

Flipped Lecture Concepts

Week 5

Ananda Guna

10.09.14

Plan today

- ST's
- K-D trees

Basic ST API

```
public class ST<Key, Value>
```

```
    ST()
```

create an empty symbol table

```
    void put(Key key, Value val)
```

put key-value pair into the table

```
    Value get(Key key)
```

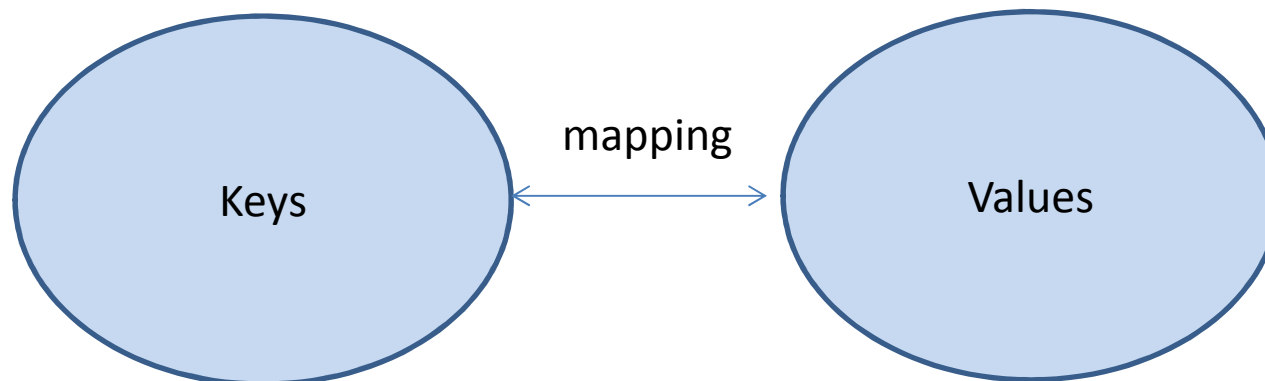
value paired with key

```
    boolean contains(Key key)
```

is there a value paired with key?

```
    void delete(Key key)
```

remove key (and its value) from table



Elementary Symbol Table (ST)

- Implementations
 - Ordered array
 - put, get, contains, delete
 - linked list
 - put, get, contains, delete
- Properties of Keys
 - Keys are comparable
 - API provides an equals method
 - why cant we use just : `key1 == key2` ?

Ordered ST

```
public class ST<Key extends Comparable<Key>, Value>
```

```
...
```

```
Key min() smallest key
```

```
Key max() largest key
```

```
Key floor(Key key) largest key less than or equal to key
```

```
Key ceiling(Key key) smallest key greater than or equal to key
```

```
int rank(Key key) number of keys less than key
```

```
Key select(int k) key of rank k
```

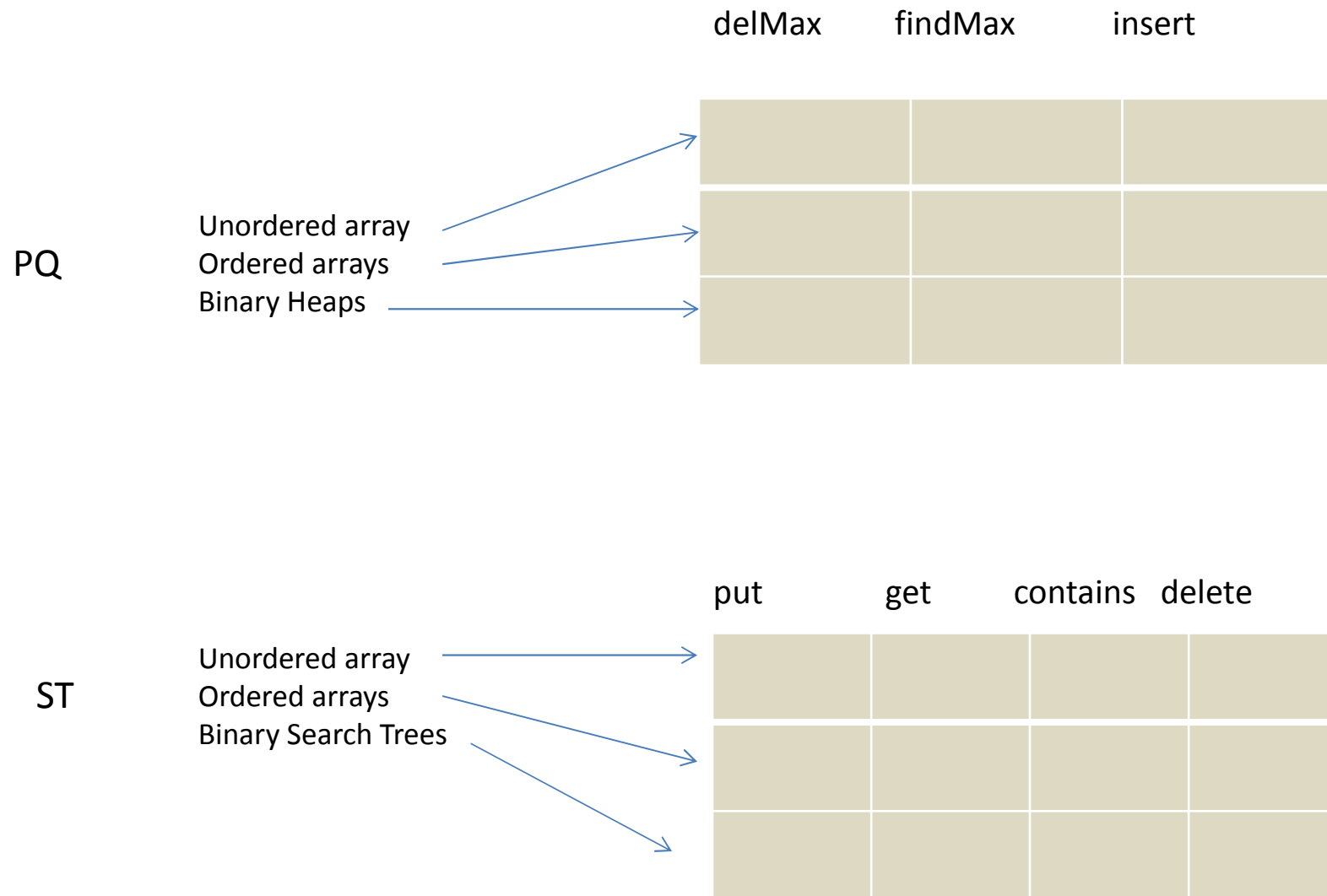
```
void deleteMin() delete smallest key
```

```
void deleteMax() delete largest key
```

```
int size(Key lo, Key hi) number of keys between lo and hi
```

```
Iterable<Key> keys() all keys, in sorted order
```

