Data & Data Structures

Peter Rounce
P.Rounce@cs.ucl.ac.uk

Data and Data Structures

This section of the course deals with data and how some of the data structures used by software to hold data. The latter covers the simple data structures to hold single values and arrays and the dynamic structures that hold queues, stack and linked lists.

C language primitives

	Architecture		
	16-bit	**32-bit**	**64-bit**
Integer	16-bits	32–bits	32–bits
Long	32 –bits	64–bits	64-bits
Float	32-bit	32-bit	32-bit
Double	64-bit	64-bit	64-bit
char	8-bit	8-bit	8-bit

[Assumes that an *n-bit* architecture has *n-bit* addresses, *n-bit* data bus and *n-bit* registers.]

In Java, the above is approximately true for a 32-bit architecture model, although the Java ‘char’ primitive type is 16-bits and uses UniCode encoding: the first 128 Unicode numbers are identical to the 128 values of the ASCII character set.

ASCII character set

American Standard Code for Information Interchange

7-bit code for teletype machines

Form basis of many existing sets

Basic Unix character set.

NB: 30(hex) = '0', 31 = '2'
41 = 'A' 61 = 'a'

Control Characters (00→1F) still used

HT (09_{16}): the tab character
LF ($0A_{16}$): the line feed character
CR ($0D_{16}$): (carriage) return character
ESC ($1B_{16}$): escape character
BS (08_{16}): backspace character

	000	001	010	011	100	101	110	111
	0	1	2	3	4	5	6	7
0000	0	NUL	DLE	SP	0	@	P	'	p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	10	LF	SUB	*	:	J	Z	j	z
1011	11	VT	ESC	+	;	K	[	k	{
1100	12	FF	FS	,	<	L	\	l	|
1101	13	CR	GS	-	=	M	]	m	}
1110	14	SO	RS	.	>	N	^	n	~
1111	15	SI	US	/	?	O	_	o	DEL

ASCII – designed for teletype machines – essentially large electric typewriter that could receive ASCII characters via a modem from a telephone line and then prints characters on the paper. No computer was involved: each end of the telephone connection had a teletype.

The 'control' characters (00_{16} to $1F_{16}$) were sent in transmissions to control the mechanics of the type writer, which gives the control characters their names:

name	hex	full name	Role
CR	$0D_{16}$	Carriage Return	move the print head back from the RHS of the paper to the start of the line
LF	$0A_{16}$	Line Feed	move the paper one line up – print head moves 1 line down page
FF	FF_{16}	Form Feed	move the paper one page up – print head moves to top of next page
BS	08_{16}	Back Space	move print head back one space – so that previous character can be over-printed
ESC	$1B_{16}$	Escape	indicated the start of a longer sequence of control information – an 'escape sequence'.

All the control characters can be generated on a modern computer by holding down the 'control' key on the keyboard and hitting the key in the same row as the control character wanted but 4 columns to the right,

BS Control-H, CR Control-M, LF Control-J

Control-H is useful to remember when your normal delete or rub-out key is malfunctioning.

LF/CR Issue

On teletypes, the normal end of a line sequence was send 'CR' and then 'LF' – 'CR' to move the carriage back to the start and 'LF' to feed the paper one line.

Having a 2 character combination allowed more complex scenarios on the teletype, e.g. overprinting a whole line if 'LF' is lost. None of these are remotely interesting now, or even then in the 1950s.

When Unix was introduced, the Unix designers decided that Unix text files only needed a single character at the end of a line and chose 'LF' ($0A_{16}$) as this single line terminating character for text files. Confusingly this character is inserted in Unix text files, when you type the 'Return' key on the keyboard: you would expect 'CR' – the carriage return character.

When Microsoft started up with DOS, they terminated the ends of lines in text files with the 2-character combination 'CR', 'LF, in this order: both inserted by a single strike on the 'Return' key of the keyboard. Microsoft systems still use this CR/LF combination, which causes a slight problem when moving text files between Unix and Microsoft systems: a text file produced on a MS system but displayed on a Unix system shows the unwanted CR as the Control-M character (displayed as '^M') at the end of each line – see above for why Control-M. This is ignored by Unix systems, but does make your text display unsightly. There are unix comands to remove/insert CR characters into text files – dos2unix and unix2dos commands.

Character Set: Windows 3.1 Latin 1

256-bit code

codes (00 to 7E)

the same as ASCII.

80- FF are sometimes

called *extended characters*

For output to a laser printer,
can select
the character set
the font
the character size
the style (bold, italic)
e.g. - a, *a*, a, a, a),
and to switch between them

19U Windows 3.1 Latin 1 (W1)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL 0	DLE 16	SP 32	0 48	@ 64	P 80	` 96	p 112	128	144	NBS 160	° 176	À 192	Đ 208	à 224	ð 240
1	SOH 1	DC1 17	! 33	1 49	A 65	Q 81	a 97	q 113	129	‘ 145	¡ 161	± 177	Á 193	Ñ 209	á 225	ñ 241
2	STX 2	DC2 18	" 34	2 50	B 66	R 82	b 98	r 114	‚ 130	’ 146	¢ 162	² 178	Â 194	Ò 210	â 226	ò 242
3	ETX 3	DC3 19	# 35	3 51	C 67	S 83	c 99	s 115	ƒ 131	“ 147	£ 163	³ 179	Ã 195	Ó 211	ã 227	ó 243
4	EOT 4	DC4 20	$ 36	4 52	D 68	T 84	d 100	t 116	„ 132	” 148	¤ 164	´ 180	Ä 196	Ô 212	ä 228	ô 244
5	ENQ 5	NAK 21	% 37	5 53	E 69	U 85	e 101	u 117	… 133	• 149	¥ 165	µ 181	Å 197	Õ 213	å 229	õ 245
6	ACK 6	SYN 22	& 38	6 54	F 70	V 86	f 102	v 118	† 134	– 150	¦ 166	¶ 182	Æ 198	Ö 214	æ 230	ö 246
7	BEL 7	ETB 23	' 39	7 55	G 71	W 87	g 103	w 119	‡ 135	— 151	§ 167	· 183	Ç 199	× 215	ç 231	÷ 247
8	BS 8	CAN 24	(40	8 56	H 72	X 88	h 104	x 120	ˆ 136	˜ 152	¨ 168	¸ 184	È 200	Ø 216	è 232	ø 248
9	HT 9	EM 25	) 41	9 57	I 73	Y 89	i 105	y 121	‰ 137	™ 153	© 169	¹ 185	É 201	Ù 217	é 233	ù 249
A	LF 10	SUB 26	* 42	: 58	J 74	Z 90	j 106	z 122	Š 138	š 154	ª 170	º 186	Ê 202	Ú 218	ê 234	ú 250
B	VT 11	ESC 27	+ 43	; 59	K 75	[91	k 107	{ 123	‹ 139	› 155	« 171	» 187	Ë 203	Û 219	ë 235	û 251
C	FF 12	FS 28	, 44	< 60	L 76	\ 92	l 108	| 124	Œ 140	œ 156	¬ 172	¼ 188	Ì 204	Ü 220	ì 236	ü 252
D	CR 13	GS 29	- 45	= 61	M 77	] 93	m 109	} 125	141	157	- 173	½ 189	Í 205	Ý 221	í 237	ý 253
E	SO 14	RS 30	. 46	> 62	N 78	^ 94	n 110	~ 126	142	158	® 174	¾ 190	Î 206	Þ 222	î 238	þ 254
F	SI 15	US 31	/ 47	? 63	O 79	_ 95	o 111	▩ 127	143	Ÿ 159	¯ 175	¿ 191	Ï 207	ß 223	ï 239	ÿ 255

A **Character Set** defines a mapping between the bit code of a character and the display on the output device for a set of characters.
The Windows 3.1 Character set above has 256 codes – the first 128 are the standard ASCII set, while the remaining 128 have other characters.
There are a large range of Character sets available on Microsoft Word and other Document preparation systems: change the current Character set and it changes the translation between keyboard presses and text in the document. In the document created, changes of Character Set are recorded, which a very wide range of characters to be embedded in the document.

