Lectures 8 and 9: Trees and Heap Sort

COMS10007 - Algorithms

Dr. Christian Konrad

19.02.2020

In-class Test

In-class Test:

- When? March 10th, 1pm (during exercise classes)
- Where? Ivy Gate G.01 and Ivy Gate 1.01, two groups

Your timetables have been updated accordingly!

- How long? 50 mins
- What should I expect? All lectures and exercise sheets are relevant (Peak Finding is excluded). Example in-class test uploaded to unit webpage
- You are allowed extra time? Get in touch with me (email)

Sorting Algorithms seen so far

Sorting Algorithms seen so far

- Insertion-Sort: $O(n^2)$ in worst, in place, stable
- Merge-Sort: $O(n \log n)$ in worst case, NOT in place, stable

Heap Sort (best of the two)

- $O(n \log n)$ in worst case, in place, **NOT** stable
- Uses a *heap data structure* (a heap is special tree)

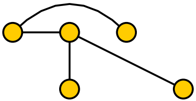
Data Structures

- *Data storage format that allows for efficient access and modification*
- Building block of many efficient algorithms
- For example, an array is a data structure

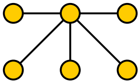
Definition: A *tree* $T = (V, E)$ of size n is a tuple consisting of

$$V = \{v_1, v_2, \dots, v_n\} \text{ and } E = \{e_1, e_2, \dots, e_{n-1}\}$$

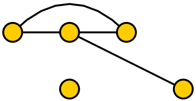
with $|V| = n$ and $|E| = n - 1$ with $e_i = \{v_j, v_k\}$ for some $j \neq k$ s.t. for every pair of vertices v_i, v_j $(i \neq j)$, there is a path from v_i to v_j. V are the nodes/vertices and E are the edges of T.



✓



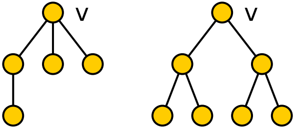
✓



✗

Rooted Trees

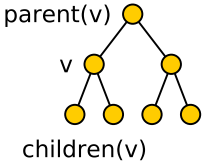
Definition: (rooted tree) A *rooted* tree is a triple $T = (v, V, E)$ such that $T = (V, E)$ is a tree and $v \in V$ is a designed node that we call the *root* of T.



Definition: (leaf, internal node) A *leaf* in a tree is a node with exactly one incident edge. A node that is not a leaf is called an *internal node*.

Children, Parent, and Degree

Further Definitions:



- The *parent* of a node v is the closest node on a path from v to the root. The root does not have a parent.
- The *children* of a node v are v's neighbors except its parent.
- The *height* of a tree is the length of a longest root-to-leaf path.
- The *degree* $\deg(v)$ of a node v is the number of incident edges to v. Since every edge is incident to two vertices we have

$$\sum_{v \in V} \deg(v) = 2 \cdot |E| = 2(n-1) \ .$$

- The *level* of a vertex v is the length of the unique path from the root to v plus 1.

Properties of Trees

Property: Every tree has at least 2 leaves

Proof Let $L \subseteq V$ be the subset of leaves. Suppose that there is at most 1 leaf, i.e., $|L| \leq 1$. Then:

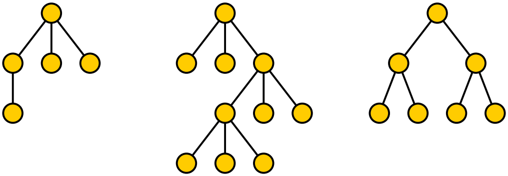
$$\begin{aligned}\sum_{v \in V} \deg(v) &= \sum_{v \in L} \deg(v) + \sum_{v \in V \setminus L} \deg(v) \\ &\geq |L| \cdot 1 + (|V| - |L|) \cdot 2 = 2|V| - |L| \geq 2n - 1 \ ,\end{aligned}$$

a contradiction to the fact that $\sum_{v \in V} \deg(v) = 2(n-1)$ in every tree. $\square$

