C Programming: Debugging, Bits, Bytes, Integers

Yipeng Huang

Rutgers University

February 13, 2024

Table of contents

Challenges in CS programming assignments, strategies to get unstuck, resources

In CS 111, 112, 211, what are reasons programming assignments are challenging?

- Not sure where to start.
- It isn't working.
- The CS 211 teachers say that knowing Java helps programming in C, but C is nothing like Java.

What are strategies to get unstuck?

Lessons and ways in which programming in class is not like the real world.

- Coding deliberately is important. Have a plan. Understand the existing code. Test assumptions. Don't code by trial and error.
- Less code is better, and more likely to be correct.
- Reading code is as important and takes more time than writing code.

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - “lint” tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound type systems
  - Formal verification

Less “formal”: Lightweight, inexpensive techniques (that may miss problems)

This isn’t an either/or tradeoff… a spectrum of methods is needed!

Even the most “formal” argument can still have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth: *“Beware of bugs in the above code; I have only proved it correct, not tried it.”*

More “formal”: eliminate *with certainty* as many problems as possible.

From: https://www.seas.upenn.edu/~cis500/current/lectures/lec01.pdf

Strategies for debugging

Reduce to minimum example

- Check your assumptions.
- Use minimum example as basis for searching for help.

Debugging techniques

- Use assertions.
- Use debugging tools: Valgrind, Address Sanitizer, GDB.
- Use debugging printf statements.

