ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Heap(sort) & Priority Queue

Prof. Dr. Goran Glavaš

9.11.2023

Content

- Heap
- Heapsort
- Priority Queue

Recap: **Insert(ion) sort**

```
insert_sort(L)
  for i = 1 to L.length - 1
    key = L[i]
    j = i-1
    while j > -1 and L[j] > key
      L[j+1] = L[j]
      j = j - 1
    L[j+1] = key
```

Algorithm design: incremental

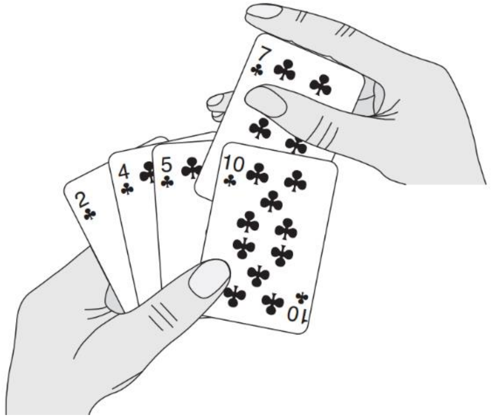


Image from *Cormen et al.*

Recap: **Merge sort**

```
merge_sort(A, p, r)
    n = r - p + 1
    if n % 2 == 1
      q = p + n//2
    else
      q = p + n/2 - 1

    merge_sort(A, p, q)
    merge_sort(A, q + 1, r)
    merge(A, p, q, r)
```

```
merge(A, p, q, r)
    n_left = q - p + 1
    n_right = r - q
    L = array[n_left]
    R = array[n_right]

    for i = 0 to n_left - 1:
      L[i] = A[p + i]
    for j = 0 to n_right - 1:
      R[j] = A[q + 1 + j]

    ind_l = 0
    ind_r = 0
    for k = p to r
      if ind_r > n_right - 1 or L[ind_l] ≤ R[ind_r]
        A[k] = L[ind_l]
        ind_l = ind_l + 1
      else
        A[k] = R[ind_r]
        ind_r = ind_r + 1
```

Algorithm design:

divide and conquer

Recap: **Quicksort**

```
quick_sort(A, p, r)
    q = partition(A, p, r)
    quick_sort(A, p, q - 1)
    quick_sort(A, q + 1, r)
```

```
partition(A, p, r)
    pivot = A[r]
    s = p - 1
    for i = p to r - 1:
        if A[i] ≤ pivot
            s = s + 1
            exchange(A[i], A[s])
    exchange(A[s+1], A[r])
    return s + 1
```

Algorithm design: divide and conquer

Sorting thus far…

Sorting Problem

Input: A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$

(Desired) Output: A permutation (reordering) of the input $\langle a'_1, a'_2, ..., a'_n \rangle$ such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

- **Insert(ion) sort**: $O(n^2)$ and **in place**
- **Merge sort**: $O(n \textit{ log } n)$ and not in place
- **Quick sort**: *worst* $O(n^2)$, **average** $O(n \textit{ log } n)$ and **in place**

- On average, **quick sort** the best solution so far

Heap and Heapsort

- New **algorithm design** „technique”: usage of a **special data structure** to manage information

- Data structure being used typically has properties that allow for the reduction of runtime complexity of the algorithm
    - The additional data structure requires memory (and maintenance)
    - Trading space for time („no free lunch”)

- **Heapsort**: sorting algorithm that relies on a data structure called **heap** – an array that represents a binary tree

Heap

- **Heap** is technically just an array
  - But elements stored so that it reflects a structure of a **binary tree**
  - Each **element** of the array is one **node** of the tree
  - The tree is completely filled (on all levels except the last)
    - Cannot add next level of the tree without filling the previous

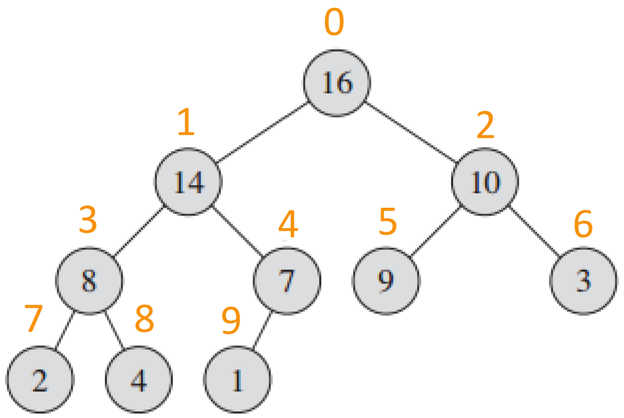
- **A**: an array we use to implement the heap
  - **A.Length**: the size of the array, i.e., the maximal possible size of the heap
  - **A.HeapSize**: the actual size of the heap (no. elements on the heap)
  - **A[0..A.Length-1]** – memory **allocated** for the heap
  - **A[0..A.HeapSize-1]** – memory actually **occupied** by the heap

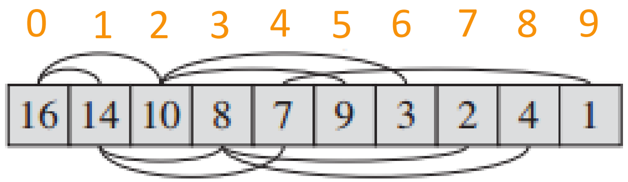
Binary Tree in the Array

- The order in which we manipulate the order of the elements is crucial

- **Binary tree**
  - Every **node** („parent”) has two „child” nodes

- First array element is the **root of the tree**
  - Second array element = root’s left child
  - Third array element = root’s right child

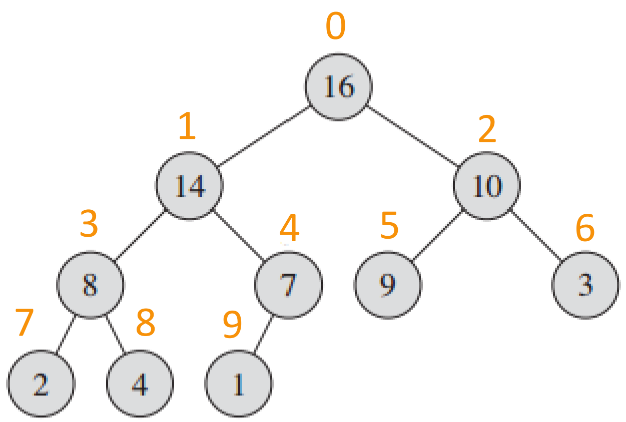


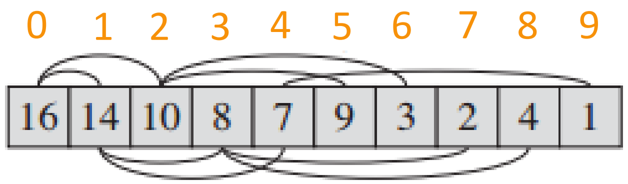


Binary Tree in the Array

- Consider the *i*-th index of the array
- At which indices would we find
  - A **PARENT** of the node at index i?
  - A **SIBLING** of the node at index i?
  - **CHILDREN** of the node at index i?

