Rony Mauricio Rojas Aguilar – 202031191
Práctica 1 - EDD

COMPLEJIDAD DE SERVICIOS CRITICOS

1) Ingreso de jugadores y sus fichas:

```
void Partida::agregarJugadores() {
    int cantidadJugadores;                  1
    cout << "\nIngrese la cantidad de jugadores: ";
    cin >> cantidadJugadores;               1

    while (cantidadJugadores < 2) {         1
        cout << "Debe haber al menos 2 jugadores, intentalo nuevamente: ";
        cin >> cantidadJugadores;           1
    }

    for (int i = 1; i <= cantidadJugadores; i++) { n
        string nombreJugador;                       n
        cout << "Ingrese el nombre del jugador " << i << ": "; n
        cin >> nombreJugador;                       n

        Jugador nuevoJugador; // creando jugador    n
        nuevoJugador.setNombre(nombreJugador); //agregandole nombre   n
        jugadores->encolar2(nuevoJugador); // encolando al jugador    n
    }
    cout << "\nJugadores registrados\n" << endl;   n
}
```

La complejidad es O(n) porque el for se recorre “n” veces y por lo tanto no puede ser constante, esto en el peor de los casos.

2) Gestión de turnos con cola:

```
void Partida::cambiarTurno() {
    if (jugadores->isVacio()) {      1
        cout << "No hay jugadores en la cola." << endl; 1
        return;                      1
    }
    // desencolando al jugador
    jugadorActual = jugadores->desencolar2();   1
    //volviendolo a encolar para ciclarlo
    jugadores->encolar2(jugadorActual);         1
}
```

La complejidad es O(n).

3) Inserción y eliminación de fichas en lista enlazada:

-Eliminación:

```
T *eliminar2(int indice) {
    //elimina el dato y lo devuelve por el indice recibido
    if (indice < 0 || indice >= tamano) {  1
        return nullptr; // Índice no válido   1
    }
    Nodo<T> *aux = cabeza;          1
    Nodo<T> *previo = nullptr;      1

    for (int i = 0; i < indice; i++) {    n
        previo = aux;                     n
        aux = aux->getNext();             n
    }
    if (previo != nullptr) {              1
        previo->setNext(aux->getNext());     1
    } else {
        cabeza = aux->getNext(); 1
    }
    T *dato = aux->getData();     1
    delete aux;    1
    tamano--;      1
    return dato;   1
```

La complejidad es O(n).

-Inserción:

```
void insertar2(T *value) {
    //viendo que el puntero no sea nulo
    if (value == nullptr) {   1
        return;       1
    }
    // creando nuevo nodo
    auto *nuevoNodo = new Nodo<T>(*value); 1
    if (this->cabeza == nullptr) {    1
        this->cabeza = nuevoNodo;    1
        tamano++;    1
        return;       1
    }
    Nodo<T> *aux = this->cabeza;       1
    while (aux->getNext() != nullptr) {  n
        aux = aux->getNext();        n
    }
    aux->setNext(nuevoNodo);       1
    tamano++;     1
}
```

La complejidad es O(n).

4) Ordenación de fichas por puntuación:

```
Nodo<Letra> *Jugador::mergeSort(Nodo<Letra> *cabeza) {
    if (cabeza == nullptr || cabeza->getNext() == nullptr) { 1
        return cabeza; // si la lista esta vacía o con un elemento, ya esta ordenada
    }

    // dividiendo la lista en dos mitades
    Nodo<Letra> *mitad = dividirLista(cabeza);   log n

    // ordenando ambas mitades de manera recursiva
    Nodo<Letra> *izquierda = mergeSort(cabeza); n
    Nodo<Letra> *derecha = mergeSort(mitad);   n

    // fusionar las mitades ya ordenadas
    return fusionarListas(izquierda, derecha);   n
}
```

```
Nodo<Letra> *Jugador::dividirLista(Nodo<Letra> *cabeza) {
    if (cabeza == nullptr) return nullptr;   1

    Nodo<Letra> *lento = cabeza;   1
    Nodo<Letra> *rapido = cabeza->getNext();  1

    // avanzando rapido dos pasos y lento un paso, para obtener el largo de la lista y la mitad
    while (rapido != nullptr && rapido->getNext() != nullptr) {  n
        lento = lento->getNext();   n
        rapido = rapido->getNext()->getNext();   n
    }

    // sepraando la lista a la mitda
    Nodo<Letra> *mitad = lento->getNext();   1
    lento->setNext(nullptr); // desvinculando la mitad de la lista original y devolviendola 1
    return mitad;    1
}
```

```
Nodo<Letra> *Jugador::fusionarListas(Nodo<Letra> *izquierda, Nodo<Letra> *derecha) {
    if (izquierda == nullptr) {  1
        return derecha;     1
    }
    if (derecha == nullptr) {    1
        return izquierda;       1
    }
    Nodo<Letra> *resultado = nullptr;  1
    // comparando puntajes de derecha e izq y fusionando listas
    if (izquierda->getValue().getPunteo() >= derecha->getValue().getPunteo()) {  1
        resultado = izquierda;      1
        resultado->setNext(fusionarListas(izquierda->getNext(), derecha));    n
    } else {
        resultado = derecha;    1
        resultado->setNext(fusionarListas(izquierda, derecha->getNext()));  n
    }
    return resultado;  1
}
```

Complejidad total del algoritmo Merge Sort O(n log n).
Esto sucede ya que Dividir lista tiene una complejidad O(n) y con las llamadas recursivas son O(log n).
Por lo que al final es O(n log n)

5) Registro de palabras jugadas con pila:

```
void push(T data) {
    //creando nodo dado el dato recibio, actualizando cima con el dato recibido
    auto *nuevoNodo = new Nodo<T>(data); 1
    nuevoNodo->setNext(this->cima);  1
    this->cima = nuevoNodo;  1
}
```

La complejidad es O(1).

