CSE 12 Winter 2022 PA8 - BST and Sorting

Due date: Tuesday, March 8 @ 11:59 PM PDT

There is an FAQ post on Piazza. Please read that post first if you have any questions.

Learning goals:

- Implement and also use Binary Search Tree to solve real-world problems
- Write JUnit tests to verify proper implementation
- Compare the efficiency of different sorting algorithms

Overview:

There are 4 parts to this assignment. All are required before the deadline and should be submitted on Gradescope:

- Part 1: Binary Search Tree [6.5 points]
  - Testing [1 point]
  - MyBST Implementation and Application [5 points]
  - Coding Style [0.5 points]
- Part 2: Sort Detective [1 point]
- Part 3: Concept Quiz [2 points]
- Part 4: PA Reflection [0.5 points]

Part 1: Binary Search Tree [6.5 points] (individual)

In this part of the assignment, you will implement a Binary Search Tree and write JUnit tests to ensure that your implementation functions correctly.

Make sure to **read the entire write-up and also the starter code (including the comments)** before you start coding. Many methods have been implemented in the starter code with commented out explanations, and you may need to implement the methods that are colored in blue only. Please complete all the parts **individually; pair programming is not allowed for this PA**. You may not change any public class/method or variable names given in the write-up or starter code.

Download the starter code from here and put it in a directory in your working environment. You will find the following files:

```
+-- PA8
|       +-- MyBST.java                          Edit this file
|       +-- MyBSTIterator.java                  Edit this file
|       +-- MyCalendar.java                     Edit this file
|       +-- MyTreeMap.java
|       +-- CustomTester.java                   Edit this file
|       +-- PublicTester.java
```

Part 1a: Testing

We provide two test files:

- PublicTester.java (commands to run the tester)
  - Contains all the public test cases we will use to grade your BST implementation and application (visible on Gradescope). See the comments in the file for test case details.
- CustomTester.java
  - This file is currently empty and you will edit this file. Come up with at least 10 different test cases by yourself to test the methods you implement.
  - Points will be **deducted** if method header descriptions are missing.
  - There is no restriction on what error message you use as the string input.
  - You MUST **NOT** copy code from the public tester file.
  - Make sure all the test cases in the custom tester file pass. Otherwise, you would receive 0 points for the entire file.

There will be no "resubmissions" for this PA after grades are released, so please make sure to thoroughly test your code! (You still can submit as many times as you want before the deadline.)

Part 1b: MyBSTNode

This class is a static nested class of the MyBST class. Objects of this class represent the nodes of the binary search tree and contain the following private instance variables:

Instance Variable	**Description**
`K key`	the key that we are using to sort our nodes This key cannot be `null` and we are not supporting duplicate keys (duplicates are defined by compareTo()

	returning that the keys are equal).
`V value`	the data stored in this MyBSTNode This value is allowed to be `null`.
`MyBSTNode<K, V> parent`	a reference to the parent of this MyBSTNode If this node has no parent, then this will be `null`.
`MyBSTNode<K, V> left`	a reference to the left child of this MyBSTNode If this node has no left child, then this will be `null`.
`MyBSTNode<K, V> right`	a reference to the right child of this MyBSTNode If this node has no right child, then this will be `null`.

Since these instance variables are all private, remember to use getter/setter methods to access/modify them.

Method to implement:

public MyBSTNode<K, V> predecessor()

- This method returns the node with the greatest key that is smaller than the key of this node. If there is no smaller key, return `null`.
- Every node in our BST should / will already have the proper connections (parent and child references) with our sorted key property.
- Hint: This method could be implemented in a similar way as the `successor` method and don't forget to explain the provided `successor` method in your comment thoroughly.
- Important: **You are not allowed to change the class signature! If you change the class signature, you'll receive 0 for this part.**

Part 1c: MyBST

The MyBST class is the public class of MyBST.java file. It represents a binary search tree where sorting is done based on the keys. This is not a self-balancing tree. Make sure you don't change the class header, otherwise you would receive 0 points for the entire file.

MyBST has the property that all nodes in the left subtree of a node will have a key smaller than `node.key`, and all nodes in the right subtree of node will have a key greater than `node.key`. **We will not be allowing duplicate keys in our tree**, but we can have many different keys associated with the same value. We will be using the `compareTo` method to compare smaller/greater/equal keys.