Examples

Roman-8, Windows 3.1 Latin 1, ISO 8859/1 Latin 1, PC-8 Code Page 437 (the character set used by MS-DOS on English systems), Math-8,
Wingdings Font.

The same bit pattern (code) gives different outputs depending on the character set used to map it to a display
59(hex) specifies

Y (Windows 3.0 Latin 1, and others)
✡ (Wingdings).

For output to a laser printer, the computer can select the character set, the font, the character size, the style (bold, italic)

e.g. - a, *a*, a, a, a,

and switch between them.

Output to laser printer

stream of 8-bit codes sent to it containing:

- characters to be printed
- control information

Hewlett Packard Print Control Language (HP PCL)

control sequences in PCL start with the ESC (1B)

Example hexadecimal byte sequence

1B 28 35 37 39 4C - selects the ***Wingdings*** character set
1B 28 39 45 selects ***Windows 3.1 Latin 2*** character set
1B 28 73 34 30 39 39 54 selects ***Courier*** as the typeface.

Unicode modern 16-bit code (UTF-16)

used by Java and other modern systems
code defines character set and character within set
Each character set resides in a "plane": there are 18 planes.
A Java char – 16-bits
– defines Character Set (top 8 bits) and character in set (low 8 bits)
ASCII is represented in Unicode by values 0000-007F_{16}.

The purpose of Unicode as given by the Unicode Consortium is:-

"The Unicode Worldwide Character Standard is a character coding system designed to support the interchange, processing, and display of the written texts of the diverse languages of the modern world. In addition, it supports classical and historical texts of many written languages.

Some supported scripts:
Arabic, Bengali, Bopomofo, Greek , Gujarati, Han, Hebrew, Ethiopic (16-bit codes 1200-137F), Cherokee(13A0-13FF), Katakana, Hiragana, Han ideographs, Braille Pattern Symbols(2800-28FF) and Runic (Ancient nordic script)

The 2012 current version is 6.2. There are more than 110000 characters defined covering a very wide range of languages including chinese ideographs. Since a 16 bit code only encodes 65356 symbols, 18 "supplementary planes" have been introduced: a character set resides in a plane. To select a plane and a character within a set in the plane, requires 2 16-bit codes: the first selects a plane and the second a character set and character. The plane selection does not need to be performed for every character, only when the plane needs to be changed.

The following was taken some years ago from

http://download.oracle.com/javase/tutorial/i18n/text/convertintro.html:-

Few text editors currently support Unicode text entry. The text editor we used to write this section's code examples supports only ASCII characters, which are limited to 7 bits. To indicate Unicode characters that cannot be represented in ASCII, such as ö, we used the \uXXXX escape sequence. Each X in the escape sequence is a hexadecimal digit. The following example shows how to indicate the ö character with an escape sequence:

```
String str = "\u00F6";
char c = '\u00F6';
Character letter = new Character('\u00F6');
```

Real Numbers

The power series representation of integers can be extended to provide for real numbers:-

$$R = ... + s_n 2^n + ... s_3 2^3 + s_2 2^2 + s_1 2^1 + s_0 2^0 + \mathbf{s_{-1} 2^{-1} + s_{-2} 2^{-2} + ... + s_{-m} 2^{-m} + ...}$$

The infinite series

$$\mathbf{s_{-1} 2^{-1} + s_{-2} 2^{-2} + ... + s_{-m} 2^{-m} + ...}$$

provides the fractional part of the number

$s_{-1}\ s_{-2}...s_{-m}$.. are in the range (0...b-1).

2^{-1} = 1/2 (0.5) ; 2^{-2} = 1/4 (0.25) ; 2^{-3} = 1/8 (0.125);

$$\mathbf{13.625_{10} = 1101.101_2}$$

$0.101_2 = 1*0.5 + 0*0.25 + 1*0.125 = 0.625_{10}$

As with base 10, any fractional number can be encoded to any degree of accuracy, although some numbers are more difficult to encode than others,

e.g. $1/3 = 0.33333333…..33…._{10} = -0.010101010101….01…01…_2$

Whereas 1/3 in tertiary is precisely 0.1!!!!!

2s-Complement of real numbers

Do 2's complement in usual way on all bits and add 1 into the rightmost fractional bit

Example :1.5_{10}== 1.1_2

-1.1_2 = -00001.1000_2 *2s complement*

= $..11110.0111_2$ + 00000.0001_2

= $..11110.1000_2$

Check the result:- **$\ldots000001.1_2$**
$\underline{\ldots111110.1_2}$
$\underline{\ldots000000.0}$

ADDITION OF 1 GOES INTO LEAST SIGNIFICANT BIT

Final thoughts on adding '1' when taking Complement!

Let's take an binary integer, e.g. 001101_2

But… 001101_2 = 001101.0000..000.0000……000….000…..

Taking 2s complement, but not adding 1 anywhere →

…111…10010.1111…111…1111……1111…1111….

The fractional part (0.11111…1111…) is effectively 1.

Thus we can write…111…10010.1111…111…11111…1111….

as …111…10010 **+ 1**

= …**111…10011$_2$ (the 2s complement of 1101$_2$)**

Adding a 1 when taking the complement of an integer compensates for ignoring the fractional part of the number!

0.1111……..1111.11111111111….1111111111…. to an infinite number of places is different from 1 in just the very least significant, value $2^{-\infty}$ or 0.

Fixed Point Representation of Real Numbers

This holds the n bit symbols surrounding the binary point
in an n-bit integer and uses integer ops to do maths

i.e. $\mathbf{S_{n-m-1} \bullet\bullet\bullet S_1 S_0 \cdot S_{-1} S_{-2} \bullet\bullet S_{-m}}$

Fixed point representation is usually implemented by
software on integer hardware.

The position of the fixed point is not recorded
anywhere the software has to remember where it is!

Fixed Point Numbers

Whereas an n-bit integer holds the n bit symbols to the left of the binary point,

i.e. $\mathbf{s_n ... s_1 s_0}$,

an n-bit fixed point real number holds the n bit symbols surrounding the binary point, i.e.

$$\mathbf{s_{n-m-1} \cdots s_1 s_0 \bullet s_{-1} s_{-2} \cdots s_{-m}}$$

where the number of symbols in the fractional part, m, is normally determined by the software, not by the hardware.

These bit symbols are stored such that $\mathbf{s_{-m}}$ goes into bit position 0, $\mathbf{s_{-m+1}}$ into bit position 1, etc. Because m is fixed by the software, the position of the binary point does not have to be explicitly recorded, but is built into the software.

Operations are identical to the integer operations. Fixed point numbers may be signed or unsigned.

For output to displays or printers, the different parts of the number have to extracted and separately manipulated.

Fixed point examples on 2 numbers in registers A and B

Let A hold 1.1_2 and B hold 1.0_2

Assume 12-bit fixed point with 4 fractional bits: point between bits 3 & 4

A = 000000011000 and B = 000000010000

↑ *position of fractional point* ↑

A+B =

000000011000
000000010000
000000101000
= 2.5_{10} √

A * B =

000000011000
000000010000
0000000110000000

Result of integer multiply ←

Need to shift right by 4 to lose last 4 zeroes

→ 000000011000 √

There are some points to note:

•since the position of the point is implicit, numbers being processed must have the same point position.

•when two n-bit numbers with point position, m, are multiplied with the hardware integer multiplier the 2n-bit result has point position, 2m, i.e. twice has many fractional symbols as the operands, so that the result has to be shifted right m places and the top n-m bits removed to get back to an n-bit result with point position, m.

There is a related problem with division in that the result of the integer division of 2 n-bit operands, is an n-bit result with m=0.

An n-bit fixed point representation with binary position, m, has a restricted range of values in the same way as integers. The value ranges are the corresponding signed and unsigned n-bit integers divided by 2^m.

