Lecture 5c

Part A

Balanced Binary Search Tree - Motivation and Property

Worst-Case RT: BST with Linear Height

Example 1: Inserted Entries with Decreasing Keys

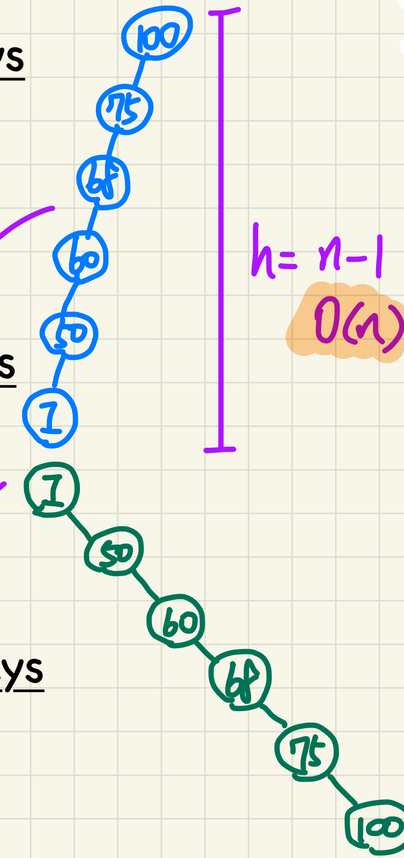
<100, 75, 68, 60, 50, 1>

key

Example 2: Inserted Entries with Increasing Keys

<1, 50, 60, 68, 75, 100>

searching worst-case RT: $O(n)$



$h = n-1$

$O(n)$

$h = n-1$

$O(n)$

Example 3: Inserted Entries with In-Between Keys

$\boxed{<1, 100, 50, 75, 60, 68>}$

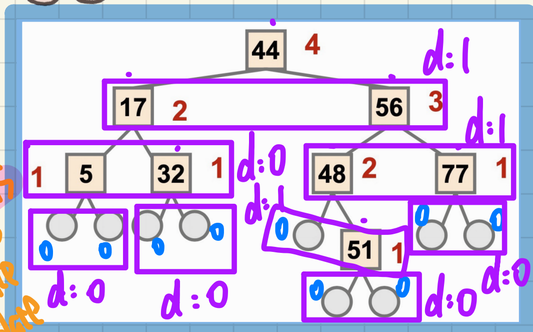
Exercise

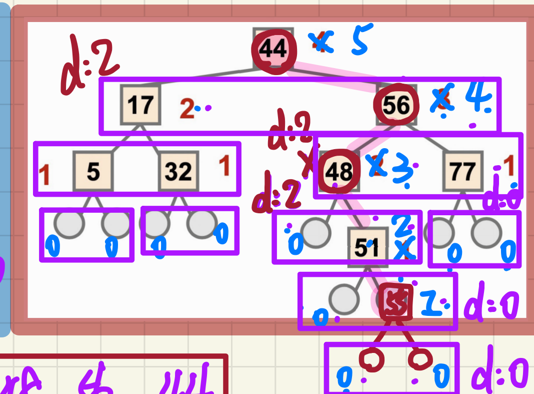
Balanced BST: Definition

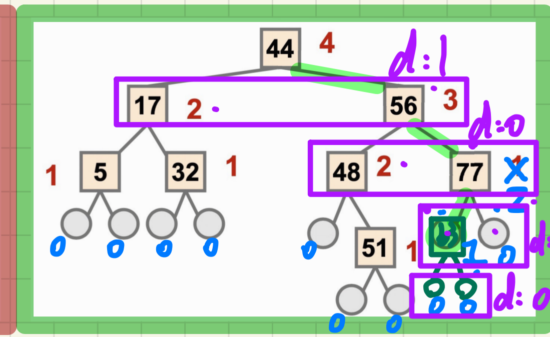
- internal node
- height
- height balance

Given a node p, the ***height*** of the subtree rooted at p is:

$$\textit{height}(p) = \begin{cases} 0 & \text{if } p \text{ is } \textit{external} \\ 1 + \textbf{MAX}\,(\{\ \textit{height}(c) \mid \textit{parent}\ (c) = p\ \}) & \text{if } p \text{ is } \textit{internal} \end{cases}$$







when rotations need to take place

ancestor path of 55: 55, 51, $\boxed{48, 56, 44}$

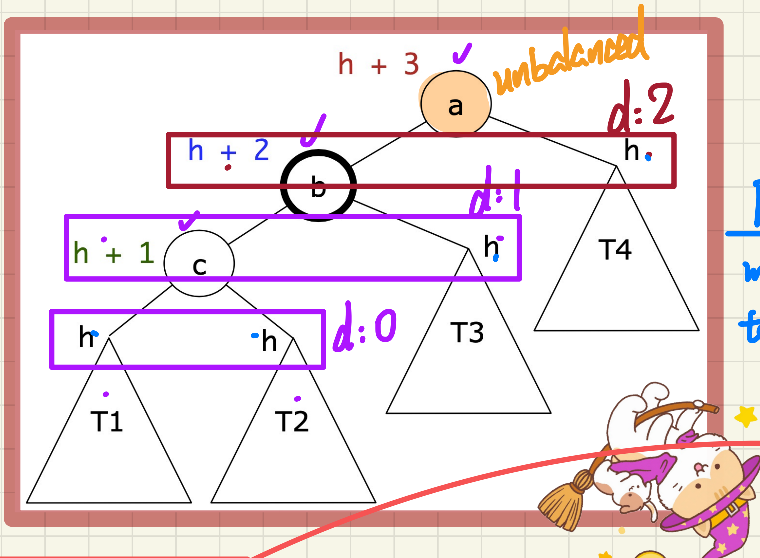
unbalanced after insertion.

