Text Compression with Huffman Codes

CMPT 145

Communicating Text

- When sending files across a network, smaller is better.
- Text compression makes text files smaller
  - Encode the file using a special code.
  - Minimize the size of the encoded data.
- We'll ignore security or secrecy for now.
- A software tool that encodes or decodes is called a codec.

What's a code?

- Code: a system of symbols used to represent a message.
- In a computer, data is encoded by sequences of `0` and `1`.
- Numbers are encoded by binary number systems.
- Computers use different codes for text-based data, e.g.,
    - ASCII
    - UNICODE
- Every character is encoded with a sequence of `0` and `1`.
- In other words, every character is also a positive integer.

Some Codes are fixed-length codes

Letter	ASCII Code	Binary	Letter	ASCII Code	Binary
a	097	01100001	A	065	01000001
b	098	01100010	B	066	01000010
c	099	01100011	C	067	01000011
d	100	01100100	D	068	01000100
e	101	01100101	E	069	01000101
f	102	01100110	F	070	01000110
g	103	01100111	G	071	01000111
h	104	01101000	H	072	01001000
i	105	01101001	I	073	01001001
j	106	01101010	J	074	01001010

All ASCII codes are 8 bits long.

Comparing Fixed Length Codes

- Suppose a file contains lots of `'a'` and `'b'`
  - But no other characters.
- ASCII codes:
  - `'a'` : $\boxed{01100001}$
  - `'b'` : $\boxed{01100010}$
- Special codes:
  - `'a'` : $\boxed{0}$
  - `'b'` : $\boxed{1}$
- Example:
  - File: `'aaab'` (32 bits)
  - Encoded: $\boxed{0001}$ (4 bits)

More characters, more bits

- Suppose a file contains only `'a'`, `'b'`, `'c'`, and `'d'`
- Special codes:
  - `'a'` : $\boxed{00}$
  - `'b'` : $\boxed{01}$
  - `'c'` : $\boxed{10}$
  - `'d'` : $\boxed{11}$
- Example:
  - File: `'aaabaacaad'` (80 bits)
  - Encoded: $\boxed{00000001000010000011}$ (20 bits)

Variable length codes

- Suppose a file contains only `'a'`, `'b'`, `'c'`, and `'d'`
- Suppose that `'a'` appears most often; `'d'` least often.
- A variable length code:
  - `'a'` : $\boxed{0}$
  - `'b'` : $\boxed{10}$
  - `'c'` : $\boxed{110}$
  - `'d'` : $\boxed{111}$
- Example:
  - File: `'aaabaacaad'` (80 bits)
  - Encoded: $\boxed{000100011000111}$ (15 bits)
- The more imbalanced the frequencies, the better!

Text compression overview

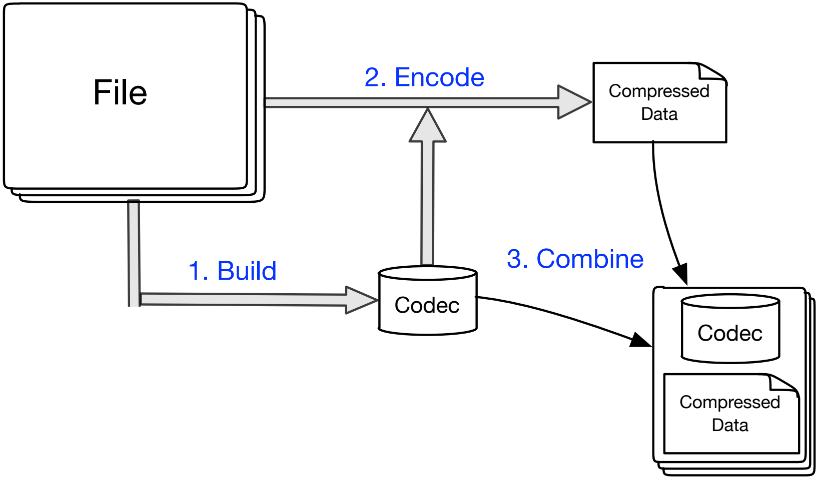
- To compress a given file:

  1. Create a code for the given file
  2. Use the code to encode the file contents
  3. Store the encoded data to a new file
  4. Store the description of the code in the new file too.

