Sixth practical exercise

Python ADT Tree

Doha Thabet Hassan Mahmoud
ID: 230AMB014

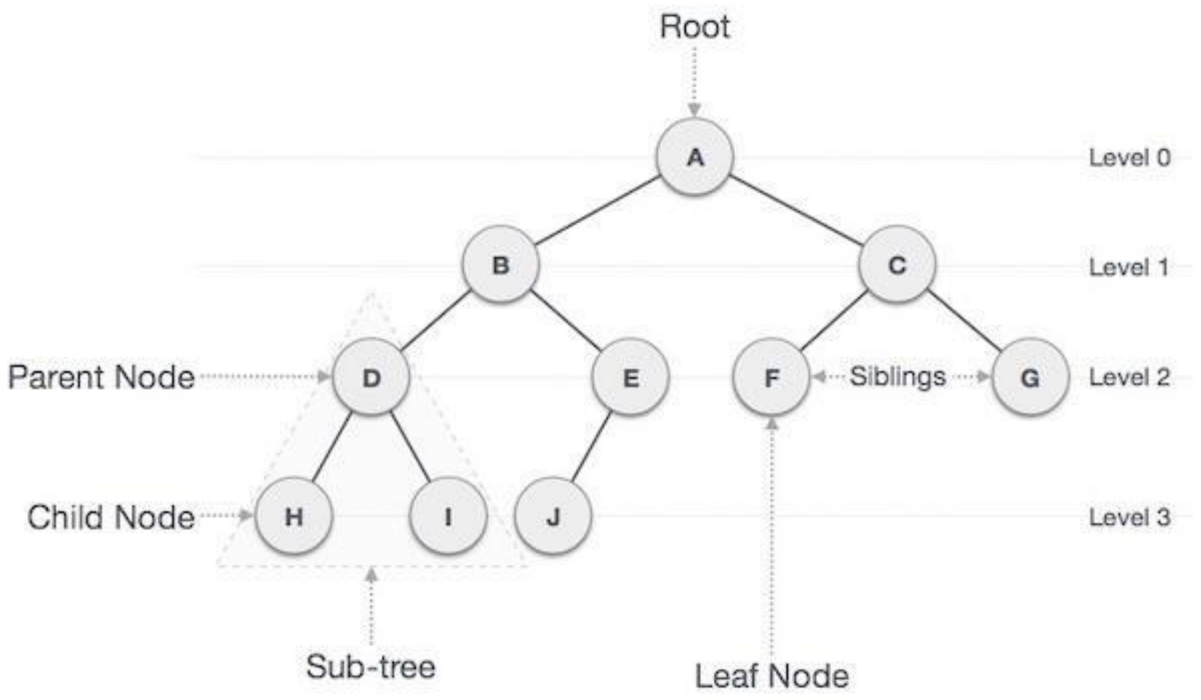
Task 1: Binary Search Tree

Introduction:

A binary search tree (BST) is a binary tree data structure with the following characteristics:

Value Ordering: The values in a BST are sorted or ordered. All of the values in a node's left subtree are less than its value, and all of the values in its right subtree are greater than its value. This ordering characteristic enables fast searching, inserting, and deleting operations.

A BST is a binary tree, which means that each node can have no more than two child nodes: a left child and a right child. The left child indicates values less than or equal to the current node, whereas the right child represents values larger than or equal to the current node. In the figure is an example of binary search tree:



Picture citation: *Data Structure and algorithms - tree*. Online Courses and eBooks Library. (n.d.). https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

Code explanation:

Node Type: A node in the binary search tree is represented by the Node class.

It has three characteristics:

- data: Holds the node's value.
- lchild: Indicates the node's left child.
- rchild: Indicates the correct child node.

Class BinarySearchTree: The binary search tree is represented by the BinarySearchTree class.

It employs the following techniques:

__init__ (self, node_list): The constructor starts the binary search tree by passing in a list of nodes. The first element of the node_list is used to generate the root node. It then loops through the remaining node_list elements, inserting each one into the tree via the insert method.

self. search (node, parent, data): This method uses a search to locate a specific value in the tree. It accepts as input a node, its parent, and the target data. Based on the comparison of the node's data with the target data, it recursively looks for the target data in the left or right subtree.

insert (data, self): The insert method adds a new node to the tree. Using the search approach, it first looks for the right spot to insert the new node. If the node does not already exist in the tree (as a result of the search), it generates a new node with the specified data.

delete (self, root, data): Removes a node from the tree. Using the search approach, it locates the node to be deleted. If the node is discovered, it handles three scenarios:

- If the node has no left child, it is replaced by its right child.
- If the node has no right child, it is replaced by its left child.
- If the node has both left and right children, it searches for the in-order successor (the smallest node in the right subtree) and replaces the node's data with the data of the successor. The successor node is then removed.

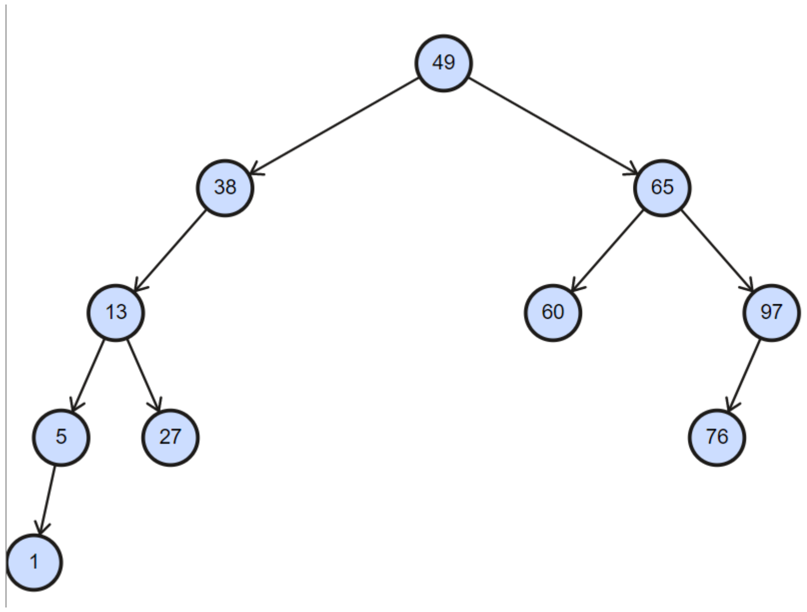
Methods for traversing trees:

- **preorder (self, node):** Traverses the tree in preorder (Root -> Left -> Right) and prints the data of each node.
- **inorder (self, node):** Traverses the tree in order (Left -> Root -> Right) and outputs the data of each node.
- **postorder (self, node):** Traverses the tree in postorder (Left -> Right -> Root) and outputs the data of each node.

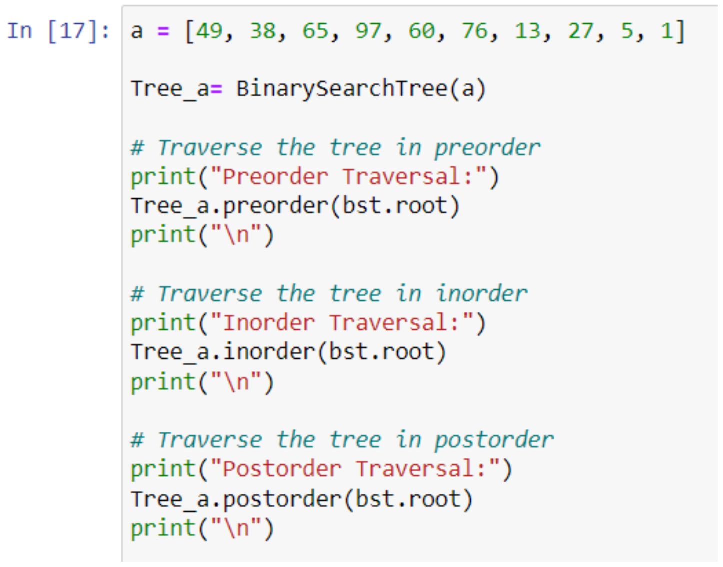
Code Execution:

1. **List a: a = [49, 38, 65, 97, 60, 76, 13, 27, 5, 1]**

Binary search tree of list a:

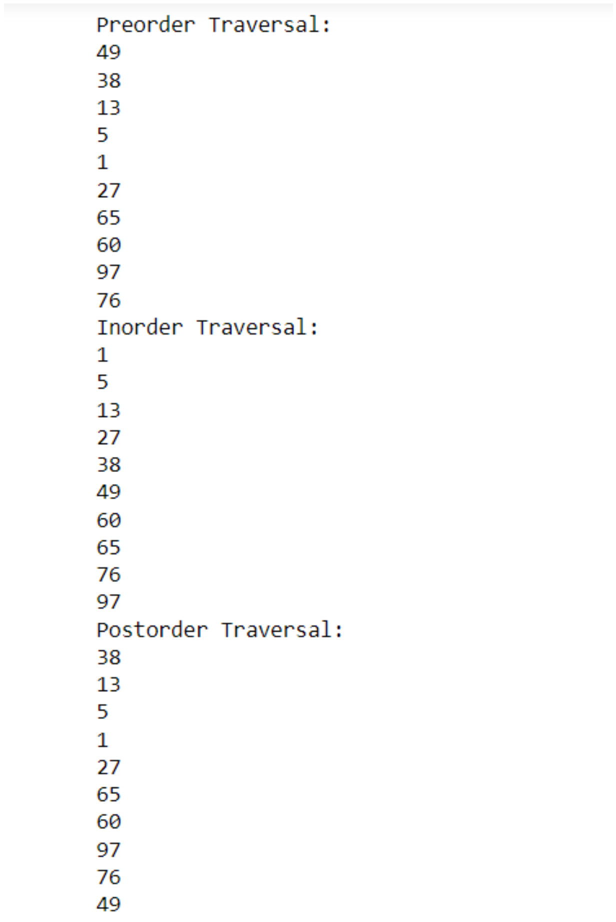


The following code is used to traverse the LIST a with the three traversing methods:



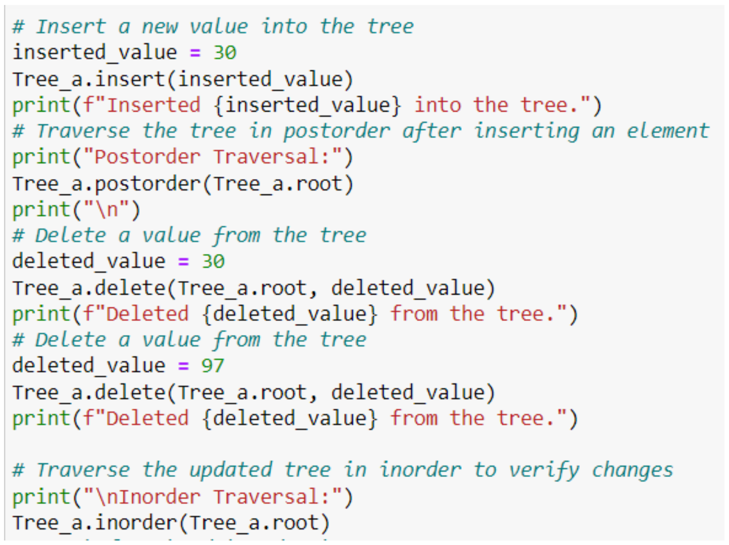
```
In [17]: a = [49, 38, 65, 97, 60, 76, 13, 27, 5, 1]

         Tree_a= BinarySearchTree(a)

         # Traverse the tree in preorder
         print("Preorder Traversal:")
         Tree_a.preorder(bst.root)
         print("\n")

         # Traverse the tree in inorder
         print("Inorder Traversal:")
         Tree_a.inorder(bst.root)
         print("\n")

         # Traverse the tree in postorder
         print("Postorder Traversal:")
         Tree_a.postorder(bst.root)
         print("\n")
```

The obtained results for traversing:



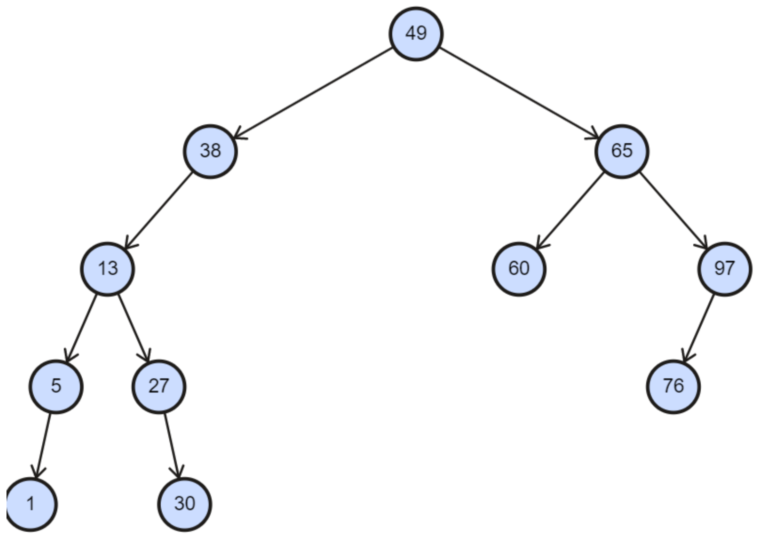
```
Preorder Traversal:
49
38
13
5
1
27
65
60
97
76
Inorder Traversal:
1
5
13
27
38
49
60
65
76
97
Postorder Traversal:
38
13
5
1
27
65
60
97
76
49
```

The following code insert a value of (30) in list a and then delete the values (30,97) and then the transverse the tree to see the changes:

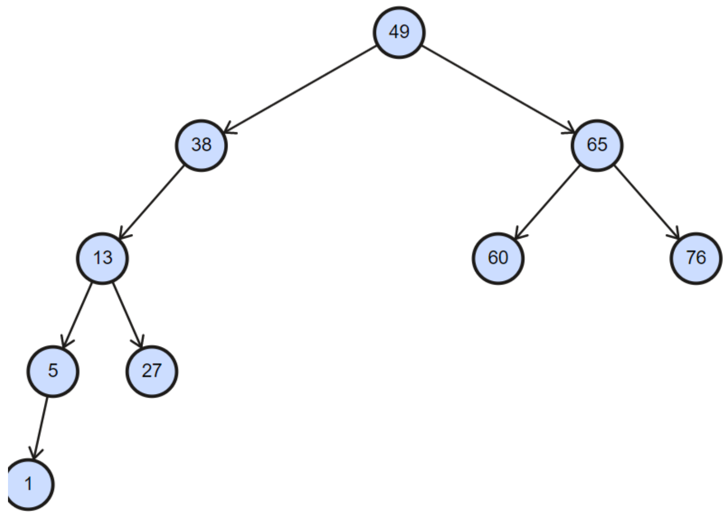


```python
# Insert a new value into the tree
inserted_value = 30
Tree_a.insert(inserted_value)
print(f"Inserted {inserted_value} into the tree.")
# Traverse the tree in postorder after inserting an element
print("Postorder Traversal:")
Tree_a.postorder(Tree_a.root)
print("\n")
# Delete a value from the tree
deleted_value = 30
Tree_a.delete(Tree_a.root, deleted_value)
print(f"Deleted {deleted_value} from the tree.")
# Delete a value from the tree
deleted_value = 97
Tree_a.delete(Tree_a.root, deleted_value)
print(f"Deleted {deleted_value} from the tree.")

# Traverse the updated tree in inorder to verify changes
print("\nInorder Traversal:")
Tree_a.inorder(Tree_a.root)
```

