

# 压缩文件大作业分享

---

基于哈夫曼编码

1951247 钟伊凡

济勤九班

# 目录

---

1. 哈夫曼编码介绍
2. 算法实现
3. 时间优化

# 哈夫曼编码介绍

---

预备知识:

贪心算法

二叉树

前缀编码

# 贪心算法（Greedy Algorithms）

---

每一步行动总是按某种指标选取最优的操作来进行，该指标只看眼前，并不考虑以后可能产生的影响。

每一步都只考虑当前最优。

因此使用贪心需要证明正确性。

全局最优解包含局部最优解（**最优子结构性质**）

按某种决策方式，不会丢失最优解（**贪心选择性质**）

（可参考《算法导论》）

# 二叉树

---

完全二叉树：若设二叉树的深度为 $h$ ，除第 $h$ 层外，其它各层 ( $1 \sim h-1$ ) 的节点数都达到最大个数，第 $h$ 层所有的节点都连续集中在最左边，这就是完全二叉树。

满二叉树：

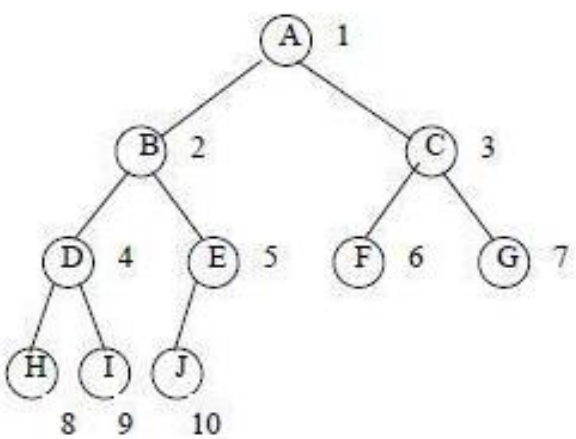
国内：

除最后一层无任何子节点外，每一层上的所有节点都有两个子节点的二叉树。

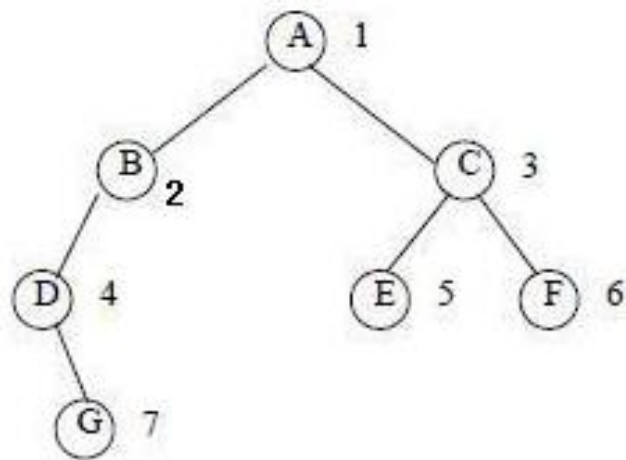
国外：

一个节点要么为叶子节点，要么同时具有左右孩子。（我们后面按这种定义）

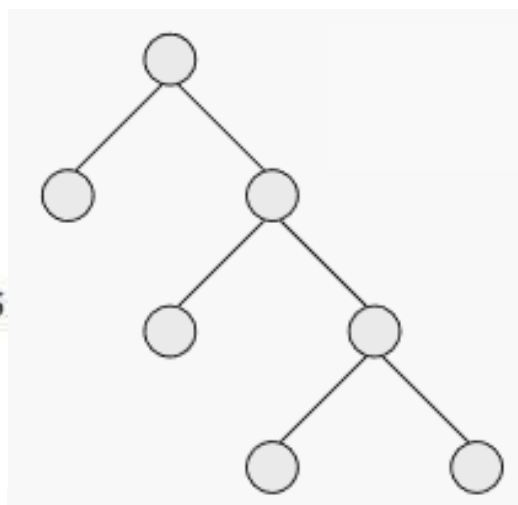
# 二叉树



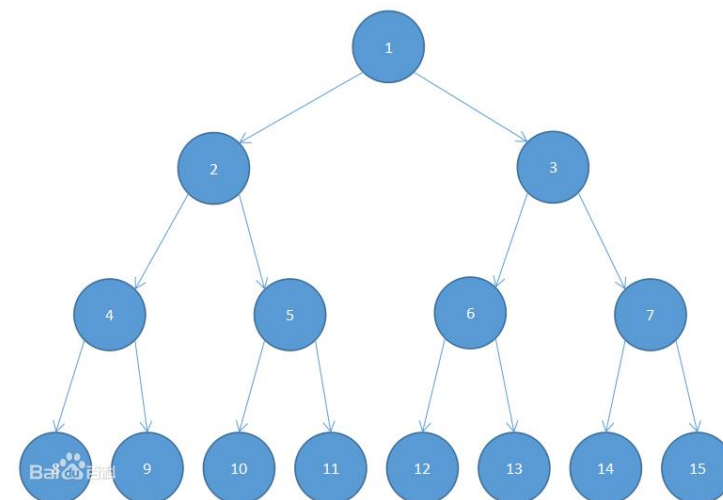
完全二叉树



不完全二叉树



满二叉树（国际上）



满二叉树（国内定义）

# 前缀编码

---

指对字符集进行编码时，要求字符集中任一字符的编码都不是其它字符的编码的前缀。

这和直观上从前往后读文本是相符的。

# 哈夫曼编码是怎样想到的

---

ASCII码是确定的，对所有ASCII字符集以内的文件都能够完成编码。

现在考虑动态的编码，具体问题具体分析。

我们现在手上有一个文本，字符频数已知，有没有更有效（更节省空间）的编码方案？

很自然地，希望频率比较高的字符，编码长度短；频率比较低，编码长度长。

结论：前缀编码都可以建成编码树。（对于字符，显然是二叉编码树）

于是，我们可以考虑对于二叉树，怎样放置字符能够使编码总长最短。



# 最优编码问题的正式定义

---

## 最优前缀码问题

### Optimal Prefix Code Problem

#### 输入

- 字符数 $n$ 以及各个字符的频数 $F = \langle f_1, f_2, \dots, f_n \rangle$

#### 输出

- 解析结果唯一的二进制编码方案 $C = \langle c_1, \dots, c_n \rangle$ ，令

$$\min \sum_{i=1}^n |c_i| \cdot f_i$$

$|c_i|$ 为字符 $i$ 的编码二进制串长度

# 稍加思考，可证明如下结论：

---

## 1. 字符都是叶子节点

如果不是叶子节点，则为根节点。那么它的编码将成为其后代的编码的前缀，矛盾。

## 2. 编码树为满二叉树

如果不是满二叉树，则存在某节点只有左儿子或右儿子。直接省去这一层，仍然可以编码，而且总长变短。

# 开始建树

---

大致在脑中能想象出编码树长什么样。（一棵满二叉树）

考察最底层。易证最低频的字符一定在这层（否则直接交换可以更优）。把这两个字符看成一个整体，总频数为频数之和，可作为一整个节点和其余节点比较。如此循环，每循环一次就减少一个根节点，最后只剩下一个根节点时建树完成。

# 伪代码

---

HUFFMAN( $C$ )

1  $n = |C|$

2  $Q = C$

3 **for**  $i = 1$  **to**  $n - 1$

4     allocate a new node  $z$

5      $z.left = x = \text{EXTRACT-MIN}(Q)$

6      $z.right = y = \text{EXTRACT-MIN}(Q)$

7      $z.freq = x.freq + y.freq$

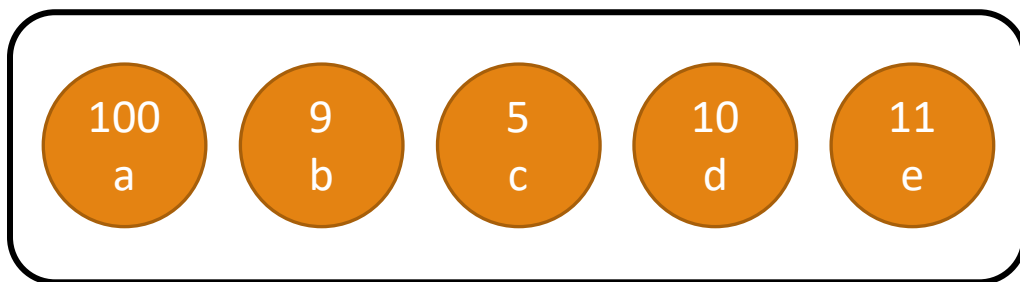
8      $\text{INSERT}(Q, z)$

9 **return**  $\text{EXTRACT-MIN}(Q)$

# 建立Huffman树

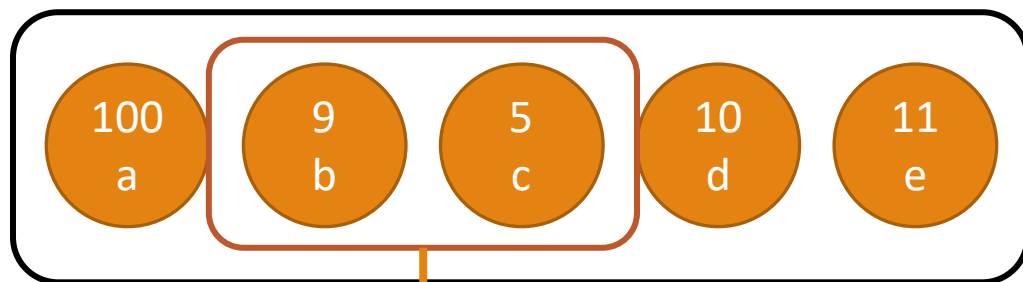
---

待建立结点

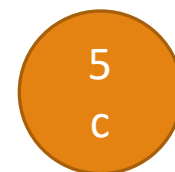
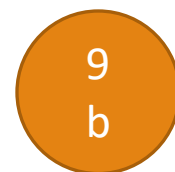


