Homework Assignment #3

Due: Noon, 9th Nov, 2017
Submit **printed** solutions via eClass
Submit **handwritten** solutions at dropbox on CSC level 1

CMPUT 204
Department of Computing Science
University of Alberta

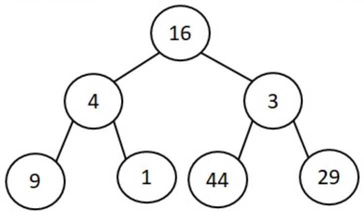
Note: All logs are base-2 unless stated otherwise.

Exercise I. What is the result of `Build-Max-Heap` on the input $A = [16, 4, 3, 9, 1, 44, 29]$? Illustrate all intermediate steps.

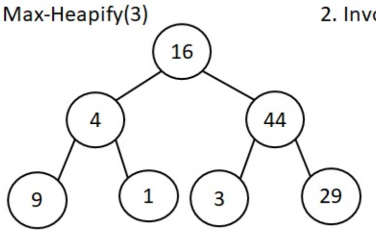
What is the result of increasing the key 1 to 34 once the max-heap is built?

Answer.

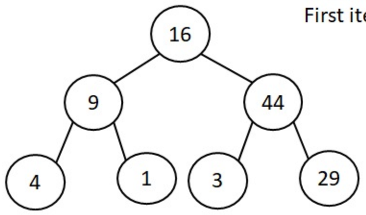
0. Initially:



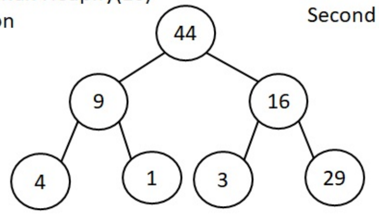
1. Invoking Max-Heapify(3)



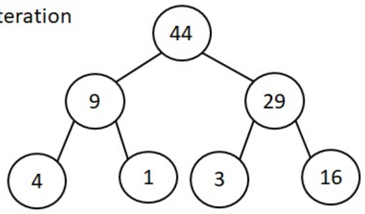
2. Invoking Max-Heapify(4)



3. Invoking Max-Heapify(16) – First iteration

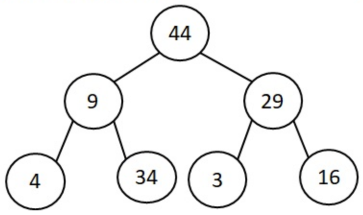


3. Invoking Max-Heapify(16) – Second iteration

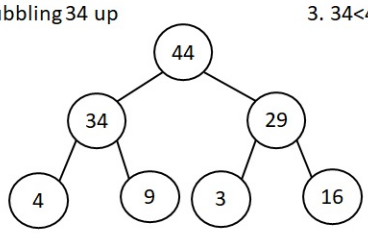


---

1. IncreaseKey(1 to 34)



2. Bubbling 34 up

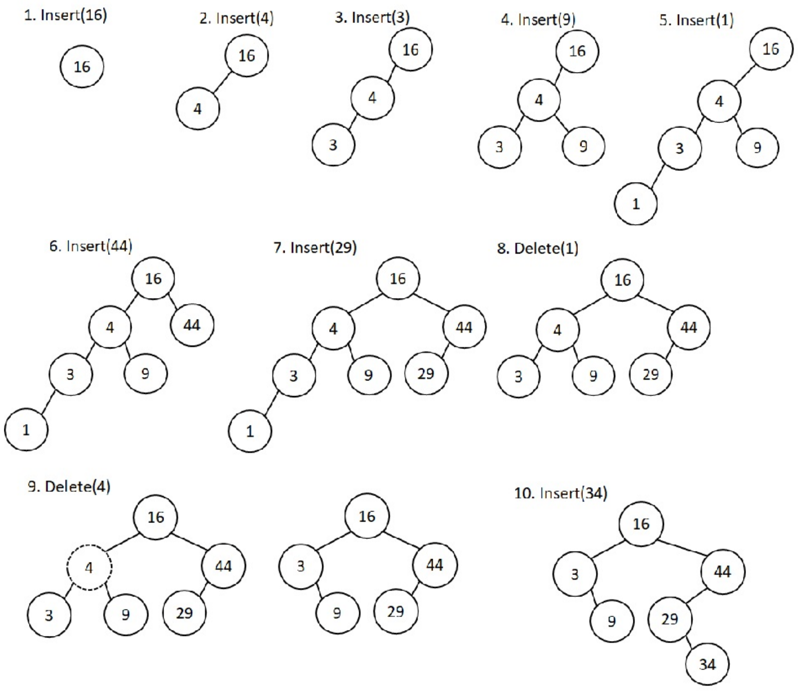


3. 34<44 so we halt.

