25/02/21

(12) File Zipper (Huffman Encoding)

Just like WinRar we'll create a project to which could compress files (for now only text (.txt) files) and it'll be a lossless compression that is there'll be no data loss.

In our project we'll display 2 things, first will be the encoded data or the compressed data and second will be a Compression Ratio (If its greater than 1 then compression is lossless or successful! and higher it goes the better comprsn it gets)

• Huffman Coding

> It's a loseless data compression algorithm

> We assign variable-length codes to input charctrs, length of which depends on frequency of chars:

ex. aaaaa bb.. → 5, 45

	Freq.	code
a	5	1100
b	45	0

> The variable-length codes assigned to input chareeters are prelix codes. } this mks sure our data is uniquely decodable

{0, 11}
Prelix Codes

{0, 1, 11}
Non prelix Codes

- Algo to create Huffman Tree

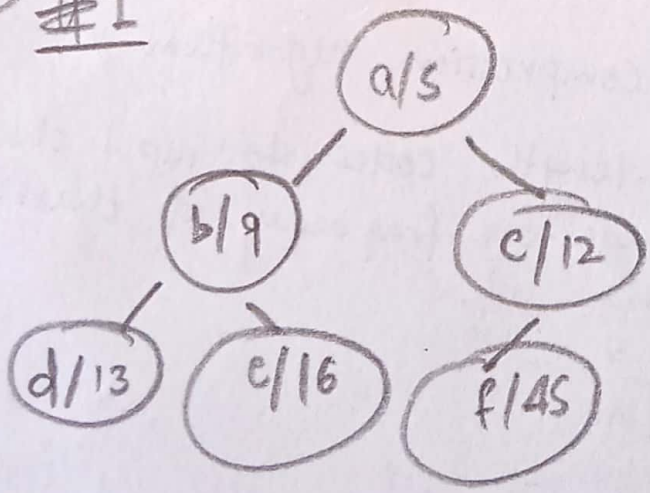
(1) Create a leaf node for each unique character and build a min heap of all leaf nodes.

(2) Extract two nodes with the min. freq. from the min. heap.

(3) Create a new internal node with freq. equal to the sum of the 2 nodes frequencies. Make the first 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.

ex #1

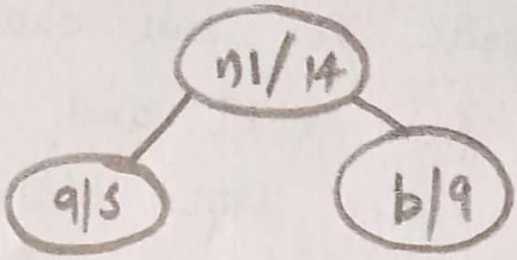


Min Heap (MH)

