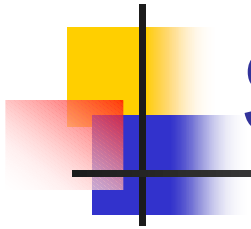
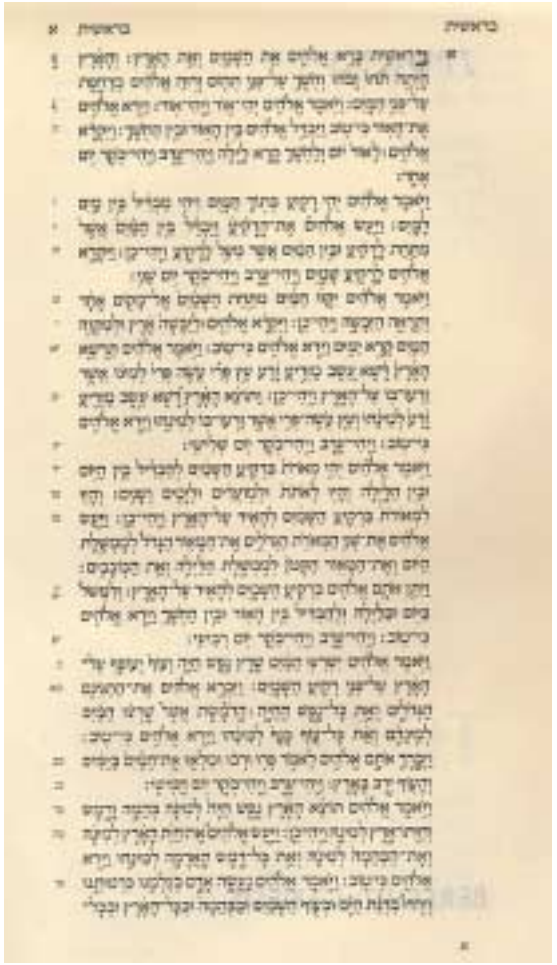


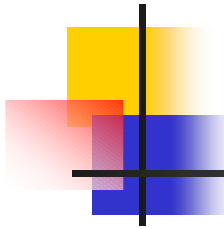
Static Huffman

- A known tree is used for compressing a file.
  - A different tree can be used for each type of file. For example a different tree for an English text and a different tree for a Hebrew text.
- Two passes on file. One pass for building the tree and one for compression.

Wrong probabilities

- What is different in this text?

