Algorithm and Data Structures
Lecture notes: Heapsort, Cormen Chap. 6

Lecturer: Michel Toulouse

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

25 mai 2020

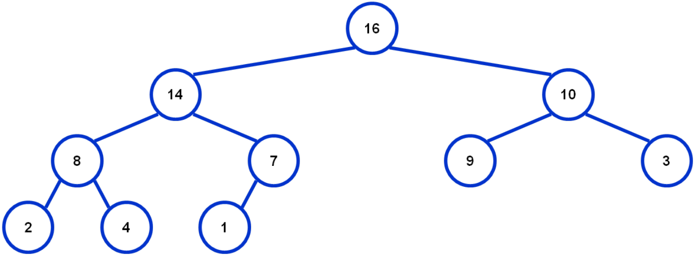
Outline

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

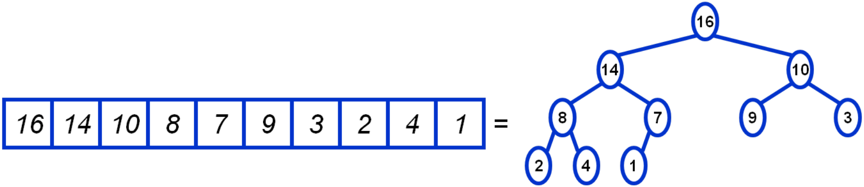
- A heap is a complete binary tree



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

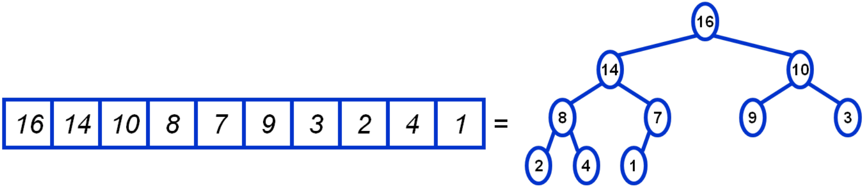
Heaps

- 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

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 :

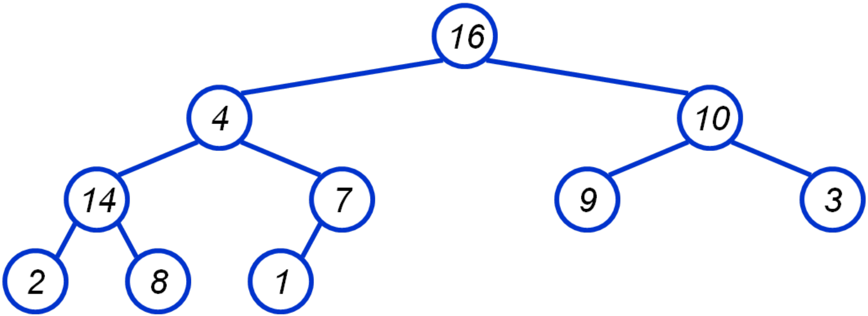
- Consider node i in the heap with children l and r
- Nodes l and r are each the root of a subtree, each assumed to be a heap
- Problem : Node i may violate the heap property
- Solution : let the value of node i "float down" in one of its two subtrees until the heap property is restored at node i