Instance Variable	**Description**
`MyBSTNode<K, V> root`	a reference to the root node of our tree If the tree is empty, then this will be `null`.
`int size`	represents the number of nodes in the tree

Methods to implement (iteration and recursion are both allowed for this PA):

public V insert(K key, V value)

- If `key` is `null`, throw a `NullPointerException`.
- Insert a new node containing the arguments `key` and `value` into the binary search tree according to the binary search tree properties. Update the tree size accordingly.
- If a node with an equal key to `key` already exists in the tree, replace the value in the current node with `value`. (diagrams)
  - Note: We won't change the key of the pre-existing node.
- Return the value replaced by the new `value`. If the insertion did not lead to a replacement (i.e., it created a new node as a leaf, note that if the tree only has a root then the root is a leaf), return `null` instead. Note that we've overloaded the meaning of returning `null` as it can mean that we've inserted a node with a new key into the tree or that we've replaced the value of a node whose original value was `null`.

public V search(K key)

- Search for a node with key equal to `key` and return the value associated with that node. The tree structure should not be affected by this.
- If no such node exists, return `null` instead. Again, note that we've overloaded the meaning of returning `null`.
- Note: `key` might be `null`. We'll assume that no other key is equal to `null` (even though the `compareTo()` of type `K` might define this differently). This means that `search(null)` should always return `null`.

public V remove(K key)

- Search for a node with key equal to `key` and return the value associated with that node. The node should be removed (disconnected) from the tree and all references should be fixed, if needed. Update the tree size accordingly.
- When removing the node, if the node is not a leaf node, then we may have to make some changes to the tree to maintain its integrity. Make sure to fix all relevant references to parents/children.
    - For leaf nodes, no replacement is needed.
    - If the node has a single child, move that child up to take its place. (diagrams)
    - If the node has two children, then we will just overwrite the data at the node we plan to remove with the data from the successor, saving us the need to reconnect all the nodes if we had completely removed the node from the tree. Then remove the successor from the tree (this does not need to be done iteratively, you can do this with a recursive call). (diagrams)
- If no such node exists, return `null` instead. Again, note that we've overloaded the meaning of returning `null`.
- Note: `key` might be `null`. We'll assume that no other key is equal to `null` (even though the `compareTo()` of type `K` might define this differently). This means that `remove(null)` should always return `null` and not remove any nodes.

public ArrayList<MyBSTNode<K, V>> inorder()

- Do an in-order traversal of the tree, adding each node to the end of an `ArrayList`, which will be returned.
- After the entire traversal, this `ArrayList` should contain all nodes in the tree. These nodes will be sorted by the key order (with the smallest key at index 0) due to the order of the traversal.
- If the tree is empty, an **empty** (not `null`) `ArrayList` should be returned.
- Hint: it might be helpful to think about what the `successor` method does and how it relates to an in-order traversal. Using sorting algorithms is not allowed for this method.

Part 1d: MyBSTIterator

MyBSTNodeIterator is an inner (abstract) class of MyBSTIterator.java.

Methods to implement:

MyBSTNode<K, V> nextNode()

- Advances the iterator to the next node and returns the node we advanced (/visited) to
- If there is no next node, throw a `NoSuchElementException` and do not modify the iterator's state.
- Upon a successful advancement, `next` should be updated to point to the inorder successor (possibly `null` if we've now reached the end of the iterator) and `lastVisited` should be updated to point to the node that `next` previously pointed to.
- Hint: Remember to use the `successor` method that we previously wrote.

And don't forget to explain the provided `remove` method **line by line** in your comment.

Part 1e - MyCalendar

In this part, you will be implementing your own calendar! Feel free to add as many events as you'd like to your calendar **while avoiding double booking**. (Double booking is defined as having more than one booked event at the same time.)

We use two Integer values (`start` and `end`) to represent the start time and the end time of the event respectively. In other words, the event will take place in the range of real numbers `i` such that `0 <= start <= i < end`.

Your implementation for BST corresponds to the TreeMap data structure in Java. In this PA, we have provided a class called MyTreeMap. You will use methods in this class to implement `MyCalendar.java`.