Accuracy of fixed point:-

Example 4-bit representation with 2 bit fractional bits

3.75 (11.11) has error ±0.125, the precision is 0.25/3.75, i.e. ~1 in 16.

However 0.25 (00.01) also has error ±0.125, but the precision is 0.25/0.25, i.e. ~1 in 1.

Small numbers have a larger relative in-accuracy than larger numbers in fixed point.

Theory of fixed point numbers

Assume *m* fractional bits

Take 3 Real fixed point numbers, A, B and C with integer representations I, J, K:

e.g. A=15.25 = 01111.01 and so with m=4 and n=12, I=000011110100

Relations of A,B,C to I,J,K are:- $A = I.2^{-m}$ $B = J.2^{-m}$ $C = K.2^{-m}$

Addition of reals: $C = A + B = I.2^{-m} + J.2^{-m} = (I+J).2^{-m}$; but $C = K.2^{-m}$

therefore: $K = I + J$ *use integer addition of I and J*

Multiplication of reals: $C = A.B = I.2^{-m} . J.2^{-m} = ((I.J).2^{-m}) 2^{-m}$; but $C = K.2^{-m}$

therefor: $K = (I.J).2^{-m}$ *use integer multiplication of I and J and lose bottom m bits*

Division: $C = A/B = I.2^{-m} / J.2^{-m} = I/J = ((I/J).2^{m}) 2^{-m}$; but $C = K.2^{-m}$

therefore: $K = (I/J).2^{m}$ *use integer division and shift left m bits.*

[Note: the division process gives an integer result, e.g. 15.25/4 gives 3 and not 3.8125. A full division takes the remainder R from the division I/J and gets the integer result of $(R2^m/J)$ which is then added to K.

Let A = 15.25 and B=4. Let m, the number of fractional bits be 4.

Let n the total number of bits to be used for a number be 12.

Then A=15.25 is represented by the 12-bit value I = 000011110100, where the underlined bits are the 4 bits of the fractional part of the number.

Also, B= 4 is represented by the 12-bit value J = 000001000000 (2^6)

The addition of A and B can be calculated by straight binary addition of I and J.

C = A + B = 19.25, which is given by K = I + J, so K= 000100110100.

C=A.B = 61 = 111101, i.e 111101000 in fixed point with m=4. So C can be produced by multiplying I.J to give the 16-bit value 0011110100000000 (since J=2^6 the result is 6 zeros appended to I).

To get the correct value, K, to represent 15.25 * 4 (i.e. 111101000), we just throw away the least significant m bits of the result (i.e. 4 bits) to give K= 001111010000.

C=A/B can be produce by dividing I by J to give the 8-bit value 00000011 (3) and the 8-bit remainder 11010000 (0.8125) appropriate to 000011110100 / 000001000000.

To get the correct value, K, to represent 15.25 / 4 (i.e. 3), we just add the 4 zero bits to the right of the result to give K= 000000110000.

Another example (again with n=12, m=4) of division is 104 (011010000000) divided by 10 (000010100000).

011010000000/ 000010100000 gives the result 10 (000000001010) and the integer fixed point result of 000010100000 after shifting left 4 bits. The remainder from the division is 000001000000. Multiplying the remainder by 2^m gives (010000000000) which when divided by the original divider (000010100000) gives a result of 6 (0110) which when added to the integer fixed point result gives 000010100110: 10.375 in decimal which is much nearer the correct answer of 10.4, which cannot be precisely represented with 4 binary bits.]

Disadvantages of fixed point:

Because the position of the binary point is not recorded, it is hard to determine what is the appropriate number of fractional bits to use: to few and small fractional values are poorly represented, too many limits the maximum value that can be held, e.g. 16 fractional bits in a 32-bit representation has numbers less than 1/65536 disappearing, while the maximum value is 65536.

No hardware support besides standard ALU: all processing done in software.

Advantages of fixed point:

All processing done in software: can have very large numbers of bit, e,g 1024 bit representation with 512 fractional bits, to give very large range with very high accuracy (~ 2^{-512}).

No hardware support besides standard ALU: makes hardware cheaper and less power hungry – much audio decoding software uses fixed point because many audio decoding hardware devices have no hardware support for floating point number representations.

When the number representation only has a small number of bits, a variable position for the binary point makes better use of the available, allowing both large and small numbers to be represented: see Floating Point Numbers.

Having hardware support for a representation makes calculations and manipulations faster. Having hardware enforces a standardised representation, which means that interchanging values between software packages from different producers becomes easier.

Doing calculations in software allows flexibility of representation, more bits in the representation, leading to better precision, so is used in special situations.

You might want to encode financial data, so that numbers are held as decimal values, albeit encoded in bits, with very large numbers and precise values. This would allow numbers to be held in the same way as in the real world, making monetary calculations exactly the same as if done on a desk calculator. However, Microsoft Excel uses floating point to do calculations, while the Microsoft Calculator uses “an arbitrary-precision arithmetic library” [http://www.joelonsoftware.com/items/2007/09/26b.html].

Cobol: business language of the 1950s onwards used fixed point. Sql supports fixed point representations. PostgreSQL has a ‘numeric’ datatype that supports fixed point numbers with a 1000 digits!

Audio decoding hardware does not require fast processing, so using software to do calculations doesn’t hit performance, and the hardware is cheaper, since no maths hardware is required, and less power-hungry, as there are fewer chips to run.

Need hardware-supported, standardised real number representation with a small number of bits and variable binary point position for everyday use – floating point!

Floating Point Representation of Real Numbers

IEEE-754 floating point standard represents a real number, X, by first essentially putting X in the form:-

$$X = (-1)^S * 1.F * 2^{E-Bias}$$

What's this?

S = 0 for a +ve number, or 1 for a –ve one!
$(-1)^0 = 1$ while $(-1)^1 = -1$

***F* is all the bits to the right of the leftmost 1**

The *Bias* is fixed integer (*127*, *1023*,..)

***(E – Bias)* is the exponent**

Why this form?

S, E and F are used to represent X

Floating Point Numbers

As the name indicates the position of the binary point is not fixed in this representation, allowing a very much larger range of numbers to be represented. Obviously the most significant bits of the number, called the *mantissa*, have to be stored and the position of these bits relative to the binary point has to be known and explicitly defined within the n-bits of the representation, reducing the number of bits available for the mantissa.Thus the computer form of floating point numbers is just a variation on the standard scientific notion:-

$$\pm M * B^{\pm E}$$

where M is the mantissa and B is the number base.

Thus the values to be represented in the n-bits representing the number are:-

- the sign of the number: 1 bit
- the mantissa, M: this is the most significant v-bits of the magnitude of the number, although IEEE normalised numbers do not hold the most significant 1.
- the exponent, E: held in e-bits - must represent both +ve and -ve exponents.
- The number base,B, is implicit to the representation and does not have to be stored.

Note that 2s complement is not used in floating point.

There is one further point to clear up: the exact way that the exponent specifies the relation between the most significant bit of the mantissa and the binary point of the number. Considering a 3-bit mantissa, 1011, with exponent 2 which of the following numbers does it represent:-

0.1011, $1.011 * 2^2$, $1011 * 2^2$, 11.011

Which is correct depends on the standard used in the implementation.

