# Documentación Técnica.

## Servicios Críticos.

1. **Ingreso de Jugadores y Fichas.**

```
/**
 * Metodo para ingresar jugadores al juego, y encolarlos en una estructura(Cola)
 * Complejidad O(n), n es el numero de jugadores. ya que solo se aceptan numeros mayores a
 * la complejidad en el mejor de los casos será O(2). Y en el pero de los casos O(infinito
 */
void Game::ingresar_jugadores()
{
    int numeroJugadores = -1;
    while (true)
    {
        std::cout << "Ingrese la cantidad de jugadores >>> ";
        std::cin >> numeroJugadores;

        //validar si la entrada es un entero
        if (std::cin.fail())
        {
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            numeroJugadores = -1;
        }
        if (numeroJugadores > 1) break;

        std::cout << "!El numero de jugadores debe ser como mínimo de 2¡" << std::endl;
    }

    //Ingresar el nombre de los jugadores
    int aux = 0;
    while (aux < numeroJugadores)
    {
        std::string nombre = " ";
        std::cout << "Ingrese el nombre de su jugador >>> ";
        std::cin >> nombre;
        jugadores_en_juego_queue.enqueue(Jugador(nombre)); // O(1)
        aux++;
    }
    asignar_turnos();
}

/**
 * Metodo para asignar turnos a los jugadores de manera aleatoria.
 * El algoritmo depende de cuantos elementos se encuentran en la cola.
 * Complejidad: O(n)
 */
void Game::asignar_turnos()
```

```
{
    const int n = jugadores_en_juego_queue.size();
    auto* arr = new Jugador[n];

    // O(n)
    for (int i = 0; i < n; ++i)
    {
        arr[i] = jugadores_en_juego_queue.dequeue(); //O(1)
    }

    //O(n)
    //Implementar el algoritmo de Fisher-Yates para mezclar el array. swap
    for (int i = n - 1; i > 0; i--)
    {
        int j = Utilidad::getRandomInt(0, i);
        //intercambiar arr[i] y arr[j]
        auto temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    //reencolar los elementos mezclados en la cola.
    // O(n)
    for (int i = 0; i < n; i++)
    {
        jugadores_en_juego_queue.enqueue(arr[i]); //O(1)
    }

    //liberar la memoria del arreglo
    delete[] arr;
}
```

2. **Gestión de turnos por cola.**

Anteriormente se debe obtener el frente de la cola en una variable auxiliar

```
//cambiar de jugador
actual = jugadores_en_juego_queue.dequeue(); //O(1)
jugadores_en_juego_queue.enqueue(actual); //O(1)
```

3. **Inserción y eliminacion de fichas en lista enlazada**

```
/**
 * Metodo para agregar un elemento al final de la lista.
 * @param data elemento a agregar a la lista.
 * O(n)
 */
template <typename T>
void LinkedList<T>::insertAtEnd(T data)
{
    // auto* temp = new Node;
    // temp->next = nullptr;
```

```
    // temp->value = data;
    auto* temp = new Node(data);
    if (this->head == nullptr)
    {
        this->head = temp;
    }
    else
    {
        auto* pointer = this->head;
        while (pointer->next)
        {
            pointer = pointer->next;
        }
        pointer->next = temp;
    }
    ++this->_size;
}

/**
 * Funcion que sirve para eliminar un elemento de la lista segun dado
 * un indice el cual debe estar dentro del rango de 0 y el tamanio de la lista -1
 * @param index indice del elemento a eliminar
 * @return valor booleano para la facil gestion de errores
 */
template <typename T>
bool LinkedList<T>::deleteAt(int index)
{
    // Verificar si el índice está dentro del rango válido
    if (index < 0 || index >= _size)
    {
        return false;
    }

    if (index == 0)
    {
        deleteAtHead();
    }
    else if (index == _size - 1)
    {
        deleteAtEnd();
    }
    else
    {
        // Eliminar un nodo en el medio de la lista
        Node* current = this->head;
        // Recorrer hasta el nodo anterior al que se desea eliminar
        for (int i = 0; i < index - 1; ++i)
        {
            current = current->next;
        }
```

```
        Node* aEliminar = current->next;
        current->next = aEliminar->next;
        delete aEliminar; // Liberar la memoria del nodo
        --this->_size; // Decrementar el tamaño de la lista
    }
    return true;
}
```

4. **Ordenar puntuaciones y palabras iniciales**

```
/**
 * Ordenamientio burbuja
 * Complejidad O(n^2)
 */
template <typename T>
void LinkedList<T>::bubble_sort()
{
    // la lista ya esta ordenada si solo tiene un elemento
    if (head == nullptr || head->next == nullptr)
    {
        return;
    }

    bool cambio;
    Node* actual;
    Node* cola = nullptr;

    do
    {
        cambio = false;
        actual = head;

        while (actual->next != cola)
        {
            // Usar el operador < para comparar elementos
            if (actual->value > actual->next->value)
            {
                // Intercambiar valores
                std::swap(actual->value, actual->next->value);
                cambio = true;
            }
            actual = actual->next;
        }
        cola = actual; // El último nodo visitado ahora está ordenado
    }
    while (cambio);
}
```