Análisis y diseño de software

dit
UPM

Tema 2: Algoritmos /diccionarios /java

José A. Mañas

http://jungla.dit.upm.es/~pepe/doc/adsw/index.html

8.3.2017

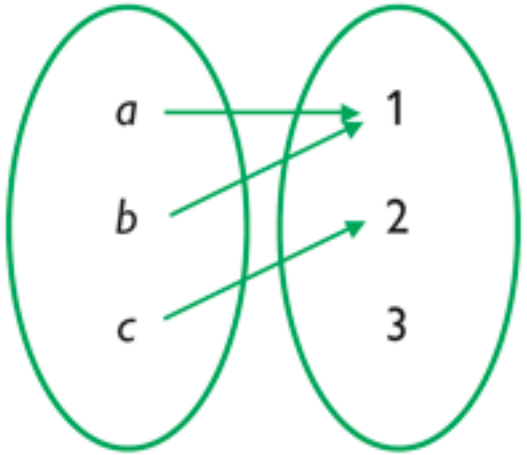
ETSIT UPM

referencias

- En la web
  - diccionarios / dictionaries
  - tablas de símbolos / symbol tables
  - lookup tables
  - map
  - algoritmos de búsqueda / search algorithms

java: Map

- Map<Clave, Valor> tabla=
  new HashMap<Clave, Valor>;
- tabla.put(“rojo”, “red”);
  tabla.put(“verde”, “green”);
  tabla.put(“negro”, “black”);
- String ingles = tabla.get(“rojo”);

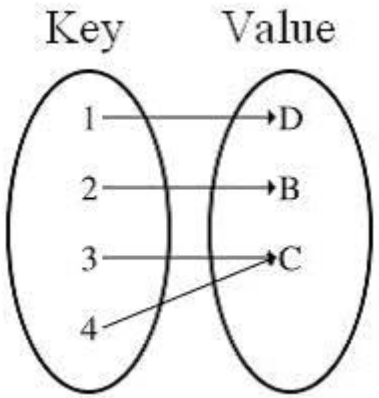


java: Set: conjunto

- Set<String> grupo=
  new HashSet<String>;
- grupo.add(“Juan”);
  grupo.add(“Ana”);
- grupo.contains(“Ana”) → TRUE

interface Diccionario

```
public interface Diccionario<K, V> {

    void put(K clave, V valor);

    V get(K clave);

    V remove(K clave);

    int size();

    void clear();
}
```



índice

diccionarios

- **arrays**
- arrays ordenados
- tablas hash
- BST - árboles binarios de búsqueda

con arrays

- put
  - O(1) si se añade al final (sin detectar duplicados)
    - List<E>
  - O(n) si se mira primero si hay un duplicado
- get
  - recorre el array comparando
  - O(n)
- remove
  - recorre el array comparando
  - O(n)

búsqueda lineal

```
String get(String clave) {
    for (int i = 0; i < n; i++) {
        CV cv = datos[i];
        if (cv.getClave().equals(clave))
            return cv.getValor();
    }
    return null;
}
```

```
public class CV {
    private final String clave;
    private String valor;

public CV(String clave, String valor) {
        this.clave = clave;
        this.valor = valor;
    }

public String getClave() { return clave; }

public String getValor() { return valor; }

public void setValor(String valor) { this.valor = valor; }
}
```

cálculo de complejidad

- recorriendo el array de 1 en 1
  - $T(n) = T(n-1) + O(1)$
  - $T(n) = T(n-2) + O(1) + O(1) = T(n-2) + 2\ O(1)$
  - $T(n) = T(n-3) + O(1) + O(1) + O(1) = T(n-3) + 3\ O(1)$
- generalizando
  - $T(n) = T(n-k) + k\ O(1)$
- se termina cuando
  - $n-k = 0 \Rightarrow k = n$
  - $T(n) = T(0) + n\ O(1) = O(n)$

índice

diccionarios

- arrays
- **arrays ordenados**
- tablas hash
- BST - árboles binarios de búsqueda

arrays ordenados

búsqueda binaria

- miramos en la mitad del array
- si ahí está el dato que buscamos,
  esa es la respuesta
- si ahí hay un dato demasiado grande
  entonces hay que buscar por la izquierda
- si no
  entonces hay que buscar por la derecha

busca recursivo

```
// busca en el intervalo [a, z)
   private int busca(String clave, int a, int z) {
      if (a >= z)
         return a;
      int m = (a + z) / 2;
      int cmp = clave.compareTo(datos[m].clave);
      if (cmp == 0)
         return m;
      if (cmp < 0)
         return busca(clave, a, m);
      else
         return busca(clave, m + 1, z);
   }
```

busca iterativo

```
// busca en el intervalo [a, z)
   private int busca(String clave, int a, int z) {
      while (a < z) {
         int m = (a + z) / 2;
         int cmp = clave.compareTo(datos[m].clave);
         if (cmp == 0)
            return m;
         else if (cmp < 0)
            z= m;
         else
            a= m+1;
      }
      return a;
   }
```

