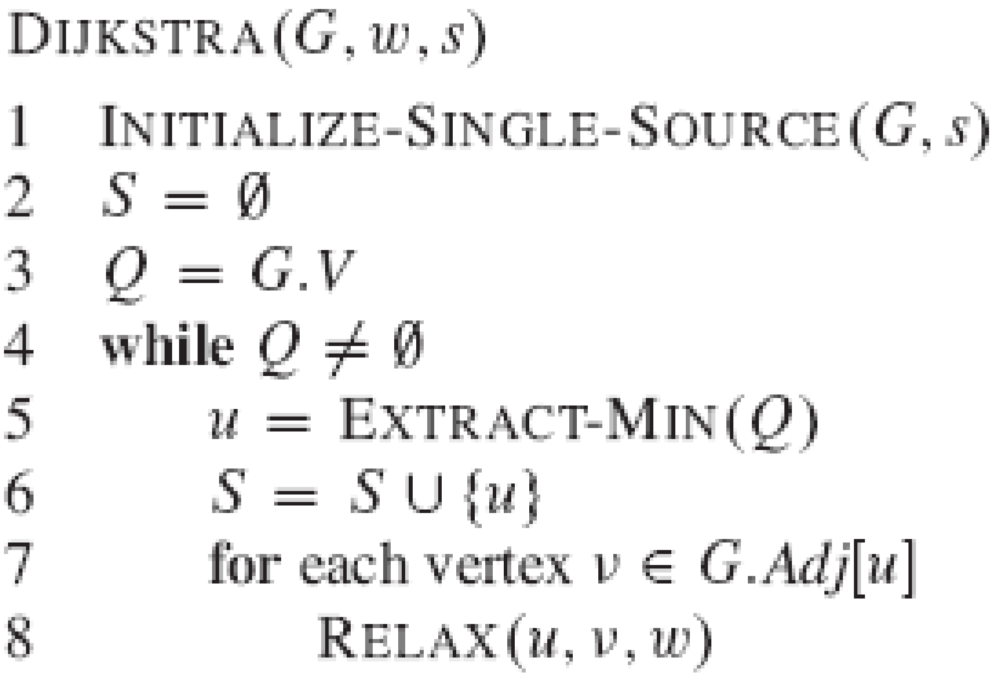
CSE408
Dijkstra,Huffmancoding

Lecture # 27

Dijkstra’s Algorithm

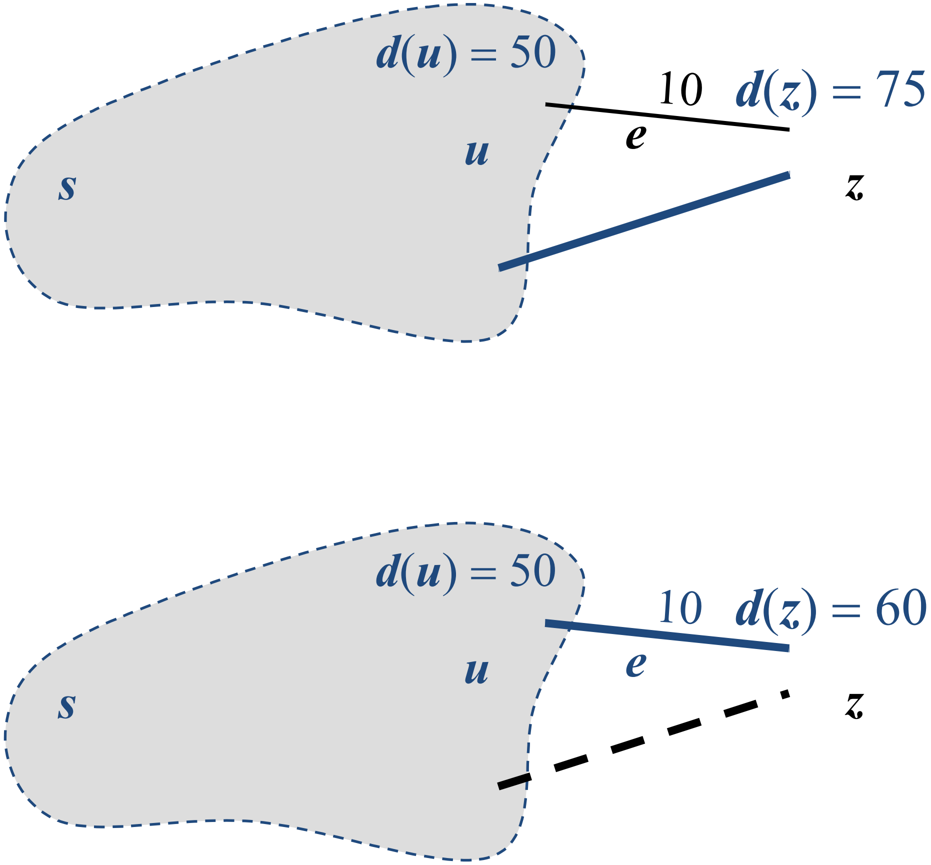


```
DIJKSTRA$(G, w, s)$
    INITIALIZE-SINGLE-SOURCE$(G, s)$
    $S = \emptyset$
    $Q = G.V$
    while $Q \neq \emptyset$
        $u =$ EXTRACT-MIN$(Q)$
        $S = S \cup \{u\}$
        for each vertex $v \in G.Adj[u]$
            RELAX$(u, v, w)$
```

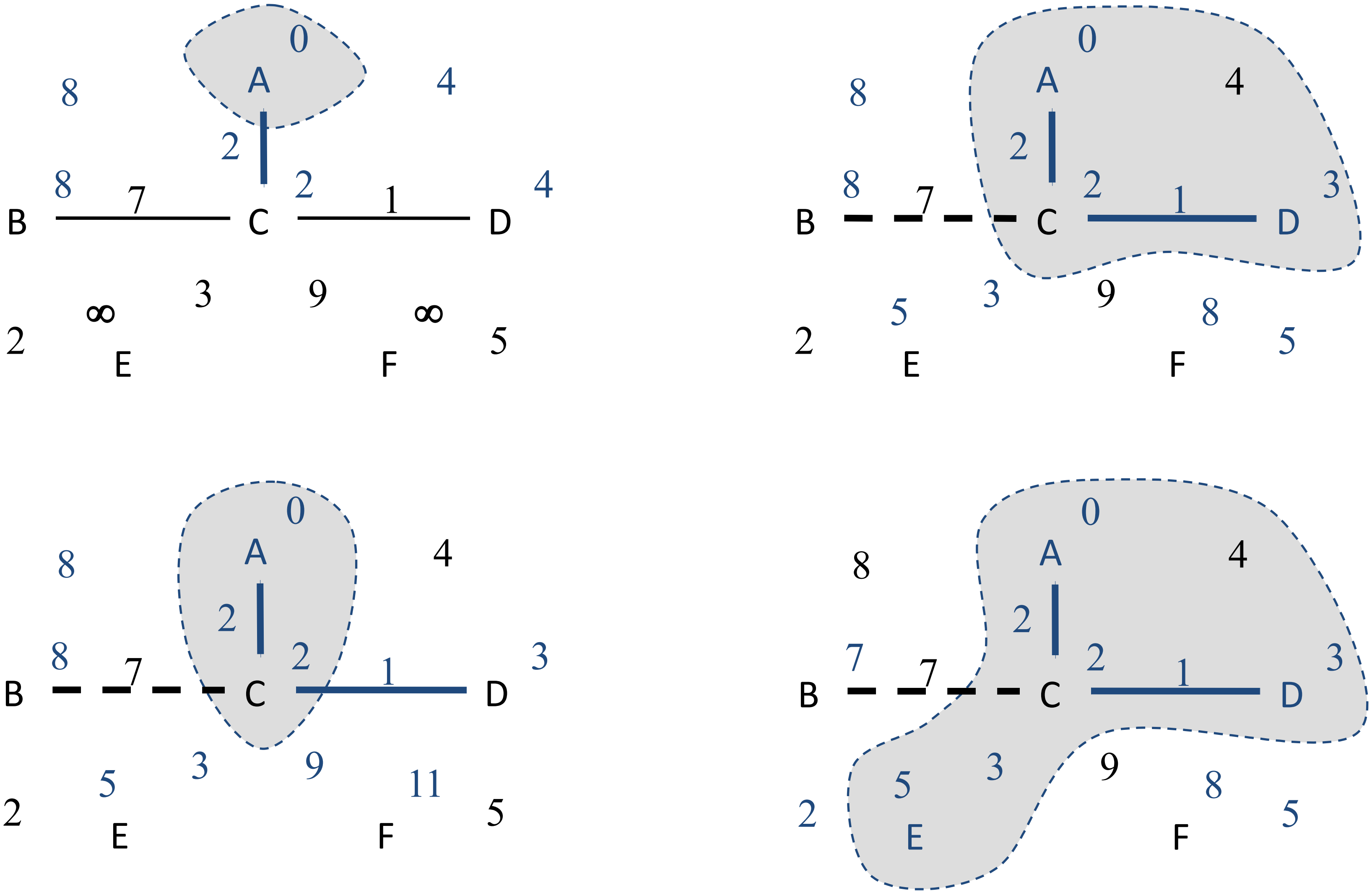
Edge Relaxation

- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

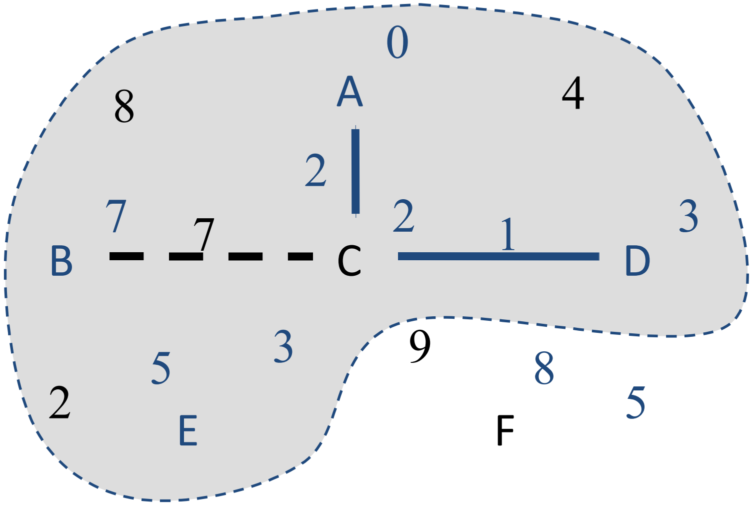
$$d(z) \leftarrow \min\{d(z), d(u) + weight(e)\}$$

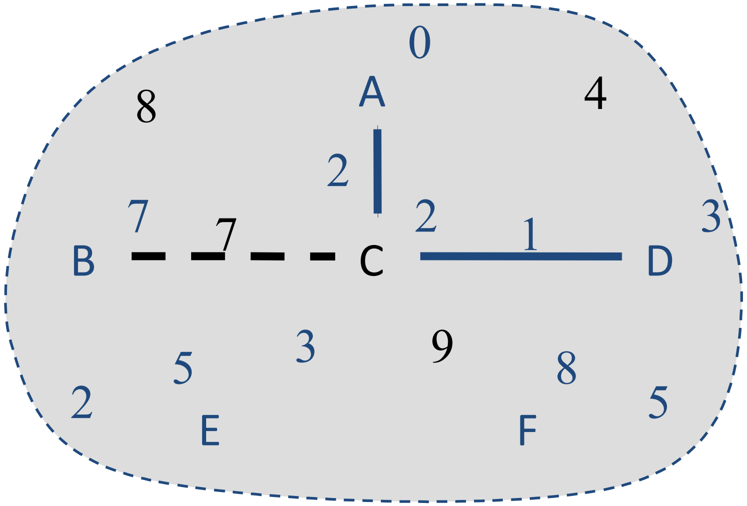


Example


0
A
8
4
2
8
7
2
1
4
B
C
D
3
9
∞
∞
2
5
E
F
0
A
4
8
2
8
7
2
1
3
B
C
D
3
9
5
8
2
5
E
F
0
A
8
4
2
8
7
2
1
3
B
C
D
3
9
5
11
2
5
E
F
0
A
8
4
2
7
7
2
1
3
B
C
D
3
9
5
8
2
5
E
F

Example (cont.)





