Algorithms and Data Structures
Lecture notes: Sorting algorithms, Cormen chapters 6 to 8

Lecturer: Michel Toulouse

Hanoi University of Science & Technology
michel.toulouse@soict.hust.edu.vn

11 mai 2023

Introduction

This section introduces the divide-and-conquer sorting algorithms, mergesort and quicksort

Also the restricted form of the Master theorem is introduced as well as its application to analyze the running time of mergesort

Quicksort average case analysis is provided

Last, heapsort is described

Properties of sorting algorithms

- In place
  - Sorting of a data structure does not require any external data structure for storing the intermediate steps
  - Selection and insertion sort algorithms are "In place". Merge sort requires intermediary arrays
- Stable
  - If two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted

9	3	3'	5	6	5'	2	1	3"

$\longrightarrow$ Unsorted array

1	2	3	3'	3"	5	5'	6	9

$\longrightarrow$ Stable

1	2	3'	3	3"	5'	5	6	9

$\longrightarrow$ Unstable

Basic sorting algorithms

These two algorithms run in $\Theta(n^2)$ using worst case analysis

```
Selection sort(A[1..n])
for i = 1 to n − 1 do
    minj = i ; minx = A[i] ;
    for j = i + 1 to n do
        if A[j] < minx then
            minj = j ; minx = A[j] ;
    A[minj] = A[i] ; A[i] = minx ;
```

```
InsertionSort(A[1..n])
for (i = 2; i <= n; i++)
    v=A[i] ; j=i-1 ;
    while(j > 0 & A[j] > v)
            A[j+1]=A[j] ; j− ;
    A[j+1] = v ;
```

Merge Sort

Merge sort is a divide-&-conquer sorting algorithm. It works as follow :

- **Divide** the array to sort in half.
- **Recursively** sort each half.
- **Merge** the two sorted halves.

Pseudo-code of merge sort is as follows :

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

Merge function

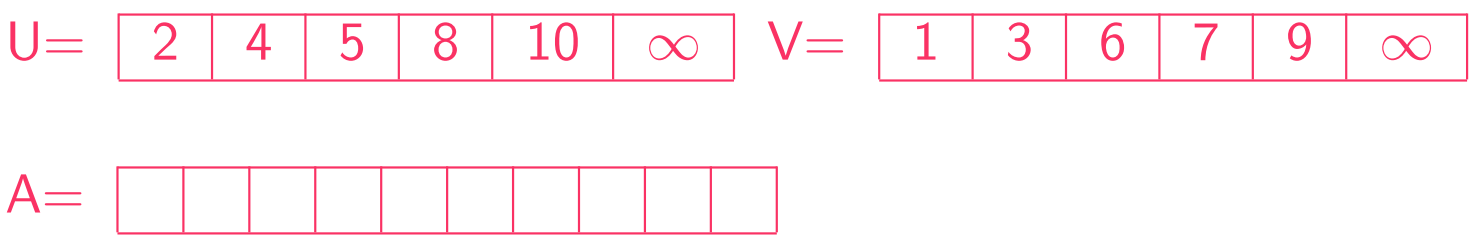
The merge function is a non-recursive component of merge sort which combines 2 sub-arrays already sorted

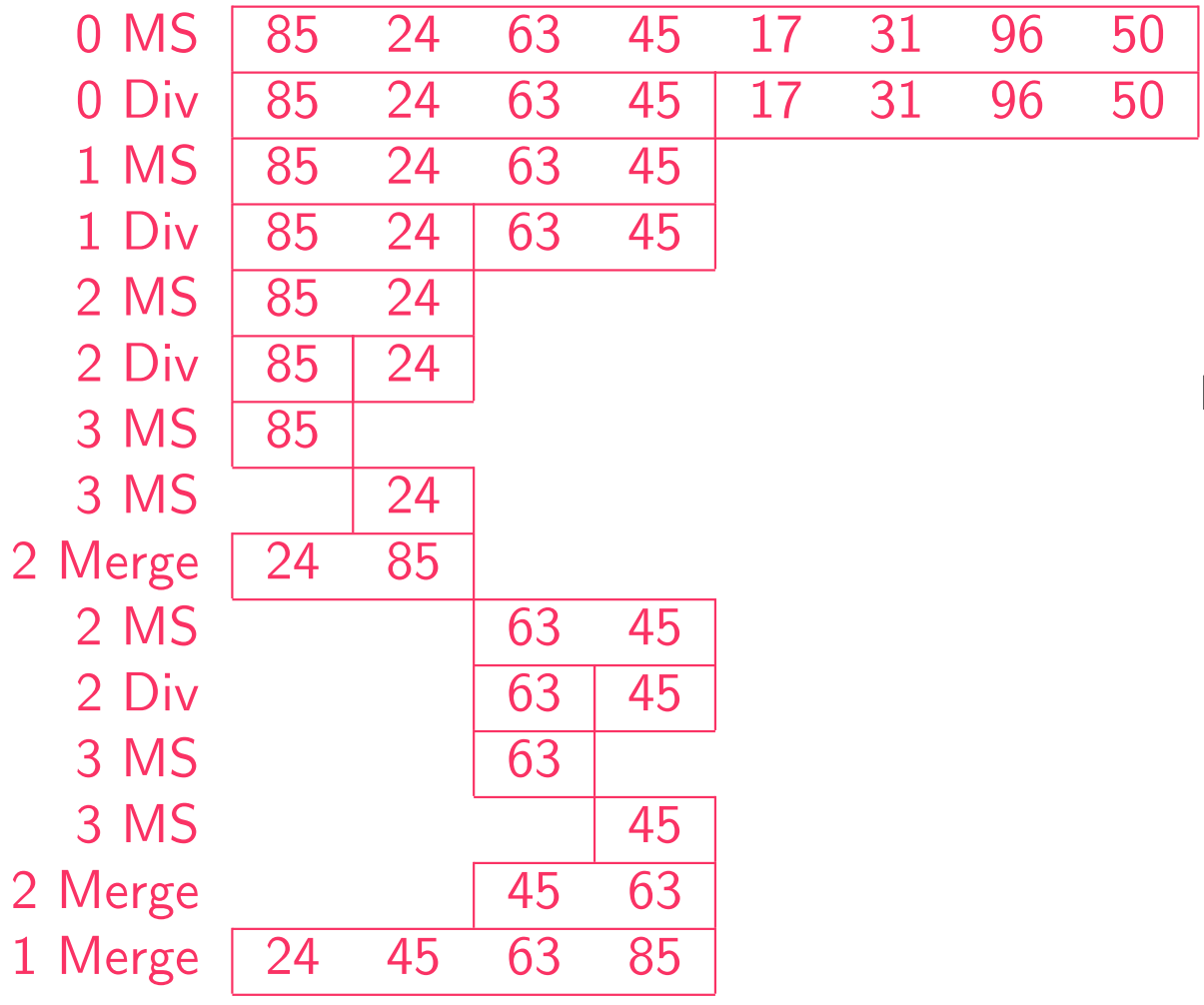
Takes in input 2 sorted sub-arrays U and V respectively of size m and n and return a sorted array A of size $m + n$

$Merge(U[1..m+1], V[1..n+1], A[1..m+n])$;
  $i = j = 1; U[m+1] = V[n+1] = \infty$;
  for $(k = 1; k \leq m + n; k++)$
    if $U[i] < V[j]$ **then**
      $A[k] = U[i]$; $i = i + 1$;
    else
      $A[k] = V[j]$; $j = j + 1$;

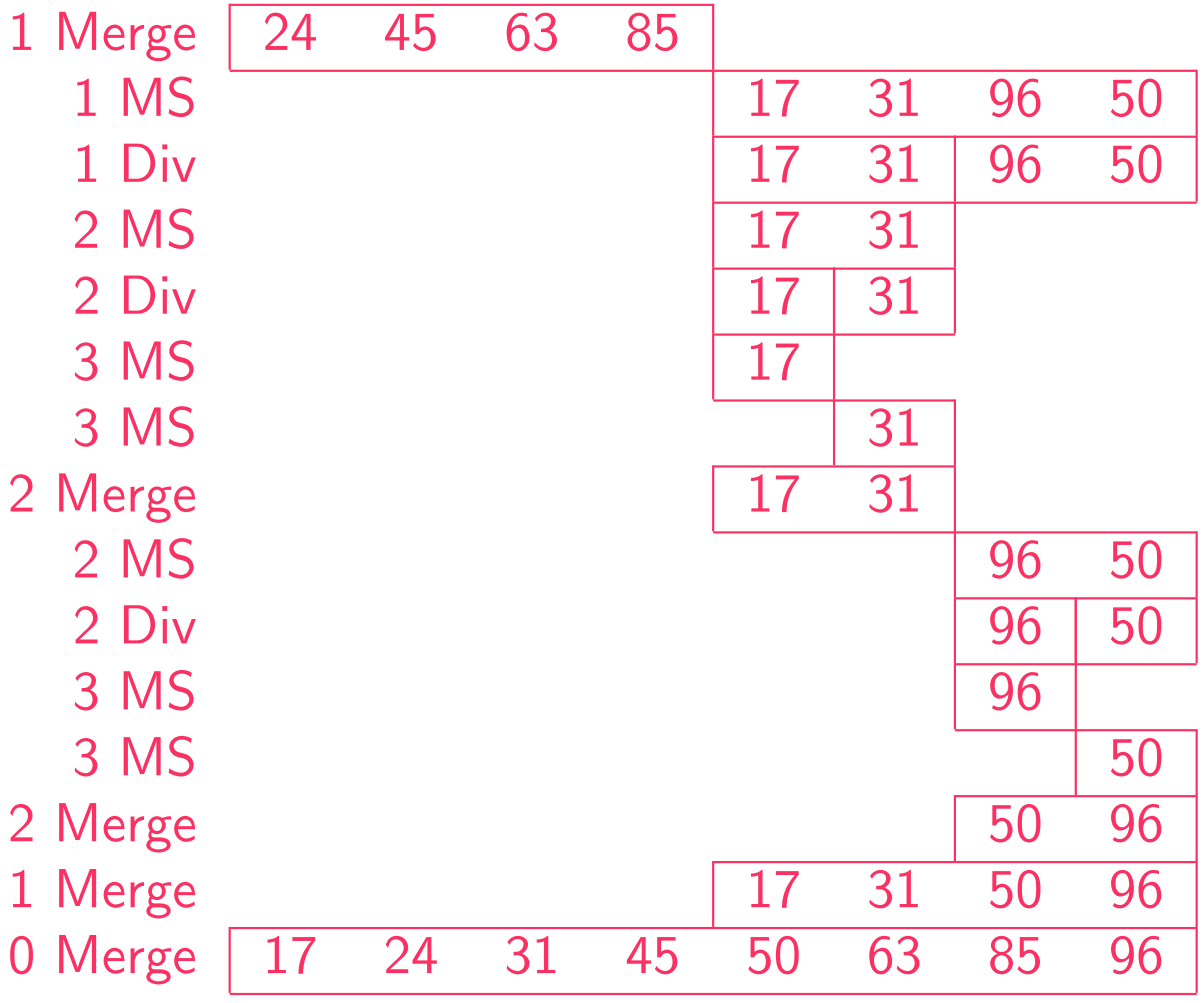
Merge algorithm

```
Merge(U[1..m+1], V[1..n+1], A[1..m+n]);
  i = j = 1; U[m+1] = V[n+1] = ∞;
  for (k = 1; k ≤ m+n; k++)
      if U[i] < V[j] then
          A[k] = U[i]; i = i+1;
      else
          A[k] = V[j]; j = j+1;
```





Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)


1 Merge
24 45 63 85
1 MS
17 31 96 50
1 Div
17 31 96 50
2 MS
17 31
2 Div
17 31
3 MS
17
3 MS
31
2 Merge
17 31
2 MS
96 50
2 Div
96 50
3 MS
96
3 MS
50
2 Merge
50 96
1 Merge
17 31 50 96
0 Merge
17 24 31 45 50 63 85 96

Divide-and-Conquer

Running time of merge sort

The running time of recursive algorithms cannot be analyzed in the same way as iterative algorithms

There are no loops, so summations to count the number of basic operations in loops do not apply for recursive algorithms

Rather recurrence relations will be used to express the number of times basic operations are executed in recursive algorithms

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

The recurrence for merge sort

The recurrence relation for merge sort is

$$\begin{aligned} T(n) &= T(\tfrac{n}{2}) + T(\tfrac{n}{2}) + O(n) \\ &= 2T(n/2) + O(n) \end{aligned}$$

which must be read as "the running time $T(n)$ of merge sort is equal to :

- the running time of solving two input instances of input size $\frac{n}{2}$ plus
- the running time of merging two sub-arrays of size n, $O(n)$"

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

Merge Sort

Mergesort($A[p..r]$)
  if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    Mergesort($A[p..q]$)
    Mergesort($A[q+1..r]$)
    Merge(A, p, q, r)

The code has two recursive calls, both are executed each time the function is entered

The size of the input is reduced by half each time a recursive call is made

The "dominant" non-recursive term is Merge() which runs $\Theta(n)$

Assuming n is a power of 2, the time $T(n)$ needed to solve an instance of side n is equal to the time to solve two sub-instances of size $\frac{n}{2}$ plus the time $\Theta(n)$ needed to compute Merge() :

$$T(n) = 2T(\frac{n}{2}) + n$$

Solving the recurrence for Merge sort

Computing $q = \lfloor \frac{p+r}{2} \rfloor$ cost $O(1)$. `Merge` is in $O(n)$, thus $m = 1$.

`Mergesort` does exactly 2 recursive calls, thus $a = 2$.

The input to the recursive calls have size $\frac{n}{2}$, thus $b = 2$.

The recurrence relation for merge sort is :

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ 2T(n/2) + O(n) & \text{otherwise} \end{cases}$$

Solving this recurrence using Master theorem, the case that applies is $a = b^m$.

Consequently, the running time of merge sort
$T(n) \in \Theta(n^m \log n) = \Theta(n \log n)$.

Quicksort

Another recursive sorting algorithm

An array A[p..r] of integers is partitioned into two subarrays A[p..q] and A[q+1..r]

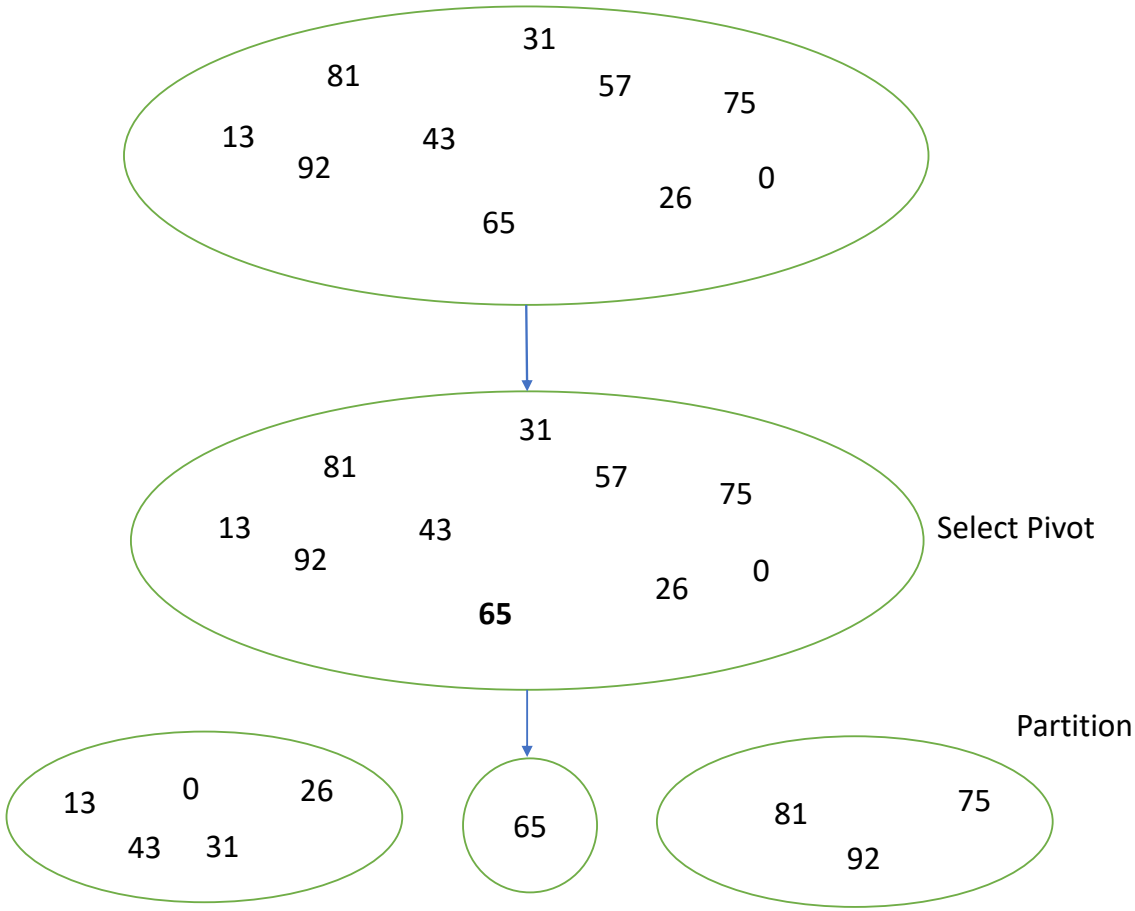
- such that the elements in A[p..q] are less than all elements in A[q+1..r]

The subarrays are recursively sorted by calls to quicksort

Quicksort Code

```
Quicksort(A, p, r)
  if (p < r) /* the array has at least 2 elements */
    q = Partition(A, p, r); /* q is the index in A of the pivot element */
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
```

Quicksort idea



After partition, the pivot is in its right position in a sorted array. Quicksort is recursively applied to the next two subarrays.

Partition

```
Quicksort(A, p, r)
  if (p < r)
    q = Partition(A, p, r);
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
```

- All the action takes place in the partition() function which rearranges the subarray
- End result : Two subarrays, all values in first subarray $<$ all values in second
- Returns the index of the "pivot" element separating the two subarrays

Partition code

Partition() takes in input the array A and the section between indexes p and r to partition

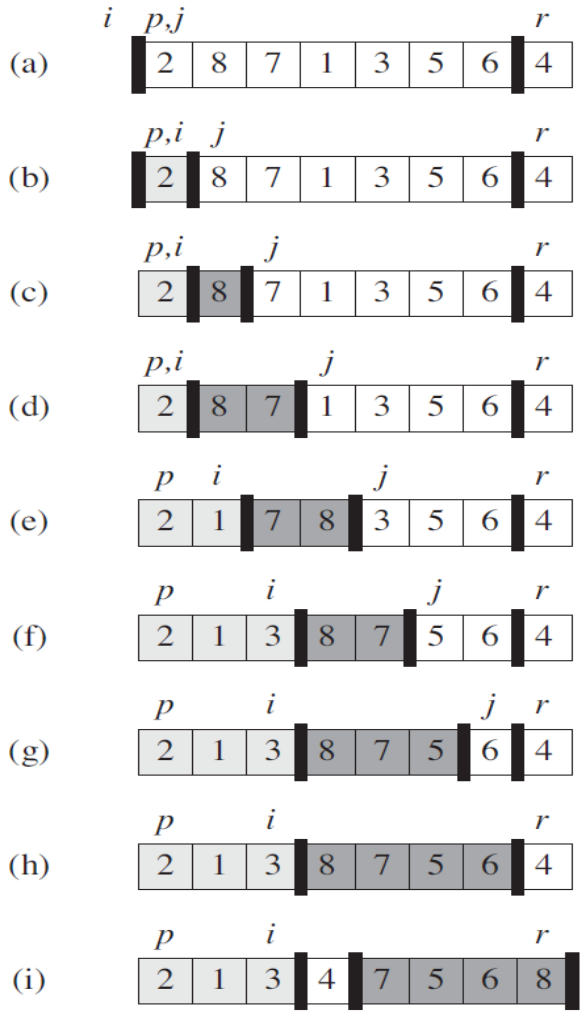
x is the pivot element, in this code the last element in the array is selected as pivot

Partition() returns the index in A of the pivot element once partition is completed

The position of the pivot element will never be part of a sub-array to partition, it is at its right position in the sorted array

```
Partition(A, p, r)
    x = A[r];
    i = p - 1;
    for (j = p; j < r; j++)
       if (A[j] ≤ x)
          i = i +1;
          swap(A[i],A[j]);
    swap(A[i+1],A[r]);
    return i+1;
```

Example with running partition()




```
Partition(A, p, r)
    x = A[r] ;
    i = p - 1 ;
    for (j = p ; j < r ; j++)
        if (A[j] ≤ x)
            i = i +1 ;
            swap(A[i],A[j]) ;
    swap(A[i+1],A[r]) ;
    return i+1 ;
```

Performance of Quicksort

Quicksort calls itself on 2 sub-arrays with likely different sizes. The asymptotic running time of Quicksort depends on how these two sub-arrays are balanced

$$T(n) = T(L) + T(R) + O(n)$$

where L and R are the number of elements in each sub-array.

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

Worst case analysis of Quicksort

Occurs when the subarrays are completely unbalanced for each recursive call. Has 0 element in one subarray and $n-1$ elements in the other subarray. The recurrence is

$$T(n) = T(n-1) + T(0) + O(n) \quad (1)$$
$$= T(n-1) + O(n) \quad (2)$$

This recurrence cannot be solved using the Master theorem, but can be solved using the substitution method or the recursion tree method (will see them later)

Intuitively, since only element of the array is sorted properly at each recursive call (the pivot element), the input size reduces only by 1, thus $n-1$ recursive calls are made.

Each recursive call entails a call to partition() which runs in $O(n)$, thus the worst case running time of Quicksort is $O(n^2)$

```
Quicksort(A, p, r)
  if (p < r)
    q = Partition(A, p, r);
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
```

Best case of Quicksort

Occurs when the subarrays are completely balanced every time.

Each subarray has $\leq n/2$ elements.

The recurrence is

$$T(n) = T(n/2) + T(n/2) + O(n) \quad (3)$$
$$= 2T(n/2) + O(n) \quad (4)$$

This recurrence can be solved using the master theorem method.

As we already know, the solution of this recurrence is
$T(n) \in \Theta(n \log n)$

Quicksort well designed

It is easy to avoid the worst case performance for quicksort.

