CS 10: Problem solving via Object Oriented Programming

Hierarchies 2: BST

Main goals

- Implement binary search trees
    - Implement find
    - Implement insert
    - Implement delete

- Analyze Binary Search Trees

Agenda

1. Binary search

2. Binary Search Trees (BST)

3. BST find analysis

4. Operations on BSTs

5. Implementation

Binary search on an array

Index	0	1	2	3	4	5	6	7	8
Data	1	5	9	14	25	53	107	214	512

Pseudo code

Looking for target = 53

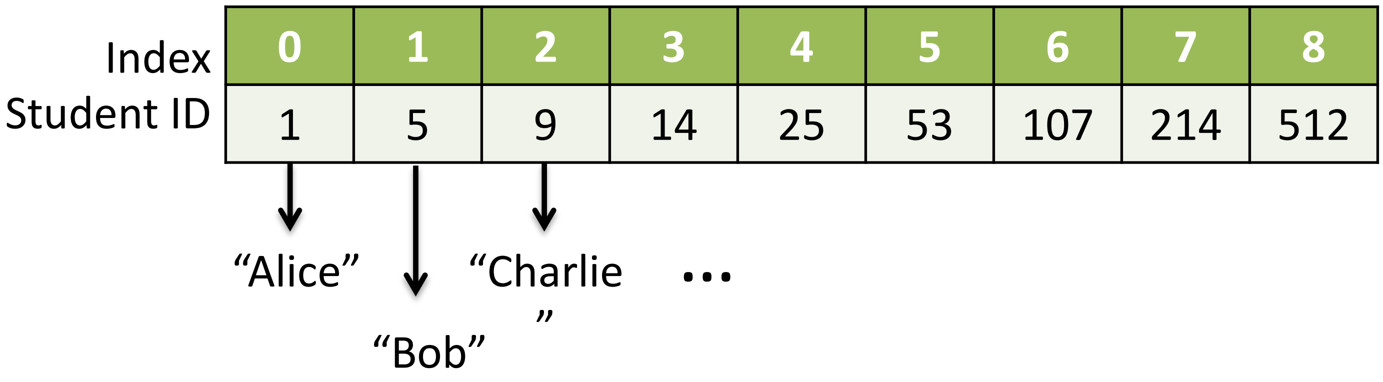
```
Set min = 0, max = n-1
While (min <= max) {
    idx = (min + max)/2
    If array[idx] == target
        return idx
    else if array[idx] > target
        max = idx-1
    else
        min = idx +1
}
```

On paper run

Complexity?

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

Key: Student ID, Value: Student name



Index	0	1	2	3	4	5	6	7	8
Student ID	1	5	9	14	25	53	107	214	512

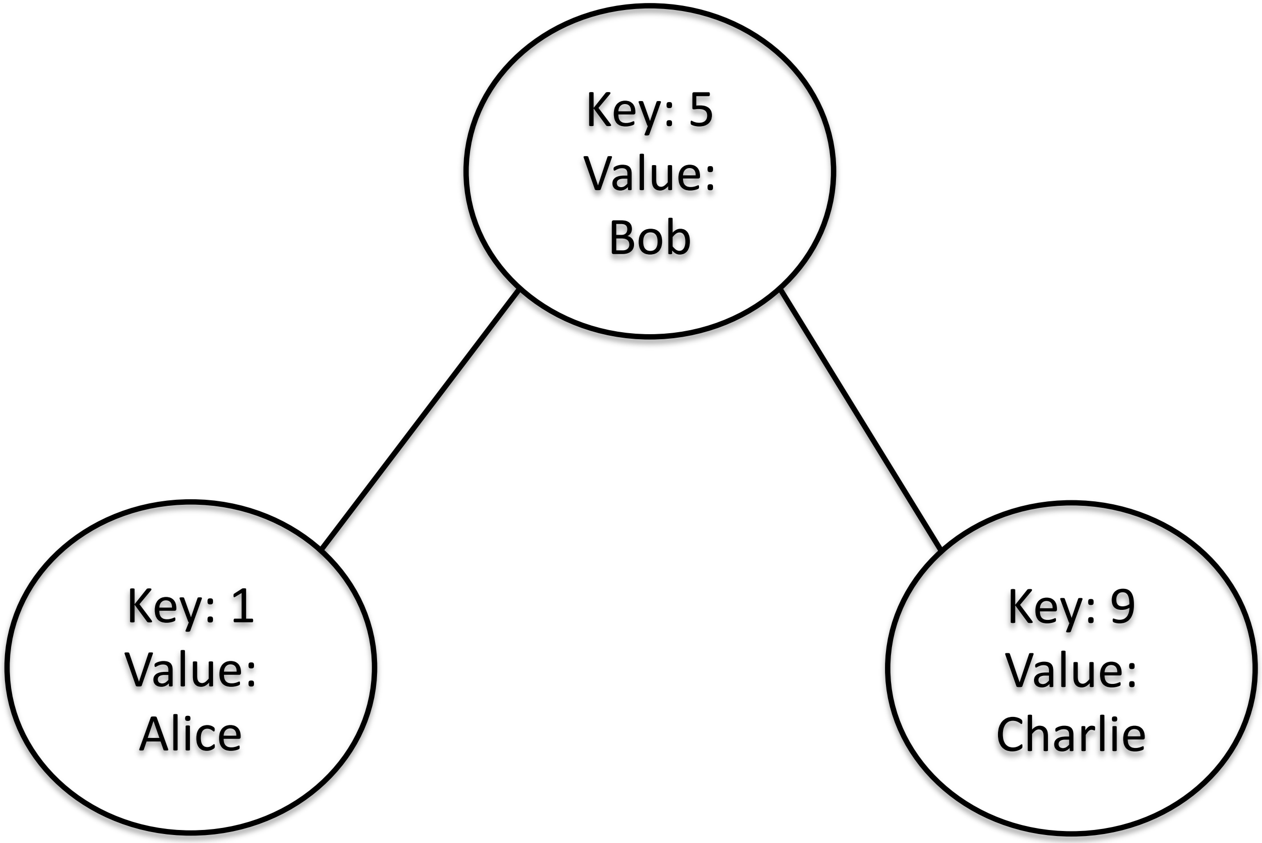
“Alice” “Bob” “Charlie” …

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

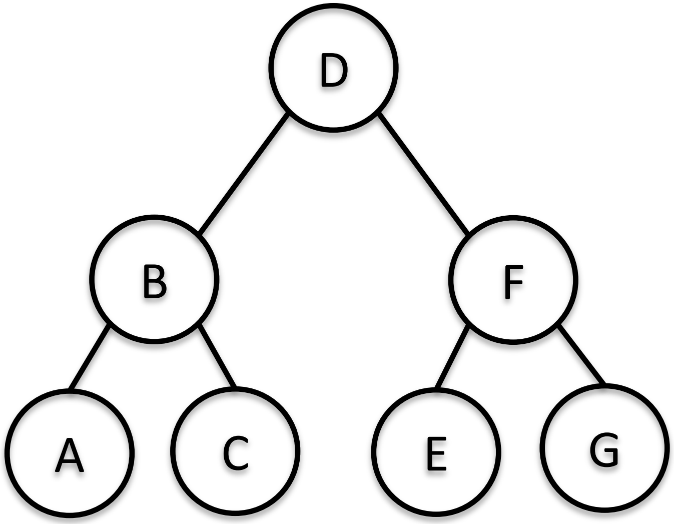
Key: Student ID, Value: Student name



Note: 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

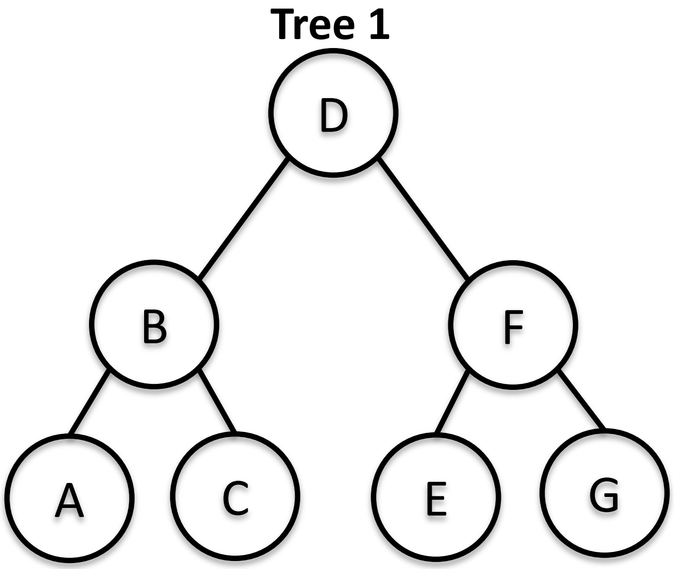


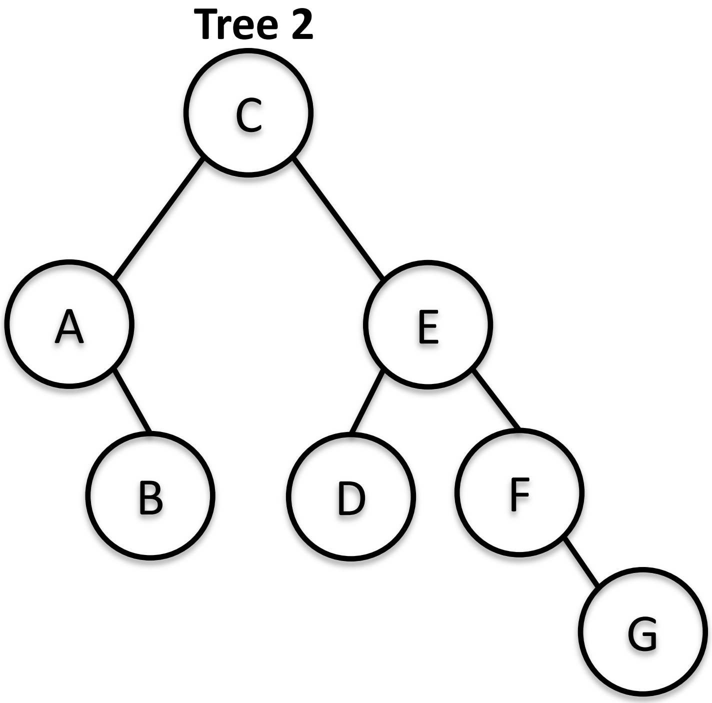
Binary Search Tree property

- Let `x` be a node in a binary search tree such that
  - `left.key < x.key`
  - `right.key > x.key`
- We will maintain this property for all nodes in the BST as we add/remove
- We will assume for now duplicate Keys are not allowed

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

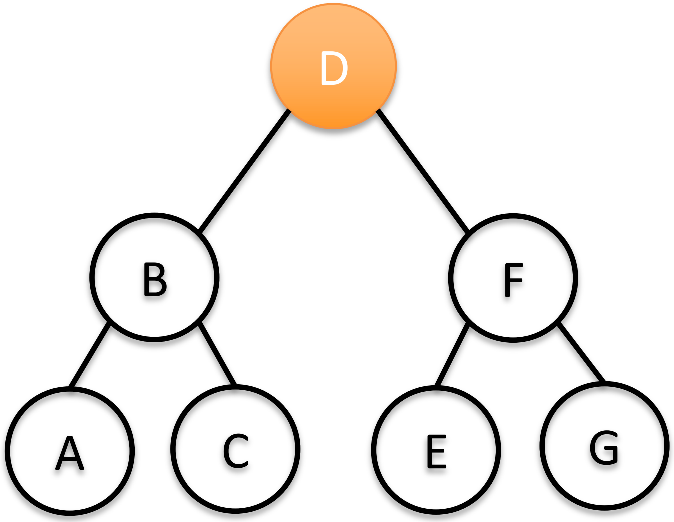
Two valid BSTs with same keys but different structure





BSTs make searching fast and simple

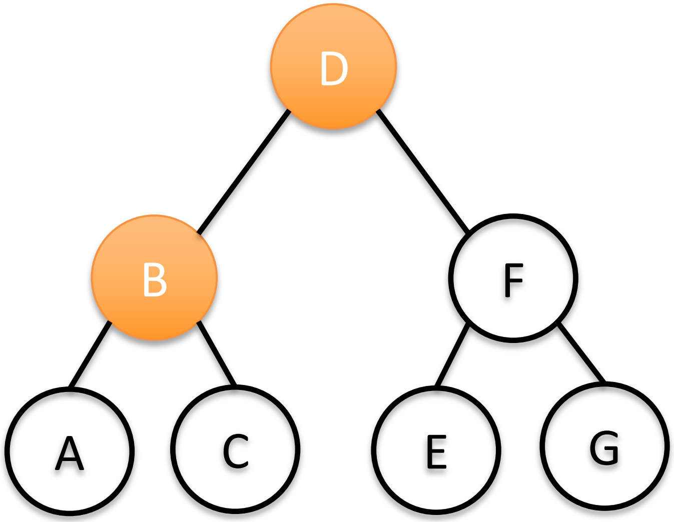
Find Key



Find Key "C"

BSTs make searching fast and simple

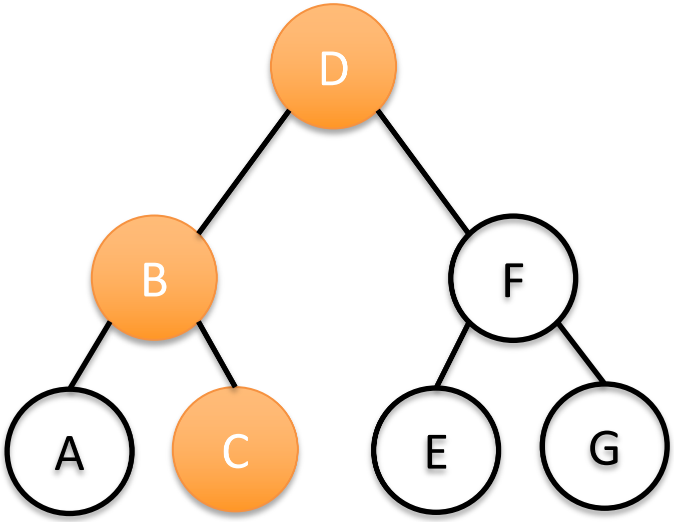
Find Key



Find Key "C"

BSTs make searching fast and simple

Find Key



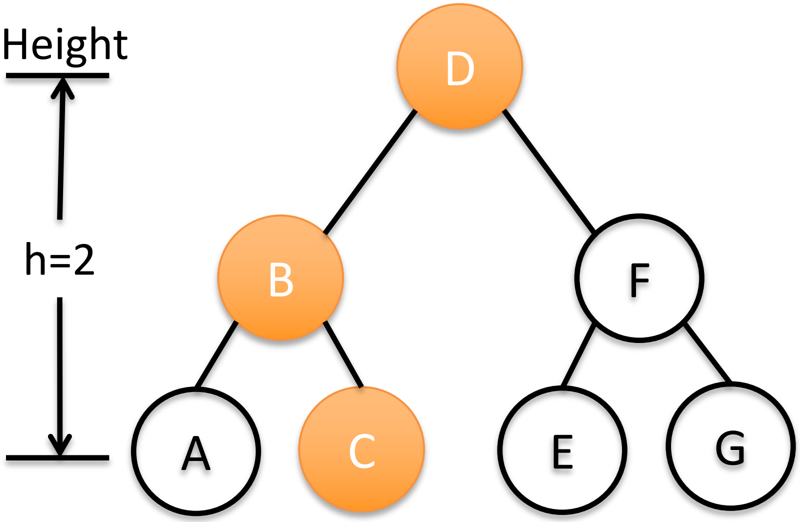
Find Key “C”

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

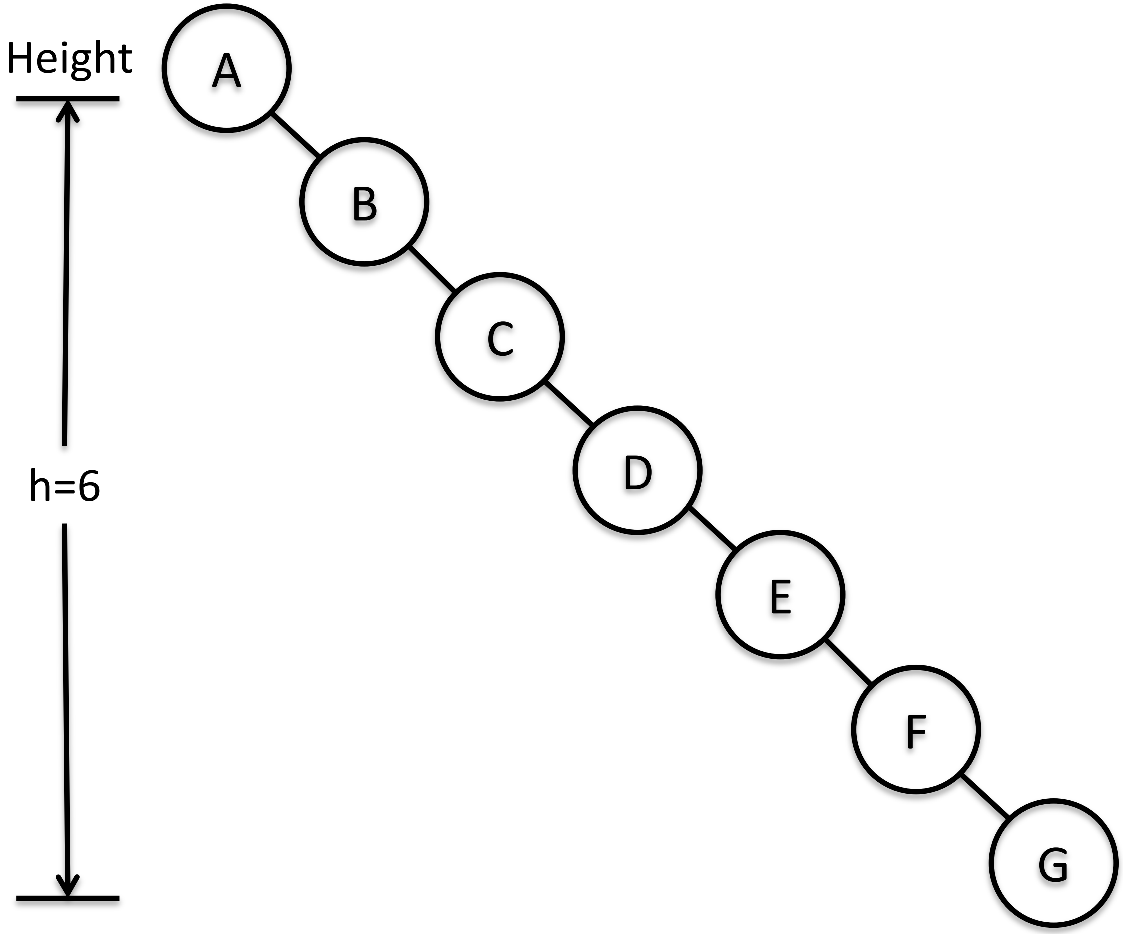
Find Key "C"



Can we say O(log(n))?

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

Find Key "G"

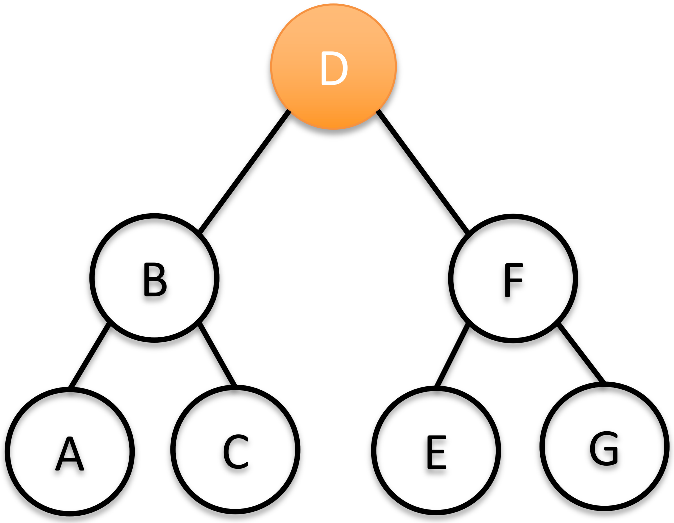


Agenda

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

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

Inserting new node with Key H

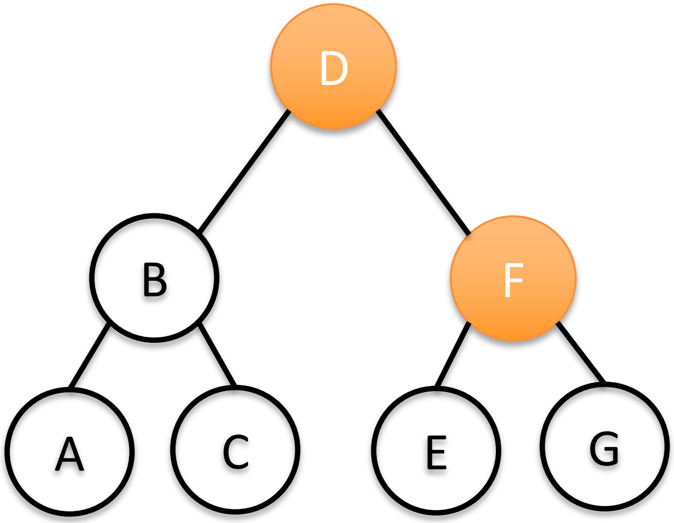


Comments

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

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

Inserting new node with Key H



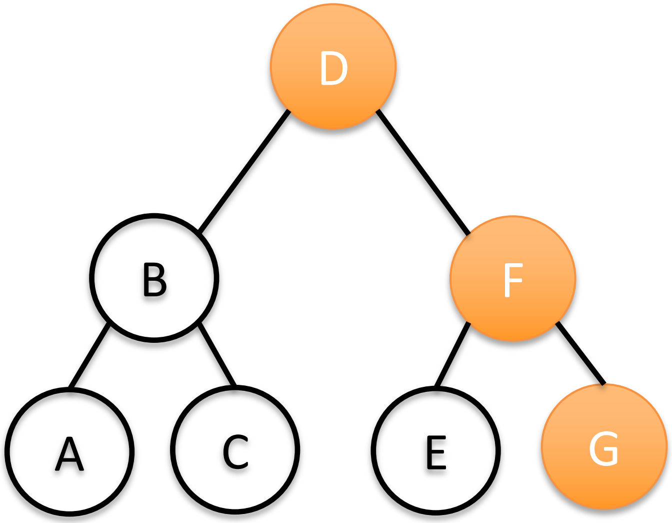
Searching for H

Comments

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

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

Inserting new node with Key H



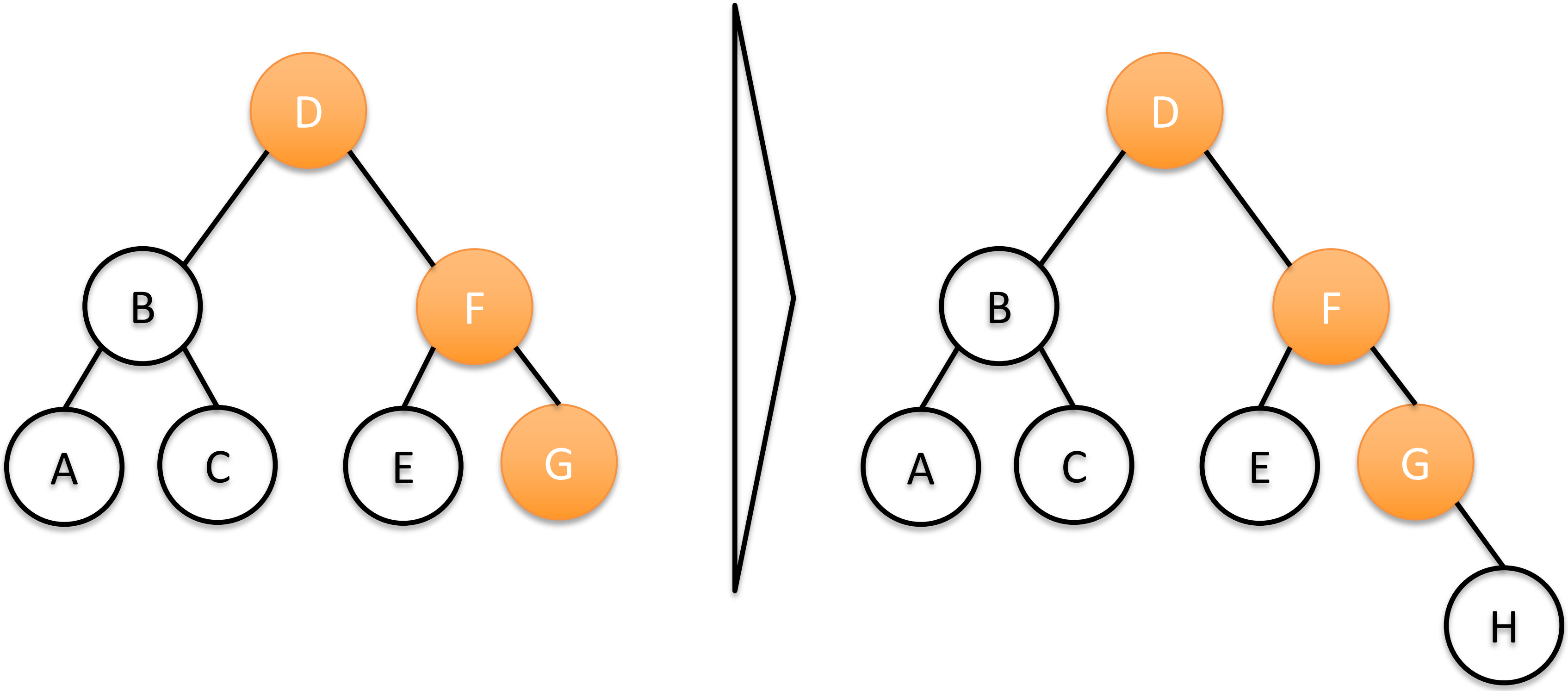
Searching for H

Comments

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

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

Inserting new node with Key H

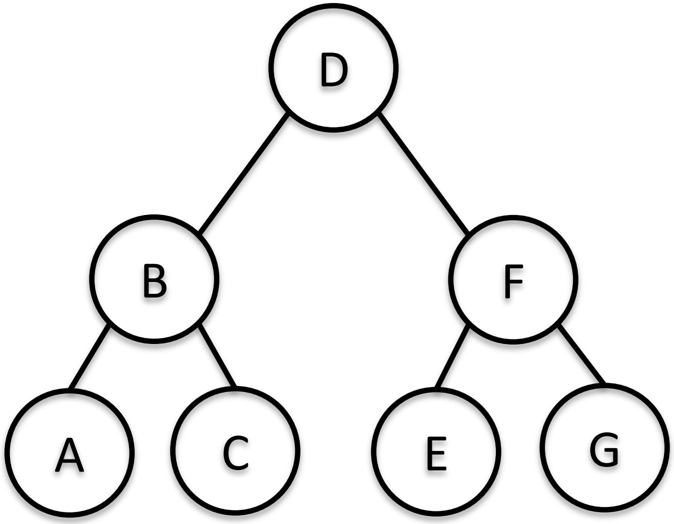


Comments

- Search for Key (H)
  - If found, replace Value
  - If hit end, 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)

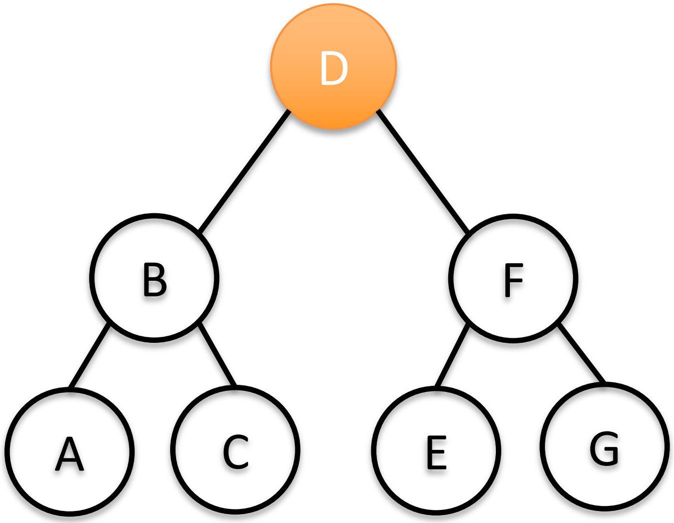


Comments

- 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)



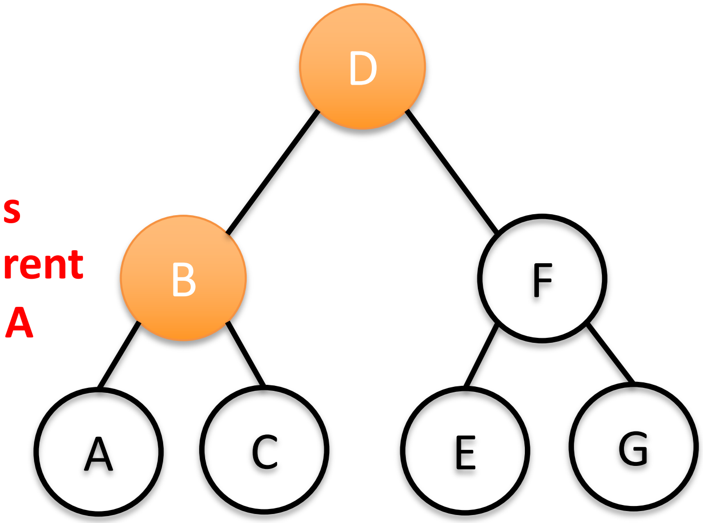
Search for parent of A

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

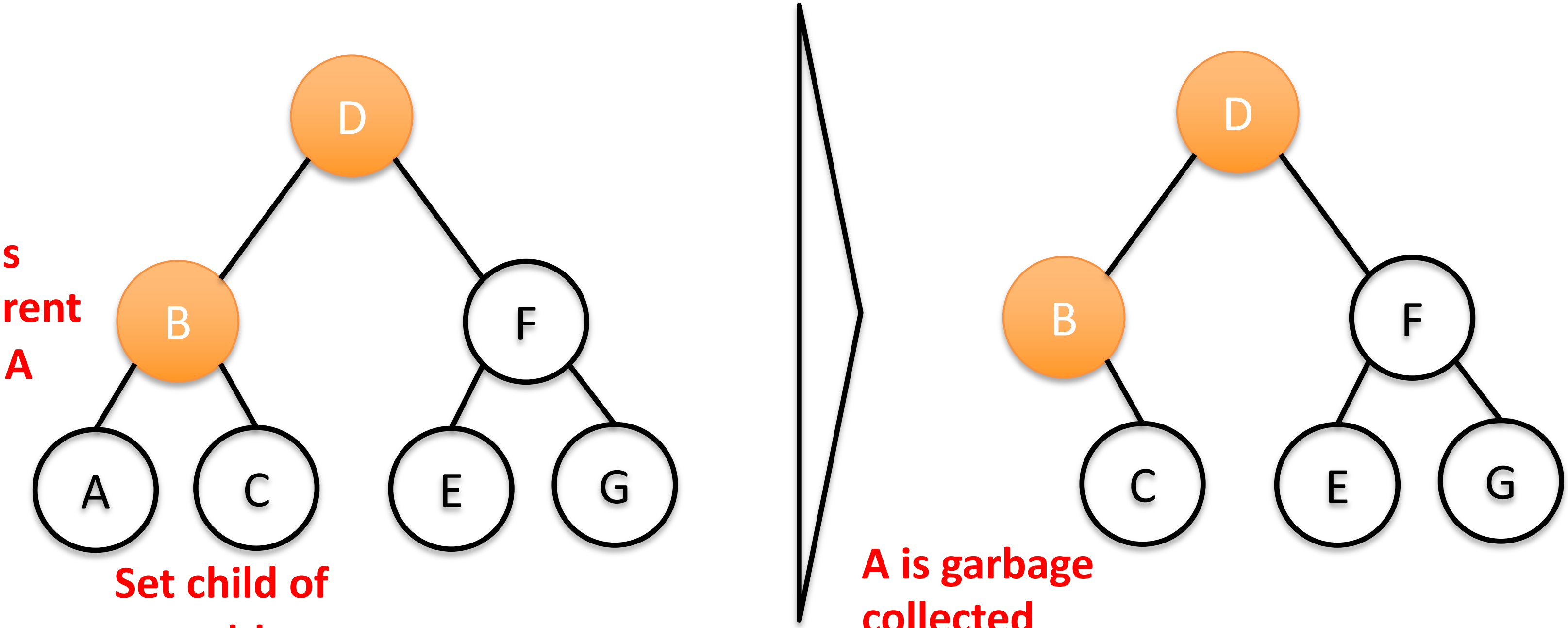
Search for parent of A

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)



B is parent of A

Set child of parent to null

A is garbage collected

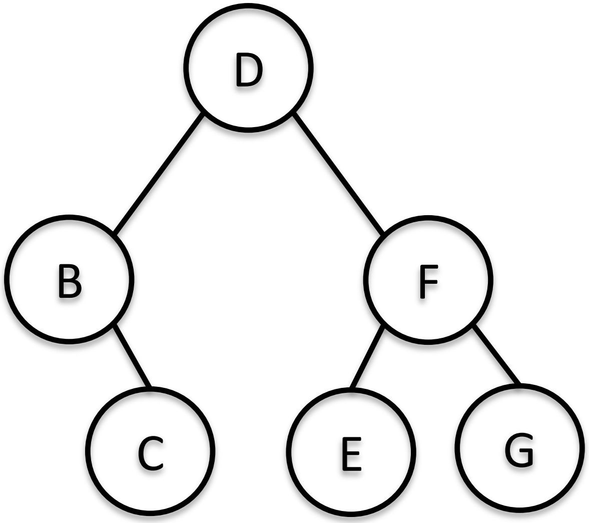
What happens to A?

