Algoritmos y Estructuras de Datos
Tema 2: Diseño de Algoritmos

Contenidos

- 1. Algoritmos recursivos
  - 1.1 Algoritmos recursivos. Recursión simple
  - 1.2 Algoritmos con vuelta atrás y ejemplos
- 2. Complejidad de los algoritmos
- 3. Algoritmos de búsqueda y su complejidad
- 4. Optimización de algoritmos

Búsqueda en array de enteros

- Si un array no está ordenado, no hay mejor algoritmo que una búsqueda lineal del primero al último

```
static final int NINGUNO = -1;  // marca de no encontrado
static int busquedaLineal(int buscado, int[] a) {
        for (int p = 0; p < a.length; p++) {
                if (buscado == a[p]) return p;
        }
        return NINGUNO;
}
```

- Complejidad *O(n)*
  - Tiempo medio en encontrar un elemento que está (suponiendo que no están repetidos) *n/2*
  - Peor caso (no se encuentra en el array) *n*

Búsqueda en array de String

- Igual que la búsqueda de enteros, *excepto*
  - Si utilizamos *str1==str2* buscamos objetos
  - Si utilizamos *str1.equals(str2)* buscamos valores

```
static final int NINGUNO= -1;  // marca de no encontrado

static int busquedaLineal (String buscado, String[] a) {
        for (int p = 0; p < a.length; p++) {
                if (buscado.equals(a[p])) return p;
        }
        return NINGUNO;
}
```

Búsqueda en array de *Object*

- Igual que la búsqueda en array de String.
  - Si las instancias de *Object* tienen definido el método equals, podemos buscar por valor
  - Si utilizamos ==, buscamos un objeto en concreto

- En Java no podemos hacer hacer un algoritmo genérico para cualquier tipo de datos. Lo mas genérico es:
  - `static int busquedaLineal (Object buscado, Object[] a)`
  - El compilador convierte automáticamente `int` a su envoltorio (`Integer`). Pero no los array (`int[]` a `Integer[]`)

Java sobrecargar *equals*

- La clase `Object` es la superclase de todas las clases e incluye la declaración:

  `public boolean equals(Object obj)`
- Su implementación por defecto es una comparación por referencia (el objeto y el parámetro son el mismo objeto)
- Con frecuencia queremos una comparación por valor
  - La clase `String` sobrecarga este método y compara los strings por valor
- Nuestras clases pueden sobrecargar *equals* y comparar con nuestras reglas
  - Una clase `Punto` puede sobrecargar *equals* y comparar dos posiciones con criterios propios
    - Por ejemplo: la distancia entre los dos puntos es menor que una constante EPSILON

Arrays/Listas ordenadas

- Un array/lista está **ordenado** en orden **ascendente** si no existe un elemento **menor** que cualquier elemento anterior del array/lista. Este es el orden por **defecto**
- El orden es **descendente** si no existe un elemento **mayor**
- Un **array/lista** de *Object* **no se puede ordenar**
  - *Object* no define métodos para determinar cuando un objeto es mayor/menor que otro

El interfaz *Comparable*

- El interfaz *java.lang.Comparable* incluye el método:
  public int compareTo(Object that)
  - Este método devuelve:
    - < 0 si el objeto es menor que *that*
    - 0 si el objeto y *that* son iguales
    - >0 si el objeto es mayor que *that*
- Clases que son comparables, y pueden formar parte de arrays/listas ordenables, implementan el interface *Comparable*

```
class MiClase implements Comparable {
   public int compareTo(Object that) {...}}
```

- Algunas clases que lo implementan: *Date, Integer, Boolean, String*

Garantías de las implementaciones de *Comparable*

- Hay que garantizar:
  - x.compareTo(y) y y.compareTo(x) o devuelven los dos 0, o uno devuelve positivo y el otro negativo
  - x.compareTo(y) levanta excepción si y solo si también la levanta y.compareTo(x)
  - La regla es transitiva:
    - (x.compareTo(y)>0 && y.compareTo(z)>0) implica x.compareTo(z)>0
    - Si x.compareTo(y)==0, entonces x.compareTo(z) y y.compareTo(z) devuelven valores del mismo signo
- Consistencia entre compareTo y equals
  - x.compareTo(y)==0 implica x.equals(y)

