CS 331 — Heaps

Mattox Beckman

1 Introduction

The queue data structure is useful for simulating a line or a buffer. It is very fair, whatever comes in first goes out first, and all elements have the same priority. But sometimes not all things are equal. Sometimes we want certain things to go to the front of the line, even if they got in line later than the rest of the elements. In this situation, we want to give every element a *priority*. Higher priority items will move ahead of lower priority items.

Such a data structure is called a *priority queue*. There are many implementations of it. The one we will discuss is called a *binary heap*, though we will call them *heaps* from now on.

2 Tree Representation

We implement a heap using a *complete binary tree*. A complete binary tree is perfectly balanced. All the levels are full, except the lowest level; and that one is filled from left to right. Figure 1 shows an example tree.

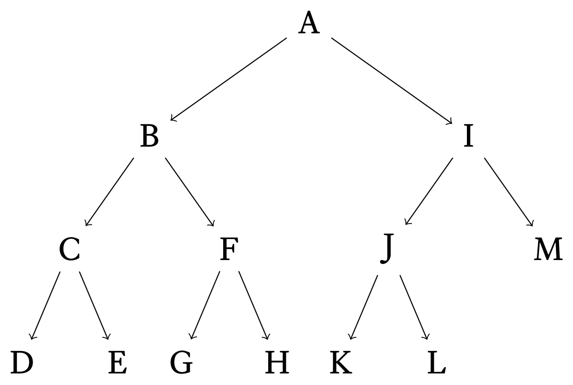


Figure 1: A complete tree. Notice that the bottom level does not have to be completely filled.

A heap tree has one other property. Every node must be of higher priority than the children. In this handout, we will use *min-heaps*. The lower the number, the higher the priority. Figure 2 gives two example heaps. Note that we do not care about the relative priority of the two children!

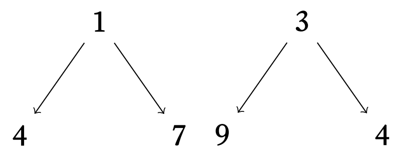


Figure 2: Two example heaps.

Question 1: Consider the heaps in figure 3. Which are min heaps, and which are not?

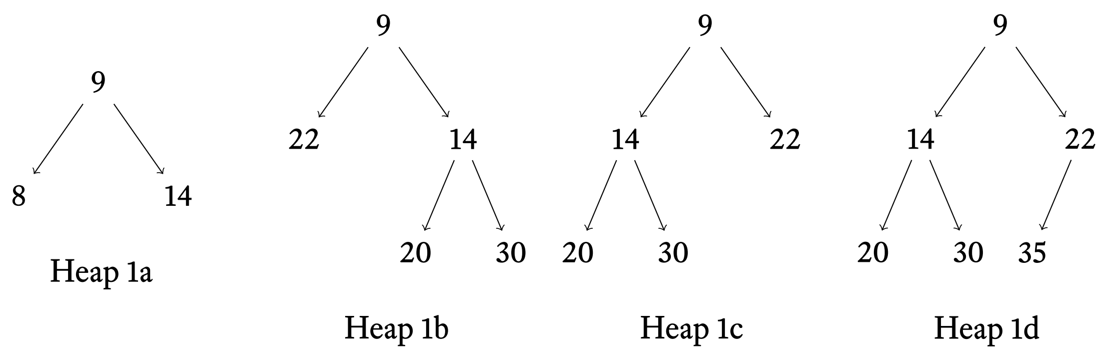


Figure 3: Heaps for question 1

2.1 Adding an element

To add an element to a heap, we first place the item in the next available position. Since the heap is a complete balanced tree, there is only one spot. In figure 1, the next spot would be the right child of M. In the first heap of figure 2, the node would be a right child of 4.

Once we've added the node, we keep swapping the node with the parent until the heap property is restored. Suppose we add a 3 to the first heap in figure 2. The parent would be 4, so we swap the two. The new parent is 1, so we are done. If we add a 2 to the second heap, the item will be the child of 9, and will be swapped all the way to the root. Figure 4 shows the result.

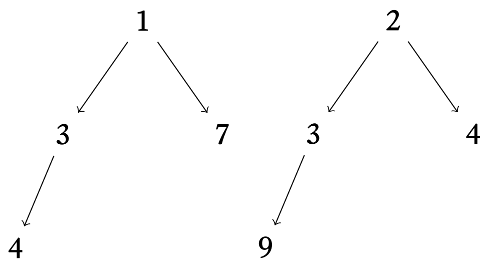


Figure 4: Two example heaps after an add.

Question 2: Now try adding a 2 to the first heap in figure 4 and a 6 to the second one.

The time complexity is easy to work out. We add at the bottom of the tree, which is guaranteed to be balanced, and could swap the item all the way to the root. So the time complexity is $\mathcal{O}(\lg n)$.

