Outline

- Huffman trees
- Huffman coding
- implementation of Huffman coding
- review of lambda expressions (from ICS 111)

Huffman Trees

- trees seen so far:
  - binary trees
  - binary search trees
- in these trees, each node has **data** and zero, one, or two **children**
- Huffman trees are trees where each node has zero or two children (a *full* binary tree), and only leaf nodes have data
- the data in each leaf node is unique

Huffman coding

- suppose you wanted to compress a text file, that is, represent it using the least possible number of bits
- suppose also you want to use a unique string of bits for each character -- that is, we are not going to encode entire words
- it would be good to use fewer bits for characters that occur frequently, and as many bits as needed for other characters
- Huffman coding is an algorithm for assigning variable-length bitstrings to symbols so as to minimize the length of the encoded string
- The encoding length is minimized by using shorter bit strings to encode more commonly-used characters, and longer bit strings to encode less frequently-used characters

Huffman coding example

- for example, in the string "hello world", I could use the following encoding:

  h 11000

  e 11001

  l 0

  o 10

  ' ' (space) 11010

  w 11011

  r 1110

  d 1111

- in-class exercise: what does the bit string (25 bits) "1100 0100 1111 1101 0111 1101 0111 0" represent?
- note that the above string takes 45 bits if each character is represented with a constant 5 bits/character (which is the smallest number of bits to represent all 26 characters and space)
- so Huffman coding can save space!

algorithm for building a Huffman coding tree

1. make a list of all symbols with their frequencies
2. sort the list so symbols with lower frequency are in front
3. if the list only has one element, the element is the root of the tree and we are done
4. remove the first two elements from the list and put them as subtrees of a new node – a new binary tree
5. add the frequencies of the two subtrees to give the frequency of this binary tree
6. insert this tree in the right place in the sorted list
7. return to step 3

- in-class exercise: do this for the characters in “hello world”
  - 3 L, 2 O, and one each of H, E, space, W, R, and D

Using a Huffman coding tree to encode a string of characters

- to encode
  - find the leaf with the next character to encode
  - starting from the root to this leaf, place a 0 bit for each time you take the left subtree, and a 1 bit for each time you take the right subtree
- in-class exercise: use this algorithm and the tree from the last exercise to encode the string "wow"
- can do the above search once for each character, put the results in a table, then use the table to do the actual encoding quickly

Using a Huffman coding tree to decode a string of bits

- to decode:
  - use each bit to go left (0 bit) or right (1 bit) in the tree
  - if you have reached a leaf, put this character into the result string, then
  - start at the root of the tree with the next bit
- in-class exercise: use this algorithm and the above tree to decode as many bits as possible of the bit string "1100 0100 1101 0111 1101 0111 0"

compression using Huffman coding

- an encoding based on letter frequencies in one string (or in a large sample) can be used for encoding many different strings
- if so, a single copy of the table (tree) can be kept, and Huffman coding is guaranteed to do no worse than fixed-length encoding as long as all the strings have the same character frequencies
- otherwise, a separate table (tree) is needed for each compression, and the table has to be included in the count of bytes to be stored or transmitted after compression
- in such cases, Huffman coding might actually give a somewhat larger size than the original
- in practice, even including the table, Huffman coding is usually worthwhile for sufficiently long strings in natural languages, which have lots of redundancy and different letter frequencies

an implementation for Huffman trees data structures

- many possible implementations, this is one (textbook, chapter 6.1 and 6.7)
- use several data structures:
  - a priority queue to hold the sorted data
  - each priority queue element refers to a tree node
  - an interior tree node has no value, but has two children
  - a leaf node has a value, but has no children
  - each node has a frequency of occurrence, which is used as a priority in the queue (low priority returned first)

an implementation for Huffman trees algorithm

- begin by computing the frequency of each value
  - perhaps by using a hash table -- explained later
- once the frequency of values is known, insert each value into the priority queue, using its frequency as a priority
- remove the front two elements from the queue, create an interior node to refer to these two elements, and insert the interior node back into the queue with, as priority, the sum of the priorities of the two nodes
- once there is only one element left in the queue, this is the huffman tree
- a further step would be to build an encoding table from the tree
- in-class exercise: do this given the string "there is no place like home"

lambda expressions

- a lambda expression is an unnamed function
  - that can be assigned to a variable
  - or passed as a parameter
- so another view is that lambda expressions allow function variables
- some languages make heavy use of lambda expressions: for example, Javascript and LISP
- older versions of Java (before Java 8) did not have lambda expressions

lambda expressions in Java

- a lambda expression in Java has 3 parts:
  - the interface definition, defining the type
  - the lambda expression, defining the computation
  - the application of the lambda expression

lambda expressions example

- `Comparator<T>` defines `int compare(T t1, T t2)`
- `Comparator<T> f = (a, b) - > {`

```
    if (a.compareTo(b) > 0) return 1;
    if (a.compareTo(b) == 0)return 0;
    return -1;
  }
```

- `int comparison=f(value, a[middle]);`
- note: `f` doesn't have to be called `compare`

Defining the interface

- any interface that declares exactly one method is a functional interface, and can be used to declare a lambda expression
- common examples include:
  - `Comparator<T>` defines `int compare(T a,T b)`
  - `Function<T,R>` defines `R apply(T t)`
  - `Predicate<T>` defines `boolean test (T t)`
- anyone can use these types
- or you may define your own

Defining the computation

- a lambda expression has two parts, joined by the arrow: ->
- the first part is the parameter list
  - types may be omitted if Java can figure them out
  - parentheses may be omitted if only one parameter
  - or use () to indicate no parameters
- the second part is the expression, enclosed in {}
  - the parameter values may be used in the expression
  - the expression may also use local variables of the enclosing method – this is called a *closure*

applying lambda expressions

- a lambda expression defined by interface I
- simply call the only method of I
- example:
- ```
  Function<String, Integer> f =
        s - > { return s.length(); } ;
  ```
- ```
  Integer x = f.apply(“hello world”)
  + 2;
  ```