12 | Árbol binario ordenado

Meta

Que el alumno domine el manejo de información almacenada en un *árbol binario ordenado*.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- Visualizar el uso correcto de referencias para implementar estructuras tipo árbol.
- Manejar con familiaridad el almacenamiento de datos en una estructura utilizando referencias.
- Implementar con mayor habilidad métodos recursivos y/o iterativos para manipular estructuras de datos con referencias.

Antecedentes

Definición 12.1: Árbol

Un *árbol* es una colección de elementos llamados *nodos*, uno de los cuales se distingue como *raíz*, junto con una relación de «paternidad» que impone una estructura jerárquica sobre los nodos Vargas 1998.

Formalmente:

1. Un solo nodo es, por sí mismo, un árbol. Ese nodo es también la raíz de dicho árbol.

2. Supóngase que n es un nodo y que $A_1, A_2, ..., A_k$ son árboles con raíces $n_1, n_2, ..., n_k$, respectivamente. Se puede construir un árbol nuevo haciendo que n se constituya en el padre de los nodos $n_1, n_2, ..., n_k$. En dicho árbol, n es la raíz y $A_1, A_2, ..., A_k$ son los *subárboles* de la raíz. Los nodos $n_1, n_2, ..., n_k$ reciben el nombre de *hijos* del nodo n.

Definición 12.2: Árbol binario

Un *árbol binario* se puede definir de manera recursiva:

1. Un *árbol binario* es un árbol vacío.
2. Un nodo que tiene un elemento y dos *árboles binarios*: izquierdo y derecho.

Definición 12.3: Camino

Dado un árbol A que contiene al conjunto de nodos N, un *camino* en A se define como una secuencia de nodos distinta del vacío:

$$C = \{n_1, n_2, ..., n_k\}, \tag{12.1}$$

donde $n_i \in N$, para $1 \leqslant i \leqslant k$ tal que el i-ésimo nodo en la secuencia n_i, es el padre del $< i+1 >$-ésimo nodo en la secuencia, n_{i+1}.

La *longitud* del camino C es $k-1$ (i.e. el número de aristas recorridas).

Obsérvese que existe un único camino entre la raíz de un árbol y cualquiera de sus nodos [Figura 12.1]. Por otro lado, el *altura* de un nodo $n_i \in N$ es la longitud del camino más largo del nodo n_i a una hoja.

Recorridos

Los árboles binarios pueden ser recorridos en:

Amplitud Los nodos son visitados nivel por nivel, comenzando por la raíz. Se utiliza una cola como estructura auxiliar para formar a los nodos que serán visitados.

Profundidad Son fáciles de implementar utilizando una pila como estructura auxiliar, la implementación iterativa es más eficiente, pues utiliza memoria constante.

Preorden Se visita el dato en la raíz, luego el subárbol izquierdo y luego el derecho, recursivamente.

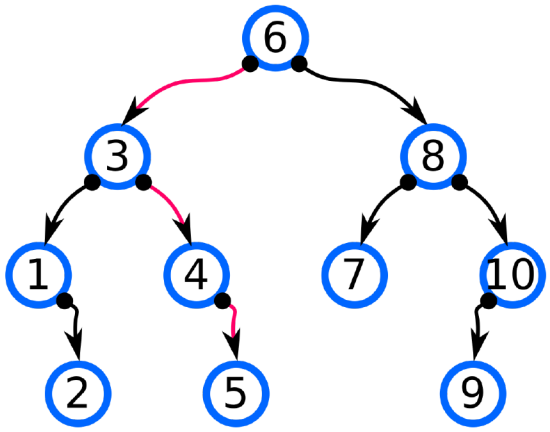

Figura 12.1 El camino de la raíz a un nodo es único. Su longitud es el número de aristas que contiene.

Inorden Se visita el dato en el subárbol izquierdo, luego la raíz y luego el derecho, recursivamente.

Preorden Se visita el dato en el subárbol izquierdo, luego el derecho recursivamente y finalmente, la raíz.