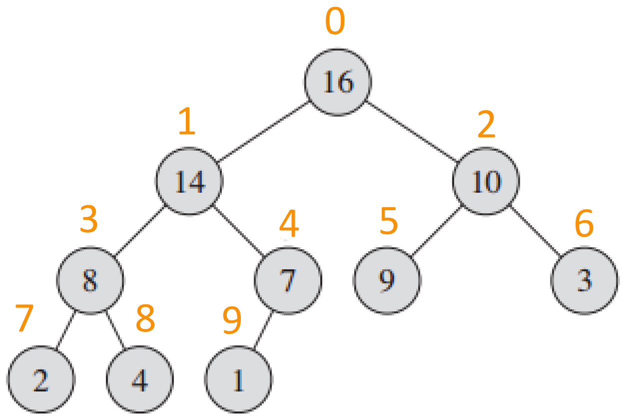
- How many nodes would we find at the j-th level of the tree (root is at level 0)

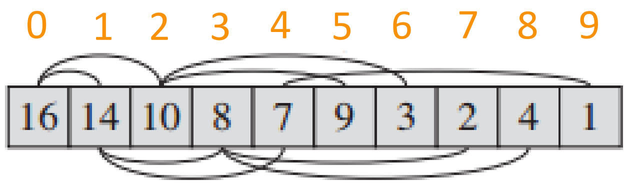




Binary Tree in the Array

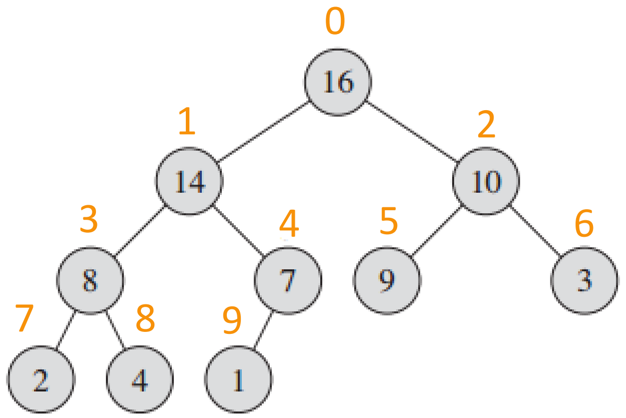
- How many nodes would we find at the j-th level of the tree (root is at level 0)?
- If at the j-th level we have m elements, how many elements are at the (j+1)-th level?
- What is the „height” (or „depth”) of the *full* binary tree that has n elements?
    - Level 0: **1** element
    - Level 1: **2** elements
    - Level 2: **4** elements
    - …

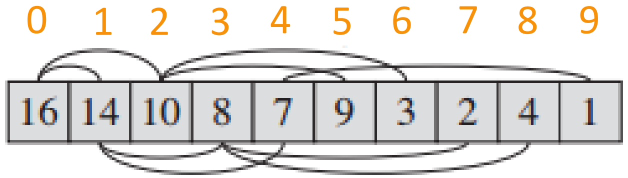




Binary Tree in the Array

- Consider the *i*-th index of the array
- At which indices would we find
  - A **PARENT** of the node at index i?
  - A **SIBLING** of the node at index i?
  - **CHILDREN** of the node at index i?





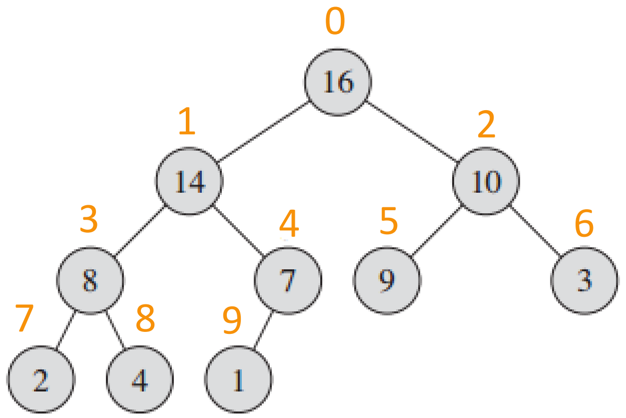

```
parent(i)
  if i % 2 == 0
    return i/2 - 1
  else
    return i//2
# or just (i-1)//2
```

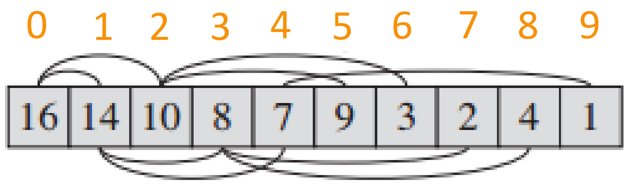
```
sibling(i)
  if i % 2 == 0
    return i - 1
  else
    return i + 1
```

```
left_child(i)
  return 2*i + 1

right_child(i)
  return 2*i + 2
```

Heap

- This is just a binary tree implemented in an array

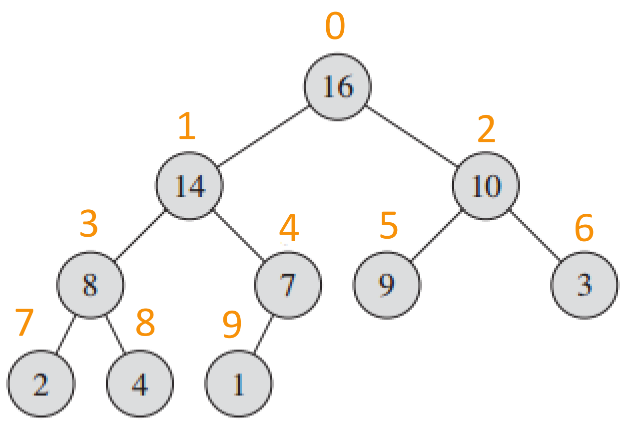


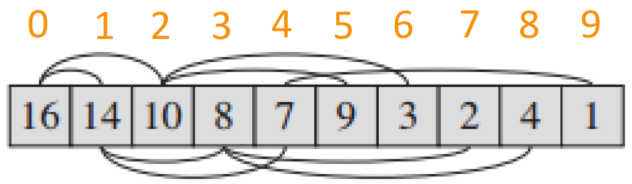


- In order for it to be a **heap**, it has to satisfy the **heap property** (for all nodes except the root)
- **Max-heap** (max-heap property):
  - `A[parent(i)] ≥ A[i]`
- **Min-heap** (min-heap property):
  - `A[parent(i)] ≤ A[i]`

Heap

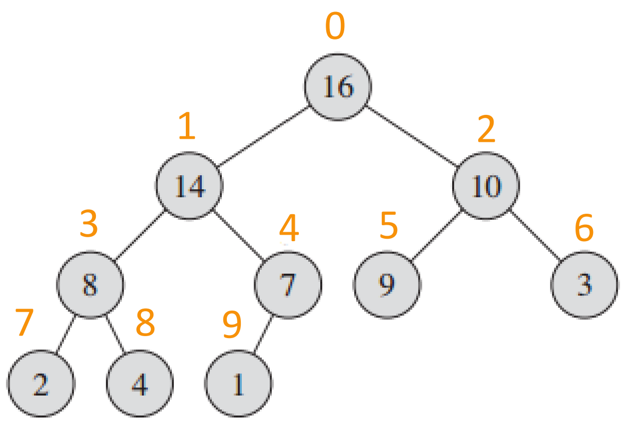
- The **height** of the tree that has between *n/2+1* and *n* elements is $log_2 n$
- **Heaps** are used for two things
  - To implement an abstract data structure called **priority queue**
  - To allow for an efficient **sorting algorithm**
  - Both applications demand the **maintenance** of the max-heap/min-heap property

