Heaps and Heapsort
Heaps

• A heap is a binary tree of $T$ that satisfies two properties:
  – Global shape property: it is a complete binary tree
  – Local ordering property: the label in each node is “smaller than or equal to” the label in each of its child nodes
Heaps

- A **heap** is a *binary tree of* $T$ that satisfies two properties:
  - Global shape property: it is a **complete** binary tree
  - Local ordering property: the label in each node is “smaller than or equal to” the label in each of its child nodes

A **complete** binary tree is one in which all levels are “full” except possibly the bottom level, with any nodes on the bottom level as far left as possible.
Heaps

- A heap is a binary tree of $T$ that satisfies two properties:
  - Global shape property: it is a complete binary tree
  - Local ordering property: the label in each node is "smaller than or equal to" the label in each of its child nodes

Also in the picture is (as with BSTs, sorting, etc.) a total preorder that makes this notion precise.
Simplification

• For simplicity in the following illustrations, we use only one kind of example:
  – $T = \text{integer}$
  – The ordering is $\leq$

• Because heaps are used in sorting, where duplicate values may be involved, we allow that multiple nodes in a heap may have the same labels (i.e., we will not assume that the labels are unique)
The Big Picture
This tree’s root label $y$ satisfies $x \leq y$.
Observe: This tree is also a heap; and for each node in this tree with label $z$, we have:

$$x \leq y \leq z.$$
This tree’s root label $y$ satisfies $x \leq y$
Observe: This tree is also a heap; and for each node in this tree with label \( z \), we have:

\[
x \leq y \leq z.
\]
Examples of Heaps

1. 7
   2. 1
     3. 2
       4. 5
   5. 1
     6. 8

1. 1
   2. 1
     3. 8
     4. 2

Non-Examples of Heaps

1

2

8

2

5

8

5

1

1

1

1

2

5

2

2

1

1
Non-Examples of Heaps

Shape property is violated here: not all nodes at the bottom level are as far left as possible.
Non-Examples of Heaps

Ordering property is violated here: value is out of order with that of its right child.
Non-Examples of Heaps

Shape property is violated: two levels are not “full”.
Heapsort

• A heap can be used to represent the values in a SortingMachine, as follows:
  – In `changeToExtractionMode`, arrange all the values into a heap
  – In `removeFirst`, remove the root, and adjust the slightly mutilated heap to make it a heap again
Heapsort

• A heap can be used to sort the values in a SortingMachine, as follows:
  – In `changeToExtractionMode`, arrange all the values into a heap
  – In `removeFirst`, remove the root, and adjust the slightly mutilated heap to make it a heap again

Why should this work?
How `removeFirst` Can Work

- If the root is the only node in the heap, then after removing it, what remains is already a heap; nothing left to do
- If the root is not the only node, then removing it leaves an “opening” that must be filled by moving some other value in the heap into the opening
How `removeFirst` Can Work

- If the root is the only node in the heap, then after removing it, what remains is already a heap; nothing left to do.
- If the root is not the only node, then removing it leaves an “opening” that must be filled by moving some other value in the heap into the opening.

The question is: which one?
Example: A First Attempt
Example: A First Attempt

If we remove the root, leaving this opening ...
Example: A First Attempt

... we might move up the smaller child ...
Example: A First Attempt

... now creating another opening ...

4 2 5

3
Example: A First Attempt

... so, we might move up the smaller child.
Example: A First Attempt

Is the result necessarily a heap?
Example: A Second Attempt
Example: A Second Attempt

This time, let’s maintain the shape property ...
Example: A Second Attempt

... by promoting the last node on the bottom level.
Example: A Second Attempt

Now, we can “sift down” the root into its proper place ...
Example: A Second Attempt

... by swapping it with its smaller child ...
Example: A Second Attempt

... and then “sifting down” the root of that subtree.
Example: A Second Attempt

Is the result necessarily a heap?
Pseudo-Contract

/**
 * Restores a complete binary tree to be a heap
 * if only the root might be out of place.
 * @updates t
 * @requires
 * [t is a complete binary tree] and
 * [both subtrees of the root of t are heaps]
 * @ensures
 * [t is a heap with the same values as #t]
 */

public static void siftDown (BinaryTree<T> t)
{
...}
Building a Heap In the First Place

• Suppose we have $n$ values in a complete binary tree, but they are arranged without regard to the heap ordering
• How can we “heapify” it?
Pseudo-Contract

/**
 * Makes a complete binary tree into a heap.
 * @updates t
 * @requires [t is a complete binary tree]
 * @ensures [t is a heap with the same values as #t]
 */

public static void heapify (BinaryTree<T> t) 
{...}
Hint

• To see how you might implement heapify, compare the contracts of siftDown and heapify

• The only difference: before we can call siftDown to make a heap, both subtrees of the root must already be heaps
  – Once they are heaps, just a call to siftDown will finish the job
Hint

• To see how you might implement `heapify`, compare the contracts of `siftDown` and `heapify`.

• The only difference: before we can call `siftDown` to make a heap, both subtrees of the root must already be heaps.
  – Once they are heaps, just a call to `siftDown` will finish the job.
Example
First, recursively “heapify” the left subtree.
Example

Then, recursively “heapify” the right subtree.
Then “sift down” the root, because now only the root might be out of place.
Embedding a Heap in an **Array**

- While one could represent a heap using a `BinaryTree<T>` (as suggested in the pseudo-contracts above), it is generally not done this way.
- Instead, a heap is usually represented “compactly” using an `Array<T>`.
Interpreting an Array as a Heap
Interpreting an *Array* as a Heap

Because it’s a *complete* binary tree, the nodes can be numbered top-to-bottom, left-to-right.
At what index in the array is the left child of the node at index $i$?
At what index in the **Array** is the **right child** of the node at index \( i \)?
Resources

• Wikipedia: Heapsort

• Wikipedia: Heap (data structure)

• *Big Java*, Section 16.8
  – http://osu.worldcat.org/title/big-java/oclc/754642794

• *Big Java*, Section 16.9
  – http://osu.worldcat.org/title/big-java/oclc/754642794