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

#2 a/5 & b/9

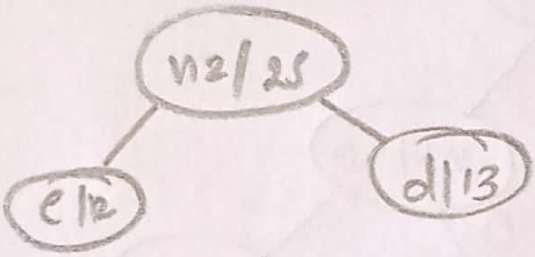
3



char	freq
n1	14
c	12
d	13
e	16
f	45

3 MH

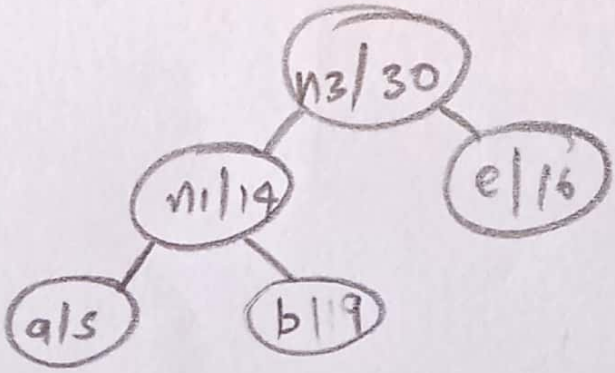
#4 → #2 & #3



char	freq
n1	14
n2	25
e	16
f	45

3 MH

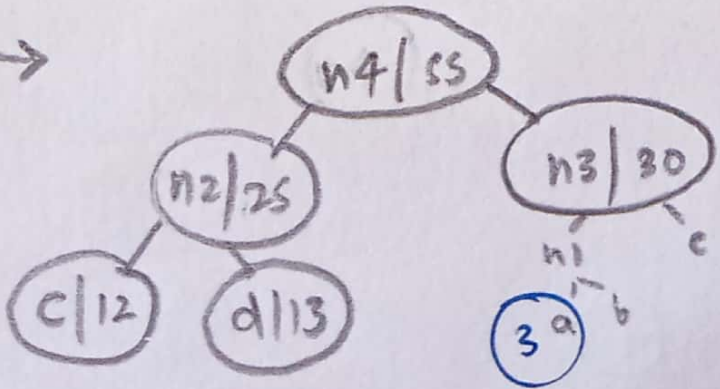
→ #2 & #3



char	freq
n2	25
n3	30
f	45

3 MH

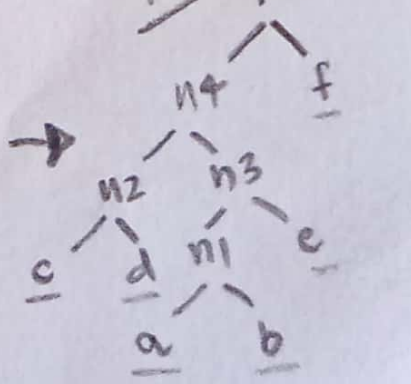
→



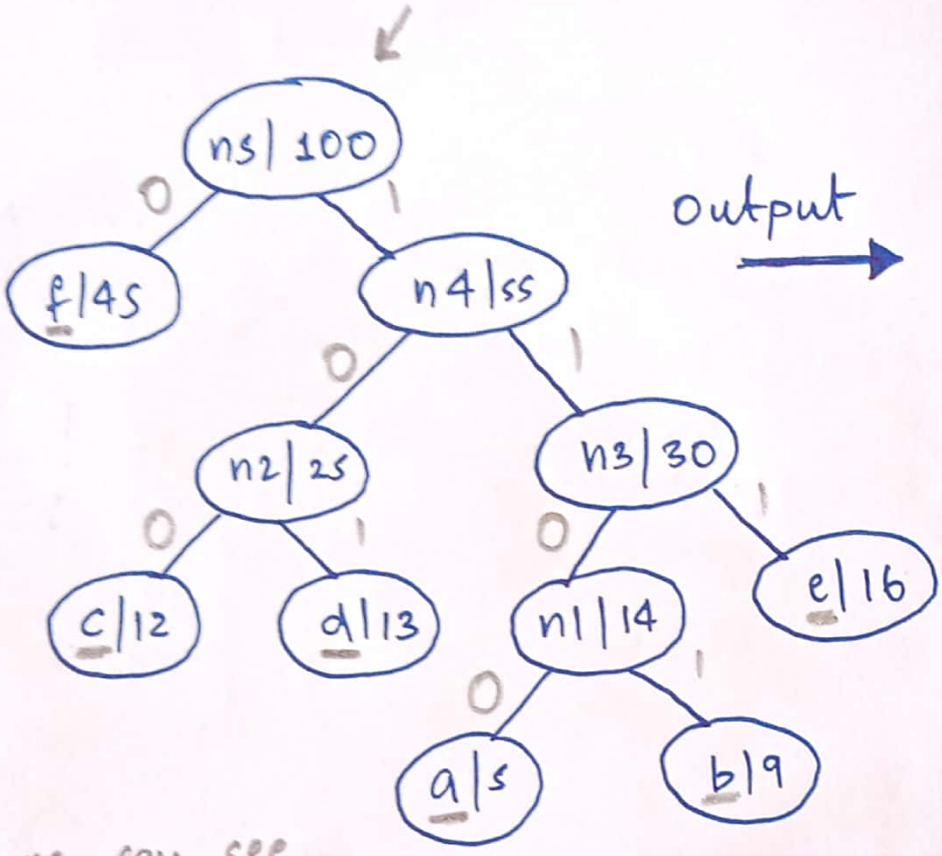
c	f
n4	55
f	45

→

Huffman tree



Now we'll generate code for our chars through this tree by traversn from root and assign '0' to left child and '1' to the right child, and once we encounter a leaf node we print the code.



output →

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

As we can see that higher the freq. of a char lower is its code length.

$$\therefore \text{len of code} \propto \frac{1}{\text{freq. of char.}}$$

take one more (ex).

