

Clase 27

Árboles binarios de búsqueda

Árboles binarios de búsqueda

- En la clase anterior, definimos el concepto de árbol binario de búsqueda como un *árbol binario* de nodos que contienen una clave *ordenada* con la siguiente propiedad adicional: el subárbol a la izquierda de cada nodo (si existe) debe contener sólo nodos con claves menores o iguales al padre y el subárbol a la derecha (si existe) debe contener sólo nodos con claves mayores o iguales al padre.
- También vimos que el recorrido *inorder* de dicho árbol está ordenado.

Objetivos

- Implementaremos un árbol binario de búsqueda y analizaremos la implementación.
- Las generalizaciones de los árboles binarios de búsqueda aportan aplicaciones importantes, incluidas las bases de datos comerciales.
- La implementación le dará una idea de los métodos de árboles que suelen ser más "visuales" o "topológicos" que las implementaciones de arrays o listas.

3

Claves y valores

- Si los árboles binarios de búsqueda están ordenados, deben estarlo según alguna *clave* incluida en cada nodo del árbol.
- Un nodo puede contener únicamente la *clave*, pero suele resultar útil permitir que cada nodo contenga una *clave* y un *valor*.
- La *clave* se utiliza para analizar el nodo. El *valor* es un dato adicional del nodo indexado por la *clave*.

4

Mapas

- Las estructuras de datos con pares clave/valor suelen recibir el nombre de *mapas*.
- Como ejemplo, considere las entradas de una agenda de teléfonos tal como se insertarían en un árbol binario de búsqueda. El apellido del abonado sería la *clave* y el número de teléfono sería el *valor*.

5

Tipo de claves y valores

- A veces el valor no es necesario, pero si un nodo tiene un valor, probablemente quiera que todos lo tengan.
- Tampoco hemos especificado el tipo de la clave y del valor. Pueden ser de cualquier *clase* que desee, pero todos los nodos deben contener claves y valores que sean instancias de la misma clase.
- Las claves y los valores *no* tienen por qué ser del mismo tipo. Se pueden tratar como objetos en la implementación del árbol, salvo en el momento en que necesitemos compararlos.

6

Claves duplicadas

- ¿Permitimos que el árbol contenga nodos con claves idénticas?
- No hay nada en el concepto de los árboles binarios de búsqueda que impida las claves duplicadas.
- En la interfaz que implementemos, utilizaremos las claves para acceder a los valores. Si permitimos la existencia de claves idénticas, no podríamos distinguir nodos distintos que posean la misma clave.
- Por tanto, en esta implementación, no se permiten claves duplicadas. Los valores duplicados asociados a claves distintas sí están permitidos.

7

Interfaz SortedMap

```
public interface SortedMap {  
    // las claves no pueden ser null pero los valores sí  
    public boolean isEmpty();  
    public void clear();  
    public int size();  
    public Object get( Object key );  
    public Object firstKey();  
    public Object lastKey();  
    public Object remove( Object key );  
    public Object put( Object key, Object value );  
    public MapIterator iterator();  
}
```

8

Notas de SortedMap

- `get()`, `firstKey()`, `lastKey()` devuelven los valores adecuados y dejan el par clave/valor en el mapa.
- `firstKey()` y `lastKey()` devuelven `null` si el mapa está vacío
- `remove()` devuelve el valor si la clave está presente; de lo contrario, devuelve `null`
- `put()`: si la clave no está en el mapa, inserta el nuevo par clave/valor; si la clave ya está presente, devuelve el valor antiguo y lo sustituye por el valor nuevo en el mapa

9

Interfaz MapIterator

```
public interface MapIterator
{
    public boolean hasNext();
    public Object next()
        throws NoSuchElementException;
    public Object remove()
        throws IllegalStateException;
    public Object key()
        throws IllegalStateException;
    public Object getValue()
        throws IllegalStateException;
    public Object setValue( Object value )
        throws IllegalStateException;
}
```

10

Comparator

Si las claves son objetos arbitrarios, ¿cómo se ordenan?

