

# Huffman Coding | Greedy Algo-3

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

The variable-length codes assigned to input characters are [Prefix Codes](#), means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

See [this](#) for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

## Algorithm:

The method which is used to construct optimal prefix code is called **Huffman coding**.

This algorithm builds a tree in bottom up manner. We can denote this tree by **T**  
Let,  $|c|$  be number of leaves

$|c| - 1$  are number of operations required to merge the nodes. Q be the priority queue which can be used while constructing binary heap.

*Algorithm Huffman (c)*

{

$n = |c|$

$Q = c$

    for  $i \leftarrow 1$  to  $n-1$

```

do
{

    temp <- get node ()

    left [temp] Get_min (Q) right [temp] Get Min (Q)

    a = left [temp] b = right [temp]

    F [temp] <- f[a] + [b]

    insert (Q, temp)

}

return Get_min (0)
}

```

### ***Steps to build Huffman Tree***

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

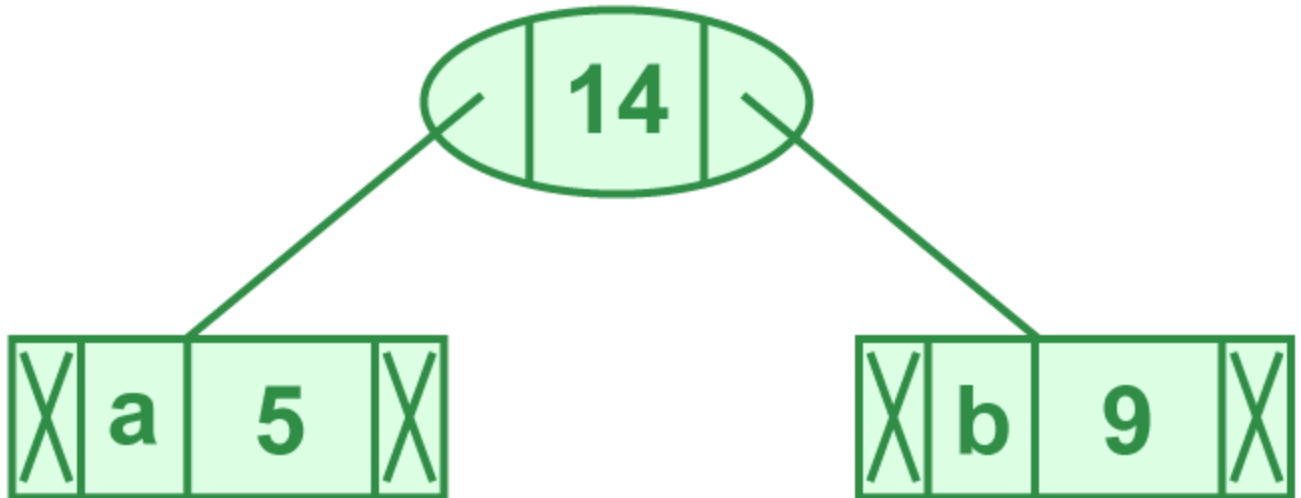
Let us understand the algorithm with an example:

character      Frequency

a	5
b	9
c	12
d	13
e	16
f	45

**Step 1.** Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

**Step 2** Extract two minimum frequency nodes from min heap. Add a new internal node with frequency  $5 + 9 = 14$ .



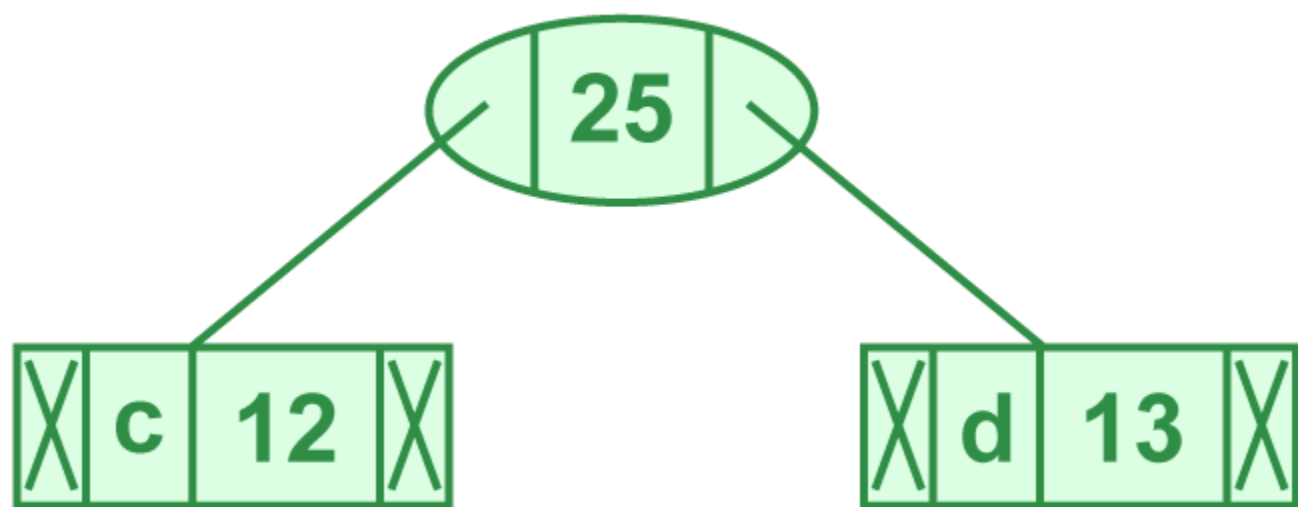
*Illustration of step 2*

Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character                      Frequency

