COMP9319 Web Data Compression and Search

LZW,
Adaptive Huffman

1

1

Dictionary coding

- Patterns: correlations between part of the data
- Idea: replace recurring patterns with references to dictionary
- LZ algorithms are adaptive:
  - Universal coding (the prob. distr. of a symbol is unknown)
  - Single pass (dictionary created on the fly)
  - No need to transmit/store dictionary

2

2

LZ77 & LZ78

- LZ77: referring to previously processed data as dictionary
- LZ78: use an explicit dictionary

3

3

Lempel-Ziv-Welch (LZW) Algorithm

- Most popular modification to LZ78
- Very common, e.g., Unix compress, TIFF, GIF, PDF (until recently)
- Read http://en.wikipedia.org/wiki/LZW regarding its patents
- Fixed-length references (12bit 4096 entries)
- Static after max entries reached

4

4

Patent issues again

From Wikipedia: "In 1993–94, and again in 1999, Unisys Corporation received widespread condemnation when it attempted to enforce licensing fees for LZW in GIF images. The 1993–1994 Unisys-Compuserve (Compuserve being the creator of the GIF format) controversy engendered a Usenet comp.graphics discussion *Thoughts on a GIF-replacement file format*, which in turn fostered an email exchange that eventually culminated in the creation of the patent-unencumbered Portable Network Graphics (PNG) file format in 1995. Unisys's US patent on the LZW algorithm expired on June 20, 2003 ..."

5

5

LZW Compression

```
p = nil;    // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

6

6

Example

Input: ^WED^WE^WEE^WEB^WET

p	c	output	index	symbol
NIL	^			

Dict [index]

```
p = nil;     // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

7

7

Example

Input: ^WED^WE^WEE^WEB^WET

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W

Dict [index]

```
p = nil;     // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

8

8

Example

Input: ^WED^WE^WEE^WEB^WET

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE

Dict [index]

```
p = nil;     // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

9

9

Example

Input: ^WED^WE^WEE^WEB^WET

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			

Dict [index]

```
p = nil;     // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

10

10

Example

Input: ^WED^WE^WEE^WEB^WET

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE

Dict [index]

```
p = nil;     // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

11

11

Example

Input: ^WED^WE^WEE^WEB^WET

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE
E	^	E	261	E^

Dict [index]

```
p = nil;     // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

12

12

Example

Input: ^WED^WE^WEE^WEB^WET

Dict [index]

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE
E	^	E	261	E^
^	W			

```
p = nil;    // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

13

13

Example

Input: ^WED^WE^WEE^WEB^WET

Dict [index]

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE
E	^	E	261	E^
^	W			
^W	E			

```
p = nil;    // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

14

14

Example

Input: ^WED^WE^WEE^WEB^WET

Dict [index]

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE
E	^	E	261	E^
^	W			
^W	E			
^WE	E	260	262	^WEE

```
p = nil;    // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

15

15

Example

Input: ^WED^WE^WEE^WEB^WET

Dict [index]

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE
E	^	E	261	E^
^	W			
^W	E			
^WE	E	260	262	^WEE
E	^			

```
p = nil;    // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

16

16

Example

Input: ^WED^WE^WEE^WEB^WET

Dict [index]

p	c	output	index	symbol
NIL	^			
^	W	^	256	^W
W	E	W	257	WE
E	D	E	258	ED
D	^	D	259	D^
^	W			
^W	E	256	260	^WE
E	^	E	261	E^
^	W			
^W	E			
^WE	E	260	262	^WEE
E	^			
E^	W	261	263	E^W
W	E			
WE	B	257	264	WEB
B	^	B	265	B^
^	W			
^W	E			
^WE	T	260	266	^WET
T	EOF	T		

```
p = nil;    // p for prev char
while read(c):
    if pc ∈ Dict:
        p = pc;
    else:
        add pc to Dict;
        output code(p);
        p = c;
```

17

17

LZW Compression

- Original LZW used dictionary with 4K entries, first 256 (0-255) are ASCII codes.
- In the above example, a 19 symbols reduced to 7 symbols & 5 code. Each code/symbol will need 8+ bits, say 9 bits.
- Reference: Terry A. Welch, "A Technique for High Performance Data Compression", IEEE Computer, Vol. 17, No. 6, 1984, pp. 8-19.

18

18

LZW Decompression

