Parma University, Math. Department

# Algorithms and Data Structures (Heaps and Heap Sort)

prof. Marco Pellegrini



## Summary of lecture 5

- A DS for Priority queues: Heaps,
  Heap-sort,
  Data Compression (Huffman codes)

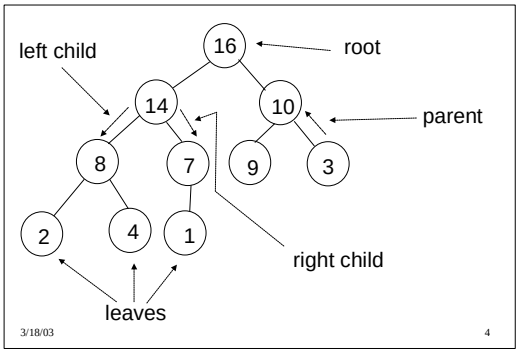


## Sorted arrays and heaps

- If we have a SORTED ARRAY A, searching is fast using BINARY-SEARCH. Insertion of new elements is rather slow (LINEAR in the WORST CASE).
- We will see how an array can be organized in a DS called HEAP (MIN-HEAP, MAX-HEAP) so that insertion of new elements is fast.
- In HEAPS searching is not fast, but we can FIND and DELETE the largest (or smallest) element efficiently.
- Heaps implement the Priority Queue ADT.



## HEAP as a TREE






## Some tree properties

- A TREE has nodes and links.
- The NODES hold the DATA.
- The LINKS link the NODES.
- The top NODE is called ROOT.
- The bottom nodes are called LEAVES
- Other nodes are called INTERMEDIATE NODES
- Going up through a link we go from a NODE to its PARENT.
- Going down the left (right) from a NODE we go to its LEFT CHILD (RIGHT CHILD).



## More on trees

- A Tree where each node can have at most two children (LEFT and RIGHT) is called a BINARY TREE.
- A BINARY TREE is *COMPLETE* if all NODES except the leaves have two children.
- The HEIGHT of a NODE is the number of links between the NODE and the ROOT.
- The HEIGHT of a TREE is the maximum of all the height of its nodes.
- Nodes at the same height form a LEVEL.

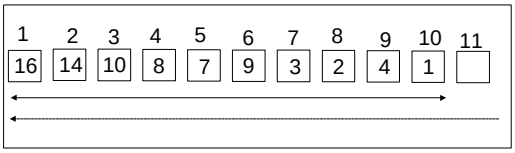
## HEAP as a TREE

- A HEAP is a COMPLETE BINARY Tree in which we have possibly missing some leaves consecutively starting from the rightmost leaf.
- Moreover a HEAP MUST SATISFY the PROPERTY: VALUE(Parent(i)) ≥ VALUE(i), where i is a NODE of the HEAP.
- Clearly the largest value will be at the ROOT.
- There is no relation between the value of a right child and the value of a left child.



## How to Squeeze a HEAP in an ARRAY

- Put the ROOT at index 1.
- Put the LEFT CHILD of i at position 2i.
- Put the RIGHT CHILD of i at position 2i+1.
- As a consequence Parent(i) = $\lfloor i/2 \rfloor$.






## Observations

- The array holding the HEAP can and should be larger of the HEAP itself so that we may insert new elements in the HEAP.
- So HEAP-SIZE(A) ≤ LENGTH(A).
- Exercise: check that the array and the tree given in previous examples are the same heap.
- Note that the HEAP is NOT SORTED!
- Travelling on a HEAP in its array form is easy using binary coding of the indices and shift operations.



## Operations on HEAPS.

- HEAPIFY(A,i) is a procedure that takes as input an array A and an index i, the sub-trees rooted in LEFT(i) and RIGHT(i) are HEAPS. As result of running this operation the subtree rooted at i is a HEAP.
- BUILD-HEAP(A) takes an array A and organizes its content into a HEAP.
- INSERT(A,x) inserts x in a HEAP A so that the final data structure is still a HEAP.
- EXTRACT-MAX(A) returns the maximum element in the HEAP A and deletes it from A. The final data structure is still a HEAP.