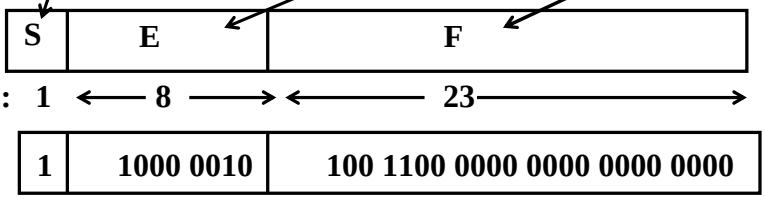
Floating Point Example : Turn -12.75_{10} into single precision form

Single precision* (Java Float): *32-bit, Bias = +127

$-12.75_{10} = -1100.11_2 = -1.10011_2 * 2^3 = (-1)^1 * 1.10011_2 * 2^{130-127}$

not 2's complement

S = 1 $E = 130_{10} = 10000010_2$ F = 1001100…00..0



➔1100 0001 0100 1100 0000 0000 0000 0000 ➔$C14C0000_{16}$

IEEE Standard for Floating Point Numbers (see http://en.wikipedia.org/wiki/IEEE_754-1985)

This standard defines 3 basic formats, **single**, **double** and **quad** precision with bit widths of 32, 64 and 128 bits respectively. An IEEE floating point number, X, is formally defined as:-

$$X = -1^S * 1.F * 2^{E-Bias}$$

Where S = sign bit (0 or 1) : S = 0 → $-1^0 = 1$ → + ; S = 1 → $-1^1 = -1$ → -

E = exponent biased by *Bias* value

F = fractional mantissa, the full mantissa is 1.F with the leading 1 implicit or *hidden.*

For 32-bit numbers, there is 1-bit for the sign, 8-bits for the biased exponent, 23-bits for the mantissa, and the bias is 127_{10}.The bias allows the exponent stored, E, to be unsigned, and has a further advantage discussed below. The standard has 3 number forms, *normalised*, *un-normalised* and *not-a-number (NaN)*, with the different forms distinguished by the value of E, the exponent field. Largest single precision value is 1.11..11 2^{127}, smallest +ve normalised value 1.0 2^{-126}

Normalised Numbers

For these numbers, it is required that the mantissa is **normalised**, so that there are no leading zeroes in the mantissa, and the mantissa is in the range $1 \le$ mantissa < 2, so that all **normalised** mantissa are of the form 1.xxxxx.The 1 is not stored, as all mantissas start with 1: this bit is called the *hidden* bit and provides an extra significant bit in the representation. The normalised form deals with the majority of numbers, positive and negative, but not zero or very small numbers: the hidden one precludes it from being used for zero.

Double precision (Java Double)

64-bit representation, 11 bit E field, 52-bit F field, Bias=1023

largest +ve value $1.1111...1 * 2^{1023}$, smallest +ve = $1.0 * 2^{-1022}$ (normalised)

Quad precision

128-bit representation, 15 bit E field, 112-bit F field, Bias=16383

largest +ve value $1.1111...1 * 2^{16383}$, smallest +ve = $1.0 * 2^{-16382}$ (normalised)

The most recent version of the standard is at http://en.wikipedia.org/wiki/IEEE_754-2008

Non-Normalised Numbers: very small numbers and zero

Representation: $X = (-1)^S * 0.F * 2^{E+1-Bias} : E=0$

E = 0 identifies a non-normalised number : F is 23-bits

Zero: $(-1)^0 * 2^{0+1-Bias} * 0.000000000$

S = 0 E = 0 =00000000_2 F = 0000000…00..0

0	0000 0000	000 0000 0000 0000 0000 0000

Floating point zero is same as integer 0!

Non-zero normalised numbers range from
$1.11111111111111111111*2^{-127}$ to $1.0*2^{-149}$
$1.11111111111111111111*2^{-127}$ is just less than smallest Normalised number

Single Precision is not very accurate: 1 in 2^{24} (1 bit in 24) : use Double!!

Non-Normalised Numbers

The *non-normalised* form has an exponent form with E=0 and deals with zero and very small numbers. There is no hidden one.

A first major advantage is that zero has sign bit S=0, exponent E=0, and mantissa of zero, so that so the representation of 0 is all bits 0. This makes floating point zero identical to integer zero. This makes it easy to set a floating point number to zero and also to test for zero.

Non-normalised single precision numbers start just below the smallest normalised number (2^{-126}) and range down to 2^{-146}. Thus numbers get progressively smaller towards zero, although the accuracy of representation, the precision, decreases also.

Purpose of bias

From the examples it can be seen that large +ve numbers have large exponents and the exponent decreases as the number represented decreases. This allows floating point numbers to be compared with standard comparison logic (provided the numbers are made +ve first). This would not have been the case if 2s complement had been used for the exponent.

Not-a-Number, *NaN*, form

The *NaN* forms provides for out of range numbers and non-real numbers, i.e. $\pm\infty$, $\sqrt{-1}$. It is distinguished by having maximum exponent: e=255 for single precision. The value placed in the mantissa field can be used to carry information on what is being represented. I have no idea if it being used in practice anywhere.

Errors introduced when Processing numbers

Floating point and fixed point representations are not necessarily precise.

If hold n bits of number, accuracy is essentially 1 in 2^n

Addition: accuracy of result is same as that of operands.

Multiplication/division: accuracy worsens by factor of 2.

Subtraction: accuracy of result can decrease without limit.

IEEE Single precision example:

A*C*D → 1100 1010 0001 0000 1101 0000 1111 : only hold top 24 bits

B*C*D → 1100 1010 0001 0000 1101 0000 0000 :only hold top 24 bits

A*C*D-B*C*D should have result 1111, but FP subtraction gives 0 as result.

Error as percentage of result is very large: always use Double!

Floating Point The accuracy of representation is limited by the number of bits in the representation:-

e.g. with a 4 bit representation the value 1111, can be thought to represent the range 14.5 to 15.5, and the accuracy is ± 0.5. For this number the precision or relative accuracy (error/value)

$= 1/15 \sim 1/16$ or $1/2^4$ (1 bit in 2^4)

Let the true value of a number be *V*, while its representation is N and there is an error in the representation of the true value of ΔN, where ΔN is always positive,

i.e. $V = N \pm \Delta N$ with precision $2\,\Delta N/N$

Assume 2 numbers:

$V_1 = N_1 \pm \Delta N_1$ with precision $2\,\Delta N_1/N_1$

$V_2 = N \pm \Delta N_2$ with precision $2\,\Delta N_2/N_2$

$\boldsymbol{V_1 + V_2} = \mathbf{N_1 + N_2} \pm \Delta N_1 \pm \Delta N_2$

with **precision** $= 2\,(\Delta N_1 + \Delta N_2)/(N_1 + N_2)$, *forming the maximum error,*

or **2 ΔN/N** assuming $\Delta N_1 \approx \Delta N_2$ and $N_1 \approx N_2$

$\boldsymbol{V_1 * V_2} = (N_1 \pm \Delta N_1) * (N_2 \pm \Delta N_2) = \mathbf{N_1 * N_2} \pm N_1 * \Delta N_2 \pm N_2 * \Delta N_1 \pm \Delta N_1 * \Delta N_2$

with **precision** $= 2\,(N_1 * \Delta N_2 + N_2 * \Delta N_1 + \Delta N_1 * \Delta N_2)/(N_1 * N_2)$

or **4 ΔN**/N assuming $\Delta N_1 \approx \Delta N_2$ and $N_1 \approx N_2$ and $\Delta N_1 * \Delta N_2 << N_1 * N_2$

$\boldsymbol{V_1 / V_2} = (N_1 \pm \Delta N_1) / (N_2 \pm \Delta N_2) = (\mathbf{N_1 / N_2} \pm \Delta N_1/N_2)/(1 \pm \Delta N_2 / N_2)$

$= N_1/N_2 \pm \Delta N_1/N_2 \pm N_1 \Delta N_2/(N_2)^2$ ignoring terms in ΔN^2 and larger powers

with **precision** $= 2\,(\Delta N_1/N_2 + N_1 \Delta N_2/(N_2)^2)/(N_1/N_2)$

or **4 ΔN/N** assuming $\Delta N_1 \approx \Delta N_2$ and $N_1 \approx N_2$)

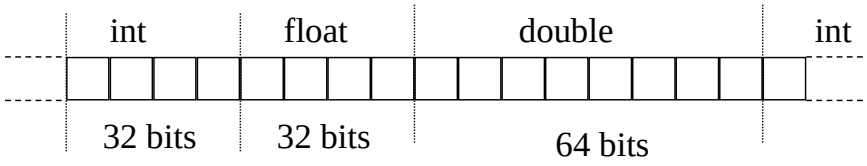
but $\boldsymbol{V_1 - V_2} = \mathbf{N_1 - N_2} \pm \Delta N_1 - \pm \Delta N_2$ with **precision** $= 2\,(\Delta N_1 + \Delta N_2)/(N_1 - N_2)$

and so with $\Delta N_1 \approx \Delta N_2$ and $N_1 \approx N_2$ **the precision** $\rightarrow \infty$

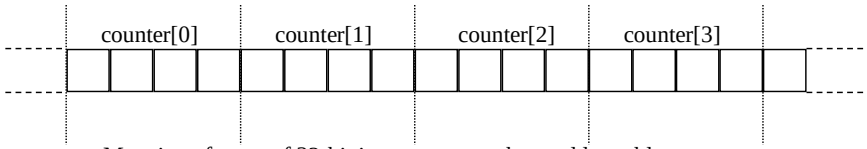
Subtracting two floating point large numbers can be lead to large relative errors.

