DESIGN AND ANALYSIS OF ALGORITHMS

CS 4120/5120

ELEMENTS OF GREEDY STRATEGY

AGENDA

- Elements of greedy strategy
- Huffman codes

ELEMENTS OF GREEDY STRATEGY

- ***Greedy-choice*** property
  - Assemble a globally optimal solution by making locally optimal (greedy) choices.
- ***Optimal substructure***
  - Required, four steps to discover

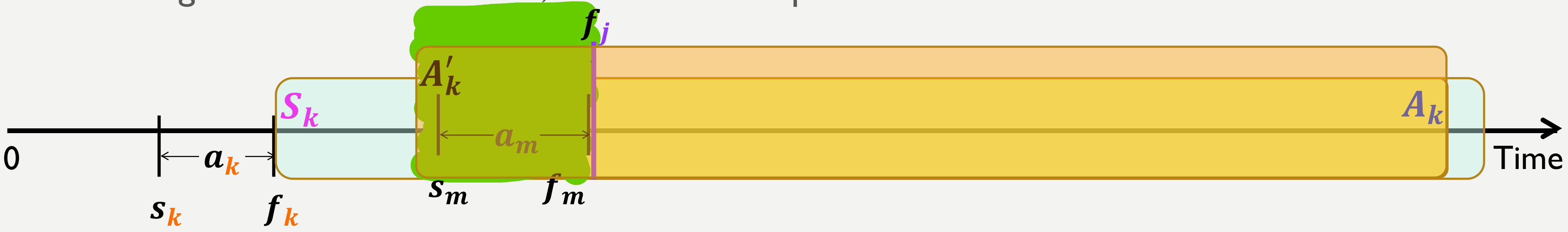
GREEDY-CHOICE PROPERTY

- Make the choice that ***looks best in the current problem***, without considering results from subproblems.
  - The choice may depend on choices so far, but it cannot depend on any future choices.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

GREEDY-CHOICE PROPERTY PROOF

- Prove the optimality of the greedy choice at each step.
  - Start by examining the globally optimal solution to some subproblem.
    - Examine A_k, where $a_j \in A_k$ is with the ***earliest finish time***.
  - Shows how to modify the solution to substitute the greedy choice for some other choice.
    - Substitute a_m for a_j to construct $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$
  - The resulting solution is one similar, but smaller subproblem.



DYNAMIC PROGRAMMING VS GREEDY STRATEGY

- Dynamic programming
  - Key ingredients
    - ***Optimal substructure***
    - Overlapping subproblems
  - Algorithms
    - **Top-down** and **bottom-up**
    - Solve **subproblems first**
  - Making a choice
    - Depend on solutions to subproblems

- Greedy strategy
  - Key ingredients
    - ***Optimal substructure***
    - Greedy choice property
  - Algorithms
    - **Top-down** (recursive and/or iterative)
    - Solve the **current problem first**
  - Making a choice
    - Depend on the choices made so far

DP VS GREEDY THE KNAPSACK PROBLEM

- A thief has broken into a jewelry store trying to take some gemstones.
- He finds three beautiful gemstones, weigh 60 *lbs*, in the store's collection, but he is only able to carry 50 *lbs*.
  - Considering that he needs to carry the tools to commit the break-in.

THE KNAPSACK PROBLEM THE GREEDY THIEF

- The thief now is facing a problem: the three stones weigh differently, they have different values, and he is only able to take up to 50 *lbs*.
  - The table shows the weights of the three stones and their respective values.

Stone	1	2	3
Value	$\$60$	$\$100$	$\$120$
Weight	$10\ lbs$	$20\ lbs$	$30\ lbs$
V/W	$\$6 \text{ per } lbs$	$\$5 \text{ per } lbs$	$\$4 \text{ per } lbs$

- Intuitively, the thief wants to see which item (stone) has the ***greatest value per pound*** as he would like to make a worthy "adventure."

THE 0-1 KNAPSACK PROBLEM

- The thief can only choose to ***take an item or not take it***. (Take it or leave it)
  - Keep in mind that the thief can only take up to 50 *lbs* of stone.
- Given the information below, a greedy thief would choose the item that ***seems the best at the moment*** as it has ***the greatest value per pound***.

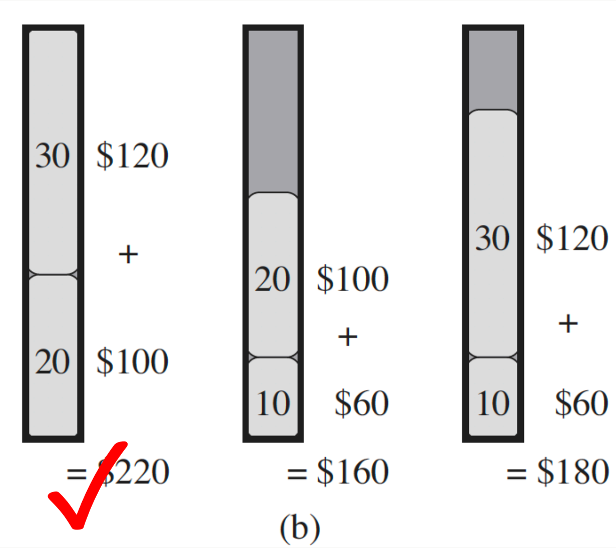
Stone	1	2	3
Value	$\$60$	$\$100$	$\$120$
Weight	$10\ lbs$	$20\ lbs$	$30\ lbs$
V/W	$\$6$ per lbs	$\$5$ per lbs	$\$4$ per lbs

- Would this first greedy choice lead to an optimal solution?

THE 0-1 KNAPSACK PROBLEM SOLUTION CHART

- The chart on the right shows all combinations of items that the thief can take.
- Obviously, taking item 1 is *not included* in the optimal solution.

Stone	1	2	3
Value	\$60	\$100	\$120
Weight	$10\ lbs$	$20\ lbs$	$30\ lbs$
V/W	~~\$6 per *lbs*~~	\$5 per *lbs*	\$4 per *lbs*



- The optimal solution is taking item 2 and 3.
- The greedy-choice property does not hold for 0-1 Knapsack.

