Heaps.. continued

Binary Heap

- A binary heap (or just heap) is a binary tree that is complete.
  - All levels of the tree are full except possibly for the bottom level which is filled from left to right:

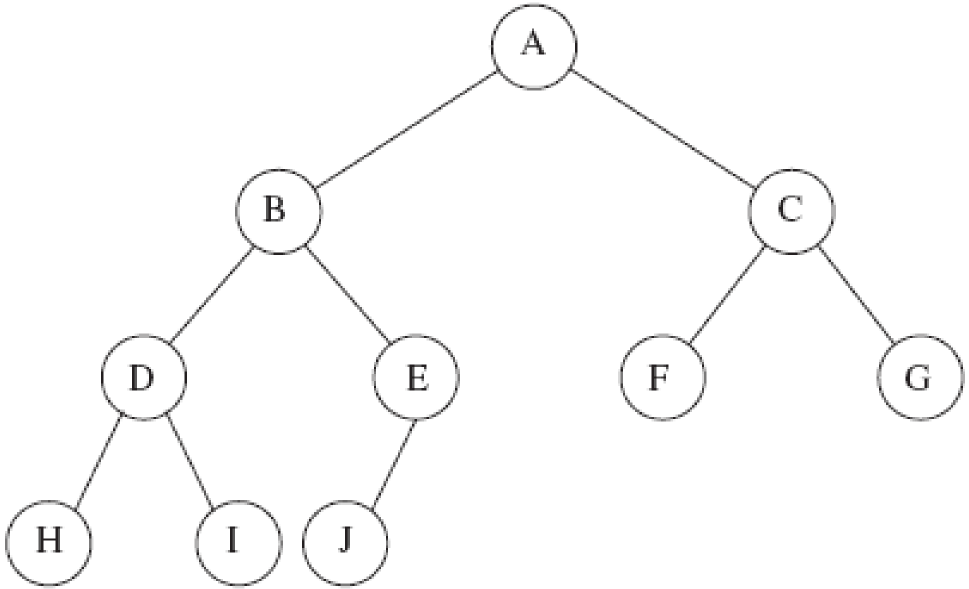
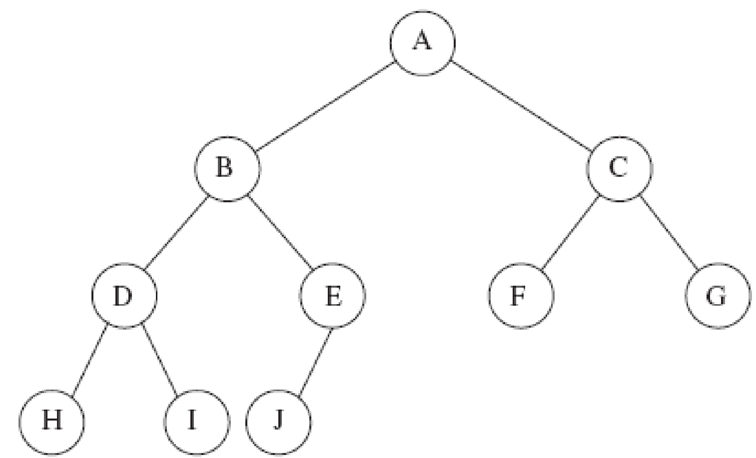


Figure 6.2 A complete binary tree

Binary Heap

- Conceptually, a heap is a binary tree.
- But we can implement it as an array.
- For any element in array position i:
  - Left child is at position $2i$
  - Right child is at position $2i + 1$
  - Parent is at position $\lfloor i / 2 \rfloor$



0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	B	C	D	E	F	G	H	I	J			

Figure 6.3 Array implementation of complete binary tree

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012

Heap-Order Priority

- We want to find the minimum value (highest priority) value quickly.
- Make the minimum value always at the root.
  - Apply this rule also to roots of subtrees.
- Weaker rule than for a binary search tree.
  - Not necessary that values in the left subtree be less than the root value and values in the right subtree be greater than the root value.

