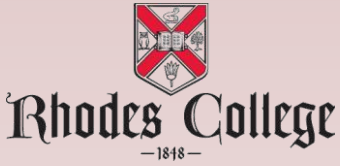
COMP 355
Advanced Algorithms

Minimizing Lateness &
Huffman Encoding



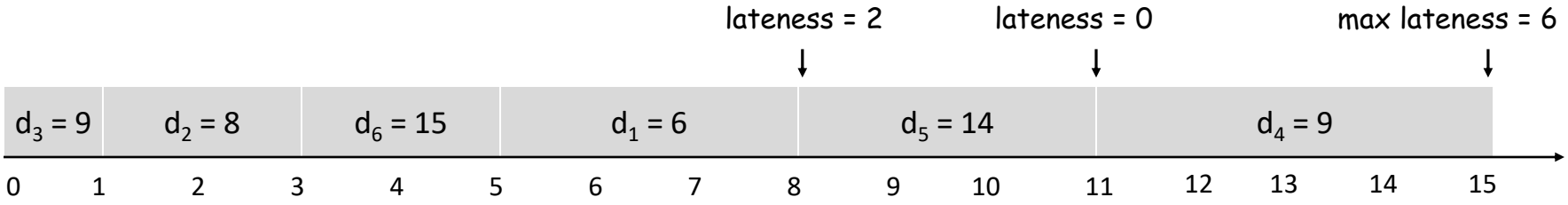
Scheduling to Minimizing Lateness

Minimizing lateness problem.

- Single resource processes one job at a time.
- Job j requires t_j units of processing time and is due at time d_j.
- If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

- [Earliest deadline first] Consider jobs in ascending order of deadline d_j.

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

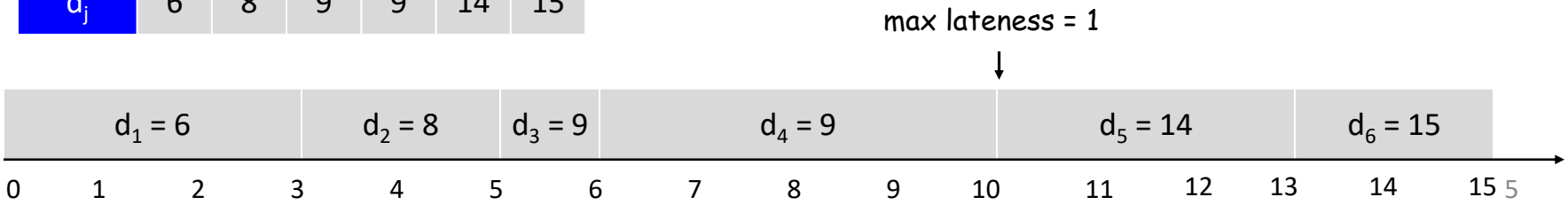
counterexample

Minimizing Lateness: Greedy Algorithms

Greedy algorithm. Earliest deadline first.

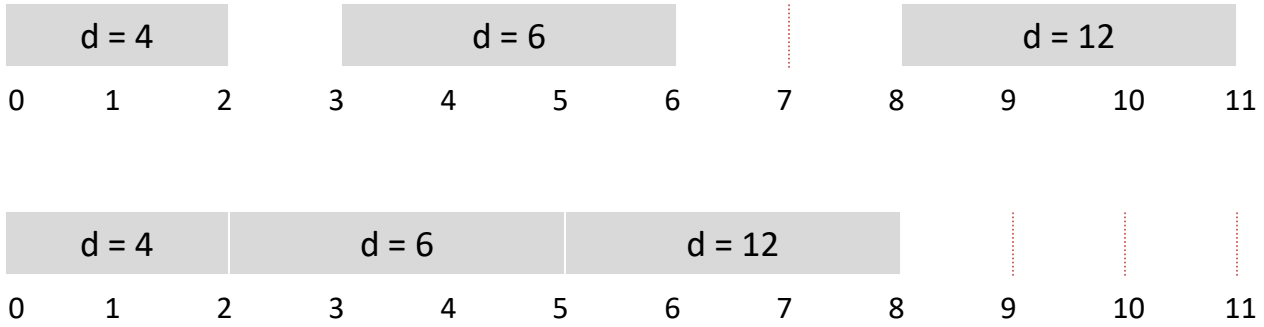
```
Sort n jobs by deadline so that d_1 ≤ d_2 ≤ … ≤ d_n

t ← 0
for j = 1 to n
    Assign job j to interval [t, t + t_j]
    s_j ← t, f_j ← t + t_j
    t ← t + t_j
output intervals [s_j, f_j]
```

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: No Idle Time

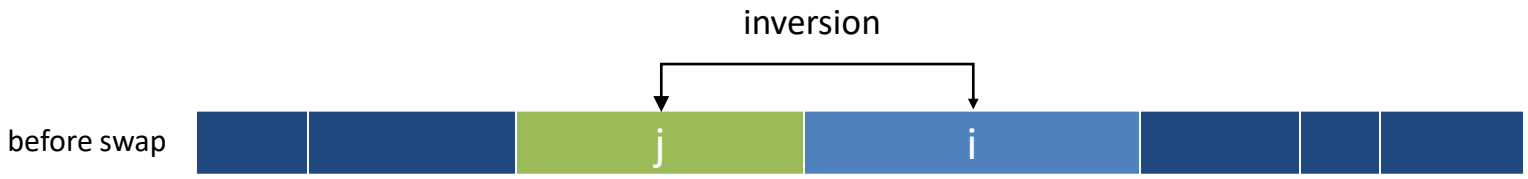
Observation. There exists an optimal schedule with no idle time.



Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.

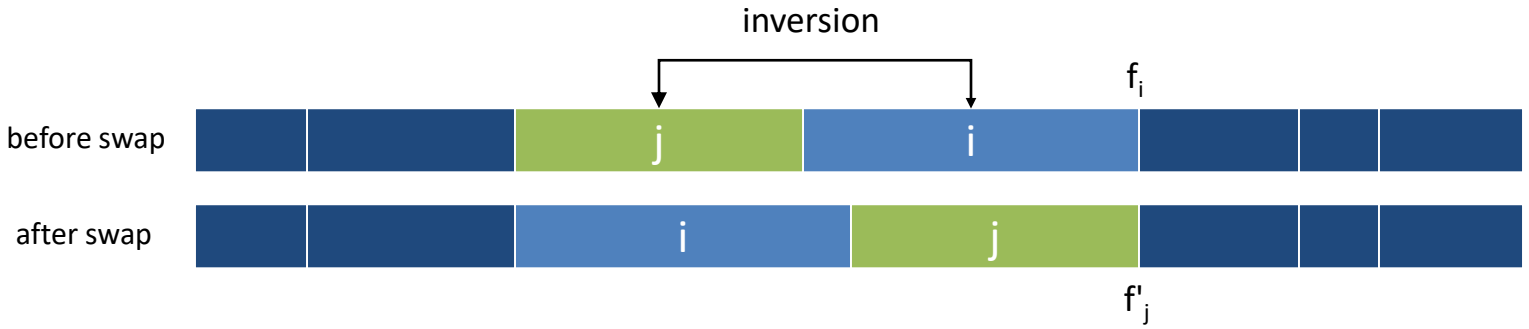


Observation. Greedy schedule has no inversions.

Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Minimizing Lateness: Inversions

Def. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.



Claim. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

Pf. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job j is late:

$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(definition)} \\ &= f_i - d_j && (j \text{ finishes at time } f_i) \\ &\leq f_i - d_i && (i < j) \\ &\leq \ell_i && \text{(definition)} \end{aligned}$$

Minimizing Lateness: Analysis of Greedy Algorithm

Theorem. Greedy schedule G is optimal.

Claim: There is an optimal schedule O with no idle time.

- If O has no inversions, then G = O.
- If O has an inversion, let i-j be an adjacent inversion.
  - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
  - we can continue to do this until there are no more inversions in O. When that is the case, G = O.

Greedy Analysis Strategies

Greedy algorithm stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

Exchange argument. Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

Structural. Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

Data Storage

Normal encoding:

- ASCII or Unicode, each character represented by a fixed-length codeword of bits (8 or 16 bits/character)
- Easy to decode
- Not the most efficient way to store data

Fixed Length Encoding

Suppose we have a 4-character alphabet {a, b, c, d}

Character	a	b	c	d
Fixed-Length Codeword	00	01	10	11

Given the string “abacdaacac”, it would be encoded as

a	b	a	c	d	a	a	c	a	c
00	01	00	10	11	00	00	10	00	10

The final 20-character binary string would be “00010010110000100010”.

But what if we knew the frequency of the characters in advance?

Variable Length Encoding

Using variable length codes

Character	a	b	c	d
Probability	0.60	0.05	0.30	0.05
Variable-Length Codeword	0	110	10	111

Given the string “abacdaacac”, it would be encoded as

a	b	a	c	d	a	a	c	a	c
0	110	0	10	111	0	0	10	0	10

The resulting 17-character string would be “01100101110010010”. (savings of 3 bits)

Resulting string is 1.5n compared to 2n, for a savings of 25% in expected encoding length.

$n(0.60 \cdot 1 + 0.05 \cdot 3 + 0.30 \cdot 2 + 0.05 \cdot 3) = n(0.60 + 0.15 + 0.60 + 0.15) = 1.5n.$

Prefix Codes

How to decode variable-length codes?

Character	a	b	c	d
Probability	0.60	0.05	0.30	0.05
Variable-Length Codeword	0	110	10	111

In the variable-length codes given in the example above no codeword is a prefix of another (very important!)

Observe: If two codewords did share a common prefix, e.g. a → 001 and b → 00101, then when we see 00101, how do we know whether the first character of the encoded message is “a” or “b”?

Conversely: If no codeword is a prefix of any other, then as soon as we see a codeword appearing as a prefix in the encoded text, then we know that we may decode it

Prefix Codes

Mapping of codewords to characters so that no codeword is a prefix of another.

Character	a	b	c	d
Codeword	0	110	10	111

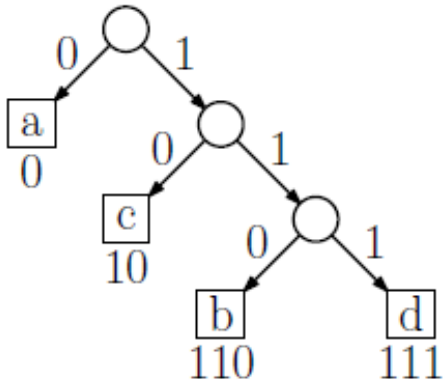


Fig. 25: A tree-representation of a prefix code.

Expected Encoding Length

$$B(T) = n \sum_{x \in C} p(x) d_T(x).$$

Optimal Code Generation: Given an alphabet C and the probabilities p(x) of occurrence for each character x ∈ C, compute a prefix code T that ***minimizes*** the expected length of the encoded bit-string, B(T).

n = # of characters in the encoded string

Huffman's Algorithm

- We are given the occurrence probabilities for the characters.
- Build the tree up from the leaf level.
- Take two characters x and y, and "merge" them into a single super-character called z (prob(z) = prob(x) + prob(y)), which then replaces x and y in the alphabet.
- Continue recursively building the code on the new alphabet, which has one fewer character.
- When done, if codeword for z is 010, then x is 0100 and y is 0101.

Huffman's Algorithm

Huffman's Algorithm

```
huffman(char C[], float prob[]) {
    for each (x in C) {
        add x to Q sorted by prob[x]      // add all to priority queue
    }
    n = size of C
    for (i = 1 to n-1) {                  // repeat until 1 item in queue
        z = new internal tree node
        left[z]  = x = extract-min from Q // extract min probabilities
        right[z] = y = extract-min from Q
        prob[z]  = prob[x] + prob[y]      // z's probability is their sum
        insert z into Q                   // z replaces x and y
    }
    return the last element left in Q as the root
}
```

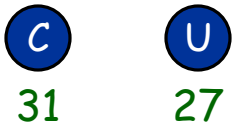
Huffman Code Construction

Character count in text.

Char	Freq
E	125
T	93
A	80
O	76
I	73
N	71
S	65
R	61
H	55
L	41
D	40
C	31
U	27

Huffman Code Construction

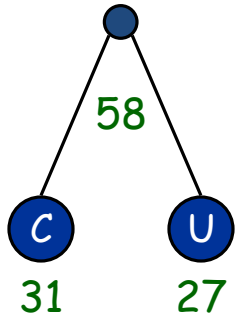
Char	Freq
E	125
T	93
A	80
O	76
I	73
N	71
S	65
R	61
H	55
L	41
D	40
C	31
U	27



