1 Question 1

1.1 a.

Given a node of index i with m children per node, the last index of i's child is given by $i * m + 1$. To get the first index of i's child, we subtract $(m - 1)$ because we are removing all children except for the first one. Let $p =$ parent index. Then this gives the equation:

$i = p * m + 1 - (m - 1)$
$i = p * m + 1 - m + 1$
$i = p * m - m + 2$

Now we solve for the parent index which gives:

$p = \frac{i+m-2}{m}$

We now take the floor of this because the next whole integer will not appear until we reach the next node's children. This gives:

Algorithm 1 M-ARY-PARENT(i)

return $\left\lfloor \frac{i+m-2}{m} \right\rfloor$

Now to get the child node, we use the same idea as above, but instead of subtracting $(m-1)$ we subtract $(m - j)$ because we want to find the j'th child given m children. Given i as the parent and j as i's j'th child, we have:

$child = i * m + 1 - (m - j)$
$child = i * m + 1 - m + j$
$child = i(m - 1) + 1 + j$

Algorithm 2 M-ARY-Child(i,j)

return $i(m + 1) + 1 + j$

1.2 b.

Given that each node will split m times subsequent level of the tree, we have that the $height = log_m(n)$.

1.3 c.

For our algorithm, we need to swap the first (largest) element of the heap with the last (smallest) element of the heap. Then we run MAX-HEAPIFY on the root to ensure the

max-heap properties still hold.

Algorithm 3 Extract-max(A,i,n,m)

```
max = A[i]
exchange A[i] with A[n]
n = n − 1
MAX-HEAPIFY(A, i, n, m)
return max
```

Our MAX-HEAPIFY method will be very similar to a binary MAX-HEAPIFY method except for the fact that we will need to iterate through all of the children to find the maximum rather than just looking at the left and right child.

For the runtime of Extract-max, all of the calls are O(1) except for MAX-HEAPIFY. For a binary heap, MAX-HEAPIFY has a runtime of $\theta(lg(n))$ because we are checking each child of a node and ensuring the max-heap properties hold. It follows that for a MAX-HEAPIFY method of m children, we have a runtime of $\theta(log_m(n))$.

1.4 d.

For our algorithm, we need to insert the element to the last index of the heap and then percolate up.

Algorithm 4 Insert(A, key)

```
heapSize = heapSize + 1;
A[heapSize] = key;
PercolateUp(A, heapsize, m)
```

For the runtime of Insert, all the calls are O(1) except for the PercolateUp call. For a binary heap, PercolateUp has a runtime of $\theta(lg(n))$ because it will swap the key until it reaches the a location that makes the heap properties hold. For a heap with m children, the idea is the same except our runtime will be $\theta(log_m(n))$.

2 Question 2

For this problem, I assumed a min heap. The idea for this is very analogous to searching a balanced binary search tree. Suppose we take at an arbitrary node of index i. Then, because we are assuming a min heap, i's children will be greater than or equal i. This tells us that if the value of i is greater than the key, we do not have to search all of the children of i because we already know all those values will be greater than the key. This implies that we will only be searching for the nodes less than or equal to our key because if any node is greater than our key, we can remove that node and all of its children. This idea leads to the following

algorithm with has a runtime of $O(p)$.

Algorithm 5 ArrayList findKeyElements(int[] A, int key, Arraylist aList, int index)

```
n = length A
if ((index < n and A[index] <= key) then
    add A[index] to aList
    left = 2 * index
    right = 2 * index + 1
    findKeyElements(A, key, aList, left)
    findKeyElements(A, key, aList, right)
end if
return aList
```

3 Question 3

3.1 a.

Let i be any element in the diagonal from the bottom left to the top right of the array. Then by the constraints of the array, any element above i will be less than i and any element below will be greater than i. Additionally, any element to the left of i is less than i and any element to the right is greater than i. Now let l be the first element of the diagonal (bottom left of the array). If we compare the key to l and see that the key is greater than l, we must increase the current value by moving one index to the right. Similarly, If we see that the key is less than l, we must decrease the current value by moving one index up. We then repeat this process until the key is found or we reach the boundary of the array. This leads to the following algorithm.

Algorithm 6 findKey(int[][] A, int n, int key)

```
i = n
j = 0
while (j <= n) do
  if (i < 0) then
    print(key not found)
    break
  end if
  if (A[i][j] == key) then
    print((i,j))
    break
  else if (A[i][j] > key) then
    i = i - 1
  else
    j = j + 1
  end if
end while
```

This algorithm's runtime is less than $\theta(n^2)$ because we do no have to search every element of the matrix. Instead, we are searching, at most, $2n$ elements because in the worst case, index i will iterate from $n...0$ and index j will iterate from $0...n$. This implies that our algorithm is $\theta(n)$.

To prove the correctness of this algorithm, we must construct a loop invariant and show that it is true before, during, and after each iteration. At each step, elements to the left and elements above of the current index will be less than the value of the current index. This will be our loop invariant because, from the logic above, this is always true independent of the current index and the elements in the array (assuming a "correct" array). In the first instance, we start at the bottom left of the array. Clearly there are no elements to the left and all elements above are less than our current value so our initialization step is true. Next we compare the current element to the key. In either case (move one index up or move one index to the right), due to the sortedness of the array, all elements to the left and above our current index will be less than the current index. This suggest that our maintenance step holds. In the worst case where no key is found, our final index will be on the right side of the array. Again, from the sortedness of the array, our loop invariant still holds in this stage which shows that our termination step holds as well. It follows that for any index we are currently at, the loop invariant is true which shows that our algorithm is correct.