Your textbooks introduces two possible solutions to this issue :

- Randomize the input, or
- Pick a random pivot element

This solve the bad worst-case behavior because no particular input can be chosen that makes quicksort runs in $O(n^2)$

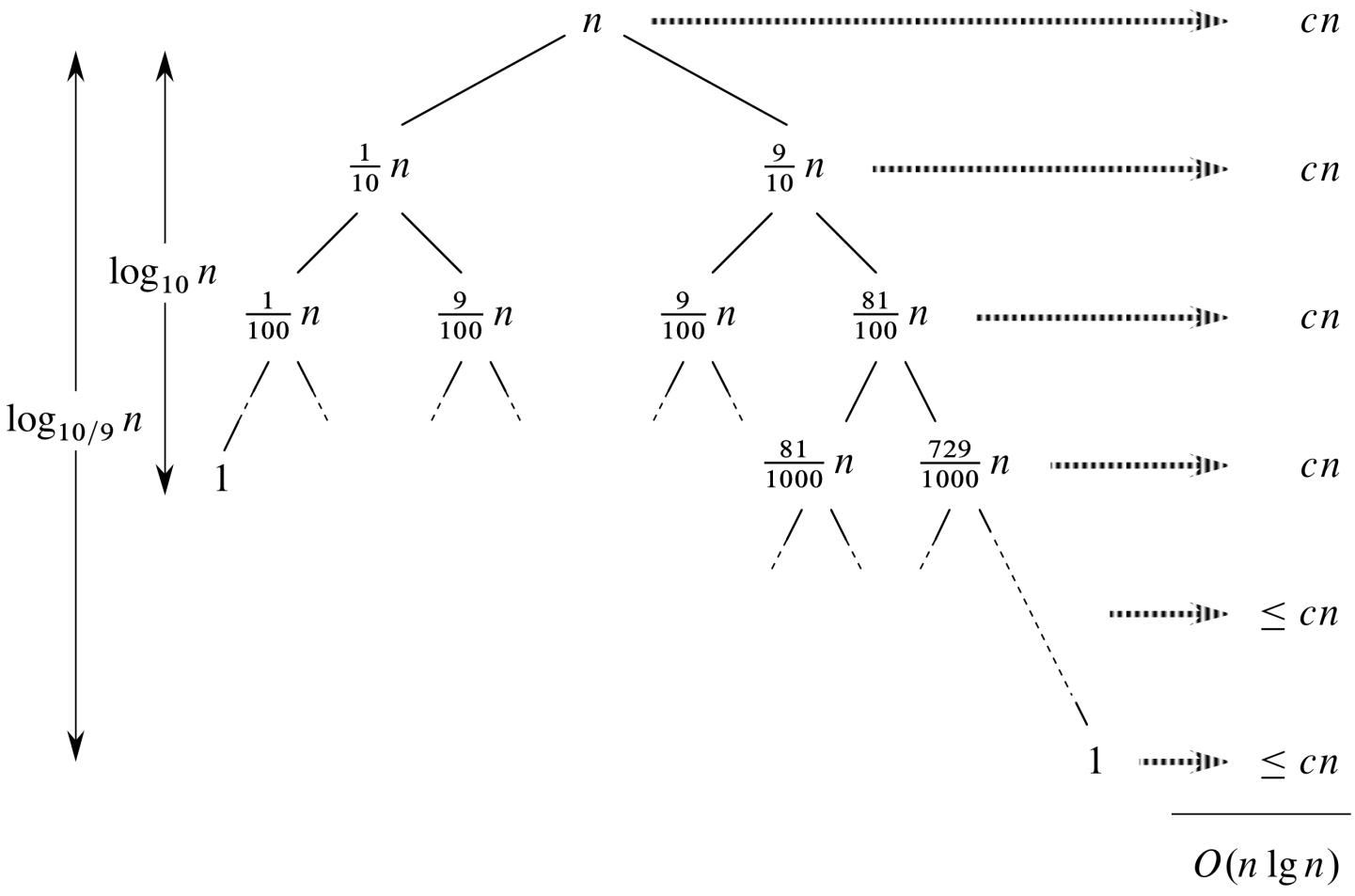
Analyzing quicksort : effects of partitioning

- Imagine that Partition always produces a 9-to-1 split.
- Get the recurrence :

$$\begin{aligned} T(n) &\leq T(9n/10) + T(n/10) + O(n) \\ &= O(n \log n) \end{aligned}$$

- See the recurrence tree in the next slide, the number of levels of the partitioning is $\log_{\frac{10}{9}} n$, local work at each level cost no more than cn, therefore quicksort runs in $O(n \log n)$ even for this bad partitioning

Analysing quicksort : effects of partitioning



Analyzing Quicksort : Average Case

We need to consider all the possible inputs of size n that can be submitted to a quicksort routine.

These are the n possible ways that "partition" can split an array into 2 sub-arrays : 0 : n-1 ; 1 : n-2 ; 2 : n-3 ; ..., n-1 : 0

Assumptions : all splits equally likely to occur, thus, each split has probability $1/n$ to occur

The recurrence relation is the following :

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

Since $\sum_{k=0}^{k=n-1} T(k) = \sum_{k=0}^{k=n-1} T(n-1-k)$, $T(n)$ can be written as

$$\frac{2}{n} \sum_{k=0}^{k=n-1} T(k) + O(n)$$

Analyzing Quicksort : Average Case

- We can solve this recurrence using the substitution method
- Guess the answer T(n) = O(n lg n)
- Assume that the inductive hypothesis holds
  - $T(n) \leq cn \log n$ for some constant c
- Substitute it in for some value $<$ n
  - The value k in the recurrence

  Prove that it follows for n

Analyzing Quicksort : Average Case

$$
\begin{aligned}
T(n) &= \frac{2}{n}\sum_{k=0}^{k=n-1} T(k) + O(n) \\
&\leq \frac{2}{n}\sum_{k=0}^{k=n-1} (ck \log k) + O(n) \text{ inductive hypothesis} \\
&= \frac{2c}{n}\sum_{k=0}^{k=n-1} k \log k + O(n) \\
&\leq \frac{2c}{n}\left(\frac{1}{2}n^2 \log n - \frac{1}{8}n^2\right) + O(n)
\end{aligned}
$$

The summation $\sum_{k=1}^{k=n-1} k \log k$ can be bound above by $\frac{1}{2}n^2 \log n - \frac{1}{8}n^2$ but this is not proved here.

Analyzing Quicksort : Average Case

$$\begin{aligned} T(n) &\leq \frac{2c}{n}\left(\frac{1}{2}n^2 \log n - \frac{1}{8}n^2\right) + O(n) \\ &= cn \log n - \frac{c}{4}n + O(n) \\ &= cn \log n + \left(O(n) - \frac{c}{4}n\right) \\ &= cn \log n \text{ for } c \text{ such that } \frac{cn}{4} > O(n) \end{aligned}$$

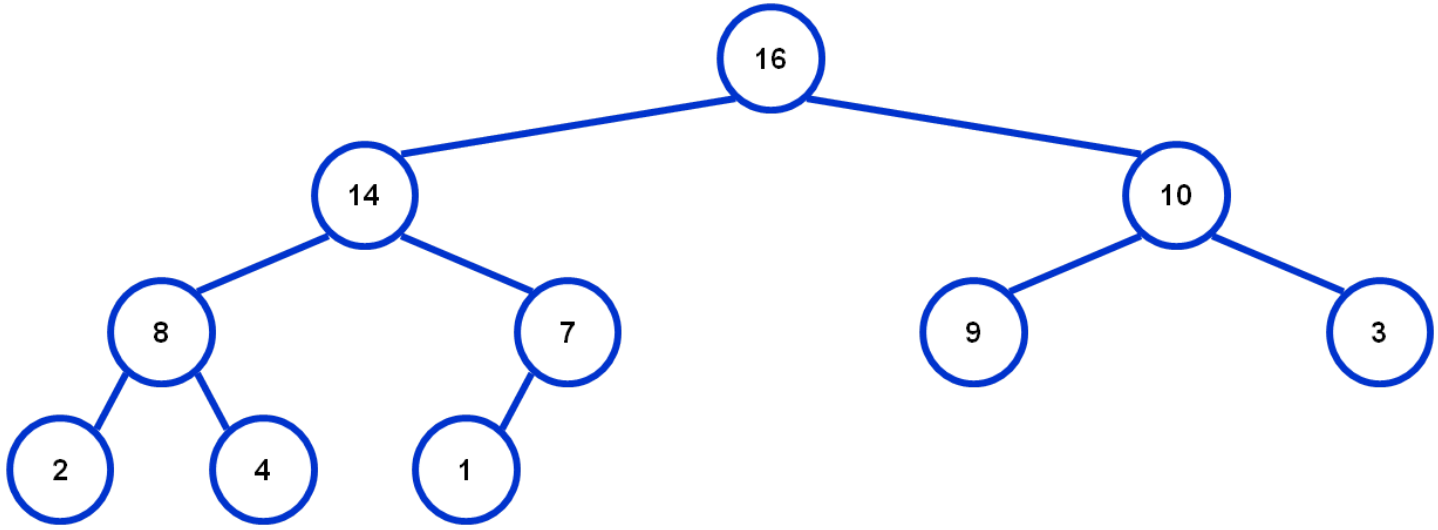
Thus $T(n) \in O(n \log n)$

Sorting

- So far we have seen different sorting algorithms such as selection sort, and insertion, and merge sort and quicksort
  - Merge sort runs in $O(n \log n)$ both in best, average and worst-case
  - Insertion/selection sort run in $O(n^2)$, but insertion sort is fast when array is nearly sorted, runs fast in practice
- Next on the agenda : Heapsort
- Prior to describe heapsort, we introduce the heap data structure and operations on that data structure

Heap : definition

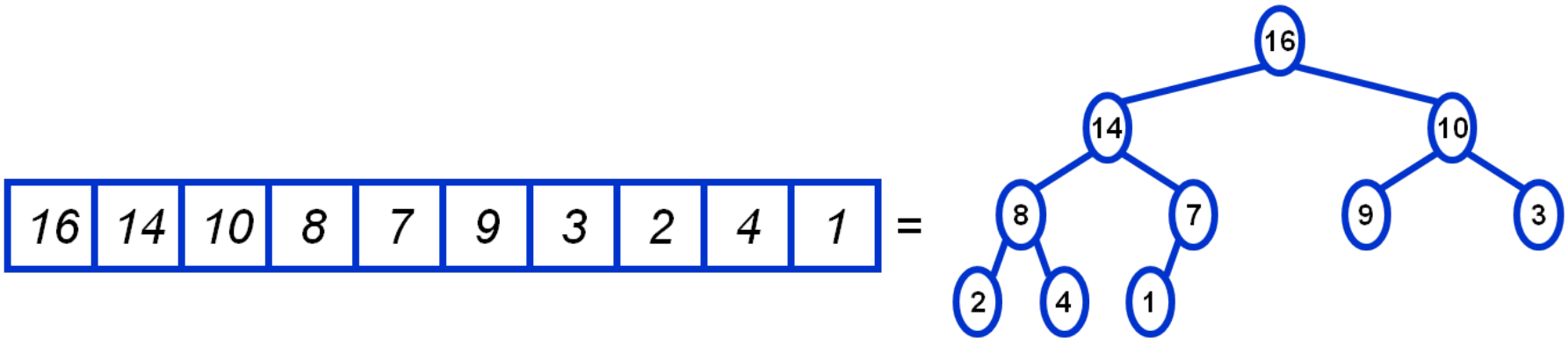
- A heap is a complete binary tree



- *Binary* because each node has at most two children
- *Complete* because each internal node, except possibly at the last level, has exactly two children