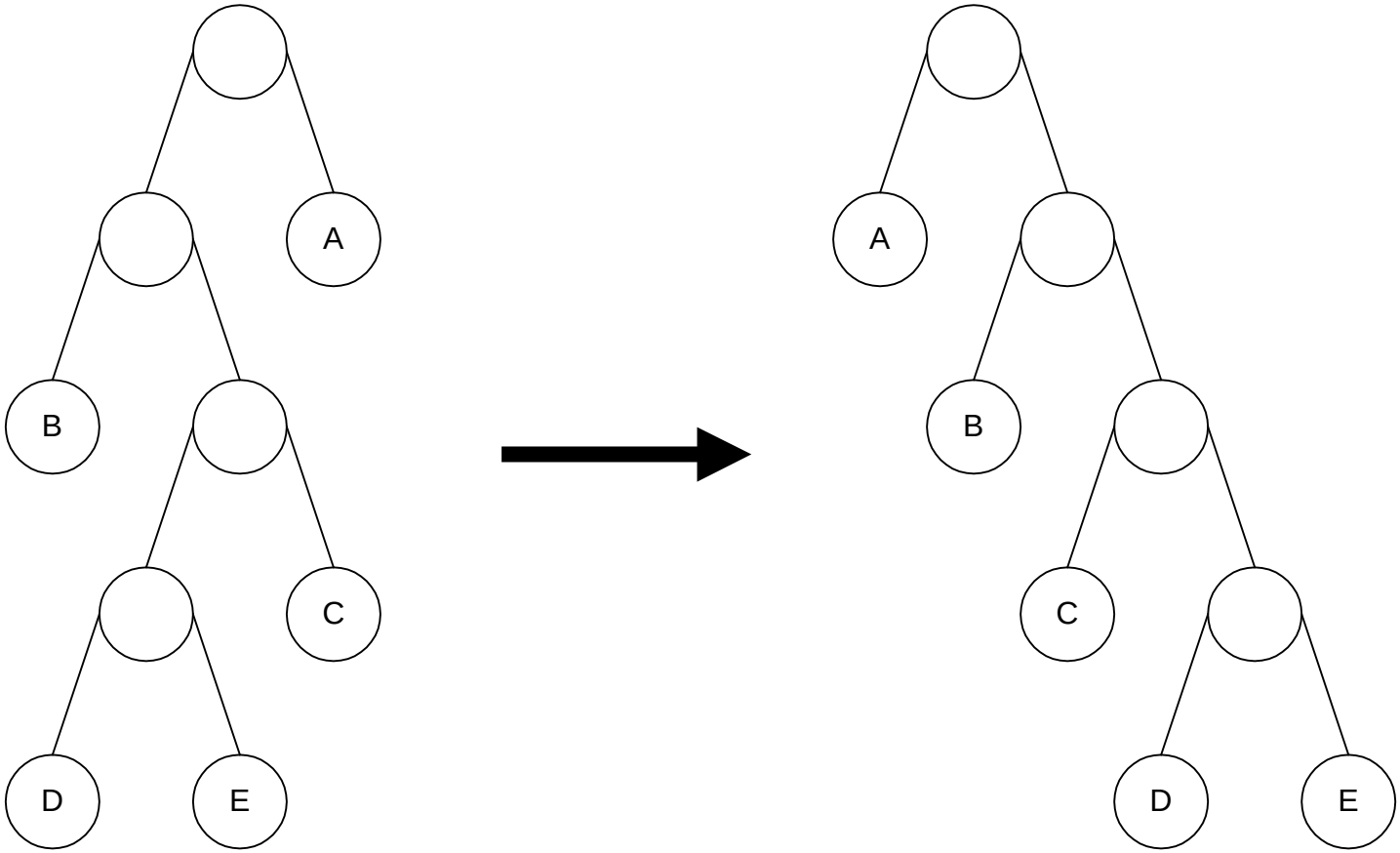




Adaptive Huffman

- x_t is encoded with tree of $x_1,\ldots,x_{t-1}$.
- Tree is changed during compression.
- Only one pass.
- No need to transmit the tree.
  - Two possibilities:
    - At the beginning assume each item appeared once.
      - At the beginning probabilities are wrong but after a large amount of data the error is negligible.
    - When a new character appears send an escape character before it.

Canonical Huffman Trees



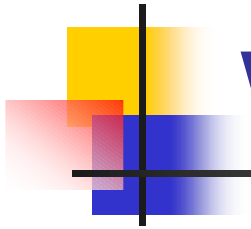
Algorithm for canonical trees

- Find a Huffman tree with lengths $L_1,\ldots,L_n$ for the items.
- Sort the items according to their lengths.
- Assign to each item the first L_i bits after the binary point of $\sum_{j=1}^{i-1} 2^{-L_j}$

Example of a canonical tree

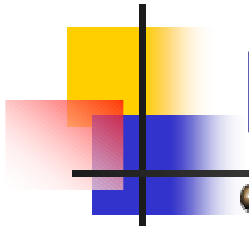
- Suppose the lengths are:
  - A-3, B-2, C-4, D-2, E-3, F-3, G-4
- The sorted list is:
  - B-2, D-2, A-3, E-3, F-3, C-4, G-4

Item	L_i	2^{-L_i}	$\sum_{j=1}^{i-1} 2^{-L_j}$
B	2	0.01	0.**00**000
D	2	0.01	0.**01**000
A	3	0.001	0.**100**00
E	3	0.001	0.**101**00
F	3	0.001	0.**110**00
C	4	0.0001	0.**1110**0
G	4	0.0001	0.**1111**0



Why canonical trees?

- A canonical tree can be transferred easily.
  - Send number of items for every length.
  - Send order of items.
- Canonical codes synchronize faster after errors.
- Canonical codes can be decoded faster.

