# Heap and Heap Sort

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

## Simulation Weblink

https://www.cs.usfca.edu/~galles/visualization/Heap.html

## 1 Motivation

In this lecture, we discuss another fast sorting algorithm, known as *heap sort*, using which we can sort an array in $O(n \log n)$ time. Having said that, our real intention is to study a classical data structure, called *heap*. Heap sort is rather a straightforward byproduct of this data structure.

A heap (or more specifically, a *min heap*) over a sequence of numbers is a data structure that in principle supports the following 3 operations on the sequence:

- $insert(val)$: inserts a new number
- $getMin()$: returns the minimum value
- $deleteMin()$: deletes the minimum value

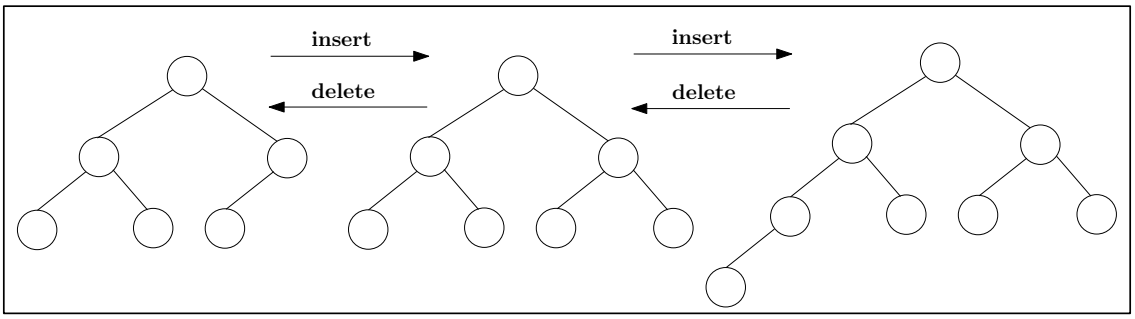
At this point, we see that all of the above operations can be supported by a binary search tree as well. Then, *why study a different data structure?* The reasons are two fold.

First, a heap can support all of the above operations in $O(\log n)$ time (in fact, it can support $getMin()$ in $O(1)$ time), whereas binary search trees need $O(n)$ time in the worst case, as we have seen previously. Balanced binary search trees can be used to perform these operations in exactly the same time bounds, but they are rather complicated. This brings us to the second advantage – heaps can be implemented by using arrays, making them far more practically suitable for applications requiring only insertion and find/delete minimum operations.

## 2 Complete Binary Tree

Recall the definition of a **binary tree** from our discussions on binary search tree. A *complete binary tree* is a binary tree with the following two properties:

- Each level, except possibly the bottom-most level, is full. In other words, if the tree has $k$ levels, with level 1 being root and level $k$ being the bottom-most level, then each node in the levels from 1 through $k-2$ has exactly two children.
- The nodes in the bottom-most level are aligned as far to the left as possible. This means a node cannot have a right child without having a left child. Also, a node cannot have a child if a node to its left does not have a child.




Insertion or deletion in a complete binary tree must maintain the above two properties.

When we insert a new node into the tree, we always insert it into the next empty slot, whereas deletion is always on the last filled slot. Thus, we will insert left-to-right in a level, whereas deletion is right-to-left. Also, a level must be completely filled before inserting into the level below, whereas a level must be completely empty before deletion in the level above.

See figure above for illustration.

# 3 Min Heap

A **min heap** (or simply, a heap) is a complete binary tree, where each node is assigned a value and the tree satisfies the following property.

**Min-Heap Property:** The value of a node is smaller than or equal to the value of either of its children. In other words, the value of a node is larger than or equal to the value of its parent.

## 3.1 Inserting *val*

Recall that insertion is always at the next empty slot. Thus, to insert *val*, we first create a new node with *val* inside it and place it at the next empty slot. Now, we simply compare *val* with its parent, and swap if the value of parent is larger. Keep repeating the swapping process until we have reached a node, where the value of the parent is smaller (or same), or we have reached the root node. In other words, keep swapping with parent as long as the heap property is not satisfied. See figure on next page for illustration.

## 3.2 Finding Minimum

Finding minimum is trivial – minimum is always at the root node, which is a direct consequence of the heap property.

## 3.3 Deleting Minimum

Recall that deletion in a complete binary tree is always at the last filled spot, i.e., the last node in the tree. However, our objective is to delete the minimum value in a heap, which is located at the root. So, the first step is a swap of the values stored in the root and in the last node. Now, remove the last node from the heap.

