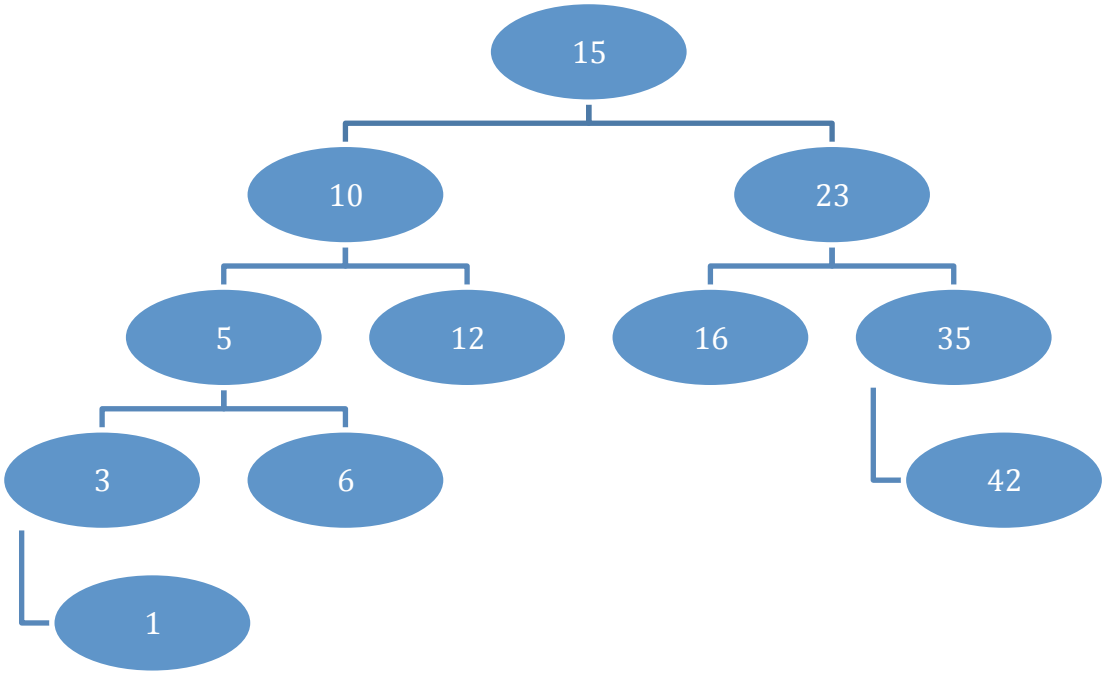
CS532 Homework 2
Archana Machireddy

Question 1

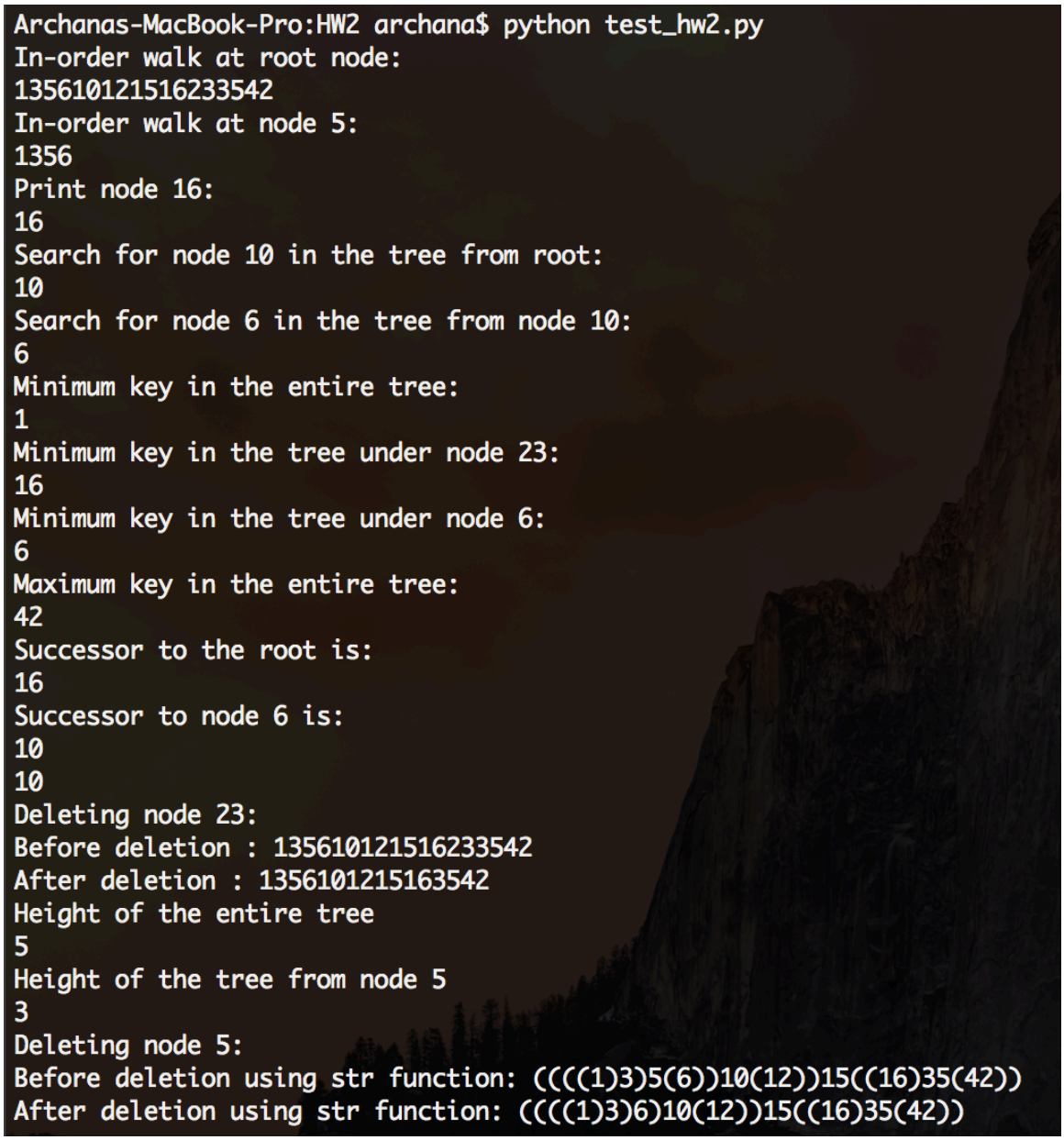
The tree I created is:



1 is the left child of 3, and 42 is the right child of 35.

```
t = hw2.Tree()
t.root = hw2.Node(15)
t.Insert(hw2.Node(10))
t.Insert(hw2.Node(5))
t.Insert(hw2.Node(6))
t.Insert(hw2.Node(12))
t.Insert(hw2.Node(3))
t.Insert(hw2.Node(1))
t.Insert(hw2.Node(23))
t.Insert(hw2.Node(35))
t.Insert(hw2.Node(42))
t.Insert(hw2.Node(16))

print('In-order walk at root node:')
print(t.root.InOrderWalk())

print('In-order walk at node 5:')
print(t.root.left.left.InOrderWalk())
```

```
print('Print node 16:')
print(t.root.right.left.key)

print('Search for node 10 in the tree from root:')
print(t.root.Search(10).key)
print('Search for node 6 in the tree from node 10:')
print(t.root.left.Search(6).key)

print('Minimum key in the entire tree:')
print(t.root.Min().key)
print('Minimum key in the tree under node 23:')
print(t.root.right.Min().key)
print('Minimum key in the tree under node 6:')
print(t.root.left.left.right.Min().key)

print('Maximum key in the entire tree:')
print(t.root.Max().key)
print('Successor to the root is:')
print(t.root.Succ().key)
print('Successor to node 6 is:')
print(t.root.left.left.right.Succ().key)
print(t.root.Search(6).Succ().key)

print('Deleting node 23:')
print('Before deletion : %s' %(t.root.InOrderWalk()))
t.Delete(t.root.Search(23))
print('After deletion : %s' %(t.root.InOrderWalk()))

print('Height of the entire tree')
print(t.root.Height(t.root.Search(15)))
print('Height of the tree from node 5')
print(t.root.Height(t.root.Search(5)))

print('Deleting node 5:')
print('Before deletion using str function: %s' %(str(t.root)))
t.Delete(t.root.Search(5))
print('After deletion using str function: %s' %(str(t.root)))
```



The delete function provided does not work when the node to the right of the node to be deleted is a leaf node. For example the given code does not work to delete node 5 or 10 in the above tree. Line 42 of the given code is '`y.right.parent = y`'. While deleting 5, y is 6, which make y.right as None. So accessing y.right.parent gives an error as there is no parent to None. For this reason I added in the check '`if y.right is not None:`'. When y is already None, its child's parents need not be set to None.