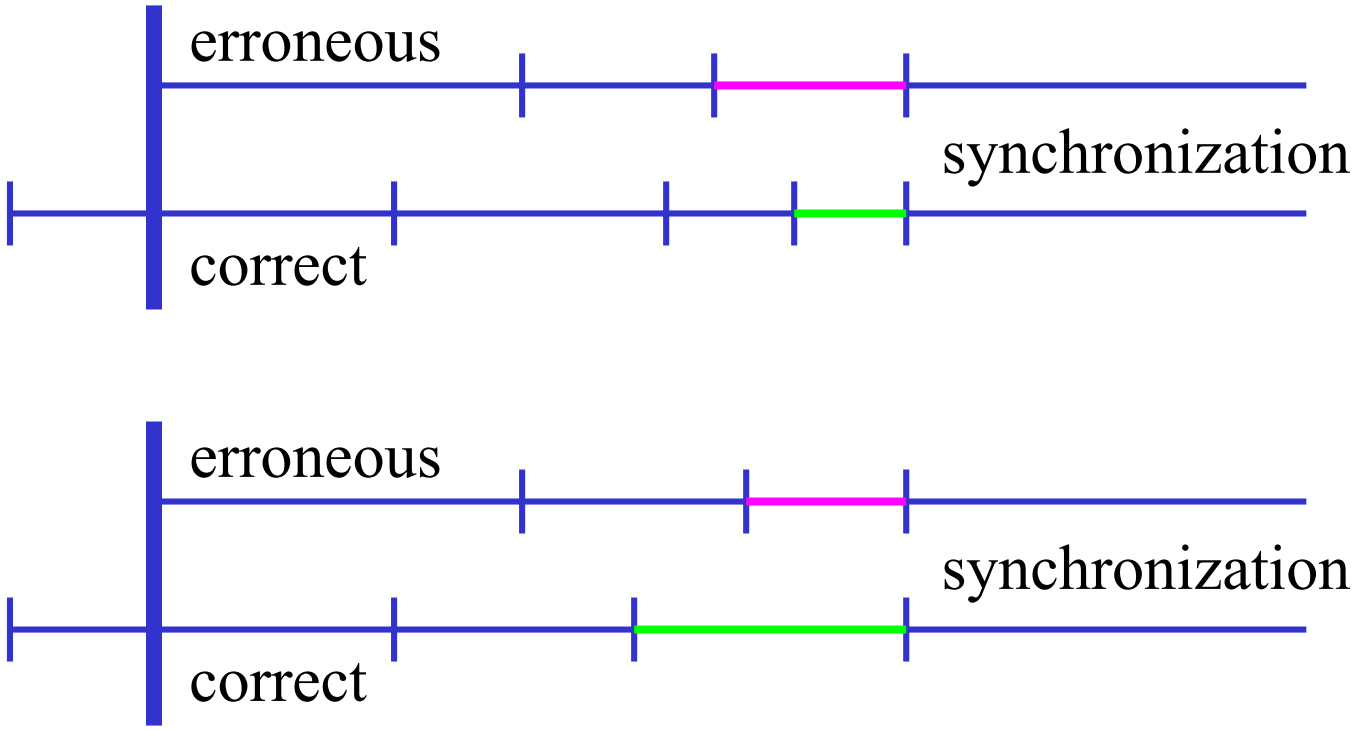


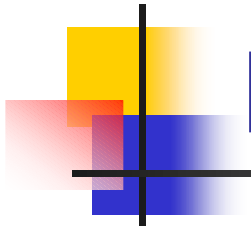
Errors in Huffman coded files

- In the beginning God created the heaven and the earth. And the earth was without form, and void; and darkness was upon the face of the deep. And the Spirit of God moved upon the face of the waters. And God said, Let there be light: and there was light. And God saw the light, that it was good: and God divided the light from the darkness. And God called the light Day, and the darkness he called Night. And the evening and the morning were the first day. And God said, Let there be a firmament in the midst of the waters, and let it divide the waters from the waters...
- What will happen if the compressed file is read from arbitrary points?
- ***: ac*** darkness was upon the face
- ***csoaraters.*** And God said, Let there be light
- ***d lnra***that it was good: and God divided
- ***.aauya dy***, and the darkness he called Night
- ***c y.*** And God said, Let there be a firmament in the midst

Synchronization after error

- If the code is not an affix code:





Definitions

- Let $P_1,\ldots,P_n$ be the probabilities of the items (The leaves of the Huffman tree).
- Let $L_1,..,L_n$ be the length of codewords.
- Let X be the set of internal nodes in the Huffman tree.
- $\forall x \in X$ let I_x be the set of leaves in the sub-tree rooted by x.

Formulas

- Average codeword's length is $W = \sum_{i=1}^{n} P_i L_i$
- The Probability that an arbitrary point in the file will be in node x is: $P(x) = \frac{\sum_{y \in I_x} P_y}{W}$
- $\forall x \in X$ and $\forall y \in I_x$ define:
- $Q(x,y) = \begin{cases} 1 & \text{if the path from x to y corresponds to a sequence of one or more codewords in the code} \\ 0 & \text{otherwise} \end{cases}$

