CSE 100: ITERATORS AND RUNNING TIME ANALYSIS

Announcements

- Ungraded reading is not optional reading. In particular, 3.3 (and other parts that are assigned) can REALLY help with the PA.
- Don't forget about the integrity agreement

- iClicker registration form
- Stepik reading assignment graded for correctness

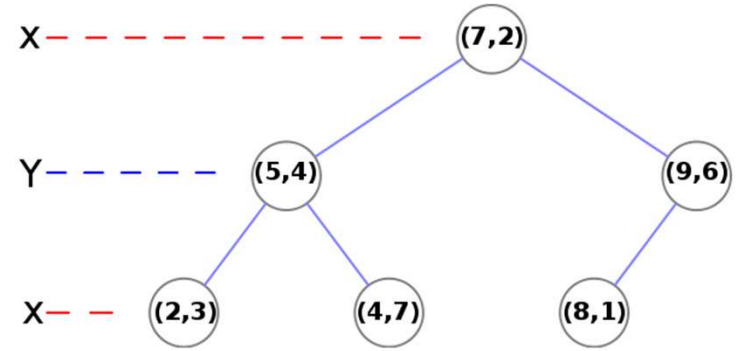
- HW1 is out.
  - Homework 1 covers Binary Search Trees and K-d Trees.
  - Deadline: 10/16 (Tuesday) @ 11:59PM
  - Submissions must be made on Gradescope!

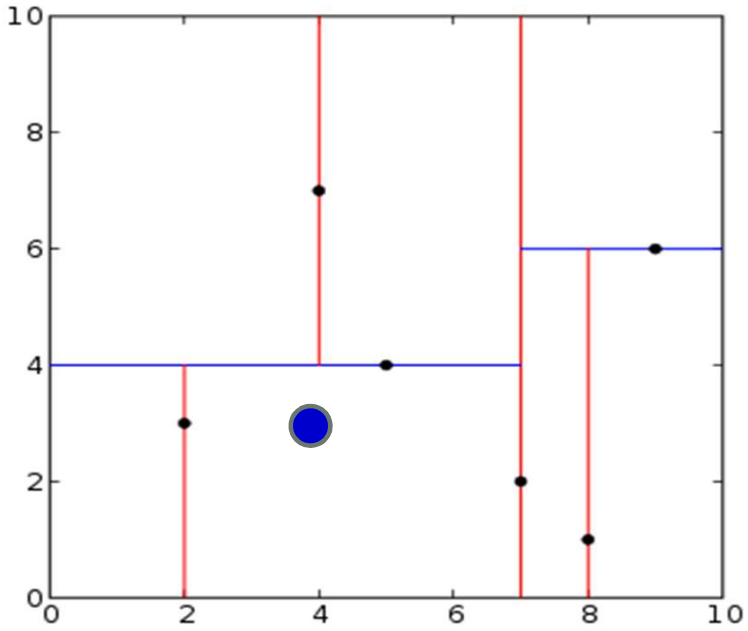
Goals for today

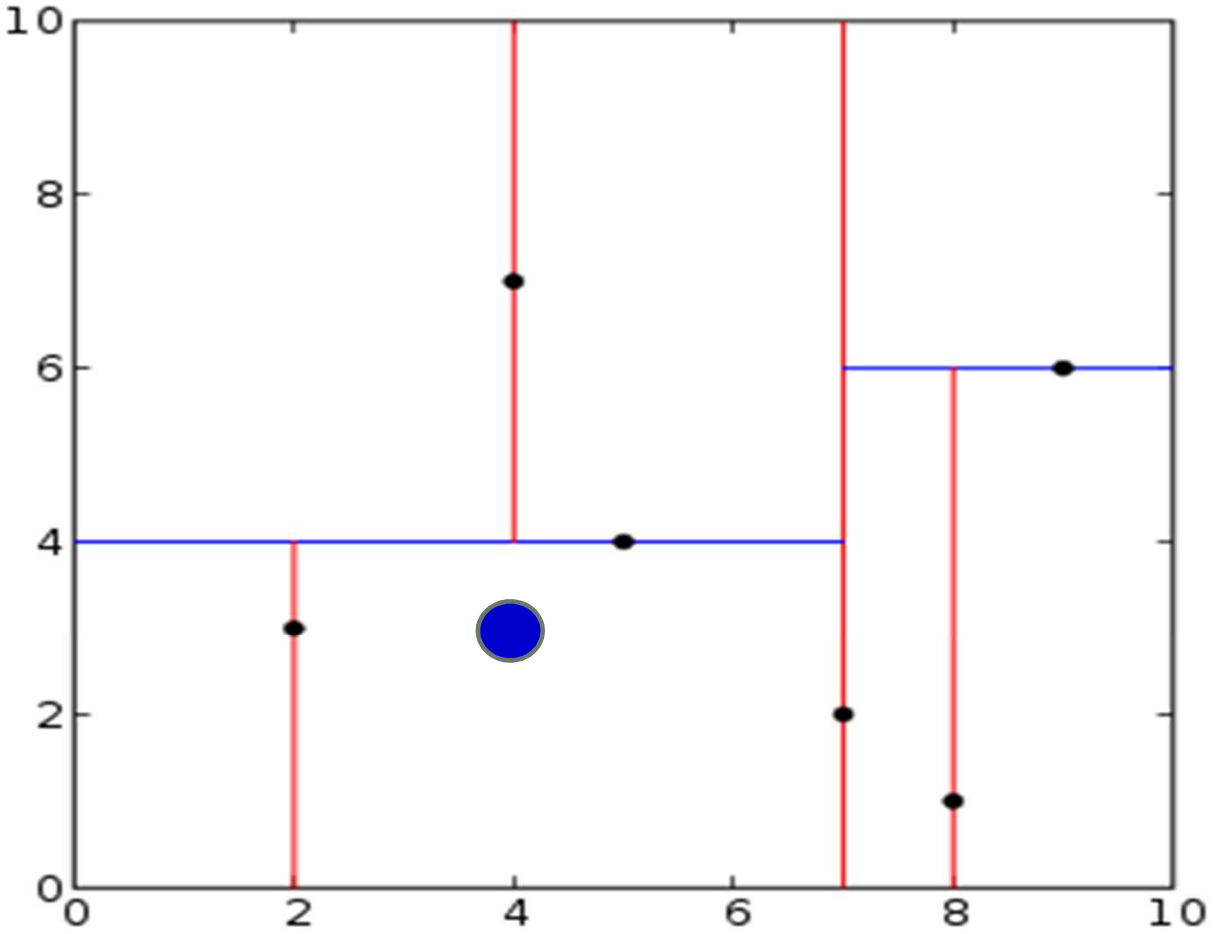
- findNN on KD-tree
- Explain the Iterator pattern
- Practice with Big-O

