Coradini Elias y Milessi Ayrton

Universidad Nacional de Entre Ríos

Facultad de Ingeniería

Tecnicatura Universitaria en Procesamiento y Explotación de Datos

“Algoritmos y Estructuras de Datos”

Dr. Javier Eduardo Diaz Zamboni

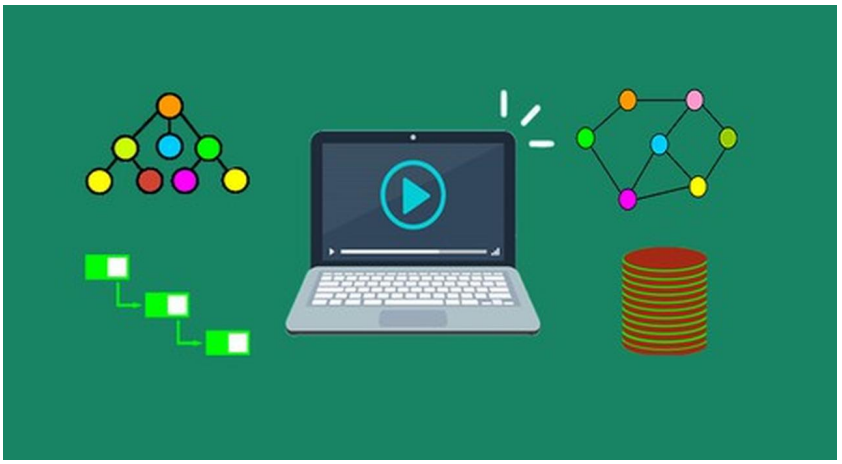
Bioing. Jordán F. Insfrán

Bioing. Diana Vertiz del Valle

Bruno M. Breggia

Trabajo Práctico N°1

Oro Verde - 2023



Actividad N°1

Se nos pedía implementar una Lista Doblemente Enlazada, usando la estructuras de datos de nodos doblemente enlazados. Con ella, también se requería que la estructura permitiera realizar ciertas operaciones.

Lo primero que hicimos fue definir una clase llamada “Nodo”. En él, se asignaba el dato a cada nodo y se los conectaba con el anterior y el siguiente.

En la segunda parte, definimos una clase llamada “ListaDobleEnlazada” en la cual se hacen todas las operaciones solicitadas:

- La función “esta_vacia” verifica si la lista está vacía. Devuelve True si la lista no tiene datos.
- La función “__len__” devuelve el tamaño de la lista. Esta función retorna el valor de la variable ‘tamanio’, esta variable aumenta en el caso de que se agregue un nodo en la lista y decrementa cuando se extrae un nodo.
- La función “agregar_al_inicio” permite agregar un dato en la cabeza de la lista. Si la lista está vacía, lo asigna directamente en la cabeza. En cambio si la lista no está vacía, se guarda el nuevo dato en la variable temporal ‘aux’. Al nodo siguiente de ‘aux’, se le asigna el valor de la cabeza. Luego el nodo anterior a cabeza, se le asigna ‘aux’ y finalmente se asigna a cabeza el nuevo dato agregado.
- La función “agregar_al_final” agrega un nuevo dato en la cola de la lista. Hace el mismo procedimiento que la función anterior, pero en vez de trabajar con la cabeza de la lista, esta función trabaja con la cola.
- La función “insertar” agrega un nuevo dato en una posición específica. A través de unos condicionales, se determina donde se ubicará el nuevo dato:
  - Si la lista está vacía, será insertada en la cabeza.
  - Si la posición no fue agregada por el usuario o es el mismo número que el tamaño de la lista, se insertará en la cola.

- ➢ Si la posición ingresada es igual a 0, se insertará en la cabeza.
- ➢ Pero si no es ninguna de las anteriores, mientras la posición sea mayor a 0 y menor al tamaño, se recorre la lista hasta la posición ingresada por el usuario, luego se inserta el dato y se reacomodan los apuntadores.
- ➢ En caso de que la posición ingresada no corresponda con ninguno de los casos anteriores se le dará un mensaje de error al usuario, por ingresar un valor mayor al tamaño de la lista.
- ➢ Al finalizar, al tamaño de la lista se le suma 1.