Binary Trees

Definition: (k-ary tree) A (rooted) tree is *k-ary* if every node has at most k children. If $k = 2$ then the tree is called binary. A k ary tree is

- *full* if every internal node has exactly k children,
- *complete* if all levels except possibily the last is entirely filled (and last level is filled from left to right),
- *perfect* if all levels are entirely filled.



complete 3-ary tree   full 3-ary tree   perfect binary tree

Height of Perfect and Complete k-ary Trees

Height of k-ary Trees

- The number of nodes in a perfect k-ary tree of height $i-1$ is

$$\sum_{j=0}^{i-1} k^j = \frac{k^i - 1}{k-1} \ .$$

- In other words, a perfect k-ary tree on n nodes has height:

$$\begin{aligned} n &= \frac{k^i - 1}{k-1} \\ k^i &= n(k-1)+1 \\ i &= \log_k(n(k-1)+1) = O(\log_k n) \ . \end{aligned}$$

- Similarly, a complete k-ary tree has height $O(\log_k n)$.

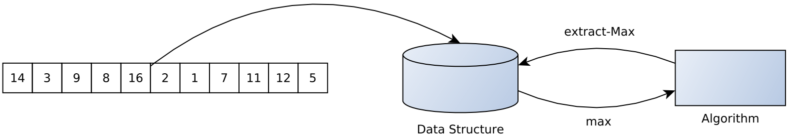
Remark: The runtime of many algorithms that use tree data structures depends on the height of these trees. We are therefore interested in using complete/perfect trees.

Priority Queue:

Data structure that allows the following operations:

- Build(.): Create data structure given a set of data items
- Extract-Max(.): Remove the maximum element from the data structure
- *others...*

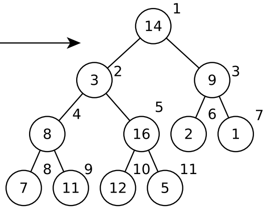
Sorting using a Priority Queue



From Array to Tree

Interpretation of an Array as a Complete Binary Tree

1	2	3	4	5	6	7	8	9	10	11
14	3	9	8	16	2	1	7	11	12	5

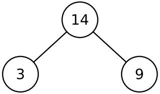


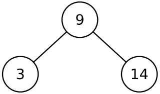
Easy Navigation:

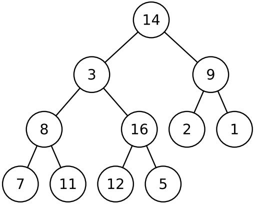
- Parent of i: $\lfloor i/2 \rfloor$
- Left/Right Child of i: $2i$ and $2i+1$

The Heap Property

Key of nodes larger than keys of their children





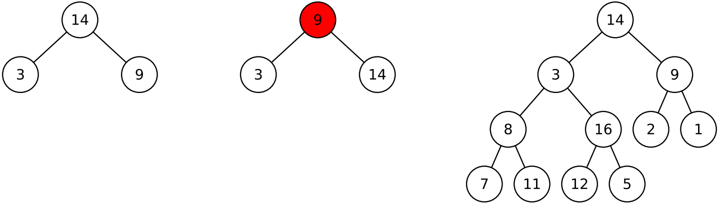


Heap Property $\rightarrow$ Maximum at root
Important for Extract-Max(.)

Heap Property

The Heap Property

Key of nodes larger than keys of their children

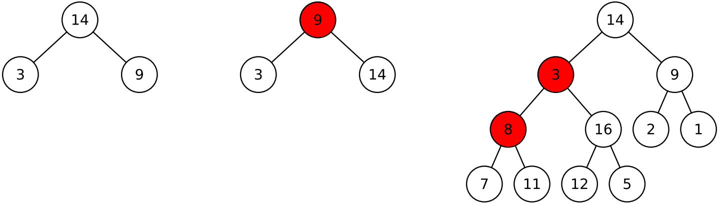


Heap Property $\rightarrow$ Maximum at root
Important for Extract-Max(.)