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)

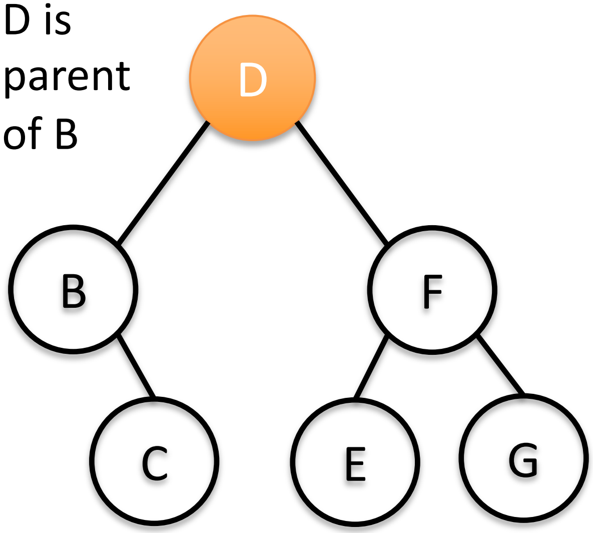


Comments

- 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)



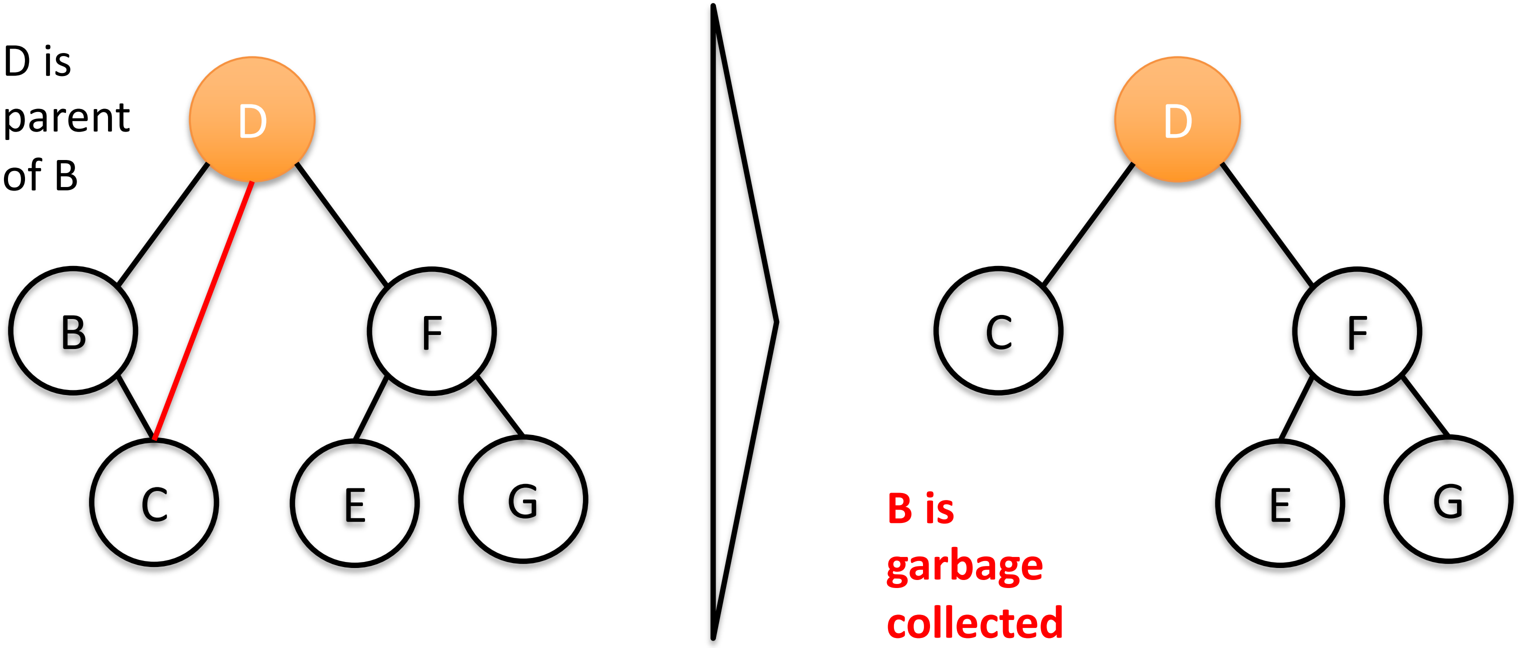
D is parent of B

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)



D is parent of B

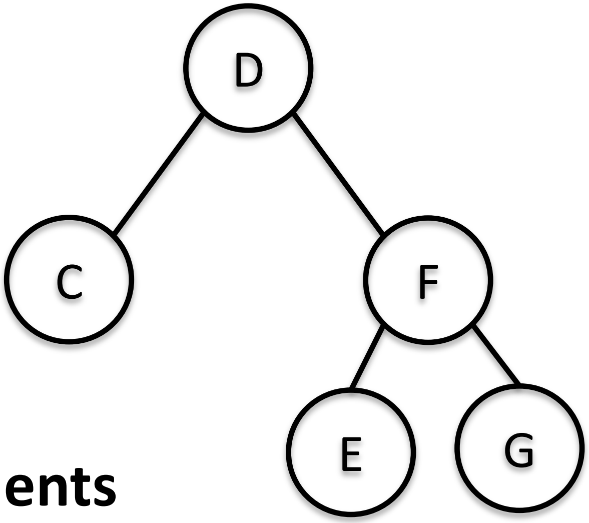
B is garbage collected

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)

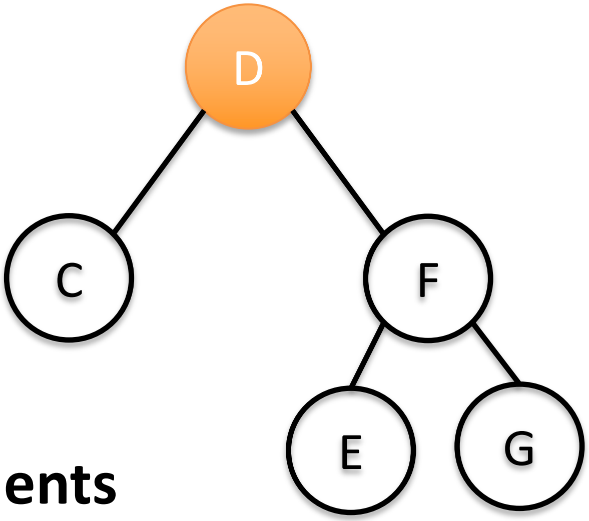


Comments

- 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)

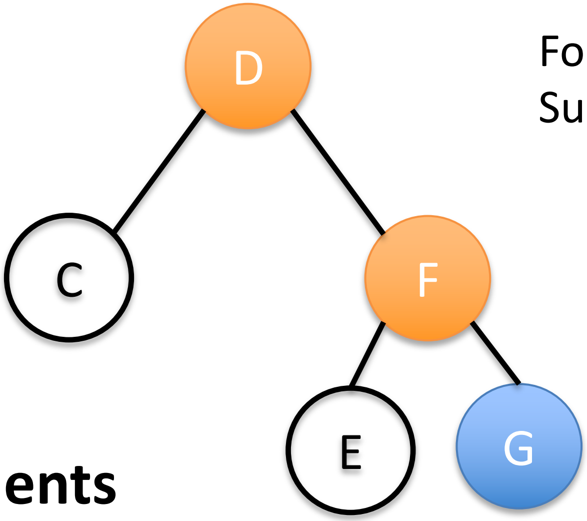


Comments

- 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)



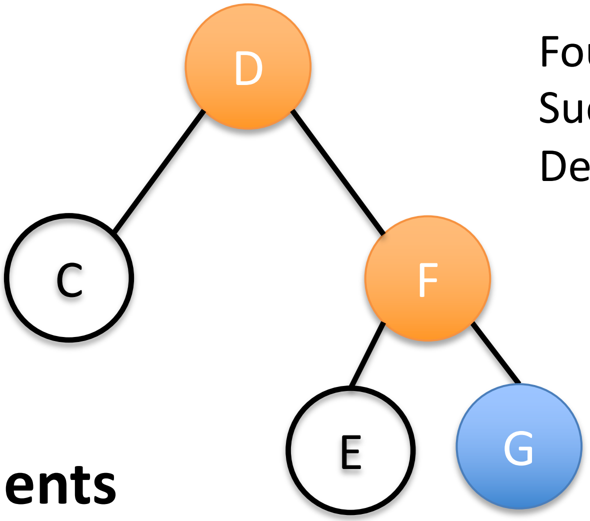
Found F
Successor is smallest on right (G here)

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)



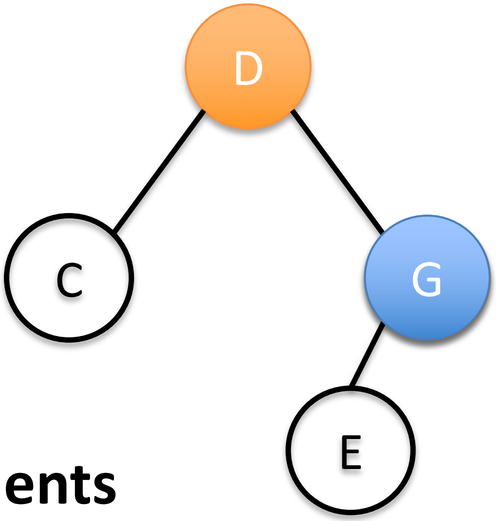
Found F
Successor is smallest on right (G here)
Delete successor

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)



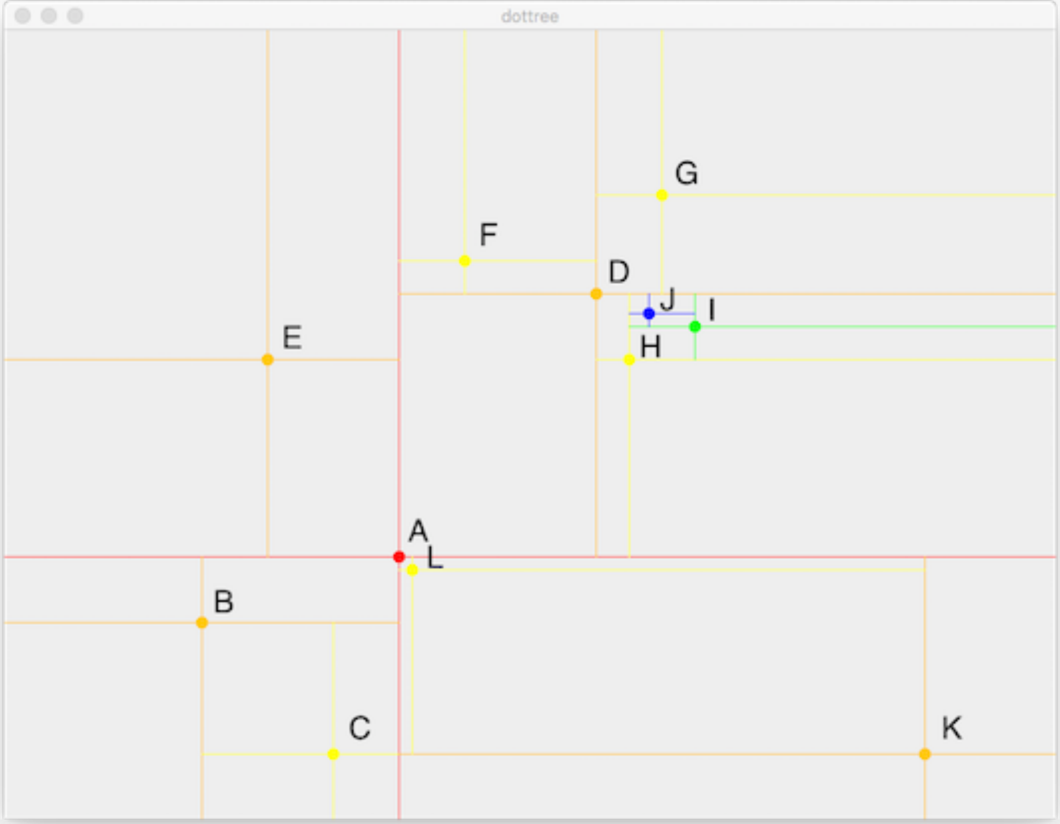
Found F
Successor is smallest on right (G here)
Delete successor
Replace F Key and Value with G Key and Value

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

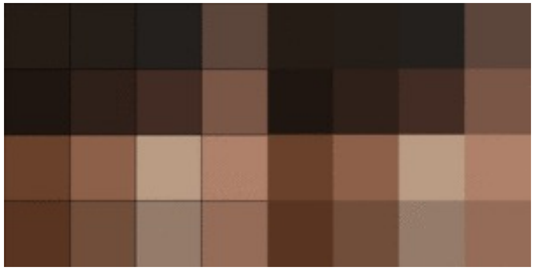
PS-2

Implement quadtree



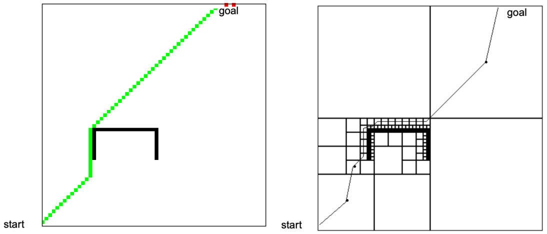
Example of applications

Image compression



Source: wikipedia

Robot path planning



Source: [Yahja et al., 1998, ICRA]

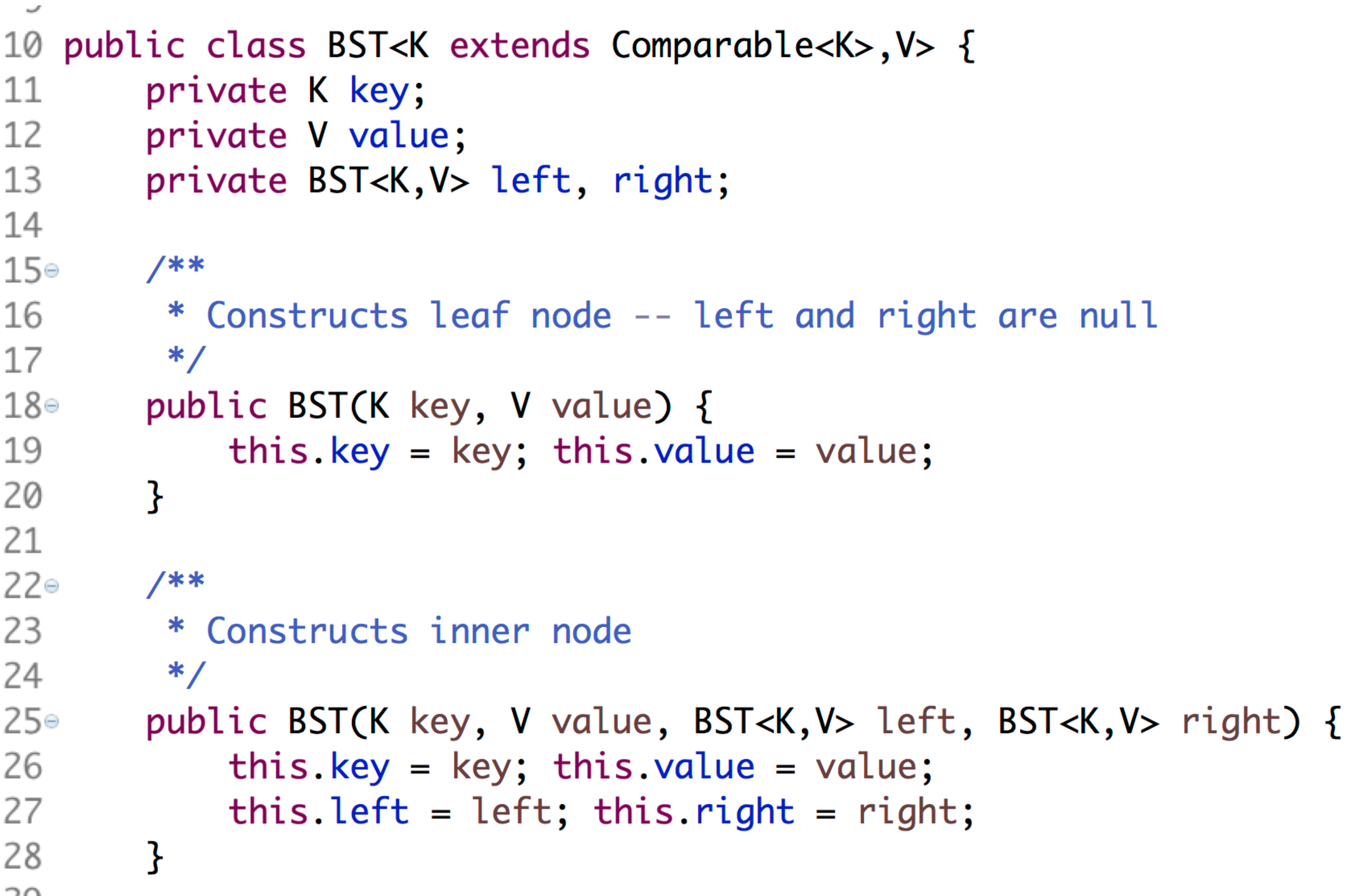
https://www.cs.dartmouth.edu/cs10/PS-2.html

Agenda

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

Binary Search Tree with Key and Value – Key extends comparable

BST.java



```java
public class BST<K extends Comparable<K>,V> {
    private K key;
    private V value;
    private BST<K,V> left, right;

    /**
     * Constructs leaf node -- left and right are null
     */
    public BST(K key, V value) {
        this.key = key; this.value = value;
    }

    /**
     * Constructs inner node
     */
    public BST(K key, V value, BST<K,V> left, BST<K,V> right) {
        this.key = key; this.value = value;
        this.left = left; this.right = right;
    }
```

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

PointWithCompareTo.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

```
/**
 * Compare this blob with another blob
 * @param comparePoint point to compare to this point
 * @return   0 if same,
 *          1 if this point is higher up than comparePoint,
 *         -1 otherwise */
public int compareTo(PointWithCompareTo comparePoint) {
   if (this.y < comparePoint.getY())
     return 1; //this Point is higher up, so it’s bigger
   else if (this.y > comparePoint.getY())
     return -1; //this Point is lower, so it’s smaller
   else return 0; //at same height, so same
}
```

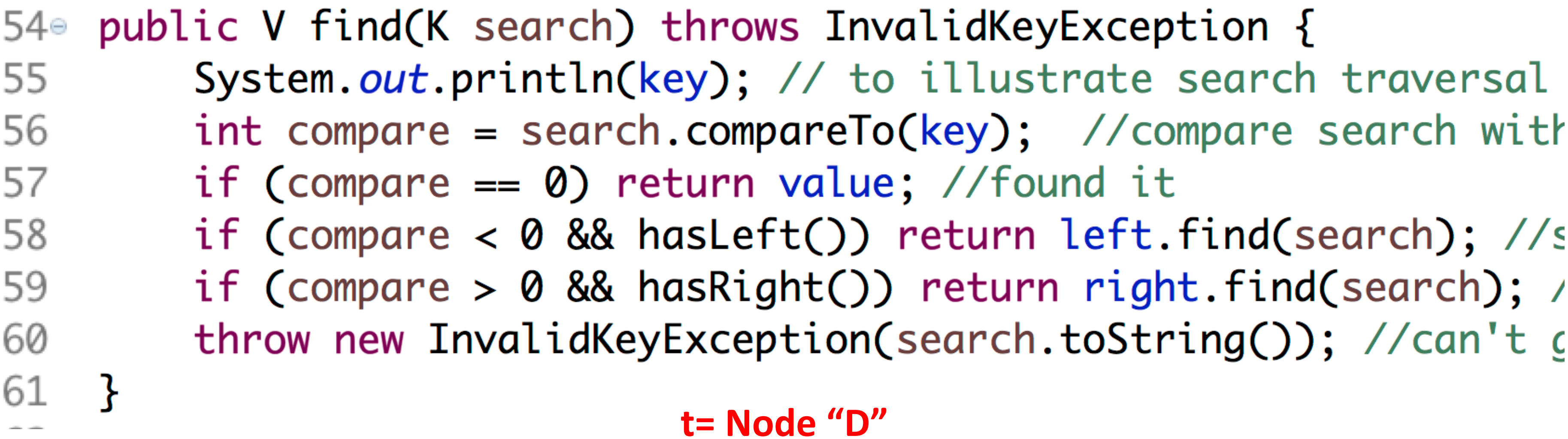
In Class declaration add “implements Comparable” so Java knows class follows interface (not shown)

- **Compare this Point with another Point using whatever metric you decide makes one bigger**
- **Return a positive integer if this Point > compared Point**
- **Return negative integer if this Point < compared Point**
- **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

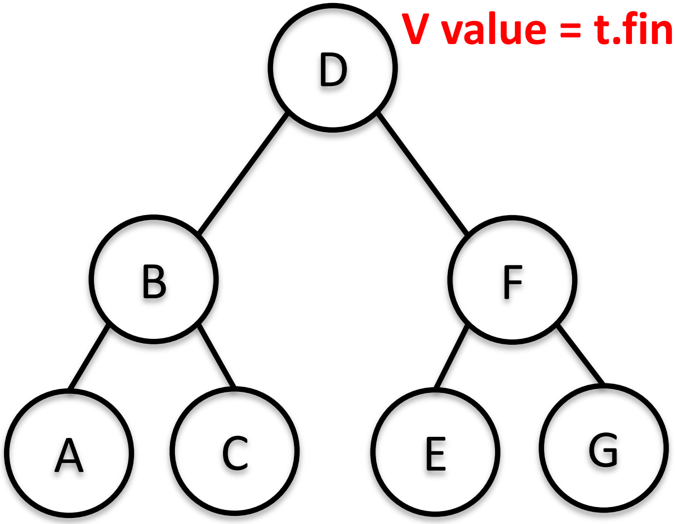
BST.java



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

t= Node “D”

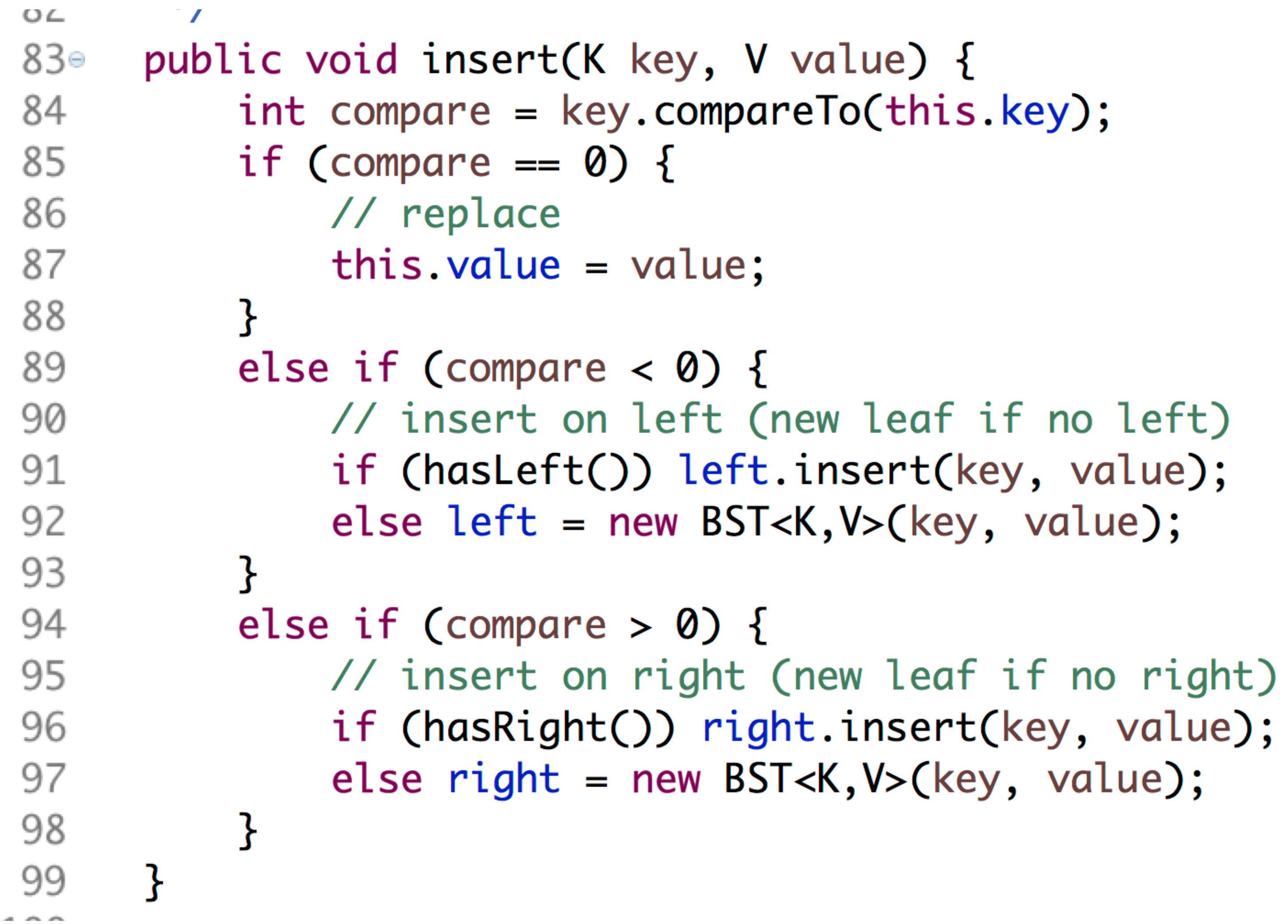
V value = t.find(“C”)



On paper run

Comparable also helps inserting new Nodes

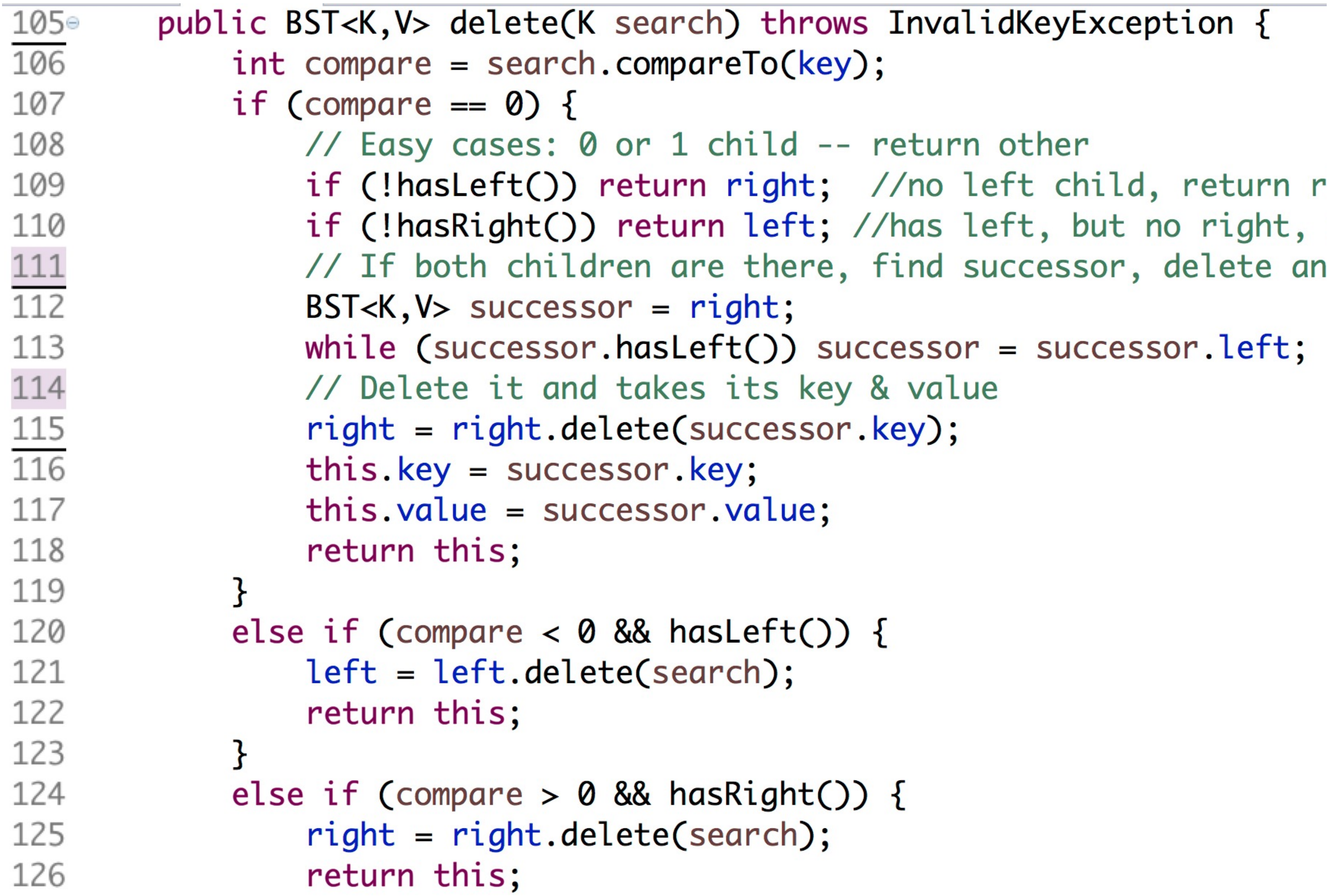
BST.java



```
     */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

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

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

Summary

- Binary search tree is very powerful for binary search and differently from arrays, BST can be easily modified
  - It has more efficient look-up than lists

Next

- Information retrieval

Additional Resources

PointWithCompareTo.java

ANNOTATED SLIDES

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

PointWithCompareTo.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

```
/**
 * Compare this blob with another blob
 * @param comparePoint point to compare to this point
 * @return   0 if same,
 *         1 if this point is higher up than comparePoint,
 *        -1 otherwise */
public int compareTo(PointWithCompareTo comparePoint) {
   if (this.y < comparePoint.getY())
     return 1; //this Point is higher up, so it's bigger
   else if (this.y > comparePoint.getY())
     return -1; //this Point is lower, so it's smaller
   else return 0; //at same height, so same
}
```

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

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

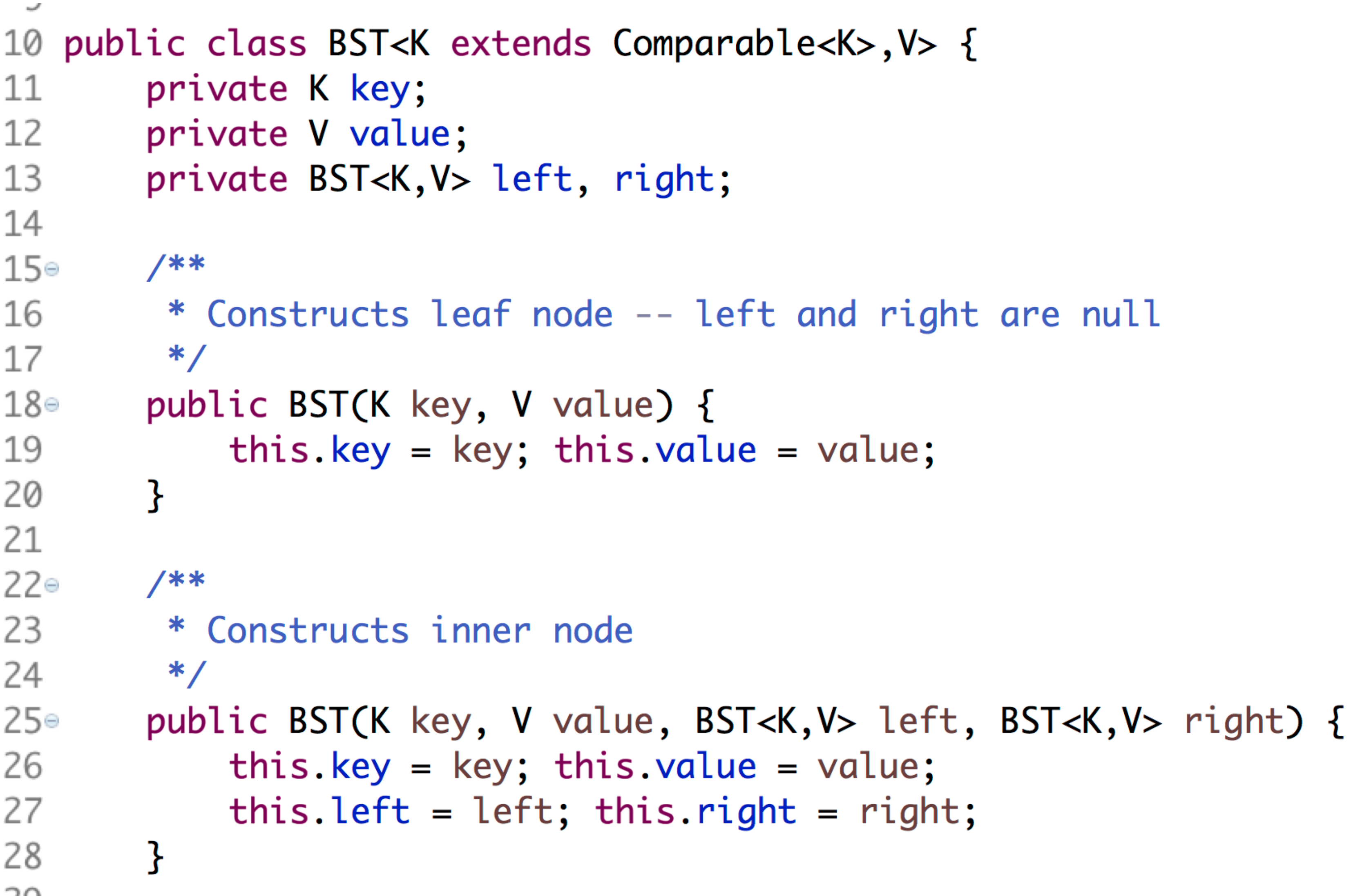
- **Compare this Point with another Point using whatever metric you decide makes one bigger**
- **Return a positive integer if this Point > compared Point**
- **Return negative integer if this Point < compared Point**
- **Return 0 if equal**

BST.java

ANNOTATED SLIDES

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

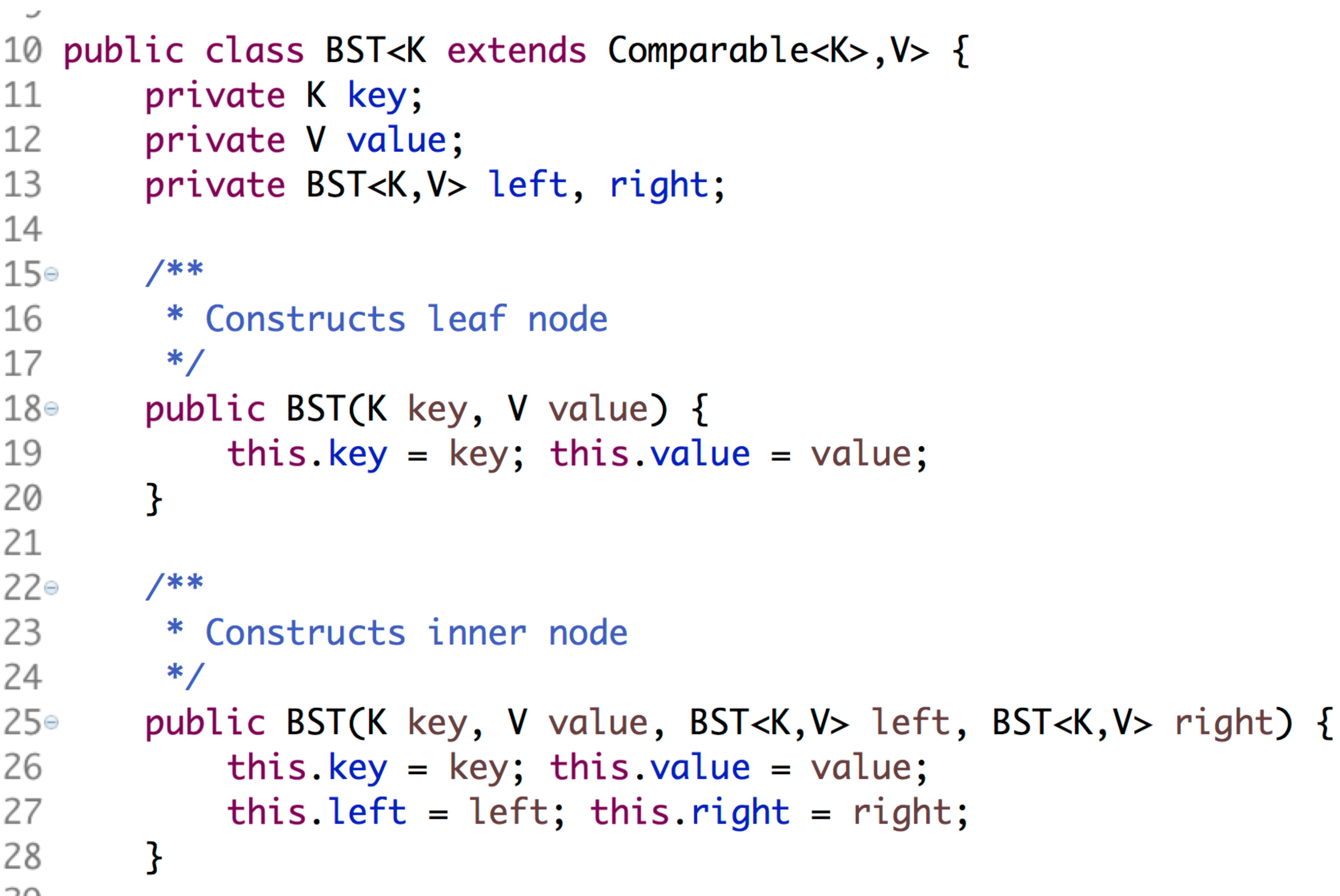
BST.java



```
public class BST<K extends Comparable<K>,V> {
    private K key;
    private V value;
    private BST<K,V> left, right;

    /**
     * Constructs leaf node -- left and right are null
     */
    public BST(K key, V value) {
        this.key = key; this.value = value;
    }

    /**
     * Constructs inner node
     */
    public BST(K key, V value, BST<K,V> left, BST<K,V> right) {
        this.key = key; this.value = value;
        this.left = left; this.right = right;
    }
```

BST Keys extend Comparable so we can evaluate generic Keys

BST.java



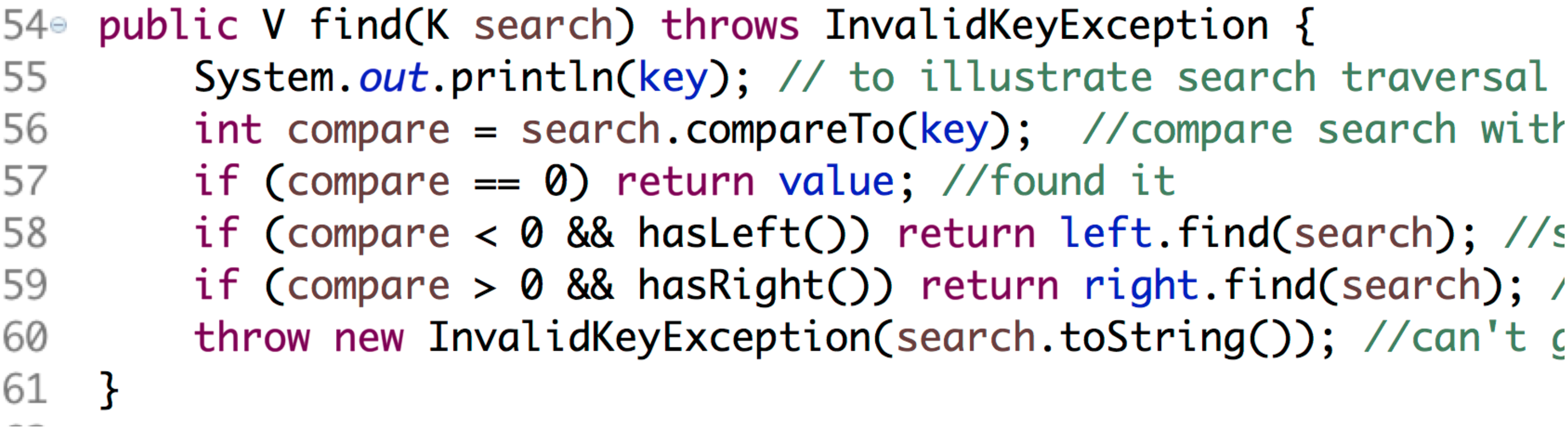
```java
public class BST<K extends Comparable<K>,V> {
    private K key;
    private V value;
    private BST<K,V> left, right;

    /**
     * Constructs leaf node
     */
    public BST(K key, V value) {
        this.key = key; this.value = value;
    }

    /**
     * Constructs inner node
     */
    public BST(K key, V value, BST<K,V> left, BST<K,V> right) {
        this.key = key; this.value = value;
        this.left = left; this.right = right;
    }
```

BST.java - find

ANNOTATED SLIDES

Using Comparable makes finding a Key in a BST easy

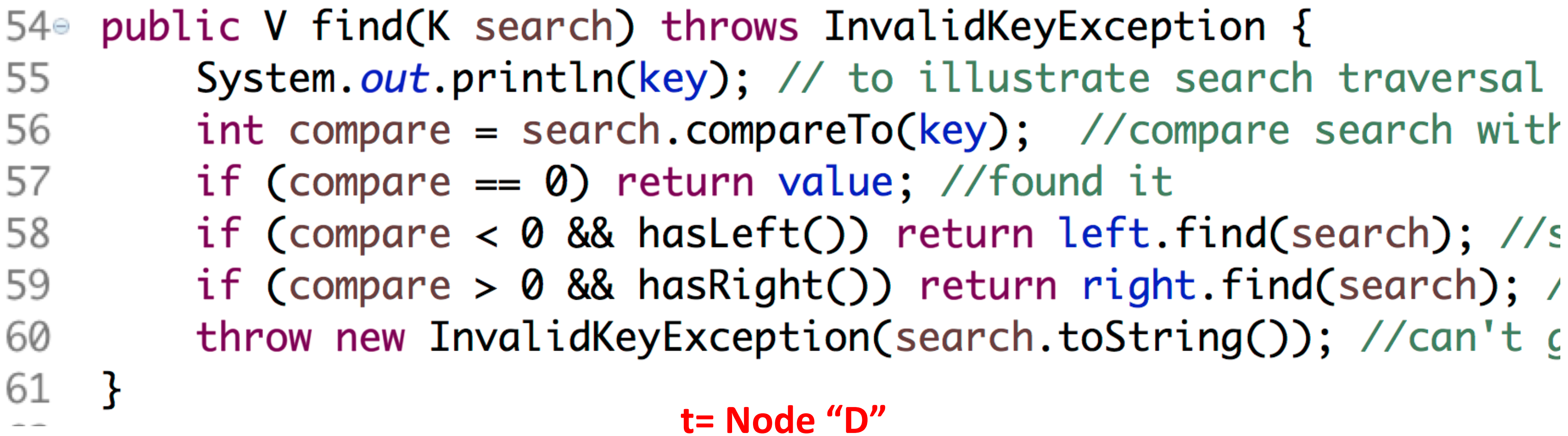
BST.java



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

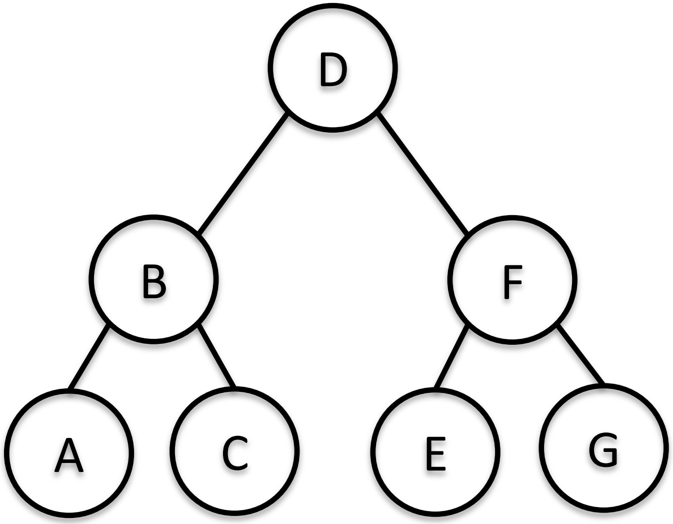
Using Comparable makes finding a Key in a BST easy

BST.java



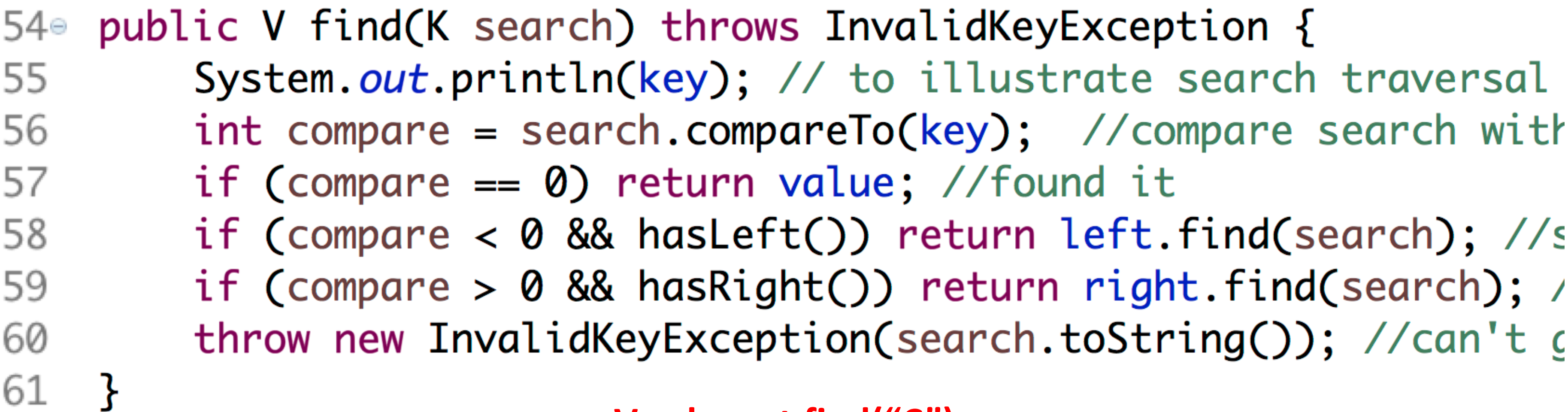
```
public V find(K search) throws InvalidKeyException {
        System.out.println(key); // to illustrate search traversal
        int compare = search.compareTo(key);  //compare search with
        if (compare == 0) return value; //found it
        if (compare < 0 && hasLeft()) return left.find(search);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

t= Node "D"



