- intro to binary trees
- binary search trees
- traversals
- example application

Next week and homework

- next week: begin implementation of binary trees
- homework by next week:
  - (there are no example programs this week)
  - read Horstmann, sections 17.3 – 17.4

Lab

- lab assigned in ‘Implementation of linked lists’. See Canvas for due date

Review last week

- finished consideration of lists
  - full, generic implementation
  - circular linked lists
  - recursion and linked lists
- did some recursive method exercises, to get us to start thinking recursively

Introduction to this week

- introduce the next standard data structure
  - concepts, terminology, ideas this week
  - implementations begin next week

Intro to binary trees
Objective: introduce main ideas, definitions, terminology

Binary tree definition is recursive

- a binary tree "is either empty or contains a single node (called the root), which may point to left and right subtrees which are binary trees"
  - e.g., using letters to identify nodes:

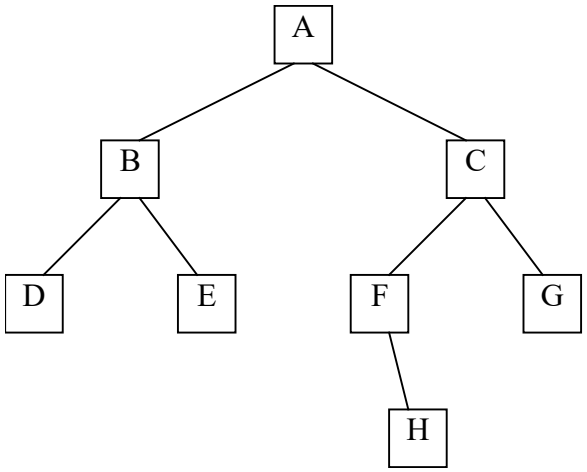

Figure 1 a binary tree

  - see that it's a tree, but drawn upside down

- a recursive definition, we use recursion to process trees
  - show root – is A here
  - show left subtree – is a binary tree with B as root
  - show right subtree – binary tree with C as root

Tree terminology

- a node is "each thing in a tree"
  - is used to store our information
  - implemented in Java as a class with the information and two references for the child nodes

- each node may have a left child and a right child, e.g.
  - F is left child of C
  - C is right child of A
- siblings are “nodes at the same level with the same parent”, e.g.
  - D and E; B and C
- leaves are “nodes without children”, e.g.
  - E; H
- to be a tree, a node (other than root) must have only a single parent node, e.g.
  - F is the parent of H
  - A is the parent of B
- an ancestor of a node is either the "parent of the node, or a parent of an ancestor", e.g.
  - complete set of ancestors of H are: F, C, A
- trees in general can have nodes with any number of children
  - binary tree – nodes have max two children
  - so is a special case of general trees
- some more examples of binary trees:

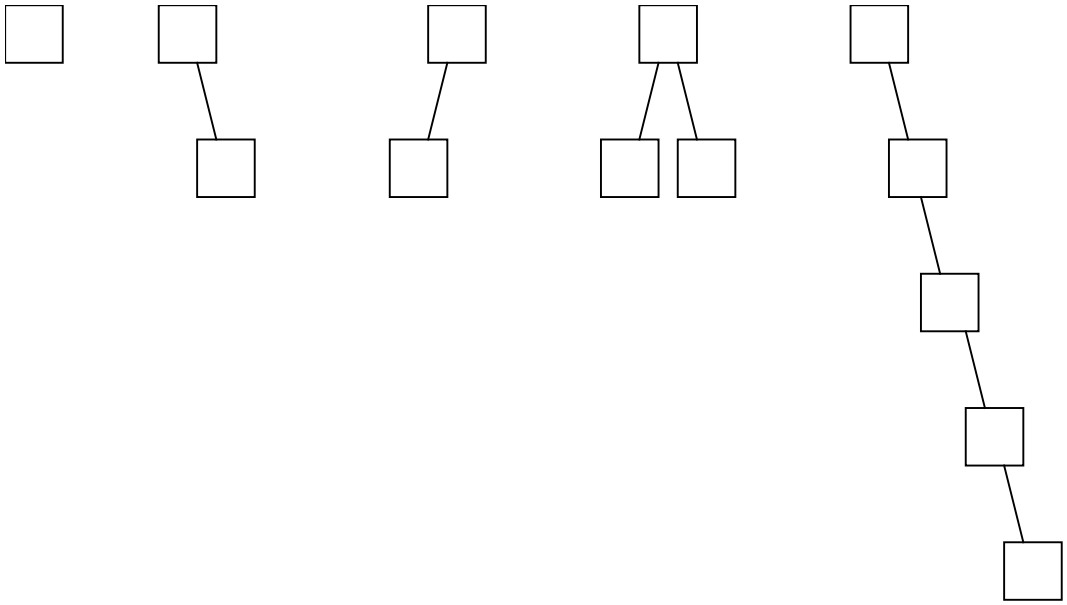

Figure 2 some more examples of binary trees

  - this last is a special case called a degenerate tree…
  - …has properties of a list, not a binary tree

- some examples that are not binary trees:

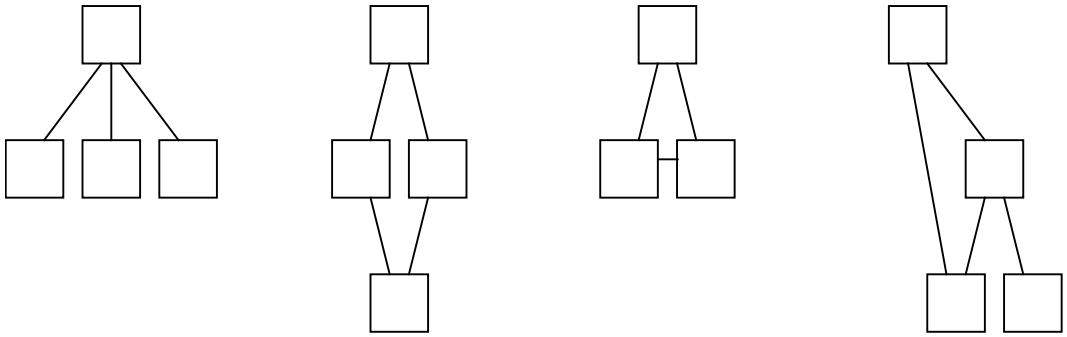

Figure 3 some examples that are not binary trees

  - a node must have only one parent

- the level of a node can be defined recursively:

  0 if node is root
  otherwise level of parent + 1

  - e.g.

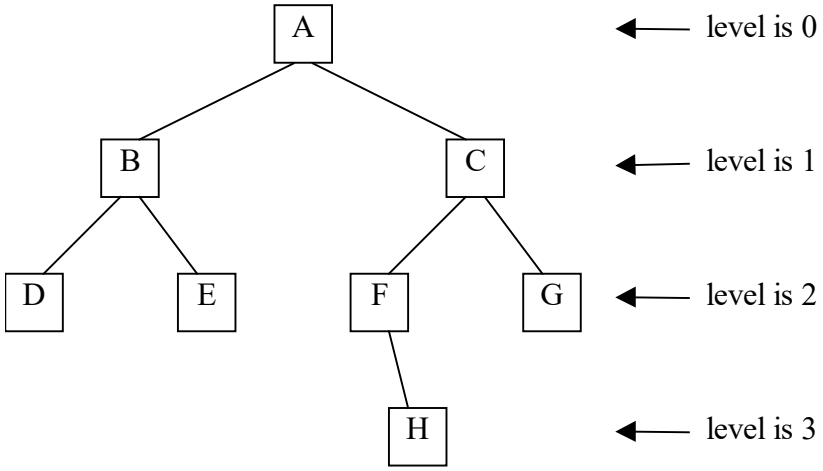

Figure 4 the levels of a binary tree

- the depth or level (N) of an entire tree is the "level of the node with the greatest level"
  - e.g. is 3 above

