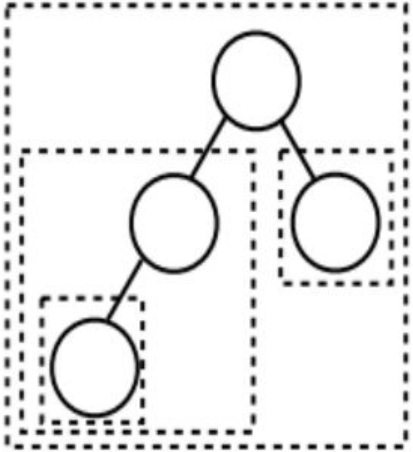
Lecture 16-17-18

Binary Trees
Binary Search Trees
Search in BST

Binary Tree

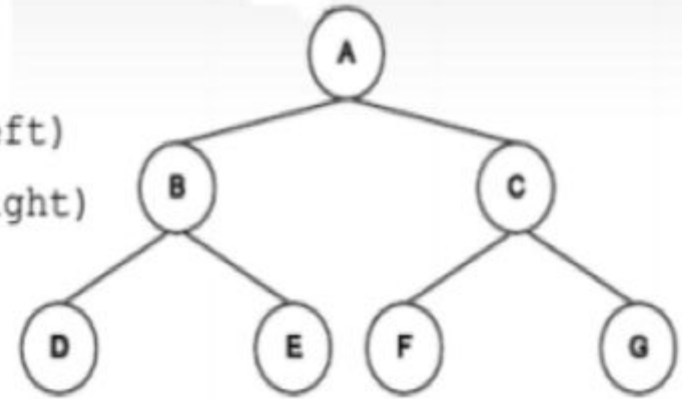
- Each node in a tree is the *root* of its own *sub-tree.*
- The gray boxes below show all possible sub-trees.



Binary Tree Traversal

Pre-order traversal

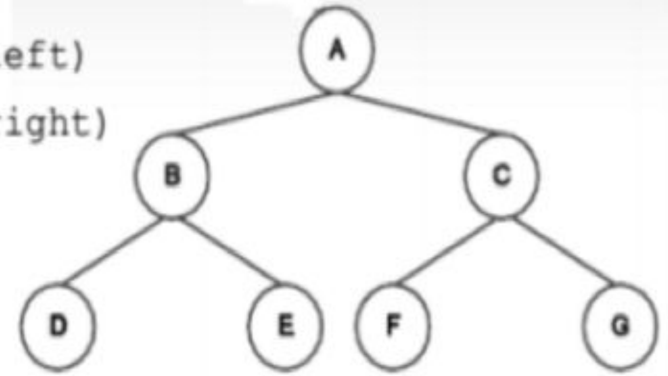
```
preorder(node) {
if (node != null){
      visit this node
      preorder(node.left)
      preorder(node.right)
}
}
```



A. D B E A F C G
B. A B D E C F G
C. A B C D E F G
D. D E B F G C A
E. Other/none/more

Post-order traversal

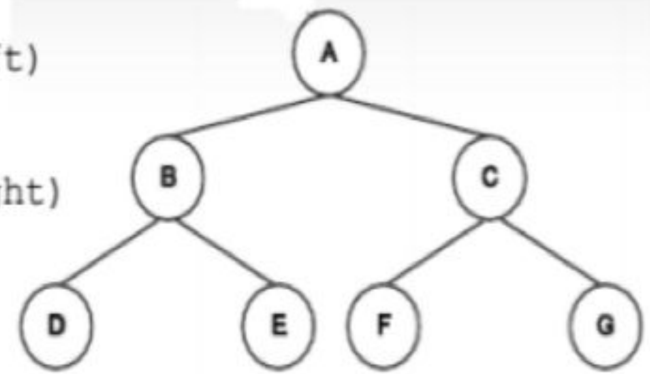
```
postorder(node) {
if (node != null){
      postorder(node.left)
      postorder(node.right)
      visit this node
}
}
```



A. D B E A F C G
B. A B D E C F G
C. A B C D E F G
D. D E B F G C A
E. Other/none/more

In-order traversal

```
inorder(node) {
if (node != null){
      inorder(node.left)
      visit this node
      inorder(node.right)
}
}
```



A. D B E A F C G

B. A B D E C F G

C. A B C D E F G

D. D E B F G C A

E. Other/none/more

Possible?