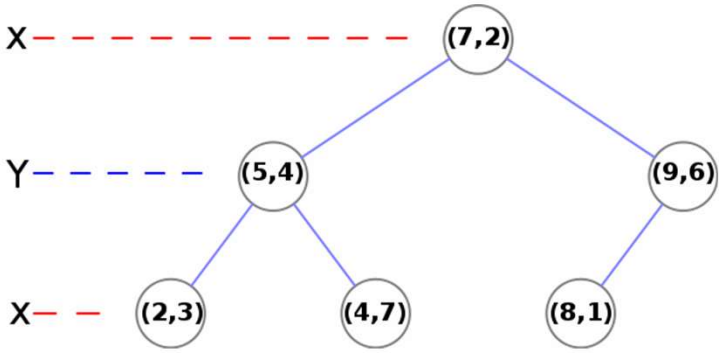
KD Tree Nearest Neighbor

- Intuition:
- Make your best guess similar to find.
- Only explore alternative paths if they might produce a better result



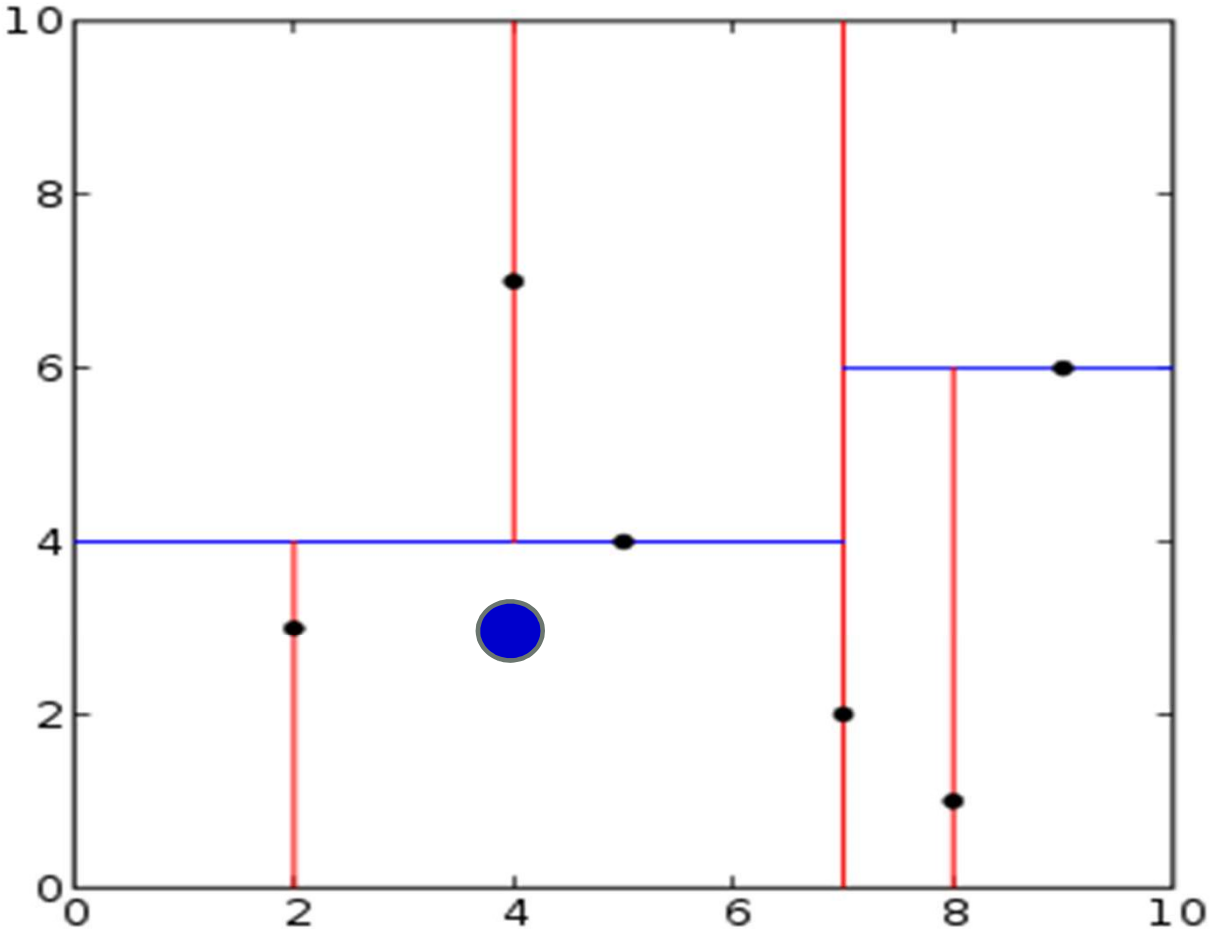


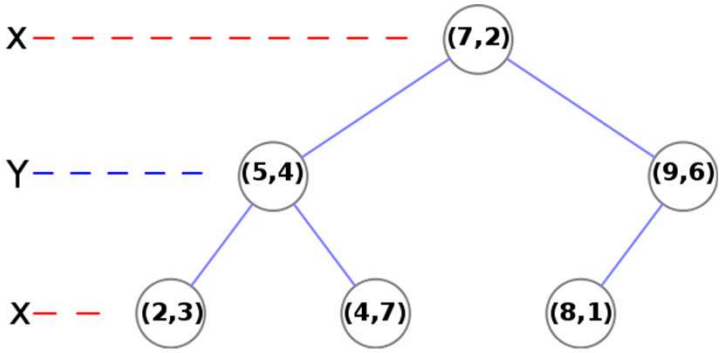


