Table of contents

Announcements

Programming assignment 2: Queues, trees, and graphs
  Using graphutils.h
  A DFS approach for solving isTree (using recursion)

Bits and bytes
  Why binary
  Decimal, binary, octal, and hexadecimal
  Representing characters
  Bitwise operations

Integers and basic arithmetic
  Representing negative and signed integers
Quiz 3
1. Due tomorrow, Friday 2/13.
2. 45 minutes.
3. Two tries.
4. Experimenting and identifying memory bugs.
5. Reviews recent concepts that would be fair game for exams.

Programming assignment 2
1. Due Friday 2/24.
2. More data structures: queues, BSTs, graphs; solidify managing memory.
All about integers

1. We will launch in to our chapter on representing data in computers
2. First: all about integers, signs, capacities, operations.
Table of contents

Announcements

Programming assignment 2: Queues, trees, and graphs
Using graphutils.h
A DFS approach for solving isTree (using recursion)

Bits and bytes
Why binary
Decimal, binary, octal, and hexadecimal
Representing characters
Bitwise operations

Integers and basic arithmetic
Representing negative and signed integers
Programming assignment 2: Queues, trees, and graphs

Programming Assignment 2 parts

1. bstLevelOrder: needs a queue (available in pa2/queue, will discuss today)
2. edgelist: will discuss today
3. isTree: needs DFS (stack)
4. solveMaze: needs BFS (queue)
5. mst: a greedy algorithm
6. findCycle: needs either DFS (stack) or BFS (queue)
7. matChainMul: another dynamic programming problem and prelude to integer operations
Using `graphutils.h`

- The adjacency list representation
- The edgelist representation
- The query
A DFS approach for solving isTree (using recursion)

- Solution using DFS
- Using recursion
- The visited array of Booleans indicating if a node already visited
- Careful not to backtrack
- Where is the stack data structure??
Announcements

Programming assignment 2: Queues, trees, and graphs
   Using graphutils.h
   A DFS approach for solving isTree (using recursion)

Bits and bytes
   Why binary
   Decimal, binary, octal, and hexadecimal
   Representing characters
   Bitwise operations

Integers and basic arithmetic
   Representing negative and signed integers
Why binary

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - ... and represent and manipulate numbers, sets, strings, etc...
- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

1.1V 0.9V 0.2V 0.0V
## Decimal, binary, octal, and hexadecimal

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0b0000</td>
<td>0o0</td>
<td>0x0</td>
</tr>
<tr>
<td>1</td>
<td>0b0001</td>
<td>0o1</td>
<td>0x1</td>
</tr>
<tr>
<td>2</td>
<td>0b0010</td>
<td>0o2</td>
<td>0x2</td>
</tr>
<tr>
<td>3</td>
<td>0b0011</td>
<td>0o3</td>
<td>0x3</td>
</tr>
<tr>
<td>4</td>
<td>0b0100</td>
<td>0o4</td>
<td>0x4</td>
</tr>
<tr>
<td>5</td>
<td>0b0101</td>
<td>0o5</td>
<td>0x5</td>
</tr>
<tr>
<td>6</td>
<td>0b0110</td>
<td>0o6</td>
<td>0x6</td>
</tr>
<tr>
<td>7</td>
<td>0b0111</td>
<td>0o7</td>
<td>0x7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Octal</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0b1000</td>
<td>0o10</td>
<td>0x8</td>
</tr>
<tr>
<td>9</td>
<td>0b1001</td>
<td>0o11</td>
<td>0x9</td>
</tr>
<tr>
<td>10</td>
<td>0b1010</td>
<td>0o12</td>
<td>0xA</td>
</tr>
<tr>
<td>11</td>
<td>0b1011</td>
<td>0o13</td>
<td>0xB</td>
</tr>
<tr>
<td>12</td>
<td>0b1100</td>
<td>0o14</td>
<td>0xC</td>
</tr>
<tr>
<td>13</td>
<td>0b1101</td>
<td>0o15</td>
<td>0xD</td>
</tr>
<tr>
<td>14</td>
<td>0b1110</td>
<td>0o16</td>
<td>0xE</td>
</tr>
<tr>
<td>15</td>
<td>0b1111</td>
<td>0o17</td>
<td>0xF</td>
</tr>
</tbody>
</table>

In C, format specifiers for printf() and fscanf():

1. decimal: ‘%d’
2. binary: none
3. octal: ‘%o’
4. hexadecimal: ‘%x’
Decimal, binary, octal, and hexadecimal

How to represent the range of unsigned char in each?

Unsigned char is one byte, 8 bits.