Give a tree with at least 3 nodes (all nodes must have different keys) such that both its in-order read and its pre-order read are the same, or prove that there is no such tree.

```
inorder(node) {
if (node != null){
      inorder(node.left)
      visit this node
      inorder(node.right)
}
```

```
preorder(node) {
if (node != null){
      visit this node
      preorder(node.left)
      preorder(node.right)
}
```

Binary search trees

Binary SEARCH tree

- A binary search tree (BST) is a binary-tree based data structure that offers O(log n) **average-case** time costs for:
    - Add(element)
    - Find(element)
    - Remove(element)
    - findLargest/removeLargest(element)

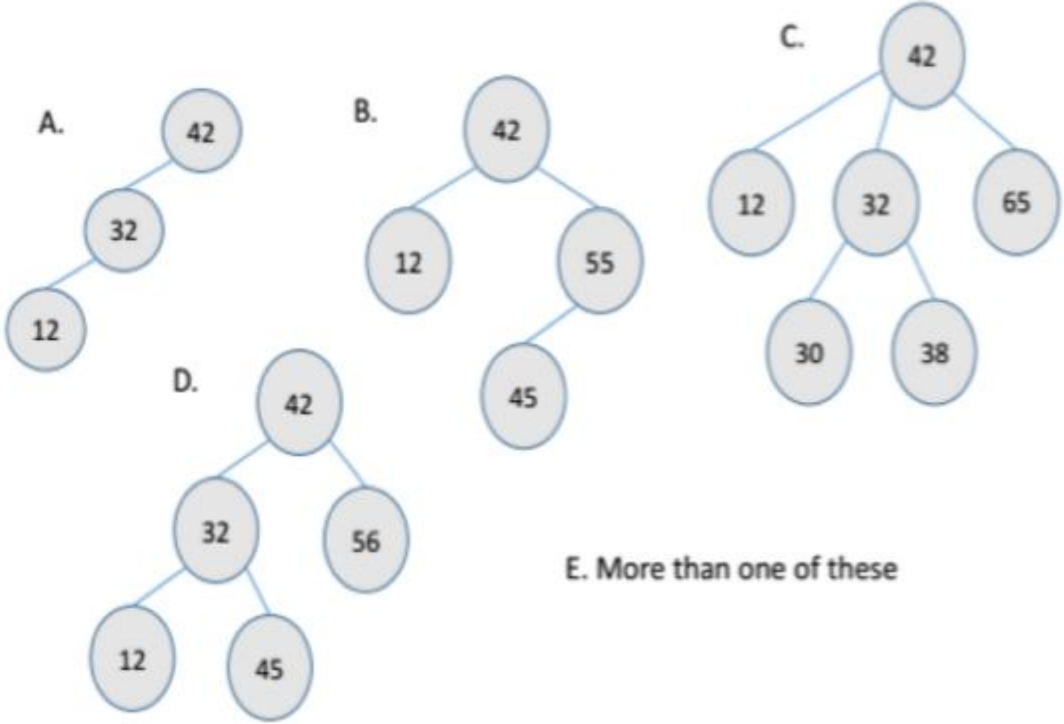
BST

- Binary Search Tree is a node-based binary tree data structure which has the following properties:
    - The left subtree of a node contains only nodes with keys smaller than the node's key.
    - The right subtree of a node contains only nodes with keys greater than the node's key.

Reminders

- Mic
- If main with magic numbers and lost points, let me know (PA04)

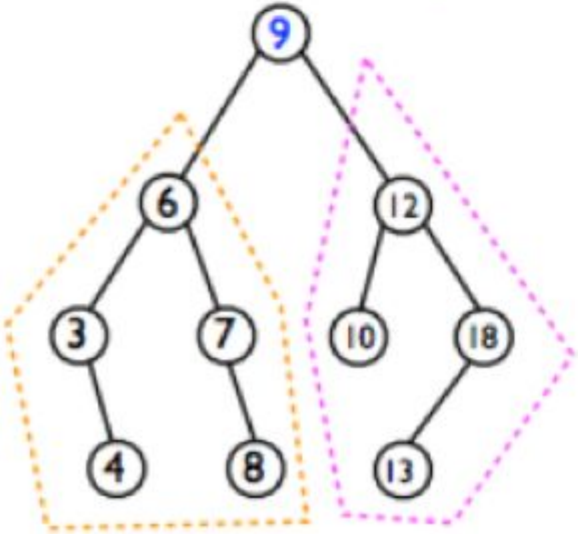
Which of the following is/are a binary search tree?



