Trabajo Práctico 2: Árboles de Intervalos

1. Motivación del problema

Un *intervalo cerrado* es un par ordenado de números reales de la forma $[a, b]$ con $a \leq b$. Dados dos intervalos $[a, b]$ y $[c, d]$, se dice que ellos se intersectan si el conjunto

$$\{x \in \mathbb{R} : a \leq x \leq b \wedge c \leq x \leq d\}$$

es no vacío.

El objetivo de este trabajo es brindar una implementación para conjuntos dinámicos de intervalos cerrados, es decir, conjuntos donde sea posible insertar y eliminar intervalos de manera dinámica, que cuente con una función `intersectar([a,b])` para consultar de manera eficiente si algún intervalo del conjunto se intersecta con el intervalo $[a, b]$.

Por ejemplo, en el siguiente conjunto

$$\{[-1.4, 2.0], [10.0, 11.1], [3.4, 12.0], [-2.7, 2.7]\} \tag{1}$$

al consultar por el intervalo $[0.0, 1.0]$ la función respondería afirmativamente, dado que tiene intersección con $[-1.4, 2.0]$, y en cambio, al consultar por el intervalo $[3.0, 3.3]$, respondería negativamente.

Una posible implementación podría ser mediante listas enlazadas, pues permiten insertar y eliminar elementos de manera dinámica. Respecto a `intersectar([a,b])`, su implementación comenzaría por intersectar $[a, b]$ con el primer elemento de la lista y, en caso de no haber intersección, pasaría a analizar el siguiente elemento, y así sucesivamente. Es decir, en el peor caso se deben recorrer todos los elementos del conjunto.

Otra alternativa consiste en usar listas enlazadas ordenadas, utilizando el orden dado por el extremo izquierdo del intervalo, es decir, $[a, b]$ se posiciona antes que $[c, d]$ si $a \leq c$. En el conjunto 1 los elementos se ubicarían en el siguiente orden

$$[-2.7, 2.7], [-1.4, 2.0], [3.4, 12.0], [10.0, 11.1]$$

En este caso, `intersectar([a,b])` puede aprovechar el orden de la lista para dejar de recorrerla cuando llega a un elemento $[x, y]$ con $b < x$, dado que en ese caso se tiene la certeza de que él y todos los elemento posteriores no tienen intersección con $[a, b]$. A pesar de esta aparente mejora, en el peor caso aún es necesario recorrer todos los elementos del conjunto, por ejemplo al consultar en el conjunto 1 por $[13.0, 14.0]$.

Mediante *árboles de intervalos* es posible implementar `intersectar([a,b])` de manera más eficiente (más precisamente con complejidad $O(\ln n)$, donde n es la cantidad de elementos del conjunto). A continuación se presenta su definición.