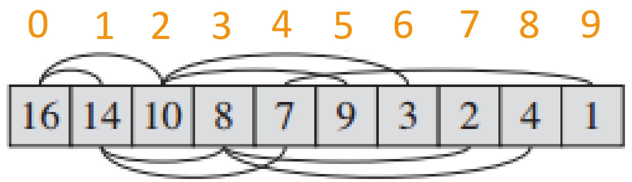




Heap: Maintaining the Heap Property

- **HEAPIFY** procedure (assume max-heap)
  - **Recursive** (algorithms operating on trees are often recursive)
  - Assumes subtrees rooted in each of the children nodes are already max-heaps
  - Takes the array and the index of a **node**





Heap: Maintaining the Heap Property

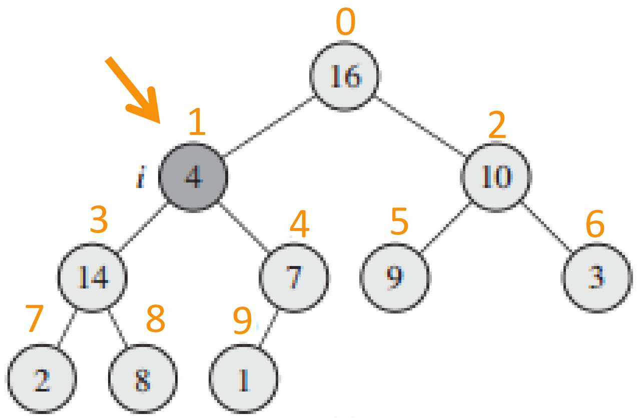
- **HEAPIFY** procedure (assume max-heap)
  - Recursive (many algorithms operating on trees are)
  - Assumes subtrees rooted in each of the children nodes are already max-heaps
  - Takes the array and the index of a node as input
  - **Q**: how to modify `heapify` to maintain min-heap?

```
heapify(A, i)
  l = left_child(i) # 2*i + 1
  r = right_child(i)
  if l < A.HeapSize and A[l] > A[i]
    largest = l
  else
    largest = i
  if r < A.HeapSize and A[r] > A[largest]
    largest = r

  if largest ≠ i
    exchange(A[i], A[largest])
    heapify(A, largest)
```

Maintaining the heap property

```
heapify(A, i)
  l = left_child(i)
  r = right_child(i)
  if l < A.HeapSize and A[l] > A[i]
    largest = l
  else
    largest = i
  if r < A.HeapSize and A[r] > A[largest]
    largest = r

  if largest ≠ i
   exchange(A[i], A[largest])
   heapify(A, largest)
```

heapify(A, 1)

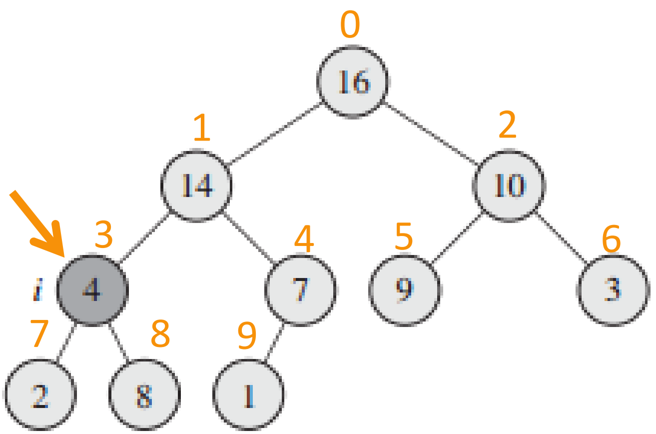



```
i = 1, A[i] = 4
l = 3, r = 4
A[l] (14) > A[i] (4) → True
largest = l = 3
A[r] (7) > A[largest] (14) → False
largest (3) ≠ i (1) → True
exchange(A[1](4), A[3](14))
--
heapify(A, 3) # recursive call
```

Maintaining the heap property

```
heapify(A, i)
  l = left_child(i)
  r = right_child(i)
  if l < A.HeapSize and A[l] > A[i]
    largest = l
  else
    largest = i
  if r < A.HeapSize and A[r] > A[largest]
    largest = r

  if largest ≠ i
   exchange(A[i], A[largest])
   heapify(A, largest)
```

heapify(A, 3)

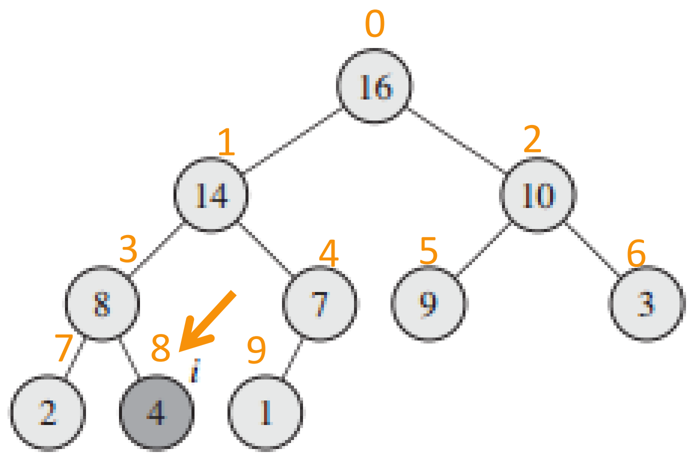



```
i = 3, A[i] = 4
l = 7, r = 8
A[l] (2) > A[i] (4) → False
A[r] (8) > A[largest] (4) → True
largest = r = 8
largest (8) ≠ i (3) → True
exchange(A[3](4), A[8](8))
--
heapify(A, 8) # recursive call
```

Maintaining the heap property

```
heapify(A, i)
  l = left_child(i)
  r = right_child(i)
  if l < A.HeapSize and A[l] > A[i]
    largest = l
  else
    largest = i
  if r < A.HeapSize and A[r] > A[largest]
    largest = r

  if largest ≠ i
   exchange(A[i], A[largest])
   heapify(A, largest)
```

heapify(A, 8)




```
i = 8, A[i] = 4
l = 17, r = 18
l (17) < A.HeapSize (10) → False
largest = i = 8
l (17) < A.HeapSize (10) → False
largest (8) ≠ i (8) → False
# end of execution
```

Heapify – running time

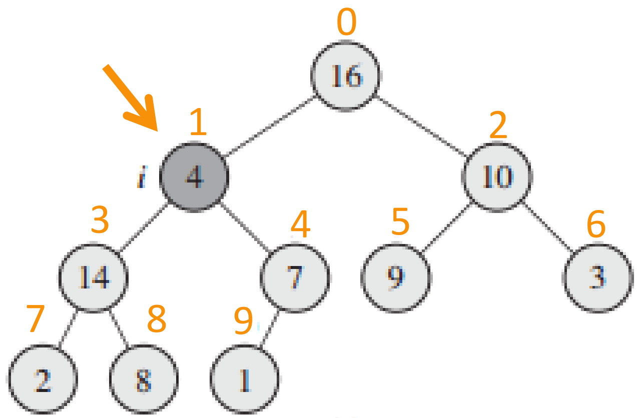
(1) Finding the lagest amont `i`, `l`, and `r`
constant time → O(1)
(2) Exchange of the elements → **O(1)**

- If n is the size of (number of elements in) subtree of `i`, what is the **worst case** number of executions of **(1)** and **(2)**?

