Ordenamientos

Algoritmos basados en comparaciones

Verónica E. Arriola-Rios

Facultad de Ciencias, UNAM

3 de agosto de 2021

Bubble Sort

El ordenamiento burbuja

- Par por par, se verifica que los datos estén en el orden correcto.
- En cada iteración se garantiza que el elemento más a la derecha se encuentra en la posición correcta.
- Tiene complejidad $\mathcal{O}(n^2)$

10	9	8	7	6	5	4	3	2	1
9	8	7	6	5	4	3	2	1	10
8	7	6	5	4	3	2	1	9	10
7	6	5	4	3	2	1	8	9	10
6	5	4	3	2	1	7	8	9	10
5	4	3	2	1	6	7	8	9	10
4	3	2	1	5	6	7	8	9	10
3	2	1	4	5	6	7	8	9	10
2	1	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

4	8	3	6	1	5
4	3	6	1	5	8
3	4	1	5	6	8
3	1	4	5	6	8
1	3	4	5	6	8

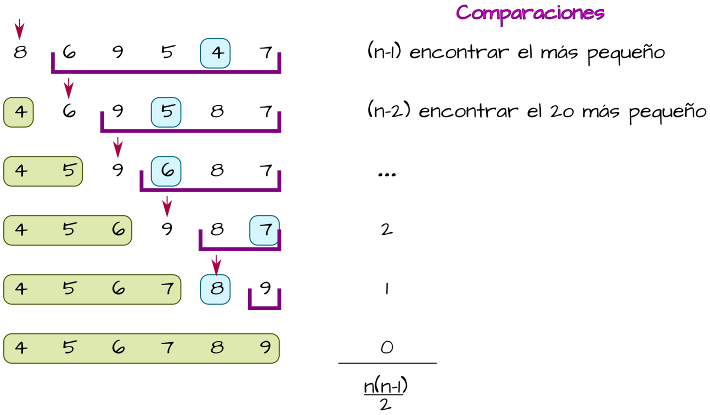
Código 1: Ordenamiento burbuja

```java
public static <C extends Comparable<? super C>>
       void bubleSort(C[] a) {
  boolean swapped = false;
  for(int i = 0; i < a.length - 1; i++) {
    swapped = false;
    for(int j = 1; j < a.length - i; j++) {
      if(a[j-1].compareTo(a[j]) > 0) {
        swap(a, j-1,j);
        swapped = true;
      }
    }
    if(!swapped) return;
  }
}
```

Selection Sort

Selecciona y acomoda

- Busca el elemento más pequeño en el subarreglo restante y lo coloca en su posición.



Código 2: Ordenamiento por selección

```
public static void selectionSort(Comparable[] a){
  int primero;
  for (int i = 0; i < a.length - 1; i++) {
    primero = i;   // primer elemento en el arreglo
    for(int j = i + 1; j < a.length; j++){
      // elemento más pequeño entre 1 e i.
      if(a[j].compareTo(a[primero]) < 0) primero = j;
    }
    swap(a, primero, i);
  }
}
```

Insertion Sort

Acomoda en su lugar

- Inserta cada número en su posición correcta.

	Comparaciones	Peor caso
8 6 9 5 4 7	1 insertar en lista de un elemento	
6 8 9 5 4 7	1	2
6 8 9 5 4 7	3	3
5 6 8 9 4 7	4	4
4 5 6 8 9 7	3	(n-1)
4 5 6 7 8 9	0	$\frac{n(n-1)}{2}$

Divide y vencerás

Temas

4. Divide y vencerás
   - Quick Sort
   - Merge Sort

Alrededor del pivote

- Se elige algún número como pivote. Se pueden usar los criterios siguientes:
  - El elemento en la posición `[0]`
  - El elemento en la posición `[a.lenght-1]`
  - El número en la posición media del (sub)arreglo. (Estadísticamente es el mejor)
- Los menores que el pivote se acomodan a su izquierda,
- los mayores a su derecha.
- Después de cada iteración el pivote queda en su lugar.
- Repertir con cada segmento del arreglo hasta que tenga un sólo elemento.
- Complejidad en promedio:
  - En **promedio** se realizan $\log(n)$ pasos recursivos.
  - Cada elemento del arreglo es revisado durante el paso recursivo en alguna de las subdivisiones.
  - La complejidad total es $\mathcal{O}(n \log n)$
- **Peor** caso: $\mathcal{O}(n^2)$

