61B Lecture #21: Tree Searching

Divide and Conquer

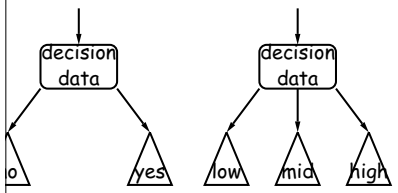
) computation is devoted to finding things in response rms of query.

h for response can be expensive, especially when data ge for primary memory.

o have criteria for *dividing* data to be searched into sively

figure for $\lg N$ algorithms: at 1 μsec per comparison, s 10^{300000} items in 1 sec.

ural framework for the representation:



Binary Search Trees

Property:

ontain *keys,* and possibly other data.

eft subtree of node have *smaller* keys.

ight subtree of node have *larger* keys.

ans any complete transitive, anti-symmetric ordering on

e of $x \prec y$ and $y \prec x$ true.

$y \prec z$ imply $x \prec z$.

fy, won't allow duplicate keys this semester).

nary database, node label would be (*word, definition*): key.

ness here, we'll just use the standard Java convention mpareTo.

A Binary Search Type

se the following simple binary search tree type. Ignore violations, please.

```
 of a binary search tree associating a value of type VALUE
 key of type KEY.  (Thus, the labels in this tree are
lue pairs.) */
Key extends Comparable<Key>, Value> {
y;
value;
y, Value> left, right;

y key0, Value value0,
T<Key, Value> left0, BST<Key, Value> right0) {
dy left to the reader.

y key0, Value value0) {
is(key0, value0, null, null);
```

Key extends Comparable<Key> stuff for now. It just ys (of type Key) can be compared to each other.)

Finding

r 50 and 49:


```
/** Return node in T containing L.  Null if none. */
static <Key extends Comparable<Key>, Value>
  BST<Key, Value> find(BST<Key, Value> T, Key L) {
    if (T == null)
        return T;
    if (L.compareTo(T.key) == 0)
        return T;
    else if (L.compareTo(T.key) < 0)
        return find(T.left, L);
    else
        return find(T.right, L);
}
```

s show which node labels we look at.

odes examined is proportional to height of tree.

Inserting


```
/** Insert V in T with key K, replacing existing
 *  value if present. Return the modified tree. */
static <Key extends Comparable<Key>, Value>
  BST<Key, Value> insert(BST<Key, Value> T,
                         Key K, Value V) {
    if (T == null)
        return new BST(K, V);
    if (K.compareTo(T.key) == 0)
        T.value = V;
    else if (K.compareTo(T.key) < 0)
        T.left = insert(T.left, K, V);
    else
        T.right = insert(T.right, K, V);
    return T;
}
```

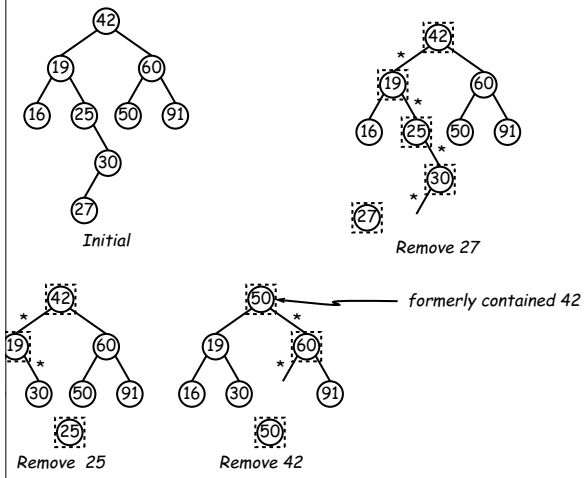
es are set (to themselves, unless initially null).

roportional to height.

