

Heap Sort

COMS10017 - (Object-Oriented Programming and) Algorithms

Dr Christian Konrad

Sorting Algorithms seen so far

Sorting Algorithms seen so far

Sorting Algorithms seen so far

- Insertion-Sort: $O(n^2)$ in worst, in place, stable

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

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)

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

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)

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

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*

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

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

Priority Queue:

Data structure that allows the following operations:

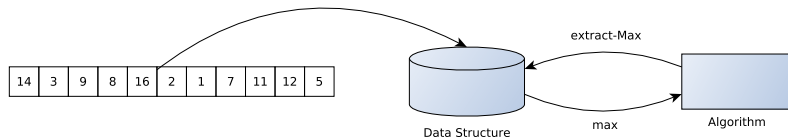
- `Build(.)`: Create data structure given a set of data items
- `Extract-Max(.)`: Remove the maximum element from the data structure and return it
- *others...*

Priority Queue:

Data structure that allows the following operations:

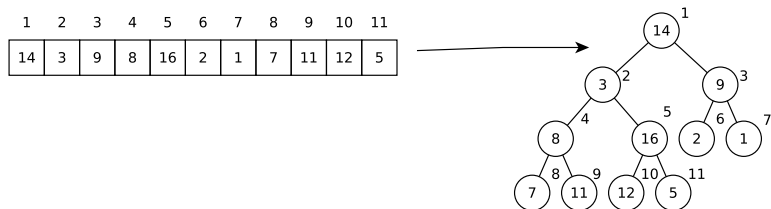
- `Build(.)`: Create data structure given a set of data items
- `Extract-Max(.)`: Remove the maximum element from the data structure and return it
- *others...*

Sorting using a Priority Queue

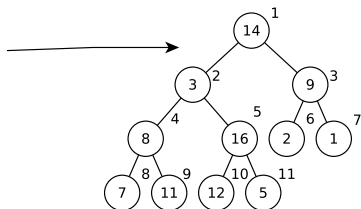
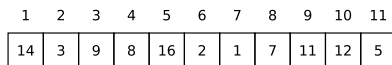


Interpretation of an Array as a Complete Binary Tree

Interpretation of an Array as a Complete Binary Tree

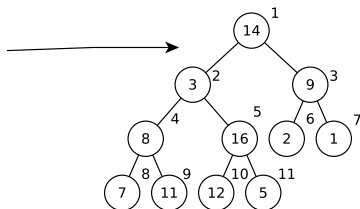
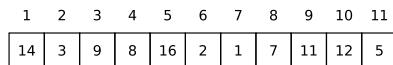


Interpretation of an Array as a Complete Binary Tree



Easy Navigation:

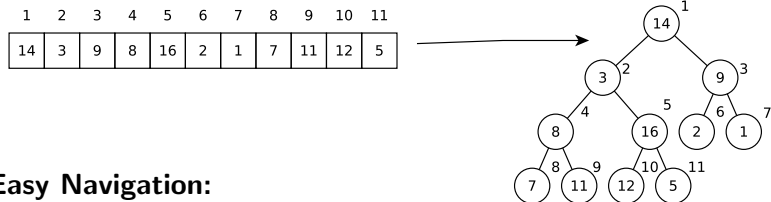
Interpretation of an Array as a Complete Binary Tree



Easy Navigation:

- Parent of i : $\lfloor i/2 \rfloor$

Interpretation of an Array as a Complete Binary Tree

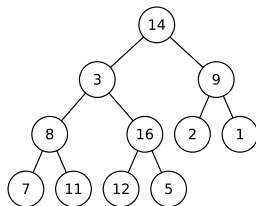
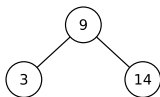
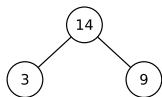


Easy Navigation:

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

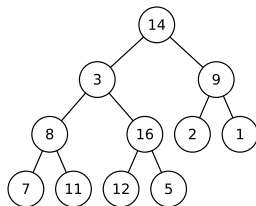
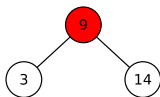
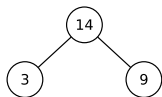
The Heap Property

Key of nodes larger than keys of their children



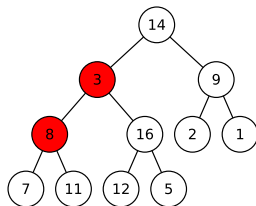
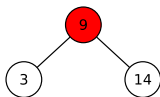
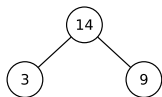
The Heap Property

