Huffman compression

By Ori Sharim

What is huffman compression

Huffman compression is a lossless compression algorithm.

Lossless compression means that i can compress a file(making a much smaller file to store the original file's data) without losing any data when i reconstruct the original file again from the new smaller file.

01

algorithm

02

implementation

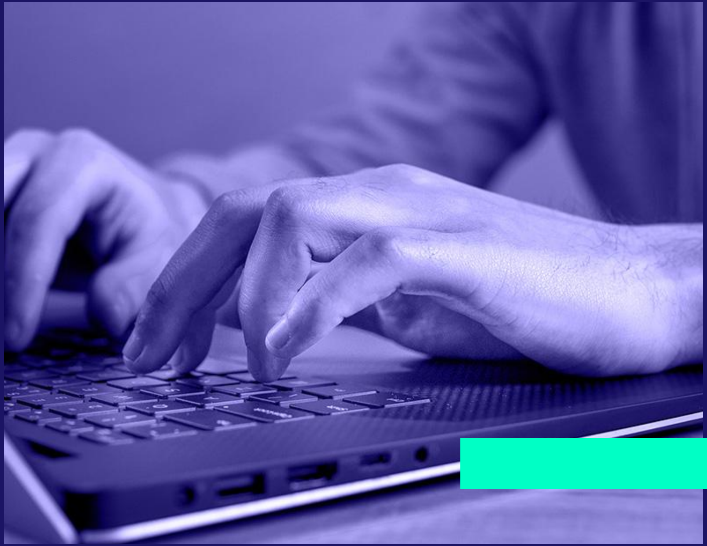
03

executing the code

algorithm

algorithm

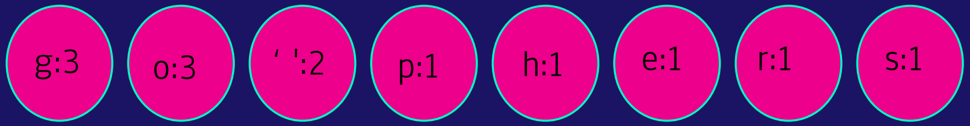
The algorithm itself builds a binary tree using the frequencies of the different characters in the file to represent new smaller codes for each of the characters than their ascii values.



How it works

We start by counting the frequencies of the different characters of the text in a file - “go pogos herg”(for example)

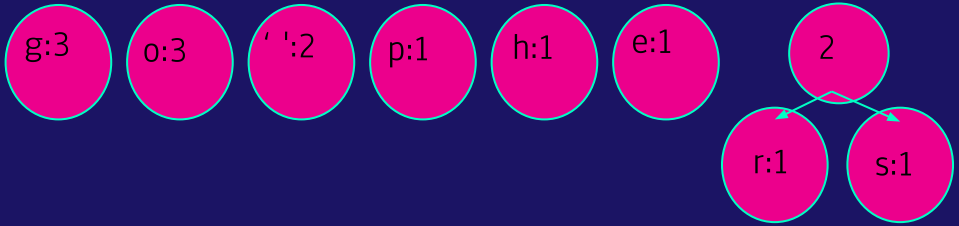
So we have:



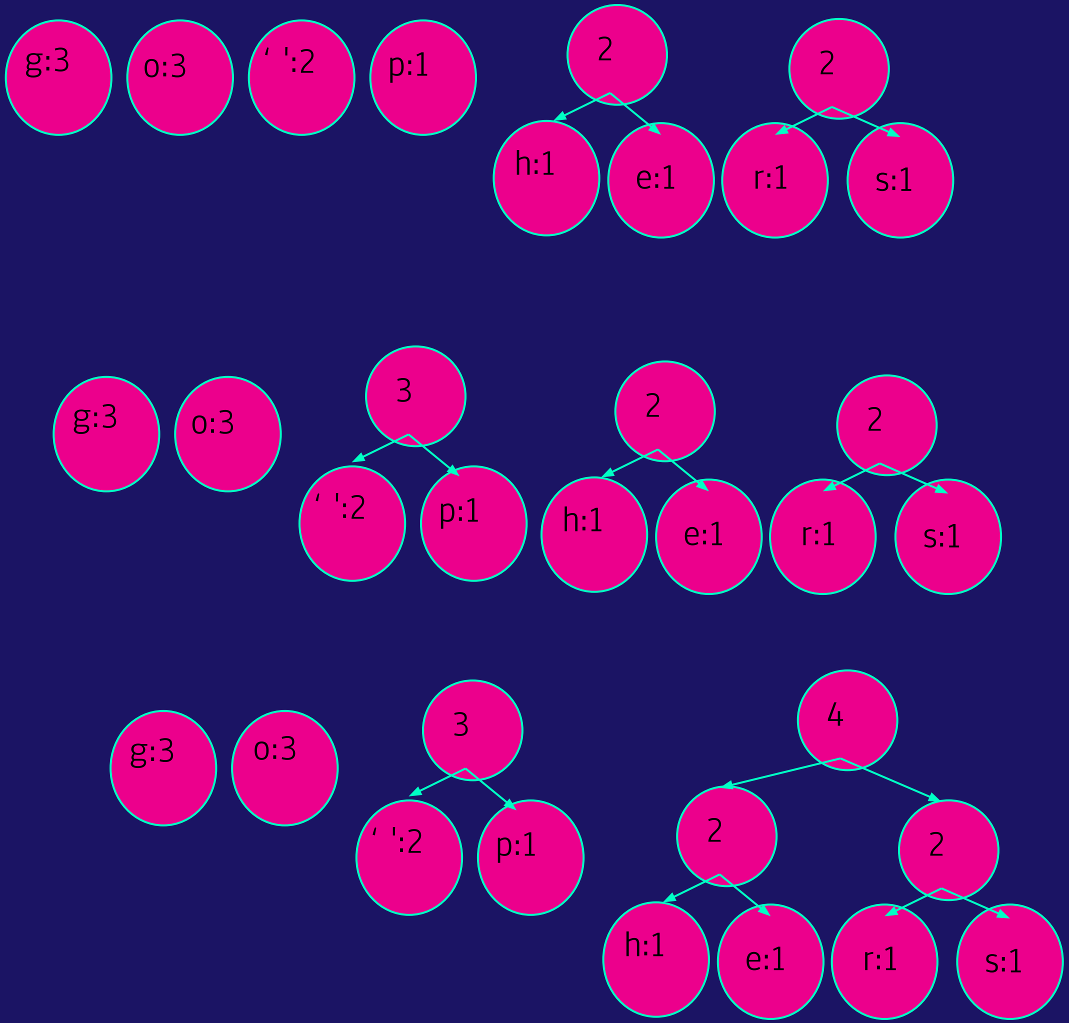
g:3 o:3 ‘ ':2 p:1 h:1 e:1 r:1 s:1

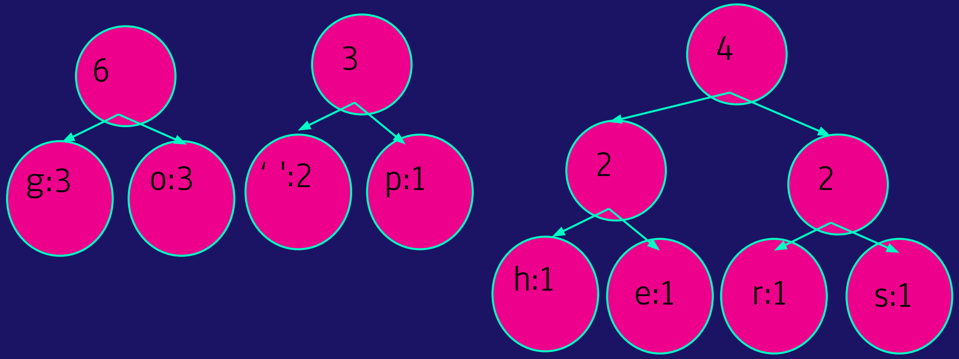
Building the tree

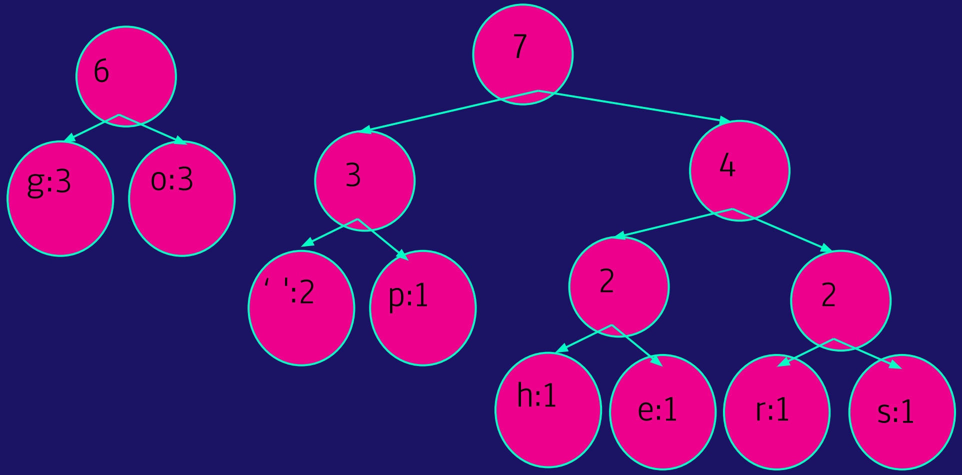
Now we take 2 of the nodes with the lowest amount of frequencies and create sub binary tree with them:



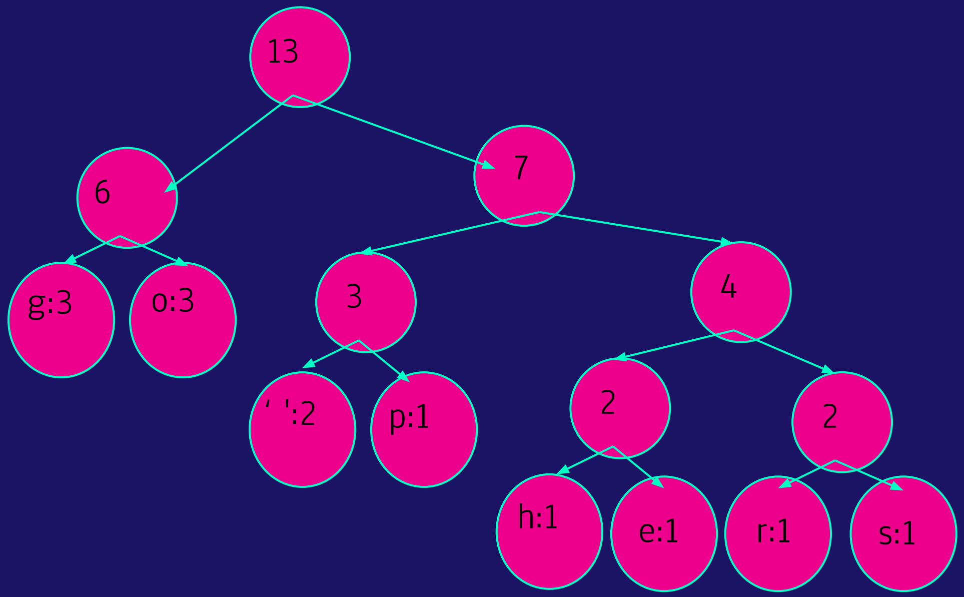
Now we repeat this process until we are left with one binary tree containing all of the characters and their frequencies.


g:3
o:3
‘ ':2
p:1
2
h:1
e:1
2
r:1
s:1
g:3
o:3
3
‘ ':2
p:1
2
h:1
e:1
2
r:1
s:1
g:3
o:3
3
‘ ':2
p:1
4
2
h:1
e:1
2
r:1
s:1


