# Heaps 9

The word *heap* is used in a couple of different contexts in Computer Science. A heap sometimes refers to an area of memory used for dynamic (i.e. run-time) memory allocation. Another meaning, and the topic of this chapter, is a data structure that is conceptually a complete binary tree. Heaps are used in implementing priority queues, the heapsort algorithm, and some graph algorithms. Heaps are somewhat like binary search trees in that they maintain an ordering of the items within the tree. However, a heap does not maintain a complete ordering of its items. This has some implications for how a heap may be used.

## 9.1 Chapter Goals

By the end of this chapter you should be able to answer the following questions:

- What is a heap and how is it used?
- What is the computational complexity of adding and deleting items from a heap?
- Would you use a heap to look up items or not?
- When would you use a heap?
- In the heapsort algorithm, why is it advantageous to construct a largest-on-top heap?

## 9.2 Key Ideas

To understand heaps we'll start with a definition. A *largest-on-top heap* is a complete ordered tree such that every node is $\geq$ all of its children (if it has any). An example will help illustrate this definition. Conceptually, a heap is a tree that is full on all levels except possibly the lowest level which is filled in from left to right. It takes the general shape shown in Fig. 9.1.

K.D. Lee and S. Hubbard, *Data Structures and Algorithms with Python*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-13072-9_9

**Fig. 9.1** Heap Shape

Conceptually a heap is a tree, but heaps are generally not stored as trees. A *complete* tree is a tree that is full on all levels except the lowest level which is filled in from left to right. Because heaps are complete trees, they may be stored in an array. An example will help in understanding heaps and the complete property better. Consider a largest on top heap with the root node stored at index 0 in an array. Conceptually, Fig. 9.2 is a heap containing integers.

The data in this conceptual version is stored in an array by traversing the tree level by level starting from the root node to the heap. The conceptual heap in Fig. 9.2 would be stored in an array as organized in Fig. 9.3.

There are two properties that a heap exhibits. They are:

- *Heap Structure Property*: The elements of the heap form a complete ordered tree.
- *Heap Order Property*: Every parent $\geq$ all children (including all descendants).

The heap in Fig. 9.2 maintains these two properties. The array implementation of this heap in Fig. 9.3 also maintains these properties. To see how the properties are maintained in the array implementation we need to be able to compute the location

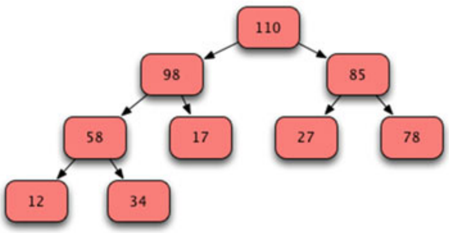

**Fig. 9.2** Sample Heap

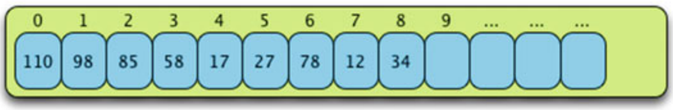

**Fig. 9.3** Heap Organization

of children and parents. The children of any element of the array can be calculated from the index of the parent.

$$leftChildIndex = 2 * parentIndex + 1$$

$$rightChildIndex = 2 * parentIndex + 2$$

Using these formulae on Fig. 9.3 we can see that the children of the root node (i.e. index 0) are 98 (at index 1) and 85 (at index 2). Likewise, the children of 85 are located at index 5 and 6 which are the values 27 and 78, which we can verify are the same children as in the conceptual model.

Of course, not every node has a child or even two children. If the computed *leftChildIndex* or *rightChildIndex* are greater than or equal to the number of values in the heap, then the node in question is a leaf node.

It is also possible to go in the other direction. Given a child's index, we can discover where the parent is located.

$$parentIndex = (childIndex - 1)//2$$

The // in the previous formula represents integer division. It means that the result is always an integer. If there were a fractional part we round down to the next lower integer. So, the index of the parent of the 34 in Fig. 9.3 is computed as

$$parentIndex = (8 - 1)//2 = 3$$

Consulting the conceptual model in Fig. 9.2, we see that the value at index 3 in the array, the 58, is indeed the parent of the 34. It should be noted that not every node in a heap has a parent. In particular, the root node, at index 0, does not have a parent. All other nodes in a heap have parents.

## 9.3 Building a Heap

Now that we've seen what a heap looks like, we'll investigate building a heap. Heaps can be built either largest on top or smallest on top. We'll build a largest on top heap. A *Heap* class will encapsulate the data and methods needed to build a heap. Heap objects contain a list and count of the number of items currently stored in the heap. We'll call this count the *size* of the heap. To encapsulate the data we'll want a method that will take a sequence of values and build a heap from it. We'll call this method *buildFrom*. A private method will also be useful. The *buildFrom* method will call the *_siftUpFrom* to get each successive element of the sequence into its correct position within the heap.

### 9.3.1 The buildFrom Method

```python
def buildFrom(self, aSequence):
        '''aSequence is an instance of a sequence collection which
        understands the comparison operators. The elements of
        aSequence are copied into the heap and ordered to build
        a heap. '''
```

```
def __siftUpFrom(self, childIndex):
        '''childIndex is the index of a node in the heap. This method sifts
        that node up as far as necessary to ensure that the path to the root
        satisfies the heap condition. '''
```

The sequence of values passed to the *buildFrom* method will be copied into the heap. Then, each subsequent value in the list will be sifted up into its final location in the heap. Consider the list of values [71, 15, 36, 57, 101]. We'll trace this through showing the resulting heap at each stage. To begin the list is copied into the heap object in the order given here. Then *siftUpFrom* is called on each subsequent element. To begin, *siftUpFrom* is called on the second element, the 57 in this case. Calling *siftUpFrom* on the root of the heap would have no effect. Normally, the parent index of the node is computed. The parent will already be greater than the other child (if there is one). If the value at the current child index is greater than the value at the parent index, then the two are swapped and the process repeats. This process repeats as many times as are necessary: either until the root node is reached (i.e. index 0 of the list) or until the new node is in the proper location to maintain the heap property.

