Lecture 5b

Part B

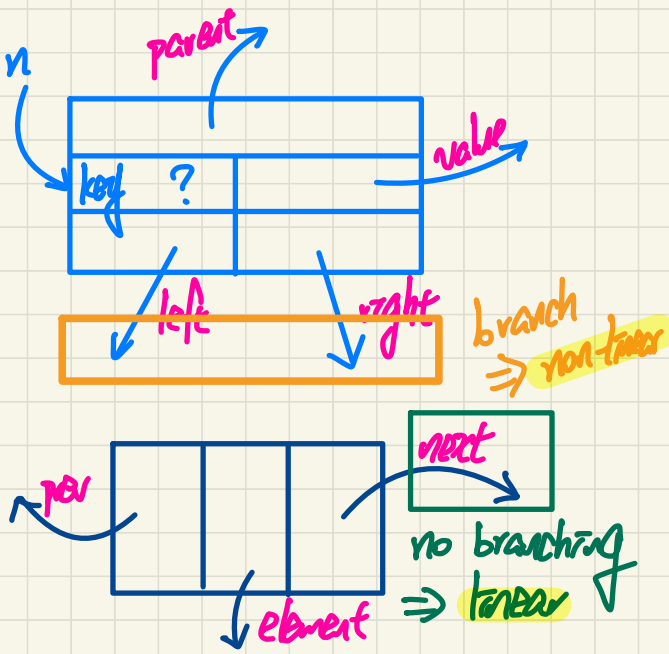
Binary Search Trees -
Implementing a Generic BST in Java
Tree Construction and Traversal

Generic, Binary Tree Nodes

```
public class BSTNode<E> {
  private int key; /* key */
  private E value; /* value */
  private BSTNode<E> parent; /* unique parent node */
  private BSTNode<E> left; /* left child node */
  private BSTNode<E> right; /* right child node */

  public BSTNode() { ... }
  public BSTNode(int key, E value) { ... }

  public boolean isExternal() {
    return this.getLeft() == null && this.getRight() == null;
  }
  public boolean isInternal() {
    return !this.isExternal();
  }
  public int getKey() { ... }
  public void setKey(int key) { ... }
  public E getValue() { ... }
  public void setValue(E value) { ... }
  public BSTNode<E> getParent() { ... }
  public void setParent(BSTNode<E> parent) { ... }
  public BSTNode<E> getLeft() { ... }
  public void setLeft(BSTNode<E> left) { ... }
  public BSTNode<E> getRight() { ... }
  public void setRight(BSTNode<E> right) { ... }
}
```



Compare:

+ prev ref.
+ next ref.

in a DLN.

Generic, Binary Tree Nodes - Traversal

```
import java.util.ArrayList;
public class BSTUtilities<E> {
  public ArrayList<BSTNode<E>> inOrderTraversal(BSTNode<E> root) {
    ArrayList<BSTNode<E>> result = null;
    if(root.isInternal()) {
      result = new ArrayList<>();

      if(root.getLeft().isInternal) {
        result.addAll(inOrderTraversal(root.getLeft()));
      }

      result.add(root);

      if(root.getRight().isInternal) {
        result.addAll(inOrderTraversal(root.getRight()));
      }
    }
    return result;
  }
}
```

otherwise, root. isExternal ⇒ return null

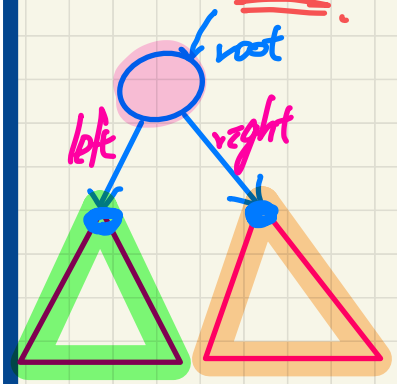
assumed to be the root of some BST

accumulate the returned result from LST

accumulate the returned result from RST

external node

↳ in-order tra. returns null.



