Homework #7

Analysis of Algorithms

Illya Starikov

Due Date: October 20$^{\text{th}}$ 2017

```python
class Node:
    value = -1
    left_child, right_right = None, None

    def __init__(self, value):
        self.value = value
        self.left_child, self.right_child = None, None

    @property
    def number_of_children(self):
        if self.left_child is not None and self.right_child is not None:
            return 2
        elif self.left_child is not None or self.right_child is not None:
            return 1
        else:
            return 0

    @property
    def valid_children(self):
        number_of_children = self.number_of_children

        if number_of_children == 0:
            return None
        elif number_of_children == 1:
            if self.left_child is not None:
                return self.left_child
            else:
                return self.right_child
        else:
            return (self.left_child, self.right_child)

    def __str__(self):
        return str(self.value)


class BinarySearchTree:
```

```
    root = -1

    def __init__(self):
        self.root = None

    def insert(self, value):
        if self.root is None:
            self.root = Node(value)
        else:
            self.__insert_node(self.root, value)

    def exists(self, value):
        return False if self.__find_node(self.root, value) is None else True

    def delete(self, value):
        self.root = self.__delete_node(self.root, value)

    def print_tree(self):
        if not self.root:
            return

        current_level = [self.root]

        while current_level:

            print(' '.join(str(node) for node in current_level))

            next_level = list()
            for n in current_level:
                if n.left_child:
                    next_level.append(n.left_child)
                if n.right_child:
                    next_level.append(n.right_child)
                current_level = next_level

    # MARK: Private Methods
    def __insert_node(self, current_node, value):
        # for the values <= `value', we put on the left side of the tree
        if value <= current_node.value:
            if current_node.left_child is None:
                current_node.left_child = Node(value)
            else:
                self.__insert_node(current_node.left_child, value)

        # for values > `value', we put on the right side of the tree
        else:
            if current_node.right_child is None:
                current_node.right_child = Node(value)
            else:
                self.__insert_node(current_node.right_child, value)

```

```python
    def __find_node(self, current_node, value):
        if current_node is None:
            return None
        elif value < current_node.value:
            return self.__find_node(current_node.left_child, value)
        elif value > current_node.value:
            return self.__find_node(current_node.right_child, value)
        else:
            return current_node

    def __delete_node(self, current_node, key):
        if not current_node:
            return current_node

        if current_node.value > key:
            current_node.left_child = self.__delete_node(current_node.left_child, key)
        elif current_node.value < key:
            current_node.right_child = self.__delete_node(current_node.right_child, key)
        else:
            if not current_node.left_child:
                right_child = current_node.right_child
                del current_node
                return right_child

            elif not current_node.right_child:
                left_child = current_node.left_child
                del current_node
                return left_child

            else:
                successor = current_node.right_child
                while successor.left_child:
                    successor = successor.left_child

                current_node.value = successor.value
                current_node.right_child = self.__delete_node(current_node.right_child, successor.value)

        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.

```
Element  3739 Found: True
Element  5550 Found: True
Element   600 Found: True
Element  3262 Found: True
Element  8611 Found: True
Element  4750 Found: True
Element  5246 Found: True
Element  6712 Found: True
Element  1323 Found: True
Element  1831 Found: True
Element  5520 Found: True
Element  3063 Found: True
Element  9300 Found: True
Element  1004 Found: True
Element  7164 Found: True
Element  4466 Found: True
Element  8890 Found: True
```

