CSE 12: PA7

2-18-21

Focus: PA7, BST & File System Filtering

Overview of PA7

- Part I: An Implementation of `DefaultMap`
  - Given the interface `DefaultMap`, implement `BST.java`
  - Descriptions are given in the interface
- Part II: FileSystem Filter
  - Similar to PA6, you will create a file system that has specific methods for filtering through the files
- Part III: Gradescope Questions
- Style

Binary Search Trees (BSTs)

Binary Search Tree (BST)

Binary Tree

- Each node has at most 2 children (left child, right child)
  - Node Class
    - Key
    - Value
    - Left child
    - Right child

```
class Node<K,V> {
  K key;
  V value;
  Node<K,V> left, right;
  public Node(K key, V value,
              Node<K,V> left, Node<K,V> right) {
    this.key = key;
    this.value = value;
    this.left = left;
    this.right = right;
  }
}
```

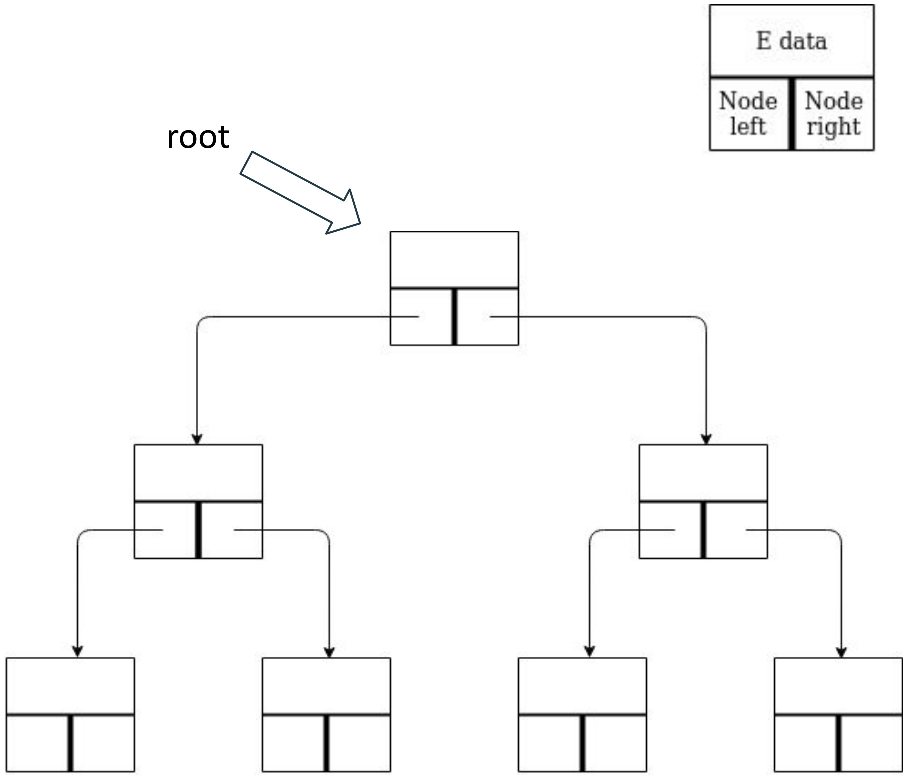
BST

- Nodes belonging to the Left subtree have keys less than the parent node key, and nodes belonging to the right subtree have keys greater than the parent node key

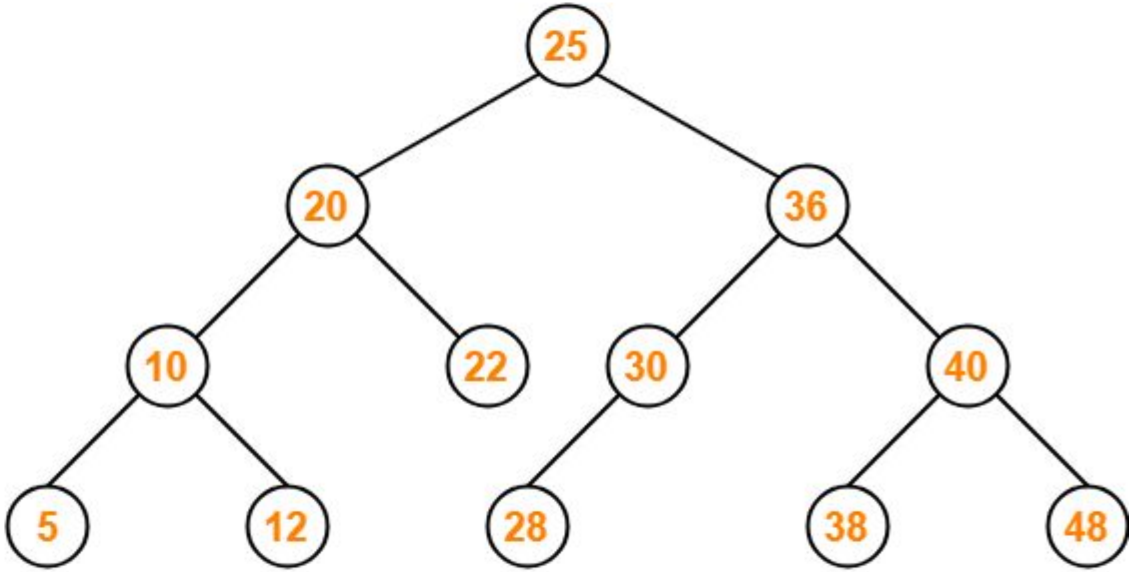
Helpful Terminology

Terminology

- **Root**: the top node in the tree
- **Leaf**: a node with no children
- **Breadth:** the number of leaves
- **Depth:** the distance between a node and the root
- **Level:** 1 + depth
- **Subtree:** a tree of node T and all of its descendants
- **Edge:** the connection between one node and another
- **Height**: number of edges on the longest path from the root to a leaf