SOLVING THE 0-1 KNAPSACK PROBLEM

- The problem definition
  - Given a set of n items. The ith item, $1 \leq i \leq n$ is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers.

i	1	2	…	i	…	j	n
v_i	v_1	v_2	…	v_i	…	v_j	v_n
w_i	w_1	w_2	…	w_i	…	w_j	w_n

  - We want to find ***a maximum-value subset*** (***the most valuable load***)of items that ***weighs at most W pounds***.

DISCOVER THE OPTIMAL SUBSTRUCTURE OF 0-1 KP

- **Step 1**: A solution to the problem consists of making a choice.
- **Step 2**: Suppose that for a given problem, you are given the choice that leads to an optimal solution.
  - Two cases
    - Case 1: ***Suppose*** that we are given the information that item 1 up to i is included in the optimal load.
    - Case 2: ***Suppose*** that we are given the information that item i is NOT included in the optimal load.

i	1	2	…	i	…	j	n
v_i	v_1	v_2	…	v_i	…	v_j	v_n
w_i	w_1	w_2	…	w_i	…	w_j	w_n

DISCOVER THE OPTIMAL SUBSTRUCTURE OF 0-1 KP

- **Step 3**: Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
  - Case 1: ***Suppose*** that we are given the information that item i is included in the optimal load.
    - Let $m[i, W]$ be the ***maximum-value*** knapsack with item i being considered.
    - The subproblem can be characterized as finding ***a maximum-value knapsack*** (***the most valuable load***) of items excluding i that ***weighs at most*** $\underline{\quad W - w_i \quad}$ ***pounds***.
    - $m[i, W] = \underline{\quad m[i-1, W - w_i] \quad} + \underline{\quad v_i \quad}$

DISCOVER THE OPTIMAL SUBSTRUCTURE OF 0-1 KP

- **Step 3**: Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
  - Case 2: ***Suppose*** that we are given the information that item i is **NOT** included in the optimal load.
    - Let $m[i, W]$ be the ***maximum- value*** knapsack with item i being considered.
    - The subproblem can be characterized as finding ***a maximum-value knapsack*** (***the most valuable load***) of items excluding i that ***weighs at most*** ___W___ ***pounds***.
    - $m[i, W] =$ ___$m[i-1, W]$___ $+$ ___$\cancel{\quad}$___

DISCOVER THE OPTIMAL SUBSTRUCTURE OF 0-1 KP

- **Step 3**: Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
  - Taking into account both cases

$$m[i, W] = \begin{cases} 0 & \text{if } i = 0, \\ \max\limits_{1 \le i \le n} \{m[i-1, W - w_i] + v_i, m[i-1, W]\} & \text{if } i \ne 0. \end{cases}$$

DISCOVER THE OPTIMAL SUBSTRUCTURE OF 0-1 KP STEP 4

- **Step 4**: Show the solutions to the subproblem used within an optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique.
  - Case 1: $m[i, W] = m[i-1, W-w_i] + v_i$

i. **Assume** that $m[i-1, W-w_i]$ is NOT the optimal value of subproblem $W - w_i$.

ii. **There exist an optimal solution**, c that is the optimal load obtainable from the items excluding item i, yielding c __>__ $m[i-1, W-w_i]$.

iii. We can **construct a** new solution whose optimal value $m^*[i, W]$ __>__ $m[i, W]$, contradiction.

iv. **Therefore**, $m[i-1, W-w_i]$ is the optimal value of subproblem $W - w_i$

  - Similarly, we can prove case 2.

THE FRACTIONAL KNAPSACK PROBLEM

- The set up is the same, but the thief (may be more professional) can take fractions of items.
  - Rather than having to make a binary (0-1) choice for each item.
- Here we go again, we have the items, their values, and their weights, and ***value per pound*** value for each item.

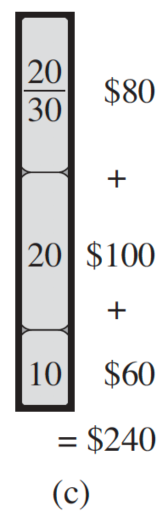
Stone	1	2	3
Value	$\$60$	$\$100$	$\$120$
Weight	$10\ lbs$	$20\ lbs$	$30\ lbs$
V/W	$\$6$ per lbs	$\$5$ per lbs	$\$4$ per lbs

- Would this first greedy choice lead to an optimal solution?

THE FRACTIONAL KNAPSACK GREEDY SOLUTION

Stone	1	2	3
Value	$60	$100	$120
Weight	$10\ lbs$	$20\ lbs$	$30\ lbs$
V/W	$6 per lbs	$5 per lbs	$4 per lbs

- The greedy strategy works as follows.
  - Step 1: The thief takes item 1 and put it in his knapsack.
    - Now he can take up to $50 - 10 = 40\ lbs$
  - Step 2: The thief puts item 2 in his knapsack.
    - Now he can take up to $40 - 20 = 20\ lbs$
  - Step 3: The thief cuts item 3 into 1/3 and 2/3 fractions, then he puts 2/3 of item 3 in his knapsack.
    - Now he can take up to $20 - 30 \times \frac{2}{3} = 0\ lbs$.
- In the end, the thief walks away with $240 worth of gems.



It appears that the ***greedy strategy*** works very well for the ***fractional knapsack*** problem.

SOLVING THE FRACTIONAL KNAPSACK PROBLEM

- The problem definition
  - Given a set of n items. The ith item, $1 \leq i \leq n$ is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers.
  - The items are solved in monotonically increasing order by their value per pound value
    - $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$ for $1 \leq i \leq j \leq n$

i	1	2	…	i	…	j	n
v_i	v_1	v_2	…	v_i	…	v_j	v_n
w_i	w_1	w_2	…	w_i	…	w_j	w_n

  - We want to find ***a maximum-value subset*** (***the most valuable load***)of items that ***weighs at most W pounds***.

