Design and Analysis of Algorithms Lab (18ISL47)

Department of Information Science and Engineering
Gogte Institute of Technology

Experiment-4

Implement Heap Sort algorithm and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted and plot a graph of the time taken versus n.

Transform and Conquer

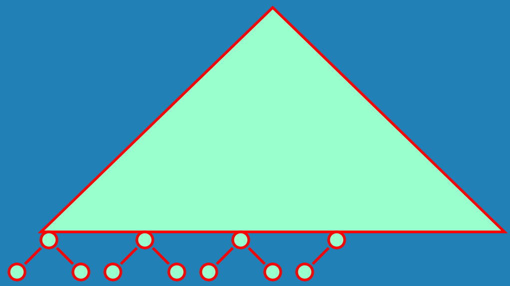
This group of techniques solves a problem by a *transformation* to

- a simpler/more convenient instance [eg sorted] of the same problem (*instance simplification*)
- a different representation of the same instance [eg different data structure] (*representation change*)
- a different **problem** for which an algorithm is **already available** (*problem reduction*)

Heaps and Heapsort

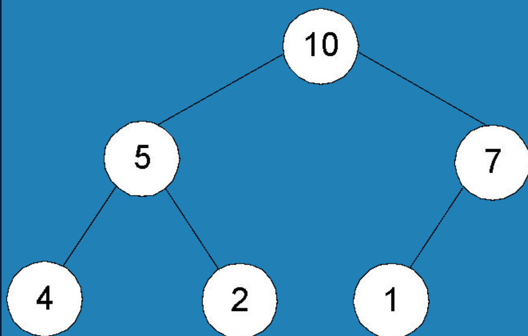
Definition A *heap* is a binary tree with keys at its nodes (one key per node) such that:

- It is essentially complete, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing

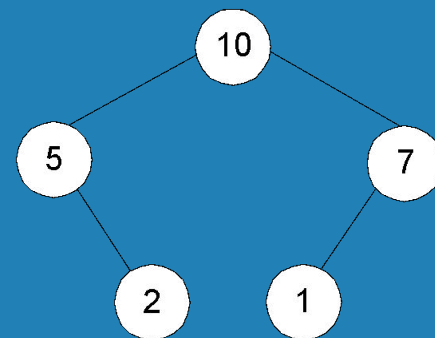


- The key at each node is $\geq$ all keys in its children (and descendents)

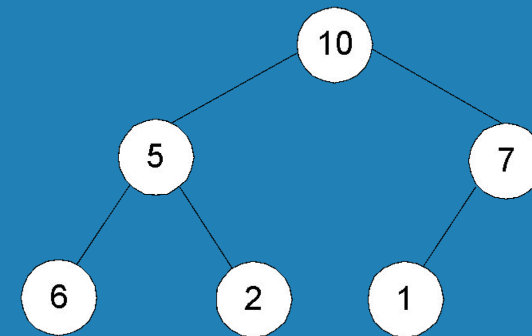
Illustration of the heap's definition



a heap



not a heap



not a heap

Note: Heap's elements are ordered top down (along any path down from its root), but they are not ordered left to right

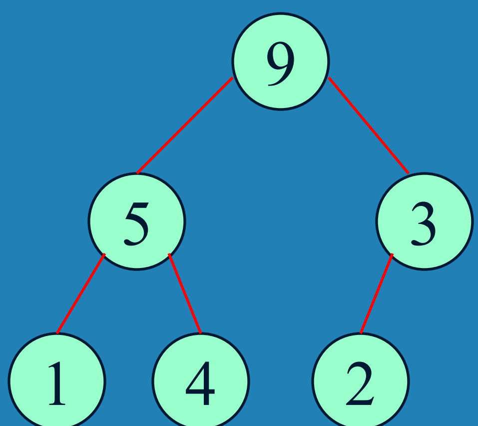
Some Important Properties of a Heap

- Given n, there exists a unique binary tree (structure) with n nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key
- Every subtree rooted at every node of a heap is also a heap
- A heap can easily be represented as an array (and usually is). Look at the following example, and then explain why the array representation always works.

Heap's Array Representation

Store heap's elements in an array (whose elements indexed, for convenience, 1 to n) in top-down left-to-right order

Example:



1	2	3	4	5	6
9	5	3	1	4	2

- Left child of node j is at $2j$
- Right child of node j is at $2j+1$
- Parent of node j is at $\lfloor j/2 \rfloor$
- Parental (ie interior) nodes are in the first $\lfloor n/2 \rfloor$ locations

Heapsort

Stage 1: Construct a heap for a given list of *n* keys

Stage 2: Repeat operation of root removal *n*-1 times:

- **Exchange keys in the root and in the last (rightmost) leaf**
- **Decrease heap size by 1**
- **If necessary, repeatedly swap new root (node) with larger child until the heap condition again holds**

Heap Construction (bottom-up)

