A Problem

Suppose you have a hospital in which patients are attended based on their ages. The oldest are always first, no matter when he/she got in the queue.

You cannot just keep track of the oldest one because if you pull he/she out, you don't know the next oldest one.

How to solve this problem?

CPT108 Data Structures and Algorithms

Lecture 9

Sorting

Heap Sort

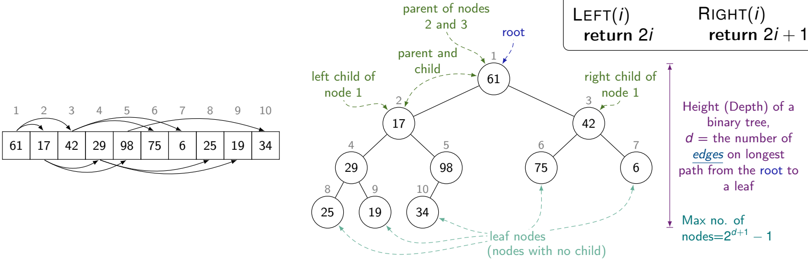
Array and Binary tree

PARENT(i)
return $\lfloor i/2 \rfloor$

Note here that, i starts from 1, instead of 0!

LEFT(i)
return $2i$

RIGHT(i)
return $2i+1$



Array | Binary tree

Other properties of a binary tree:

- Each *node* can have zero, one, or two children
- for a node `x`, we denote the left child, right child, and the parent of `x` as `left(x)`, `right(x)`, and `parent(x)`.

Binary tree (cont.)

Complete binary tree is a binary tree such that

- All levels of the tree are filled completely except the lowest level nodes, i.e., the leaf nodes
- All leaf nodes must lean towards the left
- The last element may not have sibling
- Number of nodes and Height (depth)
  - A complete binary tree with N nodes has height $\log N$
  - A complete binary tree with height d has, in total, $2^{d+1} - 1$ nodes, where d is the depth of the tree
    (proof by *mathematical induction)*

Note: The largest depth of a binary tree of N nodes is N.

(Binary) Heap

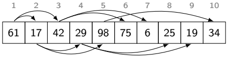
- Based on the concept of complete binary tree, a (binary) heap can be implemented using an array

PARENT(i)
  return $\lfloor i/2 \rfloor$

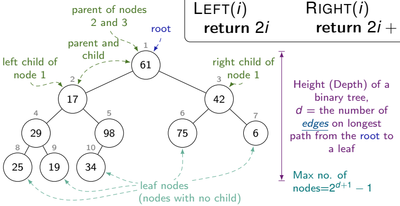
Note here that, i starts from 1, instead of 0!

LEFT(i)
  return $2i$

RIGHT(i)
  return $2i + 1$



Array



Binary tree

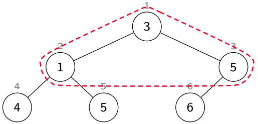
Potential Problems

- An estimate of the maximum heap size is required in advance in implemented heap using an array
  - but we can resize the array if reeded
- Side notes: it is not wise to store normal binary tree in arrays as it may generate many holes

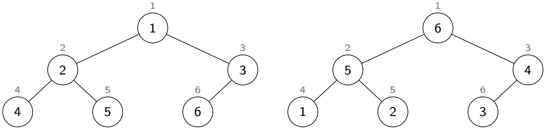
(Binary) Heap (cont.)

Two types, according to the heap ordering property:

- Max-heap,
  - i.e., $A[Parent(i)] \geq A[i]$
- Min-heap,
  - i.e., $A[Parent(i)] \leq A[i]$