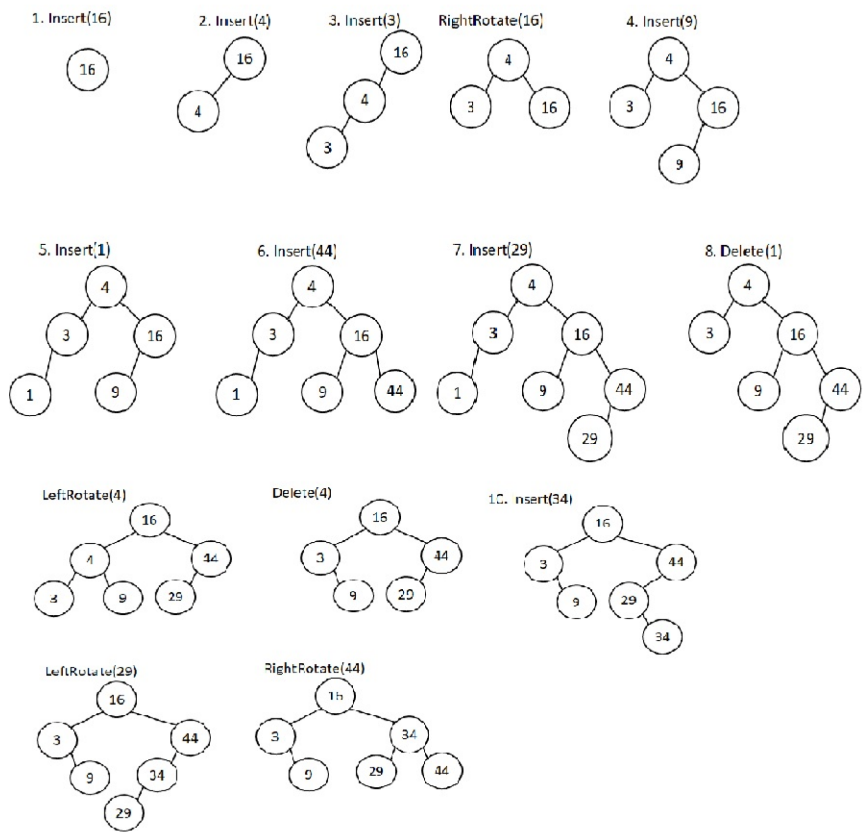
Exercise II. What is the result of the following sequence of operations on a (i) standard BST, (ii) an AVL-tree and (iii) a RB-tree:
`Insert(16)`, `Insert(4)`, `Insert(3)`, `Insert(9)`, `Insert(1)`, `Insert(44)`,
`Insert(29)`, `Delete(1)`, `Delete(4)`, `Insert(34)`.

Answer.

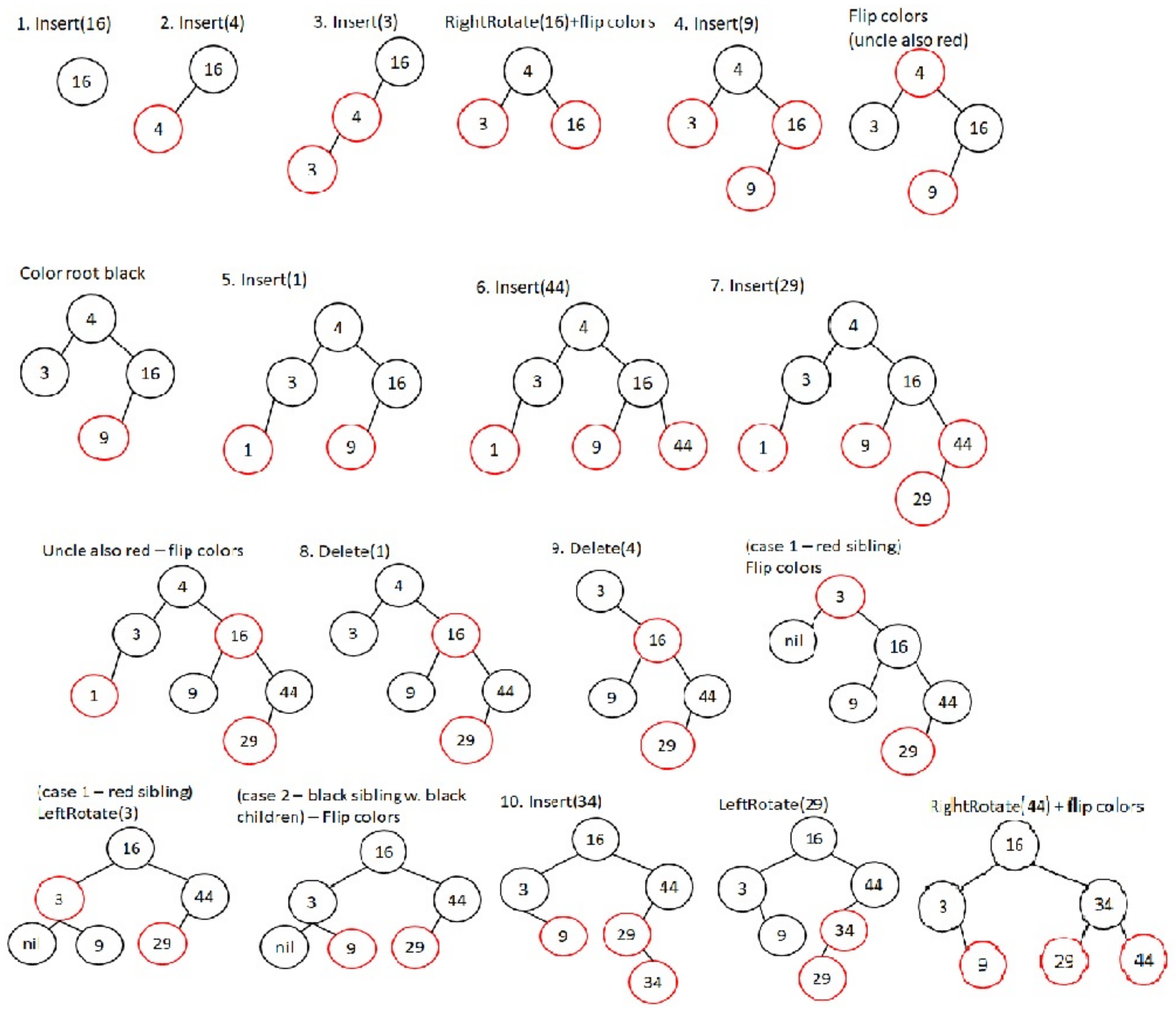
(i) BST:



(ii) AVL:


1. Insert(16)
2. Insert(4)
3. Insert(3)
RightRotate(16)
4. Insert(9)
5. Insert(1)
6. Insert(44)
7. Insert(29)
8. Delete(1)
LeftRotate(4)
Delete(4)
10. Insert(34)
LeftRotate(29)
RightRotate(44)

(iii) RB:


1. Insert(16)
2. Insert(4)
3. Insert(3)
RightRotate(16)+flip colors
4. Insert(9)
Flip colors (uncle also red)
Color root black
5. Insert(1)
6. Insert(44)
7. Insert(29)
Uncle also red – flip colors
8. Delete(1)
9. Delete(4)
(case 1 – red sibling) Flip colors
(case 1 – red sibling) LeftRotate(3)
(case 2 – black sibling w. black children) – Flip colors
10. Insert(34)
LeftRotate(29)
RightRotate(44) + flip colors

Exercise III. Given a BST T, let us add a field *size* that keeps track of the number of nodes in T.

1. (2pt) How would you alter `Insert()` to update the size of the tree (and all its subtrees) when we add a new node to T, in a way that doesn't increase the runtime of `Insert()`?

   Answer. Here is one way — and there are others. The new tree is created with $size = 0$ and we invoke the following code.

   ```
   NewInsert($T, key$) ** Inserts into $T$ a new leaf holding $key$
   Insert($T, key$) ** standard BST-insert
   $T \leftarrow$ Find($T, key$)
   while ($T \neq$nil) do
       $T.size \leftarrow T.size + 1$
       $T \leftarrow T.root.parent$
   ```

   Note that if standard `Insert()` places the new leaf at level l, this means that it took $O(l)$-time to insert the new node; but then finding the leaf holding *key* will also take $O(l)$-time, and the `while`-loop will also iterate $O(l)$ times. Therefore, if the original insertion takes $O(l)$ — then so is the insertion.

   Other technique could be to insert the new tree with $size = 1$ and change the code of `Insert()` so that upon a recursive call (whether we turn left or right) we afterward increment $T.size$. This increments in turn the *size* field of all ancestors of T.

2. (3pt) How would you alter `Delete()` to update the size of the tree (and all its subtrees) when we remove a node from T, in a way that doesn't increase the runtime of `Delete()`?

   Answer. Here we need to alter the `DeleteTree()` code. Once we invoke a deletion that actually removes a node from the BST, we have to travel from that node through all of its ancestors, up to the root, and decrement their *size*-fields. Thus, before the code calls an actual `delete` T (after it connects T's parent with T's single-child), we have to invoke the following lines:

   ```
   $T' \leftarrow T$
   while ($T' \neq$nil) do
       $T'.size \leftarrow T'.size - 1$
       $T' \leftarrow T'.root.parent$
   delete $T$
   ```

   Note that yet again, if we ended up deleting a node on the l-th layer, then the deletion (including with the initial `Find()`) costs $O(l)$, so climbing from that node through all of its ancestors and up to the root — to decrement each ancestor's *size* field — takes $O(l)$ in total.

