Data Structures & Algorithms 2

Topic 2 – Data Compression

Uses for Binary Trees

- The kind of tree where you go left or right depending on a key value is called a binary search tree
- There are many other uses for trees other than for storing and searching
- We will consider a binary tree used to compress data called the Huffman coding tree
- Huffman Codes are used in nearly every application that involves the compression and transmission of digital data, such as fax machines, modems, computer networks, and high-definition television (HDTV)

Compression

- Compression is very important in multimedia for reducing the amount of space a file takes up
- You can have lossy or lossless compression
- Lossy compression works by chopping off some of the signal that you don't need – e.g. jpg, mp3
- Lossless compression allows you to uncompress exactly what you had before – e.g. winZip
- Obviously if we're compressing text or a computer program, we want it to remain perfect

Binary Code

- Everything on a computer is stored in terms of 1s and 0s (on and off)
- ASCII, the international coding standard uses a 7 bit code
- Every letter is represented by 7 bits
- Such encodings are called
  - fixed-length or
  - block codes
- They are attractive because the encoding and decoding is extremely simple

ASCII

- For example: the sentence
  The cat sat on the mat

  is encoded in ASCII as

  1010100 110100 011001 0101 .....
- Note that the spaces are there simply to improve readability ... they don't appear in the encoded version.

Example

- The following bit string is an ASCII encoded message:

  10001001100101110001111011111100100110100
  0111011101100111010000011010011110011010
  0000110010111000011110011111001

Example

- And we can decode it by chopping it into smaller strings each of 7 bits in length and by replacing the bit strings with their corresponding characters:

  1000100(D)1100101(e)1100011(c)1101111(o)1100100(d)1101001(i)1101110(n)1100111(g)0100000()1101001(i)1110011(s)0100000()1100101(e)1100001(a)1110011(s)1111001(y)

Problem?

- Block codes can be inefficient:
  n symbols produce (n x b) bits with a block code of length b
- For example,
  - if n = 100,000 (the number of characters in a typical 200-page book)
  - b = 7 (e.g. 7-bit ASCII code)
  - then the characters are encoded as 700,000 bits
- Can we do better than this?

Binary Code

- In binary code, each number is twice the value of the number after it (as opposed to ordinary decimal numbers where each number is 10 times greater)
- For example, the number 1001 in decimal means
  - 1 thousand 0 hundreds 0 tens 1 unit = a total of one thousand and one
- The number 1001 in binary means
  - 1 eight 0 fours 0 twos 1 unit = a total of nine
- Therefore having 7 binary bits in an ASCII code allows us to store up to 128 different characters
  - 1111111 in binary = 127 in decimal

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

David Huffman

- In 1951, David Huffman and his MIT information theory classmates were given the choice of an assignment or a final exam
- The professor assigned the problem of finding the most efficient possible binary code
- Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the exam when he hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient
- In doing so, the student outdid his professor, who had worked with information theory inventor Shannon to develop a similar code and failed

Saving Space

- We don't have to use exactly 7 bits for every character
- We can encode the characters with a different number of bits, depending on their frequency of occurrence
- Use fewer bits for the more frequent characters
- Use more bits for the less frequent characters
- Such a code is called a variable-length code
- Works out shorter

Problem

- First problem with variable length codes:
  - when scanning an encoded text from left to right (decoding it)
  - How do we know when one codeword finishes and another starts?

- We require each codeword not be a prefix of any other codeword
  - We couldn't have two codes like 10 for A and 1001 for B

Prefix Codes

- Codes that satisfy the prefix property are called prefix codes
- Prefix codes are important because
  - we can uniquely decode an encoded text with a left-to-right scan of the encoded text
  - it doesn't matter that codes have different length because we know where one code ends and the next starts
  - binary tree can be used to ensure that codes are prefix codes

Example

- Figure out this using the following prefix codes

01 101 110 00 101 111 100 101 01 01 101 110 00

  - N 100
  - E 00
  - B 110
  - R 111
  - O 101
  - T 01

Non-prefix Codes

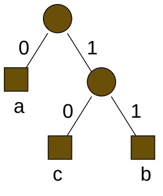
- Now can you get this one?

