CS 440 Theory of Algorithms / CS 468 Algorithms in Bioinformatics

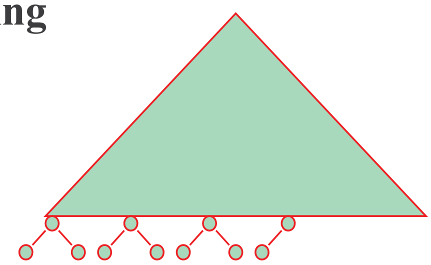
Transform-and-Conquer

Heaps and Heapsort

Heaps and Heapsort

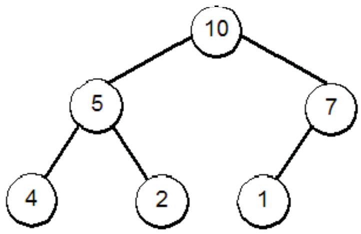
Definition A *heap* is a binary tree with keys at its nodes (one key per node) such that:

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

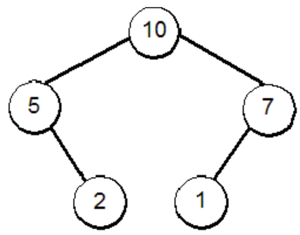


- The key at each node is ≥ keys at its children

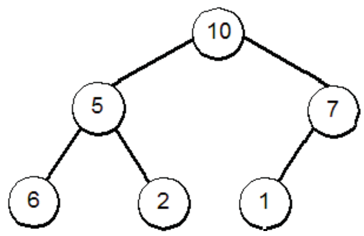
Illustration of the heap's definition



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

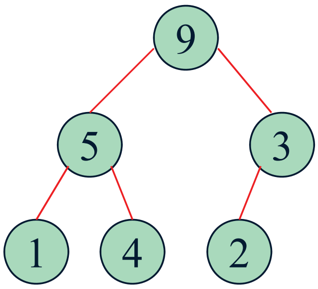
Some Important Properties of a Heap

- **Given *n*, there exists a unique binary tree with *n* nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$**
- **The root contains the largest key**
- **The subtree rooted at any node of a heap is also a heap**
- **A heap can be represented as an array**

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 nodes are represented in the first $\lfloor n/2 \rfloor$ locations

Heap Construction (bottom-up)

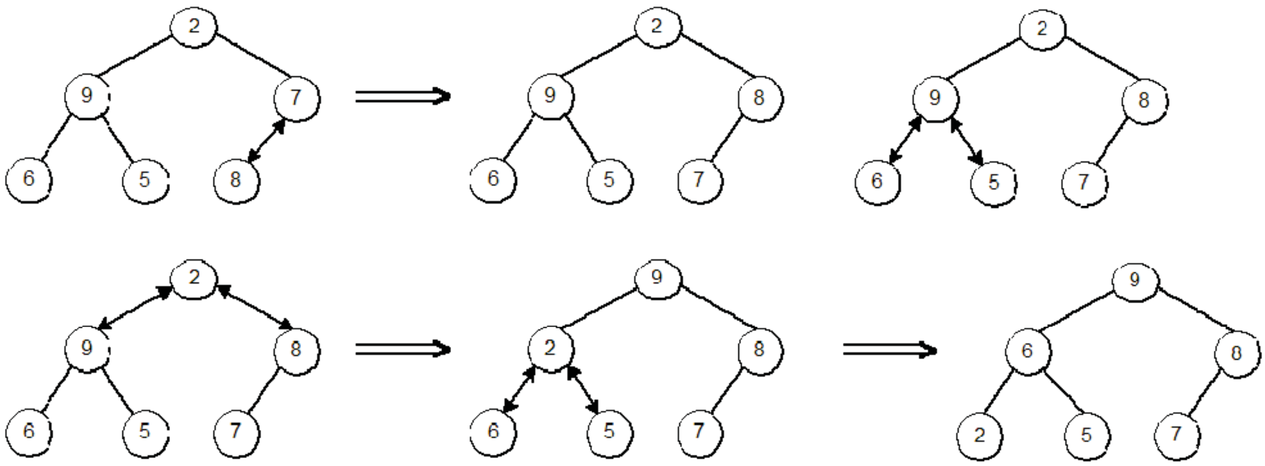
Step 0: Initialize the structure with keys in the order given

