BINARY HEAP IMPLEMENTATION

Trees

- Code is all written out for you in the Jupyter Notebook!
- Read through Wikipedia article first, before viewing this lecture!

Trees

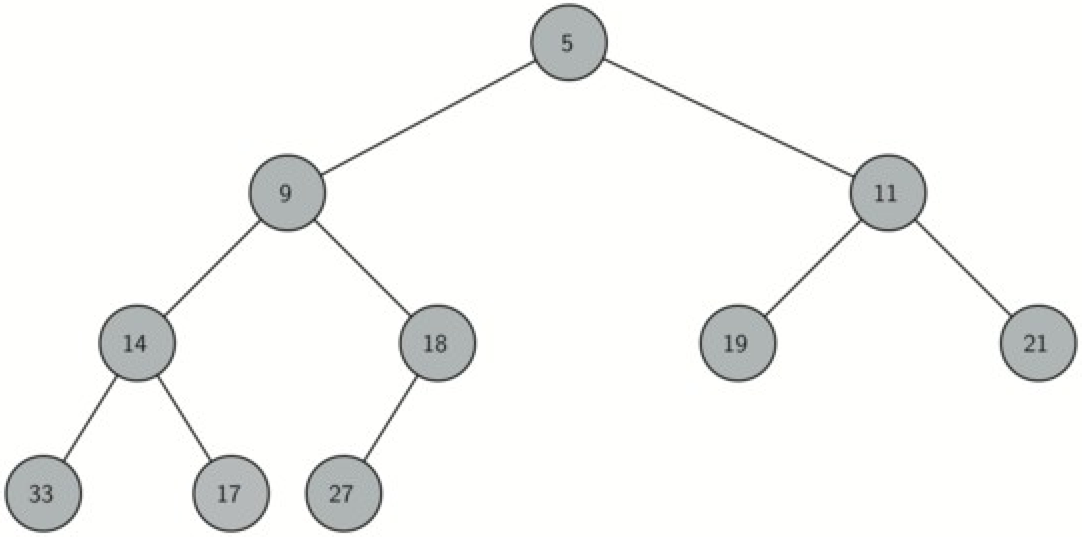
- In order to make our heap work efficiently, we will take advantage of the logarithmic nature of the binary tree to represent our heap.
- In order to guarantee logarithmic performance, we must keep our tree balanced.

