CS 10:
Problem solving via Object Oriented Programming

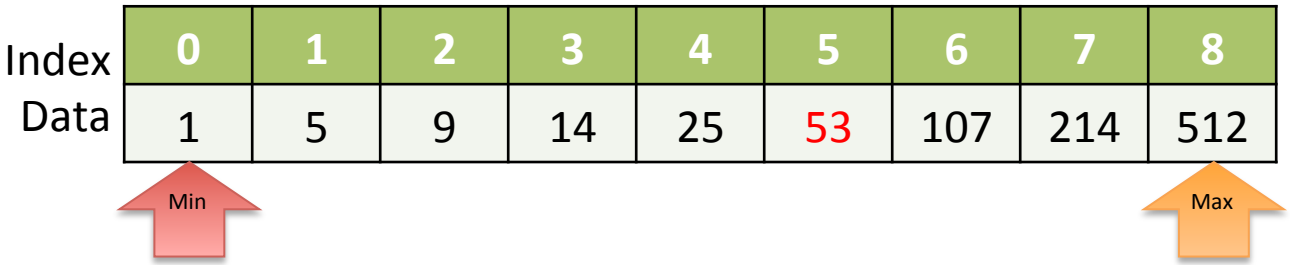
Hierarchies Part 2

Agenda

1. Binary search
2. Binary Search Trees (BST)
3. BST find analysis
4. Operations on BSTs
5. Implementation

Binary search can quickly find items if the data is ordered

Binary search on an array



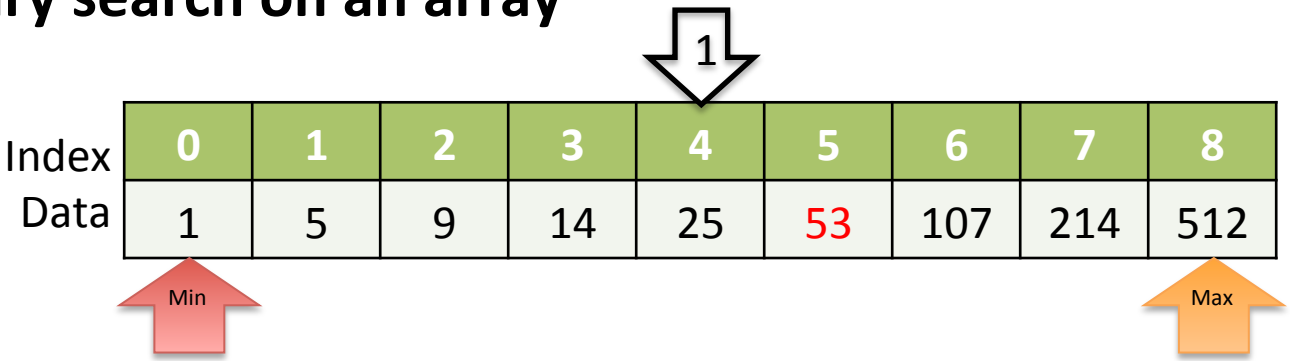
Pseudo code

Looking for target = 53

```
Set min = 0, max = n-1
While (min <= max) {
      idx = (min + max)/2
      If array[idx] == target
            return idx
      array[idx] > target
            max = idx-1
      else
            min = idx +1
}
```

At each iteration half of the indexes are eliminated

Binary search on an array



Pseudo code

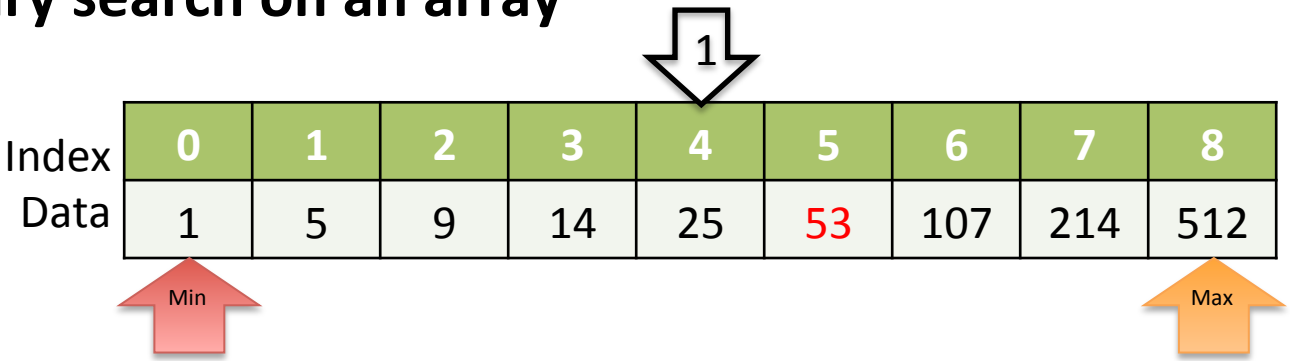
Looking for target = 53

```
Set min = 0, max = n-1
While (min <= max) {
      idx = (min + max)/2
      If array[idx] == target
            return idx
      array[idx] > target
            max = idx-1
      else
            min = idx +1
}
```

Target 53
Min = 0
Max = 8
Idx = (0+8)/2 = 4
Array[idx] = 25

At each iteration half of the indexes are eliminated

Binary search on an array



Pseudo code

Looking for target = 53

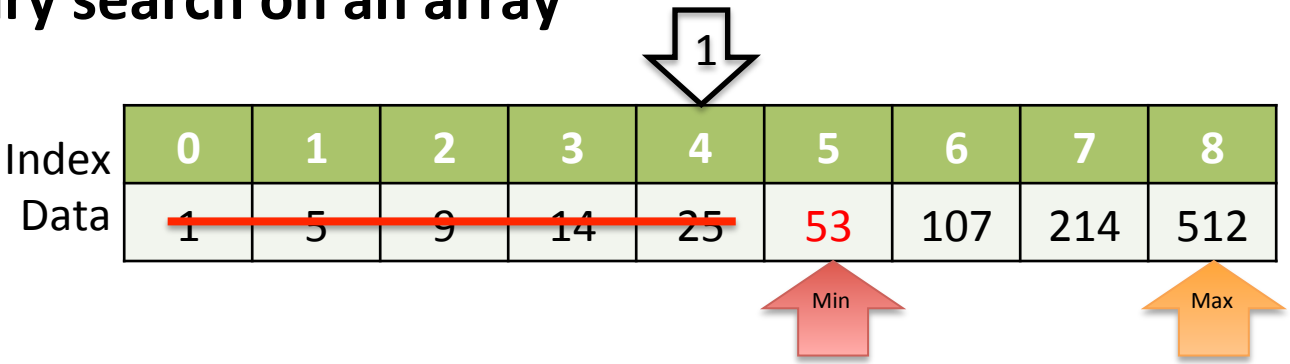
```
Set min = 0, max = n-1
While (min <= max) {
      idx = (min + max)/2
      If array[idx] == target
            return idx
      array[idx] > target
            max = idx-1
      else
            min = idx +1
}
```

Target 53
Min = 0
Max = 8
Idx = (0+8)/2 = 4
Array[idx] = 25

25 < 53
53 must be in right half
move up *min* to *idx* +1

At each iteration half of the indexes are eliminated

Binary search on an array



Pseudo code

Looking for target = 53

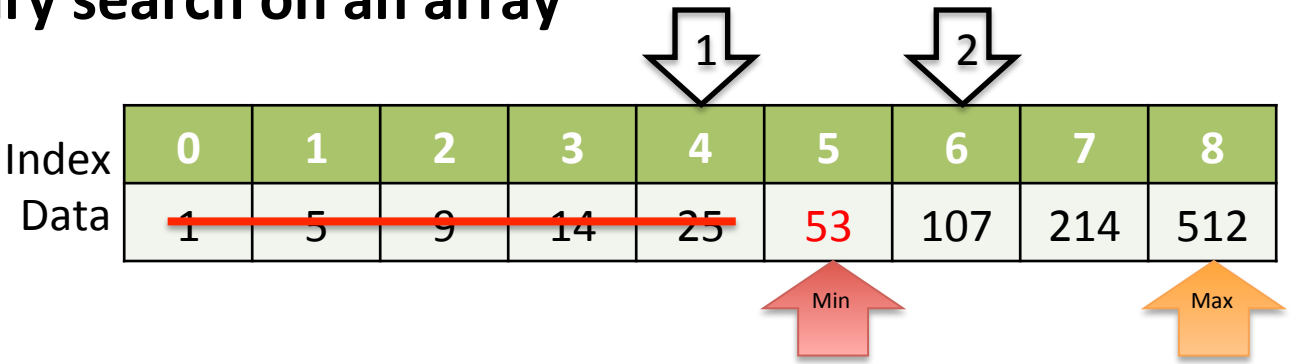
```
Set min = 0, max = n-1
While (min <= max) {
    idx = (min + max)/2
    If array[idx] == target
        return idx
    array[idx] > target
        max = idx-1
    else
        min = idx +1
}
```

Target 53
Min = 5
Max = 8
Idx = (5+8)/2 = 6
Array[idx] = 107

Eliminated half of the original items

At each iteration half of the indexes are eliminated

Binary search on an array



Pseudo code

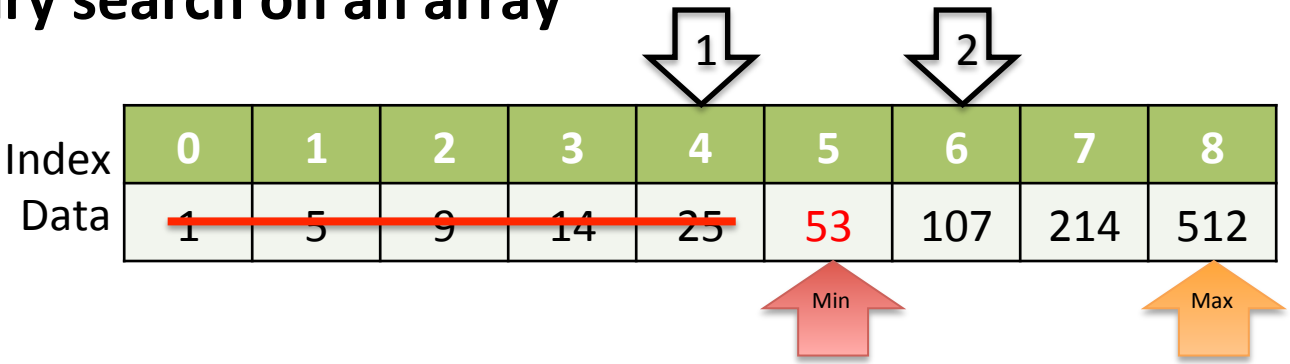
Looking for target = 53

```
Set min = 0, max = n-1
While (min <= max) {
    idx = (min + max)/2
    If array[idx] == target
        return idx
    array[idx] > target
        max = idx-1
    else
        min = idx +1
}
```

Target 53
Min = 5
Max = 8
Idx = (5+8)/2 = 6
Array[idx] = 107

At each iteration half of the indexes are eliminated

Binary search on an array



Pseudo code

Looking for target = 53

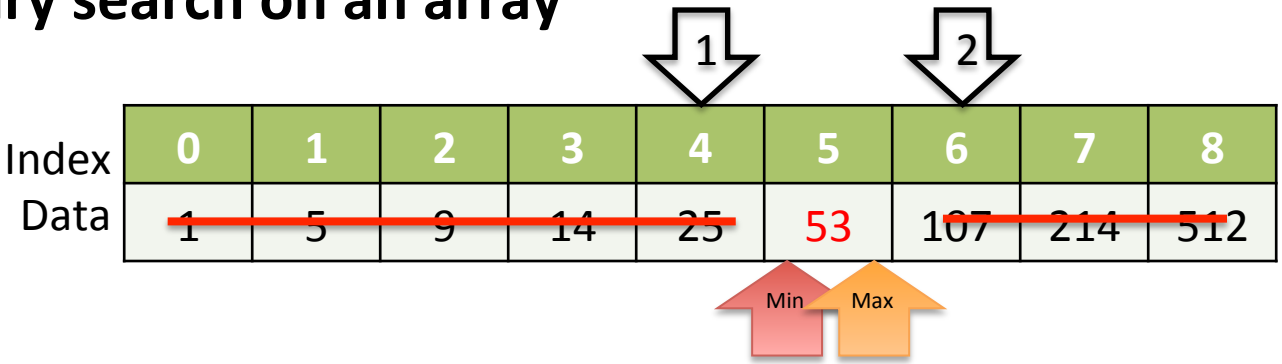
```
Set min = 0, max = n-1
While (min <= max) {
    idx = (min + max)/2
    If array[idx] == target
        return idx
    array[idx] > target
        max = idx-1
    else
        min = idx +1
}
```

Target 53
Min = 5
Max = 8
Idx = (5+8)/2 = 6
Array[idx] = 107

107 > 53
53 must be in left half
move *max* to *idx* -1

Binary search finds data generally faster than linear search

Binary search on an array



Pseudo code

Looking for target = 53

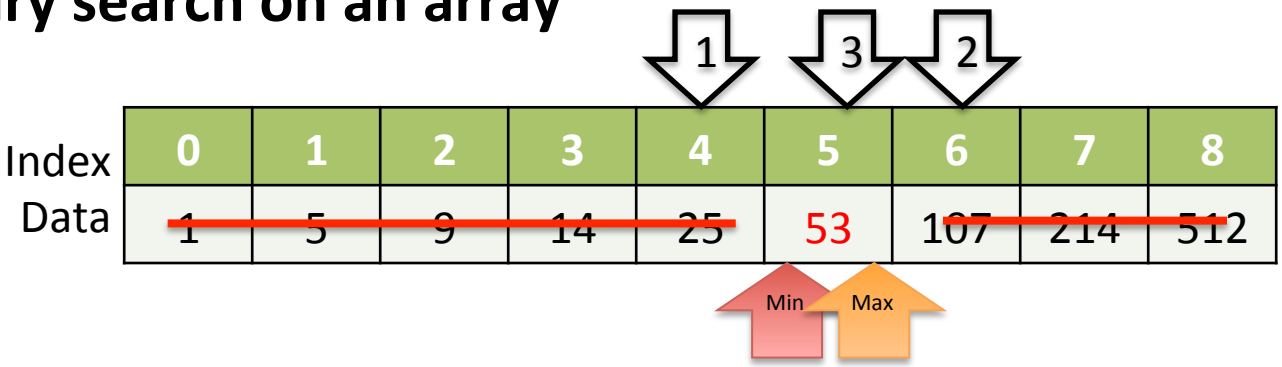
```
Set min = 0, max = n-1
While (min <= max) {
    idx = (min + max)/2
    If array[idx] == target
        return idx
    array[idx] > target
        max = idx-1
    else
        min = idx +1
}
```

Target 53
Min = 5
Max = 5
Idx = (5+5)/2 = 5
Array[idx] = 53

Eliminated half of the remaining items

Binary search finds data generally faster than linear search

Binary search on an array



Pseudo code

Looking for target = 53

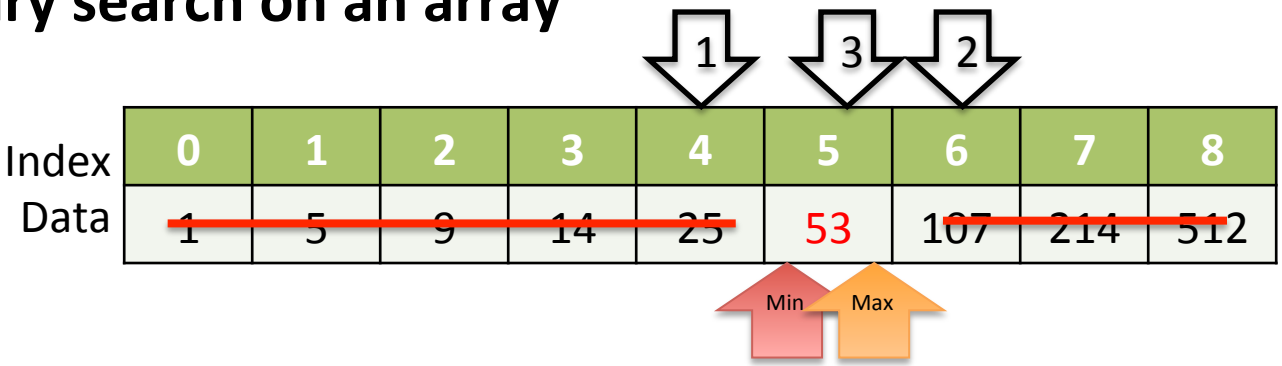
```
Set min = 0, max = n-1
While (min <= max) {
    idx = (min + max)/2
    If array[idx] == target
        return idx
    array[idx] > target
        max = idx-1
    else
        min = idx +1
}
```

