CSCI 2110 Data Structures and Algorithms

Module 6: Binary Trees

DALHOUSIE UNIVERSITY

CSCI 2110: Module 6

Srini Sampalli

1

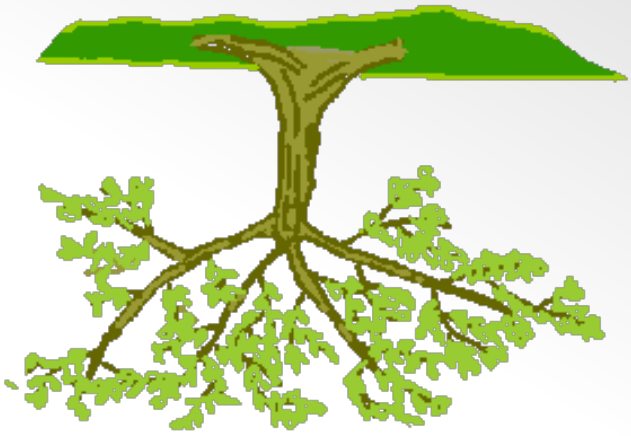
Learning Objectives

- Define the tree and binary tree data structures.
- Understand the terminology of binary trees.
- Know different types of binary tree traversals.
- Build the binary tree class.
- Learn one application of binary trees: Huffman coding.

A natural tree

DALHOUSIE UNIVERSITY

A tree in the eyes of a computer scientist!



DALHOUSIE UNIVERSITY

The tree data structure

- A **tree** is a data structure consisting of a set of nodes arranged in a hierarchy (that is, they are arranged in levels).
- Each node contains data.
- There is one special node called the **root** which is the starting point of the tree. The root is at Level 0.
- The root is connected to zero or more nodes at Level 1 by **branches** or links.
- Similarly each node at Level i is connected to zero or more nodes at Level i+1.
- If a node at Level i connects to nodes at Level i+1, then the node at Level i is called the **parent node** to the nodes that it connects.
- The nodes that it connects to at Level i+1 are called its **children or child nodes.**
- A node that does not have any children is called a **leaf node**.
- Non-leaf nodes are called **internal nodes.**

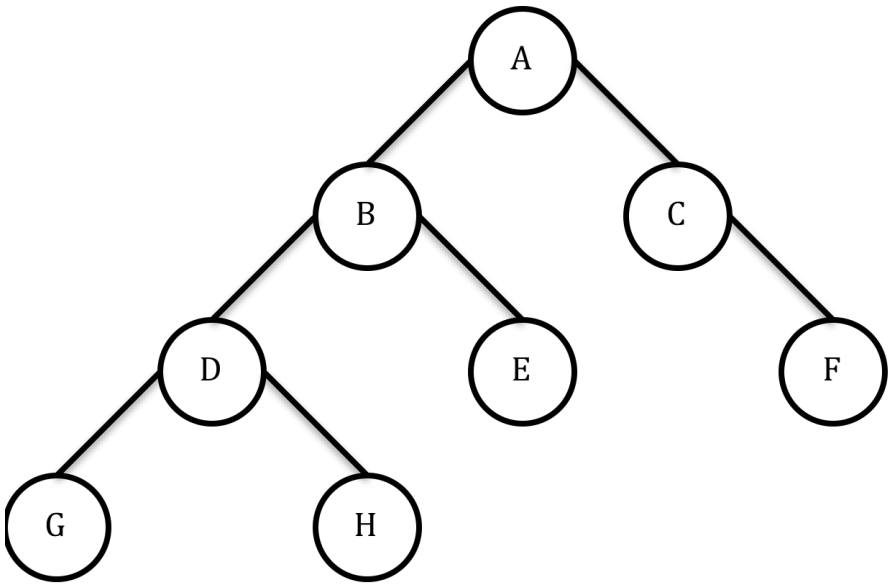
DALHOUSIE UNIVERSITY

Binary Trees: Definition and Terminology

A BINARY TREE IS A TREE IN WHICH EVERY NODE HAS AT MOST TWO CHILDREN.

- The two children of a node are connected to its left and right branches, and are respectively called the **left child** and the **right child** of the node.
- In the example, the node at the top level, A, is the **root** of the binary tree.
- B and C are the **left and right children** of A.
- A is the **parent** of B and C.
- E, F, G and H are **leaf nodes**.
- A, B, C and D are **internal nodes**.
- The binary tree rooted at the left child of a node is called the **left subtree** of the node.
- Similarly, the binary tree rooted at the right child of a node is called the **right subtree** of the node.

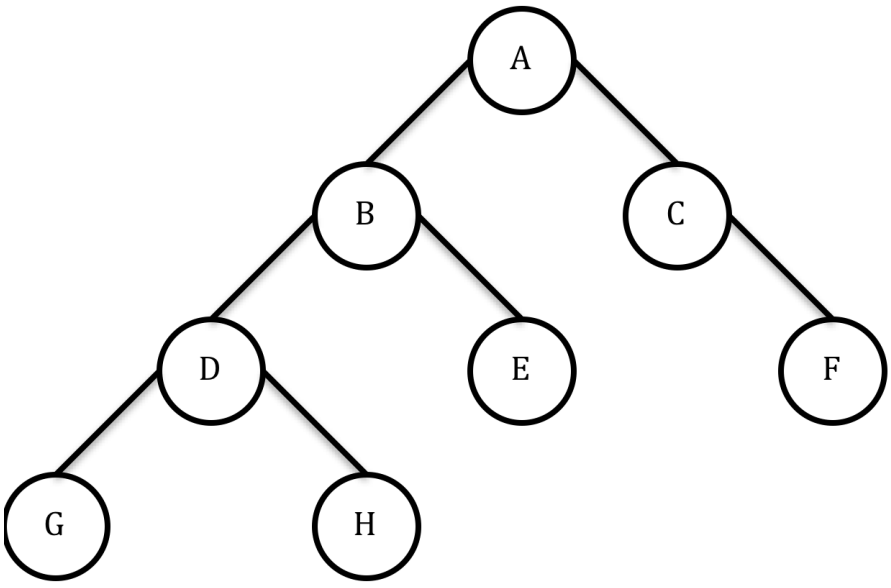
DALHOUSIE UNIVERSITY



Binary Tree Terminology (cont'd.)

- There is a **single path** from any node to any other node.
- The distance between two nodes is counted in terms of the number of branches.
- Distance of a node from the root is called its **depth**.
- Depth of the root = 0.
- All nodes at the same depth are said to be at the same **level**.
- The **height** of a binary tree is the maximum depth at which there is a node.
- The **maximum number of nodes N_{max}** in a binary tree of height h is **$2^{(h+1)} - 1$.**
- The height of a binary tree with the maximum number of nodes = N_{max} (all levels filled) is **$h = \log_2(N_{max}+1)-1$**

DALHOUSIE UNIVERSITY



Strictly Binary and Complete Binary

STRICTLY BINARY TREE

- A strictly binary tree is one in which every node has either no child or two children (i.e., no node can have only one child).

COMPLETE BINARY TREE

- In a complete binary tree, every level but the last must have the maximum number of nodes possible at that level.
- The last level may have fewer than maximum possible, but they should be arranged from left to right without any empty spots.

DALHOUSIE UNIVERSITY

EXERCISE: Mark each of the following binary trees as strictly binary or complete binary or both or none.

Tree	Strictly Binary?	Complete Binary?

Recursive definitions for the binary tree

Recursive definition of binary tree:

- A binary tree is either empty or consists of a special node called root that has a left subtree and a right subtree that are mutually disjoint binary trees.

Recursive definition of number of nodes:

- The number of nodes in an empty binary tree is zero. Otherwise, the number of nodes is one plus the number of nodes in the left and right subtrees of the root.

