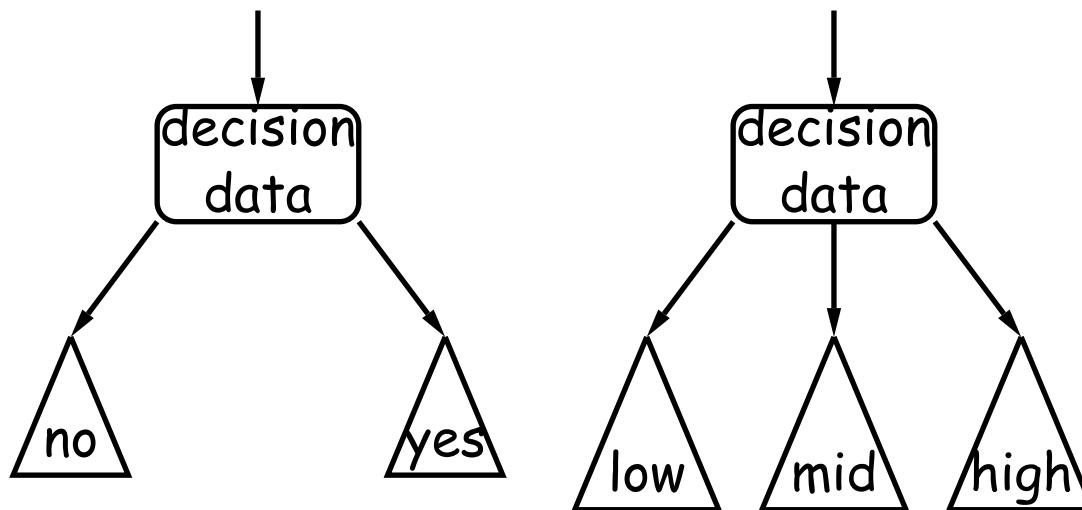


# CS61B Lecture #21: Tree Searching

# Divide and Conquer

- Much (most?) computation is devoted to finding things in response to various forms of query.
- Linear search for response can be expensive, especially when data set is too large for primary memory.
- Preferable to have criteria for *dividing* data to be searched into pieces recursively
- We saw the figure for  $\lg N$  algorithms: at  $1 \mu\text{sec}$  per comparison, could process  $10^{300000}$  items in 1 sec.
- Tree is a natural framework for the representation:



# Binary Search Trees

## Binary Search Property:

- Tree nodes contain *keys*, and possibly other data.
- All nodes in left subtree of node have *smaller* keys.
- All nodes in right subtree of node have *larger* keys.
- “Smaller” means any complete transitive, anti-symmetric ordering on keys:
  - exactly one of  $x \prec y$  and  $y \prec x$  true.
  - $x \prec y$  and  $y \prec z$  imply  $x \prec z$ .
  - (To simplify, won't allow duplicate keys this semester).
- E.g., in dictionary database, node label would be (*word, definition*): *word* is the key.
- For concreteness here, we'll just use the standard Java convention of calling `.compareTo`.

# A Binary Search Type

- Here, we'll use the following simple binary search tree type. Ignore all the style violations, please.

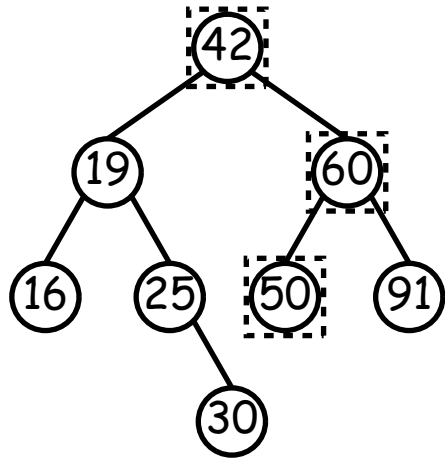
```
/** A node of a binary search tree associating a value of type VALUE
 * with a key of type KEY. (Thus, the labels in this tree are
 * key/value pairs.) */
class BST<Key extends Comparable<Key>, Value> {
    Key key;
    Value value;
    BST<Key, Value> left, right;

    BST(Key key0, Value value0,
        BST<Key, Value> left0, BST<Key, Value> right0) {
        Body left to the reader.
    }
    BST(Key key0, Value value0) {
        this(key0, value0, null, null);
    }
}
```

- (Ignore the `Key extends Comparable<Key>` stuff for now. It just says that keys (of type `Key`) can be compared to each other.)

