Valgrind

http://valgrind.org/docs/manual/quick-start.html

% valgrind -v --leak-check=full a.out
...
==23596== Invalid write of size 4
==23596==   at 0x80486DF: f (memory.c:22)
==23596==   by 0x80486FC: main (memory.c:29)
...
==23596== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==23596==   at 0x4023595: malloc (vg_replace_malloc.c:149)
==23596==   by 0x80486D5: f (memory.c:21)
==23596==   by 0x80486FC: main (memory.c:29)

see
memory.c
Hexadecimal

0x01, ah ah ah....
0x02, ah ah ah...
0x03, ah ah ah...

Endianness

Bitwise Operators

&    bitwise AND
|    bitwise OR
^    bitwise XOR
~    ones complement
<<   left shift
>>   right shift
Bitwise Operators

- AND (\&)
  - A: 0 1
  - B: 0 1

- OR (|)
  - A: 0 1
  - B: 0 1

- XOR (^)
  - A: 0 1
  - B: 0 1

- ones complement (~)
  - A: 0 1

See `binary.c, tolower.c, toupper.c`
Bitwise Operators

Swapping Values

```c
int FOO = 1;
int BAR = 4;

// base-2 value in x           base-2 value in y
int x = FOO;                   // 001
int y = BAR;                   // 001

x = x ^ y;                     // 001 ^ 100               100
                                      // 101
y = x ^ y;                     // 101
                                      //
x = x ^ y;                     // 101 ^ 001               001
                                      // 001
                                      // 100
```

see

`swap2.c`
Bitwise Operators

Swapping Values

```c
int FOO = 1;
int BAR = 4;

int x = FOO;    // FOO
int y = BAR;    // FOO

x = x ^ y;      // FOO ^ BAR
y = x ^ y;      // FOO ^ BAR

x = x ^ y;      // (FOO ^ BAR) ^ FOO
                // FOO ^ BAR ^ FOO
                // FOO ^ FOO ^ BAR
                // (FOO ^ FOO) ^ BAR
                // 0 ^ BAR
                // BAR

value in x

value in y

BAR

BAR

(FOO ^ BAR) ^ BAR

FOO ^ (BAR ^ BAR)

FOO ^ 0

FOO

see swap2.c
```
## Hashing Tables

### Linear Probing

<table>
<thead>
<tr>
<th>table[0]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>table[1]</td>
<td></td>
</tr>
<tr>
<td>table[2]</td>
<td></td>
</tr>
<tr>
<td>table[3]</td>
<td></td>
</tr>
<tr>
<td>table[4]</td>
<td></td>
</tr>
<tr>
<td>table[5]</td>
<td></td>
</tr>
<tr>
<td>table[6]</td>
<td></td>
</tr>
</tbody>
</table>

\[
\vdots
\]

<table>
<thead>
<tr>
<th>table[24]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>table[25]</td>
<td></td>
</tr>
</tbody>
</table>
In a room of n CS 50 students, what’s the probability that at least two students share the same birthday?
Hashing Tables

The Birthday Problem

\[
\bar{p}(n) = 1 \cdot \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{n-1}{365}\right)
\]

\[
= \frac{365 \cdot 364 \cdots (365 - n + 1)}{365^n}
\]

\[
= \frac{365!}{365^n(365 - n)!}
\]

Hashing Tables

The Birthday Problem

Image from http://www.mste.uiuc.edu/reese/birthday/probchart.GIF.
Hash Tables

Separate Chaining

Figure from Lewis and Denenberg’s Data Structures & Their Algorithms.
Trees

Figure by Larry Nyhoff.
Binary Search Trees

Figure from http://cs.calvin.edu/books/c++/ds/1e/.
Figure from Lewis and Denenberg’s Data Structures & Their Algorithms.
## Morse Code

