Huffman Codes | Greedy Algorithm

We are going to use Binary Tree (https://www.codesdope.com/blog/article/binary-trees/) and Minimum Priority Queue (https://www.codesdope.com/blog/article/priority-queue-using-heap/) in this chapter. You can learn these from the linked articles if you are not familiar with these.

Huffman code is a data compression algorithm which uses the greedy technique (/course/algorithms-greedy-algorithm/) for its implementation. The algorithm is based on the frequency of the characters appearing in a file.

We know that our files are stored as binary code in a computer and each character of the file is assigned a binary character code and normally, these character codes are of fixed length for different characters. For example, if we assign 'a' as 000 and 'b' as 001, the length of the codeword for both the characters are fixed i.e., both 'a' and 'b' are taking 3 bits.

Character	a	b	c
Code	000	001	010
Length	3	3	3

Huffman code doesn't use fixed length codeword for each character and assigns codewords according to the frequency of the character appearing in the file. Huffman code assigns a shorter length codeword for a character which is used more number of time (or has a high frequency) and a longer length codeword for a character which is used less number of times (or has a less frequency).

High Frequency Low Frequency

Character	a	b	c
Variable Length Code	000	101	1101
Length	1	3	4

Since characters which have high frequency has lower length, they take less space and save the space required to store the file. Let's take an example.

Character	a	b	c	d	e	f
Fixed Length Code	000	001	010	011	100	101
Frequency	51	20	2	3	9	15
Variable Length Code	0	111	1100	1101	100	101

In this example, 'a' is appearing 51 out of 100 times and has the highest frequency, 'c' is appearing only 2 out of 100 times and has the least frequency. Thus, we are assigning 'a' the codeword of the shortest length i.e., 0 and 'c' a longer one i.e., 1100.

Now if we use characters of fixed length, we need 100*3 = 300 bits (each character is taking 3 bit) to represent 100 characters of the file. But to represent 100 characters with the variable length character, we need 51*1 + 20*3 + 2*4 + 3*4 + 9*3 + 15*3 = 203 bits (51*1 as 'a' is appearing 51 out of 100 times and has length 1 and so on). Thus, we can save 32% of space by using the codeword for variable length in this case.

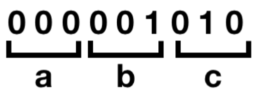
Also, the Huffman code is a lossless compression technique. You can see that we are not losing any information, we are just using a different way to represent each character.

One general doubt which comes here while reading this is "why we are using longer length codeword for characters which have less frequency, can't we use a shorter length for them too? E.g.- couldn't we use code of 3-bit long code to represent 'c' instead of 4?".

The answer is no and this is due to the prefix code. We are going to discuss prefix code in a while but let's first discuss how storing, encoding and decoding is done.

Storing, Encoding and Decoding

We basically concatenate characters while storing them into a file. For example, to store 'abc', we would use 000.001.010 i.e., 000001010 using a fixed character codeword. Now for the decoding, we know that all of our characters are 3 bits long, so we would break the code for every 3 bits and we can easily get 000, 001 and 010 which can be translated back to 'abc'.



Now, we can't use any specific length which can separate our character if we are using variable length codeword and to simplify decoding the codeword back to characters, we use prefix codes.

Prefix Codes

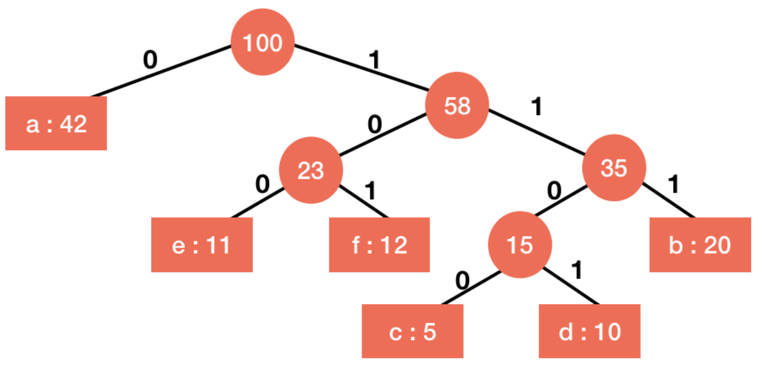
In variable length codeword, we only use such code which are not the prefix of any other character and these codes is known as **prefix codes**. For example, if we use 0, 1, 01, 10 to represent 'a', 'b', 'c' and 'd' respectively (0 is prefix of 01 and 1 is prefix of 10), then the code 00101 can be translated into 'aabab', 'acc', 'aadb', 'acab' or 'aabc'. To avoid this kind of ambiguity, we use prefix codes.