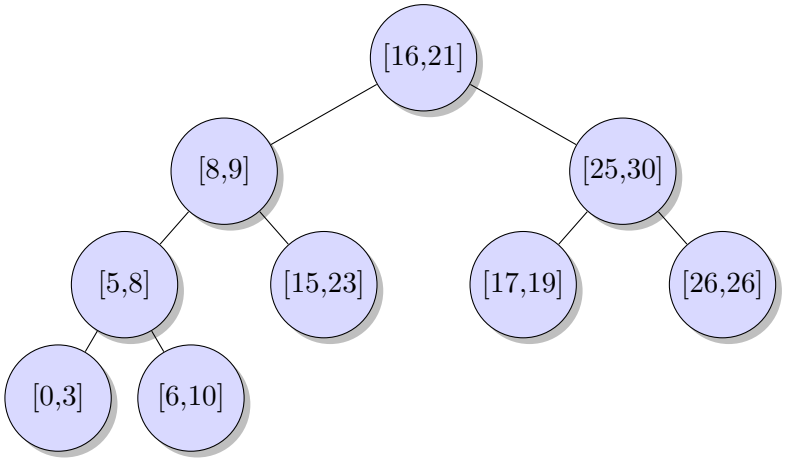

Figura 1: Ejemplo de árbol de intervalos

2. Árboles de intervalos

Un *árbol de intervalos* es una estructura de árbol ordenada donde los nodos almacenan intervalos. Por ejemplo, el árbol de la Figura 1 es un árbol de intervalos donde los nodos están ordenados según el siguiente criterio: en todo intervalo $[a, b]$ de un nodo interno, los intervalos del subárbol izquierdo tienen su extremo izquierdo menor o igual que a, y los intervalos del subárbol derecho tienen su extremo izquierdo mayor que a. Así por ejemplo, $[8, 9]$ y $[15, 23]$ se ubican el el subárbol izquierdo del nodo raíz pues $8 \leq 16$ y $15 \leq 16$; y en por el contrario $[25, 30]$ y $[26, 26]$ se ubica en el subárbol derecho del nodo raíz pues $25 > 16$ y $26 > 16$.

La estructura que poseen los árboles de intervalos permite que la implementación de `intersectar([a,b])` evite en algunos casos recorrer determinados subárboles. Por ejemplo, para determinar si el intervalo $[14, 15]$ tiene intersección con alguno de los intervalos del árbol de la Figura 1, un algoritmo comenzaría recorriendo el árbol por la raíz. Dado que $[14, 15]$ y $[16, 21]$ no tienen intersección, el algoritmo debe continuar recorriendo sus hijos. Sin embargo, por la estructura del árbol de intervalos, todos los intervalos del subárbol derecho tienen su extremo izquierdo mayor a 16, por lo cual ninguno de ellos se intersecta con $[14, 15]$ y este subárbol puede ser desestimado.
No obstante, la estructura presentada no siempre permite desestimar uno de los subárboles. Por ejemplo, para determinar si el intervalo $[22, 24]$ tiene intersección con alguno de los intervalos del árbol de la Figura 1, el algoritmo comenzaría nuevamente recorriendo el árbol por la raíz. Dado que $[22, 24]$ y $[16, 21]$ no tienen intersección, el algoritmo debe continuar recorriendo sus hijos. En este caso, no existe a priori ninguna garantía de que uno de los subárboles tenga o no inteserción con $[22, 24]$, por lo que deben recorrerse ambos.

Para solucionar este inconveniente, la estructura de árboles de intervalos se aumenta, agregando a cada nodo un valor extra que representa el máximo extremo derecho entre los intervalos contenidos en ese subárbol. Mediante esta nueva estructura, el árbol de la Figura 1 tendría

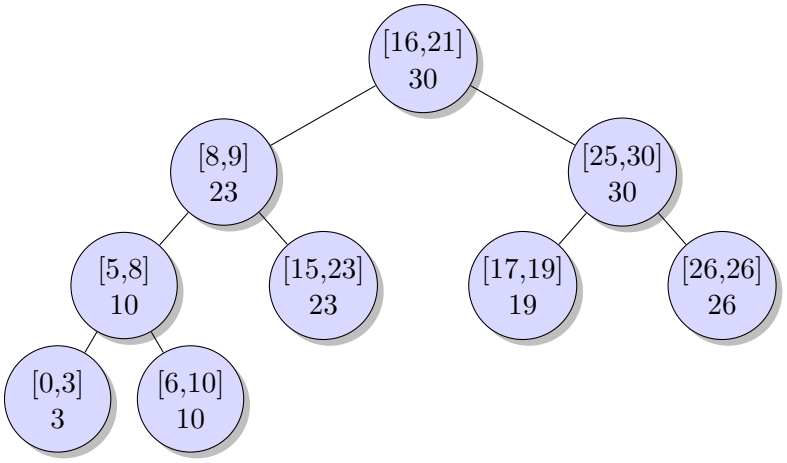

Figura 2: Ejemplo de árbol de intervalos aumentado

en su raíz el intervalo $[16, 21]$ y el valor 30 (alcanzado por su descendiente $[25, 30]$), su hijo izquierdo tendría el intervalo $[8, 9]$ y el valor 23 (alcanzado por su descendiente $[15, 23]$), mientras que su hijo derecho tendría el intervalo $[25, 30]$ y el valor 30 (alcanzado por él mismo). El resultado final se muestra en la Figura 2.