Key of nodes larger than keys of their children



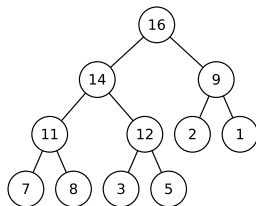
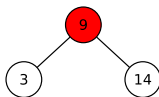
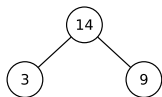
The Heap Property

Key of nodes larger than keys of their children



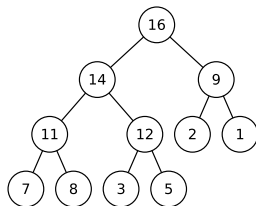
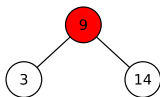
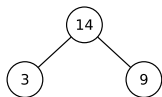
The Heap Property

Key of nodes larger than keys of their children



The Heap Property

Key of nodes larger than keys of their children



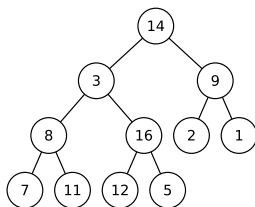
Heap Property \rightarrow Maximum at root
Important for Extract-Max(.)

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()**

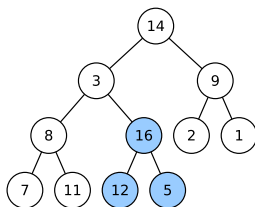


The Heapify Operation

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()**

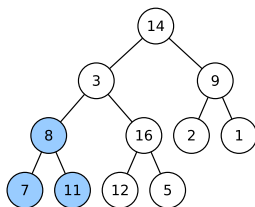


The Heapify Operation

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()**

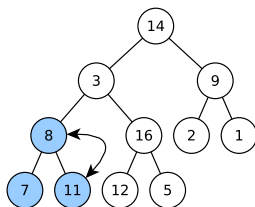


The Heapify Operation

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()**

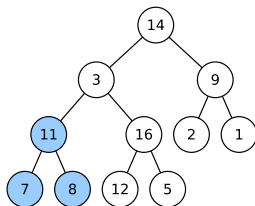


The Heapify Operation

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()**

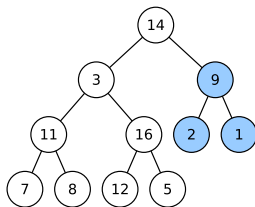


The Heapify Operation

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()**

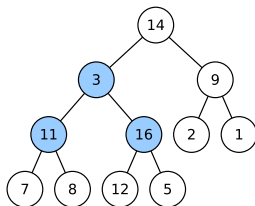


The Heapify Operation

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()**

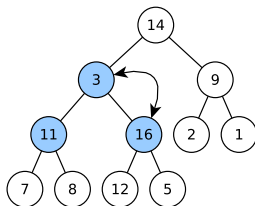


The Heapify Operation

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()**

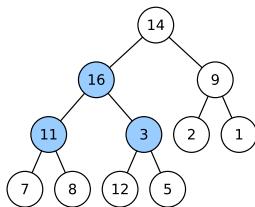


The Heapify Operation

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()**

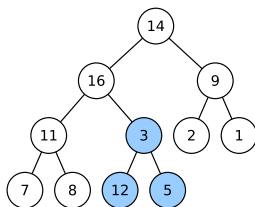


The Heapify Operation

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()**

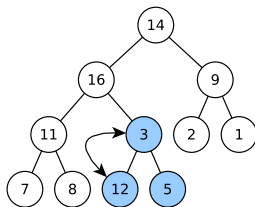


The Heapify Operation

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()**

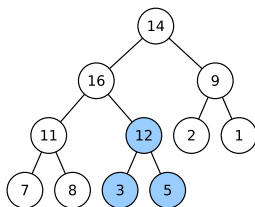


The Heapify Operation

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()**

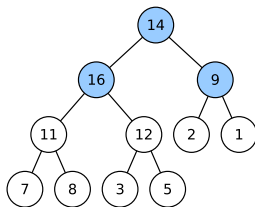


The Heapify Operation

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()**

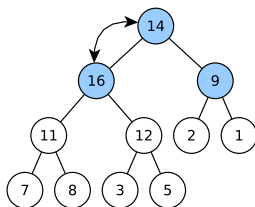


The Heapify Operation

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()**

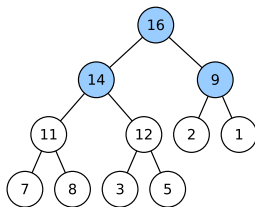


The Heapify Operation

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()**



Runtime of Heapify()

Heapify()

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

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 $c = \max\{c_1, c_2\}$

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

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 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:

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 $O(1)$

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()

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**: $O(\log n)$.