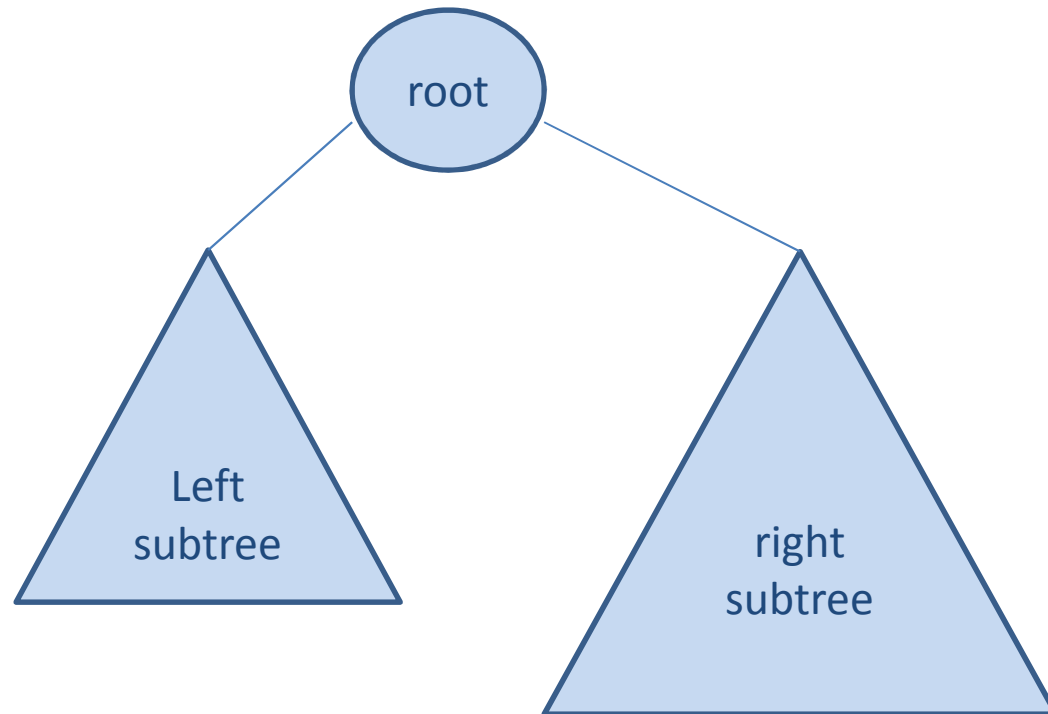
PQ versus ST



ST Performance

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	N	N	<code>equals()</code>
binary search (ordered array)	$\log N$	N	$\log N$	N	<code>compareTo()</code>

BST's



Two invariants

- Shape – It is a binary tree and recursively defined
- Order - $\text{Left subtree} < \text{Root} < \text{Right subtree}$

BST API

```
public class BST<Key extends Comparable<Key>, Value>
{
    private Node root;

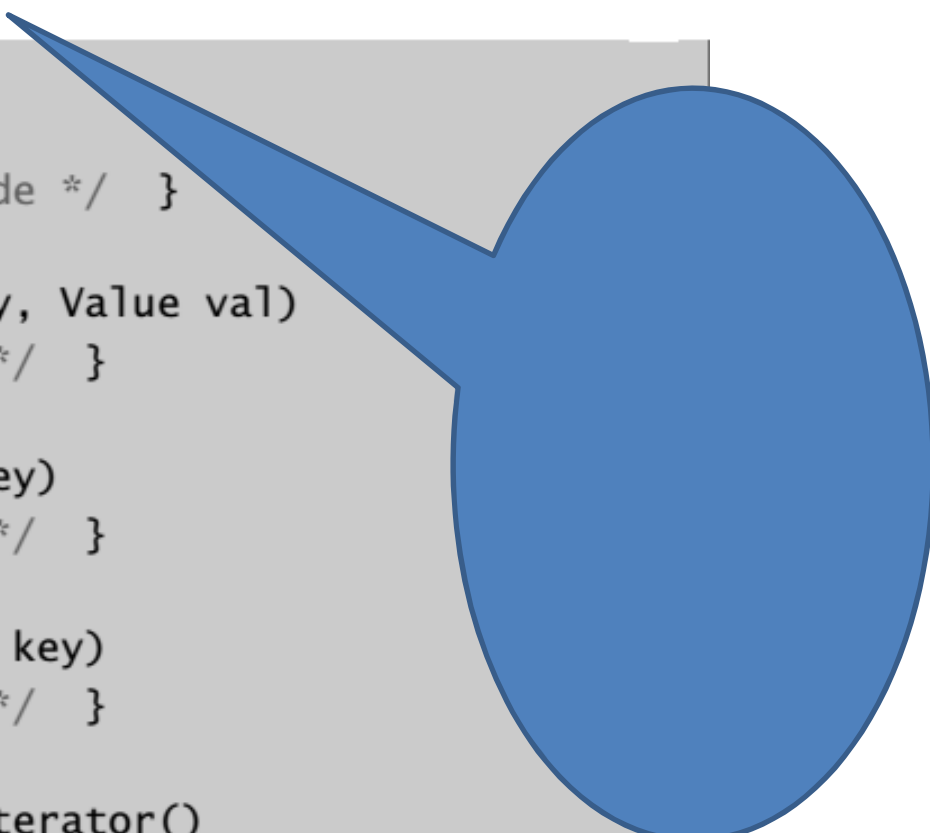
    private class Node
    { /* see previous slide */ }

    public void put(Key key, Value val)
    { /* see next slides */ }

    public Value get(Key key)
    { /* see next slides */ }

    public void delete(Key key)
    { /* see next slides */ }

    public Iterable<Key> iterator()
    { /* see next slides */ }
}
```