Not a heap



Heaps

Maintaining the heap property: Heapify

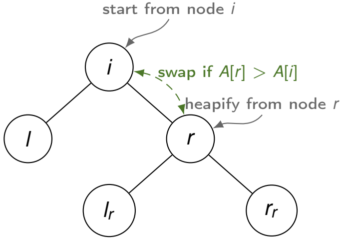
Let's create a max-heap

- i.e., $A[Parent(i)] \geq A[i]$

Algorithm

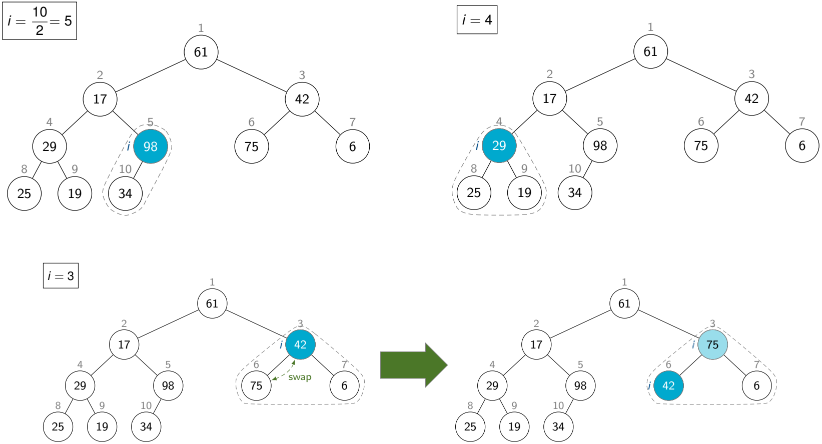
MAX-HEAPIFY(A, i)

```
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 ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```



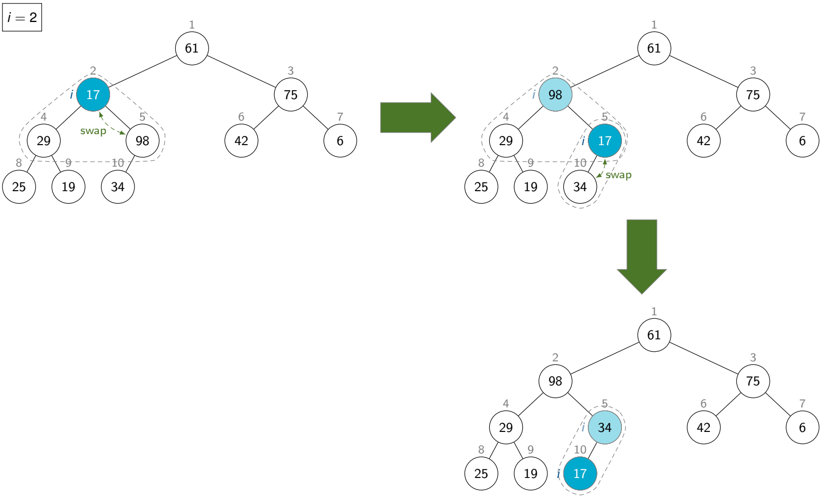
Maintaining the heap property: Heapify

Example



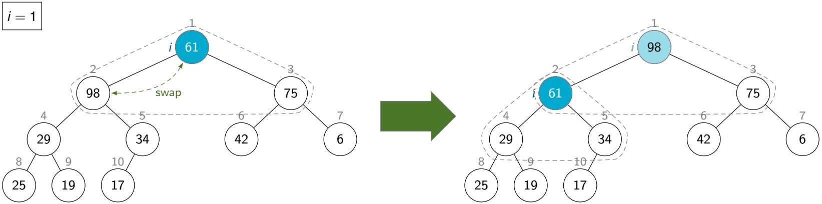
Maintaining the heap property: Heapify

Example (cont.)



Maintaining the heap property: Heapify

Example (cont.)



Done!!

We have completed the heap set up *process!*

BUILD-MAX-HEAP(A, n)

1 $A.heap\text{-}size = n$
2 **for** $i = \boxed{\lfloor n/2 \rfloor \textbf{ downto } 1}$
3 MAX-HEAPIFY(A, i)

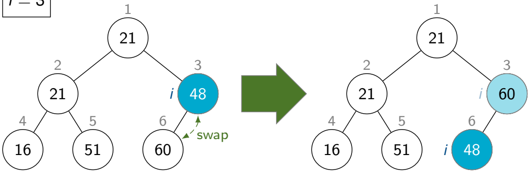
Why starts from $\lfloor n/2 \rfloor$ here?

Building the Heap: Heapify

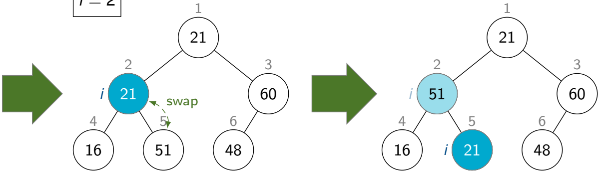
Exercise

1	2	3	4	5	6
21	21	48	16	51	60

$\boxed{i = 3}$

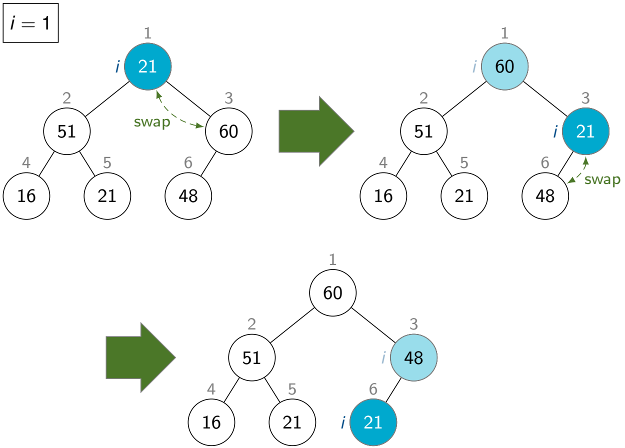


$\boxed{i = 2}$



Building the Heap: Heapify

Exercise (cont.)



Heap sort

Pseudocode

Build a heap from the given input array
Repeat the following steps until the heap contains only one element
  Swap the root element with the last element of the heap
  Remove the last element of the heap (which is now in correct position)
  Heapify the remaining elements of the heap

Algorithm

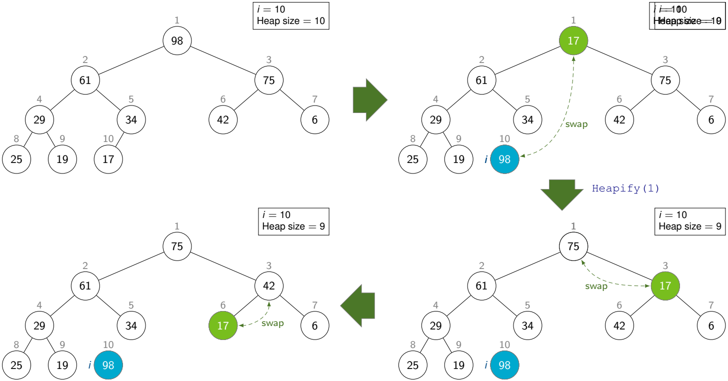
```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i = n downto 2
    swap A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

