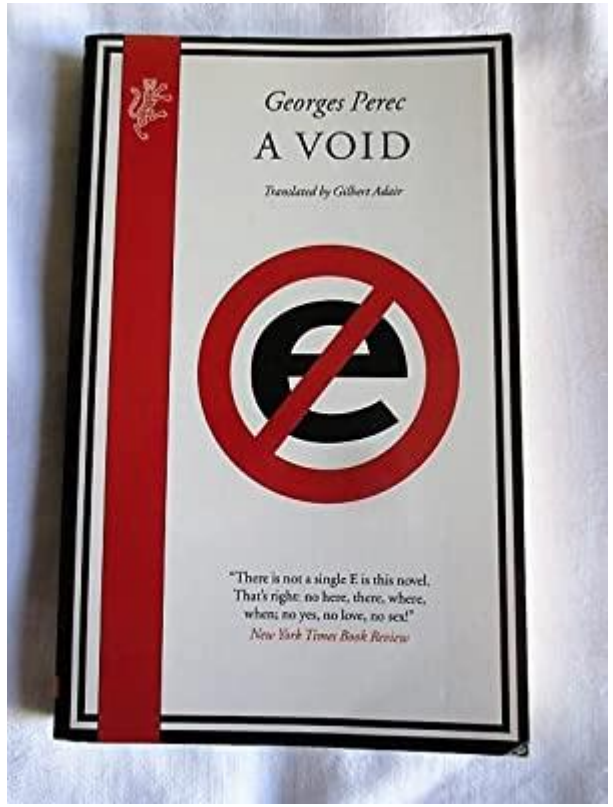
Compression

- Scenario: communicating over "expensive" line
- Goal: zip function, $f: \Sigma \to \{0,1\}^*$
- Requirement: easily invertible (i.e., f^{-1})
- Hope: good compression (i.e., $|f(text)| \ll |text|$)
- Impossible, why?
  - f must be injective
  - If some strings are "zipped" others **must** "expand"

Huffman Coding

Huffman Coding

- Main idea: chars in human text do not distribute uniformly
- Use corpus to build compression function $H: \Sigma \to \{0,1\}^*$
- Frequent letters → short encoding
- Alice: compress message with $H(msg)$
- Bob: decompress message (with H^{-1})
- Which corpus?



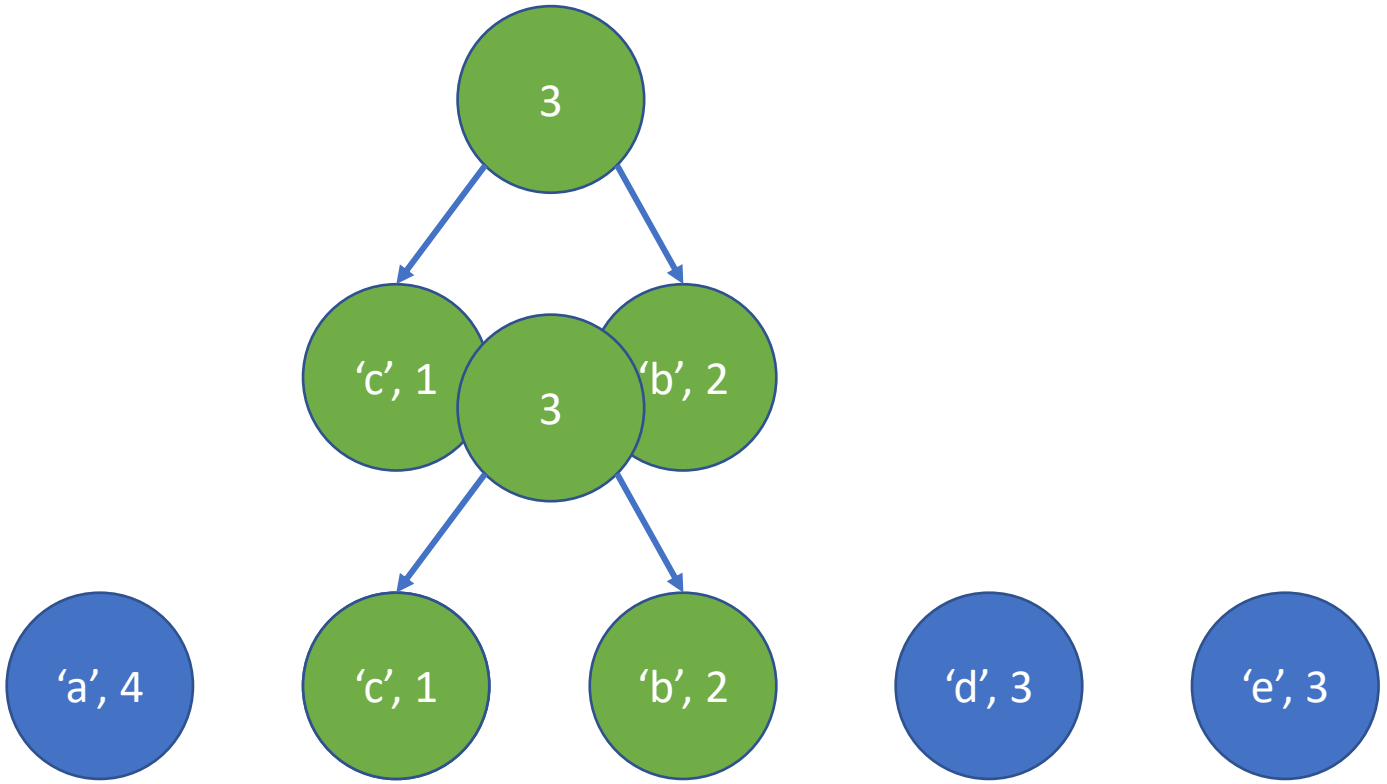
Huffman Coding

corpus = ‘aaaabbcdddeee’

Pseudocode:

```
build forest from corpus
while |forest| > 1:
    extract 2 min trees: t1, t2
    union t’ = t1 + t2
    put t’ in forest
```

Huffman Coding



corpus = 'aaaabbcdddeee'

Pseudocode:

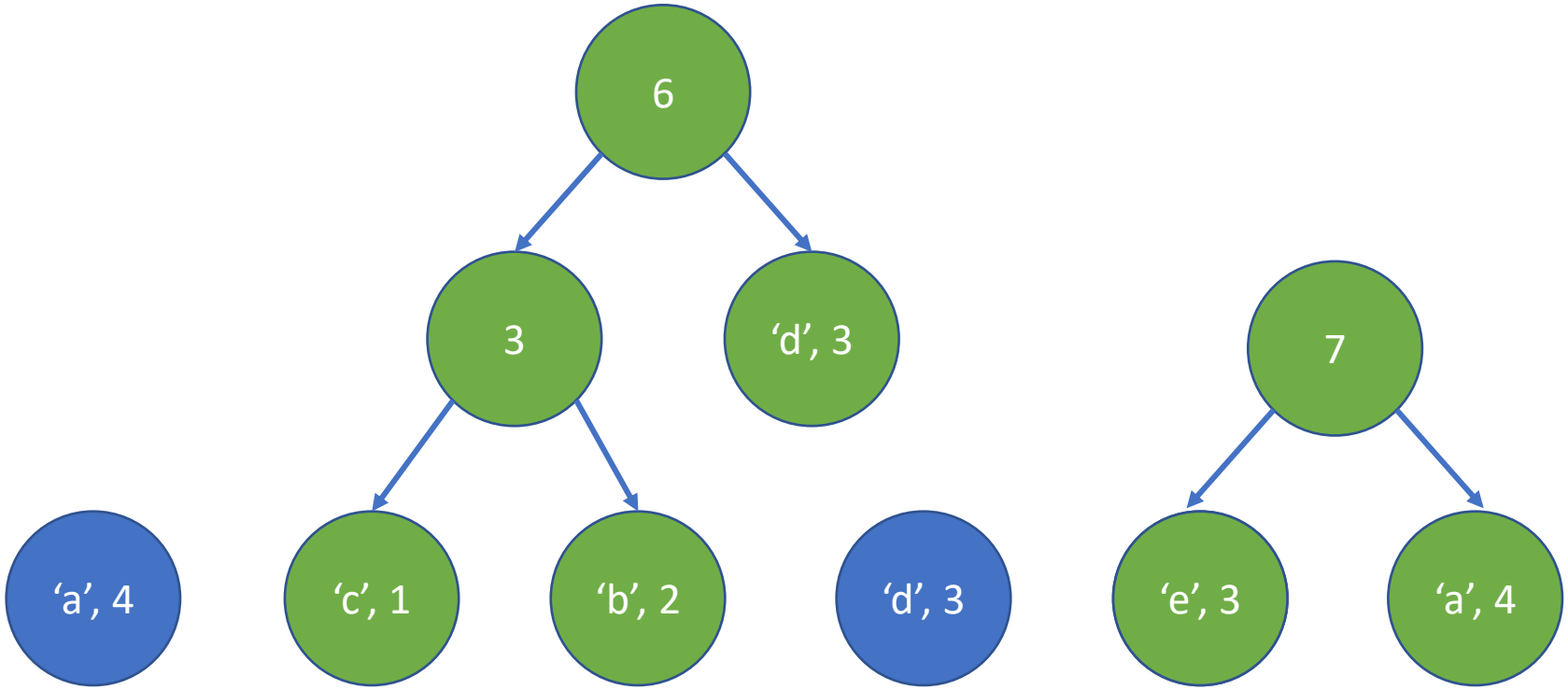
```
build forest from corpus
while |forest| > 1:
    extract 2 min trees: t1, t2
    union t' = t1 + t2
    put t' in forest
```