Dijkstra’s Algorithm

- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - *insert*(*k*,*e*) returns a locator
  - *replaceKey*(*l*,*k*) changes the key of an item
- We store two labels with each vertex:
  - Distance (d(v) label)
  - locator in priority queue

```
Algorithm DijkstraDistances(G, s)
    Q ← new heap-based priority queue
    for all  v ∈ G.vertices()
        if  v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        l ← Q.insert(getDistance(v), v)
      setLocator(v,l)
  while  ¬Q.isEmpty()
      u ← Q.removeMin()
      for all  e ∈ G.incidentEdges(u)
          { relax edge e }
          z ← G.opposite(u,e)
          r ← getDistance(u) + weight(e)
          if  r < getDistance(z)
              setDistance(z,r)

Q.replaceKey(getLocator(z),r)
```

Analysis

- Graph operations
  - Method incidentEdges is called once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

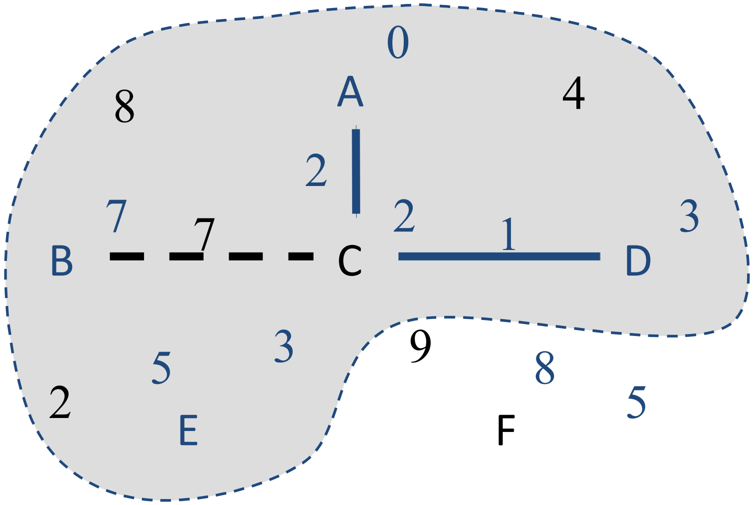
Extension

- Using the template method pattern, we can extend Dijkstra’s algorithm to return a tree of shortest paths from the start vertex to all other vertices
- We store with each vertex a third label:
  - parent edge in the shortest path tree
- In the edge relaxation step, we update the parent label

```
Algorithm DijkstraShortestPathsTree($G, s$)

      ...

    for all $v \in G.vertices()$
      ...
         setParent($v, \emptyset$)
      ...

      for all $e \in G.incidentEdges(u)$
          { relax edge $e$ }
          $z \leftarrow G.opposite(u,e)$
          $r \leftarrow getDistance(u) + weight(e)$
          if $r < getDistance(z)$
              setDistance($z,r$)
              setParent($z,e$)


  Q.replaceKey(getLocator($z$),$r$)
```

Why Dijkstra’s Algorithm Works

- Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn’t find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct.
  - But the edge (D,F) was **relaxed** at that time!
  - Thus, so long as d(F)≥d(D), F’s distance cannot be wrong. That is, there is no wrong vertex.



Purpose of Huffman Coding

- Proposed by Dr. David A. Huffman in 1952
  - *"A Method for the Construction of Minimum Redundancy Codes"*
- Applicable to many forms of data transmission
  - Our example: text files

The Basic Algorithm

- Huffman coding is a form of statistical coding
- 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

The Basic Algorithm

- Any savings in tailoring codes to frequency of character?
- Code word lengths are no longer fixed like ASCII.
- Code word lengths vary and will be shorter for the more frequently used characters.

The (Real) Basic Algorithm

1. Scan text to be compressed and tally occurrence of all characters.
2. Sort or prioritize characters based on number of occurrences in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.

Building a Tree Scan the original text

- Consider the following short text:

  Eerie eyes seen near lake.

- Count up the occurrences of all characters in the text

Building a Tree Scan the original text

Eerie eyes seen near lake.

- What characters are present?

E e r i space

y s n a r l k .

Building a Tree Scan the original 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		

Building a Tree 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

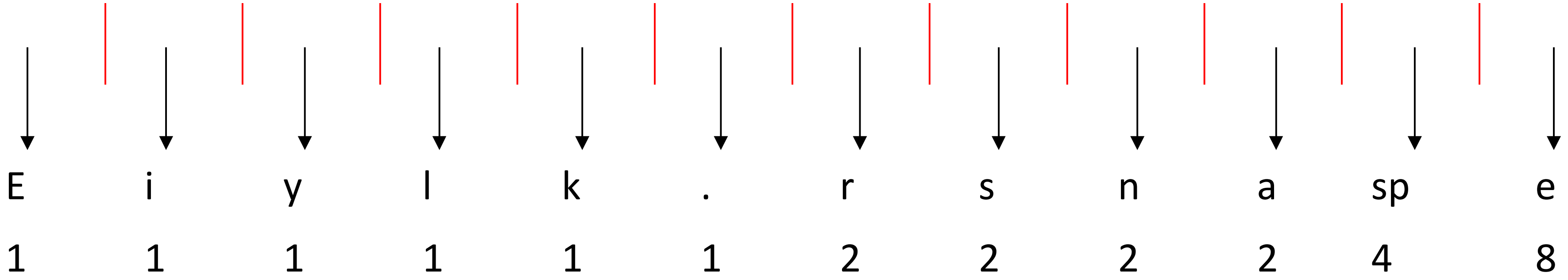
Building a Tree Prioritize characters

- Uses binary tree nodes

```
public class HuffNode
{
   public char myChar;
   public int myFrequency;
   public HuffNode myLeft, myRight;
}

priorityQueue myQueue;
```

Building a Tree

- The queue after inserting all nodes



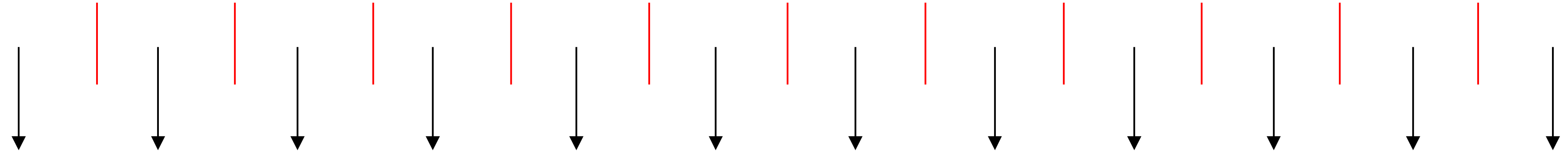
E	i	y	l	k	.	r	s	n	a	sp	e
1	1	1	1	1	1	2	2	2	2	4	8

- Null Pointers are not shown

Building a Tree

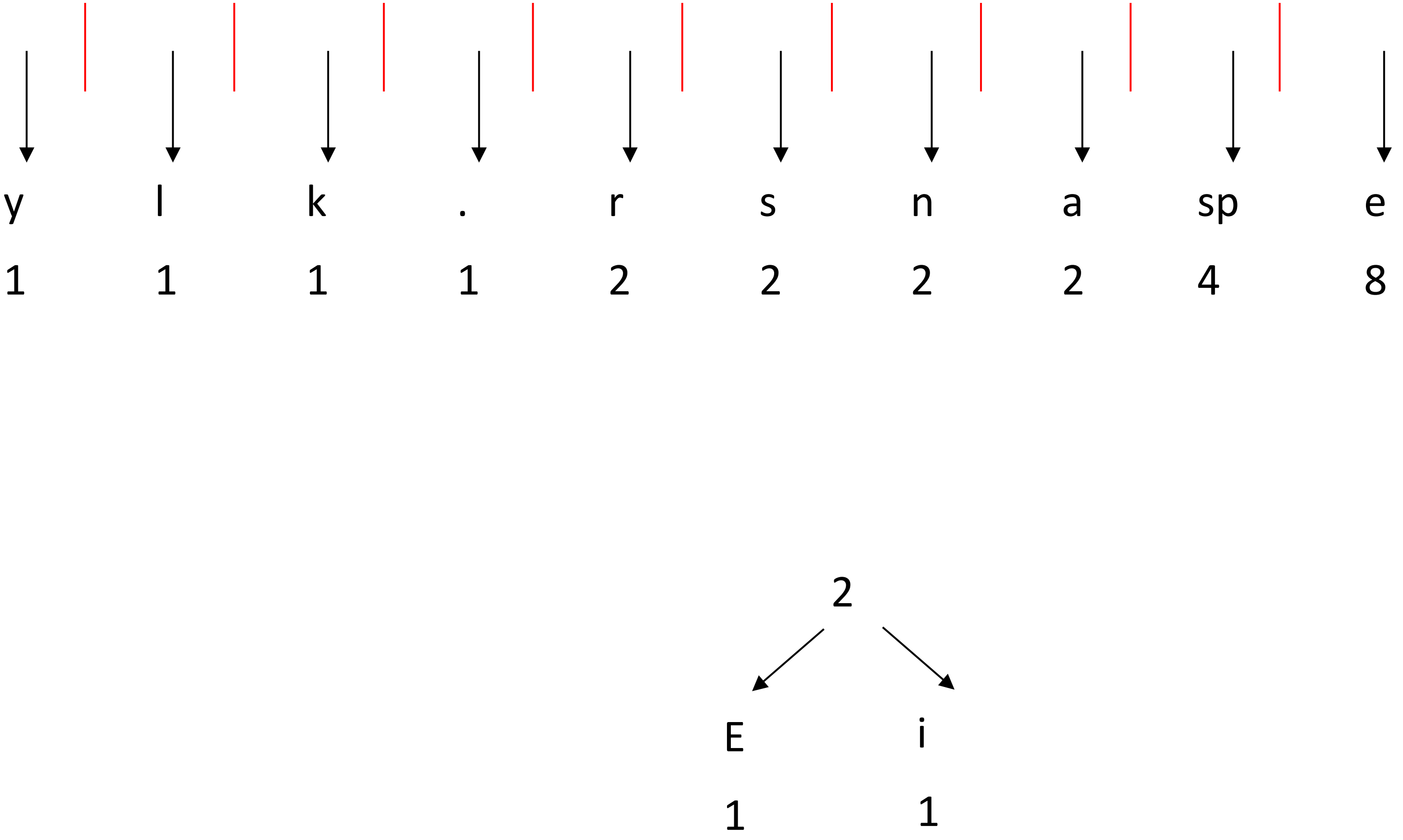
- While priority queue contains two or more nodes
  - Create new node
  - Dequeue node and make it left subtree
  - Dequeue next node and make it right subtree
  - Frequency of new node equals sum of frequency of left and right children
  - Enqueue new node back into queue

Building a Tree



E	i	y	l	k	.	r	s	n	a	sp	e
1	1	1	1	1	1	2	2	2	2	4	8

Building a Tree



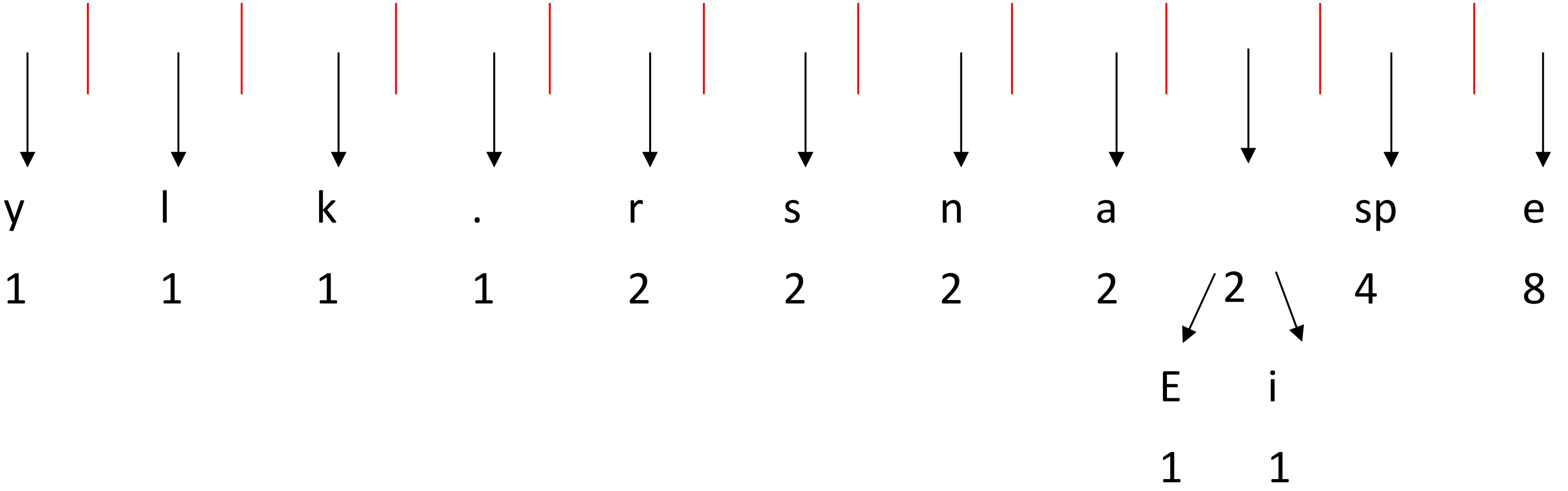
y	l	k	.	r	s	n	a	sp	e
1	1	1	1	2	2	2	2	4	8