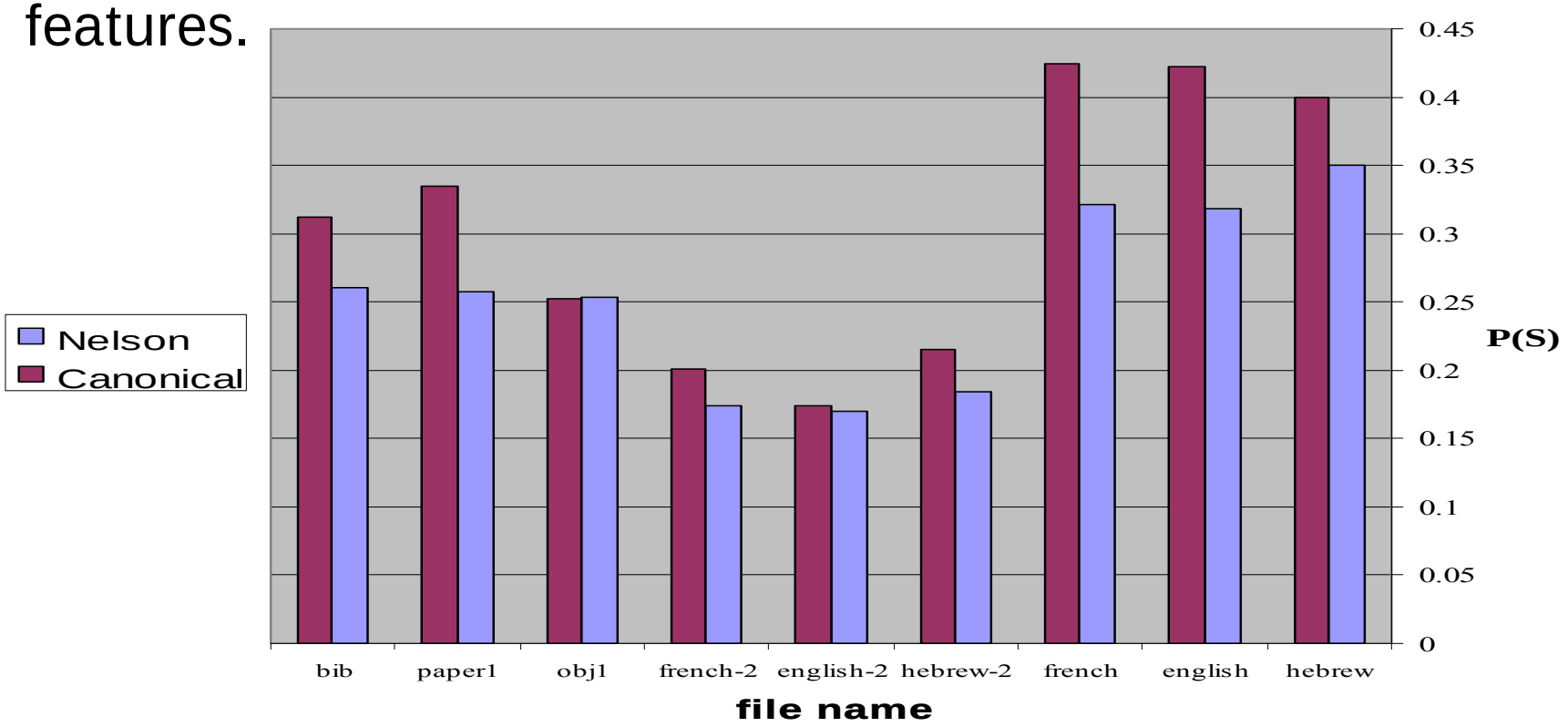
Synchronization's probability

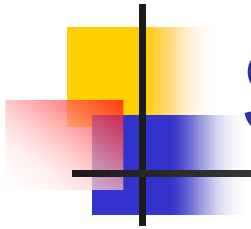
- Let S denote the event that the synchronization point is at the end of the codeword including $x \in X$.

$$P(S) = \sum_{x \in X} \frac{\sum_{y \in I_x} P_y Q(x,y)}{\sum_{y \in I_x} P_y} \cdot P(x) = \frac{\sum_{x \in X} \sum_{y \in I_x} P_y Q(x,y)}{W}$$

Probability for a canonical tree

- Nelson's trees are the trees described in 2.2 with some other features.