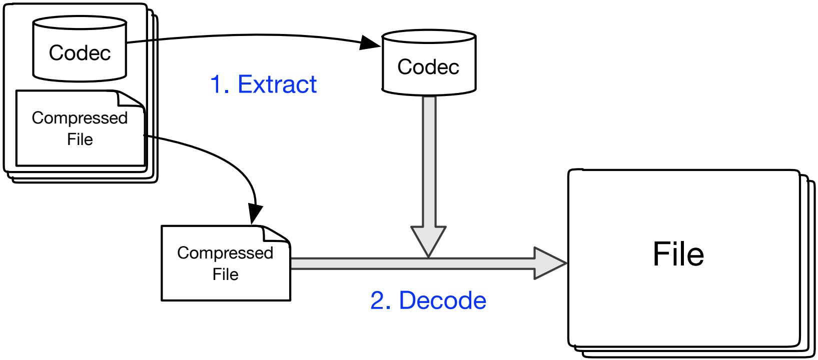
- To undo the effects of text compression:

  1. Open a compressed file
  2. Read the description of the code in the file
  3. Decode the rest of the file
  4. Store the decoded data in a new file

Compression



De-compression



Huffman Codes

- Used for text compression.
- Variable length codes.
- The codes depend on the file's contents.
- A character's code depends on frequency in the file:
  - Frequent letters have shorter codes.
  - Infrequent letters have longer codes.

Huffman Encoding Algorithm

- Count each character in the file.
- Build a binary tree from the counts.
- Build a code from the binary tree.
- Encode the file contents.

Counting Characters

```python
def count_characters(contents):
    """
    Purpose:
        Count the characters in the contents.
    Pre-conditions:
        :param contents: a list of strings.
    Return:
        :return: a list of (character,frequency) tuples
    """
    freqs = dict()
    for line in contents:
        for char in line:
            if char in freqs:
                freqs[char] += 1
            else:
                freqs[char] = 1
    return list(freqs.items())
```

Notes on Counting Characters

- The file has already been opened and read.
- The contents are stored in a list of strings.
- This design decision is pedagogical.
- The Python is shorter if the content is one long string.
- Debugging is easier with a list of strings.

Huffman Trees

- A Huffman Tree is a binary tree node, with 4 fields:

  char A single character
  freq The number of times the `char` occurred.
  left A reference to another `HuffmanTree`.
  right A reference to another `HuffmanTree`.

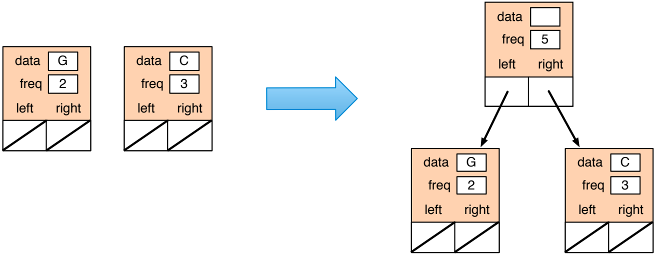
HuffmanTree Class

```python
class HuffmanTree(object):

    def __init__(self,freq=0,char=None,left=None,right=None):
        """
        Purpose:
            Initializes the HuffmanTree object.
            :param freq: a positive integer
            :param char: a character
            :param left: another HuffmanTree
            :param right: another HuffmanTree
        """
        self.char = char
        self.left = left
        self.right = right

        if left is None and right is None:
            self.freq = freq
        else:
            self.freq = left.freq + right.freq
```

Combining two HuffmanTrees



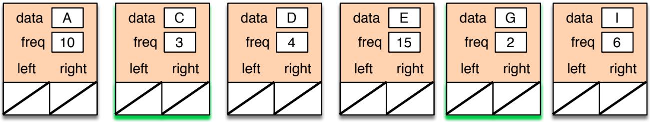

```
leafa = HuffmanTree(char='G', freq=2)
leafb = HuffmanTree(char='C', freq=3)

combined = HuffmanTree(left=leafa, right=leafb)
```