c	12
d	13
Internal Node	14
e	16
f	45

**Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency  $12 + 13 = 25$

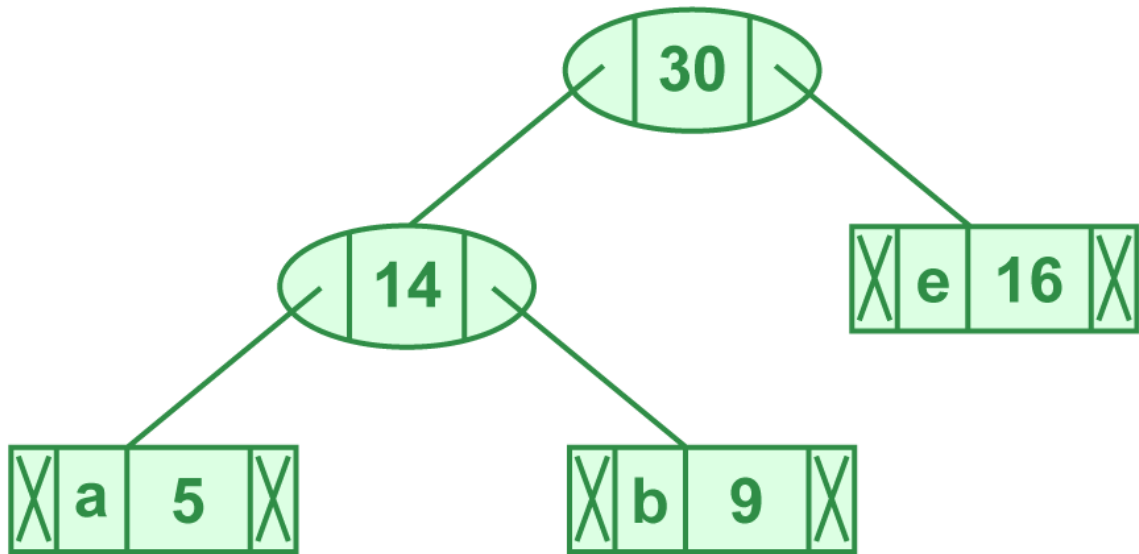


*Illustration of step 3*

Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

**Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency  $14 + 16 = 30$

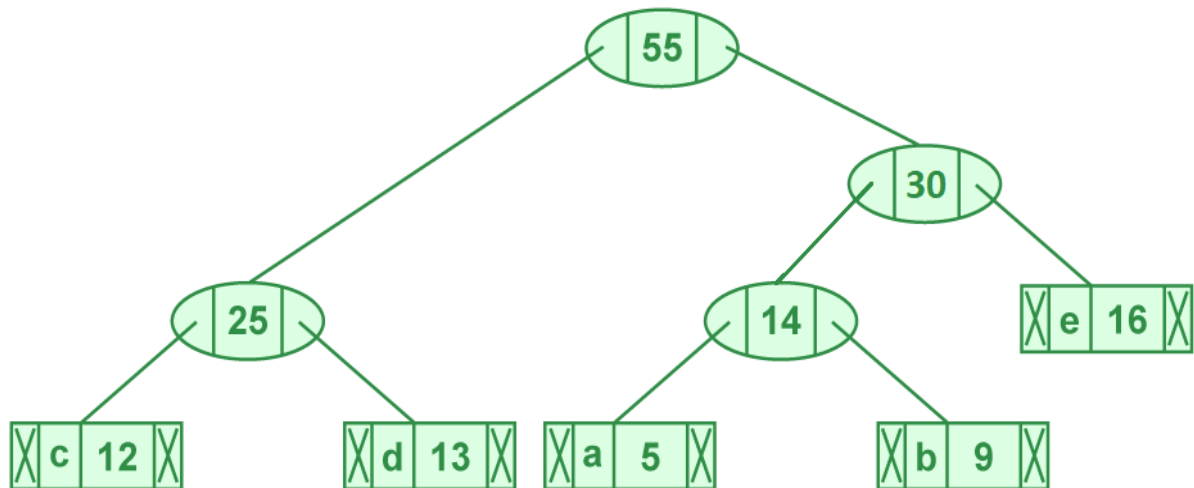


*Illustration of step 4*

Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

**Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency  $25 + 30 = 55$



*Illustration of step 5*

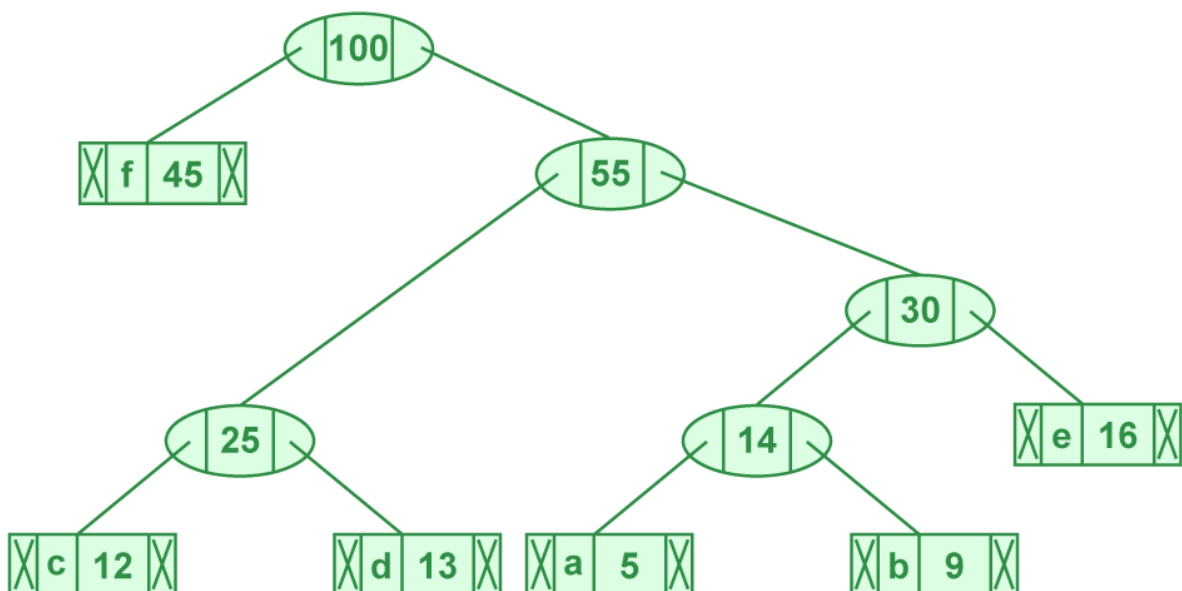
Now min heap contains 2 nodes.

character	Frequency
-----------	-----------

f	45
---	----

Internal Node	55
---------------	----

**Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency  $45 + 55 = 100$



*Illustration of step 6*

Now min heap contains only one node.