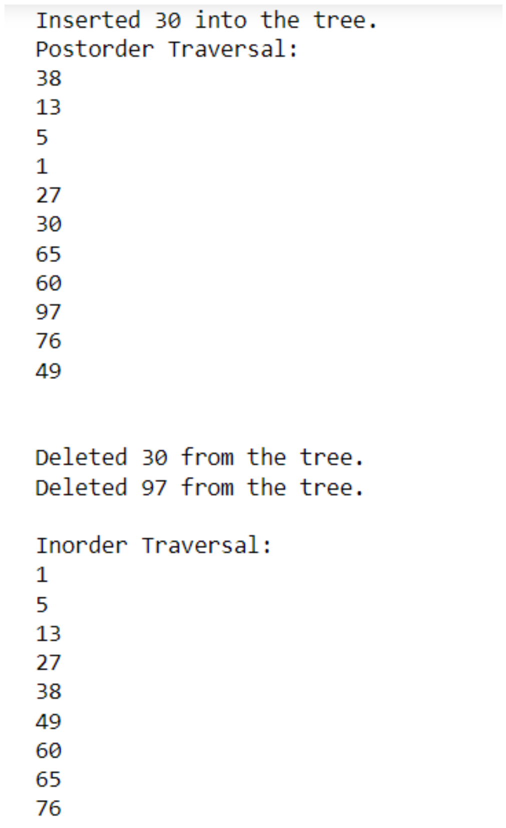
The tree after inserting the element (30):



The tree after deleting elements (30,97):



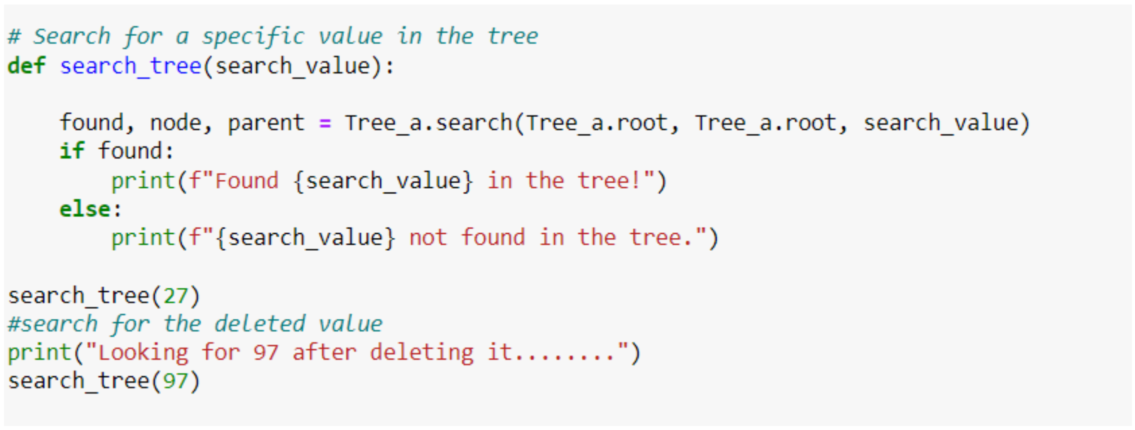
The obtained results:



```
Inserted 30 into the tree.
Postorder Traversal:
38
13
5
1
27
30
65
60
97
76
49


Deleted 30 from the tree.
Deleted 97 from the tree.

Inorder Traversal:
1
5
13
27
38
49
60
65
76
```

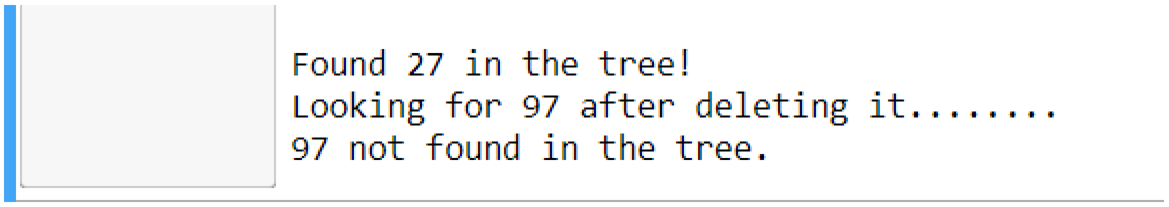
The following code to apply the search method two times:

- First one: search for element (27) which is existing in the tree.
- Second one: search for element (97) after deleting it.



```python
# Search for a specific value in the tree
def search_tree(search_value):

    found, node, parent = Tree_a.search(Tree_a.root, Tree_a.root, search_value)
    if found:
        print(f"Found {search_value} in the tree!")
    else:
        print(f"{search_value} not found in the tree.")

search_tree(27)
#search for the deleted value
print("Looking for 97 after deleting it........")
search_tree(97)
```

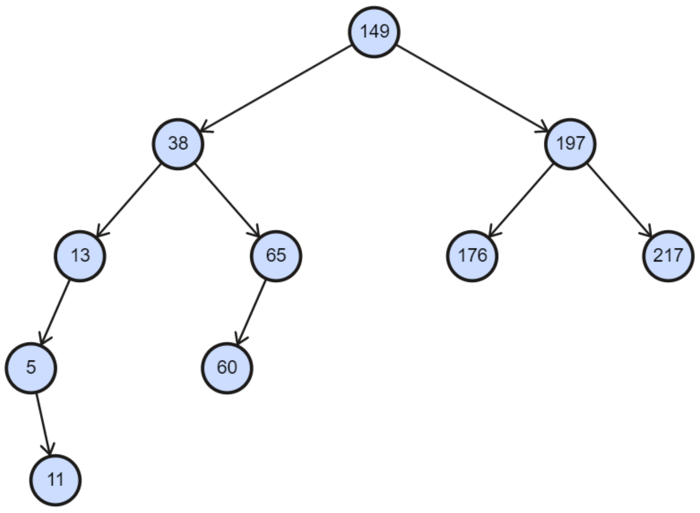
Obtained results:



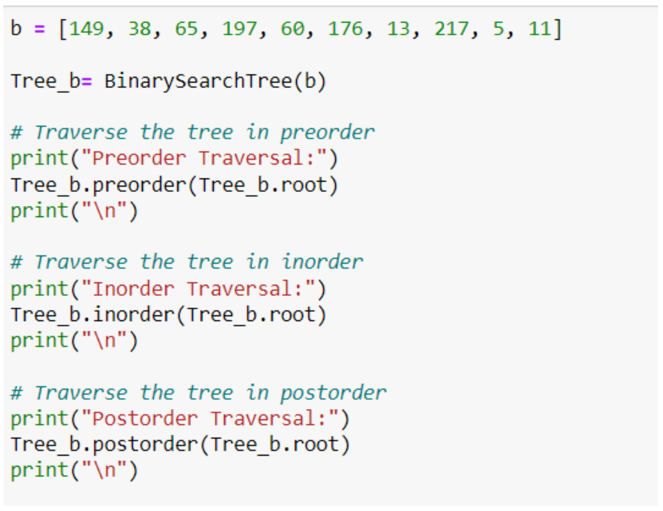
```
Found 27 in the tree!
Looking for 97 after deleting it........
97 not found in the tree.
```

2. List b: b = [149, 38, 65, 197, 60, 176, 13, 217, 5, 11]

Binary search tree of list a:

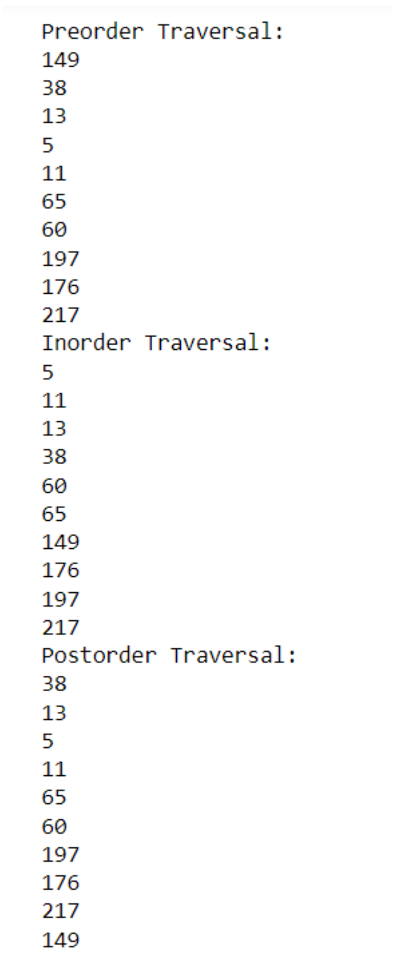


The following code is used to traverse the LIST b with the three traversing methods:



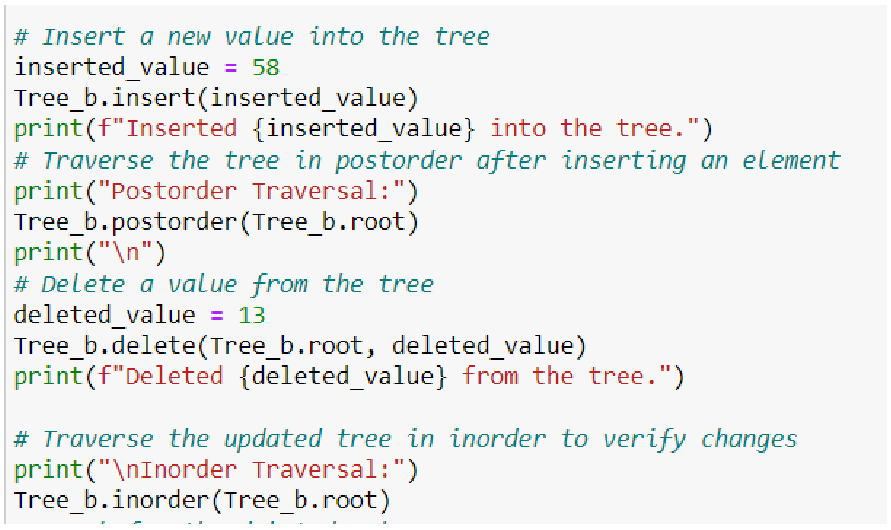
```python
b = [149, 38, 65, 197, 60, 176, 13, 217, 5, 11]

Tree_b= BinarySearchTree(b)

# Traverse the tree in preorder
print("Preorder Traversal:")
Tree_b.preorder(Tree_b.root)
print("\n")

# Traverse the tree in inorder
print("Inorder Traversal:")
Tree_b.inorder(Tree_b.root)
print("\n")

# Traverse the tree in postorder
print("Postorder Traversal:")
Tree_b.postorder(Tree_b.root)
print("\n")
```

Obtained results:



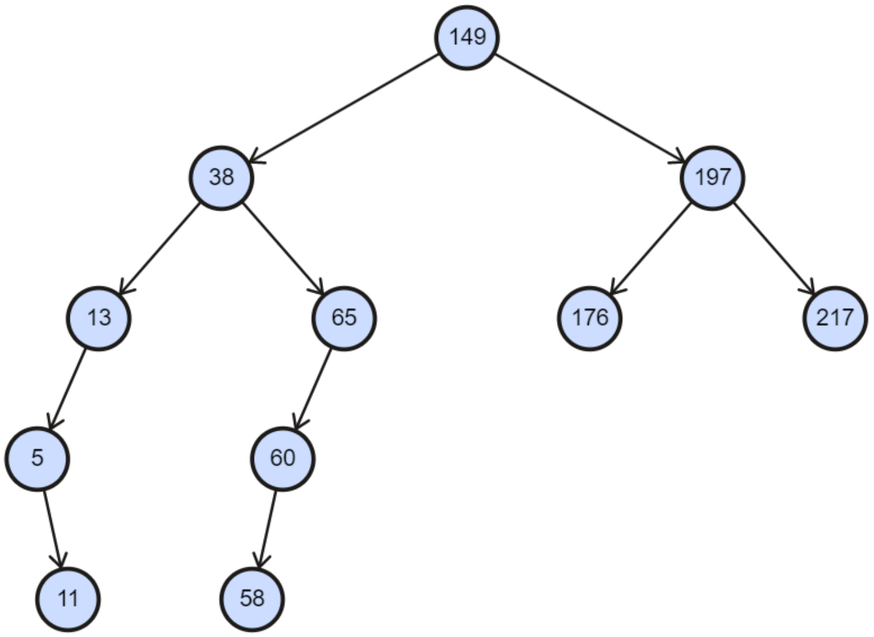
```
Preorder Traversal:
149
38
13
5
11
65
60
197
176
217
Inorder Traversal:
5
11
13
38
60
65
149
176
197
217
Postorder Traversal:
38
13
5
11
65
60
197
176
217
149
```

The following code insert a value of (58) in list a and then delete the values (13) and then the transverse the tree to see the changes:

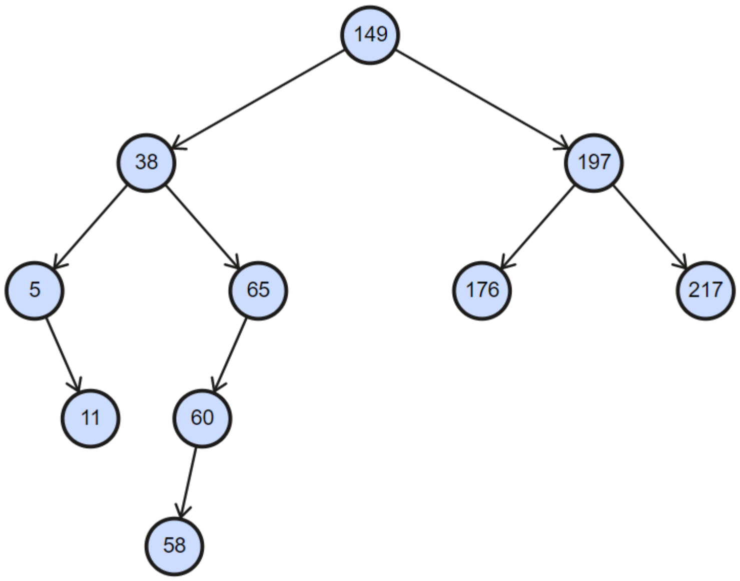


```python
# Insert a new value into the tree
inserted_value = 58
Tree_b.insert(inserted_value)
print(f"Inserted {inserted_value} into the tree.")
# Traverse the tree in postorder after inserting an element
print("Postorder Traversal:")
Tree_b.postorder(Tree_b.root)
print("\n")
# Delete a value from the tree
deleted_value = 13
Tree_b.delete(Tree_b.root, deleted_value)
print(f"Deleted {deleted_value} from the tree.")

# Traverse the updated tree in inorder to verify changes
print("\nInorder Traversal:")
Tree_b.inorder(Tree_b.root)
```

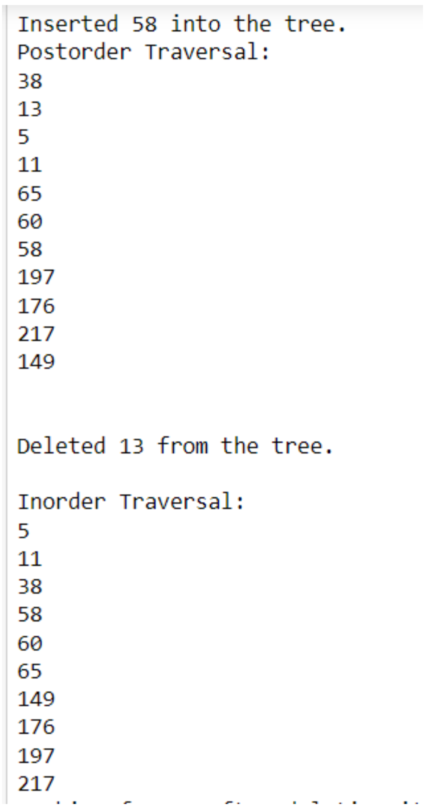
The tree after inserting the element (58):



