16 | TDA Árbol

Definiciones

Definición 16.1: 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 16.2: 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 16.3: Camino

Dado un árbol A que contiene al conjunto de nodos N, un *camino* en A se

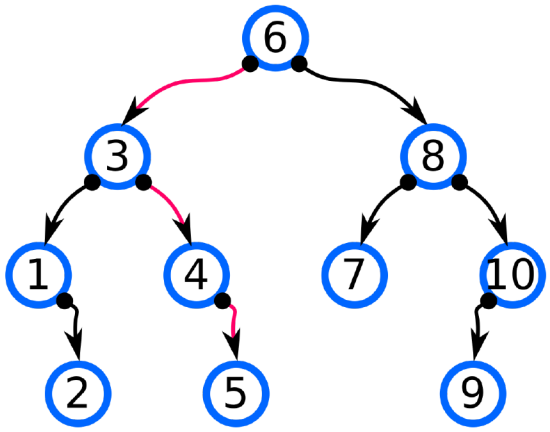

Figura 16.1 El camino de la raíz a un nodo es único. Su longitud es el número de aristas que contiene.

define como una secuencia de nodos distinta del vacío:

$$C = \{n_1, n_2, ..., n_k\}, \tag{16.1}$$

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 16.1]. 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.

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 16.4: 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.

17 | Árbol binario ordenado con referencias

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.

Código Auxiliar 17.1: Árbol binario ordenado

`https://github.com/computacion-ciencias/ed-arbol-binario-ordenado-cs`

Antecedentes

Nodos

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:

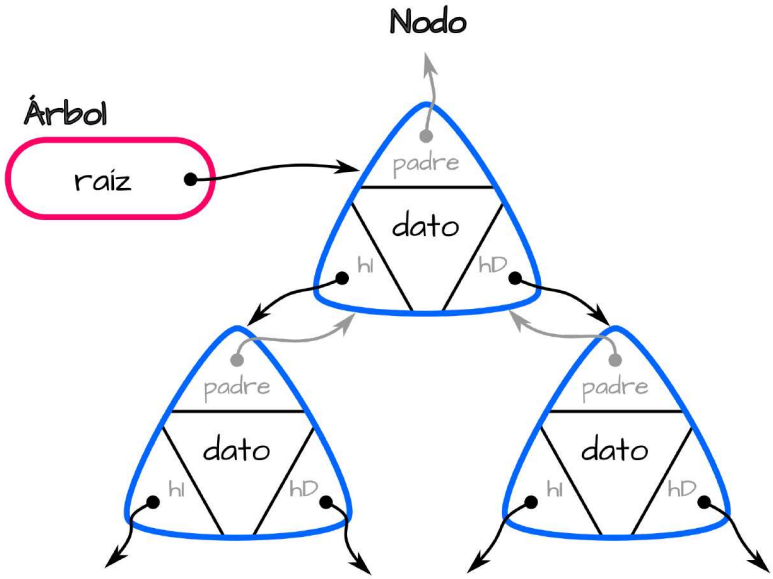

Figura 17.1 Estructura de datos árbol binario con nodos y referencias.

- 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 17.1 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, una vez más, implementar la interfaz `ICollection<C>`. Para ello deberemos añadir un requisito: en esta ocasión, el tipo de datos almacenable en la estructura debe implementar la interfaz `IComparable<C>`. A continuación se exponen los pasos a seguir.

Iteradores

Dado que hay varios criterios para recorrer los elementos en un árbol binario ordenado, es posible definir un iterador por cada tipo de recorrido. La idea básica para implementar estos recorridos es hacer uso de estructuras auxiliares. Qué estructura utilizar depende del tipo de recorrido:

Pila para los recorridos en *profundidad*: *preorden*, *inorden* y *postorden*.

Cola para los recorridos en amplitud.

