double rotation → نلف يمين بعدين يسار

1st

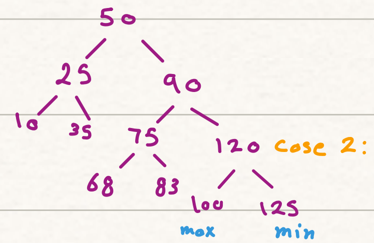


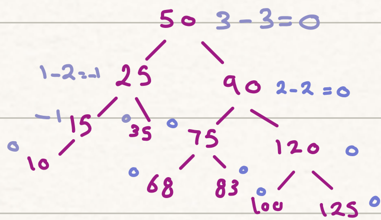


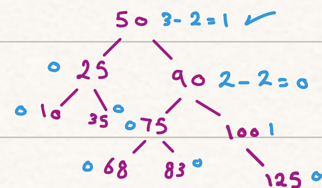
2nd



delet(120)

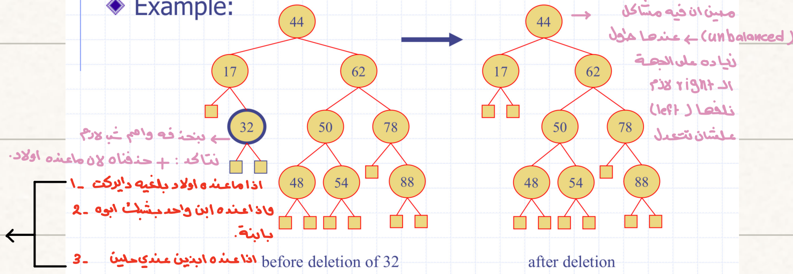






Removal

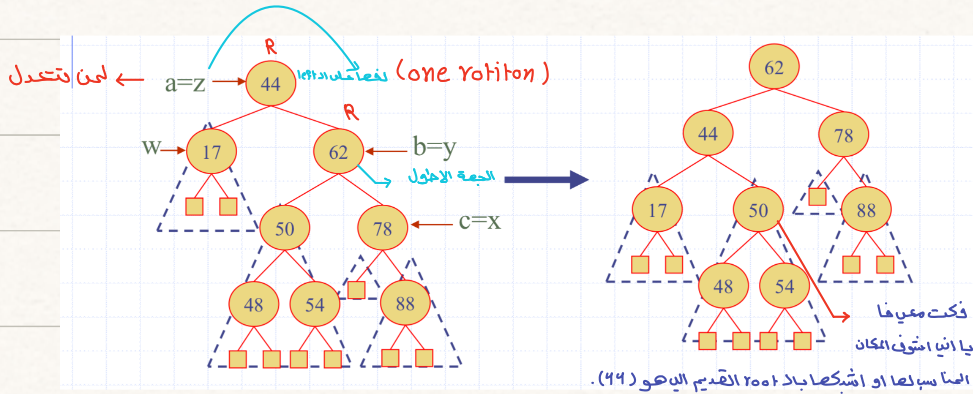
- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, *may* cause an imbalance.
- Example:



الـ (remove) العاديه الي تعلمناها من الـ (tree).

