Trabajo de Estructuras de Datos de la UNED

04/09/2020

Tarea 1

Pregunta 1

En los cuatro métodos, la complejidad la marcará la llamada recursiva al propio método. En el peor de los casos, queremos acceder o modificar el elemento más profundo de la pila, por lo que tendremos que ir quitando elementos, llamando a recursivamente y volviendo a colocar ordenados los elementos, lo que nos va a dejar con complejidades $O(n)$

Complejidad del get

```
public E get(int pos) {
    E top = stack.getTop();      // O(1)
    if ( pos == 1 ) {
        return top;              // O(1)
    } else {
        stack.pop();             // O(1)
        E result = get(pos-1);   // Recursividad => O(n)
        stack.push(top);         // O(1)
        return result;           // O(1)
    }
}
```

La complejidad de esta operación es $O(n)$. Lo que marca la complejidad será la llamada recursiva. En el peor de los casos, tendremos que deshacer la pila hasta su elemento más profundo, con `n` llamadas recursivas, quedando en una complejidad $O(n)$

Complejidad del set

```
public void set(int pos, E e) {
    if ( pos == 1 ) {
```

```
        stack.pop();            // O(1)
        stack.push(e);          // O(1)
    } else {
        E top = stack.getTop(); // O(1)
        stack.pop();            // O(1)
        set(pos-1,e);           // Recursividad => O(n)
        stack.push(top);        // O(1)
    }
}
```

La complejidad de esta operación es $O(n)$. Del mismo modo, en el peor de los casos, tenemos que deshacer la pila hasta el elemento más profundo, con `n` llamadas recursivas hasta la posición `1`

Complejidad del insert

```
public void insert(int pos, E e) {
    if ( pos == 1 ) {
        stack.push(e);              // O(1)
    } else {
        E top = stack.getTop();     // O(1)
        stack.pop();                // O(1)
        insert(pos-1,e);            // Recursividad => O(n)
        stack.push(top);            // O(1)
    }
}
```

La complejidad de esta operación es $O(n)$. Del mismo modo, en el peor de los casos, tenemos que deshacer la pila hasta el elemento más profundo, con `n` llamadas recursivas hasta la posición `1`

Complejidad del remove

```
public void remove(int pos) {
    if ( pos == 1 ) {
        stack.pop();                // O(1)
    } else {
        E top = stack.getTop();     // O(1)
        stack.pop();                // O(1)
        remove(pos-1);              // Recursividad => O(n)
        stack.push(top);            // O(1)
    }
}
```

La complejidad de esta operación es $O(n)$. Del mismo modo, en el peor de los casos, tenemos que deshacer la pila hasta el elemento más profundo, con `n` llamadas recursivas hasta la posición `1`

Pregunta 2

Apartado a)

Siguiendo los principios de la programación orientada a objetos, se tienen que emplear los métodos públicos de la clase `Stack.java` para poder acceder correctamente a la estructura de pila del atributo `stack`. Si miramos los atributos de la clase `Stack.java`, vemos que los hereda de la clase abstracta `Sequence`, que tiene atributos y clases tanto `private` como `protected` (que en la práctica, al no estar heredando de esa clase en la clase `StackAsList`, es `private`). Así que no podemos intentar acceder a la estructura de pila directamente a través de atributos, aunque de poder tampoco sería una buena idea, salvo casos muy concretos.

Apartado b)

Para empezar, el recorrer la estructura de datos `StackAsList` con el iterador no debería eliminar elementos del contenedor. Para ello añadimos un atributo para controlar la posición en la que estamos, por ejemplo, `private int iteratorPos;` Para el método `hasNext`, comprobamos si el valor de `iteratorPos` es igual al tamaño de la pila, con el método de la clase `StackAsList`: `size` Para el método `getNext`, llamamos al método de la clase `StackAsList`: `get(iteratorPos)`, con el que tomamos el elemento para devolverlo, y aumentamos en una unidad el valor `iteratorPos` Para el método `reset`, simplemente asignamos el valor de `iteratorPos` al inicial, es decir, `iteratorPos = 1`

Apartado c)

Claramente el método `reset` tendría una complejidad $O(1)$, al utilizarse simplemente una asignación a un atributo El método `hasNext` también tiene complejidad $O(1)$ al hacer una comprobación entre el valor de `iteratorPos` y el tamaño de la pila El método `getNext` hace uso del método `get()` de la clase `StackAsList`, de complejidad $O(n)$, e incrementa el contador. Por lo tanto, tendrá complejidad $O(n)$

Usar directamente estos métodos en la clase `StackAsList` no tiene demasiado sentido, porque no nos estamos aprovechando de la estructura de lista. Si vamos a usar estos métodos, estaremos usando lógica de iterador, y por ello, lo adecuado sería instanciar un iterador y usar los métodos del iterador, aunque subyacientemente nos estemos apoyando en un `StackAsList`