The first time some movement occurs is when 57 is added to the heap. The 57 is swapped with the 15 to arrive at its final location resulting in the heap in Fig. 9.4.

The first four elements of the list now make up a heap. But, the 101 is not in its final position. We need to sift it up in the heap to get it to its final position. Looking at the conceptual view of the heap (i.e. the tree), you can see that 101 is a child of the node containing 57. Clearly, that violates the heap property. So, 101 and 57 are swapped to sift the 101 up as shown in Fig. 9.5.

Without looking at the conceptual model you can still compute the parent of the node containing 101 in part one of Fig. 9.5. When 101 was at index 4 in the list, the parent index was computered as follows.

$$parentIndex = (4 - 1)//2 = 1$$

So, the 101 is compared to the 57 at index 1 in part one above. Then, the swap is made because 101 is greater than 57 in part two. However, the 101 is still not in the

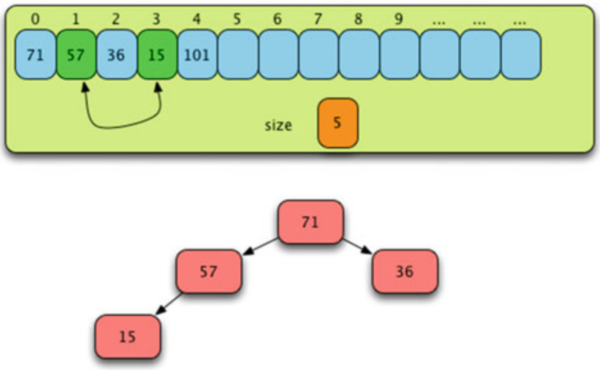

**Fig. 9.4** Building a Heap Part One

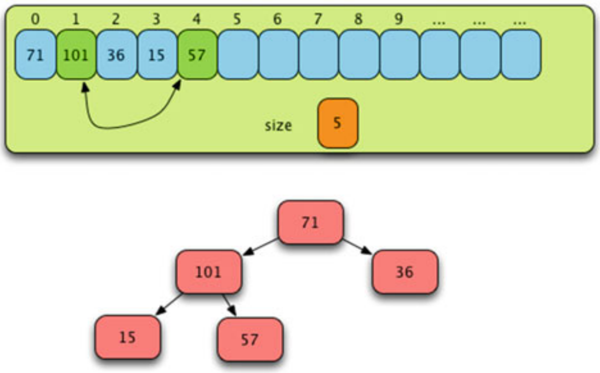

**Fig. 9.5** Building a Heap Part Two

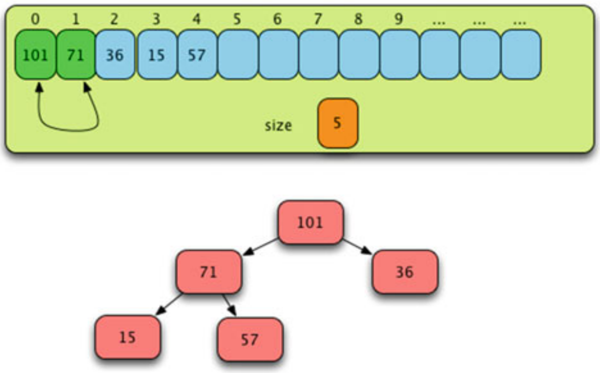

**Fig. 9.6** Building a Heap Part Three

right place.

$$parentIndex = (1 - 1)//2 = 0$$

We compare 101 with the 71 and swap the two elements. This is the last iteration of sifting up because 101 has now reached the root (i.e. index 0) of the heap. After swapping the two values we get the heap in Fig. 9.6.

## 9.4 The Heapsort Algorithm Version 1

Heaps have two basic operations. You can add a value to a heap. You can also delete, and retrieve, the maximum value from a heap if the heap is a largest on top heap. Using these two operations, or variations of them, we can devise a sorting algorithm by building a heap with a list of values and then removing the values one by one in descending order. We'll call these two parts of the algorithm phase I and phase II. To implement phase I we'll need one new method in our Heap class.

### 9.4.1 The addToHeap Method

```python
def addToHeap(self,newObject):
'''If the heap is full, double its current capacity.
   Add the newObject to the heap, maintaining it as a
   heap of the same type.  Answer newObject.'''
```

This new method can use the *__siftUpFrom* private method to get the new element to its final destination within the heap. Version 1 of Phase I calls *addToHeap n* times. This results in O(n log n) complexity. The specific steps of phase I include:

1. double the capacity of the heap if necessary.
2. data[size] = newObject
3. __siftUpFrom(size)
4. size + = 1.

As you can see, *__siftUpFrom* will be called *n* times, once for each element of the heap. Each time *__siftUpFrom* is called, the heap will have grown by 1 element. Consider the heap in Fig. 9.7 just before pass #9.

We are about to sift the 98 up to its rightful location within the heap. Conceptually we have the picture of the heap shown in Fig. 9.8.

To move the 98 to the correct location we must compute the parent index from the child indices as shown in Table 9.1.

