

Huffman

Fall Semester

Topic Outline

- Decode Hint
 - Array-based tree
 - Codeword length
- File IO
 - FILE* handles
 - Multiple files
- Structs
 - Dot notation
 - Using **typedef**
- Tree Nodes
 - Pointer to struct
 - Using **malloc**
 - Array of trees
- Encode Algorithm
 - Sort by frequency
 - Glue two trees together
 - Collapsing forest
- Up Next: Projectile Motion

Decode Hint (1)

- First idea. Use a hashtable, key is symbol and value is codeword.
- Be careful reading the codeword's string into a `char*` variable.

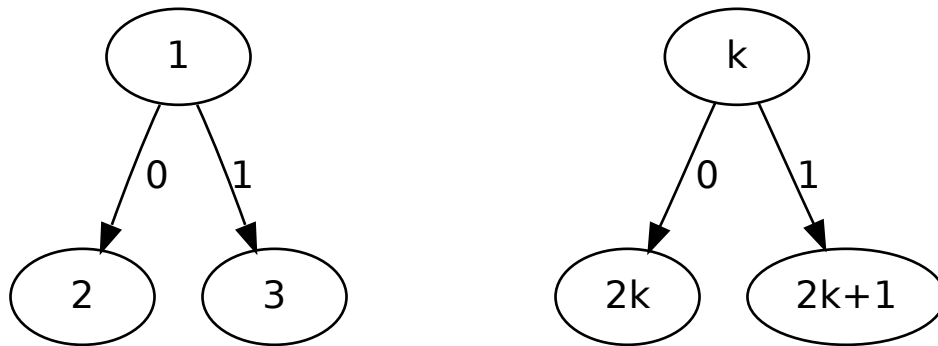
```
char* ht[256];  
ht['h']="011";           // ASCII value as index  
ht['a']="0100";         // text file is trickier
```

Or:

```
char ht[256][12];       // easier, but wasteful  
                        // ...not that wasteful
```

Decode Hint (2)

- Better idea. Key is “1”+codeword and value is symbol.
- Use array-based tree trick previously seen in heapsort.
- Branching. Move left for a zero and move right for a one.

**Table: A few examples.**

Index	Binary	Codeword
6	110	10
7	111	11
8	1000	000
9	1001	001

0	1	2	3	4	5	6	7	8	9
N/A	Root	Left	Right	Lt-Lt	Lt-Rt	Rr-Lt	Rr-Rt	L-L-L	L-L-R

Decode Hint (3)

- How long are the longest codewords? Maybe twelve bits? Then use an array of 10,000 chars. That's ~ 10 KB of memory.

```
#include<string.h>
#define N 10000 // preprocessor directive

char ht[N]; // flat array
memset(ht,0,N); // initialize to all '\0'

ht[11]='h'; // text file is easy now
ht[20]='a'; // 'a' is "1"+"0100"=20
```

File IO (1)

```
FILE* fin; // file handle

fin=fopen(fname,"r"); // open file for reading
if(fin!=NULL) // NULL indicates no file
{
    n=fread(&ch, // n bytes actually read
           sizeof(char),1,fin);
    if(n!=0) // n==0 indicates EOF
    {
        ... // &ch is a pointer
    }
}
```

File IO (2)

```
FILE *fin,*msg,*fout;
char fname[30];
sprintf(fname,"%s.out",    // build filename
        argv[1]);        // command-line argument
fin=fopen("scheme","r");  // multiple input files
msg=fopen("msg.in","r");
fout=fopen(fname,"w");    // open file for writing
...
fprintf(fout,"%c",ch);    // output to text file
fclose(fout);             // dumps buffer to file
```

Structs (1)

```
struct Widget // like a simple object
{
    int x; // no methods, just data
    double y; // data called "members"
};

struct Widget a; // type has annoying name

a.x=12; // dot notation
a.y=3.14;
```

Structs (2)

```
typedef struct                // better name w/ typedef
{                             // anonymous struct now
    int x;
    double y;
} Widget;                    // need a semicolon...
                               // variables here, messy

Widget a;

a.x=sizeof(Widget);          // sum sizes of members
a.y=3.14;
```

Array of Structs (1)

```
Widget arr[10];           // non-pointer elements
arr[0].x=12;             // static allocation
arr[0].y=3.14;          // 120 contiguous bytes

Widget *brr[10];         // pointer elements
brr[0]=(Widget*)malloc( // dynamic allocation
    sizeof(Widget));    // "new" w/o constructor

arr[1]=arr[0];           // copies member data
brr[1]=brr[0];          // copies pointer only
```

