UNIVERSITAT DE LLEIDA
Escola Politècnica Superior
Grau en Enginyeria Informàtica
Programació II
Profesor: Juan Manuel Gimeno Illa

Práctica 1: Integral Types

Autor: Naïm Saadi Gallego
DNI: 49259968V
Grupo de prácticas: GPraLab3
Fecha de entrega: 17 de marzo de 2024

Introducción

En esta práctica, he trabajado con diferentes tipos de números binarios, como *byte*, *short*, *integer* o *long*, representados en arrays de enteros, y he creado varios métodos para poder trabajar con estos números. He documentado los diferentes métodos uno a uno, incluyendo los métodos auxiliares que he usado.

Apartado 1

boolean allBits (int[] num)

Este método tiene que comprobar, dado un array de enteros, si este está compuesto únicamente de ceros o unos. Este método ha sido muy sencillo:

- He usado un bucle *for* que recorra de la primera a la última posición del array y si en algún momento se encuentra un número diferente a 0 y a 1, el método retorna *false*. De lo contrario, simplemente sale del bucle y retorna *true*.

Apartado 2

void copy (String from, int[] to)

Este método tiene que copiar inversamente el *String from* en el array *to*.

- Como habrá que recorrer la cadena y el array en diferentes sentidos, he creado dos índices diferentes: *iArray* y *iString*.
    - Inicializo *iArray* a 0, que recorre hasta *to.length - 1* y *iString* al final de la cadena, *from.length - 1*, que recorre hasta el 0.
- Uso un *for* en el que voy variando *iString*, pero no *iArray*. Esto lo hago para que *iArray* no deje espacios vacíos si me encuentro un carácter que no es ni 0 ni 1.
- Para tomar los carácteres de *from* he usado la función *charAt()*, y como el valor entero del carácter no se corresponde con el valor real, le he restado el valor del carácter *'0'*, así me da simplemente 0 o 1.
    - Es importante denotar que esto solo lo hace si el número es 0 o 1, y en este caso sí que aumento *iArray*, ya que un espacio del array ha sido llenado

Apartado 3

int[] narrow (int[] from, int toLength)

Este método devuelve un array de tamaño *toLength* con los bits menos significativos de *from*.

- Para hacer esto, simplemente he tomado *from* como una String (usando *toString*) y lo he copiado en un array llamado *narrowed*.
    - Crear dicho array *narrowed* ayuda a que no se devuelva el mismo array, sino otro diferente con los mismos valores.
    - Si *toLength* resulta ser 0, el bucle de *copy* no se realiza y simplemente se retorna un array vacío.

Apartado 4

int[] widen (int[] from, int toLength)

Este método devuelve un array de tamaño toLength con los bits de *from*, rellenando los espacios sobrantes con el signo del número.

- Primeramente, he declarado un array *widened* del tamaño de *toLength* y he copiado cada uno de los bits de *from* a *widened* usando *copy,* cómo en la función *narrow*.
- Después, si el array *from* era de tamaño 0, simplemente retorno widened.
- Si es mayor que 0, entonces uso el método auxiliar *fillWithSign* para rellenar *widened* con el signo.
    - Cabe destacar que no debo usar fillWithSign con un array de tamaño 0 porque este accede a la posición *from.length - 1* del array, siendo -1 la posición a buscar, cosa que es imposible ya que no existe dicha posición en el array.

Método auxiliar (I)

void fillWithSign (int[] num, int signPosition)

Este método auxiliar rellena un array del número en la posición signPosition.

- Tiene dos parámetros: *num*, que es el array que se rellena, y *signPosition*, que es la posición en la que se encuentra el signo con el que tenemos que rellenar.
    - Dados estos parámetros, simplemente recorremos desde la posición siguiente a *signPosition* hasta el final de *num*, y a cada posición del array le damos el valor en la posición *signPosition*, es decir, el signo

Apartado 5

int[] cast (int[] from, int toLength)

Este método hace *narrow* o *widen* dependiendo de el tamaño deseado.

- Con un simple condicional, y viendo si *toLength* es mayor o menor a *from.length*, realizo *widen* o *narrow*.
    - En el caso de que sea igual utilizo *narrow*, ya que tiene menos líneas de código.