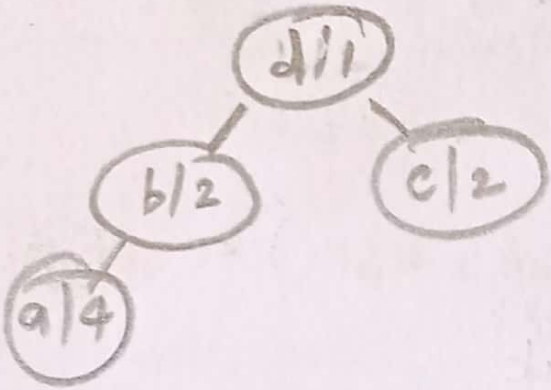
Input → abacdabac

- first we'll create a freq. arr. for all unique chars.

we get → a - 4
b - 2
c - 2
d - 1

- Now we'll create leaf node (of char & freq) for all unique chars and build min heap with 'em.

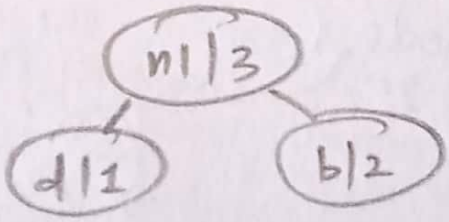
#1



char	freq
d	1
b	2
c	2
a	4

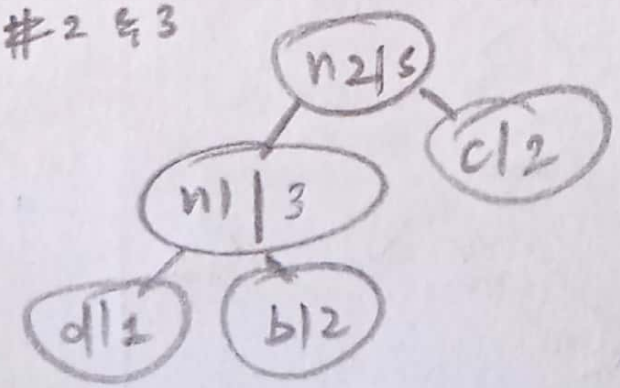
- Now tk 2 nods with min freq. & mk a node of thm.

#2 & 3

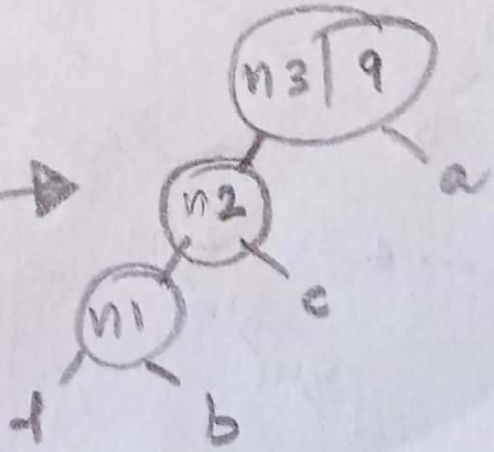


char	freq
n1	3
c	2
a	4

- #2 & 3

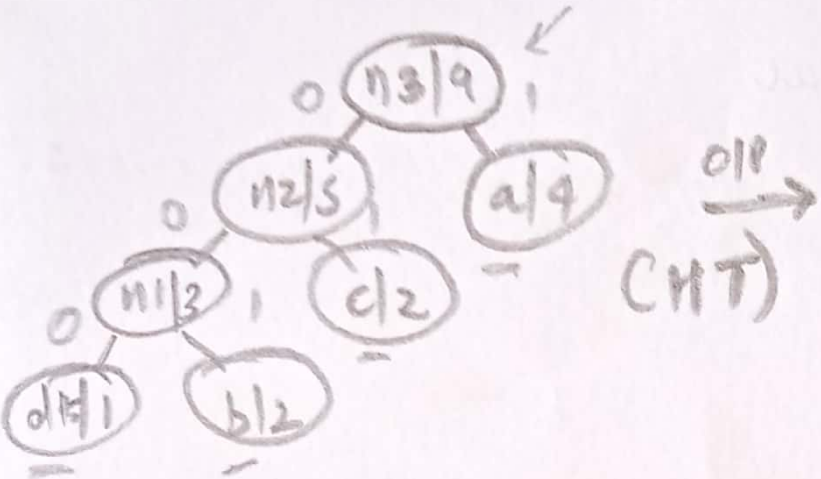


c	f
n2	5
a	4



(5)

