Chapter 6

Michelle Bodnar, Andrew Lohr

May 22, 2017

Exercise 6.1-1

At least 2^h and at most $2^{h+1} - 1$. Can be seen because a complete binary tree of depth $h - 1$ has $\Sigma_{i=0}^{h-1} 2^i = 2^h - 1$ elements, and the number of elements in a heap of depth h is between the number for a complete binary tree of depth $h - 1$ exclusive and the number in a complete binary tree of depth h inclusive.

Exercise 6.1-2

Write $n = 2^m - 1 + k$ where m is as large as possible. Then the heap consists of a complete binary tree of height $m - 1$, along with k additional leaves along the bottom. The height of the root is the length of the longest simple path to one of these k leaves, which must have length m. It is clear from the way we defined m that $m = \lfloor \lg n \rfloor$.

Exercise 6.1-3

If there largest element in the subtee were somewhere other than the root, it has a parent that is in the subtree. So, it is larger than it's parent, so, the heap property is violated at the parent of the maximum element in the subtree

Exercise 6.1-4

The smallest element must be a a leaf node. Suppose that node x contains the smallest element and x is not a leaf. Let y denote a child node of x. By the max-heap property, the value of x is greater than or equal to the value of y. Since the elements of the heap are distinct, the inequality is strict. This contradicts the assumption that x contains the smallest element in the heap.

Exercise 6.1-5

Yes, it is. The index of a child is always greater than the index of the parent, so the heap property is satisfied at each vertex.

Exercise 6.1-6

No, the array is not a max-heap. 7 is contained in position 9 of the array, so its parent must be in position 4, which contains 6. This violates the max-heap property.

Exercise 6.1-7

It suffices to show that the elements with no children are exactly indexed by $\{\lfloor n/2 \rfloor + 1, \dots, n\}$. Suppose that we had an i in this range. It's childeren would be located at $2i$ and $2i+1$ but both of these are $\geq 2\lfloor n/2 \rfloor + 2 > n$ and so are not in the array. Now, suppose we had an element with no kids, this means that $2i$ and $2i+1$ are both $> n$, however, this means that $i > n/2$. This means that $i \in \{\lfloor n/2 \rfloor + 1, \dots, n\}$.

Exercise 6.2-1

27	17	3	16	13	10	1	5	7	12	4	8	9	0
27	17	10	16	13	3	1	5	7	12	4	8	9	0
27	17	10	16	13	9	1	5	7	12	4	8	3	0

Exercise 6.2-2

Algorithm 1 MIN-HEAPIFY(A,i)

```
1: $l = LEFT(i)$
2: $r = RIGHT(i)$
3: if $l \leq A.heap-size$ and $A[l] < A[i]$ then
4:     $smallest = l$
5: else $smallest = i$
6: end if
7: if $r \leq A.heap-size$ and $A[r] < A[smallest]$ then
8:     $smallest = r$
9: end if
10: if $smallest \neq i$ then
11:     exchange $A[i]$ with $A[smallest]$
12:     MIN-HEAPIFY($A, smallest$)
13: end if
```

The running time of MIN-HEAPIFY is the same as that of MAX-HEAPIFY.

Exercise 6.2-3

The array remains unchanged since the if statement on line line 8 will be false.

Exercise 6.2-4

If $i > A.heap - size/2$ then l and r will both exceed $A.heap - size$ so the if statement conditions on lines 3 and 6 of the algorithm will never be satisfied. Therefore $largest = i$ so the recursive call will never be made and nothing will happen. This makes sense because i necessarily corresponds to a leaf node, so MAX–HEAPIFY shouldn't alter the heap.

