# Priority Queues and Heaps

A **queue** is an abstract data type that supports addition and removal of items, with the following restrictions:

- new elements can only be added to the end of the queue
- only the item at the beginning of the queue can be accessed and/or removed

A queue can be implemented with an array:

- addition and removal take O(1) time
- size is limited
- need to keep track of where the beginning and end of the queue are in the array
- the queue may eventually "wrap around" the end of the array - not a problem but just a special case that we need to deal with

An alternative implementation is with a linked list:

- size is not limited
- addition and removal take O(1) time
- no special cases to worry about
- linked lists take more memory space than arrays, and are a bit slower

A **priority queue** is a queue in which each item has a priority attached. We will assume that 1 is the **lowest** priority. Priority queues have the following operations:

- add new elements
- find (and/or remove) the item with the highest priority
- in practice, we want items with equal priority be sequenced in such a way that items added earlier reach the head of the queue before items added later (for example, if item A and item B both have the same priority and A is added to the priority queue before B is added, then item A should come to the head of the priority queue before B does)

Note that searching the set is not a required operation. Some implementations of priority queues do support search operations – we will discuss this later.

Priority queues have many applications including hospital emergency wards, airport plane landing sequencing, and operating system task scheduling.

In every application, the priority of an item is just one of its attributes and a full implementation will have to store all of the data for each item. However in our discussion we will focus only on the priority values of the items. We will assume that the rest of the data “goes along for the ride”. In our diagrams we will only show the priority values of the items unless we need to distinguish between items that happen to have the same priority.

**Implementation of a Priority Queue:**

We can certainly implement a priority queue with an array or linked list:

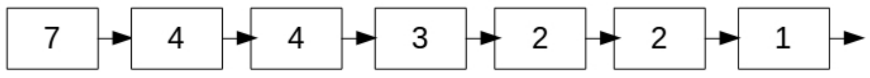
- a new item with priority p is inserted **after** all items with priority $\geq$ p, and **before** all items with priority < p
- adding an item is in O(n) where n is the number of items in the queue
- accessing (and/or removing) the item with maximum priority is in O(1)

Example of the problem when using an array:

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 4 | 4 | 3 | 3 | 3 | 3 | 2 | 1 | 1 | 1 | | | | |

To insert a new item with priority 7, almost all the values need to be moved one space to the right to create an empty space for the new item – this is O(n)

Example of the problem when using a linked list:



To insert a new item with priority 2, we need to walk through most of the list to find the insertion point – this is O(n)

Note that if the set of possible priorities is limited to the set {1,2, ... k} for some fixed integer k, then we can implement the priority queue with a set of  separate queues, one for each priority class.  Adding a new item takes O(1) time, and accessing (and/or removing) the item with maximum priority takes O(k) time, and since k is a fixed integer, this is O(1).

That's a nice result (O(1) complexity is always good) but not very interesting ... so we focus on the situation where the set of possible priorities is unlimited.

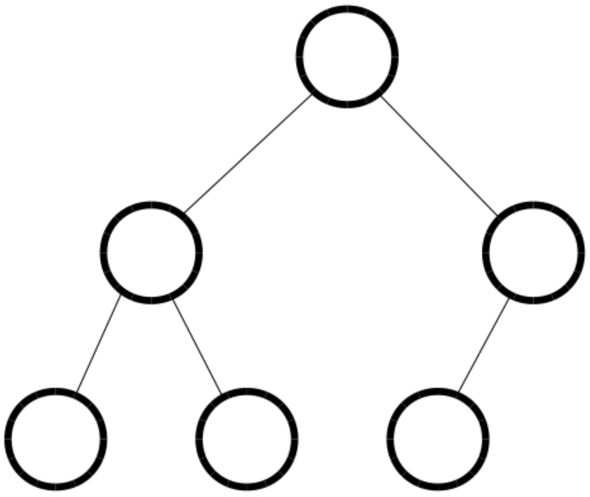
If the set of possible priorities is not limited, we need to be smart to improve on the O(n) time to add new items.  We will use a structure called a **max-heap**.

