

Ordenamientos

Algoritmos basados en comparaciones

Verónica E. Arriola-Rios

Facultad de Ciencias, UNAM

14 de noviembre de 2023



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: Ordenamiento burbuja

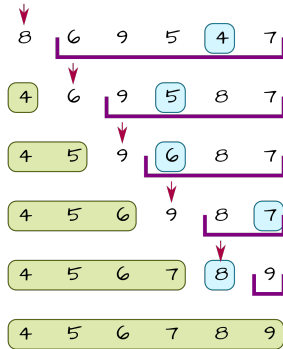
```
1 public static void BubleSort<C>(C[] a) where C : IComparable<C>
2 {
3     bool yaEstá;
4     for(int i = 0; i < a.Length - 1; i++)
5     {
6         yaEstá = true;
7         for(int j = 0; j < a.Length - 1 - i; j++)
8         {
9             if(a[j].CompareTo(a[j+1]) > 0)
10             {
11                 Swap(a, j, j+1);
12                 yaEstá = false;
13             }
14         }
15         if (yaEstá) break;
16     }
17 }
```

Selection Sort

- 1 Bubble Sort
- 2 Selection Sort
- 3 Insertion Sort
- 4 Divide y vencerás
- 5 Heap Sort

Selecciona y acomoda

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



Comparaciones

(n-1) encontrar el más pequeño

(n-2) encontrar el 2o más pequeño

...

2

1

0

$$\frac{n(n-1)}{2}$$

Código: Ordenamiento por selección

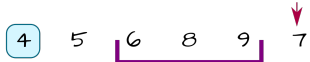
```
1 public static void SelectionSort(IComparable[] a)
2 {
3     for(int i = 0; i < a.Length; i++)
4     {
5         C min = a[i];
6         int minIndex = i;
7         for(int j = i+1; j < a.Length; j++)
8         {
9             if(a[j].CompareTo(min) < 0) {
10                 min = a[j];
11                 minIndex = j;
12             }
13         }
14         Swap(a, i, minIndex);
15     }
16 }
```

Insertion Sort

- 1 Bubble Sort
- 2 Selection Sort
- 3 Insertion Sort
- 4 Divide y vencerás
- 5 Heap Sort

Acomoda en su lugar

- Inserta cada número en su posición correcta.

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

Divide y vencerás

- 1 Bubble Sort
- 2 Selection Sort
- 3 Insertion Sort
- 4 Divide y vencerás
- 5 Heap 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.length-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.
- Repetir 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: Quick Sort

```
1 public static void QuickSort(IComparable[] a) {  
2     QuickSort(a, 0, a.Length - 1);  
3 }
```

Código: Función recursiva

```
1 private static void QuickSort(Comparable[] a, int left, int right)
2 {
3     int pivotIndex = (right + left)/2;
4     Comparable pivot = a[pivotIndex];
5     IOrdenador<C>.Swap(a, pivotIndex, right); // Pivote al final
6     int i = left;
7     int j = right;
8     while (i < j)
9     {
10         while (i <= right && a[i].CompareTo(pivot) < 0) i++;
11         while (j >= left && a[j].CompareTo(pivot) >= 0) j--;
12         if (i < j) IOrdenador<C>.Swap(a, i, j);
13     }
14     Swap(a, right, i);
15
16     // Llamada recursiva
17     if (i - left >= 2) QuickSort(a, left, i - 1);
18     if (right - i >= 2) QuickSort(a, i + 1, right);
19 }
```

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: Merge sort

```
1 public static void MergeSort(IComparable[] a) {  
2     MSortDivide(a, 0, a.Length - 1);  
3 }
```

Código: Función recursiva

```
1 private static void MSortDivide(Comparable[] a,  
2     int left, int right) {  
3     if (left < right)  
4     {  
5         int middle = (left + right) / 2;  
6         MSortDivide(a, left, middle);  
7         MSortDivide(a, middle + 1, right);  
8         Merge(a, left, middle, right);  
9     }  
10 }
```

Código: Mezcla

```
1 // middle : índice derecho subarreglo 1
2 private static void Merge(Comparable[] a,
3     int left, int middle, int right) {
4     int di = left;
5     int dd = middle + 1;
6     int i = 0;
7     int nuevoTam = right - left + 1;
8     C[] temp = new C[nuevoTam];
9     Array.Copy(a, temp, right - left + 1);
10    while (di <= middle && dd <= right)
11    {
12        if (a[di].CompareTo(a[dd]) < 0) temp[i++] = a[di++];
13        else temp[i++] = a[dd++];
14    }
15    while (di <= middle) temp[i++] = a[di++];
16    while (dd <= right) temp[i++] = a[dd++];
17    for (int j = left, jj = 0; j <= right; j++, jj++) a[j] = temp[jj];
18 }
```