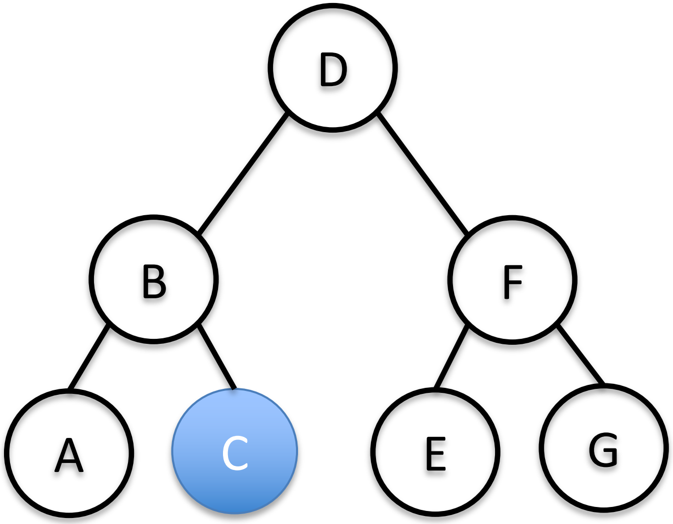
Using Comparable makes finding a Key in a BST easy

BST.java



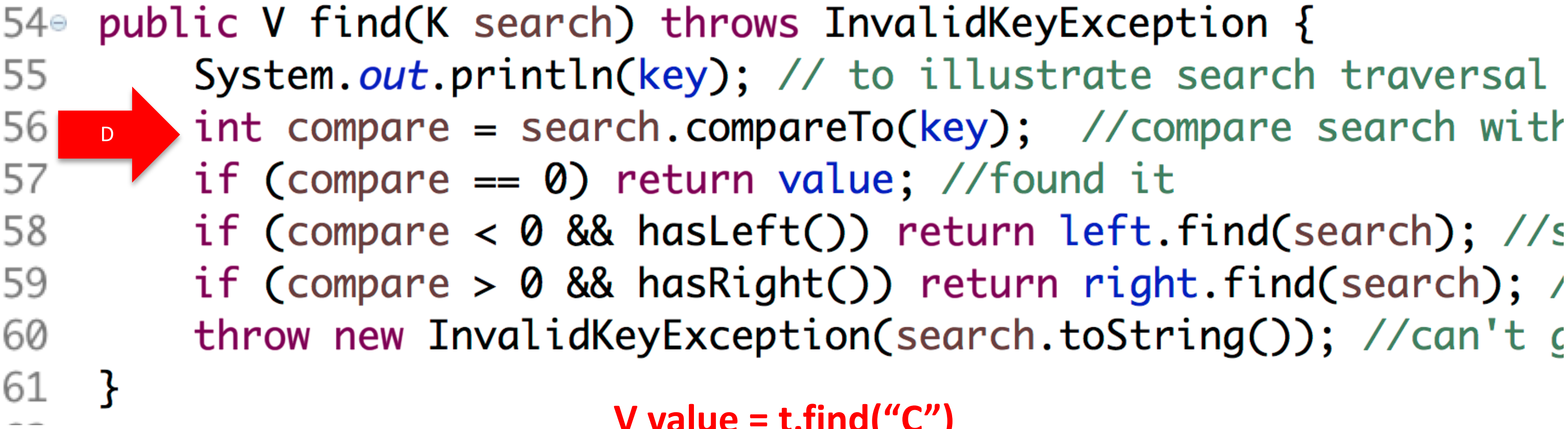
```
public V find(K search) throws InvalidKeyException {
    System.out.println(key); // to illustrate search traversal
    int compare = search.compareTo(key);  //compare search with
    if (compare == 0) return value; //found it
    if (compare < 0 && hasLeft()) return left.find(search); //s
    if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

V value = t.find(“C”)



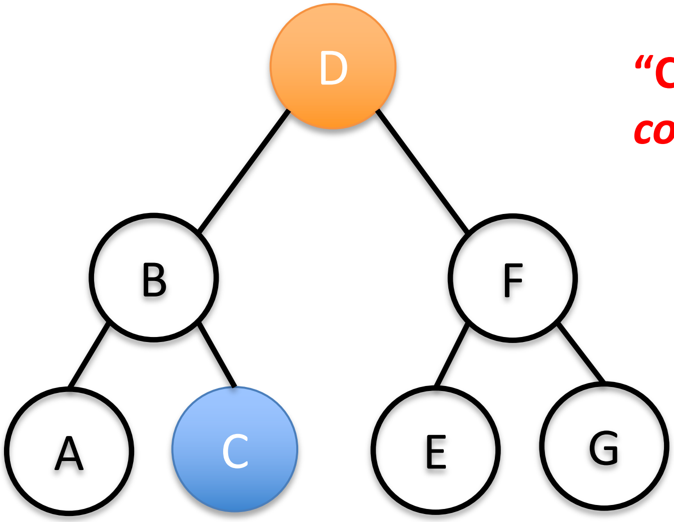
Using Comparable makes finding a Key in a BST easy

BST.java



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

V value = t.find(“C”)

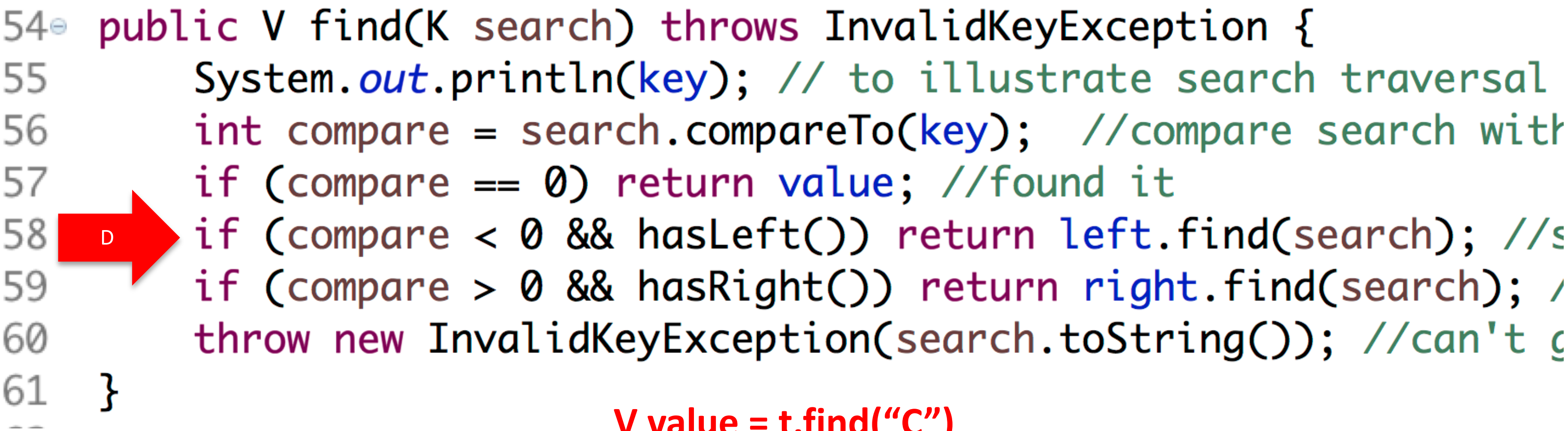


“C” < “D”

compare = -1

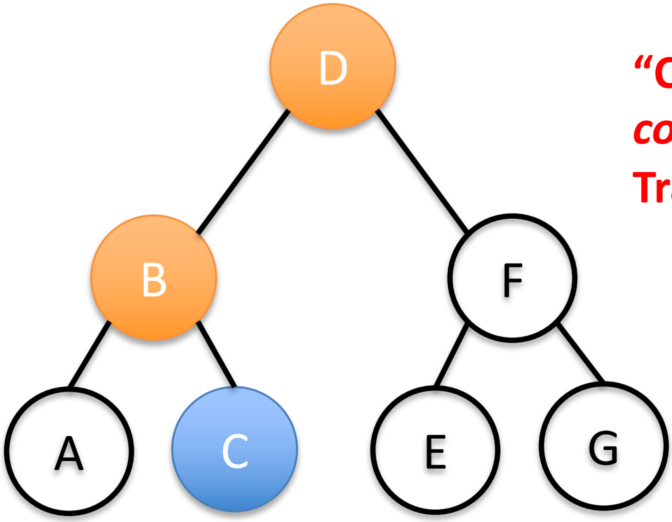
Using Comparable makes finding a Key in a BST easy

BST.java



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

V value = t.find("C")



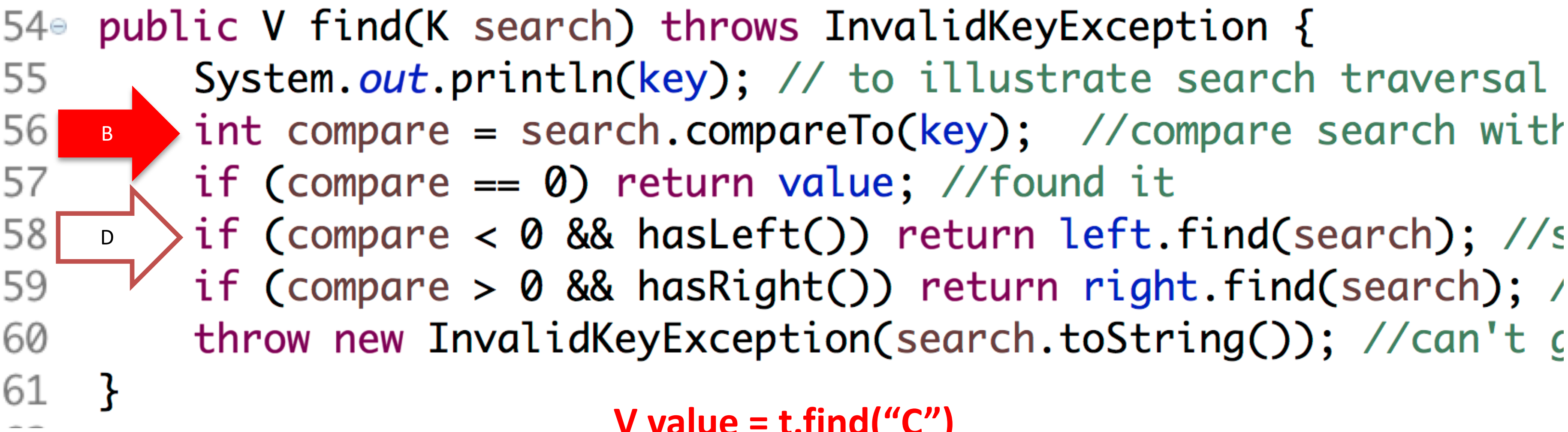
"C" < "D"

***compare* = -1**

Traverse left

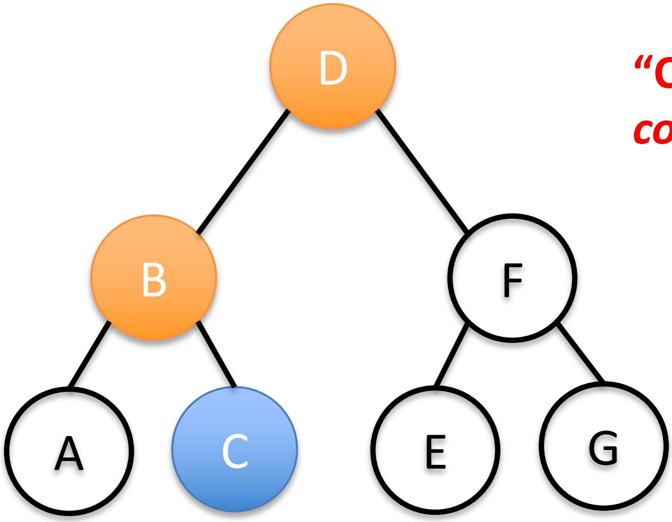
Using Comparable makes finding a Key in a BST easy

BST.java



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

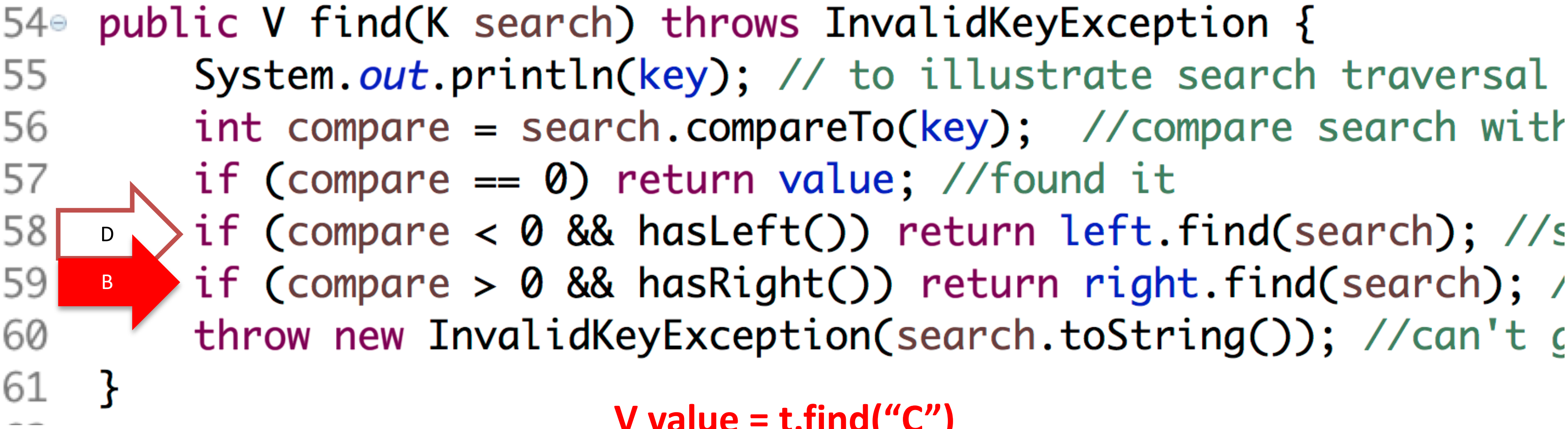
V value = t.find("C")



"C" > "B"
compare = 1

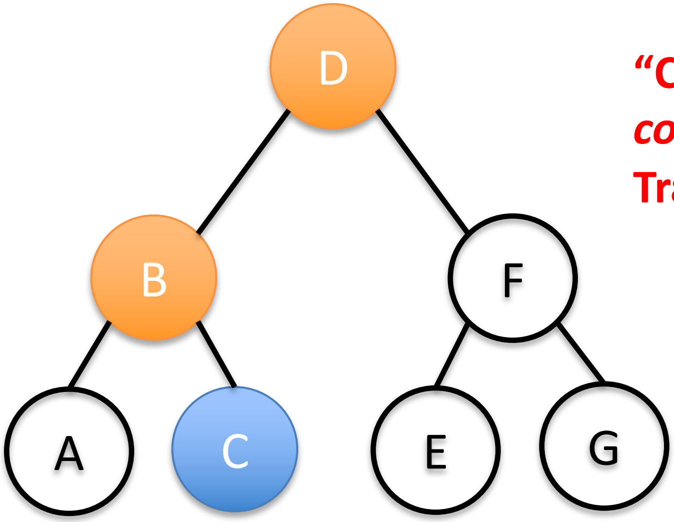
Using Comparable makes finding a Key in a BST easy

BST.java



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

V value = t.find("C")



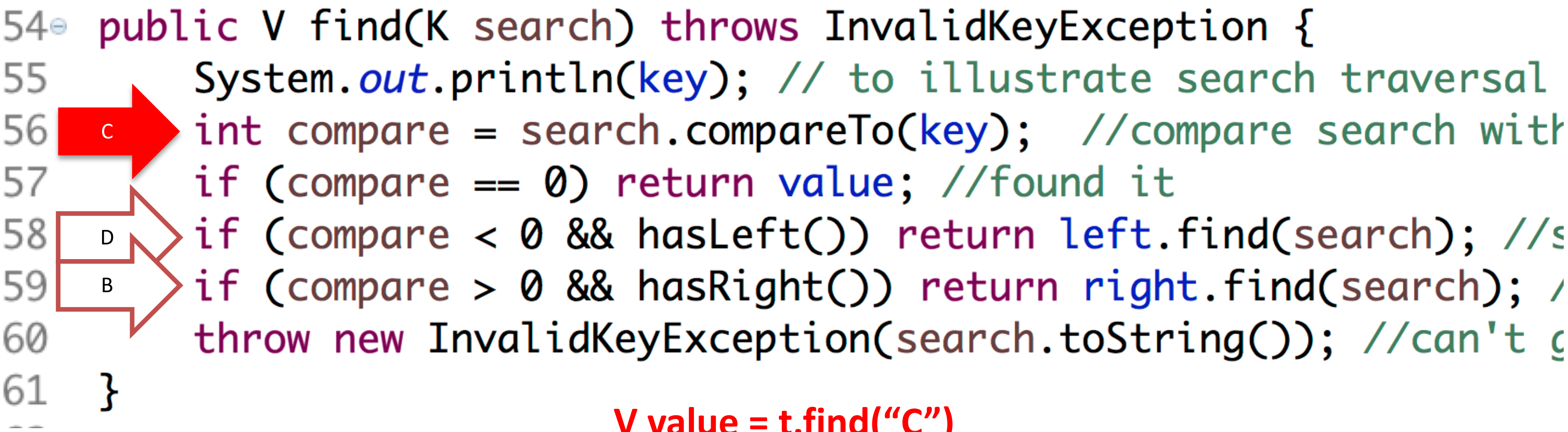
"C" > "B"

***compare* = 1**

Traverse right

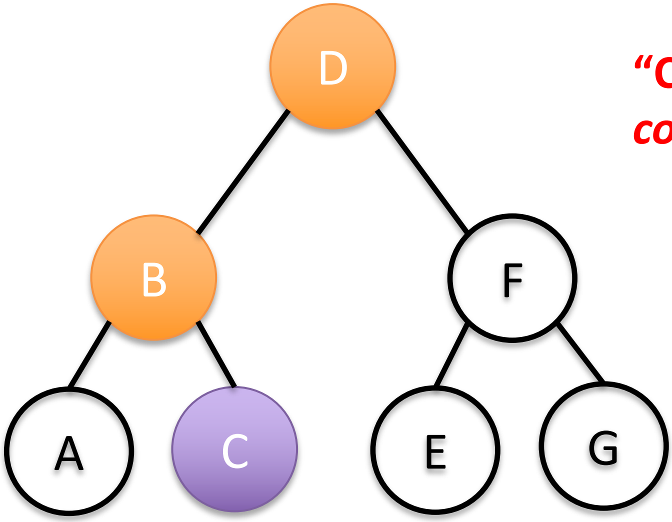
Using Comparable makes finding a Key in a BST easy

BST.java



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

V value = t.find("C")

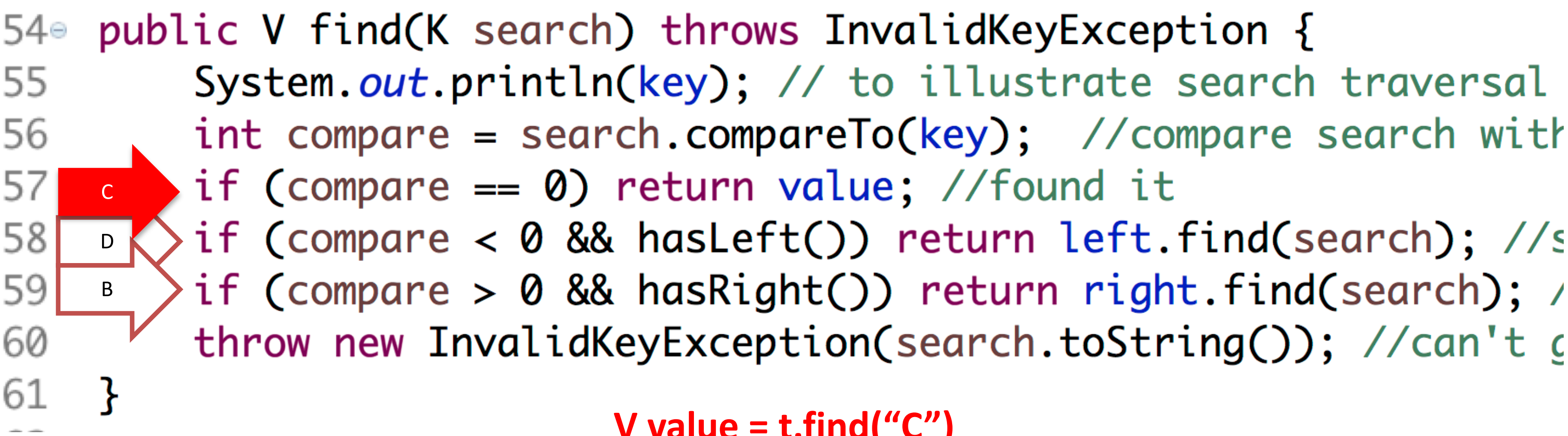


"C" = "C"

***compare* = 0**

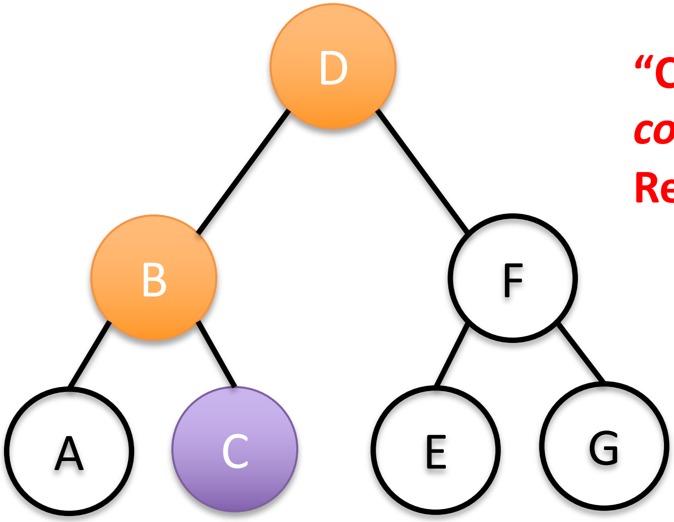
Using Comparable makes finding a Key in a BST easy

BST.java



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

V value = t.find(“C”)



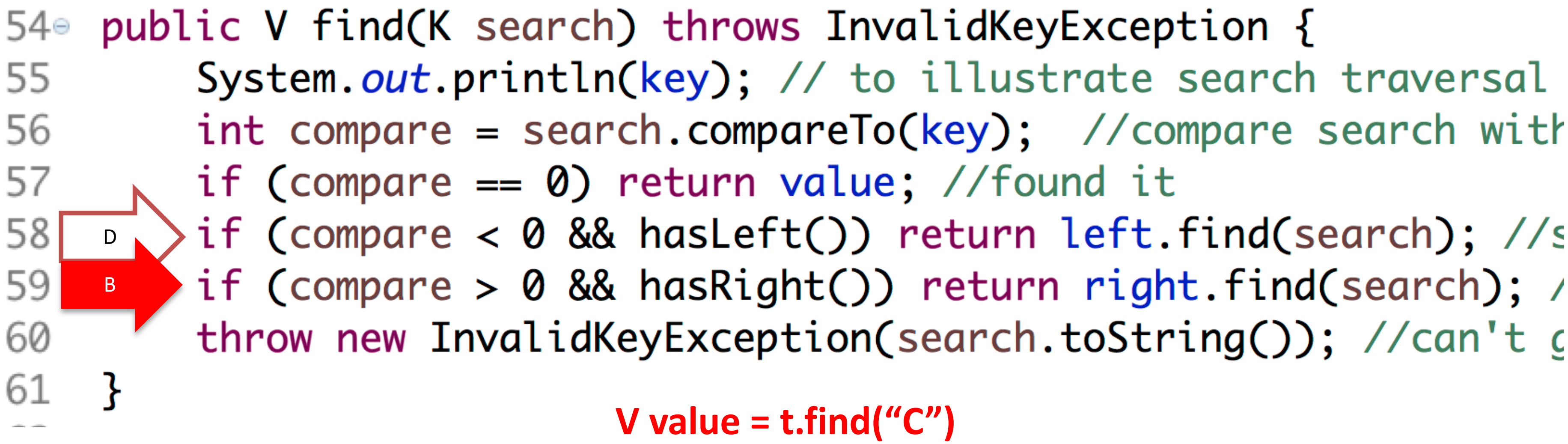
“C” = “C”

***compare* = 0**

Return Value of node “C”

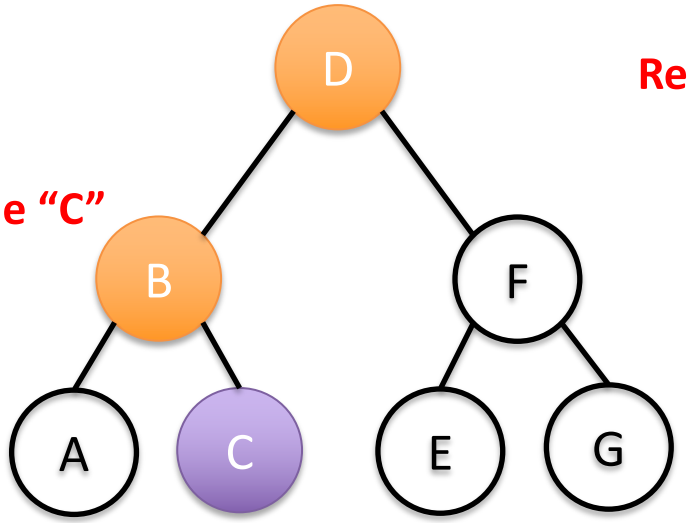
Using Comparable makes finding a Key in a BST easy

BST.java



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

V value = t.find("C")

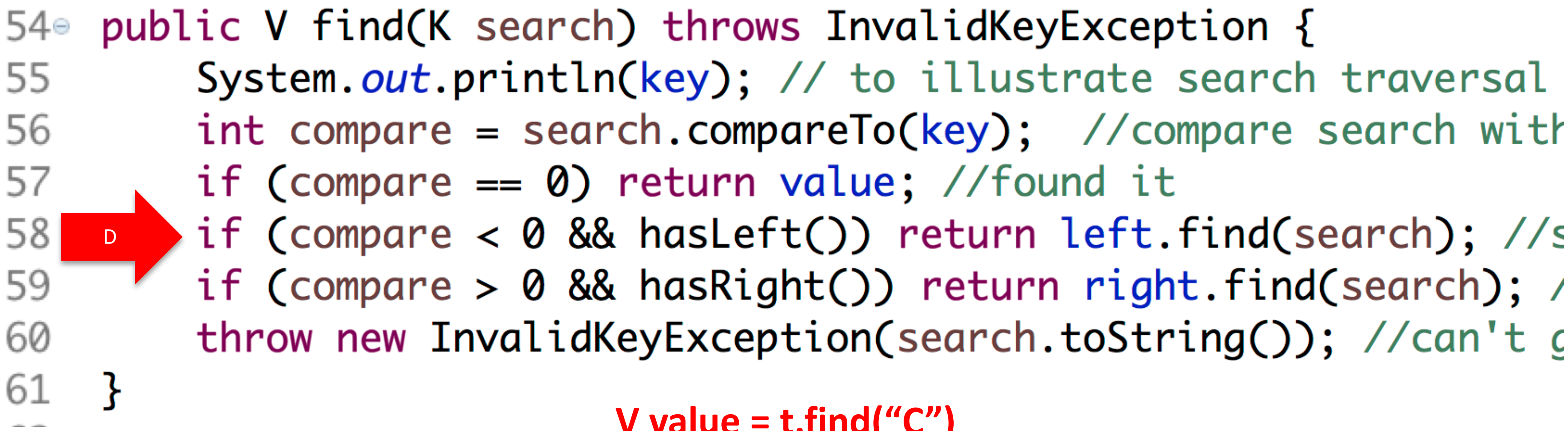


Return Value of node C

Value of node "C"

Using Comparable makes finding a Key in a BST easy

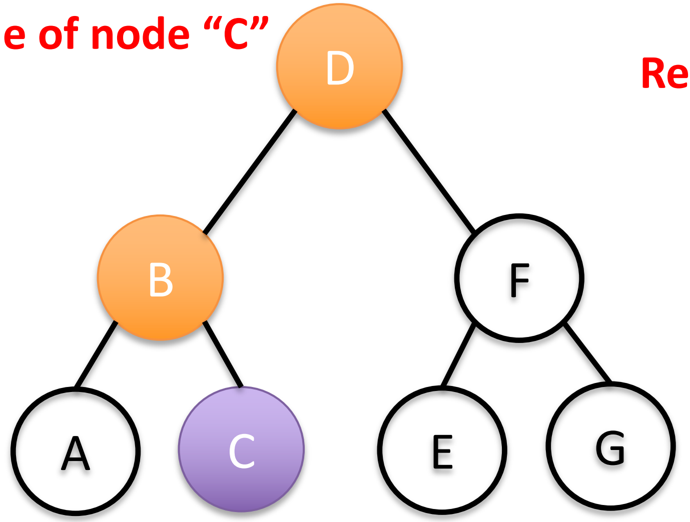
BST.java



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

V value = t.find("C")

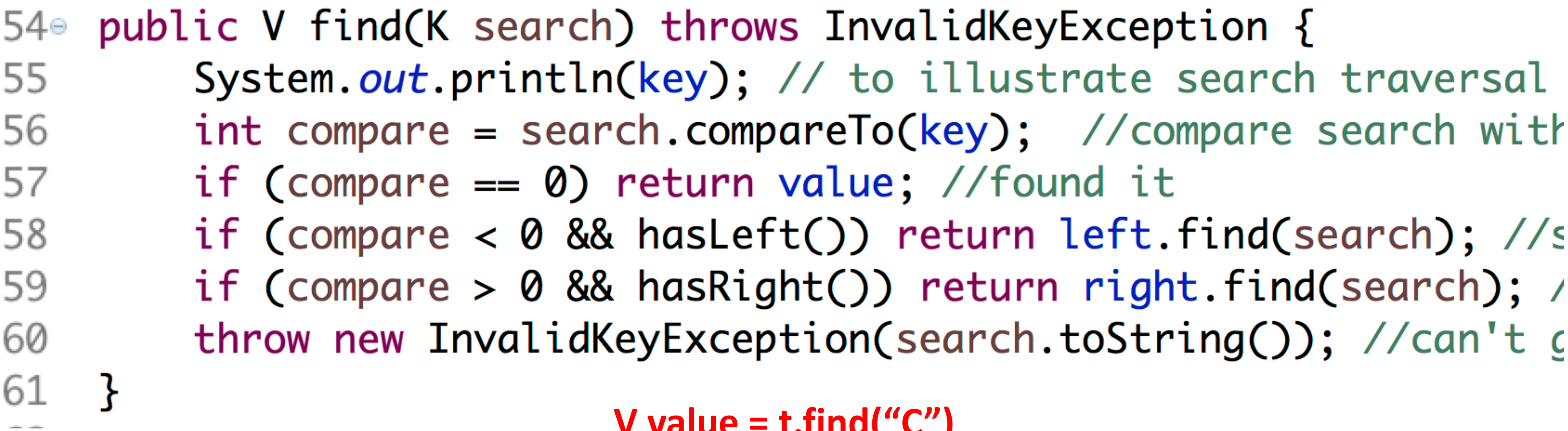
Value of node "C"



Return Value of node C

Using Comparable makes finding a Key in a BST easy

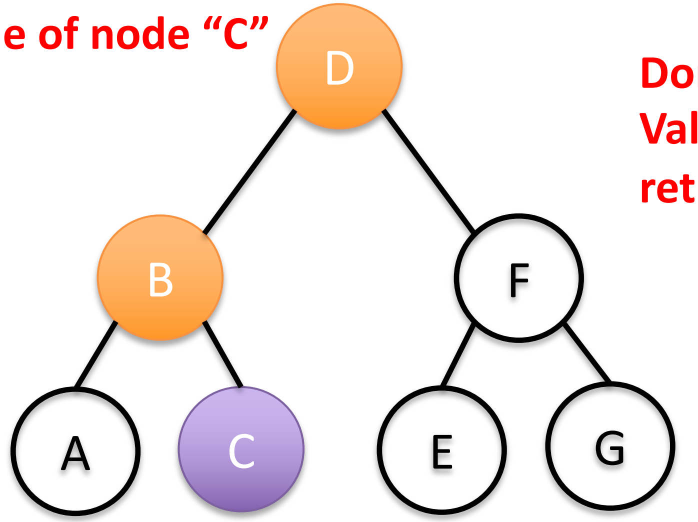
BST.java



```java
public V find(K search) throws InvalidKeyException {
    System.out.println(key); // to illustrate search traversal
    int compare = search.compareTo(key);  //compare search wit
    if (compare == 0) return value; //found it
    if (compare < 0 && hasLeft()) return left.find(search); //s
    if (compare > 0 && hasRight()) return right.find(search); /
    throw new InvalidKeyException(search.toString()); //can't
}
```

V value = t.find("C")

Value of node "C"



Done
Value of node "C"
returned

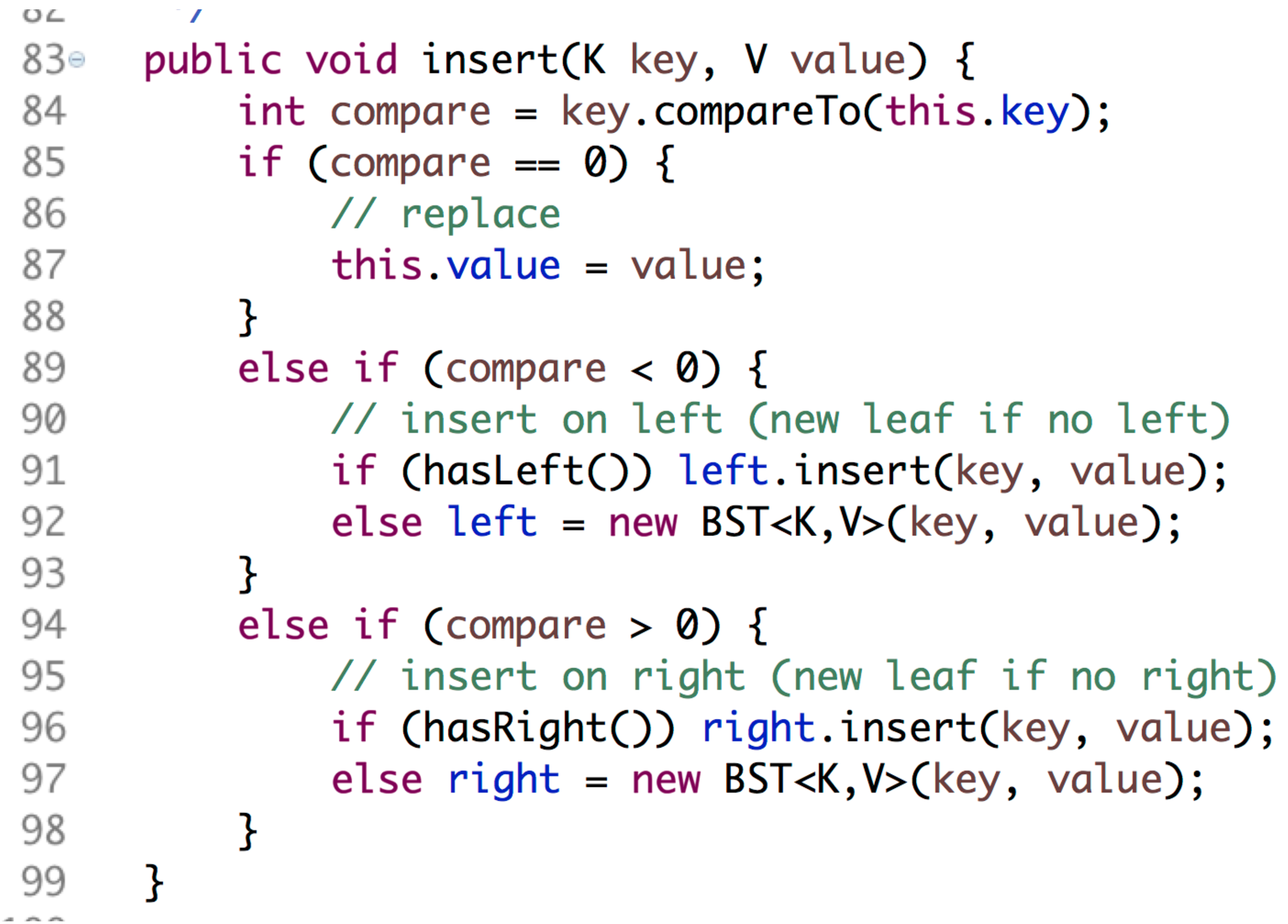
BST.java - insert

ANNOTATED SLIDES

Comparable also helps inserting new Nodes

BST.java

Inserting new K *key* and V *value*



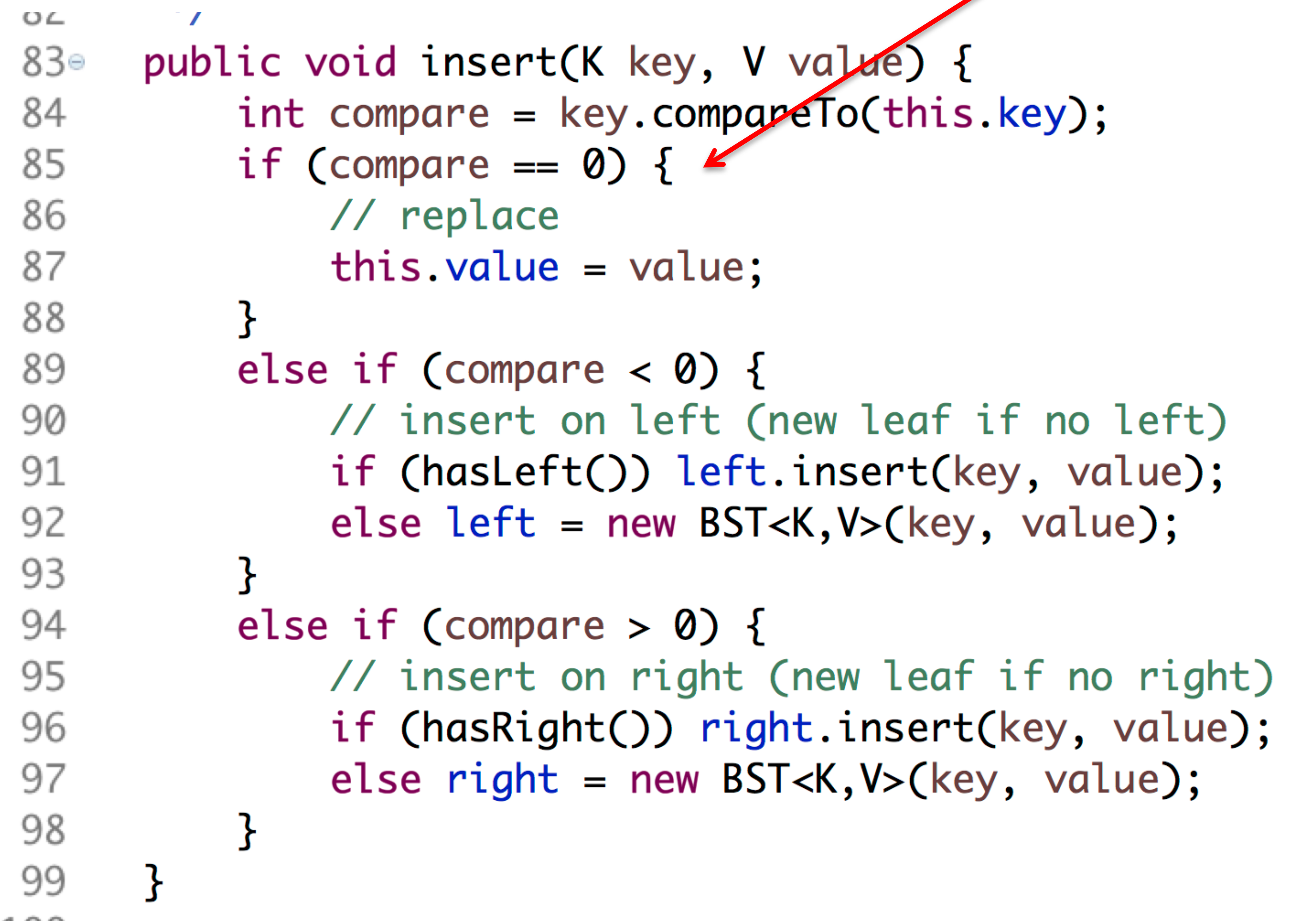
```
     */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