Problem 1. (20 pts) The benefits of `InsertionSort`.
(a) Mischievous Milly took a sorted array A with $n+k$ elements, picked k elements arbitrarily, and exchanged those with *larger* elements. In other words, there exist k elements that are out of place in A and appear in *higher* positions than they ought to appear; but if we were to omit these k elements, the remaining n elements appear in a sorted order.
(An example with $n = 10$ and $k = 3$ is $A = [4, 8, 17, 19, 31, 9, 1, 33, 34, 72, 73, 77, 16]$, where the three elements $1, 9$ and 16, appear far higher than they should.)

Assume n is very large and that k is fairly small (say, $n = 10^6$ while $k = 20$). Argue the while $k = o(\log(n))$ is it faster to use InsertionSort in this case rather than any other algorithm we've seen in this course (namely, MergeSort, HeapSort, QuickSort). Prove your claims.
Answer. First note that regardless of the number of the few random elements, we know that merge sort, heap-sort, and quick-sort each take $\Omega(n \log n)$ time in best case.

Now, consider the execution of Insertion-Sort, which iterates from $j = 2$ to $j = n+k$ and places $A[j]$ in its right place among the first $j-1$ elements. Whenever $A[j]$ is one of the k elements moved, we might need to make $O(j)$-KC to put it in its right place. In the worst case this j is fairly large, and so for each such iteration we can make $O(n)$-KC. However, for each of the remaining n elements we only make a *single*-KC: any of the k elements that were moved to a place higher than j is irrelevant for this iteration; and when we reach the jth iteration all keys that were moved and placed in front of $A[j]$ are in their right place, and so is $A[j]$.

It follows that the number of key-comparisons we make on such an instance is $O(kn + n) = O(kn)$ (we assume $k \geq 1$). As long as $k \in o(\log(n))$ it follows that the number of KC is in $o(n \log(n))$.

(b) (10 pts) Malicious Mellisa took a sorted array A with $n+k$ elements, picked k elements arbitrarily, and exchanged their positions arbitrarily. In other words, there exist k elements that are out of place in A and may appear in *any* position; but if we were to omit these k elements, the remaining n elements appear in a sorted order.
(An example with $n = 10$ and $k = 3$ is $A = [16, 4, 8, 34, 9, 11, 17, 19, 1, 31, 33, 72, 73, 77]$, where by omitting the three elements $1, 16$ and 34, the remaining 10 elements are sorted.)

Assume n is very large and that k is fairly small (say, $n = 10^6$ while $k = 20$). Argue the while $k = o(\log(n))$ is it still faster to use InsertionSort in this case rather than any other algorithm we've seen in this course (namely, MergeSort, HeapSort, QuickSort). Prove your claims.
Hint: It is easier to use a different type of analysis here. Call the k items that were moved "bad" and the remaining n items "good". Bound the overall number of key-comparisons we make at the execution of `InsertionSort` by partitioning the key comparisons into two types: those where at least one of the two compared elements is bad, and those where both compared elements are good.
Answer. Again, sorting algorithms we studied in class, aside from Insertion Sort have all best-case runtime of $\Omega(n \log(n))$. We show that in this case too the runtime of `InsertionSort` is $O(kn)$ and the result follows.

As suggested in the hint, we partition the set of KC made during the entire execution of `InsertionSort` into two types: those with at least one bad element and those with both elements being good. Obviously, as there are k bad elements and each participates in at most $n-1$ KCs, then the overall number of KCs of the former type is $O(kn)$.

We argue that all in all, we make at most $O(n)$ KCs of the latter type. Again, look at each iteration from $j = 2$ to $j = n+k$, and in fact look at any j where $A[j]$ is a good element (all KC made in iterations where $A[j]$ is bad have to be of the first type). We argue that the first good-good KC in this iteration must also be the last. Indeed, if we compare the good key $A[j]$ with a different good key $A[j']$ for some $j' < j$ then we know $A[j'] \leq A[j]$ which implies `InsertionSort` terminates this iteration and then moves to the next element. Therefore, there are at most $O(n)$ KCs where both keys are good, and so the overall number of KCs is $O(kn + n) = O(kn)$.

Problem 2. (15 pts) Recall the definition of our beloved series from HW1: $a_0 = 0$, $a_1 = 1$, and for $n \geq 2$: $a_n = 2a_{n-1} + a_{n-2}$.