Q. Is the above tree a **balanced BST**? YES.

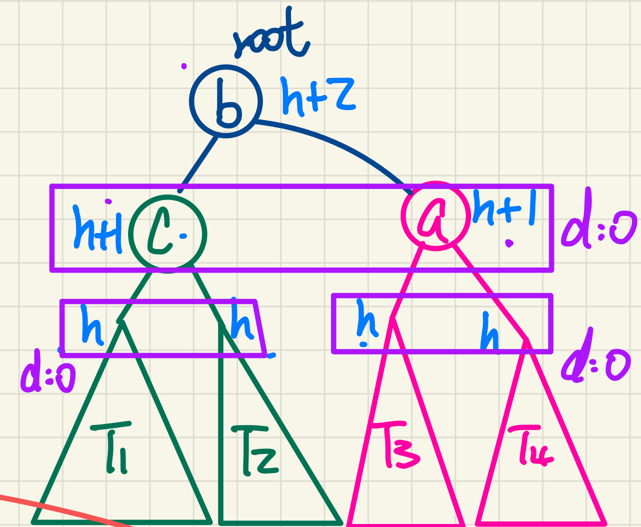
Q. Still a **balanced BST** after inserting **55**? NO.

Q. Still a **balanced BST** after inserting 63? YES.

Restoring Balance via Rotations



Rotate middle node b to the right



same

Before Rotation, In-Order Traversal:

< T1, C, T2, b, T3, a, T4 >

① balanced? ✓

② same content? ✓

③ BST? ✓

After Rotation, In-Order:

< T1, C, T2, b, T3, a, T4 >

Q. Is the above tree **balanced**?

Q. After a right-rotation on node b, is the resulting tree still a **BST**?

Lecture 5c

Part B

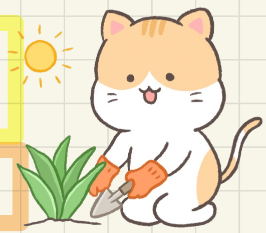
Balanced Binary Search Tree -
Trinode Restructuring after Insertion

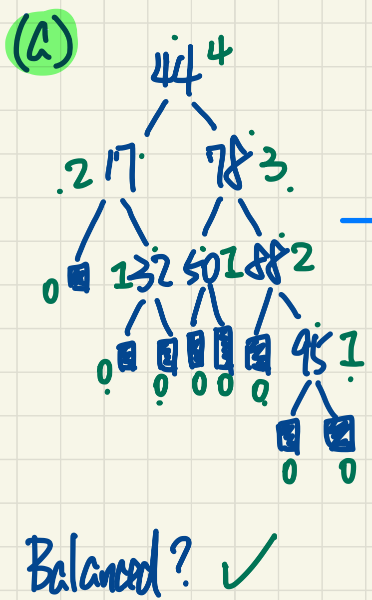
Trinode Restructuring after Insertion: Left Rotation

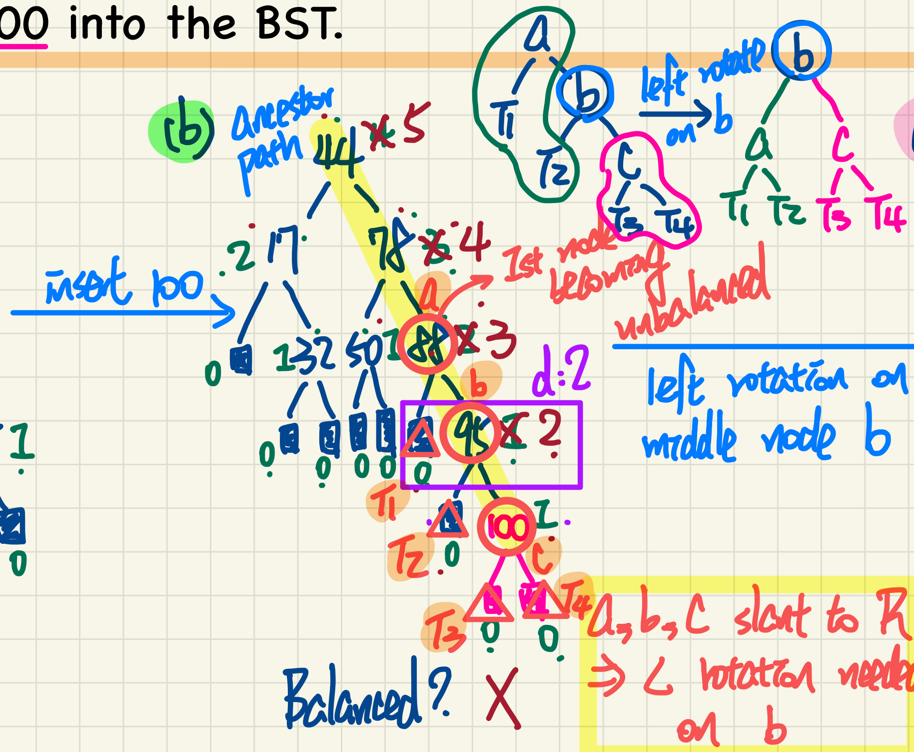
- Insert the following sequence of keys into an empty BST:

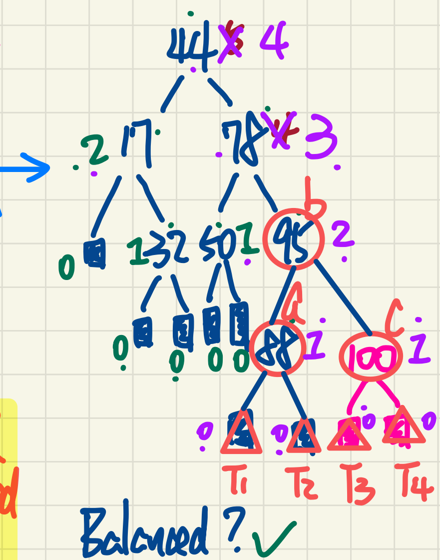
  <44, 17, 78, 32, 50, 88, 95>

- Insert 100 into the BST.







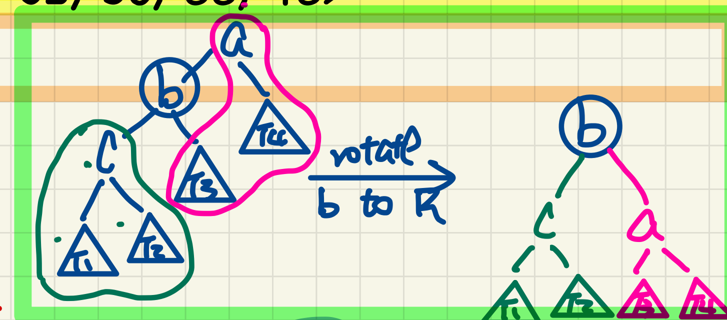


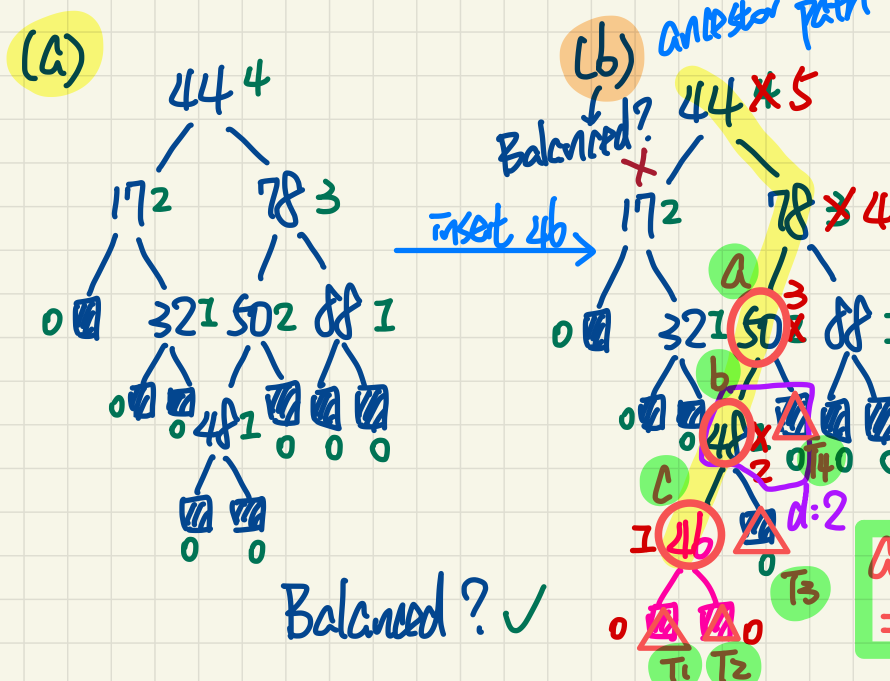
Trinode Restructuring after Insertion: Right Rotation

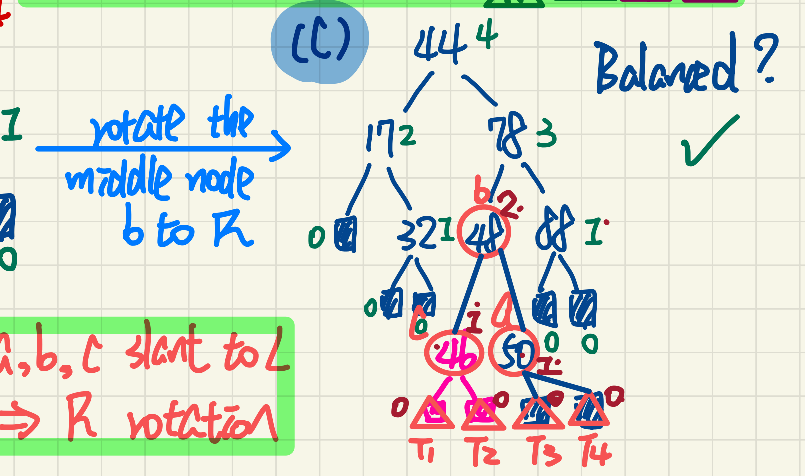
- Insert the following sequence of keys into an empty BST:

  <44, 17, 78, 32, 50, 88, 48>

- Insert 46 into the BST.