Mediante esta nueva estructura, para determinar si el intervalo $[22, 24]$ tiene intersección con alguno de los intervalos del árbol de la Figura 2, un algoritmo comenzaría recorriendo el árbol por la raíz, siguiendo los siguientes pasos:

(1) Intersectar $[22, 24]$ y $[16, 21]$. Como no tienen intersección, se deben seguir recorriendo sus hijos. Dado que el hijo izquierdo tiene como descendiente algún intervalo con extremo derecho igual a 23 (información aportada por el valor extra que se agregó en el nodo) y dicho intervalo tiene su extremo izquierdo menor o igual a 16 (debido a que se ubica a la izquierda de la raíz), se tiene la certeza de que en el subárbol izquierdo hay al menos un intervalo que intersecte a $[22, 24]$, por lo que se puede responder afirmativamente sin necesidad de recorrer el subárbol derecho. En caso de querer encontrar explícitamente el intervalo, el algoritmo continuará recorriendo el hijo izquierdo.

(2) Intersectar $[22, 24]$ y $[8, 9]$. Como no tienen intersección, se deben seguir recorriendo sus hijos. Dado que el hijo izquierdo no contiene ningún intervalo con extremo derecho mayor a 10, se tiene la certeza de que no hay interección con $[22, 24]$ en el subárbol izquierdo, entonces el algoritmo continuará recorriendo el subárbol derecho.

(3) Intersectar $[22, 24]$ y $[15, 23]$. Como tienen intersección retornar $[15, 23]$.

De esta forma, la estructura aumentada de árbol de intervalos garantiza que toda consulta a `intersectar([a,b])` requerirá recorrer a lo sumo tantos elementos del conjunto como altura tenga el árbol, que en caso de estar balanceando dependerá del logaritmo del tamaño del conjunto.

3. Consigna

a) Proveer una implementación para árboles de intervalos. Esta implementación **debe** utilizar árboles AVL, y los extremos de los intervalos deben ser `double`. La interfaz deberá contar con las siguientes funciones:

- `itree_crear`: crea un árbol de intervalos vacío.
- `itree_destruir`: destruye un árbol de intervalos.
- `itree_insertar`: inserta un intervalo en un árbol de intervalos.
- `itree_eliminar`: elimina un intervalo de un árbol de intervalos.
- `itree_intersectar`: determina si un intervalo se intersecta con alguno de los intervalos del árbol y, en caso afirmativo, retorna un apuntador al nodo correspondiente.
- `itree_recorrer_dfs`: recorrido primero en profundidad del árbol de intervalos.
- `itree_recorrer_bfs`: recorrido primero a lo ancho del árbol de intervalos.

Es requisito que la implementación respete lo consignado en la Sección 2.

b) Implementar un intérprete para manipular árboles de intervalos desde la entrada estándar. El interprete deberá soportar los siguientes comandos:

- `i [a,b]`: inserta el intervalo $[a, b]$ en el árbol.
- `e [a,b]`: elimina el intervalo $[a, b]$ del árbol.
- `? [a,b]`: intersecta el intervalo $[a, b]$ con los intervalos del árbol.
- `dfs`: imprime los intervalos del árbol con recorrido primero en profundidad.
- `bfs`: imprime los intervalos del árbol con recorrido primero a lo ancho.
- `salir`: destruye el árbol y termina el programa.

Ejemplo de uso:

```
$ ./interprete
i [-2.5, 0.02]
i [1.5, 1.5]
dfs
  [-2.5, 0.02] [1.5,1.5]
e [-2.5, 0.02]
dfs
  [1.5,1.5]
? [0,3.5]
  Si
? [-1.01, 0.02]
  No
salir
```

c) Cumplir con las **convenciones de código y de proyecto** elaboradas por la cátedra en el sitio de Comunidades.

d) Elaborar un **informe** que incluya:

- dificultades encontradas y cómo las resolvieron.
- bibliografía utilizada, en caso de haberla.
- cualquier otra información que considere relevante para la comprensión del trabajo.