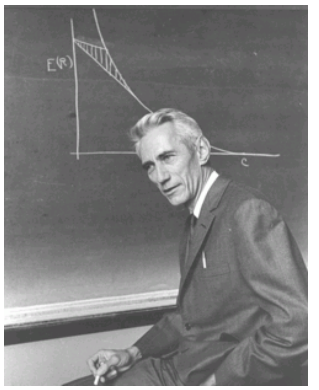
Binary Trees – Case-studies



Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Binary Trees – Case-studies

Standard reminder to set phones to silent/vibrate mode, please!

Binary Trees – Case-studies

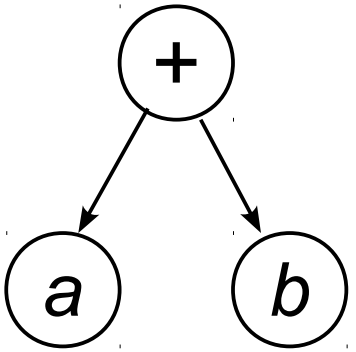
- Today's class:
  - We'll look at some examples / case-studies where trees play an important role as a useful/powerful tool.
    - Binary expression trees and reverse-Polish notation.
    - Huffman Trees (for Optimal prefix code data compression)

Binary Trees – Case-studies

- Binary expression trees:
  - Basic idea: mathematical expressions involve exclusively binary operations (the exception being the unary minus — as in, $-x$, but that can always be expressed as $0 - x$, resorting to the binary minus)
  - So, an expression involving a certain binary operation on two sub-expressions can be represented by a binary tree where the root node represents the operation, and the child nodes represent the operands — possibly sub-trees to represent the sub-expressions.

Binary Trees – Case-studies

- Very simple example:
  - The expression *a* + *b* would be represented as:

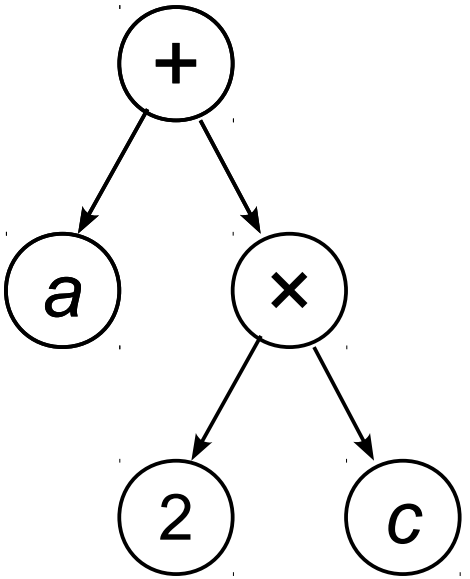


Binary Trees – Case-studies

- If instead of *b* we had another expression, such as 2×*c*, then we make it such that the right child is not a leaf node containing b, but a whole sub-tree representing the expression 2×*c*:

Binary Trees – Case-studies

- If instead of *b* we had another expression, such as 2×*c*, then we make it such that the right child is not a leaf node containing b, but a whole sub-tree representing the expression 2×*c*:



Binary Trees – Case-studies

- Given the recursive nature of these expressions (an expression being a binary operation between two sub-expressions), we can extend the binary tree representation to expressions of arbitrary complexity.

Binary Trees – Case-studies

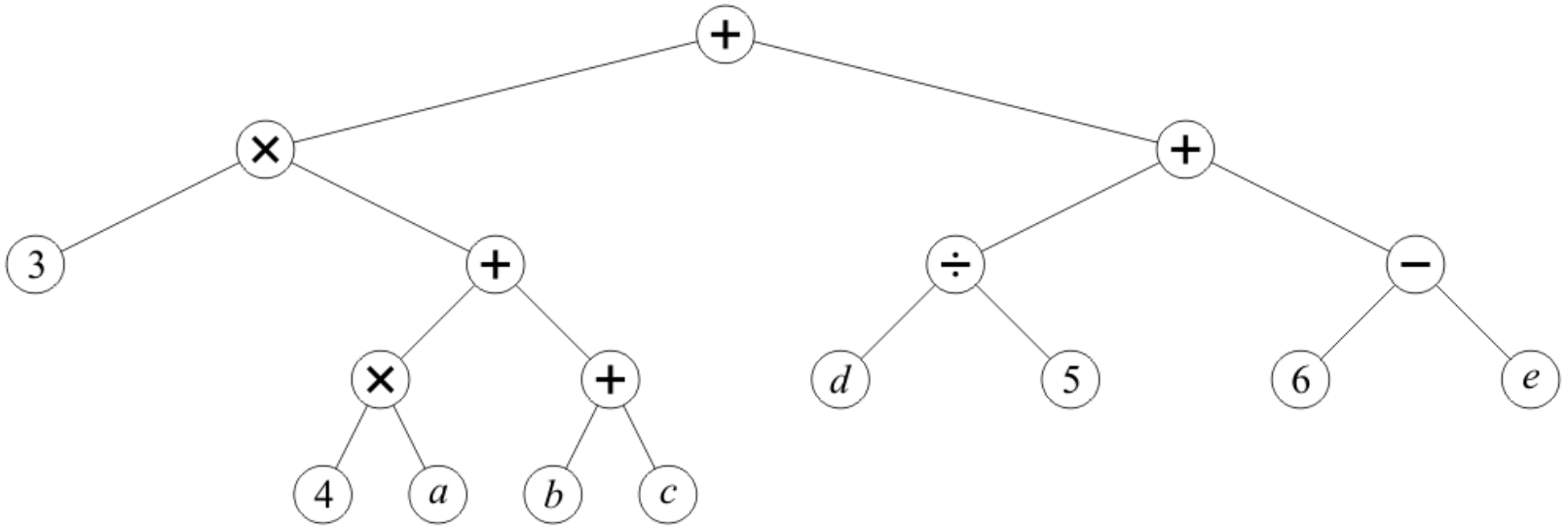
- Given the recursive nature of these expressions (an expression being a binary operation between two sub-expressions), we can extend the binary tree representation to expressions of arbitrary complexity.
  - Internal nodes (always full nodes) store operations.
  - Leaf nodes store literal values or variables.
  - This is an ordered tree! (subtraction and division are not commutative!)

Binary Trees – Case-studies

- Given the recursive nature of these expressions (an expression being a binary operation between two sub-expressions), we can extend the binary tree representation to expressions of arbitrary complexity.
  - Internal nodes (always full nodes) store operations.
  - Leaf nodes store literal values or variables.
  - This is an ordered tree! (subtraction and division are not commutative!)

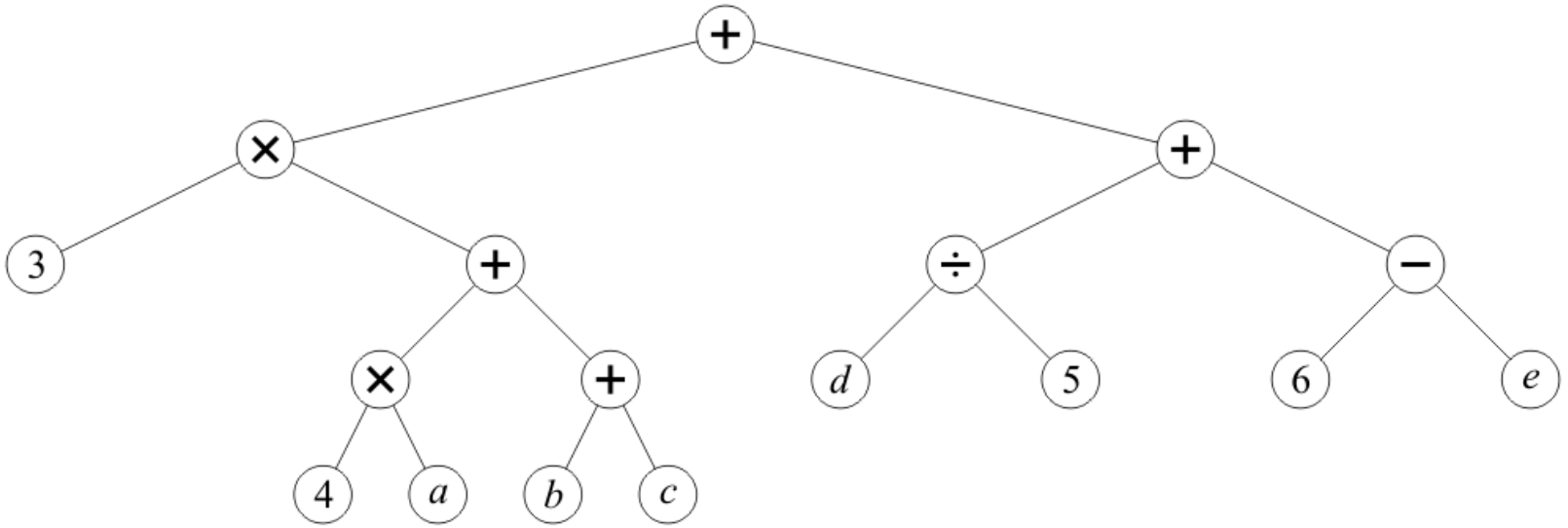
Binary Trees – Case-studies

- Example: 3 (4*a* + *b* + *c*) + *d* / 5 + (6 - *e*)



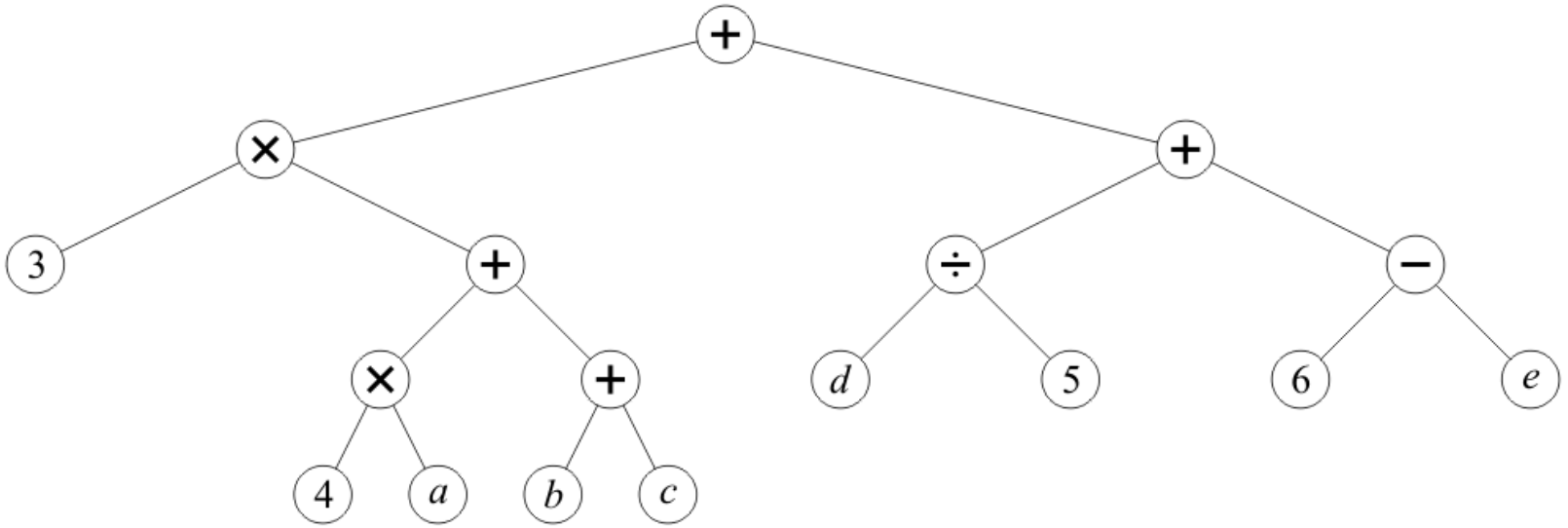
Binary Trees – Case-studies

- BTW ... How do we write (output) the contents of an expression tree? (i.e., print the represented expression given the tree)



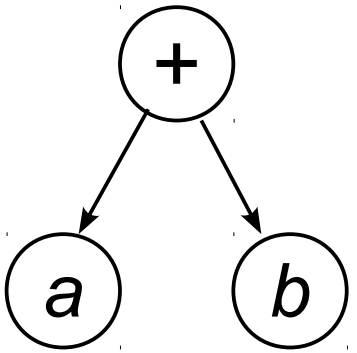
Binary Trees – Case-studies

- Breadth-first? Depth-first? Pre-, post-, or in-order?