Huffman Code Construction

Char	Freq
E	125
T	93
A	80
O	76
I	73
N	71
S	65
R	61
	58
H	55
L	41
D	40

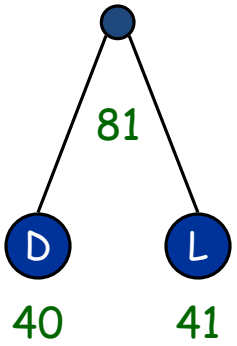
C	31
U	27

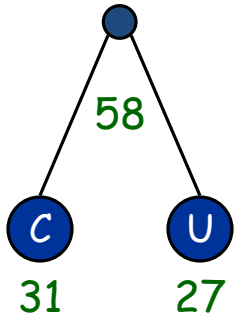


Huffman Code Construction

Char	Freq
E	125
T	93
	81
A	80
O	76
I	73
N	71
S	65
R	61
	58
H	55

L	41
D	40

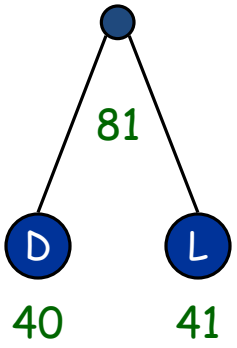


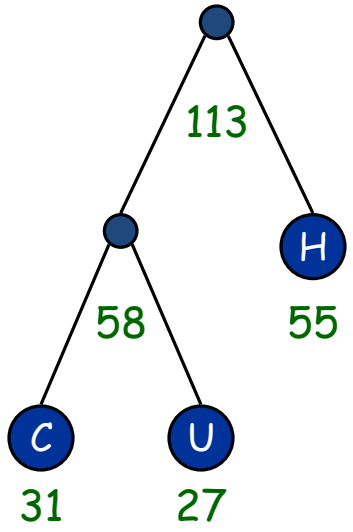


Huffman Code Construction

Char	Freq
E	125
	113
T	93
	81
A	80
O	76
I	73
N	71
S	65
R	61

	58
H	55

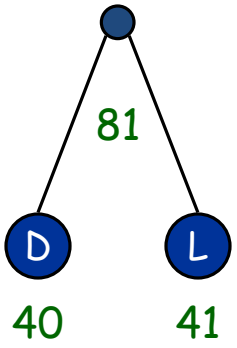


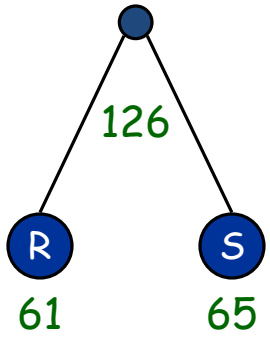


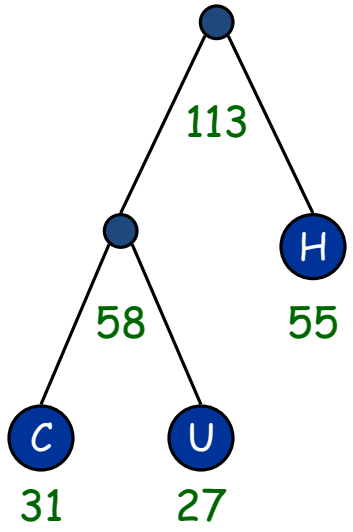
Huffman Code Construction

Char	Freq
	126
E	125
	113
T	93
	81
A	80
O	76
I	73
N	71

S	65
R	61



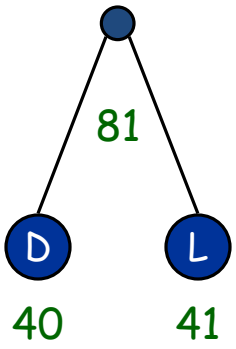


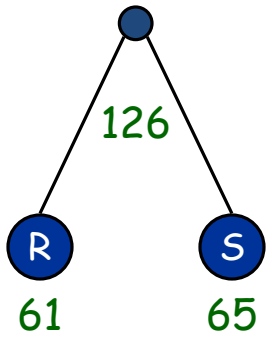


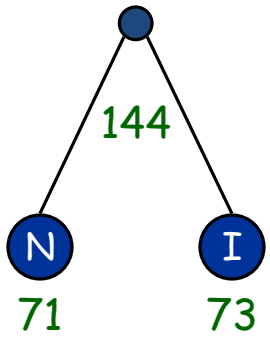
Huffman Code Construction

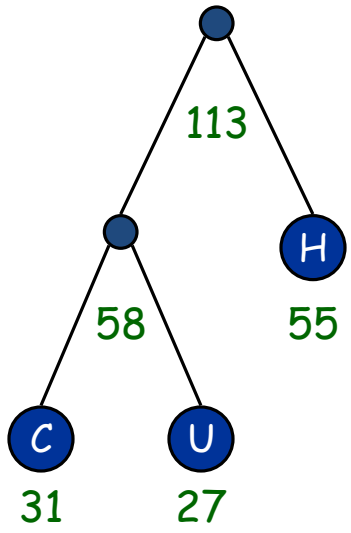
Char	Freq
	144
	126
E	125
	113
T	93
	81
A	80
O	76

