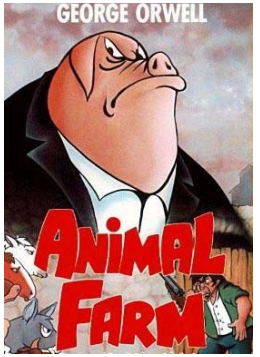
Priority Queues and Binary Heaps

Chapter 6.5

Some animals are more equal than others



- A queue is a FIFO data structure
  - the first element in is the first element out
  - which of course means the last one in is the last one out
- But sometimes we want to sort of have a queue but we want to order items according to some characteristic the item has.

Priorities

- We call the ordering characteristic the priority.
- When we pull something from this “queue” we always get the element with the best priority (sometimes *best* means lowest).
- It is really common in Operating Systems to use priority to schedule when something happens. e.g.
  - the most important process should run before a process which isn’t so important
  - data off disk should be retrieved for more important processes first

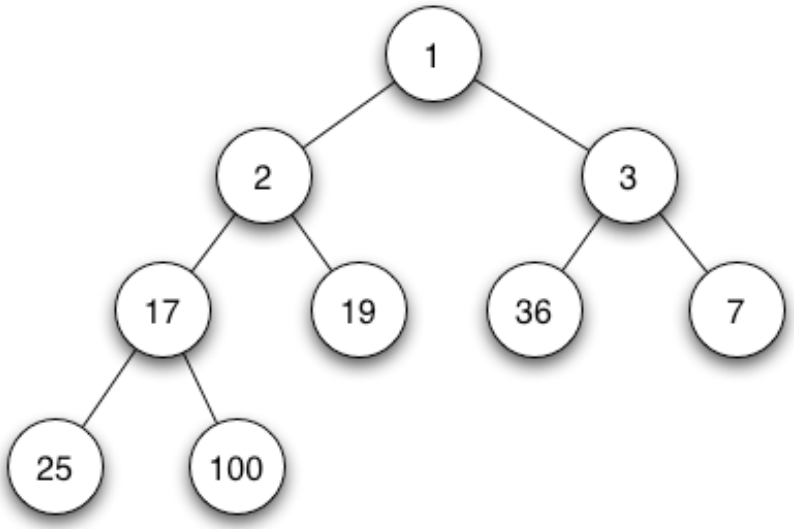
Priority Queue

- A priority queue always produces the element with the best priority when queried.
- You can do this in many ways
  - keep the list sorted
  - or search the list for the minimum value (if like the textbook - and Unix actually - you take the smallest value to be the best)
- You should be able to estimate the Big O values for implementations like this. e.g. O(n) for choosing the minimum value of an unsorted list.
- There is a clever data structure which allows all operations on a priority queue to be done in O(log n).

Binary Heap

Actually binary min heap

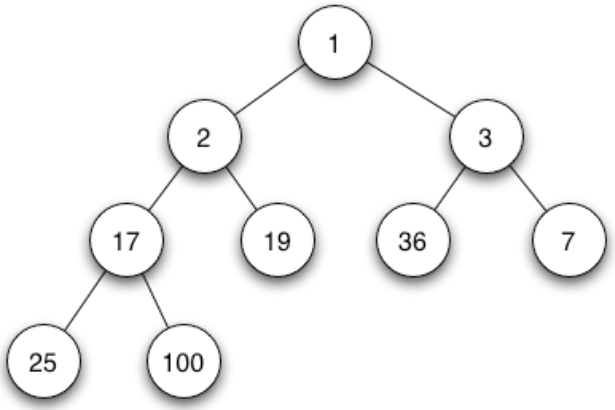
- Shape property - a complete binary tree - all levels except the last full. The last level nodes are filled from the left.
- Heap property - all parents are < or = to their children

