1 Heaps

1.1 Heaps; Complete Binary Trees

A heap is a complete binary tree. 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:

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 about indices that we’ll find useful:

- If a node has index \( i \) then its left and right children (if it has them) have indices \( 2i \) and \( 2i + 1 \) respectively.
  Example: In the above example the node with index 5 has children with indices 10 and 11.

- If a node has even index \( i \) then its parent has index \( i/2 \) and if a node has odd index \( i \) then its parent has index \((i - 1)/2\). We can combine these and say that if a node has index \( i \) then its parent has index \([i/2]\).
  Example: In the above example the node with index 7 has parent with index \([7/2] = [3.5] = 3\).

- As a special case of the above, if there are \( n \) nodes total then the largest node with children is the node with index \([n/2]\).
  Example: In the above example there are \( n = 12 \) nodes and the largest one with children is the node with index \([12/2] = 6\).

- If a node with index \( i \) has children than all nodes with smaller indices also have children.
  Example: In the above example the node with index \( i = 4 \) has children and then so do the nodes within indices 1, 2, and 3.
• Combining the previous two items tell us that if there are $n$ nodes total then the nodes with indices $1, 2, \ldots, \lfloor n/2 \rfloor$ are the ones that have children.

**Example:** In the above example the nodes with indices $1, 2, \ldots, 6$ are the ones that have children.

Observations about levels that we’ll find useful:

• The leftmost node at level $k$ (with level $k = 1$ being the level of the root) is the node with index $2^{k-1}$.

  **Example:** In the above example the leftmost node at level $3$ is the node with index $2^2 = 4$.

• A node with index $i$ is located in level $1 + \lceil \lg i \rceil$.

  **Example:** In the above example the node with index $6$ is located in level $1 + \lceil \lg 6 \rceil = 1 + \lg 6 = 3$.

• As a special case of the above, if there are $n$ nodes total then the maximum level (the leaf level) equals $1 + \lceil \lg n \rceil$.

  **Example:** In the above example there are $n = 12$ nodes and $1 + \lceil \lg 12 \rceil = 1 + \lceil 3.58 \rceil = 4$ levels.

• The number of levels between a node with index $i$ and the leaf layer, inclusive, is then $(1 + \lceil \lg n \rceil) - (1 + \lceil \lg i \rceil) + 1 = \lceil \lg n \rceil - \lceil \lg i \rceil + 1$.

  **Example:** In the above example the number of levels between the node with index $3$ and the leaf level, inclusive, is $\lceil \lg 12 \rceil - \lceil \lg 3 \rceil + 1 = 4$.

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:

![Max Heap Diagram]

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(TREE, index) function is an atrociously named function whose function isn’t reflected well in its name. A better name would be floatvaluedown(TREE, index).

The maxheapify(TREE, index) function is only ever called on a node with index \( i \) whose children are each already roots of their own max heaps. It floats the value at index \( i \) down the tree as far as necessary to ensure that the subtree rooted at index \( i \) is also a max heap.

It does this by asking “Is my value smaller than either of my children’s values?” If not, then we’re done. If so, then the value is swapped with the value of the largest child and then maxheapify is called again on that child.

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

![Max Heap Diagram with Red Node]
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.

We call $\text{maxheapify}(\text{TREE}, 2)$.

We observe that the value with index 2 is smaller than the value at index 4 and so we interchange the values with indices 2 and 4:

Now we call $\text{maxheapify}(\text{TREE}, 4)$.

We observe that the value with index 4 is smaller than the value at index 9 and so we interchange the values with indices 4 and 9:

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 the node with index $i$ 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.

**Example 1.5.** Here is the process as applied to our original tree:
Here we have \( \text{maxindex}(A) = 12 \) and so \( \text{floor}(\text{maxindex}(A)/2) = 6 \) and so we start with the node with index 6 (the last node with children). Running \( \text{maxheapify}(\text{TREE, 6}) \) interchanges values at indices along the chain \( 6 \leftrightarrow 12 \) only:

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

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

Running \( \text{maxheapify}(\text{TREE, 3}) \) interchanges values at indices along the chain \( 3 \leftrightarrow 6 \leftrightarrow 12 \) only:

Running \( \text{maxheapify}(\text{TREE, 2}) \) interchanges values at indices along the chain \( 2 \leftrightarrow 5 \) only:
Running `maxheapify(TREE, 1)` interchanges values at indices along the chain 1 ↔ 3 ↔ 6 only:

![Diagram of a max heap]

We can see that the result is now a max heap. The formal proof of this follows from the fact that running `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 (index 1) will almost certainly violate the max heap property so we fix this by running `maxheapify` again on the node with index 1 to fix the remaining tree back 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:
We interchange the values at the nodes with indices 1 and 12 and cut the node with index 12 off from the tree:

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

We interchange the values at the nodes with indices 1 and 11 and cut the node with index 11 off from the tree:

We then run `maxheapify(TREE, 1)` but only on the subtree:
We interchange the values at the nodes with indices 1 and 10 and cut the node with index 10 off from the tree:

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

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

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

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

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

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

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

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

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

We interchange the values at the nodes with indices 1 and 6 and cut node with index 6 off from the tree:

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

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

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

We interchange the values at the nodes with indices 1 and 6 and cut node with index 6 off from the tree:

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

We then run `maxheapify(TREE,1)` but only on the subtree:
We interchange the values at the nodes with indices 1 and 5 and cut the node with index 5 off from the tree:

We then run \texttt{maxheapify(TREE,1)} but only on the subtree:

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

We then run \texttt{maxheapify(TREE,1)} but only on the subtree:

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

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

At this point we’re done and we simply extract the values by the indices of the nodes:

\[1, 2, 3, 5, 6, 6, 7, 8, 9, 13, 15, 17\]

\[\blacksquare\]

### 2.2 Heapsort Worst-Case Time Complexity

Consider:

- **`maxheapify` aka `floatvaluedown`**: Denote by \( T_{mh}(i, n) \) the amount of time required to run `maxheapify` aka `floatvaluedown` on the node with index \( i \) as well as a chain of nodes below it due to all the potential recursive calls.