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) ;
```

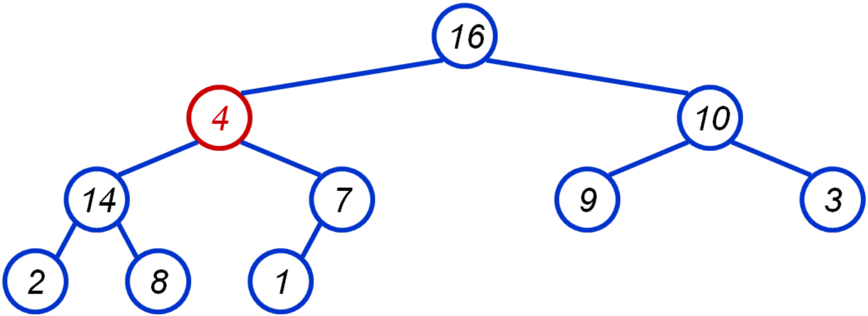
Heapify() Example



$A =$

16	4	10	14	7	9	3	2	8	1

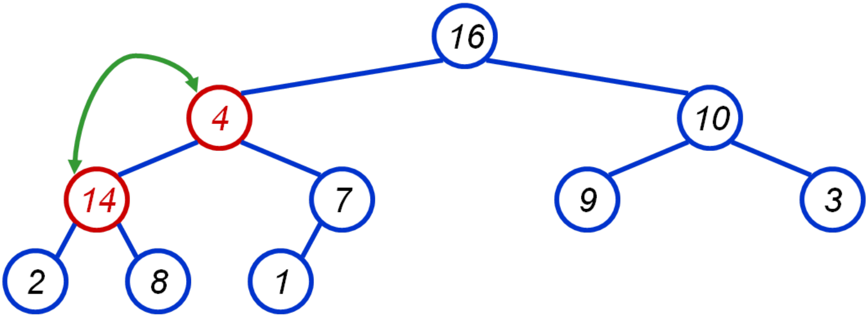
Heapify() Example



$A =$

16	4	10	14	7	9	3	2	8	1

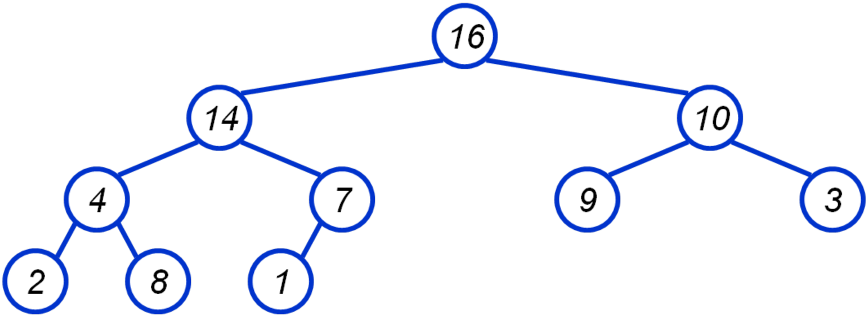
Heapify() Example



A =

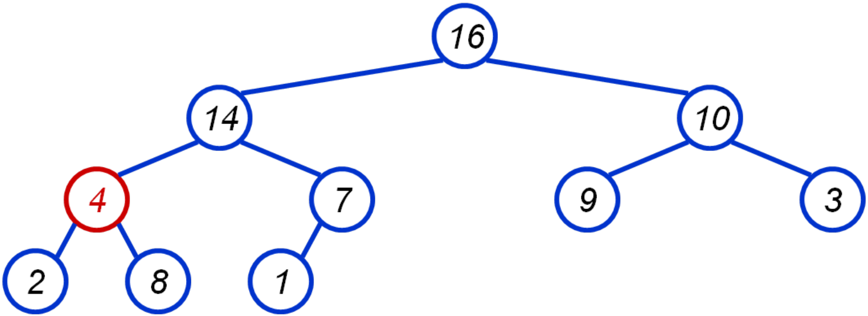
16	4	10	14	7	9	3	2	8	1

Heapify() Example



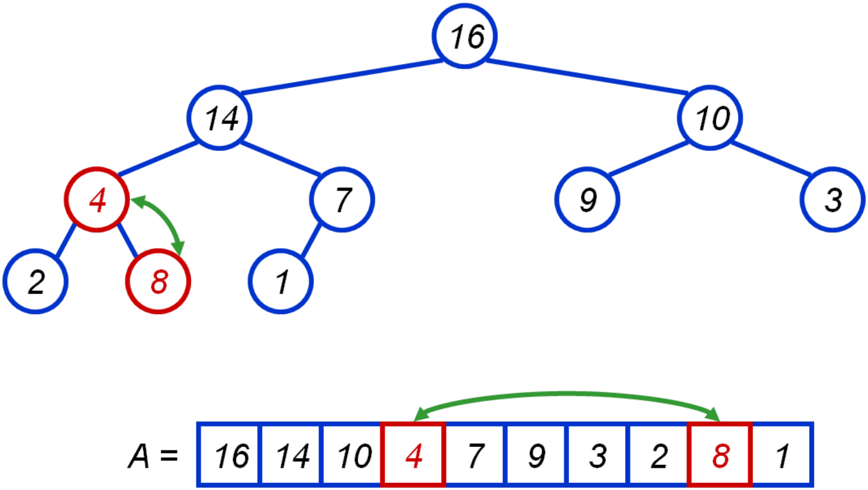
$A =$	16	14	10	4	7	9	3	2	8	1

Heapify() Example

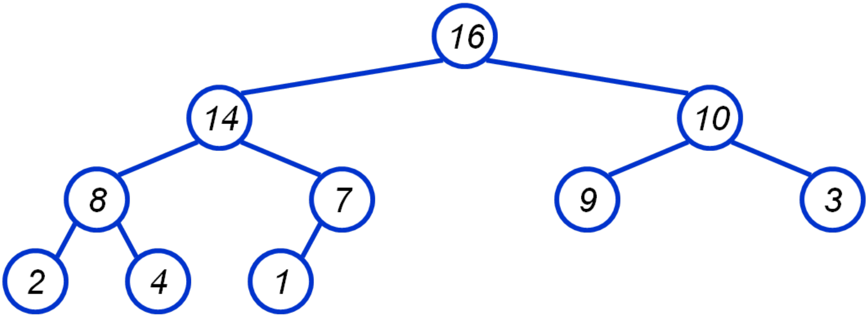


A =	16	14	10	4	7	9	3	2	8	1

Heapify() Example



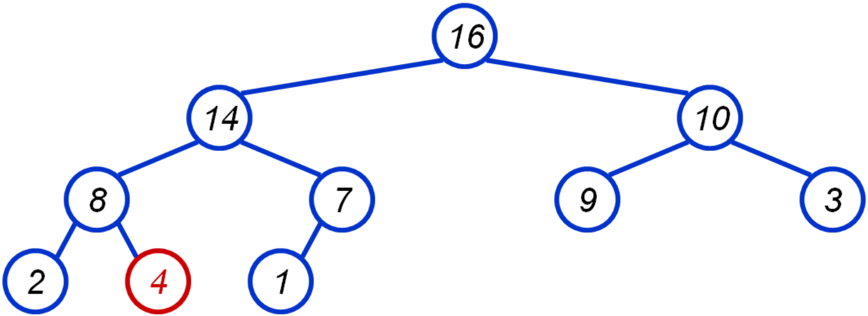
Heapify() Example



A =

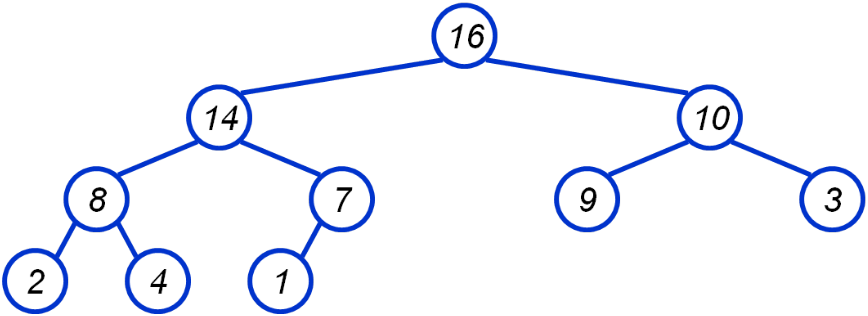
16	14	10	8	7	9	3	2	4	1

Heapify() Example



$A =$	16	14	10	8	7	9	3	2	4	1

Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

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 Heapify() is in $O(1)$
- If the heap at i has n elements, how many elements can the subtrees at l or r have? Answer : at most 2n/3 (worst case : bottom row 1/2 full)
- So time taken by Heapify() is given by the recurrence

