Algoritmos de Engenharia

Ordenação—Introdução (Caps. 6, 7 e 8)

Heapsort

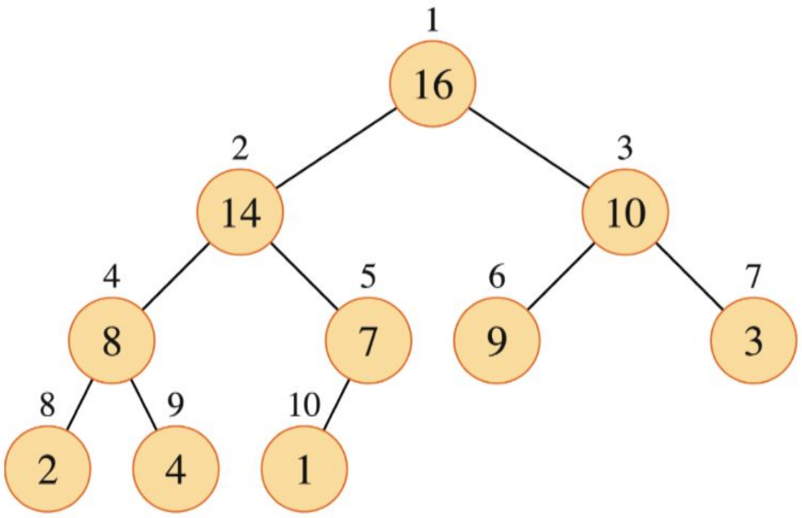
- $O(n \lg n)$ worst case—like merge sort.
- Sorts in place—like insertion sort.
- Combines the best of both algorithms.

To understand heapsort, we'll cover heaps and heap operations, and then we'll take a look at priority queues.

Heap

A heap (***not*** garbage-collected storage) is a nearly complete binary tree.

- ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
- ***Height*** of heap = height of root = $\Theta(\lg n)$.
- ***Example:*** Of a max-heap in array with $heap - size = 10$.



Heap

A heap can be stored as an array A.

- Root of tree is $A[1]$.
- Parent of $A[i] = A[\lfloor i/2 \rfloor]$.
- Left child of $A[i] = A[2i]$.
- Right child of $A[i] = A[2i + 1]$.

PARENT(i)
  return $\lfloor i/2 \rfloor$

LEFT(i)
  return $2i$

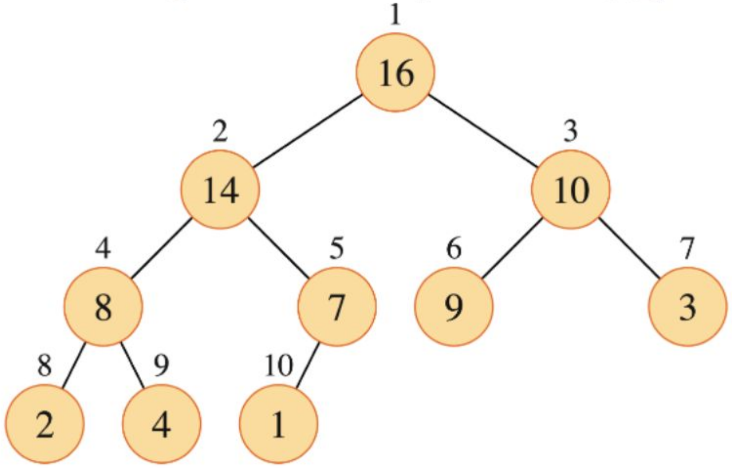
RIGHT(i)
  return $2i + 1$

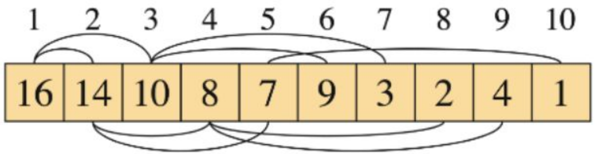
- Attribute $A.\textit{heap-size}$ says how many elements are stored in A. Only the elements in $A[1 : A.\textit{heap-size}]$ are in the heap.

Heap

Of a max-heap in array with *heap-size* $= 10$.

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes i, excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.