Heap sort

Example

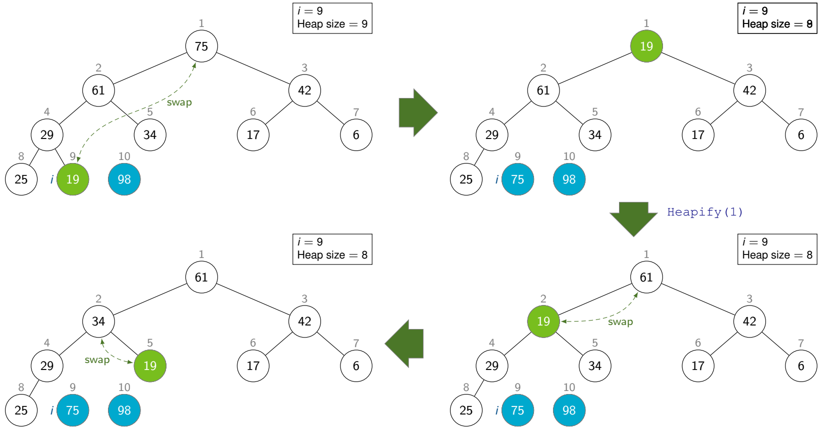
Consider the heap that we just built in the example.

1	2	3	4	5	6	7	8	9	10
98	61	75	29	34	42	6	25	19	17



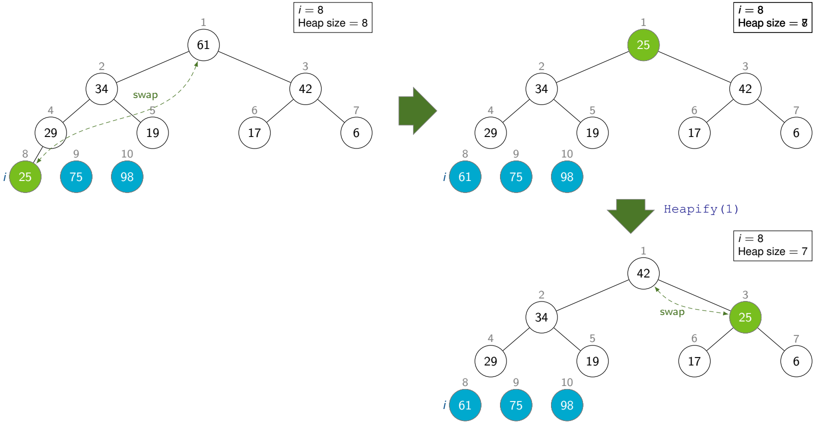
Heap sort

Example (cont.)