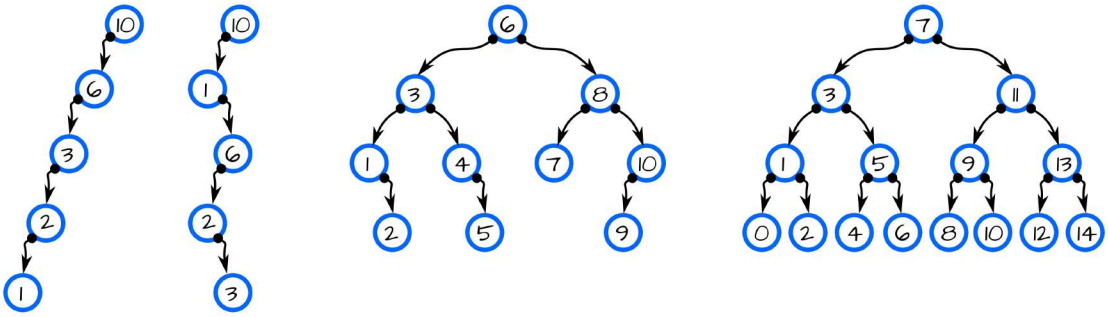

Figura 17.2 A la izquierda se muestran dos árboles degenerados; en medio, uno balanceado y a la derecha, uno lleno.

Algunas implementaciones crean estas estructuras explícitamente y almacenan en ellas los nodos que serán visitados próximamente conforme son descubiertos, de modo que el siguiente nodo a visitar es el primer nodo devuelto por la estructura auxiliar, hasta que ésta quede vacía. Observese que, en este caso, la complejidad en espacio es proporcional al número de nodos que esperan en dicha estructura. Para los recorridos en profundidad esta es la profundidad del árbol $\mathcal{O}(\log_2(n))$ y para el recorrido en amplitud es el ancho de cada nivel, si el árbol está lleno, en el último nivel esto es $\mathcal{O}(n)$.

Es posible implementar el recorriodo en profundidad en forma recursiva. De este modo la pila de ejecución de C# es utilizada para almacenar los nodos sobre la rama que se está visitando y, por cada rama, los hermanos que falta visitar. La complejidad en espacio sigue siendo $\mathcal{O}(\log_2(n))$ aunque la programación es mucho más clara, pero no permite implementar un iterador en C# donde los elementos deben ser devueltos uno por uno por cada llamada al método `MoveNext`.

La siguiente alternativa consiste en realizar una implementación iterativa. Es posible utilizar un pequeño número de variables para determinar el nodo actual y el nodo siguiente a visitar, si se analizan con cuidado los patrones de movimiento de la referencia al nodo siguiente a devolver. En este caso la complejidad espacial es $\mathcal{O}(1)$. Esta es, por lo tanto, la forma más eficiente de programar un iterador en profundidad.

Jerarquía

Los algoritmos de búsqueda, inserción y remoción expuestos anteriormente indican cómo construir y manipular árboles binarios manteniendo las propiedades de orden. Sin embargo, implementarlos así directamente puede provocar un desempeño muy poco eficiente en la práctica, pues la forma del árbol será altamente sensible al orden en que lleguen los datos. La Figura 17.2 muestra los estados en los cuales pueden quedar los árboles tras insertar algunos datos. Las operaciones tendrán complejidades $\mathcal{O}(\log_2(n))$ en el peor caso si el árbol está balanceado o lleno, pero $\mathcal{O}(n)$ si se encuentra degenerado. Por ello lo deseable es no caer en estos últimos casos.

