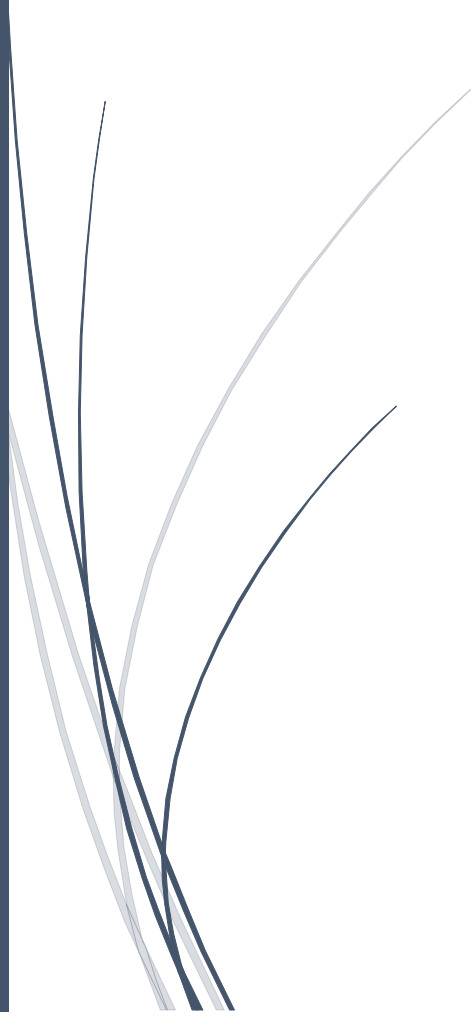
15-10-2013

Reto 2

Estructura de Datos



Juan José Montero Parodi
Vicente Martínez Rodríguez

Ejercicio 1:

El algoritmo para resolver el problema es el siguiente:

- Vamos probando primero con una x, por si el número que nos dan coinciden con uno aleatorio (es el caso de 100).
- Después probamos con dos x, tres x,… hasta 6, o que encontremos la solución.
- Las posibles operaciones están en el otro pdf, en las tablas 1 y 2. Aunque en la tabla 2 se repiten operaciones, hay que tener en cuenta que las operaciones – y /, no son conmutativas, por lo tanto si hace falta la tabla 2, aunque en la suma y multiplicación no serían necesarias, las pongo como aclaración.
- Cuando usemos la tabla 1, como sólo podemos usar una vez una x[i], hay que tener en cuenta que las operaciones en las que salga, no hay que calcularlas:
  - x1: aparece en las posiciones 0,1,2,3,4
  - x2: aparece en las posiciones 0,5,6,7,8
  - x3: aparece en las posiciones 1,5,9,10,11
  - x4: aparece en las posiciones 2,6,9,12,13
  - x5: aparece en las posiciones 3,7,10,12,14
  - x6: aparece en las posiciones 4,8,11,13,14
  - por tanto, si usamos x1 y x6, no usaremos 0, 1, 2, 3, 4, 8, 11, 13, 14 en los cuatro vectores de operaciones elementales (+, -, *, /).
- Cuando usemos la tabla 2, como tampoco podemos repetir x[i], no usaremos el vector x[i] correspondiente y además no usaremos las posiciones (x[i]: elemento→posición):
  - X[1]: 2 → 0; 3 → 1; 4 → 2; 5 → 3; 6 → 4;
  - X[2]: 1 → 0; 3 → 1; 4 → 2; 5 → 3; 6 → 4;
  - X[3]: 1 → 0; 2 → 1; 4 → 2; 5 → 3; 6 → 4;
  - X[4]: 1 → 0; 2 → 1; 3 → 2; 5 → 3; 6 → 4;
  - X[5]: 1 → 0; 2 → 1; 3 → 2; 4 → 3; 6 → 4;
  - X[6]: 1 → 0; 2 → 1; 3 → 2; 4 → 3; 5 → 4;
  - Aquí accederíamos a la suma, para acceder a la resta, sumamos 5 a la posición, a la multiplicación, sumamos 10, y a la división 15.

Mientras no encontremos la solución:

1. Recorremos los seis números dados por si coinciden con el número que nos piden. SI coincide, salimos del programa, si no, introducimos otra x.

