Código: 75.001

Asignatura: Estructura de la Información

Curso: 2010/2011 – primer semestre

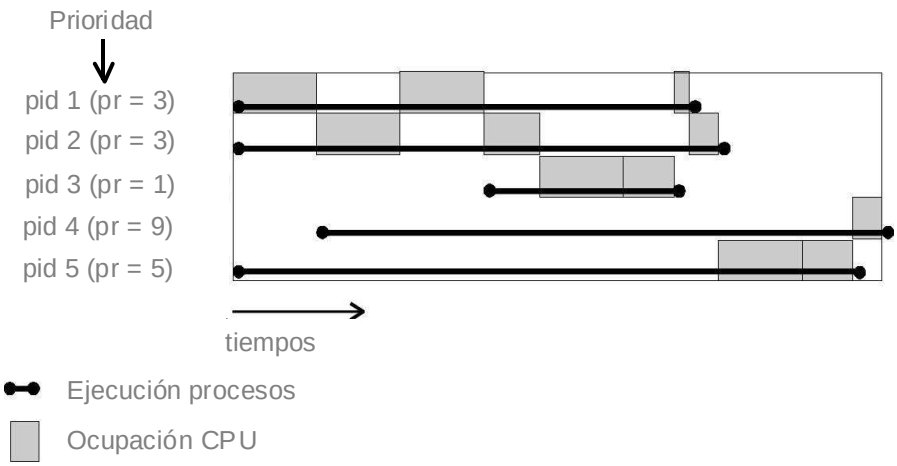
PEC 1

Ejercicio 1 [3 puntos]

Los sistemas operativos multiproceso disponen de un mecanismo que permite que el conjunto de procesos que están ejecutándose en cada momento (y compitiendo por la CPU entre ellos) sean servidos según unos ciertos criterios. Supongamos que estamos trabajando con un sistema operativo que trabaja con 10 niveles de prioridad por proceso, de 0 a 9, siendo 0 la más alta y 9 la más baja.

El sistema operativo va asignar la CPU de la siguiente forma: selecciona un proceso en ejecución (a continuación veremos cómo) y lo ejecuta durante un intervalo de tiempo limitado. Si el proceso acaba, se pasa a seleccionar otro proceso. Si no acaba, se le desasigna la CPU, se vuelve a poner en la secuencia de procesos que esperan y se selecciona otro para ser servido.

Cuando el sistema operativo debe seleccionar un proceso para servirlo selecciona aquel que tiene la prioridad más alta (más cercana a 0). Si hay más de uno, podrá seleccionar cualquiera de ellos (adviértase que esto es una simplificación de la realidad, donde normalmente las políticas para seleccionar el proceso suelen ser más complejas).



De esta forma, vemos que un proceso con una cierta prioridad no podrá ser servido a menos que hayan finalizado aquellos procesos que tienen más prioridad que él. Por otro lado, es posible eliminar un proceso de la cola de espera.

Se quieren tener las siguiente funcionalidades:

- Crear la secuencia vacía.
- Ejecutar un proceso, es decir, añadirlo a la secuencia dada, además de la información del proceso (por ejemplo, la dirección de memoria a partir de la cual reside el código a

ejecutar), su prioridad y el tiempo de CPU que necesita. Se debe retornar un identificador único, llamado PID – Process Identifier –, que identifica el proceso en la secuencia.

- Enviar un proceso a la CPU para que se sirva.
- Devolver un proceso a espera cuando ha acabado el tiempo máximo de ocupación de la CPU.
- Eliminar un proceso cuando acaba su ejecución.
- Eliminar un proceso identificado por PID de la secuencia.
- Permitir hacer un listado por pantalla de los procesos que están en espera ordenados de más a menos prioridad (a igual prioridad no importa el orden).

El TAD no decide durante cuánto tiempo se sirve un proceso ni si un proceso acaba su ejecución o se devuelve a la secuencia de procesos. Esto se decide de manera externa a nuestro TAD y por tanto no debe ser tratado.

Considerando que el número de procesos en espera no será mayor que N, se solicita:

Apartado 1.1) Define la signatura (operaciones y sintaxis) del TAD y sitúalo en la jerarquía de clases de la biblioteca de TADs de la asignatura (de donde debería extender, relaciones con otros, ...). Especializa (sustituye) los parámetros de las clases parametrizadas de la jerarquía, siempre que te sea posible, por los tipos que consideres más oportunos.

Razona tu respuesta.