Exercise 6.2-5
Iterative Max Heapify(A, i)

```
while $i < A$.heap-size do
    $l$ =LEFT($i$)
    $r$ =LEFT($i$)
    $largest = i$
    if $l \leq A$.heap-size and $A[l] > A[i]$ then
        $largest = l$
    end if
    if $l \leq A$.heap-size and $A[r] > A[i]$ then
        $largest = r$
    end if
    if $largest \neq i$ then
        exchange $A[i]$ and $A[largest]$
    elsereturn A
    end if
end while
return A
```

Exercise 6.2-6

Consider the heap resulting from A where $A[1] = 1$ and $A[i] = 2$ for $2 \leq i \leq n$. Since 1 is the smallest element of the heap, it must be swapped through each level of the heap until it is a leaf node. Since the heap has height $\lfloor \lg n \rfloor$, MAX-HEAPIFY has worst-case time $\Omega(\lg n)$.

Exercise 6.3-1

5	3	17	10	84	19	6	22	9
5	3	17	22	84	19	6	10	9
5	3	19	22	84	17	6	10	9
5	84	19	22	3	17	6	10	9
84	5	19	22	3	17	6	10	9
84	22	19	5	3	17	6	10	9
84	22	19	10	3	17	6	5	9

Exercise 6.3-2

If we had started at 1, we wouldn't be able to guarantee that the max-heap property is maintained. For example, if the array A is given by [2,1,1,3] then MAX-HEAPIFY won't exchange 2 with either of it's children, both 1's. However, when MAX-HEAPIFY is called on the left child, 1, it will swap 1 with 3. This violates the max-heap property because now 2 is the parent of 3.

Exercise 6.3-3

All the nodes of height h partition the set of leaves into sets of size between $2^{h-1}+1$ and 2^h, where all but one is size 2^h. Just by putting all the children of each in their own part of trhe partition. Recall from 6.1-2 that the heap has height $\lfloor \lg(n) \rfloor$, so, by looking at the one element of this height (the root), we get that there are at most $2^{\lfloor \lg(n) \rfloor}$ leaves. Since each of the vertices of height h partitions this into parts of size at least $2^{h-1}+1$, and all but one corresponds to a part of size 2^h, we can let k denote the quantity we wish to bound, so,

$$(k-1)2^h + k(2^{h-1}+1) \leq 2^{\lfloor \lg(n) \rfloor} \leq n/2$$

so

$$k \leq \frac{n+2^h}{2^{h+1}+2^h+1} \leq \frac{n}{2^{h+1}} \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

Exercise 6.4-1

5	13	2	25	7	17	20	8	4
5	13	20	25	7	17	2	8	4
5	25	20	13	7	17	2	8	4
25	5	20	13	7	17	2	8	4
25	13	20	5	7	17	2	8	4
25	13	20	8	7	17	2	5	4
4	13	20	8	7	17	2	5	25
20	13	4	8	7	17	2	5	25
20	13	17	8	7	4	2	5	25
5	13	17	8	7	4	2	20	25
17	13	5	8	7	4	2	20	25
2	13	5	8	7	4	17	20	25
13	2	5	8	7	4	17	20	25
13	8	5	2	7	4	17	20	25
4	8	5	2	7	13	17	20	25
8	4	5	2	7	13	17	20	25
8	7	5	2	4	13	17	20	25
4	7	5	2	8	13	17	20	25
7	4	5	2	8	13	17	20	25
2	4	5	7	8	13	17	20	25
5	4	2	7	8	13	17	20	25
2	4	5	7	8	13	17	20	25
4	2	5	7	8	13	17	20	25
2	4	5	7	8	13	17	20	25

Exercise 6.4-2

We'll prove the loop invariant of HEAPSORT by induction:

Base case: At the start of the first iteration of the for loop of lines 2-5 we have $i = A.length$. The subarray $A[1..n]$ is a max-heap since BUILD-MAX-HEAP(A) was just called. It contains the n smallest elements, and the empty subarray $A[n+1..n]$ trivially contains the 0 largest elements of A in sorted order.

