# CS101 Data Structures

## Huffman Coding

(An Application of Binary Trees and Priority Queues)

Textbook Ch 16.3

# Compression

- Definition
  - Reduce size of data
    (number of bits needed to represent data)
- Benefits
  - Reduce storage needed
  - Reduce transmission cost / latency / bandwidth

# The Basic Idea

- Not all characters occur with the same frequency!
- Yet all characters are allocated the same amount of space
  - 1 char = 1 byte, be it e or X

- Idea: tailoring codes to frequency of characters
  - Use fewer bits to represent frequent characters
  - Use more bits to represent infrequent characters

# Example

| Symbol | A | B | C | D |
|---|---|---|---|---|
| Frequency | 12.5% | 25% | 50% | 12.5% |
| Original Encoding | 00 | 01 | 10 | 11 |
| | 2 bits | 2 bits | 2 bits | 2 bits |
| Huffman Encoding | 110 | 10 | 0 | 111 |
| | 3 bits | 2 bits | 1 bit | 3 bits |

- Expected size
  - Original $\Rightarrow$ 1/8×2 + 1/4×2 + 1/2×2 + 1/8×2 = 2 bits / symbol
  - Huffman $\Rightarrow$ 1/8×3 + 1/4×2 + 1/2×1 + 1/8×3 = 1.75 bits / symbol

# Algorithm

1. Scan text to be compressed and count frequencies of all characters.
2. Prioritize characters based on their frequencies in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Encode the text using the Huffman codes.

# Scan the text

- Consider the following short text:

  *Eerie eyes seen near lake.*

- What characters are present?

**E e r i space**

**y s n a r l k .**

# Scan the text

- Consider the following short text:

  *Eerie eyes seen near lake.*

- What is the frequency of each character in the text?

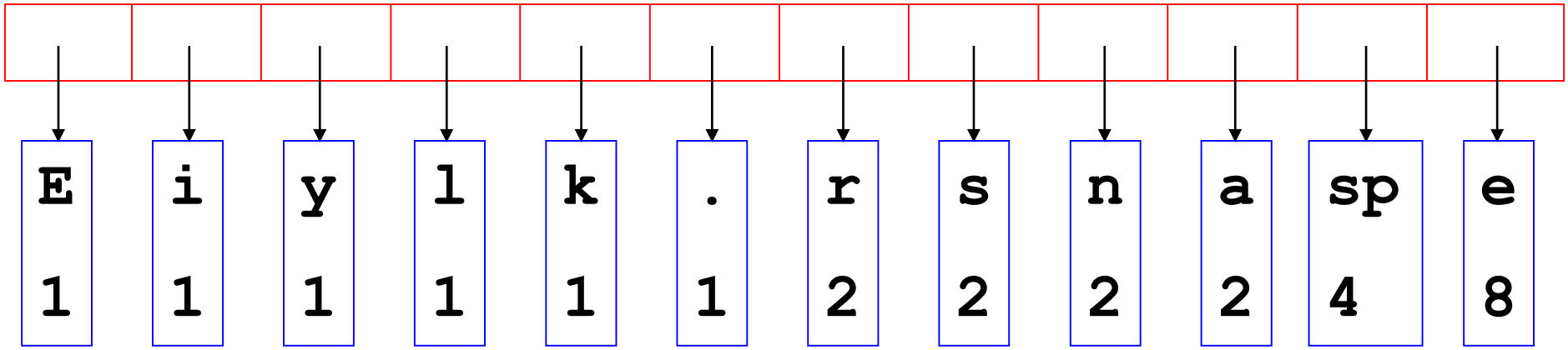
| Char | Freq. | Char | Freq. | Char | Freq. |
|---|---|---|---|---|---|
| E | 1 | y | 1 | k | 1 |
| e | 8 | s | 2 | . | 1 |
| r | 2 | n | 2 | | |
| i | 1 | a | 2 | | |
| space | 4 | l | 1 | | |

# Prioritize characters

- Create binary tree nodes with character and frequency of each character
- Place nodes in a priority queue
  - The lower the occurrence, the higher the priority in the queue

# Prioritize characters

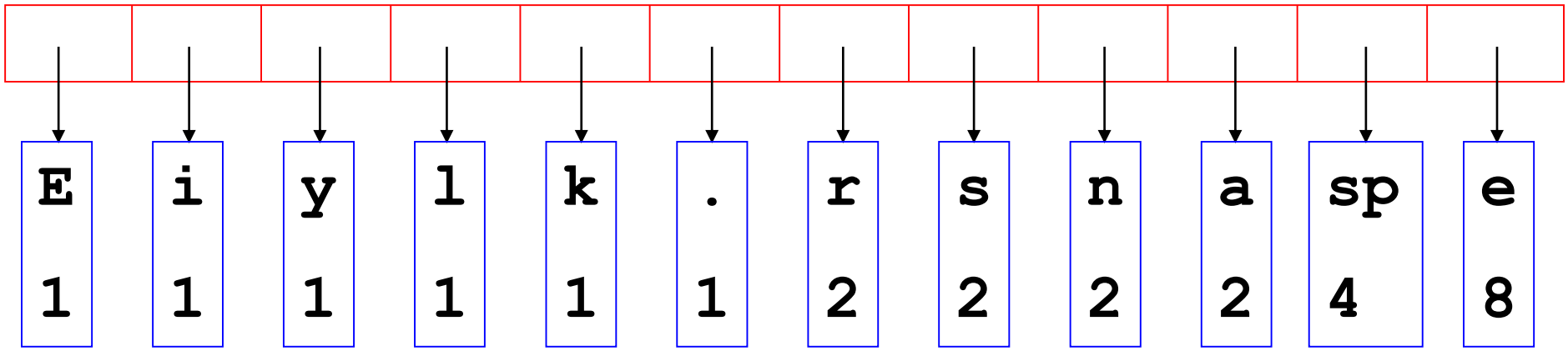
- The priority queue after inserting all nodes



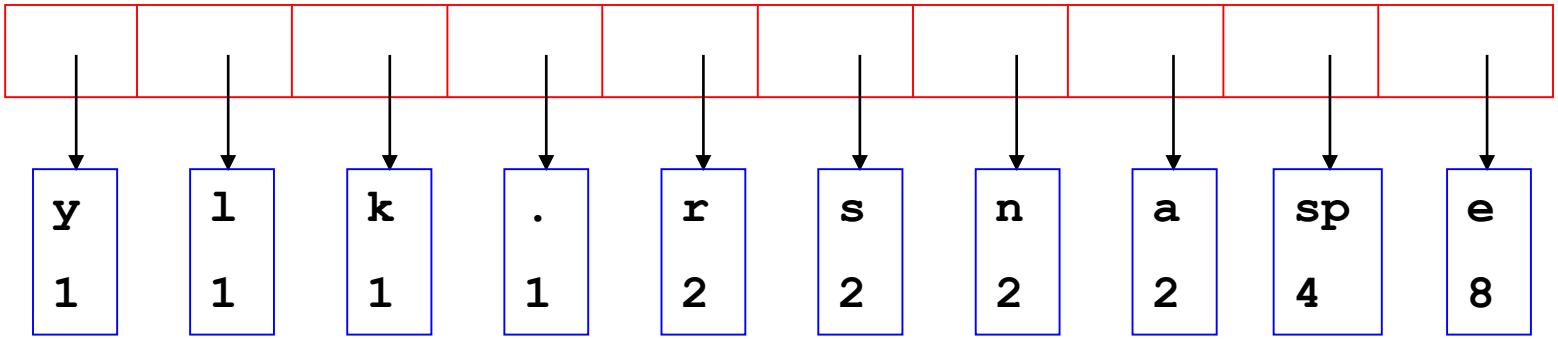
# Building a Tree

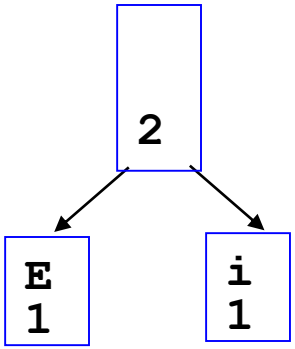
- While priority queue contains two or more nodes
  - Create new node
  - Dequeue node and make it left subtree
  - Dequeue next node and make it right subtree
  - Frequency of new node equals sum of frequency of left and right children
  - Enqueue new node back into queue