- It is the **height of the subtree** at `i`
- **Height** of the binary tree with n elements?
  - **O(log n)**

$T(n) = O(\log n) * (O(1) + O(1))$
$= O(\log n)$

```
heapify(A, i)
  l = left_child(i)
  r = right_child(i)
  if l < A.HeapSize and A[l] > A[i]
    largest = l
  else
    largest = i
  if r < A.HeapSize and A[r] > A[largest]
    largest = r

  if largest ≠ i
   exchange(A[i], A[largest])
   heapify(A, largest)
```



Heapify

- **`heapify(A, i)`** for an index **`i`** (in a max-heap) effectively pushes the element down the subtree given by that index
  - So long as the element is smaller than at least one of its children
- Does **`heapify(A, i)`** turn the subtree of **`i`** into a heap?
  - If no, why not? Provide a counter example
- **How many times** and **for which indices** (nodes) of the array do we need to call heapify in order to transform an array into a **heap**?

Building a heap

- Does `heapify(A, i)` turn the subtree of `i` into a heap? **No!**
  - If parent element larger than both its children, heapify stops; but each child could be smaller than its children, violating the **heap property**
  - If parent smaller than both children, it is **„exchanged”** only with larger child
    - Recursive call follows **only** on the subtree of the larger child
    - Smaller child's subtree **won't be** „heapified”

Building a heap

- **How many times** and **for which indices** (nodes) of the array do we need to call heapify in order to transform an array into a **heap**?

```
build_heap(A)
   A.HeapSize = A.length
   nln = n//2

   for i in nln - 1 downto 0
      heapify(A, i)
```

- `heapify` propagates the „smaller values down”

  We actually want to propagate the „larger values up”

- To convert an array into a heap, we will call heapify in a **bottom-up manner**, **for each non-leaf node**

  Binary tree has n elements:

  - how many non-leaf nodes (nln) does it have?

Building a heap – illustration

- A.length = A.HeapSize = n = 10
- Number of non-leaf nodes (nln) = 5

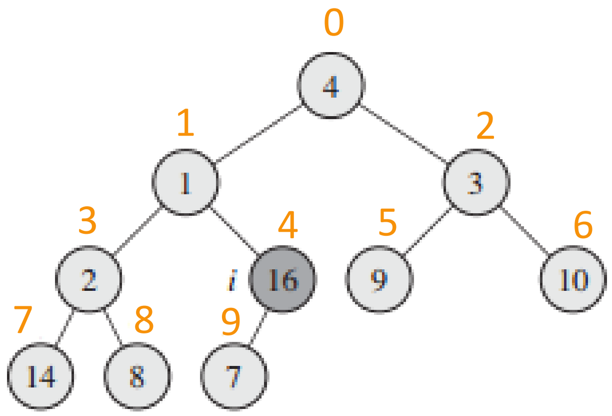
  (indices 0, 1, 2, 3, 4)

```
build_heap(A)
   A.HeapSize = A.length
   nln = n//2

   for i in nln - 1 downto 0
      heapify(A, i)
```

- Iteration **#1**:

  ```
  i = 4
  heapify(A, 4)
  ```

  A[4] (16) > A[9] (7) (its child), nothing happens



Building a heap – illustration

- A.length = A.HeapSize = n = 10
- Number of non-leaf nodes (nln) = 5

  (indices 0, 1, 2, 3, 4)

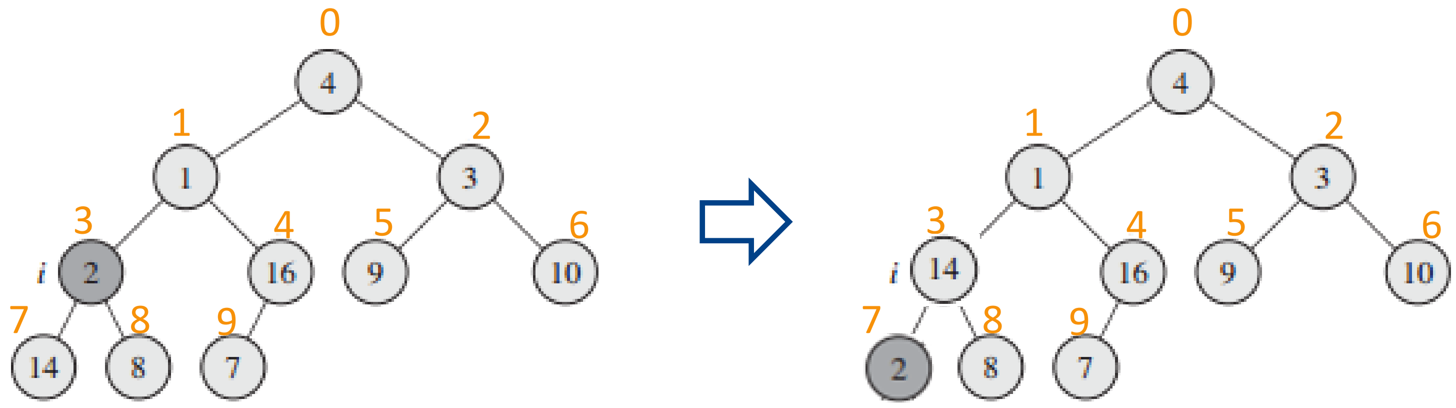
```
build_heap(A)
   A.HeapSize = A.length
   nln = n//2

   for i in nln - 1 downto 0
      heapify(A, i)
```

- Iteration **#2**:

  `i = 3`

  `heapify(A, 3)`

  A[3] (2) < A[7] (14) (its child), **exchange**



Building a heap – illustration

- A.length = A.HeapSize = n = 10
- Number of non-leaf nodes (nln) = 5

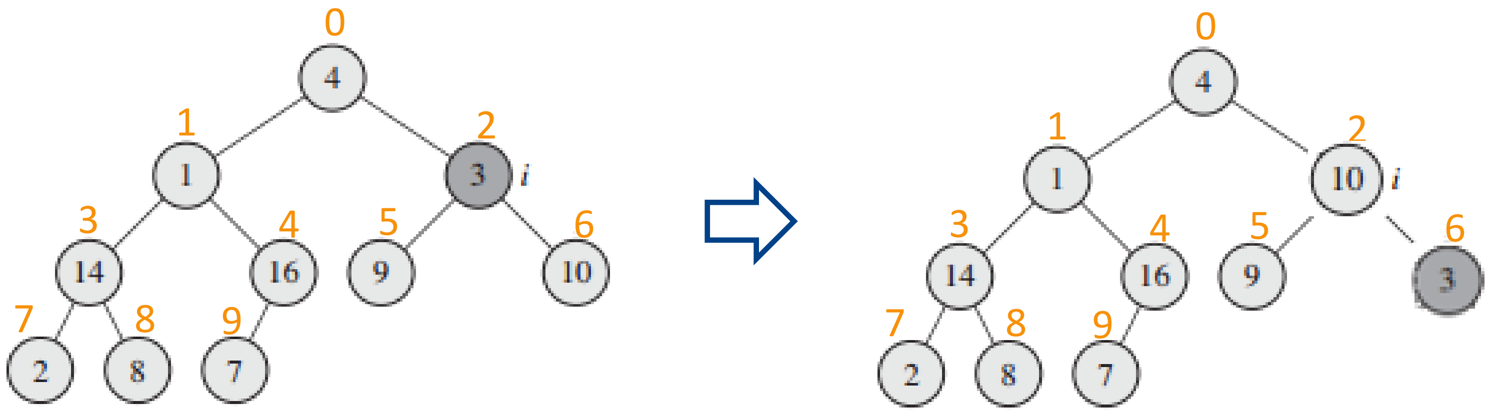
  (indices 0, 1, 2, 3, 4)

```
build_heap(A)
   A.HeapSize = A.length
   nln = n//2

   for i in nln - 1 downto 0
      heapify(A, i)
```