2. Para dos equis, comprobamos las posibles operaciones en la tabla 1 para suma y multiplicación, y en la tabla 2 para la resta y división.

3. Para tres equis, comprobamos las posibles operaciones en la tabla 1 para suma y multiplicación, y en la tabla 2 para resta y división, y también comprobamos la suma y multiplicación para otra x (la tercera) con las operaciones +, -, *, /. Es decir, cogemos un valor de la tabla (usando dos x) y comprobamos para las otras 4 x restantes las 4 operaciones elementales.

4. Para 4 x, había que comprobar la +, -, *, / para dos resultados de las tablas, o bien un resultado de las tablas +, -, *, / la tercera x +, -, *, / la cuarta x. Para la tercera x solo nos quedan 4 x posibles, y para para la 4 escogemos una entre 3. Y nos sobrarían dos.

5. Para 5 x, habría que comprobar dos resultados de las tablas +, -, *, / la quinta x, o bien un resultado de las tablas +, -, *, / la tercera x +, -, *, / la cuarta x +, -, *, / la quinta x.

6. Para las 6 x, se puede hacer cogiendo los tres resultados que dan 3 valores de las tablas, y combinamos los tres resultados con +, -, *, /. Otra forma sería coger un valor de la tabla y +, -, *, / con la tercera x +, -, *, / la cuarta x +, -, *, / la quinta x +, -, *, / la sexta x. Por último sería coger un valor de la tabla (usamos dos x) +, -, *, / la tercera x, +, -, *, / la cuarta x +, -, *, / la quinta x +, -, *, / la sexta x.

- Así nos aseguramos de que todas las combinaciones son posibles. Estos datos se guardarían en un vector, que iríamos comprobando antes de pasar de paso con el número que nos piden. Si no lo encontramos y acabamos todos los pasos, mostramos un mensaje de que no es posible con los números dados encontrar el número de tres cifras. Para mostrar el número que más se le acerca, lo que hacemos es mientras lo recorremos, lo vamos restando al número que nos piden y lo guardamos en una variable inicializada a menos infinito, de manera de que el mayor (el que más se acerca a cero) es el que más se acerca al número.
- Nota: cuando pongo +, -, *, /. Me refiero a que hay que tener en cuenta las cuatro operaciones, guardándolas para después comprobarlas.

Ejercicio 2:

Struct CeldaCola:

```
/**
* @brief Struct fomado por un objeto de tipo Pieza llamado element y un puntero sig
* a la siguiente celda de la cola.
*/

Struct CeldaCola{
        Pieza elemento;
        CeldaCola *sig;
};
```

Class Cola:

/*! \class Cola cola.h "inc/cola.h"

* \brief esto es una clase cola.

* Esto tiene una longitud (número de elementos), y dos punteros celda primera y celda primera, y un objeto de tipo Imagen de cola.

*/

Class cola{

Int longitud;

CeldaCola *primera;

CeldaCola *ultima;

Imagen Cola;

Public:

Constructor:

/* @brief Constructor de la clase cola.

* Asigna 4 por defecto.

*/

Cola();

Constructor por parametro:

/* @brief constructor de copia de la clase cola.

* @param entero n que construye una cola c.

```
*/
Cola (int n);
```

IntroPieza:

```
/* @brief introduce en la cola una pieza.
* @param pieza p a insertar.
*/
Void Poner(Pieza p);
```

PiezaRandom:

```
/* @brief Crea una pieza aleatoria y llama a Poner(Pieza p).
*/
Void PiezaRandom();
```

QuitarPieza:

```
/*@brief Quita la primera pieza en entrar (FIFO).
*/
Void Quitar();
```

PiezaFrente:

```
/*@brief Devuelve el frente de la pieza. (La primera pieza)
* @return la pieza al frente.
*/
Pieza frente();
```

PintarCola:

```
/*@brief Muestra por pantalla la cola.
*/
Void PintarCola();
```

Destructor:

```
/*@brief destruye el elemento cola.
```

```
*/
~Cola();

};
```

