Data representation: Bits, Bytes, Integers

Yipeng Huang

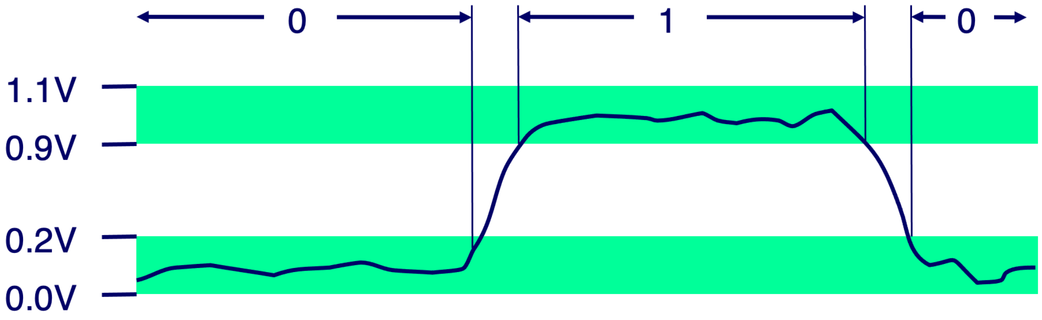
Rutgers University

October 7, 2025

Table of contents

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



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'

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)

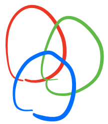
Often encountered use of hexadecimal: RGB colors

Red, green, blue values ranging from 0-255

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

0/255 R
0/255 G
0/255 B

F → 15
15×16+15 = 255
255/255 = 100%
100% R
100% G
100% B



0x6A = 6×16+10 = 106
0x75 = 7×16+5 = 112+5 = 117
0x7C = 7×16+12 = 124

0xCC = 12×16+12 = 192+12 = 204
0x00 = 0%
0x33 = 3×16+3 = 51

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

b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Column → / Row ↓	0	1	2	3	4	5	6	7
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

~: bitwise NOT

unsigned char a = 128

$$\begin{aligned} a &= 0b1000\_0000 \\ \tilde{}a &= \tilde{}0b1000\_0000 \\ &= 0b0111\_1111 \\ &= 127 \end{aligned}$$

b	~b
0	1
1	0

Bitwise operations

&: bitwise AND

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

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

Bitwise operations

| : bitwise OR

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

$$\begin{aligned} 2|1 &= 0b10|0b01 \\ &= 0b11 \\ &= 3 \end{aligned}$$

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

Bitwise operations

^: bitwise XOR

$$\begin{aligned} 3 \wedge 1 &= 0b11 \wedge 0b01 \\ &= 0b10 \\ &= 2 \end{aligned}$$

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?

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

Signed char, 1 byte, 8 bits

most neg value

0b1111_1111

= -127

Zero

0b0000_0000

0b1000_0000

largest pos val

0b0111_1111

= 127

Sign magnitude

Flip leading bit.

neg one -1

0b1000_0001

one 1

0b0000_0001

$-1+1$

$$
\begin{array}{rl}
 & 0b\,1000\_0001 \\
+) & 0b\,0000\_0001 \\
\hline
 & 1000\_0010 = -2
\end{array}
$$

Representing negative and signed integers

1s' complement

- Flip all bits as negation
- Addition in 1s' complement is sound

  remember to add back the carry
- In this encoding there are 2 encodings for 0
- -0: 0b1111
- +0: 0b0000