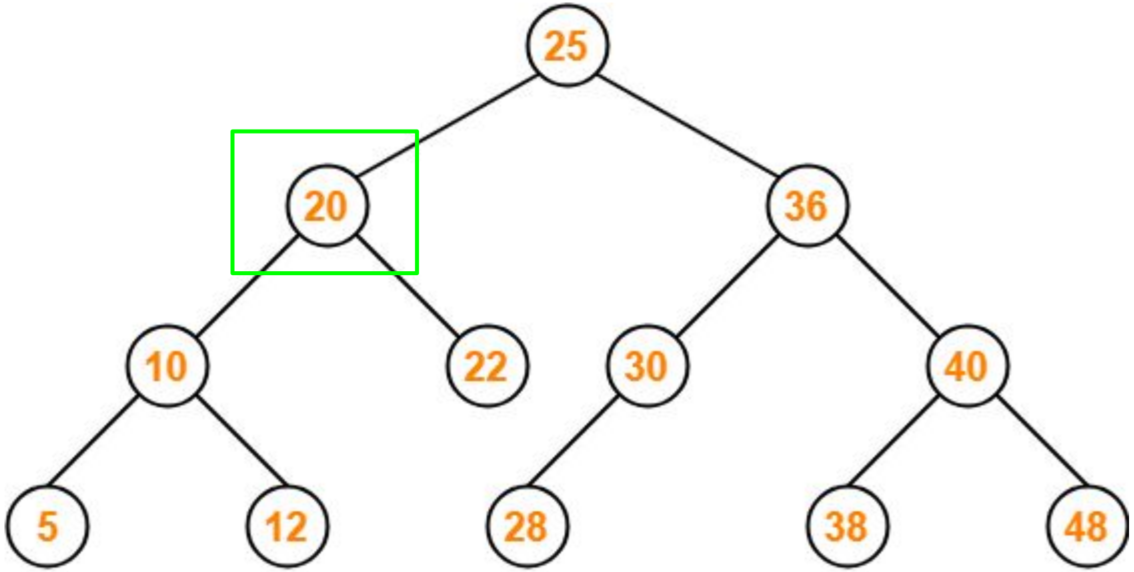


Example BST



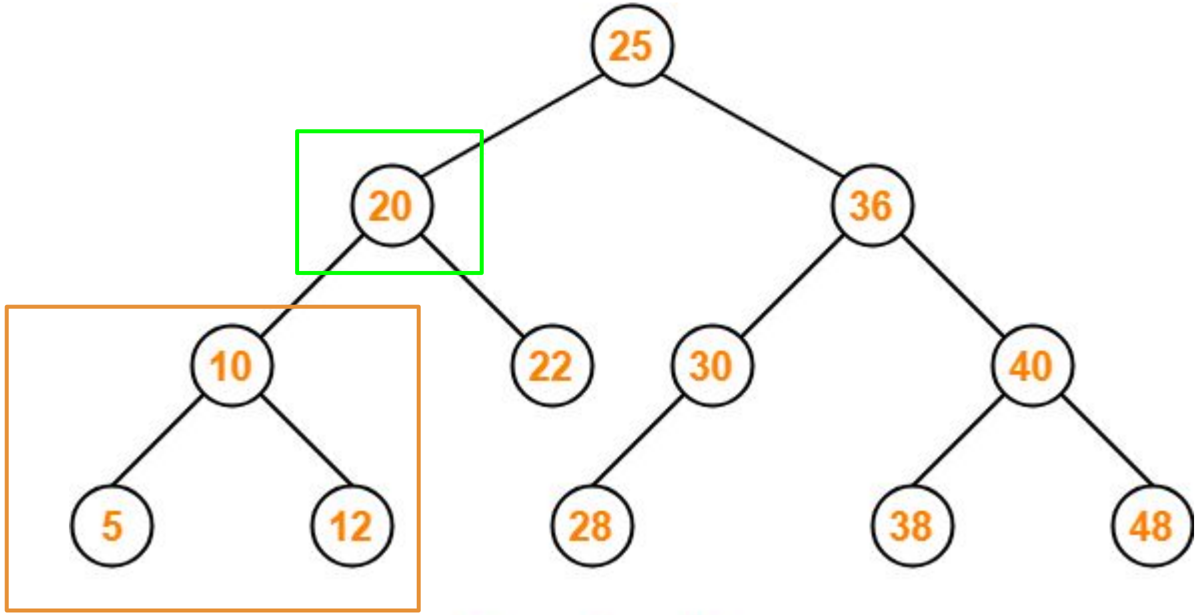
Binary Search Tree

Example BST



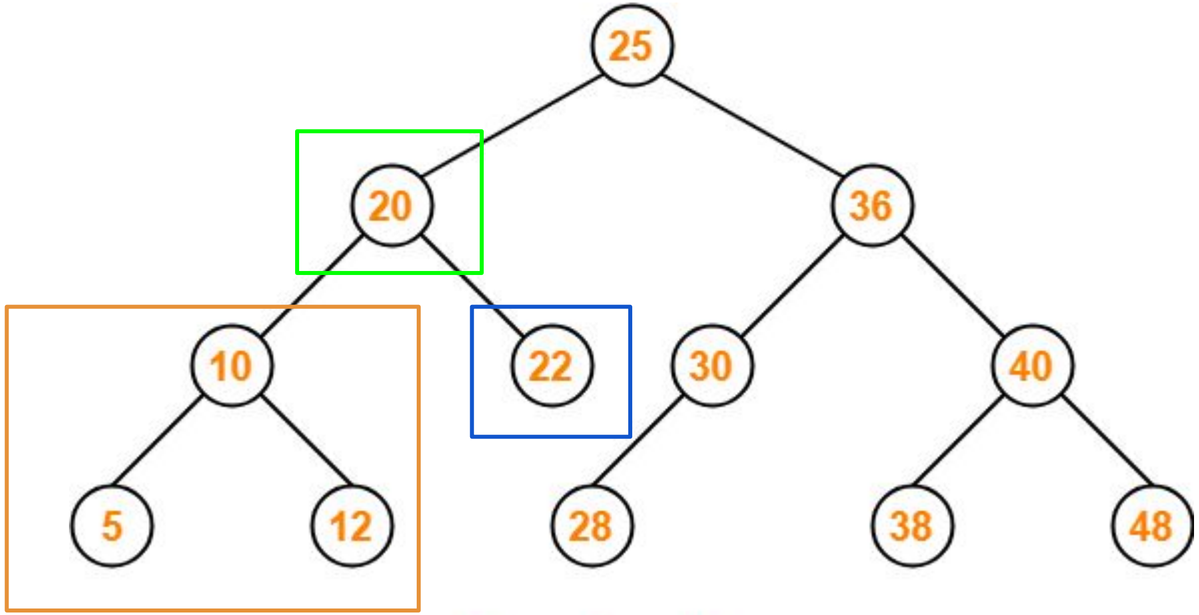
Binary Search Tree

Example BST



Binary Search Tree

Example BST



Binary Search Tree

Methods for a BST (we can use the Map interface!)

put(K key, V value)	Adds the key-value pair to the BST
get(K key)	Returns the value corresponding to the given key
remove(K key)	Removes the entry corresponding to the given key
replace(K key, V newValue)	Replace the value that maps to the given key

BST Visualization

Link to good visualizations where you can create and execute your own examples:

https://www.cs.usfca.edu/~galles/visualization/BST.html

BSTMap: remove

`remove`

What cases do we need to consider?