```
Element  8338 Found: True
Element  1853 Found: True
Element  8223 Found: True
Element  3159 Found: True
Element  4644 Found: True
Element  7706 Found: True
Element  8740 Found: True
Element  1120 Found: True
Element  2040 Found: True
Element  6653 Found: True
Element   578 Found: True
Element  9762 Found: True
Element  8565 Found: True
Element  7604 Found: True
Element  9985 Found: True
Element  4961 Found: True
Element   929 Found: True
Element  2082 Found: True
Element  5253 Found: True
Element  3405 Found: True
Element  7516 Found: True
Element  6306 Found: True
Element  8106 Found: True
Element  1347 Found: True
Element  4310 Found: True
Element  4584 Found: True
Element  2954 Found: True
Element  9614 Found: True
Element  2253 Found: True
Element  1522 Found: True
Element  8135 Found: True
Element  4475 Found: True
Element  1512 Found: True
Element  9304 Found: True
Element  6966 Found: True
Element   321 Found: True
Element  2974 Found: True
Element  6940 Found: True
Element  7754 Found: True
Element  1213 Found: True
Element  1149 Found: True
Element  8961 Found: True
Element  8359 Found: True
Element  4883 Found: True
Element  7499 Found: True
Element  4786 Found: True
Element  8126 Found: True
Element    42 Found: True
Element  4699 Found: True
Element  3902 Found: True
Element  1730 Found: True
Element  4494 Found: True
```

```
Element  5502 Found: True
Element   104 Found: True
Element  4830 Found: True
Element  6082 Found: True
Element  1899 Found: True
Element  5136 Found: True
Element  5644 Found: True
Element  2849 Found: True
Element  5675 Found: True
Element  8204 Found: True
Element  7632 Found: True
Element  1886 Found: True
Element  5119 Found: True
Element  5016 Found: True
Element  6562 Found: True
Element  3397 Found: True
Element  9971 Found: True
Element  3694 Found: True
Element  4783 Found: True
Element  2718 Found: True
Element   734 Found: True
Element  7989 Found: True
Element  7324 Found: True
Element  1128 Found: True
Element  6425 Found: True
Element  4769 Found: True
Element  2555 Found: True
Element  3476 Found: True
Element  9509 Found: True
Element  2164 Found: True
Element  6570 Found: True
```

Problem #1.2

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

```
Element  9170 Found: False
Element  1059 Found: False
Element  6807 Found: False
Element  7990 Found: False
Element  9177 Found: False
Element  9933 Found: False
Element  9290 Found: False
Element  5479 Found: False
Element   611 Found: False
Element  5458 Found: False
Element  1563 Found: False
Element  5870 Found: False
Element  3134 Found: False
Element   178 Found: False
Element  3890 Found: False
```

```
Element  7225 Found: False
Element  9820 Found: False
Element  2292 Found: False
Element  5313 Found: False
Element  4489 Found: False
Element  3835 Found: False
Element  2543 Found: False
Element  5679 Found: False
Element    46 Found: False
Element  1812 Found: False
Element  4628 Found: False
Element  7310 Found: False
Element  6839 Found: False
Element  5411 Found: False
Element  2929 Found: False
Element  2351 Found: False
Element  9107 Found: False
Element  7498 Found: False
Element  5810 Found: False
Element  6692 Found: False
Element  3986 Found: False
Element  2941 Found: False
Element  9506 Found: False
Element  3485 Found: False
Element  1113 Found: False
Element  4268 Found: False
Element  9848 Found: False
Element  1525 Found: False
Element  7304 Found: False
Element  1792 Found: False
Element  2707 Found: False
Element  4111 Found: False
Element  7696 Found: False
Element  4897 Found: False
Element   120 Found: False
Element  9588 Found: False
Element   819 Found: False
Element  5893 Found: False
Element  9990 Found: False
Element  6444 Found: False
Element  2619 Found: False
Element  3091 Found: False
Element  8431 Found: False
Element  7965 Found: False
Element  4863 Found: False
Element  9778 Found: False
Element  9601 Found: False
Element  8084 Found: False
Element  7459 Found: False
Element  7299 Found: False
Element     1 Found: False
Element  6819 Found: False
```

