COMP9319 Web Data Compression and Search

LZW,

Adaptive Huffman,

(live lecture)

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.

LZW Notes
(Handling the special case)

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			

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

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	a	256	ab

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

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	a	256	ab
b	a	b	257	ba

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

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	a	256	ab
b	a	b	257	ba
a	b			

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

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	a	256	ab
b	a	b	257	ba
a	b			
ab	a	256	258	aba

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

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	a	256	ab
b	a	b	257	ba
a	b			
ab	a	256	258	aba
a	b			

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

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	a	256	ab
b	a	b	257	ba
a	b			
ab	a	256	258	aba
a	b			
ab	a			

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

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	a	256	ab
b	a	b	257	ba
a	b			
ab	a	256	258	aba
a	b			
ab	a			
aba	b	258	259	abab

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

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	a	256	ab
b	a	b	257	ba
a	b			
ab	a	256	258	aba
a	b			
ab	a			
aba	b	258	259	abab
b	EOF	b		

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

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	**a**	256	ab
b	a	**b**	257	ba
a	b			
ab	a	**256**	258	aba
a	b			
ab	a			
aba	b	**258**	259	abab
b	EOF	**b**		

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

Decoding

Input: ab<256><258>b

```
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];
```

p	c	output	Dict [index] index	symbol
	a	a		

Decoding

Input: ab<256><258>b

```
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];
```

p	c	output	Dict [index] index	Dict [index] symbol
	a	a		
a	b	b	256	ab

Decoding

Input: ab<256><258>b

```
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];
```

p	c	output	Dict [index] index	Dict [index] symbol
	a	a		
a	b	b	256	ab
b	<256>	ab	257	ba

Decoding

Input: ab<256><258>b

```
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];
```

p	c	output	Dict [index] index	Dict [index] symbol
	a	a		
a	b	b	256	ab
b	<256>	ab	257	ba
<256>	<258>	WHAT???		

Encoding

Input: abababab

Dict [index]

p	c	output	index	symbol
nil	a			
a	b	a	256	ab
b	a	b	257	ba
a	b			
ab	**a**	256	258	aba
a	b			
ab	a			
aba	b	258	259	abab
b	EOF	b		

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

Decoding

Input: ab<256><258>b

```
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];
```

p	c	output	Dict [index] index	Dict [index] symbol
	a	a		
a	b	b	256	ab
b	<256>	ab	257	ba
<256>	<258>	**ab**a	258	aba

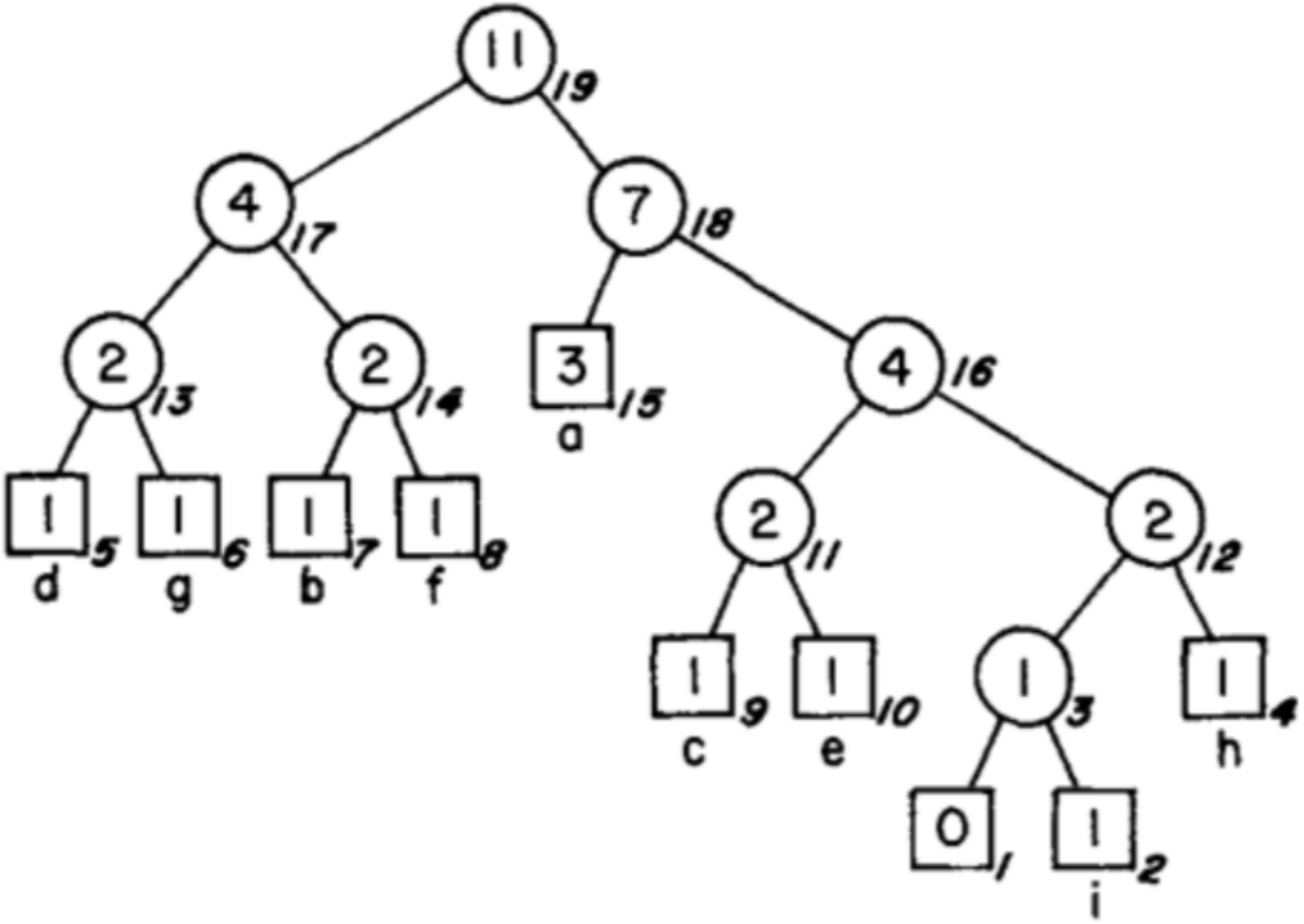
Decoding

Input: ab<256><258>b

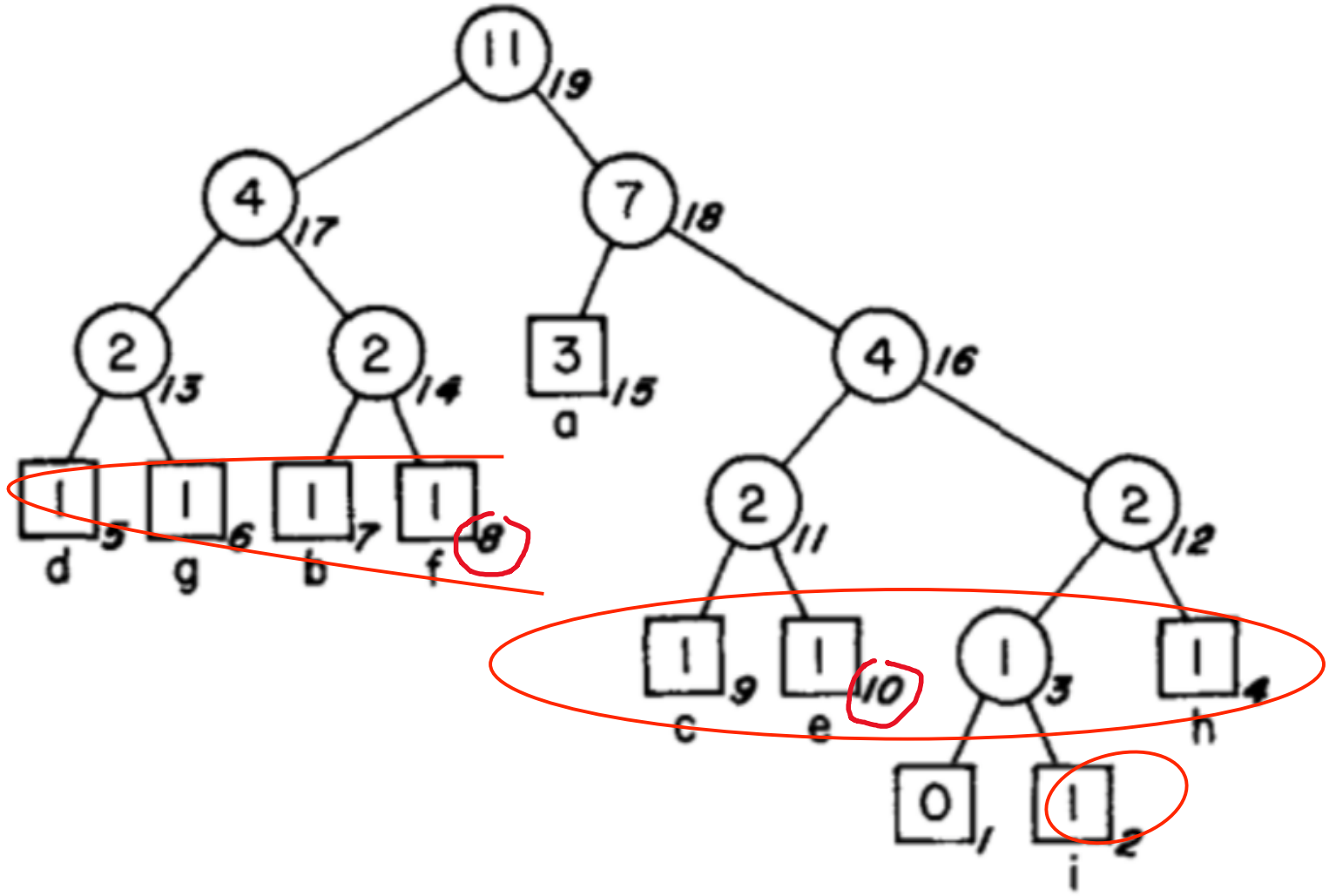
```
read(c);   // c is likely to be > 8 bits
output c;
p = c;
while read(c):
    output Dict[c];
    add p + Dict[c][0] to Dict;
    p = Dict[c];
```

p	c	output	Dict [index] index	Dict [index] symbol
	a	a		
a	b	b	256	ab
b	<256>	ab	257	ba
<256>	<258>	**ab**a	258	aba
<258>	b	b	259	abab