SOLVING THE FRACTIONAL KNAPSACK PROBLEM

- Iteratively go through the items starting at 1.
- Pick each item until item i such that $\mathrm{W}' = \sum_{k=1}^{i} w_i + v_{i+1} \geq W$.
- Take a fraction, $\frac{W-W'}{W} \cdot w_{i+1}$ of item $i+1$ to fill the knapsack up to W *lbs*.

NEXTUP
HUFFMAN CODES

DATA ENCODING

- Consider a 100,000-character file that contains only the following 6 different characters: a, b, c, d, e, and f.

- From our programming background, we know that each character can be encoded by 8-bit ASCII code or 16-bit UTF-8/16 code

- If we store the 100,000-character file as it is, how much storage space do we need?

  $8 \text{ bits/character} \times 100{,}000 \text{ characters} = 800{,}000 \text{ bits} = 781.24 \text{ KiB}$

DATA ENCODING
STORING THE CODEWORDS

- If we want to store the 100,000-character compactly, we can **take advantage of the fact that the file contains ONLY a, b, c, d, e, and f**.
  - Suppose that we pre-define ***a fixed-length binary pattern*** for each one of the SIX characters.
    - We would need a __3__-bit pattern to represent one character.
  - The table shows the coding of the characters

Character	a	b	c	d	e	f
Fixed-length codeword	000	001	010	011	100	101

FIXED-LENGTH CODING

- The code used in this scheme is called a ***fixed-length code***.

Character	a	b	c	d	e	f
Fixed-length codeword	000	001	010	011	100	101

  – Each binary string (sequence) is called a ***codeword***.

- How much storage space do we need to store the compressed file?

$$\mathbf{3}\ \text{bits/character} \times 100{,}000\ \text{characters} = \mathbf{300{,}000}\ \text{bits} \approx \mathbf{292.97}\ \textbf{KiB} < 781.24\ \text{KiB}$$

USING THE FREQUENCY

- Suppose that we are given *the* *frequencies* of the characters in the file.

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

  – This table show the number of occurrence (in thousands) of each character.

    • Character a occurs 45,000 times out of the 100,000 characters.

- We can **take advantage of the frequency** information to optimize our coding strategy by using ***shorter binary string to represent more frequently occurring character***.

VARIABLE-LENGTH CODING

- Characters and their binary codes.

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Variable-length codeword	0	101	100	111	1101	1100

  - This strategy is called ***variable-length coding***.
    - The binary string to represent each character is referred to as a ***variable-length codeword***.

VARIABLE-LENGTH CODING
STORAGE SPACE

- Characters and their binary codes.

Character	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Variable-length codeword	0	101	100	111	1101	1100

- How much storage space do we need to store the compressed file?

$$\mathbf{1}\ \text{bit/}\mathtt{a} \times 45\ \text{thou. occr} + \mathbf{3}\ \text{bits/}\mathtt{b} \times 13\ \text{thou. occr} + \mathbf{3}\ \text{bits/}\mathtt{c} \times 12\ \text{thou. occr} +$$

$$\mathbf{3}\ \text{bits/}\mathtt{d} \times 16\ \text{thou. occr} + \mathbf{3}\ \text{bits/}\mathtt{e} \times 9\ \text{thou. occr} + \mathbf{4}\ \text{bits/}\mathtt{f} \times 5\ \text{thou. occr}$$

$$= \mathbf{224}\ \text{thou. bits} = \mathbf{218.75}\ \text{KiB} < 292.97\ \text{KiB} < 781.24\ \text{KiB}$$

VARIABLE-LENGTH CODING
COMPRESSION RATE

- From the example, we can see that given the frequency information, we can use **variable-length codes** to represent characters to **achieve a greater compressing rate**.
  - Note that this is not always the case.

- We need to ***determine the number of bits*** to represent individual character.

PREFIX CODES

- We consider here only ***prefix codes***, meaning that no codeword is also a prefix of some other codeword.
  - Prefix codes are *unambiguous* as a codeword CANNOT be a prefix of another character.
  - Always *achieve optimal* data compression
  - The codeword that begins an encoded file is also *unambiguous*.

ENCODING BINARY CHARACTER CODES

- Here, we consider the problem of designing a ***binary character code***.
  - To encode binary character code, we just concatenate the codewords representing each character of the file.

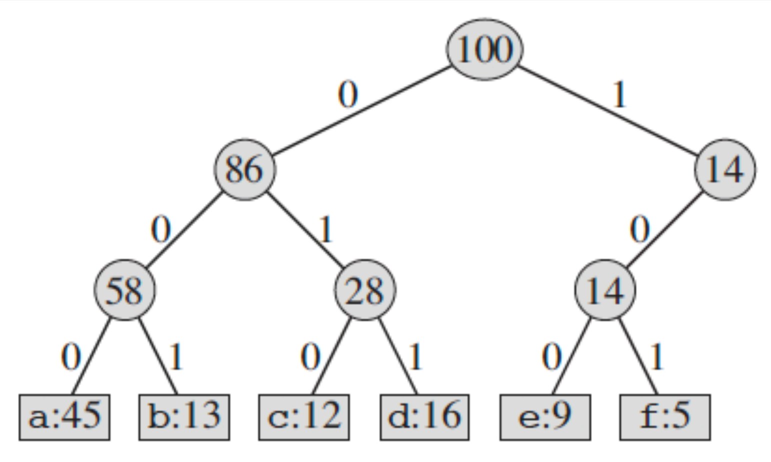
- Example
  - Given the table of variable-length codewords for each character

	a	b	c	d	e	f
Variable-length codeword	0	101	100	111	1101	1100

  - $0 \cdot 101 \cdot 100$ represents $\mathtt{abc}$, where "$\cdot$" denotes concatenation.