ejemplo

- int[] datos = {2, 3, 5, 7, 11, 13, 17, 19};
- busca(3, 0, 8)
  - a = 0; z = 8; m = 4; datos[m] = 11
  - a = 0; z = 4; m = 2; datos[m] = 5
  - a = 0; z = 2; m = 1; datos[m] = 3
  - return 1 ← donde está el dato
- busca(4, 0, 8)
  - a = 0; z = 8; m = 4; datos[m] = 11
  - a = 0; z = 4; m = 2; datos[m] = 5
  - a = 0; z = 2; m = 1; datos[m] = 3
  - a = 2; z = 2
  - return 2 ← donde estaría el dato

cálculo de complejidad

- recorriendo por mitades
  - $T(n) = T(n/2) + O(1)$
  - $T(n) = T(n/4) + O(1) + O(1) = T(n/4) + 2\ O(1)$
  - $T(n) = T(n/8) + O(1) + O(1) + O(1) = T(n/8) + 3\ O(1)$
- generalizando
  - $T(n) = T(n/2^k) + k\ O(1)$
- se termina cuando
  - $n/2^k = 1 \Rightarrow k = \log_2(n)$
  - $T(n) = T(1) + k\ O(1) = O(\log_2(n)) = O(\log(n))$

con arrays ordenados

- put
  - busca la posición donde entraría
  - reemplaza → O(1)
  - hace sitio desplazando n/2 → O(n)
  - peor: O(put) = O(busca) + O(n/2) = O(log n) + O(n) = O(n)
- get
  - busca la posición de donde saldría
  - saca o null
  - O(get) = O(busca) = O(log n)
- remove
  - busca la posición donde estaría
  - null o hay que desplazar n/2
  - peor: O(remove) = O(busca) + O(n/2) = O(log n) + O(n) = O(n)

tiempos de búsqueda

N	lineal	iterativa	recursiva
1.000	1.906.034	48.441	67.325
2.000	97.292	72.251	67.325
5.000	192.533	74.304	89.082
10.000	377.266	74.714	86.619
20.000	743.037	84.156	93.598
50.000	743.037	111.661	109.198
100.000	3.867.490	98.114	119.051
200.000	8.075.298	131.777	159.691
500.000	20.026.298	140.397	163.796
1.000.000	46.863.089	150.249	172.417
2.000.000	95.695.021	172.828	210.596
5.000.000	215.625.791	78.409	220.038
10.000.000	391.357.448	138.755	227.837

sin precalentamiento

ordenación perezosa

- Si en un objeto
  - hay que meter datos (comparables)
  - hay que buscar datos
- ¿qué nos interesa más?
  1. inserción ordenada + búsqueda binaria
  2. inserción tal cual + búsqueda lineal
  3. inserción + ordenación + búsqueda binaria
  - depende de la proporción de cada operación

ordenación perezosa

- metemos sin preocuparnos del orden
  - datos[pos++] = nuevo dato;
    ordenada = false;
- cuando hay que sacar ordenamos
  - if (ordenada = false) {
        sort(datos, 0, pos);
        ordenada = true;
    }
    return busqueda_binaria(datos, dato)

índice

diccionarios

- arrays
- arrays ordenados
- **tablas hash**
  - con listas (resolución externa)
  - direccionamiento abierto (resolución interna)
- BST - árboles binarios de búsqueda
- estructuras de datos de java
  - HashMap
  - TreeMap (BST)

tablas hash

- hash con listas de desbordamiento
  - resolución externa de colisiones

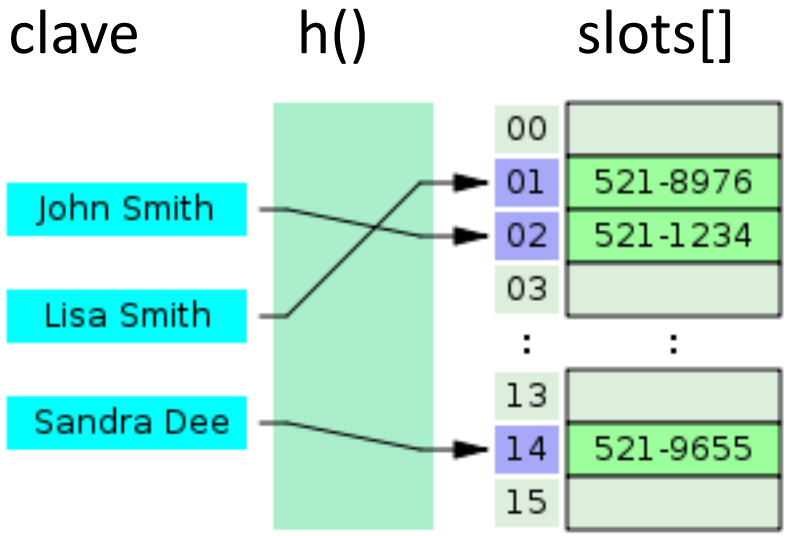
- hash abierto
  - resolución interna de colisiones

