**Heaps and Heapsort**

Table of Contents

[Binary Trees 2](#_Toc66036166)

[Heaps 3](#_Toc66036167)

[Maintaining the Heap Property 4](#_Toc66036168)

[Building A Heap 5](#_Toc66036169)

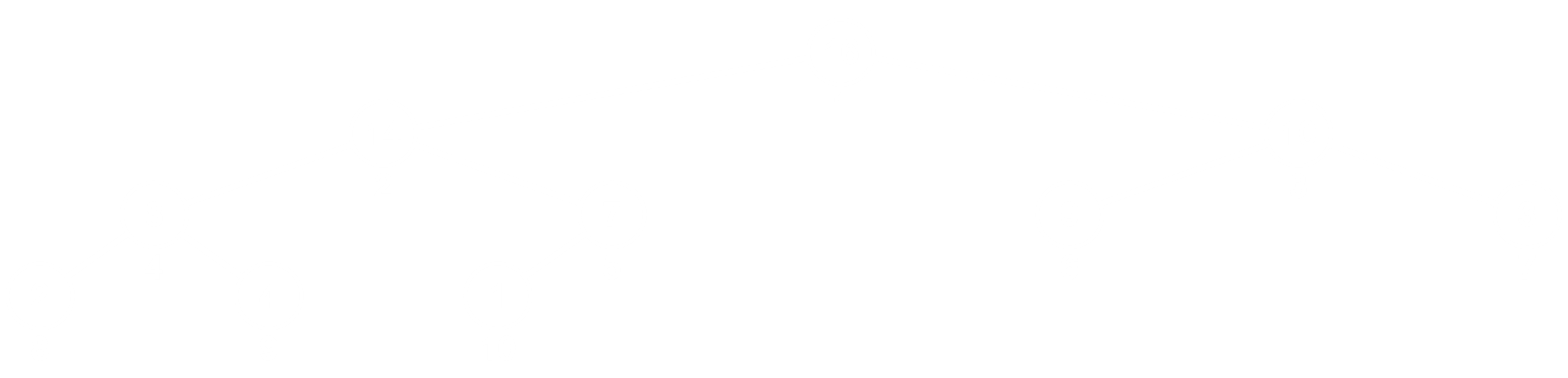
[Proving buildHeapMax Works 5](#_Toc66036170)

[Time Complexity of buildMaxHeap 6](#_Toc66036171)

[Heap Sort 8](#_Toc66036172)

## Binary Trees

Until now, we have used arrays and linked lists. Both of these are linear storage systems. A tree on the other hand, is non-linear. Data is stored as nodes, with different pieces of data being connected with edges. A tree is set up in a way such that some nodes are parents, and they are connected to children below them. Visually, this is what a tree would look like:



Keep in mind that there is no actual tree structure. This is simply a way of visualizing how everything is set up. In a binary tree like this one, each node has a maximum of two children. A node with no parent is called the root node. A node with no children is called a leaf node.

## Heaps

A heap is a nearly complete or complete binary tree. This means, for the allowed number of levels, all levels are filled or nearly filled with nodes. If there are levels, there can be nodes at most. In a heap, for any one level, children are added left to right.

There are two types of heaps, max heap and min heap. In max heap, the value of any parent node is greater than all its child nodes. In a min heap, the value of any parent node is less than all its child nodes. Thus, for max heap, for all values of other than when is the root, and for min heap, for all values of other than when is the root. These are the invariants for a heap.

The heap data structure is actually an array. The tree above, which is for a max heap, has this array:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

|  |  |
| --- | --- |
| Parent Index | Child Indices |
| 1 | 2, 3 |
| 2 | 4, 5 |
| 3 | 6, 7 |
| 4 | 8, 9 |
| 5 | 10 |

All nodes after are leaves. For every node before , the left child and the right child .

## Maintaining the Heap Property

Now we will look at how to create a heap from an array of integers. This code takes an array A, and a certain position on the array i is considered as the parent. It assumes that everything ‘below’ that position are already max heaps.

maxHeapify (A, i, heapSize)  
left = LEFT(i)  
right = RIGHT(i)  
  
if left<=heapsize and A[left]>A[i] largest = left  
else largest = i  
  
if right<=heapsize and A[right]>A[largest] largest = right  
if largest != i  
 swap A[i] with A[largest]  
 maxHeapify (A, largest, heapSize)

PSEUDOCODE

Both left and right children for that index are found and compared to the index member. If the largest member is not the parent, it is exchanged with the parent.

The last line recursively calls the function again in case there has been an exchange. The reason for this is that the change may mean that the branches below this are no longer max heaped, so they must be heapified again.

