



# Tree & Binary Trees (6)

---

College of Computer Science, CQU

# Outline

---

- ❑ **Introduction**
- ❑ **Weighted Path Length**
- ❑ **Huffman tree**
- ❑ **Huffman Codes**

# Introduction

---

- ❑ We usually encode strings by assigning **fixed-length codes** to all characters in the alphabet (for example, 8-bit coding in ASCII).
- ❑ However, if different characters occur with different frequencies, we can save memory and reduce transmittal time by using **variable-length encoding**.
- ❑ The idea is to assign shorter codes to characters that occur more often.

# Introduction

---

relative frequencies of the letters of the alphabet:

Letter	Frequency	Letter	Frequency
A	77	N	67
B	17	O	67
C	32	P	20
D	42	Q	5
E	120	R	59
F	24	S	67
G	17	T	85
H	50	U	37
I	76	V	12
J	4	W	22
K	7	X	4
L	42	Y	22
M	24	Z	2

**The letter 'E' appears about 60 times more often than the letter 'Z.'**

# Introduction

---

- ❑ We must be careful when assigning variable-length codes.
- ❑ For example, let us encode **e** with 0, **a** with 1, and **t** with 01. How can we then encode the word **tea**?
- ❑ The encoding is **0101**.
- ❑ Unfortunately, this encoding is ambiguous. It could also stand for **eat**, **eaea**, or **tt**.
- ❑ Of course this coding is unacceptable, because it results in loss of information.

# Introduction

---

- ❑ To avoid such ambiguities, we can use **prefix codes**. In a prefix code, the bit string for a character never occurs as the **prefix** (first part) of the bit string for another character.
- ❑ For example, the encoding of **e** with 0, **a** with 10, and **t** with 11 is a prefix code. How can we now encode the word **tea**?
- ❑ The encoding is **11010**.
- ❑ This bit string is unique, it can only encode the word **tea**.

# Introduction

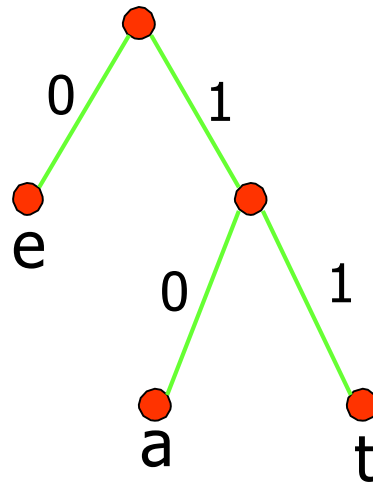
---

- ❑ We can represent prefix codes using binary tree, where the characters are the labels of the leaves in the tree.
- ❑ The edges of the tree are labeled so that an edge leading to a left child is assigned a 0 and an edge leading to a right child is assigned a 1.
- ❑ The bit string used to encode a character is the sequence of labels of the edges in the unique path from the root to the leaf labeled with this character.

# Introduction

---

- The tree corresponding to our example:



In a tree, no leaf can be the ancestor of another leaf. Therefore, no encoding of a character can be a prefix of an encoding of another character (prefix code).