Comparable also helps inserting new Nodes

BST.java

Inserting new K *key* and V *value*

- **If find *key*, replace it's *value***



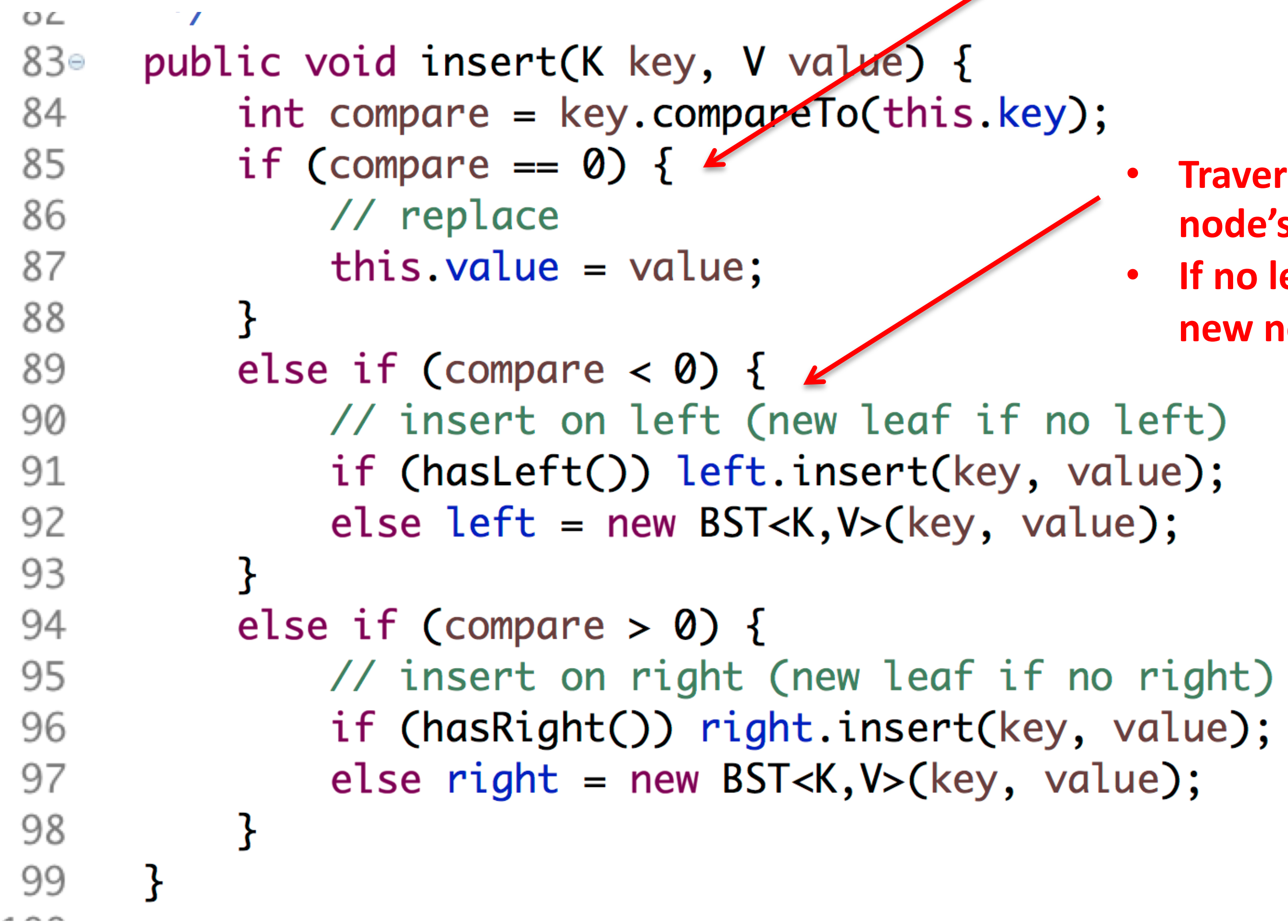
```
     */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

Comparable also helps inserting new Nodes

BST.java

Inserting new K *key* and V *value*

- If find *key*, replace it's *value*

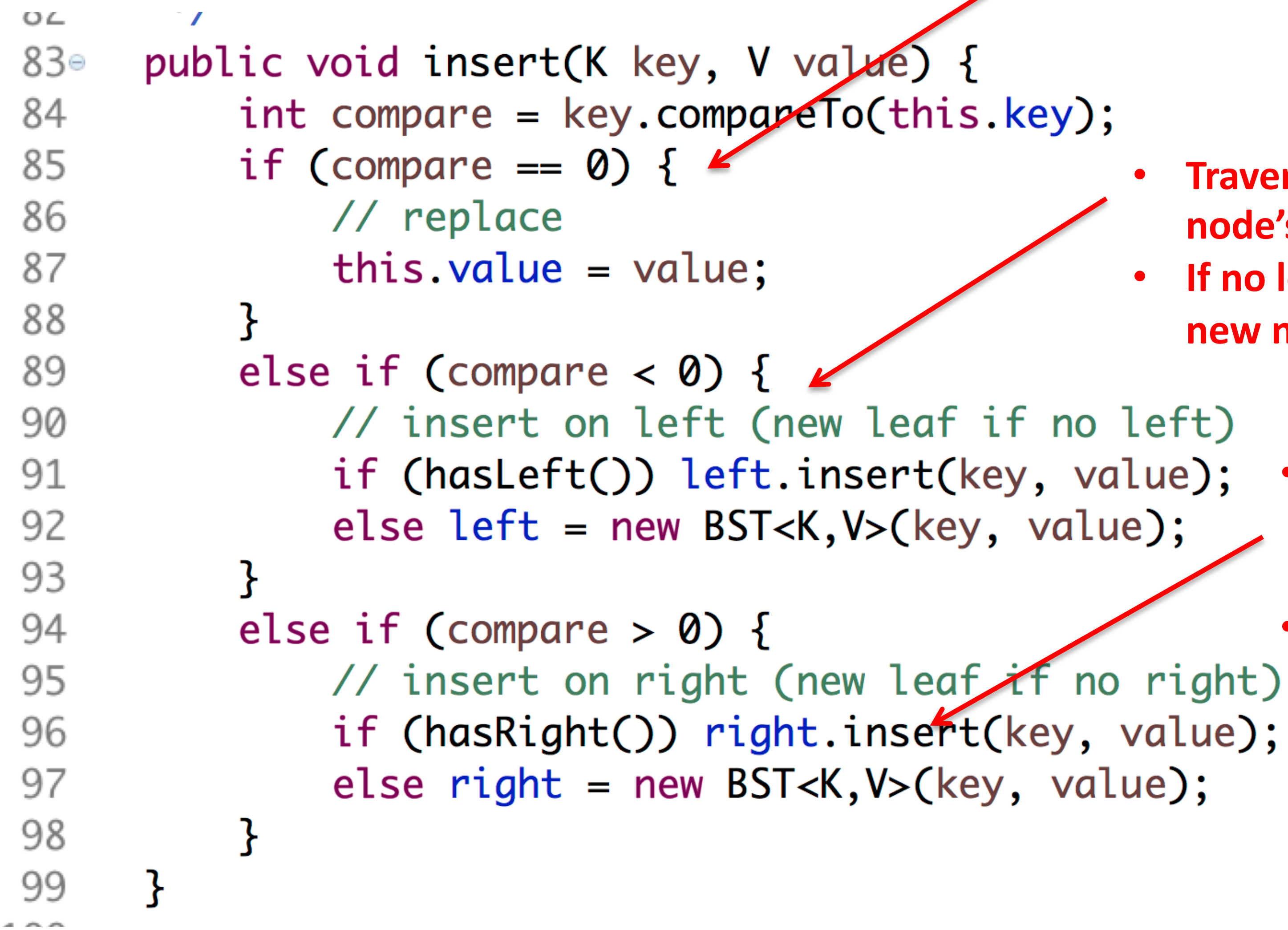


```java
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, 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



```
     */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

Inserting new K *key* and V *value*

- **If find *key*, replace it's *value***

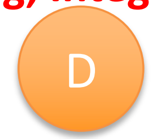
- **Traverse left if *key* < this node's *key***
- **If no left child, create a new node as the left child**

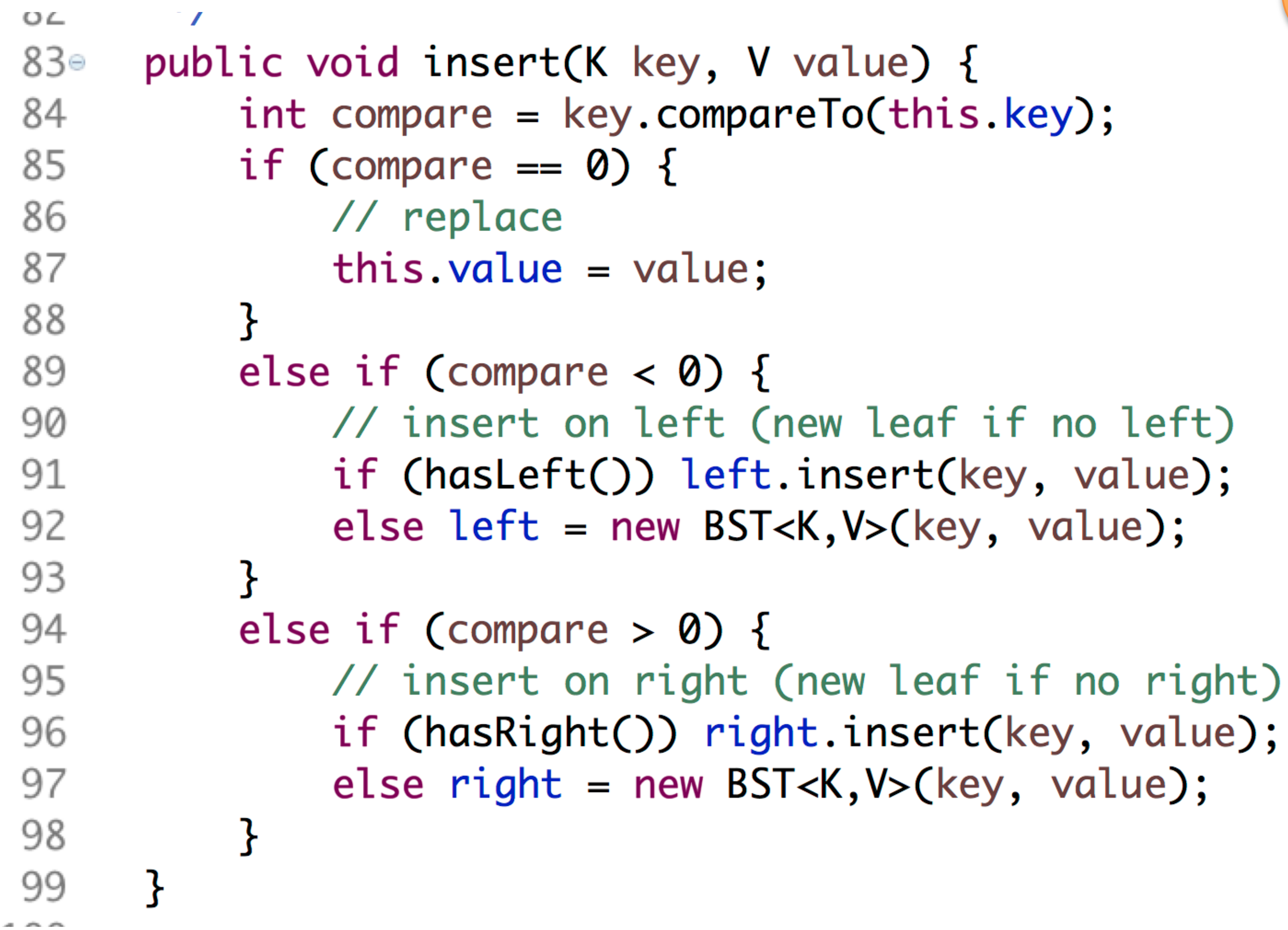
- **Traverse right if *key* > this node's *key***
- **If no right child, create a new Node as the right child**

Comparable also helps inserting new Nodes

BST.java

BST<String, Integer> t = new BST<String, Integer>(“D”,v_1);



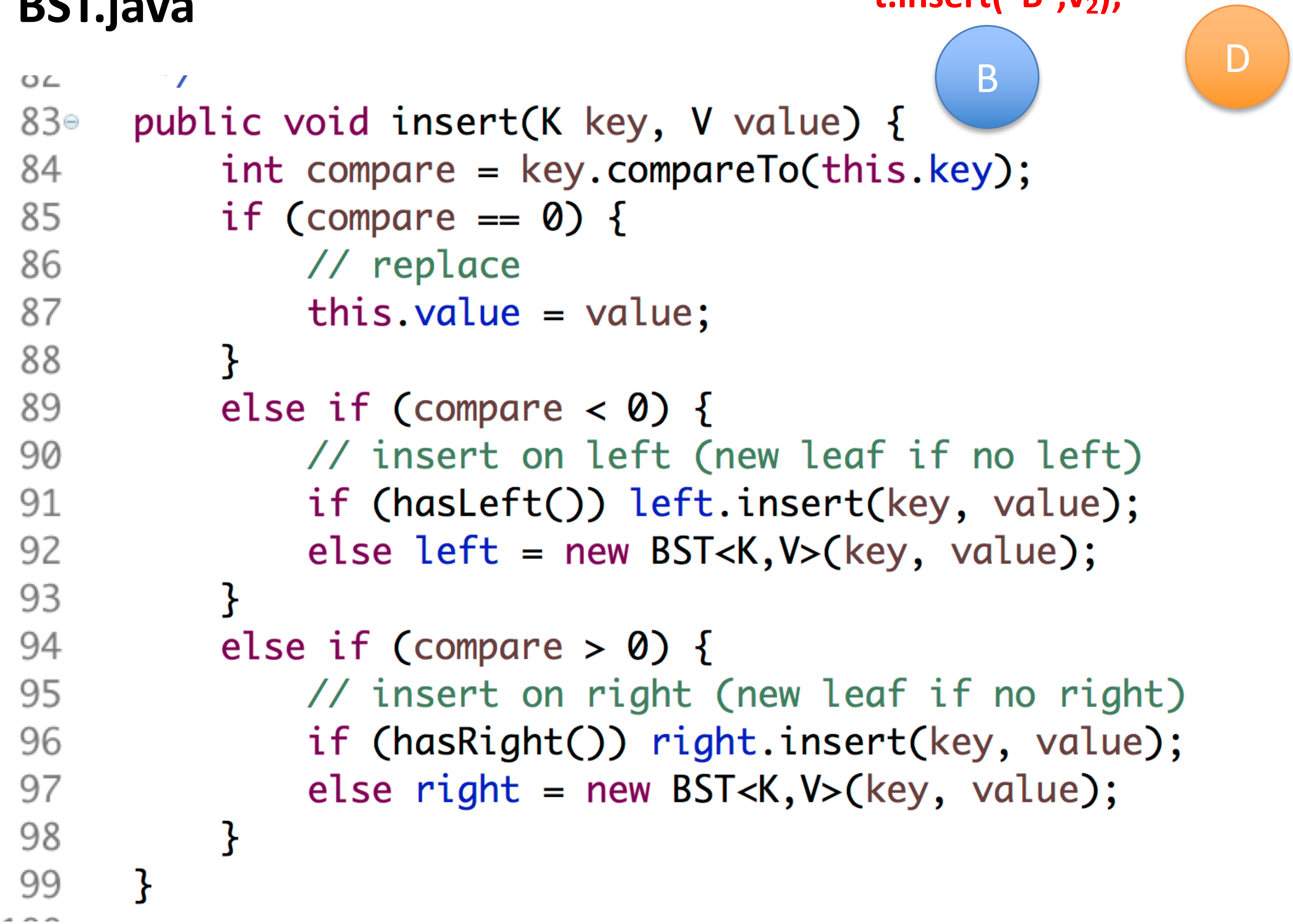


```
    */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

Comparable also helps inserting new Nodes

BST.java

t.insert(“B”,v_2);

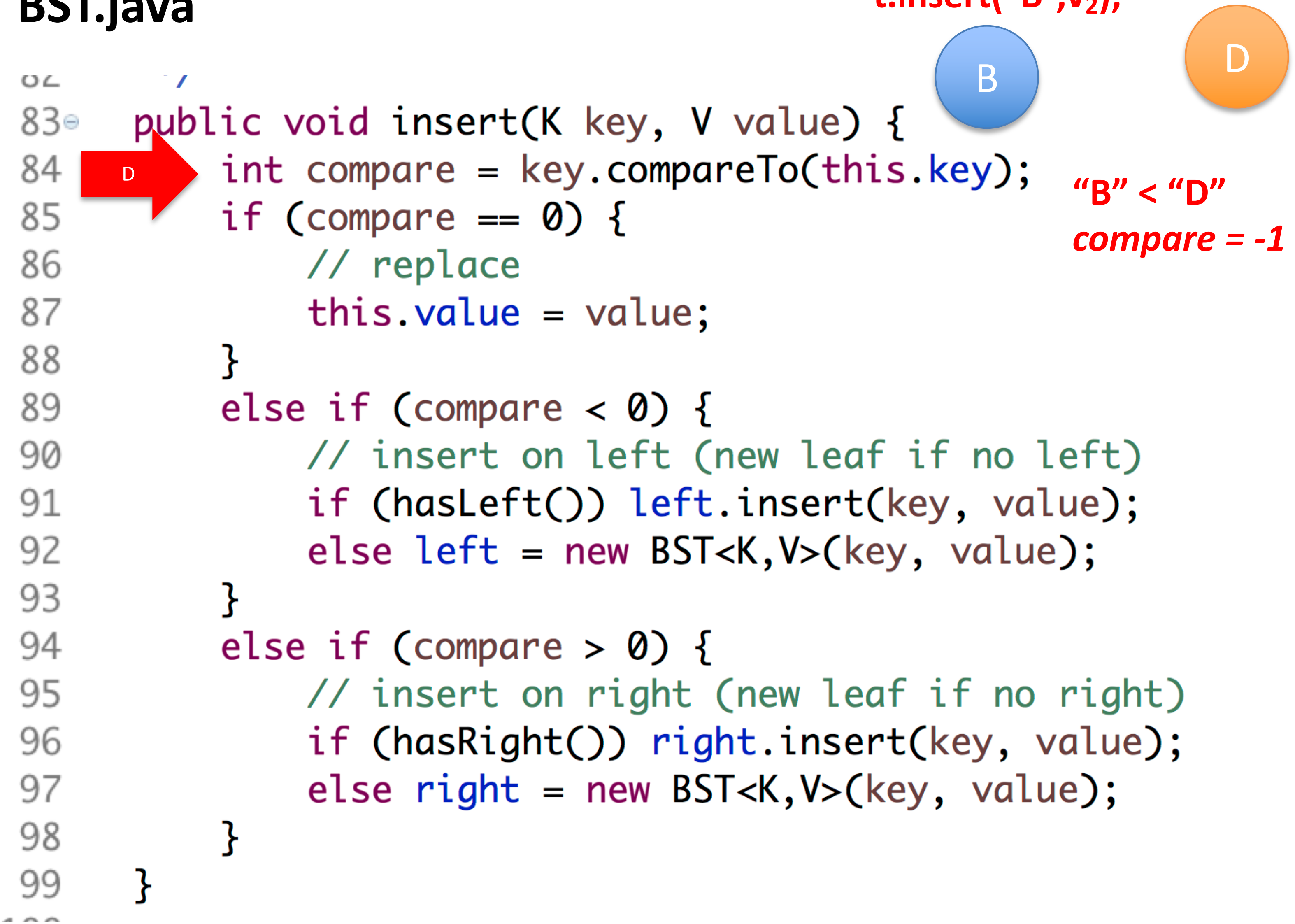


```
public void insert(K key, V value) {
    int compare = key.compareTo(this.key);
    if (compare == 0) {
        // replace
        this.value = value;
    }
    else if (compare < 0) {
        // insert on left (new leaf if no left)
        if (hasLeft()) left.insert(key, value);
        else left = new BST<K,V>(key, value);
    }
    else if (compare > 0) {
        // insert on right (new leaf if no right)
        if (hasRight()) right.insert(key, value);
        else right = new BST<K,V>(key, value);
    }
}
```

Comparable also helps inserting new Nodes

BST.java

t.insert(“B”,v_2);



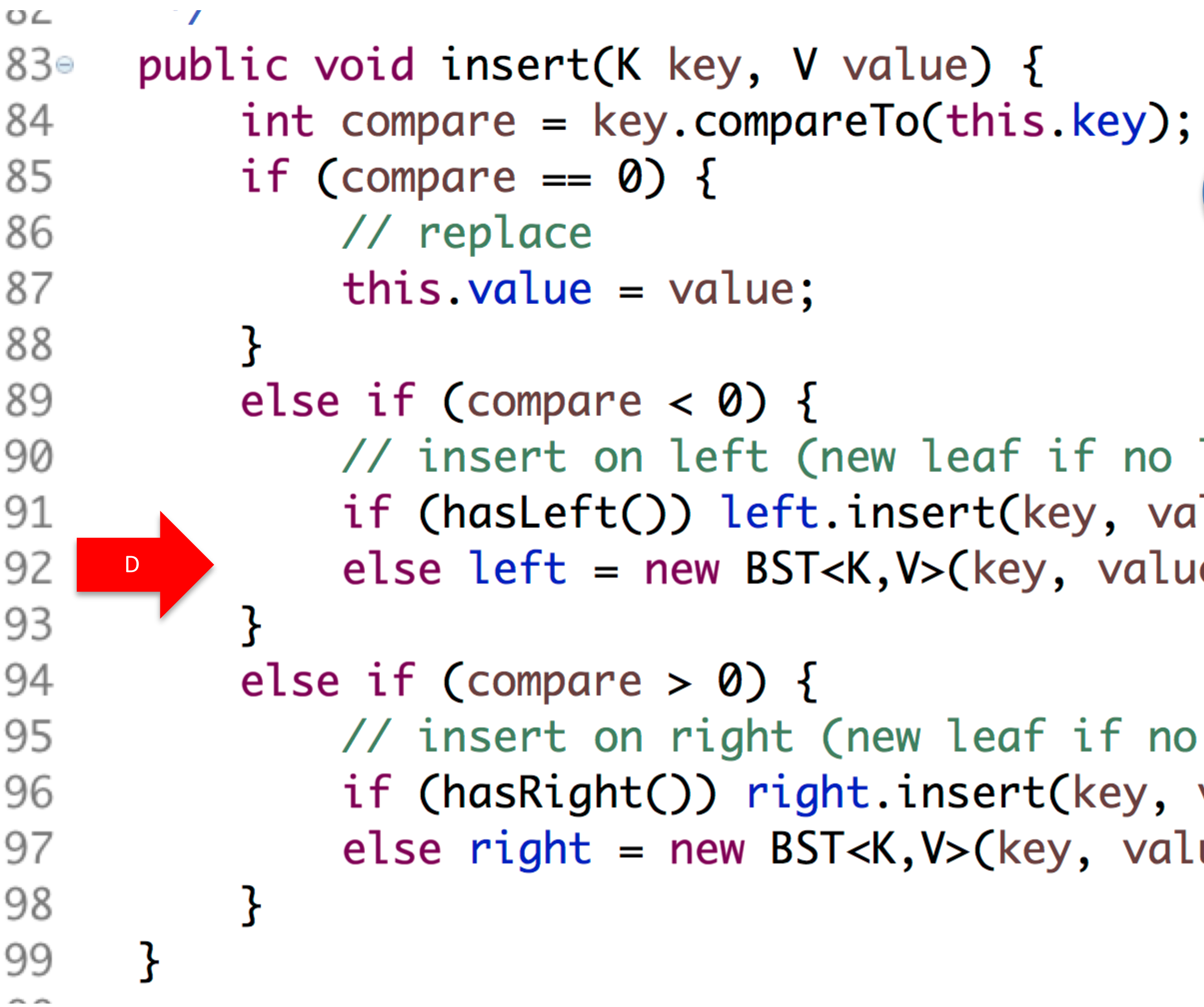
```java
       */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

“B” < “D”

compare = -1

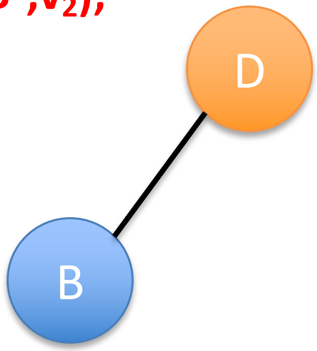
Comparable also helps inserting new Nodes

BST.java



```
     */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

t.insert(“B”,v_2);



“B” < “D”

compare = -1

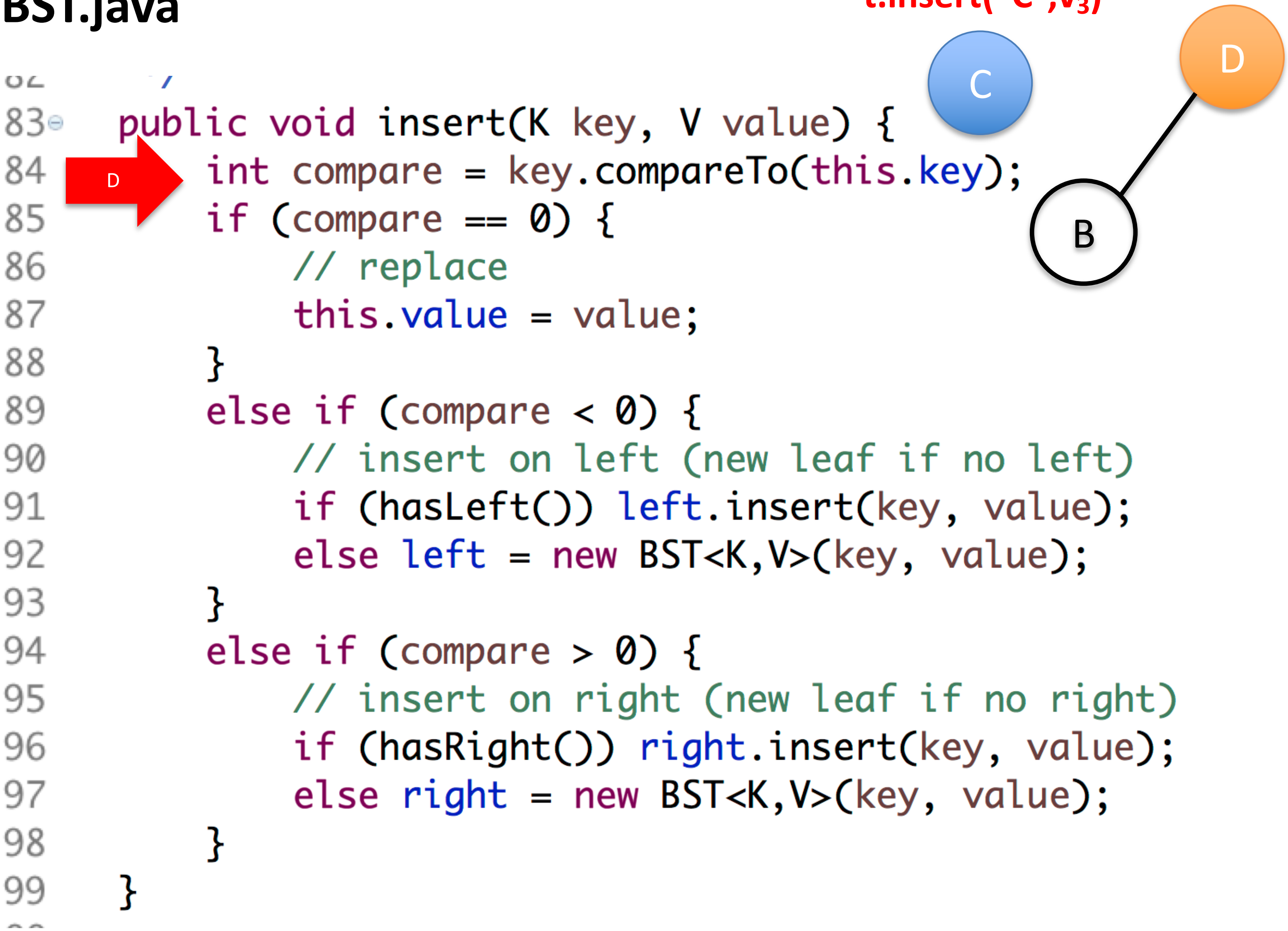
No left child

Add “B” as left

Comparable also helps inserting new Nodes

BST.java

t.insert(“C”,v_3)



```java
public void insert(K key, V value) {
    int compare = key.compareTo(this.key);
    if (compare == 0) {
        // replace
        this.value = value;
    }
    else if (compare < 0) {
        // insert on left (new leaf if no left)
        if (hasLeft()) left.insert(key, value);
        else left = new BST<K,V>(key, value);
    }
    else if (compare > 0) {
        // insert on right (new leaf if no right)
        if (hasRight()) right.insert(key, value);
        else right = new BST<K,V>(key, value);
    }
}
```

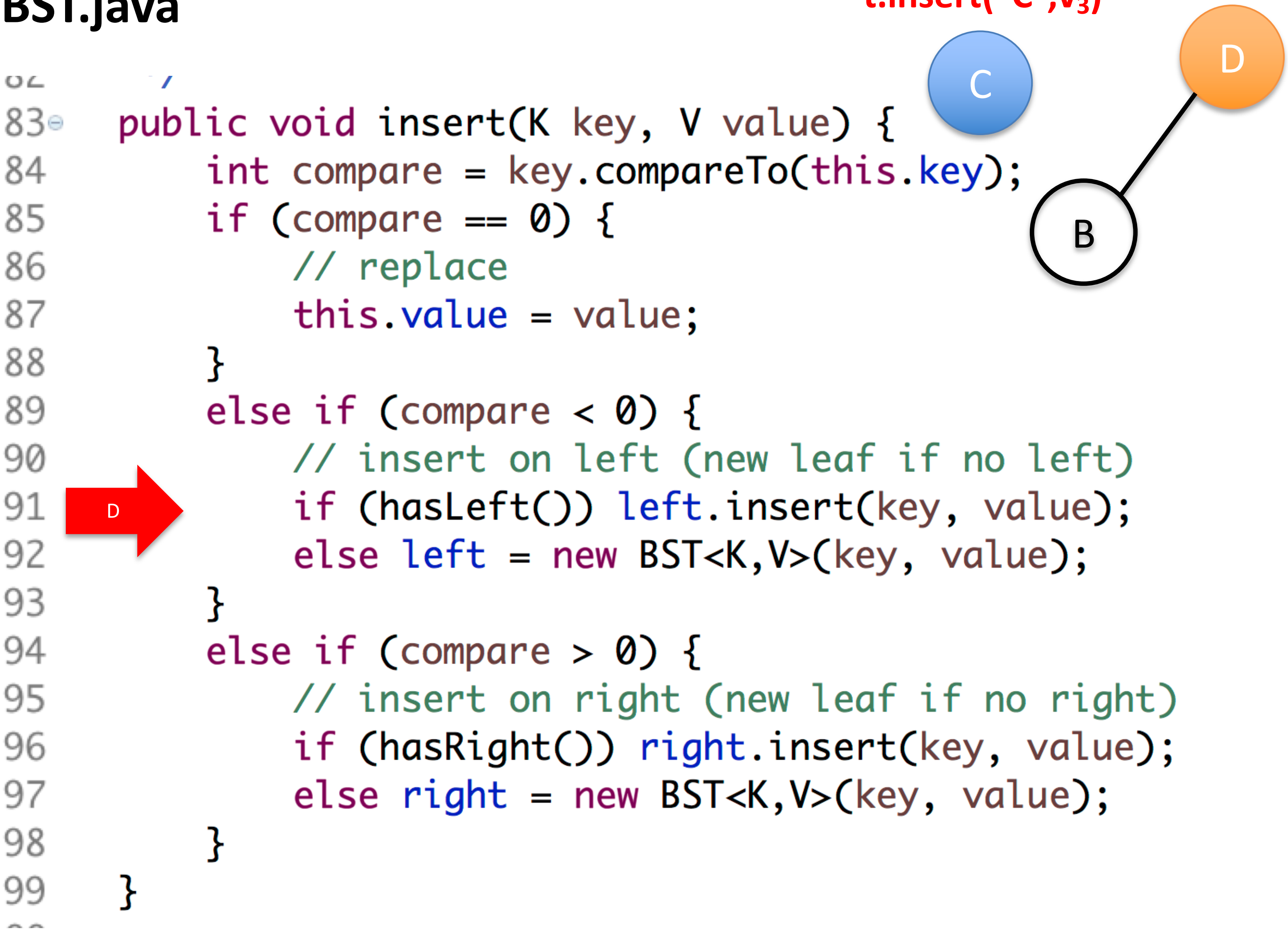
“C” < “D”

compare = -1

Comparable also helps inserting new Nodes

BST.java

t.insert(“C”,v_3)



```
     */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

“C” < “D”

compare = -1

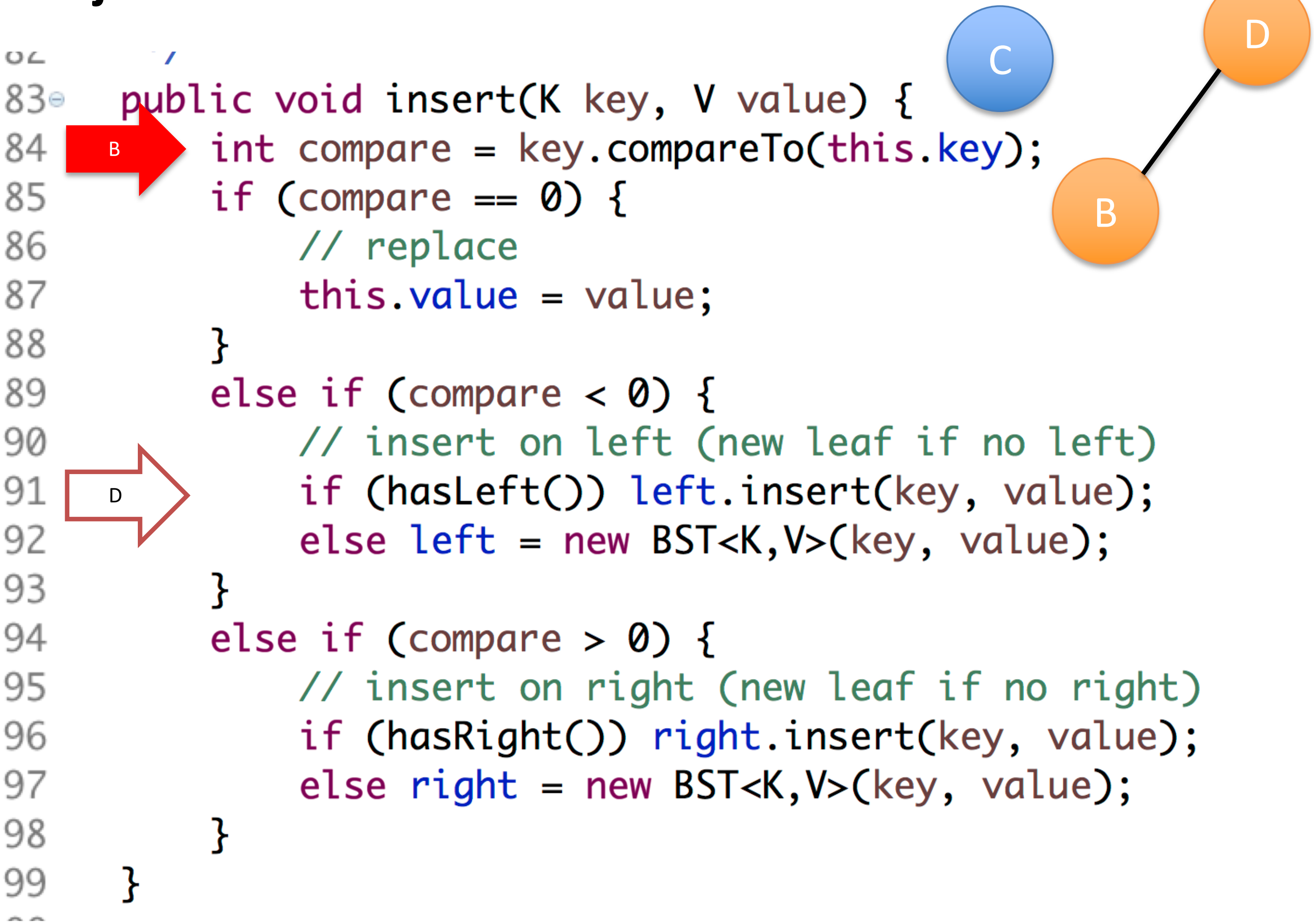
Has left

traverse left

Comparable also helps inserting new Nodes

BST.java

t.insert(“C”,v_3)