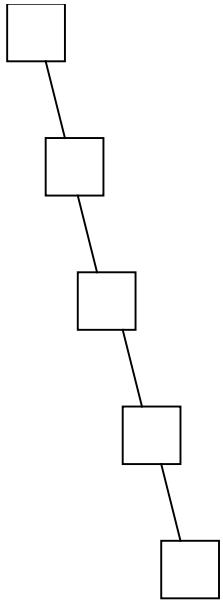

Figure 5 the depth or level (N) of an entire tree

  - 4 here

- a complete binary tree of level N is one in which "every node of level N is a leaf and every node at level < N has left and right subtrees"
  - e.g. draw complete binary trees for a few levels:

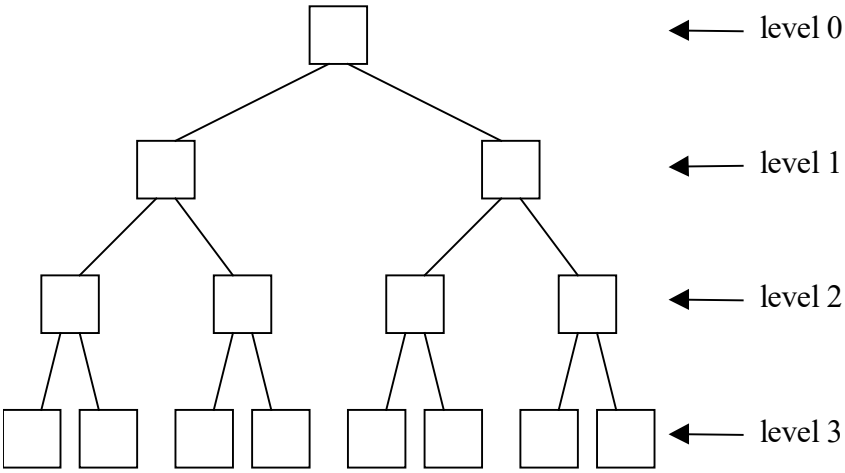

Figure 6 a complete binary tree of level 3

Binary search trees
Objective: introduce this special case of binary trees, used to do binary search. Is the most common use of binary trees

Binary search is faster than linear, sequential search

- sequential search is slow, but works for unsorted data e.g.

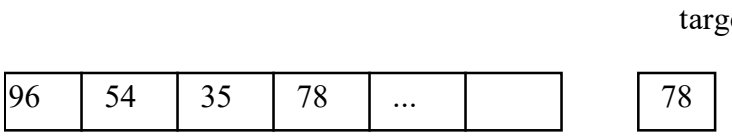

Figure 7 sequential search in an array of unsorted data

  - the target is compared against each element in turn until there's a match or no more items

- we measure the speed of algorithms in terms of the number of elements, N. Here:
  - best case: 1
  - worst case: N
  - average case: N / 2
- binary search is faster, but works only for sorted data e.g.

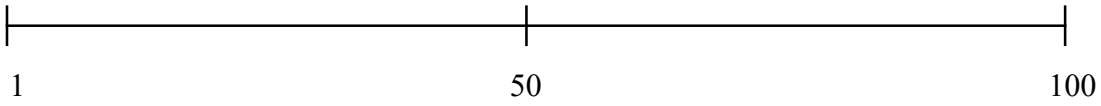

Figure 8 binary search requires sorted data

  - how do you search for some target between 1 and 100 here?
  - always guess the midpoint. So this eliminates half of the search space per comparison

- if we have N elements,
  - worst case is actually given by: $\log_2 N$
- as N increases, binary becomes much faster than sequential search:

N	sequential	binary

	(average case)	(worst case)
100	50	7
200	100	?
1 000	500	10
2 000	1 000	?
1 000 000	500 000	20
2 000 000	1 000 000	?
N	N/2	$\log_2 N$

  - see for sequential search that as N doubles, so does the number of comparisons
  - think about what happens for binary search…
  - …as N doubles, we have to do only one extra comparison!
  - i.e. from 7 to 8, or 10 to 11, and 20 to 21 above

- the advantage of binary search over linear search is shown clearly if we graph search time t against N:

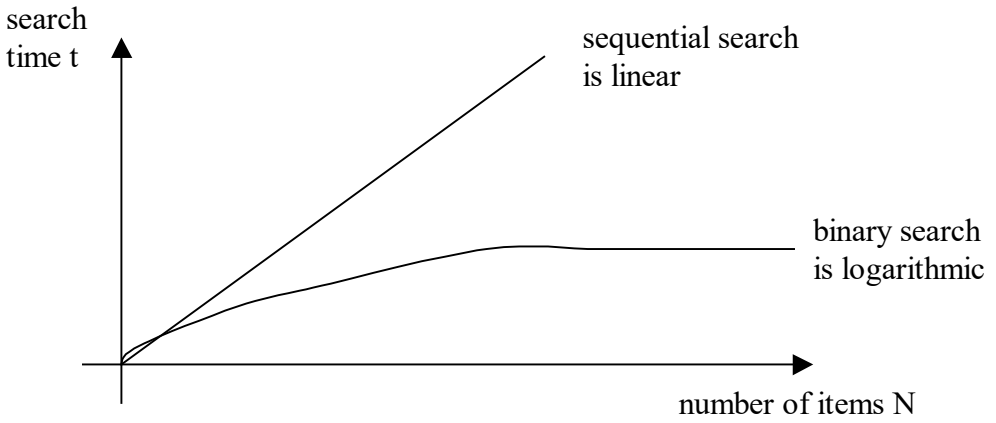

Figure 9 comparing search times as N increases

- so we often use binary trees when we need fast search of data

Binary search trees

- we implement binary search in a special case of binary tree, called a binary search tree

- "a binary tree in which each parent node is greater than its left child and less than its right child"

- so now each node requires a value, which can be compared. E.g., using simple integer values:

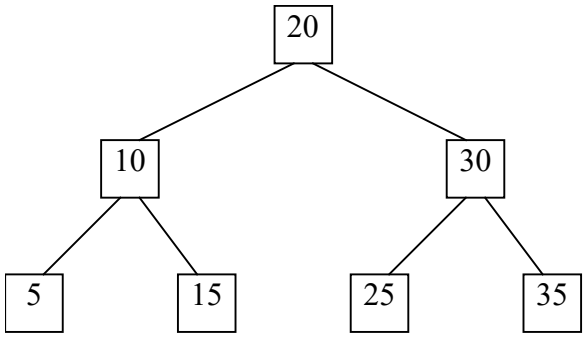

Figure 10 a binary search tree

How to search

- how to search for a number in a binary search tree?

  - question at each level – "is target less than or greater than current node"

  - e.g. search for 25

  - requires 3 comparisons here. Compares 25 with: 20, 30, 25

- so for this special case of a binary search tree, each comparison reduces search space by ½

  - gives search time proportional to $\log_2 N$

How to build

- binary search is most efficient in a complete binary search tree

  - we build bsts from a stream of data

  - e.g. build bst from following data stream:

    6 4 8 7 5 9 3

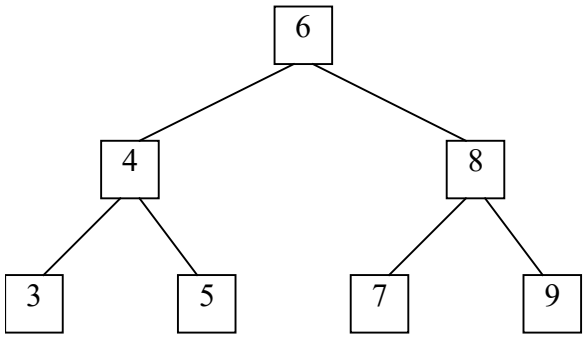

Figure 11 bst built from a stream of data

- this data stream was deliberately designed to give the best case of a complete binary search tree!
- search time for this best case: $\log_2 N$

- build worst case binary search tree from this next data stream:

3 4 5 6 7 8 9

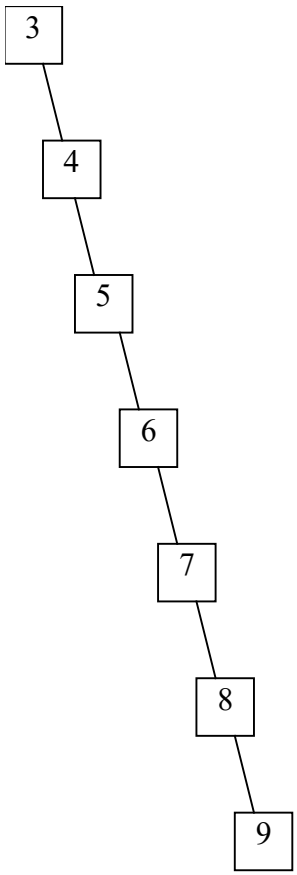

Figure 12 a worst case binary search tree

  - this deliberate worst case gives a “degenerate tree”
  - here search degenerates to sequential search
  - search time: N

- e.g. search for 9 in both trees:
  - 3 looks vs. 7 comparisons

Random data gives a balanced binary search tree

- a balanced binary search tree is “neither complete nor degenerate”
  - is somewhere between the two extremes
  - gives acceptable search times
- raw data usually gives a balanced tree, so we can generally use it e.g.

6 8 3 4 9 7 1 2 5

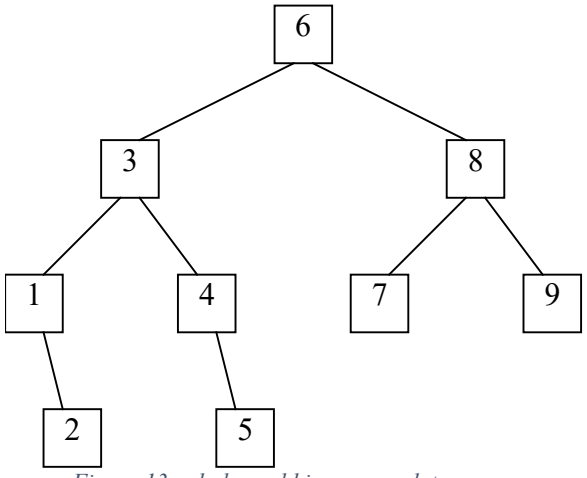

Figure 13 a balanced binary search tree

  - not complete, but close enough
  - BTW, the AVL algorithms (named for their creators Adeleson-Velsky and Landis) build and maintain close to complete trees from any data

Building a bst sorts the data

- building the bst imposes an order on the arriving data i.e. sorts it
  - process is not too difficult or expensive
  - as we see next, we can retrieve the data from a bst in different sorted orders

Summary

- bsts are the most common application of binary trees
- will see more examples and do an implementation next week
- will use bsts in one of the next labs

Traversals
Objective: traversals through a binary search tree sort the data in several useful ways

- a traversal of a tree "visits each node in turn"
  - there are three different traversal orders
  - each has a different use. Will see one use when we complete a binary search tree exercise here

Preorder traveral

- the preorder traversal is defined as:

  visit root
  visit left subtree in preorder
  visit right subtree in preorder

- when we 'visit root' in a traversal, this means that we access that node's information
  - here we'll just print the node's label, to show the order of the traversal
- e.g.