```
read(c);    // c is likely to be > 8 bits
output c;
p = c;
while read(c):
      output Dict[c];
      add p + Dict[c][0] to Dict;
      p = Dict[c];
```

19

19

Example

Input: ^WED<256>E<260><261><257>B<260>T

```
ead(c);    // c is likely to be > 8 bits
)utput c;
) = c;
vhile read(c):
   output Dict[c];
   add p + Dict[c][0] to Dict
   p = Dict[c];
```

p	c	output	Dict [index] index	symbol
	^	^		

20

20

Example

Input: ^WED<256>E<260><261><257>B<260>T

```
ead(c);    // c is likely to be > 8 bits
)utput c;
) = c;
vhile read(c):
   output Dict[c];
   add p + Dict[c][0] to Dict
   p = Dict[c];
```

p	c	output	Dict [index] index	symbol
	^	^		
^	W	W	256	^W

21

21

Example

Input: ^WED<256>E<260><261><257>B<260>T

```
ead(c);    // c is likely to be > 8 bits
)utput c;
) = c;
vhile read(c):
   output Dict[c];
   add p + Dict[c][0] to Dict
   p = Dict[c];
```

p	c	output	Dict [index] index	symbol
	^	^		
^	W	W	256	^W
W	E	E	257	WE

22

22

Example

Input: ^WED<256>E<260><261><257>B<260>T

```
ead(c);    // c is likely to be > 8 bits
)utput c;
) = c;
vhile read(c):
   output Dict[c];
   add p + Dict[c][0] to Dict
   p = Dict[c];
```

p	c	output	Dict [index] index	symbol
	^	^		
^	W	W	256	^W
W	E	E	257	WE
E	D	D	258	ED

23

23

Example

Input: ^WED<256>E<260><261><257>B<260>T

```
ead(c);    // c is likely to be > 8 bits
)utput c;
) = c;
vhile read(c):
   output Dict[c];
   add p + Dict[c][0] to Dict
   p = Dict[c];
```

p	c	output	Dict [index] index	symbol
	^	^		
^	W	W	256	^W
W	E	E	257	WE
E	D	D	258	ED
D	<256>	^W	259	D^

24

24

Example

Input: ^WED<256>E<260><261><257>B<260>T

```
ead(c);    // c is likely to be > 8 bits
)utput c;
) = c;
vhile read(c):
    output Dict[c];
    add p + Dict[c][0] to Dict
    p = Dict[c];
```

Dict [index]

p	c	output	index	symbol
	^	^		
^	W	W	256	^W
W	E	E	257	WE
E	D	D	258	ED
D	<256>	^W	259	D^
<256>	E	E	260	^WE

25

25

Example

Input: ^WED<256>E<260><261><257>B<260>T

```
ead(c);    // c is likely to be > 8 bits
)utput c;
) = c;
vhile read(c):
    output Dict[c];
    add p + Dict[c][0] to Dict
    p = Dict[c];
```

Dict [index]

p	c	output	index	symbol
	^	^		
^	W	W	256	^W
W	E	E	257	WE
E	D	D	258	ED
D	<256>	^W	259	D^
<256>	E	E	260	^WE
E	<260>	^WE	261	E^

26

26

Example

Input: ^WED<256>E<260><261><257>B<260>T

```
ead(c);    // c is likely to be > 8 bits
)utput c;
) = c;
vhile read(c):
    output Dict[c];
    add p + Dict[c][0] to Dict
    p = Dict[c];
```

Dict [index]

p	c	output	index	symbol
	^	^		
^	W	W	256	^W
W	E	E	257	WE
E	D	D	258	ED
D	<256>	^W	259	D^
<256>	E	E	260	^WE
E	<260>	^WE	261	E^
<260>	<261>	E^	262	^WEE

27

27

Example

Input: ^WED<256>E<260><261><257>B<260>T

