CMSC 351: HeapSort

Justin Wyss-Gallifent

October 8, 2024

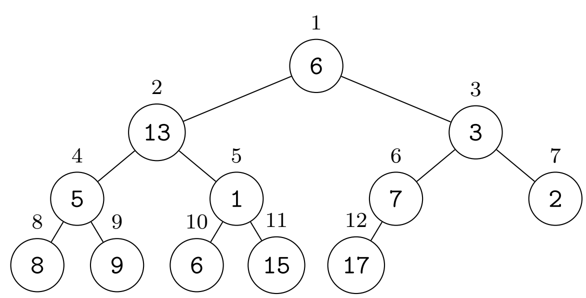
1 Complete Binary Trees

1.1 Definition

Definition 1.1.1. A *complete binary tree* is a binary tree in which all levels are completely filled, except possibly for the bottom level, and the bottom level has all entries as far left as possible.

Typically complete binary trees are 1-indexed where the indices are assigned left-to-right from the top down. In this way a complete binary tree can be represented by a simple list.

Example 1.1. Here is an example of a complete binary tree.



This may be represented by the 1-indexed list:

$$A = [6, 13, 3, 5, 1, 7, 2, 8, 9, 6, 15, 17]$$

1.2 Index Notes

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 1.2. In the above example the node with index 5 has children with indices 10 and 11.

- If a node has index i then its parent has index $\lfloor i/2 \rfloor$.

  Example 1.3. In the above example the node with index 7 has parent with index $\lfloor 7/2 \rfloor = \lfloor 3.5 \rfloor = 3$.

- As a special case of the above, if there are n nodes total then the largest node with children is the node with index $\lfloor n/2 \rfloor$.

  Example 1.4. In the above example there are $n = 12$ nodes and the largest one with children is the node with index $\lfloor 12/2 \rfloor = 6$.

- If a node with index i has children than all nodes with smaller indices also have children.

Example 1.5. 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, ... , $\lfloor n/2 \rfloor$ are the ones that have children.

Example 1.6. In the above example the nodes with indices 1, 2, ... , 6 are the ones that have children.

1.3 Level Notes

- The leftmost node at level k (with level $k = 0$ being the level of the root) is the node with index 2^k.

Example 1.7. In the above example the leftmost node at level 3 is the node with index $2^{3-1} = 4$.

- A node with index i is located in level $\lfloor \lg i \rfloor$.

Example 1.8. In the above example the node with index 6 is located in level $\lfloor \lg(6) \rfloor = \lfloor 2.58 \rfloor = 2$.

- As a special case of the above, if there are n nodes total then the maximum level (the leaf level) equals $\lfloor \lg n \rfloor$.

Example 1.9. In the above example there are $n = 12$ nodes and $\lfloor \lg(12) \rfloor = \lfloor 3.58 \rfloor = 5$ levels.

- The number of levels between a node with index i and the leaf layer, inclusive, is then $\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1$.

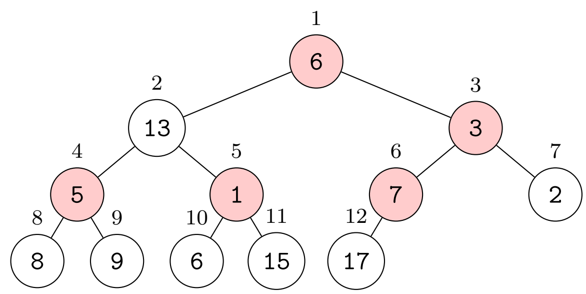
Example 1.10. In the above example the number of levels between the node with index 3 and the leaf level, inclusive, is $\lfloor \lg 12 \rfloor - \lfloor \lg 3 \rfloor + 1 = 3$.

2 Max Heaps

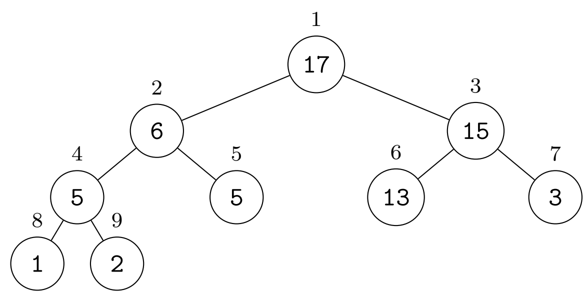
2.1 Definition

Definition 2.1.1. 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 key is greater than or equal to that node's children's key if that node has children. In other words keys non-strictly decrease (equality is acceptable) as we go down the branches.