Recursive definition of height:

- The height of an empty binary tree is -1. Otherwise, the height is one plus the maximum of the heights of the left and right subtrees of the root.

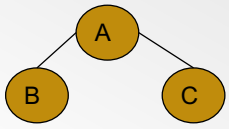
DALHOUSIE UNIVERSITY

Binary Tree Traversals

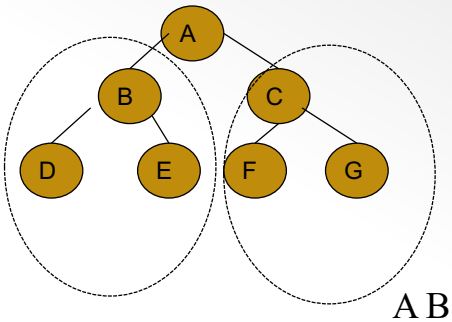
- A binary tree traversal is a systematic method of visiting and (processing) each node in the tree.
- There are three main types of traversals:
  - **Pre-order traversal**
  - **In-order traversal**
  - **Post-order traversal**

Pre-order traversal (root-L-R)

- ***Visit the root***
- ***Traverse the left subtree***
- ***Traverse the right subtree***



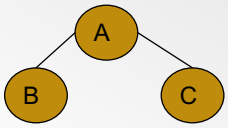
A B C



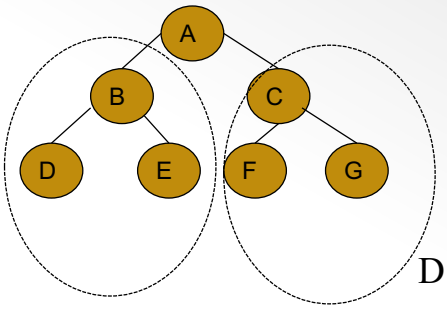
A B D E C F G

In-order traversal (L-root-R)

- *Traverse the left subtree*
- *Visit the root*
- *Traverse the right subtree*



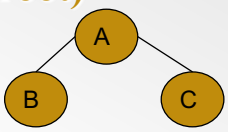
B A C



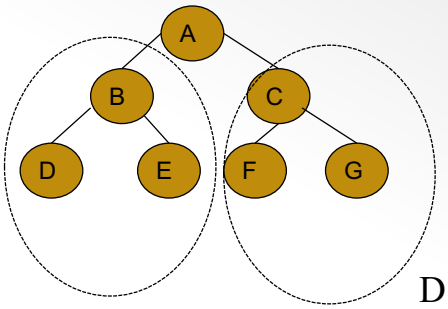
D B E A F C G

Post-order traversal (L-R-root)

- *Traverse the left subtree*
- *Traverse the right subtree*
- *Visit the root*



B C A



D E B F G C A

TREE TRAVERSALS

Determine the pre-order, in-order, post-order and level order traversals for each of the following binary trees.

Class BinaryTree<T>

For the attributes of this class, we will model each node. Each node has data, a left child, a right child and a parent. We will not call it Node but call it a BinaryTree since each node is itself a binary tree. This will help us use recursive methods.

Constructors

BinaryTree()	Creates an empty binary tree

Methods

Name	**What it does**	**Header**	**Price tag (complexity)**
makeRoot	Creates a single node tree with the specified data in that node.		
set methods	Sets the data, left, right and parent.		
get methods	Gets the data, left, right and parent.		
root	Returns the root of the tree in which this tree is a subtree		
attachLeft	Attaches the specified tree as the left subtree of this tree node.		
attachRight	Attaches the specified tree as the right subtree of this tree node		
detachLeft	Detaches the left subtree from this tree node and returns it		
detachRight	Detaches the right subtree from this tree node and returns it		
isEmpty	Returns true if this tree is empty, false otherwise		
clear	Empties out this binary tree and detaches it from its parent node		

We will also add utility methods for the traversals. The complexity will be O(n), where n is the number of nodes in the tree.

```
public class BinaryTree<T>
{
	private T data;
	private BinaryTree<T> parent;
	private BinaryTree<T> left;
	private BinaryTree<T> right;

	public BinaryTree(){
		parent = left = right = null;
		data = null;
	}

	public boolean isEmpty()
	{
		if (data == null)
			return true;
		else
			return false;
	}
	public void clear()
	{
		left = right = parent = null;
		data = null;
	}

	public void makeRoot(T data){
		if (!isEmpty())
		{
			System.out.println("Can't make root. Already exists");
		}
		else
			this.data = data;
	}

	public void setData(T data){
		this.data = data;
	}

	public void setLeft(BinaryTree<T> tree){
		left = tree;
	}
	public void setRight(BinaryTree<T> tree){
		right = tree;
	}
	public void setParent(BinaryTree<T> tree){
		parent = tree;
	}

	public T getData(){
		return data;
	}
```

```
public BinaryTree<T> getParent(){
        return parent;
}
public BinaryTree<T> getLeft(){
        return left;
}
public BinaryTree<T> getRight(){
        return right;
}
public void attachLeft(BinaryTree<T> tree)
{









}
public void attachRight(BinaryTree<T> tree)
{

      if (tree==null) return;
      else if (right!=null || tree.getParent()!=null){
         System.out.println("Can't attach");
         return;
      }
      else{

            tree.setParent(this);
            this.setRight(tree);
      }
}
public BinaryTree<T> detachLeft(){
```

```
public BinaryTree<T> detachRight()
{
	if (this.isEmpty()) return null;
	BinaryTree<T> retRight = right;
	right =null;
	if (retRight!=null) retRight.setParent(null);
		return retRight;
}

public BinaryTree<T> root()
{









}

public static <T> void preorder(BinaryTree<T> t)
{









}

public static <T> void inorder(BinaryTree<T> t)
{
	if (t!=null)
	{
		inorder(t.getLeft());
		System.out.print(t.getData() + "\t");
		inorder(t.getRight());
	}
}
```

```
	public static <T> void postorder(BinaryTree<T> t)
	{
		if (t!=null)
		{
			postorder(t.getLeft());
			postorder(t.getRight());
			System.out.print(t.getData() + "\t");
		}
	}
}


public class BinaryTreeDemo
{
	public static void main(String[] args)
	{
		BinaryTree<String> A = new BinaryTree<String>();
		BinaryTree<String> B = new BinaryTree<String>();
		BinaryTree<String> C = new BinaryTree<String>();
		BinaryTree<String> D = new BinaryTree<String>();
		BinaryTree<String> E = new BinaryTree<String>();
		BinaryTree<String> F = new BinaryTree<String>();
		A.makeRoot("A");
		B.makeRoot("B");
		C.makeRoot("C");
		D.makeRoot("D");
		E.makeRoot("E");
		F.makeRoot("F");

		A.attachLeft(B);
		A.attachRight(C);
		B.attachLeft(D);
		B.attachRight(E);
		D.attachLeft(F);

		System.out.print("Preorder:\t");
		BinaryTree.preorder(A);
		System.out.println();
		System.out.print("Inorder:\t");
		BinaryTree.inorder(A);
		System.out.println();
		System.out.print("Postorder:\t");
		BinaryTree.postorder(A);
		System.out.println();

	}
}
```

Huffman Coding
A nifty application with binary trees

What is Huffman coding?

- One of the earliest schemes used in text compression.
- Has led to the development of many other compression techniques.
- Uses the binary tree data structure to derive binary codes for each symbol in the alphabet.
- Reduces the total number of bits required for transmission of the message.

Let's look at the motivation for Huffman coding

- Suppose that we have the following six symbols in an alphabet:

t u $ e a w

- For transmitting a message composed of the above symbols, we need to encode each symbol in binary.

