CS 10: Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

Day 9 – Hierarchies Part 2

Agenda

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

Binary search can quickly find items if the data is ordered

Binary search on an array

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
      array[idx] > target
            max = idx-1
      else
            min = idx +1
}
```

At each iteration half of the indexes are eliminated

Binary search on an array

1

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

Min

Max

Pseudo code

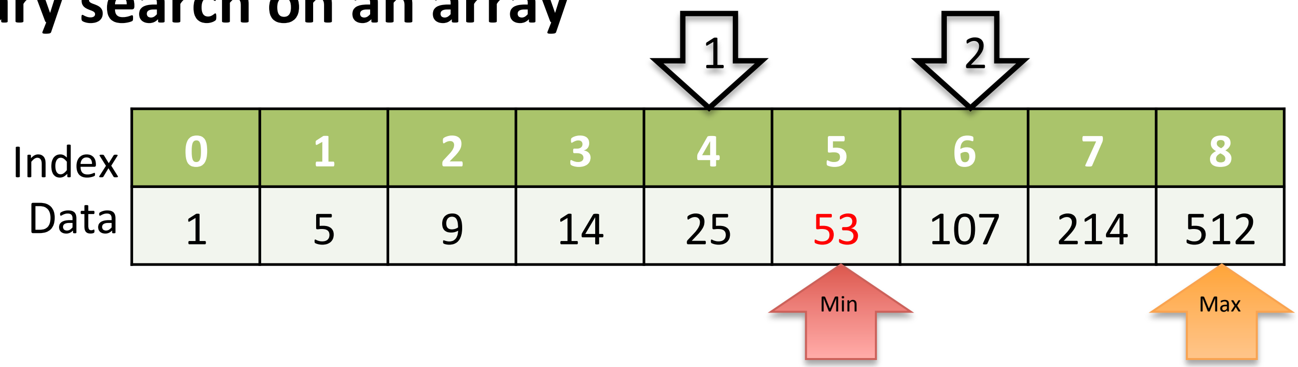
Looking for target = 53

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

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

At each iteration half of the indexes are eliminated

Binary search on an array



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

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

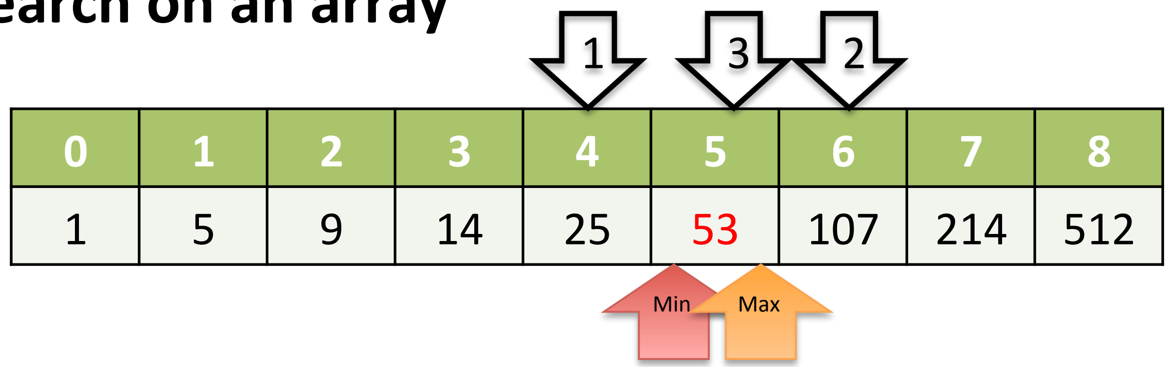
Pseudo code

Looking for target = 53

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

Binary search finds data generally faster than linear search

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

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

Pseudo code

Looking for target = 53

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

Binary vs. linear search

- Binary found item in 3 tries
- Linear search would have taken 6 tries
- On large data sets binary search can make a *huge* difference

We can extend binary search to find a key and return a value

Key is Student ID, Value is student name

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

"Alice" "Charlie" …

"Bob"

Implications

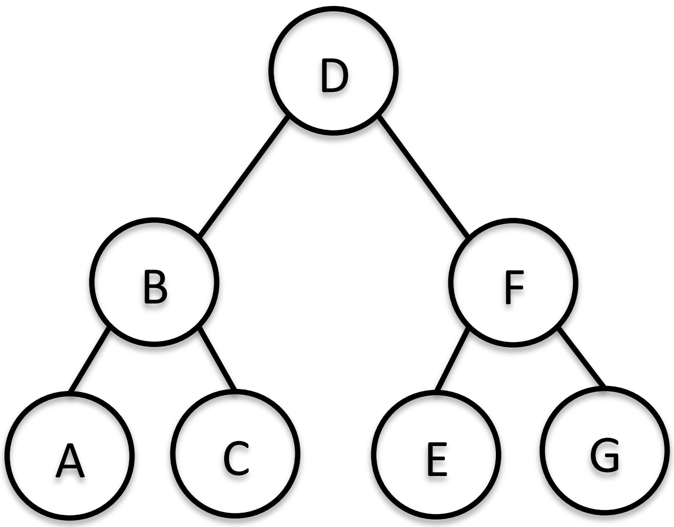
- Given a Student ID, can quickly find the student's name
- Each entry must have a key and a value
- Value is an object (e.g. String or student record)
- Of course the keys must be sorted
- How do we do that?

