

Homework #7

Analysis of Algorithms

Illya Starikov

Due Date: October 20th 2017

```
1 class Node:
2     value = -1
3     left_child, right_child = None, None
4
5     def __init__(self, value):
6         self.value = value
7         self.left_child, self.right_child = None, None
8
9     @property
10    def number_of_children(self):
11        if self.left_child is not None and self.right_child is not None:
12            return 2
13        elif self.left_child is not None or self.right_child is not None:
14            return 1
15        else:
16            return 0
17
18    @property
19    def valid_children(self):
20        number_of_children = self.number_of_children
21
22        if number_of_children == 0:
23            return None
24        elif number_of_children == 1:
25            if self.left_child is not None:
26                return self.left_child
27            else:
28                return self.right_child
29        else:
30            return (self.left_child, self.right_child)
31
32    def __str__(self):
33        return str(self.value)
34
35
36 class BinarySearchTree:
```

```

37     root = -1
38
39     def __init__(self):
40         self.root = None
41
42     def insert(self, value):
43         if self.root is None:
44             self.root = Node(value)
45         else:
46             self.__insert_node(self.root, value)
47
48     def exists(self, value):
49         return False if self.__find_node(self.root, value) is None else
True
50
51     def delete(self, value):
52         self.root = self.__delete_node(self.root, value)
53
54     def print_tree(self):
55         if not self.root:
56             return
57
58         current_level = [self.root]
59
60         while current_level:
61
62             print(' '.join(str(node) for node in current_level))
63
64             next_level = list()
65             for n in current_level:
66                 if n.left_child:
67                     next_level.append(n.left_child)
68                 if n.right_child:
69                     next_level.append(n.right_child)
70             current_level = next_level
71
72     # MARK: Private Methods
73     def __insert_node(self, current_node, value):
74         # for the values <= `value`, we put on the left side of the tree
75         if value <= current_node.value:
76             if current_node.left_child is None:
77                 current_node.left_child = Node(value)
78             else:
79                 self.__insert_node(current_node.left_child, value)
80
81         # for values > `value`, we put on the right side of the tree
82         else:
83             if current_node.right_child is None:
84                 current_node.right_child = Node(value)
85             else:
86                 self.__insert_node(current_node.right_child, value)
87

```

```

88     def __find_node(self, current_node, value):
89         if current_node is None:
90             return None
91         elif value < current_node.value:
92             return self.__find_node(current_node.left_child, value)
93         elif value > current_node.value:
94             return self.__find_node(current_node.right_child, value)
95         else:
96             return current_node
97
98     def __delete_node(self, current_node, key):
99         if not current_node:
100             return current_node
101
102         if current_node.value > key:
103             current_node.left_child = self.__delete_node(current_node.
left_child, key)
104         elif current_node.value < key:
105             current_node.right_child = self.__delete_node(current_node.
right_child, key)
106         else:
107             if not current_node.left_child:
108                 right_child = current_node.right_child
109                 del current_node
110                 return right_child
111
112             elif not current_node.right_child:
113                 left_child = current_node.left_child
114                 del current_node
115                 return left_child
116
117             else:
118                 successor = current_node.right_child
119                 while successor.left_child:
120                     successor = successor.left_child
121
122                 current_node.value = successor.value
123                 current_node.right_child = self.__delete_node(current_node
.right_child, successor.value)
124
125         return current_node

```

Question #1

With the following elements,

42	2555	4830	7516
104	2718	4883	7604
321	2849	4961	7632
578	2954	5016	7706
600	2974	5119	7754
734	3063	5136	7989
929	3159	5246	8106
1004	3262	5253	8126
1120	3397	5502	8135
1128	3405	5520	8204
1149	3476	5550	8223
1213	3694	5644	8338
1323	3739	5675	8359
1347	3902	6082	8565
1512	4310	6306	8611
1522	4466	6425	8740
1730	4475	6562	8890
1831	4494	6570	8961
1853	4584	6653	9300
1886	4644	6712	9304
1899	4699	6940	9509
2040	4750	6966	9614
2082	4769	7164	9762
2164	4783	7324	9971
2253	4786	7499	9985

Problem #1.1

Determining if elements in the binary search tree are there. All these values should be true.