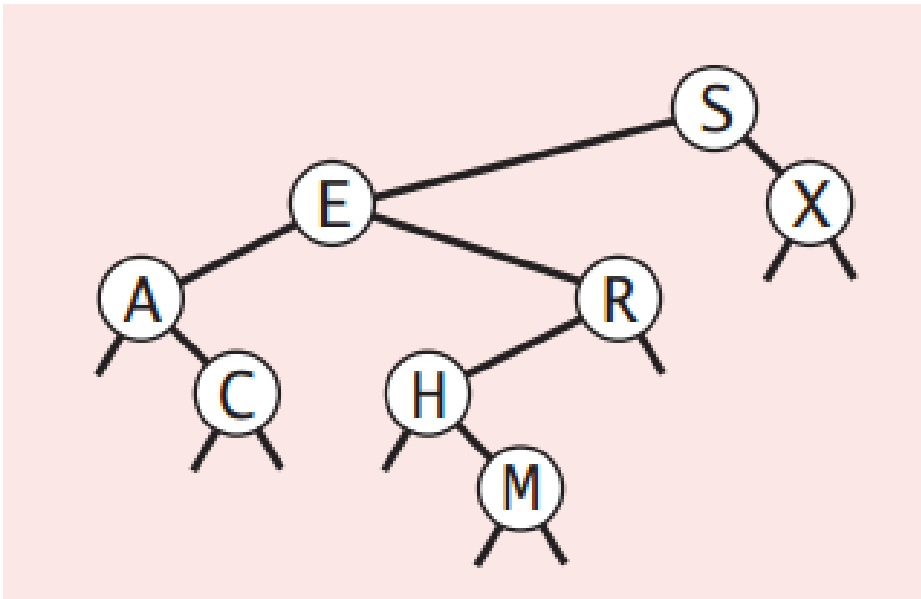
Trace the Code

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

Tree Traversals

- Inorder (L, Root, R)
 - Produces an ordered sequence when traversing a BST
 - A binary tree B is a BST iff inorder traversal of B produces an ordered sequence
- Preorder (Left, Root, Right)
- Post order (Left, Right, Root)
- Level order
 - Produces nodes by level from left to right
 - **Exercise:** Implement a level order traversal of a BST using a queue

Deleting keys



Hibbard Deletion (3 cases)

- Delete a leaf node
- Delete a node with one child
- Delete a node with two children

Delete the following keys (in that order)

- X
- H
- E

Question

- Is it possible to delete random keys using Hibbard deletion and maintain the $\lg N$ performance?
 - Answer:
- Is there a delete algorithm that can always maintain a logarithmic height
 - Answer:

Ordered BST API

```
public class ST<Key extends Comparable<Key>, Value>
```

```
...
```

Key min()	<i>smallest key</i>
-----------	---------------------

Key max()	<i>largest key</i>
-----------	--------------------

Key floor(Key key)	<i>largest key less than or equal to key</i>
--------------------	--

Key ceiling(Key key)	<i>smallest key greater than or equal to key</i>
----------------------	--

int rank(Key key)	<i>number of keys less than key</i>
-------------------	-------------------------------------

Key select(int k)	<i>key of rank k</i>
-------------------	----------------------

void deleteMin()	<i>delete smallest key</i>
------------------	----------------------------

void deleteMax()	<i>delete largest key</i>
------------------	---------------------------

int size(Key lo, Key hi)	<i>number of keys between lo and hi</i>
--------------------------	---

Iterable<Key> keys()	<i>all keys, in sorted order</i>
----------------------	----------------------------------

Order of growth of ST operations

implementation	guarantee		average case		operations on keys
	search	insert	search hit	insert	
sequential search (unordered list)	N	N	N	N	<code>equals()</code>
binary search (ordered array)	$\log N$	N	$\log N$	N	<code>compareTo()</code>
BST	N	N	$\log N$	$\log N$	<code>compareTo()</code>

Challenge

- Argue that it is not possible to build a BST in linear time.
- How many BST's of size n can be created from n distinct keys?

1D Range Search

- Points on a line



- Applications
 - Insert key-value pairs (database tables)
 - Find all values between two given keys k_1 and k_2
 - Find the count of keys between two keys

Expected Performance

order of growth of running time for 1d range search

data structure	insert	range count	range search
unordered list	1	N	N
ordered array	N	$\log N$	$R + \log N$
goal	$\log N$	$\log N$	$R + \log N$

