Binary Search Tree

A search tree is a data structure that is designed specifically to allow for a rapid search. Using such a search tree allows the user to find an object of specified value in a list without having to search through the entire list. This brings the temporal complexity of searching down from $O(n)$ to $O(\log n)$. In other words, rather than searching through n items to find the one I am looking for, I only need to look through log (n). In addition, a search tree allows for quicker insertion and deletion.

The Binary Search Tree is one specific search tree that accomplishes this task. It is dependent on the **Binary Search Algorithm**. This algorithm searches for a specific value in a sorted list by comparing it to the "middle" argument. If the value we are searching for is less than the middle argument then we look only in the first half of the list. On the other hand, if the value is greater than the middle argument, we look only in the second half. Thus, each step of our search effectively cuts the length of the list we search in half.

In a similar fashion, the **Binary Search Tree** (BST) is a sorted data structure that allows us to search for an object with a specific value using the Binary Search Algorithm. We construct a tree where each node has only 2 child nodes. The *parent* is greater than the *child* on the left and its *subtree* is less than the child on the right and its subtree. In a *perfectly balanced* BST, we search down the tree, choosing left or right until we find the desired value or end at a *leaf* (producing an unsuccessful search).

In addition, a BST is adept at inserting and deleting nodes without rewriting the entire list. When dealing with a sorted list, the only way to insert a new object is by reassigning the rest of the list after the object. This has a worst-case temporal complexity of $O(n)$. On the other hand, once we have found the right node, insertion and deletion in a BST has constant time complexity. Specific algorithms also allow the user to insert and remove nodes while still preserving the structure of the BST.

These instructions will guide the user on the construction of a BST by first addressing the algorithm design. We will explain and walk through the Binary Search Algorithm. We will then address the specific structure of the BST. Next, we will cover the algorithms required for insertion and deletion. At the end, we will provide the code required to create a BST as a Python Class.

I. Binary Search Algorithm

Introduction:

The **Binary Search Algorithm** is a recursive algorithm used to find an object in a given sorted list. If the object is in the list, it will return True, otherwise it will return False.

The algorithm depends on the list being sorted. It compares the value of the object we are searching for with the "middle" element of the list. Then if our object is greater than the middle element, we only look at the right side of the list. If it is less than the middle element, we only look to the left side of the list. We then recursively repeat this process until we find the element.

Examples:

We will search the following list for the number 6 as a demonstration.

1	2	3	4	5	6	7	8	9

1. Compare 6 to the middle-indexed element (in this first case, 5)

1	2	3	4	**5**	6	7	8	9

2. Since 6 > 5, we will now consider just the following list

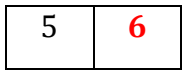
5	6	7	8	9

3. Now compare 6 to the middle-indexed element (in this list, 7)

5	6	**7**	8	9

4. Since 6 < 7, we will now consider just the following list

5. Now compare 6 to the middle-indexed element (in this list, 6)



6. Since 6 = **6**, we are done!

Notice at each step we effectively cut the list in half. This procedure is what makes this algorithm so efficient.

For the interested reader, the example we gave here does not exactly represent how it is done in code. Rather, this shows the principle of what makes the Binary Search Tree so powerful.

II. BST Structure[1]

Before introducing the structure of the BST, we must first define a few terms:

> The *children* are the outgoing neighbors of a given node, and the given node is called the parent of those children.
>
> The *root* node is a parent node with no neighbors pointing to it, and a *leaf node* is a node with no children

The Binary Search Tree is an undirected graph with the following features:

1. There is only one root
2. Each parent has at most 2 children
3. The left child of each parent is less than the parent and the right child is greater than the parent
4. There are no duplicate nodes