Step 0: Initialize structure (ie array) with keys in order given

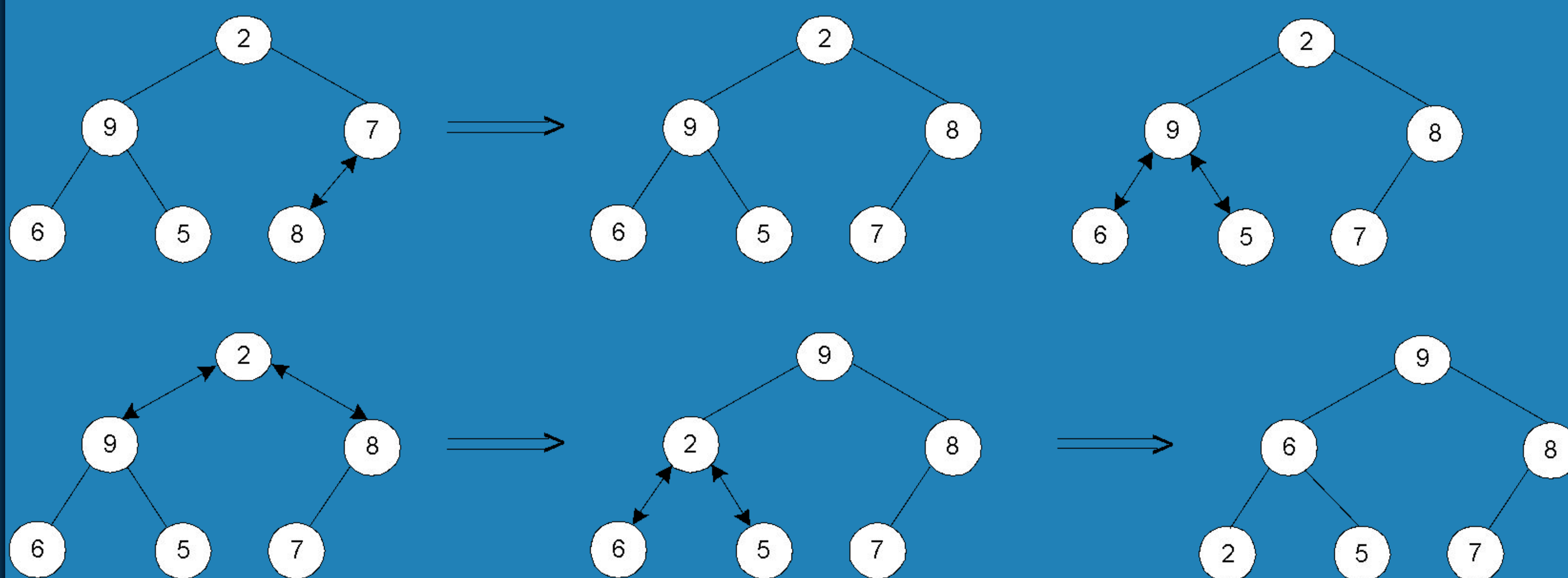
Step 1: Starting with the last (rightmost) parental (ie interior) node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds

Step 2: Repeat Step 1 for the preceding parental node

Step 1 called Heapify:

Example of Heap Construction [Bottom Up]

Construct a heap for the list 2, 9, 7, 6, 5, 8 . Insert elements into the array. Process interior nodes R to L. Why? Before and after each step? Heapify all the way down at each step. Bottom up?



Worst case? Number of comparisons for a node??

Pseudocode of bottom-up heap construction

Algorithm $HeapBottomUp(H[1..n])$
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
    $k \leftarrow i; \quad v \leftarrow H[k]$
    $heap \leftarrow$ **false**
    while not $heap$ **and** $2 * k \leq n$ **do**
        $j \leftarrow 2 * k$
        if $j < n$ //there are two children
            if $H[j] < H[j+1] \quad j \leftarrow j + 1$
        if $v \geq H[j]$
            $heap \leftarrow$ **true**
        else $H[k] \leftarrow H[j]; \quad k \leftarrow j$
    $H[k] \leftarrow v$

Example of Sorting by Heapsort

Sort the list 2, 9, 7, 6, 5, 8 by heapsort

Stage 1 (heap construction)

2 9 7 6 5 8
2 9 8 6 5 7
2 9 8 6 5 7
9 2 8 6 5 7
9 6 8 2 5 7

Stage 2 (root/max removal)

9 6 8 2 5 7
7 6 8 2 5 | 9
8 6 7 2 5 | 9
5 6 7 2 | 8 9
7 6 5 2 | 8 9
2 6 5 | 7 8 9
6 2 5 | 7 8 9
5 2 | 6 7 8 9
5 2 | 6 7 8 9
2 | 5 6 7 8 9

Analysis of Heapsort

Stage 1: Build heap for a given list of *n* keys (of height h)

worst-case

$$C(n) = \sum_{i=0}^{h-1} 2(h-i)\, 2^i \quad = \quad 2\,(\,n - \log_2(n+1)) \;\in\; \Theta(n)$$

nodes at level *i*

Stage 2: Repeat operation of root removal *n*-1 times (fix heap)

worst-case

$$C(n) = \sum_{i=1}^{n-1} 2\log_2 i \;\in\; \Theta(n\log n)$$

Both worst-case and average-case efficiency: $\Theta(n\log n)$

In-place: yes

Stability: no (e.g., apply heapsort to 1 1)