Ejemplo

Pivote = 6	2 3 4 5 1 6 8 9 10 7
Pivote = 4	2 3 1 4 5 6 8 9 10 7
Pivote = 3	2 1 3 4 5 6 8 9 10 7
Pivote = 2	1 2 3 4 5 6 8 9 10 7
Pivote = 9	1 2 3 4 5 6 8 7 9 10
Pivote = 8	1 2 3 4 5 6 7 8 9 10

Código 3: Quick Sort

```
public static void quickSort(Comparable[] a) {
  quickSort(a, 0, a.length - 1);
}
```

Código 4: Función recursiva

```java
private static void quickSort(Comparable[] a,
                              int left, int right){
  int pivotIndex = (right + left)/2;
  Comparable pivot = a[pivotIndex];
  swap(a, pivotIndex, right); // Pivote al final
  int i = left;
  int j = right;
  while(i < j) {
    while(i<=right && a[i].compareTo(pivot) < 0) i++;
    while(j>=left && a[j].compareTo(pivot) >= 0) j--;
    if(i < j) swap(a, i, j);
  }
  swap(a, right, i);

  // Llamada recursiva
  if(i - left >= 2) quickSort(a, left, i - 1);
  if(right - i >= 2) quickSort(a, i + 1, right);
}
```

Temas

4. Divide y vencerás
   - Quick Sort
   - Merge Sort

Divide, ordena y mezcla

- Parte el arreglo en mitades recursivamente hasta que sólo haya un elemento. Obsérvese que un arreglo de un elemento siempre está ordenado.
- Mezcla las mismas mitades, pero insertando los elementos en orden.
- Su complejidad siempre es $\mathcal{O}(n \log n)$ porque divide al arreglo exactamente $\log_2 n$ veces y mezclar los subarreglos ordenados en cada paso toma n operaciones.

Ejemplo

```
9 10 | 8 | 7 6 | 5 4 | 3 | 2 1
8 9 10 | 7 6 | 5 4 | 3 | 2 1
8 9 10 | 6 7 | 5 4 | 3 | 2 1
6 7 8 9 10 | 5 4 | 3 | 2 1
6 7 8 9 10 | 4 5 | 3 | 2 1
6 7 8 9 10 | 3 4 5 | 2 1
6 7 8 9 10 | 3 4 5 | 1 2
6 7 8 9 10 | 1 2 3 4 5
1 2 3 4 5 6 7 8 9 10
```

Código 5: Merge sort

```java
public static void mergeSort(Comparable[] a) {
  msortDivide(a,0,a.length-1);
}
```

Código 6: Función recursiva

```java
private static void msortDivide(Comparable[] a,
        int left, int right) {
  if(left < right) {
    int middle = (left + right) / 2;
    msortDivide(a, left, middle);
    msortDivide(a, middle + 1, right);
    merge(a, left, middle, right);
  }
}
```

Código 7: Mezcla

```java
// middle : índice derecho subarreglo 1
private static void merge(Comparable[] a,
          int left, int middle, int right) {
  int di = left;
  int dd = middle + 1;
  int i = 0;
  Comparable[] temp = new Comparable[right-left+1];
  while(di <= middle && dd <= right) {
    if(a[di].compareTo(a[dd]) < 0)
      temp[i++] = a[di++];
    else
      temp[i++] = a[dd++];
  }
  while(di <= middle) temp[i++] = a[di++];
  while(dd <= right) temp[i++] = a[dd++];
  for(int j = left, jj = 0; j <= right; j++, jj++)
    a[j] = temp[jj];
}
```

Heap Sort

Temas

5 Heap Sort
- Árboles n-arios en arreglos
- Montículo máximo
- Insertar
- Remover
- Cola de prioridades
- Ordena

Árboles n-arios en arreglos

Dado que el número de hijos en un árbol n-ario es fijo:

- El número total de nodos m en un árbol n-ario con altura l es:

$$m = 1 + n + n^2 + n^3 + ... + n^l = \frac{n^{l+1} - 1}{n - 1} \tag{1}$$

Nivel	1	2		4				8							
Valor	50	25	75	10	39	∅	100	1	∅	29	42	∅	∅	∅	120
Índice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

- Es posible determinar la posición del i-ésimo descendiente, dada la posición de su padre.

