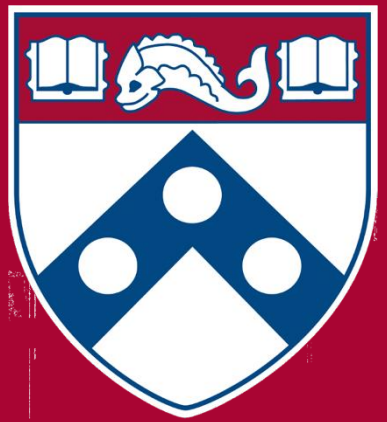
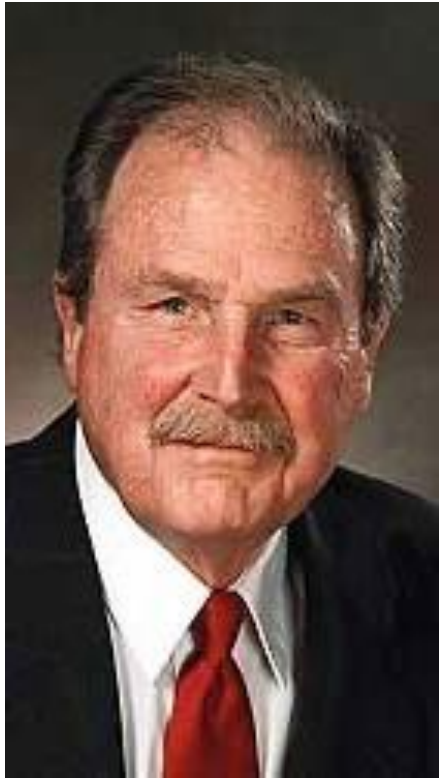
Huffman Coding Trees

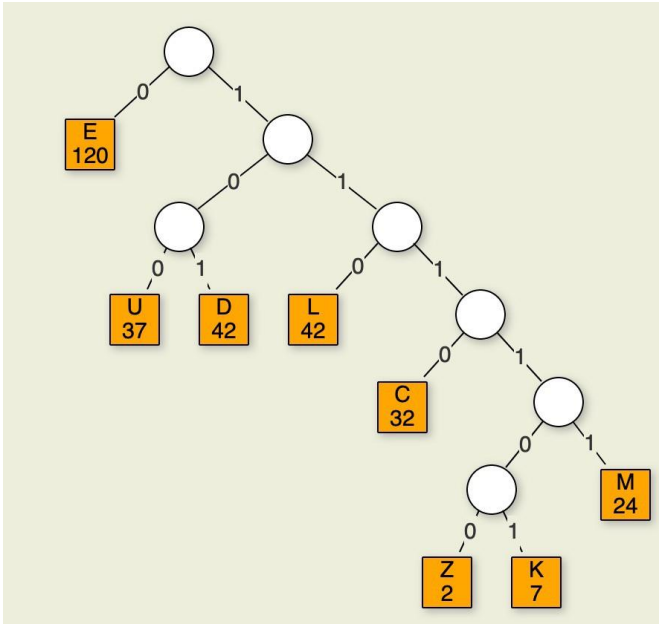
CIT 5940





David A. Huffman 1925-1999

Huffman Coding Tree



Introduction

- Fixed-length coding: encoding scheme that assigns a code to each object in the collection using codes that are all of the same length (ASCII)

- Variable-length coding: encoding scheme that assigns a code to each object in the collection using codes that can be of different lengths

Problem: Save space when storing data

- **Idea:**
  - *Use variable-length coding: assign shorter codes to frequently occurring data, assign longer codes to data occurring less often*
- At the heart of file compression techniques
- Huffman Trees help create variable-length coding
- Huffman codes use in “lossless” data compression

Priority Queue

- A sorted ADT.
- The head of a priority queue is always the smallest (or largest) element.
- Most often implemented using the *heap* data structure—next week!
- Java implementation: PriorityQueue
  - Elements are retrieved based on their natural ordering or by a comparator
  - Keep in mind: for integers, this means the smallest numbers are polled FIRST!

