Documentation

Most of the binary (search) tree functions were already explained in the lecture and slides, so I will not be talking about all of them, just the ones that I had to implement myself and any others I feel are notable.

NodeTree::removeValue

This function is very similar to the BST version, except there is no way to search for a specific value in the tree besides looking at all the nodes. To accomplish this, I had to make two recursive calls, one for the left and right subtrees, and the recursion (should) break once the item was found. One weakness in my implementation is that the search would still continue even after the item was found, but this could easily be fixed by putting a break condition if the item was found. I realize now that this function could have made use of findNode in order to get the node to remove, but I had implemented this function before I had implemented findNode so I didn’t think to use it.

NodeTree::moveValuesUpTree

This function was a little bit tricky to implement, however it does have many similarities to the BST version, and that made it a little easier. The main problem was figuring out how to replace a node without breaking the tree. I figured out that if the node to replace had two children, you needed to construct a new node that was also the parent of one of its children, and then make another recursive call to attach the other child.

NodeTree::findNode

findNode was similar to removeValue in that I had to make two recursive calls to both the left and right subtrees in order to find the node. I did implement a break case in this function because the NodeTree should only have unique values, and I didn’t want to keep searching in case the value was already found.

Output

Here is an example output from an instance of my program:

```
Input data:
111	20	57	15	167	186	179	2	1	10	64	132	6	5	27	184	170	158
161	39	142	190	75	26	4	129	24	56	76	178	139	113	116	62	90
127	136	200	18	107	130	72	31	123	32	137	146	177	35	199	51
126	38	25	114	160	156	87	143	175	118	82	22	43	176	106	83	52
13	66	117	77	131	19	150	34	119	36	194	197	100	79	165	55	183
102	80	86	61	120	182	94	138	162	112	181	48	45	115	154
Height:12
Inorder:
1	2	4	5	6	10	13	15	18	19	20	22	24	25	26	27	31	32	34	35
36	38	39	43	45	48	51	52	55	56	57	61	62	64	66	72	75	76	77
79	80	82	83	86	87	90	94	100	102	106	107	111	112	113	114	115
116	117	118	119	120	123	126	127	129	130	131	132	136	137	138
```

139 142 143 146 150 154 156 158 160 161 162 165 167 170 175
176 177 178 179 181 182 183 184 186 190 194 197 199 200

Preorder:

111 20 15 2 1 10 6 5 4 13 18 19 57 27 26 24 22 25 39 31
32 35 34 38 36 56 51 43 48 45 52 55 64 62 61 75 72 66 76
90 87 82 77 79 80 83 86 107 106 100 94 102 167 132 129 113
112 116 114 115 127 123 118 117 119 120 126 130 131 158 142
139 136 137 138 146 143 156 150 154 161 160 165 162 186 179
170 178 177 175 176 184 183 182 181 190 200 199 194 197

Postorder:

1 4 5 6 13 10 2 19 18 15 22 25 24 26 34 36 38 35 32 31
45 48 43 55 52 51 56 39 27 61 62 66 72 80 79 77 86 83 82
87 94 102 100 106 107 90 76 75 64 57 20 112 115 114 117 120
119 118 126 123 127 116 113 131 130 129 138 137 136 139 143
154 150 156 146 142 160 162 165 161 158 132 176 175 177 178
170 181 182 183 184 179 197 194 199 200 190 186 167 111

A simple way to test the correctness of the output is to note that preorder will list the root first, while postorder will show the root last. Inorder of course, shows all the elements in (ascending) order.