Lecture 8

- Trees for representation
- Tries, decision and classification trees, discrimination nets
- Huffman coding

Reading: Weiss, Ch 10

20 questions

- Suppose you're playing the game of 20 questions for famous people

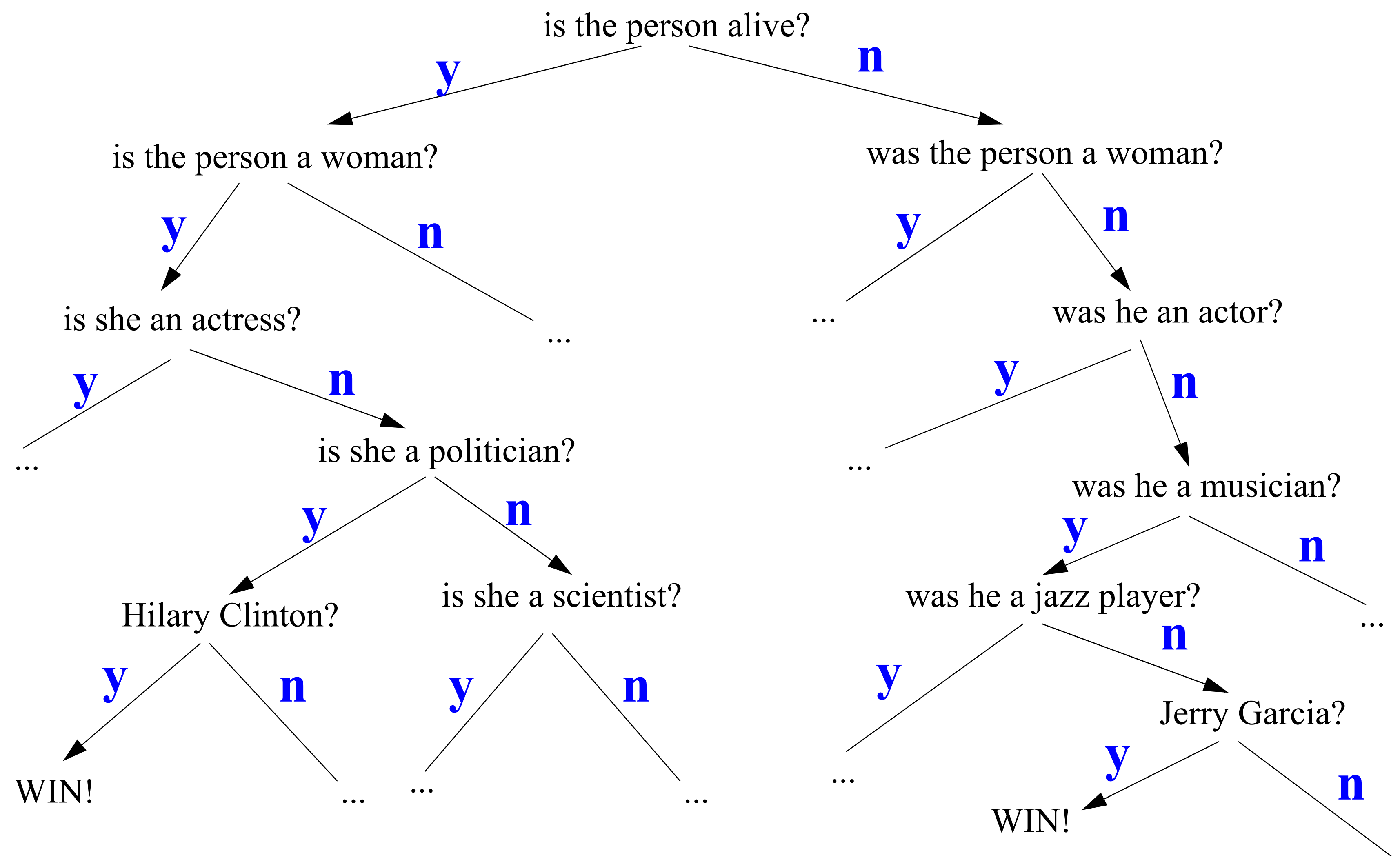
- The rules of the game are:
  - I start by thinking of the name of a famous person.
  - You ask a a yes-or-no question; I give you the answer to that question.
  - Based on my answer, you ask another yes-or-no question, etc.
  - You cannot ask more than 20 yes-or-no questions!
  - If you guess the name of the person I have in mind with one of your questions, you win; otherwise, you lose.