# Introduction

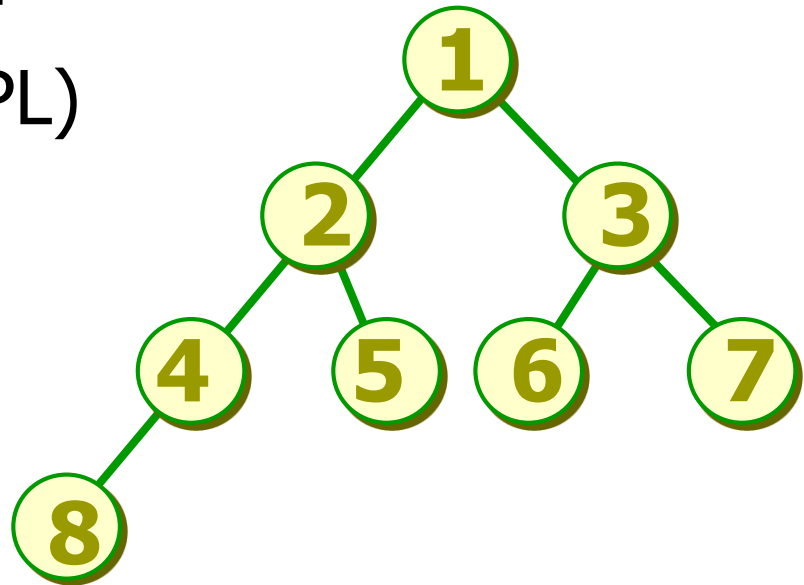
---

- ❑ To determine the optimal encoding for a given string, we first have to find the frequencies of characters in that string. Let us consider the following string:
  - eeadfeejjeggebeeggdddehhhececddeciedee
  - It contains  $1 \times a$ ,  $1 \times b$ ,  $3 \times c$ ,  $6 \times d$ ,  $15 \times e$ ,  $1 \times f$ ,  $4 \times g$ ,  $3 \times h$ ,  $1 \times i$ , and  $2 \times j$ .
- ❑ We can use **Huffman's** algorithm to build the optimal coding tree.

# Path Length (PL)

- If  $n_1, n_2, \dots, n_k$  is a sequence of nodes in the tree such that  $n_i$  is the parent of  $n_{i+1}$  for  $1 \leq i < k$ , then this sequence is called a **path** from  $n_1$  to  $n_k$ . The **length** of the path is  $k - 1$ .
- Path Length of tree (PL)

$$PL = 3*1 + 2*3 = 9$$



# Weighted Path Length

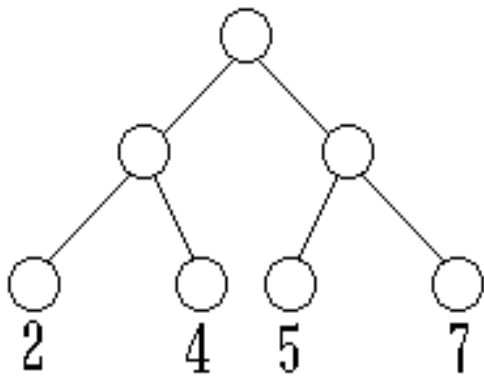
---

- **weighted path length of a leaf** is its weight times its depth.
- **weighted path length of a tree** is the sum of weighted path lengths of every leaf.

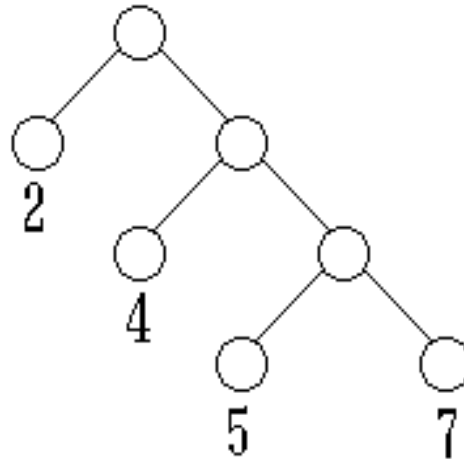
$$WPL = \sum_{k=1}^n w_k * L_k$$

- **Huffman tree has the minimum WPL**

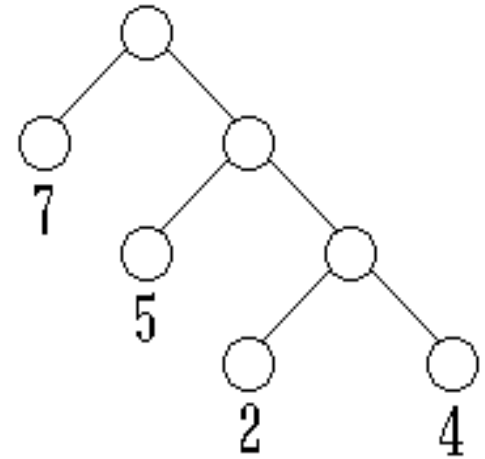
# Huffman tree



(a)  $WPL = 36$



(b)  $WPL = 46$



(c)  $WPL = 35$

□ **C is Huffman tree.**

# Building Human Trees

---

- ❑ Create a collection of  $n$  initial Huffman trees, each of which is a single leaf node containing one of the letters. Put the  $n$  partial trees onto a list in ascending order by weight (frequency).
- ❑ Next, remove the first two trees (the ones with lowest weight) from the list. Join these two trees together to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two trees. Put this new tree back on the list in the correct place necessary to preserve the order of the list.
- ❑ This process is repeated until all of the partial Huffman trees have been combined into one.

