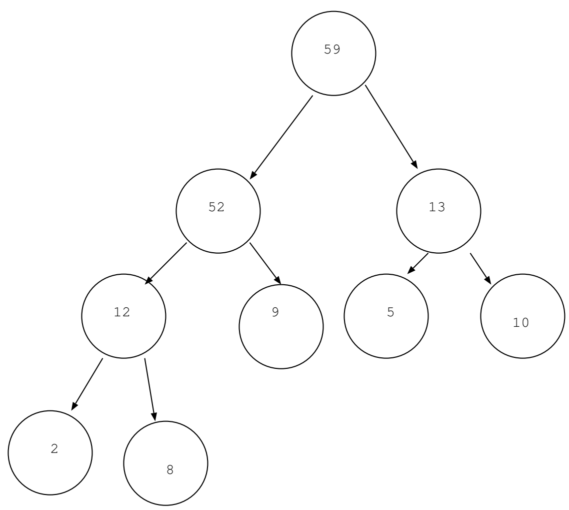
HW3

Shane Drafahl

20 October,2017

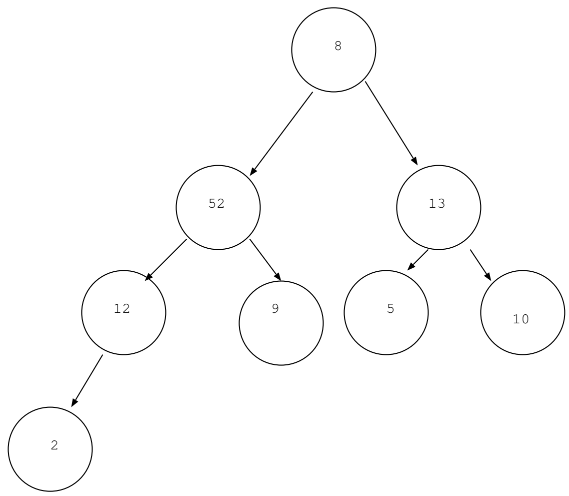
1. (a)

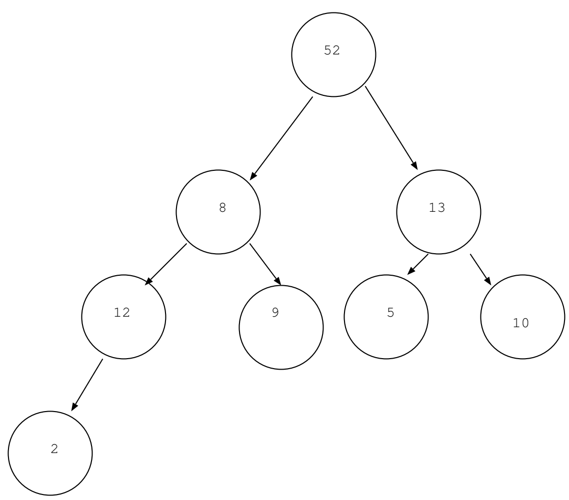


First we will replace the removed element with the last element in the tree

(b):

Then we must heapify the data structure

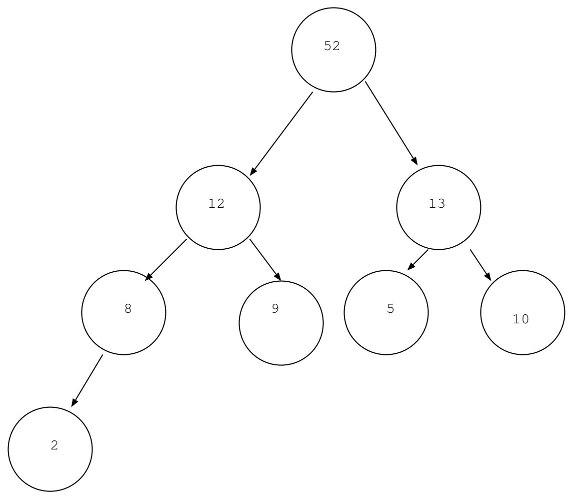




The max heap is now been heapified.

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
    merge(Array1, Array2)
}

// 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
    MergedArray[0] = max(Array1[0], Array2[0])
    temp1 = max(Array1[1], Array2[0])
    temp2 = max(Array1[0], Array2[1])
    second = min(temp1, temp2)
    MergedArray[1] = second
    return MergedArray
}
```

```
function max(a,b) {
    return (a + b + abs(a - b)) / 2
}

function min(a,b) {
    ((a+b)-abs(a-b)/2)
}
```

There is a single comparitor used in every single node. We know that the height of the tree is going to be $\frac{n}{2}$ because the base case is 2.
T(n) = T(n/2) + 1
We know that that the height of the tree is $log_2(n) - 1$ and there is a sequence $2^0 + 2^1 + 2^2 + ... + 2^{log_2(n)}$. $\sum_{i=0}^{log_2(n)-1} = n - 1$ so this algorithm takes n - 1 comparisons.

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.