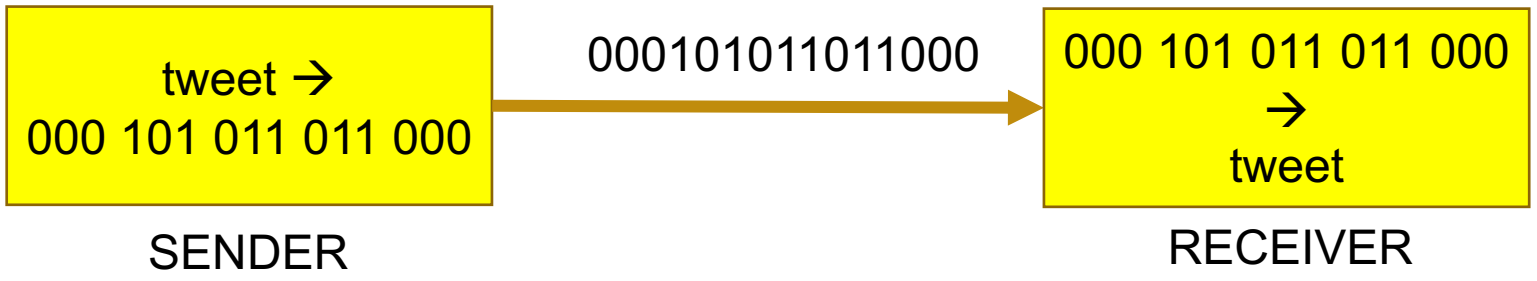
- How many bits do we need to encode each symbol?
- Using a naïve method, we say we need 3 bits to encode each symbol.

Reason: There are 6 symbols; with 3 bits we get 2^3 combinations that can encode up to 8 symbols.

Here's how we could encode the symbols

SYMBOL	CODE
t	000
u	001
$	010
e	011
a	100
w	101

With the above table, a message 'tweet' for instance would be transmitted in binary as follows:



Problem with the naïve encoding method

- Each symbol is encoded with the same number of bits.
- It assumes that each symbol occurs with the same likelihood or probability.
- This isn't true and it can result in increased number of bits for transmission.
- Suppose that we know the probability of occurrence of each symbol.
- Can we devise a technique so that the most frequently occurring symbols are encoded with the least number of bits?
- The answer is the Huffman coding technique.

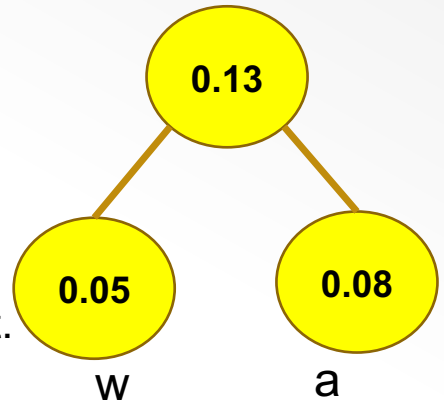
The Huffman coding technique

SYMBOL	PROBABILITY
w	0.05
a	0.08
u	0.12
$	0.20
t	0.20
e	0.35

- Suppose that the probability of the occurrence of each symbol in our example is known.
- We first arrange them in a table in increasing order of the probabilities.

The Huffman coding technique

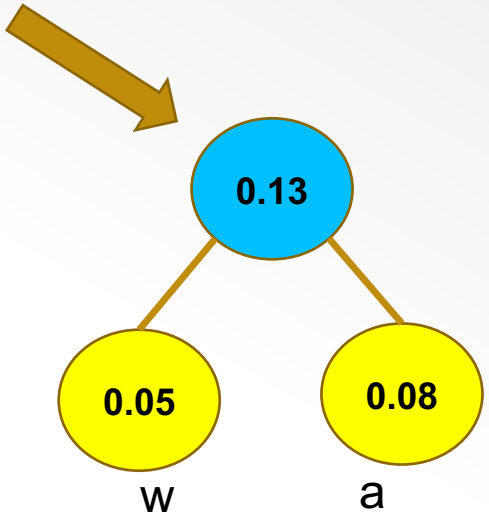
SYMBOL	PROBABILITY
~~w~~	~~0.05~~
~~a~~	~~0.08~~
u	0.12
$	0.20
t	0.20
e	0.35



- Pick the two symbols with the smallest probabilities and strike them off the list.
- Construct a one-node binary tree for each of these symbols.
- Next construct a sub-tree with three nodes. The root will have the combined probability and left and right subtrees will be the two one-node trees.

The Huffman coding technique

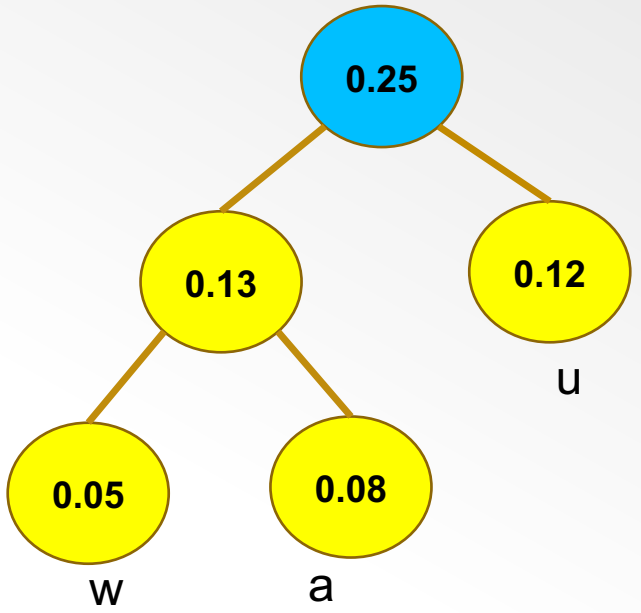
SYMBOL	PROBABILITY
~~w~~	~~0.05~~
~~a~~	~~0.08~~
u	0.12
$	0.20
t	0.20
e	0.35



- Again, we pick two smallest probabilities. But this time, the root(s) of the subtrees that we created must also be taken into consideration.
- 0.12 (u) and 0.13 are the smallest.

The Huffman coding technique

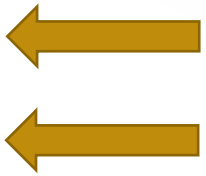
SYMBOL	PROBABILITY
~~w~~	~~0.05~~
~~a~~	~~0.08~~
~~u~~	~~0.12~~
$	0.20
t	0.20
e	0.35

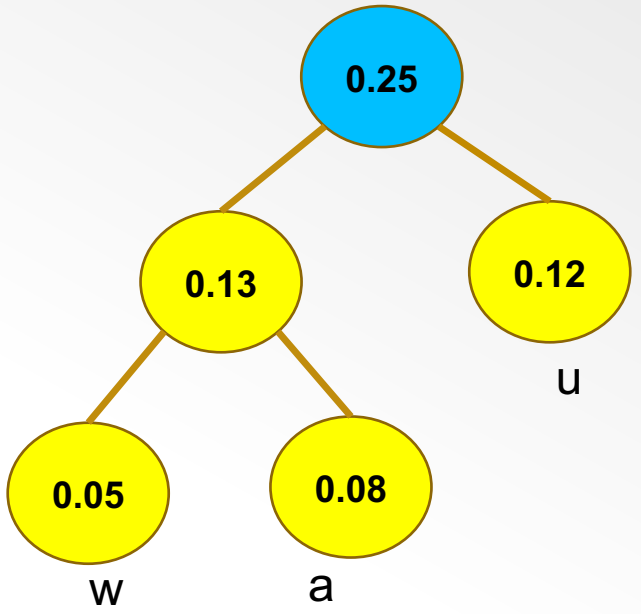


- Combine 0.12 (u) and 0.13 to form a subtree.
- Strike u off the list.