```
Element  9085 Found: False
Element  5574 Found: False
Element  8748 Found: False
Element  8790 Found: False
Element  6042 Found: False
Element  5852 Found: False
Element  5341 Found: False
Element  7671 Found: False
Element  8456 Found: False
Element  1818 Found: False
Element  6918 Found: False
Element  9723 Found: False
Element  3980 Found: False
Element  6496 Found: False
Element  5121 Found: False
Element  2553 Found: False
Element  7316 Found: False
Element  1461 Found: False
Element  3896 Found: False
Element  9382 Found: False
Element  8610 Found: False
Element  5181 Found: False
Element  3305 Found: False
Element   549 Found: False
Element  7306 Found: False
Element  4612 Found: False
Element  9525 Found: False
Element  4847 Found: False
Element  6420 Found: False
Element  2643 Found: False
Element  4538 Found: False
Element  5934 Found: False
Element  7969 Found: False
```

Problem #1.3

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

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

```

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

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

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

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

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

Current Tree After Deleting Element 2082
3739
2974 7632
1149 3405 4783 9509
734 1899 3063 3694 4475 6562 8890 9985
42 1120 1512 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
104 1004 1128 1347 1522 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604
    8565 8961 9614
321 929 1213 1886 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300
    9762
```

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

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

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

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

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

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

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

Current Tree Before Deleting Element 2849
3739
2974 7632
1149 3405 4783 9509
734 1899 3063 3694 4475 6562 8890 9985
```

```
42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
104 1004 1128 1347 1886 2849 3159 3397 4310 4494 4750 5016 5675 7324 7604 8565 8961 9614
321 929 1213 1831 2040 4466 4644 4830 5520 6082 6653 7516 8126 8611 9300 9762
578 1323 2718 4786 4883 5502 5550 6306 6570 7164 7989 8223
600 2253 5253 6425 6940 7706 8106 8135 8338
2164 2555 5136 6712 6966 7754 8204 8359
5119 5246

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

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

Current Tree After Deleting Element 5520
3739
2974 7632
1149 3405 4783 9509
734 1899 3063 3694 4475 6562 8890 9985
42 1120 1522 2954 3262 3476 3902 4584 5644 7499 8740 9304 9971
104 1004 1128 1347 1886 2040 3159 3397 4310 4494 4750 5016 5675 7324 7604 8565 8961 9614
321 929 1213 1831 2718 4466 4644 4830 5550 6082 6653 7516 8126 8611 9300 9762
578 1323 2253 4786 4883 5502 6306 6570 7164 7989 8223
```

```
600 2164 2555 5253 6425 6940 7706 8106 8135 8338
5136 6712 6966 7754 8204 8359
5119 5246
--------------------------------------------------------------------------------

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

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

Question #2

```python
class Graph:
    adjacency_list = {}
    is_directed = False

    def __init__(self, is_directed=False):
        self.adjacency_list = {}
        self.is_directed = is_directed

    def add_edge(self, start, destination):
        self.adjacency_list[start].add(destination)

        if self.is_directed:
            self.adjacency_list[destination].add(start)

```

```python
    def add_node(self, node):
        if node not in self.adjacency_list:
            self.adjacency_list[node] = set()

    def print_graph(self):
        max_row, max_column = self.max_dimensions
        list_version = [[0 for i in range(max_column)] for j in range(max_row)]

        for source, destination_set in self.adjacency_list.items():
            for destination in destination_set:
                list_version[source][destination] = 1

        print(matrix(list_version))

    @property
    def max_dimensions(self):
        max_element = max(self.adjacency_list.keys())

        for _, value in self.adjacency_list.items():
            value = list(value)

            if value:
                if max(value) > max_element:
                    max_element = max(value)

        return max_element + 1, max_element + 1

    def depth_first_search(self, start_node, preserve_order=True):
        if preserve_order:
            colors = {}
            for vertex in self.adjacency_list:
                colors[vertex] = "white"

            return self.__depth_first_search_preserve_order(start_node, colors, None)
        else:
            return self.__depth_first_search(start_node, None)

    def breadth_first_search(self, start_node, preserve_order=True):
        if preserve_order:
            return self.__breadth_first_search_preserve_order(start_node)
        else:
            return self.__breadth_first_search(start_node)

    def __depth_first_search(self, start_node, visited_nodes):
        if visited_nodes is None:
            visited_nodes = set()

        visited_nodes.add(start_node)
        for unvisited_node in self.adjacency_list[start_node] - visited_nodes:
```