1	Element	3739	Found:	True
2	Element	5550	Found:	True
3	Element	600	Found:	True
4	Element	3262	Found:	True
5	Element	8611	Found:	True
6	Element	4750	Found:	True
7	Element	5246	Found:	True
8	Element	6712	Found:	True
9	Element	1323	Found:	True
10	Element	1831	Found:	True
11	Element	5520	Found:	True
12	Element	3063	Found:	True
13	Element	9300	Found:	True
14	Element	1004	Found:	True
15	Element	7164	Found:	True
16	Element	4466	Found:	True
17	Element	8890	Found:	True

18	Element	8338	Found: True
19	Element	1853	Found: True
20	Element	8223	Found: True
21	Element	3159	Found: True
22	Element	4644	Found: True
23	Element	7706	Found: True
24	Element	8740	Found: True
25	Element	1120	Found: True
26	Element	2040	Found: True
27	Element	6653	Found: True
28	Element	578	Found: True
29	Element	9762	Found: True
30	Element	8565	Found: True
31	Element	7604	Found: True
32	Element	9985	Found: True
33	Element	4961	Found: True
34	Element	929	Found: True
35	Element	2082	Found: True
36	Element	5253	Found: True
37	Element	3405	Found: True
38	Element	7516	Found: True
39	Element	6306	Found: True
40	Element	8106	Found: True
41	Element	1347	Found: True
42	Element	4310	Found: True
43	Element	4584	Found: True
44	Element	2954	Found: True
45	Element	9614	Found: True
46	Element	2253	Found: True
47	Element	1522	Found: True
48	Element	8135	Found: True
49	Element	4475	Found: True
50	Element	1512	Found: True
51	Element	9304	Found: True
52	Element	6966	Found: True
53	Element	321	Found: True
54	Element	2974	Found: True
55	Element	6940	Found: True
56	Element	7754	Found: True
57	Element	1213	Found: True
58	Element	1149	Found: True
59	Element	8961	Found: True
60	Element	8359	Found: True
61	Element	4883	Found: True
62	Element	7499	Found: True
63	Element	4786	Found: True
64	Element	8126	Found: True
65	Element	42	Found: True
66	Element	4699	Found: True
67	Element	3902	Found: True
68	Element	1730	Found: True
69	Element	4494	Found: True

70	Element	5502	Found: True
71	Element	104	Found: True
72	Element	4830	Found: True
73	Element	6082	Found: True
74	Element	1899	Found: True
75	Element	5136	Found: True
76	Element	5644	Found: True
77	Element	2849	Found: True
78	Element	5675	Found: True
79	Element	8204	Found: True
80	Element	7632	Found: True
81	Element	1886	Found: True
82	Element	5119	Found: True
83	Element	5016	Found: True
84	Element	6562	Found: True
85	Element	3397	Found: True
86	Element	9971	Found: True
87	Element	3694	Found: True
88	Element	4783	Found: True
89	Element	2718	Found: True
90	Element	734	Found: True
91	Element	7989	Found: True
92	Element	7324	Found: True
93	Element	1128	Found: True
94	Element	6425	Found: True
95	Element	4769	Found: True
96	Element	2555	Found: True
97	Element	3476	Found: True
98	Element	9509	Found: True
99	Element	2164	Found: True
100	Element	6570	Found: True

Problem #1.2

Determining if elements **not** in binary search tree are found. These values should be false.

1	Element	9170	Found: False
2	Element	1059	Found: False
3	Element	6807	Found: False
4	Element	7990	Found: False
5	Element	9177	Found: False
6	Element	9933	Found: False
7	Element	9290	Found: False
8	Element	5479	Found: False
9	Element	611	Found: False
10	Element	5458	Found: False
11	Element	1563	Found: False
12	Element	5870	Found: False
13	Element	3134	Found: False
14	Element	178	Found: False
15	Element	3890	Found: False