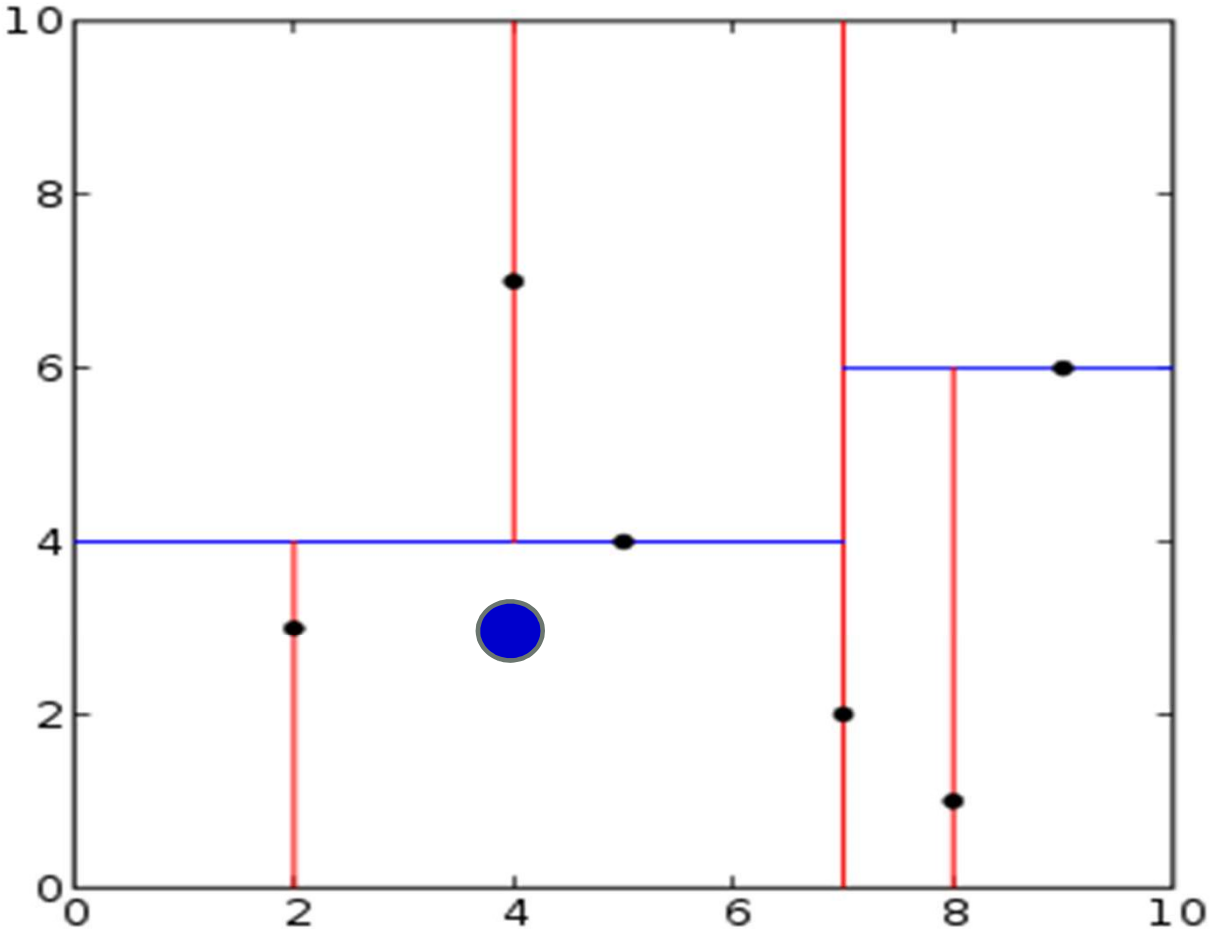


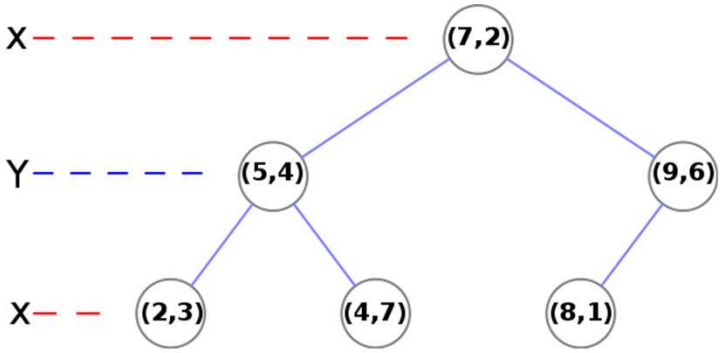
- Where would the algorithm stop if we run **find** on the point (4,3)?

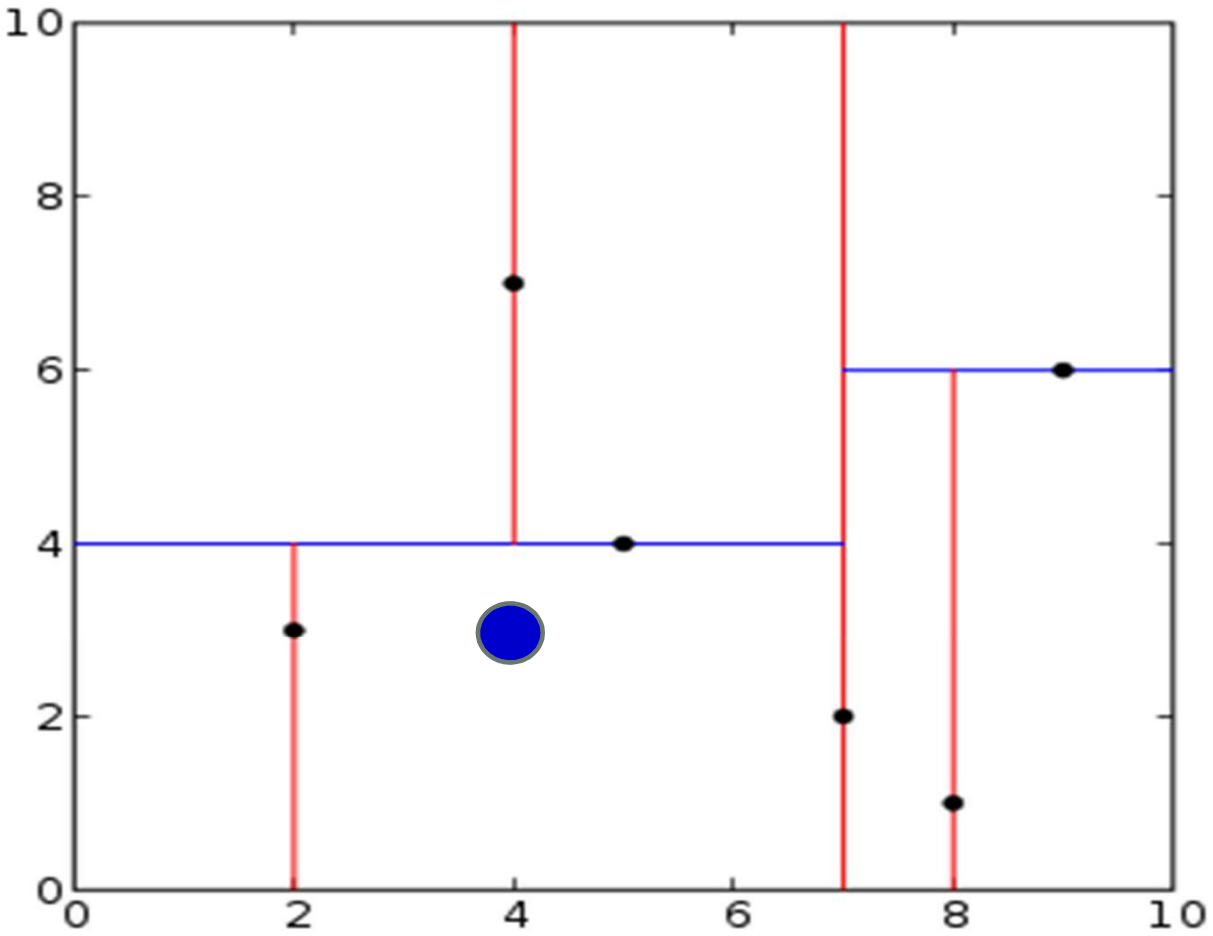
A. (2,3)
B. (4,7)
C. (5,4)
D. (7,2)
E. None of the above