Example 2.1. The example above is not a max heap. The nodes marked in red below violate the requirement because they have keys which are less than at least one of their children's keys:



Example 2.2. The following is a max heap, however:



3 Converting to a Max Heap

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 two processes.

3.1 Max Heapify - Fixing a Node

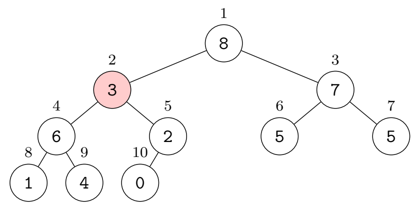
The `maxheapify` function atrociously named because its functionality isn't reflected well in its name. A better name would be `swapkeydown`.

We call the `maxheapify` function on a node whose children are each already roots of their own max heaps. It swaps the key at index i down the tree as far as necessary (if at all) to ensure that the subtree rooted at index i is also a max heap.

It does this by asking "Is my key smaller than either of my children's keys?" If not, then we're done. If so, then the key is swapped with the largest child key and then `maxheapify` calls itself again on that child.

Note 3.1.1. We really want to emphasize that it is `maxiheapify` which calls itself again, recursively. We do not need to make these recursive call manually; We only manually make the first call!

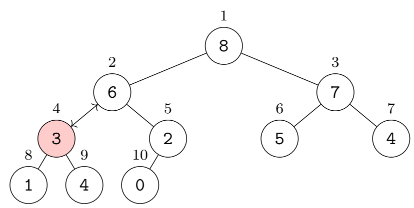
Example 3.1. For example consider the red node (node with index 2) in the following tree. Note that the subtrees rooted at its children are max heaps:



We can float this problematic key down by repeatedly following the branch to the largest key. Here is the process.

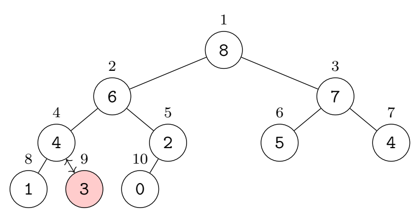
We call `maxheapify(A,2)`.

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



Our previous call to `maxheapify` then recursively calls `maxheapify(A,4)`.

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



Note 3.1.2. This process stops either when the key is larger than the keys of both its children or we reach the bottom of the tree.

Observe that because the smaller key moves down along a path which results in larger keys floating up, and because we never float a key up above a higher key, 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.

What is the time complexity of this? If i is the index of the node we're fixing then this node is in level $\lfloor \lg i \rfloor$.

In the best case we compare with the children and there's no issue, so this is $\Theta(1)$.

In the worst case we need to check all the way to the bottom level which is

$\lfloor n/2 \rfloor$ and so this is $\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1$ levels. In order to get this to depend just on n we note:

$$\lfloor \lg n \rfloor - \lfloor \lg i \rfloor + 1 \leq 1 + \lg n$$

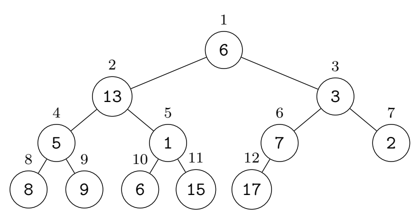
Thus in the worst case this is $\mathcal{O}(\lg n)$.

3.2 Convert to Max Heap - Fixing a Tree

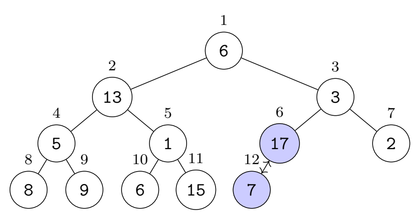
Given a complete binary tree we can convert it to a max heap by running `maxheapify` on all the nodes that have children. From our index arguments before we know this is 1, 2, ... , $\lfloor n/2 \rfloor$. We go through these in reverse order so that when we fix a node we are assured that the subtree rooted at that node is already fixed.

The process above is performed by the `converttomaxheap` function. Thankfully the name of this function is exactly what it does!

