

Ayudantía 1: STL y Juez virtual

Juez virtual

The screenshot shows a virtual judge interface for a competition titled "Listado 1: TDAs". The competition details are as follows:

- Begin:** 2024-08-07 10:20 UTC-4
- End:** 2024-08-28 10:20 UTC-4
- Elapsed:** 15:06:17:35
- Status:** Running
- Remaining:** 5:17:42:24

The navigation bar includes links for Workbook, User, Group, and Forum. Below the status bar, there are tabs for Overview, Problem, Status, Rank (15:06:17:31), 0 Comments, and Setting. A dropdown menu shows options A, B, C, D, E, F, G, and H.

A - Game of Throws

Daenerys frequently invents games to help teach her second grade Computer Science class about various aspects of the discipline. For this week's lesson she has the children form a circle and (carefully) throw around a petrified dragon egg.

The n children are numbered from 0 to $n - 1$ (it is a Computer Science class after all) clockwise around the circle. Child 0 always starts with the egg. Daenerys will call out one of two things:

1. a number t , indicating that the egg is to be thrown to the child who is t positions clockwise from the current egg holder, wrapping around if necessary. If t is negative, then the throw is to the counter-clockwise direction.
2. the phrase `undo m`, indicating that the last m throws should be undone. Note that `undo` commands never undo other `undo` commands; they just undo commands described in item 1 above.

For example, if there are 5 children, and the teacher calls out the four throw commands `8 -2 3 undo`,

