# Dualzahlen im OOP-Kontext

Wir wollen eine abhängige Klasse "Dualzahl" implementieren.

Im Hauptprogramm wollen wir solche instanziieren u. damit arbeiten.

Darstellung:

Wie stellen wir eine Dualzahl da?

Als Feld von booleschen Elementen.

Wir wählen die Größe 8 für eine Dualzahl der Größe ein Byte.

Da dies Größe in der Informatik so etabliert ist, brauchen wir sie nicht flexibel zu halten.

Zur Zuordnung von 1 u. 0 zu true u. false benutzen wir die "positive Logik":

1 entsprich true, 0 entspricht false, wie auch z.B. in C festgelegt.

Der Benutzer soll nur mit 0 u. 1 arbeiten.

Er weiß weder, was eine boolesche Variable ist noch ein Feld.

Über die Operationen, die wir dem Benutzer anbieten, müssen wir die interne Realisierung kapseln.

Unsere abhängige Klasse nennen wir "Dualzahl".

Sie besteht aus der zentralen Datenstruktur eines booleschen Feldes der Größe 8.

Deklaration des booleschen Feldes:

boolean[] dual = new boolean[8];

Operationen:

Weitere Bestandteile der abhängigen Klasse: diverser Operationen auf Dualzahlen:

* Erzeugung mit selbst definierten Konstruktoren
* Formatierte Ausgabe
* Umwandlung dual->dezimal u. umgekehrt
* Addition von Dualzahlen

Ausgabe:

Reihenfolge:

Bisher haben wir Felder von 0 bis Feldgröße-1 von links nach rechts durchnummeriert.

Da einer Dualzahl Dualzahl, wie auch einer Dezimalzahl, ein Stellenwertsystem zugrunde liegt,

das links die größeren Wertigkeiten hat als rechts, spiegeln wir die Reihenfolge:

Index 0 steht rechts, Index 7 links.

Wir führen die Begriffe ein:

Das höchstwertige Bit, das ganz links mit Index 7, heißt: "most significant bit": MSB,

das niedrigwertigste Bit, das ganz rechts mit Index 0, heißt: "least significant bit": LSB

MSB LSB  
 boolean

dual:

Index i: 7 6 5 4 3 2 1 0

Wenn wir die Feldinhalte direkt ausgeben würden, bekämen wir boolesche Elemente,

also "false" u. "true".

In einer Methode "schreibe\_Bit()" können wir die Zuordnung zu 1 u. 0 vornehmen

und die Ausgabe delegieren,

außerdem einen Tab zwischen jede Ziffer zur Übersichtlichkeit schreiben.

Wir wollen die 8 Bits in zwei Vierergruppen unterteilen,

mit einem Extra-Tab nach dem 4. u. vor dem 3. Bit.

Wie können wir das bewerkstelligen?

Version 1:

Wir prüfen in der Schleife mit einer bedingten Anweisung,

ob wir bei i=4 sind u. schreiben in diesem Fall den Extratab:

for (int i=7; i>=0; i--){

schreibe\_Bit(dual[i]);

if (i==4)

System.out.print ("\t");

}

Bewertung:

+ kurzer Quelltext

- unnötigerweise 8 mal prüfen einer Bedingung => lange Laufzeit

Version 2:

for (int i=7; i>=4; i--)

schreibe\_Bit(dual[i]);

System.out.print ("\t");

for (int i=3; i>=0; i--)

schreibe\_Bit(dual[i]);

Bewertung:

- längerer Quelltext

+ kürzer in der Laufzeit

- Codeduplizierung: Zwei fast identische Schleifen:

Gleicher Block,  
 Unterscheidung nur in den Grenzen, von wo bis wo durchlaufen werden soll.

Version 3:

Bessere Lösung:

Ich definiere mir eine Methode, die das macht,

was die beiden Schleifendurchläufe auch getan hätten,

aber übergebe die beiden differenten Werte als Parameter.

