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

Lecturer: Michel Toulouse

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

7 juin 2021

Outline

Introduction

Heaps

Operations on heaps
- Heapify
- Buildheap

Heapsort

Appendix : Priority queues

New exercises

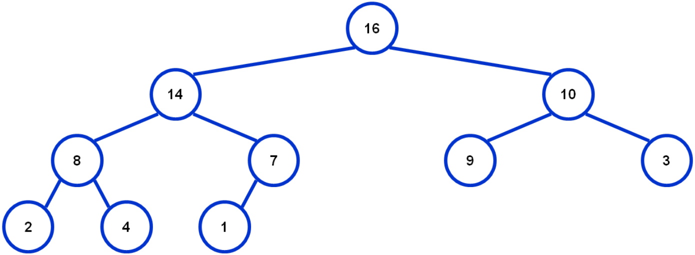
Exercises

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

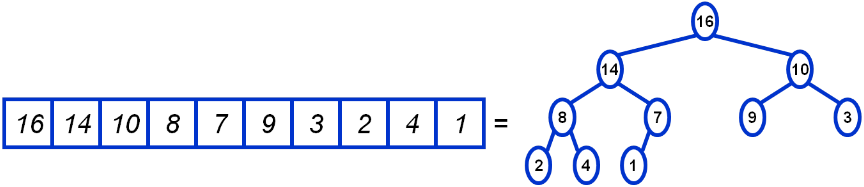
- A heap is a complete binary tree



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

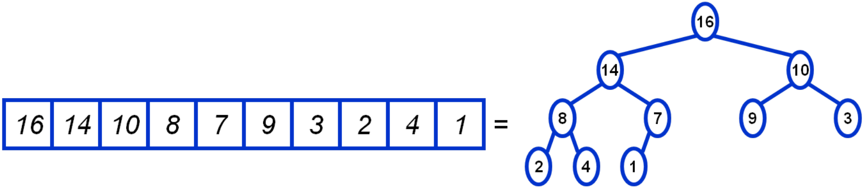
Heaps

- 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

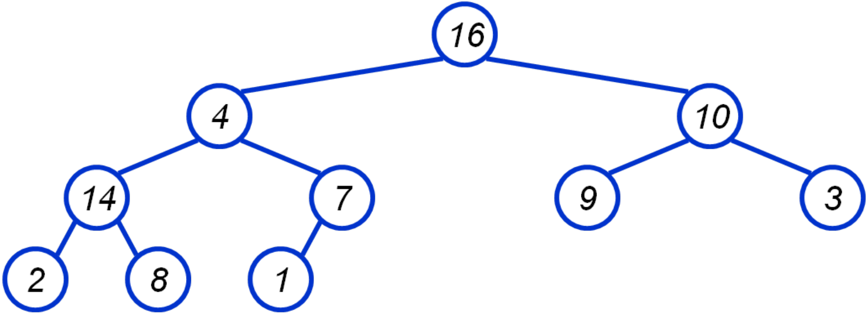
Heap Operations

There are two main heap operations : heapify and buildheap.

Heapify() : restore the heap property :

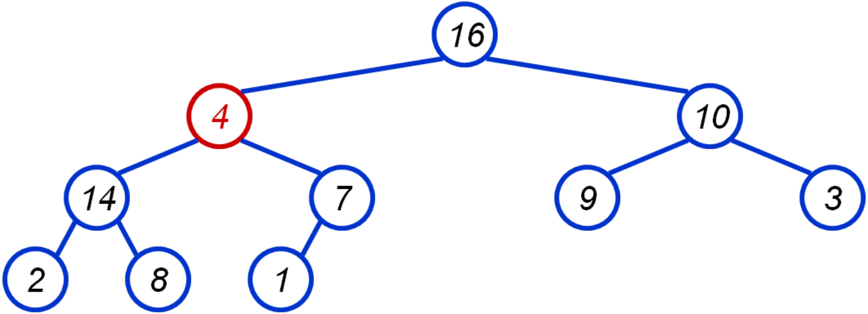
- Consider node i in the heap with children l and r
- Nodes l and r are each the root of a subtree, each assumed to be a heap
- Problem : Node i may violate the heap property
- Solution : let the value of node i "float down" in one of its two subtrees until the heap property is restored at node i