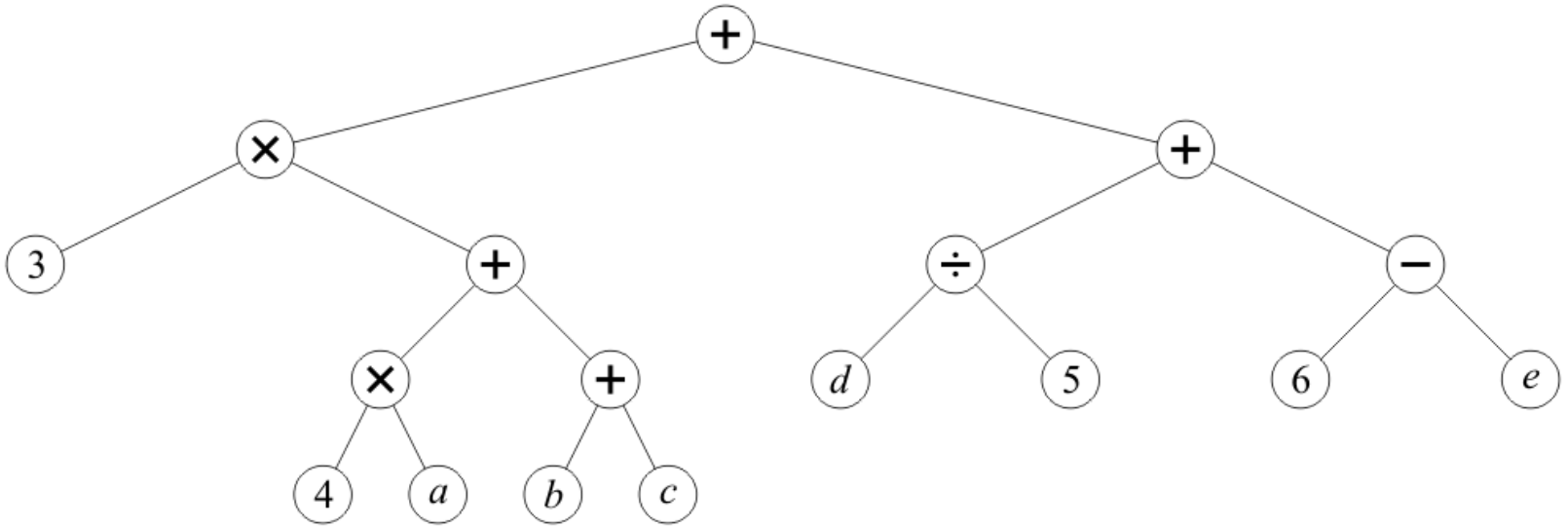
Binary Trees – Case-studies

- For the case of a very simple tree, it is clear that we want in-order traversal — we want operand1, operator, operand2 (in the simple example below, *a*, followed by +, followed by *b*)



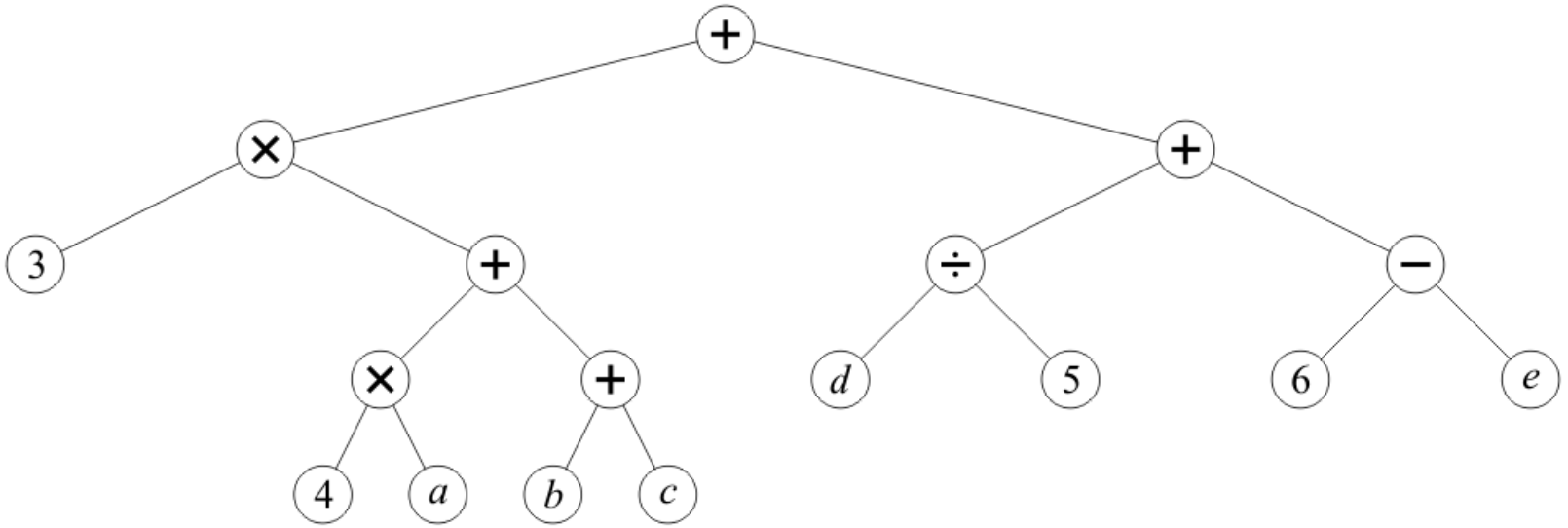
Binary Trees – Case-studies

- Does it work for this one as well?



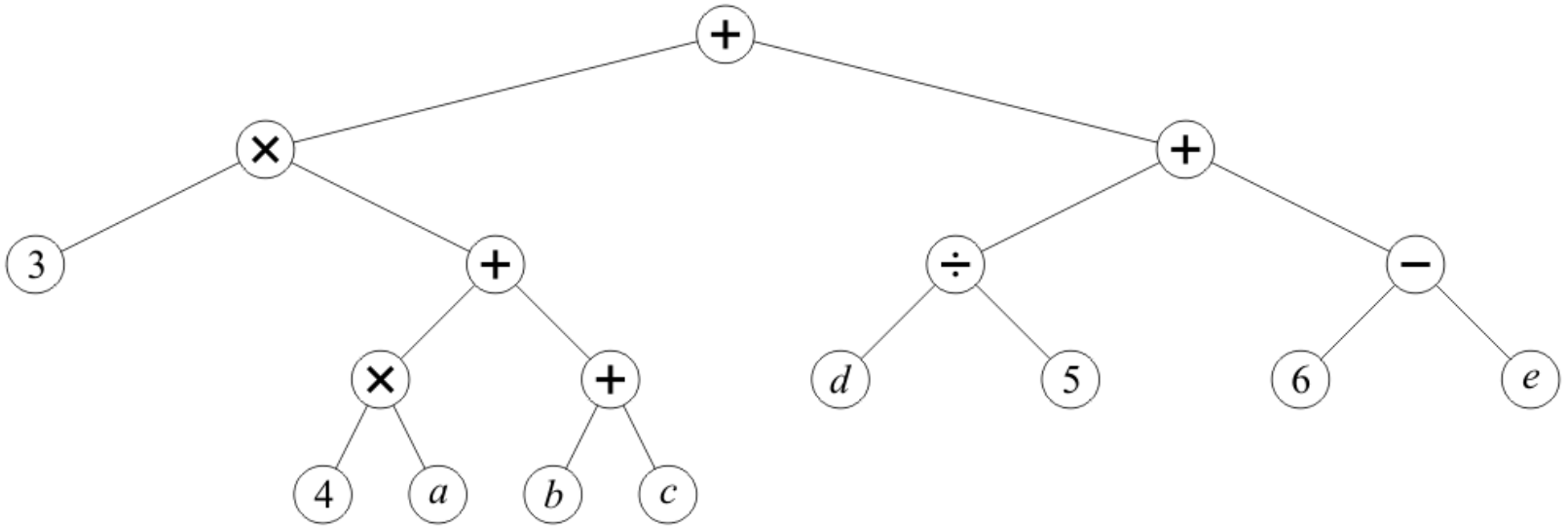
Binary Trees – Case-studies

- Perhaps as interesting: what is the output if we use post-order depth-first traversal?



Binary Trees – Case-studies

- The pattern is, both operands appear first, then the operation — does this remind you of something?



3 4 *a* × *b* *c* + + *d* 5 / 6 *e* − + +

Binary Trees – Case-studies

- **Hint:** Here's how the C++ compiler thinks that the CPU can do *a* + *b*:

```
movl -24(%rbp), %eax
movl -20(%rbp), %edx
addl %edx, %eax
```

  - Again, this is *Intel* assembler, in AT&T-style notation — a little different than what you're used to see in ECE-222, but the principle being the same)

Binary Trees – Case-studies

- The principle being: the CPU language requires that two operands to be loaded first into registers, then the instruction to perform the operation is issued.

Binary Trees – Case-studies

- The principle being: the CPU language requires that two operands to be loaded first into registers, then the instruction to perform the operation is issued.
  - So, it turns out that this notation “in reverse” could be useful after all!
  - Compilers could use this tree to:
    - Do manipulations (possibly simplifications) on the expression.
    - Determine the sequence of assembly-level instructions.

Binary Trees – Case-studies

- BTW, this is known as reverse-Polish notation, co-creation of Edsger Dijkstra (as a tool to optimize memory access by using a stack to perform operations).

(image courtesy of wikipedia.org)

Binary Trees – Case-studies

- Next, we'll look at Huffman Trees for data compression...

Binary Trees – Case-studies

- Basic idea: Suppose we want to encode a sequence of “characters” that can only take one of four possible symbols.
- We need to encode these for transmission over a digital communications channel (or to store them in some digital storage medium)

Binary Trees – Case-studies

- Basic idea: Suppose we want to encode a sequence of "characters" that can only take one of four possible symbols.
- We need to encode these for transmission over a digital communications channel (or to store them in some digital storage medium)
  - How do we proceed?

Binary Trees – Case-studies

- The most straightforward approach is to assign two-bit codes to each symbol (with two bits, we have four possible combinations, so we use one combination for each symbol).
- For example, if the symbols are A,B,C,D, we could simply say $A = 00$, $B = 01$, $C = 10$, $D = 11$
  - The transmitter and receiver agree on this encoding, and communication will be successful.

Binary Trees – Case-studies

- The sequence *ABAACAAD* would be encoded as: 0001000010000011
  - The receiver can decode the stream of bits because it knows that every two bits correspond to a character.

Binary Trees – Case-studies

- Of course, we'd like to optimize transmission speed, so we should minimize the amount of bits used, right?
  - However, it seems like we have no choice, since there are four possible symbols, so we need two bits to represent each.

Binary Trees – Case-studies

- How about this twist: what if we knew that the symbol *A* occurs 80% of the time in the sequences being transmitted, B occurs 10% of the time, and C, D occur 5% of the time each?
- Could we come up with a different encoding that would reduce the amount of bits to transmit?

Binary Trees – Case-studies

- How about this twist: what if we knew that the symbol *A* occurs 80% of the time in the sequences being transmitted, B occurs 10% of the time, and C, D occur 5% of the time each?
- Could we come up with a different encoding that would reduce the amount of bits to transmit?
  - Ok, let me give you a hint: do all symbols have to encode to a fixed number of bits? Could we use variable number of bits?

Binary Trees – Case-studies

- The trick is: if we're going to use different amount of bits for different symbols, then we minimize the total number of bits by assigning fewer bits to the symbols that occur more frequently.

  We can easily see why this is the case — if N_k is the number of times that symbol S_k appears, and $|S_k|$ is the size (the number of bits) of symbol S_k, then the total number of bits is:

$$N = N_1|S_1| + N_2|S_2| + N_3|S_3| + N_4|S_4|$$

Binary Trees – Case-studies

- One problem is: if we have variable number of bits, how do we know when a character starts and ends? What if one symbol is encoded as 00, another as 11, and another as 0011??

Binary Trees – Case-studies

- One problem is: if we have variable number of bits, how do we know when a character starts and ends? What if one symbol is encoded as 00, another as 11, and another as 0011??
- How does the receiver know whether the latter is the symbol 00 followed by symbol 11, or if it corresponds to the symbol 0011?

Binary Trees – Case-studies

- The solution is prefix codes — an encoding scheme where no code can be a prefix of another code (for example, if some symbol is encoded as 01, then no other code would start with 01).