**Max-Heaps**

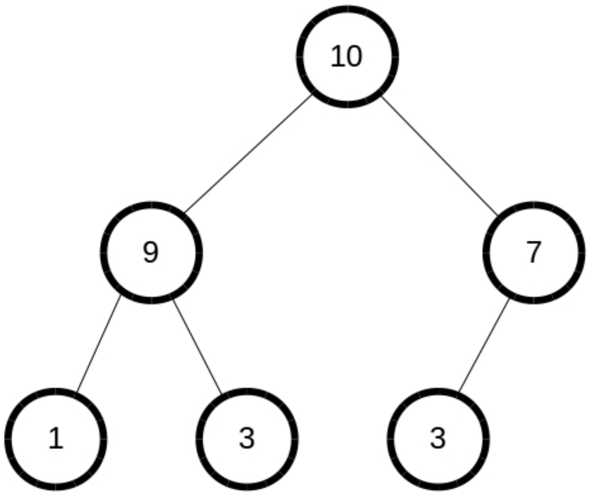
A max-heap is a binary tree such that

- the value stored at each vertex is $\geq$ the values stored at its children. Note that this results in the largest value being at the root of the tree
- all levels are full, except possibly the last one
- if the bottom level is not full, all of its vacancies are at the right side

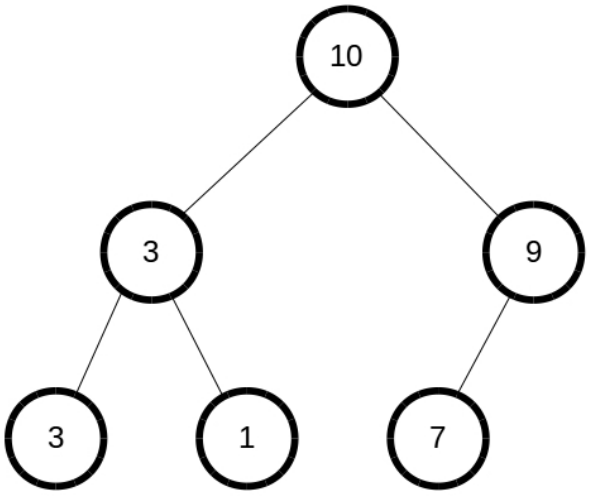
So we have one **organizational** rule, and two **structural** rules. Because of the structural rules, the "shape" of a max-heap containing a set of k values is completely fixed. For example if the set contains 6 values, a max-heap containing these values must look like this.



However, it is important to see that the organizational rule can be satisfied by different arrangements of the values. For example if the set of values is {1,3,3,7,9,10}, the heap could be arranged as



or as



or as any of several other arrangements. Note that there is no "left child $\leq$ right child" rule.

However there are two consistent features of all legal max-heap arrangements:

- the largest value is at the root of the tree

- the second largest value (which may be a duplicate of the largest value) is in one of the root's children.

Accessing the maximum value is O(1), since it is at the root of the tree.  But what is the complexity of removing that value?

**Removing the largest value from a max-heap**

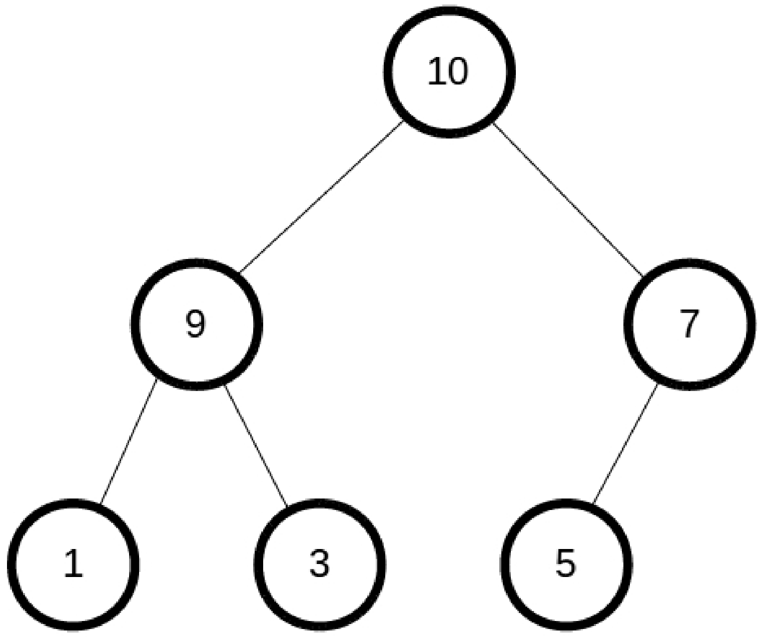
The problem of course is that we can't simply delete the first vertex (as we could if the priority queue were stored in a linked list) – we need to choose a new root.  One idea that occurs to many people is to "promote" the larger of the root's children by moving that value up into the root vertex ... then filling the newly empty vertex by promoting the larger of its children, and so on down the tree until we end up with a leaf with nothing in it.  Then delete that leaf.