Class Imagen:

```
/*! \class Imagen Imagen.h “inc/Imagen.h”
* \brief esto es una clase Imagen.
* Esta clase crea un objeto de tipo imagen, a partir de un puntero de tipo BITMAP y un identificador entero id.
*/
Class Imagen{
Bitmap *celda_img;
Int id;
Public:
```

Contructor por defecto:

```
/*@brief Crea una imagen por defecto.
*/
Imagen();
```

Constructor por parametro:

```
/*@brief carga una imagen según el identificador en el Bitmap.
* @param un entero identificador.
*/
Imagen (int ide);
```

Pintar Imagen:

```
/*@breif se encarga de pintar la imagen en pantalla.
*/
Void PintarImagen();
```

Destructor:

/*@breif Destruye la imagen.

~Imagen();

};

Class Marcador:

/*! \class Marcador Marcador.h “inc/Marcador.h”

* \brief esto es una clase Marcador.

* Por defecto está a cero, y se va incrementando con las filas enteras. Tiene un título, un nivel,

* líneas, número de piezas y el estado partida. Todas menos la primera y la última son enteros,

* las otras son string. Y marcador es de tipo Imagen.

*/

Class Marcador{

String título;

Int nivel;

Int líneas;

Int n_piezas;

String estado_partida;

Imagen marcador;

Public:

Constructor:

/*@brief Crea el constructor por defecto de Marcador.

* Título= “TETRIS”, nivel= 1; líneas= 0; n_piezas= 0; estado_partida= “Jugando…”

*/

Marcador();

ModificarTítulo:

/*@brief Modifica el título del tetris.

*/

Void ModificarTítulo();

IncrementarLíneas:

/*@brief Incremeta el número de líneas destruidas (si se ha destruido alguna)

* @param int elem[] Tiene la posición de las líneas eliminadas.

*/

Void IncrementaLineas(int elem[]);

IncrementarNivel:

/*@brief Cuando eliminas 10 líneas subes de nivel.

*/

Void IncrementaNivel();

IncrementarPieza:

/*@brief Muestra el número de piezas que se han colocado durante la partida.

*/

Void IncrementaPiezas();

ModificarEstado:

/*@brief Modifica el estado si se ha terminado de jugar. Se está jugando o si el juego

* ha terminado.

*/

Void ModificarEstado();

PintarMarcador:

/*@brief Muestra por pantalla todos los datos del marcador.

*/

Void PintarMarcador();

Destructor:

/*@brief destruye el Marcador.

*/

~Marcador();

};

Struct Celda:

```
/**
* @brief Struct formado por un bool que permite saber si una celda es libre u ocupada
* además permite formar las piezas, y tambien presenta un objeto de tipo imagen para
* que la celda pueda llegar a ser representada en el caso de que el bool sea true.
*/

Struct Celda{
        bool ocupacion;
        Imagen imcelda;
};
```

Class Tablero:

```
/**! \class Tableror Tablero.h "inc/Tablero.h"
*  @brief Clase que se encargar de realizar todas las funcionalidades de eliminación y creación
*  de celdas, finalizar el juego, ajustar el tamaño del tablero y pintar el tablero.
* /

Class Tableror{

    Imagen tabl;

    Celda **matriz;

    bool game_over;

    int fil, col;

  public:
```

Constructor:

```
    /**
    * @brief Constructor por defecto que crea e inicializa el tablero con unas medidas
    * por defecto.
    */

    Tablero();
```

Constructor por parametro:

```
    /**
    * @brief Constructor que crea e inicializa el tablero con unas medidas
    * pasadas por dos parametros.
    * @param  f: número de filas que se desean crear.
    * @param c: número de columnas que se van a crea.
    */

    Tablero(int f, int c);
```

Obtener tamaño de filas:

```
/**
* @brief  Muestra el valor de fil.
* @return fil : Devuelve el valor entero de la variable privada.
*/

int GetFil();
```

Obtener tamaño de columnas:

```
/**
* @brief  Muestra el valor de col.
* @return col : Devuelve el valor entero de la variable privada.
*/

int GetCol();
```