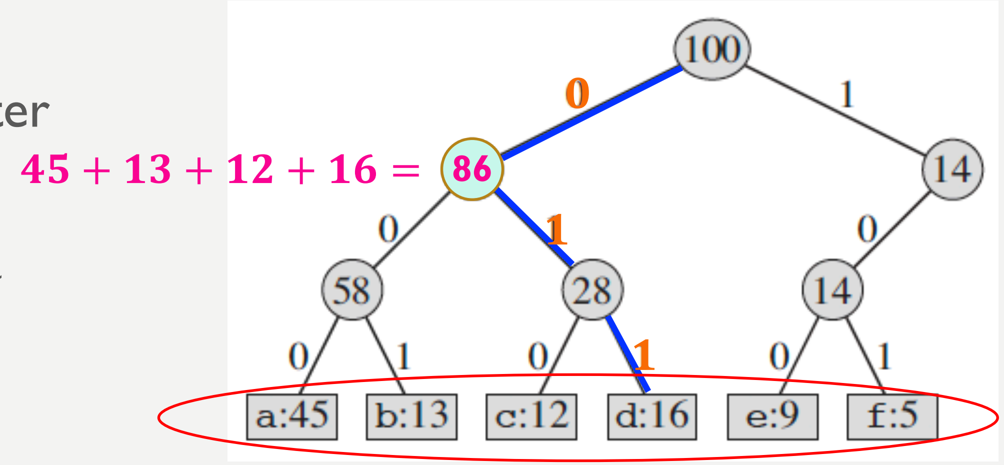
REPRESENTING PREFIX CODES BY A BINARY TREE

- A binary tree whose *leaves are the characters* offers a convenient representation for the prefix code.
- Note that this is **NOT** a binary search tree.
- The key of an internal node is the total occurrences (sometimes normalized) of the characters in the subtree rooted at that internal node.



REPRESENTING PREFIX CODES BY A BINARY TREE

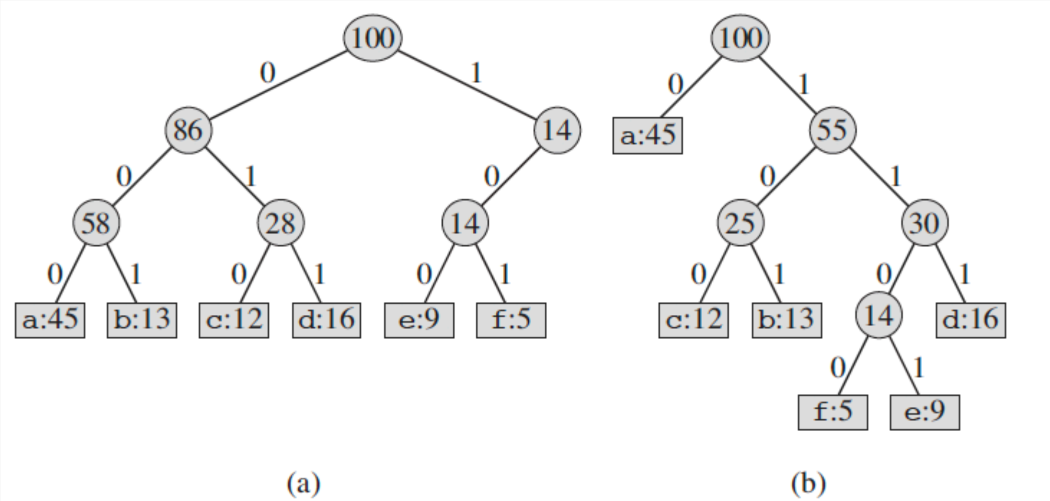
- A binary tree whose *leaves are the characters* offers a convenient representation for the prefix code.
- We interpret the binary codeword for a character as *the simple path from the root to that character*.
  - A number on an edge (a tree branch) indicates a choice
    - “0” means go to the left child
    - “1” means go to the right child



REPRESENTING PREFIX CODES BY A BINARY TREE

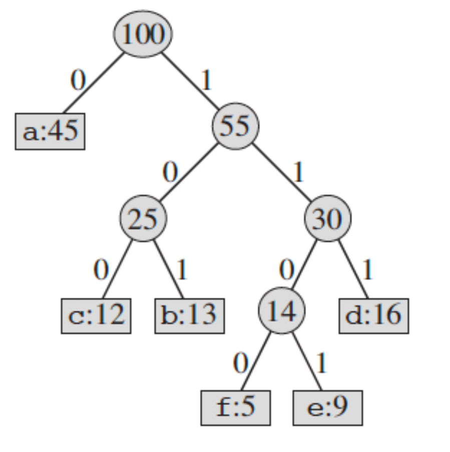
- Consider tree (a) and tree (b) shown below.
  - In (a), the codeword of character `a` is __000__, `d` __011__, and `f` __101__.
  - In (b), the codeword of character `a` is __0__, `d` __111__, and `f` __1100__.

- Obviously, (a) corresponds to a *fixed-length coding* scheme, while (b) corresponds to a *variable-length coding* scheme.



USING FULL BINARY TREE FOR OPTIMAL CODES

- **An *optimal* code** for a file is always represented by ***a full binary tree***.
  - A *full* binary tree is a tree where every non-leaf node has two children.
  - The tree (b) shown below is a *full* binary tree corresponding to an optimal set of prefix codes.

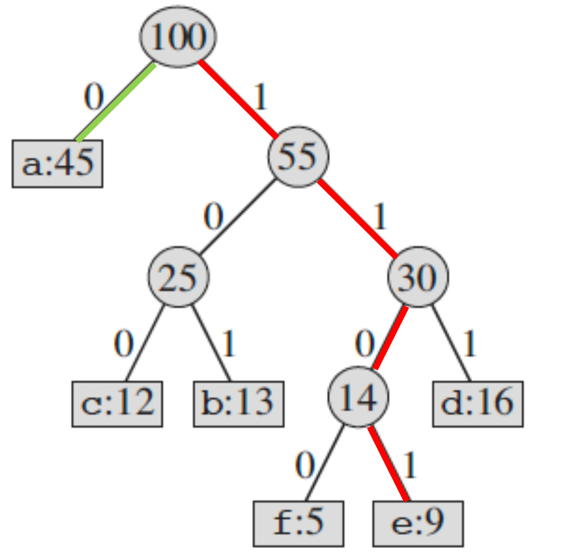


(b)

- If C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree of an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C|-1$ internal nodes.