```
ead(c);    // c is likely to be > 8 bits
)utput c;
) = c;
vhile read(c):
    output Dict[c];
    add p + Dict[c][0] to Dict
    p = Dict[c];
```

Dict [index]

p	c	output	index	symbol
	^	^		
^	W	W	256	^W
W	E	E	257	WE
E	D	D	258	ED
D	<256>	^W	259	D^
<256>	E	E	260	^WE
E	<260>	^WE	261	E^
<260>	<261>	E^	262	^WEE
<261>	<257>	WE	263	E^W
<257>	B	B	264	WEB
B	<260>	^WE	265	B^
<260>	T	T	266	^WET

28

28

Note: LZW decoding

- There is one special case that the LZW decoding pseudocode presented is unable to handle.
- This is your exercise to find out in what situation that happens, and how to deal with it.
- I'll go through this at the live lecture.

29

29

LZW implementation

- Parsing fixed number of bits from input is easy
- Fast and efficient

30

30

Types (revision)

- Block-block
  - source message and codeword: fixed length
  - e.g., ASCII
- Block-variable
  - source message: fixed; codeword: variable
  - e.g., Huffman coding
- Variable-block
  - source message: variable; codeword: fixed
  - e.g., LZW
- Variable-variable
  - source message and codeword: variable
  - e.g., Arithmetic coding

31

31

So far

We have covered:

- Course overview
- Background
- RLE
- Entropy
- Huffman code
- Arithmetic code
- LZW

32

32

More online readings

http://www.ics.uci.edu/~dan/pubs/DC-Sec1.html

http://marknelson.us/1991/02/01/arithmetic-coding-statistical-modeling-data-compression/

33

33

Lossless compression revisited

- Run-length coding
- Statistical methods
  - Huffman coding
  - Arithmetic coding
- Dictionary methods
  - Lempel Ziv algorithms

Static (Huffman, AC) vs Adaptive (LZW)

34

34

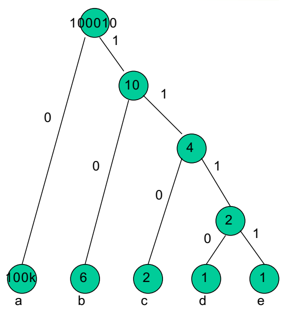
Huffman Coding (revisit)
and then
Adaptive Huffman

35

35

Huffman coding

S	Freq	Huffman
a	100000	0
b	6	10
c	2	110
d	1	1110
e	1	1111



36

36

Huffman not optimal

H = 0.9999 log 1.0001 + 0.00006 log 16668.333 + … + 1/100010 log 100010
≈ 0.00

L = (100000*1 + …)/100010
≈ 1

37

37

Problems of Huffman coding

Huffman codes have an integral # of bits.

- E.g., log (3) = 1.585 while Huffman may need 2 bits

Noticeable non-optimality when prob of a symbol is high.

=> Arithmetic coding

38

38

Problems of Huffman coding

Need statistics & static: e.g., single pass over the data just to collect stat & stat unchanged during encoding

To decode, the stat table need to be transmitted. Table size can be significant for small msg.

=> Adaptive compression e.g., adaptive huffman

39

39

Adaptive compression

<u>Encoder</u>
Initialize the model
Repeat for each input char
(
  Encode char
  Update the model
)

<u>Decoder</u>
Initialize the model
Repeat for each input char
(
  Decode char
  Update the model
)

Make sure both sides have the same Initialize & update model algorithms.

40

40

Adaptive Huffman Coding (dummy)