- If you had to think up a fixed question-asking strategy for this game in advance of playing it, what is the maximum number of famous people for which the strategy could win?

Strategies for 20 questions; and trees

- Note that a strategy for playing the 20 questions game has a binary tree structure:
- The first question you ask corresponds to the root node of the tree
- If you get a “yes” answer to the first question, the next question you ask corresponds to a child of the root
- If you get a “no” answer to the first question, the next question you ask corresponds to the other child of the root
- In playing this strategy against an opponent, each game involves following a path from the root towards the leaves
- If you get a “yes” answer to a name-guessing question, you WIN! If you get a “no” answer, the game continues (unless that was your 20th question, in which case you LOSE)

A (partial) 20-questions strategy tree



Optimal 20-questions trees

- Counting only nodes with questions in them (not WIN or LOSE nodes), what is the maximum number of levels in a 20-questions strategy tree? __________

- A 20-questions strategy tree that can win for the greatest number of famous people I might have in mind will be one that has the most "WIN" nodes of any strategy tree with that many levels.
  - What form would this tree take?
    - It would be a completely filled binary tree, with all of the name-guessing nodes at the lowest level
    - Each of those name-guessing nodes would have one "WIN" node associated with it
  - How many WIN nodes are there in such a tree? ___________

Strategy trees as data structures

- Note that a strategy tree for the 20 questions game can at the same time be used as a data structure to store and retrieve names of famous people
- Each famous person in the tree can be represented by the sequence of yes/no answers that leads to that person's name-guessing question node, in that tree
  - In that example strategy tree, Jerry Garcia is represented by the sequence nnnyn (or 00010, if 0 is "no" and 1 is "yes"), Hilary Clinton by the sequence yyny (or 1101)
- With a completely filled binary strategy tree, the result is a pretty efficient data structure: at most $\lceil \log_2 N \rceil$ steps are required to locate any of N famous people in the tree
- Question: Given 2^{19} people, how could you build a 20-questions tree for them?
- Another question: Suppose the game was changed so that you could ask more than 20 questions, but the goal is to ask the fewest number of questions on average before guessing the correct person. How would that change the structure of the strategy tree?

Tries, decision trees, discrimination nets

- A *trie* data structure is a tree in which:
  - internal nodes represent decision rules...
  - edges from a node to its children represent possible outcomes of the decision...
  - (usually only) leaves represent data
- *trie* is pronounced “try”, though it is taken from the word “re**trie**val”. Also known as a *decision tree*, or a *discrimination net*

- Tries don’t have to be binary:
  - a strategy tree for “20 questions” where the questions are multiple choice
  - an “alphabet trie” is a 26-ary tree for storing words
  - a “ternary trie” is a 3-ary tree supporting efficient prefix queries on strings

- Tries don’t have to be completely filled: in general they can have any tree structure

- There are *many* applications of tries (aka decision trees or discrimination nets)...

Some published applications of tries (decision trees)

- Astronomy: Use of decision trees for filtering noise from Hubble Space Telescope images; star-galaxy classification; galaxy counts and discovering quasars in the Second Palomar Sky Survey, etc.
- Biomedical Engineering: Use of decision trees for identifying features to be used in implantable devices
- Financial analysis: Use of decision trees for assessing the attractiveness of buy-writes, etc.
- Internet routing: Tries are used in routing tables to find the next router to handle a packet, based on a prefix sequence of bits in the packet's destination IP address
- Medicine: Decision trees are used for diagnosis in cardiology, psychiatry, and gastroenterology, for detecting microcalcifications in mammography, to analyze Sudden Infant Death (SID) syndrome, for diagnosing thyroid disorders, etc.
- Computer vision: Tree based classification has been used for recognizing three-dimensional objects, and for other high-level vision tasks