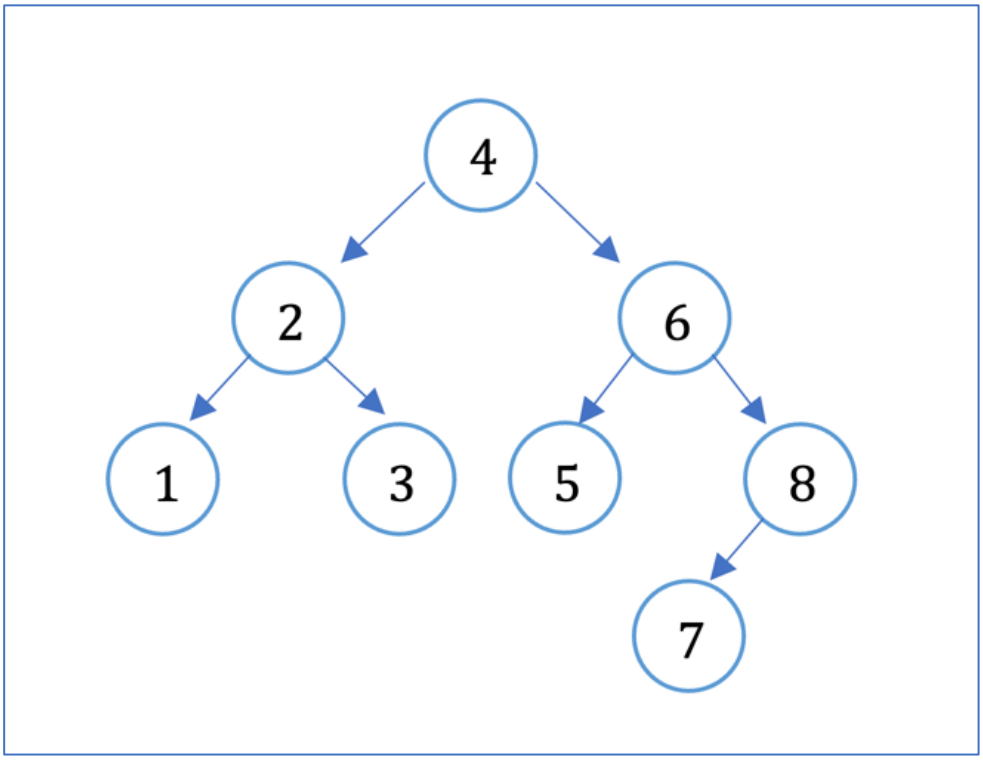


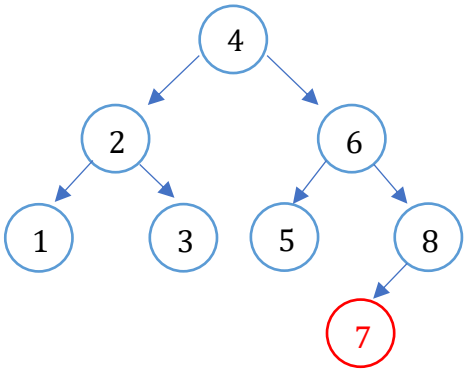
Figure 1: An example of a BST

[1] Humphreys, J., & Jarvis, T. J. (2020). Algorithms, approximation, optimization. Philadelphia: Society for Industrial and Applied Mathematics.

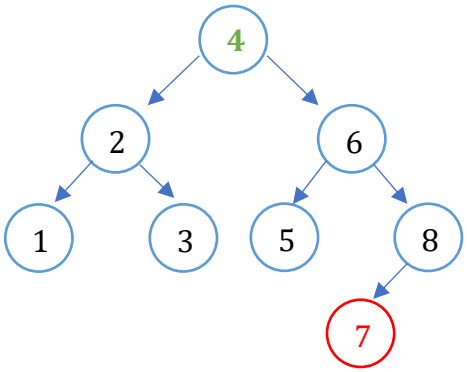
Examples:

Since the BST is ordered with every left child less than the parent and every right child greater than the parent, we can search through the tree for a given value using the Binary Search Algorithm

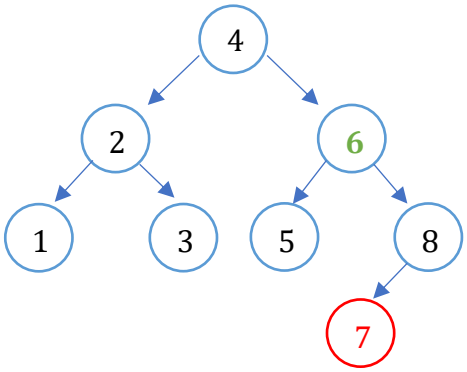
To demonstrate, consider searching for 7 in Figure 1:



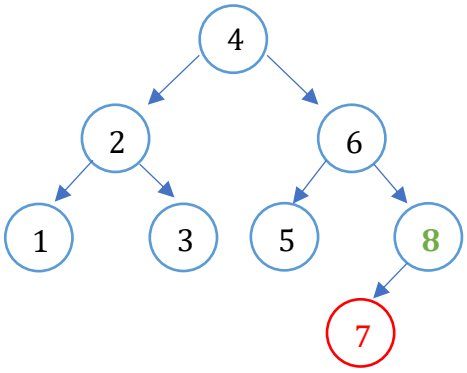
1. Start at the root node (4) and compare it to 7.



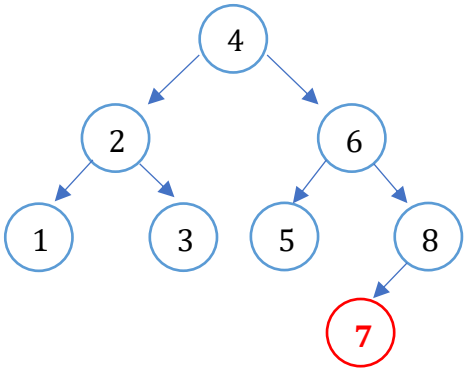
2. Since 7 > 4, move to the right child (6) and compare it with 7.



3. Since 7 > 6, move to the right child (8) and compare it with 7.



4. Since 7 < 8, move to the left child (7) and compare it with 7.



5. Since 7 = 7, we found it and we are done!

III. Insertion and Deletion

Introduction:

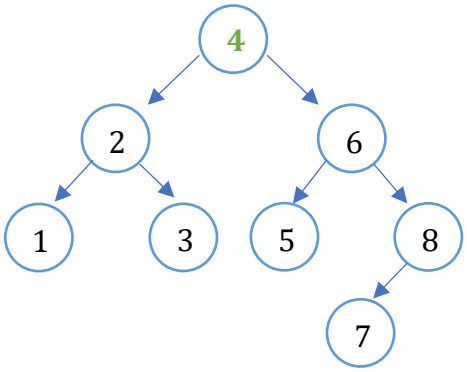
A Binary Search Tree allows for insertion and deletion in $O(\log n)$ time. This is due to the "pointing" structure of the tree. We simply reassign the pointers of the nodes to new nodes depending on the algorithm.

Insertion:

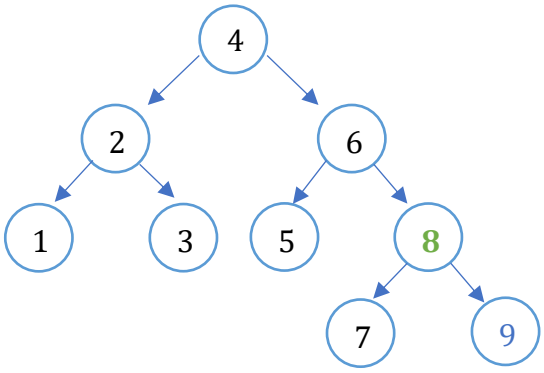
New nodes are inserted as leaf nodes. Find where they should go by the Search Algorithm, then simply add it on in the right spot

Example:

1. To add a 9, we must first find where it goes. Start at the root (**4**) and search down the tree.



2. Once we find where the node will go (in this case as the right child of **8**), add on the new node as a child.



Deletion:

Deletion is more algorithmically intense. We have three cases:

I. If the node is a leaf node, remove it

II. If the node has exactly one child, replace the node with the child and remove the node

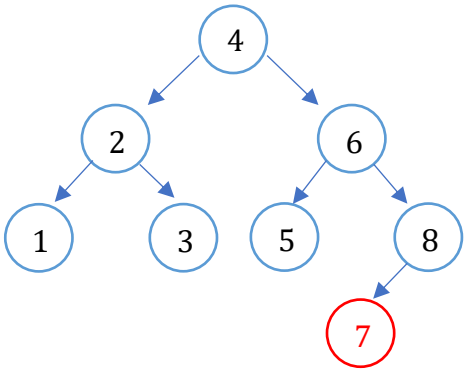
III. If the node has two or more children, replace the node with the in-order predecessor, then the node falls into Case I or Case II.

