Inheritance and binary search

4/29/26

Administrivia

- HW 5 (sorting and search order) due Friday night (5/1)

Inheritance

- extends: makes class a subtype of another
  - terminology: subclass and superclass
  - subclass has all attributes and methods of superclass
    - subclass can override methods it wants to change
  - references to subclass can be stored in superclass-typed variables

Inheritance

- extends: makes class a subtype of another
  - terminology: subclass and superclass
  - subclass has all attributes and methods of superclass
    - subclass can override methods it wants to change
  - references to subclass can be stored in superclass-typed variables

- access control
  - public: anyone can access it
  - private: can only access from within the class
  - protected: accessible from within class or a subclass

  As a general rule, be as restrictive as possible

Which of the following lines has a compile-time error?

```
class Person {  private String name;    }

class Student extends Person {                         //A
    private double credits;
    private double qualityPts;

    public double gpa() {   return qualityPts / credits;    }

    public String toString() {
        return name + " (GPA: " + gpa() + ")";     //B
    }
...
}
...
    Person p = new Student();                       //C
    System.out.println(p.gpa());                    //D
                                                    //E: Not exactly one of the above
```

Which of the following lines has a compile-time error?

```
class Person {  private String name;    }

class Student extends Person {                              //A
        private double credits;
        private double qualityPts;

        public double gpa() {   return qualityPts / credits;    }

        public String toString() {
                return name + “ (GPA: “ + gpa() + “)”;      //B
        }
...
}
...
        Person p = new Student();                           //C
        System.out.println(p.gpa());                        //D
                                                            //E: Not exactly one of the above (B & D)
```

toString

- Method inherited from Object
  - Called when an object is printed
  - Comes with default implementation, but can be overridden by something more useful

Implementing contains (ints)

```
boolean contains(int key) {
    //returns whether value is in the set
    for(int i=0; i<size; i++)
        if(vals[i] == key)
            return true;
    return false;
}
```

Implementing contains (Objects)

```
boolean contains(E key) {
    //returns whether value is in the set
    for(int i=0; i<size; i++)
        if(vals[i].equals(key))
            return true;
    return false;
}
```

Aside: Implementing .equals

```
public class Pair {  //represents a pair of ints
    private int x;  private int y;
    ...
    public boolean equals(Object o) {
        if(o instanceof Pair) {
            Pair p = (Pair) o;
            return (x == p.x) && (y == p.y);
        }
        return false;
    }
}
```

Recall: Set ADT

- Represents unordered collection
  {“hello”, “there”, “=)”, “CS 142”}
- **No duplicate elements**

- Supports add, contains, iterator
  Maybe also: size, remove, clear

Map ADT: Add value attached to each item (key)
- Supports put, get, remove, …

Array-based implementation of Set

- Store the keys in arbitrary order in an array, keeping the size in another variable

  (Basically an ArrayList, but no duplicates)

Array-based implementation of Set

- Store the keys in arbitrary order in an array, keeping the size in another variable

  (Basically an ArrayList, but no duplicates)

Fill in the following: With this implementation, add runs in ____ while contains runs in ____

A. O(1), O(1)
B. O(1), O(n)
C. O(n), O(1)
D. O(n), O(n)
E. None of the above

Array-based implementation of Set

- Store the keys in arbitrary order in an array, keeping the size in another variable

  (Basically an ArrayList, but no duplicates)

Fill in the following: With this implementation, add runs in ____ while contains runs in ____

A. O(1), O(1)
B. O(1), O(n)
C. O(n), O(1)
D. O(n), O(n)
E. None of the above

What if the list is kept sorted?

How can you search faster?