hash - motivación

- se trata de tener operaciones de inserción y búsqueda
  - tiempo constante: O(1)
- son un ejemplo de
  - o invertimos en memoria RAM
  - o invertimos en tiempo de CPU

una ranura para una clave

- Tenemos una tabla holgada donde caben muchos valores
- A partir de la clave elegimos dónde colocar el valor



función hash

- Si tenemos una tabla de k ranuras (slots)
- int h(Clave x)
  - acepta un objeto x
  - genera un valor [0 .. N-1]
- En java, todos los objetos tienen un método
  - int hashcode()
  - int h(Object x) {

```
    int codigo = Math.abs(x.hashcode());
    return codigo % k;
}
```

si no es java o no nos gusta hashcode()

- entonces tiene que inventarse algo
  - que sea rápido
  - que disperse bien los valores (sin colisiones)
- por ejemplo

```
int h(String s) {
        int n = 0;
        for (char ch : s.toCharArray())
            n = n * 31 + ch;
        return n;
    }
```

any Object

```
@Override
public boolean equals(Object o) {
   if (this == o) return true;
   if (o == null || getClass() != o.getClass()) return false;
   Fraccion fraccion = (Fraccion) o;
   return n == fraccion.n && d == fraccion.d;
}

@Override
public int hashCode() {
   return 31 * n + d;
}
```

```
public class Fraccion {
   private int n;   // numerador
   private int d;   // denominador

   public Fraccion(int n, int d) {
      if (d == 0)
         throw new IllegalArgumentException("d ==0");
      this.n = n;
      this.d = d;
}
```

otros hash

- search for non-cryptographic hash functions

Non-cryptographic hash functions [edit]

Name	Length	Type
Pearson hashing	8 bits	xor/table
Paul Hsieh's SuperFastHash[1]	32 bits	
Buzhash	variable	xor/table
Fowler–Noll–Vo hash function (FNV Hash)	32, 64, 128, 256, 512, or 1024 bits	xor/product or product/xor
Jenkins hash function	32 or 64 bits	xor/addition
Java hashCode()	32 bits	
Bernstein hash *djb2*[2]	32 bits	
PJW hash / Elf Hash	32, 64 bits	hash
MurmurHash	32, 64, or 128 bits	product/rotation
SpookyHash	32, 64 or 128 bits	see Jenkins hash function
CityHash	64, 128, or 256 bits	
numeric hash (nhash)[3]	variable	Division/Modulo
xxHash[4]	32, 64 bits	

colisiones

- ¿Qué hacer si colisionan?
  - if (h(x1) == h(x2)) { ... }
- Soluciones
  1. lista de valores
     - operaciones de meter, sacar y buscar en listas
  2. direccionamiento abierto
     - se buscan otros posibles huecos, de forma metódica, hasta que encuentra un hueco vacío
     - debe haber algún sitio vacío
     - es difícil eliminar