Target 53
Min = 5
Max = 5
Idx = (5+5)/2 = 5
Array[idx] = 53

Found target

Binary search finds data generally faster than linear search

Binary search on an array



Pseudo code

Looking for target = 53
Set min = 0, max = n-1
While (min <= max) {
    idx = (min + max)/2
    If array[idx] == target
        return idx
    array[idx] > target
        max = idx-1
    else
        min = idx +1
}

Target 53
Min = 5
Max = 5
Idx = (5+5)/2 = 5
Array[idx] = 53

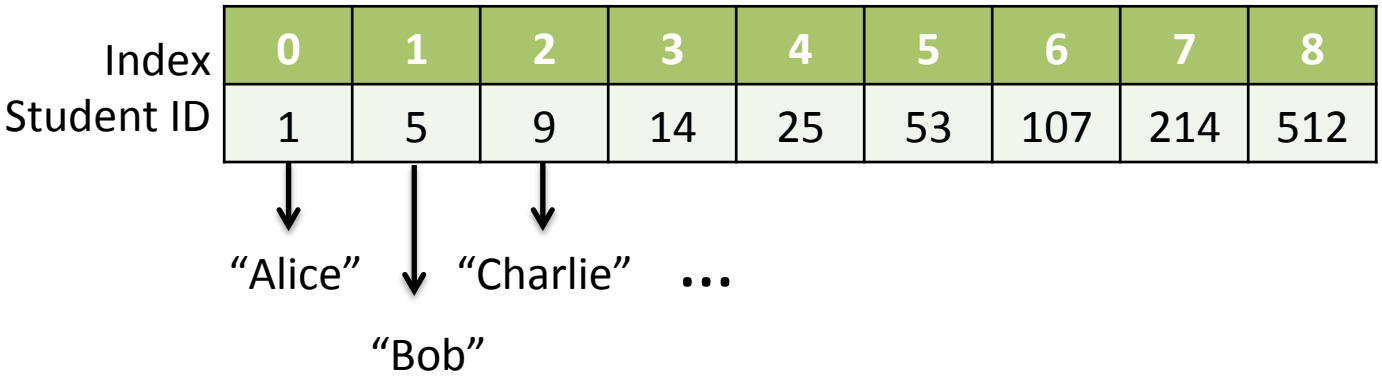
Found target

Binary vs. linear search

- Binary found item in 3 tries
- Linear search would have taken 6 tries
- On large data sets binary search can make a *huge* difference
- One million item collection takes 20 searches (one billion takes only 30)!

We can extend binary search to find a Key and return a Value

Key: Student ID, Value: Student name



Implications

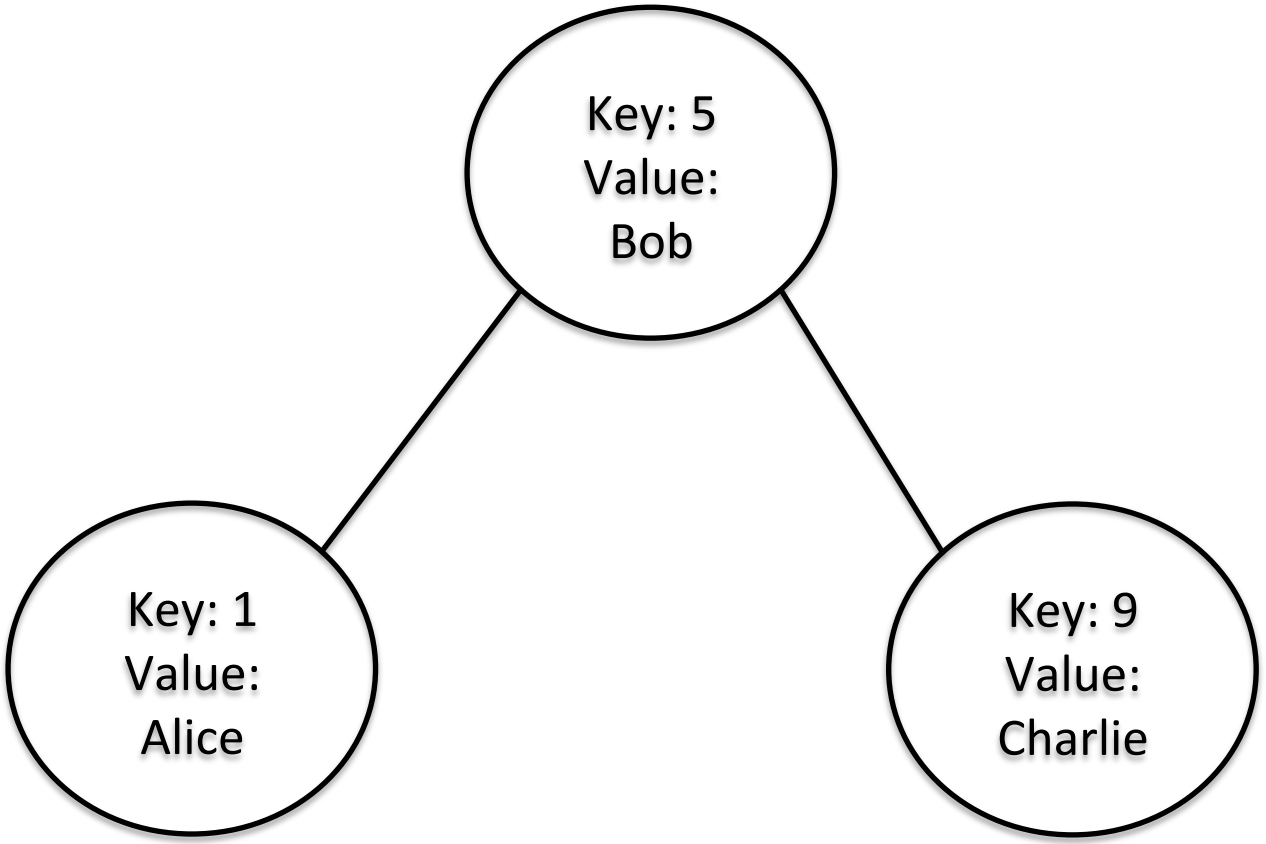
- Given a Student ID, can quickly find the student's name
- Each entry has a Key and a Value
- Value can be an object (e.g. String or student record object)
- Of course the keys must be sorted for this to work
- How do we do that?

Agenda

1. Binary search
2. Binary Search Trees (BST)
3. BST find analysis
4. Operations on BSTs
5. Implementation

BST nodes have a Key and a Value

Key: Student ID, Value: Student name



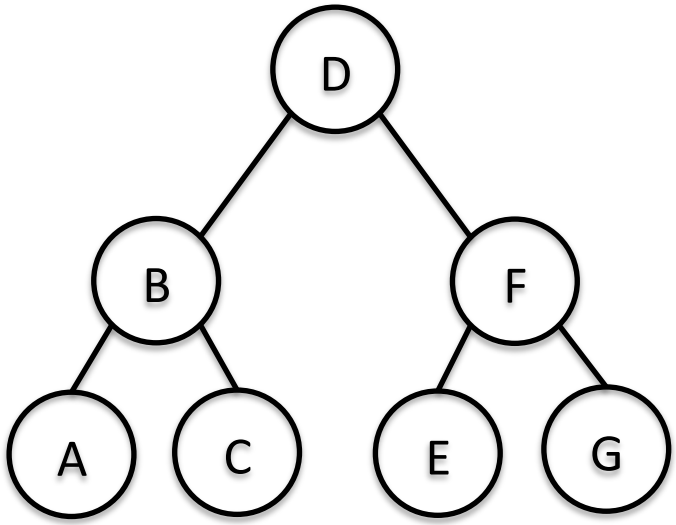
Can search for nodes by Key

Return Value if Key found

Will only show the Key in following slides

Binary Search Trees (BSTs) allow for binary search by keeping Keys sorted

Keys sorted in Binary Search Tree

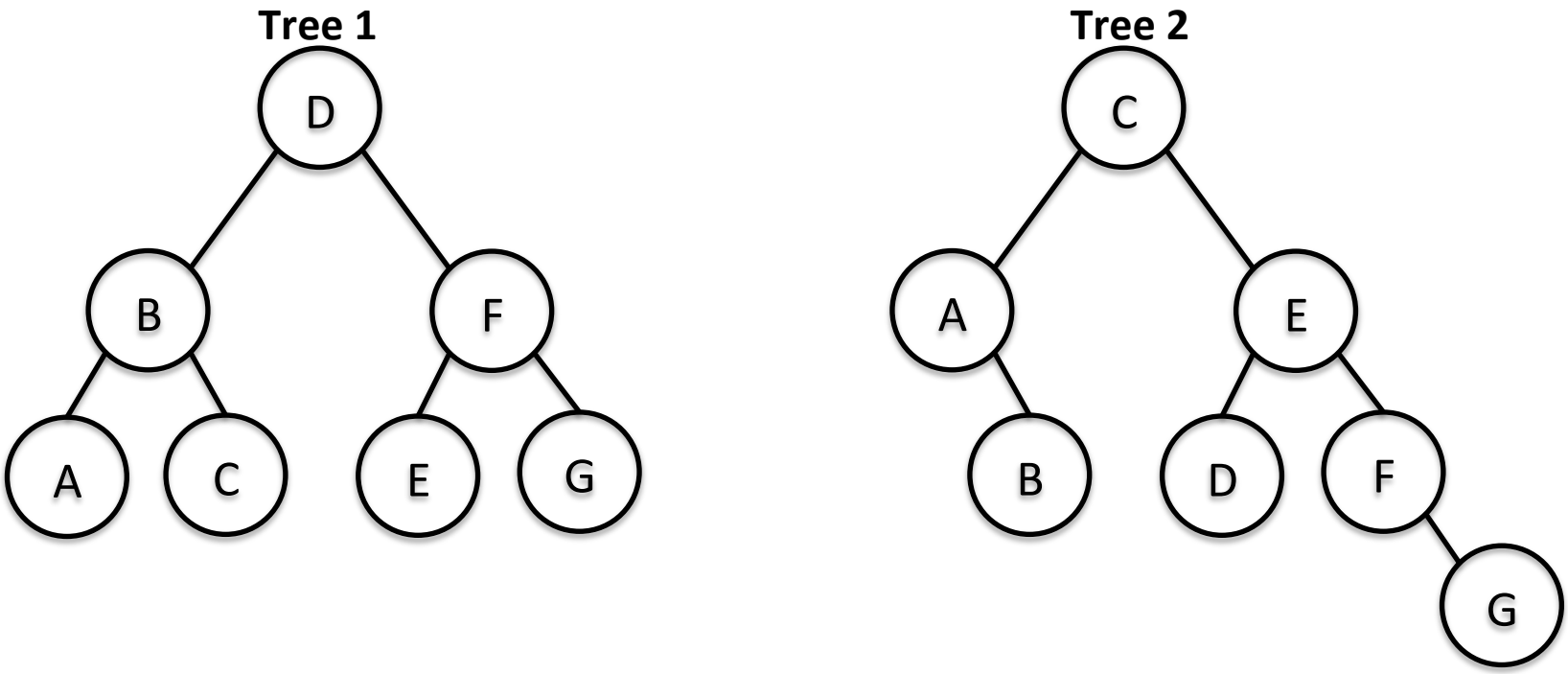


Binary Search Tree property

- Let `x` be a node in a binary search tree such that
  - `left.key < x.key`
  - `right.key > x.key`
- We will assume for now duplicate Keys are not allowed

BSTs with same keys could have different structures and still obey BST property

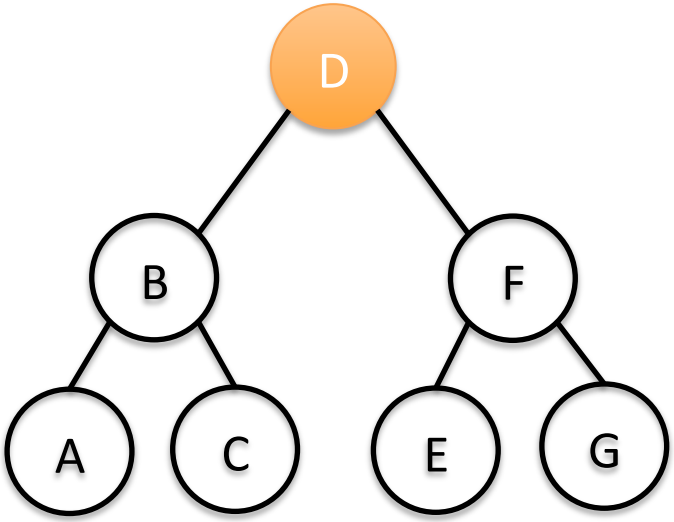
Two valid BSTs with same keys but different structure



For now we make no guarantee of balance (later in the term we will)

BSTs make searching fast and simple

Find Key

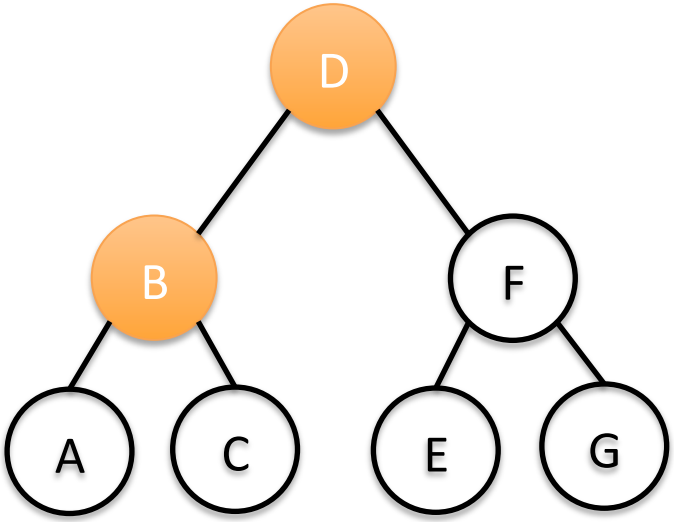


Find Key "C"

- Check root
- "D" > "C", so go left

BSTs make searching fast and simple

Find Key



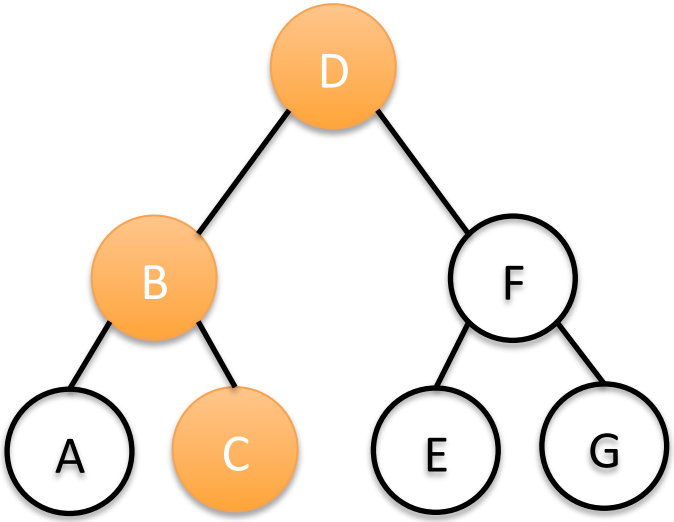
Find Key "C"

- Check root
- "D" > "C", so go left

- Check "B"
- "B" < "C", so go right

BSTs make searching fast and simple

Find Key



Find Key "C"

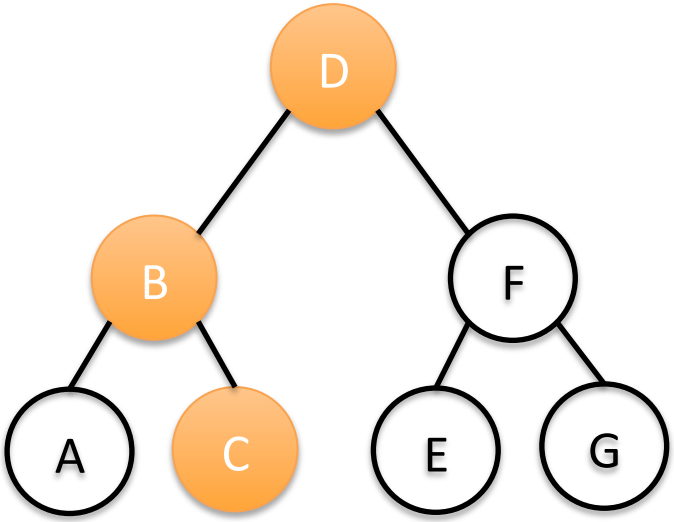
- Check root
- "D" > "C", so go left

- Check "B"
- "B" < "C", so go right

- Check "C"
- Yahtzee! Found it

BSTs make searching fast and simple

Find Key



Find Key "C"

