CME 2001
Data Structures and Algorithms

Zerrin Işık
zerrin@cs.deu.edu.tr

Heapsort

Heaps

- A heap is a complete binary tree such that:
  - It is empty, or
  - Its root contains a search key greater than or equal to the search key in each of its children, and each of its children is also a heap.
- The root contains the item with the largest search key
- ***Height*** of node = # of edges on a longest simple path from the node down to a leaf.
- ***Height*** of heap = height of root = $\Theta(\lg n)$.

Heaps

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 $\lfloor i/2 \rfloor$

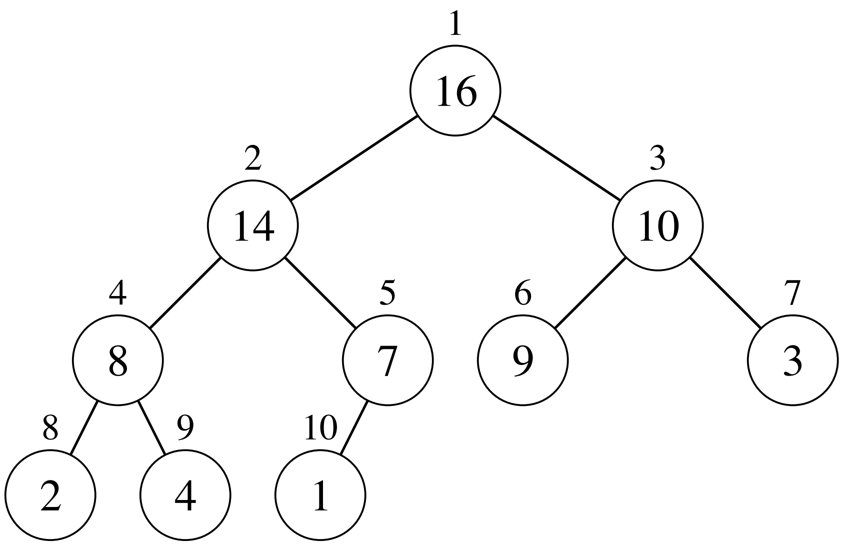
LEFT (i): return $2i$

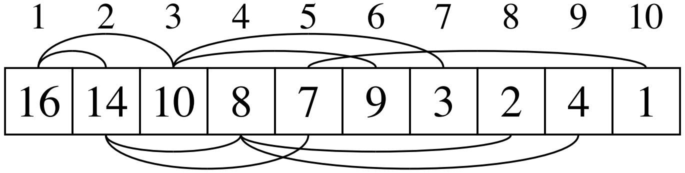
RIGHT (i): return $2i+1$

Heap Property:

- For max-heaps (largest element at root), ***max-heap property:*** for all nodes ***i***, excluding the root, **A**[PARENT(i)] ≥**A**[i].
- For min-heaps (smallest element at root), ***min-heap property:*** for all nodes ***i***, excluding the root, **A**[PARENT(i)] ≤**A**[i].

Max-Heap example





heap-size: # of elements that are already sorted in the heap.