Table of contents

Canvas timed quiz 3 and programming assignment 2

Programming assignment 2

1. Due Friday 2/23.
2. Graph algorithms and hash table.

Reading assignment: CS:APP Chapters 2.1, 2.2, 2.3

All about integers

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

bit bytes

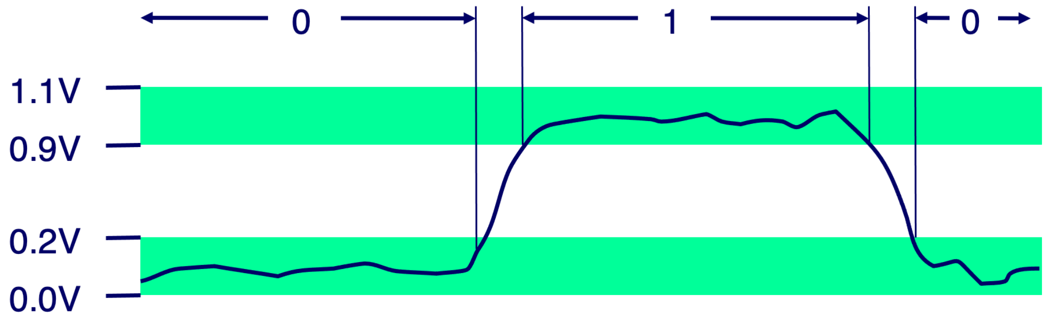
Table of contents

number ranges

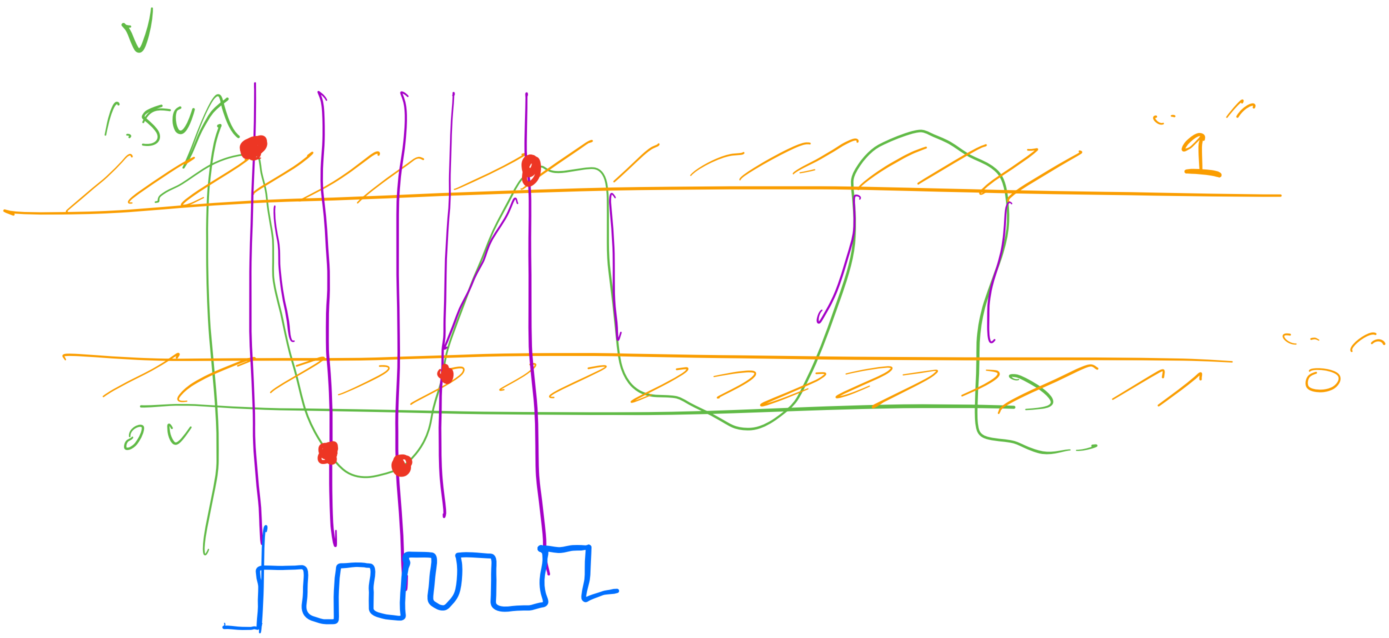
various options

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



0 1 0 0 1 ...



Analog → Digital → Quantum

1950s

1960s

2010+
2020+

$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$

Decimal, binary, octal, and hexadecimal

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

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

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

1. decimal: '%d'
2. binary: none
3. octal: '%o'
4. hexadecimal: '%x'

Base 10.

9	9	9	9	9	9
100k	10k	1000	100	10	1.

$9 \times 100k + 9 \times 10k + 9 \times 1000 + 9 \times 100 + 9 \times 10 + 9$

$= 1M - 1$

Base 2

1	1	1	1	1	1	1	1	1	1	1
1024	512	256	128	64	32	16	8	4	2	1

$1 \times 1024 + 1 \times 512 + 1 \times 256 + 1 \times 128 + 1 \times 64 + 1 \times 32$
$+ 1 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 1 \times 1$

$= 2^{11} - 1 = 2048 - 1 = 2047$

$2^{10} = 1024$
$2^{11} = 2048$

Base 8

0	1	1	1
512	64	8	1

$0 \times 512 + 1 \times 64 + 1 \times 8 + 1 \times 1$

$= 73$

```
printf("%d\n", 010 );
```

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

as oppoled to cmyk

6x16+10 = 106

0x75 = 7x16+5 = 117

0x7C = 7x16 + "C" = 112 + 12 = 124

0x33 = 3x16+3 = 51

Red, green, blue values ranging from 0-255

			???
#000000	#FFFFFF	#6A757C	#CC0033

#000000

white

#FFFFFF (255, 255, 255)

#6A757C (106, 117, 124)

#CC0033 (204, 0, 51)