Agenda

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

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

Keys sorted in Binary Search Tree



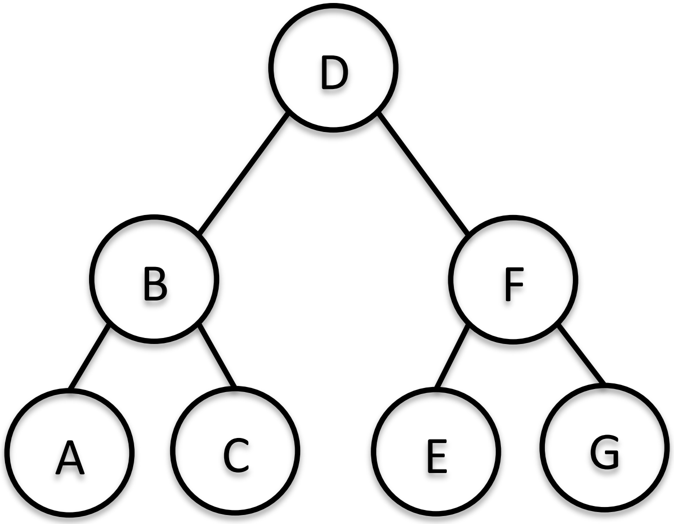
Binary Search Tree property

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

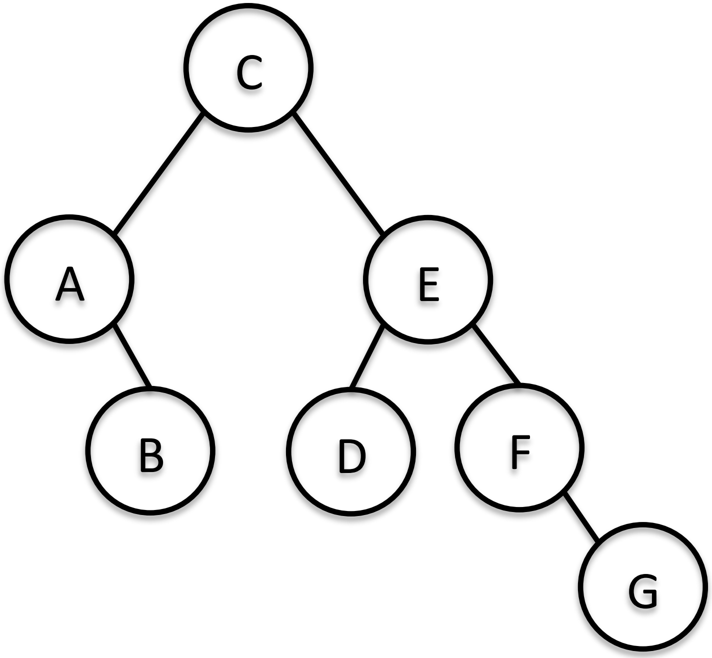
BSTs with same keys could have different structures and still obey BST property

Two valid BSTs with same keys but different structure

Tree 1

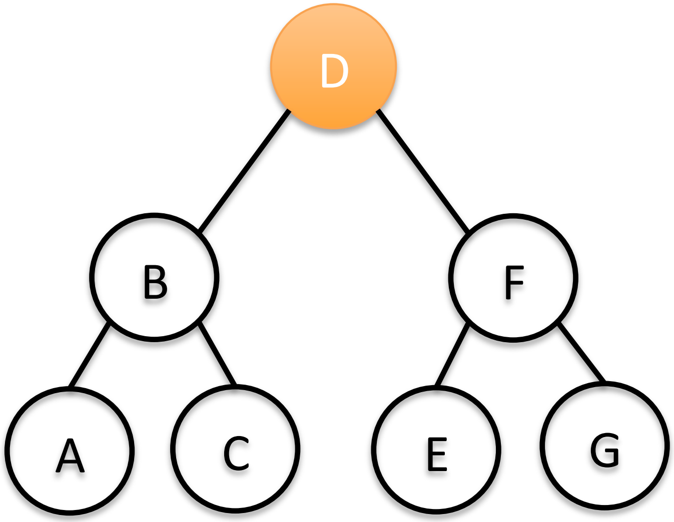


Tree 2



BSTs make searching fast and simple

Find Key "C"

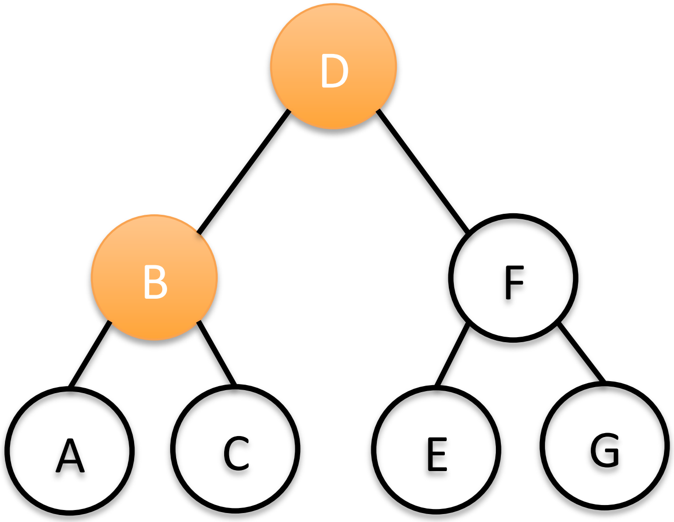


Search process

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

BSTs make searching fast and simple

Find Key "C"



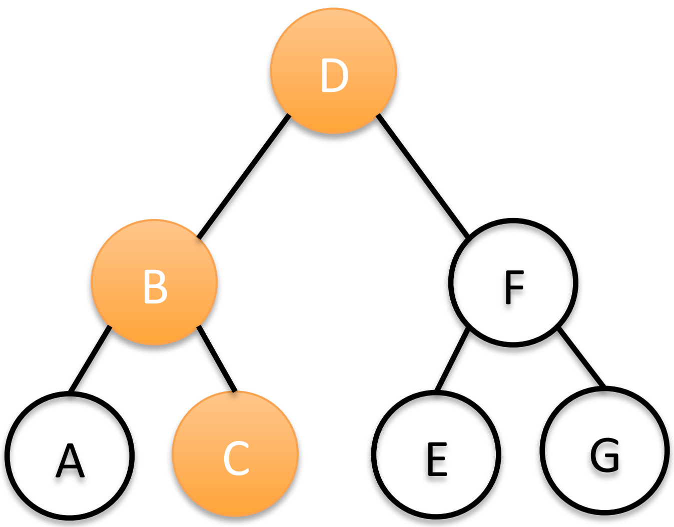
Search process

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

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

BSTs make searching fast and simple

Find Key "C"



Search process

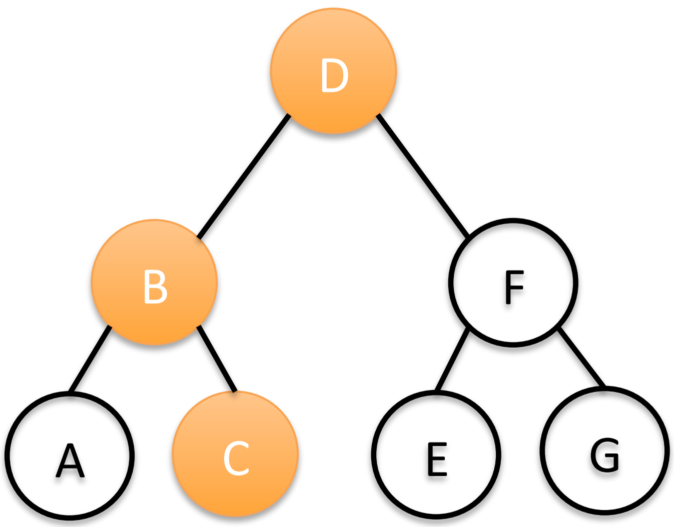
- Check root
- "D" > "C", so go left