```
// A is the array
// length is the length of the array
function sortKSorted(A, k, length) {

   r = length - k
   if(k > range) {
       U[] // unsorted array
       Q[] // sorted array of k sorted
       U = storeNonSorted(A, length - k + 1, k)
       quickSort(U, 0, U.length)
       x = 0
       l = 0 // length of sorted elements in A
       while(x < length) {
           Temp[]
           init = x // x is the index that we start at when going over the k sor
           c = 0
           for(x in range(r)) {
               Temp[c] = A[x]
```

```
            c++
        }
        r += k
        x = init + k
        quickSort(Temp, 0, r)
        c = 0
        for(c in range(r)) {
            Q[l] = Temp[c]
            l++
        }
    }
    mergeArrays(A, U, Q)
    return A
} else {
    Arrays[] // array of arrays
    x = 0
    for(x in range(k)) {
        Temp[] // new Array
        y = 0
        c = x
        while(y <= length) {
            Temp[y] = A[c]
            c += k
            y++
        }
        Arrays[x] = Temp
    }
    A = mergMultipleArrays(Arrays, 0, Arrays.length)
    return A
}
}

function mergMultipleArrays(Arrays[], first, last) {
    if(last - first == 1) {
        T[]
        return mergeArrays(T, Arrays[first], Arrays[last])
    }
    A1[] = mergMultipleArrays(Arrays[], first, floor(last/2))
    A2[] = mergMultipleArrays(Arrays[],  ceil(last/2), last)
    T1[]
```

```
    return mergeArrays(T1, A1, A2)
}


 // this function stores elements that are not sorted
 function storeNonSorted(A[], from, to) {
    Temp[] // temporary array
    x = 0
    for(from in range(to)) {
        temp[x] = A[from]
    }
    return Temp
 }

 function mergeArrays(A, Array1, Array2) {
    x = 0
    first = 0
    second = 0
    length = max(Array1.length, Array2.length)
    for(x in range(length)) {
        if(second >= Array2.length || (first < Array1.length && Array1[first] > A
            A[x] = Array1[first]
            first++
        } else {
            A[x] = Array2[second]
            second++
        }
    }
    return A
 }

 quickSort(arr[], low, high)
 {
     if (low < high)
     {
         /* pi is partitioning index, arr[p] is now
            at right place */
         pi = partition(arr, low, high);

         quickSort(arr, low, pi - 1);  // Before pi
```

```
        quickSort(arr, pi + 1, high); // After pi
    }
}


partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

This algorithm has two cases.If n are all the elements in the array and k is some value greater or equal to 1 and less than or equal $n-1$. If k is greater than n - k there are unsorted elements. Otherwise the array is completely made from parallel arrays that are merged together. If $k \geq (n-k)$ then there will be two sets of each end of the array that are k sorted. Quicksort is used and each set is of length k so it must take $2 * k * log(k)$. The unsorted part of the array that is not k sorted is $n - 2k$. Quicksort is used on this subarray of elements so it is $(n-2k)log(n-2k)$. These arrays are merged which is $O(n)$ because both arrays in total in length n. So the runtime of this algorithm is $O(n) + (n-2k)log(n-2k) + 2 * k * log(k)$ or $O(n + (n-2k)log(n-2k) + 2 * k * log(k))$. If $k \leq (n-k)$ then all the values in the array are k sorted. The algorithm would create k arrays. Each array is $\frac{n}{k}$ length so it would take n time. These arrays are then recursively merged so we can view it as a tree. Each layer of the tree costs n. The tree is of the height $log_2(k)$. So the runtime of this algorithm is $O(n * log_2(k))$ for this case.

5. Let $A = [a_1, a_2, ..., a_n]$ be an array of integers. Given $1 \leq i < j \leq n$.

Using divide and conqure we will find the greatest max subarray. The max subarray could either be in one of the divided sections or between two

```
function maxSubArray(A[], i, j) {
    // The base case
    if(i == j) {
        T[]
        T[0] = i
        T[1] = j
        T[2] = A[i]
        return T
    }

    mid = floor((i + j)/2)
    L[] = maxSubArray(A, i, mid)
    R[] = maxSubArray(A, mid + 1, j)
    C[] = findMaxMid(A, i, mid, j)
    if(L[2] >= R[2] && L[2] >= C[2]) {
        return L
    } else if(R[2] >= L[2] && R[2] >= C[2]) {
        return R
    } else {
        return C
    }

}

function findMaxMid(A[], low, mid, high) {
    sum1 = -1
```

```
    sum = 0
    i = mid
    leftMaxIndex
    // i decrements to low
    for(i in range(low)) {
        sum += A[i]
        if(sum > sum1) {
            sum1 = sum
            leftMaxIndex = i
        }
    }
    rightMaxIndex
    sum2 = -1
    sum = 0
    j = mid + 1
    // incraments from j to high
    for(j in range(high)) {
        sum += A[j]
        if(sum > sum2) {
            sum2 = sum
            rightMaxIndex = j
        }
    }
    D[] // (leftMaxIndex, rightMaxIndex, sum1 + sum2)
    D[0] = leftMaxIndex
    D[1] = rightMaxIndex
    D[2] = sum1 + sum2
    return D
}
```

Using recurrences to solve the runtime of this algorithm. The runtime of findMaxMid is $O((high-low))$ or $O(n)$. So the recurrences of maxSubArray are $T(n) = T(n/2) + T(n/2) + O(n)$. If we visualize this as a tree each layer of the tree takes n steps. The height of the tree is $log_2(n)$ so the runtime of this algorithm is $O(n * log_2(n))$.