Instance Variable	**Description**
`MyTreeMap<Integer, Integer> calendar`	represents the TreeMap that stores the booked events

Methods to implement:

public MyCalendar()

- Initializes the calendar object.

public boolean book(int start, int end)

- Throw `illegalArgumentException` if `start < 0` or `start >= end`.
- If adding the event does not cause a double booking, add the event and return true.
- Otherwise, return false and do not add the event.

Example:

```
MyCalendar calendar = new MyCalendar();
calendar.book(0, 5); // return true
calendar.book(5, 10); // return true because 5 is not booked yet
calendar.book(9, 12); // return false because 9 is already booked
```

Part 1f - Style

The following files will be checked for all rubric items in our style guide. You need to make sure everything complies with our style guide, including what we provide to you in the starter code. To be consistent with Javadoc format and avoid warning messages generated by style grader, make sure you do not have dash(-) or colon after @param or @return. If there's no parameter / return value, make sure to not include the corresponding tag (instead of writing `@param none`).

- MyBST.java
- MyBSTIterator.java
- MyCalendar.java

The following file will be checked for only file headers and method headers.

- Custom Tester.java

Comments of the following methods will also be graded:

- successor in MyBST.java
- remove in MyBSTIterator.java

Note: for the `successor` method, add enough comments such that other programmers can understand your code just by reading the comment. For the iterator's `remove` method, add comments for each line of the entire code block.

Submission

Submit all of the following files to Gradescope by **Tuesday, March 8 @11:59PM PST.**

- MyBST.java
- MyBSTIterator.java
- MyCalendar.java
- CustomTester.java

Part 2: Sort Detective [1 point] (individual)

This assignment is developed from http://nifty.stanford.edu/2002/LevineSort/labWriteUp.htm

In this assignment, you are given a program which is designed to measure comparisons, data movements, and execution time for some of the sorting algorithms we discussed in class and in your reading: **insertion sort, selection sort, merge sort and quicksort. This assignment is an individual assignment.** You may ask Professors/TAs/Tutors for some guidance and help. You can, of course, discuss the assignment with your classmates, but don't just give away the answers!

The following files are provided for you and can be found in the release page of the Github repo:

- sort-detective.jar
- SortingExperiment.java

You will submit the following file for this assignment:

- **detective.json**

As you know from class, if you double the size of the input data that you give to a quadratic algorithm, it will do four times the work; by contrast, an $O(n \log n)$ algorithm will do a bit more than twice as much; and, a linear algorithm will do only twice as much work. As you also know, the properties of the input data set can affect the expected performance of many of our sorting algorithms. Before you begin the homework, you should review the expected performance of the algorithms on various data sets. You may go through the first few questions in the concept quiz to help you review that.

1. Execute sort-detective.jar (to execute the program type 'java -jar sort-detective.jar' at the command line), enter your **PID** on the top right corner and play with it a bit. Each of you would get a different matching based on your PID, and we will grade based on your PID as well. Notice that the Greek letters do not give any indication which sort they will execute. Create a list with some size and property of your choice by clicking "Create The List".
2. Click on each Greek letter to see the number of comparisons, movement and running time of this execution.
3. As you click each Greek letter, take careful notes as you go. From your note and the expected behavior of different sorting algorithms, match each Greek letter to some sorting algorithm and think about the rationale behind your guess.
4. Select the sorting algorithms for each Greek letter, and click on "Save to File" to export a json file named **detective.json** that you'll submit to gradescope. **Make sure your PID is correct!**
5. We have also provided SortingExperiment.java for you to know how we implemented each algorithm in sort-detective.jar and how we generate number of comparisons and execution time. Looking at the code is optional but encouraged which is a review of how we can implement these algorithms, and moreover, make your guess more accurate.

Submit **detective.json** to part 2 on Gradescope. Double-check the output to make sure your selections are accurate. Your grade on this part will be based on whether you've correctly identified the sort. In addition to this, you'll need to answer some questions relevant to this simulation in the concept quiz.

