Web Data Management

Compression and Search

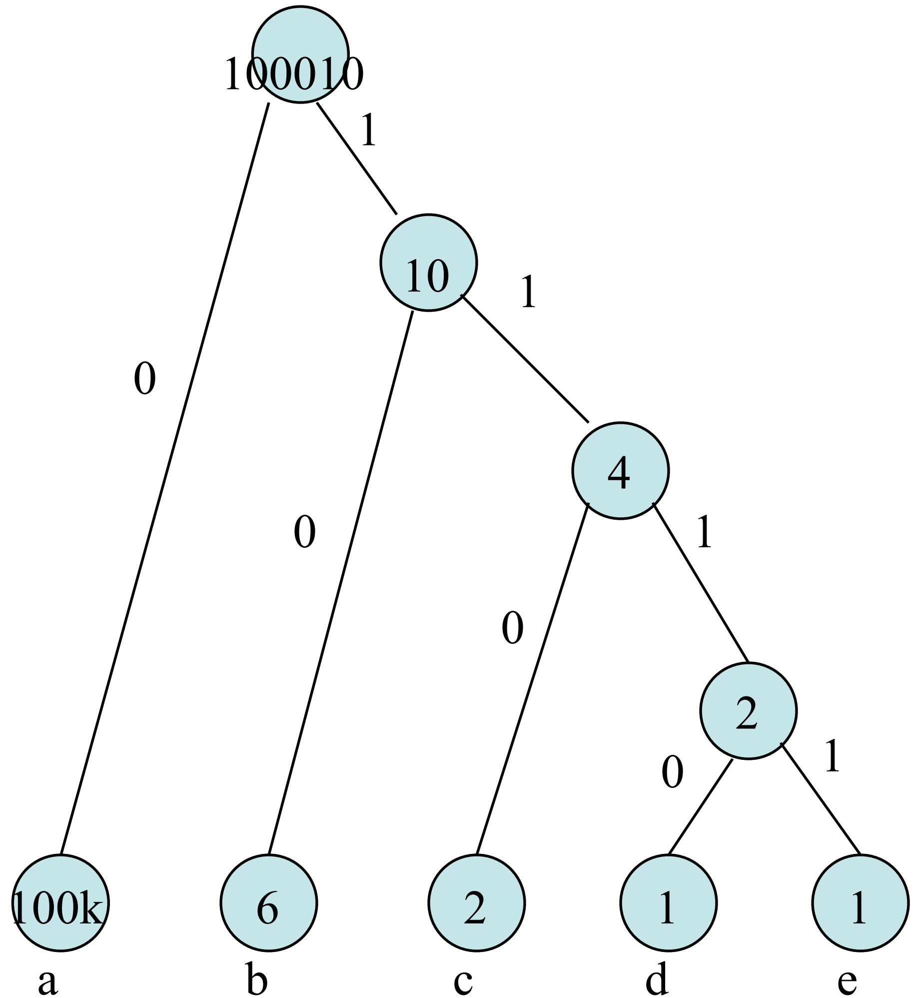
Lecture 2: Adaptive Huffman, BWT

Course schedule

- Data compression
- Search
- Data compression + Search

Huffman coding

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



Huffman not optimal

H $= 0.9999 \log 1.0001 + 0.00006 \log 16668.333$
$+ \ldots + 1/100010 \log 100010$
$\approx$ **0.00**

L $= (100000*1 + \ldots)/100010$
$\approx$ **1**

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

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

Adaptive compression

Encoder

Initialize the model

Repeat for each input char

(

Encode char

Update the model

)

Decoder

Initialize the model

Repeat for each input char

(

Decode char

Update the model

)

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

Adaptive Huffman Coding (dummy)

Encoder

Reset the stat

Repeat for each input char

(

Encode char

Update the stat

Rebuild huffman tree

)

Decoder

Reset the stat

Repeat for each input char

(

Decode char

Update the stat

Rebuild huffman tree

)

Adaptive Huffman Coding (dummy)

Encoder

Reset the stat

Repeat for each input char

(

Encode char

Update the stat

Rebuild huffman tree

)

Decoder

Reset the stat

Repeat for each input char

(

Decode char

Update the stat

Rebuild huffman tree

)

This works but too slow!

The key idea

- The key idea is to build a Huffman tree that is optimal for the part of the message already seen, and to reorganize it when needed, to maintain its optimality

Pro & Con - I

- Adaptive Huffman determines the mapping to codewords using a *running estimate* of the source symbols probabilities
  - **Effective exploitation of locality**

    For example, suppose that a file starts out with a series of a character that are not repeated again in the file. In static Huffman coding, that character will be low down on the tree because of its low overall count, thus taking lots of bits to encode. In adaptive huffman coding, the character will be inserted at the highest leaf possible to be decoded, before eventually getting pushed down the tree by higher-frequency characters

Pro & Con - II

– **only one pass over the data**

- overhead

  In static Huffman, we need to transmit someway the model used for compression, i.e. the tree shape.

  This costs about **2*n*** bits in a clever representation.

  As we will see, in adaptive schemes the overhead is $n\log n$.

- sometimes encoding needs some more bits w.r.t. static Huffman. But adaptive schemes generally compare well with static Huffman

Some history

- Adaptive Huffman coding was first conceived independently by Faller (1973) and Gallager (1978)
- Knuth contributed improvements to the original algorithm (1985) and the resulting algorithm is referred to as algorithm FGK
- A more recent version of adaptive Huffman coding is described by Vitter (1987) and called algorithm V

An important question

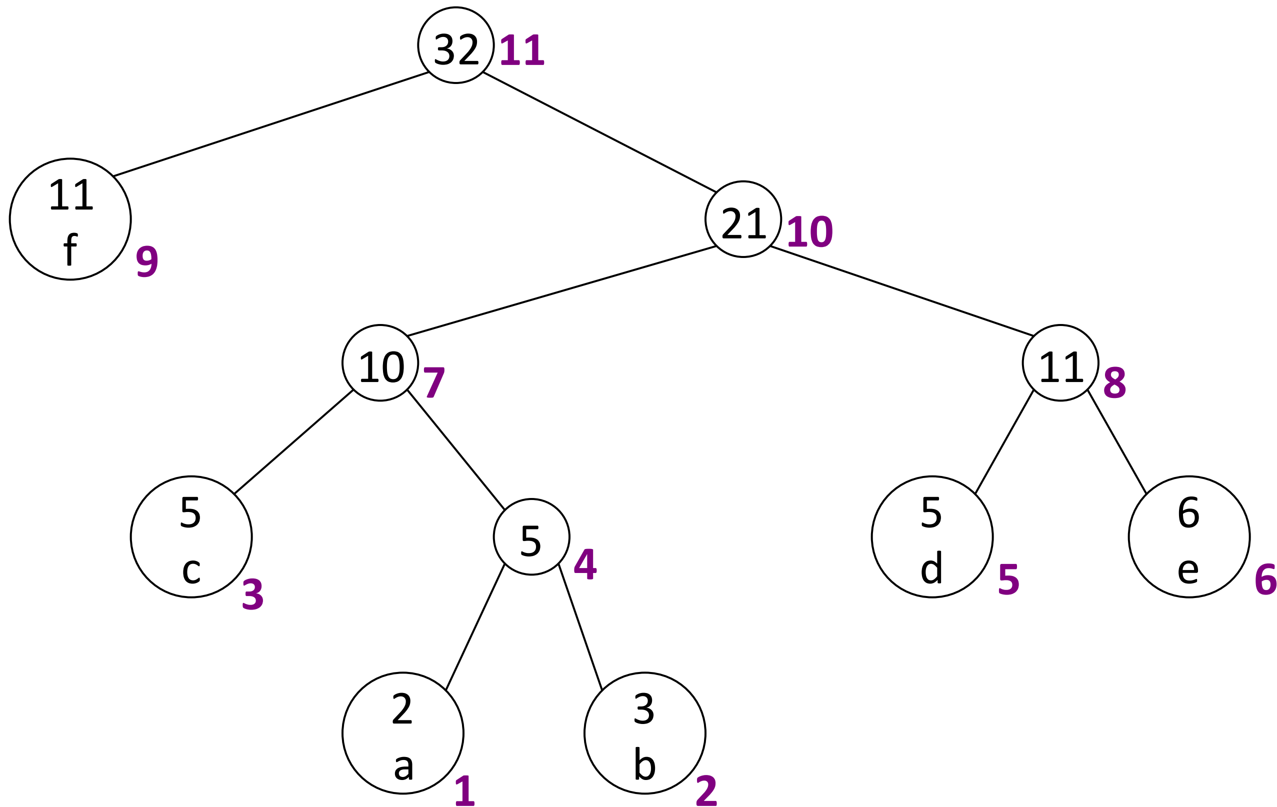
- Better exploiting locality, adaptive Huffman coding is sometimes able to do better than static Huffman coding, i.e., for some messages, it can have a greater compression
- ... but we've assessed optimality of static Huffman coding, in the sense of minimal redundancy

There is a contradiction?

Algorithm FGK - I

- The basis for algorithm FGK is the Sibling Property (Gallager 1978)
  - A binary code tree with nonnegative weights has the sibling property if each node (except the root) has a sibling and if the nodes can be numbered in order of nondecreasing weight with each node adjacent to its sibling.
  - Moreover the parent of a node is higher in the numbering
- A binary prefix code is a Huffman code if and only if the code tree has the sibling property

Algorithm FGK - II

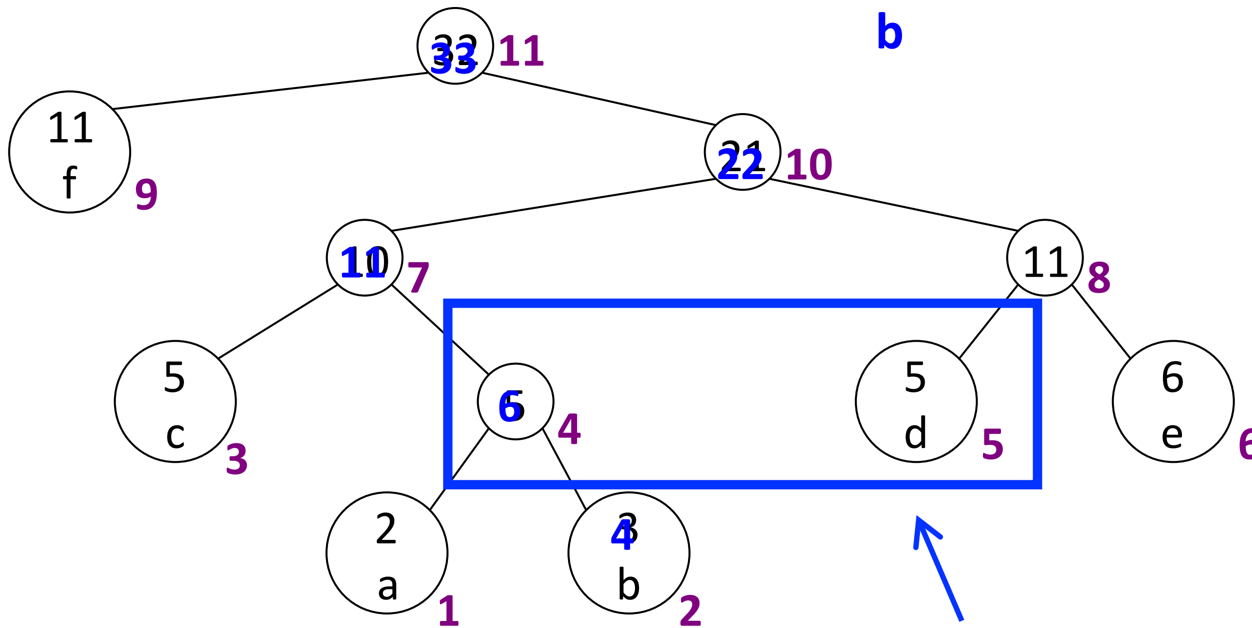


- Note that node numbering corresponds to the order in which the nodes are combined by Huffman's algorithm, first nodes 1 and 2, then nodes 3 and 4 ...