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

- Check "C"
- Yahtzee! Found it

BSTs make searching fast and simple

Find Key "C"



Search process

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

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

- Check "C"
- Yahtzee! Found it

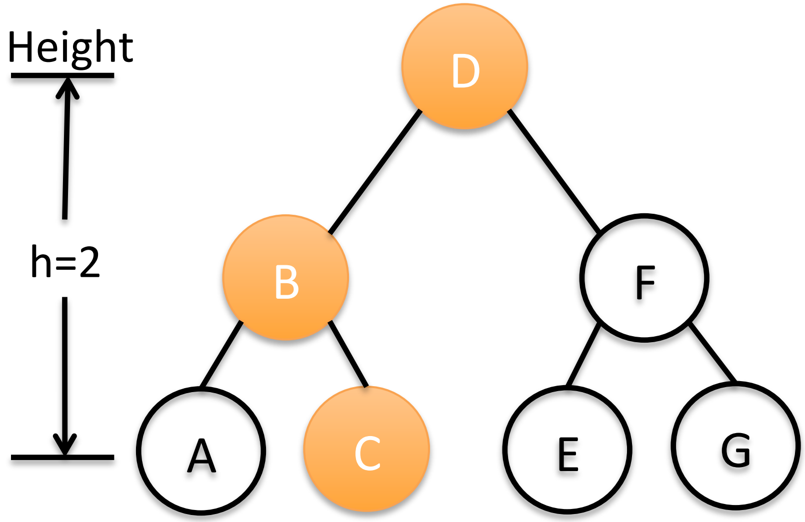
- Would know by now if key not in BST

Agenda

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

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

Find Key "C"

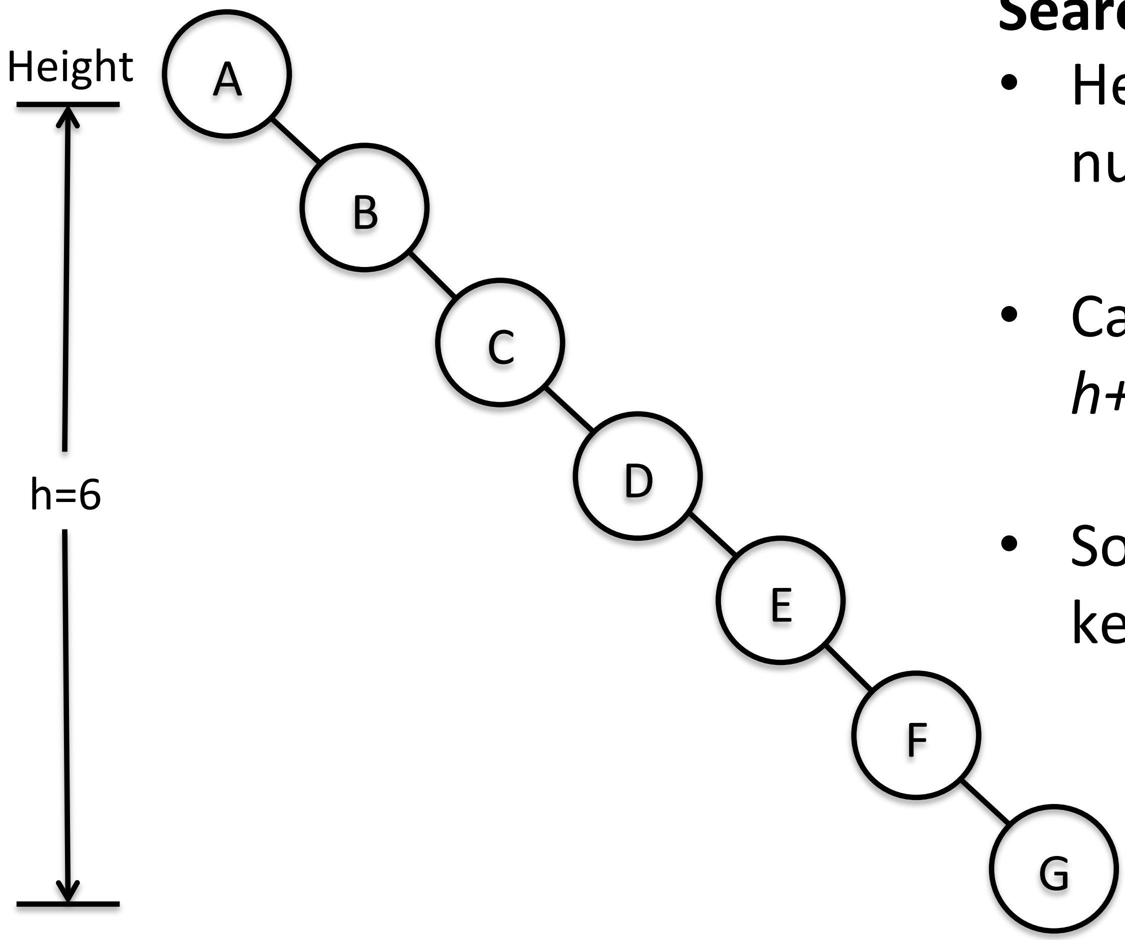


Search process

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

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

Find Key "G"



Search process

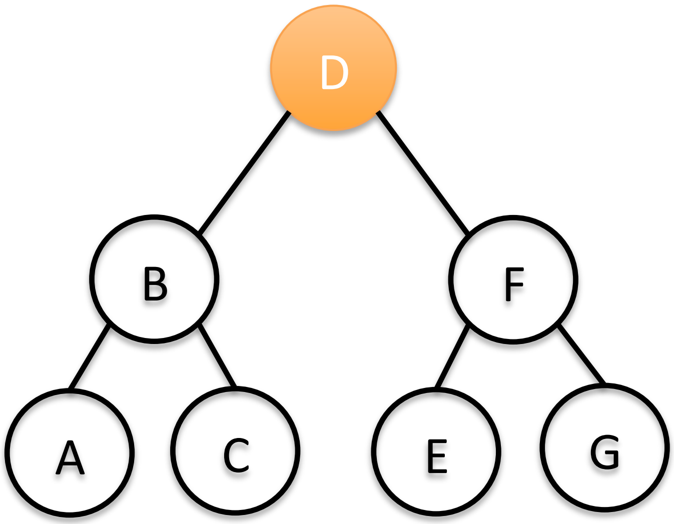
- Height *h = 6* (count number of edges to leaf)
- Can take no more than *h+1* checks, O(h)
- Soon we will see how to keep trees balanced

Agenda

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

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

Inserting new node with key H

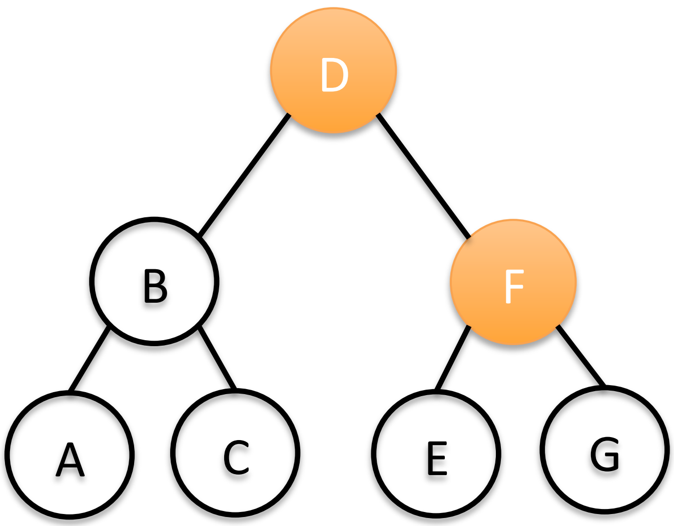