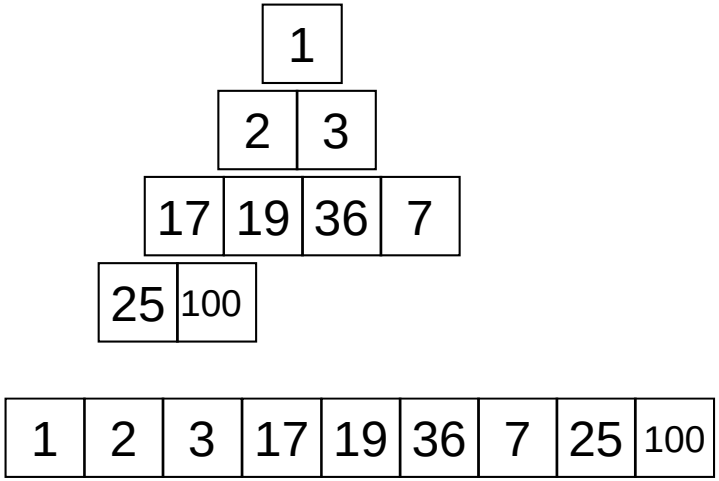


http://en.wikipedia.org/wiki/Binary_heap

Cool cleverness

- Complete binary trees can be represented very nicely in arrays or lists.
- Because there are no gaps we can represent the rows of the tree as a series of lists which we can then join together into one list.





My child is 2 x Me

- If we add an empty element at the beginning of the list we can then find the left child of any node at position p at 2 x p, and right child at 2 x p + 1.

- e.g. The children of the element at position 2 are in position 4 and 5. In this case the children of 2 are 17 and 19.