Algorithm: Building a Huffman Tree

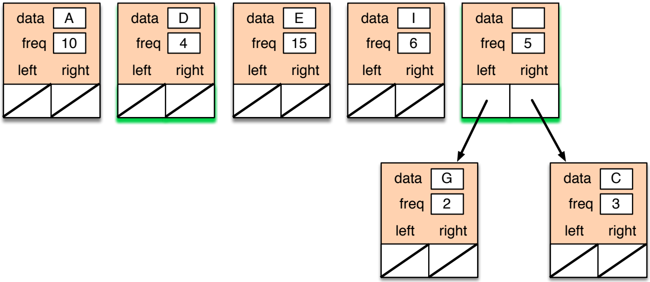
1. Create a leaf node, for every character, with its frequency.
2. Repeat until there is one single Huffman Tree:
   - Pick the two lowest frequency trees
   - Combine these as children to new node, adding frequencies.

Building a Huffman Tree: Step 1

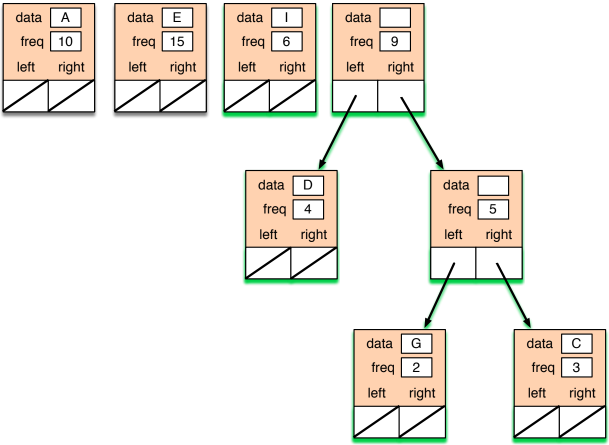


- Create a leaf node, for every character, with its frequency.
- Identify the two trees with the lowest frequency (highlighted green)
- Combine!

