Fundamental Algorithms
Lecture #2

Cluj-Napoca, 2020

Agenda

- Review – conclusions
- Divide et impera evaluation
- Particular cases
- Master Theorem
- Sorting
  - Heap Sort

Review – conclusions

- **Correctness**
  - How do we know an algorithm is correct?
  - **Testing** never shows an algorithm is correct. It can only show it is INCORRECT (by finding bugs)
  - **Absence of evidence ≠ Evidence of absence**
  - Dijkstra: "Testing shows the presence, not the absence of bugs. "
  - So, how can we know an algorithm is correct?
  - **Proof**!
  - if the ***pre-conditions*** are satisfied, the ***post-conditions*** will be true when the algorithm *terminates*;
    - ***partial*** correctness = whenever preconditions are satisfied, the post-conditions are true;
    - ***total*** correctness = partial correctness + termination condition

Review – conclusions

- **Complexity**
  - Evaluate **time** and **space** requirements
  - **Time** as an estimation of the ***amount of work*** done
    - As an expression of ***# of atomic*** operations
    - Identify the operations done, count their ***number*** and estimate their growths
    - Depends on the ***size of the input data*** (***n***)
    - Depends on ***case*** (best, worst, average to be evaluated)
  - **Space** requirements as an expression of ***supplementary*** memory
    - Need algorithms using ***constant extra space***
    - Some times, algs with ***lgn*** extra space are accepted

Review – conclusions

- **Complexity**
  - Time = amount of work = as a function of n (size of input data)
  - We need its asymptotic growth
  - Lower bound **Ω** depends on the **problem**
  - Upper bound **O** depends on the **algorithm**
  - **Efficiency** compare algorithms (their corresponding **O** function) among each other – one is more/less efficient
  - **Optimality Ω = O** in the worst case scenario - compare an algorithm with the problem lower bound

Review – conclusions

- **Stability**
  - The property of an algorithm to preserve the **relative order of equal elements** from the input (initial/original data) in the output (final data/result)
  - Desired property
    - Choose stable algorithms, if possible
      - When and why?

Review – conclusions

- Correctness
- Efficiency
- Stability

Divide et impera evaluation

```
divide_et_impera(n, I, O)
    if n<=n0
        then    direct_solution(n, I, O)
        else    divide(n, I1,I2,…,Ia)
                divide_et_impera(n/b,I1,O1)     //a rec. calls
                divide_et_impera(n/b,I2,O2)     //b division factor
                …
                divide_et_impera(n/b,Ia,Oa)
                combine(O1,O2,…,Oa,O)
```

Divide et impera evaluation – contd.

- $f(n) = n^c$
- $$t(n) = \begin{cases} t_0 & \text{if } n < n0 \\ at(n/b) + f(n) & \text{if } n >= n0 \end{cases}$$

a = number of recursive calls

b = the ratio to which the original domain is divided

c = degree of the polynomial expressing the execution time of the *divide et impera* sequence except for the recursive calls

It is reasonable to assume f(n) is polynomial as we are seeking for overall polynomial running time algorithms

Divide et impera evaluation – contd.

$t(n)=n^c[1+a/b^c+(a/b^c)^2+\ldots(a/b^c)^{\log_b n-1}]$

Cases: 1. $q<1;\ a<b^c \Rightarrow$ $O(n^c)$

2. $q=1;\ a=b^c \Rightarrow$ $O(n^c \cdot \log_b n)$

3. $q>1;\ a>b^c \Rightarrow$ $O(n^{\log_b a})$!!

It's polynomial

Small power

Independent of c

Obs: b should be scaler (**b>1**)

composition should comply to the **partition** rule!

In most cases, either divide, or combine is O(1)

Ex: quick sort combine = done by default (sort in situ) – no time at all

merge sort divide O(1): compute the middle index

Particular cases

1. $c=1 \quad\Rightarrow f(n)=n$

$$t(n)=\begin{cases} O(n) & \text{if } a<b \\ O(n\cdot\log_b n) & \text{if } a=b \\ O(n^{\log_b a}) & \text{if } a>b \end{cases}$$

Ex: qsort $a=b=2 \Rightarrow O(n\cdot\log_2 n)=O(n\cdot\log n)$

Is qsort optimal? Justify!

It ($a=b=2$) is NOT the worst case!

Are there means of avoiding worst case?

See the following courses/seminars

Particular cases – cont.

2. c=0 => f(n)=ct

Q? Is this possible ? Does such algs exist?

$$t(n)=\begin{cases} N/A & \text{if } a<b^0 \Leftrightarrow a<1 \text{ not possible!} \\ O(\log_b n) & \text{if } a=b^0 \Leftrightarrow a=1 \\ O(n^{\log_b a}) & \text{if } a>b^0 \Leftrightarrow a>1 \end{cases}$$

Ex: a=1, b=2 search in BST => O(logn)

a=2, b=2 tree traversal => O(n)

Master Theorem to remember/to keep close

- $f(n)= n^c$
- $$t(n)=\begin{cases} t_0 & \text{if } n<n_0 \\ at(n/b)+f(n) & \text{if } n>=n_0 \end{cases}$$

1. $q<1; a<b^c => O(n^c)$
2. $q=1; a=b^c => O(n^c * \log_b n)$
3. $q>1; a>b^c => O(n^{\log_b a})$

Homework

- Consider your personal computer/notebook. Check the number of instructions/second it can execute, then compute which is the maximum problem size (i.e. *n*) that a (1) *polynomial* and (2) *exponential* algorithm can solve in:
  - 1 day
  - 1 week
  - 1 month
  - 1 year
  - 1.000 years
  - 1.000.000 years

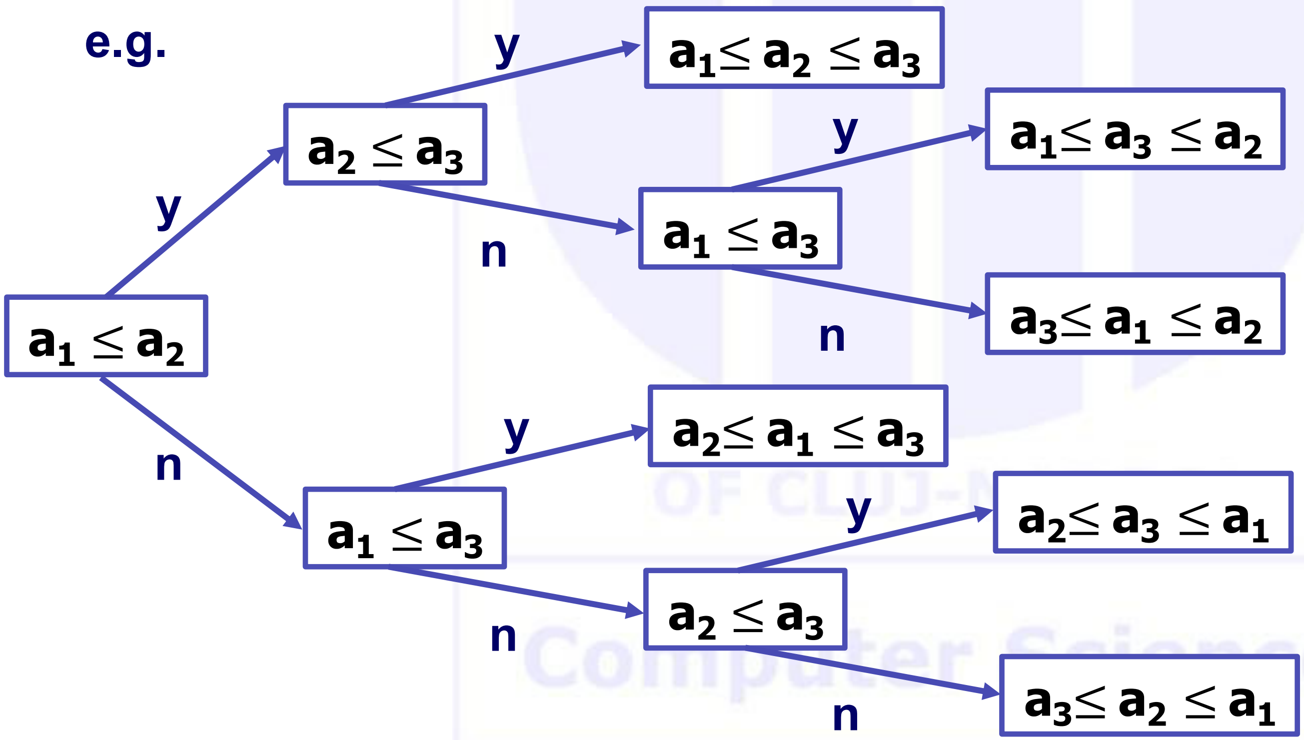
Sorting algorithms

- Sorting problem **Ω(nlgn)**
- What is all about?
- Direct strategies – seminary
- Advanced strategies – course

Sorting problem Ω

- ***Lemma***: Any comparison-based sorting alg. performs **Ω(nlgn)** comparisons in the worst case to sort **n** objects