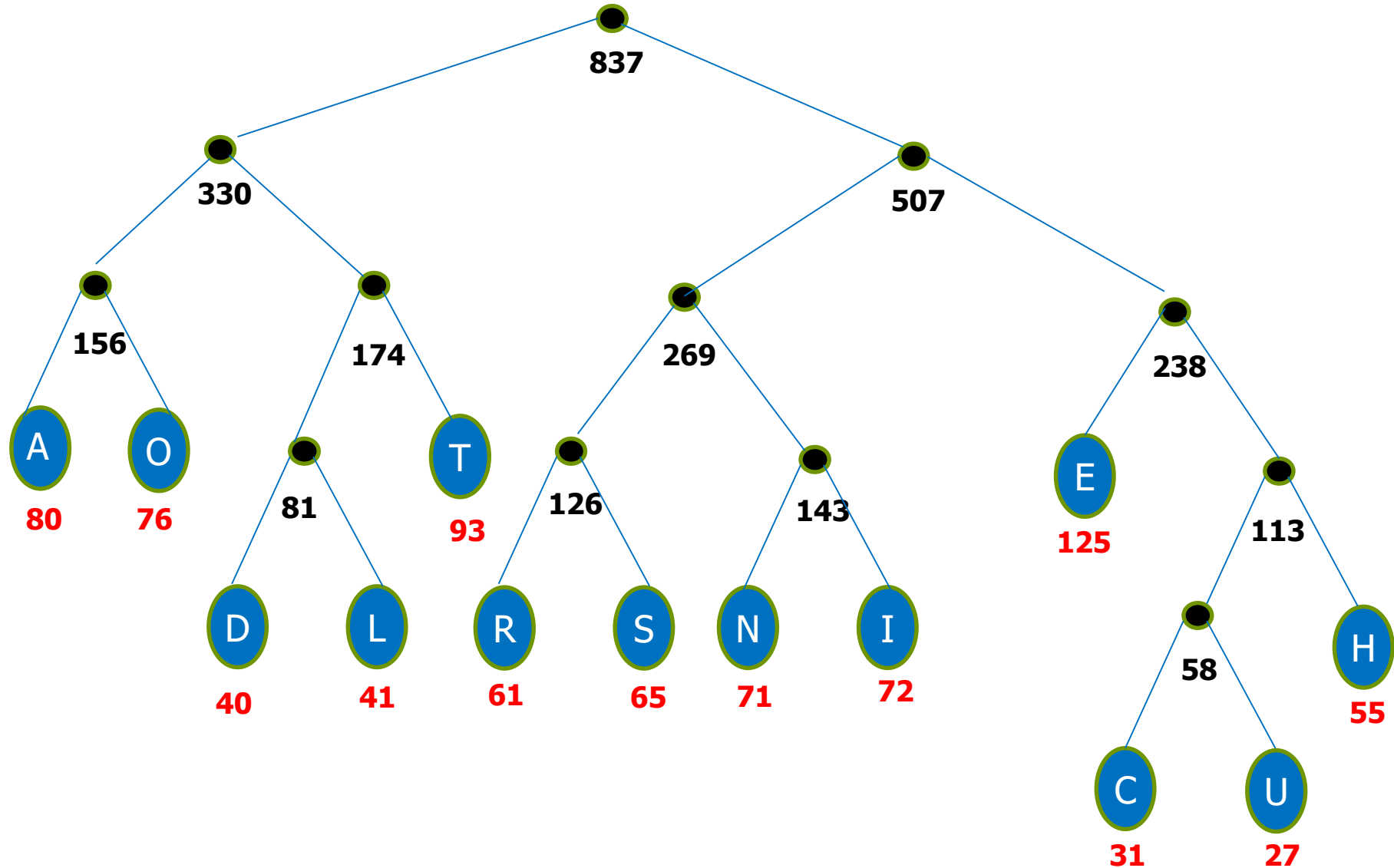
# Example

- Character count in text:

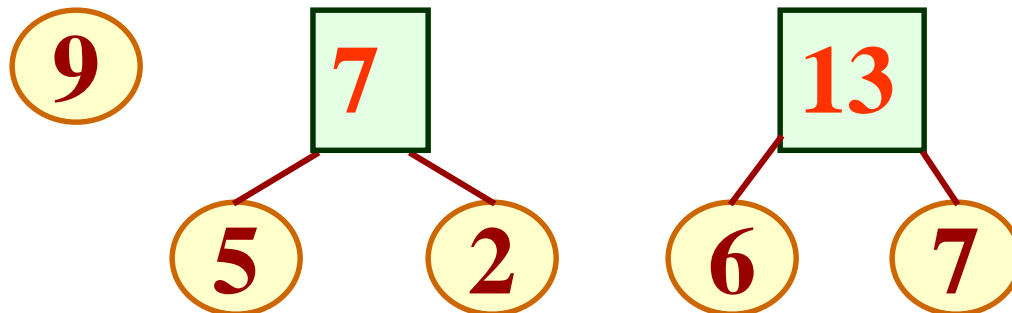
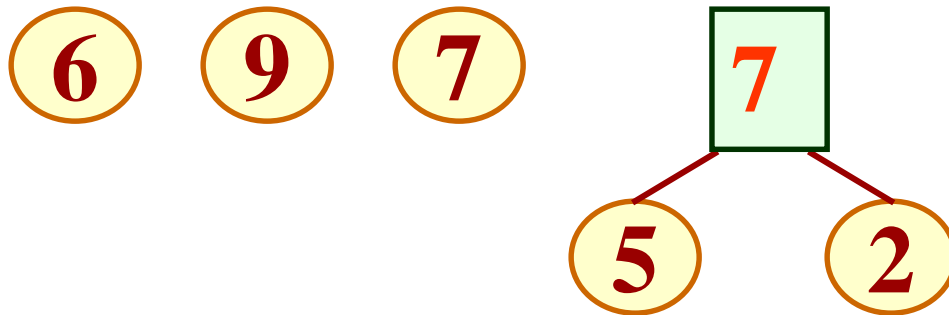
Char	E	T	A	O	I	N	S	R	H	L	D	C	U
Freq	125	93	80	76	72	71	65	61	55	41	40	31	27

- At first, there are 13 partial trees.

# Huffman Tree Construction

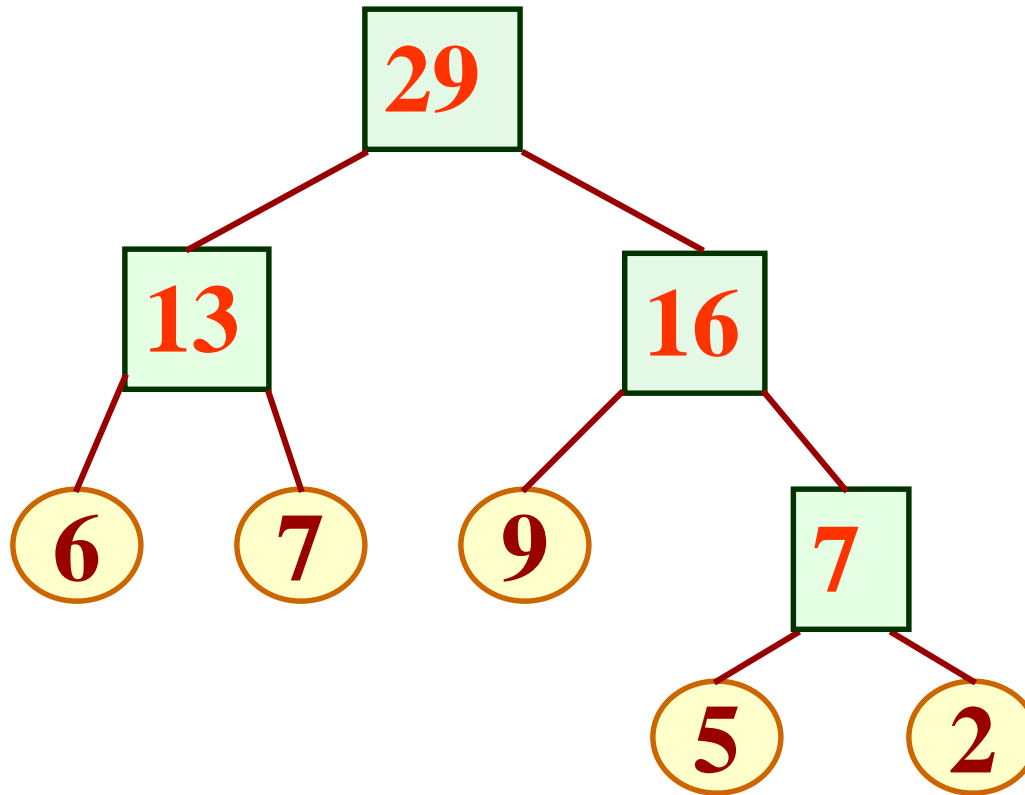
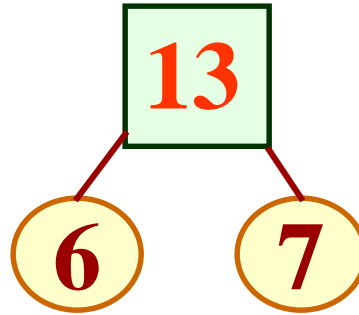
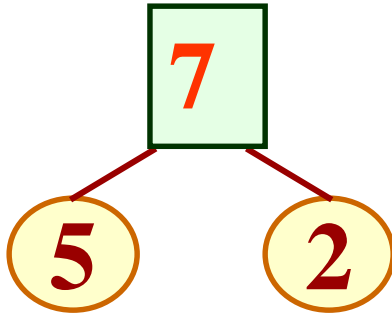