Using prefix codes made decoding unambiguous. In the above example, we have used 0, 111 and 1100 for 'a', 'b' and 'c' respectively. None of the code is the prefix of any other codes and thus any combination of these codes will decode into unique value. For example, if we write 01111100, it will uniquely decode into 'abc' only. Give it a try and try to decode it into something else.

Now, we know what is Huffman code and how it works. Let's now focus on how to use it.

Implementing Huffman Code

Let's first look at the binary tree given below.



We construct this type of binary tree from the frequencies of the characters given to us and we will learn how to do this in a while. Let's focus on using this binary tree first and then we will learn about its construction.

Let's first focus on the decoding process.

Decoding

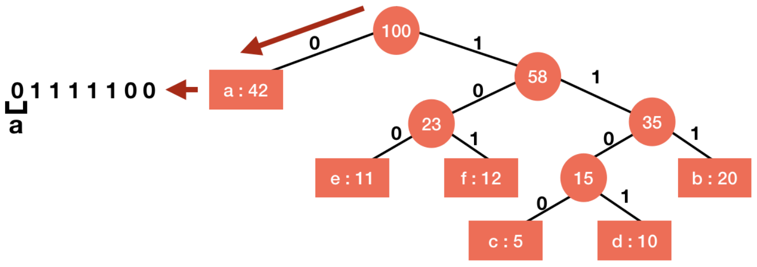
One thing to notice here is that all the characters are on the leaves of the tree and to get the codeword for any character, we start from the root of the tree and proceed to that character. Now, if we move right from any node, we interpret that movement as 1 and if left, then 0. This is also indicated on the branch of the binary tree in the picture given above. So, we move from root to the leaf containing that character and combining the 0s and 1s of each movement, we get the codeword of the character. This is described in the picture given below.

In this way, we can get the code of each character.

Character	Frequency	Code
a	12	0
b	20	111
c	5	1100
d	10	1101
e	11	100
f	12	101

We can proceed in a similar way with any code given to us to decode it. For example, let's take a case of 01111100.

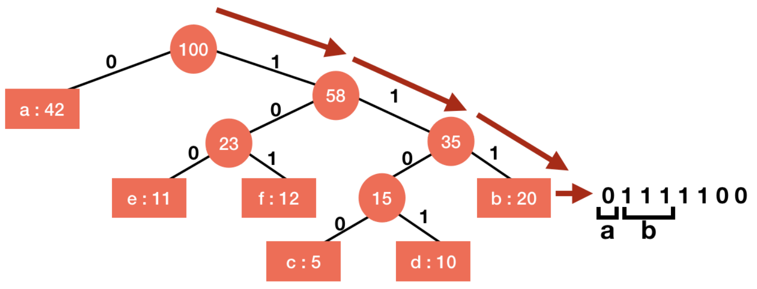
We will start from the root and since the first number is 0, so we will move left. By moving left, we encountered a character 'a' and thus the first character is a.



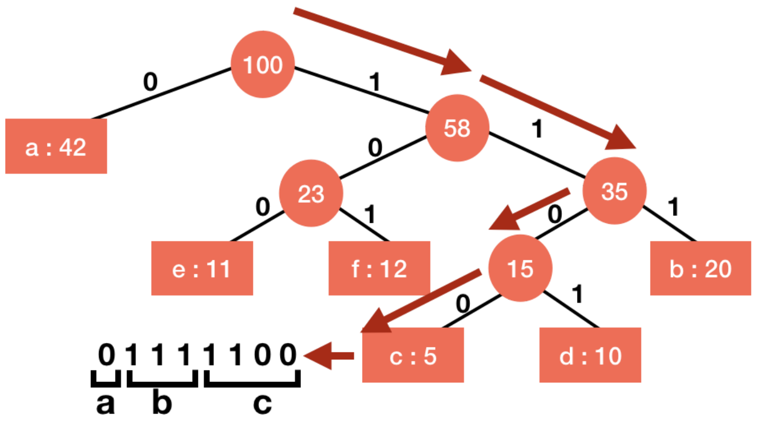
Now, we will again start from the root, we have 1 in the code, so we will move right.

Since we have not reached any of the leaves, we will continue it. The next number is also 1 and so we will again move right.

Still, we have not reached the leaf, so continuing, the next number is also 1. Moving right this time will give us the character 'b'. Thus, 'ab' is the string we have decoded till now.



