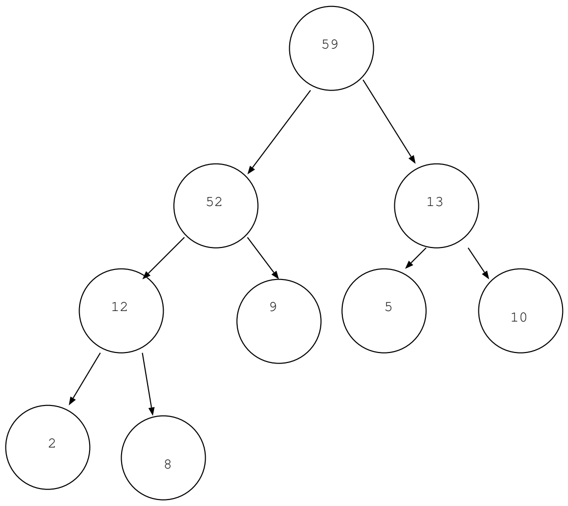
HW3

Shane Drafahl

20 October,2017

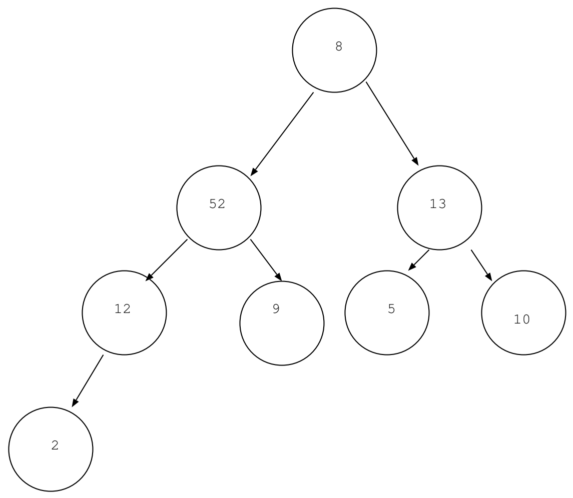
1. (a)

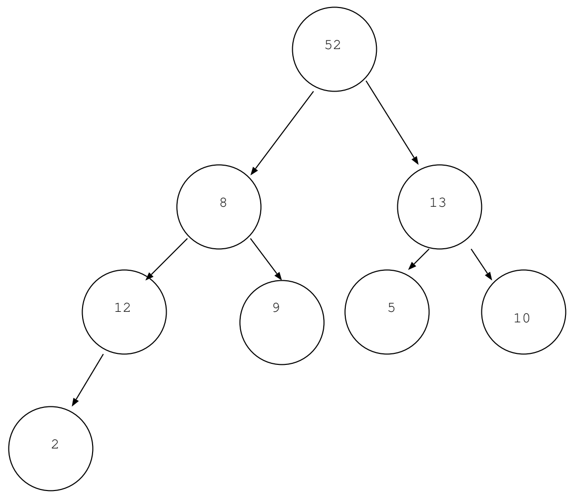


First we will replace the removed element with the last element in the tree

(b):

Then we must heapify the data structure

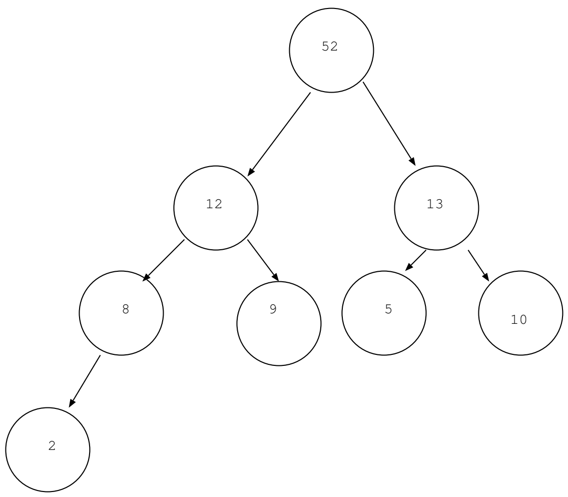




The max heap is now been heapified.

2. Solve the following recurrences. You can not use master theorem to solve them. You must show the steps in your derivation.