Apartado 6

int[] and (int[] arg1, int[] arg2)

Este método realiza el AND booleano bit a bit, dando como resultado un *integer* un *long*.

- Para hacer esto, primeramente compruebo que tipo de número son los parámetros (*long*, integer, short o byte), usando el método *chooseLength*.
    - Una vez compruebo si son *long* u otro (por lo tanto, *integer*), asigno dicha largada a la variable *resultSize*, uso *widen* (ya que el nuevo array siempre será igual o más largo a los anteriores) para transformar ambos parámetros a la largada correspondiente, los guardo en dos nuevos arrays (para no modificar *arg1* o *arg2*) y creo una array de tamaño *resultSize*.
    - Después de esto, hago un bucle de 0 a *resultSize* (de ahí que use una variable, para no tener que hacer dos *for* diferentes para cada longitud). Para hacer un AND booleano, si los dos son 1 debo poner 1 y en caso contrario, 0; pero como el array donde coloco el resultado está inicializado todo a ceros (porque uso *zeroOfSize)*, solo me hace falta poner un 1 en el caso de que ambos sean 1, y no hacer nada en los demás casos.

Método auxiliar (IIa)

int chooseLength (int[] arg1, int[] arg2)

Este método retorna la longitud a la que deberán ser transformados los array.

- Observo las largadas de ambos parámetros (*arg1* y *arg2*), y si uno de ellos tiene más de 32 bits de longitud (comprobado con el método *isLong*), retorno *LONG_SIZE*, es decir, 64. En caso contrario, retorno *INTEGER_SIZE*, es decir, 32.

Apartado 7

int[] or (int[] arg1, int[] arg2)

Este método realiza el OR booleano bit a bit, dando como resultado un *integer* un *long*.

- Este metodo es igual a la anterior, solamente cambiando el final. Como que el OR solo devuelve 0 si ambos son 0, he negado lógicamente esta expresión (es decir, cuando un argumento u otro sea diferente de 0), y he hecho que ponga un 1 en este caso (en caso contrario, dejará el 0 que estaba inicialmente).

Apartado 8

int[] leftShift (int[] num, int numPos)

Este método desplaza los bits hacia la izquierda *numPos* posiciones, rellenando los espacios vacíos con 0.

- Aunque la función indique que es hacia la izquierda, nuestros arrays están en orden invertido, por lo tanto debemos realizar un desplazamiento hacia la derecha del array.
- Primeramente habrá que convertir *num* a *integer* o a *long* usando *widen*, con un esquema similar a las funciones *and* y *or,* usando *chooseLength* también. De esta forma también obtendré *resultSize*, que me ayudará en el bucle, además de usarlo para crear un nuevo array de dicho tamaño y calcular *numPos* módulo *resultSize*, al que llamaremos *distance*.
- Hecho esto, simplemente he hecho un bucle *for* que recorra de la posición *i* a *resultSize* - *distance* (es decir, obviaremos los *distance* bits de más peso), y colocaremos cada bit *i* de *num* en un bit *i* + *numPos* de un array *result*, es decir, *numPos* posiciones a la derecha, dejando espacio para los 0 restantes.
    - Como uso *zeroOfSize* para crear result, no me hace falta rellenar los demás espacios de ceros.

Método auxiliar (IIb)

int chooseLength (int[] num)

Este método es igual al que tiene dos parámetros, simplemente que aquí solo comprueba si el parámetro *num* es *long* u otro.

Apartado 9

int[] unsignedRightShift (int[] num, int numPos)

Este método desplaza los bits hacia la derecha *numPos* posiciones, rellenando los espacios vacíos con 0.

- Al igual que antes, como el array es inverso a la *String* como tal, tendremos que moverlo hacia la izquierda.
- El proceso es casi idéntico al anterior, con la diferencia en la dirección a la que mover los bits.
- Para mover los bits a la izquierda, he usado el método auxiliar *shiftToTheRight*.
    - Como que he usado *zeroOfSize* para crear el array a retornar, no me hace falta rellenarlo con ceros manualmente.