练习：已知权值  $W=\{ 5, 6, 2, 9, 7 \}$





9



# Implementation for Huffman Tree with heap

```
// Huffman tree node abstract base class
template <typename E> class HuffNode {
public:
    virtual ~HuffNode() {}           // Base destructor
    virtual int weight() = 0;        // Return frequency
    virtual bool isLeaf() = 0;       // Determine type
};

template <typename E>                // Leaf node subclass
class LeafNode : public HuffNode<E> {
private:
    E it; // Value
    int wgt; // Weight
public:
    LeafNode(const E& val, int freq) // Constructor
        { it = val; wgt = freq; }
    int weight() { return wgt; }
    E val() { return it; }
    bool isLeaf() { return true; }
};
```



```

template <typename E> // Internal node subclass
class IntlNode : public HuffNode<E> {
private:
    HuffNode<E>* lc; // Left child
    HuffNode<E>* rc; // Right child
    int wgt; // Subtree weight
public:
    IntlNode(HuffNode<E>* l, HuffNode<E>* r)
        { wgt = l->weight() + r->weight(); lc = l; rc = r; }
    int weight() { return wgt; }
    bool isLeaf() { return false; }
    HuffNode<E>* left() const { return lc; }
    void setLeft(HuffNode<E>* b)
        { lc = (HuffNode<E>*)b; }
    HuffNode<E>* right() const { return rc; }
    void setRight(HuffNode<E>* b)
        { rc = (HuffNode<E>*)b; }
};

```

# Implementation for Huffman Tree

```
template <typename E>
class HuffTree {
private:
    HuffNode<E>* Root;                // Tree root
public:
    HuffTree(E& val, int freq)         // Leaf constructor
    { Root = new LeafNode<E>(val, freq); }
    // Internal node constructor
    HuffTree(HuffTree<E>* l, HuffTree<E>* r)
    { Root = new IntlNode<E>(l->root(), r->root()); }
    ~HuffTree() {} // Destructor
    HuffNode<E>* root() { return Root; } // Get root
    int weight() { return Root->weight(); } // Root weight
};
```



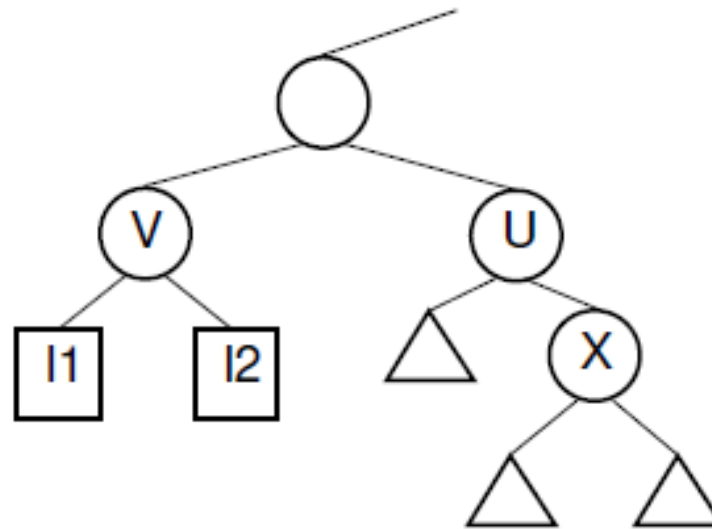
# Building Huffman tree

```
template <typename E> HuffTree<E>*  
buildHuff(HuffTree<E>** TreeArray, int count) {  
    heap<HuffTree<E>*, minTreeComp>* forest =  
        new heap<HuffTree<E>*, minTreeComp>(TreeArray, count, count);  
    HuffTree<char> *temp1, *temp2, *temp3 = NULL;  
    while (forest->size() > 1) {  
        temp1 = forest->removefirst(); // Pull first two trees  
        temp2 = forest->removefirst(); // off the list  
        temp3 = new HuffTree<E>(temp1, temp2);  
        forest->insert(temp3); // Put the new tree back on list  
        delete temp1; // Must delete the remnants  
        delete temp2; // of the trees we created  
    }  
    return temp3;  
}
```