Algorithm FGK - III

- In algorithm FGK, both encoder and decoder maintain dynamically changing Huffman code trees. For each symbol the encoder sends the codeword for that symbol in current tree and then update the tree
  - The problem is to change quickly the tree optimal after t symbols (not necessarily distinct) into the tree optimal for t+1 symbols
  - If we simply increment the weight of the t+1-th symbols and of all its ancestors, the sibling property may no longer be valid → we must rebuild the tree

Algorithm FGK - IV



- Suppose next symbol is “b”
- if we update the weigths...
- ... sibling property is violated!!
- This is no more a Huffman tree

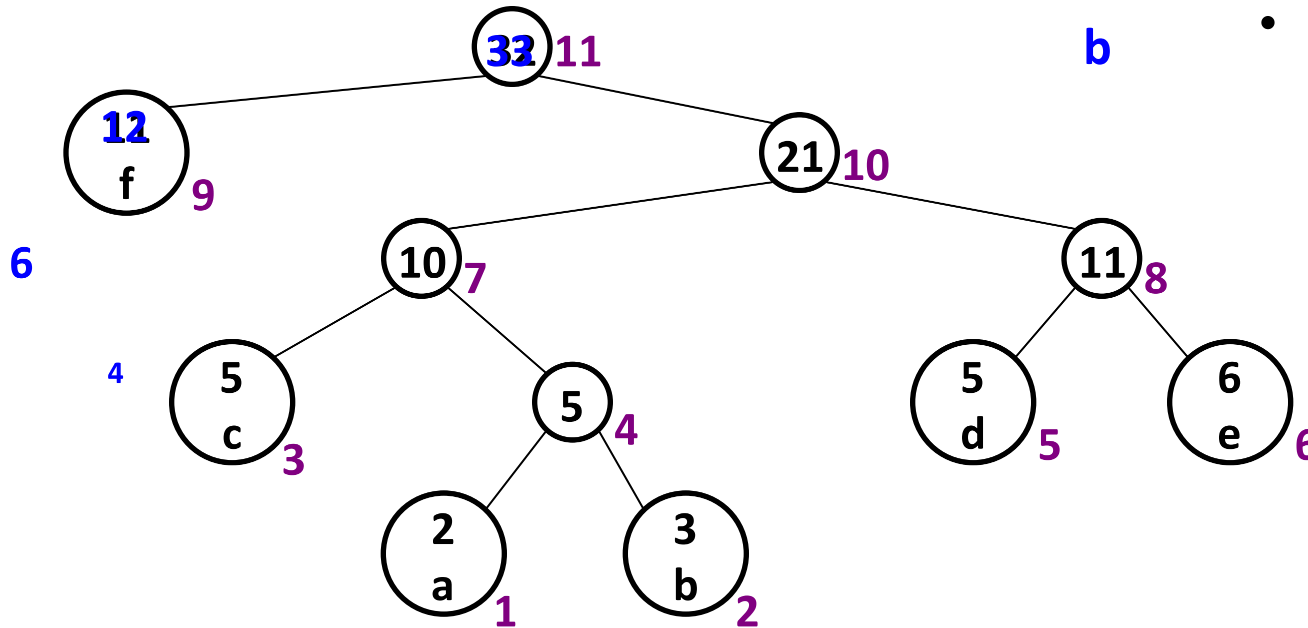
Algorithm FGK - V

- The solution can be described as a two-phase process
  - first phase: original tree is transformed into another valid Huffman tree for the first **t** symbols, that has the property that **simple increment process can be applied succesfully (How?)**
  - second phase: increment process, as described previously

Algorithm FGK - V

- The first phase starts at the leaf of the t+1-th symbol
- We swap this node and all its subtree, but not its numbering, with the highest numbered node of the same weight
- New current node is
  - the parent of this latter node
- The process is repeated until we reach the root

Algorithm FGK - VI



- **First phase**
  - Node 2: nothing to be done
  - Node 4: to be swapped with node 5
  - Node 8: to be swapped with node 9
  - Root reached: stop!
- **Second phase**

Why FGK works?

- The two phase procedure builds a valid Huffman tree for t+1 symbols, as the sibling properties is satisfied
  - In fact, we swap each node which weight is to be increased with the highest numbered node with the same weight
  - After the increasing process there is no node with previous weight that is higher numbered

The Not Yet Seen problem - I

- When the algorithm starts and sometimes during the encoding we encounter a symbol that has not been seen before.
  How do we face this problem?

The Not Yet Seen problem - I

- We use a single 0-node (with weight 0) that represents all the unseen symbols. When a new symbol appears we send **the code for the 0-node and some bits to discern which is the new symbol**.
  - As each time we send log$n$ bits to discern the symbol, total overhead is $n$log$n$ bits
  - It is possible to do better, sending only the index of the symbol in the list of the current unseen symbols.

    In this way we can save some bit, on average

The Not Yet Seen problem - II

- Then the 0-node is splitted into two leaves, that are sibling, one for the new symbol, with weight 1, and a new 0-node
- Then the tree is recomputed as seen before in order to satisfy the sibling property

Algorithm FGK - summary

- The algorithm starts with only one leaf node, the 0-node. As the symbols arrive, new leaves are created and each time the tree is recomputed
- Each symbol is coded with its codeword in the current tree, and then the tree is updated
- Unseen symbols are coded with 0-node codeword and some other bits are needed to specify the symbol

Algorithm FGK - VII

- Algorithm FGK compares favourably with static Huffman code, if we consider also overhead costs (it is used in the Unix utility *compact*)
- **Exercise**
  - Construct the static Huffman tree and the FGK tree for the message *e eae de eabe eae dcf* and evaluate the number of bits needed for the coding with both the algorithms, ignoring the overhead for Huffman

Algorithm FGK - VIII

- if T="total number of bits transmitted by algorithm FGK for a message of length t containing n distinct symbols", then

$$S - n + 1 \leq T \leq 2S + t - 4n + 2$$

where S is the performance of the static Huffman (Vitter 1987)

- So the performance of algorithm FGK is never much worse than twice optimal

Algorithm V - I

- Vitter in his work of the 1987 introduces two improvements over algorithm FGK, calling the new scheme algorithm Λ
- As a tribute to his work, the algorithm is become famous... with the letter flipped upside-down... algorithm V

The key ideas - I

- swapping of nodes during encoding and decoding is onerous
  - In FGK algorithm the number of swapping (considering a double cost for the updates that move a swapped node two levels higher) is bounded by $\lceil d_t/2 \rceil$, where d_t is the length of the added symbol in the old tree (this bound require some effort to be proved and is due to the work of Vitter)
  - In algorithm V, the number of swapping is bounded by 1

The key ideas - II

- Moreover algorithm V, not only minimize $\sum_i w_i l_i$ as Huffman and FGK, but also $\max_i l_i$, i.e. the height of the tree, and $\sum_i l_i$, i.e. is better suited to code next symbol, given it could be represented by any of the leaves of the tree

- This two objectives are reached through a new numbering scheme, called *implicit numbering*

Implicit numbering

- The nodes of the tree are numbered in increasing order by level; nodes on one level are numbered lower than the nodes on the next higher level
- Nodes on the same level are numbered in increasing order from left to right
- If this numbering is satisfied (and in FGK it is not always satisfied), certain types of updates cannot occur

An invariant

- The key to minimize the other kind of interchanges is to maintain the following *invariant*
  - *for each weight w, all leaves of weight w precede (in the implicit numbering) all internal nodes of weight w*
- The interchanges, in the algorithm V, are designed to restore implicit numbering, when a new symbol is read, and to preserve the invariant

Algorithm V - II

- if T=“total number of bits transmitted by algorithm V for a message of length t containing n distinct symbols“, then

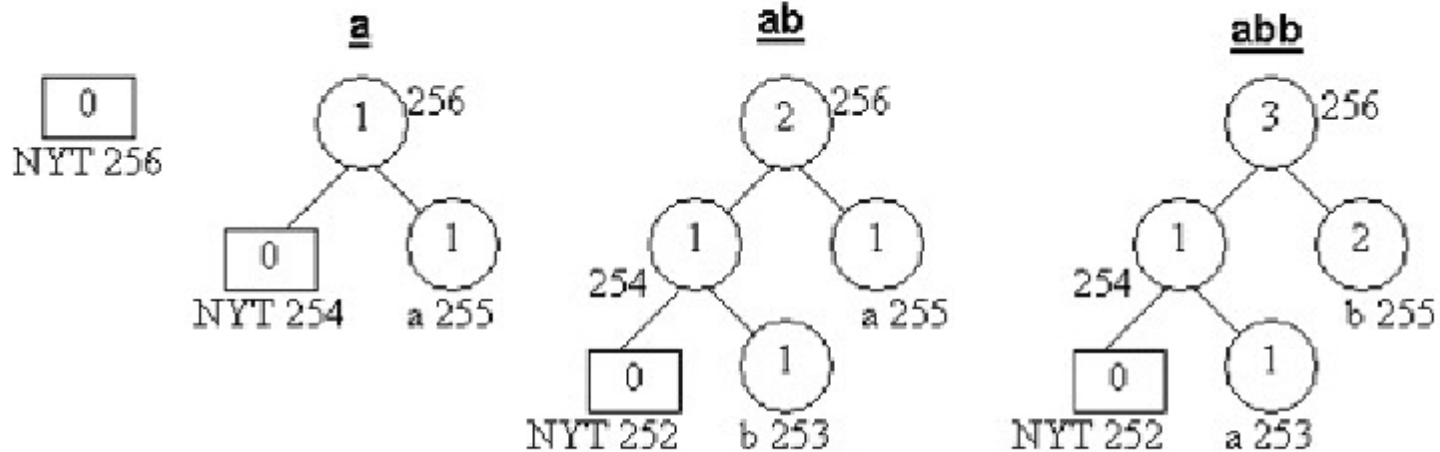
$$S - n + 1 \le T \le 2S + t - 2n + 1$$

- At worst then, Vitter's adaptive method may transmit one more bit per codeword than the static Huffman method
- Empirically, algorithm V slightly outperforms algorithm FGK