```java
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

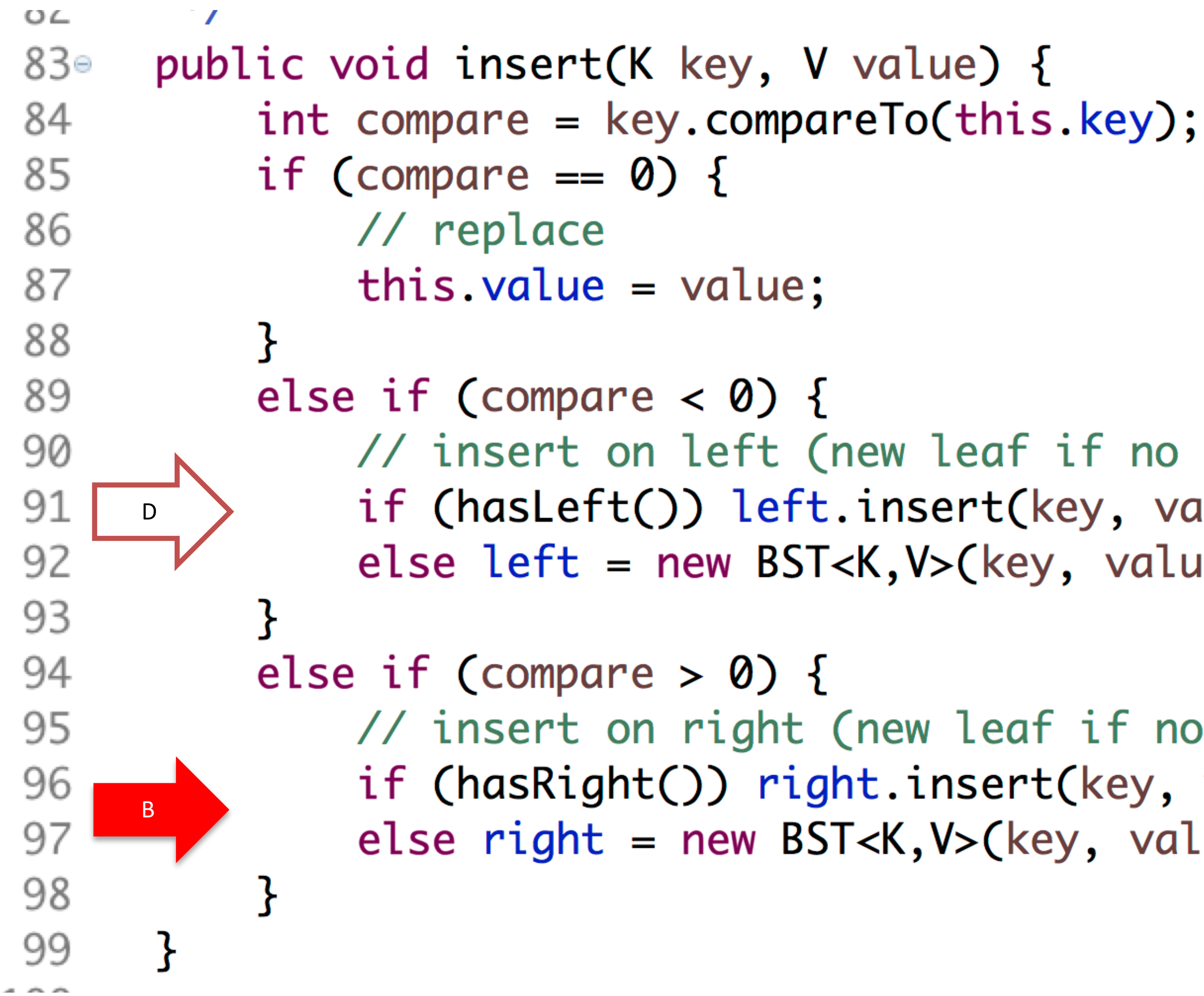
“C” > “B”

compare = 1

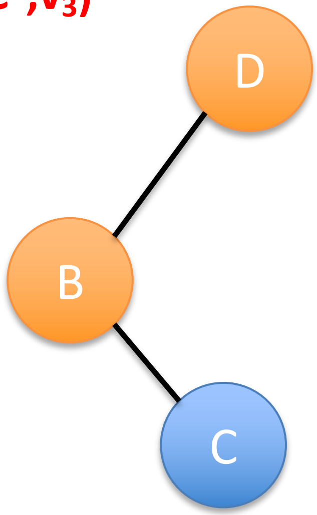
Comparable also helps inserting new Nodes

BST.java

t.insert(“C”,v_3)



```
     */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```



“C” > “B”

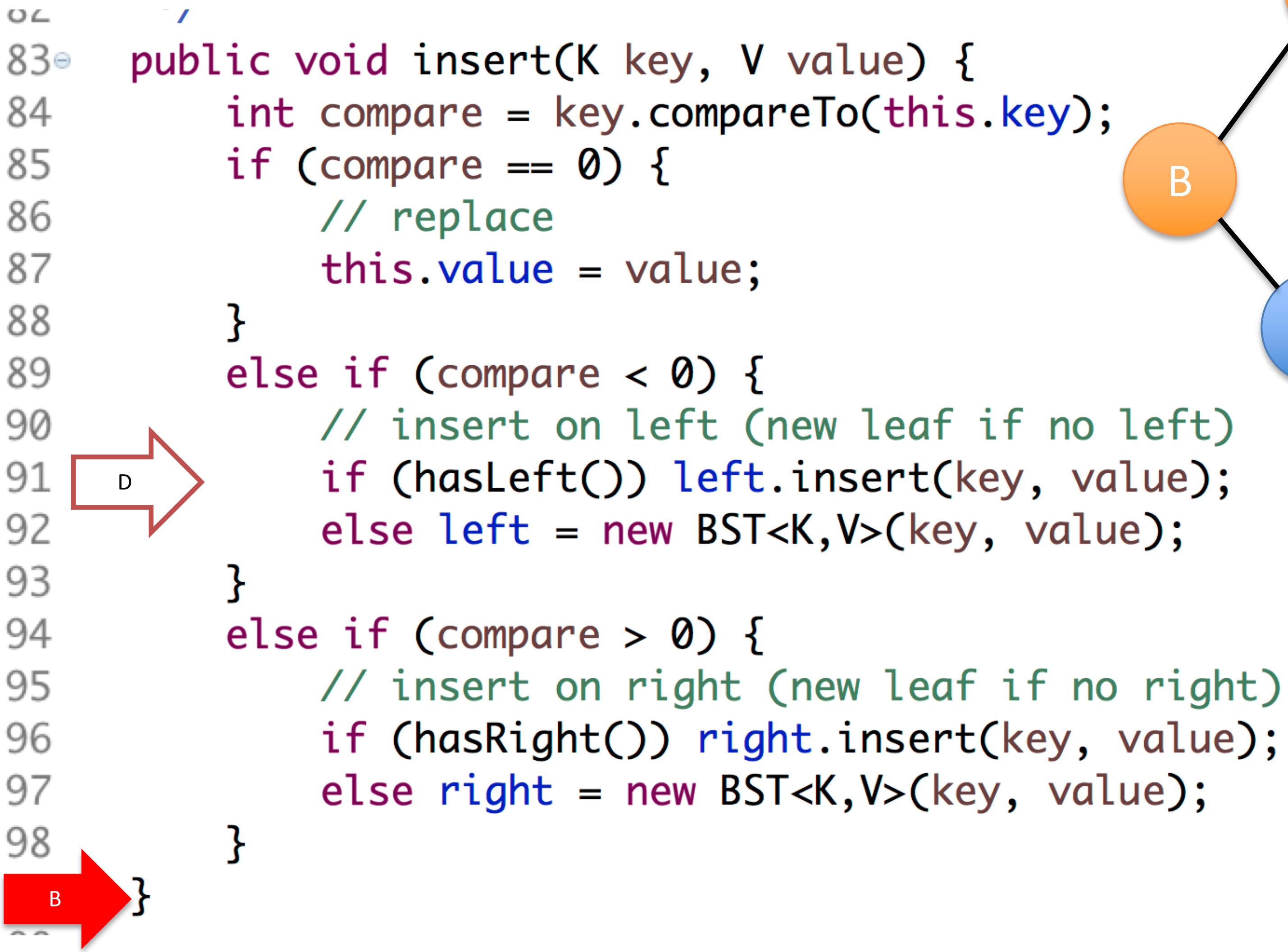
compare = 1

No right child

Add “C” as right

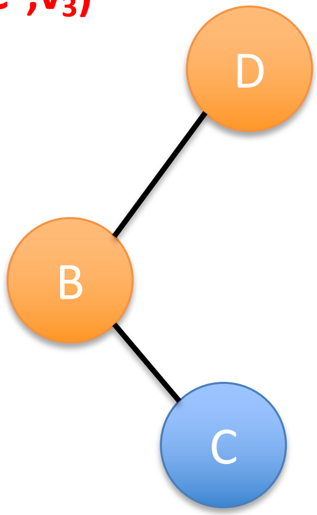
Comparable also helps inserting new Nodes

BST.java



```
    */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
}
```

t.insert(“C”,v_3)



“C” > “B”

compare = 1

No right child

Add “C” as right

B ends

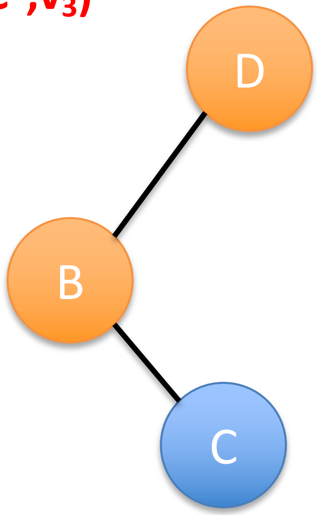
Comparable also helps inserting new Nodes

BST.java

```
     */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

D

t.insert("C",v_3)



"C" > "B"

compare = 1

No right child

Add "C" as right

B ends

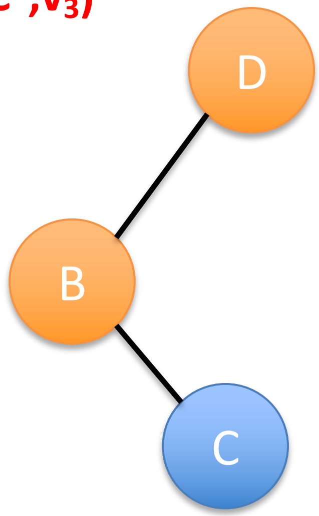
D ends

Comparable also helps inserting new Nodes

BST.java

```java
     */
    public void insert(K key, V value) {
        int compare = key.compareTo(this.key);
        if (compare == 0) {
            // replace
            this.value = value;
        }
        else if (compare < 0) {
            // insert on left (new leaf if no left)
            if (hasLeft()) left.insert(key, value);
            else left = new BST<K,V>(key, value);
        }
        else if (compare > 0) {
            // insert on right (new leaf if no right)
            if (hasRight()) right.insert(key, value);
            else right = new BST<K,V>(key, value);
        }
    }
```

t.insert("C",v$_3$)



"C" > "B"

compare = 1

No right child

Add "C" as right

B ends

D ends

Done

BST.java - delete

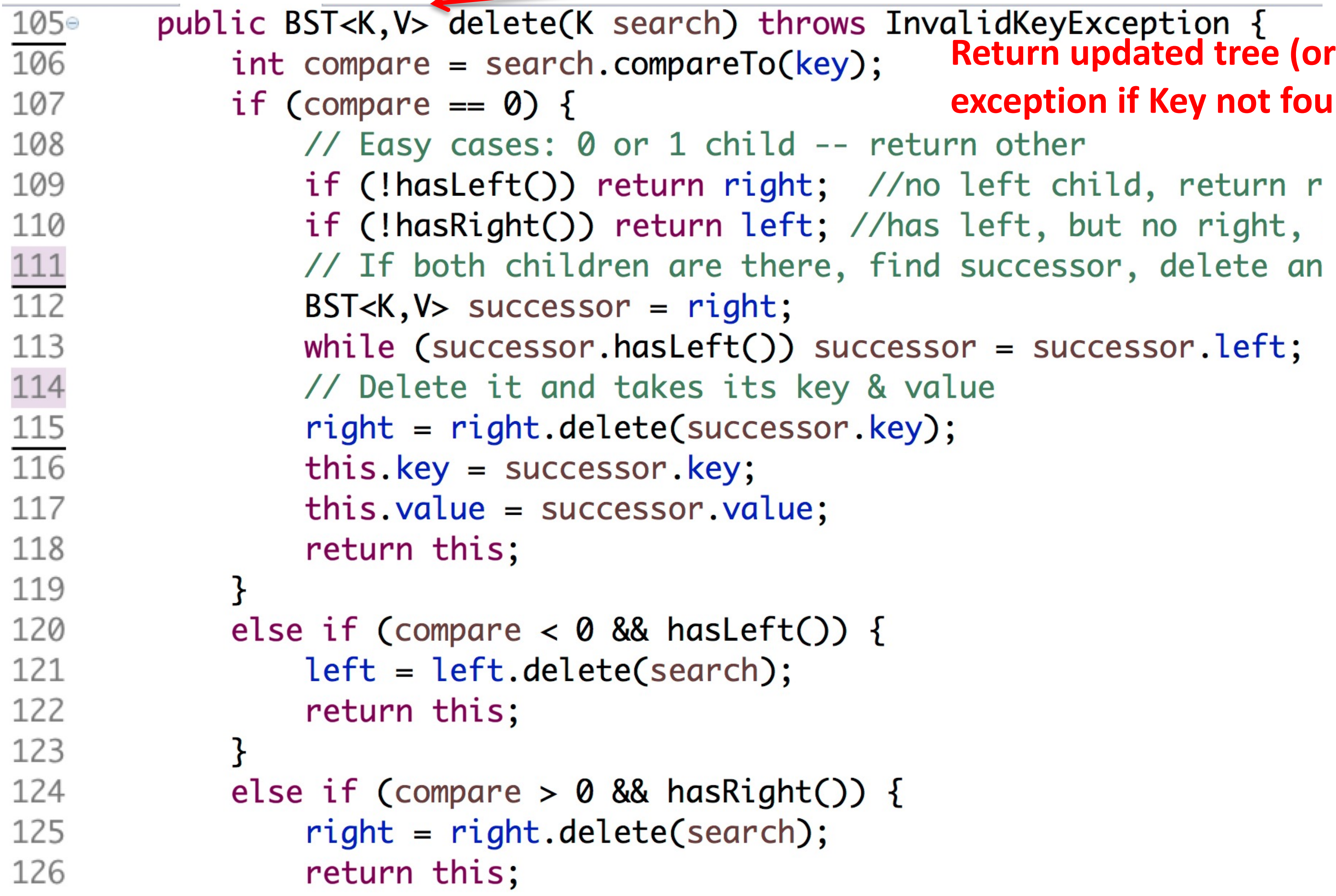
ANNOTATED SLIDES

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

BST.java

Delete node with Key *search*

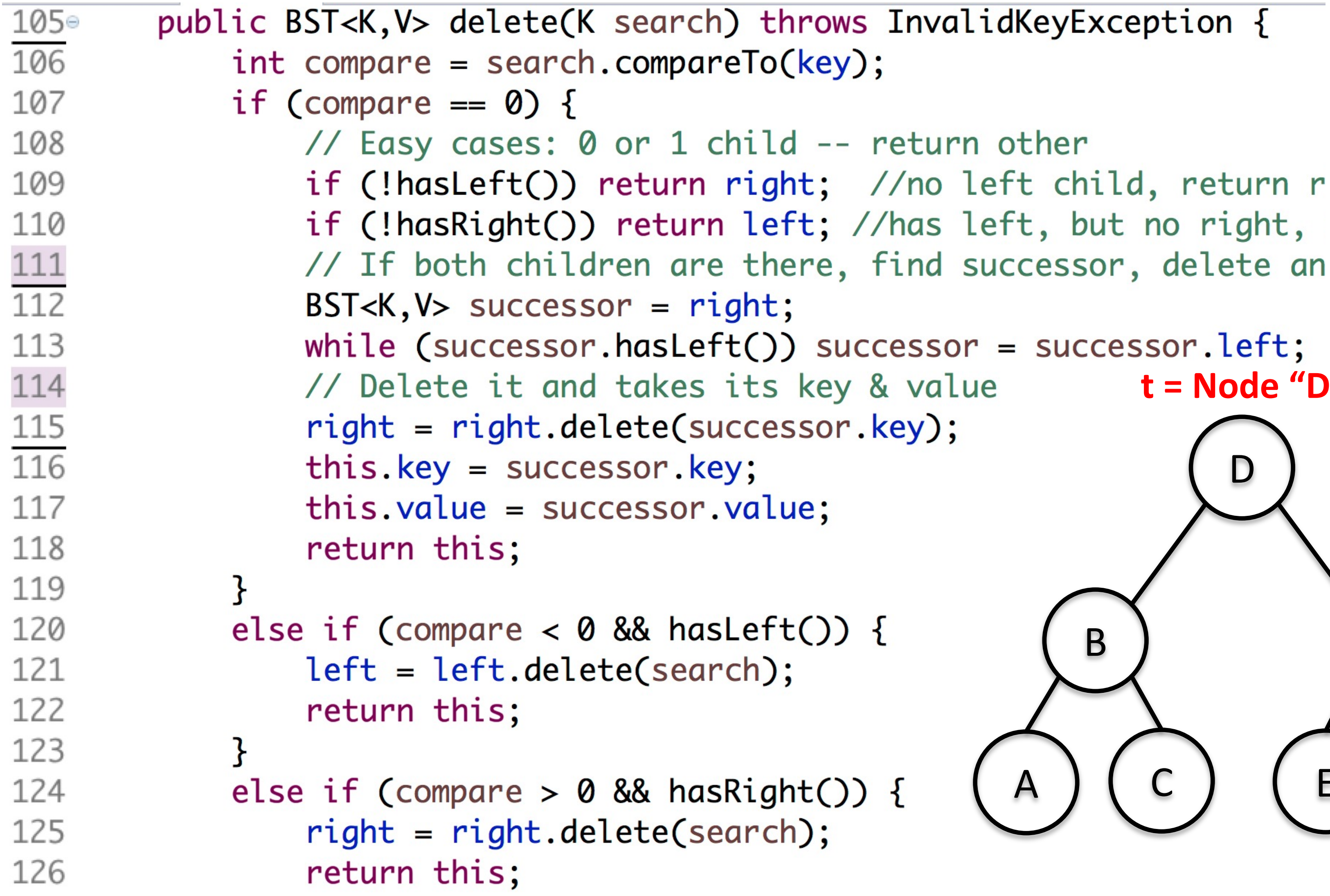
Return updated tree (or throw exception if Key not found)



```java
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;  //no left child, return r
        if (!hasRight()) return left; //has left, but no right,
        // If both children are there, find successor, delete an
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it and takes its key & value
        right = right.delete(successor.key);
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
```

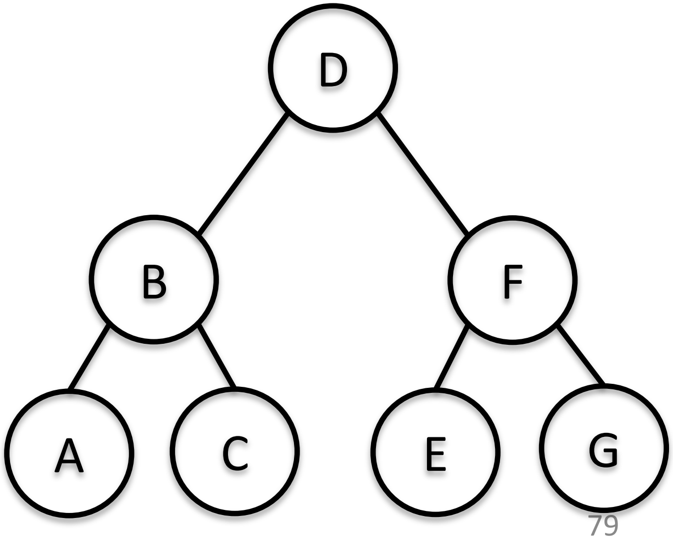
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



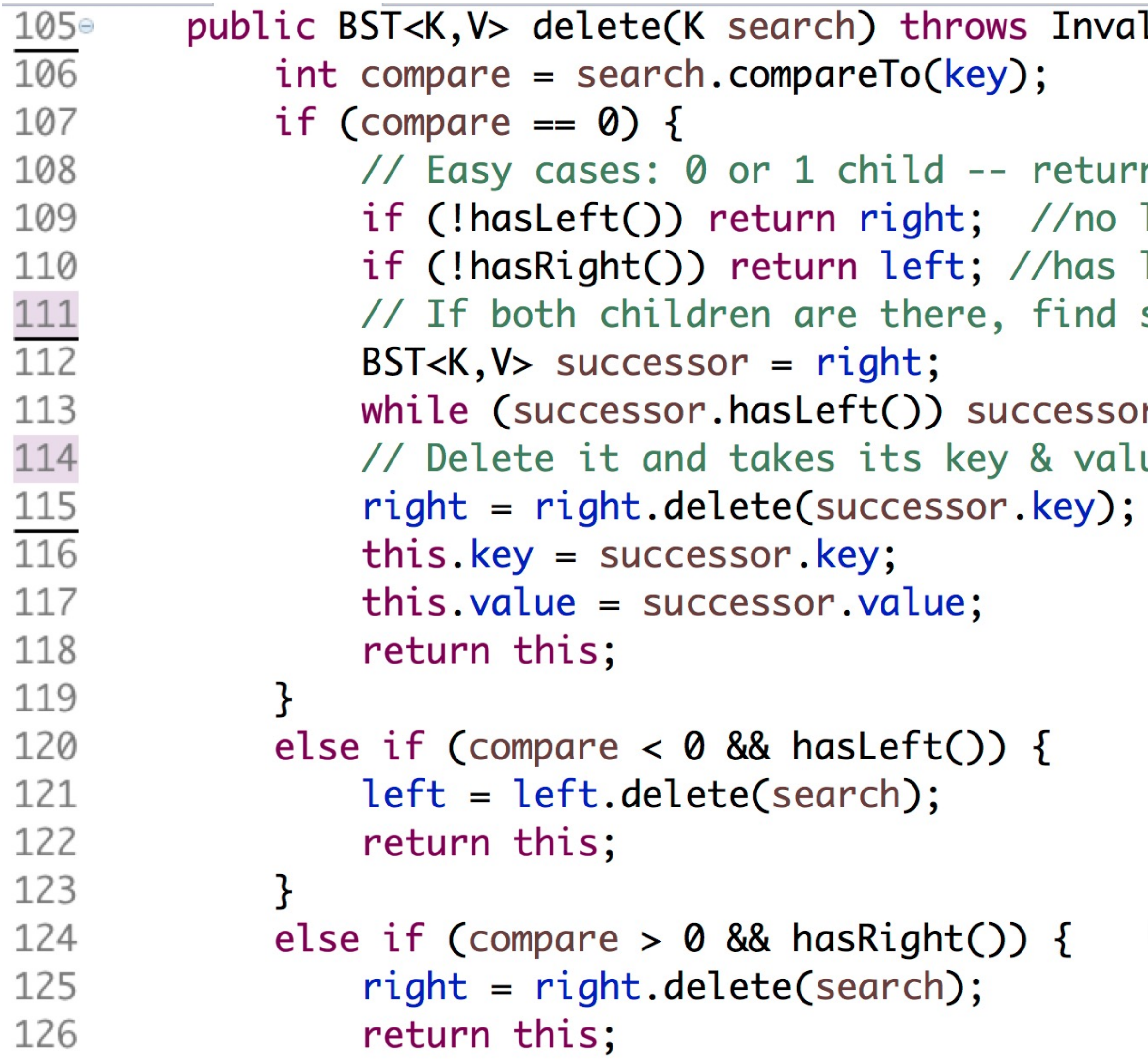
```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

t = Node "D"



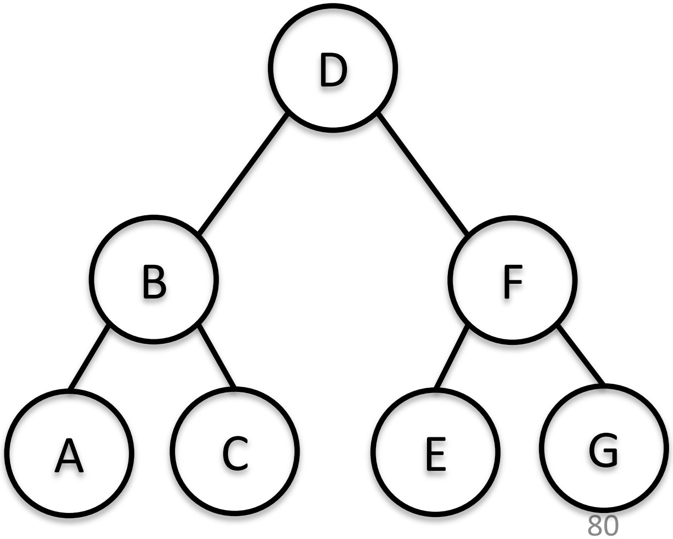
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



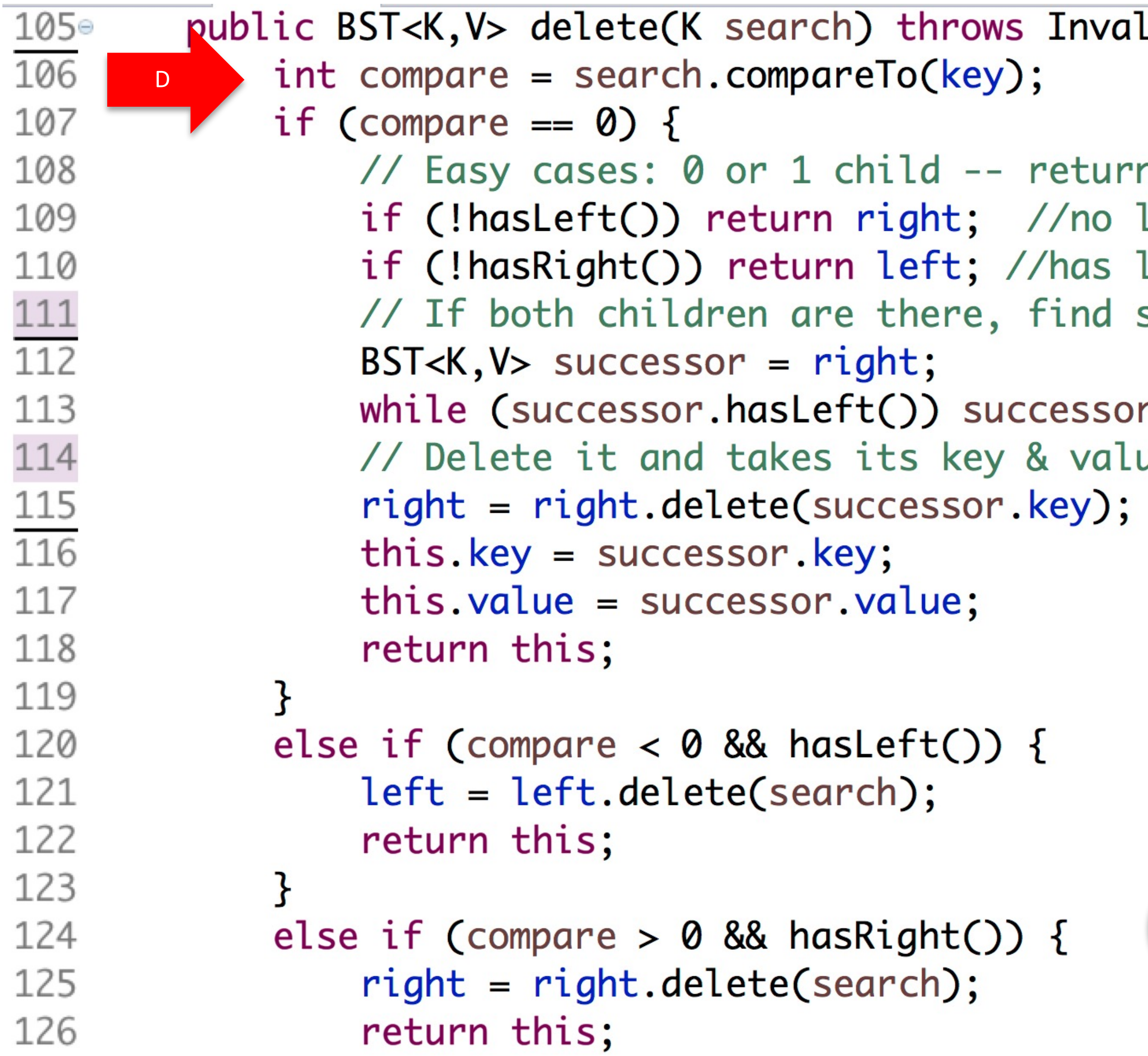
```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

t = t.delete("A")



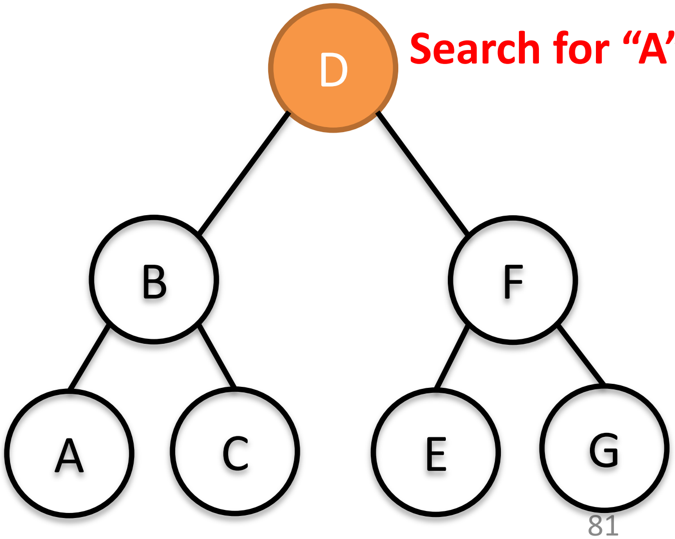
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

t = t.delete("A")



Search for "A"

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

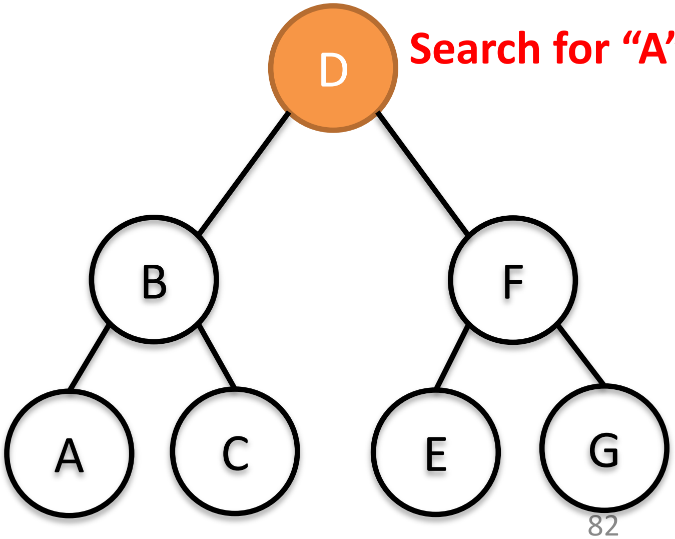
BST.java

```
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;  //no left child, return r
        if (!hasRight()) return left; //has left, but no right,
        // If both children are there, find successor, delete an
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it and takes its key & value
        right = right.delete(successor.key);
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
```

D

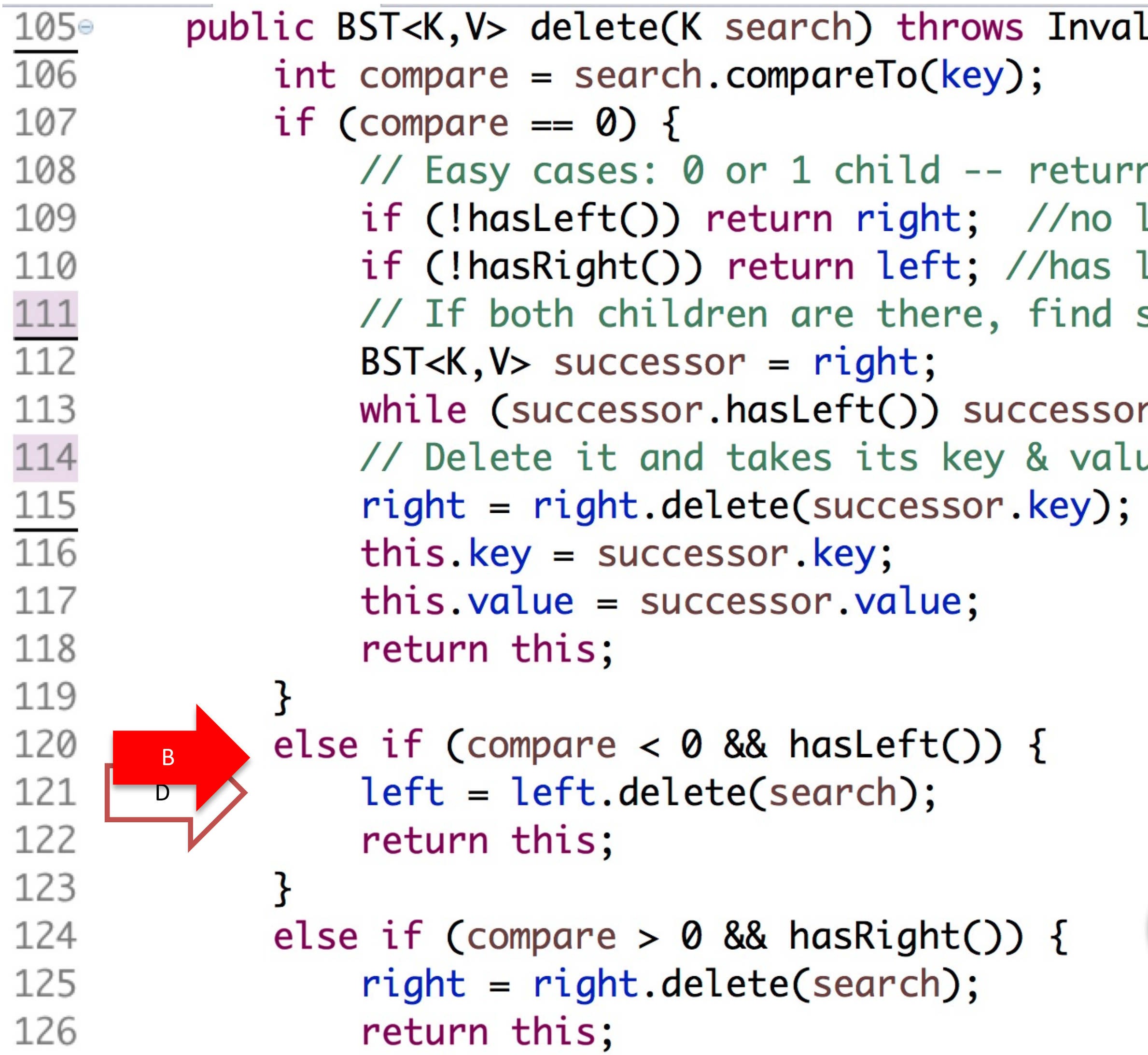
t = t.delete("A")

Search for "A"



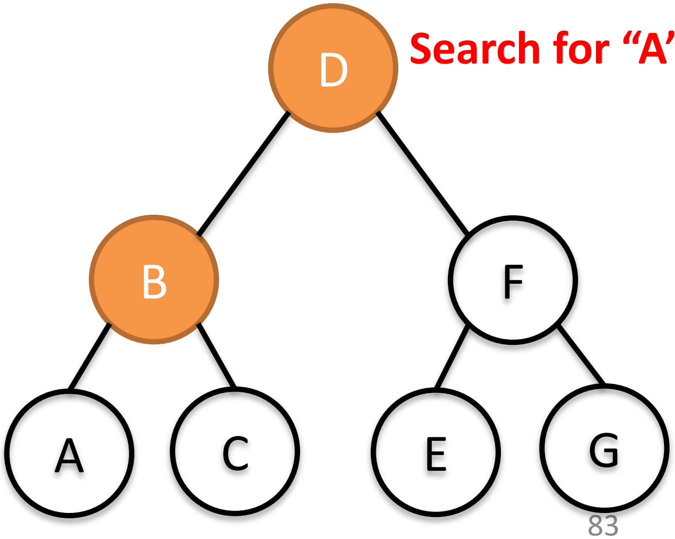
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

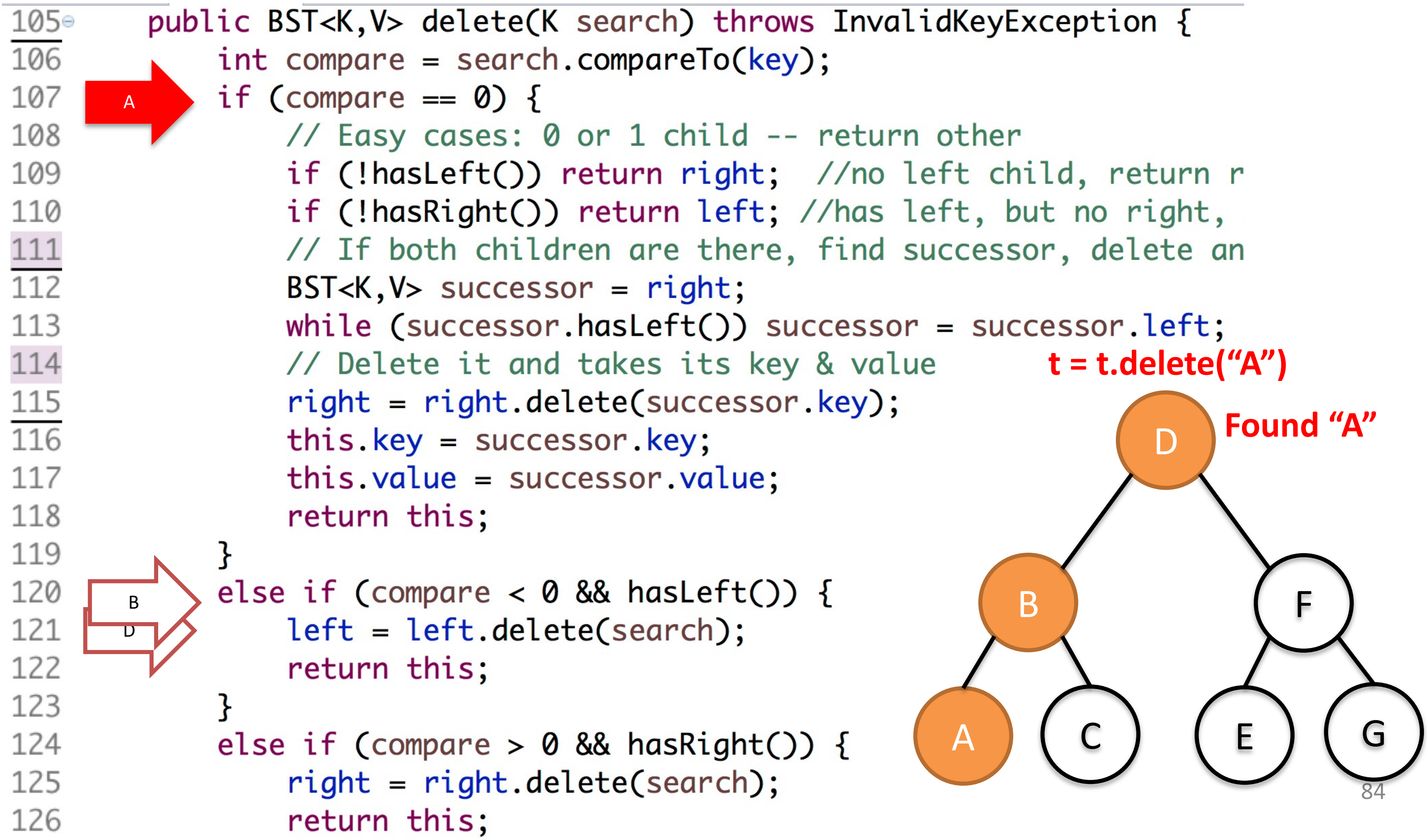
t = t.delete("A")