<u>Encoder</u>
Reset the stat
Repeat for each input char
(
  Encode char
  Update the stat
  Rebuild huffman tree
)

<u>Decoder</u>
Reset the stat
Repeat for each input char
(
  Decode char
  Update the stat
  Rebuild huffman tree
)

41

41

Adaptive Huffman Coding (dummy)

<u>Encoder</u>
Reset the stat
Repeat for each input char
(
  Encode char
  Update the stat
  Rebuild huffman tree
)

<u>Decoder</u>
Reset the stat
Repeat for each input char
(
  Decode char
  Update the stat
  Rebuild huffman tree
)

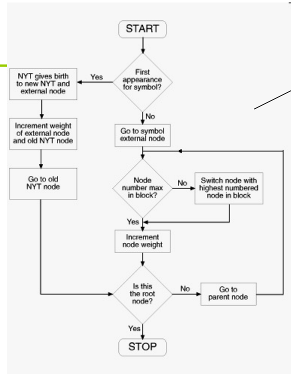
This works but too slow!

42

42

Adaptive Huffman (Algorithm outline)

1. If current symbol is NYT, add two child nodes to NYT node. One will be a new NYT node the other is a leaf node for our symbol. Increase weight for the new leaf node and the old NYT and go to step 4. If not, go to symbol's leaf node.
2. If this node does not have the highest number in a block, swap it with the node having the highest number
3. Increase weight for current node
4. If this is not the root node go to parent node then go to step 2. If this is the root, end.

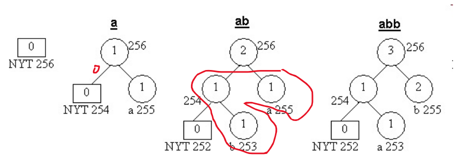


The update procedure from Introduction to Data Compression by by Sayood Khalid

Adaptive Huffman

abbbbba: 0110000101100010011000100110001001100010011000100110001001100001

abbbbba: 01100001001100010001111101



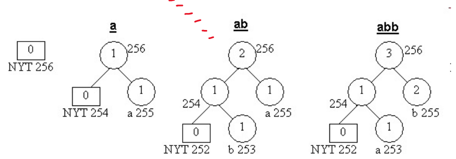
a: 01100001
b: 01100010

From an old Wikipedia page

Adaptive Huffman

abbbbba: 0110000101100010011000100110001001100010011000100110001001100001

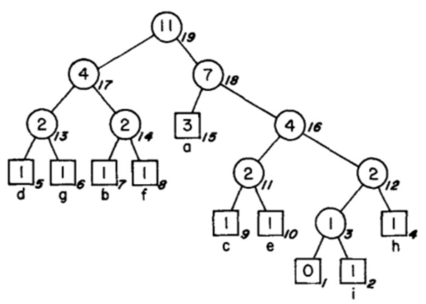
abbbbba: 01100001001100010001111101



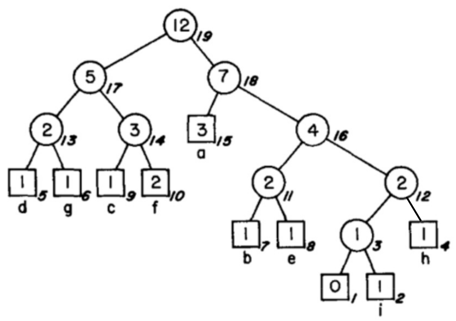
a: 01100001
b: 01100010

From an old Wikipedia page

Adaptive Huffman (FGK)



Adaptive Huffman (FGK): when f is inserted



Adaptive Huffman (FGK vs Vitter)

1.
FGK: (Explicit) node numbering
Vitter: Implicit numbering

2.
Vitter's Invariant:

(*) For each weight w, all leaves of weight w precede (in the implicit numbering) all internal nodes of weight w.

Adaptive Huffman (Vitter'87)

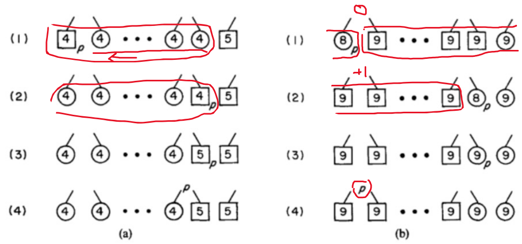
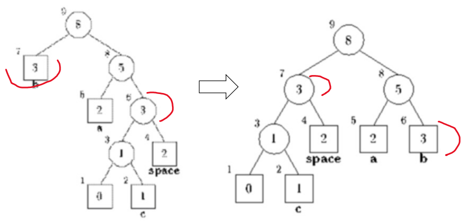
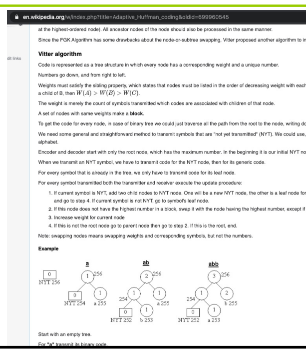


FIG. 6. Algorithm Λ's *SlideAndIncrement* operation. All the nodes in a given block shift to the left one spot to make room for node p, which slides over the block to the right. (a) Node p is a leaf of weight 4. The internal nodes of weight 4 shift to the left. (b) Node p is an internal node of weight 8. The leaves of weight 9 shift to the left.

Adaptive Huffman (Vitter's Invariant)



Issues with Wikipedia



en.wikipedia.org/w/index.php?title=Adaptive_Huffman_coding&oldid=699960545

at the highest-ordered node). All ancestor nodes of the node should also be processed in the same manner.