# Building a Tree

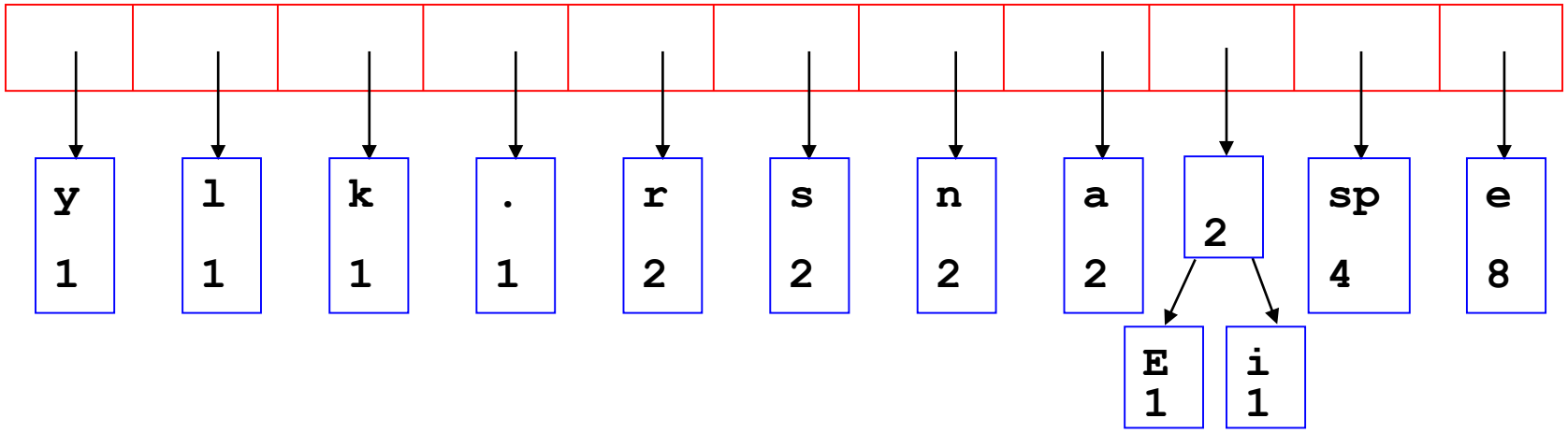


# Building a Tree

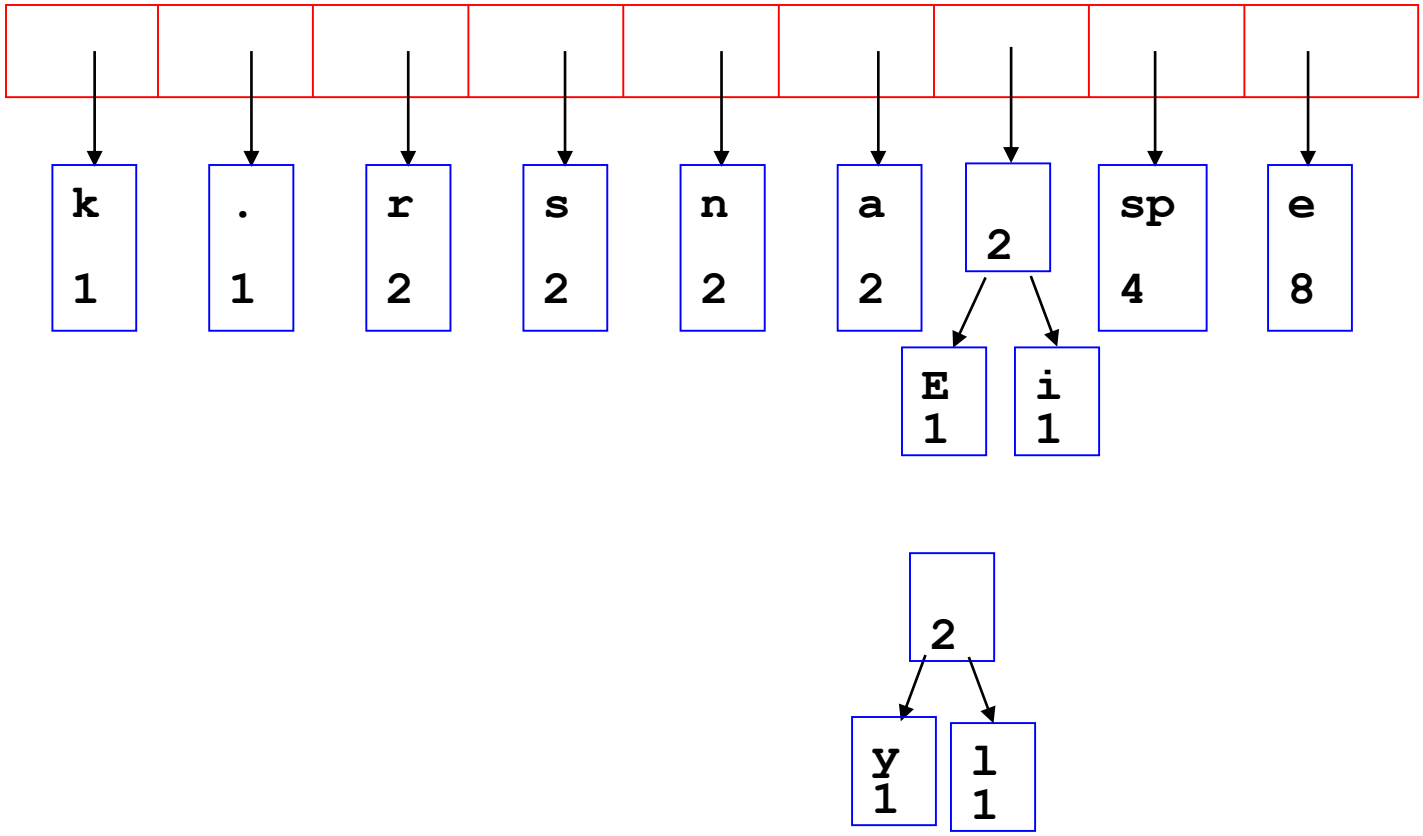


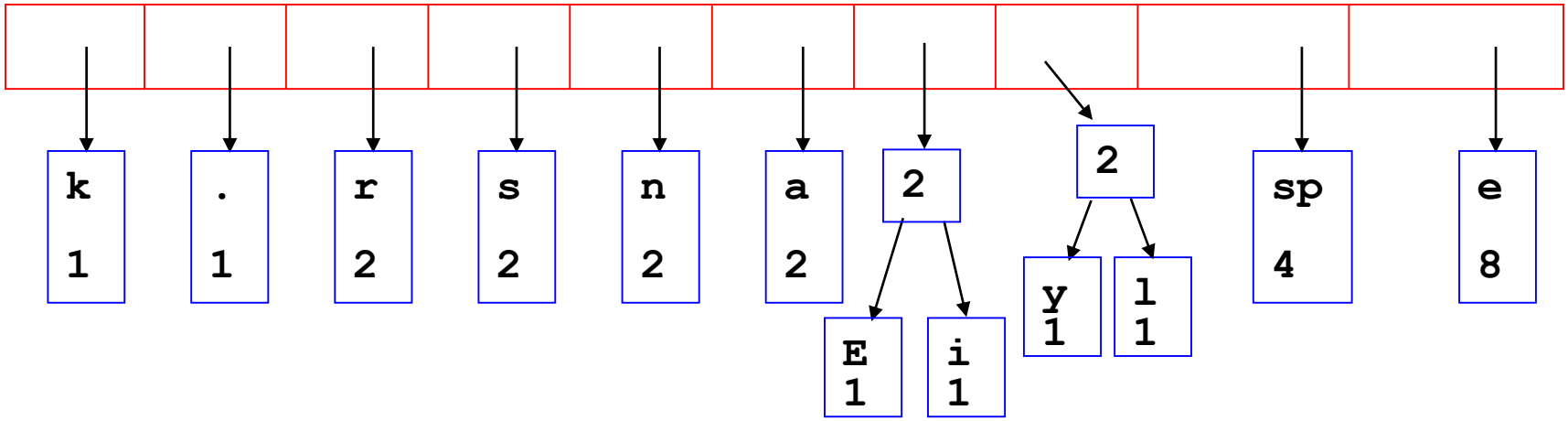
# Building a Tree



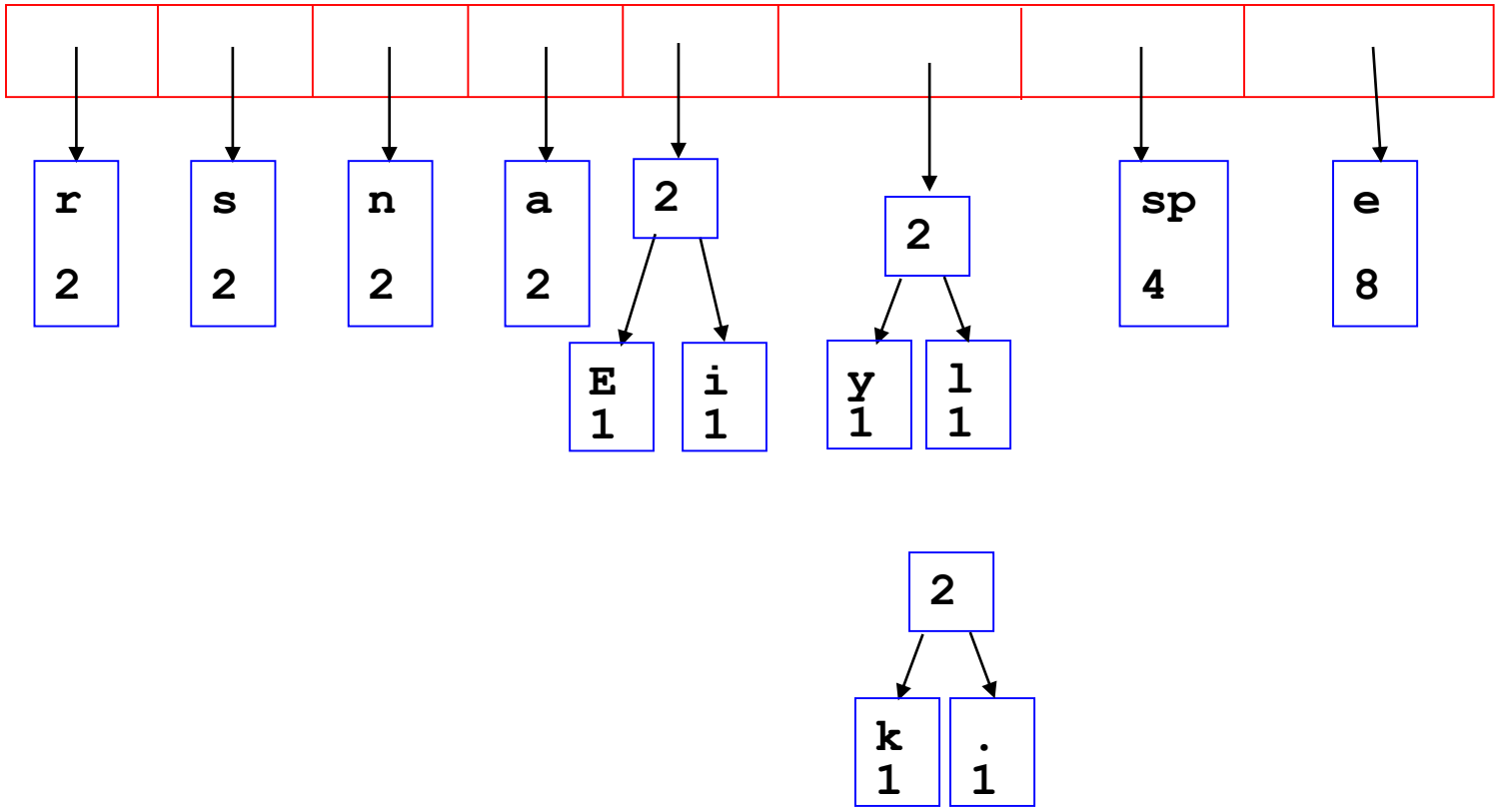
# Building a Tree



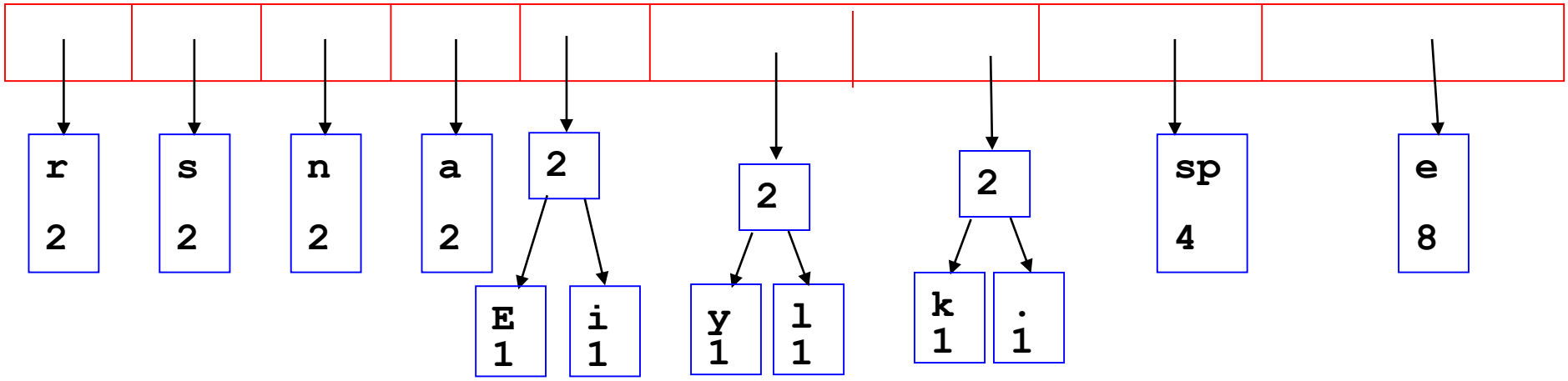
