Heapsort

- **Heap**: a binary tree $T$ that satisfies
  1. $T$ is complete through depth $h-1$
  2. All paths to a leaf of depth $h$ are to the left of all paths to a leaf of depth $h-1$, i.e., leaves at level $h$ are filled from left to right.
  3. Key at any node is greater than or equal to the keys at each of its children (for maximizing heap). This is called heap property or partial order tree property.
Maintain a heap

Left and right subtrees of $i$ are already heaps
Make subtree rooted at $i$ a heap

```
Heapify(A, i)
  1  l ← left(i); // l = 2i
  2  r ← right(i); // r = 2i+1
  3  if (1 ≤ n and A[l] > A[i])
     4      then largest ← l;
     5      else largest ← i;
  6  if (r ≤ n and A[r] > A[largest])
     7      then largest ← r;
     8  if (largest ≠ i)
     9      then exchange A[i] ↔ A[largest];
    10      Heapify(A, largest);
```

Analysis:
- time is proportional to height of $i$
  - $O(\log n)$
Construct a Heap
- Convert an array A[1..n] into a heap
- Elements from n/2 +1 to n correspond to leaves, and themselves are 1-element heaps already.

constructHeap(A)
1. For (i= floor(n/2); i > 1; i--)
2. heapify(A, i);
Analysis of constructHeap
n calls to heapify = n O(lg n) = O(n lg n)

A tighter analysis
\[ T(n) = T(n-r-1) + T(r) + 2 \lg(n) \]
\[ = 2T((n-1)/2) + 2 \lg(n) \quad // \text{complete binary tree} \]
\[ \in \Theta(n) \quad // \text{master theorem} \]

Heapsort

Heapsort(A)
int heapsize = n;
1. constructHeap(A);    // O(n)
2. for(i = heapsize; i > 2; i--)
3. heapsize = heapsize – 1;     // O(1)
4. Heapify(A, 1);             // O(lg n)

Time analysis:
T(n) = O(n) + O(n lg n) = O(n lg n)
A more exact analysis

\[ T(n) = \Theta(n) + 2 \sum_{i=1}^{n-1} \log i \leq \Theta(n) + 2 \int_1^n (\log e) \ln x \, dx = \Theta(n) + 2 (\log e) (n \ln n - n) = \Theta(n) + 2(n \log(n) - 1.443 n) = 2n \log(n) + O(n) \]

Heap used as priority queue

- Insert(S,x) inserts element x into the set S (time:  )
- Maximum(S) returns the element of S with the largest key (time:  )
- Extract-Max(S) removes and returns the element of S with the largest key (time:  )

A heap can support any priority-queue operations on a set of size n in _ time.
Heap-extract-max(A)
1. If heap-size(A) < 1
2. then error “heap underflow”
3. max = A[1];
5. Heap-size(A) = heap-size(A) -1
6. Heapify(A, 1)
7. Return max;

Note: Heap-extract-max takes only O(lg n) time

Heap-insert(A, key)
1. heap-size(A) = heap-size(A) + 1
2. i = heap-size(A)
3. while (i > 1 and A[parent(i)] < key)
5. i = parent(i)
6. A[i] = key

Note: this is bubble-up heap, only requires one comparison at each level to float a big key to its right position (in contrast to heapify which requires two comparisons to filter down a small key)
Algorithm | Worst-case | Average | Space usage
--- | --- | --- | ---
Insertionsort | n^2/2 | \(\Theta(n^2)\) | in place
Quicksort | n^2/2 | \(\Theta(n \log n)\) | extra space \log n
Mergesort | n lg n | \(\Theta(n \log n)\) | extra space n
Heapsort | 2n lg n | \(\Theta(n \log n)\) | in place
Accl. Heapsort | n lg n | \(\Theta(n \log n)\) | in place

Heap-insert(A, 15)