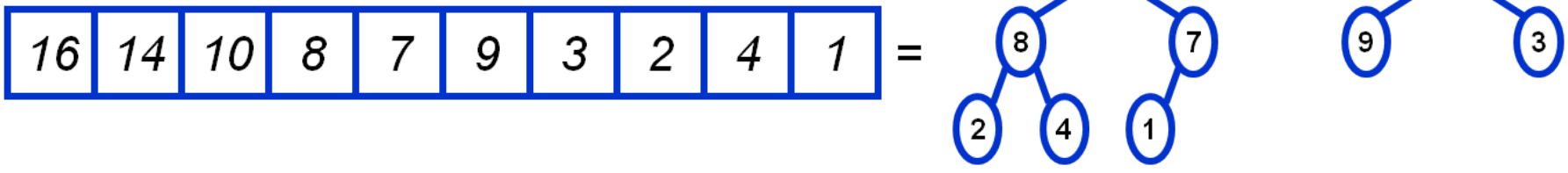
Heaps

- In practice, heaps are usually implemented as arrays



Heaps

- How to represent a complete binary tree as an array :
  - The root node is $A[1]$
  - Ordering nodes per levels starting at the root, and from left to right in a same level, then node i is $A[i]$
  - The parent of node i is $A[\lfloor i/2 \rfloor]$
  - The left child of node i is $A[2i]$
  - The right child of node i is $A[2i + 1]$



Referencing Heap Elements

Assuming the array index starts at 1 :

function Parent(i)
  return $\lfloor i/2 \rfloor$;

function Left(i)
  return $2 \times i$;

function Right(i)
  return $2 \times i + 1$;

The Heap Property

- Heaps must satisfy the following relation :

$$A[Parent(i)] \geq A[i] \text{ for all nodes } i > 1$$

- In other words, the value of a node is at most the value of its parent

Heap Height

- The height of a node in the tree = the number of edges on the longest downward path to a leaf
- The height of a tree = the height of its root
- What is the height of an n-element heap? Why?
- Heap operations take at most time proportional to the height of the heap

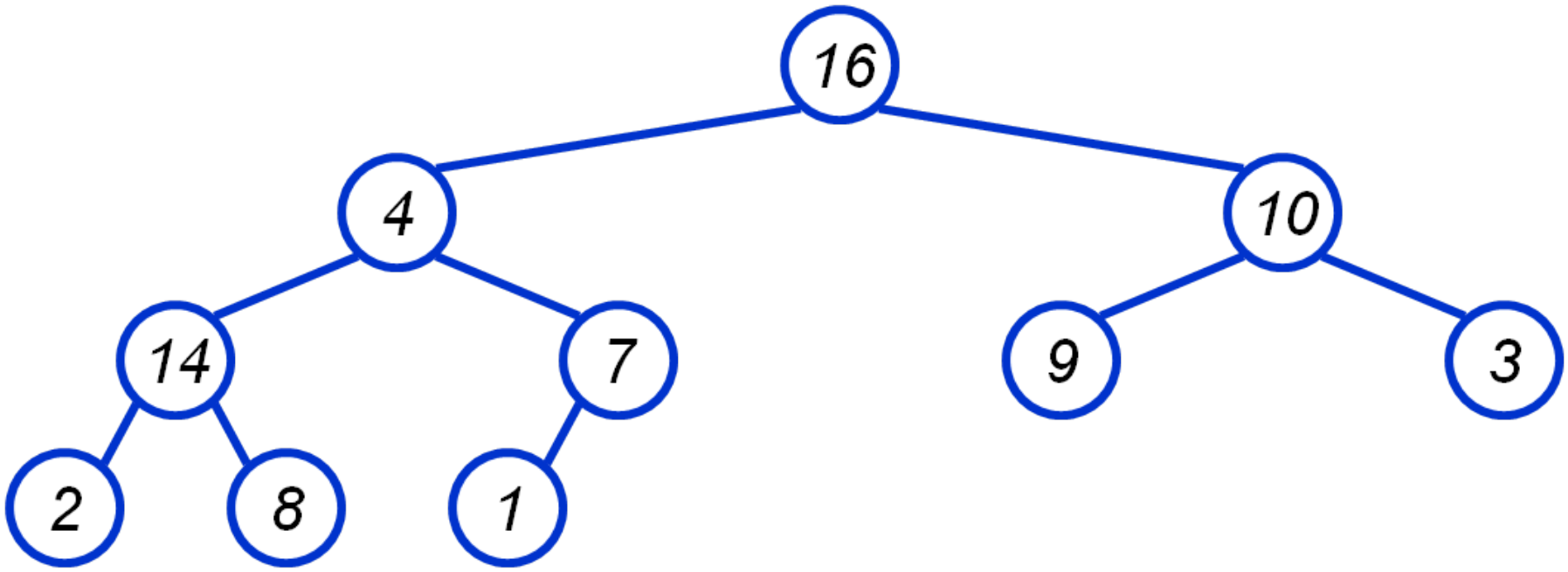
Heap Operations

There are two main heap operations : heapify and buildheap.

Heapify() : restore the heap property :

- Consider node i in the heap with children l and r
- Nodes l and r are each the root of a subtree, each assumed to be a heap
- Problem : Node i may violate the heap property
- Solution : let the value of node i "float down" in one of its two subtrees until the heap property is restored at node i