Heap Property

The Heap Property

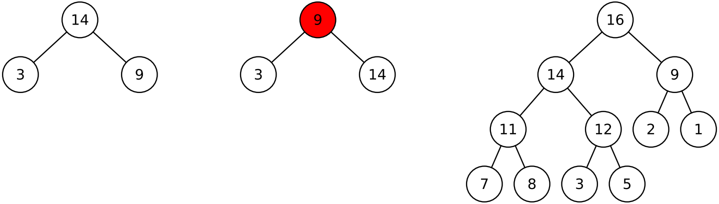
Key of nodes larger than keys of their children



Heap Property → Maximum at root
Important for Extract-Max(.)

The Heap Property

Key of nodes larger than keys of their children

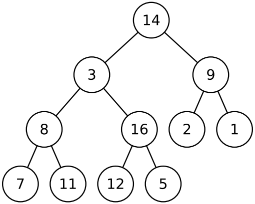


Heap Property → Maximum at root
Important for Extract-Max(.)

The Heapify Operation

Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

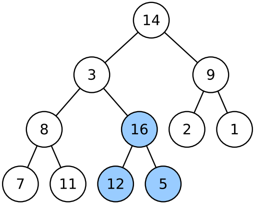
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



The Heapify Operation

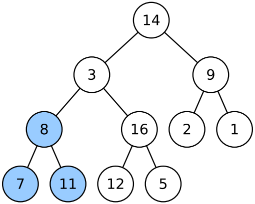
Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

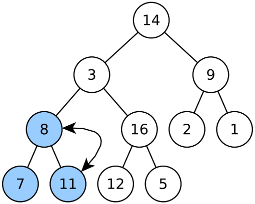
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



The Heapify Operation

Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

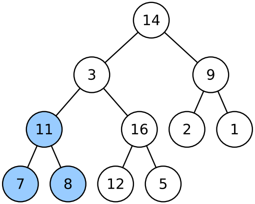
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



The Heapify Operation

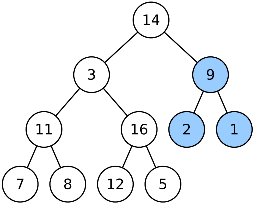
Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

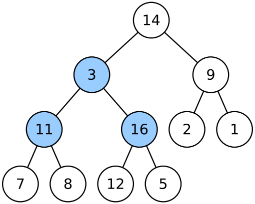
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



The Heapify Operation

Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

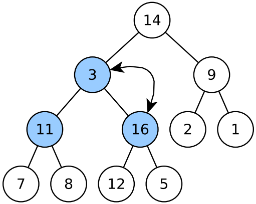
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



The Heapify Operation

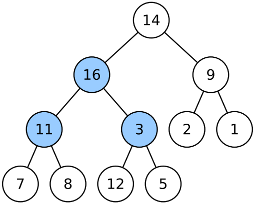
Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**

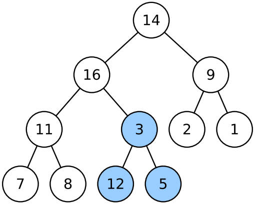


The Heapify Operation

Constructing a Heap: Build(.)

Given a binary tree, transform it into one that fulfills the Heap Property

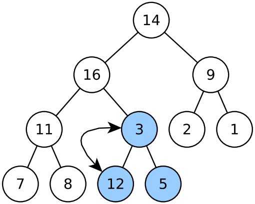
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



The Heapify Operation

Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

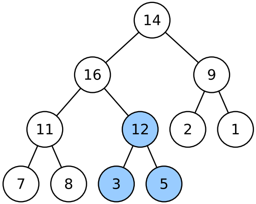
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