So vermeide ich sowohl eine Codeduplizierung als auch unnötige Abfragen:

schreibe\_halbes\_Byte(7,4);

System.out.print ("\t");

schreibe\_halbes\_Byte(3,0);

public void schreibe\_halbes\_Byte(int von, int bis)

for (int i=von; i>=bis; i--)

schreibe\_Bit(dual[i]);

Bewertung:

+ keine unnötigen Abfragen

+ keine Codeduplizierung

- länger im Quelltext

Konstruktoren:

Wenn ich den Benutzer mit dem Standardkonstruktor eine Dualzahl erzeugen würde, müsste er

* die Bits einzeln belegen, also 8 (!) Zuweisungen für ein Byte
* die Implementierung kennen als Feld
* wissen, wie man Feldelemente adressiert u. lesend wie schreibend zugreift
* wissen, was boolesche Elemente sind
* ob die positive oder negative Logik zugrunde liegt.

Das ist für den Benutzer unzumutbar.

Deshalb wollen wir einen Satz von Konstruktoren bereitstellen,

mit denen er nach Belieben u. mit möglichst wenig Aufwand,

nur unter Übergabe von Nullen u. Einsen eine Dualzahl erzeugen kann.

Wir stellen gegenüber, welche Bitfolgen wir häufig brauchen

sowie wie wir Konstruktoren definieren können:

* Am häufigsten brauchen wir den "Nullvektor", die Folge von 8 Nullen
* Danach den "Einsvektor", die Folge von 8 Einsen
* Schließlich müssen wir auch eine Bitfolge beliebig definieren können,  
  z.B.: 0 1 0 0 1 1 0 1

Was für Konstruktoren können wir definieren?

* Der einfachste wäre der ohne Parameter: Dualzahl()
* Der nächste der mit genau einem, einer 0 oder einer 1:   
  Dualzahl(0) oder Dualzahl(1)
* Für die beliebige Bitfolge werden wir alle 8 Werte einzeln entgegen nehmen müssen:  
  Dualzahl(0,1,0,0, 1,1,0,1)

Es liegt folgende Zuordnung nahe:

Die Bitfolge, die wir am häufigsten brauchen, den Nullvektor,

erzeugen wir mit dem einfachsten Konstruktor, dem ohne Parameter:

Dualzahl d1 = new Dualzahl(); => Nullvektor

Als zweithäufigstes brauche ich den Einsvektor.

Als nächsten einfachsten Konstruktor wäre der,

bei dem ich nur genau eine Zahl übergebe.

Dafür liegt nahe zu definieren:

Wenn ich einen Konstruktor aufrufe mit nur einem Parameter,

soll dieser in alle 8 Bits geschrieben werden:

Dualzahl d1 = new Dualzahl(1); => Einsvektor

Wenn wir konsequent sein wollen,

würde die Übergabe einer Null dann auch den Nullvektor erzeugen,

so wie auch schon der Konstruktor ohne Parameter:

Dualzahl d1 = new Dualzahl(0); => Nullvektor

An der Stelle müssen wir uns über Fehleingaben des Benutzers nachdenken:

Was tun wir, wenn nicht eine 0 oder 1 kommt, sondern ein anderer Wert?

Möglichkeit 1: Fehlermeldung u. Setzen auf Fehlerwerte, naheliegend: 0

Möglichkeit 2: Keine Fehlermeldung, sondern von uns definierte implizite Fehlerbehandlung,  
wie oben: ungültigen Wert auf 0 setzen.

Wir entscheiden uns für Möglichkeit 2:

Dualzahl d1 = new Dualzahl(17); => Nullvektor

Implementierung der impliziten Fehlerbehandlung:

Wir unterscheiden bei der Übergabe eines Parameters nun drei Fälle:

Fall 1: Null übergeben => Bit auf false

Fall 2: Eins übergeben => Bit auf true

Fall 3: Unsinn übergeben => Bis auf false

Angenommen, wir bekommen einen int übergeben: "bit"

Version 1:

if (bit==0) // Fall 1  
dual[i] = false;

else if (bit==1) // Fall 2  
 dual[i] = true;

else // Fall 3  
dual[i] = false;

Uns fällt auf, dass in den Fällen 1 u. 3 die Anweisung gleich ist.

Deshalb fassen wir beide Fälle zusammen u. können damit auf eine  
nur zweiseitig bedingte Anweisung reduzieren:

Version 2:

if (bit==1) // Fall 2  
 dual[i] = true;

else // Fälle 1 u. 3  
 dual[i] = false;

Wir wollen noch mehr vereinfachen.