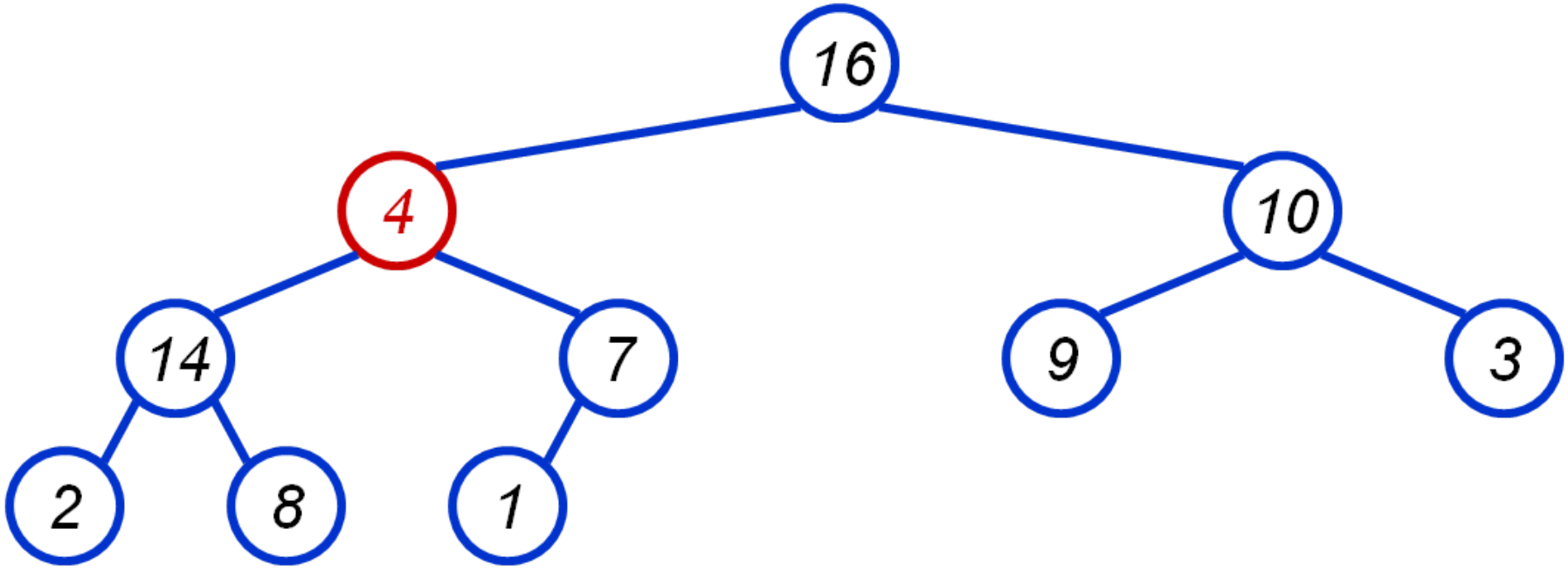
Heapify() Example



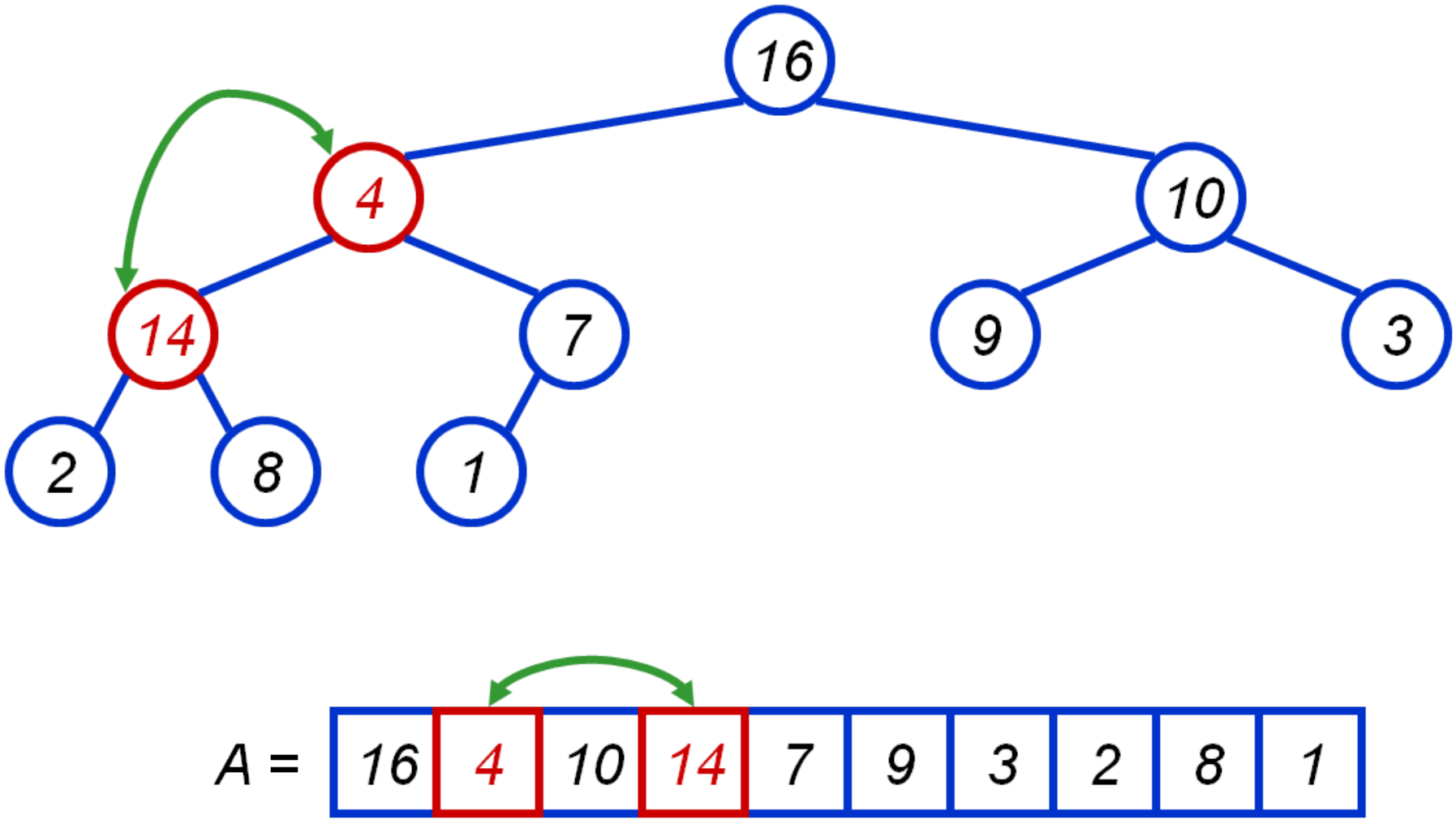
$A =$

16	4	10	14	7	9	3	2	8	1

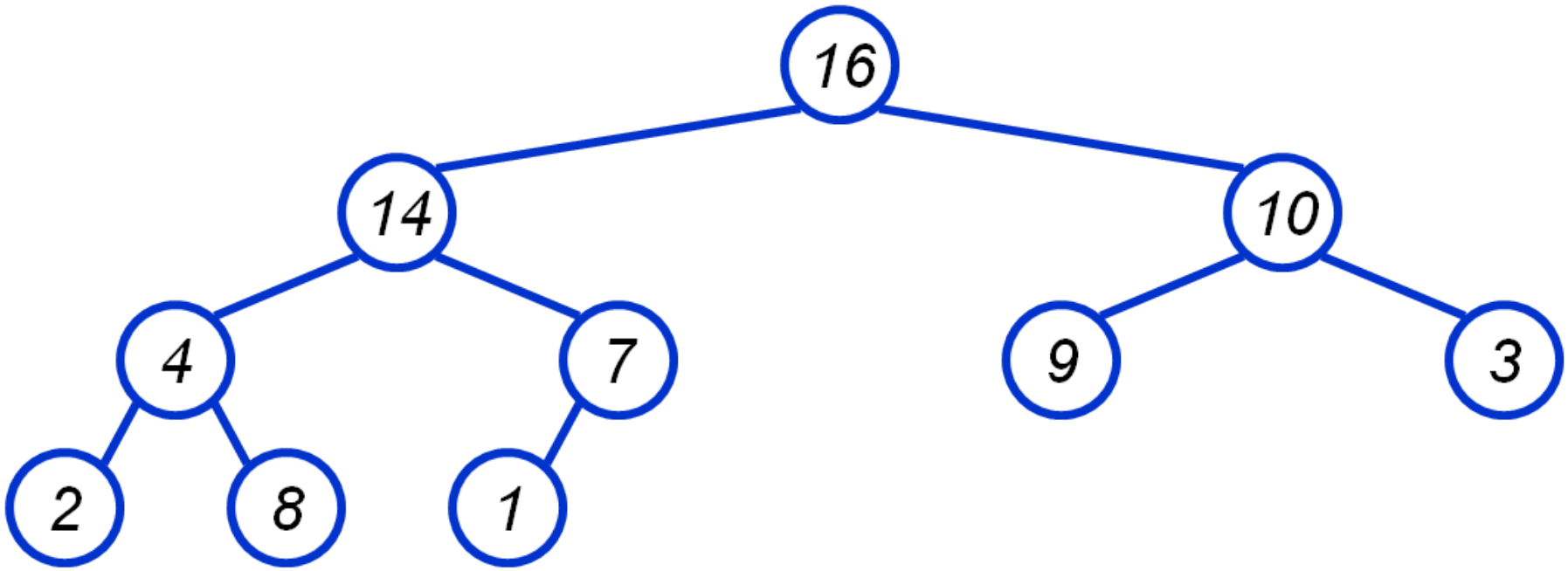
Heapify() Example



Heapify() Example



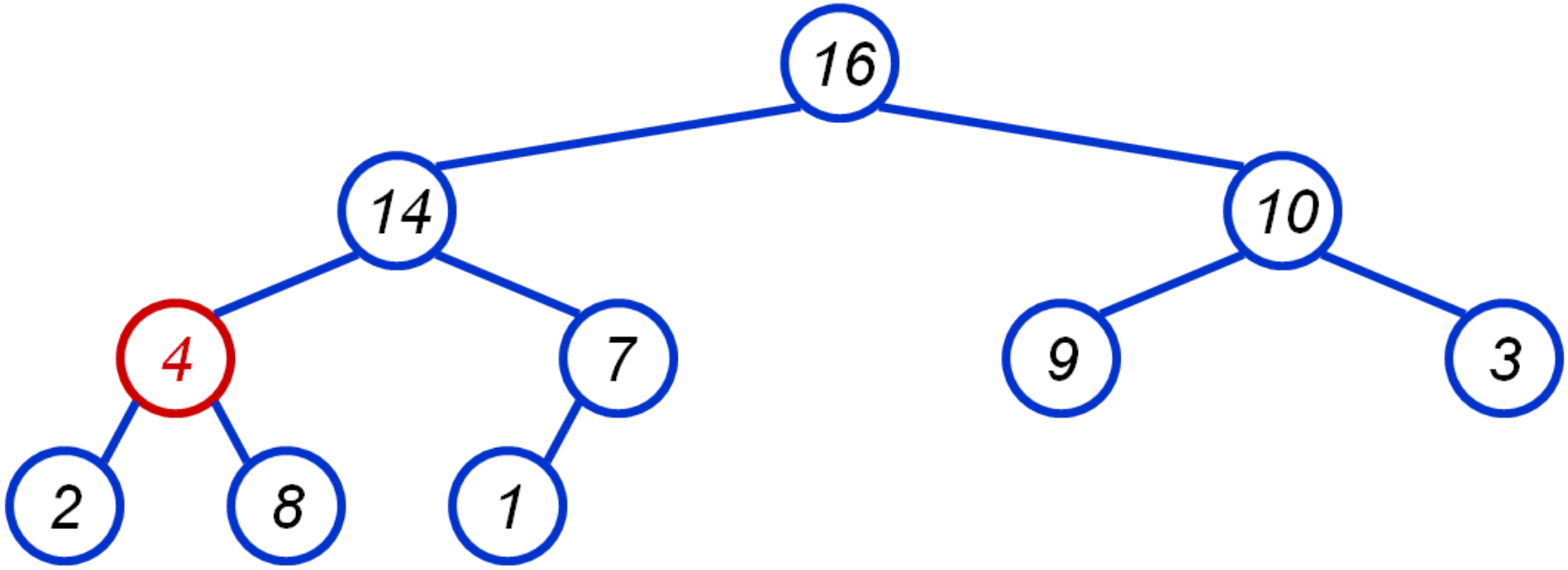
Heapify() Example

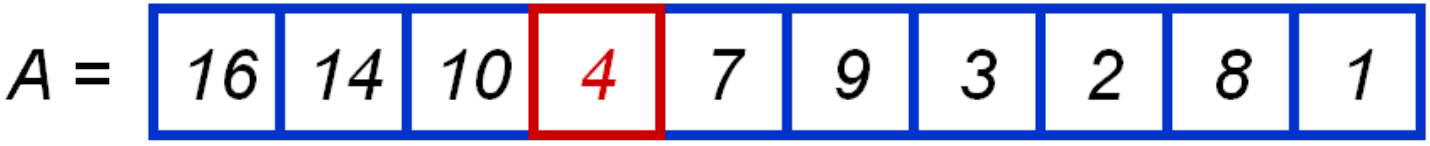


$A =$

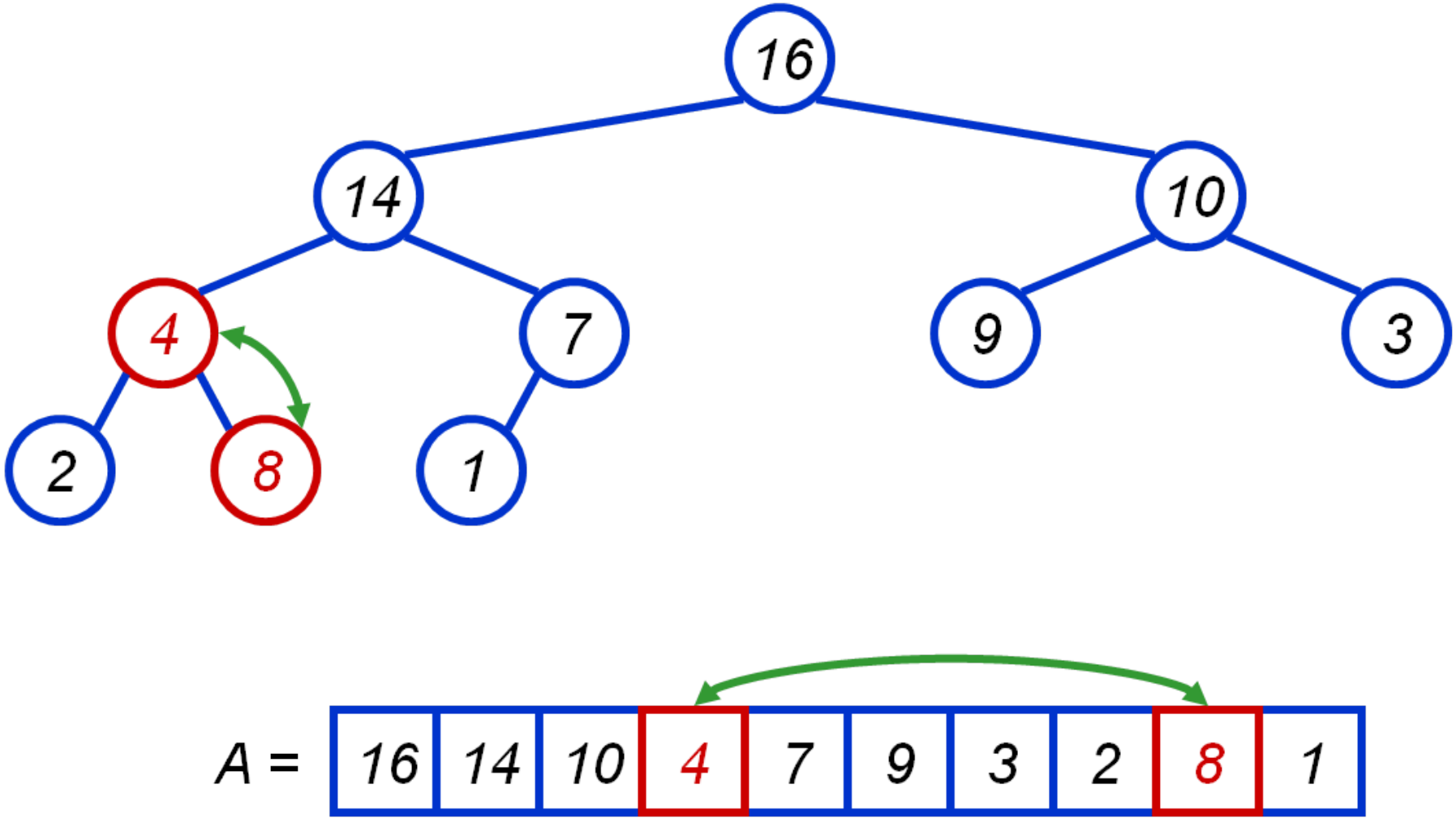
16	14	10	4	7	9	3	2	8	1

Heapify() Example

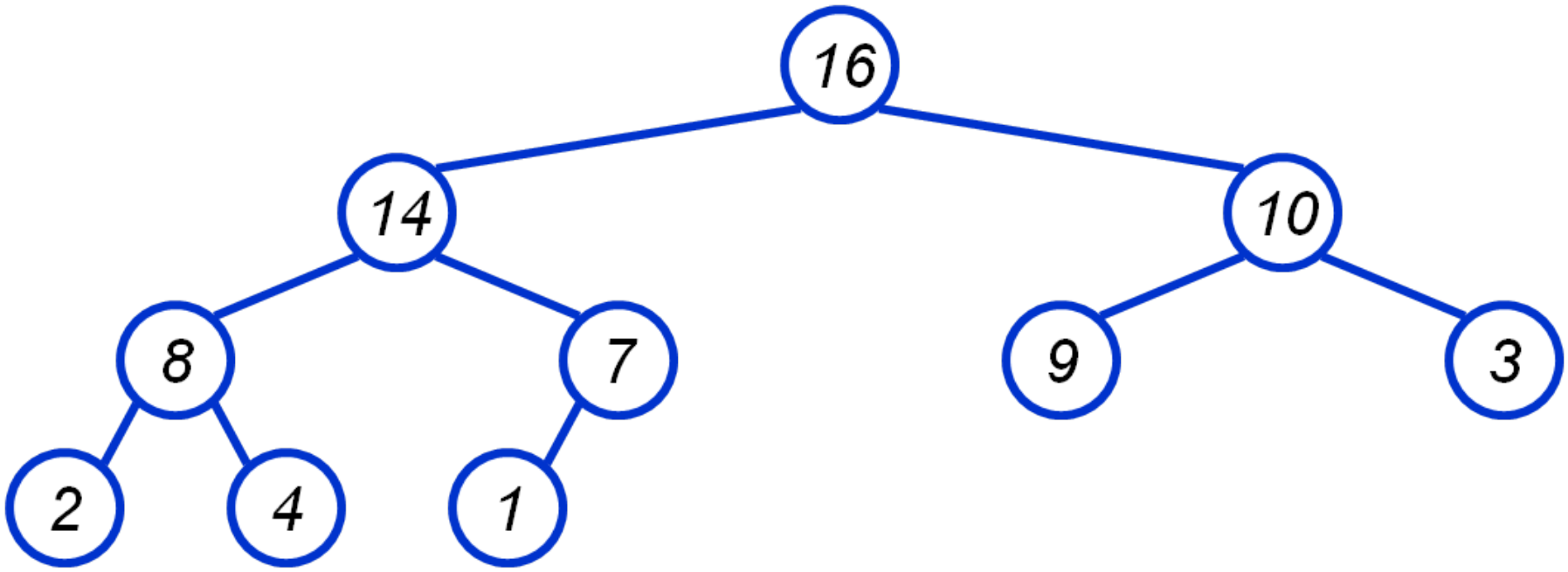




Heapify() Example



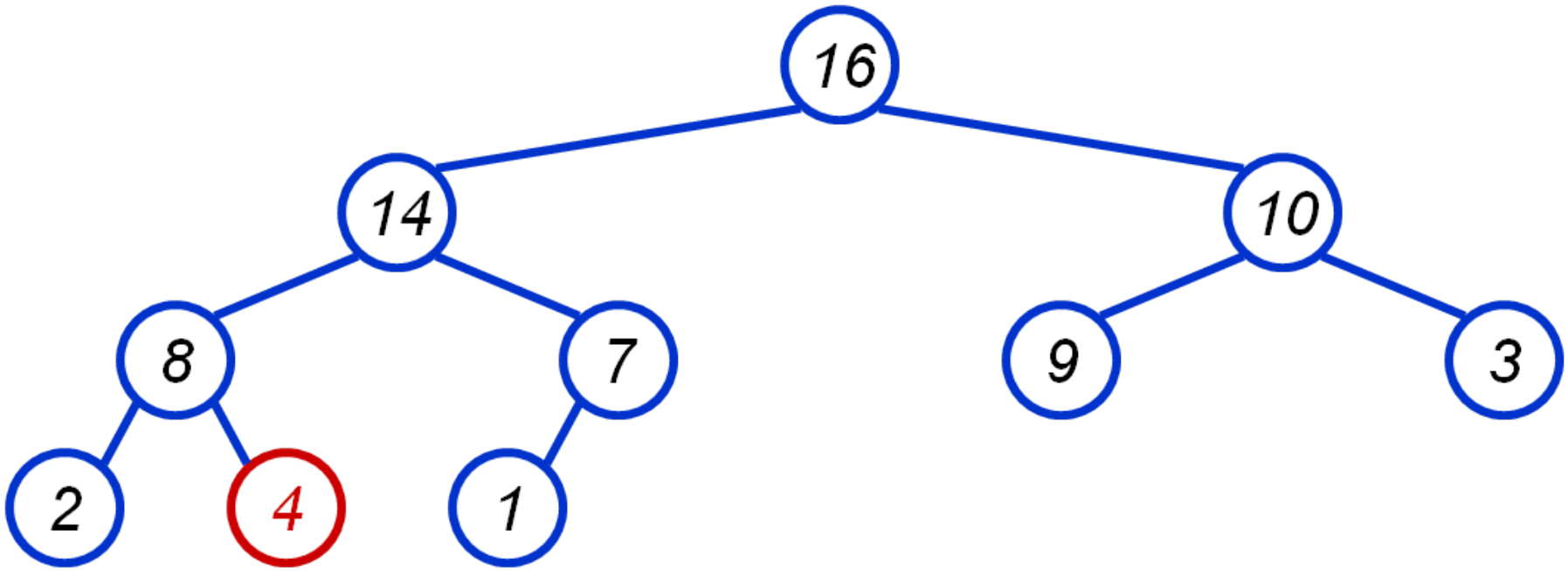
Heapify() Example



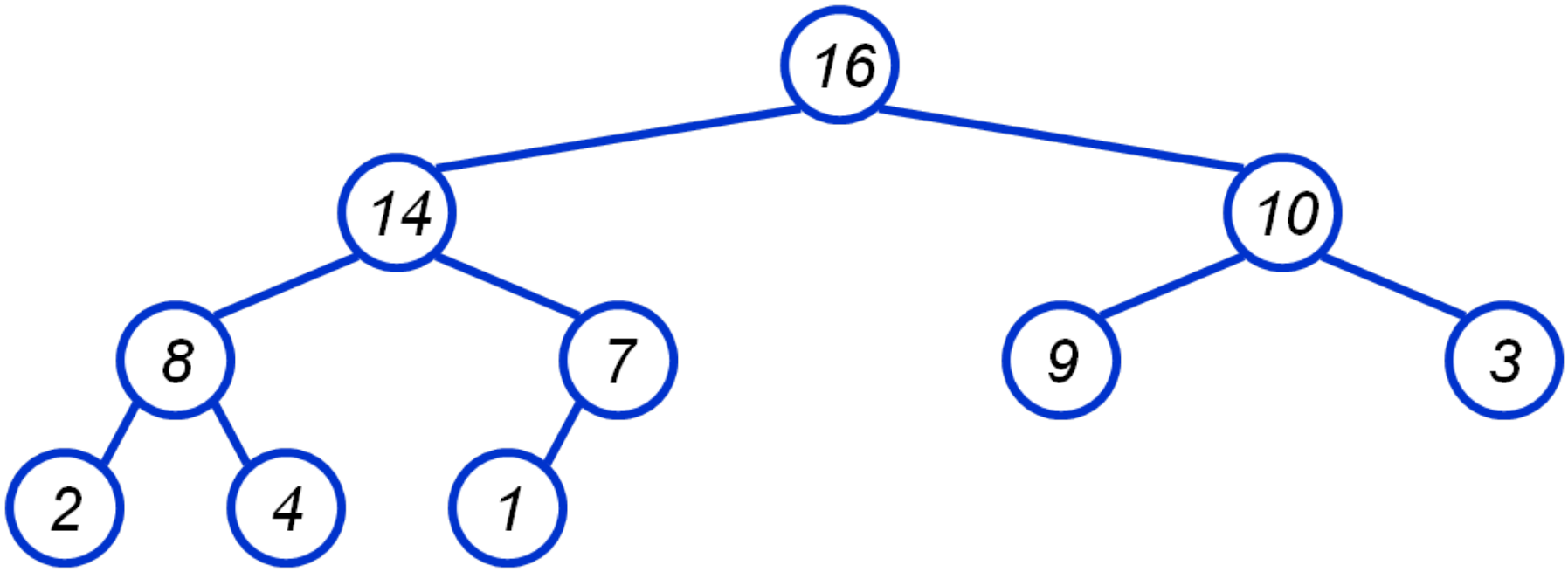
$A =$

16	14	10	8	7	9	3	2	4	1

Heapify() Example



Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

Algorithm for heapify

An array $A[]$, where heap_size(A) returns the dimension of A

```
Heapify(A, i)
  l = Left(i) ; r = Right(i) ;
  if (l ≤ heap_size(A) & A[l] > A[i])
    largest = l ;
  else
    largest = i ;
  if (r ≤ heap_size(A) & A[r] > A[largest])
    largest = r ;
  if (largest != i)
    Swap(A, i, largest) ;
    Heapify(A, largest) ;
```

Example : heapify

This array $A = [23, 11, 14, 9, 13, 10, 1, 5, 7, 12]$ is not a heap as Parent(5) $= \lfloor \frac{i}{2} \rfloor = 2$, A[2] $= 11$ in the array, which violates the max-heap property as A[5] $= 13$ is greater than A[2].

Call heapify(A,2) which swap A[2] with $Right(2) = 2i + 1 = A[5]$

$$A = [23, 13, 14, 9, 11, 10, 1, 5, 7, 12]$$

Left(5) $= 10$ and $A[10] > A[5]$ which violates the heap property $A[Parent(i)] \geq A[i]$. Thus heapify continues, swap A[5] with $Left(5) = 2i = A[10]$

$$A = [23, 13, 14, 9, 12, 10, 1, 5, 7, 11]$$

Analyzing Heapify()

- Number of basic operations performed before calling itself ?
- How many times can Heapify() recursively call itself ?
- What is the worst-case running time of Heapify() on a heap of size n ?