2

E i

1 1

Building a Tree



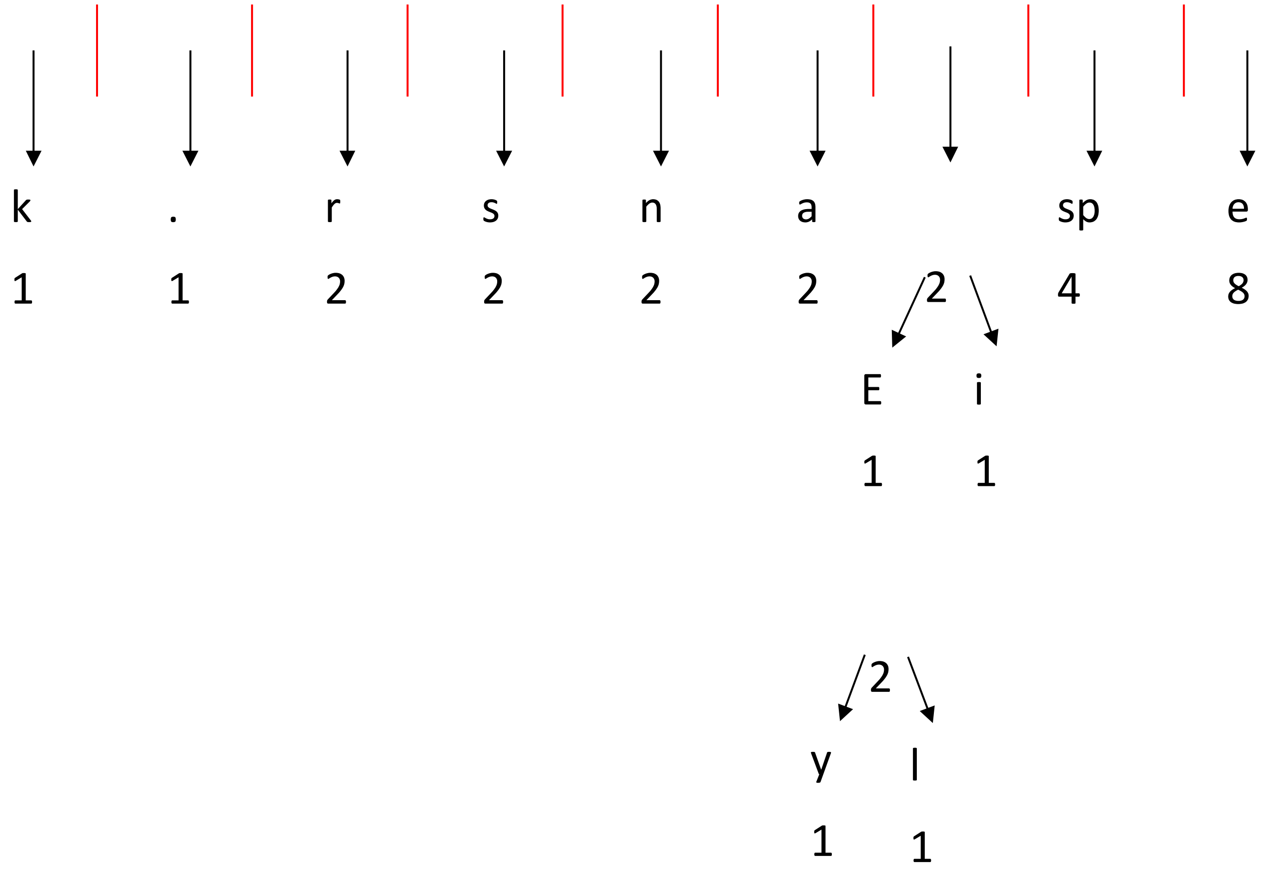
y l k . r s n a sp e

1 1 1 1 2 2 2 2 2 4 8

E i

1 1

Building a Tree



k . r s n a sp e

1 1 2 2 2 2 2 4 8

E i

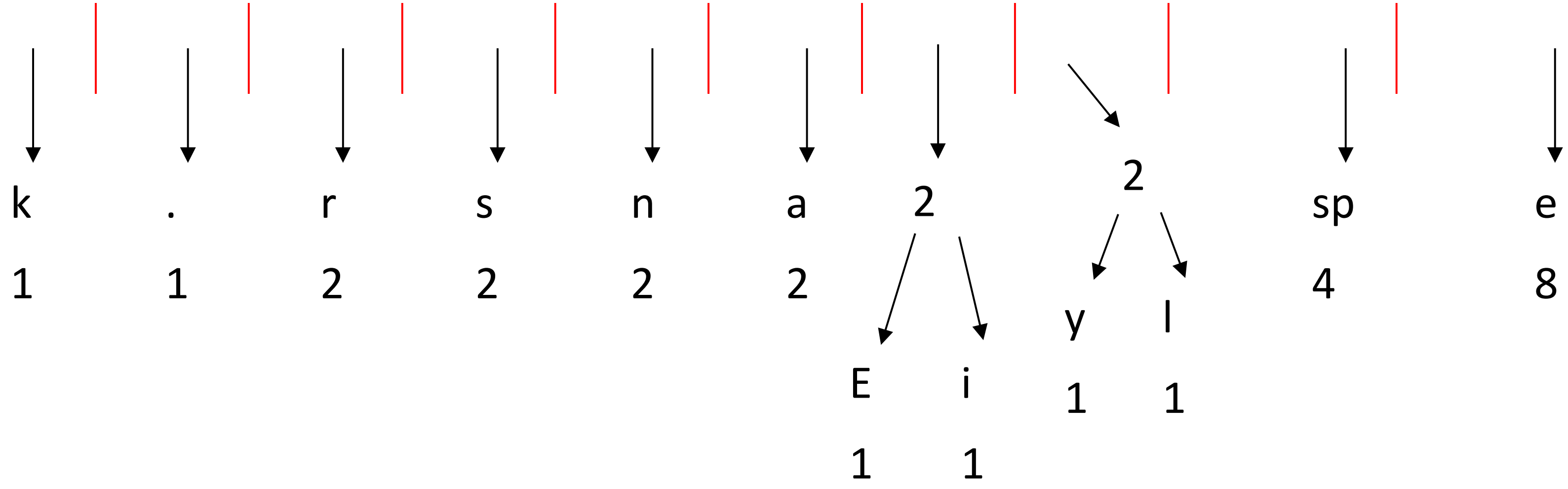
1 1

2

y l

1 1

Building a Tree



k 1

. 1

r 2

s 2

n 2

a 2

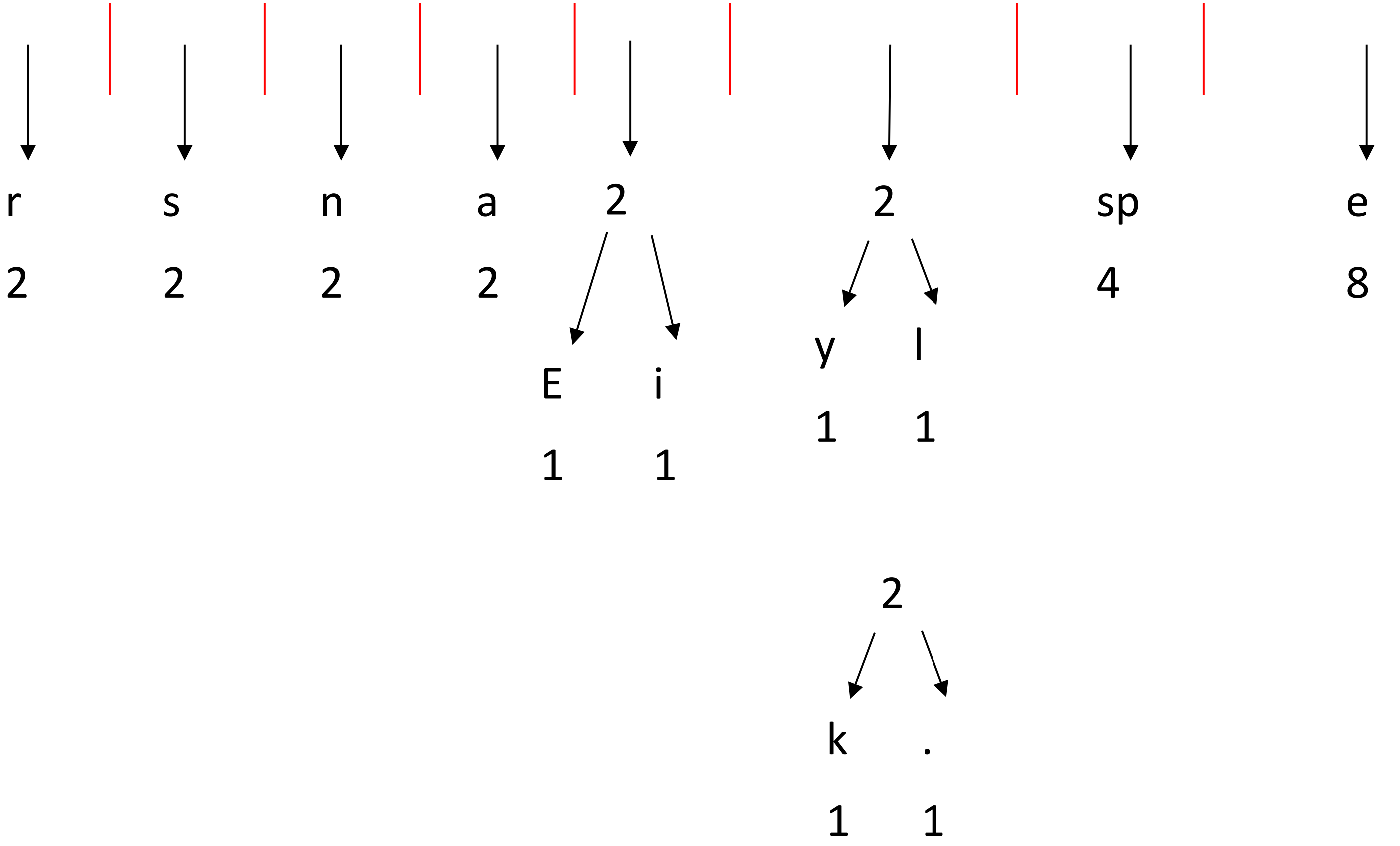
2
E 1
i 1

2
y 1
l 1

sp 4

e 8

Building a Tree



r 2

s 2

n 2

a 2

2 — E 1, i 1

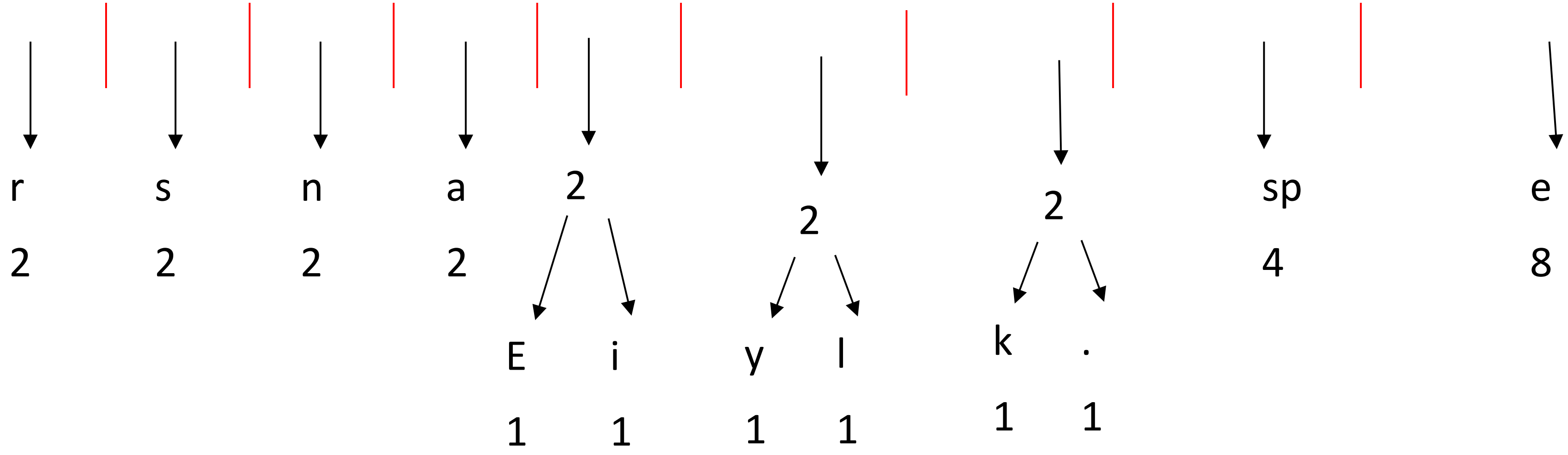
2 — y 1, l 1

sp 4