6
g:3
o:3
3
‘ ':2
p:1
4
2
2
h:1
e:1
r:1
s:1


6
g:3
o:3
7
3
4
‘ ':2
p:1
2
2
h:1
e:1
r:1
s:1

Final tree

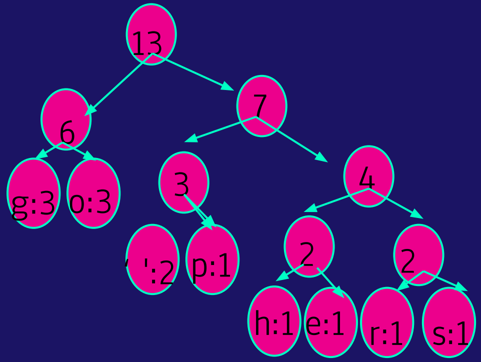


Codes

Now using the route to each character we can assign a code to each character.
Every time we go right we put 1 and when we go left we put 0.

Codes:
g = 00
o = 01
‘ ‘ = 100
p = 101
h = 1100
e = 1101
r = 1110
s = 1111

Notice how all of these new codes are much shorter than the original ascii value of each of the characters, now we can use these new shorter codes for each character and create a new file to save the data of the original file that uses less space.
But before writing the all of the codes instead of the characters to the new file, we need a way to find out how the binary tree looks like when we are decompressing so we will know which code is assigned to which character.
To do that we add the different characters and their frequencies so we can use it to build the tree using the same function we used when we compressed the file,



implementation

Data structures

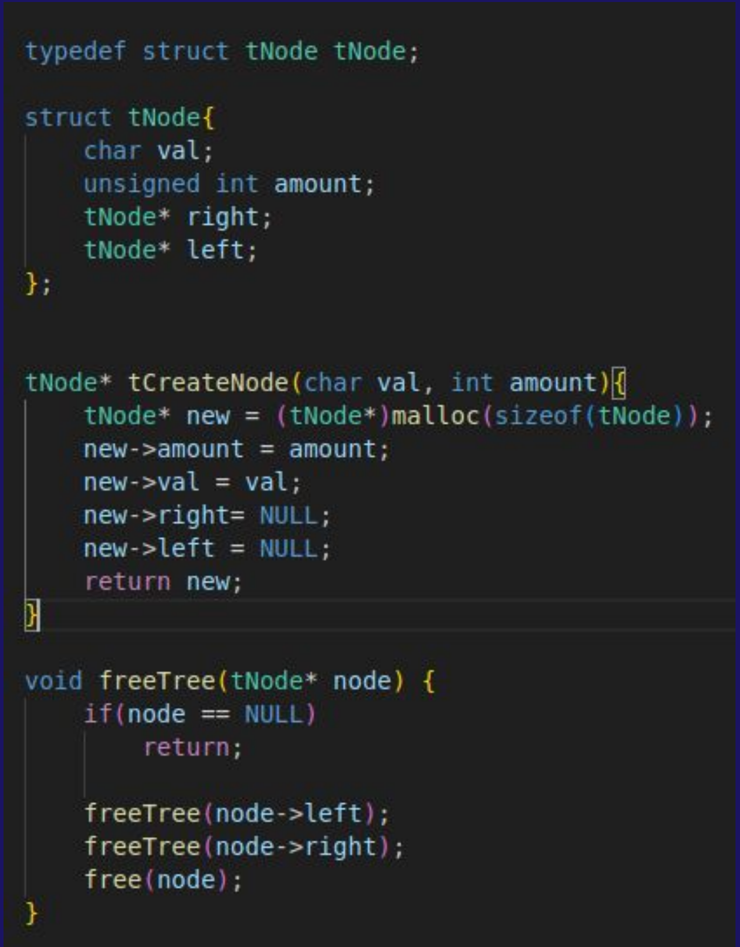
Queue

I used a modified version of a queue data structure.
I saved it in the memory like a list but with push and pop functions to add an element at the end and to get the first element