❖ La función “extraer” elimina el dato de la posición ingresada. Como la función anterior, a través de condicionales para saber el valor de posición y donde ingresa para realizar el procedimiento adecuado para que siga siendo una lista doblemente enlazada. Al finalizar le resta 1 al tamaño de la lista y retorna el dato extraído.

❖ La función “copiar” hace una copia profunda de la actual lista. Comienza creando una nueva lista vacía. Luego, recorre la lista original con un bucle while y agrega cada elemento de la lista original al final de la lista copiada con la función “agregar_al_final”. Finalmente, la función devuelve la lista copiada, manteniendo la estructura de lista doblemente enlazada.

❖ La función “invertir” invierte los datos de la lista. Esta comienza estableciendo dos punteros, “comienzo” en la cabeza y “final” en la cola. A continuación, se entra en un bucle while, este bucle se ejecuta mientras los punteros no se encuentren o crucen en el medio de la lista. Dentro del bucle, los datos a los que apuntan comienzo y final se intercambian. Luego de cada intercambio el puntero comienzo se mueve un nodo hacia delante y el puntero final se mueve un nodo hacia atrás.

❖ La función “ordenar” es una implementación del ordenamiento “Quick Sort”. Para poder usar este tipo de ordenamiento, lo que hicimos fue:

- ➢ Verificar si la lista no está vacía o si tiene un solo elemento. Si cumple las condiciones, llama a la función “ordenacion_auxiliar”.
- ➢ Esta nueva función, tiene que cumplir una condición, en la que el primero sea menor que el último. Al cumplir esta condición se determina el punto de división en la lista con la función “particion”.
- ➢ Esta función asignará el pivote y las marcas izquierda y derecha para encontrar el punto donde la lista se debe partir para luego ordenarla. Aparta los datos menores al pivote con el pivote por un lado, y por otro los mayores al pivote. Esta función retornará la posición del pivote.
- ➢ La función “ordenacion_auxiliar” se llama recursivamente para ordenar la lista.

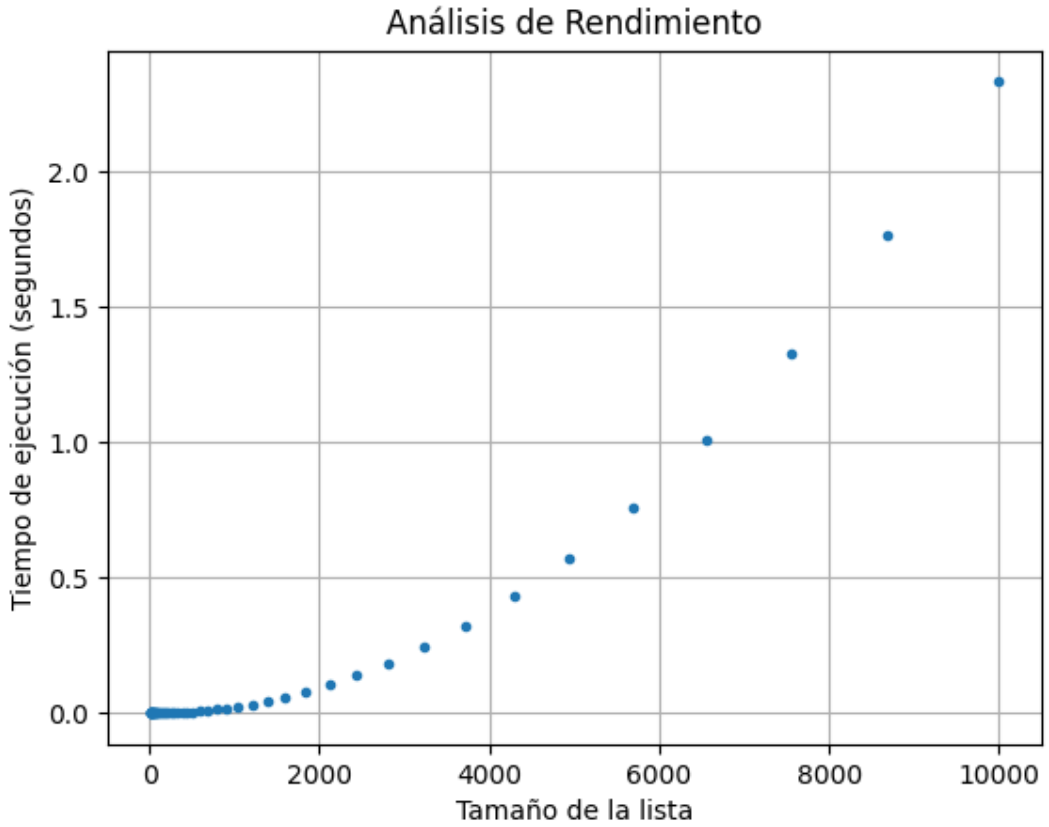
❖ La función “concatenar” posibilita “sumar” una nueva lista a la lista doblemente enlazada. Lo que hicimos fue verificar si ninguna está vacía. Si las dos tienen elementos, la lista a “sumar” se copiará en la variable ‘copy2’ y luego se le agrega en la cola de la lista a concatenar, estableciendo los enlaces correspondientes.