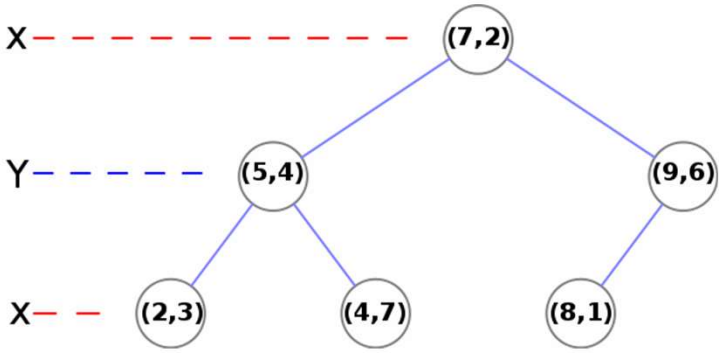

10
8
6
4
2
0
0
2
4
6
8
10


X
Y
X
(7,2)
(5,4)
(9,6)
(2,3)
(4,7)
(8,1)


10
8
6
4
2
0
0
2
4
6
8
10


X
Y
X
(7,2)
(5,4)
(9,6)
(2,3)
(4,7)
(8,1)

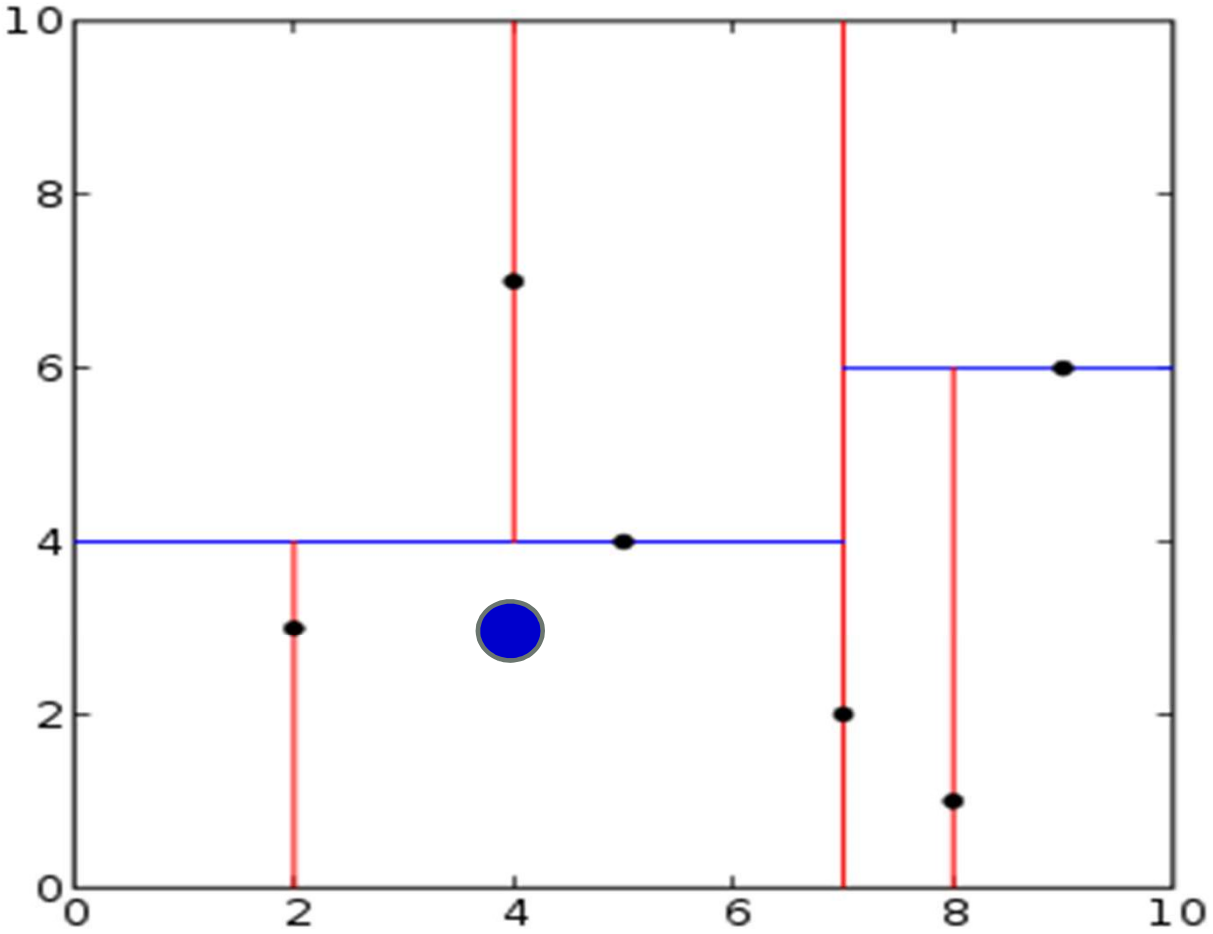


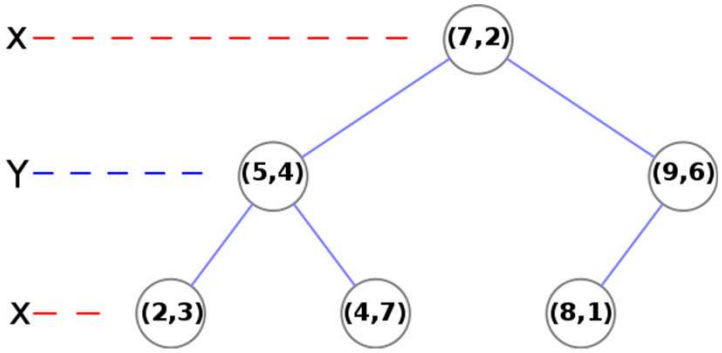


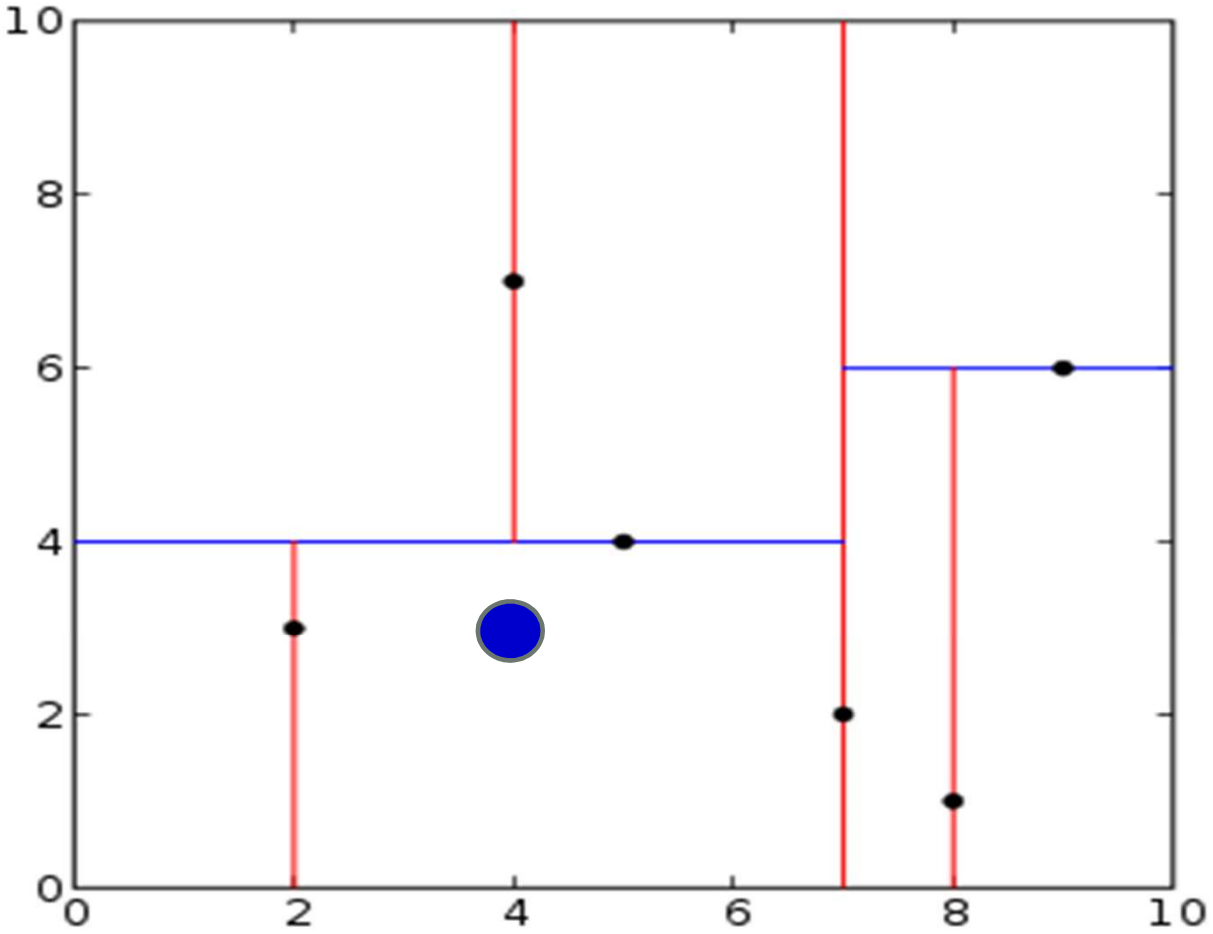
Do we need to check the right subtree of (5,4)?

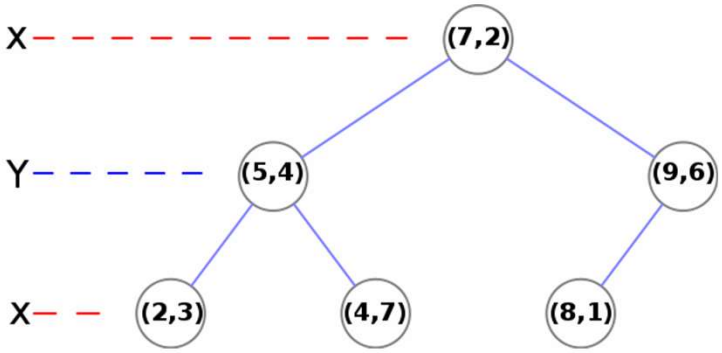
A. Yes

B. No


10
8
6
4
2
0
0
2
4
6
8
10


X
Y
X
(7,2)
(5,4)
(9,6)
(2,3)
(4,7)
(8,1)





Do we need to check the right subtree of (7,2)?

A. Yes

B. No

PA1 Part 2

- Implement:

Build

Find nearest neighbor

Main2 (mainly dealing with i/o)

C++STL and BSTs

- The C++ Standard Template Library is a very handy set of three built-in components:
  - Containers: Data structures
  - Iterators: Standard way to search containers
  - Algorithms: These are what we ultimately use to solve problems

Motivation for iterators