Modified delete code:

```
def Delete(self,z):
    if z.left is None:
        self.Transplant(z,z.right)
```

```
    elif z.right is None:
        self.Transplant(z,z.left)
    else:
        y = z.right.Min()
        if y.parent is not None:
            self.Transplant(y,y.right)
            y.right = z.right
            if y.right is not None:
                y.right.parent = y
        self.Transplant(z,y)
        y.left = z.left
        y.left.parent = y
```

Question 2

```
def __str__(self):
    s = ""
    if self.left is not None:
        s += '('
        s += str(self.left)
        s += ')'
    s += str(self.key)
    if self.right is not None:
        s += '('
        s += str(self.right)
        s += ')'
    return s
```

Question 3

```
def Insert_new(self,z):
    y = None
    x = self.root

    if x is None:
        self.root = z

    while x is not None:
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.parent = y
    if z.key < y.key:
```

```
        y.left = z
    else:
        y.right = z
```

Question 3 Critique

I forgot to add a return statement after inserting the new node at the root. I checked the logic just by going through the code, and checked it on a tree that already had nodes inserted. I should have checked it by inserting into an empty tree. I would have caught the error. The modified function with return statement is:

```
def Insert_new(self,z):
    y = None
    x = self.root

    if x is None:
        self.root = z
        return

    while x is not None:
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.parent = y
    if z.key < y.key:
        y.left = z
    else:
        y.right = z
```

Question 4

Inserting them in ascending order will result in a tree with all elements strictly on the rightmost branch as each inserted element will the greater than all the ones inserted so far. (Order : 1,2,3,4,5,6,7)

Inserting them in descending order will result in a tree with all elements strictly on the leftmost branch as each inserted element will the lesser than all the ones inserted so far. (Order : 7,6,5,4,3,2,1)

To get a strictly balanced tree, first the middle element needs to be inserted (4). The elements remaining on the left are 1,2 and 3. The middle element among these, i.e. 2, needs to be inserted next, followed by 1 and 3. Similarly on the right side, first 6 needs to

be inserted followed by 5 and 7. (Order : 4,2,6,3,5,1,7) This is because of the property of the binary search tree that the key at a given node is greater than all the keys to its left and smaller than all the keys to its right. Inserting in this order will give a strictly balanced tree of height 3 on both sides.

```
Archanas-MBP:HW2 archana$ python test_hw2.py
Inserting in ascending order: 1(2(3(4(5(6(7))))))
Inserting in descending order: ((((((1)2)3)4)5)6)7
Inserting in balanced order: ((1)2(3))4((5)6(7))
```

Question 4 Critique

I forgot to mention that insert only happens at leaves. I took that for granted and mentioned that while inserting in ascending order all elements get inserted to the right and vice versa.

Question 5

```
def Height(self,x):
    if x is None:
        return 0
    else:
        left_height = self.Height(x.left)
        right_height = self.Height(x.right)

    return 1 + max(left_height,right_height)
```

As we are visiting each node in the tree once, a tree with n nodes will have time complexity O(n). As we are going through all the nodes the lower bound is Ω(n) and upper bound is O(n), therefore it is Θ(n) too.

Question 5 Critique

I misunderstood the line in the question ‘To get the height of the tree, simply pass to it the root of the tree’. I thought that node from which we want to calculate the height of the tree, has to be passed into the function. That is why I was passing the node as an input. I felt that this was a bit roundabout while I was calling the function. But the answer in the key, calling it directly on the node, makes more sense.

Question 6

```
def BuildTree1023():
    numbers = random.sample(range(1,1024), 1023)
    t = hw2.Tree()
    for i in range(len(numbers)):
        t.Insert(hw2.Node(numbers[i]))
    return t.root.Height(t.root)

def BuildTrees():
    height = np.zeros(1000)
    for i in range(1000):
        height[i] = BuildTree1023()
    print('Average height of the tree is %f'%np.mean(height))
```

```
Archanas-MacBook-Pro:HW2 archana$ python test_hw2.py
Average height of the tree is 22.011000
```

The smallest height it can have is 10.
In a tree with $n=2^m-1$ nodes, the minimum height of the binary tree will be $h(n) = m = \log_2(n)$.

The average height differs from the optimal height by 10 units as the insertion of nodes is random, and will not give the perfect balanced tree.

Question 6 Critique

I compared it only with the optimal case. I should have compared it with the worse case to put it in better perspective. I also should have compared it as twice or with the rate of difference instead of saying 10 units longer.