	1	2	3	17	19	36	7	25	100
0	1	2	3	4	5	6	7	8	9

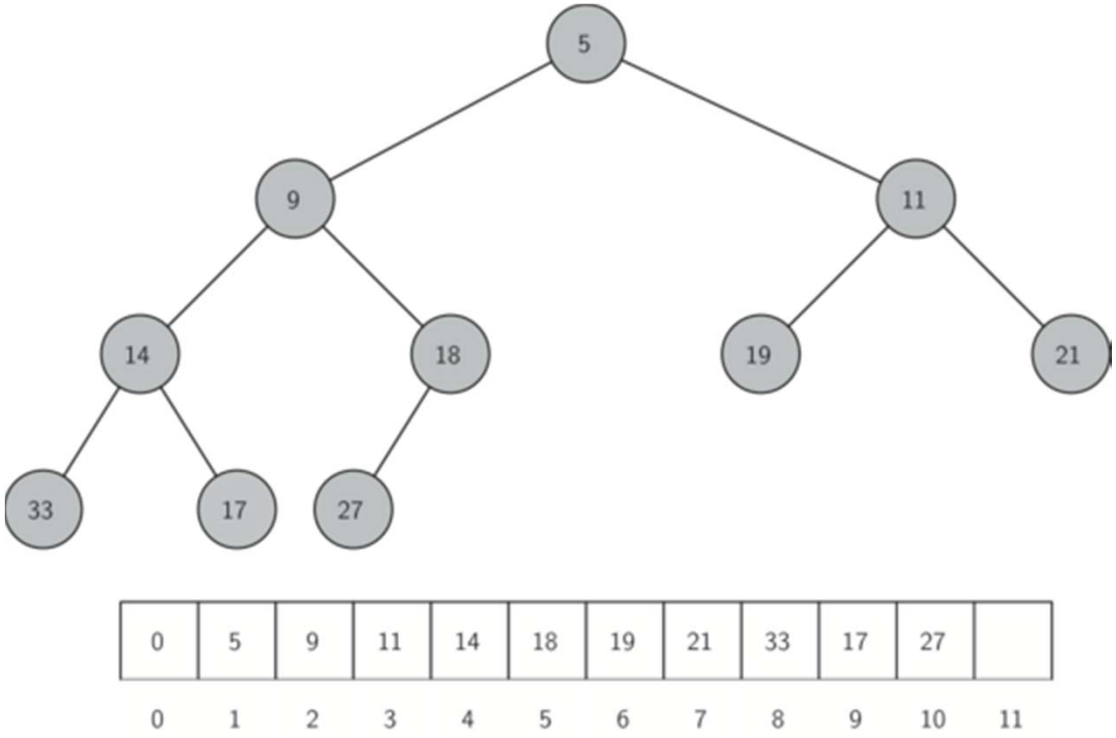


Figure 6.9: A Complete Binary Tree along with its List Representation

Binary Heap Operations

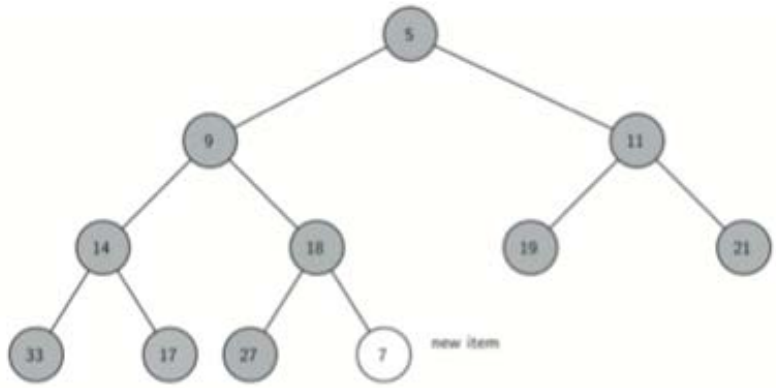
- Create an empty binary heap
- Insert an item in the heap
- Retrieve the smallest item (removing it)
- Get the size of the heap
- Build a heap from a list

Create a new binary heap

```
# We could start with something like:

class BinHeap:
    def __init__(self):
      self.heap_list = [0]

# We will develop something better later.
# N.B. I don't keep track of the size as the textbook
# does because we can use len(self.heap_list) - 1.

    def size(self):
        """Return the size of the heap."""
        return len(self.heap_list) - 1
```

Insert into the heap

- We want to maintain the two properties
  - the shape property (so the tree stays complete)
  - the heap property (so the smallest value is at the root)
- The shape property is easy to maintain. We just add the value into the next leaf position of our tree. In the list implementation this means we append the element onto the list.



Preserve the heap

- To keep the heap property we may need to move the new value further up the tree.
- We repeatedly compare it to its parent exchanging them if it is smaller than its parent.
- This is sometimes called percolating because the smallest values bubble to the top.