Trees

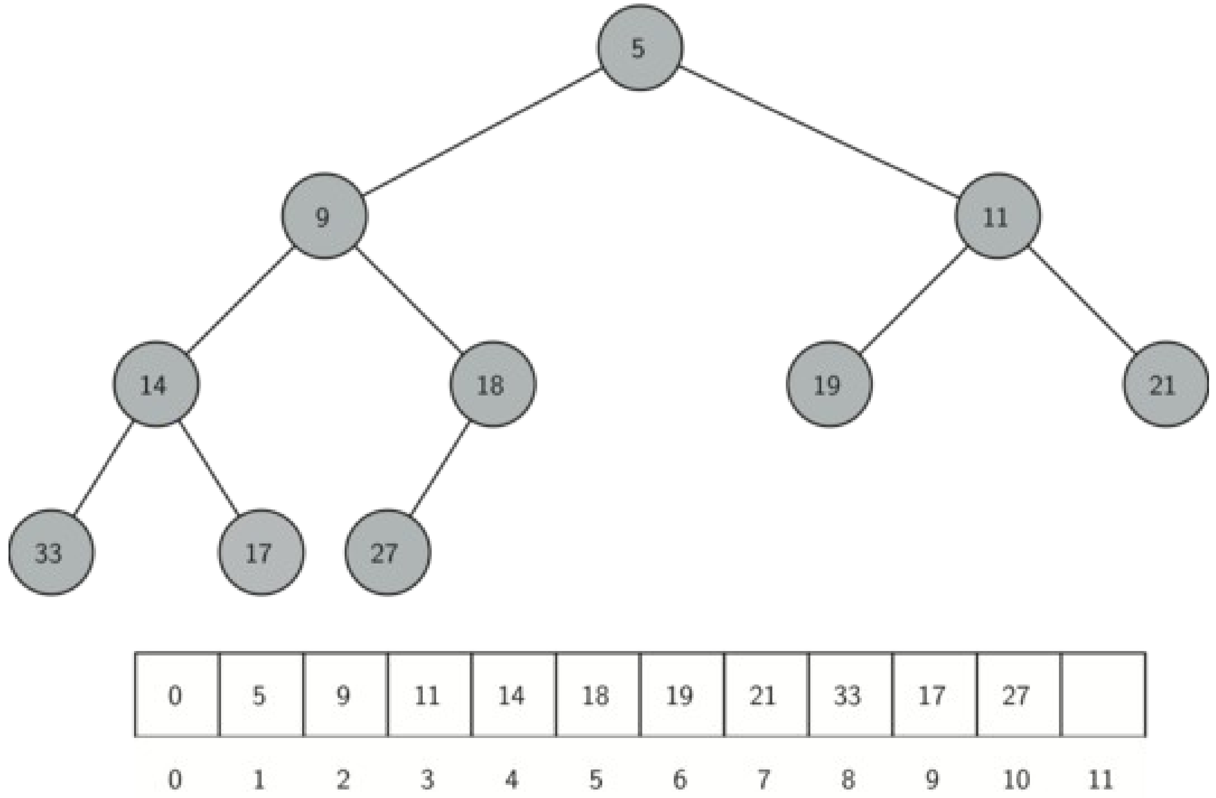
- A balanced binary tree has roughly the same number of nodes in the left and right subtrees of the root.
- In our heap implementation we keep the tree balanced by creating a **complete binary tree**.
- A complete binary tree is a tree in which each level has all of its nodes.

Trees

- Example Binary Tree



List Representation of Trees



Binary Heaps

- Reminder, as we continue on, use the Jupyter Notebook to reference the complete code