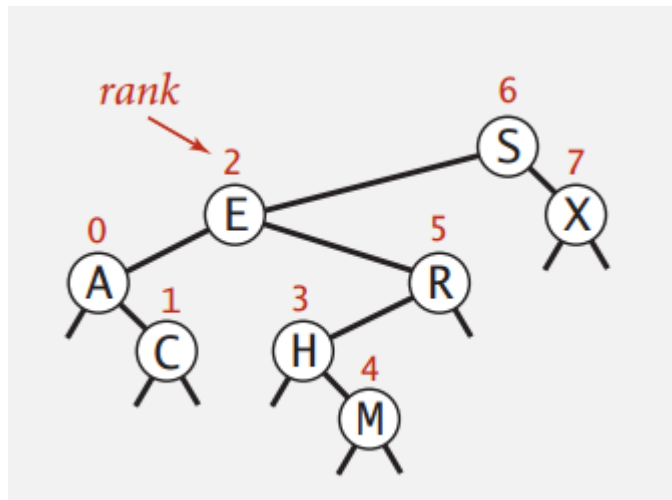
N = number of keys

R = number of keys that match

Range count

```
public int size(Key lo, Key hi)
{
    if (contains(hi)) return rank(hi) - rank(lo) + 1;
    else               return rank(hi) - rank(lo);
}
```

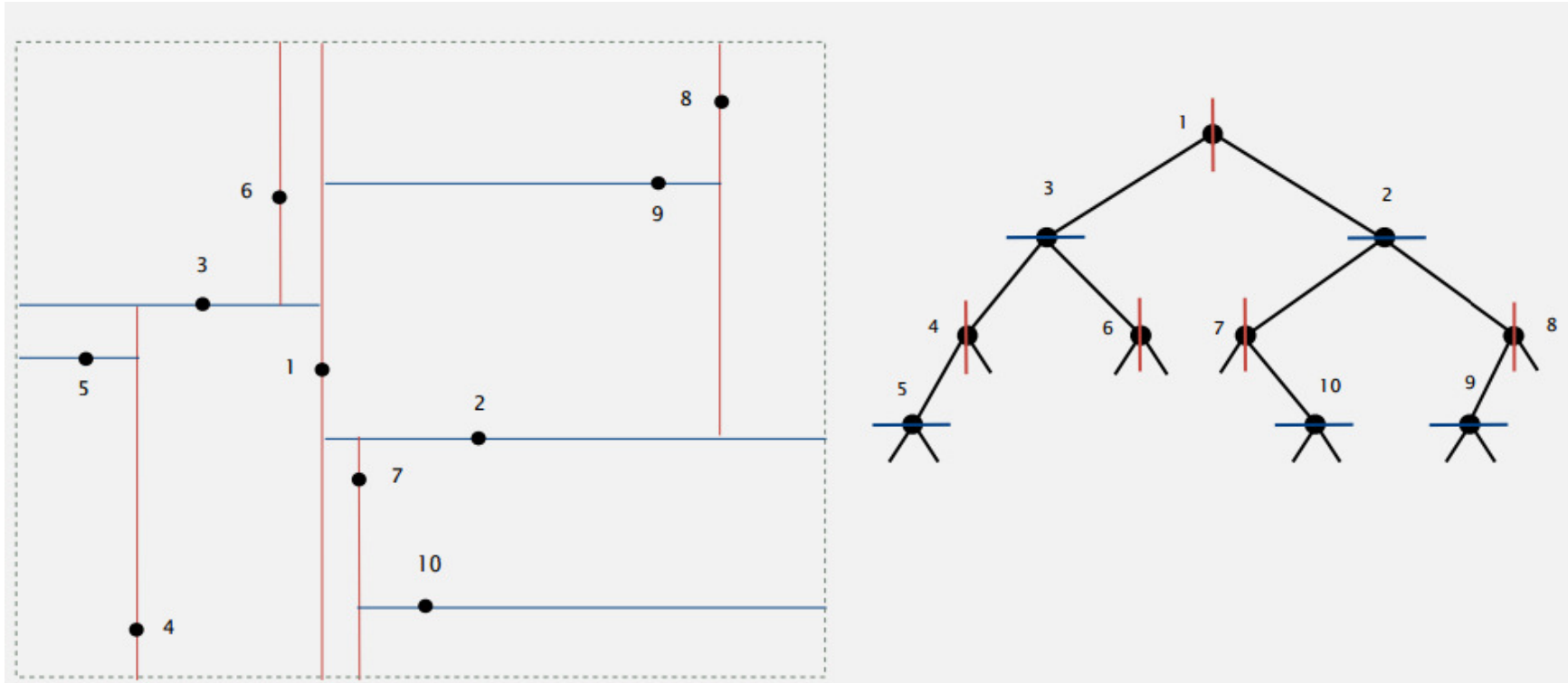
← number of keys < hi



KdTrees

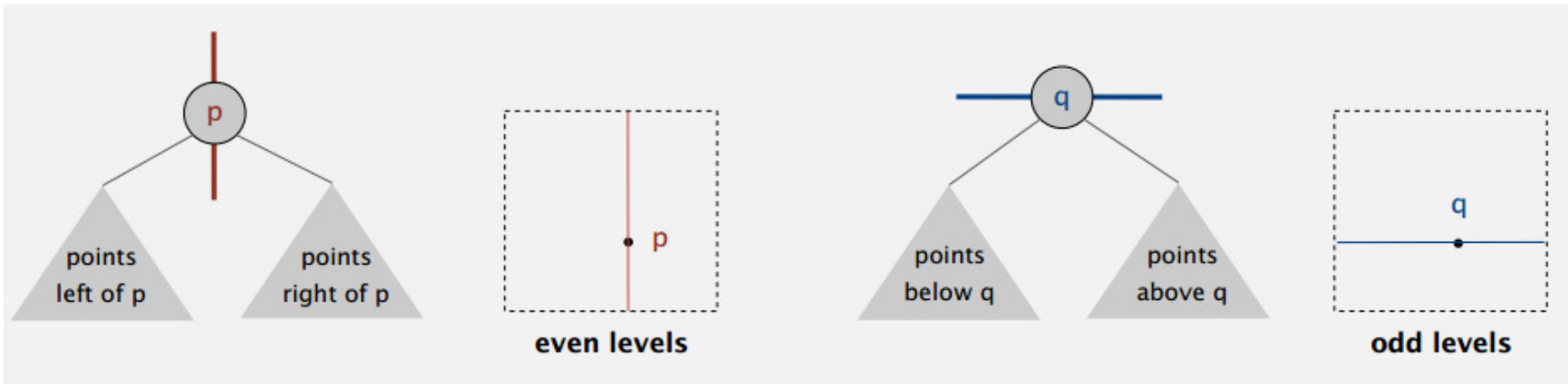
- A way to order k-dimensional trees in space to make search more efficient
- We will consider 2dTrees as an example
 - Given a set of points in the form (x,y) , how do we partition the 2d space so that a search can be performed more efficiently?