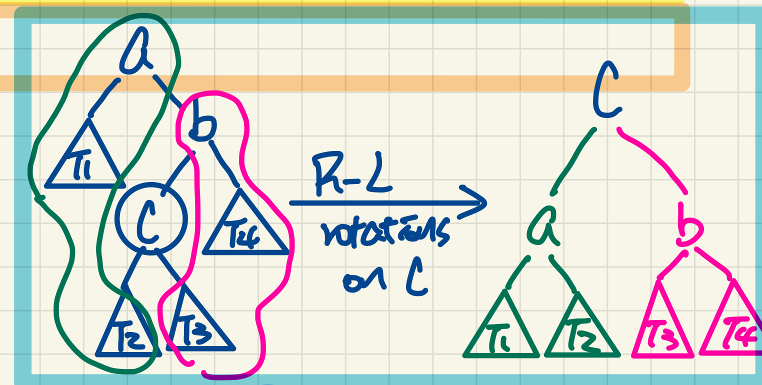


Trinode Restructuring after Insertion: R-L Rotations

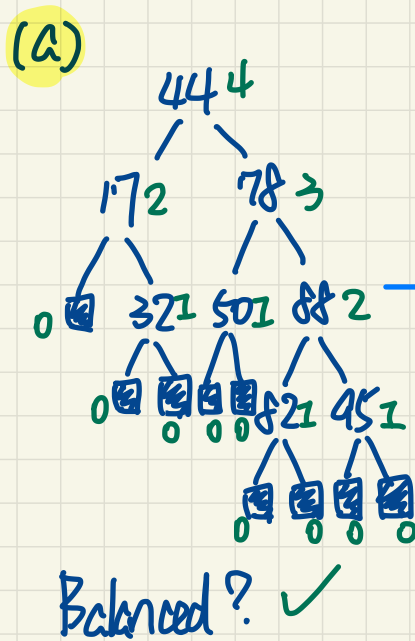
- Insert the following sequence of **keys** into an empty BST:

  <44, 17, 78, 32, 50, 88, 82, 95>

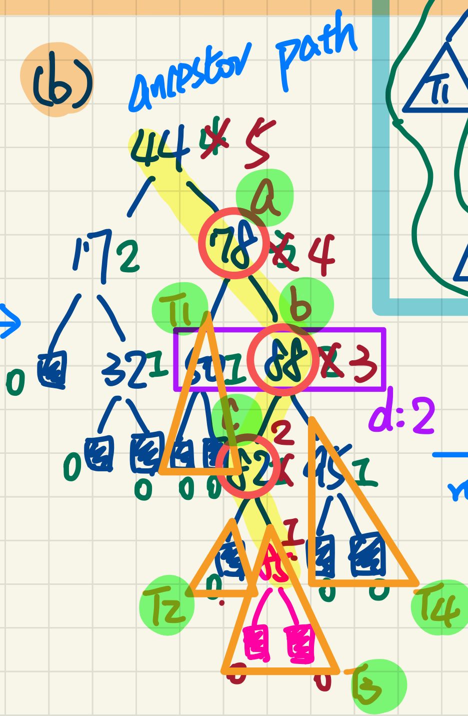
- Insert 85 into the BST.



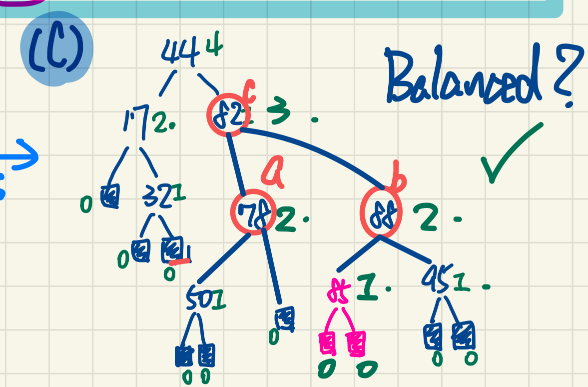
(a)



(b)



(c)



Trinode Restructuring after Insertion: L-R Rotations

- Insert the following sequence of keys into an empty BST:

  <44, 17, 78, 32, 50, 88, 48, 62>

- Insert 54 into the BST. Exercise

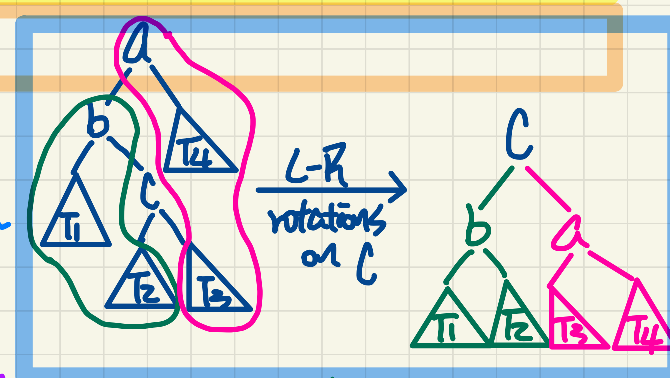
Trinode Restructuring after Insertion: L-R Rotations

Solution

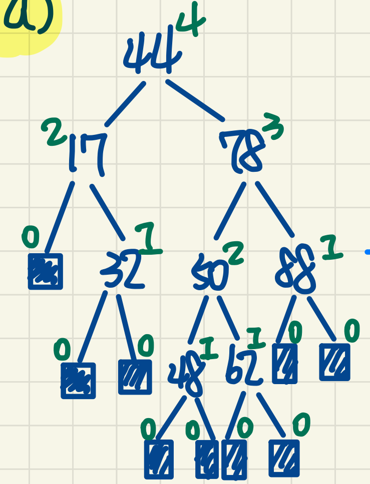
- Insert the following sequence of keys into an empty BST:

  <44, 17, 78, 32, 50, 88, 48, 62>

- Insert 54 into the BST.



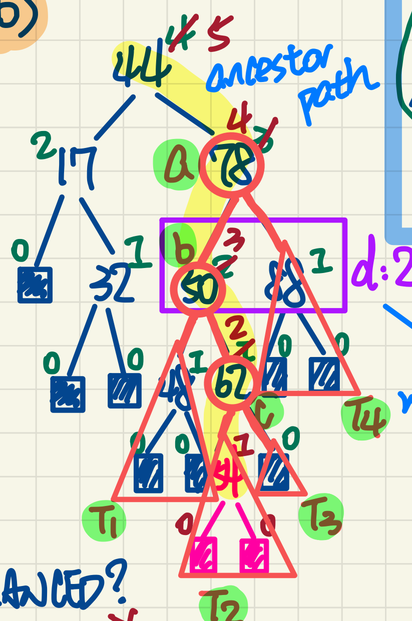
(a)



BALANCED? ✓

Insert 54 →

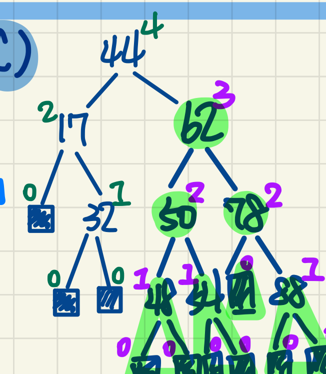
(b)



BALANCED? ✗

L-R rotations on C (62) →

(C)



BALANCED? ✓

Lecture 5c

Part C

Balanced Binary Search Tree -
Trinode Restructuring after Deletion

Trinode Restructuring after Deletion: Single Rotation

- Insert the following sequence of keys into an empty BST:

  <44, 17, 62, 32, 50, 78, 48, 54, 88>

- Delete 32 from the BST.

Exercise

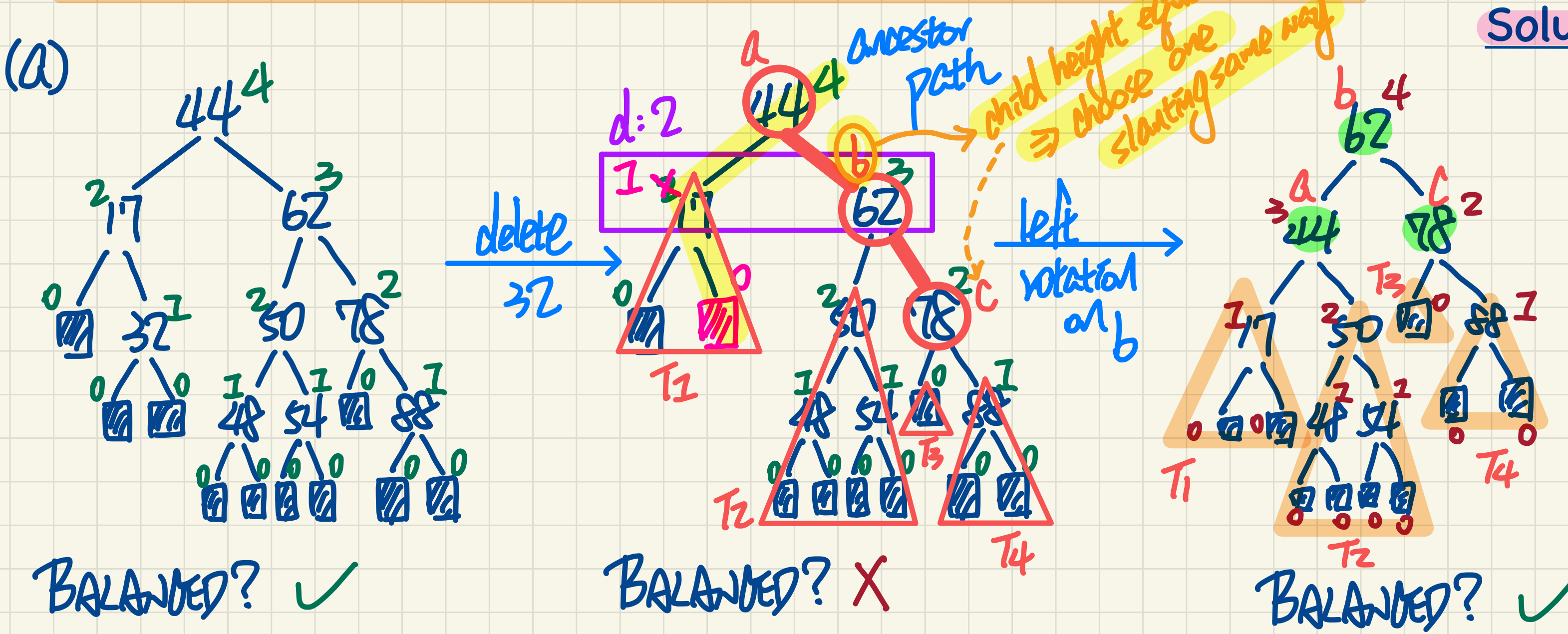
Trinode Restructuring after Deletion: Single Rotation