16	Element	7225	Found: False
17	Element	9820	Found: False
18	Element	2292	Found: False
19	Element	5313	Found: False
20	Element	4489	Found: False
21	Element	3835	Found: False
22	Element	2543	Found: False
23	Element	5679	Found: False
24	Element	46	Found: False
25	Element	1812	Found: False
26	Element	4628	Found: False
27	Element	7310	Found: False
28	Element	6839	Found: False
29	Element	5411	Found: False
30	Element	2929	Found: False
31	Element	2351	Found: False
32	Element	9107	Found: False
33	Element	7498	Found: False
34	Element	5810	Found: False
35	Element	6692	Found: False
36	Element	3986	Found: False
37	Element	2941	Found: False
38	Element	9506	Found: False
39	Element	3485	Found: False
40	Element	1113	Found: False
41	Element	4268	Found: False
42	Element	9848	Found: False
43	Element	1525	Found: False
44	Element	7304	Found: False
45	Element	1792	Found: False
46	Element	2707	Found: False
47	Element	4111	Found: False
48	Element	7696	Found: False
49	Element	4897	Found: False
50	Element	120	Found: False
51	Element	9588	Found: False
52	Element	819	Found: False
53	Element	5893	Found: False
54	Element	9990	Found: False
55	Element	6444	Found: False
56	Element	2619	Found: False
57	Element	3091	Found: False
58	Element	8431	Found: False
59	Element	7965	Found: False
60	Element	4863	Found: False
61	Element	9778	Found: False
62	Element	9601	Found: False
63	Element	8084	Found: False
64	Element	7459	Found: False
65	Element	7299	Found: False
66	Element	1	Found: False
67	Element	6819	Found: False

```

68 Element 9085 Found: False
69 Element 5574 Found: False
70 Element 8748 Found: False
71 Element 8790 Found: False
72 Element 6042 Found: False
73 Element 5852 Found: False
74 Element 5341 Found: False
75 Element 7671 Found: False
76 Element 8456 Found: False
77 Element 1818 Found: False
78 Element 6918 Found: False
79 Element 9723 Found: False
80 Element 3980 Found: False
81 Element 6496 Found: False
82 Element 5121 Found: False
83 Element 2553 Found: False
84 Element 7316 Found: False
85 Element 1461 Found: False
86 Element 3896 Found: False
87 Element 9382 Found: False
88 Element 8610 Found: False
89 Element 5181 Found: False
90 Element 3305 Found: False
91 Element 549 Found: False
92 Element 7306 Found: False
93 Element 4612 Found: False
94 Element 9525 Found: False
95 Element 4847 Found: False
96 Element 6420 Found: False
97 Element 2643 Found: False
98 Element 4538 Found: False
99 Element 5934 Found: False
100 Element 7969 Found: False

```

Problem #1.3

Deleting entire tree. This output is ≈ 2000 , so please refer to output for entire tree.

```

1 Current Tree Before Deleting Element 4769
2 3739
3 2974 7632
4 1149 3405 4783 9509
5 734 1899 3063 3694 4475 6562 8890 9985
6 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
7 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
   8565 8961 9614
8 321 929 1213 1853 2040 4466 4644 4769 4830 5520 6082 6653 7516 8126 8611
   9300 9762
9 578 1323 1831 1886 2718 4699 4786 4883 5502 5550 6306 6570 7164 7989 8223
10 600 1730 2253 4961 5253 6425 6940 7706 8106 8135 8338
11 2164 2555 5136 6712 6966 7754 8204 8359
12 2082 5119 5246

```



```

13
14 Current Tree After Deleting Element 4769
15 3739
16 2974 7632
17 1149 3405 4783 9509
18 734 1899 3063 3694 4475 6562 8890 9985
19 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
20 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
21 321 929 1213 1853 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
22 578 1323 1831 1886 2718 4699 4786 4883 5502 5550 6306 6570 7164 7989 8223
23 600 1730 2253 4961 5253 6425 6940 7706 8106 8135 8338
24 2164 2555 5136 6712 6966 7754 8204 8359
25 2082 5119 5246
26 -----

27 Current Tree Before Deleting Element 1853
28 3739
29 2974 7632
30 1149 3405 4783 9509
31 734 1899 3063 3694 4475 6562 8890 9985
32 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
33 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
34 321 929 1213 1853 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
35 578 1323 1831 1886 2718 4699 4786 4883 5502 5550 6306 6570 7164 7989 8223
36 600 1730 2253 4961 5253 6425 6940 7706 8106 8135 8338
37 2164 2555 5136 6712 6966 7754 8204 8359
38 2082 5119 5246
39
40 Current Tree After Deleting Element 1853
41 3739
42 2974 7632
43 1149 3405 4783 9509
44 734 1899 3063 3694 4475 6562 8890 9985
45 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
46 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
47 321 929 1213 1886 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
48 578 1323 1831 2718 4699 4786 4883 5502 5550 6306 6570 7164 7989 8223
49 600 1730 2253 4961 5253 6425 6940 7706 8106 8135 8338
50 2164 2555 5136 6712 6966 7754 8204 8359
51 2082 5119 5246
52 -----

53 Current Tree Before Deleting Element 4699
54 3739
55 2974 7632
56 1149 3405 4783 9509