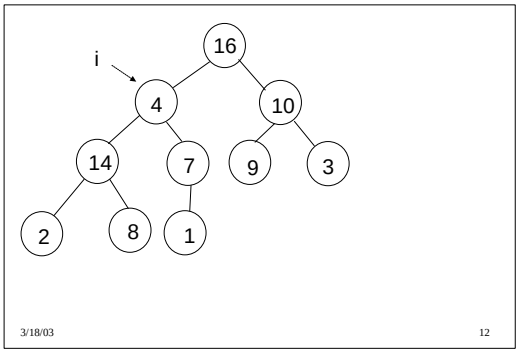


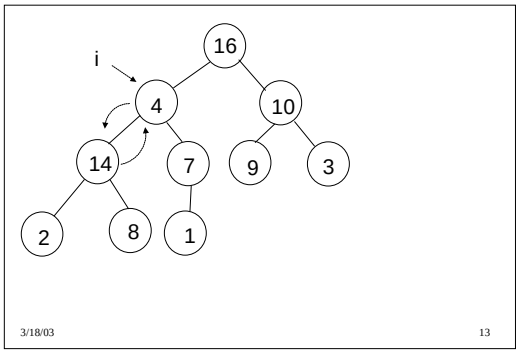
## Trees and Arrays

- Trees are easy to visualize.
- It is easy to explain HEAPIFY on a tree.
- Arrays are easy to implement and use little extra storage beyond that needed to hold the data.
- Access to parents and children using address arithmetic is fast on most computers.
- It is easier to analyze an algorithm running on a tree than one working on an array.

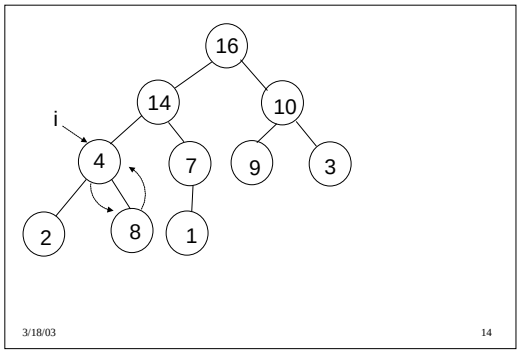


## Example of HEAPIFY

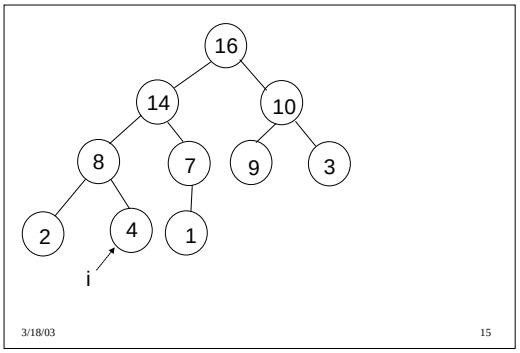











## HEAPIFY on the Array

```
HEAPIFY(A: array ,i: index)
left := LEFT(i)
right := RIGHT(i)
IF (left ≤HEAPSIZE(A)) AND (A[left] > A[i])
  THEN largest := left
  ELSE largest := i
END IF
IF (right ≤ HEAPSIZE(A)) AND (A[right] > A[largest])
  THEN largest := right
END IF
IF largest ≠ i
  THEN EXCHANGE A[i] WITH A[largest]
      HEAPIFY(A, largest)
END IF
```



## Analysis of HEAPIFY

A Complete Binary Tree with n leaves has 2n-1 nodes in total.

The height of a complete binary tree with n leaves is log n.

A HEAP with N nodes has
HEIGHT = $\lceil \log (N+1)/2 \rceil$.

Since at each call of HEAPIFY we decrease by one the level of the current node there can be at most O(log N) calls.

So HEAPIFY uses time O(log n).



## Exercise

- Simulate HEAPIFY(A,3) with A =[27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]
- Write the content of A after the call.
- Is A an HEAP after the call?