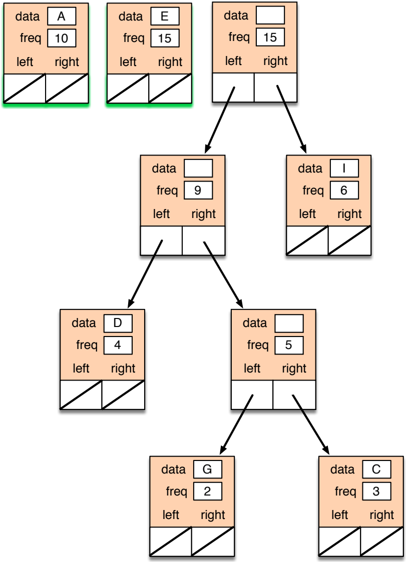
Building a Huffman Tree: Step 2



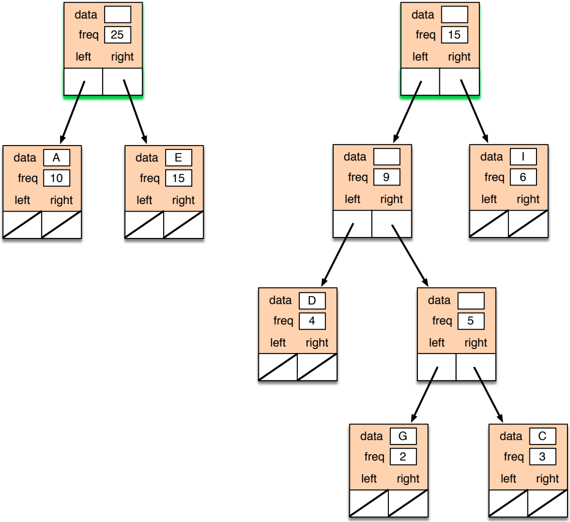
Building a Huffman Tree: Step 3



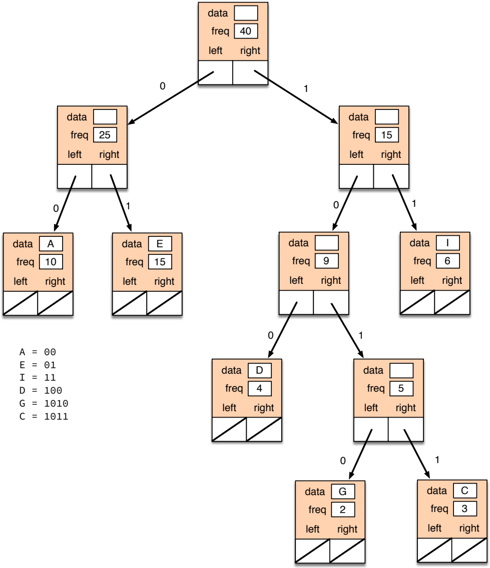
Building a Huffman Tree: Step 4



Building a Huffman Tree: Step 5



Building a Huffman Tree: Step 6




```
A = 00
E = 01
I = 11
D = 100
G = 1010
C = 1011
```

Python for building the tree

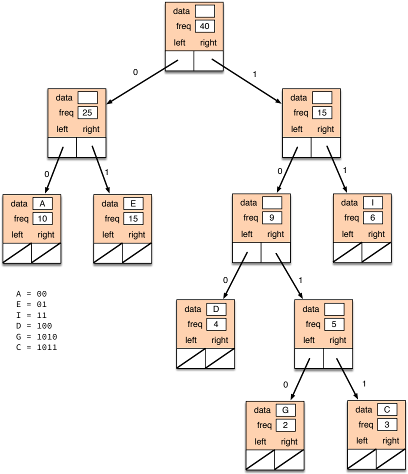
```python
def build_huffman(freq_list):
    """
    Purpose:
        Build a HuffmanTree from the frequency list.
    Pre-conditions:
        freq_list: A list of (character,frequency) pairs.
    Return:
        A HuffmanTree
    """
    trees = [HuffmanTree(freq=f,char=c) for c,f in freq_list]
    while len(trees) > 1:
        t1 = delete_min(trees)
        t2 = delete_min(trees)
        trees.append(HuffmanTree(left=t1, right=t2))

    return trees[0]
```

Creating the Codes from the Tree

To create the codes:

1. Walk path from root to each leaf
2. Use $\boxed{0}$ when you go left
3. Use $\boxed{1}$ when you go right




```
A = 00
E = 01
I = 11
D = 100
G = 1010
C = 1011
```

Creating the Codes from the Tree

- Strategy:
    - A normal recursive traversal of the tree
    - But keep track of the code so far
    - Append $\boxed{0}$ to the code when you go left
    - Append $\boxed{1}$ to the code when you go right
    - If you reach a leaf, store the code in a dictionary.
- Terminology: a codec is a dictionary mapping from single character to its code.

Creating the Codes

```python
def build_codec(htree):
    """
    Purpose:
        Build a dictionary of char-code pairs from the tree.
    Return:
        :return: a dictionary mapping character to code
    """
    codec = {}
    def encoder(tree, code):
        if is_leaf(tree):
            codec[tree.char] = code
        else:
            encoder(tree.left,  code+'0')
            encoder(tree.right, code+'1')

    if is_leaf(htree):
        # special case: message contains only one character
        codec[htree.char] = '0'
    else:
        encoder(htree, '')
    return codec
```

Notes on the Creating the Codec

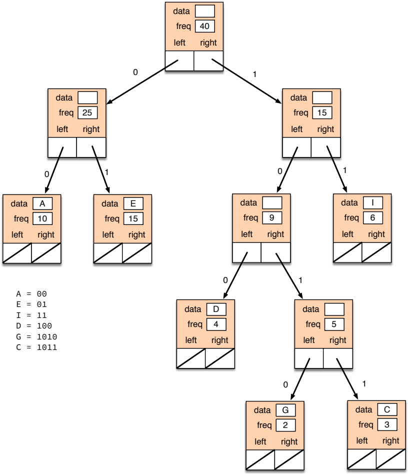
- The internal function uses an external dictionary.
- This design decision made the Python slightly easier to read.
- The Huffman encoding algorithm assumes the text has at least 2 different characters.
- A trivial, sub-optimal code is created if the file only contains one character.