Adaptive Huffman

abbbbba: 011000010110001001100010011000100110001001100010011000010

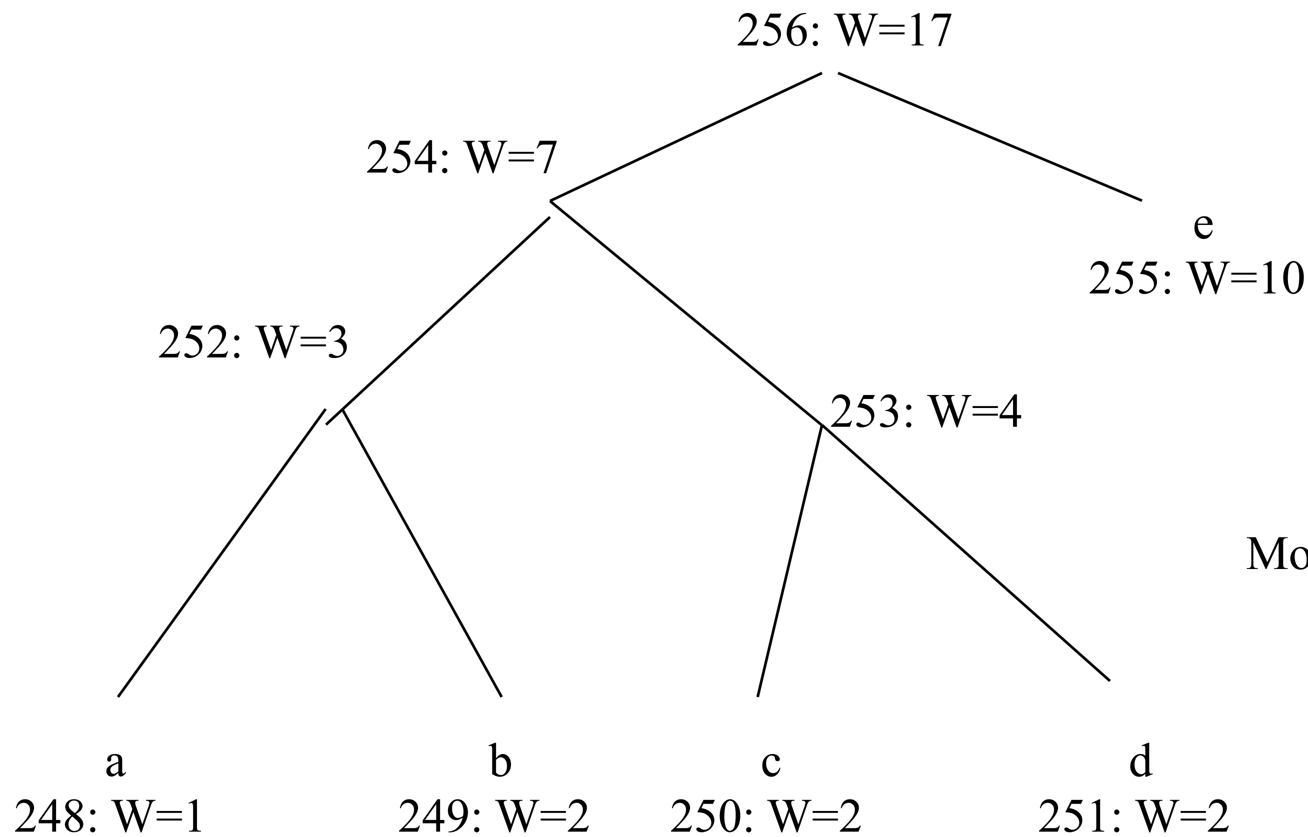
abbbbba: 01100001**0**01100010**0111101**