## Building A Heap

From the way the pseudocode is set up, it should be obvious that it must be worked with from bottom up. Also, this code is for just one parent. We must call it for each parent in turn. We know that there are no parents after (A.heapsize)/2. Thus,

buildMaxHeap(A, heapSize)  
for i = (A.heapsize)/2 to i = 1  
 maxHeapify (A, i, heapSize)

PSEUDOCODE

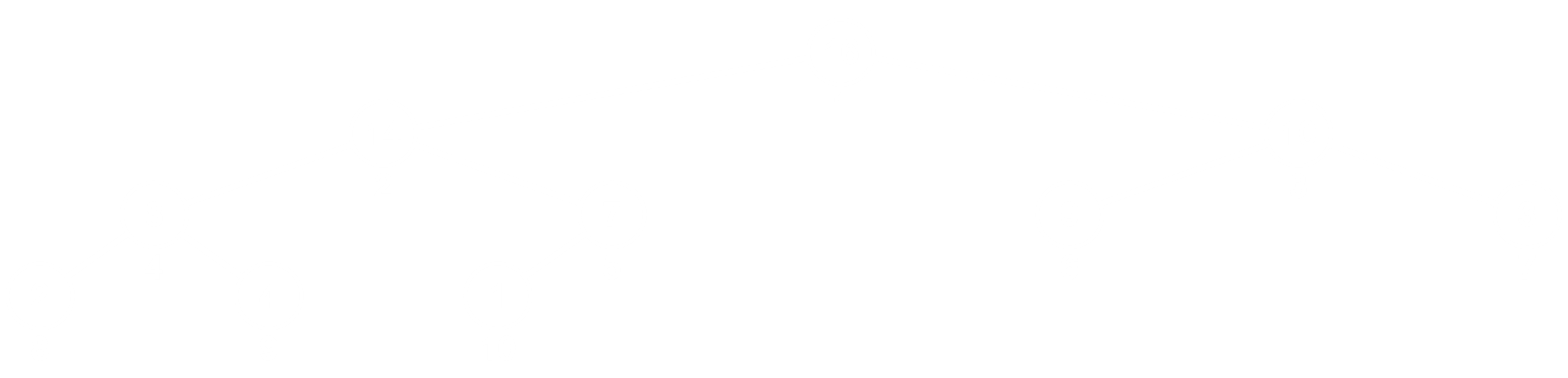
### Proving buildHeapMax Works

The algorithm goes bottom up, making each index it goes to the root of a max heap and fixing everything below it accordingly. This mean for each node the function is called on, that node becomes the maximum node, so we always have at least that part as a max heap. This is similar to how we had a sorted part of the array at all times with insertion sort.

Formally, for every iteration i, every item from i+1 to the end of the array are each the root of a max heap. Thus, at the end, every single node is the root of a max heap, which makes the entire structure a max heap.

### Time Complexity of buildMaxHeap

Consider the following binary tree:



Say we declare the tree to have 4 levels, from 0 to 3 from top to bottom. We can say that each level has a height, , from 0 at the bottom to 3 at the top. If the tree were full, it would have 15 nodes in it. For the complete binary tree, the height of the root is .

In the maxHeapify function, in the worst case, a node at height thus has swaps due to recursive calls (since there are levels below it). The root would thus have the most possible swaps. If there are elements, the root would be at a height of , and thus it would make swaps in the worst case. Since we call the maxHeapify function times in total, the total worst case time complexity for building a max heap is , thus O(n log n).

We could make a slightly more complicated analysis though. Going from bottom up, at , there are elements, with each element making swaps in the worst case. At , there are elements, with each element making swap in the worst case. Moving on in the same fashion, if we consider each swap to take time , we get an equation that looks a little like this (may be slightly inaccurate but that’s not the point):