Árbol Binario Ordenado

Definición 12.4: Árbol binario ordenado

Un *árbol binario ordenado* contiene elementos de un tipo C tal que todos ellos son comparables mediante una relación de orden. En un árbol ordenado cada nodo cumple con la propiedad siguiente:

1. Todo dato almacenado a la *izquierda* de la raíz es *menor* que el dato en la raíz.
2. Todo dato almacenado a la *derecha* de la raíz es *mayor o igual* que el dato en la raíz.

Gracias a la relación de orden establecida en los árboles binarios ordenados, es posible definir algoritmos de inserción y remoción deterministas, que indican precisamente dónde colocar el dato y, por lo mismo, recuperarlo eficientemente.

Búsqueda

Para encontrar un dato en el árbol se comienza preguntando desde la raíz:

1. Si el dato es igual al nodo actual, se devuelve este nodo.
2. Si el dato es menor, se procede a preguntar en el subárbol izquierdo.
3. Si el dato es mayor, se pregunta en el subárbol derecho.
4. Si ya no hay subárboles dónde preguntar, se sabe que el dato no está en el árbol.

Inserción

Básicamente, se utiliza el álgoritmo de búsqueda, hasta encontrar un nodo sin el subárbol donde debiera continuar buscando al dato que se desea insertar, entonces se crea un nodo nuevo con el dato y se convierte en el subárbol correspondiente.

Remoción

Sólo se remueven los nodos hoja. Si se desea remover un dato en un nodo interno, se realiza el procedimento siguiente desde el nodo donde se encuentra el dato:

- Si el nodo no tiene hijos, se remueve.
- Si tiene hijo izquierdo, se intercambia el dato por el del nodo con el dato más grande del subárbol izquierdo y se continúa el algoritmo desde ese nodo.
- Si tiene hijo derecho, se intercambia el dato por el del nodo con el dato más chico del subárbol derecho y se continúa el algoritmo desde ese nodo.

Implementación

La forma más natural de implementar árboles binarios es mediante referencias y una estructura tipo *Nodo*, semejante a la utilizada en la lista doblemente ligada, pero con referencias a los elementos siguientes:

- El dato almacenado en el nodo.
- Una referencia a su nodo padre, en caso de tenerlo.
- Una referencia a su hijo izquierdo, en caso de tenerlo.
- Una referencia a su hijo derecho, en caso de tenerlo.

Representaremos al árbol vacío con `null`.

Al igual que antes, contaremos con una clase de tipo *Árbol* cuyo principal atributo será la referencia al nodo *raíz*. La Figura 12.2 muestra esta estructura.

En el caso del árbol ordenado, dado que tenemos un criterio y algorimos deterministas para la inserción, recuperación y remoción de elementos, nos será posible,

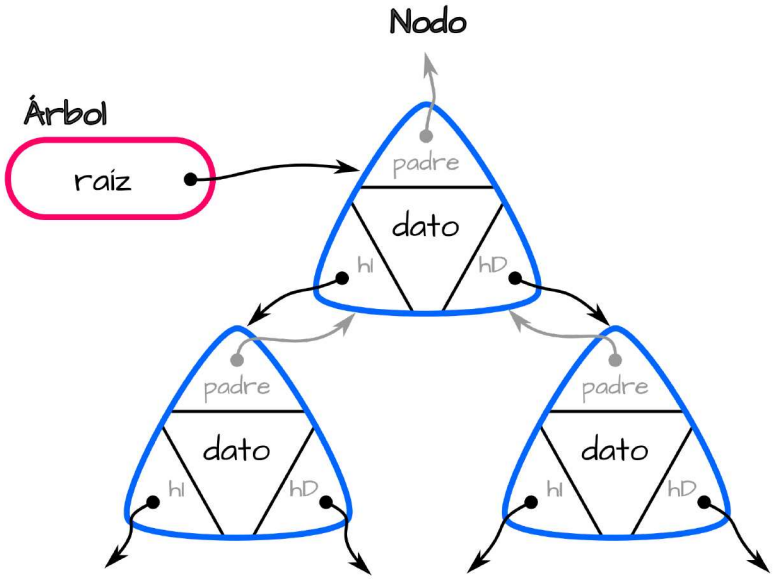

Figura 12.2 Estructura de datos árbol binario con nodos y referencias.

una vez más, implementar la interfaz `Collection<E>` y reutilizar código de la clase `ColecciónAbstracta<E>`. Para ello deberemos añadir un requisito: en esta ocasión, el tipo de datos almacenable en la estructura debe implementar la interfaz `Comparable<T>`. A continuación se exponen los pasos a seguir.