The Heapify Operation

Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

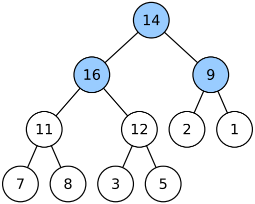
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



The Heapify Operation

Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

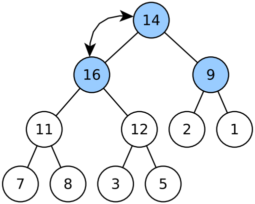
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



The Heapify Operation

Constructing a Heap: Build(.)
Given a binary tree, transform it into one that fulfills the Heap Property

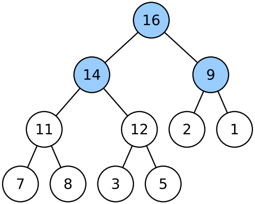
1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



Constructing a Heap: Build(.)

Given a binary tree, transform it into one that fulfills the Heap Property

1. Traverse tree with regards to right-to-left array ordering
2. If node does not fulfill Heap Property: **Heapify()**



Runtime of Heapify()

Heapify()

Let p be the key of a node and let c_1, c_2 be the keys of its children

- Let $c = \max\{c_1, c_2\}$
- If $c > p$ then exchange nodes with keys p and c
- call **Heapify()** at node with key c

Runtime:

- Exchanging nodes requires time $O(1)$
- The number of recursive calls is bounded by the height of the tree, i.e., $O(\log n)$
- Runtime of **Heapify**: $O(\log n)$.

Constructing a Heap: Build(.) Runtime $O(n \log n)$

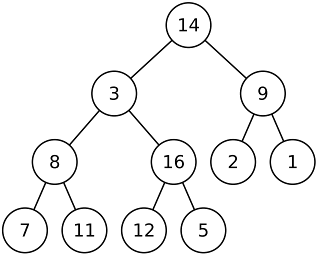
Improved Analysis of Heap Construction

More Precise Analysis of the Heap Construction Step

- Heapify(x): $O(\text{depth of subtree rooted at } x) = O(\log n)$
- **Observe:** Most nodes close to the "bottom"

Analysis:

- Let i be the largest integer such that $n' := 2^i - 1$ and $n' < n$
- There are at most n' internal nodes (candidates for Heapify())
- These nodes are contained in a perfect binary tree
- This tree has height $i - 1$



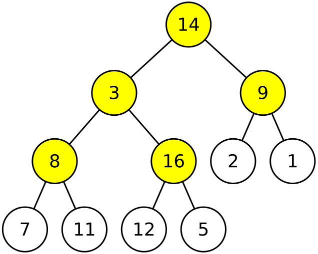
Improved Analysis of Heap Construction

More Precise Analysis of the Heap Construction Step

- Heapify(x): $O(\text{depth of subtree rooted at } x) = O(\log n)$
- **Observe:** Most nodes close to the "bottom"

Analysis:

- Let i be the largest integer such that $n' := 2^i - 1$ and $n' < n$
- There are at most n' internal nodes (candidates for Heapify())
- These nodes are contained in a perfect binary tree
- This tree has height $i - 1$



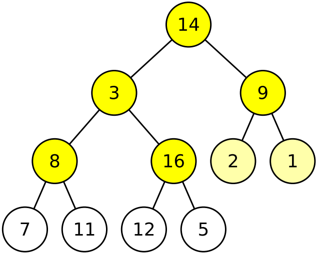
Improved Analysis of Heap Construction

More Precise Analysis of the Heap Construction Step

- Heapify(x): $O(\text{depth of subtree rooted at } x) = O(\log n)$
- **Observe:** Most nodes close to the "bottom"

Analysis:

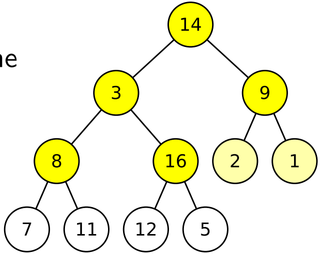
- Let i be the largest integer such that $n' := 2^i - 1$ and $n' < n$
- There are at most n' internal nodes (candidates for Heapify())
- These nodes are contained in a perfect binary tree
- This tree has height $i - 1$