El usuario puede crear su propio comparador

```
public interface Comparator
{
    // Ni o1 ni o2 pueden ser null
    public int compare(Object o1, Object o2);
}
```

11

StringComparator

```
public class StringComparator
    implements Comparator
{
    public int compare(Object o1, Object o2)
    {
        return ((String) o1).compareTo((String) o2);
    }
}

BinarySearchTree btree =
    new BinarySearchTree( new StringComparator() );
```

12

Interfaz Comparable

- En versiones recientes del JDK, todas las clases empaquetadas para tipos incorporados implementan una interfaz Comparable independiente:

```
public interface Comparable
{
    public int compareTo(Object o);
}
```

- Como String implementa la interfaz Comparable, no necesitamos implementar el comparador especial StringComparator.
- El método compareTo() devuelve ints con el mismo sentido que la interfaz Comparator.

13

Implementación de BinarySearchTree

- Tendremos un árbol y clases de nodos independientes para BinarySearchTree.
- Utilizaremos una clase interna estática privada para la clase del nodo, Branch, tal como hicimos para la clase SLink en SLinkedList.

```
public class BinarySearchTree
    implements SortedMap
{
    private Branch root = null;
    private int length = 0;
    private Comparator comparator = null;
```

14

Clase interna Branch

- Branch contiene enlaces a su nodo padre y a los subárboles derecho e izquierdo.
- Con esto, algunas operaciones, como eliminar un nodo, resultan mucho más eficaces.

```
static private class Branch
{
    private Branch left = null;
    private Branch right = null;
    private Branch parent = null;
    private Object key;
    private Object value = null;
```

15

Métodos sencillos

```
public BinarySearchTree( Comparator c )
{ comparator = c; }

public BinarySearchTree()
{ this( null ); }

public void clear() {
    root = null;
    length = 0;
}

public boolean isEmpty() {
    return ( root == null );
}
```

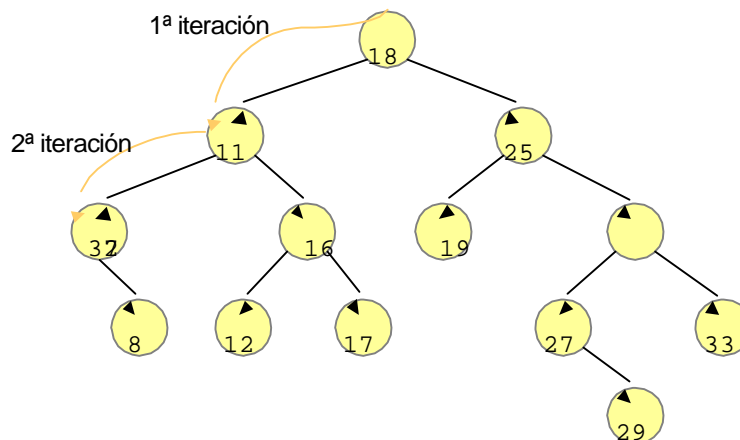
16

Método firstKey()

```
public Object firstKey() {  
    if ( root == null ) return null;  
    else return root.first().key;  
}  
  
// Branch:  
Branch first() {  
    Branch f = this;  
    while ( f.left != null )  
    { f = f.left; }  
    return f;  
}
```