Array of Structs (2)

```
Widget *brr[10];           // need stdlib.h
brr[0]=(Widget*)malloc(    // cast of void* type
    sizeof(Widget));      // manual:  man malloc

(*brr[0]).x=12;           // dereference pointer
(*brr[0]).y=3.14;         // syntax is annoying

brr[0]->x=12;             // arrow syntax is better

arr[0]->x=12;             // ERROR, not a pointer
```

Tree Node

- The `typedef` name cannot be used within the definition of the struct because it hasn't happened yet. Instead, use the annoying form of the name for the declaration of a self-referential type.
- If `left` and `right` aren't pointers...disaster!!!

```
typedef struct t_node
{
    char symbol;
    int frequency;
    struct t_node *left,*right;
} Node;
```

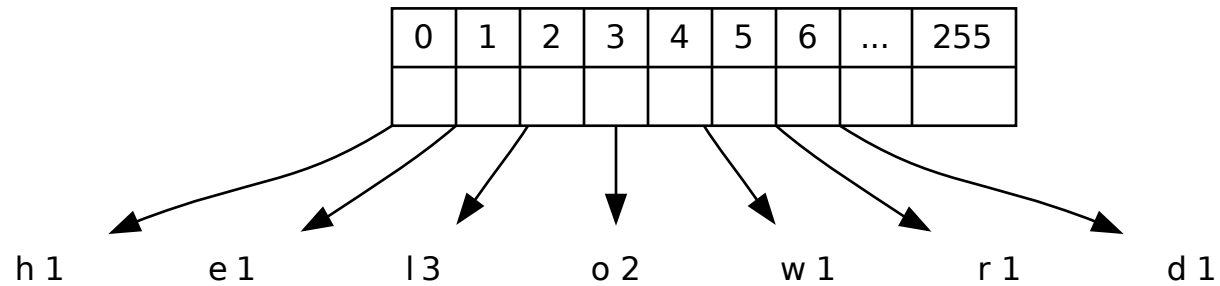
Outline of the Encode Algorithm

- There are at most 256 distinct symbols in an ASCII message. Count up the frequency of each symbol and malloc a one-node “tree” for it. Sort the array, glue the low two together, and repeat until there’s only one left whose root will be in `arr[0]`.
- Determine codewords by walking the tree, 0=left and 1=right.

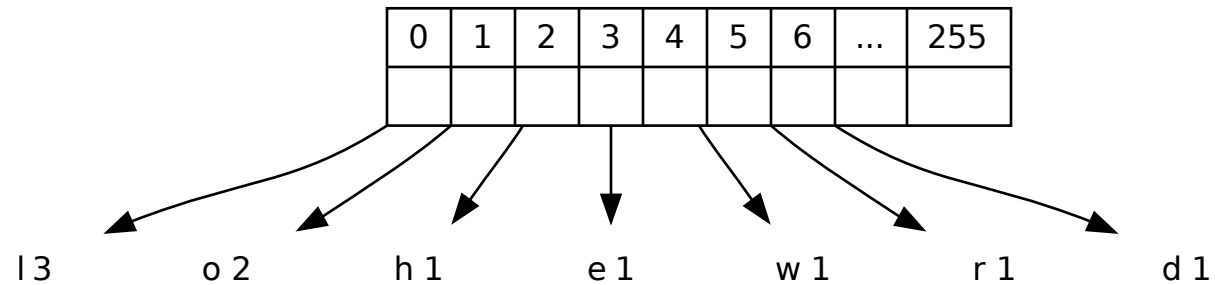
```
Node* arr[256];  
build_forest(arr,msg);  
collapse_forest(arr);  
build_scheme(arr[0]);  
encode_message(msg);
```

Trace of the Encode Algorithm (1)

Original message:

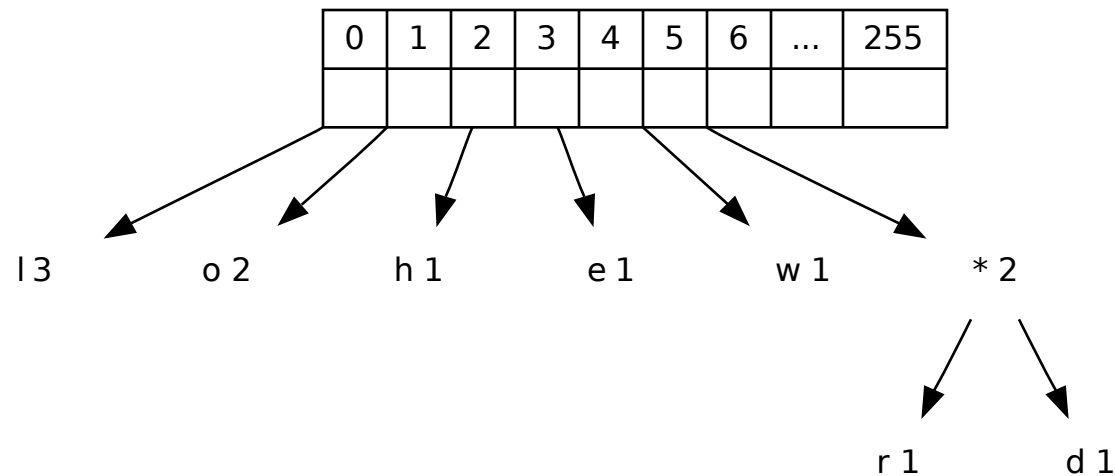


Sorted by frequency:



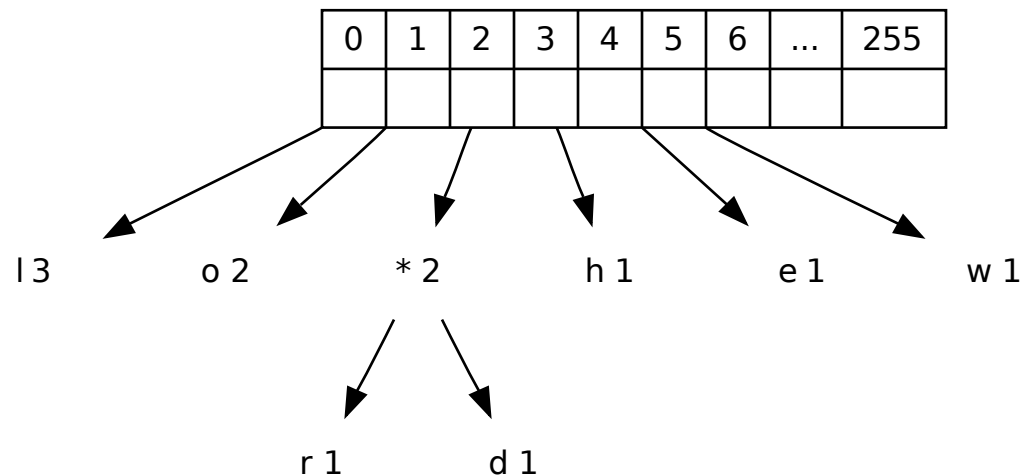
Trace of the Encode Algorithm (2)

- The new node's frequency must be the sum of the frequencies of the two nodes being glued together. The new node's symbol is irrelevant but for debugging purposes, if you want to print the array of trees, use something conspicuous like an asterisk.



Trace of the Encode Algorithm (3)

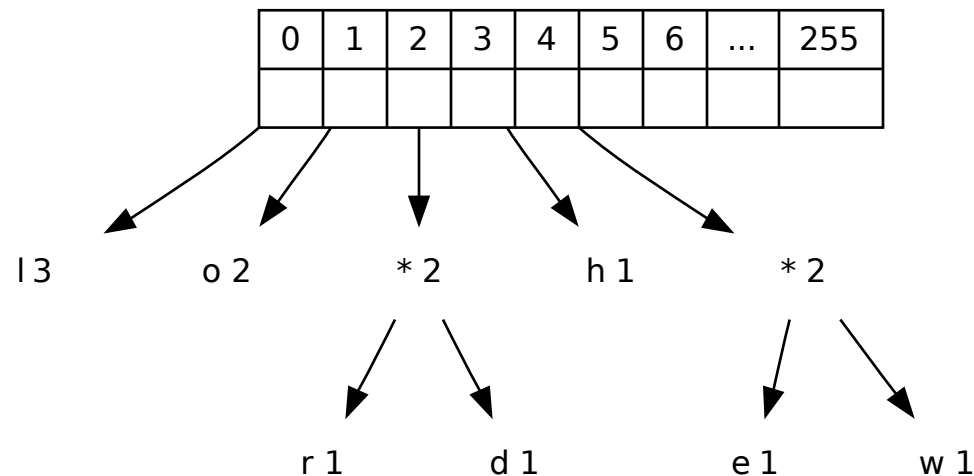
Re-sorted by frequency:



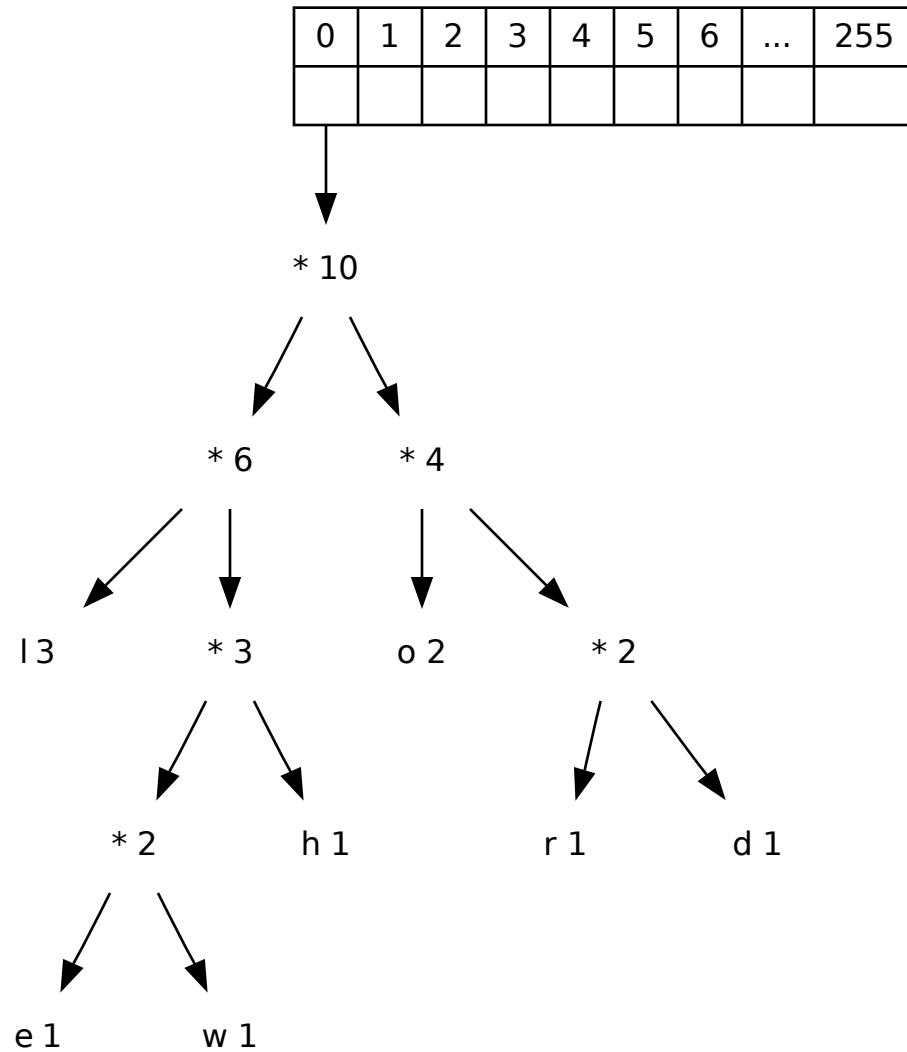
- Note. All the symbols from the original message are leaves of the tree; this is why no codeword can contain another codeword.

Trace of the Encode Algorithm (4)

Next low two glued together:



- Note. Low frequency symbols are glued together first, thus they end up further down in the tree and will have longer codewords.



Example Program: File IO

- command-line arguments
- `sprintf`, build filename
- `stderr`, error code return value
- backup file created

Example Program: Displaying a Tree

- display tree, right-then-left recursion trick
- tilt your head to the left to see the tree
- `qsort`, `stdlib.h`, manual: `man qsort`
- `node_cmp`, like a Comparator in Java, tricky pointers

Lab Assignment: Huffman Encode

- Input a message, build an array of trees, and repeat:
 - Sort the trees by frequency.
 - Glue the last two together.
- Output the custom scheme.
 - Walk the tree starting with a blank codeword.
 - Add “0” when you move left and “1” when you move right.
 - When you reach a leaf output the symbol and its codeword.
- Use the custom scheme to output an encoded message.
- Note. Arbitrary decisions give rise to equivalent schemes.