Comments

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

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

Inserting new node with key H

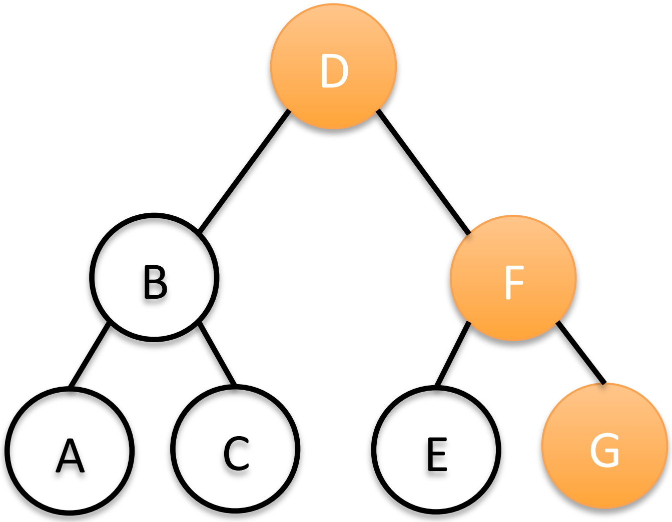


Comments

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

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

Inserting new node with key H



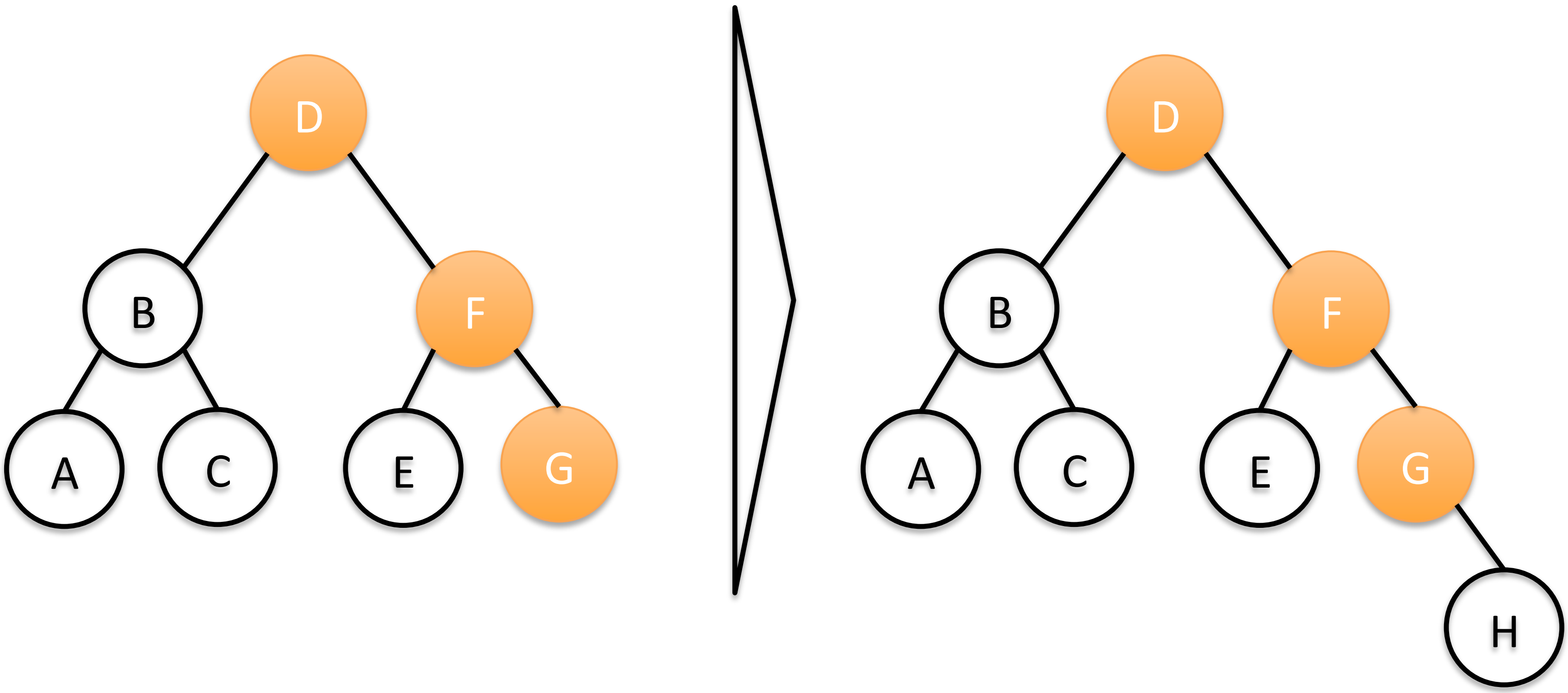
G is a leaf, add new node

Comments

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

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

Inserting new node with key H

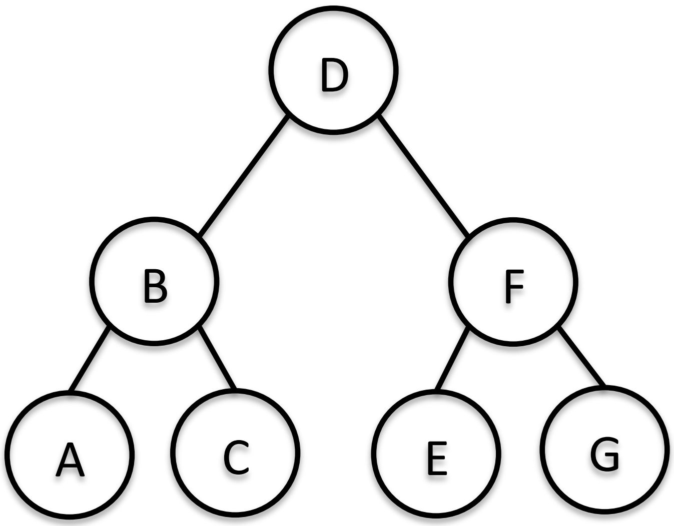


Comments

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

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

Deleting node A (no children)

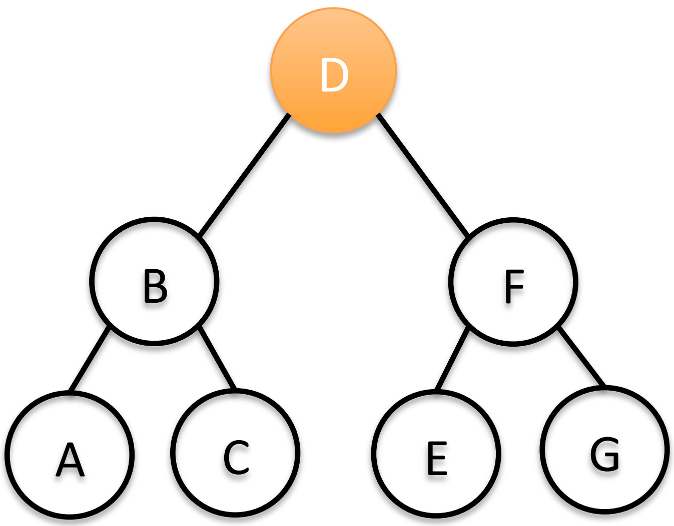


Comments

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

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

Deleting node A (no children)



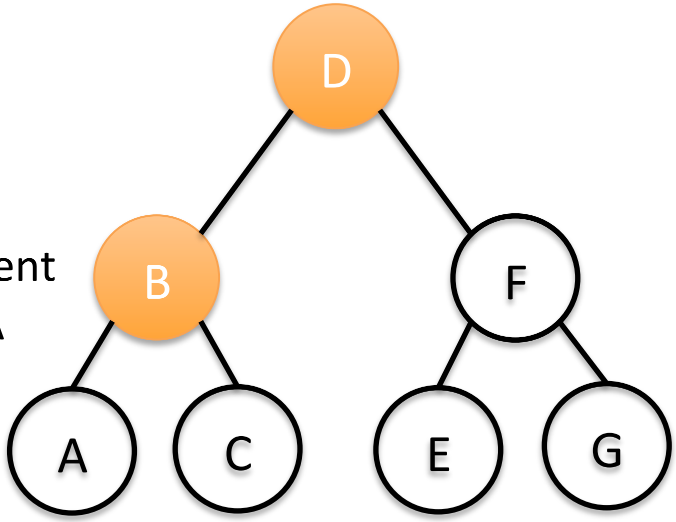
Comments

- 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