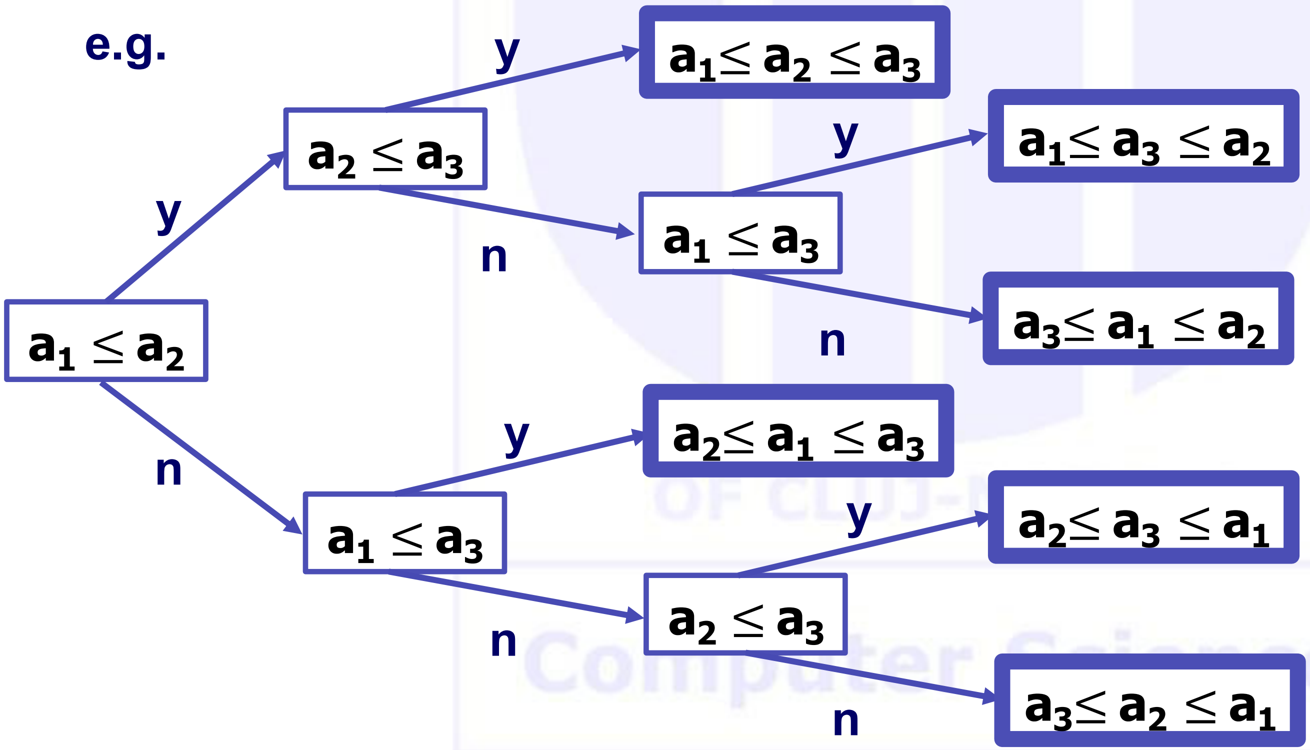
e.g.



Sorting problem Ω

- ***Lemma***: Any comparison-based sorting alg. performs **Ω(nlgn)** comparisons in the worst case to sort **n** objects

e.g.



leaves = each possible answer for any given input

How many leaves? (ℓ)

Sorting problem **Ω**

- ***Lemma***: Any comparison-based sorting alg. performs **$\Omega(n\lg n)$** comparisons in the worst case to sort **n** objects
- $\ell =$ n! leaves in the tree
- Worst-case running time $\equiv$? (related to what from the tree)

Sorting problem Ω

- ***Lemma***: Any comparison-based sorting alg. performs **Ω(nlgn)** comparisons in the worst case to sort **n** objects
- $\ell = n!$ leaves in the tree
- Worst-case running time ≡ height of the tree (h_T)
- h_T **?** $\log_2 \ell$
  - (hint) What is the maximum no. of leaves (max ℓ) for a tree of height h_T?

Sorting problem Ω

- ***Lemma***: Any comparison-based sorting alg. performs **Ω(nlgn)** comparisons in the worst case to sort **n** objects
- $\ell =$ n! leaves in the tree
- Worst-case running time $\equiv$ height of the tree (h_T)
- $h_T > \log_2 \ell$ (motivate!)

Sorting problem Ω

- ***Lemma***: Any comparison-based sorting alg. performs **Ω(nlgn)** comparisons in the worst case to sort **n** objects
- $\ell =$ n! leaves in the tree
- Worst-case running time ≡ height of the tree (h_T)
- $h_T > \log_2 \ell$ (motivate!)

$$h_T > \log_2(n!) = \log_2(1 * 2 * 3 * \ldots * n)$$

$$= \log_2 1 + \log_2 2 + \ldots + \log_2 n$$

$$\geq \log_2 \frac{n}{2} + \ldots + \log_2 n \quad \text{//take only second half of sum}$$

$$\geq \frac{n}{2} \log_2 \frac{n}{2} \quad \text{//replace all terms with first}$$

$$= \Omega(n\lg n) \quad \text{//ignore constants}$$

If not convinced, checkout: http://www.bowdoin.edu/~ltoma/teaching/cs231/fall11/Lectures/6-moresorting/sortLB.pdf

Heap sort

- Sorting with the aid of a heap structure
- Heap = **array** viewed (logical perspective) as a BT
- Representation (logical persp.) based on the index

i = parent

$2 \cdot i$ $\quad$ $2 \cdot i+1$ = children

- Property: A[parent(i)]**>=**A[i]

Other properties may be defined

- Parent/child property => implies a **partial order** relation
- Q? What is a partial order relation?
- There is NO property between siblings
- Example - blackboard

Heap sort – cont.

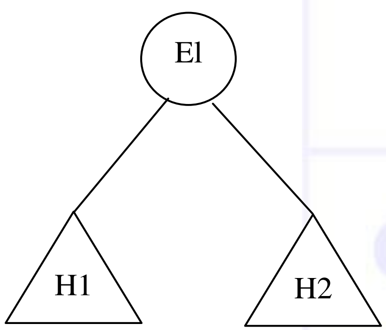
- Q1: identify a **maximal subset** on which the partial order relation becomes a **total order relation**.
  - A branch.
- Q2: based on the heap property, what consequence (**post condition**) follows?
  - The root contains the max value;
  - **Max** value in case the property based on which the heap is built is **>=**.
  - The root would have some other particularity in case another property is the choice.

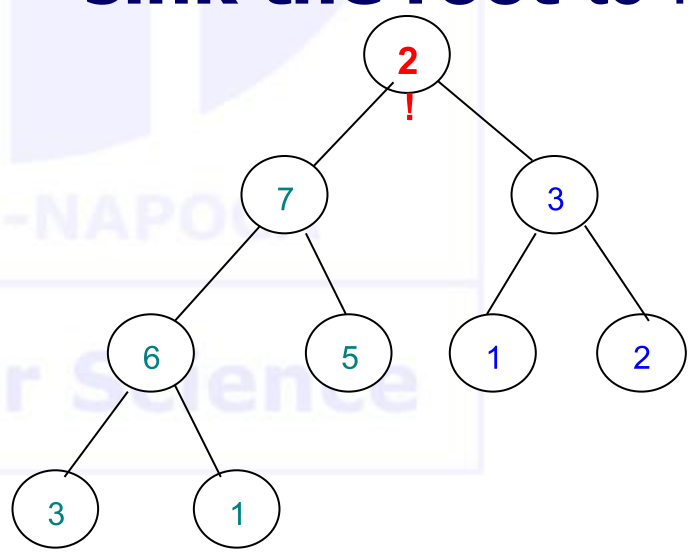
Heap sort – Heap procedures

- Heapify – Reconstituie heap
  - “Adds ” the root to 2 left and right children rooted heaps
- Build-Heap
  - Constructs the whole heap structure (on the entire array), by repeatedly applying heapify
- Heapsort
  - Reorganizes the array by repeatedly extracting the root of the heap and placing it in the “right” position of the sorted array

Heapify (Reconstituie heap)

- **Pre-condition** – 2 heaps (H1, H2)
- Goal: add a single element El s.t. the triple (El and H1, H2) represents a larger heap: H
- **Post-condition** – 1 single heap H (Root+H1+H2)
- The strategy: **top-down** = **sink the root** to its correct place in the heap