- Tries are also used as a data structure in the algorithm for constructing optimal Huffman codes, which we will look at next

Tries that are efficient, on average

- A completely-filled binary trie of height H can store $N = 2^H$ data items
  - all data items are stored in the the leaves, and all leaves are at level H
- It then takes $\log_2 N = H$ steps to access any item best-, worst-, and average-case
  - because you must start at the root and follow H links to find any item
- This is great... *if* each item will be accessed equally often
- But if an item X is going to be accessed much more often than an item Y, it makes sense to store X nearer the root of the trie than Y
- This can produce an unbalanced tree, with poor *worst*-case access path lengths... but if the bad cases happens very rarely, the *average*-case access path length will be improved
- Q: How to construct an optimal trie: one that has, of all possible tries storing a set of items, the smallest average-case path length (averaged according to the probability of accessing each item)
- Huffman's algorithm constructs such a trie...

Coding: some history

- In 1948, Claude Shannon published a paper titled “A mathematical theory of communication” that began the field of information theory
- Shannon proved many fundamental theorems about optimum codes
  - An optimum code uses the minimum possible number of bits to represent the output of an information source, given statistical properties of the source
- However, Shannon did not provide an algorithm for constructing an optimum code!
- In 1952, David Huffman wrote a paper (a term paper in a class he was taking, actually...!) describing an algorithm for constructing an optimal, uniquely decodable, variable-length code for an information source with independent probabilities
- Huffman’s algorithm is still widely used when it is necessary to encode information efficiently, using a relatively small number of bits

Coding and representation

- Suppose you have a set of N possible items and you want to represent them uniquely
- You could construct a binary trie of height $\lceil \log_2 N \rceil$ and use it to represent these items
    - Each item would be represented by a unique sequence of no more than $\lceil \log_2 N \rceil$ yes/no decisions that start at the root of the tree and continue to the leaf where that item is stored
    - The sequence could be represented as a sequence of no more than $\lceil \log_2 N \rceil$ bits: 1 means yes, 0 means no
    - Since data items are stored only in leaves, this gives a "prefix code" for the items: No item is represented by a sequence that is a prefix of the sequence representing any other item
- Using this kind of idea, you could represent a sequence of K items drawn from a set of N possible items using no more than K $\lceil \log_2 N \rceil$ bits (not counting whatever may be needed to store the discrimination net data structure itself)
- So you don't need to use more bits than that... But Shannon and Huffman showed how you can do it with many fewer bits, under some circumstances

Bit efficiency

- Consider a sequence of length K (the “message”) of items drawn independently from a set of N possible items (the “alphabet of symbols”)
- We want to represent the message with as small a total number of bits as possible
- If in that message each of the N possible symbols occur *equally* often...
  - then the best coding scheme uses between K $\lceil \log_2 N \rceil$ and K $\lfloor \log_2 N \rfloor$ bits to represent the message
- But what if some of the symbols occur very rarely? And some of the symbols occur very often?
  - Then if you are smart you will represent the common symbols with short bit sequences, and the rare symbols with longer bit sequences, and use less bits overall
  - As an extreme example, suppose that only 2 of the N possible symbols appear in the message at all:
    - Then you can use “1” to represent one of the symbols, and “0” to represent the other, and encode the message using only K bits!

- Huffman’s algorithm will construct a coding scheme that (among all codes that treat input symbols independently) uses the fewest bits possible to encode a message, given the frequency of occurrence of each possible symbol in the message