<table>
<thead>
<tr>
<th>Letter</th>
<th>Morse Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>••</td>
</tr>
<tr>
<td>B</td>
<td>••••</td>
</tr>
<tr>
<td>C</td>
<td>•••</td>
</tr>
<tr>
<td>D</td>
<td>••</td>
</tr>
<tr>
<td>E</td>
<td>•</td>
</tr>
<tr>
<td>F</td>
<td>•••</td>
</tr>
<tr>
<td>G</td>
<td>•</td>
</tr>
<tr>
<td>H</td>
<td>•••••</td>
</tr>
<tr>
<td>I</td>
<td>•</td>
</tr>
<tr>
<td>J</td>
<td>••••••</td>
</tr>
<tr>
<td>K</td>
<td>••</td>
</tr>
<tr>
<td>L</td>
<td>•••</td>
</tr>
<tr>
<td>M</td>
<td>•</td>
</tr>
<tr>
<td>N</td>
<td>•</td>
</tr>
<tr>
<td>O</td>
<td>•••••</td>
</tr>
<tr>
<td>P</td>
<td>•••</td>
</tr>
<tr>
<td>Q</td>
<td>••••</td>
</tr>
<tr>
<td>R</td>
<td>••••</td>
</tr>
<tr>
<td>S</td>
<td>•••</td>
</tr>
<tr>
<td>T</td>
<td>•</td>
</tr>
<tr>
<td>U</td>
<td>•••</td>
</tr>
<tr>
<td>V</td>
<td>••••</td>
</tr>
<tr>
<td>W</td>
<td>••••</td>
</tr>
<tr>
<td>X</td>
<td>••••</td>
</tr>
<tr>
<td>Y</td>
<td>•••••</td>
</tr>
<tr>
<td>Z</td>
<td>•••・・・・</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Digit</th>
<th>Morse Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>••••••</td>
</tr>
<tr>
<td>2</td>
<td>••••••</td>
</tr>
<tr>
<td>3</td>
<td>••••••</td>
</tr>
<tr>
<td>4</td>
<td>••••••</td>
</tr>
<tr>
<td>5</td>
<td>••••••</td>
</tr>
<tr>
<td>6</td>
<td>••••••</td>
</tr>
<tr>
<td>7</td>
<td>••••••</td>
</tr>
<tr>
<td>8</td>
<td>••••••</td>
</tr>
<tr>
<td>9</td>
<td>••••••</td>
</tr>
<tr>
<td>0</td>
<td>••••••</td>
</tr>
</tbody>
</table>

Image adapted from Wikipedia.
Initialize a list of one-node binary trees containing weights \( w_1, w_2, \ldots, w_n \), one for each of the characters \( C_1, C_2, \ldots, C_n \).

1. **Do the following** \( n - 1 \) times:
   1. Find two trees \( T' \) and \( T'' \) in this list with roots of minimal weight \( w' \) and \( w'' \).
   2. Replace these two trees with a binary tree whose root has weight \( w' + w'' \) and whose subtrees are \( T' \) and \( T'' \); label the pointers to these subtrees 0 and 1, respectively:

   ![Diagram](image.png)

   Figure by Larry Nyhoff.

2. **The code for character** \( C_i \) **is the bit string labeling the path from root to leaf** \( C_i \) **in the final binary tree.**
Huffman Coding

Example

“ECEABEADCAEDEEEECEADEEEEEEDBAAEABDBBBAAEAAAC
DDCCEABEEDCBEEDEAEEEEEAAEDBCEBEEEADEAEEADAEBCC
DEDEAEDCDEEEAEEEE”

<table>
<thead>
<tr>
<th>character</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
<td>0.15</td>
<td>0.45</td>
</tr>
</tbody>
</table>
Huffman Coding

Example

Figure by Larry Nyhoff.
Huffman Coding

Example

Figure by Larry Nyhoff.
Huffman Coding

Example

Figure by Larry Nyhoff.
Huffman Coding

Example
Huffman Coding

Example

Figure by Larry Nyhoff.
Huffman Coding
In C

typedef struct node
{
    char symbol;
    int frequency;
    struct node *left;
    struct node *right;
} node;