E. More than one of these

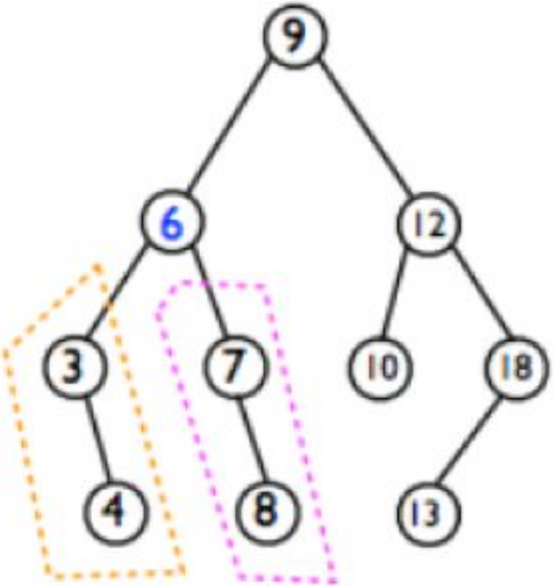
Binary search trees

Left sub-tree < Node (9) < Right sub-tree



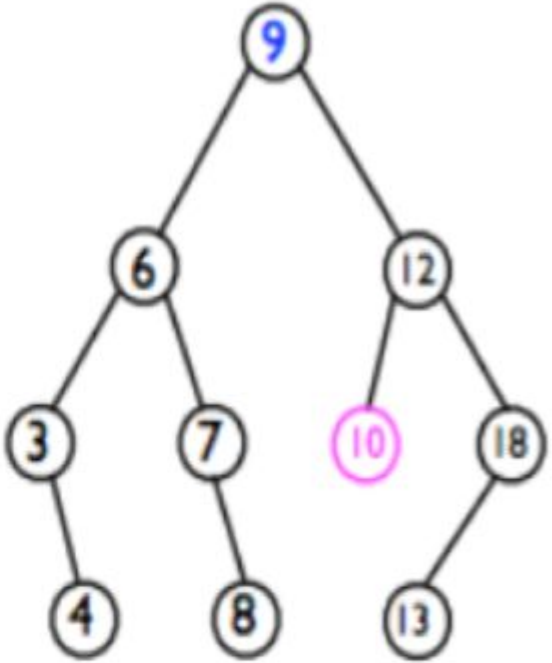
Binary search trees

Left sub-tree < Node (6) < Right sub-tree



Where to find a min element?

- In a given a binary tree?



BST IMPLEMENTATION

CompareTo

compareTo, simplified

- Allows us to compare Objects: Students, Cars, Apples, Animals…
- In order to use this method, one needs to let Java know how to compare:
  - https://docs.oracle.com/javase/9/docs/api/java/lang/Comparable.html#compareTo-T-

- If the values are the same then 0 is returned.
- If one value is less than the argument then the *negative* integer is returned.
- If one value is greater than the argument then the *positive* integer is returned.

Example

```
public class Test {

    public static void main(String args[]) {
        Integer x = 5;

        System.out.println(x.compareTo(3));    # 1
        System.out.println(x.compareTo(5));    # 0
        System.out.println(x.compareTo(8));    # -1
    }
}
```

Example

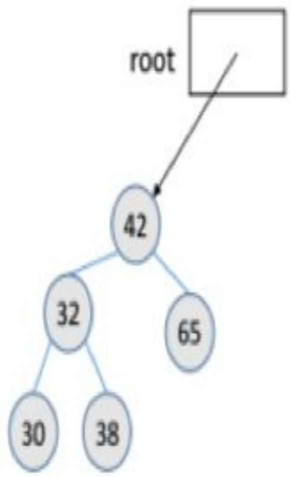
```
public class Student {
    int age;

    int compareTo(Student st)  {
        if this.age == st.age return 0;
        if this.age > st.age return 1;
        If this.age < st.age  return
-1;
       ...
    }

    }
```

```
public static void main(String args[])
{
        Student st1 = new Student(19);
        Student st2 = new Student(22);
        SOS(st1.compareTo(st2));    # -1
```

BST implementation

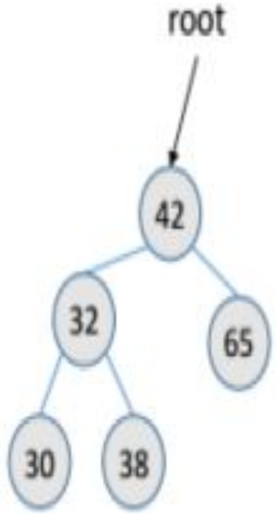
The BST and BSTNode Classes

```
public class BST<E extends Comparable<? super E>>
{
    /** Inner class for the BSTNode */
    private class BSTNode {
        protected BSTNode leftChild;
        protected BSTNode rightChild;
        protected E element;

        public BSTNode(E elem)
        {
            element = elem;
        }
    }
    protected BSTNode root;
```



BST Contains: Let's write it!