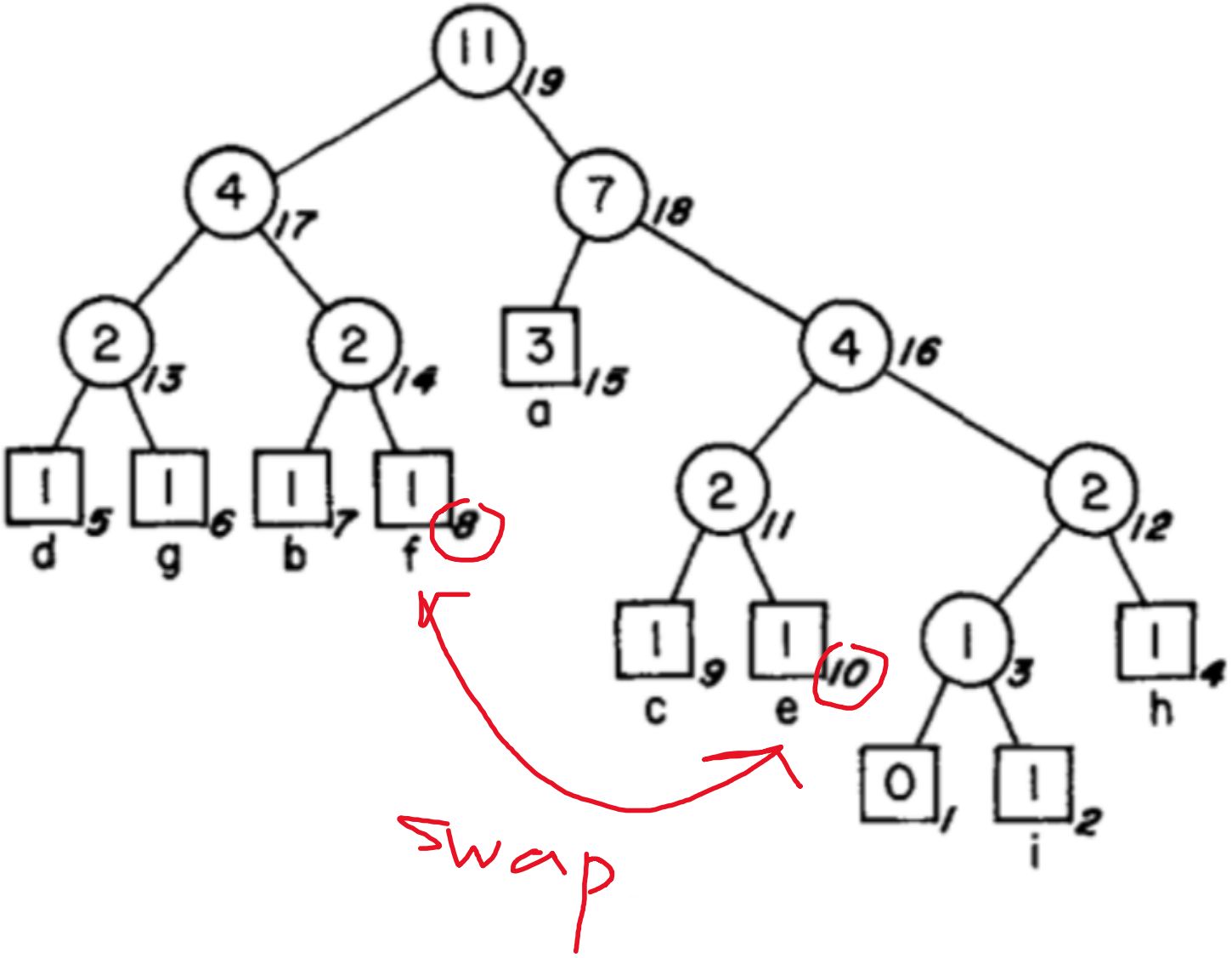
Adaptive Huffman (FGK)



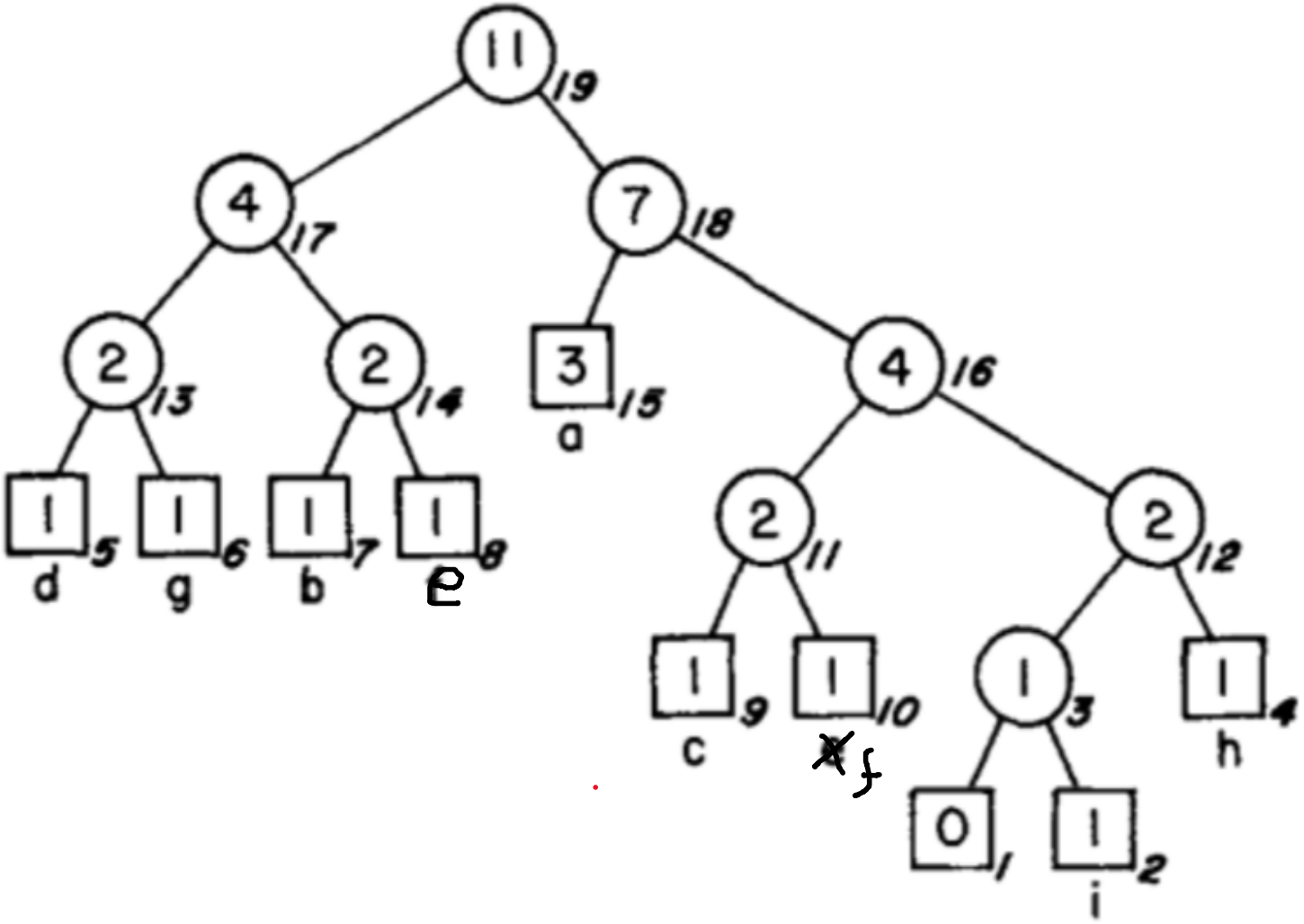
Adaptive Huffman (FGK)



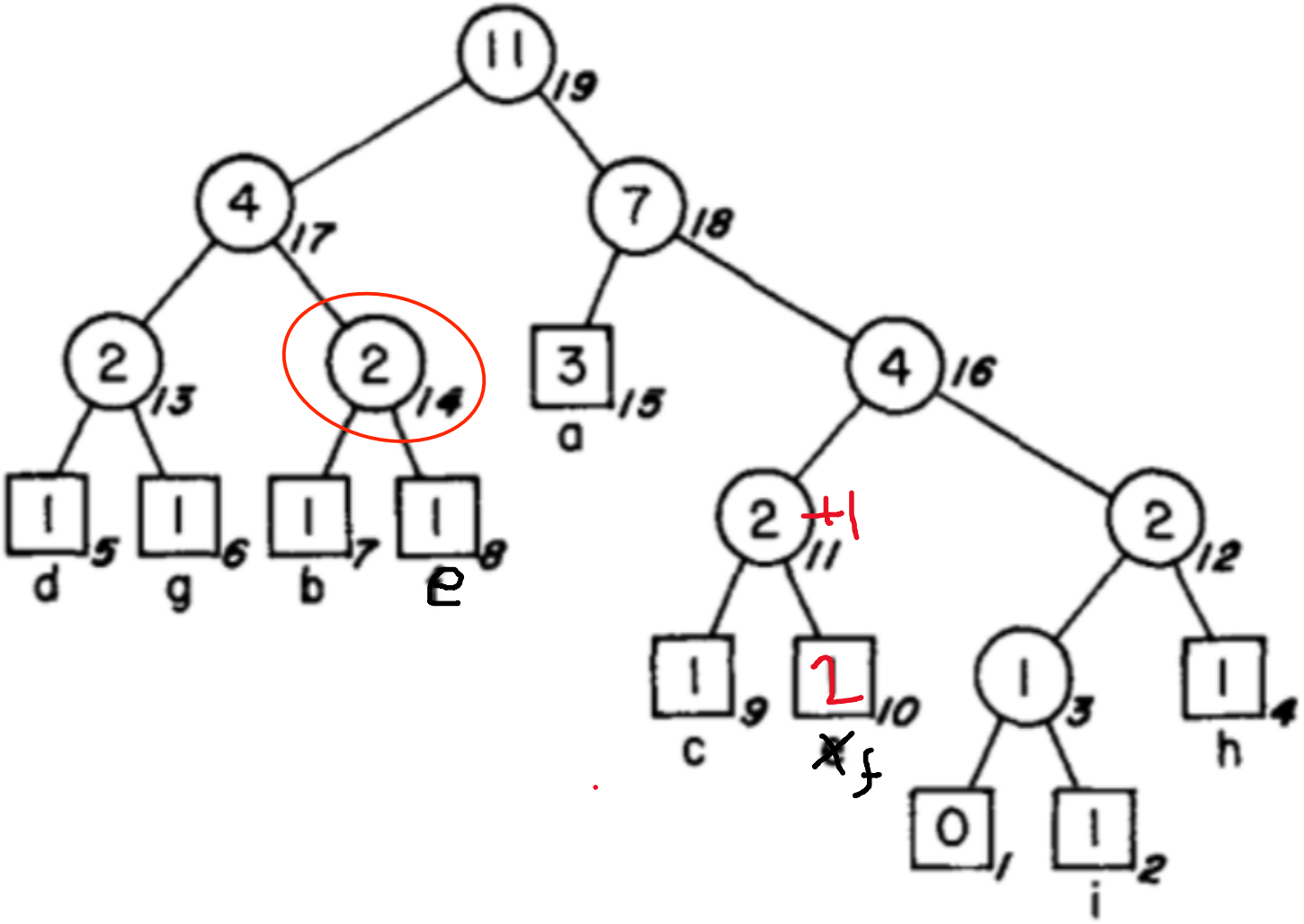
Adaptive Huffman (FGK)



