# Lecture 5: Binary heaps

## Sorting algorithms: Heapsort and Quicksort

- array $A$ of objects with 2 special attributes: $A.length$ and $A.heap\_size$.
- it represents a complete binary tree with $A.heap\_size$ nodes
  - The tree is completely filled on all levels except possibly the lowest, which is filled from left to right
  - $A.length$ represents the maximum number of nodes of the tree. Therefore, $A.heap\_size \leq A.length$
- The index of the parent, left child, and right child of a node with index $i$ are computed as follows:

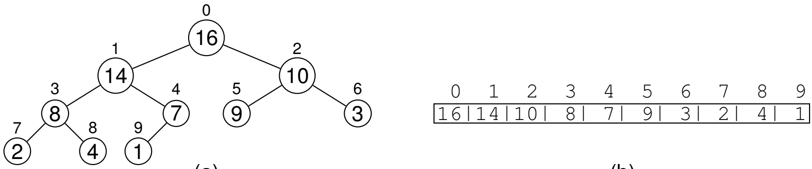
$$parent(i) := \begin{cases} \lfloor (i-1)/2 \rfloor & \text{if } i \neq 0 \\ -1 & \text{if } i = 0 \end{cases}$$

$$left(i) := 2 \cdot i + 1$$

$$right(i) := 2 \cdot i + 2$$

- The **heap property** must hold: $A[parent(i)] \geq A[i]$ for all $i \neq 0$.

# Binary heaps: Example




(a) (b)

A heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number next to a node is the corresponding index in the array.

## AUXILIARY NOTIONS

- **height of a node** in a tree := number of edges from the tree to a leaf.
- **height of the tree** := height of the root of the tree.

# Remarks

- The height of a binary heap is $\Theta(\log_2(n))$ – obvious.
- FIND / INSERT / REMOVE operations in binary heaps take $O(\log_2(n))$ time – we shall prove this.
- We are interested in the efficient implementation of:
  1. HEAPIFY$(A, i)$
  2. BUILDHEAP$(A)$
  3. HEAPSORT$(A)$
  4. EXTRACTMAX$(A)$
  5. INSERT$(A, key)$

  The purpose of these procedures will be explained later.

# HEAPIFY($A$, $i$)

- Takes as input an array $A$ and an index $i$, such that
  - the subtrees rooted at *left*($i$) and *right(i)* are binary heaps.
  - The subtree rooted at $i$ may not be a binary heap, because $A[i]$ is smaller than its children.
- Rearranges the elements of $A$ by letting $A[i]$ "float down" so that the subtree rooted at index $i$ becomes a binary heap.

**Thus, the purpose of** HEAPIFY **is to maintain the heap property of an array of values.**

# HEAPIFY($A$, $i$)

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 $A[r] > A[largest]$

    $largest := r$

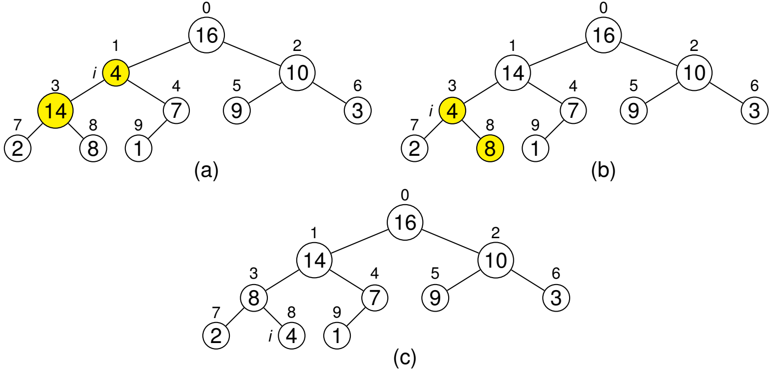
**if** $largest \neq i$

    exchange $A[i] \leftrightarrow A[largest]$

    HEAPIFY($A$, $largest$)

# Example

The action of HEAPIFY($A$, 1), where $A.heap\_size = 10$. Configuration **(a)** lacks heap property at index 1. The heap property for index 1 is restored in **(b)** by exchanging $A[1]$ with $A[3]$, which destroys the heap property for index 3. There recursive call HEAPIFY($A$, 3) sets $i = 3$, swaps $A[3] \leftrightarrow A[8]$ as shown in **(c)**, and the recursive call HEAPIFY($A$, 8) yields no further change to the data structure.



# Properties of HEAPIFY

- The running time complexity of HEAPIFY$(A, i)$ is $O(h)$, where $h$ is the height of node with index $i$.
- $\Rightarrow$ In general, the running time of HEAPIFY$(A, i)$ is $O(\log_2(n))$.
- For a proof, check the references.

