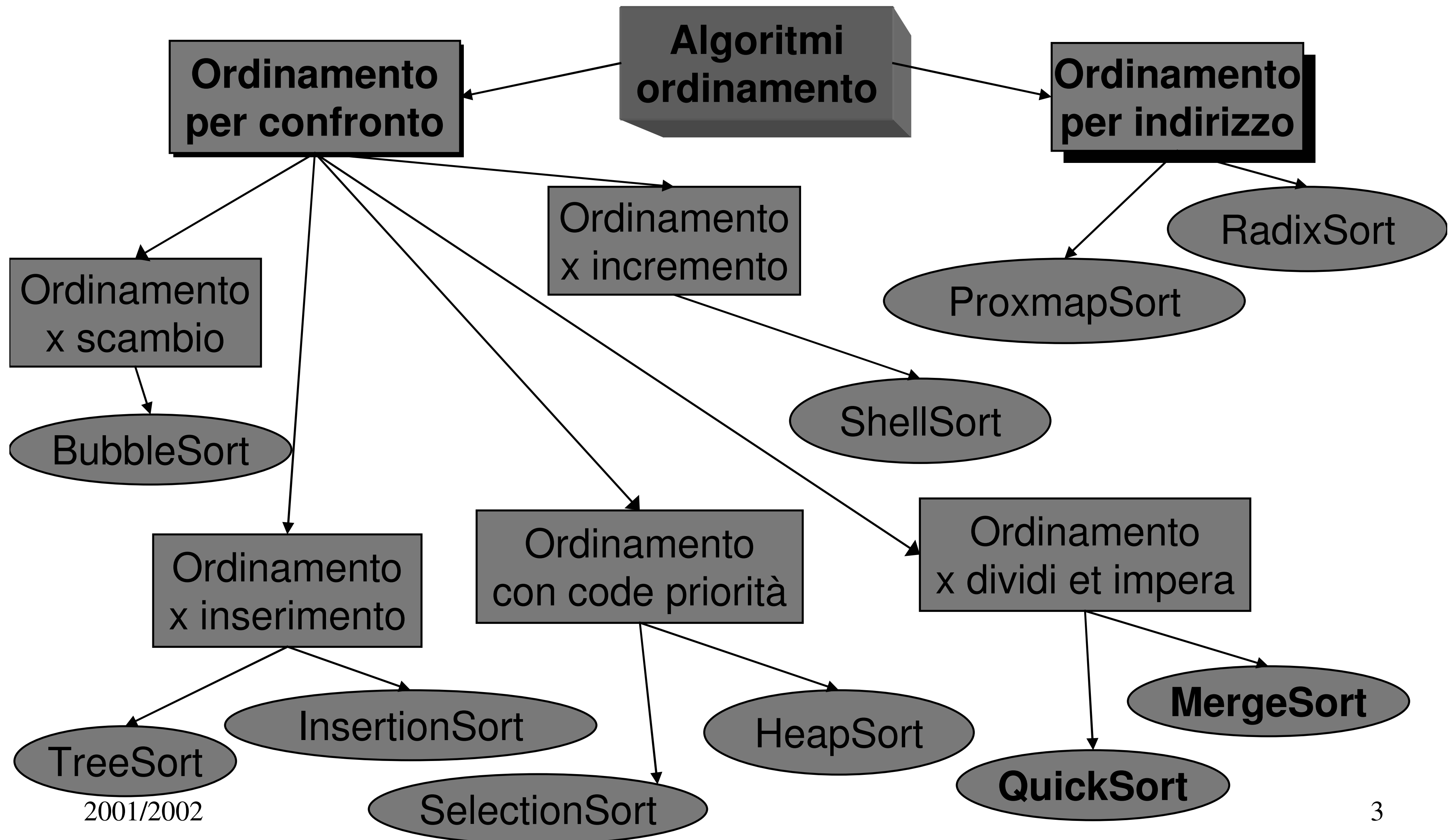
Algoritmi di ordinamento (II parte)

E3: sommario

- Studio di due implementazioni di algoritmi avanzati
  - Algoritmo ordinamento veloce (QuickSort)
  - Algoritmo per fusione (MergeSort)
- Metodi offerti in java.util
- Confronto tempi di esecuzione tra:
  - InsertionSort, ShellSort, QuickSort e MergeSort

