Lecture 20 - April 1

Binary Search Trees, Balanced BSTs

BST: Searching, Insertion
Hight Balance Property
Priority Queue: Introduction

Announcements/Reminders

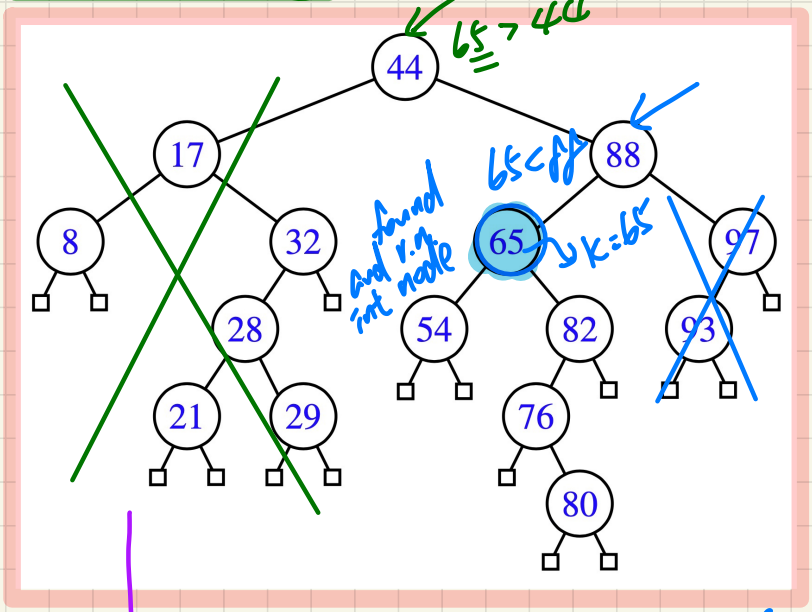
- Assignment 4 (on linked Trees) released
- Makeup Lecture (for ProgTest2) to be posted
- Bonus opportunity: Final Course Evaluation
- Office hours 3pm Tue/Wed/Thu this week
- Lecture notes template, Office Hours, TA Contact

* for a subsequent insertion, this ext. node can be set with a key value k s.t. $65 < k < 76$

BST Operation: Searching a Key

searching
① successful
↳ rtn. int. node with the matching key value
② unsuccessful
↳ rtn. ext. node with k: $65 < k < 76$

Search key 65

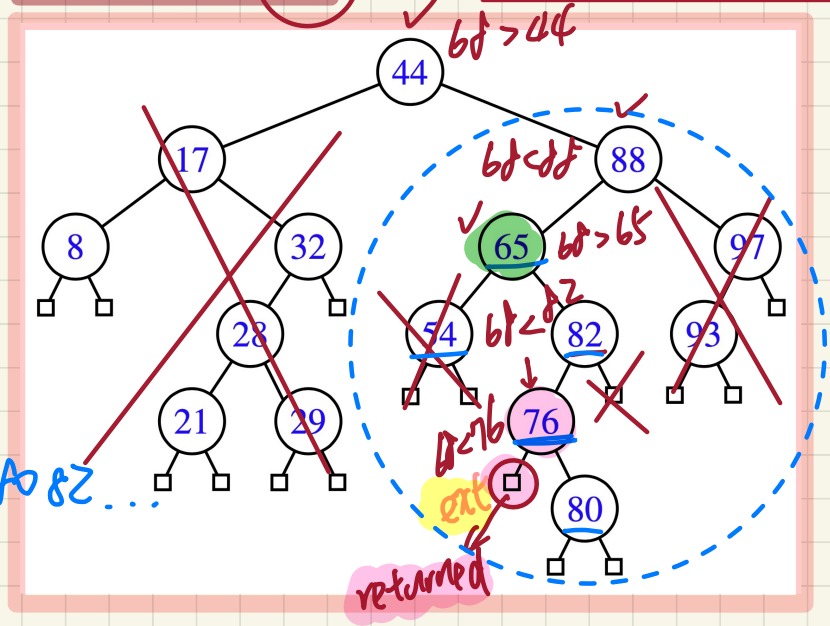


eliminate this subtree from search space (how large? 1/2 or O(1))

i.o.t. 54 65 76 A 82 ...

* ext.