Remove Cases

Node to remove:

1. Is a leaf (no children)
2. Has 1 child
3. Has 2 children

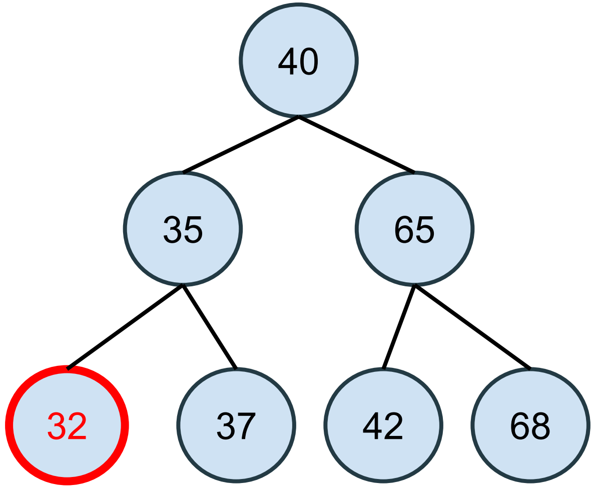
Note on the following examples

There are multiple correct solutions for removing an element - any that preserve the rules of a BST would work: for each Node, every Node in its left subtree has a smaller key, and every Node in its right subtree has a larger key.

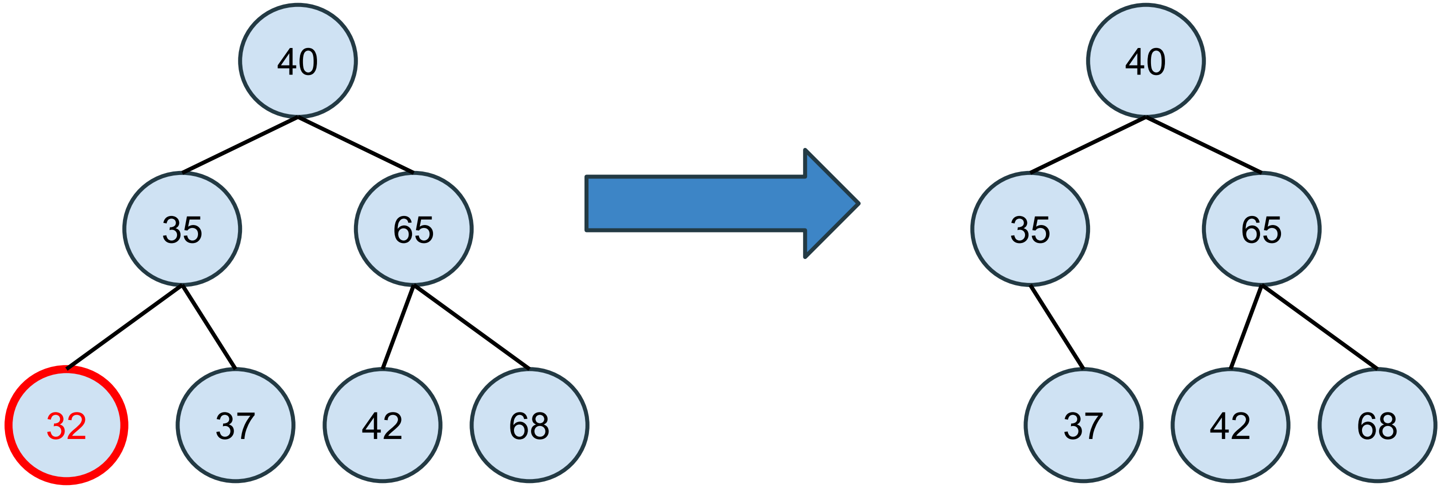
For this example, we use the following general algorithm:

> use binary search to find the node with the key we want to remove, then find the Node with the minimum key in the right subtree, replace our node's fields with the minimum's key and value, then remove this "minimum" Node. This accomplishes "swapping" the Node we want to remove with the Node that has the minimum key of its right subtree.