- Iteration **#3**:

  ```
  i = 2
  heapify(A, 2)
  ```

  A[2] (3) < A[6] (10) (its child), **exchange**



Building a heap – illustration

- A.length = A.HeapSize = n = 10
- Number of non-leaf nodes (nln) = 5

  (indices 0, 1, 2, 3, 4)

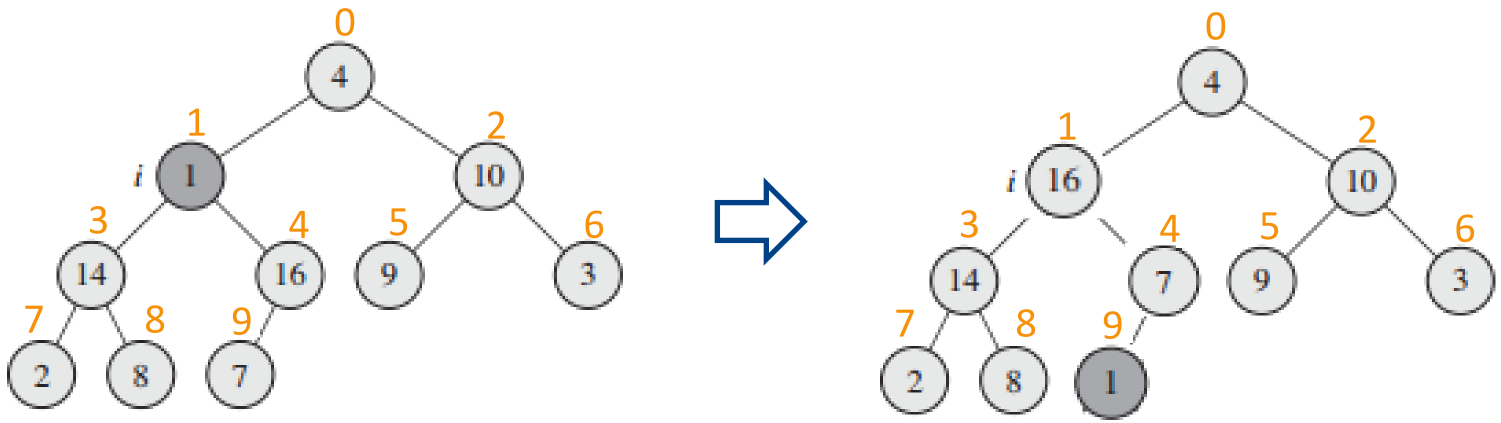
```
build_heap(A)
   A.HeapSize = A.length
   nln = n//2

   for i in nln - 1 downto 0
      heapify(A, i)
```

- Iteration **#4**:

```
i = 1
heapify(A, 1)
```

A[1] (1) < A[4] (16)
exchange

A[4] (1) < A[9] (7)
exchange



Building a heap – illustration

- A.length = A.HeapSize = n = 10
- Number of non-leaf nodes (nln) = 5
  (indices 0, 1, 2, 3, 4)

```
build_heap(A)
   A.HeapSize = A.length
   nln = n//2
   for i in nln - 1 downto 0
      heapify(A, i)
```

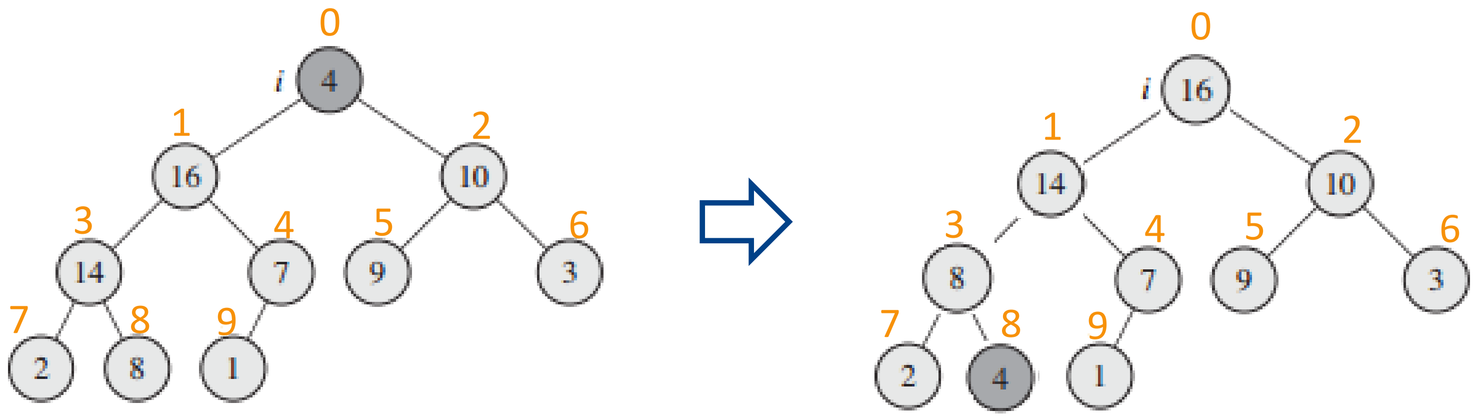
- Iteration **#5**:

  ```
  i = 0
   heapify(A, 0)
  ```

  A[0] (4) < A[1] (16)
  Exchange

  A[1] (4) < A[3] (14)
  exchange

  A[3] (4) < A[8] (8)
  exchange



Build heap – **runtime**

- We established that `heapify(A, i)` has runtime of **O(log n)**
- And we call heapify once for every non-leaf node, so **n/2** times
- Obvious **upper bound**: **T(n) = O(log n) * n/2 = O(n log n)**
  - **Q:** is it a tight bound?

- For „deeper” nodes, `heapify` will on average run much faster
  - For a node in the penultimate level, its subtree will have n’ = 2 or 3 nodes
  - For such nodes, runtime of heapify **O(log n’)** is much lower than **O(log n)**, where n is the size of the whole tree

Build heap – **runtime**

- Let H be the **height** of the tree, $H = \lfloor \log_2 n \rfloor$
  - Let h be the **height** of a node/index
  - Let d be the depth of a node/index, $d = H - h$
- For leaf nodes, $h = 0$, for root $h = H$

- **Q**: How many nodes (at most) do we have at some height h?
  - $h = H$ $(d = 0)$ → 1 node
  - $h = H - 1$ $(d = 1)$ → 2 nodes
  - ...
  - $h = 0$ $(d = H)$ → 2^d $(= 2^H)$ nodes
- Runtime of `heapify` for a node at height h is **O(h)**