- Check root
- "D" > "C", so go left

- Check "B"
- "B" < "C", so go right

- Check "C"
- Yahtzee! Found it

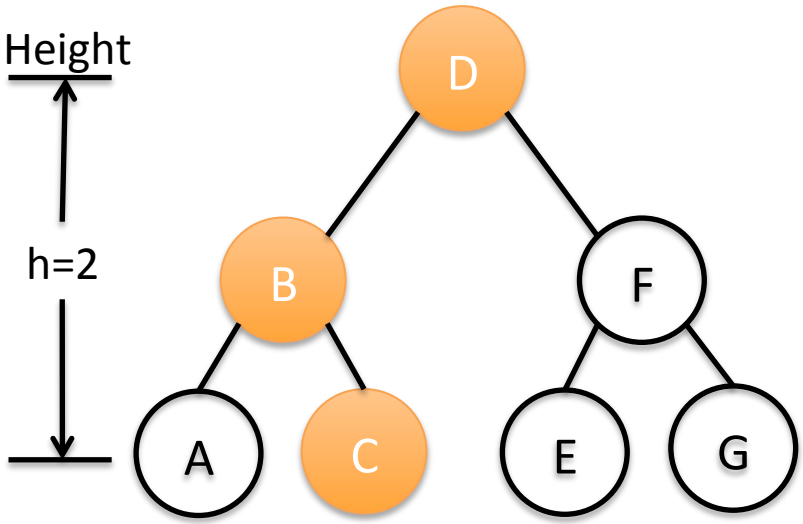
- Would know by now if key not in BST because we hit a leaf

Agenda

1. Binary search
2. Binary Search Trees (BST)
3. BST find analysis
4. Operations on BSTs
5. Implementation

BST takes at most *height+1* checks to find Key or determine the Key is not in the tree

Find Key "C"

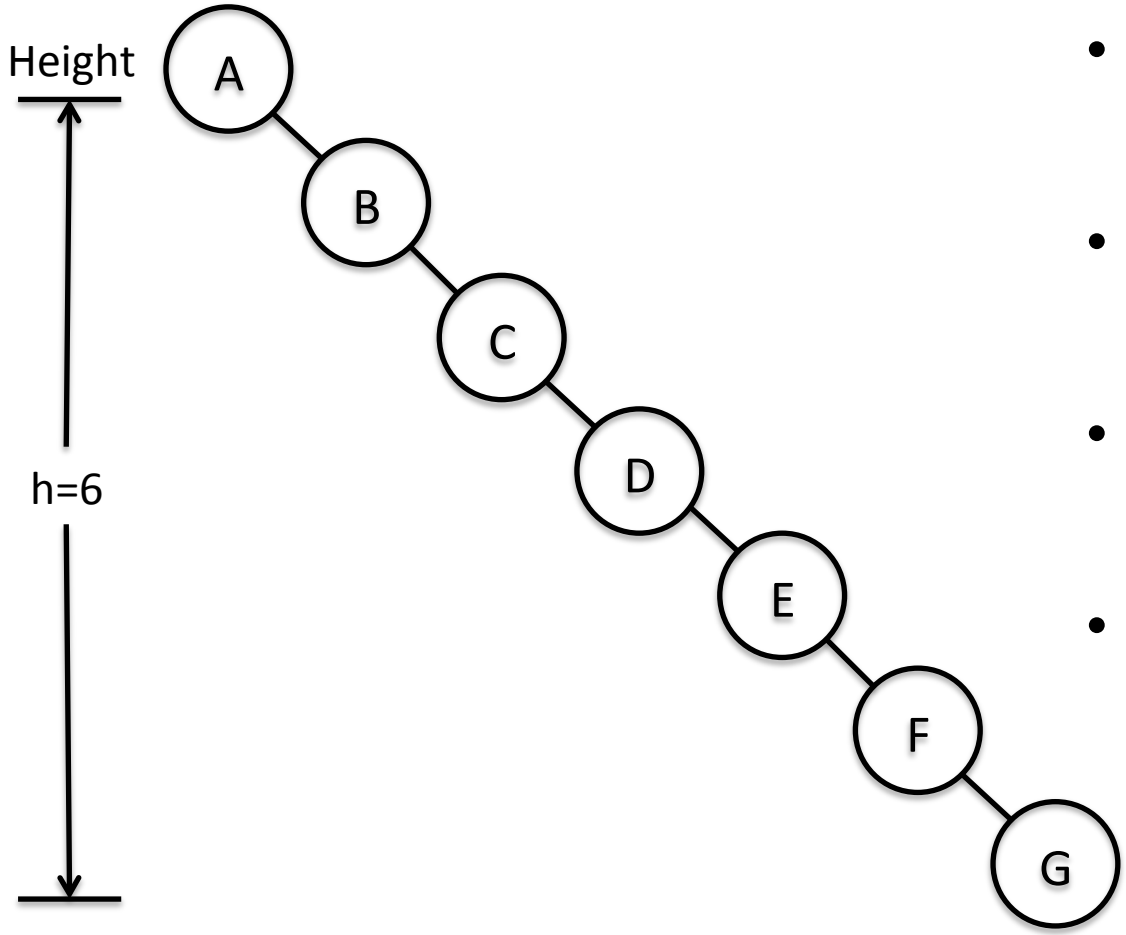


Search process

- Height *h = 2* (count number of edges on longest path to leaf)
- Can take no more than *h+1* checks, O(h)
- Can we say anything more specific about search time? O(log n)? Careful, it's a trap!

BSTs do not have to be balanced! Can not make tight bound assumptions! (yet)

Find Key "G"



Search process

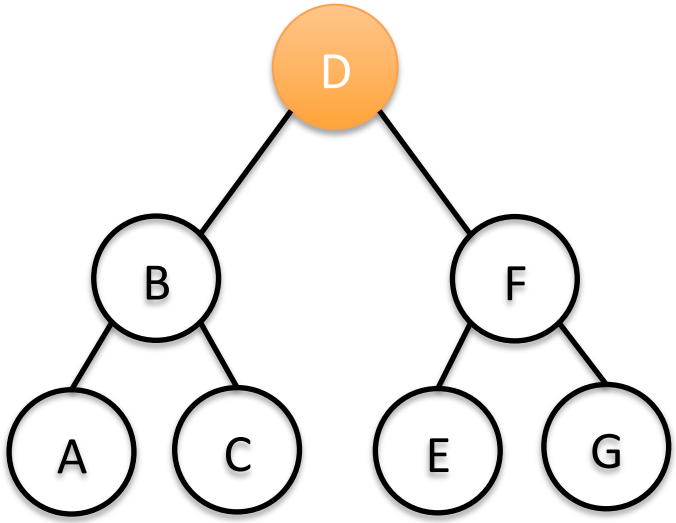
- Same data as last slide but still valid BST
- Height *h = 6* (count number of edges to leaf)
- Can take no more than *h+1* checks, O(h)
- An arrangement like this sometimes called a "vine"

Agenda

1. Binary search
2. Binary Search Trees (BST)
3. BST find analysis
4. Operations on BSTs
5. Implementation

Inserting a new Key is easy (compared with sorted array)

Inserting new node with Key H

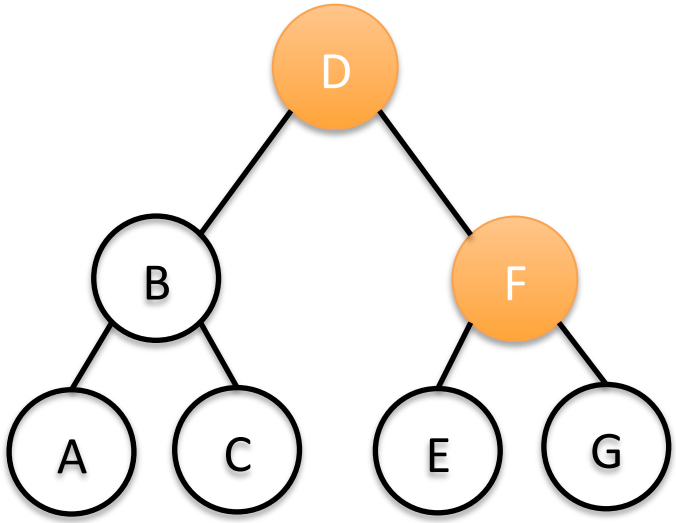


Comments

- Search for Key (H)
  - If found, replace Value
  - If hit leaf, add new node as left or right child of leaf

Inserting a new Key is easy (compared with sorted array)

Inserting new node with Key H



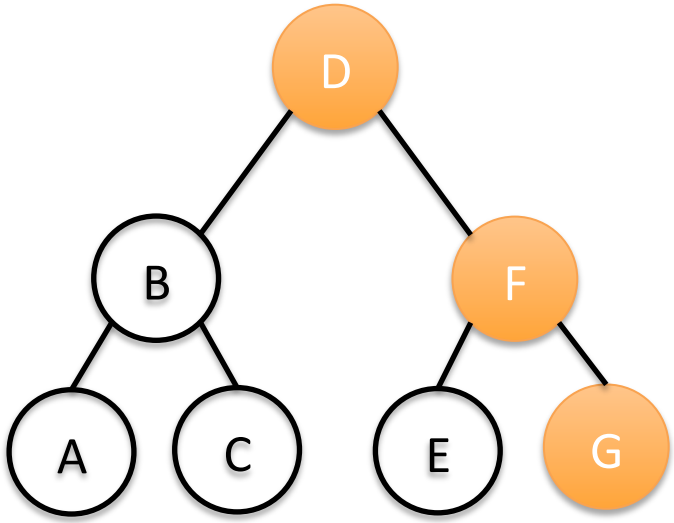
Searching for H

Comments

- Search for Key (H)
  - If found, replace Value
  - If hit leaf, add new node as left or right child of leaf

Inserting a new Key is easy (compared with sorted array)

Inserting new node with Key H



Searching for H

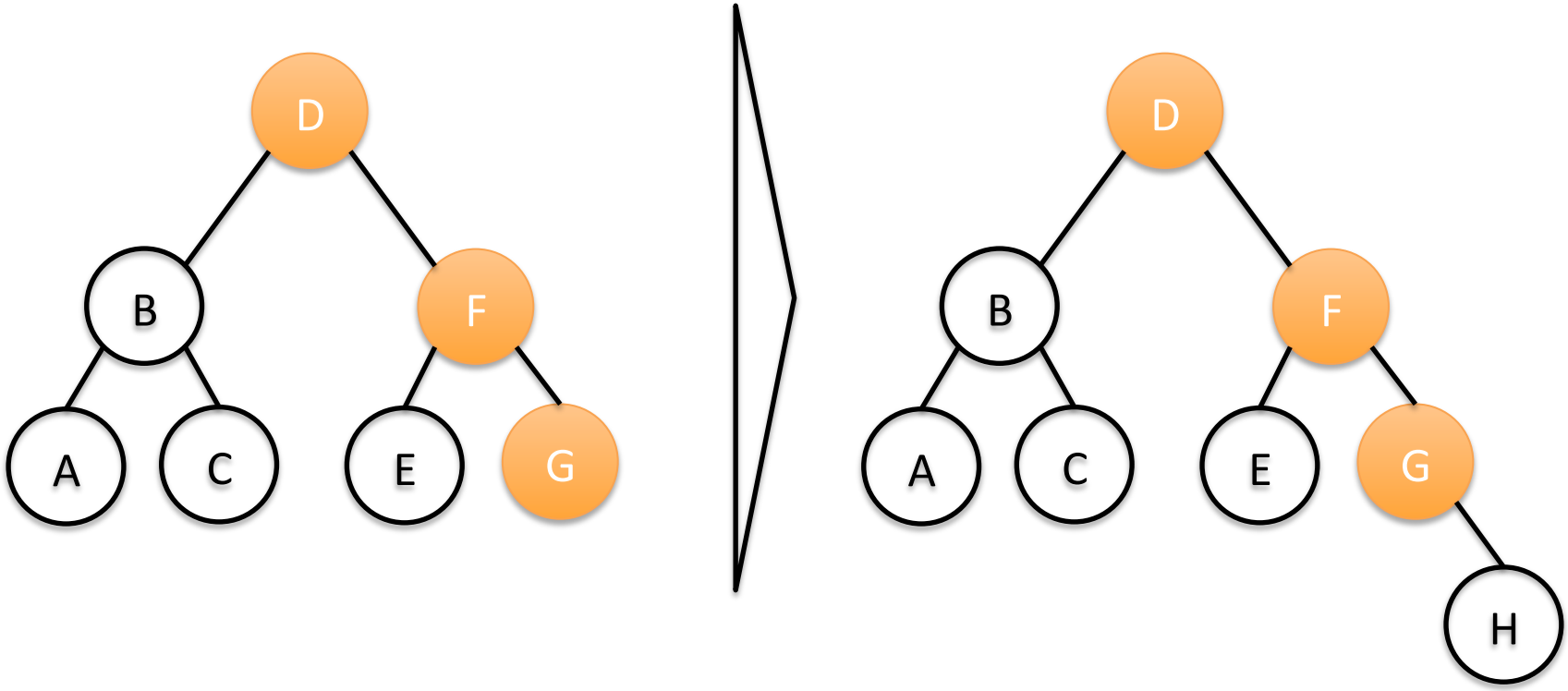
G is a leaf
H is not in the Tree
Add new node to G
Choose left or right child based on Key of new node (H here)

Comments

- Search for Key (H)
  - If found, replace Value
  - If hit leaf, add new node as left or right child of leaf

Inserting a new Key is easy (compared with sorted array)

Inserting new node with Key H

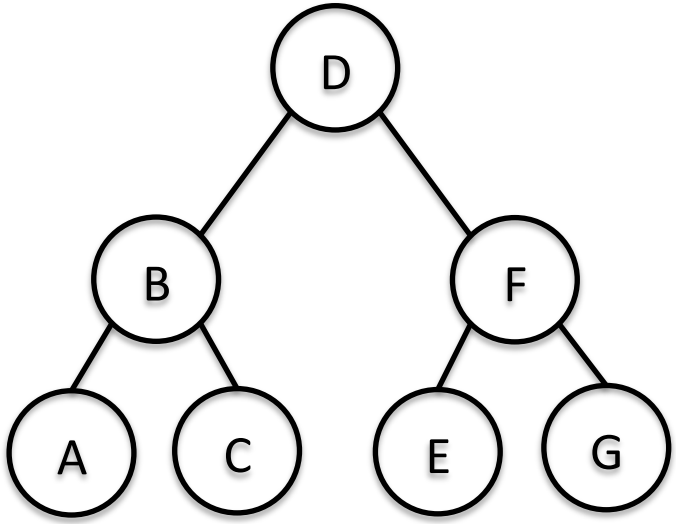


Comments

- Search for Key (H)
  - If found, replace Value
  - If hit leaf, add new node as left or right child of leaf

Deletion is trickier, need to consider children, but no children is easy

Deleting node A (no children)

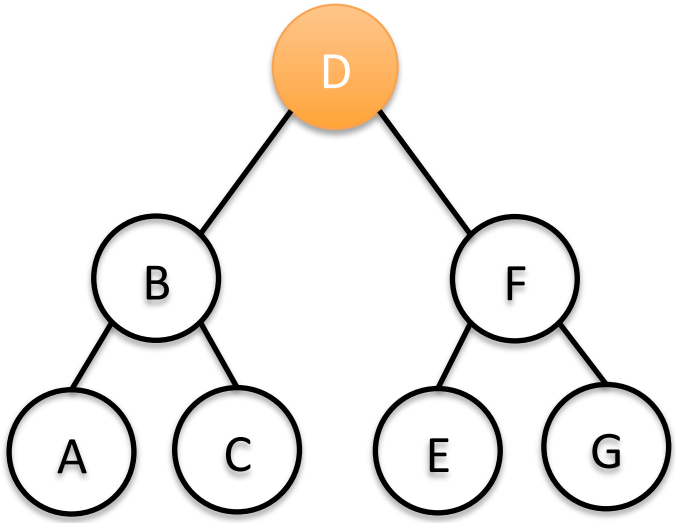


Comments

- Search for parent of A
  - If found and A has no children, set appropriate left or right to null on parent

Deletion is trickier, need to consider children, but no children is easy

Deleting node A (no children)



