Pimpri Chinchwad College of Engineering
Computer Laboratory - IV

Assignment No - A1

1 TITLE :

Using Divide and Conquer Strategies design a cluster/Grid of Computers in network to run a function for Binary Search Tree using Python.

2 OBJECTIVE :

To study Divide and Conquer strategies for Binary Search Tree

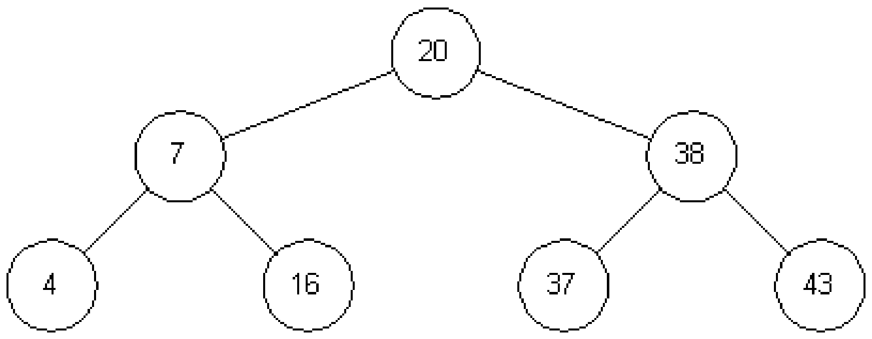
3 THEORY :

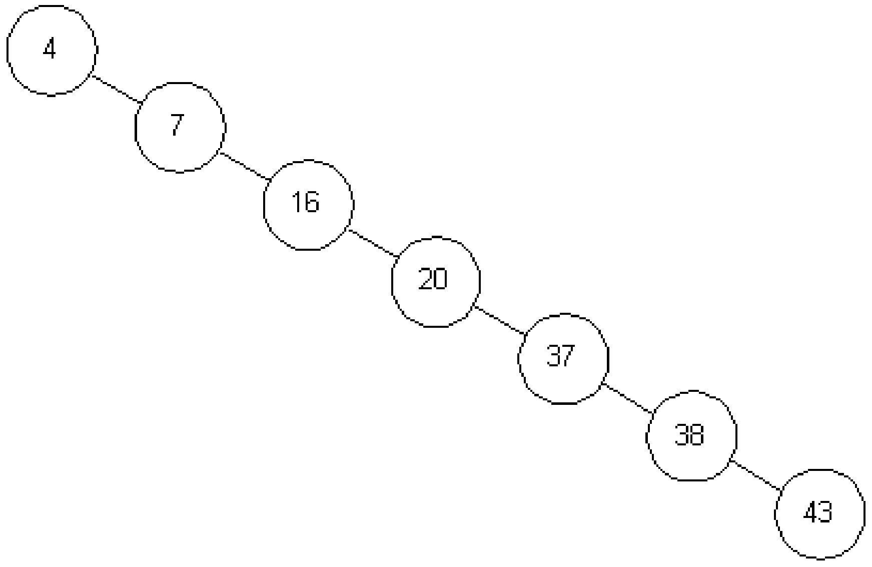
3.1 Divide and Conquer :

- Given a function to compute on an inputs the divide and conquer strategy suggest splitting the input into k distinct input sets ,such that 1¡k¡n yielding k sub problems.
- These sub problems must be solve and then method must be found to combine the sub solution into a solution of a whole.
- If the sub problems are relatively large then the divide and conquer strategy can be possibly reapplied.
- Often the sub problems are of the same type as the same original problem for which the reapplication of divide and conquer principle is naturally expressed by a recursive algorithm.