=>heap-size = 10

Max-Heapify

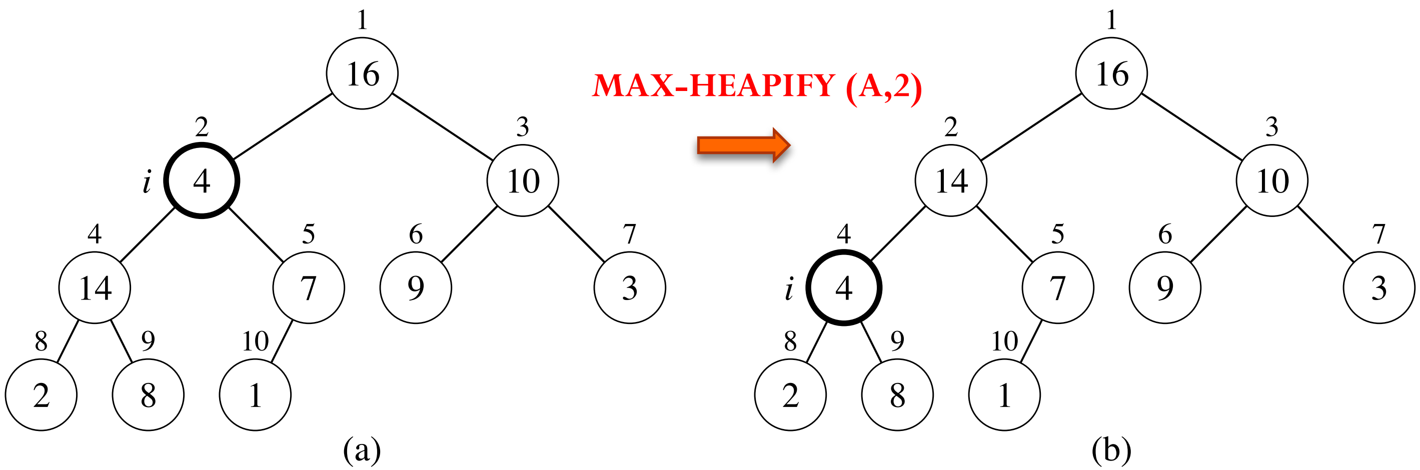
- Used to maintain the max-heap property.

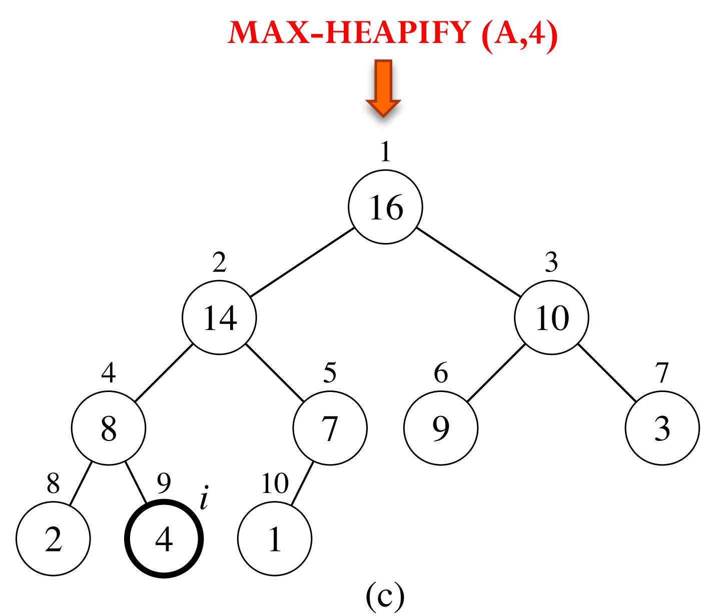
MAX-HEAPIFY(A, i, n)

$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq n$ and $A[l] > A[i]$
  $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
  $largest = r$
if $largest \neq i$
  exchange $A[i]$ with $A[largest]$
  MAX-HEAPIFY$(A, largest, n)$

n: heap-size
LEFT (i): return 2i
RIGHT (i): return 2i+1

Max-Heapify …





- Node 2 violates max-heap property (a).
- Compare node 2 with its children, and then swap it with the larger of the two children (b).
- Continue swapping until the value is properly placed at the root of a subtree that is a max-heap (c).

Max-Heapify Analysis

$\text{MAX-HEAPIFY}(A, i, n)$

$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \le n$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \le n$ and $A[r] > A[largest]$
    $largest = r$
if $largest \ne i$
    exchange $A[i]$ with $A[largest]$
    $\text{MAX-HEAPIFY}(A, largest, n)$

Analysis

- $\Theta(1)$: Fix relations among the A[i], A[LEFT(i)], A[RIGHT(i)]
- Children subtreees have size at most 2n/3