In-order predecessor: The highest valued node less than the given node. Found by going to the left child of the given node, then traversing as far down to the right as possible.

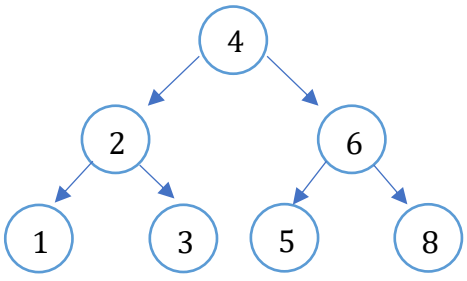
Examples:

Case I: Removing a leaf node

1. We wish to remove 7 (which is a leaf node)

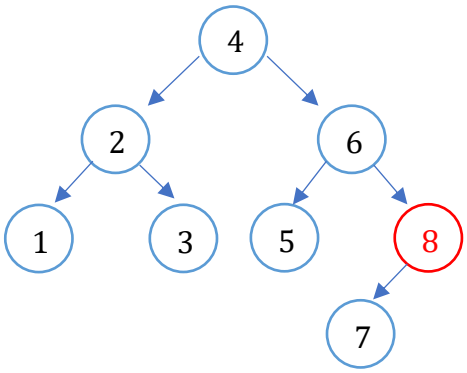


2. Simply remove it!

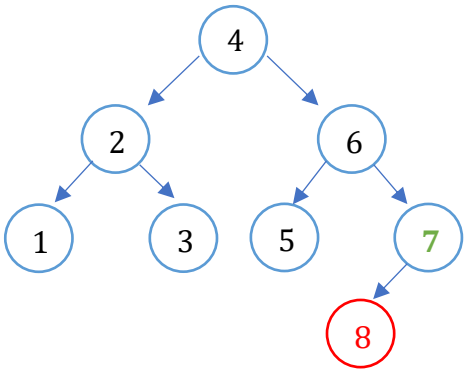


Case II: Removing a node with one child

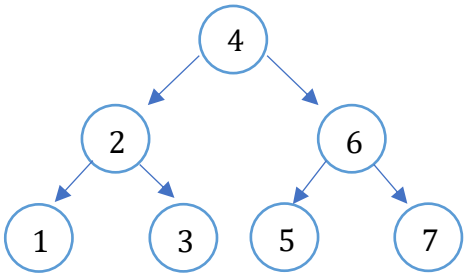
1. We wish to remove 8 (which has one child)



2. Swap the node 8 with 7



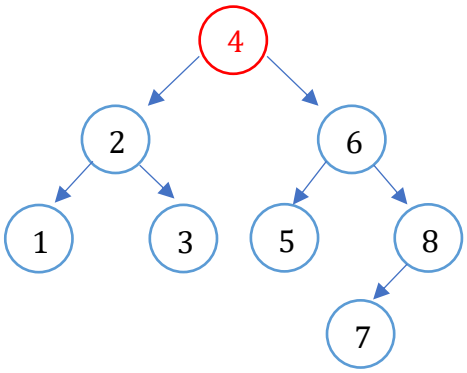
3. Remove 8



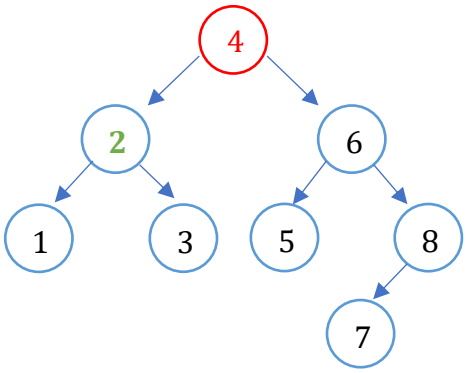
Case III: Removing a node with two children

Reminder:
If the node has two or more children, replace the node with the in-order predecessor, then the node falls into Case I or Case II.