Submit

Status My Status

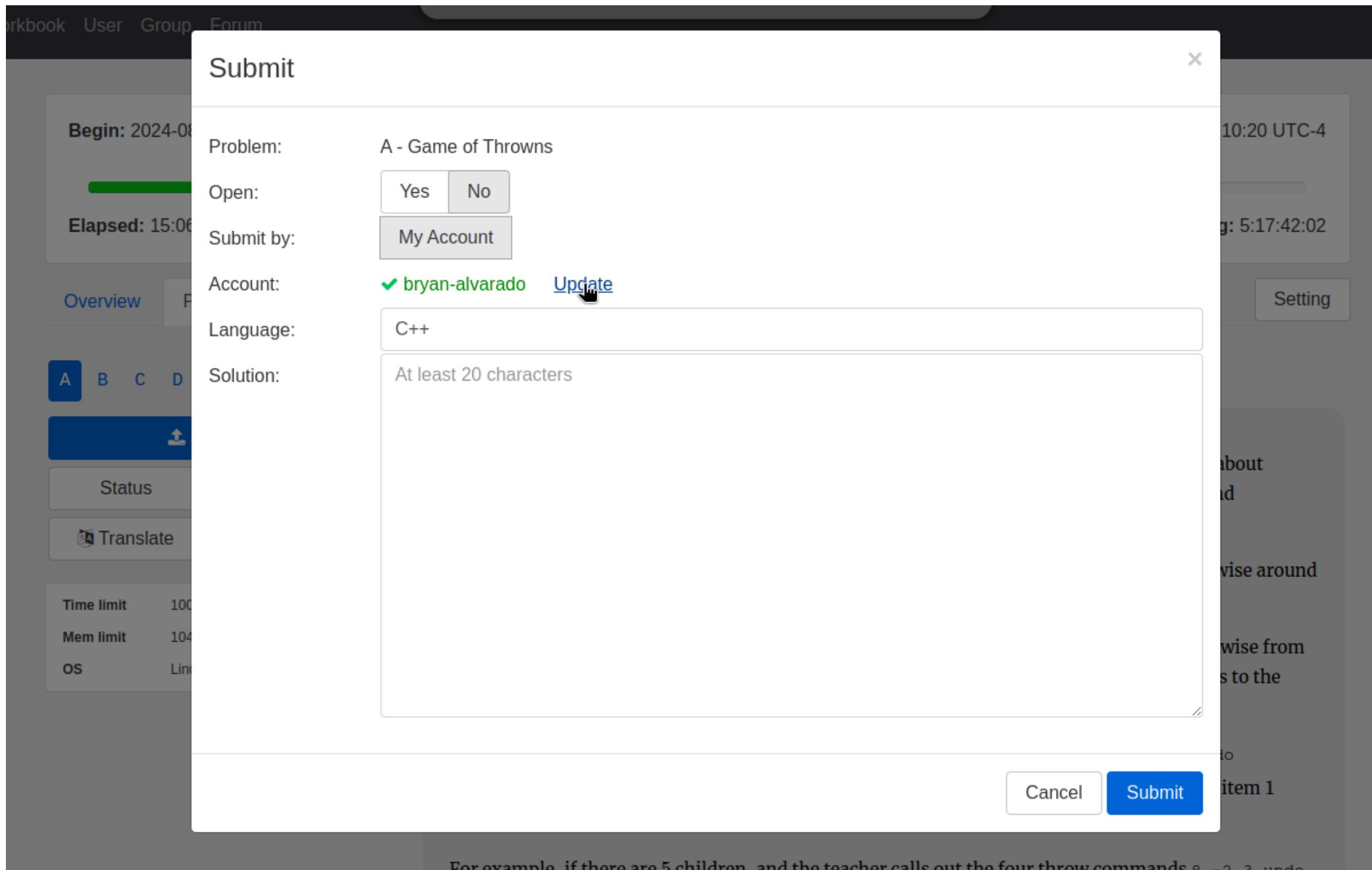
Translate PDF

Time limit 1000 ms

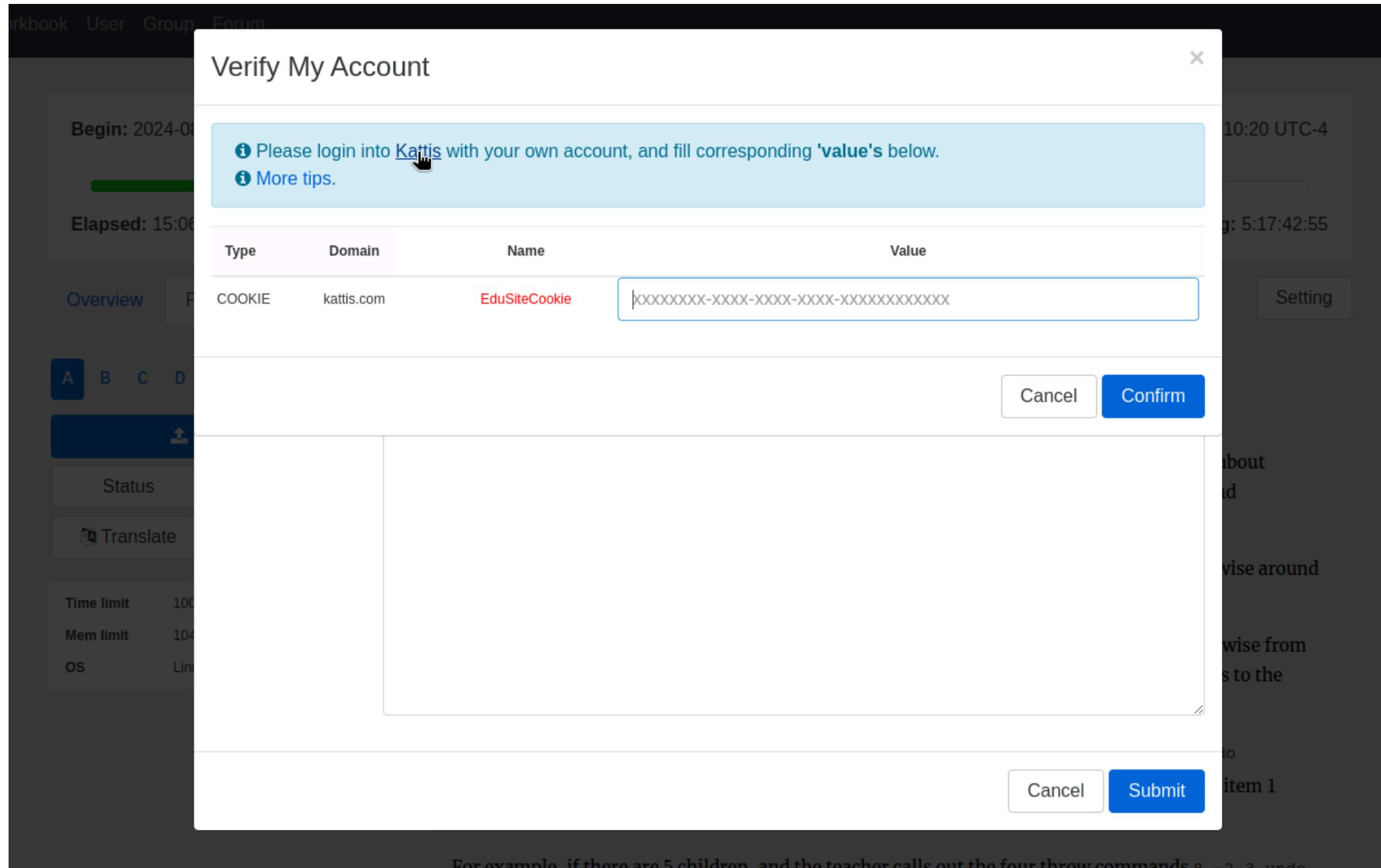
Mem limit 1048576 kB

OS Linux

Juez virtual



Juez virtual

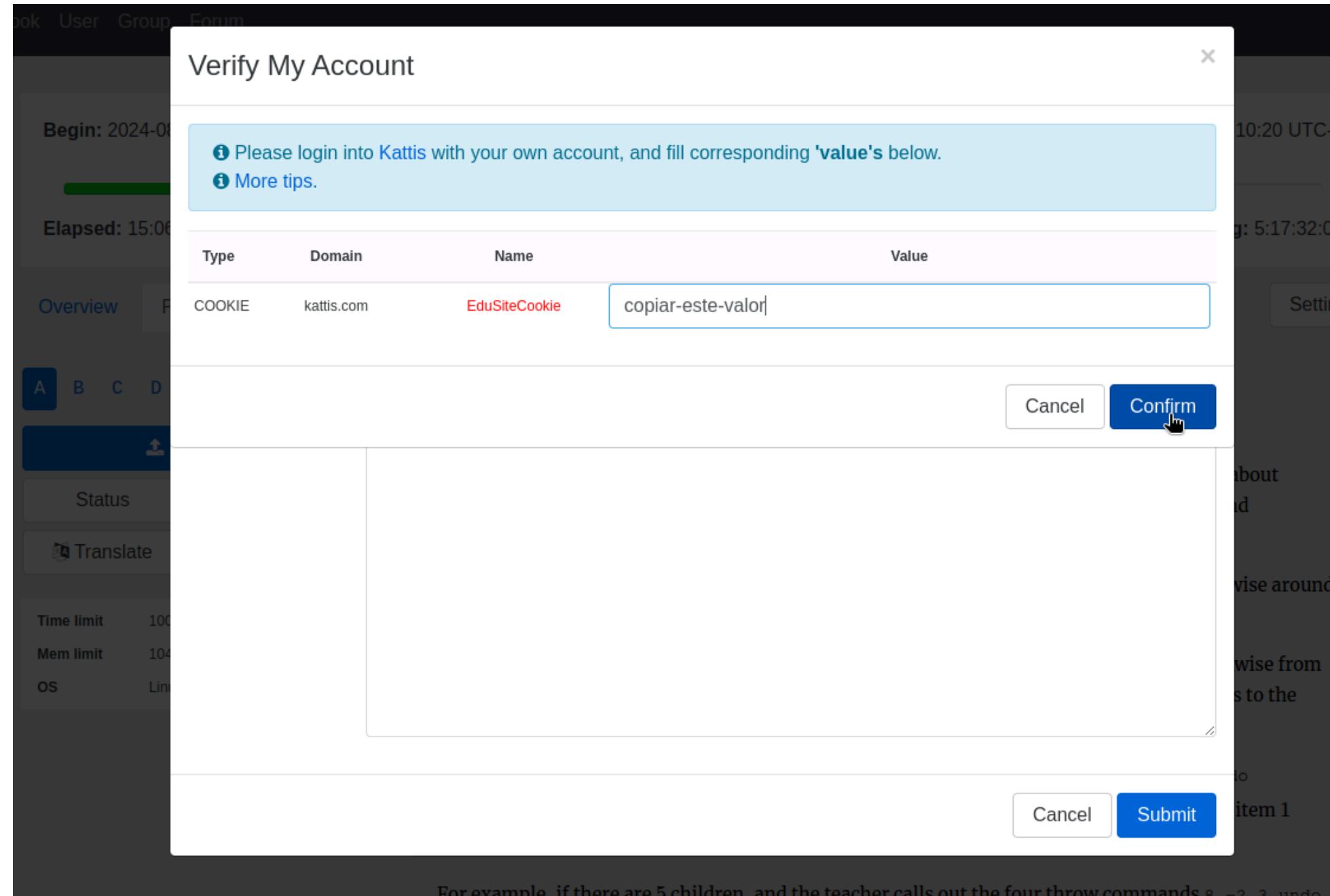


Juez virtual

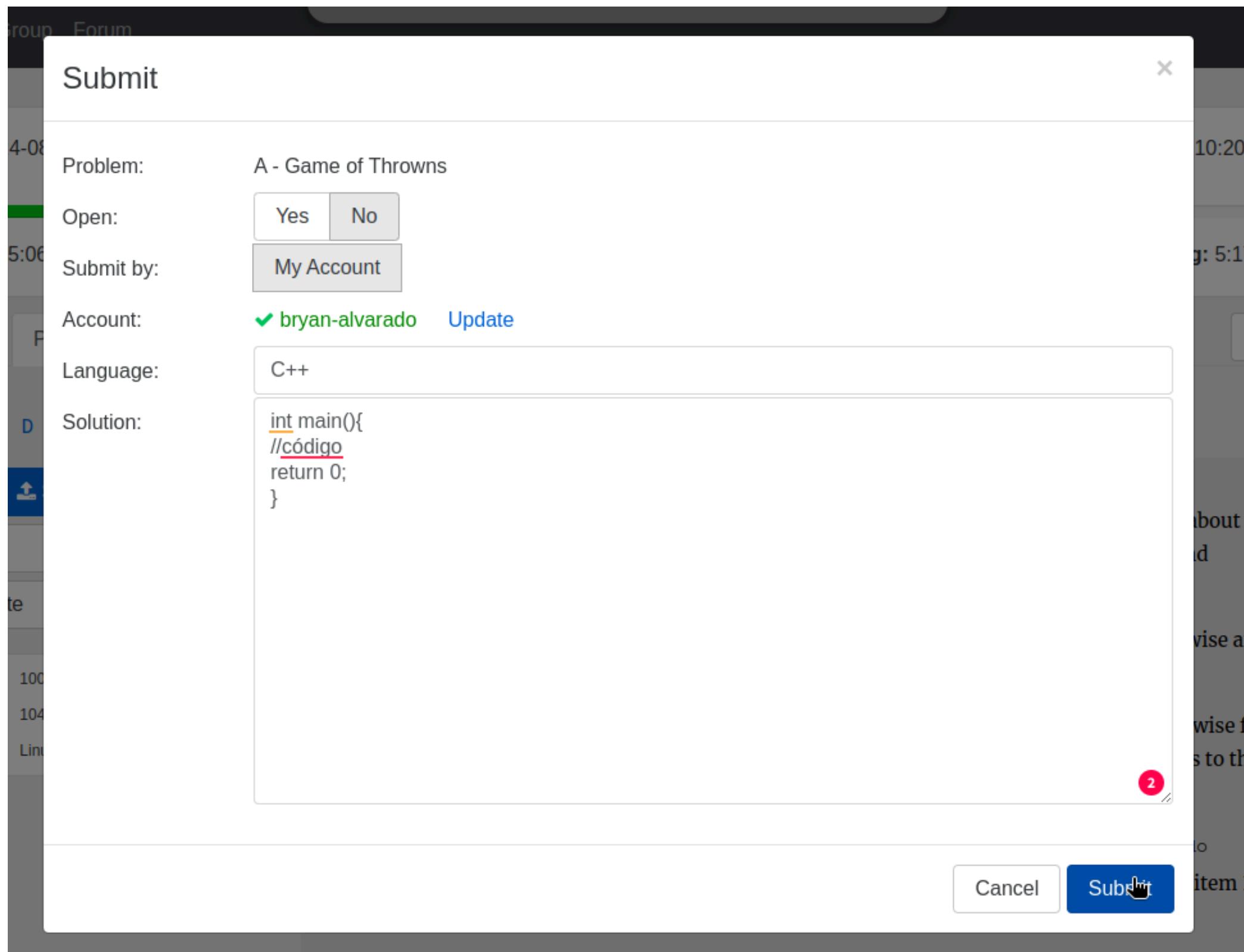
The screenshot shows a browser window with the Kattis user profile of Bryan Alvarado. The profile page displays basic information like rank (190910), score (1.0), and country (unspecified). Below the profile, there are tabs for Solved problems, Submissions, Profile settings, E-mail addresses, Login settings (which is currently selected), and Delete account. The right side of the screen features the Chrome developer tools Application panel, specifically the Cookies tab. This tab lists several cookies, including EduSiteCookie, _ga, _ga_3B9C221E6S, _gid, cf_clearance, and timezone. The timezone cookie is highlighted with a blue selection bar. The developer tools also show other storage areas like Local storage, Session storage, IndexedDB, and Web SQL.