Heap

MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

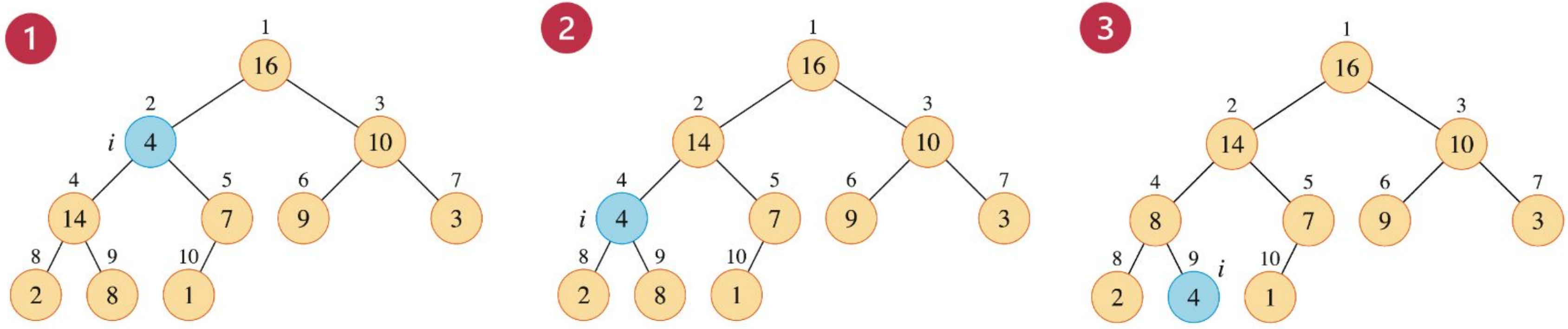
- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume that left and right subtrees of i are max-heaps. (No violations of maxheap property within the left and right subtrees. The only violation within the subtree rooted at i could be between i and its children.)
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

Heap

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Heap

Run MAX-HEAPIFY on the following heap example.



Time: $O(\lg n)$.

Heap

The following procedure, given an unordered array $A[1:n]$, will produce a max-heap of the n elements in A.

```
BUILD-MAX-HEAP(A, n)
A.heap-size = n
for i = ⌊n/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Heap

```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

Priority Queue

- Maintains a dynamic set S of elements.
- Each set element has a ***key***—an associated value.
- Max-priority queue supports dynamic-set operations:
  - INSERT(S, x, k): inserts element x with key k into set S.
  - MAXIMUM(S): returns element of S with largest key.
  - EXTRACT-MAX(S): removes and returns element of S with largest key.
  - INCREASE-KEY(S, x, k): increases value of element x's key to k. Assumes $k \geq x$'s current key value.
- Example max-priority queue application: schedule jobs on shared computer. Scheduler adds new jobs to run by calling INSERT and runs the job with the highest priority among those pending by calling EXTRACT-MAX.

Quick Sort

```
QUICKSORT(A, p, r)
if p < r
    // Partition the subarray around the pivot, which ends up in A[q].
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)   // recursively sort the low side
    QUICKSORT(A, q + 1, r)   // recursively sort the high side
```

Initial call is QUICKSORT$(A, 1, n)$.

Sorting in linear time

Counting sort

Depends on a *key assumption*: numbers to be sorted are integers in $\{0, 1, \ldots, k\}$.

Input: $A[1:n]$, where $A[j] \in \{0, 1, \ldots, k\}$ for $j = 1, 2, \ldots, n$. Array A and values n and k are given as parameters.

Output: $B[1:n]$, sorted.

Auxiliary storage: $C[0:k]$

Sorting in linear time

```
COUNTING-SORT(A, n, k)
let B[1 : n] and C[0 : k] be new arrays
for i = 0 to k
    C[i] = 0
for j = 1 to n
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
// Copy A to B, starting from the end of A.
for j = n downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1    // to handle duplicate values
return B
```