Suppose that at the start of the i^{th} iteration of of the for loop of lines 2-5, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$ and the subarray $A[i+1..n]$ contains the $n-i$ largest elements of $A[1..n]$ in sorted order. Since $A[1..i]$ is a max-heap, $A[1]$ is the largest element in $A[1..i]$. Thus it is the $(n-(i-1))^{th}$ largest element from the original array since the $n-i$ largest elements are assumed to be at the end of the array. Line 3 swaps $A[1]$ with $A[i]$, so $A[i..n]$ contain the $n-i+1$ largest elements of the array, and $A[1..i-i]$ contains the $i-1$ smallest elements. Finally, MAX-HEAPIFY is called on $A, 1$. Since $A[1..i]$ was a max-heap prior to the iteration and only the elements in positions 1 and i were swapped, the left and right subtrees of node 1, up to node $i-1$, will be max-heaps. The call to MAX-HEAPIFY will place the element now located at node 1 into the correct position and restore the max-heap property so that $A[1..i-1]$ is a max-heap. This concludes the next iteration, and we have verified each part of the loop invariant. By induction, the loop invariant holds for all iterations.

After the final iteration, the loop invariant says that the subarray $A[2..n]$ contains the $n-1$ largest elements of $A[1..n]$, sorted. Since $A[1]$ must be the n^{th} largest element, the whole array must be sorted as desired.

Exercise 6.4-3

If it's already sorted in increasing order, doing the BUILD-MAX-HEAP call on line 1 will take $\Theta(n\lg(n))$ time. As we call MAX-HEAPIFY(A,i) in BUILD-HEAP, we know that $A[i]$ is larger than all elements to its right. This means that the time it takes will be $\lfloor \lg(n) - \lg(i) \rfloor$. So the total time to build the heap will be

$$\sum_{i=1}^{i=\lfloor n/2 \rfloor} \lg(n) - \lg(i) \in \Theta(n\lg(n))$$

There will be n iterations of the for loop in HEAPSORT, each taking $\Theta(\lg(n))$ time because the element that was at position i was the smallest and so will have $\lfloor \lg(n) \rfloor$ steps when doing MAX-HEAPIFY on line 5. So, it will be $\Theta(n\lg(n))$ time.

If it's already sorted in decreasing order, then the call on line one will only take $\Theta(n)$ time, since it was already a max heap to begin with. This is because elements in earlier positions correspond to begin further up in the heap, and since the array is descending, those have larger values. So, each of the MAX-HEAPIFY calls in the BUILD-MAX-HEAP procedure will only take constant time since the check on line 8 of MAX-HEAPIFY will always be false. It will still take $n\lg(n)$ peel off the elements from the heap and re-heapify. This is because each time we swap the head of the heap with the last element, we now have the smallest element at the head, so it will have to have as many recursive calls of MAX-HEAPIFY as the depth of the heap, which is about $\lg(n)$.

Exercise 6.4-4

Consider calling HEAPSORT on an array which is sorted in decreasing order. Every time $A[1]$ is swapped with $A[i]$, MAX-HEAPIFY will be recursively called a number of times equal to the height h of the max-heap containing the elements of positions 1 through $i-1$, and has runtime $O(h)$. Since there are 2^k nodes at height k, the runtime is bounded below by

$$\sum_{i=1}^{\lfloor \lg n \rfloor} 2^i \log(2^i) = \sum_{i=1}^{\lfloor \lg n \rfloor} i2^i = 2 + (\lfloor \lg n \rfloor - 1)2^{\lfloor \lg n \rfloor} = \Omega(n \lg n).$$

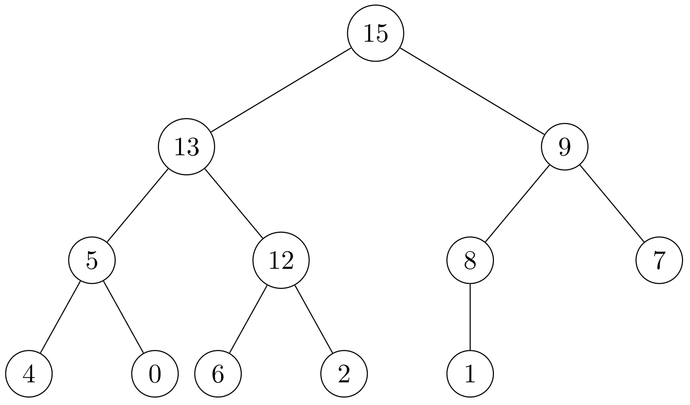
Exercise 6.4-5

Since the call on line one could possibly take only linear time (if the input was already a max-heap for example), we will focus on showing that the for loop takes n log n time. This is the case because each time that the last element is placed at the beginning to replace the max element being removed, it has to go through every layer, because it was already very small since it was at the bottom level of the heap before.

