CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 18

April 15, 2014

Reminder – File Compression

- Last time: file compression by substituting short codes for characters.
- Frequent characters should get shorter codes.
- Prefix codes is a set of codes such that no code is the prefix of another code.
- This guarantees that each code can be matched to a character uniquely.

Huffman's Coding

Char	sp	nl	0	1	2	3	4	5	6	7	8	9
Freq.	30	20	10	7	6	5	4	3	3	3	2	2

4

Char	sp	nl	0	1	2	3	8	9	4	5	6	7
Freq.	30	20	10	7	6	5	2	2	4	3	3	3

4 6

Char	sp	nl	0	1	6	7	2	3	8	9	4	5
Freq.	30	20	10	7	3	3	6	5	2	2	4	3

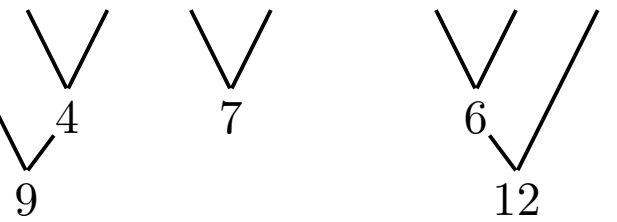
6 4 7

Char	sp	nl	0	4	5	1	6	7	2	3	8	9
Freq.	30	20	10	4	3	7	3	3	6	5	2	2

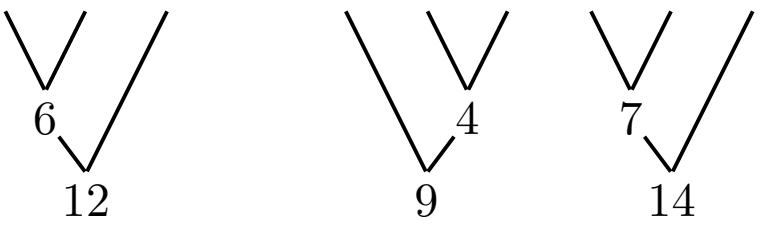
7 6 4 9

Huffman's Coding

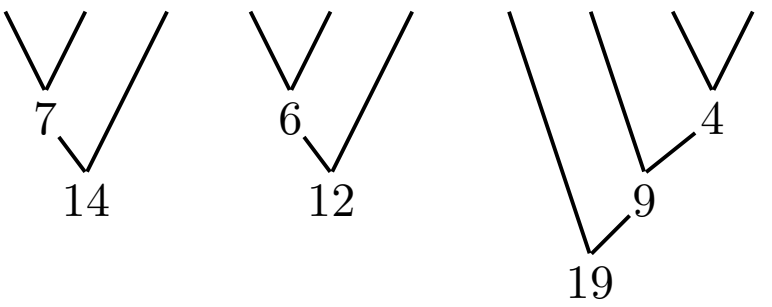
Char	sp	nl	0	3	8	9	4	5	1	6	7	2
Freq.	30	20	10	5	2	2	4	3	7	3	3	6