• Acc. we'll write code for it:



011 → (HT)

char	code
d	0 0 0
b	0 0 1
c	0 1
a	1

∴ with the help of our o/p we can generate the code:

encoded

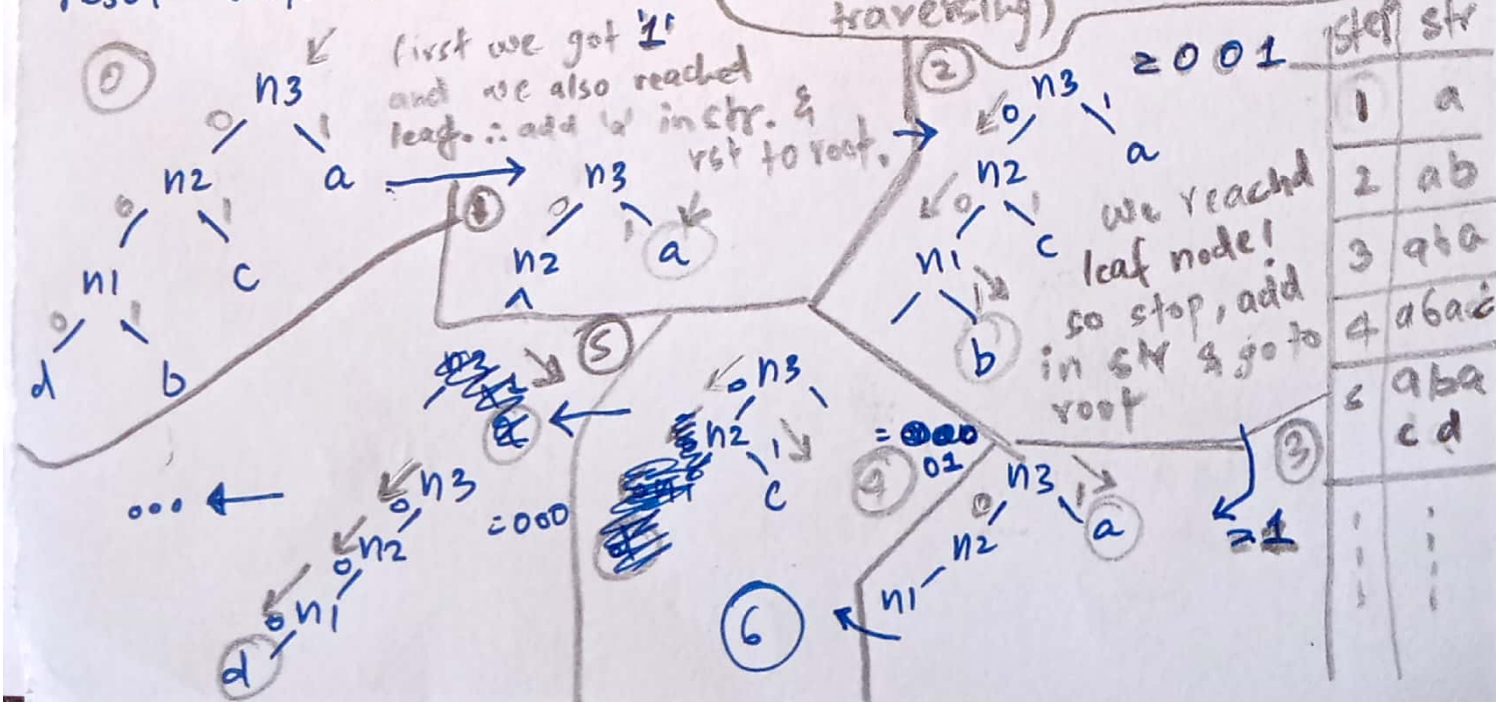
abacdabac → 1 001 1 01 000 1 001 1 01

17 bits

∴ After encoding we reduced 72 bits (9 chars x 8 bits) to 17 bits!