Huffman Coding

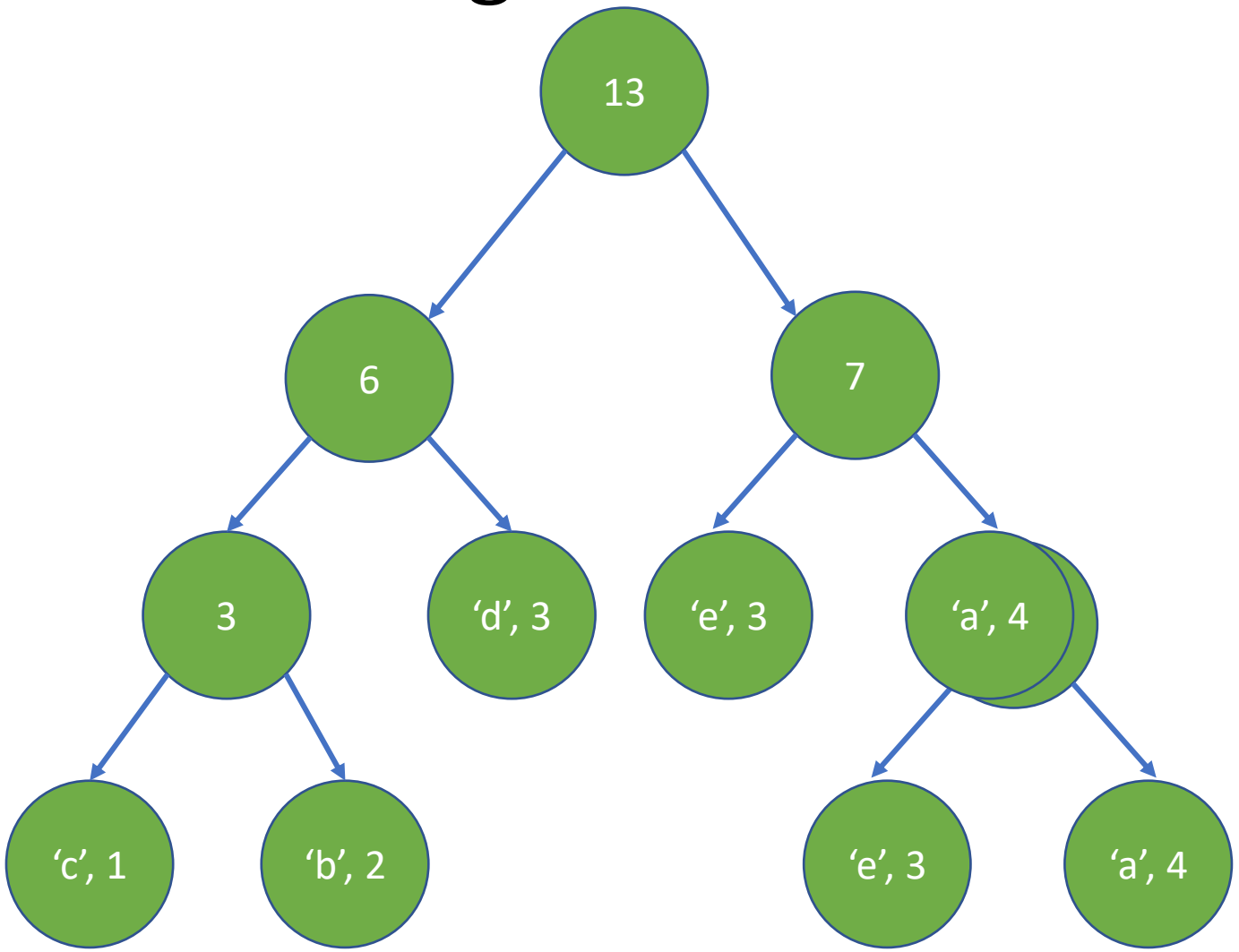
corpus = ‘aaaabbcdddeee’

Pseudocode:

```
build forest from corpus
while |forest| > 1:
    extract 2 min trees: t1, t2
    union t’ = t1 + t2
    put t’ in forest
```



Huffman Coding

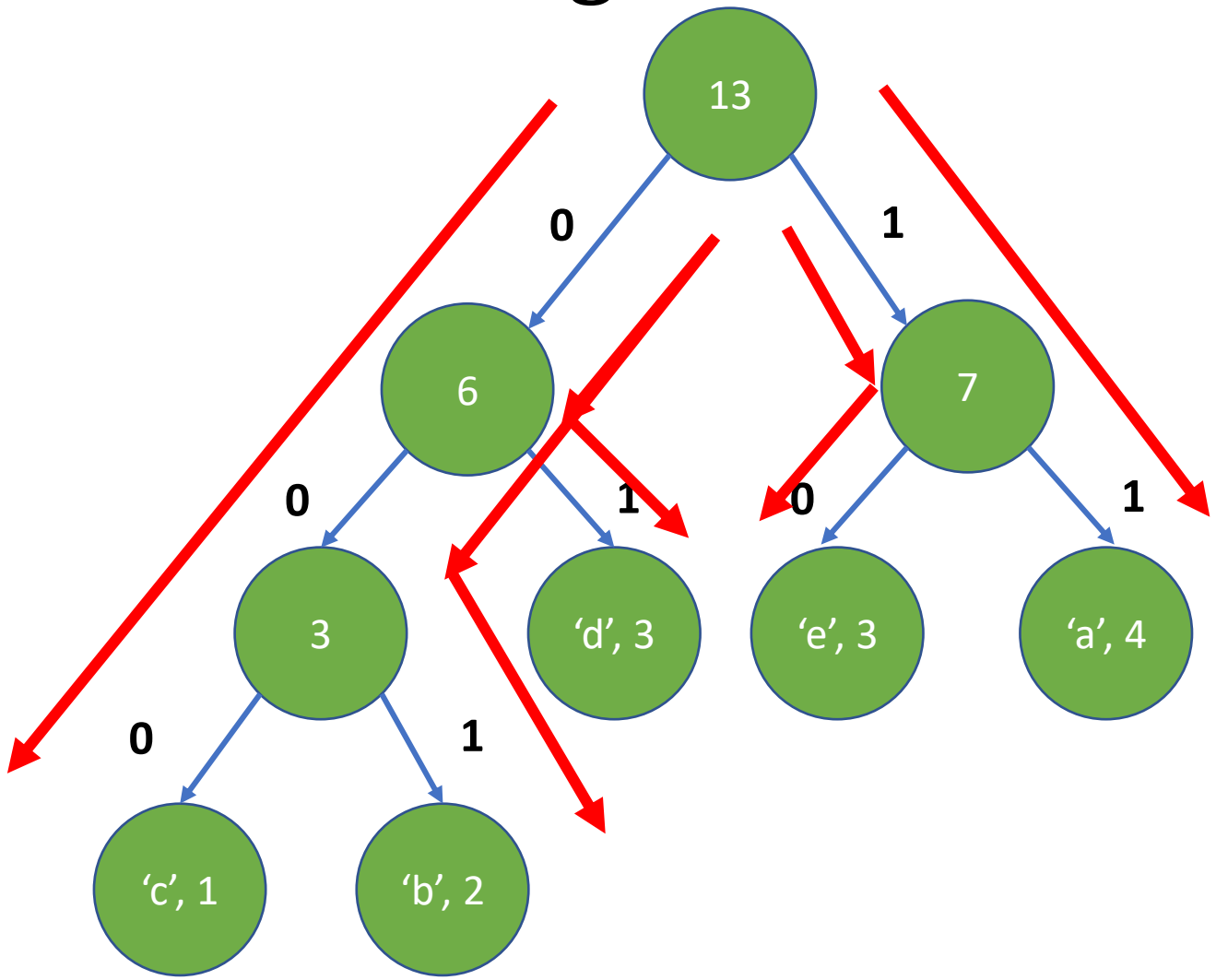


corpus = 'aaaabbcdddeee'

Pseudocode:

```
build forest from corpus
while |forest| > 1:
    extract 2 min trees: t1, t2
    union t' = t1 + t2
    put t' in forest
```

Huffman Coding



corpus = 'aaaabbcdddeee'

Pseudocode:

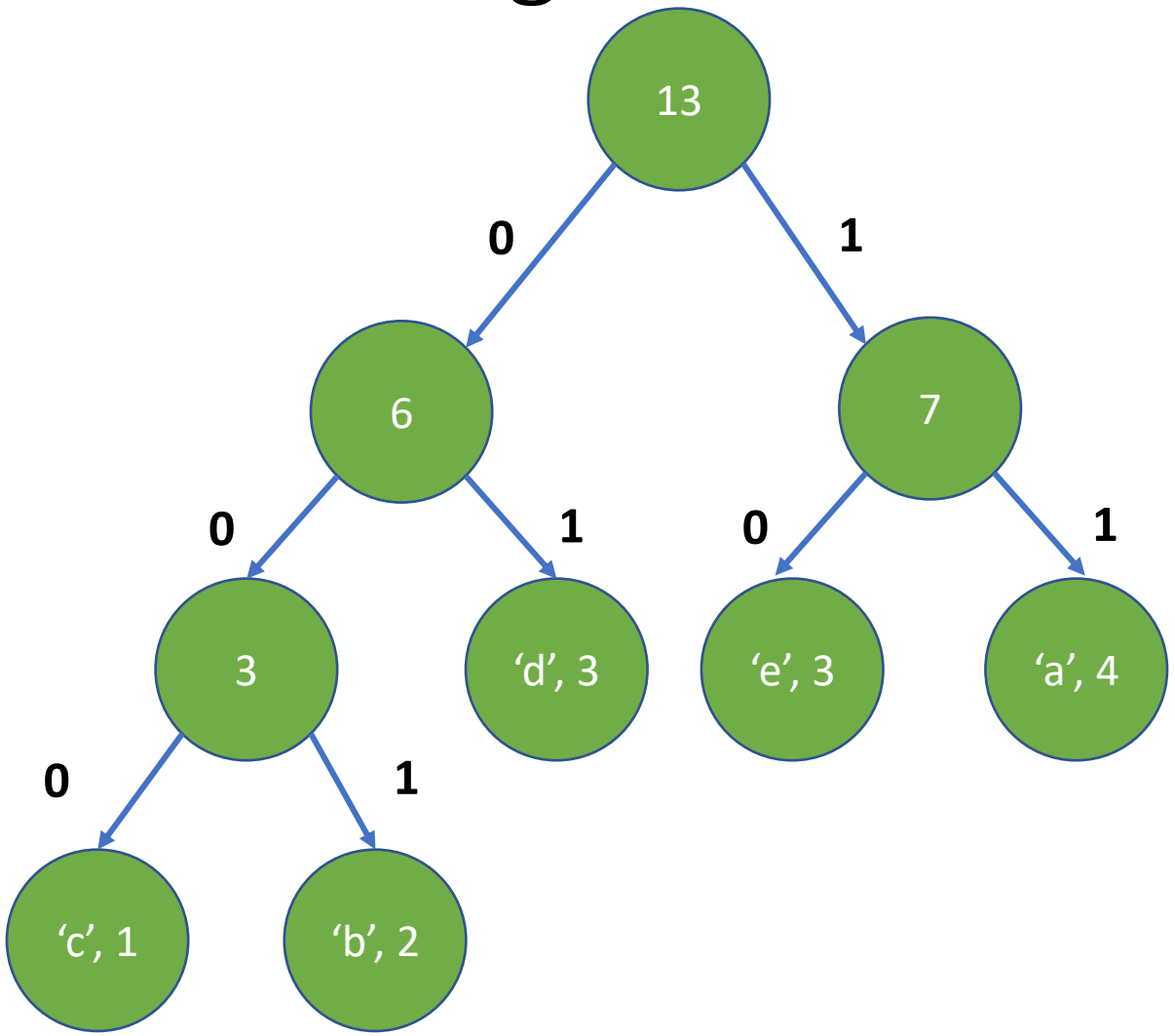
color edges:

left = 0, right = 1

$H(c) = path(c)$

- $H('c') = 000$
- $H('b') = 001$
- $H('d') = 01$
- $H('e') = 10$
- $H('a') = 11$

Huffman Coding



Claim: Char ↔ leaves

Pf: Induction on $|forest|$

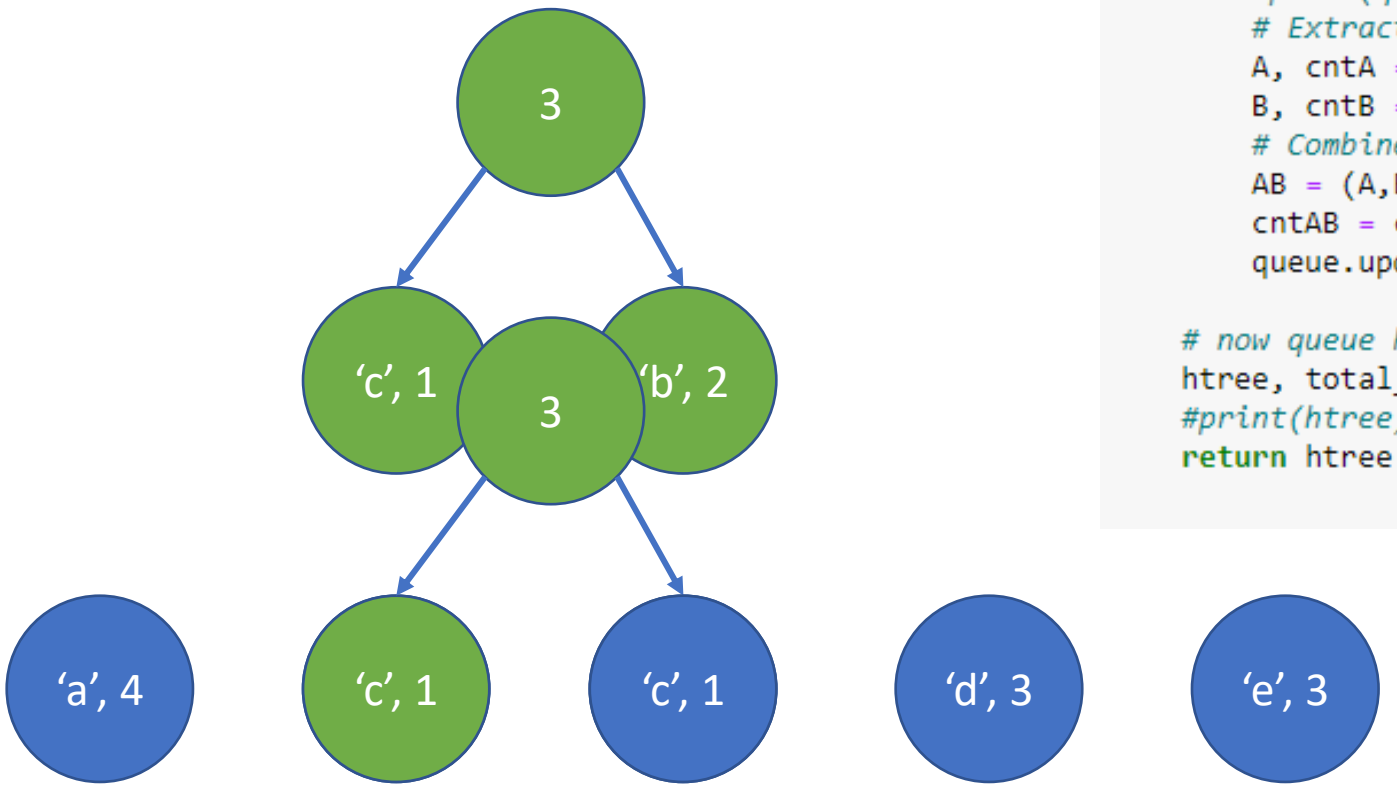
Claim: Code is prefix free

Pf: If not, $H(x) = H(y) + \cdots$ but then x is descendant of y…

Why is this important?

Forest by lists

{‘a’ : 4, ‘e’ : 3, ‘d’ : 3, ‘c’ : 1, ‘b’ : 2}




```
def build_huffman_tree(char_count_dict):
    """ Recieves dictionary with char:count entries
        Generates Huffman tree represented as a tuple (left, right) """
    queue = char_count_dict.copy() #keep input intact

    while len(queue) > 1:
        #print(queue)
        # Extract minimum twice
        A, cntA = extract_min_cnt(queue) # key, val with minimal val
        B, cntB = extract_min_cnt(queue) # next minimal
        # Combine two into one and insert
        AB = (A,B)
        cntAB = cntA + cntB
        queue.update({AB: cntAB})

    # now queue has only one pair {htree: total_cnt}
    htree, total_cnt = queue.popitem() # total count must be sum(queue)
    #print(htree)
    return htree   # a tuple representing the tree structure
```

```
def extract_min_cnt(d):
    min_node = min(d, key = lambda k: d[k])
    min_cnt = d[min_node]
    d.pop(min_node)
    return min_node, min_cnt
```

Cont.