Um den Schritt zu sehen, abstrahiere ich von der Bedingung u. dem Operanden:

if (<Bed>) // Fall 2  
 <Element> = true;

else // Fälle 1 u. 3  
 <Element> = false;

Anders formuliert:  
Wenn eine Bedingung true ist, setze ich ein Element auf true;  
wenn die gleiche Bedingung false ist, setze ich das Element auf false.

Küzer:

Ich setze das Element auf das Erebnis der Bedingung:

<Element> = <Bed>;

Umwandlung einer Dualzahl in eine Dezimalzahl

Gegeben ist eine Dualzahl von 8 Bits.

Gesucht ist der dezimale Wert:

z.B. 0100 11012 = ?10

In einem Stellenwertsystem hat jede Stelle eine Wertigkeit.

Wie sie sich unterscheiden, hängt von der Basis ab:

Im Dezimalsystem mit Basis 10 ist eine Stelle weiter links das 10-fache,

im Dualsystem mit Basis 2 das 2-fache "Wert".

Es besteht folgender Zusammenhang:

In Indexstelle "i" ist die Wertigkeit " Basisi ":

Hier im Dualsystem: 2i

MSB LSB  
 boolean

dual:

Index i: 7 6 5 4 3 2 1 0

Wertigkeit 2i: 27 26 25 24 23 22 21 20

128 64 32 16 8 4 2 1

Algorithmus zur Umwandlung einer Dualzahl in eine Dezimalzahl:

Gegeben ist eine Dualzahl, gesucht ist der dezimale Wert davon.

Wir definieren eine int-Hilfsvariable "dezi", die wir mit 0 init.

Dann durchlaufen wir die Bits des Bytes,

hier wahlweise von oben nach unten oder von unten nach oben.

Steht an der aktuellen Stelle im Feld ein "true",

addieren wir die Wertigkeit der Stelle zu dezi.

Steht ein false, tun wir nichts.

Haben wir das Byte als Ganzes durchlaufen,

steht in der Hilfsvariable der dezimale Wert der Dualzahl.

Beispiel: 0100 11012 = ?10

MSB LSB  
 boolean

dual: 0 1 0 0 1 1 0 1

Index i: 7 6 5 4 3 2 1 0

Wertigkeit 2i: 27 26 25 24 23 22 21 20

128 64 32 16 8 4 2 1

dezi: 64 + 8 + 4 + 1 = 7710

Antwort: 0100 11012 = 7710

Dies implementieren wir (vorläufig) mit der Hilfmethode "zweihoch()",

die wir als Übung für die Infixnotation erarbeitet haben.

In Pseudocode:

Gegeben sei eine Dualzahl "dual" als Feld von booleschen Elementen.

Gegeben sei weiter eine Hilfmehtode "zweihoch()",

die für einen Ganzahl-Parameter "n" 2n zurück gibt.

Initialisiere eine Ganzzahl-Hilfsvariable "dezi" mit 0.

Laufe über Ganzzahl-Variable "i" von 0 bis 7:

Steht an dual[i] ein "true",

addiere die Wertigkeit der Stelle i,

nämlich zweihoch(i) zur Hilfsvariablen "dezi"

In der Hilfsvariablen "dezi" steht nun der dezimale Wert

der Dualzahl "dual".

**public** **int** dual\_zu\_dezi() {

// Berechnet den dezimalen Wert der Dualzahl,

// mit der sie aufgerufen wurde

**int** dezi = 0;

**for** (**int** i=0; i<8; i++)

**if** (dual[i])

dezi += zweihoch(i);

**return** dezi;

}

**public** **int** zweihoch(**int** n) {

**int** ergebnis=1;

**for** (**int** i=1; i<=n; i++) // i:1..n

ergebnis \*= 2;

**return** ergebnis;

}

Wir berechnen hier für jede Stelle die Wertigkeit als Zweierpotenz neu.

In der eigentlichen Methode laufen wir von 0 bis 7,

um in der Hilfsmethode für jeden Aufruf von 1 bis n zu laufen.

Damit haben wir eine quadratische Laufzeit: O(n2).