Binary Trees – Case-studies

- With this idea in mind, we see that a better encoding, if we know that A occurs 80% of the time, B 10% of the time, and C, D 5% of the time each, is:

$$A = 0,\ B = 10,\ C = 110,\ D = 111$$

Binary Trees – Case-studies

- Let's try encoding the following:

  AAAABAAACAAAADAAABAA

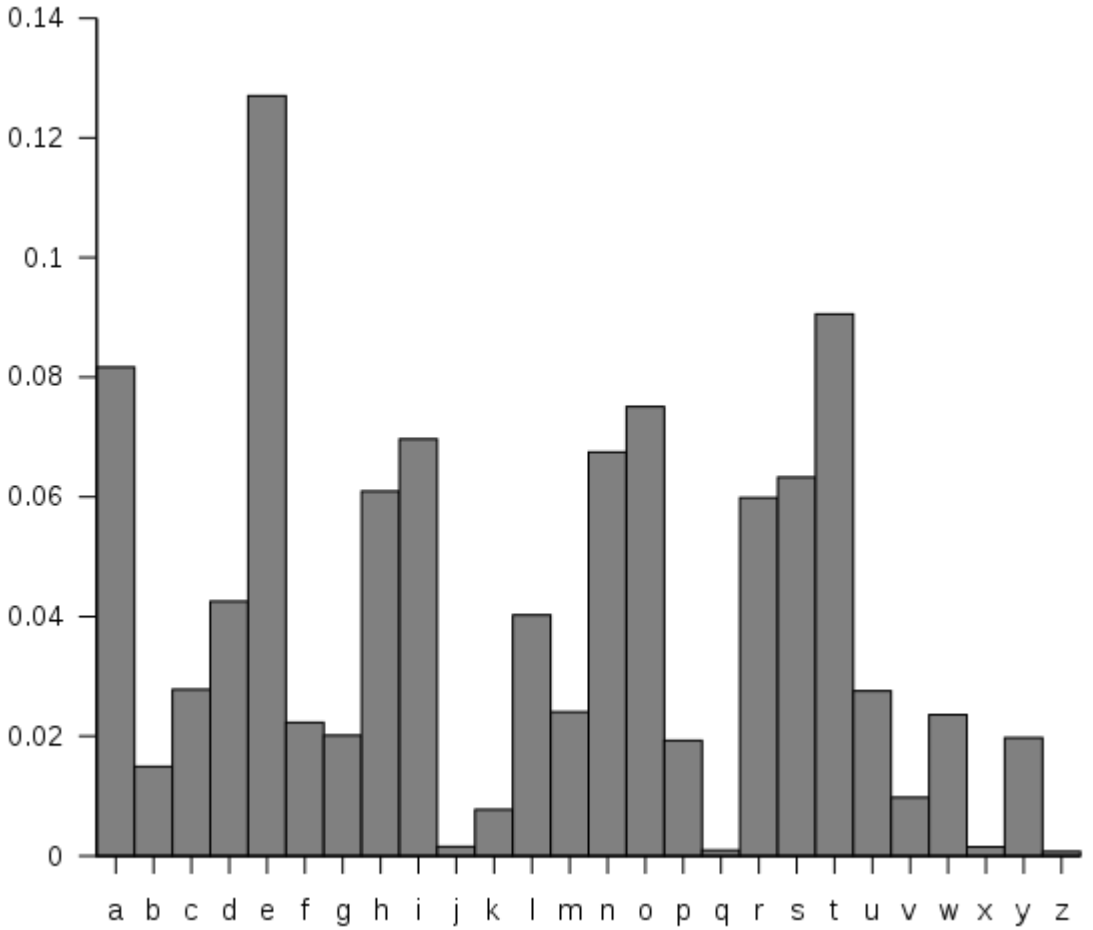
- It's 20 characters, so with our straightforward encoding, it would take 20×2 = 40 bits.
- With A = 0, B = 10, C = 110, D = 111, it encodes to:

  00001000011000001110001000 (only 26 bits!)

Binary Trees – Case-studies

- Now, you may be asking (and granted, it is a very fair question!) «*what on earth can this possibly have to do with reality?*»
- Well, have you seen English text lately? Do all letters occur with the same frequency?
  - Putting aside capitalization and punctuation, we have 26 letters; that would require 5 bits per letter.
  - It turns out that English text can be encoded with approximately ONE bit per letter!!

Binary Trees – Case-studies

- 

(image courtesy of wikipedia.org)

Binary Trees – Case-studies

- This is a very simplified version of the main idea behind *Information Theory*, one of the most ground breaking mathematical theories developed in recent centuries — and we all co-existed with its creator, Claude E. Shannon (1916–2001)!

Binary Trees – Case-studies

- This is a very simplified version of the main idea behind *Information Theory*, one of the most ground breaking mathematical theories developed in recent centuries — and we all co-existed with its creator, Claude E. Shannon (1916–2001)!

  (BTW, we also co-existed with Edsger Dijkstra — another genius from the 20th century; he died in 2002)

Binary Trees – Case-studies

- Incidentally, Claude Shannon, having created this Information Theory and given it a solid mathematical foundation, he then applied it to Data compression, and he got it wrong!!

Binary Trees – Case-studies

- Incidentally, Claude Shannon, having created this Information Theory and given it a solid mathematical foundation, he then applied it to Data compression, and he got it wrong!!
  - Not horribly wrong, though — his method, the Shannon-Fano compression scheme, is reasonably efficient.... It's just not *optimal*.

Binary Trees – Case-studies

- Incidentally, Claude Shannon, having created this Information Theory and given it a solid mathematical foundation, he then applied it to Data compression, and he got it wrong!!
  - Not horribly wrong, though — his method, the Shannon-Fano compression scheme, is reasonably efficient.... It's just not *optimal*.
  - A few years later, David Huffman realized a key detail in the mechanism, and came up with an optimal compression scheme for prefix codes!

Binary Trees – Case-studies

- Incidentally, Claude Shannon, having created this Information Theory and given it a solid mathematical foundation, he then applied it to Data compression, and he got it wrong!!
  - Not horribly wrong, though — his method, the Shannon-Fano compression scheme, is reasonably efficient.... It's just not *optimal*.
  - A few years later, David Huffman realized a key detail in the mechanism, and came up with an optimal compression scheme for prefix codes!

Binary Trees – Case-studies

- Huffman encoding is based on building the so-called *Huffman Encoding Tree*, or just Huffman Tree.
  - It is a tree that determines the optimal encoding for a set of symbols with given probabilities of occurrence.
  - It also allows for efficient decoding of a stream of bits — when receiving a 0, we take the left child, with a 1, we take the right child, and when we reach a leaf node, we know that a character is complete (and the leaf node stores the corresponding symbol)

Binary Trees – Case-studies

- Let's find the optimal encoding for the set of eight symbols A,B,C,D,E,F,G,H, where they occur with probabilities 5%, 10%, 7%, 35%, 2%, 17%, 20%, 4%, respectively.

Binary Trees – Case-studies

- The idea is quite simple — each symbol is a node storing the symbol and its probability (initially all disconnected).

Binary Trees – Case-studies

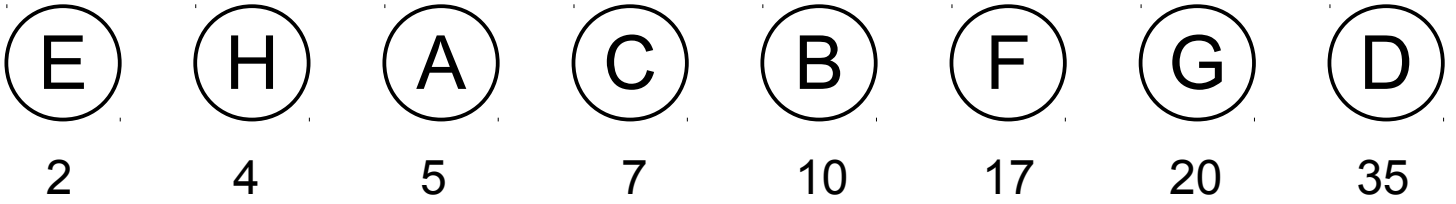
- The idea is quite simple — each symbol is a node storing the symbol and its probability (initially all disconnected).
- We pick the two nodes with lowest probability, and create a new node as the parent of those two nodes. That node is assigned a probability given by the sum of the two children's probability.

Binary Trees – Case-studies

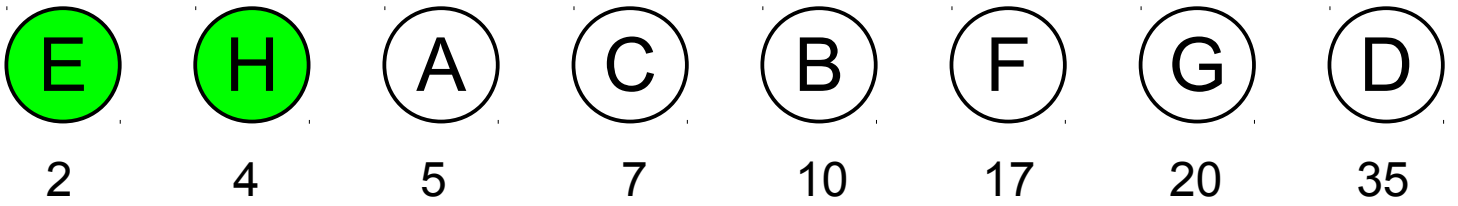
- The idea is quite simple — each symbol is a node storing the symbol and its probability (initially all disconnected).
- We pick the two nodes with lowest probability, and create a new node as the parent of those two nodes. That node is assigned a probability given by the sum of the two children's probability.
- We simply repeat that until we have tree (that is, until we have a single root node)

Binary Trees – Case-studies

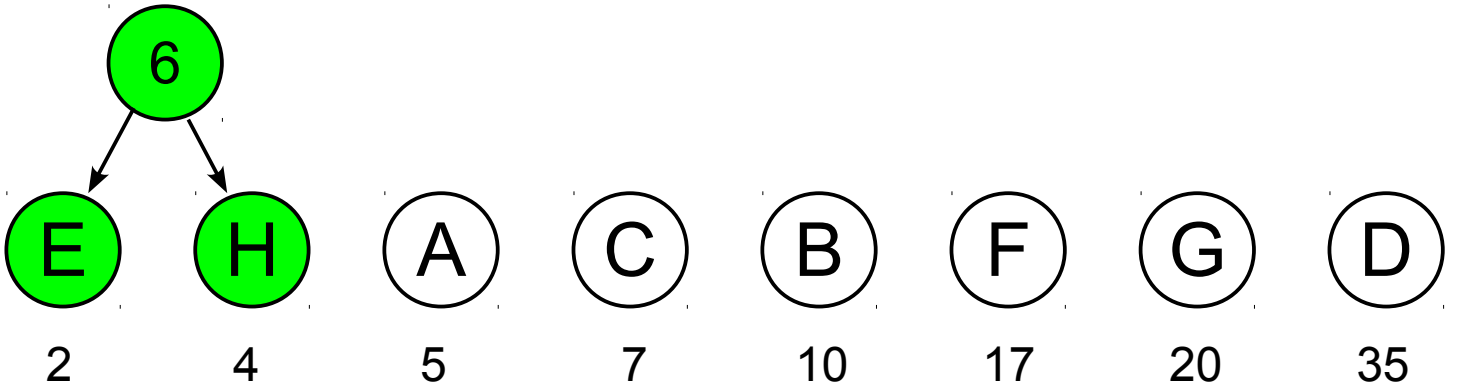
- Example: A,B,C,D,E,F,G,H, where they occur with probabilities 5%, 10%, 7%, 35%, 2%, 17%, 20%, 4%