Since the FGK Algorithm has some drawbacks about the node-or-subtree swapping, Vitter proposed another algorithm to in

Vitter algorithm

Code is represented as a tree structure in which every node has a corresponding weight and a unique number.

Numbers go down, and from right to left.

Weights must satisfy the sibling property, which states that nodes must be listed in the order of decreasing weight with each a child of B, then $W(A) > W(B) > W(C)$.

The weight is merely the count of symbols transmitted which codes are associated with children of that node.

A set of nodes with same weights make a **block**.

To get the code for every node, in case of binary tree we could just traverse all the path from the root to the node, writing dc

We need some general and straightforward method to transmit symbols that are "not yet transmitted" (NYT). We could use, alphabet.

Encoder and decoder start with only the root node, which has the maximum number. In the beginning it is our initial NYT no

When we transmit an NYT symbol, we have to transmit code for the NYT node, then for its generic code.

For every symbol that is already in the tree, we only have to transmit code for its leaf node.

For every symbol transmitted both the transmitter and receiver execute the update procedure:

1. If current symbol is NYT, add two child nodes to NYT node. One will be a new NYT node, the other is a leaf node for and go to step 4. If current symbol is not NYT, go to symbol's leaf node.
2. If this node does not have the highest number in a block, swap it with the node having the highest number, except if
3. Increase weight for current node
4. If this is not the root node go to parent node then go to step 2. If this is the root, end.

Note: swapping nodes means swapping weights and corresponding symbols, but not the numbers.

Example

Start with an empty tree.

For "a" transmit its binary code.

COMP9319 students correcting lots of Wiki pages

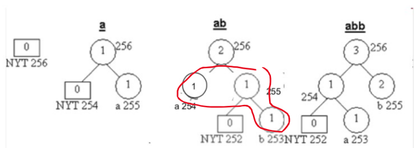
en.wikipedia.org/w/index.php?title=Adaptive_Huffman_coding&action=history

- (cur | prev) 11:23, 17 March 2016 Victor.choudhary (talk | contribs) m . . (8,346 bytes) (-32) . . (→Algorithm :-) (undo) (Tag: Visual edit)
- (cur | prev) 11:20, 17 March 2016 Victor.choudhary (talk | contribs) m . . (8,378 bytes) (-31) . . (→Algorithm) (undo) (Tag: Visual edit)
- (cur | prev) 11:20, 17 March 2016 Victor.choudhary (talk | contribs) m . . (8,409 bytes) (+33) . . (→Algorithm :-) (undo) (Tag: Visual edit)
- (cur | prev) 06:33, 17 March 2016 Yobot (talk | contribs) m . . (8,376 bytes) (-224) . . (Removed invisible unicode characters + other fixes, replaced: → (38) using AWB (11972)) (undo)
- (cur | prev) 06:32, 17 March 2016 Bebound (talk | contribs) m . . (8,600 bytes) (+4) . . (Fix ref error) (undo)
