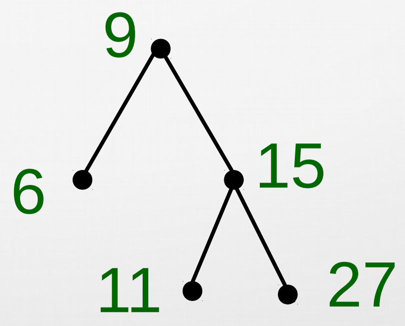
11.2 *Applications of Trees*

Binary Search Trees

Binary search tree, or BST, is a binary tree with *keys* assigned to vertices/nodes, where every vertex/node *v* has the following property:

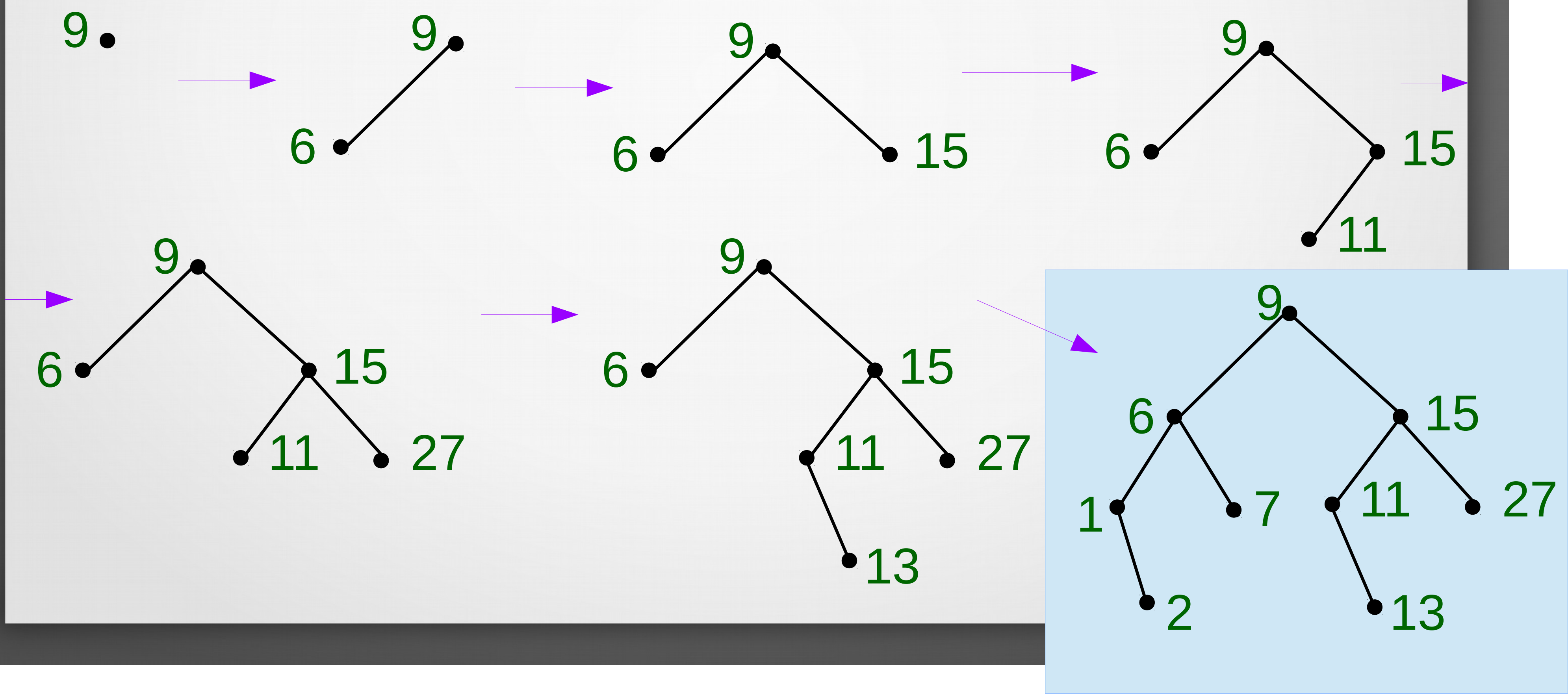
- each *key* in the left subtree is less than the *key* at the vertex *v*,
- each *key* in the right subtree is greater than the *key* at the vertex *v*.



11.2 *Applications of Trees*

Binary Search Trees

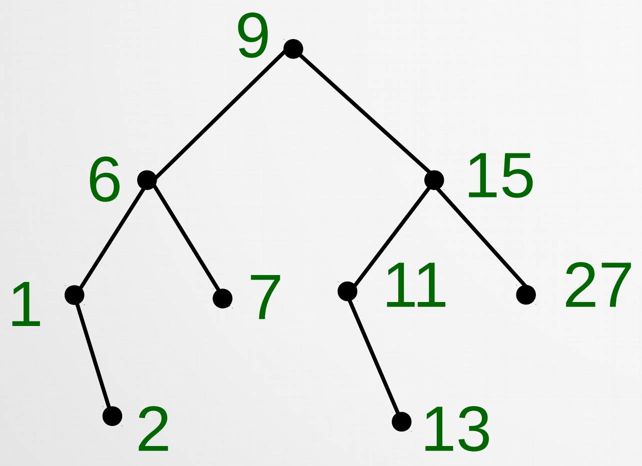
Inserting keys into a BST (one by one), starting with an empty tree: 9, 6, 15, 11, 27, 13, 1, 7, and 2.



11.2 *Applications of Trees*

Binary Search Trees

Inserting keys into a BST (one by one), starting with an empty tree: 9, 6, 15, 11, 27, 13, 1, 7, and 2.

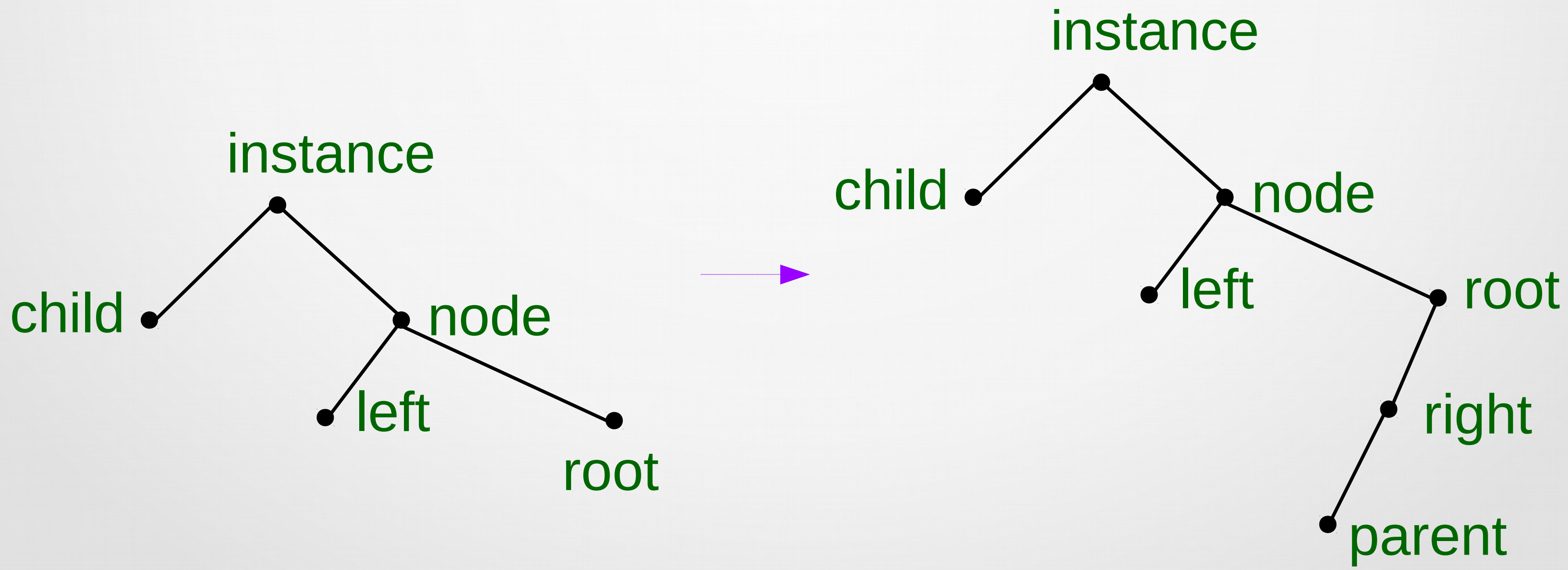


No duplicates can be inserted!