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

- Master Theorem applies to solve this recurrence, which corresponds to the case 2 of the restricted Master Theorem.

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

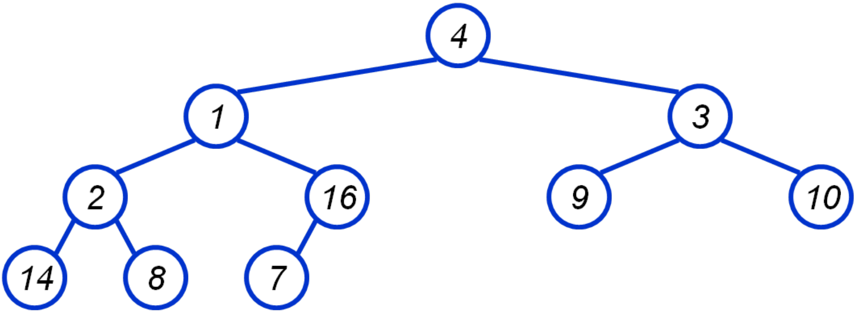
Heap Operations : BuildHeap()

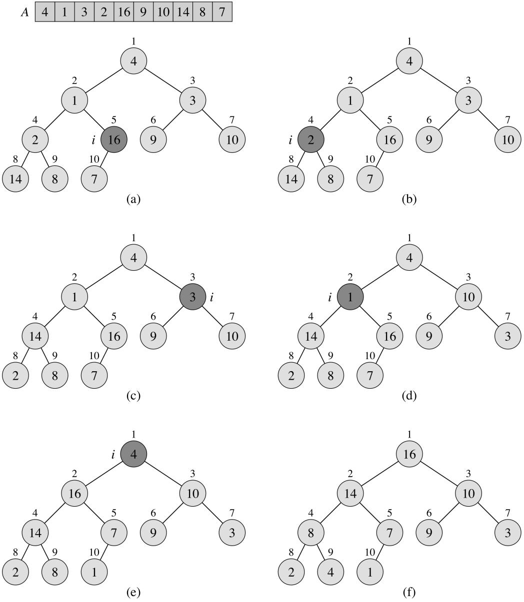
- We can build a heap in a bottom-up manner by running Heapify() on successive subarrays
  - Note : for array of length n, all elements in range $A[n/2+1..n]$ are heaps (Why ?)
  - Walk backwards through the array from n/2 to 1, calling Heapify() on each node.
- given an unsorted array A, make A a heap

BuildHeap(A)
  heap_size(A) = length(A) ;
  for (i = $\lfloor length(A)/2 \rfloor$ 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)

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

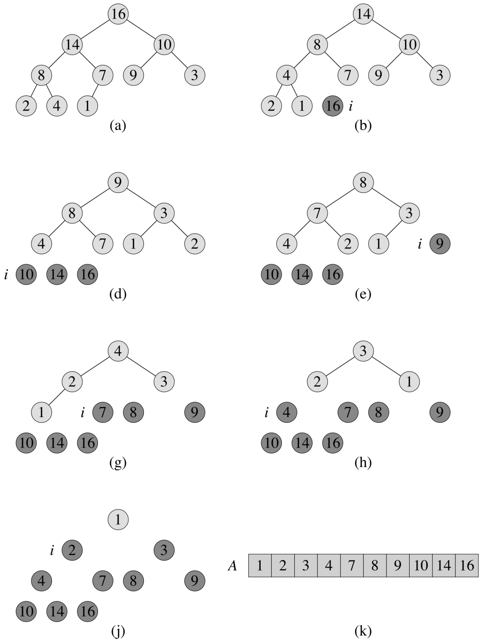
$$
\begin{aligned}
\sum_{h=0}^{\infty} \frac{h}{2^h} &= \sum_{h=0}^{\infty} h(\frac{1}{2})^h \\
&= \sum_{h=0}^{\infty} hx^h \text{ where } x = \frac{1}{2} \\
&= \frac{1/2}{(1-\frac{1}{2})^2} \text{ the closed form of } \sum_{h=0}^{\infty} h(\frac{1}{2})^h \\
&= 2
\end{aligned}
$$

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) ;
```

Heapsort



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 $\log n$

Exercises

1. Insertion sort and merge sort are stable algorithms while heapsort and quicksort are not. Can you explain why this is so ?
2. Does this array $A = [23, 11, 14, 9, 13, 10, 1, 5, 7, 12]$ is a heap (max-heap) ? If not makes it a heap.
3. Run $Heapify(A, 3)$ on the array $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$
4. $Heapify(A, i)$ in the class notes is a recursive algorithm. Write an equivalent iterative algorithm.
5. Run the algorithm $BuildHeap(A)$ on the array $A = [8, 4, 27, 10, 72, 19, 9, 22, 6]$
6. Run $Heapsort(A)$ on the the array $A = [3, 15, 2, 29, 6, 14, 25, 7, 5]$
7. What is the running time of *Heapsort* on an array A of length n that is sorted in decreasing order ?
8. Show what happens when $Heap - Insert(A, 10)$ is run on the heap $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$