• Now let decode it back:

- traverse from root. create an empty string. If we reach a leaf node then add its char to string & reset back to root node. (follow encoded data for traversing)

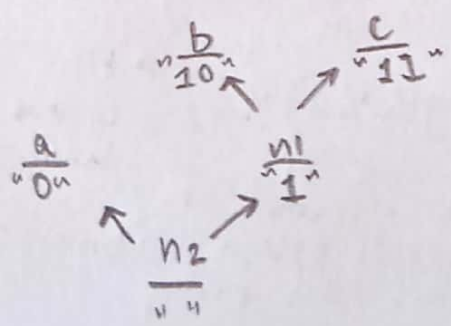


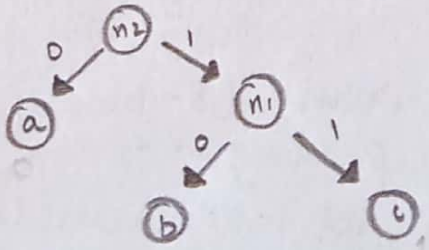
des + logic

```
import {BinaryHeap} from './heap.js';
export {HuffmanCoder}

class HuffmanCoder {

    stringify(node){
        if (typeof(node[1])==="string"){
            return '1' + node[1];
        }
        return '0' + this.stringify(node[0]) +
                     '1' + this.stringify(node[1]);
    }

    getMappings(node, path){ → follows DFS
        if (typeof(node[1])==="string"){
            this.mappings[node[1]] = path;
            return;
        }
        this.getMappings(node[0], path + "0");
        this.getMappings(node[1], path + "1");
    }
```

(5)

Jo leaf nodes honge unka 2nd para chartr hoga and in js chars and strs are reprsntd similarly





encode(data){ → Jo string ko encode krna hota hai: ex. "abc"

this is the max heap ← {this.heap = new BinaryHeap();

```
const mp = new Map();
for(let i=0; i<data.length; i++){
    if (data[i] in mp){
        mp[data[i]] = mp[data[i]] + 1;
    }
    else{
        mp[data[i]] = 1;
    }
}
```

① (covers the for loop)

```
for (const key in mp){
    this.heap.insert([-mp[key], key]);
}
```

②

```
while (this.heap.size() > 1){
    const node1 = this.heap.extractMax();
    const node2 = "          ";
    const node = [node1[0]+node2[0], [node1, node2]];
}
```

③

④ { const huffman_encoder = this.heap.extractMax();

⑤ { this.mappings = {};
this.getMappings(huffman_encoder, "");

```
let binary_string = "";
for(let i=0; i<data.length; i++){
    binary_string = binary_string + this.mappings[data[i]];
}
```

⑥ we also need to store it as while decoding apnko original string chahie hogi to decode

```
let rem = (8 - binary_string.length%8)%8;
let padding = "";
for(let i=0; i<rem; i++){
    padding += "0";
}
binary_string += padding;
```

⑦ → kitne extra 0bits chahie & bit banane ke liye

→ ye utne 0's string format me store krlega thn binary-string me add krdeg

⑧

```
let result = " ";
for (let i=0; i<binary_string.length; i+=8){
    let num = 0;
    for (let j=0; j<8; j++){
        num = num*2 + (binary_string[i+j] - "0");
    }
    result = result + String.fromCharCode(num);
}
```

(8)

(9) `let final_res = this.stringify(huffman_encoder) + '\n' + rem + '\n' + result;`

(10) `let info = "Compression ratio:" + data.length/final_res.length;`

`info = "Comprsn ~~ratio~~ complt & file sent for download" + info;`

(11) `return [final_res, this.display(huffman_encoder, false), info);`

`}`

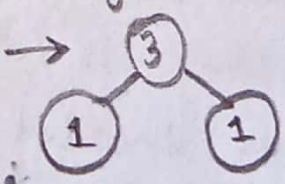
- Code explanation for encode fn():

(1) We made an frequency hashmap for each characters, we can also create an array of size 256.

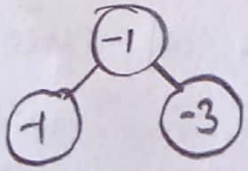
(2) Now we add all the chars along with their freq. ex. (3,"a") in heap. As it's a max. heap we add -freq. which gives us a min heap!