11.2 *Applications of Trees*

Binary Search Trees

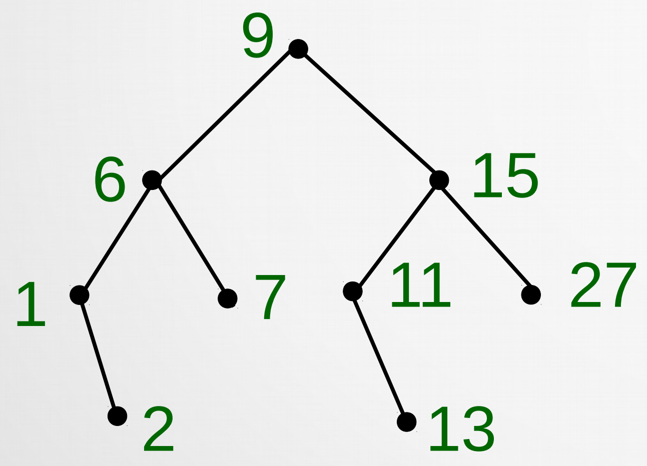
Inserting word keys into a BST (one by one), starting with an empty tree: instance, node, root, child, left, right, and parent.



11.2 *Applications of Trees*

Binary Search Trees

See the pseudocode for inserting an item into a BST on page 759 in the book.

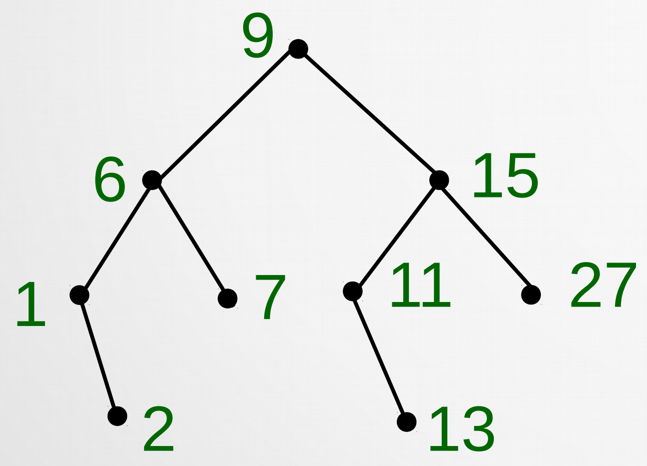


a *balanced binary tree*

11.2 *Applications of Trees*

Binary Search Trees

See the pseudocode for inserting an item into a BST on page 759 in the book.



To determine the computational complexity of *insertion operation* let's see how many comparisons are performed in a worst-case scenario: *the item is added to the longest path's leaf*.

length of the longest path = height of the tree =

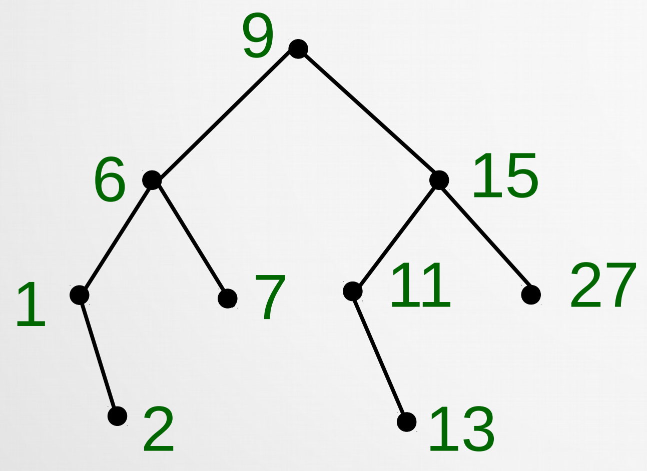
a balanced tree: $= \lceil \log_2(n+1) \rceil = \lceil \log(n+1) \rceil$ comparisons

an unbalanced tree: $= n$ comparisons

11.2 *Applications of Trees*

Binary Search Trees

See the pseudocode for inserting an item into a BST on page 759 in the book.



a *balanced binary tree*

Algorithms have been devised to re-balance a BST when new items are added.

- this is covered in CSI33, as well as deletion of vertices from a BST tree.

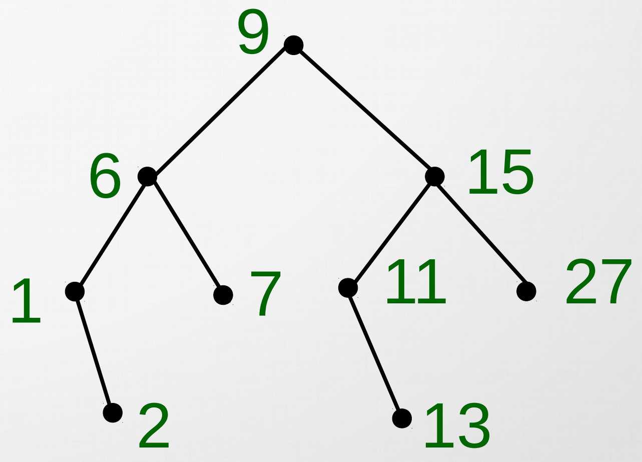
11.2 *Applications of Trees*

Decision Trees

Rooted trees can be used to model problems in which a series of decisions leads to a solution.

Example: BST can be used to locate items based on a series of comparisons.

Let's check if 11 is in the BST (searching for 11)



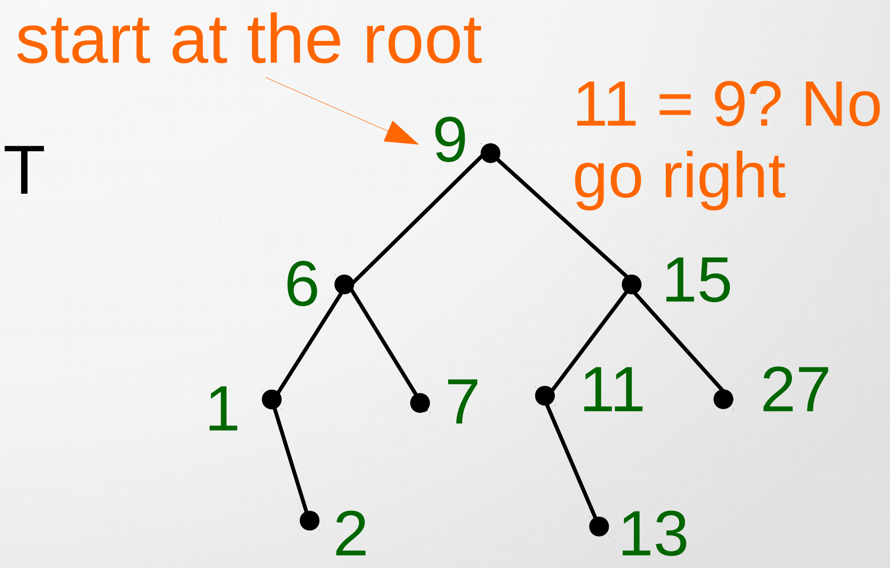
11.2 *Applications of Trees*

Decision Trees

Rooted trees can be used to model problems in which a series of decisions leads to a solution.

Example: BST can be used to locate items based on a series of comparisons.

Let's check if 11 is in the BST (searching for 11)



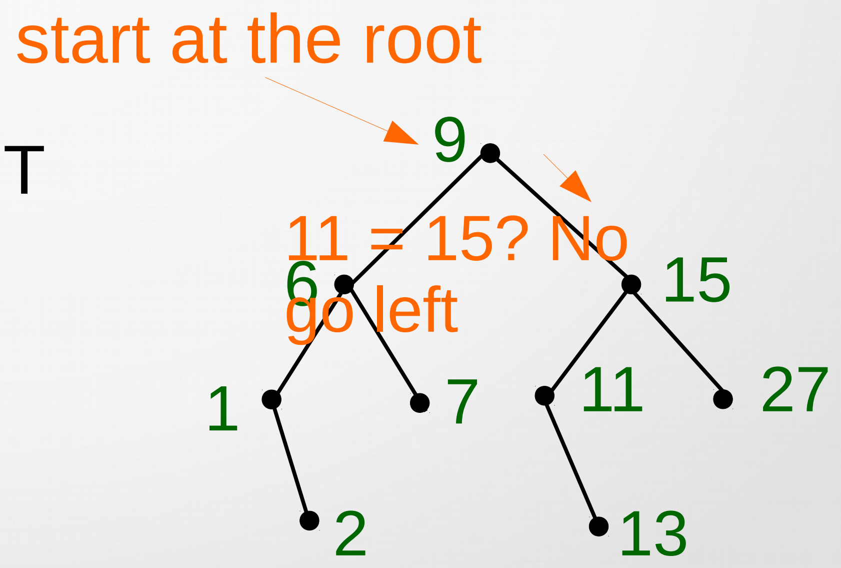
11.2 *Applications of Trees*

Decision Trees

Rooted trees can be used to model problems in which a series of decisions leads to a solution.