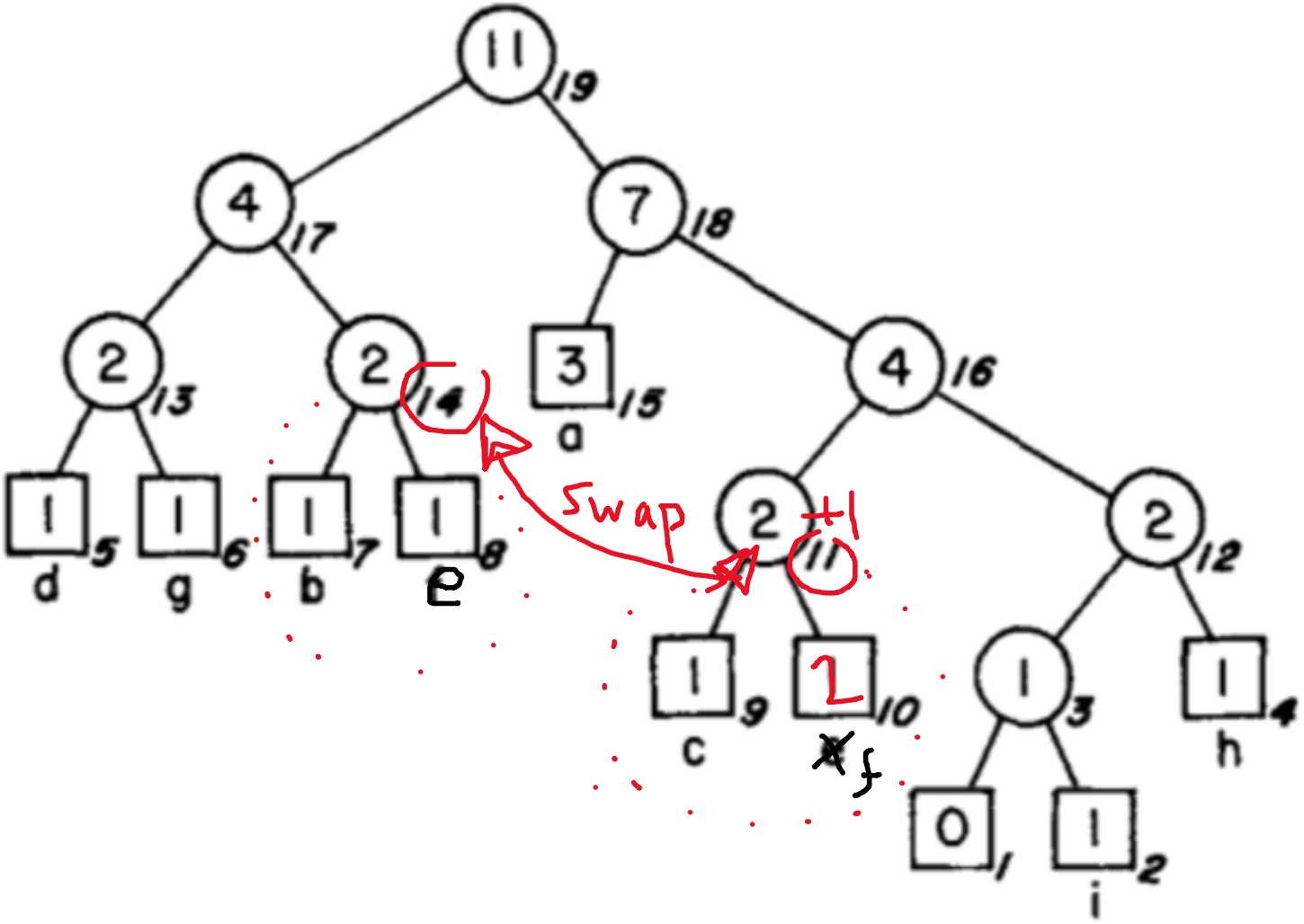
Adaptive Huffman (FGK)



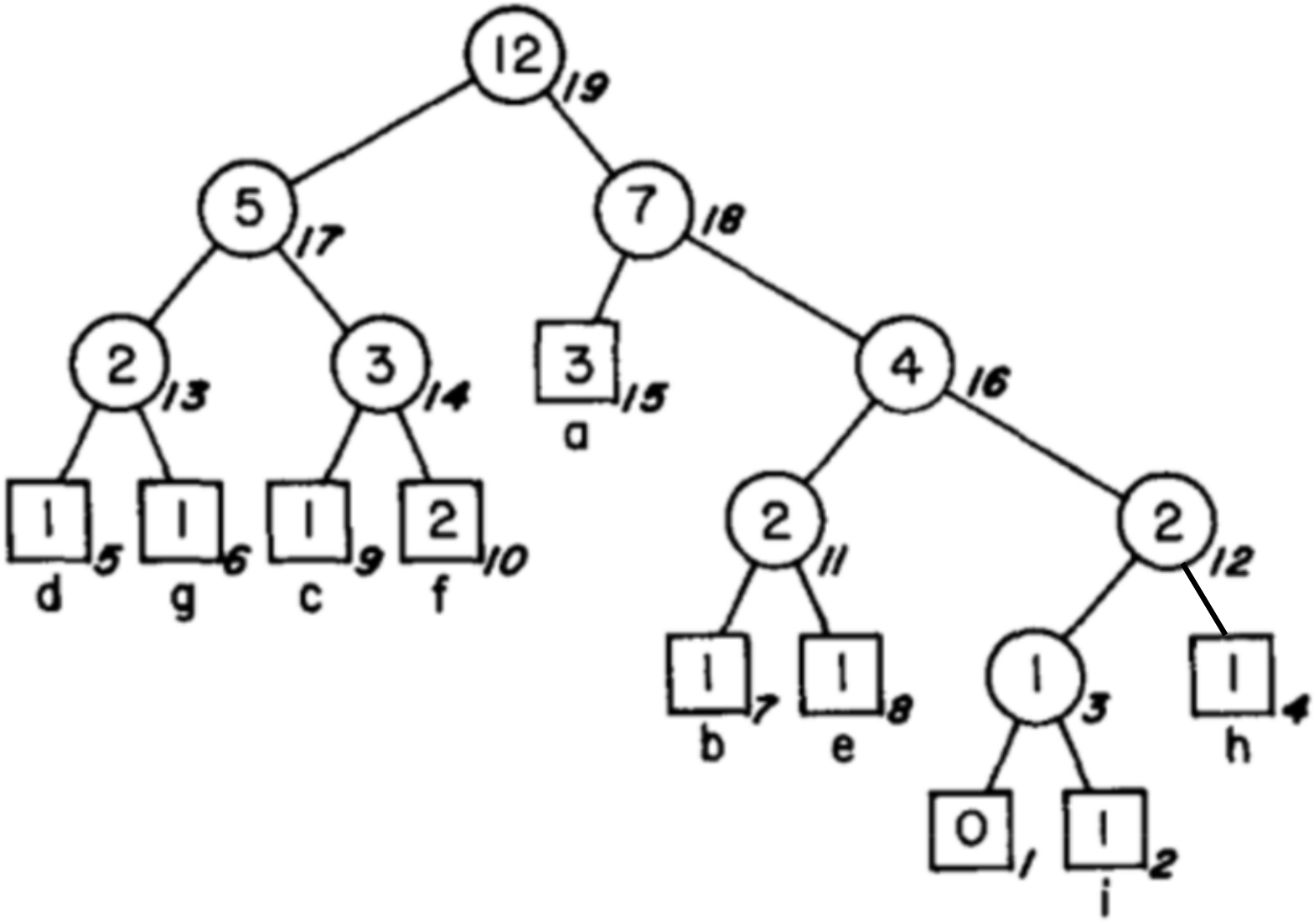
Adaptive Huffman (FGK)



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.

In order to maintain the invariant (*), we must keep separate blocks for internal and leaf nodes.

Adaptive Huffman (Vitter'87)

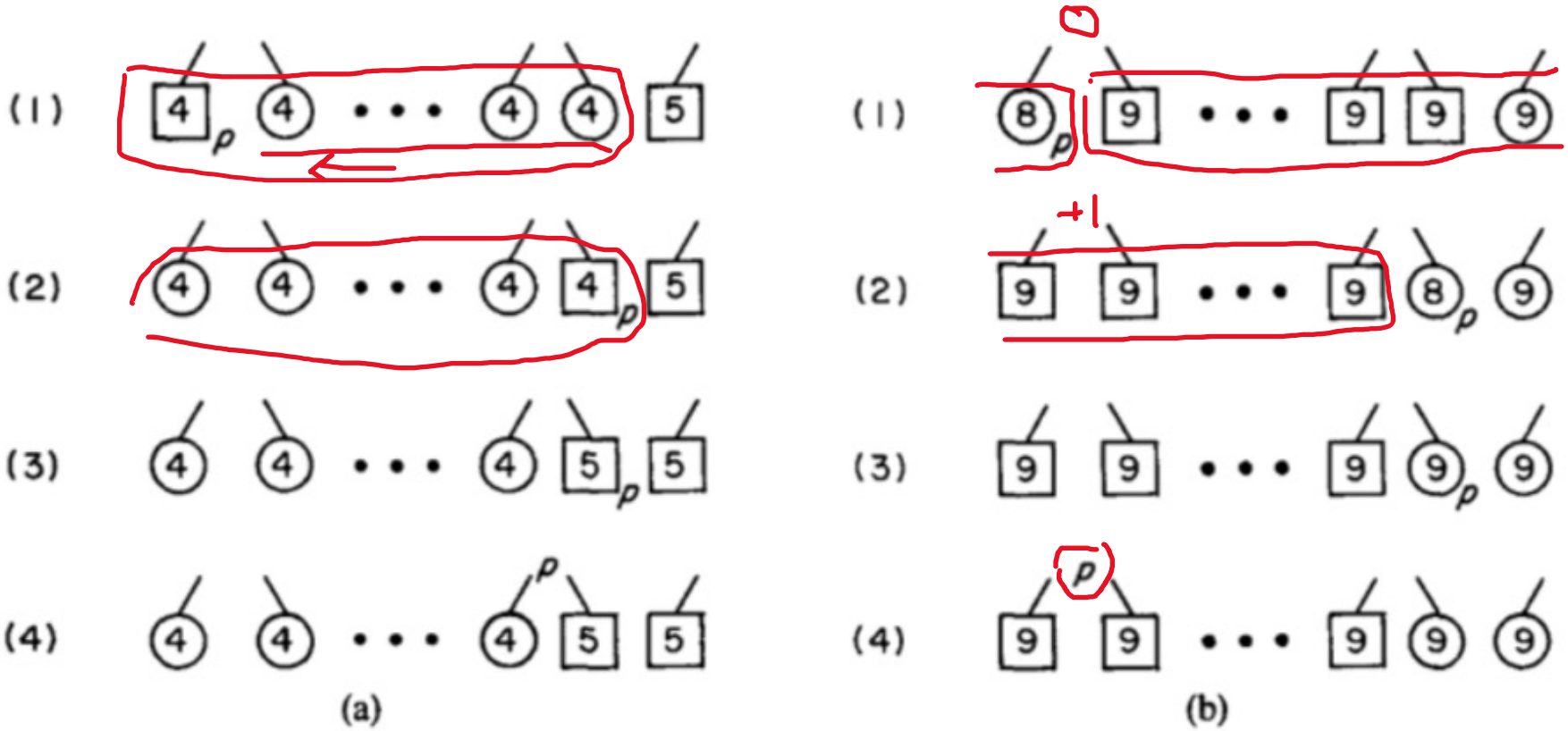


FIG. 6. Algorithm A's *SlideAndIncrement* operation. All the nodes in a given block shift to the left one spot to make room for node p, which slides over the block to the right. (a) Node p is a leaf of weight 4. The internal nodes of weight 4 shift to the left. (b) Node p is an internal node of weight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

