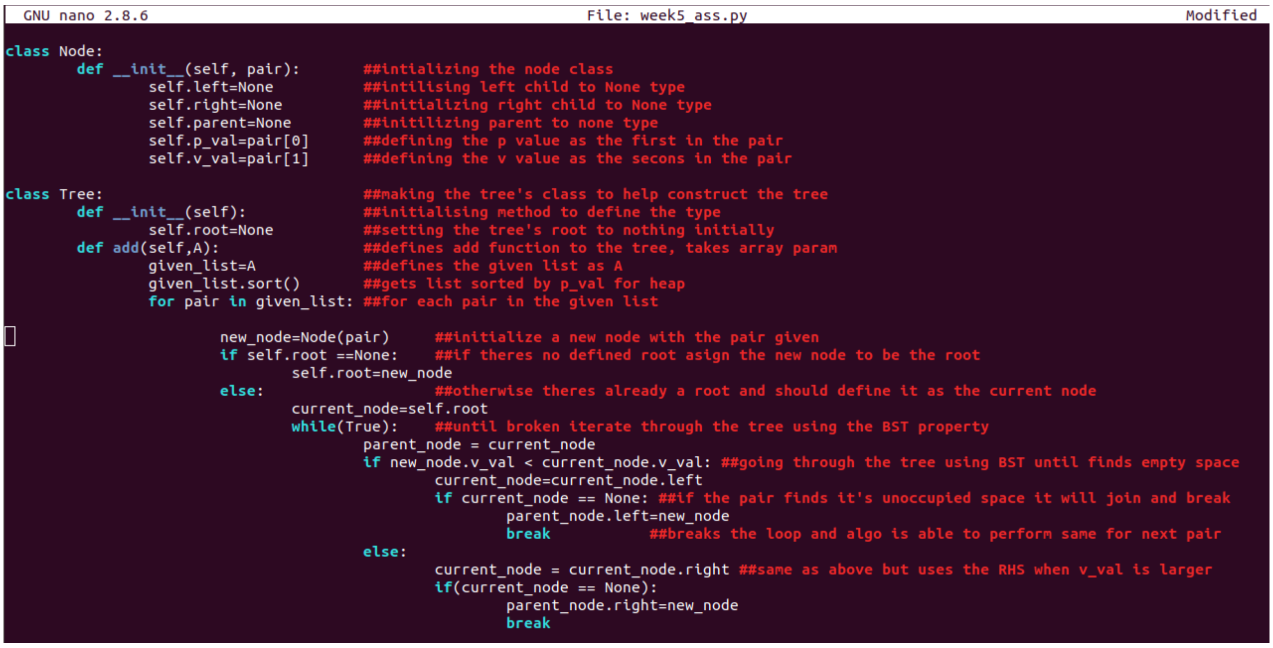
Assignment Week 5: COMP2123

***Problem 1:***

***Let ( p 0 , v 0 ) , . . . , ( p n − 1 , v n − 1 ) be a collection of n priority-value pairs. We say that a tree T storing these items is a binary search heap if T has the binary search property for the values of the items and has the heap-order property for the priority of the items. Your task is to design an algorithm for constructing a binary search heap given n priority-value pairs.***

My algorithm:



```
  GNU nano 2.8.6                          File: week5_ass.py                          Modified

class Node:
        def __init__(self, pair):       ##intializing the node class
                self.left=None          ##intilising left child to None type
                self.right=None         ##initializing right child to None type
                self.parent=None        ##initilizing parent to none type
                self.p_val=pair[0]      ##defining the p value as the first in the pair
                self.v_val=pair[1]      ##defining the v value as the secons in the pair

class Tree:                             ##making the tree's class to help construct the tree
        def __init__(self):             ##initialising method to define the type
                self.root=None          ##setting the tree's root to nothing initially
        def add(self,A):                ##defines add function to the tree, takes array param
                given_list=A            ##defines the given list as A
                given_list.sort()       ##gets list sorted by p_val for heap
                for pair in given_list: ##for each pair in the given list

                        new_node=Node(pair)     ##initialize a new node with the pair given
                        if self.root ==None:    ##if theres no defined root asign the new node to be the root
                                self.root=new_node
                        else:                   ##otherwise theres already a root and should define it as the current node
                                current_node=self.root
                                while(True):    ##until broken iterate through the tree using the BST property
                                        parent_node = current_node
                                        if new_node.v_val < current_node.v_val: ##going through the tree using BST until finds empty space
                                                current_node=current_node.left
                                                if current_node == None: ##if the pair finds it's unoccupied space it will join and break
                                                        parent_node.left=new_node
                                                        break           ##breaks the loop and algo is able to perform same for next pair
                                        else:
                                                current_node = current_node.right ##same as above but uses the RHS when v_val is larger
                                                if(current_node == None):
                                                        parent_node.right=new_node
                                                        break
```