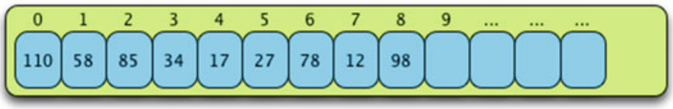

**Fig. 9.7** Adding 98 to the Heap

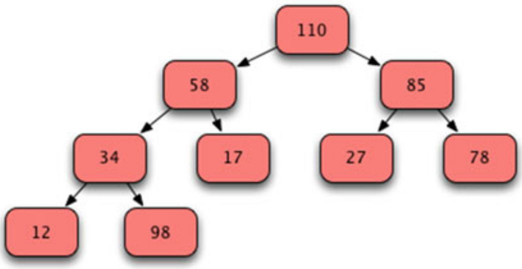

**Fig. 9.8** Conceptual View While Adding 98 to the Heap

**Table 9.1** Child and Parent Indices

| childIndex | parentIndex = (childIndex − 1)//2 |
|---|---|
| 8 | 3 (swap) |
| 3 | 1 (swap) |
| 1 | 0 (stop) |

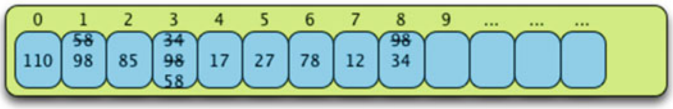

**Fig. 9.9** Heap After Moving 98 to Correct Location

Sifting up the 98 in the heap's list results in two swaps before it reaches its final location. Figure 9.9 shows the 98 is swapped with the 34 at index 3. Then it is swapped again with the 58 at index 1. At this point no more swaps are done because 101 is greater than 98. The 98 has reached its proper position within the heap.

## 9.5 Analysis of Version 1 Phase I

The approach taken in version 1 of phase I is slow as we shall see. Consider a perfect complete binary tree. One which is completely full on all levels with *h* levels as shown in Fig. 9.10.

Consider the relationship between the number of levels and the number of items in the heap as shown in Table 9.2.

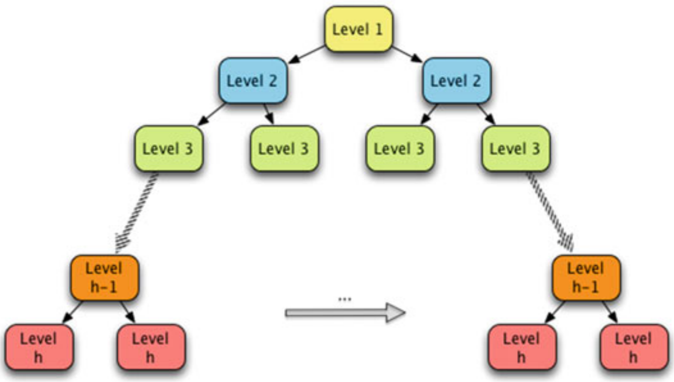

**Fig. 9.10** A Perfect Binary Tree

**Table 9.2** Heap levels versus Heap size

| Level | # of nodes at level |
|---|---|
| 1 | $(2^{1-1} = 1)$ |
| 2 | $(2^{2-1} = 2)$ |
| 3 | $(2^{3-1} = 4)$ |
| ... | ... |
| h | $(2^{h-1})$ |

For a heap with $n$ items in it, the value of $n$ can be computed by adding up all the nodes at each level in the heap's tree. To simplify our argument we'll assume that the heap is a full binary tree.

$$n = 1 + 2^1 + 2^2 + \cdots + 2^{h-1} \textit{for some } h$$

This is the sum of a geometric sequence. The sum of a geometric sequence can be computed as follows.

$$1 + r + r^2 + r^3 + \cdots + r^m = \frac{r^{m+1} - 1}{r - 1} \textit{if } r \neq 1$$

Applying this formula to our equation above the number of nodes in a complete binary tree (i.e. a full binary heap) with $h$ levels is given by this formula below.

$$n = 1 + 2^1 + 2^2 + \cdots + 2^{h-1} = \frac{2^h - 1}{2 - 1} = 2^h - 1$$

This implies that $n + 1 = 2^h$. We can solve this equation for $h$. Doing so we get

$$h = \lceil log_2(n + 1) \rceil$$

The brackets above are the ceiling operator and it simply means that we should round up to the next highest integer. Rounding up takes into account that not every heap tree is completely full so there may be some values of $n$ that won't give us an integer for $h$ if we didn't round up. The following inequality will be useful in determining the computational complexity of phase I of the heapsort algorithm.

$$log_2 I \leq \lceil log_2(I + 1) \rceil \leq log_2 I + 1 \ \ \forall I \geq 2$$

So far we have been able to determine that the height of a complete binary tree (i.e. the number of levels) is equivalent to the ceiling of the log, base 2, of the number of elements in the tree $+1$. Phase I of our algorithm appends each value to the end of the list where it is sifted up to its final location within the heap. Since sifting up will go through at most $h$ levels and since the heap grows by one each time, the following summation describes an upper limit of a value that is proportional to the amount of work that must be done in Phase I.

$$\sum_{I=2}^{N} \lceil log_2(I + 1) \rceil$$

But applying the inequality presented above we have the following. The N − 1 term comes from the last summation from 2 to N. From the inequality above there are N − 1 ones that are a part of the summation. These can be factored out as N − 1.

$$\sum_{I=2}^{N} log_2\, I \le \sum_{I=2}^{N} \lceil log_2(I+1) \rceil \le \sum_{I=2}^{N} (log_2\, I + 1) = (\sum_{I=2}^{N} log_2\, I) + (N-1)$$

We now have a lower and upper bound for our sum. The same summation appears in both the lower and upper bound. But what does $\sum_{I=2}^{N} log_2\, I$ equal? The following equivalences will help in determining this summation.

$$y = log_2\, x \Leftrightarrow 2^y = x \Leftrightarrow y\, ln\, 2 = ln\, x \Leftrightarrow log_2\, x = y = \frac{1}{ln\, 2} ln\, x$$