Example: BST can be used to locate items based on a series of comparisons.

Let's check if 11 is in the BST (searching for 11)



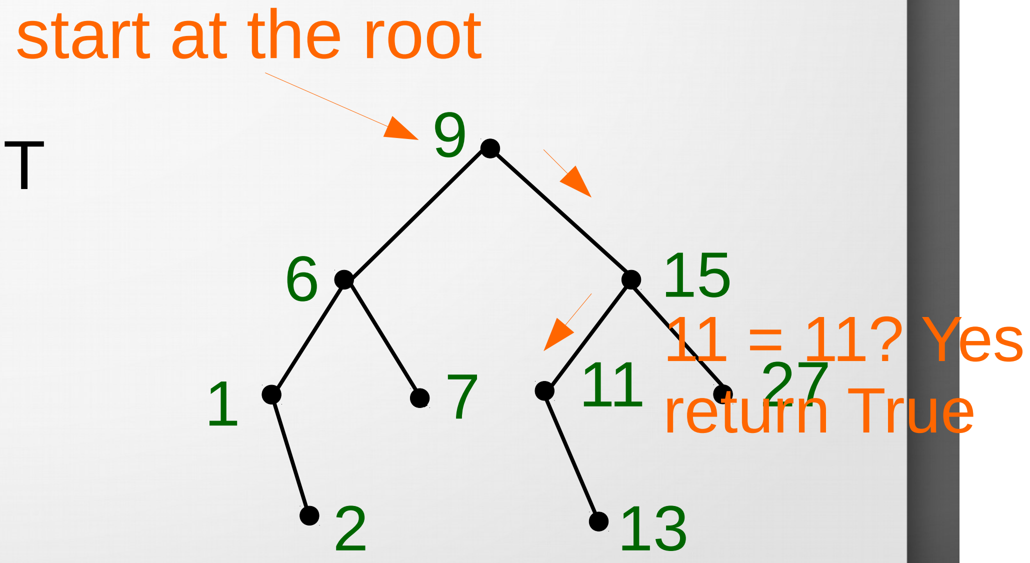
11.2 *Applications of Trees*

Decision Trees

Rooted trees can be used to model problems in which a series of decisions leads to a solution.

Example: BST can be used to locate items based on a series of comparisons.

Let's check if 11 is in the BST (searching for 11)



11.2 *Applications of Trees*

Decision Trees

Rooted trees can be used to model problems in which a series of decisions leads to a solution.

A rooted tree in which each internal vertex corresponds to a *decision*, with a subtree at these vertices for each possible outcome of the decision, is called a *decision tree*.

The *possible solutions of the problem* correspond to the paths to the leaves of this rooted tree.

11.2 *Applications of Trees*

Decision Trees

Example: suppose there are seven coins of the same weight, and a counterfeit coin that weighs less than the others.
How many weighings are necessary using a balance scale to determine which coin is the counterfeit one?
Give an algorithm.

11.2 *Applications of Trees*

Decision Trees

Example: suppose there are seven coins of the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which coin is the counterfeit one? Give an algorithm.

Solution: 8 coins, and balance scale.

11.2 *Applications of Trees*

Decision Trees

Example: suppose there are seven coins of the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which coin is the counterfeit one? Give an algorithm.

Solution: 8 coins, and balance scale.
3 outcomes for weighing two coins:
equal, left pan is heavier, right pan is heavier.
Let's use 3-ary decision tree.

11.2 *Applications of Trees*

Decision Trees

Example: suppose there are seven coins of the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which coin is the counterfeit one? Give an algorithm.

Solution: 8 coins, and balance scale.
3 outcomes for weighing two coins:
equal, left pan is heavier, right pan is heavier.
Let's use 3-ary decision tree.
There are 8 possible outcomes, hence at least 8 leaves.

11.2 *Applications of Trees*

Decision Trees

Example: suppose there are seven coins of the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which coin is the counterfeit one? Give an algorithm.

Solution: 8 coins, and balance scale.
3 outcomes for weighing two coins:
equal, left pan is heavier, right pan is heavier.
Let's use 3-ary decision tree.
There are 8 possible outcomes, hence at least 8 leaves.
Height of the decision tree = largest number of weighings

11.2 *Applications of Trees*

Decision Trees

Example: suppose there are seven coins of the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which coin is the counterfeit one? Give an algorithm.

Solution: 8 coins, and balance scale.
3 outcomes for weighing two coins:
equal, left pan is heavier, right pan is heavier.
Let's use 3-ary decision tree.
There are 8 possible outcomes, hence at least 8 leaves.
Height of the decision tree = largest number of weighings

$h \geq \lceil \log_3 8 \rceil = 2$

$h \geq \lceil \log_m l \rceil$ from Corolary (Section 11.1)

11.2 *Applications of Trees*

Decision Trees

Example: suppose there are seven coins of the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which coin is the counterfeit one? Give an algorithm.

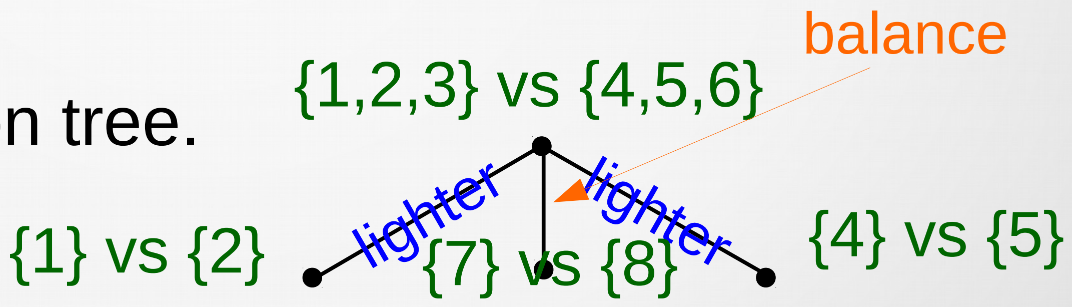
Solution: 8 coins, and balance scale.
3 outcomes for weighing two coins:
equal, left pan is heavier, right pan is heavier.
Let's use 3-ary decision tree.
There are 8 possible outcomes, hence at least 8 leaves.
Height of the decision tree = largest number of weighings
$h \geq \lceil \log_3 8 \rceil = 2$ Therefore at least 2 weighing are needed.

11.2 *Applications of Trees*

Decision Trees

Example: suppose there are seven coins of the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which coin is the counterfeit one? Give an algorithm.

Solution:
Let's use 3-ary decision tree.
There are 8 possible outcomes
at least 2 weighing are needed.

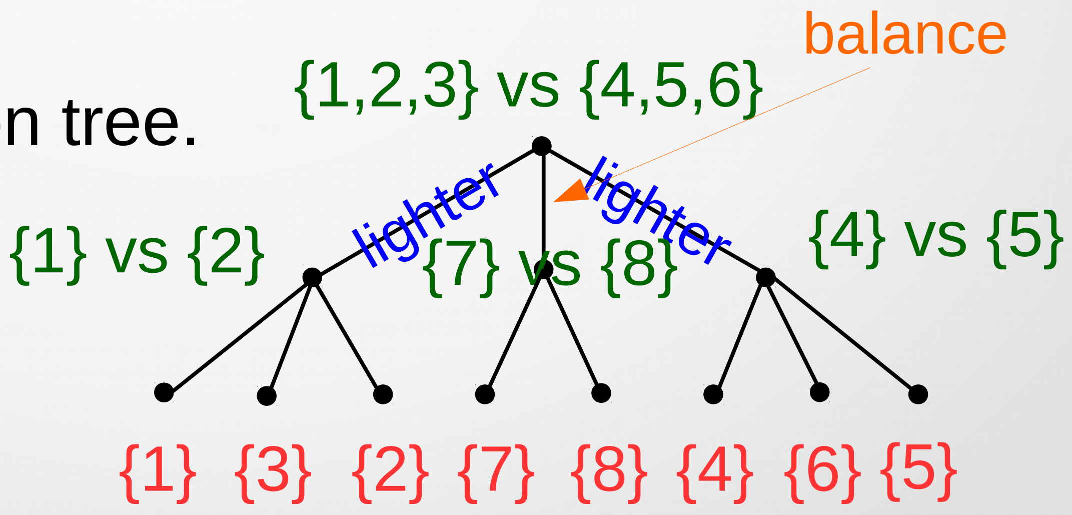


11.2 *Applications of Trees*

Decision Trees

Example: suppose there are seven coins of the same weight, and a counterfeit coin that weighs less than the others. How many weighings are necessary using a balance scale to determine which coin is the counterfeit one? Give an algorithm.