Build heap – **runtime**

- Let H be the height of the tree, H = $\lfloor \log_2 n \rfloor$
  - Let h be the **height** of a node/index
  - Let d be the depth of a node/index, d = H – h
- **T(n)** $= \sum_{h=0}^{H} 2^d * O(h)$

$$= \sum_{h=0}^{H} 2^{(H-h)} * O(h)$$

$$= \sum_{h=0}^{H} 2^H/2^h * O(h)$$

$$\leq \sum_{h=0}^{H} n/2^h * O(h)$$

H = $\lfloor \log_2 n \rfloor$ means that

$2^H \leq n < 2^{H+1}$

$$= O\left(n \sum_{h=0}^{H} \frac{O(h)}{2^h}\right)$$

$$= \boldsymbol{O(n)}$$

O(h) means T(h) = c*h

When H is large (approx. infinity)

$$\sum_{h=0}^{\infty} \frac{c * h}{2^h} = c * \mathbf{2}$$

Content

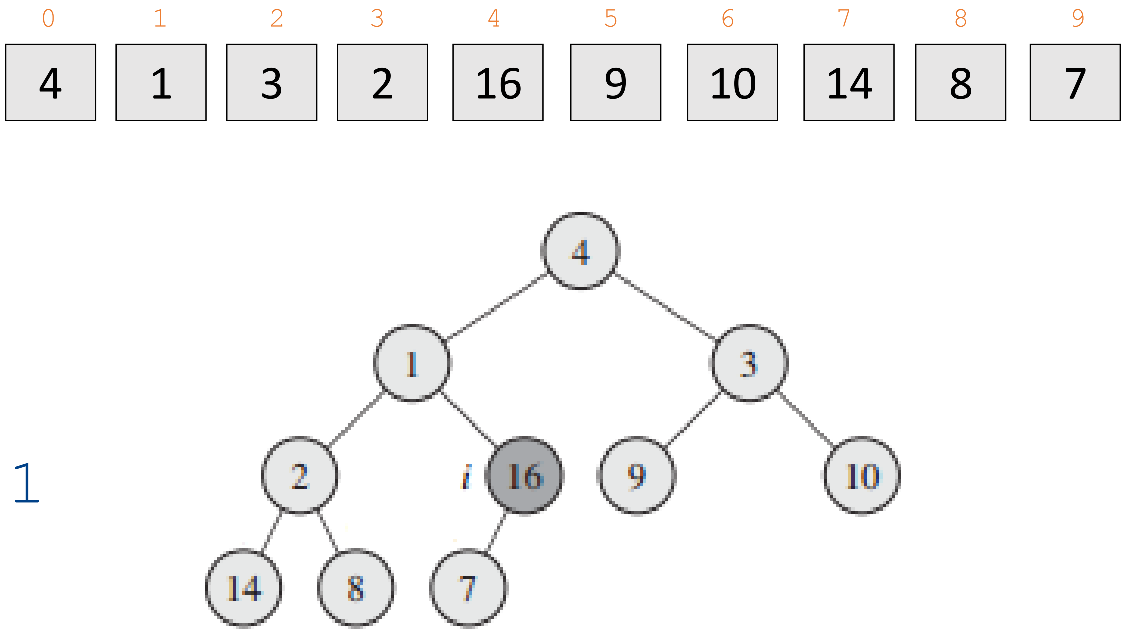
- Heap
- Heapsort
- Priority Queue

Heapsort

- Given an array, heapsort **first builds a heap** from it, then relies on it's **heap property** to ensure a sorted array

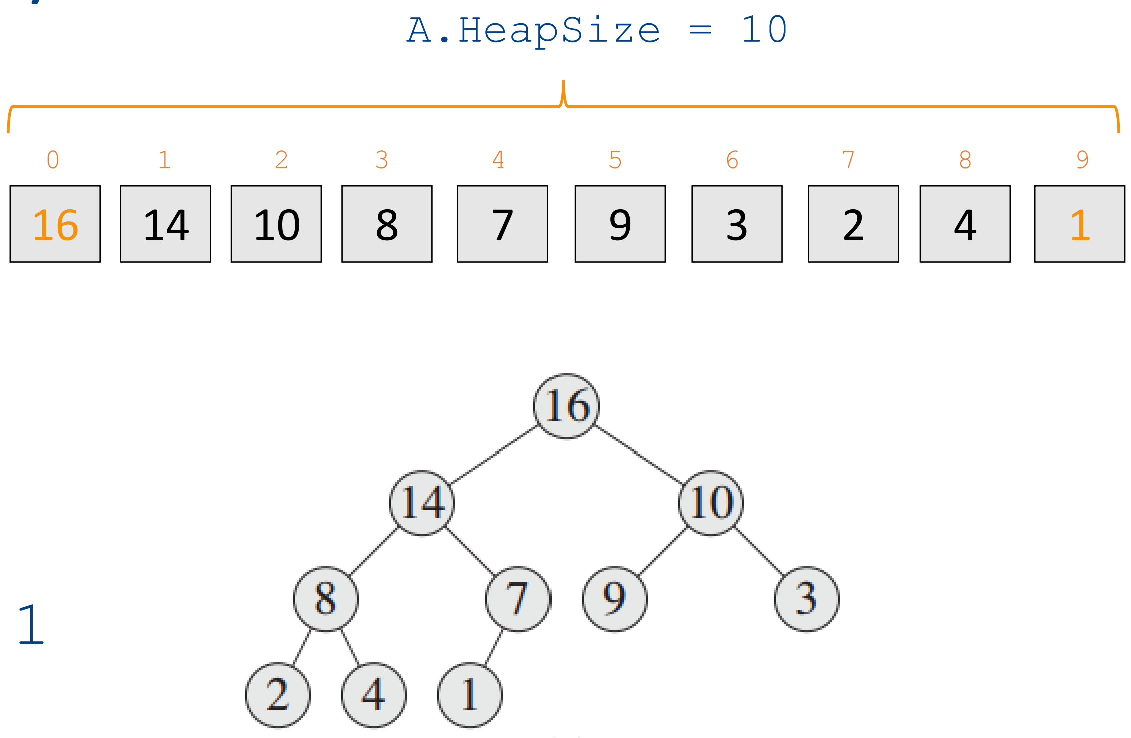
Input: array A

```
Heapsort(A)
  build_heap(A)
  len = A.HeapSize
  for i = len - 1 downto 1
    exchange(A[0], A[i])
    A.HeapSize = A.HeapSize - 1
    heapify(A, 0)
```



Heapsort

- Given an array, heapsort **first builds a heap** from it, then relies on it's **heap property** to ensure a sorted array

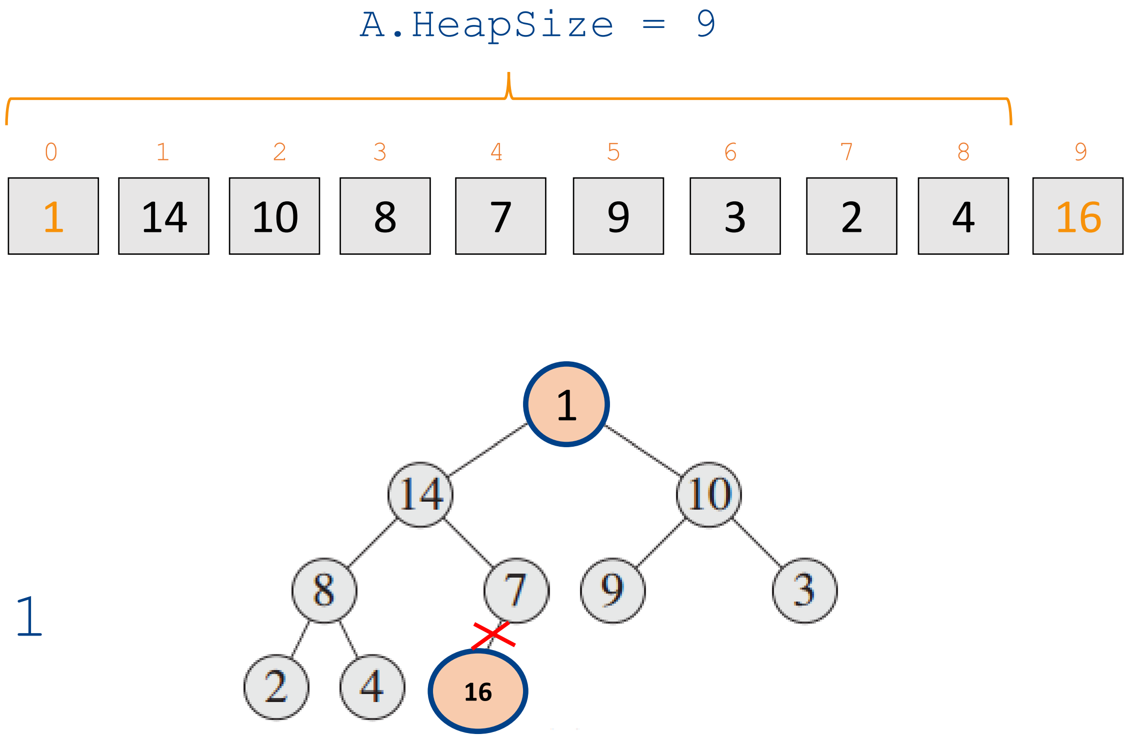



```
Heapsort(A)
  build_heap(A)
  len = A.HeapSize
  for i = len - 1 downto 1
    exchange(A[0], A[i])
    A.HeapSize = A.HeapSize - 1
    heapify(A, 0)
```