# Finding

- Searching for 50 and 49:

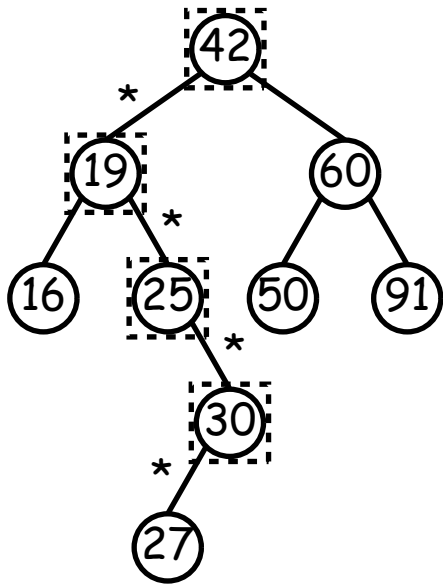


```
/** Return node in T containing L. Null if none. */
static <Key extends Comparable<Key>, Value>
BST<Key, Value> find(BST<Key, Value> T, Key L) {
    if (T == null)
        return T;
    if (L.compareTo(T.key) == 0)
        return T;
    else if (L.compareTo(T.key) < 0)
        return find(T.left,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

- Dashed boxes show which node labels we look at.
- Number of nodes examined is proportional to height of tree.

# Inserting

- Inserting 27

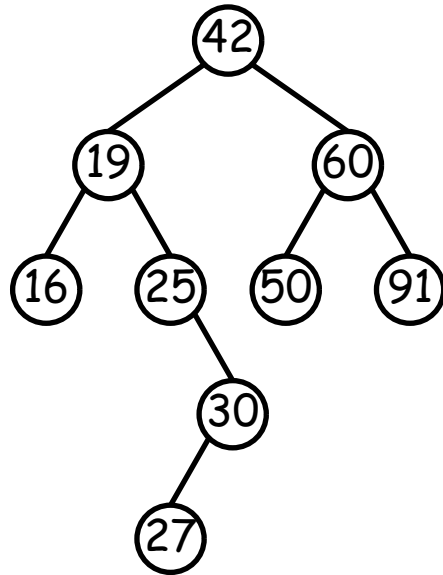