Exercises: change to

1. BSTNode<E>[]
2. SLLNode<E>
3. pre-order
4. post-order

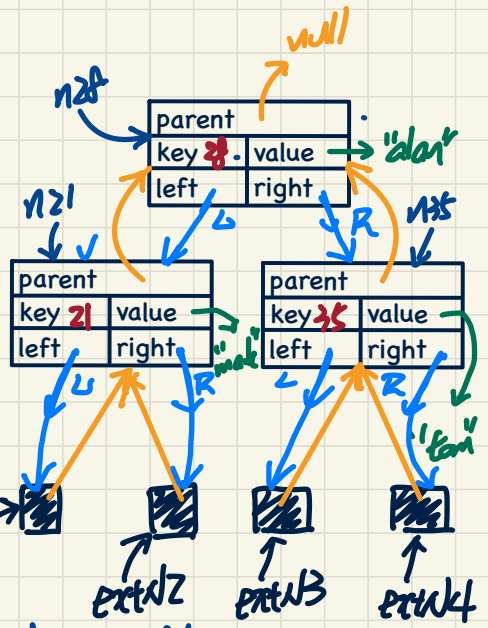
Tracing: Constructing and Traversing a BST

```java
@Test
public void test_binary_search_trees_construction() {
  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<>();
  ArrayList<BSTNode<String>> inOrderList = u.inOrderTraversal(n28);
  assertTrue(inOrderList.size() == 3);
  assertEquals(21, inOrderList.get(0).getKey());
  assertEquals("mark", inOrderList.get(0).getValue());
  assertEquals(28, inOrderList.get(1).getKey());
  assertEquals("alan", inOrderList.get(1).getValue());
  assertEquals(35, inOrderList.get(2).getKey());
  assertEquals("tom", inOrderList.get(2).getValue());
}
```

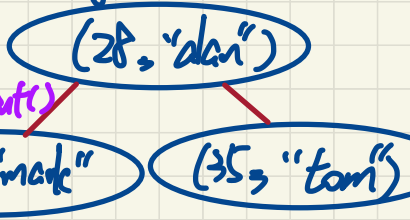
internal nodes

external nodes

sorting properties on keys.



alternatively



tracing

① n28 == n28.getLeft().getParent() T

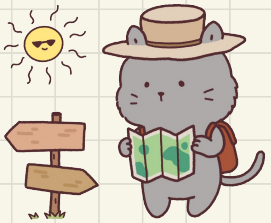
② n28.getLeft() == n28.getLeft().getRight().getParent() T