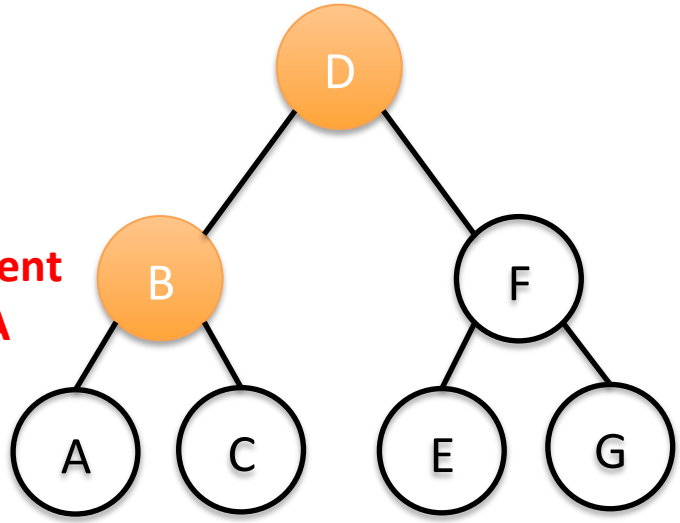
Search for parent of A

Comments

- Search for parent of A
  - If found and A has no children, set appropriate left or right to null on parent

Deletion is trickier, need to consider children, but no children is easy

Deleting node A (no children)



B is parent of A

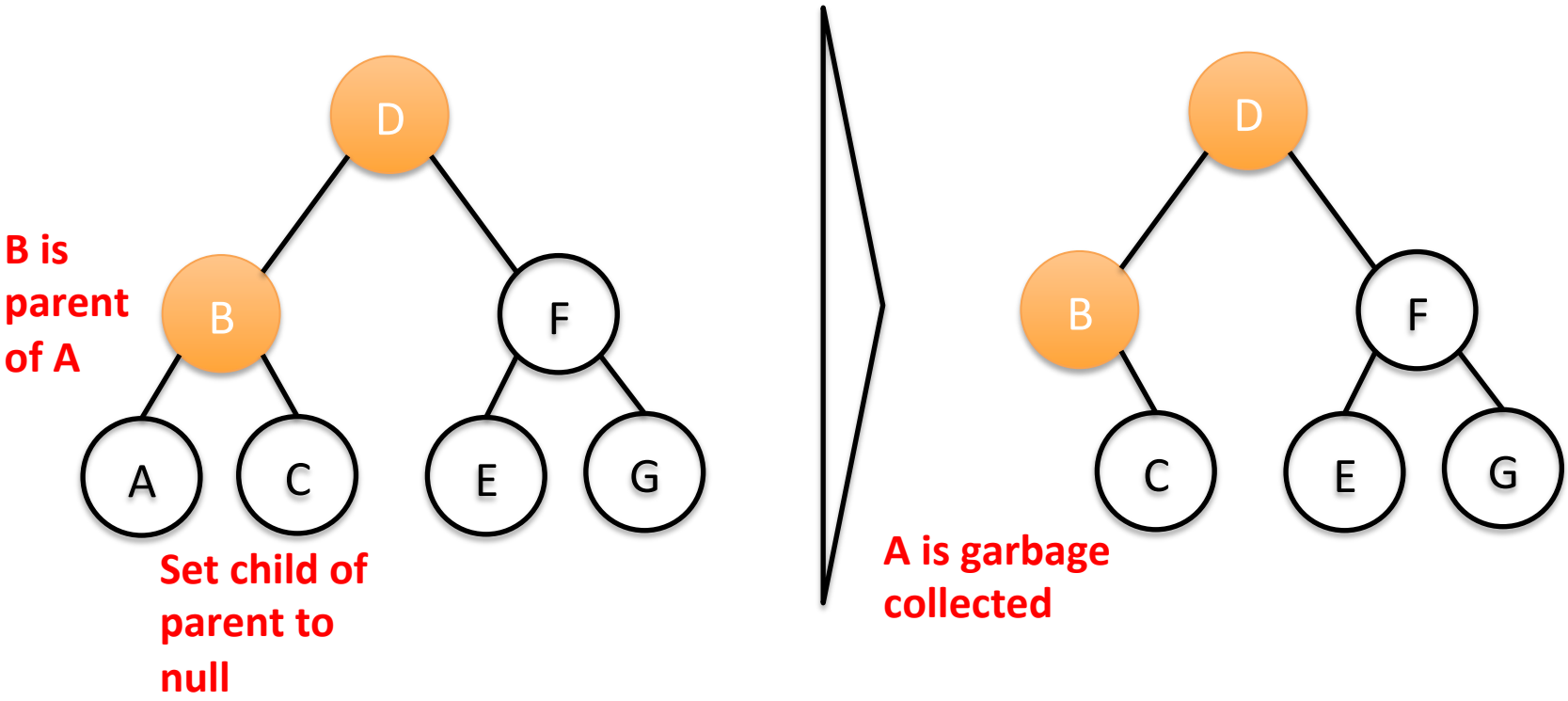
Search for parent of A

Comments

- Search for parent of A
  - If found and A has no children, set appropriate left or right to null on parent

Deletion is trickier, need to consider children, but no children is easy

Deleting node A (no children)

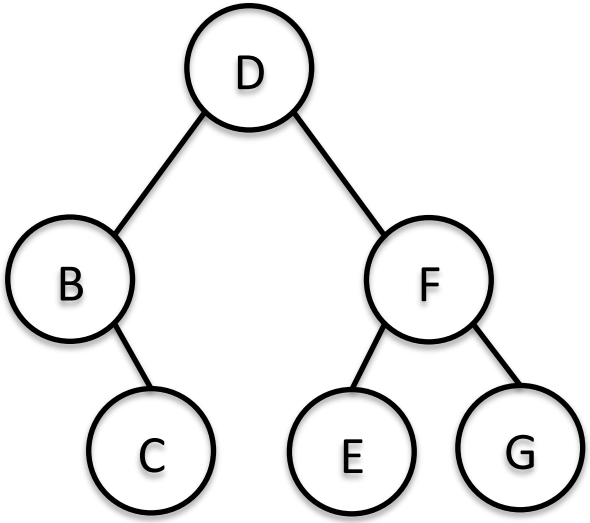


Comments

- Search for parent of A
  - If found and A has no children, set appropriate left or right to null on parent

Deleting with one child is not difficult

Deleting node B (1 child)

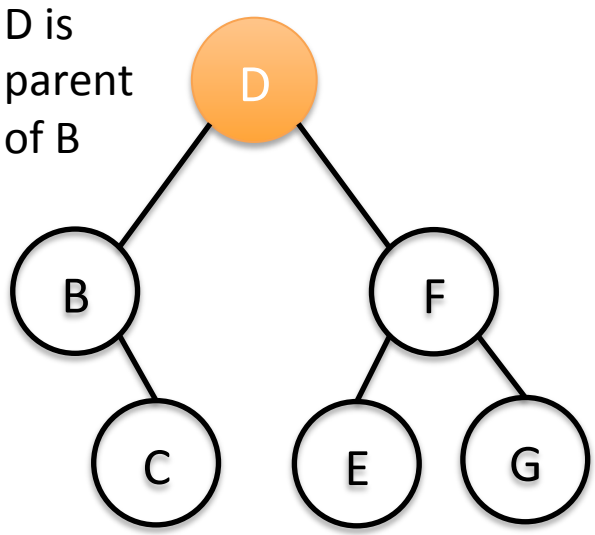


Comments

- Search for parent of B
  - If found and B has 1 child, set appropriate left or right on parent to B's only child

Deleting with one child is not difficult

Deleting node B (1 child)

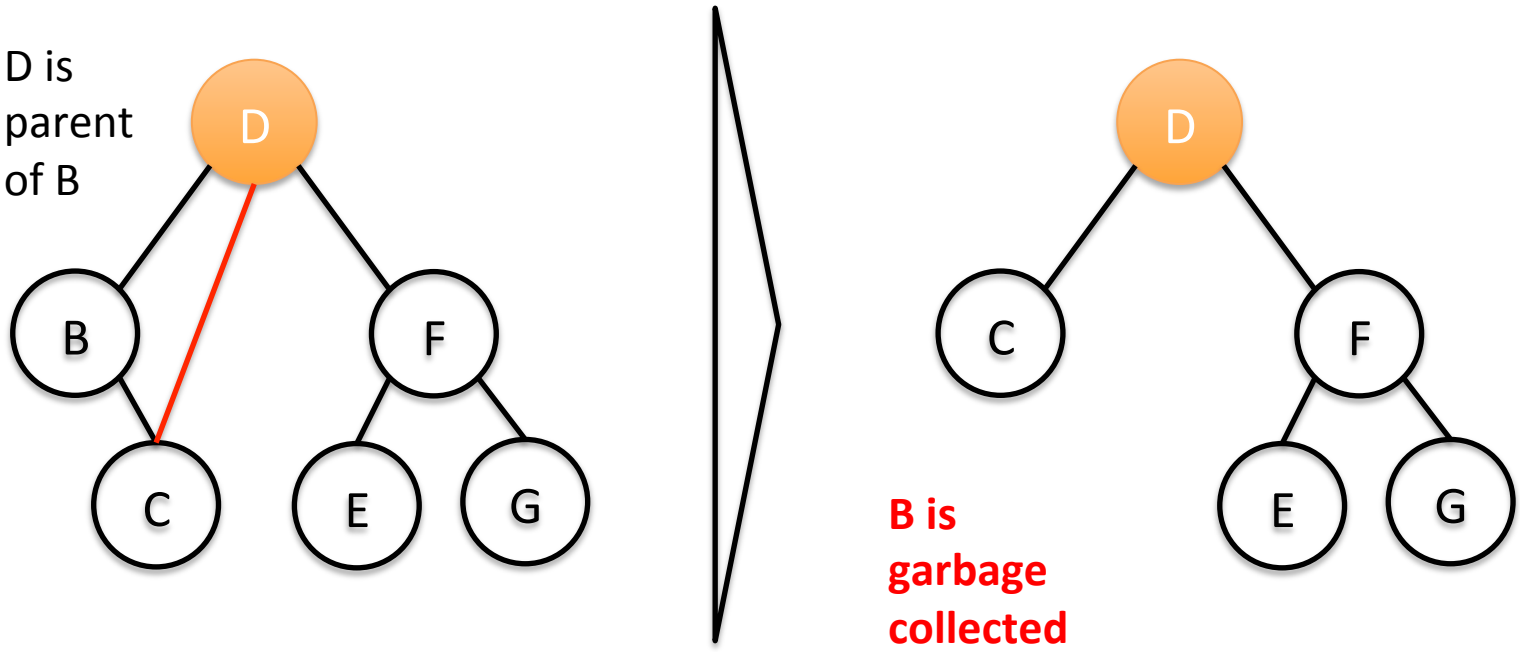


Comments

- Search for parent of B
  - If found and B has 1 child, set appropriate left or right on parent to B's only child

Deleting with one child is not difficult

Deleting node B (1 child)

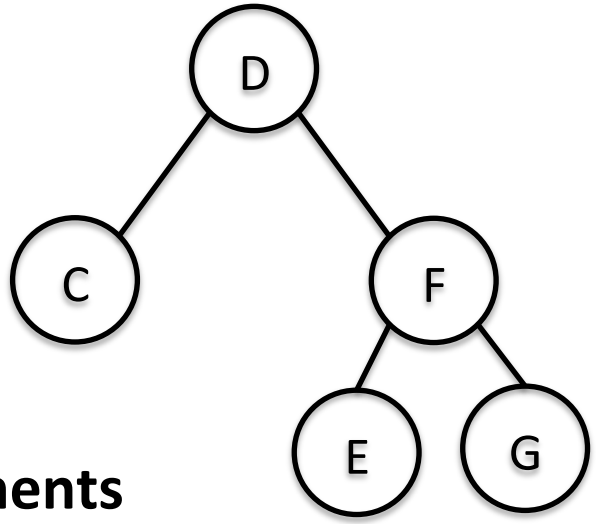


Comments

- Search for parent of B
  - If found and B has 1 child, set appropriate left or right on parent to B's only child

Deleting node with 2 children requires finding the node's "successor"

Deleting node F (2 children)

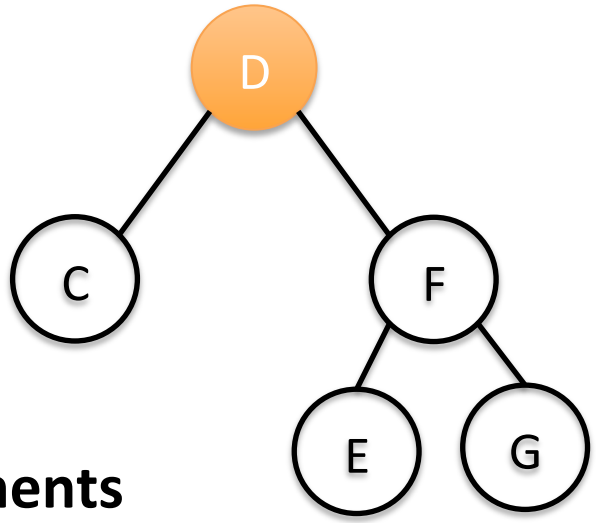


Comments

- Search for F
- If found and F has 2 children, find successor (smallest on right)
- Successor will be greater than E and less than or equal to G
- May have to traverse down right child's left descendants
- Delete successor, but save successor's Key and Value
- Replace F with Key and Value of successor

Deleting node with 2 children requires finding the node's "successor"

Deleting node F (2 children)

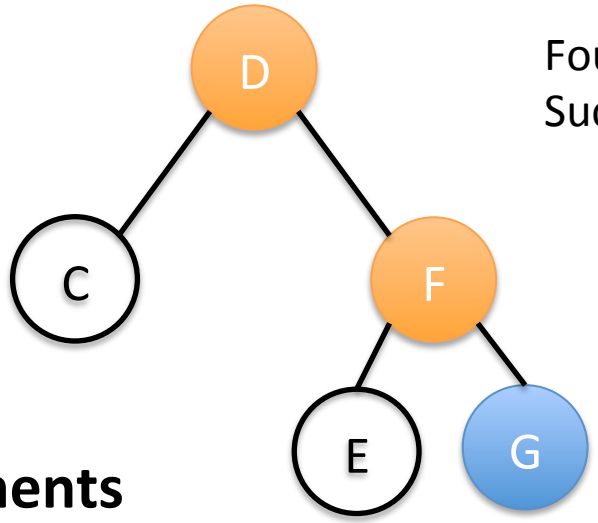


Comments

- Search for F
- If found and F has 2 children, find successor (smallest on right)
- Successor will be greater than E and less than or equal to G
- May have to traverse down right child's left descendants
- Delete successor, but save successor's Key and Value
- Replace F with Key and Value of successor

Deleting node with 2 children requires finding the node's "successor"

Deleting node F (2 children)



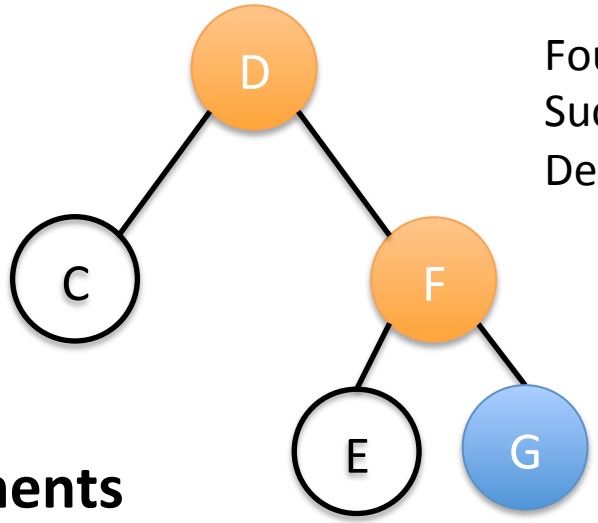
Found F
Successor is smallest on right (G here)

Comments

- Search for F
- If found and F has 2 children, find successor (smallest on right)
- Successor will be greater than E and less than or equal to G
- May have to traverse down right child's left descendants
- Delete successor, but save successor's Key and Value
- Replace F with Key and Value of successor

Deleting node with 2 children requires finding the node's "successor"

Deleting node F (2 children)



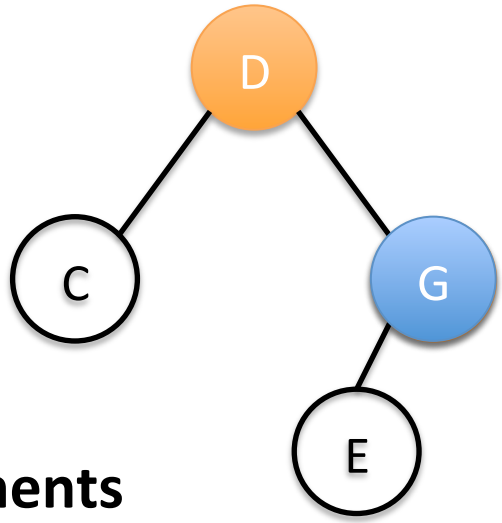
Found F
Successor is smallest on right (G here)
Delete successor

