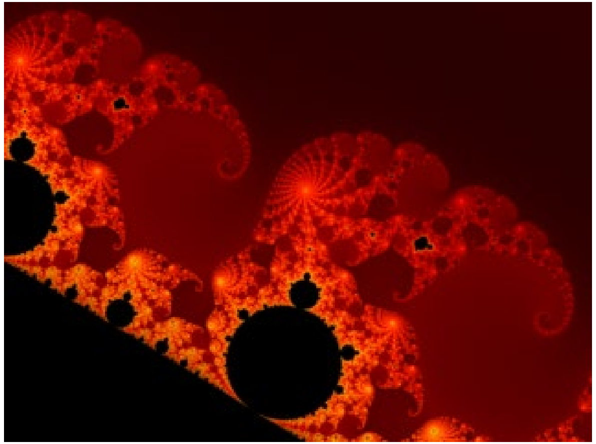
文件压缩算法简介

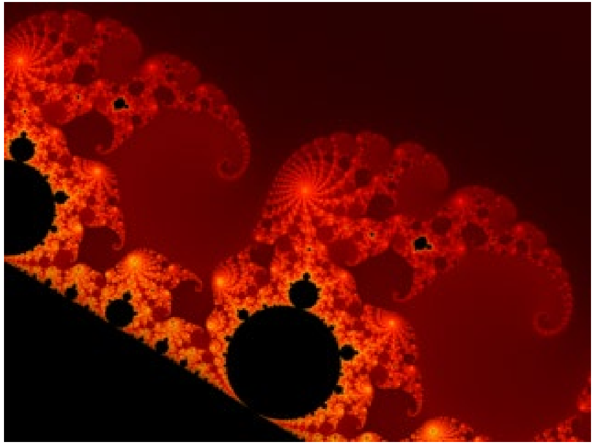
林日中 (1951112)
2022年4月8日

数据压缩

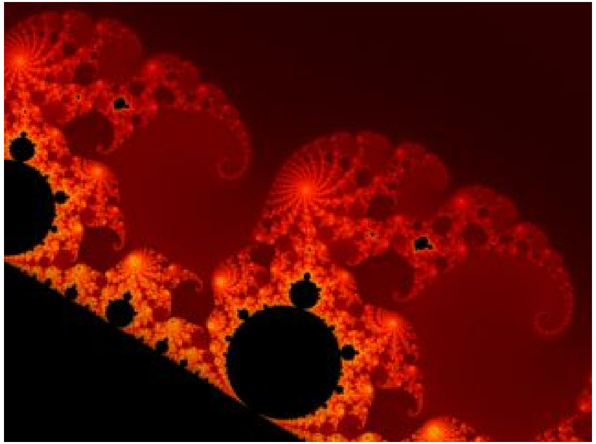
数据压缩依赖于 利用可预测的规律 和/或 牺牲精确性。



未经压缩的图像
320×240, 225kB



无损压缩的图像
320×240, 61.1kB



有损压缩的图像
320×240, 11.6kB

数据压缩的效率

- 数据压缩率（Data Compression Ratio, DCR）

$$\mathrm{DCR} = \frac{\text{原文件大小}}{\text{压缩文件大小}} \text{ or } \frac{\text{原数据率}}{\text{压缩数据率}}$$

- 节省百分比

$$\text{Percentage Savings} = \left(1 - \frac{1}{\mathrm{DCR}}\right) \times 100\%$$

- 保真度（压缩前后数据的差别）

数据压缩算法的分类

- 有损压缩
  a. 小波变换（Wavelet Transform）
- 无损压缩
  a. 游程编码（Run Length Encoding）
  b. 熵编码（Entropy Encoding）
     i. 哈夫曼编码（Huffman Coding，霍夫曼／赫夫曼）
     ii. 香农-范诺编码（Shannon-Fano Coding）
  c. 字典编码
     i. LZ 系列编码

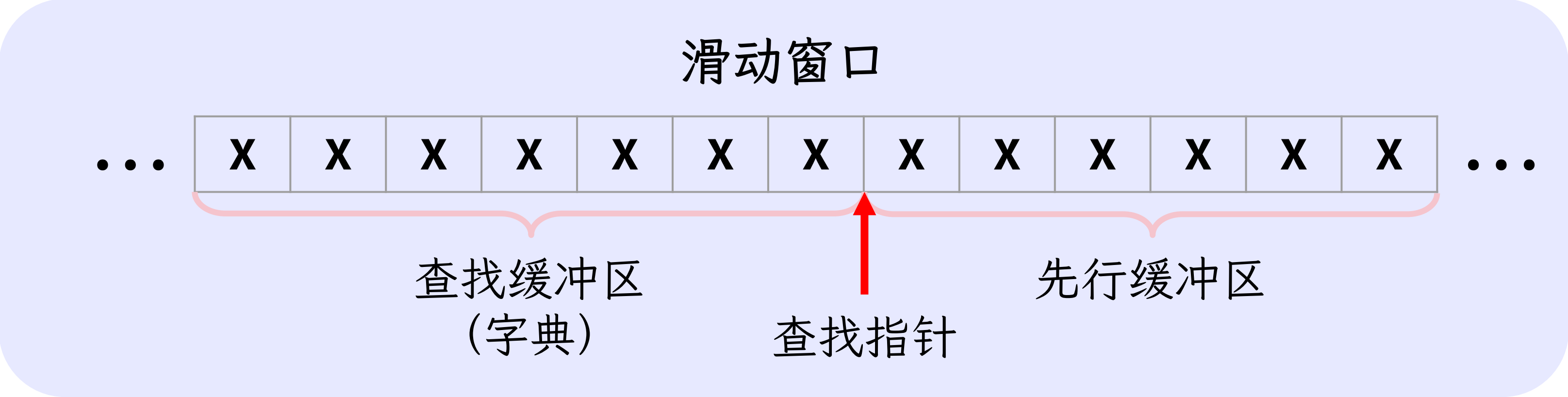
样例程序：删去重复内容

```
1. #Software: Hllpoj Server 1.0.1 / Logger 1.0.0 built 0001
2. #Version: 1.0
3. #Date: 2019-03-13 00:00:00
4. #Fields: date time s-ip cs-method cs-uri-stem cs-uri-query s-port cs-username c-ip cs(User-Agent) cs(Referer) sc-status sc-substatus sc-win32-status time-taken
5. 2019-03-13 00:00:00 ***.***.***.*** POST /login.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2 http://***.***.***.***/index.html 200 0 0 114
6. 2019-03-13 00:00:00 ***.***.***.*** POST /default/******.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2 http://***.***.***.***/default/******.htm 200 0 0 23
7. 2019-03-13 00:00:00 ***.***.***.*** POST /default/******.php - 80 - ***.***.***.*** Mozilla/5.0+(Linux;+U;+Android+8.0.0;+en-us;+MIX+2+Build/OPR1.170623.027)+AppleWebKit/537.36+(KHTML,+like+Gecko)+Version/4.0+Chrome/61.0.3163.128+Mobile+Safari/537.36+XiaoMi/MiuiBrowser/10.5.2 http://***.***.***.***/default/******.htm 200 0 0 2
```

LZ算法

- LZ77是一种基于字典的算法，是用其发明者的名字（Abraham Lempel 和 Jacob Ziv）命名的。它将长字符串（也称为短语）编码成短小的标记，用小标记代替字典中的短语，从而达到压缩的目的。
- LZ算法形成了包括GIF和在PNG和ZIP中使用的DEFLATE算法等压缩方案的基础。