Binary Heaps

- Start off with our list representation code

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0
```

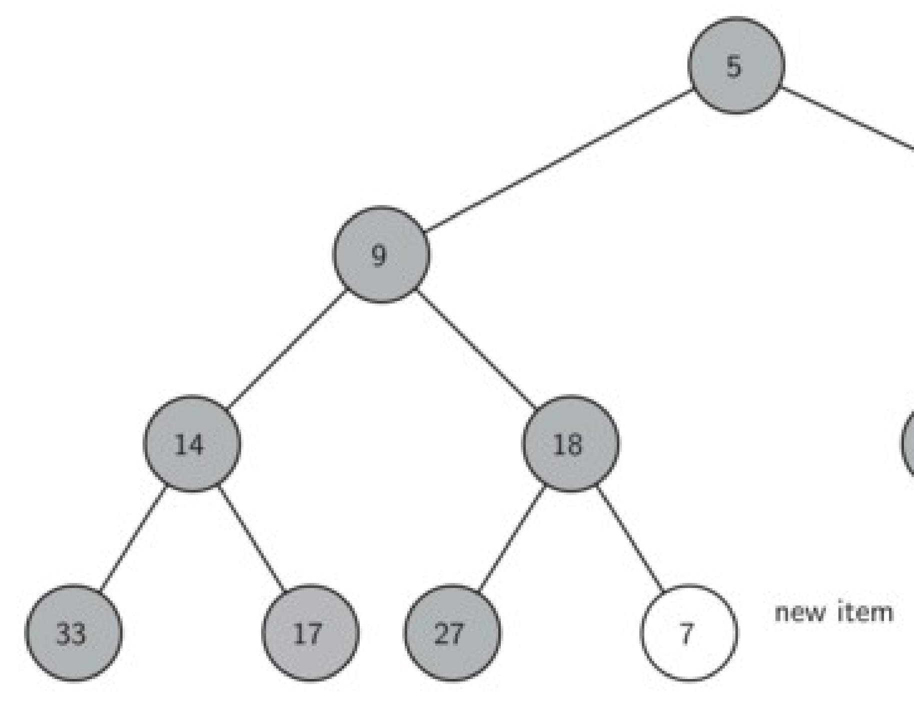
Binary Heaps

- The next method we will implement is insert. The easiest, and most efficient, way to add an item to a list is to simply append the item to the end of the list.
- The good news about appending is that it guarantees that we will maintain the complete tree property.
- The bad news about appending is that we will very likely violate the heap structure property.

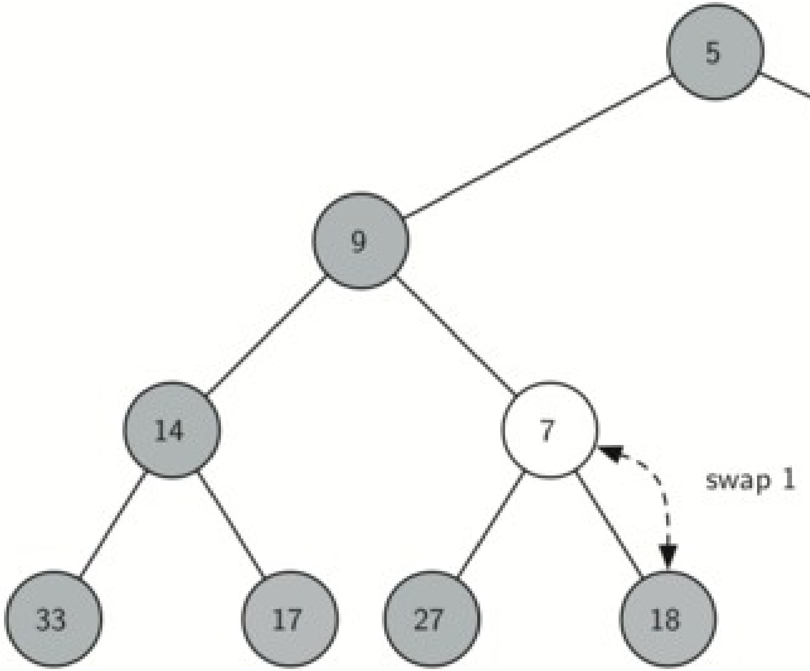
Binary Heaps

- However, it is possible to write a method that will allow us to regain the heap structure property by comparing the newly added item with its parent.