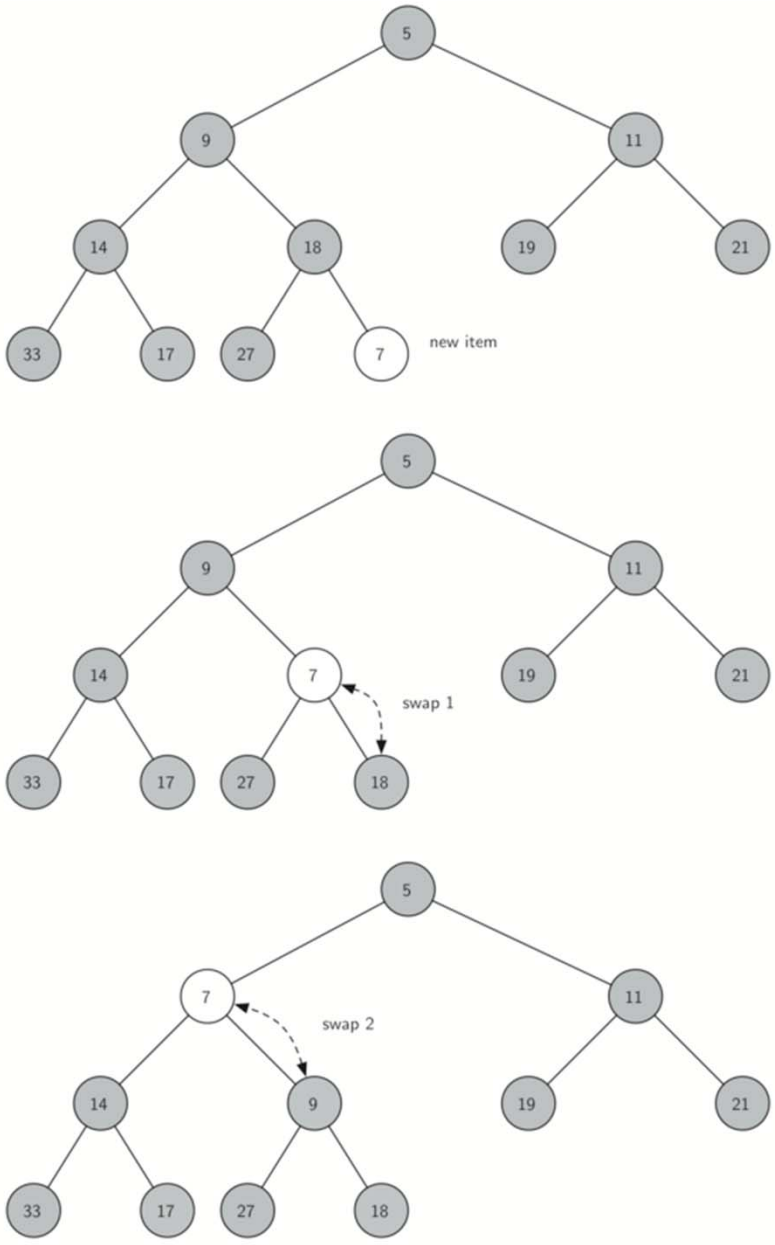


Figure 6.10: Percolate the New Node up to Its Proper Position

Why is the exchange safe?

- When we are swapping a value with its parent (say we are halfway up the tree) we know that the heap property will still be true for the subtree we are currently looking at because
  - The new value is smaller than its parent and if the other child was greater than or equal to the parent value it must be greater than the new value.
  - Any subtree below the current position of the new value must have values all greater than the parent so they are still in the correct positions.

Insertion code

```
def insert(self, k):
   self.heap_list.append(k)
   self.perc_up(self.size())


def perc_up(self, i):
   # we keep comparing with the parent until we reach the top
   # or the parent is smaller than the child
   while i // 2 > 0 and self.heap_list[i] < self.heap_list[i // 2]:
      self.heap_list[i], self.heap_list[i // 2] = \
          self.heap_list[i // 2], self.heap_list[i]
      i = i // 2
```

Getting the minimum value

- With a min heap this is trivial.
  - It is the root element or in our list implementation the element at index 1.
- If we remove it we need to fix the shape and heap properties.
  - The shape is easy to fix once again - we move the last leaf in the heap to the first position (or root).

Preserving the heap again

- This time we have almost certainly moved a larger value into the root.
- We want to percolate this value down the tree.