Signatura del TAD:

TAD WaitingQueue{

WaitingQueue();

int runProcess(long codeAddress, long totalCpuTime, byte priority);

boolean cpuIsBusy();

int processInCpu();

void toCpu();

void toWaitingQueue();

void finishProcess();

void deleteProcess(int pid);

Iterador waitingProcessList();

}

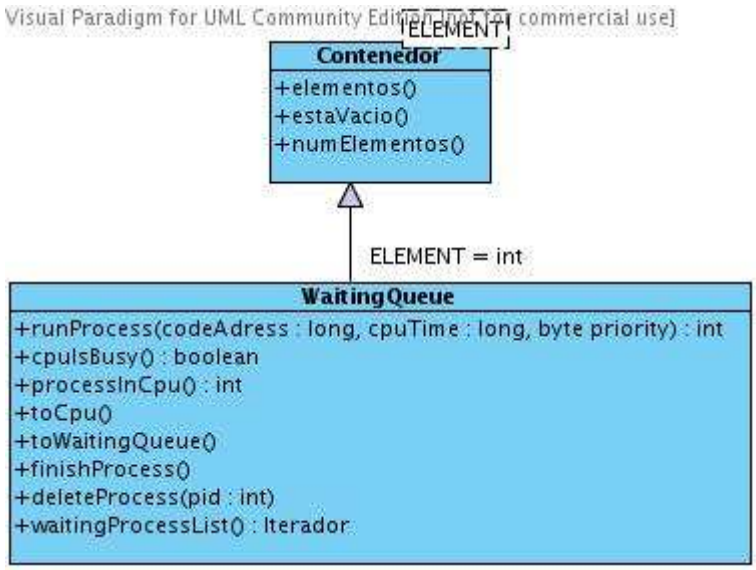
Observad que se han añadido algunas operaciones necesarias para identificar el proceso que se encuentra en la CPU.

Éste es un TAD muy específico que lo haremos heredar directamente de Contenedor, pues aunque tiene cierto parecido a una cola, sus operaciones son específicas y no guardan relación con las del TAD Cola.

Teniendo en cuenta la implementación, probablemente partiríamos de una cola prioritaria que enriqueceríamos con las operaciones específicas de la signatura, aunque otras posibilidades serian también factibles.

Se ha supuesto que el TAD trabaja con identificadores de procesos (PIDs) y dispone de métodos para consultar información del proceso a partir del PID.

Por ello WaitingQueue además de heredar de Contenedor especializa el parametro ELEMENT estableciéndolo como tipo entero.



Apartado 1.2) Haz la especificación del TAD. Usa el estilo de especificación visto en el apartado 1.2.3. Se valorará especialmente la concisión (ausencia de elementos redundantes o innecesarios), precisión (definición correcta del resultado de las operaciones), completitud (consideración de todos los casos posibles en que se puede ejecutar cada operación) y carece de ambigüedades (conocimiento exacto de cómo se comporta cada operación en todos los casos posibles) de su solución. Es importante responder este apartado usando una descripción condicional y no procedimental. La experiencia nos demuestra que no siempre resulta fácil distinguir entre ambas descripciones, es por ello que hacemos especial hincapié insistiendo que pongáis mucha atención en vuestras definiciones. A título de ejemplo indicaremos que la descripción condicional (la correcta a usar en el contrato) de *llenar un vaso vacío con agua* seria:

> @pre el vaso se encuentra vacío.
> @post el vaso se encuentra lleno de agua

En cambio una descripción procedimental (incorrecta para usar en el contrato) tendría una forma similar a:

@pre el vaso deberia encontrarse vacío, si no se encontrara vacío debería vaciarse.
@post Se acerca el vaso al grifo y se echa agua hasta que esté lleno

Debéis también tener en cuenta que un contrato debería disponer de invariante siempre que esta fuera necesaria para describir el TAD.

TAD WaitingQueue{

@pre cierto

@post el contenedor creado está vacío *($this.elementos()=0)*.

WaitingQueue();

@pre *codeAddress* es la dirección de memoria donde se empieza el código del processo, *totalCpuTime* el tiempo que el proceso necesitará usar la CPU y *priority* se encuentra dentro del intervalo [0, 9]

@post El sistema ha reservado los recursos necesarios para la ejecución del proceso y mantiene guardada la información identificada con un identificador numérico (PID) el cual pasa a formar parte de *$this*. El proceso se encuentra en espera de usar la CPU.

$return el identificador numérico (PID) con el que el sistema ha identificado la información necesaria para ejecutar el proceso.

int runProcess(long codeAddress, long totalCpuTime, byte priority);

@pre cierto;

@post $return hay algún proceso ejecutándose en la CPU.

boolean cpuIsBusy()

@pre hay un proceso ejecutándose en la CPU *(cpuIsBusy())*.