I	73
N	71





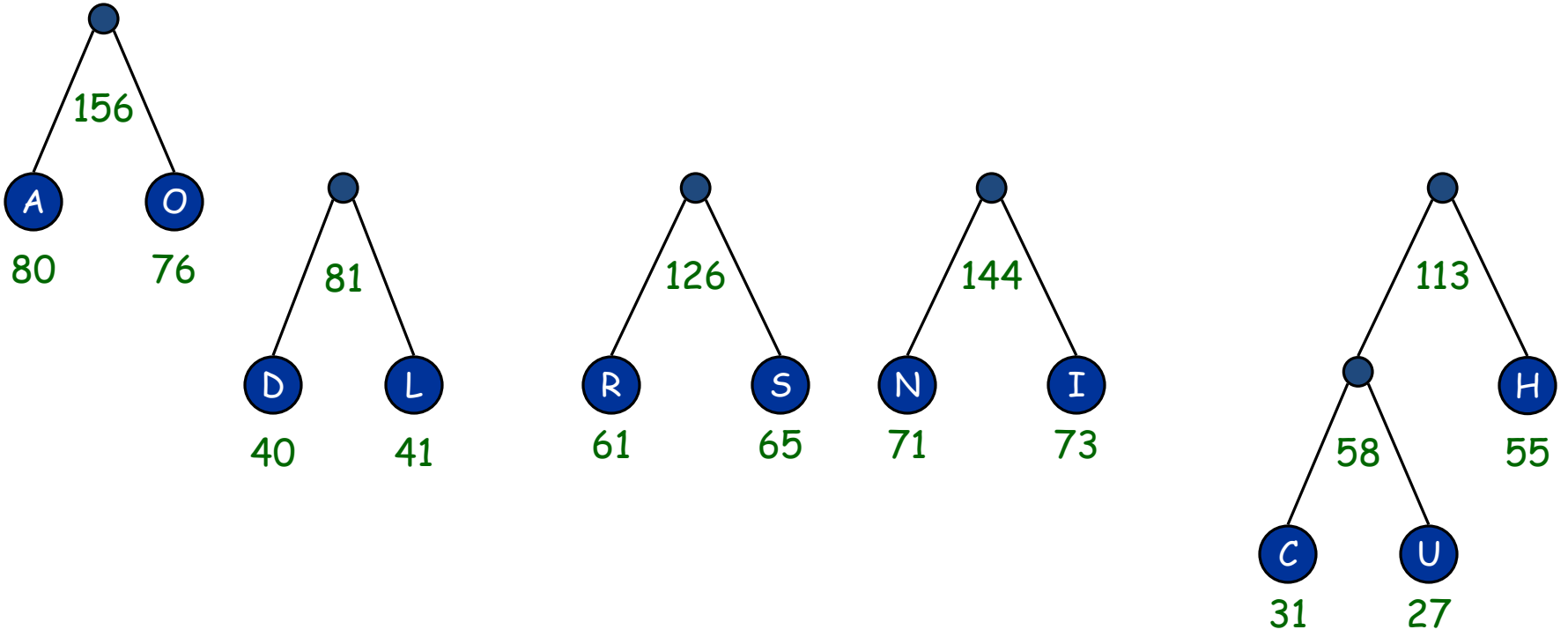




Huffman Code Construction

Char	Freq
	156
	144
	126
E	125
	113
T	93
	81

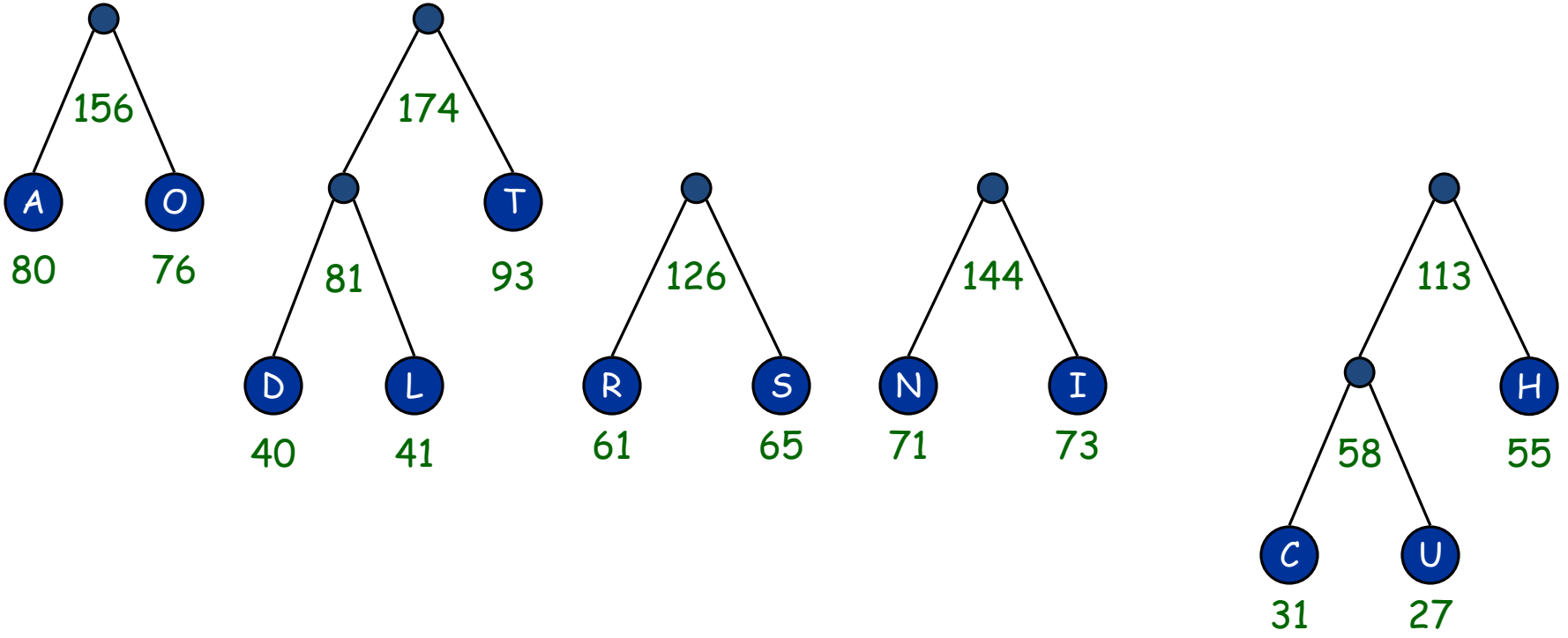
A	80
O	76



Huffman Code Construction

Char	Freq
	174
	156
	144
	126
E	125
	113

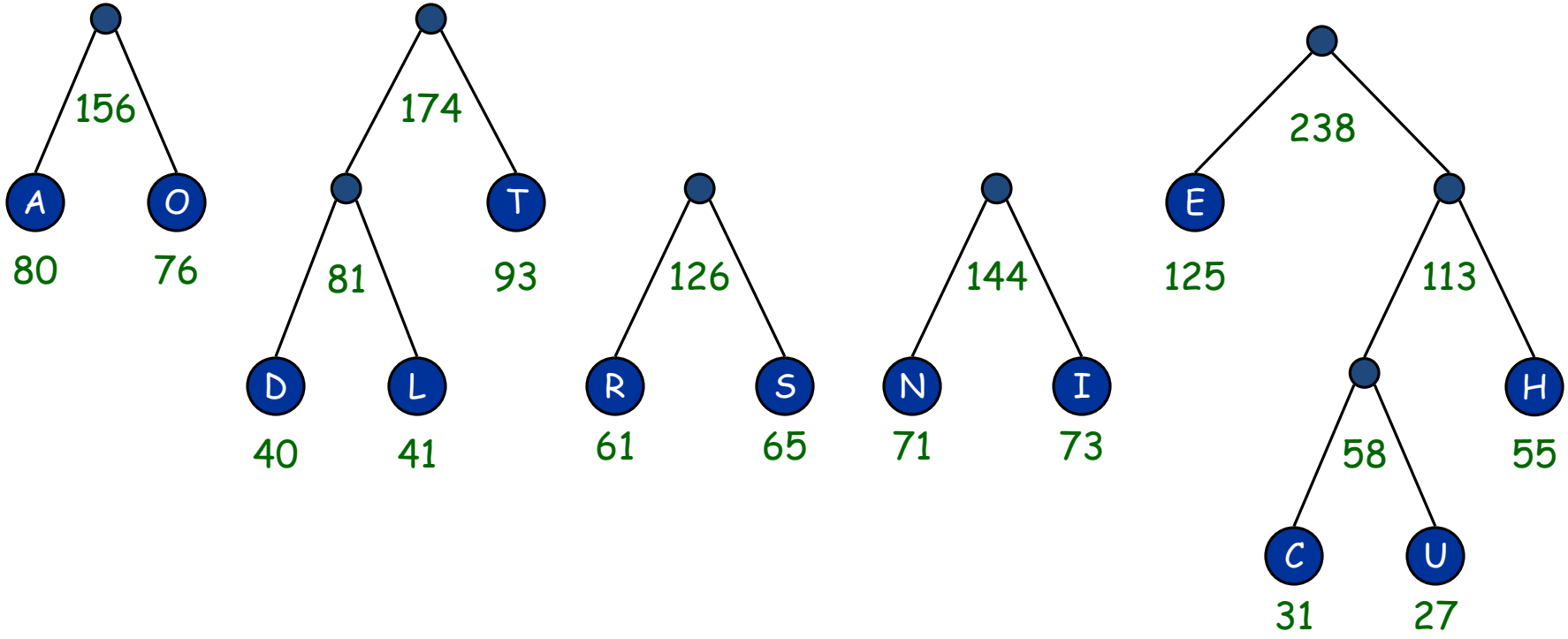
T	93
	81



Huffman Code Construction

Char	Freq
	238
	174
	156
	144
	126

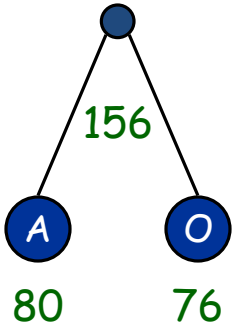
E	125
	113

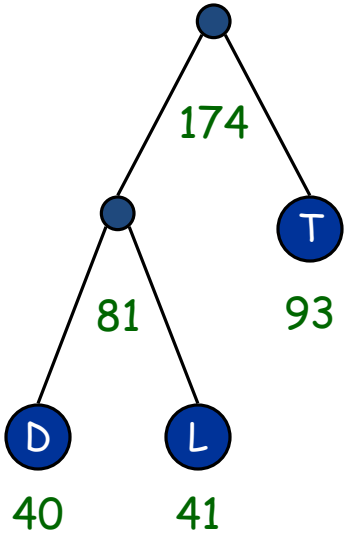


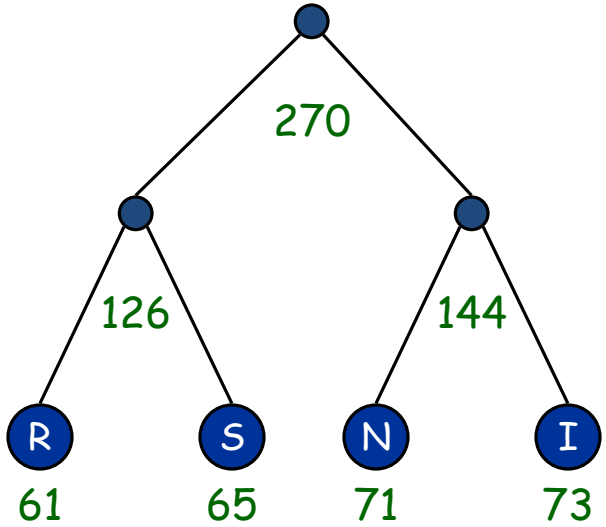
Huffman Code Construction