(ex) we had map = { {a:3}, {b:1}, {c:1} }

so if apn '-' nhi lagate to heap esa banta →



It's just a hack to get min heap.

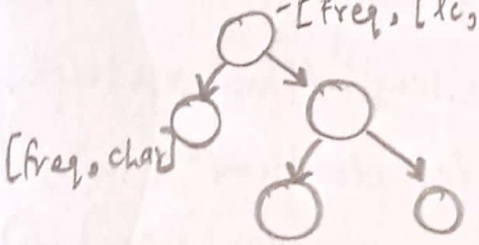


← but as we use min heap and pick top 2 nodes with min freq, we add '-' in

(9) front of every freq & we get:

(3) Here while we have more than 1 nodes in heap (n_1 & n_2) we take top 2 nodes with min freqs and create a new internal node with n_1 & n_2's freq sum as 1st para and an array of left (n_1) & right (n_2) child as 2nd para. Then we finally insert the newly created node back in the heap.

(4) Now finally we'll extract the root node which'll be the huffman tree (HT).



Structure of HT:

Intrnal node

all the non-leaf nodes will have a node structure of freq. and an arr of size 2 with left child & right child. ([freq, [-,-]]) and all left nodes'll have freq & char only. ([freq, char])

(5) Through this we get the code for all the leaf nodes, in mapping. this gave us → mappings = {{a:0}, {b:10}, {c:11}.}

(6) This creates the encoded code by apndn all the codes of chars → 0 10 11 00

(7) & (8) Initially we had data = "abcaa" which was of 8×5 = 40 bits, Now although we have encoded it, it still is in the form of char which is of 7×5 = 35 bits! to reduce it:- we'll convert them to ASCII chars! Now as one char is of 8 bits we'll take starth 8 chars from our encoded data and convert to it ASCII charctr then we'll do with the remaining chars of encoded data, if the remaining chars are less than 8 we'll add remaining zeroes and then convert to ASCII this'll give us 0101100 & 0_ = e (b) we'll cnvrt this to it's ASCII & save it.

will reduce our bits from 40 to 8! & from 5 chars 1 char.

(8) specifically convert our 8 bit codes to ASCII chars and ~~add~~ apnd to result. finally our result will have all 8 bit ASCII chars.

* Aftr encryption our data abcaa will become X as 01011000's ASCII is X. and X doesn't mean anything :)

(9) stringify func) gets string vr of our huffman tree. n apn "1" islye add krte str_ func) me kyoki agr apne data me 0 or 1 hote to diffrntiate nhi krte pate path (ie 0s & 1s) se islye we use "<".

to decode the code apn final_res banate hai which (10) apart from result extra info store krta hai like remaing bits and string vr. of our tree to finally apne bits 40 se lagbg 16 to gaye which is good!

*Note → In some cases our compression might not work. when our data is less like abc (24 bits) our final_res might come (50 bits). So to get meaningful o/p get data which has more than 10k or gtr chars.

(10) Compression ration = $\frac{\text{original data len}}{\text{new}} \Rightarrow \frac{5}{1} = 5$

this tells how good compresn was.

(11) Finally we return all the informatn like final_res, info and also call display fn to display our tree.

(11)

- Remaining code of Huffman Coder Class

```
destringify (data) {
    let node = [];
    if (data[this.ind] == '\') {
        this.ind++;
        node.push(data[this.ind]);
        this.ind++;
        return node;
    }
    this.ind++;
    let left = this.destringify(data);
    node.push(left);
    this.ind++;
    let right = this.destringify(data);
    node.push(right);
    return node;
}
```

3 — 3i, 3ii

```
decode (data) {
    data = data.split('\n'); }
    if (data.length == 4) {
        data[0] = data[0] + '\n' + data[1];
        data[1] = data[2];
        data[2] = data[3];
        data.pop();
    }
```

1

2

```
this.ind = 0;
const huffman_decoder = this.destringify(data[0]);
const text = data[2];

let binary_string = "";
for(let i=0; i<text.length; i++){
    let num = text[i].charCodeAt(0);
    let bin = "";
    for(let j=0; j<8; j++){
        bin = num%2 + bin;
        num = Math.floor(num/2);
    }
    binary_string += bin;
}
```

to make our binary string of 8 bits we added some padding. So to remove them we use data[1] and get our original binary string back.

```
binary_string = binary_string.substring(0, bin_str.length - data[1]);
console.log(binary_string.length);

let res = "";
let node = huffman_decoder;
for(let i=0; i<binary_string.length; i++){
    if(binary_string[i] == '0'){
        node = node[0];
    }else{
        node = node[1];
    }
    if(typeof(node[0]) == "string"){
        res += node[0];
        node = huffman_decoder;
    }
}
let info = [res, this.display(huffman_decoder, true), info];
return info;
```

- **Explanation**

from the encode fn() we recieve data in the form of "Tree(HT) \n Rem(padding) \n Encoded data".