3.2 Binary Search Tree :

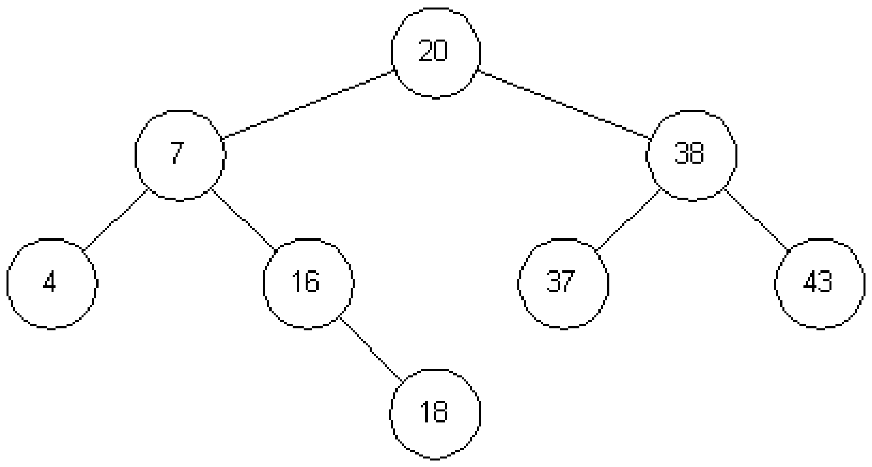
- A binary search tree is a tree where each node has a left and right child. Either child, or both children, may be missing. Figure 3-2 illustrates a binary search tree. Assuming k represents the value of a given node, then a binary search tree also has the following property: all children to the left of the node have values smaller than k, and all children to the right of the node have values larger than k. The top of a tree is known as the root, and the exposed nodes at the bottom are known as leaves. In Figure 3-2, the root is node 20 and the leaves are nodes 4, 16, 37, and 43. The height of a tree is the length of the longest path from root to leaf. For this example the tree height is 2.
- To search a tree for a given value, we start at the root and work down. For example, to search for 16, we first note that 16 ¡ 20 and we traverse to the left child. The second comparison finds that 16 ¿ 7, so we traverse to the right child. On the third comparison, we succeed.
- Each comparison results in reducing the number of items to inspect by one-half. In this respect, the algorithm is similar to a binary search on an array. However, this is true only if the tree is balanced. For example, Figure 3-3 shows another tree containing the same values. While it is a binary search tree, its behavior is more like that of a linked list, with search time increasing proportional to the number of elements stored.

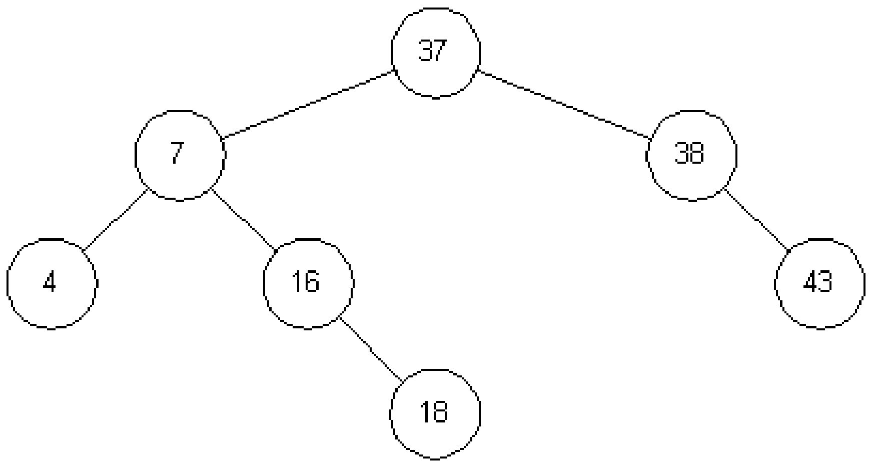




3.2.1 Insertion and deletion :

- Let us examine insertions in a binary search tree to determine the conditions that can cause an unbalanced tree. To insert an 18 in the tree in Figure 3-2, we first search for that number. This causes us to arrive at node 16 with nowhere to go. Since 18 ¿ 16, we simply add node 18 to the right child of node 16 (Figure 3-4).
- Now we can see how an unbalanced tree can occur. If the data is presented in an ascending sequence, each node will be added to the right of the previous node. This will create one long chain, or linked list. However, if data is presented for insertion in a random order, then a more balanced tree is possible.
- Deletions are similar, but require that the binary search tree property be maintained. For example, if node 20 in Figure 3-4 is removed, it must be replaced by node 37. This results in the tree shown in Figure 3-5. The rationale for this choice is as follows. The successor for node 20 must be chosen such that all nodes to the right are larger. Therefore we need to select the smallest valued node to the right of node 20. To make the selection, chain once to the right (node 38), and then chain to the left until the last node is found (node 37). This is the successor for node 20.