Búsqueda binaria

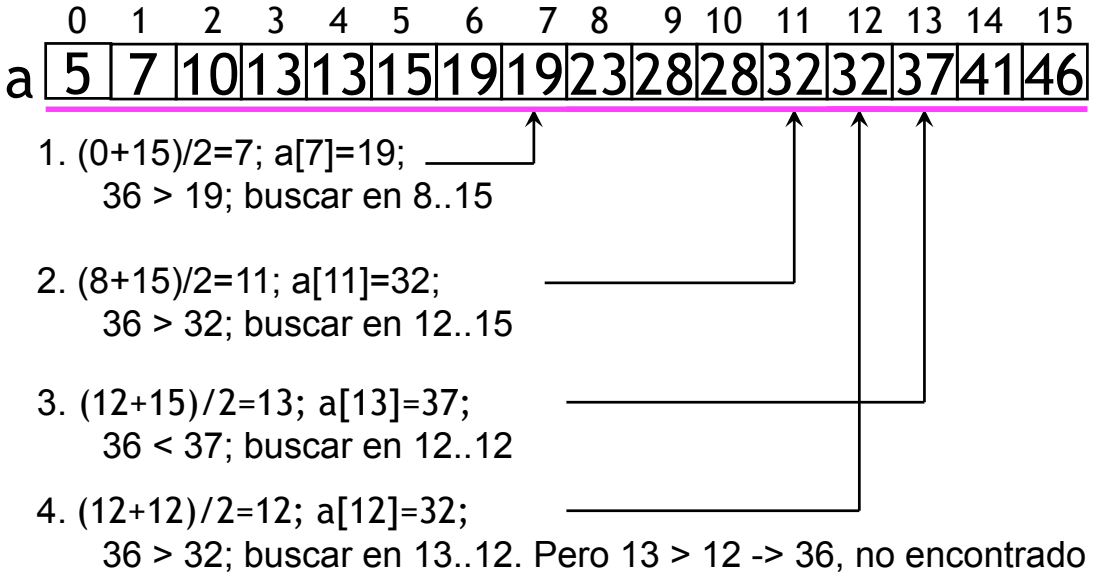
- Para saber si un elemento en a[izq..der] es igual a buscado (donde a ésta ordenado):

```
int i = izq; int d = der;
while (i <= d) {
   int m=(i + d)/2;
   if (buscado == a[m]) return m;
   else if (buscado < a[m]) d = m-1;
   else /*buscado > a[m]*/  i = m+1;
}
return NINGUNO;
```

```
int i = izq; int d = der;
while (i <= d) {
   int m=(i + d)/2;
   int cmp =
      buscado.compareTo(a[m]);
   if (cmp == 0) return m;
   else if (cmp < 0) d = m-1;
   else /* (cmp > 0) */ i = m+1;
}
return NINGUNO;
```

Ejemplo búsqueda binaria

Buscar *36* en el siguiente array



Búsqueda binaria es *O(log n)*

- El algoritmo va partiendo el subarray en 2 y sigue buscando en la mitad correspondiente
- Repetimos la partición en 2 hasta que encontramos el valor o llegamos a un subarray de tamaño 1
- Si empezamos con un array de tamaño n, repetimos el bucle $log_2 n$ veces
- La complejidad es *O(log n)*
  - Para un array de tamaño 1000, el orden de la búsqueda binaria es 100 veces el de la búsqueda lineal ($2^{10} \sim= 1000$)

Búsquedas en Java

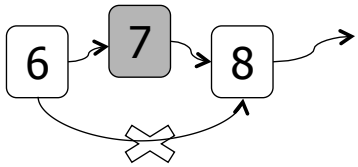
- Las clases *Array**s*** y *Collection**s*** incluyen los métodos (sobrecargados):
  - *sort*: ordena el array/lista con un algoritmo de orden *n*log n*
  - *binarySearch*: es una implementación de búsqueda binaria (*O(log n)*) pero los elementos deben estar ordenador con *sort*
    - *Collections*: si la implementación de la lista no soporta acceso aleatorio hay que buscar la posición media mediante bucle y el algoritmo es *O(n)*
- O(log *n*) ⊂ O(*n*) ⊂ O(*n* log *n*)
- Si hacemos 1 búsqueda:
  - Compensa ordenar y buscar binario o es mejor una búsqueda lineal?
- Y si hacemos *n* búsquedas?
- Varios interfaces (por ejemplo *List*, *Set*) y clases (por ejemplo *ArrayList*, *LinkedHashSet*) incluyen el método *contains*; cada uno tiene implementaciones diferentes

Búsqueda lineal vs binaria

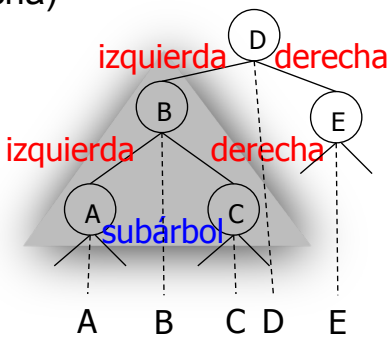
- La búsqueda **lineal** tiene una **complejidad lineal**
- La búsqueda **binaria** tiene una **complejidad logarítmica**
- Para grandes arrays/listas la búsqueda **binaria** es mas **eficiente**
  - Pero primero **tenemos que ordenar** el array/lista
  - **Insertar** en array no ordenado es *O(1)* y en ordenado *O(n)* (hay que hacer hueco). En la lista depende de implementación
- El análisis debe decidir **cuando compensa ordenar** las colecciones e insertar ordenado, y que algoritmos de búsqueda debemos utilizar