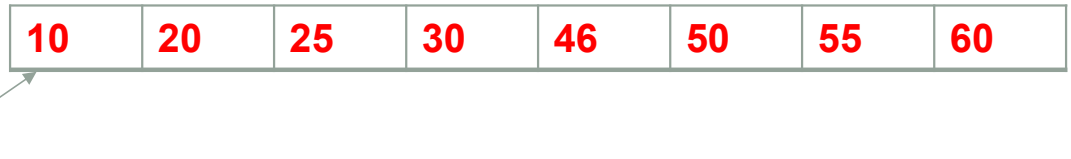
- The same algorithms can be applied to multiple container classes
- C++ STL avoids rewriting the same algorithm for different container classes by using ITERATORS
- Algorithms interface with containers via iterators

STL container classes

```
array
vector
deque
forward_list
list
stack
queue
priority_queue
set
multiset (non unique keys)
unordered_set
map
unordered_map
multimap
bitset
```

C++ STL Iterators

- Iterators are generalized pointers.
- Let's consider a very simple algorithm (printing in order) applied to a very simple data structure (sorted array)



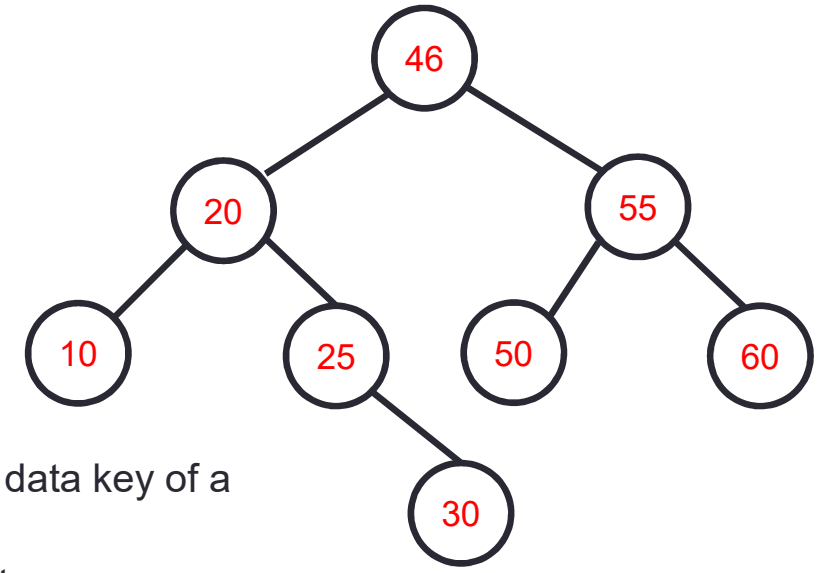

```
void print_inorder(int * p, int size) {
    for(int i=0; i<size; i++) {
        std::cout << *p << endl;
        ++p;
    }
}
```

- We would like our print "algorithm" to also work with other data structures
- How do we need to modify it to print the elements of a BST?