Heapify() Example



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

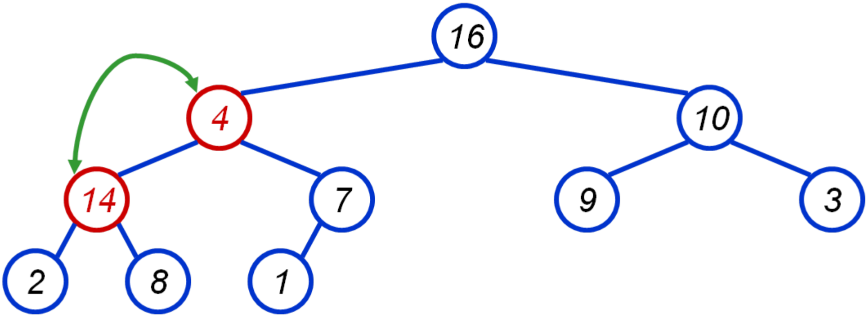
Heapify() Example

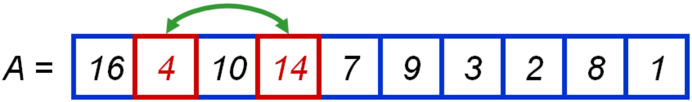


$A =$

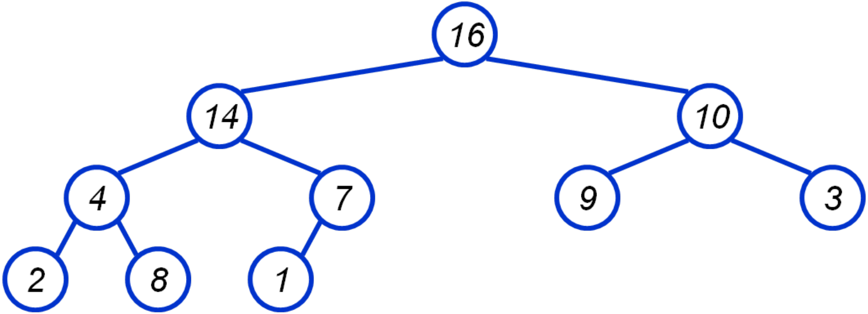
16	4	10	14	7	9	3	2	8	1

Heapify() Example



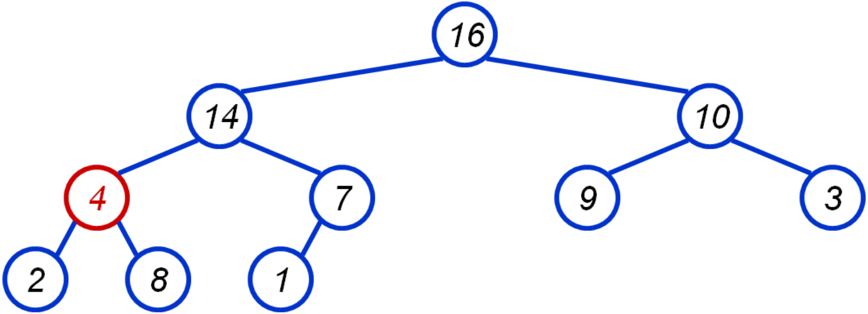


Heapify() Example



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

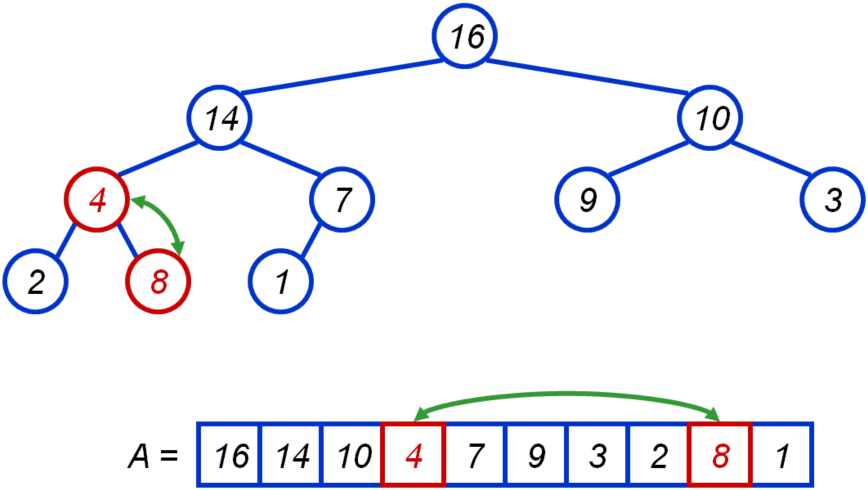
Heapify() Example



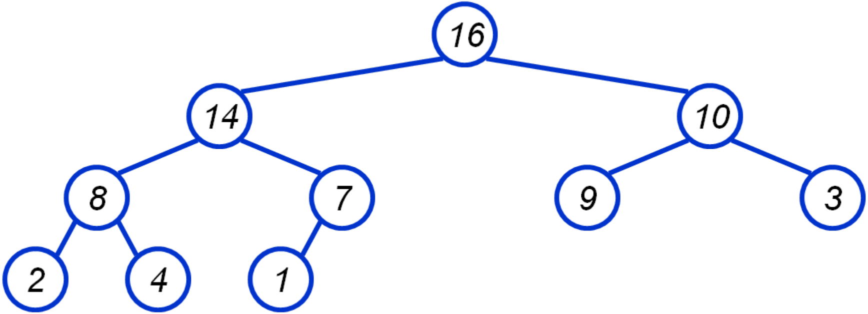