Binary tree

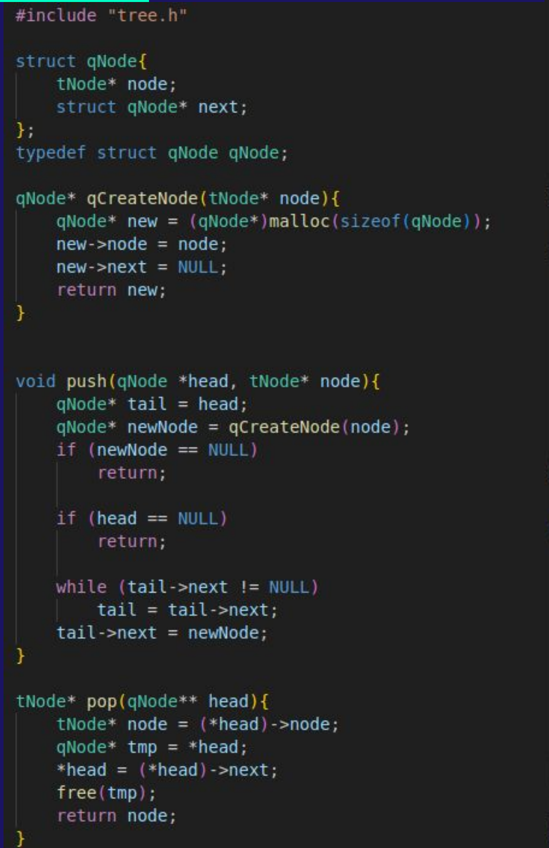
Because we need a binary tree to get the codes.

Binary tree

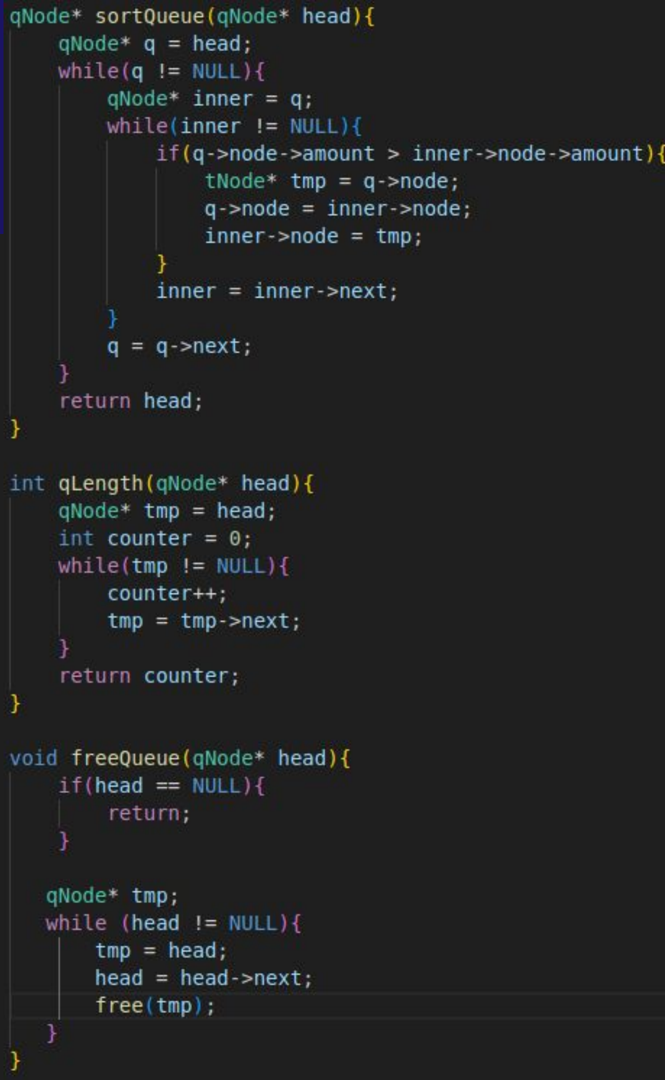


```c
typedef struct tNode tNode;

struct tNode{
    char val;
    unsigned int amount;
    tNode* right;
    tNode* left;
};


tNode* tCreateNode(char val, int amount){
    tNode* new = (tNode*)malloc(sizeof(tNode));
    new->amount = amount;
    new->val = val;
    new->right= NULL;
    new->left = NULL;
    return new;
}

void freeTree(tNode* node) {
    if(node == NULL)
        return;

    freeTree(node->left);
    freeTree(node->right);
    free(node);
}
```

