CS 624
Lecture 3: Heapsort

1 Pre-heaps

Definition *The* height *of a node in a tree is the number of edges on the longest path from that node down to a leaf.*

For instance, in the tree on the left of Figure 1,

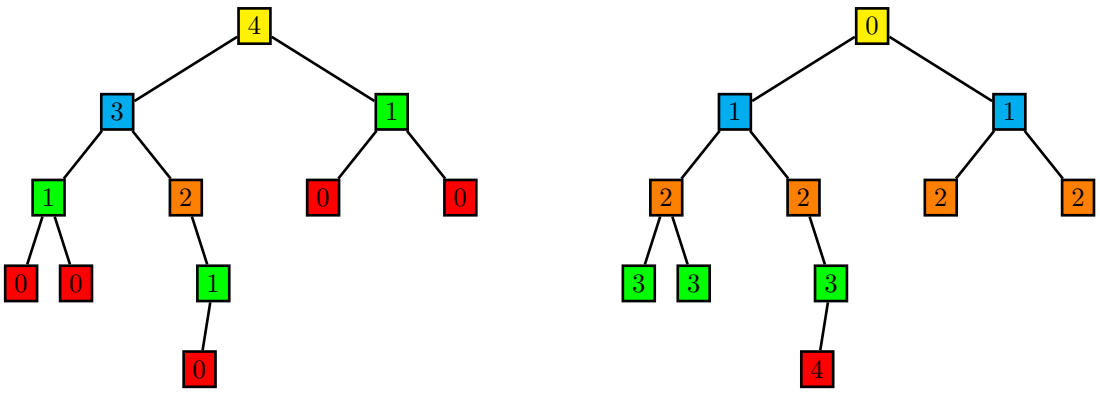


Figure 1: Heights (on the left) and levels (on the right).

each node has its height indicated by a number. The red nodes are at height 0, the green nodes are at height 1, and so on. In this picture, the tree was a binary tree, but certainly this definition of height applies to nodes in any tree.

Definition *The* height *of a tree is the height of the root.*

We will generally use the variable H to refer to the height of a tree.

We say that each node in a tree is on some level. The root is at level 0. The children of the root are at level 1. And in general, the children of a node at level k are at level $k+1$. So we could also say that the height H of a tree is the greatest level of any node in the tree.

In a binary tree, there are at most 2^k nodes at level k. Further, if the tree has level H *and if all the levels are completely filled in*, then level H contains 2^H nodes, and the tree contains

$$1 + 2 + 4 + \cdots + 2^H = 2^{H+1} - 1 \tag{1}$$

nodes.

1.1 Exercise *Prove the identity (1).*

We are going to study a particular kind of data structure called a *heap*. In computer science generally, the term *heap* normally refers to memory that is allocated explicitly by the programmer (for instance, by a "new" statement in C or C++). In the world of algorithms, however, a heap is a particular kind of data structure. That is the meaning we will use in the course.

There are two parts to the definition of a heap:

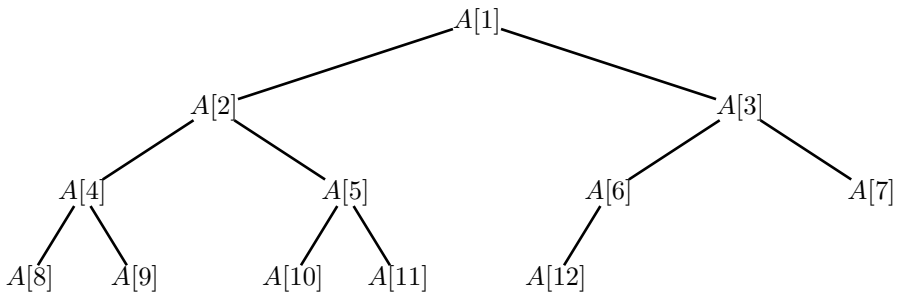
- It is a special kind of binary tree—a binary tree with a specific kind of shape.
- Each node contains some data satisfying certain constraints.