```
/** Insert V in T with key K, replacing existing  
 * value if present. Return the modified tree. */
```

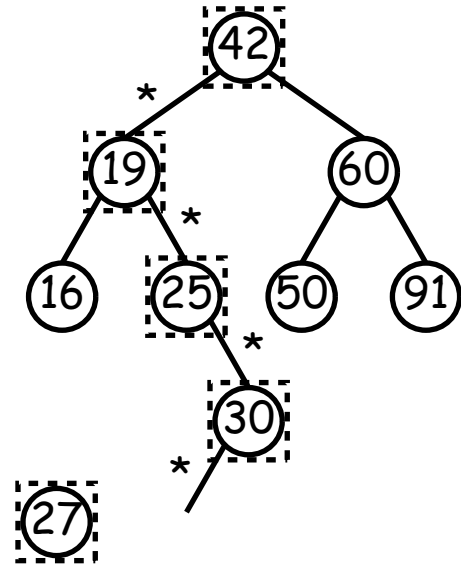
```
static <Key extends Comparable<Key>, Value>  
BST<Key, Value> insert(BST<Key, Value> T,  
                       Key K, Value V) {  
    if (T == null)  
        return new BST(K, V);  
    if (K.compareTo(T.key) == 0)  
        T.value = V;  
    else if (K.compareTo(T.key) < 0)  
        T.left = insert(T.left, K, V);  
    else  
        T.right = insert(T.right, K, V);  
    return T;  
}
```

- Starred edges are set (to themselves, unless initially null).
- Again, time proportional to height.

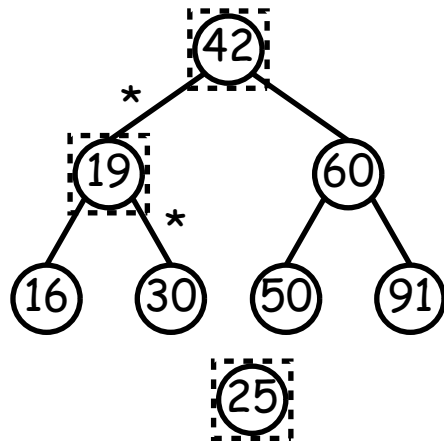
# Deletion



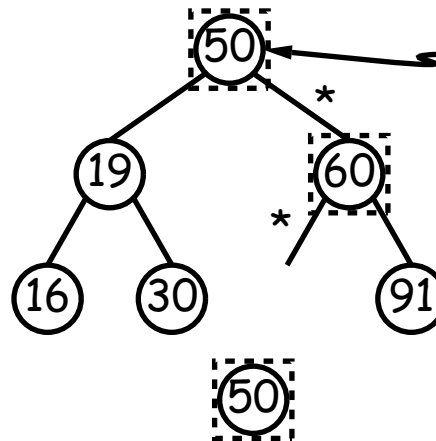
Initial



Remove 27



Remove 25



Remove 42

← formerly contained 42

# Deletion Algorithm

```
/** Remove K from T, and return the new tree. */
static <Key extends Comparable<Key>, Value>
    BST<Key, Value> remove(BST T, Key K) {

    if (T == null)
        return null;
    if (K.compareTo(T.key) == 0) {
        if (T.left == null)
            return T.right;
        else if (T.right == null)
            return T.left;
        else {
            BST<Key, Value> smallest = minNode(T.right); // ??
            T.value = smallest.value;
            T.key = smallest.key;
            T.right = remove(T.right, smallest.key);
        }
    }
    else if (K.compareTo(T.key) < 0)
        T.left = remove(T.left, K);
    else
        T.right = remove(T.right, K);
    return T;
}
```



## Minimum Node in a Subtree

```
/** Return the node containing the minimal key T.  
 * T must not be null. */  
public static <Key extends Comparable<Key>, Value>  
    BST<Key, Value> minNode(BST<Key, Value> T) {  
  
    // ???  
  