At this point, the heap property may have been violated. We store it using the following steps:

1. Let $currNode = root$.

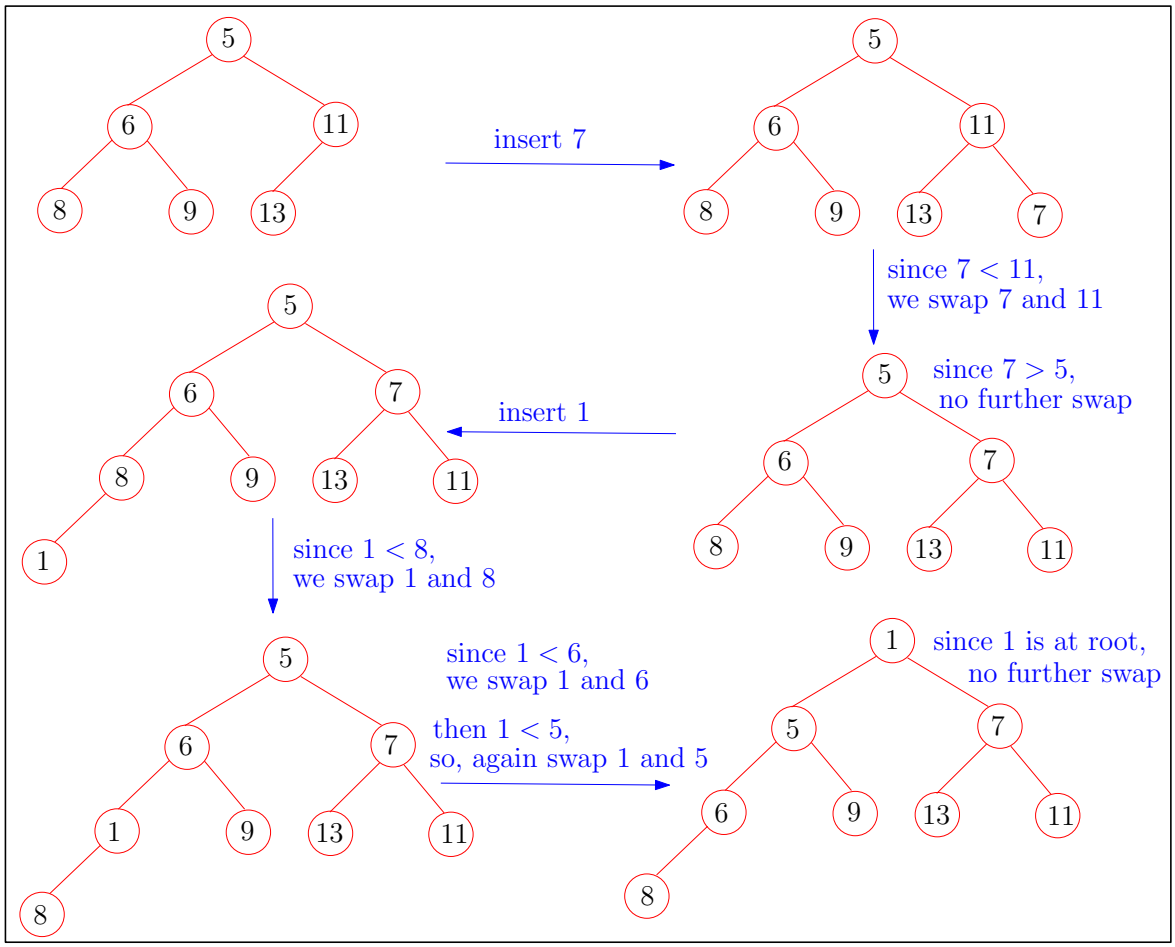



Figure 1: Insertion in a Heap

2. First check if $currNode$ has a left child. If there is no left child, stop the process. Otherwise, check if $currNode$ has a right child. If there is no right child, then assign left child as $minChild$. Otherwise, $minChild$ is the child of $currNode$ which has the minimum value.

3. If the value of $minChild$ is larger than the value of $currNode$, then stop the process. Otherwise, first swap the value of $currNode$ and $minChild$. Then, update $currNode$ to $minChild$, and repeat the process starting from Step 2.

See next page for illustration.