a: 01100001
b: 01100010

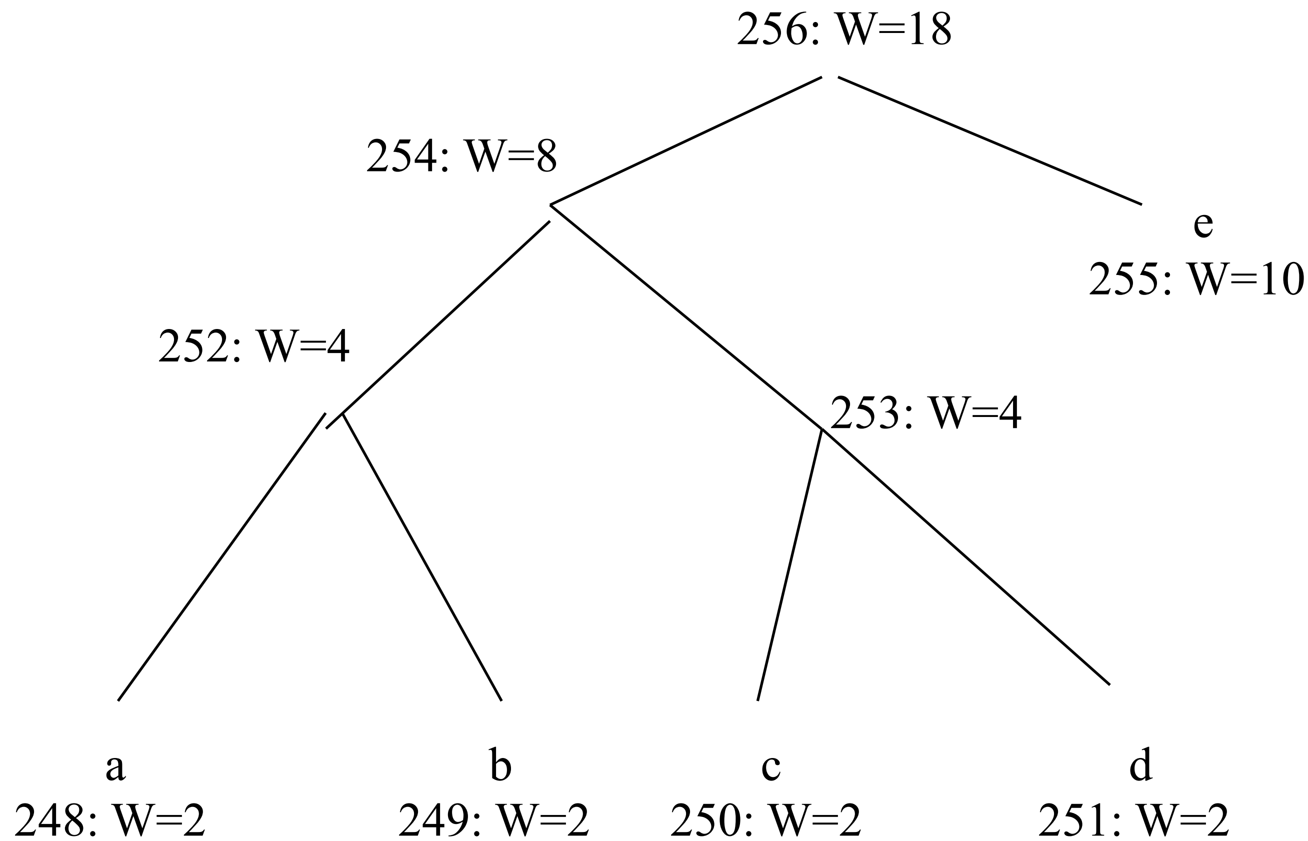
Modified from Wikipedia

More example

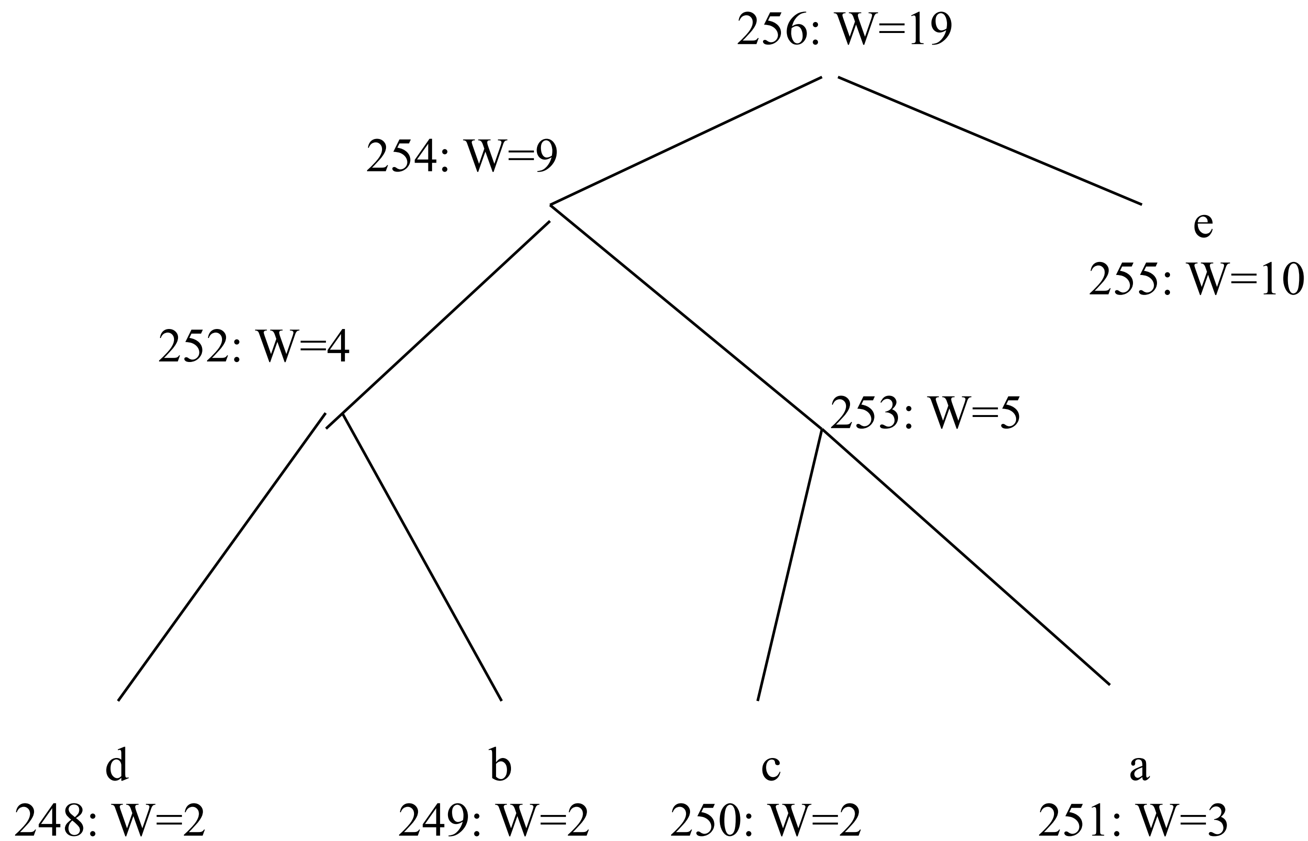


More aaaa…. coming

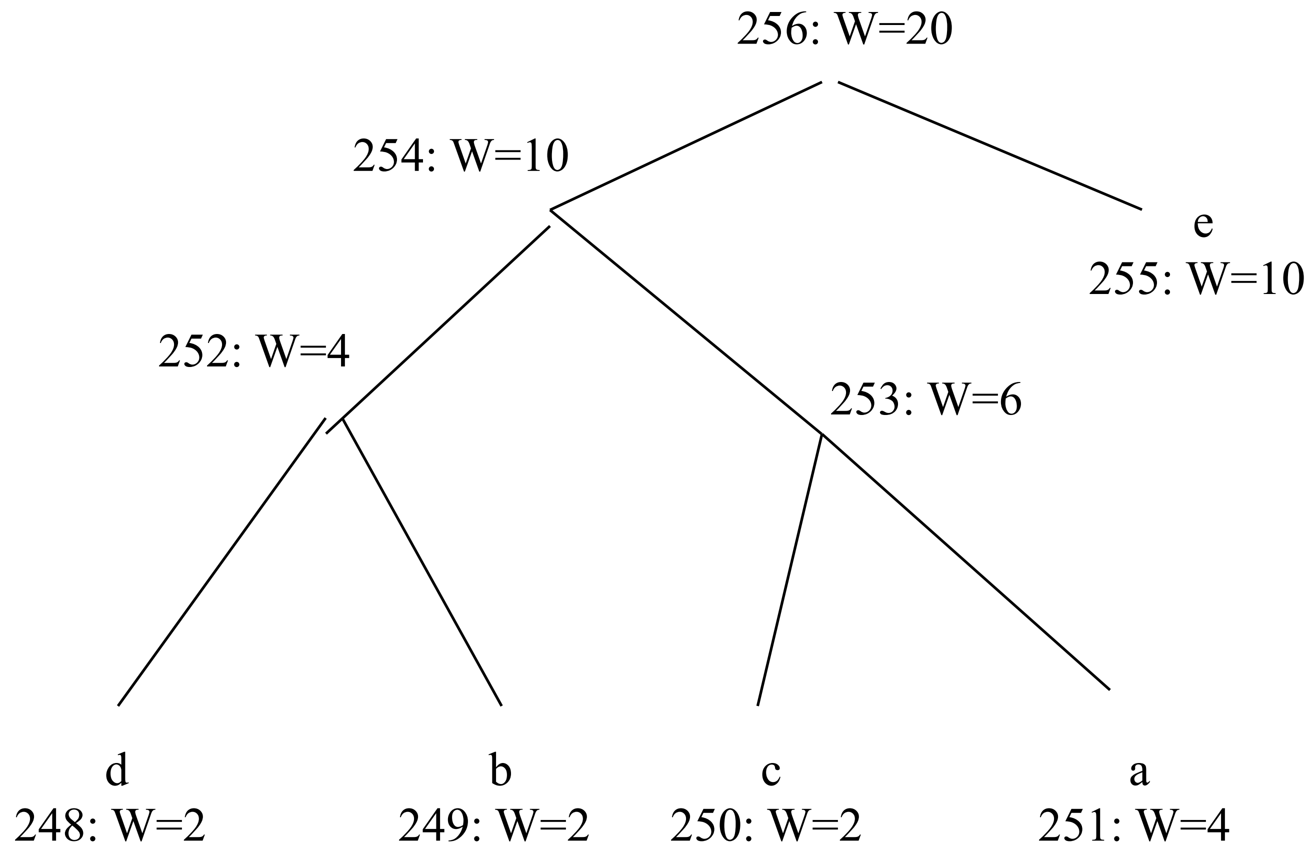
More example



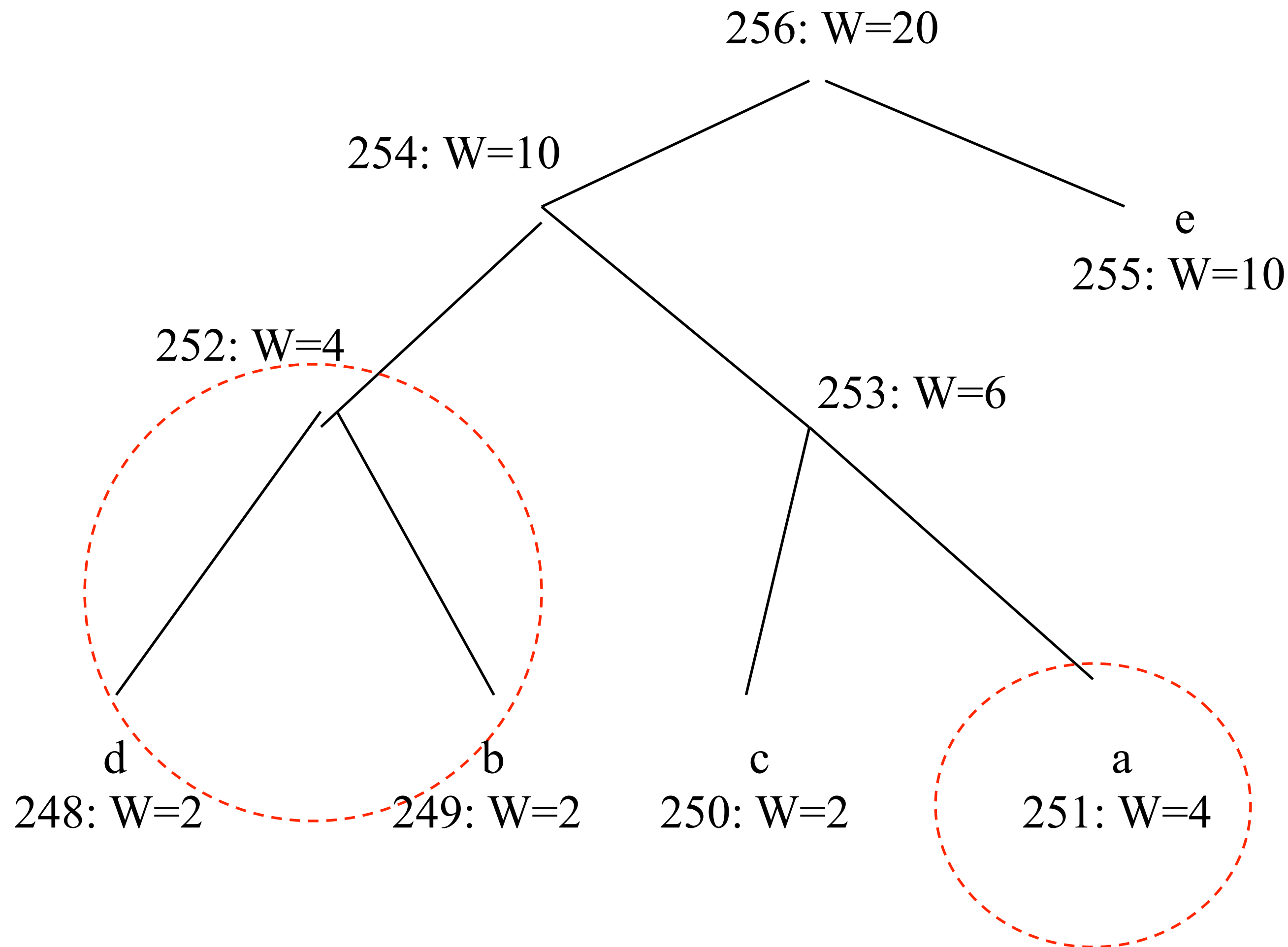
More example



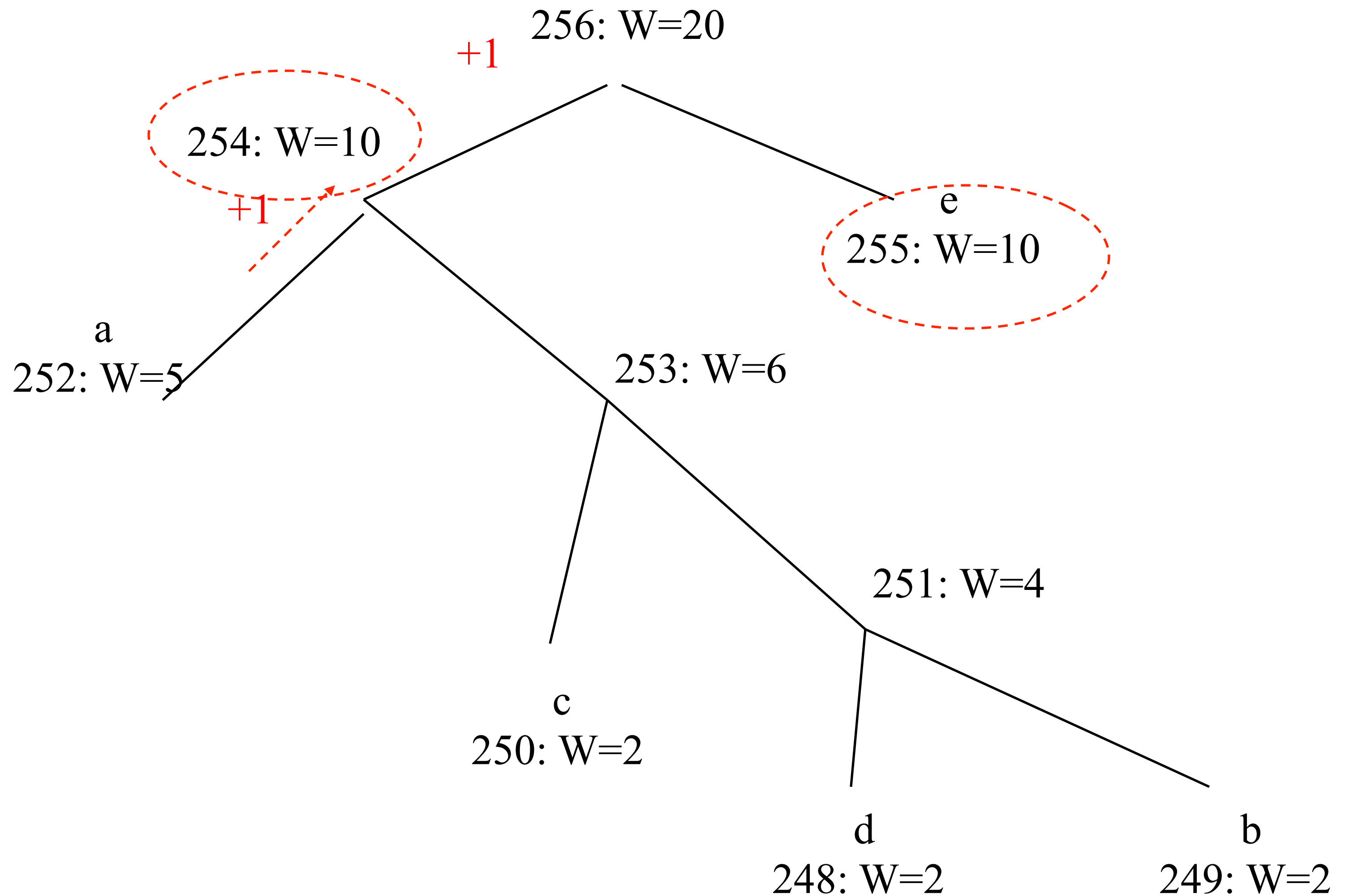
More example



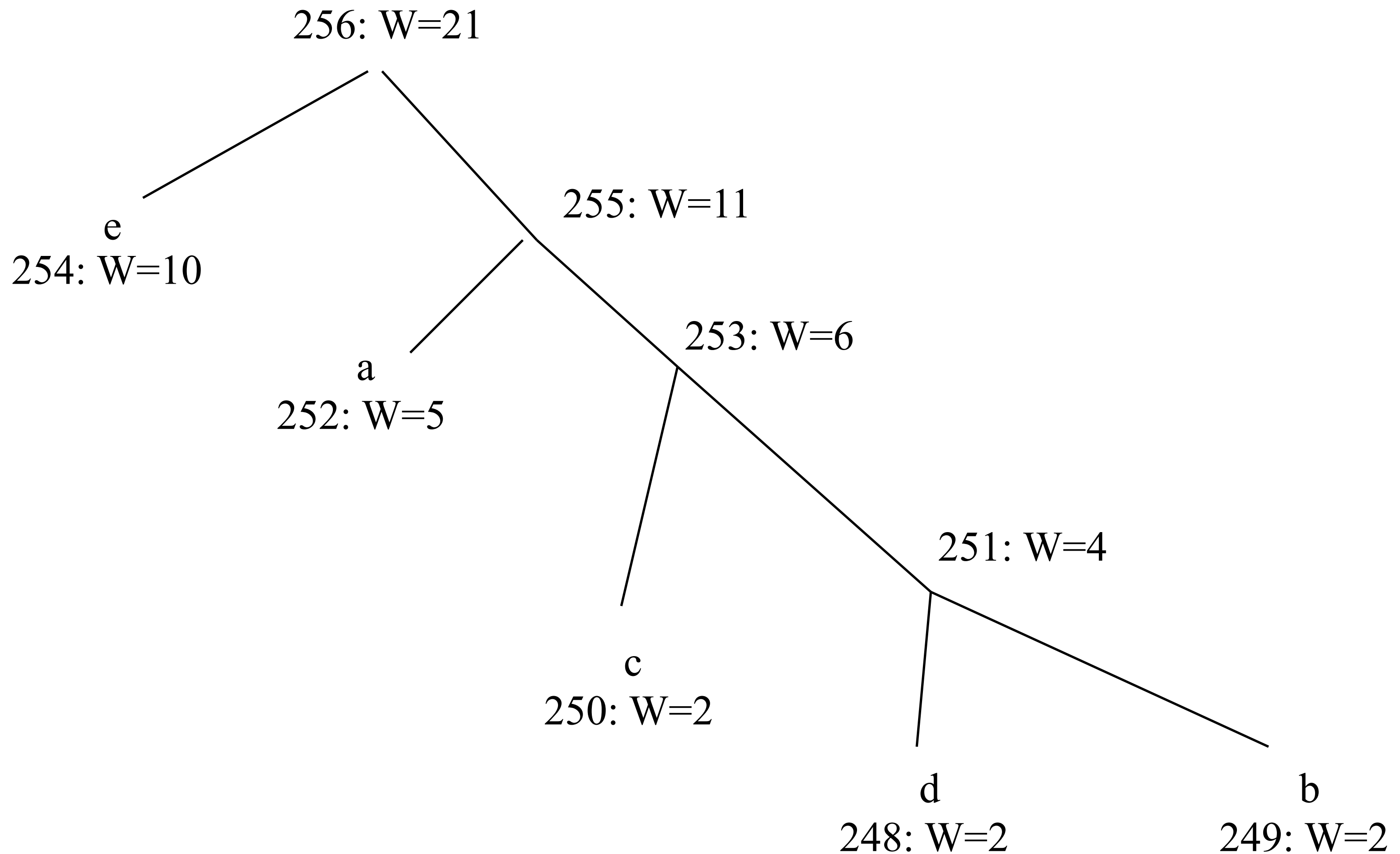
More example



More example



More example



Compared with Static Huffman

- Dynamic and can offer better compression (cf. Vitter’s experiments)
  - i.e., the tree can be smaller (hence shorter the code) before the whole bitstream is received.
- Works when prior stat is unavailable
- Saves symbol table overhead (cf. Vitter’s expt)

Assignment 1

- **Size Determination of Huffman & LZW Encoded File**
  - Your task in this assignment is to implement a C program that determines the size of a Huffman encoded (static & adaptive) file or LZW encoded file when a UTF-8 encoded file is given as input to your program.
  - You should detail the sizes
    - i.e. code + tree

Assignment 1

- It is in bytes. When the total size is not an integral number of bytes, round the remaining bits up to a byte.
- Your submitted file should be called csize.c/csize.cpp
- Your program should accept the commandline argument –sh, -ah, or -l to determine if Static Huffman, Adaptive Huffman or LZW, respectively, should be used.
- When -l is given, a number between 9 and 20 (inclusively) is expected from the commandline to specify the fixed width (in bits) of the codes. Finally, the input filename is given as the last argument in the commandline.

Assignment 1

- Your solution should read the input file as read-only (because you might not have write permission to the file) and should **not** write out any external files.