Analyzing Heapify()

- The work done in each line of Heapify() is $\Theta(1)$ except for the recursive call
- If the heap at i has n elements, how many elements can the subtrees at l or r have? Answer : at most $\lfloor 2n/3 \rfloor$ (worst case : bottom row 1/2 full)
- So time taken by Heapify() is given by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1)$$

- Restricted Master Theorem applies : $a = 1$, $b = 3/2$, $m = 0$. We have $a = b^m = \frac{3}{2}^0 = 1$, which corresponds to case $\Theta(n^m \log n) = \Theta(n^0 \log n) = \Theta(\log n)$, thus

$$T(n) \in \Theta(\log n)$$

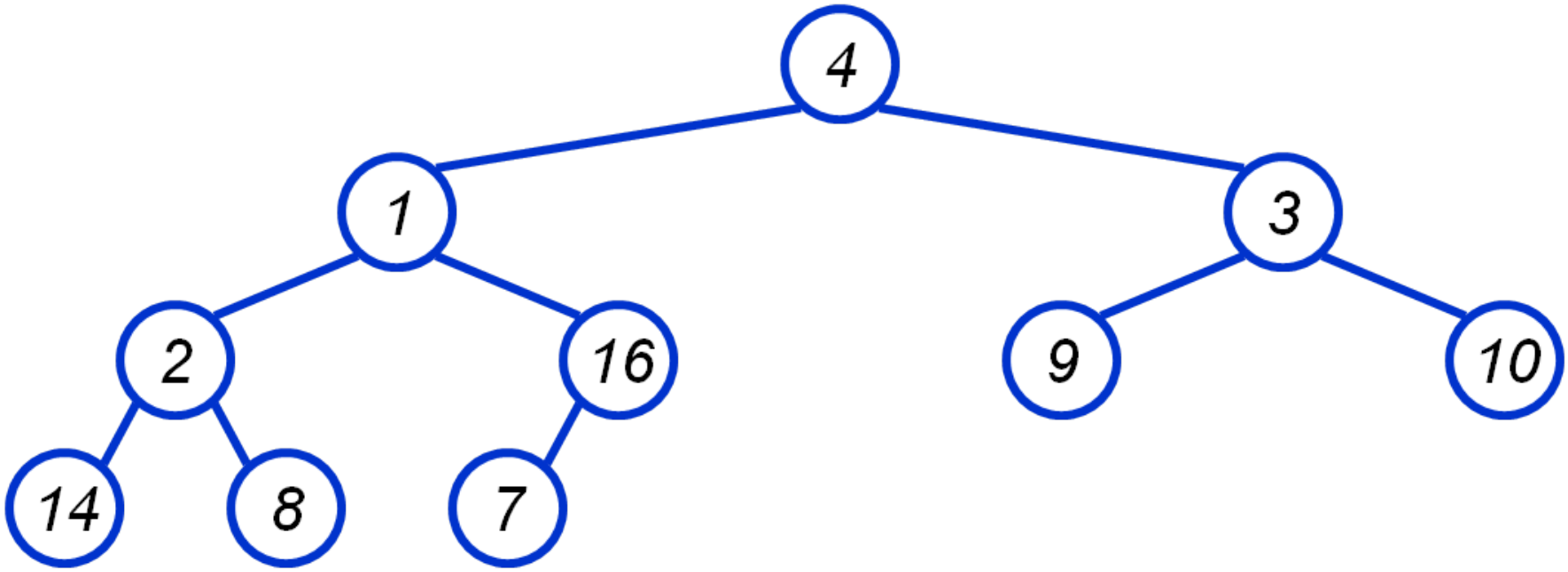
Heap Operations : BuildHeap()

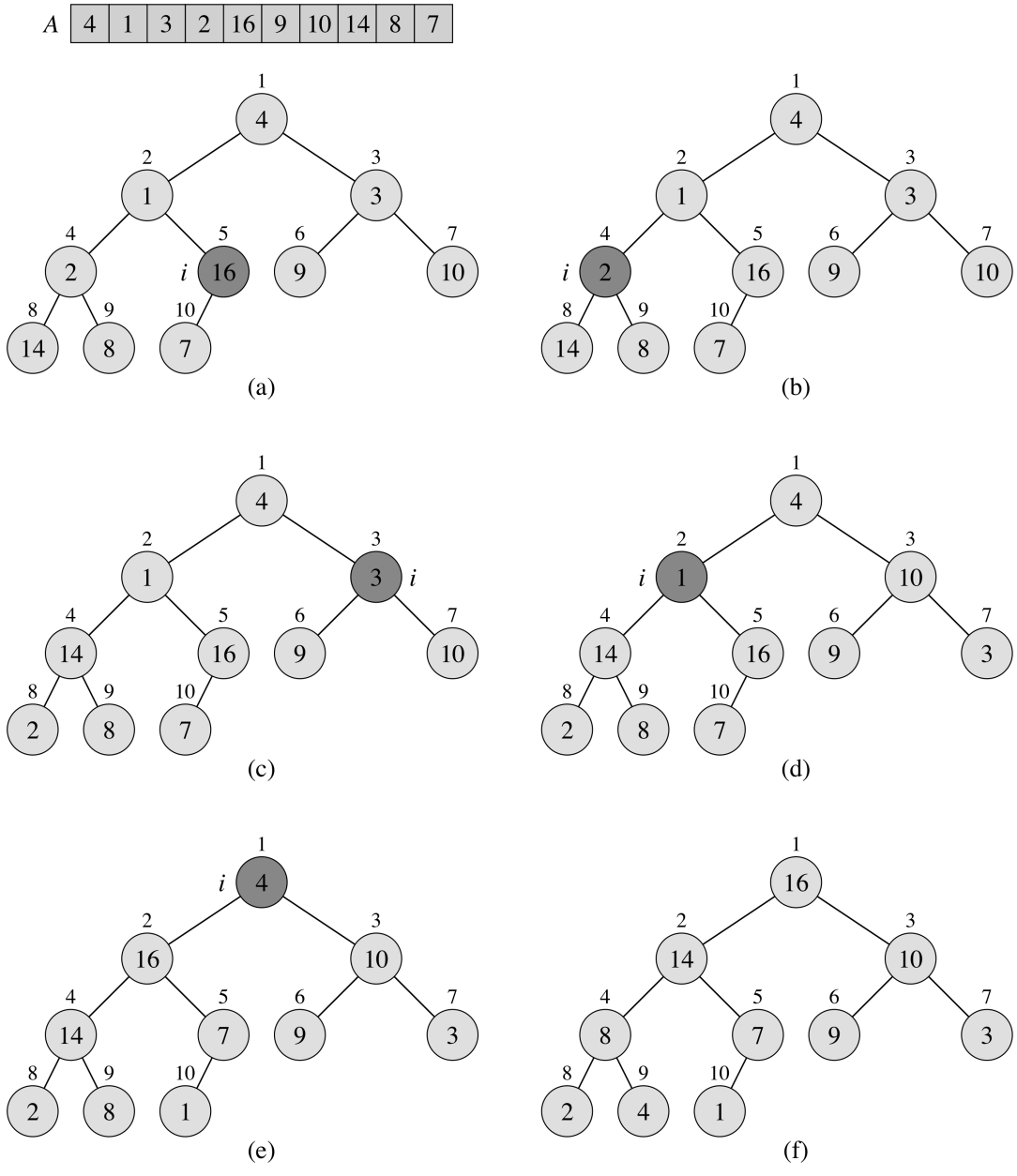
- We can build a heap in a bottom-up manner by running Heapify() on successive subarrays
  - Note : for array of length n, all elements in range $A[(\lfloor n/2 \rfloor + 1)..n]$ are heaps (Why ?)
  - Walk backwards through the array from $\lfloor n/2 \rfloor$ to 1, calling Heapify() on each node.
- given an unsorted array A, make A a heap

```
BuildHeap(A)
  heap_size(A) = length(A);
  for (i = ⌊length(A)/2⌋ downto 1)
    Heapify(A, i);
```

BuildHeap() Example

Work through example $A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$




A
4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)
(e)
(f)

BuildHeap : a second example

Run the algorithm $BuildHeap(A)$ on the array $A = [5, 3, 17, 10, 84, 19, 6, 22, 9]$

Find $i = \lfloor \frac{length(A)}{2} \rfloor = \lfloor \frac{9}{2} \rfloor = 4$, the entry in A where BuildHeap starts

BuildHeap starts at $A[4]$, from which heapify is run. Left(4) $= 8$, A[8] $= 10$; Right(4) $= 9$, A[9] $= 9$, so nothing to change

Next $A[3] = 17$, Left(3) $= 6$, A[6] $= 19$; Right(3) $= 7$, A[7] $= 6$. $A[6] > A[3]$, so heapify is needed on $A[3]$, yielding
$A[5, 3, 17, 22, 84, 19, 6, 10, 9]$
$A[5, 3, 19, 22, 84, 17, 6, 10, 9]$

Next $A[2] = 3$, and so on

$A[5, 84, 19, 22, 3, 17, 6, 10, 9]$
$A[84, 5, 19, 22, 3, 17, 6, 10, 9]$
$A[84, 22, 19, 5, 3, 17, 6, 10, 9]$
$A[84, 22, 19, 10, 3, 17, 6, 5, 9]$

Analyzing BuildHeap()

- Each call to Heapify() takes $O(\log n)$ time
- There are $O(n)$ such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is $O(n \log n)$
  - Is this a correct asymptotic upper bound ?
  - Is this an asymptotically tight bound ?
- A tighter bound is $O(n)$

Analyzing BuildHeap() : Tight

Heap-properties of an n-element heap

- Height $= \lfloor \log n \rfloor$
- At most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of any height h
- The time for Heapify on a node of height h is $O(h)$

$$\begin{aligned}
\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) &= O(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}) \\
&= O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) \\
&= O(n)
\end{aligned}$$

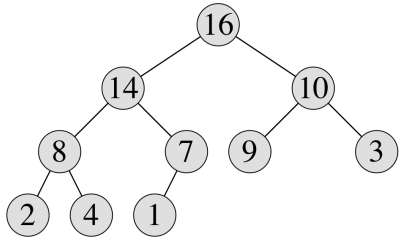
Analyzing BuildHeap() : Tight

$$
\begin{aligned}
\sum_{h=0}^{\infty} \frac{h}{2^h} &= \sum_{h=0}^{\infty} h(\frac{1}{2})^h \\
&= \sum_{h=0}^{\infty} hx^h \text{ where } x = \frac{1}{2} \\
&= \frac{1/2}{(1-\frac{1}{2})^2} \text{ the closed form of } \sum_{h=0}^{\infty} h(\frac{1}{2})^h \\
&= 2
\end{aligned}
$$

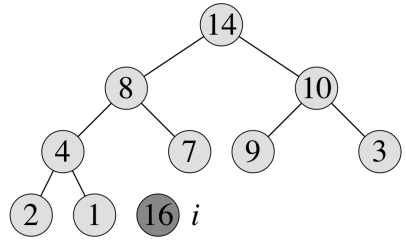
Heapsort

- Given BuildHeap(), a sorting algorithm is easily constructed :
  - Maximum element is at A[1]
  - Swap A[1] with element at A[n], A[n] now contains correct value
  - Decrement heap_size[A]
  - Restore heap property at A[1] by calling Heapify()
  - Repeat, always swapping A[1] for A[heap_size(A)]

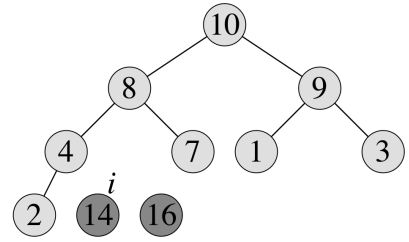
```
Heapsort(A)
  BuildHeap(A);
  for (i = length(A) downto 2)
    Swap(A[1], A[i]);
    heap_size(A) = heap_size(A) - 1;
    Heapify(A, 1);
```