2dTree idea

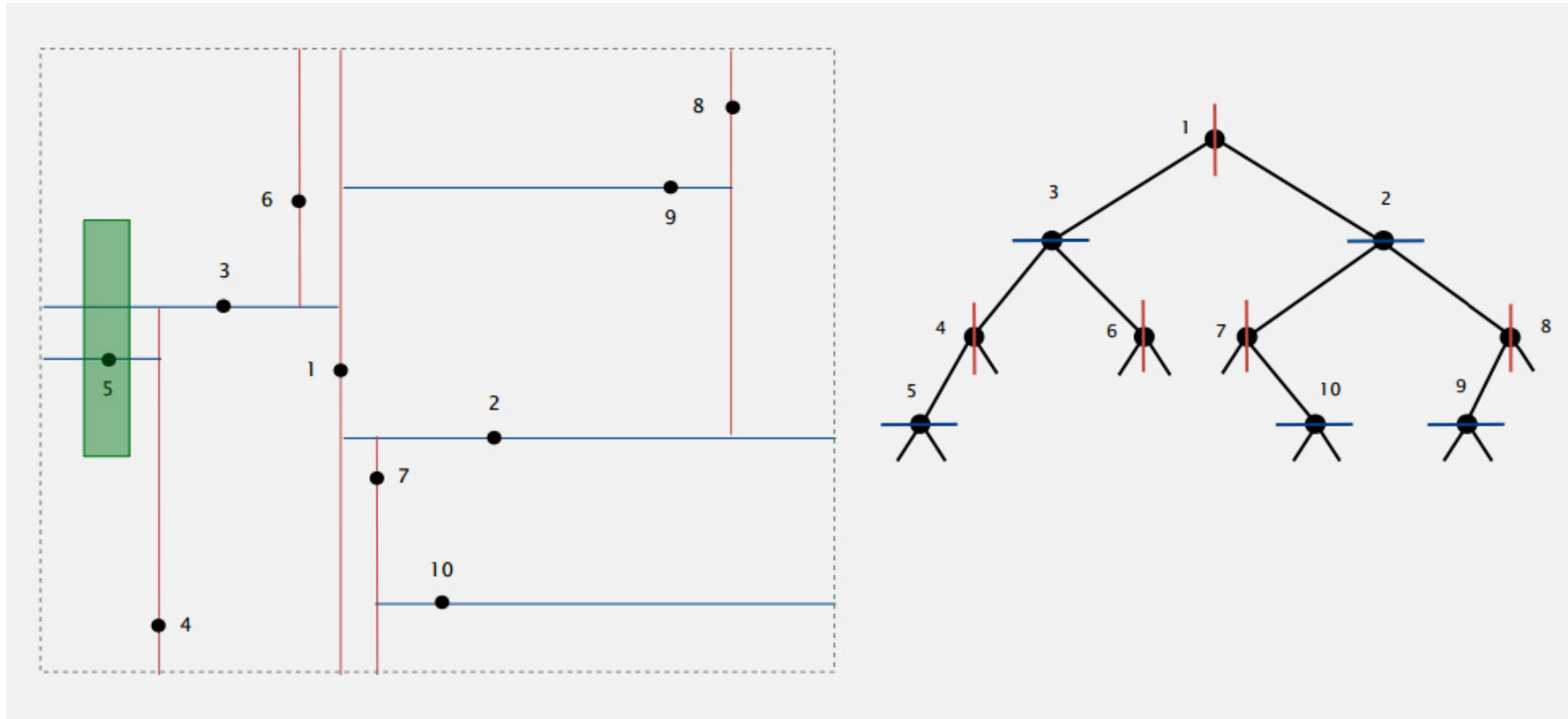


- Recursively partition space into rectangles.
- Each rectangle is a bounding box for some point
- The rectangles are determined by a BST (shown on right) called a KdTree
- See a demo of building a KdTree