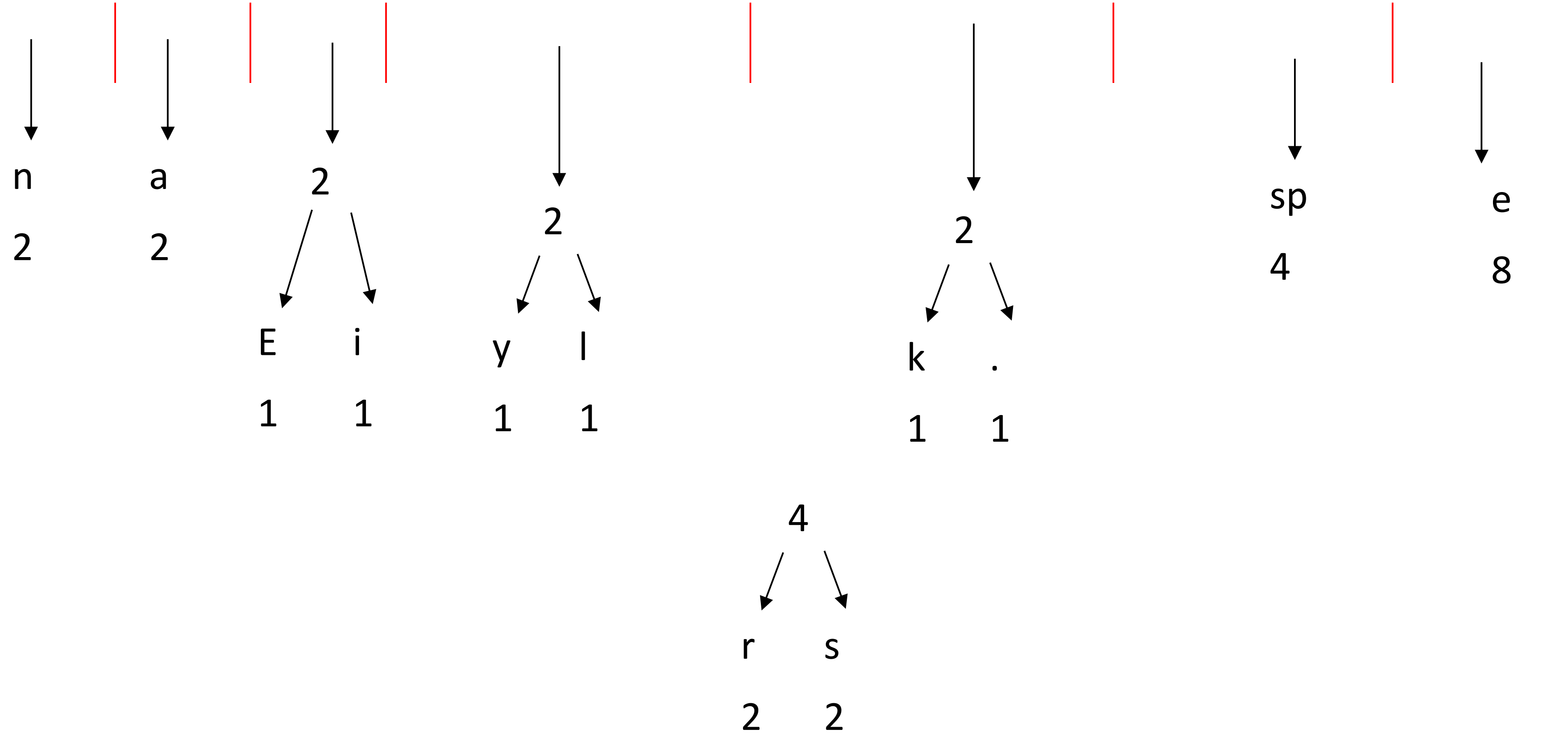
e 8

2 — k 1, . 1

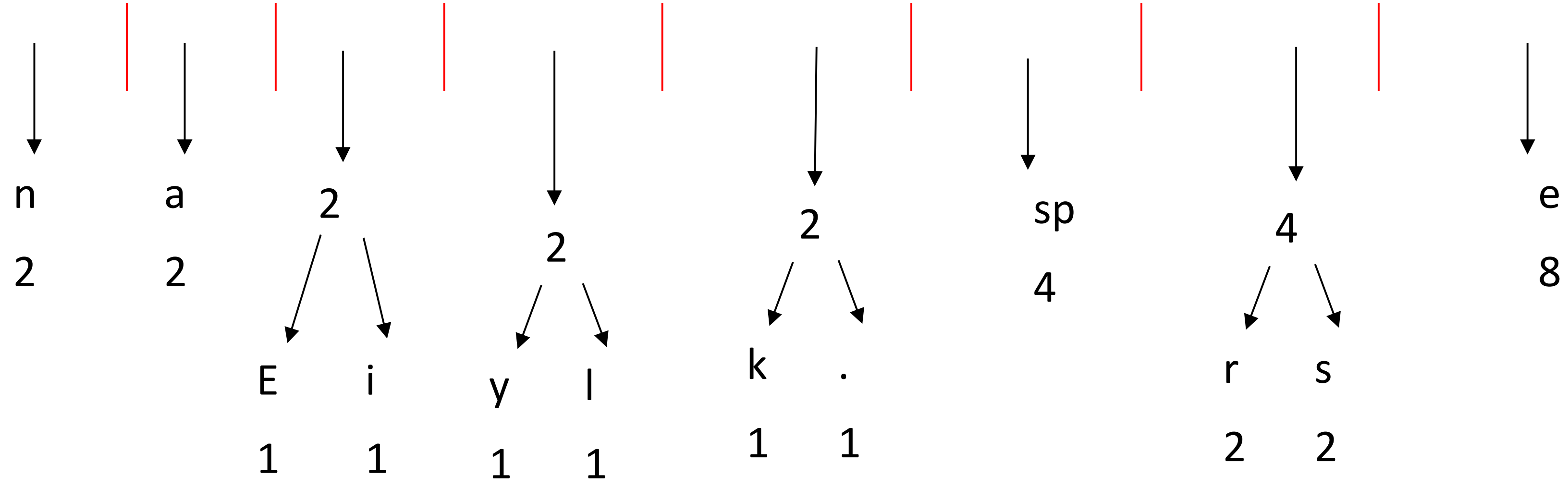
Building a Tree



Building a Tree



Building a Tree



n 2

a 2

2
E 1 i 1

2
y 1 l 1

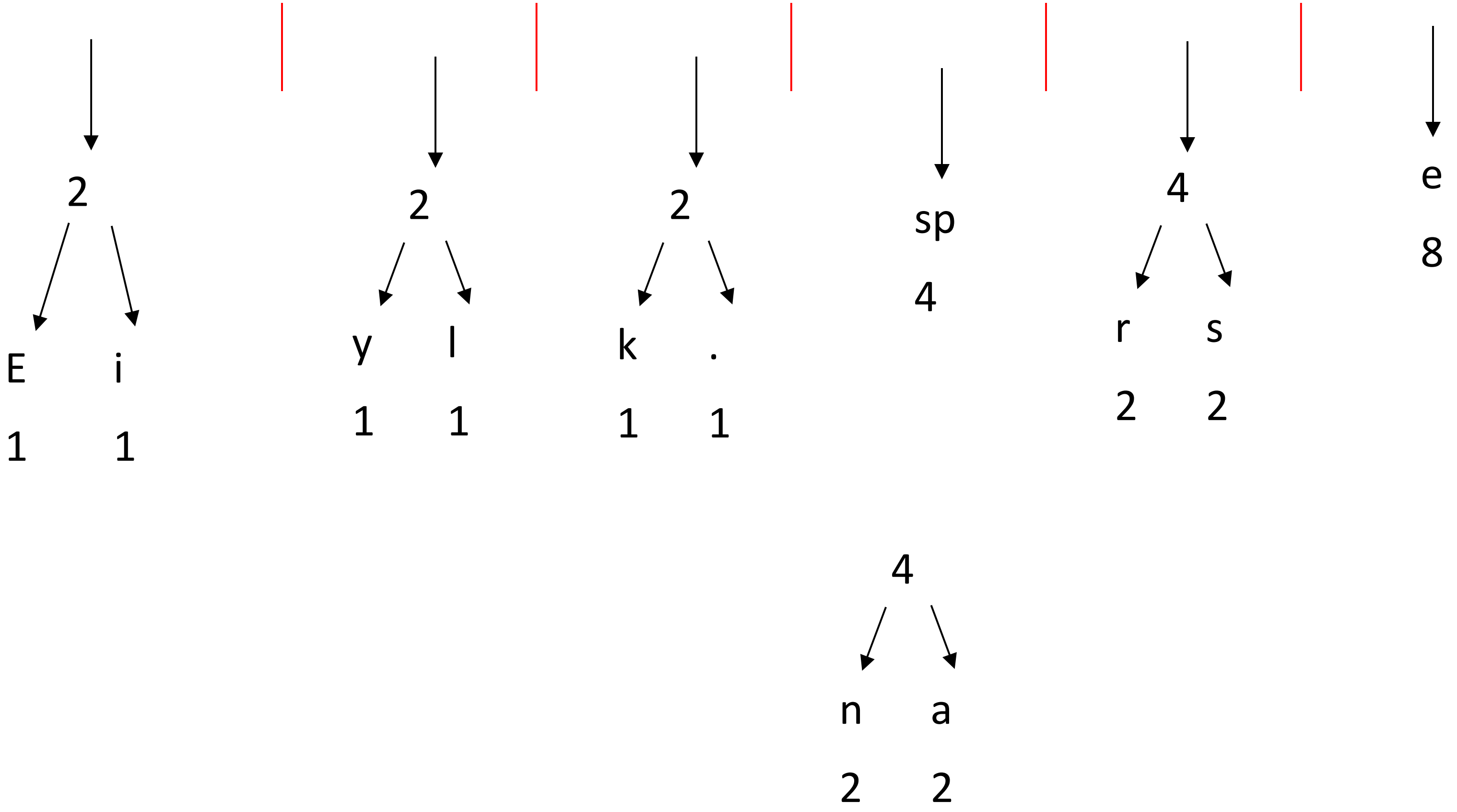
2
k 1 . 1

sp 4

4
r 2 s 2

e 8

Building a Tree



2

E 1

i 1

2

y 1

l 1

2

k 1

. 1

sp 4

4

r 2

s 2

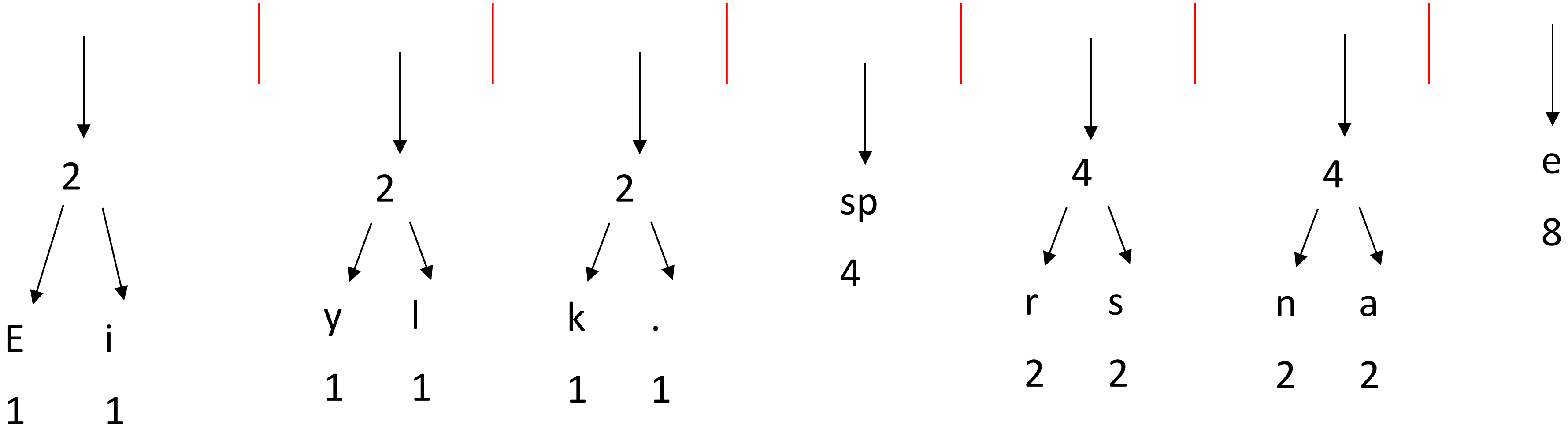
e 8

4

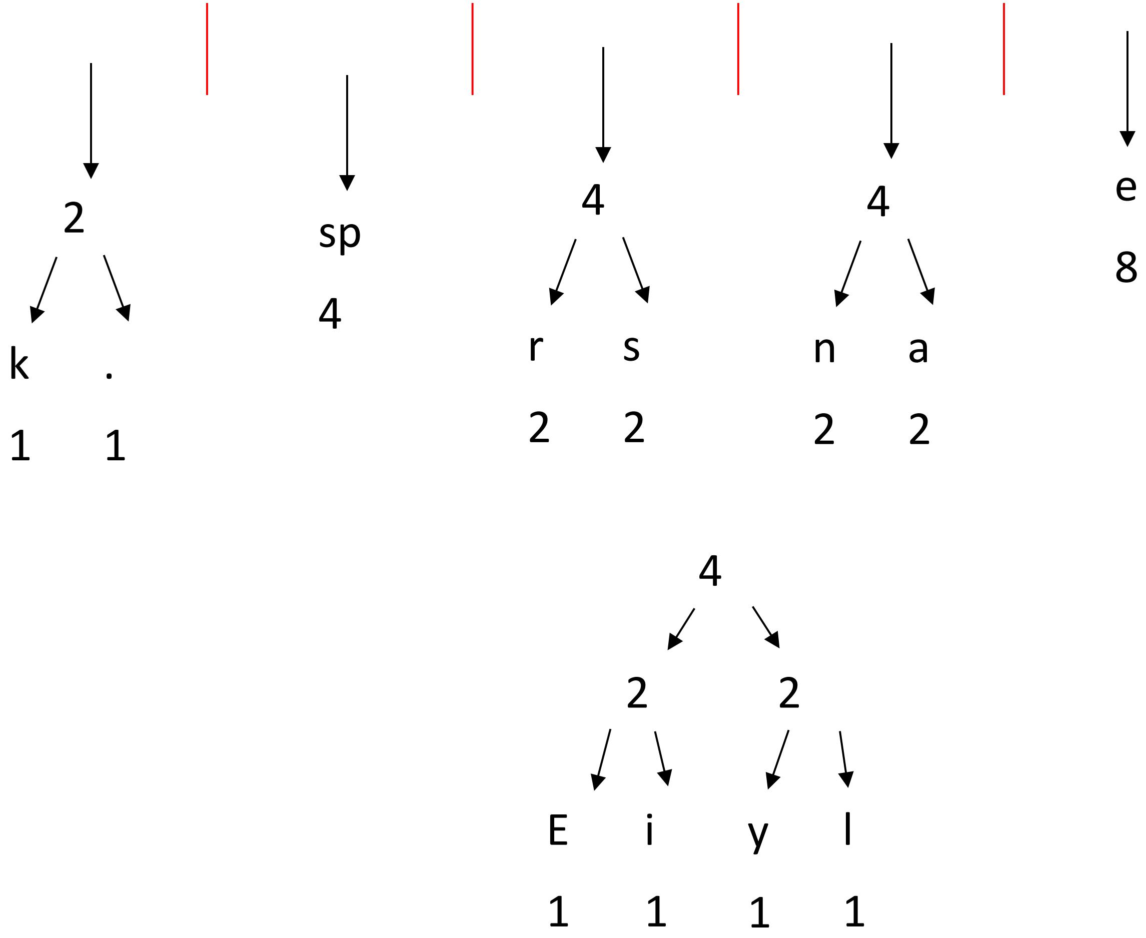
n 2

a 2

Building a Tree



Building a Tree



2

k .