# Building a Tree



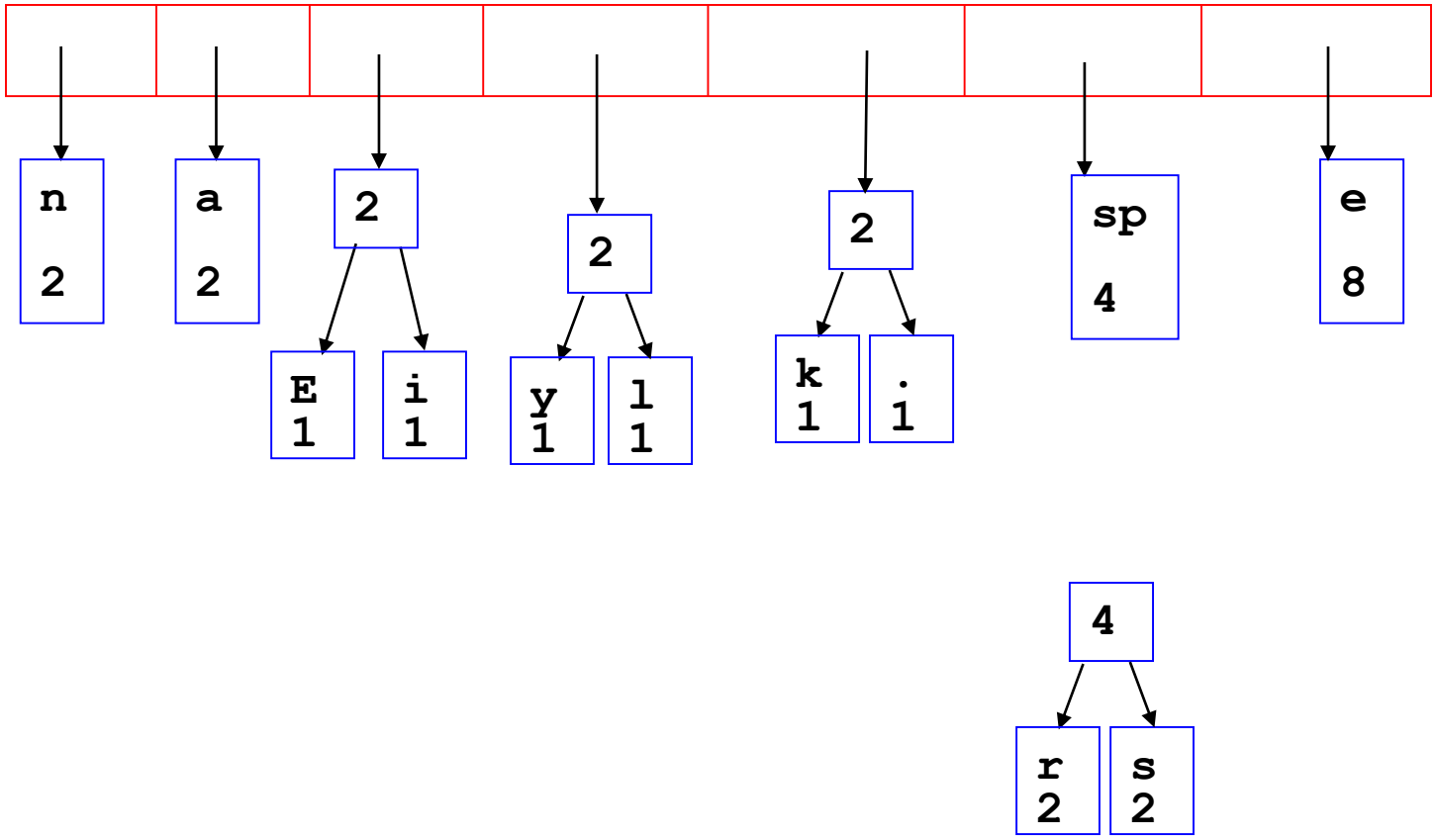
# Building a Tree



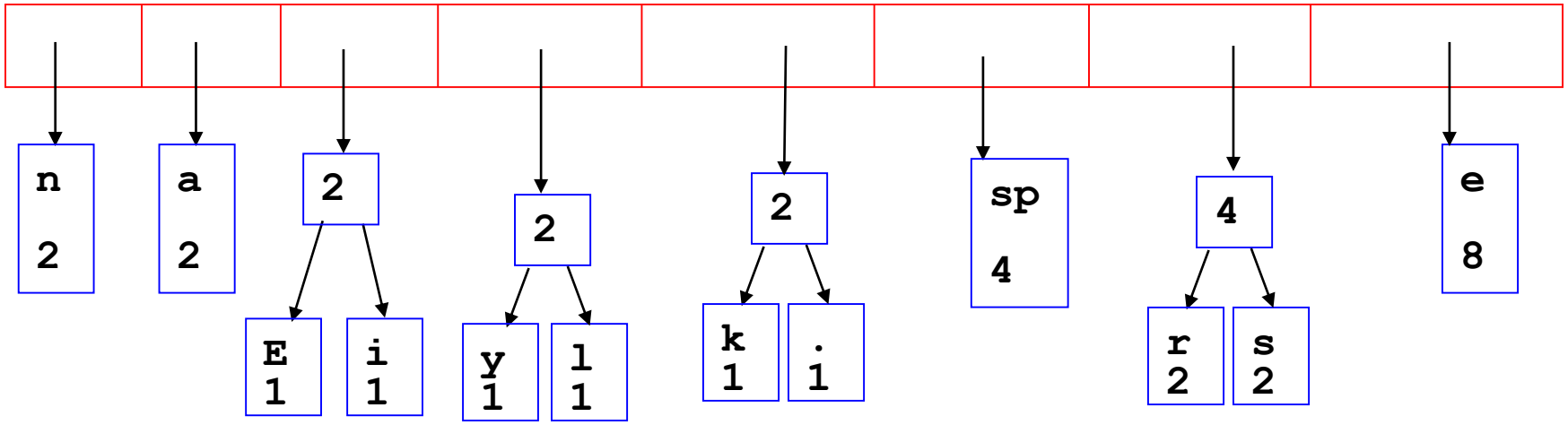
# Building a Tree



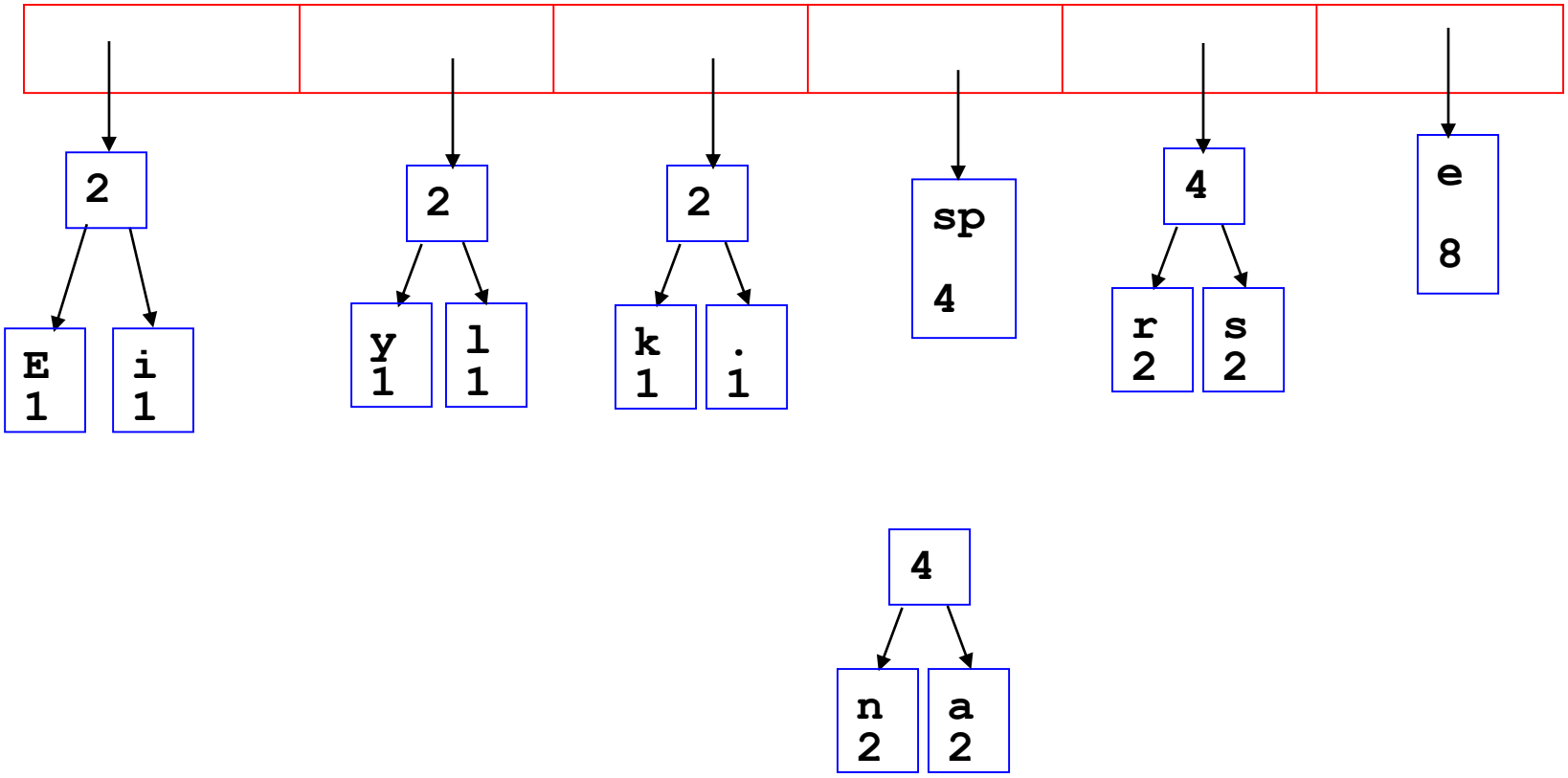
# Building a Tree



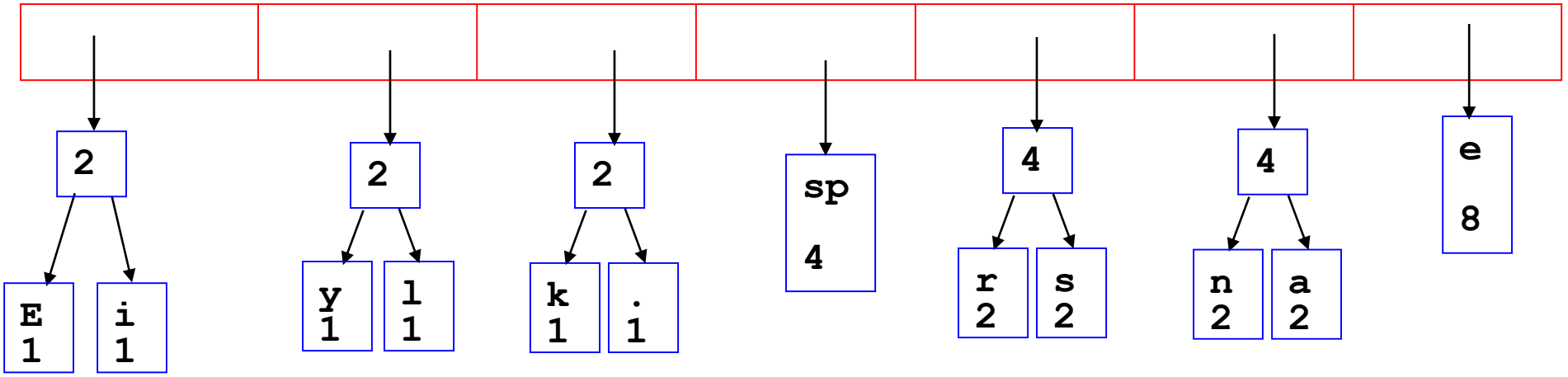
# Building a Tree



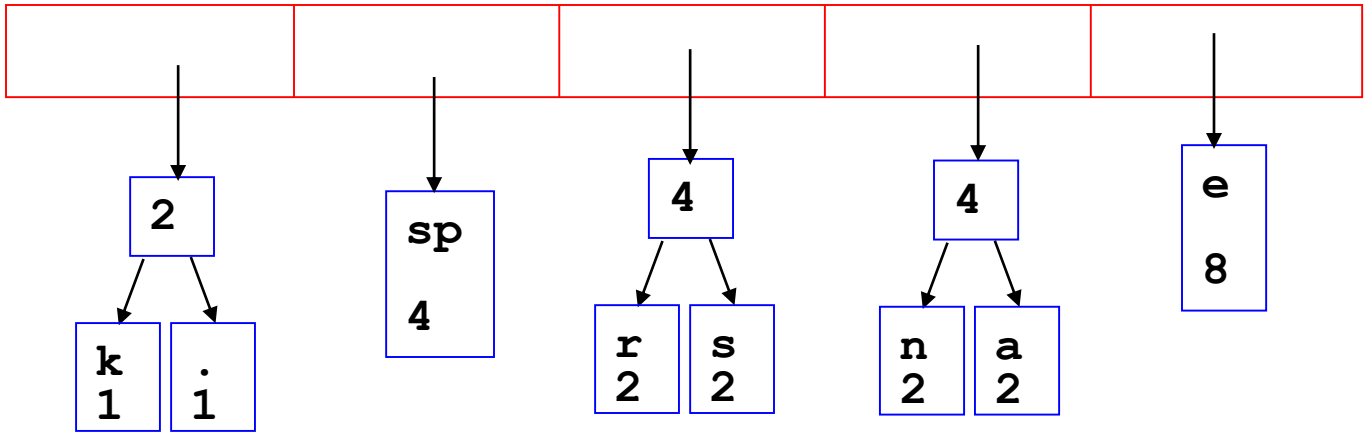