(a) (5 pts) Consider the following matrix: $M = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$. Prove by induction that for each $n \geq 1$:

$$M^n = \begin{bmatrix} a_{n-1} & a_n \\ a_n & a_{n+1} \end{bmatrix}$$

Answer. We prove by induction on n. The base case $n = 1$ is obvious given that $a_2 = 2 \cdot 1 + 0 = 2$. Induction step: Suppose the claim holds for some value of $n \geq 1$. For $n + 1$ we have:

$$\begin{aligned} M^{n+1} &= \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}^n = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} a_{n-1} & a_n \\ a_n & a_{n+1} \end{bmatrix} \\ &= \begin{bmatrix} a_n & a_{n+1} \\ a_{n-1} + 2a_n & a_n + 2a_{n+1} \end{bmatrix} = \begin{bmatrix} a_n & a_{n+1} \\ a_{n+1} & a_{n+2} \end{bmatrix} \quad \blacksquare \end{aligned}$$

(b) (10 pts) Design an $O(\log n)$-time algorithm to compute a_n. Justify the running time of your algorithm.
Answer. In class we have seen an algorithm that computes exponents in $O(\log n)$ time. So to compute M^n we first compute $M^{n/2}$ and then multiply the result by itself if n is even, or compute first M^{n-1} and multiply the results by M if n is odd. So each time, by at most two recursive calls the size of the problem is reduced from n to $\lfloor \frac{n}{2} \rfloor$. So it only needs $O(\log n)$ matrix multiplications to compute M^n. Since for each of these matrix multiplications we only need a constant number of scalar multiplications the running time is still $O(\log n)$.
(In particular, there's no need to use Strassen's algorithm, as each matrix multiplication is of a (2×2)-matrix with another (2×2)-matrix, which requires $8 = O(1)$ scalar multiplications. If you truly want to be nit-picking, observe that since M is symmetric then for any i it holds that M^i is symmetric; so the multiplication only requires 6 scalar multiplications since coordinate $(2, 1)$ will always be the same as coordinate $(1, 2)$.)
Once we have computed M^n, by part (a) we have found $F(n)$ as it is the element in coordinate $(2, 1)$ of M^n.

```
procedure Fib($n$)
$M \leftarrow \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$
$A \leftarrow$FastExp($M, n$)
return $A_{1,2}$
```

$**$ while there's **no need** to write the code for fast-exponentiation of matrices, we bring it here
$**$ just so you everything will be perfectly clear.

```
procedure FastExp($M, n$)
if ($n = 0$) then
    return $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
else if ($n$ is odd) then
    $X \leftarrow$FastExp($M, n-1$)
    return $\begin{bmatrix} M_{1,1}X_{1,1} + M_{1,2}X_{2,1} & M_{1,1}X_{1,2} + M_{1,2}X_{2,2} \\ M_{2,1}X_{1,1} + M_{2,2}X_{2,1} & M_{2,1}X_{1,2} + M_{2,2}X_{2,2} \end{bmatrix}$ ** returning $M \cdot X$
else
    $X \leftarrow$FastExp($M, n/2$)
    return $\begin{bmatrix} X_{1,1}X_{1,1} + X_{1,2}X_{2,1} & X_{1,1}X_{1,2} + X_{1,2}X_{2,2} \\ X_{2,1}X_{1,1} + X_{2,2}X_{2,1} & X_{2,1}X_{1,2} + X_{2,2}X_{2,2} \end{bmatrix}$ ** returning $X \cdot X$
```

Problem 3. (20 pts) In this question, we will discuss multiplying a $(n \times n)$-matrix M with a n-dimensional vector $\bar{v}$.
(a) (5 pts) Describe the naïve loop-based algorithm for computing $M\bar{v}$ and argue it runs in time $\Theta(n^2)$.
Answer.

```
procedure Multiply($M, \bar{v}, n$)
** precondition: $M$ is a matrix of size $n \times n$ and $\bar{v}$ is a vector of size $n$
$res \leftarrow$ a vector of size $n$
for ($i$ from 1 to $n$) do
    $res_i \leftarrow 0$
    for ($j$ from 1 to $n$) do
        $res_i \leftarrow res_i + M_{i,j} \times v_j$
return $res$
```

For each of the n coordinates of the output, the naïve algorithm makes n multiplications: the n scalars in the suitable row of M with the n coordinates of $\bar{v}$. It is therefore evident that the runtime of the algorithm is in $O(n) \cdot O(n) = O(n^2)$.

(b) (15 pts) We define a series of specific matrices as follows. The k-th matrix in our series is a $(2^k \times 2^k)$-matrix defined by the following recursion: $M_0 = [1]$ which is a (1×1)-matrix, and $M_{k+1} = \begin{bmatrix} M_k & M_k \\ M_k & -2M_k \end{bmatrix}$ for $k \geq 0$.

So, for example, $M_1 = \begin{bmatrix} 1 & 1 \\ 1 & -2 \end{bmatrix}$ and $M_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -2 & 1 & -2 \\ 1 & 1 & -2 & -2 \\ 1 & -2 & -2 & 4 \end{bmatrix}$, and M_3 is a (8×8)-matrix.