Similarly, again starting from the root and moving for the next numbers will make us reach 'c'.



Thus, we have decoded 01111100 into 'abc'.

We now know how to decode for Huffman code. Let's look at the encoding process now.

Encoding

As stated above, encoding is simple. We just have to concatenate the code of the characters to encode them. For example, to encode 'abc', we will just concatenate 0, 111 and 1100 i.e., 01111100.

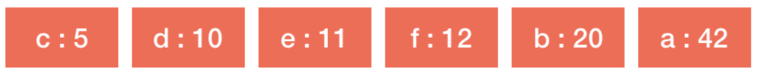
Now, our next task is to create this binary tree for the frequencies we have.

Construction of Binary Tree for Huffman Code

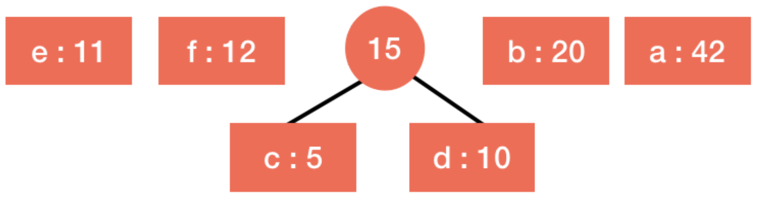
This is the part where we use the greedy strategy. Basically, we have to assign shorter code to the character with higher frequency and vice-versa. We can do this in different ways and that will result in different trees, but a full binary tree (a tree in which every node has 2 children, except the leaves) gives us the optimal code i.e., using that code will save the maximum space in storing the file.

To construct this tree for optimal prefix code, Huffman invented a greedy algorithm which we are going to use for the construction of the tree.

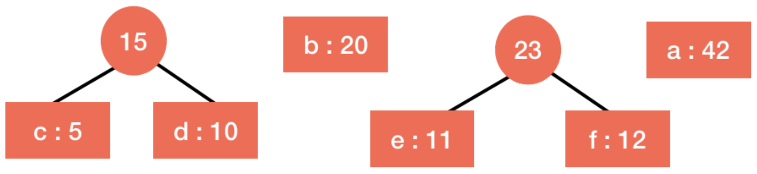
We start by sorting the characters according to their frequencies.

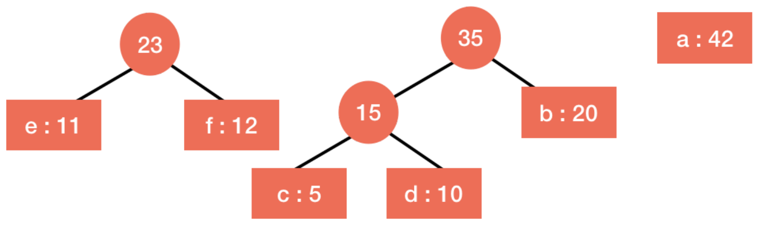


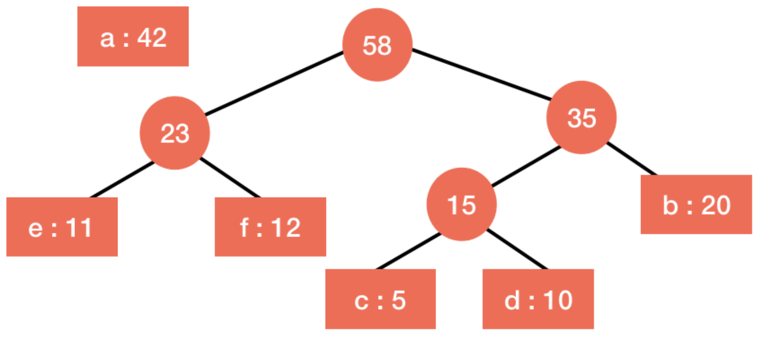
Now, we make a new node and then greedily pick the first two nodes from the sorted character and make the first node left child of the new node and second node as the right child of the new node. The value of the new node is the summation of the values of the children nodes.

