2015

Documentación de la Práctica 3

ESTRUCTURA DE DATOS

PAU SANCHEZ VALDIVIESO Y ALBERT ESPÍN ROMÁN

| 22.04.2015

Índice

1. Implementación de las estructuras y el algoritmo de juego

A continuación explicamos las diferentes implementaciones de clases y módulos relevantes para la práctica.

1.1. LinkedList

LinkedList es la clase base para implementar el tipo de dato abstracto de la lista enlazada, que consiste en un conjunto de nodos, objetos que contienen un dato a almacenar y un enlace o referencia que indica la posición en memoria del siguiente nodo, lo cual permite superar la limitación de tener físicamente seguidos en memoria los datos a guardar; el último nodo tiene su referencia al siguiente vacía, mientras que el primer nodo se conoce como cabeza o *head* y es un atributo de la lista.

LinkedList consta de muchas funciones para llevar a cabo acciones similares a las de las listas de Python pero con una gestión interna diferente; así, por ejemplo, disponemos de los métodos de inserción *insertFirst()*, *insertLast()* e *inserti()* que añaden un nodo en una posición concreta (la primera, la última o la indicada por parámetro, respectivamente) con un dato pasado por parámetro, de modo que el enlace *next* del anterior nodo apunta al nuevo nodo y el de éste al que antes era el siguiente del previamente mencionado; en cuanto a la sustitución, posible con los métodos *replace()*, *replaceAll()* y *replacei()* se cambia el *data* de un nodo (el que tiene el dato pasado por parámetro, todos los que tengas ese dato o el de una posición indicada, respectivamente) por aquél que se pasa por parámetro; en cuanto a la eliminación dispone de los métodos *removeFirst()*, *removeLast()* y *removei()*, que permiten eliminar un nodo (el primero, el último o el indicado, respectivamente), de modo que el que era anterior pasa a tener por *next* el nodo que era el siguiente al que se elimina.

LinkedList cuenta también con otros métodos distintivos, como lo son *isEmpty()*, para comprobar si está vacía de nodos, *traverse()*, para mostrar por pantalla la sucesión de contenidos de los nodos y *find()*, que retorna si ha encontrado un dato indicado por parámetro contenido en algún nodo de la lista enlazada.

1.2. CircularLinkedList

La lista enlazada circular es una subclase de *LinkedList* con la propiedad especial que el último nodo apunta con su enlace al siguiente al nodo inicial, consiguiéndose en la práctica la circularidad de la lista, donde *head* sigue siendo el primer elemento mientras que *tail* indica el elemento final, que de otro modo sería indistinguible, ya que no posee un enlace al siguiente nulo como en el caso de la lista circular simple.

Los métodos de la lista enlazada circular son en esencia los de su superclase, si bien son sobrescritos para que cuando se inserta o elimina un elemento al final de la lista se enlace con el primero o cabeza como exige conceptualmente la circularidad de esta clase.

1.3. DoubleLinkedList

La lista enlazada doble es una subclase de la lista enlazada básica formada por nodos especiales que además de contar con un apuntado o enlazado hacia delante entre los nodos (cada uno tiene información explícita de su siguiente), incorporan una referencia a su predecesor en la lista

(*previous*), de modo que son nodos bidireccionales o instancias de la clase *TwoWayNode*, subclase del nodo básico *Node*.

Los métodos de la lista enlazada doble son en esencia los de su superclase, si bien son sobrescritos para que cuando se inserta o elimina un elemento se gestionen correctamente no sólo los enlaces al siguiente sino también aquellos que referencian al predecesor en la lista.

1.4. CircularDoubleLinkedList

La lista enlazada circular doble cuenta tanto con las características de la lista enlazada doble como con aquellas propias de la lista enlazada circular; como ambas clases descienden de la lista enlazada simple y se busca la simplicidad así como evitar las duplicidades y otros problemas que la herencia múltiple puede ocasionar, se ha optado por hacer heredar la clase que nos ocupa de *DoubleLinkedList*, porque al implementar ya ésta el tratamiento deseado de los nodos bidireccionales resulta más simple que hacerla subclase de *CircularLinkedList*, de la cual sólo hay que añadir en *CircularDoubleLinkedList* su conexión del enlace de siguiente del último elemento (*tail*) al primero (*head*) para disponer de toda la funcionalidad buscada.