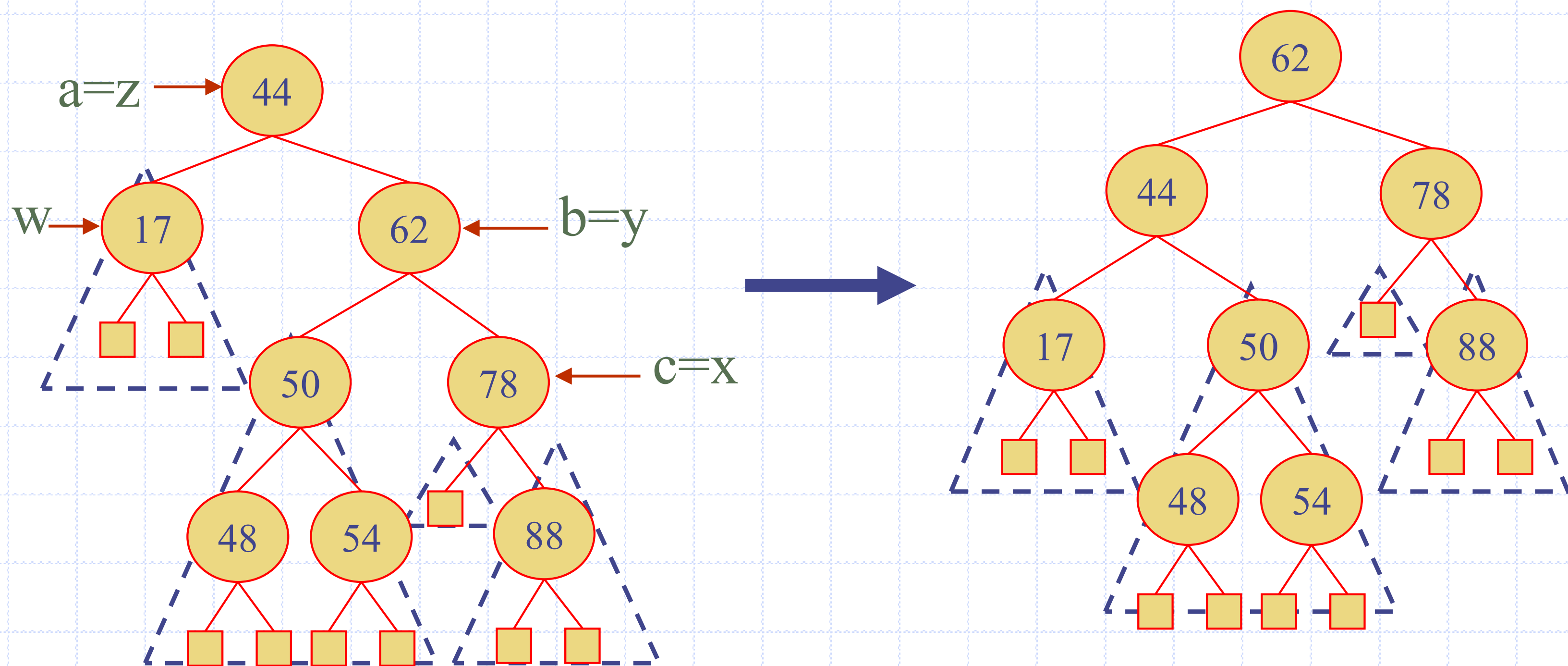
يا اروح للجهة الـ (left) واخذ الـ node الكبير وابدله بالقيمه بعدين الغيه من تحت.

او اروح للجهة الـ (right) واخذ الـ node الصغيرة وابدله بالقيمه و بعدين الغيه من تحت.

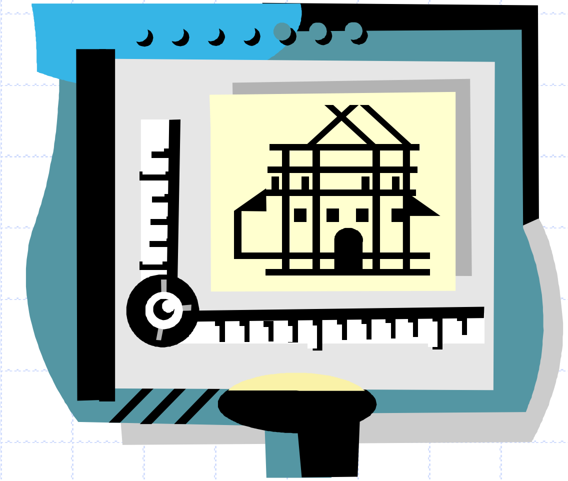


Rebalancing after a Removal

- Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height
- We perform a trinode restructuring to restore balance at z
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



AVL Tree Performance



- AVL tree storing n items
  - The data structure uses O(n) space
  - A single restructuring takes O(1) time
    - using a linked-structure binary tree → نرتبها او اضيف خانه او اشيل .
  - Searching takes O(log n) time → لاني بروح يمين ويسار
    - height of tree is O(log n), no restructures needed
  - Insertion takes O(log n) time → لان ببحث بالاول بعدين بضيف
    - initial find is O(log n)
    - restructuring up the tree, maintaining heights is O(log n)
  - Removal takes O(log n) time → نبحث اول شيء بعدين بحذف وبعدله الـ tree .
    - initial find is O(log n)
    - restructuring up the tree, maintaining heights is O(log n)

Java Implementation

```
1  /** An implementation of a sorted map using an AVL tree. */
2  public class AVLTreeMap<K,V> extends TreeMap<K,V> {
3    /** Constructs an empty map using the natural ordering of keys. */
4    public AVLTreeMap() { super(); }
5    /** Constructs an empty map using the given comparator to order keys. */
6    public AVLTreeMap(Comparator<K> comp) { super(comp); }
7    /** Returns the height of the given tree position. */
8    protected int height(Position<Entry<K,V>> p) {
9      return tree.getAux(p);
10   }
11   /** Recomputes the height of the given position based on its children's heights. */
12   protected void recomputeHeight(Position<Entry<K,V>> p) {
13     tree.setAux(p, 1 + Math.max(height(left(p)), height(right(p))));
14   }
15   /** Returns whether a position has balance factor between −1 and 1 inclusive. */
16   protected boolean isBalanced(Position<Entry<K,V>> p) {
17     return Math.abs(height(left(p)) − height(right(p))) <= 1;
18   }
```