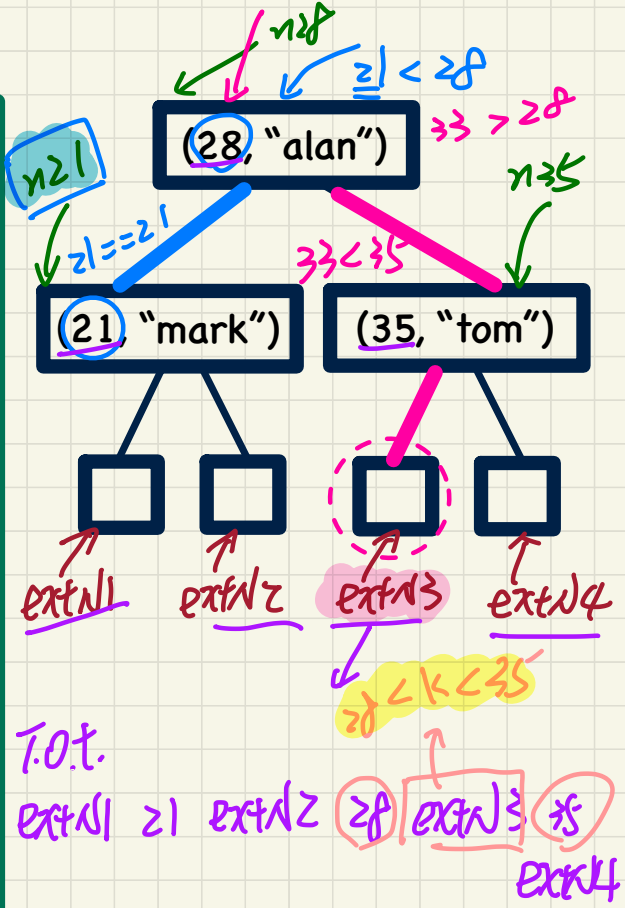
Search key 68



Tracing: Searching through a BST

```
@Test
public void test_binary_search_trees_search() {
  BSTNode<String> n28 = new BSTNode<>(28, "alan");
  BSTNode<String> n21 = new BSTNode<>(21, "mark");
  BSTNode<String> n35 = new BSTNode<>(35, "tom");
  BSTNode<String> extN1 = new BSTNode<>();
  BSTNode<String> extN2 = new BSTNode<>();
  BSTNode<String> extN3 = new BSTNode<>();
  BSTNode<String> extN4 = new BSTNode<>();
  n28.setLeft(n21); n21.setParent(n28);
  n28.setRight(n35); n35.setParent(n28);
  n21.setLeft(extN1); extN1.setParent(n21);
  n21.setRight(extN2); extN2.setParent(n21);
  n35.setLeft(extN3); extN3.setParent(n35);
  n35.setRight(extN4); extN4.setParent(n35);

  BSTUtilities<String> u = new BSTUtilities<>();
  /* search existing keys */
  assertTrue(n28 == u.search(n28, 28));
  assertTrue(n21 == u.search(n28, 21));
  assertTrue(n35 == u.search(n28, 35));
  /* search non-existing keys */
  assertTrue(extN1 == u.search(n28, 17)); /* *17* < 21 */
  assertTrue(extN2 == u.search(n28, 23)); /* 21 < *23* < 28 */
  assertTrue(extN3 == u.search(n28, 33)); /* 28 < *33* < 35 */
  assertTrue(extN4 == u.search(n28, 38)); /* 35 < *38* */
}
```



* Recall: given N nodes: ___ ≤ h ___

Running Time: Search on a BST

```
public BSTNode<E> search(BSTNode<E> p, int k) {
  BSTNode<E> result = null;
  if(p.isExternal()) {
    result = p; /* unsuccessful search */
  }
  else if(p.getKey() == k) {
    result = p; /* successful search */
  }
  else if(k < p.getKey()) {
    result = search(p.getLeft(), k);
  }
  else if(k > p.getKey()) {
    result = search(p.getRight(), k);
  }
  return result;
}
```