Método auxiliar (III)

void shiftToTheRight (int[] from, int[] to, int distance)

En este método, tomo los *bits* del array *to* y los copio en *from* desplazados *distance* posiciones a la izquierda.

- Para ello, empiezo en la posición *distance* (es decir, *i* = *distance*) y muevo cada bit de *from* a una posición *i* - *distance* de *to*, habiendo así despreciado los bits de menos peso (aquellos en el intervalo de posiciones (0, distance -1).

Apartado 10

int[] signedRightShift (int[] num, int numPos)

Este método desplaza los bits hacia la derecha numPos posiciones, rellenando los espacios vacíos con el signo del número original.

- Este método sigue el mismo procedimiento que *unsignedRightShift*, con diferencia en que al final hacemos *fillWithSign*.
    - En este caso, el signo se encuentra desplazado numPos posiciones a la izquierda, así que he pasado el valor en (*resultSize* -1) - distance como *signPosition*, es decir, el valor a una distancia distance de la última posición.

¿Qué he aprendido haciendo esta práctica?

Esta ha sido la primera práctica que he tenido que documentar más formalmente, y sinceramente, más que la prática en sí, ha sido el hacer el informe y el pensar que se iban a fijar en la claridad de mi código lo que me ha hecho esforzarme verdaderamente en intentar mejorarlo lo más seguidamente posible. Cuando pasé todos los tests de la práctica, pensaba que ya había terminado, pero realmente, a medida que me iba fijando en mi código, he podido ver pequeños fallos o imperfecciones que pulir (¡una vez llegué a optimizar 16 líneas de código en 5!). Creo que hacer esta práctica me ha ayudado a ser más claro y simple con mi código y a esforzarme en pulirlo.

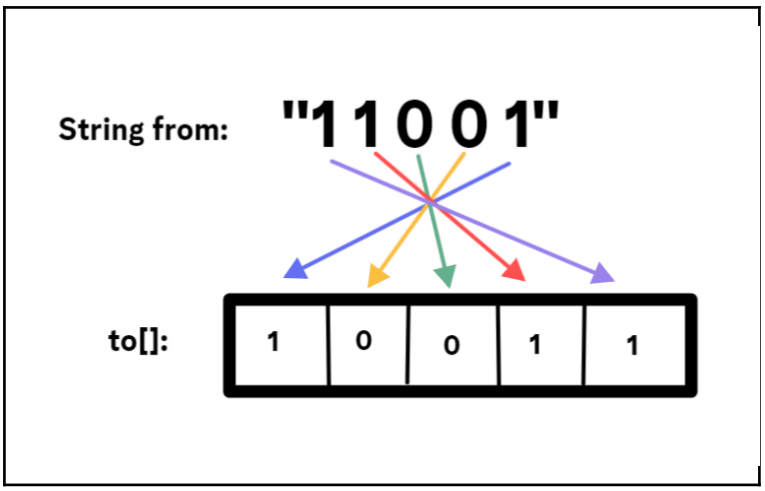
A parte de eso, también me ha ayudado mucho usar los tests para observar en que partes mi código fallaba. Me he podido familiarizar más con el IDE y he aprendido a usar el debugger correctamente.