{‘a’ : 4, ‘e’ : 3, ‘d’ : 3, (‘c’, ‘b’) : 3}

{‘a’ : 4, ‘e’ : 3} ‘d’ : 3, (‘c’, ‘b’) : 3

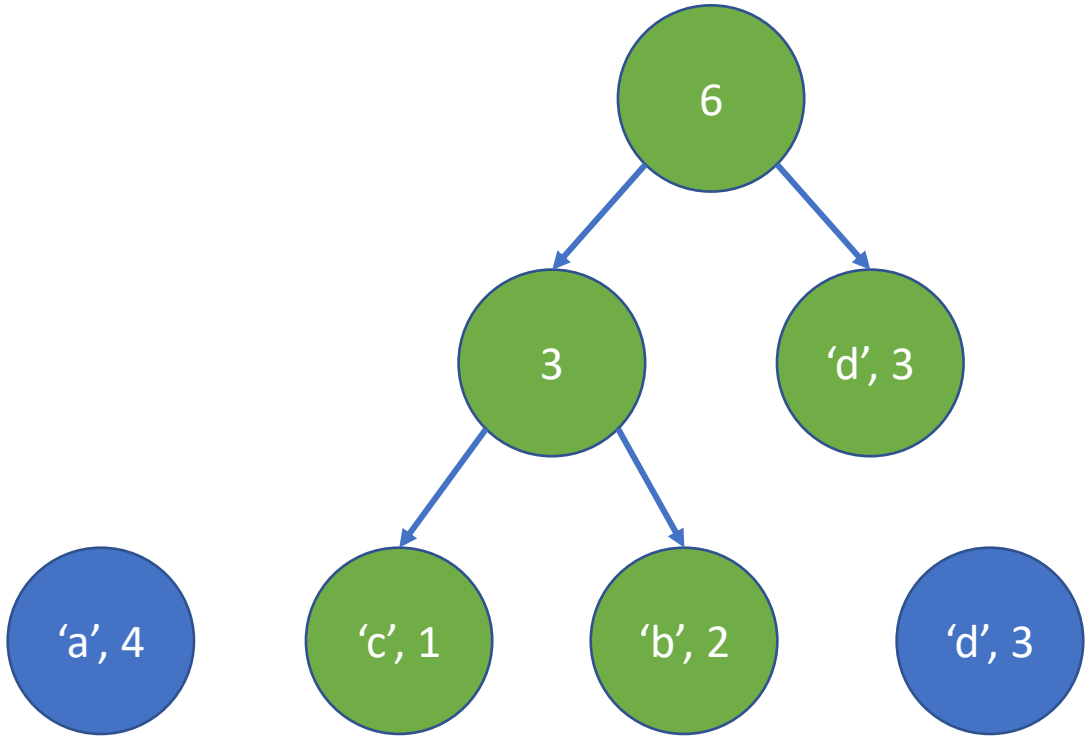
{‘a’ : 4, ‘e’ : 3} ((‘c’, ‘b’), ‘d’) : 6

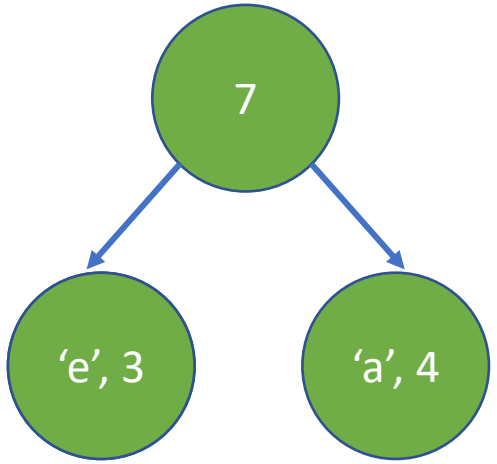
{‘a’ : 4, ‘e’ : 3, ((‘c’, ‘b’), ‘d’) : 6}

{((‘c’, ‘b’), ‘d’) : 6} ‘a’ : 4, ‘e’ : 3

{((‘c’, ‘b’), ‘d’) : 6} (‘e’, ‘a’) : 7

{((‘c’, ‘b’), ‘d’) : 6, (‘e’, ‘a’) : 7}



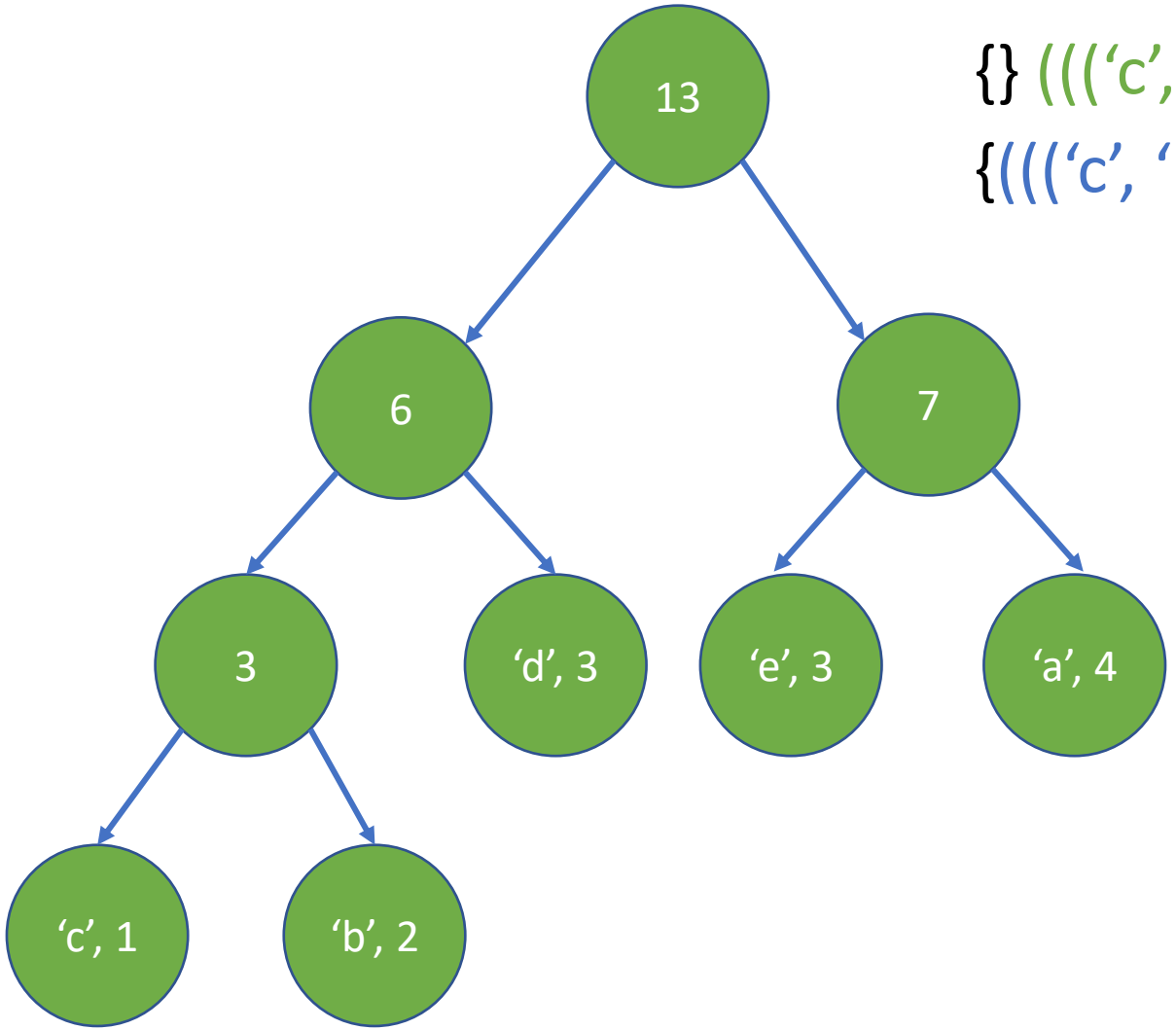


Cont.