E3: Classificazione



E3: QuickSort

- Idea algoritmo
  1. dato un array A[0..n],
  2. si sceglie un elemento (*pivot),*
  3. si divide l’array in due parti, una contiene gli elementi <= pivot, l’altra contiene elementi >= pivot,
  4. si iterano i passi 2-3 sulle due parti fino a quando le parti hanno dimensione = 1 (impera)
  5. si restituisce l’array A (ordinamento crescente)

E3: QuickSort - (richiamo)

- Osservazioni
  - algoritmo di tipo divide et impera (ricorsivo)
  - migliori prestazioni quando chiamate ricorsive agiscono su sottoarray di pari lunghezza (circa)
  - scelta pivot cruciale; diverse strategie:
    - + semplice: scegliere primo elemento array
    - + cauta: scegliere elemento posto al centro array
    - + indipendente: scegliere un elemento casuale array
    - + precisa: scegliere l'elemento mediana array

E3: QuickSort - implementazione

- Implementazione versione cauta

```
/** QuickSort(Comparable a[]) esegue un ordinamento crescente
  * sul vettore a di comparable * /
public static void QuickSort(Comparable a[]) {
  quickSort(a, 0, a.length-1);
}
private static void quickSort(Comparable a[], int lower, int upper) {
  if (lower<upper) {
    int pivotIndex = prudentPartition(a, lower, upper);
    quickSort(a, lower, pivotIndex);
    quickSort(a, pivotIndex+1, upper);
  }  }
```

E3: QuickSort - implementazione

```
private static int prudentPartition(Comparable a[], int lower, int upper) {

 //scelgo come pivot l'elemento che sta nel mezzo tra lower e upper.
 //e lo sposto al primo posto dell’array
  int half = (lower+upper)/2;
  Comparable pivot = a[half];
  a[half] = a[lower];
  a[lower] = pivot;

  //inizio procedura di partizionamento
  Comparable temp = null;
```

E3: QuickSort -implementazione

```
while (true) {
  while (a[upper].compareTo(pivot)>0)//cerco da dx elemento <= pivot
    upper--;
  while (a[lower].compareTo(pivot)<0)//cerco da sx elemento >= pivot
    lower++;
  if (lower<upper) { //se li ho trovati, scambio e aggiorno indici
    temp = a[lower];
    a[lower++] = a[upper];
    a[upper--] = temp;
  } else //altrimenti array partizionato, restituisco indice confine partizione
    return upper;
 }
}
```

E3: QuickSort - esempio

- Esempio di partizione
  - input A =

0	1	2	3	4
D	**M**	**H**	**K**	**B**

pivot = H

0	1	2	3	4
H	**M**	**D**	**K**	**B**

  - 1 ciclo
    - prima if (lower<upper) =

0	1	2	3	4
H	**M**	**D**	**K**	**B**

    - dopo if =

0	1	2	3	4
B	**M**	**D**	**K**	**H**

  - 2 ciclo
    - prima if (lower<upper) =

0	1	2	3	4
B	**M**	**D**	**K**	**H**

    - dopo if =

0	1	2	3	4
B	**D**	**M**	**K**	**H**

  –

output =>

0	1	2	3	4
B	**D**	**M**	**K**	**H**

E3: QuickSort - (richiamo)

- Complessità
  - caso migliore: quando pivot è mediana --> O(n lg n)
  - caso peggiore: quando pivot è min o max --> $O(n^2)$
  - caso medio: più vicino al caso migliore --> O(n lg n)
- Osservazioni
  - può essere reso più efficiente sostituendo ricorsione con iterazione
  - se n<30 QuickSort è più lento di InsertionSort (Cook&Kim1980)
  - in media QuickSort è migliore degli altri algoritmi di ordinamento di un fattore 2

E3: MergeSort - (richiamo)

- Idea algoritmo:
  - rendere regolare il partizionamento dell'array e 'ordinare' durante fusione dei sottoarray ordinati

```
MergeSort(A[0..n]) {
  if (n>2)
    MergeSort(A[0..n/2])
    MergeSort(A[n/2+1..n])
    merge(A[0..n/2], A[n/2+1..n] in una lista
                    ordinata A[0..n])
  altrimenti ritorna;
}
```

E3: MergeSort - (richiamo)

– fondere due sottoarray in una lista ordinata è il task principale

```
merge(A, B, C) {
  inizializza opportunamente i1, i2, i3
  while sia B che C contengono elementi
   if B[i2] < C[i3]
      A[i1++] = B[i2++]
   else
      A[i1++] = C[i3++]
  inserisci in A tutti elementi rimasti di B o C
}
```