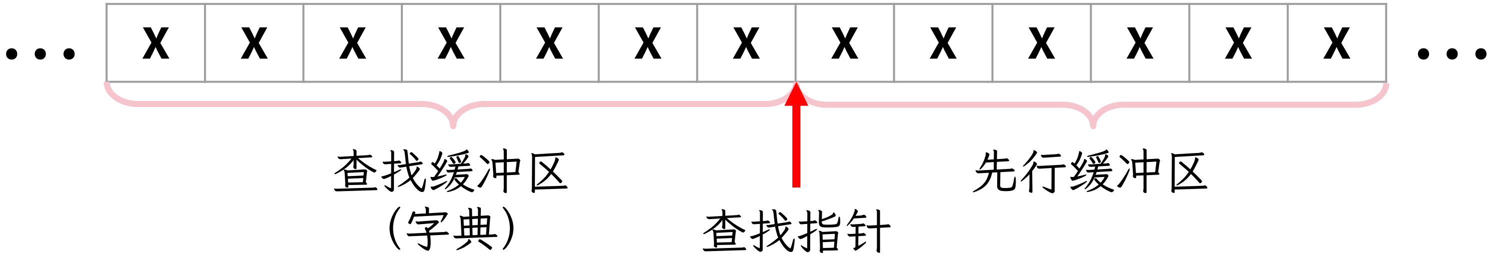
LZ77算法



- 由于缓冲区具有固定的长度，所以，当LZ算法编码器运行时，看起来像在文件中“滑动”，所以这个结构被称为“滑动窗口”。

LZ77算法中的三元组 <o，l，c>

- Offset – 偏移量，向左移动后的指针与先行缓冲区的距离
- Length of match – 匹配长度
- Codeword – 先行缓冲区中位于匹配项之后的字符



LZ77算法的压缩过程

- 考察字符串 "**CABRAC**ADABRARRARRAD"

<偏移量，匹配长度，匹配项后的符号>

							C	A	B	R	A	C	

C -> <0, 0, C>

LZ77算法的压缩过程

							C	A	B	R	A	C

`C -> <0, 0, C>`

						C	A	B	R	A	C	A

`A -> <0, 0, A>`

"CABRACADABRARRARRAD"

LZ77算法的压缩过程

						C	A	B	R	A	C	A

`A -> <0, 0, A>`

					C	A	B	R	A	C	A	D

`B -> <0, 0, B>`

"CA**BRACAD**ABRARRARRAD"

LZ77算法的压缩过程

					C	A	B	R	A	C	A	D

`B -> <0, 0, B>`

				C	A	B	R	A	C	A	D	A

`R -> <0, 0, R>`

"CABRACADABRARRARRAD"

LZ77算法的压缩过程

				C	A	B	R	A	C	A	D	A

`R -> <0, 0, R>`

			C	A	B	R	A	C	A	D	A	B

`A -> <3, 1, C>`

"CABRACADABRARRARRAD"

LZ77算法的压缩过程

			C	A	B	R	A	C	A	D	A	B

A -> <3, 1, C>

	C	A	B	R	A	C	A	D	A	B	R	A

A -> <2, 1, D>

"CABRACADABRARRARRAD"

LZ77算法的压缩过程

	C	A	B	R	A	C	A	D	A	B	R	A

`A -> <2, 1, D>`

A	B	R	A	C	A	D	A	B	R	A	R	R

`A -> <7, 4, R>`

"CABRACADABRARRARRAD"

LZ77算法的压缩过程

A	B	R	A	C	A	D	A	B	R	A	R	R

A -> <7, 4, R>

A	D	A	B	R	A	R	R	A	R	R	A	D

R -> <3, 5, D>

"CABRACADABRARRARRAD"

LZ77算法的解压缩过程

<0, 0, C>

<0, 0, A>

<0, 0, B>

<0, 0, R>

<3, 1, C>

<2, 1, D>

<7, 4, R>

<3, 5, D>

						C

CABRACADABRARRARRAD

LZ77算法的解压缩过程

<0, 0, C>

<0, 0, A>

<0, 0, B>

<0, 0, R>

<3, 1, C>

<2, 1, D>

<7, 4, R>

<3, 5, D>

						C

					C	A

CABRACADABRARRARRAD

LZ77算法的解压缩过程

<0, 0, C>

<0, 0, A>

<0, 0, B>

<0, 0, R>

<3, 1, C>

<2, 1, D>

<7, 4, R>

<3, 5, D>

						C
					C	A
				C	A	B

CABRACADABRARRARRAD

LZ77算法的解压缩过程

<0, 0, C>

<0, 0, A>

<0, 0, B>

<0, 0, R>

<3, 1, C>

<2, 1, D>

<7, 4, R>

<3, 5, D>

						C
					C	A
				C	A	B
			C	A	B	R

CABRACADABRARRARRAD

LZ77算法的解压缩过程

- <0, 0, C>
- <0, 0, A>
- <0, 0, B>
- <0, 0, R>
- <3, 1, C>
- <2, 1, D>
- <7, 4, R>
- <3, 5, D>

						C
					C	A
				C	A	B
			C	A	B	R
	C	A	B	R	A	C

CABRACADABRARRARRAD

LZ77算法的解压缩过程

<0, 0, C>

<0, 0, A>

<0, 0, B>

<0, 0, R>

<3, 1, C>

<2, 1, D>

<7, 4, R>

<3, 5, D>

						C
					C	A
				C	A	B
			C	A	B	R
	C	A	B	R	A	C
A	B	R	A	C	A	D

CABRACADABRARRARRAD

LZ77算法的解压缩过程

- <0, 0, C>
- <0, 0, A>
- <0, 0, B>
- <0, 0, R>
- <3, 1, C>
- <2, 1, D>
- <7, 4, R>
- <3, 5, D>

						C	
					C	A	
				C	A	B	
			C	A	B	R	
	C	A	B	R	A	C	
A	B	R	A	C	A	D	
A	D	A	B	R	A	R	R A R

CABRACADABRARRARRAD

LZ77算法的解压缩过程

<0, 0, C>							C	
<0, 0, A>						C	A	
<0, 0, B>					C	A	B	
<0, 0, R>				C	A	B	R	
<3, 1, C>		C	A	B	R	A	C	
<2, 1, D>	A	B	R	A	C	A	D	
<7, 4, R>	A	D	A	B	R	A	R	R A R
<3, 5, D>	R	R	A	R	R	A	D	

LZ4算法

- LZ77算法的改进版，使用16k大小的<u>哈希表</u>来存储字典并简化检索。

<u>LZ4 Sequence</u>

Token : ==> 4-high-bits : literal length / 4-low-bits : match length

Token	*Literal length+ (optional)*	Literals	Offset	*Match length+ (optional)*
1-byte	0-n bytes	0-L bytes	2-bytes (little endian)	0-n bytes

anchor，“锚点”：两个锚点之间的数据块将被一次性存储。

1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**
0	1	2	3	4	5	6	7	8	9	10
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
11	12	13	14	15	16	17	18	19	20	21
13	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
22	23	24	25	26	27	28	29	30	31	32
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**2**	**3**
33	34	35	36	37	38	39	40	41	42	43
4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
44	45	46	47	48	49	50	51	52	53	54
15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
55	56	57	58	59	60	61	62	63	64	65
26	**27**	**28**	**29**	**30**	**31**	**32**				
66	67	68	69	70	71	72				

literal	
literal length	
match length	
offset	

anchors	0

1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**
0	1	2	3	4	5	6	7	8	9	10
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
11	12	13	14	15	16	17	18	19	20	21
13	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
22	23	24	25	26	27	28	29	30	31	32
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**2**	**3**
33	34	35	36	37	38	39	40	41	42	43
4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
44	45	46	47	48	49	50	51	52	53	54
15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
55	56	57	58	59	60	61	62	63	64	65
26	**27**	**28**	**29**	**30**	**31**	**32**				
66	67	68	69	70	71	72				

