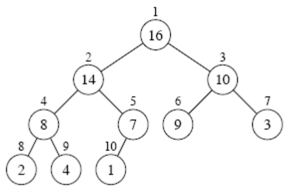
Data Structures

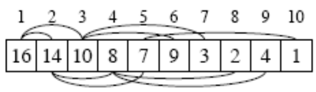
Heap, Heap Sort & Priority Queue

Heap

- Is a nearly **complete** binary tree.
  - height is $\Theta(\lg n)$.

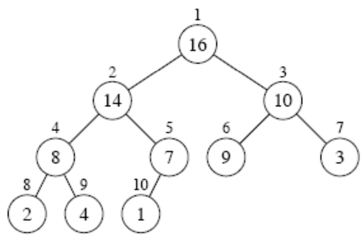


  - In general, heaps can be k-ary tree instead of binary.
- A heap can be stored as an **array** A.
  - Root of tree is A[1].
  - Parent of A[i] = A[⌊i/2⌋].
  - Left child of A[i] = A[2i].
  - Right child of A[i] = A[2i + 1].



Heap property

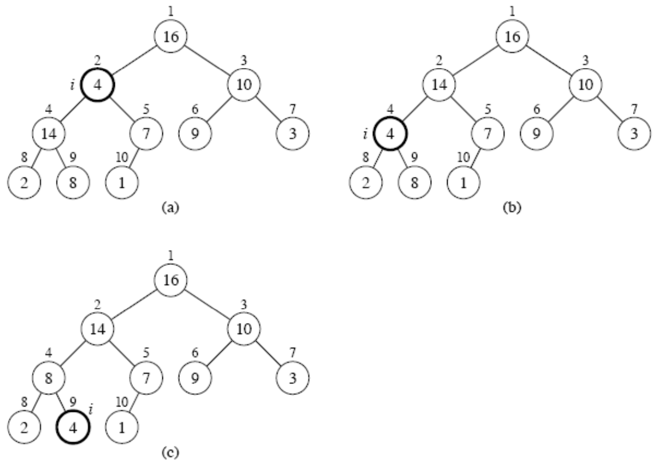
- **Max-Heap** property:
  - for all nodes i, excluding the root, A[parent(i)] ≥ A[i].
- **Min-Heap** property:
  - for all nodes i, excluding the root, A[parent(i)] ≤ A[i].



Max-Heapify

- **Before** max-heapify
  - A[i] may be smaller than its children.
  - Assume left and right sub-trees of i are max-heaps.
- **After** max-heapify
  - Sub-tree rooted at i is a max-heap.

Max-Heapify



Max-Heapify

maxHeapify (A, i, n)
  l = **left**(i), r = **right**(i)
  largest = i
  if (l ≤ n && A[l] > A[largest]) **then** largest = l
  if (r ≤ n && A[r] > A[largest]) **then** largest = r
  if (largest != i) **then**
    exchange A[i] **with** A[largest]
    maxHeapify(A, largest, n)

- **Complexity**: O(lg n)

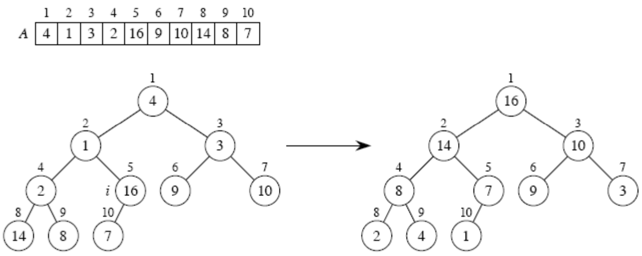
Tzachi (Isaac) Rosen

Building a Max-Heap

buildMaxHeap (A, n)

  for (i = ⌊n/2⌋ downto 1) **do**

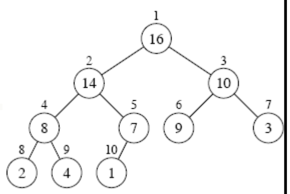
    maxHeapify(A, i, n)



Building a Max-Heap

- **Simple bound**:
  - O(n) calls to MAX-HEAPIFY,
  - Each of which takes O(lg n),
  - Complexity: O(n lg n).

Building a Max-Heap



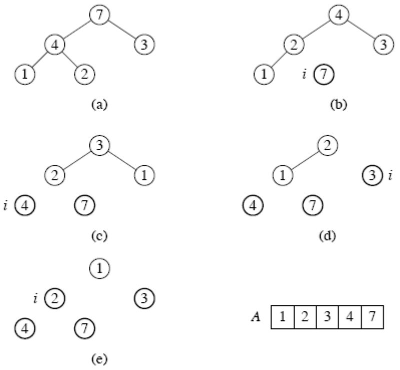
- **Tighter analysis**:
  - Number of nodes of height h ≤ $\lceil n/2^{h+1} \rceil$
  - The height of heap is lg n,
  - The time required by maxHeapify on a node of height h is O(h),
  - So the total cost of is: $\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$.
  - $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$ (substituting $x = 1/2$ in the formula $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ for $|x| < 1$
  - Thus, the running time of BUILD-MAX-HEAP is O(n).

Heapsort

- O(n lg n) **worst case**.
  - Like merge sort.
- Sorts **in place**.
  - Like insertion sort.
- Combines the **best** of both algorithms.

Heapsort

heapSort (A, n)
  buildMaxHeap(A, n)
  for (i = n downto 2) **do**
    exchange A[1] **with** A[i]
    maxHeapify(A, 1, i − 1)



- Complexity:
  - buildMaxHeap: O(n)
  - for loop**:**
    - n − 1 times
    - exchange elements: O(1)
    - maxHeapify: O(lg n)
  - Total time: O(n lg n).

Priority Queue

- Each element has a **key**.
- **Max-priority queue** supports operations:
  - **insert** (S, x): inserts element x into set S.
  - **maximum** (S): returns largest key element of S.
  - **extractMax** (S): removes and returns the largest key element of S.
  - **increaseKey** (S, x, k): increases value of element x’s key to k. (Assume k ≥ x’s current key value).
- **Min-priority queue** supports similar operations.

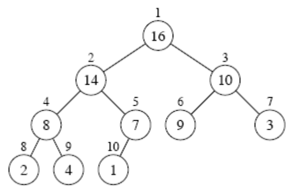
Maximum

maximum (A)
  return A[1]

- Complexity : O(1).

extractMax (A, n)
  max = A[1]
  A[1] = A[n]
  maxHeapify(A, 1, n – 1)
  return max

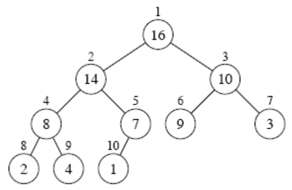
- Complexity : O(lg n).



Increasing Key

increasingKey (A, i, key)
  A[i] = key
  while (i > 1 & A[**parent**(i)] < A[i]) **do**
    exchange A[i] **with** A[**parent**(i)]
    i ← **parent(i)**

- Complexity : O(lg n).



Insertion

insert (A, key, n)

  A[n + 1] = −∞

  increasingKey(A, n + 1, key)

- Complexity : O(lg n).