root

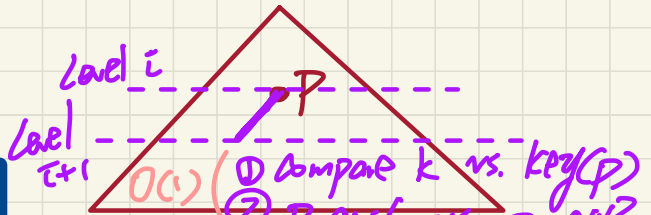
key to search

ext. node

ext. node

R1

R2



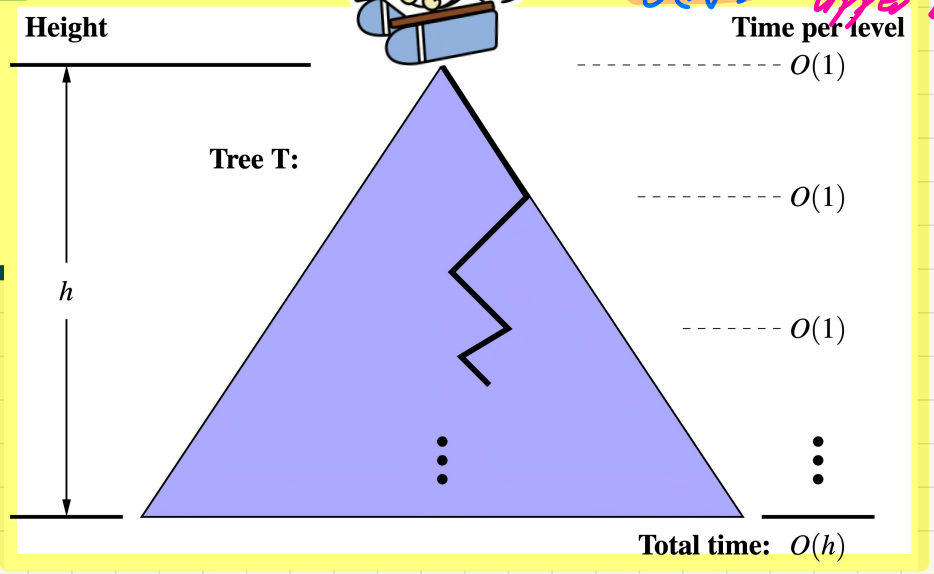
How many levels?

tight, attainable

(A1) * O(h)

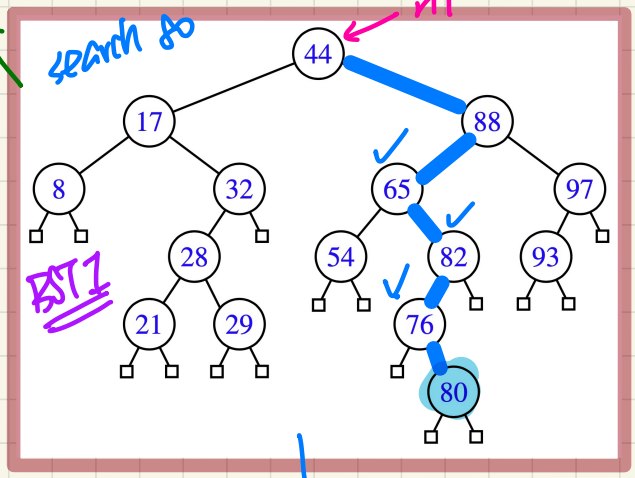
(A2) O(log N) → best case, lower bound

(A3) O(N) → worst case, upper bound

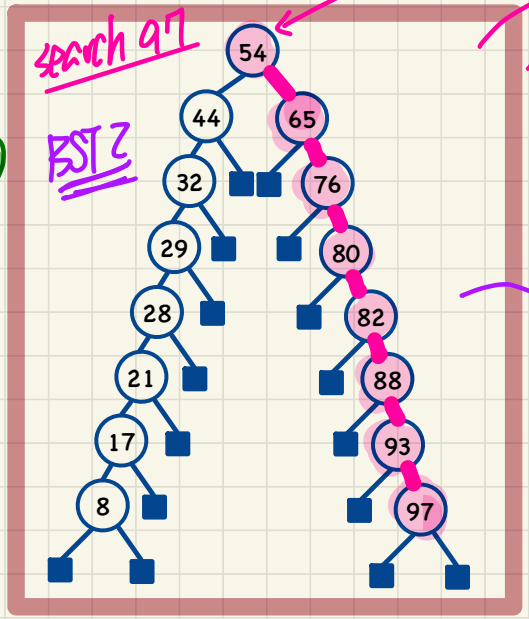


Binary Search: Non-Linear vs. Linear Structures

balanced.



N=15
h=4 ≈ O(log N)



unbalanced
N=15
h=7 ≈ N/2
O(N)

searching 97 on BST2 ≈ performing a linear search on a.

