Now consider the problem of adding a new item \( x \) to the set (remember: \( x \) is the priority of the new item – in a real application, the priority would be just one attribute of the item but for the time being we are only focusing on the priorities). The first thing to think about is what the heap will look like in terms of its structure. It should be clear that the “shape” of the heap is independent of the values it contains. The heap after adding a new value will have one more vertex, added in the first vacant position in the bottom level of the tree. We start by creating this new vertex (call it \( p \)). Then we find out if the value stored in \( p \)’s parent is < \( x \). If it is, we move the value from the parent down to \( p \), and test again to see if \( x \) can be stored in \( p \)’s parent. In this way we move up the tree until we finally find a safe location to store \( x \).

```plaintext
locate the first vacancy in the bottom level of the tree
create a new vertex and attach it at this point

call this vertex \( p \)

while \( p \neq \text{root} \) and \( p.\text{parent}.\text{value} < x \):
    \( p.\text{value} = p.\text{parent}.\text{value} \)  # move the parent’s value down
    \( p = p.\text{parent} \)

\( p.\text{value} = x \)
```

Each iteration of the loop takes \( O(1) \) time, and the loop executes at most once for each level of the tree. Since the tree is as dense as possible its height is \( O(\log n) \), so adding an element to the heap takes \( O(\log n) \) time ... provided we can find the first vacancy in \( O(1) \) time. This is basically the same stumbling point as we ran into for removing the largest element: we need to quickly locate a specific vertex at the bottom of the tree.

We can imagine that the vertices are numbered, (starting at 1 of course), going left to right across each level.
In this diagram it is quite clear that the bottom-right occupied location is vertex 5, and the location where the next new vertex must go will be as the left child of vertex 3, and it will be numbered 6. The problem, as stated several times already, is to find these positions in the tree quickly.

Now, what data structure do we know that associates an integer with each data location, and gives us constant time access to each of those data locations? The answer of course is a 1-dimensional array.

So we can model the vertices of the max-heap by using the numbering defined above and storing each vertex in the corresponding element of a 1-dimensional array. But what about the edges of the heap – for a given vertex of the heap stored in element $k$ of this 1-dimensional array, how can we …

- locate the parent of this vertex (ie find the array index that contains the parent)
- locate the children of this vertex

Looking at the figure again gives us the answers.

The parent of vertex $k$ has number $\left\lfloor \frac{k}{2} \right\rfloor$ (i.e. $\frac{k}{2}$ rounded down) which in many languages is just calculated as $k/2$ using integer division.

The children of vertex $k$ have numbers $2^1 k$ (the left child) and $2^1 k + 1$ (the right child).

Thus we don’t need to represent the edges of the max-heap – we can figure out where they are using trivial arithmetic.
Suppose we are using an array of size 10 to store the heap shown above. The values would be placed as follows:

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>12</th>
<th>9</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

You should check the calculations stated above to confirm that I have placed the values from the max-heap in the proper elements of the array.

This leaves the first element of the array (annoyingly indexed with 0) unused. But we have a very good use for it: we store the number of elements currently in the heap in this location. Let us use the name A for this array. For the example we are working with, A would look like this.

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>12</th>
<th>9</th>
<th>7</th>
<th>1</th>
<th>3</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Now see how clever this is! The last occupied location of the heap is given by the value of A[0], and the first vacancy in the heap is given by the value of A[0] + 1.

This means that the mysterious “we’ll talk about that later” steps in both in remove-largest and insert algorithms can be solved in O(1) time … which in turn means that both of those algorithms run in O(log n) time … which is a major improvement over storing the max-heap in a linked list or a sorted array (both of which give O(n) for these algorithms).

We can rewrite our remove-largest() and insert(x) methods to utilize this very simple and extremely efficient implementation of the max-heap.
Let A be the array in which we are storing our max-heap (which is the ADT we are using to implement the priority queue ADT). To initialize A with an empty max-heap, we simply set A[0] = 0. Then the function to remove the largest value looks like this:

def remove_max(A):
    if A[0] == 0:
        ERROR("Cannot remove from empty max-heap")
    else:
        max_value = A[1]
        mover = A[A[0]]
        A[0] -= 1
        temp_pos = 1
        left_child = temp_pos * 2
        while left_child <= A[0]:
            max_child = left_child
            if left_child + 1 <= A[0] and A[left_child + 1] > A[left_child]:
                max_child = left_child + 1
            if mover >= A[max_child]:
                break
            else:
                temp_pos = max_child
                left_child = temp_pos * 2
        A[temp_pos] = mover
    return max_val
Some things to observe about this algorithm:

- We delete the now-empty vertex at the bottom of the tree by decrementing A[0]

- We check to see if the current vertex (temp_pos) has any children by computing the index location where its left child would be. We compare this to A[0] to see if it is an active vertex in the tree.