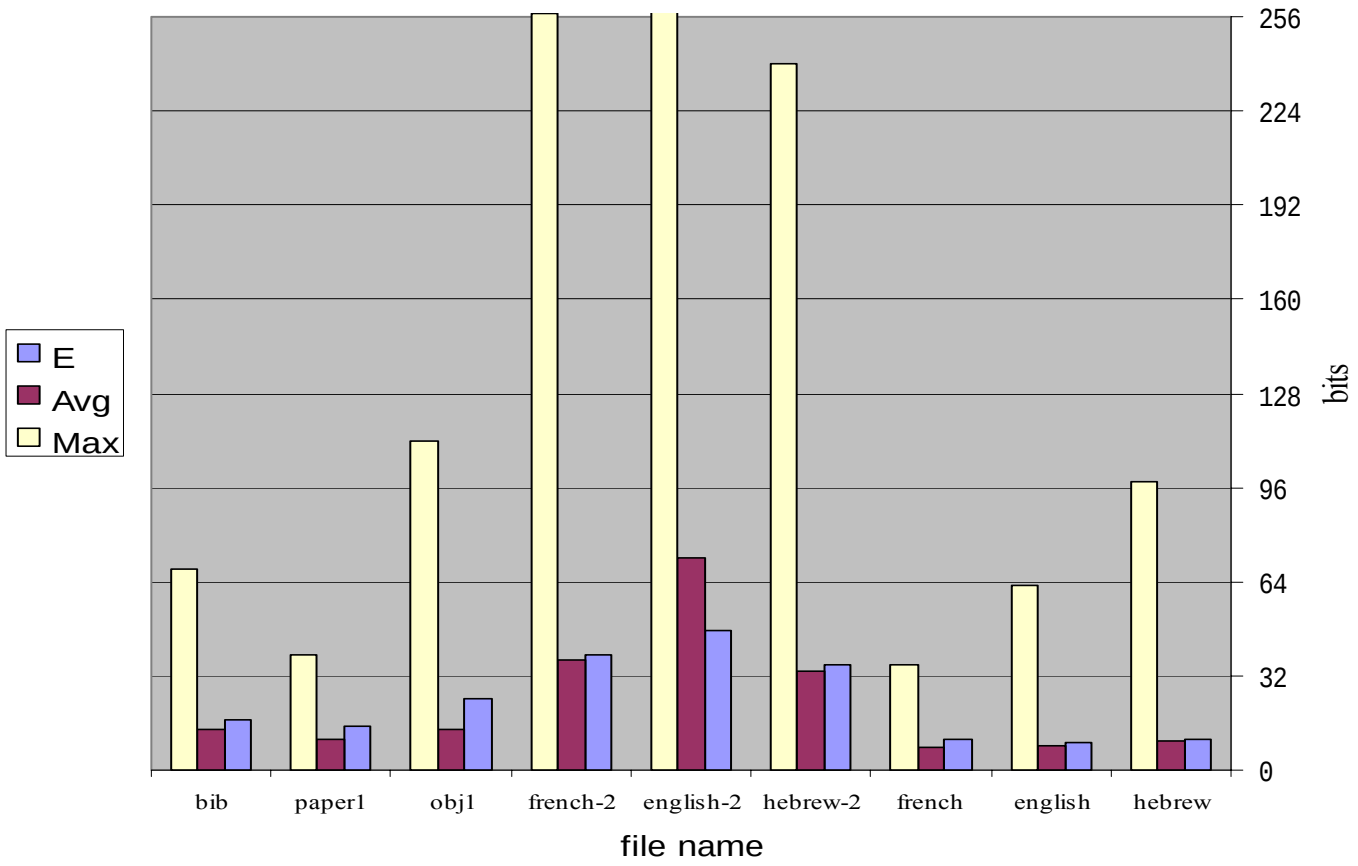


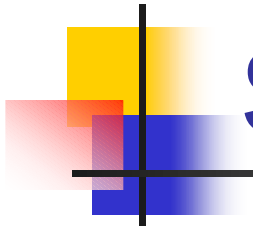
Synchronization

- canonical trees synchronize better since every sub-tree of a canonical tree is a canonical tree itself.
- Expected number of bits until synchronization is:

$$E = \frac{W}{P(S)}$$

Expected synchronization





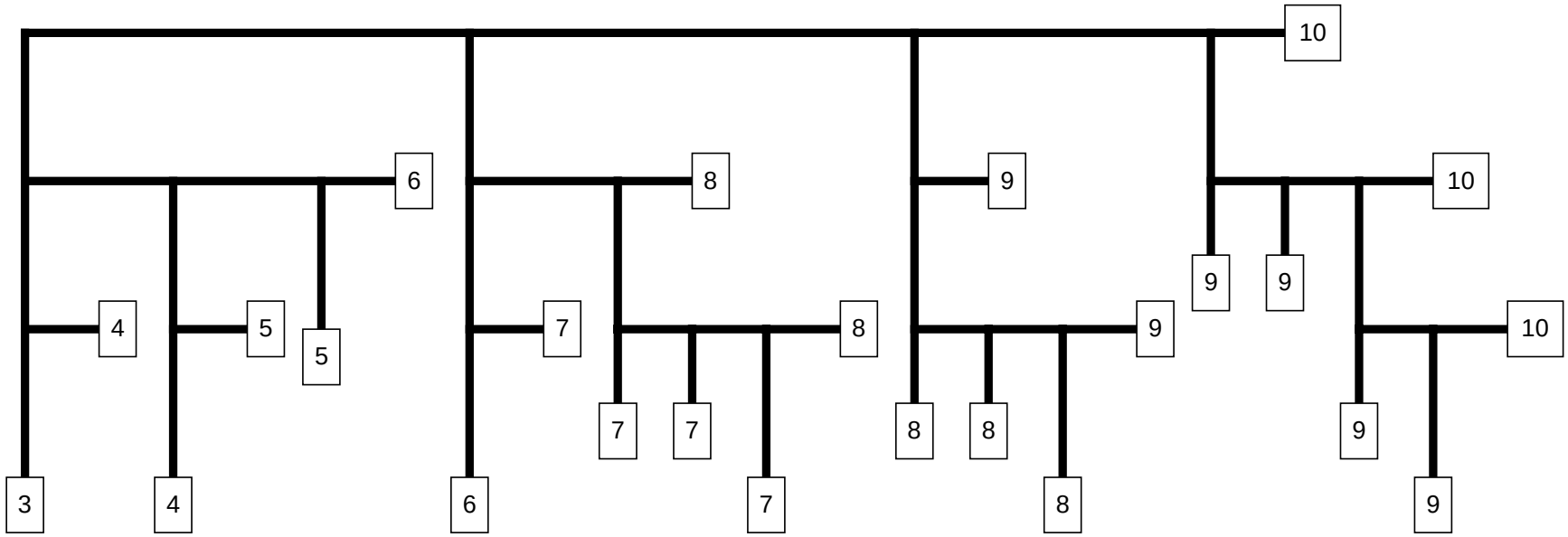
Skeleton Trees

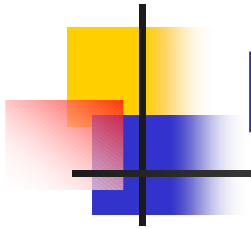
- No need to save the whole tree e.g. if a codeword starts with 1101, it ought to be of length 9 bits. Thus, we can read the following 5 bits as a block.

```
  0 000
  1 0010
  2 0011
  3 0100
  4 01010
  5 01011
  6 01100
  7 01101
  8 011100
  9 011101
 10 011110
 11 011111
 12 100000
 13 100001
 14 100010
 15 100011
 16 1001000
 17 1001001
 18 1001010
     ...
 29 1010101
 30 1010110
 31 10101110
 32 10101111
     ...
 62 11001101
 63 110011100
 64 110011101
     ...
125 111011010
126 1110110110
127 1110110111
     ...
199 1111111111
```

Illustration of a Skeleton Tree

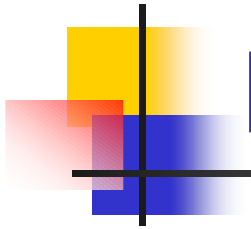
- This is the skeleton tree for the code on the previous slide. It has 49 nodes, while the original one has 399 nodes.





Definition

- Let $m=\min\{ l \mid n_l>0 \}$ where n_l is the number of codewords of length l.
- Let base(l) be:
  - $base(m)=0$
  - $base(l)=2(base(l-1)+n_{l-1})$
- Let seq(l) be:
  - $seq(m)=0$
  - $seq(l)=seq(l-1)+n_{l-1}$



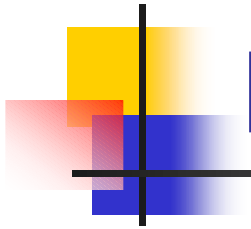
Definition (cont.)

- Let $B_s(k)$ denote the s-bit binary representation of the integer k with leading zeros if necessary.
- Let I(w) be the integer value of the binary string w, i.e. if w is of length of l, $w=B_l(I(w))$.
- I(w)-base(l) is the relative index of codeword w within the block of codewords of length l.
- seq(l)+I(w)-base(l) is the relative index of w within the full list of codewords. This can be rewritten as I(w)-diff(l), for diff(l)=base(l)-seq(l).
- Thus all one needs is the list of diff(l).