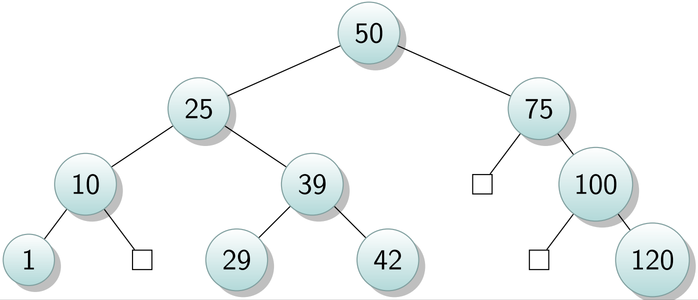
$$\mathrm{hijo}_j(i) = ni + (j + 1) \tag{2}$$

donde i es la posición del nodo; j, el índice de su hijo, contando desde cero de izquierda a derecha y n, el grado del árbol.

- Y la posición del padre dada la posición del hijo:

$$\mathrm{padre}(i) = \lfloor \frac{i-1}{n} \rfloor \tag{3}$$

Árbol binario en arreglo



50	25	75	10	39	∅	100	1	∅	29	42	∅	∅	∅	120
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

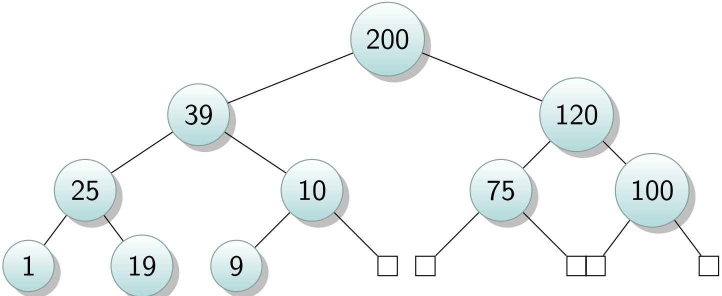
$$\text{hijoIzquierdo}(i) \rightarrow 2i + 1 \qquad \text{padre}(i) \rightarrow \lfloor (i-1)/2 \rfloor$$
$$\text{hijoDerecho}(i) \rightarrow 2(i+1)$$

Temas

5. Heap Sort
   - Árboles n-arios en arreglos
   - Montículo máximo
   - Insertar
   - Remover
   - Cola de prioridades
   - Ordena

Montículo máximo

- Los datos en los hijos de un nodo son menores que el dato en su padre.



200	39	120	25	10	75	100	1	19	9	∅	∅	∅	∅	∅
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

$\mathrm{hijoIzquierdo}(i) \rightarrow 2i+1 \qquad \mathrm{padre}(i) \rightarrow \lfloor (i-1)/2 \rfloor$

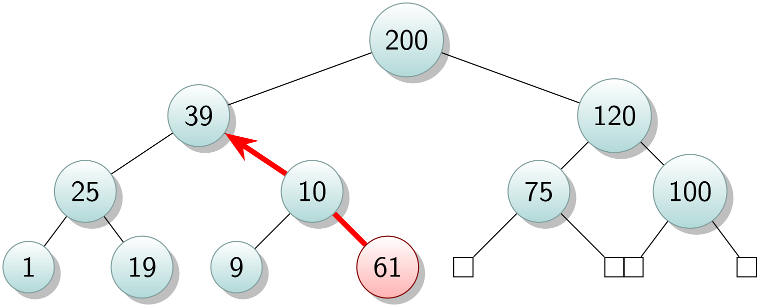
$\mathrm{hijoDerecho}(i) \rightarrow 2(i+1)$