Solution:
Let's use 3-ary decision tree.
There are 8 possible outcomes
at least 2 weighing are needed.



11.2 *Applications of Trees*

Decision Trees

Many different sorting algorithms have been developed. To decide whether a particular algorithm is efficient, its complexity is determined.

Using decision trees as models, a lower bound for the worst-case complexity of sorting algorithms that are based on binary comparisons (two elements at a time) can be found.

11.2 *Applications of Trees*

Decision Trees

Many different sorting algorithms have been developed. To decide whether a particular algorithm is efficient, its complexity is determined.

Using decision trees as models, a lower bound for the worst-case complexity of sorting algorithms that are based on binary comparisons (two elements at a time) can be found.

Note: given *n* elements, there are *n!* possible orderings of them.

_   _   _   …   _

n   *(n-1)*   *(n-2)*   *…*   *1*

11.2 *Applications of Trees*

Decision Trees

Many different sorting algorithms have been developed. To decide whether a particular algorithm is efficient, its complexity is determined.

Using decision trees as models, a lower bound for the worst-case complexity of sorting algorithms that are based on binary comparisons (two elements at a time) can be found.

Note: given *n* elements, there are *n!* possible orderings of them.

$$\underset{n}{\_}\quad \underset{(n-1)}{\_}\quad \underset{(n-2)}{\_}\quad \underset{\ldots}{\cdots}\quad \underset{1}{\_}$$

So each leaf represents one of the *n!* orderings.

11.2 *Applications of Trees*

Decision Trees

Note: given *n* elements, there are *n!* possible orderings of them.

$$\underset{n}{\_} \quad \underset{(n-1)}{\_} \quad \underset{(n-2)}{\_} \quad \underset{\ldots}{\ldots} \quad \underset{1}{\_}$$

So each leaf represents one of the *n!* orderings.

By Corollary from Section 11.1, the height of the tree with *n!* leaves is at least $\lceil \log_2 n! \rceil$, i.e. $h \geq \lceil \log_2 n! \rceil$

The most comparisons used will be along the longest path, i.e. its length is *h*.

Therefore, at least $\lceil \log_2 n! \rceil$, comparisons are needed.

11.2 *Applications of Trees*

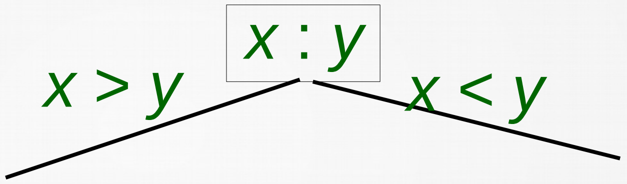
Decision Trees

Example: let's draw a decision tree that orders elements x,y and z.

$n = 3$

$h \geq \lceil \log_2 3! \rceil = \lceil \log_2 6 \rceil = 3$

So $h \geq 3$



11.2 *Applications of Trees*

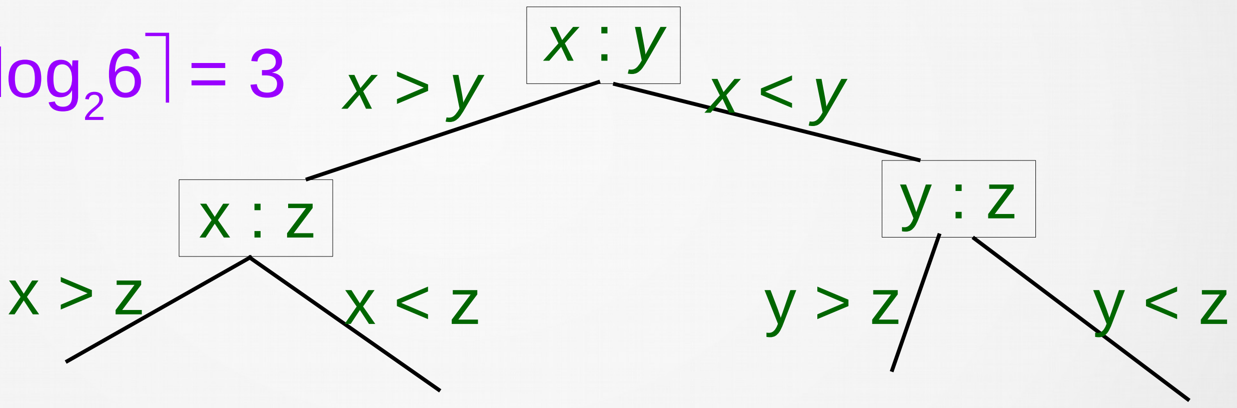
Decision Trees

Example: let's draw a decision tree that orders elements *x*,*y* and *z*.

$n = 3$

$h \geq \lceil \log_2 3! \rceil = \lceil \log_2 6 \rceil = 3$

So $h \geq 3$



11.2 *Applications of Trees*

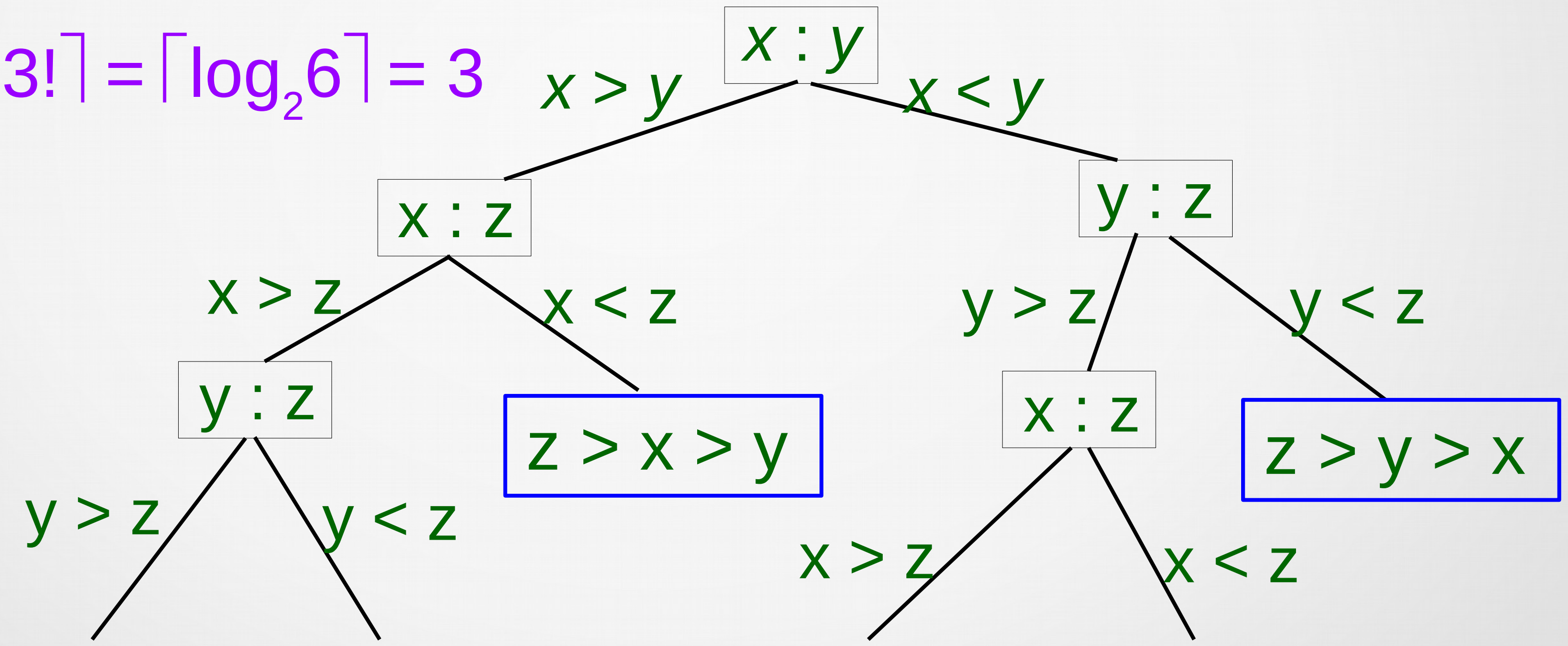
Decision Trees

Example: let's draw a decision tree that orders elements *x*,*y* and *z*.