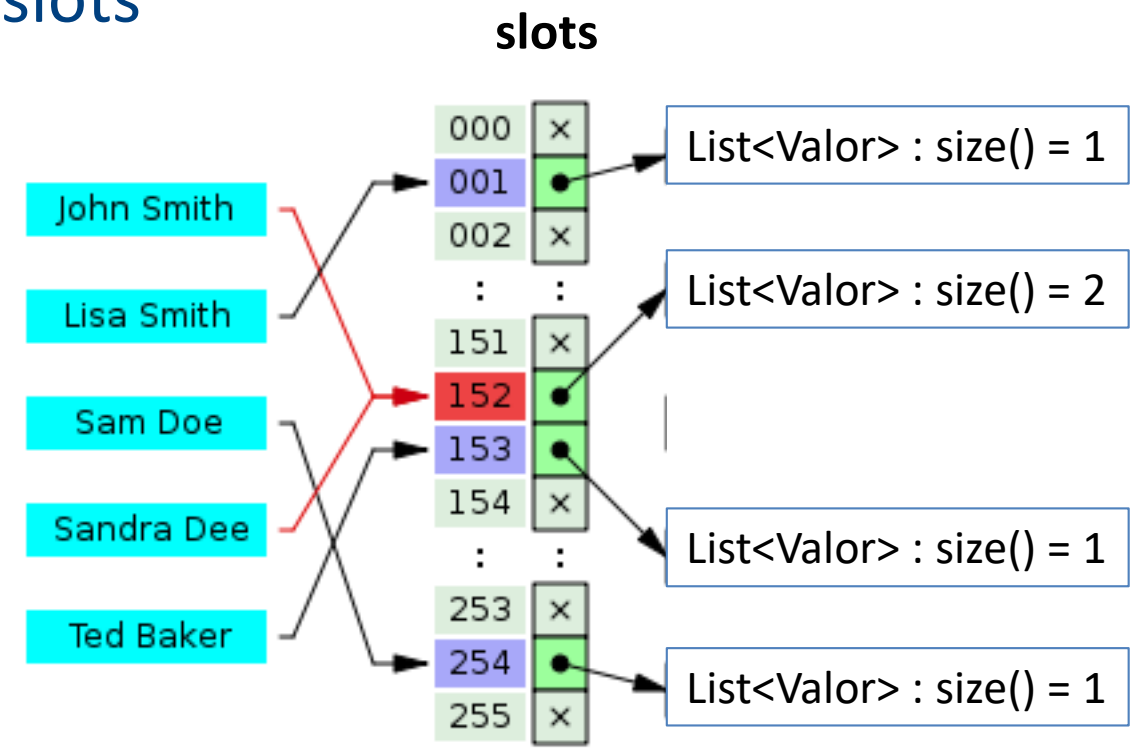
índice

diccionarios

- arrays
- arrays ordenados
- tablas hash
  - **con listas (resolución externa)**
  - direccionamiento abierto (resolución interna)
- BST - árboles binarios de búsqueda
- estructuras de datos de java
  - HashMap
  - TreeMap (BST)

hash con lista de valores

- En lugar de 1 valor
  ponemos una lista de valores
  - List<Valor>[] slots



tablas hash con listas

- complejidad
  - depende de la complejidad de las listas
- insertar, buscar, eliminar
  - O(1) + O(operación en la lista)
- si hay k ranuras (slots)
  y las claves están uniformemente distribuidas
  - lista.size() $\sim N/k$
  - el tiempo se divide por k
  - complejidad $\sim O(1) + O(N/k) = O(N)$
- gastamos RAM; pero ahorramos tiempo

índice

diccionarios

- arrays
- arrays ordenados
- tablas hash
  - con listas (resolución externa)
  - **direccionamiento abierto (resolución interna)**
- BST - árboles binarios de búsqueda

direccionamiento abierto

- vamos buscando por la lista
  - pos = hash(valor)
  - while (slots[pos] está ocupada)
  - pos = siguiente(pos);
- puede llenarse la tabla

direccionamiento abierto

prueba lineal

- pos = hash(valor);
- for (int i= 0; i < N; i++)
- probamos con “(pos + i) % N”

- problema: se forman ristras de valores
  - en cuanto un valor cae en la ristra hay que recorrerla hasta el final

direccionamiento abierto

prueba cuadrática

pos = hash(valor);

for (int i= 0; i < N; i++)

probamos con “(pos + i * i) % N”

- se forman ristras de valores
  - pero son ristras llenas de agujeros
    y es **“poco probable”** que un valor caiga en la trampa

direccionamiento abierto

prueba con segundo hash

```
pos = hash(valor);
h2= hash_alternativo(valor);
for (int i= 0; i < N; i++)
    probamos con "(pos + i * h2) % N"
```

- se forman ristras de valores
  - pero son ristras llenas de agujeros
    y es **"poco probable"** que un valor caiga en la trampa

direccionamiento abierto

- si la tabla está poco llena
  - hay pocas colisiones
    o se resuelven deprisa
  - algoritmos O(1)
- si la tabla de slots se llena
  - hay muchas colisiones
    y exigen recorrer la tabla
  - algoritmos O(n)
- o sea, o gastamos RAM o gastamos tiempo

tablas hash

- interface Map
- class HashMap implements Map
  - diccionario con listas para colisiones
  - cuando la tabla de slots se llena,
    se duplica la tabla y se reubican valores en nuevas listas
  - “garantiza” O(1)
- class TreeMap implements Map
  - diccionario con árbol de búsqueda
  - operaciones O(log n)
  - ventaja: mantiene los datos ordenados