# Building a Tree

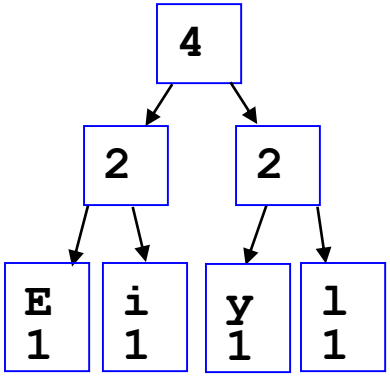


# Building a Tree

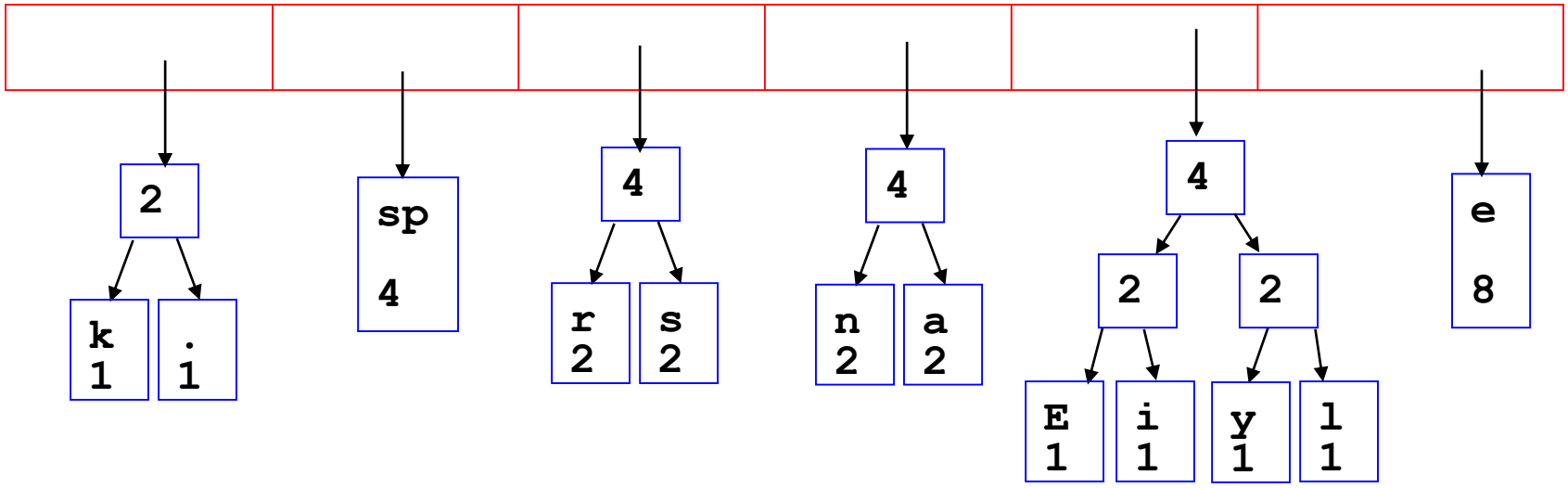


# Building a Tree

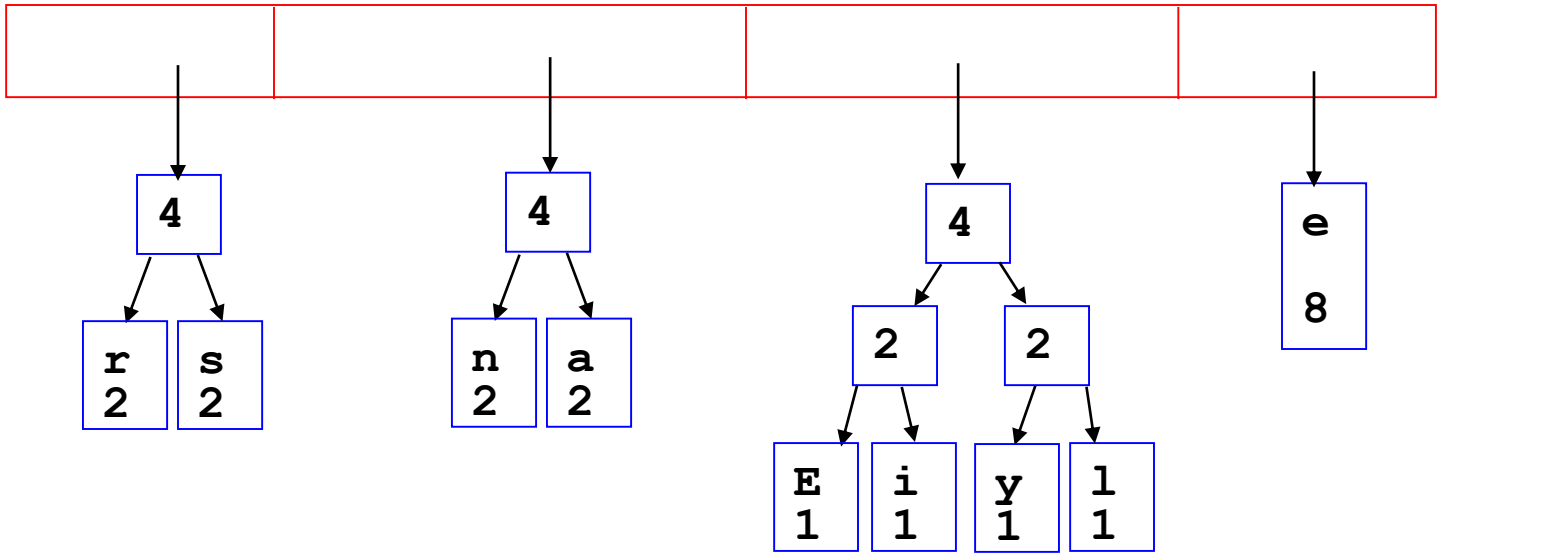


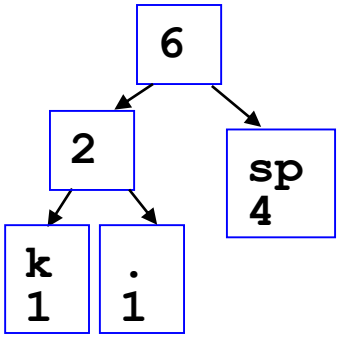



# Building a Tree

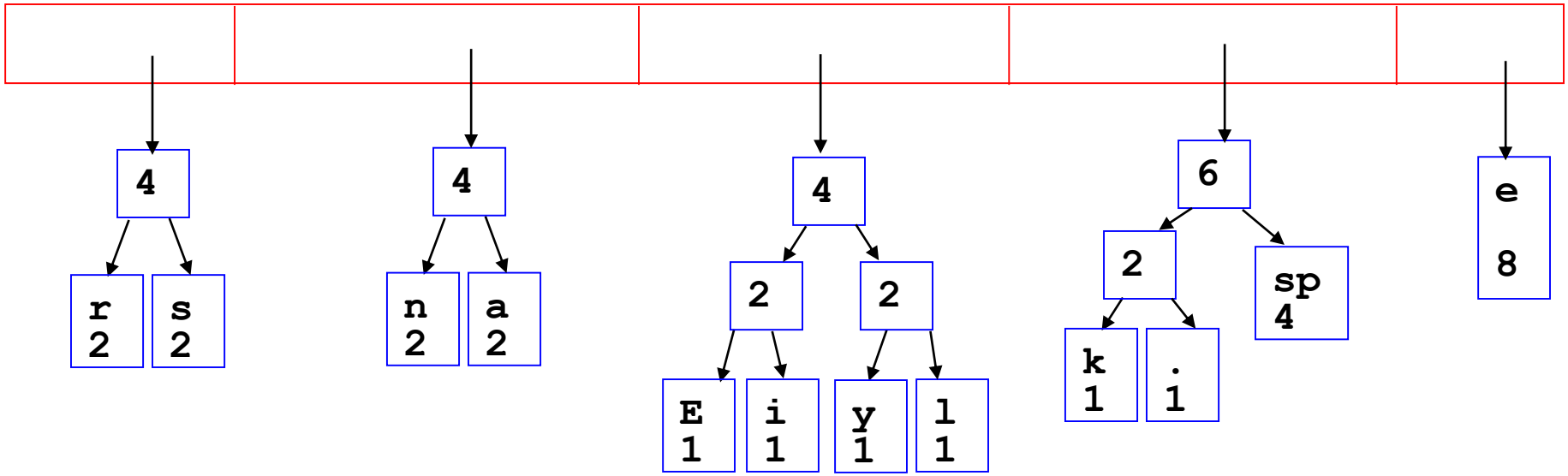


