Q1) For each T1 internal node, add a new external node to T2 with the value of the T1 internal node. Swap the value of this new node with its parent (making it an internal node). Reheap up (compare and swap new node with its parent until new node is root or new node’s parent value is greater than new node’s value).

*/
 * The mergingTrees function combines two binary trees (T1 and T2), each maintaining the heap-order property, into a single binary tree T'. It achieves this by ensuring the root of T' is the smaller root of T1 or T2. The function then recursively merges the right subtree of the tree with the smaller root with T2, applying a heapify process to maintain a heap-order property throughout the tree. This process is conducted in a bottom-up manner, affecting only those nodes along the merge path, thus preserving the heap-order without full tree traversal. The process ensures the combined tree T' maintains the heap-order property and aligns with the O(h1 + h2) time complexity goal by focusing only on necessary nodes and avoiding redundant operations.
 */

```
// // Function to merge two binary trees with heap-order property

// Function mergingTrees(T1, T2):
//     // Checking if trees are empty
//     If T1 is empty:
//         Return T2
//     If T2 is empty:
//         Return T1

//     // Confirming that T1 has the smallest root
//     If root(T1) > root(T2):
//         Swap T1 with T2

//     // Merging T2 into T1 while maintaining heap order property using recursive calls
//     // Assigning the smallest root as the root of the merged tree
//     mergedRoot = new Node(root(T1))
//     mergedRoot.leftChild = T1.leftChild
//     mergedRoot.rightChild = mergeWithHeapify(T1.rightChild, T2)

//     Return mergedRoot

// // Function to merge the right child of T1 and T2 with a heapify process
// Function mergeWithHeapify(T1_rightChild, T2):
//     // Perform the merging process ensuring it's bottom-up
//     If T1_rightChild is null:
//         Return heapify(T2)
//     If T2 is null:
//         Return heapify(T1_rightChild)

//     If root(T1_rightChild) > root(T2):
//         Swap T1_rightChild with T2
//     mergedRoot = new Node(root(T1_rightChild))
//     mergedRoot.leftChild = T1_rightChild.leftChild
//     mergedRoot.rightChild = mergeWithHeapify(T1_rightChild.rightChild, T2)
//     mergedRoot = heapify(mergedRoot)

//     Return mergedRoot

// // Function to ensure the heap property in a subtree, bottom-up

// Function heapify(node):
//     If node is null:
//         return null

//     // checking the heap-order property and adjust if necessary
//     If not heapOrdered(node):
//         smallest = findSmallestChild(node)
//         If smallest != null:
//             Swap node with smallest
//             heapify(node) // Continue heapifying at the subtree of the smallest

//     Return node

// // Function to find the smallest child of a node
// Function findSmallestChild(node):
//     smallest = node
//     If node.leftChild is present and node.leftChild < smallest:
//         smallest = node.leftChild
//     If node.rightChild is present and node.rightChild < smallest:
//         smallest = node.rightChild
//     Return smallest

// // Function to check if a node and its descendants maintain heap order
// Function heapOrdered(node):
//     If node is null:
//         return True

//     // Confirm the heap-order property for the left and right children
//     isLeftHeapOrdered = (node.leftChild is null) or (node.value <= node.leftChild.value and
heapOrdered(node.leftChild))
//     isRightHeapOrdered = (node.rightChild is null) or (node.value <= node.rightChild.value
and heapOrdered(node.rightChild))

//     Return isLeftHeapOrdered and isRightHeapOrdered
```

Q2)

Chaining

0	1	2	3	4	5	6	7 8 9 10 11 12 13 14	15	16
23		2	16		17- 34- 5 1	94	44 11 88- 12 13 20	39	