@post *$return* el identificador numérico (PID) del proceso que se está ejecutando en la CPU

int processInCpu();

@pre el contenedor no está vacío (elementos() > 0) y no hay ningún proceso ejecutándose en la CPU *(!cpuIsBusy())*

@post se está ejecutando un proceso en la CPU (cpuIsBusy()). El proceso que se está ejecutando en la CPU tiene una prioridad menor o igual a cualquier otro proceso contenido en el contenedor *($all(pid:waitingProcessList(), priority(pid) <= priority(processInCpu()))*. Hay un proceso menos esperando *($old(elementos()) = elementos()+1)*. El proceso que se està ejecutando en la CPU no está esperando *(!$exists(pid:waitingProcessList(), pid=processInCpu()))*.

void toCpu();

@pre hay un proceso ejecutándose en la CPU *(cpuIsBusy())*.

@post no hay ningún proceso ejecutándose en la CPU *(!cpuIsBusy())*. El proceso que estaba en la CPU se encuentra esperando *($exists(pid:waitingProcessList(), pid=$old(processInCpu())))*.

void toWaitingQueue();

@pre hay un proceso ejecutándose en la CPU *(cpuIsBusy())*.

@post no hay ningún proceso ejecutándose en la CPU *(!cpuIsBusy())*. El proceso que estaba en la CPU ha finalizado y por tanto no se encuentra esperando *(!$exists(pid:waitingProcessList(), pid=$old(processInCpu())))*. Se han liberado los recursos asociados al proceso terminado.

void finishProcess();

@pre pid es un identificador que se encuentra en la cola de espera *($exists(id:waitingProcessList(), id=pid))*

@post Hay un proceso menos esperando ($old(elementos()) = elementos()+1). El proceso identificado por pid no está esperando (!$exists(id:waitingProcessList(), id=pid)). Se han liberado los recursos asociados al proceso identificado por pid.

void deleteProcess(int pid);

@pre cierto

@post $return Iterador conteniendo todos los procesos que se encuentran esperando el uso de la CPU.

Iterador waitingProcessList();

}

Ejercicio 2 [2 puntos]

Se desea implementar un algoritmo para evaluar un polinomio de grado *n*. Los coeficientes del polinomio se guardan en un vector de *n+1* elementos. Por ejemplo, un polinomio de grado 2 como $2x^2+3x-5$ se guardaría en un vector de tres posiciones V=[2, 3, -5]. A continuación se muestra una solución al problema:

```
(1)   long evalua(int[] vector, int x, int grado){
(2)       long s, pot;
(3)       int i, j;
(4)       s=0;
(5)       i=0;
(6)       while(i<=grado){
(7)           j=0;
(8)           pot=1;
(9)           while(j< i){
(10)              pot *= x;
(11)              j++;
(12)          }
(13)          s+=pot*vector[i];
(14)          i++;
(15)      }
(16)      return s;
(17)  }
```

Se pide:

Apartado 2.1) Indicar detalladamente la complejidad asintótica de la solución propuesta.

(4) O(1)
(5) O(1)
(6) O(1)
(7) O(1)
(8) O(1)
(9) O(1)
(10) O(1)

(11) O(1)

(12) n*(O(1)+O(1)+O(1)+O(1)) = O(n)

(13) O(1)

(14) O(1)

(15) n*(O(n)+O(1)+O(1))=O(n^2)

(16) O(1)

(17) O(1)+O(1)+O(n^2)+O(1) = O(n^2)

Coste del bucle de la instrucción (12) = $T(I_{12}) = 1 + \sum_{j=0}^{i-1}(1+1+1) = 1 + 3i$

Coste del bucle de la instrucción (6) serà:

$$T(I_6) = 1 + \sum_{i=1}^{n+1}\left(1+1+1+1+\sum_{j=0}^{i-1}(1+1+1)\right) = 1 + \sum_{i=1}^{n+1}(1+1+1+1+3i)$$

$$T(I_6) = 1 + n + n + n + n + \frac{3(1+n+1)(n+1)}{2} = 1 + 4n + \frac{3n^2+9n+6}{2} = \frac{3n^2+17n+8}{2}$$

El coste es pues O(n^2)