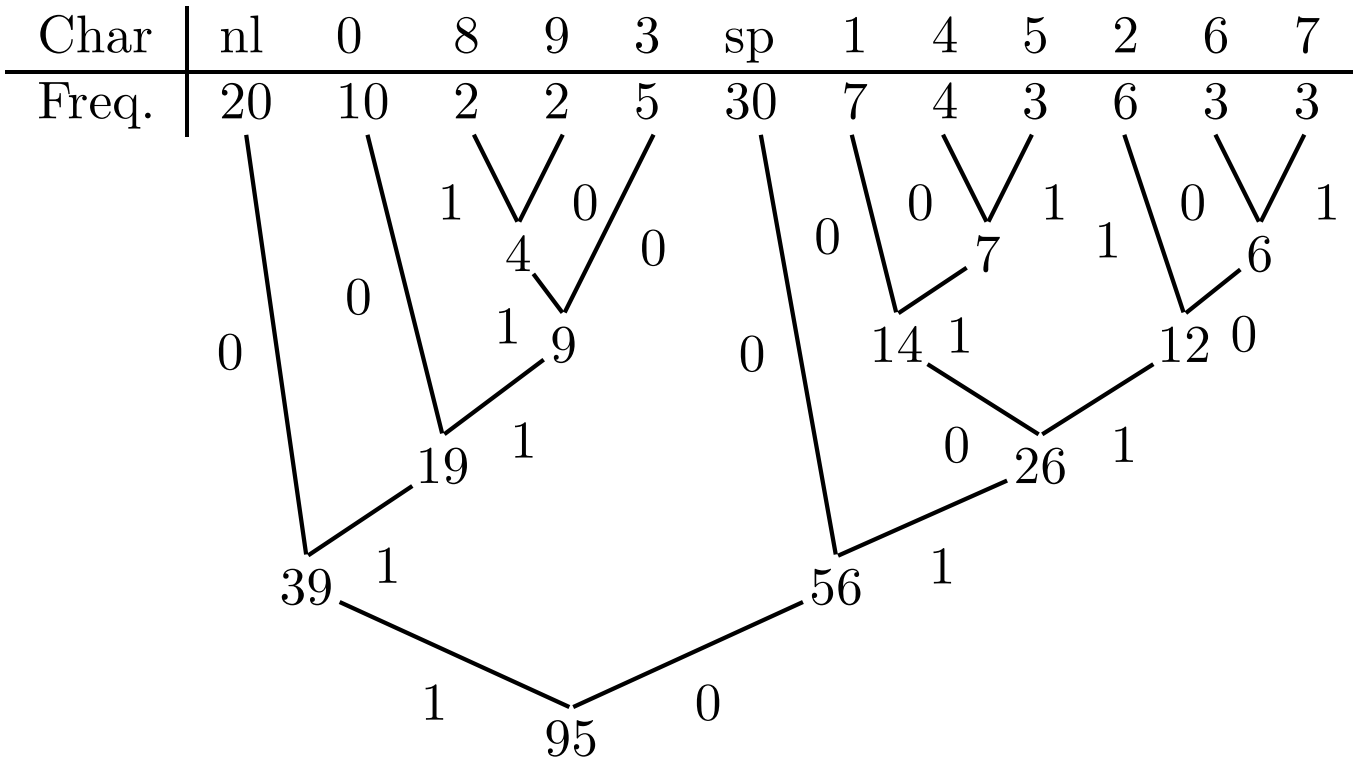
Char	sp	nl	6	7	2	0	3	8	9	4	5	1
Freq.	30	20	3	3	6	10	5	2	2	4	3	7



Char	sp	nl	4	5	1	6	7	2	0	3	8	9
Freq.	30	20	4	3	7	3	3	6	10	5	2	2



Huffman's Coding

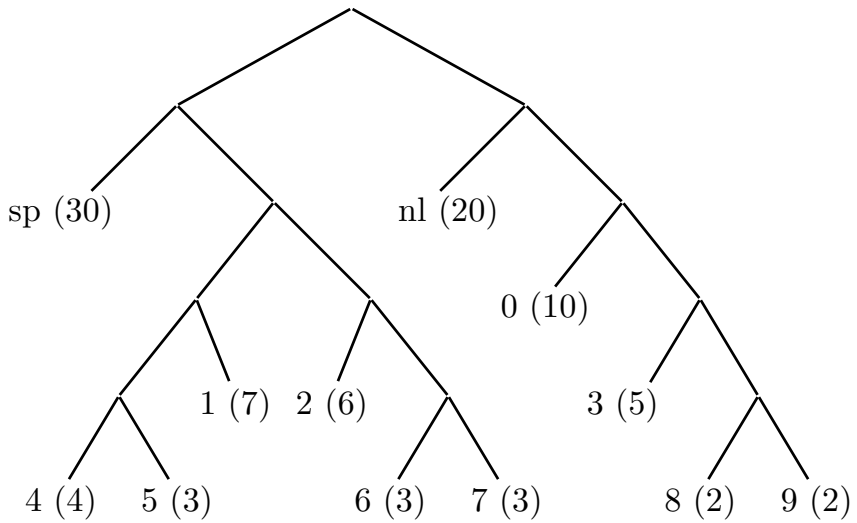


Convention to Assign code: larger weight = 0, smaller weight=1, random code for same weights