```
            self.__depth_first_search(unvisited_node, visited_nodes)

        return visited_nodes

    def __depth_first_search_preserve_order(self, start_node, colors, visited_nodes):
        if visited_nodes is None:
            visited_nodes = []

        if colors[start_node] is "white":
            colors[start_node] = "grey"
            visited_nodes += [start_node]

        for unvisited_node in [x for x in self.adjacency_list[start_node] if x not in visited_nodes]:
            if colors[unvisited_node] is "white":
                self.__depth_first_search_preserve_order(unvisited_node, colors, visited_nodes)

        colors[start_node] = "black"

        return visited_nodes

    def __breadth_first_search(self, start):
        visited, queue = set(), [start]

        while queue:
            vertex = queue.pop(0)

            if vertex not in visited:
                visited.add(vertex)
                queue.extend(self.adjacency_list[vertex] - visited)

        return visited

    def __breadth_first_search_preserve_order(self, start):
        colors = {}

        for vertex in self.adjacency_list:
            colors[vertex] = "white"

        queue = [start]
        visited_nodes = [start]

        while queue:
            node = queue.pop(0)

            for unvisited_node in list(self.adjacency_list[node]):
                if colors[unvisited_node] is "white":
                    colors[unvisited_node] = "grey"
                    queue.insert(0, unvisited_node)
                    visited_nodes += [unvisited_node]
```

```python

            colors[node] = "black"

        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).

$$
\begin{gathered}
3 \to 4 \to 2 \to 8 \to 0 \to 9 \to 1 \to 6 \to 5 \to 7 \\
0 \to 9 \to 1 \to 2 \to 8 \to 3 \to 4 \to 6 \to 5 \to 7 \\
8 \to 0 \to 9 \to 1 \to 2 \to 3 \to 4 \to 6 \to 5 \to 7 \\
5 \to 4 \to 2 \to 8 \to 0 \to 9 \to 1 \to 3 \to 6 \to 7 \\
0 \to 9 \to 1 \to 2 \to 8 \to 3 \to 4 \to 6 \to 5 \to 7 \\
6 \to 0 \to 9 \to 1 \to 2 \to 8 \to 3 \to 4 \to 5 \to 7 \\
3 \to 4 \to 2 \to 8 \to 0 \to 9 \to 1 \to 6 \to 5 \to 7 \\
5 \to 4 \to 2 \to 8 \to 0 \to 9 \to 1 \to 3 \to 6 \to 7 \\
7 \to 8 \to 0 \to 9 \to 1 \to 2 \to 3 \to 4 \to 6 \to 5 \\
9 \to 1 \to 2 \to 8 \to 0 \to 3 \to 4 \to 6 \to 5 \to 7 \\
3 \to 4 \to 2 \to 8 \to 0 \to 9 \to 1 \to 6 \to 5 \to 7 \\
8 \to 0 \to 9 \to 1 \to 2 \to 3 \to 4 \to 6 \to 5 \to 7 \\
5 \to 4 \to 2 \to 8 \to 0 \to 9 \to 1 \to 3 \to 6 \to 7 \\
0 \to 9 \to 1 \to 2 \to 8 \to 3 \to 4 \to 6 \to 5 \to 7 \\
2 \to 8 \to 0 \to 9 \to 1 \to 3 \to 4 \to 6 \to 5 \to 7
\end{gathered}
$$

Problem #2.2

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

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

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).