Name	Value	D...	Path	Ex...	Size	Ht...	Se...	Sa...	Pa...	Pr...
EduSiteCookie	copiar-este-valor	.k...	/	Se...	30	✓	✓	N...	M...	M...
_ga	a copiar-este-valor	b	/	20...	4				M...	M...
_ga_3B9C221E6S			/	20...	15				M...	M...
_gid	c	.k...	/	20...	5				M...	M...
cf_clearance	d	.k...	/	20...	13	✓	✓	N...	ht...	M...
timezone	e open.	/	Se...	9						M...

Juez virtual



Juez virtual



Biblioteca estándar de Lenguaje

- Conjunto de herramientas, clases y funciones predefinidas que forman parte de un lenguaje de programación y que proporcionan funcionalidades comunes que los programadores pueden usar para desarrollar sus aplicaciones de manera más eficiente.
- Ejemplos:
 - C++: La Standard Template Library (STL) es una biblioteca estándar que incluye contenedores como vector, list, deque, map, set, y algoritmos para manipular estos contenedores.
 - Python: La biblioteca estándar de Python incluye módulos para realizar operaciones matemáticas (math), manipulación de cadenas (string), acceso a archivos (os, io).

Standar Template Libray

La Biblioteca Estándar de C++ (STL) ofrece una colección de estructuras de datos y algoritmos. La STL está diseñada para ofrecer un rendimiento eficiente y un uso intuitivo.

Documentación:

- cppreference.com
- cplusplus.com
- [time complexity](#)

Contenido:

- Contenedores: como vectores, listas, map, set, que se utilizan para almacenar y manipular datos
- Algoritmos: sort, find, se pueden utilizar para manipular datos almacenados en contenedores
- Iteradores: objetos que proveen formas de recorrer elementos en un contenedor

Estructura de contenedores:

- Sequences
- Container Adapters
- Associative containers

Sequences (contenedores base)

Vector

Contenedor secuencial que almacena elementos en un arreglo dinámico. Los elementos se almacenan de manera contigua en la memoria, lo que permite un acceso aleatorio eficiente.

Fortalezas

- Acceso rápido: Acceso aleatorio a elementos en tiempo constante $O(1)$.
- Redimensionamiento dinámico: Capacidad para redimensionarse automáticamente al agregar elementos, con amortiguación del costo de reubicación.

Debilidades

- Inserción/eliminación costosa: Operaciones de inserción o eliminación en el medio del vector pueden ser costosas en $O(n)$ debido a la necesidad de mover elementos.
- Redimensionamiento: Aunque es dinámico, el redimensionamiento puede implicar la reallocación de todos los elementos, lo que conlleva un costo adicional.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Agregar un elemento al final
    vec.push_back(6);

    // Acceso a elementos por índice
    std::cout << "Elementos en el vector: ";
    for (size_t i = 0; i < vec.size(); ++i) {
        std::cout << vec[i] << " "; // Output: 1 2 3 4 5 6
    }
    std::cout << std::endl;

    return 0;
}
```

List

Contenedor secuencial que implementa una lista doblemente enlazada. Los elementos no están almacenados de manera contigua en la memoria, sino que cada elemento apunta al siguiente y al anterior.

Fortalezas

- Inserción/eliminación eficiente: Operaciones de inserción y eliminación en cualquier posición se realizan en tiempo constante O(1).
- No invalidación de iteradores: Las operaciones de inserción/eliminación no invalidan los iteradores a otros elementos de la lista.

Debilidades

- Acceso aleatorio lento: No se puede acceder a un elemento en particular en tiempo constante, se requiere tiempo lineal O(n).
- Mayor uso de memoria: Dado que almacena punteros a los elementos anterior y siguiente, ocupa más memoria que un vector.