Part 3: Concept Quiz [2 points] (individual)

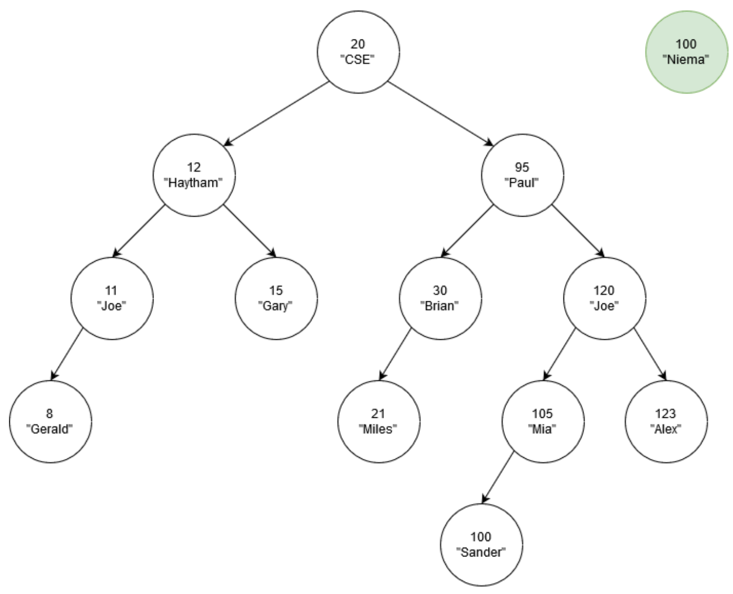
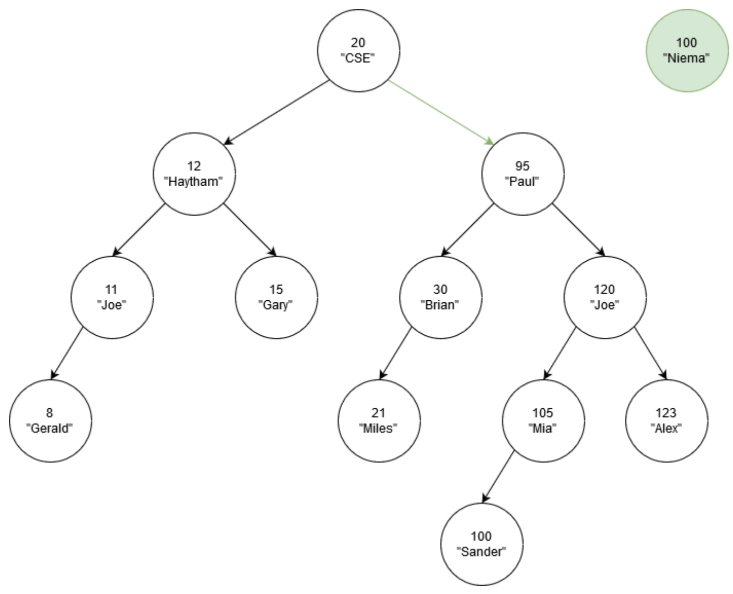
You are required to complete the Concept Quiz on Gradescope. There is no time limit on the Quiz but you must submit the Quiz by the PA deadline. You can submit multiple times before the deadline. Your latest submission will be graded.

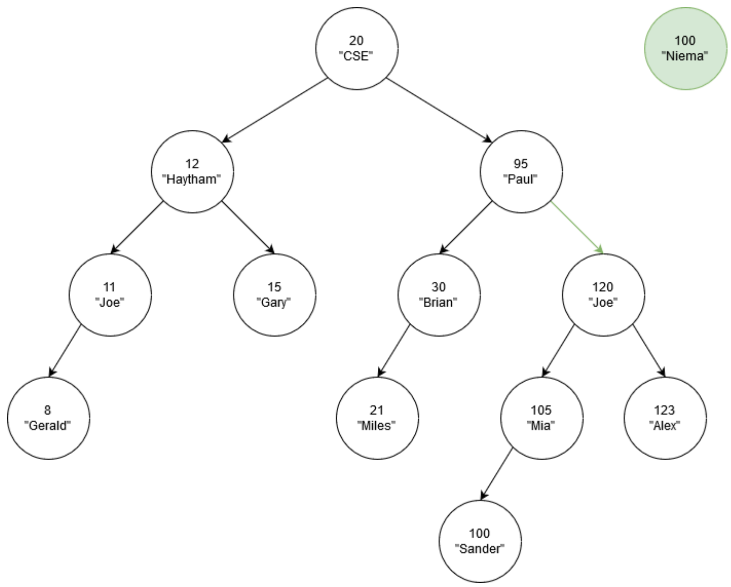
Part 4: PA Reflection [0.5 points] (individual)

Complete the PA8 reflection. There is an automatic 36-hour grace period for this part.