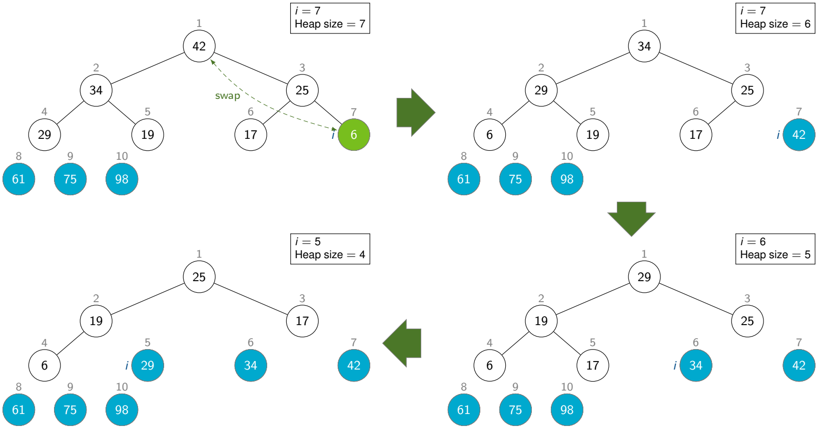
Heap sort

Example (cont.)



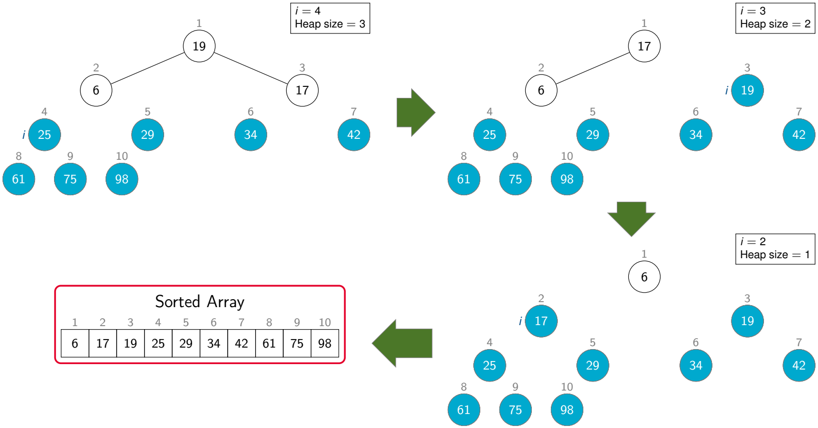
Heap sort

Example (cont.)



Heap sort

Example (cont.)

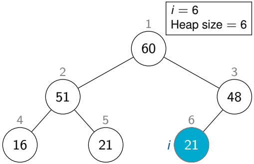


1	2	3	4	5	6	7	8	9	10
6	17	19	25	29	34	42	61	75	98

Heap sort

Exercise

Consider the heap we have created in the exercise.



Heap sort

Exercise (cont.)

Heap sort

Exercise (cont.)

Complexity

Algorithms

MAX-HEAPIFY(A, i)

```
l = LEFT(i)   O(1)
r = RIGHT(i)   O(1)
if l ≤ A.heap-size and A[l] > A[i]   O(2)
      largest = l   O(1)
else largest = i   O(1)
if r ≤ A.heap-size and A[r] > A[largest]   O(2)
      largest = r   O(1)
if largest ≠ i   O(1)
      exchange A[i] with A[largest]   O(3)
      MAX-HEAPIFY(A, largest)   lg n times
```

Lines 1–9: $O(c)$, where c is a constant

Whole procedure: $c \lg n = O(\lg n)$

BUILD-MAX-HEAP(A, n)

```
A.heap-size = n   O(1)
for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i)   O(lg n)
```

Lines 2–3: $\frac{n}{2}$ times

Whole procedure: $1 + \left(\frac{n}{2}\right) \lg n = O(n \lg n)$

HEAPSORT(A, n)

```
BUILD-MAX-HEAP(A, n)   O(n lg n)
for i = n downto 2
      swap A[1] with A[i]   O(3)
      A.heap-size = A.heap-size − 1   O(1)
      MAX-HEAPIFY(A, 1)   O(lg n)
```

Lines 2–5: $n - 1$ times

Whole procedure: $n \lg n + (n-1)(3 + 1 + \lg n) = O(n \lg n)$

Priority queue

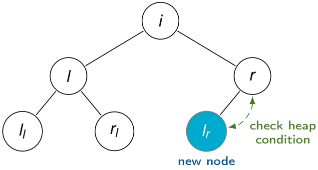
Functions required

- Insert new element to the end of the queue
- Remove element with the highest priority from the queue

Priority queue (cont.)