$T(n) \le T(2n/3) + 1$

Apply recurrence: $T(n) = O(\lg n)$

Building a Heap

- Given an unsorted array, build a max-heap

$\text{BUILD-MAX-HEAP}(A, n)$

$\quad$ **for** $i = \lfloor n/2 \rfloor$ **downto** 1

$\qquad \text{MAX-HEAPIFY}(A, i, n)$

n: unsorted array size

- Why does it start from n/2?
  - All elements A[(n/2+1), …, n] are on the leaves

Build-Max-Heap Analysis

Analysis

- Each call of Max-Heapify: O(lgn)
- For loop runs n/2 times: O(n)
- Upper bound: O(n. lgn)

BUILD-MAX-HEAP(A, n)

for $i = \lfloor n/2 \rfloor$ **downto** 1

MAX-HEAPIFY(A, i, n)

Heap Usage

- Heap sort
  - one of the best sorting methods - not quadratic in the worst-case
- Selection algorithms
  - finding the min, max, median, k^{th} element in sublinear time
- Graph algorithms
  - Prim's minimal spanning tree
  - Dijkstra's shortest path

Heap Sort Algorithm

Given an input array:

- Builds a max-heap from the array.
- Start with the root, place the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- "Discard" this last node by decreasing the heap size, and calling *MAX-HEAPIFY* on the new root.
- Repeat this "discarding" process until only one node (the smallest element) remains.

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

Analysis

- *BUILD-MAX-HEAP*: O(n lgn)
- for loop runs (n-1) times
- exchange elements: O(1)
- *MAX-HEAPIFY*: O(lgn)
- **Total time:** O(n lgn)

Priority Queues – A Heap Application

- 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): inserts element x 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. Assume k $\geq$ x's current key value.
- e.g. Max-priority queue application : schedule jobs on shared computer.

Priority Queue Operations

HEAP-MAXIMUM(A)
  return $A[1]$

- Running Time $= \Theta(1)$.

Priority Queue Operations

HEAP-EXTRACT-MAX(A, n)

if $n < 1$
    error "heap underflow"
$max = A[1]$
$A[1] = A[n]$
$n = n - 1$
MAX-HEAPIFY$(A, 1, n)$ // remakes heap
return max

- Running Time = O(lgn).

Priority Queue Operations

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
  error "new key is smaller than current key"
$A[i] = key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  exchange $A[i]$ with $A[\text{PARENT}(i)]$
  $i = \text{PARENT}(i)$

- Running Time = O(lgn).

Priority Queue Operations

MAX-HEAP-INSERT(A, key, n)

$n = n + 1$
$A[n] = -\infty$
HEAP-INCREASE-KEY(A, n, key)

- Running Time = O(lgn).

Quicksort

Quicksort

Quicksort is based on *divide-and-conquer* paradigm, similar to Mergesort.

Steps:

1. Partition an array into two subarrays
2. Sort each subarray independently,
3. Combine sorted subarrays.

Quicksort Steps

To sort the subarray $A[p \ldots r]$:

- **Divide:** Partition $A[p \ldots r]$ into two subarrays $A[p \ldots q\text{-}1]$ and $A[q+1 \ldots r]$, such that each element in the first subarray $A[p \ldots q\text{-}1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray $A[q+1 \ldots r]$. Compute index ***q*** as a part of partition procedure
- **Conquer:** Sort the two subarrays by recursive calls to QUICKSORT.
- **Combine:** No work is needed to combine the subarrays, because they are sorted in place.

$\text{QUICKSORT}(A, p, r)$

```
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

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

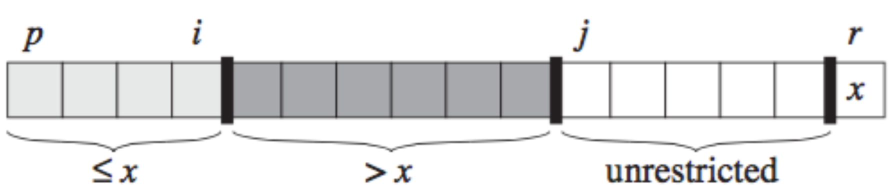
$\text{PARTITION}(A, p, r)$

```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Runtime of PARTITION : Θ(n)

where n=r-p+1

- PARTITION always selects the last element A[r] in the subarray A[p…r] as the ***pivot*** - the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty.