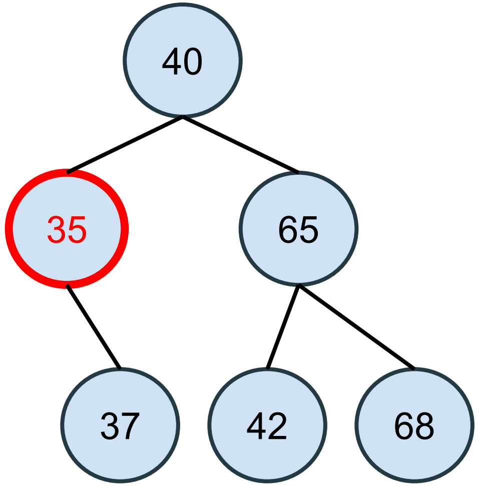
Case 1, Leaf: `remove(32);`



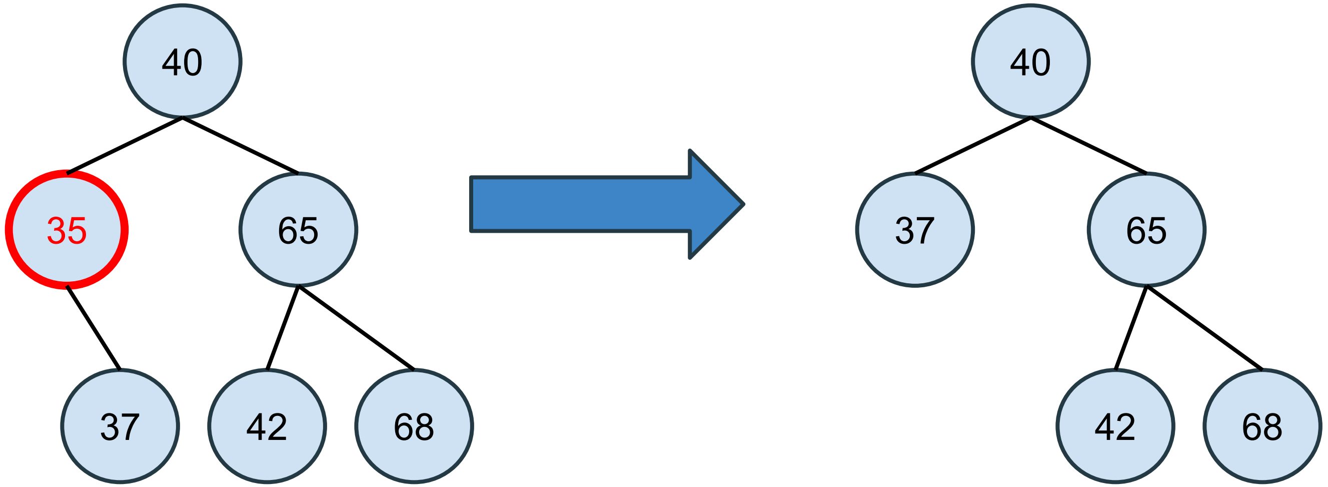
Case 1, Leaf: `remove(32);`



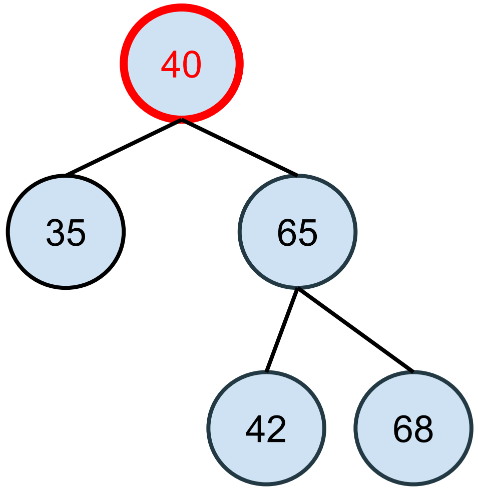
Case 2, Has 1 child: `remove(35);`



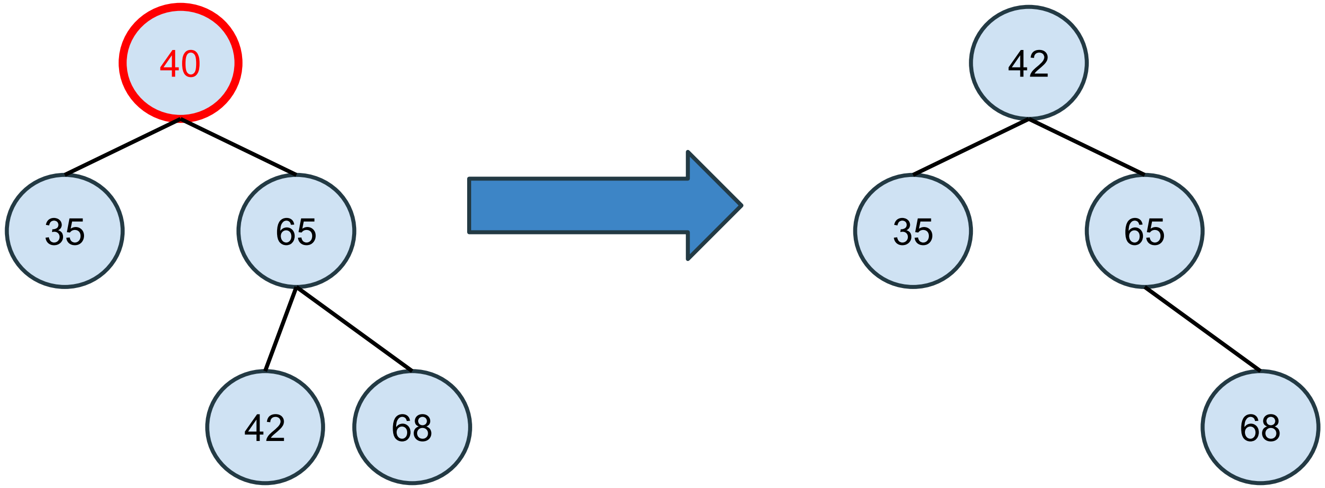
Case 2, Has 1 child: `remove(35);`



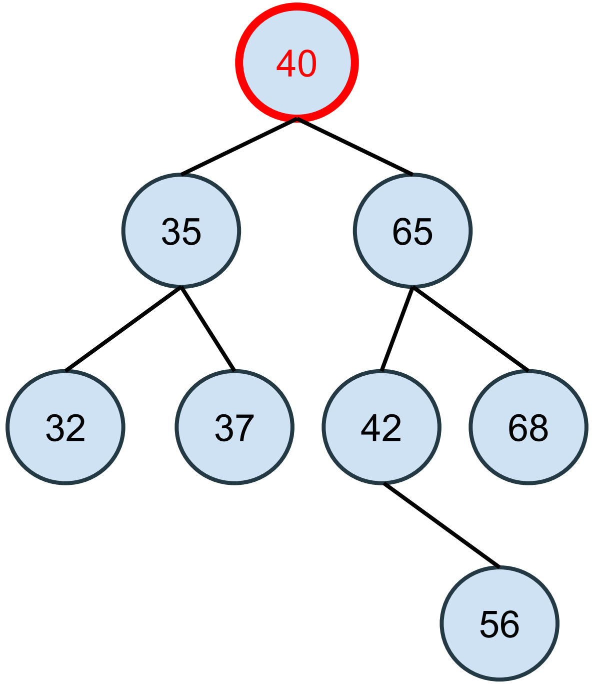
Case 3, Has 2 Children: `remove(40);`



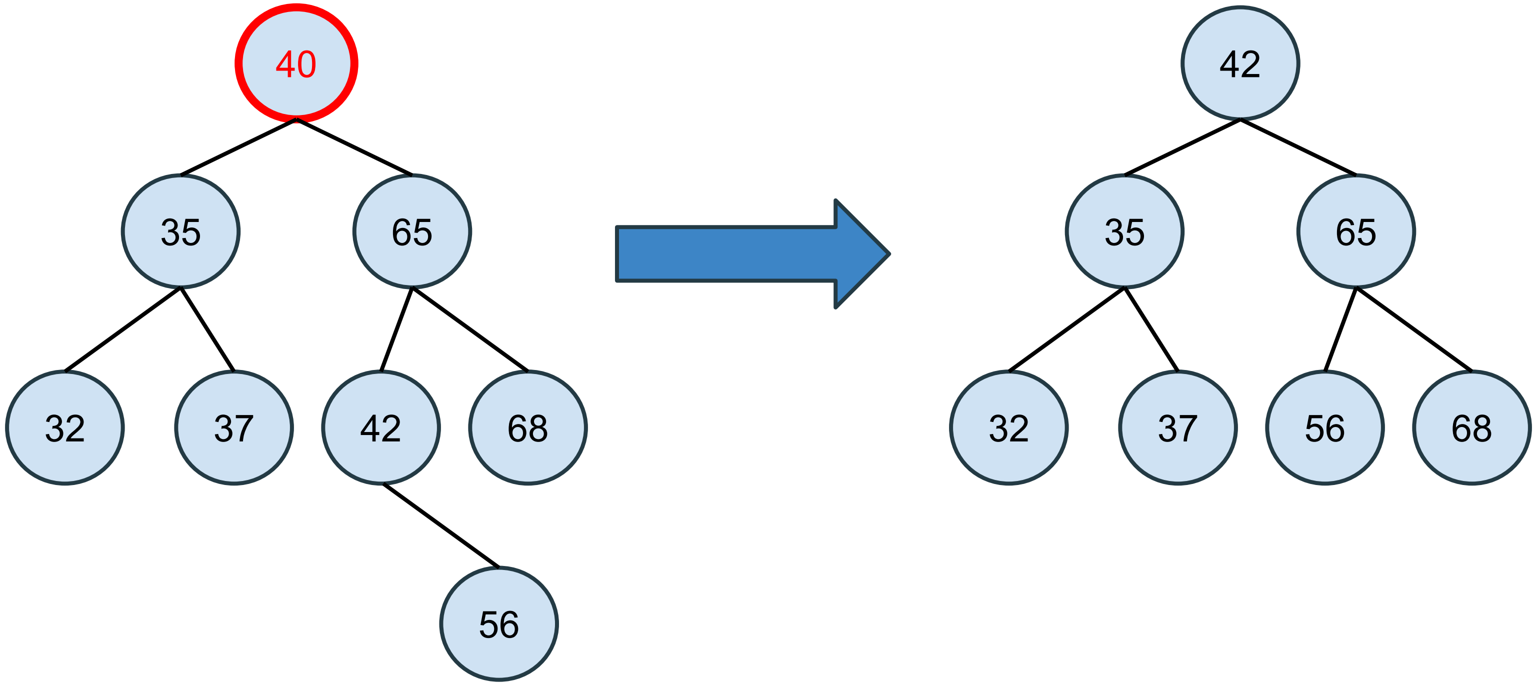
Case 3, Has 2 Children: `remove(40);`



Something to think about



Recursion handles cases like this elegantly

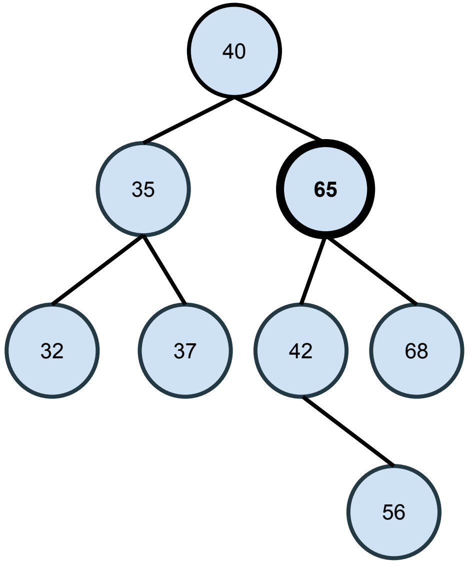


Example code & stack trace for remove

Stack trace of BSTMap remove example

```
@B    Node
        key = 40
        value = 1
        left = @C
        right = @D

@C    Node
        key = 35
        value = 2
        left = @E
        right = @F

@D    Node
        key = 65
        value = 3
        left = @G
        right = @H

@E    Node
        key = 32
        value = 4
        left = null
        right = null

@F    Node
        key = 37
        value = 5
        left = null
        right = null

@G    Node
        key = 42
        value = 6
        left = null
        right = @I

@H    Node
        key = 68
        value = 7
        left = null
        right = null

@I    Node
        key = 56
        value = 8
        left = null
        right = null
```

```
@A    BSTMap
        root = @B
        size = 6
        comparator =
         String::compare
```



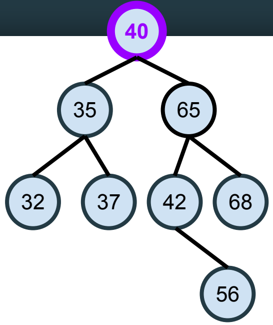
In the interest of readability of the following stack trace, we will only include stack snapshots and Nodes relevant to the example of removing the Node with the `key`, **65**.

Anything highlighted in red represents what is currently "happening" in a particular step

Nodes highlighted in purple represent the Node represented by the variable `node` in the current step

**We suggest zooming in on parts if it is difficult to see, but please let us know if this is not feasible for you such that you can't follow along, so we can address it!

We call `removeRecursively`, passing in the root Node and the `key` that we wish to remove, 65. We determine the `keyToRemove` is greater than 40 and call the method again on the right child of `node`, with `key`, 65.