The Huffman coding technique

SYMBOL	PROBABILITY
~~w~~	~~0.05~~
~~a~~	~~0.08~~
~~u~~	~~0.12~~
$	0.20
t	0.20
e	0.35

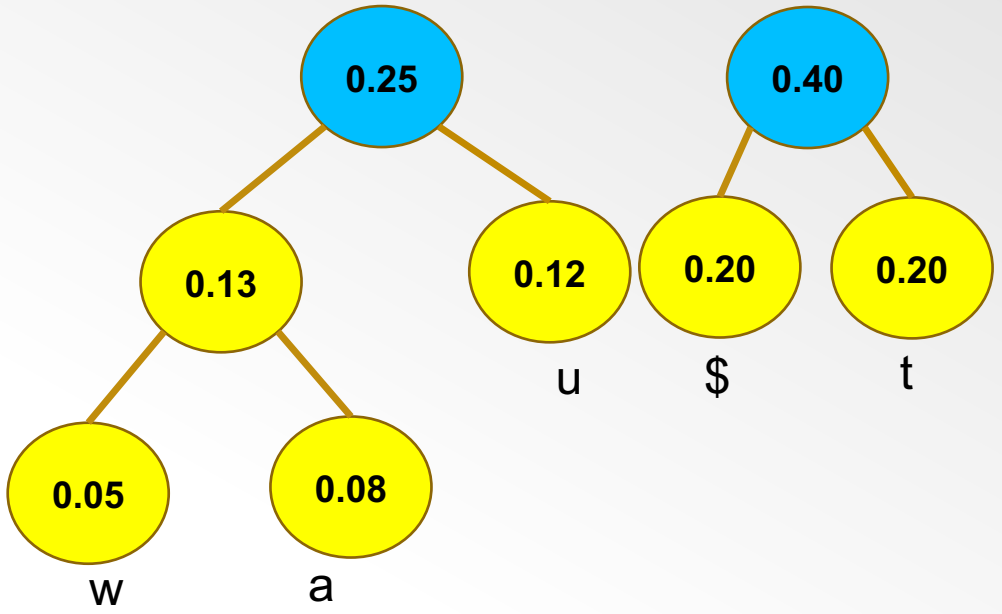




- Now we have 0.20, 0.20 and 0.25 as contenders.
- Form the subtree with 0.20 ($) and 0.20 (t) since they are the smallest probabilities.

The Huffman coding technique

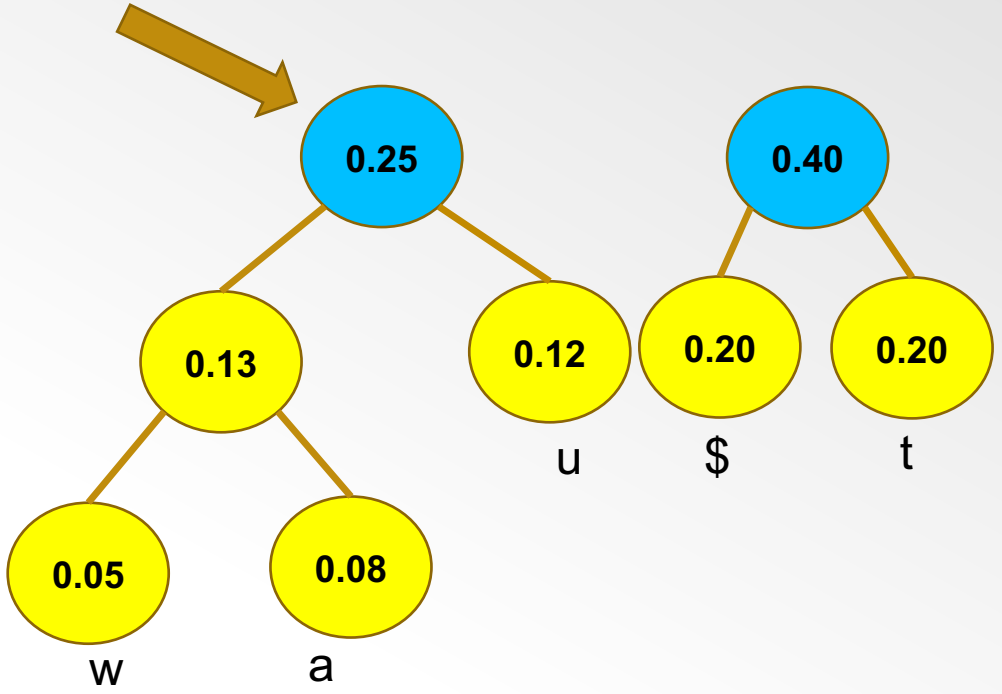
SYMBOL	PROBABILITY
~~w~~	~~0.05~~
~~a~~	~~0.08~~
~~u~~	~~0.12~~
~~$~~	~~0.20~~
~~t~~	~~0.20~~
e	0.35



- Now we have 0.20, 0.20 and 0.25 as contenders.
- Form the subtree with 0.20 ($) and 0.20 (t) since they are the smallest probabilities.
- Strike $ and t off the list.

The Huffman coding technique

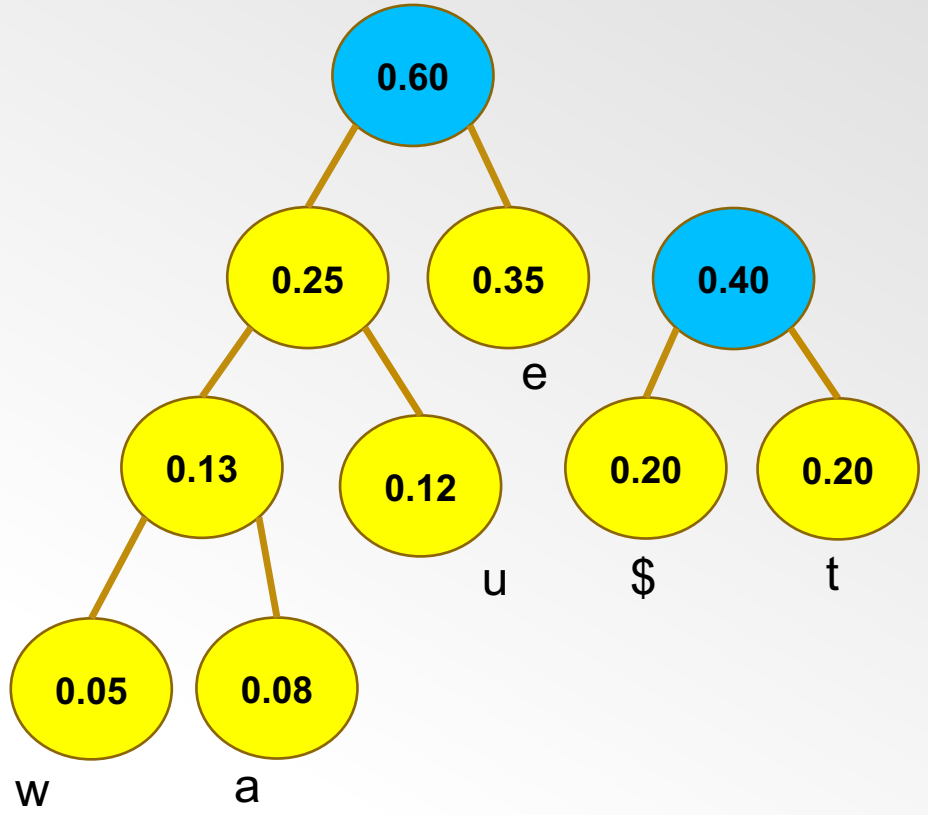
SYMBOL	PROBABILITY
~~w~~	~~0.05~~
~~a~~	~~0.08~~
~~u~~	~~0.12~~
~~$~~	~~0.20~~
~~t~~	~~0.20~~
e	0.35



- Now we have 0.25, 0.35 and 0.40 as contenders.
- Form the subtree with 0.25 and 0.35 (e) since they are the smallest probabilities.