The tree after deleting the element (13):



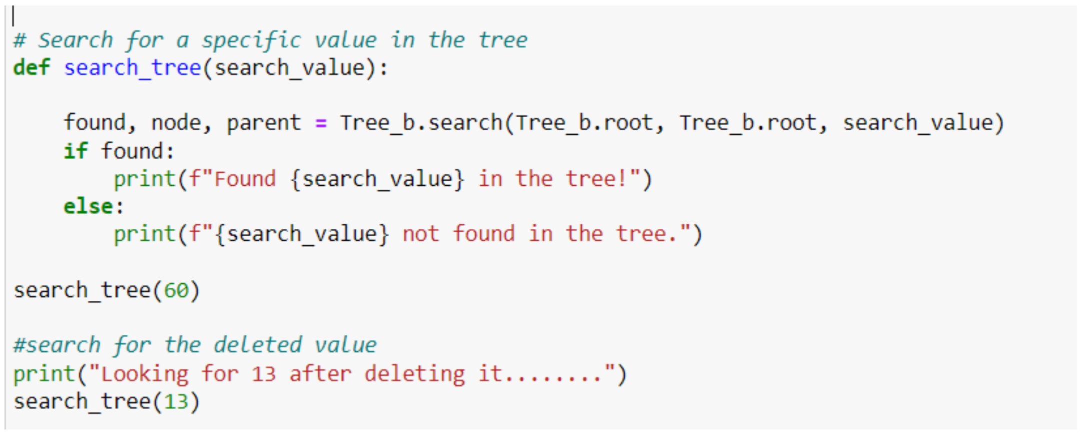
Obtained results:



```
Inserted 58 into the tree.
Postorder Traversal:
38
13
5
11
65
60
58
197
176
217
149


Deleted 13 from the tree.

Inorder Traversal:
5
11
38
58
60
65
149
176
197
217
```

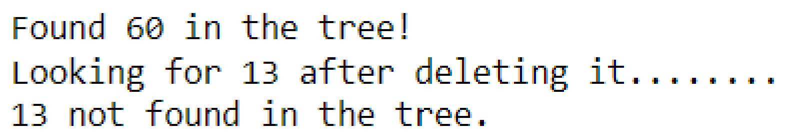
The following code to apply the search method two times:

- First one: search for element (60) which exists in the tree.
- Second one: search for element (13) after deleting it.



```python
# Search for a specific value in the tree
def search_tree(search_value):

    found, node, parent = Tree_b.search(Tree_b.root, Tree_b.root, search_value)
    if found:
        print(f"Found {search_value} in the tree!")
    else:
        print(f"{search_value} not found in the tree.")

search_tree(60)

#search for the deleted value
print("Looking for 13 after deleting it........")
search_tree(13)
```

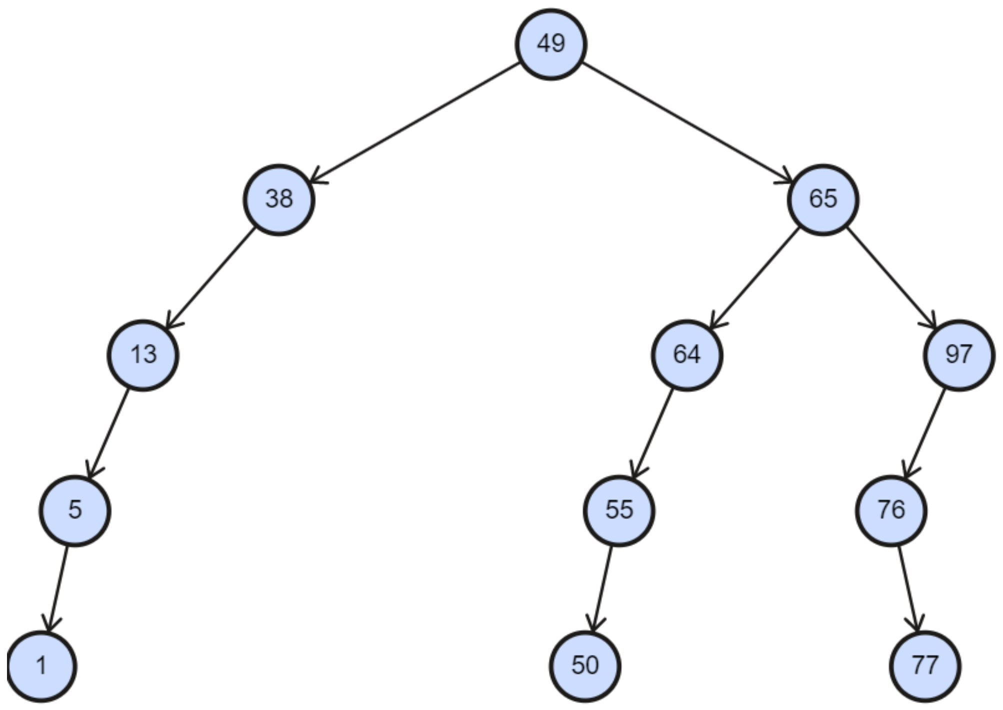
Obtained results:



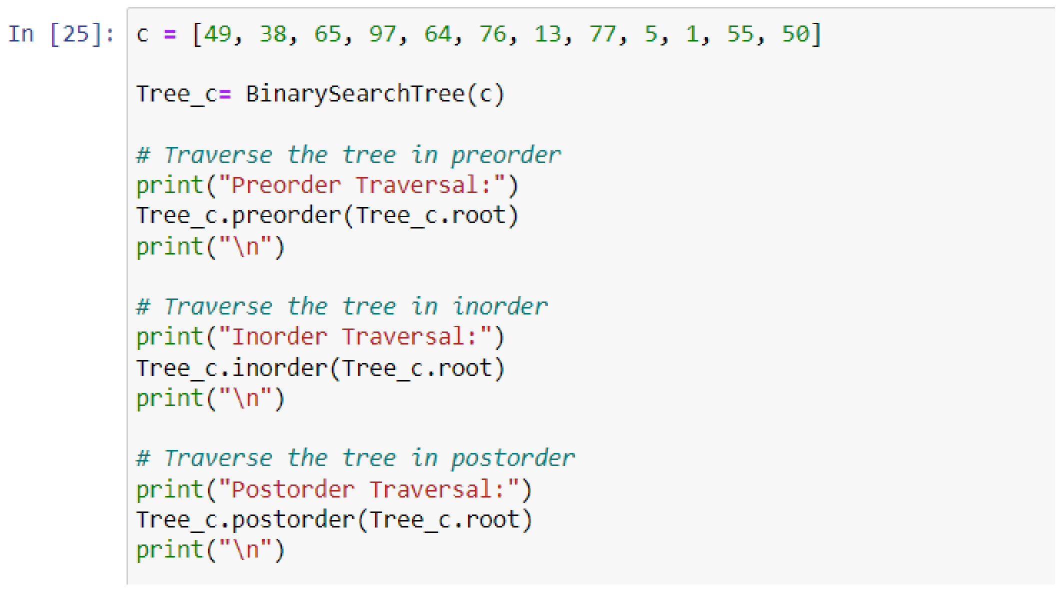
```
Found 60 in the tree!
Looking for 13 after deleting it........
13 not found in the tree.
```

3.List c: c = [49, 38, 65, 97, 64, 76, 13, 77, 5, 1, 55, 50]

Binary search tree of list c:

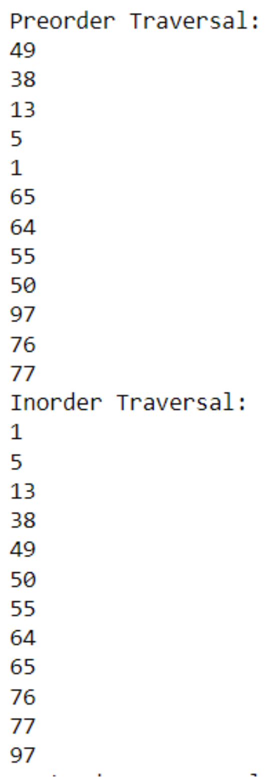


The following code is used to traverse the LIST c with the three traversing methods:

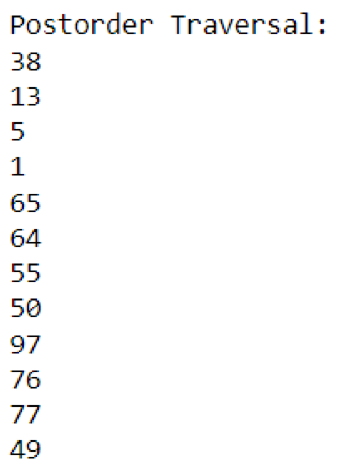


```
In [25]: c = [49, 38, 65, 97, 64, 76, 13, 77, 5, 1, 55, 50]

         Tree_c= BinarySearchTree(c)

         # Traverse the tree in preorder
         print("Preorder Traversal:")
         Tree_c.preorder(Tree_c.root)
         print("\n")

         # Traverse the tree in inorder
         print("Inorder Traversal:")
         Tree_c.inorder(Tree_c.root)
         print("\n")

         # Traverse the tree in postorder
         print("Postorder Traversal:")
         Tree_c.postorder(Tree_c.root)
         print("\n")
```

Obtained results:

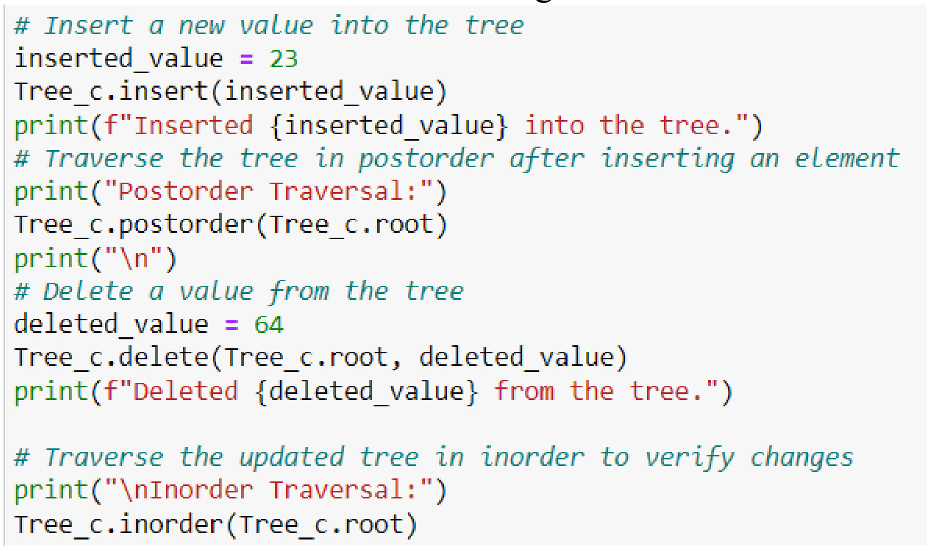


```
Preorder Traversal:
49
38
13
5
1
65
64
55
50
97
76
77
Inorder Traversal:
1
5
13
38
49
50
55
64
65
76
77
97
```



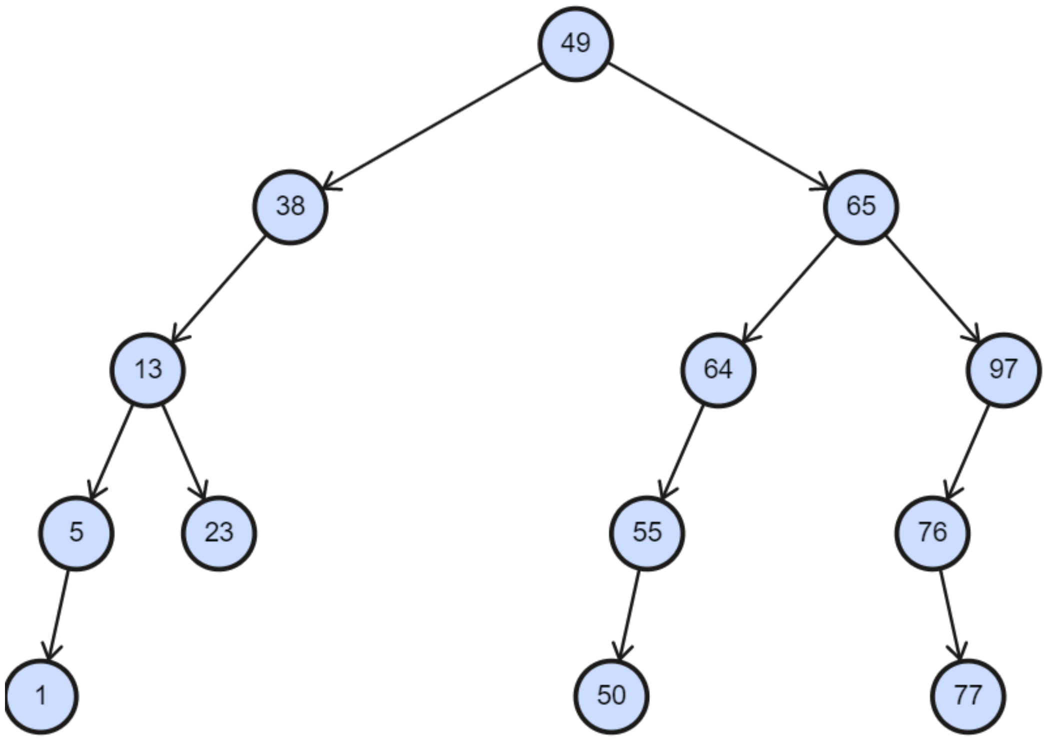
```
Postorder Traversal:
38
13
5
1
65
64
55
50
97
76
77
49
```

The following code insert a value of (23) in list a and then delete the values (64) and then the transverse the tree to see the changes:

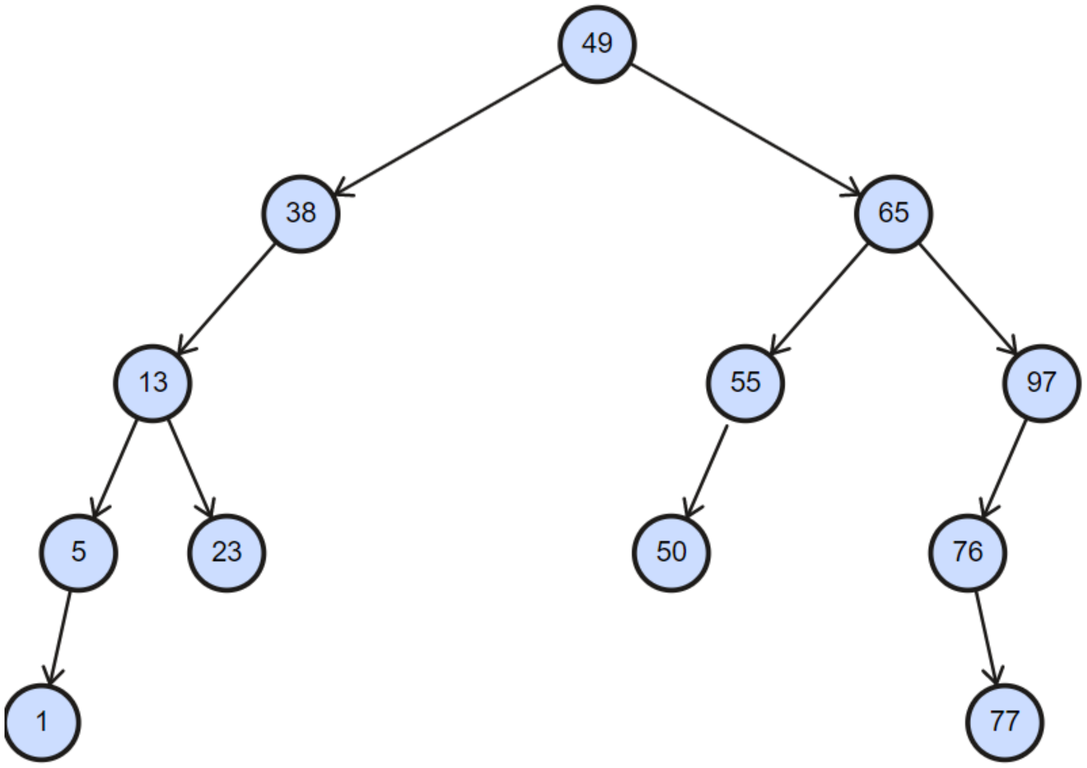


```
# Insert a new value into the tree
inserted_value = 23
Tree_c.insert(inserted_value)
print(f"Inserted {inserted_value} into the tree.")
# Traverse the tree in postorder after inserting an element
print("Postorder Traversal:")
Tree_c.postorder(Tree_c.root)
print("\n")
# Delete a value from the tree
deleted_value = 64
Tree_c.delete(Tree_c.root, deleted_value)
print(f"Deleted {deleted_value} from the tree.")

# Traverse the updated tree in inorder to verify changes
print("\nInorder Traversal:")
Tree_c.inorder(Tree_c.root)
```

The tree after inserting the element (23):



The tree after deleting elements (64):

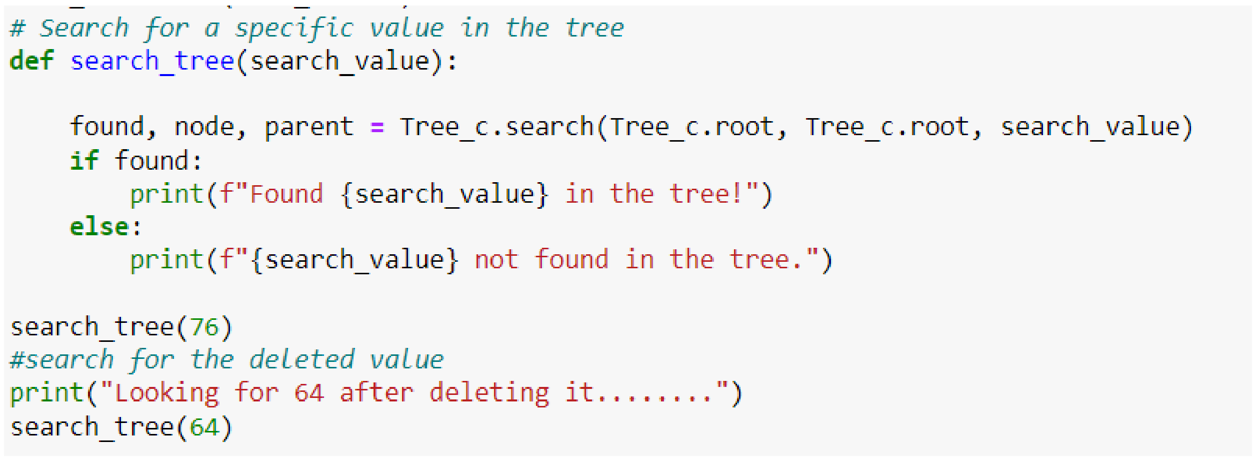


The obtained results:

```
Inserted 23 into the tree.
Postorder Traversal:
38
13
5
1
23
65
64
55
50
97
76
77
49


Deleted 64 from the tree.

Inorder Traversal:
1
5
13
23
38
49
50
55
65
76
77
97
```

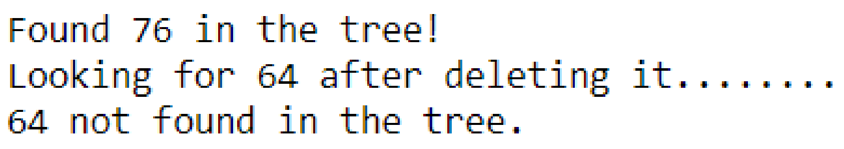
The following code to apply the search method two times:

- First one: search for element (76) which exists in the tree.
- Second one: search for element (64) after deleting it.



```python
# Search for a specific value in the tree
def search_tree(search_value):

    found, node, parent = Tree_c.search(Tree_c.root, Tree_c.root, search_value)
    if found:
        print(f"Found {search_value} in the tree!")
    else:
        print(f"{search_value} not found in the tree.")

search_tree(76)
#search for the deleted value
print("Looking for 64 after deleting it........")
search_tree(64)
```

Obtained results:

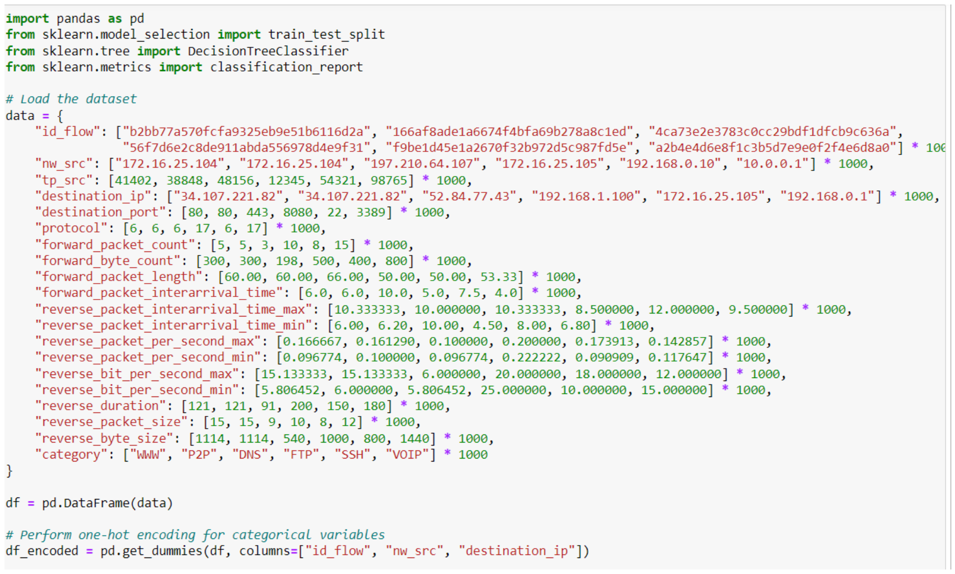


```
Found 76 in the tree!
Looking for 64 after deleting it........
64 not found in the tree.
```

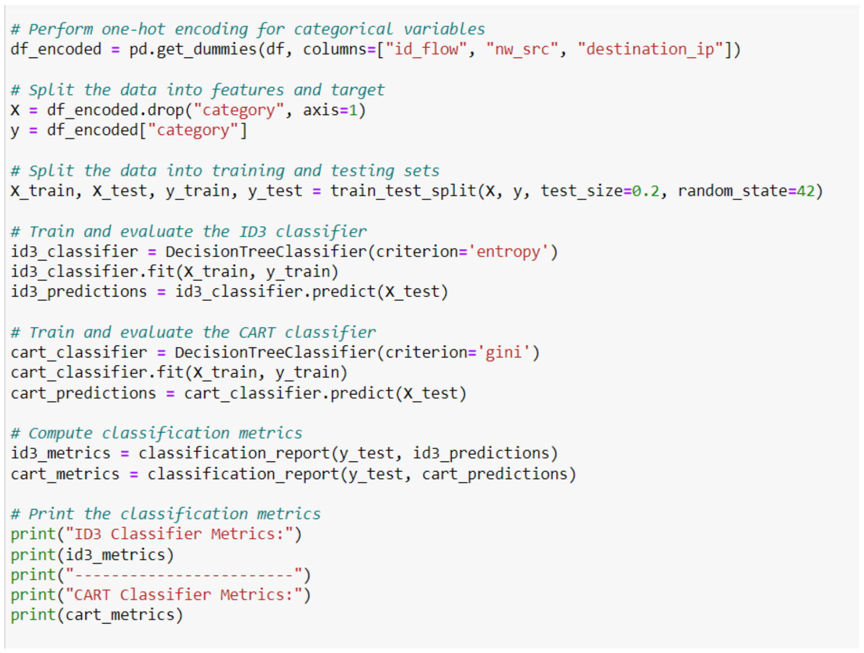
Task 2: SDN Traffic classification with DT

- A sample data is taken from the csv file (I did this because I face problems working on the CSV directly) the sample is repeated 1000 times to make a larger dataset:
- Split the data into features and target.
- Split the data into testing and training.
- Train using ID3.
- Train using CART.