```

```

57 734 1899 3063 3694 4475 6562 8890 9985
58 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
59 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
   8565 8961 9614
60 321 929 1213 1886 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
   9762
61 578 1323 1831 2718 4699 4786 4883 5502 5550 6306 6570 7164 7989 8223
62 600 1730 2253 4961 5253 6425 6940 7706 8106 8135 8338
63 2164 2555 5136 6712 6966 7754 8204 8359
64 2082 5119 5246
65
66 Current Tree After Deleting Element 4699
67 3739
68 2974 7632
69 1149 3405 4783 9509
70 734 1899 3063 3694 4475 6562 8890 9985
71 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
72 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
   8565 8961 9614
73 321 929 1213 1886 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
   9762
74 578 1323 1831 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
75 600 1730 2253 4961 5253 6425 6940 7706 8106 8135 8338
76 2164 2555 5136 6712 6966 7754 8204 8359
77 2082 5119 5246
78 -----
79 Current Tree Before Deleting Element 2082
80 3739
81 2974 7632
82 1149 3405 4783 9509
83 734 1899 3063 3694 4475 6562 8890 9985
84 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
85 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
   8565 8961 9614
86 321 929 1213 1886 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
   9762
87 578 1323 1831 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
88 600 1730 2253 4961 5253 6425 6940 7706 8106 8135 8338
89 2164 2555 5136 6712 6966 7754 8204 8359
90 2082 5119 5246
91
92 Current Tree After Deleting Element 2082
93 3739
94 2974 7632
95 1149 3405 4783 9509
96 734 1899 3063 3694 4475 6562 8890 9985
97 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
98 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
   8565 8961 9614
99 321 929 1213 1886 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
   9762

```

```

100 578 1323 1831 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
101 600 1730 2253 4961 5253 6425 6940 7706 8106 8135 8338
102 2164 2555 5136 6712 6966 7754 8204 8359
103 5119 5246
104 -----

105 Current Tree Before Deleting Element 4961
106 3739
107 2974 7632
108 1149 3405 4783 9509
109 734 1899 3063 3694 4475 6562 8890 9985
110 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
111 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
112 321 929 1213 1886 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
113 578 1323 1831 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
114 600 1730 2253 4961 5253 6425 6940 7706 8106 8135 8338
115 2164 2555 5136 6712 6966 7754 8204 8359
116 5119 5246
117
118 Current Tree After Deleting Element 4961
119 3739
120 2974 7632
121 1149 3405 4783 9509
122 734 1899 3063 3694 4475 6562 8890 9985
123 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
124 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
125 321 929 1213 1886 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
126 578 1323 1831 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
127 600 1730 2253 5253 6425 6940 7706 8106 8135 8338
128 2164 2555 5136 6712 6966 7754 8204 8359
129 5119 5246
130 -----

131 Current Tree Before Deleting Element 1512
132 3739
133 2974 7632
134 1149 3405 4783 9509
135 734 1899 3063 3694 4475 6562 8890 9985
136 42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
137 104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
138 321 929 1213 1886 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
139 578 1323 1831 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
140 600 1730 2253 5253 6425 6940 7706 8106 8135 8338
141 2164 2555 5136 6712 6966 7754 8204 8359
142 5119 5246
143

```

```

144 Current Tree After Deleting Element 1512
145 3739
146 2974 7632
147 1149 3405 4783 9509
148 734 1899 3063 3694 4475 6562 8890 9985
149 42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
150 104 1004 1128 1347 1886 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
151 321 929 1213 1831 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
152 578 1323 1730 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
153 600 2253 5253 6425 6940 7706 8106 8135 8338
154 2164 2555 5136 6712 6966 7754 8204 8359
155 5119 5246
156 -----

```

```

157 Current Tree Before Deleting Element 1730
158 3739
159 2974 7632
160 1149 3405 4783 9509
161 734 1899 3063 3694 4475 6562 8890 9985
162 42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
163 104 1004 1128 1347 1886 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
164 321 929 1213 1831 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
165 578 1323 1730 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
166 600 2253 5253 6425 6940 7706 8106 8135 8338
167 2164 2555 5136 6712 6966 7754 8204 8359
168 5119 5246
169

```

```

170 Current Tree After Deleting Element 1730
171 3739
172 2974 7632
173 1149 3405 4783 9509
174 734 1899 3063 3694 4475 6562 8890 9985
175 42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
176 104 1004 1128 1347 1886 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
177 321 929 1213 1831 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
178 578 1323 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
179 600 2253 5253 6425 6940 7706 8106 8135 8338
180 2164 2555 5136 6712 6966 7754 8204 8359
181 5119 5246
182 -----

```

```

183 Current Tree Before Deleting Element 2849
184 3739
185 2974 7632
186 1149 3405 4783 9509
187 734 1899 3063 3694 4475 6562 8890 9985