For a n-bit representation the precision for a floating point can be considered to be 1 bit in 2^n

Single datatypes mapped on to 8-bit memory locations



int counters[20] ;



Mapping of array of 32-bit integers on to a byte-addressable memory

Arrays

organised collection of a number of instances of a single datatype, with a (simple) numerical referencing system for accessing a single element,

e.g. 1-dimensional arrays, 2-dimensional arrays, etc.

Languages such as C++ and Fortran provide for multi-dimensional arrays, and the compiler has to map these into the memory locations.

1-dimensional Arrays, e.g. int counters[20] ; // *counter[0]* to *counter[19]*

Accessing an element in the counter array with the MIPS:-

// get address of counter[2] into register $6 – where address of start of array is represented by the 'label' "counter".

// get most significant 16-bits of address into register *$1*

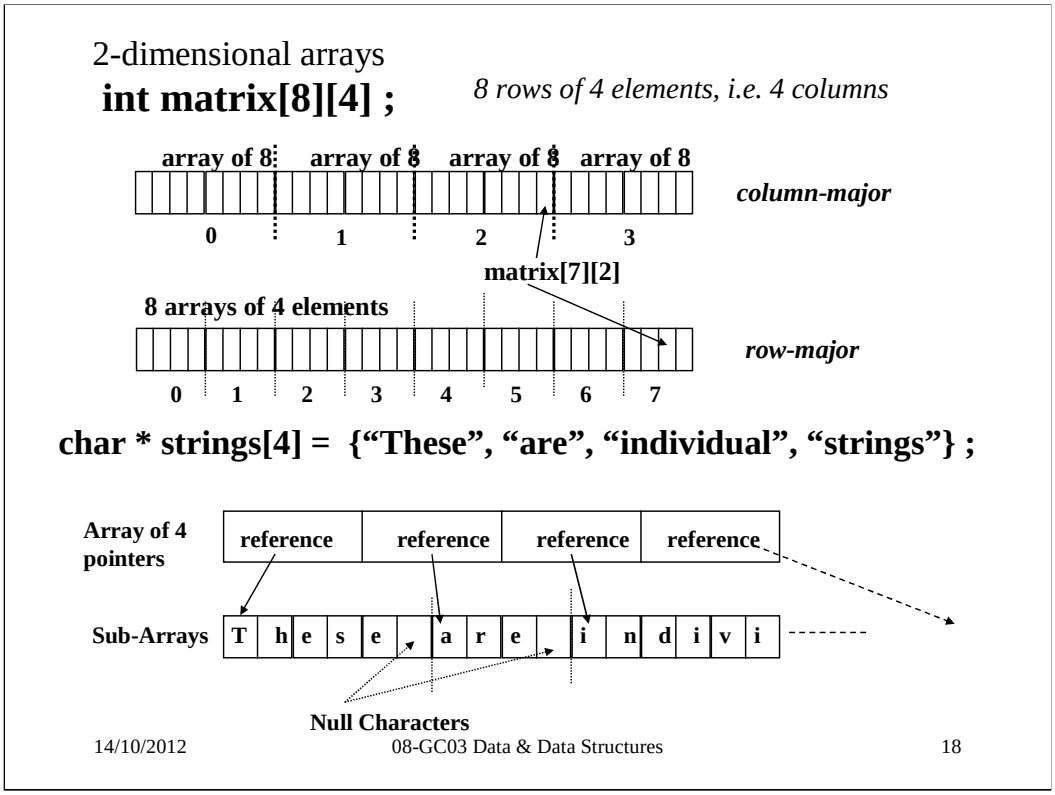
```
        lui     $1,             #counter>>16
```

// or in least significant bits of address

```
        ori     $1,             #counter&0xffff
```

// read counter[2] – counter[2] is 8 bytes down array

```
        lw      $s6,            8($1)
```



2-dimensional Arrays

An example from C of a 2-dimensional array declaration is:-

int matrix[8][4] ;

I believe that the standard understanding of this is that there are 8 rows with 4 elements each!!

This matrix can be held in memory in ***column major*** form as ***4 1-dimensional arrays of 8*** - each section of 8 holds the data of a column, or ***row major*** form as ***8 1-dimensional arrays of 4*** - each section of 4 holds the data of a row!

A key feature of the **matrix** array and its mapping on to memory is that all the sub-arrays are of the same size, and it is this that allows a regular memory structure, and it is easier to locate element. Matrix(m,n) is in position n*4 +n of the memory in the column major form, while it is position 8*m +n for row major form. These expressions assume row and column numbers start at 0, I.e. rows 0-7, columns 0-3.

An example of a 2-dimensional structure in which the sub-arrays are not all the same size is an array of strings in a C program, e.g.

char * strings[4] = {"These", "are", "individual", "strings"} ;

[Java: String[] strings = {"These", "are", "individual", "strings"} ;]

This is implemented as an array of pointers, with the references identifying where in the memory the strings are held. It is done this way so that finding one of the elements of the array is easy.

Access to the 3rd element of *strings*, e.g. string[2], in MIPS assembler:-

```
// get most significant 16-bits of address into register $1
   lui     $1,   #strings>>16
// or in least 16 significant bits of address - $1 has address of start of 1-dimensional array of references
   ori     $1    # strings &0xffff      // read strings[2] – strings[2] is 8 bytes down array
   lw      $8,   12($1)                  // Register $8 has address of start of strings[2] – "individua……
```

C/C++ Structures

A typical example of an instance of a C/C++ structure is:-

```
struct car {
    double wdth ;
    double lngth ;
    int wheels ;
    int seats ;
};
struct car triumph_tr4A ;
triumph_tr4A.wdth = 1.6, triumph_tr4A.seats = 2
```

Definition of struct

A structure is a bit like a class except there are no methods and everything is public.

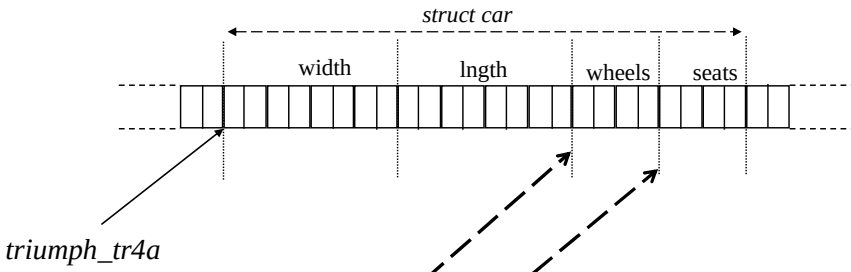
*Instantiation of struct **car** with name **triumph_tr4A***

Reference to data item within struct

Other instances of car can now be made, e.g. :-

```
struct car ford_prefect ;
```

Mapping of instance of *struct car* to memory:-

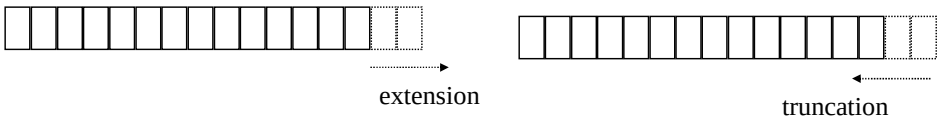


If the address of the start of the *struct* is held in $8, then the following *lw* instructions access the elements wheels and seats

```
triumph_tr4a.wheels:    lw    $4,   16($8)
triumph_tr4a.seats:     lw    $5,   20($8)
```

Stacks, Queues, Trees

A Stack: a 1-dimensional structure which is modified dynamically at one end only, by adding or removing elements from this end.