tiempos tablas hash

- factor de carga = 60%

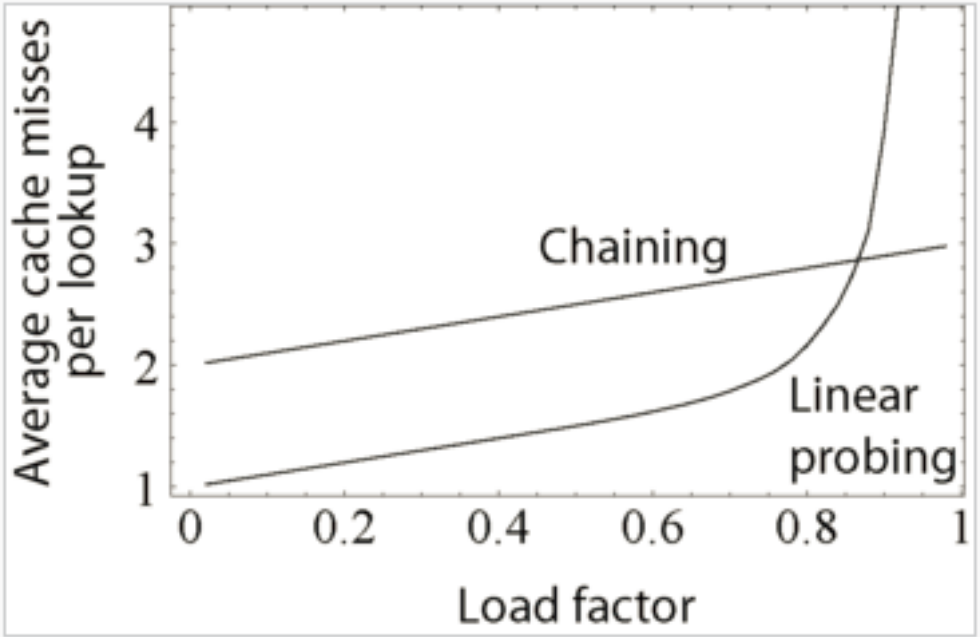
N	listas	abierta	HashMap	TreeMap
1.000	52.957	51.725	58.705	67.736
2.000	51.726	52.136	64.041	82.925
5.000	43.104	45.157	61.168	83.336
10.000	50.905	43.925	43.105	124.388
20.000	44.746	44.747	23.399	92.367
50.000	34.894	34.484	27.095	102.630
100.000	38.589	39.410	28.325	169.134
200.000	45.567	44.336	38.999	59.936
500.000	50.494	48.442	35.536	72.662

tiempos tablas hash

en función del factor de carga

carga	abierta	listas
0,50	33.252	113.304
0,60	36.947	49.262
0,70	34.894	59.115
0,80	39.821	53.367
0,90	59.935	66.505
0,95	82.514	69.788
0,96	97.703	46.799
0,97	101.398	51.314
0,98	275.459	55.420
0,99	236.049	52.547
1,00		56.651
2,00		101.398
5,00		63.630
10,00		70.199

tablas hash



This graph compares the average number of cache misses required to look up elements in tables with chaining and linear probing. As the table passes the 80%-full mark, linear probing's performance drastically degrades.

https://en.wikipedia.org/wiki/Hash_table

índice

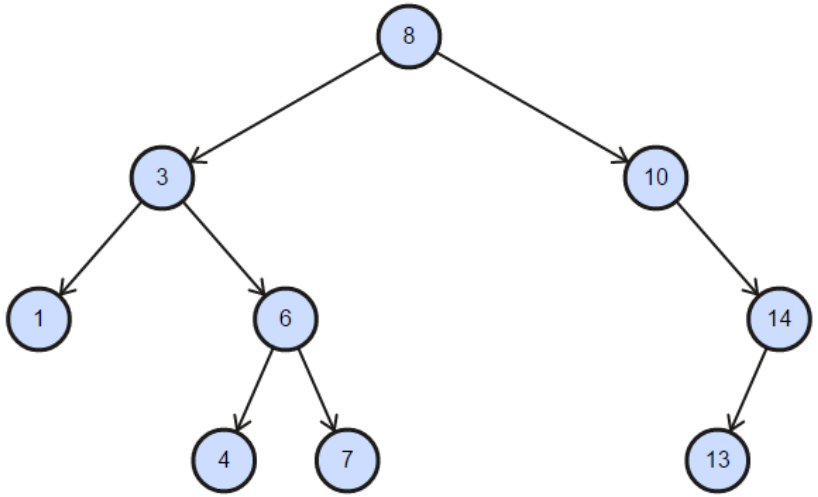
diccionarios

- arrays
- arrays ordenados
- tablas hash
- **BST - árboles binarios de búsqueda**

árboles binarios de búsqueda

Binary Search Trees

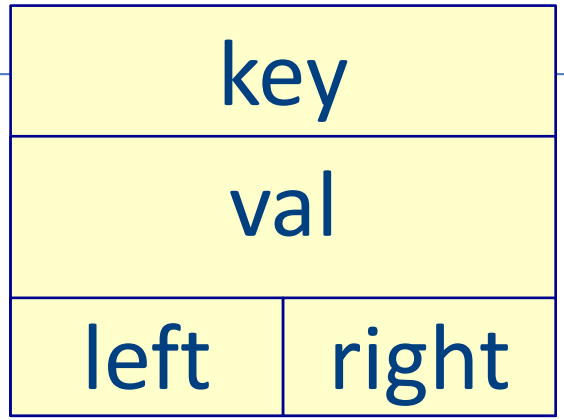
- árbol
  - cada nodo padre tiene 2 hijos
  - todos los descendientes por la izquierda tienen valores menores que el padre
  - todos los descendientes por la derecha tienen valores mayores que el padre



BST- estructura

```
class BST {
   private Node root;
```

```
class Node {
      String key;
      Object val;
      Nodo left;
      Nodo right;
   }
```