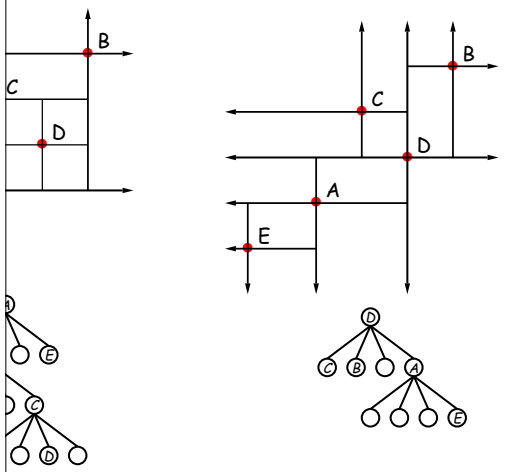
Deletion Algorithm

```
n T, and return the new tree. */
nds Comparable<Key>, Value>
> remove(BST T, Key K) {

l)
ull;
eTo(T.key) == 0) {
ft == null)
rn T.right;
(T.right == null)
rn T.left;

Key, Value> smallest = minNode(T.right);  // ??
lue = smallest.value;
y = smallest.key;
ght = remove(T.right, smallest.key);


ompareTo(T.key) < 0)
 remove(T.left, K);

= remove(T.right, K);
```


Deletion



Classical Quadtree: Example



ore Than Two Choices: Quadtrees

ex information about 2D locations so that items can be position. But how to compare positions "binarily?"

o so using the same standard data-structuring trick as *e and Conquer*—but with more subtrees.

(2D) space into four *quadrants,* and store items in the quadrant. Repeat this recursively with each quadrant s more than one item.

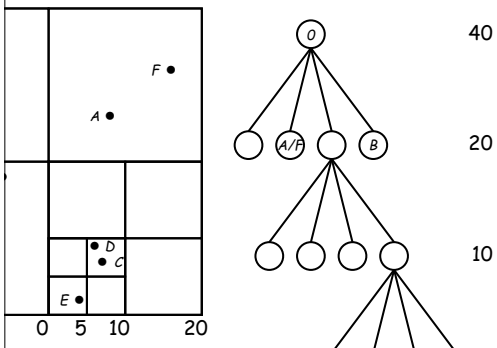
nition: a quadtree is either

t some position (x, y), called the root, plus

trees, each containing only items that are northwest, , southwest, and southeast of (x, y).

at if you are looking for point (x', y') and the root is not are looking for, you can narrow down which of the four the root to look in by comparing coordinates (x, y) with

Example of PR Quadtree



oints per leaf)

Point-region (PR) Quadtrees

Quadtree to track moving objects, it may be useful to *lete* items from a tree: when an object moves, the it goes in may change.

do with the classical data structure above, so we'll de-

consists of a bounding rectangle, B and either

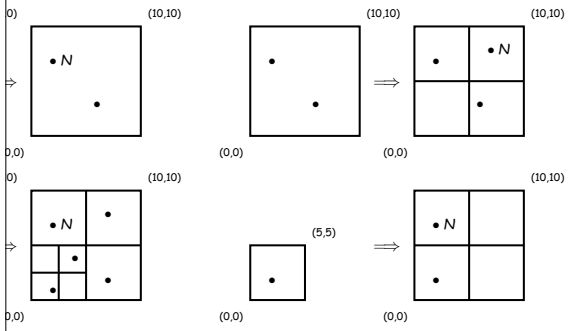
a small number of items that lie in that rectangle, or

trees whose bounding rectangles are the four quadrants f equal size).

empty quadtree can have an arbitrary bounding rect- can wait for the first point to be inserted.

Insertion into PR Quadtrees

r inserting a new point N, assuming maximum occupancy
showing initial state $\Longrightarrow$ final state.



Navigating PR Quadtrees

em at (x, y) in quadtree T,

s outside the bounding rectangle of T, or T is empty,
) is not in T.

e, if T contains a small set of items, then (x, y) is in T
nong these items.

e, T consists of four quadtrees. Recursively look for
ach (however, step #1 above will cause all but one of
nding boxes to reject the point immediately).

edure works when looking for all items within some rect-

s not intersect the bounding rectangle of T, or T is
en there are no items in R.

e, if T contains a set of items, return those that are in

e, T consists of four quadtrees. Recursively look for
R in each one of them.