Building Huffman Coding Trees (for *n* letters/characters)

1. Create a collection of *n* initial Huffman trees, each tree is a single leaf node containing one of the letters and its frequency
2. Put the *n* partial trees onto a priority queue organized by weight (frequency)
3. Remove the first two trees (the ones with lowest weight) from the priority queue
4. Join the two trees together to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two trees
5. Put this new tree back into the priority queue
6. Repeat until all of the partial Huffman trees have been combined into one

Activity

- Build the CharCounter for the following text to get the frequencies of each character.

‘TONI MORRISON’

(include space)

Activity

- Build the CharCounter for the following text to get the frequencies of each character.

‘TONI MORRISON’

T: 1
O: 3
N: 2
I: 2
<SPACE>: 1
M: 1
R: 2
S:1

Activity

- Build the Huffman tree for the following text, including a count for a <PSEUDO_EOF> character.

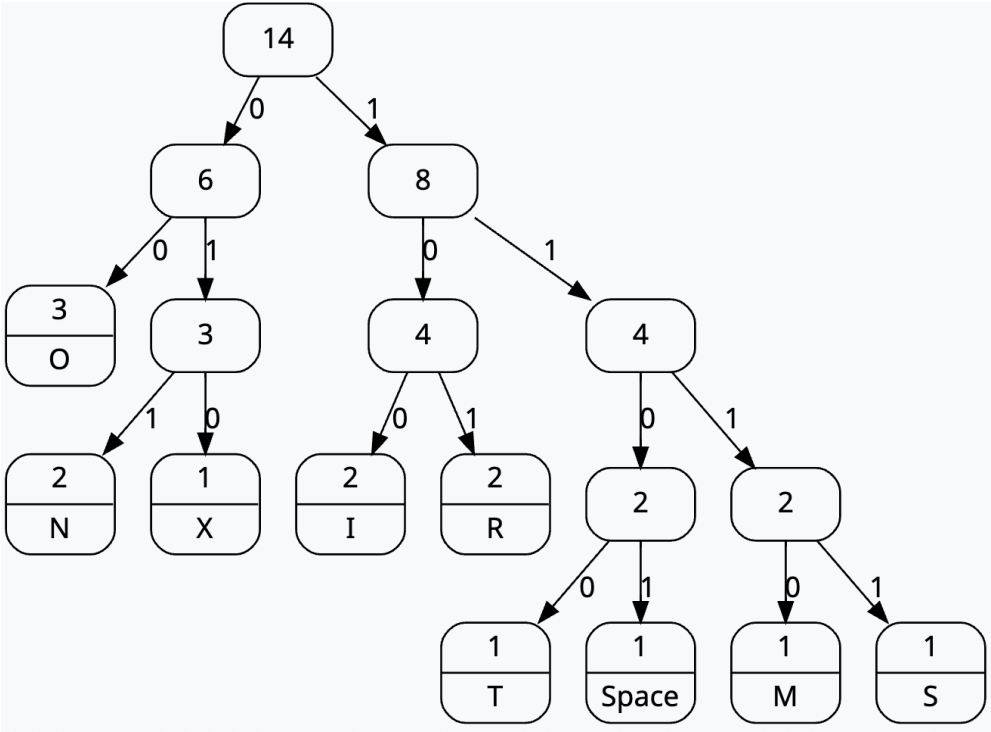
‘TONI MORRISON’

T: 1 <SPACE>: 1
O: 3 M: 1
N: 2 R: 2
I: 2 S:1
<P_EOF> : 1

Activity

- Build the Huffman tree for the ‘TONI MORRISON’ including a count for a <PSEUDO_EOF> character.