Heap Sort

- 1 Bubble Sort
- 2 Selection Sort
- 3 Insertion Sort
- 4 Divide y vencerás
- 5 **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:

$$m = 1 + n + n^2 + n^3 + \dots + n^l = \frac{n^{l+1} - 1}{n - 1} \quad (1)$$

1	2		4				8							
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

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

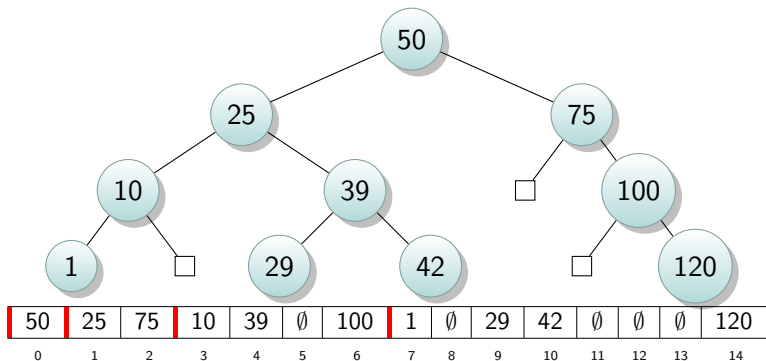
$$\text{hijo}_j(i) = ni + (j + 1) \quad (2)$$

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:

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

Árbol binario en arreglo



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

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

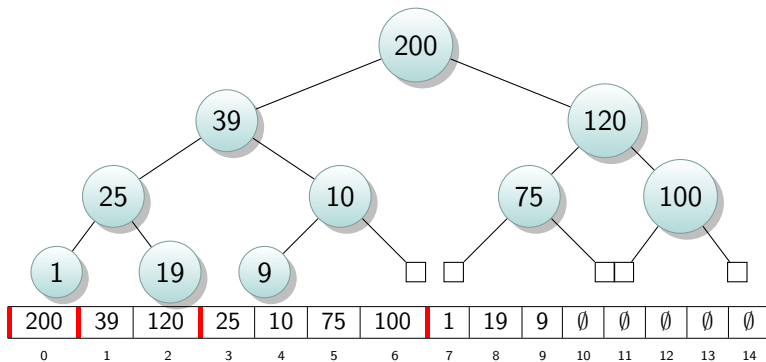
Temas

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.



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

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

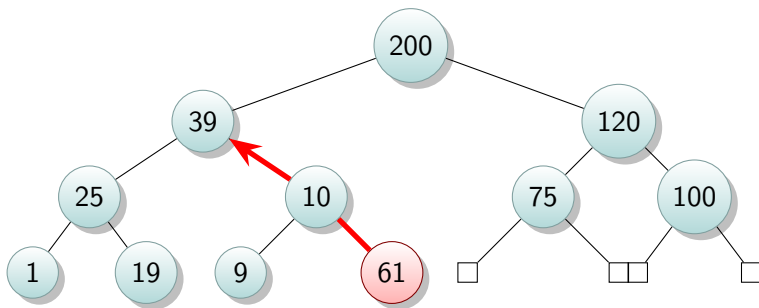
Temas

5 Heap Sort

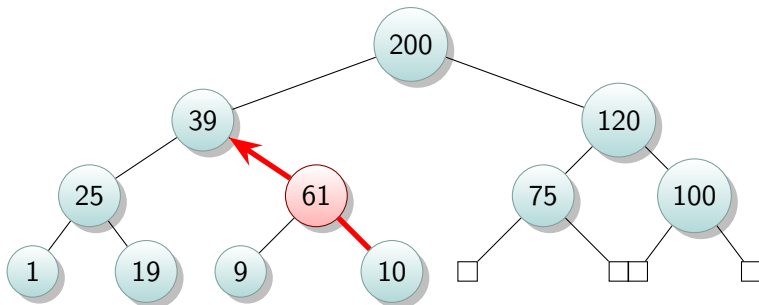
- Árboles n-arios en arreglos
- Montículo máximo
- **Insertar**
- Remover
- Cola de prioridades
- Ordena

Inserta

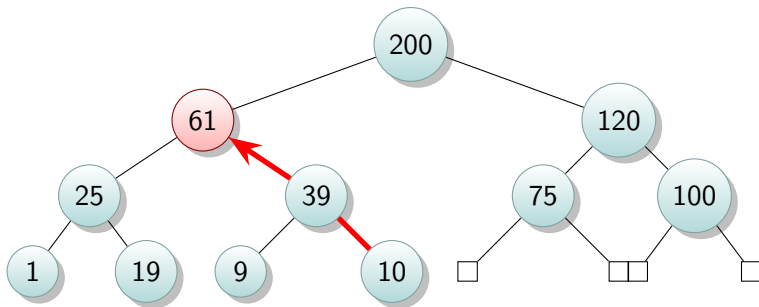
- Se inserta siempre en el siguiente nodo vacío.
- El dato se intercambia con su padre hasta que su padre no sea más chico que él o llegue a la raíz.