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;
```

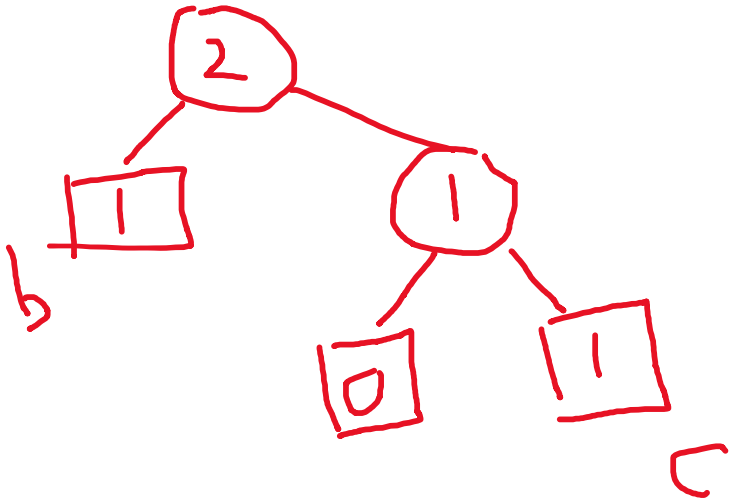
Add a new node

Update the node next to NYT

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

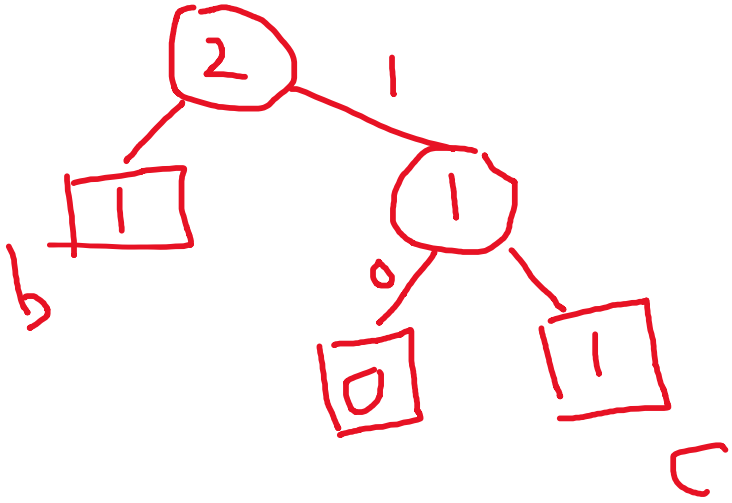


01100010 001100011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

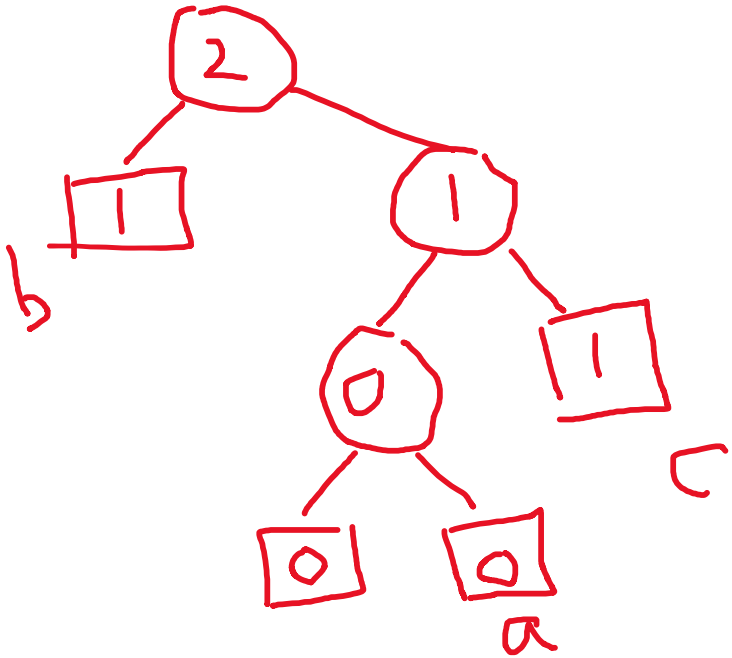


01100010 001100011 1001100001

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

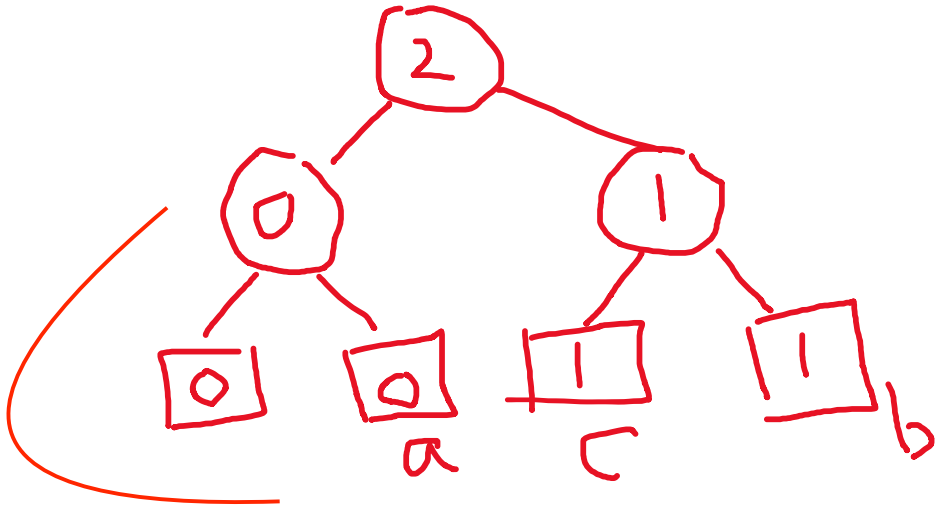


01100010 001100011 1001100001

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

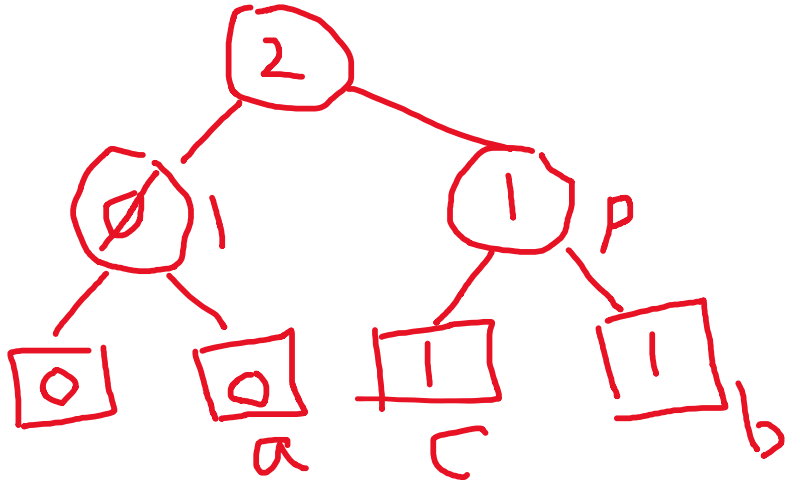


01100010 001100011 1001100001

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

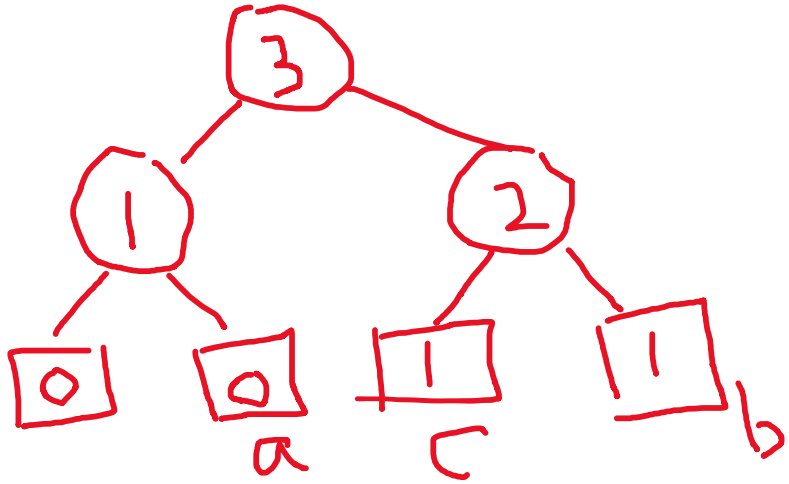


01100010 001100011 1001100001

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