Let

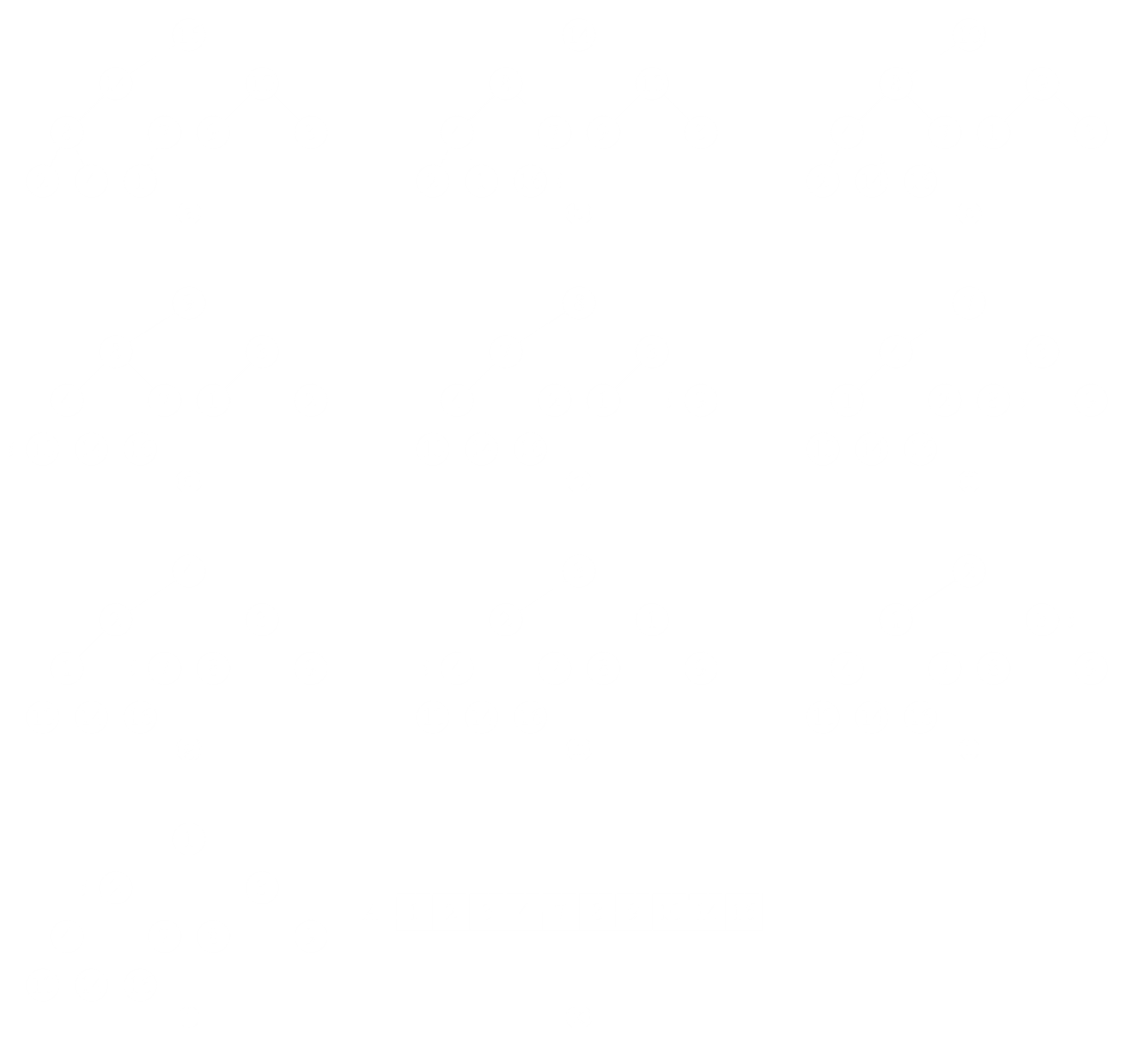
Since is a constant, and the value of is much smaller than the value of , for larger values of , we can ignore both of those terms. Thus, we are left with,

Thus, buildMaxHeap is O(n).

## Heap Sort

In a max heap, we know that the top element is the maximum of the array. Thus, in heap sort, we can repeatedly add the top element to an array, and then heapify the entire thing again. To avoid breaking the top link and thus creating two separate heaps, we replace the top element with the last element of the heap (this may not be the least value), and break the link of the last element with its parent. Then we heapify the heap again. Notice that we get a smaller heap and we need to keep track of the ending position. We do this with a separate variable called heapSize.

Visually, the process looks something like this:



We are creating an extra array to store our sorted array. This means we have a space complexity of O(n). We can reduce this though. Instead of creating a new array, we can simply swap the position of the top element with the position of the end element, the one whose link we have broken. Since that position is no longer a part of the heap, doing this does not affect our algorithm. And since we are reusing space that we already had, no extra space is needed. Thus, the space complexity becomes O(1). Heap sort is thus linear in terms of time complexity, and constant in terms of space complexity. It is an in-place sorting algorithm.

heapSort(A, heapSize)  
length = heapSize;

buildMaxHeap(A, length);  
while (length>0)  
 maxHeapify(A, 1, length);  
 temp = A[1];  
 A[1] = A[length];  
 A[length] = temp;  
 length--;

PSEUDOCODE