Given n which is a power of 2 (i.e., $n = 2^k$ for some k), describe an algorithm whose input is a n-dimensional vector $\bar{v}$, and returns the product of the matrix M_k with the given vector, i.e. $M_k \cdot \bar{v}$.
Explain why your algorithm is correct and prove that its runtime is $O(n\log(n))$.
You may use the fact (without proving it) that adding/subtracting two n-dimensional vectors takes $O(n)$ time.
Answer. Our algorithm is a divide and conquer algorithm.

For inputs of dimension 1, the algorithm simply returns $\bar{v}$. (Note, $H_0 \cdot \bar{v} = 1 \cdot \bar{v} = \bar{v}$).
For inputs of dimension 2^k for some $k \geq 1$, the algorithm partitions v into two parts:

$$\bar{v} = \begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix}$$

Observe that this implies that:

$$H_k \cdot \bar{v} = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -2H_{k-1} \end{bmatrix} \cdot \begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix} = \begin{bmatrix} H_{k-1} \cdot \bar{x} + H_{k-1} \cdot \bar{y} \\ H_{k-1} \cdot \bar{x} - 2H_{k-1} \cdot \bar{y} \end{bmatrix}$$

So our algorithm finds $\bar{a} = H_{k-1} \cdot \bar{x}$ and $\bar{b} = H_{k-1} \cdot \bar{y}$ using two recursive calls, and then — using one addition and one subtraction — returns the vector $\begin{bmatrix} \bar{a} + \bar{b} \\ \bar{a} - 2\bar{b} \end{bmatrix}$. Let $T(n)$ denote the algorithm's runtime over inputs of dimension n, then due to the two recursive calls on $\frac{n}{2}$-dimensional instances, and the $O(n)$ time operations we need to add and subtract (and copy into the right place) the vectors $\bar{a} + \bar{b}$ and $\bar{a} - 2\bar{b}$ we have that

$$T(n) = \begin{cases} 1 & , \text{ if } n = 1 \\ 2T(\frac{n}{2}) + n & , \text{ if } n \geq 2 \end{cases}$$

Master Theorem (case 2) is applicable to this recursion (in fact, this is the same recursion we get in `MergeSort`), and so we get that $T(n) = \Theta(n \log(n))$.

Problem 4. (45 pts) Various Sorting Problems.

(a) (5 pts) How would you merge two heaps of size n each into a single heap of size $2n$ at the best asymptotic runtime?

Answer. Here is a simple algorithm: copy all $2n$ elements into a single array and then run `Build-Heap` on the one array. Seeing as `Build-Heap` takes $O(2n) = O(n)$ time, this takes $O(n)$-time overall (including copying the elements into a new array).

Note however that in any case we have to store the entire new heap in a $2n$-length array, or, at the very least, read all $2n$ elements in both heaps. This means the runtime of our algorithm must by $\Omega(n)$. Thus, our algorithm, despite its naïvity, is efficient.

(b) (10 pts) Suppose we have two sorted arrays A and B each of which contains n integers, for a total of $2n$ elements. Give an algorithm with time $O(\log n)$ which finds the median of these $2n$ elements. (I.e., the element x s.t $\#\{y \in A \cup B : \ y \le x\} = n$.) You may assume all integers are unique.

Answer. It is evident that when $n = 1$ then the minimum of $A[1]$ and $B[1]$ must be the median. Otherwise, if $n > 1$, the our algorithm compares the middle element in A with the middle element in B. Without loss of generality $A[\lfloor \frac{n+1}{2} \rfloor] > B[\lfloor \frac{n+1}{2} \rfloor]$. This means that all elements in $A[\lfloor \frac{n+1}{2} \rfloor + 1, \ldots, n]$ can't be the median - they are strictly greater than $\lfloor \frac{n+1}{2} \rfloor$ elements in A and additional $\lfloor \frac{n+1}{2} \rfloor$ elements in B, so overall they are strictly greater than $2\lfloor \frac{n+1}{2} \rfloor \ge 2\left(\frac{n+1}{2} - \frac{1}{2}\right) = n+1-1 = n$ elements. Similarly, all elements in $B[1, \ldots, \lceil \frac{n+1}{2} \rceil - 1]$ can't be the median as they are strictly smaller than $n - \lceil \frac{n+1}{2} \rceil + 1$ elements in B and from $n - \lfloor \frac{n+1}{2} \rfloor + 1$ elements in A, over all there are $2n+2-\lceil \frac{n+1}{2} \rceil - \lfloor \frac{n+1}{2} \rfloor = 2n+2-(n+1) = n+1$ elements strictly greater than them. We thus recurse on $A[1, \ldots \lfloor \frac{n+1}{2} \rfloor]$ and $B[\lceil \frac{n+1}{2} \rceil, \ldots, n]$.
(Of course, if we had $A[\lfloor \frac{n+1}{2} \rfloor] < B[\lfloor \frac{n+1}{2} \rfloor]$ then we'd recurse on $A[\lceil \frac{n+1}{2} \rceil, \ldots, n]$ and $B[1, \ldots \lfloor \frac{n+1}{2} \rfloor]$.)