17

firstKey() en acción



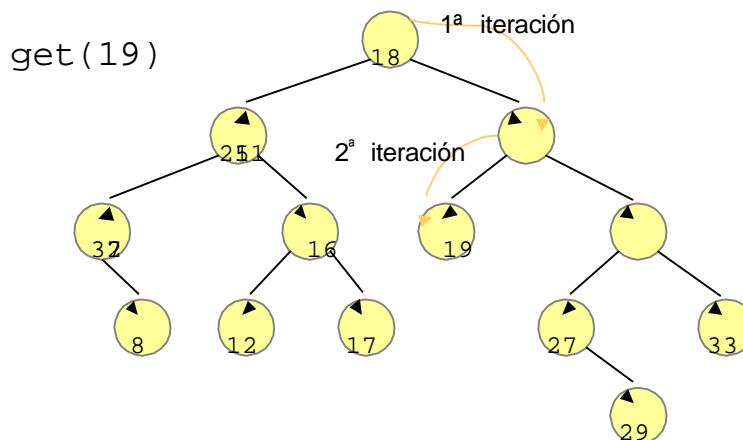
18

Método get()

```
public Object get( Object k ) {  
    Branch b = find( k );  
    return ( b == null ) ? null : b.value;  
}  
  
private Branch find( Object k ) {  
    Branch b = root;  
    int c;  
    while ( b != null && ( c=compare( k, b.key )) != 0 ) {  
        if ( c < 0 ) b = b.left;  
        else b = b.right;  
    }  
    return b;  
}
```

19

get() en acción



20

Método compare()

```
private int compare(Object k1, Object k2) {  
    if ( k1 == null || k2 == null )  
        throw new IllegalArgumentException(  
            "Clave null no permitida" );  
    // si hay un comparador, utilízelo  
    if ( comparator != null )  
        return comparator.compare( k1, k2 );  
    else {  
        return ((Comparable)k1).compareTo(k2);  
    }  
}
```

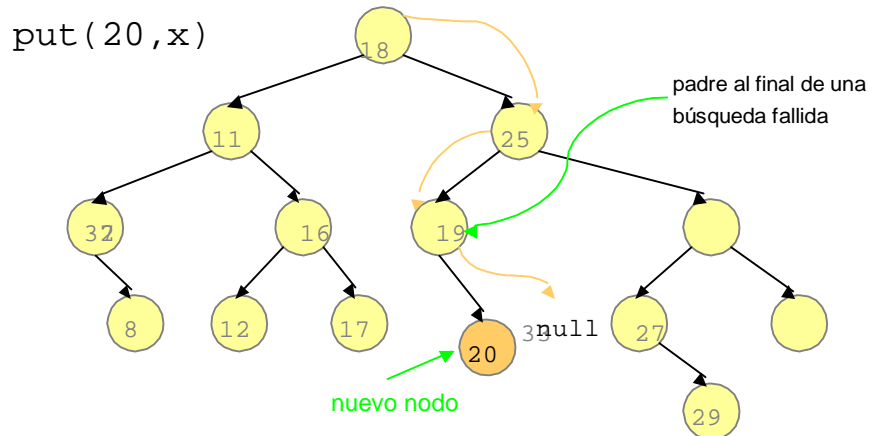
21

Estrategia put()

- Nuestra estrategia es buscar la clave suministrada como un argumento.
- Si la encontramos, devolveremos el valor antiguo después de reemplazarlo con el nuevo.
- Mientras buscamos el nodo, seguiremos de cerca nuestra búsqueda manteniendo una referencia al padre del nodo actual.
- Si no lo encontramos, la búsqueda finalizará al llegar a una rama *null*.
- En este caso, la referencia padre identificará al padre del nuevo nodo insertado.

22

put () en acción



23

Método put (), 1

```
public Object put(Object k, Object v)
{
    Branch p = null; Branch b = root; int c = 0;
    // busca el nodo con clave k manteniendo la ref p al padre
    while ( b != null ) {
        p = b; c = compare( k, b.key );
        if ( c == 0 ) {
            // lo encuentra; inserta un nuevo valor, devuelve
            old Object oldValue = b.value; b.value = v; return
            oldValue;
        } else if ( c < 0 )
            b = b.left;
        else
            b = b.right;
    }
}
```