For clarity, I'm going to abstract out just the first part—we will call such a data structure a *pre-heap*[1]

Definition *A* pre-heap *is a binary tree such that*

- *All leaves are on at most two adjacent levels.*
- *With the possible exception of the lowest level, all the levels are completely filled. The leaves on the lowest level are filled in, without gaps, from the left.*

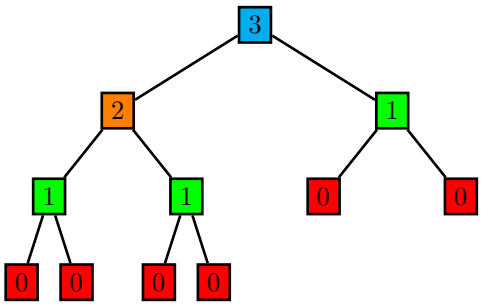
Because of the shape of a pre-heap, we can represent the elements simply in a 1-dimensional array:



[1]Please note that this term is not at all standard. I made it up just for these notes. I doubt you would find it anywhere else.

1.2 Exercise *The children of the node holding $A[n]$ (if those children exist) are the nodes holding $A[2n]$ and $A[2n+1]$, and the parent of the the node holding $A[n]$ is $A\big[\lfloor n/2 \rfloor\big]$.*

Here is another example of a pre-heap. Each node is tagged with its height. The height of the tree itself is 3. All the levels less than 3 are completely filled in, and there are a total of $2^3 - 1$ nodes at those levels.



If we have a pre-heap with n nodes, let us denote its height by H. By what we have already seen above, the tree must look like what is shown in Figure 2. Based on that figure, we can see that

$$\boxed{2^H \leq n \leq 2^{H+1} - 1 < 2^{H+1}}$$

Equivalently,

$$\boxed{H = \lfloor \log_2 n \rfloor}$$

1.3 Lemma *In a pre-heap with height H, there are at most 2^H leaves.*

PROOF. Let us call the pre-heap P.

The leaves of P consist of all the nodes at level H, together with those nodes at level $H-1$ that have no children.

Actually though, we want to consider three kinds of nodes in P:

1. Those nodes at level H whose parents have two children. (These nodes are all leaves of P, and they of course occur in pairs, right?)
2. The node—if such a node exists—at level H whose parent has only one child. There can be at most one such node. If it exists, let us call it n_1, and let us call its parent p_1. n_1 is also a leaf of P, but it is not part of a pair of leaves.
3. The nodes at level $H-1$ having no children. And of course there might be no such nodes at all. But any such nodes are leaves of P.

Now suppose we perform the following operations on the pre-heap P, for the nodes in those three categories:

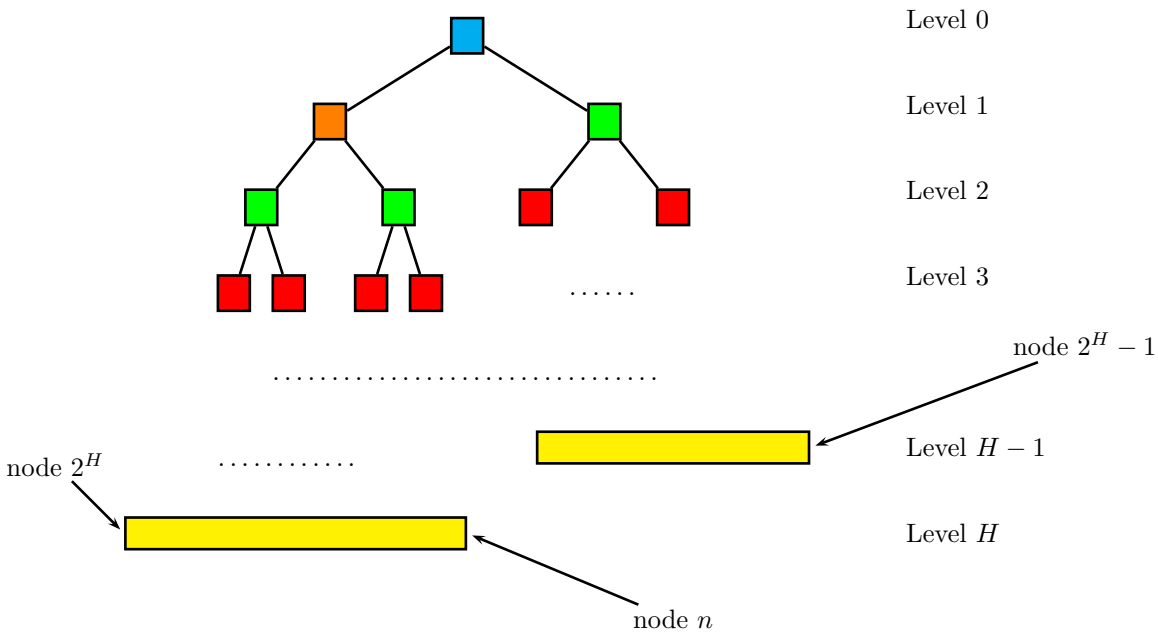


Figure 2: A pre-heap with levels and heights indicated. The leaves are the elements in the yellow regions.