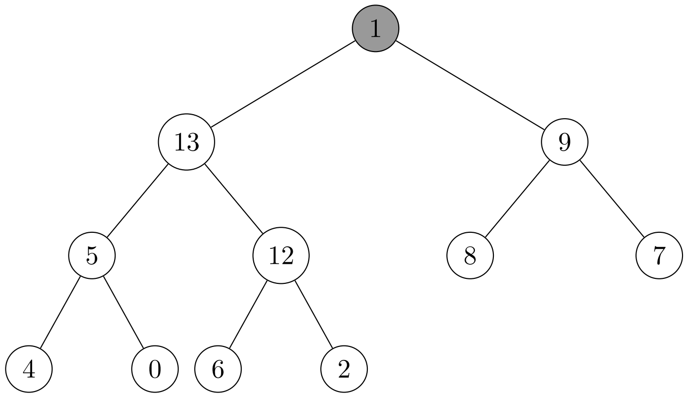
Exercise 6.5-1

The following sequence of pictures shows how the max is extracted from the heap.

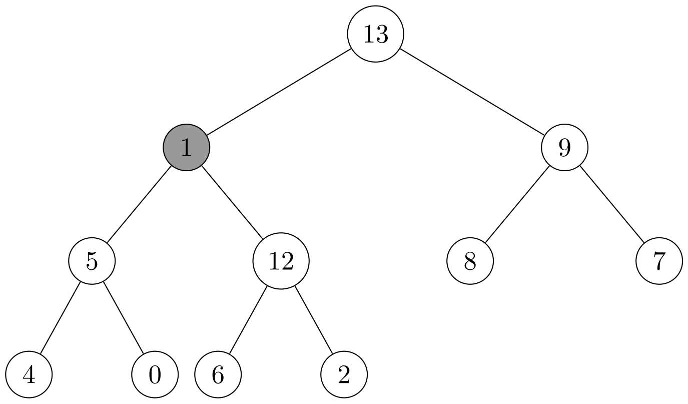
1. Original heap:



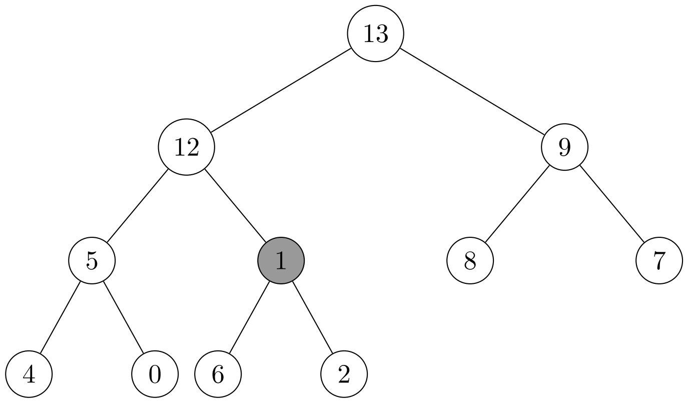
2. we move the last element to the top of the heap



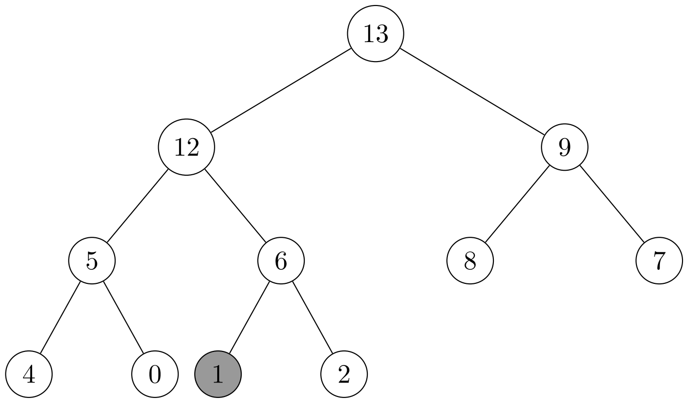
3. $13 > 9 > 1$ so, we swap 1 and 13.