Code:



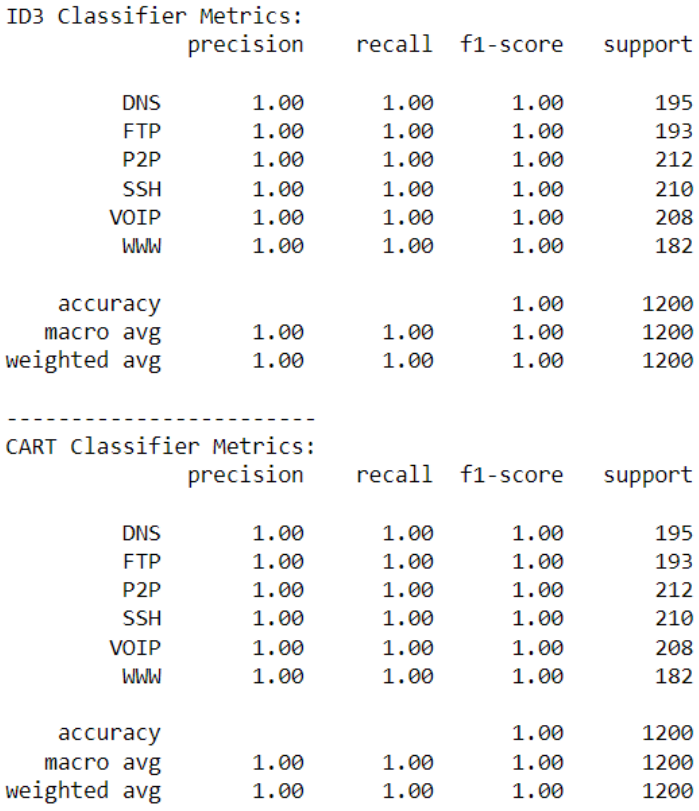
```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report

# Load the dataset
data = {
    "id_flow": ["b2bb77a570fcfa9325eb9e51b6116d2a", "166af8ade1a6674f4bfa69b278a8c1ed", "4ca73e2e3783c0cc29bdf1dfcb9c636a",
                "56f7d6e2c8de911abda556978d4e9f31", "f9be1d45e1a2670f32b972d5c987fd5e", "a2b4e4d6e8f1c3b5d7e9e0f2f4e6d8a0"] * 10(
    "nw_src": ["172.16.25.104", "172.16.25.104", "197.210.64.107", "172.16.25.105", "192.168.0.10", "10.0.0.1"] * 1000,
    "tp_src": [41402, 38848, 48156, 12345, 54321, 98765] * 1000,
    "destination_ip": ["34.107.221.82", "34.107.221.82", "52.84.77.43", "192.168.1.100", "172.16.25.105", "192.168.0.1"] * 1000,
    "destination_port": [80, 80, 443, 8080, 22, 3389] * 1000,
    "protocol": [6, 6, 6, 17, 6, 17] * 1000,
    "forward_packet_count": [5, 5, 3, 10, 8, 15] * 1000,
    "forward_byte_count": [300, 300, 198, 500, 400, 800] * 1000,
    "forward_packet_length": [60.00, 60.00, 66.00, 50.00, 50.00, 53.33] * 1000,
    "forward_packet_interarrival_time": [6.0, 6.0, 10.0, 5.0, 7.5, 4.0] * 1000,
    "reverse_packet_interarrival_time_max": [10.333333, 10.000000, 10.333333, 8.500000, 12.000000, 9.500000] * 1000,
    "reverse_packet_interarrival_time_min": [6.00, 6.20, 10.00, 4.50, 8.00, 6.80] * 1000,
    "reverse_packet_per_second_max": [0.166667, 0.161290, 0.100000, 0.200000, 0.173913, 0.142857] * 1000,
    "reverse_packet_per_second_min": [0.096774, 0.100000, 0.096774, 0.222222, 0.090909, 0.117647] * 1000,
    "reverse_bit_per_second_max": [15.133333, 15.133333, 6.000000, 20.000000, 18.000000, 12.000000] * 1000,
    "reverse_bit_per_second_min": [5.806452, 6.000000, 5.806452, 25.000000, 10.000000, 15.000000] * 1000,
    "reverse_duration": [121, 121, 91, 200, 150, 180] * 1000,
    "reverse_packet_size": [15, 15, 9, 10, 8, 12] * 1000,
    "reverse_byte_size": [1114, 1114, 540, 1000, 800, 1440] * 1000,
    "category": ["WWW", "P2P", "DNS", "FTP", "SSH", "VOIP"] * 1000
}

df = pd.DataFrame(data)

# Perform one-hot encoding for categorical variables
df_encoded = pd.get_dummies(df, columns=["id_flow", "nw_src", "destination_ip"])
```



```python
# Perform one-hot encoding for categorical variables
df_encoded = pd.get_dummies(df, columns=["id_flow", "nw_src", "destination_ip"])

# Split the data into features and target
X = df_encoded.drop("category", axis=1)
y = df_encoded["category"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train and evaluate the ID3 classifier
id3_classifier = DecisionTreeClassifier(criterion='entropy')
id3_classifier.fit(X_train, y_train)
id3_predictions = id3_classifier.predict(X_test)

# Train and evaluate the CART classifier
cart_classifier = DecisionTreeClassifier(criterion='gini')
cart_classifier.fit(X_train, y_train)
cart_predictions = cart_classifier.predict(X_test)

# Compute classification metrics
id3_metrics = classification_report(y_test, id3_predictions)
cart_metrics = classification_report(y_test, cart_predictions)

# Print the classification metrics
print("ID3 Classifier Metrics:")
print(id3_metrics)
print("------------------------")
print("CART Classifier Metrics:")
print(cart_metrics)
```

Obtained results:

The two algorithms show the same results because a small data set is provided, but the same concept is applied for larger data sets.



```
ID3 Classifier Metrics:
              precision    recall  f1-score   support

         DNS       1.00      1.00      1.00       195
         FTP       1.00      1.00      1.00       193
         P2P       1.00      1.00      1.00       212
         SSH       1.00      1.00      1.00       210
        VOIP       1.00      1.00      1.00       208
         WWW       1.00      1.00      1.00       182

    accuracy                           1.00      1200
   macro avg       1.00      1.00      1.00      1200
weighted avg       1.00      1.00      1.00      1200

------------------------
CART Classifier Metrics:
              precision    recall  f1-score   support

         DNS       1.00      1.00      1.00       195
         FTP       1.00      1.00      1.00       193
         P2P       1.00      1.00      1.00       212
         SSH       1.00      1.00      1.00       210
        VOIP       1.00      1.00      1.00       208
         WWW       1.00      1.00      1.00       182

    accuracy                           1.00      1200
   macro avg       1.00      1.00      1.00      1200
weighted avg       1.00      1.00      1.00      1200
```