Desarrollo

En esta práctica implementarás no sólo un árbol binario ordenado, sino uno que servirá como clase base para los árboles balanceados de las prácticas siguientes. Por ello algunas de las decisiones de diseño que se incorporarán en esta práctica podrían parecer extrañas, pero sus beneficios saldrán a la luz más adelante.

Como un auxiliar para esta práctica, se provee una interfaz gráfica que dibuja los árboles si implementan correctamente las interfaces en el código adjunto. Esta interfaz es un paquete que funciona de forma muy semejante a `junit`, por lo que, si deseas agregar pruebas nuevas para tus árboles puedes hacerlo. Observa los ejemplos en el archivo `ed/visualización/demos/DemoÁrbolesBinariosOrdenados.java`.

Para ver los árboles de manera gráfica, se provee un paquete que los dibuja en pantalla, con la condición de que se encuentren en un estado consistente. El código siguiente muestra el uso del decorador `@DemoMethod` para graficar los árboles.

Para implementar este **árbol binario ordenado** se deben programar las clases:

- `NodoBOLigado<C>`

- `ÁrbolBOLigado<C>`

1. Comienza por programar la clase `NodoBOLigado<C>` y asegúrate de que compile bien.

   Esta clase debe implementar la interfaz `NodoBinarioOrdenado<C>` y sus elementos son del tipo genérico `<C extends Comparable<C> >`.

   - Los atributos de esta clase serán de acceso protegido, pues serán reutilizados más adelante, así que no olvides documentarlos.
   - Programa los métodos de lectura y escritura para el dato almacenado y los nodos padre, hijo izquierdo e hijo derecho, con los nombres indicados por la interfaz `NodoBinario<E>`.

     TIP: Haz que todos los métodos que devuelven un `NodoBinario<C>` tengan como tipo de regreso `NodoBOLigado<C>`. Esto es válido porque `NodoBOLigado<C>` implementa la interfaz y te ahorrará muchas conversiones de tipo (*castings*). Desafortundamente no podrás hacer lo mismo con los argumentos de los métodos porque alterarías su firma, por lo que tendrás que revisar a mano que los argumentos sean de la clase correcta. Para ello puedes utilizar `instaceof` o el atributo `.class` de los objetos.
   - Crea los constructores siguientes:
     - ⋆ `NodoBOLigado(C dato)`
     - ⋆ `NodoBOLigado(C dato, NodoBOLigado<C> padre, NodoBOLigado<C> hijoI, NodoBOLigado<C> hijoD)`

     Pero ojo, no llamarás ninguno de estos constructores en los métodos directamente, los vas a llamar sólo dentro de un método que se llamará `creaNodo` y estará en la clase árbol.
   - Programa los otros métodos definidos en las interfaces `NodoBinario<E>` y `NodoBinarioOrdenado<C>` siguiendo las indicaciones en la documentación.

     TIP: para el método `altura()` agrega de una vez un atributo `altura` que contenga el altura de cada nodo, esto permitirá responder en $\mathcal{O}(n)$. Cada vez que insertes o remuevas nodos del árbol deberás actualizar el valor de este atributo para todos los nodos a lo largo del camino de inserción/remoción, lo cual no altera el orden de complejidad de estos métodos. Si bien es posible programar el método `altura()` utilizando la definición recursiva de altura de un nodo, la complejidad por cada llamada es $\mathcal{O}(n)$.