BST– funciones típicas

```
public int depth() {
      return depth(root);
   }

   private int depth(Node node) {
      if (node == null)
         return 0;
      return 1 + max(depth(node.left) + depth(node.right));
   }
```

```
public int size() {
      return size(root);
   }

   private int size(Node node) {
      if (node == null)
         return 0;
      return 1 + size(node.left) + size(node.right);
   }
```

BST– funciones típicas

```
public String getMin() {
      if (root == null)
         return null;
      return getMin(root);
   }

   private String getMin(Node node) {
      if (node.left != null)
         return getMin(node.left);
      return node.key;
   }
```

```
public String getMin() {
      if (root == null)
         return null;
      Node node = root;
      while (node.left != null)
         node = node.left;
      return node.key;
   }
```

BST- búsqueda

```
public Object get(String key) {
   return get(root, key);
}

private Object get(Node node, String key) {
   if (node == null)
      return null;
   if (key.equals(node.key))
      return node.val;
   if (key.compareTo(node.key) < 0)
      return get(node.left, key);
   else
      return get(node.right, key);
}
```

```
public Object get(String key) {
      Node node = root;
      while (node != null) {
         if (key.equals(node.key))
            return node.val;
         if (key.compareTo(node.key) < 0)
            node = node.left;
         else
            node = node.right;
      }
      return null;
   }
```

BST - modificación

- inserción
  - put(String key, Object val)
- eliminación
  - remove(String key)
- invariante
  - boolean isBST()

BST - invariante

```
public boolean isBST() {
      return isBST(root, getMin(root), getMax(root));
   }

   private boolean isBST(Node node, String min, String max) {
      if (node == null)
         return true;
      if (min.compareTo(node.key) > 0)
         return false;
      if (max.compareTo(node.key) < 0)
         return false;
      return isBST(node.left, min, node.key) &&
             isBST(node.right, node.key, max);
   }
```

BST - inserción

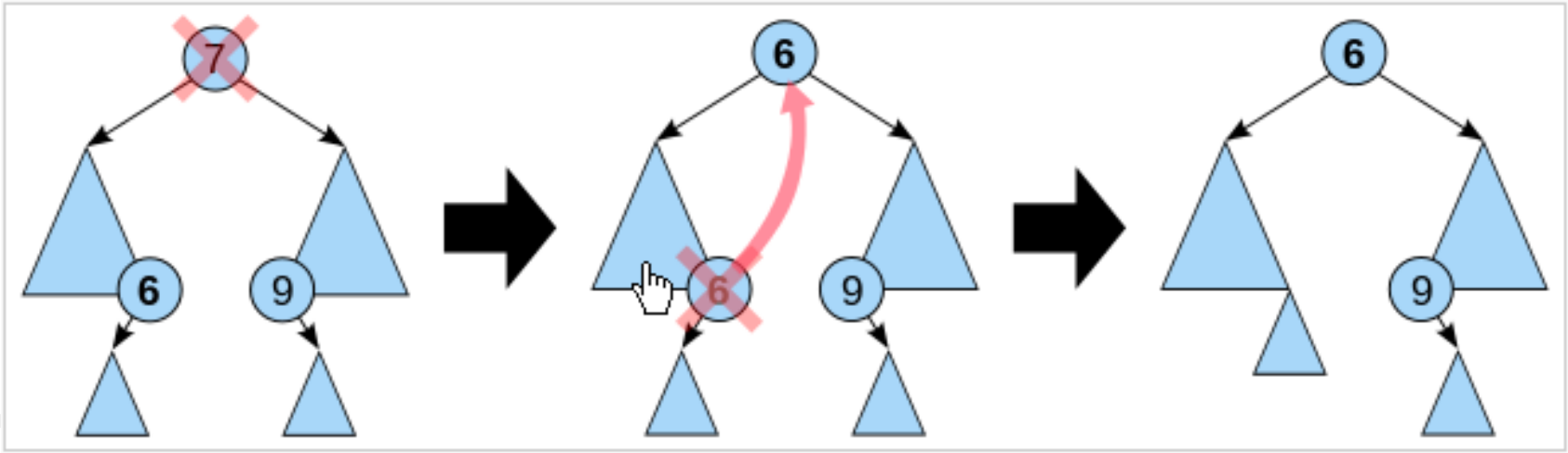
```
public void put(String key, Object val) {
      root = put(root, key, val);
   }

   private Node put(Node node, String key, Object val) {
      if (node == null)
         return new Node(key, val);
      if (key.equals(node.key))
         node.val = val;
      else if (key.compareTo(node.key) < 0)
         node.left = put(node.left, key, val);
      else
         node.right = put(node.right, key, val);
      return node;
   }
```

BST - eliminación

- caso 1: sin hijos
  - se retira el enlace del padre
- caso 2: 1 hijo
  - se reemplaza por el hijo
- caso 3: 2 hijos
  1. se busca el mayor hijo izquierdo
  2. reemplaza al nodo a eliminar
  - [… o el menor hijo derecho]