$n = 3$

$h \geq \lceil \log_2 3! \rceil = \lceil \log_2 6 \rceil = 3$

So $h \geq 3$



11.2 *Applications of Trees*

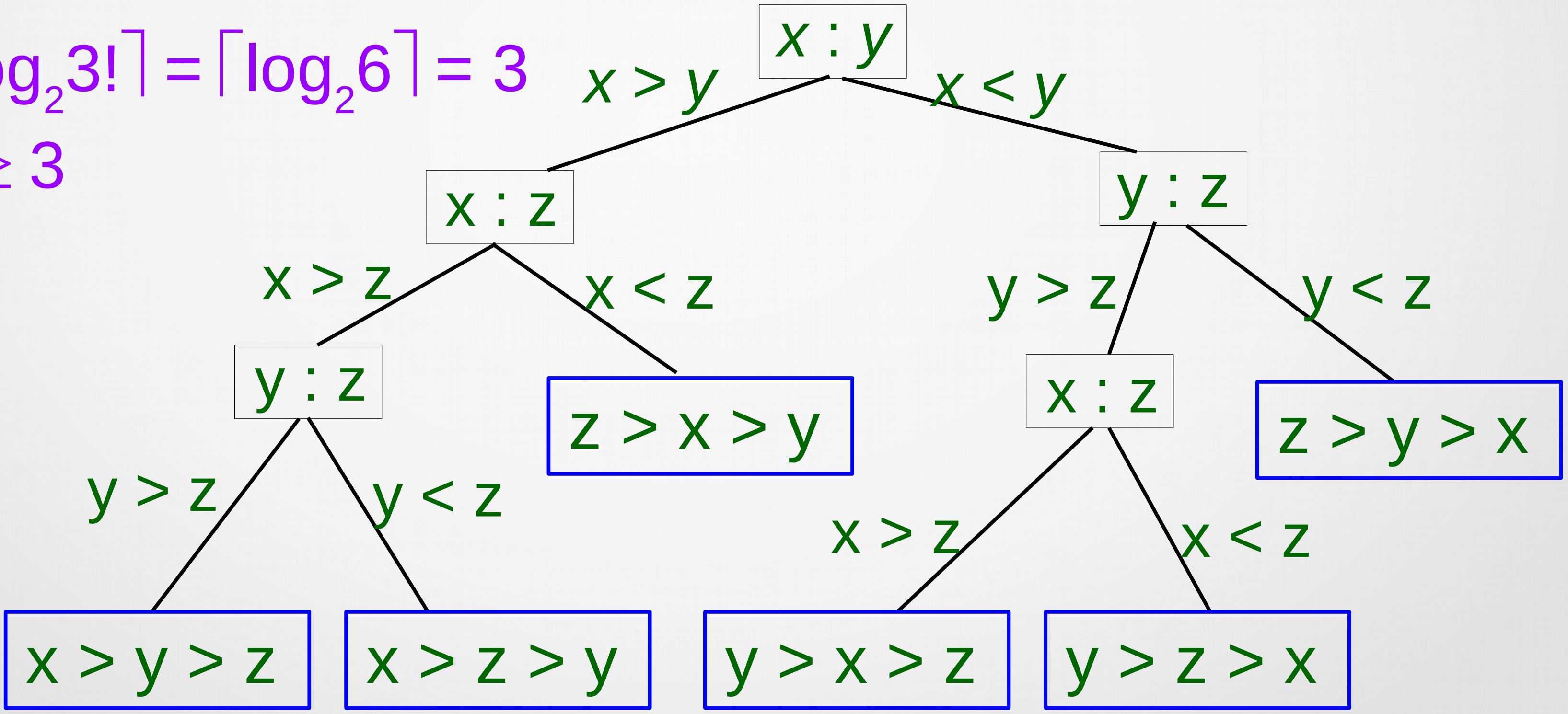
Decision Trees

Example: let's draw a decision tree that orders elements *x*,*y* and *z*.

$n = 3$

$h \geq \lceil \log_2 3! \rceil = \lceil \log_2 6 \rceil = 3$

So $h \geq 3$



11.2 *Applications of Trees*

Prefix Codes

Consider a problem of using bit strings to encode the letters of the English alphabet (not case sensitive)

We can use bit strings of length 5 : $2^5 = 32$ different bit strings of length 5 (26 letters in the alphabet)

$$\underline{\quad} \quad \underline{\quad} \quad \underline{\quad} \quad \underline{\quad} \quad \underline{\quad}$$

$$2 \times 2 \times 2 \times 2 \times 2 = 2^5 = 32$$

11.2 *Applications of Trees*

Prefix Codes

Consider a problem of using bit strings to encode the letters of the English alphabet (not case sensitive)

We can use bit strings of length 5 : $2^5 = 32$ different bit strings of length 5 (26 letters in the alphabet)

$$\underline{\quad\;\;}\quad \underline{\quad\;\;}\quad \underline{\quad\;\;}\quad \underline{\quad\;\;}\quad \underline{\quad\;\;}$$

$$2 \times 2 \times 2 \times 2 \times 2 \qquad = 2^5 = 32$$

Of course we would like to use fewer bits if possible, i.e. not every letter should get a bit-string of length 5!
So we will use bit strings of different lengths to encode letters. More frequently occurring letters will have shorter bit strings encoding.

11.2 *Applications of Trees*

Prefix Codes

Consider a problem of using bit strings to encode the letters of the English alphabet (not case sensitive)

We will use bit strings of different lengths to encode letters. More frequently occurring letters will have shorter bit strings encoding.

Notice, that we have to be careful, for example,
If *a* is encoded with 0, *b* with 1, and *c* with 01,
The bit string 10101 can correspond to:
bcc (1 01 01),
babc (1 0 1 01),
bcab (1 01 0 1), and so forth.

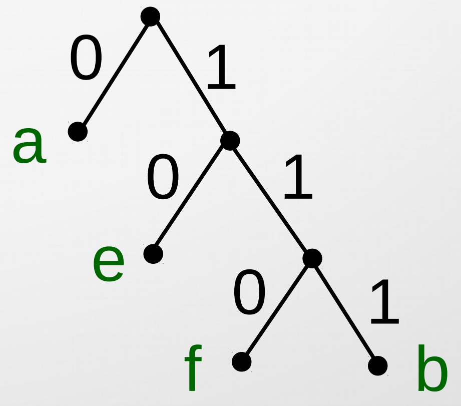
11.2 *Applications of Trees*

Prefix Codes

Consider a problem of using bit strings to encode the letters of the English alphabet (not case sensitive)

Let's consider *prefix codes*: letters are encoded in such a way that *the bit string for a letter never occurs as the first part of the bit string for another letter*.

Prefix codes can be represented by a binary tree, where leaves' labels are characters.
“left edge” is assigned 0, and
“right edge” is assigned 1.



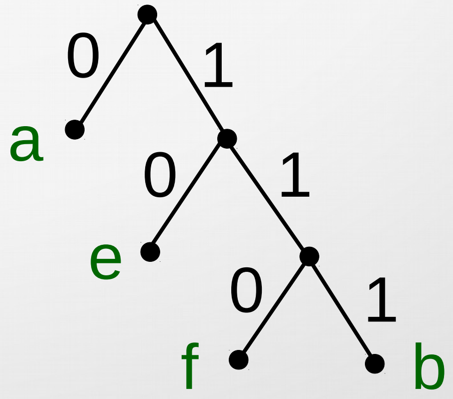
11.2 *Applications of Trees*

Prefix Codes

Consider a problem of using bit strings to encode the letters of the English alphabet (not case sensitive)

Let's consider *prefix codes*: letters are encoded in such a way that *the bit string for a letter never occurs as the first part of the bit string for another letter*.

Prefix codes can be represented by a binary tree, where leaves' labels are characters.
“left edge” is assigned 0, and
“right edge” is assigned 1.



<u>codes</u>
a: 0
e: 10
f: 110
b: 111

11.2 *Applications of Trees*

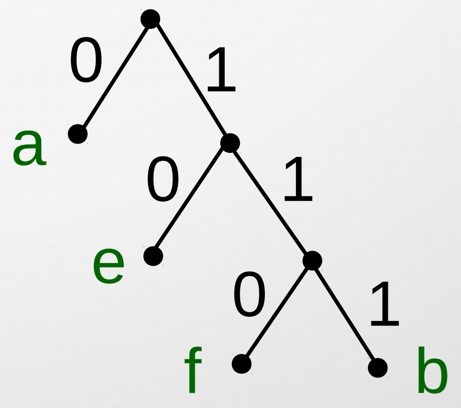
Prefix Codes

Consider a problem of using bit strings to encode the letters of the English alphabet (not case sensitive)

Let's consider *prefix codes*: letters are encoded in such a way that *the bit string for a letter never occurs as the first part of the bit string for another letter*.

Prefix codes can be represented by a binary tree, where leaves' labels are characters.
"left edge" is assigned 0, and
"right edge" is assigned 1.