# Building a HEAP.

- We have an array A[1..n] and we want to organize its content in a HEAP.
- If we look at A as if it were already a HEAP, then the elements $A[\lfloor n/2 \rfloor+1,..,n]$ are the leaves and each leaf satisfy the HEAP property.
- So we need to reorganize only the elements in $A[1,..,\lfloor n/2 \rfloor]$.
- 
```
BUILD-HEAP(A)
  HEAPSIZE(A) := Length(A)
  FOR i := ⌊ length(A)/2 ⌋ DOWNTO 1
    DO HEAPIFY(A,i)
  END FOR
```



# Observations on BUILD-HEAP

- The order in which we make the calls to HEAPIFY make sure that, when we call HEAPIFY(A,i) the subtree rooted at LEFT(i) and RIGHT(i) are already been made into HEAPS.
- Analysis: BUILD-HEAP calls HEAPIFY (n/2) times and each call costs O(log n).
- So BUILD-HEAP takes time O(n log n).
- If we make a more careful analysis (using the levels of the heap) we can obtain a better result, that is O(n).



# HEAP-SORT

Sorting in decreasing order

- The idea is to take the unsorted input array A and make it into a MAX-HEAP.
- Then we take elements out of the heap and the heap shrinks. We place elements in sorted order at the end of the array and the sorted part grows.
- 
```
HEAP-SORT(A: array of numbers)
  BUILD-HEAP(A)
  FOR i := length(A) DOWNTO 2
    DO EXCHANGE A[1] WITH A[i]
      HEAPSIZE(A) := HEAPSIZE(A)-1
      HEAPIFY(A,1)
  END FOR
```



# Observations and homework

- What is the asymptotic time function of HEAP-SORT?
- Note that HEAP-SORT uses only 1 array, that is the same used to provide the input.
- If we want to sort in Increasing order (as we made so far) we use a MIN-HEAP which has the condition $A[parent(i)] \leq A[i]$.
- HW1.4: Rewrite the code for HEAPIFY, BUILD-HEAP, HEAPSORT for a MIN-HEAP.



# Dynamic operations: Extract-Max

- 
```
EXTRACT-MAX(A: max heap)
  IF HEAPSIZE(A) < 1 THEN ERROR
  max := A[1]
  A[1] := A[HEAPSIZE(A)]
  HEAPSIZE(A) := HEAPSIZE(A) -1
  HEAPIFY(A,1)
  RETURN(max)
```
- Is it true that this procedure returns the maximum element in the HEAP and deletes it from the heap and what is left is still a HEAP?
- What is the time complexity?



# Dynamic operations: Insertion

- 
```
INSERT(A: maxheap, x: element)
  HEAPSIZE(A) := HEAPSIZE(A) +1
  i := HEAPSIZE(A)
  A[i] := x
  WHILE i > 1 AND A[parent(i)] < A[i]
    DO  EXCHANGE A[i] WITH A[parent(i)]
       i := parent(i)
  END WHILE
```
- The idea is to place x as the last leaf and then make it move up the tree until it finds its place.
- We never check the other child of any node. Why?

## An application: job scheduling

- Suppose a computer receive requests for jobs (programs to execute, clients to serve, airplanes to land).
- Each job has a priority and the first to be executed is the one with maximum priority.
- Then a heap is an efficient data structure to handle such a situation since insertion, obtain the maximum priority job and cancel it when completed can be performed efficiently (in time O(log n) for a heap of n elements at the time of the operation).



## Conclusions

- We have seen a data structure (HEAP) that organize data in an array so that we can efficiently:
- Insert new data
- Access the maximum element
- Access and Delete the maximum element
- This has applications in SORTING (HEAP-SORT), Job Scheduling.
- Example of analysis using height of trees



## More Homework

- HW1.5: Write code of INSERT and EXTRACT-MIN for a MIN-HEAP.
- HW1.6 Read pages 337-340 of CLR on the use of priority queues to build efficient data compression scheme for files.