The problem with this is that it spoils the structural rules of the max-heap: unless the empty leaf we delete happens to be the right-most one on the bottom level, we will end up with a gap in some level.  Well, so what?  Rules are made to be broken, right?  But the danger here is that by allowing this and performing subsequent deletions in the same way, our tree may end up looking very sparse ... in fact it may end up looking like a linked list.  And in that (extreme) case, subsequent "remove largest" operations will be in O(n) because we would go all the way down the tree, promoting the value at each level.  This is bad because our goal is to avoid O(n) operations.

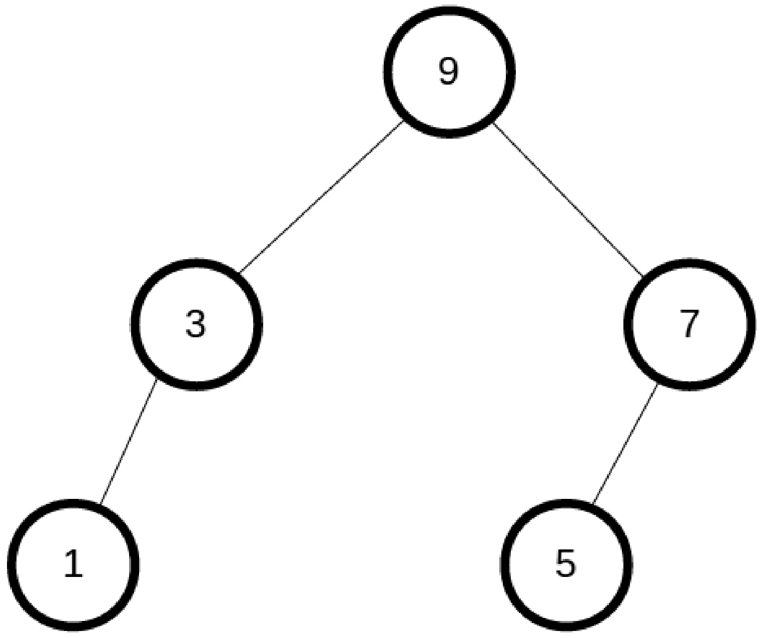
When we remove the largest value in the set, the structure of the resulting heap is predetermined: it must look exactly like the original heap, but without the right-most leaf on the bottom level.

Here is an approach that people often propose:  Since we know that the process just described will end up deleting a leaf, and we also know that the heap needs to "lose" its bottom-right leaf to be properly structured, why not fill the gap where the leaf has been deleted (because its value has been promoted) with the value in the leaf that "needs" to be deleted?

Unfortunately, this may not result in a valid heap.   Consider this example:



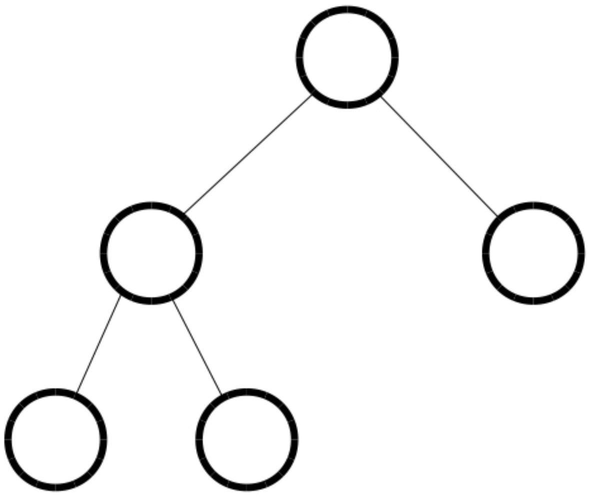
If we remove the 10 and attempt to replace it with its larger child (9), and then replace that with its larger child (3) we end up with this



and if we now move the leaf that needs be gone (the one containing 5) over to where we just lost a leaf (due to promoting 3) we end up with 5 having 3 as its parent ... which violates the heap rule.   We would then have to promote the 5 upwards to a valid location.