Comments

- Search for F
- If found and F has 2 children, find successor (smallest on right)
- Successor will be greater than E and less than or equal to G
- May have to traverse down right child's left descendants
- Delete successor, but save successor's Key and Value
- Replace F with Key and Value of successor

Deleting node with 2 children requires finding the node's "successor"

Deleting node F (2 children)



Found F
Successor is smallest on right (G here)
Delete successor
Replace F Key and Value with G Key and Value

Comments

- Search for F
- If found and F has 2 children, find successor (smallest on right)
- Successor will be greater than E and less than or equal to G
- May have to traverse down right child's left descendants
- Delete successor, but save successor's Key and Value
- Replace F with Key and Value of successor

Agenda

1. Binary search
2. Binary Search Trees (BST)
3. BST find analysis
4. Operations on BSTs
5. Implementation

Binary Search Tree nodes each take a Key and Value, also have left and right children

BST.java

```
10 public class BST<K extends Comparable<K>,V> {
11     private K key;
12     private V value;
13     private BST<K,V> left, right;
14
15     /**
16      * Constructs leaf node -- left and right are null
17      */
18     public BST(K key, V value) {
19         this.key = key; this.value = value;
20     }
21
22     /**
23      * Constructs inner node
24      */
25     public BST(K key, V value, BST<K,V> left, BST<K,V> right) {
26         this.key = key; this.value = value;
27         this.left = left; this.right = right;
28     }
```

- Key (K) and Value (V) are generics (can be any object type)
- Use wrapper for primitive types (e.g., Integer for int)

- Example: Key=Student ID as String, Value=Student object with name, year, list of classes taken
- Has left and right child like Binary Tree from last class

BST Keys extend Comparable so we can evaluate generic Keys

BST.java

```
10 public class BST<K extends Comparable<K>,V> {
11     private K key;
12     private V value;
13     private BST<K,V> left, right;
14
15     /**
16      * Constructs leaf node
17      */
18     public BST(K key, V value) {
19         this.key = key; this.value = value;
20     }
21
22     /**
23      * Constructs inner node
24      */
25     public BST(K key, V value, BST<K,V> left, BST<K,V> right) {
26         this.key = key; this.value = value;
27         this.left = left; this.right = right;
28     }
```

- Keys are generic, can be any type
- To maintain BST property, need to determine if Key < or > other Key
- Key extends Comparable for this purpose
- Comparable requires class used as Key to implement *compareTo()* method
- Can't use class as Key without it
- compareTo() already implemented for built in classes such as Integer or String
- Must implement in our own classes if we use them as Keys (e.g., if Blobs were Keys, how is one Blob <, =, > another?)
- Blob class would have to tell us (maybe the highest one, or the leftmost one, …)

Need to implement *compareTo()* if using custom class as Key

BlobWithCompareTo.java

If you use your own class as a Key, then must implement *compareTo()*

Can't use your class as Key in BST.java if you do not

- **In Class definition add "implements Comparable" so Java knows class follows interface (not shown)**

```
/**
 * Compare this blob with another blob
 * @param compareBlob blob to compare to this blob
 * @return    0 if same,
 *            1 if this blob is higher up than compareBlob,
 *           -1 otherwise
 */
public int compareTo(BlobWithCompareTo compareBlob) {
    if (this.y < compareBlob.getY())
        return 1; //this Blob is higher up
    else if (this.y > compareBlob.getY())
        return -1; //this Blob is lower
    else return 0; //at same height
}
```

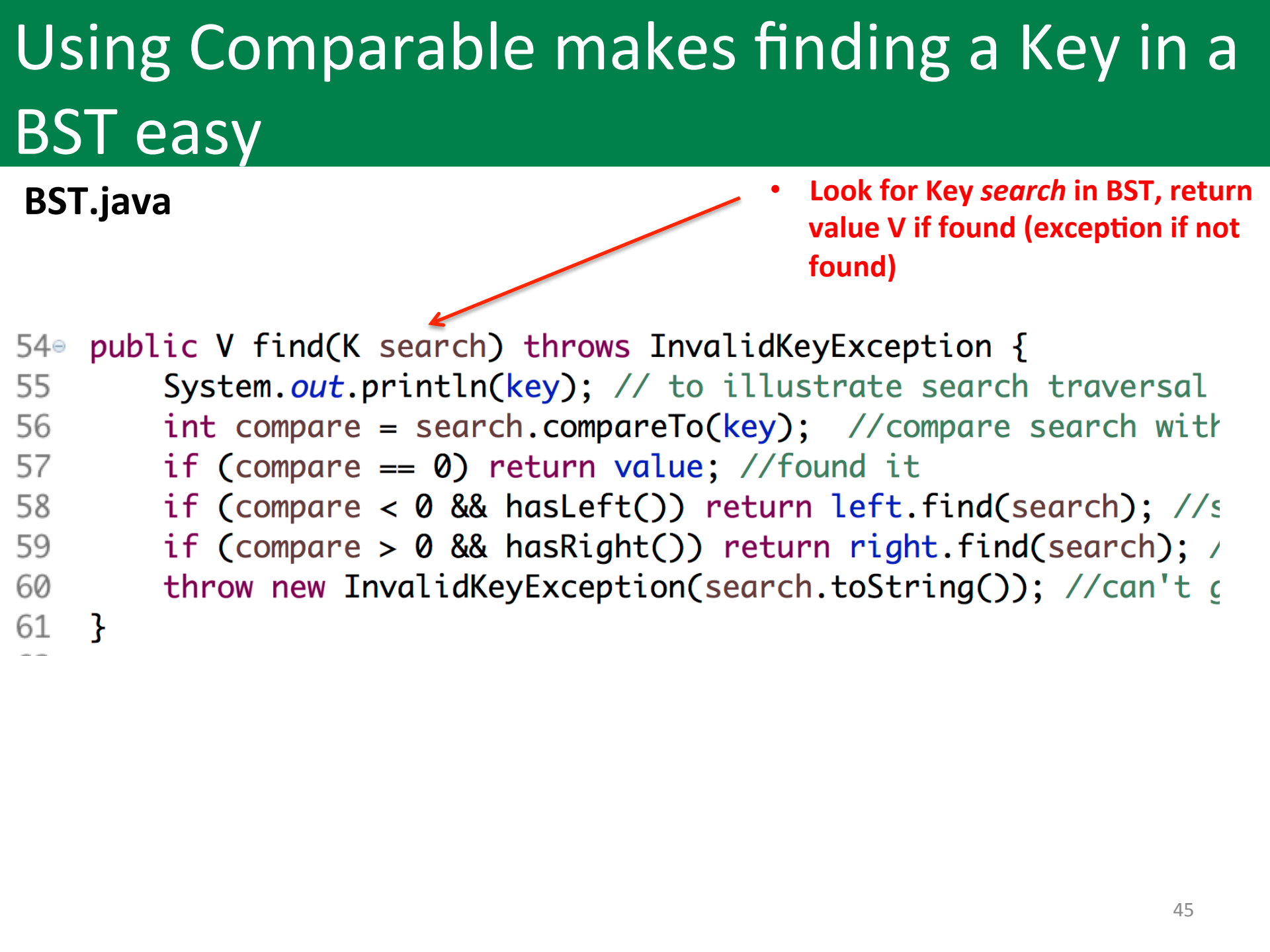
- **Compare this Blob with another Blob using whatever metric you decide makes one bigger**
- **Return a positive number if this Blob > compared Blob**
- **Return negative number if this Blob < compared Blob**
- **Return 0 if equal**

- **Return values not limited to just -1, 0 or 1**
- **Only need to be negative, positive or zero integers**

Using Comparable makes finding a Key in a BST easy

BST.java

- **Look for Key *search* in BST, return value V if found (exception if not found)**



```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

Using Comparable makes finding a Key in a BST easy

BST.java

- **Look for Key *search* in BST, return value V if found (exception if not found)**

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

- **Use *compareTo()* to evaluate *search* Key with this node's Key**
- **Return this node's Value if found**

Using Comparable makes finding a Key in a BST easy

BST.java

- **Look for Key *search* in BST, return value V if found (exception if not found)**

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

- **Use *compareTo()* to evaluate *search* Key with this node's Key**
- **Return this node's Value if found**

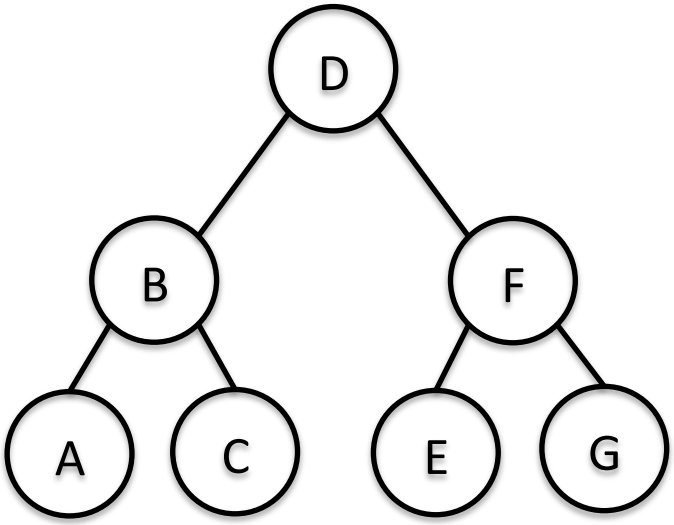
- **Traverse left or right based on Key comparison**
- **Throw exception if make it all the way to a leaf and haven't found Key**

Using Comparable makes finding a Key in a BST easy

BST.java

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

t= Node “D”

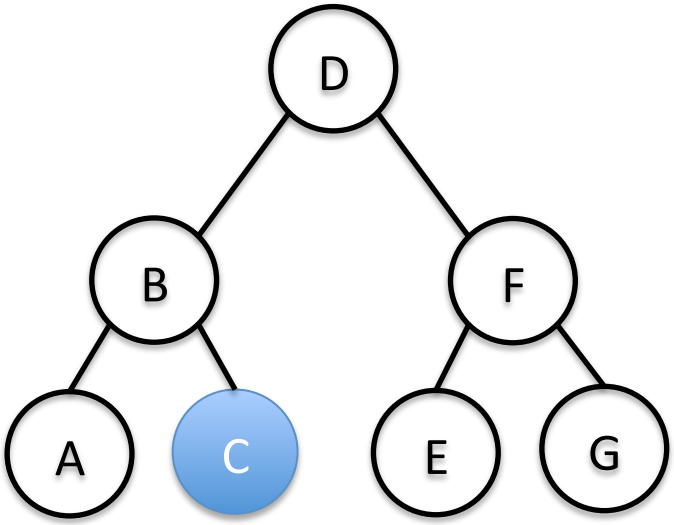


Using Comparable makes finding a Key in a BST easy

BST.java

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

V value = t.find("C")



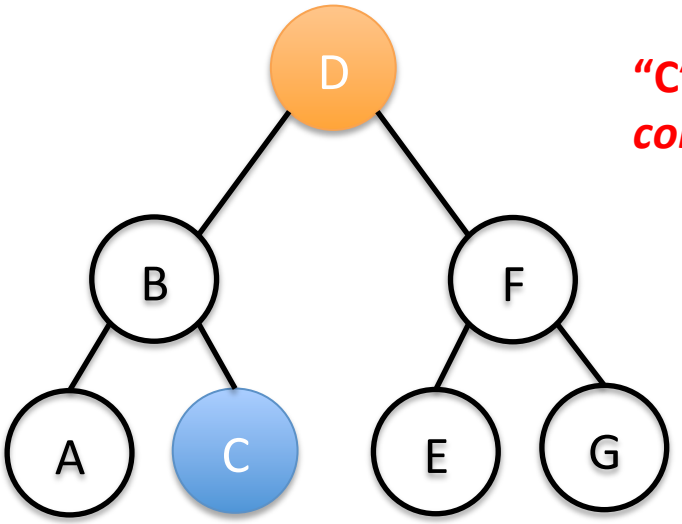
Using Comparable makes finding a Key in a BST easy

BST.java

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

D

V value = t.find("C")



"C" < "D"

***compare* = -1**

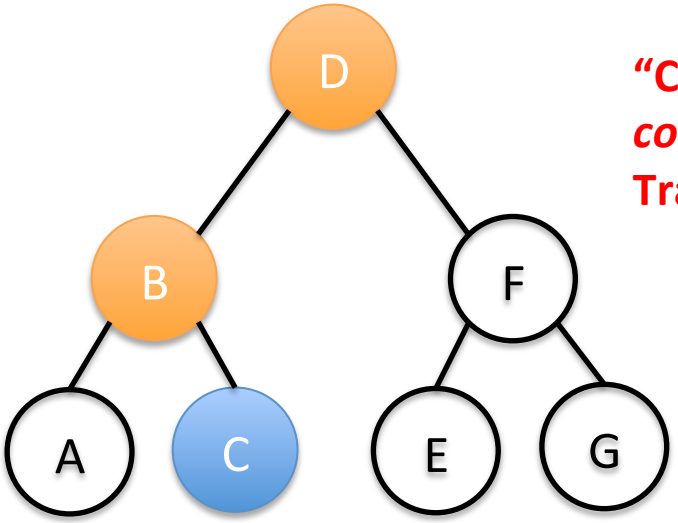
Using Comparable makes finding a Key in a BST easy

BST.java

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

D

V value = t.find("C")



"C" < "D"

***compare* = -1**

Traverse left

Using Comparable makes finding a Key in a BST easy

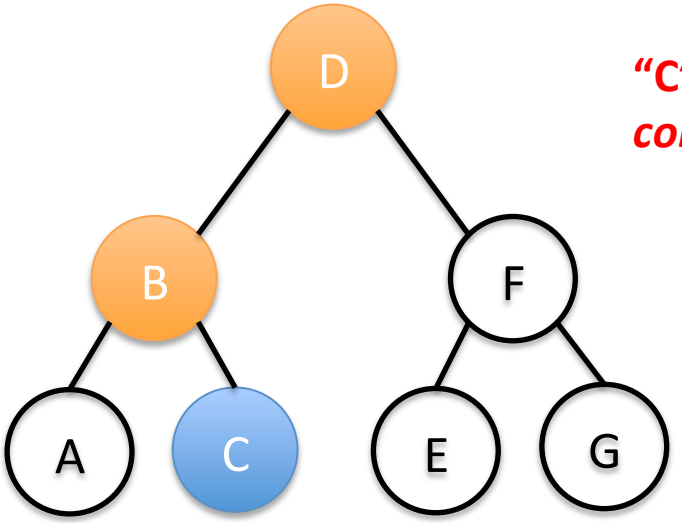
BST.java

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

B

D

V value = t.find("C")



"C" > "B"

***compare* = 1**

Using Comparable makes finding a Key in a BST easy

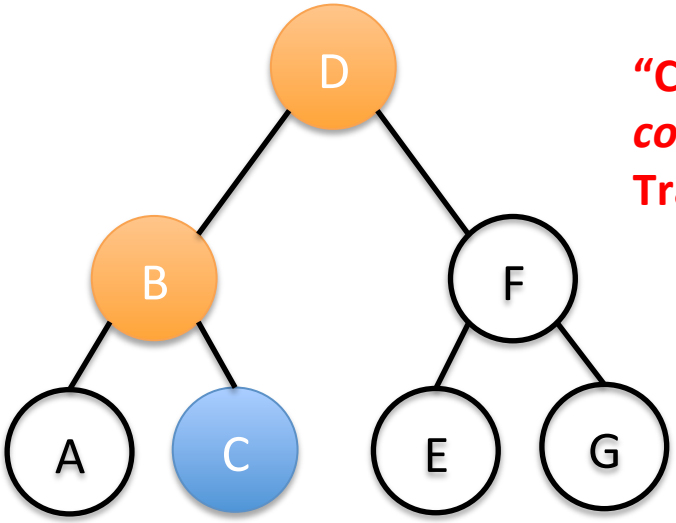
BST.java

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

D

B

V value = t.find("C")



"C" > "B"

***compare* = 1**

Traverse right

Using Comparable makes finding a Key in a BST easy

BST.java

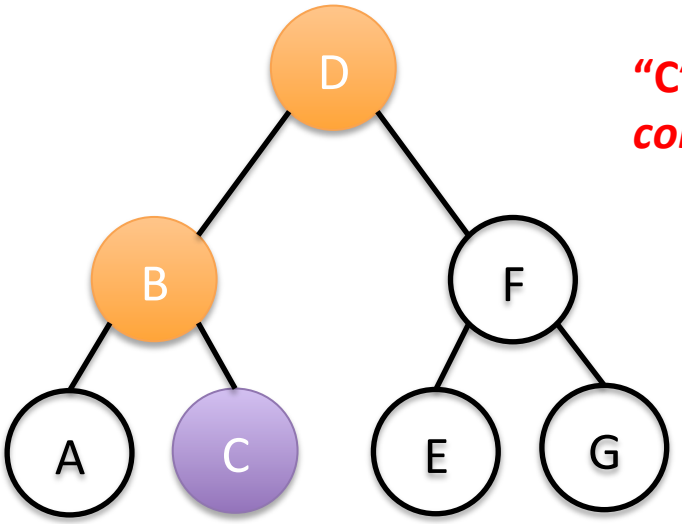
```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