    return T;  
}
```

- This is a *generic function*. As before, let's ignore all the "<Key extends Comparable...>" stuff for now.

## Minimum Node in a Subtree

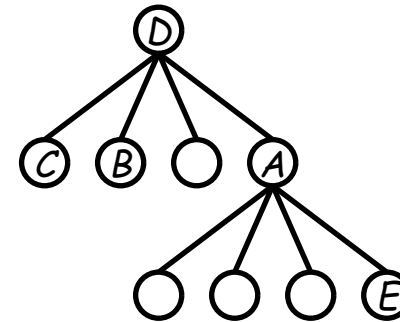
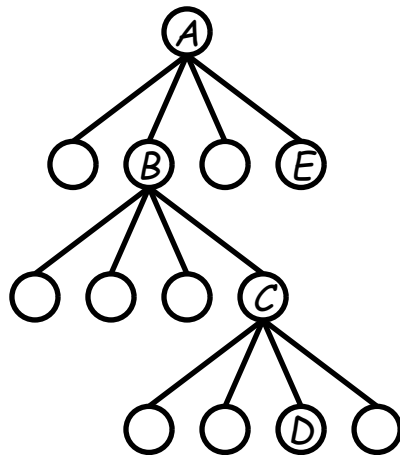
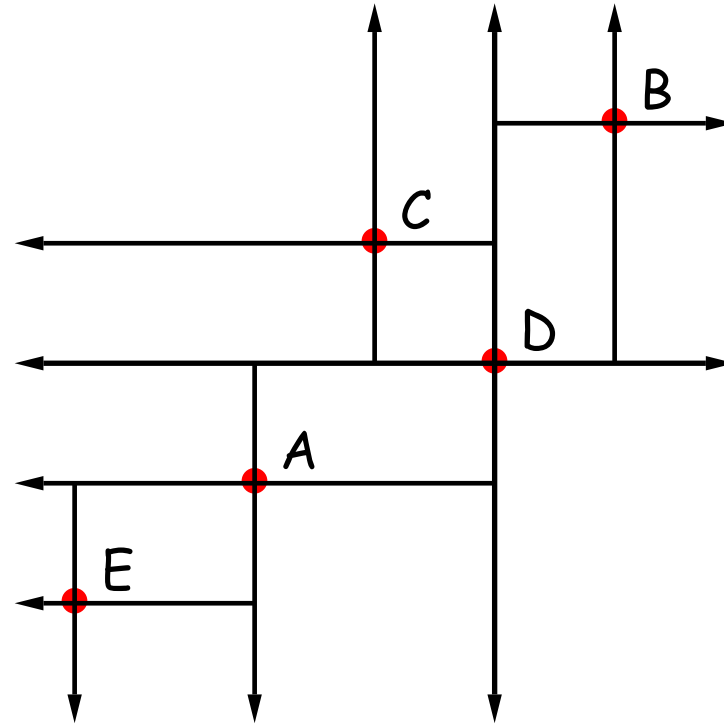
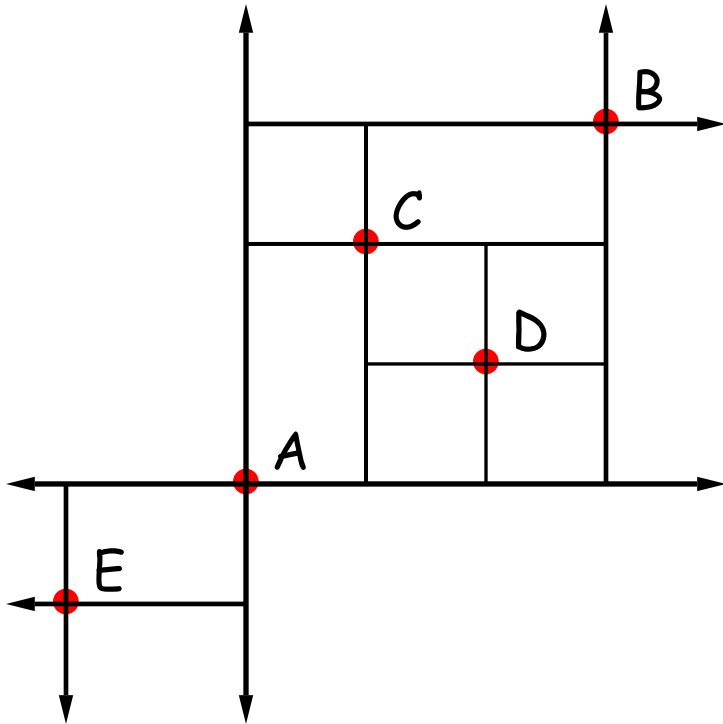
```
/** Return the node containing the minimal key T.  
 * T must not be null. */  
public static <Key extends Comparable<Key>, Value>  
    BST<Key, Value> minNode(BST<Key, Value> T) {  
  