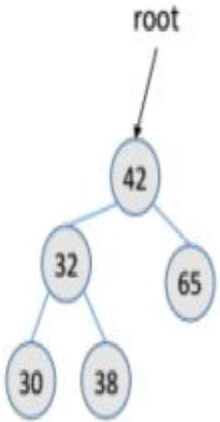
```
// Return true if toFind is in the BST
public boolean contains(E toFind) {
```



contains(32)?

BST Contains: Let's write it!

```
// Return true if toFind is in the BST
public boolean contains(E toFind) {
    //RECURSION!
    return containsHelper(root, toFind);
}

// This recursive method returns true if toFind is in the
// tree rooted at currRoot, and false otherwise
private boolean containsHelper(BSTNode currRoot, E toFind)
{
   // To write!
}
```



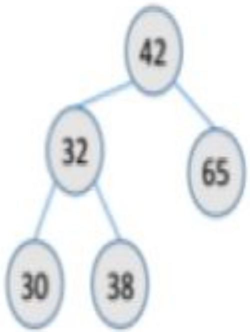
contains(32)?

BST Contains: Let's write it!

```
// Return true if toFind is in the BST rooted at currRoot,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
```

Base case(s): When do we know we are done?

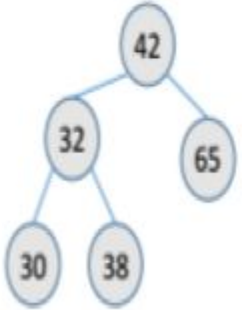
A. toFind is less than currRoot's element
B. toFind is greater than currRoot's element
C. toFind is equal to currRoot's element
D. currRoot is null
E. More than one of these



BST Contains: Let's write it!

```
// Return true if toFind is in the BST rooted at currRoot,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
```

Base case 1: (sub)tree is empty, so we know currRoot is not in it



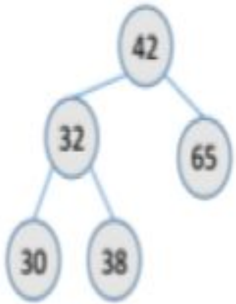
BST Contains: Let's write it!

```
// Return true if toFind is in the BST rooted at currRoot,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
    if (currRoot == null) return false;
```

Base case 2: Element is found

We will roll this in with our recursive step

So what is our recursive step...?



contains(32)?

contains(65)?

contains(42)?

contains(40)?

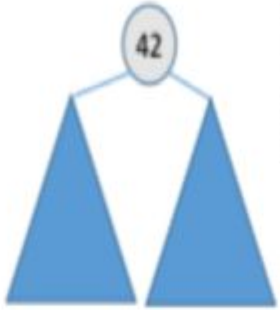
BST Contains: Let's write it!

```
// Return true if toFind is in the BST rooted at currRoot,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
    if (currRoot == null) return false;
```

Base case 2: Element is found

We will roll this in with our recursive step

So what is our recursive step...?



contains(32)?
contains(65)?
contains(42)?
contains(40)?

```
rn true if toFind is in the BST rooted at currRoot,
e otherwise
 contains(BSTNode currRoot, E toFind) {
(currRoot == null) return false;
( ______________________________ )
eturn contains( ____________, toFind );
e if ( ______________________________ )
eturn contains( ____________, toFind );
e
eturn ____________;
```

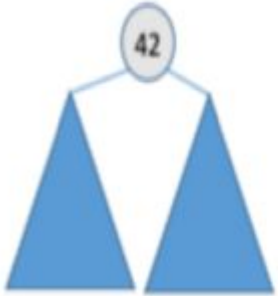
rsive step and base case 2: How do you know which
to go? Fill in the blanks above. Hint: use compareTo.
u need another hint, check out the next slide.

contains(32)?
contains(65)?
contains(42)?
contains(40)?

```
// Return true if toFind is in the BST rooted at root,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
   if ( currRoot == null ) return false;
   if ( toFind.compareTo(currRoot.getElement()) < 0 )
      return _______1______________________;
   else if (toFind.compareTo(currRoot.getElement()) > 0 )
      return ________2_____________________;
   else
      return ________3_____________________;
```

contains(32)?
contains(65)?
contains(42)?
contains(40)?

```
// Return true if toFind is in the BST rooted at root,
// false otherwise
boolean contains(BSTNode currRoot, E toFind) {
   if ( currRoot == null ) return false;
   if ( toFind.compareTo(currRoot.getElement()) < 0 )
      return contains(currRoot.getLeft(), toFind);
   else if (toFind.compareTo(currRoot.getElement()) > 0 )
      return contains(currRoot.getRight(), toFind);
   else
      return true;
```