```

```

188 42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
189 104 1004 1128 1347 1886 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
190 321 929 1213 1831 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
191 578 1323 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
192 600 2253 5253 6425 6940 7706 8106 8135 8338
193 2164 2555 5136 6712 6966 7754 8204 8359
194 5119 5246
195
196 Current Tree After Deleting Element 2849
197 3739
198 2974 7632
199 1149 3405 4783 9509
200 734 1899 3063 3694 4475 6562 8890 9985
201 42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
202 104 1004 1128 1347 1886 2040 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
203 321 929 1213 1831 2718 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
204 578 1323 2253 4786 4883 5502 5550 6306 6570 7164 7989 8223
205 600 2164 2555 5253 6425 6940 7706 8106 8135 8338
206 5136 6712 6966 7754 8204 8359
207 5119 5246
208 -----

209 Current Tree Before Deleting Element 5520
210 3739
211 2974 7632
212 1149 3405 4783 9509
213 734 1899 3063 3694 4475 6562 8890 9985
214 42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
215 104 1004 1128 1347 1886 2040 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
216 321 929 1213 1831 2718 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
217 578 1323 2253 4786 4883 5502 5550 6306 6570 7164 7989 8223
218 600 2164 2555 5253 6425 6940 7706 8106 8135 8338
219 5136 6712 6966 7754 8204 8359
220 5119 5246
221
222 Current Tree After Deleting Element 5520
223 3739
224 2974 7632
225 1149 3405 4783 9509
226 734 1899 3063 3694 4475 6562 8890 9985
227 42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
228 104 1004 1128 1347 1886 2040 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
229 321 929 1213 1831 2718 4466 4644 4830 5550 6082 6653 7516 8126 8611 9300
    9762
230 578 1323 2253 4786 4883 5502 6306 6570 7164 7989 8223

```

```

231 600 2164 2555 5253 6425 6940 7706 8106 8135 8338
232 5136 6712 6966 7754 8204 8359
233 5119 5246
234 -----

235 Current Tree Before Deleting Element 8890
236 3739
237 2974 7632
238 1149 3405 4783 9509
239 734 1899 3063 3694 4475 6562 8890 9985
240 42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
241 104 1004 1128 1347 1886 2040 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
242 321 929 1213 1831 2718 4466 4644 4830 5550 6082 6653 7516 8126 8611 9300
    9762
243 578 1323 2253 4786 4883 5502 6306 6570 7164 7989 8223
244 600 2164 2555 5253 6425 6940 7706 8106 8135 8338
245 5136 6712 6966 7754 8204 8359
246 5119 5246
247
248 Current Tree After Deleting Element 8890
249 3739
250 2974 7632
251 1149 3405 4783 9509
252 734 1899 3063 3694 4475 6562 8961 9985
253 42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
254 104 1004 1128 1347 1886 2040 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 9300 9614
255 321 929 1213 1831 2718 4466 4644 4830 5550 6082 6653 7516 8126 8611 9762
256 578 1323 2253 4786 4883 5502 6306 6570 7164 7989 8223
257 600 2164 2555 5253 6425 6940 7706 8106 8135 8338
258 5136 6712 6966 7754 8204 8359
259 5119 5246

```

Question #2

```

1 class Graph:
2     adjacency_list = {}
3     is_directed = False
4
5     def __init__(self, is_directed=False):
6         self.adjacency_list = {}
7         self.is_directed = is_directed
8
9     def add_edge(self, start, destination):
10        self.adjacency_list[start].add(destination)
11
12        if self.is_directed:
13            self.adjacency_list[destination].add(start)
14

```

```

15     def add_node(self, node):
16         if node not in self.adjacency_list:
17             self.adjacency_list[node] = set()
18
19     def print_graph(self):
20         max_row, max_column = self.max_dimensions
21         list_version = [[0 for i in range(max_column)] for j in range(
max_row)]
22
23         for source, destination_set in self.adjacency_list.items():
24             for destination in destination_set:
25                 list_version[source][destination] = 1
26
27         print(matrix(list_version))
28
29     @property
30     def max_dimensions(self):
31         max_element = max(self.adjacency_list.keys())
32
33         for _, value in self.adjacency_list.items():
34             value = list(value)
35
36             if value:
37                 if max(value) > max_element:
38                     max_element = max(value)
39
40         return max_element + 1, max_element + 1
41
42     def depth_first_search(self, start_node, preserve_order=True):
43         if preserve_order:
44             colors = {}
45             for vertex in self.adjacency_list:
46                 colors[vertex] = "white"
47
48             return self.__depth_first_search_preserve_order(start_node,
colors, None)
49         else:
50             return self.__depth_first_search(start_node, None)
51
52     def breadth_first_search(self, start_node, preserve_order=True):
53         if preserve_order:
54             return self.__breadth_first_search_preserve_order(start_node)
55         else:
56             return self.__breadth_first_search(start_node)
57
58     def __depth_first_search(self, start_node, visited_nodes):
59         if visited_nodes is None:
60             visited_nodes = set()
61
62         visited_nodes.add(start_node)
63         for unvisited_node in self.adjacency_list[start_node] -
visited_nodes:

```

```

64         self.__depth_first_search(unvisited_node, visited_nodes)
65
66     return visited_nodes
67
68     def __depth_first_search_preserve_order(self, start_node, colors,
visited_nodes):
69         if visited_nodes is None:
70             visited_nodes = []
71
72         if colors[start_node] is "white":
73             colors[start_node] = "grey"
74             visited_nodes += [start_node]
75
76         for unvisited_node in [x for x in self.adjacency_list[start_node]
if x not in visited_nodes]:
77             if colors[unvisited_node] is "white":
78                 self.__depth_first_search_preserve_order(unvisited_node,
colors, visited_nodes)
79
80             colors[start_node] = "black"
81
82     return visited_nodes
83
84     def __breadth_first_search(self, start):
85         visited, queue = set(), [start]
86
87         while queue:
88             vertex = queue.pop(0)
89
90             if vertex not in visited:
91                 visited.add(vertex)
92                 queue.extend(self.adjacency_list[vertex] - visited)
93
94     return visited
95
96     def __breadth_first_search_preserve_order(self, start):
97         colors = {}
98
99         for vertex in self.adjacency_list:
100             colors[vertex] = "white"
101
102         queue = [start]
103         visited_nodes = [start]
104
105         while queue:
106             node = queue.pop(0)
107
108             for unvisited_node in list(self.adjacency_list[node]):
109                 if colors[unvisited_node] is "white":
110                     colors[unvisited_node] = "grey"
111                     queue.insert(0, unvisited_node)
112                     visited_nodes += [unvisited_node]

```



```

113
114         colors[node] = "black"
115
116     return visited_nodes

```

We test examples with the following adjacency list.

$$\begin{bmatrix}
 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0
 \end{bmatrix}$$

Problem #2.1

For Depth-First Search, we get the following ordering (with the root taken at random).

$3 \rightarrow 4 \rightarrow 2 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 6 \rightarrow 5 \rightarrow 7$
 $0 \rightarrow 9 \rightarrow 1 \rightarrow 2 \rightarrow 8 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 7$
 $8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 7$
 $5 \rightarrow 4 \rightarrow 2 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7$
 $0 \rightarrow 9 \rightarrow 1 \rightarrow 2 \rightarrow 8 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 7$
 $6 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 2 \rightarrow 8 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7$
 $3 \rightarrow 4 \rightarrow 2 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 6 \rightarrow 5 \rightarrow 7$
 $5 \rightarrow 4 \rightarrow 2 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7$
 $7 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5$
 $9 \rightarrow 1 \rightarrow 2 \rightarrow 8 \rightarrow 0 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 7$
 $3 \rightarrow 4 \rightarrow 2 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 6 \rightarrow 5 \rightarrow 7$
 $8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 7$
 $5 \rightarrow 4 \rightarrow 2 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7$
 $0 \rightarrow 9 \rightarrow 1 \rightarrow 2 \rightarrow 8 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 7$
 $2 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 5 \rightarrow 7$

Problem #2.2

For Breadth-First Search, we get the following ordering (with the root taken at random).

3 → 4 → 6 → 0 → 8 → 5 → 7 → 9 → 1 → 2
 0 → 9 → 3 → 4 → 1 → 2 → 8 → 5 → 6 → 7
 8 → 0 → 3 → 5 → 6 → 7 → 9 → 1 → 4 → 2
 5 → 4 → 2 → 6 → 0 → 8 → 3 → 7 → 9 → 1
 0 → 9 → 3 → 4 → 1 → 2 → 8 → 5 → 6 → 7
 6 → 0 → 8 → 3 → 5 → 7 → 9 → 1 → 4 → 2
 3 → 4 → 6 → 0 → 8 → 5 → 7 → 9 → 1 → 2
 5 → 4 → 2 → 6 → 0 → 8 → 3 → 7 → 9 → 1
 7 → 8 → 0 → 3 → 5 → 6 → 9 → 1 → 4 → 2
 9 → 1 → 4 → 6 → 0 → 8 → 3 → 5 → 7 → 2
 3 → 4 → 6 → 0 → 8 → 5 → 7 → 9 → 1 → 2
 8 → 0 → 3 → 5 → 6 → 7 → 9 → 1 → 4 → 2
 5 → 4 → 2 → 6 → 0 → 8 → 3 → 7 → 9 → 1
 0 → 9 → 3 → 4 → 1 → 2 → 8 → 5 → 6 → 7
 2 → 8 → 0 → 3 → 5 → 6 → 7 → 9 → 1 → 4

Question #3

Our graph is exactly the same as the previous version; however, this one has the boolean property (`is_directed` set to false). We test examples with the following adjacency list.

$$\begin{bmatrix}
 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0
 \end{bmatrix}$$

Problem #3.1

For Depth-First Search, we get the following ordering (with the root taken at random).