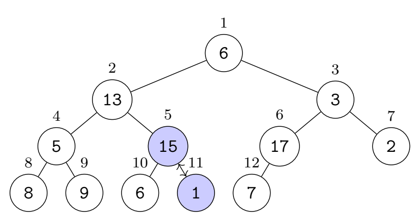
Example 3.2. Here is the process as applied to our original tree:



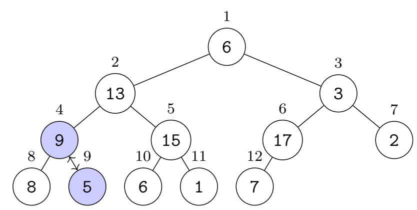
Here we have `maxindex(A)=12` and so `floor(maxindex(A)/2)==6` and so we start with the node with index 6 (the last node with children). Running `maxheapify(A,6)` interchanges keys at indices along the chain $6 \leftrightarrow 12$ only:



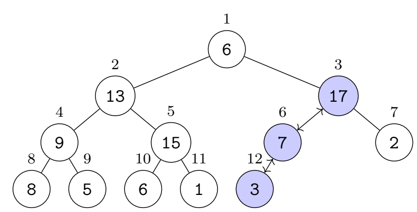
Running `maxheapify(A,5)` interchanges keys at indices along the chain $5 \leftrightarrow 11$ only:



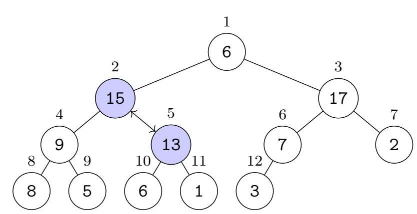
Running `maxheapify(A,4)` interchanges keys at indices along the chain $4 \leftrightarrow 9$ only:



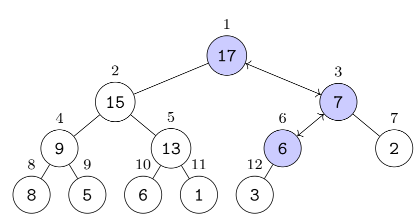
Running `maxheapify(A,3)` interchanges keys at indices along the chain $3 \leftrightarrow 6 \leftrightarrow 12$ only:



Running `maxheapify(A,2)` interchanges keys at indices along the chain $2 \leftrightarrow 5$ only:



Running `maxheapify(A,1)` interchanges keys 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 `maxheapify` on any particular node preserves the max-heap property of the two child subtrees and induces the max-heap property on the full subtree.

What is the time complexity of this? Consider we're running the process on $\lfloor n/2 \rfloor$ nodes.

In the best case we are running a $\Theta(1)$ process $\lfloor n/2 \rfloor$ times. Since we know that $n/2 - 1 \leq \lfloor n/2 \rfloor \leq n/2$ we know this is $\Theta(n)$.

In the worst case we are running a $\mathcal{O}(\lg n)$ process $\lfloor n/2 \rfloor$ times. Since we know that $\lfloor n/2 \rfloor \le n/2$ we know this is $\mathcal{O}(n \lg n)$.

Note 3.2.1. We should note here that while it's obvious that `converttomaxheap` is worst-case $\mathcal{O}(n \lg n)$ it is actually the case that it's worse-case $\Theta(n)$. The proof of this requires a much more detailed analysis of the number of swaps actually occurring during the many calls to `maxheapify`.

4 Heapsort

4.1 Algorithm

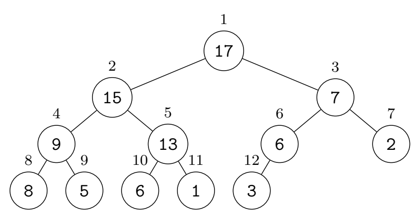
A max binary heap is structured such that extracting the keys in a sorted manner is very easy. There are several ways to do this, all are based on the observation that the largest key is at the root node so that key needs to be last in our sorted list. What we'll do is exchange it with the key in the 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.

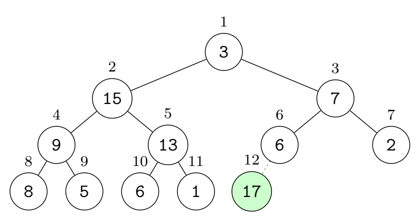
We then repeat the process on the new tree and keep repeating until we're done.