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

Problem #3.2

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Question #4

```python
class ParentheticalOrder(Graph):
    def print_parenthetical_order(self, start_node):
        colors = {}
        for vertex in self.adjacency_list:
            colors[vertex] = "white"

        for node in sorted(self.adjacency_list):
            if colors[node] is "white":
                self.__print_parenthetical_order(node, colors)

    def __print_parenthetical_order(self, node, colors):
        colors[node] = "grey"
        print("({} ".format(node), end='')
        for neighbor in sorted(self.adjacency_list[node]):
            if colors[neighbor] is "white":
                self.__print_parenthetical_order(neighbor, colors)
        colors[node] = 'black'
        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

```python
class WrestleMania(Graph):
    def determine_valid_rivalry(self):
        rivalry, not_visited = {}, list(self.adjacency_list.keys())

        for vertex in self.adjacency_list:
            rivalry[vertex] = "none"

        while "none" in rivalry.values():
            current_depth, start = 0, not_visited[-1]
            queue = [start]

            while queue:
                current_depth += 1
                node = queue.pop(0)

                for unvisited_node in list(self.adjacency_list[node]):
                    if rivalry[unvisited_node] is "none":
                        if current_depth % 2 == 0:
                            rivalry[unvisited_node] = "good guy"
                        else:
                            rivalry[unvisited_node] = "bad guy"

                        not_visited.remove(unvisited_node)
                        queue.insert(0, unvisited_node)

        for wrestler, adjacency_wrestlers in self.adjacency_list.items():
            for adjacent_wrestler in adjacency_wrestlers:
                if rivalry[wrestler] == rivalry[adjacent_wrestler]:
                    return False

        return True
```

For the following sample inputs

$$X_1 = \begin{array}{cc} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} x \\ y \\ z \end{array} & \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \end{array} \qquad X_2 = \begin{array}{cc} & \begin{array}{cccc} w & x & y & z \end{array} \\ \begin{array}{c} w \\ x \\ y \\ z \end{array} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad X_3 = \begin{array}{cc} & \begin{array}{cccccc} u & v & w & x & y & z \end{array} \\ \begin{array}{c} u \\ v \\ w \\ x \\ y \\ z \end{array} & \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

```
No configuration could be found.
========================================
Wrestler y Is A Baby Face
```

```
Wrestler x Is A Heel
Wrestler z Is A Heel
Wrestler w Is A Baby Face
========================================
Wrestler u Is A Heel
Wrestler w Is A Heel
Wrestler v Is A Baby Face
Wrestler y Is A Heel
Wrestler x Is A Baby Face
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.

$\geq$ *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.

```python
class ArticulateGraph(Graph):
    time = 0
    times, colors, lows = {}, {}, {}

    def find_minimum(self, start_node, visited_nodes=None):
```

```python
        if visited_nodes is None:
            self.times, self.colors, self.lowes = {}, {}, {}
            self.time = 0

            for vertex in self.adjacency_list:
                self.colors[vertex] = "white"

            visited_nodes = []

        for unvisited_node in [x for x in self.adjacency_list[start_node] if x not in visited_nodes]:
            if self.colors[unvisited_node] is "white":
                self.colors[start_node] = "grey"
                self.time += 1
                visited_nodes += [start_node]

                self.times[start_node] = self.time
                self.lows[start_node] = self.times[start_node]

                for adjacent_node in self.adjacency_list[start_node]:
                    self.find_minimum(adjacent_node, visited_nodes)

                    if self.colors[adjacent_node] is "white":
                        if self.time[start_node] < self.lows[self.adjacency_list]:
                            self.lows[start_node] = self.times[adjacent_node]

        self.colors[start_node] = "black"
        self.time += 1

        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:

$$8 \to 3$$
$$2 \to 2$$
$$3 \to 1$$
$$4 \to 4$$
$$7 \to 9$$

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 $u \to v \to w$, 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 ifs 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 nay two edges of any bi-connected components lie on a command 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.