Lecture 22: The Heaps Data Structure

11:00 AM, Mar 12, 2021

Contents

Objectives

By the end of these notes, you will know:

- The definition of a heap
- How to insert and delete elements of a heap

1 Reviewing Data Structure Options for Alerts

In lecture, we discussed how well our existing data structures would support accessing security alerts according to their priorities. We said that alerts need to be inserted and deleted quickly as they arise and are processed (respectively).

In other words we need a data structure that has good performance on (1) inserting elements, (2) finding the element with a max (or min) value of interest (in this case the priority), and (3) removing the element with the max (min) priority. Finding the element with the max priority is of particular importance.

How do our data structures to date stack up?

- (unsorted) List: insertion is constant, but finding the max and deleting are both linear.
- Sorted List: finding the max is constant, but insertion and deletion are both linear
- Sorted Array: No more efficient than a list, due to the need to shift elements on insertions and deletions
- Binary Search Tree: insert and delete are both worst-case linear (if the tree is badly imbalanced); to have balance, the root element must be somewhere in the middle(ish), which means finding the max element will be logarithmic at best and linear at worst.

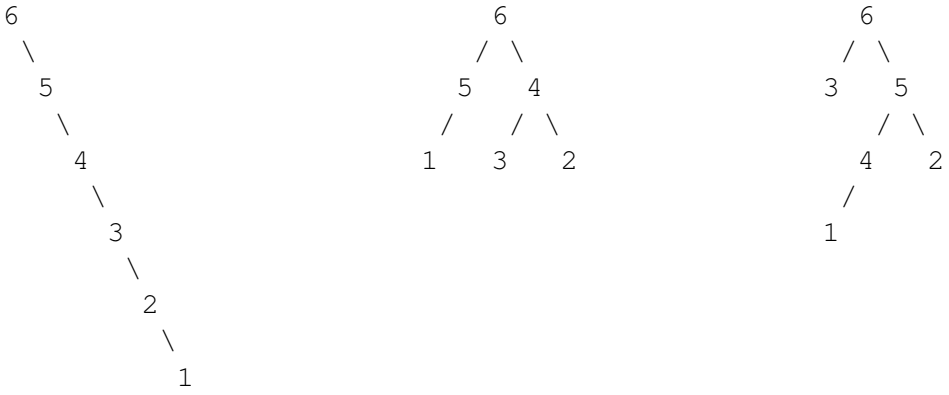
None of these seem ideal for maintaining priorty among items. Ideally, we want constant-time access to the max element and better-than-linear insertion and deletion.

2 Heaps: A New Data Structure

Heaps are form of binary tree that are designed for giving quick access to the max (or min) element in a set of elements.[1] Heaps are binary trees with the following invariant:

> *I*n a heap, the largest element in the tree is at the root, and the left and right subtrees are also heaps.

Here are three examples of max-heaps on the numbers 1 through 6:



What do you notice about these compared to binary search trees?

- They can also be unbalanced
- There is no apparent order of elements between the left and right subtrees
- Searching for an element appears to be linear, since we can't tell where a particular element might lie relative to the root.

So heaps are NOT binary *search* trees, even though both are built upon binary trees.

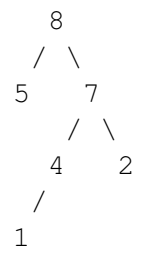
Relative to our operations of interest, they are efficient for getting the max element, since it sits at the root. How do they perform on insert and delete, the two operations that are linear in a sorted list?

Before we can answer that, we have to figure out how insert and delete should even work.

[1] The rest of these notes are written relative to prioritizing a max value. The case for prioritizing min is analogous.

2.1 Inserting into a Heap

Imagine that you had the following heap and wanted to insert 6 into it. What answer should you get?



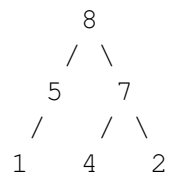
There are a lot of possible answers here: the 6 could become the left child of 8, with the 5 a child of the 6; the 6 could become a child of the 7, with the existing children pushed further down. Had we decided to insert a smaller number (such as 3) there could be even more options. Since the heap invariant is fairly permissive (other than "max at the root") there are many possible answers. We want an efficient algorithm, so we'd like to avoid restructuring the underlying tree more than necessary.