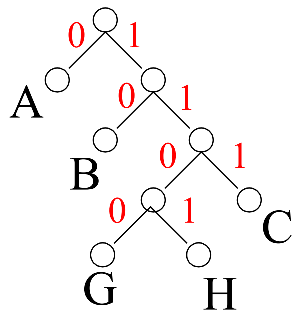
Huffman’s algorithm: frequencies

- Huffman’s algorithm requires first determining the probability (or frequency of occurrence, or count) of each of the possible N symbols as they occur in the information source to be coded
    - (“adaptive” Huffman coding constructs a code “on-line”, as the input is being observed; but we will discuss only the “static”, “off-line”version)
- This can be done using a theoretical probabilistic model, or by collecting statistics from other sequences that you think are similar, or by looking at the actual sequence you are going to code and counting the occurrence of each possible item in it
- For example, suppose:
    - the possible items are the symbols A,B,C,D,E,F,G,H (so N=8)
    - and the message sequence to be coded is AAAAABBAHHBCBGCCC (so K=17)
- Using a completely filled trie, we could construct a code for the possible symbols that would use $\log_2(8) * 17 = 51$ bits to represent this sequence
- But we can do better! The table of counts or frequencies for this message would be:

A: 6; B: 4; C: 4; D: 0; E: 0; F: 0; G: 1; H: 2

How to construct the optimum coding trie?

- The basic idea is:
  - We want common items to be near the root, so they have a short code; rare items can be farther from the root
  - In fact, we want an optimum trie: one that gives the smallest average code length
- We could attempt to build the trie from the top down. For example:
  - A in this example is the most common item... make it a child of the root
  - B,C are the next most common items... put them farther down the tree
  - G,H are the rarest items... make them farthest from the root



  - But while pretty good, this coding trie is *not* optimum, and it’s not easy to come up with an efficient algorithm that will build the optimum coding trie top-down
- Huffman’s breakthrough was instead to build the trie from the bottom up, and he proved that his approach is guaranteed to produce an optimum coding trie

Huffman’s algorithm: constructing the tree

- Once you have the frequencies for all the possible items in the sequence to be coded, Huffman’s algorithm will construct an optimal binary trie for coding
- The algorithm starts with a “forest” of single-node trees, one for each symbol in the input alphabet
- At each step of the algorithm, two trees T1, T2 in the “forest” (which two, do you think?) are joined into one tree T3 by creating a new node that is the root of the new tree T3, and has T1, T2 as its left and right subtrees
- Thus each step of the algorithm reduces the number of trees in the forest by one; the algorithm terminates when there is only one tree, which is the desired coding trie
- The resulting trie is a “full”, but not necessarily “completely filled”, binary tree: every node that is not a leaf has the maximum possible number (2) of children
- ...and the trie is guaranteed to have the smallest possible average path length (average taken according to probability of occurrence of symbols)

Huffman’s algorithm pseudocode

0. Determine the count of each symbol in the input message.

1. Create a forest of single-node trees. Each node in the initial forest represents a symbol from the set of possible symbols, and contains the count of that symbol in the message to be coded. Symbols with a count of zero are ignored (consider them to be impossible).

2. Loop while there is more than 1 tree in the forest:

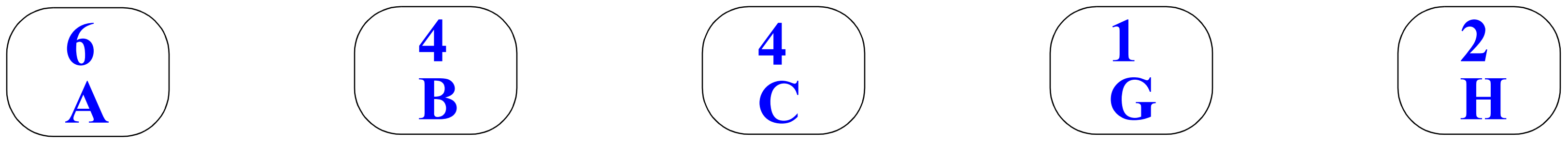
2a. Remove the two trees from the forest that have the lowest count contained in their roots.

2b. Create a new node that will be the root of a new tree. This new tree will have those two trees just removed in step 2a as left and right subtrees. The count in the root of this new tree will be the sum of the counts in the roots of its subtrees. Label the edge from this new root to its left subtree “1”, and label the edge to its right subtree “0”.