literal	
literal length	
match length	
offset	
anchors	0

1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**
0	1	2	3	4	5	6	7	8	9	10
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
11	12	13	14	15	16	17	18	19	20	21
13	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
22	23	24	25	26	27	28	29	30	31	32
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**2**	**3**
33	34	35	36	37	38	39	40	41	42	43
4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
44	45	46	47	48	49	50	51	52	53	54
15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
55	56	57	58	59	60	61	62	63	64	65
26	**27**	**28**	**29**	**30**	**31**	**32**				
66	67	68	69	70	71	72				

literal	
literal length	
match length	
offset	

anchors	0

1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**
0	1	2	3	4	5	6	7	8	9	10
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
11	12	13	14	15	16	17	18	19	20	21
13	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
22	23	24	25	26	27	28	29	30	31	32
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**2**	**3**
33	34	35	36	37	38	39	40	41	42	43
4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
44	45	46	47	48	49	50	51	52	53	54
15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
55	56	57	58	59	60	61	62	63	64	65
26	**27**	**28**	**29**	**30**	**31**	**32**				
66	67	68	69	70	71	72				

literal	[1, 2, 3, 4, 5]
literal length	5
match length	10
offset	5

anchors	0, 15

1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**
0	1	2	3	4	5	6	7	8	9	10
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
11	12	13	14	15	16	17	18	19	20	21
13	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
22	23	24	25	26	27	28	29	30	31	32
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**2**	**3**
33	34	35	36	37	38	39	40	41	42	43
4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
44	45	46	47	48	49	50	51	52	53	54
15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
55	56	57	58	59	60	61	62	63	64	65
26	**27**	**28**	**29**	**30**	**31**	**32**				
66	67	68	69	70	71	72				

literal	
literal length	
match length	
offset	

anchors	0, 15

Why not these yellow blocks?

1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**
0	1	2	3	4	5	6	7	8	9	10
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
11	12	13	14	**15**	16	17	18	19	20	21
13	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
22	23	24	25	26	27	28	29	30	31	32
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**2**	**3**
33	34	35	36	37	38	39	40	41	42	43
4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
44	45	**46**	47	48	49	50	51	52	53	54
15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
55	56	57	58	59	60	61	62	63	64	65
26	**27**	**28**	**29**	**30**	**31**	**32**				
66	67	68	69	70	71	72				

literal	[6, 7, 8, ..., 32]
literal length	27
match length	4
offset	41

anchors	0, 15, 46

1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**
0	1	2	3	4	5	6	7	8	9	10
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
11	12	13	14	15	16	17	18	19	20	21
13	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
22	23	24	25	26	27	28	29	30	31	32
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**2**	**3**
33	34	35	36	37	38	39	40	41	42	43
4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
44	45	46	47	48	49	50	51	52	53	54
15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
55	56	57	58	59	60	61	62	63	64	65
26	**27**	**28**	**29**	**30**	**31**	**32**				
66	67	68	69	70	71	72				

literal	<NULL>
literal length	0
match length	22
offset	31

anchors	0, 15, 46, 68

1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**
0	1	2	3	4	5	6	7	8	9	10
2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
11	12	13	14	**15**	16	17	18	19	20	21
13	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
22	23	24	25	26	27	28	29	30	31	32
24	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**2**	**3**
33	34	35	36	37	38	39	40	41	42	43
4	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
44	45	**46**	47	48	49	50	51	52	53	54
15	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**
55	56	57	58	59	60	61	62	63	64	65
26	**27**	**28**	**29**	**30**	**31**	**32**				
66	67	**68**	69	70	71	72				

literal	[28, 29, 30, 31, 32]
literal length	5
match length	0
offset	-

anchors	0, 15, 46, 68

Why 5 elements left?

完成

LZ4算法的压缩效率

```
PS E:\GradeOnePrograme\HomeWork\SimpleCompression\Debug> .\SimpleCompression.exe c test.log out.log
压缩率: 5.02592%
消耗时间: 89ms
PS E:\GradeOnePrograme\HomeWork\SimpleCompression\Debug> .\SimpleCompression.exe d out.log test1.log
消耗时间: 15ms
```

5.03%, <100ms

```
"Hello C!"
```

01001000 01100101
01101100 01101100
01101111 00100000
01000011 00100001

```
"Hello C!"
```

01001000 01100101
01101100 01101100
01101111 00100000
01000011 00100001

`"Hello C!"`

10010001 10010111

01100110 11001101

11101000 00100001

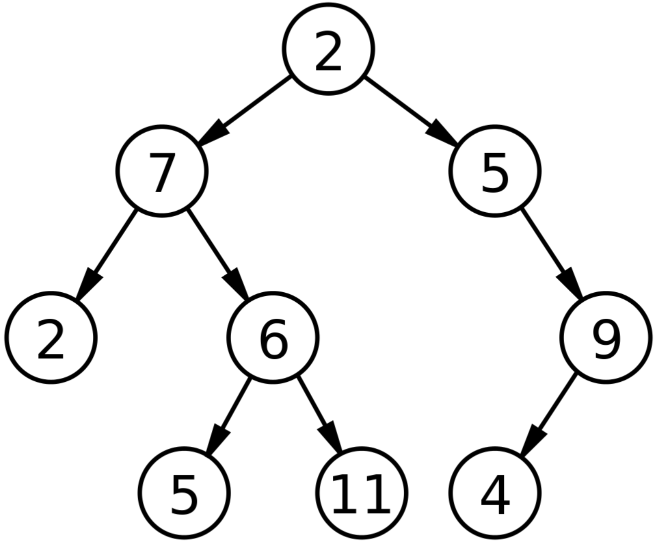
10100001

87.5%?

二叉树（Binary Tree）

- 二叉树是一种树形数据结构，其中每个节点最多拥有两个孩子，被称为左孩子和右孩子。

```
typedef struct BiTNode
{
    TElemType data;
    struct BiTNode *lchild, *rchild;
} BiTNode, *BiTree;
```



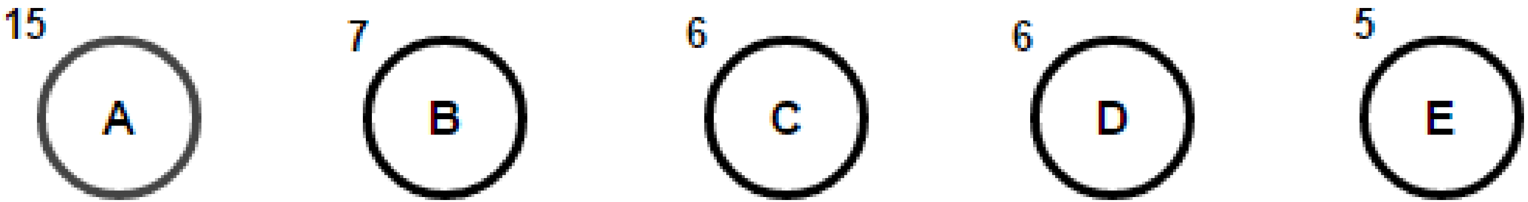
哈夫曼编码

- 哈夫曼编码（Huffman Coding），又译为霍夫曼编码、赫夫曼编码，是一种用于无损数据压缩的熵编码（权编码）算法。由美国计算器科学家戴维·霍夫曼（David Albert Huffman）在1952年发明。

哈夫曼编码算法

- 为每个角色创建一个叶子节点，并将它们添加到优先级队列中。
- 当队列中存在多个节点时：
  a. 从队列中删除优先级最高（频率最低）的两个节点。
  b. 创建一个新的内部节点，以这两个节点为子节点，频率等于两个节点的频率之和。
  c. 将新的节点添加到优先级队列中。
- 剩下的节点就是根节点了，树就完成了！