Insertion

- Add the new element to the end of the heap
- Then, we can maintain the max-(min-) heap property by:
  - Re-build the heap
    - $\Rightarrow$ time consuming ($O(n \lg n)$)
  - Restore the max-(min-)heap property if violated:
    - $\Rightarrow$ "bubble-up": if the parent of the elements is smaller (larger) than the new element, then interchange the parent and child ($O(\lg n)$)



(Binary) Heap

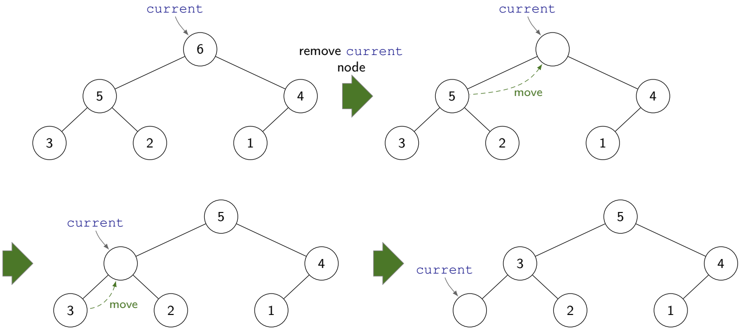
Deletion

First Attempt

1. Mark `root` as `current` node
2. Delete `current` node
3. An empty spot is created
4. Compare the two children of the `current` node
5. Bring the `larger` (`lesser`) of the two children of the `current` node to the empty spot
6. Mark the `larger` (`lesser`) node as the new `current` node
7. Go to Step 2 until the `current` node has no child

(Binary) Heap

Deletion: First Attempt (cont.)



- Heap property is preserved, but *completeness* is *NOT* preserved!

(Binary) Heap

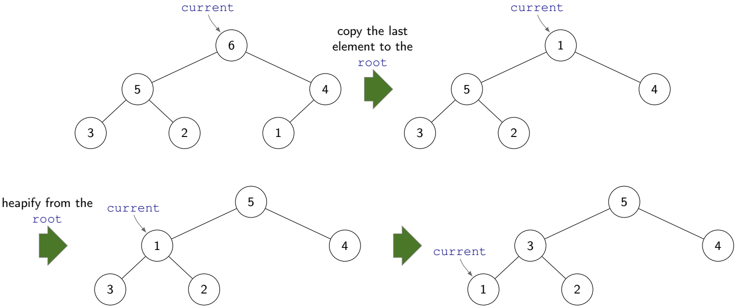
Deletion (cont.)

HEAPIFYDELETE

1. Copy the last element to the `root` and reduce the heap size by 1
   - i.e., overwrite the maximum (minimum) element stored at `root`
2. Heapify the heap from the `root`

(Binary) Heap

Deletion: HeapifyDelete (cont.)



(Binary) Heap (cont.)

Heap Properties

- Insert in $O(\lg N)$ time
- Locate the current maximum (or respectively, minimum) in $O(1)$ time
- Delete the current maximum (or respectively, minimum) in $O(\log n)$ time
- Auxiliary space can be $O(1)$ for iterative implementation

Heap sort: Complexities

	Worst	Best
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$
Bubble sort	$O(n^2)$	$O(n^2)$
Improved bubble sort	$O(n^2)$	$O(n)$
Merge sort	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n^2)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \lg n)$

Heap sort (Williams, 1964)

Remarks

- A *(binary) heap* is a useful data structure if quick access to the largest (or smallest) element is need because that item will always be the first element in the array.
- The remainder of the array is kept partially unsorted. So, it is a good way to deal with incoming events or data and always have access to the earliest/biggest.
- Useful for priority queues, schedulers (where the earliest item is desired), etc.

Heap sort

Summary (Geeksforgeeks.org, 2024)

Advantages

- Combine the good qualities of insertion sort (sort in place) and merge sort (speed, $O(n \log n)$ in all case)
- Memory usage can be minimal if implemented with iterative heapify (instead of recursive one)
- Simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion

Drawbacks

- Costly: since the constants are higher compared to merge sort even if the time complexity are the same
- Unstable: it might rearrange the relative order
- Efficient: not very efficient when working with highly complex data

Reading

- Chapter 6, Cormen (2022)

References

Geeksforgeeks.org (2024). *Heap Sort — Data Structure and Algorithm Tutorials*. Online: https://www.geeksforgeeks.org/heap-sort/. [last accessed: 20 Mar 2024].

Williams, J. W. J. (June 1964). “Algorithms 232 (Heapsort)”. In: *Communications of the ACM* 7.6, pp. 347–349.