Das lässt sich reduzieren, indem wir berücksichtigen,

dass die Wertigkeit einer Stelle weiter rechts,

also des nächst größeren i,

genau das Doppelte der aktuellen ist:

2n+1 = 2n \* 2

Dafü führen wir eine Hilfsvariable "wertigkeit" ein,

die wir mit der Wertigkeit des LSBs, also "1" initialisieren,

für die Berechnung verwenden u. danach mit 2 multiplizieren,

weil die Wertigkeit des nächst größeren i

genau das Doppelte der bisherigen ist.

Somit können wir mit nur einer Multiplikation

die Wertigkeit der nächsten Stelle ermitteln,

auf die Hilfsmethode verzichten u. dadurch die Komplexität

von O(n2) auf O(n) reduzieren!

**public** **int** dual\_zu\_dezi() {

// Berechnet den dezimalen Wert der Dualzahl,

// mit der sie aufgerufen wurde

**int** dezi = 0, wertigkeit=1;

**for** (**int** i=0; i<8; i++) {

**if** (dual[i])

dezi += wertigkeit;

wertigkeit \*= 2;

}

**return** dezi;

}

Umgekehrte Richtung:

Umwandlung einer Dezimalzahl in eine Dualzahl

Gegeben ist eine Dezimalzahl, die als Parameter übergeben wird.

Gesucht ist die duale Darstellung der Dezimalzahl.

Sie soll in das Feld der Dualzahl geschrieben werden,

mit der die Methode aufgerufen wurde.

7710 = ?2

Algorithmus:

Wir haben einen int-Parameter "dezi".

Dieses Mal müssen wir die Schleife von oben nach unten durchlaufen:

Wir prüfen, beim MSB beginnend, ob der noch verbleibende Rest von "dezi"

größer oder gleich der Wertigkeit des aktuellen Bits ist.

* Wenn ja, setzen wir das Bit auf "true"   
  u. ziehen von dezi die Wertigkeit ab.
* Wenn nein, setzen wir das Bit auf "false"  
  u. lassen "dezi" unverändert.

Beispiel: 7710 = ?2

MSB LSB  
 boolean

dual:

Index i: 7 6 5 4 3 2 1 0

Wertigkeit 2i: 27 26 25 24 23 22 21 20

128 64 32 16 8 4 2 1

dezi = 77

i = 7: Passt 27 = 128 in 77 ? Nein, deshalb schreibe ich für dual[7] ein false  
 und lasse dezi mit 77 unverändert.

MSB LSB  
 boolean

dual: 0

Index i: 7 6 5 4 3 2 1 0

Wertigkeit 2i: 27 26 25 24 23 22 21 20

128 64 32 16 8 4 2 1

dezi = 77

i = 6: Passt 26 = 64 in 77 ? Ja, deshalb schreibe ich für dual[6] ein true  
 und subtrahiere von dezi die Wertigkeit:   
 77 – 64 = 13.

dezi = 13

MSB LSB  
 boolean

dual: 0 1

Index i: 7 6 5 4 3 2 1 0

Wertigkeit 2i: 27 26 25 24 23 22 21 20

128 64 32 16 8 4 2 1

MSB LSB  
 boolean

dual: 0 1

Index i: 7 6 5 4 3 2 1 0

Wertigkeit 2i: 27 26 25 24 23 22 21 20

128 64 32 16 8 4 2 1

dezi = 13

i = 5: Passt 25 = 32 in 13 ? Nein, deshalb schreibe ich für dual[5] ein false  
 und lasse dezi unverändert:   
dezi = 13

MSB LSB  
 boolean

dual: 0 1 0

Index i: 7 6 5 4 3 2 1 0

Wertigkeit 2i: 27 26 25 24 23 22 21 20

128 64 32 16 8 4 2 1

u.s.w.

Habe ich die Schleife bis 0 durchlaufen,

steht in der Dualzahl die duale Darstellung der übergebenen Dezimalzahl.

In Pseudocode:

Gegeben ist eine Dezimalzahl,

die ich als Parameter "dezi" der Methode "dezi\_zu\_dual()" übergeben bekommen.

Gesucht ist die duale Darstellung der Dezimalzahl.