Stack

@A.removeRecursively(@B, 65)	
@B.right	
return	

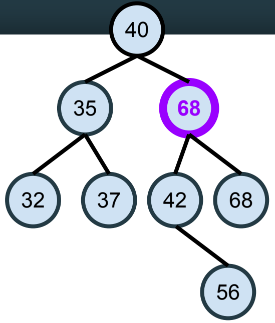
@B	Node
	key = 40
	value = 1
	left = @C
	right = @D

@D	Node
	key = 65
	value = 3
	left = @G
	right = @H

@H	Node
	key = 68
	value = 7
	left = null
	right = null

```
public Node<K, V> removeRecursively(Node<K, V> node, K keyToRemove) throws NoSuchElementException{
    // If tree is empty
    if (node == null) { return null; }

    int compared = this.comparator.compare(node.key, keyToRemove);
    if (compared > 0) {
      System.out.println("Calling remove on left child: <" + node.key + ", " + node.value + ">");
      node.left = removeRecursively(node.left, keyToRemove);
    } else if (compared < 0){
      System.out.println("Calling remove on right child: <" + node.key + ", " + node.value + ">");
      node.right = removeRecursively(node.right, keyToRemove);
    }
    // node has the key we're looking to remove
    else {
      System.out.println("We found the key we're looking for: <" + node.key + ", " + node.value + ">");
      // Case: node with only one child or no children
      if (node.left == null){
        return node.right;
      } else if (node.right == null){
        return node.left;
      }

      // Case: node with two children
      // Get minimum from right subtree, then remove it
      Node<K, V> nextLargest = nodeWithMinimumKey(node.right); //see method in our posted source code
      node.key = nextLargest.key;
      node.value = nextLargest.value;

      // Remove nextLargest node
      node.right = removeRecursively(node.right, node.key);
    }
    return node;
}
```

We found the Node with the key, 65. We then find the Node with the minimum key in the right subtree node, which has key 68. We do a deep copy to replace the key and value of node with this minimum Node's fields. We then call removeRecursively on the right child, the other Node with key 68 (which also happens to be the minimum we just found), and the keyToRemove, 68, to remove the now "duplicate" key.



Stack

@A.removeRecursively(@D, 65)	
@D.key	68
@D.value	7
@D.right	
return	

@A.removeRecursively(@B, 65)	
@B.right	
return	

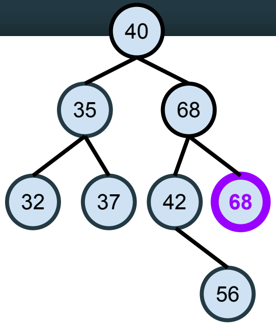
@B	Node
	key = 40
	value = 1
	left = @C
	right = @D

@D	Node
	key = ~~65~~ 68
	value = ~~3~~ 7
	left = @G
	right = @H

@H	Node
	key = 68
	value = 7
	left = null
	right = null

```
public Node<K, V> removeRecursively(Node<K, V> node, K keyToRemove) throws NoSuchElementException{
    // If tree is empty
    if (node == null) { return null; }

    int compared = this.comparator.compare(node.key, keyToRemove);
    if (compared > 0) {
      System.out.println("Calling remove on left child: <" + node.key + ", " + node.value + ">");
      node.left = removeRecursively(node.left, keyToRemove);
    } else if (compared < 0){
      System.out.println("Calling remove on right child: <" + node.key + ", " + node.value + ">");
      node.right = removeRecursively(node.right, keyToRemove);
    }
    // node has the key we're looking to remove
    else {
      System.out.println("We found the key we're looking for: <" + node.key + ", " + node.value + ">");
      // Case: node with only one child or no children
      if (node.left == null){
        return node.right;
      } else if (node.right == null){
        return node.left;
      }

      // Case: node with two children
      // Get minimum from right subtree, then remove it
      Node<K, V> nextLargest = nodeWithMinimumKey(node.right); //see method in our posted source code
      node.key = nextLargest.key;
      node.value = nextLargest.value;

      // Remove nextLargest node
      node.right = removeRecursively(node.right, node.key);
    }
    return node;
}
```

We found the Node with `key` 68. It's left child is `null`, so we return its right child, which also happens to be `null`.



Stack

@A.removeRecursively(@H, 68)
return null

@A.removeRecursively(@D, 65)	
@D.key	68
@D.value	7
@D.right	
return	

@A.removeRecursively(@B, 65)	
@B.right	
return	

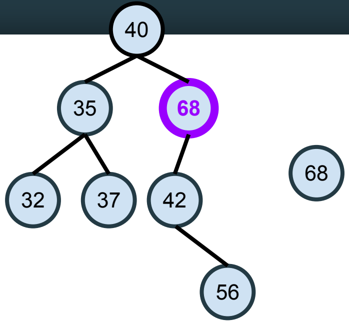
@B	Node
	key = 40
	value = 1
	left = @C
	right = @D

@D	Node
	key = 68
	value = 7
	left = @G
	right = @H

@H	Node
	key = 68
	value = 7
	left = null
	right = null

```
public Node<K, V> removeRecursively(Node<K, V> node, K keyToRemove) throws NoSuchElementException{
    // If tree is empty
    if (node == null) { return null; }

    int compared = this.comparator.compare(node.key, keyToRemove);
    if (compared > 0) {
      System.out.println("Calling remove on left child: <" + node.key + ", " + node.value + ">");
      node.left = removeRecursively(node.left, keyToRemove);
    } else if (compared < 0){
      System.out.println("Calling remove on right child: <" + node.key + ", " + node.value + ">");
      node.right = removeRecursively(node.right, keyToRemove);
    }
    // node has the key we're looking to remove
    else {
      System.out.println("We found the key we're looking for: <" + node.key + ", " + node.value + ">");
      // Case: node with only one child or no children
      if (node.left == null){
        return node.right;
      } else if (node.right == null){
        return node.left;
      }

      // Case: node with two children
      // Get minimum from right subtree, then remove it
      Node<K, V> nextLargest = nodeWithMinimumKey(node.right); //see method in our posted source code
      node.key = nextLargest.key;
      node.value = nextLargest.value;

      // Remove nextLargest node
      node.right = removeRecursively(node.right, node.key);
    }
    return node;
}
```

The `null` returned is assigned as the right child Node of @D, which “cuts out” @H from our tree. It will be garbage collected upon returning from this call. We return node.