❖ La función “__add__” se establece para poder usar el operador “+” para concatenar dos listas doblemente enlazadas.

❖ La función “__iter__” es un método que usamos para poder iterar una clase. Se hace uso de yield, ya que es mucho más eficiente que hacerlo con un return.

❖ La función “__str__” fue implementada para que en la siguiente actividad (en el juego de guerra) nos permita imprimir de manera correcta las cartas.

Como solicitaba la consigna, realizamos una función que gráfica el rendimiento de nuestro método de ordenamiento con la lista doblemente enlazada. Elegimos el método de ordenamiento quicksort ya que es rápido y de complejidad O(n log n) (aunque puede degradarse a O(n2)).



Como se puede observar nuestro ordenamiento cumple con lo esperado.

Actividad N°2

Para esta actividad, se nos solicitó que simuláramos el juego de cartas "Guerra". Para poder realizarlo importamos la lista doblemente enlazada de la actividad anterior para poder usarlo en el mazo. Mientras que el mazo contenía las cartas con su valor y palo correspondiente. También importamos la biblioteca random para usar la función 'shuffle' la cual nos permite barajar el mazo de una manera más aleatoria. Además, se usaron funciones de la lista doblemente enlazada para poder utilizarlas en el juego. Estas funciones son:

- **"agregar_al_inicio":** Para poner arriba del mazo la carta correspondiente.
- **"agregar_al_final":** Para agregar una carta, pero esta vez en la parte de abajo del mazo.
- **"esta_vacia":** Sirve para verificar que el mazo de un jugador no este vacio, ya que al estarlo significa que se quedó sin cartas para seguir jugando.
- **"extraer":** Se utiliza para sacar la carta de arriba del mazo de cada jugador para luego hacer la comparación de quién ganó la ronda.

Todo esto se colocó en la clase 'Mazo'.

Posteriormente, se declaró otra clase, pero esta vez el de 'JuegoDeGuerra'. Este contiene funciones como:

- **"__init__":** el constructor que almacena las variables del ganador del juego, el máximo de turnos de la partida y el número de turnos jugados.
- La función **"imprimir_mazo"** imprime el mazo sin mostrar las cartas.
- La función **"imprimir_botin_guerra"** imprime el botín que hay en cada ronda. El botín es una variable que almacena todas las cartas que se llevará el ganador de la guerra. Lo que posibilita esta función es que puede mostrar por la consola cierta cantidad de guerra seguidas, lo que es muy poco probable, pero puede suceder.

- La función **“jugar_ronda”** es de las más importantes porque acá es donde se desarrolla la lógica del juego. La primera parte, el código extrae del mazo de cada jugador una carta. Luego que pase un condicional (verifica que la carta extraída no sea None porque si es, significa que un jugador no tiene más cartas en su mazo), se muestra el estado actual del juego como el número del turno, el mazo de cada jugador y las cartas extraídas anteriormente. La segunda parte es la de comparación de las cartas extraídas, se encuentra el índice del valor de la carta y se le suma un 2 ya que si sale un 2♠, su índice va a ser 0, pero sumándole 2, su valor final será de 2 (así sucede con todas las cartas. Otro ejemplo es la A♠, su valor final será 14). Luego hay una serie de condicionales para ver qué carta es la que vale más, si la de jugador 1, la del jugador o si hay un empate. En el caso de que haya empate, se lanza un mensaje de Guerra y se llama a la función **“guerra”**.