- Any solution that fails to compile on a MS C/C++ Compiler, fails to read a read-only file, or writes out external files, will receive **zero** points for the entire assignment.

Recall from Lecture 1's RLE and BWT example

rabcabcababaabacabcabcabcababaa$

aabbbbccacccrcbaaaaaaaaaabbbbba$

aab4ccac3rcba10b5a$

A simple example

Input:

#BANANAS

Excerpted from Wikipedia

All rotations

```
#BANANAS
S#BANANA
AS#BANAN
NAS#BANA
ANAS#BAN
NANAS#BA
ANANAS#B
BANANAS#
```

Sort the rows

```
#BANANAS
ANANAS#B
ANAS#BAN
AS#BANAN
BANANAS#
NANAS#BA
NAS#BANA
S#BANANA
```

Output

```
#BANANAS
ANANAS#B
ANAS#BAN
AS#BANAN
BANANAS#
NANAS#BA
NAS#BANA
S#BANANA
```

Exercise: you can try the example

rabcabcababaabacabcabcabcababaa$

aabbbbccacccrcbaaaaaaaaaabbbbba$

Now the inverse…

Input:

S

B

N

N

#

A

A

A

First add

S

B

N

N

#

A

A

A

Then sort

```
#
A
A
A
B
N
N
S
```

Add again

```
S#
BA
NA
NA
#B
AN
AN
AS
```

Then sort

```
#B
AN
AN
AS
BA
NA
NA
S#
```

Then add

```
S#B
BAN
NAN
NAS
#BA
ANA
ANA
AS#
```

Then sort

#BA
ANA
ANA
AS#
BAN
NAN
NAS
S#B

Then add

```
S#BA
BANA
NANA
NAS#
#BAN
ANAN
ANAS
AS#B
```

Then sort

```
#BAN
ANAN
ANAS
AS#B
BANA
NANA
NAS#
S#BA
```

Then add

```
S#BAN
BANAN
NANAS
NAS#B
#BANA
ANANA
ANAS#
AS#BA
```

Then sort

```
#BANA
ANANA
ANAS#
AS#BA
BANAN
NANAS
NAS#B
S#BAN
```

Then add

```
S#BANA
BANANA
NANAS#
NAS#BA
#BANAN
ANANAS
ANAS#B
AS#BAN
```

Then sort

```
#BANAN
ANANAS
ANAS#B
AS#BAN
BANANA
NANAS#
NAS#BA
S#BANA
```

Then add

```
S#BANAN
BANANAS
NANAS#B
NAS#BAN
#BANANA
ANANAS#
ANAS#BA
AS#BANA
```

Then sort

```
#BANANA
ANANAS#
ANAS#BA
AS#BANA
BANANAS
NANAS#B
NAS#BAN
S#BANAN
```

Then add

```
S#BANANA
BANANAS#
NANAS#BA
NAS#BANA
#BANANAS
ANANAS#B
ANAS#BAN
AS#BANAN
```

Then sort (?)

```
#BANANAS
ANANAS#B
ANAS#BAN
AS#BANAN
BANANAS#
NANAS#BA
NAS#BANA
S#BANANA
```

BWT(S)

function BWT (string s)

create a table, rows are all possible rotations of s

sort rows alphabetically

return (last column of the table)