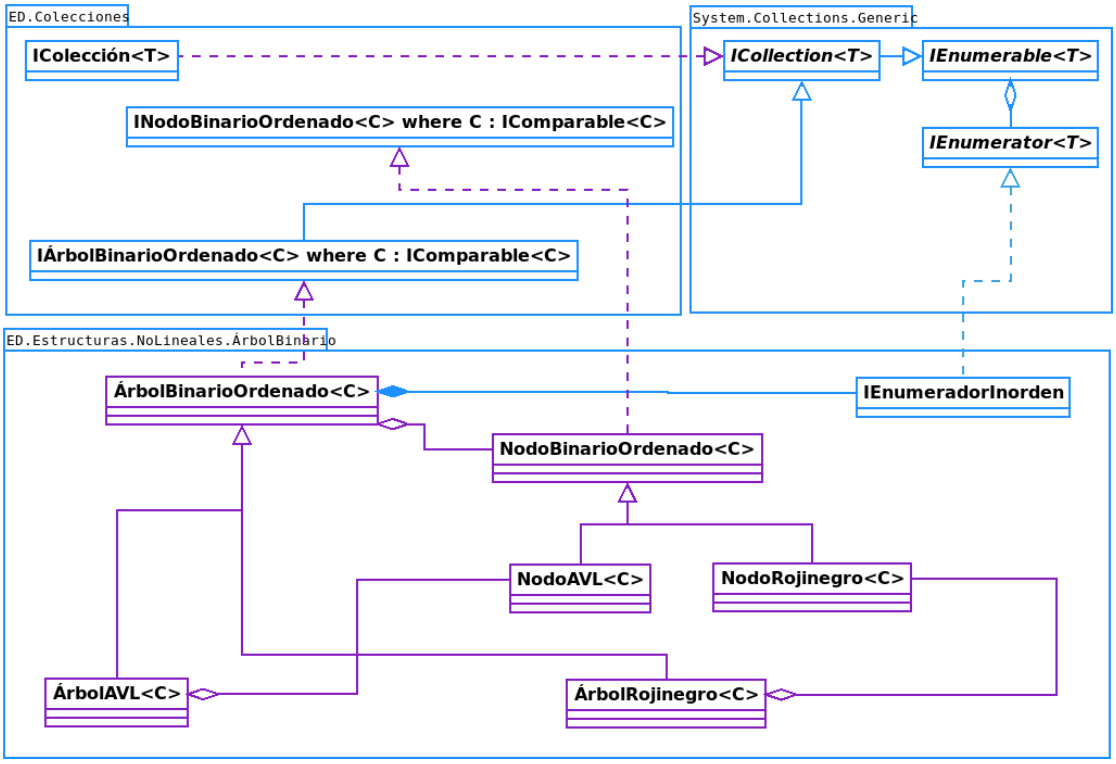

Figura 17.3 Jerarquía de clases para las implementaciones de árboles.

Para atender esta dificultad existen métodos de balanceo. En las prácticas siguientes se implementarán dos de ellos. En esta práctica implementarás un árbol binario ordenado que servirá como clase base para los árboles balanceados de las prácticas siguientes. Algunas de las decisiones de diseño, que se incorporarán en esta práctica, permitirán implementar las siguientes con mayor facilidad.

Para visualizar este diseño, la Figura 17.3 muestra la jerarquía de clases para los árboles que implementarás.

Desarrollo

Para implementar este **árbol binario ordenado** se deben programar, en el espacio de nombres `ED.Estructuras.NoLineales.ÁrbolBinario`, las clases:

- `NodoBinarioOrdenado<C>`, que implementa la interfaz `INodoBinarioOrdenado<C>` y sus datos son del tipo genérico `<C> where C : IComparable<C>`.
- `ÁrbolBinarioOrdenado<C>`, que implementa la interfaz `IÁrbolBinarioOrdenado<C>`, contendrá todo el código aplicable a cualquier árbol binario ordenado. No tiene que estar balancedo. Dado que `ÁrbolBinarioOrdenado<C>` implementa

IÁrbolBinarioOrdenado<C>,
que a su vez extiende ICollection<E> debe implementar todos los métodos definidos por las dos interfaces.

TIP: Programa todos los métodos de las clases NodoBinarioOrdenado<C> y ÁrbolBinarioOrdenadoI pero de forma que sólo devuelvan null, false y -1 según su tipo de regreso. Asegúrate de poder compilar. Cuando todo compile podrás ejecutar

```
dotnet test NPruebasÁrbolBinario/NPruebasÁrbolBinario.csproj
```

para ejecutar las pruebas unitarias. No olvides agregar la referencia de las pruebas unitarias hacia tu proyecto con el árbol y el nodo.

1. Comienza por programar la clase NodoBinarioOrdenado<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 INodoBinarioOrdenado<C>.