C++ STL Iterators

- Consider your BST implementation from PA1

```
void print_inorder(BSTNode<int> *p, int size)
{
    for(int i=0; i<size; i++)
    {
        std::cout << *p << endl;
        ++p;
    }
}
```



When will the above code work?

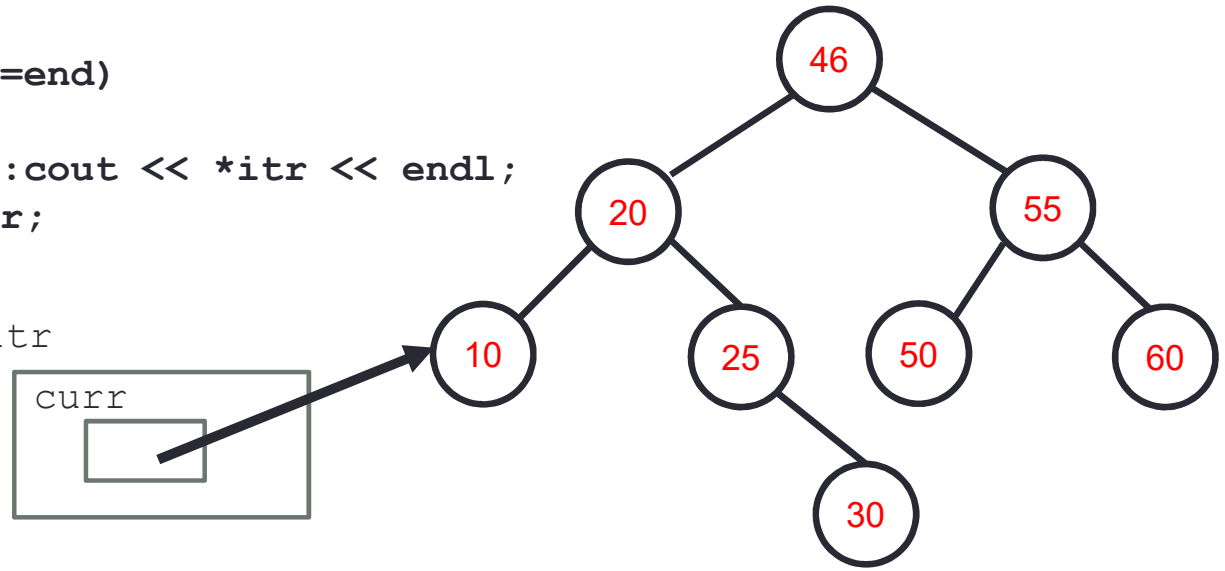
A. The operator "<<" is overloaded to print the data key of a BSTNode
B. The BSTNode class overloads the ++ operator
C. Both A and B
D. None of the above

C++ STL Iterators

- To solve this problem the BST (not BSTNode) class has to supply to the client (print) with a generic pointer (an iterator object) which can be used by the client to access data in the container sequentially, without exposing the underlying details of the class

```
void print_inorder(BST<int> & bst)  {
  BST<int>::iterator itr = bst.begin();
  BST<int>::iterator end = bst.end();

  while(itr!=end)
    {
        std::cout << *itr << endl;
        ++itr;
    }
}
```



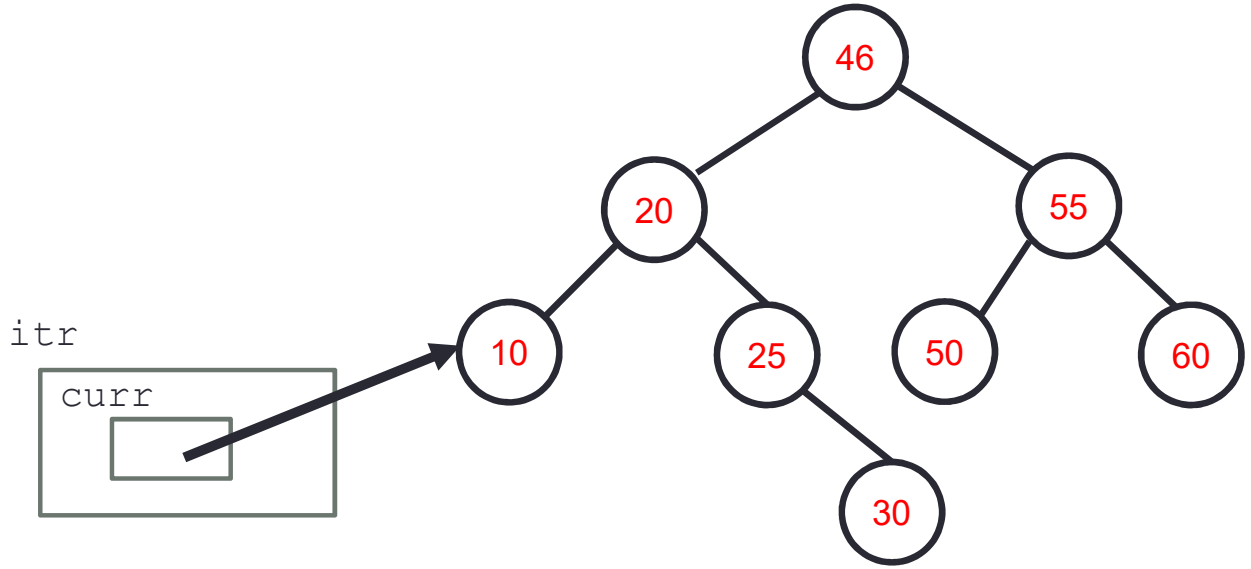
C++ STL Iterators

```
void print_inorder(BST<int> & bst)  {
  BST<int>::iterator itr = bst.begin();
  BST<int>::iterator end = bst.end();

  while(itr!=end)
    {
        std::cout << *itr << endl;
        ++itr;
    }
}
```

What should **begin()** return?