The Huffman coding technique

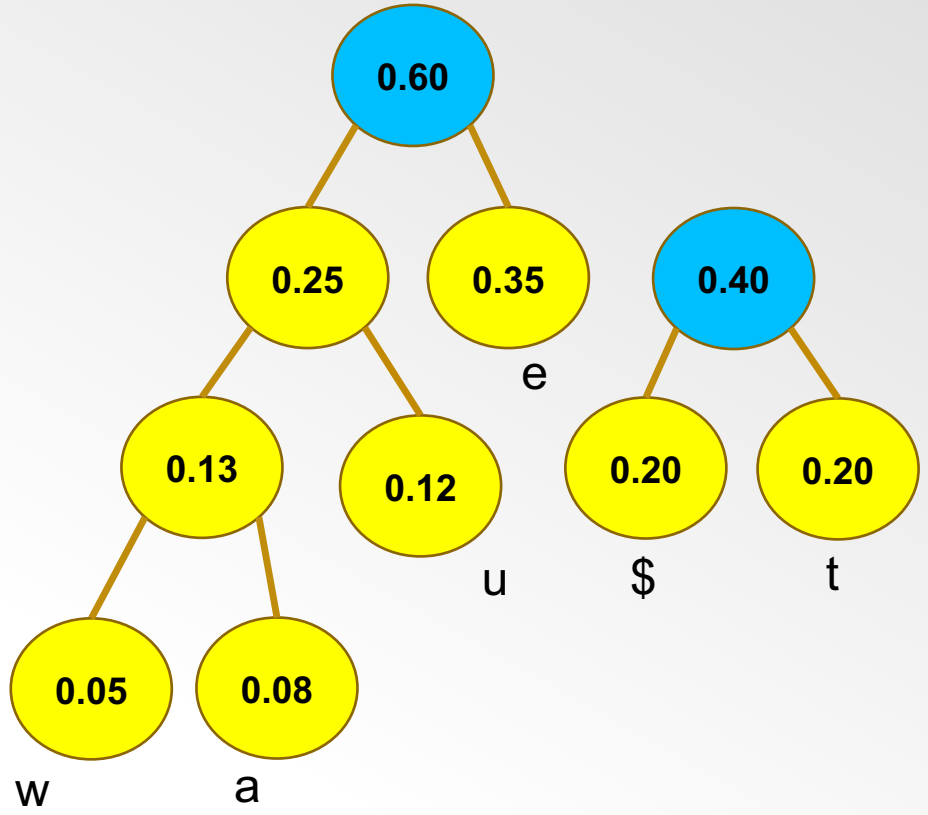
SYMBOL	PROBABILITY
~~w~~	~~0.05~~
~~a~~	~~0.08~~
~~u~~	~~0.12~~
~~$~~	~~0.20~~
~~t~~	~~0.20~~
~~e~~	~~0.35~~



- Now we have 0.25, 0.35 and 0.40 as contenders.
- Form the subtree with 0.25 and 0.35 (e) since they are the smallest probabilities.
- Strike e off the list.

The Huffman coding technique

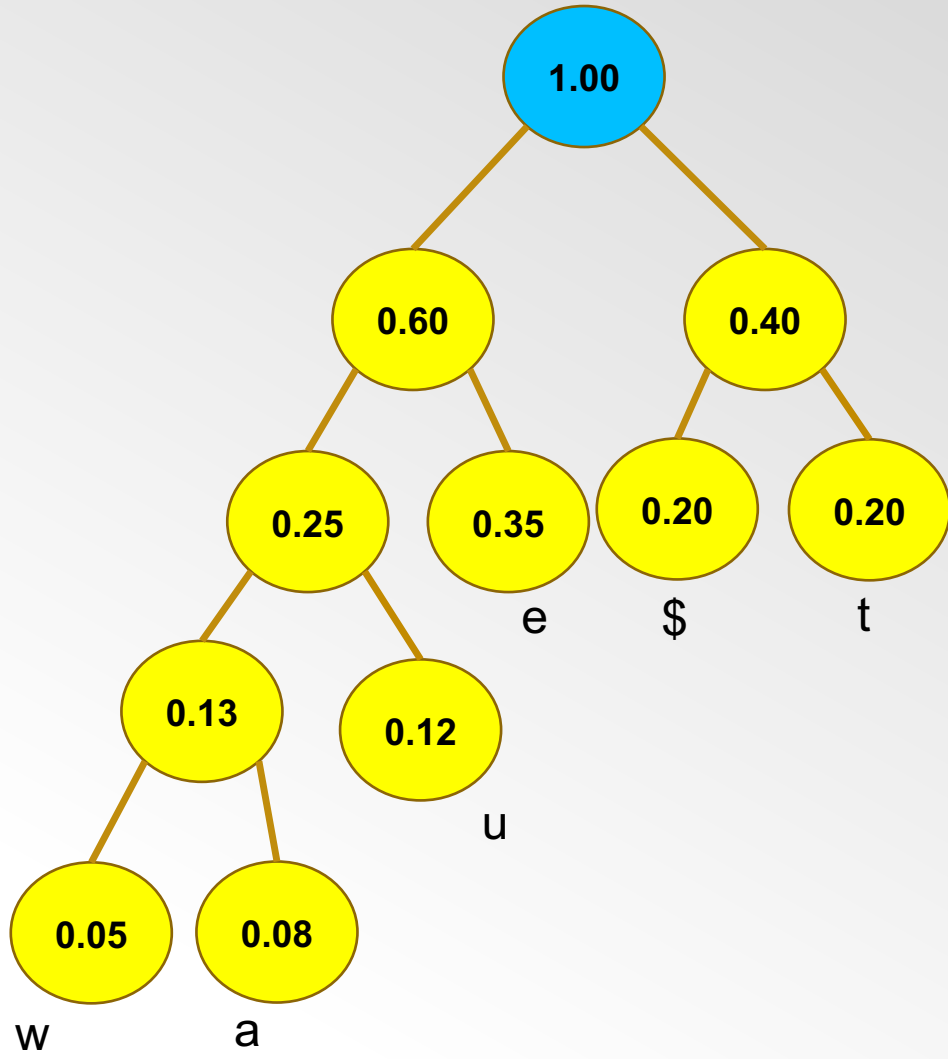
SYMBOL	PROBABILITY
~~w~~	~~0.05~~
~~a~~	~~0.08~~
~~u~~	~~0.12~~
~~$~~	~~0.20~~
~~t~~	~~0.20~~
~~e~~	~~0.35~~



- Now we have 0.25, 0.35 and 0.40 as contenders.
- Form the subtree with 0.25 and 0.35 (e) since they are the smallest probabilities.
- Strike e off the list.