     Encontrarás que, al cambiar el tipo de regreso de los métodos/propiedades definidos en la interfaz, habrá que programarlos dos veces de un modo un tanto curioso. Por ejemplo, para obtener el hijo izquierdo de un nodo, harás:

```
public class NodoBinarioOrdenado<C> : INodoBinarioOrdenado<
    ↪ C> where C : IComparable<C>
{
    public NodoBinarioOrdenado<C>? HijoI { get; set; }
    INodoBinarioOrdenado<C>? INodoBinarioOrdenado<C>.HijoI
    {
        get => HijoI;
        set => HijoI = (NodoBinarioOrdenado<C>)value!;
    }
}
```

     Esta técnica permite que C#detecte que cumples con el requisito de la interfaz, pero que al llamar la propiedad el tipo del objeto que regresa sea el más específico.

     TIP: Agrega el método ToString, te ayudará a depurar tu código.

   - Crea los constructores siguientes:
     - ⋆ NodoBinarioOrdenado(C dato)
     - ⋆ NodoBinarioOrdenado(C dato, NodoBOLigado<C> padre)

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. Esto permitirá que las clases descendientes hereden el código para insertar nodos y ya sólo añadan sus modificaciones.

- Programa los otros métodos definidos en las interfaces `INodoBinarioOrdenado<C>` siguiendo las indicaciones en la documentación.

  TIP: La propiedad `Altura` debe contener el altura de cada nodo, esto permitirá responder en $\mathcal{O}(1)$. 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 sería posible programar un método `Altura()` utilizando la definición recursiva de altura de un nodo, la complejidad por cada llamada es $\mathcal{O}(n)$, por lo que en la práctica esto queda descartado.

2. Continúa con la clase `ÁrbolBinarioOrdenadoLigado<C>`.

   Crea la propiedad `Raíz` y no olvides llevar la cuenta de cuántos nodos tienes en `Count`.

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

```
protected NodoBinarioOrdenado<C> CreaNodo(C dato) { ... }
protected NodoBinarioOrdenado<C> CreaNodo(C dato,
                                          NodoBinarioOrdenado<C>
                                              ↪ padre) {
  ...
}
```

   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.

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

```
protected NodoBinarioOrdenado<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 en uso de memoria), o en el nodo si optas por la versión recursiva. Úsalo después para implementar las otras versiones de agregar que se te solicitan. No olvides actualizar el altura de los nodos en el camino.

5. Crea otro método auxiliar que te permita encontrar al primer nodo que contenga al dato pasado como parámetro y lo devuelva, al aprovechar las propiedades de orden

del árbol la complejidad deberá ser $\mathcal{O}(\log(n))$. Lo usarás en los métodos `Contains` y `Remove`.

6. Para remover nodos también debes usar un método auxiliar que implemente el algoritmo expuesto anteriormente, de modo que la complejidad promedio sea $\mathcal{O}(\log(n))$. Aquí utilizaremos un truco curioso para el futuro: este método debe implementar todo el algoritmo para remover, excepto la última parte en la que se quitaría al nodo hoja; en lugar de ello debe regresar la referencia a ese nodo. Esta es la parte del trabajo común a todas las implementaciones que programarás. Ahora, desde el método que mande llamar a éste es posible utilizar la propiedad `Padre` y el método `RemueveHijo` de `INodoBinarioOrdenado<C>` para borrar este último nodo.

7. Los métodos de `ICollection<E>` que debes implementar en `ÁrbolBinarioOrdenado<C>` son:

   - `public void Add(C e)` El método de inserción debe tener una complejidad promedio de $\mathcal{O}(\log n)$. Ya sólo necesitas llamar al auxiliar que programaste.
   - `public IEnumerator<C> GetEnumerator()` Deberás devolver un iterador inorden. No agregaremos métodos para insertar o remover con el iterador, pues más adelante no podremos garantizar que el iterador produzca un recorrido consistente después de remover un dato.
   - `public void Clear()` Para reducir el esfuerzo del recolector de basura, realiza un recorrido postorden desconectando todos los nodos.

Observa que `IÁrbolBinarioOrdenado<C>` solicita una propiedad pública que devuelve el nodo raíz, esta fue necesaria para que las pruebas unitarias pudieran realizar su tarea en forma eficiente. En general, si se quiere manejar la estructura del árbol es necesario mostrar los nodos. 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, en cuyo caso no hubiéramos querido que viera los nodos.

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?