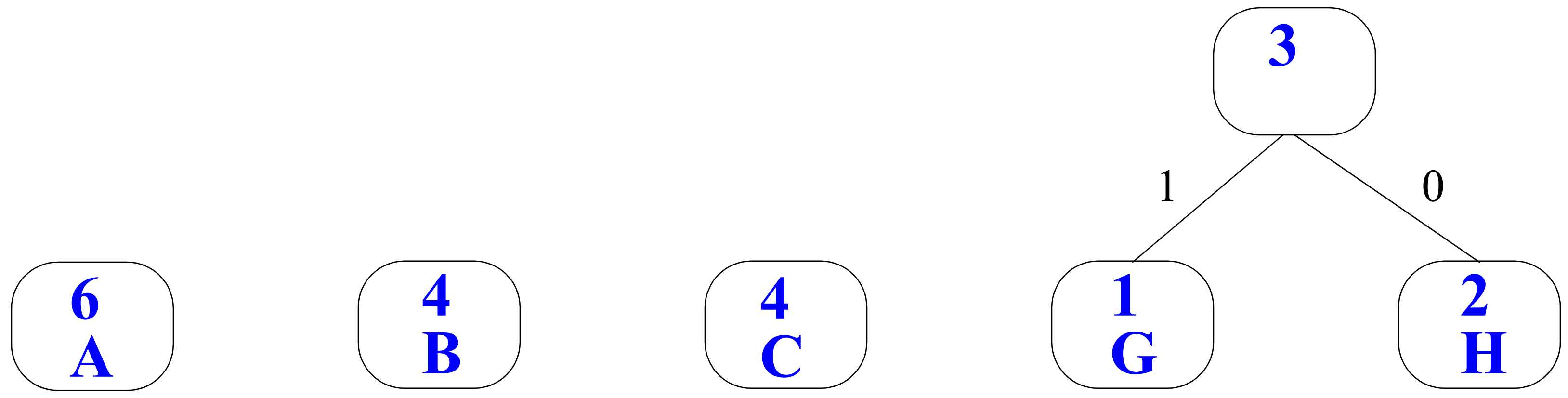
2c. Insert this new tree in the forest, and go to 2.

3. Return the one tree in the forest as the Huffman code tree.

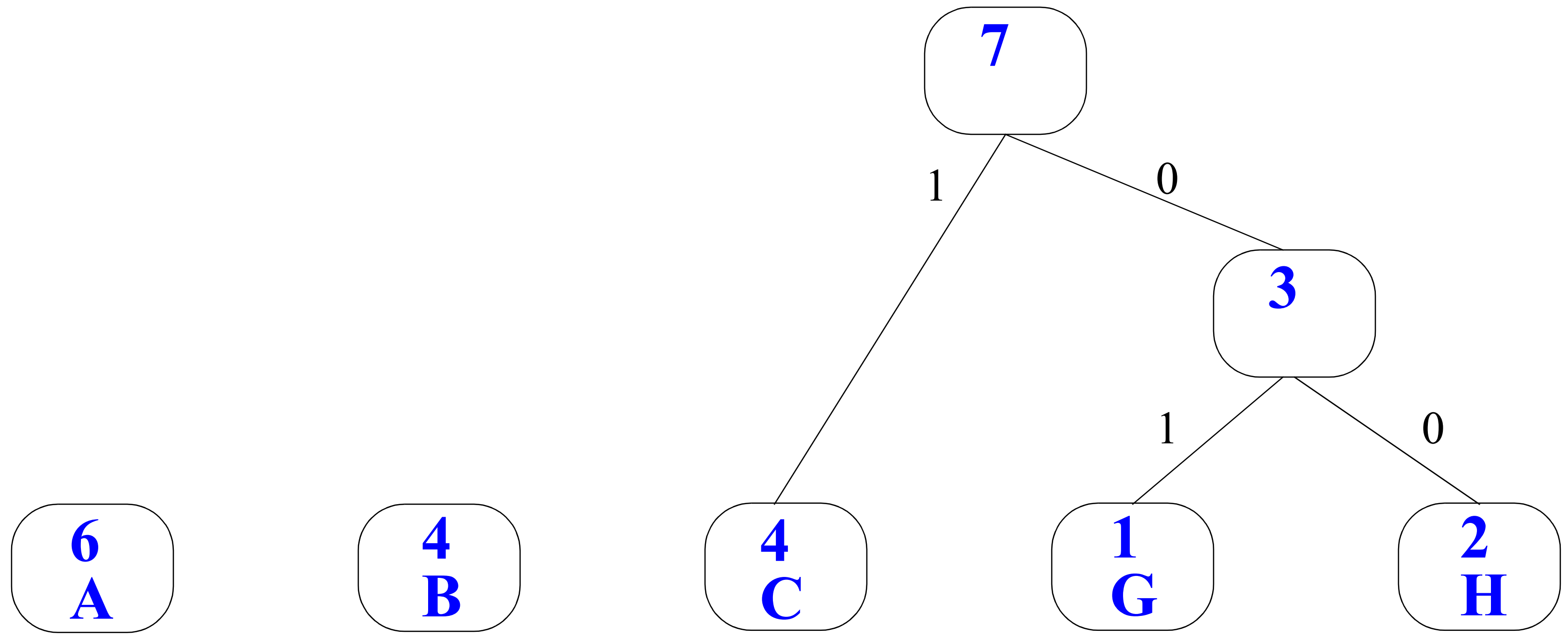
Building the coding tree: the initial forest