2dTree insert

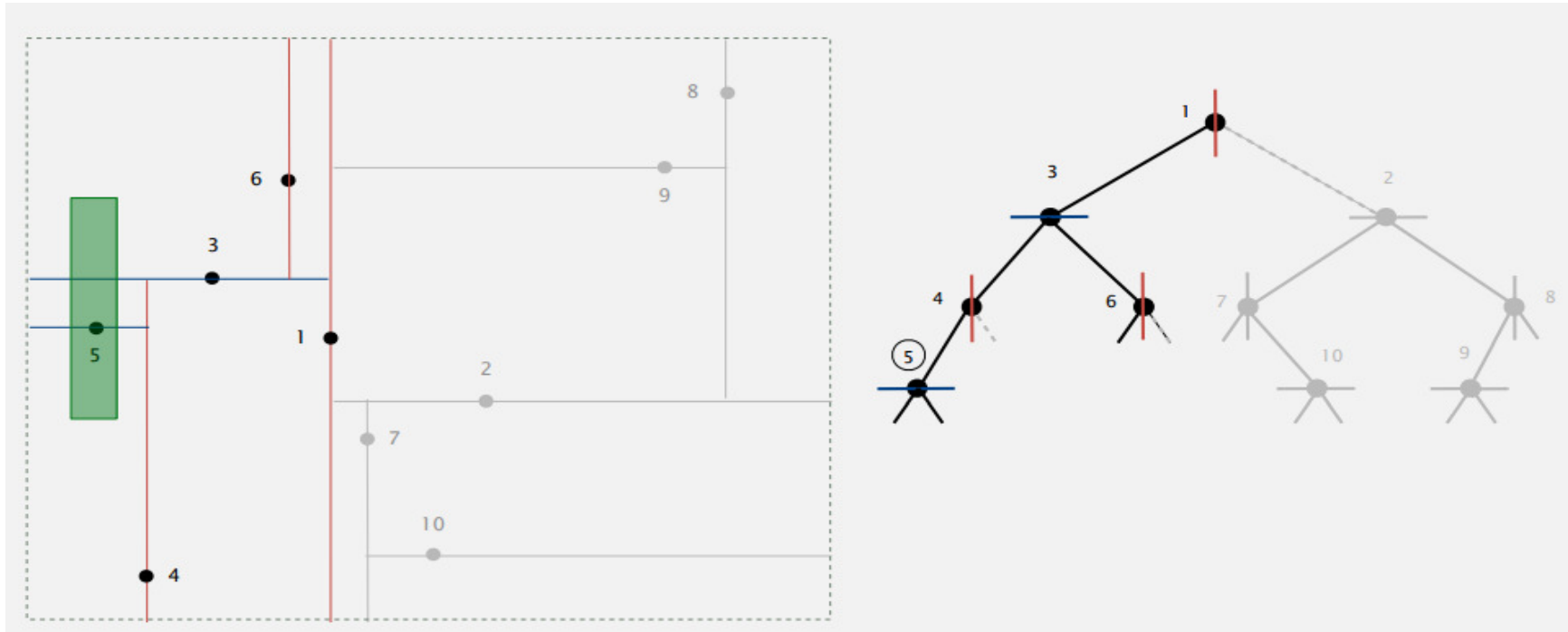


2D range search



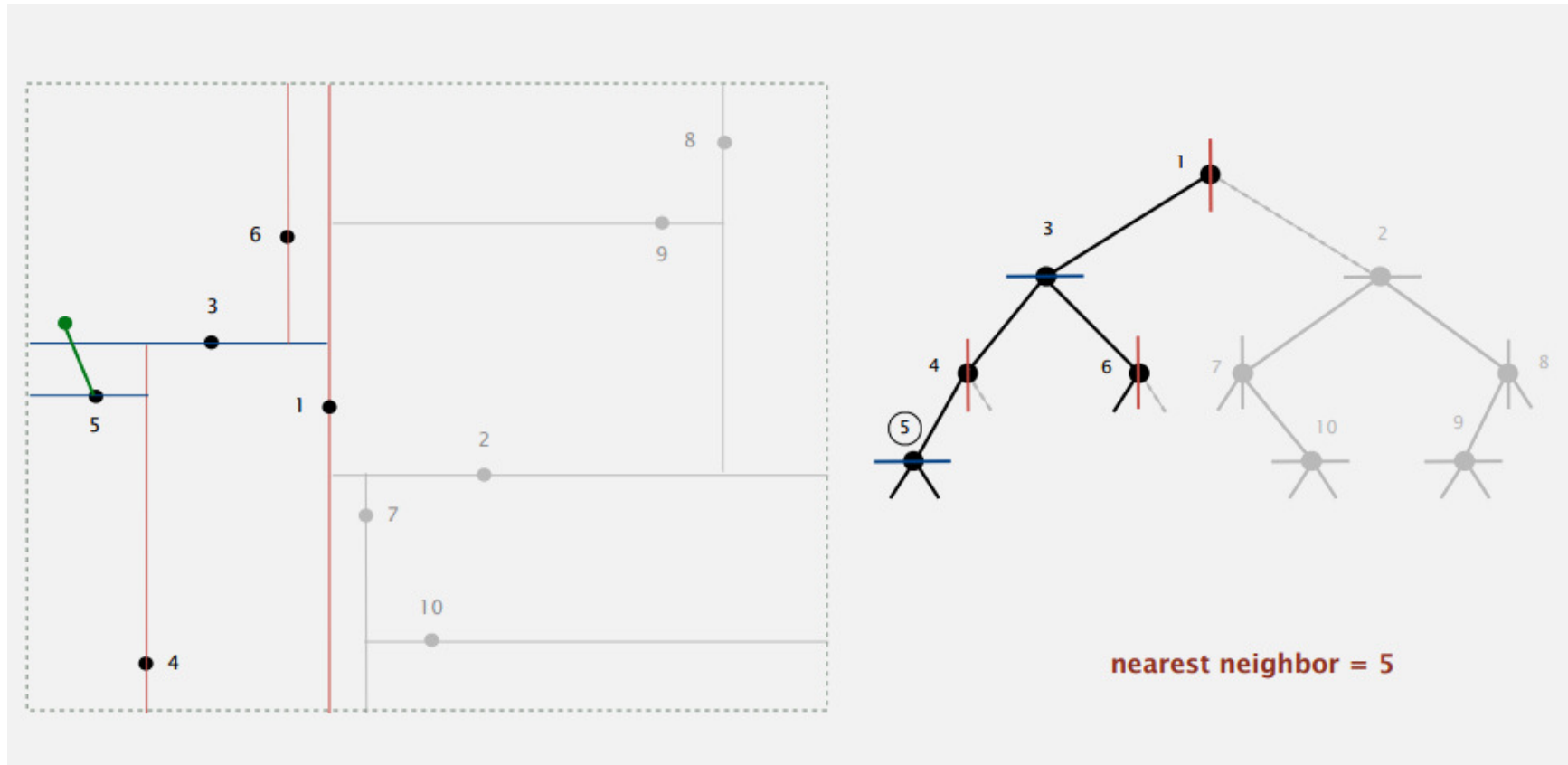
- Recursively search Left and Right subtrees to find the points that fall within the green search rectangle
- Since the bounding rectangle for point 2 or $[(2, -\infty), (\infty, \infty)]$ do not intersect the green search box, we can “prune” the entire right half of the tree.
 - That is, there is no way the points $\{2, 7, 10, 8, 9\}$ can be inside the search rectangle. So we can focus on the left subtree