Heapsort

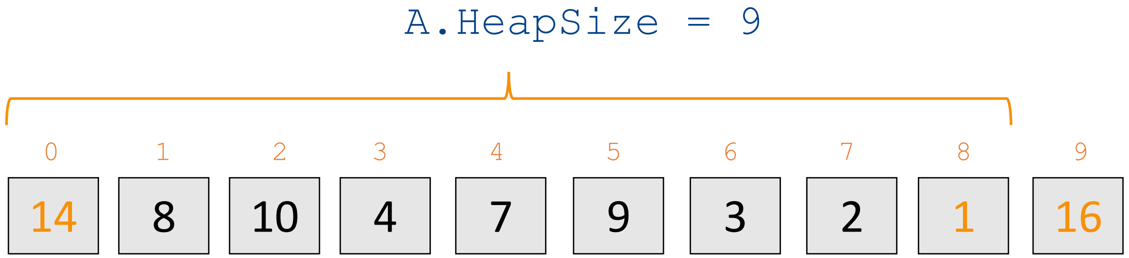
- Given an array, heapsort **first builds a heap** from it, then relies on it's **heap property** to ensure a sorted array

```
Heapsort(A)
  build_heap(A)
  len = A.HeapSize
  for i = len - 1 downto 1
    exchange(A[0], A[i])
    A.HeapSize = A.HeapSize - 1
    heapify(A, 0)
```

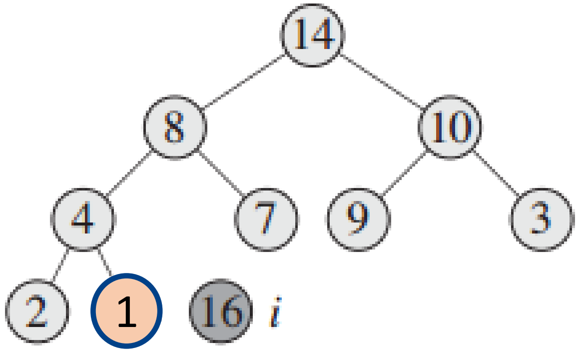


Heapsort

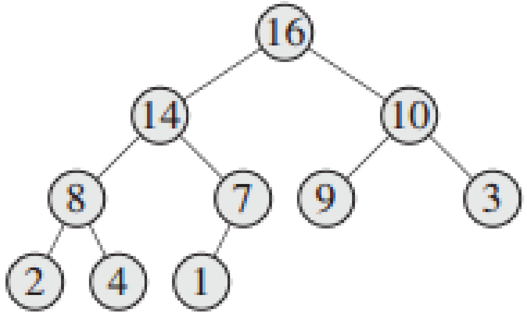
- Given an array, heapsort **first builds a heap** from it, then relies on it's **heap property** to ensure a sorted array



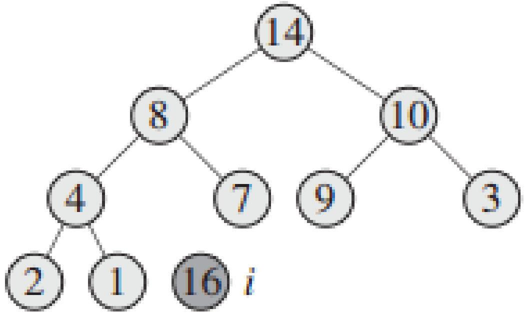

```
Heapsort(A)
  build_heap(A)
  len = A.HeapSize
  for i = len - 1 downto 1
    exchange(A[0], A[i])
    A.HeapSize = A.HeapSize - 1
    heapify(A, 0)
```



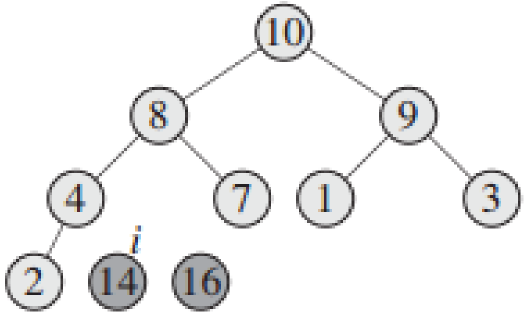
Heapsort



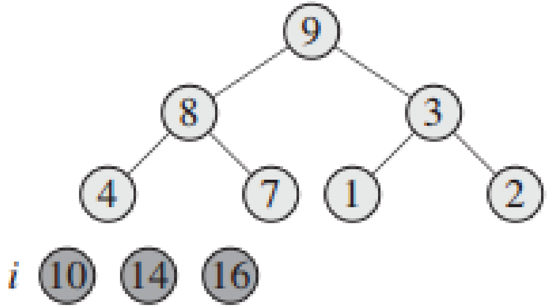
Heap built



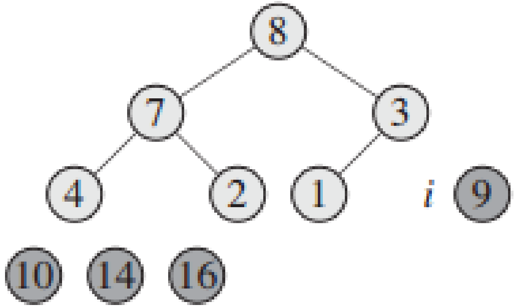
After iteration 1 (i = 9)



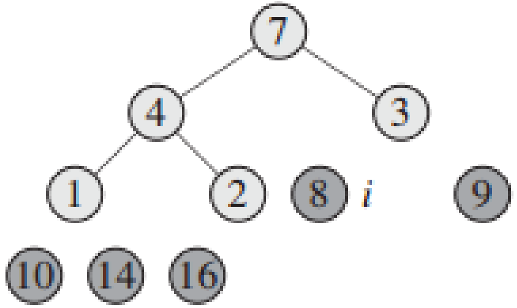
After iteration 2 (i = 8)



After iteration 3 (i = 7)

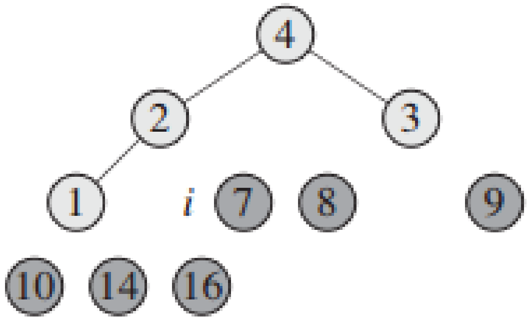


After iteration 4 (i = 6)