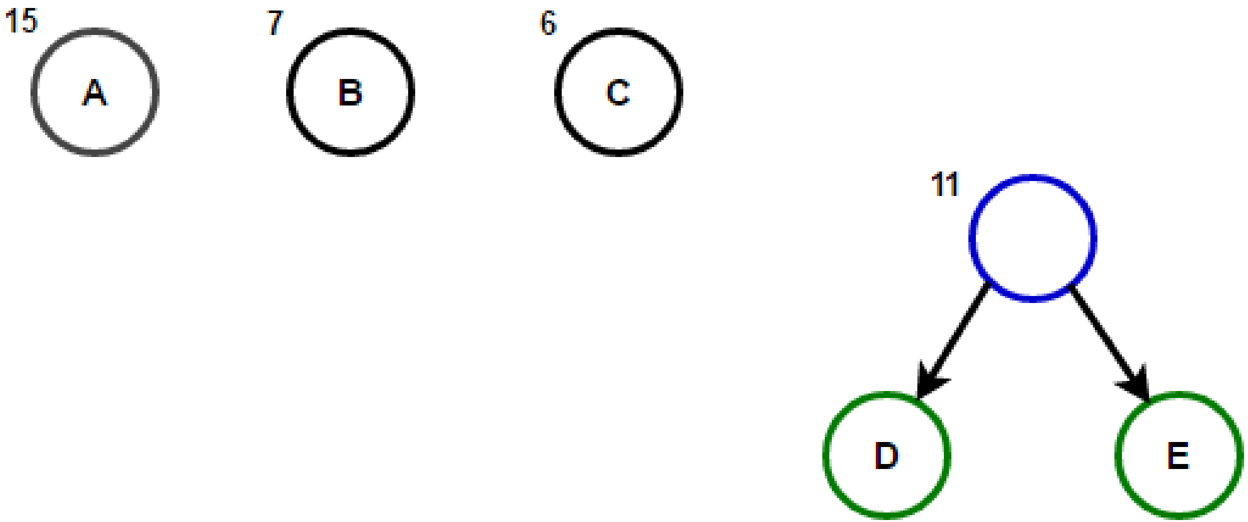
哈夫曼编码过程



DEADDAEBCABADBAABCAAEACAAABEADDABEBCCCA

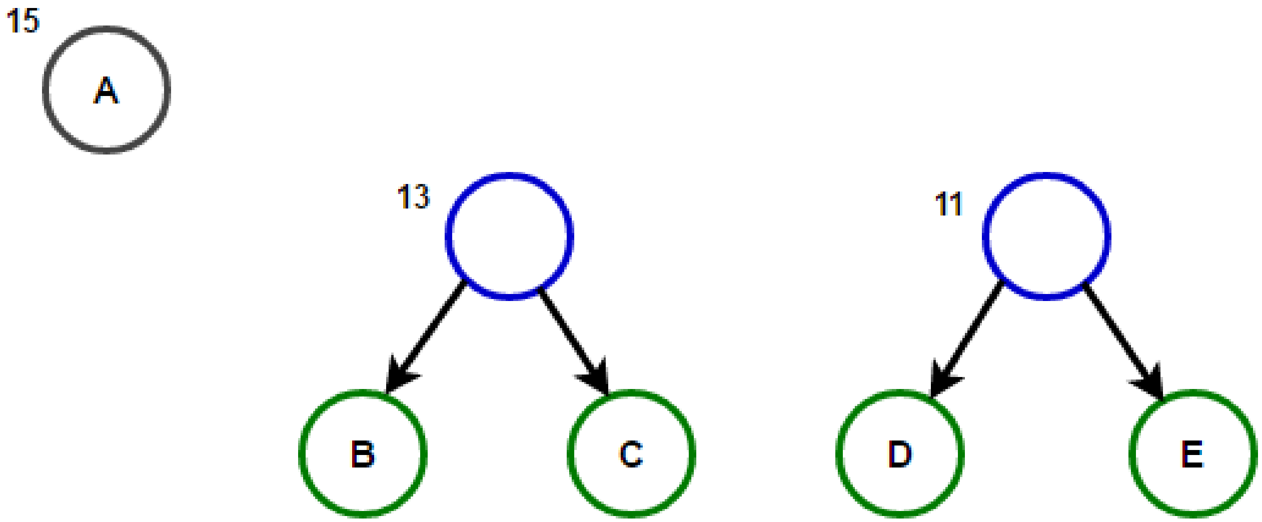
5: (E)
6: (D)
6: (C)
7: (B)
15: (A)

哈夫曼编码过程



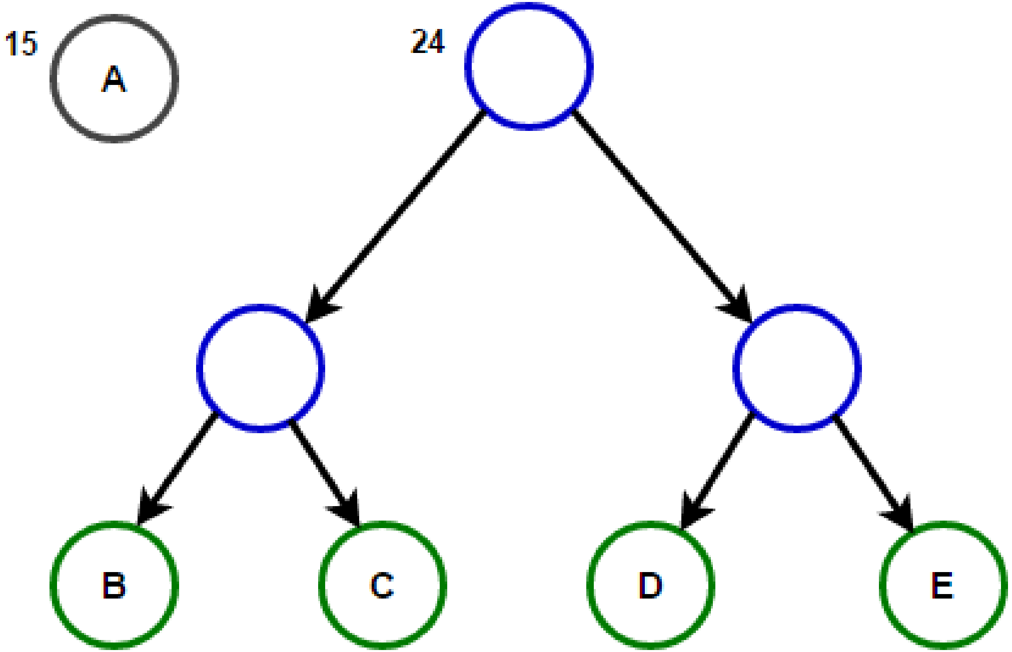
6: (C)
7: (B)
11: (D,E)
15: (A)