{} (('c', 'b'), 'd') : 6, ('e', 'a') : 7

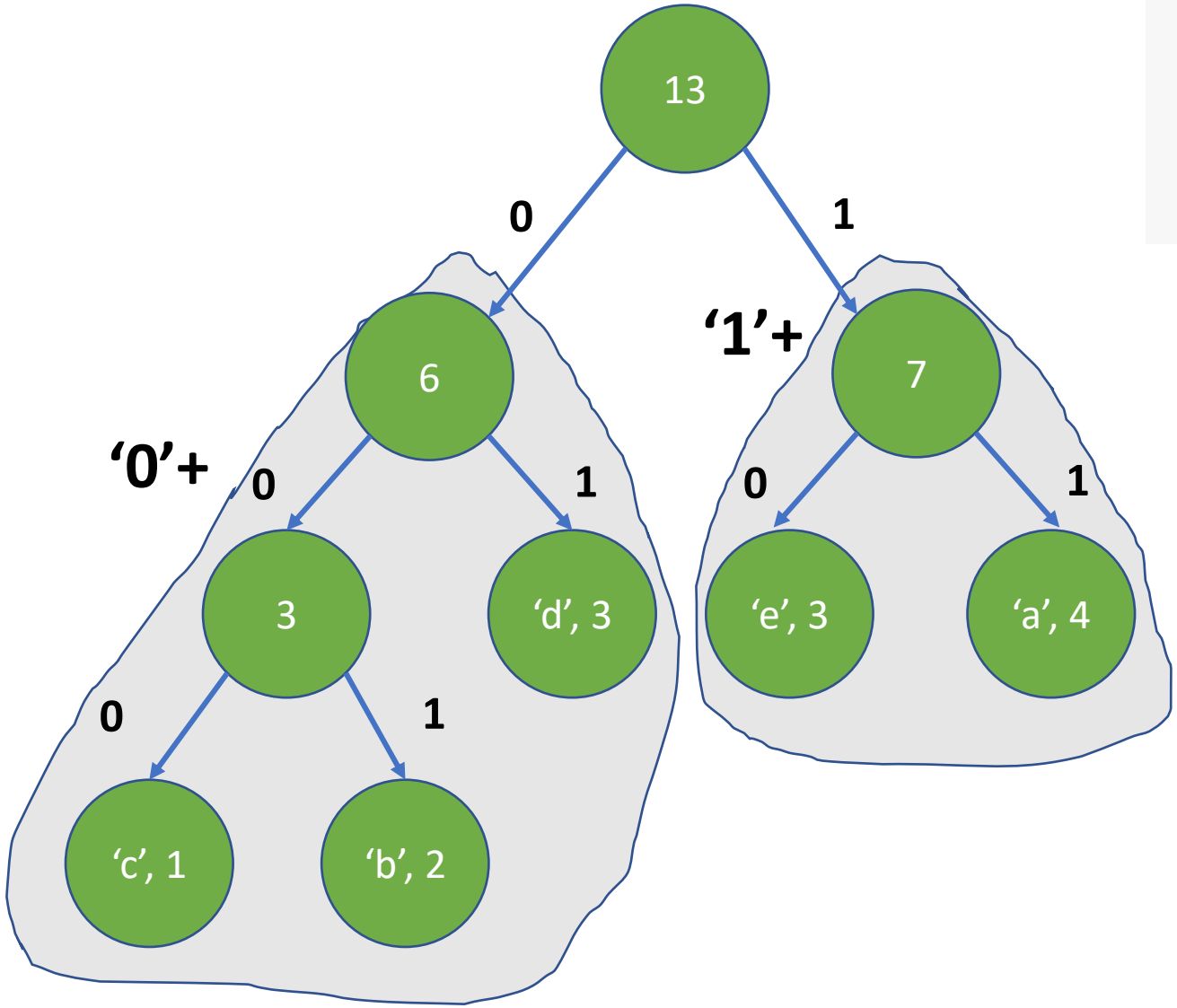
{} ((('c', 'b'), 'd'), ('e', 'a')) : 13

{((('c', 'b'), 'd'), ('e', 'a')) : 13}



Recursive traversal

corpus = ‘aaaabbcdddeee’




```
def generate_code(huff_tree, prefix=""):
    if isinstance(huff_tree, str): # a leaf
        return {huff_tree: prefix}
    else:
        lchild, rchild = huff_tree[0], huff_tree[1]
        codebook = {}

        codebook.update(generate_code(lchild, prefix+'0'))
        codebook.update(generate_code(rchild, prefix+'1'))
        return codebook
```

code((((‘c’, ‘b’), ‘d’), (‘e’, ‘a’)) , ‘’):

- C1 = code(((‘c’, ‘b’), ‘d’), ‘0’)
- C2 = code((‘e’, ‘a’), ‘1’)
- return C1 ∪ C2

Compressing and decompressing

Alice: H('babd') = 0011100101

Bob: 0011100101

- 0?
- 00?
- 001? **b**

Bob: 0011100101

- 1?
- 11? **a**

Bob: 00111100101…

- $H('c') = 000$
- $H('b') = 001$
- $H('d') = 01$
- $H('e') = 10$
- $H('a') = 11$

Compressing and decompressing

Alice: H('babd') = 0011100101

Why does it work?

- $f('a') = 1$
- $f('b') = 01$
- $f('c') = 011$

Alice: 'c' → 011

Bob: 011 → 'ba'

H is prefix free! f is not.

- $H('c') = 000$
- $H('b') = 001$
- $H('d') = 01$
- $H('e') = 10$
- $H('a') = 11$

PF and UD

Let $f: \Sigma \to \{0,1\}^*$ be a compression scheme

- We say f is Prefix Free if $f(x)$ is never a prefix of $f(y)$
- We say f is Uniquely Decodable if for any string $m \in \{0,1\}^*$ there is at most 1 message $\sigma_1 \dots \sigma_k \in \Sigma^k$ such that $f(\sigma_1 \dots \sigma_k) = m$

Easy: PF $\to$ UD

Other direction?

PF and UD

	PF	Not PF
UD	$a \to 011$ $b \to 10$	$a \to 0$ $b \to 01$
Not UD		$a \to 0$ $b \to 01$ $c \to 001$

How to decode?

msg = 001

ab? c?

Huffman - Time

Let:

- n be the length of the corpus
- m be the length of the message
- b be the bit length of $H(msg)$
- $|\Sigma| = O(1)$

Huffman - Time

- n be the length of the corpus
- m be the length of the message
- b be the bit length of $H(msg)$
- $|\Sigma| = O(1)$

```python
def char_count(corpus):
    """ Counts the number of each character in text.
        Returns a dictionary, with keys being the observed characters,
        values being the counts """
    d = {}
    for ch in corpus:
        if ch in d:
            d[ch] += 1
        else:
            d[ch] = 1
    return d
```

$O(n)$ on average

```python
def build_huffman_tree(char_count_dict):
    """ Recieves dictionary with char:count entries
        Generates Huffman tree represented as a tuple (left, right) """
    queue = char_count_dict.copy() #keep input intact

    while len(queue) > 1:
        #print(queue)
        # Extract minimum twice
        A, cntA = extract_min_cnt(queue) # key, val with minimal val
        B, cntB = extract_min_cnt(queue) # next minimal
        # Combine two into one and insert
        AB = (A,B)
        cntAB = cntA + cntB
        queue.update({AB: cntAB})

    # now queue has only one pair {htree: total_cnt}
    htree, total_cnt = queue.popitem() # total count must be sum(queue)
    #print(htree)
    return htree   # a tuple representing the tree structure
```

Queue has size $|\Sigma| = O(1)$
Each iteration pulls minimum from queue twice and combines it back into queue
Done when $|Queue| = 1$
In total: $O(1)$ time

Huffman - Time

```
66 def generate_hcode(htree, prefix=""):
67     """ Receives a Huffman tree (tuple) with embedded encoding,
68         and a prefix of encodings.
69         Returns a dictionary where characters are
70         keys and associated binary strings are values."""
71 
72     if isinstance(htree, str): # a leaf in the tree = a single char
73         return {htree: prefix}
74     else:
75         left_tree, right_tree = htree[0], htree[1]
76         hcode = {}
77 
78         hcode.update(generate_hcode(left_tree,  prefix+'0'))
79         hcode.update(generate_hcode(right_tree, prefix+'1'))
80         # oh, the beauty of recursion...
81 
82         return hcode
```

- n be the length of the corpus
- m be the length of the message
- b be the bit length of $H(msg)$
- $|\Sigma| = O(1)$

Tree has size $O(|\Sigma|)$

Each recursive call is on one node in the tree

Each node does $O(1)$ work

Total: $O(1)$ work

Huffman - Time

- n be the length of the corpus
- m be the length of the message
- b be the bit length of $H(msg)$
- $|\Sigma| = O(1)$

```
85 def compress(text, hcode):
86     """ compress text using encoding dictionary """
87     assert isinstance(text, str)
88     return "".join((hcode[ch] for ch in text))
```

$O(m)$ on average

```
91 def decompress(bits, htree):
92     """ get a bit string representing compressed text,
93         and the Huffman tree (tuple) used to compress it.
94         Return original text """
95     node = htree # htree = (htree_left, htree_right)
96     result = []
97 
98     for bit in bits:
99         node = node[int(bit)] #left or right?
100        if isinstance(node, str): # a leaf in the tree = a single char
101            result.append(node)
102            node = htree # restart, back to root
103 
104    return "".join(result)  # converts list of chars to a string
```

Each “step” in the tree takes $O(1)$ time and “produces” one output bit

In total: $O(b)$ time

Optimality

Among all codes C in which each character is encoded separately:

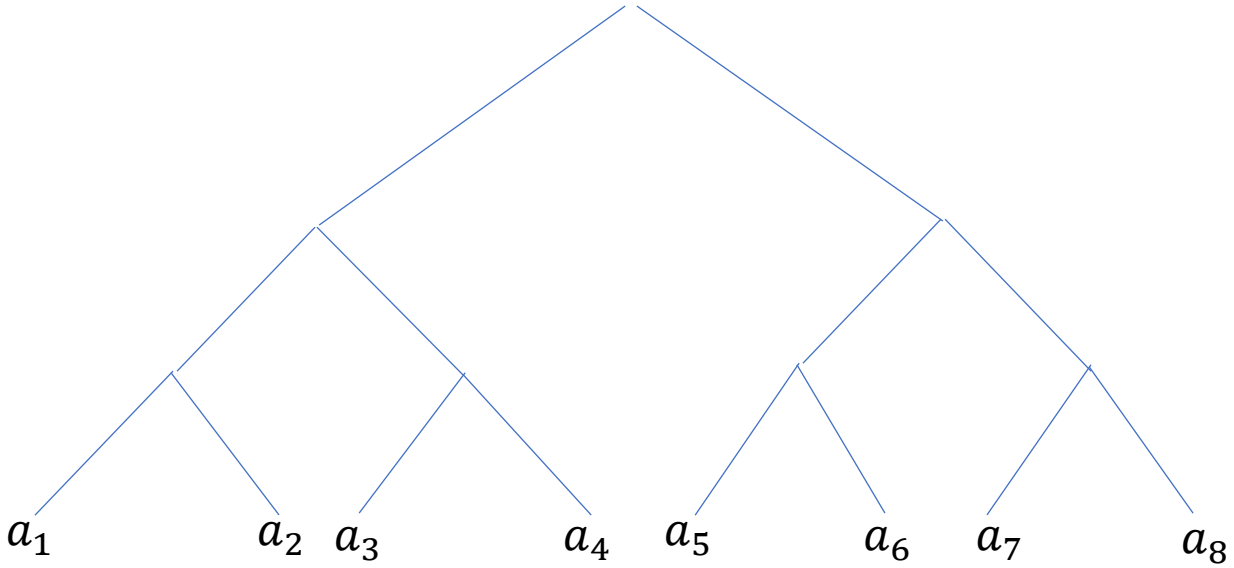
Huffman is optimal!

- Optimal = minimizes $\sum_{a_i \in \Sigma} (|C(a_i)| \cdot w_i)$
- Where $C(a_i)$ is the length of the encoding of a_i and w_i is the number of appearances of a_i in the corpus
- Interesting when text distributes as corpus

Example 1

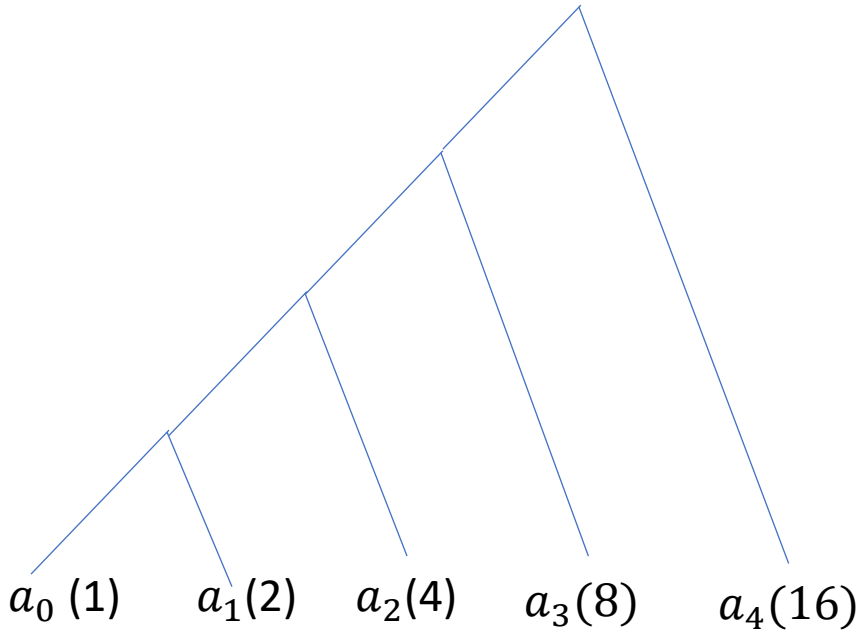
- Draw Huffman tree for a corpus with $|\Sigma| = 2^n$, where each character appears once

Shortest encoding = longest encoding = n



Example 2

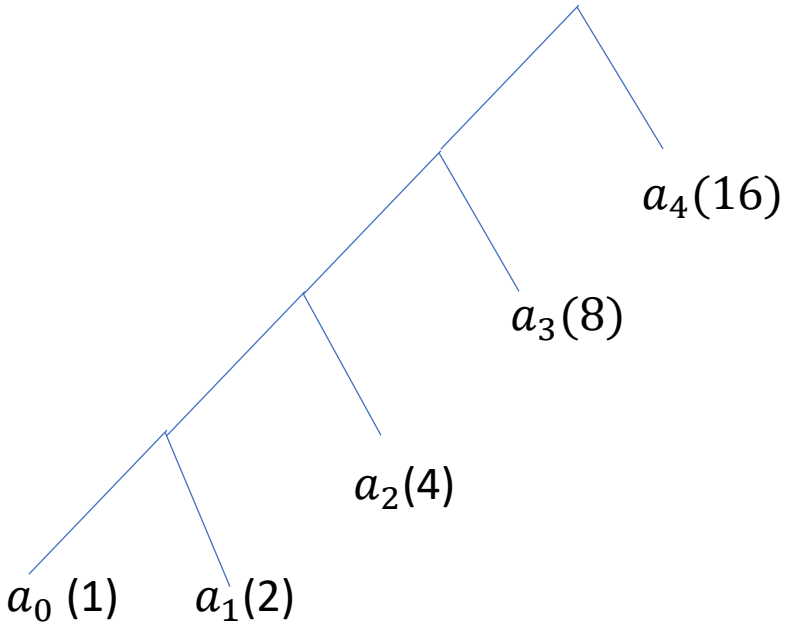
- Draw Huffman tree for a corpus with $|\Sigma| = n$, where the i-th character appears 2^i times $(0 \leq i \leq n-1)$



Example 2

- Draw Huffman tree for a corpus with $|\Sigma| = n$, where the i-th character appears 2^i times $(0 \leq i \leq n-1)$

Shortest encoding = 1
longest encoding = n-1

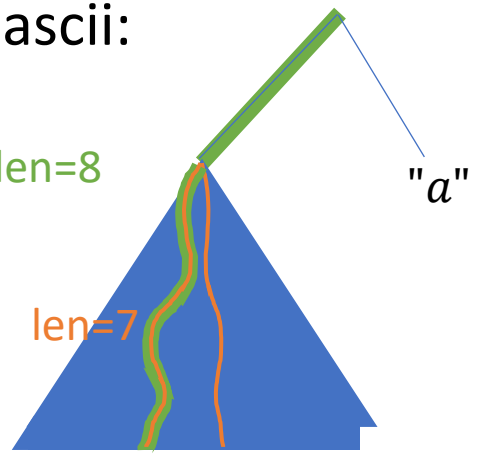


Compression ratio

```
def compression_ratio(text, corpus):
    d = generate_hcode(build_huffman_tree(char_count(corpus)))
    len_compress = len(compress(text, d))
    len_ascii = len(ascii2bit_stream(text)) #len(text)*7
    return len_compress/len_ascii
```

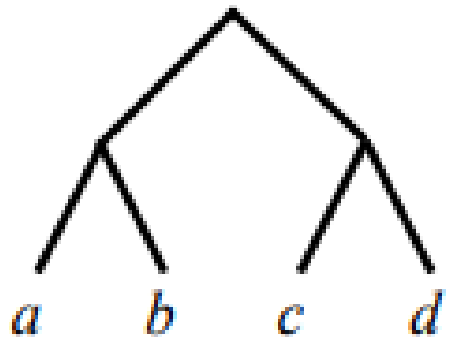
- compression_ratio(“ab”, “ab”) → 1/7
  - len_compress = 2
  - len_ascii = 14
- compression_ratio(“hello”, “a”*100 + ascii) → 8/7
  - Huffman tree for corpus “a”*100 + ascii:

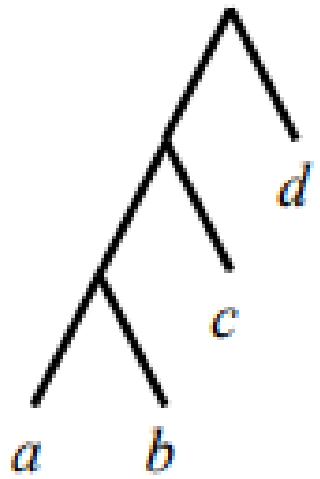
For most characters: len=8



Exam question: Alternative Trees

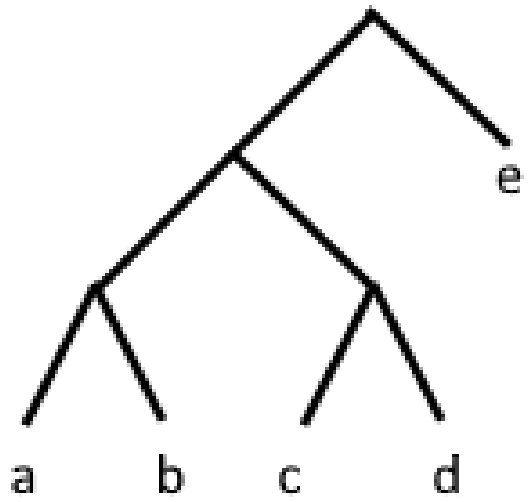
- Two Huffman trees are alternative if they can be generated from the same corpus but have different multiset of lengths:
- Example: corpus = ‘abcdd’
- Lengths: (2, 2, 2, 2) and (1, 2, 3, 3)

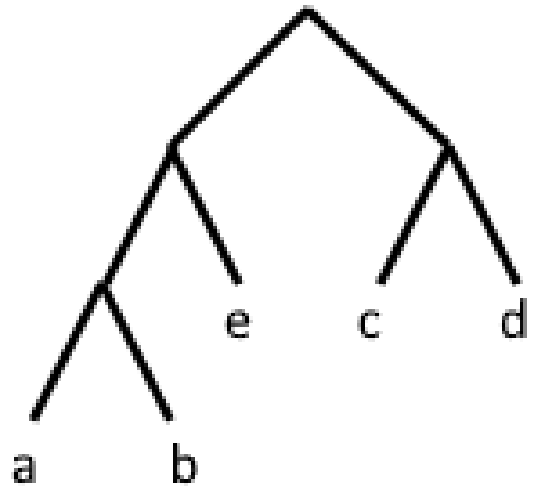




Alternative Trees

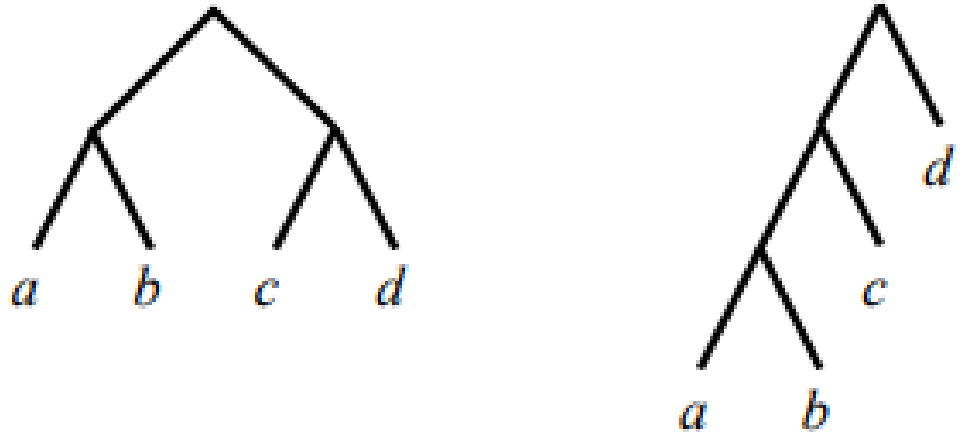
- Show two alternative trees for corpus = ‘abcdee’





Alternative Trees

- Prove/disprove: if corpus frequencies are unique, no alt. trees
- Key observation: weights are unique initially, but not necessarily throughout the algorithm execution!

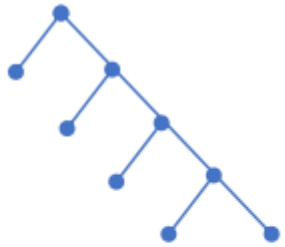


corpus = ‘aa,bbb,cccc,ddddd’

Exam Question (2019aa)

5. נתון קורפוס שמכיל את התווים a,b,c,d,e. שכיחויות התווים בקורפוס הינן: w_a , w_b , w_c , w_d , w_e, בהתאמה. כל השכיחויות שלמות וגדולות ממש מ-0. על סמך השכיחויות הללו, בונים עץ האפמן ע"י האלגוריתם שלמדנו.
בכל תת-סעיף נתון מבנה של עץ ועליכם לסמן את התנאים על ערכי השכיחויות **שבהכרח** יגרמו לבניית עץ בעל מבנה כזה ע"י אלגוריתם האפמן (עד כדי הבדלים של שמאל ימין), או לסמן שאין תנאי כזה מבין האפשרויות. במקרה כזה (ורק במקרה כזה) עליכם גם להסביר, ולכלול בהסבר אחד משניים: ציון תנאים אחרים עבורם בהכרח מתקבל עץ כזה, או נימוק מדוע זה לא ייתכן. שימו לב שיתכנו מספר תשובות נכונות עבור כל עץ ועליכם לסמן את כולן.

Exam Question (2019aa)



א.

a. $w_a > w_b > w_c > w_d > w_e$

b. $w_a = \frac{1}{2}w_b = \frac{1}{4}w_c = \frac{1}{8}w_d = \frac{1}{16}w_e$

c. $w_a = w_b = w_c\,, \quad w_d > 3w_a, \quad w_e > 2w_d$

d. $w_a = \frac{1}{2}w_b = \frac{1}{3}w_c = \frac{1}{4}w_d = \frac{1}{5}w_e$

e. אף אחד מהנ"ל לא מתאים עבור עץ זה

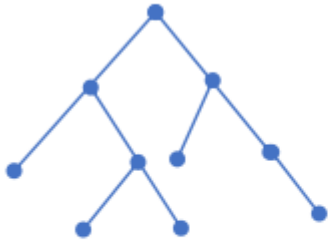
נימוק, אם סימנתם את תשובה e: _____

_____

_____

- א', דוגמא נגדית: משקלים 1, 2, 3, 4, 5
- ב', נכון: ראינו מקודם
- ג', נכון: סדר החיבור הוא תמיד $a \to b \to c \to d \to e$
  כיוון שמשקל d גדול מ-a, b, c ביחד ומשקל e גדול מפעמיים d ולכן גדול מ-a, b, c, d ביחד
- ד', דוגמא נגדית: כמו מקודם

Exam Question (2019aa)



ב.

a. $w_a = \frac{1}{2}w_b = \frac{1}{4}w_c\,, \quad w_d = \frac{1}{16}w_e$

b. $w_a > w_b > 2w_c > 4w_d > 8w_e$

c. $w_a = w_b = w_c\,, \quad w_d = w_e > 10w_a$

d. $w_a = w_b = w_c\,, \quad w_d = w_e > 3w_a$

e. אף אחד מהנ"ל לא מתאים עבור עץ זה

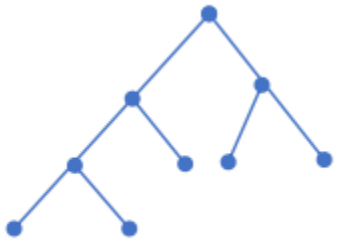
נימוק, אם סימנתם את תשובה e: _____

_____

_____

- בעץ האפמן לכל צומת יש 0 או 2 בנים
- בפרט, אין צומת עם בן אחד כמו בעץ שבתמונה

Exam Question (2019aa)



ג. $w_a = \frac{1}{2}w_b < \frac{1}{4}w_c = \frac{1}{4}w_d = \frac{1}{5}w_e$ .a

$w_a = \frac{1}{10}w_b = \frac{1}{20}w_c = \frac{1}{25}w_d = \frac{1}{30}w_e$ .b

$w_a > w_b > w_c$, $w_d < w_e$ .c

$w_a = w_b = w_c < w_d = w_e$ .d

אף אחד מהנ"ל לא מתאים עבור עץ זה .e

נימוק, אם סימנתם את תשובה e: _____

- א', דוגמא נגדית: משקלים 1, 2, 80, 80, 100
- ב', נכון: $w_a + w_b < w_c$ וגם $w_a + w_b + w_c > w_d, w_e$
- ג', דוגמא נגדית: 1, 2, 4, 8, 16
- ד', דוגמא נגדית: 1, 1, 1, 10, 10

Lempel-Ziv Compression

LZ Compression

- Main idea: human text is repetitive
- text =‘abcxyabc’ → ‘abcxy’, go back 5 chars and repeat 3 chars
- Succinct: [‘a’, ‘b’, ‘c’, ‘x’, ‘y’, [5, 3]] ← Intermediate representation
- Binary encoding:
  - $c \rightarrow bin(ord(c))$
  - $[back, rep] \rightarrow bin(back), bin(rep)$
- From binary to intermediate rep.
  - 100110101 ?
    - chr(100) + chr(110) + chr(101)
    - chr(100) + chr(110101)
    - chr(100110) + chr(10) + chr(1)
    - What about repetitions?