Inserta



Inserta



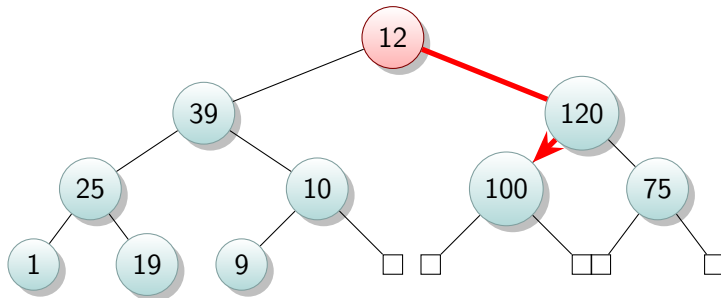
Temas

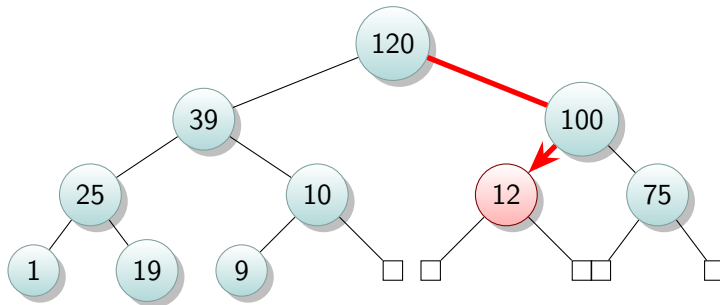
5 Heap Sort

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

Maxmonticuliza (*Heapify*)

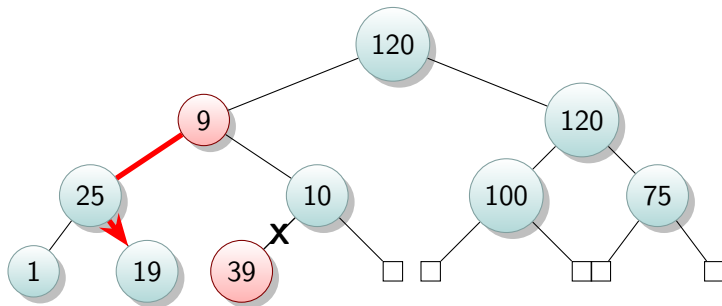
- Si los dos subárboles de un nodo son montículos máximos, pero el dato en el nodo es menor que alguno de sus hijos:
 - Intercambiar al dato del nodo con el mayor de sus hijos.
 - Repetir recursivamente hasta que el dato intercambiado sea mayor que sus dos hijos.





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.

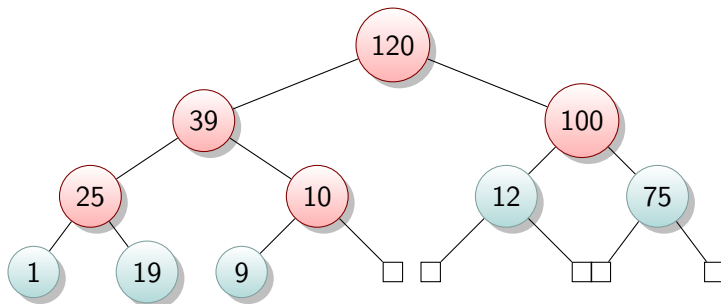
Temas

5 Heap Sort

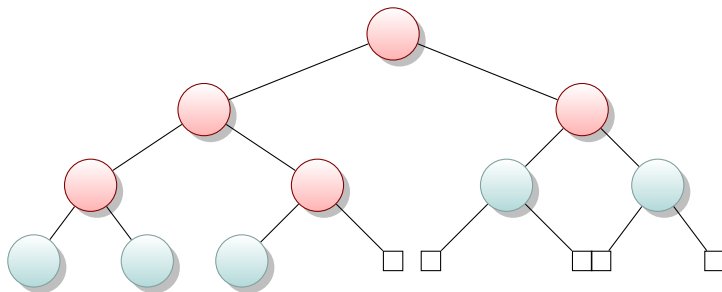
- Árboles n-arios en arreglos
- Montículo máximo
- Insertar
- Remover
- Cola de prioridades
- Ordena

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 $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$

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

Creative Commons
Atribución-No Comercial-Compartir Igual