Stack

@A.removeRecursively(@D, 65)	
@D.key	68
@D.value	7
@D.right	null
return @D	

@A.removeRecursively(@B, 65)	
@B.right	
return	

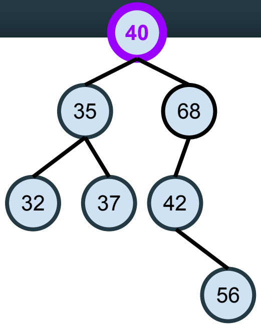
@B	Node
	key = 40
	value = 1
	left = @C
	right = @D

@D	Node
	key = 68
	value = 7
	left = @G
	right=~~@H~~ null

@H	Node
	key = 68
	value = 7
	left = null
	right = null

```java
public Node<K, V> removeRecursively(Node<K, V> node, K keyToRemove) throws NoSuchElementException{
    // If tree is empty
    if (node == null) { return null; }

    int compared = this.comparator.compare(node.key, keyToRemove);
    if (compared > 0) {
      System.out.println("Calling remove on left child: <" + node.key + ", " + node.value + ">");
      node.left = removeRecursively(node.left, keyToRemove);
    } else if (compared < 0){
      System.out.println("Calling remove on right child: <" + node.key + ", " + node.value + ">");
      node.right = removeRecursively(node.right, keyToRemove);
    }
    // node has the key we're looking to remove
    else {
      System.out.println("We found the key we're looking for: <" + node.key + ", " + node.value + ">");
      // Case: node with only one child or no children
      if (node.left == null){
        return node.right;
      } else if (node.right == null){
        return node.left;
      }

      // Case: node with two children
      // Get minimum from right subtree, then remove it
      Node<K, V> nextLargest = nodeWithMinimumKey(node.right); //see method in our posted source code
      node.key = nextLargest.key;
      node.value = nextLargest.value;

      // Remove nextLargest node
      node.right = removeRecursively(node.right, node.key);
    }
    return node;
}
```

The Node returned is assigned as the right child of the Node with `key` 40. We then return this node. It will be assigned as the `root` of the map ([see our source code to see the other method that called `removeRecursively` originally]()).



Stack

@A.removeRecursively(@B, 65)	
@B.right	@D
return @B	

@B	Node
	key = 40
	value = 1
	left = @C
	right = @D

@D	Node
	key = 68
	value = 7
	left = @G
	right = null

@H	Node
	key = 68
	value = 7
	left = null
	right = null

```
public Node<K, V> removeRecursively(Node<K, V> node, K keyToRemove) throws NoSuchElementException{
    // If tree is empty
    if (node == null) { return null; }

    int compared = this.comparator.compare(node.key, keyToRemove);
    if (compared > 0) {
      System.out.println("Calling remove on left child: <" + node.key + ", " + node.value + ">");
      node.left = removeRecursively(node.left, keyToRemove);
    } else if (compared < 0){
      System.out.println("Calling remove on right child: <" + node.key + ", " + node.value + ">");
      node.right = removeRecursively(node.right, keyToRemove);
    }
    // node has the key we're looking to remove
    else {
      System.out.println("We found the key we're looking for: <" + node.key + ", " + node.value + ">");
      // Case: node with only one child or no children
      if (node.left == null){
        return node.right;
      } else if (node.right == null){
        return node.left;
      }

      // Case: node with two children
      // Get minimum from right subtree, then remove it
      Node<K, V> nextLargest = nodeWithMinimumKey(node.right); //see method in our posted source code
      node.key = nextLargest.key;
      node.value = nextLargest.value;

      // Remove nextLargest node
      node.right = removeRecursively(node.right, node.key);
    }
    return node;
}
```

File System

FileData.java

This class represents the file that contains the information for name, directory, and last modified date.

Two Methods

FileData() - constructor, initializes the instance variables

toString() - returns the string representation of the data in the FileData object

FileSystem.java

- FileSystem represents the entire structure of the system. This FileSystem though as a focus on filtering through the files to only have files with either a specific date or name.
- Three instance variables:
  - **BST<String, FileData> nameTree;**
  - **BST<String, ArrayList<FileData>> dateTree;**
  - **DateComparator dc;**

Comparators

```
class Person { String name; int age; }
// And sometimes we want to order People by name, and other times by age. We
// could define a Comparator for each of those cases:

class AgeComparator implements Comparator<Person> {
  public int compare(Person p1, Person p2) { return p1.age - p2.age; }
}
class NameComparator implements Comparator<Person> {
  public int compare(Person p1, Person p2) { return p1.name.compareTo(p2.name);
}
}
```

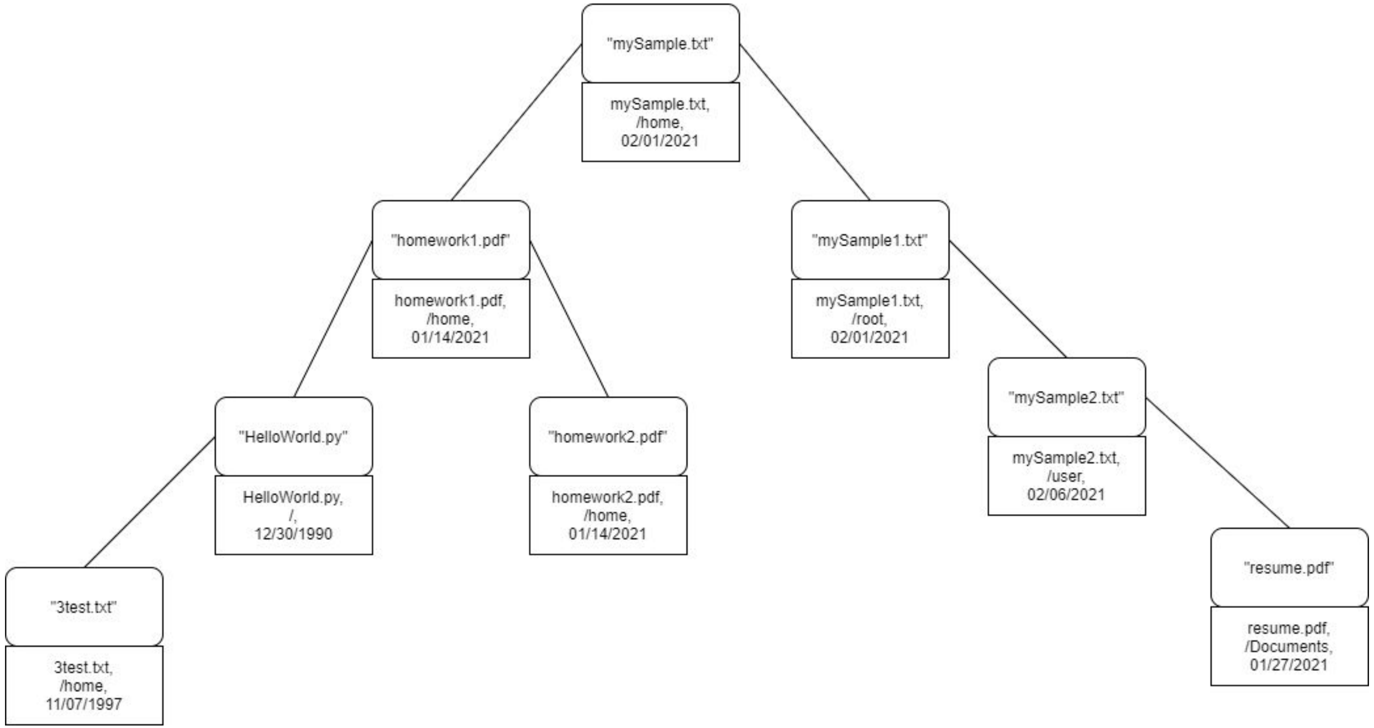
DateComparator

```
public int compare(String date1, String date2)
```