Bibliografia

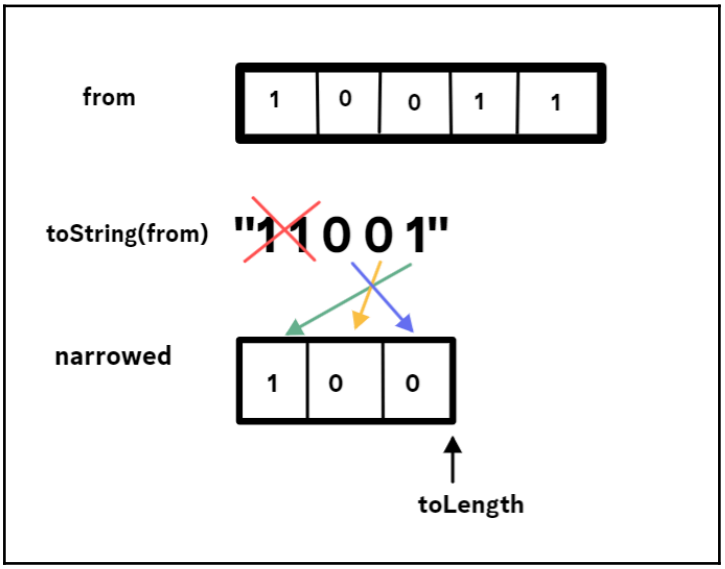
Apuntes Tema 2, Programación II

Diagramas

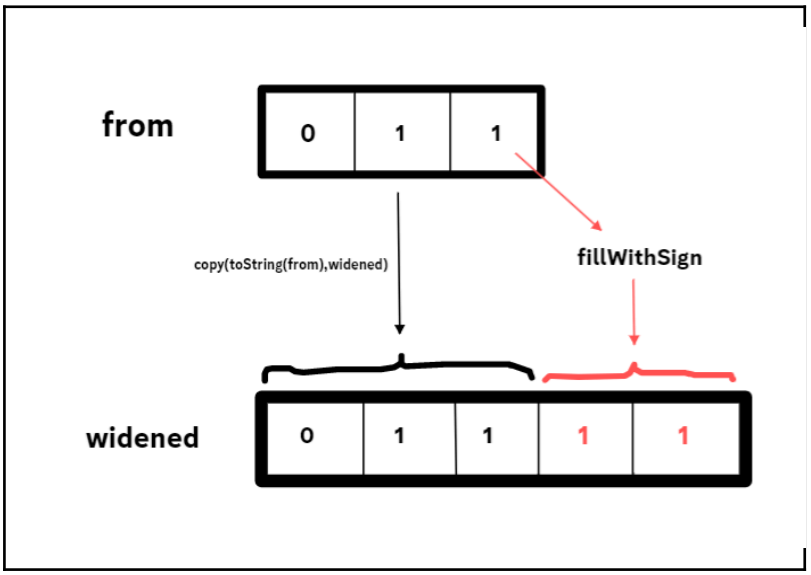
copy



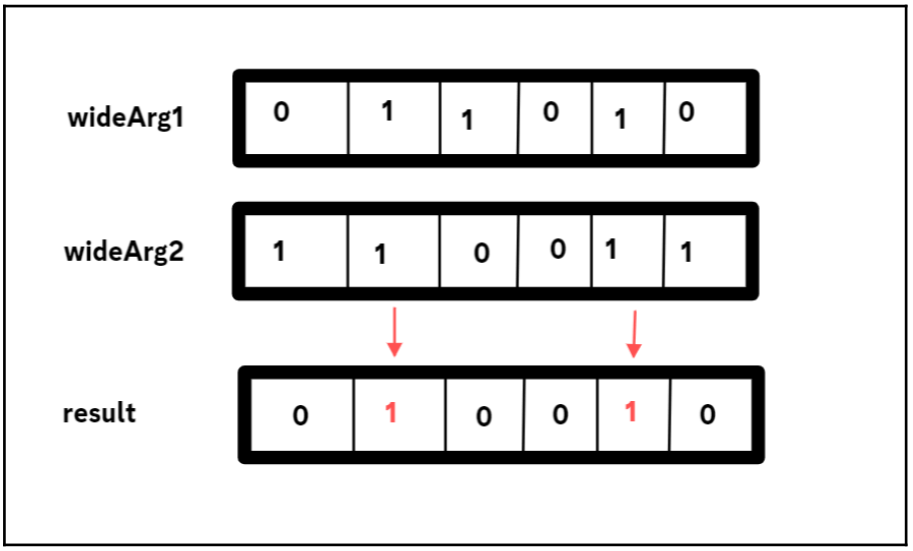
narrow