To determine what the summation above is equal to we can establish a couple of inequalities that bound the sum from above and below. In Fig. 9.11, the summation can be visualized as the green area. The first term in the summation would provide the first green rectangle, the second green rectangle corresponds to the second term in the summation and so on. The black line in the figure is the plot of the log base 2 of $x$. Clearly the area covered by the green rectangles is bigger than the area under

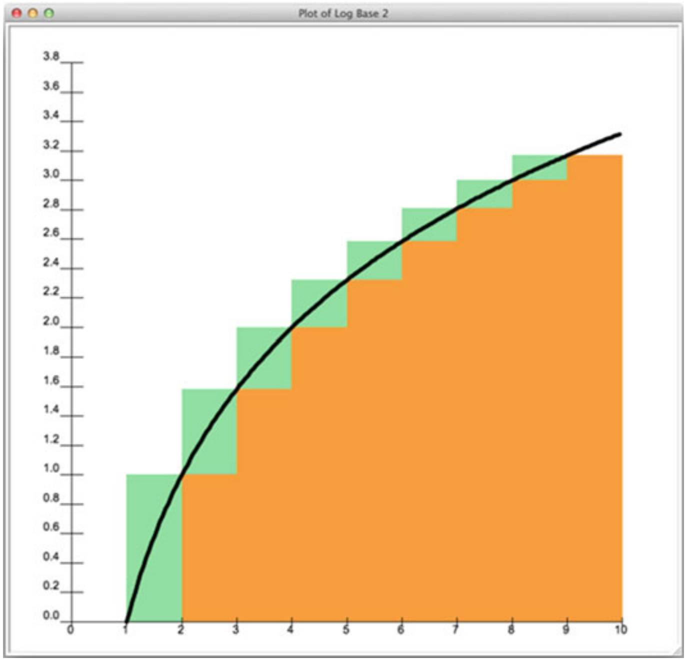

**Fig. 9.11** Plot of log(n)

the curve of the log. The area under the curve can be found by taking the definite integral from 1 to N, which in the picture is 9 but in general would be N. From this we get the following inequality.

$$\int_1^N log_2\, x\, dx \le \sum_{I=2}^{N} log_2\, I$$

Now, consider shifting the entire green area to the right by one. In the figure above, that's the orange area. The orange and green areas are exactly the same size. The orange is just shifted right by one. Now look at the plot of the log base 2 of *x*. The area below the curve is now clearly bigger than the orange area. If we imagine this graph going out to N, then we'll have to include N + 1 in our definite integral (since we shifted the orange area to the right). So we get the following inequality.

$$\sum_{I=2}^{N} log_2\, I \le \int_2^{N+1} log_2\, x\, dx$$

Putting the two inequalities together we have a lower and upper bound for our summation.

$$\int_1^N log_2\, x\, dx \le \sum_{I=2}^{N} log_2\, I \le \int_2^{N+1} log_2\, x\, dx$$

It is easier to integrate using natural log so we'll rewrite the integral as follows.

$$\int_1^N log_2\, x\, dx = \int_1^N \frac{1}{ln\, 2} ln\, x\, dx$$

The constant term in the integral can be factored out. So we'll look at the following integral.

$$\int_1^N ln\, x\, dx$$

We can find the result of the definite integral that appears above by doing integration by parts. The integration by parts rule is as follows.

$$\int_a^b u\, dv = uv\Big|_a^b - \int_a^b v\, du$$

Applying this to our integral we have the following

$$u = ln\, x \text{ and } dv = dx$$

$$du = \frac{1}{x}dx \text{ and } v = x$$

$$\Rightarrow \int_1^N ln\, x\, dx = x\, ln\, x\Big|_1^N - \int_1^N x\, \frac{1}{x}dx$$

$$= x\, ln\, x\Big|_1^N - \int_1^N 1dx = x\, ln\, x\Big|_1^N - x\Big|_1^N$$

$$= N\, ln\, N - (N - 1)$$

We have proved that the lower bound is proportional to *N log N*. Similarly, we could prove that the upper bound is also proportional to *N log N*. Therefore the work done by inserting N elements into a heap using the *__siftUpFrom* method is $\theta$(*N log N*). We can do better! If the values in the heap were in the correct order we could achieve *O*(*N*) complexity. Using a different approach we will be able to achieve *O*(*N*) complexity in all cases.

## 9.6 Phase II

Later, we will investigate how to improve the performance of phase I. Recall that phase I of the heapsort algorithm builds a heap from a list of values. Phase II takes the elements out of the heap, one at a time, and places them in a list. To save space, the same list that was used for the heap may be used for the list of values to be returned. Each pass of phase II takes one item from the list and places it where it belongs and the size of the heap is decremented by one. The key operation is the *__siftDownFromTo* method (Fig. 9.12).

### 9.6.1 The siftDownFromTo Method

```
def __siftDownFromTo(self, fromIndex, lastIndex):
        '''fromIndex is the index of an element in the heap.
        Pre: data[fromIndex..lastIndex] satisfies the heap condition,
        except perhaps for the element data[fromIndex].
        Post:  That element is sifted down as far as neccessary to
        maintain the heap structure for data[fromIndex..lastIndex].'''
```

To illustrate this method, let's take our small heap example and start extracting the values from it. Consider the heap in Fig. 9.13 where both the conceptual view and the organization of that heap are shown. 101 is at the top of the heap and is also the largest value.