2.2 Deleting an element

Removing an element is also very constrained. There is only one element that can be removed: the root.[1] To remove the root, you take the *last* element of the heap (element L in figure 1 and 7 and 4 in figure 2) and overwrite the root with it.

After that, you have to do an operation called *percolate down*. If either of the children has higher priority than the item, you swap with the child of highest priority. Consider figure 5. Try to guess what will happen if we run delete twice on it.

The first delete moves the 16 to the root. It will swap with 5 and then with 15.

The next deletion will remove the 5. The 18 will swap with the 9 and the 10.

[1]Frequently the exam will have you remove an element from an example heap. Sometimes I get a student asking me which element to delete. I can tell already what they are going to score on that one.

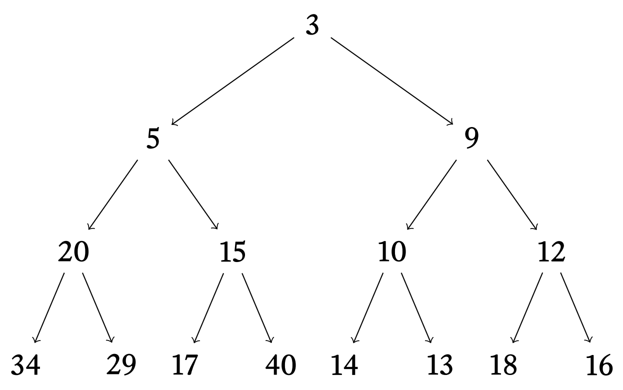


Figure 5: A heap before deleting.

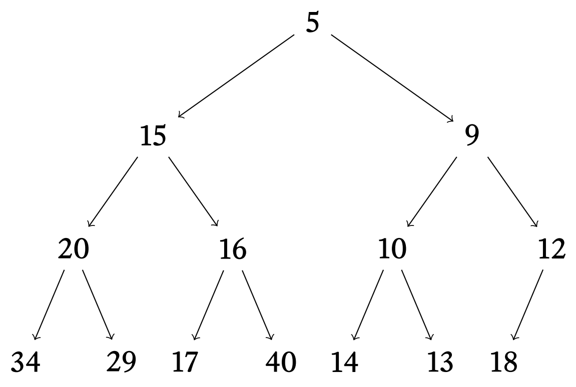


Figure 6: Figure 5 after one deletion.

Notice how the swapping will take different paths through the heap, depending on the values of the elements.

3 Using Arrays

Trees are difficult to use for heaps, because the path will need to be calculated each time. It is much easier to use an array. Consider figure 8: it shows the index each item would have in an array. The array version is in figure 9.

The formulae to relate these are simple.

$$\begin{array}{lcl} parent(i) & = & \lfloor \frac{i-1}{2} \rfloor \\ left(i) & = & 2i+1 \\ right(i) & = & 2i+2 \end{array}$$

Question 3: What is the time complexity of adding an element to a heap?

Question 4: What is the time complexity of deleting an element from a heap?

Question 5: Why do we use arrays instead of trees to store heap data?

Question 6: Given the following heap, show the result of adding element 1 to it.

0	1	2	3	4	5	6	7
2	3	5	8	12	10		

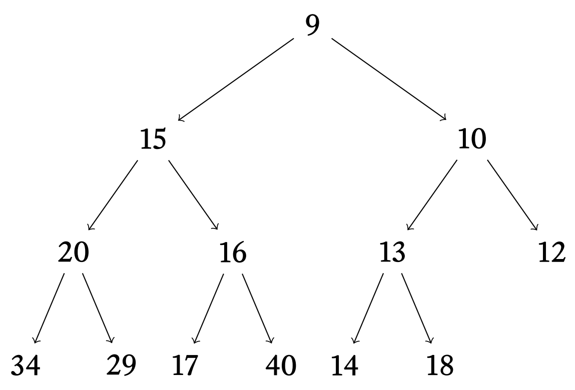


Figure 7: Figure 5 after two deletions.

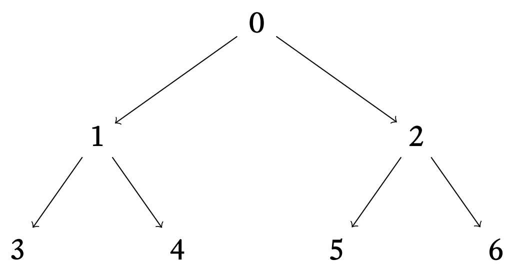


Figure 8: A heap with array indices shown.

Question 7: Given the following heap, show the result of adding element 4 to it.

0	1	2	3	4	5	6	7
2	3	5	8	12	10		

Question 8: Given the following heap, show the result of deleting two elements from it.

0	1	2	3	4	5	6	7
2	3	5	8	12	10		

4 Heap Sort

There is a nice sorting algorithm you can use if you have heaps. It's called *heap sort*, as you might expect. It runs in $\mathcal{O}(n \lg n)$ time.