Binary search

```
int rank(E key) {        //returns where key goes in keys
    int lo = 0;          //lowest index that could have key
    int hi = size-1;     //highest index that could have key
    while(lo <= hi) {
        int mid = lo + (hi – lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if(cmp < 0)         hi = mid – 1;
        else if(cmp > 0) lo = mid + 1;
        else return mid;
    }
    return lo;
}
```

Binary search

```
int rank(E key) {        //returns where key goes in keys
    int lo = 0;          //lowest index that could have key
    int hi = size-1;     //highest index that could have key
    while(lo <= hi) {
        int mid = lo + (hi – lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if(cmp < 0)         hi = mid – 1;
        else if(cmp > 0) lo = mid + 1;
        else return mid;
    }
    return lo;
}
```

How long does this take?

A) O(1)
B) O(log n)
C) O(n)
D) O(n^2)
E) None of the above

Binary search

```
int rank(E key) {        //returns where key goes in keys
    int lo = 0;          //lowest index that could have key
    int hi = size-1;     //highest index that could have key
    while(lo <= hi) {
        int mid = lo + (hi – lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if(cmp < 0)      hi = mid – 1;
        else if(cmp > 0) lo = mid + 1;
        else return mid;
    }
    return lo;
}
```

How long does this take?

A) $O(1)$
B) $O(\log n)$
C) $O(n)$
D) $O(n^2)$
E) None of the above

How long does it take to add a new key into the sorted array?

A. O(1)
B. O(1) amortized
C. O(log n)
D. O(n)
E. None of the above

How long does it take to add a new key into the sorted array?

A. O(1)
B. O(1) amortized
C. O(log n)
D. O(n)
E. None of the above

Fill in the following: With a sorted linked list implementation of a set, add runs in ____ while contains runs in ____

A. O(1), O(1)
B. O(1), O(log n)
C. O(log n), O(1)
D. O(n), O(log n)
E. None of the above

Fill in the following: With a sorted linked list implementation of a set, add runs in _____ while contains runs in _____