```
#include <iostream>
#include <list>

int main() {
    std::list<int> lst = {1, 2, 3, 4, 5};

    // Insertar un elemento en una posición específica
    auto it = lst.begin();
    std::advance(it, 2);
    lst.insert(it, 10);

    // Eliminar un elemento
    lst.erase(++it); // Eliminar el elemento después del nuevo 10

    // Iteración
    std::cout << "Elementos en la lista: ";
    for (int val : lst) {
        std::cout << val << " "; // Output: 1 2 10 4 5
    }
    std::cout << std::endl;

    return 0;
}
```

Dequeue

Contenedor secuencial que permite inserciones y eliminaciones eficientes en ambos extremos (inicio y final). Los elementos no están almacenados de forma contigua, sino en segmentos de memoria.

Fortalezas

- Inserción/eliminación en ambos extremos: Operaciones eficientes O(1) en ambos extremos de la cola.
- Acceso aleatorio: Permite acceso aleatorio a los elementos, aunque es menos eficiente que en un std::vector.

Debilidades

- Acceso aleatorio menos eficiente: El acceso aleatorio no es tan rápido como en std::vector debido a su estructura en segmentos.
- Mayor complejidad: Es un poco más complejo que un vector en términos de estructura interna y operaciones.

```
#include <iostream>
#include <deque>

int main() {
    std::deque<int> deq = {1, 2, 3, 4, 5};

    // Insertar elementos en el inicio y en el final
    deq.push_front(0);
    deq.push_back(6);

    // Iteración
    std::cout << "Elementos en el deque: ";
    for (int val : deq) {
        std::cout << val << " "; // Output: 0 1 2 3 4 5 6
    }
    std::cout << std::endl;

    return 0;
}
```

Forward List

Contenedor secuencial que implementa una lista enlazada simple. Cada elemento apunta solo al siguiente, y no hay punteros al elemento anterior.

Fortalezas

- Eficiencia en inserción/eliminación: Las operaciones de inserción y eliminación son eficientes en cualquier lugar de la lista.
- Uso de memoria reducido: Al tener punteros solo al siguiente elemento, usa menos memoria que una lista doblemente enlazada (std::list).

Debilidades

- Acceso aleatorio no soportado: No se puede acceder a un elemento en particular en tiempo constante ni moverse hacia atrás en la lista.
- Menos flexible que std::list: No soporta algunas operaciones como la eliminación en el medio sin recorrer la lista.

```
#include <iostream>
#include <forward_list>

int main() {
    std::forward_list<int> fwd_lst = {1, 2, 3, 4, 5};

    // Insertar un elemento después del primer elemento
    fwd_lst.insert_after(fwd_lst.begin(), 10);

    // Eliminar un elemento
    fwd_lst.erase_after(fwd_lst.begin()); // Eliminar el elemento después del 10

    // Iteración
    std::cout << "Elementos en la forward_list: ";
    for (int val : fwd_lst) {
        std::cout << val << " "; // Output: 1 10 3 4 5
    }
    std::cout << std::endl;

    return 0;
}
```

Array

Contenedor de tamaño fijo que encapsula un arreglo C. A diferencia de los arreglos C tradicionales, std::array ofrece una interfaz más amigable y es compatible con otras funcionalidades de la STL.

Fortalezas

- Tamaño fijo y contiguo: Los elementos se almacenan de manera contigua en memoria y el tamaño es fijo en tiempo de compilación.
- Eficiencia: Las operaciones de acceso son extremadamente rápidas O(1), y no hay sobrecarga de memoria adicional.

Debilidades

- Tamaño fijo: El tamaño no puede cambiar después de la creación, lo que limita su flexibilidad.
- Sin redimensionamiento: No tiene capacidad para crecer o reducir su tamaño, como std::vector.

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> arr = {1, 2, 3, 4, 5};

    // Acceso y modificación
    arr[2] = 10;

    // Iteración
    for (int val : arr) {
        std::cout << val << " "; // Salida: 1 2 10 4 5
    }

    return 0;
}
```

Container Adaptors

Queue

Contenedor adaptador que implementa una estructura de datos FIFO (First-In-First-Out). Los elementos se insertan al final y se eliminan del principio.

Fortalezas

- Simplicidad y eficiencia: Ideal para escenarios donde se requiere procesar elementos en el orden en que se agregan.
- Acceso a extremos: Permite acceso rápido a los extremos de la cola con operaciones eficientes para inserción (push()) y eliminación (pop()).

Debilidades

- Acceso limitado: No permite acceder a elementos intermedios, solo al frente y al final.
- No permite operaciones aleatorias: No se pueden realizar búsquedas ni modificaciones en elementos específicos.

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;

    // Insertar elementos
    q.push(1);
    q.push(2);
    q.push(3);

    // Eliminar y acceder a elementos en el orden FIFO
    std::cout << "Elementos en orden FIFO: ";
    while (!q.empty()) {
        std::cout << q.front() << " "; // Output: 1 2 3
        q.pop();
    }
    std::cout << std::endl;

    return 0;
}
```

Stack

Contenedor adaptador que implementa una estructura de datos LIFO (Last-In-First-Out). Los elementos se apilan y desapilan en un orden inverso al de su inserción.

Fortalezas

- Simplicidad y eficiencia: Ideal para escenarios donde se requiere procesar elementos en un orden inverso al de su inserción.
- Acceso rápido al tope: Permite acceso rápido al elemento en la parte superior de la pila con top().

Debilidades

- Acceso limitado: Solo se puede acceder al elemento en la parte superior y no a elementos intermedios.
- No permite iteración: No se pueden iterar sobre los elementos de la pila.

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> stk;

    // Insertar elementos
    stk.push(1);
    stk.push(2);
    stk.push(3);

    // Acceder y eliminar elementos en el orden LIFO
    std::cout << "Elementos en orden LIFO: ";
    while (!stk.empty()) {
        std::cout << stk.top() << " "; // Output: 3 2 1
        stk.pop();
    }
    std::cout << std::endl;

    return 0;
}
```

Priority Queue

Contenedor adaptador que implementa una cola de prioridad.