1. decimal: 0 to 255
2. binary: 0b0 to 0b11111111
3. octal: 0 to 0o377 (group by 3 bits)
4. hexadecimal: 0x00 to 0xFF (group by 4 bits)
Often encountered use of hexadecimal: RGB colors

<table>
<thead>
<tr>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>#000000</td>
<td>#FFFFFF</td>
<td>#6A757C</td>
<td>#CC0033</td>
</tr>
</tbody>
</table>

Red, green, blue values ranging from 0-255
Often encountered use of hexadecimal: RGB colors

<table>
<thead>
<tr>
<th>Hex Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#000000</td>
<td>Black</td>
</tr>
<tr>
<td>#FFFFFF</td>
<td>White</td>
</tr>
<tr>
<td>#6A757C</td>
<td>Gray</td>
</tr>
<tr>
<td>#CC0033</td>
<td>Red</td>
</tr>
</tbody>
</table>
Representing characters

▶ char is a 1-byte, 8-bit data type.
▶ ASCII is a 7-bit encoding standard.
▶ "man ascii" to see Linux manual.
▶ Compile and run ascii.c to see it in action.
▶ Some interesting characters: 7 (bell), 10 (new line), 27 (escape).

Figure: ASCII character set. Image credit Wikimedia
Bitwise operations

Why are bitwise operations important?

- Network and UNIX settings using bit masks (e.g., umask)
- Hardware and microcontroller programming (e.g., Arduinos)
- Instruction set architecture encodings (e.g., ARM, x86)
Bitwise operations

\[ \sim: \text{bitwise NOT} \]

unsigned char \( a = 128 \)

\[
\begin{align*}
a &= 0b1000_0000 \\
\sim a &= \sim 0b1000_0000 \\
&= 0b0111_1111 \\
&= 127
\end{align*}
\]
Bitwise operations

&: bitwise AND

$$3 \& 1 = 0b11 \& 0b01$$
$$= 0b01$$
$$= 1$$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>a &amp; b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Bitwise operations

|: bitwise OR

\[
\begin{align*}
3 \mid 1 &= 0b11 \mid 0b01 \\
&= 0b11 \\
&= 3 \\
2 \mid 1 &= 0b10 \mid 0b01 \\
&= 0b11 \\
&= 3
\end{align*}
\]

\[
\begin{array}{ccc}
 a & b & a \mid b \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1
\end{array}
\]
Bitwise operations

^: bitwise XOR

3 \land 1 = 0b11 \land 0b01
= 0b10
= 2

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a ^ b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
inplaceSwap.c: **Swapping variables without temp variables.**

How does it work?
Don’t confuse bitwise operators with logical operators

Bitwise operators

▶ ~
▶ &
▶ |
▶ ^

Logical operators

▶ !
▶ &&
▶ ||
▶ != (for bool type)
Table of contents

Announcements

Programming assignment 2: Queues, trees, and graphs
   Using graphutils.h
   A DFS approach for solving isTree (using recursion)

Bits and bytes
   Why binary
   Decimal, binary, octal, and hexadecimal
   Representing characters
   Bitwise operations

Integers and basic arithmetic
   Representing negative and signed integers
Representing negative and signed integers

Ways to represent negative numbers

1. Sign magnitude
2. 1s’ complement
3. 2’s complement
Representing negative and signed integers

Sign magnitude
Flip leading bit.
Representing negative and signed integers

1s’ complement

- Flip all bits
- Addition in 1s’ complement is sound
- In this encoding there are 2 encodings for 0
- -0: 0b1111
- +0: 0b0000
Representing negative and signed integers
2’s complement

<table>
<thead>
<tr>
<th>signed char</th>
<th>weight in decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>00001000</td>
<td>8</td>
</tr>
<tr>
<td>00010000</td>
<td>16</td>
</tr>
<tr>
<td>00100000</td>
<td>32</td>
</tr>
<tr>
<td>01000000</td>
<td>64</td>
</tr>
<tr>
<td>10000000</td>
<td>-128</td>
</tr>
</tbody>
</table>

Table: Weight of each bit in a signed char type

- what is the most positive value you can represent? 127
- what is the most negative value you can represent? -128
- how to represent -1? 11111111
- how to represent -2? 11111110
Representing negative and signed integers

2’s complement

<table>
<thead>
<tr>
<th>signed char</th>
<th>weight in decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>1</td>
</tr>
<tr>
<td>00000010</td>
<td>2</td>
</tr>
<tr>
<td>00000100</td>
<td>4</td>
</tr>
<tr>
<td>00001000</td>
<td>8</td>
</tr>
<tr>
<td>00010000</td>
<td>16</td>
</tr>
<tr>
<td>00100000</td>
<td>32</td>
</tr>
<tr>
<td>01000000</td>
<td>64</td>
</tr>
<tr>
<td>10000000</td>
<td>-128</td>
</tr>
</tbody>
</table>

Table: Weight of each bit in a signed char type

- MSB: 1 for negative
- To make a number negative: flip all bits and add 1.
- Addition in 2’s complement is sound
Importance of paying attention to limits of encoding

Figure: Image credit: CS:APP

Figure: Image credit: CS:APP
Importance of paying attention to limits of encoding

Figure: Image credit: CS:APP