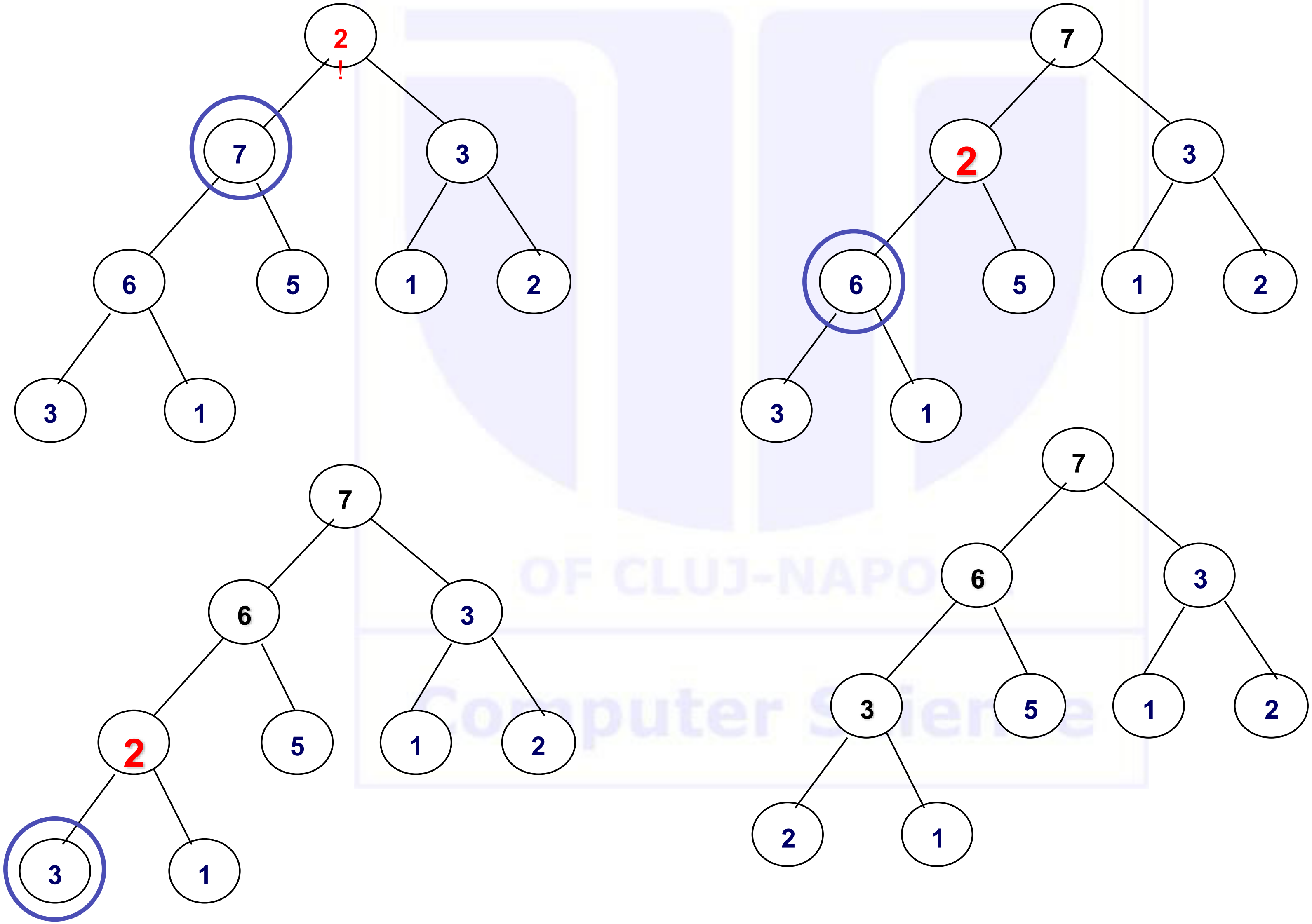


Heapify

```
heapify (A,i) //i-index of the root (= El to be added on top of the heap)
largest<-                          //root, left or right child index
  index_of_max_bet(A[i],A[left(i)],A[right(i)])
if largest <> i                    //one of the children larger than root
  then    A[i]<->A[largest]        //swap root with largest child
          heapify(A, largest)      //continue the process on the heap
              //branch of the largest child. The other branch (i.e. heap)
              //is not  affected at all
```

- it applies a **top-down** strategy

Heapify - example



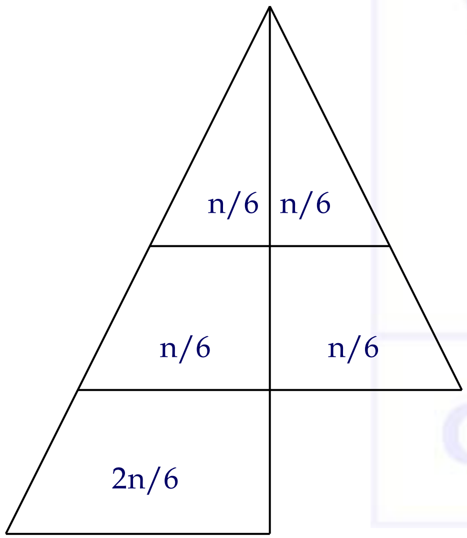
Heapify – running time

- Running time:
  - O(1) running time at one level
  - Recursive calls (how many times?)
    - Best: none => O(1)
    - **Worst**: every time => repeated down to the level of the leaves
      - ***Intuitive***: height of a full BT=lgn; you have to "sink" the root down to the level of a leaf (O(h), h=lgn for a complete tree)
      - ***Exact*** evaluation:
      - The last row of the tree is exactly half full, and we go on that branch
      - If full BT, half of the nodes are leaves
      - *Why b=3/2? Explained later (next 2 slides)*
      - $t(n)=t(2n/3)+O(1)$: a=1, b=3/2, c=0
      - => Apply Master (case #2) and get $O(\log_b n) = O(\log_{3/2} n) =$

$= O(\lg n/\lg(3/2)) = O(\lg n/(\lg 3-1))=O(c\cdot \lg n)=$ **$O(\lg n)$**

Heapify – running time

- Why $t(n)=t(2n/3)+O(1)$?
- Why 2n/3 nodes on the rec call (b=3/2)?
- Picture 3*n/6 (internal) and 3n/6 (leaves)



All other levels – multiple levels

Leaves' parents (on the left) and leaves (on the right) – 1 level

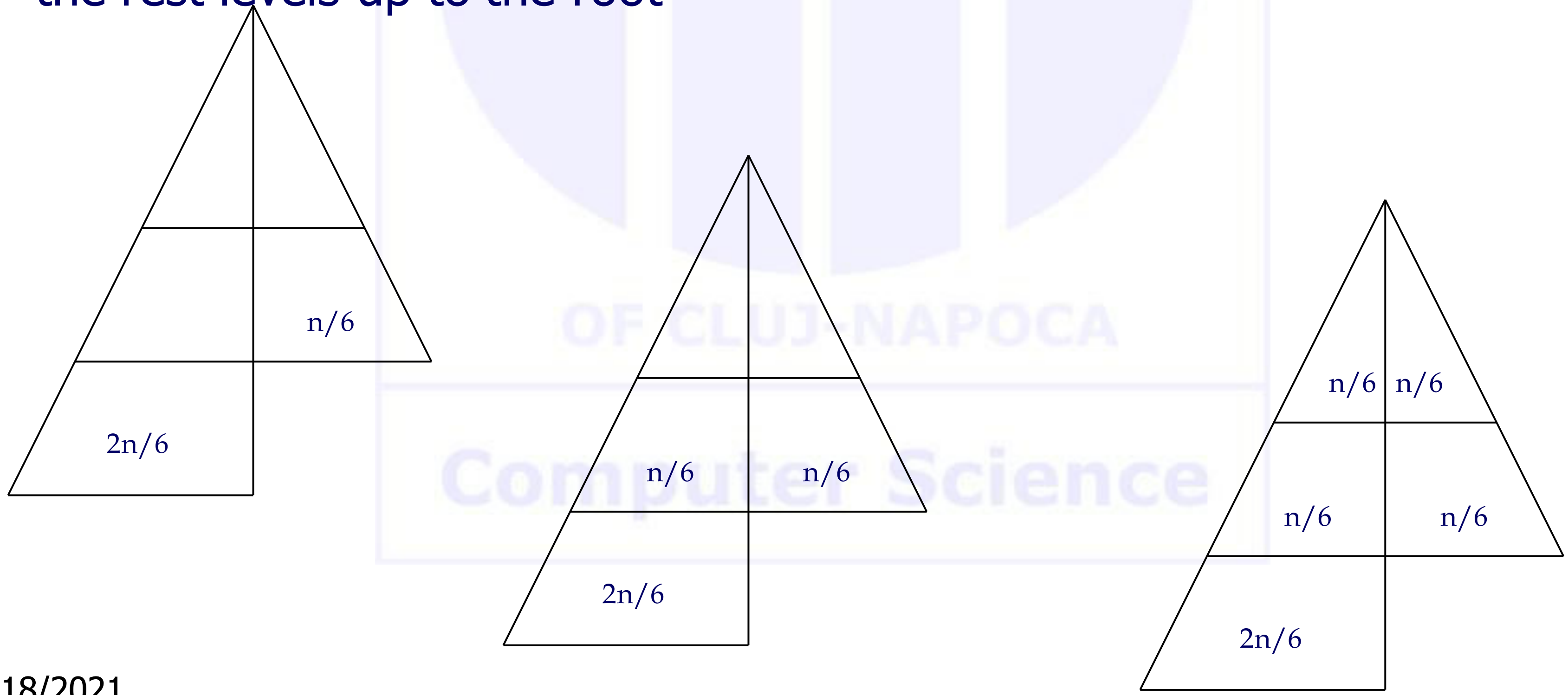
Leaves (on the left half only) – 1 level