3.3 Code:

3.3.1 Server.py

```
import socket, pickle

class BST:
def __init__(self, val=None):
self.left = None
self.right = None
self.val = val

def __str__(self):
return ”[
def isEmpty(self):
return self.left == self.right == self.val == None

def insert(self, val):
if self.isEmpty():
self.val = val
elif val ¡ self.val:
if self.left is None:
self.left = BST(val)
else:
self.left.insert(val)
else:
if self.right is None:
self.right = BST(val)
else:
self.right.insert(val)
```

```
HOST = 'localhost'
PORT = 50007
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
data = conn.recv(4096)
if not data: break
data_arr = pickle.loads(data)
conn.close()
print 'Received', repr(data_arr)

iter = BST()
for i in data_arr:
#print i
iter.insert(i)
print iter
```

3.3.2 Client.py

```
import socket, pickle

HOST = 'localhost'
PORT = 50007
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
arr = [7,1,9,12,6,88,11,65]
data_string = pickle.dumps(arr)
s.send(data_string)
s.close()
```

3.3.3 Output

```
student@ubuntu: pythonserver.py
Connectedby('127.0.0.1', 52393)
Received[7, 1, 9, 12, 6, 88, 11, 65]
/[[None, 1, [None, 6, None]], 7, [None, 9, [[None, 11, None], 12, [[None, 65, None], 88, None]]]]


student@ubuntu : python client.py
```

3.4 MATHEMATICAL MODEL :

S=s,e,X,Y,Fme,Mem-shared,DD,NDD
s=start state
e=end state
X=input(numbers)
a[n] = no.1, no.2, ... , no.n
n = size of array
search key = Element which is to be search
Y=output(formation of binary search tree)
status = search key found or not Mem-shared=memory shared by the program
DD=Deterministic Data
NDD=Non-Deterministic Data
Fme=Algorithm(BST)

1. Calculating Top, Bottom : top = a[0]
bottom = a[n-1]
2. Calculating mid : mid = (top + bottom)/2 status = true j mid = searchkey top = mid + 1 j mid ¡ searchkey bottom = mid - 1 j mid ¿ searchkey status = false j top ¿ bottom

4 Testing

4.1 Positive Testing

Sr. No.	**Test Condition**	**Steps to be Executed**	**Expected Result**	Actual Result
1	Enter the Key to be found	Press Enter	Display	Same as Expected
2	Find the key at particular position	Press Enter	Position of key	Same as Expected