# 建立Huffman树

待建立结点

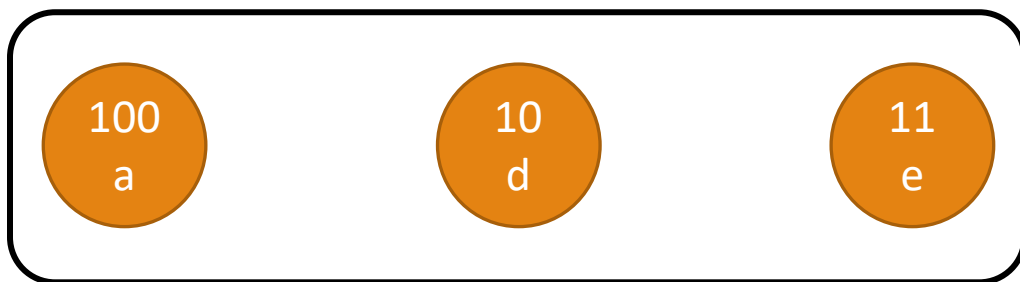


选择最小的两个

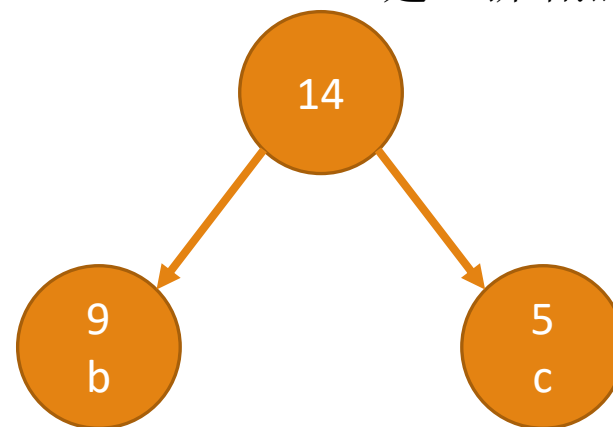


# 建立Huffman树

待建立结点

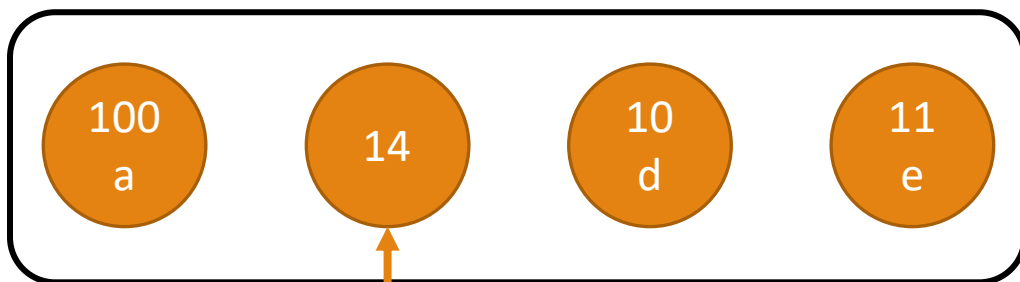


建立新结点连接两个结点

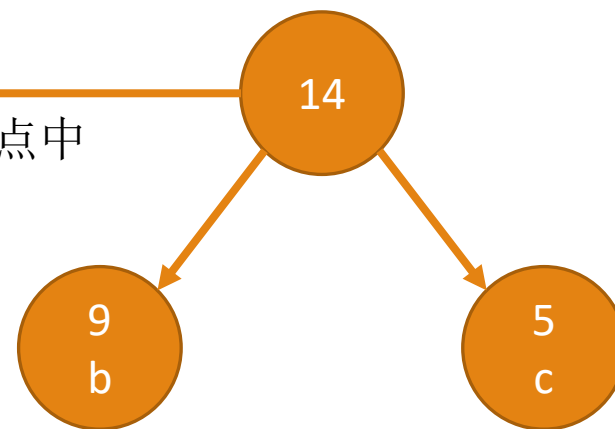


# 建立Huffman树

待建立结点



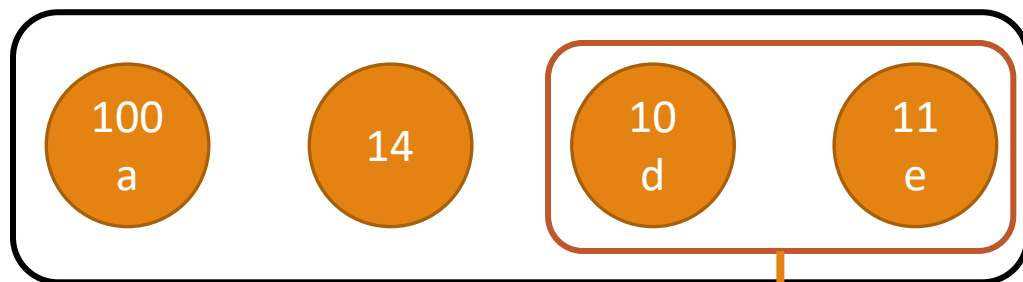
将新结点加入到待建立结点中



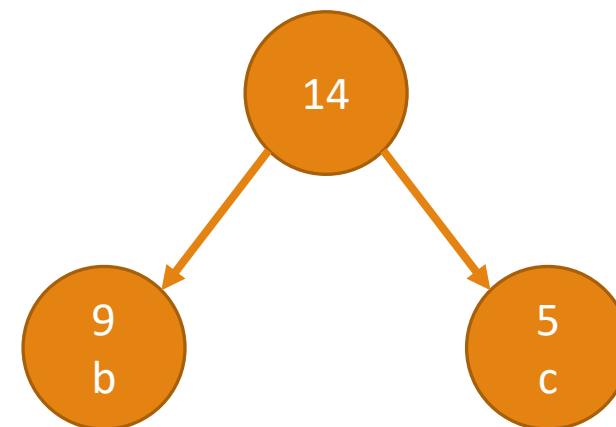
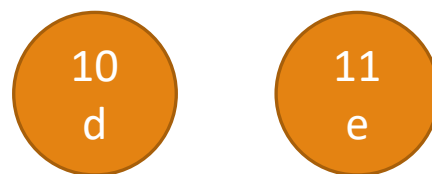


# 建立Huffman树

待建立结点



选择最小的两个

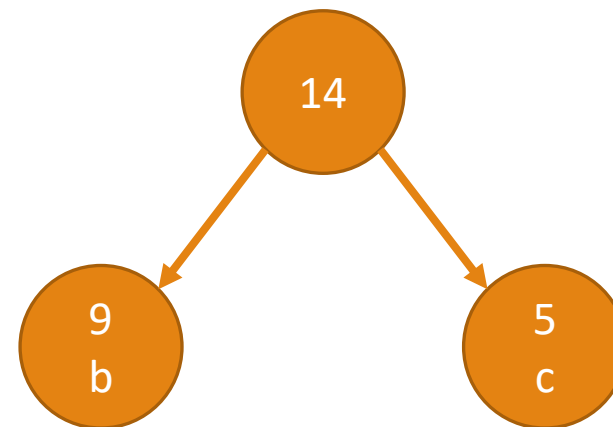
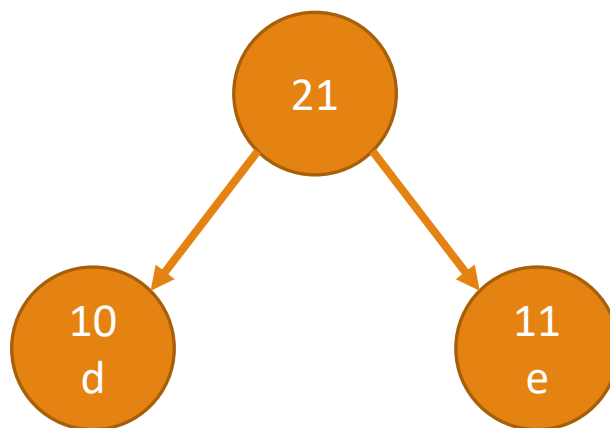


# 建立Huffman树

待建立结点

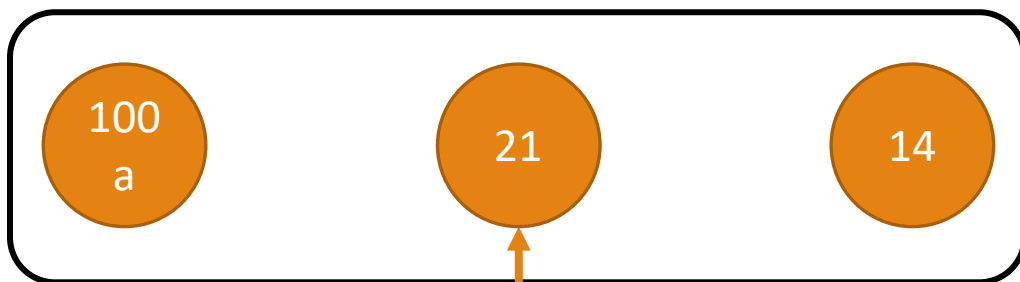


建立新结点连接两个结点

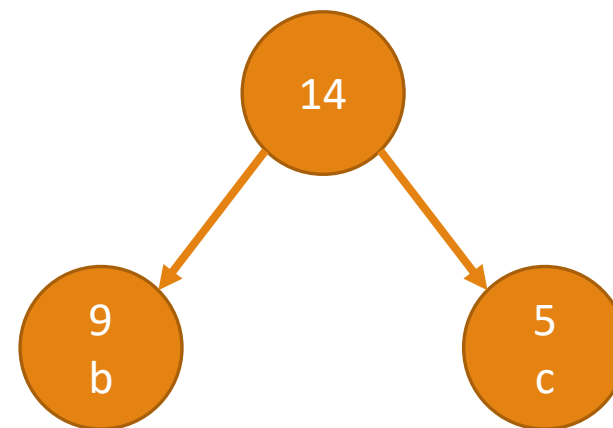
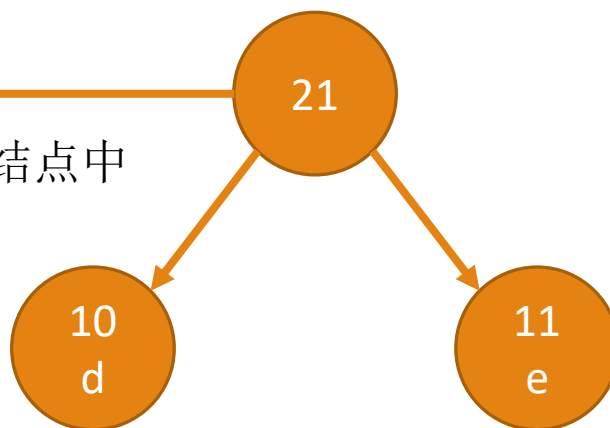