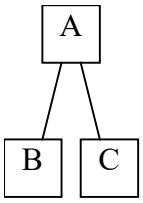

Figure 14 an example binary search tree to illustrate traversals

  - gives: A B C

Inorder traveral

- inorder traversal is:

  visit left subtree in inorder
  visit root
  visit right subtree in inorder

- e.g.

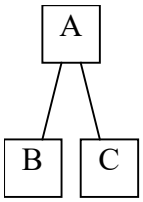

Figure 15 an example binary search tree to illustrate traversals

- gives: B A C

Postorder traversal

- postorder traversal is:

  visit left subtree in postorder
  visit right subtree in postorder
  visit root

- e.g.

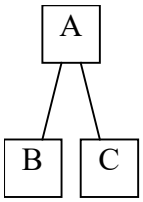

Figure 16 an example binary search tree to illustrate traversals

- gives: B C A

- HINT: see from these that the traversal terminology is with respect to the root node:
  - preorder means visit root first – root left right
  - inorder means visit root in the middle – left root right
  - postorder means visit root last – left right root

Some example traversals

- let's do some examples:

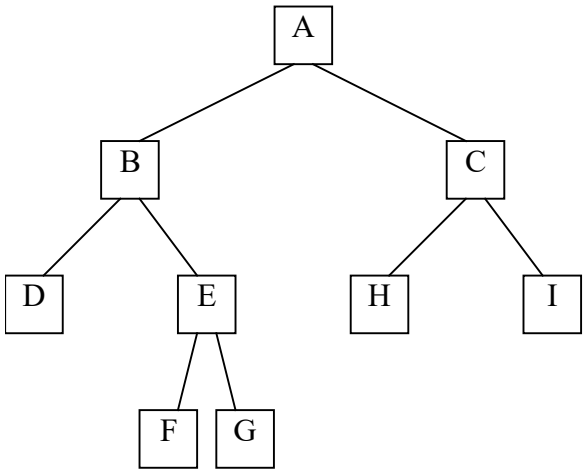

Figure 17 an example binary search tree to illustrate traversals

- work through preorder traversal
  - is "root left right"
  - gives:

    A B D E F G C H I

- inorder
  - is "left root right":

    D B F E G A H C I

- postorder
  - "left right root":

    D F G E B H I C A

Binary search tree exercise

- do a final bst exercise on a piece of paper:
  - build the binary search tree from the following input stream, then write the result of an inorder traversal

    2 3 8 4 6 9 1 5 7

- will review this exercise and the importance of the inorder traversal next week

Summary

- you should see from this exercise, when to use an inorder traversal
- will see the usefulness of the other traversals next week

Example application
Objective: binary trees have many uses. Let's look at data compression

Huffman encoding

- data compression reduces the number of bytes required to store information:

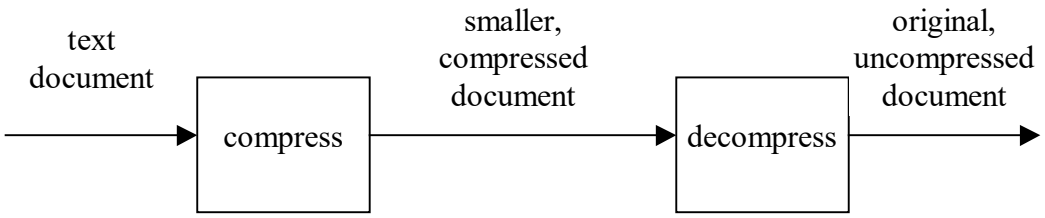

Figure 18 data compression with a document

- in uncompressed text files:
  - most common character E is stored in 8 bits
  - least Q 8
  - so average is 8 bits / character
- the Huffman algorithm uses fewer bits for the most common characters
  - so the average will be < 8 bits / character
  - meaning the file is compressed
  - (our example will give 2.13 bits / char)