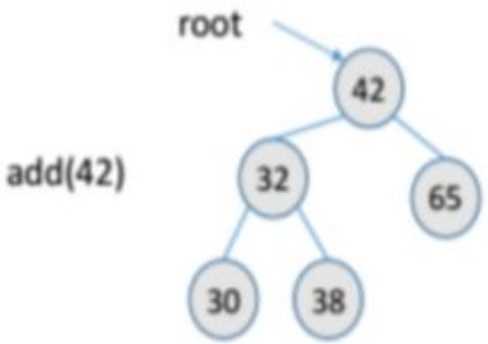


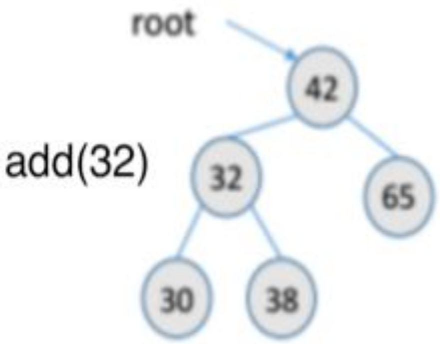
contains(32)?
contains(65)?
contains(42)?
contains(40)?

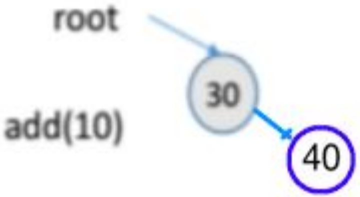
Insert

BST Add: With recursion!

Consider the following:







Do we need recursion for the third one?

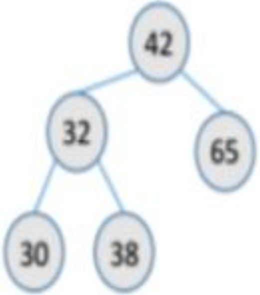
A: Yes
B: No

BST Add: Recursively

```
boolean add( E toAdd ) {
  if (root ==  null) {
    root = new BSTNode(toAdd);
    return true;
  }
   return addHelper( root, toAdd );
}

boolean addHelper( BSTNode currRoot, E toAdd )
{
     ...
}
```

Which of these is/are a base case for addHelper?

A. currRoot is null
B. currRoot's element is equal to toAdd
C. Both A & B
D. Neither of these

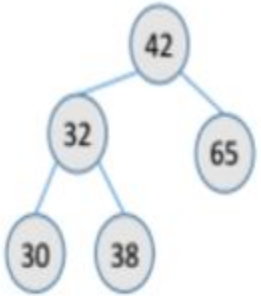


BST Add: Recursively

```
boolean add( E toAdd ) {
  if (root == null) {
    root = new BSTNode(toAdd);
    return true;
  }
   return addHelper( root, toAdd );
}
```

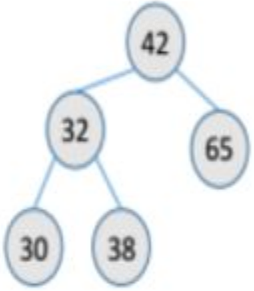
```
boolean addHelper( BSTNode currRoot, E toAdd )
{
    int compare = toAdd.compareTo(currRoot.getElement());
    if (compare == 0) {
        return                    ;
    }
}
```

Which of these is/are a base case for addHelper?
currRoot's element is equal to toAdd



BST Add: Recursively

```
boolean add( E toAdd ) {
  if (root ==  null) {
    root = new BSTNode(toAdd);
    return true;
  }
   return addHelper( root, toAdd );
}

boolean addHelper( BSTNode currRoot, E toAdd )
{
    int compare = toAdd.compareTo(currRoot.getElement());
    if (compare == 0) {
        return false;
    }
    // Finish the code…

}
```



Which of these is/are a base case for addHelper?
currRoot's element is equal to toAdd

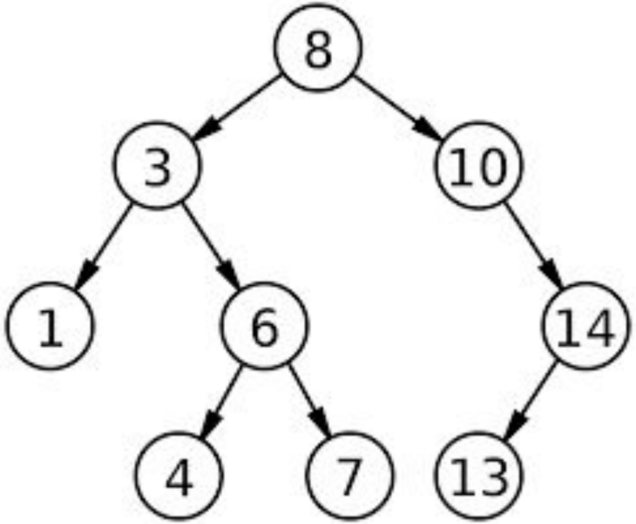
BST Add: Recursively

```
void addHelper( BSTNode currRoot, E toAdd )
{
   int value = toAdd.compareTo( currRoot.getElement() );
   if (value == 0) return false;
   if ( value < 0 ) {
      if ( currRoot.getLeftChild() == null ) {
         currRoot.setLeftChild(new BSTNode( toAdd ));
      }
      else {
         addHelper( currRoot.getLeftChild(), toAdd );
      }
   }
   else { // ( value > 0 )
      // Repeat for other side
   }
   return true;
}
```

Duplicate keys

- Allow them and be consistent with the rule
- Node is a linked (Array) list. All duplicates are linked.