Huffman's Coding

char	code	freq.	total bits
sp	00	30	60
nl	10	20	40
0	110	10	30
1	0101	7	28
2	0110	6	24
3	1110	5	20
4	01000	4	20
5	01001	3	15
6	01110	3	15
7	01111	3	15
8	11110	2	10
9	11111	2	10

Total bits = 287



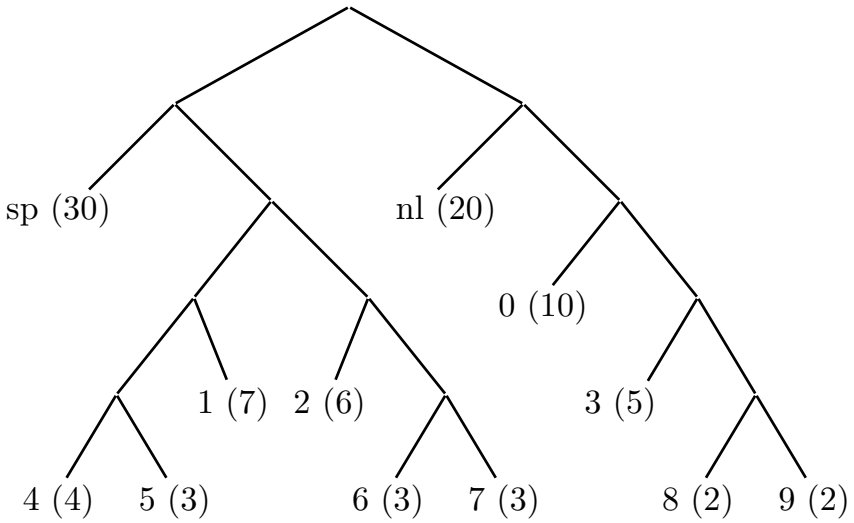
Why is Huffman's Algorithm Optimal?

- Consider the optimal coding scheme for n characters.
- It will have a longest code, corresponding to the lightest char.
- It will have at least two codes of this longest length, or we could shorten one (in other words – there are no nodes with one child, or we can replace the child by the parent and get a shorter tree).
- The two chars of lightest weight will be of this particular longest length.
- Thus an optimal coding scheme has its two lightest-weight chars with codes of the same length, the longest code length.

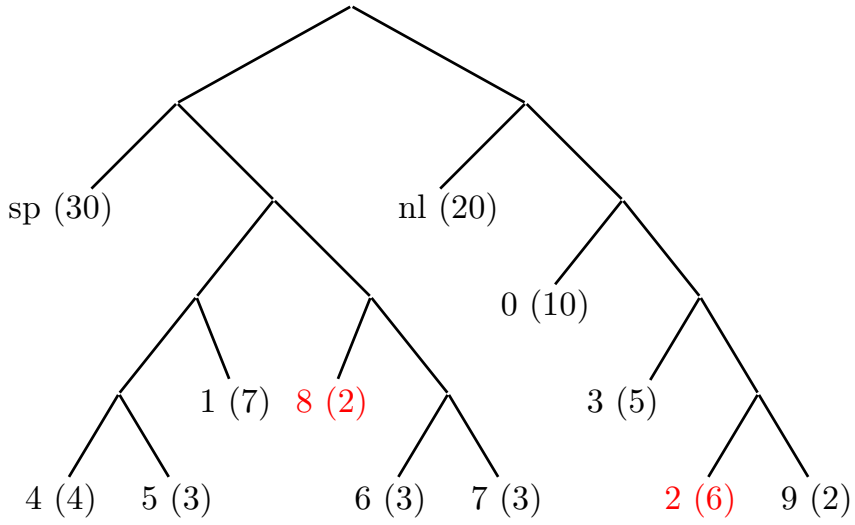
Why is Huffman's Algorithm Optimal?

- If there are more than two codes of this length, they are interchangeable, so get the two lightest-weight ones paired up to share all but the last bit of their codes.
- Then merge the two characters into one new imaginary char with summed weight.
- The optimal coding scheme for this new char set will yield the optimal coding scheme for the original set. (To actually show this step you need to write down the sum being minimized and see whats important in the minimization.)
- Similarly, we keep coalescing until the problem is trivial, and then expand back.