Queue

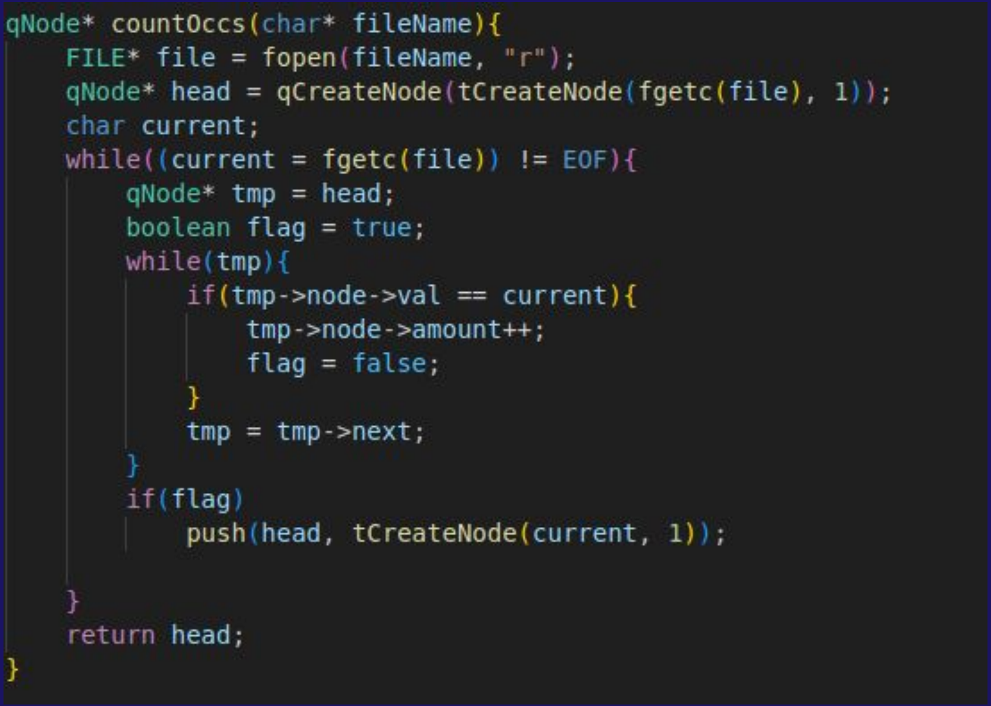


```c
#include "tree.h"

struct qNode{
    tNode* node;
    struct qNode* next;
};
typedef struct qNode qNode;

qNode* qCreateNode(tNode* node){
    qNode* new = (qNode*)malloc(sizeof(qNode));
    new->node = node;
    new->next = NULL;
    return new;
}


void push(qNode *head, tNode* node){
    qNode* tail = head;
    qNode* newNode = qCreateNode(node);
    if (newNode == NULL)
        return;

    if (head == NULL)
        return;

    while (tail->next != NULL)
        tail = tail->next;
    tail->next = newNode;
}

tNode* pop(qNode** head){
    tNode* node = (*head)->node;
    qNode* tmp = *head;
    *head = (*head)->next;
    free(tmp);
    return node;
}
```



