# Introduction

The problem we are faced with today is constructing an image with python of a BST (binary search tree), creating a new and iterative version of the search function provided to us, constructing a balanced BST from a sorted list, extracting elements from a BST into a sorted list, and printing out elements from a BST by ordered depth.

# Problem 1

This problem was the one that gave me the most trouble due to how different it was to what we have worked on recently. The route I chose to take was to use the previous code we had from lab 1, which we drew a BST. I learned 2 new things, a circle.plot function, and text plotting into the graph. Using those 2 new things, I modified my previous code to work with the T node we use to draw.

# Problem 2

This second problem was not to complicated, but it is a tricky to do if you do not understand trees a whole lot yet. The first thing I did was check if K was already in the tree, or if T was none. If either of those were true, the method would return reference to Node T. The next thing I did was create a while loop that would iterate as long as T was not None. The first case I used was an if statement that checked if K was in T.item, and would return the reference to T. The next case I did was check if K>t.item and that would cause us to iterate to the right side if true, t is set equal to t.right. If None of the if statements were true, the next thing that would happen would be an iteration to the left side that updates t to be equal to T.left.

# Problem 3

The way this problem was solved was very time consuming because it took a lot of debugging to finally get it to work. The first thing we did was check the length of the array, and made sure that the length was greater than 0. Once we check that, we begin by splitting the array in half. We had 2 if statements that checked for the len of the ordered list and checked if it was equal to 1 or to 2. In the first if statement(len = 1), we add the number to the node and we return the node. In the second if statement(len = 2), we add the number a[mid] to the node, and then we add a[0] to the node.left of the balance tree and then we return the node. Next, we update our node.item to be equal to the mid item of the sorted list. After that we we only have 2 recursive calls that update the node to go into the direction that we are updating, if we update .left, we recursively send .left, and we splice the list to send the left side of the list, excluding the mid point that we used to update our node, then we return.

# Problem 4

The first base case we have is to check if the node is none, and if it is, we return the list. The next case makes sure our node is not none and recursively allows us to iterate to the left side first, and to the right side after (this is done to get the smallest first and then the larger numbers) and we append to the list, and return.

# Problem 5

For problem five we created a base case that makes sure to only allow us to work if the Node is not None. Next we print our I element and our T.item. After all that, we iterate recursively to the node.left and to the node.right, and we update I+1.

# Code

1. # Code to implement a binary search tree
2. # Programmed by Cesar Lopez
3. # Last modified February 27, 2019
4. **import** matplotlib.pyplot as plt
6. **class** BST(object):
7. # Constructor
8. **def** \_\_init\_\_(self, item, left=None, right=None):
9. self.item = item
10. self.left = left
11. self.right = right
13. **def** Insert(T,newItem):
14. **if** T == None:
15. T =  BST(newItem)
16. **elif** T.item > newItem:
17. T.left = Insert(T.left,newItem)
18. **else**:
19. T.right = Insert(T.right,newItem)
20. **return** T
22. **def** Delete(T,del\_item):
23. **if** T **is** **not** None:
24. **if** del\_item < T.item:
25. T.left = Delete(T.left,del\_item)
26. **elif** del\_item > T.item:
27. T.right = Delete(T.right,del\_item)
28. **else**:  # del\_item == T.item
29. **if** T.left **is** None **and** T.right **is** None: # T is a leaf, just remove it
30. T = None
31. **elif** T.left **is** None: # T has one child, replace it by existing child
32. T = T.right
33. **elif** T.right **is** None:
34. T = T.left
35. **else**: # T has two chldren. Replace T by its successor, delete successor
36. m = Smallest(T.right)
37. T.item = m.item
38. T.right = Delete(T.right,m.item)
39. **return** T
41. **def** InOrder(T):
42. # Prints items in BST in ascending order
43. **if** T **is** **not** None:
44. InOrder(T.left)
45. **print**(T.item,end = ' ')
46. InOrder(T.right)
48. **def** InOrderD(T,space):
49. # Prints items and structure of BST
50. **if** T **is** **not** None:
51. InOrderD(T.right,space+'   ')
52. **print**(space,T.item)
53. InOrderD(T.left,space+'   ')
55. **def** SmallestL(T):
56. # Returns smallest item in BST. Returns None if T is None
57. **if** T **is** None:
58. **return** None
59. **while** T.left **is** **not** None:
60. T = T.left
61. **return** T
63. **def** Smallest(T):
64. # Returns smallest item in BST. Error if T is None
65. **if** T.left **is** None:
66. **return** T
67. **else**:
68. **return** Smallest(T.left)
70. **def** Largest(T):
71. **if** T.right **is** None:
72. **return** T
73. **else**:
74. **return** Largest(T.right)
76. **def** Find(T,k):
77. # Returns the address of k in BST, or None if k is not in the tree
78. **if** T **is** None **or** T.item == k:
79. **return** T
80. **if** T.item<k:
81. **return** Find(T.right,k)
82. **return** Find(T.left,k)
84. **def** FindIter(T, k):
85. **if** T **is** None **or** T.item == k:
86. **return** T
87. **while** T **is** **not** None:
88. **if** T.item==k:
89. **return** T
90. **if** T.item < k:
91. T = T.right
92. **else**:
93. T = T.left
94. **return** None
96. **def** FindAndPrint(T,k):
97. f = Find(T,k)
98. **if** f **is** **not** None:
99. **print**(f.item,'found')
100. **else**:
101. **print**(k,'not found')
103. **def** Drawing(ax, x, y, T, r):
104. #makes sure to iterate only when our node is not none
105. **if** T **is** **not** None:
106. c= plt.Circle([x,y], .5, color='k', fill=False)#Creates a circle which we use for out tree
107. ax.add\_artist(c)#adds the circle into the figure
108. ax.text(x-.2, y+.2, T.item, size=10) #we use this to place our text into the image
109. **if** T.left **is** **not** None:
110. #with this line of code, we plot our left side of the tree
111. ax.plot((x, x-r),(y,y-2),color='k')
112. #checks our right side, to plot it.
113. **if** T.right **is** **not** None:
114. #with this line of code, we plot our right side of the tree
115. ax.plot((x, x+r),(y,y-2),color='k')
116. #this is our first recursive case, which recurses to the right side of the binary tree
117. Drawing(ax, x+r, y-2, T.right, r/2)
118. #this is our second recursive case, which recurses to the left side of the binary tree
119. Drawing(ax, x-r, y-2, T.left, r/2)
120. #draws our left side
121. #if right side is none, we use this to plot left side
122. **if** T.left **is** **not** None:
123. #with this line of code, we plot our left side of the tree
124. Drawing(ax, x-r, y-2, T.left, r/2)
126. **def** Balance(a, z):
127. #only allows us to work on the list if the list is greater than 0
128. **if**(len(a) > 0):
129. #creates mid to help us get our pivot which is the middle value of the list
130. mid = (len(a)//2)
131. #checks len if it is 1 because then we will only have 1 value in the list
132. **if** len(a) == 1:
133. z = BST(a[mid])
134. **return** z
135. #checks if len is 2 because we will have 2 values, we append the last value first, and then the 1st value to left because it is less than the first in the list.
136. **if** len(a) == 2:
137. z = BST(a[mid])
138. z.left = BST(a[0])
139. **return** z
140. #we append our middle value to our balance tree, and next we iterate recursively to the left and the right with z.left, z.right
141. #we also use list splicing to split our list, we make sure to avoid using the middle value again to make sure we do not add it twice to the node
142. z = BST(a[mid])
143. z.left = Balance(a[0:mid], z.left)
144. z.right = Balance(a[mid+1:], z.right)
145. **return** z
147. **def** LFT(T, a):
148. #checks if our node is none, if it is, we return our list
149. **if** T **is** None:
150. **return** a
151. #this allows us to operate on the list as long as it is not none.
152. #we recursively go to the left side first and append our left most item first because it is the smalles and we build up from there
153. #next we append our item value that we have to get our middle
154. #and last we iterate to the right to get the values that our to the right
155. **if** T **is** **not** None:
156. LFT(T.left, a)
157. a.append(T.item)
158. LFT(T.right, a)
159. **return** a