(1) ∴ To retrieve these information we use split fn which splits data on the baris of "\n". Thats how we get 3 diff. info in data.

(2) Handles Special case. If we have '\n' (new line) in our HT then we do this.

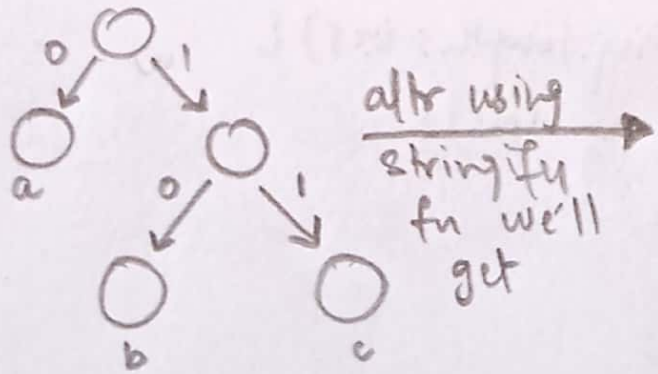
eg) in our tree we get '\n' → Tree

∴ Instead of 2 '\n' we get 3 '\n' and string breaks into 4 parts.

1st para '\n'
2nd para. etc

(3) Now we have string rs of HT in our data[0], we need to get our original HT to decode the data.

let's take an (ex) to understand:-



alr using stringify fn we'll get → 0'a10'b1'c

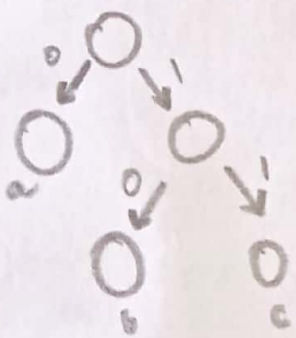
Now to destringify it or to get the original HF we'll handle 2 case:

(3i) For leaf node

(3ii) For Internal node

(3ii) Every internal # node will have left and right child. (3ii) will go on these nodes. By ↑marking "ind" we ensure that we catch only leaf nodes, baaki for left and right children (ie for 0 & 1) we just keep traversing.

(3i) As soon as we get '6' we can confirm that we've got to leaf node. Aftr reaching we increament "ind" to reach charctr thn push it in node thn agn ↑mnt to read nxt char.

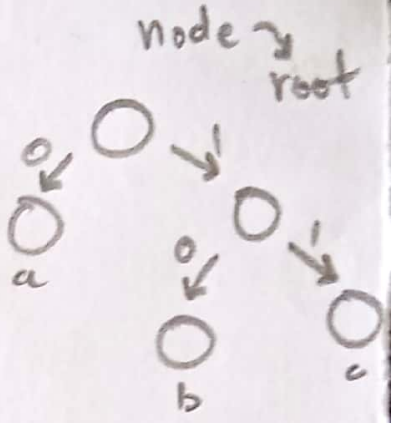


0'a10'b1'c

(4) Now we've to get our binary string from encoded text (ex) "!," jaise code se → 1001101----.
so that we could decode it back!
suppose our encoded text was → "!," now this was created via binary string, now we've to again get back our binary string via encoded text.
lets assume for "!" ASCII code is 1000110, first we get "!" number (tht ranges from 0-256) thn we get its binary string and apnd it in "binary-string", similarly we do with ",".

(5) Now here we'll use our binary-string after removing the padding, Huffman Tree (HT) to get back the original data

from "!," we got (after removing padding) 0110010 ← binary-string



we'll traverse the binary string, first we start from root node. then we got "0" so we update our node to left child node, like this we keep on going (for 0 we go left & for 1 right).

And when we get leaf node, we add its char to result ("res") and reset our node to root node.

(6) Finally we get our decoded data in "res". In end return.