# 建立Huffman树

待建立结点

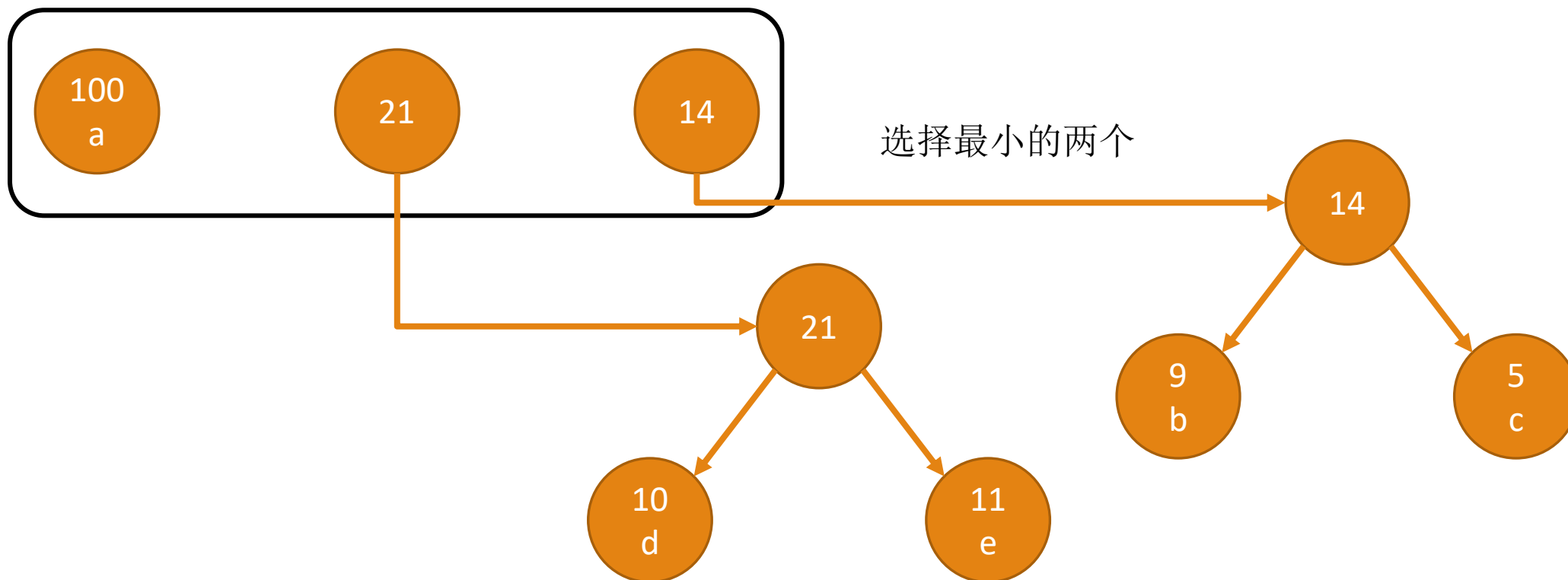


将新结点加入到待建立结点中



# 建立Huffman树

待建立结点

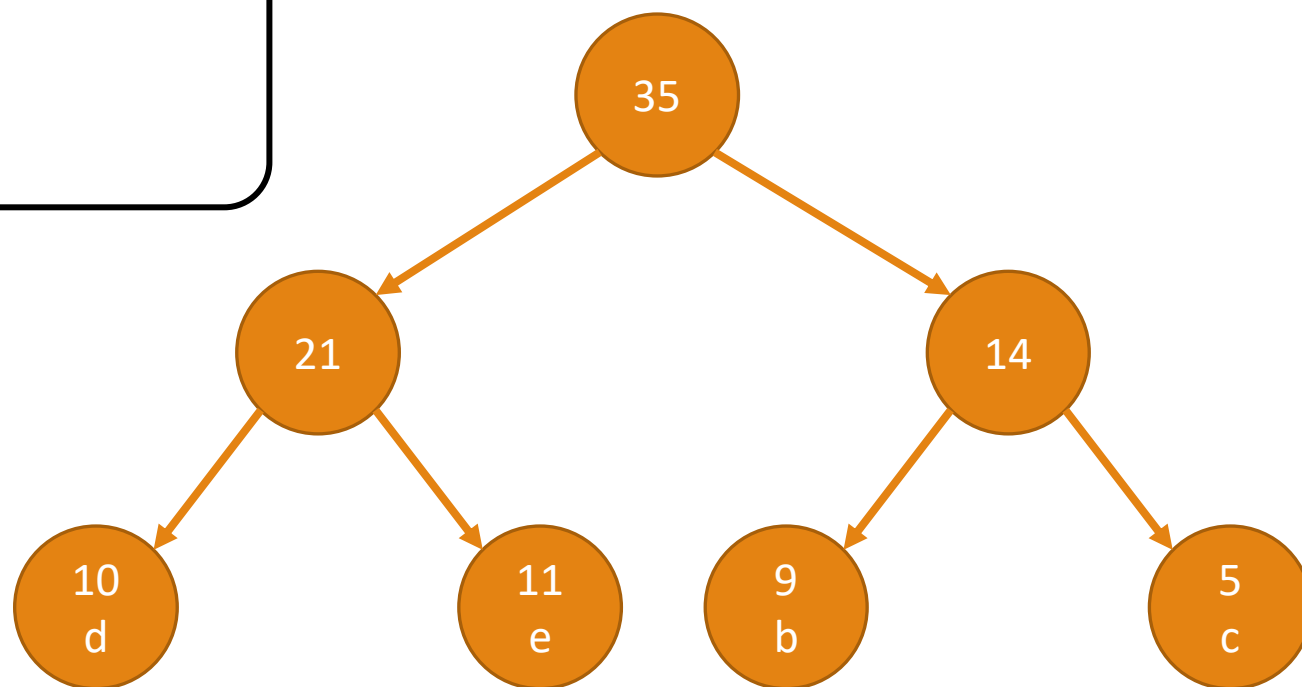


# 建立Huffman树

待建立结点



建立新结点连接两个结点

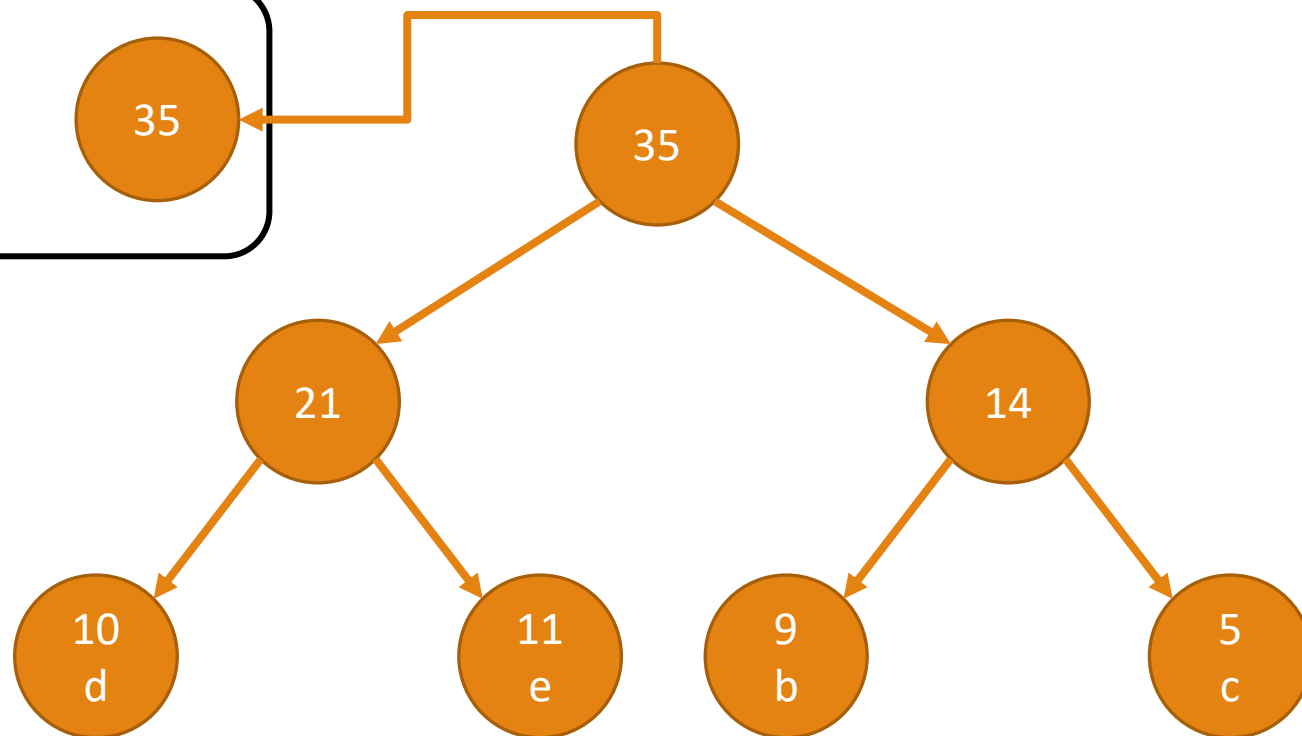


# 建立Huffman树

待建立结点

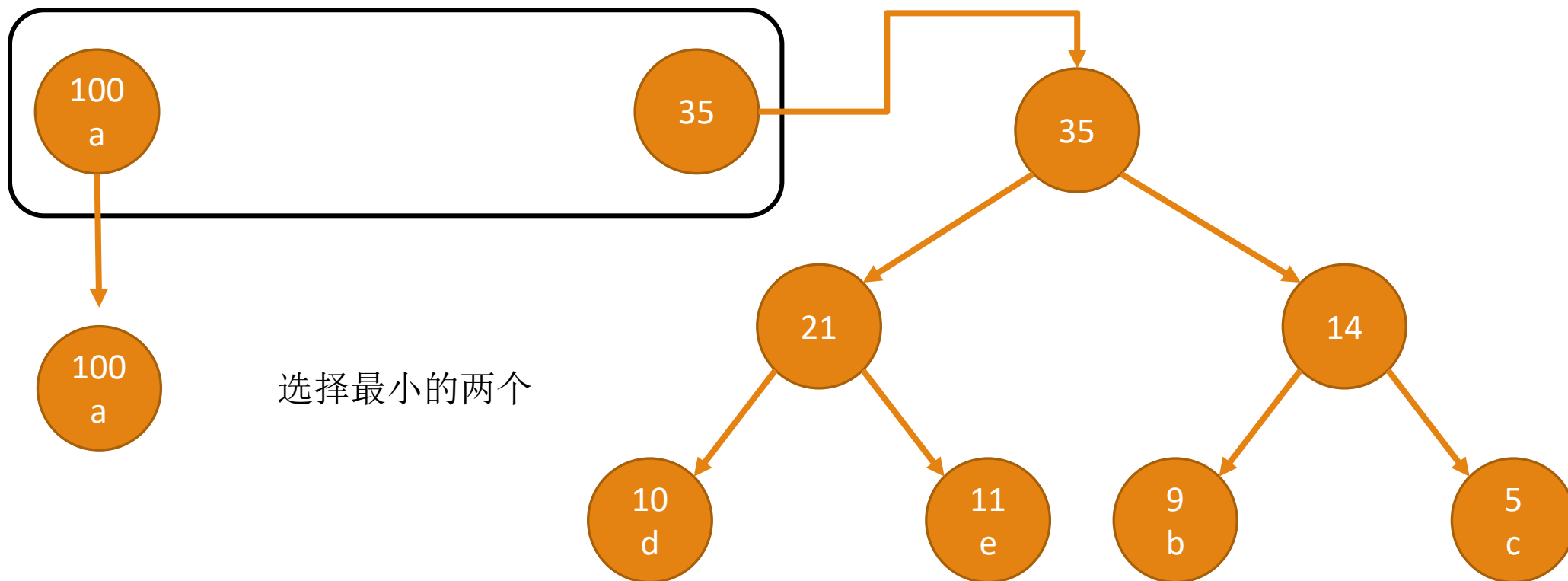


将新结点加入到待建立结点中



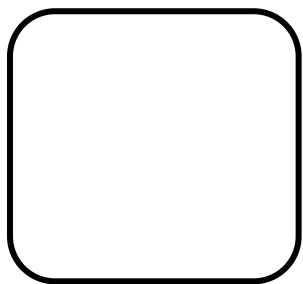
# 建立Huffman树

待建立结点

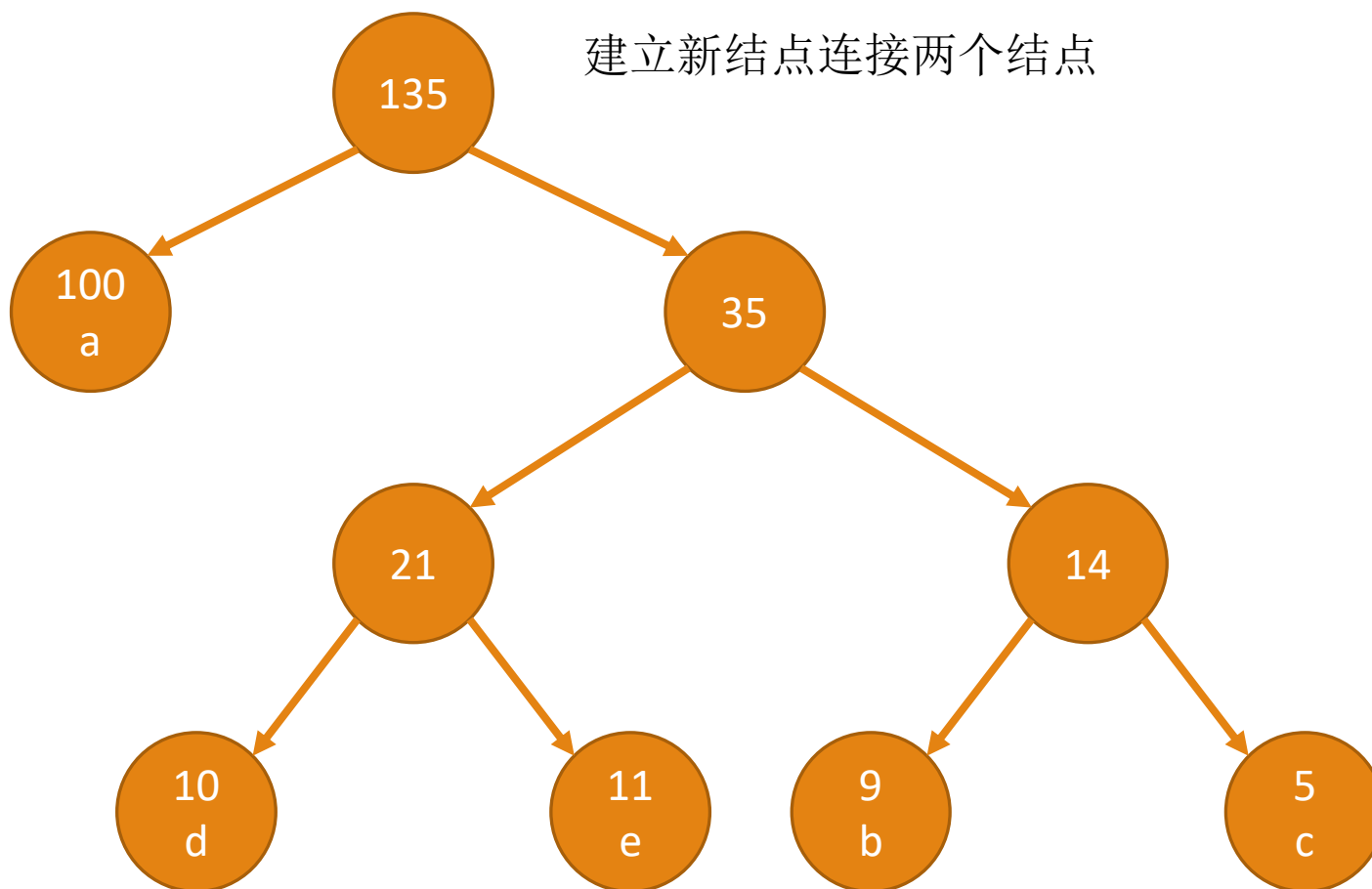