Heapify - Justification of # of nodes

Half of the nb. of nodes are leaves (2/3 on the left, 1/3 on the right)

Nb. of parents of leaves from left= half the number of those leaves (and at the same time = nb. of leaves from right)

The rest of the elements (= n-2n/6-n/6-n/6) are equal split (left/right) on the rest levels up to the root



Heapify - Justification of # of nodes

Nb. Of nodes on the worst case: nodes on the largest branch (left one)

$=n/6+n/6+2n/6=4n/6=2n/3$

So $t(n)=t(2n/3)+O(1)$ (claimed 3 slides before) is justified

Build-Heap

- Heapify starts from the assumption we already have 2 heaps. Where are they from?
- 1 single node **is** a (very basic) **heap**.
- So, half of the # of nodes are already heaps; we get the strategy
  - Start with 2 heaps each of dimension 1
  - Add their common parent node to build a heap of dimension 3
- Adopt a **bottom-up** strategy:
  - ½ out of all nodes are heaps from the very beginning (leaves in a complete binary tree)
  - Apply heapify to the first non-leaf node (the node in the tree with the largest index, having at least one child)
  - Go to the "next" indexed node (sibling to the left of the first processed element)
  - Continue the process until reach the root

Build-Heap – code

```
Build-Heap(A)
for i<-|A|/2 downto 1        //from the non-leave nodes to the root
   do heapify(A,i)           // build the heap out of 2 already built
                             // heaps and 1 node
```

It applies a **bottom-up** strategy

Running time:

- it **seems** to be n/2 ·lg n
  - We apply n/2 times (on all non-leaf nodes) heapify
  - heapify in worst case is O(lgn)
  - Means n/2 times O(lgn) goes to n/2 ·lg n
  - CL: only building the heap takes n/2 ·lg n
  - So we cannot sort on n/2 ·lg n!!!

Build-Heap – running time

- Running time – a first evaluation:
  - n/2 times heapify => nlgn. Not good ☹
- Running time – a closer look:
  - For all leaves, heapify does **not** apply
    - Half of the nodes are leaves – no operation applied
  - For all the parents of all the leaves it only takes O(1)
    - nb. of leaves' parents = half of the nb. of leaves
    - time require to heapify all of them (=nb*time): $\frac{1}{2}^2 \cdot n \cdot 1$
  - For half of the remaining elements, it takes 2 steps to "heapify" them:
    - half of the rest is $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot n = 1/8 \cdot n$
    - time require to heapify them: $1/8 \cdot n \cdot 2$
  - At each of the next steps, the nb. of elements halves, while the nb. of steps required to heapify each increases by 1

Build-Heap – running time

$t(n) =$

// #· individual time

$n/2 \cdot 0+$ // (leaves)

$n/2^2 \cdot 1+$ // (leaves'parents) ….

$n/2^3 \cdot 2+$

$n/2^4 \cdot 3+\ldots$

$$= \sum_{0}^{[\lg n]} [n/2^{h+1}] \cdot O(h)$$

Build-Heap – running time

To evaluate the sum on the prev slide, start from:

$\sum x^k$	$= (1-x^{n+1})/(1-x)$	(geom prog., first =1, q=x)
$\sum x^k$	$=1/(1-x)$	For $x<1$, $n\to\infty$ we get:
$(\sum x^k)'$	$= [1/(1-x)]'$	(derive)
$\sum k\cdot x^{k-1}$	$= 1/(1-x)^2$	(multiply by x)

$$\sum k\cdot x^k = x/(1-x)^2 \qquad \textbf{(1)}$$

Use the result (for a particular value of x) to calculate the desired sum from before

Build-Heap – running time

$$t(n) = \sum_{0}^{[\lg n]} [n/2^{h+1}] \cdot O(h)$$

$$= \sum_{0}^{[\lg n]} n[1/2^{h+1}] \cdot h$$

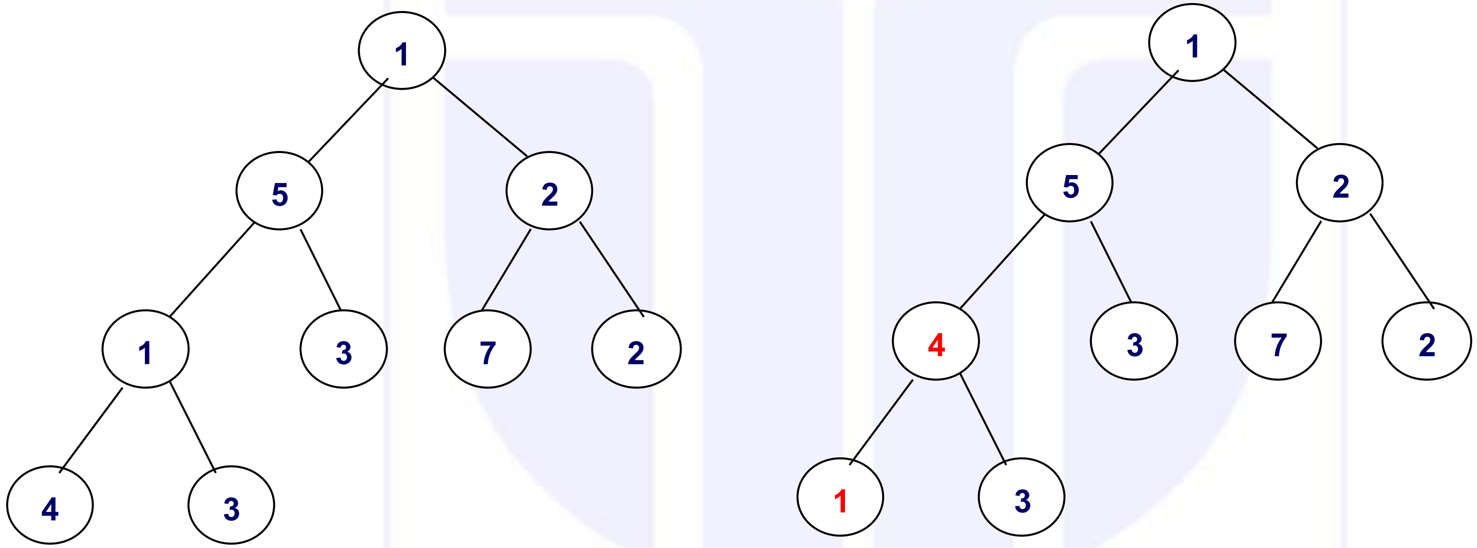
$$= n/2 \cdot \sum_{0}^{[\lg n]} [1/2^{h}] \cdot h = n/2 \cdot \sum h \cdot (1/2)^{h}$$

But since $\sum k \cdot x^k = x/(1-x)^2$ (from (1) previous slide), for **$x=1/2$** we get

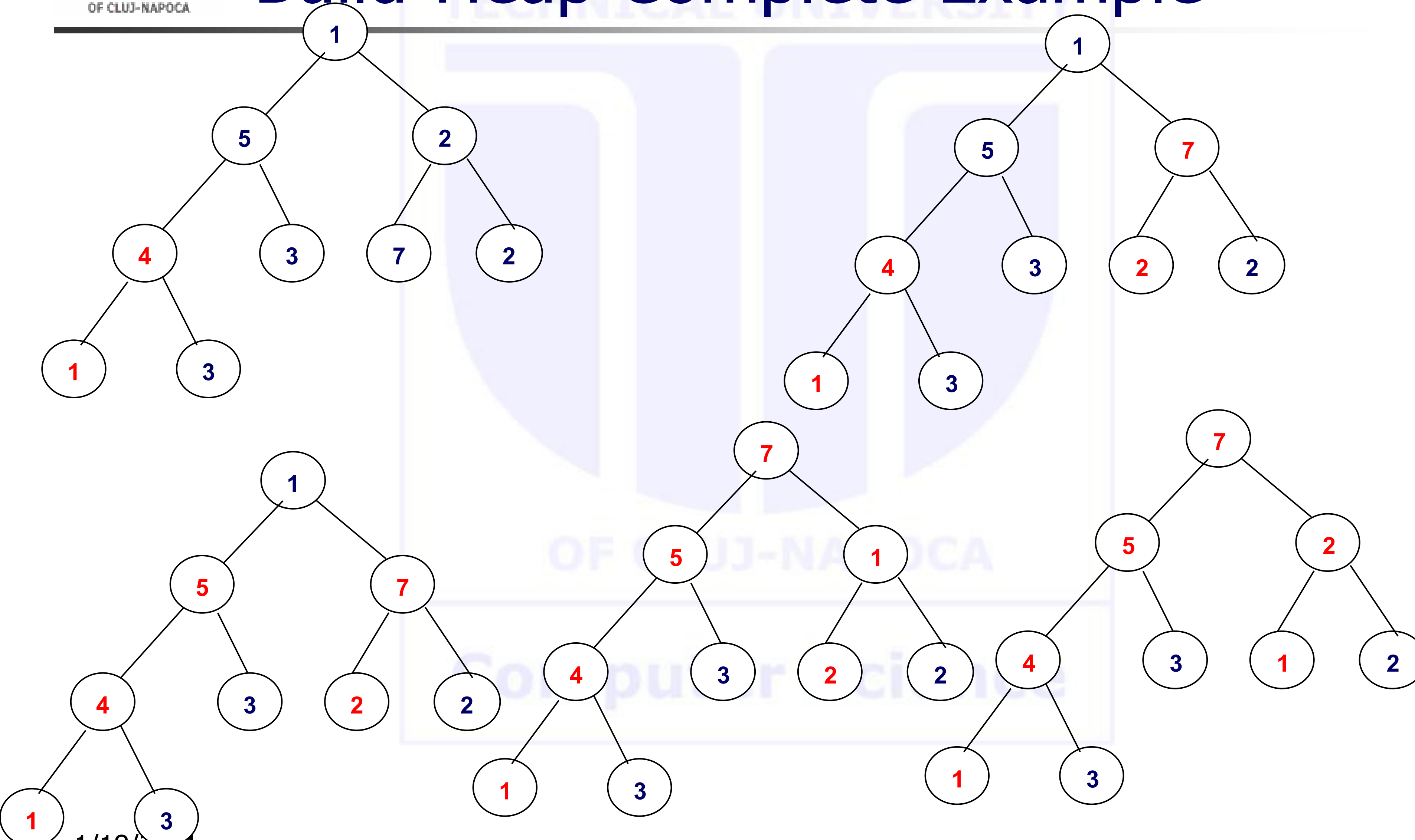
$\sum k \cdot x^k = (1/2)/(1-1/2)^2 = (1/2)/(1/2)^2 = 2$

