Lab 1 Report Template

1. Given elements 1, 2, 3, 4, 5, 6, 7. Implement BST algorithms.

(1) Every node in the BST has two child nodes, which are called left-node and right-node. All of the nodes in the BST could point to None value.

```
class BST_node():

    def __init__(self, value=None):
        self.value = value
        self.left = None
        self.right = None

    def get_value(self):
        if self != None:
            return self.value
```

(2) In the BST tree, only the function ***insert*** is used to add a new node into the tree. While the head node does not has value, the value will be added to the head node of the tree. In the other conditions, the value will search for a node whose child node that suitable for the value is point to a None value.

```
class BST_tree ():

    def __init__(self, value=None):
        self.head = BST_node (value)

    def is_empty(self):
        if not self.head.value:
            return True
        else:
            return False

    def insert(self, value):
        if self.is_empty ():
            self.head.value = value
        else:
            current_node = self.head
            while True:
                if current_node.value < value:
                    if current_node.right == None:
                        current_node.right = BST_node (value)
                        return
                    else:
                        current_node = current_node.right

                else:
                    if current_node.left == None:
                        current_node.left = BST_node (value)
                        return
                    else:
                        current_node = current_node.left
```

2. Insert them into a binary search tree in the orders:

4, 2, 1, 3, 6, 5, 7 and 1, 2, 3, 4, 5, 6, 7

(1) Test code:

```
BBB = BST_tree()
bbb = BST_tree()
for i in range(8):
    BBB.insert(value=i)
for p in [4, 2, 1, 3, 6, 5, 7]:
    bbb.insert(value=p)
bbb.show()
print()
BBB.show()
```

(2) I use a method called ***show*** to display the tree.

```
def show(self):
    current_level = [self.head]
    next_level = []
    while current_level != []:
        for node in current_level:
            if node != None:
                if node.left:
                    next_level.append(node.left)
                else:
                    next_level.append(None)
                if node.right:
                    next_level.append(node.right)
                else:
                    next_level.append(None)
        print(list(map(BST_node.get_value, current_level)))
        current_level = next_level
        next_level = []
```

(3) Output of the test:

```
[4]
[2, 6]
[1, 3, 5, 7]
[None, None, None, None, None, None, None, None]

[1]
[None, 2]
[None, 3]
[None, 4]
[None, 5]
[None, 6]
[None, 7]
[None, None]
```

3. Delete element 2, and then delete element 6

(1) The ***delete*** method:

If we want to delete a node, we also have to find the father node of it, since while the node does not have 2 child node, we need to do some changes in its father node. If there is no child, the father just point to None. Also, if there is only one child just let the father point to the child. While there are two children, just delete the node containing the minimum of the right-subtree and use the minimum to replace the value of the node.

```
def delete(self, value):
    father_node = None
    current_node = self.head
    flag = None
    while True:
        if current_node.value < value:
            if current_node.right == None:
                return
            else:
                father_node = current_node
                flag = "right"
                current_node = current_node.right

        elif current_node.value > value:
            if current_node.left == None:
                return
            else:
                father_node = current_node
                flag = "left"
                current_node = current_node.left

        elif current_node.value == value:
            break
```

```
if current_node.right == None and current_node.left == None:
    if flag == "right":
        father_node.right = None
    else:
        father_node.left = None

elif current_node.right != None and current_node.left == None:
    if flag == "right":
        father_node.right = current_node.right
    else:
        father_node.left = current_node.right

elif current_node.right == None and current_node.left != None:
    if flag == "right":
        father_node.right = current_node.left
    else:
        father_node.left = current_node.left
```

```
elif current_node.right != None and current_node.left != None:
    mark = "right"
    point = current_node.right
    fa_point = current_node
    while point.left != None:
        fa_point = point
        point = point.left
    current_node.value = point.value
    if mark == "left":
        if point.right == None:
            fa_point.left = None
        else:
            fa_point.left = point.right
    else:
        if point.right == None:
            fa_point.right = None
        else:
            fa_point.right = point.right
```

(2) The Test code:

```
BBB = BST_tree()
bbb = BST_tree()
for i in range(8):
    BBB.insert(value=i)
for p in [4, 2, 1, 3, 6, 5, 7]:
    bbb.insert(value=p)
bbb.show()
print()
bbb.delete(2)
bbb.show()
print()
bbb.delete(6)
bbb.show()
print()
BBB.show()
print()
BBB.delete(2)
BBB.show()
print()
BBB.delete(6)
BBB.show()
```

(3) Output:

```
[4]
[2, 6]
[1, 3, 5, 7]
[None, None, None, None, None, None, None, None]

[4]
[3, 6]
[1, None, 5, 7]
[None, None, None, None, None, None]

[4]
[3, 7]
[1, None, 5, None]
[None, None, None, None]
```

```
[1]
[None, 2]
[None, 3]
[None, 4]
[None, 5]
[None, 6]
[None, 7]
[None, None]

[1]
[None, 3]
[None, 4]
[None, 5]
[None, 6]
[None, 7]
[None, None]

[1]
[None, 3]
[None, 4]
[None, 5]
[None, 7]
[None, None]
```