Use a binary tree to generate a Huffman encoding

- 3 steps. First, statistically analyze text (or English in general) to find frequency of occurrence of each different character
  - as an example, will look here at a simple language with only 6 characters! Sort by probability of occurrence:

char	p(char)
C1	0.50
C2	0.15
C3	0.12
C4	0.10
C5	0.09

C6	0.04

	1.00

- now build a binary Huffman tree from this table
  - successively combine the two letters with the lowest probabilities, to build the tree from the leaves to the root:

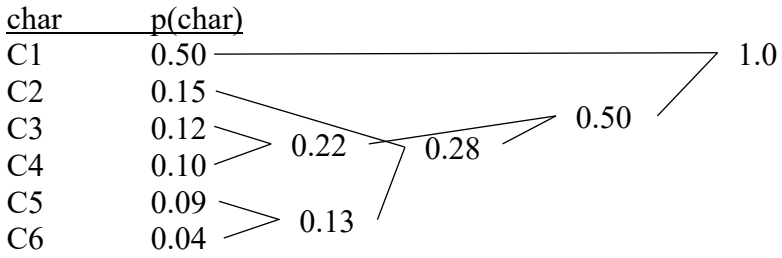

Figure 19 building a binary Huffman tree from the table

  - showing the resulting tree more clearly:

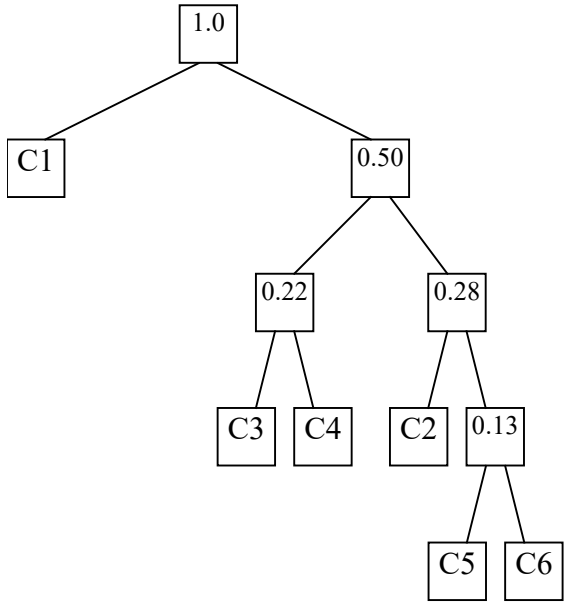

Figure 20 showing the Huffman tree

- finally, assign codes to characters, starting from the root:
  - use 0 to go left
  - 1 to go right

– gives:

char	p(char)	Huffman encoding
C1	0.50	0
C2	0.15	110
C3	0.12	100
C4	0.10	101
C5	0.09	1110
C6	0.04	1111

– see that the most frequently occurring characters get the shortest Huffman codes

Example

- use this Huffman encoding to compress this message:

  C2 C3 C5 C1

  – encodes as:

  110 100 1110 0

  – (a stream of bits without breaks around characters)

- check, that it decodes to:

  C2 C3 C5 C1

  – because each Huffman code can be translated unambiguously – no code is a prefix of any other

- compression saving here:

  – no compression: 4 chars * 8 bits/char = 32 bits

  – Huffman: 11 bits

- average bits/character

  – no compression: 8 bits/character

  – Huffman: is number of bits * probability, for every character
  = 1 * 0.5 + 3 * 0.15 + …
  gives 2.13 bits/character

Summary

- example of use of binary trees

- BTW, Huffman is obsolete, superseded by more efficient, public domain Lempel-Ziv-Welch (LZW) compression

Next week and homework

- next week: begin implementation of binary trees
- homework by next week:
  - (there are no example programs this week)
  - read Horstmann, sections 17.3 – 17.4

Lab

- lab assigned in ‘Implementation of linked lists’. See Canvas for due date