An example for values

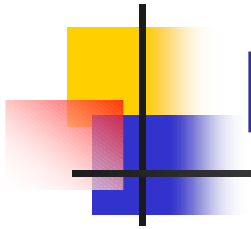
- These are the values for the code depicted on the previous slides.

l	n_i	base(l)	seq(l)	diff(l)
3	1	0	0	0
4	3	2	1	1
5	4	10	4	6
6	8	28	8	20
7	15	72	16	56
8	32	174	31	143
9	63	412	63	349
10	74	950	126	824



Decoding Algorithm

- tree_pointer←root
- i←1
- start←1
- while i<length_of_string
  - if string[i]=0 tree_pointer←left(tree_pointer)
  - else tree_pointer←right(tree_pointer)
  - if value(tree_pointer)>0
    - codeword←string[start…(start+value(tree_pointer)-1)]
    - output←table[I(codeword)-diff[value(tree_pointer)]]
    - start←start+value(tree_pointer)
    - i←start
    - tree_pointer←root
  - else i++

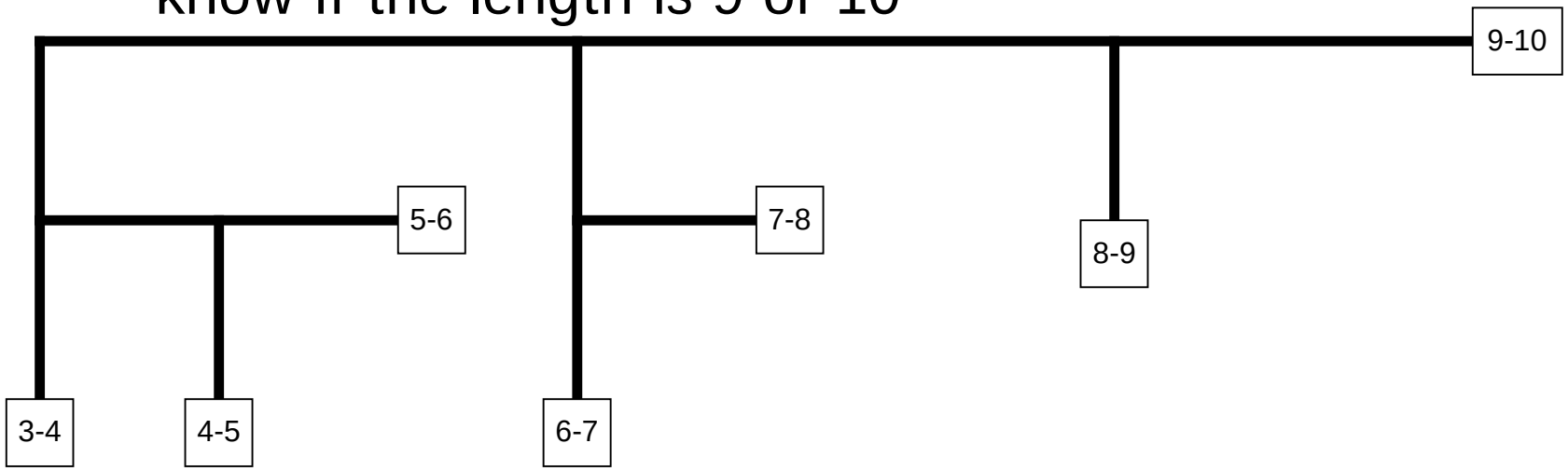


Reduced Skeleton Trees

- Define for each node v of the Skeleton Tree:
  - If v is a leaf
    - lower(v)=upper(v)=value(v)
  - If v is an internal node
    - lower(v)=lower(left(v))
    - upper(v)=upper(right(v))
- Reduced Skeleton Tree is the smallest sub-tree of the original Skeleton Tree for which all the leaves w hold:
  - $upper(w) \leq lower(w)+1$

Illustration of a reduced tree

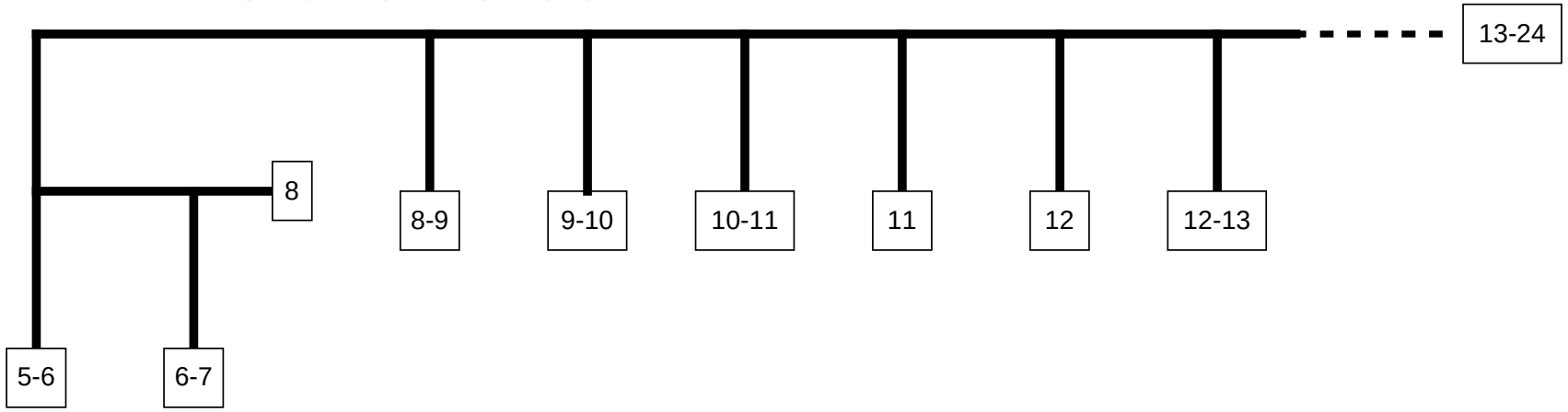
- This is the reduced tree of the previous depicted tree.
- If 111 have been read we can know that the length of the codeword is either 9 or 10.
- If the bits after 111 were 0110, we would have performed four more comparisons and still cannot know if the length is 9 or 10