Example 4.1. Here is the process on our heap from earlier:

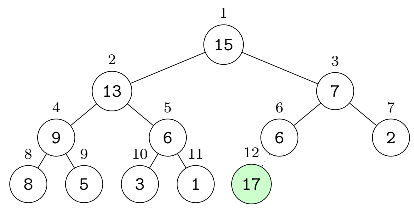
We start with:



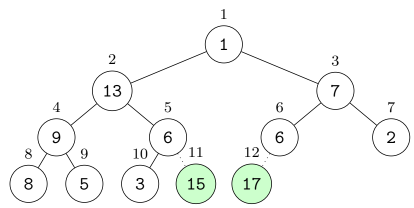
We interchange the keys at the nodes with indices 1 and 12 and cut the node with index 12 off from the tree:



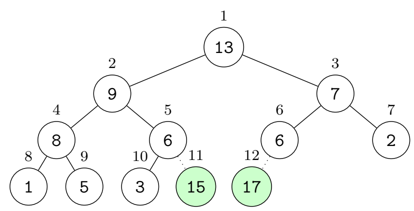
We then run `maxheapify` but only on the subtree:



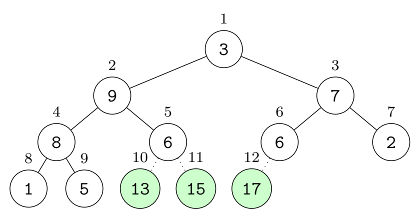
We interchange the keys at the nodes with indices 1 and 11 and cut the node with index 11 off from the tree:



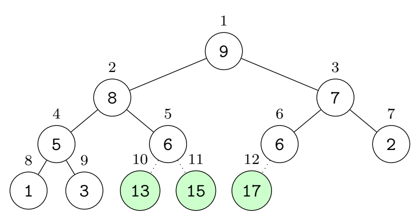
We then run `maxheapify` but only on the subtree:



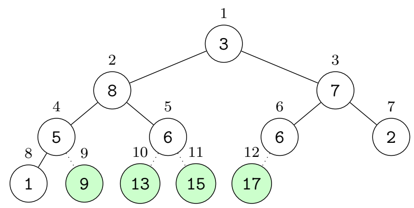
We interchange the keys at the nodes with indices 1 and 10 and cut the node with index 10 off from the tree:



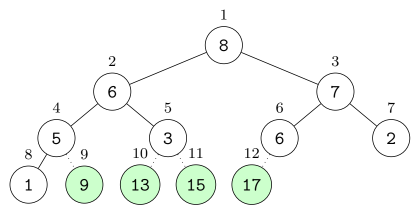
We then run `maxheapify` but only on the subtree:



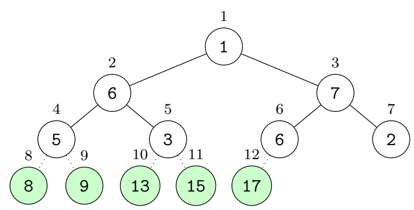
We interchange the keys at the nodes with indices 1 and 9 and cut the node with index 9 off from the tree:



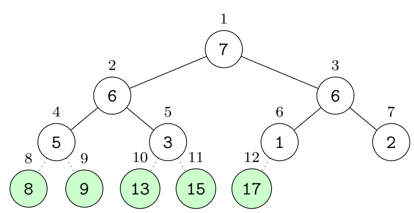
We then run `maxheapify` but only on the subtree:



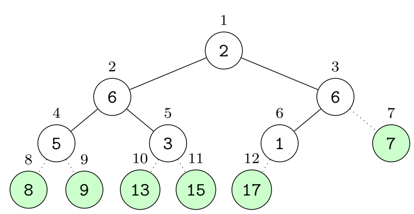
We interchange the keys at the nodes with indices 1 and 8 and cut node with index 8 off from the tree:



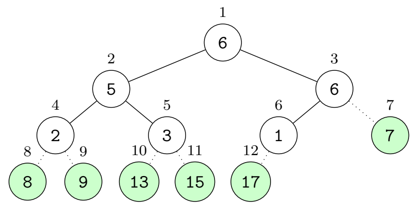
We then run `maxheapify` but only on the subtree:



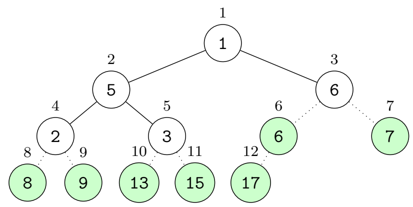
We interchange the keys at the nodes with indices 1 and 7 and cut the node with index 7 off from the tree:



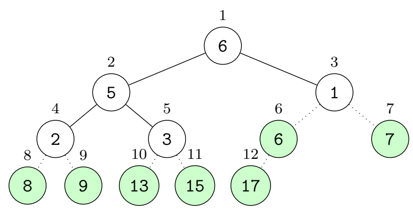
We then run `maxheapify` but only on the subtree:



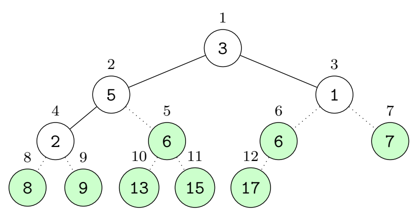
We interchange the keys at the nodes with indices 1 and 6 and cut node with index 6 off from the tree:



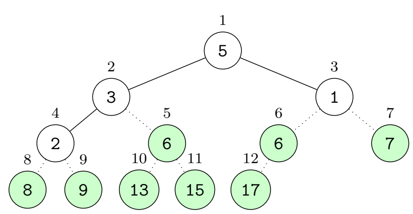
We then run `maxheapify` but only on the subtree:



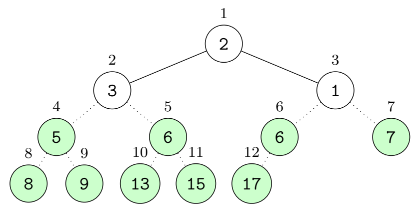
We interchange the keys at the nodes with indices 1 and 5 and cut the node with index 5 off from the tree:



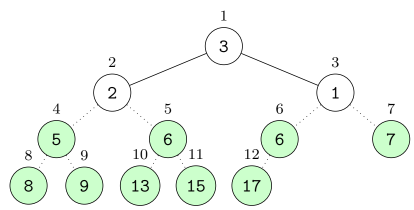
We then run `maxheapify` but only on the subtree:



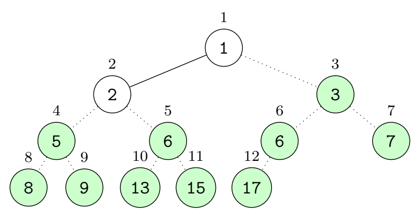
We interchange the keys at the nodes with indices 1 and 4 and cut the node with index 4 off from the tree:



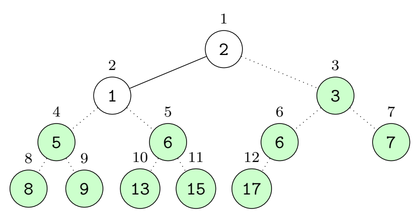
We then run `maxheapify` but only on the subtree:



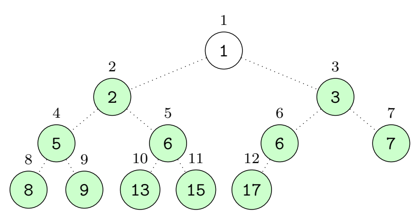
We interchange the keys at the nodes with indices 1 and 3 and cut the node with index 3 off from the tree:



We then run `maxheapify` but only on the subtree:



We interchange the keys 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 keys by the indices of the nodes:

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

4.2 Heapsort Worst-Case Time Complexity

Consider that in the worst case we go through the following process:

(a) We `converttomaxheap`. We've seen that this is worst-case $\mathcal{O}(n \lg n)$ or arguably $\Theta(n)$.

(b) For $i = n, n-1, ..., 2$ we swap node i with node 1 (this is $\Theta(1)$), we cut node i off the tree (this is $\Theta(1)$), then we run `maxheapify` on node 1. Running `maxheapify` is worse-case $\mathcal{O}(\lg n)$ but this is for a tree with n nodes. At this point our tree has $i-1$ nodes so it's $\mathcal{O}(\lg(i-1))$.

The total time required is then as follows, assuming `converttomaxheap` is $\mathcal{O}(n \lg n)$:

$$\begin{aligned}
\mathcal{O}(n \lg n) + \sum_{i=2}^{n} [\Theta(1) + \Theta(1) + \mathcal{O}(\lg(i-1))] &\leq \mathcal{O}(n \lg n) + \sum_{i=2}^{n} \mathcal{O}(\lg(i-1)) \\
&\leq \mathcal{O}(n \lg n) + \sum_{i=2}^{n} \mathcal{O}(\lg n) \\
&\leq \mathcal{O}(n \lg n) + (n-1)\mathcal{O}(\lg n)
\end{aligned}$$

Thus the time complexity is $\mathcal{O}(n \lg n)$.