- “X” here refers to the <PSEUDO_EOF> character
- There are multiple valid trees, but here’s one



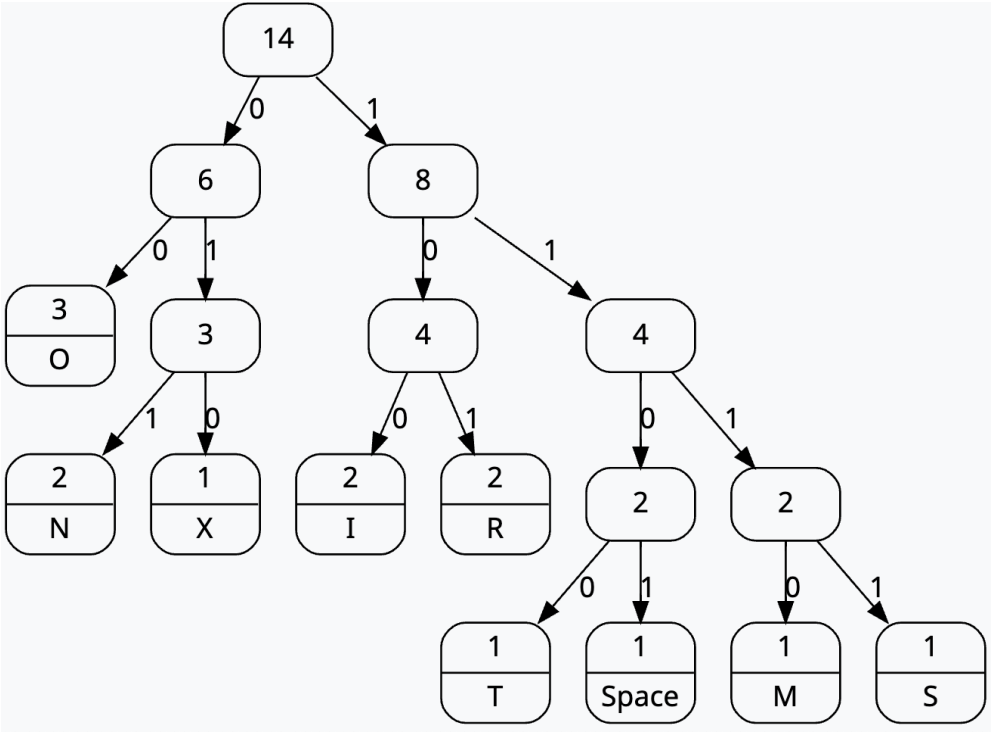
Assigning Huffman Codes: Encoding

- Beginning at the root:
  - Assign either a '0' or a '1' to each edge in the tree:
    - '0' is assigned to edges connecting a node with its left child
    - '1' to edges connecting a node with its right child
- Generate the codes for each letter
  - The code is the concatenation of the labels/values of the edges forming a path from the root to the letter

Activity

- Build the Huffman code using this tree.

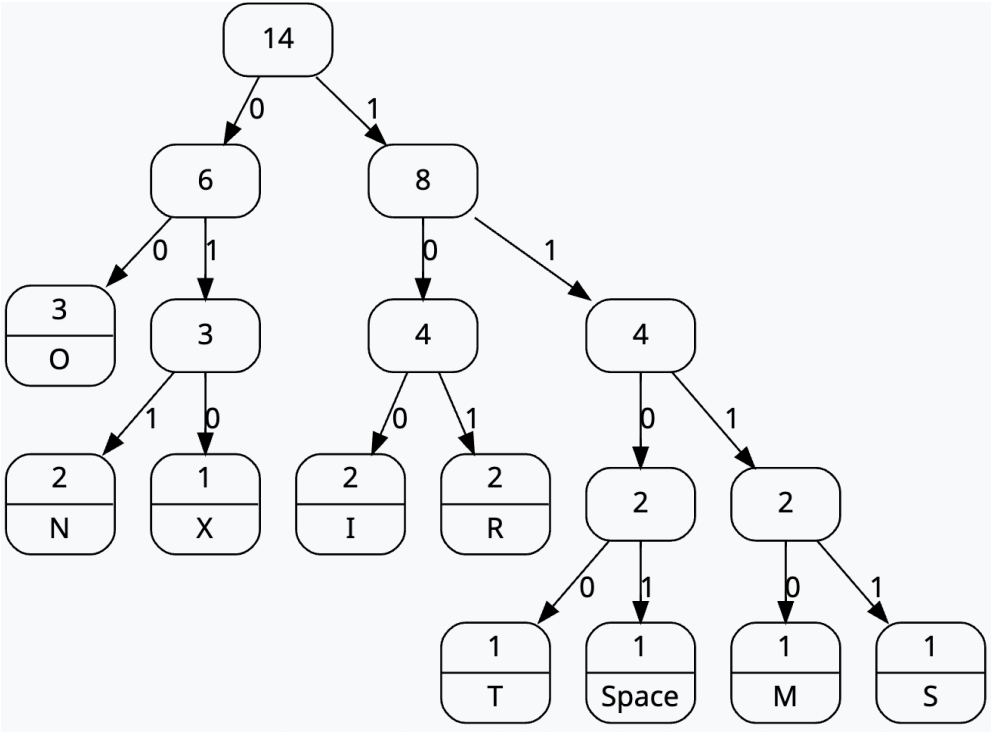
char	code
O	
N	
EOF (X)	
I	
R	
T	
Space	
M	
S	