- Insert the following sequence of keys into an empty BST:

  <44, 17, 62, 32, 50, 78, 48, 54, 88>

- Delete 32 from the BST.

Solution

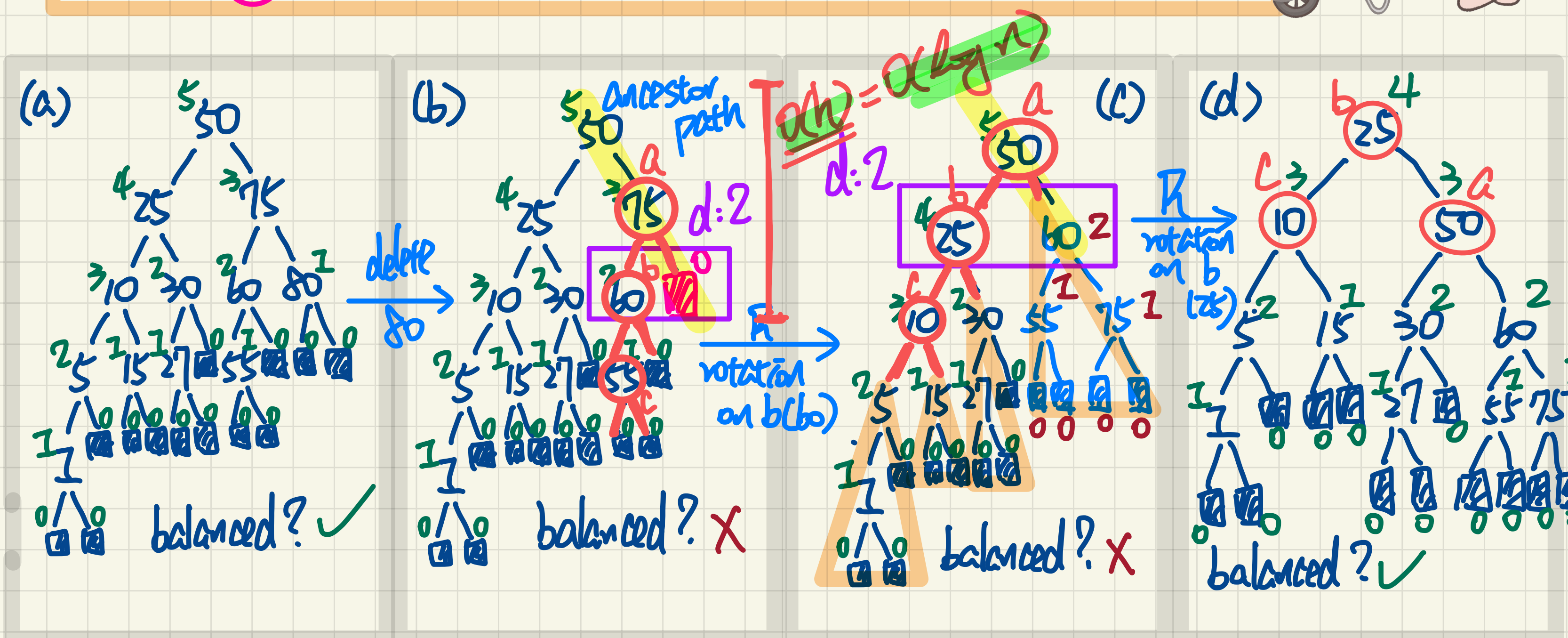


Trinode Restructuring after Deletion: Multiple Rotations

- Insert the following sequence of keys into an empty BST:

  <50, 25, 10, 30, 5, 15, 27, 1, 75, 60, 80, 55>

- Delete 80 from the BST.