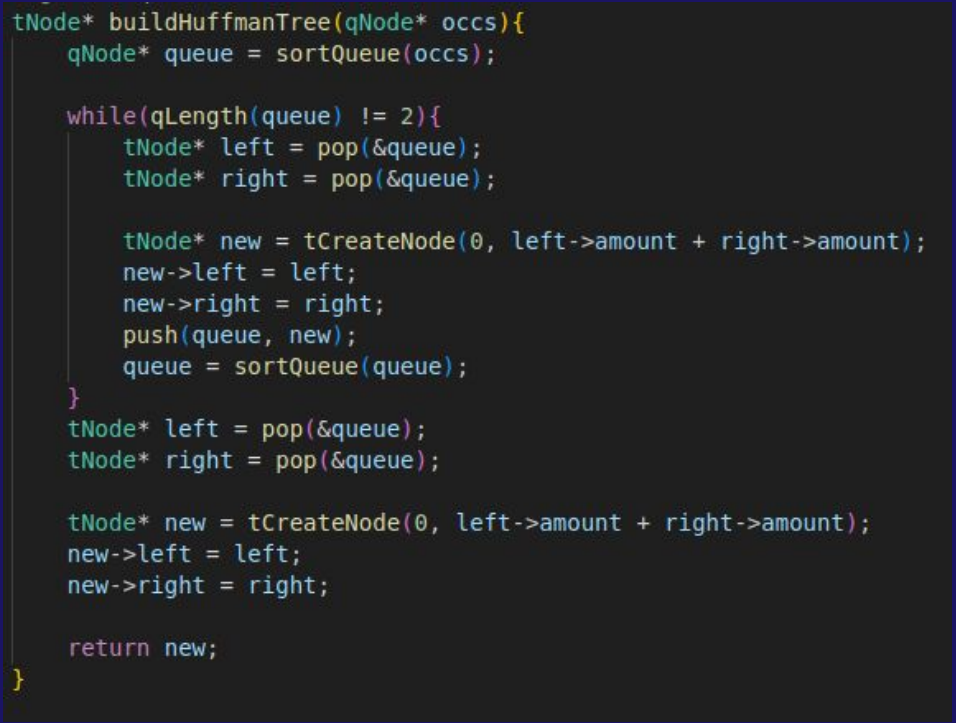
```c
qNode* sortQueue(qNode* head){
    qNode* q = head;
    while(q != NULL){
        qNode* inner = q;
        while(inner != NULL){
            if(q->node->amount > inner->node->amount){
                tNode* tmp = q->node;
                q->node = inner->node;
                inner->node = tmp;
            }
            inner = inner->next;
        }
        q = q->next;
    }
    return head;
}

int qLength(qNode* head){
    qNode* tmp = head;
    int counter = 0;
    while(tmp != NULL){
        counter++;
        tmp = tmp->next;
    }
    return counter;
}

void freeQueue(qNode* head){
    if(head == NULL){
        return;
    }

   qNode* tmp;
   while (head != NULL){
       tmp = head;
       head = head->next;
       free(tmp);
   }
}
```

Counting occurences



```c
qNode* countOccs(char* fileName){
    FILE* file = fopen(fileName, "r");
    qNode* head = qCreateNode(tCreateNode(fgetc(file), 1));
    char current;
    while((current = fgetc(file)) != EOF){
        qNode* tmp = head;
        boolean flag = true;
        while(tmp){
            if(tmp->node->val == current){
                tmp->node->amount++;
                flag = false;
            }
            tmp = tmp->next;
        }
        if(flag)
            push(head, tCreateNode(current, 1));

    }
    return head;
}
```