Heap-Order Priority

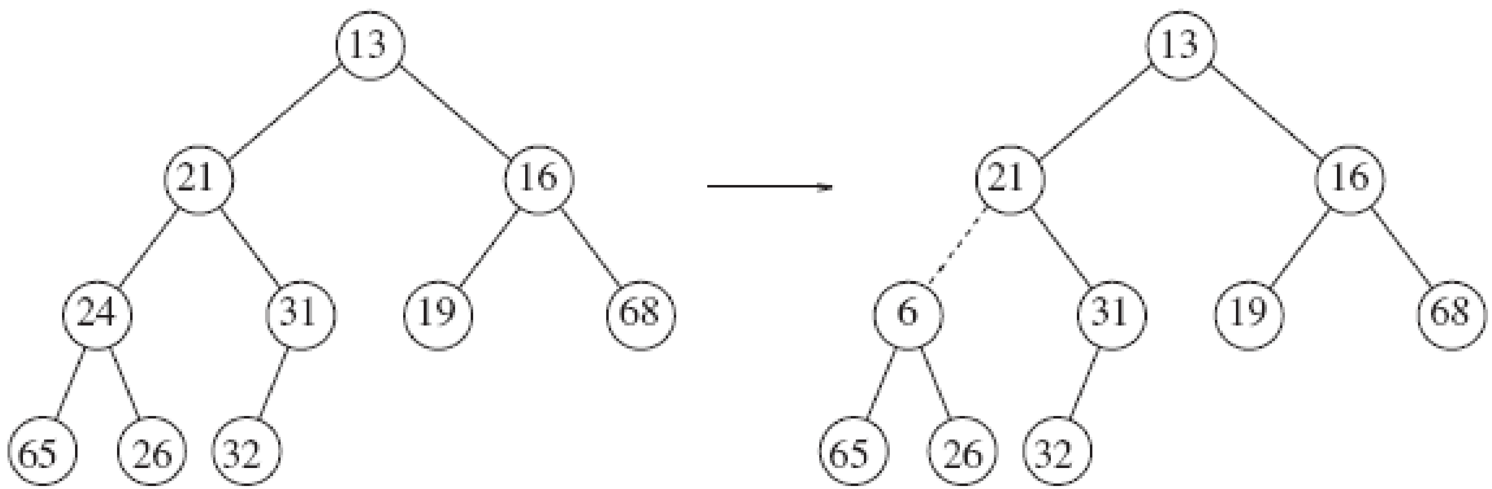


Figure 6.5 Two complete trees (only the left tree is a heap)

Build a Heap from Scratch

- Two ways:
  1. Repeatedly do a heap insertion
     on the list of values to insert.
     - Need to percolate up the hole each time from the bottom of the heap.
  2. Simply stuff all the values into the heap in any order.
     - Since we implement a heap as a complete binary tree, which in turn we implement as a simple array, "stuffing the values" is just appending to the end of the array.
     - Afterward, establish the heap-order priority by repeatedly percolating down the nodes above the bottom row.
     - This way takes $O(N)$ time for N nodes.

Build a Heap from Scratch, *cont'd*

- Construct the heap from an array of values.
  - First stuff the underlying array with values
    in their original order.
  - Then call **buildHeap**() to establish heap-order priority.

```
public BinaryHeap(AnyType[] items)
{
    currentSize = items.length;
    array = (AnyType[]) new Comparable[(currentSize + 2)*11/10
];

    int i = 1;
    for (AnyType item : items) {
        array[i++] = item;
    }

    buildHeap();
}
```

Constructor

Build a Heap from Scratch, *cont'd*

- Call **`percolateDown()`** on nodes above the bottom row.

```
private void buildHeap()
{
     for (int i = currentSize/2; i > 0; i--) {
          percolateDown(i);
     }
}
```