Search 80 on BST1 ≈ performing a binary search on a

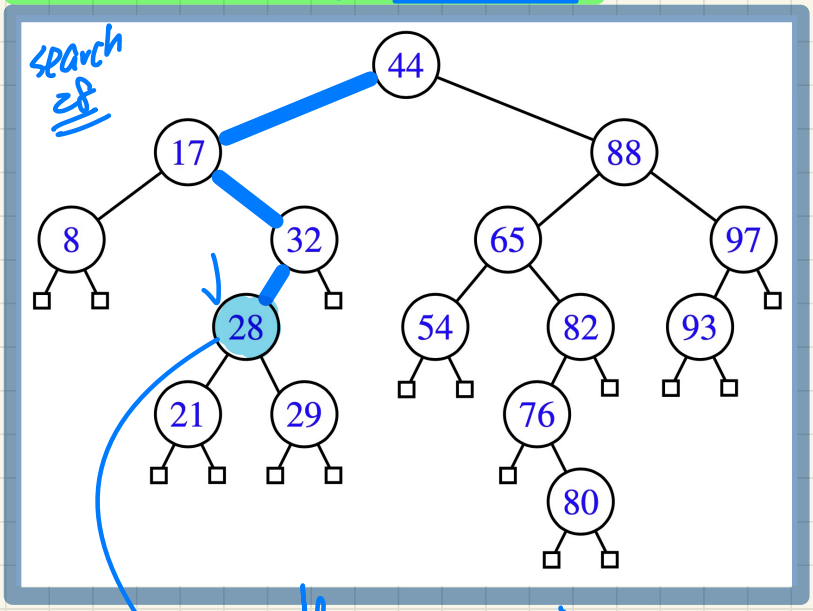
T.O.t.(n1) = T.O.t(n2) = a

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
8	17	21	28	29	32	44	54	65	76	80	82	88	93	97

REVIEW !

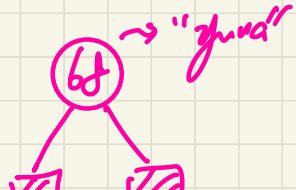
Visualizing BST Operation: Insertion

Insert Entry (28, "suyeon")

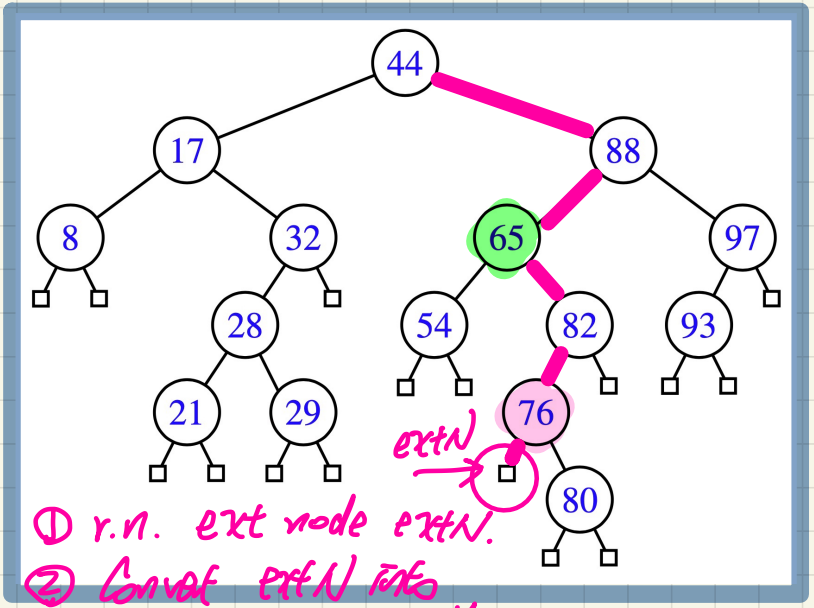


① r.n. int. node with key 28. node.setEle("suyeon");

②



Insert Entry (68, "yuna")



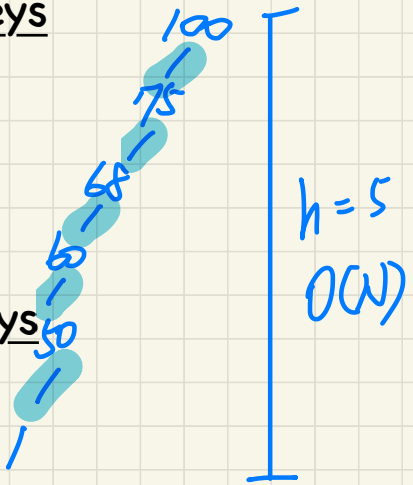
① r.n. ext node extN.

② Convert extN into

Worst-Case RT: BST with Linear Height

Example 1: Inserted Entries with Decreasing Keys

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



Example 2: Inserted Entries with Increasing Keys

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

entries (height? N vs. log N ?)