Lecture 5b

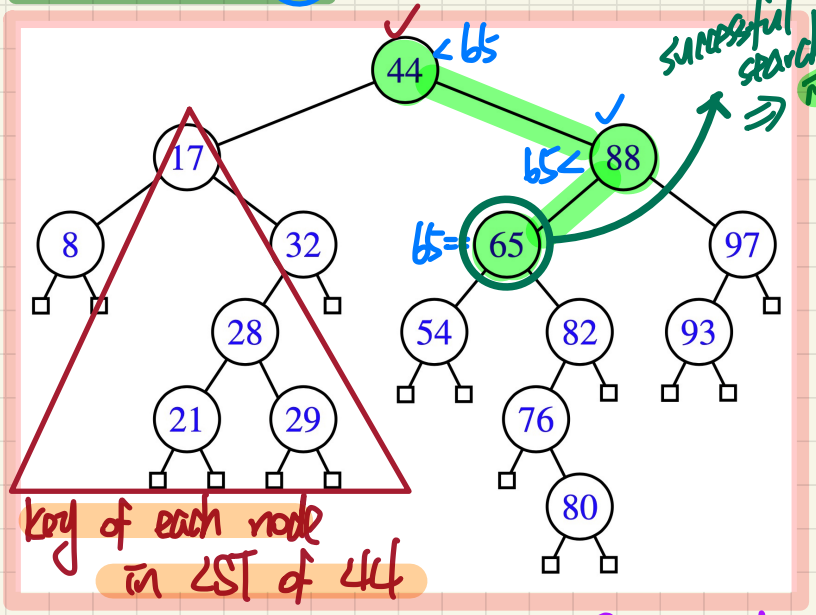
Part C

Binary Search Trees -
Implementing a Generic BST in Java
Searching

BST Operation: Searching a Key



Search key 65

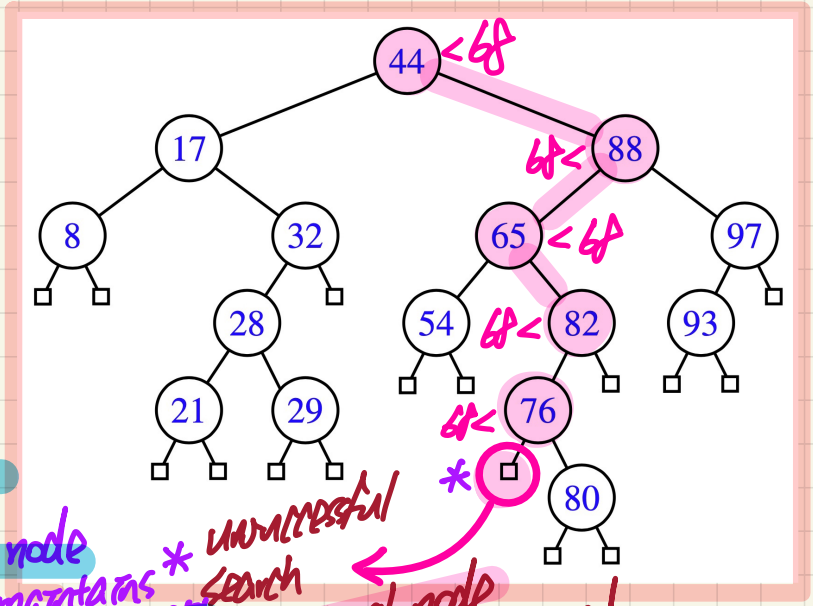


key of each node in LST of 44 must be < 44 (assumed search property)

successful search ⇒ external node storing 65 returned

Assumption: BST ⇒ search property

Search key 68



* Storing this returned ext. node with key 68 maintains the search property ⇒ unsuccessful search external node returned

Tracing: Searching through a BST

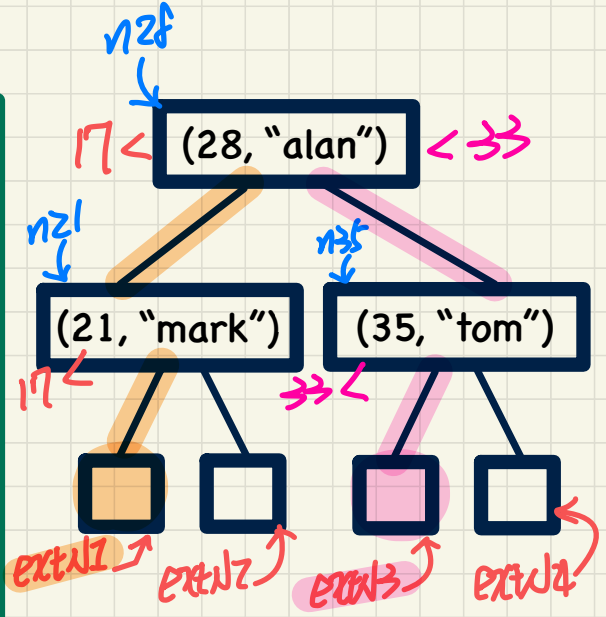
```
@Test
public void test_binary_search_trees_search() {
  BSTNode<String> n28 = new BSTNode<>(28, "alan");
  BSTNode<String> n21 = new BSTNode<>(21, "mark");
  BSTNode<String> n35 = new BSTNode<>(35, "tom");
  BSTNode<String> extN1 = new BSTNode<>();
  BSTNode<String> extN2 = new BSTNode<>();
  BSTNode<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* */
}
```

internal nodes -> successful search

external nodes -> unsuccessful search.