USING FULL BINARY TREE FOR OPTIMAL CODES (CONT'D)

- We can easily see the number of bits to encode a character from the optimal tree.
  - Character **a**'s optimal prefix code has ________ bit(s)
  - Character **e**'s optimal prefix code has ________ bit(s)



(b)

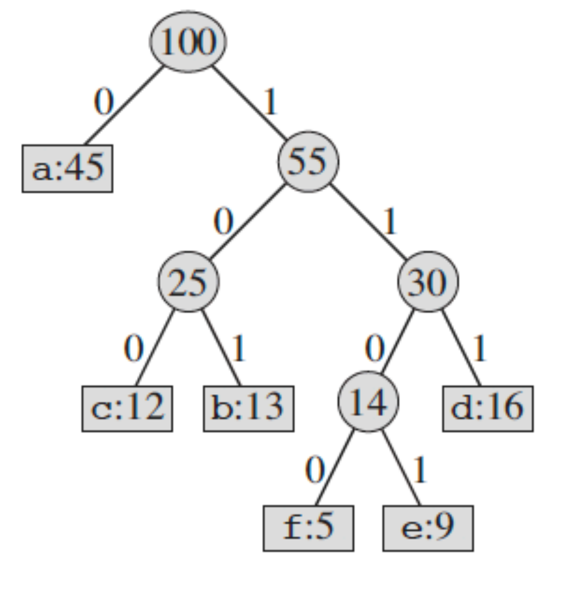
- The **number of bits** (or **length of codeword**) required to encode a character is **the same as the depth** of the leaf node representing that character.

THE COST OF AN OPTIMAL-CODE BINARY TREE

- Given **a tree** T corresponding to a prefix code, we can easily **compute the number of bits** required to encode a file.
  - For **each character** c in the alphabet C
    - Let **attribute** $c.freq$ denote **the frequency** of c in the file
    - Let $d_T(c)$ **denote the depth** of c's leaf in the tree
      - Note that $d_T(c)$ is also the length of the codeword for character c.
  - The number of bits required to encode a file is thus

$$B(T) = \sum c.freq \cdot d_T(c)\,,$$

which we define as the **cost** of the tree T.



(b)

Cost of the tree (b)?

HUFFMAN CODES

- ***Huffman code*** is an optimal prefix code that is constructed by a greedy algorithm invented by David A. Huffman.
  - Surprise! Huffman was born in Ohio in August 1925.

THE HUFFMAN ALGORITHM

- Input
  - An alphabet denoted by $\boldsymbol{C}$
    - C is **a set of** n characters
    - Each character $\boldsymbol{c} \in C$ is an object with **an attribute** $c.\boldsymbol{freq}$.

- Like a normal greedy algorithm, HUFFMAN algorithm builds an optimal prefix code in a ***bottom-up*** manner.

HUFFMAN $(\boldsymbol{C})$

```
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left  = x = EXTRACT-MIN (Q)
    z.right = y = EXTRACT-MIN (Q)
    z.freq  = x.freq + y.freq
    INSERT (Q, z)
return EXTRACT-MIN (Q) // return the root of the tree
```

DATA STRUCTURE USED BY THE HUFFMAN ALGORITHM

- The algorithm uses a ***min-priority queue*** $\boldsymbol{Q}$, keyed on the $freq$ attribute to identify the two least-frequent object .
  - A **queue** is a ***first-in-first-out*** (FIFO) data structure.
  - In addition to being a regular queue, each element of a min-priority queue has a "***priority***" associated with it.

HUFFMAN (C)
$n = \|C\|$
$\boldsymbol{Q} = C$
for $i = 1$ **to** $n - 1$
allocate a new node z
$z.left \;= x =$ **EXTRACT-MIN** $(\boldsymbol{Q})$
$z.right = y =$ **EXTRACT-MIN** $(\boldsymbol{Q})$
$z.freq \;= x.freq + y.freq$
INSERT $(\boldsymbol{Q}, \boldsymbol{z})$
return EXTRACT-MIN $(\boldsymbol{Q})$// return the root of the tree