# Building a Tree

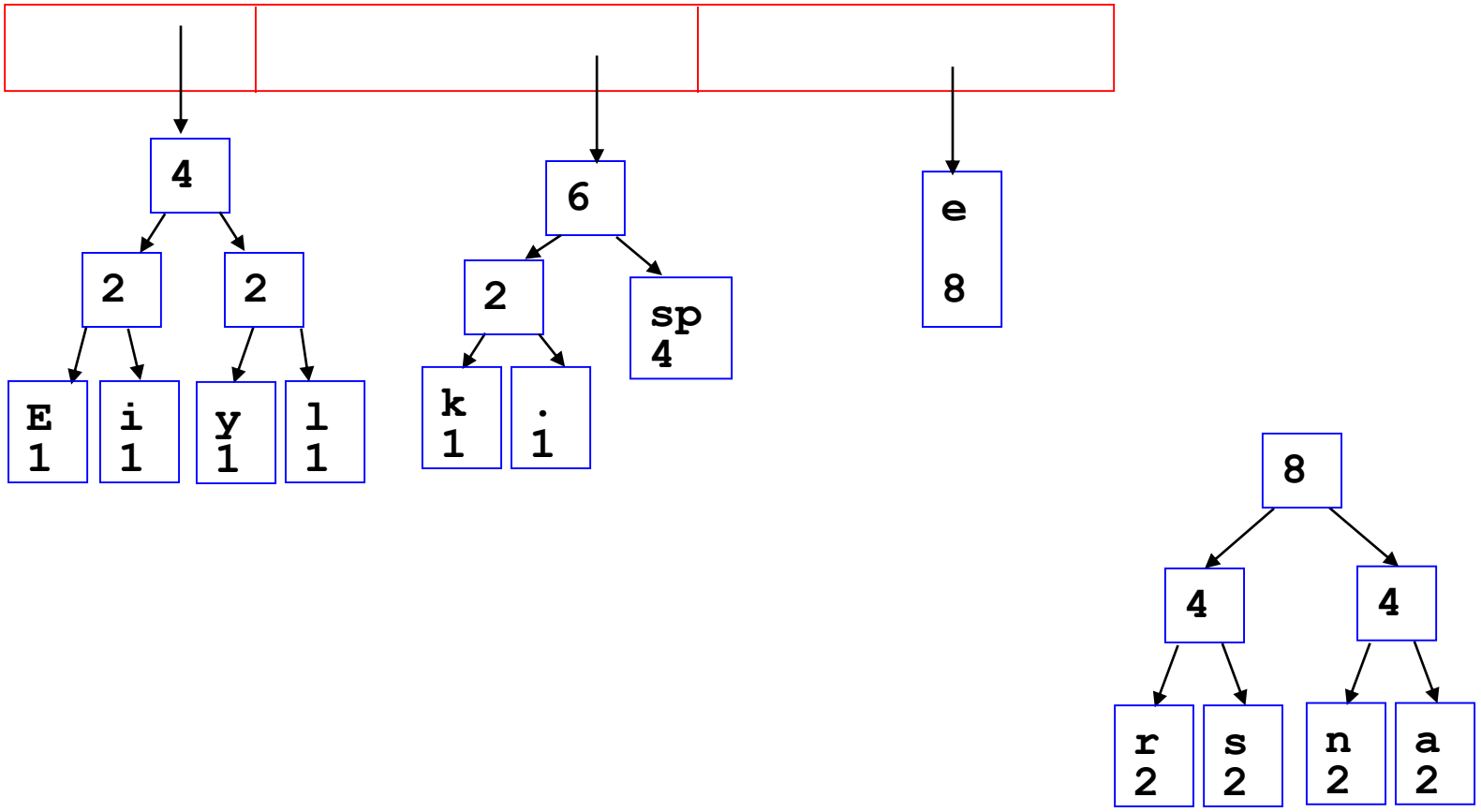


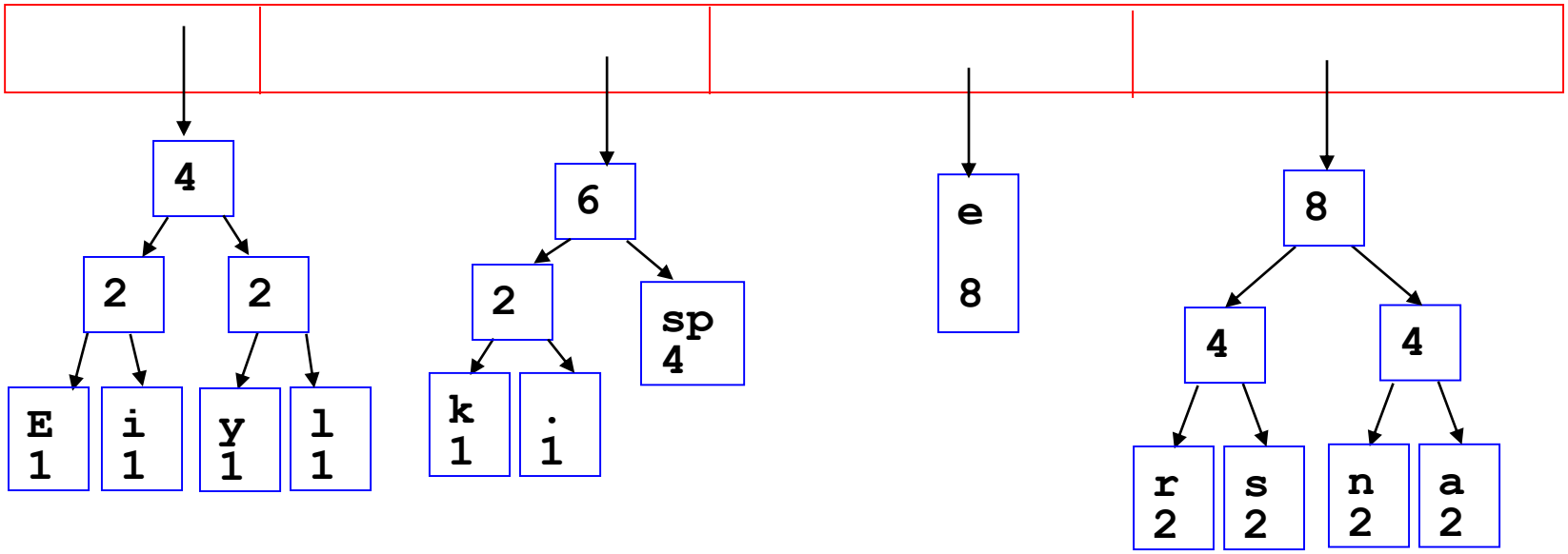
# Building a Tree



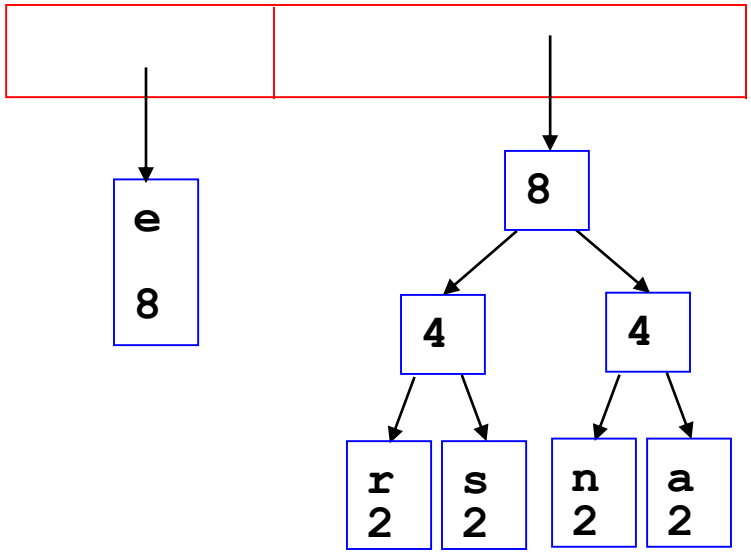
# Building a Tree

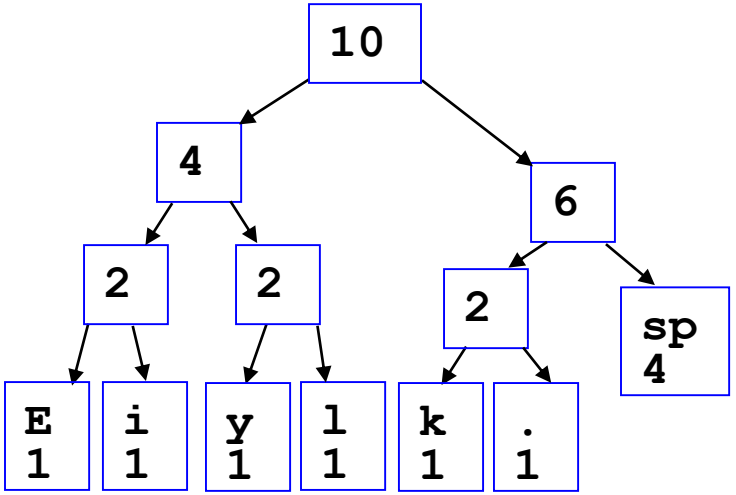


# Building a Tree

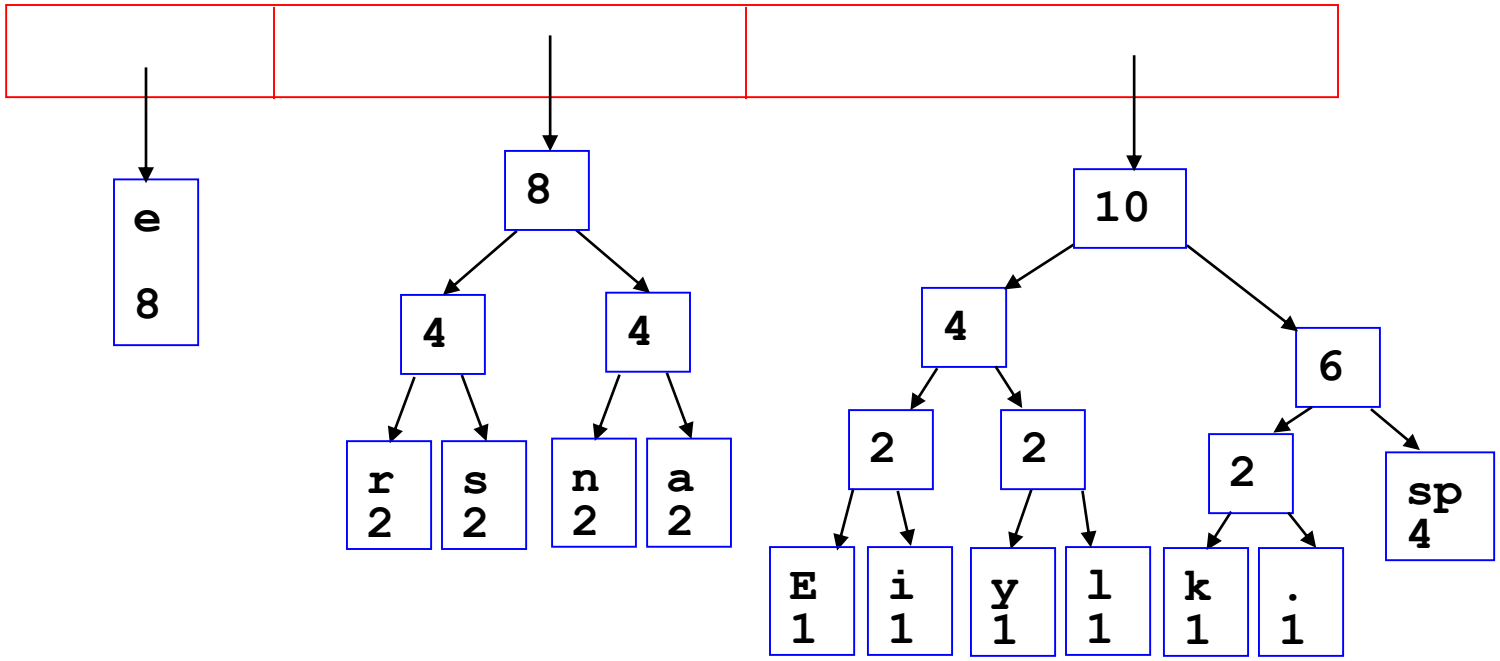