Appendix

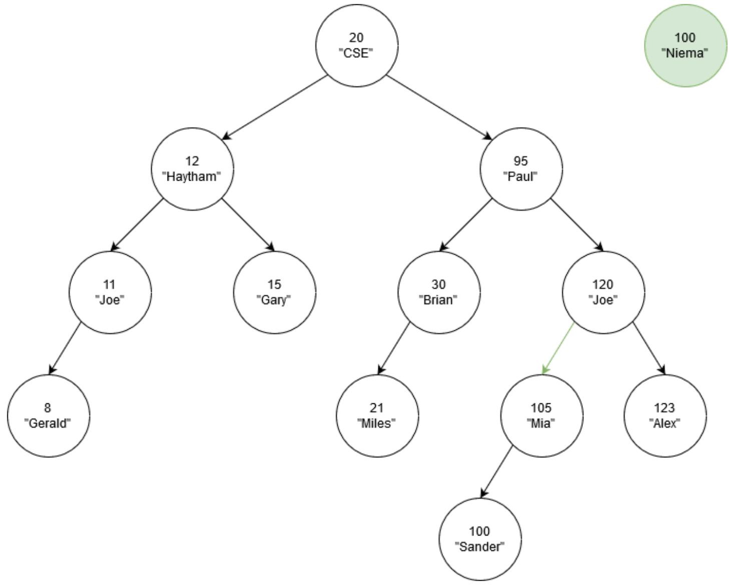
insert an element

 20 "CSE" 100 "Niema" 12 "Haytham" 95 "Paul" 11 "Joe" 15 "Gary" 30 "Brian" 120 "Joe" 8 "Gerald" 21 "Miles" 105 "Mia" 123 "Alex" 100 "Sander"	Call to `insert(100, "Niema")` Begin insertion of a node with key = 100 and value = "Niema".
 20 "CSE" 100 "Niema" 12 "Haytham" 95 "Paul" 11 "Joe" 15 "Gary" 30 "Brian" 120 "Joe" 8 "Gerald" 21 "Miles" 105 "Mia" 123 "Alex" 100 "Sander"	Go down the BST to find the place of insertion. First compare with the root. Since the key of the root (20) is less than 100, we go right, as indicated by the green arrow.

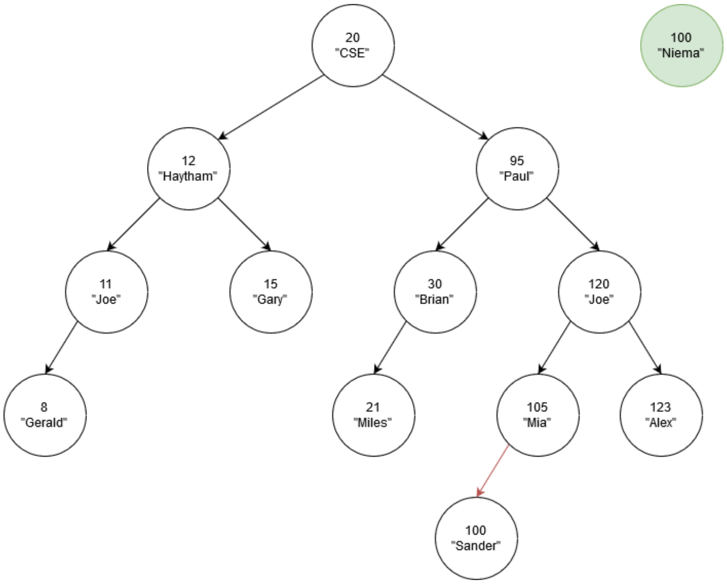
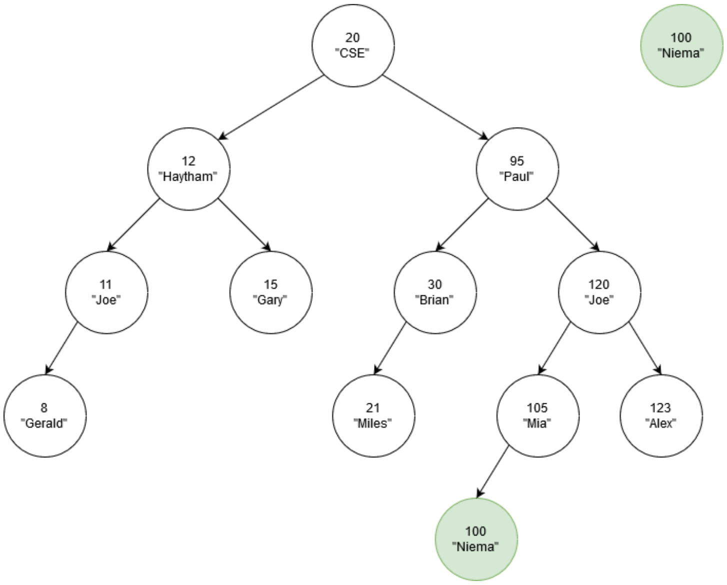


