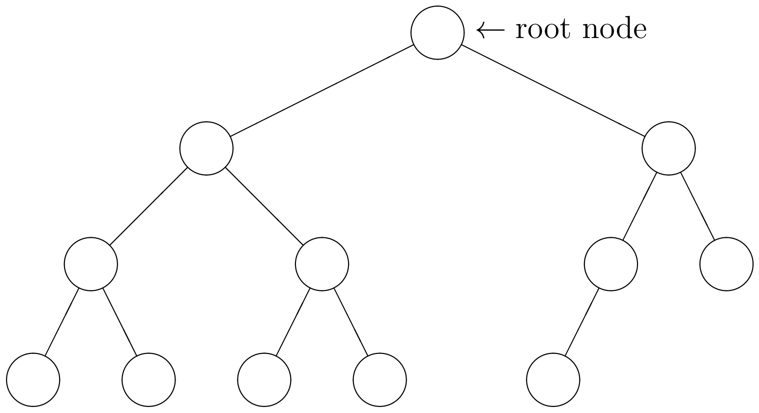
COMP 460 Lecture Notes

R. I. Greenberg

1 Heapsort

- Uses data structure called a *(binary) heap* that can be viewed as a binary tree. (See CLRS 3rd ed. Sec. B.5.3 of for tree terminologies.)



  Each node connects up to two “children” (drawn below it), a left and/or right child. Each node except the root has one parent.

  Each node holds one of the data values to be sorted.

  In a heap, the tree is nearly complete, i.e., each row is full of nodes except that the last row can be empty to the right of some point.

  The heap can actually be represented by an array:

  - root at position 1
  - left child of i at $2i$
  - right child of i at $2i + 1$

  This implies that for $i > 1$, the parent of i is at $\lfloor i/2 \rfloor$.

- HEAPSORT *sorts in place*, i.e., it requires only a constant amount of storage outside of the input array. This was true of insertion sort but not merge sort (w. naive merge).

- Sometimes, only a portion of the array belongs to the heap:
  - length$[A] =$ no. of elts. in array A
  - heap-size$[A] =$ no. of elts. in heap
- **The heap property:**

$$A[\textsc{Parent}(i)] \geq A[i] \; .$$

1.1 A Key Procedure: $\textsc{Heapify}(A, i)$

Given that left and right subtrees below i are heaps, make subtree at i into a heap.

Compare $A[i]$ with its two children. Swap largest of the 3 values into $A[i]$, and if it came from one of the children, step down tree in that direction, and recur.

The running time of HEAPIFY is $O(h)$ for a node of height h.

(The height of a node is the no. of edges on a longest path from the node down to a leaf, and it is $O(\lg n)$ for any node in an n elt. heap.)

1.2 Building a Heap

Make entire array into a heap by applying HEAPIFY in a bottom-up fashion:

```
   BUILD-HEAP($A$)
1  heap-size[$A$] ← length[$A$]
2  for $i$ ← ⌊length[$A$]/2⌋ downto 1
3      HEAPIFY($A, i$)
4  endfor
```

Time of BUILD-HEAP is certainly

$$\leq \frac{n}{2} \cdot O(\lg n) = O(n \lg n)$$

based on the maximum time for HEAPIFY.

Actually, the time is:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil n/2^{h+1} \right\rceil \cdot O(h) \qquad \text{based on time for HEAPIFY on a node of height } h$$

$$\leq \; O(n \sum_{h=0}^{\infty} h/2^h)$$

$$= \; O(n) \text{ by differentiating a geom. series as in Section A.1 of CLRS 3rd ed.}$$

1.3 Heapsort Algorithm

Build a heap. Then repeatedly swap first (max.) elt. in heap with last and rebuild a heap that is one elt. smaller.

Rebuild is just one call of Heapify$(A, 1)$ after decrementing heap-size$[A]$.

Heapsort time is $\underset{\text{Build-Heap}}{O(n)} + \underset{\substack{n-1 \text{ calls to} \\ \text{Heapify}}}{(n-1)O(\lg n)} = O(n \lg n)$.

1.4 Priority Queues

In practice, Heapsort is probably not quite the fastest sorting method , but heaps have another application:

A priority queue supports:

- Insert(S, x): Insert elt. x into set S.
- Maximum(S): Return largest elt. of S.
- Extract-Max(S): Remove and return largest elt. of S.
- Increase-key(S, x, k): Increase value of elt. x's key to k

Applications:

- scheduling jobs on a shared computer according to priority
- scheduling events in a simulator
- ...

Jobs (or events) can arise at any time and be added to queue. When ready to service a job (or simulate an event), pull out the one with highest priority (or earliest time of occurrence).

Implementations:

- Maximum: Just read off $A[1]$. Time: $\Theta(1)$.
- Extract-Max: After reading $A[1]$, replace it with last elt. of heap and use Heapify to rebuild heap one size smaller.
  Time: dominated by call of Heapify: $O(\lg n)$.
- Insert: Increase heap size by 1 and put new elt. at end. Then walk up tree to slip it into correct place (like adding an elt. in Insertion-Sort).
  Time: $O(\lg n)$
- Increase-Key: Just increase the key value and walk up as for Insert.