Método Encaja:

```
/**
* @brief Metodo que comprueba si una pieza encaja en una cierta posición dentro del
*  tablero.
*  @param p: objeto de tipo Pieza al que se comprueba si encaja en una cierta
*  posición.
*   @return encaje: variable bool que si es true encaja, si es false no.
*/

bool Encaja(Pieza p);
```

Método NuevaLinea:

```
/**
* @brief Método que se encarga de crear una fila de celdas nueva en la posición 0.
*/

Void NuevaLinea();
```

Método EliminarLineas:

```
/**
* @brief Metodo que se encarga de eliminar las filas en las que cada celda presente
* su atributo ocupacion a true, llamará al metodo NuevaLinea().
* @param elim[] es un array que presenta 4 posiciones y en ella se indica la fila que se
* va a eliminar.
*/

void EliminarLineas(int elim[]);
```

Método ComprobarLinea:

```
/**
* @brief Método que introduce en un array de enteros las posiciones de las filas que
* se deben eliminar.
* @param elim[] : array donde se anotan las posiciones de las filas a eliminar.
* @param tam : tamaño del array que por defecto va a ser cuatro ya que es el máximo
* de líneas que se pueden eliminar con una pieza.
*/

void ComprobarLineas();
```

Método ActualizarTablero:

```
/**
* @brief Método que llamará a los métodos de ComprobarLineas y EliminarFilas.
*/

void ActualizarTablero();
```

Método EncajeFinal:

```
/**
* @brief Método que copia las celdas que forman la pieza dentro de la matriz de
* celdas del tablero, finalmente llama a Game_Over().
* @param p : Objeto de tipo pieza al se le copia las celdas validas en el tablero.
* @return siguiente : Variable booleana que permitirá o no que la siguiente pieza
* de la pila se cargue en el tablero.
*/

bool EncajeFinal(Pieza p);
```

Método GameOver:

```
/**
* @brief Método comprueba si la pieza no ha entrado completamente en el tablero y
* pone la variable privada game_over a true o false para terminar al juego.
*/

Void GameOver();
```

Método FinDelJuego:

```
/**
* @brief Método que comprueba si la variable game_over esta true y de ser así hace
* terminar todo el juego.
*/

Void FinDelJuego();
```

Método PintarTablero:

```
/**
* @brief Método que carga muestra por pantalla el objeto Imagen tabl.
*/

Void PintarTablero();
```

Destructor:

```
/**
* @brief Libera toda la memoria ocupada por el objeto tablero.
*/

~Tablero();
};
```

Clase Pieza:

```
/**
* @ brief Clase que se encargar de construir una pieza a partir de puntero doble de
* celdas, un identificador de pieza y una orientacion de la pieza.
*/

Class Pieza{
        int identificador;
        Celda **forma;
        int orientacion;
    public:
```

Constructor por defecto:

```
        /**
        * @brief Constructor por defecto que da valores iniciales a un objeto Pieza.
        */

        Pieza();
```

Constructor por parámetro:

```
        /**
        * @brief Constructor que pasa un parámetro y partir del valor del parámetro
        * crea un objeto Pieza diferente.
        * @param id : parámetro de tipo entero.
        */

        Pieza(int id);
```

Método de Asignación:

```
/**
* @brief Asigna a un objeto  dado los datos de otro objeto de tipo Pieza.
* @param p : Parámetro de tipo Pieza.
*/

void CopiarPieza(Pieza p);
```

Método Rotar:

```
/**
* @brief Método que modifica la variable rotacion de la pieza según un bool,
* @param rotacion : parámetro de tipo bool, false rota a izquierda y true a
* derecha.
*/

void Rotar(bool rotacion);
```

Método Mover:

```
/**
* @brief Método que mueve o rota una pieza según la tecla.
* @param tecla: parámetro que según el tipo se realizará una acción sobre la
* tecla.
*/

void Mover(Tecla t);
```

Método Consulta de bloque:

```
/**
* @brief Método que consulta la ocupación de un bloque.
* @param f : parámetro entero que indica la posición de la fila de la celda.
* @param c: parámetro entero que indica  posición de la columna de la celda.
* @return ocupación : devuelve un bool con la ocupación de la celda.
*/
};
```