Search for "A"

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

BST.java



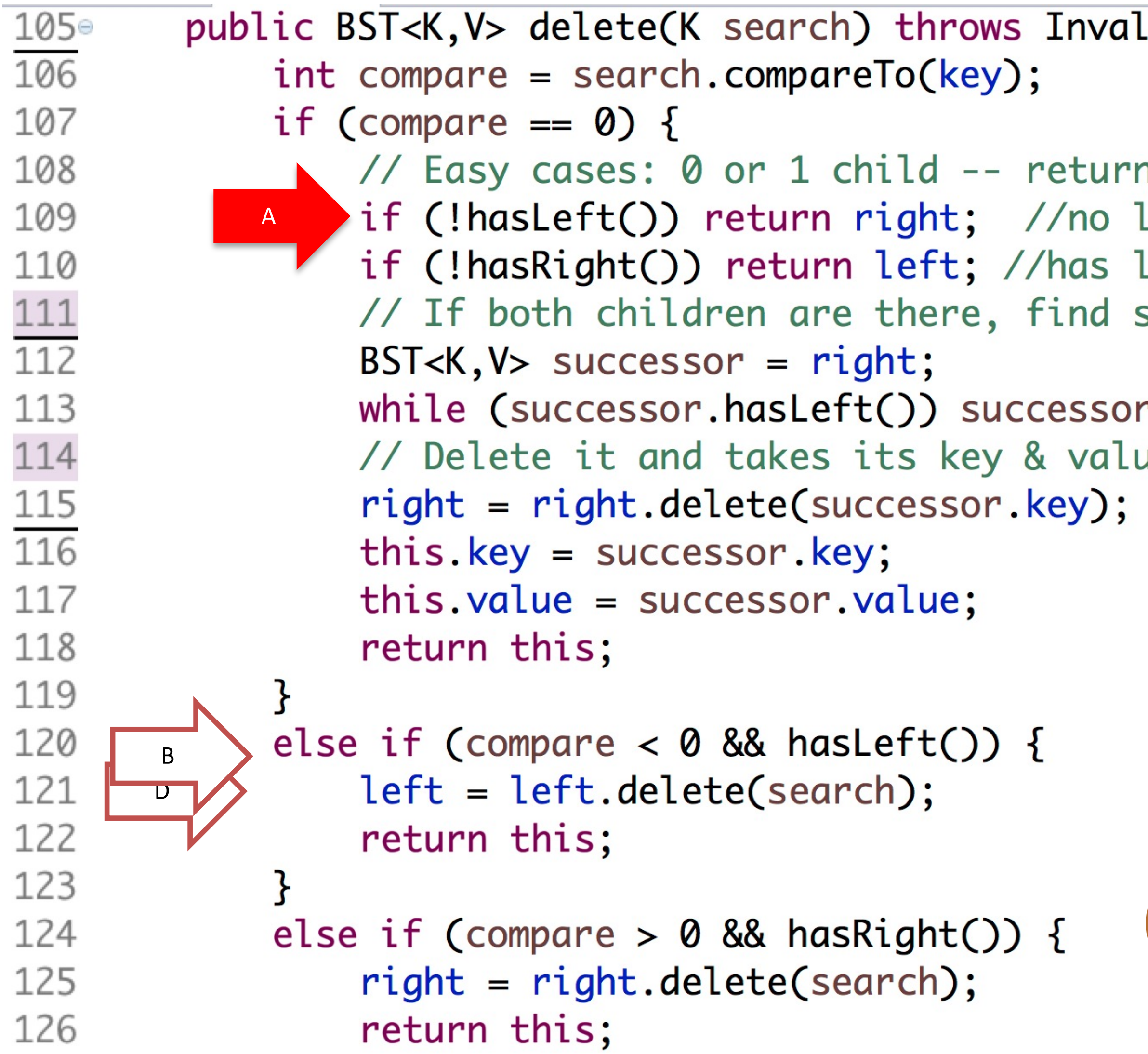
```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

t = t.delete("A")

Found "A"

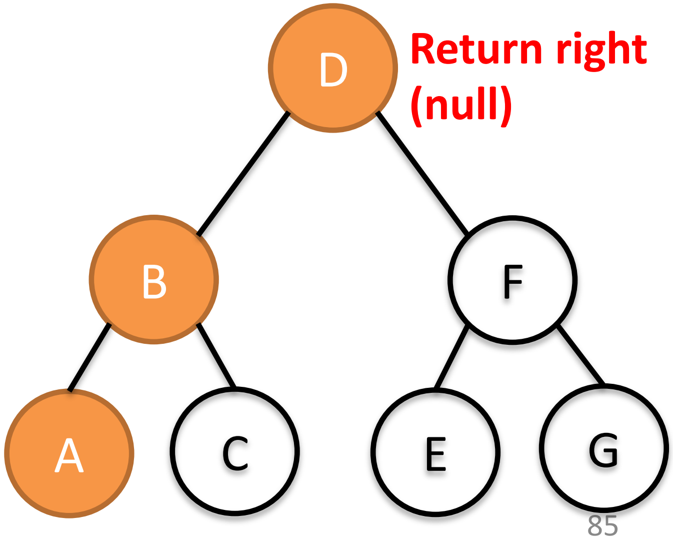
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;   //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

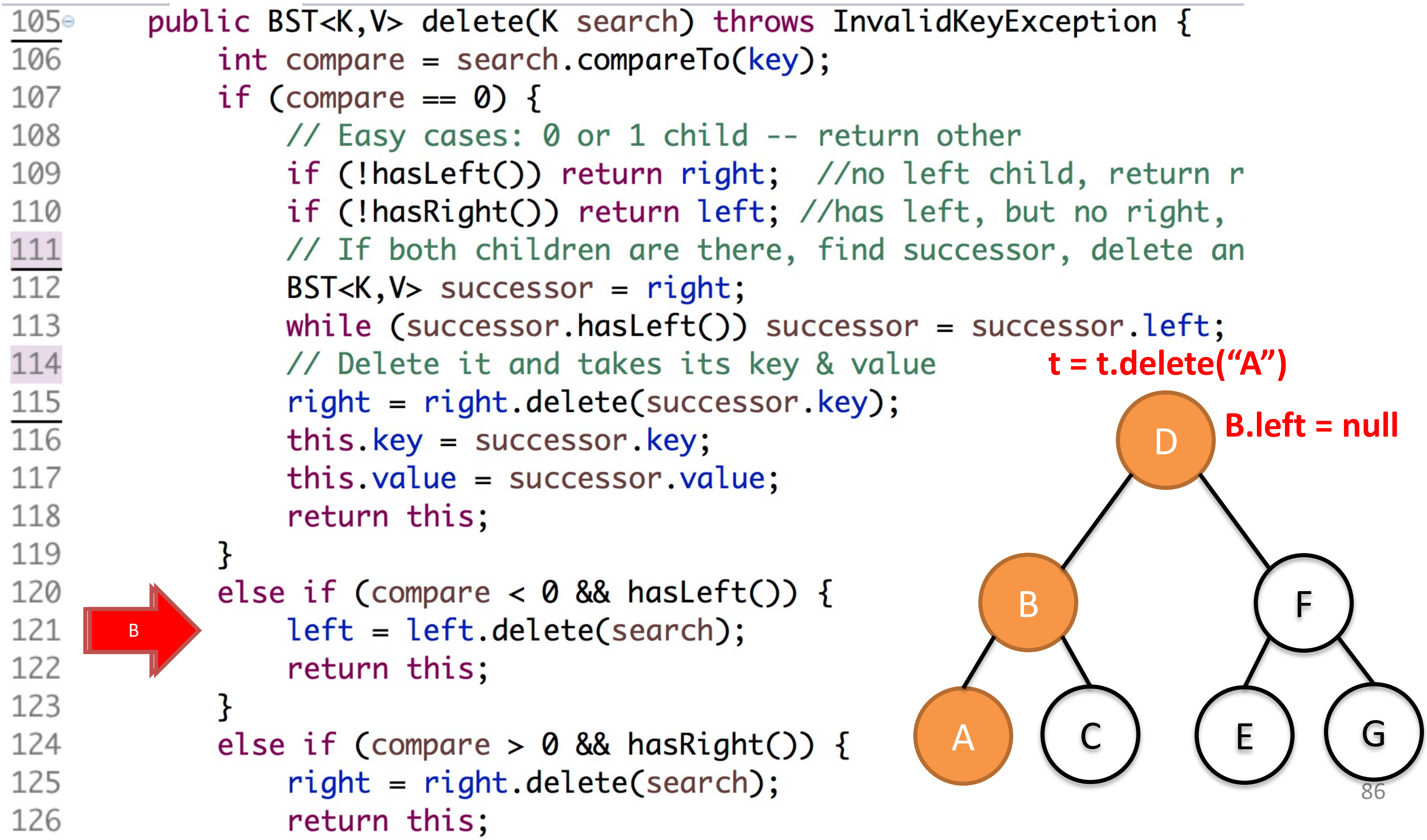
t = t.delete("A")



Return right (null)

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

BST.java



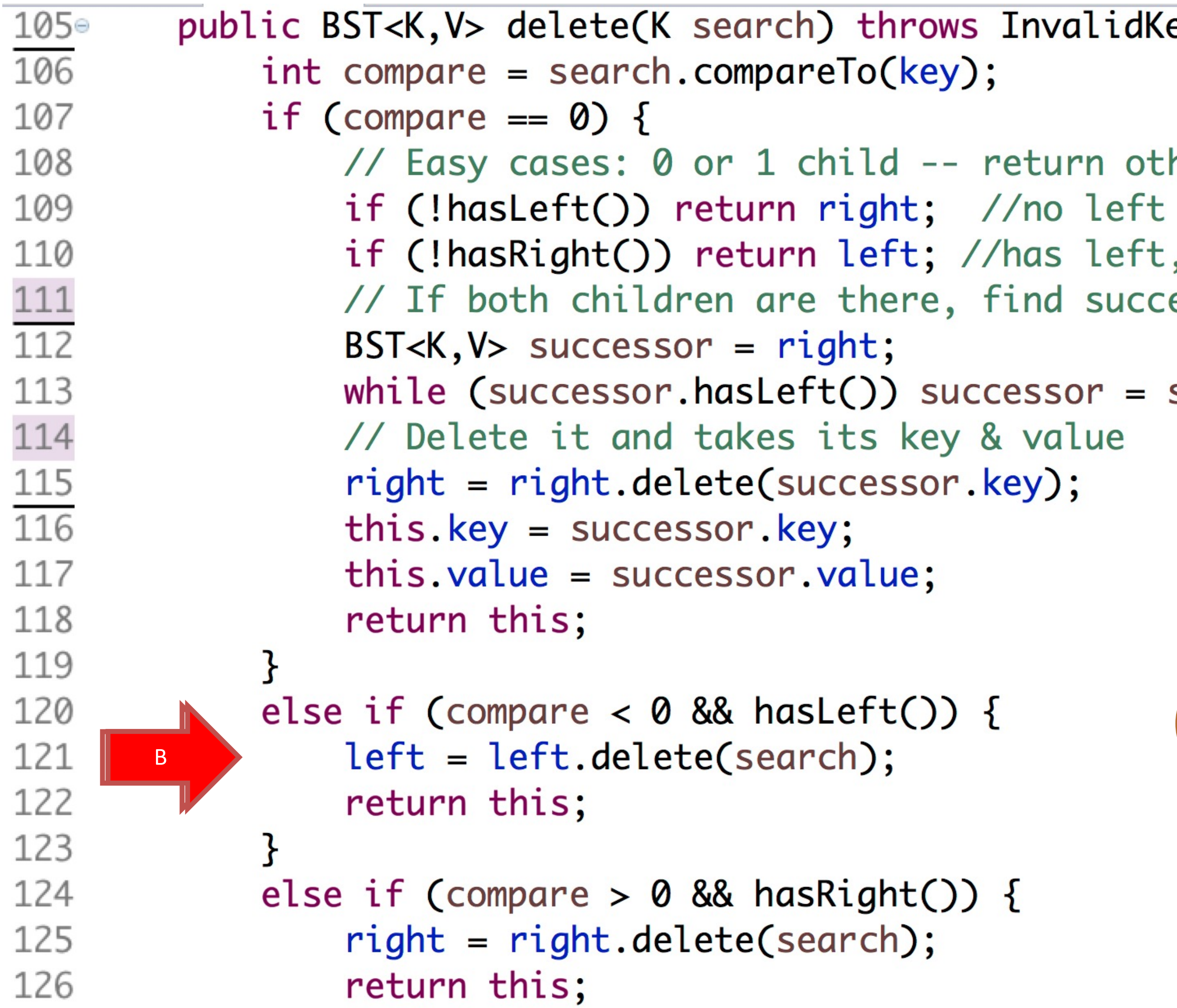
```java
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;  //no left child, return r
        if (!hasRight()) return left; //has left, but no right,
        // If both children are there, find successor, delete an
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it and takes its key & value
        right = right.delete(successor.key);
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
```

t = t.delete("A")

B.left = null

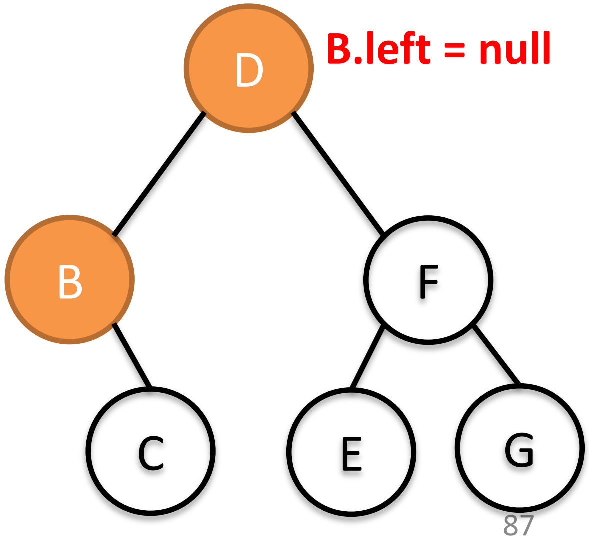
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

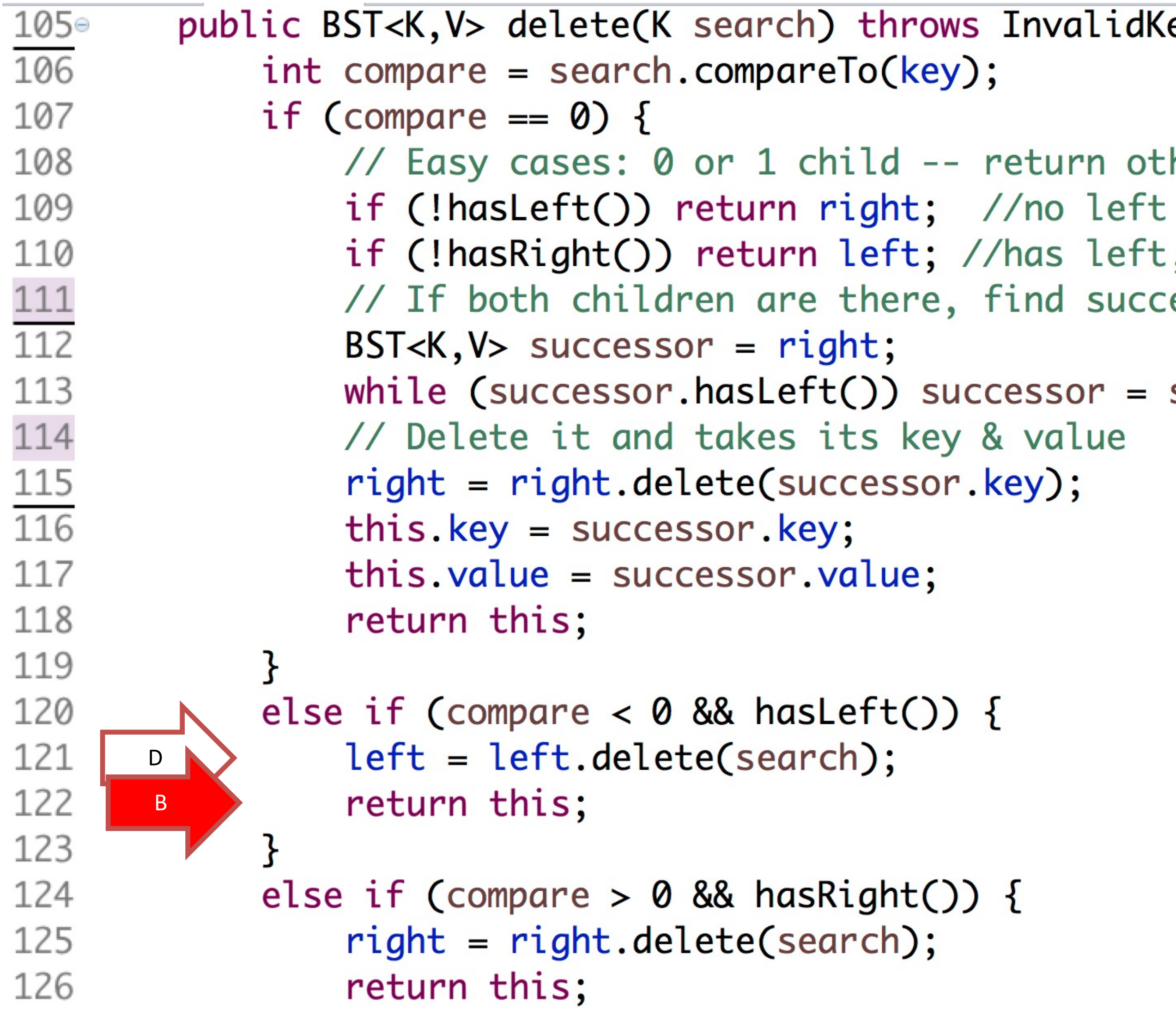
t = t.delete("A")



B.left = null

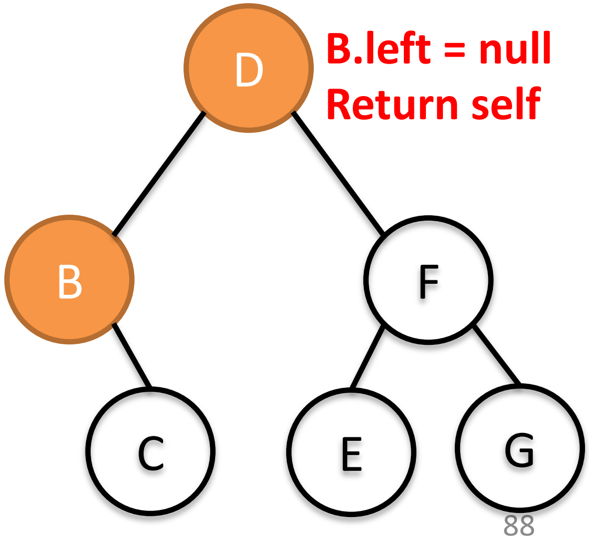
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

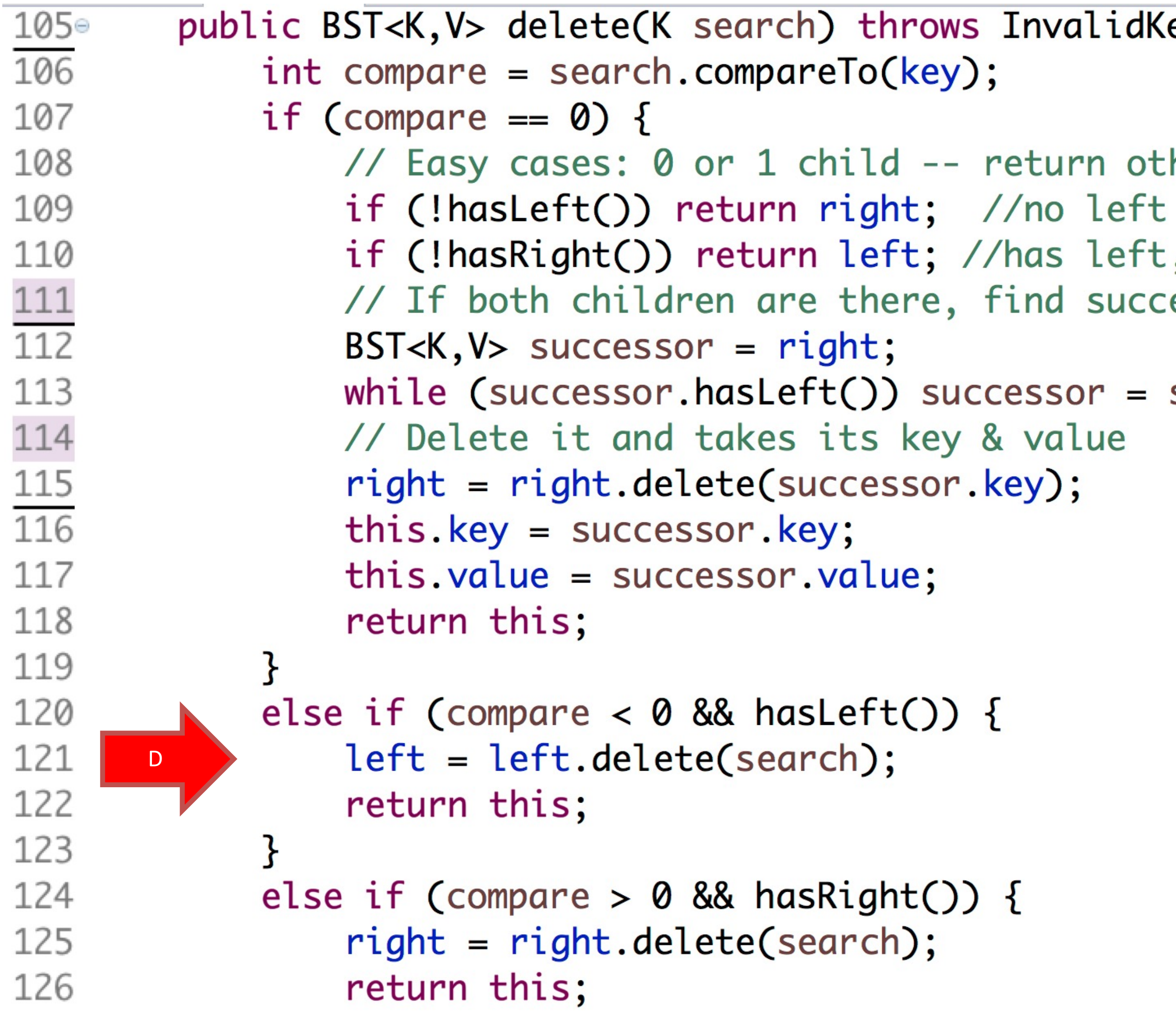
t = t.delete("A")



B.left = null
Return self

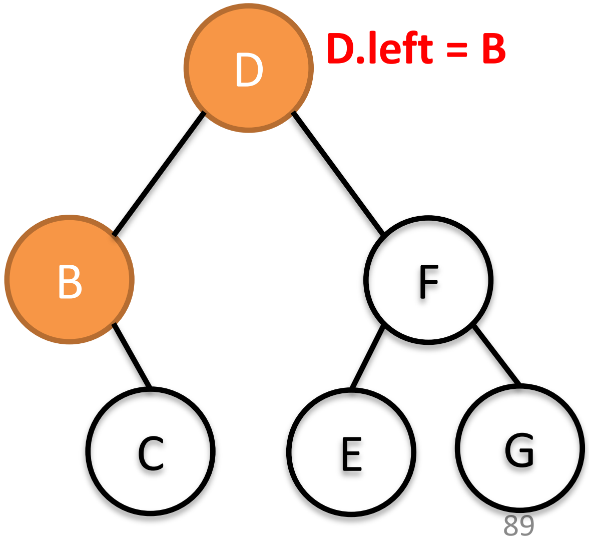
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

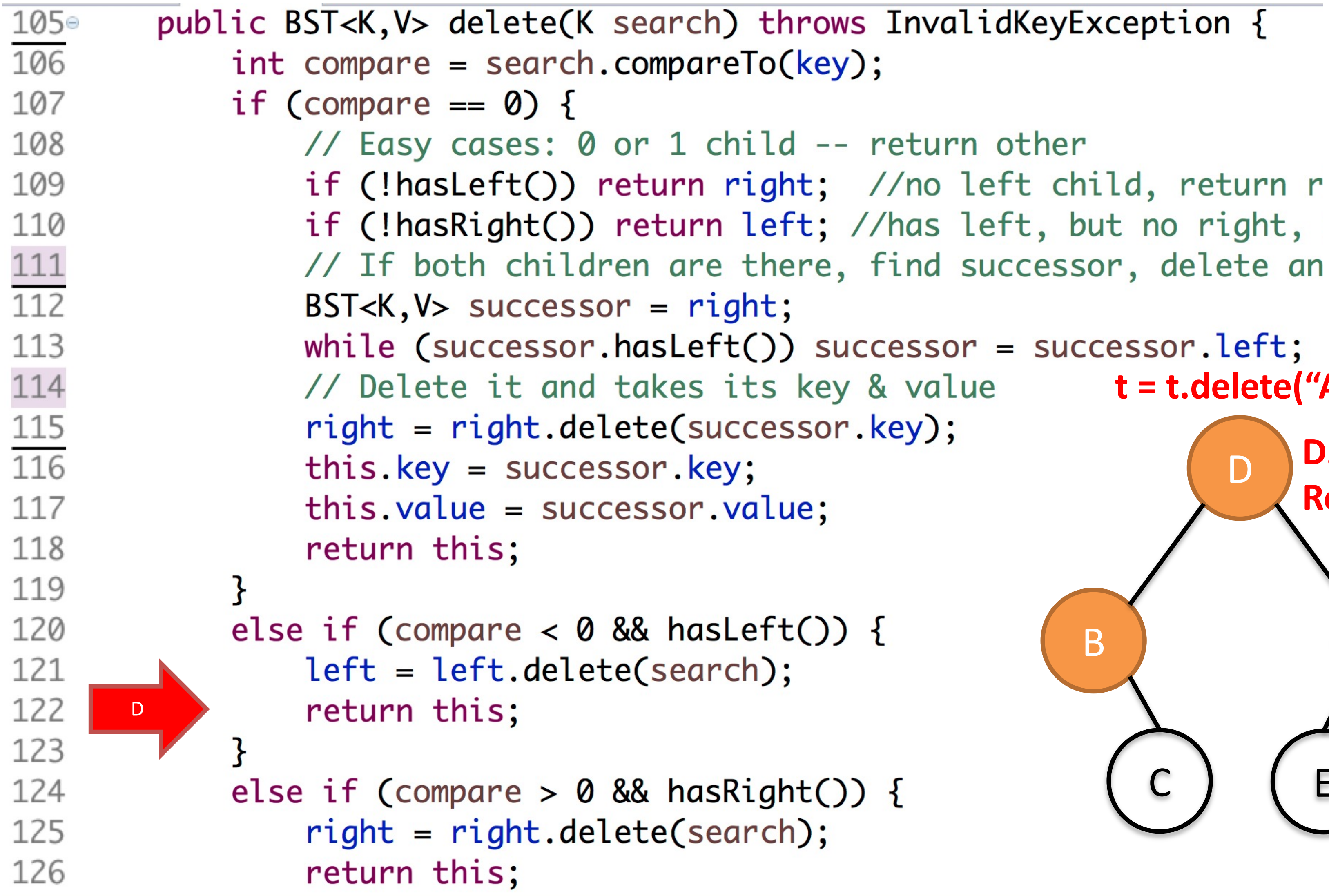
t = t.delete("A")



D.left = B

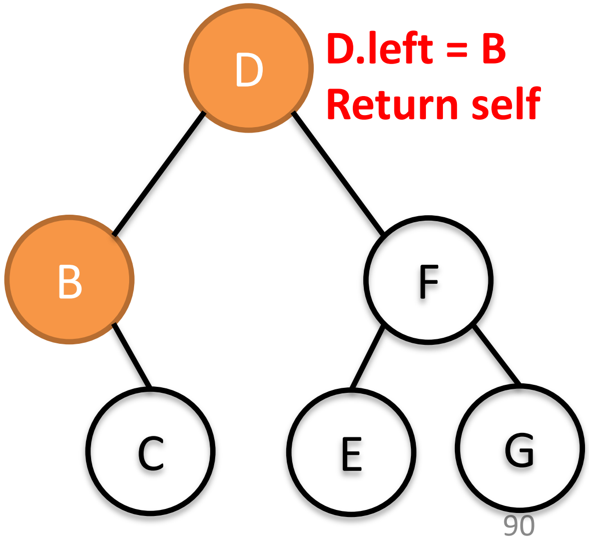
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

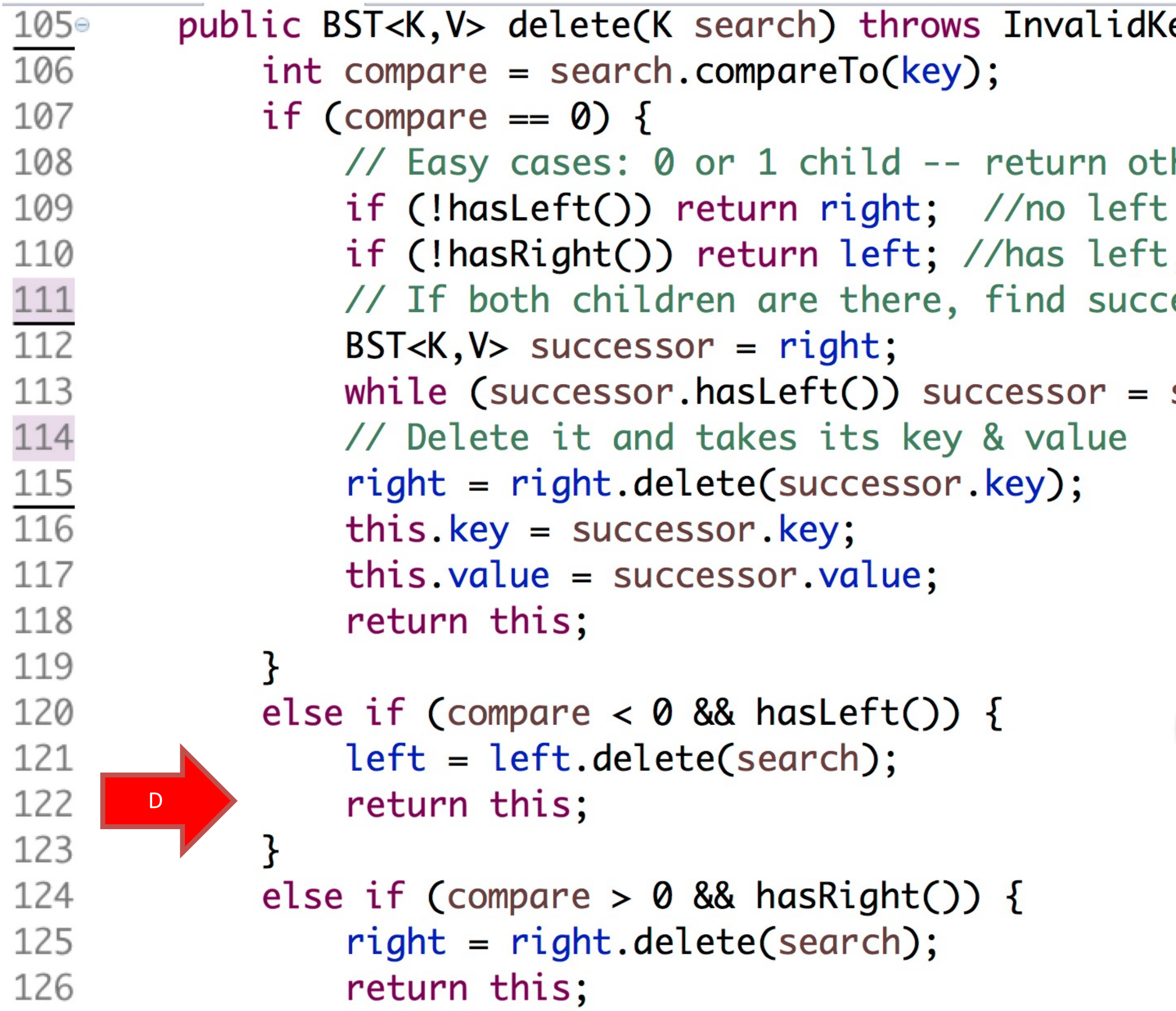
t = t.delete("A")



D.left = B
Return self

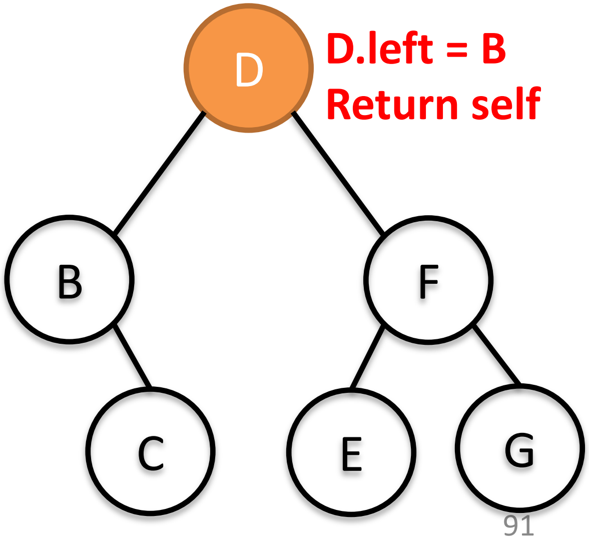
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

t = t.delete("A")



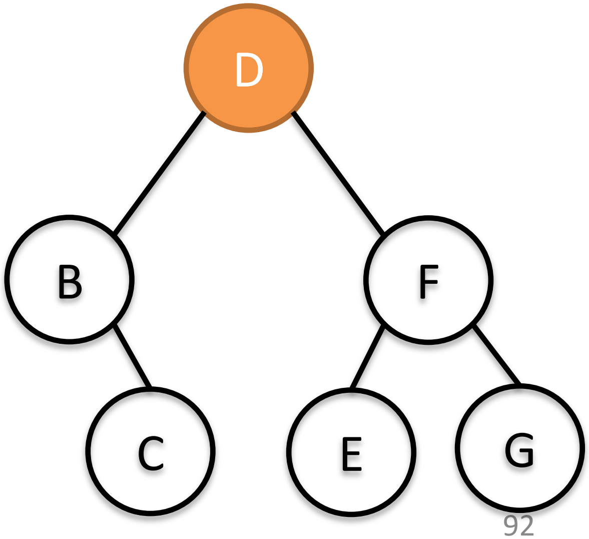
D.left = B
Return self

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

BST.java

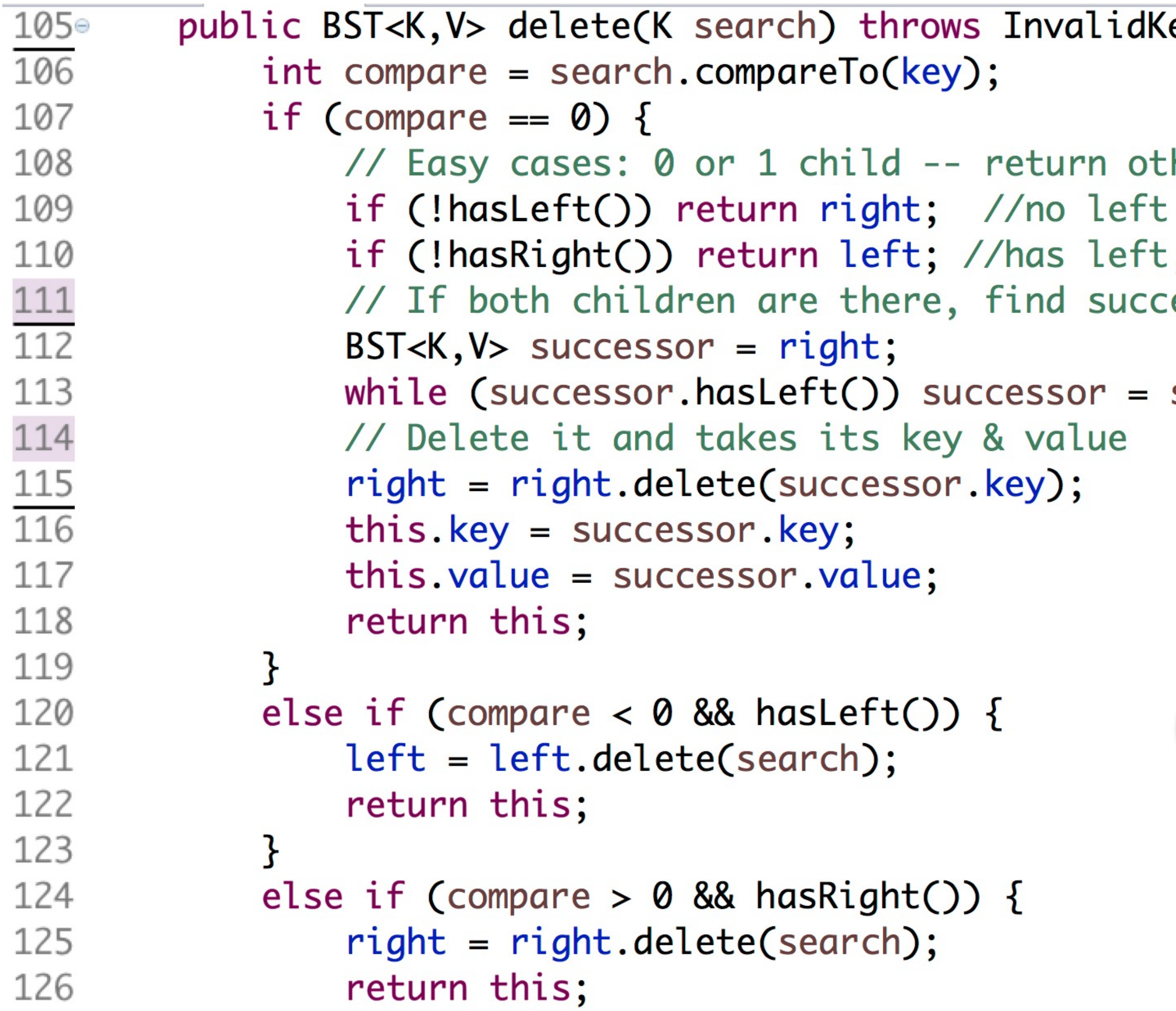
```java
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;  //no left child, return r
        if (!hasRight()) return left; //has left, but no right,
        // If both children are there, find successor, delete an
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it and takes its key & value
        right = right.delete(successor.key);
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
```

t = Node "D"



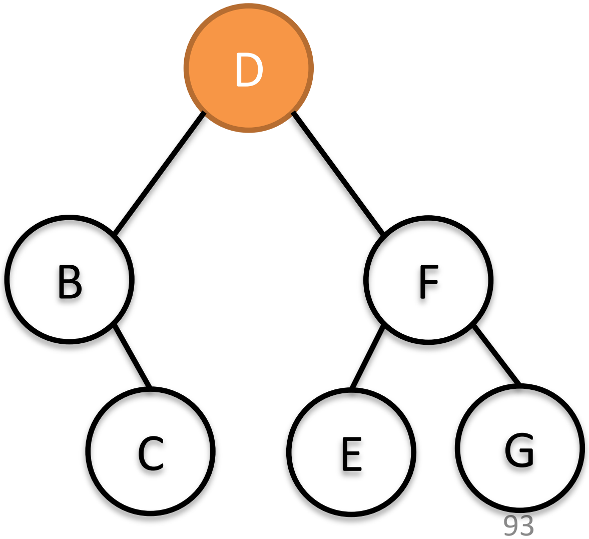
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



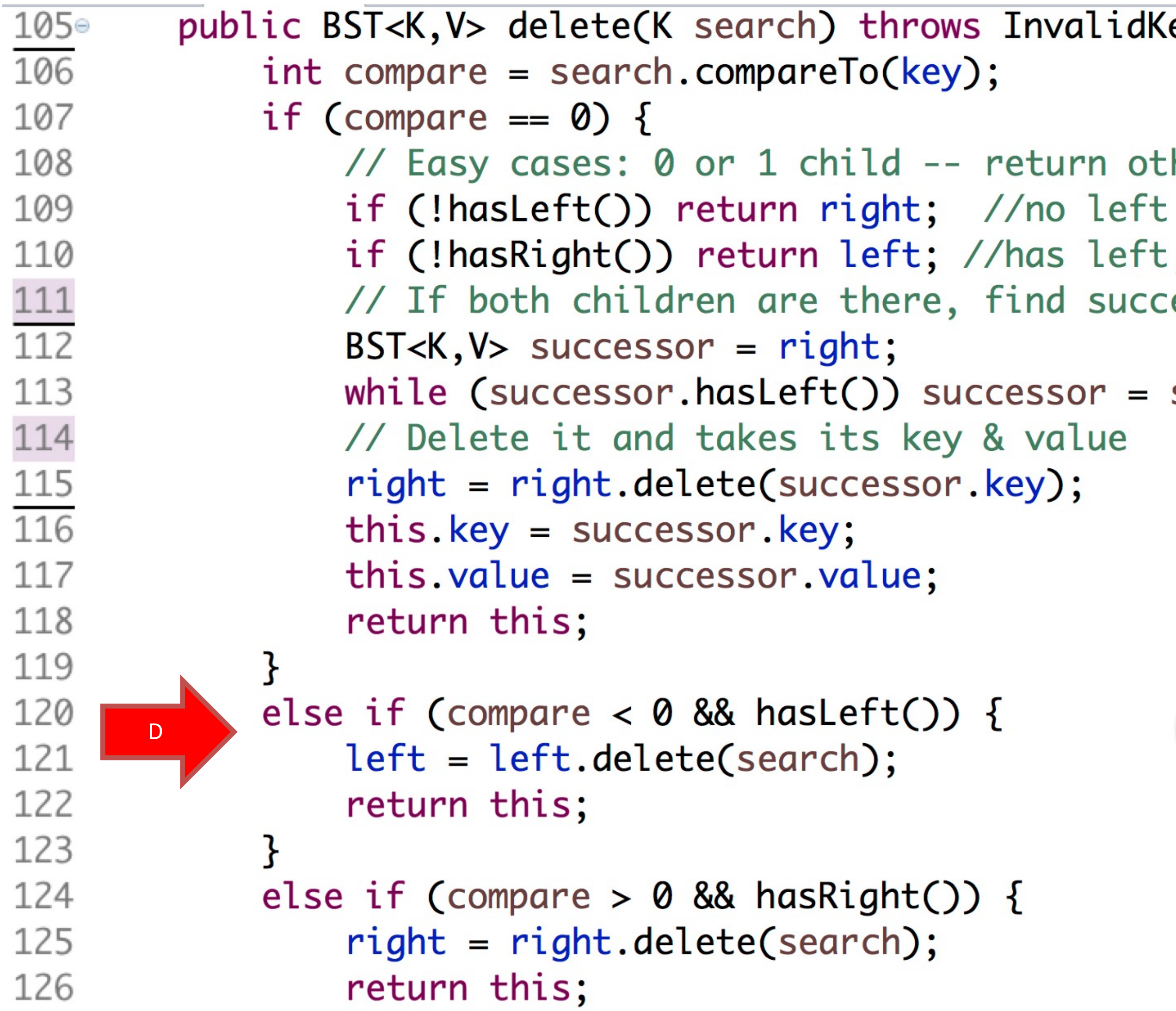
```java
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

t = t.delete("B")