En la clase del juego *ONE*, hemos implementado la lista de jugadores como una lista enlazada doble circular, y nos hemos servido de la circularidad y la posibilidad de consulta del elemento previo y siguiente de ésta para implementar eficientemente los métodos relacionados con esa lista de jugadores, como cambiar el turno con *change_turn()* (insertando el que fuera primer jugador al final de la lista, de modo que el que estaba segundo pasa a jugar), método usado repetidamente cuando aparece la carta especial *Skip*, que deja a un jugador sin turno, un efecto producido también al usar las cartas de robo de la baraja *+2* y *+4*. La lista de jugadores también está implicada en la inversión de turnos asociada a la carta *Reverse* e implementada en el método *invert_turns()*, que dispone de forma contraria a la previamente vigente el posicionamiento de los jugadores en los nodos de la lista enlazada y cambia el jugador del momento.

1.5. LinkedStack

LinkedStack conserva la naturaleza propia de la pila como tipo de dato abstracto al que se añaden y retiran los elementos por el final; el cambio realizado al hacer que heredara de la lista enlazada básico consiste en que el método de adición, *push()*, usa *insertLast()*, ya que se inserta al final, y que el método de substracción, *pop()*, aprovecha el método de *LinkedList removeLast()*.

1.6. LinkedQueue

La clase LinkedQueue ofrece la funcionalidad de la cola pero se implementa internamente a partir de LinkedList. Para cumplir el principio *first in first out* (*FIFO*) propio de la cola, el método de añadido, *enqueue()*, implementa *insertLast()*, mientras que el de extracción, *dequeue()*, usa *removeFirst()*.

1.7. LinkedPriorityQueue

La implementación de la cola de prioridad enlazada es muy similar a la de la cola de prioridad anterior, el único cambio que se ha requerido (teniendo en cuenta las modificaciones en la que

ahora es su superclase, *LinkedQueue*) ha sido añadir *getData()* para los *self[i]* del método *enqueue()*, ya que lo que forman la lista enlazada son nodos, pero queremos comparar su contenido (como las cartas, en el juego UNO).

1.8. ONE con cartas especiales opcionales

Esta clase es la principal del juego UNO, ya que lo ejecuta con la función *run_game()*, que comienza llamando a *prepare_game()* para mostrar una bienvenida, definir a los jugadores, y repartir las cartas iniciales una vez inicializado el mazo y la pila del juego y crenado un sistema de turnos por azar; posteriormente, mientras no se cumple la condición de fin de juego (*stop_criterion()*, que el jugador actual se quede sin cartas), éste, si puede tirar, tira hasta que se queda sin cartas válidas, en caso contrario roba cartas del mazo hasta que puede tirar, y se va mostrando el estado de juego en pantalla con *visualize_state()*; cuando el jugador se queda sin cartas válidas para jugar después de tirar, pasa su turno a otro con *change_turn()* de forma sucesiva hasta que en algún momento uno de los jugadores se queda sin cartas, cumpliendo la condición para que se acabe el juego, que muestra una felicitación con *announce_champion()*.

Cabe añadir que se ha ampliado un poco el volumen del código para la gestión de las cartas especiales; estas se detectan como tales y se ven gestionados sus efectos por la función *check_special_card()* de la clase *ONE*. Después que el jugador tire una carta se llama esta función, que comprueba si la carta visible de *Discard_Pile* (la última lanzada, por tanto) es especial (las especiales tienen en el atributo "*special*" de *Card* indicando su tipo de especialidad, una cadena de caracteres diferente a "", que indica que no son especiales): en caso que sea especial, analiza el tipo de especialidad y desarrolla sus efectos (si es la de dejar sin turno al siguiente jugador, llama dos veces a *change_turn()*, si es invertir el orden de juego, reestructura la lista de jugadores, revirtiendo su orden en *invert_turns()* y reasignando un jugador actual).