MERGING TWO OBJECTS IN THE QUEUE

- When the algorithm identifies the two characters with ***the least frequencies***, it "**merges**" the two objects.
  - The result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN (C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left  = x = EXTRACT-MIN (Q)
    z.right = y = EXTRACT-MIN (Q)
    z.freq  = x.freq + y.freq
    INSERT (Q, z)
return EXTRACT-MIN (Q) // return the root of the tree
```

	a	b	c	d	e	f
Freq.	45	13	12	16	9	5

	a	b	c	d	merge(e,f)
Freq.	45	13	12	16	14

THE HUFFMAN ALGORITHM

INITIALIZATION

	a	b	c	d	e	f
Freq.	45	13	12	16	9	5

- Suppose the $\boldsymbol{Q}$ is implemented by a MIN-HEAP
  - Modify the BUILD-MAX-HEAP and MAX-HEAPIFY pseudocodes to allow the building of a **MIN**-HEAP.
  - Follow the procedure of building a MIN-HEAP.
- Min-priority queue $\boldsymbol{Q} = \{f, e, c, b, d, a\}$
  - Iteration $i = 3$, $\boldsymbol{Q} = \{a, b, f, d, e, c\}$
  - Iteration $i = 2$, $\boldsymbol{Q} = \{a, e, f, d, b, c\}$
  - Iteration $i = 1$, $\boldsymbol{Q} = \{f, e, c, d, b, a\}$

BUILD-**MIN**-HEAP (A)
$A.heap\text{-}size = A.length$
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
MIN-HEAPIFY (A, i)

MIN-HEAPIFY (A, i)
$l =$ LEFT (i)
$r =$ RIGHT (i)
if $l \leq A.leap\text{-}size$ and $A[l] < A[i]$
$smallest = l$
else $smallest = i$
if $r \leq A.heap\text{-}size$ and $A[r] < A[smallest]$
$smallest = r$
if $smallest \neq i$
exchange $A[i]$ with $A[smallest]$
MIN-HEAPIFY $(A, smallest)$

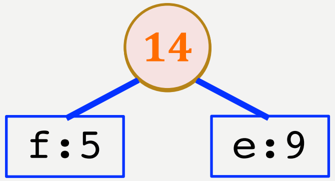
THE HUFFMAN ALGORITHM

IN ACTION $i = 1$

	a	b	c	d	~~e~~	~~f~~
Freq.	45	13	12	16	~~9~~	~~5~~

- Min-priority queue $\boldsymbol{Q} = \{\cancel{f}, \cancel{e}, c, d, b, a\}$
- After two extractions (each involves a MIN-HEAPIFY), min-priority queue $\boldsymbol{Q} = \{c, d, b, a\}$
- **Insert** $\boldsymbol{z}$ (process involves a MIN-HEAPIFY) min-priority queue $\boldsymbol{Q} = \{c, b, d, \boldsymbol{z}, a\}$
- Visualization

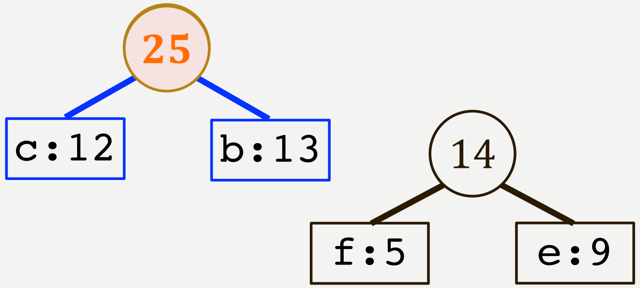
HUFFMAN (C)
$n = \lvert C \rvert$
$Q = C$
for $i = 1$ **to** $n - 1$
allocate a new node $\boldsymbol{z}$
$z.left = x =$ EXTRACT-MIN (Q)
$z.right = y =$ EXTRACT-MIN (Q)
$\boldsymbol{z}.freq = x.freq + y.freq$
INSERT $(\boldsymbol{Q}, \boldsymbol{z})$
return EXTRACT-MIN (Q) // return the root of the tree



THE HUFFMAN ALGORITHM
IN ACTION $i = 2$

	a	~~b~~	~~c~~	d	~~e~~	~~f~~
Freq.	45	~~13~~	~~12~~	16	~~9~~	~~5~~

- Min-priority queue $\boldsymbol{Q} = \{\cancel{c}, \cancel{b}, d, \mathbf{14}, a\}$
- After two extractions (each involves a MIN-HEAPIFY),
  min-priority queue $\boldsymbol{Q} = \{\mathbf{14}, d, a\}$
- **Insert** $\boldsymbol{z}$ (process involves a MIN-HEAPIFY)
  min-priority queue $\boldsymbol{Q} = \{14, \boldsymbol{z}, d, a\}$
- Visualization



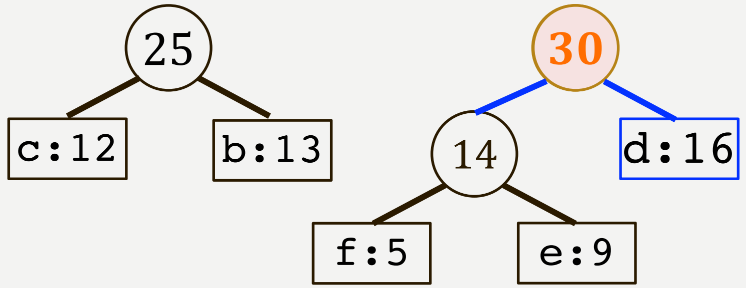
HUFFMAN (C)
$n = \lvert C \rvert$
$Q = C$
for $i = 1$ **to** $n - 1$
allocate a new node $\boldsymbol{z}$
$z.left = x =$ EXTRACT-MIN (Q)
$z.right = y =$ EXTRACT-MIN (Q)
$\boldsymbol{z}.freq = x.freq + y.freq$
INSERT $(\boldsymbol{Q}, \boldsymbol{z})$
return EXTRACT-MIN (Q) // return the root of the tree

THE HUFFMAN ALGORITHM
IN ACTION $i = 3$

	a	b	c	d	e	f
Freq.	45	~~13~~	~~12~~	~~16~~	~~9~~	~~5~~

- Min-priority queue $\boldsymbol{Q} = \{\cancel{14}, \mathbf{25}, \cancel{d}, a\}$
- After two extractions
  min-priority queue $\boldsymbol{Q} = \{25, a\}$
- **Insert** $\boldsymbol{z}$, min-priority queue $\boldsymbol{Q} = \{25, z, a\}$
- Visualization

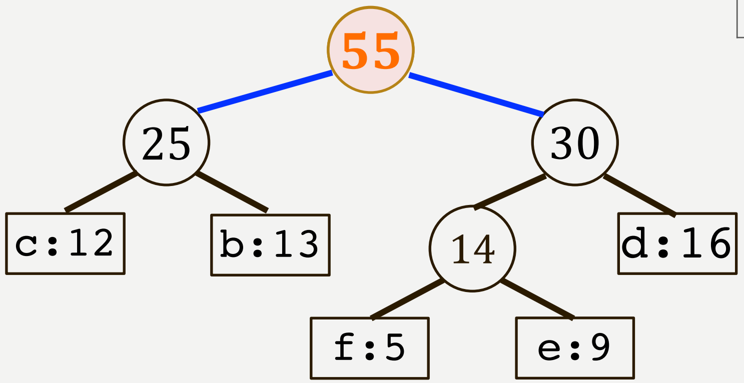
```
HUFFMAN (C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left  = x = EXTRACT-MIN (Q)
    z.right = y = EXTRACT-MIN (Q)
    z.freq  = x.freq + y.freq
    INSERT (Q, z)
return EXTRACT-MIN (Q) // return the root of the tree
```



THE HUFFMAN ALGORITHM IN ACTION $i = 4$

	a	b	c	d	e	f
Freq.	45	~~13~~	~~12~~	~~16~~	~~9~~	~~5~~

- Min-priority queue $\boldsymbol{Q} = \{\cancel{25}, \cancel{30}, a\}$
- After two extractions
  min-priority queue $\boldsymbol{Q} = \{a\}$
- **Insert** $\boldsymbol{z}$, min-priority queue $\boldsymbol{Q} = \{a, \boldsymbol{z}\}$
- Visualization



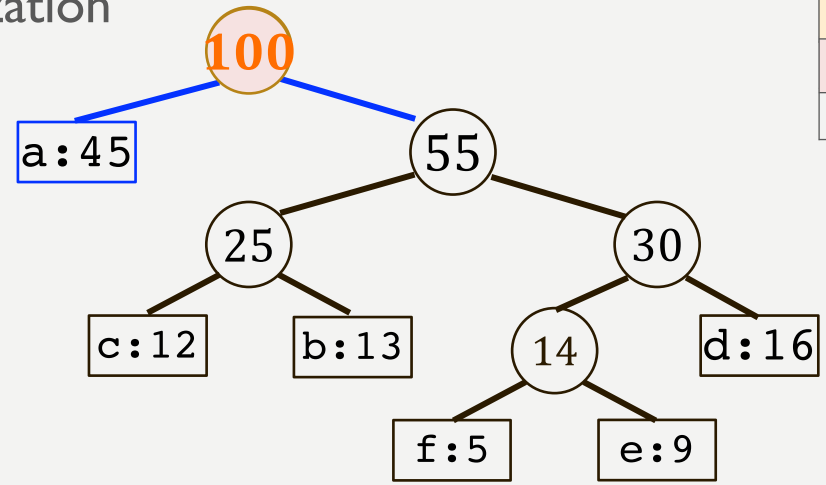
HUFFMAN (C)