- If the newly added item is less than its parent, then we can swap the item with its parent.
- Let’s see the series of swaps needed to percolate the newly added item up to its proper position in the tree!

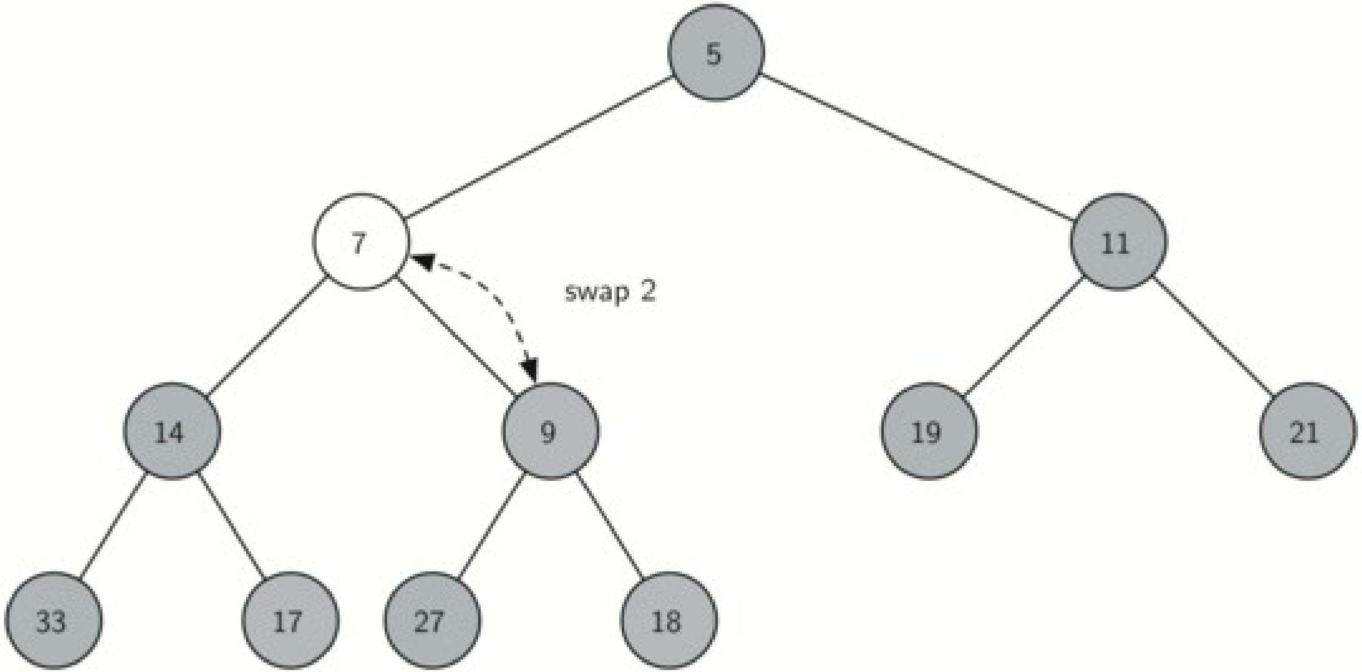
Binary Heaps



Binary Heaps



Binary Heaps



Binary Heaps

- Notice that when we percolate an item up, we are restoring the heap property between the newly added item and the parent.
- We are also preserving the heap property for any siblings.
- Of course, if the newly added item is very small, we may still need to swap it up another level.
- In fact, we may need to keep swapping until we get to the top of the tree.

Binary Heaps

- Methods for insertion