### 3.4 Analysis

It is easy to see that $getMin$ takes $O(1)$ time, since we are only inspecting the root. The other two operations ($insert$ and $deleteMin$) require $O(\log n)$ time, where $n$ is the number of nodes in the heap. This is because the height of the heap is at most $(1 + \log n)$.

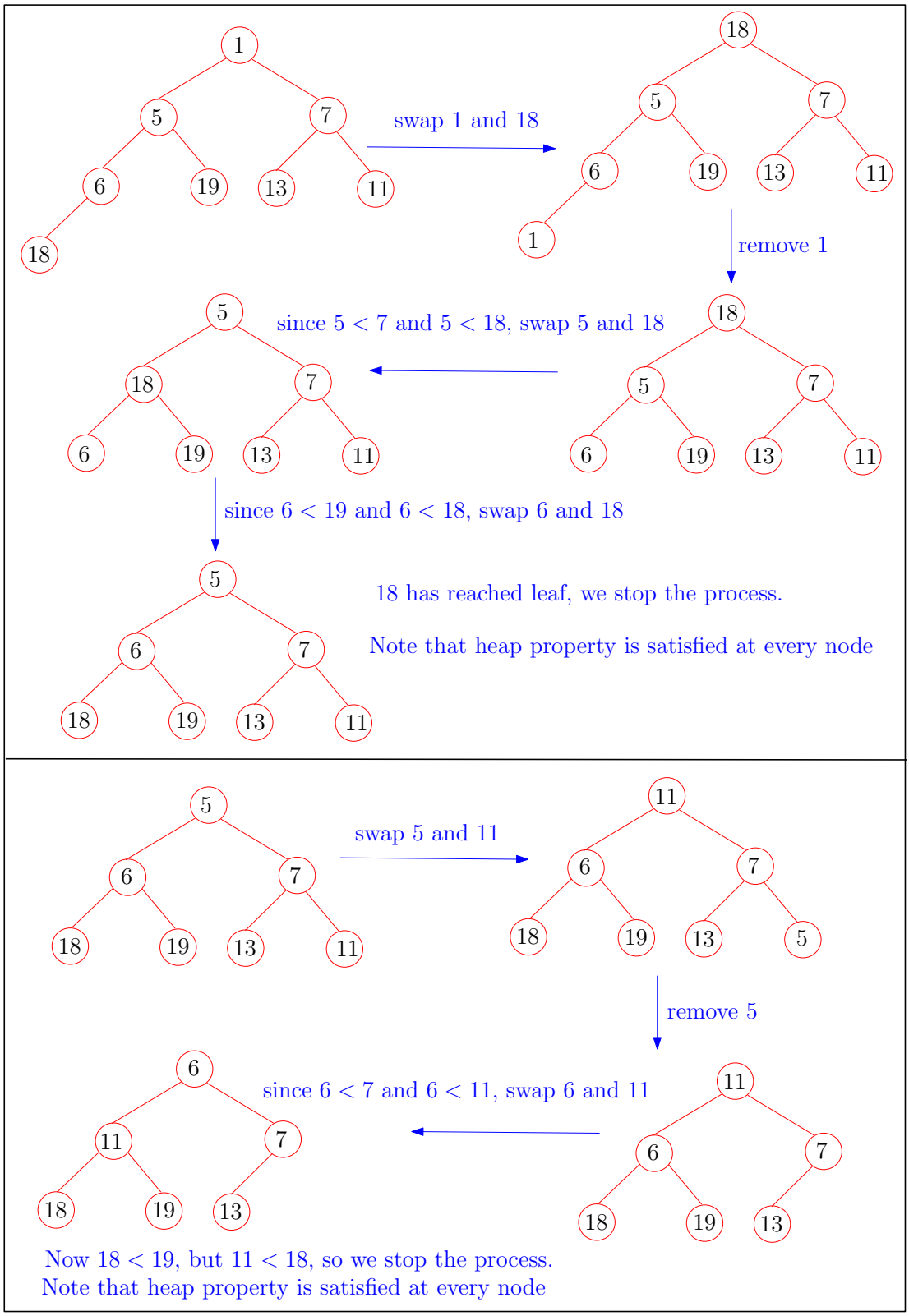



Figure 2: Deletion in a Heap

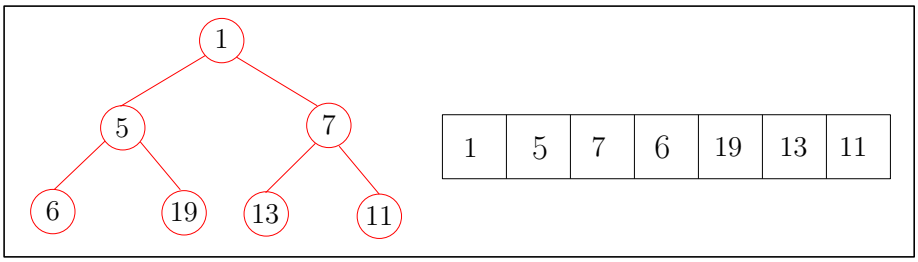



Figure 3: Implementation of a Heap

## 4 Implementation

As claimed earlier, a heap (in fact any complete binary tree) can be simulated using an array, where the array is filled up as follows. Starting from the root node, make a left-to-right traversal at each level and fill up the array before moving to the next level. Thus, for the heap above, the array is $[1, 5, 7, 6, 19, 13, 11]$, as shown. Let $currSize$ be the current number of nodes on the heap. Thus, $currSize = 7$ for this heap. The following are some crucial observations:

- The last value on the heap is given by $heapArray[currSize - 1]$. A new number is inserted at index $currSize$ of $heapArray$. The minimum is at index 0 of $heapArray$.
- If a node does not have a right child, then its left child (if exists) is the last node on the heap.
- The left child of the node corresponding to the $i$th index in the heap-array is given by $(2*i+1)$, whereas the right child is given by $(2 * i + 2)$.

  Thus the left child of 5 (which is at index 1) is at index $2 * 1 + 1 = 3$ and $heapArray[3] = 6$, which is what we need. Similarly, the right child of 5 is at index $2 * 1 + 2 = 4$ and $heapArray[4] = 19$, as desired.
- The parent of the node corresponding to the $i$th index in the heap-array is given by $(i - 1)/2$.

  Thus the parent of 6 (which is at index 3) is at index $(3 - 1)/2 = 1$, and $heapArray[1] = 5$, as desired. Likewise, the parent of 11 (which is at index 6) is at index $(6 - 1)/2 = 2$, and $heapArray[2] = 7$, as desired.

Using this information, we can now implement insertion and deletion in heap as illustrated the following pseudo-codes.

**Inserting a new value into a heap**

- Let $currentIndex$ = size of the heap