- La función **“guerra”** tiene como primera parte la comprobación de que los dos jugadores tengan las suficientes cartas para desarrollar la guerra. Si pasan esos condicionales, se guardarán las cartas que dieron el resultado de guerra en la variable ‘botin’. En la línea 189 es donde se encuentra la variable ‘hay_guerra’, esta variable es la que posibilita de que si hay más de una guerra seguida, se siga repitiendo esta función para que cumpla con la lógica del juego, ya que si esta en True, el while siempre se ejecutará. Dentro de este bucle, está anidado un bucle for que su rol es sacar las tres cartas correspondientes a cada jugador, mientras que las saca, la va almacenando en **‘botin’**. Cuando se sale de este bucle, se sacan una carta más a cada jugador (son las nuevas cartas que se comparan para saber quién ganará la guerra) y también son guardadas en ‘botin’. Luego se hace lo mismo que en la función “**jugar_ronda**”, se hace la lógica para que se imprima los mazos de cada jugador y el botin en juego (las primeras dos cartas que dieron como resultado el empate, las seis cartas y las últimas dos que son las que definirán el ganador de la guerra). Seguidamente de esto, se hace la serie de condicionales para verificar el ganador de la ronda o si hay que ir a otra guerra.

- En la función **“game_play”** se llama a la clase **‘Mazo’**, a las funciones barajar y repartir para que se pueda desarrollar el juego correctamente. Se establece un bucle while para controlar el límite de rondas y poder desarrollar las rondas.

Para finalizar, está un condicional que imprime el ganador en el caso que exista o el mensaje de empate en el caso de que se haya llegado al máximo de rondas.

Actividad N°3

En este problema se nos pidió que hagamos un ordenamiento externo (mezcla directa). A continuación una breve explicación de nuestro código:

La función “mezcla_directa” toma un archivo de entrada y un tamaño de bloque como parámetros. Lee los datos del archivo de entrada en una lista, divide en bloques de tamaño “B”, ordena cada bloque, escribe los bloques ordenados en archivos temporales y luego fusiona estos bloques en un archivo de salida. Finalmente, elimina los archivos temporales.

Función “ord_bloques” toma una lista de nombres de archivos de bloques como entrada. Abre estos archivos, lee el primer valor de cada archivo, encuentra el valor mínimo entre los valores leídos, lo agrega a una lista ordenada, avanza al siguiente valor en el archivo correspondiente y repite este proceso hasta que todos los bloques se lean por completo. Luego, cierra los archivos de bloques y devuelve la lista ordenada.

La función “verificaciones” lee los datos del archivo original y el archivo ordenado en listas, luego compara si son iguales y también compara los tamaños de los archivos para verificar si coinciden. Devuelve una tupla con dos booleanos: uno para la verificación de orden y otro para la verificación de tamaño.

Conclusiones:

El manejo de estructuras de datos como la lista doblemente enlazada son muy importantes ya que te permiten hacer una variedad de acciones con este tipo de estructura. Además, al tratarse de una gestión de datos posiblemente muy grande, constantemente se tuvo en cuenta la eficiencia del código para que pueda ser muy bajo su costo temporal y espacial.

Por otro lado, la representación de un juego de cartas en código implica mucho diseño y lógica en el algoritmo, lo que fue difícil de llevar a cabo. Pero con la implementación de clases y funciones bien definidas, donde cada una cumplía un rol primordial para que la lógica del juego se pueda desarrollar como corresponde, se pudo finalizar el algoritmo. Además, la buena legibilidad que le dimos al código para que pueda mostrar en la consola el estado actual de cada ronda, nos hizo dar cuenta aun más de las cosas que teníamos que corregir y de las cosas que ya estaban bien hechas.

La última actividad fue un reto, ya que no sabíamos cómo teníamos que realizarla, sin embargo, cuando la fuimos entendiendo fue bastante simple a comparación del algoritmo de guerra. Esta actividad nos enseña a como ordenar de manera eficiente un gran tamaño de datos en pocos segundos y con poca memoria.