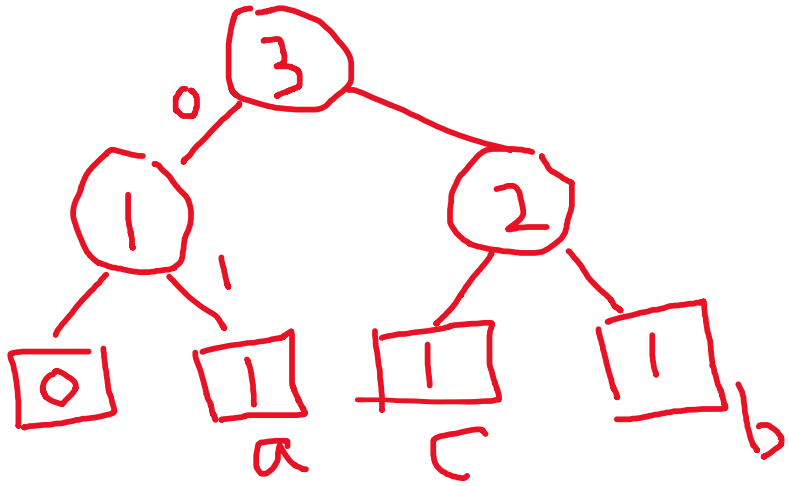


01100010 001100011 1001100001

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

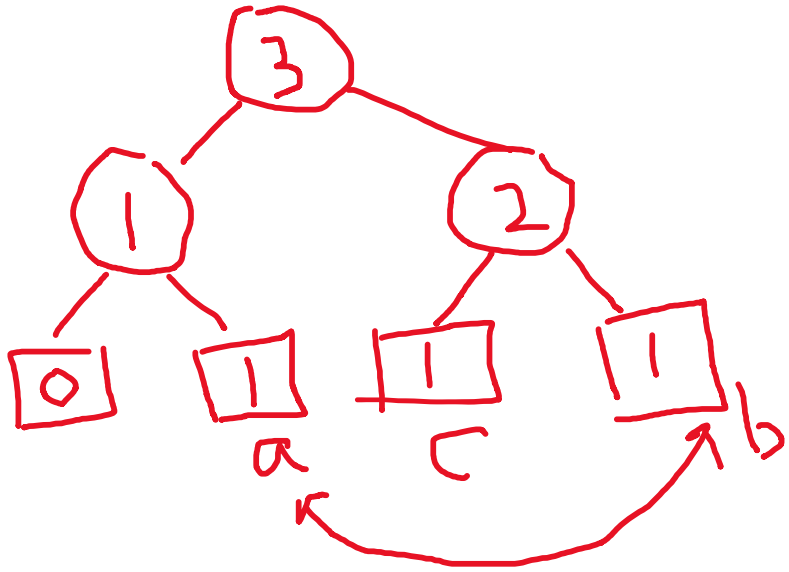


01100010 001100011 1001100001 01

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

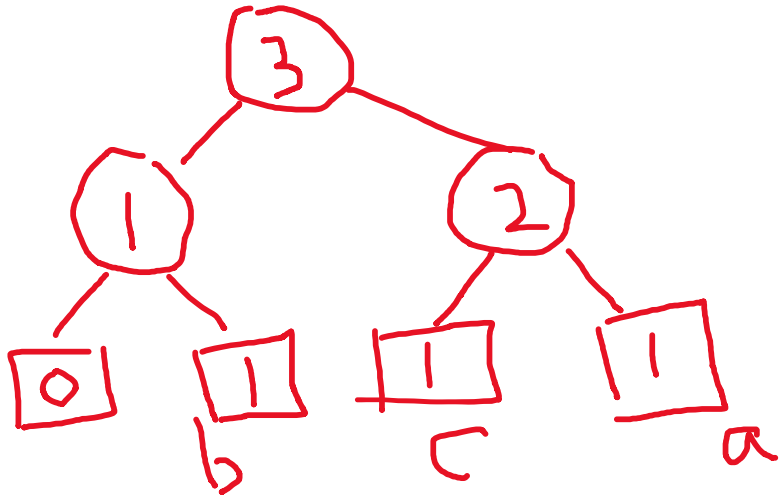


01100010 001100011 1001100001 01

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

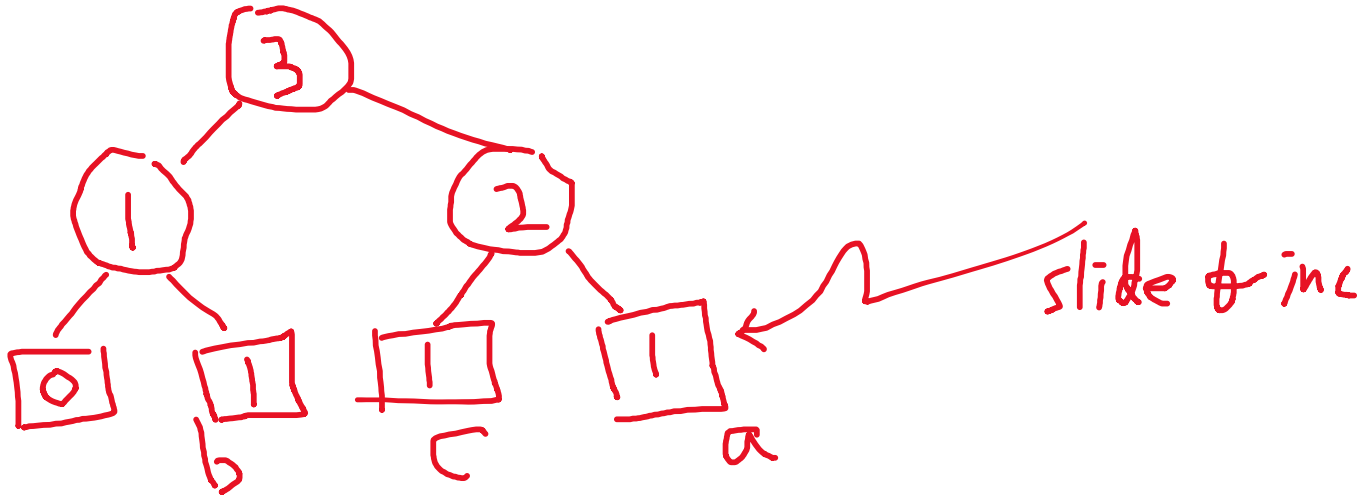


01100010 001100011 1001100001 01

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.



01100010 001100011 1001100001 01

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

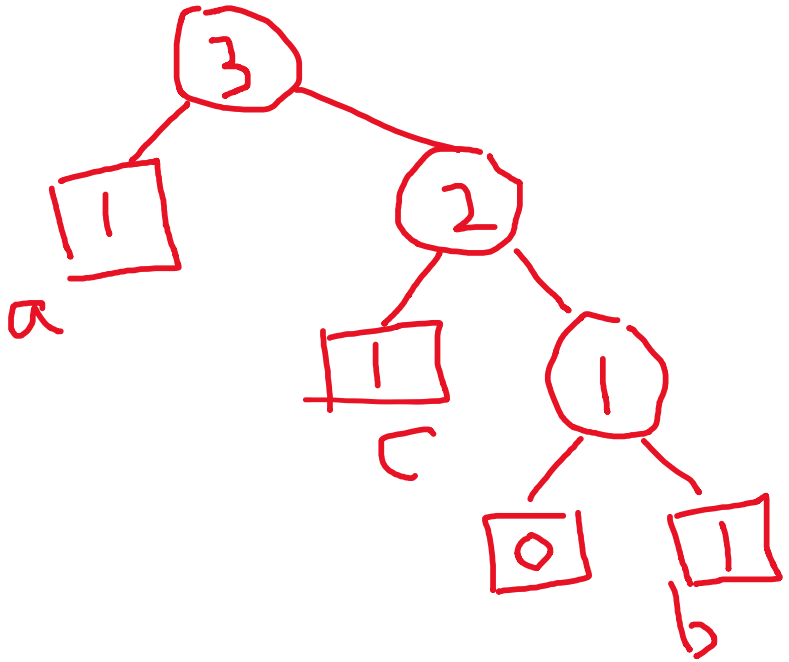
Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

01100010 001100011 1001100001 01

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