Exercise



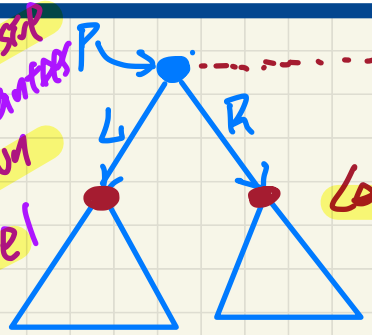
Running Time: Search on a BST

```
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;
}
```

external node

internal node

Each recursive call ~~guarantees~~ moves down by 1 level



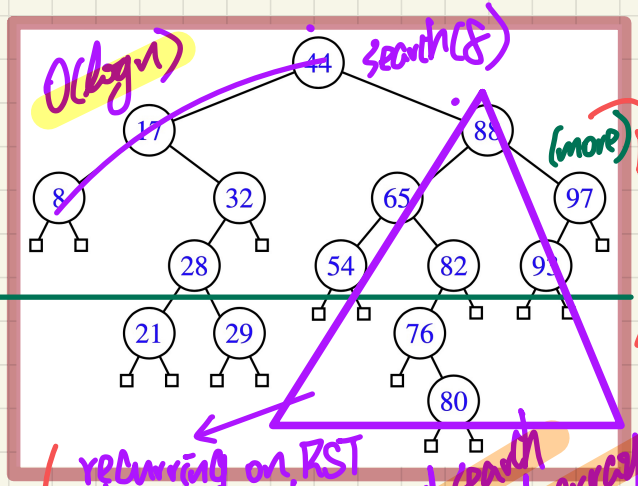
correct: O(h)

RT

O(n) X ∵ RT less if the BST is balanced

O(log n) X ∵ RT more if the BST is ill-balanced

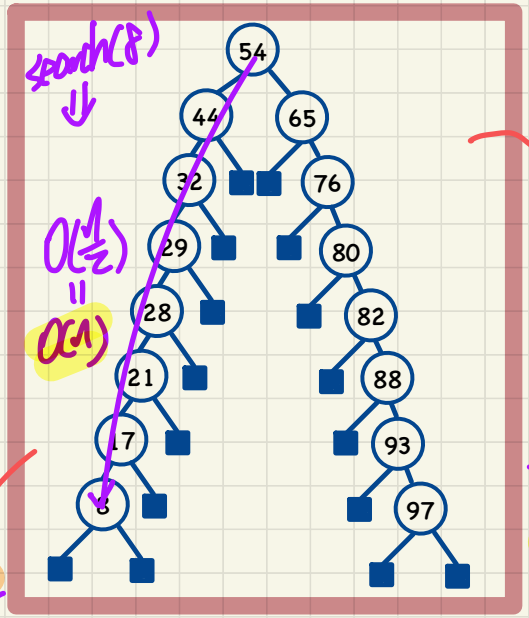
Binary Search: Non-Linear vs. Linear Structures



balanced
$h \approx \log n$
$\log 15 = 3...$

(more)

recurring on RST reduces the search space by half.



→ ill-balanced

O(log n)

half guarantee the search space is reduced by half.

sorted array (in-order traversal of BST) half.

A binary search on a sorted array is equivalent to as far as RT is concerned a search on a balanced BST.

recurring on right

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!