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 $O(n \log n)$

More Precise Analysis of the Heap Construction Step

More Precise Analysis of the Heap Construction Step

- Heapify(x): $O(\text{depth of subtree rooted at } x) = O(\log n)$

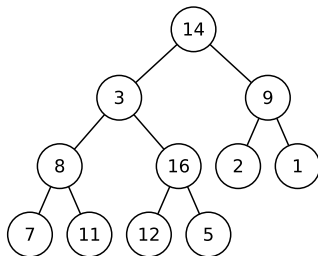
More Precise Analysis of the Heap Construction Step

- Heapify(x): $O(\text{depth of subtree rooted at } x) = O(\log n)$
- **Observe:** Most nodes close to the “bottom” in a complete binary tree

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” in a complete binary tree

Analysis:

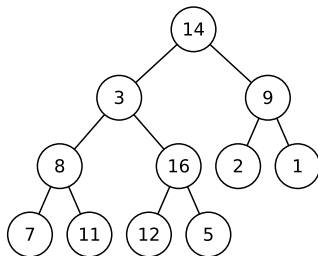


More Precise Analysis of the Heap Construction Step

- Heapify(x): $O(\text{depth of subtree rooted at } x) = O(\log n)$
- **Observe:** Most nodes close to the “bottom” in a complete binary tree

Analysis:

- Let i be the largest integer such that $n' := 2^i - 1$ and $n' < n$

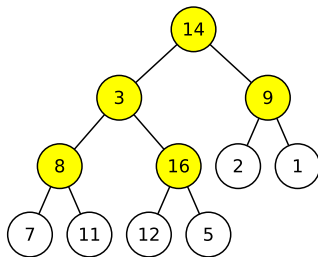


More Precise Analysis of the Heap Construction Step

- Heapify(x): $O(\text{depth of subtree rooted at } x) = O(\log n)$
- **Observe:** Most nodes close to the “bottom” in a complete binary tree

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())

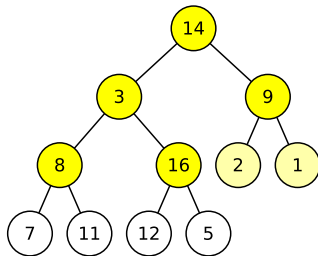


More Precise Analysis of the Heap Construction Step

- Heapify(x): $O(\text{depth of subtree rooted at } x) = O(\log n)$
- **Observe:** Most nodes close to the “bottom” in a complete binary tree

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

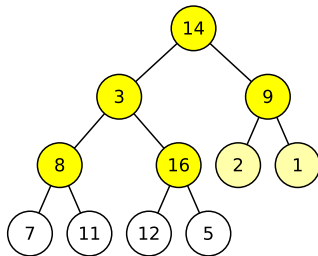


More Precise Analysis of the Heap Construction Step

- Heapify(x): $O(\text{depth of subtree rooted at } x) = O(\log n)$
- **Observe:** Most nodes close to the “bottom” in a complete binary tree

Analysis:

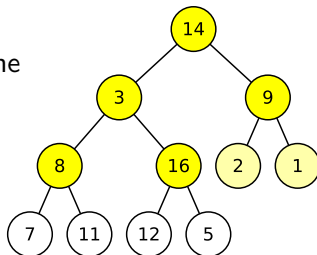
- Let i be the largest integer such that $n' := 2^i - 1$ and $n' < n$
- There are at most n' internal nodes (candidates for Heapify())
- These nodes are contained in a perfect binary tree
- This tree has i levels



Improved Analysis of Heap Construction

Analysis

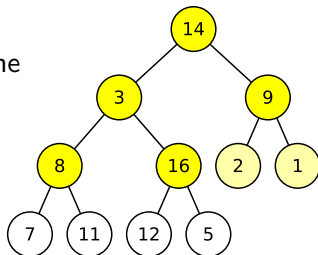
We sum over all relevant levels, count the number of nodes per level, and multiply with the depth of their subtrees:



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:

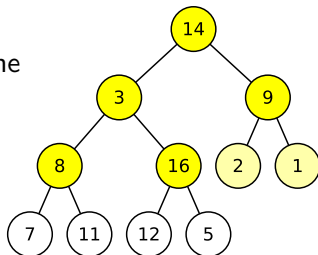


$$\text{Runtime} = \sum_{j=1}^i \# \text{ nodes at level } (i - j + 1) \cdot \text{depth of subtree} \cdot O(1)$$