1. For the nodes in the first category, don't do anything.

2. If the node p_1 exists, add a second child to p_1. This will be a new leaf of P. So in this case, P will now have one more leaf than it did originally.

3. If any nodes at level $H-1$ having no children exist, add two children to each of them. This will replace each such node (which is a leaf in the original pre-heap) by two new leaves. So each time we do this we will be adding 1 to the number of leaves of P.

When we have finished, we will have a new pre-heap—call it P'—with level H completely filled out. It will have 2^H nodes (since level H contains 2^H nodes). And as we have seen, the pre-heap P' has at least as many nodes as the original preheap P does. So the number of nodes in the original pre-heap P must be at most 2^H. And that completes the proof. □

This next result is intuitive, but also essential for what we do below.

1.4 Theorem *In a pre-heap with n elements, there are at most $\frac{n}{2^h}$ nodes at height h.*

Proof. We have just seen that there are at most 2^H leaves in such a tree, and the leaves are just the nodes at height 0.

Now if we take away the leaves, we have a smaller pre-heap with at most 2^{H-1} leaves, and these leaves are exactly the nodes at height 1 in the original tree. Continuing, we see that there are at most 2^{H-h} nodes at height h in the original tree. And

$$2^{H-h} = \frac{2^H}{2^h} \leq \frac{n}{2^h}$$

so we are done. □

2 Heaps

Definition *A* heap *is a binary tree with a key in each node. The keys must be comparable—that is, we must be able to say which of two keys is greater or lesser. (So for our purposes here, we can think of the keys as being numbers.) To be a heap, such a binary tree must have the following properties:*

- *The tree must be a pre-heap. That is,*
  - *All leaves are on at most two adjacent levels.*
  - *With the possible exception of the lowest level, all the levels are completely filled. The leaves on the lowest level are filled in, without gaps, from the left.*

  and in addition,
- *The key at each node is greater than or equal to the key in any descendant of that node.*

The last of these conditions could also be phrased recursively, like this:

- *The key in the root is greater than or equal to that of its children, and its left and right subtrees are again heaps.*

2.1 Exercise *Prove the last statement. That is, prove that we could replace the statement*

- *The key at each node is greater than or equal to the key in any descendant of that node.*

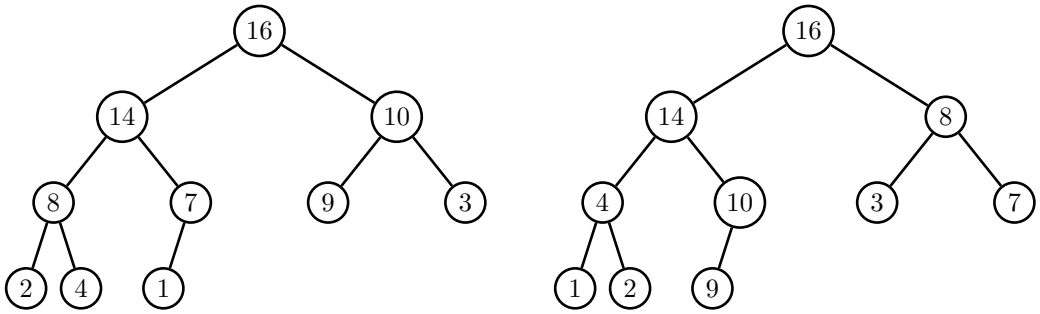
in the definition above by the statement

- *The key in the root is greater than or equal to that of its children, and its left and right subtrees are again heaps.*

This second statement is really a recursive statement. Now one thing you ought to know (if you don't already) is this: recursion is essentially equivalent to mathematical induction. Any proof that a recursive statement is true will always[2] *be a proof by mathematical induction. So that's the way you need to structure your proof.*

[2]Well, I can't think of any case where this is not true.

Even though the shape of a heap containing n elements is uniquely determined (since it is a pre-heap), the arrangement of those n elements is not. For example, here are two heaps, both containing the same set of keys:



Because of the regular structure of a pre-heap, it is most efficient to store a heap as an array. The key held by element i is $A[i]$.