17(in list[5]) and 88(in list[11]) represent the heads linked lists

Linear probing, decreasing

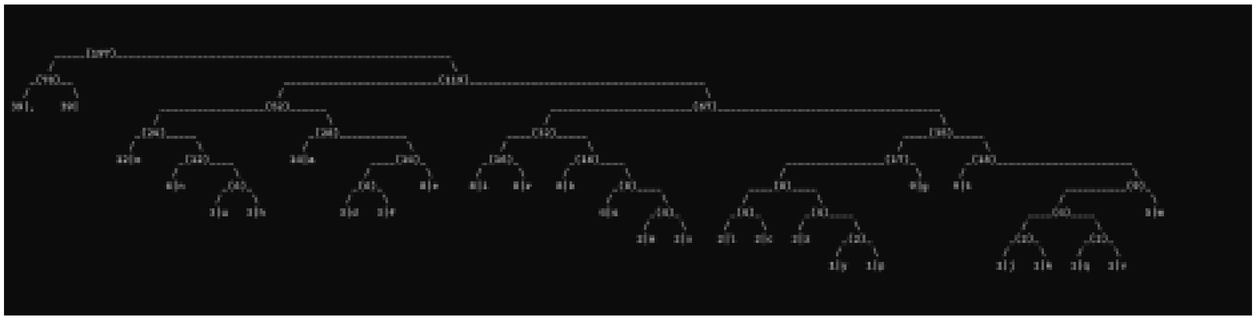
0	1	2	3	4	5	6	7 8 9 10 11 12 13 14	15	16
23	51	2	16	34	17	94	44 20 11 88 12 13	39	

Double hashing

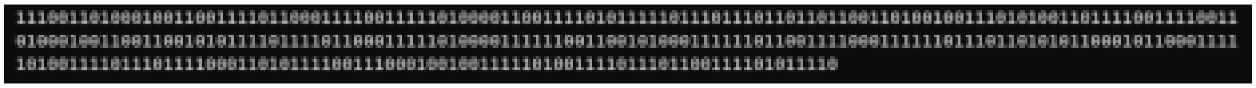
0	1	2	3	4	5	6	7 8 9 10 11 12 13 14	15	16
23		24	51		17	94	34 44 11 88 12 20 13	39	

Q3)
YOU, ME, WE, SHE, HE, SOW, COW, DOG, PIG, RIG, GOLD, SEA, RUG, HAT, CAT, ROW, MOB, LOG, BOX, TAB, BAR, EAR, TAR, JAR, DIG, FAN, BIG, TEA, NOW, FOX, BOG, BAT, BIT, KIT, SIT, ZEN, RAN, FAN, QUIZ, VAN

The huffman tree:

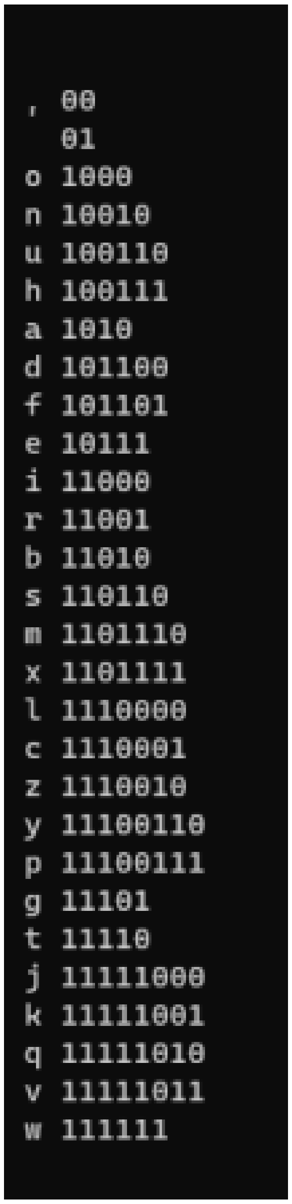


Sentence using dictionary:



```
11100110100010011001111011000111100111110100000110011110101111110111011101101100110100100111010100110111100111110011
0100010011001100101011110111101100011111010000011111100110010100011111101100111100011111101110110101011000101100001111
101001111011101111000110101111001110001001001111101001111011101100111101011110
```

Huffman dictionary:



```
, 00
  01
o 1000
n 10010
u 100110
h 100111
a 1010
d 101100
f 101101
e 10111
i 11000
r 11001
b 11010
s 110110
m 1101110
x 1101111
l 1110000
c 1110001
z 1110010
y 11100110
p 11100111
g 11101
t 11110
j 11111000
k 11111001
q 11111010
v 11111011
w 111111
```