$A =$

16	14	10	4	7	9	3	2	8	1

Heapify() Example



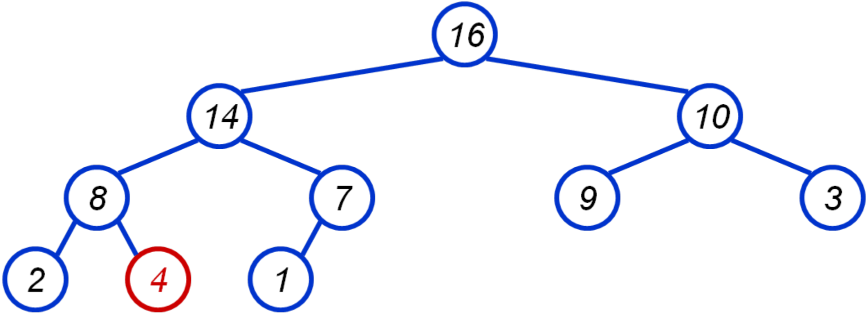
Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

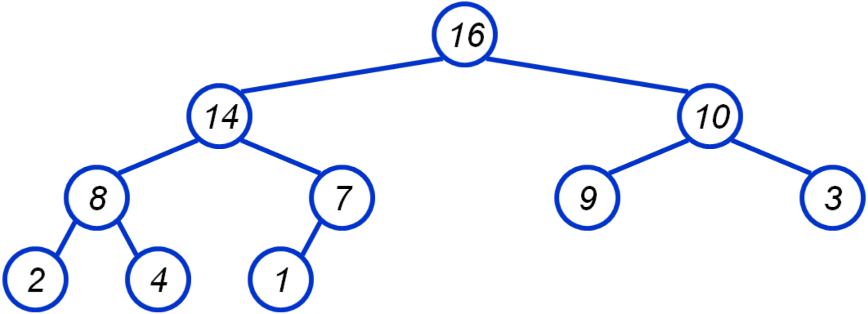
Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

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 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.

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

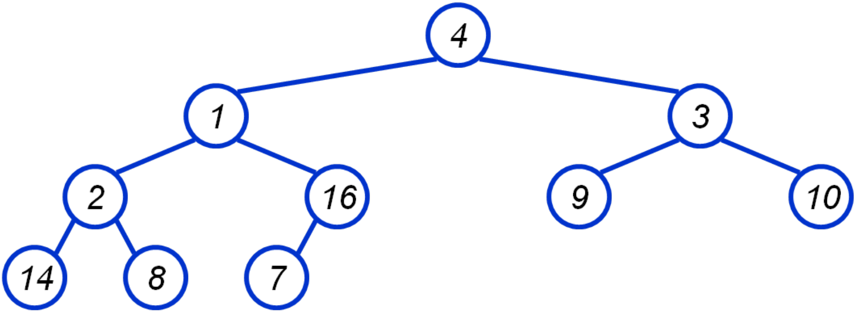
Heap Operations : BuildHeap()