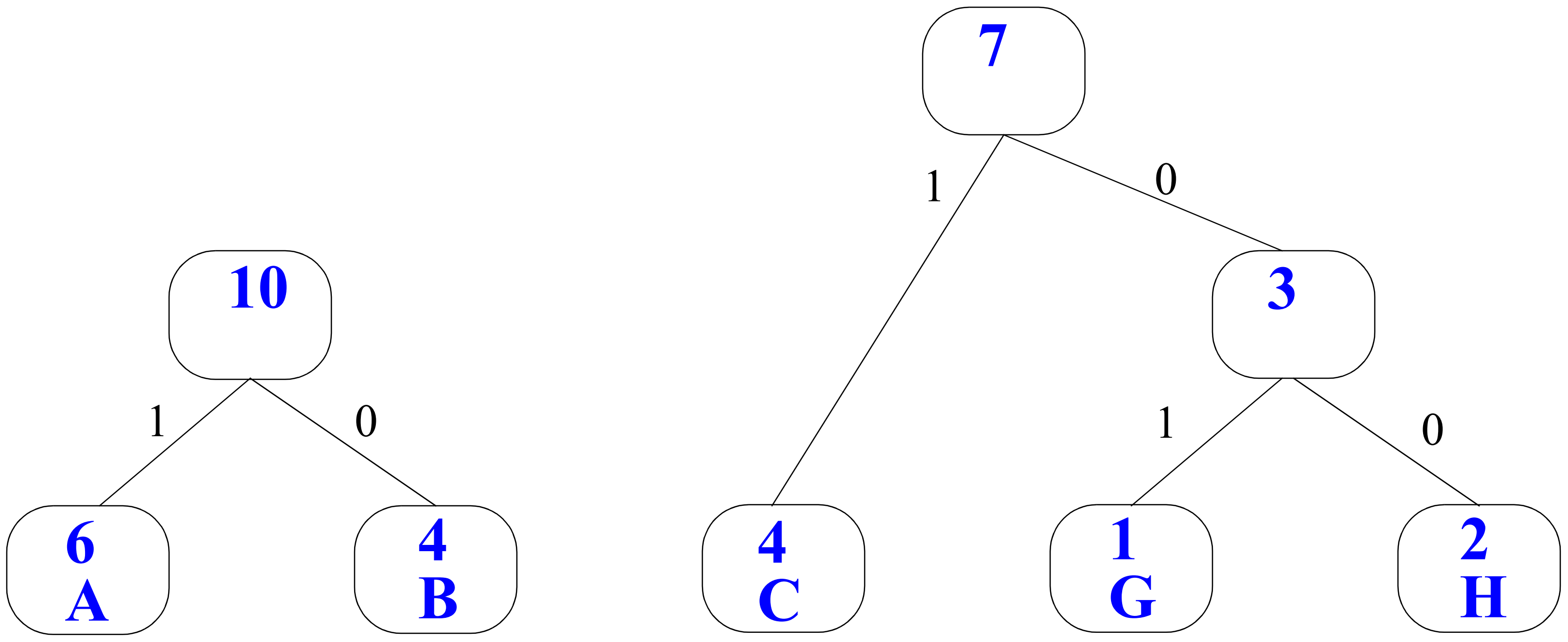
Building the coding tree: forest after step 1