A. The address of the root in the set container class
B. The address of the node with the smallest data key
C. The address of the smallest data key
D. The address of an iterator object
E. None of the above

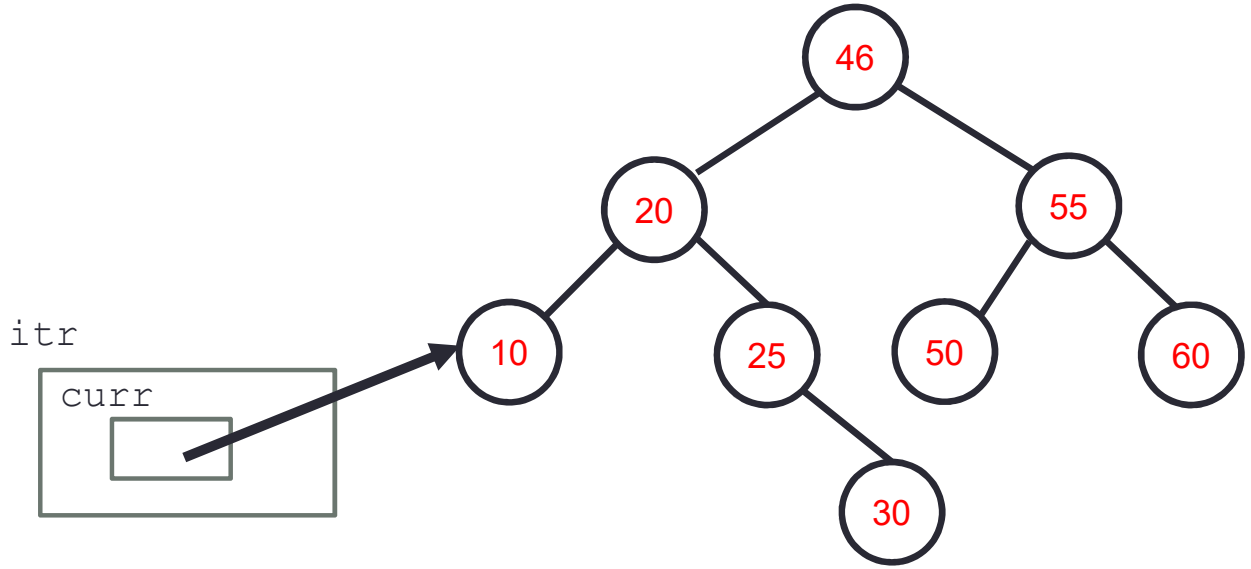


C++ STL Iterators

```
void print_inorder(BST<int> & bst)   {
  auto itr = bst.begin();
  auto end = bst.end();

  while(itr!=end)
    {
        std::cout << *itr << endl;
        ++itr;
    }
}
```

Make the code more compact using the keyword “auto”



Warm-up with Big-O

```
void dist(vector<int> a) {
    int n = a.size();
    for(int i = 0; i < n-1; i++) {
        for(int j = i+1; j < n; j++) {
            cout << a[j] << " - " << a[i] << " = " << (a[j]-a[i]) << endl;
        }
    }
}
```

What is the *tightest Big-O bound* for the code above?

A. O(log n)
B. O(n)
C. O(n * log n)
D. $O(n^2)$

Deriving the Big-O bound

```
void dist(vector<int> a) {
    int n = a.size();
    for(int i = 0; i < n-1; i++) {
        for(int j = i+1; j < n; j++) {
            cout << a[j] << " - " << a[i] << " = " << (a[j]-a[i]) << endl;
        }
    }
}
```

Big-O vs Big-Theta vs Big-Omega

```
void dist(vector<int> a) {
    int n = a.size();
    for(int i = 0; i < n-1; i++) {
        for(int j = i+1; j < n; j++) {
            cout << a[j] << " - " << a[i] << " = " << (a[j]-a[i]) << endl;
        }
    }
}
```

$$f(n) = \frac{(n-1)n}{2}$$

A Big-O Challenge

```
void tricky(int n) {
    int operations = 0;
    while(n > 0) {
        for(int i = 0; i < n; i++) {
            cout << "Operations: " << operations++ << endl;
        }
        n /= 2;
    }
}
```

What is the *tightest Big-O bound* for the code above?

A. O(log n)
B. O(n)
C. O(n * log n)
D. $O(n^2)$

Analysis of data structures

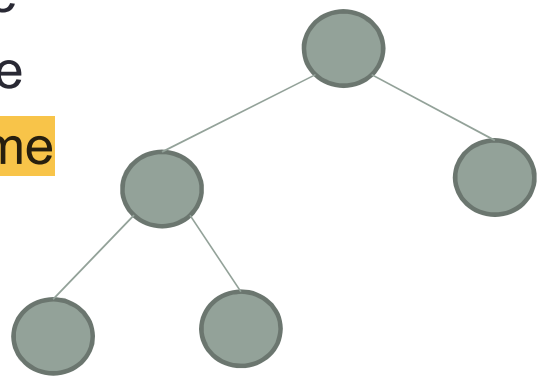
How long does it take to find whether or not an element is in a sorted singly linked list *in the best case*?

A. O(1)
B. O(logN)
C. O(N)
D. O(N*logN)
E. Not enough information

Is the Big-O worst case time to find an element in a BST better or worse than for a Linked List?

Time to find is O(n) in the worst case in a Linked List. Is the time to find better, worse or the same in the worst case for a BST with n elements?

A. A BST has a better worst-case running time
B. A BST has a worse worst-case running time
C. They have the same worst-case running time



What about the *average* case?

How long does it take to find an element that *is* in the linked list (successful find) *in the average case*?

A. O(1)
B. O(logN)
C. O(N)
D. O(N*logN)
E. Not enough information

Determining *average* case for a successful find in a Linked List with N elements?

What assumption did we make in our analysis?

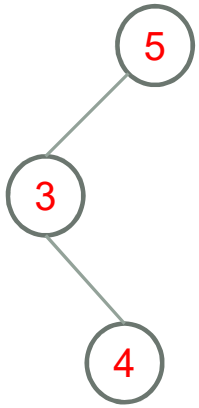
Finding the average case running time

It can be tricky! But here's the process in general:

1. Enumerate all of the possible instantiations of the problem
2. Calculate the "running time" (number of steps, for some definition of "step") for each
3. Take the weighted average of these running times, weighting each by the probability that the instantiation will occur (often we make assumptions which make them all equally likely)

Average case BST successful find: Approach

For a BST, average case analysis is harder, but the principle is the same. Let's consider how we would find the average number of comparisons to successfully find an element in a BST with 3 nodes.

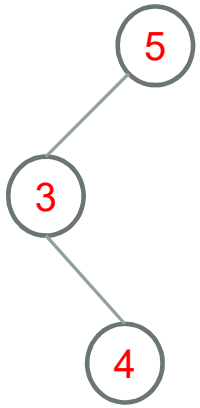


Assume you are looking for one of the elements in this BST. What is the average number of comparisons you need to find it?