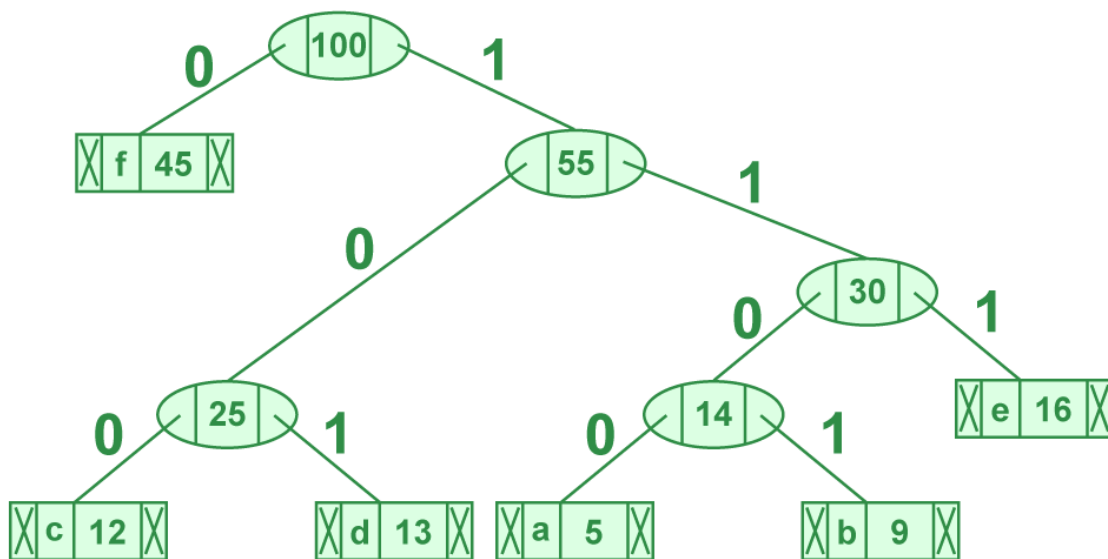
character	Frequency
-----------	-----------

Internal Node	100
---------------	-----

Since the heap contains only one node, the algorithm stops here.

***Steps to print codes from Huffman Tree:***

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



*Steps to print code from HuffmanTree*

The codes are as follows:

character	code-word
-----------	-----------

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

Recommended Problem

## Huffman Encoding

Submission count: 27.7K

Below is the implementation of above approach:

```
// C program for Huffman Coding

#include <stdio.h>

#include <stdlib.h>

// This constant can be avoided by explicitly
// calculating height of Huffman Tree

#define MAX_TREE_HT 100

// A Huffman tree node

struct MinHeapNode {

    // One of the input characters

    char data;
```



```
// Frequency of the character

unsigned freq;


// Left and right child of this node

struct MinHeapNode *left, *right;

};


// A Min Heap:  Collection of

// min-heap (or Huffman tree) nodes

struct MinHeap {

    // Current size of min heap

    unsigned size;


    // capacity of min heap

    unsigned capacity;


    // Array of minheap node pointers
```

```
    struct MinHeapNode** array;

};

// A utility function allocate a new
// min heap node with given character
// and frequency of the character

struct MinHeapNode* newNode(char data, unsigned freq)
{
    struct MinHeapNode* temp = (struct MinHeapNode*)malloc(
        sizeof(struct MinHeapNode));

    temp->left = temp->right = NULL;

    temp->data = data;

    temp->freq = freq;

    return temp;
}
```

```
// A utility function to create
// a min heap of given capacity

struct MinHeap* createMinHeap(unsigned capacity)

{

    struct MinHeap* minHeap

        = (struct MinHeap*)malloc(sizeof(struct MinHeap));

    // current size is 0

    minHeap->size = 0;

    minHeap->capacity = capacity;

    minHeap->array = (struct MinHeapNode**)malloc(

        minHeap->capacity * sizeof(struct MinHeapNode));

    return minHeap;

}
```

```
// A utility function to
// swap two min heap nodes

void swapMinHeapNode(struct MinHeapNode** a,

                     struct MinHeapNode** b)

{

    struct MinHeapNode* t = *a;

    *a = *b;

    *b = t;

}

// The standard minHeapify function.

void minHeapify(struct MinHeap* minHeap, int idx)

{


```

```

int smallest = idx;

int left = 2 * idx + 1;

int right = 2 * idx + 2;


if (left < minHeap->size

    && minHeap->array[left]->freq

        < minHeap->array[smallest]->freq)

    smallest = left;


if (right < minHeap->size

    && minHeap->array[right]->freq

        < minHeap->array[smallest]->freq)

    smallest = right;


if (smallest != idx) {

    swapMinHeapNode(&minHeap->array[smallest],

                    &minHeap->array[idx]);

    minHeapify(minHeap, smallest);

}

```

```
}

// A utility function to check
// if size of heap is 1 or not

int isSizeOne(struct MinHeap* minHeap)

{

    return (minHeap->size == 1);

}

// A standard function to extract
// minimum value node from heap

struct MinHeapNode* extractMin(struct MinHeap* minHeap)

{

    struct MinHeapNode* temp = minHeap->array[0];

    minHeap->array[0] = minHeap->array[minHeap->size - 1];
```

```

        --minHeap->size;

        minHeapify(minHeap, 0);

        return temp;
    }

// A utility function to insert
// a new node to Min Heap

void insertMinHeap(struct MinHeap* minHeap,

                   struct MinHeapNode* minHeapNode)

{

    ++minHeap->size;

    int i = minHeap->size - 1;

    while (i