Note the special trivial case: if the tree contains only one node, then automatically it must be a heap. This is used as the base of the recursive algorithm for creating heaps; it is also used as the base of the inductive argument showing that the algorithm is correct. Of course those two uses are at bottom the same.

3 The HEAPIFY procedure

Heaps are remarkable data structures in that they seem to carry "just the right amount" of information. Not too much—they don't have the set of elements completely sorted, and not too little—they do have some extremely important order relations encoded in them.

For this reason, it is extraordinarily efficient to create and modify heaps. We do this using a helper routine named HEAPIFY. And as we will see, HEAPIFY is very cheap indeed.

The idea is this: we have a binary tree in the shape of a heap (but perhaps not actually a heap). For convenience, we represent the tree as an array $A[1..n]$, where $n = A.heapsize$. Further, we specify a distinguished node i (holding the value $A[i]$). (It would probably be better to be more careful and refer to this node as "the node having index i".)

We will follow our text in using the following notation:

- LEFT(i) is the index of the left child of the node having index i, if that node exist. So if the node having index i has a left child, then LEFT(i) $= 2i$.
- RIGHT(i) is the index of the right child of the node having index i, if that node exists. So if the node having index i has a right child, then RIGHT(i) $= 2i + 1$.

HEAPIFY takes as input the array A and one of the nodes, say i. So it is called like this: HEAPIFY(A, i). It is assumed (and it *must* be true) that the children of node i are the roots of heaps. That is, if

the node having index i has two children, then it must be the case that

- The tree rooted at $l = \textsc{Left}(i)$ is a heap.
- The tree rooted at $r = \textsc{Right}(i)$ is a heap.

We do not, however, assume that $A[i] \geq \max\{A[l], A[r]\}$. Equivalently, we do not assume that the tree rooted at i is a heap. And it also might be that

- node i has only one child (in which case that one child must be the root of a heap), or
- node i has no children.

If node i has no children there there is no assumption that has to be made; in such a case, Heapify will return without doing anything.

Heapify, when applied to this situation, will reorganize the nodes at and under $A[i]$ so that they form a heap rooted at $A[i]$. (That is, it will move values from one node to another, so that when it is done, the value of $A[i]$ may be different from what it was originally. If so, however, the original value will be at some descendant of $A[i]$.) It's important to note that this procedure does not change the shape of the tree—it only changes *where* values are located in the tree.

The procedure works by letting the value $A[i]$ "float down" to its proper position in the heap.

In the following pseudocode the condition "**if** $l \leq A.heapsize$" is just a test to make sure that there actually is a node at position l.

```
Heapify(A, i)
    l ← Left(i)
    r ← Right(i)

    // First figure out which of the top three nodes (i.e., the node at
    // i and its children, whether there are 0, 1, or 2 of them) holds
    // the largest value.
    if l ≤ A.heapsize and A[l] > A[i] then
        largest ← l
    else
        largest ← i
    if r ≤ A.heapsize and A[r] > A[largest] then
        largest ← r

    // Then switch that node with the node at position i.
    if largest ≠ i then
            exchange A[i] ↔ A[largest]
            Heapify(A, largest)
```

Figure 3 shows an example (from the text) of how Heapify works.

You should convince yourself (by looking at the pseudocode) that Heapify correctly handles the cases in which node i has only 1 child, or no children at all.