Lecture 5b

Part D

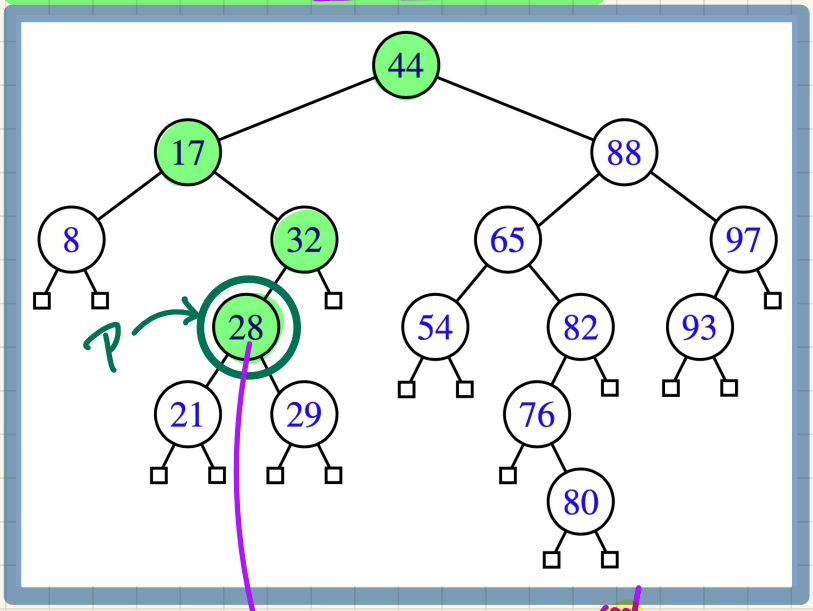
Binary Search Trees -
Implementing a Generic BST in Java
Insertion

Visualizing BST Operation: Insertion

search key

data value (not for searching)

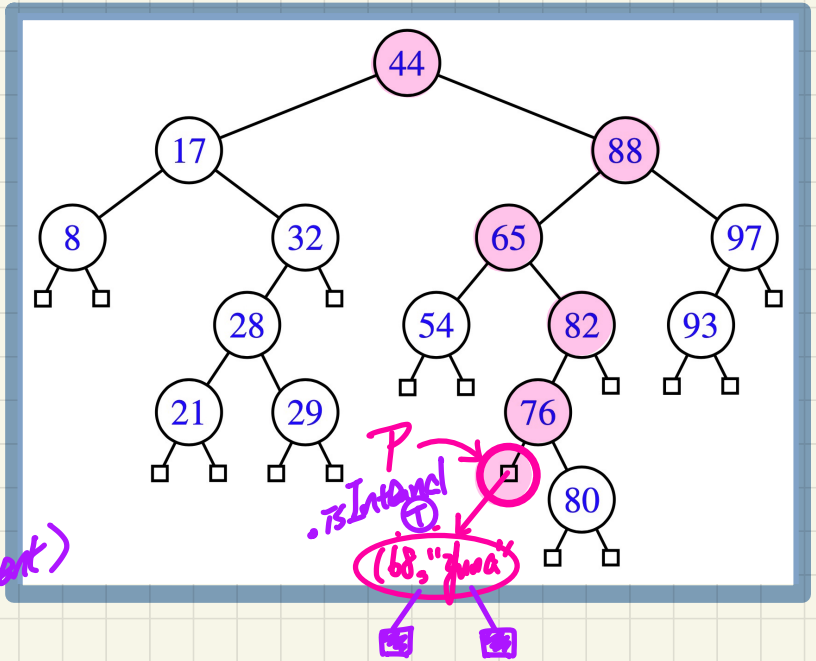
Insert Entry (28, "suyeon")



Replace whatever value that's associated with key 28 by "suyeon" (e.g. setElement)

O(h) ~ height of tree

Insert Entry (68, "yuna")