How to delete an element from a BST


8
3
10
1
6
14
4
7
13

Lazy deletion

Complexity

Complexity

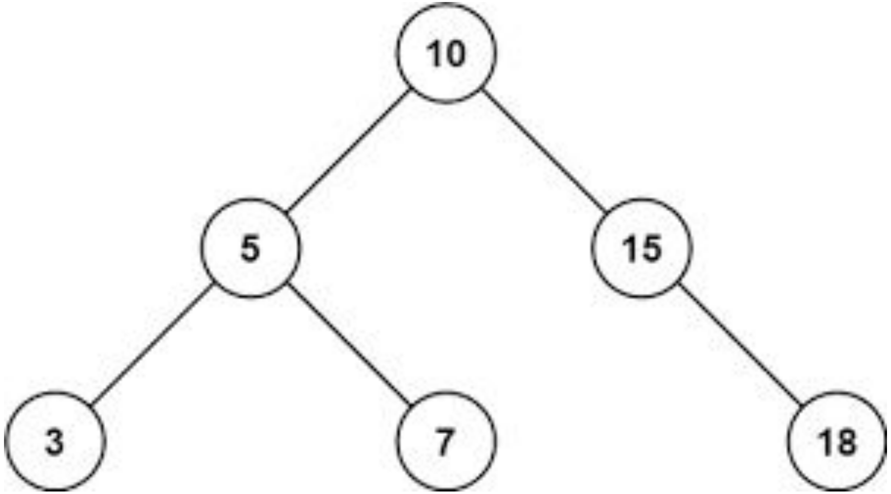
What is the BEST CASE cost for doing find() in BST

A. O(1)
B. O(log n)
C. O(n)
D. O(n log n)
E. $O(n^2)$

What is the WORST CASE cost for doing find() in a BST?

A. $O(1)$
B. $O(\log n)$
C. $O(n)$
D. $O(n \log n)$
E. $O(n^2)$

What is the output?




```
inorder(node) {
if (node != null){
    inorder(node.left)
    visit this node
    inorder(node.right
}
```

A: 10, 5, 3, 7, 15, 18

B: 3, 5, 7, 10, 15, 18

C: 3, 7, 5, 18, 15, 10

D: 3, 5, 7, 15, 10, 18

E: Something else

Iterators

What is the time complexity?

```
for (int i = 0; i< linkedlist.size(); i++) {
        elem  = linkedlist.getElem( i ) ;
        print(elem);
}
```

A: O(1)

B: O(n)

C: O(n log n)

D: O(n^2)

E: Something else

Iterators in Java, performance benefits

- An “iterator” object helps us to avoid this wasted computation.
- An iterator is a “helper object” with which the user can iterate across all elements in a data structure.
- The iterator will “remember” where it left off.
- Even very different data structures --e.g., graphs and lists -- can both support iterators.
- Python has them as well

Iterator interface

In Java, the `Iterator` interface contains three (well, four now) method signatures:

- boolean hasNext();
- E next();
- void remove();

[Java documentation]

Iterator for a Linked List

```
import java.util.*;

class IteratorExample {

   public static void main(String [] args) {

     // Create an linked list

     LinkedList <Integer> intList = new LinkedList <>();

     for (int i=10; i<20; i++)

          intList.add(i);
```

```java
import java.util.*;

class IteratorExample {
    public static void main(String [] args) {
      // Create an linked list
      LinkedList <Integer> intList = new LinkedList <>();
      for (int i=10; i<20; i++)
          intList.add(i);

//To start, we need to obtain an Iterator from a Collection; this is done by calling the iterator() method.
// we'll obtain Iterator instance from a list:


 Iterator itr = intList.iterator();  // created the iterator object
```

```java
import java.util.*;

class IteratorExample {
     public static void main(String [] args) {
      // Create an linked list
      LinkedList <Integer> intList = new LinkedList <>();
      for (int i=10; i<20; i++)
          intList.add(i);
     Iterator itr = intList.iterator();  // created the iterator object
     while(itr.hasNext()) {

          Object element = itr.next();

          System.out.print(element + " ");

     }

     System.out.println();
```

