Huffman Codes

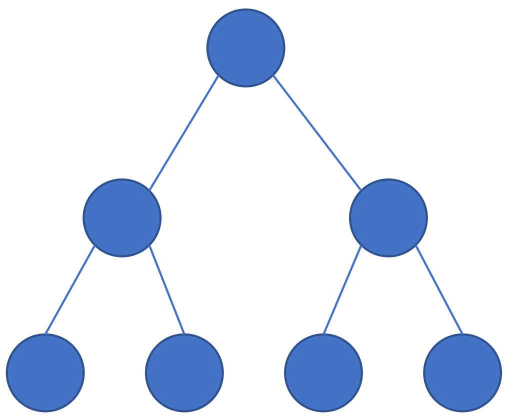
PPT by Brandon Fain

Outline

- **Review: Binary Search Trees**
- **Application: Data Compression and Prefix Codes**
- **Huffman Encoding:** A *lossless* compression code for the characters of a *static* alphabet.

Review: Binary Search Trees

- Binary search trees are data structures with search times dependent on the height of the tree.



- At worst $O(\log_2 n)$ if there are n elements and the tree is balanced.
- Can maintain balance dynamically with a red-black tree.
- What's it good for?

Application: Data Compression and Prefix Codes

- Suppose you want to save a book onto a computer.
- Suppose there are m characters: $\{a_1, a_2, \ldots, a_m\}$ (for example, m=27 to include the lowercase Latin alphabet and blank).
- A document is an array of n characters.
- We want to represent these n characters with as few bits as possible.

Data Compression and Prefix Codes

- The naïve algorithm is as follows:
  - Use binary strings of length $\lceil \log_2 m \rceil$.
  - Each character is uniquely identified with a string.
- This does not exploit any structure of the problem. Suppose we have three characters {a, b, c}, and they appear in our book the following number of times:
  - a appears 1,000,000 times
  - b and c appear 50,000 times each
- The naïve algorithm uses 2,100,000 bits.

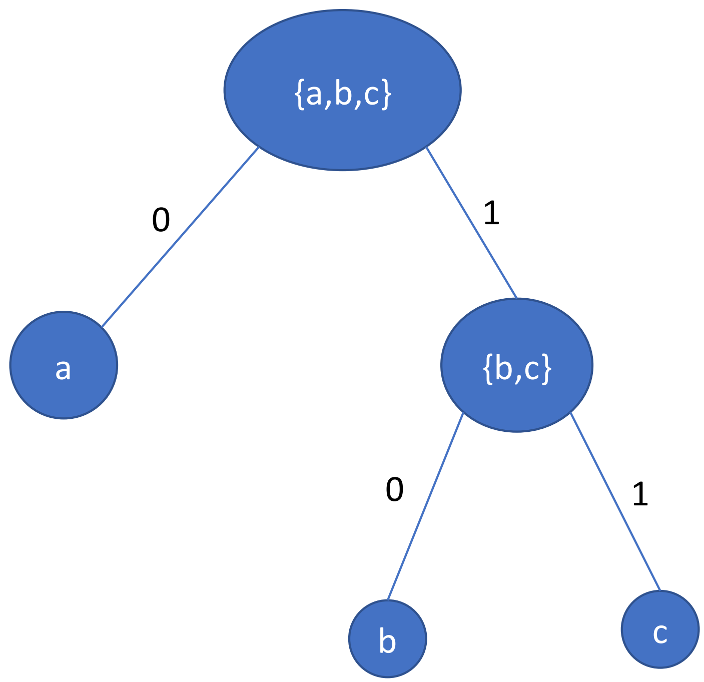
Data Compression and Prefix Codes

- But I claim we could have used just 1,100,000 bits.
- We want the code for the more common character ‘a’ to be shorter than the codes for the less common ‘b’ and ‘c.’ What about:
  - a = 1
  - b = 10
  - c = 11
- Suppose you are trying to decode “1011” Is it “baa” or “bc?”
- To fix this problem, we will use a **prefix code**.

Data Compression and Prefix Codes

- No code string should be a prefix to another. Try:
  - a = 0
  - b = 10
  - c = 11
- Then “1011” is unambiguously “bc.”
- How would we keep track of this in a way that we can look it up quickly when coding/decoding?
- Use a binary tree!

Data Compression and Prefix Codes



- To encode – Search for the leaf corresponding to the character. It’s encoding is the string of bits on edges from the root to the leaf.
- To decode – Every bit gives you an edge to take from the root. Stop when you hit a leaf.
- This means encoding/decoding a character takes time proportional to the depth of the character.