- If the current vertex has no children, we exit the loop. So when we enter the loop, we know there is a left child.

- We check to see if the current vertex has a right child in a similar manner to determining if it has a left child. If it has only one child, we set max_child = left_child. If it has two children we compare their values to determine which is larger.

- If the moving value is >= the larger child of the current vertex, we exit the loop. If not, we move the larger child up and set temp_pos to be the now-empty location of the larger child

- Once we are out of the loop, the current vertex is the proper location for the moving value

- Note that we never explicitly delete any values – we just over-write them when we need to
We should also look at the insert algorithm, adapted to storing the heap in an array:

```python
insert(x):
    if A[0] == len(A) - 1:
        ERROR(“Max-heap is full; cannot insert”)
    else:
        A[0] += 1
        temp_pos = A[0]
        while temp_pos > 1:
            parent = temp_pos // 2
            if A[parent] >= x:
                break
            else:
                temp_pos = parent
        A[temp_pos] = x
```

You should make up some data and work through a few insertions and deletions on a max-heap to make sure you understand these operations.

These two algorithms give us an extremely efficient implementation of the max-heap ADT ... but at the cost of having a fixed maximum size on the heap.

If there is no known upper bound on the size of the set we are storing in the heap, we have two possible solutions. The first is to copy the entire heap to a new, larger array whenever this becomes necessary. This is an O(n) operation but if we make the new array considerably larger than the old one (for example, we can double the size) then we will only have to increase the array size once in a while – the long-term cost of doing this is quite small.

For example, suppose we double the array size whenever we need it to grow, and suppose we start with an array that can store a heap of size 32. When we try to add another value, we have to copy the existing 32 values. Then the heap can grow to 64 before we need to double the array again. This time we have to copy 64 values ... and so on. If we look at the total number of copy operations divided by the total number of values in the heap, we see it is this:
and the next time we have to copy, it will be because we have exceeded 128 values, and the ratio will be

$$\frac{32 + 64 + 128}{128} = 1.75$$

With a little bit of thought, you will discover that this ratio will never reach the value 2. Thus over the life-span of the heap, the average number of copy operations per value in the set is < 2, which means it is O(1). Thus the total number of copy operations is O(n*1) = O(n). This is clearly optimal (you can’t even create a set of n values in less than O(n) time), so it turns out that the “copy when the array gets full” solution is an excellent one.

The second method for having no upper limit on the size of the max-heap is to actually store the heap in a binary tree using vertex objects and pointers. In class we explored a clever way to figure out the path from the root to the last vertex on the bottom level, and also to the point where the next vertex will be added. Time permitting, I will write up a supplemental note about this. It is not central to our discussion of max-heaps because it offers no advantage over the array-based method we have looked at … and in practice will be significantly slower because operations involving pointers are slower than operations involving array elements.

We have seen that a max-heap allows us to implement a priority queue very efficiently. This is not the only use of max-heaps. We will now explore a very clever sorting algorithm based on a max-heap.

**Heapsort**

Suppose an array A contains a max-heap, stored in the manner we have already discussed: A[0] contains the size of the heap, and A[1] contains the root, etc. For example, suppose A[0] = 5. The values in the set occupy A[1] through A[5], and A[1] is the largest value. When the
set is sorted, these values should still occupy positions A[1] through A[5], and the largest value should be in A[5].

We could create the sorted set by creating a new array B. First we could remove the largest value from the heap in A, put that value in B[5], and fix the heap in A. Then we could remove the new largest value from the heap in A, put that value in B[4], and fix the heap in A … and so on. When we finish, the values would be properly ordered in B, and we could just copy them back to A.

But we don’t need the array B! (Boldface is justified because what we are about to see is one of the coolest things in the world.)


This is like magic – the position we need to store each value from the top of the heap becomes vacant just at the moment we need it. It’s like watching the RCMP Musical Ride – the horses weave in and out and the space each one needs to be in is miraculously empty when it gets there.