162. **def** KeysAtDepths(T, i):
163. **if** T **is** **not** None:
164. **print**("Keys at depth ",i,": ",T.item)
165. KeysAtDepths(T.right,i+1)
166. KeysAtDepths(T.left,i+1)

169. # Code to test the functions above
170. T = None
171. A = [70, 50, 90, 130, 150, 40, 10, 30, 100, 180, 45, 60, 140, 42]
172. **for** a **in** A:
173. T = Insert(T,a)
175. InOrder(T)
176. **print**()
177. InOrderD(T,'')
178. **print**()
180. **print**(SmallestL(T).item)
181. **print**(Smallest(T).item)
183. FindAndPrint(T,40)
184. FindAndPrint(T,110)
186. n=60
187. **print**('Delete',n,'Case 1, deleted node is a leaf')
188. T = Delete(T,n) #Case 1, deleted node is a leaf
189. InOrderD(T,'')
190. **print**('####################################')
192. n=90
193. **print**('Delete',n,'Case 2, deleted node has one child')
194. T = Delete(T,n) #Case 2, deleted node has one child
195. InOrderD(T,'')
196. **print**('####################################')
198. n=70
199. **print**('Delete',n,'Case 3, deleted node has two children')
200. T = Delete(T,n) #Case 3, deleted node has two children
201. InOrderD(T,'')
203. n=40
204. **print**('Delete',n,'Case 3, deleted node has two children')
205. T = Delete(T,n) #Case 3, deleted node has two children
206. InOrderD(T,'')
207. **print**("search in iterator way ###############")
208. **print**(FindIter(T, 50))
209. **print**("print D ##############")
210. D = None
211. ne = [10, 4, 15, 2, 3, 8, 5, 7, 9, 1, 12, 18]
212. **for** NE **in** ne:
213. D = Insert(D, NE)
214. InOrderD(D,'')
215. **print**("tree of D #############")
216. plt.close("all")
217. fig, ax = plt.subplots()
218. x = 5
219. Drawing(ax, x, 10, D, x/2)
220. **print**("B tree ##############")
221. order = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
222. z = BST
223. z = Balance(order, z)
224. InOrder(z)
225. **print**("")
226. InOrderD(z,'')
227. **print**("List from Tree ###########")
228. empty = []
229. empty = LFT(T, empty)
230. **print**(empty)
231. **print**("Keys at depth ##############")
232. KeysAtDepths(T, 0)

# Running Times

X is the size of our list, and y is the time it took to operate with that list size

# Problem 1

# Problem 2

# Problem 3

# Problem 4

# Problem 5