Announcements

• Homework #5 due date moved to Monday after spring break (3/28) at midnight.
Approaching a Linked-List Problem

**Problem:** Insert $v$ into ordered list $L$ nondestructively.

$L: \boxed{7 \rightarrow 15 \rightarrow 17}$

Result: $\boxed{- \rightarrow 7 \rightarrow 15 \rightarrow 17}$

$L: \boxed{7 \rightarrow 15 \rightarrow 17}$

Result: $\boxed{- \rightarrow 7 \rightarrow 15}$

$L: \boxed{7 \rightarrow 15 \rightarrow 17}$

Result: $\boxed{- \rightarrow 7 \rightarrow 15}$

$L: \boxed{7 \rightarrow 15 \rightarrow 17}$

Result: $\boxed{- \rightarrow 7 \rightarrow 15 \rightarrow 16}$
Breaking It Down

- Here are the repeated steps:

  - Dashed arrows show information needed to do each step.
  - These are the variables you’ll need.
  - After using them, must move them into position for next time.
next = Link(curr.first)
last.rest = next
# For next loop:
curr = curr.rest
last = next
A Worked Example: Decoding with Trees

- In Unicode or ASCII, characters are represented in internally with fixed-length groups of bits:

<table>
<thead>
<tr>
<th>Character</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>00000000001100001</td>
</tr>
<tr>
<td>'0'</td>
<td>00000000000110000</td>
</tr>
<tr>
<td>':'</td>
<td>00000000000111010</td>
</tr>
<tr>
<td>'π'</td>
<td>00000011110000000</td>
</tr>
<tr>
<td>'♡'</td>
<td>0010011001100001</td>
</tr>
</tbody>
</table>

- These are easy to deal with internally; characters represented as pairs of 8-bit bytes.
- Obvious where one ends and the next begins.
- But not very compact.
Prefix Codes

• An alternative is to use a *variable-length code* in which characters are represented with varying numbers of bits.

• Shorter sequences of bits can represent more common characters, minimizing total length of message.

• But the input is just a sequence of bits; have to decide where each character starts in this sequence.

• One approach is to use a *prefix code*:

  The bit sequence representing a character is never a prefix of the bit sequence representing any other character.

• So, can always tell when a character ends: one more bit will yield an invalid sequence. No special separator needed.
### Example

<table>
<thead>
<tr>
<th>Character</th>
<th>Bits</th>
<th>Character</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1110</td>
<td>N</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>101100</td>
<td>O</td>
<td>1101</td>
</tr>
<tr>
<td>C</td>
<td>01000</td>
<td>P</td>
<td>101101</td>
</tr>
<tr>
<td>D</td>
<td>11111</td>
<td>Q</td>
<td>001011010</td>
</tr>
<tr>
<td>E</td>
<td>011</td>
<td>R</td>
<td>1000</td>
</tr>
<tr>
<td>F</td>
<td>00110</td>
<td>S</td>
<td>1001</td>
</tr>
<tr>
<td>G</td>
<td>111100</td>
<td>T</td>
<td>000</td>
</tr>
<tr>
<td>H</td>
<td>0101</td>
<td>U</td>
<td>01001</td>
</tr>
<tr>
<td>I</td>
<td>1100</td>
<td>V</td>
<td>001010</td>
</tr>
<tr>
<td>J</td>
<td>001011001</td>
<td>W</td>
<td>111101</td>
</tr>
<tr>
<td>K</td>
<td>0010111</td>
<td>X</td>
<td>001011011</td>
</tr>
<tr>
<td>L</td>
<td>10111</td>
<td>Y</td>
<td>00100</td>
</tr>
<tr>
<td>M</td>
<td>00111</td>
<td>Z</td>
<td>001011000</td>
</tr>
</tbody>
</table>

So, "CAT" encodes as

```
00000000001000011000000000010000010000000001010100
C A T
```

using plain Unicode, but using the code above:

```
010001110000
C A T
```
Problem: How to decode?

- Can’t just look up a character code until we know where it ends.
- Our plan is to consume bits (actually, we’ll use strings of characters ‘0’ and ‘1’ for simplicity) from the left.
- Emit a decoded character whenever we determine that we’ve accumulated a complete character code.
- A binary tree is one way to do this.

For each character:
- Start at top.
- Go left on each ‘0’.
- Go right on each ‘1’.
- Stop at a node labeled with a decoded character.

```
A: 1110
C: 01000
O: 1101
T: 000
```
Coding and Decoding

See 22.py file for Python3 source code for performing coding, decoding, and tree-building for this data structure.