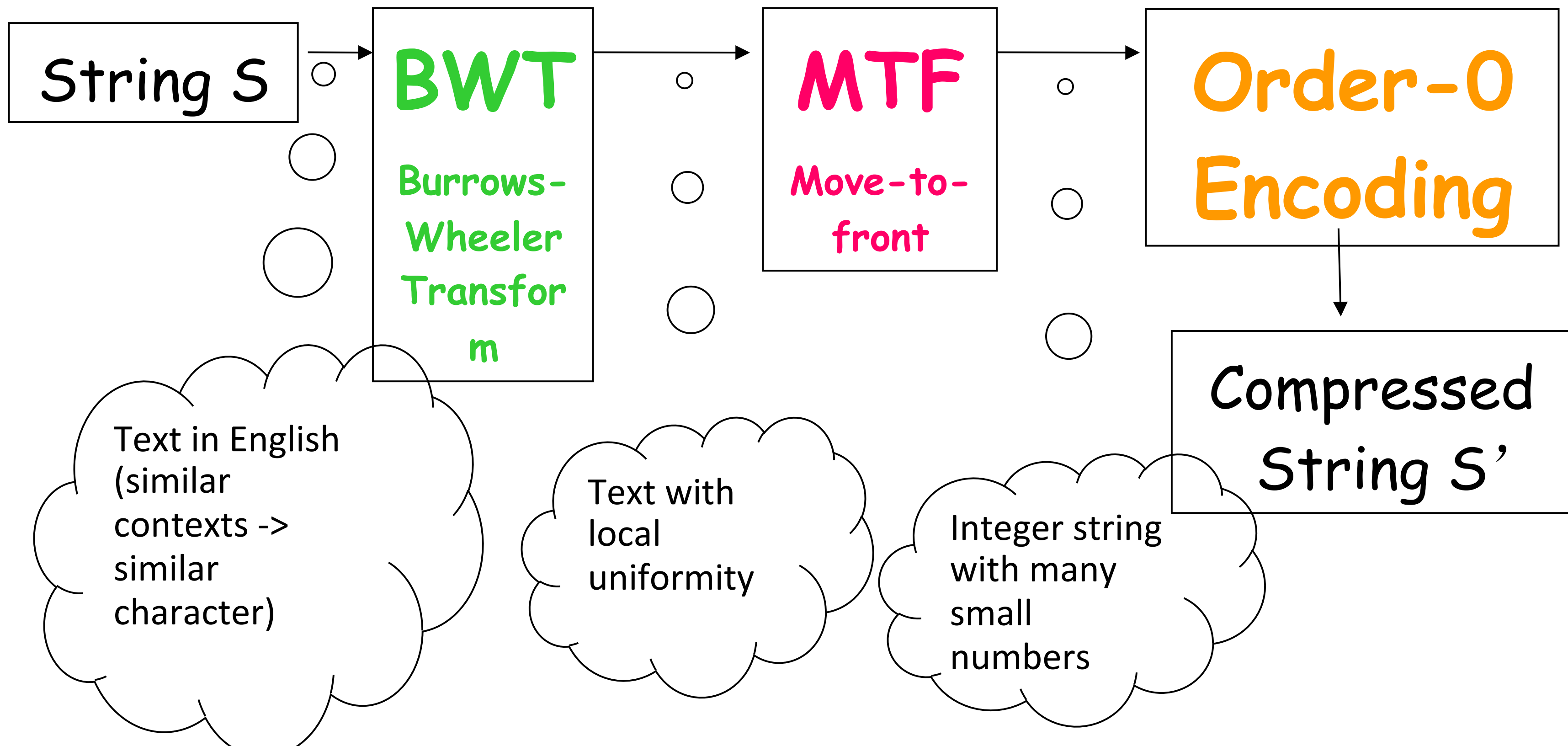
InverseBWT(S)

```
function inverseBWT (string s)
  create empty table

  repeat length(s) times
     insert s as a column of table before first
          column of the table    // first insert creates
                                    first column
     sort rows of the table alphabetically
  return (row that ends with the 'EOF' character)
```

BW0

The Main Burrows-Wheeler Compression Algorithm:



The BWT

S string with context-regularity

mississippi

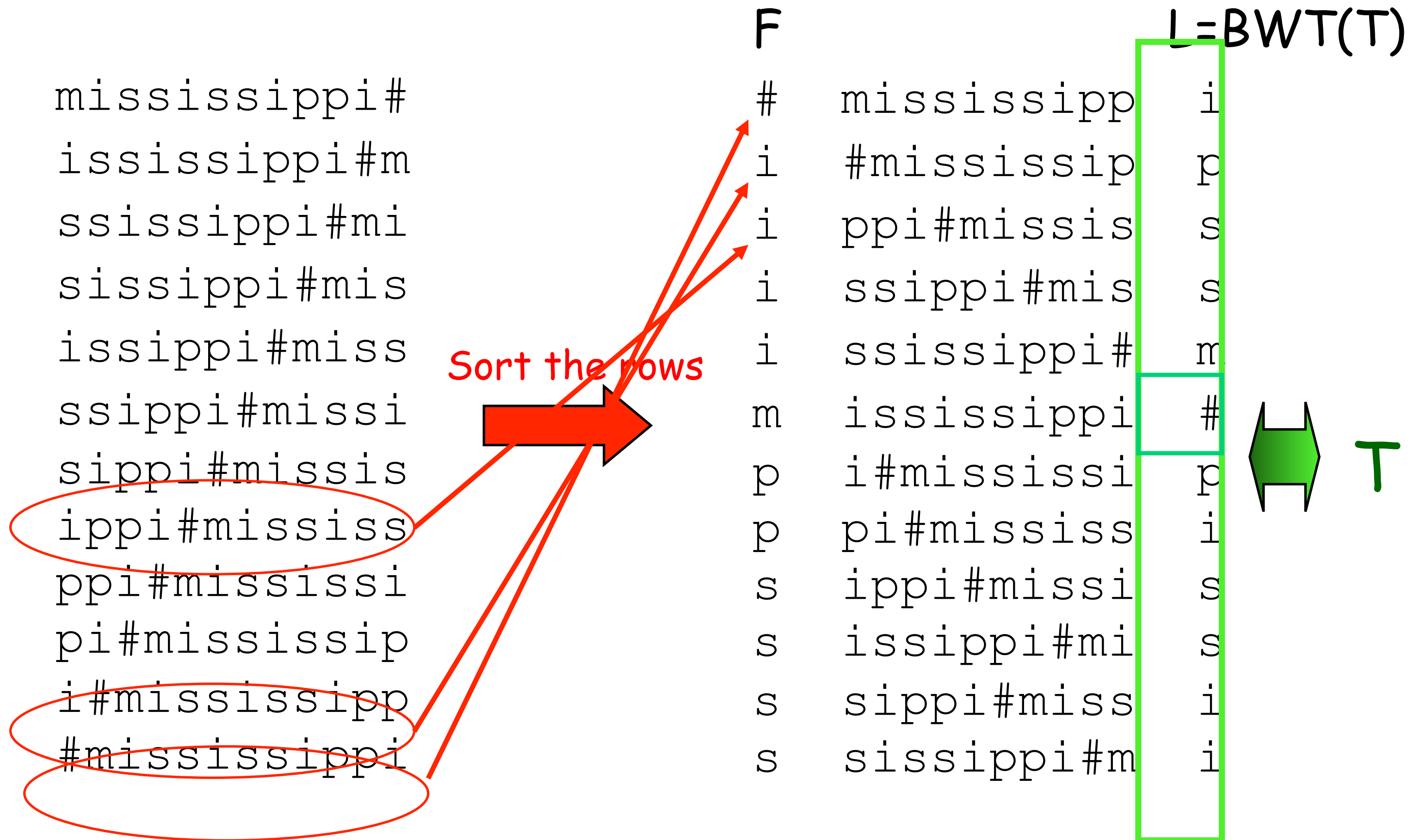
BWT

ipssmpissii

$\hat{S}$ string with spikes (close repetitions)

The BWT

T = mississippi#



BWT sorts the characters by their post-context

Move to Front (MTF)

- Reduce entropy based on local frequency correlation
- Usually used for BWT before an entropy-encoding step
- Author and detail:
  - http://www.arturocampos.com/ac_mtf.html

Move To Front

- By Bentley, Sleator, Tarjan and Wei（’86)

$\hat{S}$ string with spikes (close repetitions)

ipssmpissii

0,0,0,0,0,2,4,3,0,1,0

S' integer string with small numbers

Move to Front

<u>**a**</u>bracadabra		<u>**a**</u>,b,r,c,d

Move to Front

abracadabra		**a**,b,r,c,d
a**b**racadabra	0	a,**b**,r,c,d

Move to Front

abracadabra		**a**,b,r,c,d
a**b**racadabra	0	a,**b**,r,c,d
ab**r**acadabra	0,1	b,a,**r**,c,d

Move to Front

abracadabra		**a**,b,r,c,d
a**b**racadabra	0	a,**b**,r,c,d
ab**r**acadabra	0,1	b,a,**r**,c,d
abr**a**cadabra	0,1,2	r,b,**a**,c,d

Move to Front

abracadabra		**a**,b,r,c,d
a**b**racadabra	0	a,**b**,r,c,d
ab**r**acadabra	0,1	b,a,**r**,c,d
abr**a**cadabra	0,1,2	r,b,**a**,c,d
abra**c**adabra	0,1,2,2	a,r,b,**c**,d

Move to Front

abracadabra		**a**,b,r,c,d
a**b**racadabra	0	a,**b**,r,c,d
ab**r**acadabra	0,1	b,a,**r**,c,d
abr**a**cadabra	0,1,2	r,b,**a**,c,d
abra**c**adabra	0,1,2,2	a,r,b,**c**,d
abrac**a**dabra	0,1,2,2,3	c,**a**,r,b,d

Move to Front

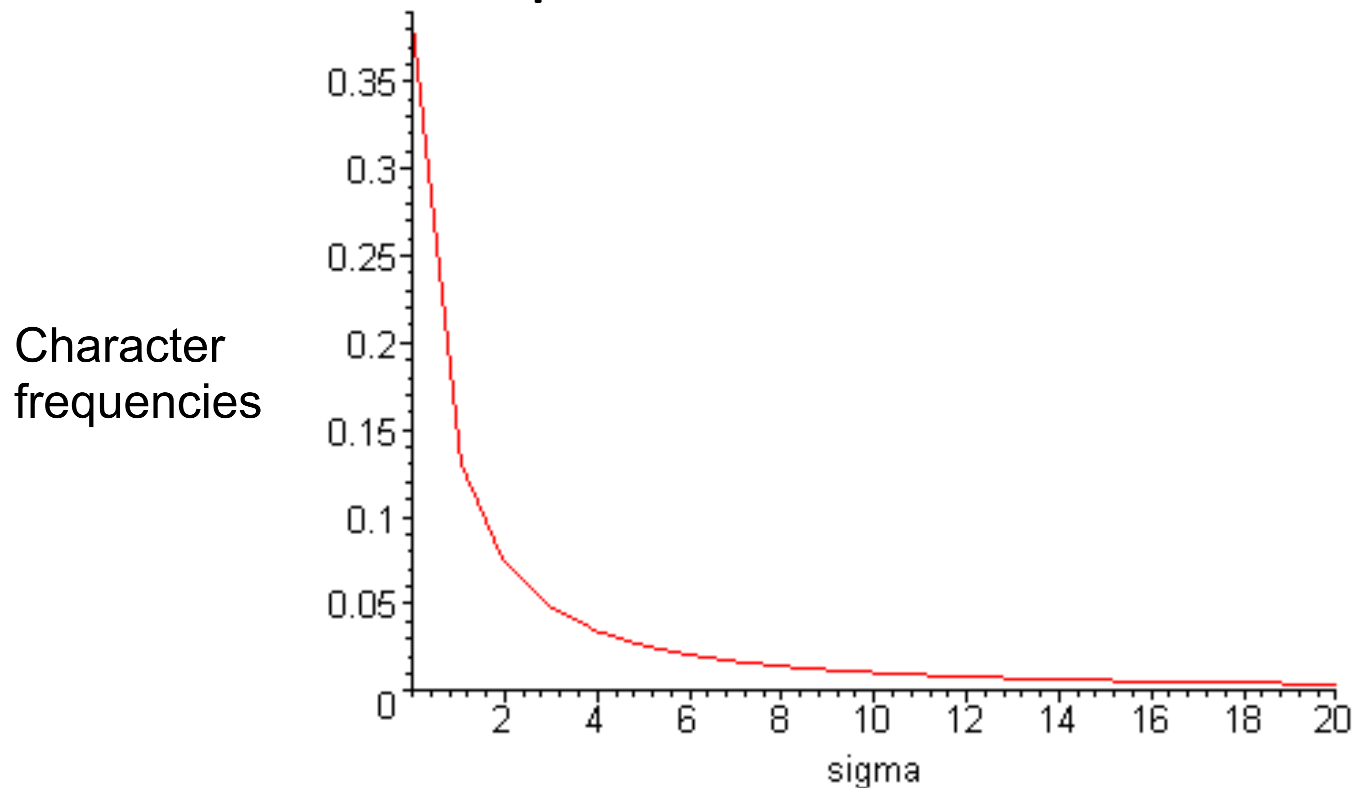
abracadabra		**a**,b,r,c,d
a**b**racadabra	0	a,**b**,r,c,d
ab**r**acadabra	0,1	b,a,**r**,c,d
abr**a**cadabra	0,1,2	r,b,**a**,c,d
abra**c**adabra	0,1,2,2	a,r,b,**c**,d
abrac**a**dabra	0,1,2,2,3	c,**a**,r,b,d
abraca**d**abra	0,1,2,2,3,1	a,c,r,b,**d**