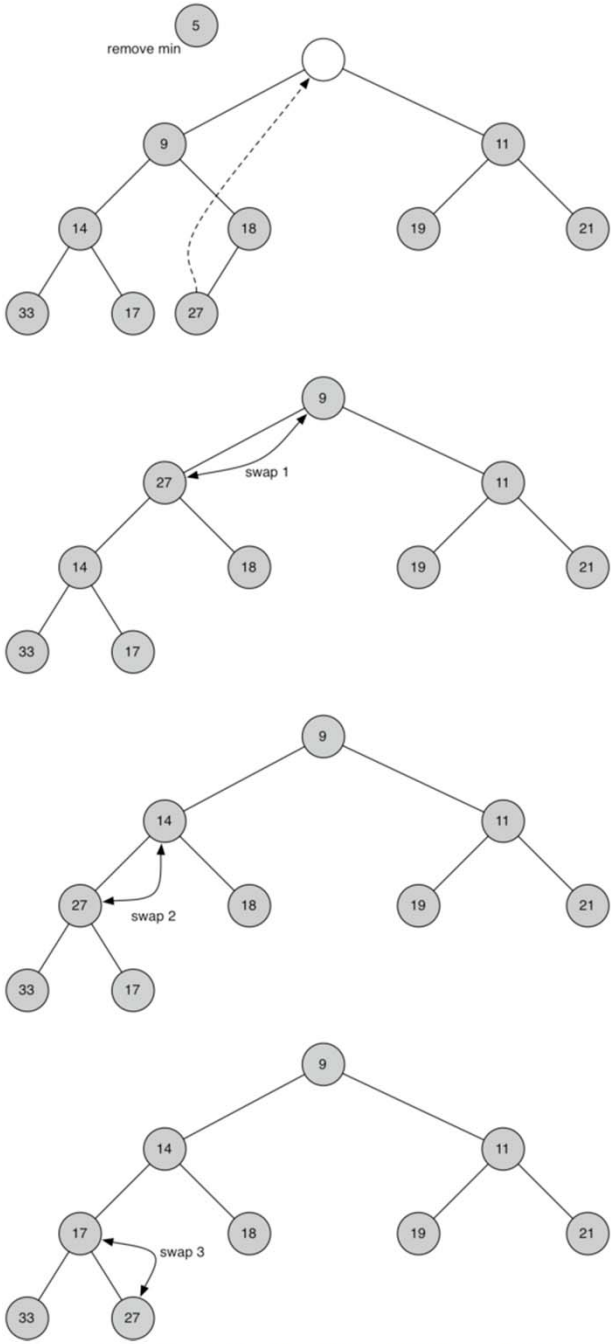


Figure 6.11: Percolating the Root Node down the Tree

Going down is a little trickier than going up

- Percolating up was straight-forward because each child only has one parent.
- Percolating down we want to swap with the smallest child (to preserve the heap property).

Deleting code

```
def del_min(self):
    ret_val = self.heap_list[1]
    replacement = self.heap_list.pop()
    if self.size() > 0:
        self.heap_list[1] = replacement
    self.perc_down(1)
    return ret_val
```

Percolating down

```
def perc_down(self, i):
   while (i * 2) <= self.size():
      child = self.min_child(i)
      if self.heap_list[i] > self.heap_list[child]:
         self.heap_list[child], self.heap_list[i] = \
            self.heap_list[i], self.heap_list[child]
      i = child


def min_child(self, i):
   """Return the index of the minimum child of index i."""
   left = i * 2
   right = left + 1
   if right > self.size():
      return left
   if self.heap_list[left] < self.heap_list[right]:
      return left
   else:
      return right
```

Building a heap from a list

- We can do it the boring way.
  - Start with an empty list and insert each element in turn.
  - This will be O(n log n) as there are n elements to add and each could percolate up the levels of the tree.
- Or we can do it the clever way.

The Efficient Way

- We start half way through the list.

  - Any nodes in the last half of the list are leaves because of the shape of our complete binary tree. Each level of the tree has one more node than the sum of the nodes in all previous levels.

- Moving backwards through the list percolate each element down to its correct position.

- This is O(n) - but I am not going to prove it :).