Items are *pushed* on to the stack

Items are *popped* off the stack

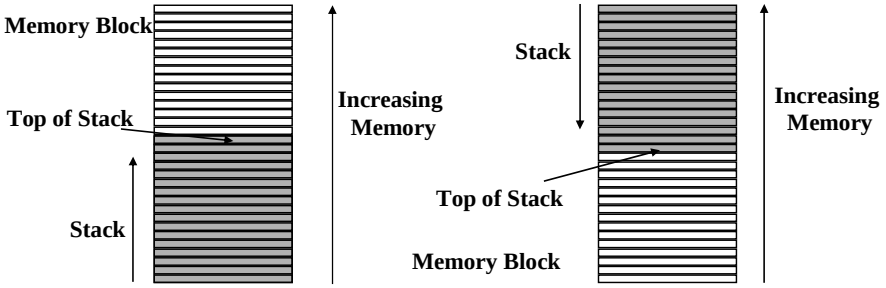
A stack is an example of a LIFO system: last-in-first-out

The last value input (i.e. the most recent) is the first value removed.

Stacks

A stack is so named, because its operation is similar to the stacks of everyday life (stacks of plates, books, papers, cans) that we regularly use: you stack plates one on top of the other and remove them again from the top. For a stack in a computer, new elements are *pushed* on to the *top of the stack* and *popped* from the *top of the stack*. An element can only be removed from the top of the stack; elements cannot be removed from the middle of a stack. Although as we shall see, the contents of elements within the stack can be modified..

Mapping of stacks to memory



Stack that grows upwards through the memory block

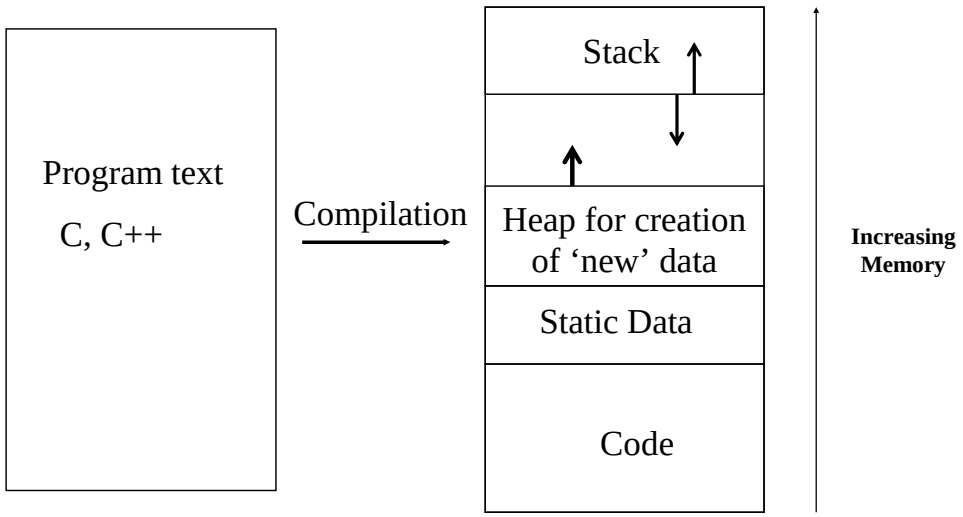
Stack that grows downwards through the memory block

A stack can obviously be mapped on to a block of memory locations which it uses as necessary. Provided the size of the block of memory is larger than the maximum size that the stack reaches, then the stack will use more or less of the allocated memory block as it grows and shrinks. This is the nice feature of a stack; it reuses the same memory locations again and again, as it grows and shrinks, and all that needs to be tracked is the position of the top of the stack.

The slide shows 2 different ways of implementing a stack on a block of memory: on the left the stack *grows upwards* through the block with the bottom of the stack at the *lowest memory address*, while on the right the stack *grows downwards* with the bottom of the stack at the *highest memory address*. Both forms operate successfully. User defined stacks can be implemented in either fashion, but the CPU-supported stack usually grows downward through memory. Regardless of the implementation, the top of the stack is the end which is modified.

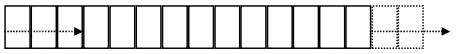
Compiler has model of the memory space that it use to compile a program

Memory Model



Stacks grow downwards in all the computers I can remember.

A Queue is a 1-dimensional structure which can be dynamically modified by *extension at one end* and *truncation at the other*.



truncation extension

A queue is an example of a FIFO system: first-in-first-out

The first value input (or the oldest value in the queue) is the first removed.

Queues

A queue is so named, because its operation is similar to the queues of everyday life.

You join one end of a supermarket check-out queue and leave at the other end after paying.

For a queue in a computer, new elements are only *added* at the *back of the queue* and only *removed* from the *front of the queue*. Elements are not removed in general from the middle .

In computer systems as in everyday usage, *head* and *tail* are often used instead of *front* and *back*: head normally refers to the front of the queue, where elements are removed, and tail to the back where elements are added.

A queue cannot be mapped on to a block of memory locations as simply as a stack can.

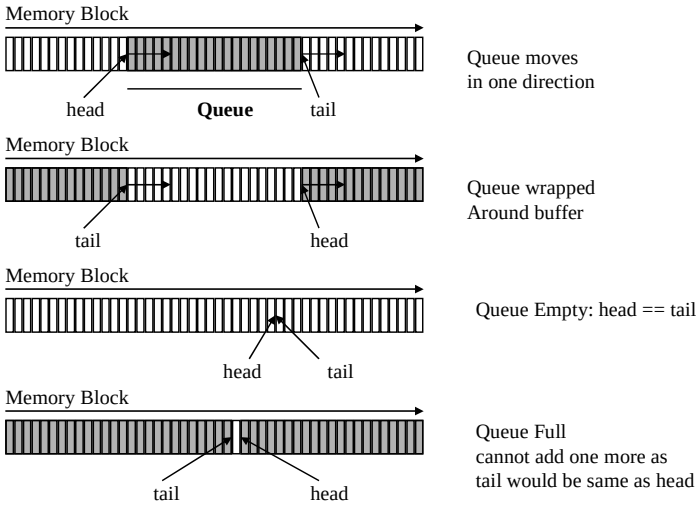
The problem is that **both ends of the stack move in the same direction**, so that the queues only move in one direction through a block of memory,

(unlike a stack which moves back and forward over the same memory locations.)

A solution

- wrap the queue around the memory block,
- when the either head or tail reaches the end of the memory block it moves around to the beginning of the block: **a circular buffer**.