0xCC = 12x16+12 = 204

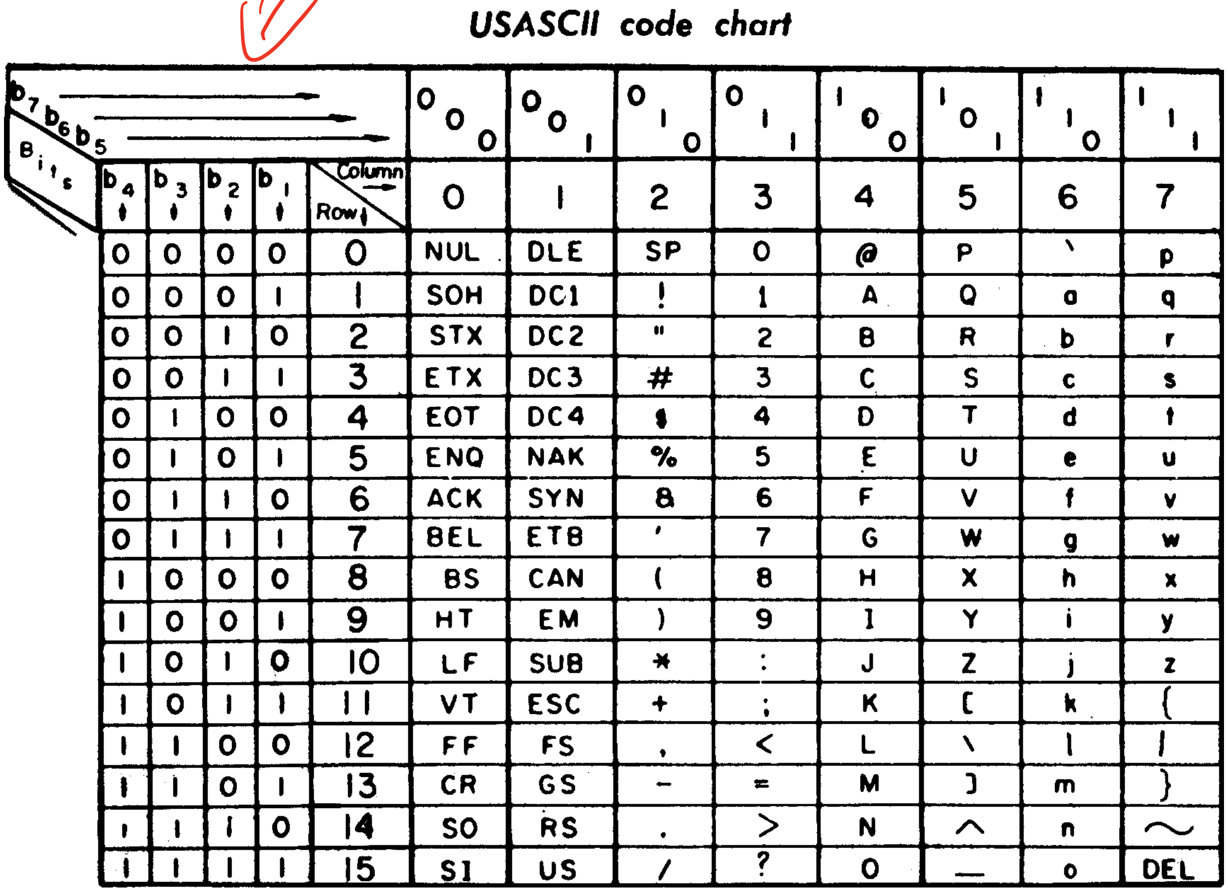
Often encountered use of hexadecimal: RGB colors

Red, green, blue values ranging from 0-255

#000000	#FFFFFF	#6A757C	#CC0033

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).