Método put (), 2

```
// La clave k no existe en el árbol;
// inserta un nuevo nodo debajo de p(adre)
Branch newBranch = new Branch( k, v, p );
length++;
if ( p == null ) {
    root = newBranch;
} else {
    // c sigue conservando la última comparación
    if ( c < 0 )
        p.left = newBranch;
    else
        p.right = newBranch;
}
return null;
}
```

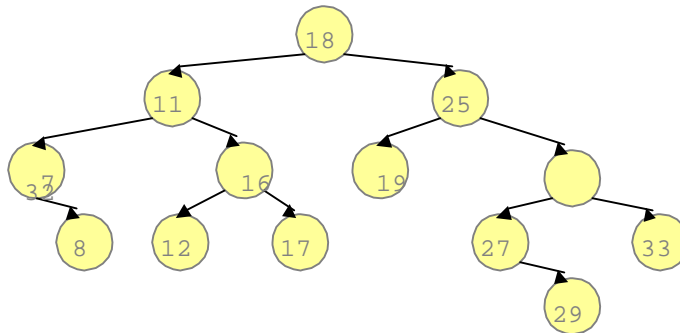
Estrategia delete()

- **remove()** llama a **delete()** para que haga casi todo.
- Cuando eliminamos un nodo de un árbol binario de búsqueda, debemos comprobar que la estructura resultante (1) sigue siendo un árbol y (2) que el árbol sigue cumpliendo la regla de un árbol binario de búsqueda.
- La regla requiere que, para cada nodo, las claves del subárbol izquierdo (si existe) preceda a la clave del nodo que debe preceder a las claves del subárbol derecho.

Casos de delete(), 1

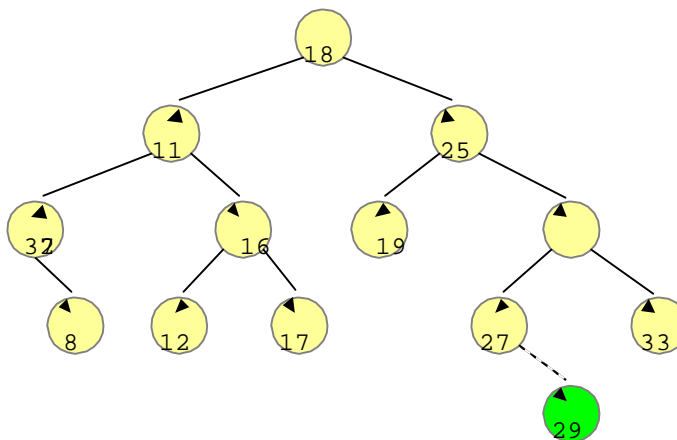
Hay tres casos de eliminación que debemos tener en cuenta:

1. El nodo eliminado no tiene hijos, p.ej., nodo 29 debajo.
2. El nodo eliminado tiene un hijo, p.ej., nodo 7 debajo.
3. El nodo eliminado tiene dos hijos, p.ej., nodo 25 debajo.