哈夫曼编码过程

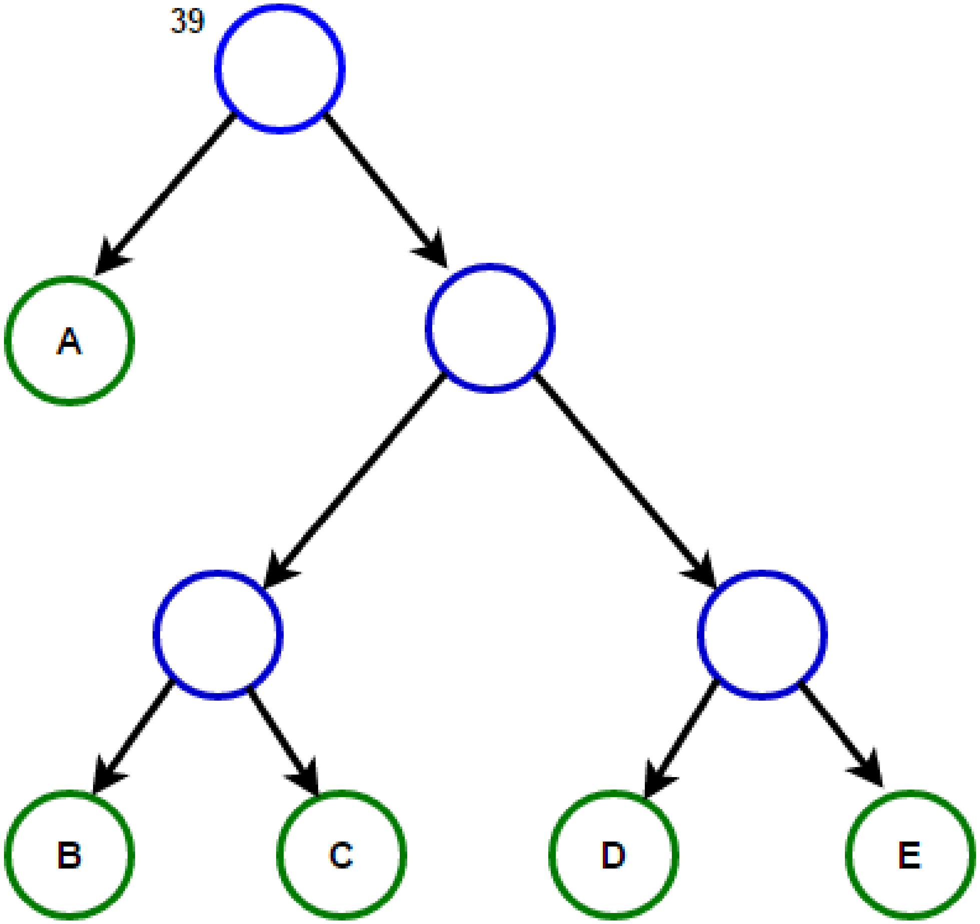


11: (D,E)
13: (B,C)
15: (A)

哈夫曼编码过程



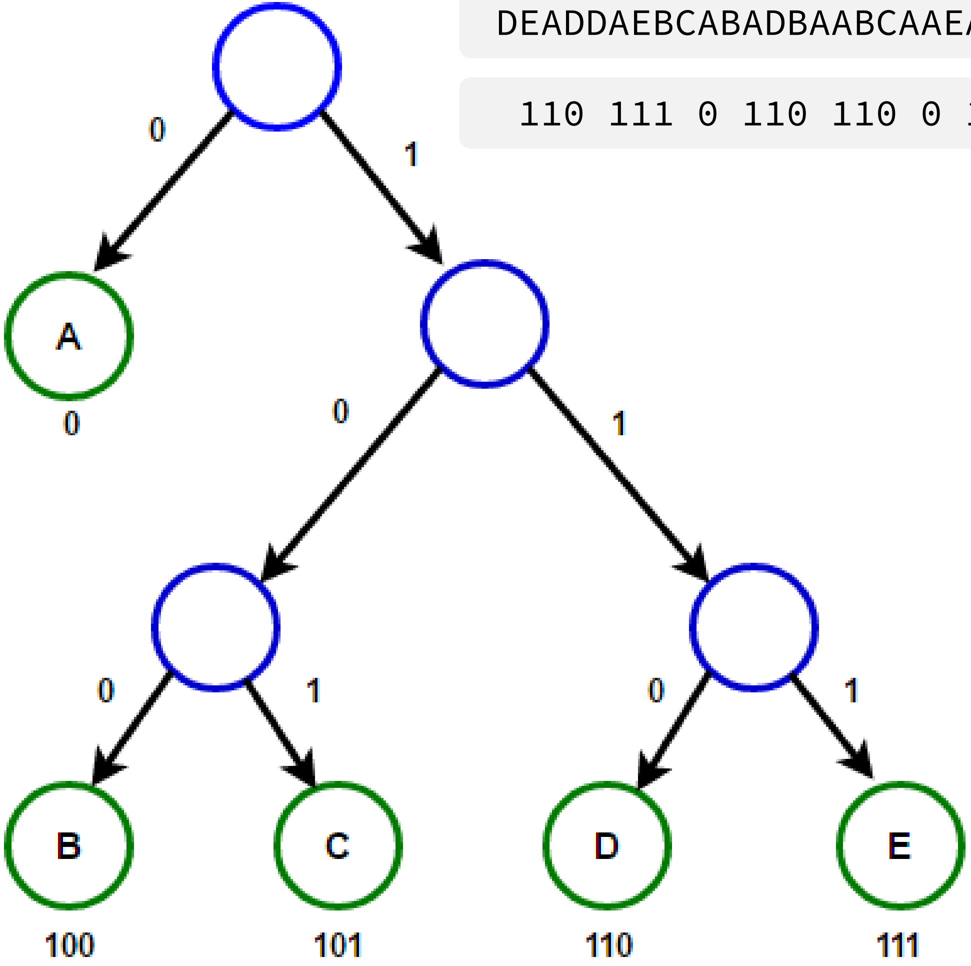
15: (A)
24: ((B,C),(D,E))



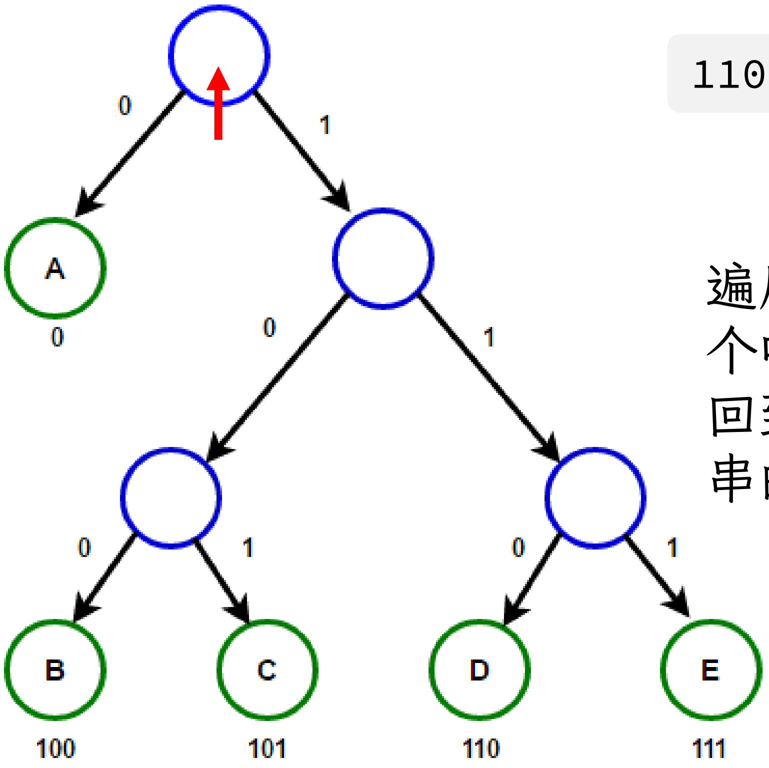
39: ((A),((B,C),(D,E)))

DEADDAEBCABADBAABCAAEACAAABEADDABEBCCCA

110 111 0 110 110 0 111 100 101 0 ...