# 建立Huffman树

待建立结点



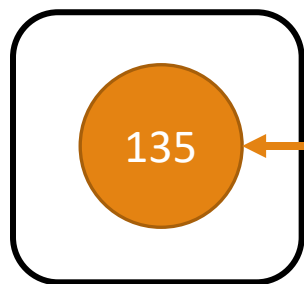
建立新结点连接两个结点



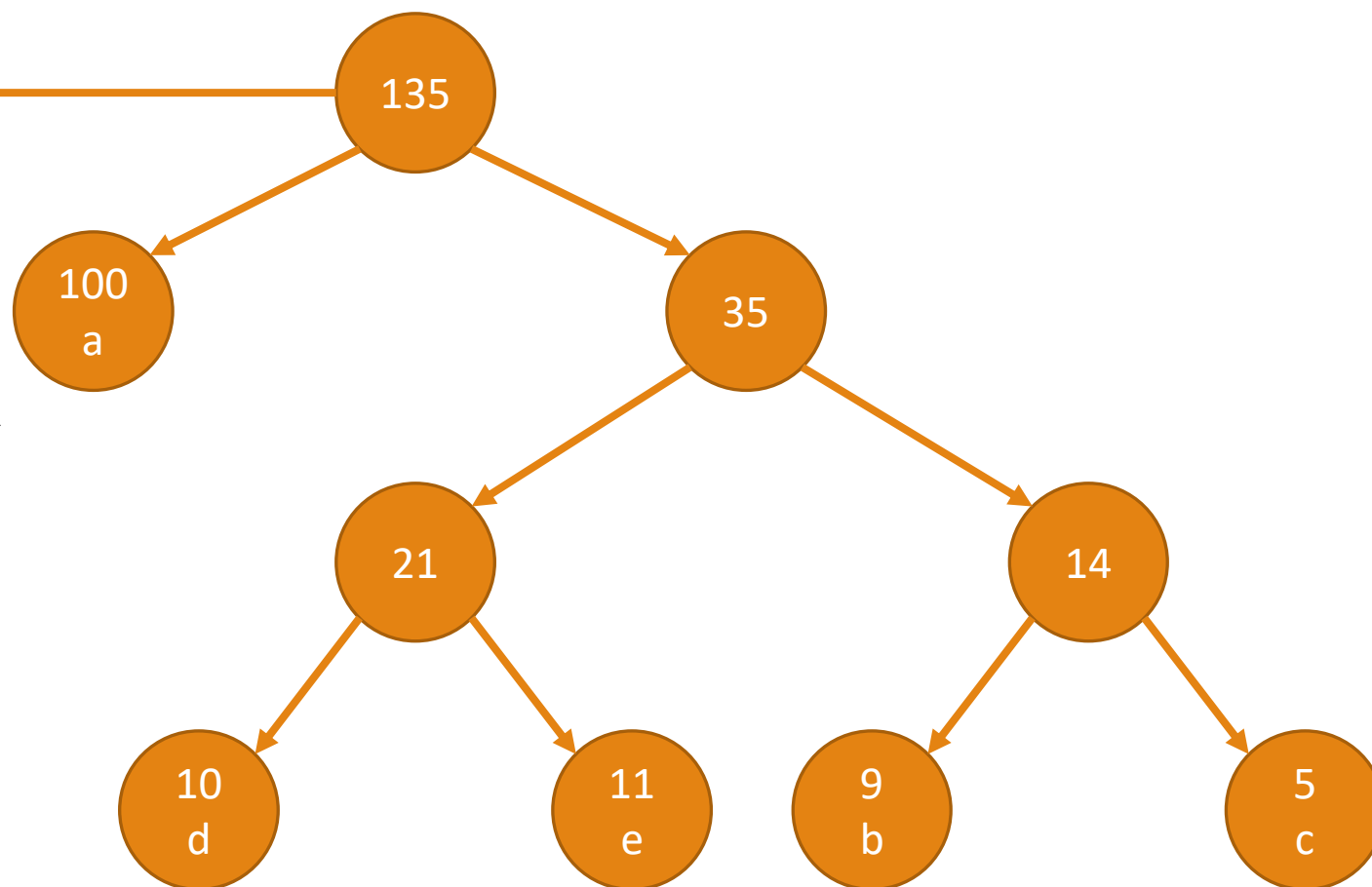


# 建立Huffman树

待建立结点

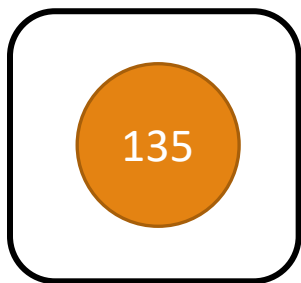


将新结点加入到待建立结点中

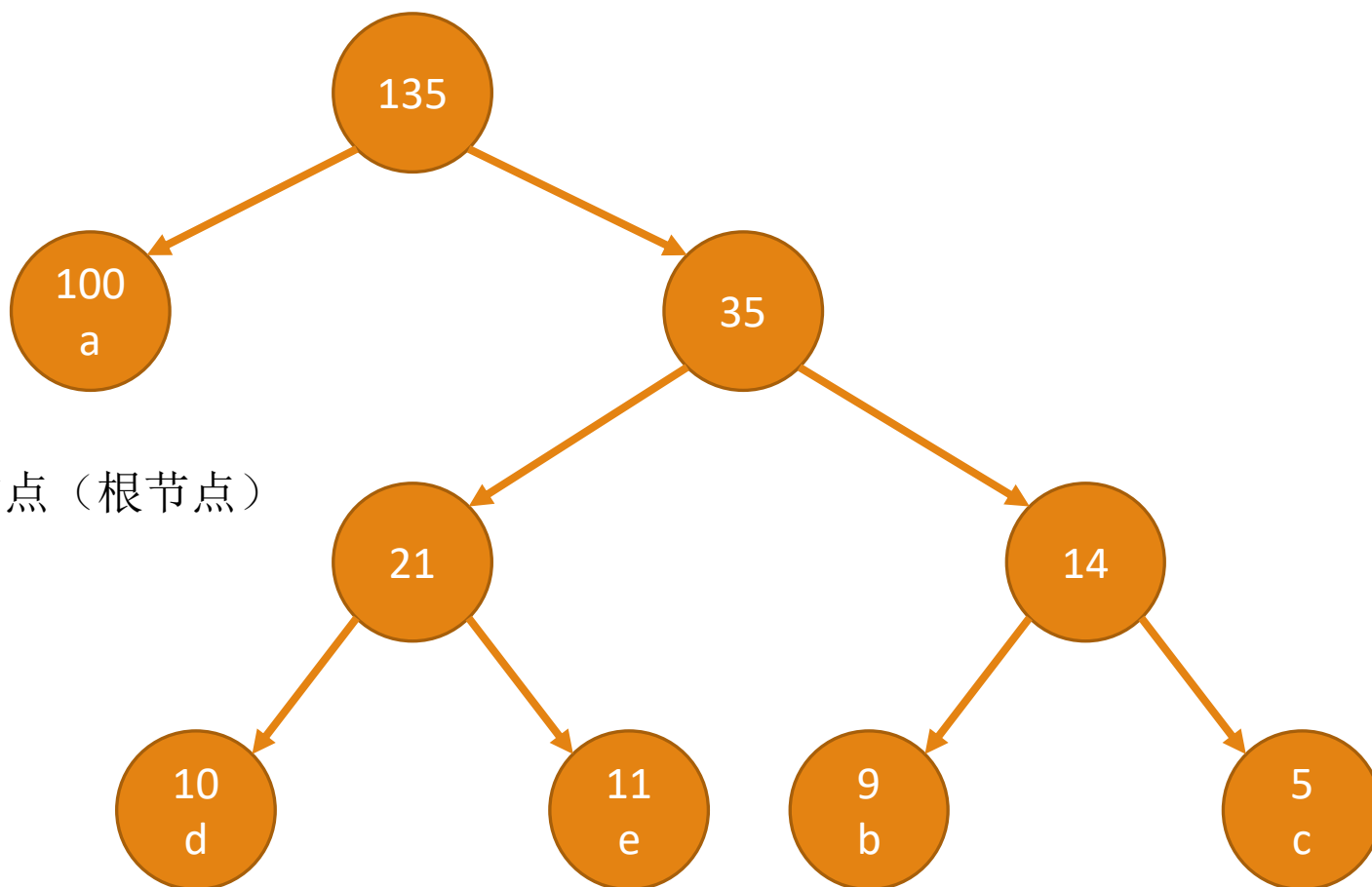


# 建立Huffman树

待建立结点



此时待建立结点只剩下一个  
那么剩下的这一个一定是头结点（根节点）  
建立完成



# 哈夫曼树示例二

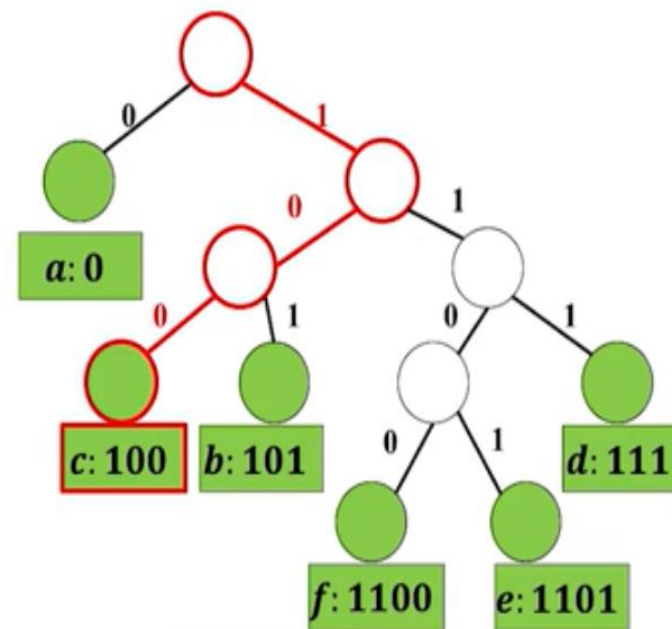
按照左为0，右为1的规则（规则可自定，因此编码不唯一，但有限、可列）在树上分配编码，满足了要求：