A. O(1), O(1)
B. O(1), O(log n)
C. O(log n), O(1)
D. O(n), O(log n)
E. None of the above: O(n), O(n)

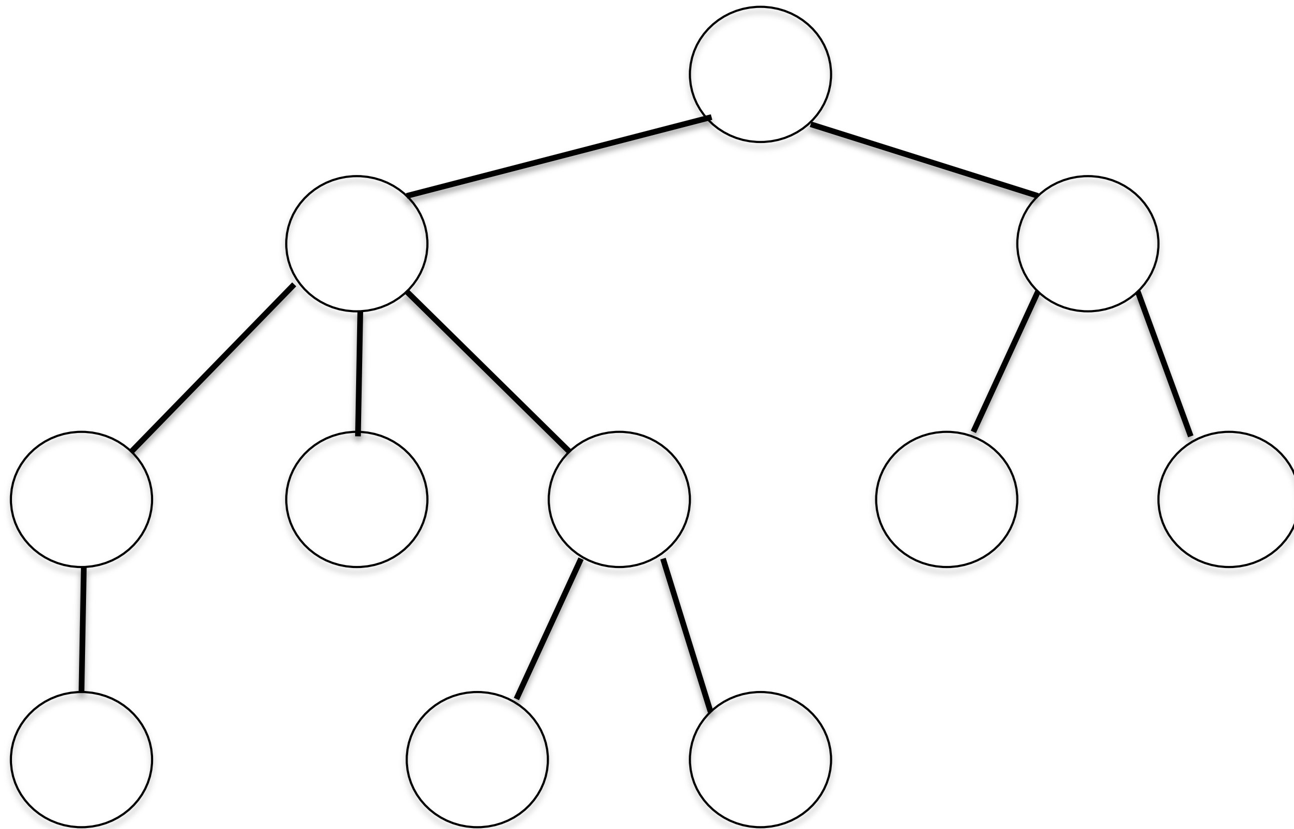
Overview of set implementations

- Array w/ vals in arbitrary order, but no duplicates
  - O(n) add, O(n) contains
- Array with values in sorted order
  - Allows binary search (O(log n)), but slow insertion (O(n))
- Linked list
  - Can more easily add in the middle, but contains is slow