# Building a binary heap
## BUILDHEAP($A$)

- Rearranges the elements of an array $A$, to have the binary heap property.
- The rearrangement is achieved by successive runs of HEAPIFY($A, i$)

BUILDHEAP($A$)

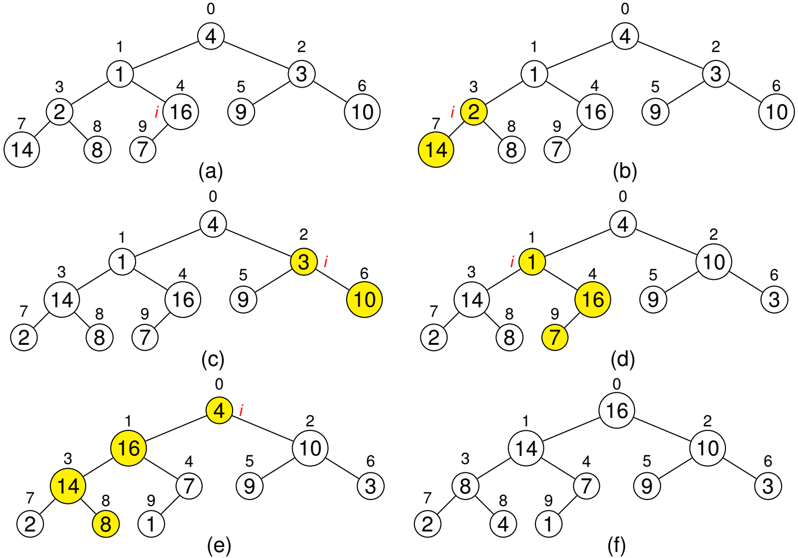
1 $\quad$ $heap\_size(A) := A.length$
2 $\quad$ **for** $i := \lfloor (A.length - 1)/2 \rfloor$ **downto** 0
3 $\quad\quad$ HEAPIFY($A, i$)

**Remarks**

- The order in which the nodes are processed guarantees that the subtrees rooted at children of a node $i$ are heaps before HEAPIFY is run at that node.
- There are $O(n)$ calls of HEAPIFY($A, i$), which has time complexity $O(\log_2 n) \Rightarrow$ time complexity $O(n \log_2 n)$.
- Tighter bound of the total runtime of step 3: $O(n)$ (see refs.)

# Example

BUILDHEAP($A$) for A={4,1,3,2,16,9,10,14,8,7}.



# The Heapsort algorithm

HEAPSORT($A$) rearranges the elements of an array $A$ in ascending order, using the following method:

1. Call BUILDHEAP($A$) $\Rightarrow$ a heap on the elements of the array $A[0..n-1]$
2. $A[0]$ is the maximum element of $A$
   - ▷ exchange $A[0] \leftrightarrow A[n-1]$, to place $A[0]$ into its correct final position.
3. Discard $A[n-1]$ from the heap by decrementing $A.heap\_size$. We still have to sort $A[0..n-2]$
   - $A[0..n-2]$ is *almost* a binary heap: 0 is the only index that may violate the heap property.
   - We run HEAPIFY($A, 0$) to rearrange $A[0..n-2]$ into binary heap.
   - The Heapsort algorithm repeats this process for the heap of size $n-1$ down to a heap of size 2.

# Heapsort

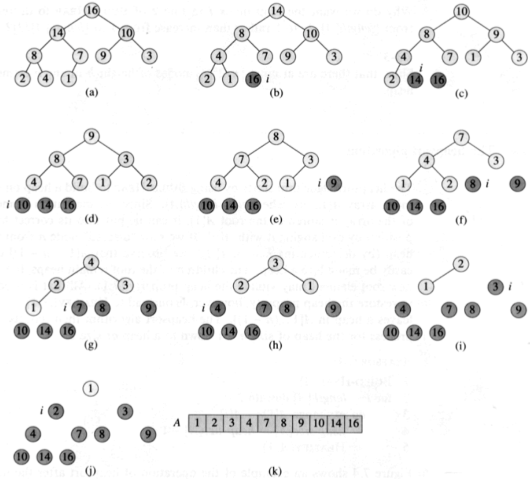
```
HEAPSORT(A)
  BUILDHEAP(A)
  for i := A.length − 1 downto 1
      exchange A[0] ↔ A[i]
      A.heap_size := A.heap_size − 1
      HEAPIFY(A, 0)
```

### TIME COMPLEXITY ANALYSIS

- BUILDHEAP($A$) takes $O(n)$ time.
- There are $n-1$ calls to HEAPIFY($A, 0$), and each one takes $O(log_2 n)$ time.