Encoding a message

To encode a string:

- For each letter
- Look up the code
- Append the code to the output string.

The word 'EDGE' is encoded by $\boxed{01100101001}$




```
A = 00
E = 01
I = 11
D = 100
G = 1010
C = 1011
```

Encoding the contents of a file

```python
def encode(strings, codec):
    """
    Purpose:
        Use the codec to encode the strings.
    Pre-conditions:
        strings: A list of strings to encode.
        codec: A dictionary mapping characters to codes
    Return:
        a list of encoded strings
    """
    output = []

    # encode the message
    for s in strings:
        encoded = []
        for char in s:
            encoded.append(codec[char])
        output.append(''.join(encoded))

    return output
```

Notes on encoding a file

- The code is designed based on the file's contents.
- Each file will get a different code.
- We cannot decode the file if we don't know the codec!
- So we include the codec in the compressed file.
- Including the codec adds to the file.
- The codec itself cannot be encoded!

Decoding an encoded message

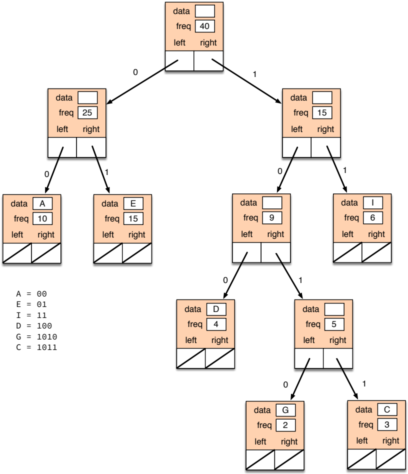
- Decoding is harder!
- The codes are variable length, but the encoded message is one long string.
  - E.g., $\boxed{\texttt{101100101001}}$
- No easy way to separate the codes into the right lengths.
- Two strategies:
  - Use the code to direct a path through the tree
  - Create an inverse codec, mapping from code to character.

Decoding an encoded message

To decode a message:

- Start at the root
- Look at the message, one bit at a time.
- Go left on $\boxed{0}$
- Go right on $\boxed{1}$
- When you hit a leaf, take the character; return to root.

The message $\boxed{101100101001}$ takes us through the tree 4 times: 'CAGE'



Decoding using an inverse codec

- The inverse codec is a mapping from codes to characters.
- Problem: The codes are variable length, so we cannot simply look up the right code.
- Solution:

  1. Set $k = 1$.
  2. If the first k bits are in the codec, look up the character.
  3. Otherwise, set $k = k + 1$, and try step 2 again.

Decoding using an inverse codec: Example

- $k = 1$: $\boxed{\texttt{101100101001}}$
- $k = 2$: $\boxed{\texttt{101100101001}}$
- $k = 3$: $\boxed{\texttt{101100101001}}$
- $k = 4$: $\boxed{\texttt{101100101001}}$: 'C'
- $k = 1$: $\boxed{\texttt{101100101001}}$
- $k = 2$: $\boxed{\texttt{101100101001}}$: 'A'
- ...

- Inverse Codec:
  - $\boxed{\texttt{00}}$: 'A'
  - $\boxed{\texttt{01}}$: 'E'
  - $\boxed{\texttt{11}}$: 'I'
  - $\boxed{\texttt{100}}$: 'D'
  - $\boxed{\texttt{1010}}$: 'G'
  - $\boxed{\texttt{1011}}$: 'C'

Summary

- Counting: Every character in the file.
- Building the Huffman tree: Combine trees until 1 remains.
- Building the codes: Walk every path from root to leaf.
- Encoding the file: Every character in the file.
- Decoding: Two strategies!

Are there any places we could do better?

Decoding: Which strategy is better?

- Inverse Codec:
  - Dictionary is checked once per coded digit
- Tree:
  - Take a step in the tree once per bit in the code.
- Which is more costly: check or step?

Algorithm: Building a Huffman Tree (recap)

1. Create a leaf node, for every character, with its frequency.
2. Repeat until there is one single Huffman Tree:
    - Remove the two lowest frequency trees
    - Combine these as children to new node, adding frequencies.
    - Add the combined tree to the list.

- Can we do better than linear search in step 2?
- Is it worth trying?

