

Name : Shraddha Rajkumar Kotwar
Roll No.: 14

ASSIGNMENT NO : 2

Write a program to implement Huffman Encoding using a greedy strategy.

A Huffman Tree Node:

```
class node:
    def __init__(self, freq, symbol, left=None, right=None):
        # frequency of symbol
        self.freq = freq

        # symbol name (character)
        self.symbol = symbol

        # node left of current node
        self.left = left

        # node right of current node
        self.right = right

        # tree direction (0/1)
        self.huff = ""
```

```
# utility function to print huffman
# codes for all symbols in the newly
# created Huffman tree
```

```
def printNodes(node, val=""):
    # huffman code for current node
    newVal = val + str(node.huff)

    # if node is not an edge node
    # then traverse inside it
    if(node.left):
```

```

printNodes(node.left, newVal)
if(node.right):
    printNodes(node.right, newVal)

# if node is edge node then
# display its huffman code
if(not node.left and not node.right):
    print(f"{node.symbol} -> {newVal}")

# characters for huffman tree
chars = ['a', 'b', 'c', 'd', 'e', 'f']

# frequency of characters
freq = [ 5, 9, 12, 13, 16, 45]

# list containing unused nodes
nodes = []

# converting characters and frequencies
# into huffman tree nodes
for x in range(len(chars)):
    nodes.append(node(freq[x], chars[x]))

while len(nodes) > 1:
    # sort all the nodes in ascending order
    # based on their frequency
    nodes = sorted(nodes, key=lambda x: x.freq)

    # pick 2 smallest nodes
    left = nodes[0]
    right = nodes[1]

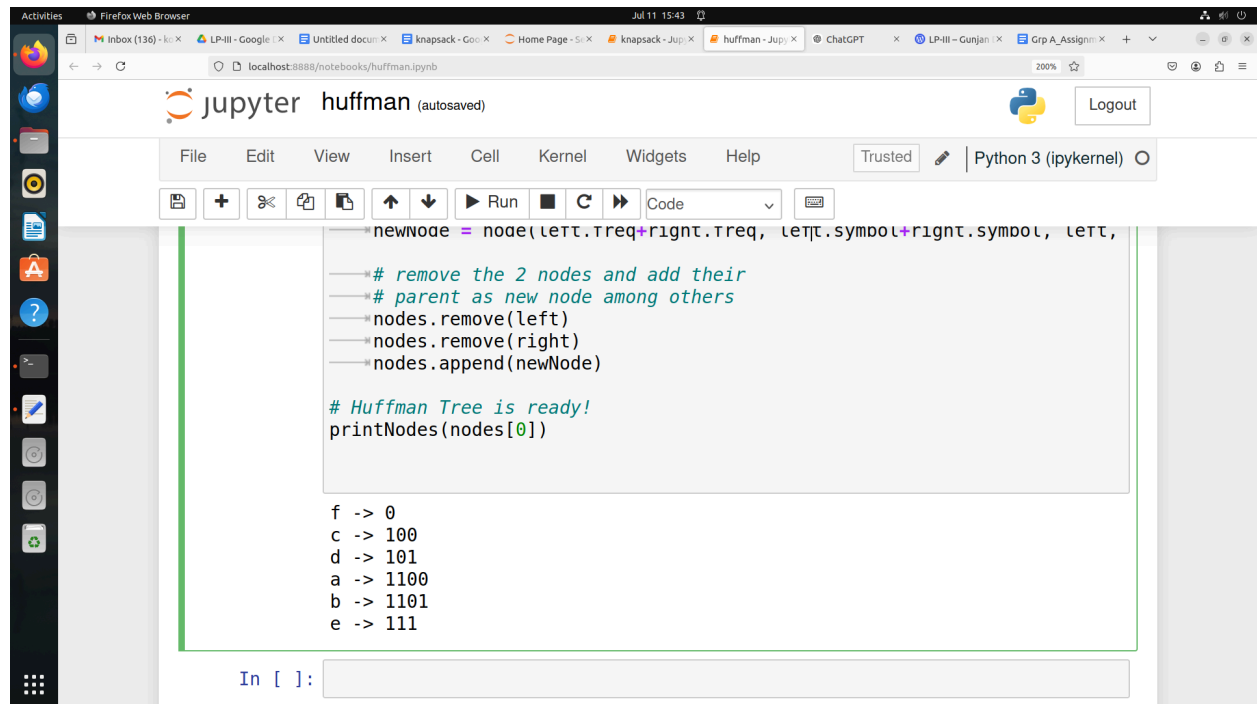
    # assign directional value to these nodes
    left.huff = 0
    right.huff = 1

    # combine the 2 smallest nodes to create
    # new node as their parent
    newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

```

```
# remove the 2 nodes and add their
# parent as new node among others
nodes.remove(left)
nodes.remove(right)
nodes.append(newNode)
```

```
# Huffman Tree is ready!
printNodes(nodes[0])
```



```
#newNode = node(left.freq+right.freq, left.symbol+right.symbol, left,
—# remove the 2 nodes and add their
—# parent as new node among others
—nodes.remove(left)
—nodes.remove(right)
—nodes.append(newNode)

# Huffman Tree is ready!
printNodes(nodes[0])

f -> 0
c -> 100
d -> 101
a -> 1100
b -> 1101
e -> 111
```