Activity

- Build the Huffman code using this tree.

char	code
O	00
N	011
EOF (X)	010
I	100
R	101
T	1100
Space	1101
M	1110
S	1111



Prefix property

- Prefix property: *given a collection of strings, the collection has the prefix property if no string in the collection is a prefix for another string in the collection*

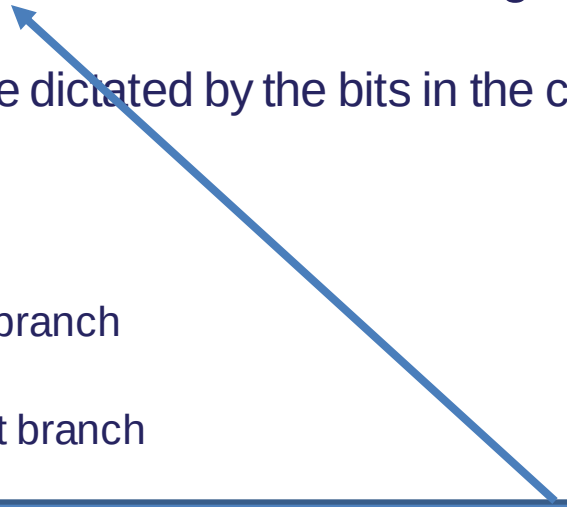
- Huffman codes meet the prefix property. Any prefix for a code correspond to an internal node, and all codes correspond to leaf nodes

Using Huffman Codes: Decoding

- Given a Huffman code and the tree used for encoding:
  - follow a path through the tree dictated by the bits in the code string
  - Starting at the root
    - Each '0' bit indicates a left branch
    - Each '1' bit indicates a right branch

Using Huffman Codes: Decoding

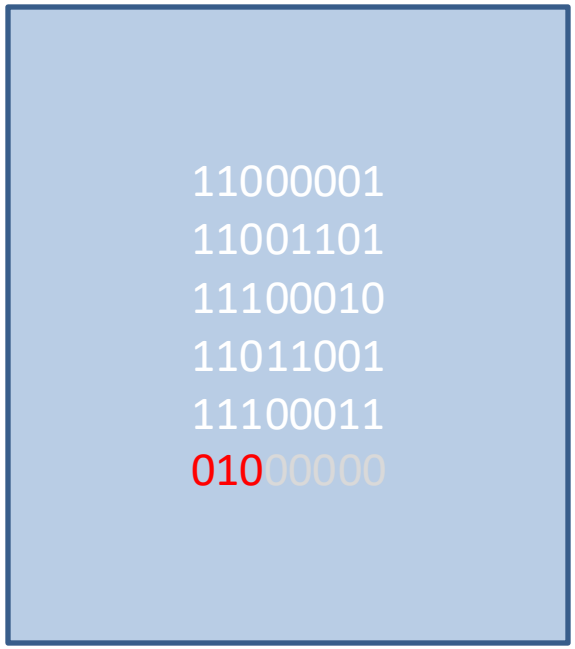
- Given a Huffman code **and the tree** used for encoding:
  - follow a path through the tree dictated by the bits in the code string
  - Starting at the root
    - Each '0' bit indicates a left branch
    - Each '1' bit indicates a right branch



PROBLEM: If I send you a compressed file, how are you supposed to know what encoding I used?