Step 1: Starting with the last (rightmost) parental 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

Example of Heap Construction

Construct a heap for the list 2, 9, 7, 6, 5, 8



Pseudopodia 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 $H[j] < H[j+1]$ $j \leftarrow j+1$
        if $v \geq H[j]$
            $heap \leftarrow$ **true**
        else $H[k] \leftarrow H[j];\quad k \leftarrow j$
    $H[k] \leftarrow v$

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, swap new root with larger child until the heap condition holds**

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

worst-case

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

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

Priority Queue

A *priority queue* is the ADT of a set of elements with numerical priorities with the following operations:

- find element with highest priority
- delete element with highest priority
- insert element with assigned priority (see below)

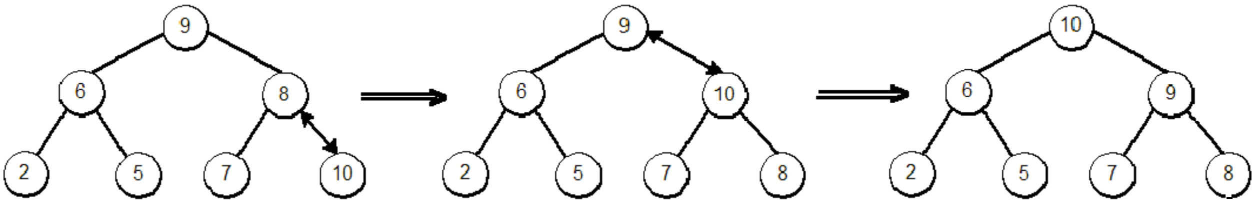
- Heap is a very efficient way for implementing priority queues

- Two ways to handle priority queue in which highest priority = smallest number

Insertion of a New Element into a Heap

- **Insert the new element at last position in heap.**
- **Compare it with its parent and, if it violates heap condition, exchange them**
- **Continue comparing the new element with nodes up the tree until the heap condition is satisfied**

Example: Insert key 10



Efficiency: O(log *n*)

Other Important Notes on Heaps

- **Min Heap**
  - **The key at each node is ≤ keys at its children**

- **Heap is useful for obtaining the *m* smallest or largest from *n* items when *m* is much smaller than *n***
  - **E.g., building MSTs**
    we only need |V| – 1 edges, but |E| could be an big as |V| (|V| - 1) / 2
  - **What is the efficiency?**

Priority Queue and Java

- java.util.PriorityQueue
- Not supported before Java 1.5
- Insertion: offer()
- Deletion: poll()
- Sample code next page

/* License for Java 1.5 'Tiger': A Developer's Notebook (O'Reilly) example package, Java 1.5 'Tiger': A Developer's Notebook (O'Reilly) by Brett McLaughlin and David Flanagan. ISBN: 0-596-00738-8. You can use the examples and the source code any way you want, but please include a reference to where it comes from if you use it in your own products or services. Also note that this software is provided by the author "as is", with no expressed or implied warranties. In no event shall the author be liable for any direct or indirect damages arising in any way out of the use of this software.*/ /* The program is modified from the source stated above. */

```
import java.util.Comparator;
import java.util.PriorityQueue;

public class PriorityQueueTester {
  public static void main(String[] args) {
    PriorityQueue<Integer> pq =
      new PriorityQueue<Integer>(20,
        new Comparator<Integer>() {
          public int compare(Integer i, Integer j) {
            int result = i - j;
            return result;
          }
        }
      );
// Fill up with data, in an odd order
    for (int i=0; i<20; i++) {
      pq.offer(20-i*2);
    }

    // Print out and check ordering
    for (int i=0; i<20; i++) {
      System.out.println(pq.poll());
    }
  }
}
```