Los elementos se ordenan automáticamente en función de su prioridad, con el elemento con la mayor prioridad (por defecto el mayor valor) en la parte superior.

Fortalezas

- Acceso rápido al elemento de mayor prioridad: Permite acceder rápidamente al elemento con la mayor prioridad mediante el método `top()`.
- Operaciones eficientes: La inserción (`push()`) y la eliminación del elemento con mayor prioridad (`pop()`) son eficientes, con una complejidad de tiempo de $O(\log n)$.

Debilidades

- Acceso aleatorio no permitido: No se pueden acceder a elementos arbitrarios ni realizar iteraciones de manera directa.
- Solo un nivel de prioridad: Los elementos están ordenados de acuerdo con una sola prioridad, sin soporte para múltiples niveles de prioridad.

```
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq;

    // Insertar elementos
    pq.push(10);
    pq.push(5);
    pq.push(20);

    // Acceder y eliminar el elemento con la mayor prioridad
    std::cout << "Elementos en orden de prioridad (máximo primero): ";
    while (!pq.empty()) {
        std::cout << pq.top() << " ";
        pq.pop();
    }
    std::cout << std::endl;

    return 0;
}
```

Associative Containers

Set

Contenedor asociativo que almacena elementos únicos en un orden específico.

Fortalezas

- Elementos únicos: No permite elementos duplicados, garantizando que cada elemento es único.
- Ordenación automática: Los elementos se almacenan en orden automáticamente (por defecto, orden ascendente).

Debilidades

- Sin acceso aleatorio: No soporta acceso aleatorio a los elementos.
- Inserciones lentas comparado con contenedores secuenciales:

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s = {1, 2, 3, 4, 5};

    // Insertar un elemento
    s.insert(6);

    // Intentar insertar un elemento duplicado
    s.insert(3); // No se agrega porque ya existe

    // Iteración
    std::cout << "Elementos en el set (ordenados, sin duplicados): ";
    for (int val : s) {
        std::cout << val << " "; // Output: 1 2 3 4 5 6
    }
    std::cout << std::endl;

    return 0;
}
```

Map

Contenedor asociativo que almacena pares clave-valor, donde cada clave es única y se asocia a un único valor.

Fortalezas

- Elementos únicos: Cada clave es única, y los elementos se almacenan en orden basado en la clave.
- Ordenación automática: Los elementos se ordenan automáticamente por clave.

Debilidades

- Inserciones lentas comparado con contenedores secuenciales.

```
#include <iostream>
#include <map>

int main() {
    std::map<int, std::string> m;

    // Insertar pares clave-valor
    m[1] = "apple";
    m[2] = "banana";
    m[3] = "cherry";

    // Acceso a elementos por clave
    std::cout << "Elemento con clave 2: " << m[2] << std::endl; // Output: banana

    // Iteración
    std::cout << "Pares en el map (ordenados por clave):\n";
    for (const auto& [key, value] : m) {
        std::cout << key << ": " << value << "\n";
    }
    // Output:
    // 1: apple
    // 2: banana
    // 3: cherry

    return 0;
}
```

Multiset

Contenedor asociativo que almacena elementos en un orden específico y permite duplicados. Los elementos se mantienen ordenados según el criterio de comparación.

Fortalezas

- Almacenamiento ordenado: Los elementos se mantienen automáticamente en orden, lo que facilita la búsqueda y el rango de consultas.
- Permite duplicados: A diferencia de std::set, permite elementos duplicados.

Debilidades

- Rendimiento de búsqueda: Aunque las búsquedas son eficientes, no son tan rápidas como en std::unordered_multiset.
- Operaciones costosas: Las operaciones de inserción y eliminación tienen una complejidad de tiempo de O(logn).

```
#include <iostream>
#include <set>

int main() {
    std::multiset<int> ms = {1, 2, 2, 3, 4, 4, 4};

    // Insertar elementos adicionales
    ms.insert(3);
    ms.insert(5);

    // Iteración y acceso a elementos ordenados con duplicados
    std::cout << "Elementos en el multiset (ordenados, con duplicados): ";
    for (int val : ms) {
        std::cout << val << " "; // Output: 1 2 2 3 3 4 4 4 5
    }
    std::cout << std::endl;

    return 0;
}
```

Multimap

Contenedor asociativo que almacena pares clave-valor.

Permite claves duplicadas y mantiene los pares ordenados por clave.

Fortalezas

- Almacenamiento ordenado: Los pares se mantienen ordenados por clave, lo que facilita búsquedas de rango y ordenadas.
- Permite claves duplicadas: A diferencia de std::map, permite múltiples valores para la misma clave.

Debilidades

- Acceso más lento a elementos: Aunque las búsquedas son eficientes, no son tan rápidas como en std::unordered_multimap.
- Complejidad de operaciones: Las operaciones de inserción y búsqueda tienen una complejidad de tiempo de O(logn).