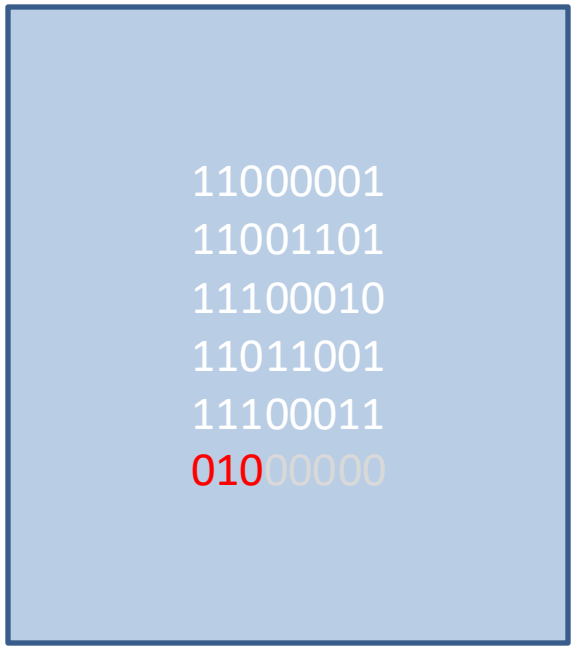
Using Huffman Codes: Decoding

- You receive *compressed.txt:*



Using Huffman Codes: Decoding

- You receive *compressed.txt:*



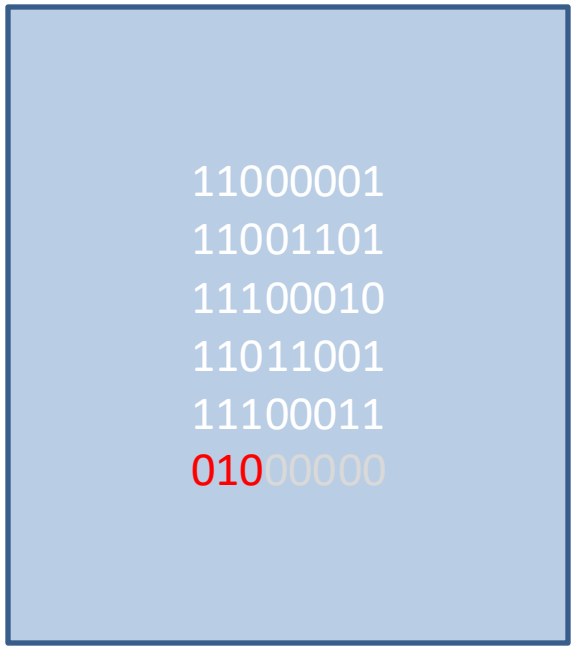
+

char	code
O	00
N	011
EOF (X)	010
I	100
R	101
T	1100
Space	1101
M	1110
S	1111

= TONI MORRISON

Using Huffman Codes: Decoding

- You receive *compressed.txt:*



+

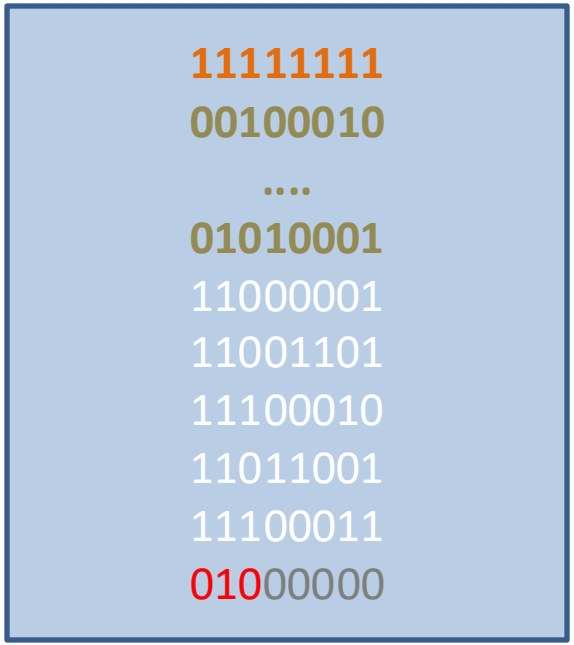
char	code
O	00
N	011
EOF (X)	010
I	100
R	101
T	1100
Space	1101
M	1110
S	1111