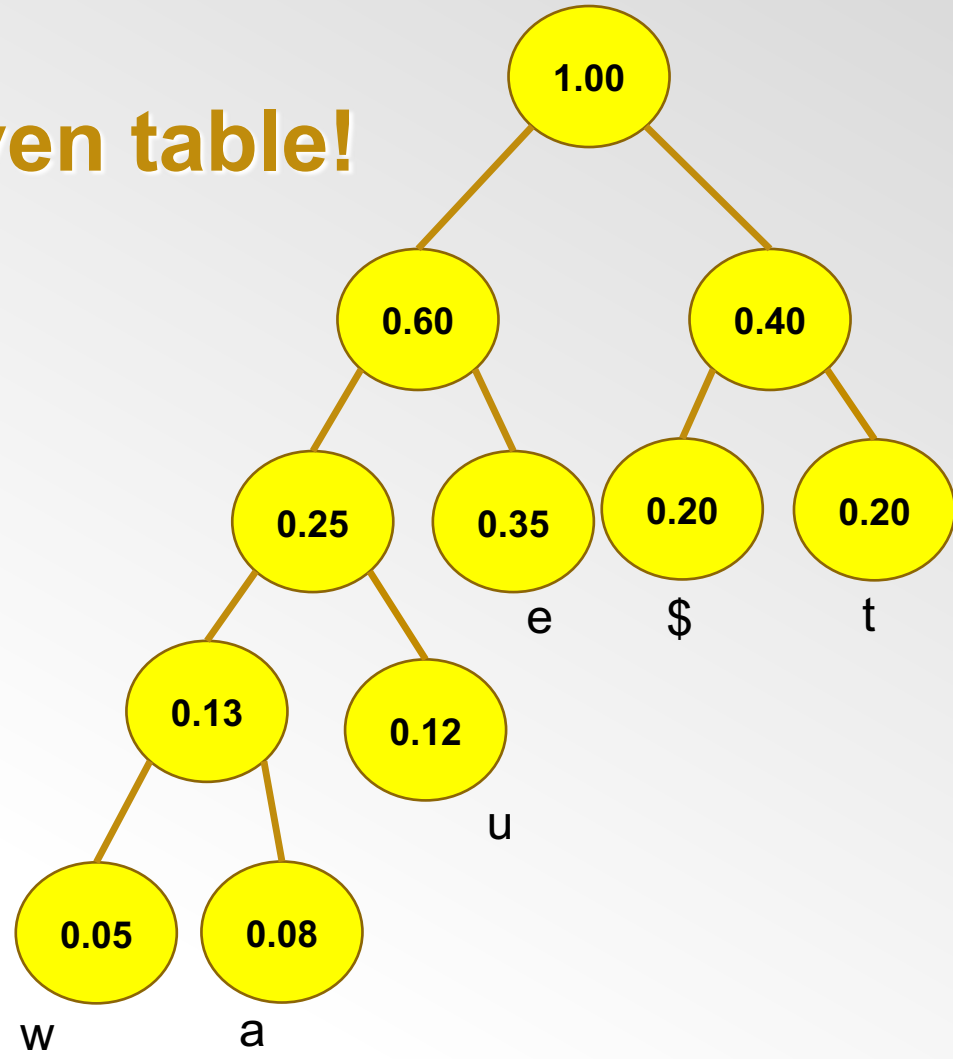
The Huffman coding technique

SYMBOL	PROBABILITY
w	0.05
a	0.08
u	0.12
$	0.20
t	0.20
e	0.35



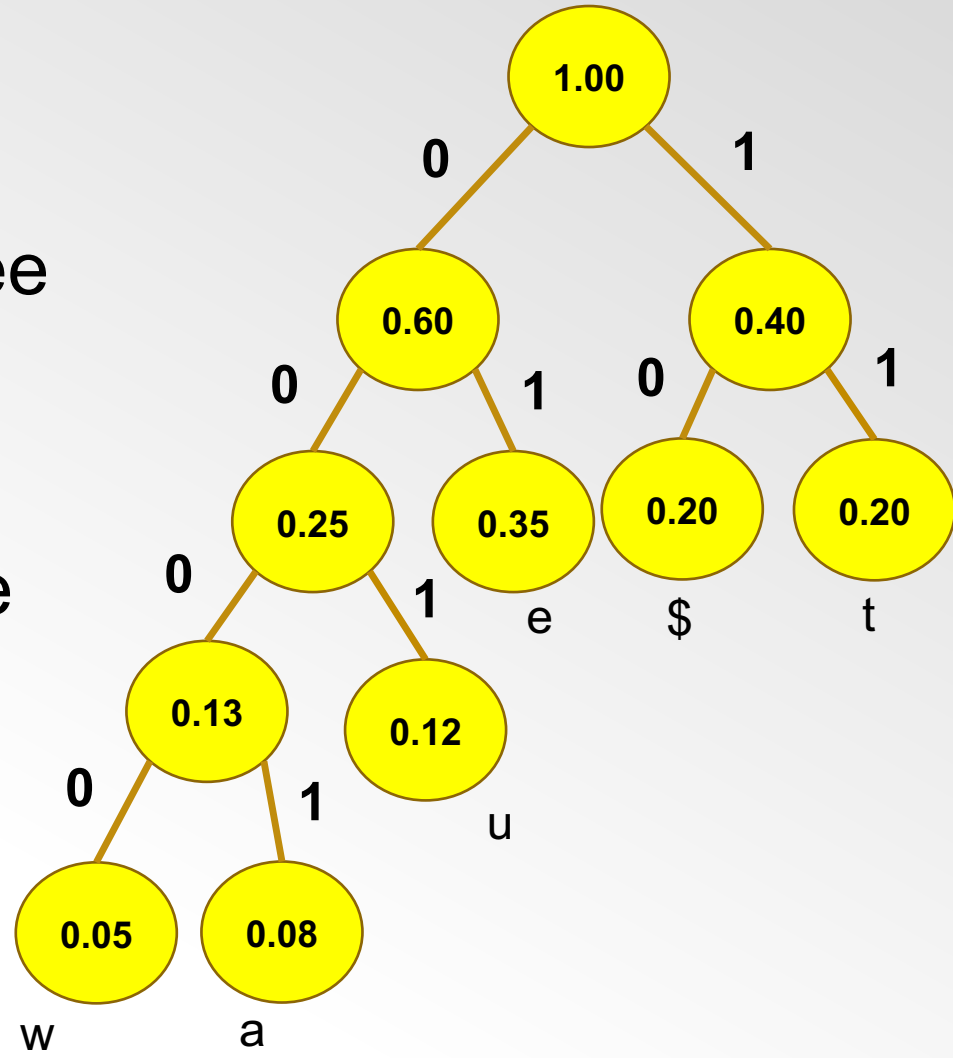
- Combine the two remaining nodes.
- The final root of the tree will have a value of 1.00.

This is your Huffman tree for the given table!