We continue to go down the BST to find the place of insertion.

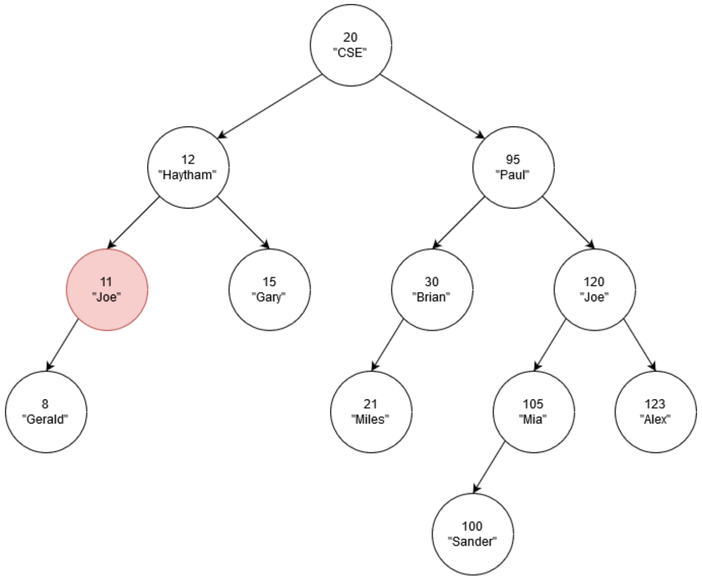
Since the key of the right child of the root (95) is less than 100, we continue to go right, as indicated by the green arrow.



Now we want to go left because the key at the current node (120) is greater than 100.

 20 "CSE" 100 "Niema" 12 "Haytham" 95 "Paul" 11 "Joe" 15 "Gary" 30 "Brian" 120 "Joe" 8 "Gerald" 21 "Miles" 105 "Mia" 123 "Alex" 100 "Sander"	We continue to go left because the key at the current node (105) is greater than 100.
 20 "CSE" 100 "Niema" 12 "Haytham" 95 "Paul" 11 "Joe" 15 "Gary" 30 "Brian" 120 "Joe" 8 "Gerald" 21 "Miles" 105 "Mia" 123 "Alex" 100 "Niema"	However, because a node with a key equal to 100 already exists in the BST, we will replace the original value ("Sander") with the new value ("Niema").