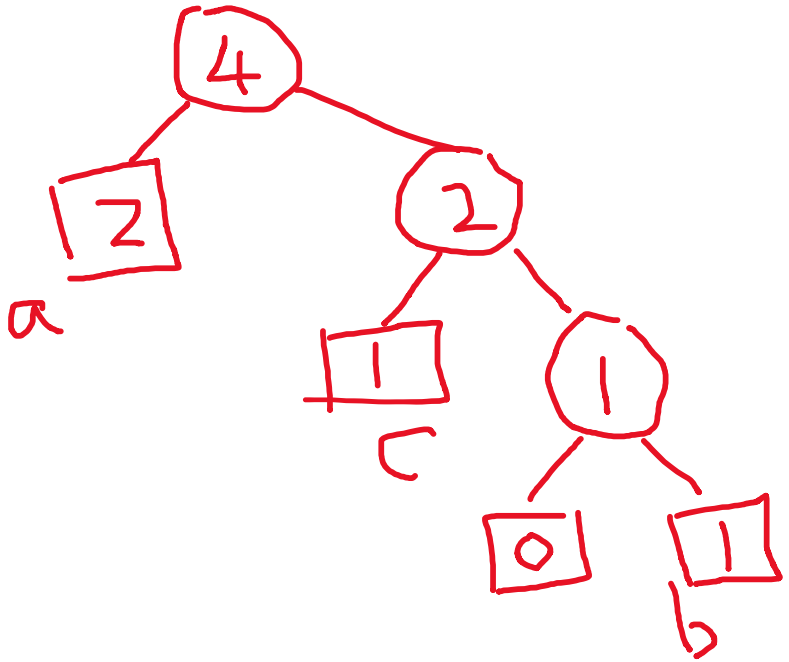


01100010 001100011 1001100001 01

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

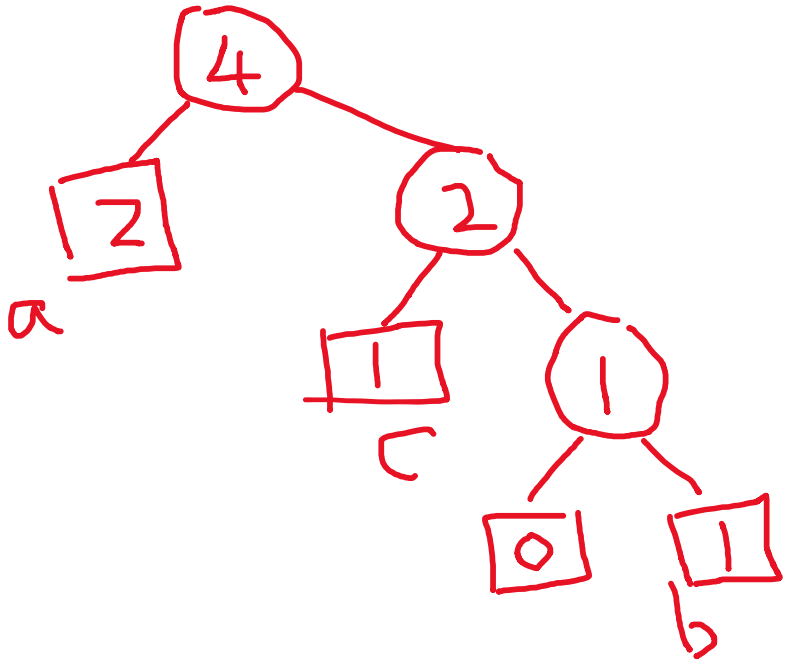


01100010 001100011 1001100001 01

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

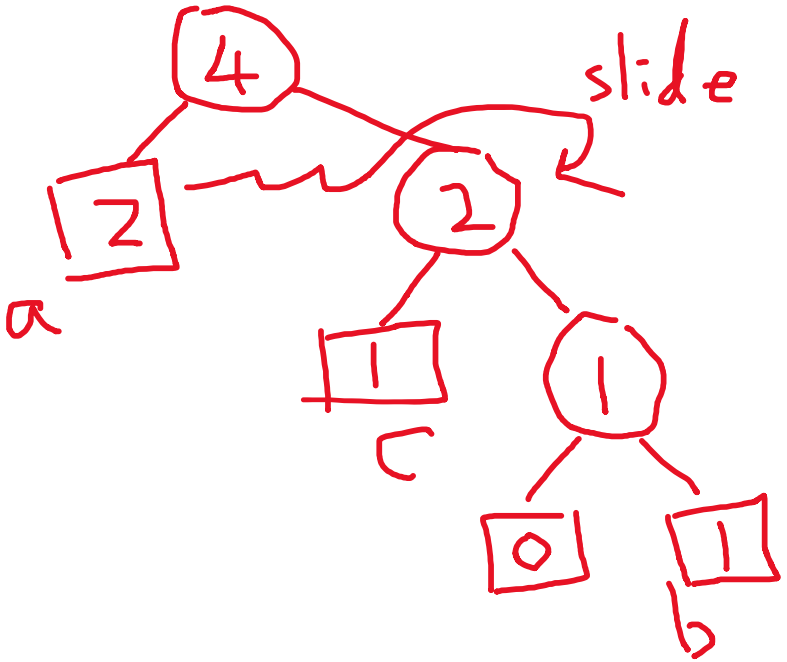


01100010 001100011 1001100001 01 0

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

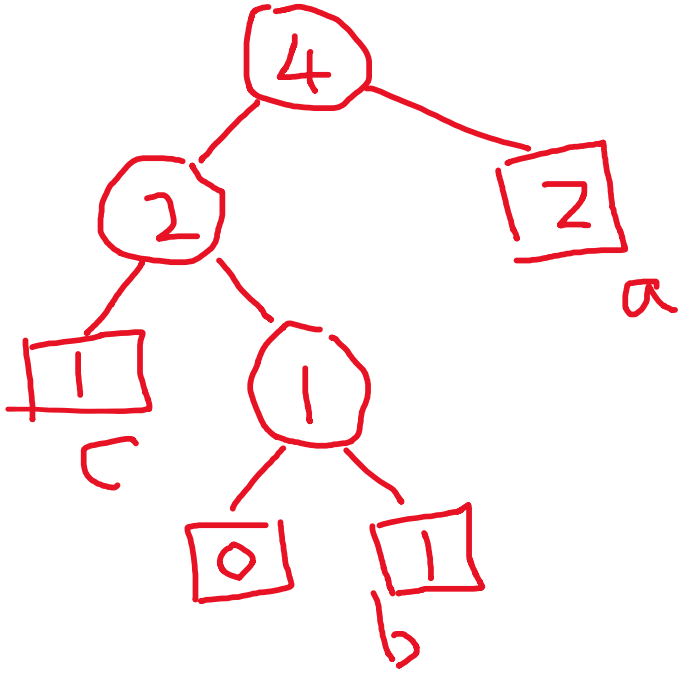


01100010 001100011 1001100001 01 0

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

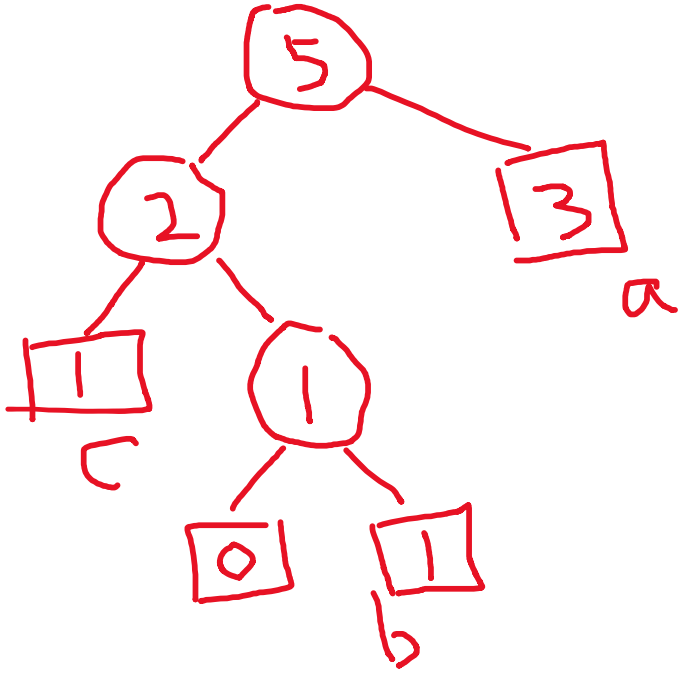


01100010 001100011 1001100001 01 0

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

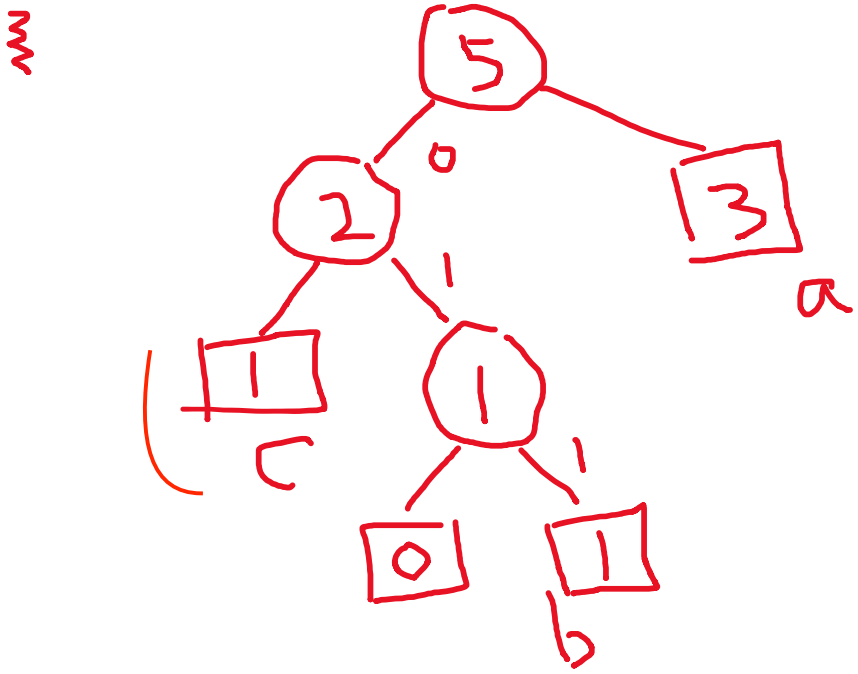


01100010 001100011 1001100001 01 0

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