In the name of efficiency, we'd also like the underlying tree to be balanced as much as possible (meaning it isn't taller than necessary for the given number of elements). The heap above is taller than it needs to be (the 1 is in a depth by itself even though there is room under the 5).

Formally, we will say that a tree is *balanced* if at every node in the tree, the difference in depth (height) between the left and right subtrees differs by at most one. By that definition, only the middle tree in our three heap examples above is balanced.

So let's add a constraint to our insert algorithm: we want to assume that we start with a balanced binary tree (meaning it has minimal depth), then guarantee that we still have a balanced tree (and the heap invariant) after insertion is done (for today, we'll do this intuitively only).

For example, perhaps our initial heap was actually the following:

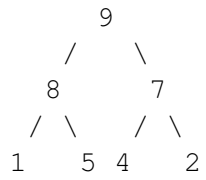


A balanced heap that contains 6 and restructures as little as possible would put the 6 where the 5 is and put the 5 as the right child of the 6. We can see that with our eyes, but how would we find that solution efficiently in an algorithm?

Sifting Up If we are trying to maintain balance in this case, there is only one spot that should get filled if we insert a new element (to the right of the 5 – any other position will add an unnecessary level to the tree). So let's put the 6 there temporarily and ask what needs to happen to restore the heap invariant.

Once the 6 is inserted, the invariant breaks at node 5 (since 6 is larger than 5). So we should swap 6 and 5 within the tree. This restores the invariant.

What if instead of inserting 6 we had inserted 9? Since 9 is larger than every other element, it should end up as the root of the heap when we are done. We would put the 9 in to the right of the 5. We would then move the 9 up, swapping it with its parent value until the parent is larger than the inserted value. in this case, we would swap 9 and 5, then we would swap 9 and 8. Here's the resulting heap:

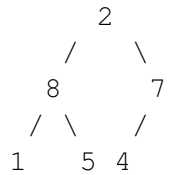


What if multiple leaf spots were open? If we had had two open spaces in the lowest level of the tree, which one should we use? Actually, we will avoid that problem by always inserting elements into the leftmost available spot in a level. If we followed this rule from the outset, then there would be no gaps within the nodes of the tree.

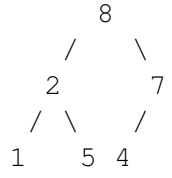
2.2 Deleting The Max Element From a Heap

Deletion follows a similar strategy of putting a value in a temporary position, then swapping parent/child nodes until the invariant is restored. In the case of deletion, we put the rightmost value from the lowest level up at the root (replacing the deleted value), then drop that element down the tree until it is larger than the nodes beneath it.

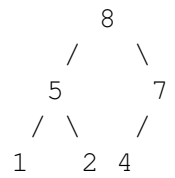
For example, imagine that we wanted to remove the 9 (the largest element) from our previous heap. We would first move the 2 to the root.



Note that we have now broken the heap invariant (since 2 is smaller than both 7 and 8). We swap 2 with the larger of 7 and 8 (the values at the children).



One more time, we swap the 2 and the 5 to restore the heap invariant:



This requires a log number of swaps to move the 2 back into place.

Are we guaranteed to maintain balance with this strategy? We don't add new levels, but we do remove a node from the lowest level. Can we construct an example in which the tree isn't properly balanced if the numbers fall just right?

We'll get back to this question, and the data structures that let us implement heaps safely, in the next lecture.

3 Study Questions

Think about these before our next class meeting. You don't need to submit your answers, but you should still write your answers down for your own learning benefit.

- Draw a heap that is NOT also a binary search tree (don't copy one from the notes, make a new one for yourself)
- Draw a heap that is also a binary search tree
- Imagine that we gave you a collection of numbers and told you to create a binary search tree for those numbers with a specific number at the root. How many different binary search trees could you make?
- Imagine that we gave you a collection of numbers and told you to create a heap from those numbers. Roughly how many heaps could you make? You don't need a numeric answer – a sense of what would determine the number of heaps you can make is fine.
- Can we make a heap of strings? What would decide "priority" in that case?
- Can we make a heap of objects from an arbitrary class? What would we need in order to do that?
- When we insert an element into the heap, we initially place it at a leaf and move it upwards into place. What is the benefit of doing that instead of starting from the top of the heap and finding a place to insert the node?

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.