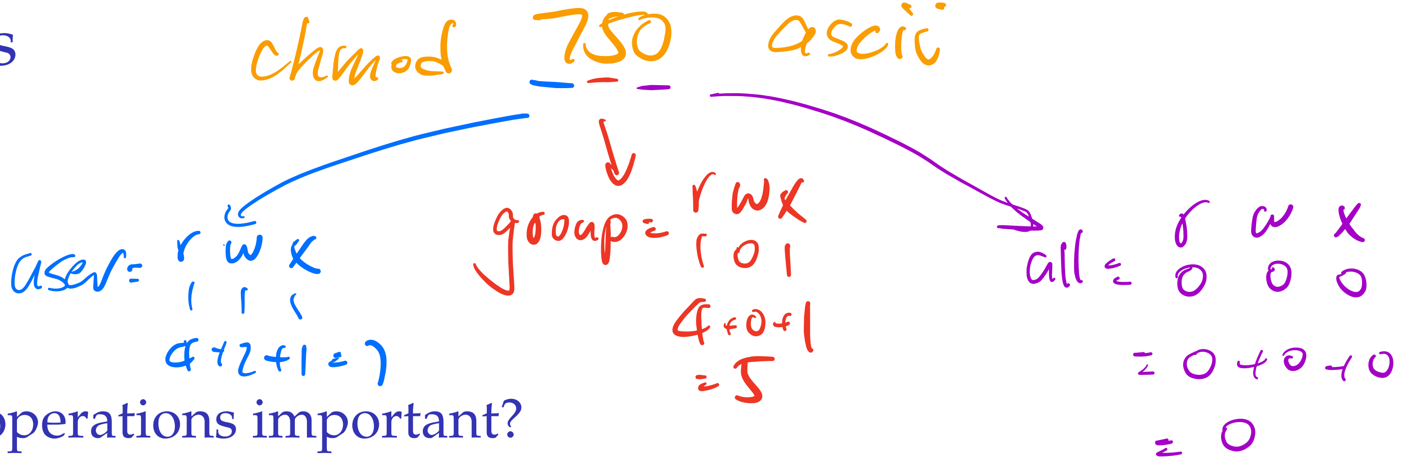


USASCII code chart

b4	b3	b2	b1	Row ↓ / Column →	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

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

```
int a = 128;
printf("%d\n", ~a);
```

127

~: bitwise NOT

unsigned char a = 128

$$a = 0b1000\_0000 = 128$$

$$\tilde{\ } a = \tilde{\ } 0b1000\_0000$$

$$= 0b0111\_1111$$

$$= 127$$

b	~b
0	1
1	0

Bitwise operations

printf("%d\n", 3&1);

3 => 0b00011
& 1 => 0b00001

0b00001

&: bitwise AND

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

$$
\begin{aligned}
3\&1 &= 0b11\&0b01 \\
&= 0b01 \\
&= 1
\end{aligned}
$$

Bitwise operations

| vs. ||

| : bitwise OR

bitwise OR — logical OR

printf("%d\n", 3|1)

$$3|1 = 0b11|0b01$$
$$= 0b11$$
$$= 3$$

3 => 0b0011
| 1 => 0b0001
0b0011

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

$$2|1 = 0b10|0b01$$
$$= 0b11$$
$$= 3$$

2 => 0b0010
| 1 => 0b0001
0b0011

Bitwise operations

printf("%d", 3^1);

THIS IS NOT 3^1

^: bitwise XOR

3 => 0b0011

^ 1 => 0b0001

0b0010

$$3 \wedge 1 = 0b11 \wedge 0b01$$
$$= 0b10$$
$$= 2$$

#

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

`inplaceSwap.c`: Swapping variables without temp variables.

How does it work?

```
  y = (x ^ y);
=> x = x ^ y;   = x ^ (x ^ y) = (x ^ x) ^ y
                = 0 ^ y
                = y;
=> y = x ^ y;
   = y ^ (x ^ y);
   = y ^ (y ^ x);
   = (y ^ y) ^ x
   = 0 ^ x
   = x;
```

```
x <= y;
y <= x;
```

Don't confuse bitwise operators with logical operators

Bitwise operators

- ~
- &
- |
- ^

Logical operators

- !
- &&
- ||
- != (for bool type)

Table of contents

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

signed char	weight in decimal
00000001	1
00000010	2
00000100	4
00001000	8
00010000	16
00100000	32
01000000	64
10000000	-128

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

signed char	weight in decimal
00000001	1
00000010	2
00000100	4
00001000	8
00010000	16
00100000	32
01000000	64
10000000	-128