1 1

sp

4

4

r s

2 2

4

n a

2 2

e

8

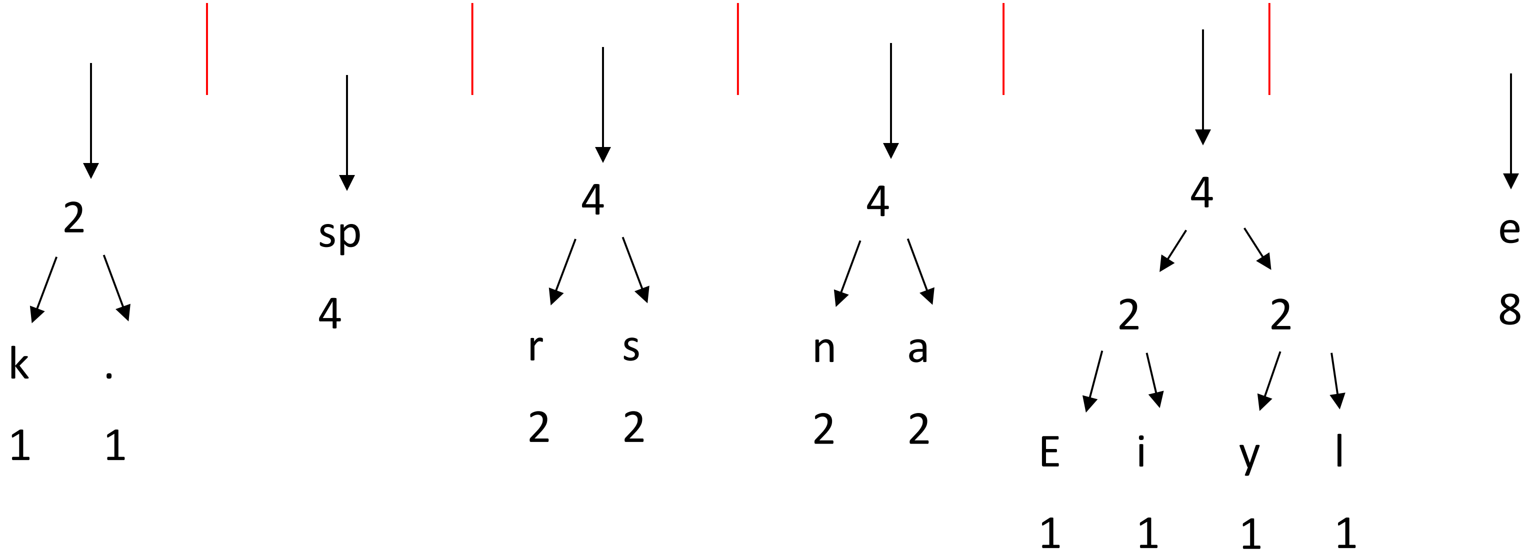
4

2 2

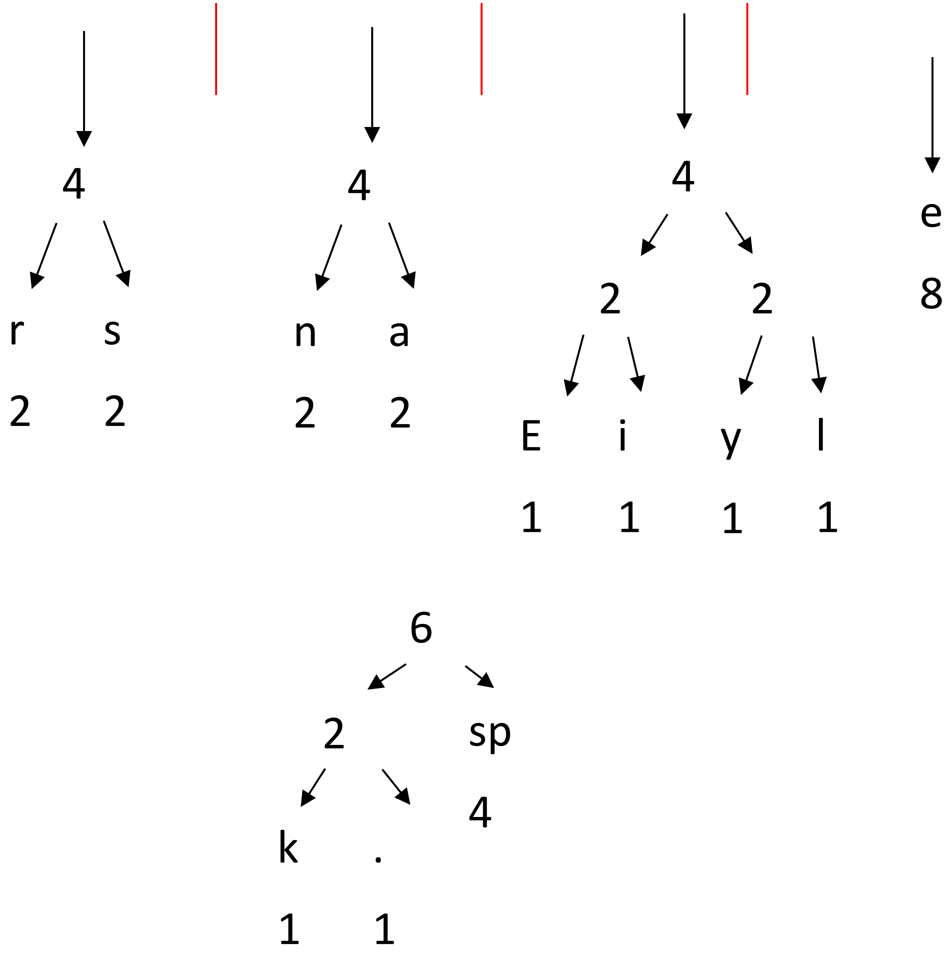
E i y l

1 1 1 1

Building a Tree



Building a Tree



4 4 4 e

r s n a 2 2 8

2 2 2 2 E i y l

1 1 1 1

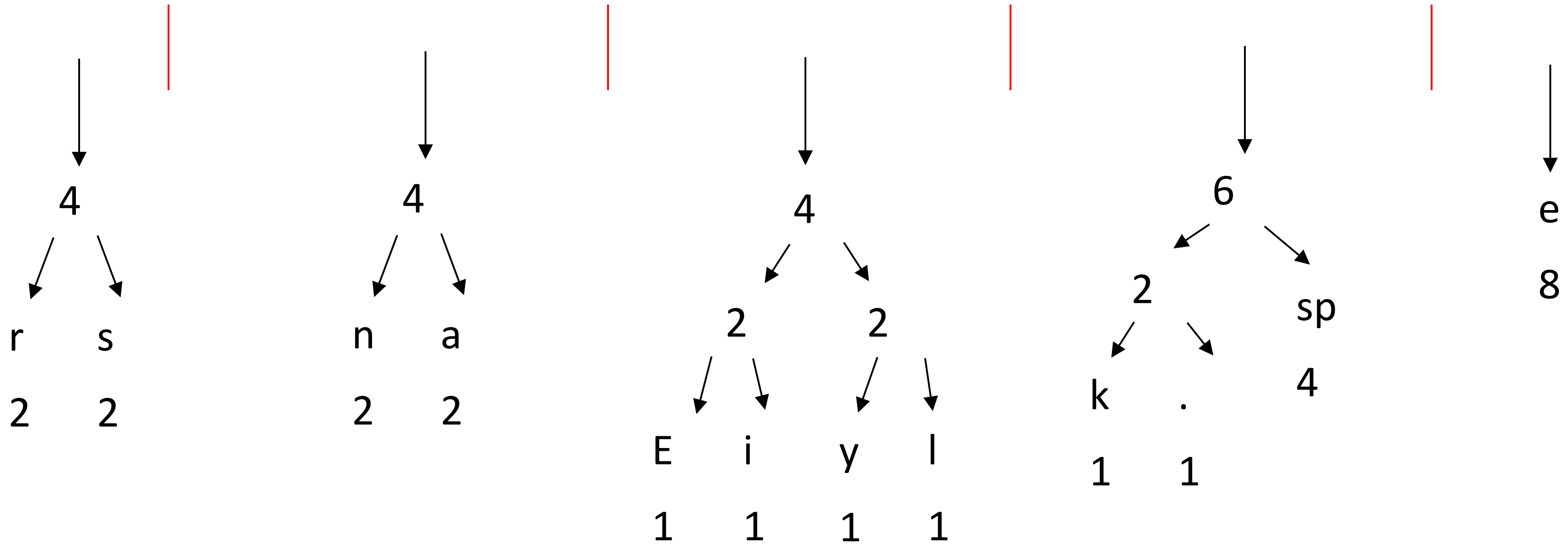
6

2 sp

k . 4

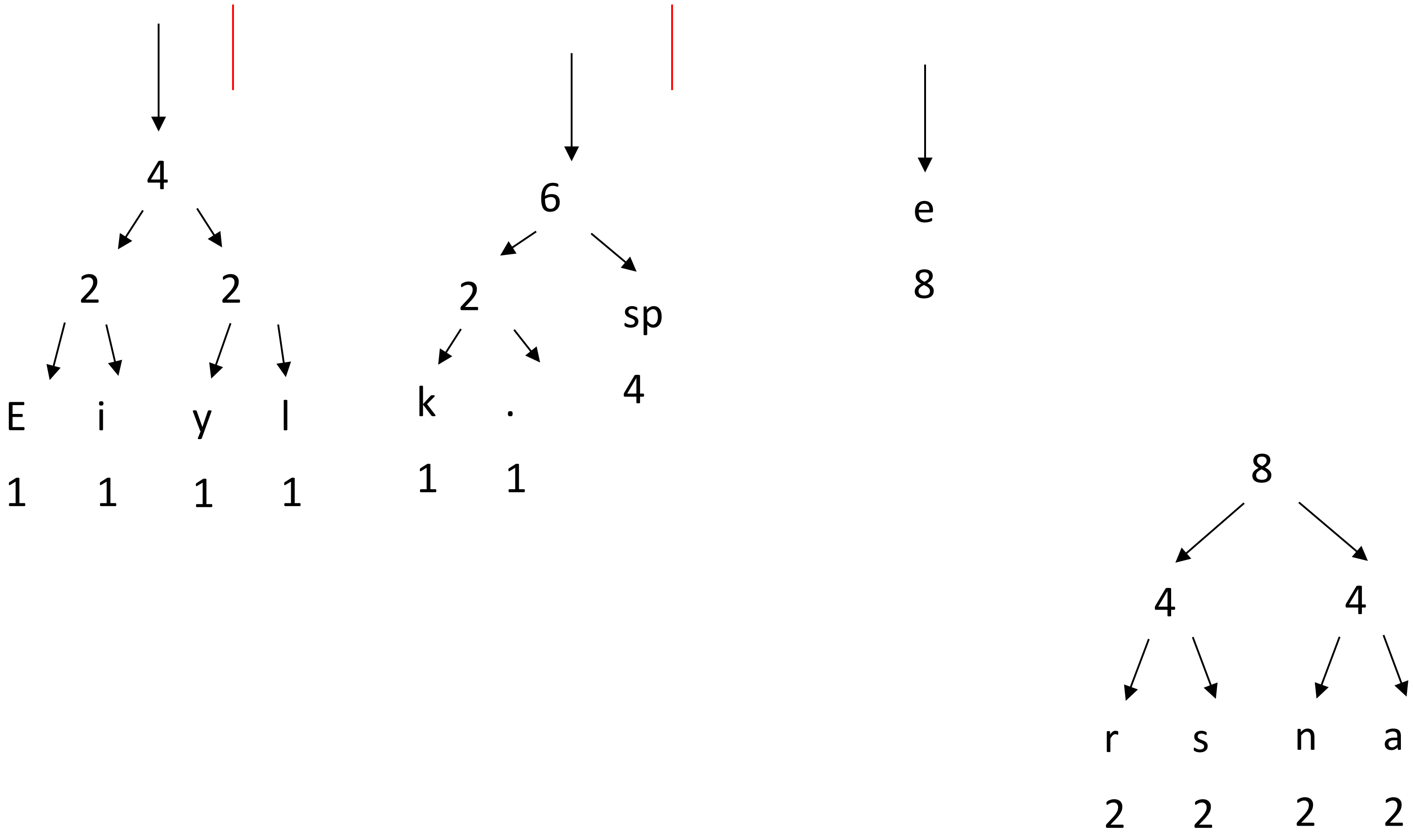
1 1

Building a Tree



What is happening to the characters with a low number of occurrences?