In []:

```
*left.huff = 0
*right.huff = 1

*newNode = node(left.freq+right.freq, left.symbol+right.symbol, left,
               *nodes.remove(left)
               *nodes.remove(right)
               *nodes.append(newNode)

printNodes(nodes[0])

f -> 0
c -> 100
d -> 101
a -> 1100
b -> 1101
e -> 111
```

```
class node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right
        self.huff = ""
```

```
def printNodes(node, val=""):
    newVal = val + str(node.huff)
    if(node.left):
        printNodes(node.left, newVal)
    if(node.right):
        printNodes(node.right, newVal)
    if(not node.left and not node.right):
        print(f'{node.symbol} -> {newVal}')
```

```
chars = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
freq = [ 5, 9, 12, 13, 16, 45]
```

```
nodes = []
```

```

for x in range(len(chars)):
    nodes.append(node(freq[x], chars[x]))

while len(nodes) > 1:
    nodes = sorted(nodes, key=lambda x: x.freq)
    left = nodes[0]
    right = nodes[1]
    left.huff = 0
    right.huff = 1

    newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

    nodes.remove(left)
    nodes.remove(right)
    nodes.append(newNode)

printNodes(nodes[0])

```

The screenshot shows a Jupyter Notebook running in a Firefox browser. The notebook is titled 'huff' and shows the implementation of a Huffman tree. The code is as follows:

```

for x in range(len(chars)):
    nodes.append(node(freq[x], chars[x]))

while len(nodes) > 1:
    nodes = sorted(nodes, key=lambda x: x.freq)
    left = nodes[0]
    right = nodes[1]
    left.huff = 0
    right.huff = 1

    newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

    nodes.remove(left)
    nodes.remove(right)
    nodes.append(newNode)

printNodes(nodes[0])

```

The output of the code is:

```

f -> 0
c -> 100
d -> 101
a -> 1100
b -> 1101
e -> 111

```

```
class Node:
    def __init__(self, freq, symbol, left=None, right=None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right

def buildTree(nodes):
    if len(nodes) < 2:
        return nodes[0]
    nodes = sorted(nodes, key=lambda x: x.freq)
    left = nodes[0]
    right = nodes[1]
    left.huff = 0
    right.huff = 1

    newNode = Node(left.freq+right.freq, left.symbol+right.symbol, left, right)
    nodes.remove(left)
    nodes.remove(right)
    nodes.append(newNode)

    return buildTree(nodes)

printNodes(nodes[0])

f -> 0
c -> 100
d -> 101
a -> 1100
b -> 1101
e -> 111
```

Time Complexity

1. Frequency Calculation:

- **Time Complexity:** $O(n)$
- We need to iterate through the input data to calculate the frequency of each character, where n is the length of the input data.

2. Priority Queue Construction:

- **Time Complexity:** $O(d \log d)$
- We insert each of the d distinct characters into the priority queue (min-heap). Insertion into a heap takes $O(\log d)$ time, and we perform this operation d times.

3. Tree Construction:

- **Time Complexity:** $O(d \log d)$
- We perform $d-1$ merge operations (since we start with d nodes and end up with one tree). Each merge operation involves extracting the two minimum elements from the heap (each extraction takes $O(\log d)$ time) and inserting the new node back into the heap (insertion also takes $O(\log d)$ time). Thus, each merge operation takes $O(\log d)$ time, and we perform this operation $d-1$ times.

4. Code Generation:

- **Time Complexity:** $O(d)$
- We traverse the Huffman tree to generate the binary codes for each character. This traversal takes $O(d)$ time because we visit each node once.

Combining these steps, the overall time complexity of Huffman Encoding is:

$O(n + d \log d)$

Here, n is the length of the input data, and d is the number of distinct characters.

Space Complexity

1. Frequency Table:

- **Space Complexity:** $O(d)$
- We need to store the frequency of each distinct character.

2. Priority Queue:

- **Space Complexity:** $O(d)$
- The priority queue contains at most d nodes at any time.

3. Huffman Tree:

- **Space Complexity:** $O(d)$
- The Huffman tree contains $2d-1$ nodes in total (d leaf nodes and $d-1$ internal nodes). However, the number of nodes is proportional to d .

4. Code Table:

- **Space Complexity:** $O(d)$
- We need to store the binary code for each character, which is at most d codes.

Combining these components, the overall space complexity of Huffman Encoding is:

$O(d)$

- **Time Complexity:** $O(n + d \log d)$
 - n is the length of the input data.
 - d is the number of distinct characters.
- **Space Complexity:** $O(d)$
 - d is the number of distinct characters.

Huffman Encoding is efficient in both time and space for compressing data, especially when the number of distinct characters (d) is much smaller than the length of the input data (n).