Performance of Quicksort

The running time of quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then quicksort can run as fast as mergesort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst-case Partitioning

- Occurs when the sub-arrays are completely unbalanced.
  i.e. one part with "n-1" elements, the other with only 0 element

  $PARTITION : \Theta(n)$

  $$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k)$$

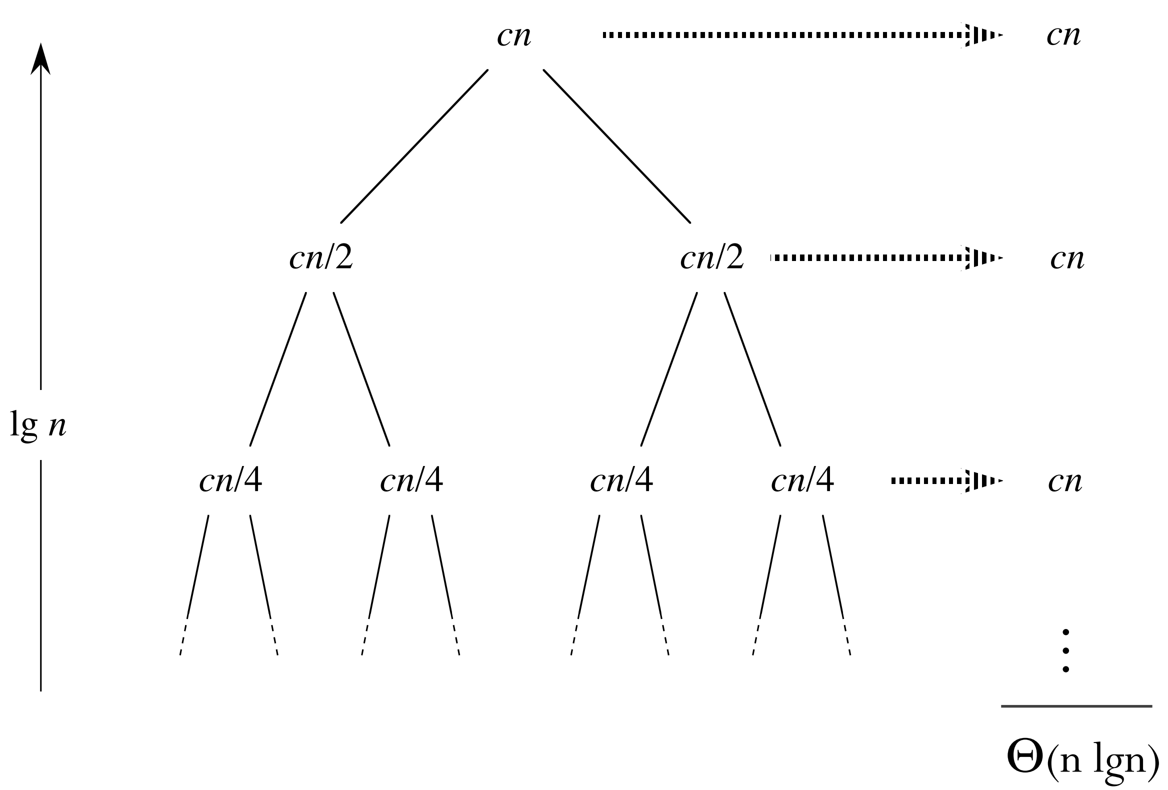
  $$T(n) = \Theta(n^2)$$

- Same running time as insertion sort.
- It happens when input array is already ordered!

Best-case Partitioning

- Assume that PARTITION always produces n/2 splits.

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$$



Balanced Partitioning

- Assume that PARTITION always produces a 9-to-1 split.

  $T(n) = T(9n/10) + T(n/10) + n = \Theta(n \lg n)$