Building a Tree



4

2 2

E i y l

1 1 1 1

6

2 sp

k . 4

1 1

e

8

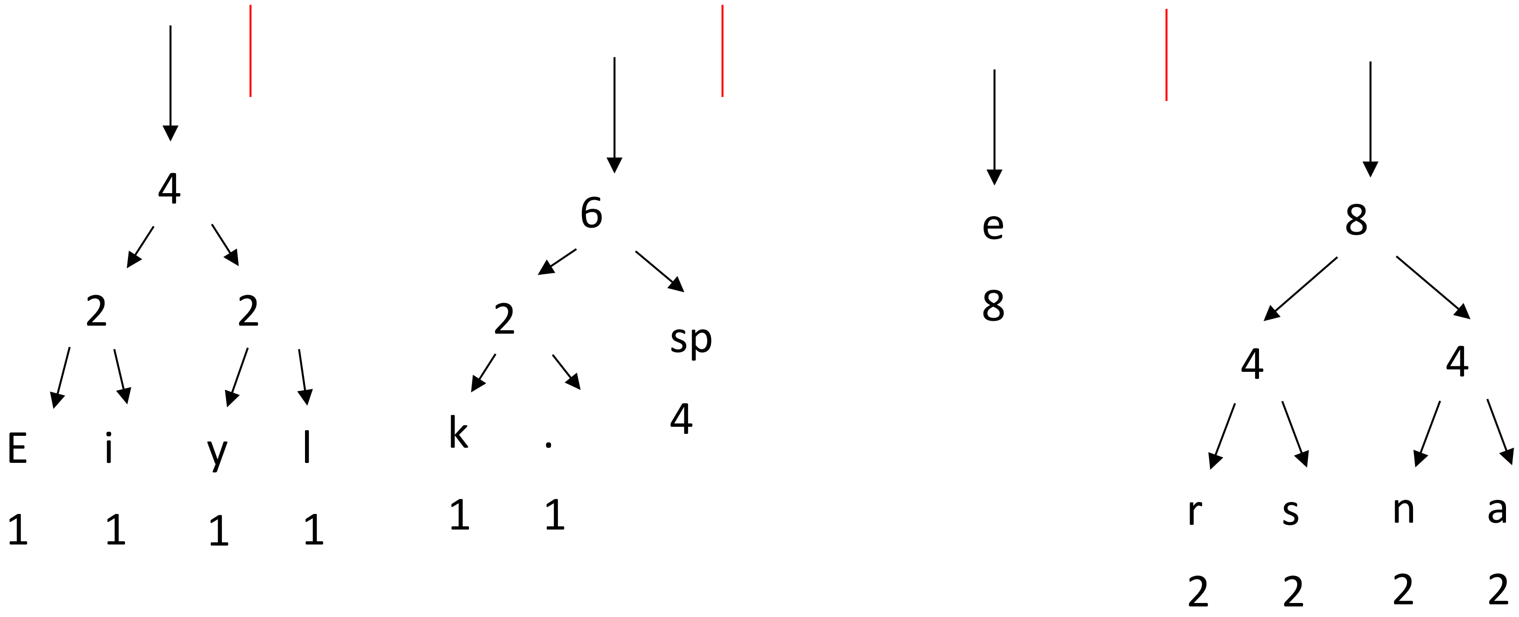
8

4 4

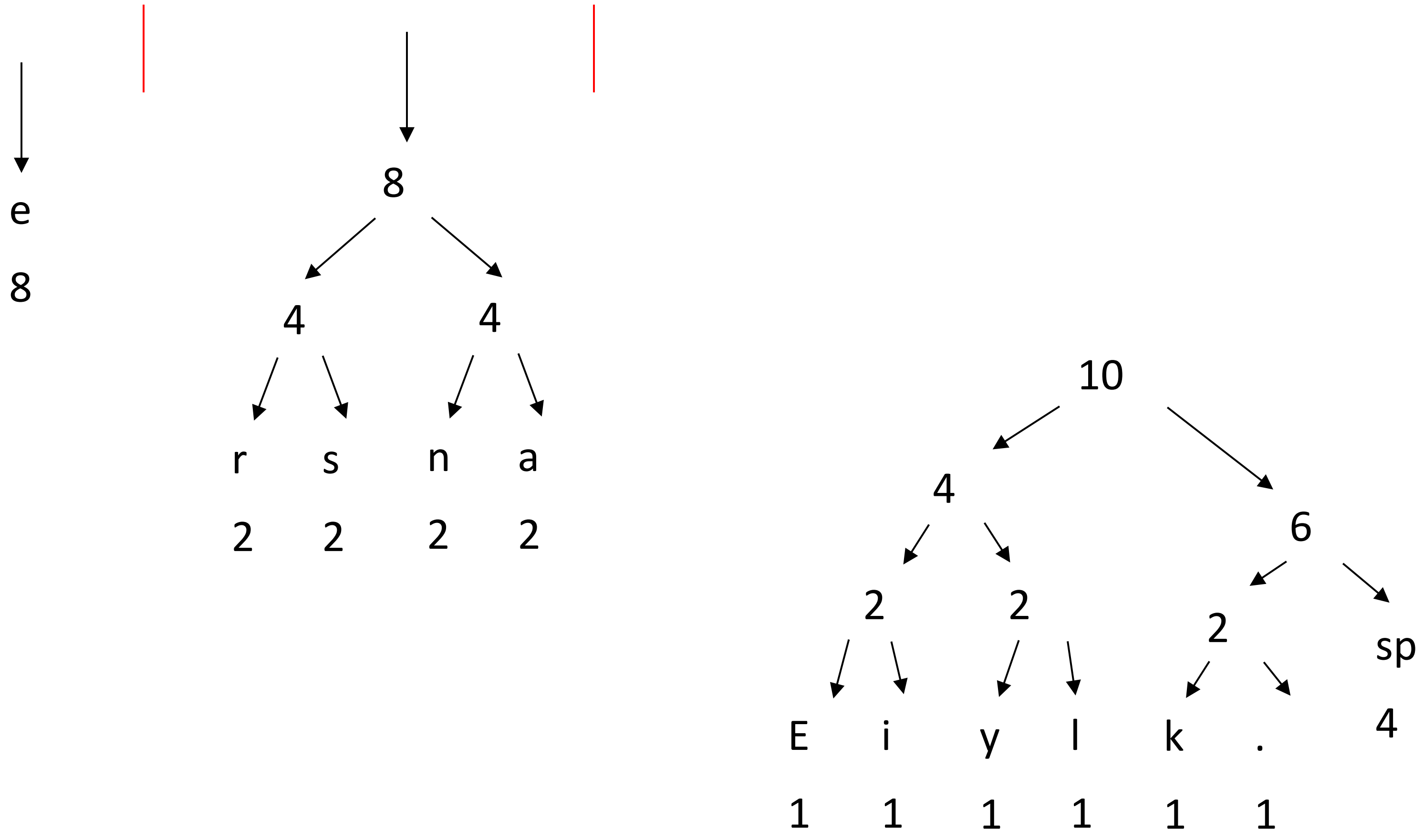
r s n a

2 2 2 2

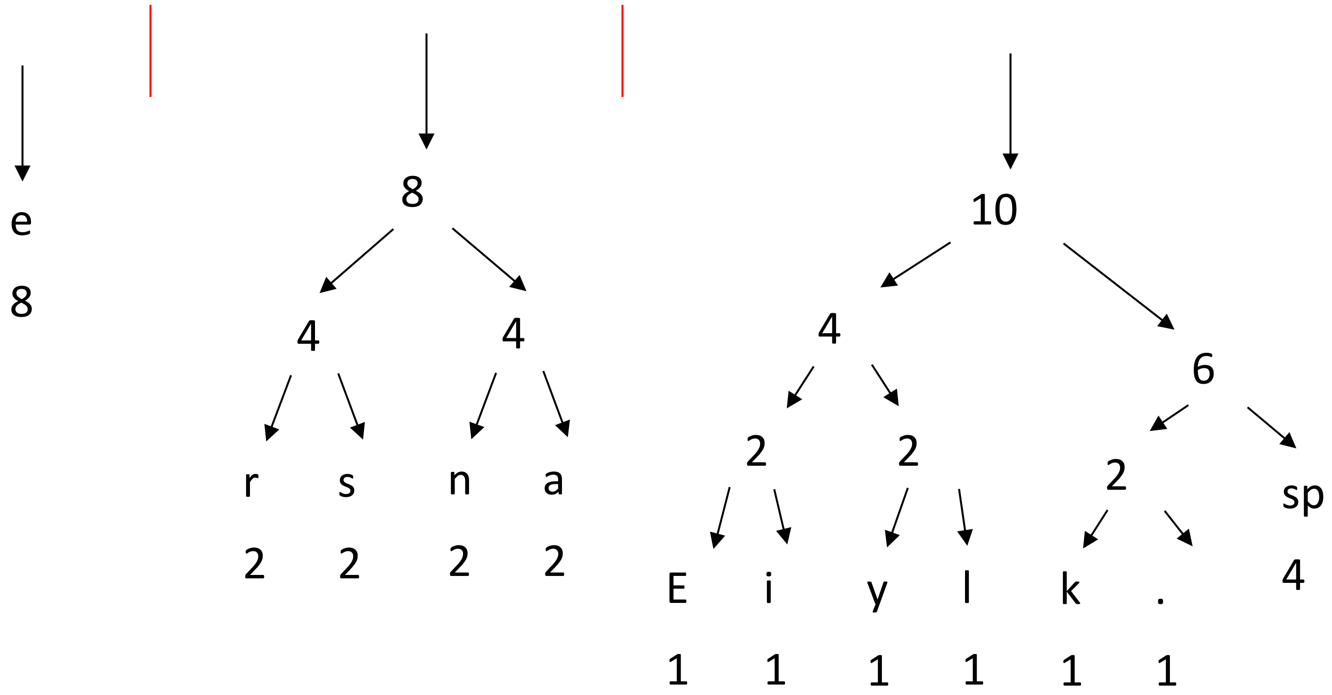
Building a Tree



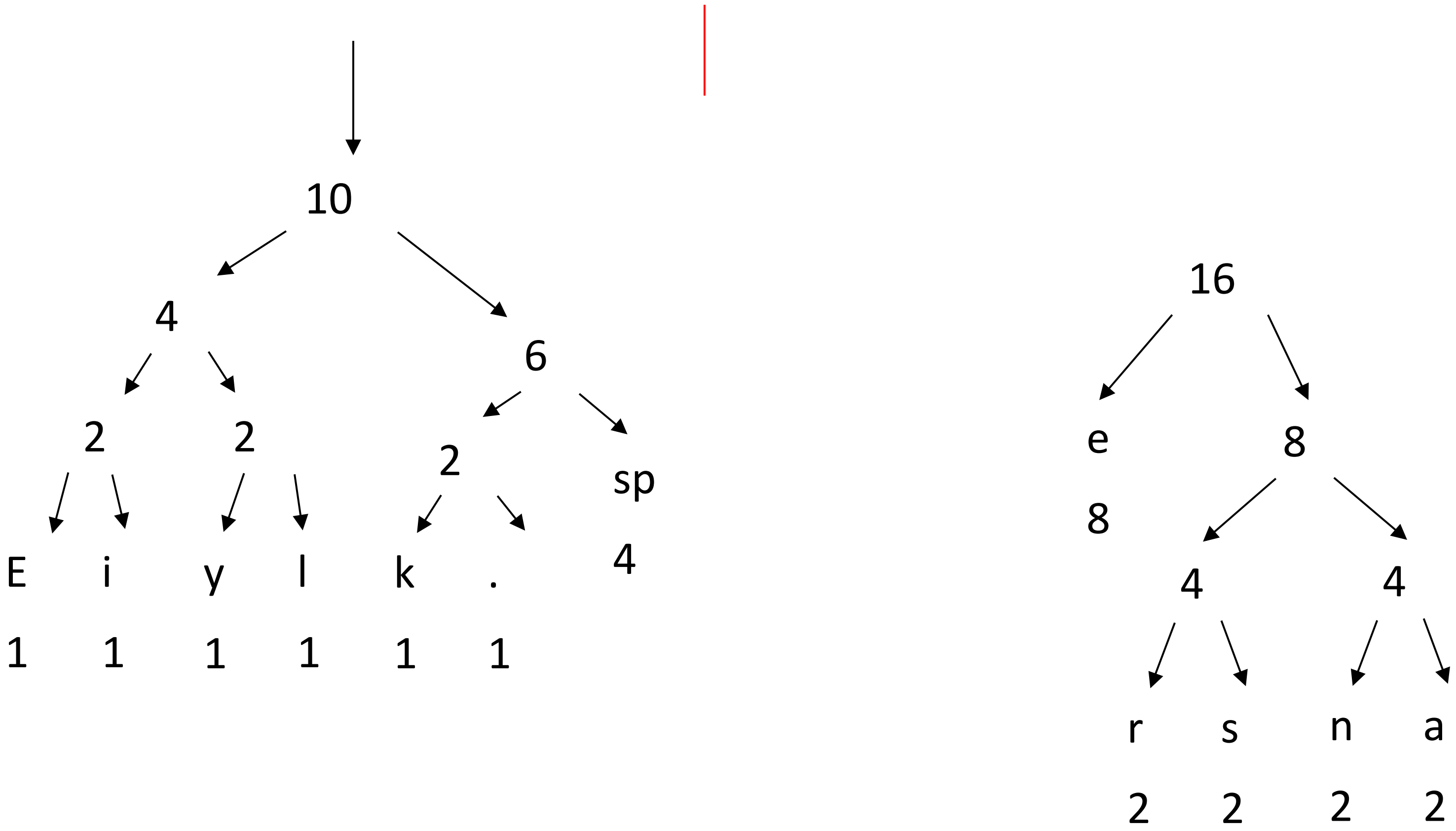
Building a Tree



Building a Tree



Building a Tree



10

4

6

2

2

2

sp

4

E

1

i

1

y

1

l

1

k

1

.