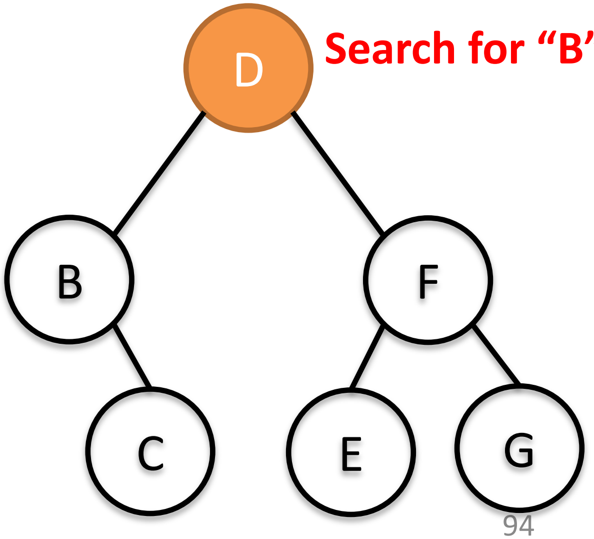
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```java
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

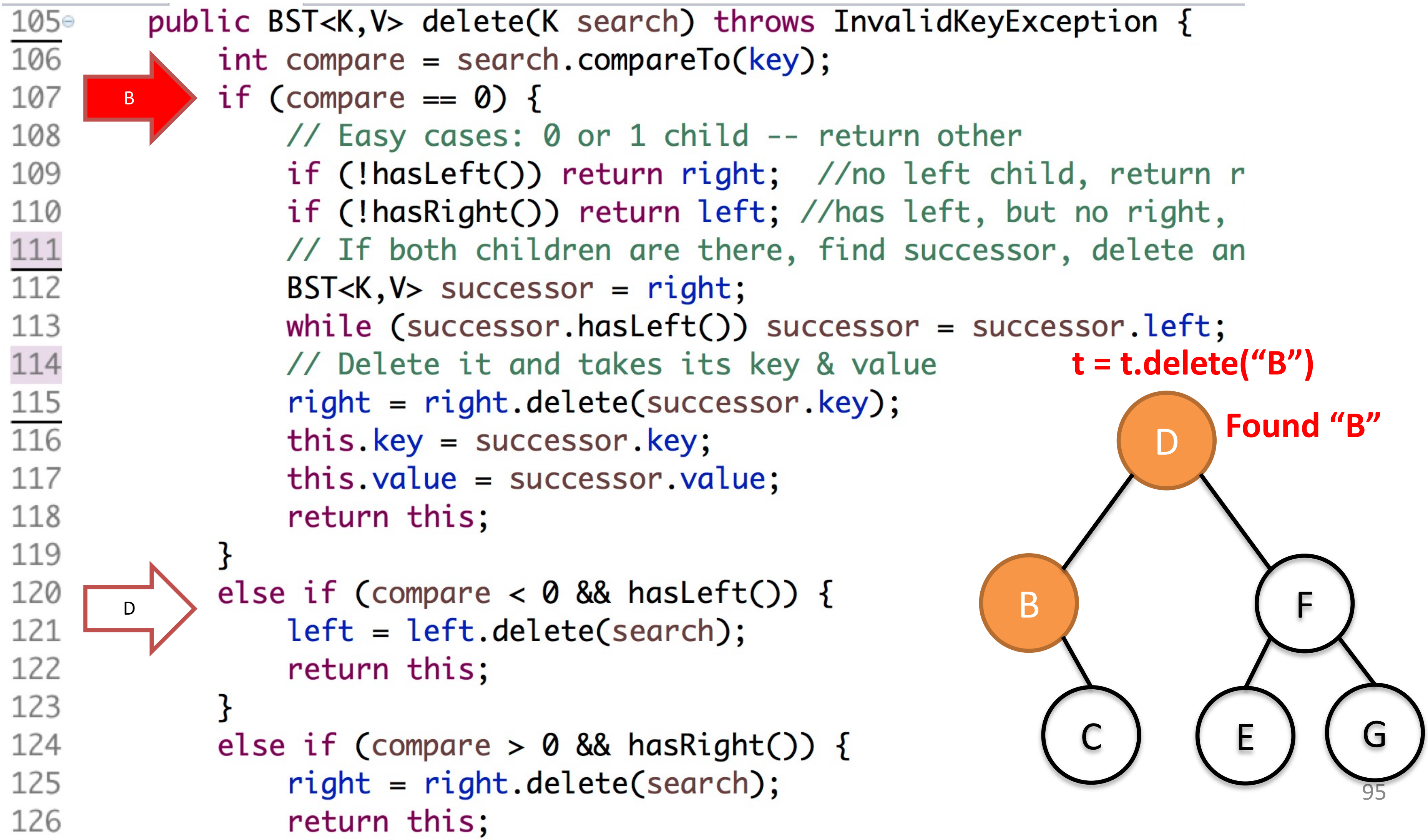
t = t.delete("B")



Search for "B"

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

BST.java



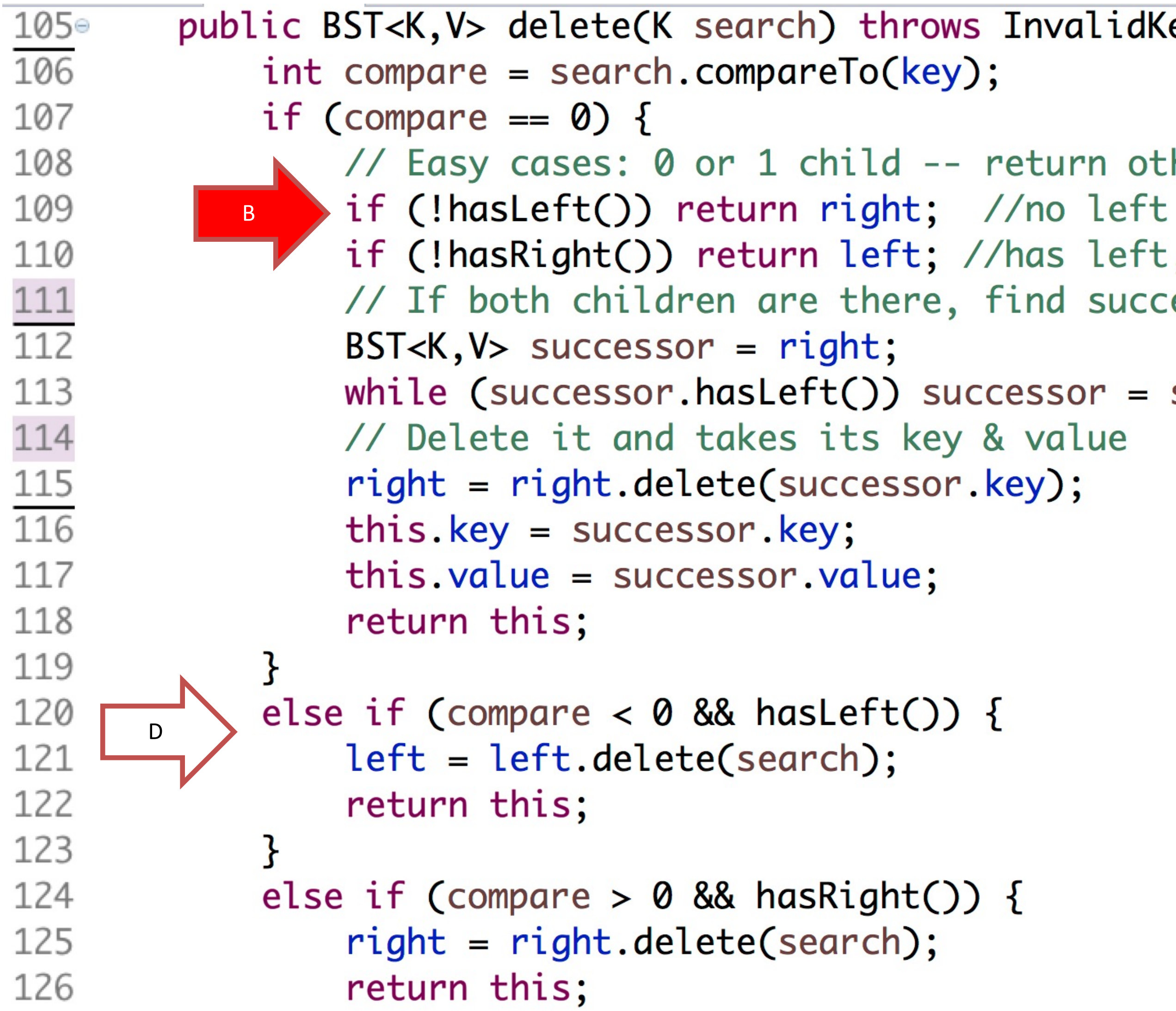
```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

t = t.delete("B")

Found "B"

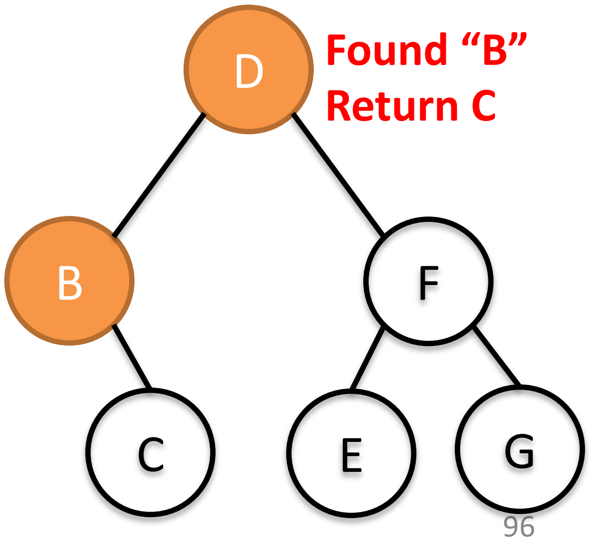
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```java
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;   //no left child, return r
        if (!hasRight()) return left; //has left, but no right,
        // If both children are there, find successor, delete an
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it and takes its key & value
        right = right.delete(successor.key);
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
```

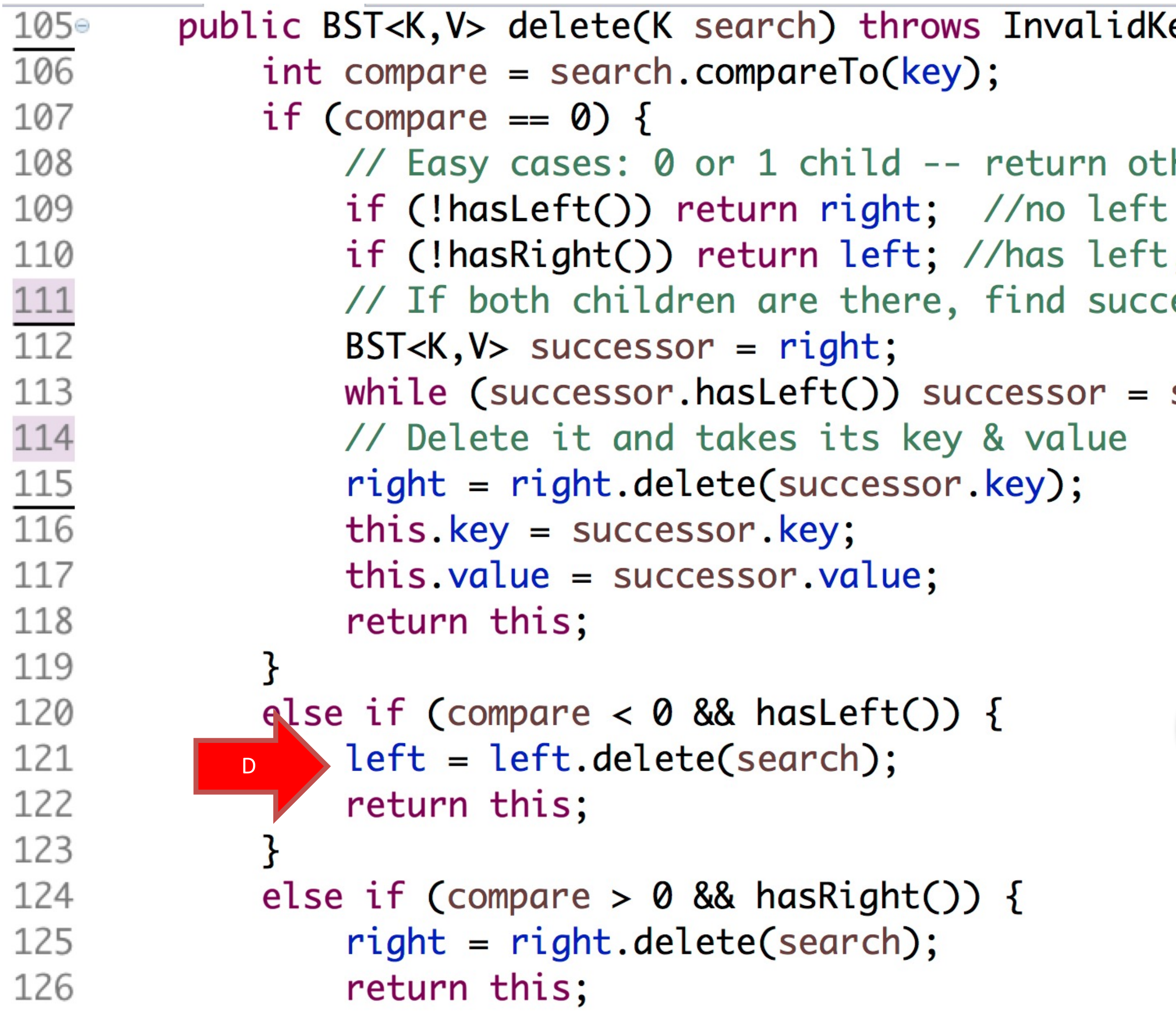
t = t.delete("B")



Found "B"
Return C

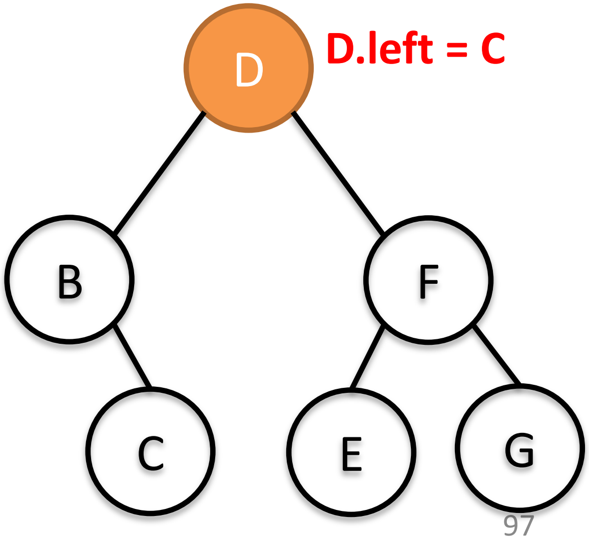
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```java
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

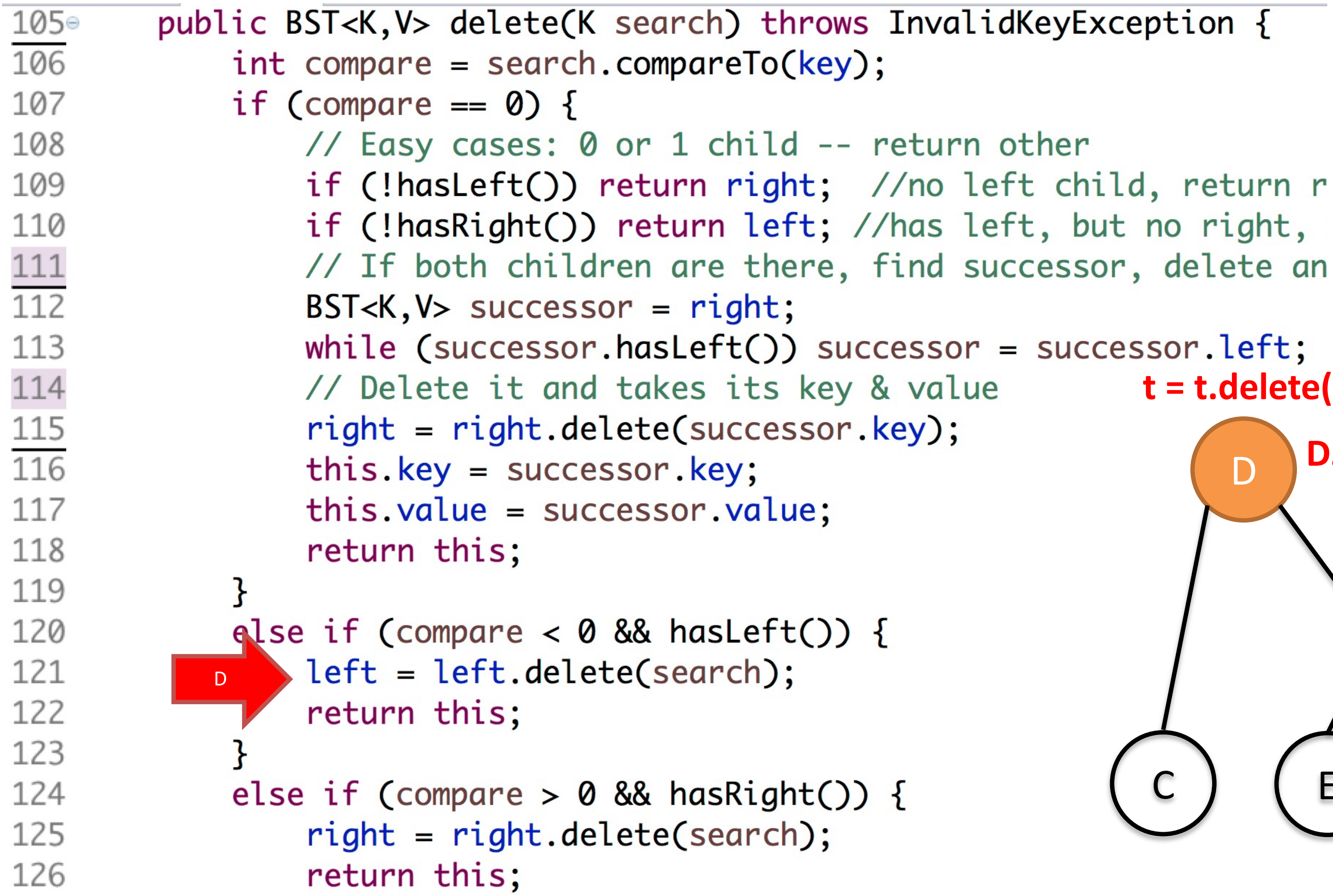
t = t.delete(“B”)



D.left = C

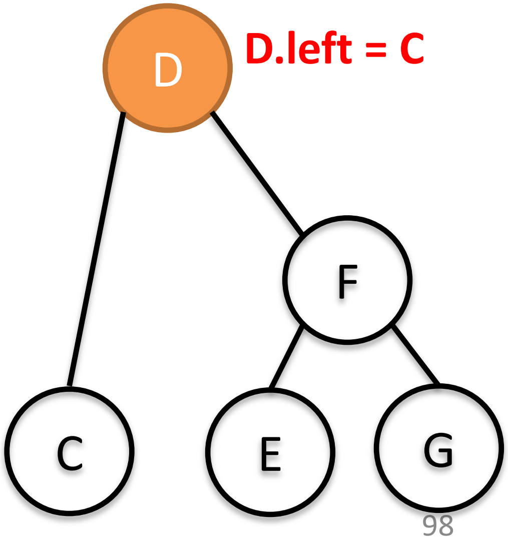
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```java
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

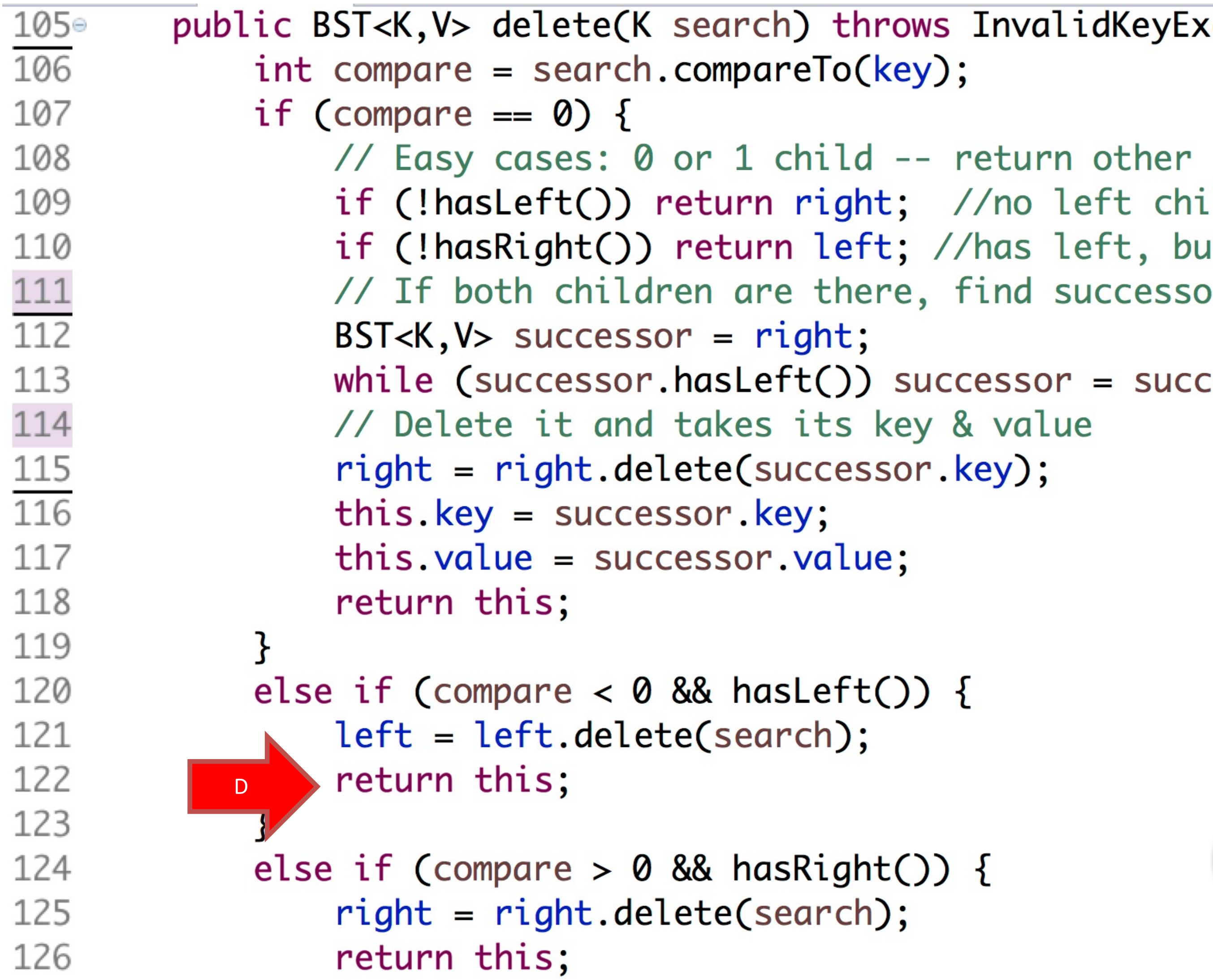
t = t.delete("B")



D.left = C

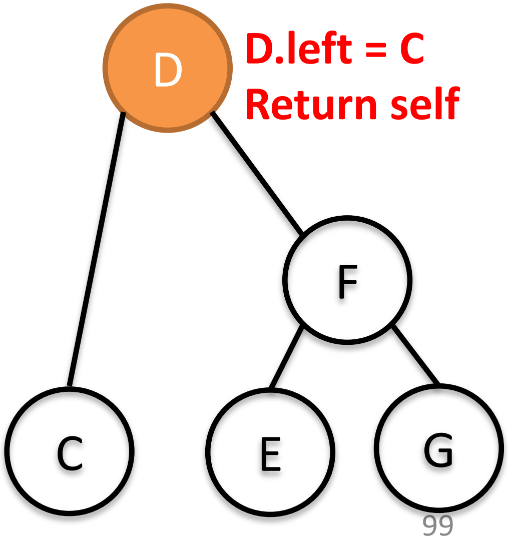
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```java
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

t = t.delete("B")

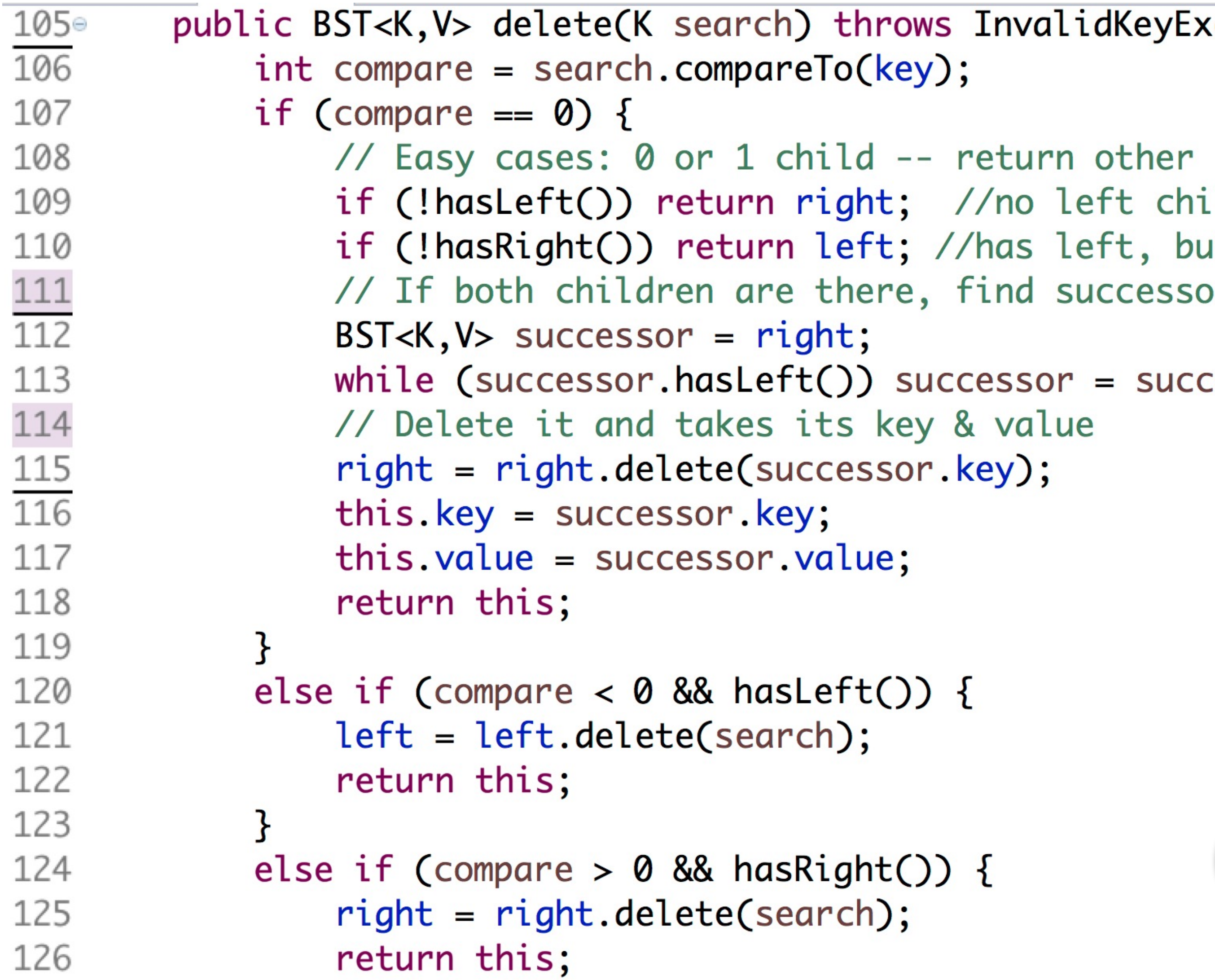


D.left = C

Return self

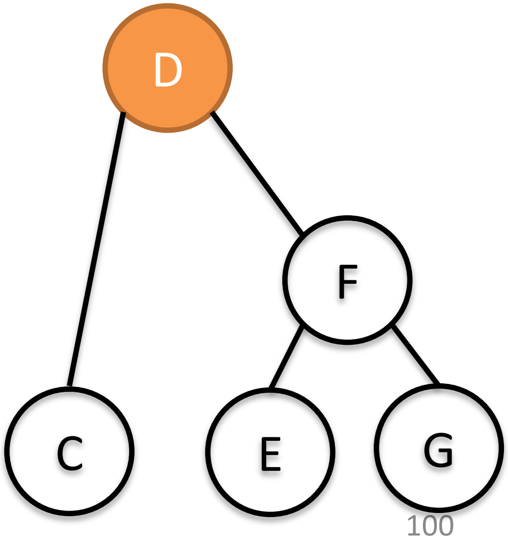
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