- boolean hasNext();
- E next();
- void remove();

```java
import java.util.*;

class IteratorExample {
     public static void main(String [] args) {
      // Create an linked list
      LinkedList <Integer> intList = new LinkedList <>();
      for (int i=10; i<20; i++)
          intList.add(i);
     Iterator itr = intList.iterator();  //created the iterator object
     while(itr.hasNext()) {

          Object element = itr.next();

          System.out.print(element + " ");

        }

        System.out.println();
```

10 11 12 13 14 15 16 17 18 19

```java
import java.util.*;
```

https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html

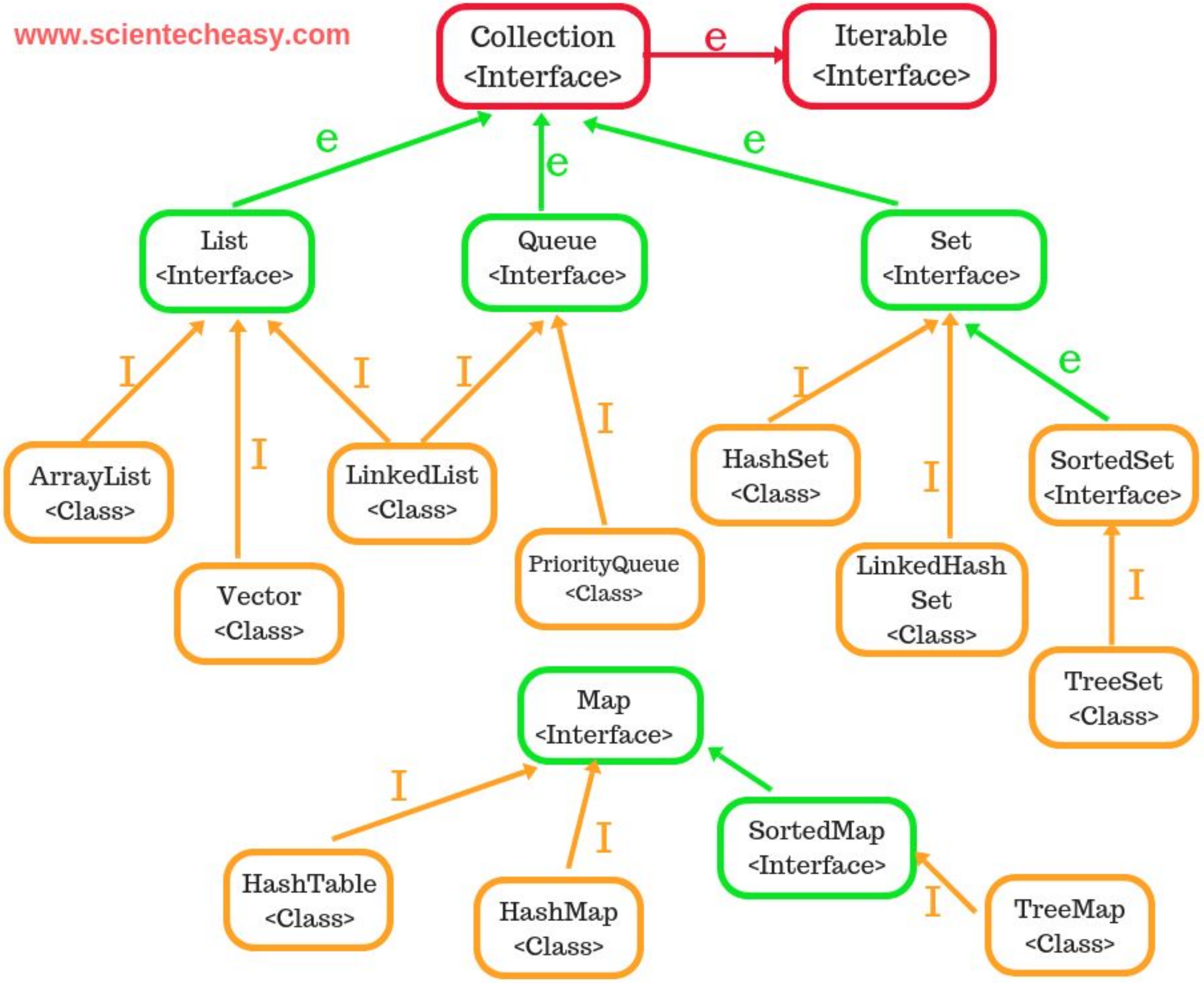
```java
class IteratorExample {
     public static void main(String [] args) {
       // Create an linked list
      LinkedList <Integer> intList = new LinkedList <>();
      for (int i=10; i<20; i++)
          intList.add(i);
     Iterator itr = intList.iterator();  // created the iterator object
     while(itr.hasNext()) {

          Object element = itr.next();

          System.out.print(element + " ");

        }

        System.out.println();
```

10 11 12 13 14 15 16 17 18 19

Iterators for other data structures

- When you develop your own data structure, you want a user to be able to iterate over it.
- When the data structure is linear (lists, arrays, sets) then it is clear what the next element is.
- But if you have a tree? Or a graph?