1s' comp

most neg number
0b1000_0000
= -127

zero
0b0000_0000
0b1111_1111

most pos number
0b0111_1111
= 127

neg one
0b1111_1110

pos one
0b0000_0001

-1 + 1

0b1111_1110
+ 0b0000_0001

0b1111_1111 = 0

-1 + 2

1 1111 11
0b1111_1110
+ 0b0000_0010

0b0000 0000
1

0b0000_0001 = +1

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

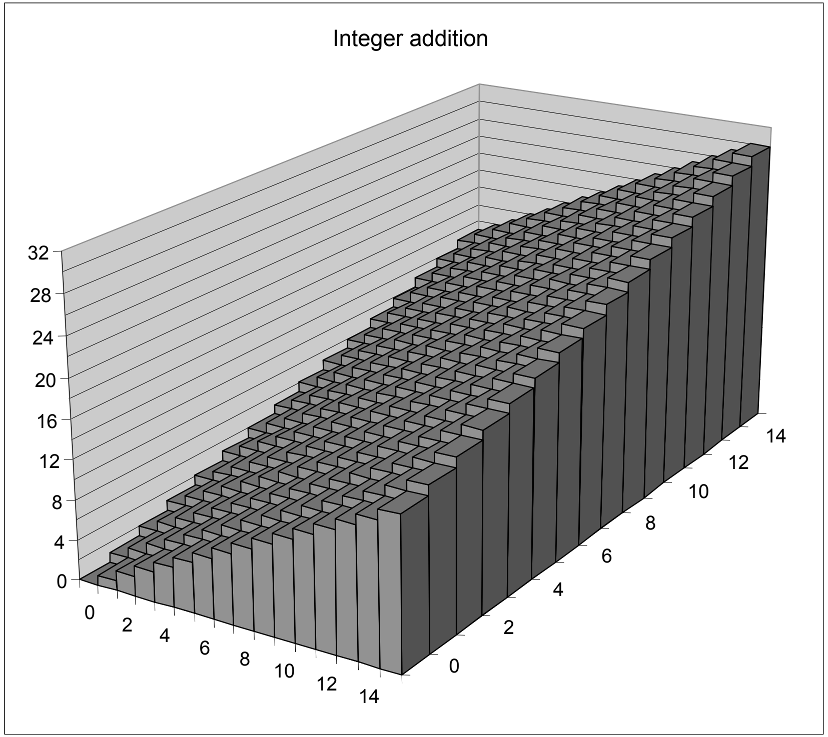


Figure: Image credit: CS:APP

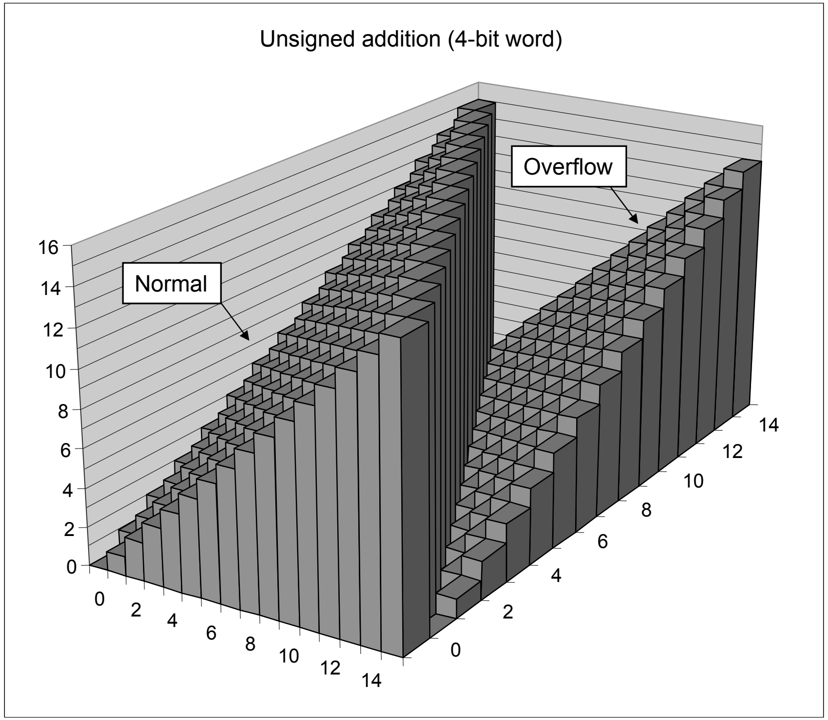


Figure: Image credit: CS:APP

Importance of paying attention to limits of encoding

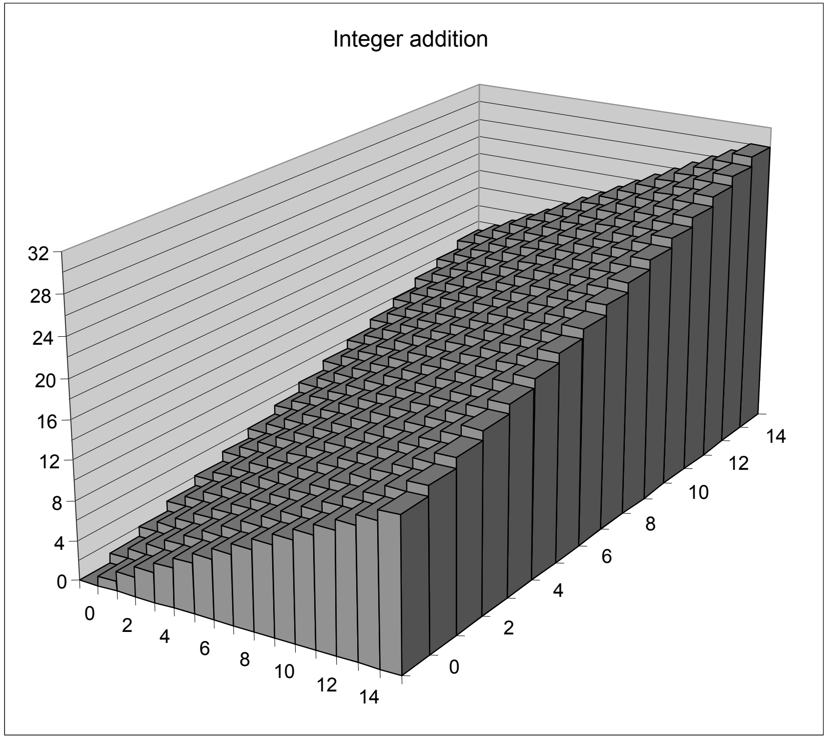


Figure: Image credit: CS:APP

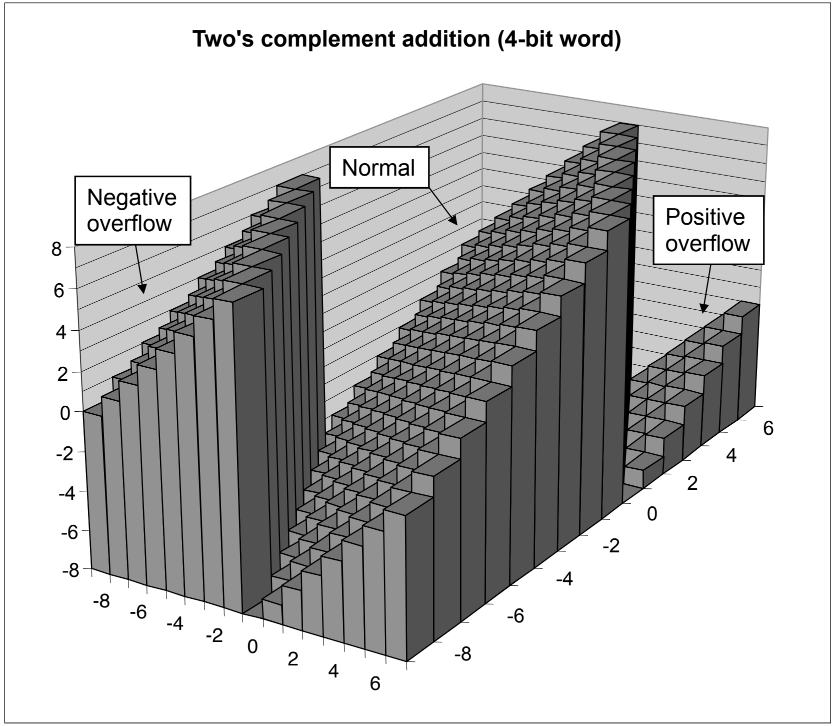


Figure: Image credit: CS:APP

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

Table of contents

Unsigned fixed-point binary for fractions

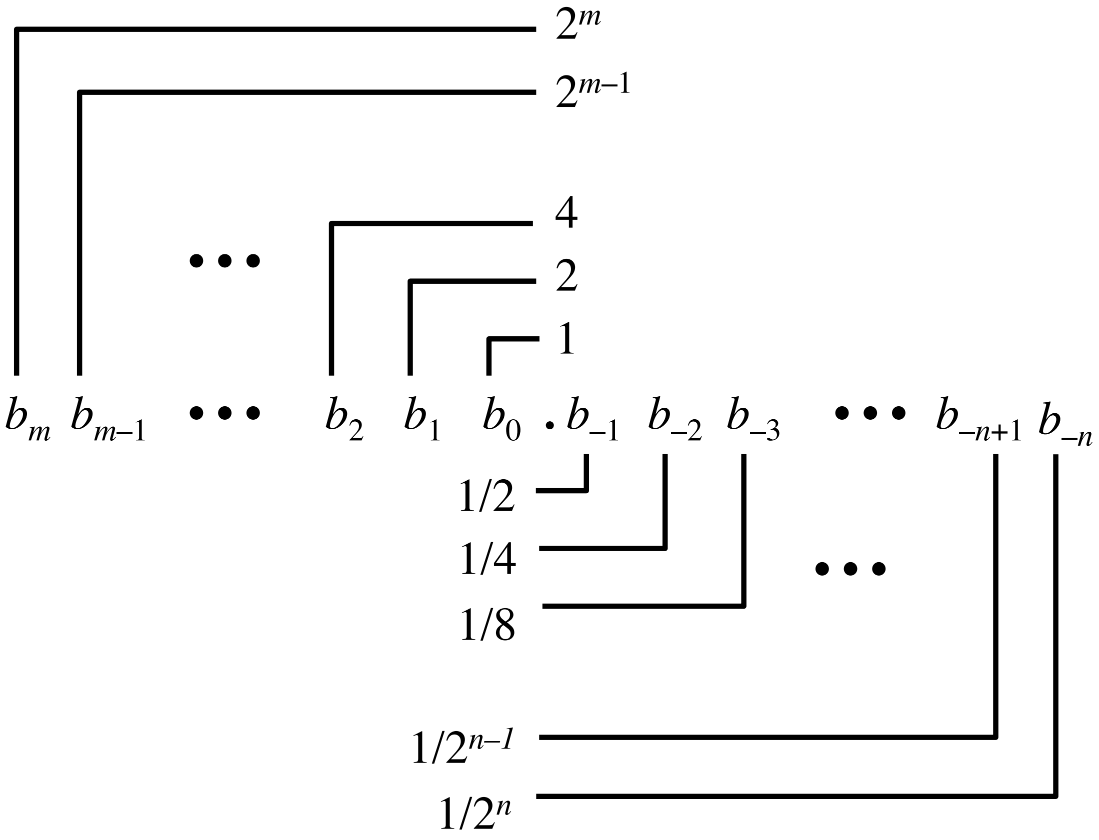


Figure: Fractional binary. Image credit CS:APP

Unsigned fixed-point binary for fractions