Move to Front

abracadabra		**a**,b,r,c,d
a**b**racadabra	0	a,**b**,r,c,d
ab**r**acadabra	0,1	b,a,**r**,c,d
abr**a**cadabra	0,1,2	r,b,**a**,c,d
abra**c**adabra	0,1,2,2	a,r,b,**c**,d
abrac**a**dabra	0,1,2,2,3	c,**a**,r,b,d
abraca**d**abra	0,1,2,2,3,1	a,c,r,b,**d**
abracadabra	0,1,2,2,3,1,4,1,4,4,2	

After MTF

- Now we have a string with small numbers: lots of 0s, many 1s, …
- Skewed frequencies: Run Arithmetic!



Example: abaabacad

Symbol	Code	List
a	0	abcde…..
b	1	bacde…..
a	1	abcde…..
a	0	abcde…..
b	1	bacde…..
a	1	abcde…..
c	2	cabde…..
a	1	acbde…..
d	3	dacbe…..

To transform a general file, the list has 256 ASCII symbols.

BWT compressor vs ZIP

ZIP (i.e., LZW based)

BWT+RLE+MTF+AC

File Name	Raw Size	PKZIP Size	PKZIP Bits/Byte	BWT Size	BWT Bits/Byte
bib	111,261	35,821	2.58	29,567	2.13
book1	768,771	315,999	3.29	275,831	2.87
book2	610,856	209,061	2.74	186,592	2.44
geo	102,400	68,917	5.38	62,120	4.85
news	377,109	146,010	3.10	134,174	2.85
obj1	21,504	10,311	3.84	10,857	4.04
obj2	246,814	81,846	2.65	81,948	2.66

From http://marknelson.us/1996/09/01/bwt/

Other ways to reverse BWT

Consider L=BWT(S) is composed of the symbols $V_0 \ldots V_{N-1}$, the transformed string may be parsed to obtain:

- The number of symbols in the substring $V_0 \ldots V_{i-1}$ that are identical to V_i.
- For each unique symbol, V_i, in L, the number of symbols that are lexicographically less than that symbol.

Example

Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

???????]

Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

??????A]

Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

?????NA]

Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

????ANA]

Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

???NANA]

Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

??ANANA]

Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

?BANANA]

Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

[BANANA]

Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

[BANANA]

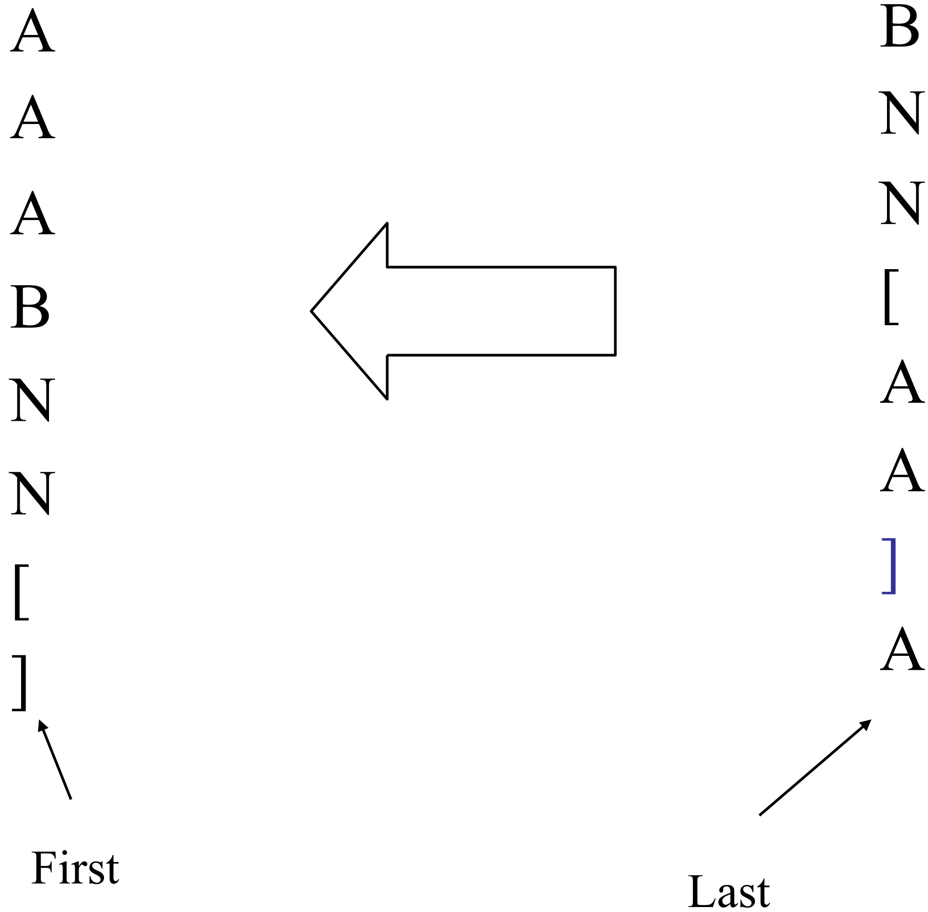
Position	Symbol	#Matching
0	B	0
1	N	0
2	N	1
3	[	0
4	A	0
5	A	1
6	]	0
7	A	2

Occ / Rank

Symbol	#LessThan
A	0
B	3
N	4
[	6
]	7

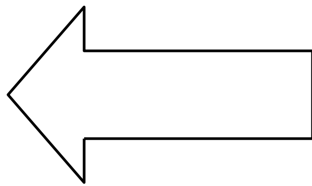
C[]

An illustration



]

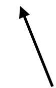
A
A
A
B
N
N
[
]



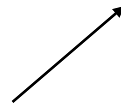
B
N
N
[
A
A
]
A

If we know:

'[' is the first char



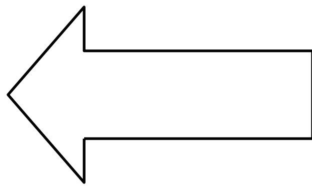
First



Last

]

A
A
A
B
N
N
[
]



B
N
N
[
A
A
]
A

Or : #[BANANA]

is smaller than others

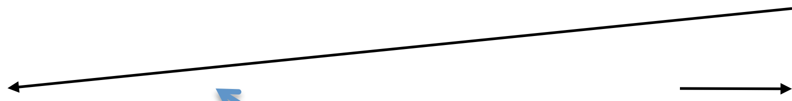
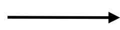
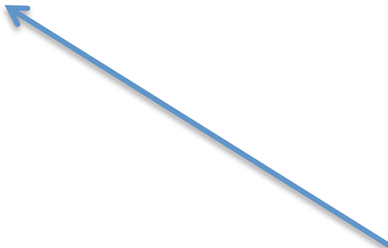
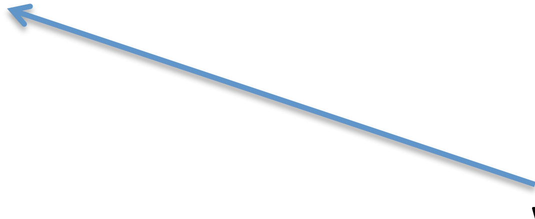
First

Last

A]

A
A
A
B
N
N
[
]

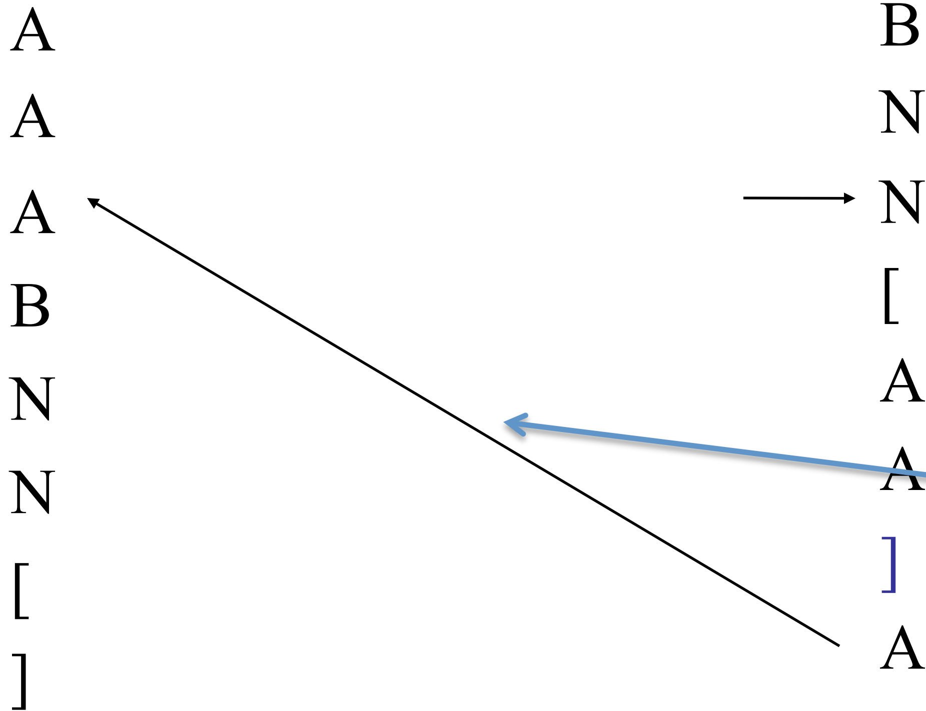
B
N
N
[
A
A
]
A

Why?

