CMSC 351: HeapSort

Justin Wyss-Gallifent

March 9, 2021

1 Heaps .................................................. 2
   1.1 Complete Binary Trees .................................. 2
   1.2 Max (Binary) Heap ....................................... 3
   1.3 Converting .............................................. 4
2 Relation to Sorting .............................. 8
   2.1 Heapsort ............................................. 8
   2.2 Pseudocode .......................................... 13
   2.3 Worst-Case Time Complexity .......................... 14
   2.4 Best-Case Time Complexity .......................... 16
3 Auxiliary Space ............................................. 16
4 Stability .................................................. 16
5 In-Place .................................................. 16
6 Usage Note ............................................... 16
7 Thoughts, Problems, Ideas .................. 17
8 Python Test ............................................... 19
1 Heaps

1.1 Complete Binary Trees

A complete binary tree is a binary tree in which all levels are completely filled, except possibly for the last level, and the last level has all entries as far left as possible.

Example 1.1. Here is an example:

```
          1
         /\  
         2 / 3
         / / 
        4 5 / 
        / / 6
       8 9 7 11 12
```

Observe that we begin the node indexing at the root node at $i = 1$ instead of $0$ as is usual for arrays and lists. The reason for this is that there are some convenient node calculations which emerge. Note that in the literature this is by no means uniform.

Observations that we’ll find useful:

- For a node $i$ its left and right children (if it has them) are nodes $2i$ and $2i+1$ respectively.
- For a node $i$ its parent is node $\lfloor i/2 \rfloor$.
- As a special case of the above, if there are $n$ nodes total then the largest node with children is node $\lfloor n/2 \rfloor$.
- The leftmost node at level $k$ (with level $k = 1$ being the root) is node $2^{k-1}$.
- A node $i$ is located in level $1 + \lfloor \lg(i) \rfloor$. In the example above node $i = 6$ is located in level $1 + \lfloor \lg(6) \rfloor = 1 + \lfloor 2.58 \rfloor = 3$.
- As a special case of the above, if there are $n$ nodes total then the maximum level equals $1 + \lfloor \lg(n) \rfloor$. In the example above there are $n = 12$ nodes and $1 + \lfloor \lg(12) \rfloor = 1 + \lfloor 3.58 \rfloor = 4$.
- Since a node $i$ can only have child nodes $2i$ and $2i+1$ it follows that if a node $i$ has any children at all then we must have node $2i$. It then follows that if there are $n$ nodes total then the only nodes with children must satisfy $2i \leq n$ and so $i \leq 0.5n$. In other words the only nodes that could have children are those nodes $i = 1, 2, ..., \lfloor 0.5n \rfloor$. 
1.2 Max (Binary) Heap

A max heap (we’ll omit the word binary since all our trees will be binary) is a complete binary tree in which each node’s value is greater than or equal to that node’s children’s value if that node has children. In other words values non-strictly decrease (equality is acceptable) as we go down the branches.

Example 1.2. The example above is not a max heap. The nodes marked in red below violate the requirement because they have values which are less than at least one of their children’s values:

```
Example 1.3. The following is a max heap, however:
```

```
1.3 Converting

We have the following:

1. Introduction: Given a complete binary tree, it’s possible to rearrange the nodes so as to obtain a max heap. To do this we’ll need the processes:

2. Max Heapify - Floating Down. The `maxheapify` function is an atrociously named function whose function isn’t reflected well in its name. A better name would be `floatvaluedown` because given a node $i$ it floats the associated value down (if necessary) until the value no longer violates the max-heap property.

In other words, suppose we have a binary tree with a node $i$ such that the subtrees rooted at the children of node $i$ (if they exist) are roots of max heaps. Suppose also that the value at node $i$ is smaller than one or both of its child node values.

**Example 1.4.** For example consider the red node (node 2) in the following tree. Note that the subtrees rooted at its children are max heaps:

![Diagram](image)

We can float this problematic value down to the bottom of the tree by repeatedly following the branch to the largest value. Here is the process. First we interchange the values at nodes 2 and 4:

![Diagram](image)

Then we interchange the values at nodes 4 and 9:

![Diagram](image)
Here is the pseudocode. Here it is assumed that \(A\) is a heap and \(A.\text{heapend}\) is a variable set to be the number of nodes in the heap to be considered. The conditionals \(\text{leftnode} \leq A.\text{heapend}\) and \(\text{rightnode} \leq A.\text{heapend}\) simply check for the existence of children of node \(i\) before checking the values residing there.

The property \(A.\text{heapend}\) is a variable which can be updated, and will be updated later for reasons that are not be clear right now. To start with think of it as representing the final node in the heap that we care about.