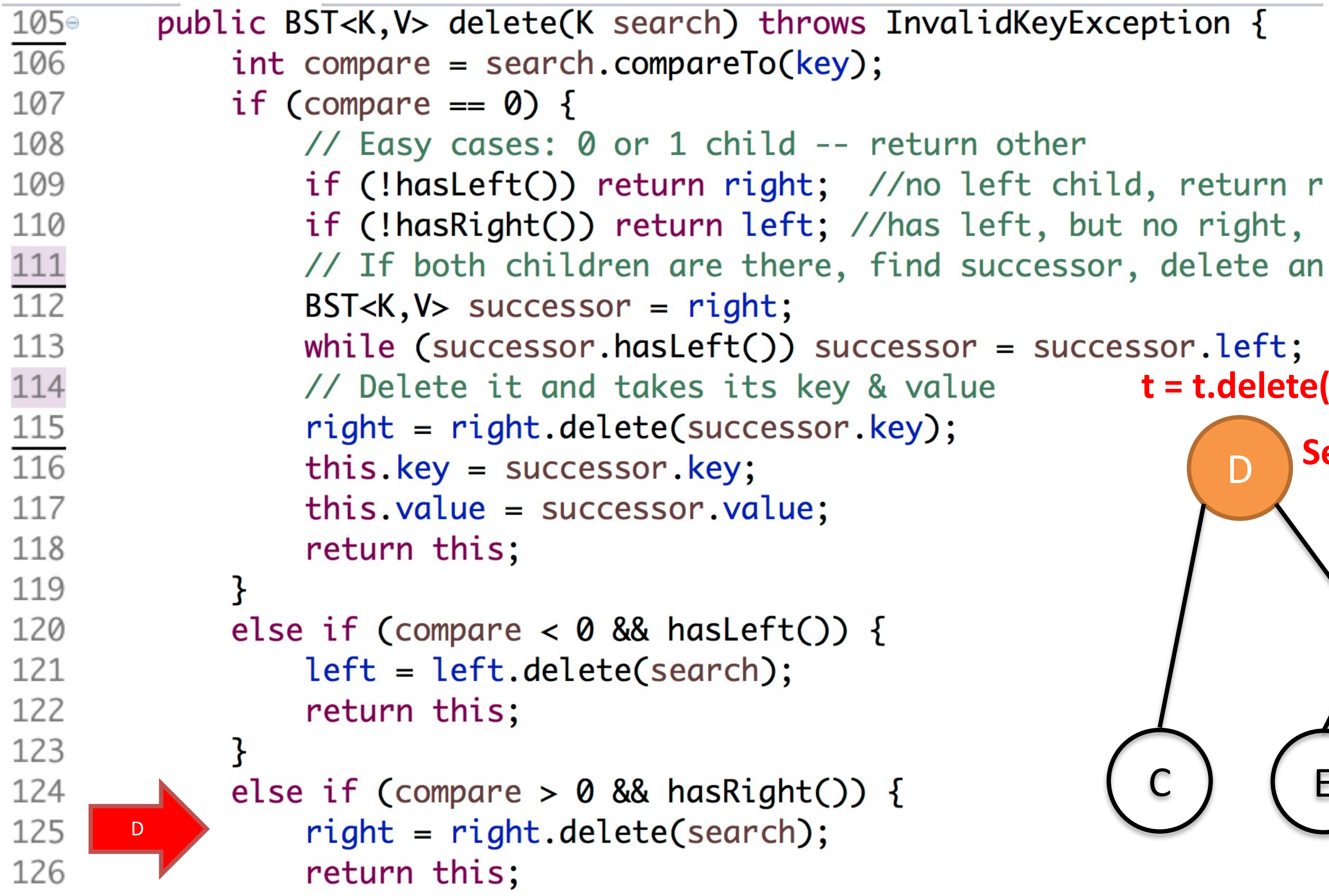
```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

t = Node “D”



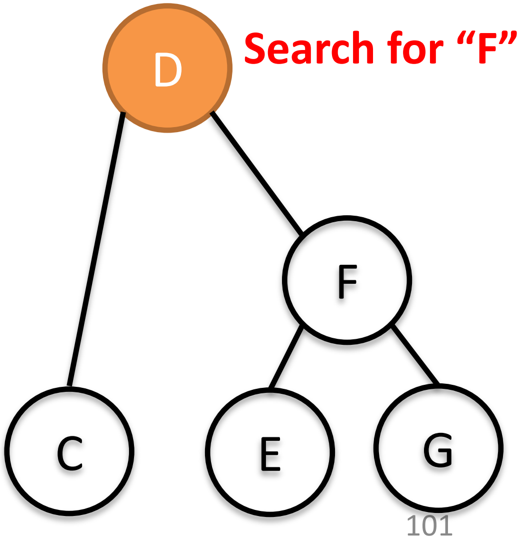
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

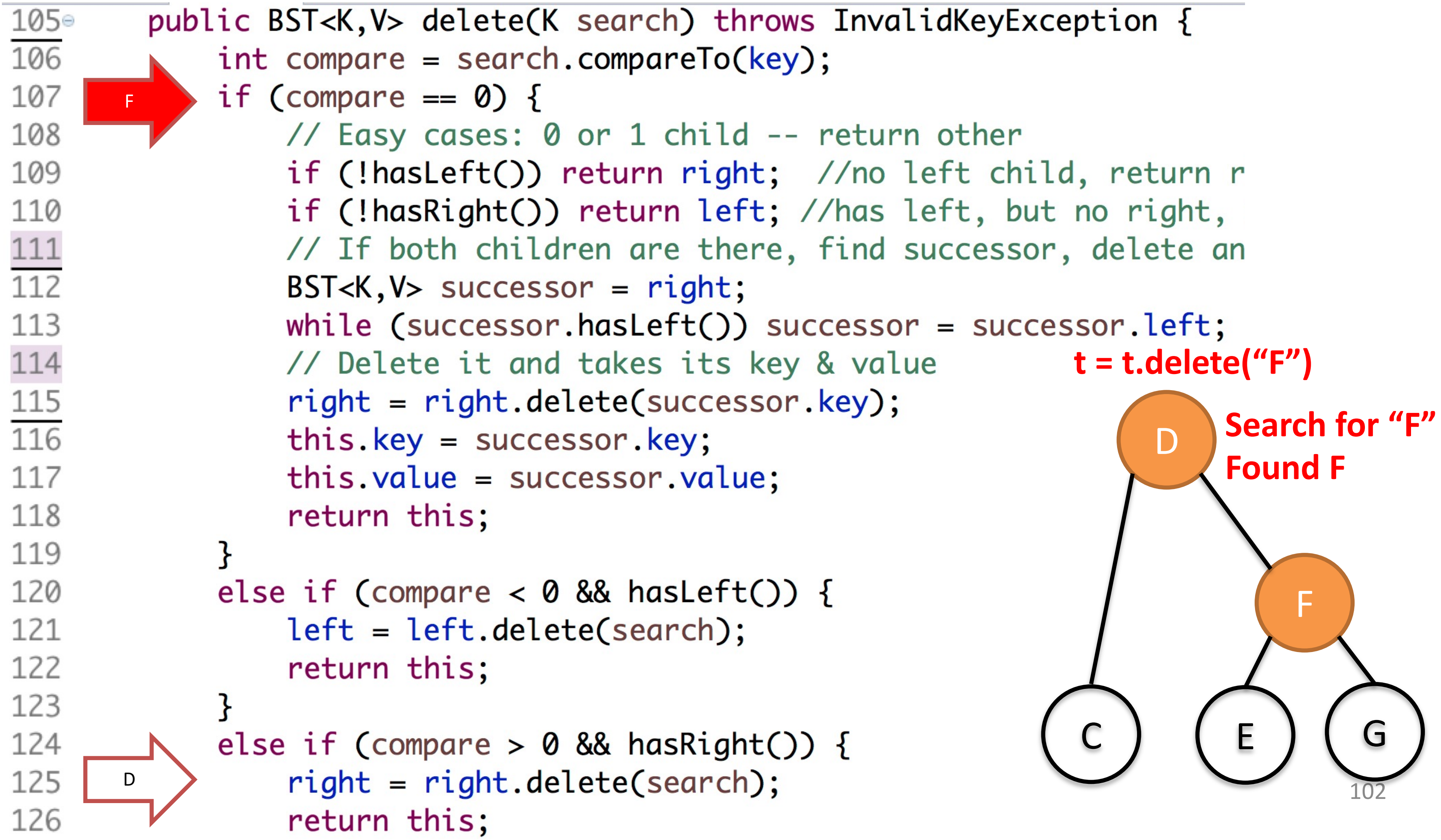
t = t.delete("F")



Search for "F"

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

BST.java



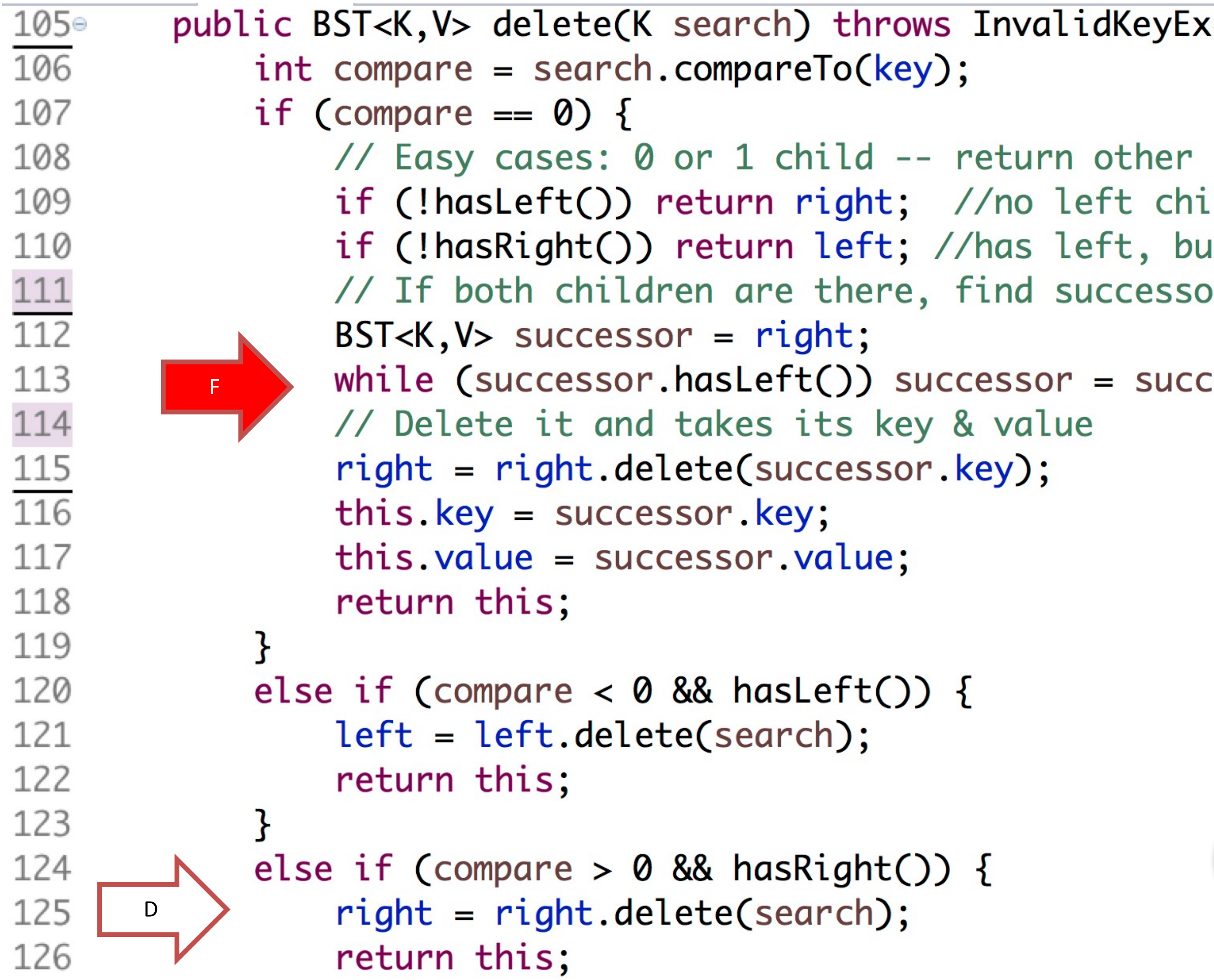
```
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;  //no left child, return r
        if (!hasRight()) return left; //has left, but no right,
        // If both children are there, find successor, delete an
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it and takes its key & value
        right = right.delete(successor.key);
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
```

t = t.delete("F")

Search for "F"
Found F

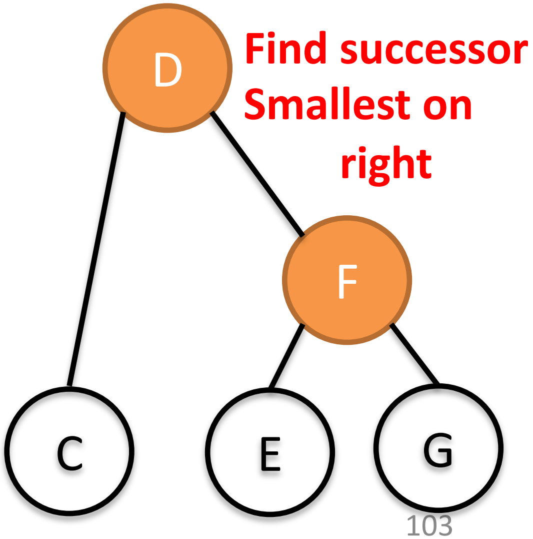
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```java
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

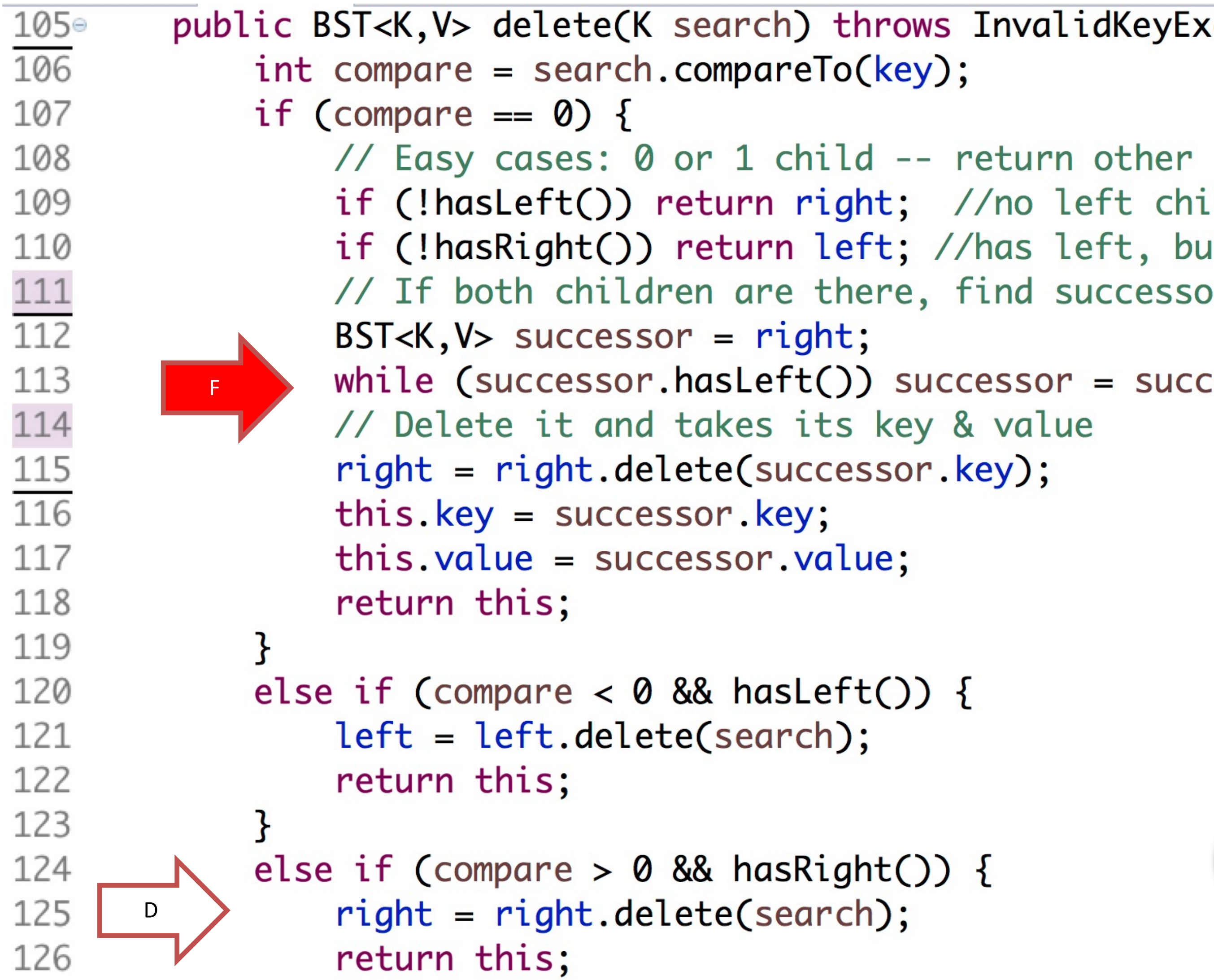
t = t.delete("F")



Find successor Smallest on right

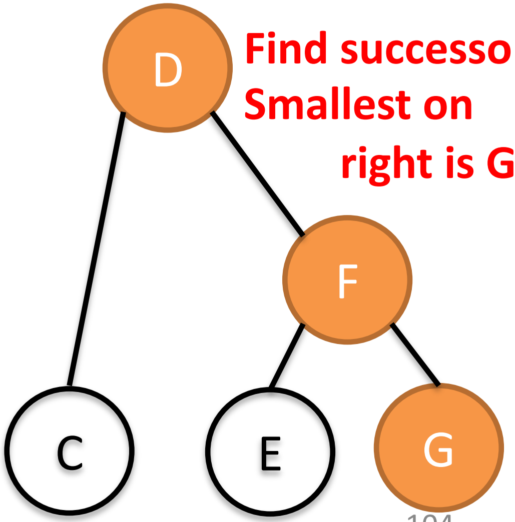
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```java
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

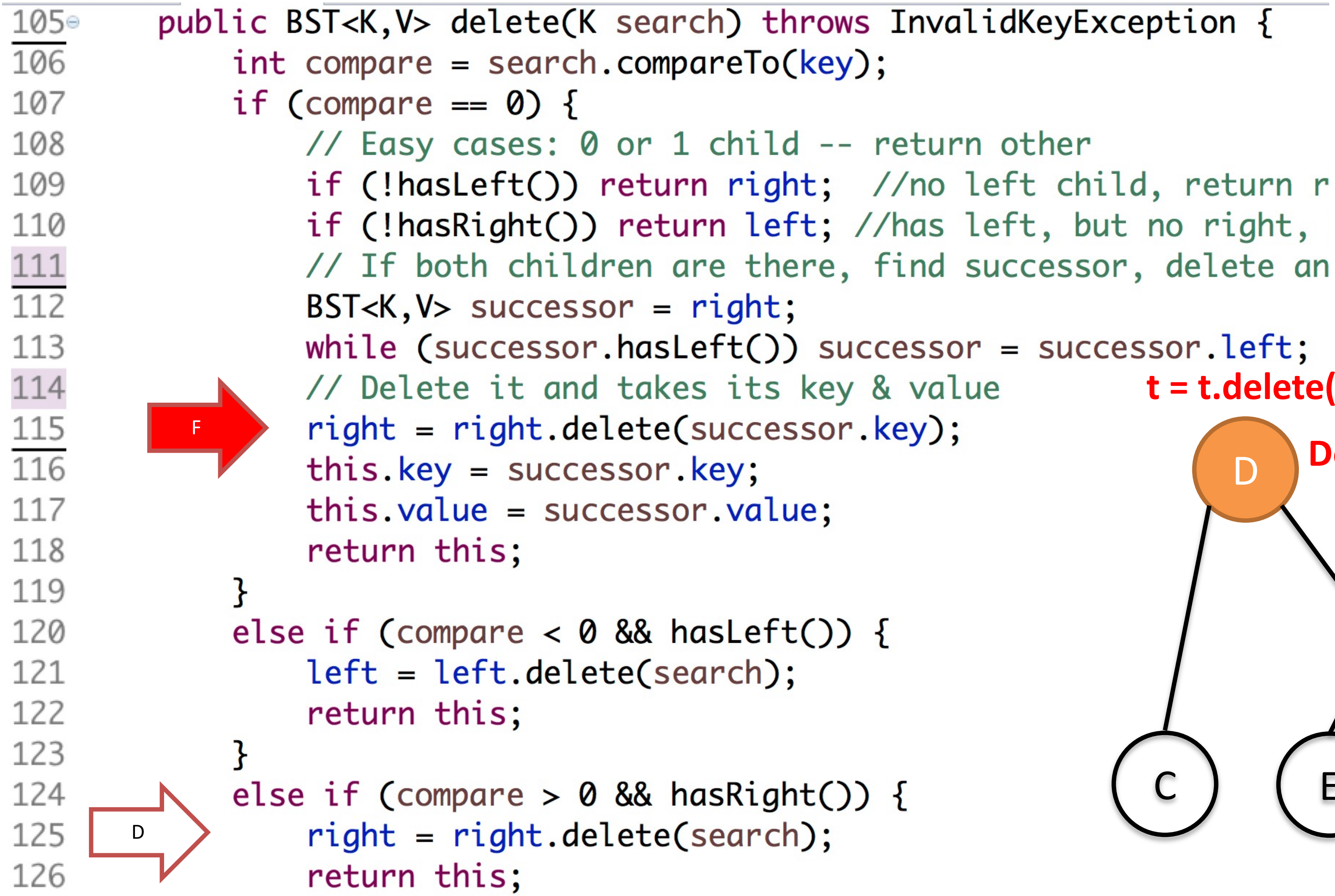
t = t.delete(“F”)



Find successor
Smallest on
right is G

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

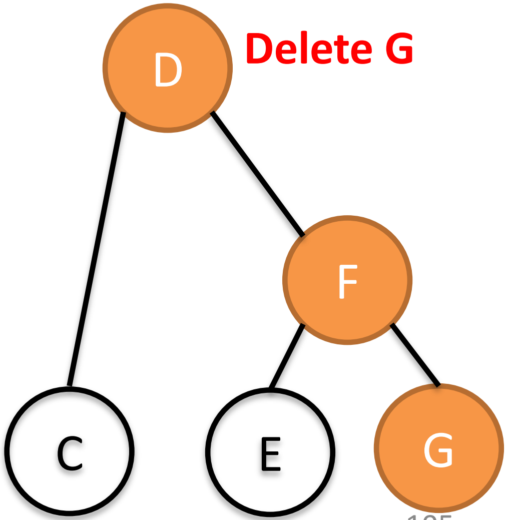
BST.java



```
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;  //no left child, return r
        if (!hasRight()) return left; //has left, but no right,
        // If both children are there, find successor, delete an
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it and takes its key & value
        right = right.delete(successor.key);
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
```

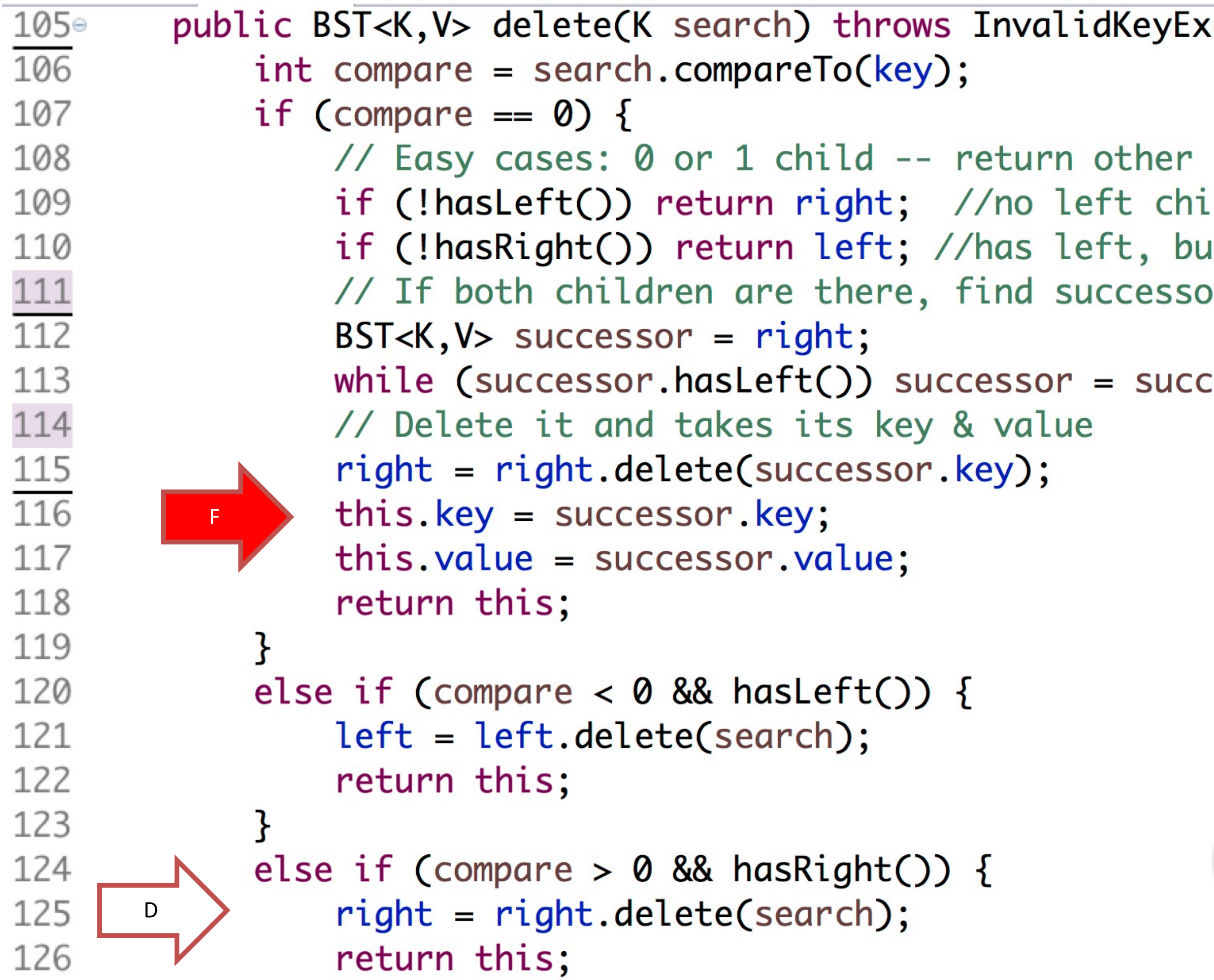
t = t.delete("F")

Delete G



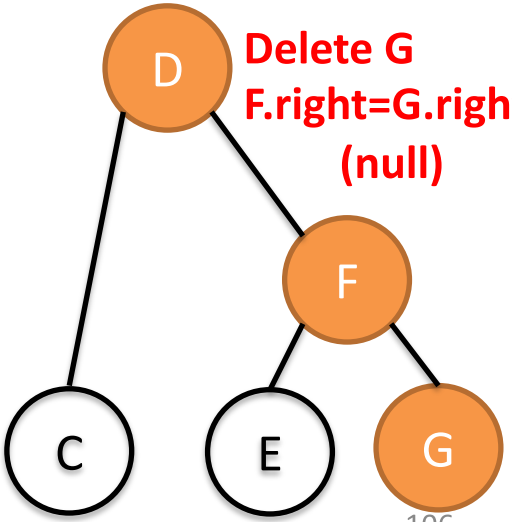
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```java
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

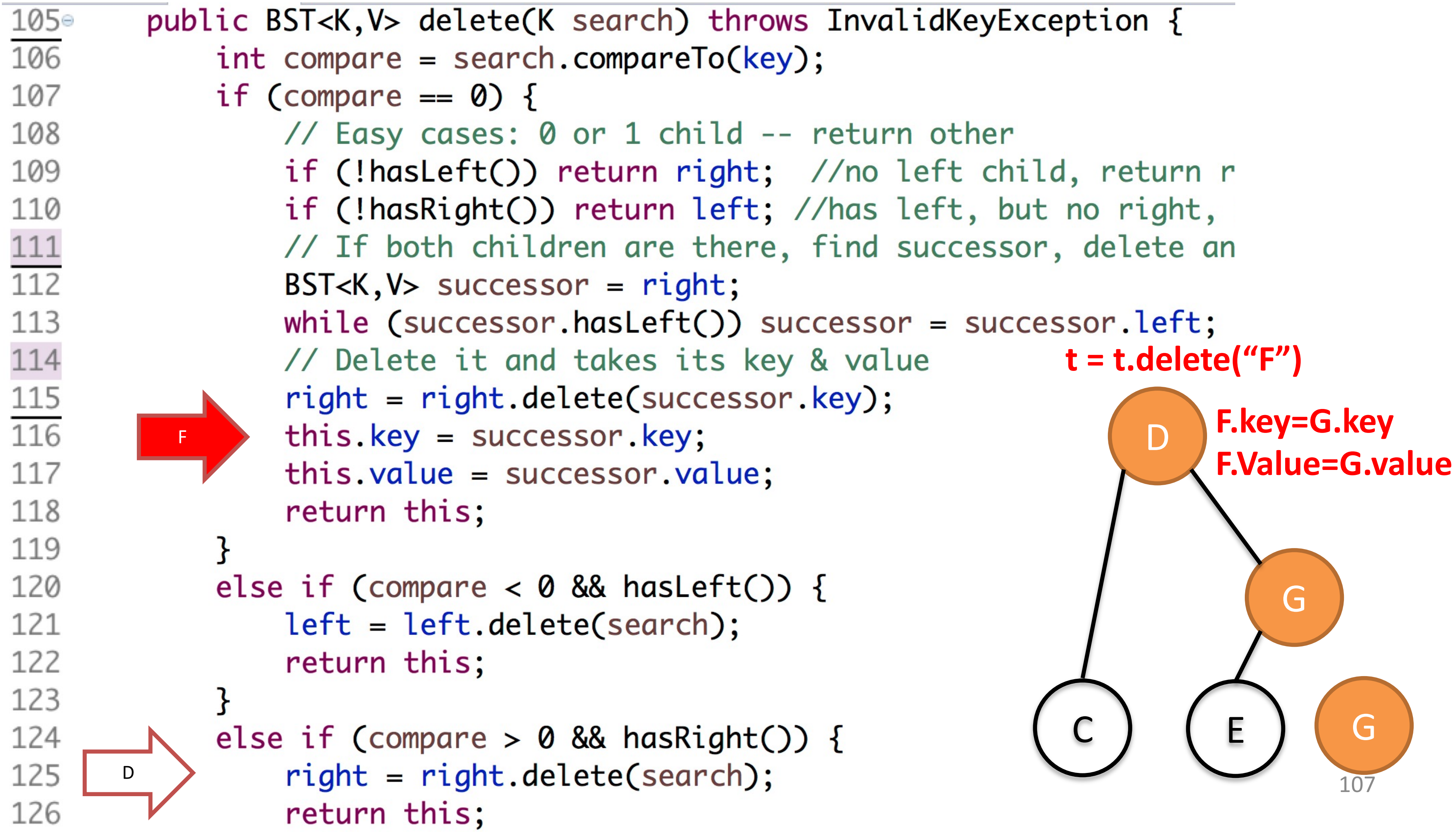
t = t.delete("F")



Delete G
F.right=G.right
(null)

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

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

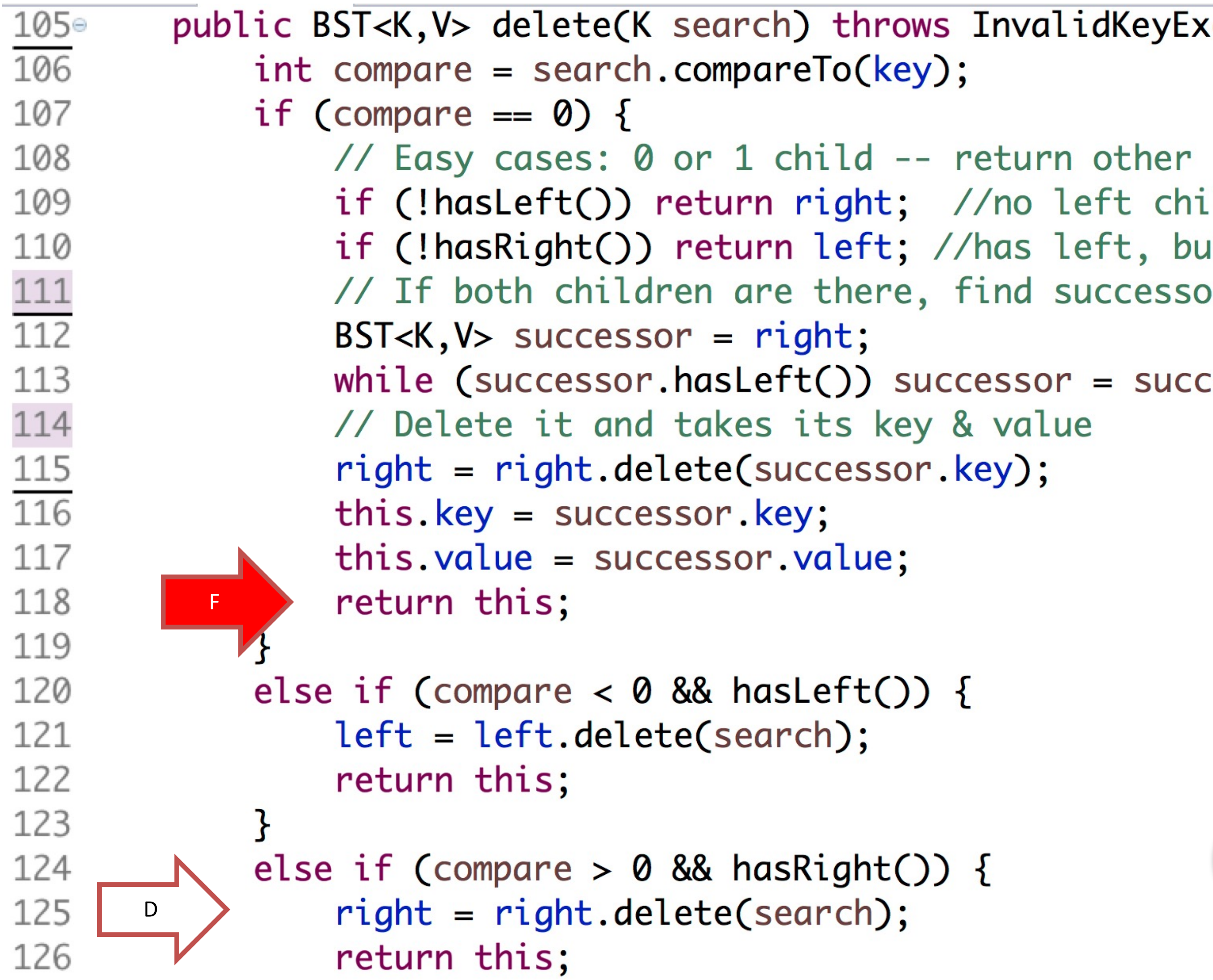
t = t.delete(“F”)

F.key=G.key

F.Value=G.value

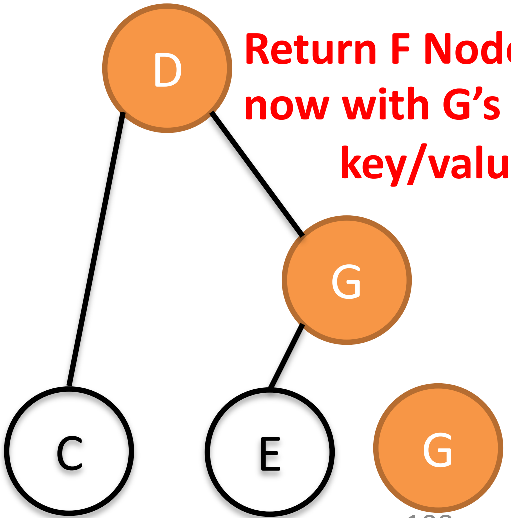
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
    public BST<K,V> delete(K search) throws InvalidKeyException {
        int compare = search.compareTo(key);
        if (compare == 0) {
            // Easy cases: 0 or 1 child -- return other
            if (!hasLeft()) return right;  //no left child, return r
            if (!hasRight()) return left; //has left, but no right,
            // If both children are there, find successor, delete an
            BST<K,V> successor = right;
            while (successor.hasLeft()) successor = successor.left;
            // Delete it and takes its key & value
            right = right.delete(successor.key);
            this.key = successor.key;
            this.value = successor.value;
            return this;
        }
        else if (compare < 0 && hasLeft()) {
            left = left.delete(search);
            return this;
        }
        else if (compare > 0 && hasRight()) {
            right = right.delete(search);
            return this;
```

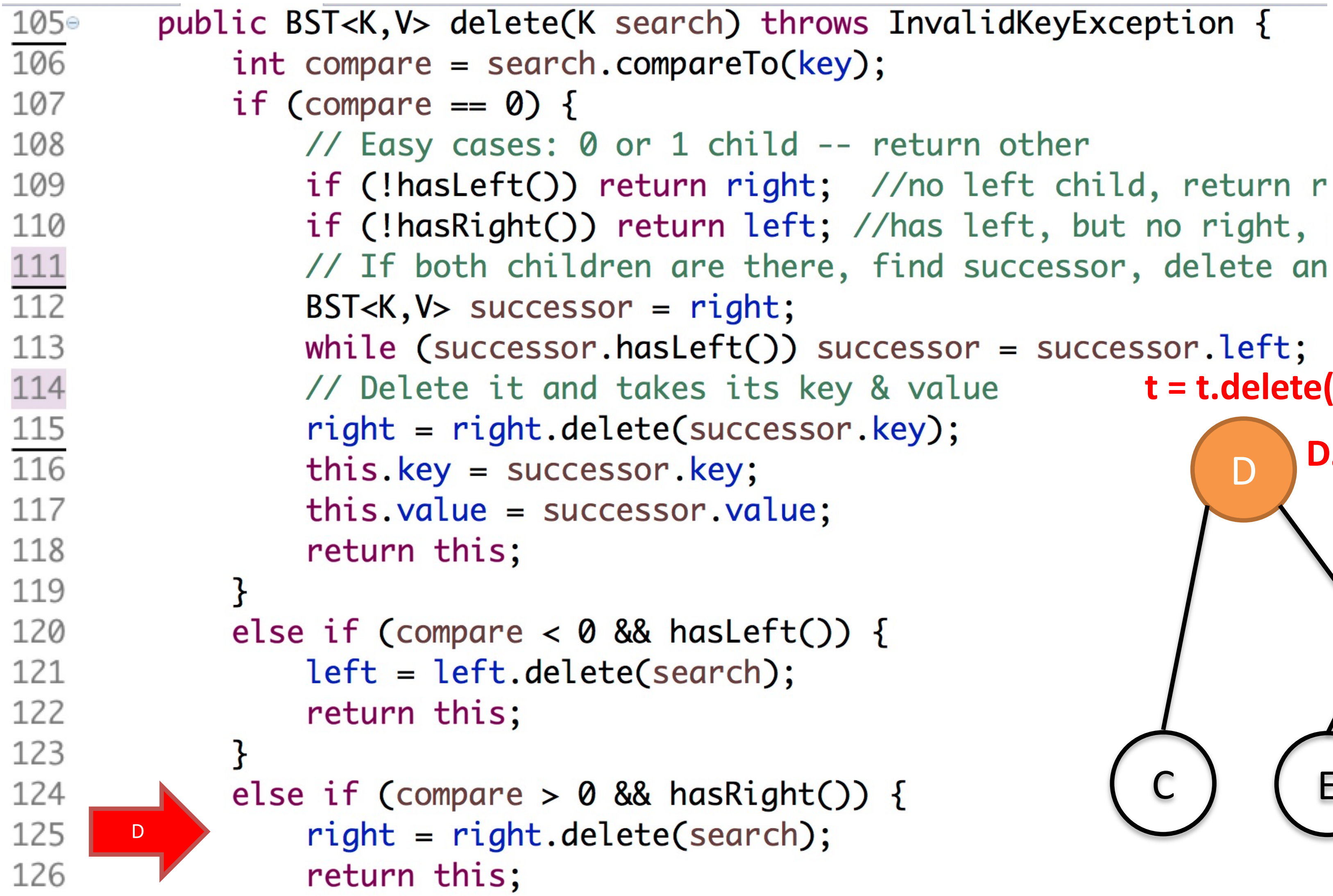
t = t.delete(“F”)



Return F Node now with G’s key/value

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

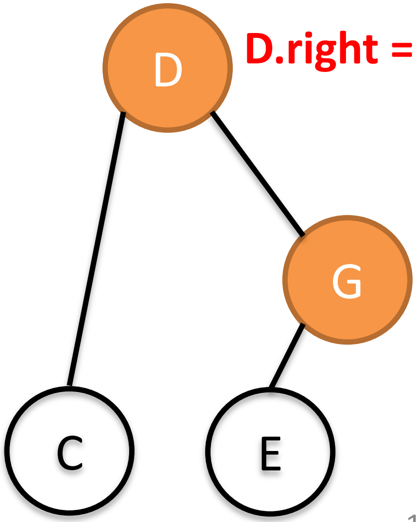
BST.java



```
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;  //no left child, return r
        if (!hasRight()) return left; //has left, but no right,
        // If both children are there, find successor, delete an
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it and takes its key & value
        right = right.delete(successor.key);
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
```

D

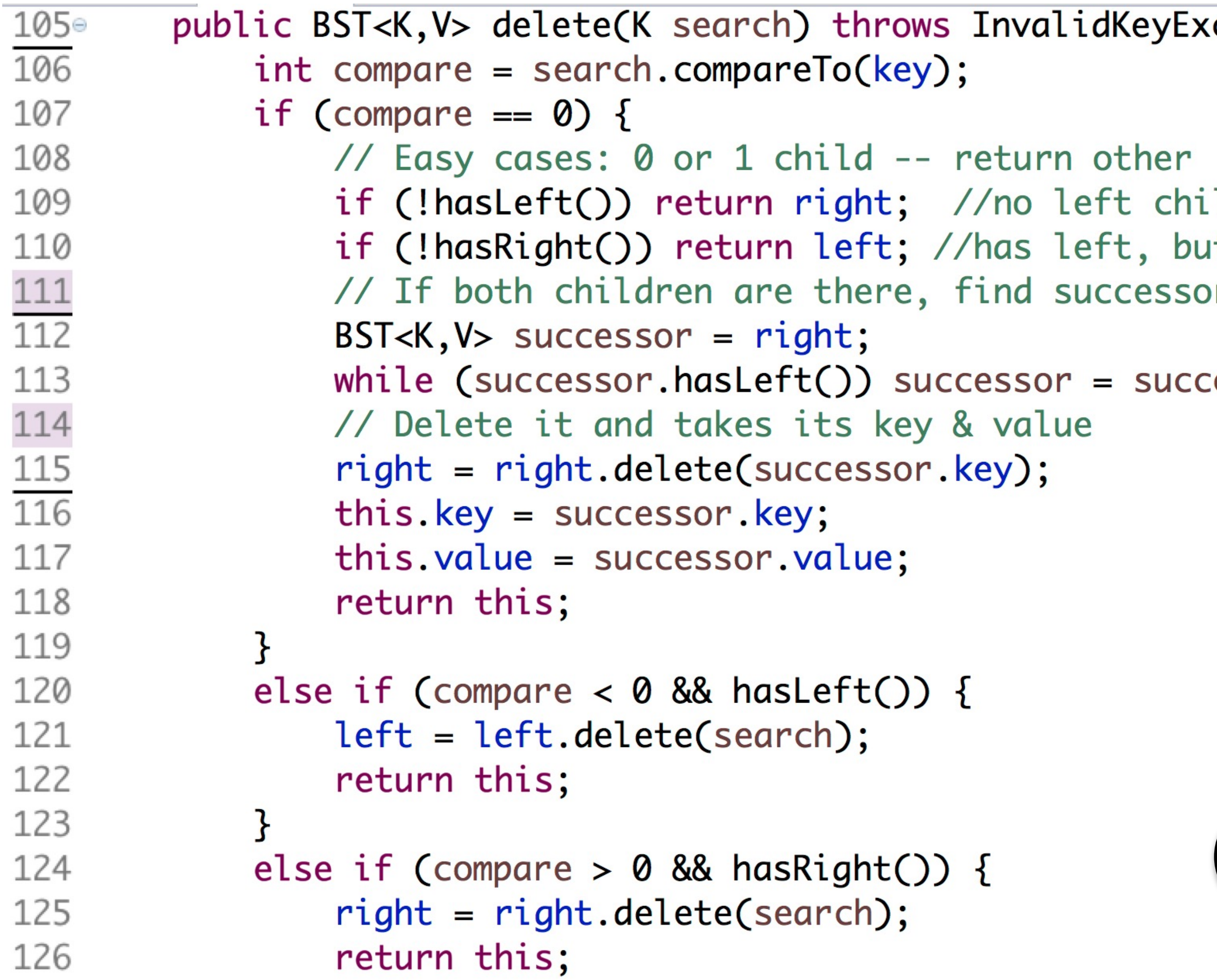
t = t.delete("F")



D.right = G

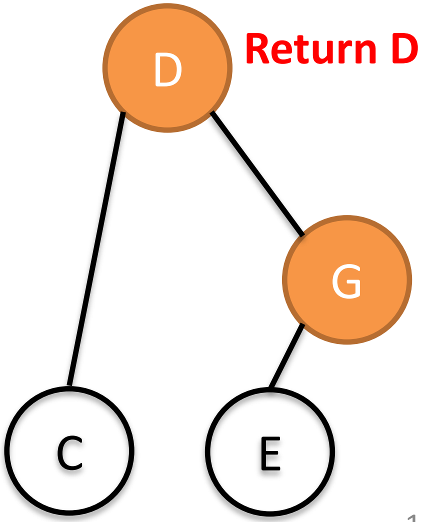
Deleting a Node removes it from the tree and returns updated tree to caller

BST.java



```
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;  //no left child, return r
        if (!hasRight()) return left; //has left, but no right,
        // If both children are there, find successor, delete an
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it and takes its key & value
        right = right.delete(successor.key);
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
```

t = Node "D"



Return D