```
#include <iostream>
#include <map>

int main() {
    std::multimap<int, std::string> mmap;

    // Insertar pares clave-valor
    mmap.insert({1, "apple"});
    mmap.insert({1, "apricot"});
    mmap.insert({2, "banana"});
    mmap.insert({3, "cherry"});
    mmap.insert({3, "coconut});

    // Iteración y acceso a pares clave-valor ordenados con claves duplicadas
    std::cout << "Pares en el multimap (ordenados, con claves duplicadas):\n";
    for (const auto& [key, value] : mmap) {
        std::cout << key << ":" << value << "\n";
    }
    // Output:
    // 1: apple
    // 1: apricot
    // 2: banana
    // 3: cherry
    // 3: coconut

    return 0;
}
```

Unordered Associative Containers

Unordered Set

Contenedor asociativo que almacena elementos únicos en una tabla hash. A diferencia de std::set, no mantiene los elementos en orden.

Fortalezas

- Acceso rápido: Operaciones de búsqueda, inserción y eliminación en promedio en tiempo constante O(1) gracias a la tabla hash.
- Elementos únicos: Similar a std::set, no permite elementos duplicados

Debilidades

- Sin orden: No mantiene ningún orden entre los elementos.
- Complejidad en peor caso: En casos de colisiones graves en la tabla hash, las operaciones pueden degradarse a tiempo lineal O(n).

```
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_set<int> uset = {1, 2, 3, 4, 5};

    // Insertar elementos adicionales
    uset.insert(6);
    uset.insert(2); // Elemento duplicado, no se agrega

    // Iteración
    std::cout << "Elementos en el unordered_set (sin orden garantizado): ";
    for (int val : uset) {
        std::cout << val << " "; // Output puede variar: 1 2 3 4 5 6
    }
    std::cout << std::endl;

    return 0;
}
```

Unordered Map

Contenedor asociativo que almacena pares clave-valor en una tabla hash. A diferencia de std::map, no mantiene ningún orden entre los elementos.

Fortalezas

- Acceso rápido: Operaciones de búsqueda, inserción y eliminación en promedio en tiempo constante O(1).
- Menor sobrecarga de memoria: En comparación con std::map, al no necesitar almacenar la estructura del árbol.

Debilidades

- Sin orden: No hay orden entre los elementos.
- Complejidad en peor caso: Las operaciones pueden degradarse a tiempo lineal O(n) en caso de colisiones severas en la tabla hash.

```
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<int, std::string> um;

    // Insertar pares clave-valor
    um[1] = "apple";
    um[2] = "banana";
    um[3] = "cherry";

    // Acceso a elementos por clave
    std::cout << "Elemento con clave 2: " << um[2] << std::endl; // Output: banana

    // Iteración
    std::cout << "Pares en el unordered_map (sin orden garantizado):\n";
    for (const auto& [key, value] : um) {
        std::cout << key << ": " << value << "\n";
    }
    // Output puede variar:
    // 1: apple
    // 2: banana
    // 3: cherry

    return 0;
}
```

Unordered Multiset

Contenedor asociativo que almacena elementos en una tabla hash. Permite duplicados y no garantiza un orden específico de los elementos.

Fortalezas

- Acceso rápido: Ofrece una complejidad promedio de O(1) para operaciones de inserción, eliminación y búsqueda.
- Permite duplicados: A diferencia de std::unordered_set, permite múltiples ocurrencias del mismo valor.

Debilidades

- Sin orden garantizado: No mantiene un orden específico de los elementos, lo que puede dificultar la búsqueda de rangos ordenados.
- Uso de memoria: Puede utilizar más memoria debido a la tabla hash y la gestión de colisiones.

```
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_multiset<int> umset = {1, 2, 2, 3, 4, 4, 4};

    // Insertar elementos adicionales
    umset.insert(3);
    umset.insert(5);

    // Iteración y acceso a elementos con duplicados, sin orden garantizado
    std::cout << "Elementos en el unordered multiset (sin orden garantizado): ";
    for (int val : umset) {
        std::cout << val << " "; // Output puede variar: 1 2 2 3 3 4 4 4 5
    }
    std::cout << std::endl;

    return 0;
}
```

Unordered Multimap

Contenedor asociativo que almacena pares clave-valor en una tabla hash. Permite claves duplicadas y no garantiza un orden específico de los pares.

Fortalezas

- Acceso rápido: Ofrece una complejidad promedio de O(1) para operaciones de inserción, eliminación y búsqueda.
- Permite claves duplicadas: A diferencia de std::unordered_map, permite múltiples valores para la misma clave.

Debilidades

- Sin orden garantizado: No mantiene un orden específico de los pares, lo que puede dificultar la búsqueda de pares ordenados.
- Uso de memoria: Puede utilizar más memoria debido a la tabla hash y la gestión de colisiones.