So $\sum h \cdot (1/2)^h = 2$, therefore **$t(n) = O(n)$**

Build-Heap Complete Example



Build-Heap Complete Example



Heapsort

- Heapsort – the complete technique
  - Build Heap which selects the max on the top of the heap
  - swap the top element (root) with the bottom one (last leaf) (i.e. move the max element in the last position of the array, where it belongs in the ordered array)
  - At this point, we destroyed both the heap structure, and we don't have an ordered one!

Heapsort cont.

- Heapsort – the technique –cont.
  - except for the **first** and **last** elements, we have a heap
  - from the second A[2] to the one before the last A[|A|-1] we have a heap
  - BUT the last element is in its right place in the ordered array already; consider it not more in the heap (thus, heap_size should decrement by 1)
  - apply heapify again on the new, smaller heap (without the last), for A[1] to sink that element in the right position
  - repeat the process until the dim of the heap becomes 1
  - while the heap's dimension decreases (by 1 each step, from the right), the already ordered array's dimension increases (with 1 each step, on the left)

Heapsort - code

```
HeapSort(A)
Build-Heap(A)   //generate the initial heap structure
heap_size[A]<-|A|
for i<-|A|downto 2//from the non-leave nodes
  do A[1]<->A[i]  //swap the root of the heap
                  //with the bottom element in the  current heap;
                  //array A[1..i-1] is a heap, array A[1..|A|] is
                  //an ordered structure
  heap_size[A]<-heap_size[A]-1
  heapify(A,1) // rebuild the heap struct. rom 1 to i
```

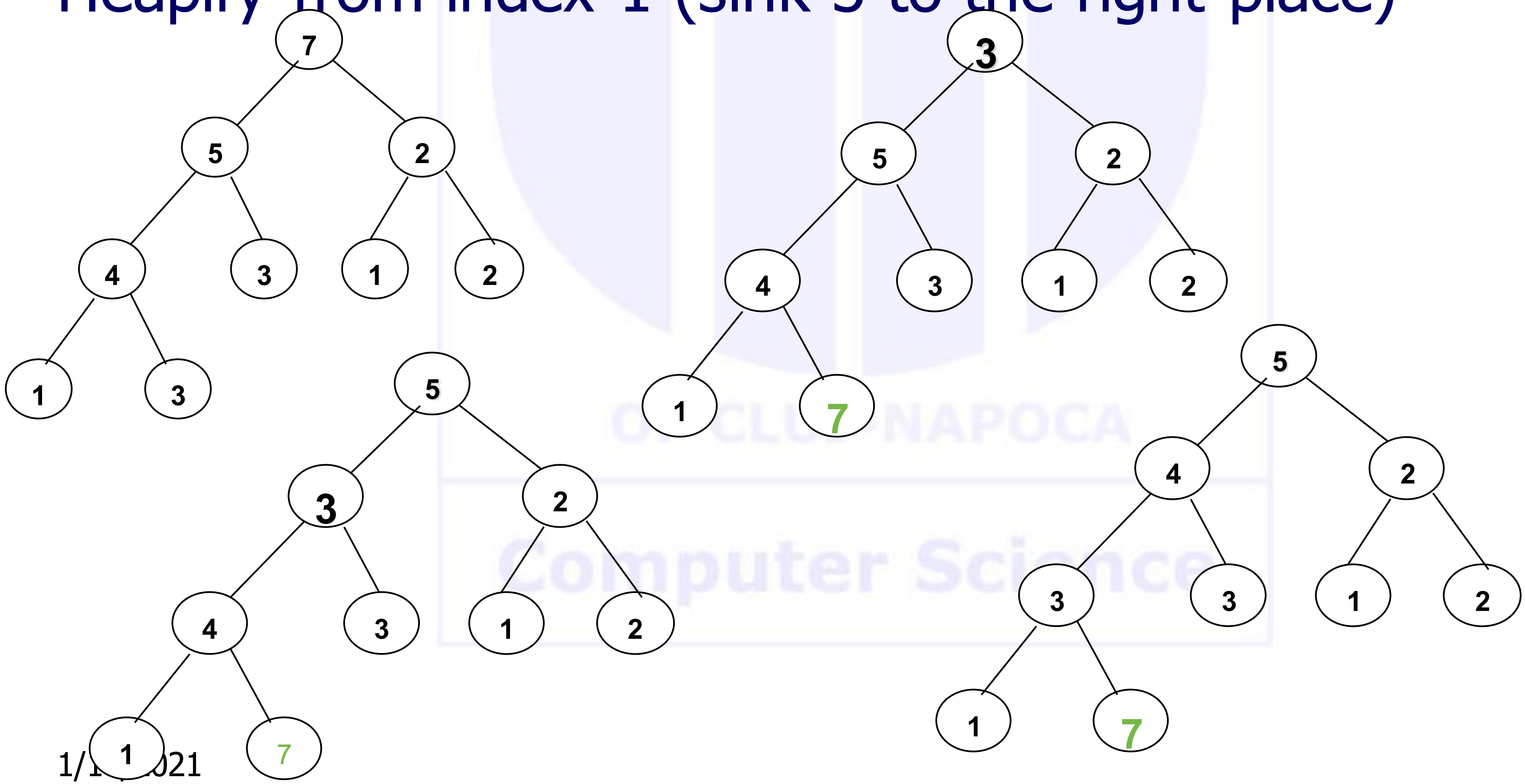
Heapsort - evaluation

- **`Build-Heap(A)`** takes O(n)
- <u>**`for`**</u> **`i<-|A|downto 2`** repeats n times
- **`heapify(A,1)`** takes O(h) where h goes down from lgn to 1, so loop<=n·lgn
- O(n)+O(n·lgn) = O(n·lgn)
- $t_{HeapSort}$ = **O(n·lgn) = Ω(n·lgn)**
- Eval in worst case => **optimal algorithm**

Heapsort – complete example (after the heap was built – the for loop)

Swap 7 (top) with 3 (bottom)