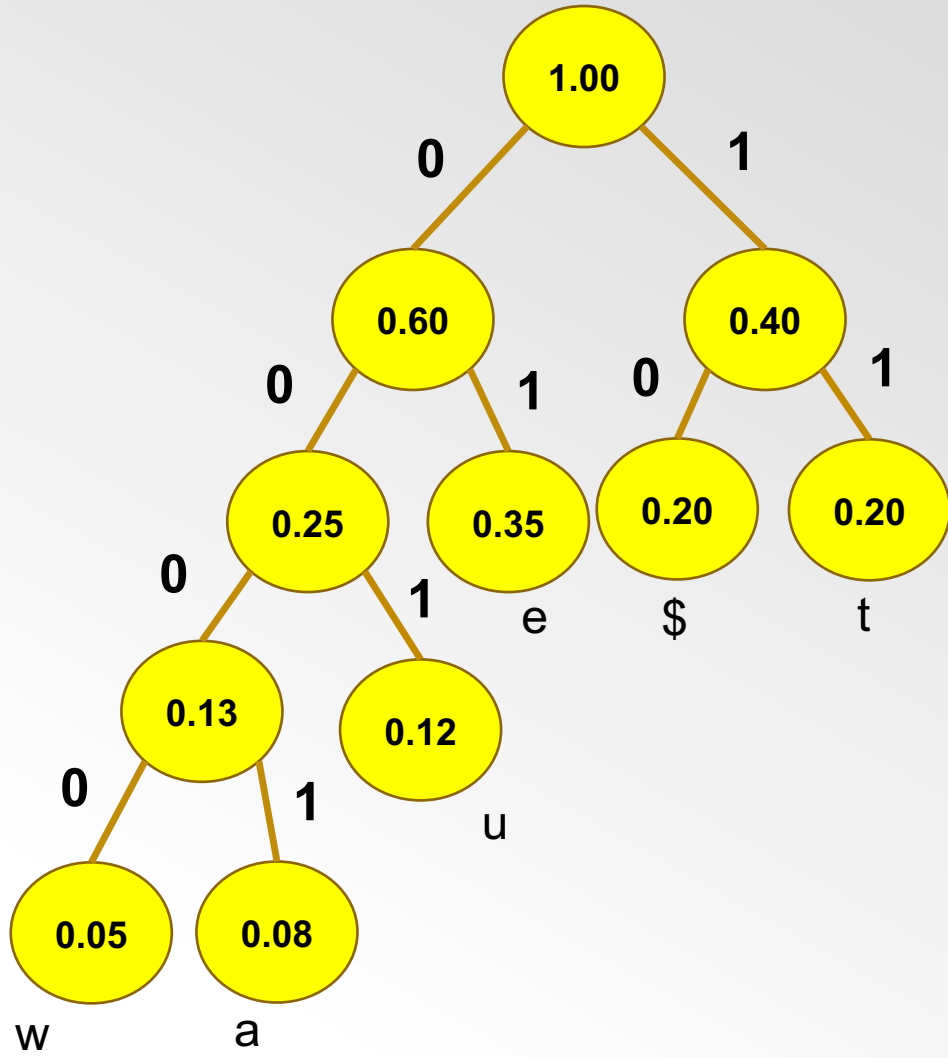
Huffman codes

- Encode every left branch of the Huffman tree with a 0 and every right branch with a 1.
- Read off the codes for each symbol starting from the root of the tree moving down to the leaf.
- For example, the code for w is 0000.
- As another example, the code for e is 01.



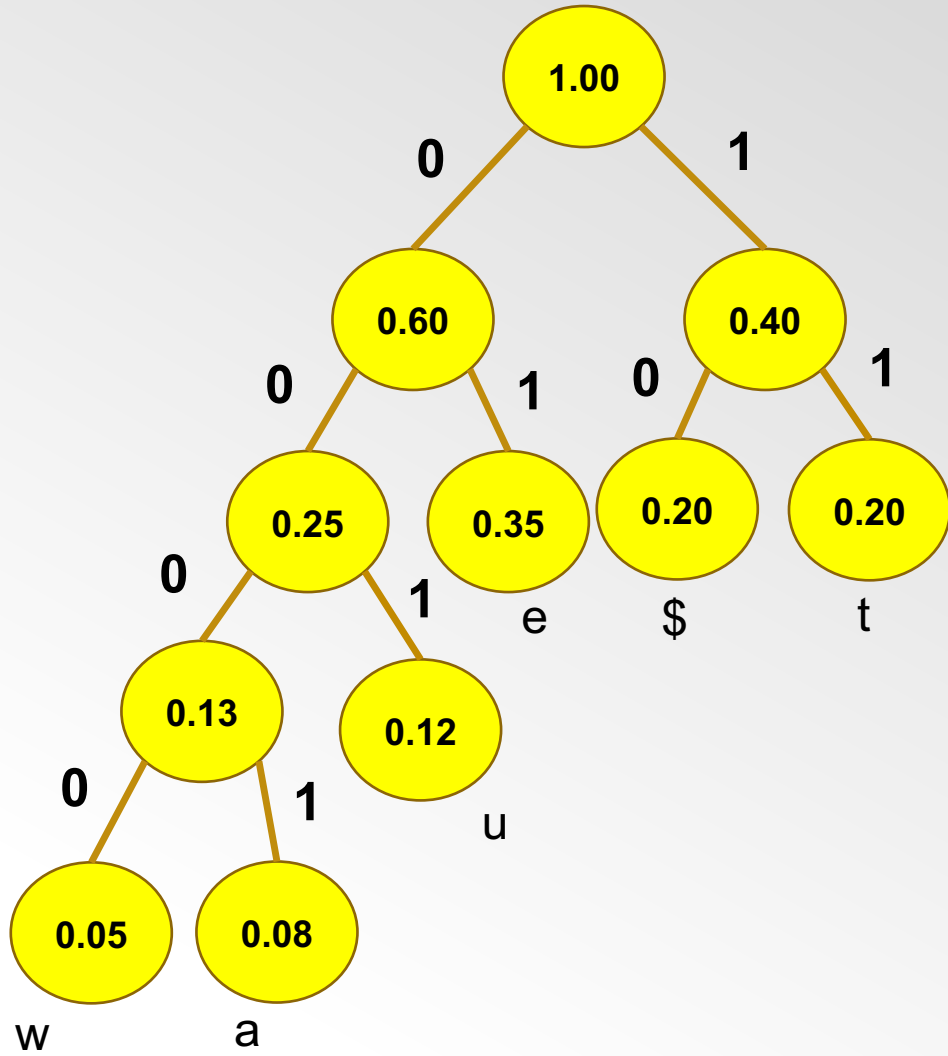
Huffman codes

SYMBOL	CODE
w	0000
a	0001
u	001
$	10
t	11
e	01



Huffman codes

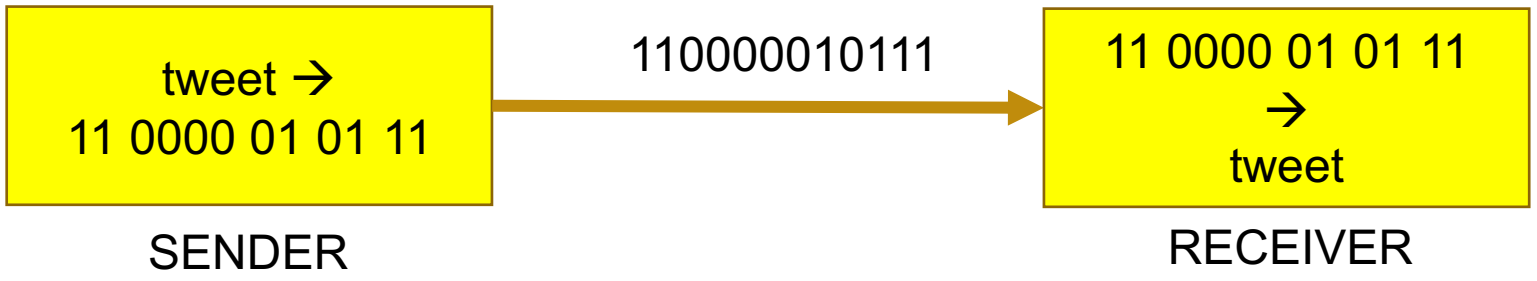
SYMBOL	CODE
w	0000
a	0001
u	001
$	10
t	11
e	01



- ***The most frequently occurring symbols get encoded with the smallest number of bits.***
- What is the secret sauce in Huffman encoding?
- ***No code is a prefix of another. This will guarantee unambiguous decoding.***