```python
def percUp(self,i):
    while i // 2 > 0:
      if self.heapList[i] < self.heapList[i // 2]:
         tmp = self.heapList[i // 2]
         self.heapList[i // 2] = self.heapList[i]
         self.heapList[i] = tmp
      i = i // 2
```

Binary Heaps

- Methods for insertion

```
def insert(self,k):
    self.heapList.append(k)
    self.currentSize = self.currentSize + 1
    self.percUp(self.currentSize)
```

Binary Heaps

- With the insert method properly defined, we can now look at the **delMin** method.
- Since the heap property requires that the root of the tree be the smallest item in the tree, finding the minimum item is easy.
- The hard part of **delMin** is restoring full compliance with the heap structure and heap order properties after the root has been removed.

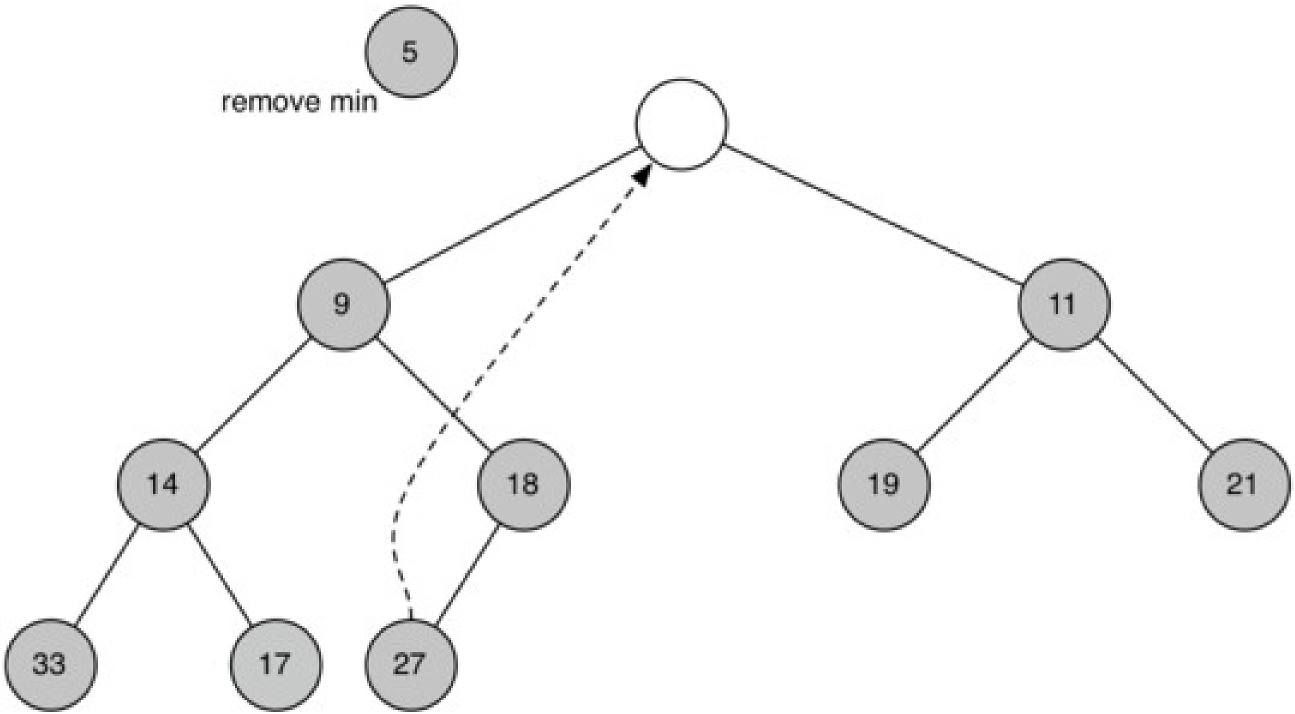
Binary Heaps

- We can restore our heap in two steps.
- First, we will restore the root item by taking the last item in the list and moving it to the root position.
- Moving the last item maintains our heap structure property.
- However, we have probably destroyed the heap order property of our binary heap.
- Second, we will restore the heap order property by pushing the new root node down the tree to its proper position.

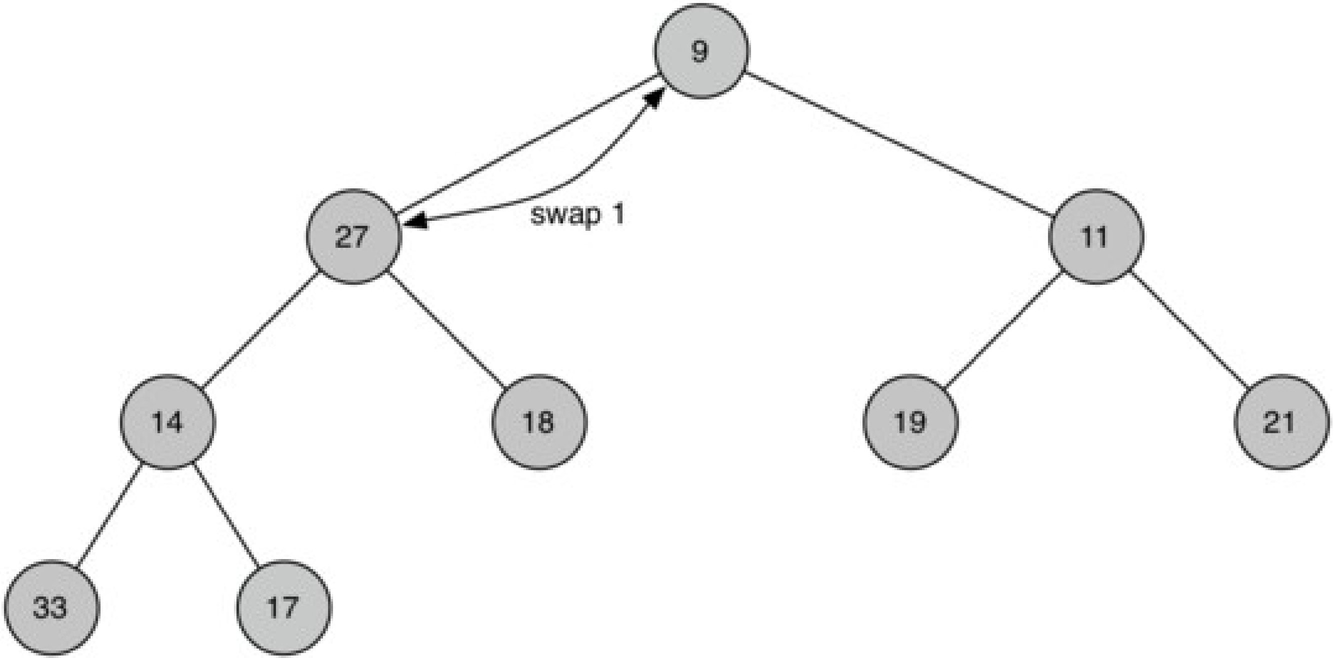
Binary Heaps

