Lecture 6, Feb 1, 2011, sorting algorithms II

We study *heap sort* in this class.

The *heap* data structure is an array object that can be viewed as a complete binary tree, as shown in the above figure. The tree is completed filled on all levels except possibly the lowest, which is filled from the left up to a point.

The root of the tree is $A[1]$, and given the index i of a node, the indices of its parent $\text{PARENT}(i)$, left child $\text{LEFT}(i)$, and right child $\text{RIGHT}(i)$ can be computed by

$$\begin{aligned} \text{PARENT}(i): &\quad \textbf{return } \text{floor(i/2)} \\ \text{LEFT }(i): &\quad \textbf{return } 2i \\ \text{RIGHT }(i): &\quad \textbf{return } 2i+1 \end{aligned}$$

Heaps also satisfy the *heap property*: for every node other than the root,

$$A[\text{PARENT}(i)] \geq A[i],$$

that is, the value of a node is at most the value of its parent. Thus, the largest element in the heap is stored at the root, and the subtrees rooted at a node contain smaller values than does the node itself.

An array A that represents a heap has two attributes:

(1) *length(A)*: equals to the number of elements stored in the array.
(2) *heap-size(A)*: equals to the number of elements in the heap stored within array A.

Example, consider two arrays

(1) $A =< 16, 4, 10, 2, 3, 8, 9 >$, $\text{length}(A) = 7$, $\text{heap-size}(A) = 7$.
(2) $B =< 16, 10, 4, 2, 3, 8, 9 >$, $\text{length}(B) = 7$, $\text{heap-size}(B) = 5$.

Algorithm HEAPIFY:

(1) Inputs: an array A and an index i into the array.
(2) Assumption: the binary trees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are heaps ($A[i]$ might violates the heap property).

```
1  ℓ ← LEFT(i)
2  r ← RIGHT(i)
3  if ℓ ≤ heap-size[A] and A[ℓ] > A[i]
4     largest ← ℓ
5  else largest ← i
6  if r ≤ heap-size[A] and A[r] > A[largest]
7     largest ← r
8  if largest ≠ i
9     exchange [A] ↔ A[largest]
```

10 heapify$(A, largest)$

At each step, the largest of the elements $A[i]$, $A[\text{LEFT}(i)]$, $A[\text{RIGHT}(i)]$ *is determined, and its index is stored in* $largest$. *If* $A[i]$ *is largest, then the subtree rooted at node* i *is a heap and the procedure terminates. Otherwise, one of the two children has the largest element, and* $A[i]$ *is swapped with* $A[largest]$, *which causes node* i *and its children to satisfy the heap property. The node* $largest$, *however, now has the original value* $A[i]$, *and thus the subtree rooted at* $largest$ *may violate the heap property. Consequently, heapify must be called on that subtree.*

Example: Consider an array $A =< 16, 4, 10, 14, 7, 9, 3, 2, 8, 1 >$. Assume heap-size$(A) = 10$, then what are the actions of heapify$(A, 2)$?

(1) $largest = 4$
(2) Swap $A[2]$ and $A[4]$

$$\Rightarrow < 16, \underline{14}, 10, \underline{4}, 7, 9, 3, 2, 8, 1 >$$

(3) Call heapify$(A, 4)$
(4) $largest = 9$
(5) Swap $A[4]$ and $A[9]$

$$\Rightarrow < 16, 14, 10, \underline{8}, 7, 9, 3, 2, \underline{4}, 1 >$$

(6) Call heapif$(A, 9)$

Algorithm BUILD-HEAP

1 heap-size$[A] \leftarrow$ length(A)
2 **for** $i \leftarrow$ floor(length$(A)/2), \ldots, 1$
3 call heapify(A, i)

We use the algorithm heapify in a bottom-up manner to convert the input into a heap. Notice that starting from $A[\text{floor}(n/2)]$, *all the elements in the array are leaves, each of which is a 1-element heap.*

Example: Consider an array $A =< 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 >$. What are the actions of build-heap(A)?

(1) $i = 5$, call heapify$(A, 5)$, A doesn't change.
(2) $i = 4$, call heapify$(A, 4)$,

$$\Rightarrow A =< 4, 1, 3, \underline{14}, 16, 9, 10, \underline{2}, 8, 7 >$$

call heapify$(A, 8)$ and A doesn't change.
(3) $i = 3$, call heapify$(A, 3)$,

$$\Rightarrow A =< 4, 1, \underline{10}, 14, 16, 9, \underline{3}, 2, 8, 7 >$$

call heapify$(A, 7)$ and A doesn't change.

(4) $i = 2$, call heapify$(A, 2)$,

$$\Rightarrow A =< 4, \underline{16}, 10, 14, \underline{1}, 9, 3, 2, 8, 7 >$$

call heapify$(A, 5)$,

$$\Rightarrow A =< 4, 16, 10, 14, \underline{7}, 9, 3, 2, 8, \underline{1} >$$

call heapify$(A, 10)$ and A doesn't change.

(5) $i = 1$, call heapfiy$(A, 1)$,

$$\Rightarrow A =< \underline{16}, \underline{4}, 10, 14, 7, 9, 3, 2, 8, 1 >$$

call heapify$(A, 2)$,

$$\Rightarrow A =< 16, \underline{14}, 10, \underline{4}, 7, 9, 3, 2, 8, 1 >$$

call heapify$(A, 4)$,

$$\Rightarrow A =< 16, 14, 10, \underline{8}, 7, 9, 3, 2, \underline{4}, 1 >$$

call heapify$(A, 9)$ and A doesn't change.

Algorithm HEAPSORT

```
1  build-heap(A)
2  for i ← length(A), ..., 2
3      exchange A[1] ↔ A[i]
4      heap-size[A] ← heap-size(A) − 1
5      heapify(A, 1)
```

We state without proof that the complexity of heap sort is $O(n \log n)$.

Comparison between heap sort and merge sort:

(1) auxiliary space for merge sort
(2) parallelization
(3) cache behavior