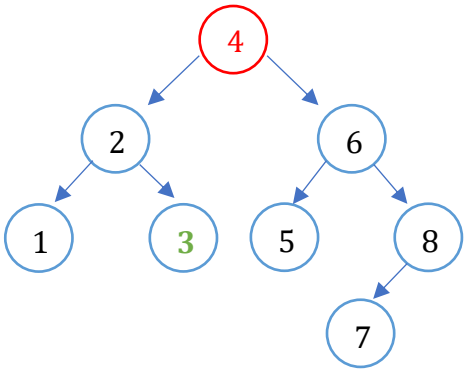
1. We wish to remove 4 (which has two children)



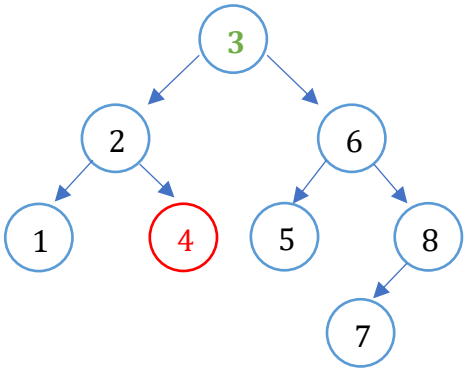
2. Do the first step of finding the in-order predecessor by going to the left child of 4 (2)



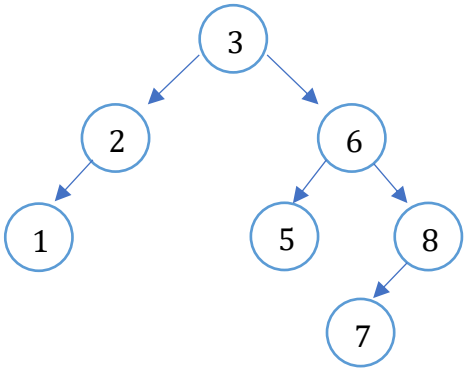
3. Now go as far right as possible from 2 (this goes to 3)



4. Swap the values of 4 with 3



5. Now remove 4



IV. Python Representation

Introduction:

In this section we will demonstrate how to implement this in Python.[2] We will use 2 classes. One class is the BST Node. The Node stores the value and references to the parent and each child. The other class is the BST itself. The BST stores which Node is the root, and it defines the methods to find, insert, and remove.

1. Initialize Classes
2. Find Method
3. Insertion Method
4. Deletion Method

1. Initialize Classes:

```
3   class BSTNode:
4       """A node class for binary search trees. Contains a value, a
5       reference to the parent node, and references to two child nodes.
6       """
7       def __init__(self, data):
8           """Construct a new node and set the value attribute. The other
9           attributes will be set when the node is added to a tree.
10          """
11          self.value = data
12          self.prev = None        # A reference to this node's parent node.
13          self.left = None        # self.left.value < self.value
14          self.right = None       # self.value < self.right.value
15
16
17  class BST:
18      """Binary search tree data structure class.
19      The root attribute references the first node in the tree.
20      """
21      def __init__(self):
22          """Initialize the root attribute."""
23          self.root = None
```

The attributes of the Node class are the .value, .prev (the parent Node), .left (the left child), and .right (the right child). The attribute of the BST class is only .root.

[2] Python represented using VS Code (https://code.visualstudio.com/)

2. Find Method

```
25      def find(self, data):
26          """Return the node containing the data. If there is no such node
27          in the tree, including if the tree is empty, raise a ValueError.
28          """
29          # Define a recursive function to traverse the tree.
30          def _step(current):
31              """Recursively step through the tree until the node containing
32              the data is found. If there is no such node, raise a Value Error.
33              """
34              if current is None:                     # Base case 1: dead end.
35                  raise ValueError(str(data) + " is not in the tree.")
36              if data == current.value:               # Base case 2: data found!
37                  return current
38              if data < current.value:                # Recursively search left.
39                  return _step(current.left)
40              else:                                   # Recursively search right.
41                  return _step(current.right)
42
43          # Start the recursion on the root of the tree.
44          return _step(self.root)
```