6) Calculo y ordenación de puntuaciones:
nullptr jsj

7) Ordenación de palabras iniciales:

```
ListaEnlazada<Palabra> *Archivo::ordenarAlfabeticamente() {
    //mandando a leer el archivo y guardar las palabras en listaPalabras
    ListaEnlazada<Palabra> *listaPalabras = this->leerCSV("../util/palabras.csv"); 1
    if (!listaPalabras || listaPalabras->isEmpty()) {   1
        return listaPalabras;  1
    }
    bool intercambio;  1
    do {
        //usando algoritmo bubble sort para la ordenacion
        intercambio = false;//para ver si se intercambio una palabra en cada recorrido del do while  n
        Nodo<Palabra> *actual = listaPalabras->getCabeza();//obteniendo la cabeza de la lista     n
        while (actual != nullptr && actual->getNext() != nullptr) {  n2
            //mientras hayan palabras en la lista
            Nodo<Palabra> *siguiente = actual->getNext();//guardando el nodo siguiente del nodo actual  n2
            if (actual->getValue().getContenido() > siguiente->getValue().getContenido()) {  n2
                //comparando contenido del actual con el del siguiente
                //aplicando intercambio de nodos
                Palabra temp = actual->getValue();//guardando el actual temporalmente  n2
                actual->setValue(siguiente->getValue());//en actual se guarda el siguiente  n2
                siguiente->setValue(temp);//en siguiente se guarda el temporal  n2
                intercambio = true;    n2
            }
            actual = actual->getNext();//guardando el siguiente en actual  n
        }
        //ciclando hasta que ya no haya intercambio
    } while (intercambio);  n
    return listaPalabras; //devolviendo la lista ordenada alfabeticamente  1
}
```

La complejidad es O(n^2).

Rony Mauricio Rojas Aguilar – 202031191
Práctica 1 - EDD

ARGUMENTACION DE METODOS DE ORDENAMIENTO

1) Fisher Yates:

El algoritmo de desordenamiento Fisher Yates es un algoritmo para generar una permutación aleatoria de un conjunto finito.
Use este algoritmo porque mezcla los elementos sin usar o crear una estructura adicional y asi ahorrar espacio en memoria.

Implemente este algoritmo para mezclar los elementos de la cola de jugadores una vez registrados. Los pasos para de este algoritmo implementados son:

1. Obtener el tamaño de la cola de los jugadores
2. Recorrer la cola con un ciclo, y en cada posición de la cola generar un índice aleatorio
3. En el índice de la posición aleatoria intercambiar el índice actual por el índice generado aleatoriamente.
4. Realizar esto con cada posición del arreglo para resultar con una cola mezclada.

2) Merge Sort:

El algoritmo Merge Sort consiste en dividir la lista en varias partes, luego estas partes se ordenan, y, estas partes se mezclan entre ellas de forma ordenada.
Use este algoritmo porque es muy eficiente con listas grandes, ya que el archivo de entrada puede ser de muchas palabras y por lo mismo pueden ser muchas las palabras a ordenar.
Implemente este algoritmo para ordenar las letras cada jugador según su punteo. Los pasos para seguir este algoritmo son:

1. Se recibe una lista y se divide en 2 mitades.
2. Se recorre una lista con 2 apuntadores, uno avanza a cada dato y el otro apuntador avanza a cada 2 nodos.
3. Al finalizar este recorrido se habrá obtenido el nodo que está a la mitad y el otro nodo esta en el final de lista.
4. Se devuelve el apuntador que está en la mitad y de esa manera ya se tiene el apuntador que esta en la cabeza y el apuntador que está a la mitad.
5. Luego de esto se aplica recursión a cada una de estas mitades para llegar a tener listas de un solo elemento.
6. Luego de esto se llama a otro método que fusiona estas listas ir comparando dato por dato y así determinar al mayor.
7. Se fusionan las listas ordenadas una por una y así se resulta con una lista completa ordenada eficientemente.

3) Bubble Sort:

Es un algoritmo sencillo de ordenamiento que busca mediante repetidas comparaciones de los elementos adyacentes ordenar un conjunto de elementos.

Implemente este algoritmo para ordenar las palabras leídas del archivo CSV, de manera alfabetica.
Use este algoritmo porque no crea o necesita de estructuras extra para ordenar, todo lo hace en la misma estructura.
Los pasos que implemente para seguir este algoritmo son:

1. Recorro la lista de palabras con el datoActual y el datoSiguienteActual.
2. En cada vuelta comparo el datoActual y el datoSiguienteActual.
3. Si actual < siguiente, se intercambian de posición.
4. En la siguiente vuelta el actual se vuelve el siguiente y el siguiente se vuelve el siguiente del siguiente.
5. Recorrer la lista hasta que ya no haya siguiente y de esta manera la lista estará ordenada de mayor a menor.