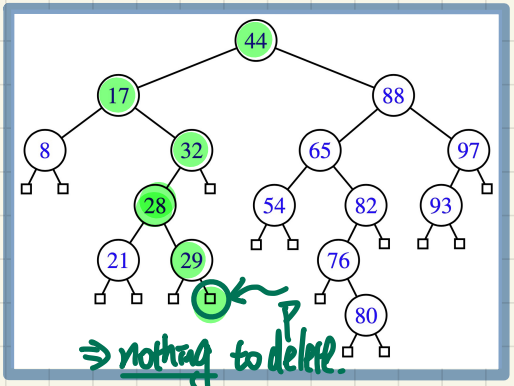
Lecture 5b

Part E

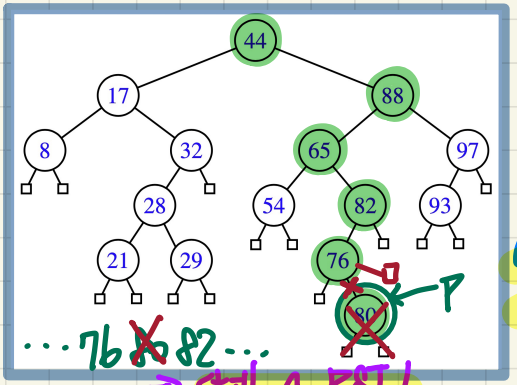
Binary Search Trees -
Implementing a Generic BST in Java
Deletion