This method is recursive in nature. The _step(current) function checks if the value of the current node matches the data we are trying to find. It has four cases:

1. If the data is not in the BST, it raises a Value Error
2. If it finds the data, it returns the proper node.
3. If the data is less than the value of the current node, call _step(current.left)
4. If the data is greater than the value of the current node, call _step(current.right)

3. Insert Method

```
46      def insert(self, data):
47          """Insert a new node containing the specified data.
48
49          Raises:
50              ValueError: if the data is already in the tree.
51          """
52          # Initialize the data as a BST node
53          new_node = BSTNode(data)
54
55          # Base case: If the tree is empty, set the root to the new_node
56          if self.root is None:
57              self.root = new_node
58
59          # Insert the node
60          else:
61              # Find the parent
62              def _find_parent(parent, new_node):
63
64                  # If the data is already in the tree, raise a Value Error
65                  if parent.value == new_node.value:
66                      raise ValueError(str(data) + " is already in the BST")
67
68                  # If the data is less than the parent, and the parent has
69                  #   no left child, link parent and new_node
70                  # Else call the function again on the left child
71                  elif parent.value > new_node.value:
72                      if parent.left is None:
73                          parent.left = new_node
74                          new_node.prev = parent
75                          return
76                      _find_parent(parent.left, new_node)
77
78                  # If the data is greater than the parent, and the parent has
79                  #   no right child, link parent and new_node
80                  # Else call the function again on the right child
81                  elif parent.value < new_node.value:
82                      if parent.right is None:
83                          parent.right = new_node
84                          new_node.prev = parent
85                          return
86                      _find_parent(parent.right, new_node)
87
88              return _find_parent(self.root, new_node)
```

This method uses a similar recursive function to find the parent of the value we wish to insert, then updates the necessary attributes of both the new node and the parent

4. Deletion Method

Case I: Remove a leaf node

```
90      def remove(self, data):
91          """Remove the node containing the specified data.
92
93          Raises:
94              ValueError: if there is no node containing the data, including if
95                  the tree is empty.
96          """
97          # Find the node to remove. If the tree is empty or if the data is not
98          #   in any node, find raises a ValueError
99          target = self.find(data)
100
101         # Initialize the children
102         left_child = target.left
103         right_child = target.right
104
105         # If target has no children, remove the node and update the neighbors
106         if left_child is None and right_child is None:
107             if target is self.root:
108                 self.root = None
109             elif target.prev.value < target.value:
110                 target.prev.right = None
111             elif target.prev.value > target.value:
112                 target.prev.left = None
```

Note if the node we wish to remove is the root, and it is the only node, we have to reassign the .root attribute to None. In the other cases, the node is deleted by assigning its parent to None. Python's garbage collector then takes care of deleting it from memory.

Case II: Remove a node with one child

```
114         # If target has one child
115         elif left_child is None or right_child is None:
116 
117             # Set child to the one that is not None
118             child = right_child
119             if right_child is None:
120                 child = left_child
121 
122             # If the target is the root, set the root to its child
123             #   and child.prev to None
124             if target is self.root:
125                 self.root = child
126                 child.prev = None
127 
128             # If the target is to the right of its parent, update
129             #   parent.right to child and child.prev to parent
130             elif target.prev.value < target.value:
131                 target.prev.right = child
132                 child.prev = target.prev
133 
134             # If the target is to the left of its parent, update
135             #   parent.left to child and child.prev to parent
136             elif target.prev.value > target.value:
137                 target.prev.left = child
138                 child.prev = target.prev
```

Case III: Remove a node with two children

```
140         # If the target has two children
141         elif left_child != None and right_child != None:
142 
143             # Find the predecessor by going all the way to the
144             #   right from the left child
145             def _find_predecessor(node):
146                 if node.right is None:
147                     return node
148                 return _find_predecessor(node.right)
149 
150             predecessor = _find_predecessor(target.left)
151 
152             # Save the predecessor value and remove that value,
153             #   then set the target node to the predecessor value
154             predecessor_value = predecessor.value
155             self.remove(predecessor_value)
156             target.value = predecessor_value
157         return
```