100111

  - E 01
  - Y 10
  - R 101
  - O 111
  - S 11
  - N 100
- The problem here is that these letters aren't all leaves in the tree!

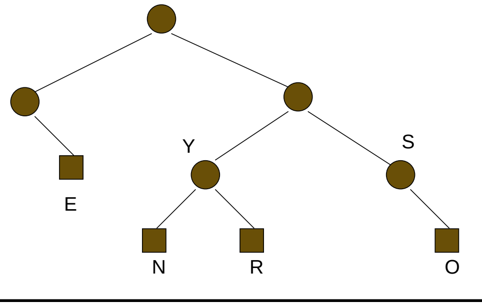
Example

- Nodes with the same prefix must have the same ancestors
- But only leaf nodes are used – only one unique path to each



a 0
b 11
c 10

Problem Tree



Using a Binary Tree

Makes it easier when you use a binary tree:

1. Read the encoded message bit by bit
2. Start at the root
3. if the bit is a 0, move left
4. if the bit is a 1, move right
5. if the node is a leaf, output the corresponding symbol and begin again at the root

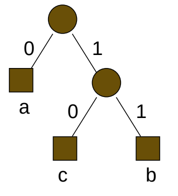
Example

- Encoded message:

  0 0 1 1 1 0 0

- Decoded message:

  A A B C A

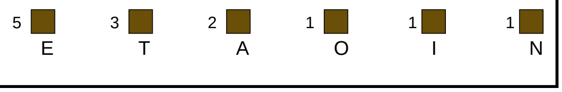


Huffman's Algorithm

- Huffman's idea was to use the leaves high up in the tree (fewer bits) for the common characters
- Use the lower leaves (more bits) for the less frequent characters
- Problem – letters might have different frequencies in different documents (e.g. semicolon is common in java but not elsewhere)
- How do we build a tree that reflects the most efficient frequencies?

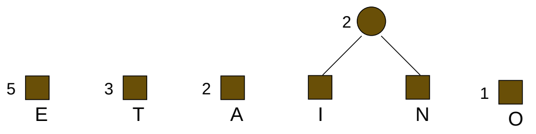
Creating the Tree

- Step 1
  - First of all, create a forest of trees for all of the letters in the piece of text
  - Each letter is a separate tree, therefore all the nodes are roots
  - Each node is associated with its probability in the document to be compressed
  - Using letter frequency works in exactly the same way
  - Place these into a priority queue structure



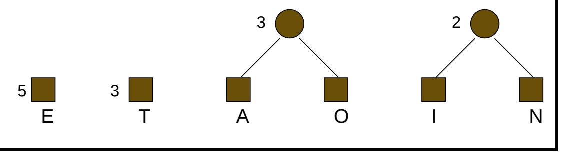
Creating the Tree

- Step 2
  - Choose the two binary trees, B1 and B2, that have the lowest frequency – if there's a tie, choose either
  - Create a new root node with B1 and B2 as its children and with frequency equal to the sum of these two frequencies

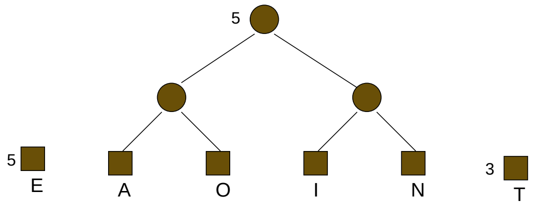


Huffman's Algorithm

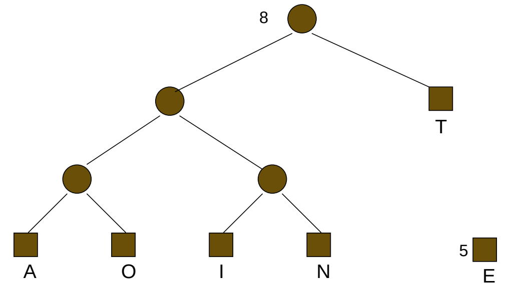
- Step 3
  - Each time two trees are combined, place them back in the priority queue according to frequency
  - Repeat this process until just one tree is left