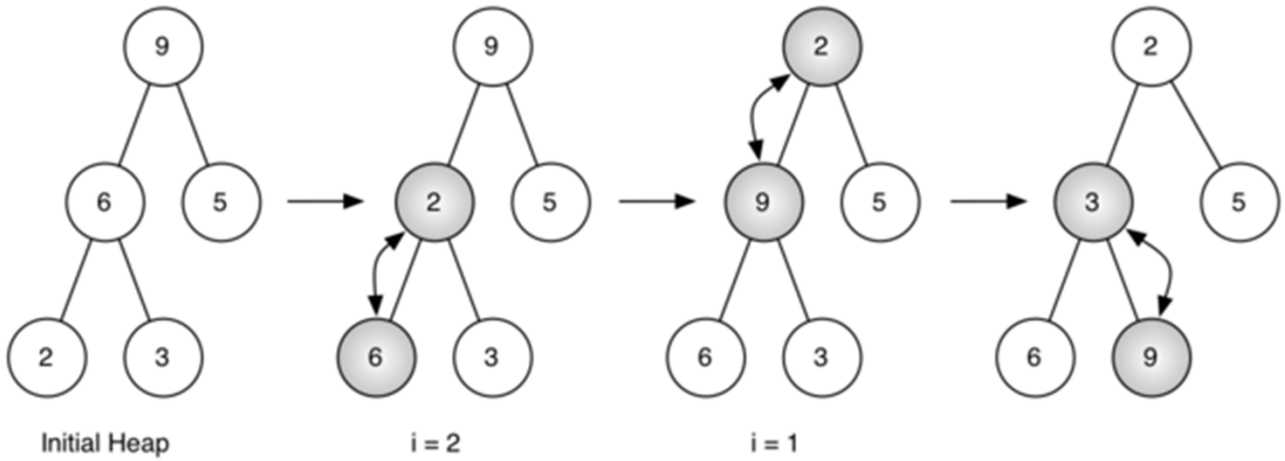
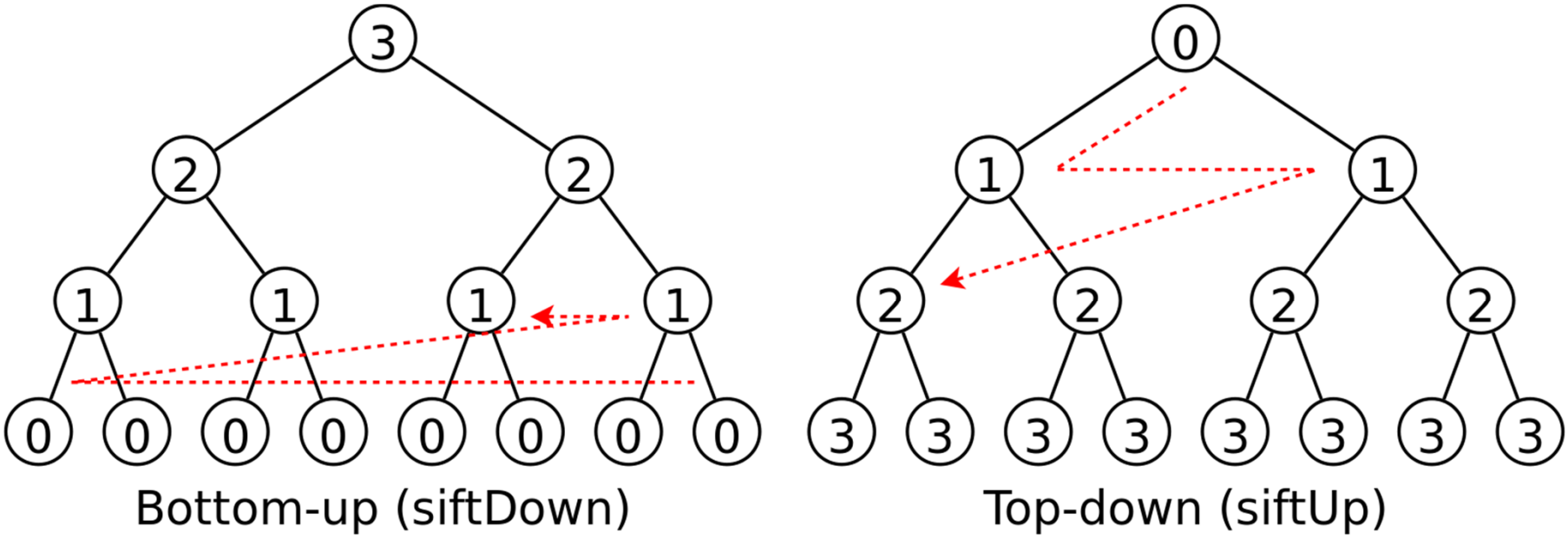


Figure 6.12: Building a Heap from the List $[9, 6, 5, 2, 3]$

This is what Figure 6.12 should look like.

Sifting up and down



The number in the circle indicates the maximum times of swapping required when adding the node to the heap.

Building the heap code

```
def __init__(self, a_list=[]):
    """Constructs a BinHeap object.

    Can either create an empty BinHeap or can construct
    it from a_list.
    """
    self.heap_list = [0] + a_list # add the dummy to list
    for i in range(self.size() // 2, 0, -1):
        self.perc_down(i)
```

Different heaps but the same outcome

```
bh = BinHeap()
bh.insert(9)
bh.insert(6)
bh.insert(5)
bh.insert(2)
bh.insert(3)
print(bh)
```

Output

```
[2, 3, 6, 9, 5]
```

```
bh = BinHeap([9,6,5,2,3])
print(bh)
```

```
[2, 3, 5, 6, 9]
```

And by the way

- The BinHeap class has a special __str__ method to print out the heap values but leave out the dummy first element.

```
def __str__(self):
    return str(self.heap_list[1:])
```