unsigned fixed-point char example	weight in decimal
1000.0000	8
0100.0000	4
0010.0000	2
0001.0000	1
0000.1000	0.5
0000.0100	0.25
0000.0010	0.125
0000.0001	0.0625

Table: Weight of each bit in an example fixed-point binary number

- $.625 = .5 + .125 = 0000.1010_2$
- $1001.1000_2 = 9 + .5 = 9.5$

Signed fixed-point binary for fractions

signed fixed-point char example	weight in decimal
1000.0000	-8
0100.0000	4
0010.0000	2
0001.0000	1
0000.1000	0.5
0000.0100	0.25
0000.0010	0.125
0000.0001	0.0625

Table: Weight of each bit in an example fixed-point binary number

- $-.625 = -8 + 4 + 2 + 1 + 0 + .25 + .125 = 1111.0110_2$
- $1001.1000_2 = -8 + 1 + .5 = -6.5$

Limitations of fixed-point

- Can only represent numbers of the form $x/2^k$
- Cannot represent numbers with very large magnitude (great range) or very small magnitude (great precision)

Bit shifting

$<< N$ Left shift by N bits

- multiplies by 2^N
- $2 << 3 = 0000\_0010_2 << 3 = 0001\_0000_2 = 16 = 2 * 2^3$
- $-2 << 3 = 1111\_1110_2 << 3 = 1111\_0000_2 = -16 = -2 * 2^3$

$>> N$ Right shift by N bits

- divides by 2^N
- $16 >> 3 = 0001\_0000_2 >> 3 = 0000\_0010_2 = 2 = 16/2^3$
- $-16 >> 3 = 1111\_0000_2 >> 3 = 1111\_1110_2 = -2 = -16/2^3$

Table of contents

Programming assignment 2: Graphs, trees, queues, hashes

Programming Assignment 2 parts

1. hashTable: implementing a separate-chaining hash table.
2. edgelist: loading and printing a graph
3. isTree: needs either DFS (stack) or BFS (queue)
4. mst: a greedy algorithm
5. findCycle: needs either DFS (stack) or BFS (queue)
6. matChainMul: a dynamic programming approach to multiplying matrices with minimal operations using recursion

Using `graphutils.h`

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

Binary search tree

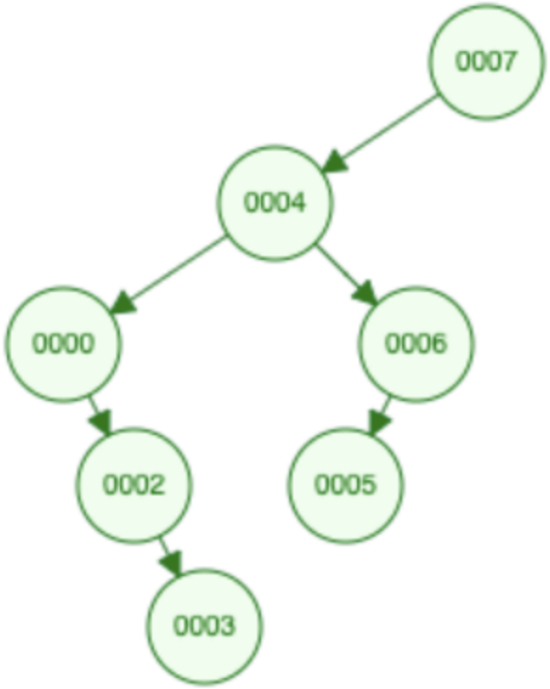


Figure: BST with input sequence 7, 4, 7, 0, 6, 5, 2, 3. Duplicates ignored.