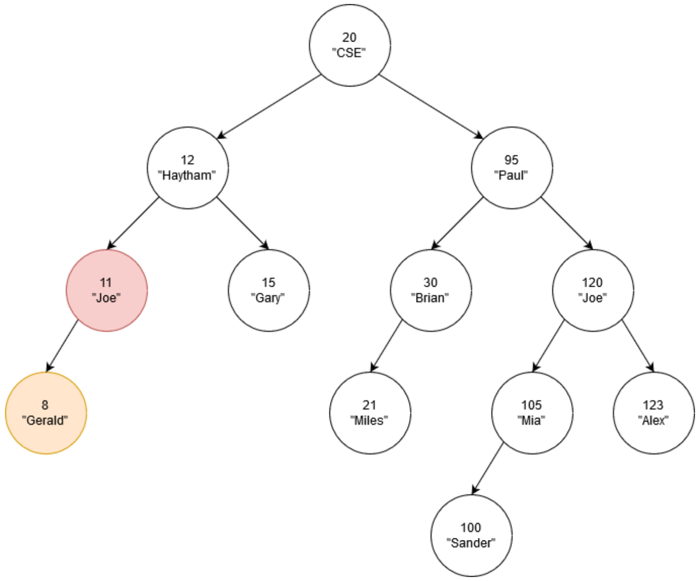
remove a node that has a single child



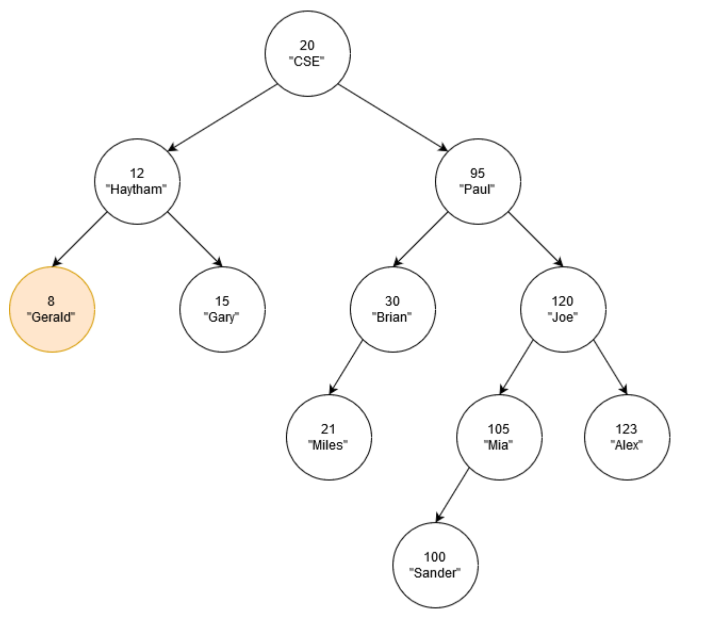
Call to `remove(11)`

Before this step, we've already iterated down the tree to find where the node with key 11 is.

Begin removal of the node with key = 11, and value = "Joe".

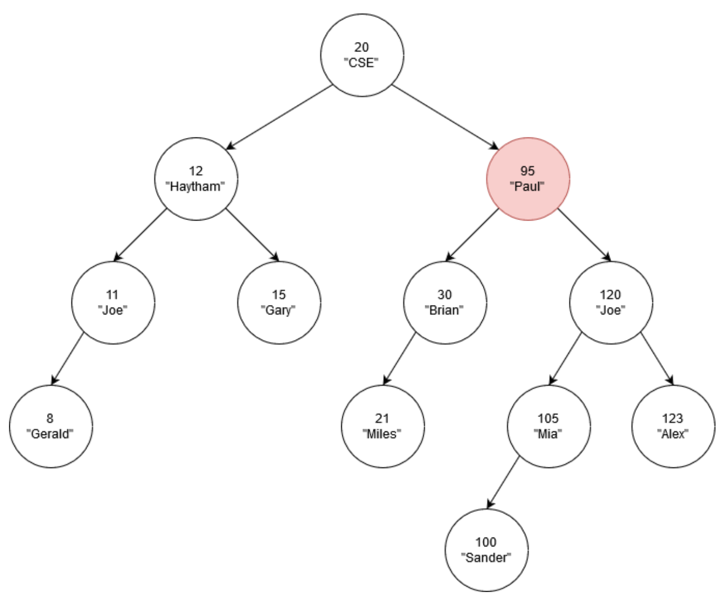


Because the node we want to remove only has one child (the left child), we simply replace the node with its only child.



We disconnect the original node from the tree and fix the parent pointer of the replacement (8:"Gerald") and child pointer of the parent (12:"Haytham"), completing the removal and replacement.