Sie soll in die Dualzahl, mit der die Methode aufgerufen wurde, geschrieben werden.

Die Wertigkeit an einer Stelle "i" bekommen wir wieder

(bis vor der nächsten Optimierung) über die Hilfsmethode "zweihoch()".

Laufe über Zählvariable i von 7 bis 0:

ist die Wertigkeit der Stelle i,

nämlich 2 hoch i,

kleiner oder gleich dem noch zu codierende dezimale Rest "dezi"?

wenn ja:

- schreibe an die Stelle "dual[i]" ein "true"

- subtrahiere die Wertigkeit zweihoch(i) von dezi

wenn nein:

- schreibe an die Stelle "dual[i]" ein "false"

1. Version unter Verwendung der Hilfsmethode "zweihoch()":
3. **public** **void** dezi\_zu\_dual(**int** dezi) {
4. // Methode schreibt den übergebenen Parameter
5. // in die Dualzahl, mit der sie aufgerufen wurde
7. **for** (**int** i=7; i>=0; i--)
8. **if** (zweihoch(i)<=dezi) {
9. dual[i] = **true**;
10. dezi -= zweihoch(i);
11. } **else**
12. dual[i] = **false**;
13. }
15. **public** **int** zweihoch(**int** n) {
16. **int** ergebnis=1;
18. **for** (**int** i=1; i<=n; i++) // i:1..n
19. ergebnis \*= 2;
21. **return** ergebnis;
22. }

Hierbei haben wir zweimal unnötiger Weise die gleich Methode aufgerufen.

Wenn wir stattdessen mit einer Hilsvariable arbeiten,

die wir auch "zweihoch" nennen,

aber nur einmal mit dem Ergebnis des Methodenaufrufes belegen,

können wir uns die doppelte Berechnung sparen.

Trotzdem bleibt die quadratische u. damit polynomielle Komplexität,

da wir in jedem den n Schleifendurchläufe neu von 1 bis n laufen.

**public** **void** dezi\_zu\_dual(**int** dezi) {

// Methode schreibt den übergebenen Parameter

// in die Dualzahl, mit der sie aufgerufen wurde

**int** zweihoch;

**for** (**int** i=7; i>=0; i--) {

zweihoch = zweihoch(i);

**if** (zweihoch<=dezi) {

dual[i] = **true**;

dezi -= zweihoch;

} **else**

dual[i] = **false**;

}

}

**public** **int** zweihoch(**int** n) {

**int** ergebnis=1;

**for** (**int** i=1; i<=n; i++) // i:1..n

ergebnis \*= 2;

**return** ergebnis;

}

2. Version:

Es soll ganz auf die Hilfsmethode "zweihoch()" verzichtet werden,

um - wie schon bei der ersten Richtung - die Komplexität

von O(n²) auf O(n) zu drücken.

Wir wissen, dass die Wertigkeit einer Stelle rechts

die Hälfte der Wertigkeite einer Stelle links ist.

2i-1 = 2i / 2

Wir müssen diese Mal von links, vom MSB aus laufen

nach rechts zum LSB.

Wie schon oben arbeiten wir mit einer Hilfsvariablen "wertigkeit",

die wir mit der Wertigkeit des ersten zu betrachteten Index initialisieren

u. dann bei jedem Schleifendurchlauf halbieren.

Wir fangen hier mit dem MSB eines Bytes an.

Es trägt den Index 7.

Seine Wertigkeit ist 27 = 128.

Damit initialisieren wir die Hilfsvariable

u. dividieren Sie bei jedem Schleifendurchlauf,

um sie für den nächsten Durchgang,

das nächst kleiner i, anzupassen:

**public** **void** dezi\_zu\_dual(**int** dezi) {

// Methode schreibt den übergebenen Parameter

// in die Dualzahl, mit der sie aufgerufen wurde

**int** wertigkeit = 128;

// Wertigkeit des MSBs: 2 hoch 7 ist 128

**for** (**int** i=7; i>=0; i--) {

**if** (wertigkeit<=dezi) {

dual[i] = **true**;

dezi -= wertigkeit;

} **else**

dual[i] = **false**;

wertigkeit /= 2;

// Wertigkeit für i-1 halbieren.

}

}