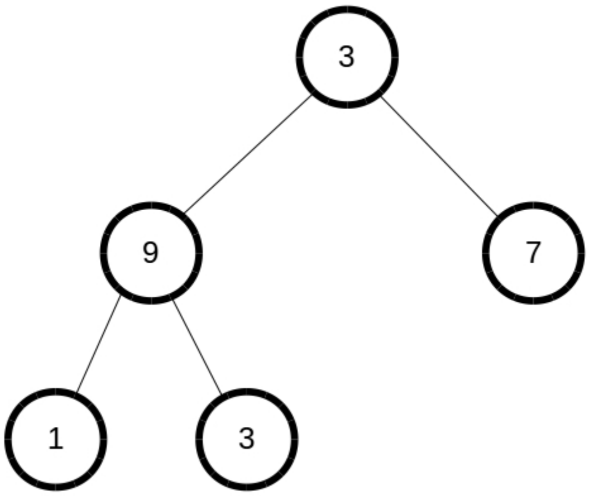
But ... it was a clever idea!

So here's a better approach: Consider the **first** max-heap example shown above. We know that after we remove the largest value, the set will have exactly 5 elements, and therefore we can see exactly what the max-heap will look like after the deletion:



So when we take out the largest value (10) we have an empty vertex at the root. But we also have to get rid of the last vertex on the bottom level, which contains 3. Hmm, a vertex that needs a new value, and a value that needs a new vertex ... what shall we do?

The answer is that we put the 3 in the empty vertex at the root, giving this



But now the organizational rule is violated. We need to fix that without changing the structure of the max-heap. Fortunately this is really easy: we compare the value we just moved up and the values of its children. If it is smaller than either of them, we "swap" it with the larger of its children (in this example, we swap the 3 in the root with the 9). Now we compare the moving 3 with the values of its new children, and if necessary we swap it with the larger of its children ... and we continue pushing it down the tree until it reaches a valid location. (In this example the 3 does not need to move down any more because it is $\geq 1$ and $\geq 3$ .) It is important to understand that this process is guaranteed to produce a properly arranged max-heap: whenever we swap a value "up", it lands in a valid location because it is $\geq$ all the values below it.

So what is the complexity of this process? Moving the value from the bottom to the top takes constant time, then each time we swap it with one of its children, that is also a constant time operation ... and because we have kept the tree as compact as possible, we know there are $\leq \log n$ levels in the tree! So this entire process is in $O(\log n)$ Win!

Note: we can improve the practical efficiency of this by not actually re-inserting the value from the bottom into any vertex until we actually find its new home. This means that all the points where we used the term "swap" are really just "promote" operations – it saves a bit of time.

Here is pseudo-code for the entire “remove largest” operation:

```
def remove_max():
   max_value = root.value
   mover = value from the right-most filled position in the bottom
               level of the tree
   delete the vertex that contained mover
   temp_pos = root
   while mover < either of the children of temp_pos:
       new_pos = temp_pos.left or temp_pos.right, whichever has the
                     larger value
       temp_pos.value = new_pos.value
       temp_pos = new_pos
   temp_pos.value = mover
   return max_value
```

For implementation, we have to handle a special case: temp_pos may have only a left child (note that there is no way a vertex can have a right child but no left child) so the logic is something like this:

```
     if temp_pos.left != None:
          max_child = temp_pos.left
          if temp_pos.right != None and
                    temp_pos.right.value > temp_pos.left.value:
               max_child = temp_pos.right
          # now max_child points to the child with the largest value
          if mover < max_child.value:
               ...
```

There is also a significant step in the algorithm left unspecified ... how do we find the right-most occupied vertex on the bottom level of the tree? **We will come back to that.**

## Inserting a Value Into a Max-Heap

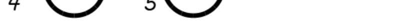
Now consider the problem of adding a new item x to the set (remember: x is the priority of the new item – in a real application, the priority would be just one attribute of the item but for the time being we are only focusing on the priorities).