# Lemma

- **Lemma 5.1** For any Huffman tree built by function **buildHuff** containing at least two letters, the two letters with least frequency are stored in siblings nodes whose depth is at least as deep as any other leaf nodes in the tree.

- **Proof:**



# Theorem

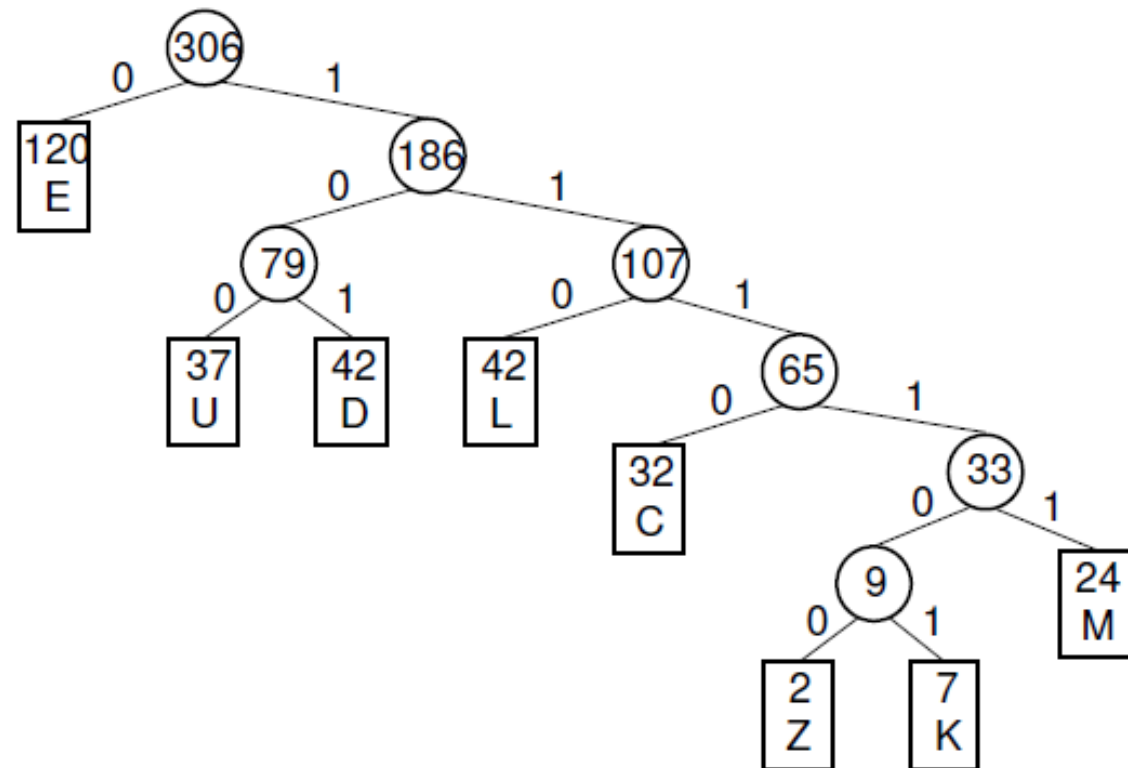
---

- ❑ Theorem 5.3 Function **buildHuff** builds the Huffman tree with the minimum external path weight for the given set of letters.
- ❑ **Proof:** The proof is by induction on  $n$ , the number of letters
  - Base Case: For  $n = 2$ , there are only two possible trees
  - Induction Hypothesis: Assume that any tree created by **buildHuff** that contains  $n - 1$  leaves has minimum external path length
  - Induction Step: Given a Huffman tree  $T$  **with  $n$**  leaves,  $n \geq 2$ , suppose that  $w_1 \leq w_2 \leq \dots \leq w_n$  where  $w_1$  to  $w_n$  are the weights of the letters. Call  $V$  the parent of the letters with frequencies  $w_1$  and  $w_2$ .