Improved Analysis of Heap Construction

Analysis

We sum over all relevant levels, count the number of nodes per level, and multiply with the depth of their subtrees:



$$\sum_{j=1}^{i} \underbrace{2^{i-j}}_{\text{nodes in level } i-j+1} \cdot \underbrace{j}_{\text{depth of subtree}}$$

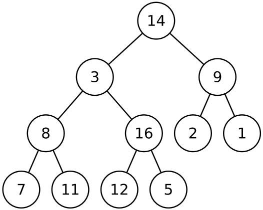
$$\sum_{j=1}^{i} 2^{i-j} \cdot j = 2^i \cdot \sum_{j=1}^{i} \frac{j}{2^j} = O(2^i) = O(n') = O(n) \; .$$

We proved $\sum_{j=1}^{i} \frac{j}{2^j} = O(1)$ in Lecture 4!

The Complete Algorithm

Putting Everything Together

14	3	9	8	16	2	1	7	11	12	5



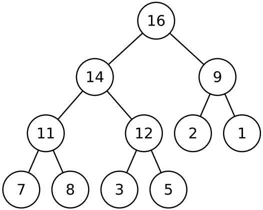
1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

The Complete Algorithm

Putting Everything Together

16	14	9	11	12	2	1	7	8	3	5

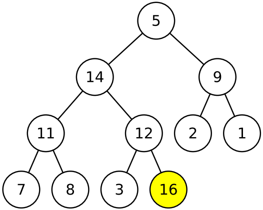
1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)



The Complete Algorithm

Putting Everything Together

5	14	9	11	12	2	1	7	8	3	16

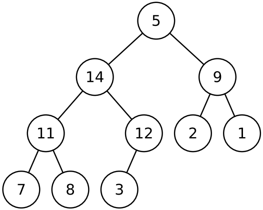


1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

Putting Everything Together

5	14	9	11	12	2	1	7	8	3	16

1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

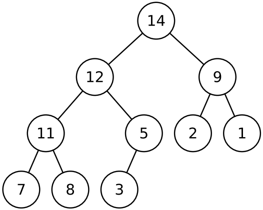


The Complete Algorithm

Putting Everything Together

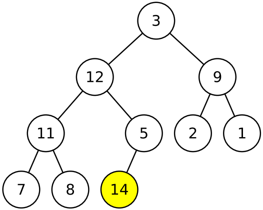
14	12	9	11	5	2	1	7	8	3	16

1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)



Putting Everything Together

3	12	9	11	5	2	1	7	8	14	16



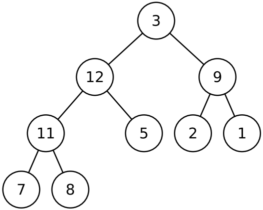
1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

The Complete Algorithm

Putting Everything Together

3	12	9	11	5	2	1	7	8	14	16

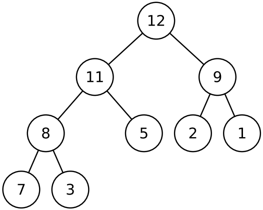
1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)



The Complete Algorithm

Putting Everything Together

12	11	9	8	5	2	1	7	3	14	16

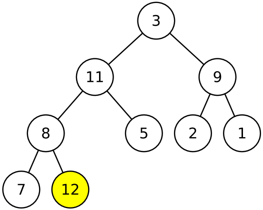


1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

The Complete Algorithm

Putting Everything Together

3	11	9	8	5	2	1	7	12	14	16

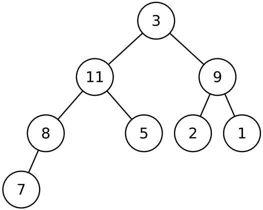


1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

The Complete Algorithm

Putting Everything Together

3	11	9	8	5	2	1	7	12	14	16



1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

The Complete Algorithm

Putting Everything Together

3	11	9	8	5	2	1	7	12	14	16

1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

...