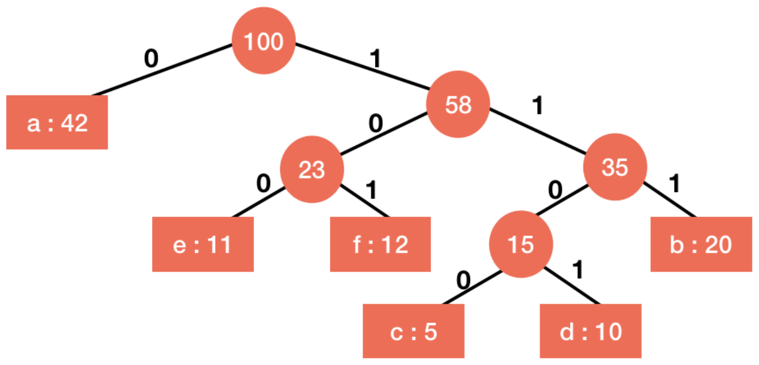


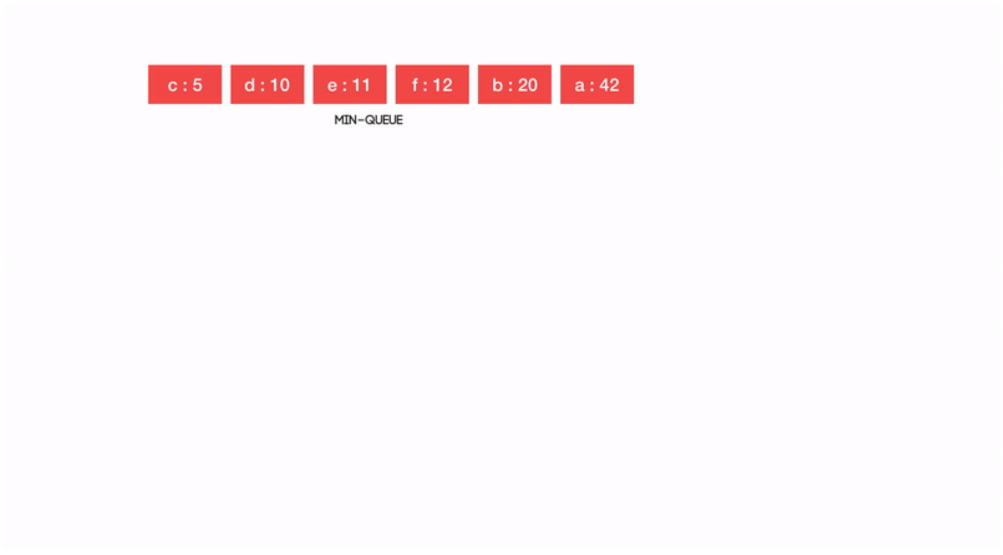
We again sort the nodes according to the values and repeat the same process with the first two nodes.





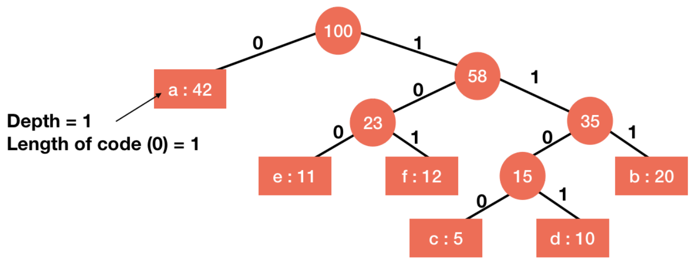






In this way, we construct the tree for optimal prefix code.

We can see that the depth of a leaf is also the length of the prefix code of the character. So, the depth d_i of leaf i is the length of the prefix code of the character at leaf i. If we multiply it by its frequency i.e., $d_i * i.freq$, then this is the number of bits used by this character in the entire file.



We can sum all these for each character to get the total number of bits used to store the file.

$$\text{Total bits} = \sum_i d_i * i.\mathit{freq}$$

This can also be used to prove why full binary tree gives us optimal codes because this summation of depths will be least in a full binary tree.

Now, we know how to construct the tree from their frequencies and then use that tree to know the prefix codes of characters and how to encode and decode. So, let's see the coding implementation for the construction of the tree.

Code for Huffman Code

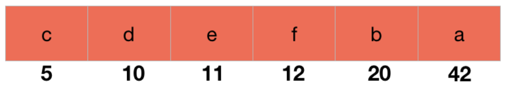
In the steps of constructing the tree, we can see that we are continuously using the sorted nodes and using the first two elements from it. This feature will be well taken care of if we use minimum priority queue because just by inserting a new node, it will go to a position where the nodes will be in the sorted order and also extracting from a minimum priority queue will give us the nodes with least values first.

We need the frequencies of the characters to make the tree, so we will start making our function by passing the array of the nodes containing the characters to it - `GREEDY-HUFFMAN-CODE(C)`, C is the array which has the characters stored in nodes.

