Disjoint Intervals

Disjoint Intervals es una estructura de datos que representa una unión de intervalos cerrado-abiertos disjuntos de $\mathbb{R}$.

- Tiempo para insertar: $O(\log(n))$.
- Tiempo para buscar si existe: $O(\log(n))$.

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <set>
#include <vector>

// Closed-open interval [L,R)
template <class T>
struct Interval
{
    using value_type = T;
    Interval() : L(0), R(0) {}
    Interval(T l, T r) : L(l), R(r) {}
    T L;
    T R;
    T size() const { return R - L; }
};

template <class T>
bool operator<(const Interval<T>& A, const Interval<T>& B)
{
    if (A.L != B.L)
        return A.L < B.L;
    return A.R < B.R;
}

template <class T>
std::ostream& operator<<(std::ostream& os, const Interval<T>& I)
{
    os << "[" << I.L << ", " << I.R << ")";
    return os;
}

using namespace std;

template <class T>
class DisjointIntervals
```

```
{
public:
    using value_type = Interval<T>;
    using iterator = typename std::set<Interval<T>>::iterator;
    using const_iterator = typename std::set<Interval<T>>::const_iterator;

    static constexpr T INF = std::numeric_limits<T>::max();

    const_iterator Insert(T a, T b) { return Insert({a, b}); }

    const_iterator FirstThatContainsOrEndsAt(T x)
    {
        auto first = lower_bound({x, x});

        if (first == m_data.begin())
            return first;

        // guaranteed to exist, since first != m_data.begin()
        auto prev = std::prev(first);

        if (prev->R >= x)
            return prev;

        return first;
    }

    const_iterator Insert(const Interval<T>& I)
    {
        auto L = I.L;
        auto R = I.R;

        //   L-----R
        // ---         <- This is the first that
        // could intersect (if it exists)
        auto first_possible = FirstThatContainsOrEndsAt(L);

        if (first_possible == m_data.end() || first_possible->L > R)
            return m_data.insert(I).first;

        L = std::min(L, first_possible->L);

        //   L-----R
        //           ---     <- First whose left
        //           is strictly > R
        auto last_possible = upper_bound({R, INF});
```

```
        // guaranteed to exist, since first_possible != m_data.end()
        auto last_intersected = std::prev(last_possible);
        R = std::max(R, last_intersected->R);

        // Erase the whole range that intersects [L,R)
        m_data.erase(first_possible, last_possible);

        return m_data.insert({L, R}).first;
    }

    const_iterator lower_bound(const Interval<T>& I) const
    {
        return m_data.lower_bound(I);
    }

    const_iterator upper_bound(const Interval<T>& I) const
    {
        return m_data.upper_bound(I);
    }

    const auto& Intervals() const { return m_data; }

private:
    std::set<Interval<T>> m_data;
};

template <class T>
std::ostream& operator<<(std::ostream& os, const DisjointIntervals<T>& D)
{
    auto& I = D.Intervals();

    auto it = I.begin();
    if (it == I.end())
    {
        os << "empty";
        return os;
    }

    os << *it;
    ++it;

    for (; it != I.end(); ++it)
    {
        os << " U " << *it;
```

```
    }
    return os;
}

using namespace std;
// Example program
int main()
{
    DisjointIntervals<int> D;
    D.Insert(0, 4);
    cout << D << endl;
    D.Insert(2, 8); // Intersects on the right
    cout << D << endl;
    D.Insert(-2, 1); // Intersects on the left
    cout << D << endl;
    D.Insert(-3, 9); // Contains
    cout << D << endl;

    D.Insert(15, 24); // Doesn't intersect at all
    cout << D << endl;

    D.Insert(10, 12); // In between, no intersect
    cout << D << endl;

    D.Insert(12, 15); // Joins two existing ones.
    cout << D << endl;

    return 0;
}
```

Output:

```
[0, 4)
[0, 8)
[-2, 8)
[-3, 9)
[-3, 9) U [15, 24)
[-3, 9) U [10, 12) U [15, 24)
[-3, 9) U [10, 24)
```