Cost of Linear Search?

- Idea #0: (original algorithm)
  - Assume the list of trees is not sorted
  - (N is the number of unique characters)
- Repeat until 1 tree left:
  - Remove a single tree: $O(N)$ (worst case)
  - Combine: $O(1)$
  - Re-insert: $O(1)$
- The list gets smaller by one until there is one tree left.
- Total cost: $O(N) + O(N-1) + \cdots + O(1) = O(N^2)$

Better than Linear Search?

- Idea #1:
  - Keep the list of trees sorted by frequency
- Initial sorting: $O(N \log N)$
- Repeat until 1 tree left:
  - Remove first tree: $O(1)$
  - Combine: $O(1)$
  - Re-insert: $O(N)$ (worst case)
- Total cost: $O(N \log N) + O(N^2) = O(N^2)$

Can we do better?

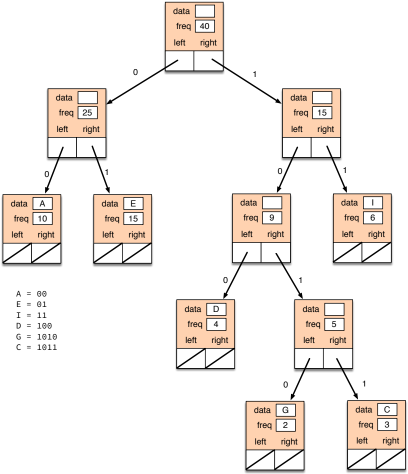
Study the final tree

Each parent node has a higher frequency than its children.

- E.g., the parent of `'A'` and `'E'`

Each combined node has higher frequency than any previous combined node.

- E.g., the root was created after its children had been created.



Better than Linear Search?

- Idea #2: Keep 2 sorted lists
  1. Sorted list of (original) leaf nodes.
  2. List of trees created by combination, initially empty.
- The smallest trees are at the front of the two lists.
- Every newly created tree is enqueued to the second list.
- Both lists stay sorted with minimal effort!

Better than Linear Search!

- Idea #2:
  - Keep 2 sorted lists
- Sort list of (original) leaf nodes: $O(N \log N)$
- Repeat until 1 tree left:
  - Remove smallest tree (either list): $O(1)$
  - Combine: $O(1)$
  - Enqueue (second list): $O(1)$
- Total cost: $O(N \log N) + O(N) = O(N \log N)$

Implementation

- Working with two lists makes the algorithm complicated!
- Simplify using a new ADT, HuffmanHeap:
  - **Initialization**:
    - Sort the characters by frequency
    - Create leaf nodes from the sorted list
    - All leaf nodes into list 1.
  - **Dequeue**:
    - Check the front of both lists for the smallest.
  - **Enqueue**:
    - Always enqueue only to the second list
- A Heap is a kind of Queue where the dequeue operation returns the minimal value.

Using the HuffmanHeap

```python
def build_codec(freq_list):
    """
    Purpose:
        Build a codec from the frequency list.
    Pre-conditions:
        :param freq_list: A list of (char,frequency) tuples.
    Return:
        :return: a dictionary mapping characters to codes
    """
    freq_list.sort(key=lambda p: p[1])
    hq = HH.HuffmanHeap([HT.HuffmanTree(freq=f,char=c) \
                         for c,f in freq_list])
    while len(hq) > 1:
        t1 = hq.dequeue()
        t2 = hq.dequeue()
        hq.enqueue(HT.HuffmanTree(left=t1, right=t2))
    survivor = hq.dequeue()
    return survivor.build_codec()
```

Notes on using the HuffmanHeap

- This looks very similar to the original function (Slide 25) .
- Sorting is accomplished by telling `sort()` to look at the frequencies.
- The loop to combine two trees calls `dequeue()` twice
- The HuffmanHeap class hides the details of maintaining two lists!

Lessons Learned

- The Huffman Coding example teaches many lessons!
    - Trees are useful in surprising ways.
    - Use data structures to solve problems.
    - Hide complex code behind the interface of an ADT.
    - Programming is not the final skill of a programmer.
    - Computer Science is the science of problem solving, not programming.