# Assigning Huffman Codes

□ Example:

Letter	C	D	E	K	L	M	U	Z
Frequency	32	42	120	7	42	24	37	2





# Using Huffman Codes

---

From the Huffman tree, we can get the codes for all eight letters.

Letter	Freq	Code	Bits
C	32	1110	4
D	42	101	3
E	120	0	1
K	7	111101	6
L	42	110	3
M	24	11111	5
U	37	100	3
Z	2	111100	6

# Encoding

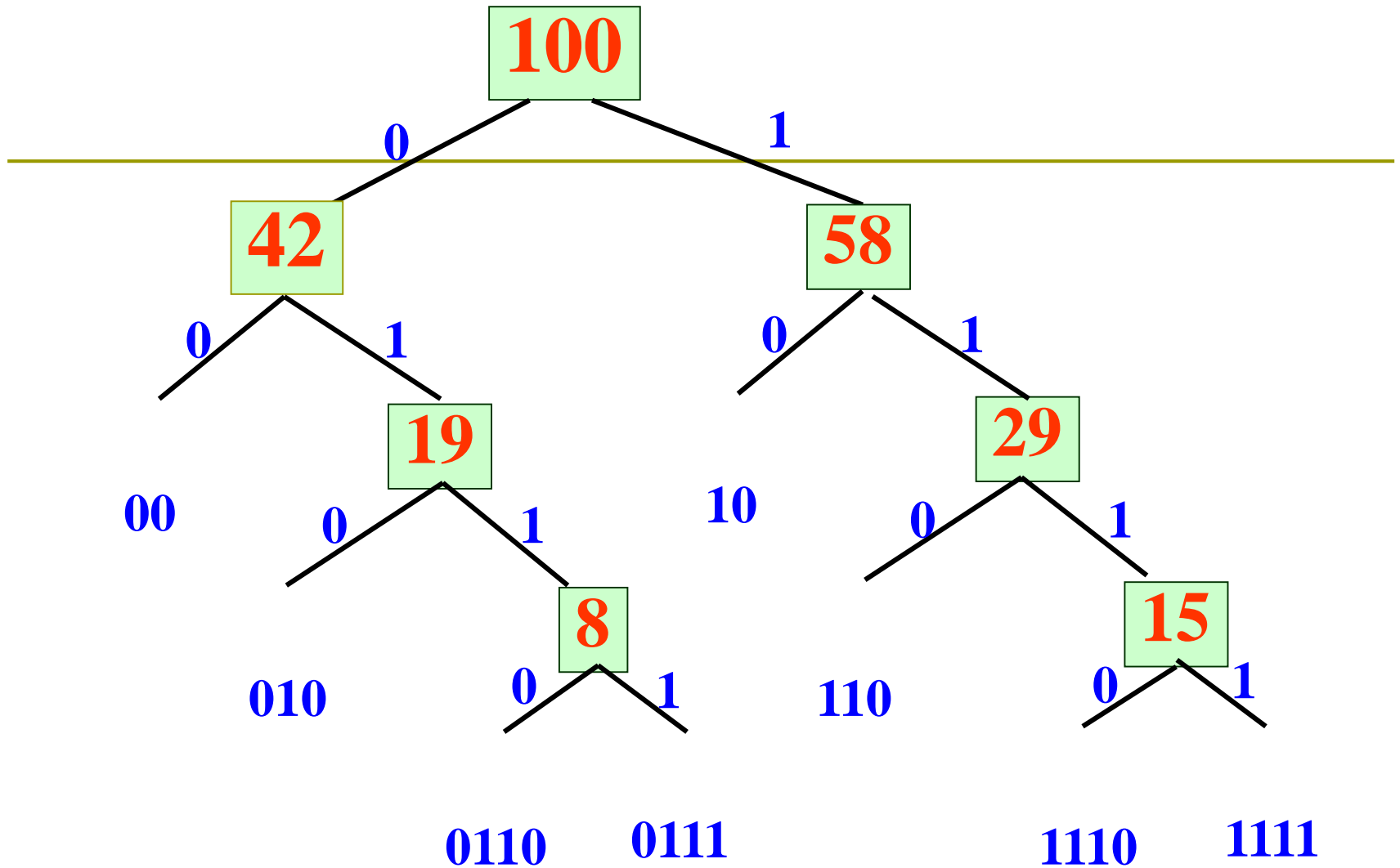
---

- ❑ replace each letter in the string with its binary code. A lookup table can be used for this purpose.
- ❑ Using the code generated by example Huffman tree
  - "DEED" is represented by the bit string "10100101"
  - "MUCK" is represented by the bit string "111111001110111101."