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:

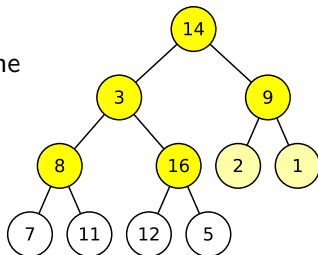


$$\begin{aligned} \text{Runtime} &= \sum_{j=1}^i \# \text{ nodes at level } (i - j + 1) \cdot \text{depth of subtree} \cdot O(1) \\ &= O(1) \sum_{j=1}^i 2^{i-j} \cdot j \end{aligned}$$

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:

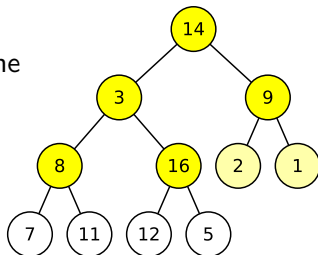


$$\begin{aligned} \text{Runtime} &= \sum_{j=1}^i \# \text{ nodes at level } (i - j + 1) \cdot \text{depth of subtree} \cdot O(1) \\ &= O(1) \sum_{j=1}^i 2^{i-j} \cdot j = O(1) \cdot 2^i \cdot \sum_{j=1}^i \frac{j}{2^j} \end{aligned}$$

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:

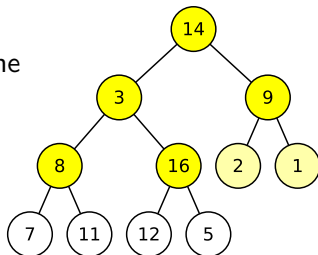


$$\begin{aligned} \text{Runtime} &= \sum_{j=1}^i \# \text{ nodes at level } (i - j + 1) \cdot \text{depth of subtree} \cdot O(1) \\ &= O(1) \sum_{j=1}^i 2^{i-j} \cdot j = O(1) \cdot 2^i \cdot \sum_{j=1}^i \frac{j}{2^j} \\ &= O(2^i) \end{aligned}$$

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:

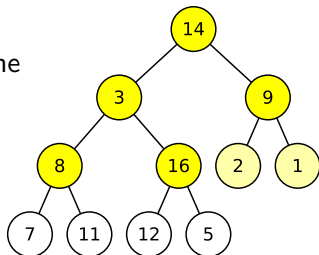


$$\begin{aligned} \text{Runtime} &= \sum_{j=1}^i \# \text{ nodes at level } (i - j + 1) \cdot \text{depth of subtree} \cdot O(1) \\ &= O(1) \sum_{j=1}^i 2^{i-j} \cdot j = O(1) \cdot 2^i \cdot \sum_{j=1}^i \frac{j}{2^j} \\ &= O(2^i) = O(n') \end{aligned}$$

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:

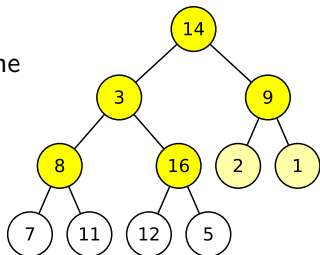


$$\begin{aligned} \text{Runtime} &= \sum_{j=1}^i \# \text{ nodes at level } (i - j + 1) \cdot \text{depth of subtree} \cdot O(1) \\ &= O(1) \sum_{j=1}^i 2^{i-j} \cdot j = O(1) \cdot 2^i \cdot \sum_{j=1}^i \frac{j}{2^j} \\ &= O(2^i) = O(n') = O(n), \end{aligned}$$

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:



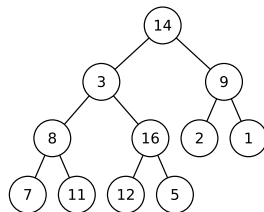
$$\begin{aligned} \text{Runtime} &= \sum_{j=1}^i \# \text{ nodes at level } (i - j + 1) \cdot \text{depth of subtree} \cdot O(1) \\ &= O(1) \sum_{j=1}^i 2^{i-j} \cdot j = O(1) \cdot 2^i \cdot \sum_{j=1}^i \frac{j}{2^j} \\ &= O(2^i) = O(n') = O(n), \end{aligned}$$

using $\sum_{j=1}^i \frac{j}{2^j} = O(1)$ (see trick on video 7).