Sorting in linear time

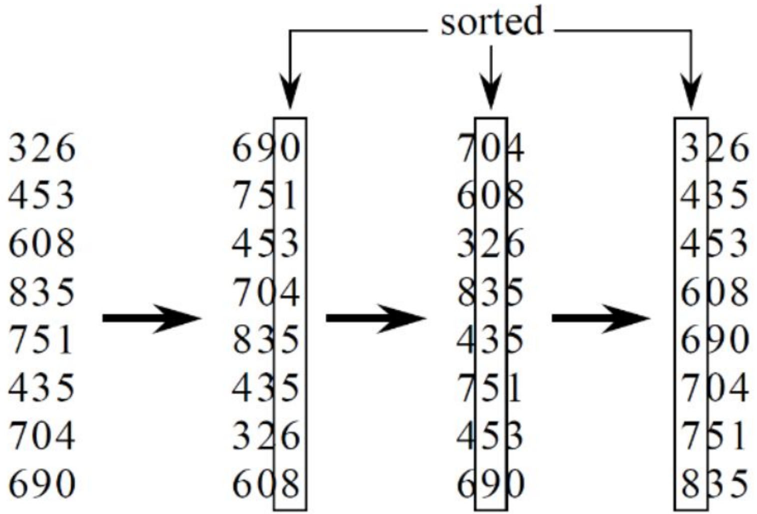
RADIX-SORT

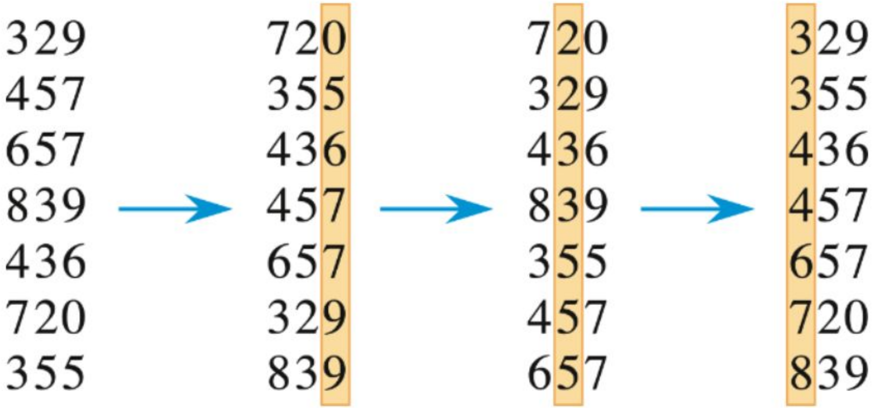
Key idea: Sort least significant digits first.

To sort d digits:

```
RADIX-SORT(A, n, d)
for i = 1 to d
        use a stable sort to sort array A[1 : n] on digit i
```

Sorting in linear time





Correctness

- Induction on number of passes (i in pseudocode).
- Assume digits $1, 2, \ldots, i-1$ are sorted.

Sorting in linear time

BUCKET SORT

Assumes that the input is generated by a random process that distributes elements uniformly and independently over $[0, 1)$.

Idea

- Divide $[0, 1)$ into n equal-sized *buckets*.
- Distribute the n input values into the buckets. *[Can implement the buckets with linked lists; see Section 10.2.]*
- Sort each bucket.
- Then go through buckets in order, listing elements in each one.

Sorting in linear time

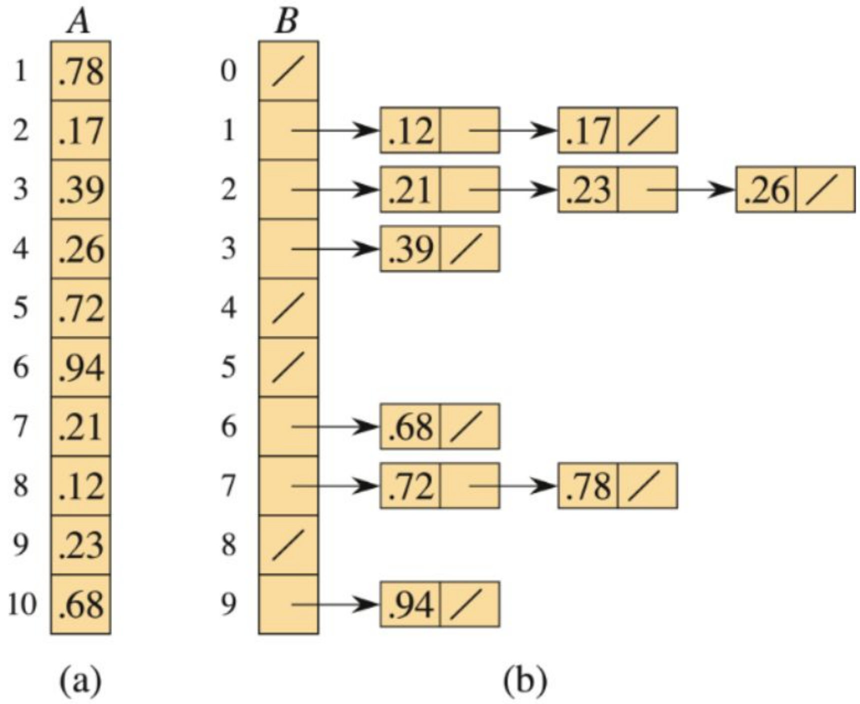
Input: $A[1:n]$, where $0 \le A[i] < 1$ for all i.

Auxiliary array: $B[0:n-1]$ of linked lists, each list initially empty.

BUCKET-SORT(A, n)

let $B[0:n-1]$ be a new array
for $i = 0$ **to** $n-1$
    make $B[i]$ an empty list
for $i = 1$ **to** n
    insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
for $i = 0$ **to** $n-1$
    sort list $B[i]$ with insertion sort
concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order
return the concatenated lists

Sorting in linear time



The buckets are shown after each has been sorted.

Exercícios

1. Implemente as funções da seção 6.5 (Priority queues) do livro do Cormen 4th Ed. em sua linguagem favorita e proponha um exemplo de uso com uma demonstração.
2. Mostre com experimentos numéricos quando suas próprias implementações de Quicksort e do Quicksort aleatório são mais vantajosas, comparando uma com a outra.
3. Mostre com experimentos numéricos quando o Radix-sort com o Count-sort é mais rápido que o Count-sort sozinho. Utilize suas próprias implementações ou alguma implementação existente explicando os resultados.