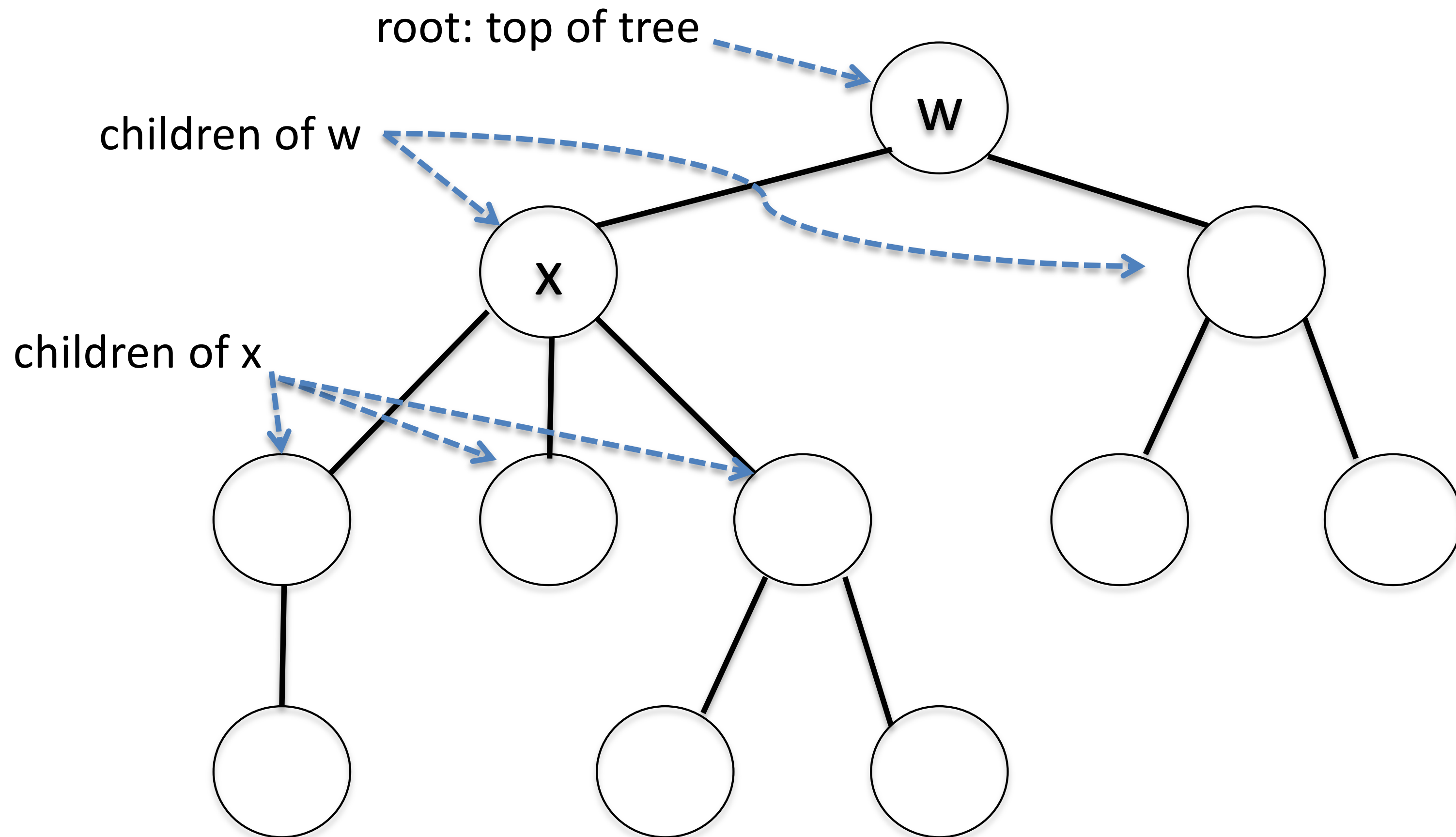
Desired properties

- Ability to quickly “jump” later in the sequence of keys (like binary search)
- Ability to easily add keys into the appropriate position (like linked lists)