Temas

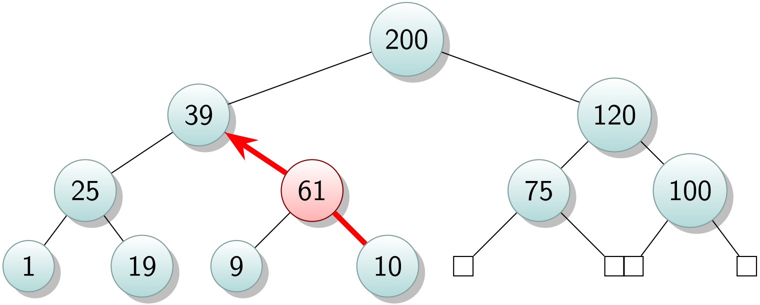
5 Heap Sort
- Árboles n-arios en arreglos
- Montículo máximo
- Insertar
- Remover
- Cola de prioridades
- Ordena

Inserta

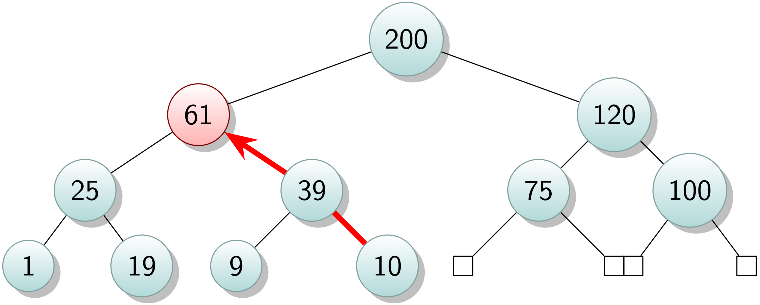
- Se inserta siempre en el siguiente nodo vacío.
- El dato se intercambia con su padre hasta que su padre no sea más chico que él o llegue a la raíz.



Inserta



Inserta

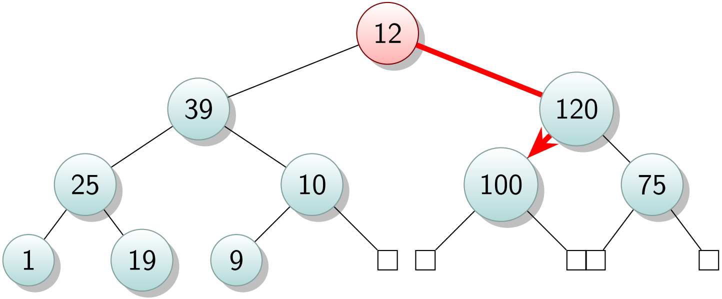


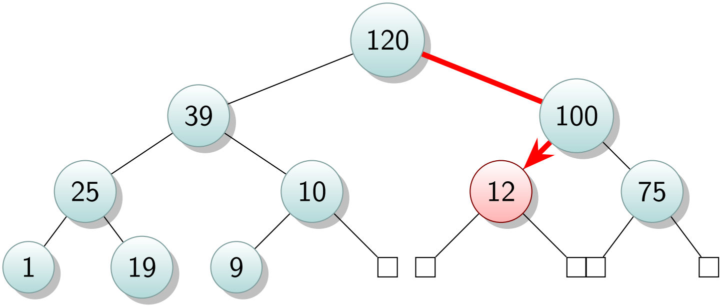
Temas

5. Heap Sort
   - Árboles n-arios en arreglos
   - Montículo máximo
   - Insertar
   - Remover
   - Cola de prioridades
   - Ordena

Maxmonticuliza (*Heapify*)

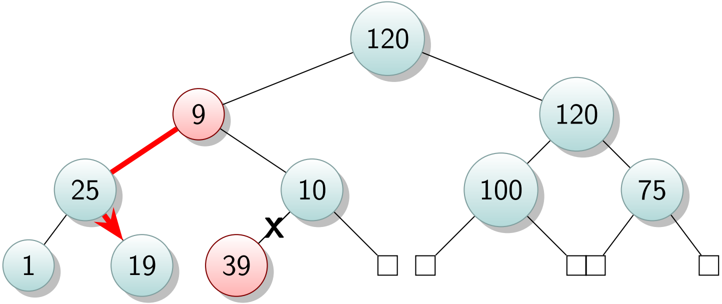
- Si los dos subárboles de un nodo son montículos máximos, pero el dato en el nodo es menor que alguno de sus hijos:
  - Intercambiar al dato del nodo con el mayor de sus hijos.
  - Repetir recursivamente hasta que el dato intercambiado sea mayor que sus dos hijos.