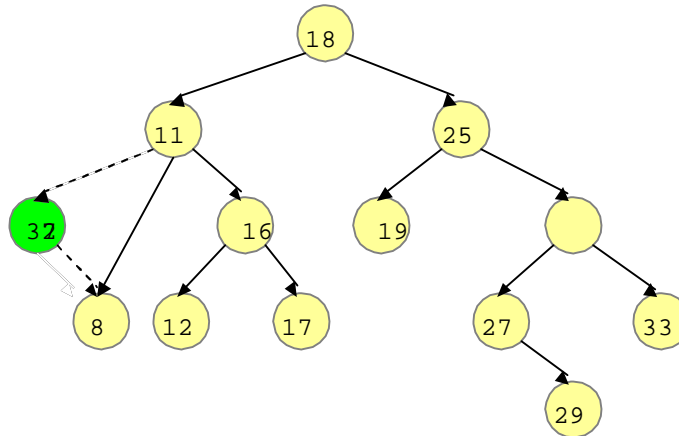
27

delete(), sin hijos



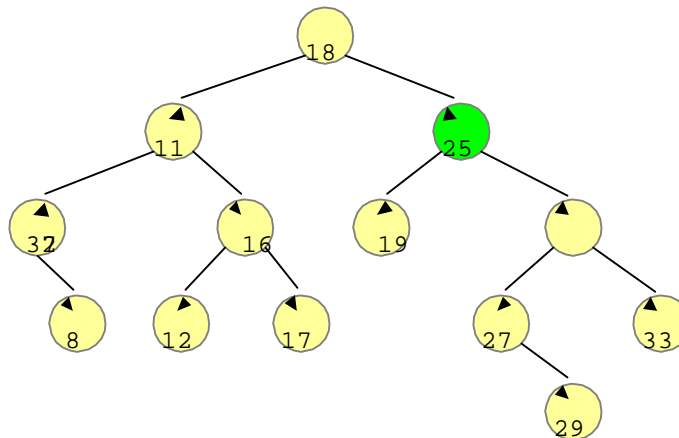
28

delete(), 1 hijo



29

delete(), 2 hijos



30

delete (), estrategia de 2 hijos

- En el último caso (dos hijos), el problema es más grave, ya que el árbol está ahora en tres partes.
- La solución comienza por darse cuenta de que es posible minimizar el problema del orden uniendo el árbol con el nodo que precede o sucede inmediatamente al nodo eliminado en la secuencia *inorder*. Estos nodos reciben el nombre de *predecesores* y *sucesores*.

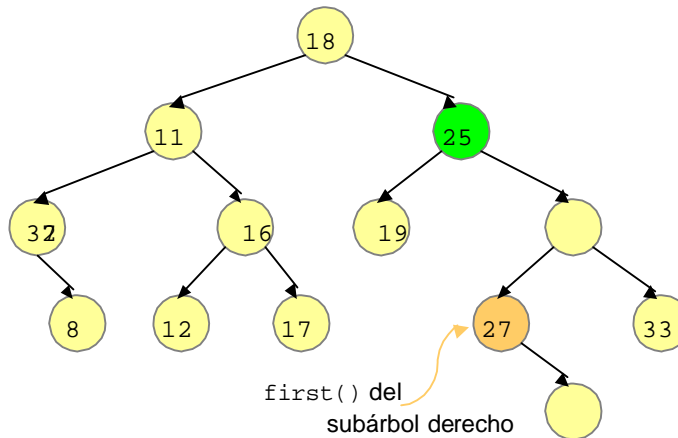
31

Uso del nodo sucesor

- Seleccionemos el nodo sucesor.
- El sucesor de un nodo con un subárbol derecho será el primer nodo `first()` de dicho subárbol.
- Si reemplazamos el nodo eliminado por su sucesor, se cumplirán todas las condiciones de orden. Pero, ¿qué ocurre con los hijos del sucesor?
- El sucesor de un nodo con dos hijos puede tener, como mucho, un hijo (a la derecha). Si tuviera un subárbol izquierdo, el nodo no podría ser el sucesor, ya que los miembros del subárbol izquierdo seguirían al nodo eliminado pero precederían al sucesor. Como el sucesor puede tener, como mucho, un subárbol, podemos mover el sucesor hacia arriba para reemplazar el nodo que queremos eliminar y volver a enlazar su subárbol derecho (si existe) como en el segundo caso mostrado anteriormente.