```

```

        && minHeapNode->freq

        < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;
}

// A standard function to build min heap

void buildMinHeap(struct MinHeap* minHeap)

{

int n = minHeap->size - 1;

int i;

```



```

        for (i = (n - 1) / 2; i >= 0; --i)

            minHeapify(minHeap, i);

    }


// A utility function to print an array of size n

void printArr(int arr[], int n)

{

    int i;

    for (i = 0; i < n; ++i)

        printf("%d", arr[i]);

    printf("\n");

}


// Utility function to check if this node is leaf

int isLeaf(struct MinHeapNode* root)

{


```

```

        return !(root->left) && !(root->right);
    }

// Creates a min heap of capacity
// equal to size and inserts all character of
// data[] in min heap. Initially size of
// min heap is equal to capacity

struct MinHeap* createAndBuildMinHeap(char data[],

                                     int freq[], int size)

{

    struct MinHeap* minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)

        minHeap->array[i] = newNode(data[i], freq[i]);

```

```

    minHeap->size = size;

    buildMinHeap(minHeap);

    return minHeap;
}

// The main function that builds Huffman tree

struct MinHeapNode* buildHuffmanTree(char data[],

                                     int freq[], int size)

{

    struct MinHeapNode *left, *right, *top;

    // Step 1: Create a min heap of capacity

    // equal to size. Initially, there are

    // nodes equal to size.

    struct MinHeap* minHeap

        = createAndBuildMinHeap(data, freq, size);

```

```
// Iterate while size of heap doesn't become 1

while (!isSizeOne(minHeap)) {

    // Step 2: Extract the two minimum

    // freq items from min heap

    left = extractMin(minHeap);

    right = extractMin(minHeap);

    // Step 3: Create a new internal

    // node with frequency equal to the

    // sum of the two nodes frequencies.

    // Make the two extracted node as

    // left and right children of this new node.

    // Add this node to the min heap

    // '$' is a special value for internal nodes, not

    // used

    top = newNode('$', left->freq + right->freq);
```

```

        top->left = left;

        top->right = right;

        insertMinHeap(minHeap, top);

    }

    // Step 4: The remaining node is the

    // root node and the tree is complete.

    return extractMin(minHeap);

}

// Prints huffman codes from the root of Huffman Tree.

// It uses arr[] to store codes

void printCodes(struct MinHeapNode* root, int arr[],

               int top)

{

```

```
// Assign 0 to left edge and recur

if (root->left) {

    arr[top] = 0;

    printCodes(root->left, arr, top + 1);

}

// Assign 1 to right edge and recur

if (root->right) {

    arr[top] = 1;

    printCodes(root->right, arr, top + 1);

}

// If this is a leaf node, then

// it contains one of the input

// characters, print the character
```

```

        // and its code from arr[]

        if (isLeaf(root)) {

            printf("%c: ", root->data);

            printArr(arr, top);

        }
    }

// The main function that builds a
// Huffman Tree and print codes by traversing
// the built Huffman Tree

void HuffmanCodes(char data[], int freq[], int size)

{

    // Construct Huffman Tree

    struct MinHeapNode* root

        = buildHuffmanTree(data, freq, size);

```

```
// Print Huffman codes using

// the Huffman tree built above

int arr[MAX_TREE_HT], top = 0;


printCodes(root, arr, top);

}


// Driver code

int main()

{

    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

    int freq[] = { 5, 9, 12, 13, 16, 45 };


    int size = sizeof(arr) / sizeof(arr[0]);


    HuffmanCodes(arr, freq, size);
```



```
    return 0;  
  
}
```

## Output

f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111

**Time complexity:**  $O(n \log n)$  where  $n$  is the number of unique characters. If there are  $n$  nodes, `extractMin()` is called  $2*(n - 1)$  times. `extractMin()` takes  $O(\log n)$  time as it calls `minHeapify()`. So, the overall complexity is  $O(n \log n)$ . If the input array is sorted, there exists a linear time algorithm. We will soon be discussing this in our next post.

**Space complexity :-  $O(N)$**

### Applications of Huffman Coding:

1. They are used for transmitting fax and text.
2. They are used by conventional compression formats like PKZIP, GZIP, etc.
3. Multimedia codecs like JPEG, PNG, and MP3 use Huffman encoding (to be more precise the prefix codes).

It is useful in cases where there is a series of frequently occurring characters.