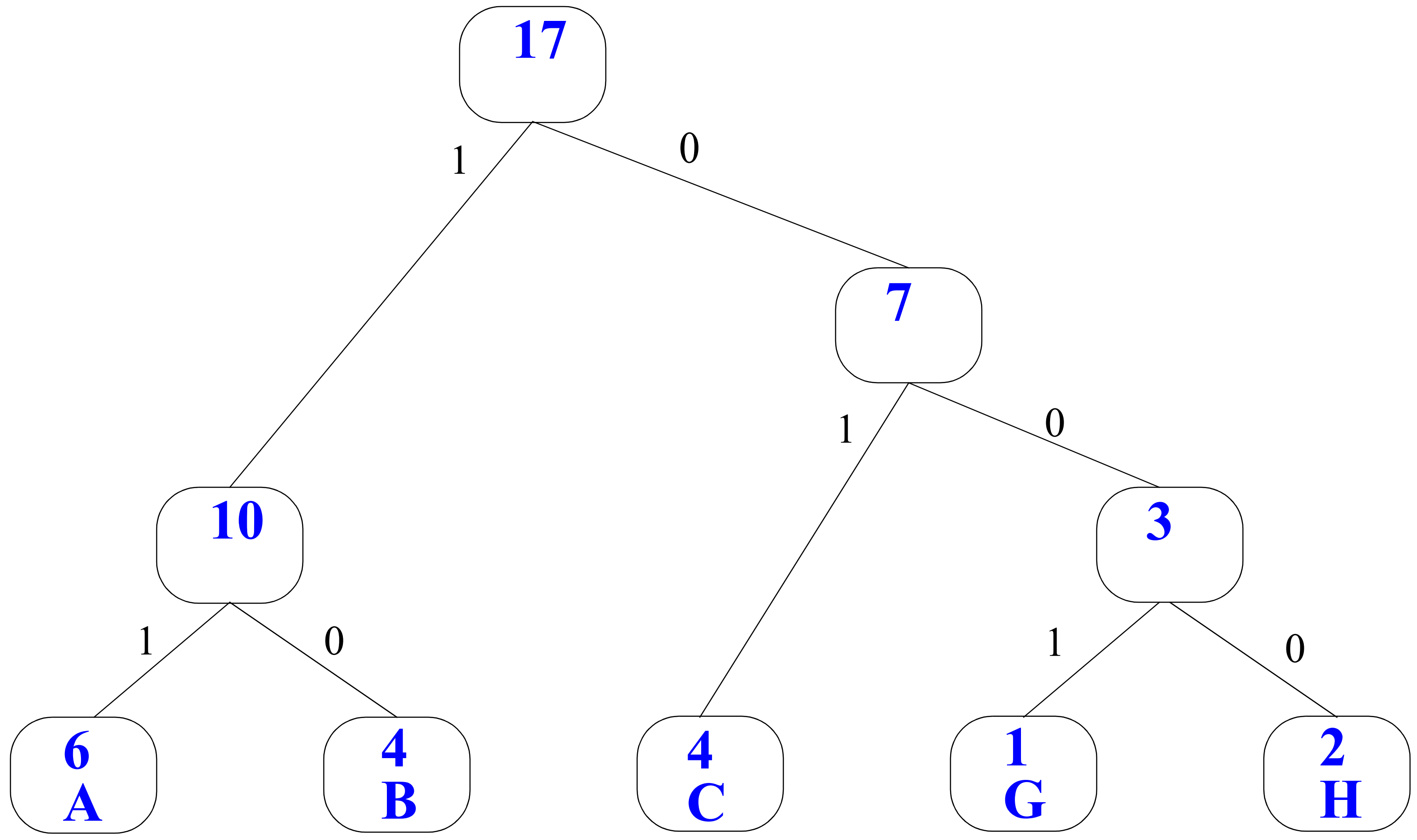
Building the coding tree: forest after step 2



Building the coding tree: forest after step 3



Building the coding tree: done!