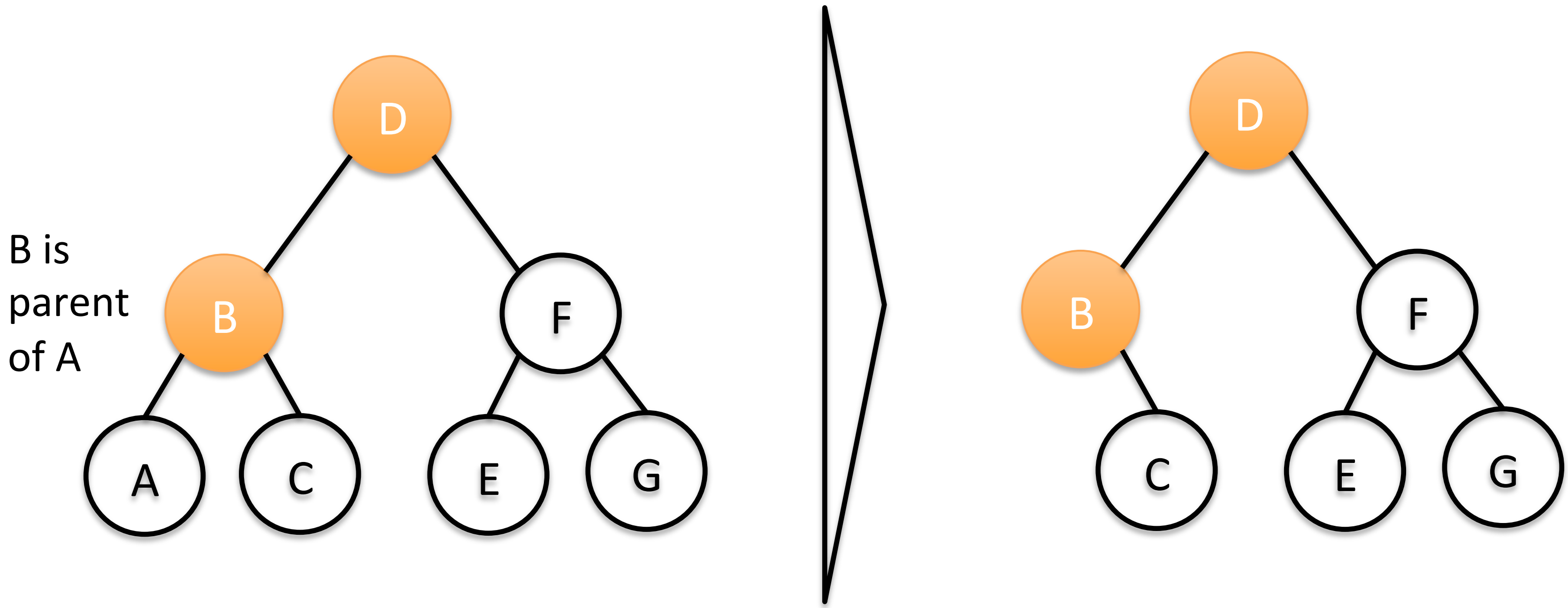


Comments

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

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

Deleting node A (no children)

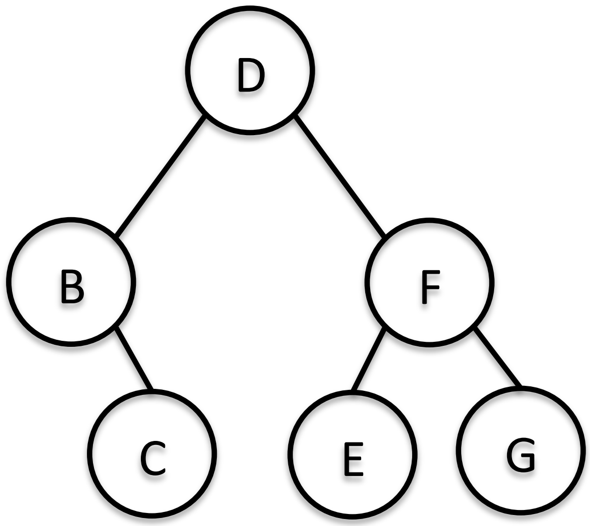


Comments

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

Deleting with one child is not difficult

Deleting node B (1 child)



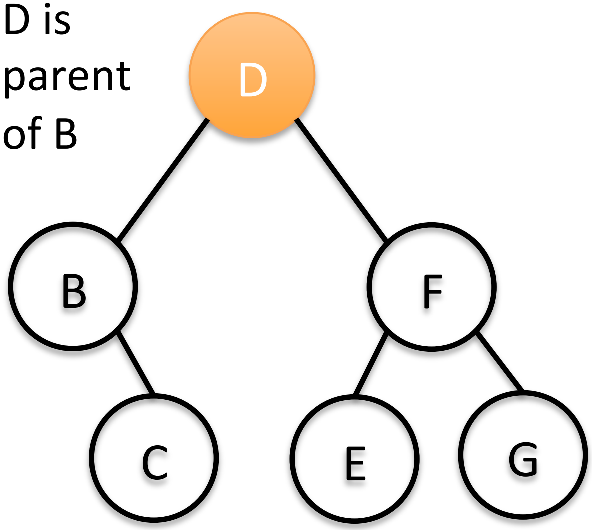
Comments

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

Deleting with one child is not difficult

Deleting node B (1 child)

D is
parent
of B

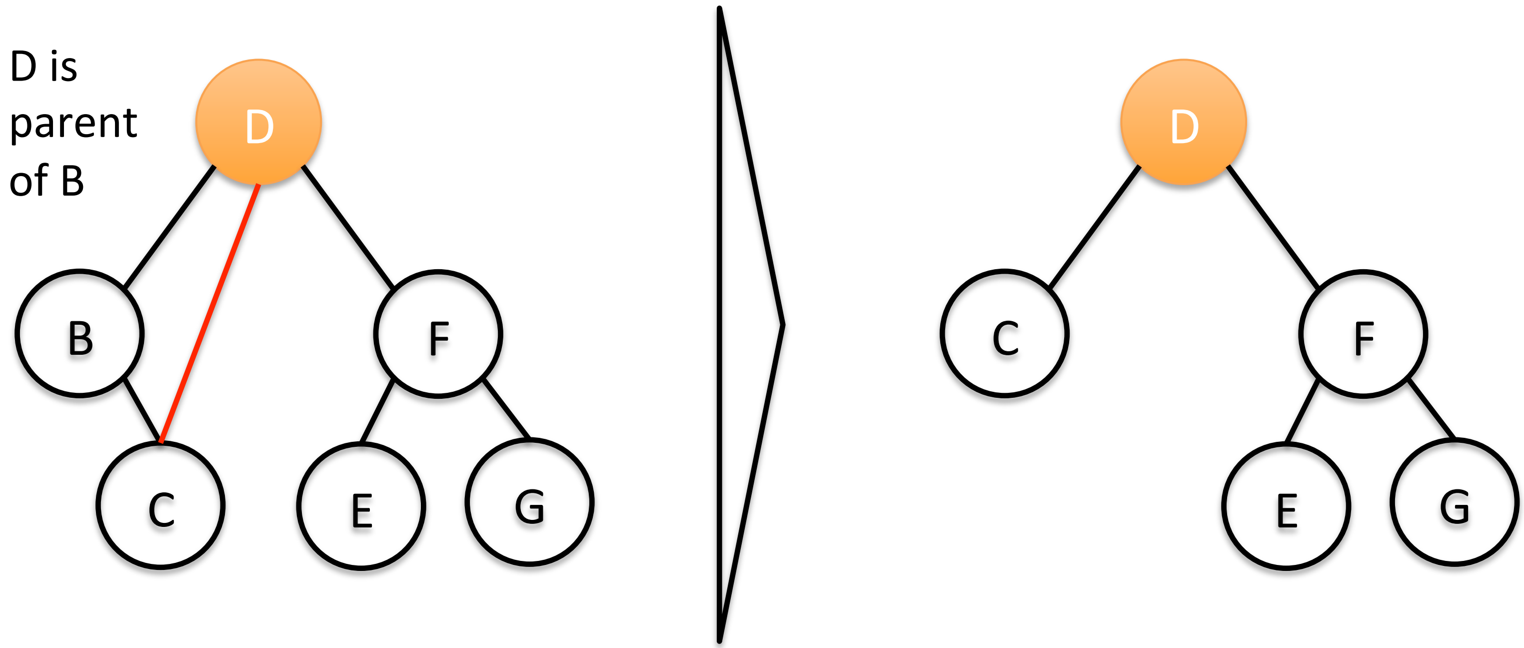


Comments

- 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)

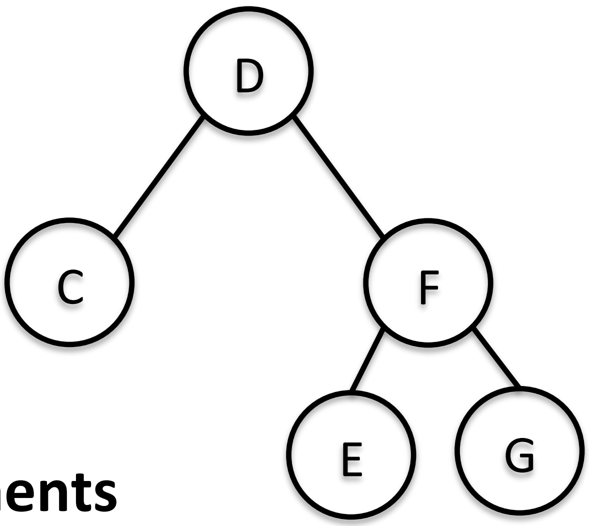


Comments

- 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)

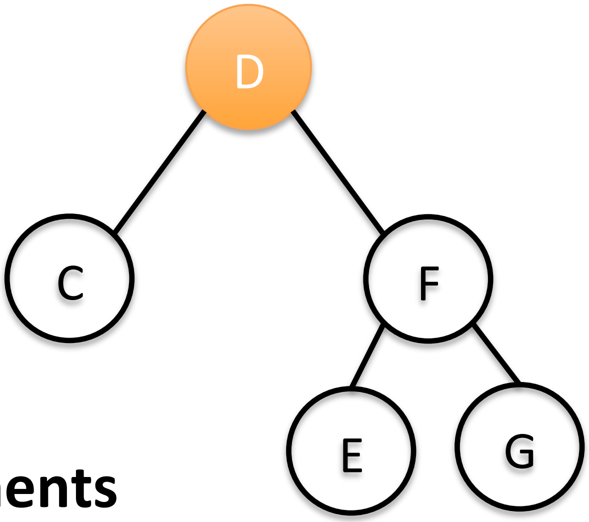


Comments

- 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 recurse 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)

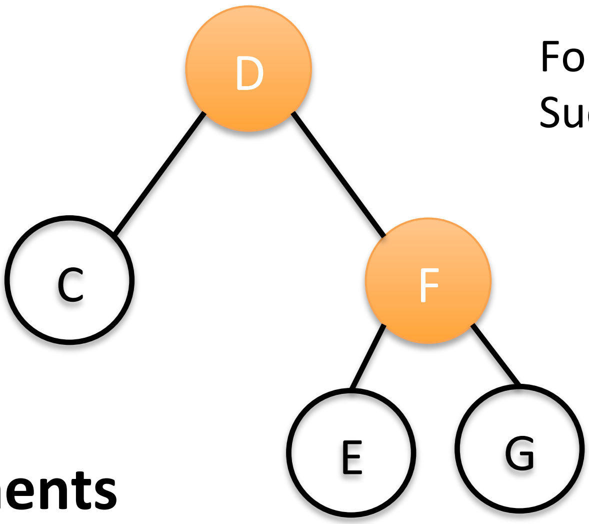


Comments