```
n = |C|
Q = C
for i = 1 to n − 1
        allocate a new node z
        z.left  = x = EXTRACT-MIN (Q)
        z.right = y = EXTRACT-MIN (Q)
        z.freq  = x.freq + y.freq
        INSERT (Q, z)
return EXTRACT-MIN (Q) // return the root of the tree
```

THE HUFFMAN ALGORITHM
IN ACTION $i = 5$

	a	b	c	d	e	f
Freq.	~~45~~	~~13~~	~~12~~	~~16~~	~~9~~	~~5~~

- Min-priority queue $Q = \{\cancel{a}, \cancel{55}\}$
- After two extractions min-priority queue $Q = \emptyset$
- **Insert z**, min-priority queue $Q = \{z\}$
- Visualization

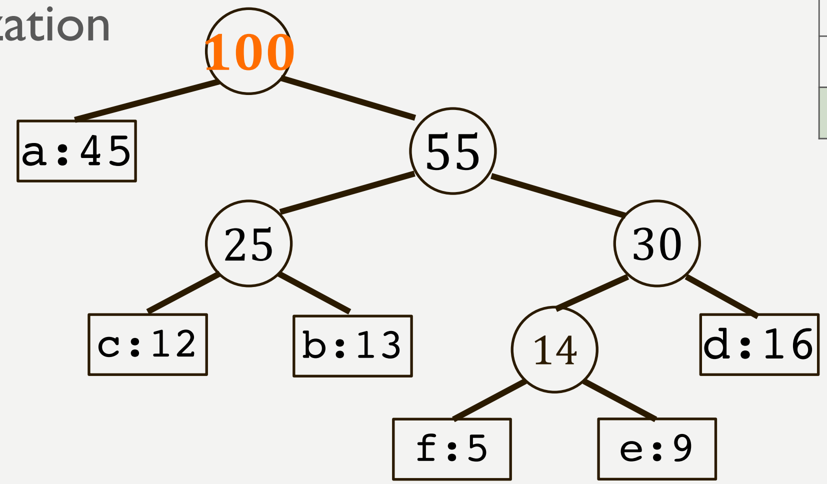


HUFFMAN (C)
$n = \lvert C \rvert$
$Q = C$
for $i = 1$ **to** $n - 1$
allocate a new node z
$z.left = x =$ EXTRACT-MIN (Q)
$z.right = y =$ EXTRACT-MIN (Q)
$z.freq = x.freq + y.freq$
INSERT (Q, z)
return EXTRACT-MIN (Q) // return the root of the tree

THE HUFFMAN ALGORITHM
IN ACTION TERMINATION

- Min-priority queue $\boldsymbol{Q} = \{\cancel{\mathbf{100}}\}$
- Return **100**

- Visualization



HUFFMAN (C)
$n = \lvert C \rvert$
$Q = C$
for $i = 1$ **to** $n - 1$
allocate a new node z
$z.left \quad = x =$ EXTRACT-MIN (Q)
$z.right = y =$ EXTRACT-MIN (Q)
$z.freq \;= x.freq + y.freq$
INSERT (Q, z)
return EXTRACT-MIN (Q) // return the root of the tree

THE HUFFMAN ALGORITHM
RUNNING TIME

- The process can be abstracted as

	Cost	Time
Build Min-Priority Queue	$f(\lvert C \rvert)$	1
for $i = 1$ **to** $n - 1$	$\Theta(1)$	n
EXTRACT-MIN (Q)	$g(\lvert Q \rvert)$	$n - 1$
EXTRACT-MIN (Q)	$g(\lvert Q \rvert)$	$n - 1$
INSERT (Q, z)	$s(\lvert Q \rvert)$	$n - 1$
return EXTRACT-MIN (Q)	$g(\lvert Q \rvert)$	1

- The running time function of HUFFMAN

$$T(|C|) = f(|C|) + \Theta(n) + \Theta\big(n \cdot g(|C|)\big) + \Theta\big(n \cdot s(|Q|)\big)$$

, where f, g, and s are the cost of building a ***min-priority queue*** $\boldsymbol{Q}$, extracting the min of $\boldsymbol{Q}$, and inserting a node to the ***min-priority queue*** $\boldsymbol{Q}$, respectively.

HUFFMAN (C)
$n = \lvert C \rvert$
$Q = C$
for $i = 1$ **to** $n - 1$
allocate a new node z
$z.left = x =$ EXTRACT-MIN (Q)
$z.right = y =$ EXTRACT-MIN (Q)
$z.freq = x.freq + y.freq$
INSERT (Q, z)
return EXTRACT-MIN (Q) // return the root of the tree

THE HUFFMAN ALGORITHM RUNNING TIME (CONT'D)

- The running time function of HUFFMAN depends on the implementation of the ***min-priority queue Q***.

- Implementing the ***min-priority queue*** using a MIN-HEAP (similar to MAX-HEAP)
  - Suppose $|C| = n$
  - $T(|C|) = f(|C|) + \Theta(n) + \Theta(n \cdot g(|C|)) + \Theta(n \cdot s(|Q|))$

$$= \underset{\text{Cost of building a MIN-HEAP}}{\underline{O(n \lg n)}} + \Theta(n) + \Theta(n \cdot \underset{\text{Cost of extracting the min of MIN-HEAP}}{\underline{O(\lg n)}}) + \Theta(n \cdot \underset{\text{Cost of inserting a node to a MIN-HEAP}}{\underline{O(\lg n)}}) = \underline{\qquad\qquad} = T(n).$$

HUFFMAN (C)
$n = \lvert C \rvert$
$Q = C$
for $i = 1$ **to** $n - 1$
allocate a new node z
$z.left \;= x =$ EXTRACT-MIN (Q)
$z.right = y =$ EXTRACT-MIN (Q)
$z.freq \;= x.freq + y.freq$
INSERT (Q, z)
return EXTRACT-MIN (Q) // return the root of the tree

THE HUFFMAN ALGORITHM
PRACTICE

	a	b	c	d
Freq.%	26	19	34	21

HUFFMAN (C)
$n = \|C\|$
$Q = C$
for $i = 1$ **to** $n - 1$
allocate a new node z
$z.left \;\; = x =$ EXTRACT-MIN (Q)
$z.right = y =$ EXTRACT-MIN (Q)
$z.freq \;\; = x.freq + y.freq$
INSERT (Q, z)
return EXTRACT-MIN (Q)

- Consider a file containing letters drawn from an alphabet $C = \{a, b, c, d\}$. (See the frequencies in the table).
- Follow the HUFFMAN algorithm and **fill the table**.
  - Draw the corresponding tree while completing the table,
  - The Q (implemented as a MIN-HEAP) column contains the min-priority queue at the end of the iteration.
  - Use $z.freq$ as the element after z is inserted in Q.

Iteration	$z.left$	$z.right$	Q

THE HUFFMAN ALGORITHM PRACTICE

	a	b	c	d
Freq.%	26	19	34	21

