

## Building heaps

Given an array  $A$  of *elements* with *priorities*, whose only *empty slots* are at the far *right*, then we can view  $A$  as a

This however, does *not* guarantee that  $A$  is a heap *unless* the elements have a *certain ordering*.

There are several options for making  $A$  into a *heap*:

1.  $A$
- 2.
3. Use **HEAPIFY**:
  - Notice that *every item* in the *second half* of  $A$  corresponds to a *leaf* in the tree represented by  $A$ ,
  - starting at the "*middle*" element (i.e., the first *nonleaf node* in the tree represented by  $A$ ),
  - We simply call **HEAPIFY** on each position of the array, working back towards position 1.

Each of these options has *complexity*:

1.

2.

3.

```
BUILD-HEAP (A)
  heapsize := size(A);
  for i := floor(heapsize/2) downto 1 do
    HEAPIFY (A, i);
  end for
END
```

**Q:** Why are we *guaranteed* that the *preconditions* for HEAPIFY are met before each call?

**Example:** If  $A = [1, 5, 7, 6, 2, 9, 4, 8]$ , then `BUILD-HEAP(A)` makes the following sequence of calls to `HEAPIFY` (you can check the result of each one by tracing it):

`HEAPIFY([1, 5, 7, 6, 2, 9, 4, 8], 4) =`

`HEAPIFY([1, 5, 7, 6, 2, 9, 4, 8], 3) =`

`HEAPIFY([1, 5, 7, 6, 2, 9, 4, 8], 2) =`

`HEAPIFY([1, 5, 7, 6, 2, 9, 4, 8], 1) =`

**Q:** How *many* calls do we make to `HEAPIFY`?

**Q:** How *long* does each one take?

Therefore, we get a *bound* of

But in fact, we can do better by analyzing more carefully.

- We call `HEAPIFY` on each subtree of *height*  $\geq 1$ .
- `HEAPIFY` runs in time *proportional* to the *height* of that subtree.
- Therefore, we can *estimate* the total *running time* as follows:

Q: What does the summation *approach*?

- A tree with  $n$  nodes contains at most nodes of height  $h$  (why?), so it contains at *most* the *same number of subtrees of height  $h$* .

Therefore, using the fact that (p. 135 of CLRS):

$$\sum_{h=0}^{\infty} \frac{h}{2^h} \leq 2$$

the *running time* is:

So BUILD-HEAP runs in time

## Complexity of Prim's Algorithm

The main loop is:

```
while ( not ISEMPTY(Q) ) do
  u := EXTRACT-MIN(Q);
  if p[u]  $\neq$  NIL then A := A U {(p[u],u)};
  for each v in adjacency-list[u] do
    if v in Q and w(u,v) < priority[v] then
      DECREASE-PRIORITY(v, w(u,v));
      p[v] := u;
    end if
  end for
end while
```

**Q:** What would you expect the *complexity* of DECREASE-PRIORITY to be?

**Q:** How many times does the *while loop* iterate?

**Q:** How many times do we *consider* each *edge*?

To answer this let's look at what happens for  $u$  the *current* vertex.

Consider the *current* vertex  $u$  that gets *inserted* into  $A$ :

- Which *edges* does the algorithm *consider*?
- How *many times* can each of these *edges* be looked at?

- What is the *worst case complexity* when considering each *edge*?
- What is the overall *loop complexity* then?
- What is the *complexity* of *building* the initial *heap* take?

Therefore, the *worst-case* running time is .

## Heap Sort

How can we use a *heap* to *sort* an array?

```
HEAP-SORT(array A) {
```

```
}
```