Algorithms (COMP3600/6466)
Data Structures: Heaps

Pascal Bercher

(working in the Intelligent Systems Cluster)

School of Computing
The Australian National University

Tuesday, 29.8.2023

Australian National University

Introduction

Motivation

Recap that we want to do (at least) the following operations efficiently:

- access, i.e., search
- min/max
- insertion/deletion

Which runtime did we have for binary search trees?
$O(h)$, where h is the tree's height.

We now try to do better.

Overview

Existing operations for heaps:

- Heapify to ensure/establish heap properties
- Insertion
- ExtractMax (i.e., find and remove maximum)

All of these operations run in $O(log(n))$ (instead of $O(h)$).

Basics

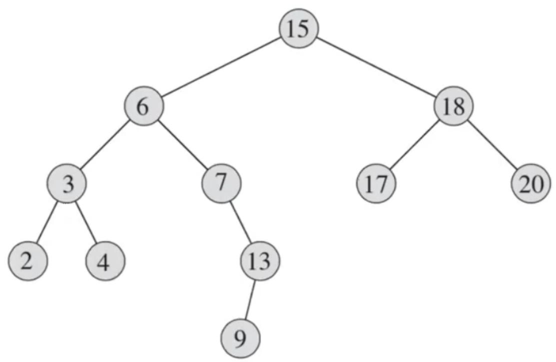
Heap

A heap is a binary tree that satisfies the *heap property*.
I.e., it holds:

- A heap is a:
  - *complete binary tree*, i.e., a perfect binary tree where missing nodes might only be right-most leaves in the last level.
  - Def.: *perfect binary tree*: all interior nodes have two children, and and all leaves are at the same level.
- Same data management as for the binary search tree:
  - Each node contains a key.
  - Each node may have satellite data.
- Each parent node has a key greater than the keys of its children. This is a *Max-heap*. Min-heaps can be defined analogously. (We only consider Max-heaps.)

Examples

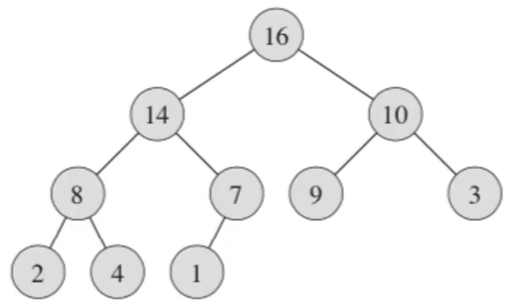
Is the following graph a heap?



→ No, e.g., 15 and 18 are wrongly ordered (for a max heap).
And it's not complete.

Examples

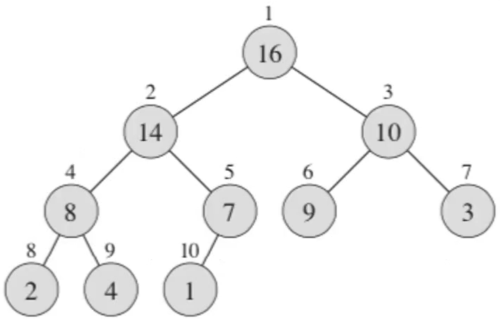
Is the following graph a heap?



→ Yes (a Max-heap)

Efficient Implementation of Heaps

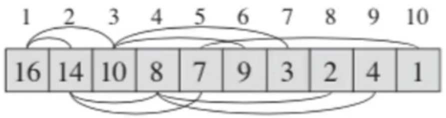
They can be stored as arrays:



$\text{Parent}(i) = \lfloor i/2 \rfloor$
$\text{Left}(i) = 2i$
$\text{Right}(i) = 2i + 1$
where i is the array position.

E.g.,
$\text{Parent}(3) = \lfloor 3/2 \rfloor = 1$
$\text{Left}(3) = 2 \cdot 3 = 6$
$\text{Right}(3) = 2 \cdot 3 + 1 = 7$

(Because it's complete!)



Heapify

Assumptions & Terminology

The **Heapify algorithm** (one call!) will be used (among others) to:

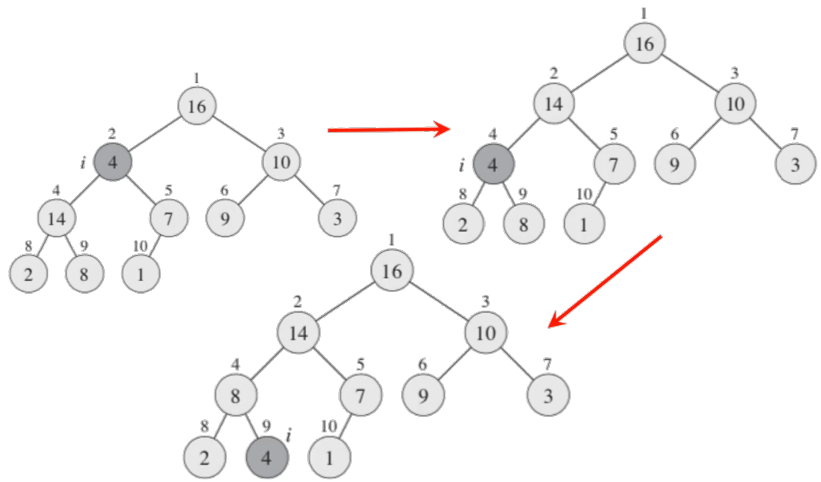
- Create a heap from an unsorted array, runs in $O(n)$
- sort an array runs in $O(n \cdot log(n))$

But the heapify algorithm itself is a single call, running in $O(log(n))$.
It assumes:

- We have a node at index i and,
- the heap property holds for both $\text{Left}(i)$ and $\text{Right}(i)$,
- but $A[i]$ might be smaller than its children.

Example

To heapify a node (that's in the tree but violating the heap property) means to traverse the tree downwards (from it) re-ordering the respective branch by switching places with the maximum.