C

D

B

V value = t.find("C")



"C" = "C"

***compare* = 0**

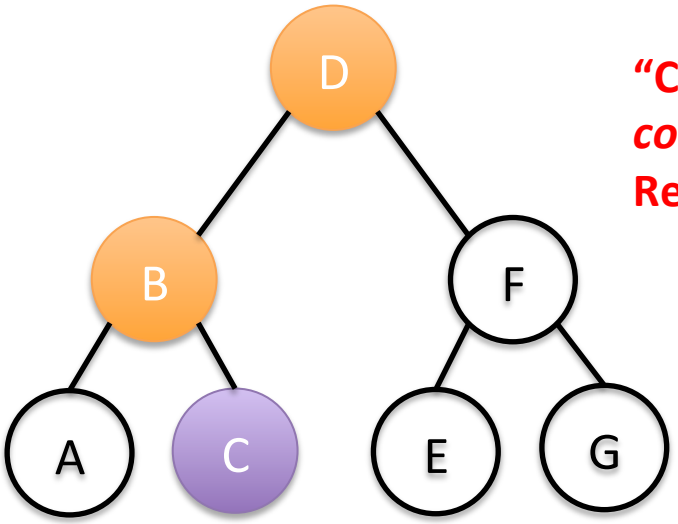
Using Comparable makes finding a Key in a BST easy

BST.java

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

C
D
B

V value = t.find("C")



"C" = "C"

***compare* = 0**

Return Value of node "C"

Using Comparable makes finding a Key in a BST easy

BST.java

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

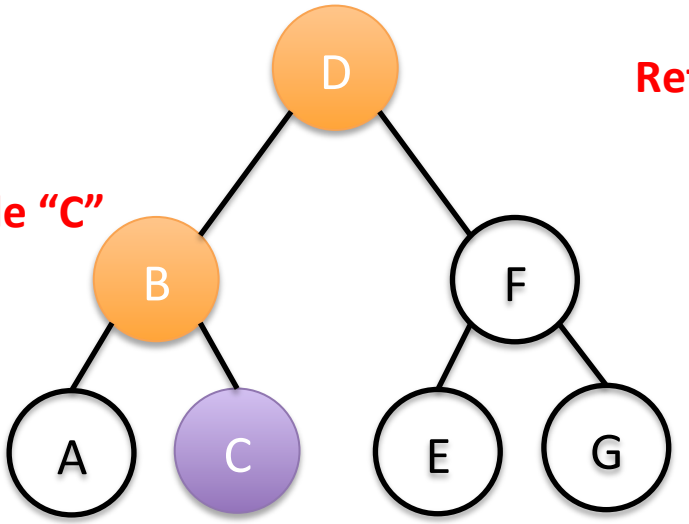
D

B

V value = t.find("C")

Return Value of node C

Value of node "C"



Using Comparable makes finding a Key in a BST easy

BST.java

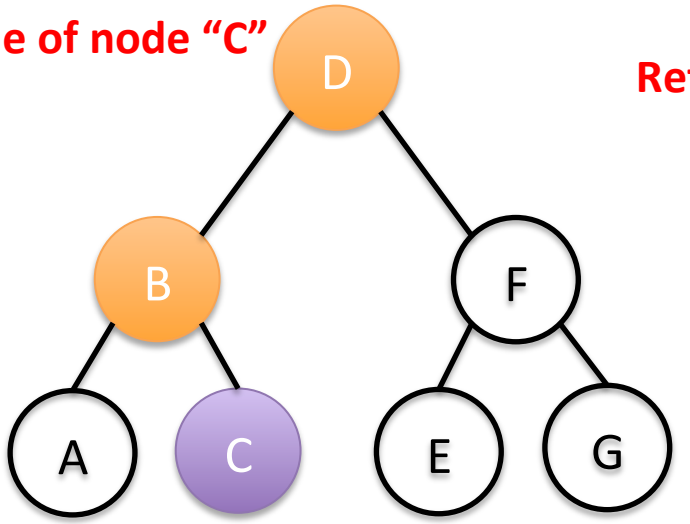
```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

D

V value = t.find("C")

Value of node "C"

Return Value of node C



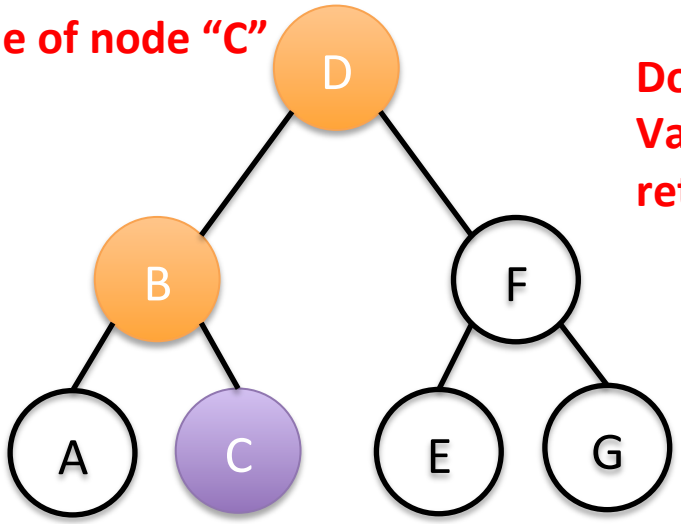
Using Comparable makes finding a Key in a BST easy

BST.java

```
54  public V find(K search) throws InvalidKeyException {
55      System.out.println(key); // to illustrate search traversal
56      int compare = search.compareTo(key);  //compare search with
57      if (compare == 0) return value; //found it
58      if (compare < 0 && hasLeft()) return left.find(search); //s
59      if (compare > 0 && hasRight()) return right.find(search); /
60      throw new InvalidKeyException(search.toString()); //can't g
61  }
```

V value = t.find("C")

Value of node "C"



Done
Value of node "C"
returned

Comparable also helps inserting new Nodes

BST.java

Inserting new K *key* and V *value*

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

Comparable also helps inserting new Nodes

BST.java

Inserting new K *key* and V *value*
- If find *key*, replace it's *value*

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

Comparable also helps inserting new Nodes

BST.java

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

Inserting new K *key* and V *value*

- **If find *key*, replace it's *value***
- **Traverse left if *key* < this node's *key***
- **If no left child, create a new node as the left child**

Comparable also helps inserting new Nodes

BST.java

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

Inserting new K *key* and V *value*

- **If find *key*, replace it's *value***
- **Traverse left if *key* < this node's *key***
- **If no left child, create a new node as the left child**
- **Traverse right if *key* > this node's *key***
- **If no right child, create a new Node as the right child**

Comparable also helps inserting new Nodes

BST.java

t = node "D"


```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

Comparable also helps inserting new Nodes

BST.java

t.insert("B")


```
82      */
83    public void insert(K key, V value) {
84        int compare = key.compareTo(this.key);
85        if (compare == 0) {
86            // replace
87            this.value = value;
88        }
89        else if (compare < 0) {
90            // insert on left (new leaf if no left)
91            if (hasLeft()) left.insert(key, value);
92            else left = new BST<K,V>(key, value);
93        }
94        else if (compare > 0) {
95            // insert on right (new leaf if no right)
96            if (hasRight()) right.insert(key, value);
97            else right = new BST<K,V>(key, value);
98        }
99    }
```

Comparable also helps inserting new Nodes

BST.java

t.insert("B")


```
83    public void insert(K key, V value) {
84        int compare = key.compareTo(this.key);
85        if (compare == 0) {
86            // replace
87            this.value = value;
88        }
89        else if (compare < 0) {
90            // insert on left (new leaf if no left)
91            if (hasLeft()) left.insert(key, value);
92            else left = new BST<K,V>(key, value);
93        }
94        else if (compare > 0) {
95            // insert on right (new leaf if no right)
96            if (hasRight()) right.insert(key, value);
97            else right = new BST<K,V>(key, value);
98        }
99    }
```

"B" < "D"

compare = -1

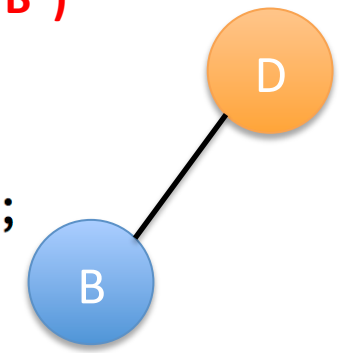
Comparable also helps inserting new Nodes

BST.java

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

D

t.insert("B")



"B" < "D"
compare = -1
No left child
Add "B" as left

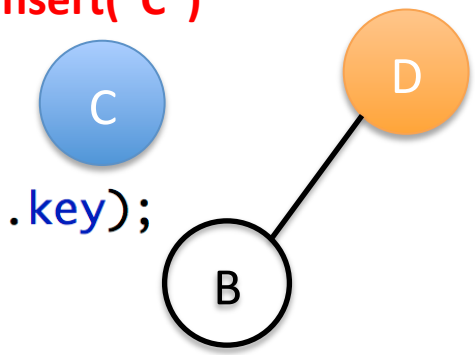
Comparable also helps inserting new Nodes

BST.java

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

D

t.insert("C")



"C" < "D"

compare = -1

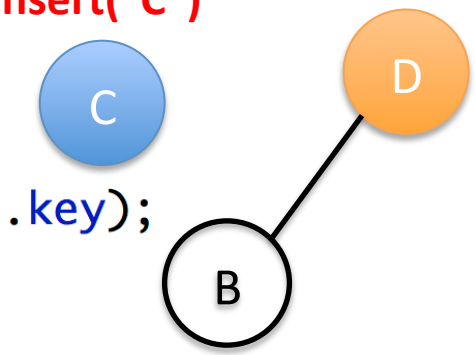
Comparable also helps inserting new Nodes

BST.java

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

D

t.insert("C")



"C" < "D"

compare = -1

Has left

traverse left

Comparable also helps inserting new Nodes

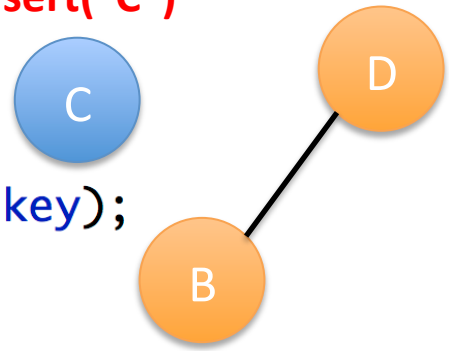
BST.java

```
83    public void insert(K key, V value) {
84        int compare = key.compareTo(this.key);
85        if (compare == 0) {
86            // replace
87            this.value = value;
88        }
89        else if (compare < 0) {
90            // insert on left (new leaf if no left)
91            if (hasLeft()) left.insert(key, value);
92            else left = new BST<K,V>(key, value);
93        }
94        else if (compare > 0) {
95            // insert on right (new leaf if no right)
96            if (hasRight()) right.insert(key, value);
97            else right = new BST<K,V>(key, value);
98        }
99    }
```

B

D

t.insert("C")



"C" > "B"

compare = 1

Comparable also helps inserting new Nodes

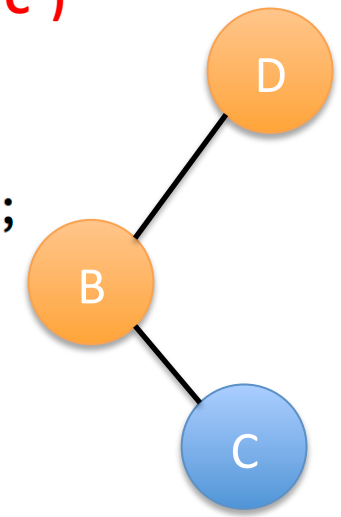
BST.java

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

D

B

t.insert("C")



"C" > "B"

compare = 1

No right child

Add "C" as right

Comparable also helps inserting new Nodes

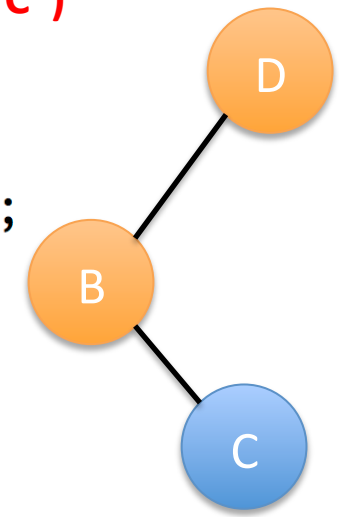
BST.java

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
    }
```

D

B

t.insert("C")



"C" > "B"

compare = 1

No right child

Add "C" as right

B ends

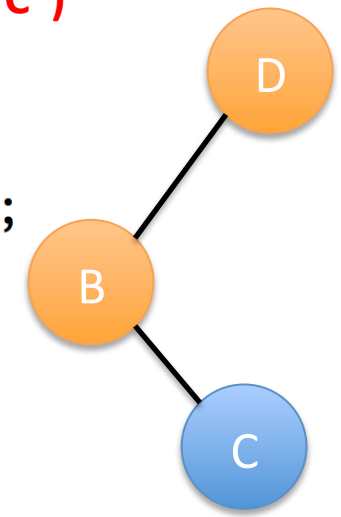
Comparable also helps inserting new Nodes

BST.java

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

D

t.insert("C")



"C" > "B"

compare = 1

No right child

Add "C" as right

B ends

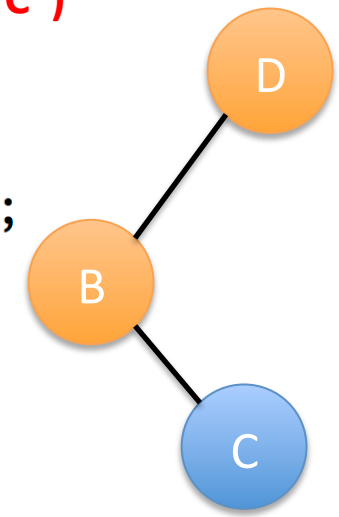
D ends

Comparable also helps inserting new Nodes

BST.java

```
83  public void insert(K key, V value) {
84      int compare = key.compareTo(this.key);
85      if (compare == 0) {
86          // replace
87          this.value = value;
88      }
89      else if (compare < 0) {
90          // insert on left (new leaf if no left)
91          if (hasLeft()) left.insert(key, value);
92          else left = new BST<K,V>(key, value);
93      }
94      else if (compare > 0) {
95          // insert on right (new leaf if no right)
96          if (hasRight()) right.insert(key, value);
97          else right = new BST<K,V>(key, value);
98      }
99  }
```

t.insert("C")



"C" > "B"

compare = 1

No right child

Add "C" as right

B ends

D ends

Done

Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

Delete node with Key *search*

Return updated tree (or throw exception if Key not found)

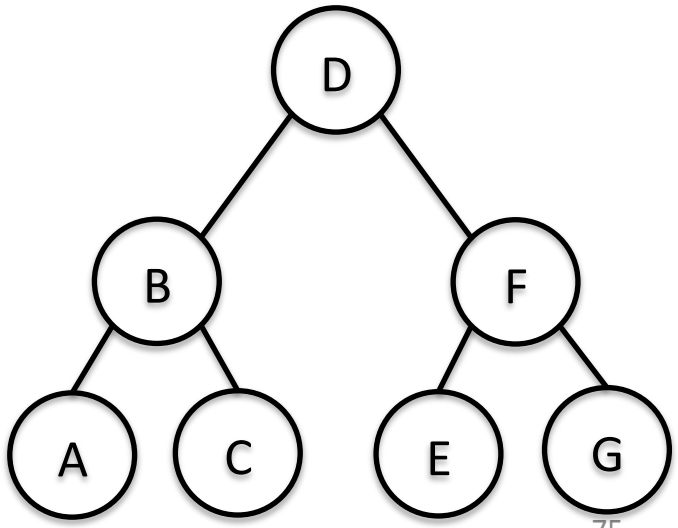
```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

t = Node “D”