The algorithm looks like this:

```plaintext
for i = A[0] down to 2:
    temp = remove_max(A)
    A[i] = temp
```

This is one of the slickest, cleverest sorting algorithms around. It runs in O(n*log n) time – which as we know is optimal (approximately n iterations of the loop, each taking O(log n) time) – and it requires only 1 extra memory location (to hold temp). Merge-sort, by comparison, takes O(n) extra space to hold the merged items, and does a lot more data-movements. Quicksort runs faster in practice than Heapsort, which makes it the better choice for important applications … but if I need to write an efficient sort that is hard to get wrong, Heapsort is the one I will write.
Now, there was a big assumption there … Heapsort only works if the data is organized into a max-heap. The likelihood of that happening by luck is somewhere between not-a-chance and no-way. But we can take a randomly ordered set of values and build a max-heap out of them in $O(n \log n)$ time, simply by inserting all $n$ values into an empty max-heap. So the full heap-sort algorithm should be written as:

Heapsort($A$):

- organize $A$ into a max-heap
- for $i = A[0]$ down to 2:
  - temp = remove_max($A$)
  - $A[i] = temp$

Since the first step has the same complexity as the loop, the over-all complexity remains the same.

It turns out we can build the heap in $O(n)$ time! This will not affect the over-all complexity of Heapsort since the loop is still in $O(n \log n)$, but it will reduce the actual running time of the algorithm. Here is how we can do this:

For the purpose of this explanation we will assume that the lowest level of the heap is full. This means than $n = 2^k - 1$ for some $k$, and the bottom level contains exactly $2^{k-1}$ vertices. This simplifying assumption does not affect the validity of the argument, as we will see. The values in the set are randomly placed - our goal is to move them around to form a proper heap. As we already know, the "physical" structure of the heap will not change - it is determined by the number of elements in the set. All we need to do is reorganize the values.

The number of vertices in the bottom level of the heap is $2^{k-1}$. The next level up has $2^{k-2}$ vertices. Make each of these the root of a three-vertex heap, and move the values to make the valid structure in each of the little heaps. This involves, at most, exchanging the root of each little heap with one of its children.

Now take each vertex in the next level up (there are no more than $2^{k-3}$ of them), and make each one the root of a heap with two of the three-vertex heaps as its children. Each of the new heaps will have seven vertices. Enforce the heap-property - this requires pushing the root
value of each new heap down to its proper position - this requires no more than two data exchanges.

Now do the same at the next level, getting $2^{k-4}$ heaps, each of which is built with no more than three data exchanges.

Carry on in this way until you end up with a single heap.

Clearly the bulk of the work in this process is the data exchanges, so we can get the complexity by counting those. The total number of data exchanges is

$$\leq 2^{k-2} \times 1 + 2^{k-3} \times 2 + 2^{k-4} \times 3 + \ldots$$

$$= 2^{k-1} \times \left( \frac{1}{2} \times 1 + \frac{1}{4} \times 2 + \frac{1}{8} \times 3 + \ldots \right)$$

$$= 2^{k-1} \times \left( \left(\frac{1}{2}\right)^1 \times 1 + \left(\frac{1}{2}\right)^2 \times 2 + \left(\frac{1}{2}\right)^3 \times 3 + \ldots \right)$$

$$\leq 2^{k-1} \times \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i \times i$$

Well that looks like it might be a really large number … but it turns out that we can show that for any x such that -1 < x < 1,

$$\sum_{i=1}^{\infty} x^i \times i = \frac{x}{(1-x)^2}$$

(One way to check this is to use your hard-won skills from Grade 12 Math to do the division on the right hand side – the result is exactly the sum on the left hand side. Another way to check it is to multiply the sum on the left side by $(1 - x)^2$ - the result is exactly x.)

In our expression for the number of data exchanges, we see that we can apply this identity with $x = \frac{1}{2}$ . We continue like this:
Thus we see that the total number of data exchanges is in $O(n)$ ... which means that building the heap is in $O(n)$, as claimed.

The fact that we can build a max-heap in $O(n)$ has another application. Suppose we have a set of $n$ values and we want to find the $k$ largest values in the set, for some value of $k$. Using a heap, we can find them in $O(n + k \cdot \log n)$ time ... the $n$ comes from creating the heap, and the $k \cdot \log n$ comes from extracting the $k$ largest values in descending order. Compare this to other possible algorithms: Sorting the whole set and then choosing the $k$ largest items takes $O(n \cdot \log n)$ time. Looking through the set to find the largest, then searching again to find the second largest, then again for the third largest, etc, takes $O(k \cdot n)$ time. The heap-sort approach clearly wins (note that for small values of $n$ and $k$ the third method might be faster because it doesn’t do any data movements, just comparisons).