Given two strings that contain the last modified date, if date1 is more recent than date2 return 1, if date1 is less recent than date2 return -1 and if date1 is equal to date2 return 0. For example, 01/01/2021 is less recent than 01/02/2021 - thus `compare("01/01/2021", "01/02/2021")` should return -1. Similarly, 12/01/2020 is less recent than 01/01/2021 and should return -1. You can assume that none of the inputs are null.

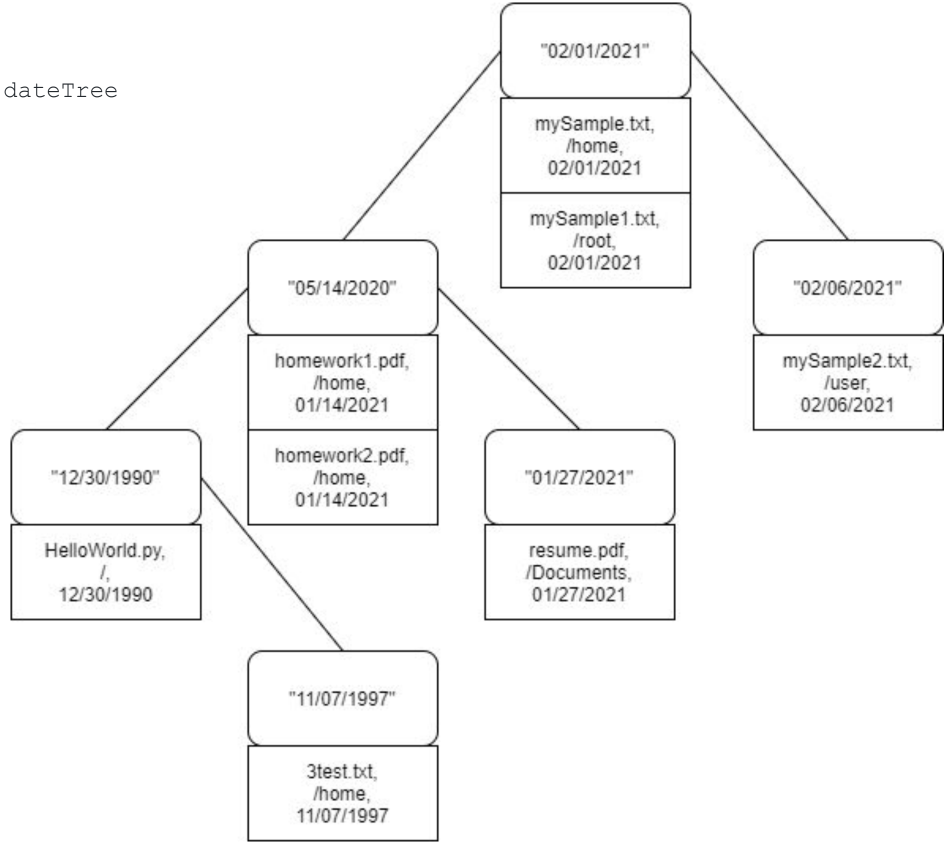
nameTree

```
BST<String, FileData> nameTree
```


"mySample.txt"
mySample.txt, /home, 02/01/2021
"homework1.pdf"
homework1.pdf, /home, 01/14/2021
"mySample1.txt"
mySample1.txt, /root, 02/01/2021
"HelloWorld.py"
HelloWorld.py, /, 12/30/1990
"homework2.pdf"
homework2.pdf, /home, 01/14/2021
"mySample2.txt"
mySample2.txt, /user, 02/06/2021
"3test.txt"
3test.txt, /home, 11/07/1997
"resume.pdf"
resume.pdf, /Documents, 01/27/2021

dateTree

```
BST<String, ArrayList<FileData>> dateTree
```


"02/01/2021"
mySample.txt, /home, 02/01/2021
mySample1.txt, /root, 02/01/2021
"05/14/2020"
homework1.pdf, /home, 01/14/2021
homework2.pdf, /home, 01/14/2021
"02/06/2021"
mySample2.txt, /user, 02/06/2021
"12/30/1990"
HelloWorld.py, /, 12/30/1990
"01/27/2021"
resume.pdf, /Documents, 01/27/2021
"11/07/1997"
3test.txt, /home, 11/07/1997

Filtering

```
- public FileSystem filter(String startDate, String endDate)

- public FileSystem filter(String wildCard)
```

```
In order traversal!
```

Output the FileSystem

mySample.txt, /home, 02/01/2021

mySample1.txt, /root, 02/01/2021

mySample2.txt, /user, 02/06/2021

outputNameTree

```
["mySample.txt: {Name: mySample.txt,
Directory: /home, Modified Date: 02/01/2021}",
"mySample1.txt: {Name: mySample1.txt,
Directory: /root, Modified Date: 02/01/2021}",
"mySample2.txt: {Name: mySample2.txt,
Directory: /user, Modified Date: 02/06/2021}"]
```

outputDateTree

```
["02/06/2021: {Name: mySample2.txt, Directory:
/user, Modified Date: 02/06/2021}",
"02/01/2021: {Name: mySample1.txt, Directory:
/root, Modified Date: 02/01/2021}",
"02/01/2021: {Name: mySample.txt, Directory:
/home, Modified Date: 02/01/2021}"]
```