The Complete Algorithm

Putting Everything Together

1	2	3	5	7	8	9	11	12	14	16

1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

Putting Everything Together

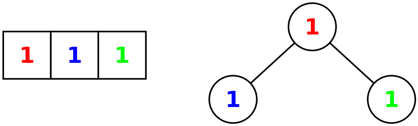
1	2	3	5	7	8	9	11	12	14	16

1. Build-heap() $O(n)$
2. Repeat n times:
   1. Swap root with last element $O(1)$
   2. Decrease size of heap by 1 $O(1)$
   3. Heapify(root) $O(\log n)$

Runtime: $O(n \log n)$

Example:

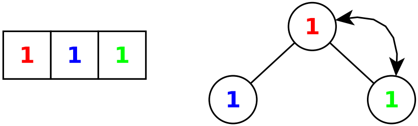
1. Build-heap()
2. Repeat n times:
    1. Swap root with last element
    2. Decrease size of heap by 1
    3. Heapify(root)



Heapsort is Not Stable

Example:

1. Build-heap()
2. Repeat *n* times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

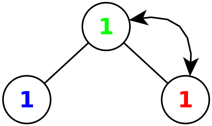


Heapsort is Not Stable

Example:

1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

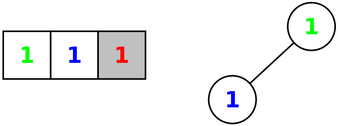
1	1	1



Heapsort is Not Stable

Example:

1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)



1 is moved from left to the right past 1 and 1

Heap-sort not stable

Heapsort is Not Stable

Example:

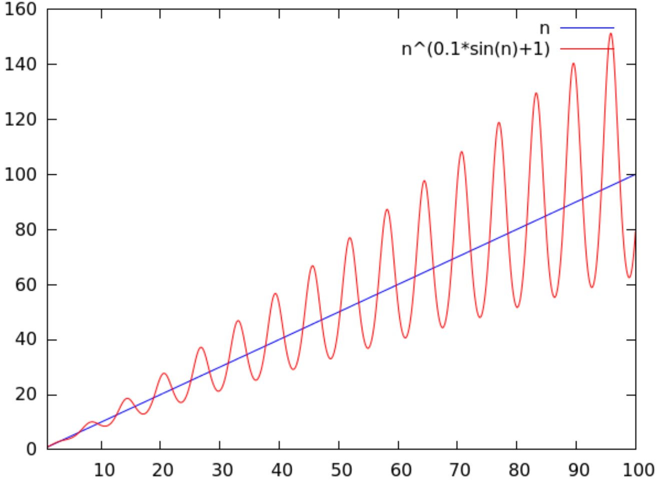
1. Build-heap()
2. Repeat n times:
   1. Swap root with last element
   2. Decrease size of heap by 1
   3. Heapify(root)

1 is moved from left to the right past 1 and 1

Heap-sort not stable

Let f, g be positive functions. Is the following statement true?

Claim. $f(n) \notin O(g(n)) \Rightarrow g(n) \in O(f(n))$. **false!**



Are all Functions Asymptotically Comparable? (2)

$f(n) = n$ and $g(n) = n^{1+0.1\sin(n)}$

Not all Functions are asymptotically comparable!

- Observe that $n^{1+0.1\sin(n)}$ is infinitely often equal to $n^{1.1}$ and infinintely often equal to $n^{0.9}$
- Therefore, neither $f(n) \in O(g(n))$ nor $g(n) \in O(f(n))$

Another Example:

- $f(n) = n$
- $g(n) = n^2$ if n even and $g(n) = \sqrt{n}$ if n odd