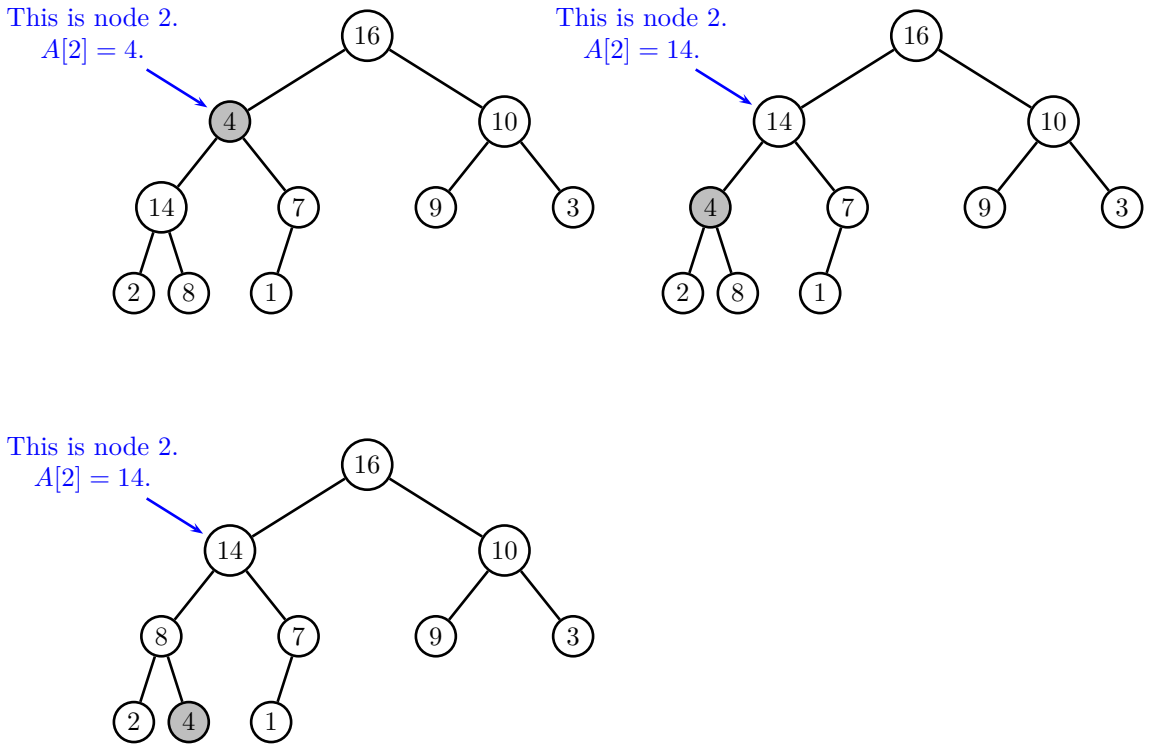


Figure 3: How HEAPIFY works. In the original picture, the children of the node 2 (with value $A[2] = 4$) are both roots of heaps. Node 2 however, is not. HEAPIFY$(A, 2)$ allows the original value in node 2 (that is, 4) to "float down" until it finds its proper place.

3.1 Exercise *Prove that the call to* HEAPIFY(A, i)*, when used as directed, does actually transform the subtree rooted at i into a heap.*

This proof will of necessity be inductive. The point of this exercise is to write your proof in a way that makes that clear.

What's the running time of HEAPIFY? The time needed to run HEAPIFY on a subtree of size n rooted at a given node i is

- time $\Theta(1)$ to fix up the relationships among the elements $A[i]$, $A\big[\text{LEFT}(i)\big]$, and $A\big[\text{RIGHT}(i)\big]$, plus
- time to run HEAPIFY on a subtree rooted at one of the children of node i. If the subtree rooted at node i has n nodes, the subtree rooted at either of its children has size at most $2n/3$—the worst case occurs when the last row of the subtree rooted at node i is exactly half full.

3.2 Exercise *Prove this last statement. Note that in any case, any subtree of a pre-heap that consists of an element and all its descendants is also a pre-heap. Then what you have to prove is that if a pre-heap has n nodes, then either subtree of the root has at most $2n/3$ nodes.*

So the running time $T(n)$ can be characterized by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1)$$

This falls into case 2 of the "master theorem", and so we must have

$$T(n) = O(\log n)$$

(Well, actually there is something to be considered here: the master theorem deals only with recursive equalities, and this is an inequality. The Remark after the Master Theorem in Lecture 2 handles this, however.)

Another way to express this is to say that the running time of HEAPIFY(A, i) is

$$O(\text{the height of the element } A[i])$$

4 Building a heap

Now that we have the helper routine HEAPIFY, we can use it to construct a heap very easily: We just start with an array A of numbers (organized in no particular order) and we rearrange the numbers in the array to form a heap. We do this starting at the "far end" of the array and working back to the beginning:

```
BUILD-HEAP(A)
    A.heapsize ← A.length
    for i ← ⌊A.length/2⌋ down to 1 do
        HEAPIFY (A, i)
```

For the proof of correctness, we use the following inductive hypothesis:

4.1 Lemma (The inductive hypothesis) *At the start of each iteration of the **for** loop, each node $i+1, i+2, \ldots, n$ is the root of a heap.*

PROOF. On the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf and is thus the root of a trivial heap. Thus the inductive hypothesis is true for the first iteration of the loop.