The string 10101111100
corresponds to eebfa
10 10 111 110 0



11.2 *Applications of Trees*

Prefix Codes

Practice:

construct a binary tree with the following prefix codes

a: 1010
e: 0
t: 11
s: 1011
n: 1001
i: 10001

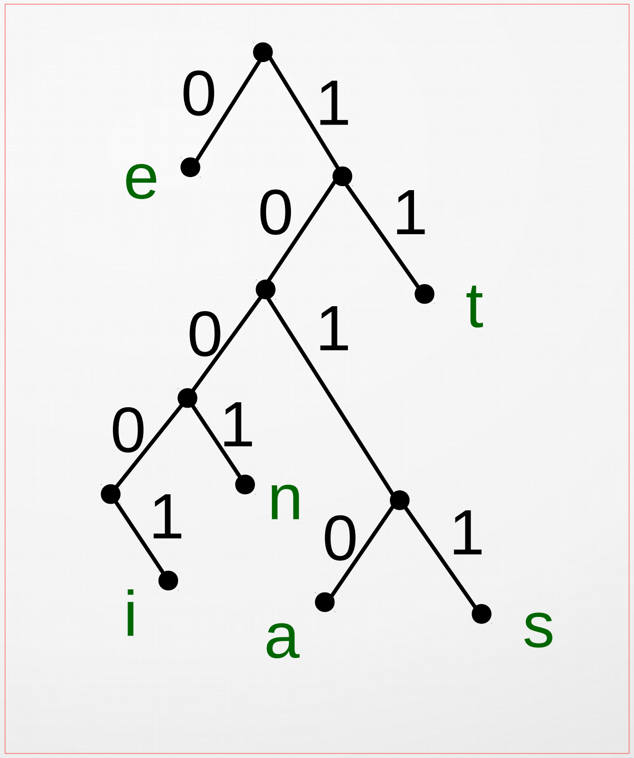
11.2 *Applications of Trees*

Prefix Codes

Practice:

construct a binary tree with the following prefix codes

a: 1010
e: 0
t: 11
s: 1011
n: 1001
i: 10001

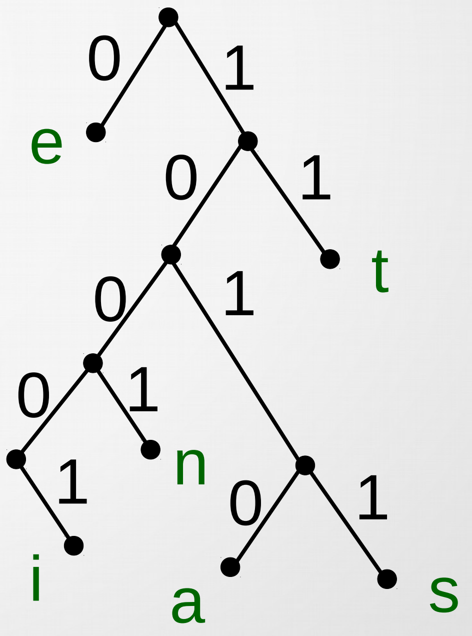


11.2 *Applications of Trees*

Prefix Codes

Practice:

Given the binary tree with the following prefix codes find the word represented by 10001100110111111010100111



11.2 *Applications of Trees*

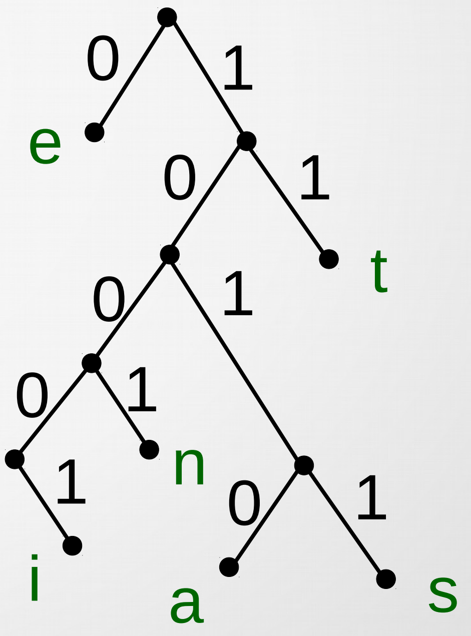
Prefix Codes

Practice:

Given the binary tree with the following prefix codes find the word represented by 10001100110111111010100111

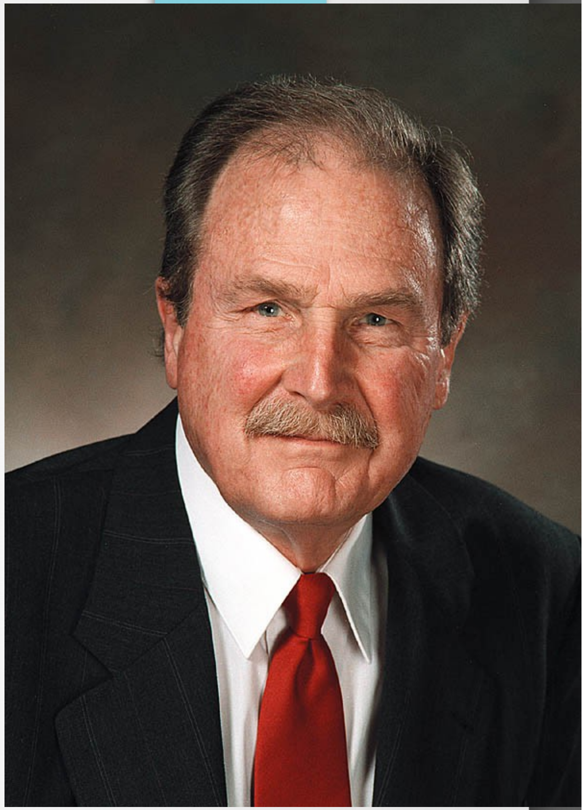
```
10001 1001 1011 11 1010 1001 11
  i     n    s   t   a    n   t
```

instant



11.2 *Applications of Trees*

Huffman Coding



An algorithm, known as *Huffman coding*, was developed by David Albert Huffman in a term paper he wrote in 1951 while a graduate student at MIT.

It allows to produce prefix code for a string using fewest possible bits, based on *frequencies* (which are the probabili-ties of occurrences) of symbols in a string

This algorithm is a fundamental algorithm in data compression (including audio and image compressions).

11.2 *Applications of Trees*

Huffman Coding

```
procedure Huffman(C: symbols $a_i$ with freq. $w_i$, $i = 1,...,n$)
F := forest of $n$ rooted trees, each contains a single vertex
      $a_i$ and assigned weight $w_i$.
while F is not a tree
    Take two rooted trees, T and T', with the least weights
    and $w(T) \geq w(T')$.
    Replace them with a tree having a new root that has T
    as its left subtree and T' as its right subtree.
    Label the edge to T with 0, and the edge to T' with 1.
    Assign $w(T)+w(T')$ as the weight of the new tree.
```

Output: Huffman code for a_i: the concatenation of labels of the edges in the unique path from the root to the vertex v_i.