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

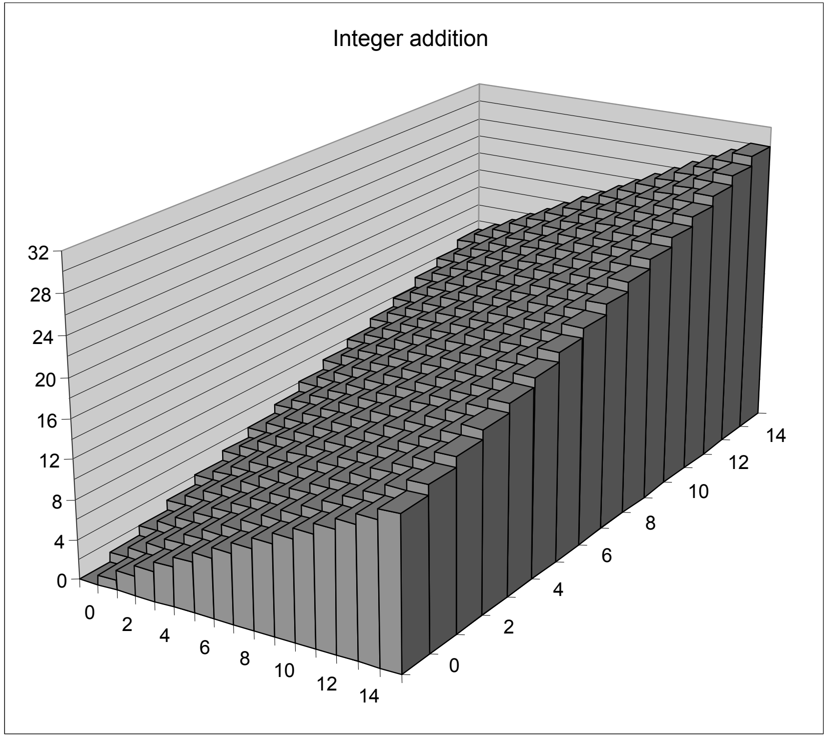


Figure: Image credit: CS:APP

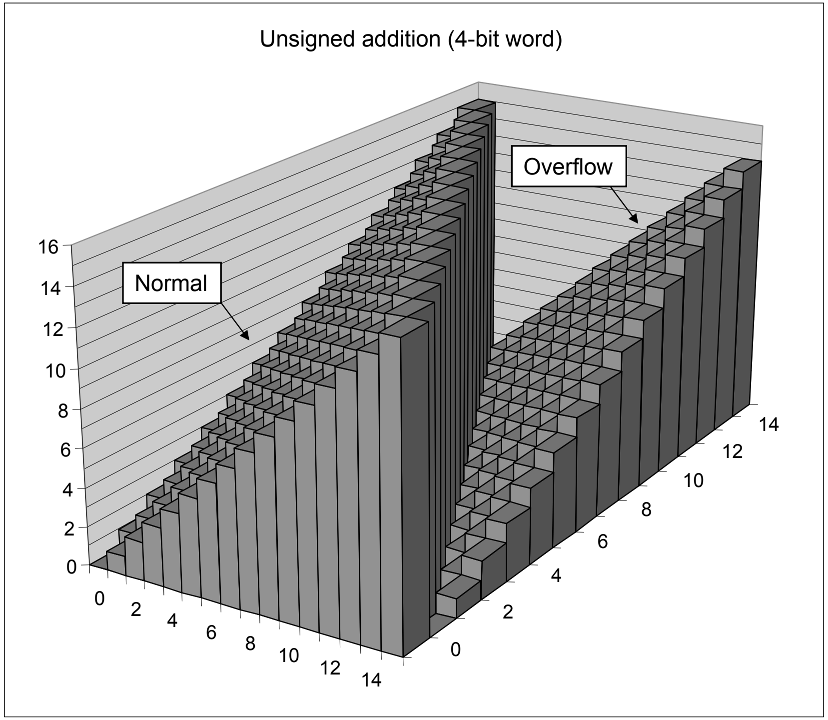


Figure: Image credit: CS:APP

Importance of paying attention to limits of encoding

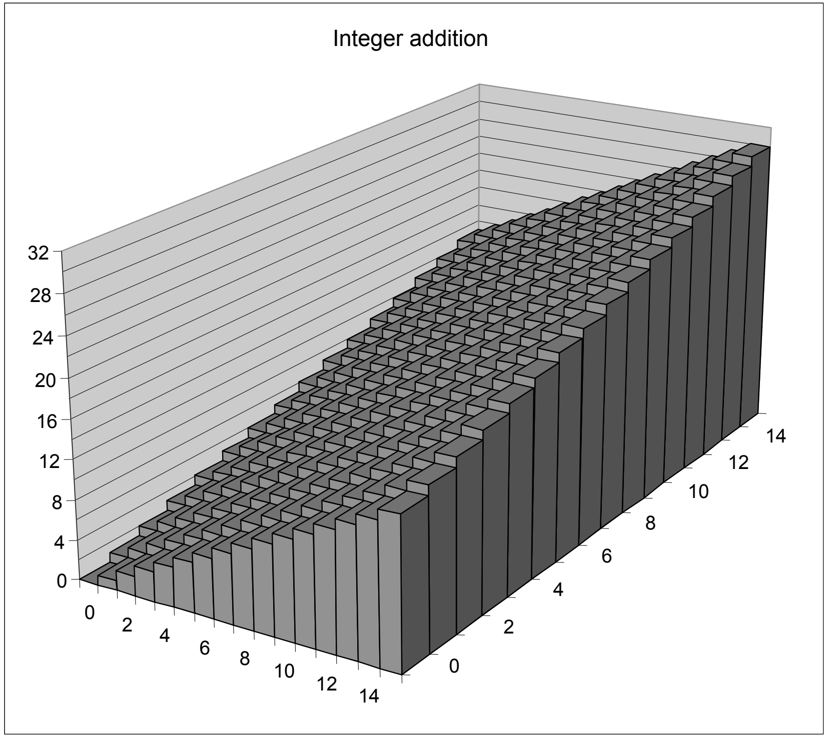


Figure: Image credit: CS:APP

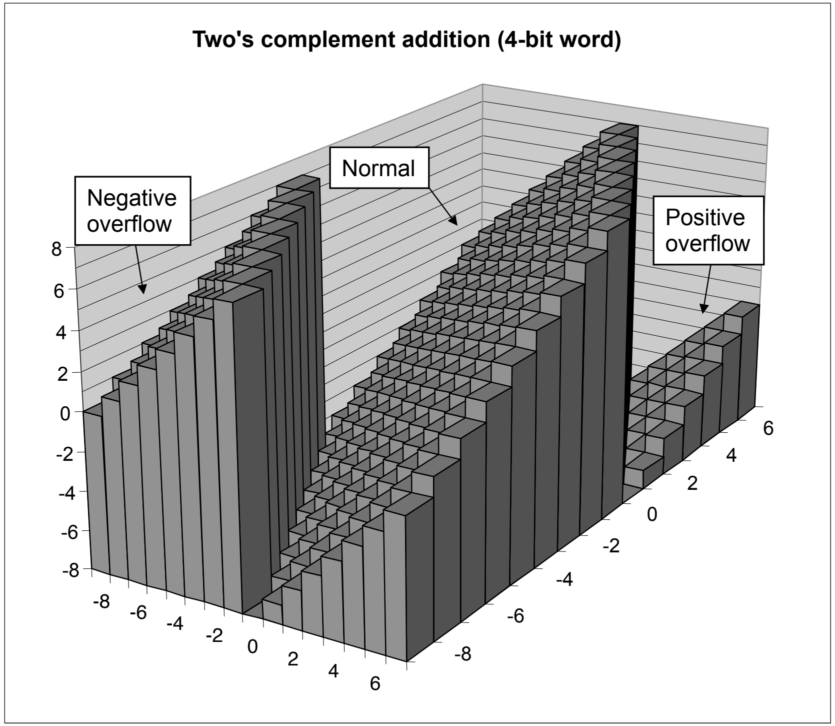


Figure: Image credit: CS:APP

https://www.theatlantic.com/technology/archive/2014/12/how-gangnam-style-broke-youtube/383389/

Table of contents

Programming assignment 2: Graphs, trees, queues, hashes

Programming Assignment 2 parts

1. edgelist: loading and printing a graph
2. isTree: needs either DFS (stack) or BFS (queue)
3. mst: a greedy algorithm
4. solveMaze: needs either DFS (stack) or BFS (queue)
5. findCycle: needs either DFS (stack) or BFS (queue)
6. hashTable: a separate chaining hash table