The first thing to think about is what the heap will look like in terms of its structure. 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 (remember, x is the value we are adding to the set). 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.

```
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 != root  and   p.parent.value < x :
     p.value = p.parent.value # move the parent's value down
     p = p.parent
p.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** easily. 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.



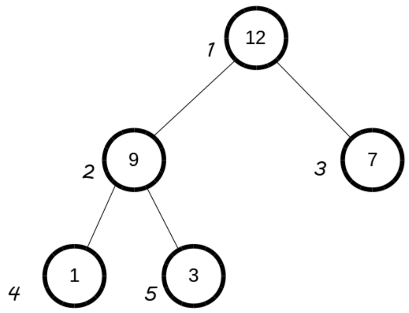
In this diagram it is quite clear that the bottom-right occupied location is vertex 5, and the location where the next new vertex must go will be as the left child of vertex 3, and it will be numbered 6. The problem, as stated several times already, is to find these positions in the tree quickly.

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*k (the left child) and 2*k + 1 (the right child).

Thus we don't need pointers to implement the edges of the max-heap – we can figure out where they go 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:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 12 | 9 | 7 | 1 | 3 | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

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. We might as well call this array A. For the example we are working with, A would look like this.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 12 | 9 | 7 | 1 | 3 | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

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 give 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]]     # the value from the last leaf on the bottom level
            A[0] -= 1           # size of the heap goes down by 1
            temp_pos = 1
            left_child = temp_pos * 2
            right_child = left_child+1
            # now push the value down until it reaches a proper location
            while left_child <= A[0]:
                  max_child = left_child
                  if right_child <= A[0] && A[right_child] > A[left_child]:
                        max_child = right_child
                  if mover >= A[max_child]:
                        break       # we have found the proper location
                  else:
                        # promote the larger child up, and move down one
                        # level of the tree
                        A[temp_pos] = A[max_child]
                        temp_pos = max_child
                        left_child = temp_pos*2
            A[temp_pos] = mover     # store the moving value in its proper location
            return max_val          # return the value that was at the top of the
                                    # heap
```

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 $\geq$ 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:

```
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:
                A[temp_pos] = A[parent]
                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:

$$\frac{32+64}{64} = 1.5$$

and the next time we have to copy it will be because we have exceeded 128 values, and the ratio will be

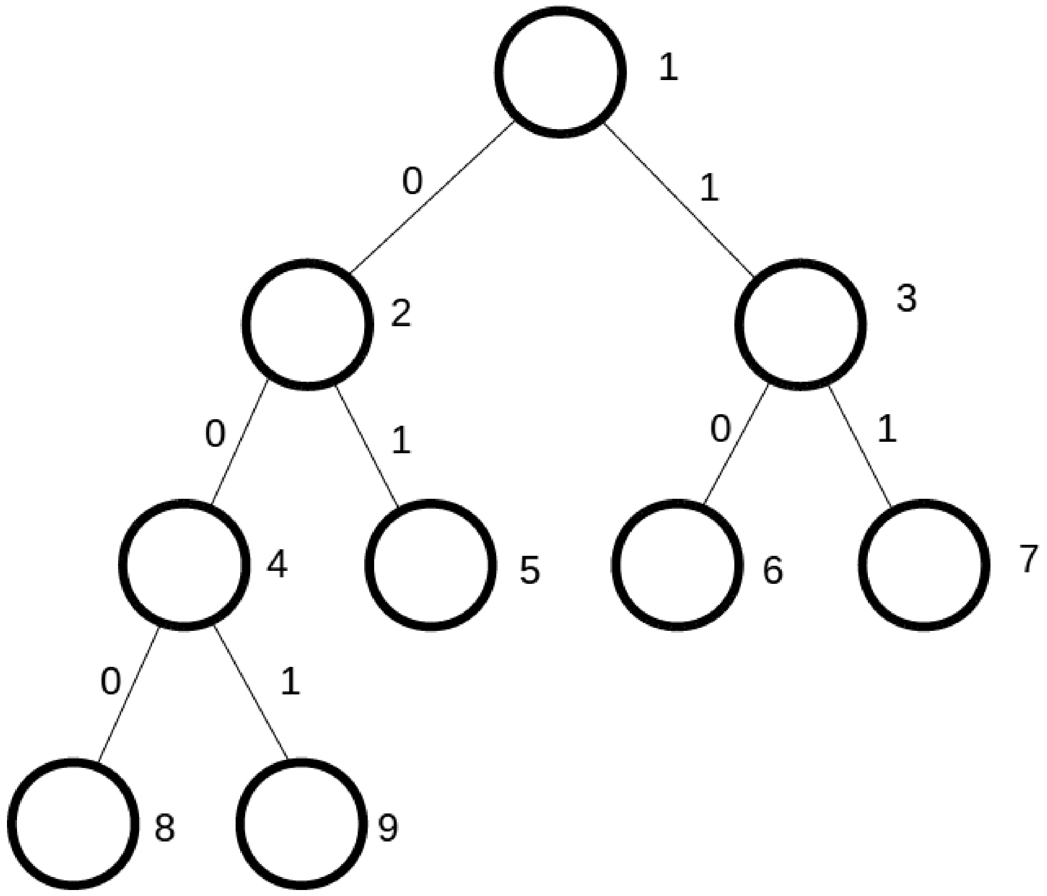
$$\frac{32+64+128}{128} = 1.75$$

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 in O(1). Thus the total number of copy operations is in 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. We can use the number of values in the heap to determine exactly where where last leaf is, and this is how:

We can label the edges of the tree with "0" on the "left-child" edges and "1" on the "right-child" edges.

Consider this heap, with the edges labelled as just described:



In this figure I have put the "number" of each vertex beside the vertex ... let's look a the relationship between each vertex number, in binary, and the sequence of labels on the edges in the path that leads from the root to the vertex.

| | | |
|---|---|---|
| 1 | 1 | - |
| 2 | 10 | 0 |
| 3 | 11 | 1 |
| 4 | 100 | 0-0 |
| 5 | 101 | 0-1 |
| 6 | 110 | 1-0 |
| 7 | 111 | 1-1 |
| 8 | 1000 | 0-0-0 |
| 9 | 1001 | 0-0-1 |

It's pretty clear that we can get the sequence of edges that lead to vertex k simply by ignoring the first bit of the binary form of k.  (This is easy to prove by induction – I recommend doing this just for the exercise.)

So if we know there are n values in the heap, we can find the right-most leaf on the bottom level by using the binary form of n ... and we can find the first empty spot on the bottom level by using the binary form of n+1.

I remember being quite amazed the first time I saw this.

However it offers no real 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.)

If we pull A[1] out of the heap in A and fix the heap, **the remaining values only occupy the positions A[1] through A[4] !!!** The value that we pulled from A[1] is supposed to end up in A[5], and A[5] is no longer part of the heap … so we can just put the value from A[1] in that location and carry on: we pull out the new value from A[1] and fix the heap … which leaves A[4] unused – so we put the value from A[1] there.

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:

```
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

$$\begin{aligned}
&\leq 2^{k-2} * 1 + 2^{k-3} * 2 + 2^{k-4} * 3 + ... \\
&= 2^{k-1} * (\frac{1}{2} * 1 + \frac{1}{4} * 2 + \frac{1}{8} * 3 + ...) \\
&= 2^{k-1} * (\frac{1}{2^1} * 1 + \frac{1}{2^2} * 2 + \frac{1}{2^3} * 3 + ...) \\
&= 2^{k-1} * \left(\left(\frac{1}{2}\right)^1 * 1 + \left(\frac{1}{2}\right)^2 * 2 + \left(\frac{1}{2}\right)^3 * 3 + ...\right) \\
&\leq 2^{k-1} * \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^i * i
\end{aligned}$$

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

(One way to check this is to use your 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 both sides by $(1-x)^2$ The result on both sides 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:

$$\begin{aligned}
\text{number of data exchanges} &\leq 2^{k-1} * \sum_{i=1}^{\infty} \left(\frac{1}{2}\right)^{i} * i \\
&= 2^{k-1} * \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} \\
&= 2^{k-1} * \frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2} \\
&= 2^{k-1} * \frac{\frac{1}{2}}{\frac{1}{4}} \\
&= 2^{k-1} * 2 \\
&= 2^{k} \\
&= n+1
\end{aligned}$$

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*log n) time ... the n comes from creating the heap, and the k*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* 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*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*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.

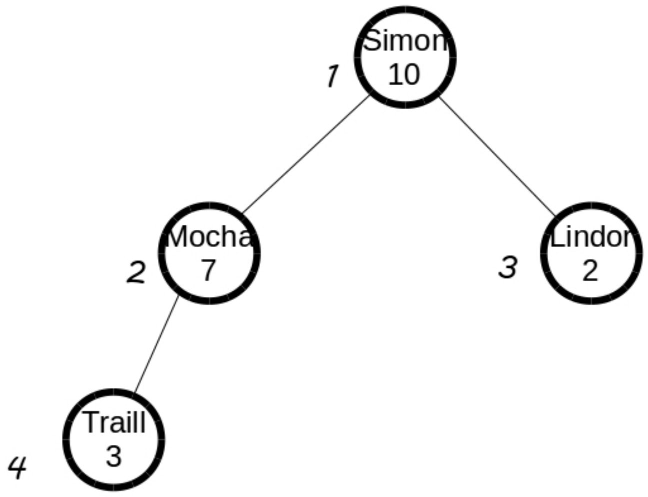
## Changing Priorities

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 haven't seen the solution to 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.

| | |
|---|---|
| | |
| “Traill” | 4 |
| “Mocha” | 2 |
| | |
| | |
| “Lindor” | 3 |
| | |
| | |
| | |
| “Simon” | 1 |

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 * 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.