# Building a Tree

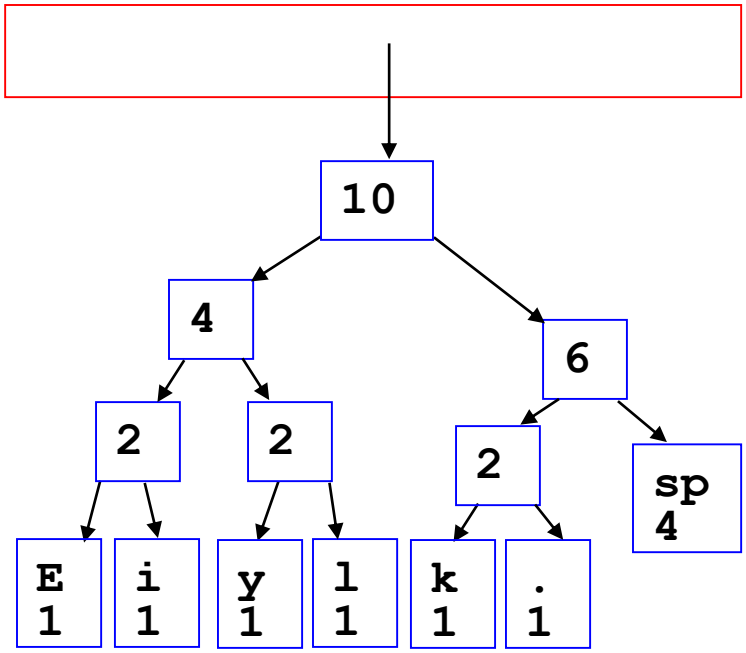


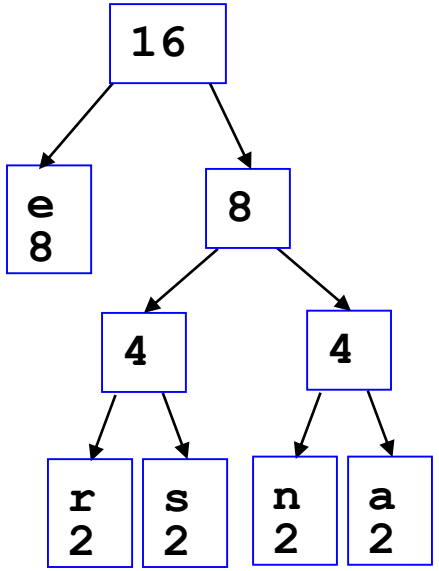



# Building a Tree

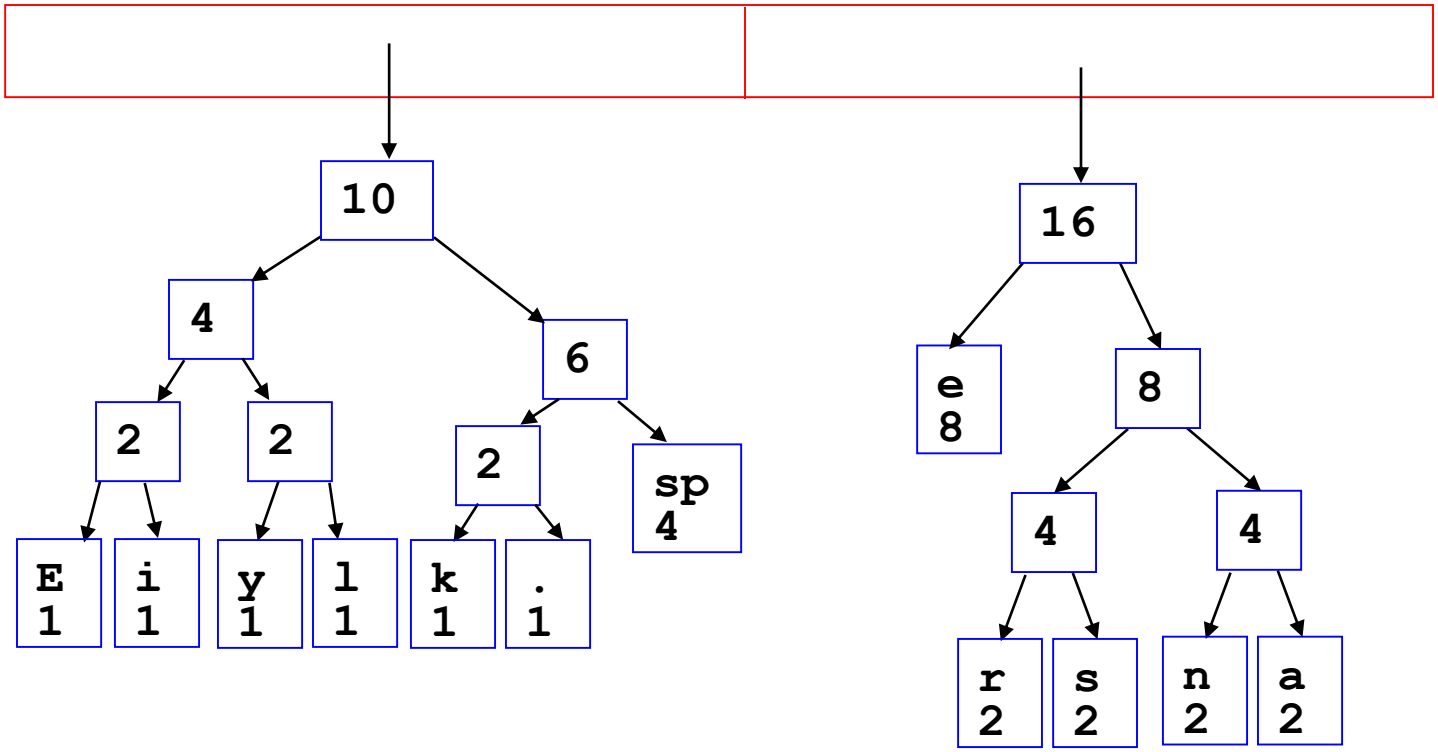


# Building a Tree

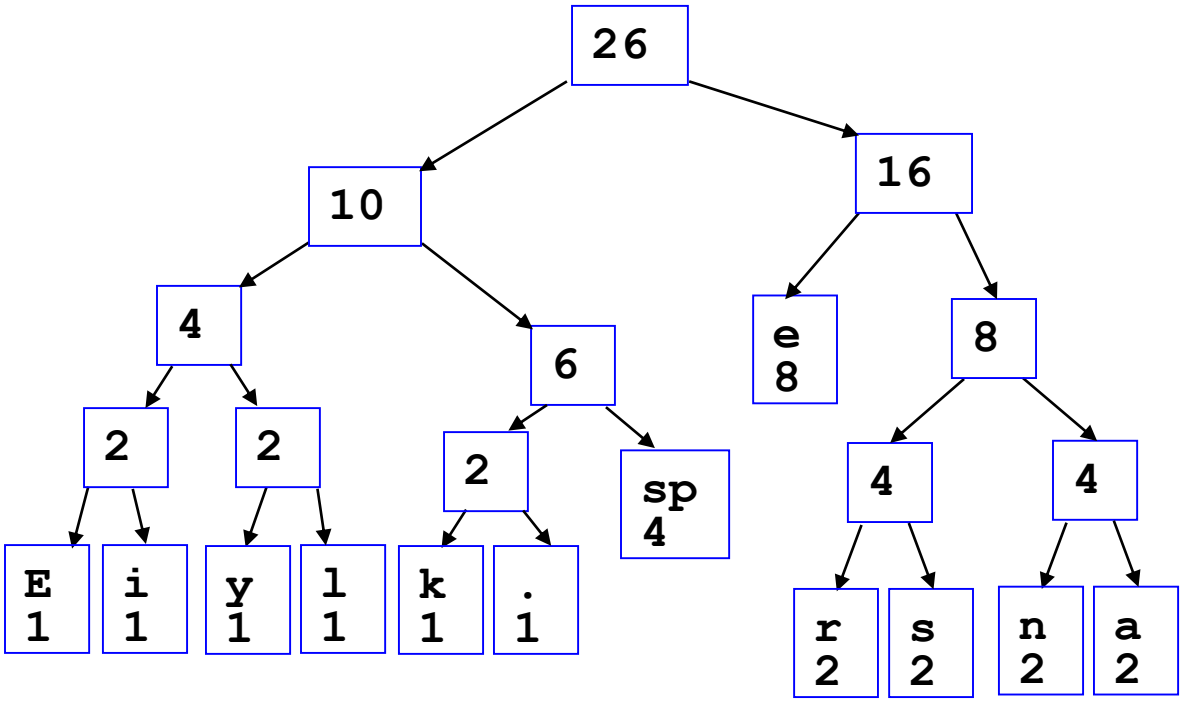


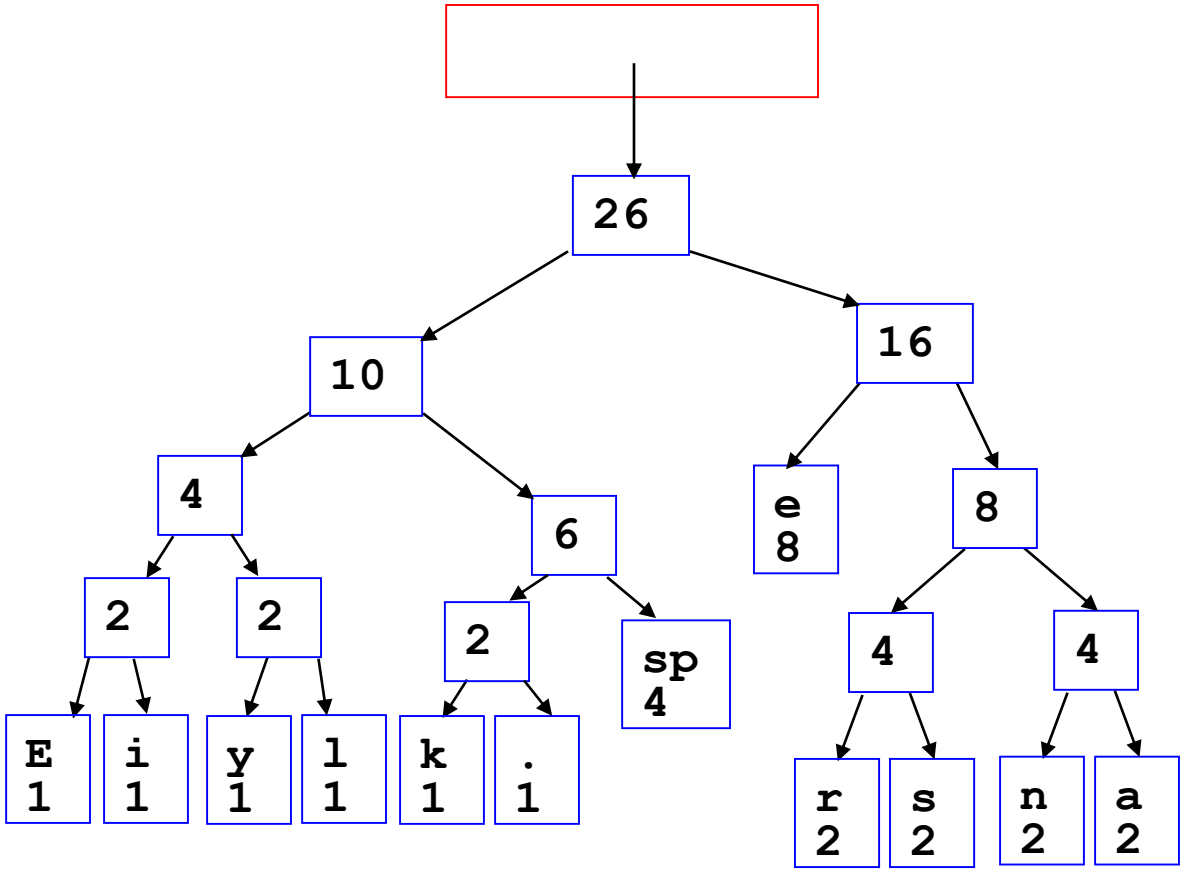