For the inductive step—going from i to $i-1$ (!), we are allowed to assume that each element $i+1, i+2, \ldots, n$ is the root of a heap. So in particular, each of the two children of node i (i.e., nodes $2i$ and $2i+1$) is the root of a heap. Therefore[3] the call to HEAPIFY(A, i) makes i the root of a heap. Further, all nodes which are not descendants of i are untouched by the call to HEAPIFY(A, i), (Do you see why?) and so we can conclude that each node $i, i+1, \ldots, n$ is now the root of a heap. □

[3] This is by the result of Exercise 3.1 on page 8.

Thus when the algorithm terminates, $i = 0$ and so each node $i = 1, 2, \ldots n$ is the root of a heap, and in particular node 1 is the root of a heap. This concludes the proof of correctness of the BUILD-HEAP algorithm.

4.2 Exercise *Rewrite the inductive hypothesis as a sequence of statements, as we have done before, and then rewrite the proof using that sequence.*

4.1 The cost of BUILD-HEAP—a remarkable result

What is the running time of BUILD-HEAP? Here's a simple estimate: there are at most n calls to HEAPIFY, and each one costs $O(\log n)$. Therefore the running time of BUILD-HEAP is $O(n \log n)$.

This is true, but it's actually an over-estimate. The reason is that the running time of the call to HEAPIFY(A, i) is linear in the height of the element $A[i]$ on which it is called. And most of these elements have very small heights. So we can do better, as follows:

Remember that the number of elements of the heap at height h is $\leq \frac{n}{2^h}$, and the cost of running HEAPIFY on a node of height h is $O(h)$. The root of a heap of n elements has height $\lfloor \log_2 n \rfloor$. Therefore the worst-case cost of running BUILD-HEAP on a heap of n elements is bounded by

$$\sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^h} O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \right) = O(n)$$

since the sum converges, so we don't care what the upper bound of the summation is.

4.3 Exercise *How do we know the sum converges?*

This is really remarkable, for the following reason: Arranging data in a heap doesn't tell us everything about the data—as we have seen, we can do this in more than one way—but it does tell us something. For instance, it tells us immediately what the greatest element is, and as we'll see in a moment, it tells us really a lot more.

But if we're willing to settle for the somewhat limited information a heap can tell us, we get something important in return: heaps are really cheap to construct. Any data structure whose cost of construction is O(n) is worth remembering.

This $O(n)$ cost for building a heap is the key to understanding why heaps are so useful.

5 Heapsort

Now that we know how to build a heap, we can use it to actually sort an array in place (that is, without using any significant additional memory).

```
HEAPSORT (A)
    BUILD-HEAP (A)
    for i ← A.length down to 2 do
        exchange A[1] ↔ A[i]
        A.heapsize ← A.heapsize − 1
        HEAPIFY (A, 1)
```

5.1 Exercise *Prove that* HEAPSORT *actually does sort the array* A.

Analysis of the running time:

The call to BUILD-HEAP takes time $O(n)$. Each of the $n-1$ calls to HEAPIFY takes time $O(\log n)$. Hence the total running time is (in the worst case) $O(n \log n)$.

This is a great result. It says that the worst-case running time of HEAPSORT is $O(n \log n)$. And further, HEAPSORT uses no additional memory—this makes it much better than MERGE-SORT, which we considered in Lecture 1.

6 Priority queues

It turns out, however, that heaps are not typically used for sorting—QUICKSORT, which we will consider in the next lecture, turns out to be a bit better in practice (although the analysis of QUICKSORT turns out to be quite a bit more complex). Heaps *are* often used however, to implement priority queues:

Definition *A* priority queue *is a data structure that maintains a set S of elements, each with an associated value called a* key. *(As usual, the keys must be comparable.) The priority queue supports the following operations:*

INSERT(S, x) *inserts the element x into the set S.*

MAXIMUM(S) *returns the element of S with the largest key.*

EXTRACT-MAX(S) *removes and returns the element of S with the largest key.*

INCREASE-KEY(S, x, k) *increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.*

A heap is a great data structure for implementing a priority queue. Here is how we can implement the four priority queue operations using a heap:

6.1 HEAP-MAXIMUM

```
HEAP-MAXIMUM(A)
    return A[1]
```

Clearly, the running time is $\Theta(1)$.