Example 3: Inserted Entries with In-Between Keys

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

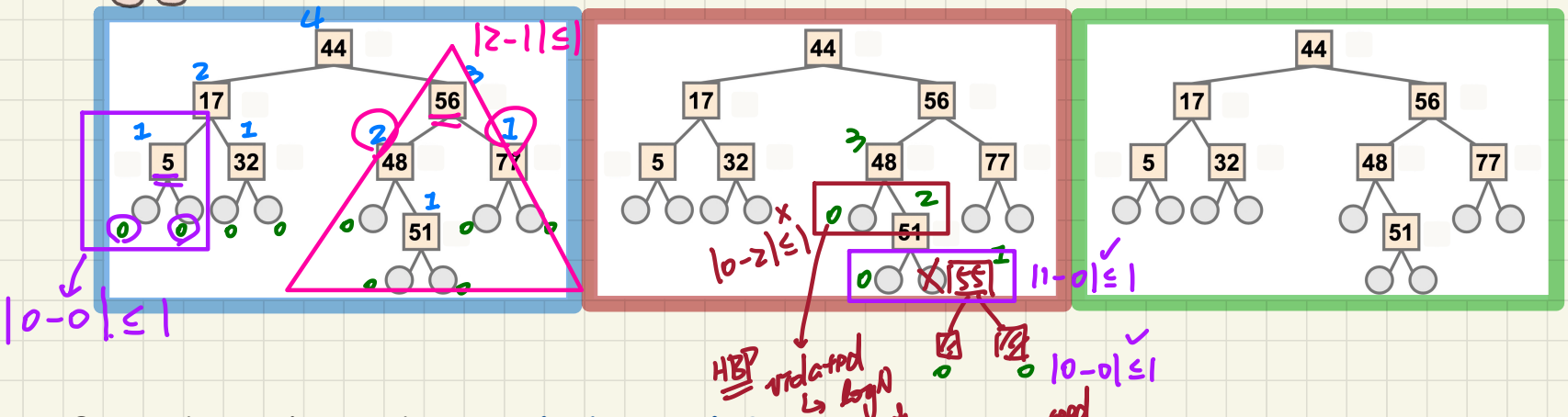
BST with O(N) height
⇒ search/insert/delete can be O(N).

Balanced BST: Definition

- internal node
- height
- height balance

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

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

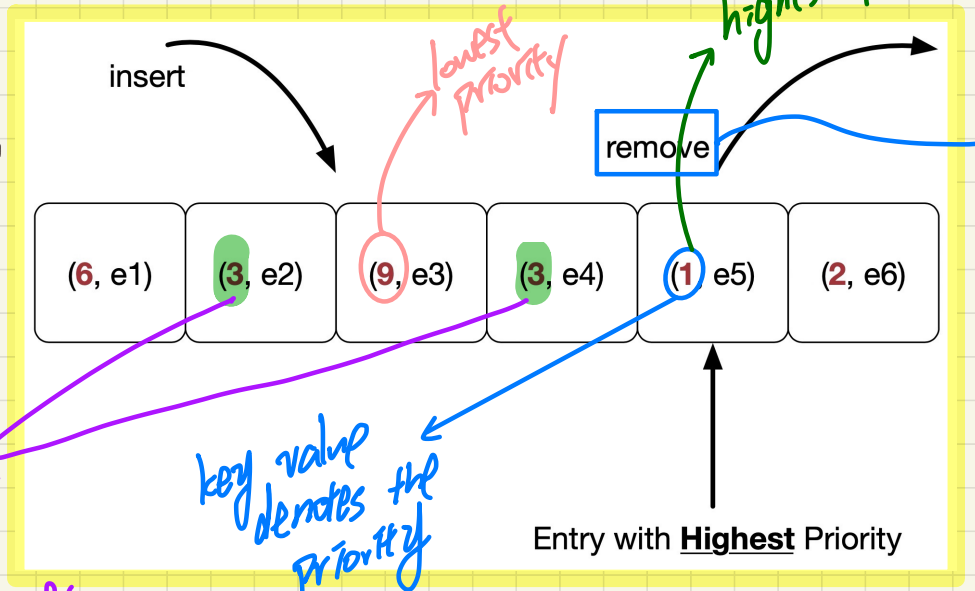


Q. Is the above tree a balanced BST?

Q. Still a balanced BST after inserting 55?

Q. Still a balanced BST after inserting 63?

What is a Priority Queue (PQ)



Entries with the same priority (does not matter which entry is chosen).

Compare PQ with FIFO queue

1. Entries removed from PQ according to priorities.
2. Entries removed from FIFO a.t. chronological order