Using the code tree to code a message

- Once you have the Huffman code tree, coding the message sequence is easy:
  - For each symbol in the message, output the sequence of bits given by the 1's and 0's on the path from the root of the code tree to that symbol
    - (In practice, this is typically done by starting with the leaf node containing the symbol, follow the path to the root with recursive calls, then output the sequence of bits as the recursive calls pop from the runtime stack; or once the tree has been constucted, you can traverse it and build a lookup table of codes indexed by symbol)
- So, the message sequence A A A A A B B A H H B C B G C C C is coded as...

..... which is ______ bits, which is less than 51

Using the code tree to code a message

```
AAAAABBAHHBCBGCCC  -->  11
AAAAABBAHHBCBGCCC  -->  1111
AAAAABBAHHBCBGCCC  -->  111111
AAAAABBAHHBCBGCCC  -->  11111111
AAAAABBAHHBCBGCCC  -->  1111111111
AAAAABBAHHBCBGCCC  -->  111111111110
AAAAABBAHHBCBGCCC  -->  11111111111010
AAAAABBAHHBCBGCCC  -->  1111111111101011
AAAAABBAHHBCBGCCC  -->  1111111111101011000
AAAAABBAHHBCBGCCC  -->  1111111111101011000000
AAAAABBAHHBCBGCCC  -->  111111111110101100000010
AAAAABBAHHBCBGCCC  -->  11111111111010110000001001
AAAAABBAHHBCBGCCC  -->  1111111111101011000000100110
AAAAABBAHHBCBGCCC  -->  1111111111101011000000100110001
AAAAABBAHHBCBGCCC  -->  111111111110101100000010011000101
AAAAABBAHHBCBGCCC  -->  11111111111010110000001001100010101
AAAAABBAHHBCBGCCC  -->  1111111111101011000000100110001010101
```

... 37 bits!

Using the code tree to decode a message

- With the Huffman code tree, decoding a coded message is also easy
  - Start with the first bit in the coded message, and start with the root of the code tree
  - As you read each bit, move to the left or right child of the current node, matching the bit just read with the label on the edge
  - When you reach a leaf, output the symbol stored in the leaf
  - Start at the root of the code tree again, and read the next bit

- Note: Many distinct Huffman codes exist for a given information source (take any one Huffman tree, exchange "0" and "1" labels on children of any nodes, and get a different code). These are all optimal codes. But it is important to use exactly the *same* code for decoding as for encoding, or you won't be able to reconstruct the input sequence

- The tree, or enough information to reconstruct the tree exactly, must be included before the beginning of the coded message (or in some other way transmitted to the receiver)

Remarks on Huffman codes

- The code constructed by Huffman's algorithm is optimal (it uses the fewest possible bits to code the message) if the occurrences of symbols in the message are probabilistically independent, and the same code is used for a symbol no matter where it occurs
  - this independence doesn't hold in many cases (ordinary English text, for example!) and in those cases other coding techniques can be better
- Huffman's algorithm requires knowing the frequency of occurrence of symbols in the message to be coded, before the code can be constructed
  - other approaches (e.g. adaptive Huffman coding, or the LZW coding used in the zip and gzip utilities) adapt to the statistics in a message "on the fly"
- Like all coding or compression schemes that depend on the message to be coded, enough information must be included in the coded message to permit the message to be decoded (decompressed) later
  - this additional information can be the frequencies of symbols in the original (uncompressed) message, or a representation of the code tree itself
  - for some message sequences (short ones, ones already compressed, etc.) this additional required information may make the "compressed" version longer than the original!

Next time...

- Analysis of Huffman coding
- Huffman coding tree implementation issues
- Priority queues and priority queue implementations
- Heaps
- Dynamic data and array representations for Huffman trees

Reading: Weiss Ch. 6, Ch. 10.1.2