    while (T.left != null) {  
        T = T.left;  
    }  
  
    return T;  
}
```

- This is a *generic function*. As before, let's ignore all the "<Key extends Comparable...>" stuff for now.

# More Than Two Choices: Quadtrees

- Want to *index* information about 2D locations so that items can be retrieved by position. But how to compare positions “binarily?”
- *Quadtrees* do so using the same standard data-structuring trick as BSTs—*Divide and Conquer*—but with more subtrees.
- Idea: divide (2D) space into four *quadrants*, and store items in the appropriate quadrant. Repeat this recursively with each quadrant that contains more than one item.
- Original definition: a quadtree is either
  - Empty, or
  - An item at some position  $(x, y)$ , called the root, plus
  - four quadtrees, each containing only items that are northwest, northeast, southwest, and southeast of  $(x, y)$ .
- Big idea is that if you are looking for point  $(x', y')$  and the root is not the point you are looking for, you can narrow down which of the four subtrees of the root to look in by comparing coordinates  $(x, y)$  with  $(x', y')$ .

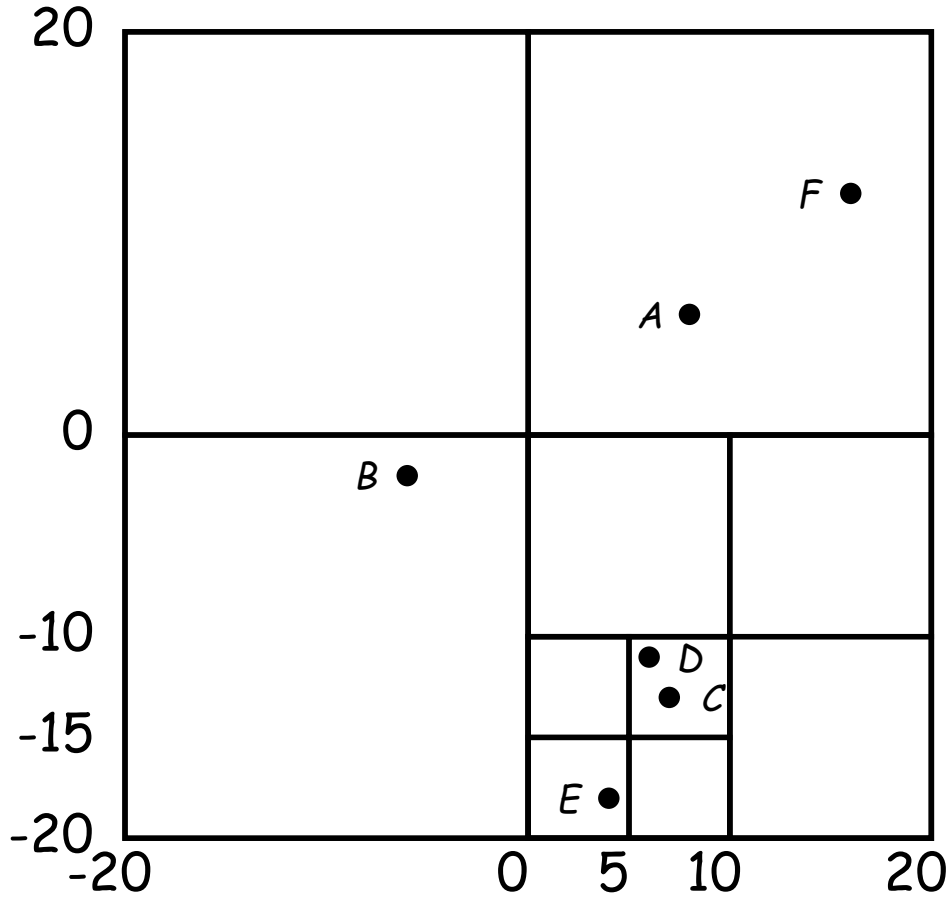
# Classical Quadtree: Example



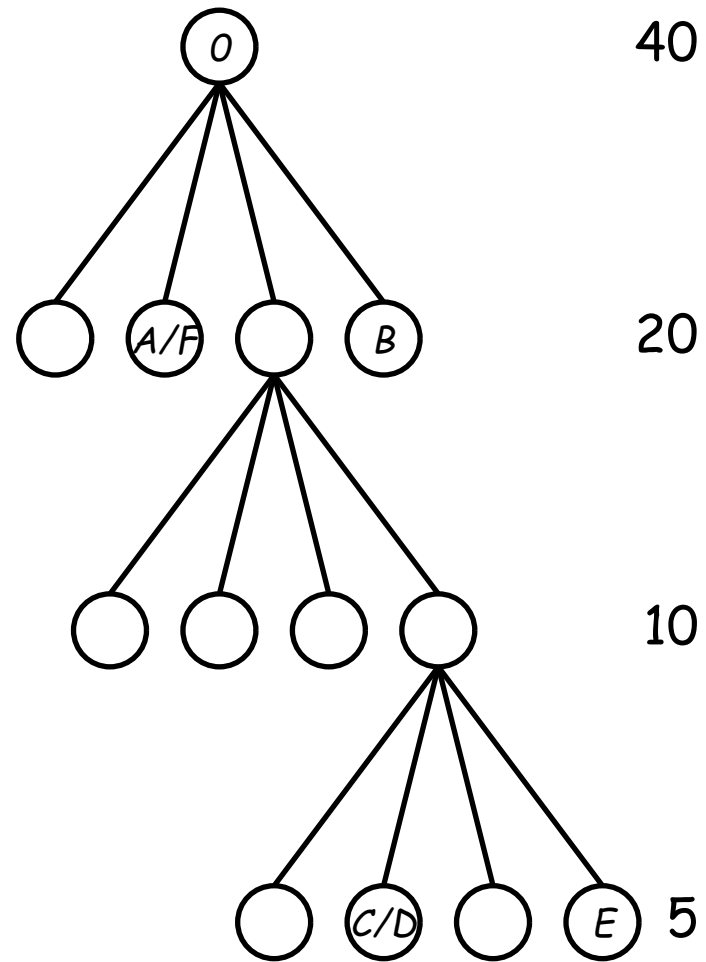
# Point-region (PR) Quadtrees

- If we use a Quadtree to track moving objects, it may be useful to be able to *delete* items from a tree: when an object moves, the subtree that it goes in may change.
- Difficult to do with the classical data structure above, so we'll define instead:
- A quadtree consists of a bounding rectangle,  $B$  and either
  - Zero up to a small number of items that lie in that rectangle, or
  - Four quadtrees whose bounding rectangles are the four quadrants of  $B$  (all of equal size).
- A completely empty quadtree can have an arbitrary bounding rectangle, or you can wait for the first point to be inserted.

# Example of PR Quadtree



( $\leq 2$  points per leaf)



# Navigating PR Quadrees

- To find an item at  $(x, y)$  in quadtree  $T$ ,
  1. If  $(x, y)$  is outside the bounding rectangle of  $T$ , or  $T$  is empty, then  $(x, y)$  is not in  $T$ .
  2. Otherwise, if  $T$  contains a small set of items, then  $(x, y)$  is in  $T$  iff it is among these items.
  3. Otherwise,  $T$  consists of four quadtrees. Recursively look for  $(x, y)$  in each (however, step #1 above will cause all but one of these bounding boxes to reject the point immediately).
- Similar procedure works when looking for all items within some rectangle,  $R$ :
  1. If  $R$  does not intersect the bounding rectangle of  $T$ , or  $T$  is empty, then there are no items in  $R$ .
  2. Otherwise, if  $T$  contains a set of items, return those that are in  $R$ , if any.
  3. Otherwise,  $T$  consists of four quadtrees. Recursively look for points in  $R$  in each one of them.

# Insertion into PR Quadrees

Various cases for inserting a new point  $N$ , assuming maximum occupancy of a region is 2, showing initial state  $\implies$  final state.