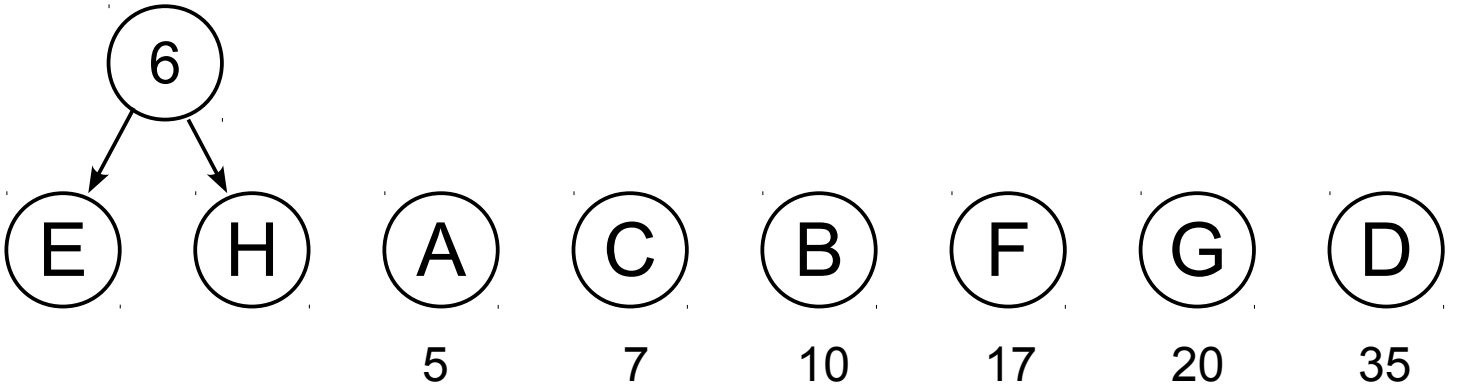
Binary Trees – Case-studies



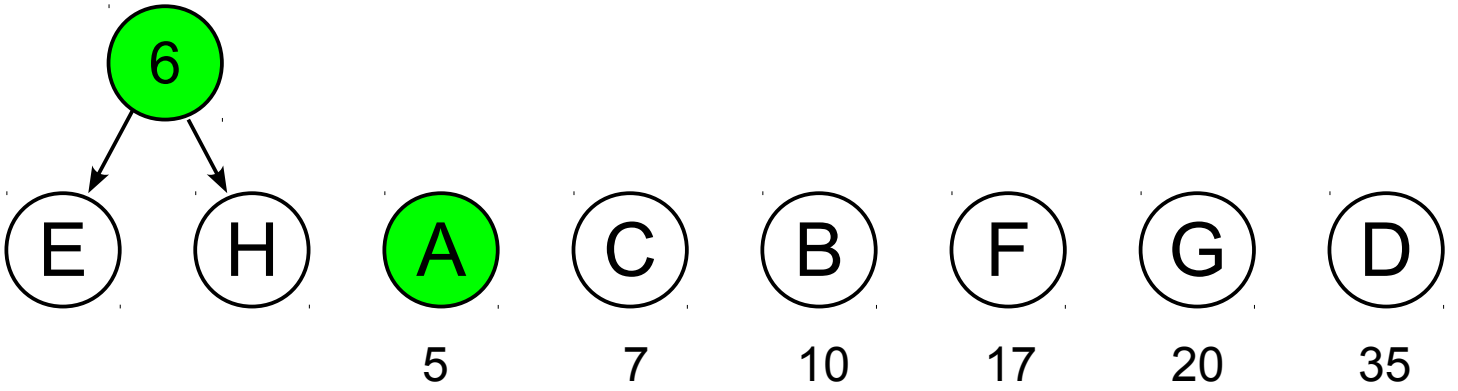
Binary Trees – Case-studies



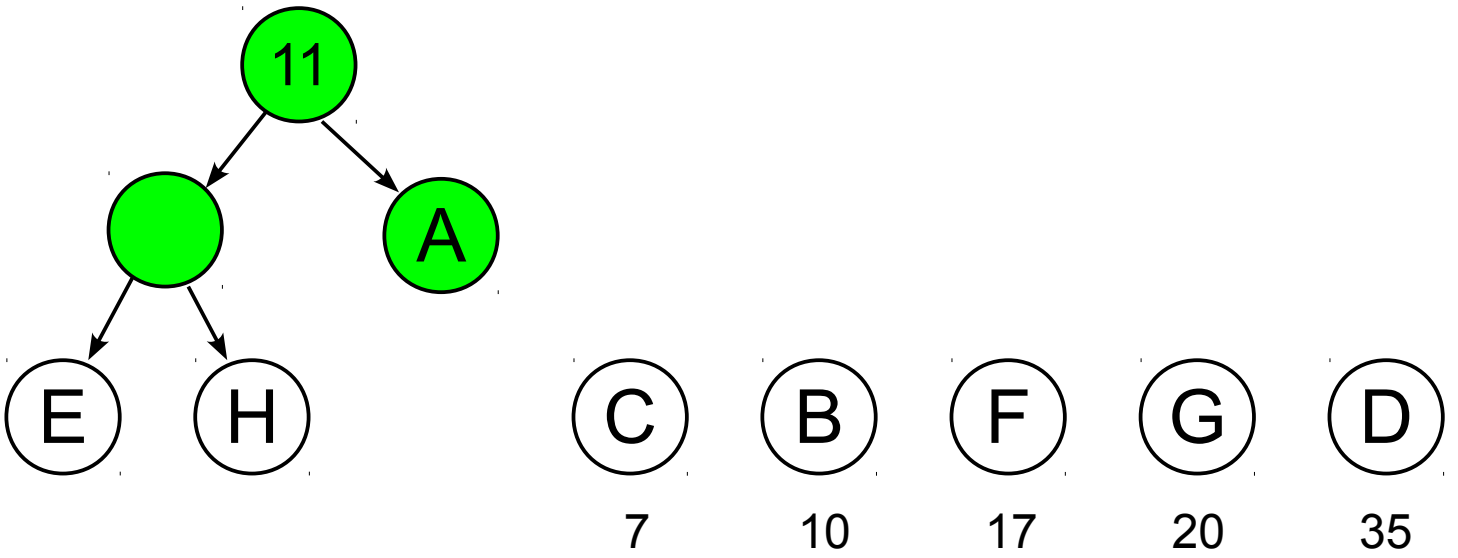
Binary Trees – Case-studies



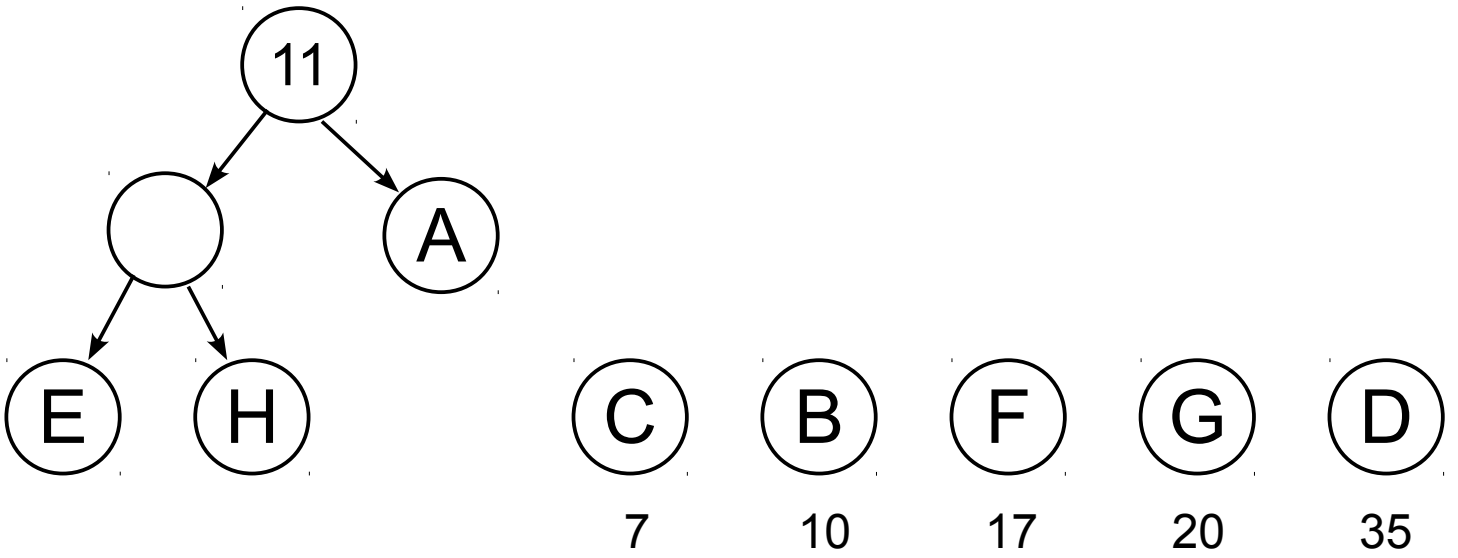
Binary Trees – Case-studies



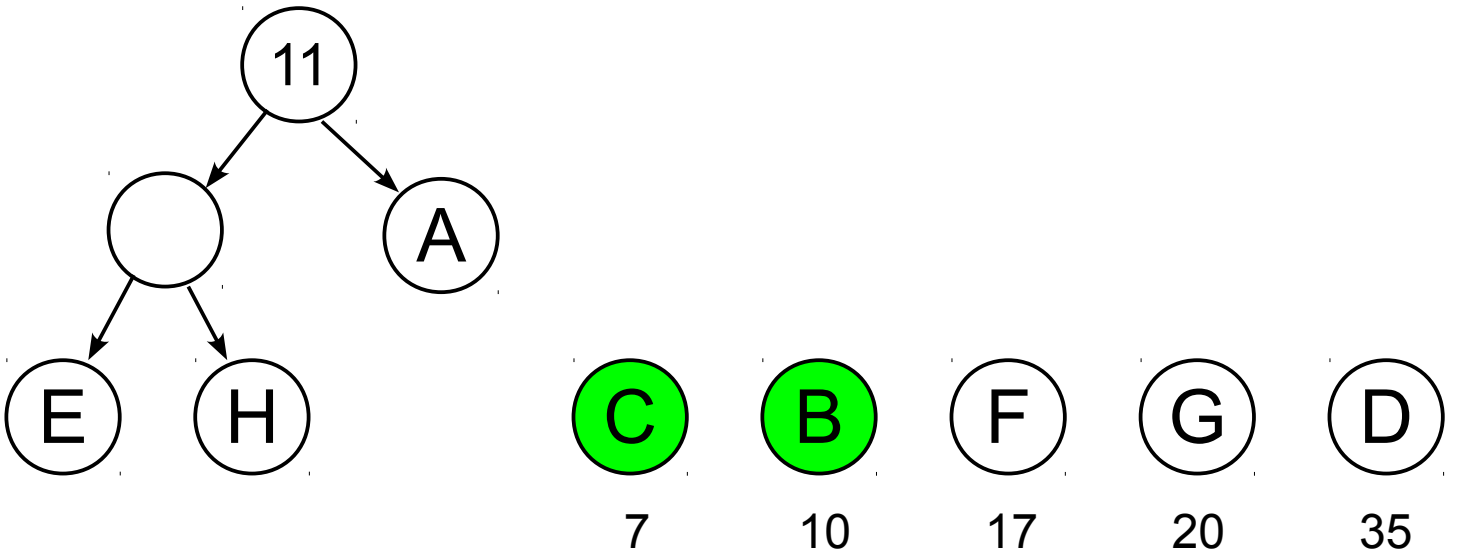
Binary Trees – Case-studies



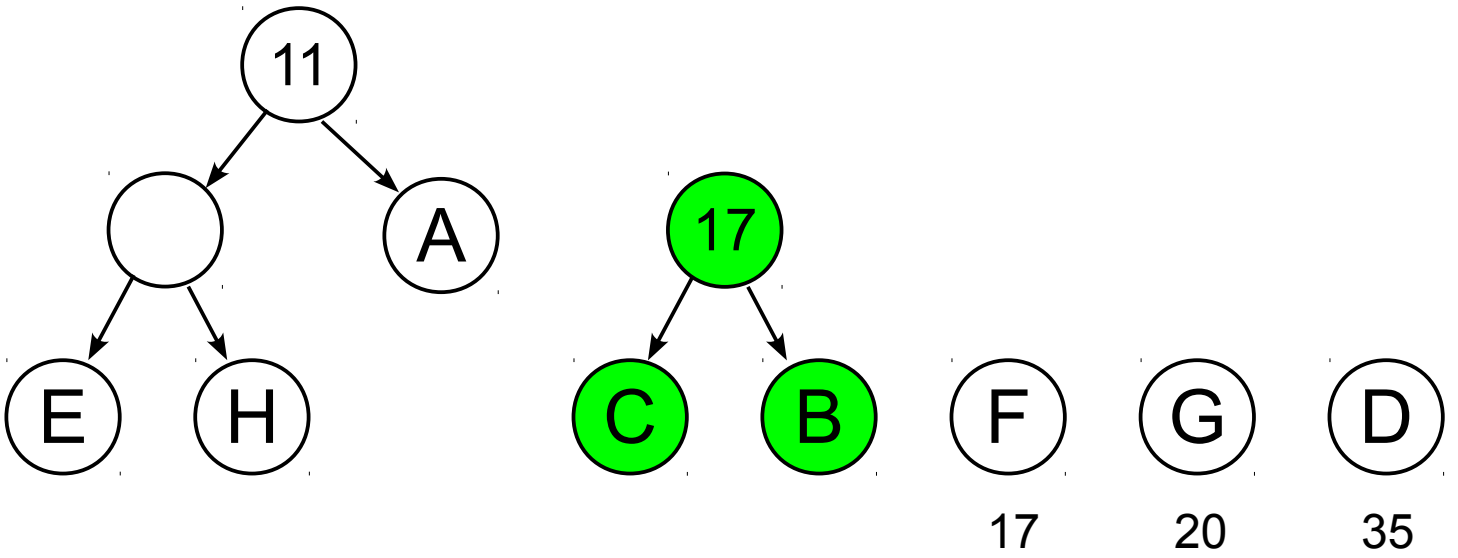
Binary Trees – Case-studies



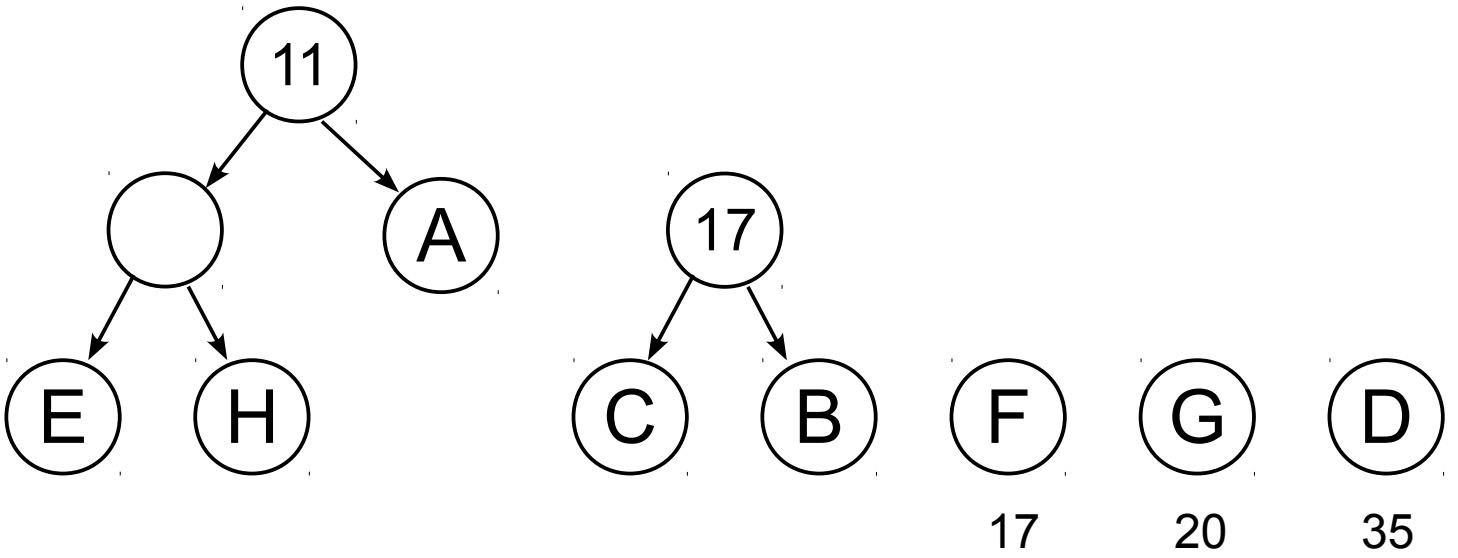
Binary Trees – Case-studies



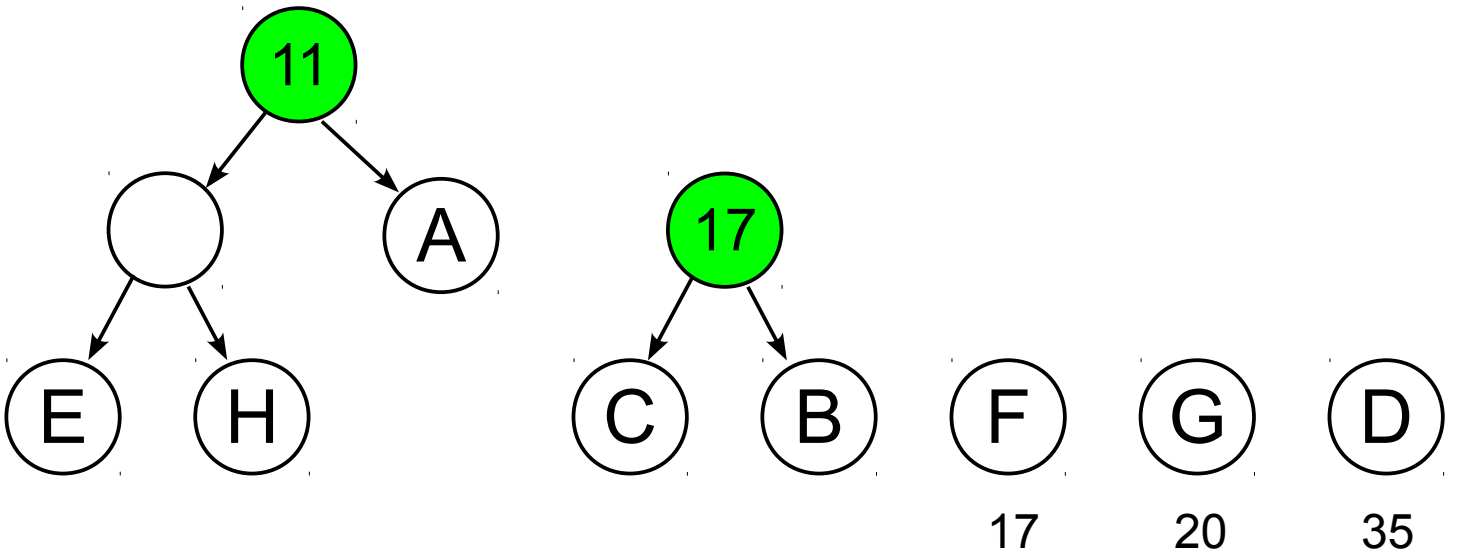
Binary Trees – Case-studies



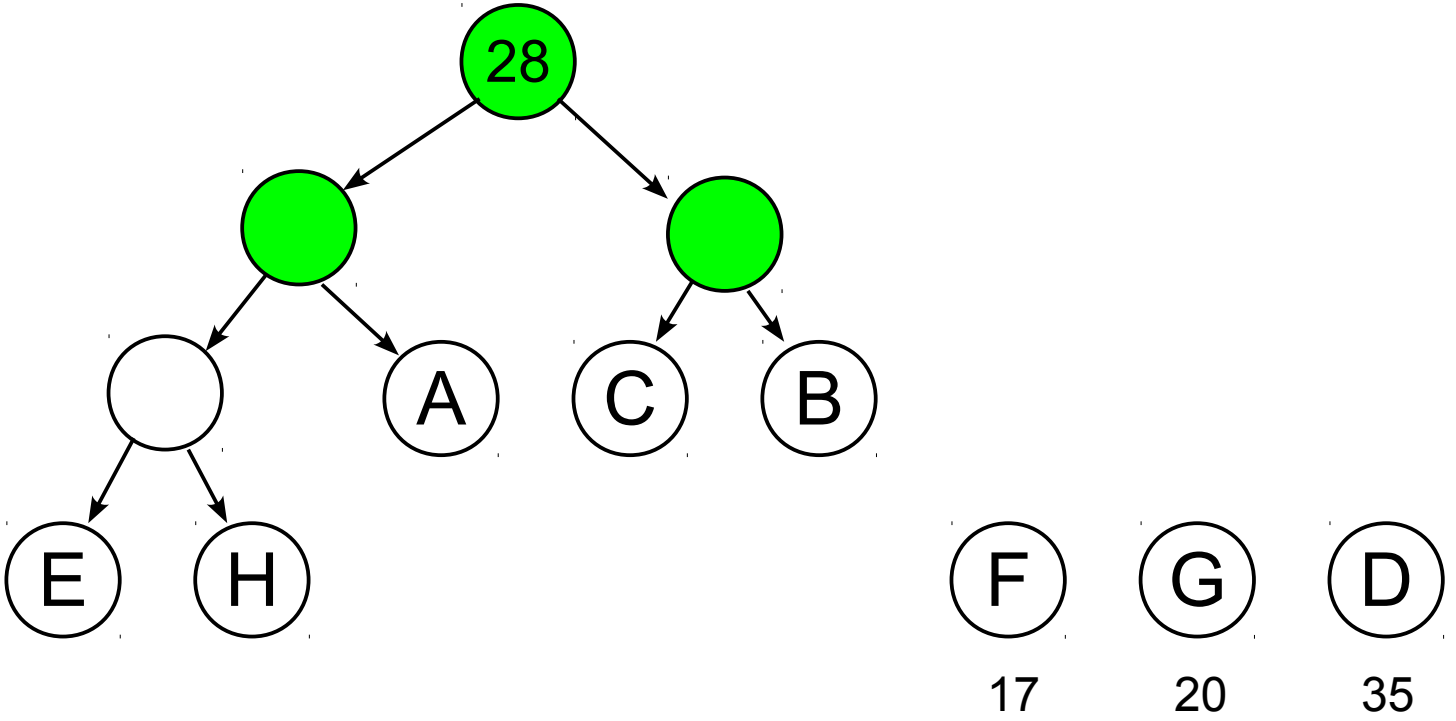
Binary Trees – Case-studies



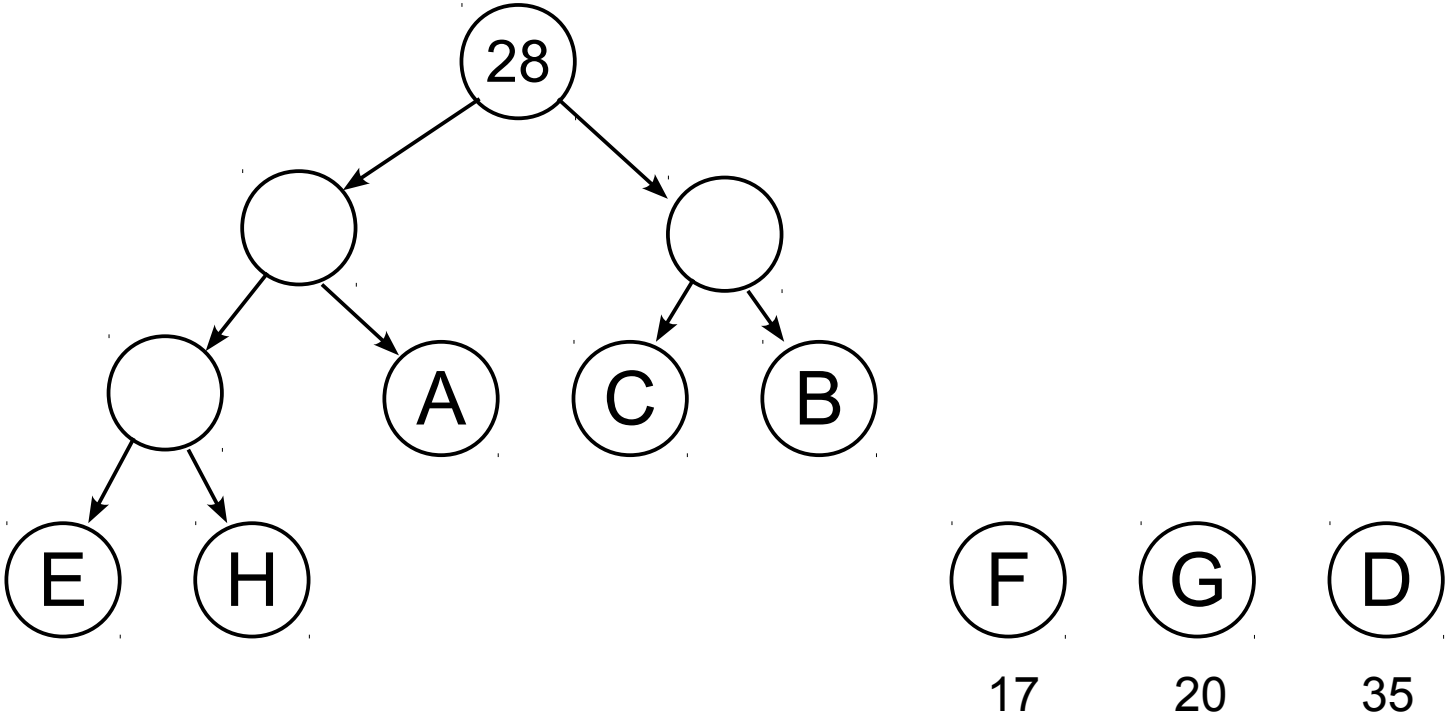
Binary Trees – Case-studies



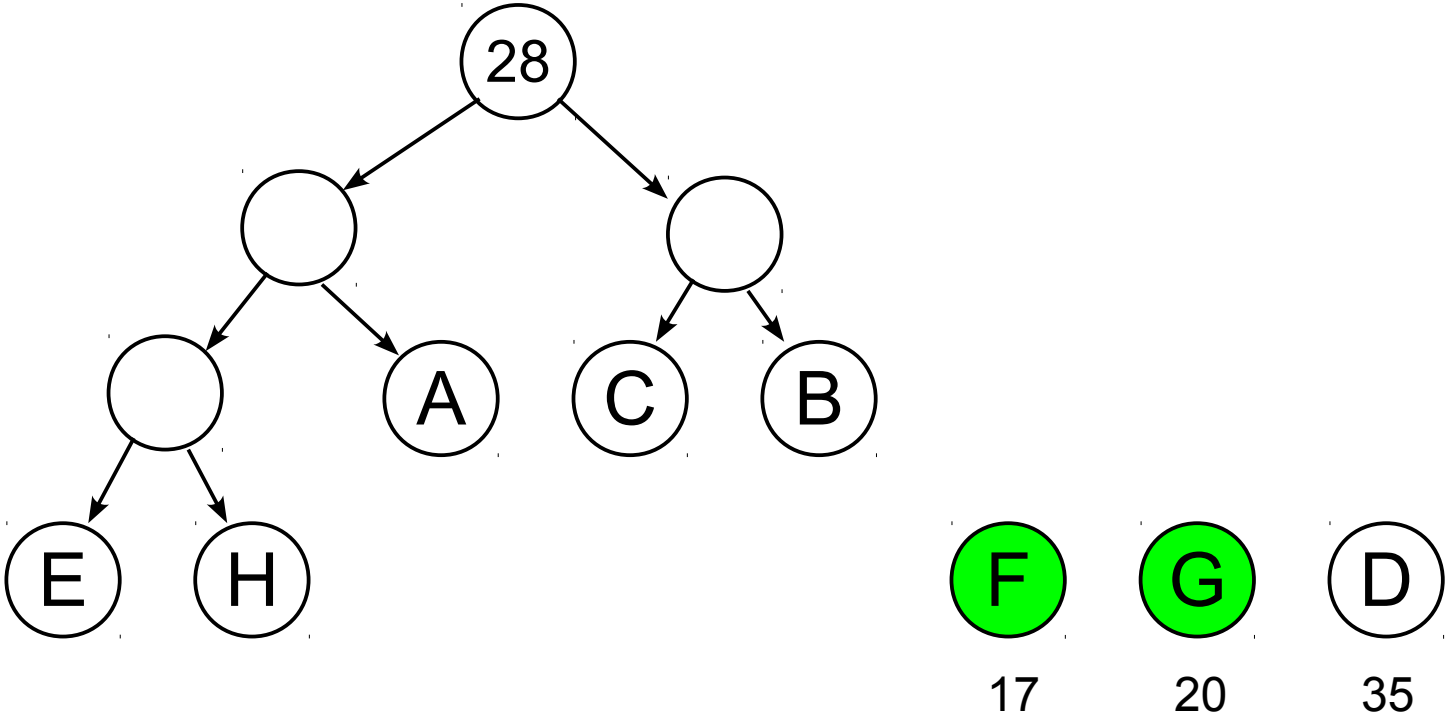
Binary Trees – Case-studies



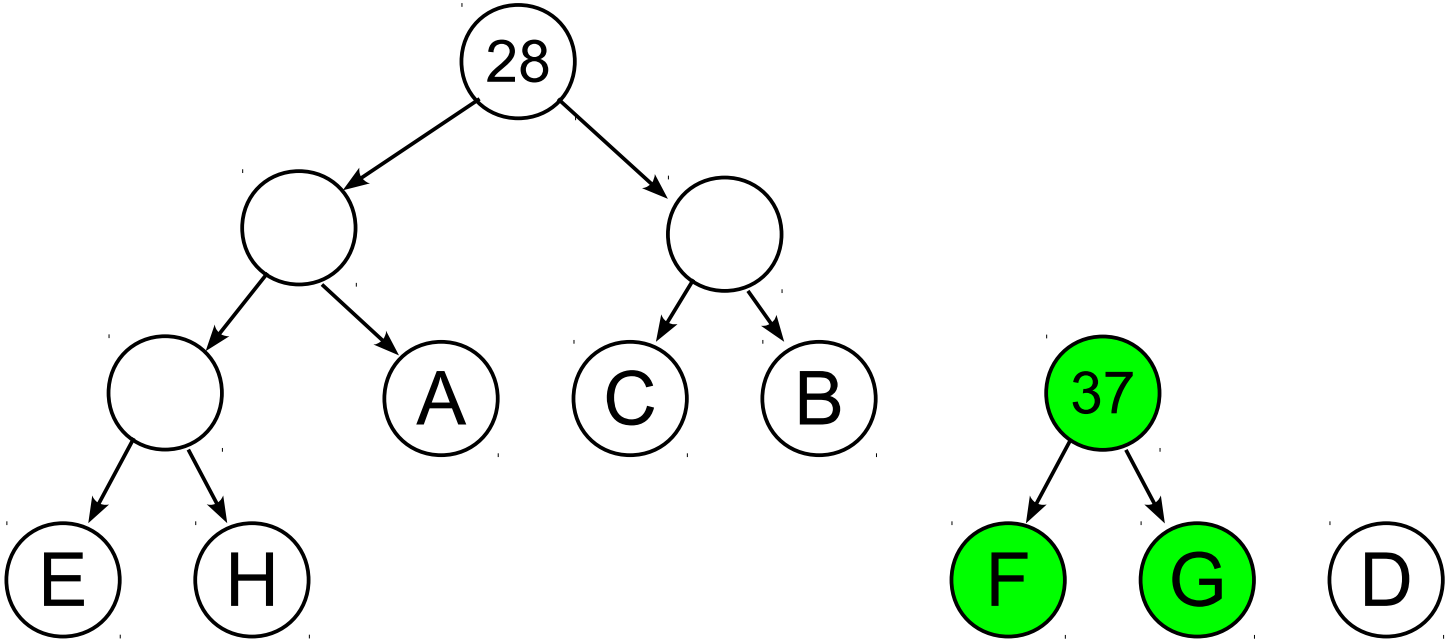
Binary Trees – Case-studies



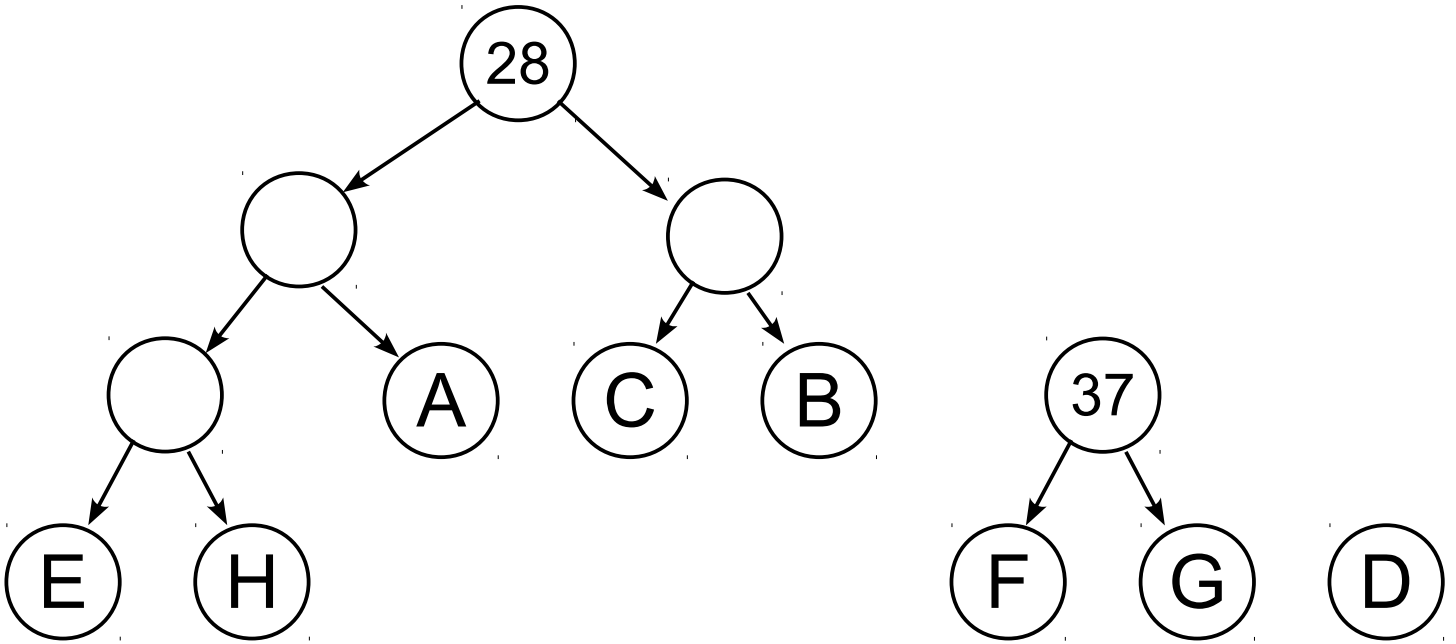
Binary Trees – Case-studies