Thus we see that the total number of data exchanges is in $O(n)$ ... which means that building the heap is in $O(n)$, as claimed.

The fact that we can build a max-heap in $O(n)$ has another application. Suppose we have a set of $n$ values and we want to find the $k$ largest values in the set, for some value of $k$. Using a heap, we can find them in $O(n + k \cdot \log n)$ time ... the $n$ comes from creating the heap, and the $k \cdot \log n$ comes from extracting the $k$ largest values in descending order. Compare this to other possible algorithms: Sorting the whole set and then choosing the $k$ largest items takes $O(n \cdot \log n)$ time. Looking through the set to find the largest, then searching again to find the second largest, then again for the third largest, etc, takes $O(k \cdot n)$ time. The heap-sort approach clearly wins (note that for small values of $n$ and $k$ the third method might be faster because it doesn’t do any data movements, just comparisons).

The heap approach for extracting the $k$ largest values in a set is particularly good if we don’t actually know what $k$ is in advance. For example, we might need to add up the largest values in the set until the total meets or exceeds some target value. There’s no point sorting the whole set if we only need a few large values, and we certainly don’t want to repetitively search for the largest remaining value over and over if it turns out we need most of the values in the set.
If we do know \( k \) in advance, there is yet another approach. Using a very clever algorithm described in Section 9.3 of our text, we can actually extract the \( k \) largest items in the set in \( O(n) \) time, but in an unsorted order. Then we can sort these in \( O(k \cdot \log k) \) time to get the items in the order we want. This algorithm is well outside the scope of CISC-235 but I recommend it to your attention.

### Changing Priorities

Sometimes it becomes necessary to change the priority of an item in a priority queue – or even change the priority of many of the items at the same time. A practical example of this would be a priority queue for a printer – a low priority item might never get printed if there is always at least one item with higher priority ahead of it in the queue. One solution to this particular problem is to look at the queue on a regular basis (such as once every hour) and raise the priority of items using a formula based on how long the item has been in the queue. In this way low-priority items eventually become high-priority items.

The difficulty does not lie in changing the priority and fixing the heap – that is just a matter of moving the item up or down in the heap. The hard part is finding the item whose priority we want to change. We don’t want to search the whole heap, since we are doing everything we can to avoid \( O(n) \) operations. We can’t search the heap in less than \( O(n) \) time because heaps are not organized as search trees: from a given vertex we have no way to know if we should recurse on the left side or the right side, so we have try both sides.

If you weren’t in class when we discussed this, take a moment to think about how you would solve this problem: we want to be able to quickly find any item in the tree. Let’s see, what data structure do we know that (when properly implemented) offers us expected search time in \( O(1) \)? The word *expected* gives it away – I am talking about a hash table in which the keys are unique identifiers for the items, and the satellite data in the hash table are pointers to the locations in the max-heap where the items are located.

Let’s look at an example. Suppose we have a heap of cats, containing “Simon” (priority 10), “Traill” (3), “Mocha” (7) and “Lindor” (2). The heap might look like this:
We also have a hash table, which might look like this (the actual placement of the cats in the hash table obviously depends on the hash function.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>“Traill”</td>
<td>4</td>
</tr>
<tr>
<td>“Mocha”</td>
<td>2</td>
</tr>
<tr>
<td>“Lindor”</td>
<td>3</td>
</tr>
<tr>
<td>“Simon”</td>
<td>1</td>
</tr>
</tbody>
</table>

If we want to change Mocha’s priority to 12, we look Mocha up in the hash table, which tells us Mocha is located in vertex 2 of the heap. If the heap is stored in an array, this is all we need. If the heap is actually stored in a binary tree, the hash table could give us a direct link to the vertex object containing Mocha. Once we have access to the Mocha item we can change its priority and push it up or down in the heap. The complexity of this is good: if the hash table is well-designed then its operations should take O(1) time, and then pushing the item up
or down is in $O(\log n)$ since there that many levels in the heap. Thus we can change the priority of an item in a priority queue in $O(\log n)$ time.

But wait – if we use this method to update the priority of every item in the heap, that will take $O(n \cdot \log n)$ time. We would better off rebuilding the heap from scratch since we know that takes $O(n)$ time. This is another situation where we need to know about our particular application before we can choose a solution. If priority changes are rare or we only change a few at a time, then pushing the changed items up or down in the heap is the best choice. But if we change a lot of priorities at the same time we should just rebuild the heap using the new priorities.