Visualizing BST Operation: Deletion

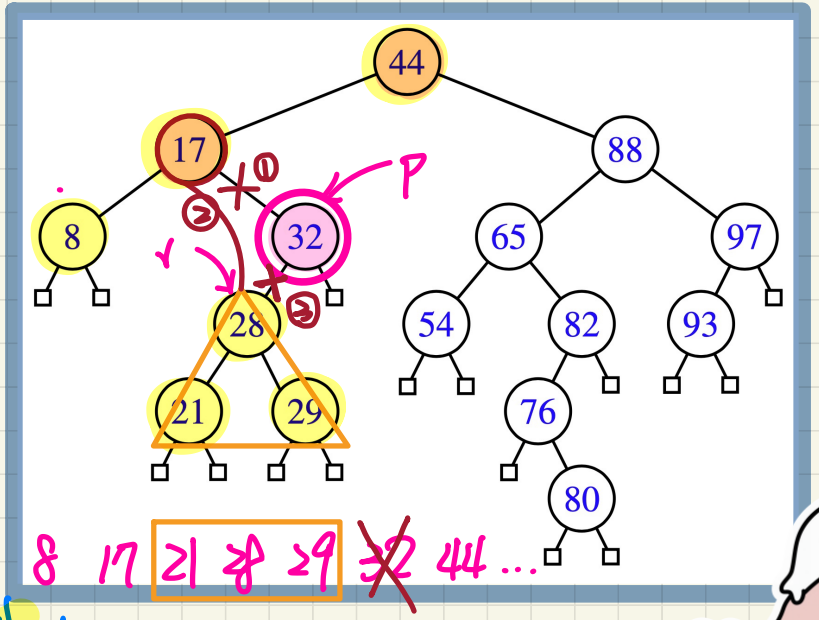
Case 1: Delete Entry with Key 31



Case 2: Delete Entry with Key 80



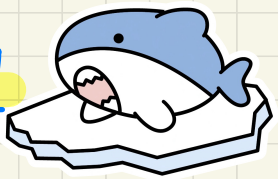
Case 3: Delete Entry with Key 32

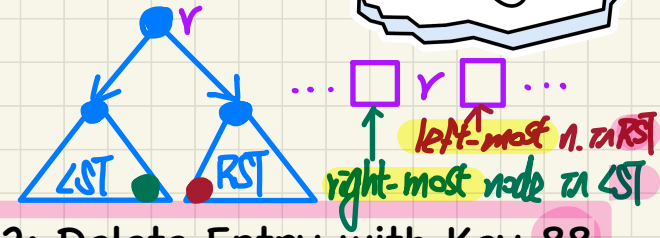


Orphan subtree: a subtree rooted at an external child node

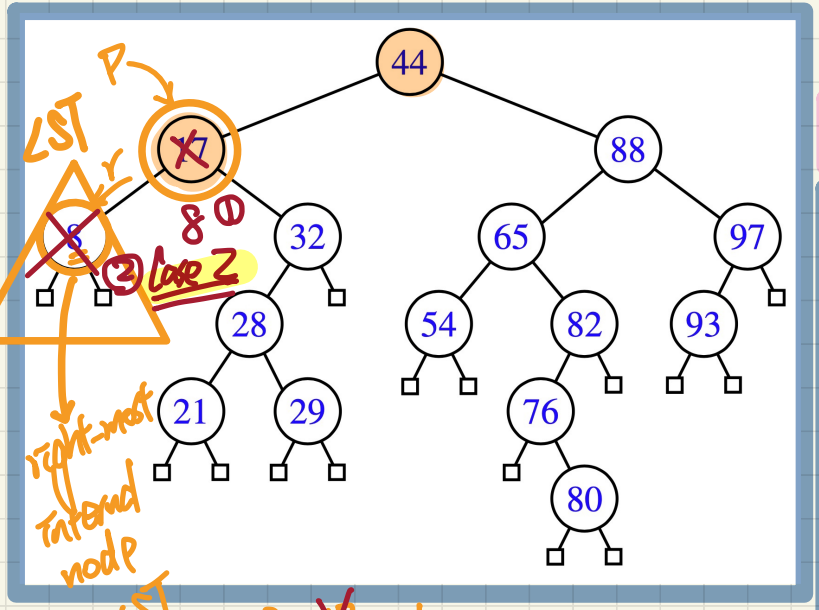
Visualizing BST Operation: Deletion

In-Order Traversal

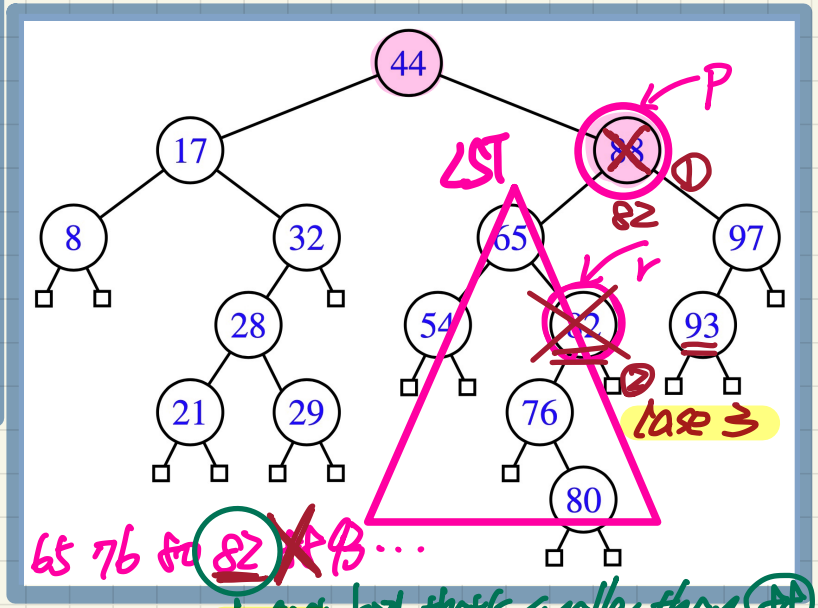