Our next task is to make a minimum priority queue using these nodes.

```
min_queue.build(C)
```

Minimum priority queue will automatically add these nodes in a position such that they are in the sorted order.



c	d	e	f	b	a
5	10	11	12	20	42

Now, we need to extract the first element from the queue and set it as the left child of a new node.

```
n = min_queue.length
z = new node
z.left = min_queue.extract()
```

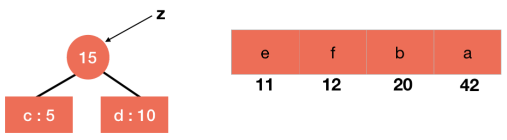
Similarly, we need to set the right child of this new node to the element we will extract next.

```
z.right = min_queue.extract()
```

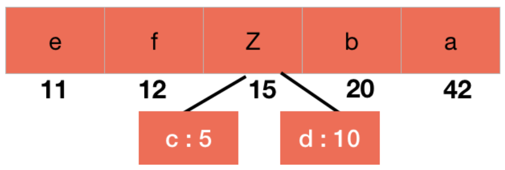
Now, the value of this new node will be the summation of the values of its children.

```
z.freq = z.left.freq + z.right.freq
```

After this, our queue and the node z will like this:

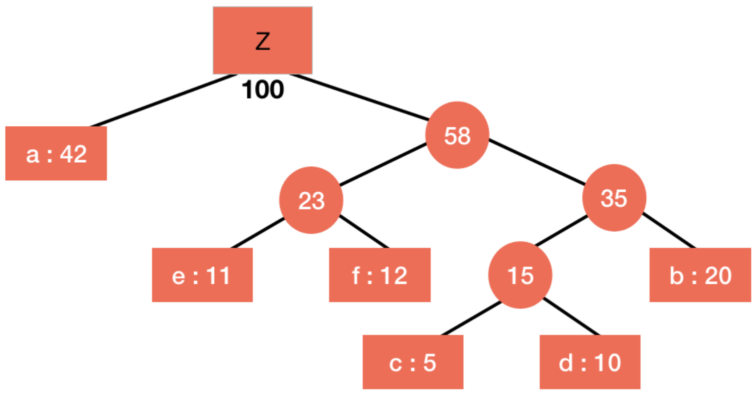


Our next task is to insert this node in the queue. Since the queue is a minimum priority queue, it will add the node to a position where the queue will remain sorted.



```
min_queue.insert(z)
```

We need to repeat this process until the length of min_queue becomes 1 i.e., only the root is stored in the queue.



Also, we are extracting two nodes and adding one node to the queue in each step, thus making an iteration from 1 to n-1 to achieve the same.

```
while min_queue.length > 1
   z.left = min_queue.extract()
   z.right = min_queue.extract()
   ...
```

At last, we will extract the root of the tree from the queue and return it.

```
while min_queue.length > 1
   ...
return min_queue.extract()
```

```
GREEDY-HUFFMAN-CODE(C)
  min_queue.build(C)

  while min_queue.length > 1
    z = new node
    z.left = min_queue.extract()
    z.right = min_queue.extract()
    z.freq = z.left.freq + z.right.freq
    min_queue.insert(z)

  return min_queue.extract()
```