E3: MergeSort - (richiamo)

– ATTENZIONE!
  - B e C sono sottoarray di A, merge deve lavorare su A
  - è necessario quindi un array temporaneo

```
merge(A, first, last) { // first e last sono gli estremi su cui agire
  mid = (first + last) / 2, i1 = 0, i2 = first, i3 = mid +1;
  while i2<mid && i3<last //sia B che C contengono elementi
       if A[i2] < A[i3]
          temp[i1++] = A[i2++]
       else
          temp[i1++] = A[i3++]
   inserisci in temp tutti elementi rimasti di A (i2..mid o i3..last)
   copia temp in A[first..last]
}
```

E3: MergeSort - (richiamo)

- Osservazioni
  - l'array ausiliario penalizza quando n >> 0
  - diversi modi per gestire array ausiliario… ma è opportuno allocare lo spazio solo una volta!

E3: MergeSort - implementazione

```
/** MergeSort
* l’array temporaneo è allocato solo una volta in due parti di dimensioni
* n/2… in modo da poter servire TUTTE le chiamate ricorsive!
*/
public static void MergeSort(Comparable a[]) {
    int half = a.length/2+1;
    Object[]  leftA  = new Object[half], //Non è possibile def. Comparable
                  rightA = new Object[half];//perché?

    mergeSort(a, 0, a.length-1, leftA, rightA);
}
```

E3: MergeSort - implementazione

```
/** mergeSort esegue la partizione di a in due parti in modo ricorsivo
* e poi fonde le due parti in ordine in a*/
private static void mergeSort(Comparable a[], int lo, int hi,
                              Object[] leftA, Object[] rightA) {
    if (lo >= hi) //base della ricorsione
      return;

    int middle = (lo + hi) / 2;

    //partiziona a in due parti e ordina loro ricorsivamente
    mergeSort(a, lo,        middle, leftA, rightA);
    mergeSort(a, middle+1, hi,      leftA, rightA);
```

E3: MergeSort - implementazione

```
//fondi le due parti ordinate
int left = 0, right = 0, sizeL = middle - lo + 1, sizeR = hi - middle;
System.arraycopy(a, lo,            leftA,  0, sizeL);
System.arraycopy(a, middle+1,rightA,0, sizeR);

 while ( (left < sizeL) && (right < sizeR) )
     a[lo++] = ( ((Comparable)leftA[left]).compareTo(rightA[right]) <= 0 )
               ? (Comparable)leftA[left++] : (Comparable)rightA[right++];
 while (left < sizeL)
     a[lo++] = (Comparable)leftA[left++];
 while (right < sizeR)
     a[lo++] = (Comparable)rightA[right++];
}
```

Arraycopy

- public static void a**rraycopy**(Object src, int srcPos, Object dest, int destPos, int length)
- src - the source array.
- srcPos - starting position in the source
- array.dest - the destination array.
- destPos - starting position in the destination data.length - the number of array elements to be copied.