```plaintext
procedure maxheapify(A, i) aka floatvaluedown(A, i)
    leftnode = 2*i
    rightnode = 2*i+1
    largestnode = i
    if leftnode \leq A.\text{heapend} and A[leftnode] > A[largestnode]
        largestnode = leftnode
    end
    if rightnode \leq A.\text{heapend} and A[rightnode] > A[largestnode]
        largestnode = rightnode
    end
    if largestnode != i
        swap(A[i], A[largestnode])
        maxheapify(A, largestnode) aka floatvaluedown(A, largestnode)
    end
end
```

Observe that because the smaller value moves down along a path which results in larger values floating up, and because we never float a value up above a higher value, not only do the subtrees rooted at the child nodes remain max heaps but the subtree rooted at node \(i\) becomes a max heap.

3. Given a complete tree we then convert it to a max heap by repeating \text{maxheapify} starting at the very last non-leaf node (recall if there are \(n\) nodes then this would be \(\lfloor n/2 \rfloor\)) and working our way back to the first node. The reason we can omit the leaf-nodes is that they are max heaps trivially.

Here is the pseudocode. Here the function \text{nodecount} returns the number of nodes in \(A\).

```plaintext
procedure converttomaxheap(A)
    for i = \text{floor}(A.\text{heapend}/2) down to 1
        maxheapify(A, i) aka floatvaluedown(A, i)
    end
end
```
**Example 1.5.** Here is the process as applied to our original tree:

Here we have $\text{heapend}(A) = 12$ and so floor($\text{heapend}(A)/2$) = 6 and so we start with node 6. Running $\text{maxheapify}(A, 6)$ interchanges values at indices along the chain 6 ↔ 12 only:

Running $\text{maxheapify}(A, 5)$ interchanges values at indices along the chain 5 ↔ 11 only:

Running $\text{maxheapify}(A, 4)$ interchanges values at indices along the chain 4 ↔ 9 only:

Running $\text{maxheapify}(A, 3)$ interchanges values at indices along the chain 3 ↔ 6 ↔ 12 only:
Running \texttt{maxheapify}(A,2) interchanges values at indices along the chain $2 \leftrightarrow 5$ only:

Running \texttt{maxheapify}(A,1) interchanges values at indices along the chain $1 \leftrightarrow 3 \leftrightarrow 6$ only:

We can see that the result is now a max heap. The formal proof of this follows from the fact that running \texttt{maxheapify} on any particular node preserves the heap-ness of the child nodes and induces heap-ness on that node.
2 Relation to Sorting

2.1 Heapsort

A max binary heap is structured such that extracting the values in a sorted manner is very easy. There are several ways to do this, all are based on the observation that the largest value is at the root node so that value needs to be last in our sorted list. What we’ll do is exchange it with the value in the final node in the tree and then ignore it from here on out, cutting it off from the tree structure.

Now then, the children of the new root node are still max heaps but the new root node will almost certainly violate the max heap property so we fix this by running maxheapify again on that node to return the remaining tree to a max heap.

We then repeat the process on the new tree and keep repeating until we’re done.

Example 2.1. Here is the process on our heap from earlier:

We start with:

```
  17
  1  
 15
  2
  7
  3
  9
  4
 13
  5
  6
  2
  7
  8
  8
  5
  9
  3
 11
 12
```

We interchange the values at nodes 1 and 12 and cut node 12 off from the tree:

```
  3
  1
 15
  2
  7
  3
  9
  4
  6
  5
  6
  2
  7
  8
  8
  5
  9
  3
 10
 11
```

We then run maxheapify(A, 1) but only on the subtree:

```
  15
  2
  13
  4
  9
  9
  6
  6
  10
  11
  8
  5
  3
  1
  17
```

We interchange the values at nodes 1 and 11 and cut node 11 off from the tree:
We then run `maxheapify(A,1)` but only on the subtree:

We interchange the values at nodes 1 and 10 and cut node 10 off from the tree:

We then run `maxheapify(A,1)` but only on the subtree:

We interchange the values at nodes 1 and 9 and cut node 9 off from the tree:

We then run `maxheapify(A,1)` but only on the subtree:
We interchange the values at nodes 1 and 8 and cut node 8 off from the tree:

We then run `maxheapify(A,1)` but only on the subtree:

We interchange the values at nodes 1 and 7 and cut node 7 off from the tree:

We then run `maxheapify(A,1)` but only on the subtree:

We interchange the values at nodes 1 and 6 and cut node 6 off from the tree:
We then run `maxheapify(A,1)` but only on the subtree:

We interchange the values at nodes 1 and 5 and cut node 5 off from the tree:

We then run `maxheapify(A,1)` but only on the subtree:

We interchange the values at nodes 1 and 4 and cut node 4 off from the tree:

We then run `maxheapify(A,1)` but only on the subtree:
We interchange the values at nodes 1 and 3 and cut node 3 off from the tree:

We then run `maxheapify(A, 1)` but only on the subtree:

We interchange the values at nodes 1 and 2 and cut node 2 off from the tree:

At this point we’re done and we simply extract the values by node:

```
1, 2, 3, 5, 6, 6, 7, 8, 9, 13, 15, 17
```
2.2 Pseudocode

The pseudocode is then as follows:

```plaintext
procedure heapsort(A)
    // Start by setting the heapend equal to the actual
    // number of nodes.
    A.heapend = nodecount(A)
    converttomaxheap(A)
    for i = nodecount(A) down to 2
        swap(A[1],A[i])
        // Reduce the heapend so we're ignoring one more node
        // when we maxheapify again.
        A.heapend = A.heapend - 1
        maxheapify(A,1) aka floatvaluedown(A,1)
    end
end

And for reference from earlier:

```plaintext
procedure maxheapify(A,i) aka floatvaluedown(A,i)
    leftnode = 2*i
    rightnode = 2*i+1
    largestnode = i
        largestnode = leftnode
    end
        largestnode = rightnode
    end
    if largestnode != i
        swap(A[i],A[largestnode])
        maxheapify(A,largestnode) aka floatvaluedown(A,largestnode)
    end
end

procedure converttomaxheap(A)
    for i = floor(A.heapend/2) down to 1
        maxheapify(A,i) aka floatvaluedown(A,i)
    end
end
```
2.3 Worst-Case Time Complexity

Consider:

- **maxheapify** aka **floatvaluedown**: Consider the node \( i \) in a tree with \( k \) nodes. Denote by \( T_{mh}(i, k) \) the amount of time required to run **maxheapify** aka **floatvaluedown** on node \( i \) if \( A.\text{heapmax} \) equals \( k \).

There is constant time \( c_1 \) required to do any necessary checking and swapping.

The swap (happens in the worst-case!) will result in a recursive call being made and this will repeat all the way until the bottom, at which point there is a check but no swap and no recursive call.

A node \( i \) is in level \( 1 + \lfloor \lg i \rfloor \) of the tree and there are \( 1 + \lfloor \lg k \rfloor \) levels in the tree (this is one more than the height). There are therefore \( \lfloor \lg k \rfloor - \lfloor \lg i \rfloor + 1 \) nodes to work through, the final one being the leaf.

The total time for all recursive calls then satisfies:

\[
T_{mh}(i, k) \leq c_1(\lfloor \lg k \rfloor - \lfloor \lg i \rfloor + 1)
\]

This is not very pretty to work with so we can tidy it up via worst-case observations:

\[
T_{mh}(i, k) \leq c_1(\lfloor \lg k \rfloor - \lfloor \lg i \rfloor + 1) \leq c_1(1 + \lg k) = c_1 + c_1 \lg k
\]

We can then say that for a tree with \( k \) nodes that for the node \( i \) that we have the following, including all recursive calls:

\[
T_{mh}(i, k) \leq c_1 + c_1 \lg k
\]

Thus:

\[
T_{mh}(i, k) = \mathcal{O}(\lg k)
\]

- **converttomaxheap**: Denote by \( T_{ctmh}(n) \) the amount of time required to run **converttomaxheap**.