Why is Huffman's Algorithm Optimal?

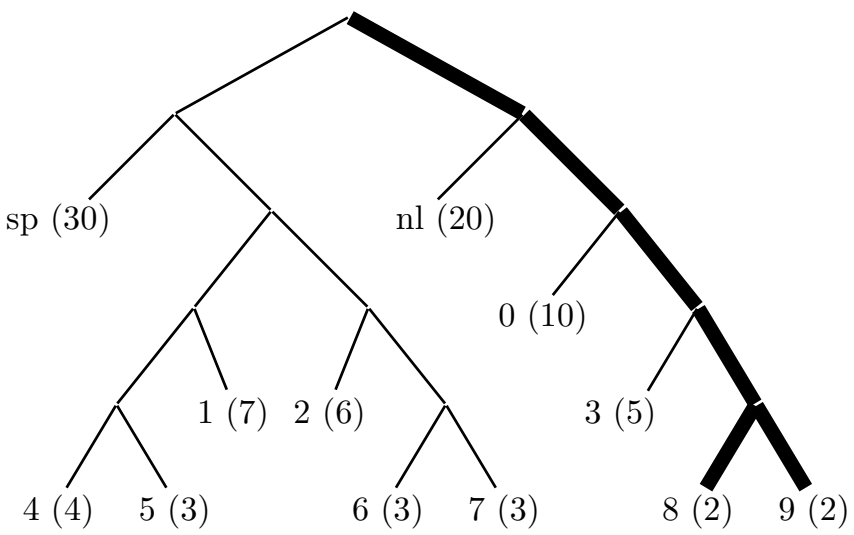


Huffman's tree

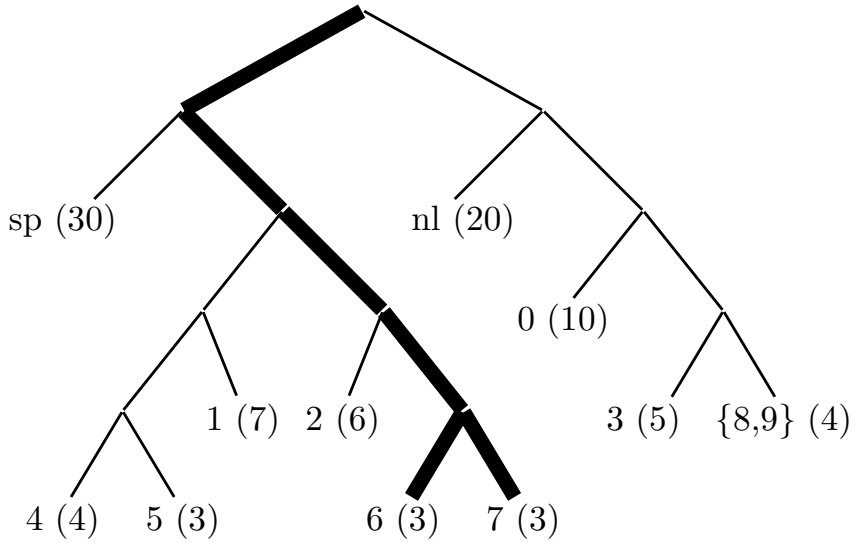


A slightly different tree

Why is Huffman's Algorithm Optimal?



Longest path before compression



Longest path after compression

Implementation of Huffman's Algorithm

- We can take a document, build a Huffman tree for its char frequencies, then compress its contents using the resulting codes.
- We send the compressed document and the char frequency table to the other end, where the same Huffman tree is built and used to decompress the compressed data.
- So there are three big parts to the implementation:
  1. Building a Huffman tree from the document, based on its char frequencies.
  2. Compressing the document using the Huffman tree.
  3. Decompressing the document using the Huffman tree.