```
#include <iostream>
#include <unordered_map>

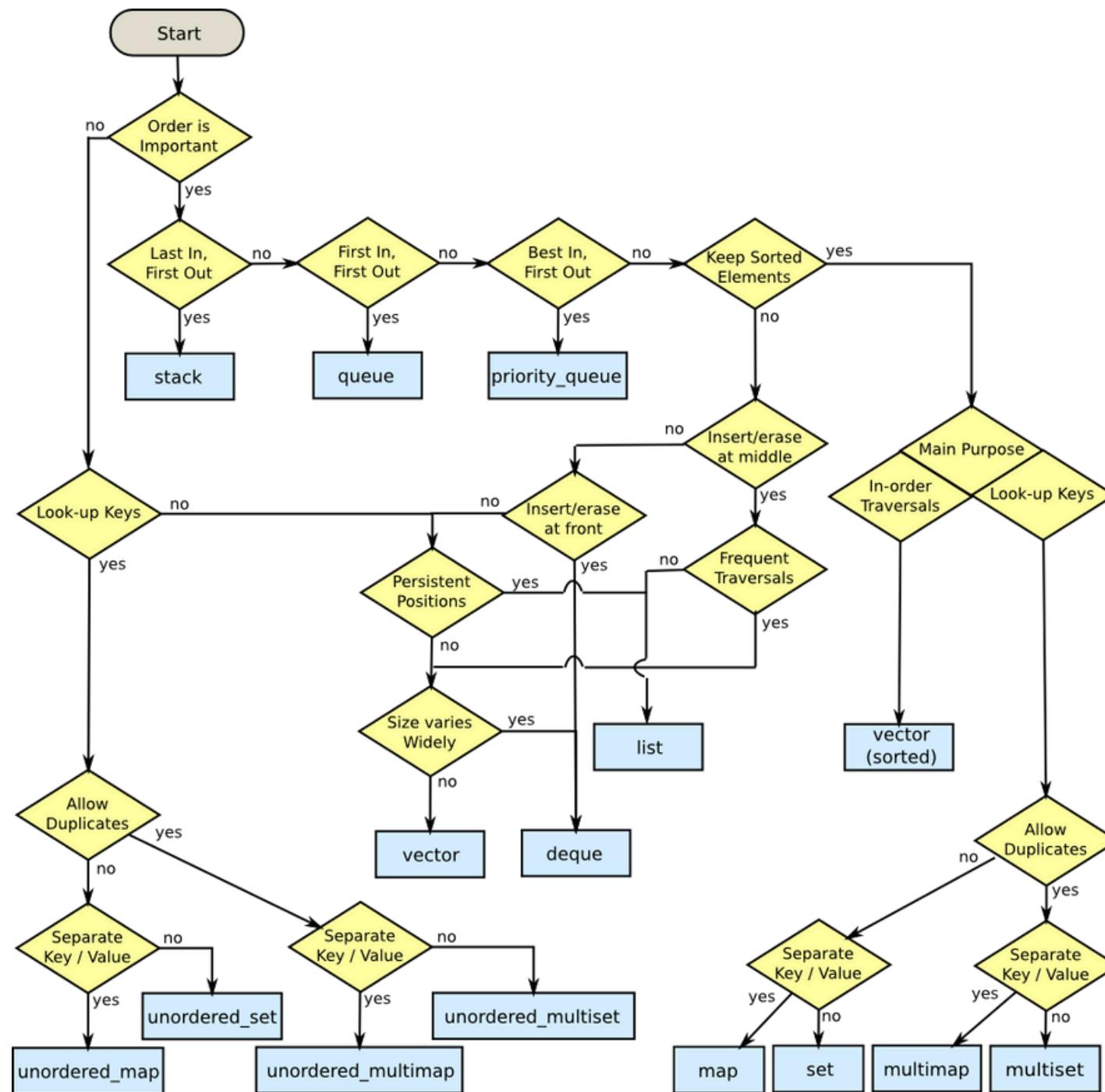
int main() {
    std::unordered_multimap<int, std::string> ummap;

    // Insertar pares clave-valor
    ummap.insert({1, "apple"});
    ummap.insert({1, "apricot"});
    ummap.insert({2, "banana"});
    ummap.insert({3, "cherry"});
    ummap.insert({3, "coconut"});

    // Iteración y acceso a pares clave-valor con claves duplicadas, sin orden garantizado
    std::cout << "Pares en el unordered multimap (sin orden garantizado):\n";
    for (const auto& [key, value] : ummap) {
        std::cout << key << ": " << value << "\n";
    }
    // Output puede variar:
    // 1: apple
    // 1: apricot
    // 2: banana
    // 3: cherry
    // 3: coconut

    return 0;
}
```

Resumen



Sobrecarga del operador <

La sobrecarga del operador < en C++ permite definir cómo se compara un objeto de una clase personalizada con otro objeto de la misma clase. Esto es particularmente útil cuando se utilizan contenedores STL que requieren un criterio de ordenación, como std::set, std::map, std::multiset, std::multimap, y std::priority_queue. La sobrecarga del operador < le dice al contenedor cómo ordenar los objetos que contiene.

Ejemplo:

Vamos a definir una clase Persona que almacena un nombre y una edad. Sobrecargaremos el operador < para que las personas se ordenen primero por edad y luego por nombre. Luego, utilizaremos esta clase en un std::set, que almacenará las personas en orden según el criterio definido.

```
#include <iostream>
#include <set>
#include <string>

// Definición de la clase Persona
class Persona {
public:
    std::string nombre;
    int edad;

    Persona(const std::string& n, int e) : nombre(n), edad(e) {}

    // Sobre carga del operador <
    bool operator<(const Persona& otra) const {
        if (edad == otra.edad) {
            return nombre < otra.nombre;
        }
        return edad < otra.edad;
    }
};

// Función principal
int main() {
    // Crear un set de personas
    std::set<Persona> personas;

    // Insertar elementos en el set
    personas.emplace("Ana", 30);
    personas.emplace("Luis", 25);
    personas.emplace("Carlos", 30);
    personas.emplace("Laura", 28);

    // Iterar sobre el set y mostrar los elementos
    std::cout << "Personas ordenadas por edad y nombre:\n";
    for (const auto& p : personas) {
        std::cout << p.nombre << ", Edad: " << p.edad << std::endl;
    }

    return 0;
}
```

FIN