Case 4.1: Delete Entry with Key 17



Case 4.2: Delete Entry with Key 88

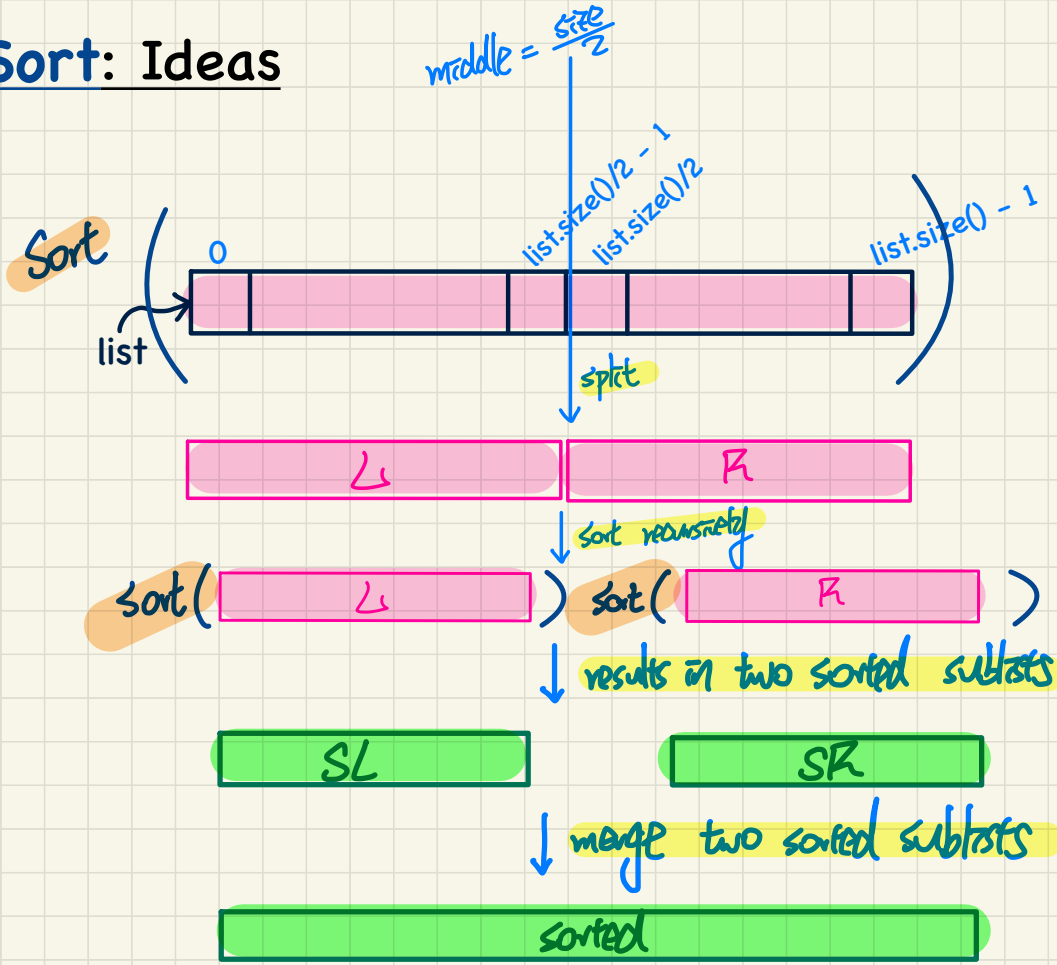


Lecture 4

Part C

Examples on Recursion
Merge Sort

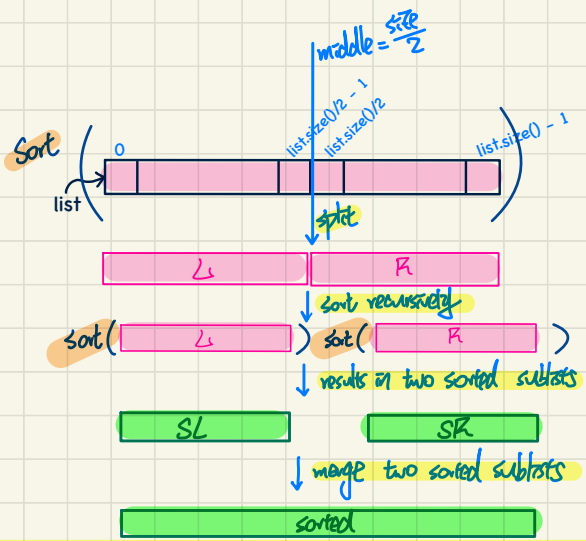
Merge Sort: Ideas

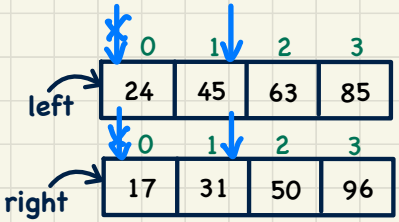


Merge Sort in Java

```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

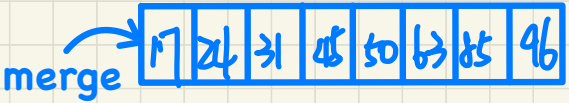




Precondition

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;
}
```



Merge Sort: Tracing

split

merge