4. Since $12 > 5 > 1$, we swap 1 and 12.



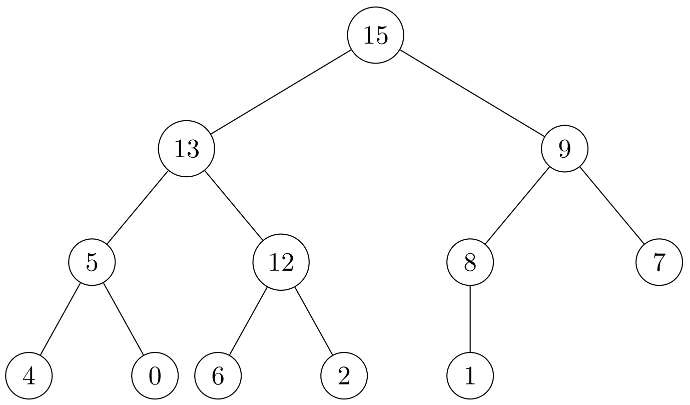
5. Since $6 > 2 > 1$, we swap 1 and 6.



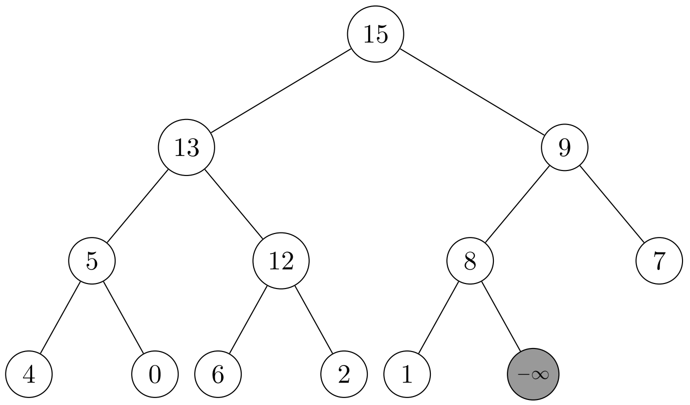
Exercise 6.5-2

The following sequence of pictures shows how 10 is inserted into the heap, then swapped with parent nodes until the max-heap property is restored. The node containing the new key is heavily shaded.

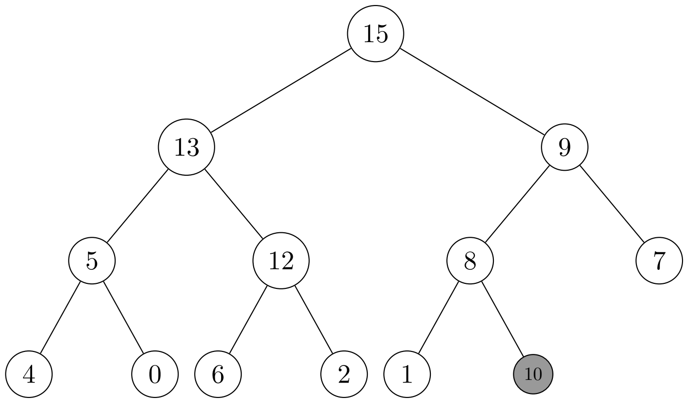
1. Original heap:



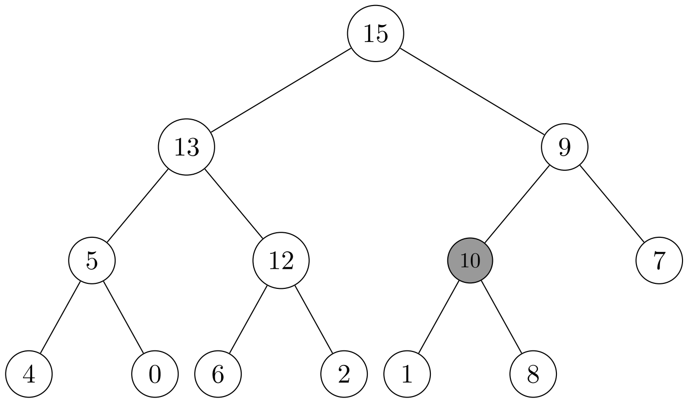
2. MAX-HEAP-INSERT(A,10) is called, so we first append a node assigned value $-\infty$:



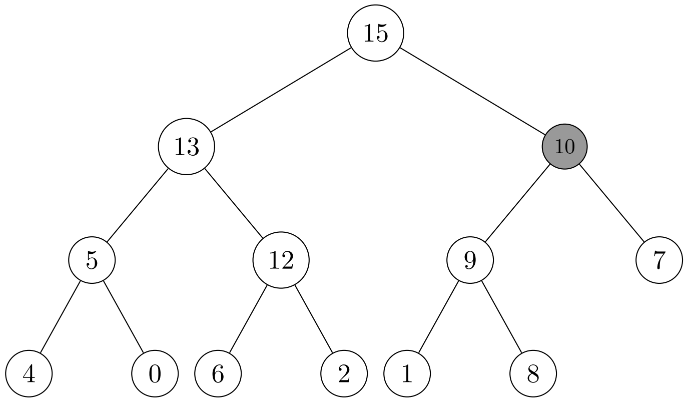
3. The key value of the new node is updated:



4. Since the parent key is smaller than 10, the nodes are swapped:



5. Since the parent node is smaller than 10, the nodes are swapped:



Exercise 6.5-3

Heap-Minimum(A)

```
1: return A[1]
```

Heap-Extract-Min(A)
Heap-decrease-key(A,i,key)
Min-Heap-Insert(A,key)

Exercise 6.5-4

If we don't make an assignment to $A[A.heap-size]$ then it could contain any value. In particular, when we call HEAP-INCREASE-KEY, it might be the case that $A[A.heap-size]$ initially contains a value larger than key, causing an error. By assigning $-\infty$ to $A[A.heap-size]$ we guarantee that no error will

```
if A.heap-size < 1 then
    Error “heap underflow”
end if
$min = A[1]$
$A[1] = A[A.heap - size]$
$A.heap - size - -$
Min-heapify(A,1)
return min
```

```
if key ¿ A[i] then
    Error “new key larger than old key”
end if
$A[i] = key$
while $i > 1$ and $A[Parent(i)] < A[i]$ do
    exchange $A[i]$ with $A[Parent(i)]$
    $i = Parent(i)$
end while
```

occur. However, we could have assigned any value less than or equal to *key* to $A[A.heap - size]$ and the algorithm would still work.

Exercise 6.5-5

Initially, we have a heap and then only change the value at i to make it larger. This can't invalidate the ordering between i and it's children, the only other thing that needs to be related to i is that i is less than it's parent, which may be false. Thus we have the invariant is true at initialization. Then, when we swap i with its parent if it is larger, since it is larger than it's parent, it must also be larger than it's sibling, also, since it's parent was initially above its kids in the heap, we know that it's parent is larger than it's kids. The only relation in question is then the new i and it's parent. At termination, i is the root, so it has no parent, so the heap property must be satisfied everywhere.

Exercise 6.5-6

Replace $A[i]$ by *key* in the while condition, and replace line 5 by “$A[i] = A[PARENT(i)]$.” After the end of the while loop, add the line $A[i] = key$. Since the key value doesn't change, there's no sense in assigning it until we know where it belongs in the heap. Instead, we only make the assignment of

```
$A.heap - size + +$
$A[A.heap - size] = \infty$
Heap-Decrease-Key(A,A.heap-size,key)
```

the parent to the child node. At the end of the while loop, i is equal to the position where *key* belongs since it is either the root, or the parent is at least *key*, so we make the assignment.