6.2 HEAP-EXTRACT-MAX

```
HEAP-EXTRACT-MAX(A)
    if A.heapsize < 1 then
        error "heap underflow"
```

$max \leftarrow A[1]$
$A[1] \leftarrow A[A.heapsize]$
$A.heapsize \leftarrow A.heapsize - 1$
HEAPIFY$(A, 1)$
return max

Here the running time is dominated by that of the call to HEAPIFY, so it is $O(\log n)$.

6.3 HEAP-INCREASE-KEY

For HEAP-INCREASE-KEY(A, i, key), we just increase the key of $A[i]$, and then let that node "float up" to its proper position.

HEAP-INCREASE-KEY(A, i, key)
  if $key < A[i]$ **then**
    error "new key is smaller than current key"
  $A[i] \leftarrow key$
  while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$ **do**
    exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
    $i \leftarrow \text{PARENT}(i)$

Figure 4 shows how this works—we apply HEAP-INCREASE-KEY$(A, 9, 15)$:

The running time is $O(\log n)$, since the tree has height $O(\log n)$.

6.4 HEAP-INSERT

HEAP-INSERT(A, key)
  $A.heapsize \leftarrow A.heapsize + 1$
  $A[A.heapsize] \leftarrow -\infty$
  HEAP-INCREASE-KEY$(A, A.heapsize, key)$

The running time here is again $O(\log_2 n)$.

Thus, a heap supports any priority queue operation on a set of size n in $O(\log n)$ time.

6.1 Exercise *Show that there is an algorithm that produces the k smallest elements of an unsorted set of n elements in time $O(n + k \log n)$.*

Be careful: To do this problem correctly, you have to do two things:

1. *State the algorithm carefully and prove that it does what it is supposed to do. (The proof can be very simple.)*
2. *Prove that the algorithm runs in time $O(n + k \log n)$.*

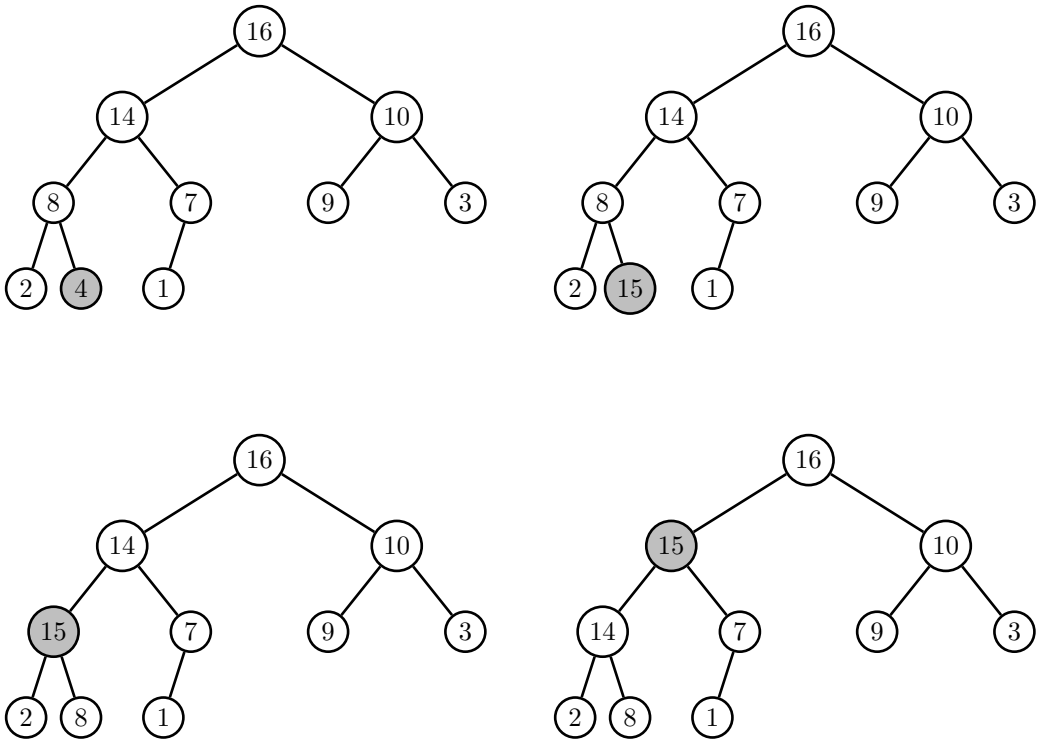


Figure 4: An example of HEAP-INCREASE-KEY. $A[9]$ (which initially has key 4) has its key increased to 15 by applying HEAP-INCREASE-KEY$(A, 9, 15)$.