Arboles de búsqueda binarios

- Los **árboles de búsqueda binarios** (BST, Binary Search Tree) buscan combinar:
  - **Flexibilidad de la inserción** de las listas enlazadas
  - **Eficiencia de la búsqueda** en arrays ordenados



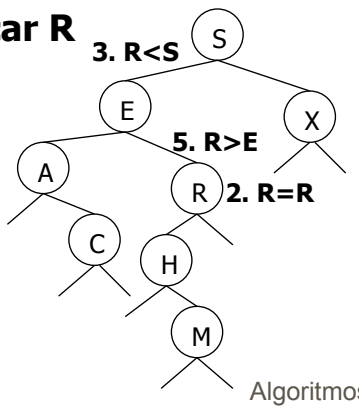
- Un BST es un árbol en el que los nodos tienen una **clave** *Comparable* y **dos enlaces** (izquierda y derecha) a otros nodos, y
  - todos los **nodos** del subárbol al que referencia **izquierda** tienen una **clave menor**
  - Todos los **nodos** del subárbol de **derecha** tienen **clave mayor**
- Ejemplo aplicación: tablas símbolos compiladores



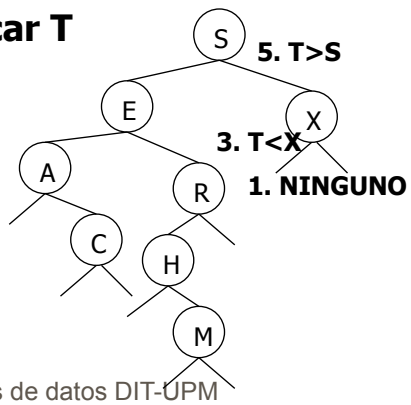
Búsqueda en BST

- Un **algoritmo** recursivo de **búsqueda** en BST
  1. Si el árbol está vacío falla la búsqueda
  2. Si la clave del nodo es la buscada -> encontrado
  3. Si la clave es menor que la del nodo
  4. recursivo-> buscar en el subárbol izquierdo
  5. Si la clave es mayor que la del nodo
  6. recursivo-> buscar en el subárbol derecho

Buscar R



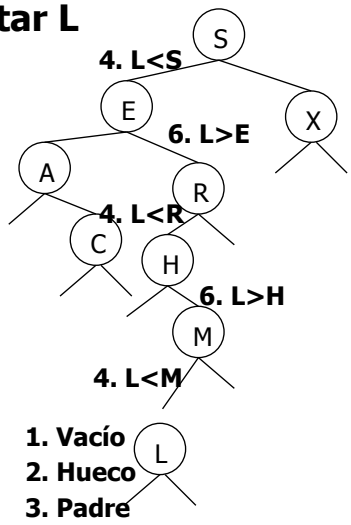
Buscar T



Insertar en BST

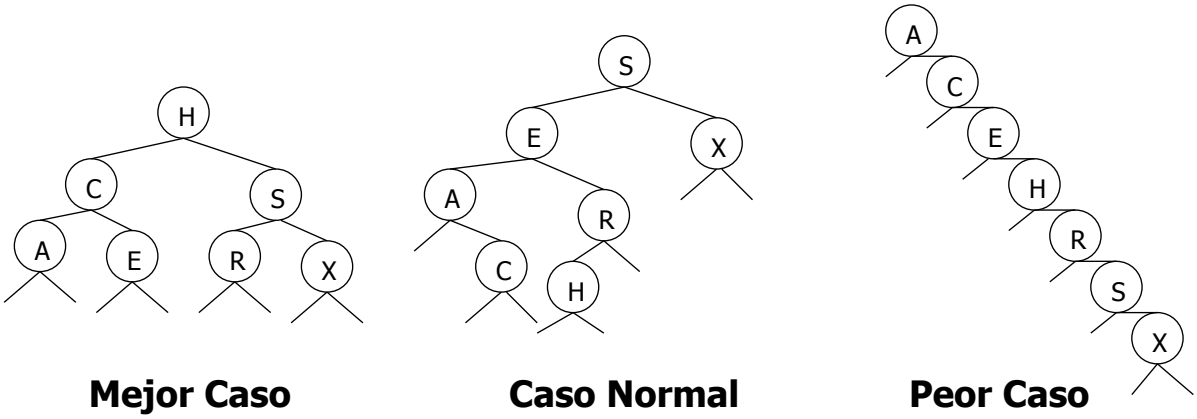
- Insertar es parecido a buscar, pero remplazamos el enlace vacío
  1. Si el árbol está vacío
  2. este es el hueco
  3. actualizar el padre
  4. Si la clave es menor que la del nodo
  5. recursivo-> buscar en el subárbol izquierdo
  6. Si la clave es mayor que la del nodo
  7. recursivo-> buscar en el subárbol derecho