1

16

e

8

8

4

4

r

2

s

2

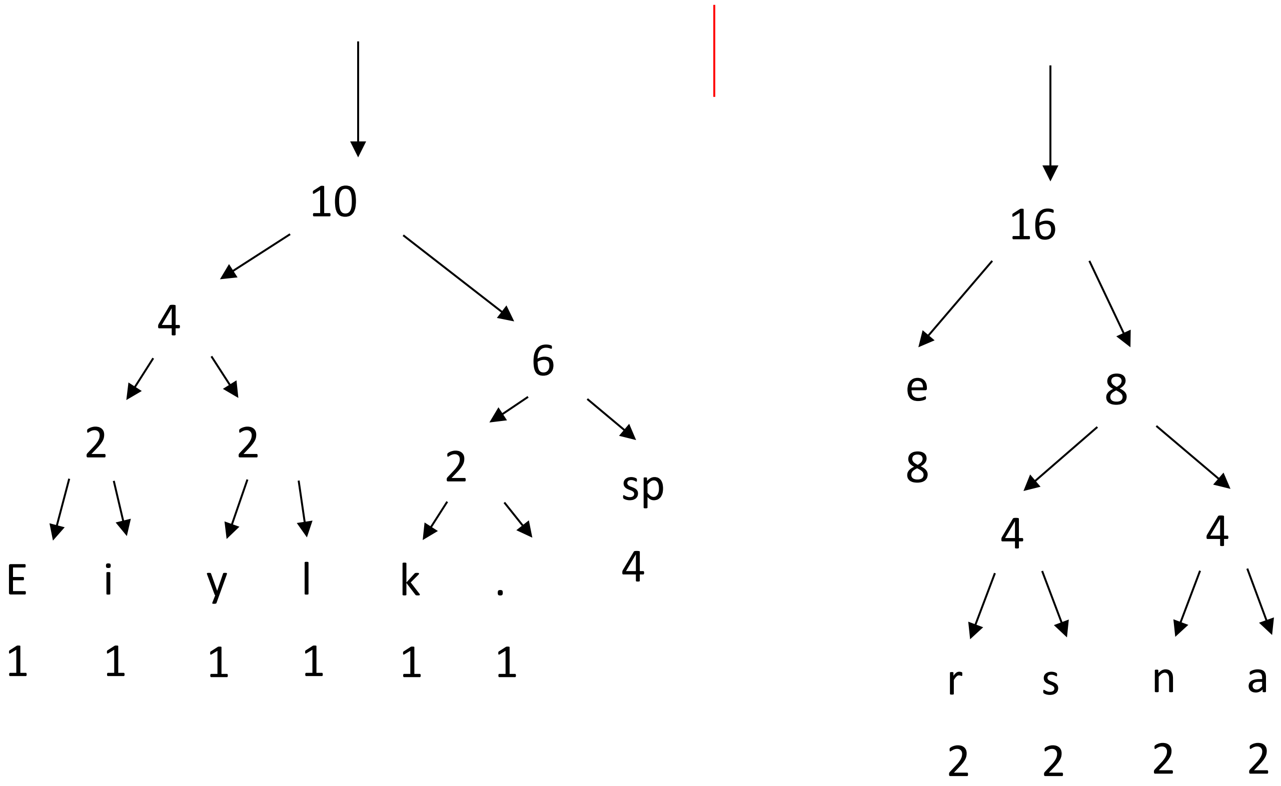
n

2

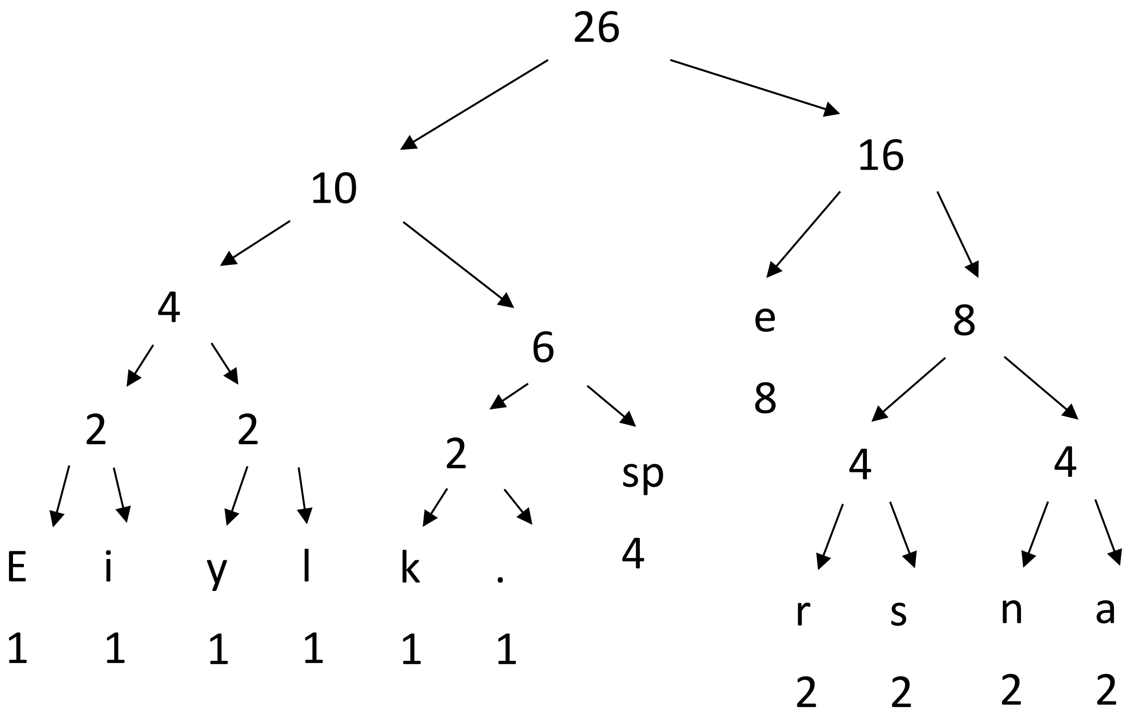
a

2

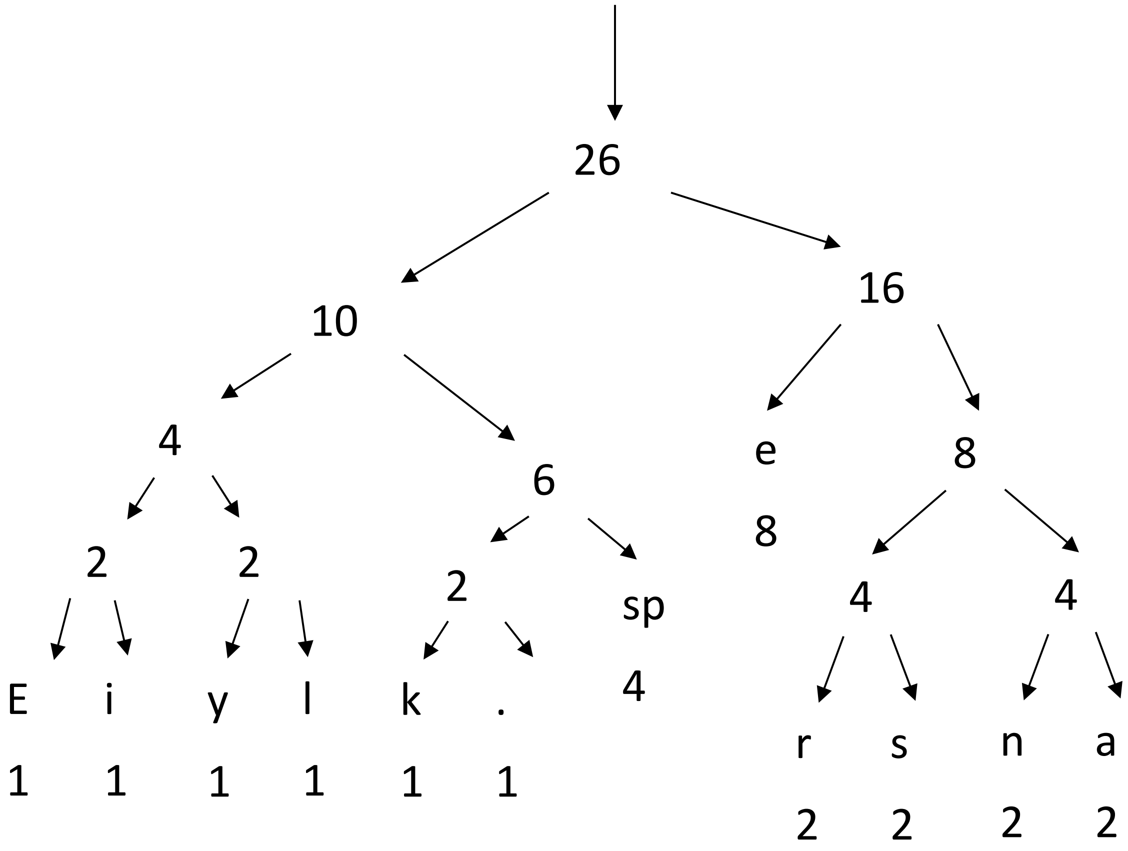
Building a Tree



Building a Tree



Building a Tree



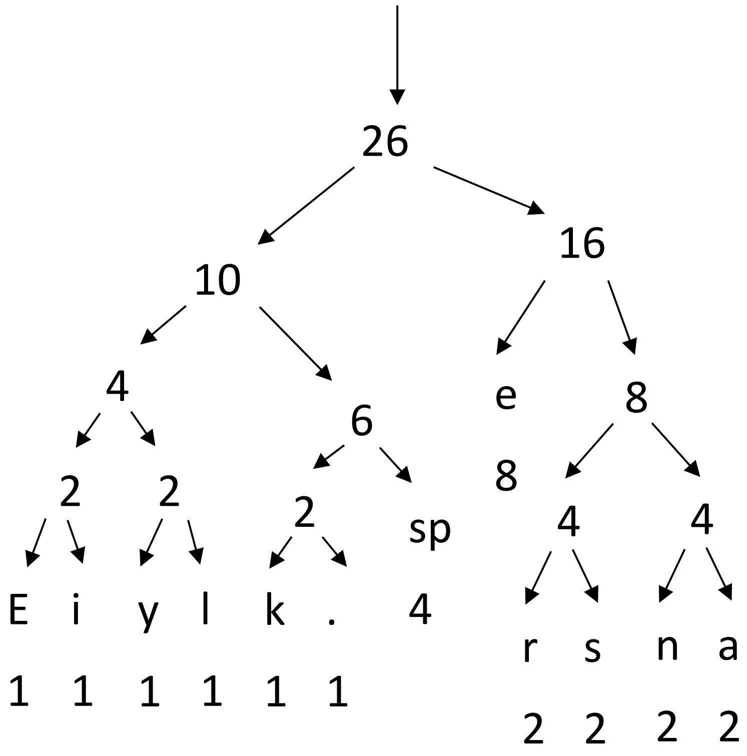
After enqueueing this node there is only one node left in priority queue.

Building a Tree

Dequeue the single node left in the queue.

This tree contains the new code words for each character.

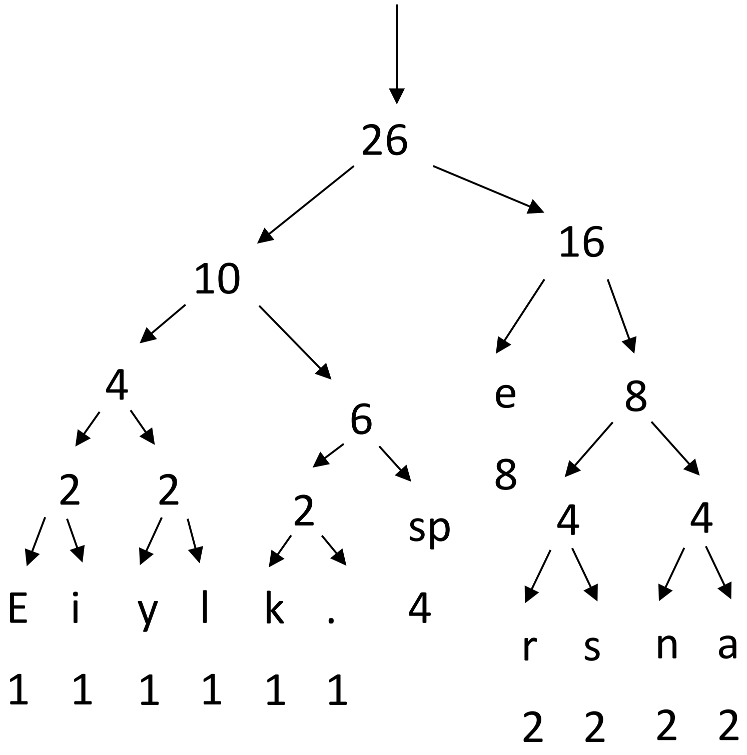
Frequency of root node should equal number of characters in text.



Eerie eyes seen near lake.   ✔ 26 characters

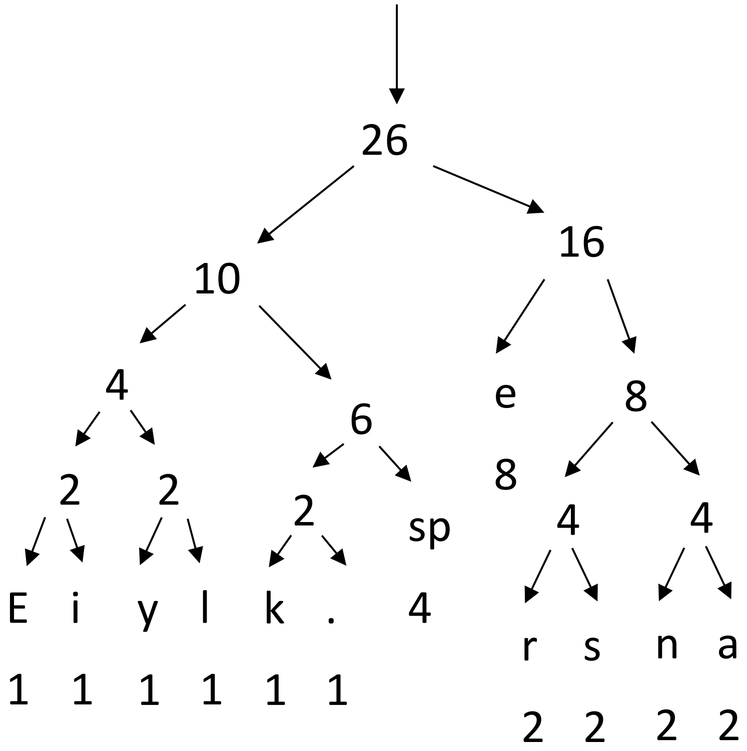
Encoding the File Traverse Tree for Codes

- 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



Encoding the File 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.

000010110000011001110001010110110100
111110101111110001100111111010010010
1

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

- Why is there no need for a separator character?

.