Iterable interface



Iterable interface

- The `Collection<E>` interface extends the `Iterable<E>` interface, which is defined as follows:

```
public interface Iterable<E> {
   public Iterator<E> iterator();
}
```

- So any class that implements `Collection<E>` must define an instance method `iterator()` that returns an `Iterator<E>` object for that instance
- And `Iterator<E>` is also an interface in the JCF…

Interface Iterator

- In Java, the `Iterator` interface contains three method signatures:

boolean hasNext();

E next();

void remove();

- The ListIterator interface adds a few more methods.

boolean hasPrevious();

E previous();

- ...

In HW6

```
public class BSTree<T extends Comparable<? super T>> implements Iterable
```

Example: how to create an iterator

To implement an iterable data structure, we need to:

1. Implement `Iterable` interface along with its methods in the said Data Structure

```
public interface Iterable<E> {
   public Iterator<E> iterator();
}
```

Example: how to create an iterator

To implement an iterable data structure, we need to:

1. Implement `Iterable` interface along with its methods in the said Data Structure

2. Create an `Iterator` class which implements `Iterator` interface and corresponding methods.

- boolean hasNext();
- E next();
- void remove();

Iterator: every second item in ArrayList

```
import java.util.*;

public class A implements Iterable{

    ArrayList  arr = new ArrayList();

    public Iterator iterator() {
        return new ArrayList_It();  # another object!
}
```

```
public interface Iterable<E> {
  public Iterator<E> iterator();
}
```

```
import java.util.*;
public class A implements Iterable{
     ArrayList  arr = new ArrayList();
     public Iterator iterator() {
          return new ArrayList_It();  # another object!}
```

```
# inner class
class ArrayList_It implements Iterator{

     int cursor;

     public ArrayList_It() {
           cursor = 0;
     }
```

```
import java.util.*;
public class A implements Iterable{
    ArrayList  arr = new ArrayList();
    public Iterator iterator() {
        return new ArrayList_It();  # another object!}
```

```
# inner class
class ArrayList_It implements Iterator{

    int cursor;

    public ArrayList_It() {
        cursor = 0;
    }
```

- boolean hasNext();
- E next();
- void remove();

```java
import java.util.*;
public class A implements Iterable{
     ArrayList  arr = new ArrayList();
     public Iterator iterator() {
        return new ArrayList_It();  # another object!}
```

```java
# inner class
class ArrayList_It implements Iterator{

     int cursor;

     public ArrayList_It() {
           cursor = 0;
     }

     public boolean hasNext() {
     }
```

- boolean hasNext();
- E next();
- void remove();

```
import java.util.*;
public class A implements Iterable{
     ArrayList  arr = new ArrayList();
     public Iterator iterator() {
         return new ArrayList_It();  # another object!}
```

```
# inner class
class ArrayList_It implements Iterator{

     int cursor;

     public ArrayList_It() {
           cursor = 0;
     }

     public boolean hasNext() {
          if (arr.size() == 0)
              return false;
     }
```

- boolean hasNext();
- E next();
- void remove();

```java
import java.util.*;
public class A implements Iterable{
     ArrayList  arr = new ArrayList();
     public Iterator iterator() {
         return new ArrayList_It();  # another object!}

# inner class
class ArrayList_It implements Iterator{

     int cursor;

     public ArrayList_It() {
           cursor = 0;
     }

     public boolean hasNext() {
          if (arr.size() == 0)
               return false;
           if (cursor==arr.size()-1)
               return true;
          if (cursor <= arr.size()-2) // needs to be tested
               return true;
           else
               return false;
     }
```

- boolean hasNext();
- E next();
- void remove();

```
import java.util.*;
public class A implements Iterable{
     ArrayList  arr = new ArrayList();
     public Iterator iterator() {
        return new ArrayList_It();  # another object!}
```

```
# inner class
class ArrayList_It implements Iterator{

     int cursor;

     public ArrayList_It() {
           cursor = 0;
     }

      public Object next() {

          int to_return = (Integer) arr.get(cursor);
          cursor = cursor + 2;
          return to_return;
     }
```

- boolean hasNext();
- E next();
- void remove();