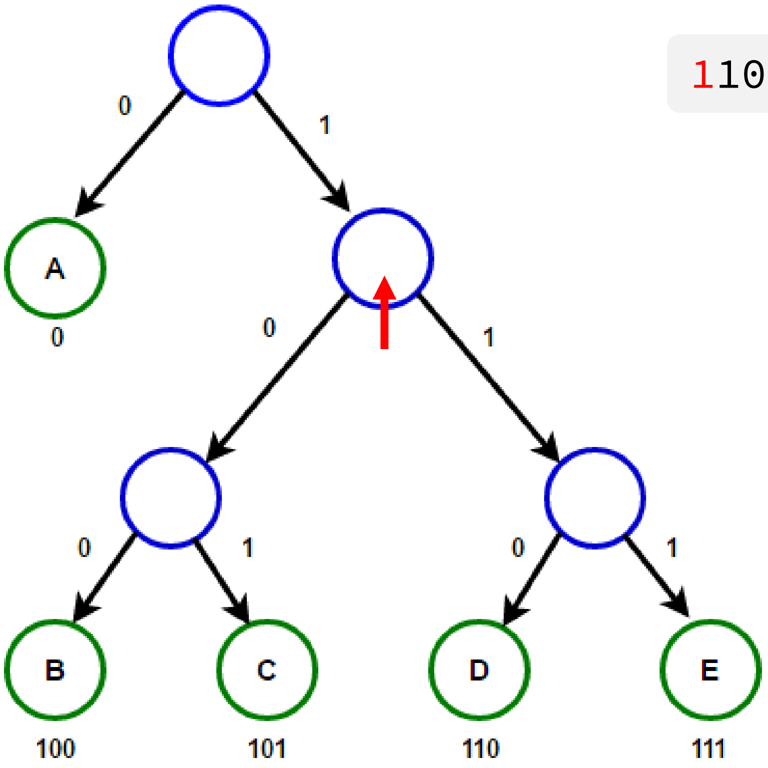
哈夫曼解码过程




```
11011101101100111100101010001 10...
```

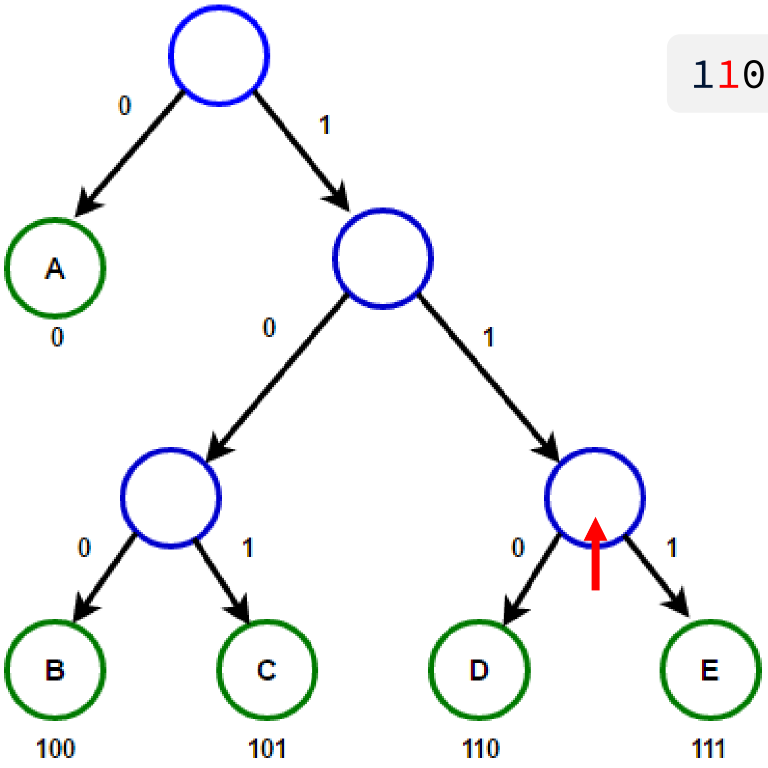
从根节点开始，根据编码的字符串遍历哈夫曼树中的节点；每当遍历到一个叶子节点（没有孩子的节点）时，就回到根节点，继续遍历，直到编码字符串的结束。

哈夫曼解码过程



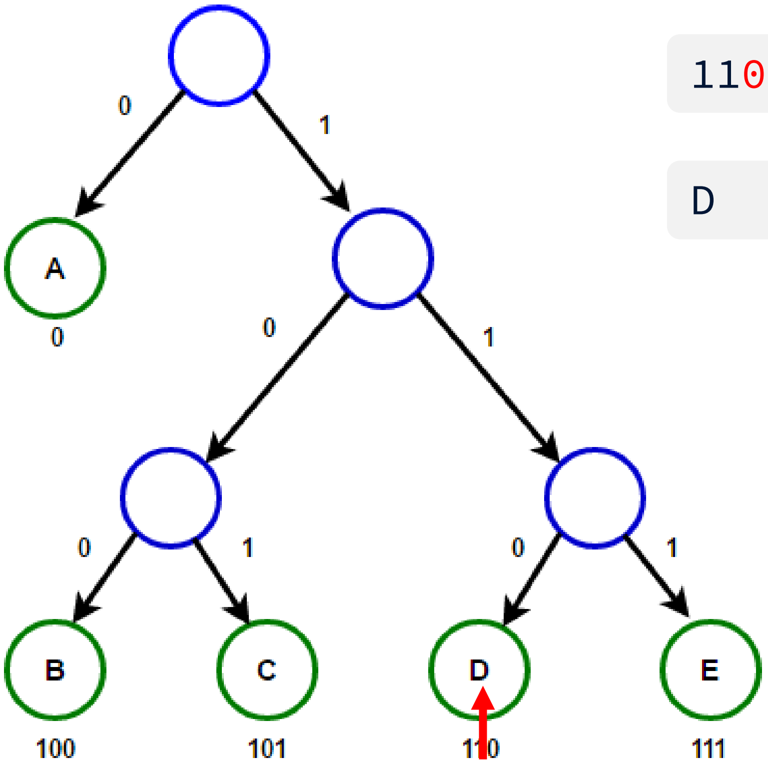
11011101101100111100101010001 10...

哈夫曼解码过程




```
110111011011001111001010100011 0...
```

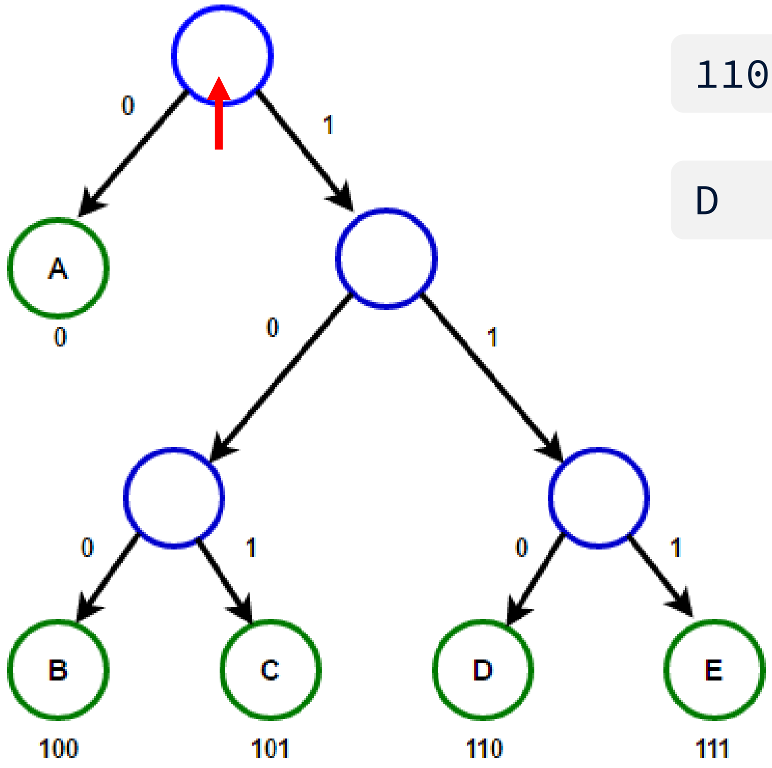
哈夫曼解码过程




```
11011101101100111100101010001 10...
```