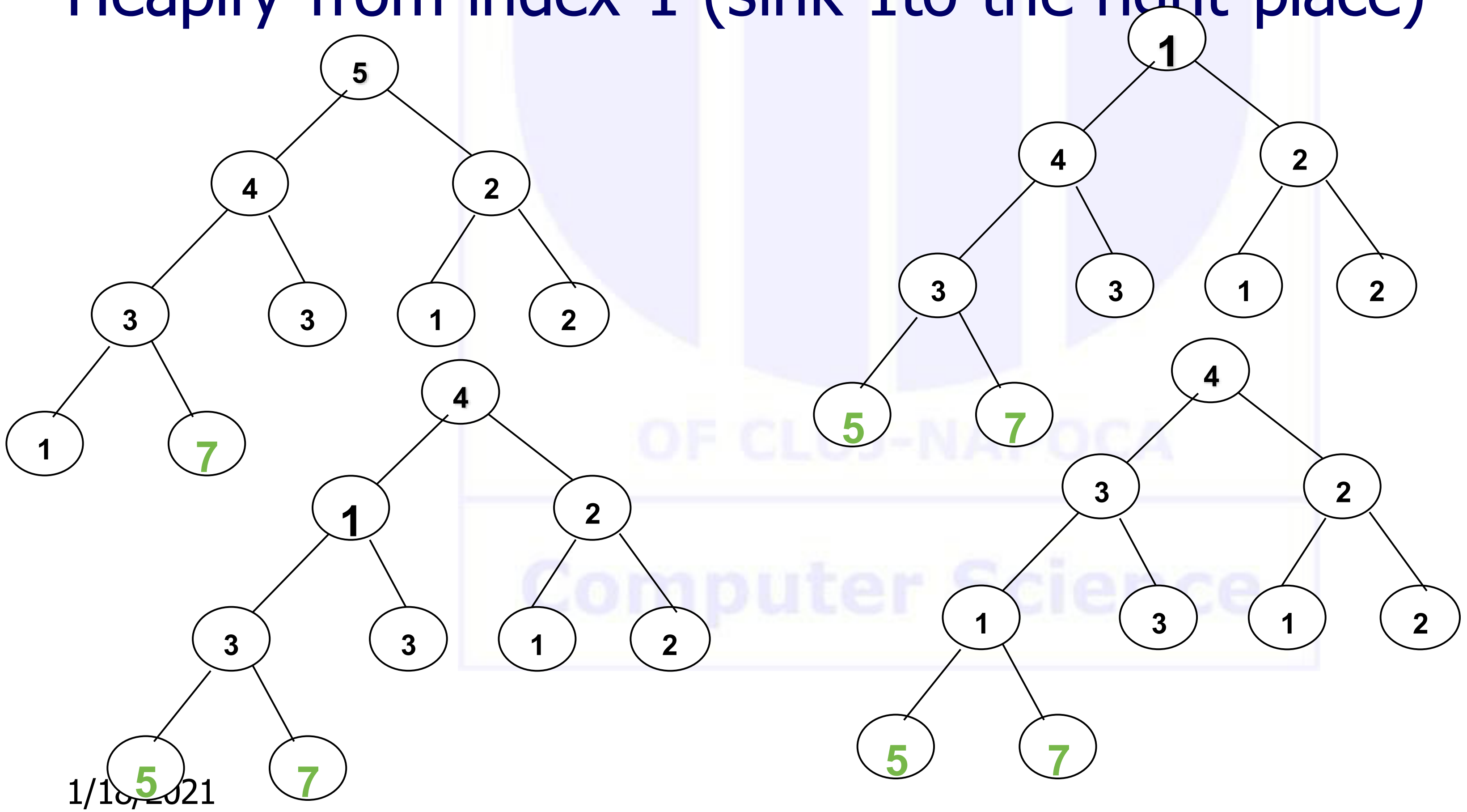
Heapify from index 1 (sink 3 to the right place)



Heapsort – complete example (green=sorted part; blue =heap part)

Swap 5 (top) with 1 (bottom)

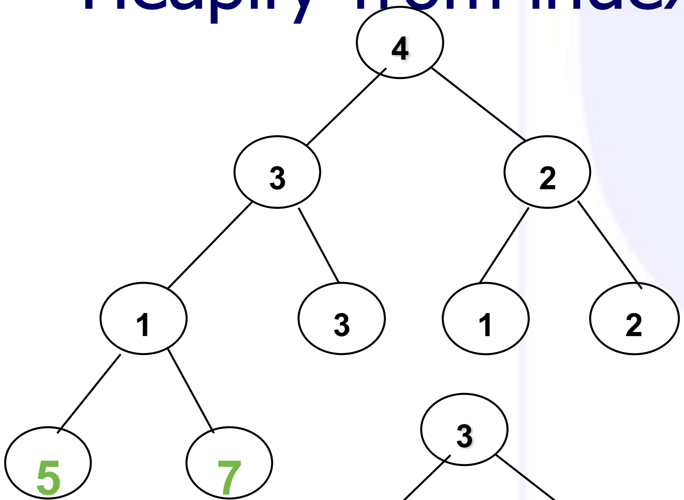
Heapify from index 1 (sink 1to the right place)

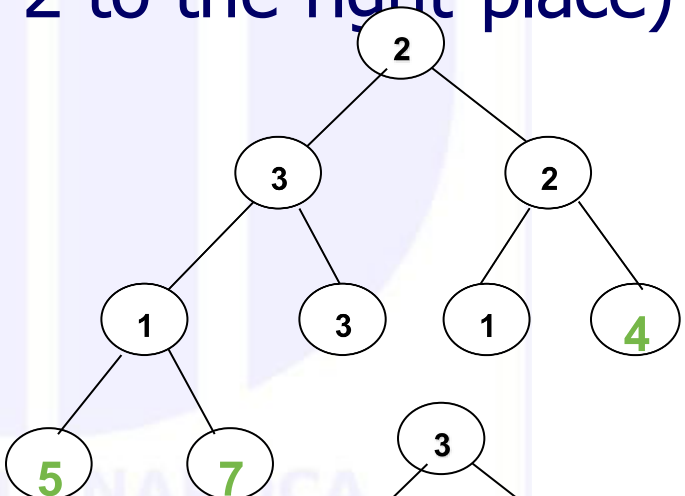


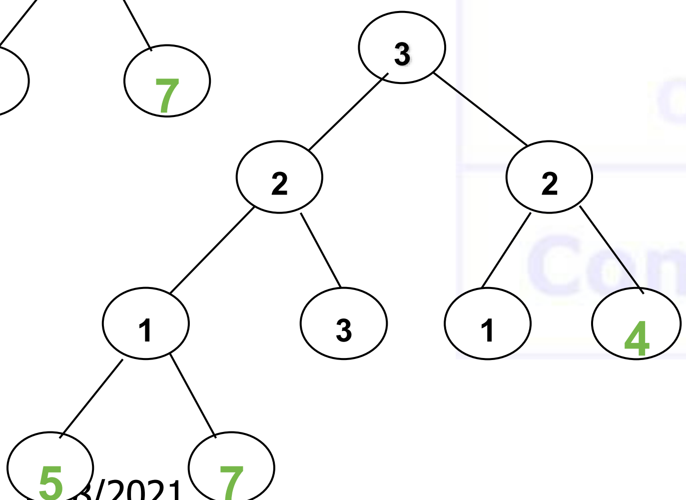
Heapsort – complete example (green=sorted part; blue =heap part)

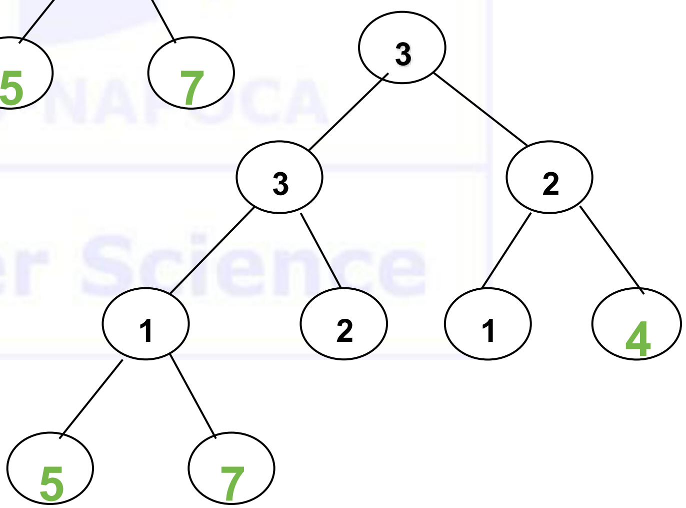
Swap 4 (top) with 2 (bottom)

Heapify from index 1 (sink 2 to the right place)





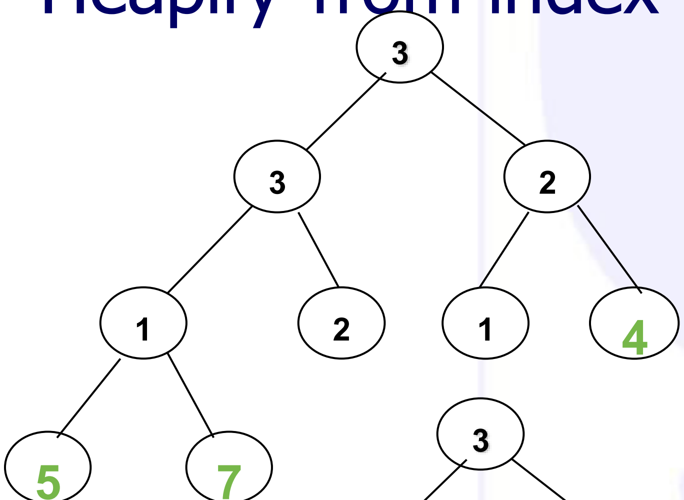


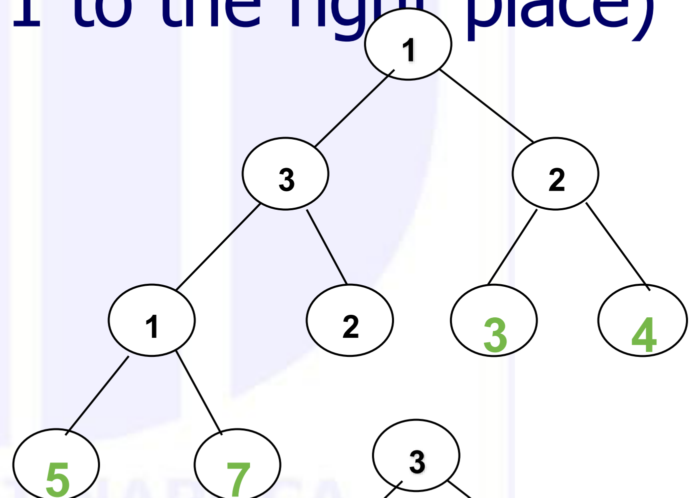


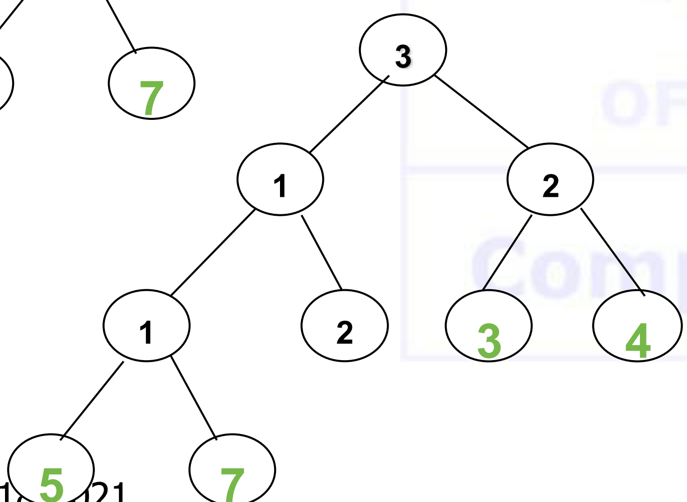
Heapsort – complete example (green=sorted part; blue =heap part)