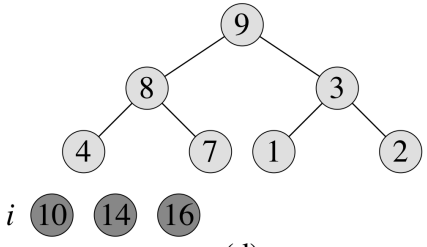
(a)



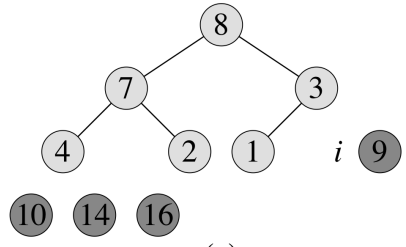
(b)



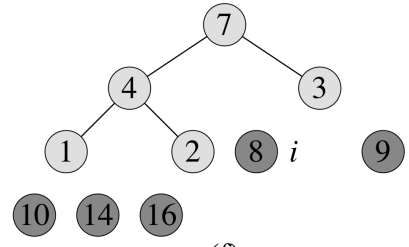
(c)



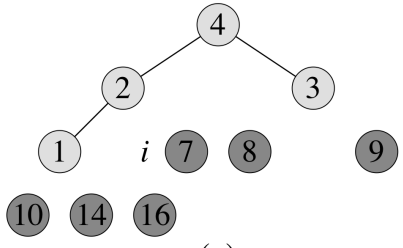
(d)



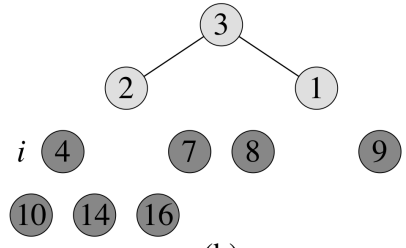
(e)



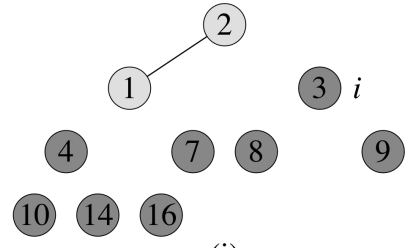
(f)



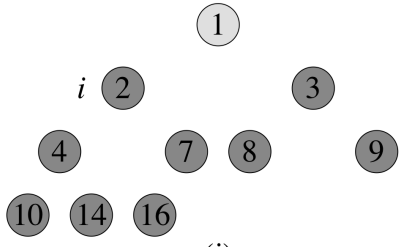
(g)



(h)



(i)



(j)

A

1	2	3	4	7	8	9	10	14	16

(k)

Analyzing Heapsort

- The call to BuildHeap() takes $O(n)$ time
- Each of the n - 1 calls to Heapify() takes $O(\log n)$ time
- Thus the total time taken by HeapSort()

$$\begin{aligned} &= O(n) + (n-1)O(\log n) \\ &= O(n) + O(n \log n) \\ &= O(n \log n) \end{aligned}$$

Note, like merge sort, the running time of heapsort is independent of the initial state of the array to be sorted. So best case and average case of heapsort are in $O(n \log n)$

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort is faster
- But the heap data structure is useful for implementing priority queues :
  - A data structure like queue or stack, but where a value or key is associated to each element, representing the priority of the corresponding element. The element with the highest priority is served first
  - Supports the operations Insert(), Maximum(), and ExtractMax()

Priority Queue Operations

- **function** Insert(S, x) inserts the element x into set S
- **function** Maximum(S) returns the element of S with the maximum key
- **function** ExtractMax(S) removes and returns the element of S with the maximum key
- Think how to implement these operations using a heap ?

Priority queue : extracting the max element

```
ExtractMax(A)
   max = A[1]
   A[1] = A[A.heap-size]
   A.heap-size = A.heap-size -1
   Heapify(A,1)
   return max
```

Since Heapify runs in $\log n$, extracting the largest element of a priority queue based on a heap takes $\log n$

Priority queue : inserting an element

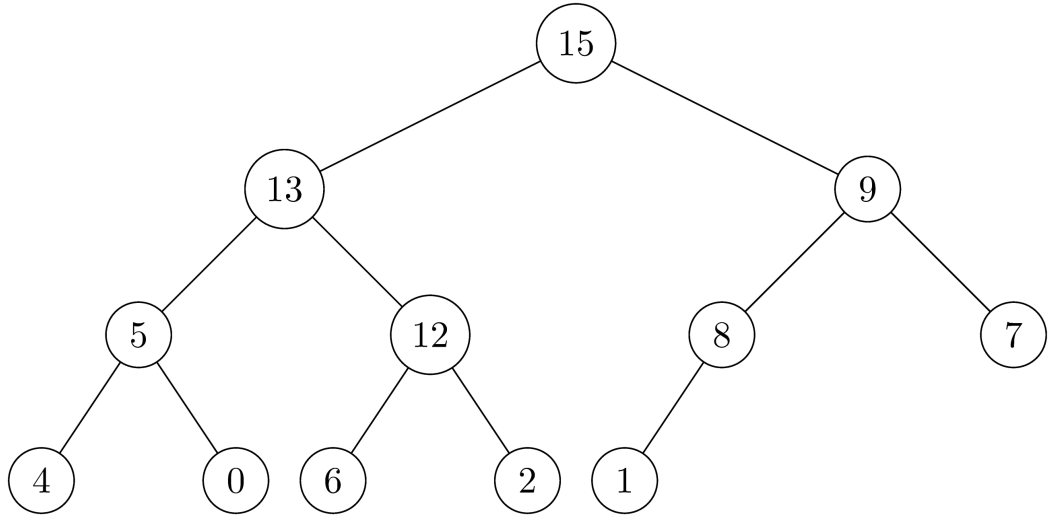
```
Insert(A, key)
   A.heap-size = A.heap-size + 1
   A[A.heap-size] = key
   i = A.heap-size
   while i > 1 and A[Parent(i)] < A[i]
      swap(A[i], A[Parent(i)])
      i = Parent(i)
```

The number of iterations execute by the while loop is bound above by $\log n$, therefore inserting an element of a priority queue based on a heap takes $\log n$

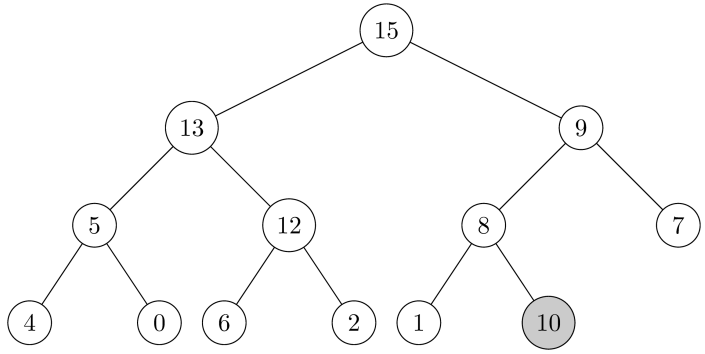
Example : Inserting an element

Insert$(A, 10)$ on the heap $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$

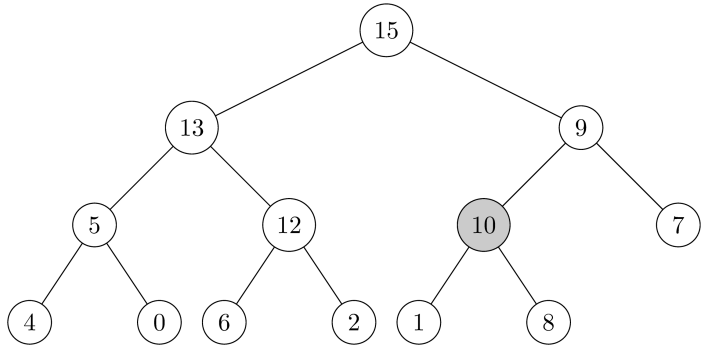
- Original heap



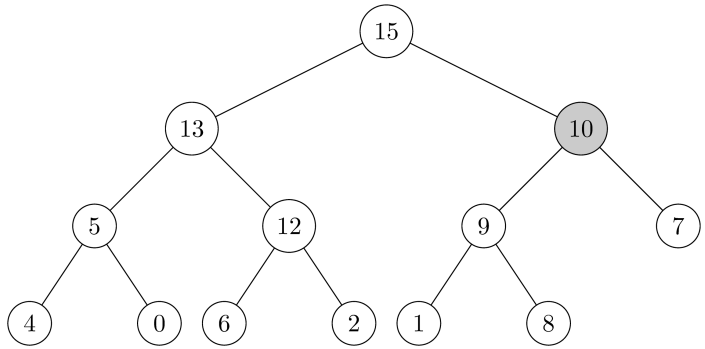
- Add the key 10 to the next position in the heap (corresponding to a new entry extending the array A).



- Since the parent key is smaller than 10, the nodes are swapped



- Since the parent key is smaller than 10, the nodes are swapped



Exercises on heapsort

1. Convert the array $A = [10, 26, 52, 76, 13, 8, 3, 33, 60, 42]$ into a maximum heap
2. Is this array [23, 17, 14, 6, 13, 10, 1, 5, 7, 12] a heap ? If not make it a heap.
3. Run the algorithm $BuildHeap(A)$ on the array $A = [12, 28, 36, 1, 37, 13, 4, 25, 3]$. Show each step of your work using the array representation of the modified heap
4. Heapsort $A[12, 28, 4, 37]$. Important, show each step of your work using the array representation
5. Heapsort $A = [25, 67, 56, 32, 12, 96, 82, 44]$ (very long)

Exercises on heapsort continue

6. What are the minimum and maximum numbers of elements in a heap of height h?
7. Where in a heap might the smallest element reside?
8. Is an array that is in reverse sorted order a heap?
9. Using the example Insert(A,10) in your class notes, show the steps in the execution of Insert(A,3) on the priority queue $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$ implemented using a heap
10. Similarly to the previous question, show the steps in the execution of ExtractMax(A) on the priority queue $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$ implemented using a heap

Exercises on heapsort continue

11. A d-ary heap is like a binary heap, but instead of 2 children, nodes have d children.
    - 11.1 Explain how would you represent a 3-ary heap in an array, i.e. give the formulas for $Parent(i)$, $Left(i)$ and $Right(i)$
    - 11.2 What is the height of a 3-ary heap of n elements ?
    - 11.3 Sketch the idea of a heapify routine for a 3-ary heap
    - 11.4 Give an implementation of ExtractMax() for a priority queue based on a 3-ary heap
12. Show how to implement a regular FIFO queue using a "min"-priority queue
13. Show how to implement a stack using a "max"-priority queue

Exercises on heapsort continue

14. Insertion sort and merge sort are stable algorithms while heapsort and quicksort are not. Can you explain why this is so ?
15. Run $\mathit{Heapify}(A, 3)$ on the array $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$
16. $\mathit{Heapify}(A, i)$ in the class notes is a recursive algorithm. Write an equivalent iterative algorithm.
17. Run $\mathit{Heapsort}(A)$ on the the array $A = [3, 15, 2, 29, 6, 14, 25, 7, 5]$
18. What is the running time of *Heapsort* on an array A of length n that is sorted in decreasing order ?