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)
```

Note that this algorithm calls itself again on one of *i*'s children.

Runtime

The runtime is (rather obviously) in $O(log(n))$, why?

- Once heapify was called for a node x (taking constant time), it is called for only *one* of its children.
- How often can we invoke it again?
  $\rightarrow$ as often as there are children!

Since the height of a complete binary tree with n nodes is $log(n)$ we get runtime of $O(log(n))$.

Runtime (alternative proof)

We also obtain $O(log(n))$ by solving the following equation:

$$T(n) \leq T(\frac{2}{3}n) + c,$$

where T is the actual runtime of the problem (and n the number of nodes and c a constant).

That the equation only has a solution for $T(n) \in O(log(n))$ follows from the *Master theorem* (proved earlier by Ahad).

We thus only show why the equation itself holds.

Runtime (alternative proof, cont'd: why does $T(n) \leq T(\frac{2}{3}n) + c$ hold?

- We know that a call to i will perform constant (c) effort and then invoke the algorithm again for one of its children.
- So we can estimate the worst-case number of nodes that the larger sub tree may have:
  $n = 1+$ # nodes in left subtree $+$ # nodes in right subtree
- The left subtree is one level deeper than the right.

$$n = 1 + \sum_{i=0}^{h} 2^i + \sum_{i=0}^{h-1} 2^i = 1 + 2^{h+1} - 1 + 2^h - 1 = 2^h(2+1) - 1$$

Now let's bring 2^h to one side: $2^h = \frac{n+1}{3}$

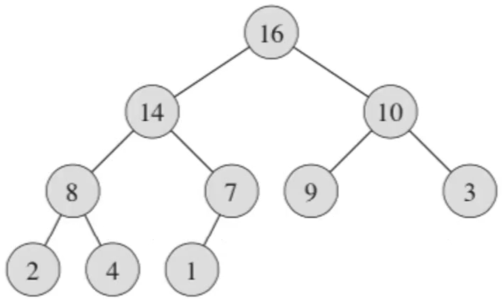
Now we can estimate the nodes in the left subtree: $2^{h+1} - 1 =$

$$2 \cdot 2^h - 1 = 2 \cdot \frac{n+1}{3} - 1 = 2 \cdot (\frac{n}{3} + \frac{1}{3}) - 1 = \frac{2}{3}n - \frac{1}{3} \leq \frac{2}{3}n$$

Insertion & Increase Key

Example

- Assume a given heap. We want to insert a key and establish the heap property again.
- Intuition: Insert it at the "next free" position and move it to an adequate position afterwards.



Name any number to insert!

Algorithm

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = -∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)

HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```


Runtime

Runtime of this code:

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = -∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)

HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

- In the worst case, lines 4–6 of are called until the root is reached.
- Therefore, the time complexity is $O(h) = O(log(n))$.

Build Heap

Build Heap – *not*

- We could build a heap of size n by inserting n times.
- However, that would lead to a runtime of $O(n \cdot log(n))$.
- We can do better!

Algorithm & Example

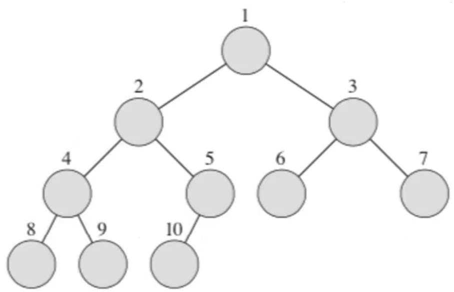
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Why do we start at the middle of the array and walk to the left?

Because Heapify assumes that LEFT(i) and RIGHT(i) satisfies heap properties! So we must work bottom-up!

Example:

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



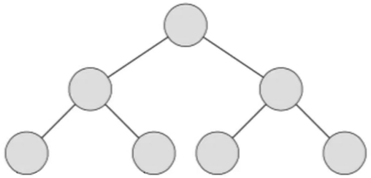
Runtime

- Heapify (runtime $O(log(n))$) is called $\frac{n}{2}$ times, so it still appears as $O(n \cdot log(n))$.
- But we claimed we could to better, $O(n)$!
  What?! Were we wrong??
- No! Not each call has runtime $O(log(n))$!
- Our analysis actually showed $O(log(h_i))$ for the height h_i of the "start node". But the height changes! And there are much more nodes on lower than on higher levels!
- As an intuition, recall that in a perfect binary tree, roughly 50% of all nodes are in the last layer, so half of our calls take constant time!

Runtime, cont'd

Let $T(n)$ be the actual runtime for a tree with n nodes.

$$T(n) \leq \sum_{h=0}^{\lfloor log(n) \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h)$$



Number of nodes at $h = 0$ is $\lceil \frac{7}{2^{0+1}} \rceil = \lceil \frac{7}{2} \rceil = \lceil 3.5 \rceil = 4$ at the bottom, then $\lceil \frac{7}{2^{1+1}} \rceil = 2$ in the middle for $h = 1$, etc.

$\lceil \frac{n}{2^{h+1}} \rceil$ refers to the number of nodes per level (at "current height h").
Important: This height is relative to where we start Heapify, so $h = 0$ is the *leafs*, *not the root*!

Runtime, cont'd

Let $T(n)$ be the actual runtime for a tree with n nodes.

$$T(n) \leq \sum_{h=0}^{\lfloor log(n) \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) \leq \sum_{h=0}^{\lfloor log(n) \rfloor} \lceil \frac{n}{2^{h+1}} \rceil c \cdot h = c \cdot \sum_{h=0}^{\lfloor log(n) \rfloor} \lceil \frac{n}{2 \cdot 2^h} \rceil h$$

$$\leq c \cdot \sum_{h=0}^{\lfloor log(n) \rfloor} \frac{n}{2^h} h \leq c \cdot n \cdot \sum_{h=0}^{\lfloor log(n) \rfloor} \frac{h}{2^h} \leq c \cdot n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} \leq c \cdot n \cdot 2$$

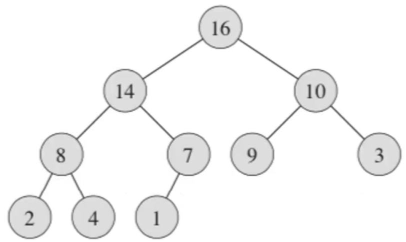
Thus we get $T(n) \in O(n)$.

$\lceil \frac{n}{2^{h+1}} \rceil$ refers to the number of nodes per level (at "current height h").
Important: This height is relative to where we start Heapify, so $h = 0$ is the *leafs*, *not the root*!

Extract Max

Algorithm, Example, and Runtime

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```



The algorithm only needs to traverse a path of the tree once. Hence, the complexity is $O(log(n))$

Applications

Heap Sort

- To sort an array, create a heap, then extract all max values one by one.
- Complexity: $O(n \cdot log(n))$
- In practice, QuickSort runs faster than Heap Sort.
- But the worst-case of heap sort is better!

Priority Queue

A priority queue is a data structure that maintains a set S, where each element is associated with a key. It features the following operations:

- Insert(S, x): Inserts element x into the set S
- Maximum(S): Returns an element of S with the largest key
- ExtractMax(S): Removes and returns an element of S with the largest key
- IncreaseKey (S, x, k): Increase the key of x to k

Common application: Search, e.g., in Automated Planning.
Here we sort by minimum, e.g., for minimal $f(n) = g(n) + h(n)$ in A^*.

Summary

Summary

Today we covered **Heaps**.

Operations considered:

- Heapify $O(log(n))$
- Insertion $O(log(n))$
- Increase-Key $O(log(n))$
- Extract-Max $O(log(n))$
- (Get-Max in max-heaps, Get-Min in min-heaps) $O(1)$
- What about Search? $\rightarrow$ takes $O(n)$!
- What about Deletion? Search, replace by right-most lowest leaf, then heapify! $\rightarrow$ takes $O(n)$

Applications mentioned:

- Sorting arrays $O(n \cdot log(n))$
- Priority Queues