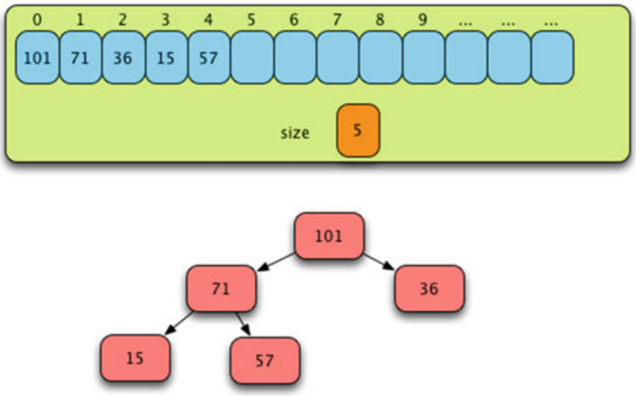

**Fig. 9.12** Just Before Phase II

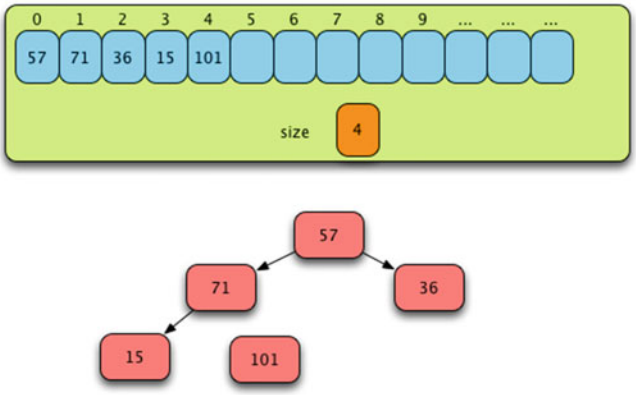

**Fig. 9.13** After Swapping First and Last Values

If sorted, the 101 would go at the end of the list. Since there are 5 elements in the heap, we'll swap the 57 and the 101. By doing this, 101 is at its final position within a sorted list. The 57 is not in the correct location within the heap. So, we call the *__siftDownFromTo* method to sift the 57 down from the 0 position within the heap to at most the *size-1* location.

The *__siftDownFromTo* method does its work and swaps the 57 with the bigger of the two children, the 71. The 57 does not need to sift down any further since it is bigger than the 15. So we have the view of the heap in Fig. 9.14 after the first pass of Phase II.

The second pass of Phase II swaps the 15 and the 71, moving the 71 to its final location in the sorted list. It then sifts the 15 down to its rightful location within the heap, producing the picture you see in Fig. 9.15.

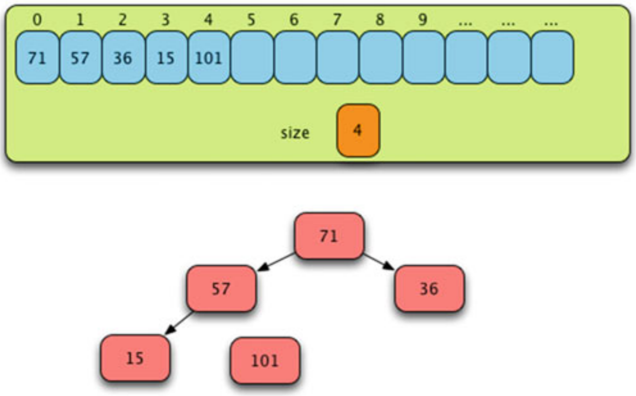

**Fig. 9.14** After the First Pass of Phase II

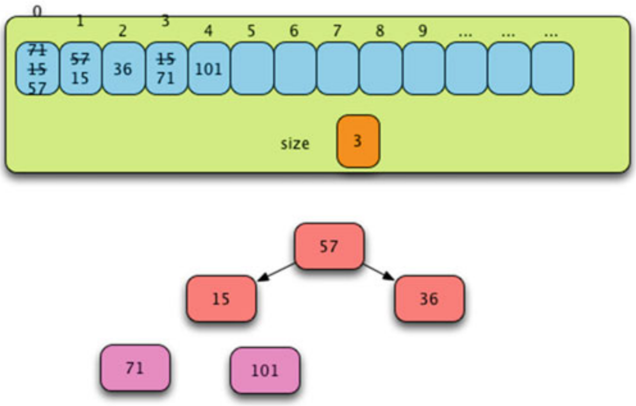

**Fig. 9.15** After the Second Pass of Phase II

During the third pass of Phase II the 57 is put in its final location and swapped with the 36 to make room for it. Although *__siftDownFromTo* is called, no movement of values within the heap occurs because the 36 is at the top and is the largest value in the heap (Fig. 9.16).

During the fourth and final pass, the 36 is swapped with the 15. No call to **_siftDownFromTo** is necessary this time since the heap is only of size 1 after the swap. Since a heap of size 1 is already sorted and in the right place, we can decrement the size to 0. The list is now sorted in place without using an additional array as shown in Fig. 9.17.

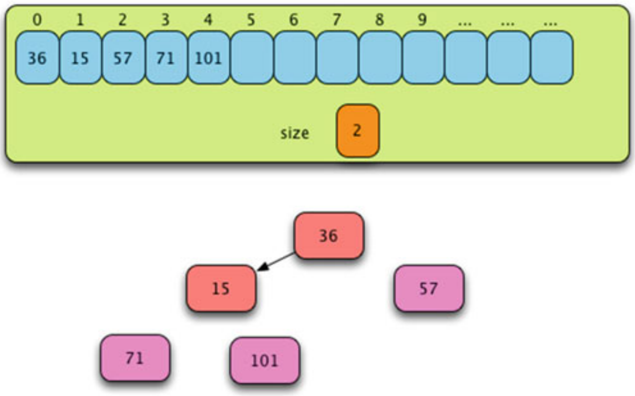

**Fig. 9.16** After the Third Pass of Phase II

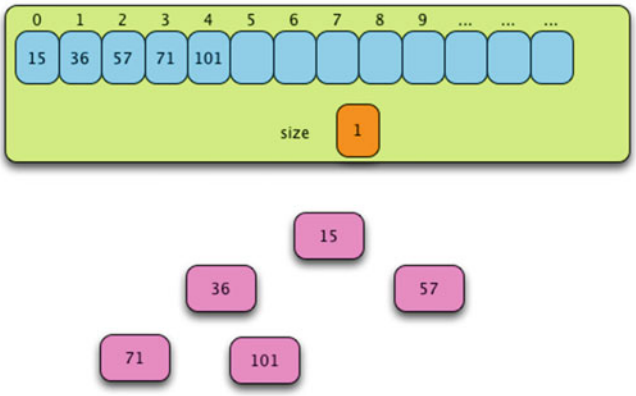

**Fig. 9.17** After the Fourth and Final Pass of Phase II

## 9.7 Analysis of Phase II

The work of Phase II is in the calls to the *__siftDownFromTo* method which is called N − 1 times. Each call must sift down an element in a tree that shrinks by one element each time. Earlier in this chapter we did the analysis to determine that the amount of work in the average and worst case is proportional to