Building the tree

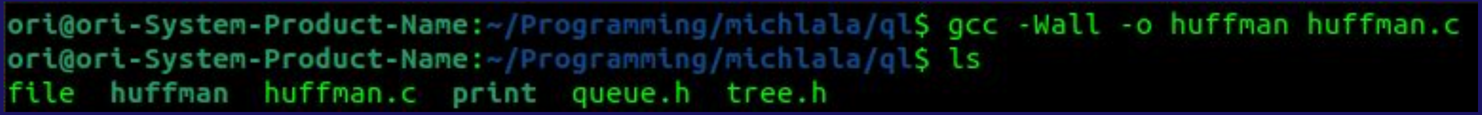


```c
tNode* buildHuffmanTree(qNode* occs){
    qNode* queue = sortQueue(occs);

    while(qLength(queue) != 2){
        tNode* left = pop(&queue);
        tNode* right = pop(&queue);

        tNode* new = tCreateNode(0, left->amount + right->amount);
        new->left = left;
        new->right = right;
        push(queue, new);
        queue = sortQueue(queue);
    }
    tNode* left = pop(&queue);
    tNode* right = pop(&queue);

    tNode* new = tCreateNode(0, left->amount + right->amount);
    new->left = left;
    new->right = right;

    return new;
}
```

executing

Compiling

Before executing the program we must compile it. In order to do that we will use the gcc command with -Wall to see extra warnings and -o so we can give a name to the compiled file.

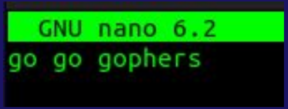


```
ori@ori-System-Product-Name:~/Programming/michlala/ql$ gcc -Wall -o huffman huffman.c
ori@ori-System-Product-Name:~/Programming/michlala/ql$ ls
file  huffman  huffman.c  print  queue.h  tree.h
```

executing

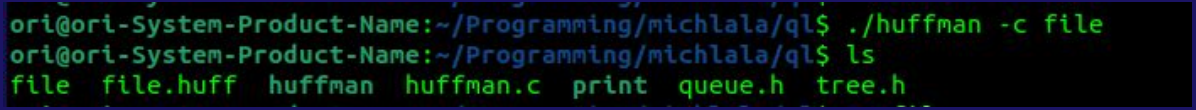
When we execute we use -c when we want to compress and we use -d when we want to decompress/extract a file that is already compressed to its original form.

File’s content :



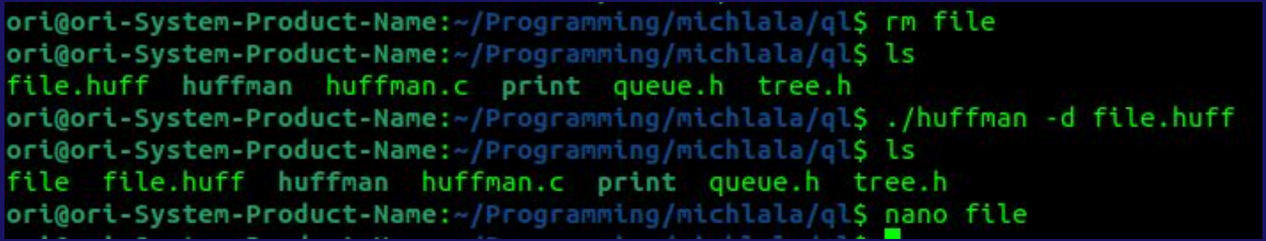
```
  GNU nano 6.2
go go gophers
```

Compressing:

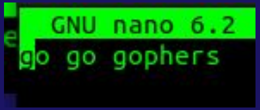


```
ori@ori-System-Product-Name:~/Programming/michlala/ql$ ./huffman -c file
ori@ori-System-Product-Name:~/Programming/michlala/ql$ ls
file  file.huff  huffman  huffman.c  print  queue.h  tree.h
```

Decompressing:



```
ori@ori-System-Product-Name:~/Programming/michlala/ql$ rm file
ori@ori-System-Product-Name:~/Programming/michlala/ql$ ls
file.huff  huffman  huffman.c  print  queue.h  tree.h
ori@ori-System-Product-Name:~/Programming/michlala/ql$ ./huffman -d file.huff
ori@ori-System-Product-Name:~/Programming/michlala/ql$ ls
file  file.huff  huffman  huffman.c  print  queue.h  tree.h
ori@ori-System-Product-Name:~/Programming/michlala/ql$ nano file
```



```
  GNU nano 6.2
go go gophers
```