The algorithm is shown in a text editor above, made with python. The comments describe what the purpose of each line is. Provided below is a text version of the algorithm.

```
class Node:
  def __init__(self, pair):    ##intializing the node class
    self.left=None       ##intilising left child to None type
    self.right=None      ##initializing right child to None type
    self.parent=None     ##initilizing parent to none type
    self.p_val=pair[0]     ##defining the p value as the first in the pair
    self.v_val=pair[1]     ##defining the v value as the secons in the pair

class Tree:           ##making the tree's class to help construct the tree
  def __init__(self):        ##initialising method to define the type
    self.root=None       ##setting the tree's root to nothing initially
  def add(self,A):        ##defines add function to the tree, takes array param
    given_list=A       ##defines the given list as A
    given_list.sort()      ##gets list sorted by p_val for heap
    for pair in given_list:    ##for each pair in the given list

      new_node=Node(pair)    ##initialize a new node with the pair given
      if self.root ==None:    ##if theres no defined root asign the new node to be the root
        self.root=new_node
      else:          ##otherwise theres already a root and should define it as the current node
        current_node=self.root
        while(True):    ##until broken iterate through the tree using the BST property
          parent_node = current_node
          if new_node.v_val < current_node.v_val: ##going through the tree using BST until finds empty space
            current_node=current_node.left
            if current_node == None: ##if the pair finds it's unoccupied space it will join and break
              parent_node.left=new_node
              break       ##breaks the loop and algo is able to perform same for next pair
```

```
else:
   current_node = current_node.right ##same as above but uses the RHS when v_val is larger
   if(current_node == None):
      parent_node.right=new_node
      Break
```

Explaining algorithm in Plain English:

Essentially this algorithm was designed to make a BST that maintained both the BST property for the first value in the given Array, I.e. the V in Array = [[p,v]], whilst maintaining a heap property for the first value, I.e. the P in Array =[[p,v]]. In order to do this The algorithm defines two classes, one of which being a class for the tree object itself, and the other being a class to define the Nodes within the tree, when given the [p,v] pair in an array.

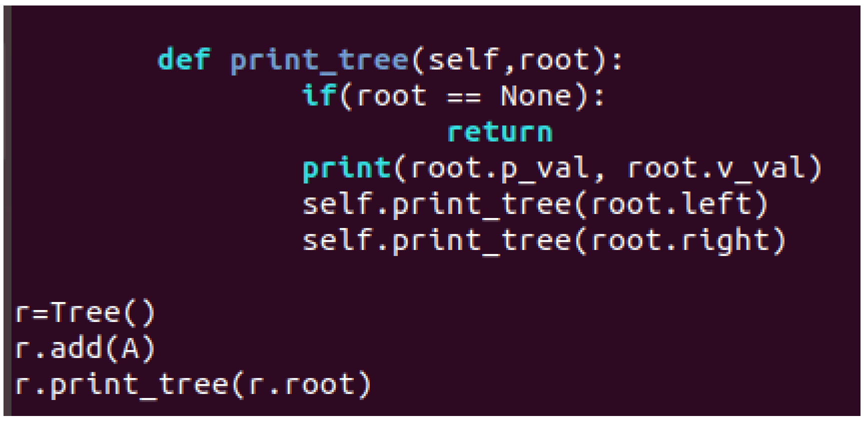
The algorithm assumes that the user will provide it with an array, either in order or out of order in terms of the p value. When an array is given to the Tree class, it has already set its root value to None. The algorithm then sorts the values in the array in order based on their p value by using pythons .sort() function. Once this is done, the array can now be arranged into a binary search tree as the heap property should be maintained (the heap property is that the p value for each Node is bigger than it's parents P value).

In order to maintain the BST property, the algorithm iterates through each pair in the given array that are now sorted in order. With each pair in the array the algorithm now pays attention to the v value. If there is nothing in the tree class yet, the algorithm will define a new Node object from the first value in the list and define it to be the root of the tree. For each following pair given in the array, the tree will iterate until it finds an empty space using the .left and .right parts of the node objects as it makes its way down the tree starting from the root. If the v value is smaller than its parent the algorithm will continue to the left and otherwise t will go to its right. When it finally finds an empty spot, the new object will place itself in the tree and the algorithm will break and repeat for the next pair in the array.