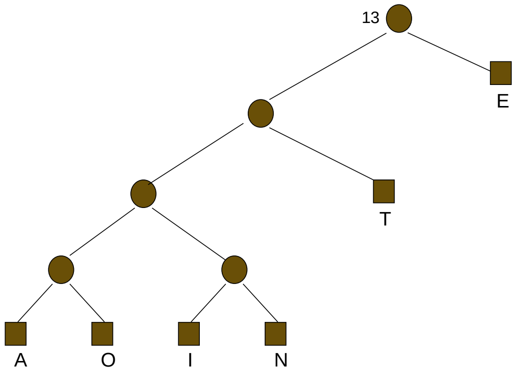


Huffman's Algorithm



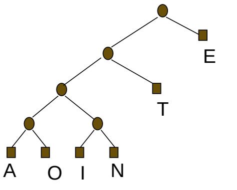
Huffman's Algorithm





Huffman's Algorithm

- The final prefix code is:
  - A 0000
  - O 0001
  - I 0010
  - N 0011
  - T 01
  - E 1

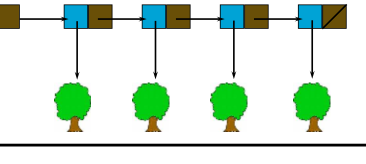


Compression ratio

- When compressing ASCII, the compression ratio is the ratio between the number of bits using ASCII versus Huffman codes
- If we had 13 letters, then this uses 13 x 7 = 91 bits in ASCII
- Using the Huffman code we have 31 bits:
  - 5 x 1 (E)
  - 3 x 2 (T)
  - 5 x 4 (A, O, I and N)
- The text has been compressed to 34% of its original size!

Implementation

- We need the following classes
  - A node class
  - A tree class
  - A main Huffman algorithm class
- We need a priority queue to manage the forest of trees



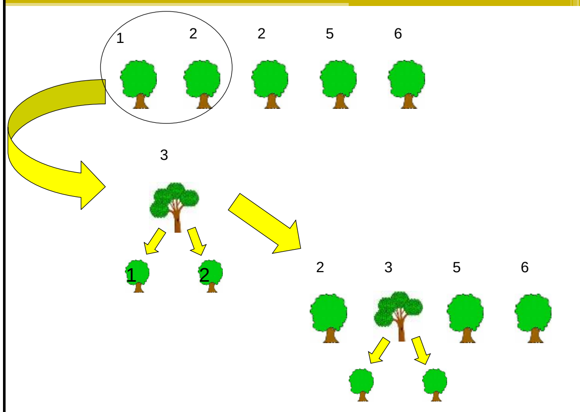
Node & Tree

- Each Node object has
  - `Node leftChild;`
  - `Node rightChild;`
  - `char letter;`

- Each Tree object has
  - `Node root;`
  - `int weighting;`

Huffman Algorithm

1. Create a new PriorityQueue
2. Create a new Tree for every letter
3. Add the Trees into the PriorityQueue
4. Keep removing the two trees with the lowest frequency from the PriorityQueue
5. Join them together into a comboTree and stick the comboTree back in the PriorityQueue
6. Keep doing this until there is just one tree left in the PriorityQueue: the Huffman Tree



Final Stage

- Now we have our tree, time to code up our text
- We need a create a code table which matches each character with its Huffman code
- Stored as an array with a slot for every letter

A	010
B	1111
C	
D	01111
E	10
F	110

Priority Queue

- You wrote your own Priority Queue in CS210
  - You can use arrays
  - You can use linked lists

- The problem with the Priority Queue you created is that the type of object stored in it was hardcoded
  - Person objects
- Many useful data structures are included in Java as generic classes
  - You can use them with any type of object