# Decoding

---

- ❑ Decoding a bit string begins at the root of the tree. To take branches depending on the bit value — left for '0' and right for '1' — until reaching a leaf node. This leaf contains the first character in the message. Then to process the next bit in the code restarting at the root to begin the next character.
- ❑ To decode the bit string "1011001110111101"
  - "DUCK"



5

29

7

8

14

23

3

11

□ (3) 由此赫夫曼树生成赫夫曼编码。

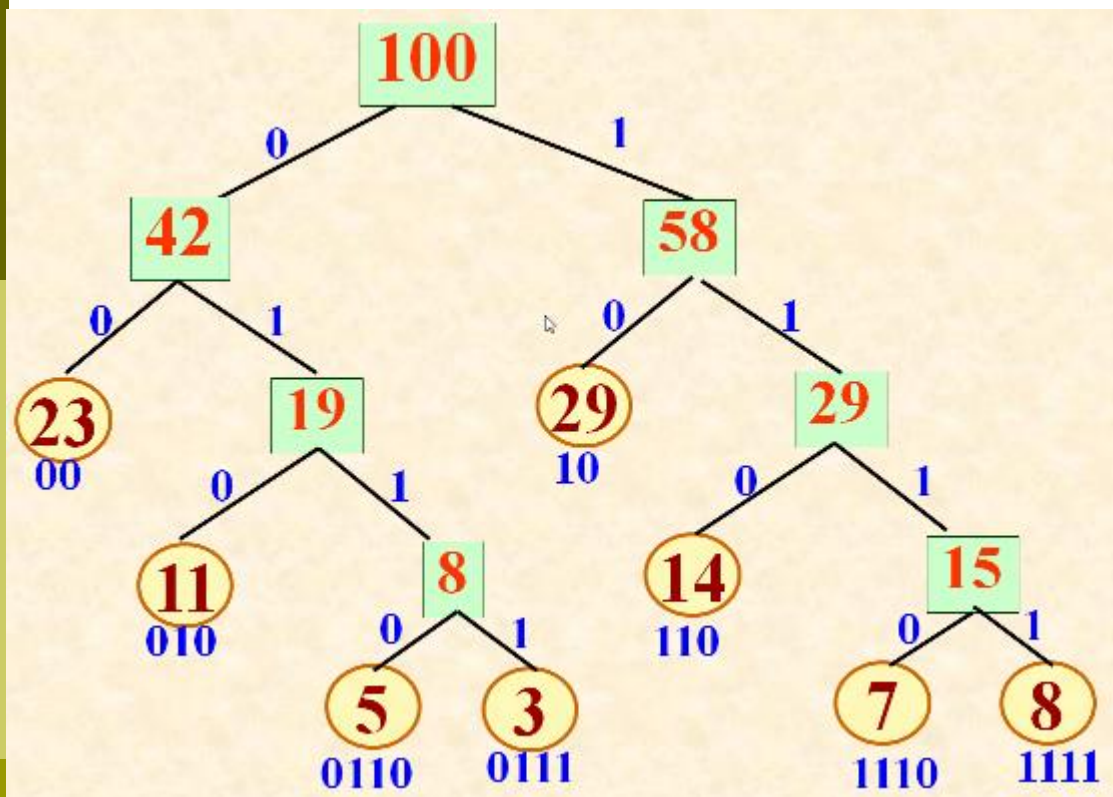
□ 最后得出每个字符的编码为：

---

字符	字符的使用频率	编码
①	0.05	0111
②	0.29	10
③	0.07	1110
④	0.08	1111
⑤	0.14	110
⑥	0.23	00
⑦	0.03	0110
⑧	0.11	010

以上的①~⑧分别代表 8 个字符。

■ 比如，发送一段编码：**0000011111010010**,



字符                  字符的使用频率

①	0.05
②	0.29
③	0.07
④	0.08
⑤	0.14
⑥	0.23
⑦	0.03
⑧	0.11

以上的①~⑧分别代表 8 个字符。

■ 接收方可以准确地通过译码得到：    ⑥   ⑥ ⑦   ⑤   ② ⑧

# Prefix Property

---

- **Huffman codes certainly have the prefix property because any prefix for a code would correspond to an internal node, while all codes correspond to leaf nodes.**

---

-End-