- 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 recurse 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)



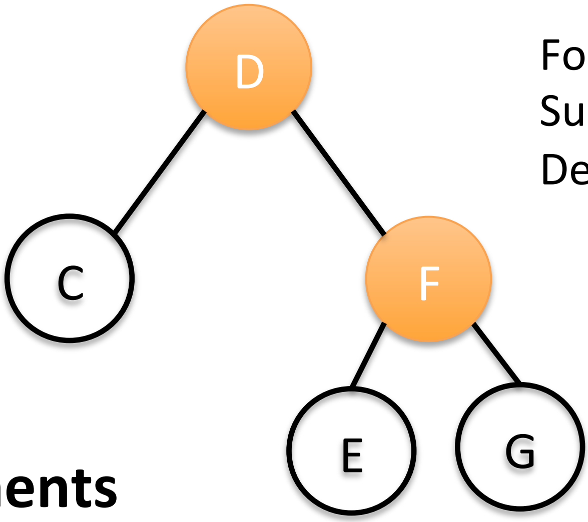
Found F
Successor is smallest on right (G here)

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 recurse 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)



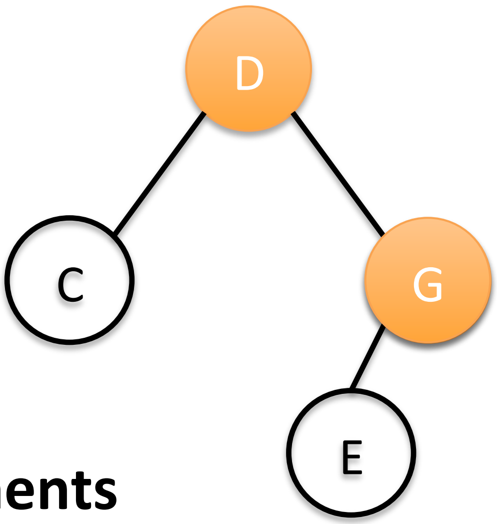
Found F
Successor is smallest on right (G here)
Delete successor

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 recurse 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 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 recurse down right child's left descendants
- Delete successor, but save successor's key and value
- Replace F with key and value of successor

Agenda

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

Binary Search Tree nodes each take a key and value, also have left and right children

Binary Search Tree declaration

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

Comments

- Key could be a String (e.g., name) and value could be BufferedImage (e.g., mugshot), both could be integers (key is zip code, value is population), depends on use case