A. 1
B. 2
C. 3
D. 4
E. Not enough information

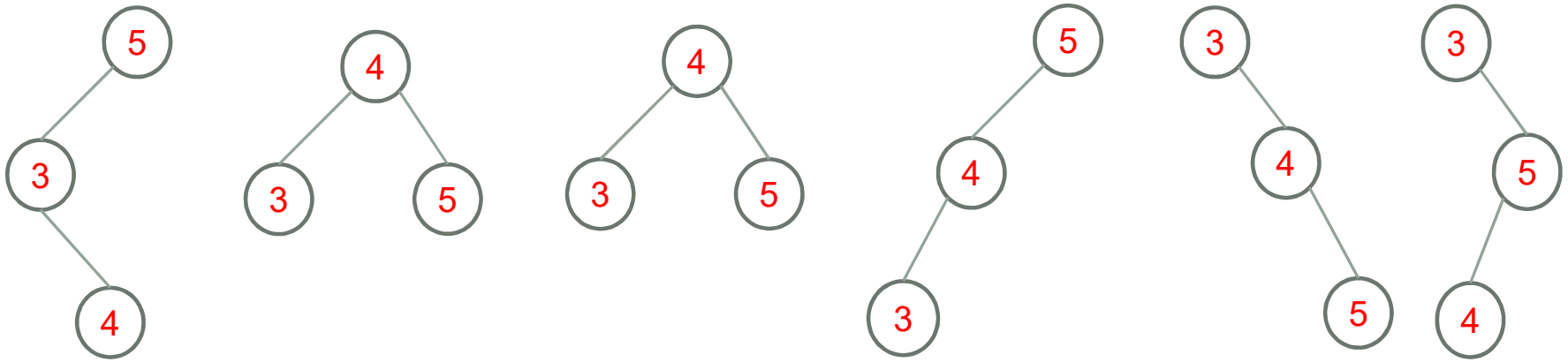
Average case BST successful find: Approach

That was just one specific BST, how many BSTs can result from 3 elements?



Bonus, what determines shape?

Average case BST successful find: Approach



Find the average # of comparisons needed to find an arbitrary element in a *specific* BST

Probabilistic assumption #1: All keys equally likely to be searched for

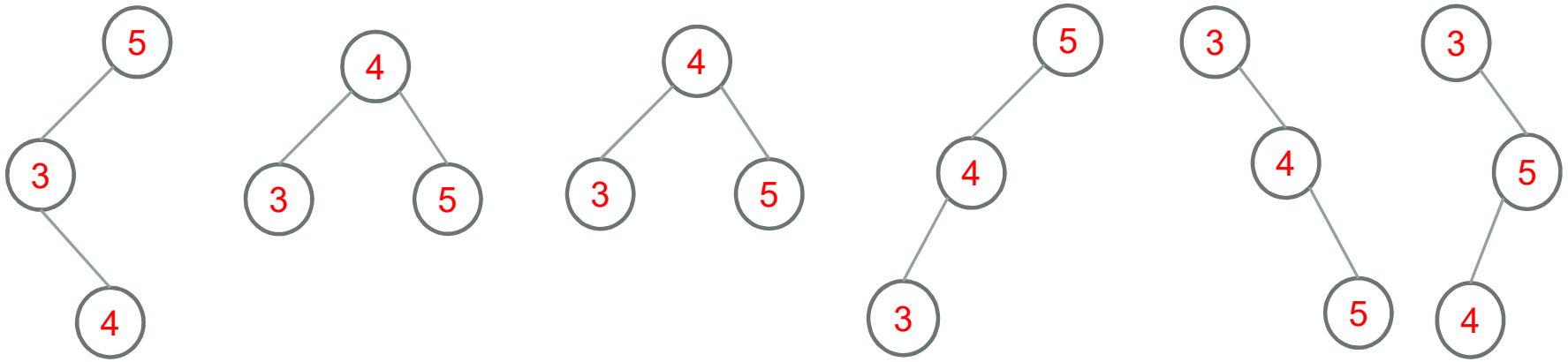
then

Average this value over all possible BSTs with N nodes

Probabilistic assumption #2: All insertion orders are equally likely

What is the average number of comparisons needed to find an element in *any* BST with 3 nodes?

A. 34/6 B. 29/5 C. 3 D. 2 E. 34/18



Find the average # of comparisons needed to find an arbitrary element in a *specific* BST

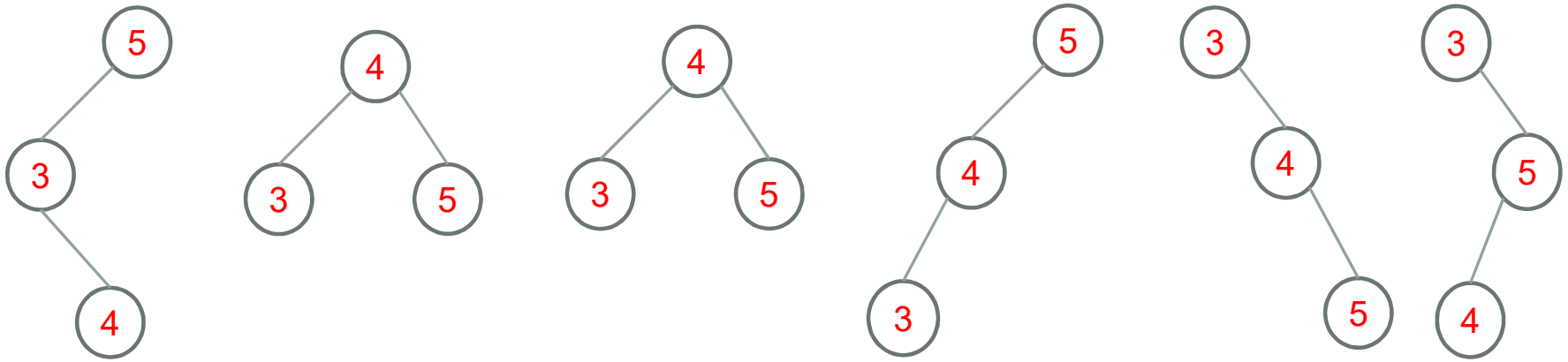
Probabilistic assumption #1: All keys equally likely to be searched for

then

Average this value over all possible BSTs with N nodes

Probabilistic assumption #2: All insertion orders are equally likely

From here you need to turn this into a function of N, which gets complicated. We won't do this here, but see the book if you are interested. The moral is that the average case of BST find is O(log n)



Find the average # of comparisons needed to find an arbitrary element in a *specific* BST

then

Average this value over all possible BSTs with N nodes

Probabilistic assumption #1: All keys equally likely to be searched for

Probabilistic assumption #2: All insertion orders are equally likely

EMPIRICAL RUNNING TIME MEASUREMENT

An alternative/supplement to big-O style analysis

Basic procedure for benchmarking

for problem size N = min,...max

1. initialize the data structure

2. get the current (starting) time

3. run the algorithm on problem size N

4. get the current (finish) time

5. timing = finish time – start time

Benchmarking pitfalls

- Not running enough reps
- Running on too small/too large a problem
- Strange behavior with the first run
- Running on the lab machines!
- And… Bugs in your code!!