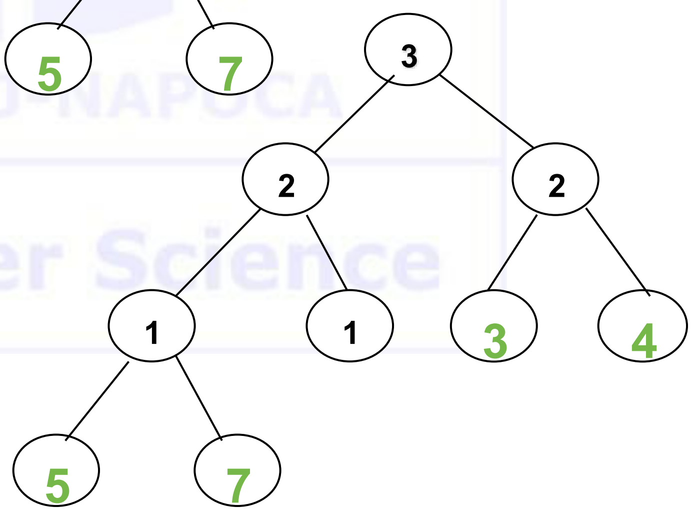
Swap 3 (top) with 1 (bottom)

Heapify from index 1 (sink 1 to the right place)





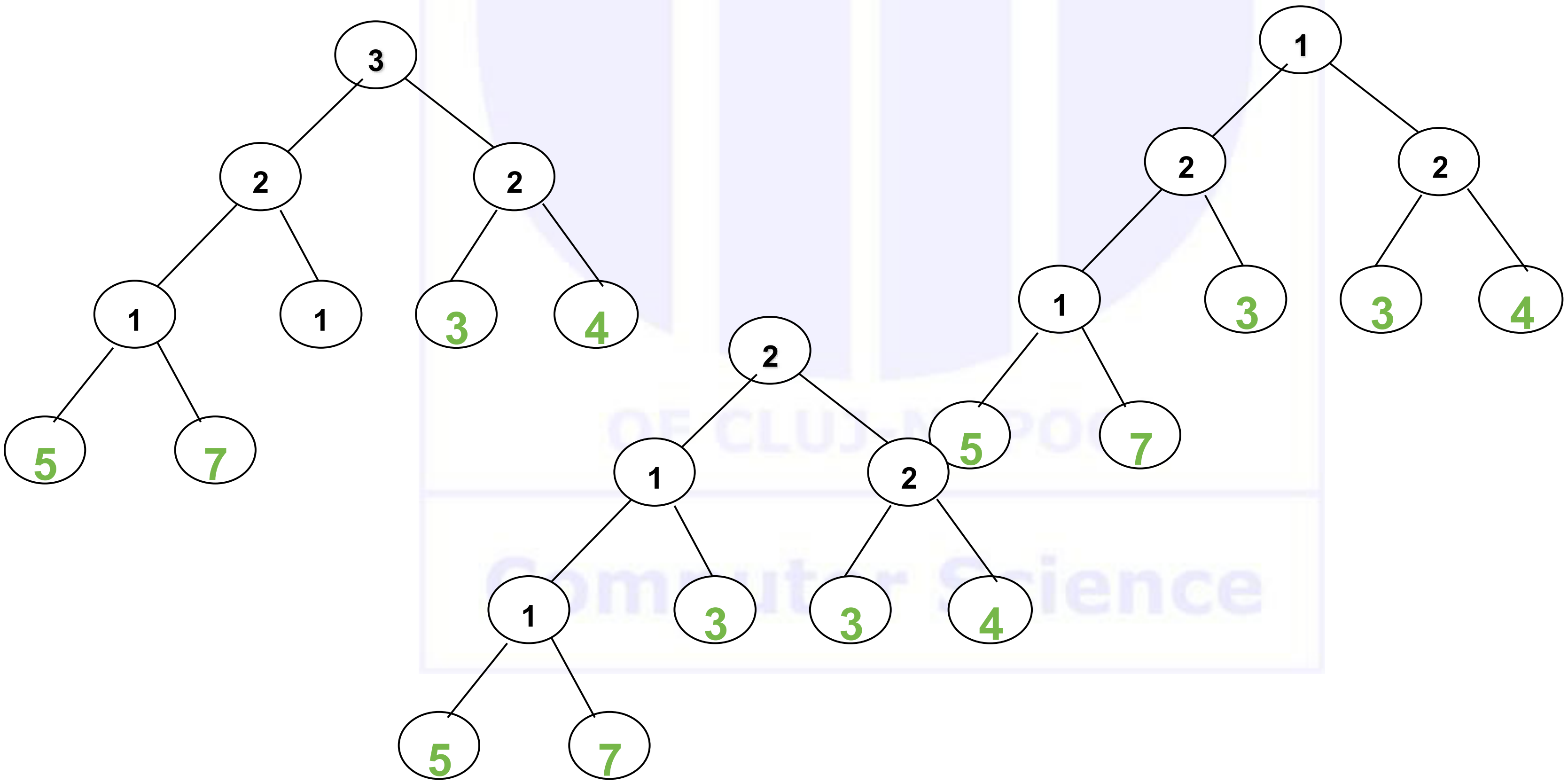




Heapsort – complete example (green=sorted part; blue =heap part)

Swap 3 (top) with 1 (bottom)

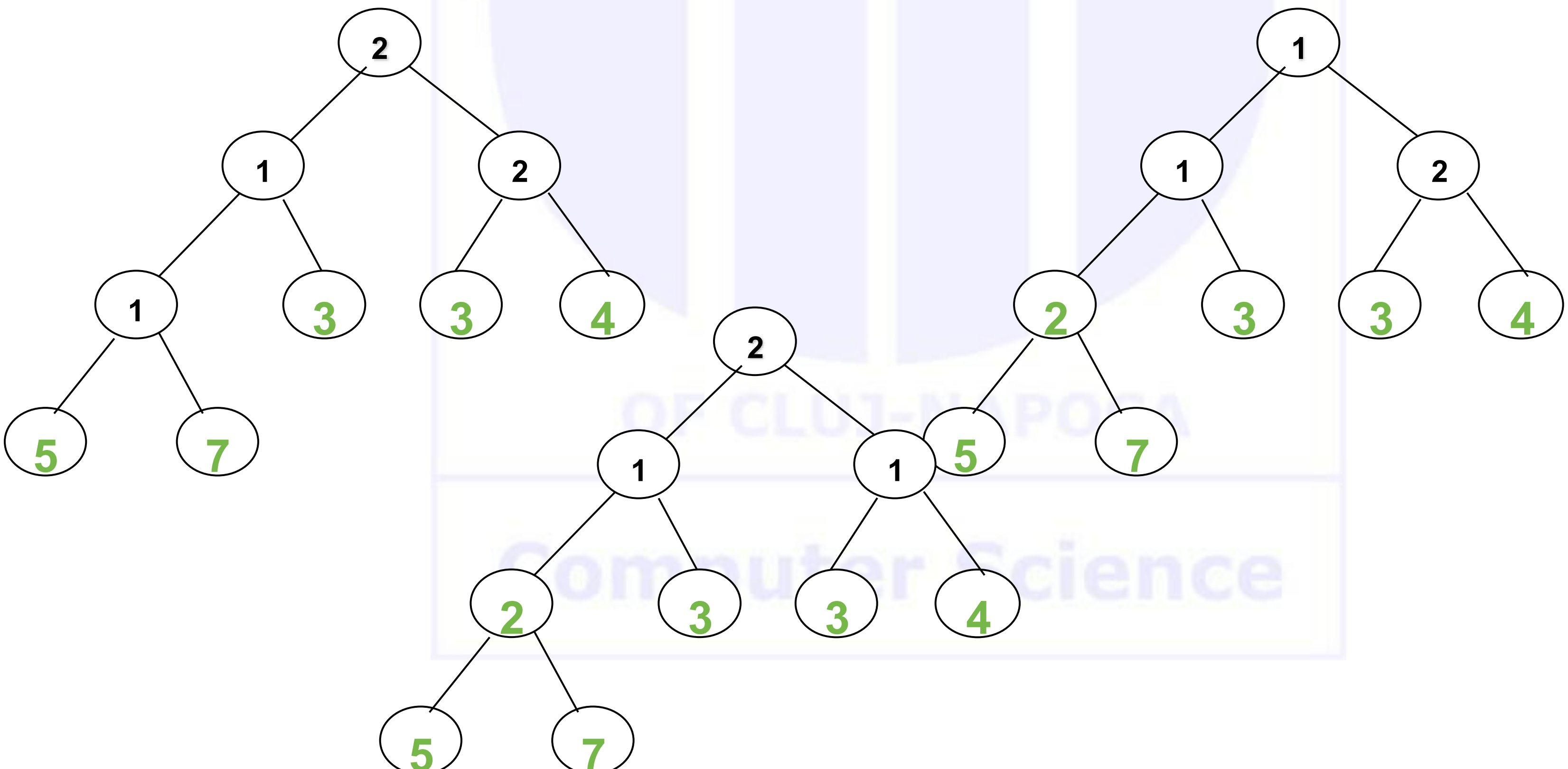
Heapify from index 1 (sink 1 to the right place)