IDEA: Send the key with the compressed file

= TONI MORRISON

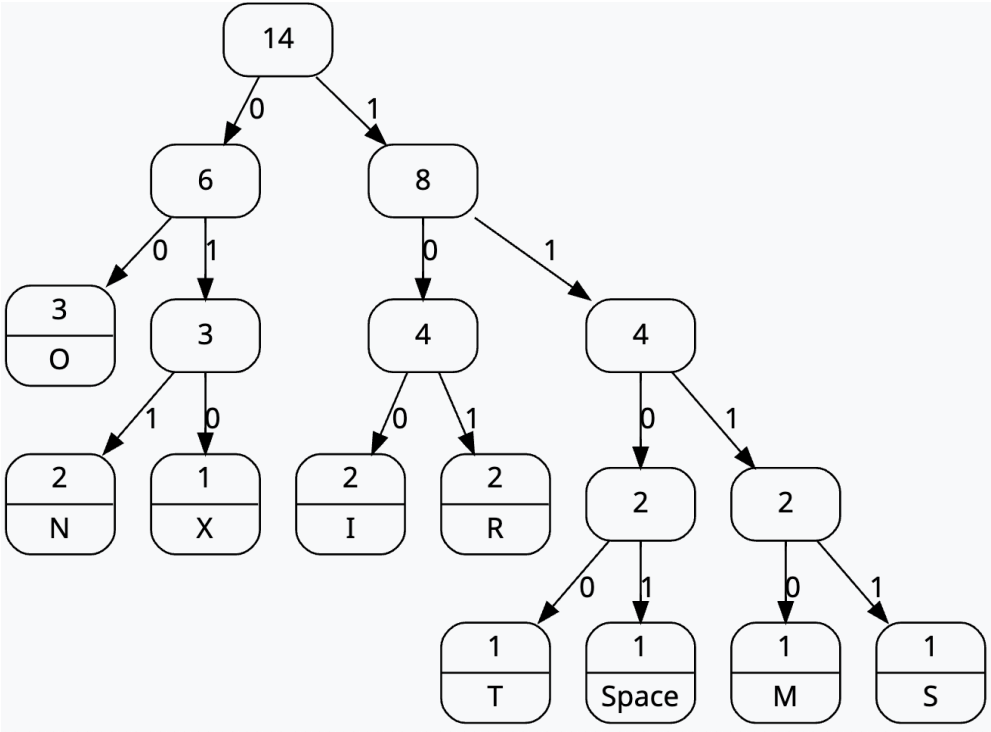
Using Huffman Codes: Header

- In *compressed.txt*, write:
- A magic number to identify the author of the file
- A traversal of the tree used for encoding that can be used to reconstruct the tree
- The compressed text of the file itself, including a PSEUDO_EOF char



Activity

- Preorder Traversal of this Huffman Tree
- Whenever we reach a non-leaf node, we write 0. Whenever we reach a leaf node, we write 1 followed by the 9 bit encoding stored inside the leaf.