BST - eliminación

- queremos eliminar el 7
- el mayor hijo por la izquierda: 6
- eliminamos el 6
  - caso 2: no tiene hijos por la derecha
- el 6 reemplaza al 7

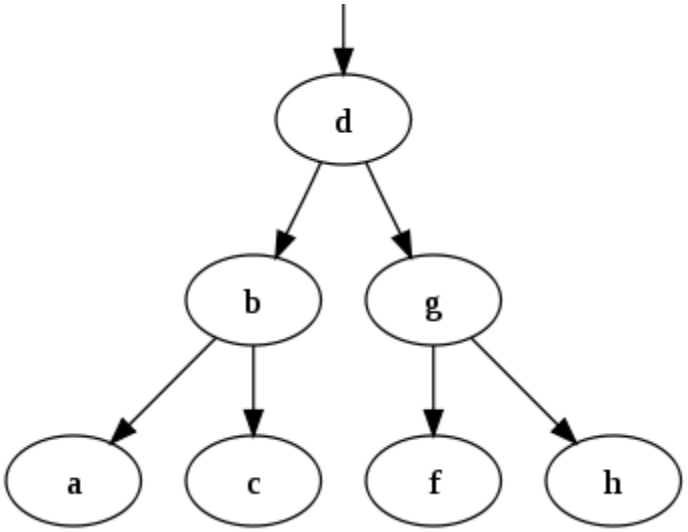


BST - complejidad

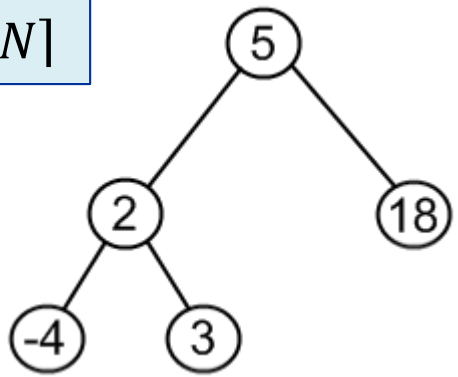
- todas las operaciones recorren un árbol
- sea k la profundidad (o altura) del árbol
  - búsqueda - get
    - $T(n) = c \times k$
  - inserción - put
    - $T(n) = c \times k$
  - eliminación - remove
    - T(n) = T(busca nodo) + T(busca nieto) + T(inserta)

árbol ideal = equilibrado

- árbol completo
- $N = 2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} = 2^k - 1$



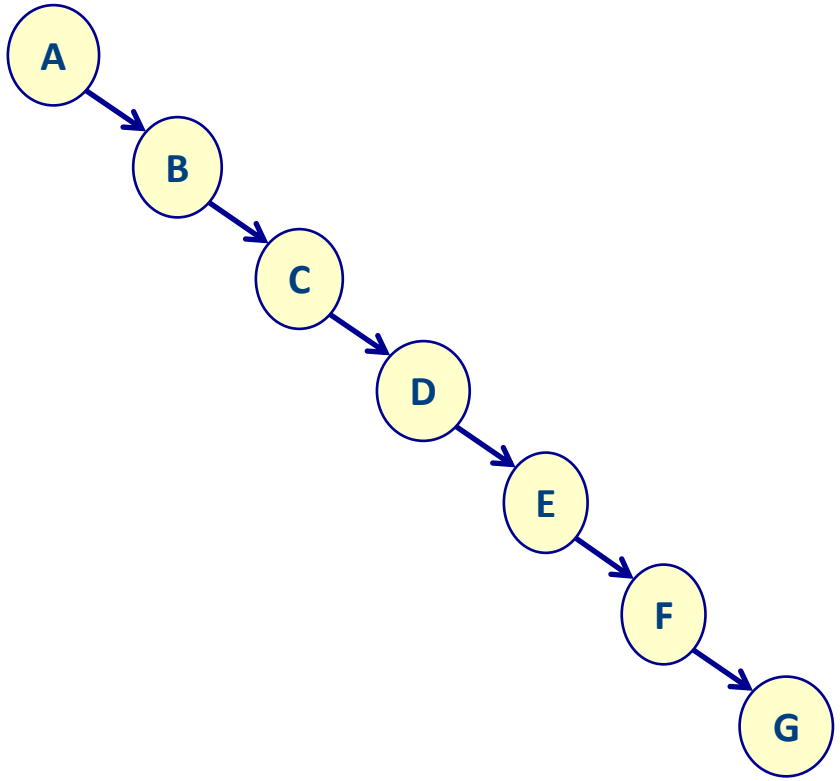
generalizando, para N nodos necesitaremos $k = \lceil \log_2 N \rceil$