1. 任何字符的编码不是其他字符前缀码
2. 总长最短

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
编码方式1	0	101	100	111	1101	1100

- 编码方式1：对"cba"编码

- 编码：cba → 10011000
- 解码：10011000 → c



# 第二部分：算法实现

---

1. 统计字符

2. 建树

3. 编码

4. 存树

5. 压缩

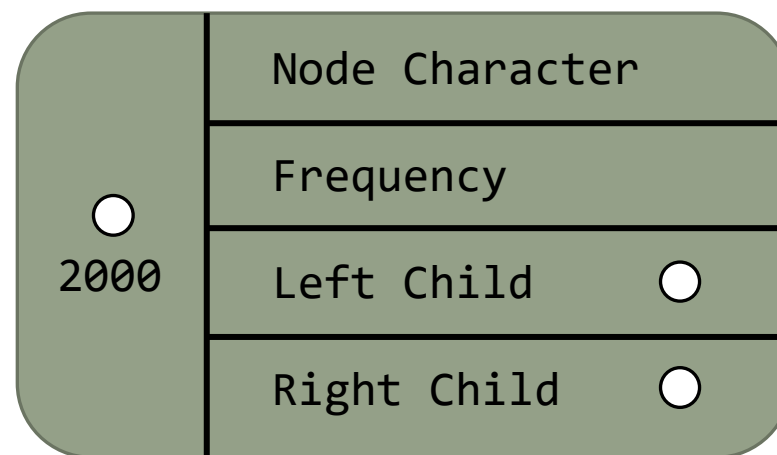
1. 读出哈夫曼树

2. 解压缩

# 结点数据结构

---


```
struct HuffNode
{
    char syb;
    int freq;
    HuffNode *left, *right;
    char hcode[20];
};
```



# 统计字符

---

字符序列

aabbadcccddaabbcdacbdcababc.....

扫描窗口

扫描到的结果(字符-出现次数): a - 1

# 统计字符

---

字符序列

a**a**bbadcccddaabbcdacbdcababc.....

扫描窗口

扫描到的结果(字符-出现次数): a - 2

# 统计字符

---

字符序列

aab**bad**ccddaabbcdacbdcababc.....

扫描窗口

扫描到的结果(字符-出现次数): a-2 b-1



# 统计字符

---

字符序列

aabbadcccddaabbcdacbdcababc.....



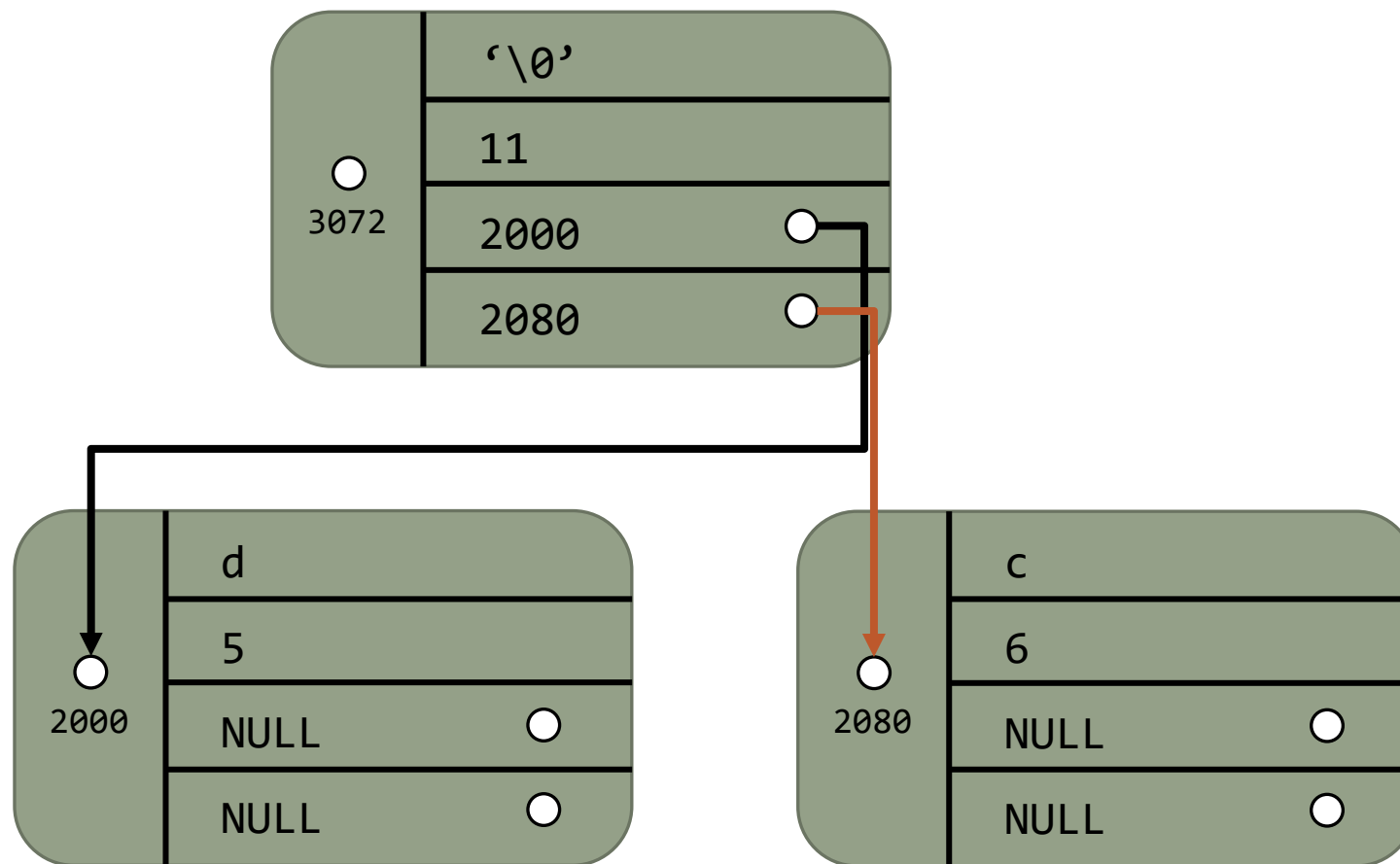
扫描窗口

扫描到的结果(字符-出现次数): a-7 b-6 c-6 d-5

# 建树

直接按照前面伪代码的思路写即可。从节点数组中寻找最小的两个，建立新节点，加入树中。

字符	频次
a	7
b	6
c	6
d	5



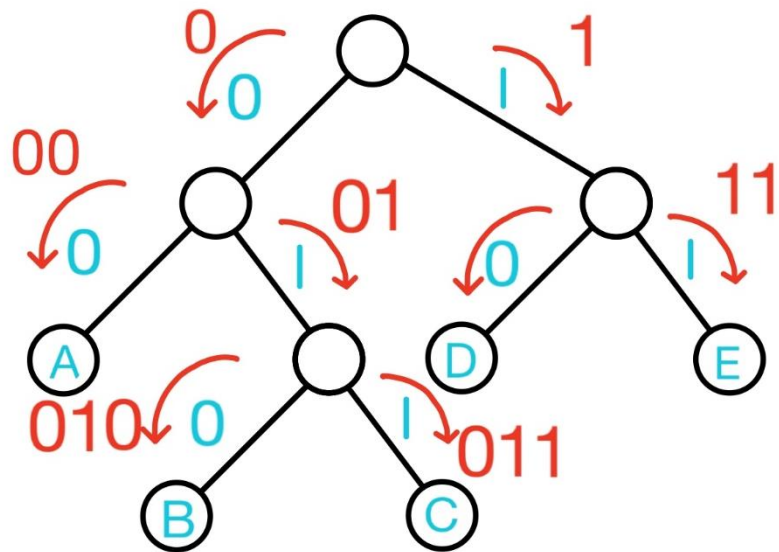
# 编码

到这里已经建好了哈夫曼树，此时保存根节点，从它即可到达任意树中节点。

对Huffman树的搜索我们可以采用递归的方式进行搜索。（深度优先搜索）

在实际编码的时候我们可以不必每次都去树中寻找编码，我们可以将树展平，即将所有的叶子节点存入字典或者数组中。

a 0 0  
b 0 1 0  
c 0 1 1  
d 1 0  
e 1 1



# ser.log文件的哈夫曼编码

字符	频数	编码	编码长度
*	1638828	110	3
.	691359	1010	4
0	535099	0101	4
	530376	0011	4
/	443852	11110	5
+	419594	11101	5
e	345395	10011	5
i	294320	01111	5
3	284062	01110	5
1	266884	01001	5
l	245576	00011	5
t	239817	00001	5
o	228970	111111	6
a	192737	111000	6
6	186003	101110	6
s	181249	101101	6
2	178290	101100	6
p	167496	100101	6
-	165237	100100	6
5	156279	100001	6
r	136358	011001	6
7	134319	010001	6
4	127767	001011	6

d	124969	001001	6
h	123550	001000	6
:	119192	000100	6
f	118391	000001	6
W	111882	000000	6
n	111643	1111101	7
T	105676	1110011	7
m	105633	1110010	7
8	89314	1011110	7
9	82453	1000111	7
c	81301	1000110	7
M	81155	1000101	7
k	72236	0110111	7
;	71451	0110110	7
)	69143	0110100	7
(	69143	0110001	7
u	65597	0110000	7
b	64369	0100000	7
K	62773	0010100	7
G	61554	0001011	7

S	53033	11111000	8
w	49723	10111111	8
z	39081	10001000	8
\n	37899	10000011	8
E	37389	10000010	8
A	37106	10000001	8
L	36293	10000000	8
N	35366	01101011	8
x	33840	01101010	8
H	33126	01000011	8
C	31958	00101011	8
_	31896	00101010	8
,	31153	00010101	8
O	30277	00010100	8

# 长于8的字符编码

---

j	29285	111110011	9
g	23910	101111100	9
P	19825	100010011	9
=	16155	010000101	9
?	15329	1111100101	10
y	12274	1011111011	10
v	10185	1000100101	10
B	7613	0100001000	10
F	7042	11111001001	11
q	6446	11111001000	11
Q	6122	10111110101	11
I	5736	10111110100	11
X	4736	10001001001	11
V	4466	01000010011	11
D	2332	100010010001	12
U	2139	100010010000	12
Y	1790	010000100100	12
R	1586	0100001001011	13
%	280	01000010010101	14
#	52	0100001001010011	16
J	40	0100001001010010	16
!	24	0100001001010000	16
Z	20	01000010010100011	17
&	8	01000010010100010	17

# 存树

---

重要问题：

压缩很嗨皮，解压的时候需要知道编码情况。

比较方便省事，也不太占空间，不太花时间的的方法是，直接把树整个二进制写进去，解压的时候直接二进制读出来

复习：

## 13.4.文件操作与文件流

### 13.4.5.对二进制文件的操作

★ 用ASCII文件的字符方式进行操作(按字节读写)

★ 用read/write进行操作

文件流对象名.read(内存空间首指针, 长度);

从文件中读长度个字节，放入从首指针开始的空间中

文件流对象名.write(内存空间首指针, 长度);

将从首指针开始的连续长度个字节写入文件中

# 压缩

---

紧跟在输出的树后面把文件按新编码写到新文件中。

思路：

从原文件读入字符（`charget`），按照新编码设置待输出字符（`charput`）的二进制位，到了八位就输出这个字符到新文件。

注意，最后一个字符不一定每一位都有效，因此需要在最后一个字符后面再输出一下它的有效位数。

# 输出

---

字符	编码
a	00
b	010
c	011
d	10

字符序列

编码结果: 00

aabbadcccddaabbcdacbdcababc.....