Apartado 2.2) Proponer un algoritmo que mejore la eficiencia asintótica temporal de la solución propuesta, indicando dicha eficiencia (no hace falta realizar el cálculo detallado).

```
(1)   long evaluaEf(int[] vector, int x, int grado){
(2)       long s, pot;
(3)       int i;
(4)       pot=1;
(5)       s=0;
(6)       i=0;
(7)       while(i<=grado){
(8)           s+=pot*vector[i];
(9)           pot*=x;
(10)          i++;
(11)      }
(12)      return s;
(13)  }
```

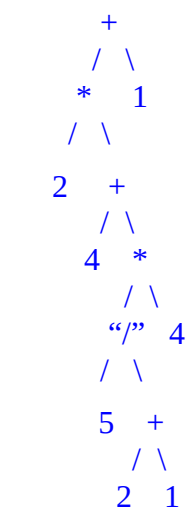
Se puede apreciar que en este caso el coste es lineal, O(n).

Ejercicio 3 [1 punto]

¿Qué estructura de datos utilizarías para representar una operación aritmética con paréntesis, por ejemplo 2 * (4 +5 / (2 +1) * 4) +1. Razonadlo y haced un esquema de como quedaría la expresión anterior. ¿Cual sería el recorrido más adecuado para evaluar el resultado de la operación?

Lo podemos representar como un árbol binario donde las hojas serían los operandos y los nodos intermedios los operadores. Para evaluar la operación tendríamos que hacer un recorrido en postorden y efectuar la operación cada vez que nos encontremos un operador.

Los paréntesis no es necesario guardarlos, pero se utilizarán para generar el árbol.



El recorrido para evaluar el resultado de la operación estaría en postorden.

Ejercicio 4 [1,5 puntos]

Ordena las siguientes cotas de complejidad de menor a mayor, indicando también las que son iguales: $O(0.5 \cdot 2^n)$, $O(n+1)$, $O(2^{n/2})$, $O(\log_{10} n)$, $O((n+1)^2)$, $O(n \cdot \log\ n)$, $O(2\log_6 n)$, $O(n^n)$, $O(3n^2)$, $O(n!)$, $O(n(n+\log\ n))$, $O(1)$, $O(1+1/n)$, $O(n^{n-2})$. Justifica brevemente las relaciones del orden obtenido.

COMPLEJIDAD		EQUIVALENTE ASINTOTICO
a) $O(1)$	$\equiv$	$O(1)$
b) $O(1+1/n)$	$\equiv$	$O(1)$
c) $O(\log_{10}\ n)$	$\equiv$	$O(\log\ n)$
d) $O(2\log_6\ n)$	$\equiv$	$O(\log\ n)$
e) $O(n+1)$	$\equiv$	$O(n)$
f) $O(n \cdot \log\ n)$	$\equiv$	$O(n \cdot \log\ n)$
g) $O((n+1)^2)$	$\equiv$	$O(n^2)$
h) $O(3n^2)$	$\equiv$	$O(n^2)$
i) $O(n(n+\log\ n))$	$\equiv$	$O(n^2)$
j) $O(2^{n/2})$	$\equiv$	$O(2^n)$
k) $O(0.5 \cdot 2^n)$	$\equiv$	$O(2^n)$
l) $O(n!)$	$\equiv$	$O(n!)$
m) $O(n^{n-2})$	$\equiv$	$O(n^n)$
n) $O(n^n)$	$\equiv$	$O(n^n)$

Para ordenar las complejidades propuestas se reducen a sus equivalentes asintóticos usando al notación O. Así pues diremos que a) y b) son equivalentes y los de menor complejidad. c) y d) son también equivalentes reduciendo los logaritmos de distintas bases a logaritmos naturales y representan el siguiente peldaño asintótico. A continuación e) representa un complejidad lineal. f) una complejidad mayor que e) pero menor que g). Des de g) hasta i) son equivalentes y representan una complejidad cuadrática. Las exponenciales de base constante (*j* y *k*) suponen el siguiente escalón asintótico, seguido de la complejidad con cálculo factorial (*l*) y por último las exponenciales de base n (*m* y *n*).

Ejercicio 5 [2 puntos]

Apartado 5.1) Se conoce la representación de las colas mediante vectores y con una estructura secuencial, y también se han estudiado una serie de mejoras a través de la utilización de una gestión circular de las mismas. Enumere brevemente las ventajas y/o inconvenientes que representaría una gestión circular en la implementación de las pilas.

No representa ninguna ventaja puesto que el primer elemento entrado en una pila no se desempila nunca mientras queden otros elementos. És decir, el primer elemento siempre se encuentra en la posición 0 y por tanto las posiciones relativas (orden de entrada) coinciden siempre con la absolutas (posición dentro del vector).