```
HUFFMAN (C)
n = |C|
Q = C
for i = 1 to n − 1
        allocate a new node z
        z.left  = x = EXTRACT-MIN (Q)
        z.right = y = EXTRACT-MIN (Q)
        z.freq  = x.freq + y.freq
        INSERT (Q, z)
return EXTRACT-MIN (Q)
```

- Consider a file containing letters drawn from an alphabet $C = \{a, b, c, d\}$. (See the frequencies in the table).
- Follow the HUFFMAN algorithm and **fill the table**.
  - Draw the corresponding tree while completing the table,
  - The Q (implemented as a MIN-HEAP) column contains the min-priority queue at the end of the iteration.
  - Use $z.freq$ as the element after z is inserted in Q.

Iteration	$z.left$	$z.right$	Q
$i = 1$	b	d	$\{a, 40, c\}$
$i = 2$	a	c	$\{40,60\}$
$i = 3$	40	60	$\{100\}$

THE HUFFMAN ALGORITHM
PRACTICE

	a	b	c	d
Freq.%	26	19	34	21

HUFFMAN (C)

- $n = |C|$
- $Q = C$
- **for** $i = 1$ **to** $n - 1$
  - allocate a new node z
  - $z.left \;\; = x =$ EXTRACT-MIN (Q)
  - $z.right = y =$ EXTRACT-MIN (Q)
  - $z.freq \;\; = x.freq + y.freq$
  - INSERT (Q, z)
- **return** EXTRACT-MIN (Q)

- Consider a file containing letters drawn from an alphabet $C = \{a, b, c, d\}$. (See the frequencies in the table).
- **Show** the tree corresponding to the HUFFMAN codes
- **Compute the cost** $B(T)$. Show your work. Arrange the addition terms in alphabetical order.
  $B(T) =$
- Show the code word of each letter by completing the table.

	a	b	c	d
Freq.%	26	19	34	21
Codeword				

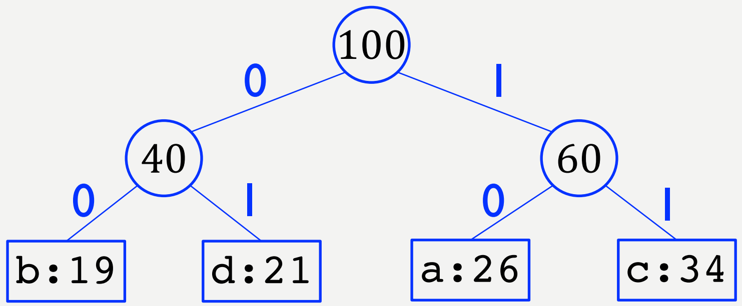
THE HUFFMAN ALGORITHM
PRACTICE

	a	b	c	d
Freq.%	26	19	34	21

```
HUFFMAN (C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left  = x = EXTRACT-MIN (Q)
    z.right = y = EXTRACT-MIN (Q)
    z.freq  = x.freq + y.freq
    INSERT (Q, z)
return EXTRACT-MIN (Q)
```

- Consider a file containing letters drawn from an alphabet $C = \{a, b, c, d\}$. (See the frequencies in the table).
- **Show** the tree corresponding to the HUFFMAN codes
- **Compute the cost** $B(T)$. Show your work. Arrange the addition terms in alphabetical order.
  $B(T) = 26\times2 + 19\times2 + 34\times2 + 21\times2 = 200$
- Show the code word of each letter by completing the table.

	a	b	c	d
Freq.%	26	19	34	21
Codeword	10	00	11	01



NEXT UP
ELEMENTARY GRAPH ALGORITHMS

REFERENCE

- https://www.netclipart.com/isee/hRwxRh_kids-clipart-nurse-cute-female-doctor-cartoon/
- https://listposts.com/lera-kiryakova-celebrities-cartoon-characters/
- https://www.computerhope.com/people/david_huffman.htm