árbol degenerado

- profundidad k = N

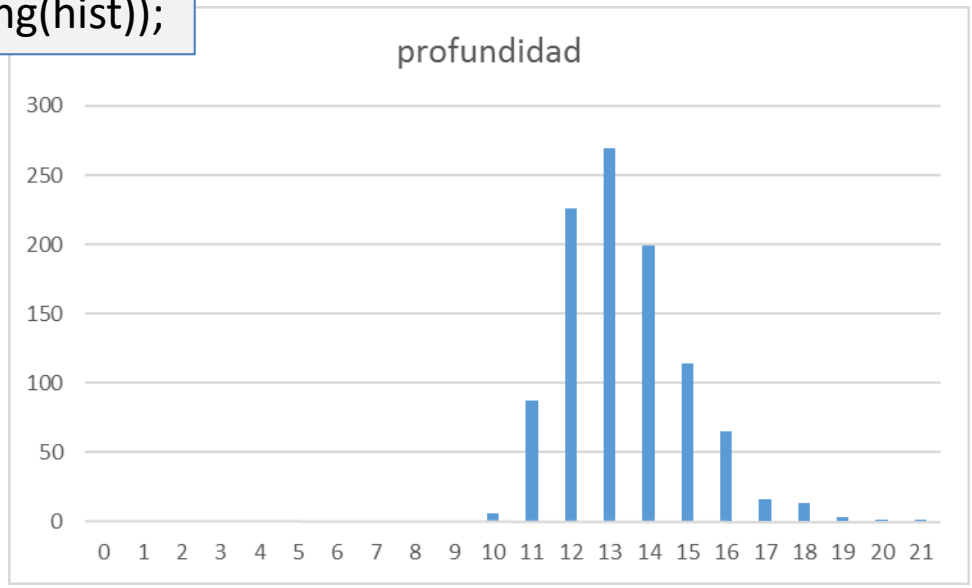
```
mete("A");
mete("B");
mete("C");
mete("D");
mete("E");
mete("F");
mete("G");
```



¿suelen estar equilibrados?

```
int[] hist = new int[100];
for (int i = 0; i < 1000; i++) {
    BST bst = load(100);
    int depth = depth(bst.getRoot());
    hist[depth]++;
}
System.out.println(Arrays.toString(hist));
```

para 100 datos:
ideal: 7 niveles (hasta 127 datos)

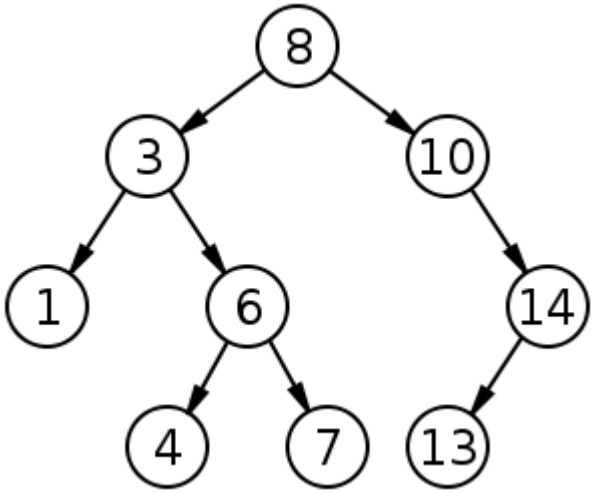


BST - complejidad

- suponiendo que
  - los casos degenerados son la excepción
  - el caso medio estará bastante equilibrado
- búsqueda
  - O(log n)
- inserción
  - O(log n)
- eliminación
  - busca sitio → busca nieto → inserta
  - O(log n) + O(log n) + O(1) = O(log n)

árboles: recorridos típicos

- preorden
  - 8 3 1 6 4 7 10 14 13
- en orden
  - 1 3 4 6 7 8 10 13 14
- postorden
  - 1 4 7 6 3 13 14 10 8




```
void inorder(List<String> list, Node node) {
    if (node != null) {
        inorder(list, node.left);
        list.add(node.key);
        inorder(list, node.right);
    }
}
```

complejidad (caso mejor)

	put	get	remove
arrays	**O(1) [con dups]** O(n) [sin dups]	O(n)	O(n)
arrays ordenados	O(n)	O(log n)	O(n)
BST	O(log n)	O(log n)	O(log n)
hash con listas	**O(1) si nSlots >> nDatos** O(N) si nSlots << nDatos		
hash abierto carga << 1	O(1)		
HashMap (con rehash)	O(1)		
TreeMap	O(log n)		

índice

diccionarios

- dicc. lineal
- dicc. binario
- tablas hash
- BST – binary search trees

conclusiones

- se puede hacer de muchas formas
  - unas son más rápidas que otras
- en la práctica tendremos que
  - elegir algoritmos según nos convenga
  - medir, medir y medir

optimización de programas

- como regla general
  - el 95% de los recursos de un programa
  - los consume el 5% del código

1. identifique ese 5%
2. optimice ese 5%
   1. elija un algoritmo adecuado
   2. refactorice el código para acelerarlo al máximo
      - variables temporales para evitar cálculos repetidos
      - reutilización de objetos para evitar new X()