扫描窗口



# 输出

---

字符	编码
a	00
b	010
c	011
d	10

字符序列

a**a**bbadcccddaabbcdacbdcabc.....

扫描窗口

编码结果:    00 00

# 输出

---

字符	编码
a	00
b	010
c	011
d	10

字符序列

a**a**badcccddaabbcdacbdcababc.....

扫描窗口

编码结果:    00 00 010

# 输出

---

字符	编码
a	00
b	010
c	011
d	10

字符序列

aabbadccddaabbcdacbdcababc.....



扫描窗口

编码结果: 000001001000100110110111010000001001001110000110101001100010011...

# 输出

字符	编码
a	00
b	010
c	011
d	10

字符序列

aabbadccddaabbcdacbdabc.....

扫描窗口

按字节分组: 00000100 10001001 10110111 01000000 10010011 10000110 10100110 0010011...

分组后按字节(char)写入文件即可

# 关闭文件

---

完成压缩

压缩率65.01%

9秒

# 解压缩

---

第一步读出压缩文件中的各种辅助解码信息。

文件头部：哈夫曼树的信息

文件尾部：最后一个字符的有效位数

打开文件直接读，同样的程序，完全相同的结构体

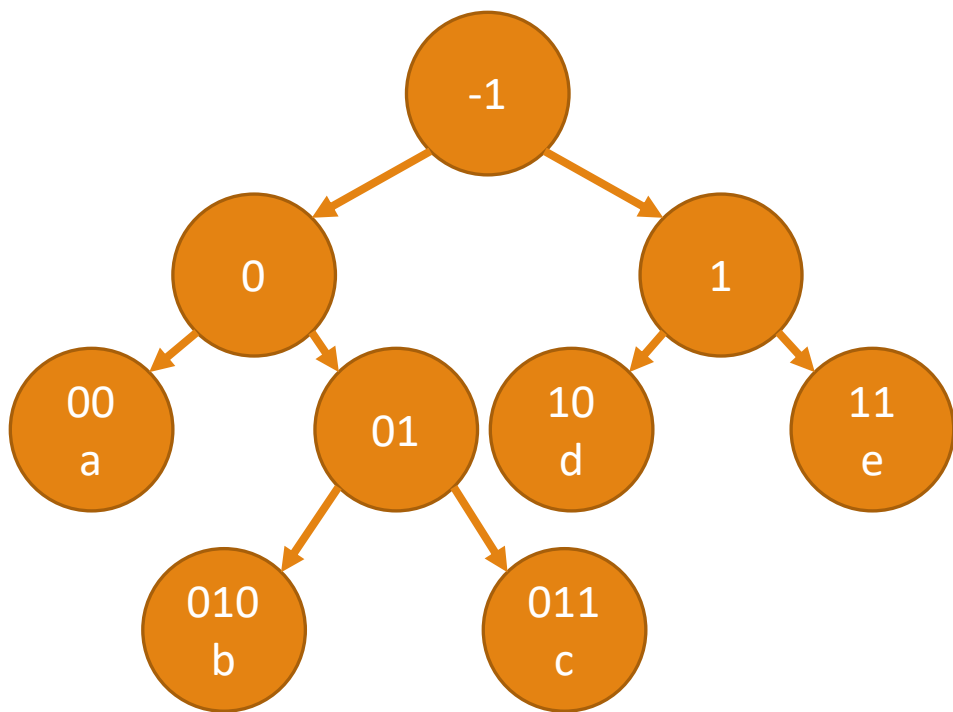
# 逐字符解压缩

---

是压缩的逆过程，思路很相似

读入一个字符，把它理解成8位01串，从头开始，如果是0就进左儿子，如果是1就进右儿子。如果到了叶子节点，就直接输出叶子的字符，并退回根节点。

# 逐字符解压缩



位序列

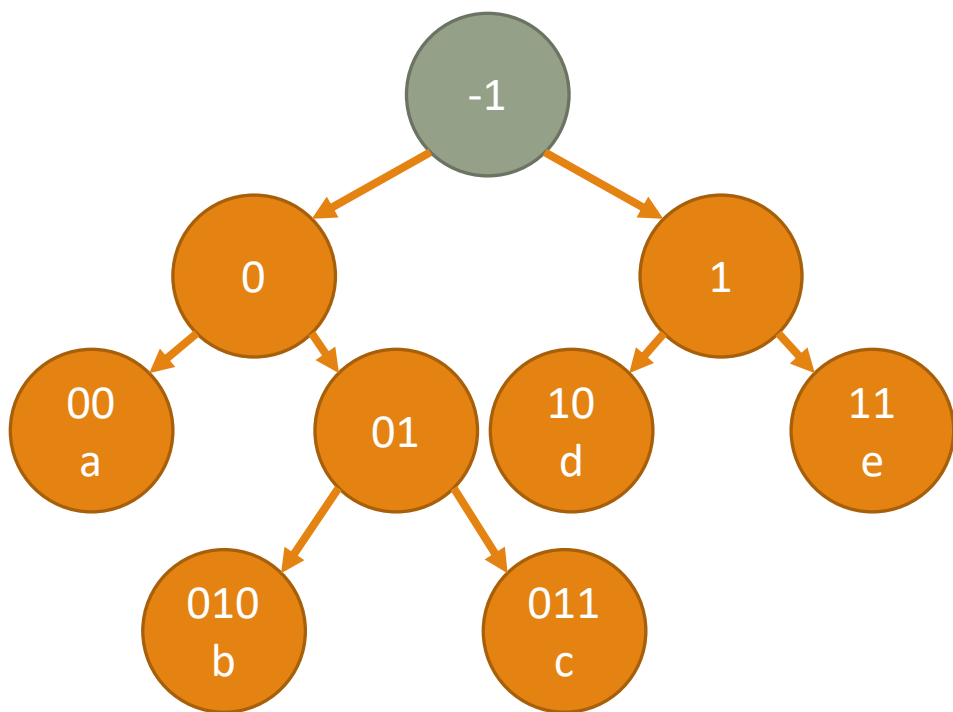
000001001000100110110111...

扫描窗口

初始化扫描



# 逐字符解压缩



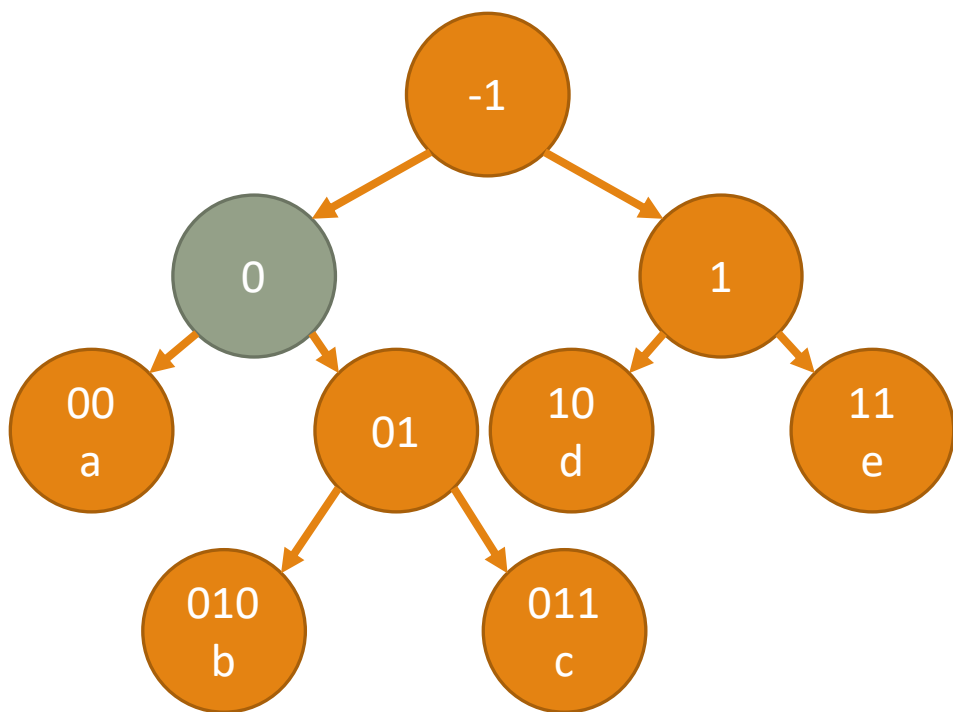
位序列

000001001000100110110111...

扫描窗口

初始化扫描，指针指向根节点，开始扫描

# 逐字符解压缩



位序列

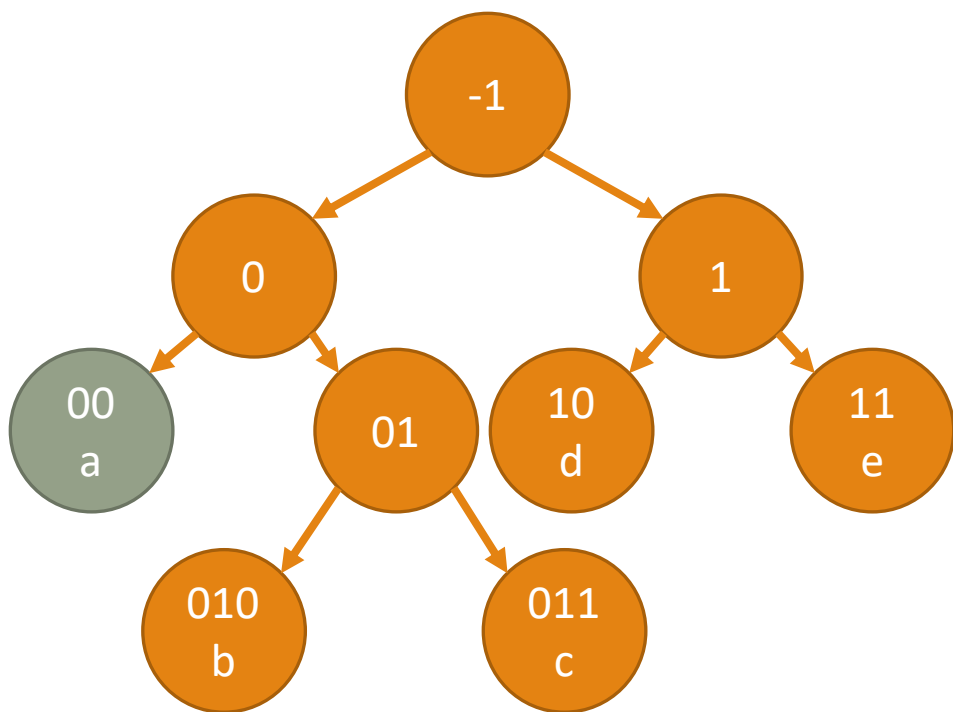
000001001000100110110111...

扫描窗口

初始化扫描，指针指向根节点，开始扫描  
遇到0，往左走，不是叶子节点，保存位置

输出序列：

# 逐字符解压缩



位序列

000001001000100110110111...