D

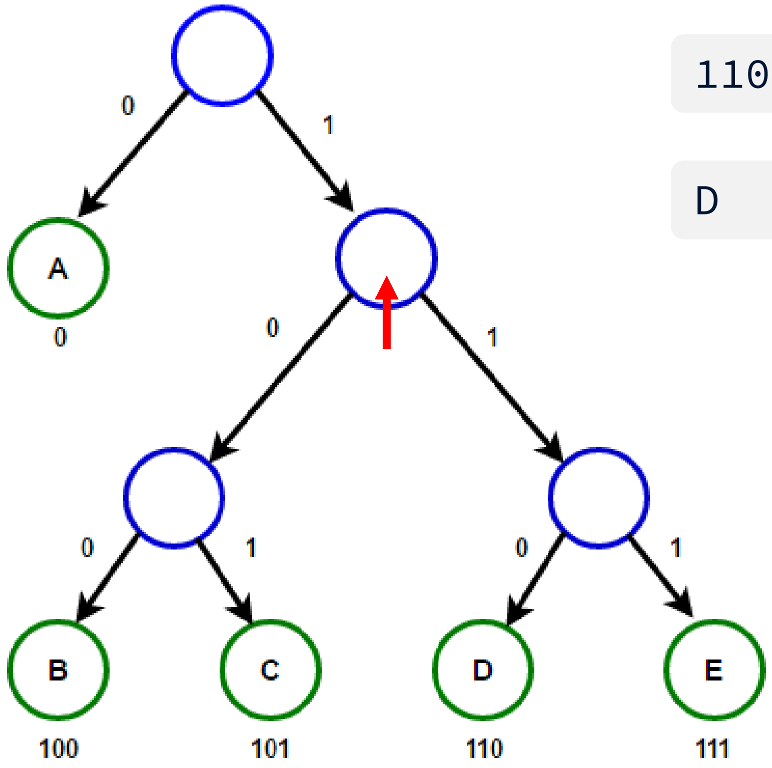
哈夫曼解码过程



11011101101100111100101010001 10...

D

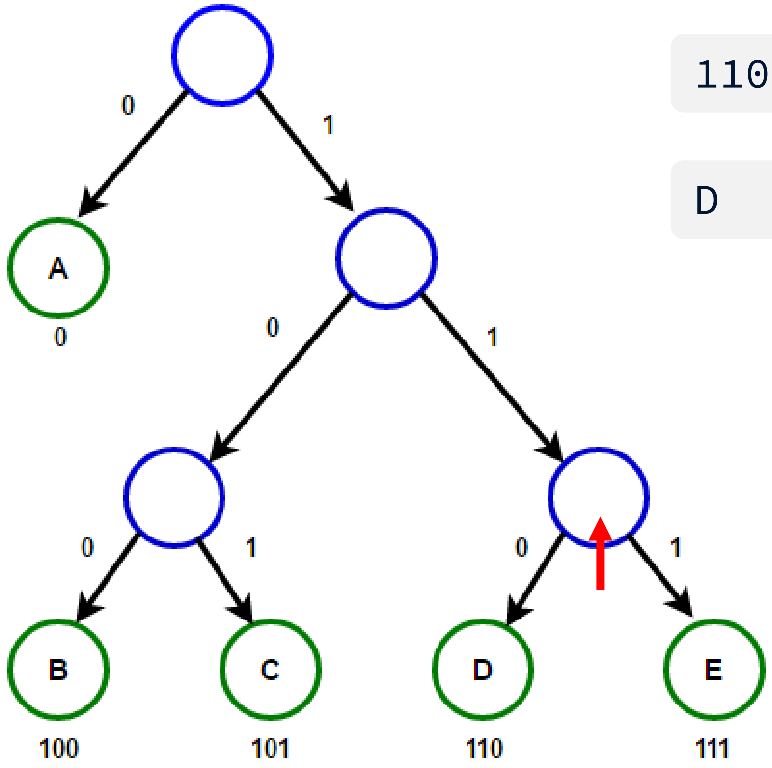
哈夫曼解码过程




```
110111011011001111001010100011 0...
```

D

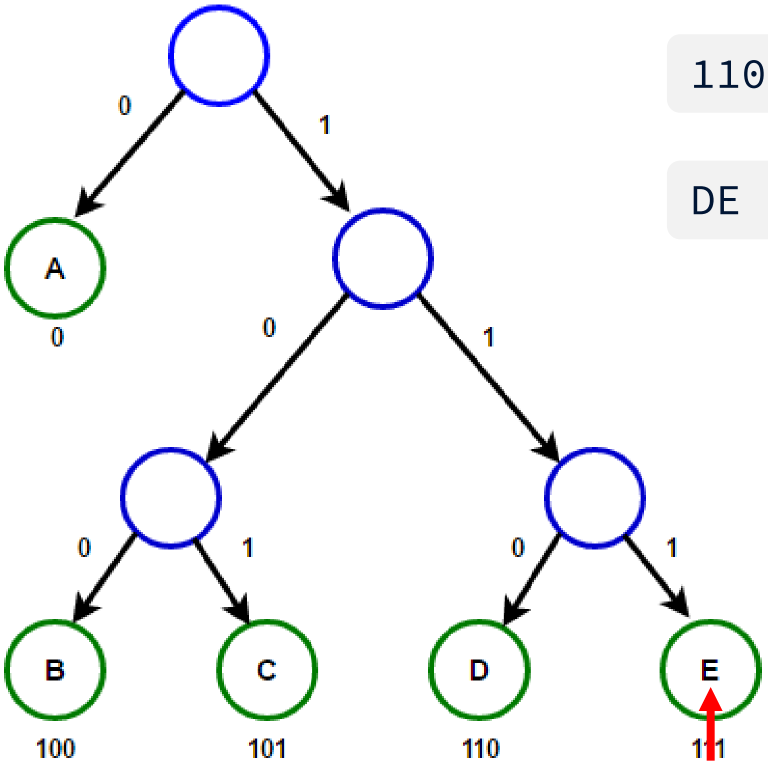
哈夫曼解码过程



1101110110110011110010101000110...

D

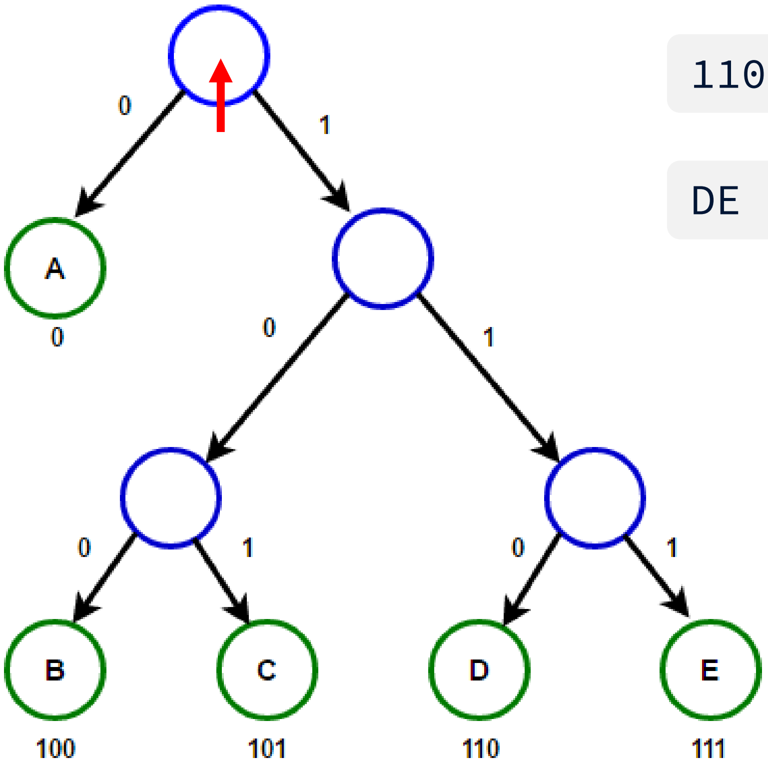
哈夫曼解码过程



1101110110110011110010101000110...