4.2 Negative Testing

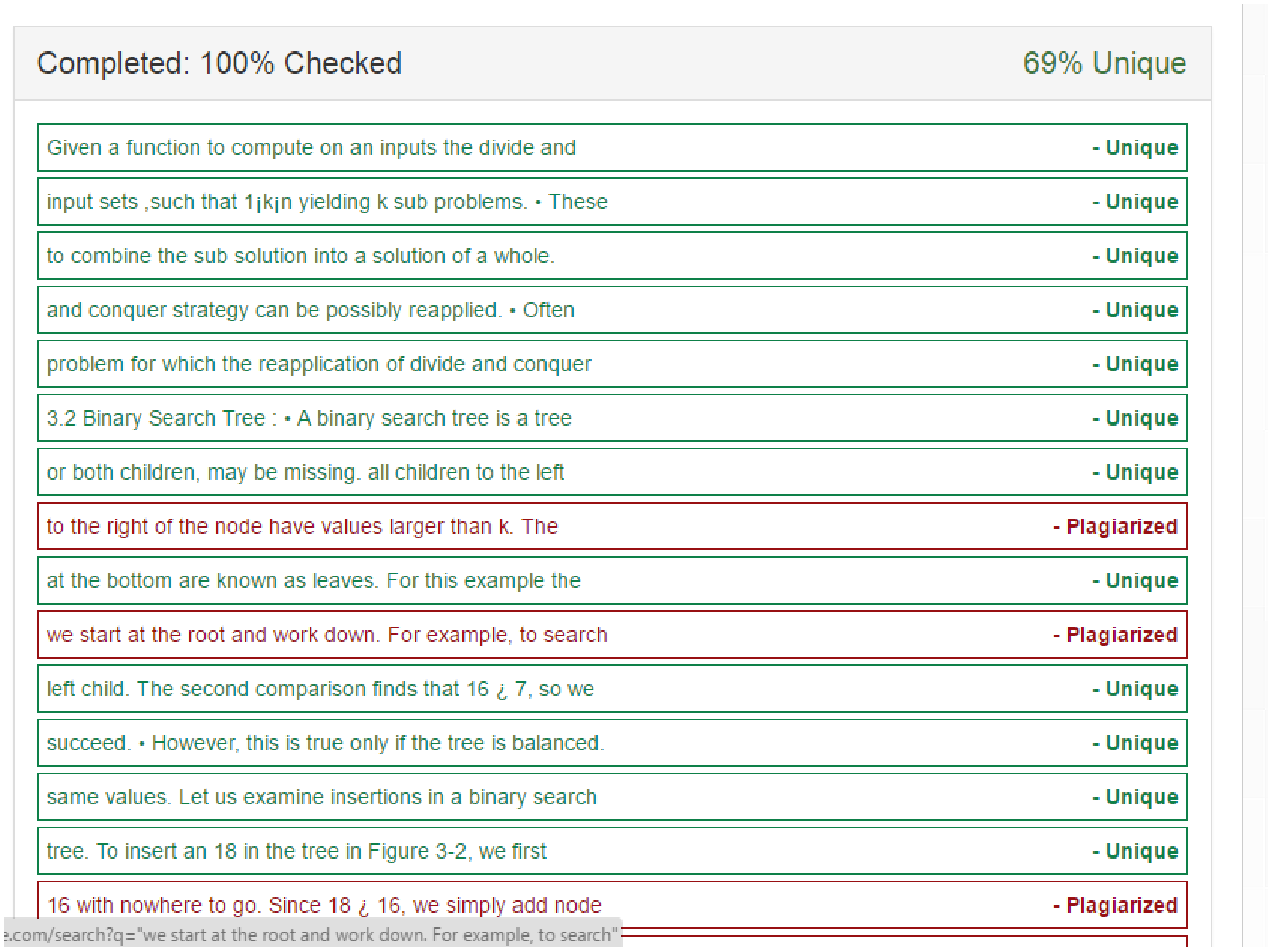
Sr. No.	**Test Condition**	**Steps**	**Expected Result**	Actual Result
1	Enter the Key not in array	Give key	If not display error message	Display position of key

5 CONCLUSION :

Thus we have successfully studied Divide and Conquer Strategies to design a cluster/Grid of Computers in network to run a function for Binary Search Tree using Python.

Roll No.	**Name of Student**	**Date of Performance**	**Date of Submission**
302	Abhinav Bakshi	20/12/16	23/12/16

6 PLAGARISM REPORT :



Completed: 100% Checked 69% Unique

Given a function to compute on an inputs the divide and	- Unique
input sets ,such that 1¡k¡n yielding k sub problems. • These	- Unique
to combine the sub solution into a solution of a whole.	- Unique
and conquer strategy can be possibly reapplied. • Often	- Unique
problem for which the reapplication of divide and conquer	- Unique
3.2 Binary Search Tree : • A binary search tree is a tree	- Unique
or both children, may be missing. all children to the left	- Unique
to the right of the node have values larger than k. The	- Plagiarized
at the bottom are known as leaves. For this example the	- Unique
we start at the root and work down. For example, to search	- Plagiarized
left child. The second comparison finds that 16 ¿ 7, so we	- Unique
succeed. • However, this is true only if the tree is balanced.	- Unique
same values. Let us examine insertions in a binary search	- Unique
tree. To insert an 18 in the tree in Figure 3-2, we first	- Unique
16 with nowhere to go. Since 18 ¿ 16, we simply add node	- Plagiarized

.com/search?q="we start at the root and work down. For example, to search"