- Let $parentIndex = (currentIndex - 1)/2$
- Add the new value at the end of the heap array
- While ($currentIndex > 0$ and (value at $parentIndex >$ value at $currentIndex$)), do:
  - swap the contents of $heapArray$ at the indexes $parentIndex$ and $currentIndex$
  - set $currentIndex = parentIndex$
  - update $parentIndex = (currentIndex - 1)/2$

Getting the minimum value in a heap

**return** the value at index 0 of the heap array

Deleting the minimum from the heap

- Update index 0 of heapArray with the last value of the heap
- Remove the last number in the heap
- Let $currentIndex = 0$, $leftIndex = 1$, and $rightIndex = 2$
- While ($leftIndex <$ size of heap), i.e., as long as current node has a left child, do:
  - let $currentKey$ be the value at $currentIndex$
  - let $leftKey$ be the value at $leftIndex$
  - if ($rightIndex <$ size of heap), i.e., if current node has a right child, do:
    - let $rightKey$ be the value at $rightIndex$
    - if ($leftKey < currentKey$ and $leftKey < rightKey$), then left child is the smallest; so:
      - swap the values of $leftIndex$ and $currentIndex$
      - set $currentIndex = leftIndex$
    - else if ($rightKey < currentKey$), then right child is the smallest; so:
      - swap the values of $rightIndex$ and $currentIndex$
      - set $currentIndex = rightIndex$
    - else current node is the smallest; so:
      - break;
  - else if ($leftKey < currentKey$) then right child does not exist, left child is the last node, and left child is the smaller; so:
    - swap the values of $leftIndex$ and $currentIndex$
    - break
  - else $currentIndex$ is the smallest; so:
    - break
  - set $leftIndex = 2 * currentIndex + 1$
  - set $rightIndex = leftIndex + 1$

## 5 Heap Sort

To heap sort an array $A[0, n-1]$, simply insert each number of the array into the heap. Once all the numbers have been inserted, perform $n$ successive $getMin$ and $deleteMin$ operations each to retrieve the array in sorted order. Since each operation of the heap takes $O(\log n)$ time, total time to sort is $O(n \log n)$.