In a worst-case situation `maxheapify` is called repeatedly until the value at index \( i \) is moved all the way to the bottom of the tree. If each check (and perhaps swap) takes \( c_1 \) constant time then since there are (as we have seen) \( \lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1 \) layers from the layer containing the node with index \( i \) to the leaf layer, the amount of time it takes `maxheapify` and all its recursive calls to run is then

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

This is not very pretty to work with so we can tidy it up via worst-case observations:
$$T_{\text{mh}}(i, n) = c_1 (\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1) \leq c_1 (1 + \lg n) = c_1 + c_1 \lg n$$

We can then say that for a tree with $n$ nodes that for the node $i$ that we have the following, including all recursive calls:

$$T_{\text{mh}}(i, n) \leq c_1 + c_1 \lg n$$

As a side-note, observe:

$$T_{\text{mh}}(i, n) = \mathcal{O}(\lg n)$$

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

  The function `converttomaxheap` iterates `maxheapify` with $i = \lfloor n/2 \rfloor, \ldots, 1$ thus yielding a total time requirement of:

  $$T_{\text{ctmh}}(n) = \sum_{i=1}^{\lfloor n/2 \rfloor} T_{\text{mh}}(i, n)$$
  $$\leq \sum_{i=1}^{\lfloor n/2 \rfloor} (c_1 + c_1 \lg n)$$
  $$\leq \lfloor n/2 \rfloor (c_1 + c_1 \lg n)$$
  $$\leq \frac{1}{2}(c_1 n + c_1 n \lg n)$$

  In summary:

  $$T_{\text{ctmh}}(n) \leq \frac{1}{2}(c_1 n + c_1 n \lg n)$$

  As a side-note, observe::

  $$T_{\text{ctmh}}(n) = \mathcal{O}(n \lg n)$$

- **heapsort**: Here we run `converttomaxheap` and then for each $i = n, n - 1, \ldots, 2$ we swap elements and cut the node with index $i$ off the tree; these two together take $c_2$ time.

  We also run `maxheapify` again specifically on the node with index 1. However for each $i = n, n - 1, \ldots, 2$ we’ve cut the node with index $i$ off the tree and so the tree now has $i - 1$ nodes to consider. Thus in this case we have, modified from above:
In total then:

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

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

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

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

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

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

Thus we have \( T(n) = \mathcal{O}(n \log n) \).
2.3 Heapsort 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 \texttt{maxheapify} does no swaps and therefore makes no recursive calls and therefore takes constant time, hence time complexity \( T_{mh}(i, n) = \Theta(1) \), instead of the worst-case logarithmic.

As a consequence \texttt{converttomaxheap} will have \( \Theta(n) \) time complexity because it runs \texttt{maxheapify} on \( \lfloor n/2 \rfloor \) nodes.

The calls that \texttt{heapsort} makes to \texttt{maxheapify} will themselves take constant time as well (no swaps) and so it follows that \texttt{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 \texttt{maxheapify} does no swaps and so \texttt{converttomaxheap} will have \( \Theta(n) \) time complexity. However the calls that \texttt{heapsort} makes to \texttt{maxheapify} will result in swaps because the smaller elements are getting swapped into the root node as \texttt{heapsort} progresses and consequently the final time complexity will still be \( \Theta(n \lg n) \).

2.4 Heapsort Auxiliary Space

HeapSort uses \( \mathcal{O}(1) \) auxiliary space.

2.5 Heapsort Stability

HeapSort is unstable.

2.6 Heapsort In-Place

HeapSort is in-place.

2.7 Heapsort 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.
3 Pseudocode for Everything

Here is the pseudocode for the various functions.

3.1 Pseudocode for Maxheapify

Here it is assumed that TREE is a heap and n is the number of nodes in TREE. The conditionals leftnode <= n and rightnode <= n simply check for the existence of children of node i before checking the values residing there.

Here is the pseudocode:

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

3.2 Pseudocode for Converttomaxheap

Here is the pseudocode.

```plaintext
function converttomaxheap(TREE,n)
    for i = floor(TREE.n/2) down to 1
        maxheapify(TREE,i,n) aka floatvaluedown(TREE,i,n)
    end
end
```

3.3 Pseudocode for Heapsort

Here is the pseudocode:

```plaintext
function heapsort(A,n)
    converttomaxheap(A,n)
    for i = n down to 2
        swap(A[1],A[i])
        maxheapify(TREE,1,i-1) aka floatvaluedown(TREE,1,i-1)
    end
end
```
4 Thoughts, Problems, Ideas

1. Consider the following complete binary tree.

(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:

```
    1
   /|
  2 9
 / |
4 5
 /|
8 10
```

Suppose you forget to `converttomaxheap` 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.
5 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
    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]
```