Send a node

Java Implementation, 2

```
19    /** Returns a child of p with height no smaller than that of the other child. */
20    protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p) {
21      if (height(left(p)) > height(right(p))) return left(p);              // clear winner
22      if (height(left(p)) < height(right(p))) return right(p);             // clear winner
23      // equal height children; break tie while matching parent's orientation
24      if (isRoot(p)) return left(p);                    // choice is irrelevant
25      if (p == left(parent(p))) return left(p);         // return aligned child
26      else return right(p);
27    }
```

© 2014 Goodrich, Tamassia, Goldwasser

Java Implementation, 3

```
33    protected void rebalance(Position<Entry<K,V>> p) {
34      int oldHeight, newHeight;
35      do {
36        oldHeight = height(p);                       // not yet recalculated if internal
37        if (!isBalanced(p)) {                        // imbalance detected
38          // perform trinode restructuring, setting p to resulting root,
39          // and recompute new local heights after the restructuring
40          p = restructure(tallerChild(tallerChild(p)));
41          recomputeHeight(left(p));
42          recomputeHeight(right(p));
43        }
44        recomputeHeight(p);
45        newHeight = height(p);
46        p = parent(p);
47      } while (oldHeight != newHeight && p != null);
48    }
49    /** Overrides the TreeMap rebalancing hook that is called after an insertion. */
50    protected void rebalanceInsert(Position<Entry<K,V>> p) {
51      rebalance(p);
52    }
53    /** Overrides the TreeMap rebalancing hook that is called after a deletion. */
54    protected void rebalanceDelete(Position<Entry<K,V>> p) {
55      if (!isRoot(p))
56        rebalance(parent(p));
57    }
58  }
```

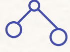
© 2014 Goodrich, Tamassia, Goldwasser

	AVL Tree		Binary Search Tree	
	Average	**Worst**	**Average**	**Worst**
Insertion	O(logn)	O(logn)	O(logn)	O(n)
Deletion	O(logn)	O(logn)	O(logn)	O(n)
Searching	O(logn)	O(logn)	O(logn)	O(n)
Traversal	O(n)	O(n)	O(n)	O(n)

BT_BSt_AVL Rules :

اول شيء اخذنا الـ BT بعدين الـ BST بعدين الـ AVL

1_ BT → maximum child (2)



نطورت وصارت

2_ BSt → maximum احلها ثنائيه بالـ
عندها ترتيب بالذات
الاب اصغر من الـ left
واكبر من الـ right

مستحيل يصير غير هالقوانين ، لازم الـ rule
تنطابق على كل الـ node اذا node واحد خرب الفكرة
معناته : بطل هذا النوع صار النوع الـ اقل منه .

3_ AVL → binary Search tree عباده عن
ان الـ (left) اصغر والـ (right) الكبر
وقاعدتها الاساسية : ان هذي الـ tree سوء اضفنا او حذفنا
بنخفظها باصغر (level) ممكنة
يكون الفرق الـ (hight) بين الاولاد الكثر
من واحد . اذا كانت اكثر من واحد بطلت
(AVL tree) بتكون (BST) .

balanced tree ← بتكون

maximum (BT) :
ماعندها قاعدة لازم
البحث بالـ tree كامله

maximum (BST) : orderd data
بيكون باقصى اليمين →

```
while (P.right != null)
    P = P.right
    return P.data
```

minum (BST) :
بيكون باقصى اليسار

```
      root
while (P.left != null)
    P = P.left
    return P.data
             or key
```

خلك ← P = P.left
تحرك

---

remeve :

(BT) = remob sub بنسوي → بلغي الـ node والي تحته

(BST) :
Same rule — 3 causes
(AVL) :

1) no child : بلغيه
2) on child : بشبك الابن بالاب
3) two child : يا باخذ الـ min من اقصى اليمين بلغيه واحطه فوق .

يتبعها ترتيب الـ tree
و نسوي balance .

Insert :

(BS) : انا احدد وين اضيف لان
ما فيه قاعدة امشي عليها

(BST) :
(AVL) :
— Same rule

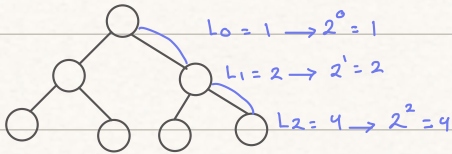
الإضافه حسب القيمه للـ key
نعمل search بعدين نقرر وين
نضيف بعدين insert.

Search : بنشوف وين المكان
المناسب ونضيفه. لكن الـ (AVL)
نحتاج لتعديل (balanced).

---

hight :

$h = 2$



2^{level} ← عدد النودات في نفس level

عدد النودات كامله بالـ tree = $1+2+4 = 7$

$2^{h+1} - 1 = 2^3 - 1 = 8 - 1 = 7$

max = $2^{h+1} - 1 = 2^3 - 1 = 7$

min = $h + 1 = 3$

نحسبه من الـ edge