# Building a Tree



# Building a Tree



# Building a Tree



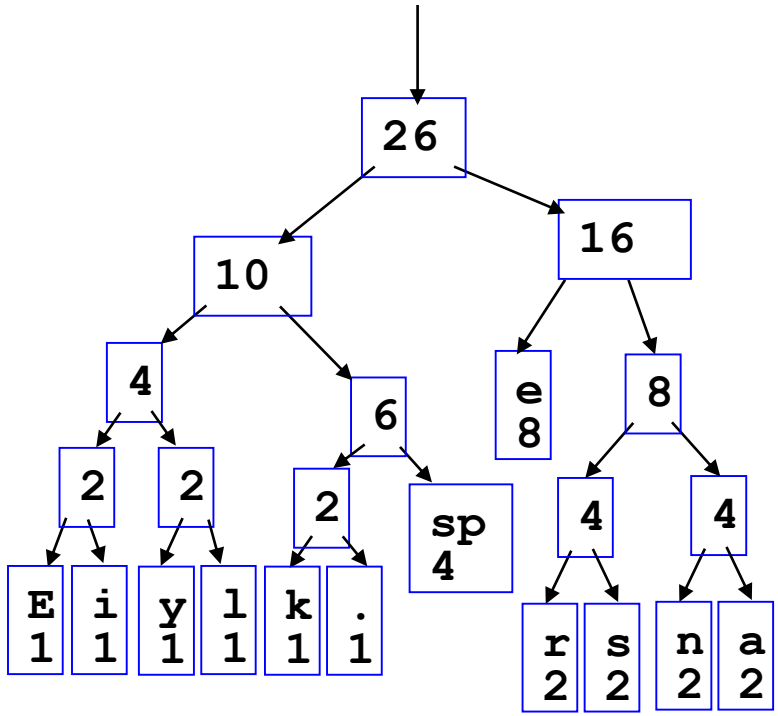

**After enqueueing this node there is only one node left in priority queue.**

# Building a Tree

- This tree contains the new code words for each character.
- Frequency of root node should equal number of characters in text.