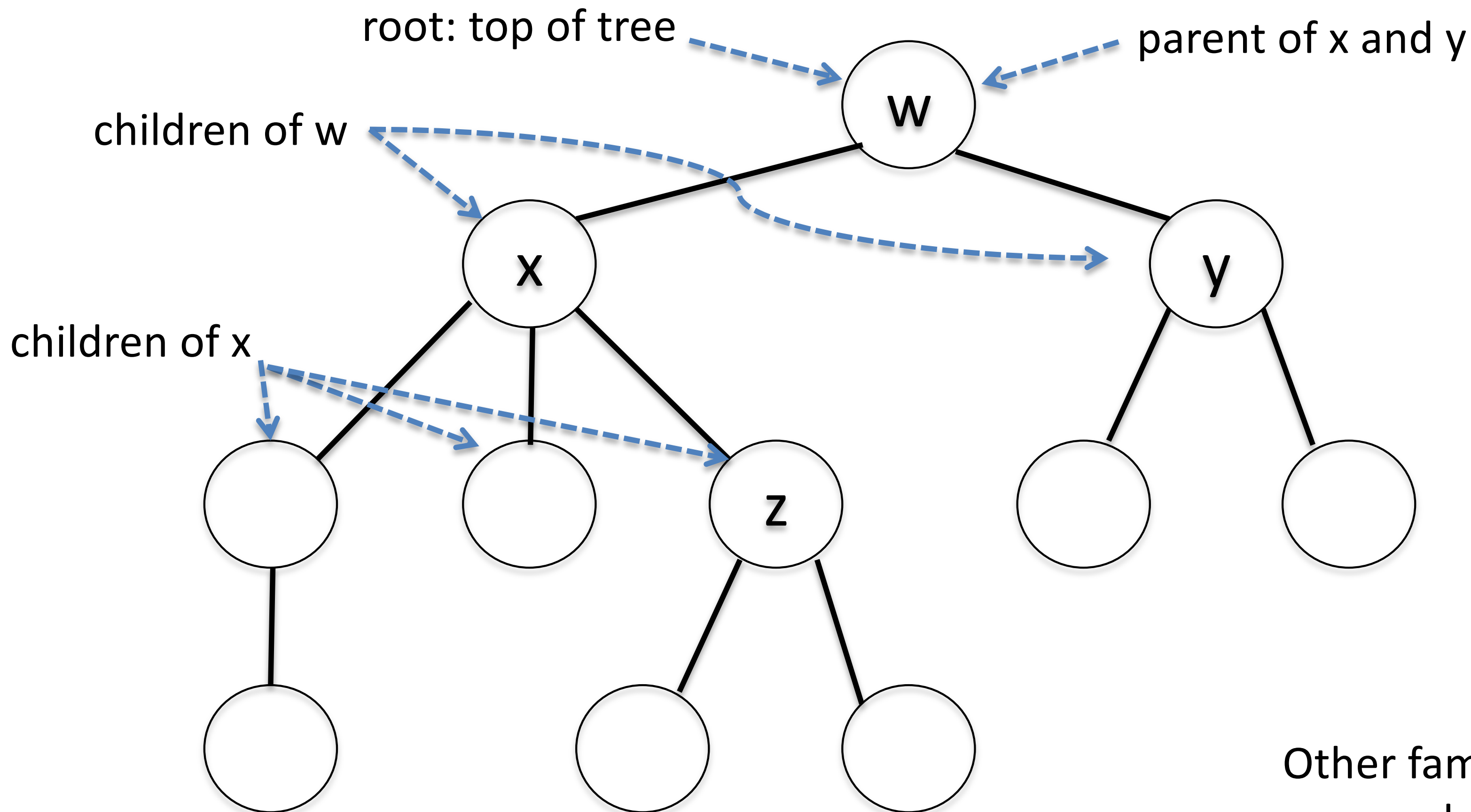
Trees



Trees



Trees



Other familial relationships:

x and y are siblings

w is grandparent of z

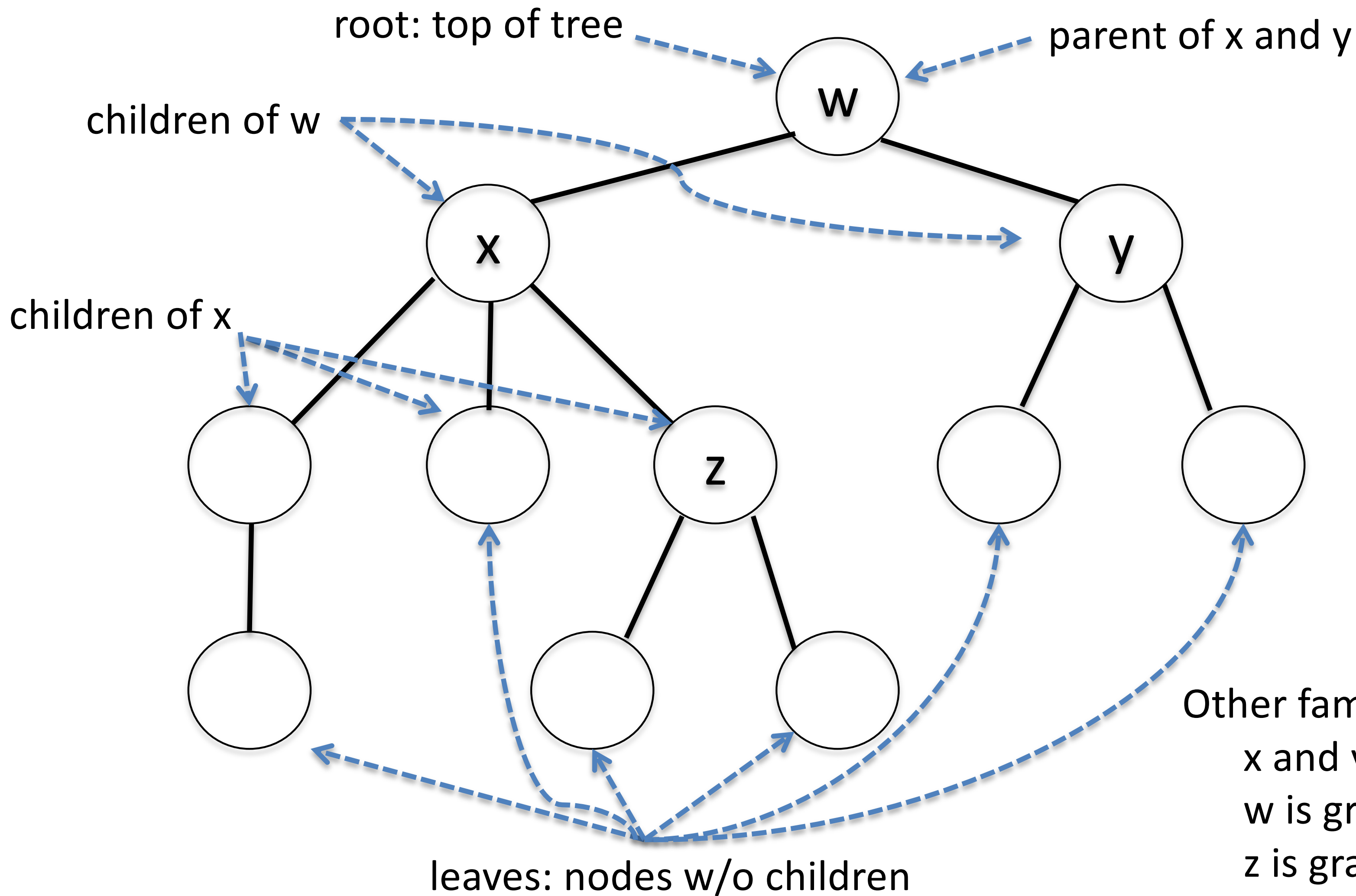
z is grandchild of w

y is uncle (or aunt) of z

ancestors are nodes toward root

descendants are nodes away from it

Trees



Other familial relationships:

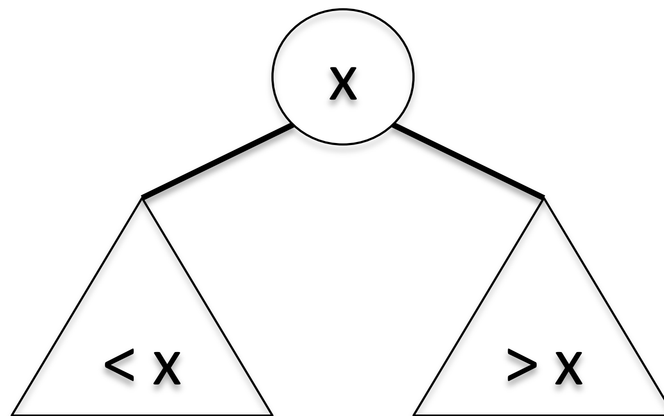
- x and y are siblings
- w is grandparent of z
- z is grandchild of w
- y is uncle (or aunt) of z

ancestors are nodes toward root

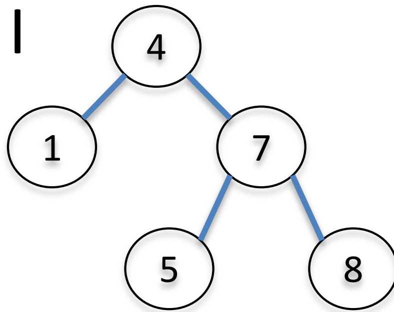
descendants are nodes away from it

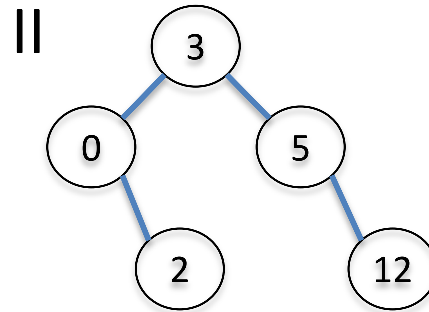
Binary Search Trees (BSTs)