If we take `converttomaxheap` as worst-case $\Theta(n)$ this does not change the result for `heapsort`.

4.3 Heapsort Best-Case Time Complexity

A few notes related to best-case time complexity:

1. If we start with a heap of distinct elements which is already a max heap then `converttomaxheap` will be $\Theta(n)$ (scan but no swaps). However when we start the swap-cut-`maxheapify` process we will be swapping smaller nodes with the root node and then `maxheapify` on the root node will be $\mathcal{O}(\lg(i-1))$ again each time for a total of:

$$\mathcal{O}(n) + \sum_{i=2}^{n} [\Theta(1) + \Theta(1) + \mathcal{O}(\lg(i-1))] = \mathcal{O}(n \lg n)$$

2. If we start with a heap of identical elements then `converttomaxheap` will be $\Theta(n)$ (scan but no swaps). In addition the swap-cut-`maxheapify` process will be swapping identical nodes with the root note and then `maxheapify` on the root node will be $\Theta(1)$ each time for a total of:

$$\mathcal{O}(n) + \sum_{i=2}^{n} [\Theta(1) + \Theta(1) + \Theta(1)] = \mathcal{O}(n)$$

4.4 Heapsort Auxiliary Space

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

4.5 Heapsort Stability

HeapSort is unstable.

4.6 Heapsort In-Place

HeapSort is in-place.

4.7 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.

5 Pseudocode for Everything

Here is the pseudocode for the various functions.

5.1 Pseudocode for Maxheapify

Here it is assumed that `A` is a 1-indexed list representing a heap and `n` is the length of `A`, meaning the number of nodes/keys in the tree. The conditionals `leftnode <= n` and `rightnode <= n` simply check for the existence of children of node i before checking the keys residing there.

Here is the pseudocode:

```
// Maxheapify node i on the tree A with n nodes.
function maxheapify(A,i,n)
    leftnode = 2*i
    rightnode = 2*i+1
    largestnode = i
    if leftnode <= n and A[leftnode] > A[largestnode]
        largestnode = leftnode
    end
    if rightnode <= n and A[rightnode] > A[largestnode]
        largestnode = rightnode
    end
    if largestnode != i
        swap(A[i],A[largestnode])
        maxheapify(A,largestnode,n)
    end
end
```

5.2 Pseudocode for Converttomaxheap

Here is the pseudocode.

```
// Run maxheapify on a tree represented
// by the 1-indexed list A with n nodes.
function converttomaxheap(A,n)
    for i = floor(n/2) down to 1
        maxheapify(A,i,n)
    end
end
```

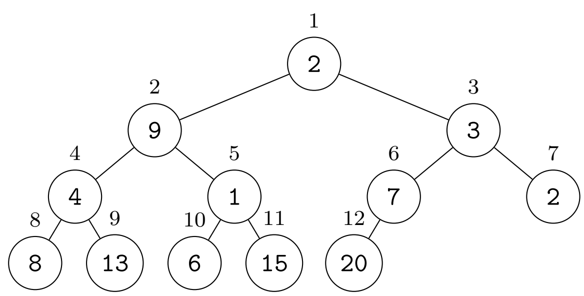
5.3 Pseudocode for Heapsort

Here is the pseudocode:

```
function heapsort(A,n)
    converttomaxheap(A,n)
    for i = n down to 2
        swap(A[1],A[i])
        maxheapify(A,1,i-1)
    end
end
```

6 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 keys 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.

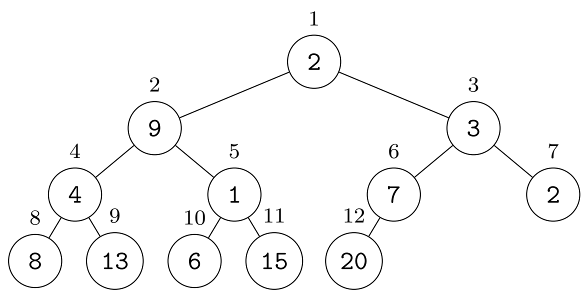
3. Prove that $\lfloor \lfloor x/2 \rfloor /2 \rfloor = \lfloor x/4 \rfloor$.

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 `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.

7 Python Test

Code:

```
# 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[i] = A[largestnode]
        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[1] = A[i]
        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]
[5, 7, 8, 31, 50, 60, 74, 82, 90]
```