Here's how the message 'tweet' will be encoded

SYMBOL	CODE
w	0000
a	0001
u	001
$	10
t	11
e	01

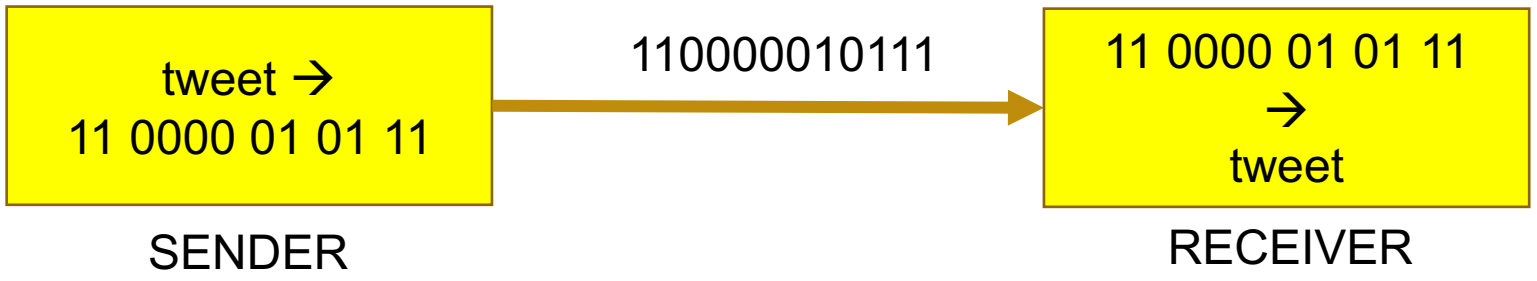


Here's how the message 'tweet' will be encoded

SYMBOL	CODE
w	0000
a	0001
u	001
$	10
t	11
e	01

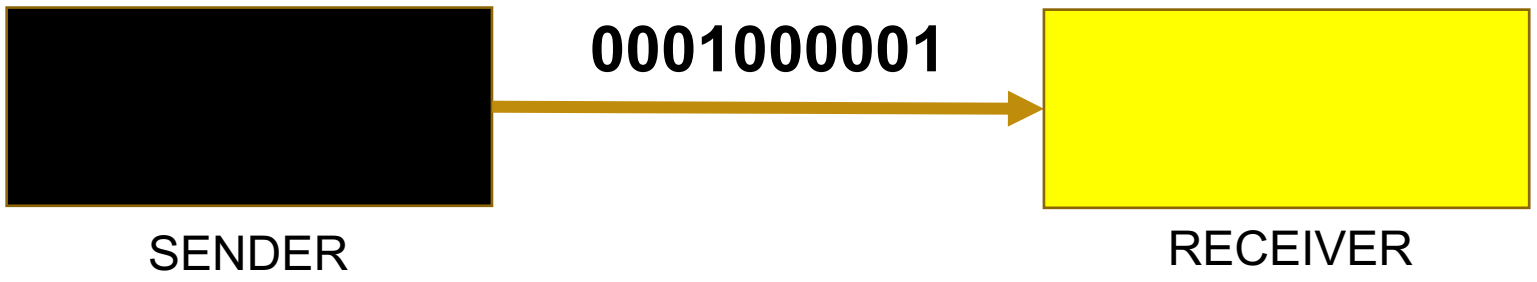
The naïve method uses 15 bits to encode 'tweet'

With Huffman coding, we use only 12 bits.



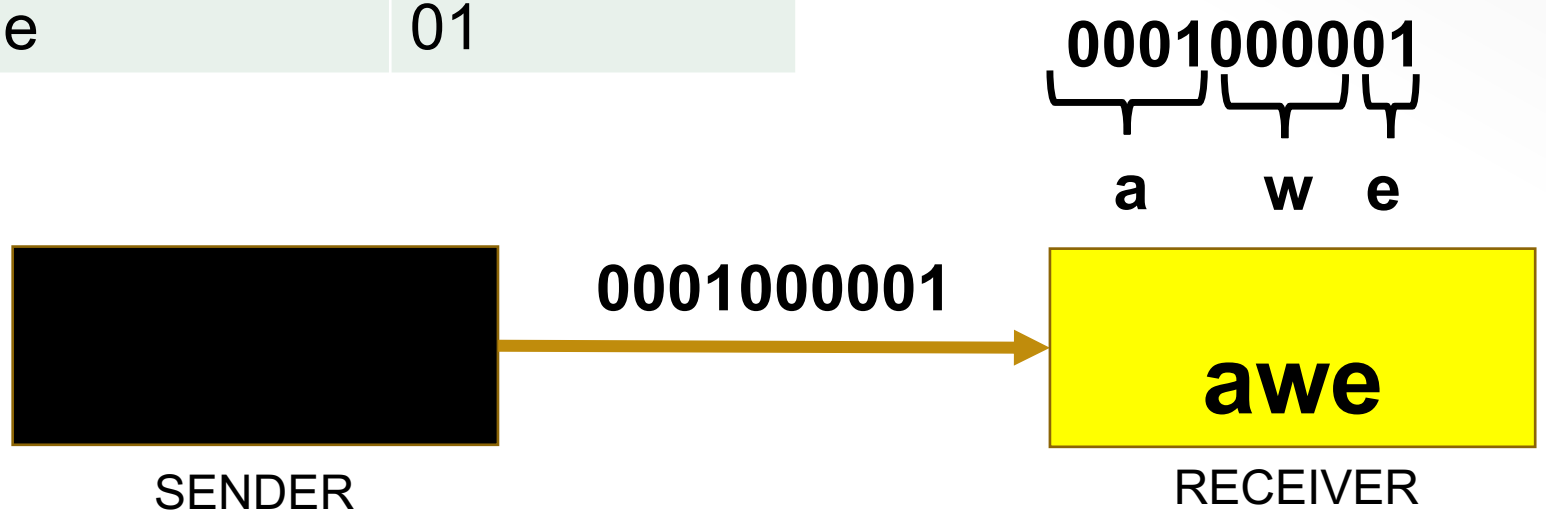
Exercise: What is the message transmitted in this example?

SYMBOL	CODE
w	0000
a	0001
u	001
$	10
t	11
e	01



Exercise: What is the message transmitted in this example?

SYMBOL	CODE
w	0000
a	0001
u	001
$	10
t	11
e	01



A formula to calculate the average code length with Huffman coding

Average code length = $\Sigma P_i C_i$

where P_i is the probability of symbol i
and C_i is the number of bits in the code for symbol i

SYMBOL	CODE
w	0000
a	0001
u	001
$	10
t	11
e	01

In our example,
Average code length with Huffman
=
4(0.05) + 4(0.08)+ 3(0.12)+2(0.20)+2(0.20)+2(0.35)
=
2.38 bits
Code Length without Huffman = 3 bits

Huffman ratio = (Avg. code length with Huffman)/
(Code length without Huffman)

Compression from Huffman coding

To transmit 1 million letters,
Without Huffman: 3,000,000 bits
With Huffman: 2,380,000 bits
This is the savings in transmission!

SYMBOL	CODE
w	0000
a	0001
u	001
$	10
t	11
e	01

Huffman Coding Example 2: Given the following table of symbols with their probabilities of occurrence, construct the Huffman tree and derive the codes for each symbol. Also determine the Huffman compression ratio.

Symbol	Probability
@	0.02
?	0.03
$	0.07
W	0.08
A	0.10
U	0.10
S	0.10
N	0.12
T	0.13
E	0.25

Pseudocode for implementing the Huffman tree

1. Construct a single node binary tree for each of the symbols. Place these binary trees in a queue S, in increasing order of probability. Let T be the result queue, initially empty.
2. Pick the two smallest weight trees, say A and B, from S and T, as follows:
    a) If T is empty, A and B are respectively the front and next to front entries of S. Dequeue them from S.
    b) If T is not empty,
    - Find the smaller weight tree of the trees in front of S and in front of T. This is A. Dequeue it.
    - Find the smaller weight tree of the trees in front of S and in front of T. This is B. Dequeue it.
3. Construct a new tree P by creating a root and attaching A and B as the subtrees of this root. The weight of the root is the combined weights of the roots of A and B.
4. Enqueue P to T.
5. Repeat steps 2 to 4 until S is empty.

SYMBOL	PROBABILITY
w	**0.05**
a	**0.08**
u	**0.12**
$	**0.20**
t	**0.20**
e	**0.35**