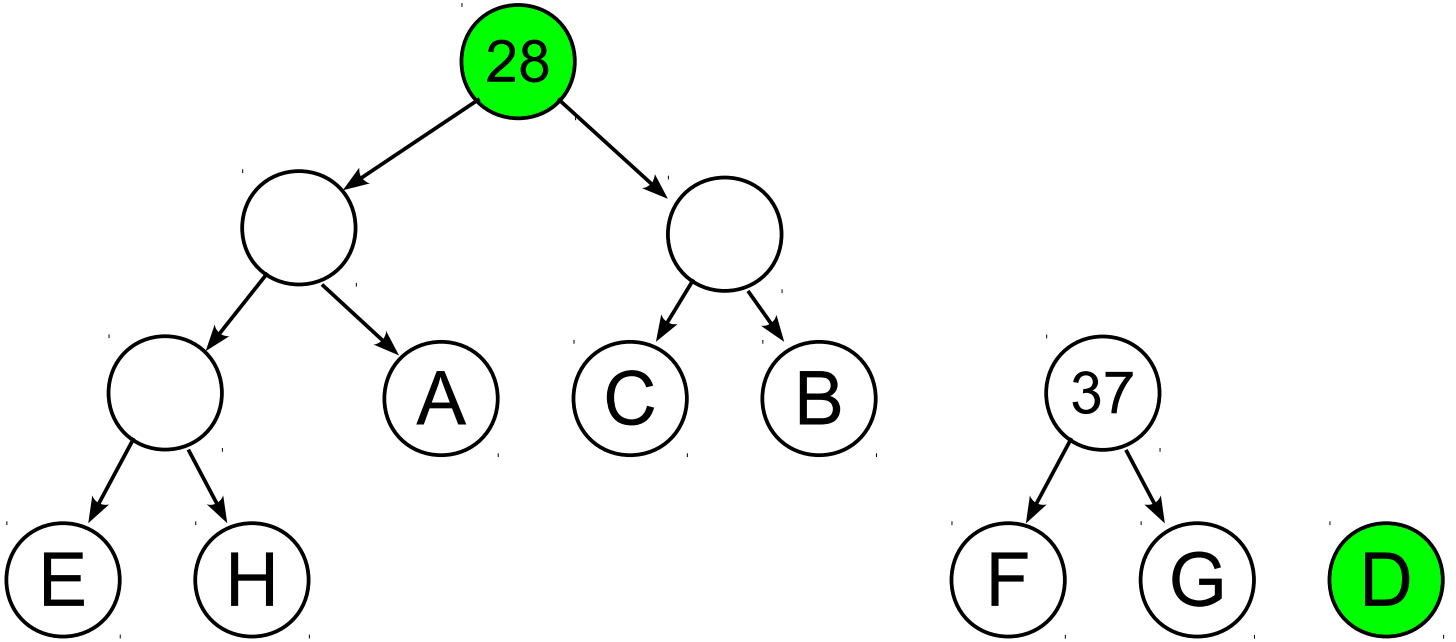
Binary Trees – Case-studies



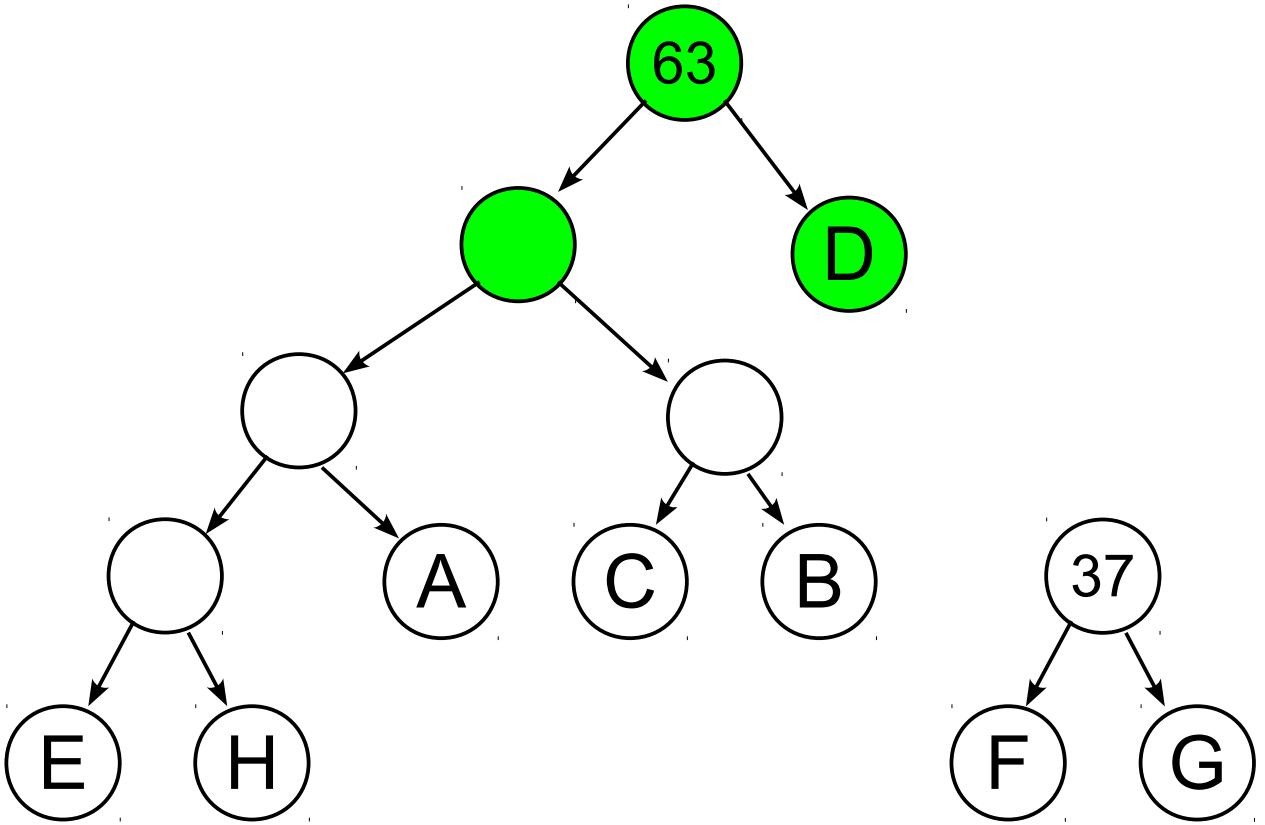
Binary Trees – Case-studies



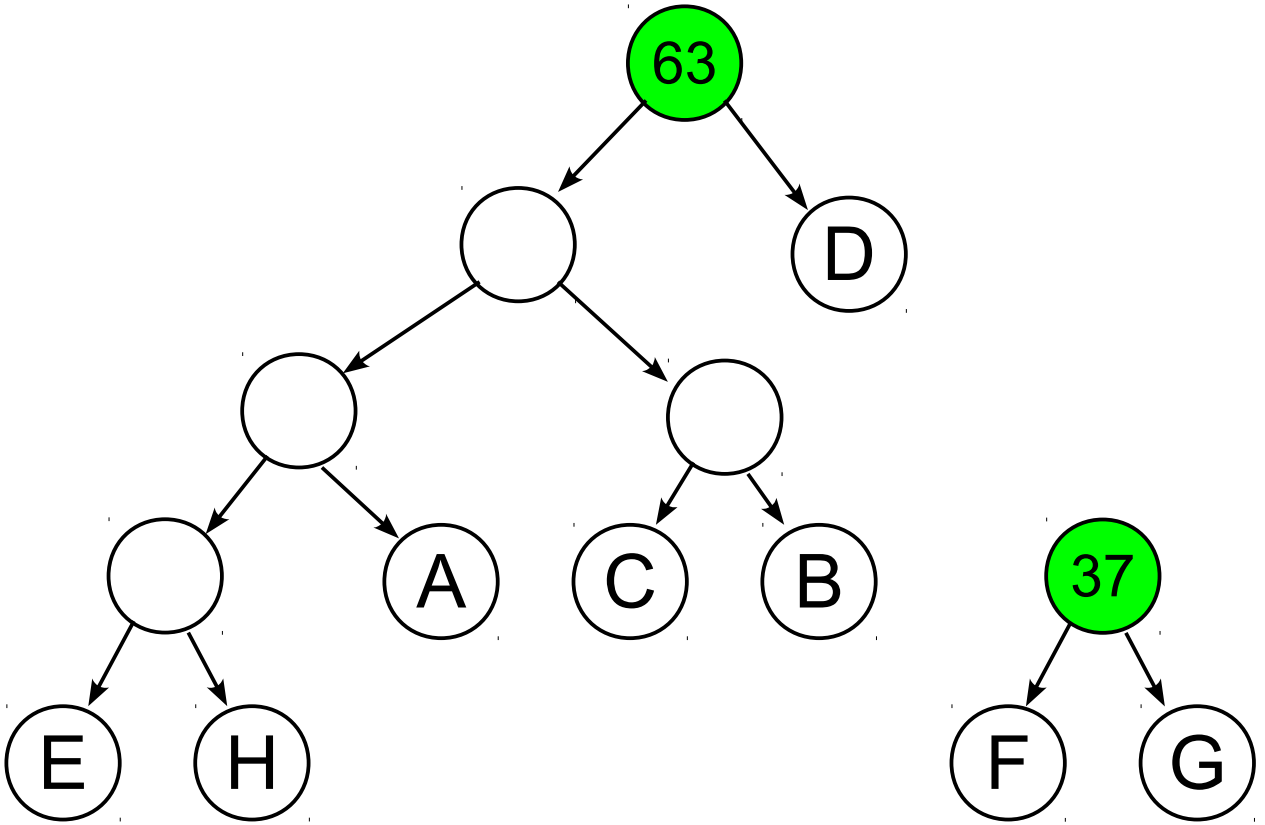
Binary Trees – Case-studies



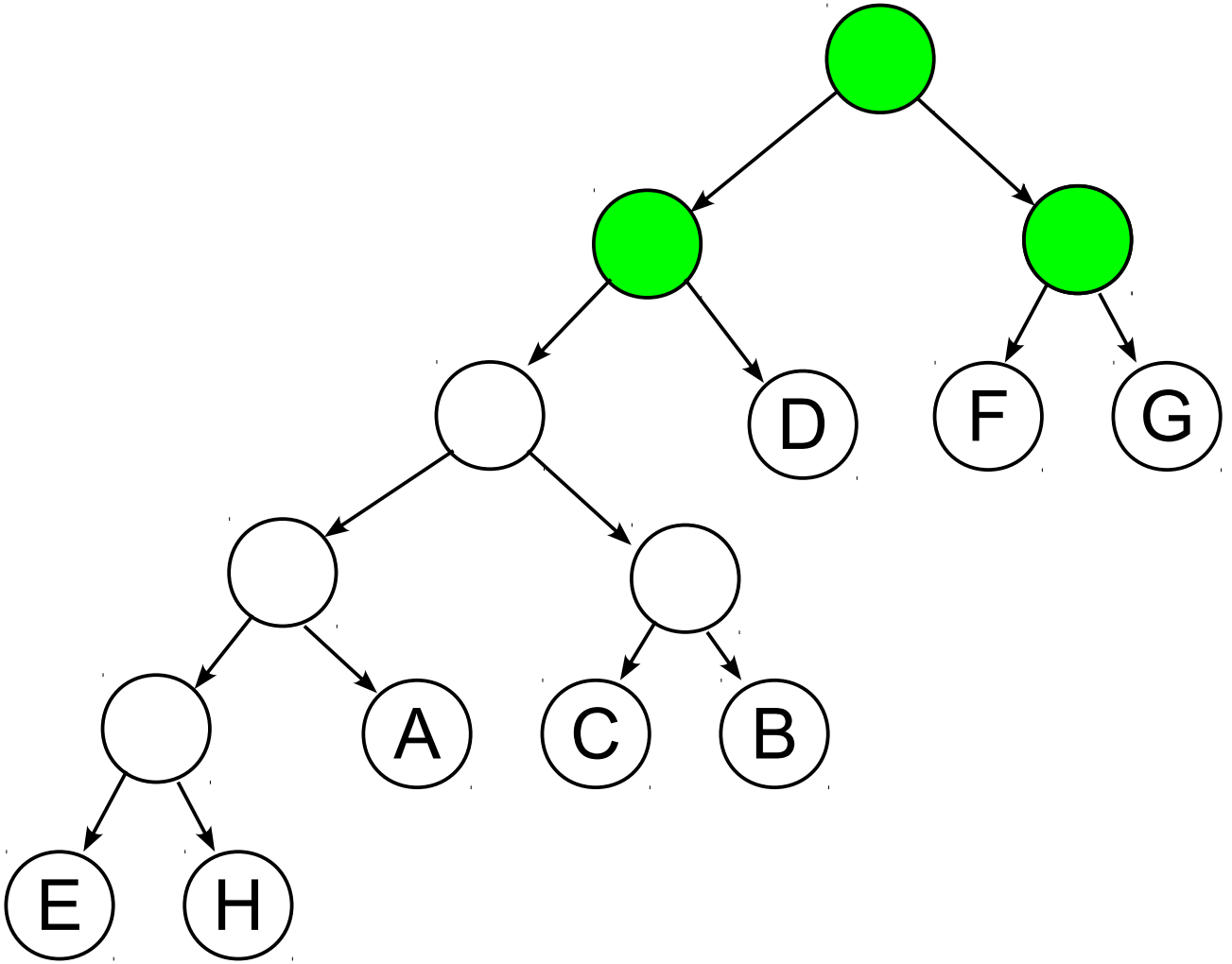
Binary Trees – Case-studies



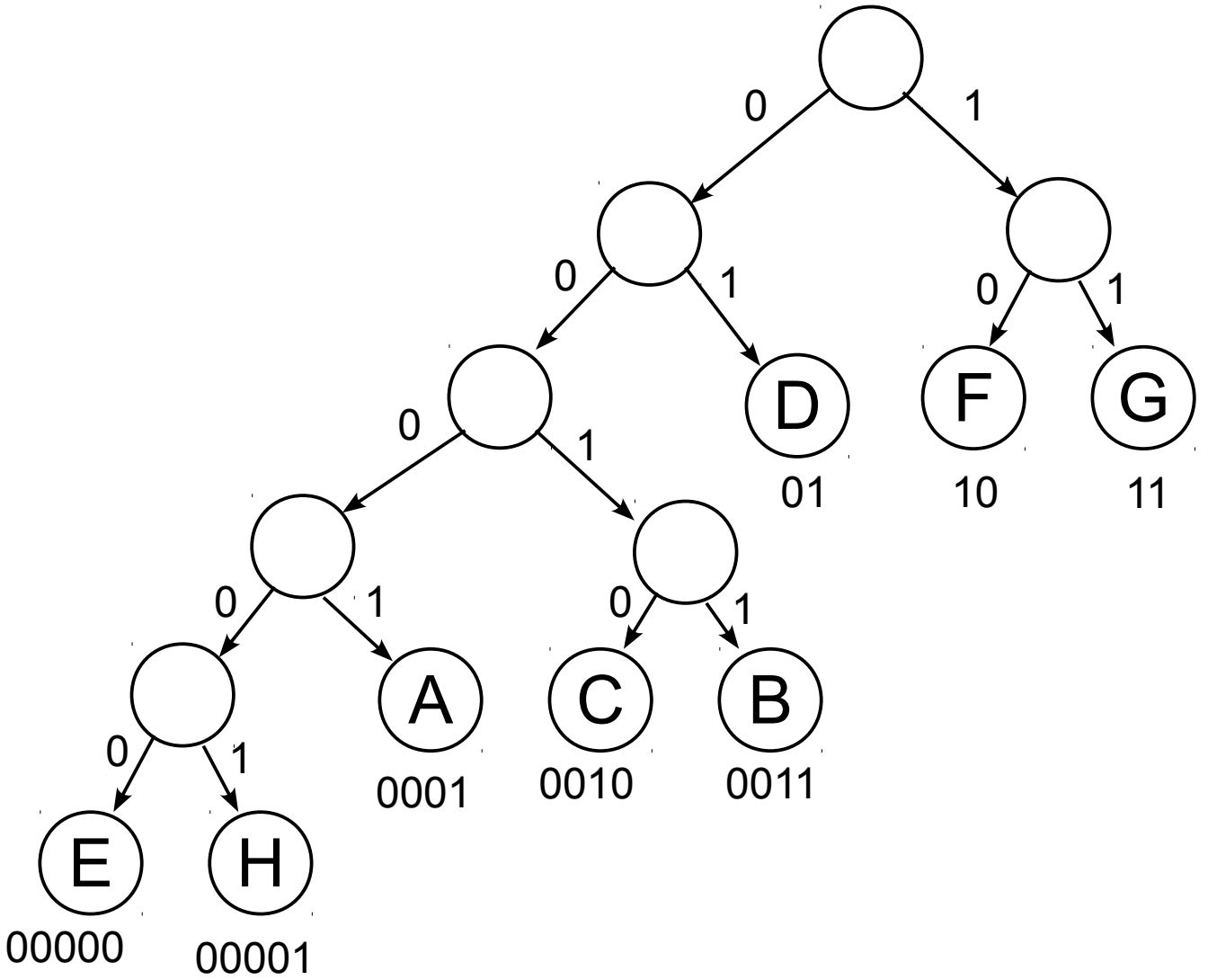
Binary Trees – Case-studies



Binary Trees – Case-studies



Binary Trees – Case-studies



Binary Trees – Case-studies

- So, if we encode a sequence of 100 characters with the “straightforward” encoding, we'd take 800 bits (right? 8 possible symbols, use three bits and assign each symbol to one of the possible 8 combinations)

Binary Trees – Case-studies

- So, if we encode a sequence of 100 characters with the “straightforward” encoding, we'd take 300 bits (right? 8 possible symbols, use three bits and assign each symbol to one of the possible 8 combinations)