11.2 *Applications of Trees*

Huffman Coding

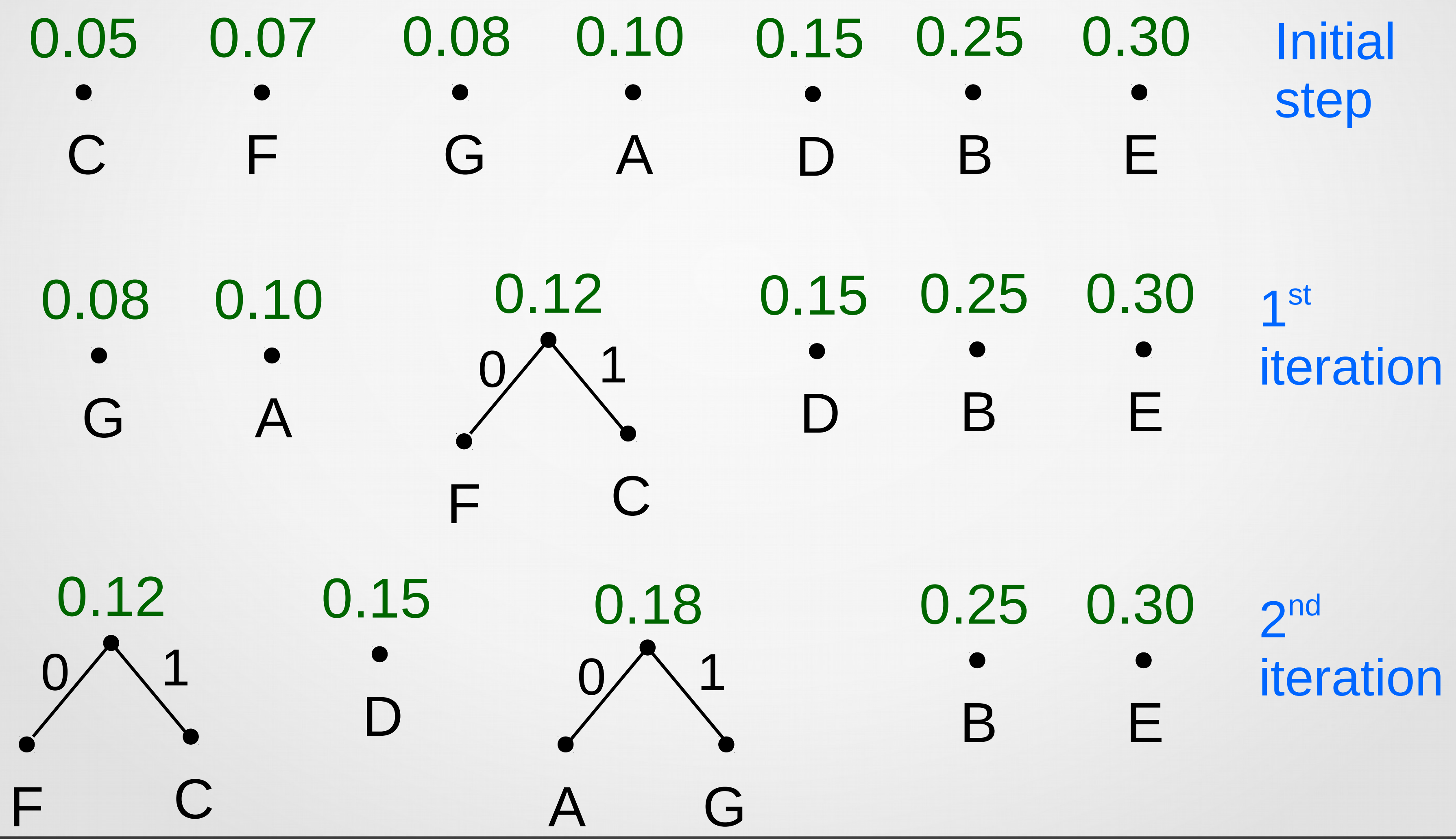
Example: Use Huffman coding to encode the following symbols with the frequencies:

A: 0.10
B: 0.25
C: 0.05
D: 0.15
E: 0.30
F: 0.07
G: 0.08

What is the average number of bits required to encode a symbol?