Another example of a reduced tree

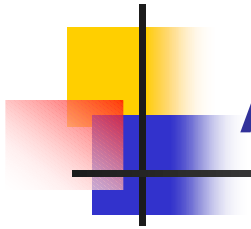
- This is a reduced Skeleton Tree for bigrams of the Hebrew Bible. Just lengths up to 13 are listed.

```
 5 00000
 6 010001
 7 0101111
 8 10011101
 9 110010011
10 1110100101
11 11110111111
12 111111010101
13 1111111011111
```



Algorithm of reduced trees

- tree_pointer←root
- i←start←1
- while i<length_of_string
  - if string[i]=0 tree_pointer←left(tree_pointer)
  - else tree_pointer←right(tree_pointer)
  - if value(tree_pointer)>0
    - len←value(tree_pointer)
    - codeword←string[start…(start+len-1)]
    - if flag(tree_pointer)=1 and 2·I(codeword)≥base(len+1)
      - codeword←string[start…(start+len)
      - len++
    - output←table[I(codeword)-diff[len]]
    - tree_pointer←root
    - i←start←start+len
  - else i++



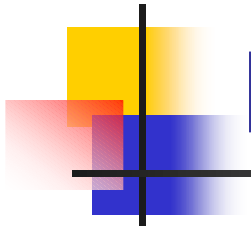
Affix codes

- Affix codes are never synchronizing, but they can be decoded backward.
- PL/1 allows files on magnetic tapes to be accessed in reverse order.
- Information Retrieval systems use concordance points to the words' locations in the text. When a word is retrieved, typically, a context of some words is displayed.

Non-Trivial Affix codes

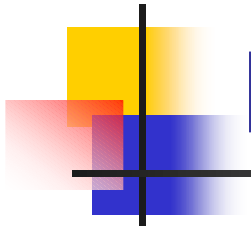
01
000
100
110
111
0010
0011
1010
1011

- Fixed length codes are called trivial affix codes.
- Theorem: There are infinite non-trivial complete affix codes.
- Proof: One non-trivial code is showed in this slide. Let $A=\{a_1,\ldots,a_n\}$ be an affix code. Consider the set $B=\{b_1,\ldots,b_{2n}\}$ defined by $b_{2i}=a_i0$, $b_{2i-1}=a_i1$ for $1\le i\le n$. Obviously B is an affix code.



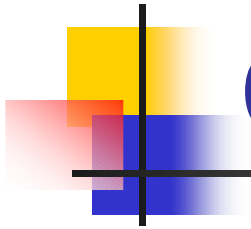
Markov chains

- A sequence of events, each of which depends only on *n* events before it, is called an n^{th} Order Markov chain.
- First order Markov chain - Event *t* is depending just on event *t-1*.
- 0^{th} order Markov chain - events are independent.
- Examples:
  - Fibonacci sequence is a second order chain.
  - An arithmetic sequence is a first order chain.



Markov chain of Huffman trees

- A different Huffman tree for each item in the set.
- The tree for an item *x* will have the probabilities of each item to appear after *x*.
- Examples:
  - u will have a much shorter codeword in q's tree, than other trees.
  - ט will have a much longer codeword after ג.
- This method implements a first order Markov chain.



Clustering

- Markov chains for Huffman trees can be expanded for n^{th} order chains.
- Overhead for saving so many trees can be very high.
- Similar trees can be clustered into a one tree, which will be the average of the original trees.
  - Example: The trees of v and b may be similar since they have a similar sound.