E3: MergeSort - esempio

– input A =

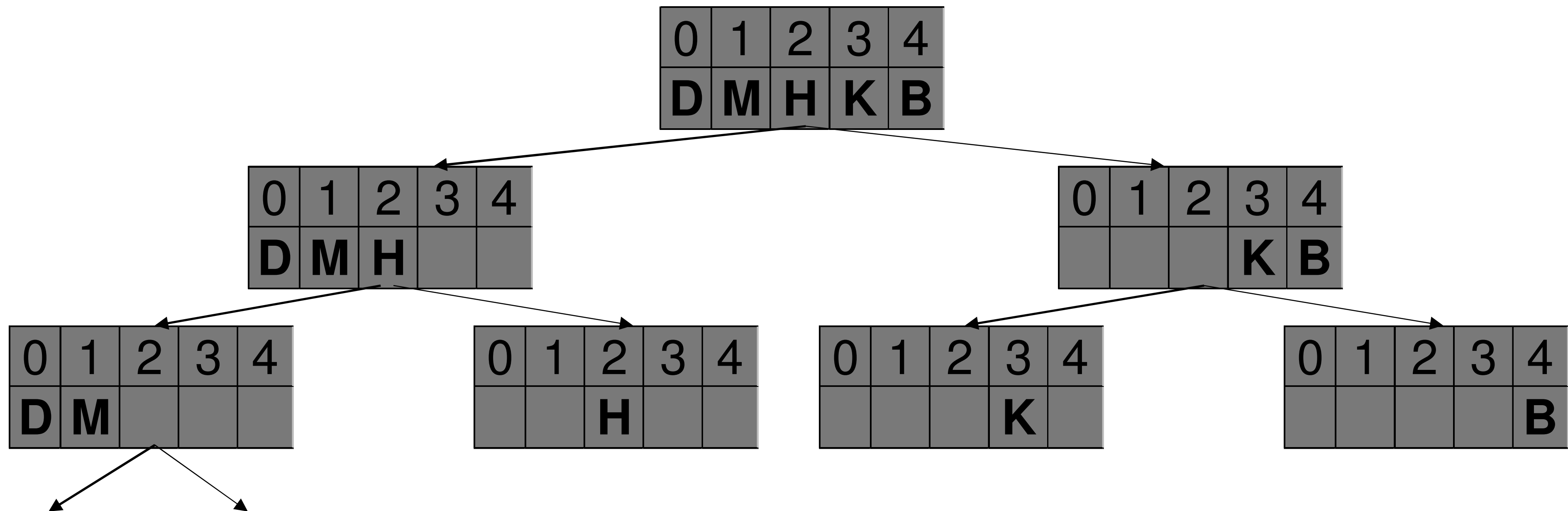
0	1	2	3	4
D	**M**	**H**	**K**	**B**

– leftA =

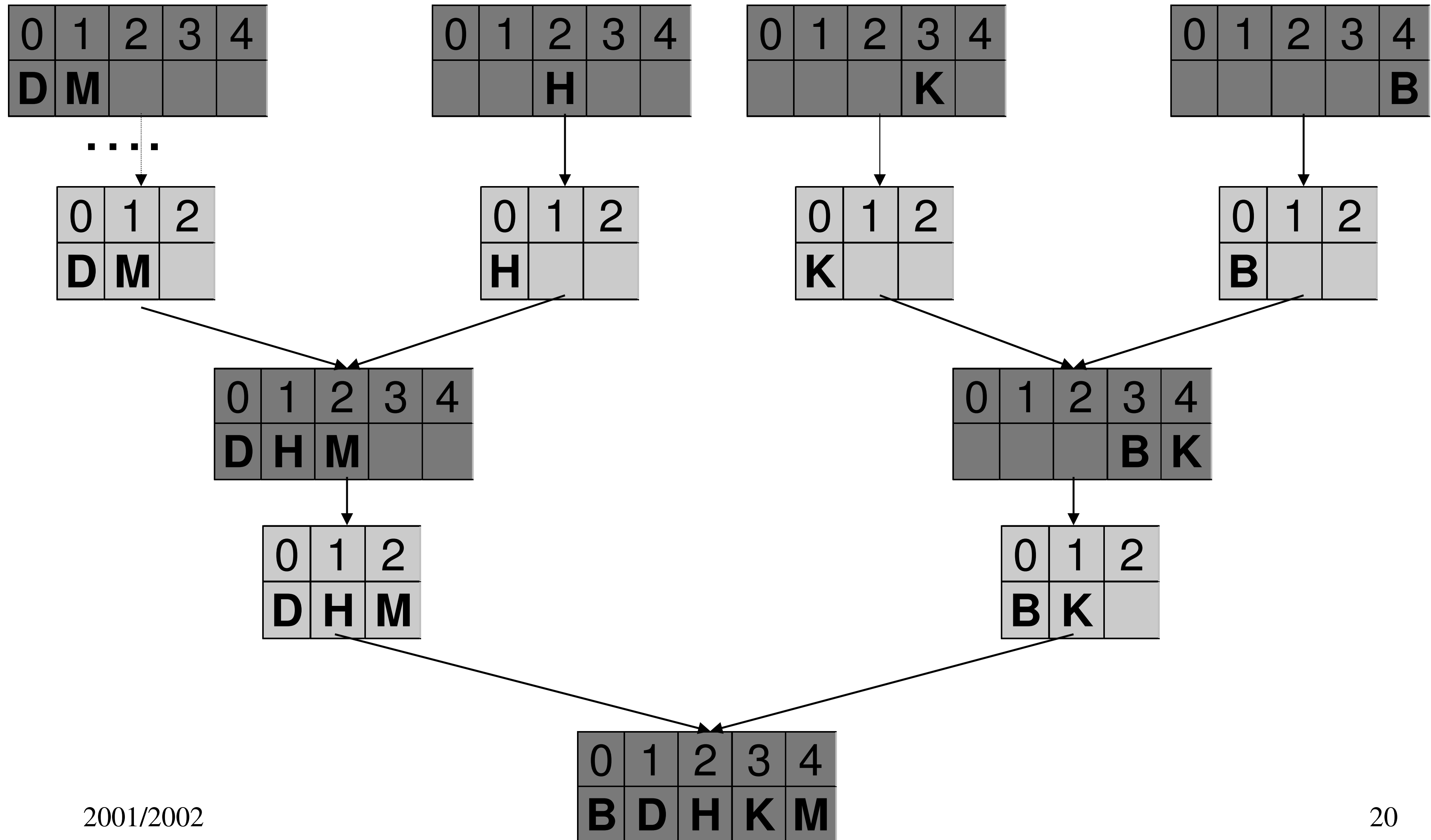
0	1	2

rightA =

0	1	2



E3: MergeSort - esempio



E3: MergeSort - (richiamo)

- Complessità
  - caso migliore: quando A è in ordine o elementi B precedono elementi C --> O(n lg n)
  - caso peggiore: quando ultimo elemento di B precede solo ultimo elemento di C --> O(n lg n)
- Osservazioni
  - può essere reso più efficiente sostituendo ricorsione con iterazione o applicando InsertionSort quando array ha dimensioni piccole
  - la richiesta di spazio aggiuntivo penalizza questo algoritmo

E3: ordinamento in java.util

- java.util offre 2 insiemi di metodi di ordinamento:
  - per array
  - per liste
- Per array: la classe Arrays contiene diversi metodi basati sul QuickSort/MergeSort:
  - public static void sort(int[] a); //char,double,…
  - public static void sort(Object[] a);
  - ...

E3: ordinamento in java.util

– public static void sort(Object[] a. Comparator c);
  - permette di ordinare array in base a criteri diversi (c) e di avere elementi nulli nell’array
  - utilizza interface java.util.Comparator che ha un solo metodo:

    public int compare(Object o1,Object o2)

    /**Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. Vedi manuale per le altre specifiche*/

E3: ordinamento in java.util

- Per liste: la classe Collections contiene 2 metodi basati sul MergeSort:
  - public static void sort(List l);
  - public static void sort(List l, Comparator comp);

E3: confronto tempi calcolo