C **Python** **Java**

```
#include <stdio.h>
#include <stdlib.h>

// node
/*
           _____
          | data|
          | freq|
          |_____|
left ch./        \right chlid address
  _____/          \ _____
 | data|           | data|
 | freq|           | freq|
 |_____|           |_____|
*/
typedef struct node
{
  int frequency;
  char data;
  struct node *left;
  struct node *right;
}node;

int heap_array_size = 100; // size of array storing heap
int heap_size = 0;
const int INF = 100000;

//function to swap nodes
void swap( node *a, node *b ) {
  node t;
  t = *a;
  *a = *b;
  *b = t;
}

/*
  function to print tree
  https://www.codesdope.com/blog/article/binary-tree-in-c-linked-representation-traversals/
*/
void inorder(struct node *root)
{
  if(root!=NULL) // checking if the root is not null
  {
    inorder(root->left); // visiting left child
    printf(" %d ", root->frequency); // printing data at root
    inorder(root->right);// visiting right child
  }
}

/*
  function for new node
*/
node* new_node(char data, int freq) {
  node *p;
  p = malloc(sizeof(struct node));
  p->data = data;
  p->frequency = freq;
  p->left = NULL;
  p->right = NULL;

  return p;
}

//function to get right child of a node of a tree
int get_right_child(int index) {
  if((((2*index)+1) <= heap_size) && (index >= 1))
    return (2*index)+1;
  return -1;
}

//function to get left child of a node of a tree
int get_left_child(int index) {
    if(((2*index) <= heap_size) && (index >= 1))
        return 2*index;
    return -1;
}
```

```
//function to get the parent of a node of a tree
int get_parent(int index) {
  if ((index > 1) && (index <= heap_size)) {
    return index/2;
  }
  return -1;
}

/*
  Functions taken from minimum priority queue
  https://www.codesdope.com/blog/article/priority-queue-using-heap/
  https://www.codesdope.com/blog/article/heap-binary-heap/
*/

void insert(node A[], node* a, int key) {
  heap_size++;
  A[heap_size] = *a;
  int index = heap_size;
  while((index>1) && (A[get_parent(index)].frequency > a->frequency)) {
    swap(&A[index], &A[get_parent(index)]);
    index = get_parent(index);
  }
}

node* build_queue(node c[], int size) {
  node* a = malloc(sizeof(node)*heap_array_size); // a is the array to store heap
  int i;
  for(i=0; i<size; i++) {
    insert(a, &c[i], c[i].frequency); // inserting node in array a(min-queue)
  }
  return a;
}

void min_heapify(node A[], int index) {
  int left_child_index = get_left_child(index);
  int right_child_index = get_right_child(index);

  // finding smallest among index, left child and right child
  int smallest = index;

  if ((left_child_index <= heap_size) && (left_child_index>0)) {
    if (A[left_child_index].frequency < A[smallest].frequency) {
      smallest = left_child_index;
    }
  }

  if ((right_child_index <= heap_size && (right_child_index>0))) {
    if (A[right_child_index].frequency < A[smallest].frequency) {
      smallest = right_child_index;
    }
  }

  // smallest is not the node, node is not a heap
  if (smallest != index) {
    swap(&A[index], &A[smallest]);
    min_heapify(A, smallest);
  }
}

node* extract_min(node A[]) {
  node minm = A[1];
  A[1] = A[heap_size];
  heap_size--;
  min_heapify(A, 1);
  node *z;
  // copying minimum element
  z = malloc(sizeof(struct node));
  z->data = minm.data;
  z->frequency = minm.frequency;
  z->left = minm.left;
  z->right = minm.right;
  return z; //returning minimum element
}

// Huffman code
node* greedy_huffman_code(node C[]) {
  node *min_queue = build_queue(C, 6); // making min-queue
```

```
    while(heap_size > 1) {
      node *h = extract_min(min_queue);
      node *i = extract_min(min_queue);
      node *z;
      z = malloc(sizeof(node));
      z->data = '\0';
      z->left = h;
      z->right = i;
      z->frequency = z->left->frequency + z->right->frequency;
      insert(min_queue, z, z->frequency);
    }
    return extract_min(min_queue);
  }

  int main() {
    node *a = new_node('a', 42);
    node *b = new_node('b', 20);
    node *c = new_node('c', 5);
    node *d = new_node('d', 10);
    node *e = new_node('e', 11);
    node *f = new_node('f', 12);
    node C[] = {*a, *b, *c, *d, *e , *f};

    node* z = greedy_huffman_code(C);
    inorder(z); //printing tree
    printf("\n");

    return 0;
  }
```

Analysis of Huffman Code

By using the min-heap to implement the queue, we can perform each operation of the queue in $O(\lg n)$ time. Also, the initialization of the queue is going to take $O(n)$ time i.e., time for building a heap.

Now, we know that each operation is taking $O(\lg n)$ time and we have already discussed that there are a total of n-1 iterations. Thus, the total running time of the algorithm will be $O(n \lg n)$.

With this, we are going to finish the section of greedy algorithms. Next, we are going to learn some graph algorithms.

“ Today is cruel. Tomorrow is crueller. And the day after tomorrow is beautiful. ”

- Jack Ma

PREV (/course/algorithms-egyptian-fraction/) (/course/algorithms-graphs/) NEXT

# Further Readings

- C++ : Linked lists in C++ (Singly linked list) (/blog/article/c-linked-lists-in-c-singly-linked-list/)
- Trees in Computer Science (/blog/article/trees-in-computer-science/)
- Binary Trees (/blog/article/binary-trees/)
- Binary Trees in C : Array Representation and Traversals (/blog/article/binary-trees-in-c-array-representation-and-travers/)
- Binary Tree in C: Linked Representation & Traversals (/blog/article/binary-tree-in-c-linked-representation-traversals/)