Exercise 6.5-7

Have a field in the structure that is just a count of the total number of elements ever added. When adding an element, use the current value of that counter as the key.

Exercise 6.5-8

The algorithm works as follows: Replace the key of the node to be deleted by ∞, ie a value which will be interpreted as greater than all other keys stored in the max-heap. Calling HEAP-INCREASE-KEY will float that node to the top of the max-heap. We then replace the value of the root node with the value of the last element in the heap, known to be smaller than $A[1]$ by the max-heap property. We update the size of the heap, then call MAX-HEAPIFY to restore the max-heap property. This has running time $O(\lg n)$ since INCREASE-KEY runs in $O(\lg n)$ and the number of times MAX-HEAPIFY is recursively called is as most the height of the heap, which is $\lfloor \lg n \rfloor$.

Algorithm 2 HEAP-DELETE(A,i)

```
1: HEAP-INCREASE-KEY(A, i, $\infty$)
2: A[1] = A[A.heap-size]
3: A.heap-size = A.heap-size - 1
4: MAX-HEAPIFY(A,1)
```

Exercise 6.5-9

Construct a min heap from the heads of each of the k lists. Then, to find the next element in the sorted array, extract the minimum element (in $O\lg(k)$ time). Then, add to the heap the next element from the shorter list from which the extracted element originally came (also $O(\lg(k))$ time). Since finding the next element in the sorted list takes only at most $O(\lg(k))$ time, to find the whole list, you need $O(n\lg(k))$ total steps.

Problem 6-1

a. They do not. Consider the array $A = \langle 3, 2, 1, 4, 5\rangle$. If we run Build-Max-Heap, we get $\langle 5, 4, 1, 3, 2\rangle$. However, if we run Build-Max-Heap', we will get $\langle 5, 4, 1, 2, 3\rangle$ instead.

b. Each insert step takes at most $O(\lg(n))$, since we are doing it n times, we get a bound on the runtime of $O(n\lg(n))$.

Problem 6-2

a. It will suffice to show how to access parent and child nodes. In a d-ary array, PARENT(i) $= \lfloor i/d \rfloor$, and CHILD(k, i) $= di - d + 1 + k$, where CHILD(k, i) gives the k^{th} child of the node indexed by i.

b. The height of a d-ary heap of n elements is with 1 of $\log_d n$.

c. The following is an implementation of HEAP-EXTRACT-MAX for a d-ary heap. An implementation of DMAX-HEAPIFY is also given, which is the analog of MAX-HEAPIFY for d-ary heap. HEAP-EXTRACT-MAX consists of constant time operations, followed by a call to DMAX-HEAPIFY. The number of times this recursively calls itself is bounded by the height of the d-ary heap, so the running time is $O(d \log_d n)$. Note that the CHILD function is meant to be the one described in part (a).

Algorithm 3 HEAP-EXTRACT-MAX(A) for a d-ary heap

```
1: if $A.heap - size < 1$ then
2:     error "heap underflow"
3: end if
4: $max = A[1]$
5: $A[1] = A[A.heap - size]$
6: $A.heap - size = A.heap - size - 1$
7: DMAX-HEAPIFY(A,1)
```

Algorithm 4 DMAX-HEAPIFY(A,i)

```
1: $largest = i$
2: for $k = 1$ to $d$ do
3:     if $CHILD(k, i) \leq A.heap - size$ and $A[CHILD(k, i)] > A[i]$ then
4:         if $A[CHILD(k, i)] > largest$ then
5:             $largest = A[CHILD(k, i)]$
6:         end if
7:     end if
8: end for
9: if $largest \neq i$ then
10:     exchange $A[i]$ with $A[largest]$
11:     DMAX-HEAPIFY($A, largest$)
12: end if
```

d. The runtime of this implementation of INSERT is $O(\log_d n)$ since the while loop runs at most as many times as the height of the d-ary array. Note that when we call PARENT, we mean it as defined in part (a).