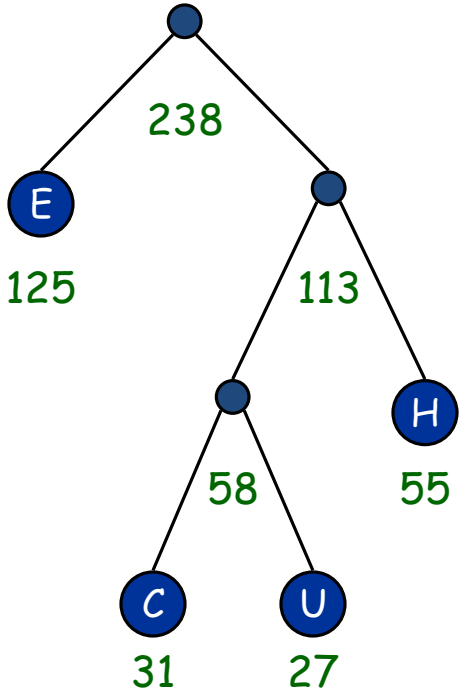
Char	Freq
	270
	238
	174
	156

	144
	126





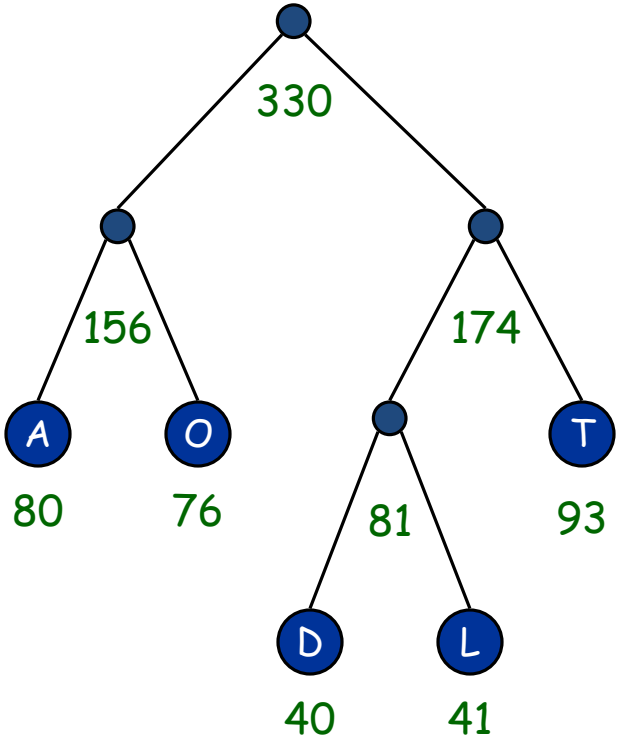


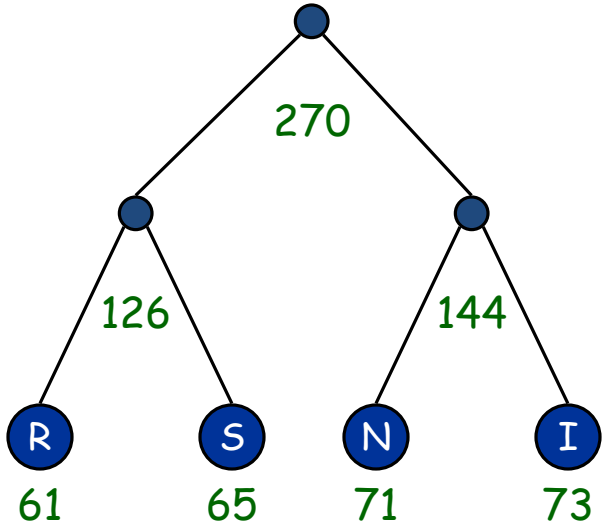


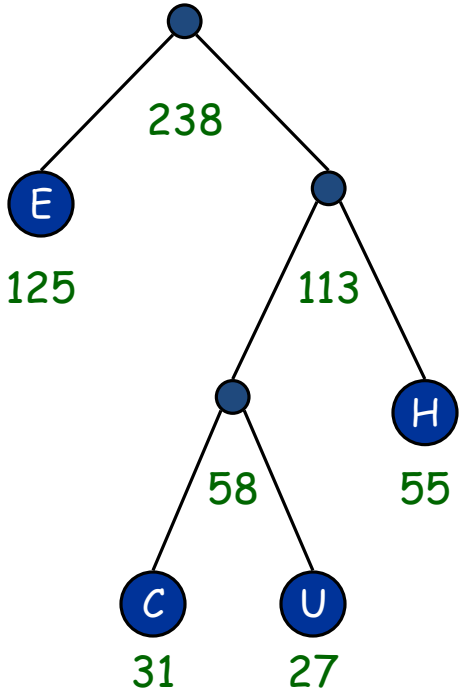
Huffman Code Construction

Char	Freq
	330
	270
	238

	174
	156



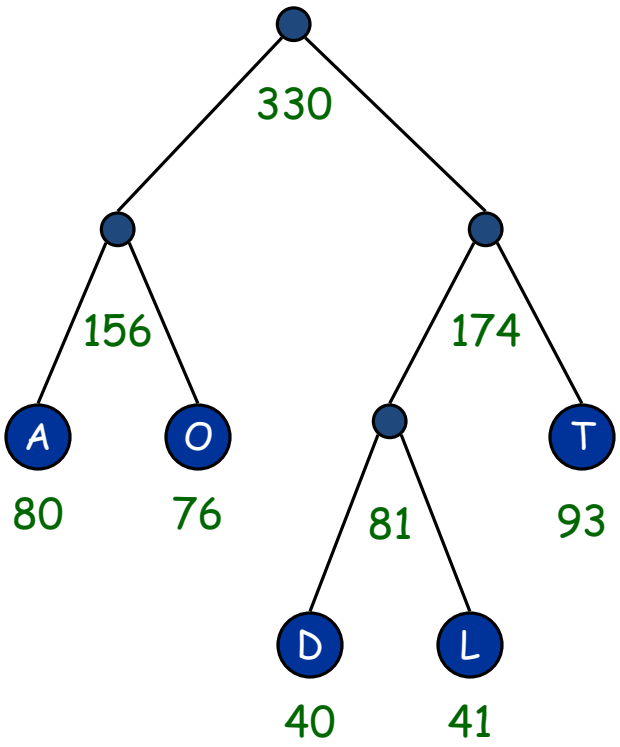


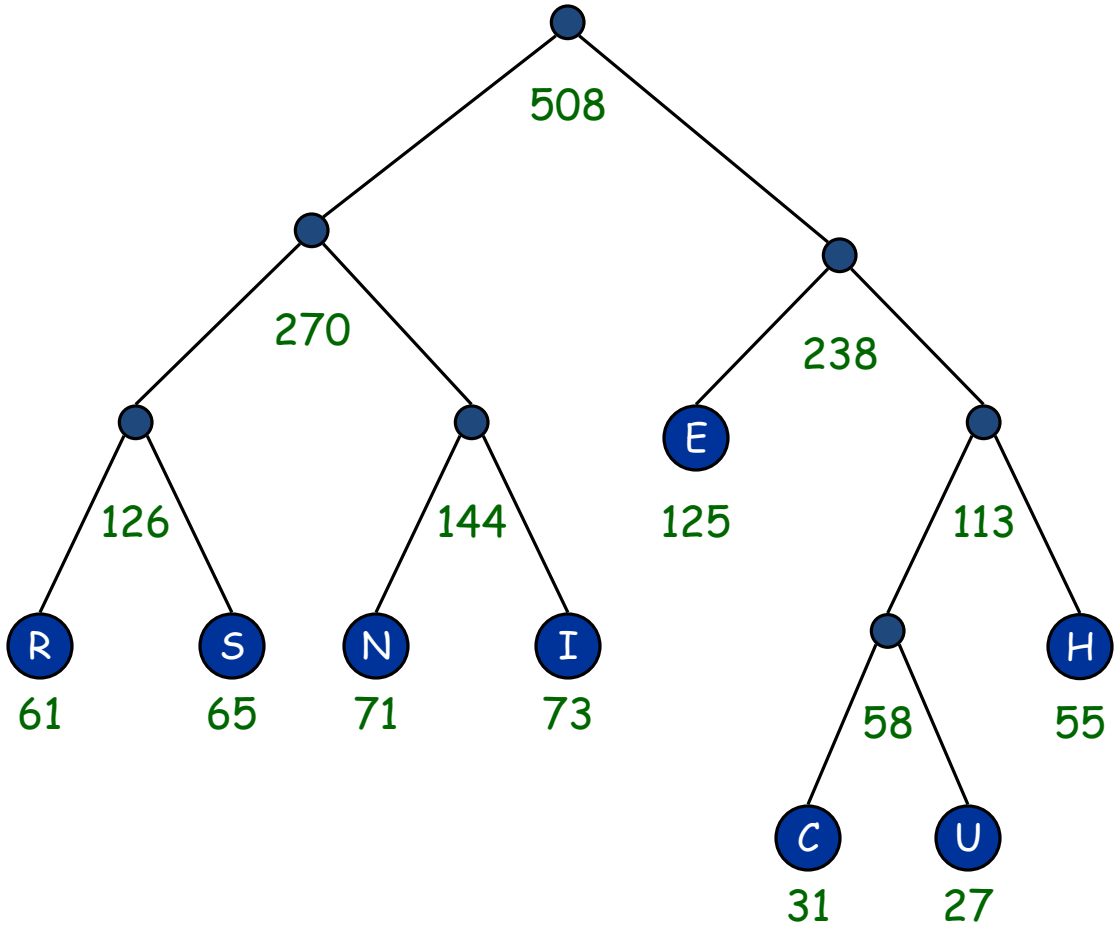


Huffman Code Construction

Char	Freq
	508
	330

	270
	238

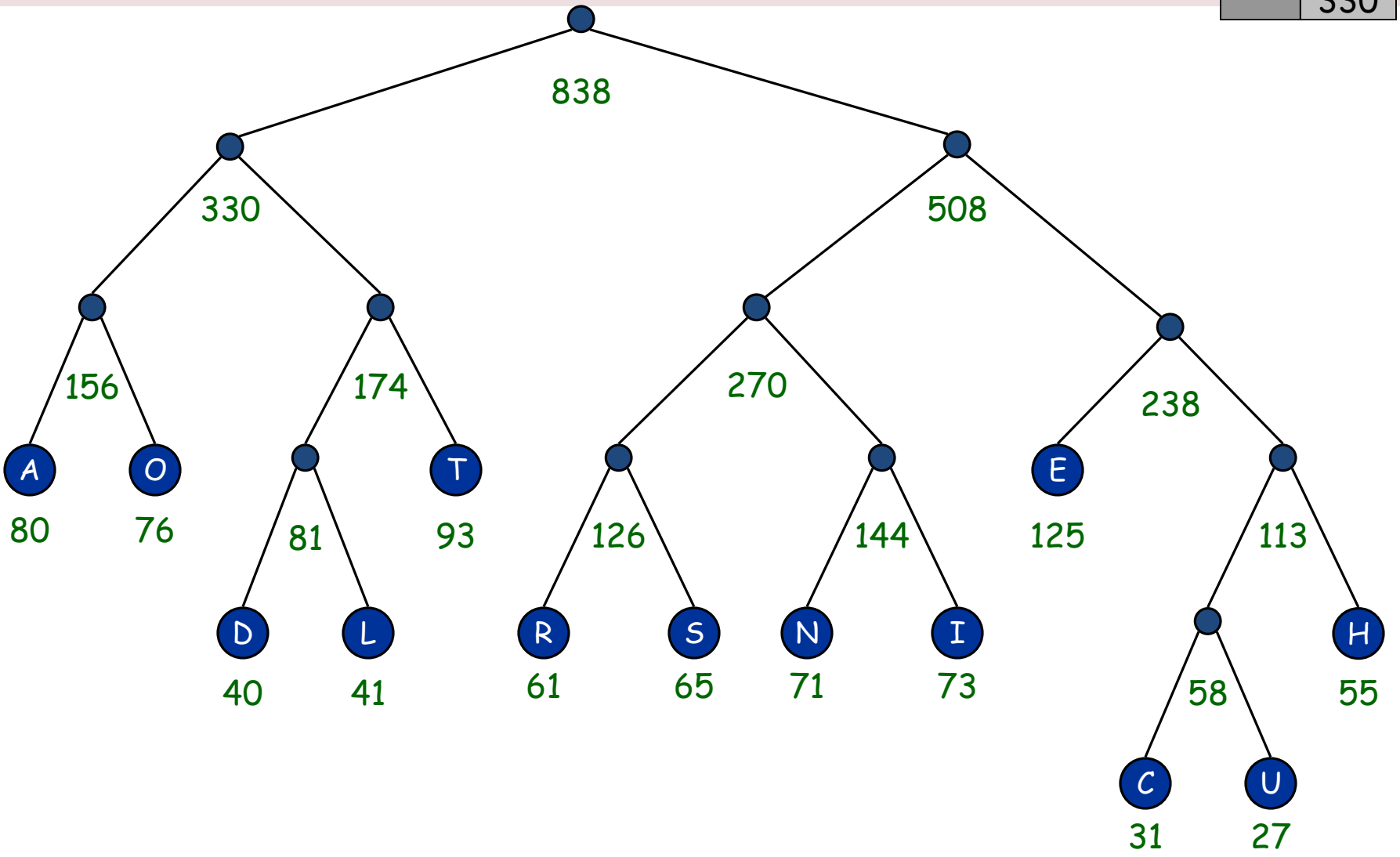




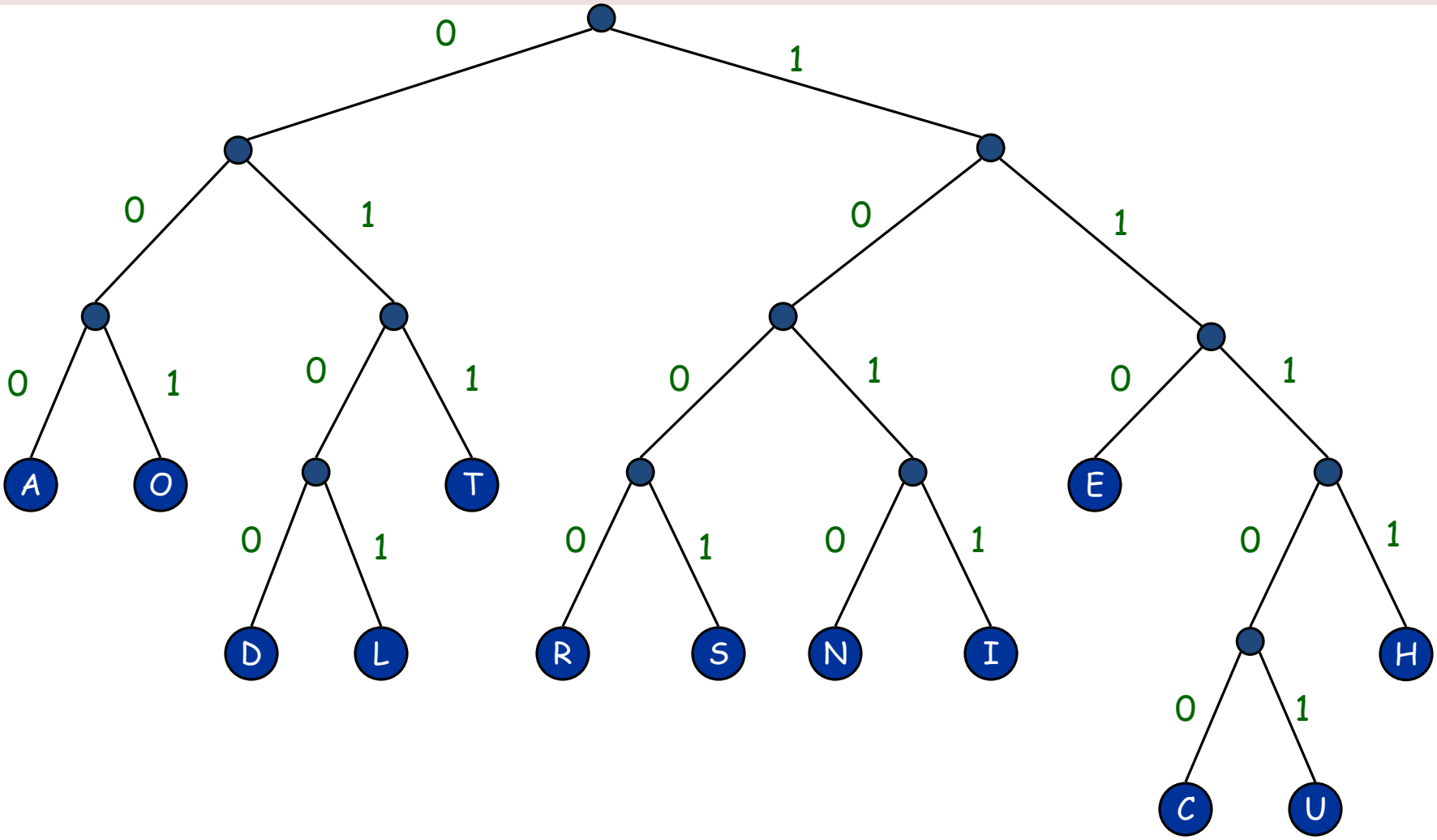
Huffman Code Construction

Char	Freq
	838

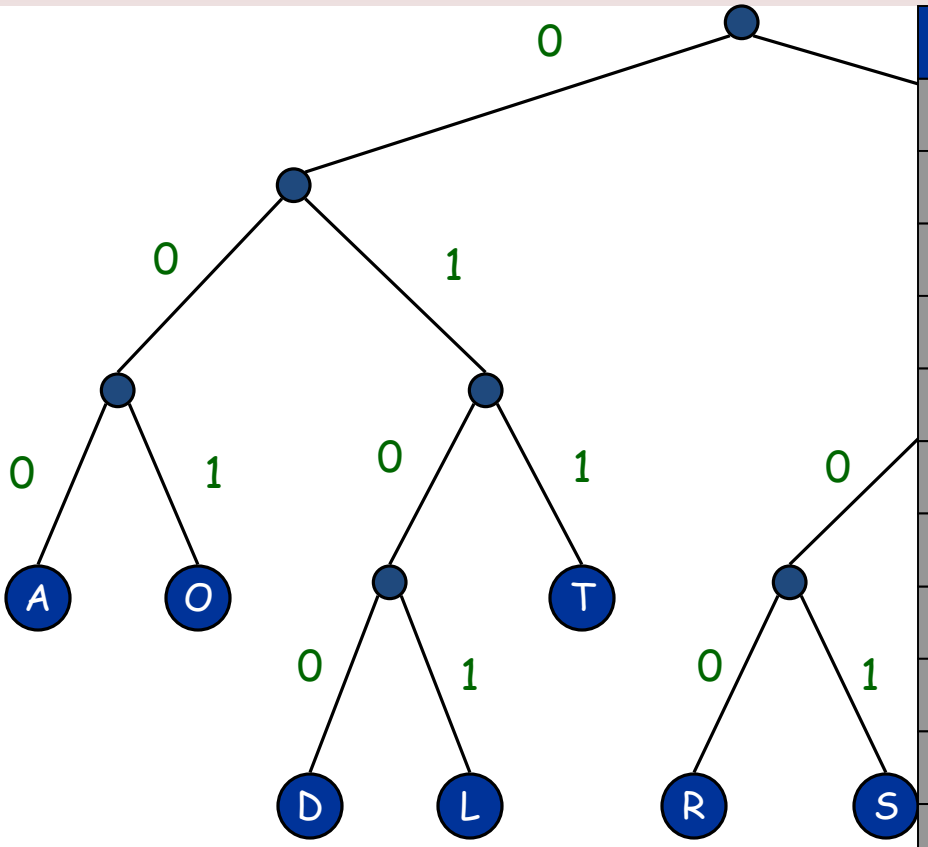