The algorithm in pseudo-code is:

```
MyMedian($A, b_A, e_A, B, b_B, e_B$)
    ** $A$ and $B$ are two sorted arrays,
    ** $f_A, l_A$ are the first and last indices (resp.) in the subarray of $A$ we consider
    ** $f_B, l_B$ are the first and last indices (resp.) in the subarray of $B$ we consider

if ($f_A = l_A$ and $f_B = l_B$) then
    return $\min\{A[f_A], B[f_B]\}$
else
    $m_A \leftarrow \lfloor \frac{f_A + l_A}{2} \rfloor$
    $m_B \leftarrow \lfloor \frac{f_B + l_B}{2} \rfloor$
    if ($A[m_A] > B[m_b]$) then
        return MyMedian($A, f_A, m_A, B, \lceil \frac{f_B + l_B}{2} \rceil, l_b$)
    else
        return MyMedian($A, \lceil \frac{f_A + l_A}{2} \rceil, l_A, B, f_B, m_B$)
    end if
end if
```

Each recursive call makes a constant number of operations and it is obvious that with each recursive call the number of elements left in each array is $\le \frac{n+1}{2}$. Thus, after $O(\log(n))$ calls we are down to a single element in each array, and the recursion terminates. All in all, it takes $O(\log(n))$ time.

(c) (10 pts) Propose an efficient algorithm for the following problem. The input is composed of a heap A of size n, and an additional element x. The algorithm's objective is to return (say, print to the screen) all elements in A that are greater than or equal to x (note that x is not necessarily in A). The running time of your algorithm should be $O(k)$ where k is the number of elements reported by the algorithm. Prove the correctness and runtime of the algorithm.

Answer. The algorithm will be a recursive traversal of the tree (heap). At each node, since we know that the largest element of that subtree is at the root, by comparing the root with x we can determine whether we must explore that subtree or not. If the root of that subtree is smaller than x then all the elements of that subtree are smaller than x. Otherwise, we will print the root and recursively check each of the subtrees rooted at the children of the current node. Here is the psuedocode (we call it with $i = 1$):

```
Report-All($A, i, x$)
    ** printss all the elements $\geq x$ in the heap rooted at $A[i]$.
if ($A[i] \geq x$) then
    Print($A[i]$)
    $lc \leftarrow leftchild(i)$
    $rc \leftarrow rightchild(i)$
    if ($lc \leq heapsize(A)$) then
        Report-All($A, lc, x$)
    end if
    if ($rc \leq heapsize(A)$) then
        Report-All($A, rc, x$)
    end if
end if
```

The correctness of the algorithm is easy. Assume for the sake of contradiction that there exists some element $A[j] \geq x$ which we have not printed. This can happen only if `Report-All` was never invoked on j. Thus, along the path from j to the root of the heap there was some j' for which $A[j'] < x$. But this implies $A[j'] < A[j]$, which means A isn't a heap — in the subheap rooted at j', the root *isn't* the largest element. Contradiction.

To argue that we take $O(k)$ time to print, we denote a function $T(k)$ which is the runtime of the algorithm on a rooted heap with exactly k elements greater than x. Clearly, $T(0) = c$ for some constant. Moreover, $T(k) = c' + T(k_1) + T(k_2)$ where c' is some other constant, and k_1 and k_2 denote the number of elements greater than x is the left and in the right sub-heaps of the root. Clearly, $k_1, k_2 \geq 0$ and $k_1 + k_2 \leq k - 1$. Denoting $C = \max\{c, c'\}$, we can prove using induction that $T(k) \leq C \cdot (2k + 1)$ which results in $T(k) \in O(k)$.
Base case: For $k = 0$ have $T(0) = c \leq C = C(2 \times 0 + 1)$.
Induction step: Fix some k. Assuming the claim holds for any $0 \leq i < k$ we show it holds for k. Indeed:

$$\begin{aligned} T(k) &= c' + T(k_1) + T(k_2) \\ &\leq C + C \cdot (2k_1 + 1) + C \cdot (2k_2 + 1) \\ &= C \cdot (2(k_1 + k_2) + 3) = C \cdot (2(k - 1) + 3) = C \cdot (2k + 1) \quad \blacksquare \end{aligned}$$

(d) (10 pts) Give an algorithm using the techniques and data structures you have seen in the course so far, for the following problem. The input is an array A of size n of integers and some $k \leq n$. The output is the k smallest elements of A. Your algorithm must run in $O(n + k \log(n))$. Explain your algorithm and analyze its running time. What is the asymptotic upper bound on k for which your algorithm is still linear in n?