Apartado 5.2) En las listas ordenadas implementadas mediante vectores podemos mejorar la eficiencia de la consulta mediante una búsqueda dicotómica. En la implementación encadenada no se puede mejorar la eficiencia de esta manera, es necesario realizar una búsqueda comenzando por el primer elemento. Comentar la mejora en la eficiencia que obtendríamos si para hacer la búsqueda dispusiéramos de un puntero al elemento que ocupa la posición media de la lista.

Para mejorar la eficiencia deberíamos tener también los punteros medios de todas las sublista obtenidas partiendo por la mitad la lista y sublistas obtenidas de esa manera mientras se puedan dividir. Dichos punteros medios deberían estar conectados entre ellos de manera que a partir de uno cualquiera pueda obtener el puntero correspondiente a la posición media de la sublista izquierda, así como el puntero de la posición media de la sublista derecha. Es decir deberiamos tener los punteros intermedios conectados en forma de arbol.

Eso supone de un lado un coste espacial doble y de otro un coste temporal importante durante la inserción pues una modificación en la lista supondría una reorganización del árbol de punteros intermedios penalizando el coste a O(n) por cada inserción que hagamos. Es decir, un conste cuadrático para el conjunto de entradas insertadas.

La complejidad temporal y espacial es tan grande que no vale la pena mantener esta información.

Apartado 5.3) Hemos visto el funcionamiento de la implementación mediante montículos de los árboles binarios. ¿Cree posible una implementación por montículo de árboles ternarios?. Si lo cree posible, ¿de qué dimensión se debe de crear el vector si queremos almacenar un árbol cuasicompleto de X elementos?. Proponga un ejemplo.

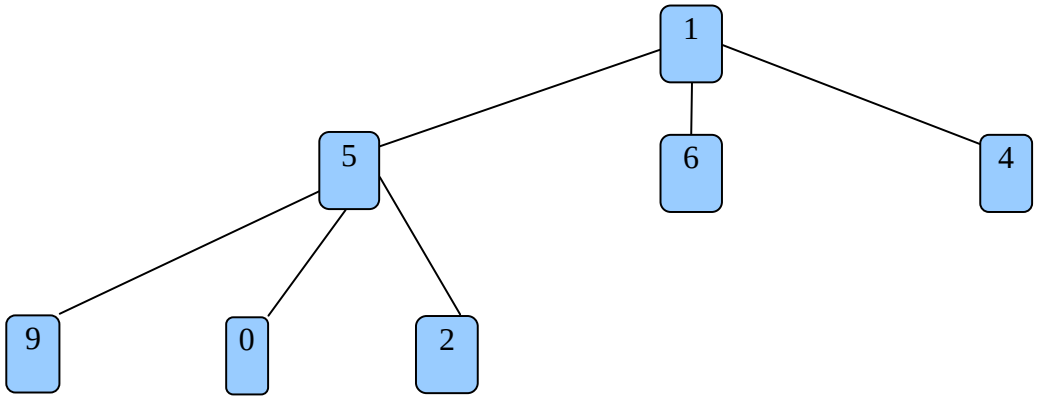
La representación por montículo de arboles ternarios es posible usando el cálculo adecuado para pasar de padres a hijos y viceversa. Así pues, desde la posición de cualquier nodo (N), excepto la raíz, calcularemos la posición del padre (P) haciendo P= (N+1)/3. Del mismo modo la posición de los hijos se hará calculando:

- N(iz) = 3P - 1
- N(cent) = 3P

– N(der) = 3P + 1

Al tratarse de un árbol cuasicompleto, aseguramos que no existirán posiciones vacías en el vector. Por tanto para albergar X elementos deberíamos dimensionar un vector de X posiciones.

Veamos un ejemplo. Para representar el árbol ternario y cuasicompleto:



necesitaremos un vector como el que sigue:

1	5	6	4	9	0	2

Apartado 5.4) Ordenar el siguiente vector mediante el algoritmo de *heap-sort* utilizando la versión que trabaja con un solo vector. Muestre todos los estados intermedios del vector y la situación de los elementos en cada paso:

4	1	7	2	6	3

Formación de la cola prioritaria

4	1	7	2	6	3
4	1	7	2	6	3
4	6	7	2	1	3
7	6	4	2	1	3

Ordenación

7	6	4	2	1	3
3	6	4	2	1	7
6	3	4	2	1	7
1	3	4	2	6	7

4	3	1	2	6	7
2	3	1	4	6	7
3	2	1	4	6	7
1	2	3	4	6	7
2	1	3	4	6	7
1	2	3	4	6	7