	508
	330



Huffman Code Construction



Huffman Code Construction



Char	Freq	Fixed	Huff
E	125	0000	110
T	93	0001	011
A	80	010	000
O	76	0011	001
I	73	0100	1011
N	71	0101	1010
S	65	0110	1001
R	61	0111	1000
H	55	1000	1111
L	41	1001	0101
D	40	1010	0100
C	31	1011	11100
U	27	1100	11101
Total	838	4.00	3.62

Huffman's Algorithm: Analysis

Recall that the cost of any encoding tree T is $B(T) = n \sum_{x \in C} p(x) d_T(x).$

Need to show that any tree that differs from the one constructed by Huffman's algorithm can be converted into one that is equal to Huffman's tree without increasing its cost

The key is showing that the greedy choice is always the proper one to make (or at least it is as good as any other choice).

Our approach is based a few observations.

1. The Huffman tree is a full binary tree, meaning that every internal node has exactly two children.
2. The two characters with the lowest probabilities will be siblings at the maximum depth in the tree.

Huffman's Algorithm: Analysis

Claim: Consider the two characters, x and y with the smallest probabilities. Then there is an optimal code tree in which these two characters are siblings at the maximum depth in the tree.

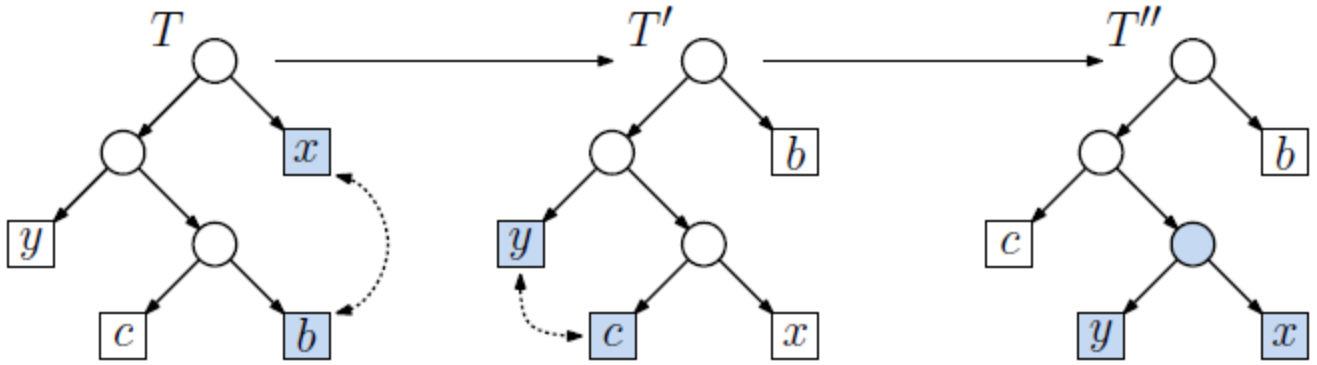


Fig. 27: Showing that the lowest probability nodes are siblings at the tree's lowest level.

Huffman's Algorithm: Analysis

Claim: Huffman's algorithm produces an optimal prefix code tree.

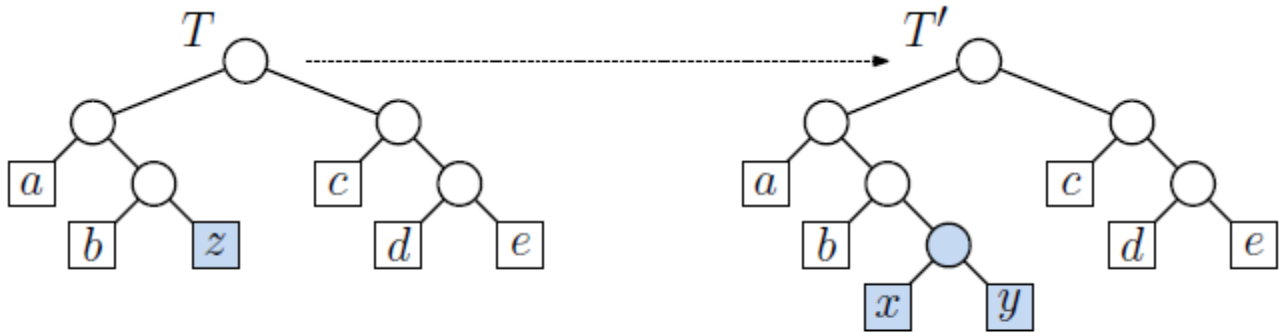


Fig. 28: Proving the correctness of Huffman's algorithm.

Practice

What is the optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a: 1, b:1, c:2, d:3, e:5, f:8, g:13, h:21