Answer. Given array A first we make a max-heap on this array by calling the `Build-Min-Heap` procedure we have seen for the heap-sort algorithm. By definition of heap, the smallest element of A is located at $A[1]$. So `Extract-Min` returns the smallest element of A. We keep calling `Extract-Min` on A iteratively for k times to find the largest elements of A among the remaining $n - (k - 1)$ elements and return it. So the algorithm will be as follows:

```
Find-Smallest($A, n, k$)
    ** $Answer[1 \ldots k]$ will hold the solution
    Build-Min-Heap($A$)
    for $i \longleftarrow 1$ to $k$ do
        $Answer[i] \longleftarrow$ Extract-Min($A$)
    end for
```

We have seen that Build-Heap takes $O(n)$ time and each call to Extract-Min on a heap takes $O(\lg n)$. Since we invoke Extract-Min k times, the running time of the loop is at most $O(k \lg n)$. Thus the total running time is $O(n + k \log n)$. The largest value of k for which this is still linear if $k \in O(\frac{n}{\log n})$.

(e) (10 pts)

1. (5pt) Given a BST T we wish to add a field *size* to each node u so that $u.size$ stores the number of nodes in the subtree rooted at u.
   Give an algorithm that takes a tree and assigns each u its correct *size* field. What is the runtime of your algorithm?

   Answer. It is clear that for any BST T we have $T.size = T.leftchild.size + T.rightchild.size + 1$, and it is also clear that to update the *size* field of each of the n nodes in the tree we must traverse all of them, so the runtime has to be $\Omega(n)$. The point is that we can do in one single visit — using *post-order traversal*. The code is just a quick adaptation of the post-order traversal, where we just invoke $T.root.size \leftarrow$Size-PostOrder(T).

   ```
   procedure Size-PostOrder($T$)
   if ($T$ =nil) then
       return 0
   $n_L$ ←procedure Size-PostOrder($T.leftchild$)
   $n_R$ ←procedure Size-PostOrder($T.righttchild$)
   $T.size \leftarrow n_L + n_R + 1$
   return $T.size$
   ```

   Correctness now follows from induction (the base case is an empty tree of 0 nodes) and the fact that $T.size = T.leftchild.size + T.rightchild.size + 1$. The runtime is identical to the (asymptotic) runtime of post-order traversal – $\Theta(n)$ (for each node of T we do $O(1)$ operations).

2. (5pt) Given a BST T we wish to add a field *quantile* to each node u so that $u.quantile$ stores the overall number of nodes in all of T whose keys is $\leq u.key$.
   Give an algorithm that takes a tree and assigns each u its correct *quantile* field. What is the runtime of your algorithm?

   Answer. Here, the tree-traversal that solves the problem is the in-order traversal. Afterall, if some node is that kth node overall (hence its *quantile* field should be set to k), that means that in the in-order traversal of T it is also the kth element to be printed. So instead of printing, we will just use a global counter *count* which is initially 0. (Namely, we invoke the function with T being the overall tree and $count = 0$.) We thus modify the in-order traversal to be as follows:

   ```
   procedure Quantile($T, count$)
   if ($T$ ≠nil) then
       Quantile($T.leftchild, count$)
       $count \leftarrow count + 1$
       $T.root.quantile \leftarrow count$
       Quantile($T.rightchild, count$)
   ```

   We argue that for every node in T we set its quantile field to the true quantile (the number of keys $\leq T.root$). The proof uses induction, and while it can be proven in multiple ways, we choose induction on the size of the tree. Formally we argue that for every tree of size n the function Quantile(T, i)

sets all *quantile* fields of all nodes in the tree to their true quantile $+i$. In particular, this implies that `Quantile(`$T, 0$`)` sets the quantile field to the true quantile.

For a tree of size 1 (a root with two `nil` children) it is clear that `Quantile(`T, i`)` sets the *quantile* field of the root to $i + 1$.

For a tree of size n, under the assumption that for any tree of size $m < n$ `Quantile(`T, i`)` sets all *quantile* fields of all nodes in the tree to their true quantile $+i$, we have that

- For every node in the left subtree of the root, we set its *quantile* field to the node's true quantile $+i$, just by the IH.
- The root's *quantile* field is set to $i+$the number of nodes in the left subtree, which is the root's true quantile.
- For every node in the right subtree of the root, we set its *quantile* field to; $i+$ the number of keys in the left subtree $+1$ (for the root) $+$ the number of keys in the right subtree which are $\leq$ this node's key ($=$ the quantile value of the node in the right subtree). By definition, this is the number of keys in the overall tree which are $\leq$ the node's key.