DE

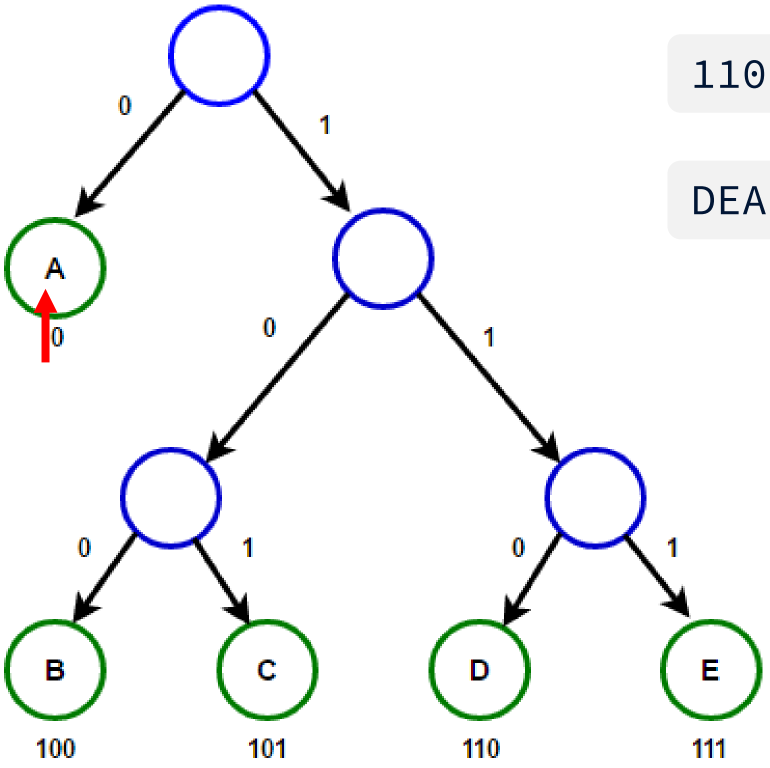
哈夫曼解码过程



1101110110110011110010101000110...

DE

哈夫曼解码过程



110111011011001111001010100011O...

DEA

哈夫曼的压缩效率

```
D:\>huffman_zipper ser.log ser_c.logc zip
Zipper 0.002! Author: Tobias
Welcome! This program is committed to lossless file compression and decompression.
Program processing...
Program terminated in 0.465s.

D:\>huffman_zipper ser_c.logc ser_decompressed.log unzip
Zipper 0.002! Author: Tobias
Welcome! This program is committed to lossless file compression and decompression.
Program processing...
Program terminated in 0.931s.
```

65.40%

ser.log	10,073 KB
ser_c.logc	6,588 KB
ser_decompressed.log	10,073 KB

LZ 算法和哈夫曼编码

- LZ 直接“引用”重复的字符（串），而哈夫曼编码是在字符维度上考虑减少编码的长度的。
- LZ 压缩对包含大量重复数据的文件/数据效果最好。但对于根本不包含任何重复性信息的文件/数据效果不好。
- 由于所有二进制代码的长度不同，而且所有的代码都是串联的，因此解码软件很难检测到编码数据是否损坏。这可能会导致错误的解码和随后的错误输出。

谢谢！

陈宇飞 Tue, 04/21/2020 7:14:19 PM

布置完这个作业，是不是就不用上课了？我觉得怪好的呢

文件压缩算法简介

林日中 (1951112)

2022年4月8日

游程编码（Run-Length Encoding）

- 游程编码（RLE）是一种简单的无损数据压缩形式，在连续多次出现相同数值的序列上运行。它对序列进行编码，存储一个值和它对应的计数。

```
WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW
```

```
-> 12W1B12W3B24W1B14W
```

```
ABCDEFG -> 1A1B1C1D1E1F1G
```

LZ算法三元组的一种存储结构

`<0, 0, C>`

0 (1 bit)	codeword (8 bits)

`<3, 1, C>`

1 (1 bit)	offset (10 bits)	length of match (5 bits)	codeword (8 bits)

哈夫曼节点

```
struct Node
{
    char c = '\0';
    int freq = 0;
    char *code = NULL;
    struct Node *left_child = NULL;
    struct Node *right_child = NULL;
};

bool is_leaf(struct Node *p)
{
    return !p->left_child && !p->right_child;
}
```

std::**bitset** <bitset>

类模板 bitset<N> 表示一个固定大小的N位序列。比特集可以通过标准的逻辑运算符进行操作，并可以在字符串和整数之间进行转换。

```
void bitset_demo()
{
    std::bitset<8> test("00110101");
    std::cout << test << std::endl; // 00110101
    test.set();
    std::cout << test << std::endl; // 11111111
    test.reset();
    std::cout << test << std::endl; // 00000000
    test.set(3); test.set(4); test.set(5); test.set(6);
    std::cout << test << std::endl; // 01111000
    std::cout <<       test.to_ulong() << std::endl; // 120
    std::cout << (char)test.to_ulong() << std::endl; // x
}
```

std::**priority_queue** <queue>

priority_queue 是一个容器适配器，它提供最大（默认）元素的恒定时间查找，代价是对数插入和提取。

```
template<
    class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

std::**unordered_map** <unordered_map>

unordered_map 是一个关联容器，包含具有唯一键的键值对。元素的搜索、插入和移除具有平均恒定的时间复杂性。

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

```cpp
void unordered_map_demo()
{
    // Create an unordered_map of three strings (that map to strings)
    std::unordered_map<std::string, std::string> u = {
        {"RED", "#FF0000"},
        {"GREEN", "#00FF00"},
        {"BLUE", "#0000FF"}};

    std::cout << "Iterate and print keys and values of unordered_map:\n";
    for (const std::pair<std::string, std::string> &n : u)
    {
        std::cout << "Key:[" << n.first << "]\tValue:[" << n.second << "]\n";
    }

    // Add two new entries to the unordered_map
    u["BLACK"] = "#000000";
    u["WHITE"] = "#FFFFFF";

    std::cout << "Iterate and print keys and values of unordered_map,\n"
                 "after two new entries added:\n";
    for (const std::pair<std::string, std::string> &n : u)
    {
        std::cout << "Key:[" << n.first << "]\tValue:[" << n.second << "]\n";
    }

    std::cout << "Output values by key:\n";
    std::cout << "The HEX of color RED:\t[" << u["RED"] << "]\n";
    std::cout << "The HEX of color BLACK:\t[" << u["BLACK"] << "]\n";

    // Erase three entries in the unordered_map
    u.erase("BLACK");
    u.erase("BLUE");
    u.erase("RED");

    std::cout << "Iterate and print keys and values of unordered_map,\n"
                 "after three entries erased:\n";
    for (const std::pair<std::string, std::string> &n : u)
    {
        std::cout << "Key:[" << n.first << "]\tValue:[" << n.second << "]\n";
    }
}
```

```
Iterate and print keys and values of unordered_map:
Key:[GREEN]     Value:[#00FF00]
Key:[BLUE]      Value:[#0000FF]
Key:[RED]       Value:[#FF0000]
Iterate and print keys and values of unordered_map,
after two new entries added:
Key:[WHITE]     Value:[#FFFFFF]
Key:[RED]       Value:[#FF0000]
Key:[BLACK]     Value:[#000000]
Key:[BLUE]      Value:[#0000FF]
Key:[GREEN]     Value:[#00FF00]
Output values by key:
The HEX of color RED:   [#FF0000]
The HEX of color BLACK: [#000000]
Iterate and print keys and values of unordered_map,
after three entries erased:
Key:[WHITE]     Value:[#FFFFFF]
Key:[GREEN]     Value:[#00FF00]
```