- Series of swaps needed to move the new root node to its proper position in the heap.

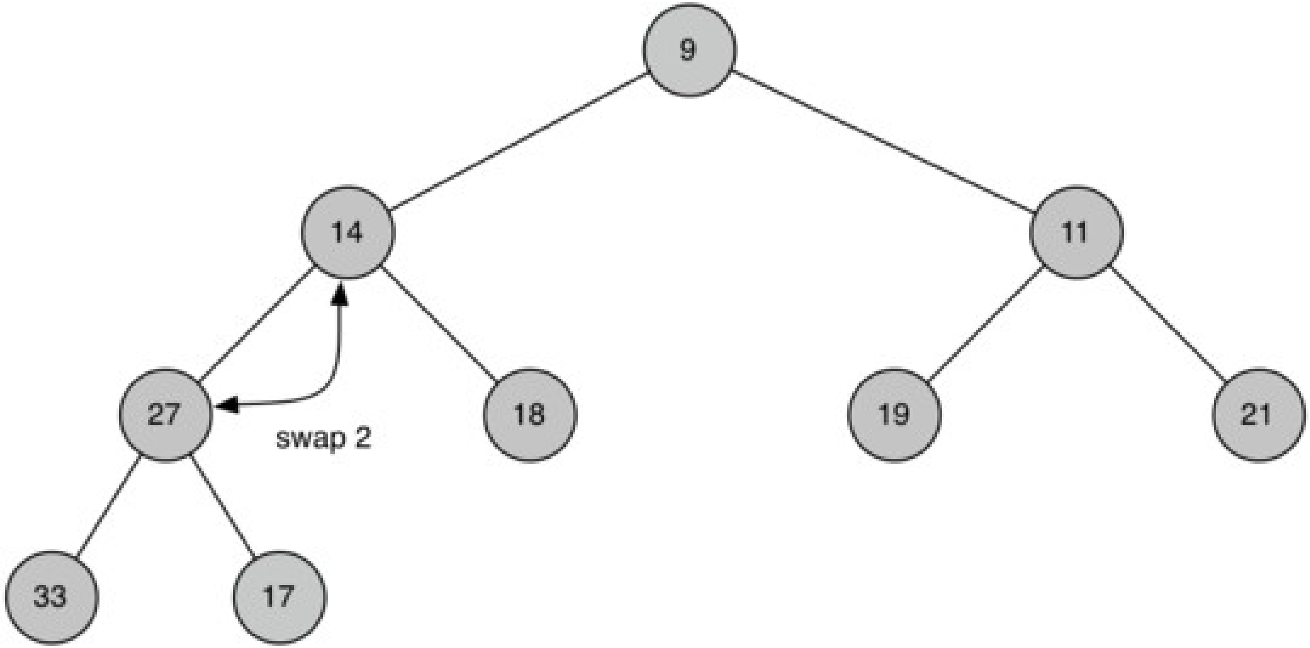
Binary Heaps



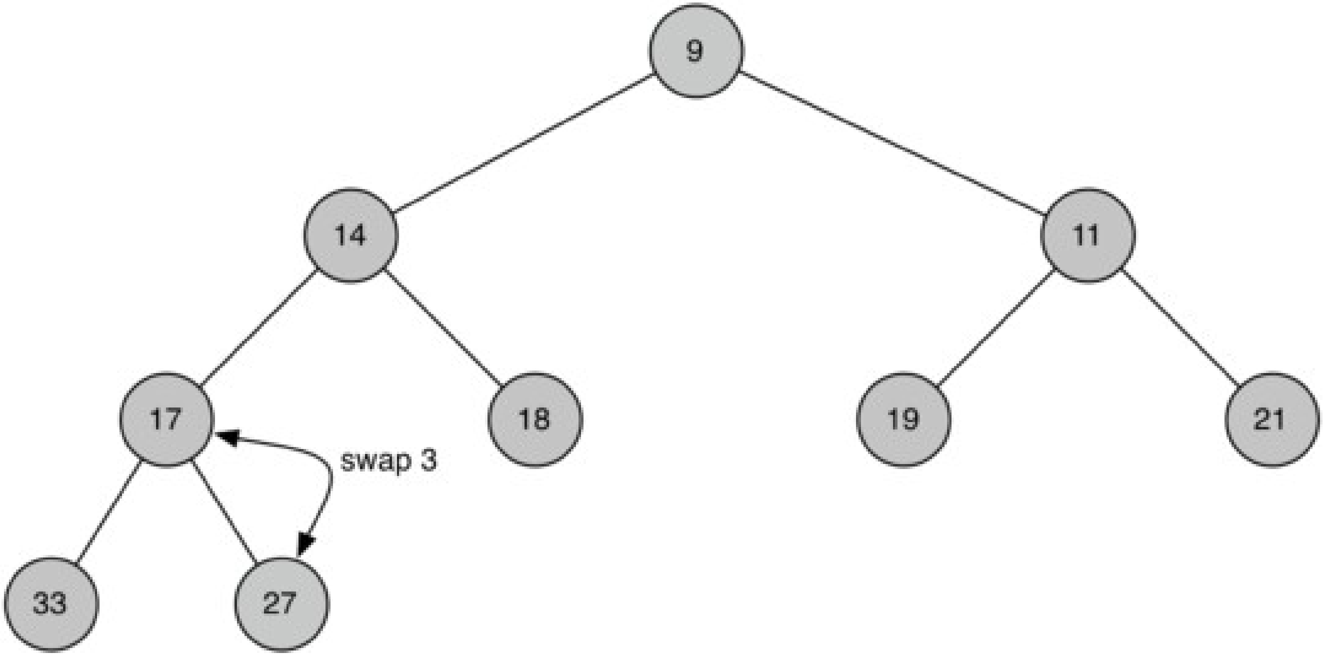
Binary Heaps



Binary Heaps



Binary Heaps



Binary Heaps

- In order to maintain the heap order property, all we need to do is swap the root with its smallest child less than the root.
- After the initial swap, we may repeat the swapping process with a node and its children until the node is swapped into a position on the tree where it is already less than both children.

Binary Heaps

- Code for percolating a node down the tree is found in the **percDown** an d **minChild**