Lecture 4

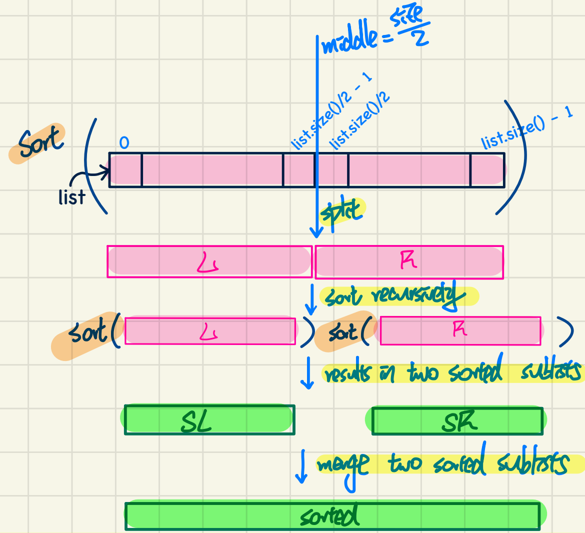
Part C

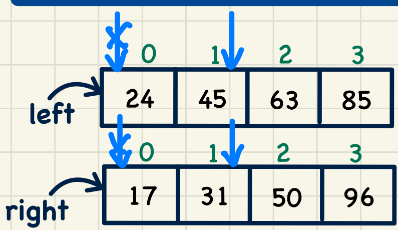
Examples on Recursion
Merge Sort
(continued)

Merge Sort in Java

```java
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>();
    sortedList.add(list.get(0));
  }
  else {
    int middle = list.size() / 2;
    List<Integer> left = list.subList(0, middle);
    List<Integer> right = list.subList(middle, list.size());
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = merge(sortedLeft, sortedRight);
  }
  return sortedList;
}
```

base cases





Precondition

L and R sorted

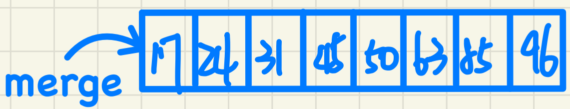
```java
/* Assumption:  L and R are both already sorted. */
private List<Integer> merge(List<Integer> L, List<Integer> R) {
  List<Integer> merge = new ArrayList<>();
  if(L.isEmpty()||R.isEmpty()) { merge.addAll(L); merge.addAll(R); }
  else {
    int i = 0;
    int j = 0;
    while(i < L.size() && j < R.size()) {
      if(L.get(i) <= R.get(j)) { merge.add(L.get(i)); i ++; }
      else { merge.add(R.get(j)); j ++; }
    }
    /* If i >= L.size(), then this for loop is skipped. */
    for(int k = i; k < L.size(); k ++) { merge.add(L.get(k)); }
    /* If j >= R.size(), then this for loop is skipped. */
    for(int k = j; k < R.size(); k ++) { merge.add(R.get(k)); }
  }
  return merge;
}
```

(a) # iterations: min(L.size(), R.size())

O(1)

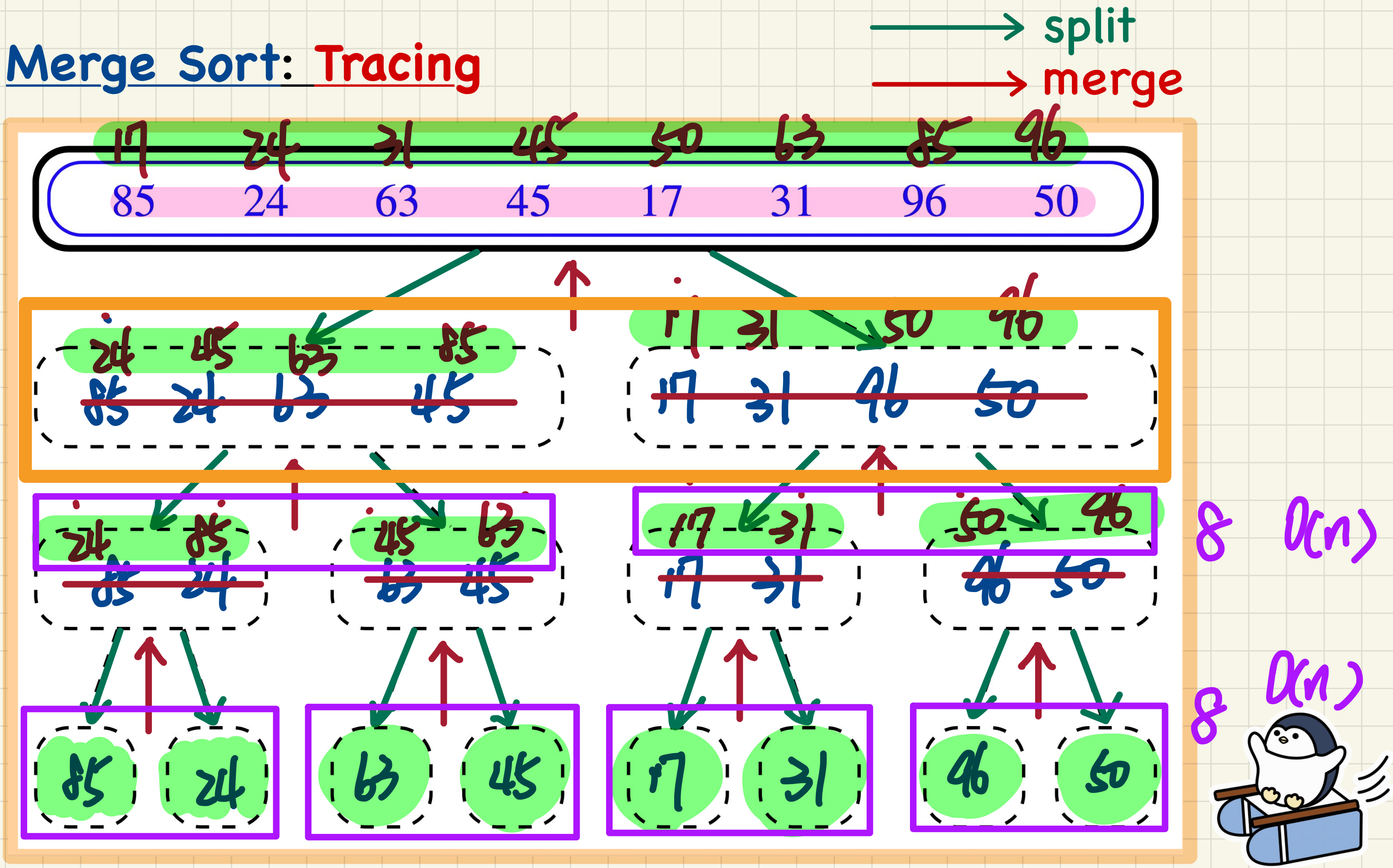
O(1)