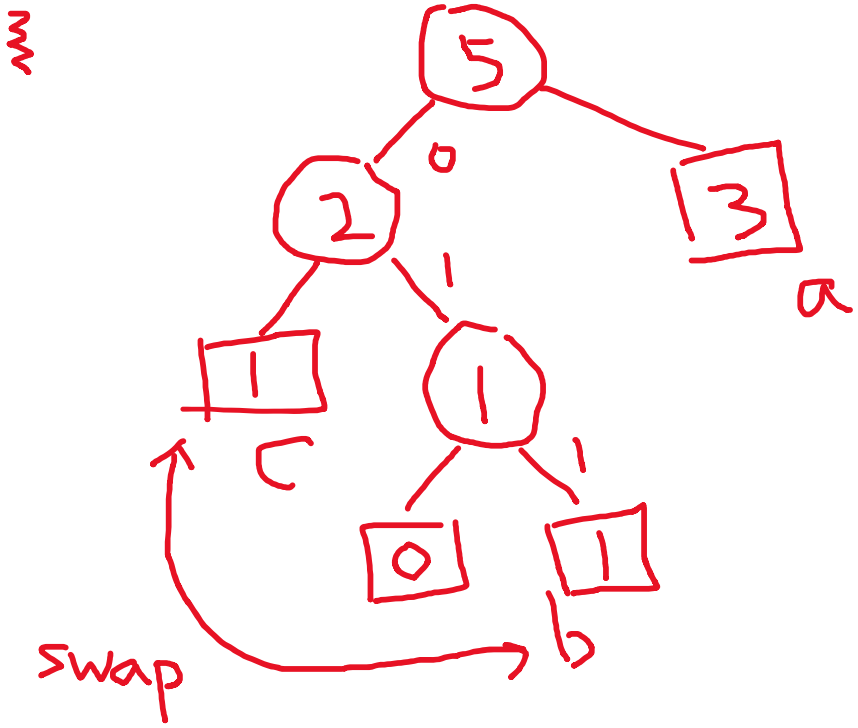


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

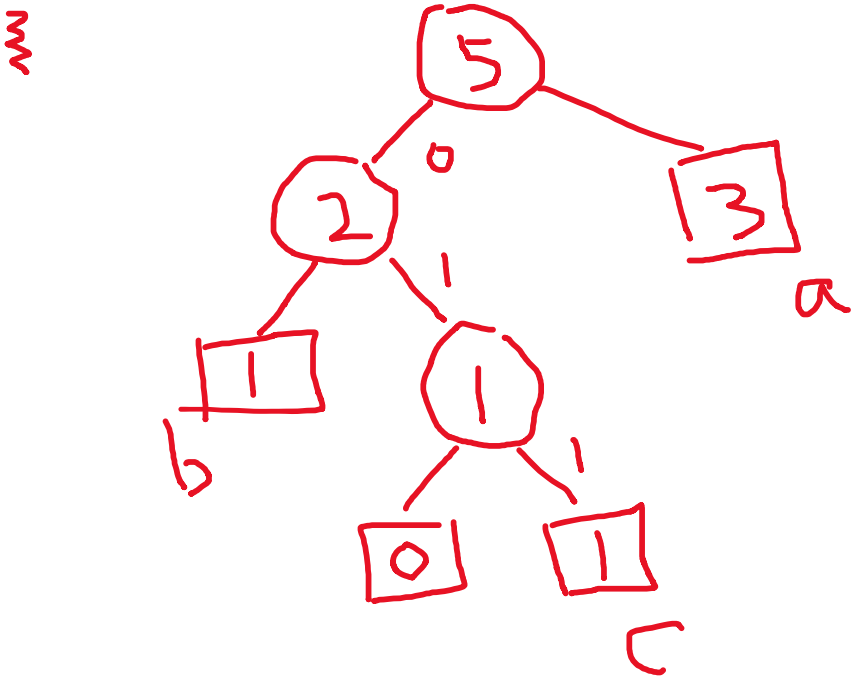


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

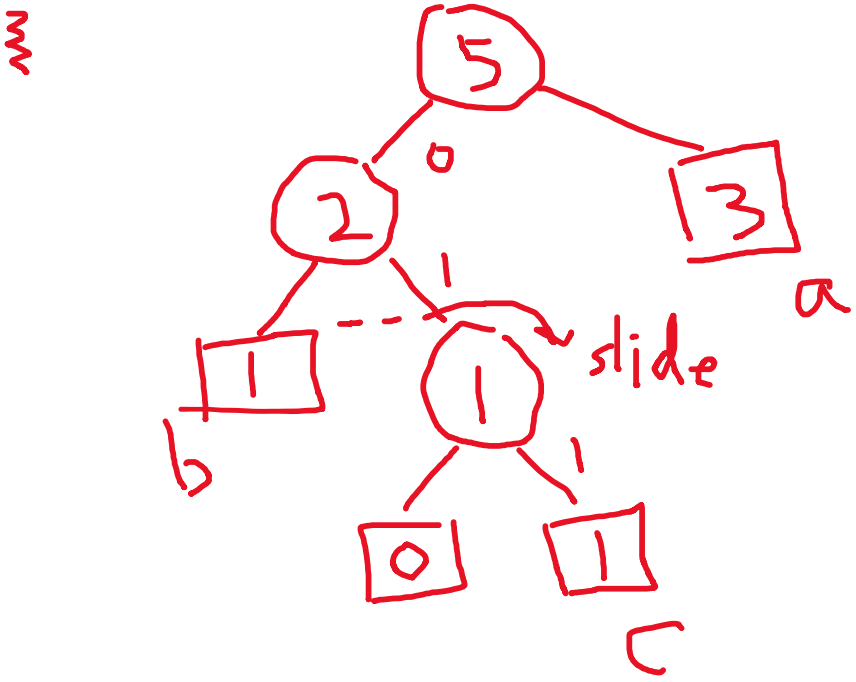


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

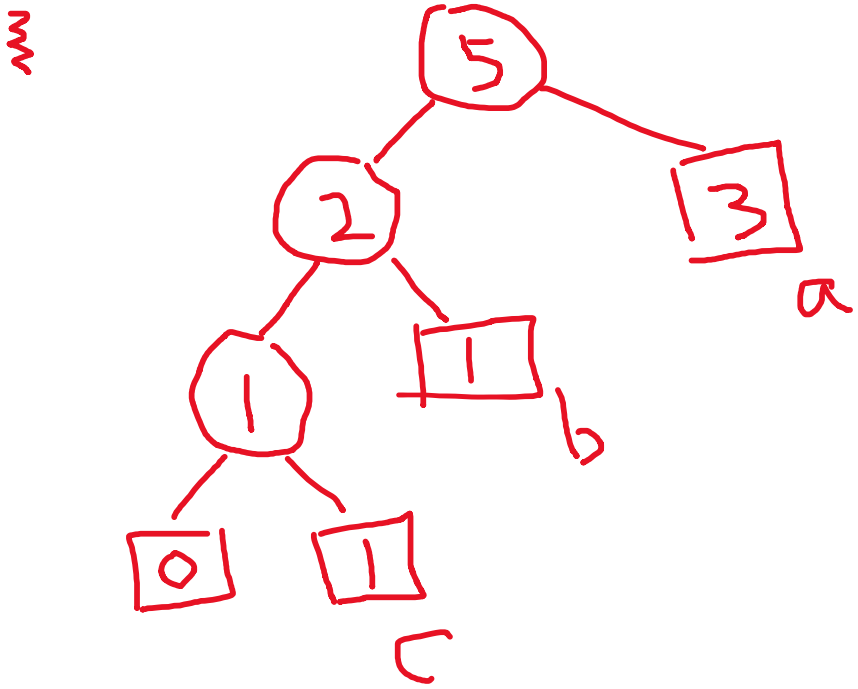


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

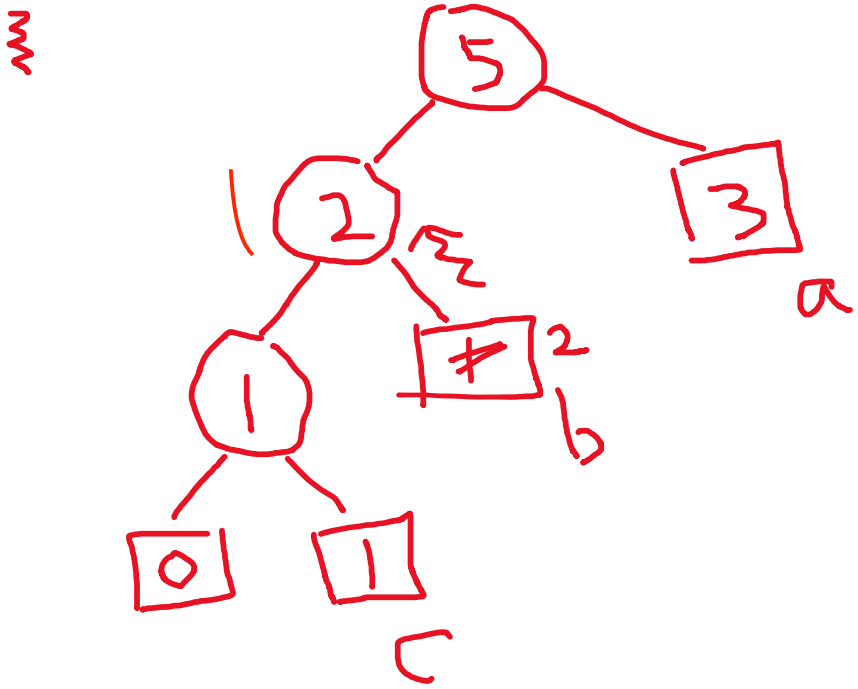


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

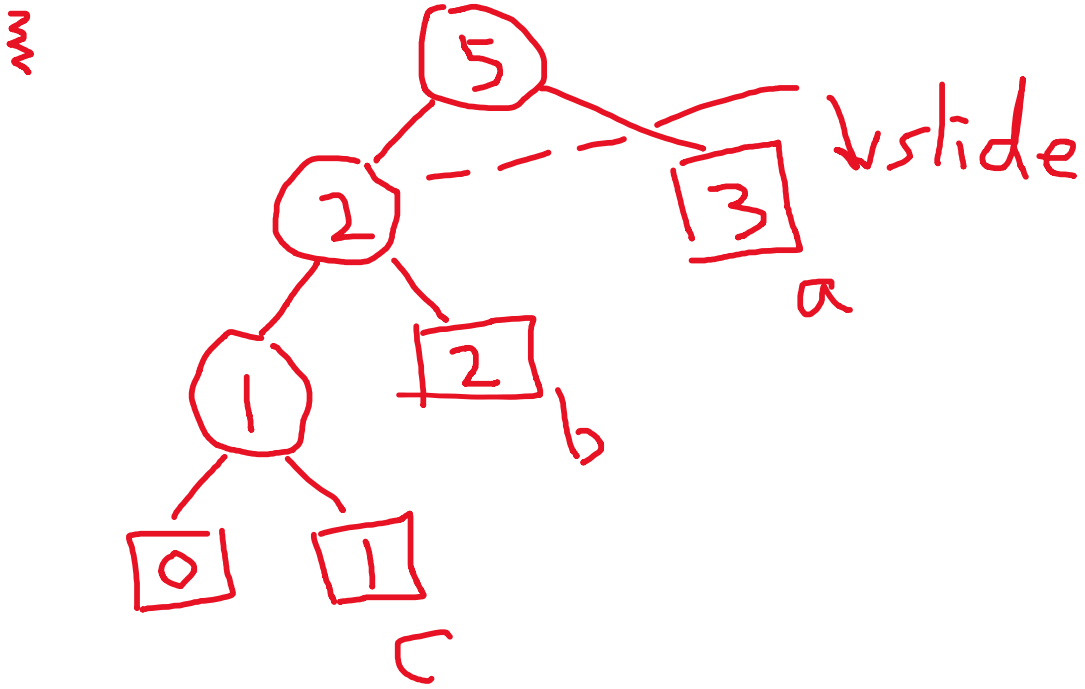


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

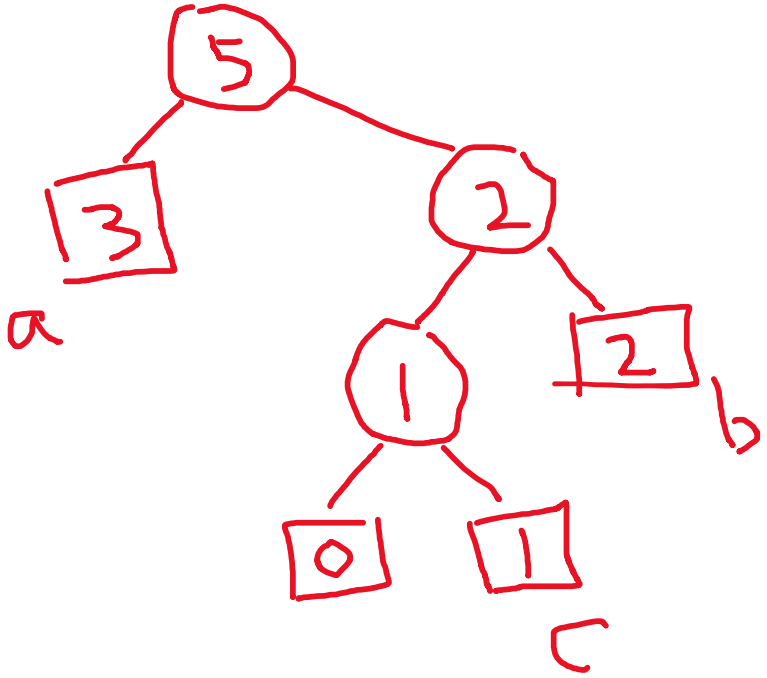


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

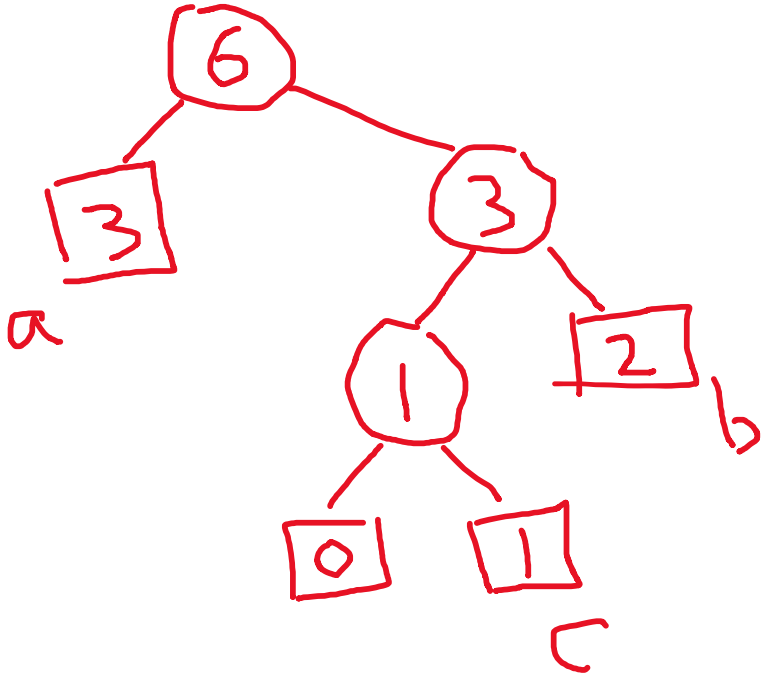


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

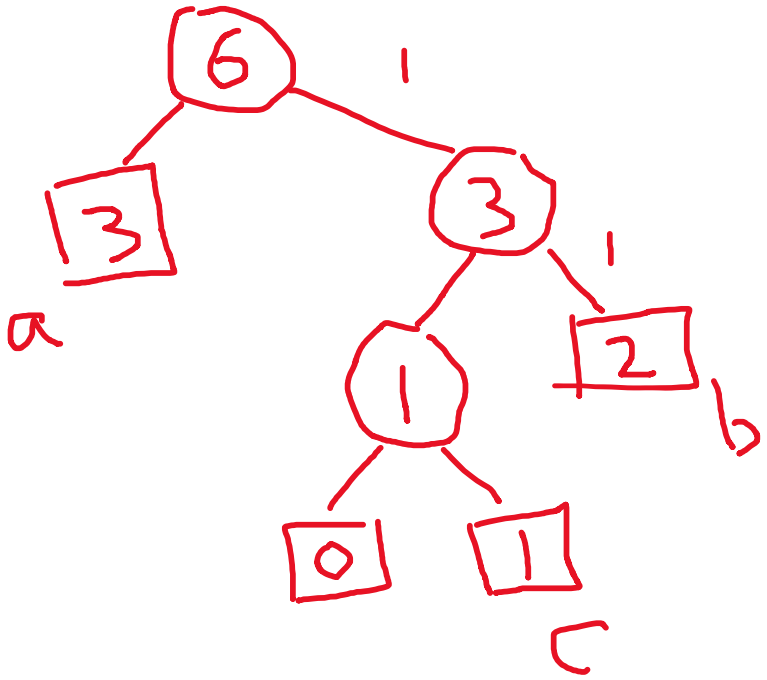


