## Huffman Tree

The Huffman's algorithm is wildly used technique for data compression saving from 20% to 90% of space depending on the characteristic of the data being compressed. The algorithm starts with creating a frequency table that represents the number of occurrences of a given character in the sequence of data. The purpose of the Huffman's algorithm is to find a variable-length binary code for each character depending on its appearance in a text file. The higher frequencies of a character the smaller number of bits assigned to represent the characters. And the lower frequencies of a character the larger number of bits for the character. This method is superior to the ASCII code that uses a fixed amount of space (1 byte) to encode a character. The Huffman's algorithm uses greedy strategy and is implemented based on the Minimum Priority Queue (n = the number of distinct characters).

```
for i = 1 to n-1
   do allocate_a_new_node z
      left[z] = x = Extract-Min(Q)
      right[z] = y = Extract-Min(Q)
      f[z] = f[x] + f[y]
      Insert(Q, z)
return Extract-Min(Q)
```

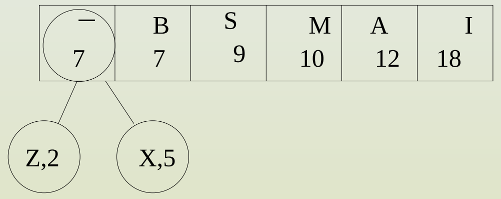
## Example

Consider a text file that generates the frequency table given below and treat it as the Minimum Priority Queue with the frequencies as the key. The priority queue is a sequence of pairs $(a, b)$ where $a$ corresponds to a unique character in a text file and $b$ represents its frequency in the text file.

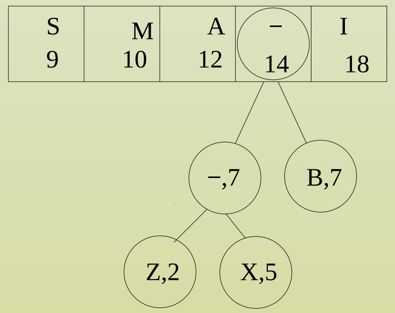
| Character | Z | X | B | S | M | A | I |
|---|---|---|---|---|---|---|---|
| Frequency | 2 | 5 | 7 | 9 | 10 | 12 | 18 |

The Huffman binary tree is built in the $n - 1$ steps, by merging subtrees in the the bottom up manner. At each iteration it extracts the first and second minimum from the Minimum Priority Queue, creates a binary tree node for each of them and creates also a node with the dummy character $(-)$ along with the sum of frequencies of the extracted minimums. See the illustration on the next slide.

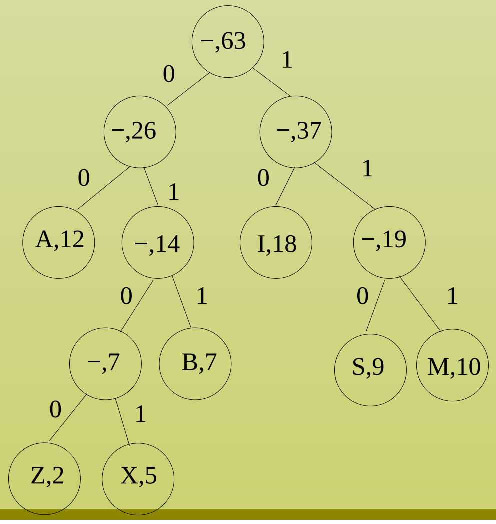
**The first merge to build a tree**



**The second merge**



**The Huffman tree after three more merges**



| Char. | Freq. | Code | Bits |
|---|---|---|---|
| A | 12 | 00 | 24 |
| Z | 2 | 0100 | 8 |
| X | 5 | 0101 | 20 |
| B | 7 | 011 | 21 |
| I | 18 | 10 | 36 |
| S | 9 | 110 | 27 |
| M | 10 | 111 | 30 |

## Decoding

The ASCII representation of this text file takes 504 bits compared to 166 bits used by the Huffman encoding which gives us around 33% compression ratio.

The decoding of the encoded binary file can also be done based on the Huffman tree following this simple algorithm:

1. Start from the root
2. If a binary value is equal to 0 go to the left subtree, otherwise go to the right subtree until the leave node is reached and the character is retrieved.
3. Repeat the previous steps until the end of the binary file is reached.

## The Running Time

The running time of the Huffman algorithm:

- The cost of creating a frequency table for a file/string of length $m$ takes $O(m)$
- The efficient cost of creating minimum priority queue based on a heap is $\mathrm{O}(n)$ where $n$ is the number of distinct characters in a text file
- Extraction of the first and the second minimum from the heap and insertion of the pair into the heap is $\mathrm{O}(\log_2 n)$
- The last step is repeated $n-1$ times that makes in total $\mathrm{O}(n \log_2 n)$ operations.

## Expected Code Length

Expected code length of an input text =

compressed text size/total # of input characters =

$\{ \sum_{i=1}^{n} \#\text{occurrences of character } i \times \text{length of Huffman code for character } i \}$

$\div$ total # of input characters

Note: The average code length cannot be calculated simply by summing up the code length of each unique character and dividing it by the total number of unique characters. Think about how you calculate your GPA. Normally you do not sum up the numerical grade of each course and divide it by the total number of courses taken.