An observation



1. Note that entire right subtree is pruned.
2. The bounding rectangles of points, $\{1, 3, 6, 4, 5\}$ intersects with search area
Upon testing.
3. We find that only the point 5 is within the box

2D Nearest Neighbor

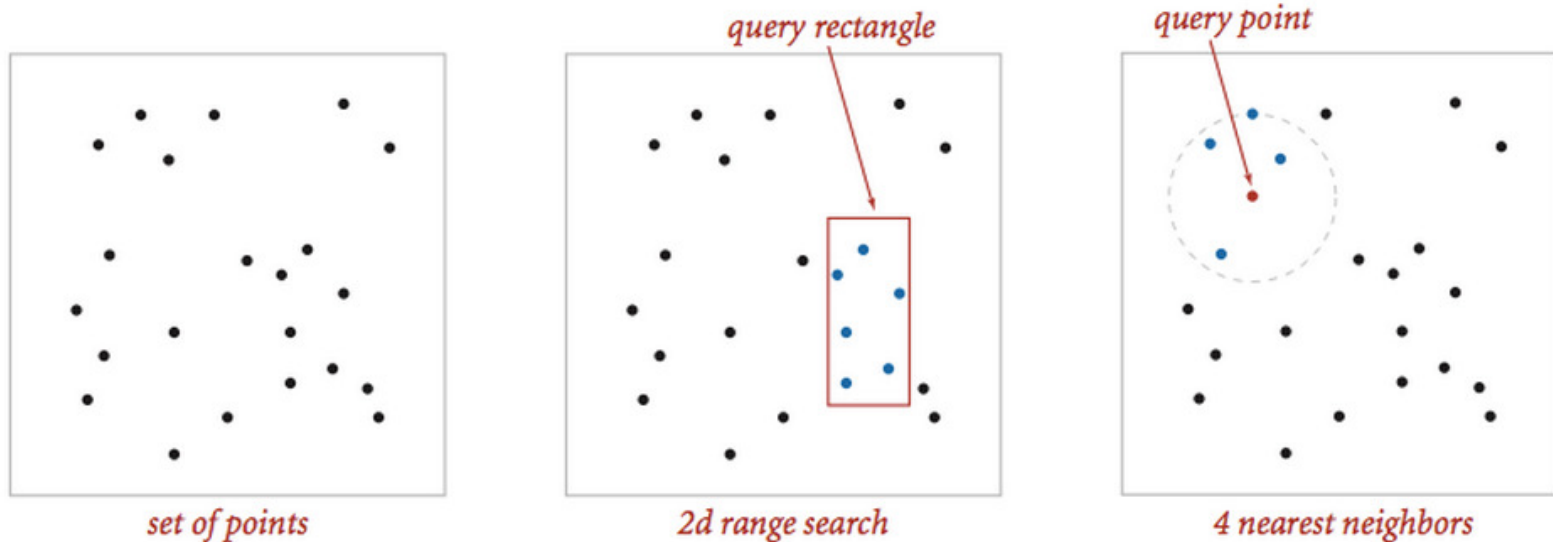


Algorithm works by pruning subtrees that may be away from the target point

Assignment 5

bonus slides

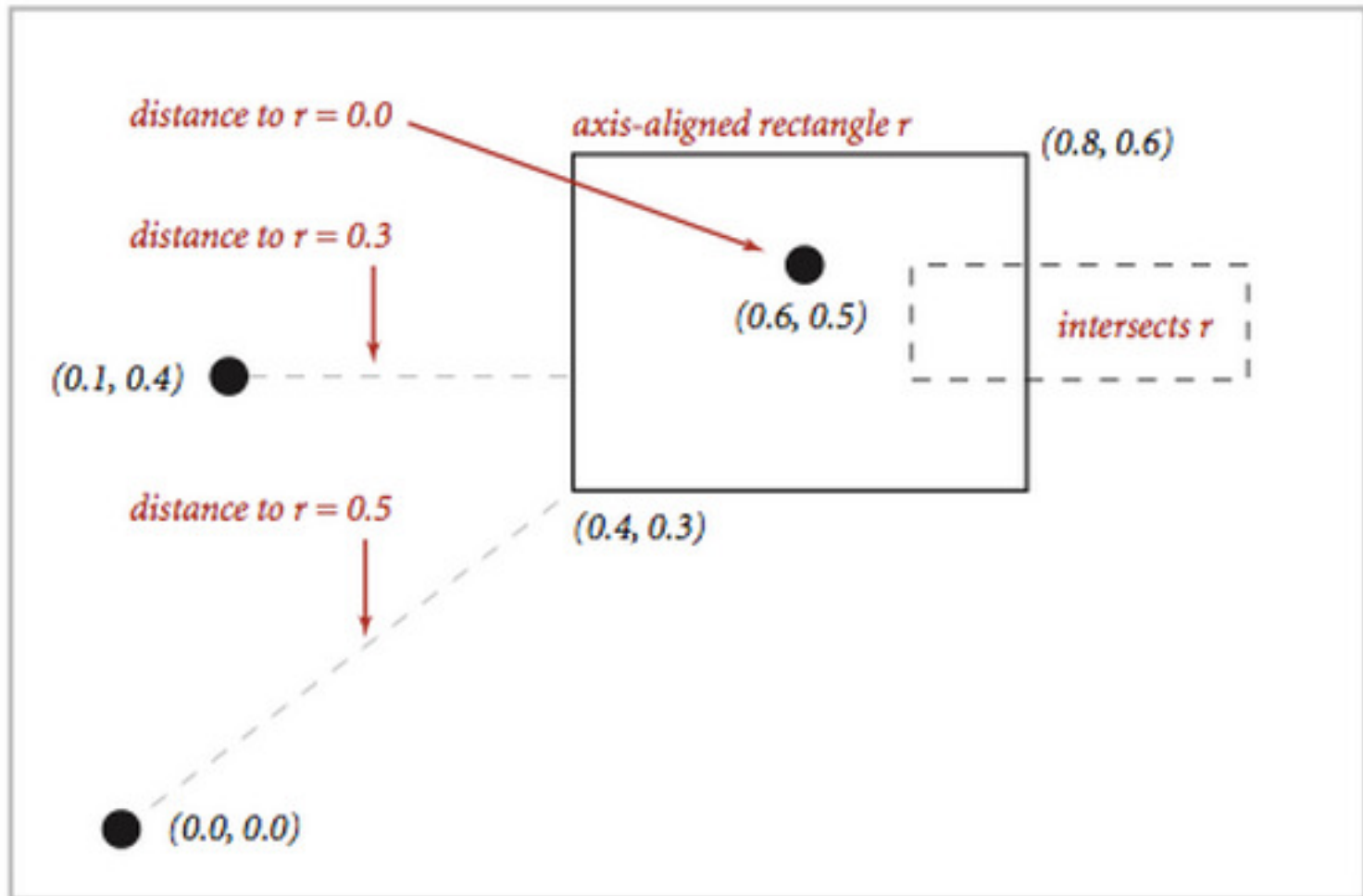
Assignment 5 – Kd trees



Support two searches

- *range search* (find all of the points contained in a query rectangle)
- *nearest neighbor search* (find a closest point to a query point).

Geometric Primitives



Implementation

2D Point type

```
public class Point2D {  
    public Point2D(double x, double y)           // construct the point (x, y)  
    public double x()                           // x-coordinate  
    public double y()                           // y-coordinate  
    public double distanceSquaredTo(Point2D that) // square of Euclidean distance between  
    public int compareTo(Point2D that)           // for use in an ordered symbol table  
    public boolean equals(Object that)           // does this point equal that?  
    public void draw()                          // draw to standard draw  
    public String toString()                    // string representation  
}
```

Use [Point2D.java](#) (part of algs4.jar)

Rectangle Class

```
public class RectHV {  
    public      RectHV(double xmin, double ymin,  
                        double xmax, double ymax)  
  
    public double xmin()  
    public double ymin()  
    public double xmax()  
    public double ymax()  
    public boolean contains(Point2D p)  
    public boolean intersects(RectHV that)  