扫描窗口

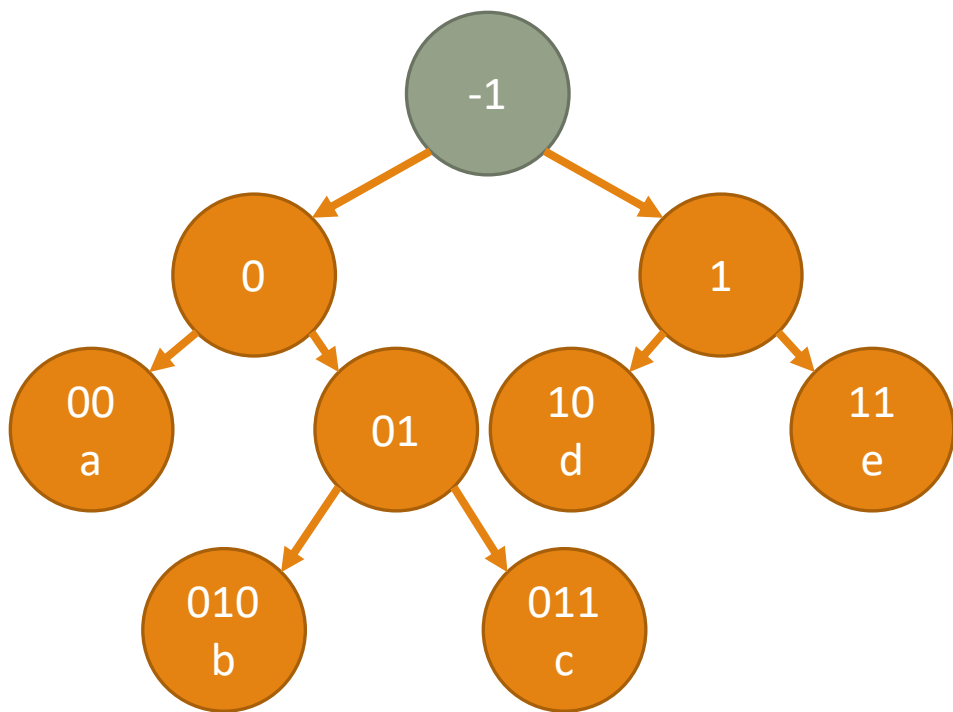
初始化扫描，指针指向根节点，开始扫描

遇到0，往左走，不是叶子节点，保存位置

遇到0，往左走，是叶子节点，输出字符，返回根节点

输出序列： a

# 逐字符解压缩



位序列

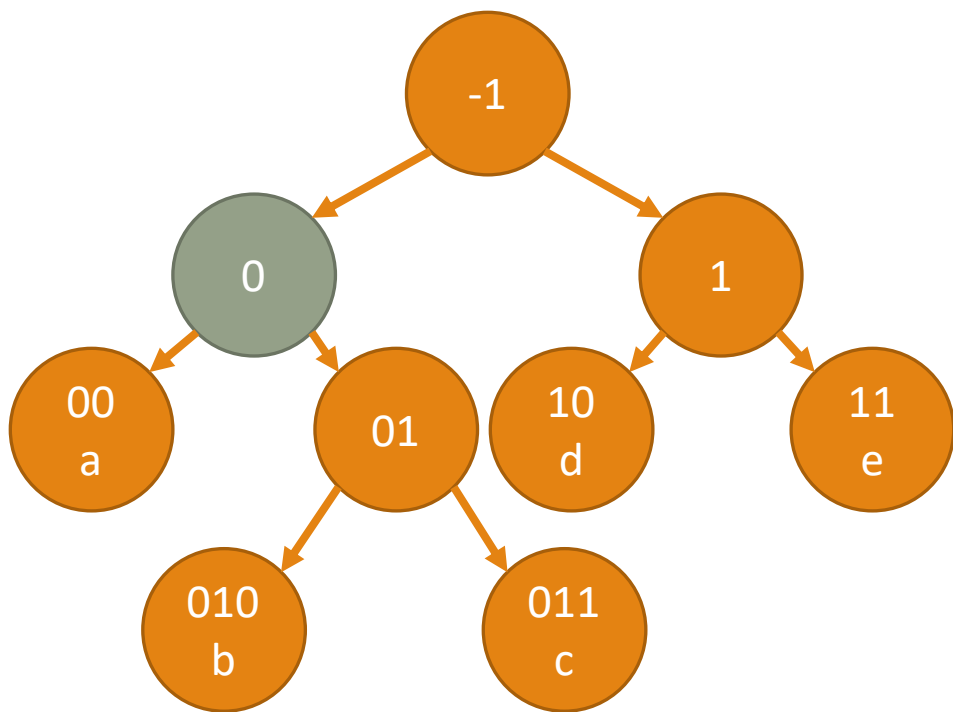
000001001000100110110111...

扫描窗口

初始化扫描，指针指向根节点，开始扫描

输出序列： a

# 逐字符解压缩



位序列

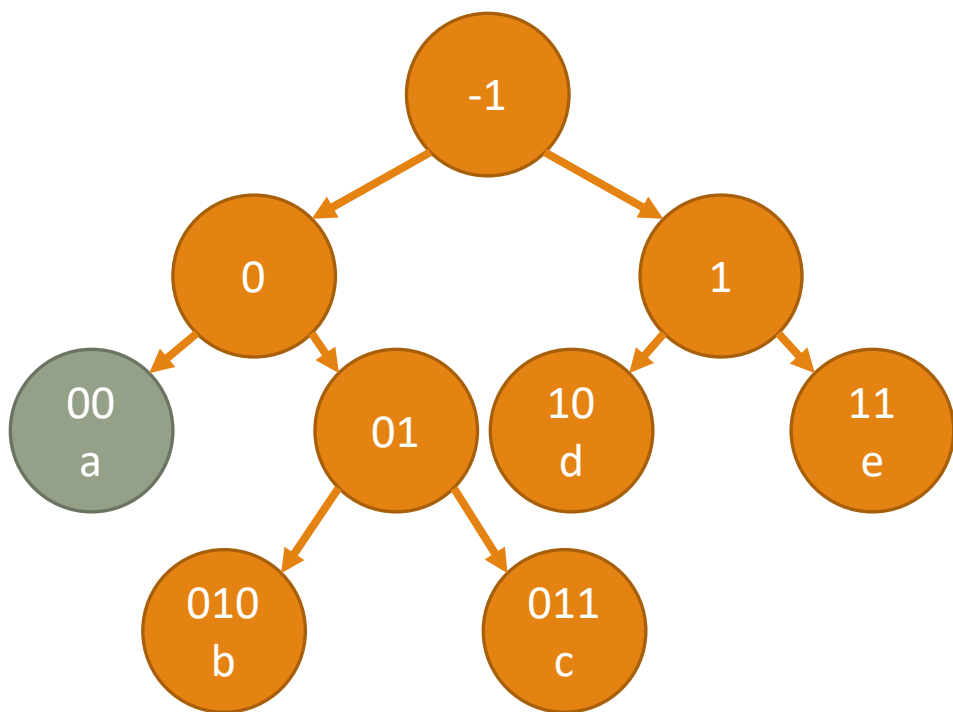
000001001000100110110111...

扫描窗口

初始化扫描，指针指向根节点，开始扫描  
遇到0，往左走，不是叶子节点，保存位置

输出序列： a

# 逐字符解压缩



位序列

000001001000100110110111...

扫描窗口

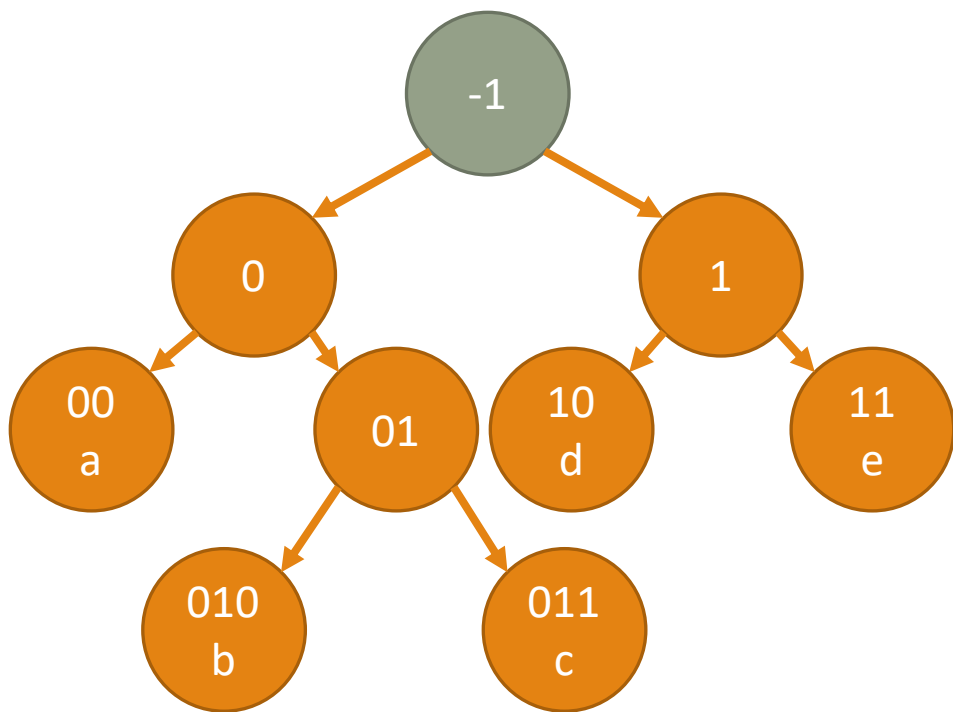
初始化扫描，指针指向根节点，开始扫描

遇到0，往左走，不是叶子节点，保存位置

遇到0，往左走，是叶子节点，输出字符，返回根节点

输出序列： aa

# 逐字符解压缩



位序列

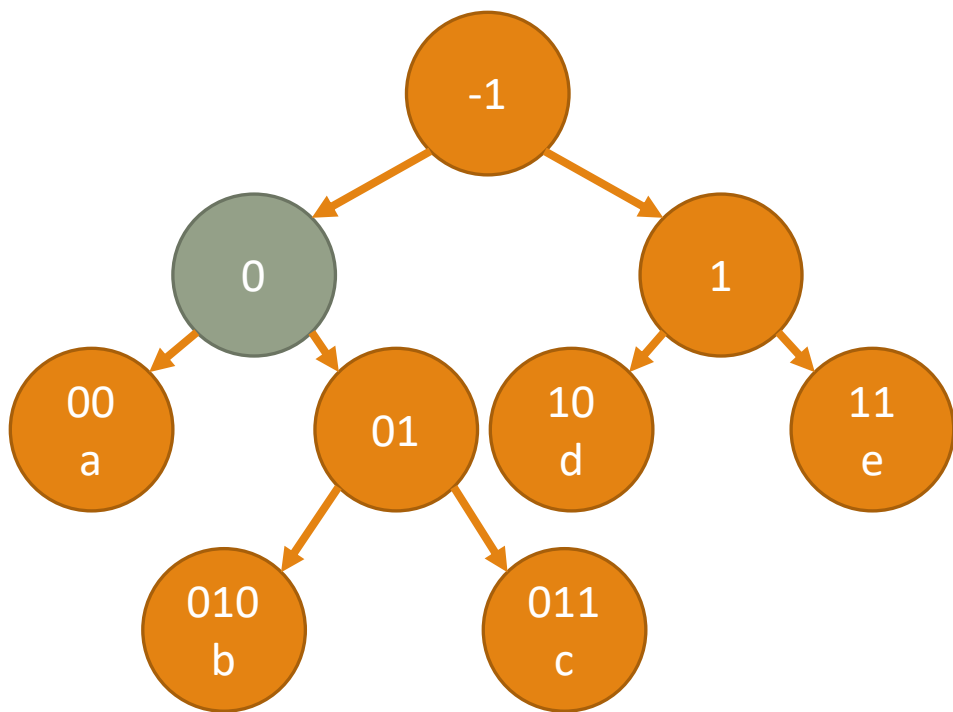
000001001000100110110111...