remove a node that has two children

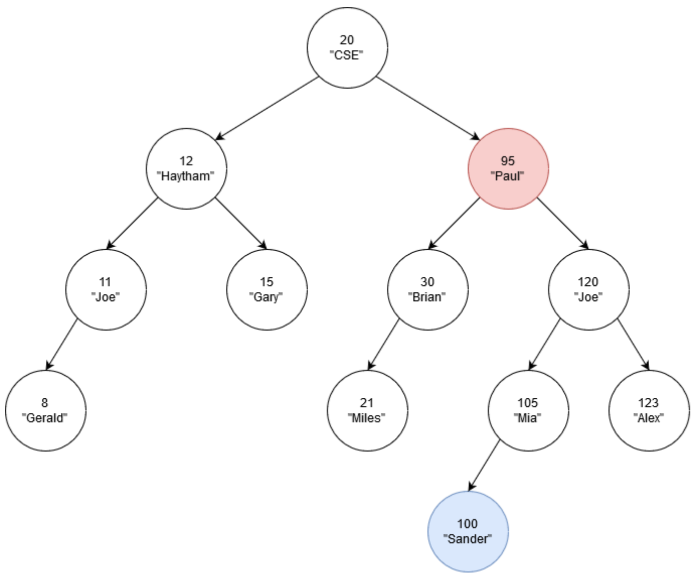


Call to `remove(95)`.

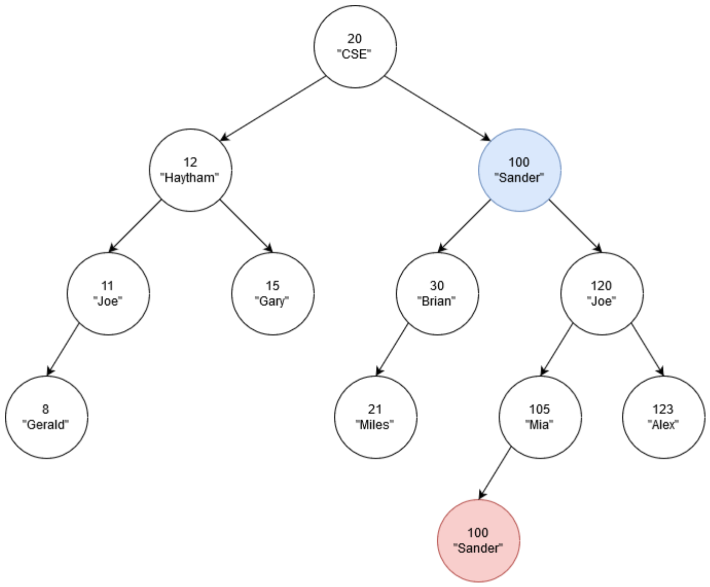
Before this step, we've already iterated down the tree to find where the node with key 95 is.

Begin removal of the node with key = 95 and value = "Paul".

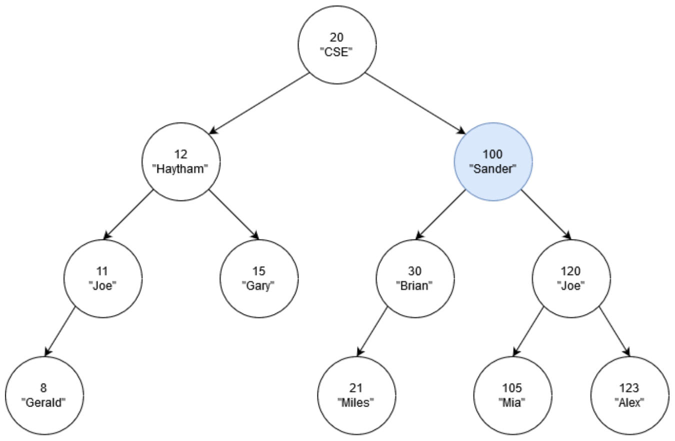
Note: the steps to locate this node are not shown.



Since this node has two children, we want to first find its successor (the blue node) and replace the data (key and value) in this node with its successor's data.



Now we replace the data (key and value) in this node with its successor's data, keeping all the parent/child references.



The final step is to remove the successor and now we are done with removing a node with two children.

How to compile and run the public tester:

on UNIX based systems (including macOS):

```
$ javac -cp libs/junit-4.12.jar:libs/hamcrest-core-1.3.jar:. PublicTester.java
$ java -cp libs/junit-4.12.jar:libs/hamcrest-core-1.3.jar:. org.junit.runner.JUnitCore PublicTester
```

on Windows systems:

```
$ javac -cp ".;libs\junit-4.12.jar;libs\hamcrest-core-1.3.jar" PublicTester.java
$ java -cp ".;libs\junit-4.12.jar;libs\hamcrest-core-1.3.jar" org.junit.runner.JUnitCore PublicTester
```