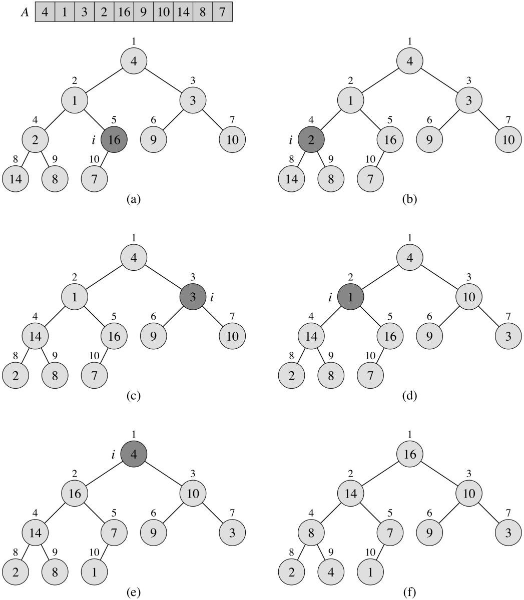
- We can build a heap in a bottom-up manner by running Heapify() on successive subarrays
  - Note : for array of length n, all elements in range $A[\lfloor n/2 \rfloor + 1..n]$ are heaps (Why ?)
  - Walk backwards through the array from n/2 to 1, calling Heapify() on each node.
- given an unsorted array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

```
BuildHeap(A)
  heap_size(A) = length(A);
  for (i = ⌊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 $O(h)$

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

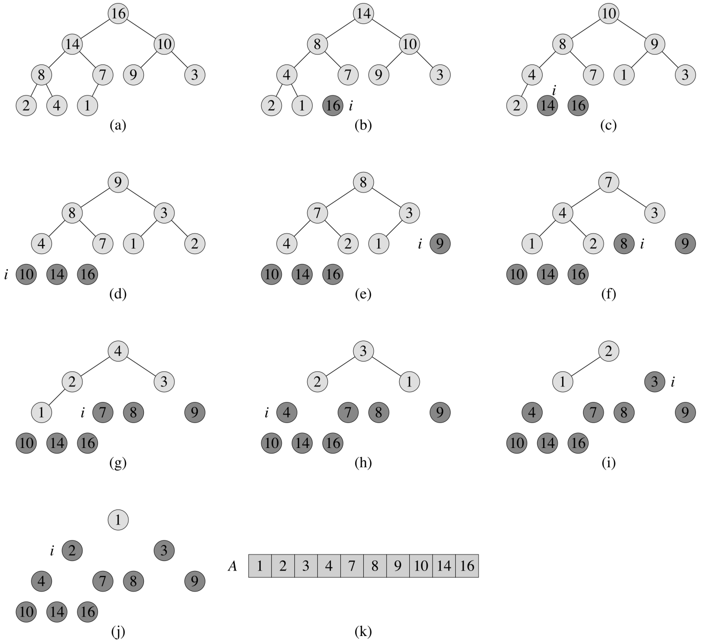
Analyzing BuildHeap() : Tight

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

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

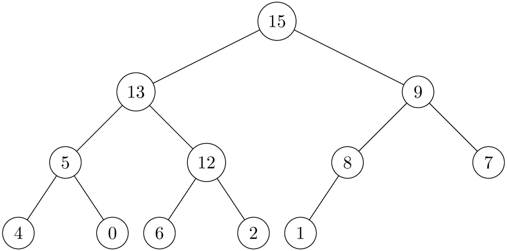
```
Insert(A, key)
   A.heap-size = A.heap-size + 1
   A[A.heap-size] = key
   i = A.heap-size
   while i > 1 and A[Parent(i)] < A[i]
      swap(A[i], A[Parent(i)])
      i = Parent(i)
```

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 $\log n$

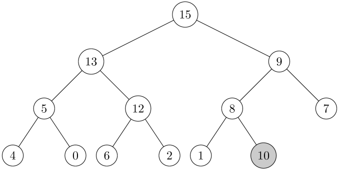
Example : Inserting an element

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

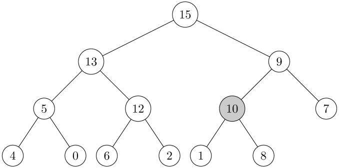
- Original heap



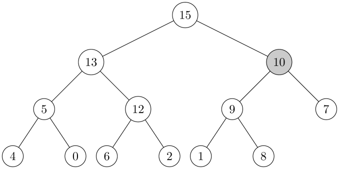
- Add the key 10 to the next position in the heap (corresponding to a new entry extending the array A).



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



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



New exercises

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)

New exercises 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

New exercises 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

1. Insertion sort and merge sort are stable algorithms while heapsort and quicksort are not. Can you explain why this is so ?
2. Run *Heapify*$(A, 3)$ on the array $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$

Exercises

4. *Heapify*(A, i) in the class notes is a recursive algorithm. Write an equivalent iterative algorithm.

Exercises

6. Run *Heapsort*(A) on the the array $A = [3, 15, 2, 29, 6, 14, 25, 7, 5]$

Exercises

What is the running time of *Heapsort* on an array A of length n that is sorted in decreasing order ?