$$\sum_{I=2}^{N} \lceil log_2(I+1) \rceil \equiv \theta(NlogN)$$

The best case of Phase II would require that all values in the heap are identical. In that case the computational complexity would be O(N) since the values would never sift down. This best case scenario brings up a good point. If we could limit how far down the value is sifted, we might be able to speed up Phase I. That's the topic or our next section.

## 9.8 The Heapsort Algorithm Version 2

In version one, the heapsort algorithm attained O(N log N) complexity during Phase I and Phase II. In version two, we will be able to speed up Phase I of the heapsort algorithm up to O(N) complexity. We do this by limiting how far each newly inserted value must be sifted down. The idea is pretty simple, but yet a powerful technique. Rather than inserting each element at the top of the heap, we'll build the heap, or heaps, from the bottom up. This means that we'll approach the building of our heap by starting at the end of the list rather than the beginning. An example will help make this more clear. Consider the list of values in Fig. 9.18 that we wish to sort using heapsort.

Rather than starting from the first element of the list, we'll start from the other end of the list. There is no need to start with the last element as we will see. We need to pick a node that is a parent of some node in the tree. Since the final heap is a binary heap, the property we have is that half the nodes of the tree are leaf nodes and cannot be parents of any node within the heap. We can compute the first parent index as follows.

$$parentIndex = (size - 2)//2$$

The size above is the size of the list to be sorted. Note that because the list has indices 0 to size-1 we must subtract two to compute the proper parentIndex in all cases. In this case, that *parentIndex* is 2. We need to start with index 2 in the list to start building our heaps from the bottom up. Index 2 will be the first *parent* and we'll sift it down as far as necessary.

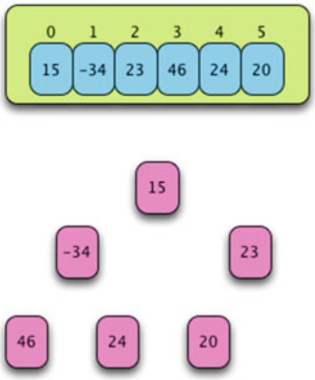

**Fig. 9.18** A List to be Heapsorted

$$childIndex1 = 2 * parentIndex + 1 = 2 * 2 + 1 = 5$$
$$childIndex2 = 2 * parentIndex + 2 = 2 * 2 + 2 = 6$$

Since the second of these indices is beyond the last index of the list, the *__siftDownFromTo* method will not consider *childIndex2*. After considering the 20 and the 23 we see that those two nodes do in fact form a heap as shown in Fig. 9.19. We will show this in the following figures by joining them with an arrow. We now have 5 heaps, one less than we started with. More importantly, we only had to sift the parent down one position at the most.

Next, we move back one more in the list to index 1. We call *__siftDownFromTo* specifying to start from this node. Doing so causes the sift down method to pick the larger of the two children to swap with, forming a heap out of the three values −34, 46, and 24 as a result. This is depicted in Fig. 9.20.

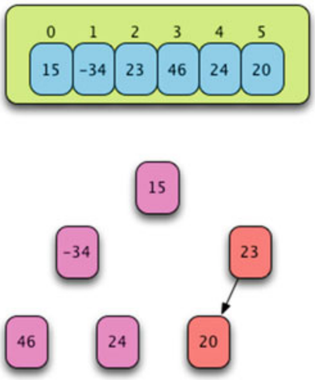

**Fig. 9.19** After Forming a Sub-Heap

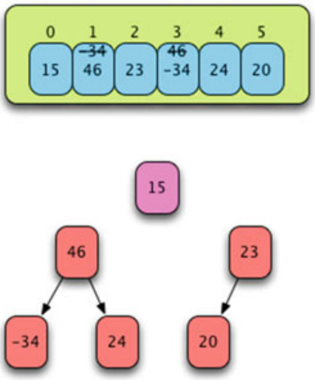

**Fig. 9.20** After Forming a Second Sub-Heap

Finally, we move backward in the list one more element to index 0. This time we only need to look at the values of the two children because they will already be the largest values in their respective heaps. Calling *__siftDownFromTo* on the first element of the list will pick the maximum value from 15, 46, and 23 and will swap the 15 with that value resulting in the situation in Fig. 9.21.

This doesn't form a complete heap yet. We still need to move the 15 down again and *__siftDownFromTo* takes care of moving the 15 to the bottom of the heap as shown in Fig. 9.22.

