1 Value representation

- All primitives have internal well-defined representations in binary.
- Java provides 5 different primitive types to represent integral values: byte (8 bits), char (16 bits), short (16 bits), int (32 bits), and long (64 bits).
- All of these types are signed except char, meaning they represent positive and negative values from $-2^{N-1}$ to $2^{N-1} - 1$, where N is their length in bits.
- Unsigned types such as char represent values from 0 to $2^N - 1$.
- Other languages allow specification of unsigned vs. signed.

2 Operations on integers

- Assignment of a value greater than the possible maximum value causes overflow – the value “wraps around”.
- For signed types, the value after the greatest value stored is the smallest value stored (the most negative).
- For unsigned types, the value after the greatest value stored is zero.

Example

```java
byte b = 127;
b = b + 1;
(b == -128) // true
```

3 Binary value representation

- For char it’s simple: bit N represents $2^N$.

Example

```java
char c0 = 0; // represented by 00000000000000002
char c1 = 17; // represented by 00000000000100012
char c2 = 65535; // represented by 11111111111111112
```

- For all other value types, the sign of the value must be stored.
- The leftmost (Nth) bit is the sign bit.

Example
4 Addition with signed types

- The above 2’s complement representation for signed values makes addition easy
- Negative numbers are added as if they were positive, and the result is just treated as a negative number afterwards depending on the sign bit
- When performing addition, all bits at positions greater than N are truncated – this leaves just the representable portion of the number

Example

byte b0 = 1; // represented by 00000001₂
byte b1 = -1; // represented by 11111111₂
byte b2 = b0 + b1; // represented by 10000000₀₂ → 00000000₂

5 Conversion

- In Java, integer primitive types can be converted using casting
- A cast from a type with more bits to one with fewer truncates the binary value to fit in the result type
- A cast from a type with fewer bits to one with more is called a promotion – we “sign-extend”, that is, fill in all the bits of the result type with the sign bit

Example

byte b0 = 2; // represented by 00000010₂
short s0 = b0; // represented by 0000000000000010₂
byte b1 = -2; // represented by 11111110₂
short s1 = b1; // represented by 1111111111111110₂

6 Bit manipulation

- In addition to manipulation of integer types as their logical (decimal) values, in Java and related languages, integer types can be treated as a sequence of bits that can be directly manipulated
- “Bitwise” operators support AND, OR, XOR, NOT, SHIFT LEFT, and SHIFT RIGHT
- NOT “~” is unary and flips all bits of its operand
AND &, OR |, and XOR ^ are binary operators that perform their operations on each bit of the two operands to produce the bit at the same place in the result

SHIFT LEFT << is an operator that takes a value to shift as the left operand and the number of bits by which to shift, N, as a right operand and moves the bits in the left operand N places to the left, extending 0s on the right and truncating bits on the left

SHIFT RIGHT (logical) >>> works the same as SHIFT LEFT except bits are shifted to the right and 0s are extended on the left and bits on the right are thrown away

SHIFT RIGHT (arithmetic) >> works like SHIFT RIGHT (logical) except the sign bit is extended on the left rather than 0s

Example
01101001₂ &
11001010₂ =
01001000₂

01101001₂ |
11001010₂ =
11101011₂

01101001₂ ^
11001010₂ =
10100011₂

~ 01101001₂ =
10010110₂

01101001₂ <<
3 =
01001000₂

11001010₂ >>
3 =
11111001₂

11001010₂ >>>
3 =
00011001₂

7 Announcements

- Homework 4 and Project 2 are up
- Feedback on feedback
- Reading: Programming Into Java – 5.3; Goodrich and Tamassia – 11.4
8 Data compression

- Suppose we have the text string:
  \textit{she sells sea shells by the seashore}
- Let’s get rid of the redundancy by replacing repeated sequences with codes that represent them:
  \textit{0e s1 2 01 by the 2shore}
- Our string is condensed, but if given that string, there would be no way to reproduce the original
- A mapping of symbols (the original strings) to codes (their compressed representations) is needed:
  \textit{0 → sh, 1 → ells, 2 → sea}
- With the mapping, we can replace each code in the compressed string with its corresponding symbol,
  reproducing the original string, so no information is lost
- Given input data with enough redundancy, the wasted space from including the mapping is outweighed
  by the gains from compression
- How do we come up with a good mapping?

9 Huffman codes (fixed length)

- For fixed length symbols, count the frequency in the input and produce a frequency table
- Produce a binary tree that represents prefix-free codes of the input
- Replace each symbol with its code to produce compressed output

\textbf{Huffman coding}

For symbols of length L bits, produce a frequency table
Build N tree nodes corresponding to the N distinct symbols
Place the N distinct symbol trees into a minimum priority queue
While there are more than 1 tree:

- Remove the two symbol trees with the lowest priority (frequency)
- Merge them into one symbol tree
- Reinsert the merged tree into the priority queue

\textbf{Example}

Input:
\textit{she sells sea shells by the seashore}

Symbol length: 16 bits (one Java \texttt{char})

Frequency Table:
\begin{verbatim}
char: a b e h l o r s t y , '
freq: 2 1 7 4 4 1 1 8 1 1 6
\end{verbatim}
Once the tree is built, codes can be read by traversing the tree, with left branches as 0 bits and right branches as 1.

Since the code is prefix-free, when reading a bit stream, we can immediately decode a sequence of bits (since any sequence of bits read from the compressed data that matches a code must correspond to its known symbol, since no code is a prefix of another code).