Tarea 2

Pregunta 1

```
public static void dutchFlag(ListIF<Character> L) {
    // Con estos tres pivotes, tengo controlados los limites de los tres grupos de letras
    int firstPivot = 1;                 // O(1)   Superior del grupo de 'R'
    int secondPivot = 1;                // O(1)   Superior del grupo de 'B' => grupo del me
    int thirdPivot = L.size() + 1;      // O(1)   Inferior del grupo de 'A'

    // Los elementos del grupo del medio son los que estan entre el segundo y tercer pivote
    // Estos son los elementos que no han sido ordenados, asi que iteramos hasta agotar
    // este grupo de elementos no ordenados
    while(secondPivot < thirdPivot){
        // Tomamos el primer elemento del grupo que no ha sido ordenado
        switch(L.get(secondPivot)){
            // Tiene que ir al grupo de las R
            // Asi que lo movemos hacia la izquierda
            case 'R':
                swap(L, firstPivot, secondPivot);   // O(1)
                firstPivot = firstPivot + 1;        // O(1)
                secondPivot = secondPivot + 1;      // O(1)
                break;

            // Tiene que ir al grupo de las R
            // Asi que lo movemos a la derecha
            case 'A':
                thirdPivot = thirdPivot - 1;        // O(1)
                swap(L, secondPivot, thirdPivot);   // O(1)
                break;                              // O(1)

            // Esta bien colocado
            // Asi que avanzamos al siguiente elemento no ordenado
            default:
                secondPivot = secondPivot + 1;      // O(1)

        }
    }
}
```

El cuerpo del `switch` se ejecuta en $O(1)$, así que la complejidad la determina el número de veces que se ejecute el bucle `while`. Por la propia descripcion del problema, vemos que solo necesitamos realizar una pasada con los dos iteradores, hasta que `secondPivot` y `thirdPivot` se crucen. Viendo el `switch`, en cualquiera de los casos o se incrementa en uno `secondPivot` o se decrementa en uno `thirdPivot`, con lo cual, se necesitan a lo sumo `n` iteraciones. Así

llegamos a que la complejidad es `O(n)`

Esto suponiendo que la operación `swap` sea $O(1)$, que depende de que la implementación del `set` y `get` de la clase que implemente `ListIF` sea $O(1)$. Si fuesen de otra complejidad, habría que multiplicar asintóticamente las complejidades.

Por ejemplo: si el `get` y `set` fuesen de complejidad $O(n)$, entonces el `swap` tendría complejidad $O(4n) = O(n)$. En el peor de los casos, las `n` iteraciones no caen en el caso `default`, por tanto, se ejecuta la operación `swap` `n` veces, con lo que quedaría:

$$O(n * n) = O(n^2)$$

Pregunta 2

Apartado a)

El primer método lo que hace es un `bubble sort` con recursividad. Se hace una pasada desde la primera posición hasta la posición `s`. En esa pasada se hace un recorrido de `bubble sort`, así el último elemento queda ordenado (a la última posición llega el elemento con mayor valor según el `compare`)

Después, se llama recursivamente a `Faux` ignorando el último elemento, que ya está ordenado. Esto simula el decrementar el contador del primer bucle de un `bubble sort`. Se repite el proceso hasta llegar al primer elemento, habiendo hecho un bubble sort con recursividad. En vez de hacer esto la forma clásica iterativa es mucho más eficiente al ahorra el sobrecoste de llamada a una función

En el segundo método también se ordena una lista haciendo una modificación del clásico `bubble sort` con una pequeña mejora. Viendo el código, si en una pasada no hay cambios, no se vuelve a llamar a la función auxiliar (en el primer método, esto no ocurría, siempre se llamaba recursivamente hasta llegar al inicio de la lista). Esto equivale a la mejora del clásico `bubble sort` de controlar con un booleano si hay cambio, y en caso de no haberlo, hacer un `break` en el bucle más externo

Sin embargo, con este código, si se encuentra un cambio se corta la pasada y se inicia una nueva, aunque al terminar la llamada recursiva se continúa con la pasada

En ambos casos, la peor lista que se le puede dar como entrada es una lista ordenada de forma inversa (en este caso, de forma decreciente según el criterio del `compare`) porque en cada pasada se tienen que hacer `n` veces la operación `swap`

Apartado b)

Para el primer algoritmo, en el caso de tamaño `n`, hacemos una pasada por la lista (`n` operaciones), y llamamos a la función quitando el último elemento, con lo que queda

$$T(n) = n + T(n-1)$$

En el segundo caso, el pero caso es que en cada llamada el primer elemento esté mal ordenado y haya que cambiarlo y llamar recursivamente, con lo que quedaría:

$$T(n) = 1 + T(n-1)$$

En ambos casos, se tiene que $T(1) = 1$ al ser el caso base que se resuelve con un `return`

Apartado c)

En la primera recurrencia tenemos:

$$T(n) = T(n-1) + n$$
$$T(1) = 1$$

En este caso, no hace falta usar ecuaciones de resolución, basta con caer en la cuenta del siguiente desarrollo, expandiendo la función recursiva:

$$T(n) = T(n-1) + n = (n - 1 + T(n-2)) = \dots$$
$$\dots = (n - 1 + (n - 2 + T(n-3))) + n = \dots$$
$$\dots = n + (n-1) + (n-2) + \dots + T(1) = n + (n-1) + (n-2) + \dots + 1 = \dots$$
$$\dots = \sum_{i=1}^{n} i = \frac{n * (n-1)}{2} \approx O(n^2)$$

Donde en el último paso hemos usado la suma de los primeros n dígitos de Gauss

En la segunda recurencia tenemos:

$$T(n) = T(n-1) + 1$$
$$T(1) = 1$$

De nuevo, podemos resolver con la misma idea pues la recurrencia es muy sencilla:

$$T(n) = T(n-1) + 1 = T(n-2) + 1 + 1 = T(n-2) + 2 = \dots$$
$$\dots = T(n-3) + 3 = \dots$$
$$\vdots$$
$$= T(n-k) + k =$$
$$\vdots$$
$$= T(n - (n-1)) + (n-1) = T(1) + (n-1) = 1 + (n-1) = n \approx O(n)$$