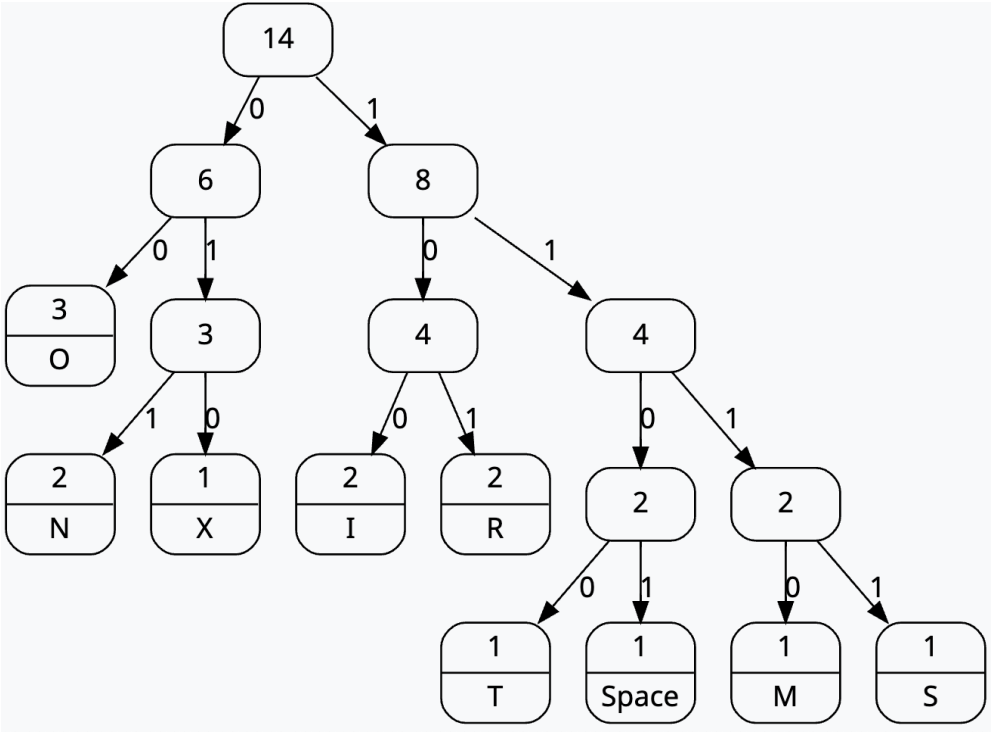


Activity

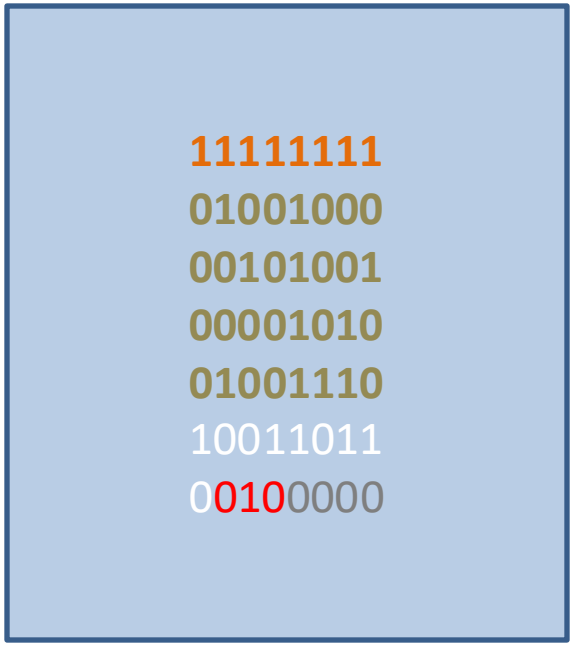
- Preorder Traversal of this Huffman Tree
- Whenever we reach a non-leaf node, we write 0. Whenever we reach a leaf node, we write 1 followed by the 9 bit encoding stored inside the leaf.

001**O**01**N**1**X**001**I**1**R**001**T**1**<SPACE>**01**M**1**S**

(the actual bits written would use 0001001111 for O, 0001001110 for N, etc)



Full Practice: Decompress this File!



01000001 01000010 01001110
A B N

Efficiency of Huffman coding

- Huffman coding does better when there is large variation in the frequencies of letters

- Huffman coding of a typical text file will save around 40% over ASCII coding if we charge ASCII coding at eight bits per character

Efficiency of Huffman coding

- Huffman coding for a binary file have a very different set of distribution frequencies and so have a different space savings

- Most commercial compression programs use two or three coding schemes to adjust to different types of files