The function **converttomaxheap** iterates **maxheapify** with \( i = \lfloor n/2 \rfloor, \ldots, 1 \) on the original tree with \( n \) nodes, thus yielding a total time requirement of:
\[ T_{ctmh}(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} T_{nh}(i, n) \leq \sum_{i=1}^{\lfloor n/2 \rfloor} [c_1 + c_1 \lg n] \leq [n/2] (c_1 + c_1 \lg n) \leq \frac{1}{2} (c_1 n + c_1 n \lg n) \]

Thus:

\[ T_{ctmh}(n) = \mathcal{O}(n \lg n) \]

**heapsort**: Here we run `converttomaxheap` and then for each \( i = n, n - 1, \ldots, 2 \) we swap elements and decrease \( A \).`heapend`, these two together take \( c_2 \) time. We also run `maxheapify` again, this time on a subtree with \( i - 1 \) nodes.

In total then:

\[ T(n) = T_{ctnh}(n) + \sum_{i=2}^{n} [c_2 + T_{nh}(1, i - 1)] \]

\[ \leq \frac{1}{2} c_1 n + \frac{1}{2} c_1 n \lg n + \sum_{i=2}^{n} [c_2 + c_1 + c_1 \lg(i - 1)] \]

\[ \leq \frac{1}{2} c_1 n + \frac{1}{2} c_1 n \lg n + \sum_{i=2}^{n} [c_2 + c_1 + c_1 \lg n] \]

\[ \leq \frac{1}{2} c_1 n + \frac{1}{2} c_1 n \lg n + (n - 1) (c_2 + c_1 + c_1 \lg n) \]

\[ \leq \frac{1}{2} c_1 n + \frac{1}{2} c_1 n \lg n + c_2 n + c_1 n + c_1 n \lg n - c_1 - c_2 - c_1 \lg n \]

\[ \leq \frac{3}{2} c_1 n \lg n + \left( \frac{3}{2} c_1 + \frac{1}{2} c_2 \right) n - c_1 \lg n - c_1 - c_2 \]

Thus we have \( T(n) = \mathcal{O}(n \lg n) \).
2.4 Best-Case Time Complexity

A few notes related to best-case time complexity:

1. If we start with a heap containing all identical elements then `maxheapify` does no swaps and therefore makes no recursive calls and therefore takes constant time, hence time complexity \( \Theta(1) \), instead of the worst-case logarithmic.

   As a consequence `converttomaxheap` will have \( \Theta(n) \) time complexity because it runs `maxheapify` on \( \lceil n/2 \rceil \) nodes.

   The calls that `heapsort` makes to `maxheapify` will themselves take constant time as well (no swaps) and so it follows that `heapsort` will have complexity:

   \[
   \Theta(n) + \sum_{n=2}^{n} (c_2 + \Theta(1)) = \Theta(n)
   \]

2. If we start with a heap which is already a max heap (but not all identical elements) then again `maxheapify` does no swaps and so `converttomaxheap` will have \( \Theta(n) \) time complexity. However the calls that `heapsort` makes to `maxheapify` will result in swaps because the smaller elements are getting swapped into the root node as `heapsort` progresses and consequently the final time complexity will still be \( \Theta(n \lg n) \).

3 Auxiliary Space

HeapSort uses \( \Theta(1) \) auxiliary space.

4 Stability

HeapSort is unstable.

5 In-Place

HeapSort is in-place.

6 Usage Note

HeapSort itself is rarely used as a general sorting algorithm because something like QuickSort is better. However max heaps are used frequently for such things as priority queues and scheduling. The reason for this is that the process of insertion and deletion is \( \Theta(\lg n) \) on a max heap versus \( \Theta(n) \) on a list and so max heaps are useful whenever these processes are critical.
7 Thoughts, Problems, Ideas

1. Consider the following complete binary tree.

![Complete Binary Tree Diagram]

(a) Which nodes violate the max heap property?
(b) Show the results of applying `converttomaxheap`. You do not need to show each step of each `maxheapify` but show the tree after each iteration of `maxheapify` executes.

2. Comparison of running times:

(a) If \( A = \{1, 2, 3, 4, 5, 6, 7\} \) is treated as a complete binary tree. If `maxheapify` takes 1 second to interchange the values at two nodes how long will it take to run `heapsort`? Assume everything else takes zero time.
(b) If \( A = \{7, 6, 5, 4, 3, 2, 1\} \) is treated as an array and if it takes 1 second to swap two entries how long will standard BubbleSort take to sort the array? Assume everything else takes zero time.