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

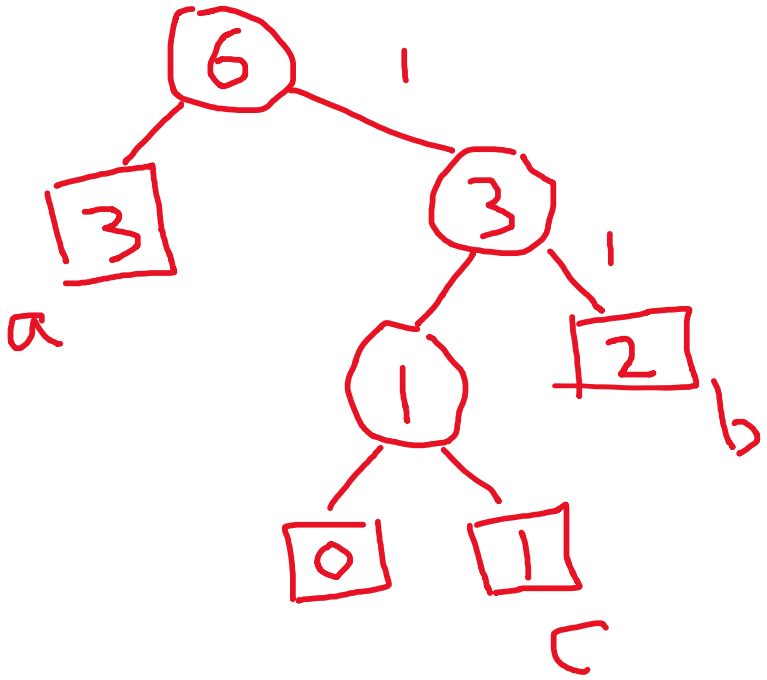


01100010 001100011 1001100001 01 0 011

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

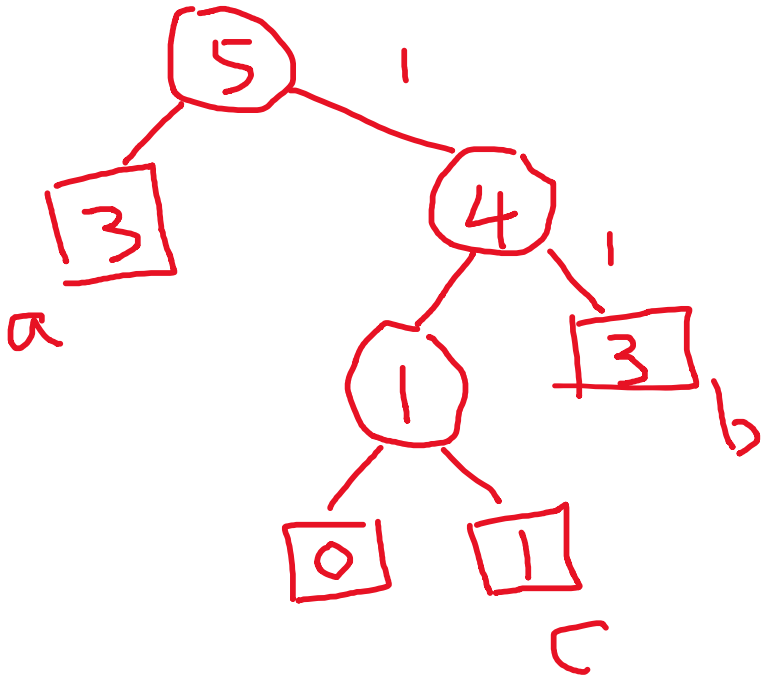


01100010 001100011 1001100001 01 0 011 11

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

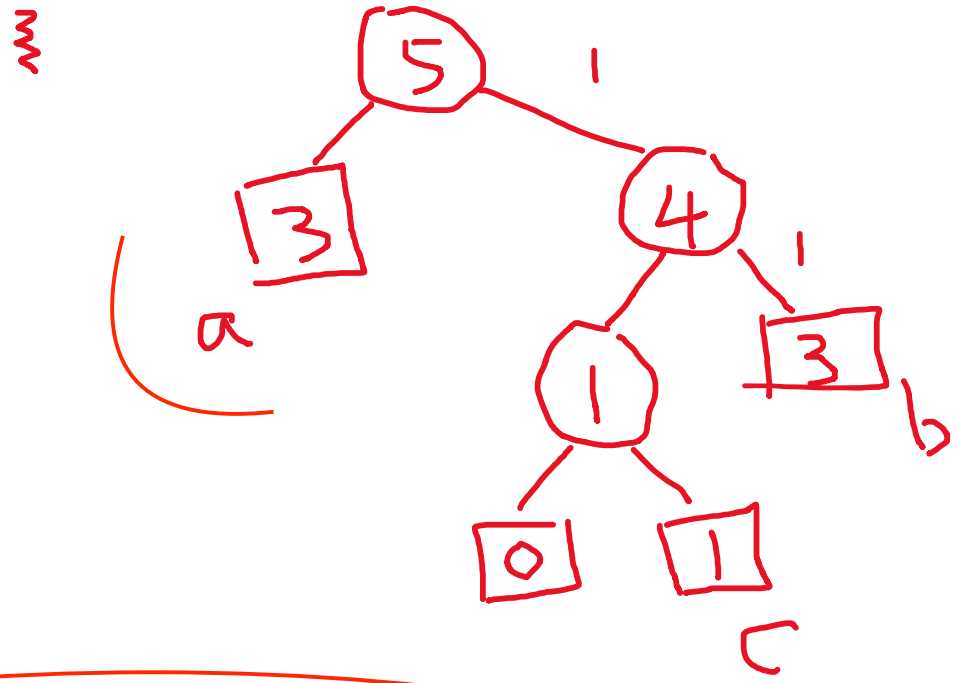


01100010 001100011 1001100001 01 0 011 11

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.

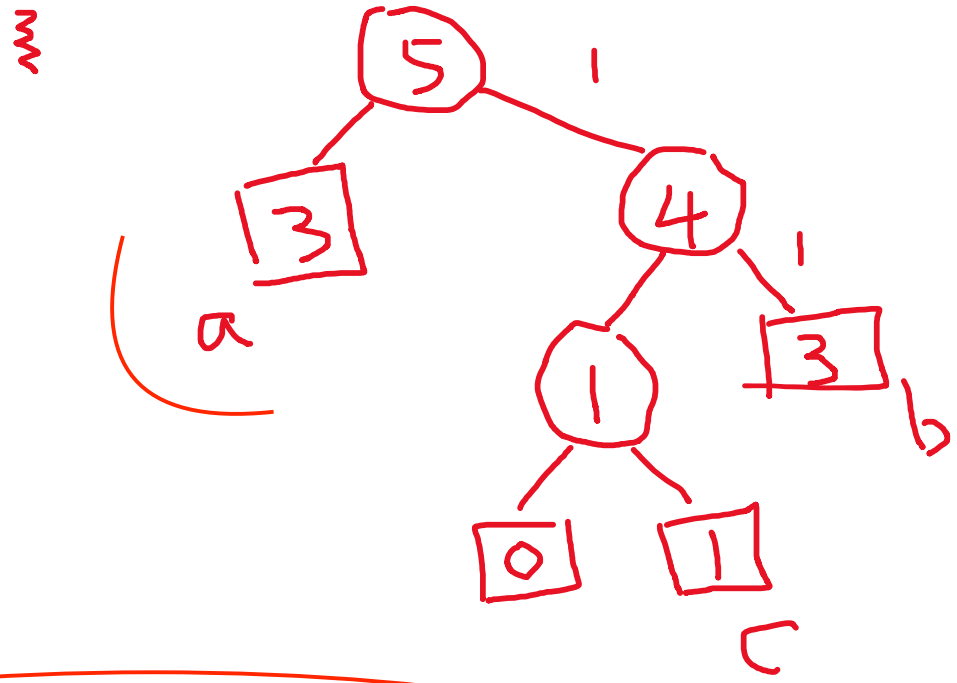


01100010 001100011 1001100001 01 0 011 11 11

Adaptive Huffman (Ex2, Q2)

The initial coding before any transmission is: a=01100001, b=01100010, c=01100011.

Derive the encoded bitstream produced by the encoder for the string **bcaaabbb**.



01100010 001100011 1001100001 01 0 011 11 11