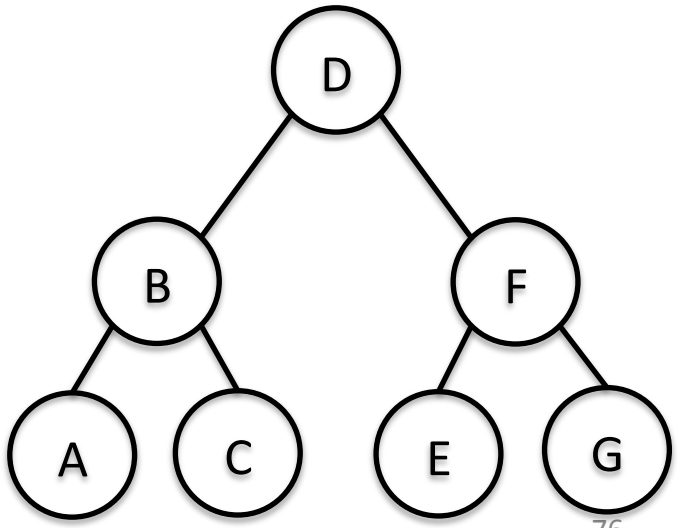


Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

t = t.delete("A")



Deleting a Node removes it from the tree and returns updated tree to caller

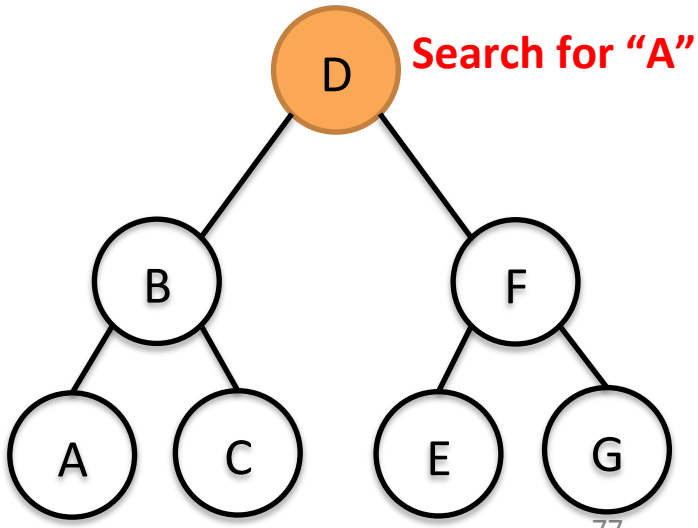
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("A")

Search for "A"



Deleting a Node removes it from the tree and returns updated tree to caller

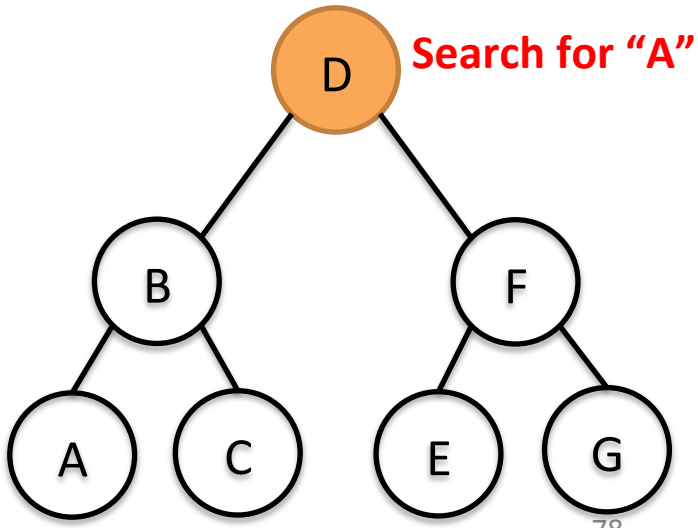
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("A")

Search for "A"



Deleting a Node removes it from the tree and returns updated tree to caller

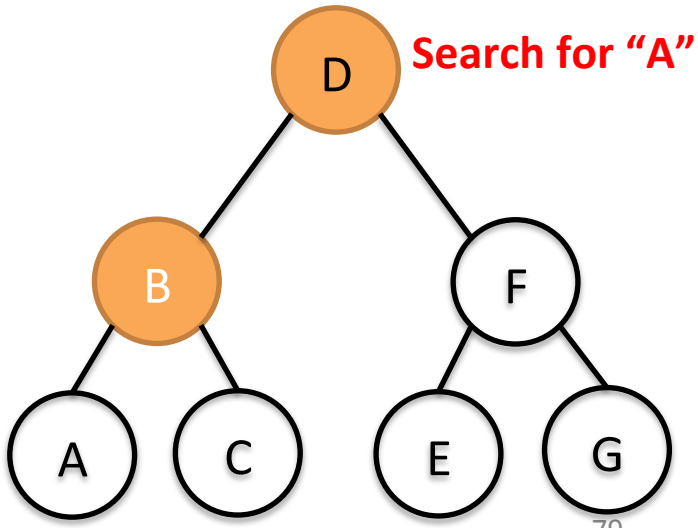
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

B
D

t = t.delete("A")

Search for "A"



Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

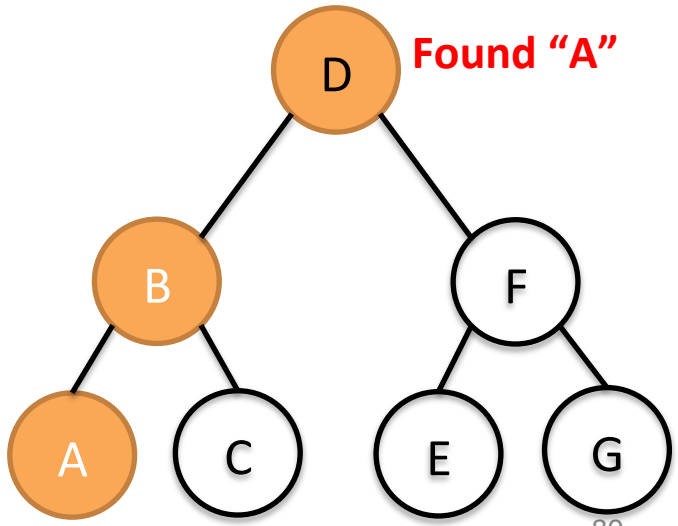
A

B

D

t = t.delete("A")

Found "A"



Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

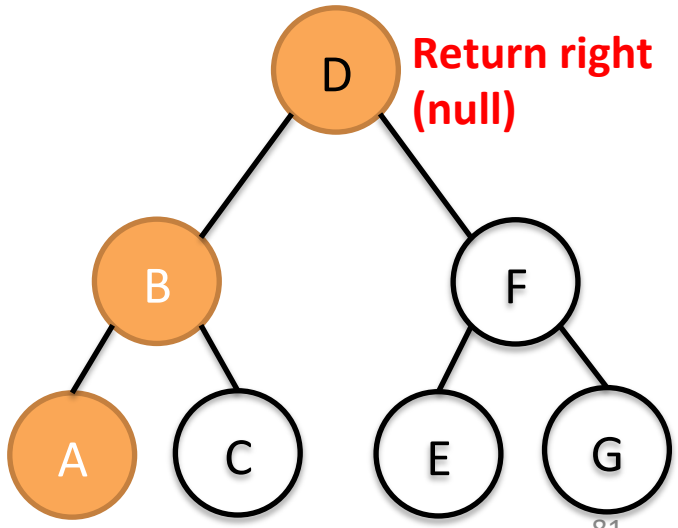
```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

A

B

D

t = t.delete("A")



Deleting a Node removes it from the tree and returns updated tree to caller

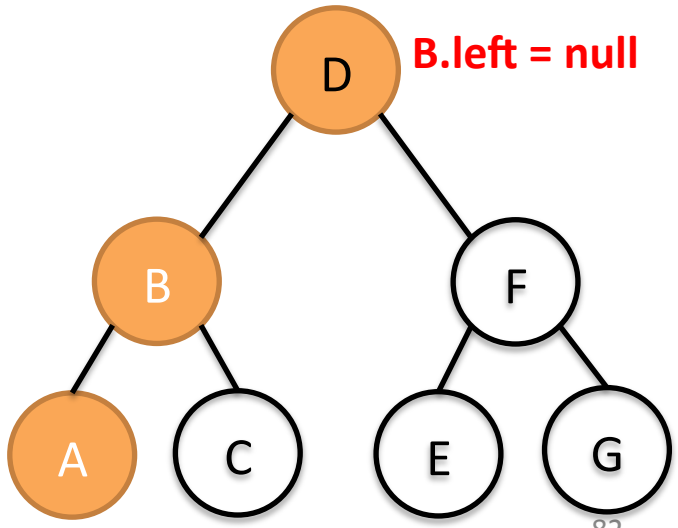
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

B

t = t.delete("A")

B.left = null



Deleting a Node removes it from the tree and returns updated tree to caller

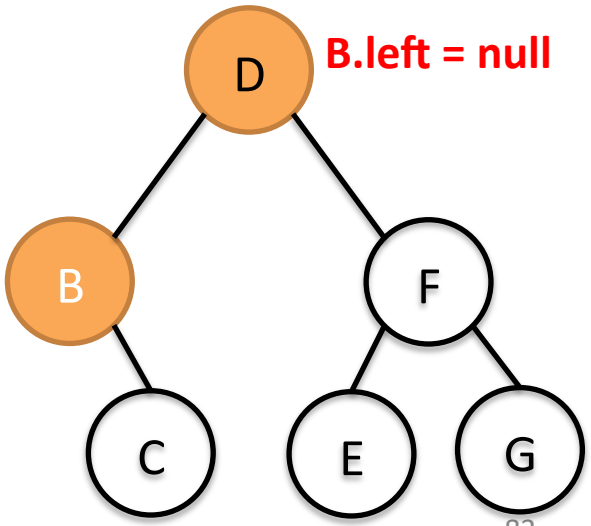
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

B

t = t.delete("A")

B.left = null



Deleting a Node removes it from the tree and returns updated tree to caller

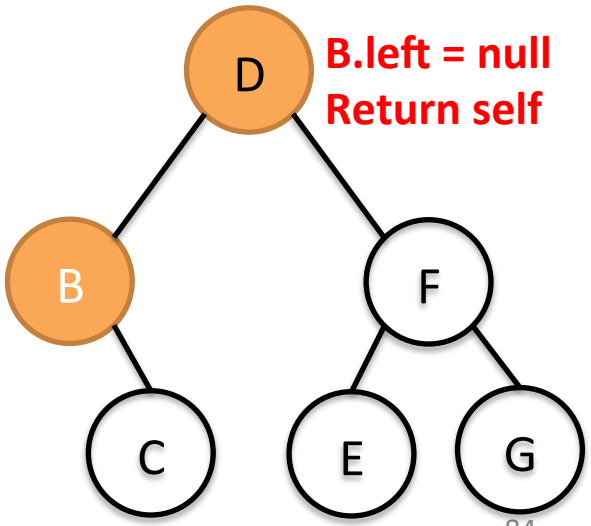
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

B

t = t.delete("A")



Deleting a Node removes it from the tree and returns updated tree to caller

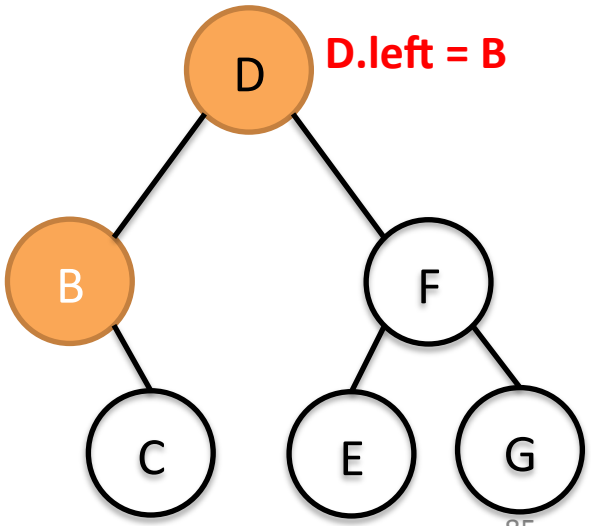
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("A")

D.left = B



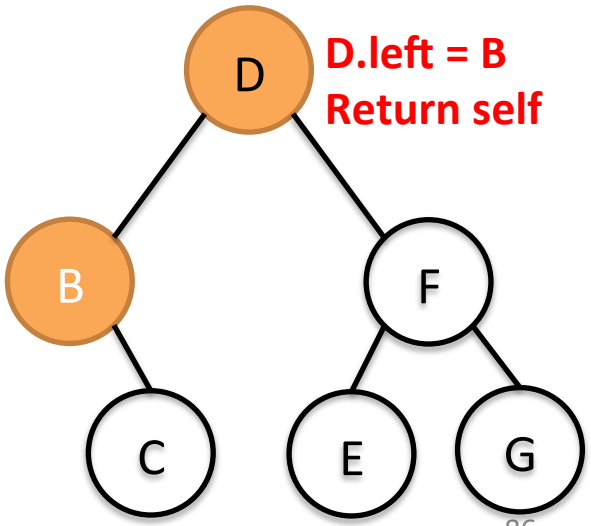
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("A")



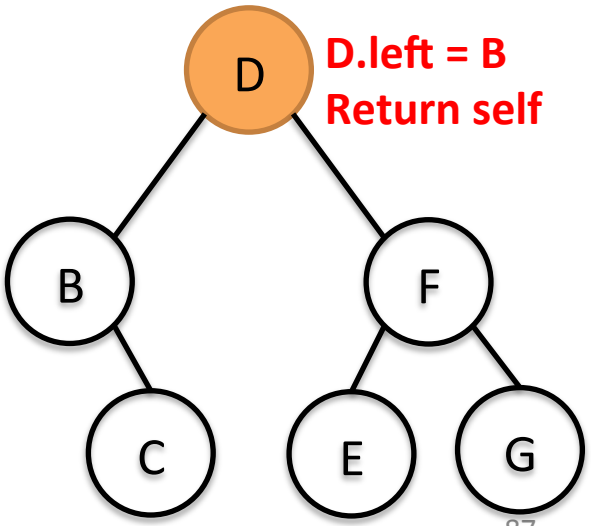
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("A")

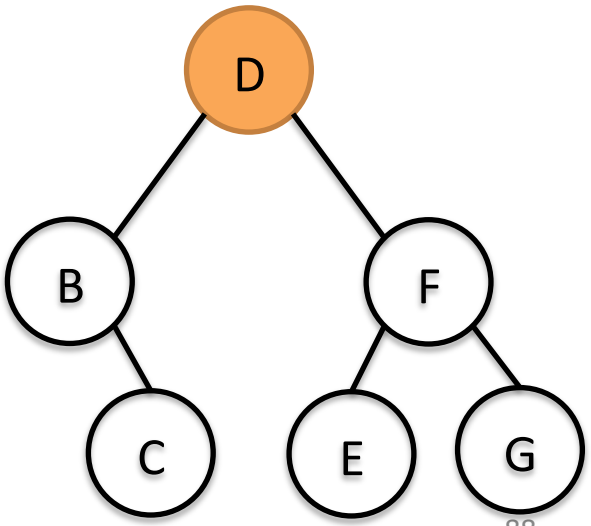


Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

t = Node "D"

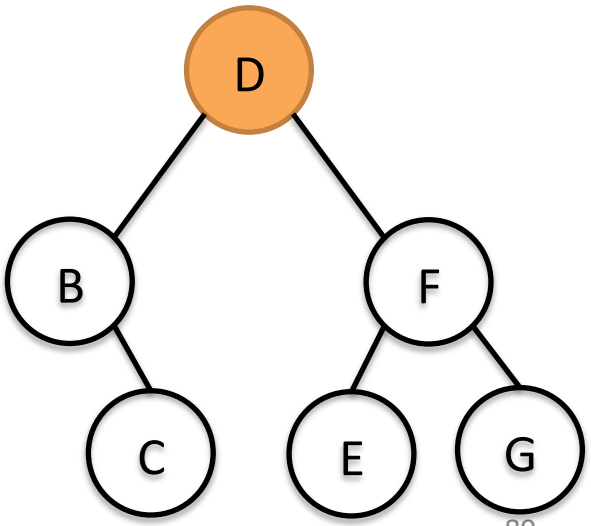


Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

t = t.delete("B")



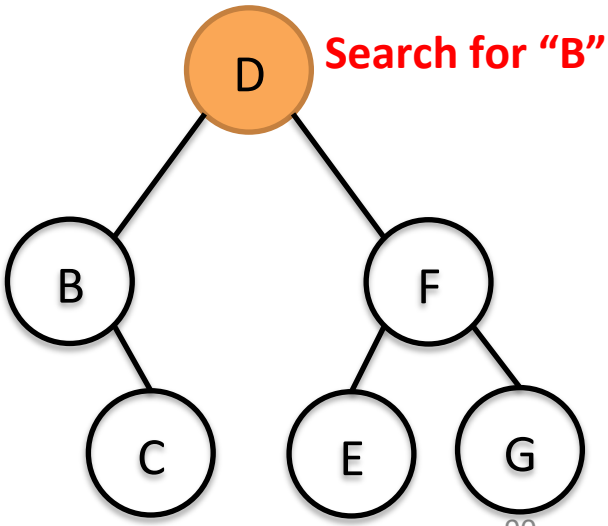
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("B")



Deleting a Node removes it from the tree and returns updated tree to caller

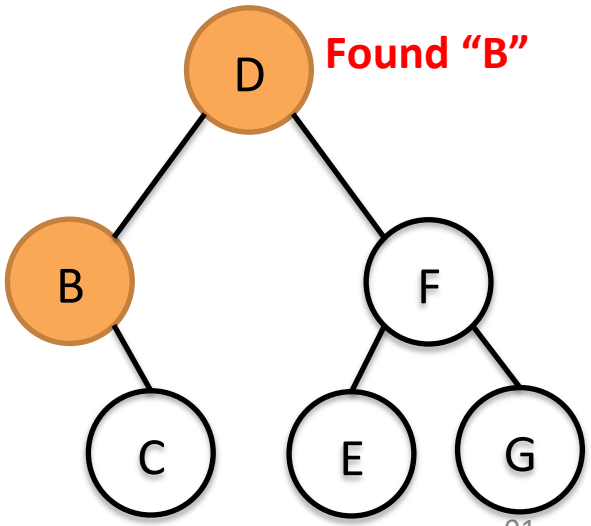
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

B

D

t = t.delete("B")



Deleting a Node removes it from the tree and returns updated tree to caller

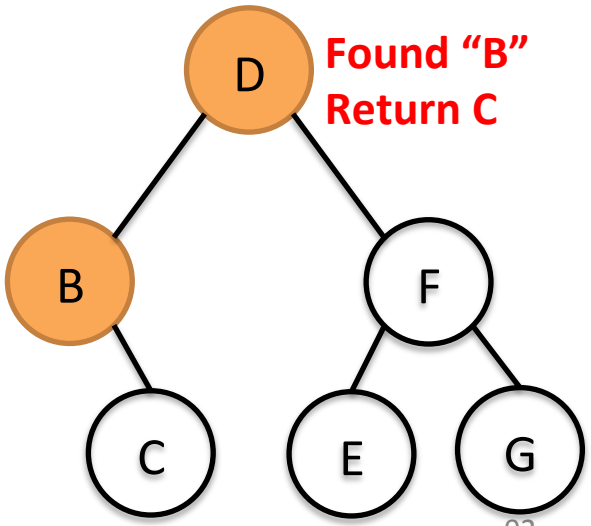
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

B

D

t = t.delete("B")



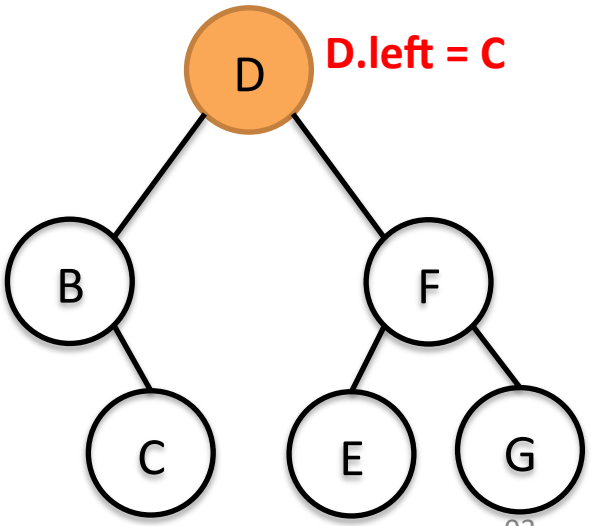
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("B")



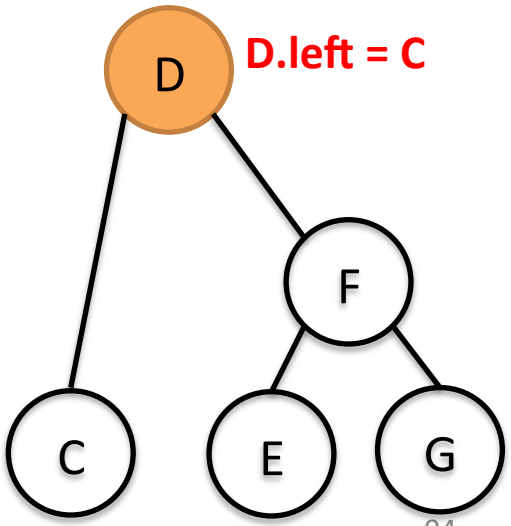
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("B")



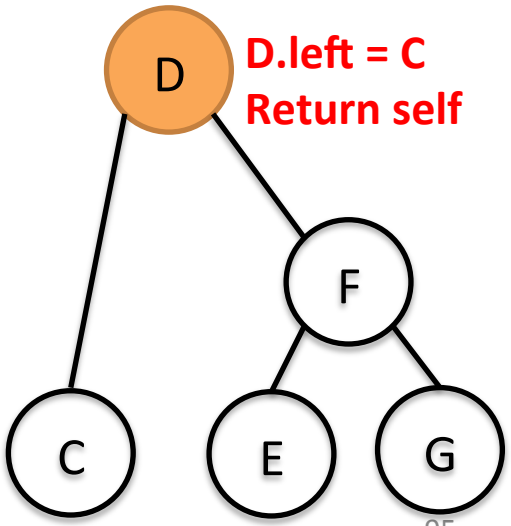
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("B")

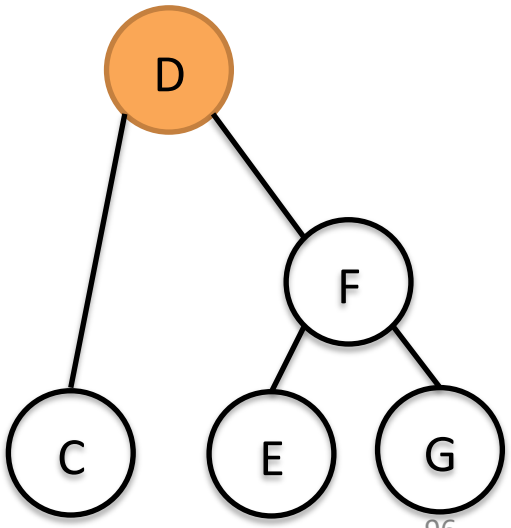


Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

t = Node "D"



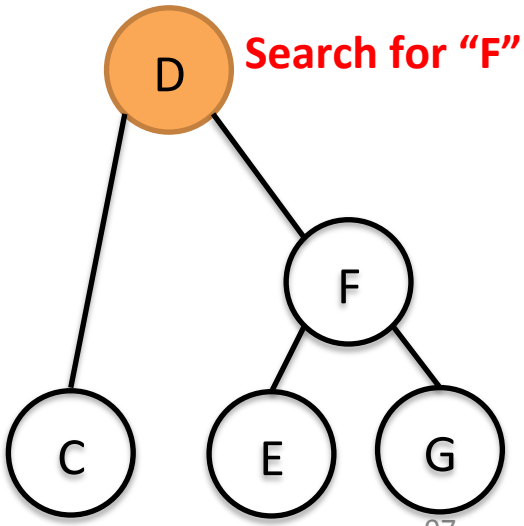
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("F")



Deleting a Node removes it from the tree and returns updated tree to caller

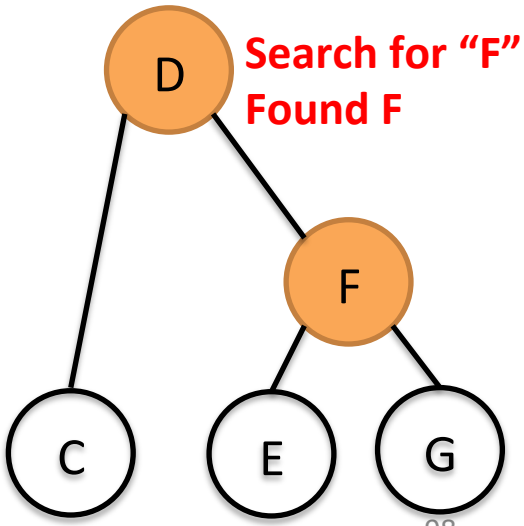
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

F

D

t = t.delete("F")



Deleting a Node removes it from the tree and returns updated tree to caller

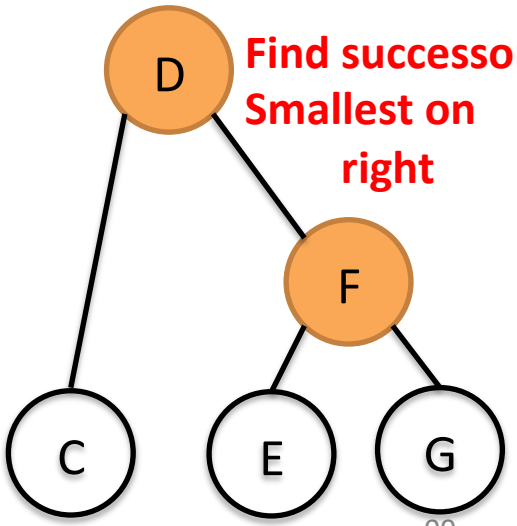
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

F

D

t = t.delete("F")



Deleting a Node removes it from the tree and returns updated tree to caller

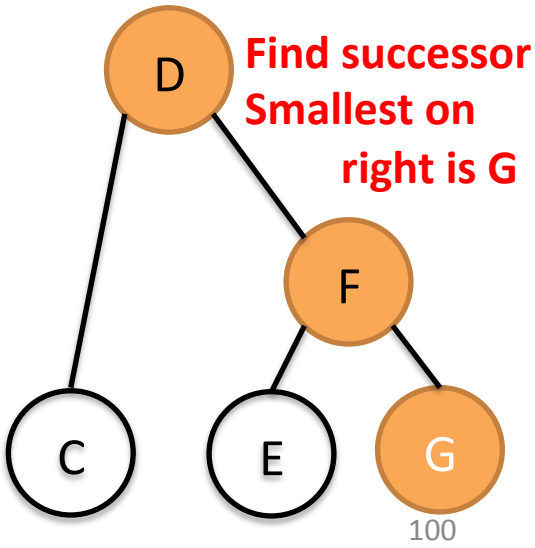
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

F

D

t = t.delete("F")



Deleting a Node removes it from the tree and returns updated tree to caller

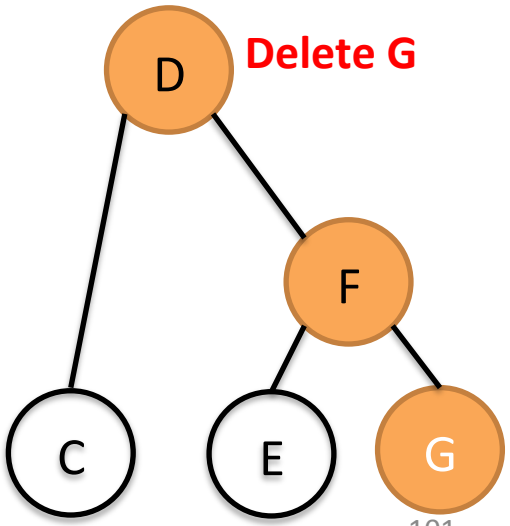
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

F

D

t = t.delete("F")



Deleting a Node removes it from the tree and returns updated tree to caller

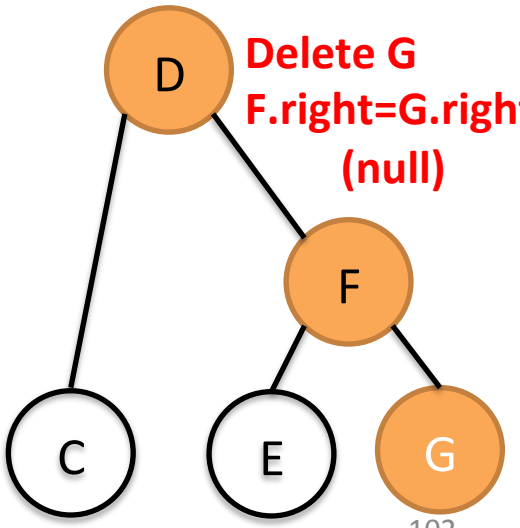
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

F

D

t = t.delete("F")



Deleting a Node removes it from the tree and returns updated tree to caller

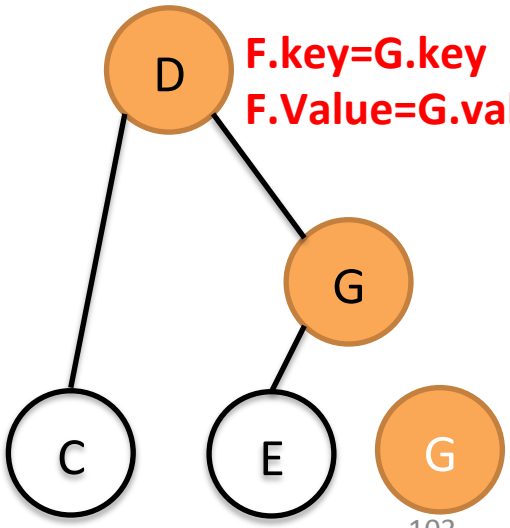
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

F

D

t = t.delete("F")



Deleting a Node removes it from the tree and returns updated tree to caller

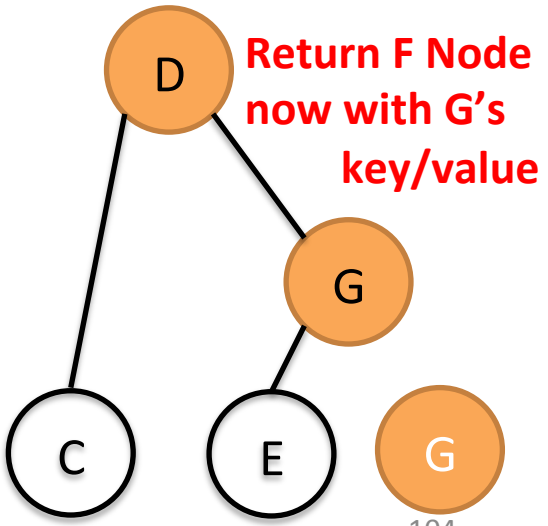
BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

F

D

t = t.delete("F")



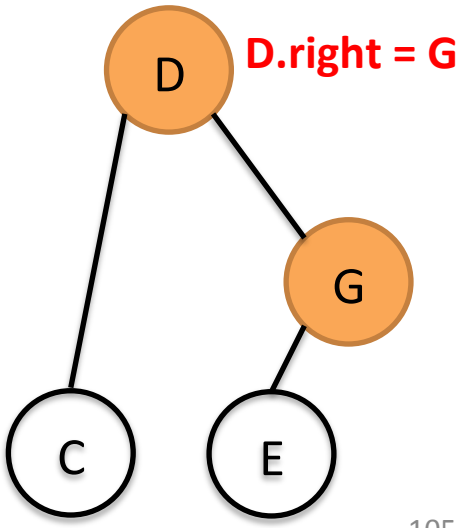
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

D

t = t.delete("F")



Deleting a Node removes it from the tree and returns updated tree to caller

BST.java

```
105     public BST<K,V> delete(K search) throws InvalidKeyException {
106         int compare = search.compareTo(key);
107         if (compare == 0) {
108             // Easy cases: 0 or 1 child -- return other
109             if (!hasLeft()) return right;  //no left child, return r
110             if (!hasRight()) return left; //has left, but no right,
111             // If both children are there, find successor, delete an
112             BST<K,V> successor = right;
113             while (successor.hasLeft()) successor = successor.left;
114             // Delete it and takes its key & value
115             right = right.delete(successor.key);
116             this.key = successor.key;
117             this.value = successor.value;
118             return this;
119         }
120         else if (compare < 0 && hasLeft()) {
121             left = left.delete(search);
122             return this;
123         }
124         else if (compare > 0 && hasRight()) {
125             right = right.delete(search);
126             return this;
```

t = Node "D"