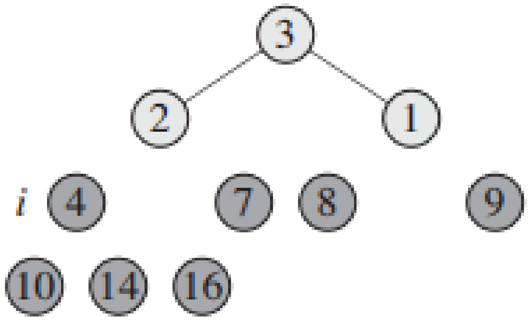


After iteration 5 (i = 5)

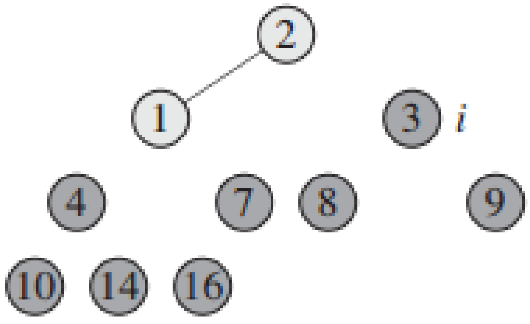
Heapsort



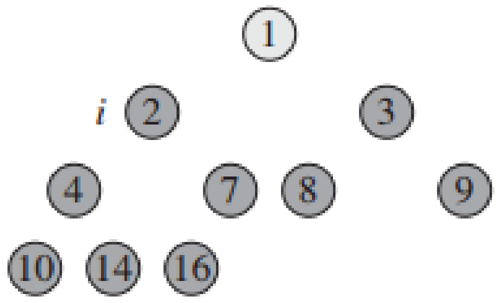
After iteration 6 (i = 4)



After iteration 7 (i = 3)



After iteration 8 (i = 2)

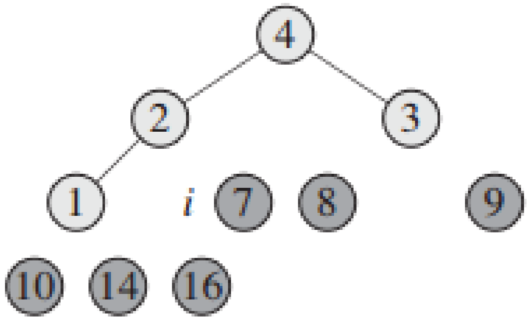


After iteration 9 (i = 1)

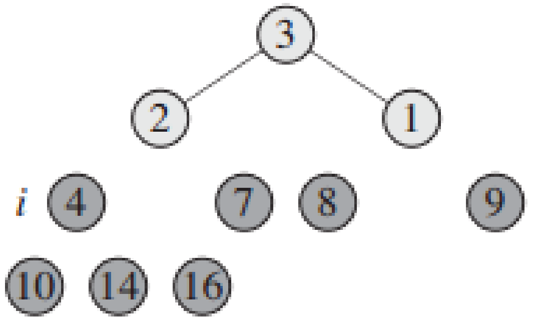
0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16

End of heapsort

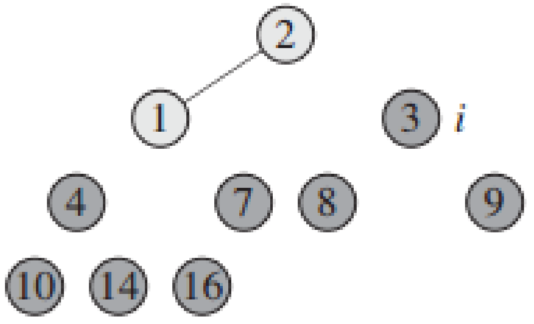
Heapsort



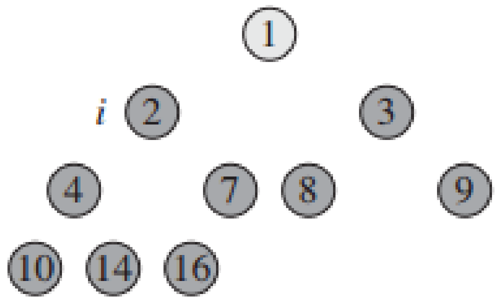
After iteration 6 (i = 4)



After iteration 7 (i = 3)



After iteration 8 (i = 2)



After iteration 9 (i = 1)

0	1	2	3	4	5	6	7	8	9
1	2	3	4	7	8	9	10	14	16

End of heapsort

Heapsort – running time

- `build heap`: O(n)
- `heapify`: O(log n)
- For loop iterates n-1 times
  - `heapify` called n-1 times

- T(n) = O(n) + (n-1) * O(log n)
  = O(n * log n)

- Q: Does heapsort sort in place?

```
Heapsort(A)
  build_heap(A)
  len = A.HeapSize
  for i = len - 1 downto 1
    exchange(A[0], A[i])
    A.HeapSize = A.HeapSize - 1
    heapify(A, 0)
```

Content

- Heap
- Heapsort
- Priority Queue

Priority Queue

- We've used heap as a data structure that supports heapsort
  - In most practical sorting applications, **quicksort** faster than **heapsort**
- But heap is useful for more than just sorting, as an actual implementation of an ADS called **priority queue**

Priority queuing

A set of elements S, each s ∈ S has a corresponding **priority number (key)** assigned to it.
Elements with higher priority should be processed before elements of lower priority.
Elements with the same priority should be processed in the order of insertion (queue).

- Example: scheduling execution of jobs (programs) on a shared computer server

Priority queue

- **Max-**Priority queue has the following operations
  - `Insert(S,x)` – inserts the element x into S (equivalent to $S = S \cup \{x\}$)
  - `Maximum(S)` – returns $s \in S$ with the highest priority (key)
  - `Extract-Max(S)` – removes and returns $s \in S$ with the highest priority
  - `Increase-Prio(S, x, k)` – increase the priority of the element x to the new priotity value k
    - For **max-PQ**, we assume we never reduce prioroty, only increase it

- **Min**-Priority queue has:
  - `Insert`, `Minimum`, `Extract-Min`, `Decrease-Prio`

Priority queue with **heap**

- Elements of the set S stored in an array A
- We assume that every element of S the heap's array is a structure with two values
  - **key** (`A[i].key`): this is the priority indicator – in max-PQ, larger key means higher priority
  - **value** (`A[i].value`): the actual data of the element (<u>not used</u> for heap organization)

```
Maximum(A)
  return A[0]
```

O(1)

```
Extract-Max(A)
  if A.HeapSize < 1
    error „underflow”
  max = A[0]
  A[0] = A[A.HeapSize - 1]
  A.HeapSize = A.HeapSize - 1
  heapify(A, 0)
  return max
```

O(log n)

Priority queue with **heap**

```
Increase-Prio(A, i, key)
  if key < A[i]
    error „new key smaller than current”

  A[i].key = key

  # restore heap property by pushing A[i] up
  while i > 0 and A[i].key > A[parent(i)].key
    exchange(A[i], A[parent(i)])
    i = parent(i)
```

O(log n)

```
Insert(A, key)
  if A.HeapSize = A.Length
    error „overflow”

  A.HeapSize = A.HeapSize + 1
  A[A.HeapSize - 1].key = -inf # some big negative value
  Increase-Prio(A, A.HeapSize - 1, key)
```

O(log n)

Questions?

Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Questions?
Frågor?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
ຄຳຖາມ?
أسئلة؟