3. Prove that \( \lfloor \lfloor x/2 \rfloor /2 \rfloor = \lfloor x/4 \rfloor \).

4. The standard way to add an element to a max heap is to add it at the end (the \( n + 1 \) position) and then run `maxheapify` on all the required nodes. As a function of \( n \), which nodes is this? What is the time complexity of this process?

5. Suppose node \( i \) is removed from a max heap. We can’t just remove it because we will no longer have a tree. Instead the standard approach is to swap it with the ending node, delete the ending node, and then run `maxheapify` to clean up node \( i \). On which nodes will this be necessary and under which conditions? What is the time complexity of this process?

6. Qualitatively speaking why might InsertSort be faster than HeapSort for smaller lists?

7. Consider the following complete binary tree:
Suppose you forget to convert to maxheap in your heapsort function. What will the result be? Would you consider the result sorted, unsorted, or something in between?

8. Given an array \( A \) indexed at 1, describe a process by which we could determine whether or not the array represents a max heap. Write the pseudocode for an algorithm which does this. What is the time complexity of this process?

9. Describe how you could find the \( k \)th largest element in a max heap. Write the pseudocode for an algorithm which does this. What is the time complexity of this process?

10. Modify the various algorithms for the min-heap case.

11. Modify the various algorithms assuming the heap is indexed starting at 0 rather than 1.

12. Provide a formal mathematical proof of the following:
   Suppose \( T \) is a complete binary tree with the property that the subtrees of the root node are themselves max heaps. Prove that running maxheapify on the root node results in a max heap overall.
8 Python Test

Code:

```python
# In order to work with the Python array as tree nodes
# starting at 1,
# We create a list A[0,...,n] and ignore the 0th entry.
import random
import math
A = []
for i in range(0,10):
    A.append(random.randint(0,100))
heapsize = len(A)-1;
nodecount = len(A)-1
def maxheapify(i):
    leftnode = 2*i
    rightnode = 2*i+1
    largestnode = i
    if leftnode <= heapsize and A[leftnode] > A[largestnode]:
        largestnode = leftnode
    if rightnode <= heapsize and A[rightnode] > A[largestnode]:
        largestnode = rightnode
    if largestnode != i:
        temp = A[i]
        A[largestnode] = temp
        maxheapify(largestnode)
def converttomaxheap():
    for i in range(math.floor(heapsize/2),0,-1):
        maxheapify(i)
def heapsort():
    global heapsize
call converttomaxheap()
    print('After converttomaxheap:')
    print(A[1:])
    for i in range(nodecount,1,-1):
        temp = A[1]
        A[i] = temp
        print('After switch:')
        print(A[1:])
        heapsize = heapsize - 1
        maxheapify(1)
        print('After maxheapify:')
        print(A[1:])
    print(A[1:])
heapsort()
print(A[1:])
```
Output:

```
[74, 60, 82, 5, 8, 7, 90, 31, 50]
After converttomaxheap:
[90, 60, 82, 50, 8, 7, 74, 31, 5]
After switch:
[5, 60, 82, 50, 8, 7, 74, 31, 90]
After maxheapify:
[82, 60, 74, 50, 8, 7, 5, 31, 90]
After switch:
[31, 60, 74, 50, 8, 7, 5, 82, 90]
After maxheapify:
[74, 60, 31, 50, 8, 7, 5, 82, 90]
After switch:
[5, 60, 31, 50, 8, 7, 74, 82, 90]
After maxheapify:
[60, 50, 31, 5, 8, 7, 74, 82, 90]
After switch:
[7, 50, 31, 5, 8, 60, 74, 82, 90]
After maxheapify:
[50, 8, 31, 5, 7, 60, 74, 82, 90]
After switch:
[7, 8, 31, 5, 50, 60, 74, 82, 90]
After maxheapify:
[31, 8, 7, 5, 50, 60, 74, 82, 90]
After switch:
[5, 8, 7, 31, 50, 60, 74, 82, 90]
After maxheapify:
[8, 5, 7, 31, 50, 60, 74, 82, 90]
After switch:
[7, 5, 8, 31, 50, 60, 74, 82, 90]
After maxheapify:
[7, 5, 8, 31, 50, 60, 74, 82, 90]
After switch:
[5, 7, 8, 31, 50, 60, 74, 82, 90]
After maxheapify:
[5, 7, 8, 31, 50, 60, 74, 82, 90]
```