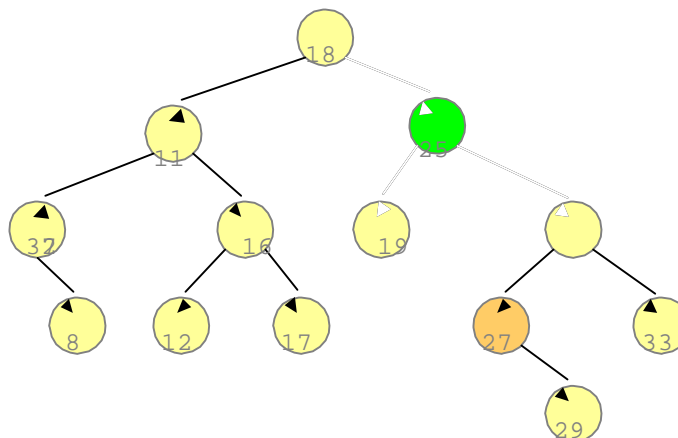
32

delete(), buscar sucesor



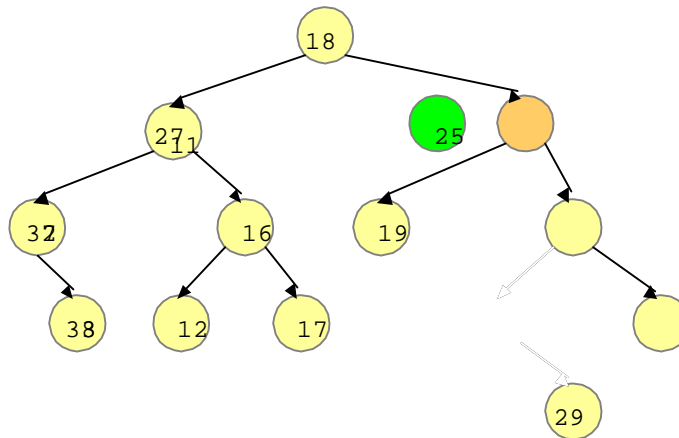
33

delete(), reemplazar sucesor 1



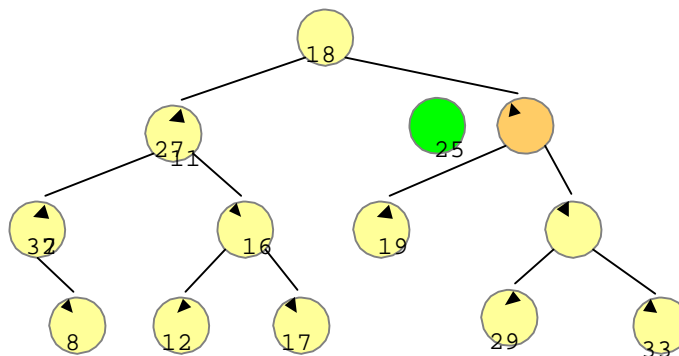
34

delete (), reemplazar sucesor 2



35

delete (), reemplazar sucesor 3



36

Eficacia de la búsqueda binaria

- Parece intuitivo pensar que las operaciones básicas del árbol binario de búsqueda deberían requerir un tiempo $O(h)$, donde h es la altura del árbol.
- Pero se deduce que la altura de un árbol binario *equilibrado* es, aprox., $\log_2(n)$, donde n es el número de elementos si el árbol permanece equilibrado.
- Se puede demostrar que, si las claves se insertan aleatoriamente en un árbol binario de búsqueda, esta condición se cumplirá y que el árbol permanecerá lo suficientemente equilibrado para que la hora de búsqueda y de inserción sea aproximadamente $O(\log n)$.

37

Equilibrio del árbol

- No obstante, existen algunos casos extremadamente simples y comunes en los que las claves no se insertan en orden aleatorio.
- Considere qué ocurriría si inserta claves en un árbol de búsqueda desde una lista ordenada. El árbol adoptará una forma degenerada equivalente a la de la lista de origen, y las horas de búsqueda e inserción disminuirán a $O(n)$.
- Hay muchas variantes de árboles, p.ej., árboles rojo-negro, árboles AVL, árboles B, que intentan solucionar este problema reequilibrando el árbol una vez concluidas las operaciones que lo desequilibran.

38

Claves insertadas en orden