Problem

LZ Compression - Solution

- Step 1: fix length
  - $char \rightarrow ascii(char)$ in exactly 7 bits
  - $[back, rep] \rightarrow 000 \ldots bin(back) + 000 \ldots\ bin(rep)$
    - Requires log |max back| + log |max rep| bits
- Step 2: add indicator to distinguish between char and repetition
  - $char \rightarrow \mathbf{0} + ascii(char)$
    - Requires 8 bits
  - $[back, rep] \rightarrow \mathbf{1} + 000 \ldots bin(back) + 000 \ldots\ bin(rep)$
    - Requires 1 + log |max back| + log |max rep| bits
- If $|\max back| = |\max rep| = n$, each rep. takes $O(\log n)$ bits
- Choose $|\max back|, |\max rep| = O(1)$. Why?

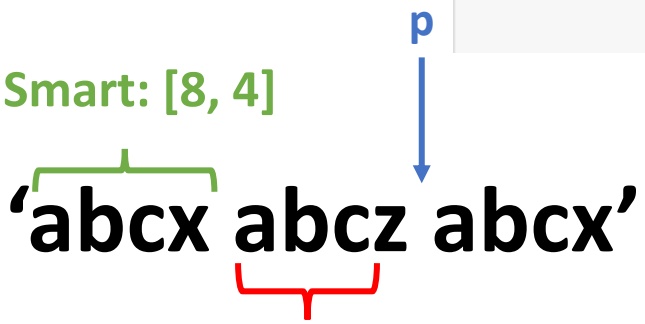
LZ Compression

- We chose $\log|\max back| = 12, \log|\max rep| = 5$
- A repetition of 1 char is bad ($8 < 18$)
- What about 2 chars?
- And 3? 4? …?

LZ algorithm - compressing

```python
def LZW_compress(text, W=2**12-1, L=2**5-1):
    """ LZW compression of an ascii text. Produces
        a list comprising of either ascii characters
        or pairs [m,k] where 1<=m<=W is an offset and
        3<=k<=L is a match """
    intermediate = []
    n = len(text)
    p = 0
    while p<n:
        m,k = maxmatch(text, p, W, L)
        if k<=2:
            intermediate.append(text[p]) #  a single char
            p+=1
        else: #k>=3
            intermediate.append([m,k]) # compressing 3+ characters
            p+=k
    return intermediate  # a list composed of chars and pairs
```

```python
def maxmatch(T, p, W=2**12-1, L=2**5-1):
    """ Finds a maximum match of length k<=L within a
        W long window, T[p:p+k] = T[p-m:p-m+k].
        Returns m (offset) and k (match length) """
    assert isinstance(T,str)
    n = len(T)
    m = 0
    k = 0
    for offset in range(1, 1+min(p, W)):
        match_len = 0
        j = p-offset #starting point of earlier repetition
        while match_len < min(n-p, L) and T[j+match_len] == T[p+match_len]:
            match_len+=1  # at this point, T[j:j+match_len]==T[p:p+match_len]
        if match_len > k:
            k = match_len
            m = offset
    return m, k # returned offset is smallest one (closest to p)
                            # among all max matches
```



Not good enough: [4, 3], ‘x’

LZ algorithm – binary representation


```python
def inter_to_bin(intermediate, W=2**12-1, L=2**5-1):
    """ converts intermediate format compressed list
        to a string of bits"""
    W_width = math.floor(math.log(W,2)) + 1
    L_width = math.floor(math.log(L,2)) + 1
    bits = []
    for elem in intermediate:
        if type(elem) == str:
            bits.append("0") #to note a single char ahead
            bits.append((bin(ord(elem))[2:]).zfill(7))
        else: #elem is a list [m,k]
            bits.append("1") #to note a repetition ahead
            m,k = elem
            bits.append((bin(m)[2:]).zfill(W_width))
            bits.append((bin(k)[2:]).zfill(L_width))
    return "".join(ch for ch in bits)
```

```python
def bin_to_inter(bits, W=2**12-1, L=2**5-1):
    """ converts a compressed string of bits
       to intermediate compressed format """
    W_width = math.floor(math.log(W,2)) + 1
    L_width = math.floor(math.log(L,2)) + 1
    inter = []
    n = len(bits)
    p = 0
    while p<n:
        if bits[p] == "0":  # single ascii char ahead (next 7 bits)
            p+=1
            char = chr(int(bits[p:p+7], 2))
            inter.append(char)
            p+=7
        elif bits[p] == "1":  # repeat [m,k] ahead
            p+=1
            m = int(bits[p:p+W_width],2)
            p+=W_width
            k = int(bits[p:p+L_width],2)
            p+=L_width
            inter.append([m,k])
    return inter
```

More examples

- “abcab”→[‘a’, ’b’, ’c’, ’a’, ’b’]
- “abcabcdabc”→[‘a’, ‘b’, ‘c’, [3,3], ‘d’, [4, 3]]
- “a”*10 → [‘a’, [1,9]]
- “a”*40 →[‘a’, [1, 31], [1, 8]]

LZ algorithm - decompressing


```
def LZW_decompress(intermediate):
    """ LZW decompression from intermediate format to ascii text"""
    text_lst = []
    for i in range(len(intermediate)):
        if type(intermediate[i]) == str:  # char, as opposed to a pair [m,k]
            text_lst.append(intermediate[i])
        else:
            m,k = intermediate[i]
            for j in range(k): # append k times to result
                text_lst.append(text_lst[-m])
                # a fixed offset m "to the left", as result itself grows
    return "".join(text_lst)
```

Compression ratio

```
def lz_ratio(text):
    inter1, bits, inter2, text2 = process(text)
    return len(bits)/(len(text)*7)
```

- lz_ratio(“hello”) → 8/7
  - Intermediate rep. [‘h’, ‘e’, ‘l’, ’l’, ‘o’]
  - len_compress = $5 \cdot 8 + 0 \cdot 18$
  - len_ascii = $5 \cdot 7$
- Lz_ratio(“hello”*2) → 0.82857…
  - Intermediate rep. [‘h’, ‘e’, ‘l’, ’l’, ‘o’, [5,5]]
  - len_compress = $5 \cdot 8 + 1 \cdot 18$
  - len_ascii = $10 \cdot 7$

LZ performance

- Empirically, LZ performs very well over human text
- In fact, still in use (WinZip, 7zip)

Exam questions: 2020bb

- For a string S let $L_{LZ}(S)$ denote the bit-length of the compressed string $LZ(S)$
- Prove/disprove: for any string S, $L_{LZ}(S) == L_{LZ}(s[::-1])$
- False! S = “aaaabaaac”
- Key observation: the repetition “aaa” can be compressed after reading “aaaa” but not before it.

Exam questions: 2020bb

- For a string S let $L_{LZ}(S)$ denote the bit-length of the compressed string $LZ(S)$
- Prove/disprove: for any string S, $L_{LZ}(S) < L_{LZ}(S + "a")$
- False! S = “aaaa”
- Key observation: the last “a” can be “absorbed” in a repetition

Exam questions: 2020bb

- For a string S let $L_{LZ}(S)$ denote the bit-length of the compressed string $LZ(S)$
- Prove/disprove: for any strings S, R $L_{LZ}(S+R) = L_{LZ}(S) + L_{LZ}(R)$
- False! S = R = “aa”
- Key observation: S+R can contain a new repetition

Exam questions: 2020bb

- Elhanan implemented a new “lossy” version of LZ
- The new version has a new function “max_almost_match” which finds the longest match in which **at most 1 char** differs from the text
- Example: max_almost_match(“abcdeaxcdz”, 5, W, max_length) will return the almost match “abcd” of length 4
- In the LZ code, Amir runs both max_match and max_almost_match. If max_almost_match returns a repetition of length ≥ 10, we use it, otherwise, we use the repetition from max_match

Exam questions: 2020bb

```
def LZW_compress(text, W =2**12 -1, max_length=2**5 -1):

    result = []
    n = len(text)
    p = 0

    while p<n:

        m,k = max_match(text, p, W, max_length)
        m2,k2 = max_almost_match(text, p, W, max_length)
        if k2>=10:
            m,k = m2,k2

        if k < 3: #no match or match too short to compress
            result.append(text[p]) # a single char
            p +=1
        else: # 3 or more chars in match
            result.append([m,k])
            p+=k

    return result
```

Exam questions: 2020bb

- Prove/disprove: in any iteration of the while loop, $k2 \geq k$
- True.
- By definition, a maximal match is also an almost maximal match.
- Thus, the set of matches found in max_match is a subset of the set of matches found in max_almost_match
- The max length element in the larger set cannot decrease

Exam questions: 2020bb

- Prove/disprove: Since each repetition has at most 1 mismatching character, for any string S, after compressing and decompressing, the recovered string S' differs from S in at most 10% of locations.
- False!
- Key observation: we can pay dearly for a single mistake
- Example: S='123456789'+'x'+ '123456789'+'y'+'y'*1000
- First ten chars have no repetition, next ten chars have an almost match, and the 'x' instead of 'y' is then copied 1000 times.