- With this encoding, we'll have (assuming the average case), 35 characters being a D, 20 of them being a G, 17 being an F, etc.

Binary Trees – Case-studies

- The important detail being:
  - 35 + 20 + 17 characters will take 2 bits (since D, G, and F all take 2 bits), then 10+7+5 characters will take 4 bits (A, B, and C take 4 bits), and 6 characters will take 5 bits (both E and H take 5 bits) for a total of:

    $72 \times 2 + 22 \times 4 + 6 \times 5 = 144 + 88 + 30 = 262$ bits!

    (compression factor is not that high, but then, the discrepancies in the probabilities were not that high, which is when good compression rates are possible)

Binary Trees – Case-studies

- If the particular data does not fit the probabilities (it could be that by simple randomness, we had 8% of E's, instead of the average of 2%), then the encoding would not be optimal.

Binary Trees – Case-studies

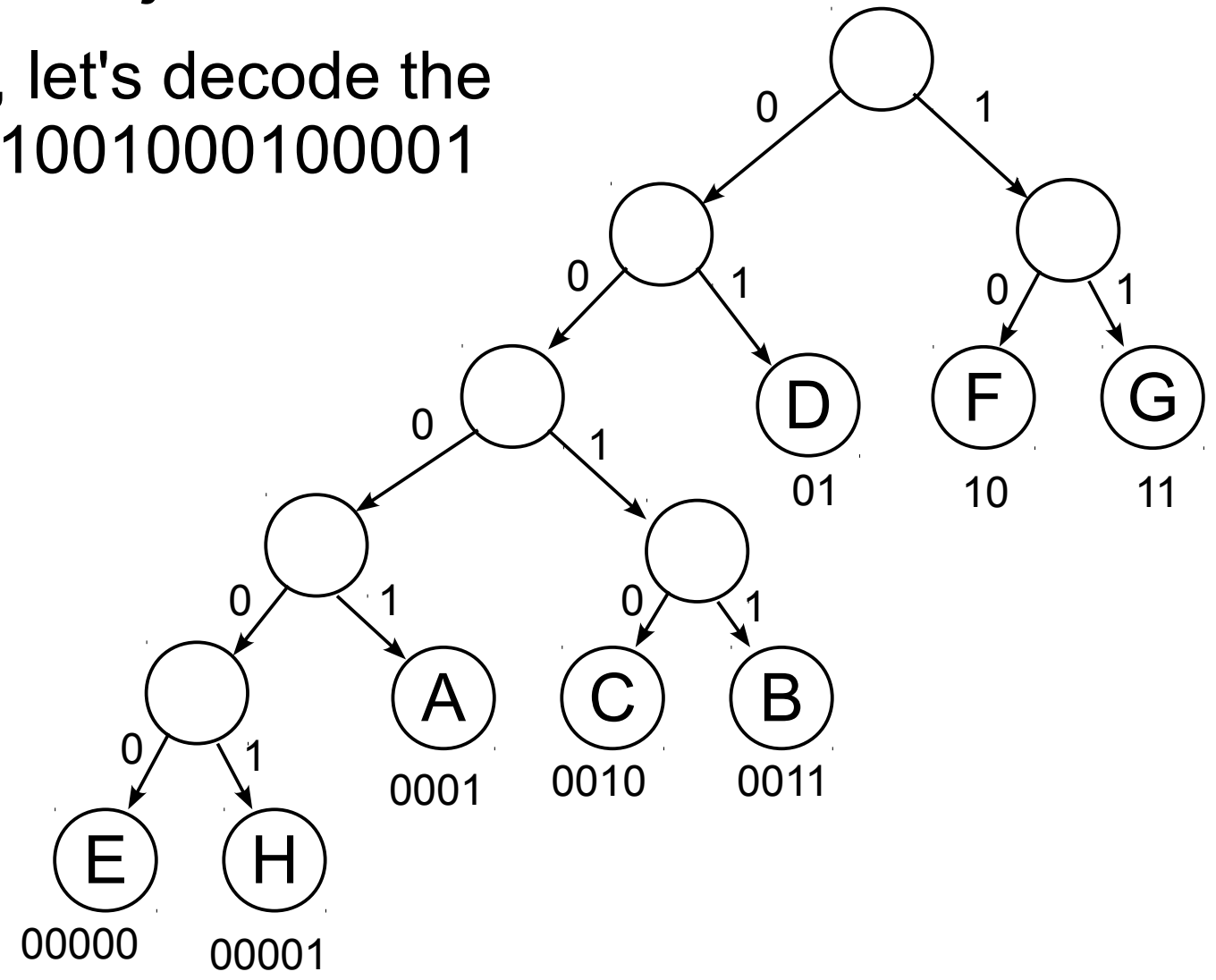