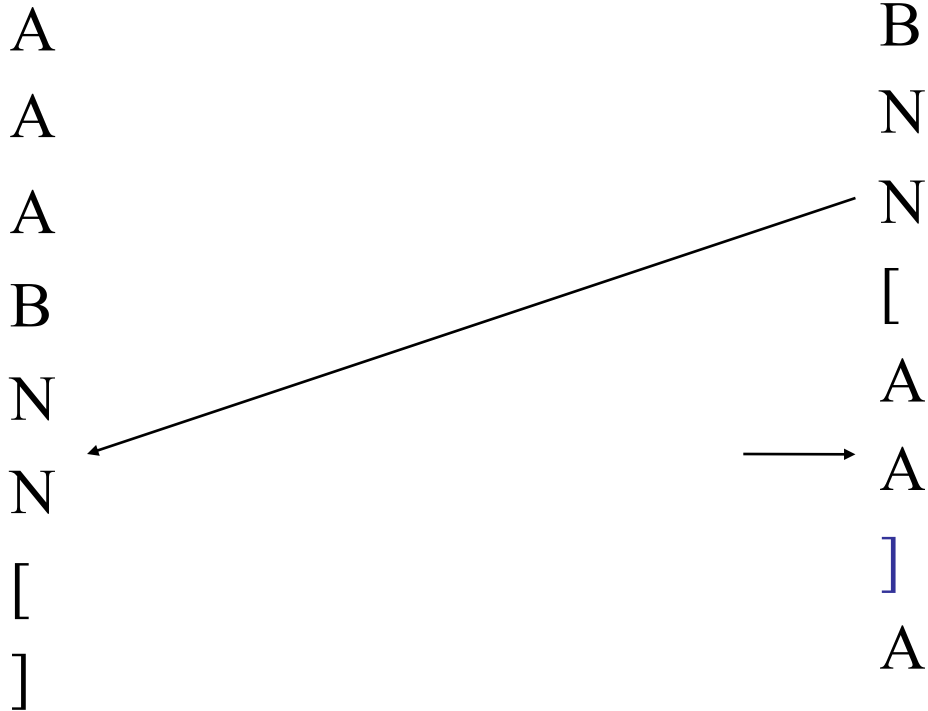
Why?

NA]

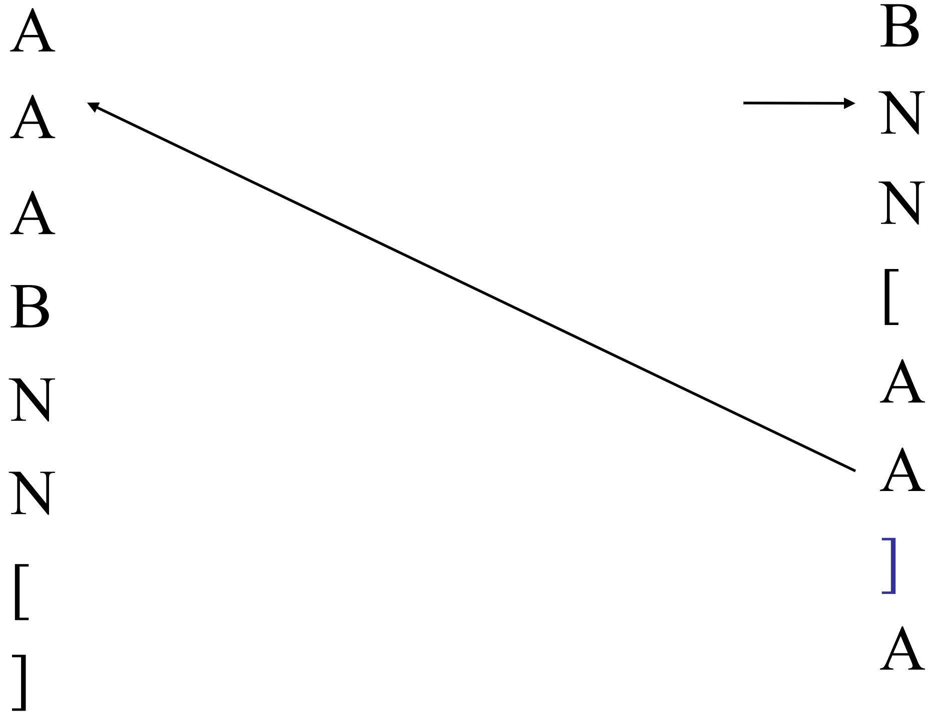


How？
2 info
Why?

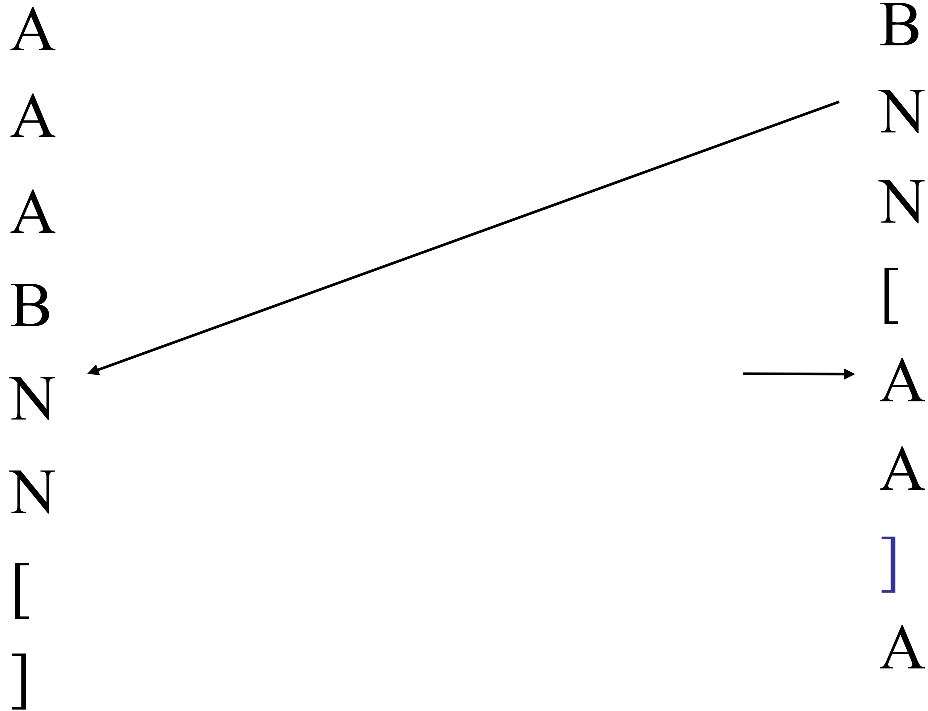
ANA]



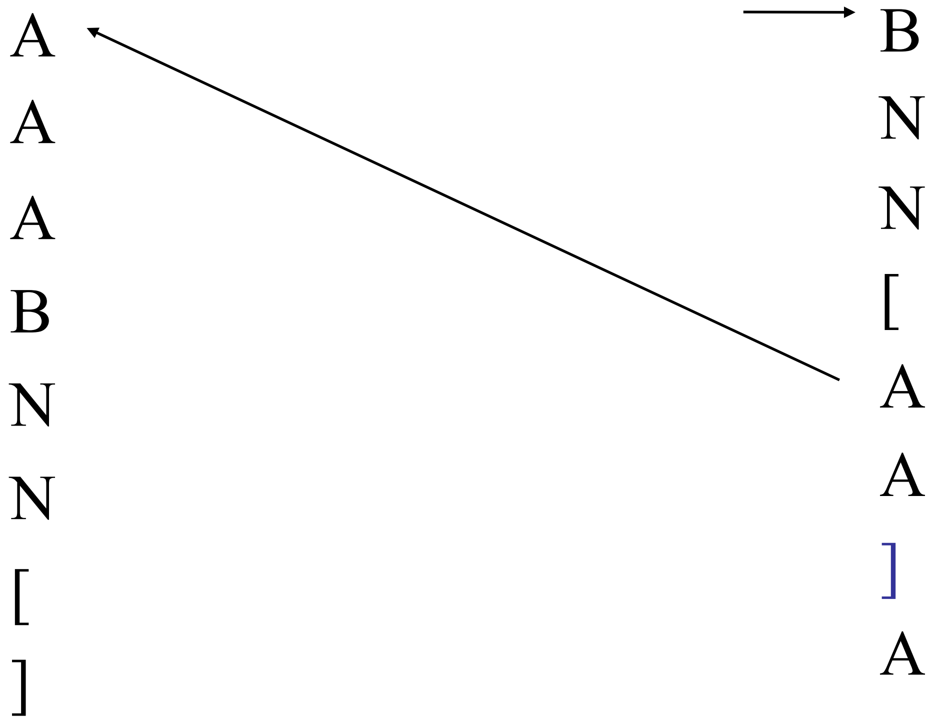
NANA]



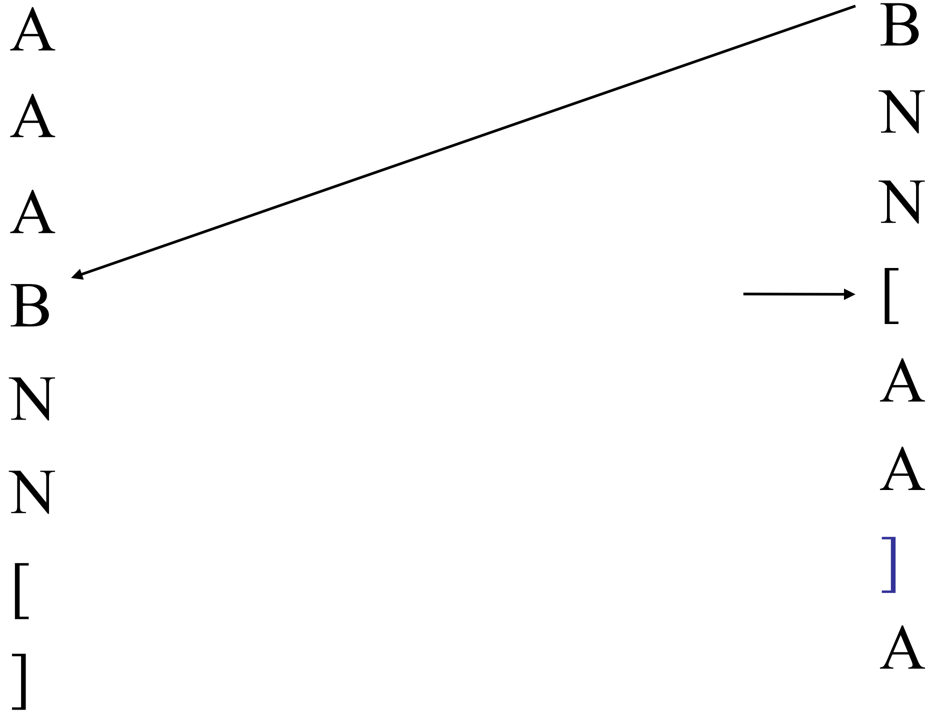
ANANA]



BANANA]



[BANANA]



Dynamic BWT ?

Instead of reconstructing BWT, local reordering from the original BWT.

Details:

Salson M, Lecroq T, Léonard M and Mouchard L (2009). "A Four-Stage Algorithm for Updating a Burrows–Wheeler Transform". Theoretical Computer Science 410 (43): 4350.