Fast Decoding Using Array Lookup

char	code
sp	00
nl	10
0	110
1	0101
2	0110
3	1110
4	01000
5	01001
6	01110
7	01111
8	11110
9	11111

Sort by highest bit value

char	code
sp	00
4	01000
5	01001
1	0101
2	0110
6	01110
7	01111
nl	10
0	110
3	1110
8	11110
9	11111

Fast Decoding Using Array Lookup

char	code
sp	00xxxxxx
4	01000xxx
5	01001xxx
1	0101xxxx
2	0110xxxx
6	01110xxx
7	01111xxx
nl	10xxxxxx
0	110xxxxx
3	1110xxxx
8	11110xxx
9	11111xxx

Set up an array of 256 bytes, one for each bit pattern, showing char being decoded

char	bit pattern
sp	00000000
sp	...
sp	00111111
4	01000000
4	...
4	01000111
5	01001000
5	...
5	01001111
1	01010000
1	...
...	...

Fast Decoding Using Array Lookup

- Suppose this is an array "byte decode[] = new byte [256];", filled in as above.
- Then if x is the first byte (8 bits) of the bit string, decode[x] is the first ASCII char in the string.
- Once you know the char, it's easy to look up its bit length (in another array, or squeezed into the values of the first array) and shift off the decoded bits, ready to interpret the first char of the remaining bits.

Fast Decoding Using Array Lookup

- This approach is really fast! In today's processors, you can do any processing in the CPU in a few nanoseconds.
- The cache can hold the lookup table, so the lookups here are done without memory reference once it gets going.
- The speed of the whole process will be limited by the access to the coded data in memory and writing the results in memory.

Character Counting Class

Count the frequency of characters in a file:

```
class CharCounter
{
    public CharCounter( )
      { }

    public CharCounter( InputStream input ) throws IOException
    {
        int ch;
        while( ( ch = input.read( ) ) != -1 )
            theCounts[ ch ]++;
    }

    public int getCount( int ch )
      { return theCounts[ ch & 0xff ]; }

    public void setCount( int ch, int count )
      { theCounts[ ch & 0xff ] = count; }

    private int [ ] theCounts = new int[ BitUtils.DIFF_BYTES ];
}
```

Huffman Tree Class

Tree is maintained as a collection of nodes:

```
// Basic node in a Huffman coding tree.
class HuffNode implements Comparable<HuffNode>
{
    public int value;
    public int weight;

    public int compareTo( HuffNode rhs )
    {
        return weight - rhs.weight;
    }

    HuffNode left;
    HuffNode right;
    HuffNode parent;

    HuffNode( int v, int w, HuffNode lt, HuffNode rt, HuffNode pt )
      { value = v; weight = w; left = lt; right = rt; parent = pt; }
}
```

Huffman Tree Methods – Tree and Get Code for a Character

```
public HuffmanTree( )
{
    theCounts = new CharCounter( );
    root = null;
}

public HuffmanTree( CharCounter cc )
{
    theCounts = cc;
    root = null;
    createTree( );
}
```

```
public int [ ] getCode( int ch )
{
    HuffNode current = theNodes[ ch ];
    if( current == null )
        return null;

    String v = "";
    HuffNode par = current.parent;

    while ( par != null )
    {
        if( par.left == current )
            v = "0" + v;
        else
            v = "1" + v;
        current = current.parent;
        par = current.parent;
    }

    int [ ] result = new int[ v.length( ) ];
    for( int i = 0; i < result.length; i++ )
        result[ i ] = v.charAt( i ) == '0' ? 0 : 1;

    return result;
}
```

To Encode a Char

- getCode on the previous slide is called.
- The char is used to locate the Huffman node at the bottom of the tree, and then the code climbs the Huffman tree, prepending '0' or '1' at each edge, to make a string of text 0's and 1's representing the binary code.
- At the end, it is converted to an array of ints 0 or 1.