Cabe destacar que para lograr el objetivo opcional de permitir jugar a un número indefinido de jugadores sin que se dé un error por nulidad de elementos restantes en el mazo se ha optado por crear una función *calculate_card_factor()* que obtiene un factor numérico usado después para crear un número de cartas apropiado para el número de jugadores activos en el juego.

Se ha implementado también la carta especial "*ChangeColor*" que muestra al jugador un menú para elegir el color al que convertir una carta (con el método *get_color_from_menu()*), así como las cartas especiales "*+2*" y "*+4*", que dejan al siguiente jugador sin turno (con *change_turn()*) después de hacerle robar varias cartas (2 o 4, respectivamente), a través de llamadas al método *pick_card()*.

Conviene indicar que se ha detectado un posible problema en el desarrollo del juego, consistente en que después que se robaran muchas cartas el mazo quedara vacío y se produjera un error al intentar retirar otra carta de ahí; se ha solucionado aprovechando la función de *Queue isEmpty()*, que retorna si el objeto está vacío, sin elementos, de tal forma que antes que el jugador robe una carta del mazo se comprueba si éste no está vacío: si lo está, se llama a un método *fill_deck()* que rellena de cartas el mazo a partir de todas las cartas de *Discard_Pile* menos la última (la visible, la carta en juego), unas cartas que no se hubieran vuelto a usar de otro modo, de forma que resulta más óptimo que crear nuevas cartas que puedan ocupar más espacio de memoria.

2. Análisis de las ventajas y desventajas en el uso de listas enlazadas y listas de Python

Las principales ventajas de las listas de Python se relacionan con la velocidad de acceso a los contenidos de las diferentes posiciones a través de los índices: los elementos están dispuestos físicamente consecutivos en la memoria, y para acceder a un elemento concreto a Python le basta con encontrar su posición con una simple fórmula (posición base de la lista sumando el producto entre el índice y el número de bits de ocupa cada posición), por lo que operaciones como la sustitución de un elemento por otro se llevan a cabo de forma muy optimizada. Sin embargo, las listas tienen diversos problemas; una de las desventajas que conllevan es la poca eficiencia de las operaciones de inserción y eliminación (ya que hay que desplazar arriba o abajo en la lista, respectivamente, los elementos posteriores al de la posición indicada), de modo que se llevan a cabo, de media, en tiempo O(n), donde n es la longitud de la lista. El otro gran problema de las listas deriva precisamente del carácter físicamente consecutivo de sus elementos: si se añaden muchos elementos, puede ser necesario tener que migrar todo el contenido de la lista a una zona con más memoria libre, y se puede desaprovechar memoria si se eliminan muchos elementos o se da un tamaño físico muy superior al lógico (al de elementos efectivamente no nulos en la lista).

La lista enlazada supone una solución para esos problemas de la lista, ya que consiste en un conjunto de objetos (nodos) que no están unidos físicamente, sino que el primero contiene una referencia al siguiente, que puede estar ubicado en otra parte de la memoria, y así sucesivamente. Las ventajas que esto aporta son evidentes: no hay limitación en el número de los elementos respecto a un sector de memoria fijo, ya que cada nodo puede emplazarse en una ubicación distinta; además la inserción se simplifica a tiempo O(i), ya que sólo se necesita hacer que el nuevo nodo apunte como siguiente al de la posición indicada, y que el que era su anterior apunte como siguiente al nuevo nodo, no hace falta alterar de ningún modo al resto de los elementos; en cuanto a la eliminación, la mejora es de grado equivalente, ya que simplemente hay que hacer que el nodo de posición anterior a la indicada apunte ahora con su referencia al nodo que era siguiente al que se quita.