Insertar L



Análisis de algoritmos de BST

- Los **tiempos** dependen de la **forma** de los árboles, que dependen de cómo evolucionen las **inserciones**
  - *Mejor caso*: el árbol tiene menor profundidad. *O(log n)*
  - *Peor caso*: mayor profundidad. *O(n)*



Mejor Caso **Caso Normal** **Peor Caso**

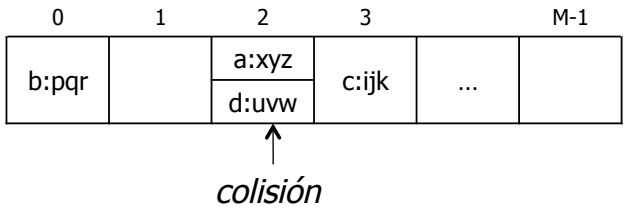
Balanceado de BST

- **Objetivo**: garantizar una profundidad máxima de *O(log n)*
- **Condición ideal** para que un árbol esté **equilibrado**:
  - Número de **elementos de izquierda y derecha** de cada **subárbol** tengan una **diferencia máxima de 1**
  - Es **difícil** de mantener (algoritmos complejos de inserción y borrado)
- Algunos tipos de árboles BST fijan **condiciones mas flexibles de implementar**, que garantizan una profundidad máxima cercana o igual a *log n*

Tablas Hash

- Las ***tablas hash*** son estructuras que permiten acceder a un *valor* a partir de una *clave*
  - Si la clave fuese un entero natural podríamos emplear directamente un array
  - Una tabla **hash transforma las claves en índices**
- Una tabla hash se puede implementar mediante un **array y una función** que transforma *claves* en índices
- Algoritmos de búsqueda basados en hash tienen **dos partes** fundamentales:
  - *Función hash*
  - *Resolución colisiones*

Clave	*Hash*	*Valor*
a	2	xyz
b	0	pqr
c	3	ijk
d	2	uvw



Función Hash

- Si tenemos un array para almacenar ***M*** pares de *clave-valor* necesitamos una función que transforme un parámetro del tipo de la ***clave*** en un entero en el rango ***0..M-1***
- **Objetivo** de una **función hash**: realizar una *distribución uniforme* entre claves y valores hash (*0..M-1*)
  - Java: la clase *Object* incluye el método
    public int hashCode()
  - Podemos redefinirla pero debe ser consistente con *equals*.
    x.equals(y) -> x.hashCode() == y.hashCode()
  - Las clases que implementan la función hash utilizan *hashCode* para transformarlo al rango *0..M-1.* Por ejemplo
    hashCode()%M

Ejemplos Función Hash en Java

- ***String***:
  - Utiliza *hash* para no tener que recalcular
  - Suma todas las letras, pero emplea 31 (primo) para evitar que M y *hashCode()* tengan factores comunes y el *hashCode()%M* cree colisiones

```
int h = hash; int len = count;
if (h == 0 && len > 0) {
   int off = offset;
   for (int i = 0; i < len; i++) {
      h = 31*h + value[off++];
   }
   hash = h;
}
return h;
```

- ***Double:***
  - Hay que generar un número que tenga en cuenta tanto los 32 bits bajos como los altos
  - Obtiene la representación en 64 bit (doubleToLongBits)
  - Hace el OR exclusivo (^) de los 32 bits altos y los bajos
  - *x >>> 32* desplaza los 32 bits altos a los bajos. Los altos quedan 0

```
long bits =
   doubleToLongBits(value);
return (int)(bits ^ (bits >>> 32));
```

Resolver colisiones

- Una forma simple es que la tabla sea una **tabla de listas** cuyos **elementos** sea los pares ***clave-valor***
- Para buscar un elemento hacemos hash y una búsqueda lineal en su lista
- Si la **función hash** distribuye las claves de manera **uniforme**, el tamaño de las listas es *N/M* donde N es el número de claves. La búsqueda es del orden *N/M*

Clave	*Hash*	*Valor*
a	2	xyz
b	0	pqr
c	3	ijk
d	2	uvw