Encoding the File

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

- If modified code used 4 bits per character are needed. Total bits 4 * 26 = 104. Savings not as great.

```
000010110000011001110001010110110100
111110101111110001100111111010010010
1
```

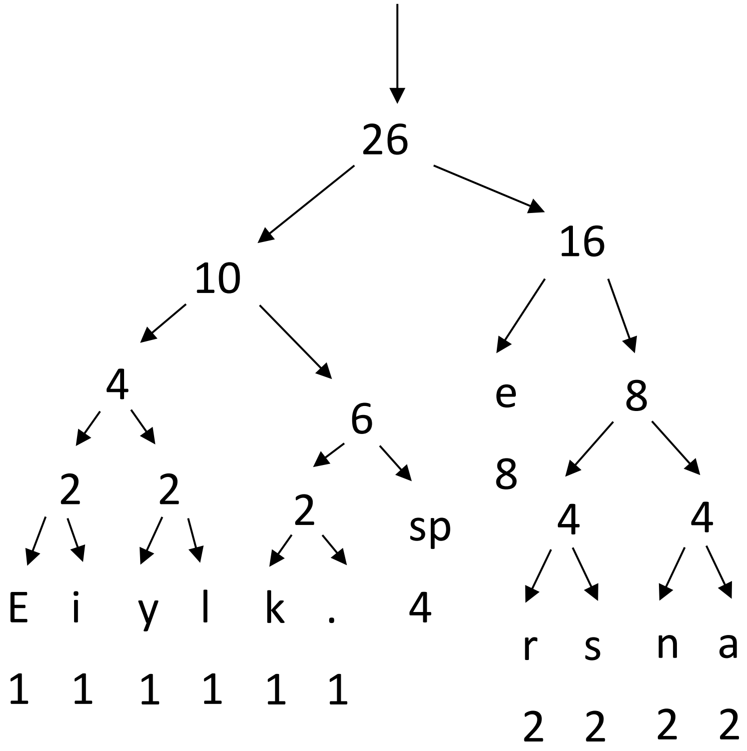
Decoding the File

- How does receiver know what the codes are?
- Tree constructed for each text file.
  - Considers frequency for each file
  - Big hit on compression, especially for smaller files
- Tree predetermined
  - based on statistical analysis of text files or file types
- Data transmission is bit based versus byte based

Decoding the File

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

101000110111101111011
11110000110101



Summary

- Huffman coding is a technique used to compress files for transmission
- Uses statistical coding
  - more frequently used symbols have shorter code words
- Works well for text and fax transmissions
- An application that uses several data structures

Thank You !!!