# 61B Lecture #21: Tree Searching



## Public Service Announcement

ill be a grad school panel hosted by the AWE and
Thursday, October 12, 5-7PM in 540AB Cory. The
e advice to women regarding how to get into grad
ne research experience required."



## 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

ne 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.



## Divide and Conquer

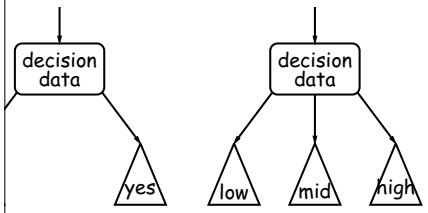
) computation is devoted to finding things in response
rms of query.

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

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

gure for $\lg N$ algorithms: at 1$\mu$sec per comparison, could
000 items in 1 sec.

ural framework for the representation:






## Inserting

```
/** Insert L in T, replacing existing
 *  value if present, and returning
 *  new tree. */
static BST insert(BST T, Key L) {
  if (T == null)
    return new BST(L);
  if (L.compareTo(T.label()) == 0)
    T.setLabel(L);
  else if (L.compareTo(T.label()) < 0)
    T.setLeft(insert(T.left(), L));
  else
    T.setRight(insert(T.right(), L));
  return T;
}
```


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

proportional to height.



## Finding

r 50 and 49:

```
/** Node in T containing L, or null if none */
static BST find(BST T, Key L) {
  if (T == null)
    return T;
  if (L.compareTo(T.label()) == 0)
    return T;
  else if (L.compareTo(T.label()) < 0)
    return find(T.left(), L);
  else
    return find(T.right(), L);
}
```


s show which node labels we look at.

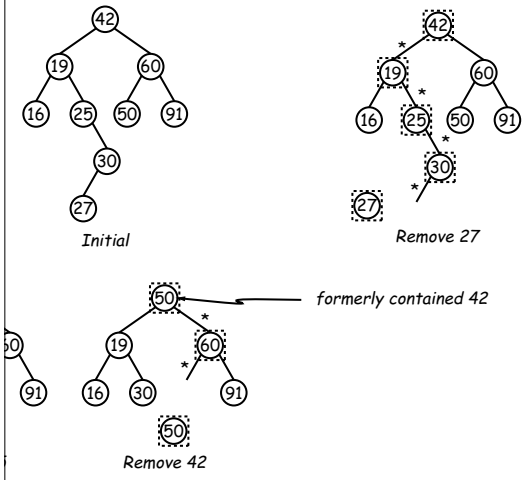
ed at proportional to height of tree.

## Deletion Algorithm

```
/** Remove L from T, returning new tree. */
static BST remove(BST T, Key L) {
  if (T == null)
    return null;
  if (L.compareTo(T.label()) == 0) {
     if (T.left() == null)
         return T.right();
     else if (T.right() == null)
         return T.left();
     else {
         Key smallest = minVal(T.right());  // ??
         T.setRight(remove(T.right(), smallest));
         T.setLabel(smallest);
     }
  }
  else if (L.compareTo(T.label()) < 0)
    T.setLeft(remove(T.left(), L));
  else
    T.setRight(remove(T.right(), L));
  return T;
}
```

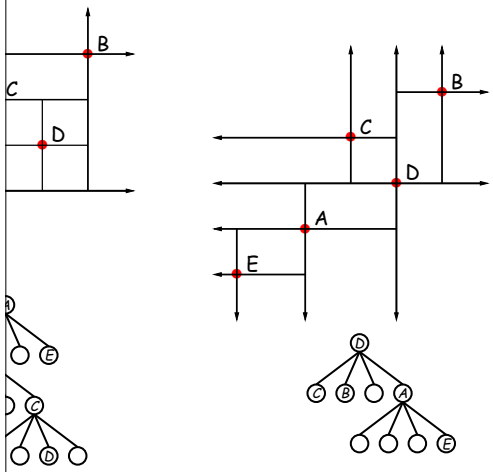


## Deletion






## Classical Quadtree: Example






## ore Than Two Choices: Quadtrees

ex information about 2D locations so that items can be position.

lo so using standard data-structuring trick: *Divide and*

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

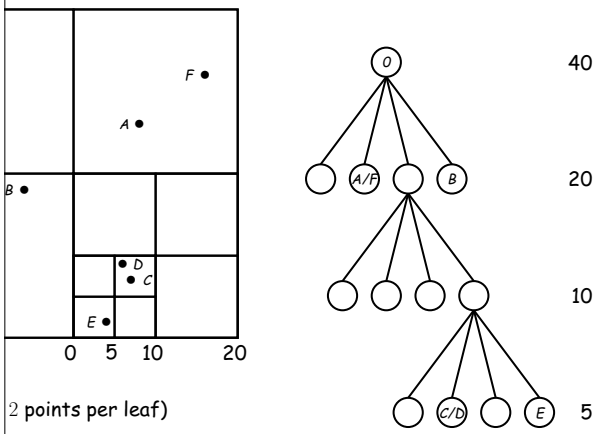
nition: a quadtree is either

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




2 points per leaf)



## Point-region (PR) Quadtrees

QuadTree to track moving objects, it may be useful to elete 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

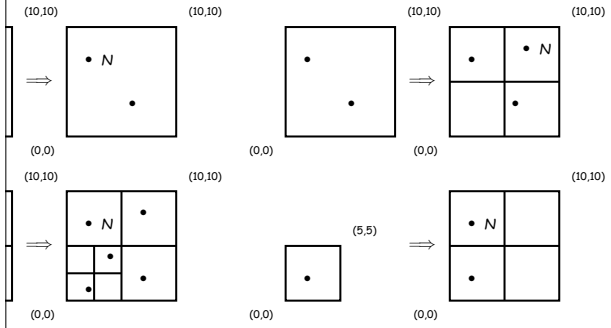
a small number of items that lie in that rectangle, or

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

or 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.