Build a Heap from Scratch, *cont'd*

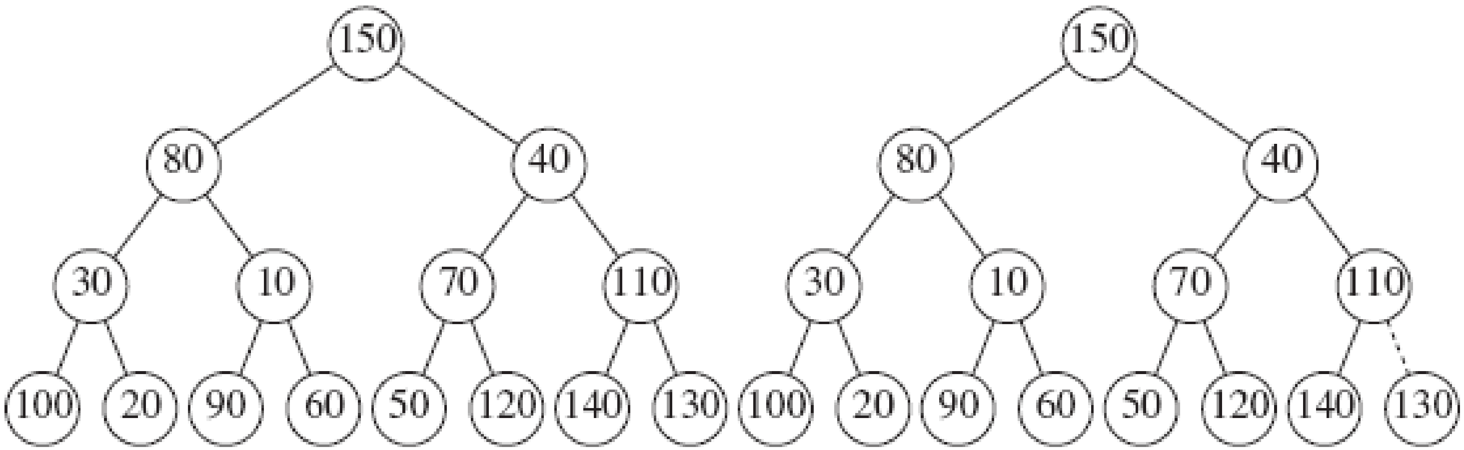


Figure 6.15 Left: initial heap; right: after `percolateDown(7)`

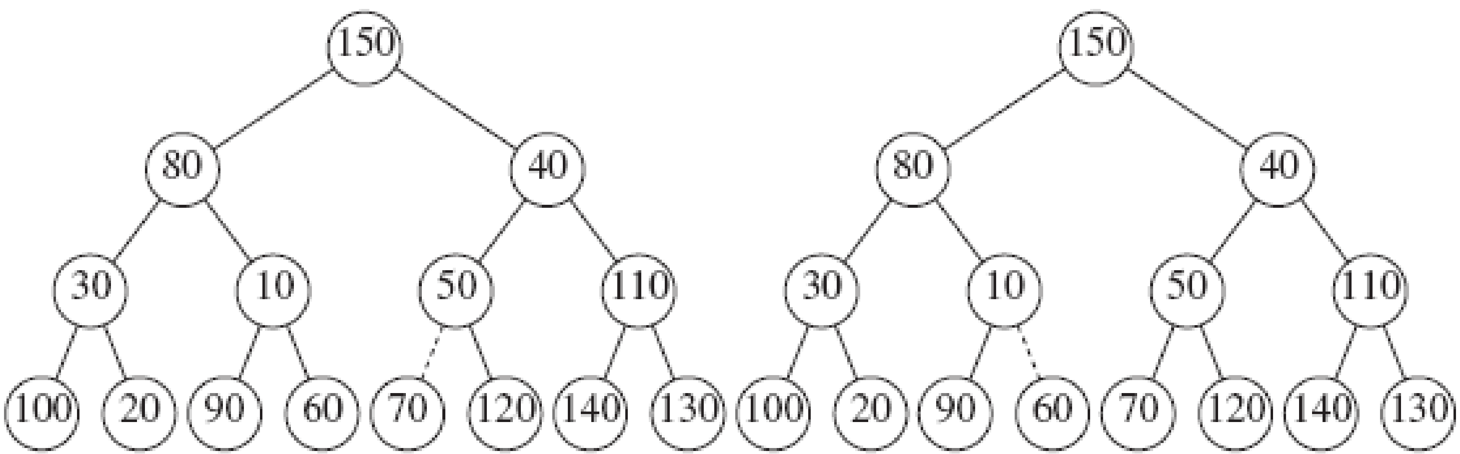


Figure 6.16 Left: after `percolateDown(6)`; right: after `percolateDown(5)`

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012

Build a Heap from Scratch, *cont'd*

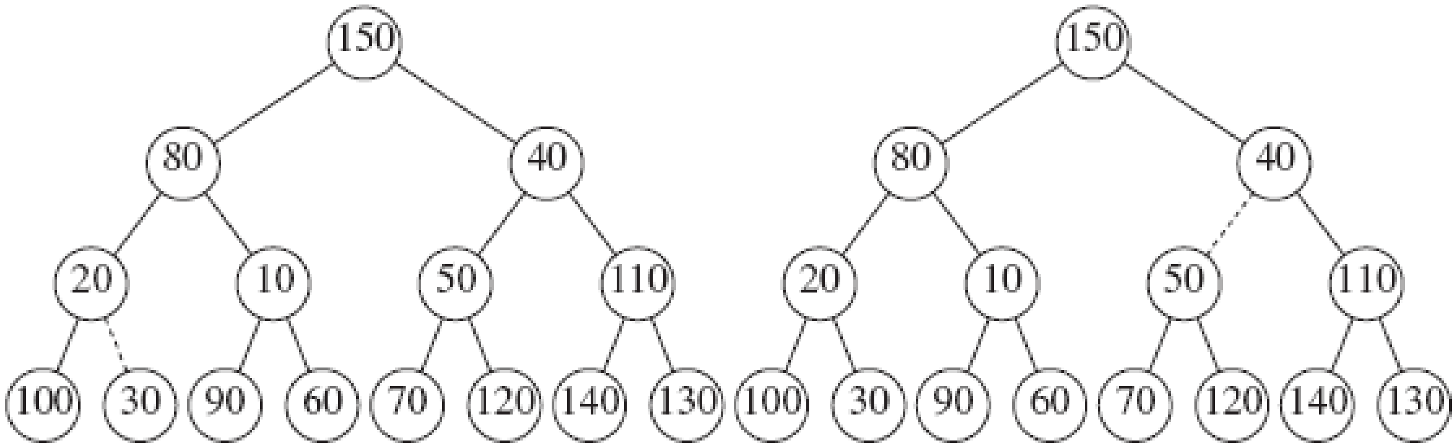


Figure 6.17 Left: after `percolateDown(4)`; right: after `percolateDown(3)`

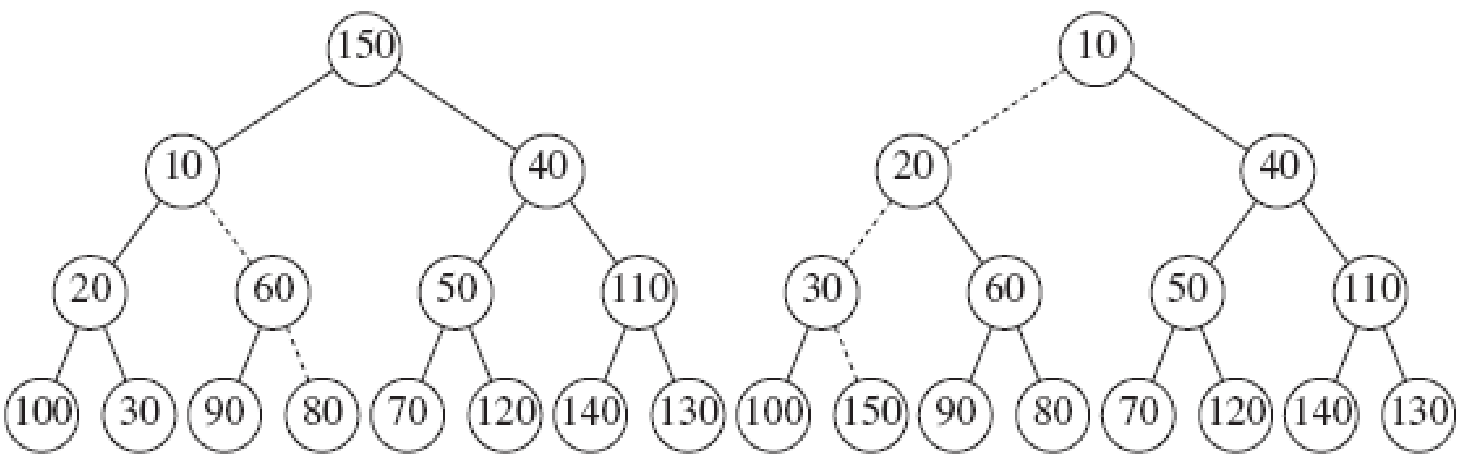


Figure 6.18 Left: after `percolateDown(2)`; right: after `percolateDown(1)`

How Long Does `buildHeap()` Take?

- Each dashed line in the figures corresponds to two comparisons during a call to `percolateDown()`:
  - One to find the smaller child.
  - One to compare the smaller child to the node.
- Therefore, to bound the running time of `buildHeap()`, we must bound the number of dashed lines.
  - Each call to percolate down a node can possibly go all the way down to the bottom of the heap.
    - There could be as many dashed lines from a node as the height of the node.
  - The maximum number of dashed lines is the sum of the heights of all the nodes in the heap.
  - Prove that this sum is $O(N)$.

How Long Does `buildHeap()` Take? *cont'd*

- **Prove:** For a perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h+1)$
  - There is one node (the root) at height h, 2 nodes at height h-1, 4 nodes at height h-2, and in general, 2^i nodes at height h-i.

$$S = \sum_{i=0}^{h} 2^i (h - i)$$

$$= h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}(1) \quad \text{(a)}$$

Multiply both sides by 2:

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots + 2^h(1) \quad \text{(b)}$$

Subtract (b) – (a). Note that $2h - 2(h-1) = 2$, $4(h-1)-4(h-2) = 4$, etc.

$$S = -h + 2 + 4 + 8 + \ldots + 2^{h-1} + 2^h$$

$$= (2^{h+1} - 1) - (h + 1)$$

How Long Does `buildHeap()` Take? *cont'd*

$$S = (2^{h+1} - 1) - (h + 1)$$

- A complete tree is not necessarily a perfect binary tree, but it contains between $N = 2^h$ and 2^{h+1} nodes. Therefore, $S = O(N)$.
- And so `buildHeap()` runs in $O(N)$ time.

Break