The basic operation is to use a heap and pull out every element. It will give you the low elements first if you use a min heap, and you can store them in another array. If you use a max-heap, then you can store the elements in the same array, placing the just deleted element into the last space of the array.

Figure 10 shows an example of a heap being sorted by successively deleting the elements and placing them at the end of the array.

The trick with heap-sort is in converting an arbitrary array into a heap in the first place. Fortunately, this is very fast. The operation is called *heapify*. It works by starting at the leaves and building larger heaps until you reach the root.

Consider the "heap" in figure 11. It is the tree representation, and it does not yet have the heap property.

0	1	2	3	4	5	6

Figure 9: The array version.

50	40	25	35	37	32	10
40	37	25	35	10	32	50
37	35	25	32	10	40	50
35	32	25	10	37	40	50
32	10	25	35	37	40	50
25	10	32	35	37	40	50
10	25	32	35	37	40	50

Figure 10: Heap-sorting an array using max-heaps.

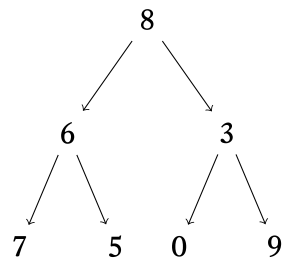


Figure 11: A random array in heap form.

Notice the leaves of this non-heap: 7,5,0, and 9. If we take them as individual trees, they are each heaps already. Now let's combine these leaf heaps with the next level. We can combine the 3 with the 0 and 9 using the percolate-down method. We can also combine the 6 with the 7 and 5. These combinations are $\mathcal{O}(\lg n)$ in the number of nodes in the sub-heaps. The result is in figure 12. We will use max-heaps here.

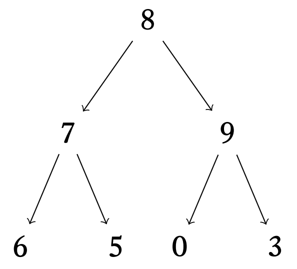


Figure 12: Heapify to the second level.

Now we have two heaps, and a root node. Again, combining them with percolate-down, we have the whole array as a max-heap, shown in figure 13.

$$\begin{tikzcd}
& & & 9 \arrow[dll] \arrow[drr] & & & \\
& 7 \arrow[dl] \arrow[dr] & & & & 8 \arrow[dl] \arrow[dr] & \\
6 & & 5 & & 0 & & 3
\end{tikzcd}$$

Figure 13: Heapify to the root.

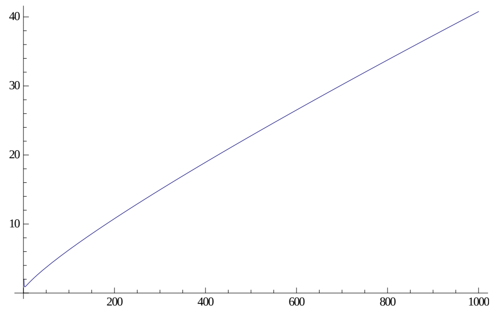
Why does this run in linear time? Half of the elements were heaps already, so we didn't have to do any work at all. Half of the remaining elements became roots to the leaf heaps. Those only needed to percolate down one level. The final node needed to percolate down two levels.

The first level (call it level one), the leaf level, will have half of the items in the array. Level two will have a forth, and so on.

So, the number of percolations looks something like this:

$$p(n) = \Sigma_{i=1}^{\lg n} i \times \frac{n}{2^i}$$

The graph of this function looks like this:



As you can see, the graph becomes linear, which means that we can convert an arbitrary array into a heap in $\mathcal{O}(n)$ time. After that, we can remove all the elements in $\mathcal{O}(n \lg n)$ time, meaning the sort has a $\mathcal{O}(n \lg n)$ running time (since the smaller $\mathcal{O}(n)$ term will drop out).

5 Solutions to exercises

Solution 1 Heap 1a is not a heap because the 8 in the bottom row is smaller than the root, 9. Heap 1b is not a heap because the tree is not filled from left to right. Heap 1c is a heap. Heap 1d is a heap.

Solution 2

$$\begin{tikzcd}
& & 1 \arrow[dl] \arrow[dr] & & & & 2 \arrow[dl] \arrow[dr] & \\
& 2 \arrow[dl] \arrow[dr] & & 7 & & 3 \arrow[dl] \arrow[dr] & & 4 \\
4 & & 3 & & 9 & & 6 &
\end{tikzcd}$$

Solution 3 $\mathcal{O}(\lg n)$

Solution 4 $\mathcal{O}(\lg n)$

Solution 5 It is faster and simpler to locate the next position of the heap using an array than it would be using a tree. Also, assuming the array is nearly full, it is more memory efficient.

Solution 6

0	1	2	3	4	5	6	7
1	3	2	8	12	10	5	

Solution 7

0	1	2	3	4	5	6	7
2	3	4	8	12	10	5	

Solution 8

0	1	2	3	4	5	6	7
5	8	12	10				