$3 \rightarrow 9 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 6$
 $9 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 6$
 $2 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 9 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$
 $4 \rightarrow 9 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 5$
 $9 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 6$
 $0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 9 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 6$
 $4 \rightarrow 9 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 5$
 $2 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 9 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7$
 $4 \rightarrow 9 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 5$
 $1 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 6$
 $0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 9 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 6$
 $7 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6$
 $7 \rightarrow 8 \rightarrow 0 \rightarrow 9 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6$
 $5 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 9 \rightarrow 4 \rightarrow 2 \rightarrow 7 \rightarrow 6$
 $4 \rightarrow 9 \rightarrow 0 \rightarrow 8 \rightarrow 1 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 2 \rightarrow 5$

Problem #3.2

For Breadth-First Search, we get the following ordering (with the root taken at random).

3 → 9 → 1 → 4 → 6 → 7 → 8 → 2 → 0 → 5
 9 → 0 → 3 → 4 → 2 → 5 → 1 → 8 → 7 → 6
 2 → 0 → 8 → 4 → 5 → 7 → 1 → 3 → 9 → 6
 4 → 9 → 2 → 3 → 5 → 1 → 8 → 7 → 0 → 6
 9 → 0 → 3 → 4 → 2 → 5 → 1 → 8 → 7 → 6
 0 → 8 → 9 → 2 → 5 → 4 → 3 → 1 → 7 → 6
 4 → 9 → 2 → 3 → 5 → 1 → 8 → 7 → 0 → 6
 2 → 0 → 8 → 4 → 5 → 7 → 1 → 3 → 9 → 6
 4 → 9 → 2 → 3 → 5 → 1 → 8 → 7 → 0 → 6
 1 → 8 → 3 → 4 → 7 → 2 → 0 → 5 → 9 → 6
 0 → 8 → 9 → 2 → 5 → 4 → 3 → 1 → 7 → 6
 7 → 8 → 1 → 2 → 3 → 9 → 4 → 6 → 5 → 0
 7 → 8 → 1 → 2 → 3 → 9 → 4 → 6 → 5 → 0
 5 → 0 → 2 → 4 → 9 → 3 → 1 → 8 → 7 → 6
 4 → 9 → 2 → 3 → 5 → 1 → 8 → 7 → 0 → 6

Question #4

```

1 class ParentheticalOrder(Graph):
2     def print_parenthetical_order(self, start_node):
3         colors = {}
4         for vertex in self.adjacency_list:
5             colors[vertex] = "white"
6
7         for node in sorted(self.adjacency_list):
8             if colors[node] is "white":
9                 self.__print_parenthetical_order(node, colors)
10
11     def __print_parenthetical_order(self, node, colors):
12         colors[node] = "grey"
13         print("({} ".format(node), end='')
14         for neighbor in sorted(self.adjacency_list[node]):
15             if colors[neighbor] is "white":
16                 self.__print_parenthetical_order(neighbor, colors)
17         colors[node] = 'black'
18         print(" {})".format(node), end='')

```

We get the following string:

(u (v (y (x x) y) v) u) (w (z z) w)

Question #5

Question #6

```

1 class WrestleMania(Graph):
2     def determine_valid_rivalry(self):
3         rivalry, not_visited = {}, list(self.adjacency_list.keys())
4
5         for vertex in self.adjacency_list:
6             rivalry[vertex] = "none"
7
8         while "none" in rivalry.values():
9             current_depth, start = 0, not_visited[-1]
10            queue = [start]
11
12            while queue:
13                current_depth += 1
14                node = queue.pop(0)
15
16                for unvisited_node in list(self.adjacency_list[node]):
17                    if rivalry[unvisited_node] is "none":
18                        if current_depth % 2 == 0:
19                            rivalry[unvisited_node] = "good guy"
20                        else:
21                            rivalry[unvisited_node] = "bad guy"
22
23                not_visited.remove(unvisited_node)
24                queue.insert(0, unvisited_node)
25
26            for wrestler, adjacency_wrestlers in self.adjacency_list.items():
27                for adjacent_wrestler in adjacency_wrestlers:
28                    if rivalry[wrestler] == rivalry[adjacent_wrestler]:
29                        return False
30
31            return True