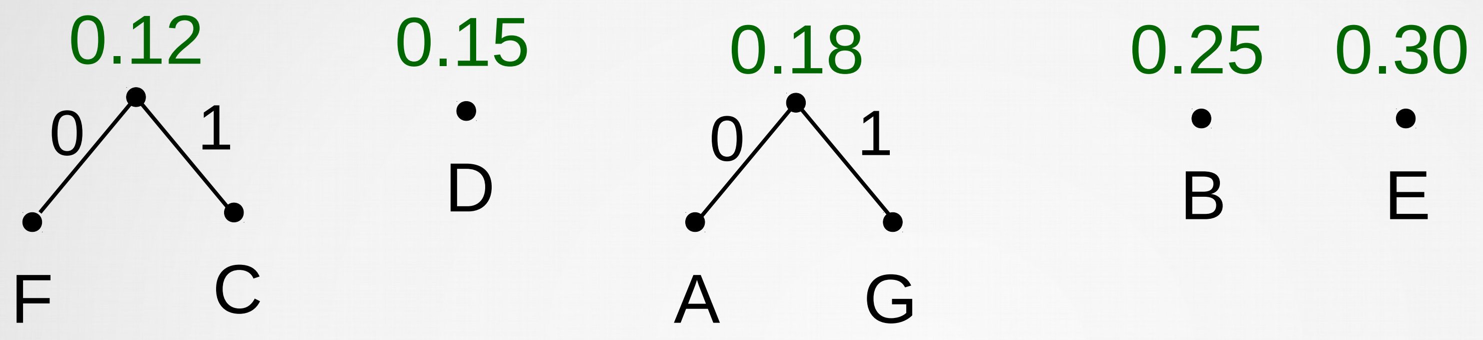
11.2 *Applications of Trees*

Huffman Coding

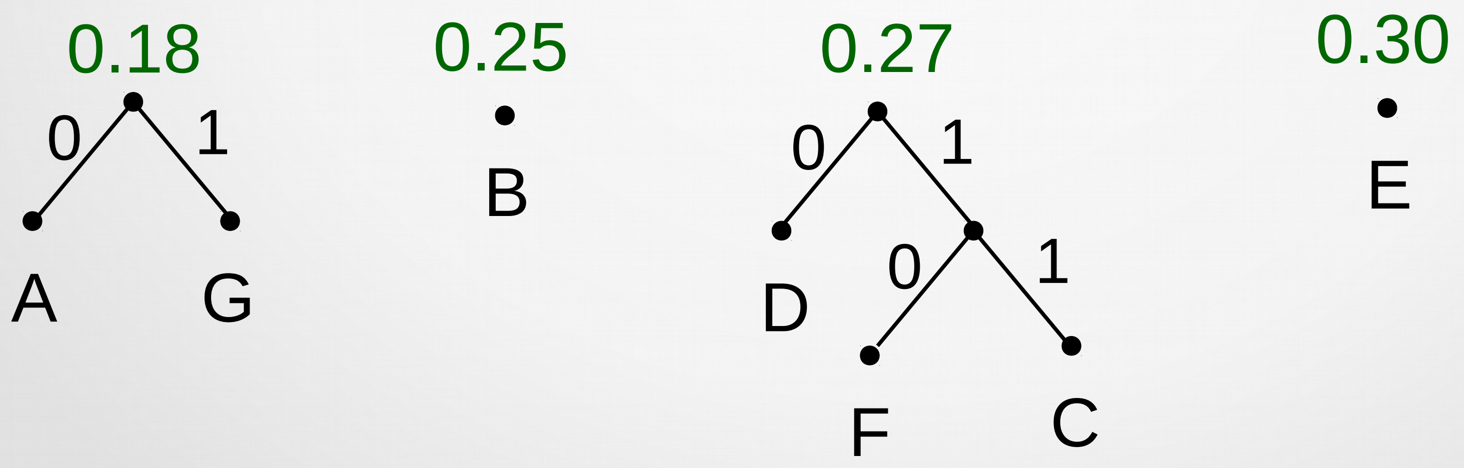


11.2 *Applications of Trees*

Huffman Coding



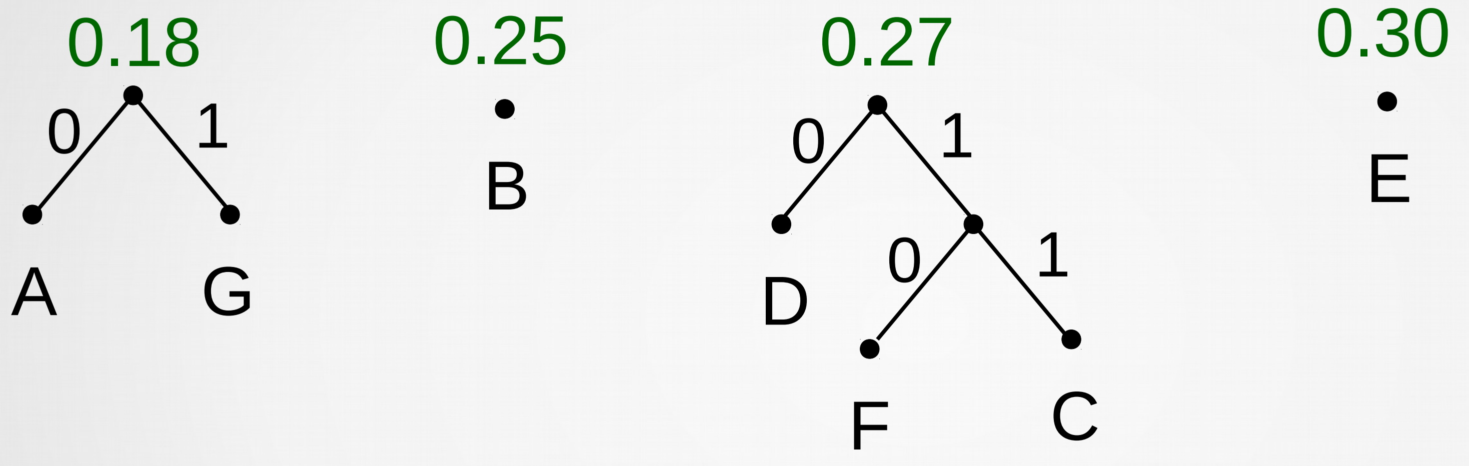
2nd iteration



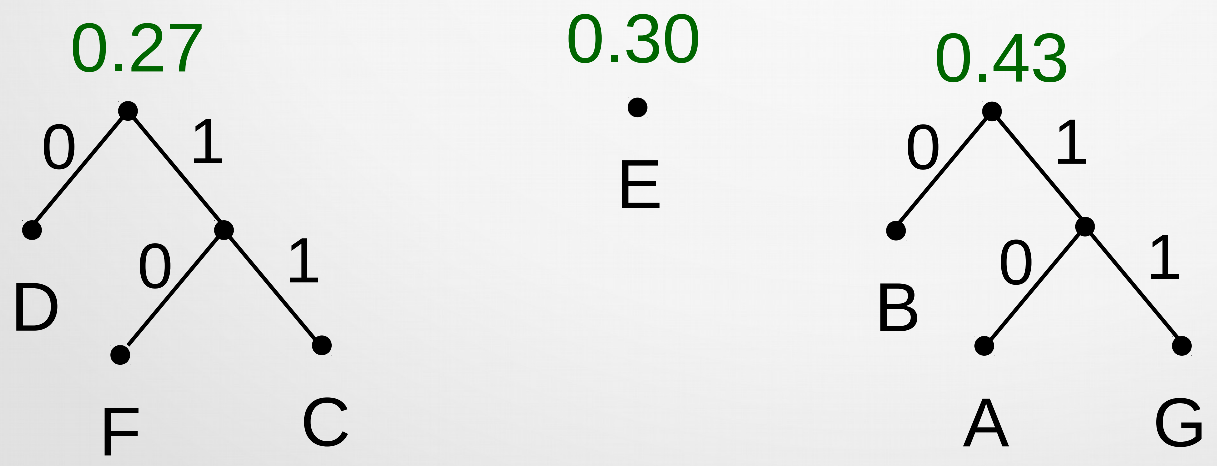
3rd iteration

11.2 *Applications of Trees*

Huffman Coding



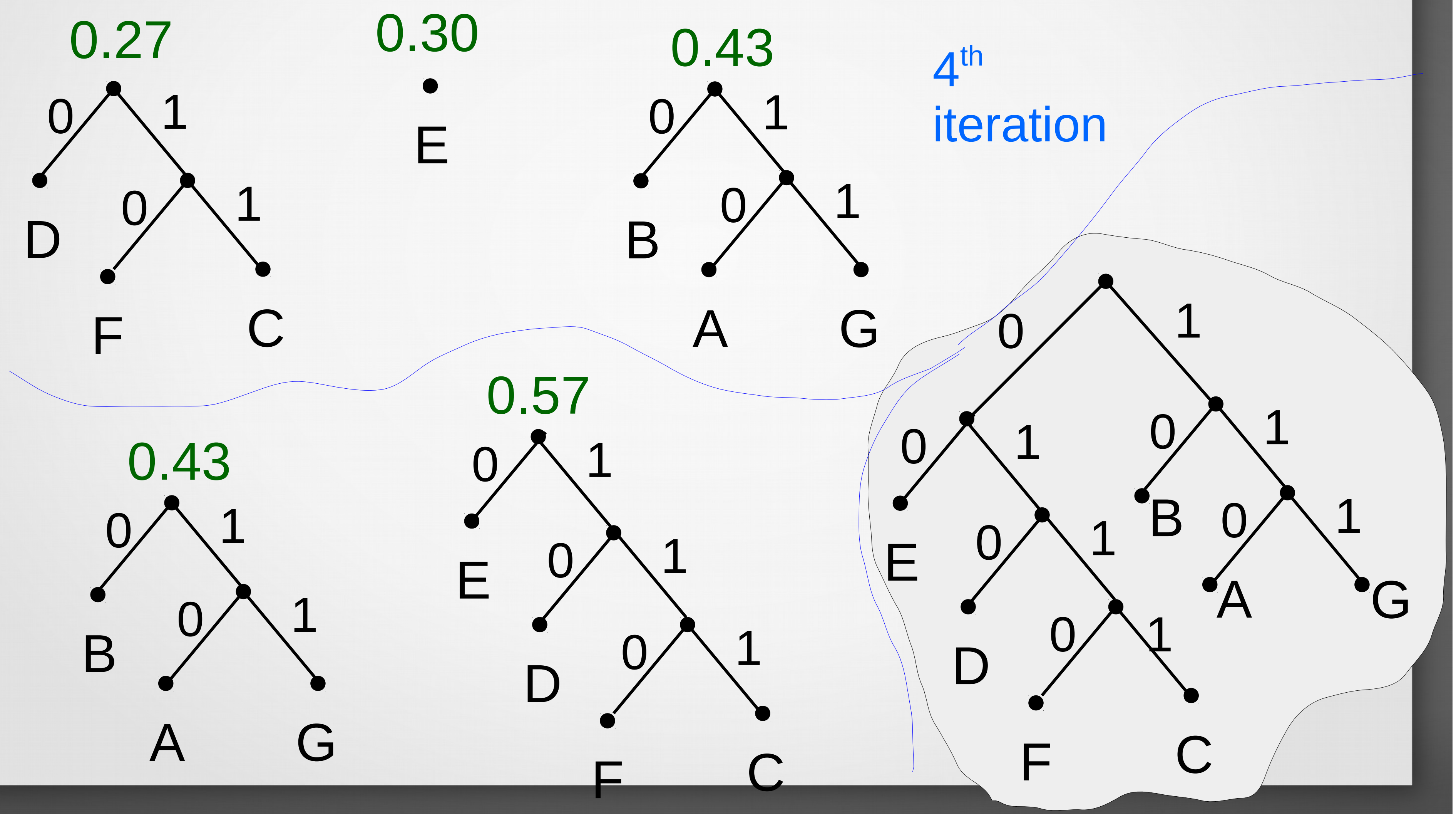
3rd iteration



4th iteration

11.2 *Applications of Trees*

Huffman Coding



4th iteration

5th iteration

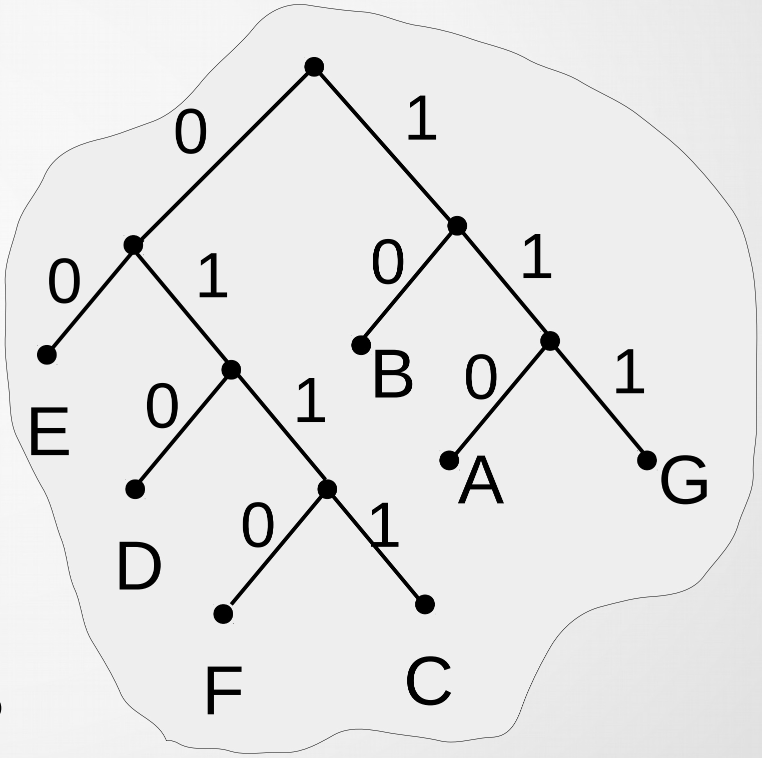
6th iteration

11.2 *Applications of Trees*

Huffman Coding

Example: Use Huffman coding to encode the following symbols with the frequencies:

A: 0.10	110
B: 0.25	10
C: 0.05	0111
D: 0.15	010
E: 0.30	00
F: 0.07	0110
G: 0.08	111



What is the average number of bits required to encode a symbol?

3 * 0.10 + 2 * 0.25 + 4 * 0.05 +
3 * 0.15 + 2 * 0.3 + 4 * 0.07 + 3 * 0.08 = 2.57

11.2 *Applications of Trees*

Huffman Coding

Note on Huffman coding algorithm:

Huffman coding is a *greedy algorithm*.
Replacing two subtrees with the smallest weight at each step leads to an optimal code in the sense that no binary prefix code for these symbols can encode these symbols using fewer bits (needs to be proved)

There are many variations of Huffman coding.