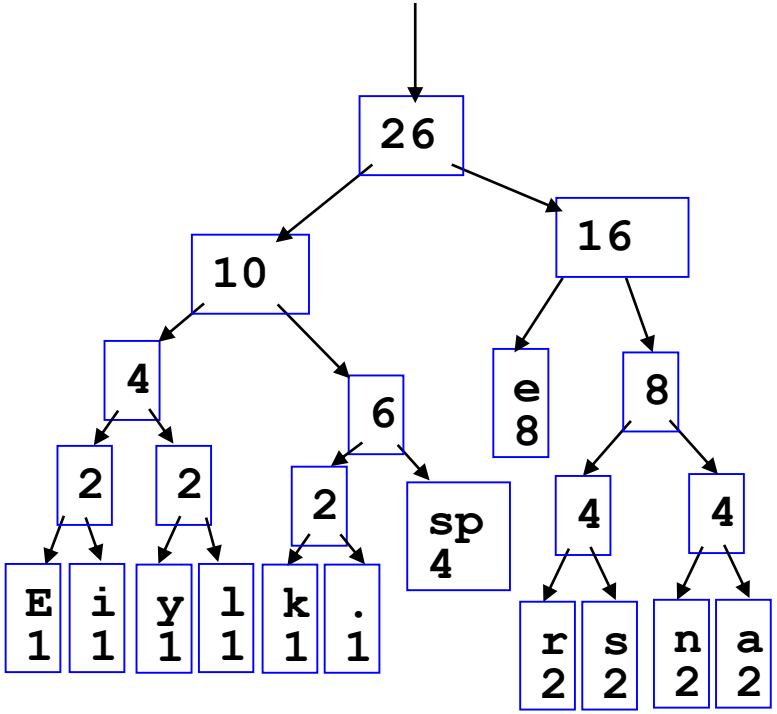
  *Eerie eyes seen near lake.*

  26 characters



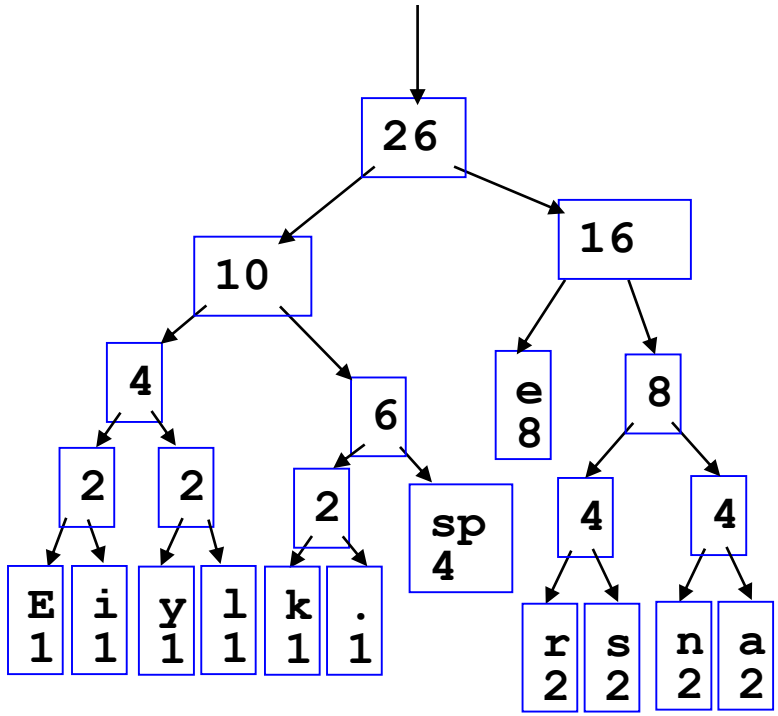
# Traverse Tree for Codes

- Perform a traversal of the tree to obtain new code words
  - Going left is a 0
  - Going right is a 1
  - Code word is only completed when a leaf node is reached



# Traverse Tree for Codes

| Char | Code |
|---|---|
| E | 0000 |
| i | 0001 |
| y | 0010 |
| l | 0011 |
| k | 0100 |
| . | 0101 |
| space | 011 |
| e | 10 |
| r | 1100 |
| s | 1101 |
| n | 1110 |
| a | 1111 |



# Encoding the File

- Rescan text and encode file using new code words

*Eerie eyes seen near lake.*

```
000010110000011001110001010110110100111110101111100011001111110100100101
```

| Char | Code |
|---|---|
| E | 0000 |
| i | 0001 |
| y | 0010 |
| l | 0011 |
| k | 0100 |
| . | 0101 |
| space | 011 |
| e | 10 |
| r | 1100 |
| s | 1101 |
| n | 1110 |
| a | 1111 |

# Encoding the File

- Have we made things any better?
  - 73 bits to encode the text
  - ASCII would take 8 * 26 = 208 bits

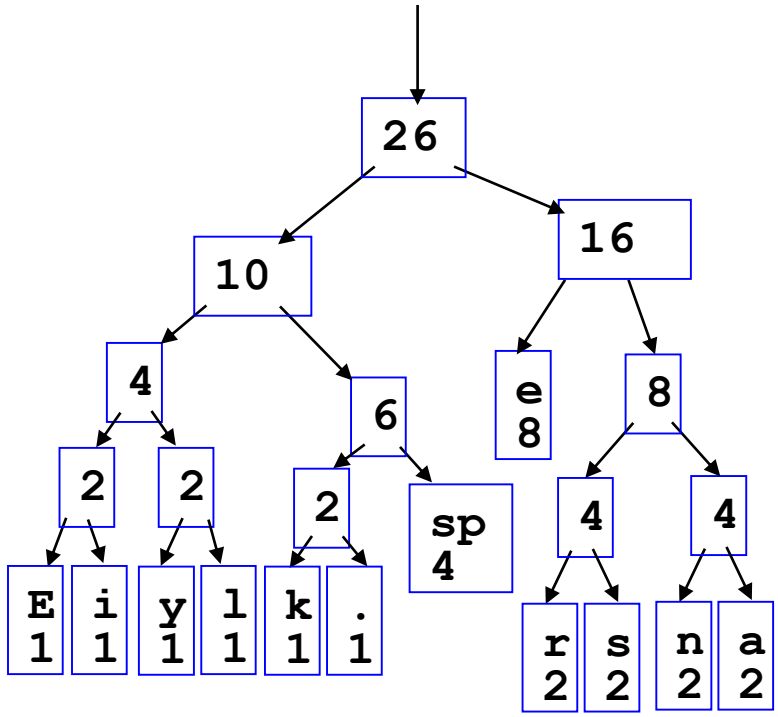
*Eerie eyes seen near lake.*

```
00001011000001100111000101011011010011111010111100011001111110100100101
```

| Char | Code |
|---|---|
| E | 0000 |
| i | 0001 |
| y | 0010 |
| l | 0011 |
| k | 0100 |
| . | 0101 |
| space | 011 |
| e | 10 |
| r | 1100 |
| s | 1101 |
| n | 1110 |
| a | 1111 |

# Encoding the File

- Why is there no need for a separator character?
- Prefix property
  - No code is a prefix of another code




*Eerie eyes seen near lake.*

```
000010110000011001110001010110110100111110101111100011001111110100100101
```

# Decoding the File

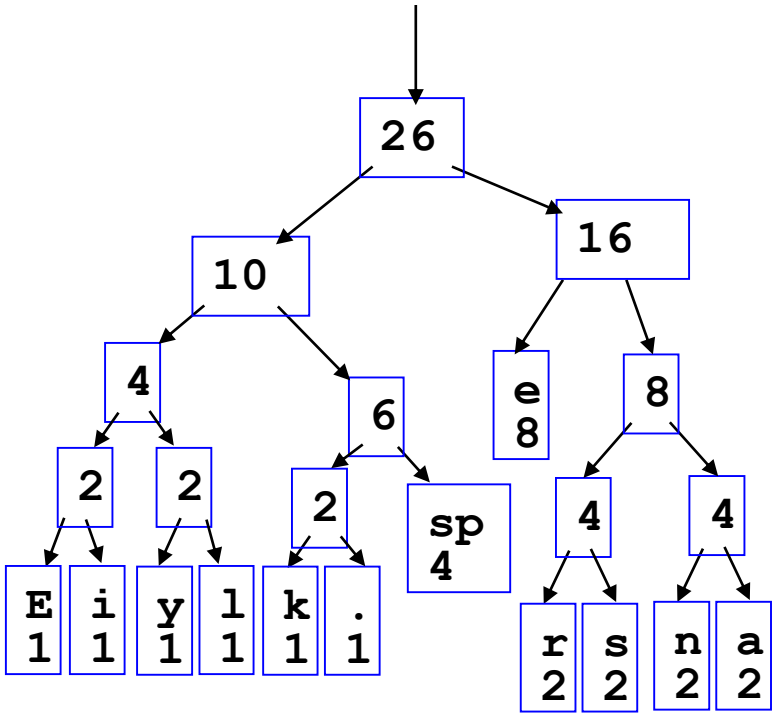
- How does receiver know what the codes are?
- Two solutions
  - Tree constructed for each text file
    - Codes customized for each file
    - Big hit on compression, especially for smaller files
  - Tree predetermined
    - Based on statistical analysis of general text files or file types

# Decoding the File

- Once receiver has the tree, it scans incoming bit stream
  - 0 ⇒ go left
  - 1 ⇒ go right

```
10100011011111011110111111110000110101
```

*eel snarl.*



# Summary

- Huffman coding is a technique used to compress files
- Uses statistical coding
  - more frequently used symbols have shorter code words
- Uses two data structures
  - Priority queue
  - Binary tree