2. Continúa con la clase `ÁrbolBOLigado<C>`. Esta clase debe implementar la interfaz

   `ÁrbolBinarioOrdenado<C>`

contendrá todo el código aplicable a cualquier árbol binario ordenado. No tiene que estar balancedo. Obsérvese que `ÁrbolBinarioOrdenado< C extends Comparable <C> >` extiende `ÁrbolBinario<E>`, que a su vez extiende `Collection<E>`. Por lo tanto tu clase `ÁrbolBOLigado<C>` debe implementar todos los métodos definidos por las tres interfaces. Algunos métodos ya se encuentran implementados en la clase `ColecciónAbstracta<E>` por lo que `ÁrbolBOLigado<C>` la extenderá.

TIP: Programa todos los métodos de las clases `NodoBOLigado` y `ÁrbolBOLigado`, pero de forma que sólo devuelvan `null`, `false` y `-1` según su tipo de regreso. Asegúrate de poder compilar con `ant` y `make`. Cuando todo compile podrás ejecutar `make run` y ver la interfaz gráfica de apoyo. El único caso que funcionará en este punto es el árbol vacío, a partir de aquí podrás ir implementando los demás métodos hasta que funcionen todos los casos de prueba.

3. A la clase `ÁrbolBOLigado<C>` agrégale los métodos:

```
protected NodoBOLigado<C> creaNodo(C dato) { ... }
protected NodoBOLigado<C> creaNodo(C dato,
                                   NodoBinario<C> padre,
                                   NodoBinario<C> hijoI,
                                   NodoBinario<C> hijoD) {
  ...
}
```

   Estos métodos crearán a los nodos del tipo correspondiente y en futuras prácticas serán sobreescritos por clases herederas de ésta. Recuerda crear a todos tus nodos con estos métodos en lugar de con sus constructores. Necesitarás usar *castings*.

4. Antes de programar al método `add`, crea un método auxiliar que hará el grueso del trabajo. Este método es:

```
protected NodoBOLigado<C> addNode(C e) {
  ...
}
```

   Este método debe devolver al nodo recién agregado. Los detalles los puedes implementar en el árbol, si tu implementación es iterativa (que es más eficiente), o en el nodo si optas por la versión recursiva. Úsalo después para implementar las otras versiones de agregar que se te solicitan.

5. Crea otro método auxiliar que te permita encontrar al primer nodo que contenga al dato pasado como parámetro y lo devuelva.

6. Para remover nodos también debes usar un método auxiliar. Aquí utilizaremos un truco curioso para el futuro: este método debe devolver el nodo que se quite del árbol. El último padre de este nodo debe eliminar su referencia hacia él, pero este

nodo debe quedarse con la referencia a quien fue su padre. De cualquier modo cuando dejemos de usar este nodo esa referencia desaparecerá, pero lo necesitaremos un poco más para balancear árboles en clases futuras.

Los métodos que debes implementar en `ÁrbolBinario<E>` son:

- `public boolean add(C e)` El método de inserción debe tener una complejidad promedio de $\mathcal{O}(\log n)$.
- `public Iterator<C> iterator()` Deberás devolver un iterador inorden.
- `public void clear()` Puedes dejarle la mayor parte de la tarea al recolector de basura.

Indicaciones adicionales:

1. Observa también que `ÁrbolBinario<E>` solicita un método que devuelve el nodo raíz, éste fue necesario para que el paquete de dibujo pudiera realizar su tarea en forma eficiente. El paquete de dibujo necesita acceso a la estructura del árbol pues eso es lo que va a dibujar. Es un caso de uso distinto al de un programador que sólo quiere al árbol para que almacene sus datos en orden y se los devuelva eficientemente.
2. Los métodos `contains`, `remove` y `add` deben cumplir con la complejidad de O(log(n)).
3. Para el método `iterator` utiliza el recorrido inorden, no implementes ni `add` ni `remove`.

Preguntas

1. Si se añaden los números del 1 al 10 en orden y luego se pregunta si el 10 están el árbol ¿cuál es la complejidad?
2. Si se añaden los números en un orden aleatorio ¿cuál es la complejidad promedio de preguntar por el 10?