$\Rightarrow$ HEAPSORT($A$) takes $O(n \log_2 n)$ time, where $n = A.length$.

# Heapsort – running example




**(a)** The heap data structure just after it has been built by BUILDHEAP. **(b)–(j)** The heap just after each call of HEAPIFY in line 5. The value of *i* at that time is shown. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array *A*.

# Priority queues

A priority queue is a data structure for maintaining a set $S$ of elements, each with an associated value called a key. It is intended to support efficient execution of the following operations:

- INSERT$(S, x)$: inserts the element $x$ into a set $S$. We denote this operation by $S := S \cup \{x\}$.
- MAXIMUM$(S)$: returns the element of $S$ with the largest key.
- EXTRACTMAX$(S)$: removes and returns the element of $S$ with the largest key.

Applications of priority queues

- Job scheduling on a shared resource
  - The queue keeps track of jobs to be performed, and their relative priorities.
  - When a job is finished or interrupted, the highest-priority job is selected from the queue, using EXTRACTMAX
  - New jobs can be added at any time using INSERT
- Event-driven simulation: time of event occurrence serves as its key.

# Priority queues

Can be implemented efficiently using binary heaps.

```
ExtractMax(A)
  if A.heap_size < 1
      error "heap underflow"
  max := A[0]
  A[0] := A[A.heap_size − 1]
  A.heap_size := A.heap_size − 1
  Heapify(A, 0)
  return max
```

Running time analysis

- Heapify$(A, 0)$ takes $O(\log_2 n)$ time

  $\Rightarrow$ ExtractMax$(A)$ takes $O(\log_2 n)$ time.

# Priority queues
## INSERT($A$, $key$)

INSERT($A$, $key$) inserts a node into a binary heap $A$:

- First, it expands the heap by adding a new leaf to the tree.
- Then, it traverses a path from this leaf toward the root, to find a proper place for the new element.

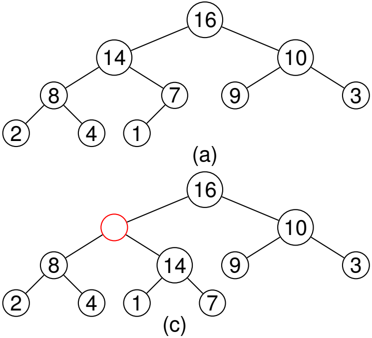
```
INSERT(A, key)
A.heap_size := A.heap_size + 1
i := A.heap_size − 1
while i > 0 and A[parent(i)] < key
        A[i] := A[parent(i)]
        i := parent(i)
A[i] := key
```

Running time analysis

- The path traced from the new leaf to the root has length $O(\log_2 n) \Rightarrow$ HEAPINSERT($A$, $key$) takes $O(\log_2 n)$ time, where $n = A.heap\_size$.

# Priority queues

## INSERT($A$, *key*) illustrated




**(a)** The heap before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** Key 15 is inserted into the tree.

- Sorting algorithm with worst-case running time $\Theta(n^2)$ on an input array of $n$ numbers.
- Very efficient on average: $\Theta(n \log n)$
- Often, the best practical choice for sorting

3-step divide-and-conquer algorithm for sorting a subarray $A[p..r]$

**Divide:** The subarray $A[p..r]$ is partitioned (rearranged) into two nonempty subarrays $A[p..q]$, $A[q+1..r]$ such that

- The elements of $A[p..q]$ are smaller than the elements of $A[q+1..r]$

The index $q$ is computed as part of this partitioning procedure.

**Conquer:** The subarrays $A[p..q]$ and $A[q+1..r]$ are sorted by recursive calls to quicksort.

**Combine:** Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

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

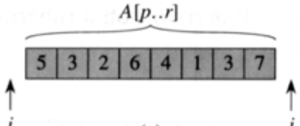
## Partitioning the array

```
PARTITION(A, p, r)
x ← A[p]
i ← p − 1
j ← r + 1
while TRUE
      do repeat j ← j − 1
            until A[j] ≤ x
         repeat i ← i + 1
            until A[i] ≥ x
         if i < j
            then exchange A[i] ↔ A[j]
            else return j
```

- Element $x = A[p]$ from $A[p..r]$ is selected as pivot around which to partition $A[p..r]$.
- The **while** loop grows two regions $A[p..i]$ and $A[j..r]$ from the top and bottom of $A[p..r]$, respectively, such that
  - Every element in $A[p..i]$ is less than or equal to $x$.
  - Every element in $A[j..r]$ is greater than or equal to $x$.