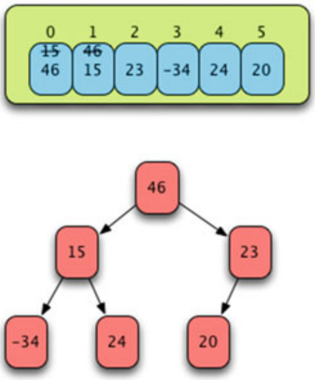

**Fig. 9.21** Sifting the 15 Down

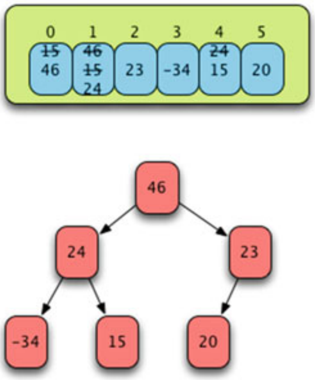

**Fig. 9.22** The Final Heap using Version 2 of Phase I

## 9.9 Analysis of Heapsort Version 2

Recall that Phase II is when the values are in a heap and extracted one at a time to form the sorted list. Version 2 Phase II of the heapsort algorithm is identical to version 1 and has the same complexity, O(N log N).

Version 2 Phase I on the other hand has changed from a top down approach to building the heap in version 1 to building the heap from the bottom up in version 2. We claimed that the complexity of this new phase I is O(N) where *N* is the number of nodes in the list. Stated more formally we have this claim. *For a perfect binary tree of height h, containing* $(2^h-1)$ *nodes, the sums of the lengths of its maximum comparison paths is* $(2^h - 1-\text{h})$.

Consider binary heaps of heights 1, 2, etc. up to height *h*. From the example for version 2 of the algorithm it should be clear the maximum path length for any call to *__siftDownFromTo* will be determined as shown in Table 9.3).

Notice that $(2^{h-1})$ represents half the nodes in the final heap (the leaf nodes) and that the max path length for half the nodes in the heap will be 0. It is this observation that leads to a more efficient algorithm for building a heap from the bottom up. If we could add up all these maximum path lengths, then we would have an upper bound for the amount of work to be done during phase I of version 2 of this algorithm.

$$S = 1 * (h-1) + 2 * (h-2) + 2^2 * (h-3) + \cdots + 2^{h-3} * 2 + 2^{h-2} * 1$$

The value *S* would be an upper bound of the work to be done, the sum of the maximum path lengths. We can eliminate most of the terms in this sum with a little manipulation of the formula. The value of S could be computed as 2S − S = S. Using this formula we can write it as

$$\begin{aligned} S = 2 * S - S = \;& 2 * (h-1) + 2^2 * (h-2) + \cdots + 2^{h-2} * 2 + 2^{h-1} * 1 \\ & - [(h-1) + 2 * (h-2) + 2^2 * (h-3) + \cdots + 2^{h-2} * 1] \end{aligned}$$

If we line up the terms in the equation above (as they are lined up right now), we can subtract like terms. In the first like term we see h − 1 − (h − 2). This simplifies to

**Table 9.3** Maximum path length for __siftDownFromTo

| Level | Max path length | # of nodes at level |
|---|---|---|
| 1 | $h-1$ | 1 |
| 2 | $h-2$ | 2 |
| 3 | $h-3$ | 4 |
| ... | ... | ... |
| $h-2$ | 2 | $2^{h-3}$ |
| $h-1$ | 1 | $2^{h-2}$ |
| $h$ | 0 | $2^{h-1}$ |

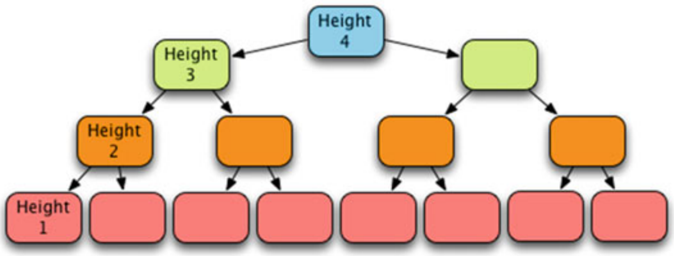

**Fig. 9.23** A Binary Heap of Height 4

h − h − 1 + 2 = 1. Similarly, the other like terms simplify so we end up with the following formula for S.

$$\begin{aligned} S &= 2 * S - S = 2 + 2^2 + \cdots + 2^{h-2} + 2^{h-1} - (h-1) \\ &= 1 + 2 + 2^2 + \cdots + 2^{h-2} + 2^{h-1} - h \\ &= 2^h - 1 - h \equiv O(N) \textit{ where } N = 2^h - 1 \textit{ nodes}. \end{aligned}$$

In the last step of the simplification above we have the sum of the first h−1 powers of 2, also known as the sum of a geometric sequence. This sum is equal to 2 raised to the h power, minus one. This can be proven with a simple proof by induction. So, we have just proved that version 2 of phase I is O(N). Phase II is still O(N log N) so the overall complexity of heap sort is O(N log N).

Consider a binary heap of height 4 (Fig. 9.23).

In such a heap, using the sift down method the first sifting occurs at height 2 in the tree where we have four nodes that may travel down one level in the tree. At height 3 we have two nodes that may travel down two levels. Finally, the root node may travel down three levels. We have the following sum of maximum path lengths.

$$\begin{aligned} 1+1+1+1+2+2+3 &= \\ 11 &= \\ 2^4 - 1 - 4 &= \\ 2^h - 1 - h \end{aligned}$$

## 9.10 Comparison to Other Sorting Algorithms

The heapsort algorithm operates in O(N log N) time, the same complexity as the quicksort algorithm. A key difference is in the movement of individual values. In quicksort, values are always moved toward their final location. Heapsort moves values first to form a heap, then moves them again to arrive at their final location