(b) # iterations: remaining # of items to loop over in the longer list



(a) + (b) = L.size() + R.size()

Merge Sort: Tracing



Merge Sort: Running Time

```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>();
    sortedList.add(list.get(0));
  }
  else {
    int middle = list.size() / 2;
    List<Integer> left = list.subList(0, middle);
    List<Integer> right = list.subList(middle, list.size());
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = merge(sortedLeft, sortedRight);
  }
  return sortedList;
}
```

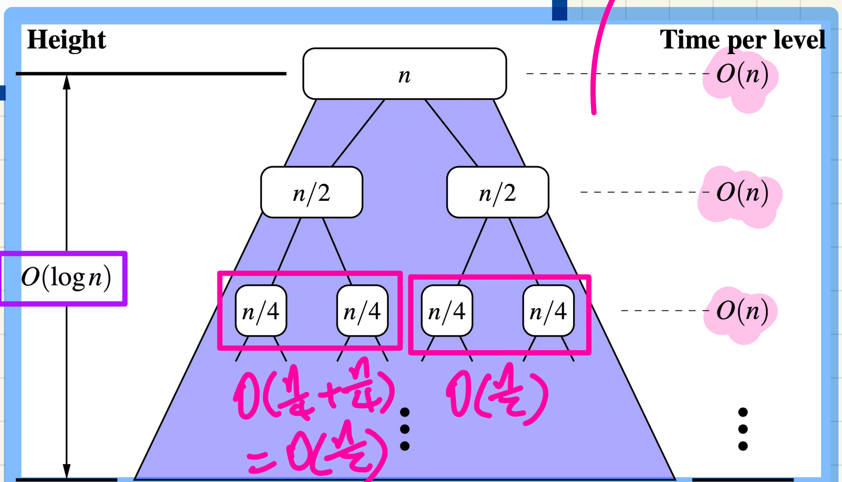
size = $\frac{n}{2}$

sort(left)
sort(right)

$O(1)$

$O(n)$

recursion tree is a full BT



Total RT:
$O(\log n \times n)$
$= O(n \cdot \log n)$

height of balanced BST

Running Time as a Recurrence Relation

$T(0) = 1$

$T(1) = 1$

$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n$

$2 \cdot T(\frac{n}{2})$

Running Time: Unfolding Recurrence Relation

$$T(0) = 1$$
$$T(1) = 1$$
$$T(n) = 2 \cdot T(n/2) + n$$

$$1 = \frac{n}{n} = \frac{n}{2^{\log n}}$$

$n = 8$

$2^{\log 8} = 8$

$$T(n) = 2 \cdot \boxed{T\left(\tfrac{n}{2}\right)} + n$$
$$= 2 \cdot \left(2 \cdot \boxed{T\left(\tfrac{n}{4}\right)} + \tfrac{n}{2}\right) + n \qquad \left[4 \cdot T\left(\tfrac{n}{4}\right) + 2n\right], \quad 4 = 2^2$$
$$= 2 \cdot \left(2 \cdot \left(2 \cdot T\left(\tfrac{n}{8}\right) + \tfrac{n}{4}\right) + \tfrac{n}{2}\right) + n \qquad \left[8 \cdot T\left(\tfrac{n}{8}\right) + 3n\right], \quad 8 = 2^3$$
$$\vdots$$
$$= \underbrace{\boxed{2^{\log n}}}_{n} \cdot \underbrace{\overset{1}{\boxed{T(1)}}}_{\frac{n}{2^{\log n}}} + \log n \cdot n = n + n \cdot \log n$$
$$= O(n \cdot \log n)$$