- Si vuole confrontare i tempi di esecuzione delle implementazioni costruite:
  - si genera un array casuale
  - si applicano tutti gli algoritmi sul medesimo array rilevando i tempi di calcolo
- Attenzione!
  - i tempi di calcolo possono essere falsati dall’iterazione del programma con sistema operativo

E3: confronto tempi calcolo

- Una possibile tecnica per ovviare al problema di tempi falsati:
  - ripetere l’ordinamento col medesimo algoritmo più volte e fare la media dei tempi di esecuzione

E3: confronto tempi calcolo

- Esempio: estratto della procedura…

```
...
Comparable[] a = new Integer[n]; Object[] b = new Integer[n];
for (i=0; i<n; i++) {          //due copie del vettore da ordinare!
    a[i] = b[i] = new Integer(rnd.nextInt(n*2));
}
for (i=0,tempo=0; i<nCicli; i++) {//Calcolo il tempo per insertionSort
    inizio = System.currentTimeMillis();
    Sort.insertionSort(a);
    fine = System.currentTimeMillis();
    tempo += (fine-inizio);
    System.arraycopy(b, 0, a, 0, n);
}
System.out.println( "Tempo (ms) insertionSort: " + (tempo/nCicli) );
```

E3: confronto tempi calcolo

- Compito esercitazione
  - confrontare i tempi di calcolo di InsertionSort, ShellSort, QuickSort, MergeSort e Array.sort per n = 1000, 2000, 4000, 8000
  - stampare la tabella con i coefficienti di crescita del tempo per ogni algoritmo
    - Esempio: tempi in ms su Celeron 500MHz con Linux 2.4

N	InsertionS.	R.I.	Shell	R.I.	Merge	R.I.	Quick	R.I.	Arrays.sort	R.I.
1000	34		4		7		2		3	
2000	116	3,4	6	1,5	14	2,0	5	2,5	5	1,7
4000	453	3,9	13	2,2	30	2,1	10	2,0	13	2,6
8000	2135	4,7	34	2,6	65	2,2	23	2,3	25	1,9