Queues: based on circular buffer

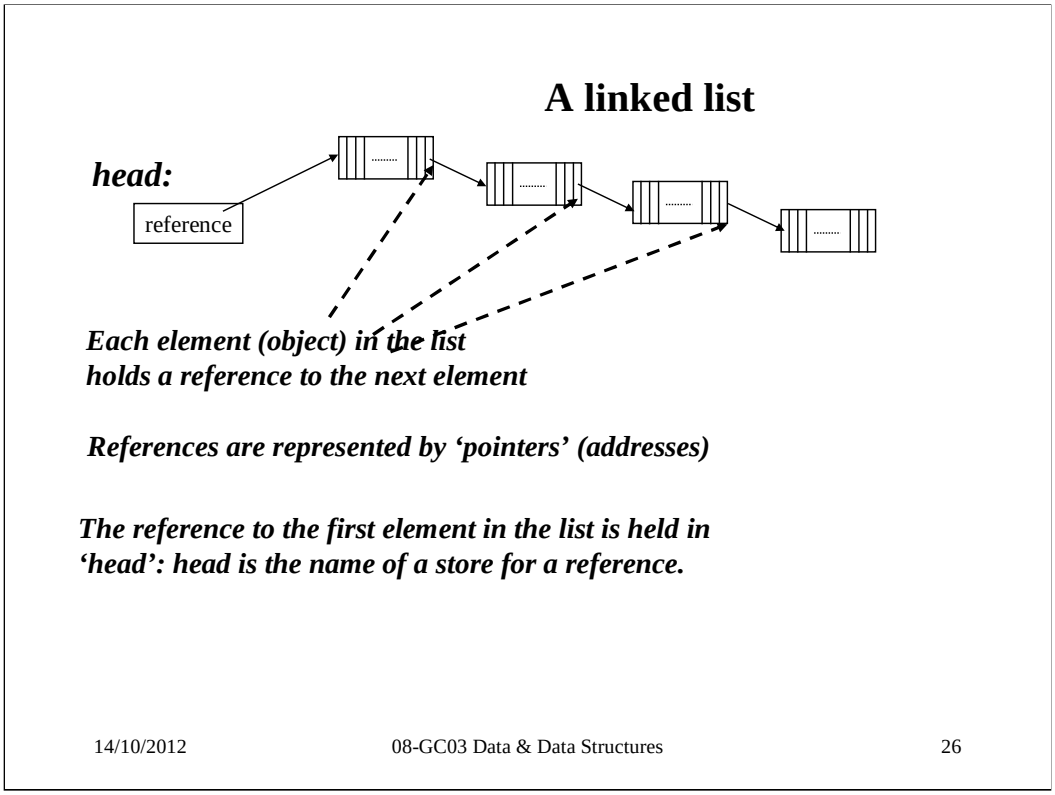


The diagram shows a queue in a circular buffer:-

- new elements added to the tail and the oldest elements removed from the head.
- When the head and tail references (pointers) are the same the queue is empty;
- when the tail is one behind the head the queue is full.
- When either head or tail is modified, there must be a check to see if they have reached the end of the block in which case they must be must be reset to the beginning.

Queues such as this are frequently used in situations where a *producer process* places data into the queue from where it is removed by a *consumer process*.

Input to and output from a computer often requires queues to hold data.



Linked Lists and Trees

Dynamic data structures can be built from units of C structures by the use of pointers.

There is a further necessity to have a capability to create 'new' units (objects) and for this a *heap* is required, which was mentioned earlier.

The figure shows a singly-linked list:-

- Each block represents a single unit of the linked list.
- Each individual unit is a struct or object of some sort.
- Each unit occupies a continuous block of memory, but the memory blocks for individual units are unrelated.
- Each unit has an element that holds a *next* pointer to the next element in the list.
- A pointer to the first unit in the linked list is held in a fixed location, the head pointer.
- The *next* pointer in the last unit in the list has a special *null* value in it to mark the list end.
- A pointer is a 'reference'. It is the address of the element pointed at.

To extend the list, a new unit has to be constructed in a free block of memory, and added to the list in the right way.

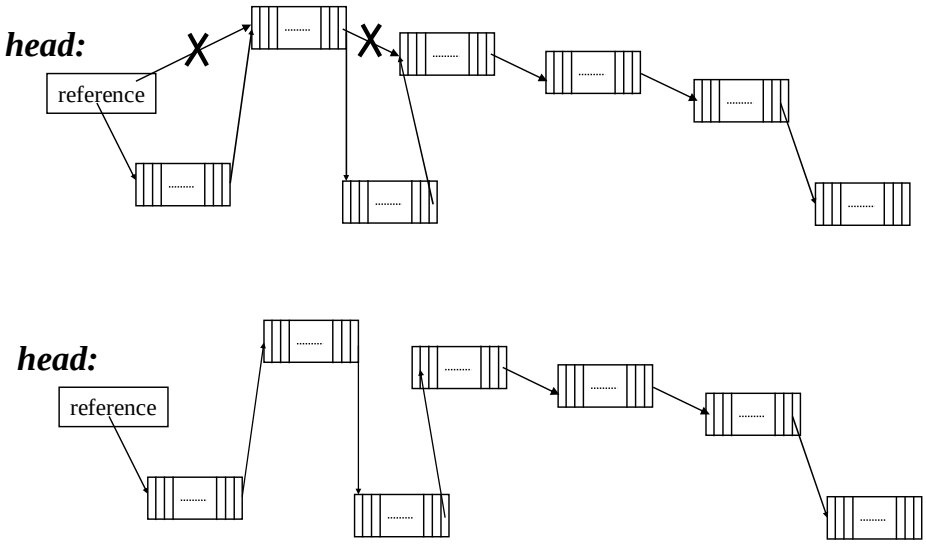
Many operating system provide a *heap*, which essentially is a large block of unallocated memory from which sub-blocks can be allocated on demand.

C provides a function call *malloc* (short for ***m***emory ***alloc***ation) which can be called with a request for a block of memory of a particular size and returns a pointer to a block of memory of this size. Java has 'new' which does the same thing, but all the messy details are hidden!!

The value of a linked list is that its size adjusts to the amount of data to be stored, unlike static data structures, so that it is efficient in its use of memory.

[If it is not known when a program is written how much data has to be stored, then an array is difficult to use (how long should the array be: 100 elements, 1000 element, 10000000 elements?), whereas a linked list will handle any amount of data up to the limit of available memory.]

Inserting new elements into the list



Once the new unit has been obtained it can be added to the linked list by manipulating the contents of the next pointers: at the beginning, at the end or even somewhere in the middle of the list.

The Vector class in Java could be implemented in this way.

Vector vec = new Vector() ;

If I was writing Vector, I would have the element referenced by ‘head’ as the last element in the vector.

Then,

vec.add(*element*) ;

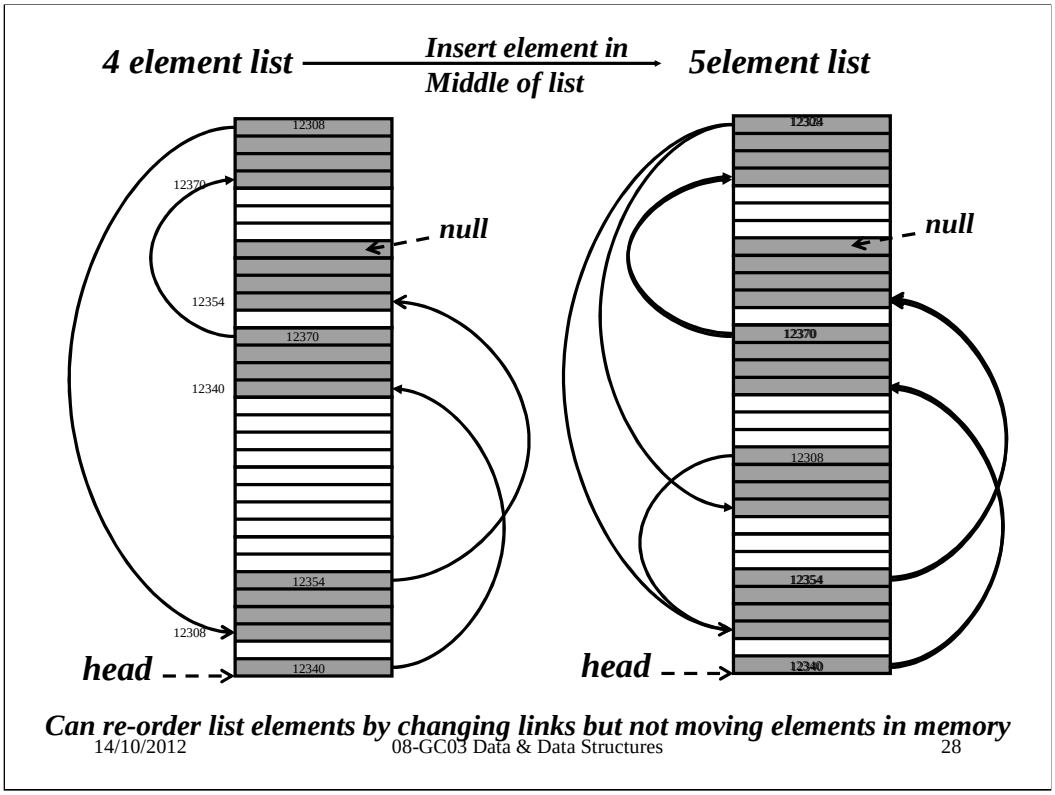
which adds *element* to the end of the Vector, would require *element* to be added at the head of the linked list, and not right down the end of the list, to get at which I have to traverse all the rest of the elements in the list!