- Remember, generics need to be objects, so use wrapper classes for primitives such as Integer and Double

We need a way to compare nodes, so the Key must implement a Comparable

Extending Comparable interface

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

Comments

- Comparable must implement compareTo(K compareKey) method
- compareTo() built in for primitive types and wrappers (e.g., String), no need to implement ourselves if Key is String
- compareTo() returns 0 if node and compareKey are the "equal"
- Return -1 if node's key < compareKey
- Return 1 if node's > compareKey

BSTs make searching fast and simple

BST.java

- Constructors set up trees as expected, just like last class in BinaryTree.java
- *public V find(K search)*
  - Compare *search* with node's key
  - If match then return node's value
  - If (compare < 0 && hasLeft()) return left.find(search)
  - If (compare > 0 && hasRight()) return right.find(search)
  - Throw exception if key not found (wasn't match and no left or right children)
- *public void insert(K key, V value)*
  - Search key
  - If key found, replace value
  - else, insert as leaf

Code to delete nodes

BST.java

- *delete(K search)* at line 105
- Compare *search* to this node's key
- If node's key < *search*, set *left = left.delete(search)*, return *this*
- If node's key > *search*, set *right= right.delete(search)*, return *this*
- If keys are the same
  - If node has one child, return child
  - If node has two children
    - Find successor (smallest on right), may have to recurve right child's left children
    - Delete successor, but save key and value
    - Set this node's key and value to successor's key and value
    - Return this node

Can find min value in tree recursively or in a loop

BST.java

Only need to traverse down left side, so like a linked list

- min() (line 66)
  - If left != null, return left.min()
  - Else return key (this will be the left most, smallest key)

- minIter() (line 74)
  - Start from current node
  - While (curr.left != null) curr = curr.left
  - Return curr.key