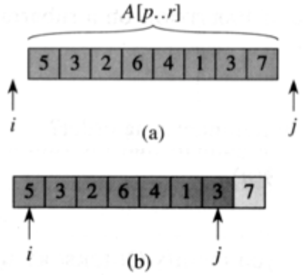
  Initially, $i = p - 1$ and $j = r + 1$, so the two regions are empty.
- Within the **while** loop, index $j$ is decremented and index $i$ is incremented, in lines 5-8, until $A[i] \geq x \geq A[j]$.
  - By exchanging $A[i]$ and $A[j]$, the two regions can be extended.
- The **while** loop repeats until $i \geq j$, at which point the entire array $A[p..r]$ has been partitioned into two subarrays $A[p..q]$ and $A[q+1..r]$ where $p \leq q < r$, such that all elements in $A[p..q]$ are smaller than or equal to any element in $A[q+1..r]$.
- The value $q = j$ is returned at the end of the procedure.

# Quicksort
## Example of how PARTITION works



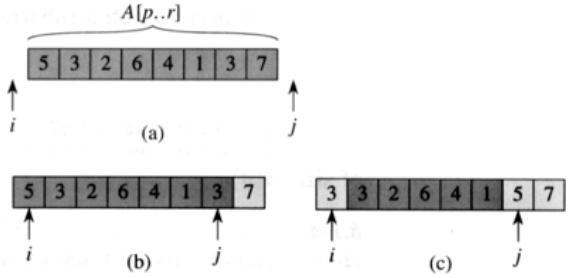
(a)

# Quicksort
## Example of how PARTITION works

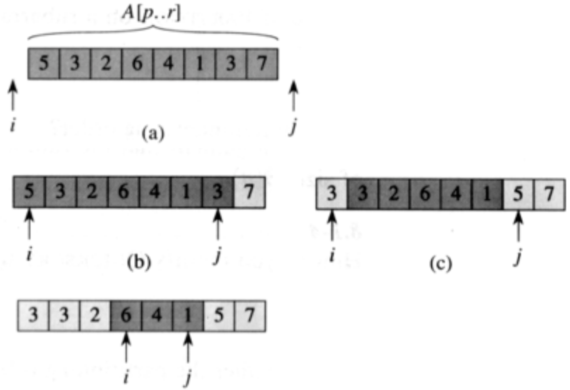


# Quicksort

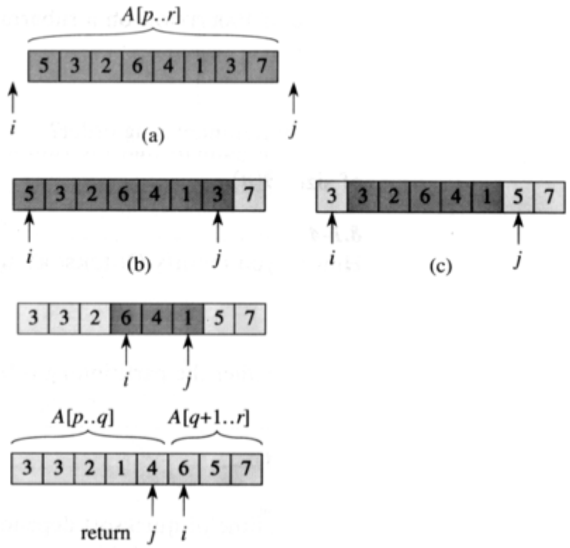
## Example of how PARTITION works



# Quicksort
## Example of how PARTITION works



Example of how PARTITION works



- The running time of PARTITION on an array $A[p..r]$ is $\Theta(r - p + 1)$.
- Worst case behavior happens when the partitioning alway produces one partition with 1 element, and the other with all the rest. In this case:
  - Partitioning an array of size $n$ takes $\Theta(n)$ time and $T(1) = \Theta(1)$.
  - The recurrence relation is $T(n) = T(n-1) + \Theta(n-1) = \ldots = \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$.

  $\Rightarrow$ in the worst case, the running time is $\Theta(n^2)$.
- Best case is when the partitioning produces regions of equal size $\Rightarrow$ the recurrence relation $T(n) = 2\,T(n/2) + \Theta(n)$.

  $\Rightarrow$ $T(n) = \Theta(n \log n)$
  (*Cf.* the Master Theorem)

# References

Chapters 7 (Heapsort) and 8 (Quicksort) from the book


- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 2000.