    public double distanceSquaredTo(Point2D p)  
    public boolean equals(Object that)  
    public void draw()  
    public String toString()  
}
```

Use [RectHV.java](#) (given)

Brute Force Implementation

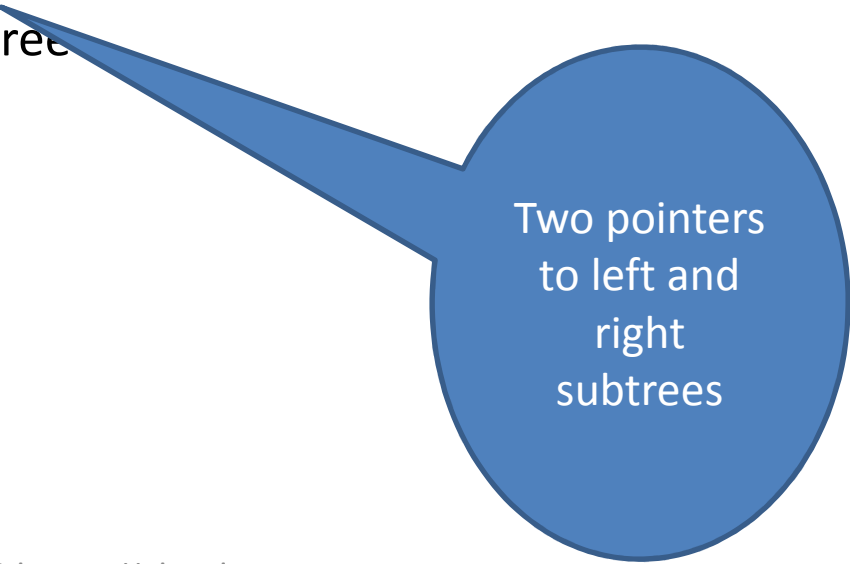
- Implement pointST.java using a RB BST

```
public class PointST<Value> {  
    public          PointST()  
    public          boolean isEmpty()  
    public          int size()  
    public          void insert(Point2D p, Value v)  
    public          Value get(Point2D p)  
    public          boolean contains(Point2D p)  
    public          void draw()  
    public Iterable<Point2D> range(RectHV rect)  
    public          Point2D nearest(Point2D p)  
    public static void main(String[] args)  
}
```

- PointST is a map: Point → arbitrary value
- using RedBlackBST from algs4.jar or java.util.TreeMap
- insert(), get() and contains() in $O(\log n)$ time
- nearest()
 - Scan all points and compute distance squared and find the ones with the minimum
- range()
 - Scan all points and find the ones within the given rectangle
- Both operations are $O(n)$

Node class design for k-d trees

```
private static class Node {  
    private Point2D p; // the point private  
    Value value; // the symbol table maps the point to this value private  
    RectHV rect; // the axis-aligned rectangle corresponding to this  
        // node  
    private Node lb; // the left/bottom subtree  
    private Node rt; // the right/top subtree  
}
```



Two pointers
to left and
right
subtrees

Assignment help session

- Friend 108
- Friday October 10, from 4;30-6:00
- Optional
 - Watch the videos and/or attend the help session
- Read the assignment before you go to help session