120
39
100
25
10
12
75
1
19
9

Remueve

- Intercambiar al nodo que se quiere remover con el último nodo (última hoja).
- Quitar la hoja.
- Aplicar `maxMonticuliza` desde la hoja que subió de posición.



Temas

5 Heap Sort
- Árboles n-arios en arreglos
- Montículo máximo
- Insertar
- Remover
- Cola de prioridades
- Ordena

Cola de prioridades

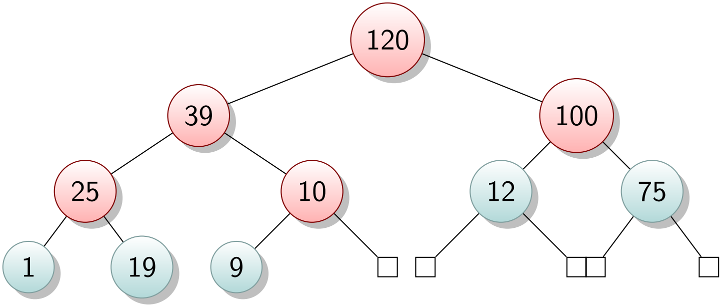
- Un montículo máximo puede ser utilizado para implementar una cola de prioridades donde el **primer elemento en salir** es el que tiene la **prioridad más alta**.
- Para ello basta con remover el dato en la raíz del montículo.
- La única desventaja es que el algoritmo de remoción **no garantiza** que, cuando haya empates, se atienda primero a quienes llegaron primero.

Temas

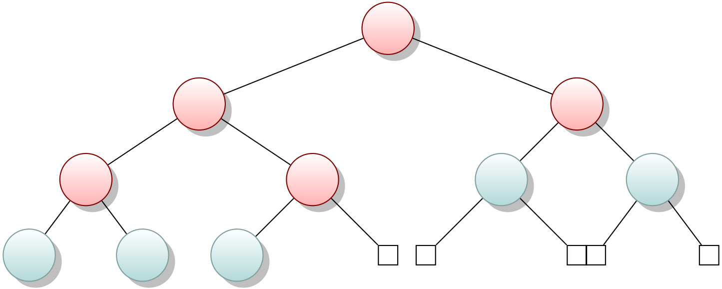
5. Heap Sort
   - Árboles n-arios en arreglos
   - Montículo máximo
   - Insertar
   - Remover
   - Cola de prioridades
   - Ordena

Construye montículo máximo

- Es posible ordenar los elementos de un árbol casi completo para que formen un montículo máximo si se repite la operación `maxMonticuliza` de abajo hacia arriba sobre la mitad de los nodos.
- Su complejidad es $\mathcal{O}(n)$.



Ejercicio: construye

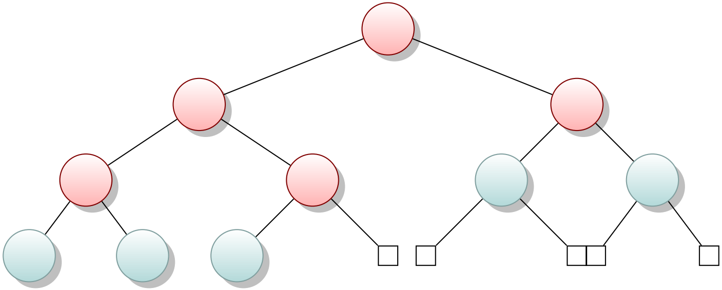


25	10	120	39	200	75	100	9	19	1	∅	∅	∅	∅	∅
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Heap Sort

- Se construye un montículo máximo. $\mathcal{O}(n)$
- Para cada dato, desde el último hacia uno antes de la raíz: $\mathcal{O}(n)$
  - "Remover la raíz" (valor más grande).
  - Dejar al nodo "removido" pero marcar que el heap termina un nodo antes (este nodo ha quedado en su lugar). $\mathcal{O}(\log n)$
- Su complejidad es $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$

Ejercicio: ordena



25	10	120	39	200	75	100	9	19	1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Bibliografía I

- Cormen, Thomas H. y col. (2009). *Introduction to Algorithms*. 3rd. The MIT Press.
- Preiss, Bruno (1999). *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. John Wiley & Sons.

Licencia