(a): If we draw out a 3 iterations then we would get



$[cn]_1 + [\frac{cn}{3} + \frac{2cn}{3}]_2]_2 + [\frac{cn}{9} + \frac{2cn}{9} + \frac{2cn}{9} + \frac{4cn}{9}]_3$
where the brackets represent the layers this can be reduced to $[cn]_1 + [cn]_2 + [cn]_3 + ...$ for all iterations. So now we just need to find the number of iterations. Every iteration divides n by 3 or by $\frac{2}{3}$. This means that the tree wont be perfectly balanced. The tree will be its deapest on the right side of the tree because it will take more iterations for it to equal 1. The right side of the tree will go for $log_{3/2}(n)$ iterations. The left side of the tree will go for $log_3(n)$ iterations. We will find the deepest path into the tree so the value is $log_{3/2}(n) * cn$.
(b): First we will draw out a few iterations to find what the infinite series would be.
The sum of the the function would be $cn + \frac{cn}{5} + \frac{cn}{25} + ...$
Notice that the denominator 5^i where i is the iteration number. So we can write this as an infinite series.
$\sum_{i=1}^{\infty}(\frac{cn}{5})^i$.
From this we can determine that the total sum is $\frac{5nc}{4}$.
(c): We will draw out the first few iterations again. $[n^{log_5(7)}]_0 + [\frac{n^{log_5(7)}}{2} + \frac{n^{log_5(7)}}{2}]_1 + [\frac{n^{log_5(7)}}{4} + \frac{n^{log_5(7)}}{2}]_4 + \frac{n^{log_5(7)}}{2}]_4 + \frac{n^{log_5(7)}}{2}]_4]_2 + ...$
As we can tell this can ben reduced to $n^{log_5(7)} + n^{log_5(7)} + n^{log_5(7)} + ...$ so that means we just have to find the height of the tree.
Notice the denominator's value is 2^i where i is value of the iteration. So we know the last iteration is when $\frac{n}{2^i} = 1$ so there will be $log_2(n)$ iterations. So the sum is $log_2(n) * n^{log_5(7)}$
3. The worst case for this algorithm is that in the first comparison before

the for loop the biggest element in the array is set to largest. In this case every the first comparison will fail and the second comparison will always be triggered. the for loop will have $(n-2)$ iterations and two comparisons per iteration so inside the for loop there are $2(n-2)$. The first comparison before the for loop will always be triggered so therefore in total there are $2n-3$ comparisons worst case.
This algorithm will use divide and conqure strategy to divide the array into two sub arrays untill it only has 2 elements in the array. These arrays are then merged together where the first element is the largest element and the second in the array is the second largest element.

```
// start is the first index inclusive
// finish is the last index inclusive
// A is an array of elements
function getGreatestAndSecondGreatestValue(A, start, finish) {
    T[] // T is an array of size 2
    if(finish - start == 1) {
        T[0] = max(A[start], A[finish])
        T[1] = min(A[start], A[finish])
        return T
    }

    Array1 = getGreatestAndSecondGreatestValue(A,start, start + floor((last-f
    Array2 = getGreatestAndSecondGreatestValue(A, start + ceil((last-first)/2
    return merge(Array1, Array2);
    // The greast value will be in the 0th index of the array returned and th
    // greatest value will be in the 1 index of the array.
}

// this merges two arrays keeping the greatest value on the left side of the
// keeping the second greatest value on the right side of the array
function merge(Array1, Array2) {
    MergedArray[] // array that will hold the merged results
    if(Array1[0] > Array2[0]) {
        MergedArray[0] = Array1[0]
        MergedArray[1] = max(Array1[1], Array2[0])
    } else {
        MergedArray[0] = Array2[0]
        MergedArray[1] = max(Array1[0], Array2[1])
    }
    return MergedArray
}
```

Using recurrences to find the number of comparisons we know that it takes 2 comparisons each iteration. We also know it is recursively called twice per call so.
T(n) = 2T(n/2) + 1
If we visualize a tree where the top is n and the second layer is $\frac{n}{2}$ and $\frac{n}{2}$ and so on. 2 is added for every node so we just have to find the total numbers of nodes. We know that the height of the tree is $log_2(n)$ so there are $2^{log_2(n)} = n$ leaf nodes. We know that there are n - 1 internal nodes so we simply add them together to get $2n - 1$. This means there are $2n - 1$ nodes. During the base case there are is an extra comparator that is used during the base case. so we have to add one for every leaf node

4. The k sorted list is assentialy several different sorted lists that occupy the same memmory space. These lists just need to be merged in the order that they are in and then whatever is left needs to be inserted into the sorted sub array.