```

For the following sample inputs

$$X_1 = \begin{bmatrix} & x & y & z \\ x & 1 & 0 & 1 \\ y & 0 & 0 & 1 \\ z & 1 & 1 & 0 \end{bmatrix} \quad X_2 = \begin{bmatrix} & w & x & y & z \\ w & 0 & 1 & 0 & 1 \\ x & 1 & 0 & 1 & 0 \\ y & 0 & 1 & 0 & 1 \\ z & 1 & 0 & 1 & 0 \end{bmatrix} \quad X_3 = \begin{bmatrix} & u & v & w & x & y & z \\ u & 0 & 1 & 0 & 0 & 0 & 0 \\ v & 1 & 0 & 0 & 0 & 0 & 0 \\ w & 0 & 0 & 0 & 1 & 0 & 0 \\ x & 0 & 0 & 1 & 0 & 0 & 0 \\ y & 0 & 0 & 0 & 0 & 0 & 1 \\ z & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

```

1 No configuration could be found.
2 =====
3 Wrestler y Is A Baby Face

```

```

4 Wrestler x Is A Heel
5 Wrestler z Is A Heel
6 Wrestler w Is A Baby Face
7 =====
8 Wrestler u Is A Heel
9 Wrestler w Is A Heel
10 Wrestler v Is A Baby Face
11 Wrestler y Is A Heel
12 Wrestler x Is A Baby Face
13 Wrestler z Is A Baby Face

```

Question #7

Problem #7.1

For this problem we have three cases.

No Children If the root has no children it has no edges so it cannot disconnect the graph.

One Child The only edge it breaks is paths to itself so it cannot disconnect graph.

≥ 2 *Children* Because there are no cross edges between the children, and because the children nodes are in the subtrees of the root, no path exists between them. Deleting the root will delete the link between the two or more children; thus, disconnecting the graph and making the root G_π an articulation point.

Problem #7.2

Suppose there is a vertex $v \in G_\pi$, where v is a non-root. Furthermore, suppose v has a child s such that there is no back edge from or any descendent of s to ancestor of v . Thus, the removal of v will lead the disconnection of the sub-tree rooted at v from the graph G_π .

Therefore, the absence of an edge between the descendants of s and ancestors of v resulted in the disconnection of the graph after removal of non-root vertex v . This implies v is an articulation point.

Problem #7.3

Because v is discovered before all of its descendants, the only edges that could affect the minimum are ancestors of v . Thus, we can do an augmented depth-first search to determine all of the low values.

```

1 class ArticulateGraph(Graph):
2     time = 0
3     times, colors, lows = {}, {}, {}
4
5     def find_minimum(self, start_node, visited_nodes=None):

```

```

6         if visited_nodes is None:
7             self.times, self.colors, self.lowses = {}, {}, {}
8             self.time = 0
9
10        for vertex in self.adjacency_list:
11            self.colors[vertex] = "white"
12
13        visited_nodes = []
14
15        for unvisited_node in [x for x in self.adjacency_list[start_node]
16 if x not in visited_nodes]:
17            if self.colors[unvisited_node] is "white":
18                self.colors[start_node] = "grey"
19                self.time += 1
20                visited_nodes += [start_node]
21
22                self.times[start_node] = self.time
23                self.lows[start_node] = self.times[start_node]
24
25                for adjacent_node in self.adjacency_list[start_node]:
26                    self.find_minimum(adjacent_node, visited_nodes)
27
28                    if self.colors[adjacent_node] is "white":
29                        if self.time[start_node] < self.lows[self.
adjacency_list]:
30                            self.lows[start_node] = self.times[
adjacent_node]
31
32                self.colors[start_node] = "black"
33                self.time += 1
34
35        return self.lows

```

With the following graph,

$$\begin{bmatrix}
 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
 \end{bmatrix}$$

We get the following:

$8 \rightarrow 3$

$2 \rightarrow 2$

$3 \rightarrow 1$

$4 \rightarrow 4$

$7 \rightarrow 9$

Problem #7.4

After applying the algorithm mentioned above for all $v \in V$, we check to see if $v.low = v.d$. If it is, no descendants of v has a back edge to a proper ancestor of v , implying v is not a articulation point.

Problem #7.5

If there is a cycle from $u \rightarrow v \rightarrow w$, where u and v have unique partitions, then removing any edge of w does not disconnect the graph. Therefore, the edges from w to u and v cannot be bridges.

Problem #7.6

A simple circuit contains an edge (u, v) if and only if

- Both of its endpoints are articulation points.
- One of its endpoints is an articulation point and the other is a vertex of degree 1.

Using our previous algorithm above, we can compute this by running `find_minimum(self, v)`, and deciding if one of the two criteria are met.

Problem #7.7

Because we have already stated that any two edges of any bi-connected components lie on a common cycle, we know all edges are lying inside a components of a graph will not be bridges (via Part E). Therefore, all bi-connected components of a graph partition the non-bridges of said graph.

Problem #7.8

Locate all bridge edges using the algorithm described in Part F. Remove each bridge, the connected components are all biconnected components.