Proving the algorithms correctness:

For the algorithm to be correct it needs to maintain for each node in the tree, the parent's p value < child's p value && that for each node in the tree left child's v value < parent's v value < right child's v value.

As we can see when we run the algorithm with a python interpreter with the Array A = [[1,3],[3,2],[2,1]] we expect the algorithm to preserve both of these properties, we expect there to be a parent and children. The algorithm used to check the values in our tree is as follows:



```
        def print_tree(self,root):
                if(root == None):
                        return
                print(root.p_val, root.v_val)
                self.print_tree(root.left)
                self.print_tree(root.right)

r=Tree()
r.add(A)
r.print_tree(r.root)
```

With this printing method we expect the tree to print the root pair first followed by the left pair and the right, if the node doesn't exist it won't print: the output we get is the following:

1, 3

2, 1

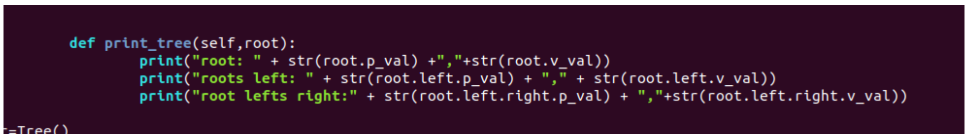
3, 2

This means the root it selected has the p value of 1 which is the lowest which maintains the heap property and the v value of 3. The algorithm would have placed the order based on the p value in order so this means that with this method the firs value in the pair should be getting larger with each printed value, which is true from what we can see so this must be correct.

In terms of whether they are right or left children in order to maintain the BST property hopefully the tree looks something like this.

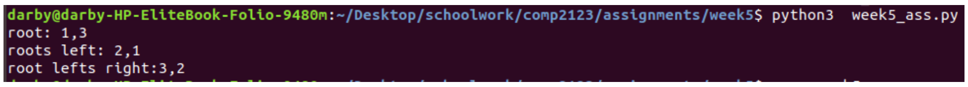
```
                 (1,3)
                 /    \
             (2,1)      None
             /    \
        None       (3,2)
```

In order to see if this is correct, as it maintains both properties, we can hard code this into the program to confirm.



```
    def print_tree(self,root):
            print("root: " + str(root.p_val) +","+str(root.v_val))
            print("roots left: " + str(root.left.p_val) + "," + str(root.left.v_val))
            print("root lefts right:" + str(root.left.right.p_val) + ","+str(root.left.right.v_val))

t=Tree()
```



```
darby@darby-HP-EliteBook-Folio-9480m:~/Desktop/schoolwork/comp2123/assignments/week5$ python3  week5_ass.py
root: 1,3
roots left: 2,1
root lefts right:3,2
```

As we can see we get the same result before as if we didn’t hard code so we know the structure fo the tree is the same as predicted with the tree above which maintains both properties.

Now we have seen that the tree works for one array would the array work for all arrays? Essentially to think about this we need to think about what we’re looking for; we know that in order to be correct the heap and the BST properties need to be maintained. The logic of the algorithm first maintains the heap property by the .sort() function which orders according to the p value in the pairs given in the array. As we can assume that no p or v values will be duplicated (as stated in the brief) we know that there will always be an order given and thus the heap property must be maintained.

This same logic can be applied to the v values in the pair. As they are assumed not be be duplicated then we can say that there will always be an answer to whether the v value of a current pair is bigger or smaller than the value of the parent’s v value. Thus, we will always be able to place it in the tree.

Therefore because we know that the logic of the algorithm works with one example we can say that the same logic can be applied to any other example assuming that within the given array v or p values cannot be repeated as said in the brief, because of this we know we will always be able to iterate effectively through the Nodes and will always be able to apply this logic to further trees.

Analysing the Complexity of the Algorithm:

As we can see the algorithm does not utilize recursion rather just for loops. In order to assess the complexity, we must look for the most time-consuming part of the algorithm which would run in the worst-case scenario. We can see that the most time-consuming part of the algorithm is the while(true) loop nested inside the for loop. This in the wore case scenario would have to run all of the primitive functions within the for-loop n times and all of the primitive functions within the while loop n times, combining these two the time complexity at its worst would be running at O(n^2).