21. Trees :: Binary Search Trees :: Introduction

A **binary search tree** (BST) is a type of binary tree where the elements stored in the tree can be ordered by a **key**, i.e., for any two elements a and b it is possible to specify that $a.key < b.key$, $a.key = b.key$, or $a.key > b.key$.

The type of the key can be any comparable type and it does not have to be the same as the type of the elements stored in the tree. For example, the elements of a BST may be instances of a class named *Student*. Suppose each *Student* has a unique identification number which is an *Integer*. Then the type of the key would be *Integer*.

The elements are stored in a BST such that for any node N which stores an element e, the keys of all of the elements in the left subtree of N are less than $e.key$ and the keys of all of the elements in the right subtree of N are greater than $e.key$. We assume that the keys are unique.

21. Trees :: Binary Search Trees :: Operations

The primary operations on a BST are (where E the type of the data stored in each node and K is the type of the key):

1. Search the tree for the element with key *key* and return the element. Return null if not found.

   ```
   find(key: K, data: E): E
   ```

2. Add a new element containing *data* with key *key* to the tree. If a node with key *key* already exists in the tree, the data in that element will be replaced with *data*.

   ```
   add(key: K, data: E): void
   ```

3. Remove the element with key *key* from the tree, assuming it exists.

   ```
   remove(key: K): E
   ```

We will discuss the pseudocode implementation of *find*() and *add*(). Removing an element from a BST is more difficult than finding and adding a node, so will just discuss the algorithm without looking at pseudocode.