Huffman Encoding

- Ideally, we want all characters to be at low depth in the tree.
- Barring that, we want *common* characters to be at low depth in the tree, potentially by allowing *uncommon* characters to take on high depth.
- Then common characters will take fewer bits of memory, *and* we can decode/encode them faster.
  - (By the way, this is how Unicode actually works)
- This motivates **Huffman encoding**, a greedy algorithm for constructing such a tree.

Huffman Encoding

- **Caveats** – This is a *lossless* code for a *static* alphabet.
- **Lossless code:** You can *always* reconstruct the exact message.
  - In contrast, many effective compression schemes for video/audio (e.g., jpeg) are *lossy*, in that they do not preserve full information.
- **Static alphabet:** The characters and their frequencies remain essentially the same throughout the document.
  - Example: a b c a b c a b c a b c a b c …
  - On the other hand: a a a a a … a b b b b b … b c c c c c … c.
    - There are better ways to store this string!

Huffman Encoding Algorithm

- Recall there are m characters: $\{a_1, a_2, \ldots, a_m\}$ (for example, m=27 to include the lowercase Latin alphabet and blank).
- Suppose character a_k occurs with frequency p_k.
- Algorithm to Construct Tree:
  - Let $A = \{(a_1, p_1), (a_2, p_2), \ldots, (a_m, p_m)\}$
  - While $(|A| > 1)$:
    - Let j and k be the indices of the two smallest values p_j and p_k in A
    - Remove (a_j, p_j) and (a_k, p_k) from A
    - Add a node $(a_j \text{ U } a_k, p_j + p_k)$ to A
    - Add leaf nodes labeled a_j and a_k, if not already present in the tree. Connect them to a parent node labeled a_j U a_k

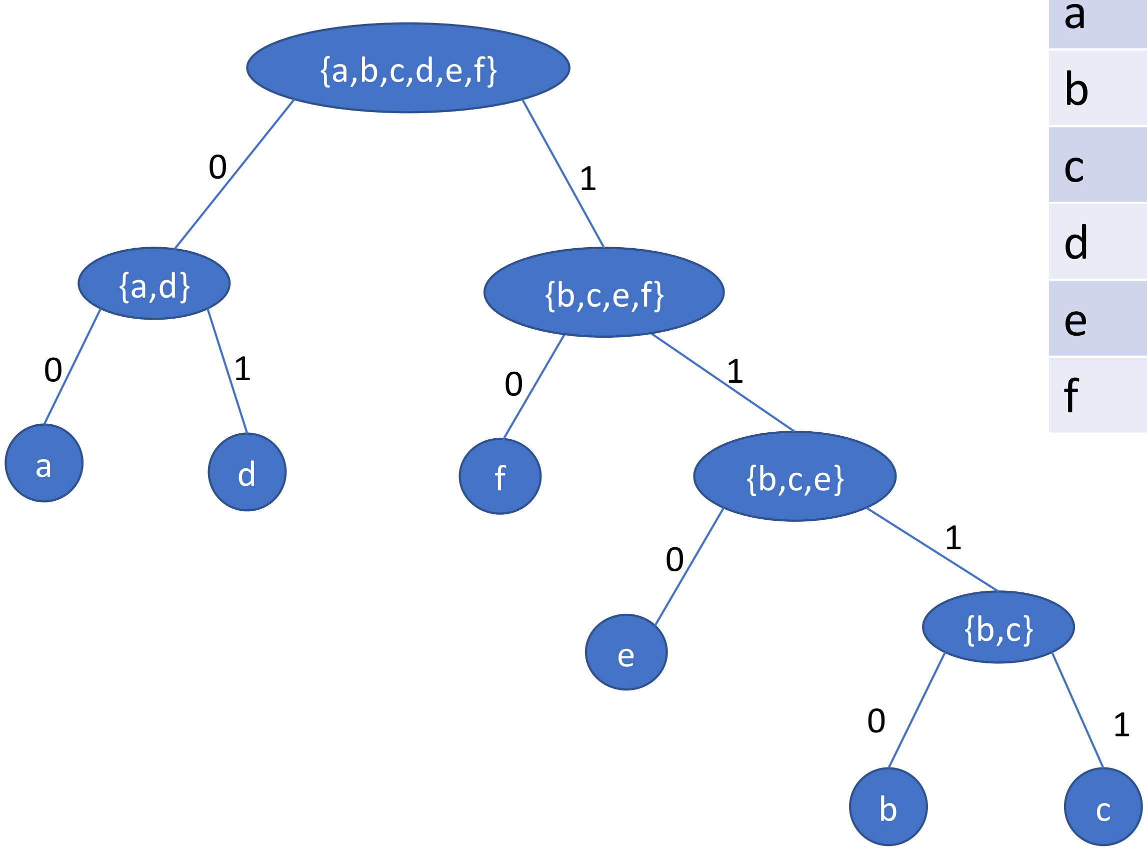
Huffman Encoding: Example

- Break into groups of 3-4.
- By hand, construct the Huffman code for the following alphabet and probabilities:

Character	Probability
a	0.24
b	0.1
c	0.03
d	0.2
e	0.12
f	0.31

- Then encode “fad” and “ceb”

Huffman Encoding: Example



Character	Probability
a	0.24
b	0.1
c	0.03
d	0.2
e	0.12
f	0.31

- “fad” = 100001
- “ceb” = 11111101110

Huffman Encoding: Efficient Implementation

- **Implementation detail** – Note that constructing the Huffman tree requires a *priority queue.*
- A priority queue is a queue maintained on an arbitrary key value, rather than just the insertion order. Supports insertion and extractMin.
- Naively, you could use an array to get O(1) insertion, O(m) extractMin.
- Better idea: use a *heap,* which can be implemented as…
  - Another Binary tree!
  - Yielding an O(log(m)) insertion and extractMin.
- Overall, makes the greedy algorithm O(mlog(m)) instead of O(m^2)

Huffman Encoding: Efficient Implementation

- **Aside** – How much does O(mlog(m)) vs $O(m^2)$ matter anyway?
- Suppose your computer can process 1 billion cycles / second (1 GHz). Then how much time difference does log(m) vs m make?

M	Time in ms for O(mlog(m)) algorithm	Time in ms for $O(m^2)$ algorithm
2^8	0.002	0.066
2^{11}	0.023	4.194
2^{14}	0.229	268.44
2^{17}	2.228	17,179.870 (~ 17 seconds)

Huffman Encoding: Inductive Proof of Optimality

- If character a_k occurs with frequency p_k and has depth d_k, then we need $\sum_{k=1}^{m} p_k d_k$ bits to encode the message.
- **Claim**. Huffman coding is optimal (for any *lossless* code with a *static* alphabet)

- **Proof.** By induction on m.
- Base case. When m=2, Huffman encoding uses a single bit for each character.
- Inductive case. Suppose Huffman encoding is optimal for m characters. Want to show optimality for any alphabet on m+1 characters.

Huffman Encoding

- **Proof (continued).** Let G be an arbitrary alphabet on m+1 characters.
- Let T_G be an optimal binary code tree on G with minimum frequency characters a_1, a_2 as *siblings* (children of a common parent node) of maximum depth in T_G.
- Since characters a_1, a_2 are *siblings,* they have the same depth $d_1 = d_2$ in T_G.
- Consider the alphabet H = (G U {a_0}) - {a_1, a_2}, where a_0 is a new character with frequency $p_0 = p_1+p_2$.
- Let $T_H = T_G$ with a_1 and a_2 removed and their parent replaced with a_0.
- The character a_0 has depth $d_1 - 1$ in the new tree T_H
- Consider encoding with T_H, using a_0 whenever you see a_1 or a_2. Let $B(T_H)$ and $B(T_G)$ be the bits required.

Huffman Encoding: Inductive Proof of Optimality

- **Proof (continued).** Then
  - $B(T_H) = B(T_G) + p_0 d_0 - (p_1 d_1 + p_2 d_2)$
  - $B(T_H) = B(T_G) + (p_1 + p_2)(d_1 - 1) - d_1(p_1 + p_2)$
  - $B(T_H) = B(T_G) - (p_1 + p_2)$
- Now consider the Huffman code trees on H and G; call them S_H and S_G. $B(S_H) \leq B(T_H)$ by the inductive hypothesis, and the same calculations as above give us that $B(S_H) = B(S_G) - (p_1 + p_2)$, so
  - $B(S_G) \leq B(T_H) + (p_1 + p_2)$
  - $B(S_G) \leq B(T_G)$

Conclusions

- Binary trees are useful beyond the “obvious” applications.
- The structure in data can often be exploited (in this case to save memory).
- Huffman Coding compresses only the characters of an alphabet.
- Other algorithms (e.g., Lempel-Ziv) compress strings and give improved compression.