Algorithm 5 INSERT(A,key)

```
1: $A.heap-size = A.heap-size + 1$
2: $A[A.heap-size] = key$
3: $i = A.heap-size$
4: while $i > 1$ and $A[PARENT(i) < A[i]$ do
5:     exchange $A[i]$ with $A[PARENT(i)]$
6:     $i = PARENT(i)$
7: end while
```

e. This is identical to the implementation of HEAP-INCREASE-KEY for 2-ary heaps, but with the PARENT function interpreted as in part (a). The runtime is $O(\log_d n)$ since the while loop runs at most as many times as the height of the d-ary array.

Algorithm 6 INCREASE-KEY(A,i,key)

```
1: if $key < A[i]$ then
2:     error "new key is smaller than current key "
3: end if
4: $A[i] = key$
5: while $i > 1$ and $A[PARENT(i) < A[i]$ do
6:     exchange $A[i]$ with $A[PARENT(i)]$
7:     $i = PARENT(i)$
8: end while
```

Problem 6-3

a.

2	3	4	5
8	9	12	14
16	∞	∞	∞
∞	∞	∞	∞

b. For every i, j, $Y[1,1] \le Y[i,1] \le Y[i,j]$. So, if $Y[1,1] = \infty$, we know that $Y[i,j] = \infty$ for every i, j. This means that no elements exist. If Y is full, it has no elements labeled ∞, in particular, the element $Y[m,n]$ is not labeled ∞.

c. Extract-Min(Y,i,j), extracts the minimum value from the young tableau Y' obtained by $Y'[i',j'] = Y[i'+i-1, j'+j-1]$. Note that in running this algorithm, several accesses may be made out of bounds for Y, define these to return ∞. No store operations will be made on out of bounds locations. Since the largest value of $i + j$ that this can be called with is $n + m$, and

```
$min = Y[i,j]$
if $Y[i,j+1] = Y[i+1,j] = \infty$ then
    $Y[i,j] = \infty$
    return min
end if
if $Y[i,j+1] < Y[i+1,j]$ then
    $Y[i,j] = Y[i,j+1]$
    $Y[i,j+1] = min$
    return Extract-min(y,i,j+1)
else
    $Y[i,j] = Y[i+1,j]$
    $Y[i+1,j] = min$
    return Extract-min(y,i+1,j)
end if
```

this quantity must increase by one for each call, we have that the runtime is bounded by $n+m$.

d. Insert(Y,key) Since $i+j$ is decreasing at each step, starts as $n+m$ and is

```
$i = m$, $j = n$
$Y[i,j] = key$
while $Y[i-1,j] > Y[i,j]$ or $Y[i,j-1] > Y[i,j]$ do
    if $Y[i-1,j] < Y[i,j-1]$ then
        Swap $Y[i,j]$ and $Y[i,j-1]$
        $j--$
    else
        Swap $Y[i,j]$ and $Y[i-1,j]$
        $i--$
    end if
end while
```

bounded by 2 below, we know that this program has runtime $O(n+m)$.

e. Place the n^2 elements into a Young Tableau by calling the algorithm from part d on each. Then, call the algorithm from part c n^2 to obtain the numbers in increasing order. Both of these operations take time at most $2n \in O(n)$, and are done n^2 times, so, the total runtime is $O(n^3)$

f. Find(Y,key). Let Check(y,key,i,j) mean to return true if $Y[i,j] = key$, otherwise do nothing

```
$i = j = 1$
**while** $Y[i, j] < key$ and $i < m$ **do**
    Check(Y,key,i,j)
    i++
**end while**
**while** $i > 1$ and $j < n$ **do**
    Check(i,j)
    **if** $Y[i, j] < key$ **then**
        j++
    **else**
        i–
    **end if**
**end while**
**return** false
```