Binary search tree level order traversal

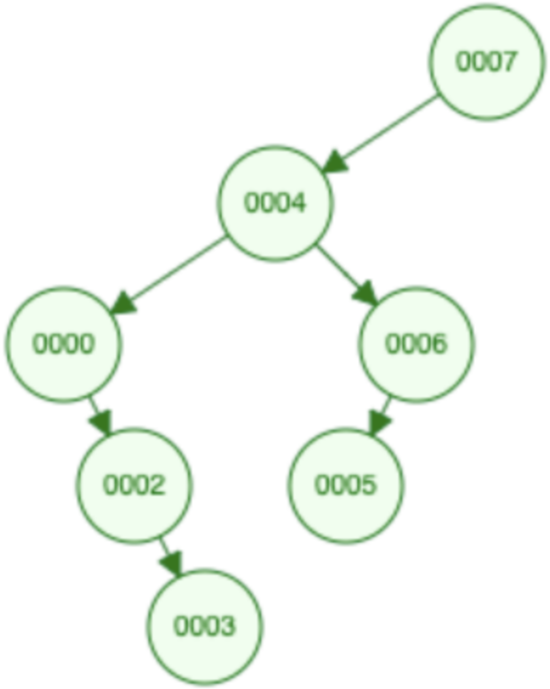


Figure: Level order, left-to-right traversal would return 7, 4, 0, 6, 2, 5, 3.

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).
- But in the example code implementation,*where is the stack data structure?*

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?

Table of contents

`matChainMul.c`: Minimum number of multiplies needed for matrix chain multiplication

Learning objectives

- Review and master recursion.
- Array subsetting using pointer arithmetic.
- Using pass-by-reference to return computed results.
- A new algorithm that most classmates have not seen before.

Cost of multiplying matrices: the number of multiplies

- $A_{l \times m} \times B_{m \times n}$
- Needs $l \times m \times n$ number of multiplies
- (Well-kept secret: fewer multiplications possible, see Strassen's algorithm)

`matChainMul.c`: Minimum number of multiplies needed for matrix chain multiplication

$$A \times B \times C = \begin{bmatrix} a_{0,0} & a_{0,1} \end{bmatrix}_{1\times 2} \times \begin{bmatrix} b_{0,0} \\ b_{1,0} \end{bmatrix}_{2\times 1} \times \begin{bmatrix} c_{0,0} & c_{0,1} \end{bmatrix}_{1\times 2}$$

Parenthesization 1: 4+4 = 8 multiplies

$$A \times (B \times C) = \begin{bmatrix} a_{0,0} & a_{0,1} \end{bmatrix}_{1\times 2} \times \begin{bmatrix} b_{0,0}c_{0,0} & b_{0,0}c_{0,1} \\ b_{1,0}c_{0,0} & b_{1,0}c_{0,1} \end{bmatrix}_{2\times 2}$$
$$= \begin{bmatrix} \left(a_{0,0}b_{0,0}c_{0,0} + a_{0,1}b_{1,0}c_{0,0}\right) & \left(a_{0,0}b_{0,0}c_{0,1} + a_{0,1}b_{1,0}c_{0,1}\right) \end{bmatrix}_{1\times 2}$$

Parenthesization 2: 2+2 = 4 multiplies

$$(A \times B) \times C = \left(a_{0,0}b_{0,0} + a_{0,1}b_{1,0}\right) \times \begin{bmatrix} c_{0,0} & c_{0,1} \end{bmatrix}_{1\times 2}$$
$$= \begin{bmatrix} \left(a_{0,0}b_{0,0} + a_{0,1}b_{1,0}\right)c_{0,0} & \left(a_{0,0}b_{0,0} + a_{0,1}b_{1,0}\right)c_{0,1} \end{bmatrix}_{1\times 2}$$

`matChainMul.c`: Minimum number of multiplies needed for matrix chain multiplication

$A \times B \times C \times D$

First partitioning

- $A(BCD)$; but what is cost of finding (BCD)? Needs decomposition.
- $(AB)(CD)$
- $(ABC)D$; but what is cost of finding (ABC)? Needs decomposition.

Second partitioning

- $A(B(CD))$
- $A((BC)D)$
- $(AB)(CD)$
- $(A(BC))D$
- $((AB)C)D$