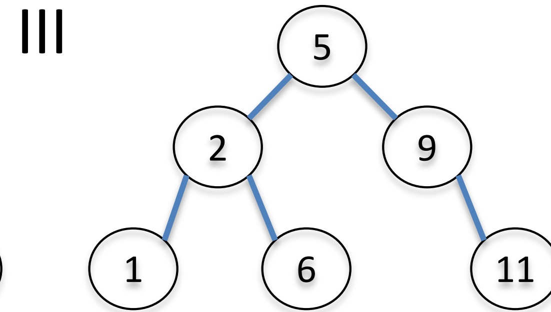
- Tree with key stored at each node such that
  - Every node has 2 children (left and right)
    - Children can be null (nodes have 0, 1, 2 actual children)
  - All keys in a node's right subtree are greater than its key and all in the left subtree are less



Which of the following is a legal BST?

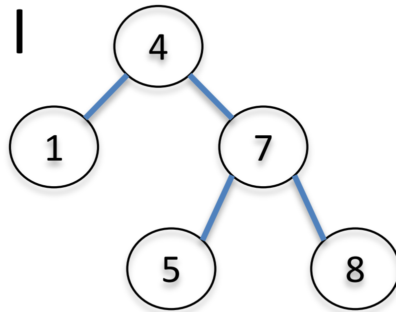


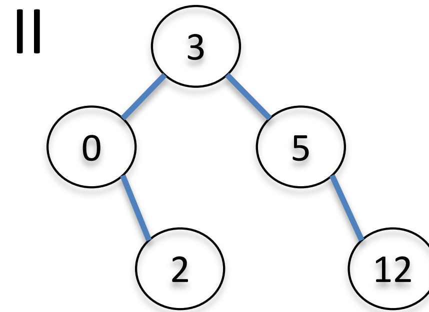


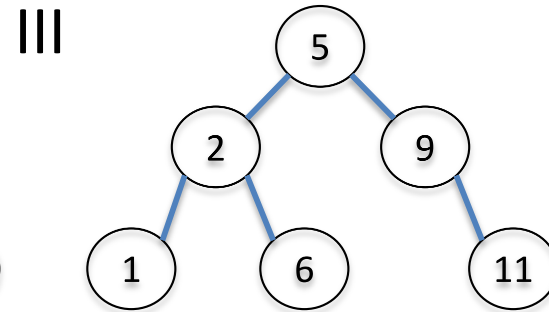


A. I only
B. I and II only
C. II and III only
D. All of them
E. None of the choices above is correct

Which of the following is a legal BST?







A. I only
B. I and II only
C. II and III only
D. All of them
E. None of the choices above is correct

Implementing a BST

```
public class BST<E extends Comparable<E>> {
    Node root;

    private class Node {
        Node left;
        Node right;
        E key;
    }
}
```

How can a BST implement contains?

How can a BST implement contains?

```
boolean contains(E key) {
    Node curr = root;
    while(curr != null) {
        int cmp = key.compareTo(curr.key) ;
        if(cmp < 0) curr = curr.left;
        else if(cmp > 0) curr = curr.right;
        else return true;
    }
    return false;
}
```

Recursive implemention

```
boolean contains(E key) {
    return contains_helper(root, key);
}

boolean contains_helper(Node curr, E key) {
    if(curr == null) return false;
    int cmp = key.compareTo(curr.key) ;
    if(cmp < 0) return contains_helper(curr.left, key);
    else if(cmp > 0) returns contains_helper(curr.right, key);
    else return true;
}
```