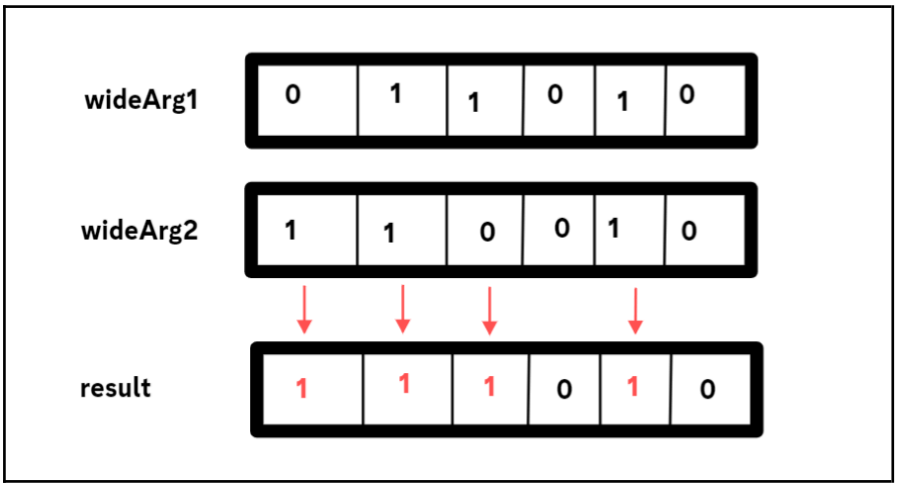
widen



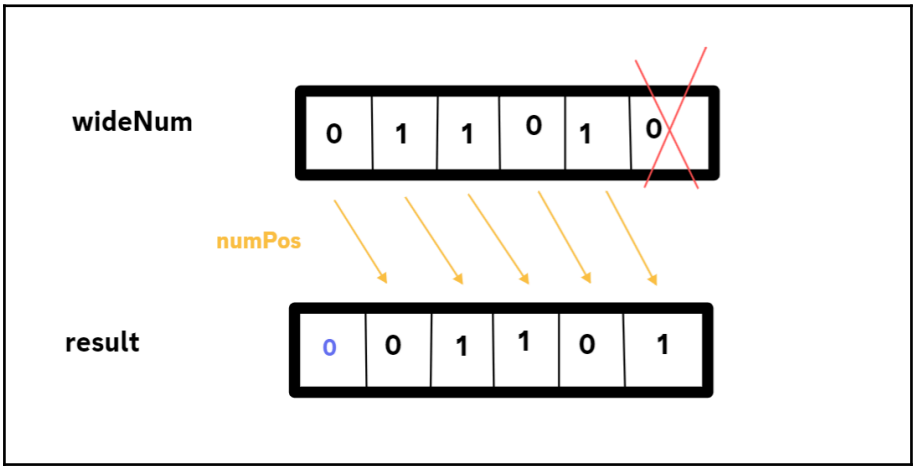
and



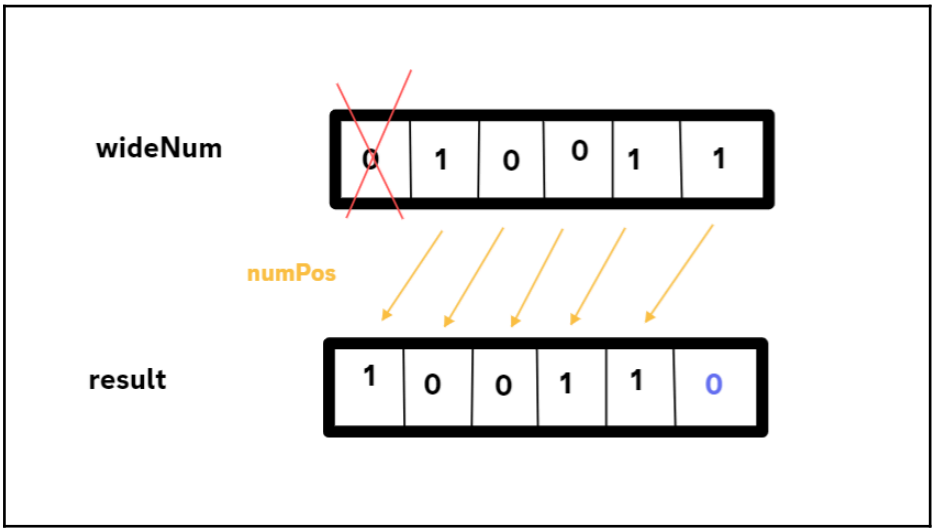
or



leftShift



unsignedRightShift



signedRightShift