Similarly

vec.remove()

would be quick, too!

The *insertElementAt* and *removeElementAt* operations of Vector would map on to inserting and removing elements from within or at the end of the list.

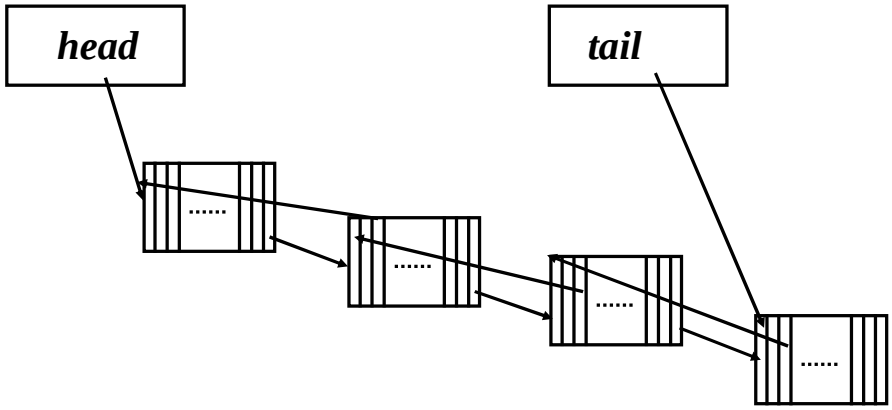


Can re-order list elements by changing links but not moving elements in memory

A possible Java version of a linked list

```
Public class Element {
// some data in object
public int value ;
// next element in list
Element next ;
Public Element() {
                    value = 0 ;
                    next = (Element) null ;
}
public void insertElementAtEnd(Element e) {
                    if (next == null) {
                                        next = e ;    return ;
                    } else {
                                        next.insertElementAtEnd(e) ;
                    }
}
public void insertElementOnCondition(Element e) {
                    if (next == null) {
                                        // if at end of list insert ;
                                         next = e ;  return ;
                    } else {if (e.value > next.value) {
                          // pass Element e to next Element in list if value in e is > than value in next Element
                                         next.insertElementOnCondition(e, value) ;
                    } else {
                         // inset Element e after this Element but before next Element
                        // 1st take Element from next and place in next of Element e
                                        e.next = next ;
                       // now place Element e inn this Element's next
                                        next = e ;
                    }
// remove method etc follow
}
```

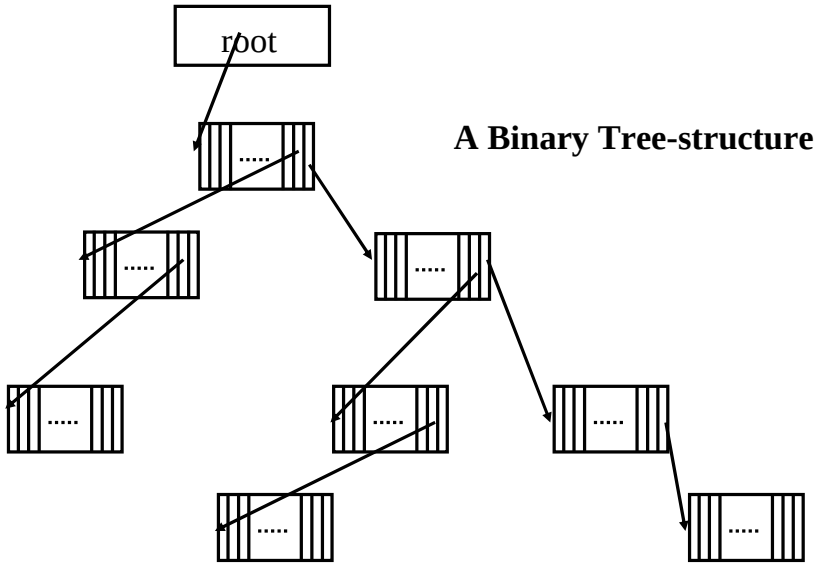
A double-linked list with *tail* pointer



The *next* field in the last element is *null*

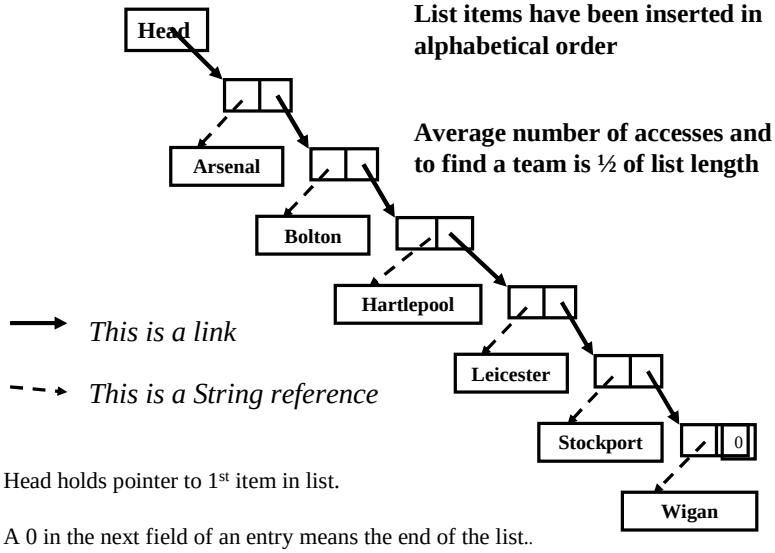
The *previous* field in the first element is *null*

more pointers…….



A Binary Tree-structure

Linked list of Football teams



List items have been inserted in alphabetical order

Average number of accesses and compares to find a team is ½ of list length

This is a link

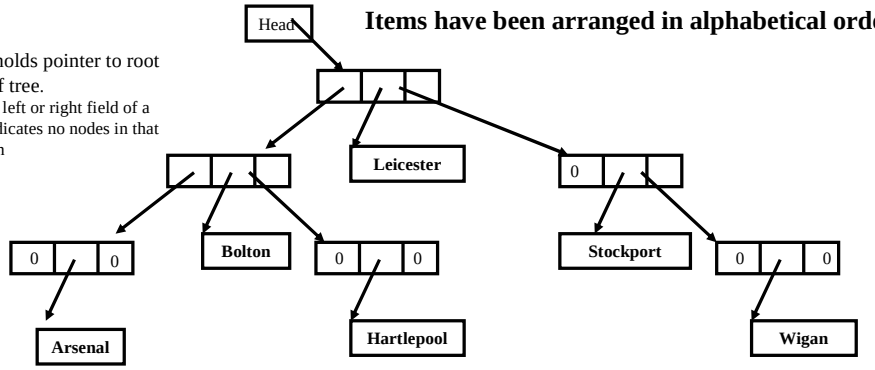
This is a String reference

Head holds pointer to 1st item in list.

A 0 in the next field of an entry means the end of the list..

Binary Tree of Football team names

Head holds pointer to root node of tree.
A 0 in a left or right field of a node indicates no nodes in that direction



Items have been arranged in alphabetical order.

On search take left branch if name is less than name in node, right if greater, stop if same

Try searching for Hartlepool: left at root node, right at next node.

Average number of accesses and compares to find a team is roughly the tree height.

An arbitrary Graph-structure