- If the particular data does not fit the probabilities (it could be that by simple randomness, we had 8% of E's, instead of the average of 2%), then the encoding would not be optimal.
  - Depending on the situation, what we could do is build a Huffman Tree for the specific given data (instead of using the “theoretical” probabilities, we just *count* occurrences of each symbol, and use those numbers to build the tree — then we get the optimal encoding for this particular sequence of characters!

Binary Trees – Case-studies

- If the particular data does not fit the probabilities (it could be that by simple randomness, we had 8% of E's, instead of the average of 2%), then the encoding would not be optimal.
  - Depending on the situation, what we could do is build a Huffman Tree for the specific given data (instead of using the “theoretical” probabilities, we just *count* occurrences of each symbol, and use those numbers to build the tree — then we get the optimal encoding for this particular sequence of characters! *Really* neat, huh?

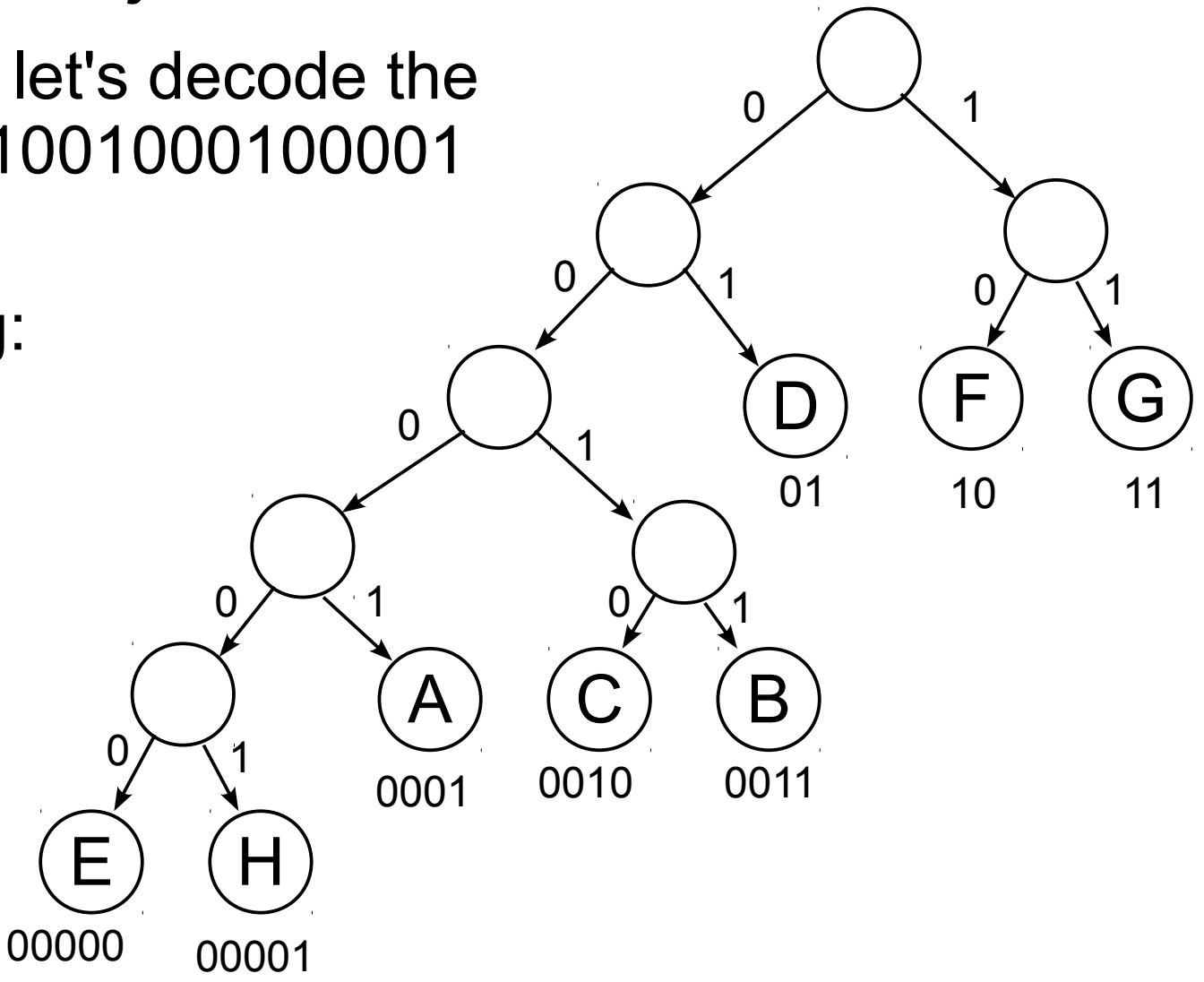
Binary Trees – Case-studies

- For decoding, we just use the tree:
  - For example, let's decode the sequence 011001000100001



Binary Trees – Case-studies

- For decoding, we just use the tree:
  - For example, let's decode the sequence 011001000100001
  - Answer being: DFDAH



Summary

- During today's session, we looked at:
  - Two case-studies, hopefully showing the usefulness and power of the Tree data structure.
  - Both cases are binary trees:
    - Expression trees — useful for compilers to translate human-readable in-fix expressions into CPU-level post-fix expressions; also, possibly to manipulate and optimize expressions.
    - Huffman encoding trees — useful for telecommunications and data storage, as the tool provides an optimal data compression mechanism for prefix codes.