Putting Everything Together

14	3	9	8	16	2	1	7	11	12	5
----	---	---	---	----	---	---	---	----	----	---

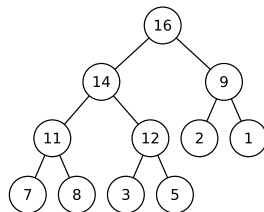
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



Putting Everything Together

16	14	9	11	12	2	1	7	8	3	5
----	----	---	----	----	---	---	---	---	---	---

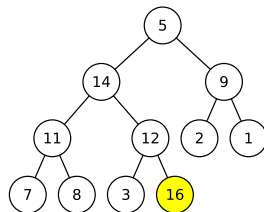
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



Putting Everything Together

5	14	9	11	12	2	1	7	8	3	16
---	----	---	----	----	---	---	---	---	---	----

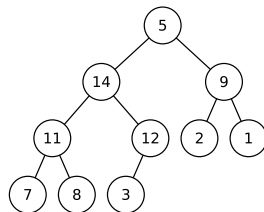
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



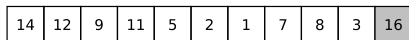
Putting Everything Together

5	14	9	11	12	2	1	7	8	3	16
---	----	---	----	----	---	---	---	---	---	----

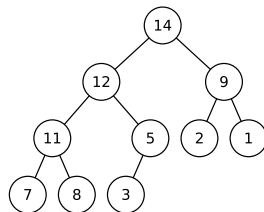
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



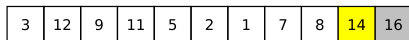
Putting Everything Together



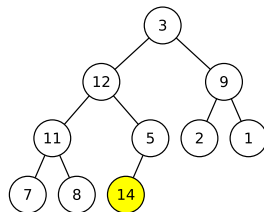
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 **Heapify(root)**



Putting Everything Together



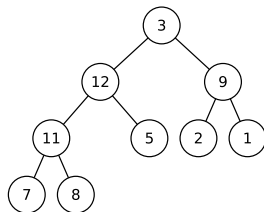
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



Putting Everything Together

3	12	9	11	5	2	1	7	8	14	16
---	----	---	----	---	---	---	---	---	----	----

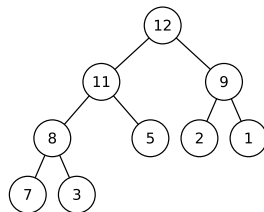
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



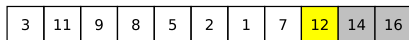
Putting Everything Together

12	11	9	8	5	2	1	7	3	14	16
----	----	---	---	---	---	---	---	---	----	----

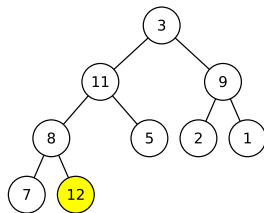
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 **Heapify(root)**



Putting Everything Together



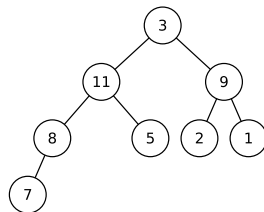
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



Putting Everything Together

3	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)



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)

...

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
---	---	---	---	---	---	---	----	----	----	----

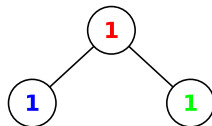
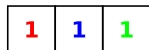
- 1 Build-heap() $O(n)$
- 2 Repeat n times:
 - 1 Swap root with last element $O(1)$
 - 2 Decrease size of heap by 1 $O(1)$
 - 3 Heapify(root) $O(\log n)$

Runtime: $O(n \log n)$

Heapsort is Not Stable

Example:

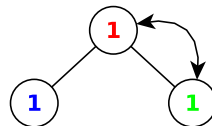
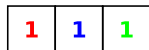
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



Heapsort is Not Stable

Example:

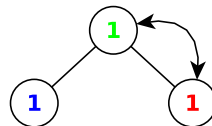
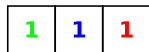
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



Heapsort is Not Stable

Example:

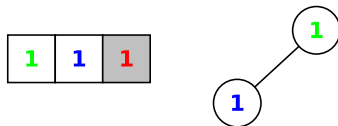
- 1 Build-heap()
- 2 Repeat n times:
 - 1 Swap root with last element
 - 2 Decrease size of heap by 1
 - 3 Heapify(root)



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 is moved from left to the right past 1 and 1

Heap-sort not stable