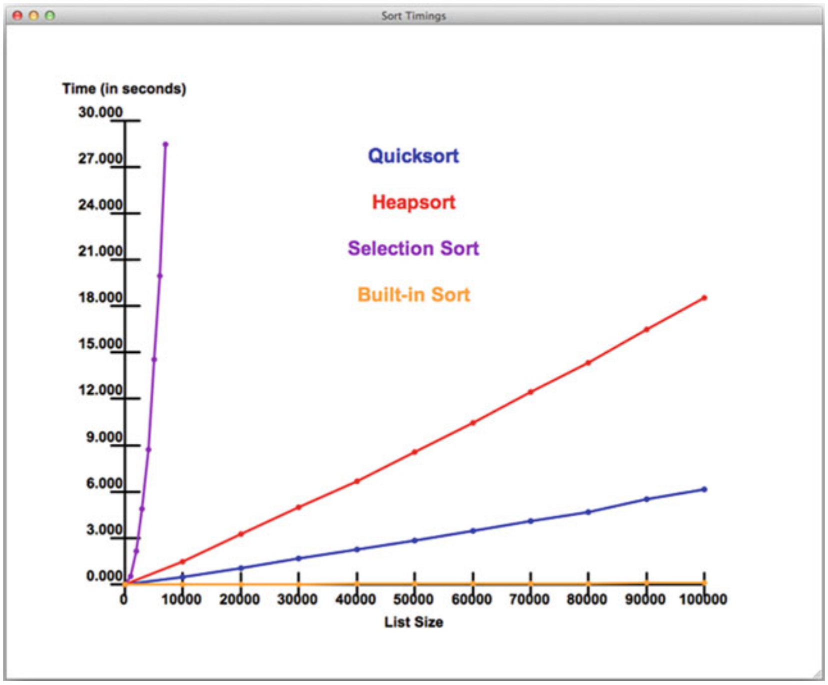

**Fig. 9.24** Comparison of Several Sorting Algorithms

within a sorted list. Quicksort is more efficient than heapsort even though they have the same computational complexity.

Examining Fig. 9.24 we see selection sort operating with $\theta(N^2)$ complexity, which is not acceptable except for very short lists. The quicksort algorithm behaves more favorably than the heapsort algorithm as is expected. The built-in sort, which is quicksort implemented in C, runs the fastest, due to being implemented in C.

## 9.11 Chapter Summary

This chapter introduced heaps and the heapsort algorithm. Building a heap can be done efficiently in O(N) time complexity. A heap guarantees the top element will be either the biggest or smallest element of an ordered collection of values. Using this principle we can implement many algorithms and datatypes using heaps. The heapsort algorithm was presented in this chapter as one example of a use for heaps.

Heaps are not good for looking up values. Looking up a value in a heap would take O(N) time and would be no better than linear search of a list for a value. This is because there is no ordering of the elements within a heap except that the largest (or smallest) value is on top. You cannot determine where in a heap a value is located without searching the entire heap, unless it happens to be equal or greater to the largest value and you have a largest on top heap. Likewise, if you have a smallest on top heap and are looking for a value, you would have to look at all values unless the value you are searching for is equal or smaller than the smallest value.

Commonly, heaps are used to implement priority queues where the elements of a queue are ordered according to some kind of priority value. An element can be added to an existing heap in O(log N) time. An element can be removed from a heap in O(log N) time as well. This makes a heap the logical choice for a priority queue implementation. Priority queues are useful in message passing frameworks and especially in some graph algorithms and heuristic search algorithms.

## 9.12 Review Questions

Answer these short answer, multiple choice, and true/false questions to test your mastery of the chapter.

1. State the heap property for a largest on top heap.
2. When removing a value from a heap, which value are you likely to remove? Why?
3. After removing a value from a heap, what steps do you have to take to ensure you still have a heap?
4. If you had a heap of height 6, what would be the total maximum travel distance for all nodes in the heap as you built it using version 2, phase I of the heapsort algorithm.
5. Use *__siftUpFrom()* from version 1 of the heapsort algorithm, adding a new element to a growing heap on each pass to construct a largest-on-top heap from the following integers:

   70, 30, 25, 90, 15, 85, 35, 87, 100

   Sketch a new picture of the binary heap each time the structure changes.
6. Use *__siftDownFromTo()* from version 2 of the heapsort algorithm on the same data as in the previous problem, sketching a new picture of the binary tree each time the structure changes.
7. Using the final heap from problem 6, execute phase II version 2 of the heapsort algorithm, using *__siftDownFromTo* to sort the data in increasing order. Sketch a new picture of the binary tree each time the structure changes.
8. Redo problems 6 and 7, this time showing the data in arrays (i.e. lists) with starting index 0, rather than drawing the tree structures. Show the new values of the structure after each pass. Use the following data:

   37, 45, 91, 5, 57, 74, 83, 45, 99

9. Why does heapsort operate less efficiently than quicksort?
10. When is a heap commonly used?

## 9.13 Programming Problems

1. Implement version 2 of the heapsort algorithm. Run your own tests using heapsort and quicksort to compare the execution time of the two sorting algorithms. Output your data in the plot format and plot your data using the PlotData.py program provided on the text website.
2. Implement version 1 and version 2 of the program and compare the execution times of the two heapsort variations. Gather experimental data in the XML format accepted by the PlotData.py program and plot that data to see the difference between using version 1 and version 2 of the heap sort algorithm.
3. Implement a smallest on top heap and use it in implementing a priority queue. A priority queue has enqueue and dequeue methods. When enqueueing an item on a priority queue, a priority is provided. Elements enqueued on the queue include both the data item and the priority. Write a test program to test your priority queue data structure.
4. Use the priority queue from the last exercise to implement Dijkstra's algorithm from Chap. 7. The priority queue implementation of Dijkstra's algorithm is more efficient. The priority of each element is the cost so far of each vertex added to the priority queue. By dequeueing from the priority queue we automatically get the next lowest cost vertex from the queue without searching, resulting in a $O(|V|\log|V|)$ complexity instead of $O(|V|^2)$.
5. Use the heapsort algorithm, either version 1 or version 2, to implement Kruskal's algorithm from Chap. 7. Use one of the sample graph XML files found on the text website as your input data to test your program.