扫描窗口

初始化扫描，指针指向根节点，开始扫描

输出序列： aa

# 逐字符解压缩



位序列

000001001000100110110111...

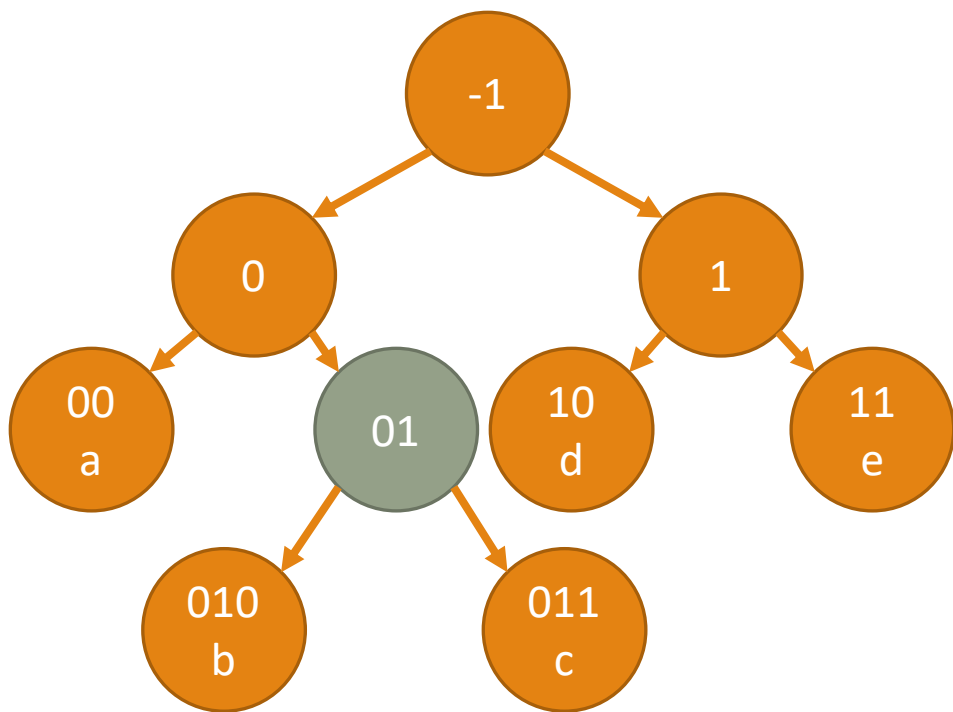
扫描窗口

初始化扫描，指针指向根节点，开始扫描  
遇到0，往左走，不是叶子节点，保存位置

输出序列： aa



# 逐字符解压缩



位序列

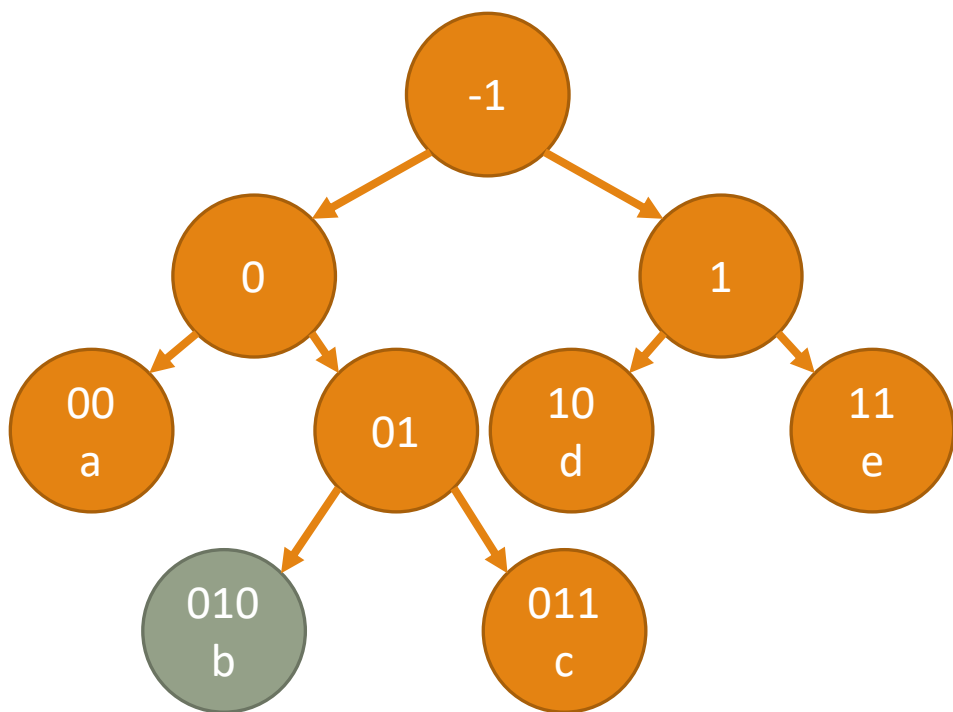
00000**1**001000100110110111...

扫描窗口

初始化扫描，指针指向根节点，开始扫描  
遇到0，往左走，不是叶子节点，保存位置  
遇到1，往右走，不是叶子节点，保存位置

输出序列： aa

# 逐字符解压缩



位序列

000001001000100110110111...

扫描窗口

初始化扫描，指针指向根节点，开始扫描

遇到0，往左走，不是叶子节点，保存位置

遇到1，往右走，不是叶子节点，保存位置

遇到0，往左走，是叶子节点，输出字符，返回根节点

输出序列： aab...

# 这样解压缩也完成了

---

时间：9秒

打开看一看，和ser.log一样

# 用到的知识点

---

输入输出流

位运算

结构体



第7章 补充-带参数的main函数.pdf



第7章 补充-单链表的基本操作.pdf



第7章 补充-输入输出流(第13章).pdf



第7章 补充-输入输出重定向.pdf



第7章 补充-位运算.pdf



第7章 补充-文件(C语言版).pdf



第7章 用户自定义数据类型-part1.pdf



第7章 用户自定义数据类型-part2.pdf

# 怎样优化呢？ (@WWS)

---

关于时间：

哪些地方涉及大量重复？

统计字符；建树（遍历寻找最小两节点）；

程序中频繁使用的慢速操作：

`fstream::get()` （统计字符频数、）

`fstream::tellg()` （解压）

`map`查找

# 解决方案

get太慢 --> 用二进制读入一大堆，在数组里面一个一个查看

```
while(!infile.eof()){
    char ch = infile.get();
    count(ch); // 统计字符
} // 到这里统计出了字符的情况和个数
```

```
void count(char c){
    /***/
    函数名: count
    功能: 统计输入文件中字符的频数
    /***/
    if(int(c) == -1) return;
    for(int i = 0; i < tot; ++i){
        if(Hufftree[i].syb == c){
            Hufftree[i].num++;
            return;
        }
    }
    Hufftree[tot].syb = c;
    Hufftree[tot].num = 1;
    ++tot;
}
```



```
int _countMap[256] = {0}; // 用数组优化查找速度
int *countMap = &_amp;_countMap[128];
char buffer[1024];
while (infile && !infile.eof())
{
    infile.read(buffer, 1024); // 一次读入1024字节，防止用get()，加快效率
    int readNum = infile.gcount();
    for (int i = 0; i < readNum; i++)
    {
        // count(countMap, buffer[i]); // 统计字符
        countMap[buffer[i]]++;
    }
} // 到这里统计出了字符的情况和个数
```

# 解决方案

建树的过程中查找最小值，每次都要遍历整个数组， $O(n)$ ，太慢！

这个过程中需要什么？只需要知道最小两个值就可以，而且在不断更新。

Priority\_queue，堆，都可以完美解决这个问题。

```
void hufftreeCreate(){
    /*******
    函数名: hufftreeCreate
    功能: 建立哈夫曼树
    *****/
    hufftreeInit();
    int final = 2*tot-2; // 总共2*tot-1个节点
    for(int i = tot; i <= final; ++i){
        int l, r;
        // 找树中最小的两个父亲节点
        findMinNode(l, r);
        // 用这两个节点建树
        createNode(l, r); // 这里面要更新总节点数
    }
}
```



```
priority_queue<int, vector<int>, cmp> huffQueue;
for (int i = -128; i < 128; i++)
{
    // 将字典内保存的信息更新到节点中
    if (countMap[i] == 0)
        continue;
    Hufftree[tot].syb = (char)i;
    Hufftree[tot].num = countMap[i];
    huffQueue.push(tot); // 将结点加入优先队列
    ++tot;
}
while (!huffQueue.empty())
{
    if (huffQueue.size() == 1)
    {
        break; // 队列只剩下一个结点，证明树已经建立完成
    }
    int l = huffQueue.top();
    huffQueue.pop();
    int r = huffQueue.top();
    huffQueue.pop();
    createNode(l, r);
    huffQueue.push(tot);
    ++tot; // 自增操作移到了这里
}
```

# 解决方案

---

编码上也有学问。

按我的程序，到了叶子节点，操作为：复制编码，存到map中

这有什么问题？

Map为红黑树，查找复杂度为 $O(\log n)$ ，大量查找不够快。考虑一：unordered\_map，哈希表， $O(1)$ ；考虑二：直接使用指针，复杂度 $O(1)$ ，而且还可以存下来编码长度。存下来编码长度，在压缩文件的时候不需要频繁strlen

```
if (Hufftree[s].lchd == -1) //哈夫曼树是满二叉树，只检查一边。是叶子节点
{
    strcpy(Hufftree[s].hcode, tempcode);
    char ch = Hufftree[s].syb;
    mp[ch].code = Hufftree[s].hcode;
    mp[ch].len = strlen(Hufftree[s].hcode);
    return;
}
```

```
struct huffCode
{
    char *code;
    int len;
};
```

```
huffCode _mp[256] = {0};
huffCode *mp = &_amp[128];
```



# 解决方案

---

压缩和解压缩，与统计的时候一样，如果一个字符一个字符地读，太慢！反复进行文件到程序的读取，比较笨。采用缓冲的方法，一次读很多

```
in.read(buffer, 1024); // 一次读入1024字节，防止用get()，加快效率
int readNum = in.gcount();
```

联系：

处理批量数据时，科学计算软件或高级语言的科学计算库，都会采用对同类型数据整体计算的方法，提高速度。

# 解决方案

---

最后，一些简单的运算操作，如位运算，设为`inline`，能加快速度。

# 优化后的结果

---

压缩率：65.01%

压缩和解压的速度：1秒之内

回忆一下，采用了那些方法？

`get(), tellg() → read(buffer, 1024);`

`Priority_queue`

`Map → huffcode`

并不复杂的优化，程序性能却有了很大幅的提升。

# 跟2345好压比一比

---

速度：差不多

压缩率：好压：4%左右

为何呢？

想一想，直观上感觉到huffman编码后文件趋向于随机、不可读，也不太适合再用于字典算法等。如果先使用其他算法压缩一轮，再用huffman，压缩率仍能减小一部分，这样或许就能达到比较小的压缩率。

# 感想

---

1. 把基础知识学好，很多问题都能解决，而且解决方法并不复杂。
2. 编程中的很多意识需要培养，比如优化意识、代码整洁度、命名规则、debug方法。
3. 关于哈夫曼编码：前人开创新理论新方法是难的，但我们后人去理解他们并不难。我们要站在巨人的肩膀上，学习前人的理论，融入自己的创造力，来解决当下的问题。

# 谢谢大家！

---