Create Tree

- These HuffNodes become elements of a PQ, and the weight is the priority
- Remove the lowest weight nodes and form one with a combined weight
- Put the new node back to the priority queue

```
private void createTree( )
{
    PriorityQueue<HuffNode> pq = new PriorityQueue<HuffNode>( );

    for( int i = 0; i < BitUtils.DIFF_BYTES; i++ )
        if( theCounts.getCount( i ) > 0 )
        {
            HuffNode newNode = new HuffNode( i,
                        theCounts.getCount( i ), null, null, null );
            theNodes[ i ] =  newNode;
            pq.add( newNode );
        }

    theNodes[ END ] = new HuffNode( END, 1, null, null, null );
    pq.add( theNodes[ END ] );

    while( pq.size( ) > 1 )
    {
        HuffNode n1 = pq.remove( );
        HuffNode n2 = pq.remove( );
        HuffNode result = new HuffNode( INCOMPLETE_CODE,
                              n1.weight + n2.weight, n1, n2, null );
        n1.parent = n2.parent = result;
        pq.add( result );
    }

    root = pq.element( );
}
```

To Decode a Char

- Some bits are turned into a string of text 0's and 1's, and getChar is called to interpret them.
- The bits (if enough of them) guide the path down from the root of the Huffman tree to some leaf, where the char resides.
- If one try doesn't work, another is done with more bits.

GetChar – Decoding a char Given a Code

```
/**
 * Get the character corresponding to code.
 */
public int getChar( String code )
{
    HuffNode p = root;
    for( int i = 0; p != null && i < code.length( ); i++ )
        if( code.charAt( i ) == '0' )
            p = p.left;
        else
            p = p.right;

    if( p == null )
        return ERROR;

    return p.value;
}
```

Unicode, ASCII and UTF-16 Encoding

- Unicode is an industry standard allowing computers to manipulate and represent text
- ASCII: 8 bit codes, 'A' = 0x41, 'a' = 0x61, space = 0x20, CR = 0x0d, etc. There are 128 codes, using 7 of the 8 bits.
- Unicode (coded with UTF-16): 16 bit codes, used for strings and chars inside Java. The ASCII chars provide the first codes: 'A' = 0x0041, 'a' = 0x0061, space = 0x0020, CR = 0x000d, etc.
- So the Java String "b a" is coded 0062 0020 0061, a total of 6 bytes.

UTF-16 Encoding

- It's easy to convert ASCII to Unicode: just add a first byte of 0x00. Java does this for us in class InputStreamReader.
- System.out.println(s) is using class PrintWriter to convert Unicode (ASCII subset) to ASCII by dropping a byte of 0x00.
- What does a Unicode above 0x007f stand for? Answer: all sorts of characters used in other languages, plus some symbols used in English that dont appear in ASCII, e.g.:
  ©= (0x00a9)
  ± = (0x00b1)
  ™= (0x2122)

UTF-8 Encoding

- It is really a kind of compression of Unicode.
- From the standard
  `http://www.faqs.org/rfcs/rfc3629.html`

Char. range (hex.)	UTF-8 octet sequence (binary)
0000 0000 – 0000 007F	0xxxxxxx
0000 0080 – 0000 07FF	110xxxxx 10xxxxxx
0000 0800 – 0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000 – 0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

a=0x0061 01100001

©=0x00a9 11000010 10101001

™=0x2122 11100010 10000100 10100010

UTF-8 Encoding

- The last rule converts the Unicode characters whose native values lie above 0xffff, that is, overflow 16 bits in their representation.
- UTF-16 codes all characters in 16 bits, including these outliers, by reserving some 16 bit codes for special use as prefixes.
- Thus to convert properly from UTF-16 to UTF-8, you need to have special cases for these prefix characters.
- See `http://en.wikipedia.org/wiki/UTF-8` if you are interested.
- UTF-8 codes are a prefix set. We can decode UTF-8 by matching the first byte

A Sample Program to Decode UTF-8 Codes

```
int ch; array of bytes bytes[]--
if (bytes[i++] >= 0)
    ch = bytes[i];
else {
    caseNo = UtfCases[bytes[i]]; // caseNo = 1, 2, or 3
    switch (caseNo) {
    case 1:
        ch = (bytes[i]&0x1f)<<6 + (byte[i+1]&0x3f);
        i += 2; break;
    case 2:
        ch = (bytes[i]&0xf)<<12 + (byte[i+1]&0x3f)<<6 +
        (byte[i+2]&0x3f);
        i += 3; break;
    case 3:
        ch =...; i += 4; break;
}
```