```
def percDown(self,i):
    while (i * 2) <= self.currentSize:
        mc = self.minChild(i)
        if self.heapList[i] > self.heapList[mc]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[mc]
            self.heapList[mc] = tmp
        i = mc

def minChild(self,i):
    if i * 2 + 1 > self.currentSize:
        return i * 2
    else:
        if self.heapList[i*2] < self.heapList[i*2+1]:
            return i * 2
        else:
            return i * 2 + 1
```

Binary Heaps

- Code for **delMin**

```
def delMin(self):
    retval = self.heapList[1]
    self.heapList[1] = self.heapList[self.currentSize]
    self.currentSize = self.currentSize - 1
    self.heapList.pop()
    self.percDown(1)
    return retval
```

Binary Heaps

- To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys.
- The first method you might think of may be like the following.
- Given a list of keys, you could easily build a heap by inserting each key one at a time.
- Since you are starting with a list of one item, the list is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately O(logn) operations.

Binary Heaps

- However, remember that inserting an item in the middle of the list may require **O(n)** operations to shift the rest of the list over to make room for the new key.
- Therefore, to insert n keys into the heap would require a total of **O(nlogn)** operations.
- However, if we start with an entire list then we can build the whole heap in **O(n)** operations.

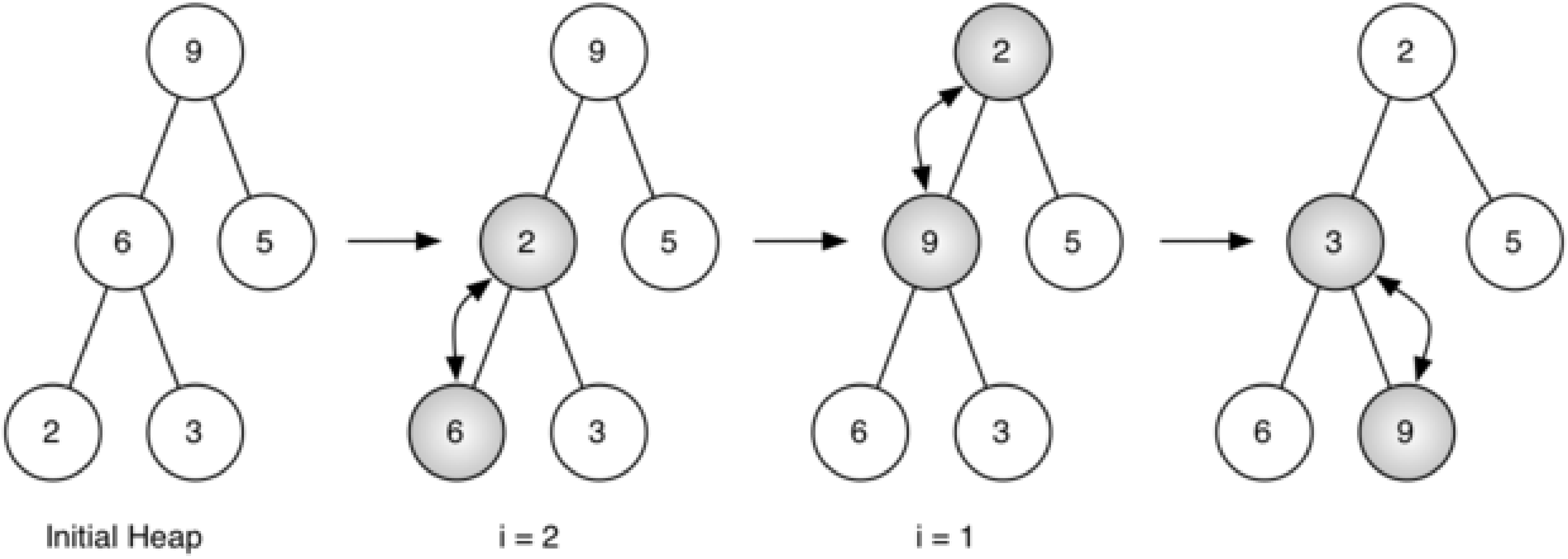
Binary Heaps

- Code to build the heap

```
def buildHeap(self,alist):
    i = len(alist) // 2
    self.currentSize = len(alist)
    self.heapList = [0] + alist[:]
    while (i > 0):
        self.percDown(i)
        i = i - 1
```

Binary Heaps

- Heap from list of [9,6,5,2,3]



Binary Heaps

- Make sure to review Jupyter Notebook and the Wikipedia article!
- This lecture will lead into the next topic – Binary Search Trees.