Java Generics

- Generics are a facility of generic programming that was added to the Java programming language in 2004 as part of J2SE 5.0
- They allow a method to operate on objects of various types
- Before that, people had to take responsibility for casting objects from one type to another
- This often resulted in programs crashing during runtime

Example with no Generics

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0);
```

- No error is detected by the compiler
- Causes a runtime exception (java.lang.ClassCastException) when executing the third line of code

Example with Generics

```
List<String> v = new ArrayList<String>();
v.add("test");
Integer i = v.get(0); // (type error)
```

- This time we state what type of object ArrayList can hold
- A compile error is detected for the third line of code
- The programmer can fix it

Generic Priority Queue

```
import java.util.PriorityQueue;

PriorityQueue < Tree >  PQ = new PriorityQueue < Tree >() ;
```

- Check java.sun.com for method summary:
  - `add(Tree myTree)`
  - `peek()`
  - `poll()`
- A priority queue must order the objects inside itself
  - Objects stored in PriorityQueue must implement the Comparable interface
  - `public class Tree implements Comparable<Tree>{…`
  - This requires a compareTo( ) method to be specified

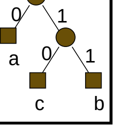
compareTo( ) interface

```
public int compareTo(Tree object){

//compare the cumulative frequencies of the tree
//must return -1,0, or 1

        if(frequency-object.frequency>0){
            return 1;
        }else if(frequency-object.frequency<0){
            return -1;
        }else{
            return 0;    //return 0 if they're the same
        }
}
```

Now we can use Java's Priority Queue class

Creating the Table

- How do we create this table of Huffman codes?
- Start at the root of the Huffman tree and traverse the full tree
- Remember the sequence of left and right choices
  - go left → add a "0" to the path
  - go right → add a "1" to the path
- Every time you arrive at a leaf node, the path you have followed is the Huffman code for that character
- Put this code into the code table at the appropriate index



Sending Messages

- If we send the Huffman encoded text to somebody on its own there's going to be a problem
- They need the code array in order to interpret the message
- Codes will vary depending on frequencies of letters in different documents
- Because we have to send this extra information, Huffman coding only yields a significant advantage when there is a lot of text encoded

Huffman Coding

- Huffman's Algorithm uses a technique called Greediness
- It uses local optimization to achieve a globally optimum solution
  - Build the code incrementally
  - Merge the two symbols that have the smallest probabilities into one new symbol
- Huffman Code is the optimal way in which single symbols can be transmitted one at a time
  - used everywhere from fax machines to modems
- Ever watched a fax machine? It goes very fast over the white space and slows down when it hits complicated patterns
  - 00 is used to represent the space symbol
  - 01 is used to represent a dot

Can we do better?

- Although it is the best symbol by symbol, you can still get better compression by representing patterns of symbols as a single chunk
- Letters do not occur randomly one after the other – the use of one letter affects the probability of the next
- For example
  - The letter g rarely follows the letter t
  - The letter u nearly always follows the letter q

- If we sample larger sequences, we can exploit these trends and obtain even more compression

Explore the Possibilities

- When you are doing your Huffman code, you can investigate if you can get better compression
- Try chunking two letters at a time or three letters at a time and build up the tree in the same way
- The only problem here is that the table starts to get very large
- It will still be worth it so long as the amount of information you want to send exceeds the size of the table

Compressionism

- Compressionism is the idea that intelligence and consciousness are related to the phenomenon of data compression
- The Hutter prize for machine intelligence is a test where programs must compress 100MB of Wikipedia text
- €500 euro is offered for every 1% improvement

Compressionism

- Schmidhuber's ambition is to build an optimal scientist better than himself so he can retire
- His idea is that compression progress explains subjective beauty, novelty, surprise, interestingness, attention, curiosity, creativity, art, science, music and jokes
- People want to extract patterns from their observations because data compression is linked to prediction