Heapsort – complete example (green=sorted part; blue =heap part)

Swap 2 (top) with 1 (bottom)

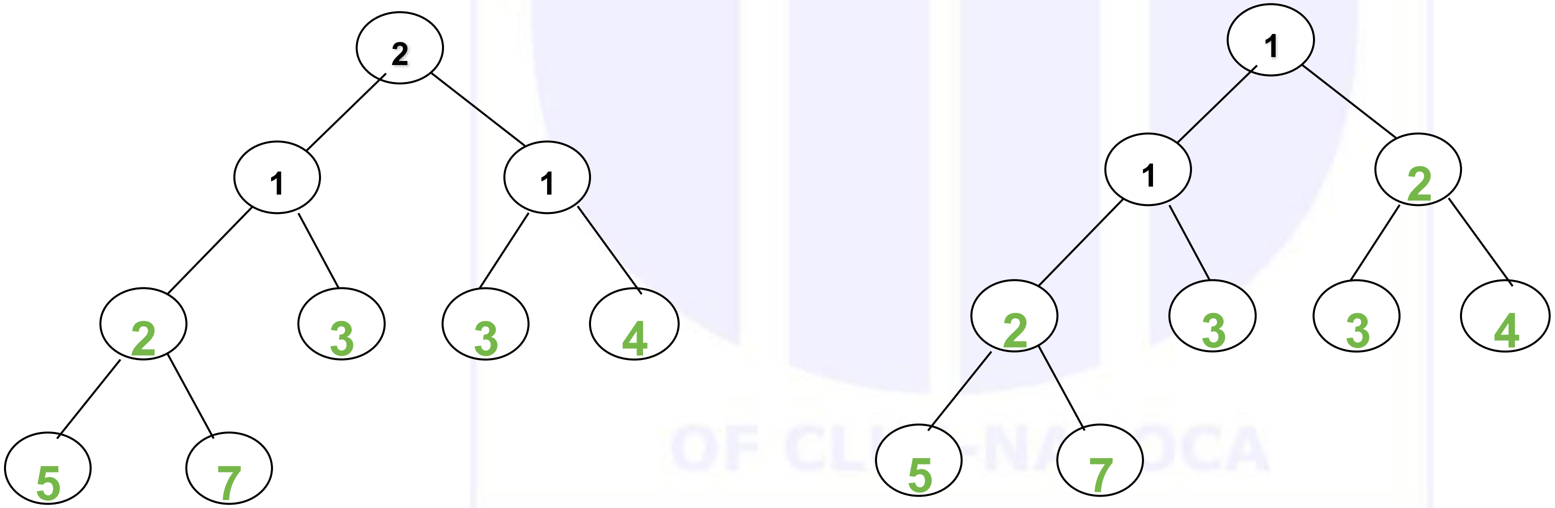
Heapify from index 1 (sink 1 to the right place)



Heapsort – complete example (green=sorted part; blue =heap part)

Swap 2 (top) with 1 (bottom)

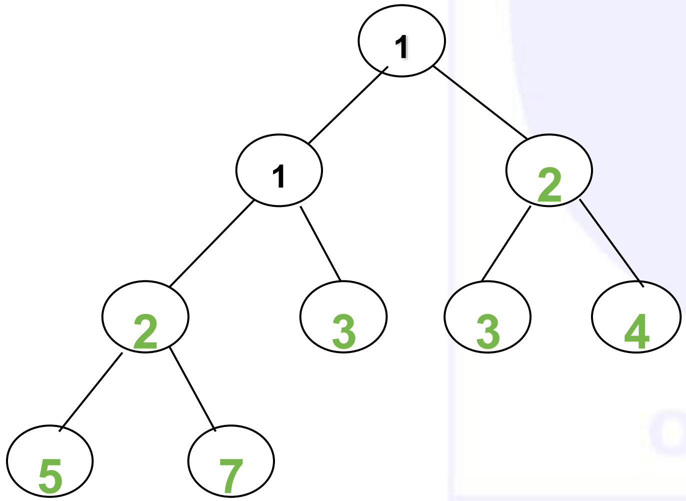
Heapify from index 1 (sink 1 to the right place)

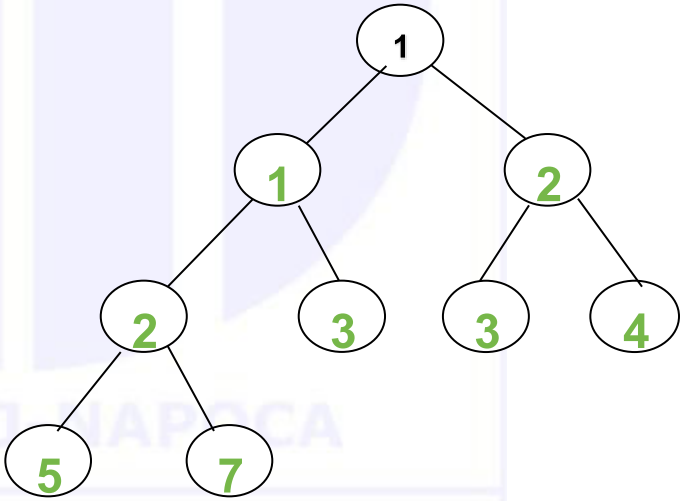


Heapsort – complete example (green=sorted part; blue =heap part)

Swap 1 (top) with 1 (bottom)

Only 1 element in the heap =>smallest=>all is array





Blue = elements in the heap

Green = elements in the ordered array

http://www.eecs.wsu.edu/~cook/aa/lectures/applets/sort1/heapsort.html

Heap – as a data structure

- Build heap strategy applies in case the **dimension** of the array is **known in advance** and has a **constant** value
- If not, define and use a heap as a data-structure => add dimension associated with the structure (size of the heap)
- Operations:
  - pop_heap extract the top from the heap
  - push_heap add one item to the heap

Heap – as a data structure – cont.

- pop_heap Extracts the top element
  - Move bottom element on top (swaps last with top, similar to 1 step of heapsort)
  - Decrements the heap size
  - Heapify the whole (from 1 to the new size), to update the heap structure =>O(lgn)
- push_heap
  - Adds a new element at the bottom
  - Rebuild heap, a bottom-up approach (bubble the bottom element upper in the heap, until it finds a larger-value parent) => O(h)=O(lgn)
- Examples on the blackboard

Heap – as a data structure – cont.

- build_heap
  - Repeats push_heap procedure
  - It takes $1+2\cdot 1+4\cdot 2+\ldots+n/2\cdot \lg n=O(n\lg n)$
- heap_sort
  - Build the heap (build_heap takes $O(n\lg n)$)
  - pop_heap (takes $O(\lg n)$)
  - add the poped element at bottom+1 (i.e. out of the heap, in the array)
  - It takes $O(n\lg n)$ (to build the heap)+ $O(n\lg n)$ (n times a pop operation)

Heap – comparison in building the heap

Approach	**Sol 1** (heapify)	**Sol2**(pop/push)
1 el approach	sinks the top (root)	bubbles a leaf
	O(h)	O(h)
all els(build heap) approach	bottom-up (starts with the last nonleaf el)	top-down (adds a new leaf)
Time to build	O(n)	O(nlgn)
advantage	faster	variable dim
drawback	fixed dim	slower
usage	sorting	priority queues

Heap-Sort - Conclusions

- Optimal sorting algorithm
- In practice, quicksort, even not optimal by initial design (with its default/classic approach) behaves better
- Good quicksort implementations (avoid worst case OR ensure best case always) ARE optimal

Recap

- Review
- Divide et impera evaluation
- Particular cases
- Master Theorem
- Sorting
  - Heap Sort

Required Bibliography

- From the Bible – Chapter 6 (Heapsort), 8.1 (sorting lower bound)