- (cur | prev) 04:48, 17 March 2016 BG19bot (talk | contribs) m . . (8,596 bytes) (-2) . . (v1.38b - WP:WCW project (Unicode control characters)) (undo) (Tag: WPCleaner)
- (cur | prev) 18:19, 16 March 2016 Victor.choudhary (talk | contribs) m . . (8,598 bytes) (+1) . . (→Algorithm :- improved spacing) (undo) (Tag: Visual edit)
- (cur | prev) 18:17, 16 March 2016 Victor.choudhary (talk | contribs) . . (8,597 bytes) (+942) . . (→Vitter algorithm: A more accurate description of Vitter's Algorithm.) (undo) (Tags: Visual edit, nowiki added)
- (cur | prev) 13:15, 16 March 2016 Pang Luo (talk | contribs) m . . (7,655 bytes) (+3) . . (Minor Formatting) (undo) (Tag: Visual edit)
- (cur | prev) 13:08, 16 March 2016 Pang Luo (talk | contribs) m . . (7,652 bytes) (-10) . . (Minor Formatting) (undo) (Tag: Visual edit)
- (cur | prev) 13:07, 16 March 2016 Pang Luo (talk | contribs) m . . (7,662 bytes) (+9) . . (Minor formatting.) (undo) (Tag: Visual edit)
- (cur | prev) 12:42, 16 March 2016 Pang Luo (talk | contribs) m . . (7,653 bytes) (+788) . . (Correcting some typos.) (undo) (Tag: Visual edit)
- (cur | prev) 12:17, 16 March 2016 Pang Luo (talk | contribs) . . (6,865 bytes) (+855) . . (The original description has a bug and doesn't exactly reflect Vitter's algorithm. The new description eliminates this bug.) (undo) (Tag: Visual edit)
- (cur | prev) 11:05, 16 March 2016 Pang Luo (talk | contribs) . . (6,010 bytes) (-27) . . (The original image has a bug that it doesn't maintain Vitter's invariant that all leaves with weight w must precede (in the implicit numbering) all internal nodes with weight w. The new image eliminates this bug.) (undo) (Tag: Visual edit)

Adaptive Huffman (Vitter's)

abbbbba: 01100001011000100110001001100010011000100110001001100001

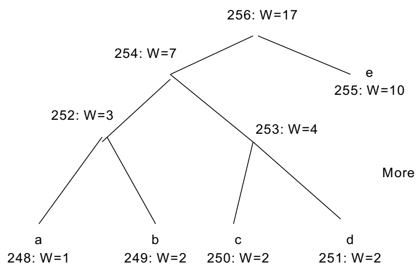
abbbbba: 0110000100110001011111101



a: 01100001
b: 01100010

Corrected fr Wikipedia

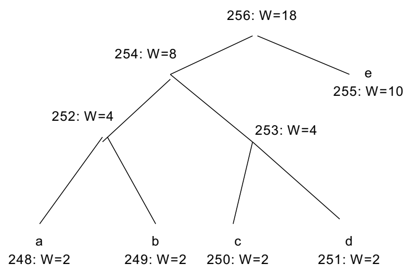
More example on Vitter's



More aaaa…. coming

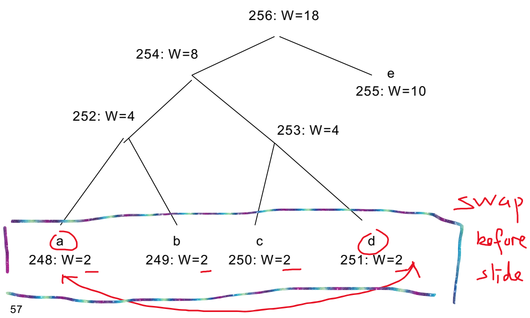
55

More example



56

More example



57

The Vitter's algo: swaps then slides

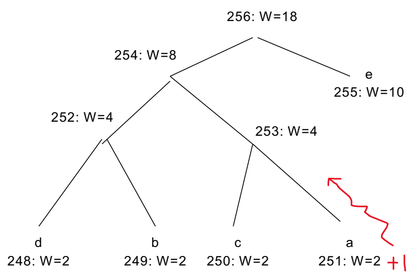
Definition 4.1. *Blocks* are equivalence classes of nodes defined by $v \equiv x$ iff nodes v and x have the same weight and are either both internal nodes or both leaves. The *leader* of a block is the highest numbered (in the implicit numbering) node in a block.

The blocks are linked together by increasing order of weight; a leaf block always precedes an internal block of the same weight. The main operation of the algorithm needed to maintain invariant $(*)$ is the *SlideAndIncrement* operation, illustrated in Figure 6. The version of *Update* we use for Algorithm Λ is outlined below:

```
procedure Update;
begin
leafToIncrement := 0;
q := leaf node corresponding to a_{i_{t+1}};
if (q is the 0-node) and (k < n − 1) then
   begin   {Special Case #1}
   Replace q by an internal 0-node with two leaf 0-node children, such that the right child
         corresponds to a_{i_{t+1}};
   q := internal 0-node just created;
   leafToIncrement := the right child of q
   end
else begin
   Interchange q in the tree with the leader of its block;
   if q is the sibling of the 0-node then
      begin   {Special Case #2}
      leafToIncrement := q;
      q := parent of q
      end
   end;
while q is not the root of the Huffman tree do
   {Main loop; q must be the leader of its block}
   SlideAndIncrement(q);
if leafToIncrement ≠ 0 then   {Handle the two special cases}
   SlideAndIncrement(leafToIncrement)
end;
```

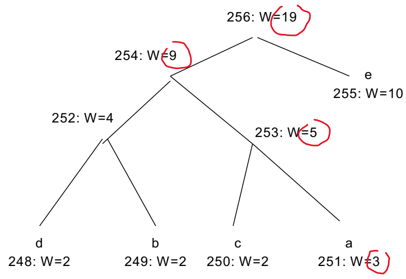

58

More example



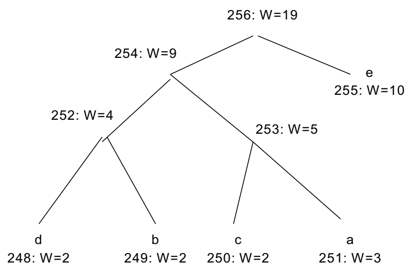
59

More example



60

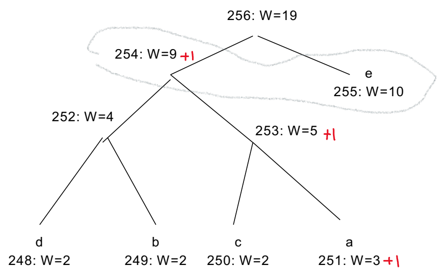
More example



61

61

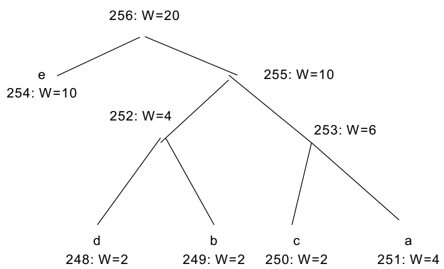
More example



62

62

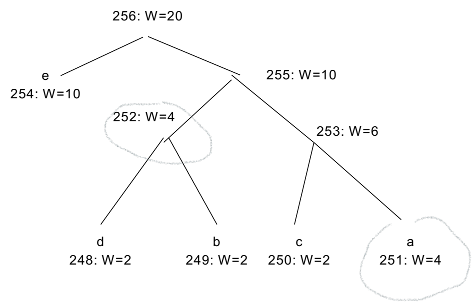
More example



63

63

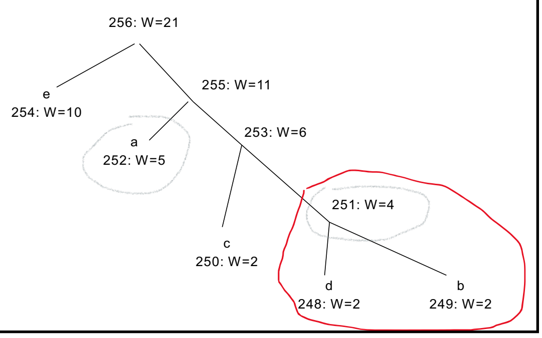
More example



64

64

More example



65

65

Adaptive Huffman

Question: Adaptive Huffman vs Static Huffman

66

66

Compared with Static Huffman

Dynamic and can offer better compression (cf. Vitter's experiments next)

Works when prior stat is unavailable

Saves symbol table overhead (cf. Vitter's expt next)

67

67

Vitter's experiments

Include overheads such as symbol tables / leaf node code etc.

t	k	S_t	b/l	$D_t^\wedge$	b/l
100	96	664	13.1	569	10.2
500	96	3320	7.9	3225	7.4
960	96	6400	7.1	6305	6.8

Exclude overheads such as symbol tables / leaf node code etc.

95 ASCII chars + <end-of-line>

From Vitter's paper. You know where it is. ☺

68

68

More experiments

t	k	S_t	b/l	$D_t^\wedge$	b/l
100	34	434	7.1	420	6.3
500	52	2429	5.7	2445	5.5
1000	58	4864	5.3	4900	5.2
10000	74	47710	4.8	47852	4.8
12280	76	58457	4.8	58614	4.8

69

69