Using `graphutils.h`

- The adjacency list representation
- The edgelist representation
- The query

Binary search tree

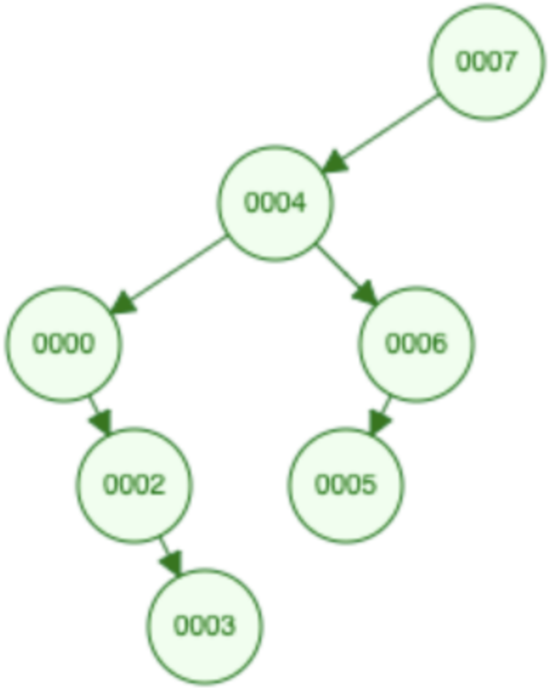


Figure: BST with input sequence 7, 4, 7, 0, 6, 5, 2, 3. Duplicates ignored.

Binary search tree level order traversal

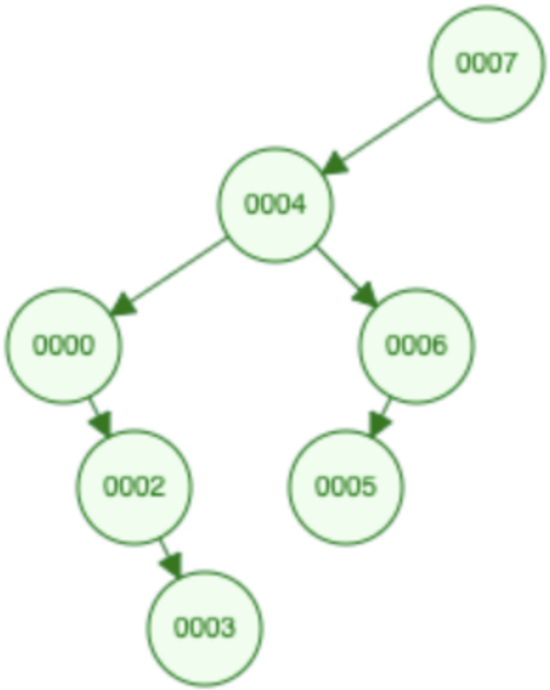


Figure: Level order, left-to-right traversal would return 7, 4, 0, 6, 2, 5, 3.

Binary search tree traversal orders

Breadth-first

- For example: level-order.
- Needs a queue (first in first out).
- Today in class we will build a BST and a Queue.

Depth-first

- For example: in-order traversal, reverse-order traversal.
- Needs a stack (first in last out).

typedef

Why types are important

- Natural language has nouns, verbs, adjectives, adverbs.
- Type safety.
- Interpretation vs. compilation.

BSTNode

```
typedef struct BSTNode BSTNode;
struct BSTNode {
    int key;
    BSTNode* l_child; // nodes with smaller key will be in left s
    BSTNode* r_child; // nodes with larger key will be in right s
};
```

QueueNode, Queue

```
// queue needed for level order traversal
typedef struct QueueNode QueueNode;
struct QueueNode {
    BSTNode* data;
    QueueNode* next; // pointer to next node in linked list
};
typedef struct Queue {
    QueueNode* front; // front (head) of the queue
    QueueNode* back; // back (tail) of the queue
} Queue;
```

Let's implement `enqueue()`

`https://visualgo.net/en/queue`

- First, consider if queue is empty.
- Then, consider if queue is not empty. Only need to touch back (tail) of the queue.

Let's implement `dequeue()`

`https://visualgo.net/en/queue`

- First, consider if queue will become empty.
- Then, consider if queue will not not empty. Only need to touch front (head) of the queue.

Subtle point: why are the function signatures (return, parameters) of `enqueue()` and `dequeue()` the way they are?