3/30

Heap & Heapsort:

Heap is a binary tree whose keys satisfy the following heap property:

The key of every node is greater than or equal to the key of any its children.

Heaps are useful to implement priority queue, an abstract data type defined by the following two operations:

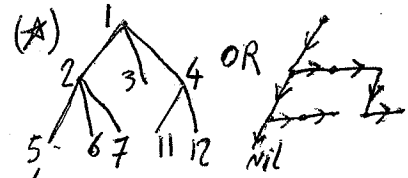
Insert(x): insert a key x into the data structure.

(max) Remove(): remove the largest key from the data structure and return it.

(Priority queues are like queues but when we dequeue - the highest-priority element is retrieved first.

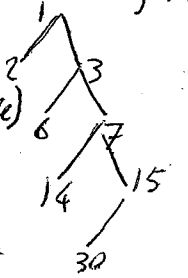
In general trees can be represented explicitly or implicitly.

Explicitly a node with k children is a record containing an array of k pointers and one pointer to the parent. Alternatively, we can keep two pointers: first to the first child and the second to the next sibling (two linked



Implicitly an array is used. Consider a binary tree. The root is stored in A[1], the children in A[2] and A[3] and in general inductively the left child of a node v stored in A[i] is stored in A[2i] and the right child is in A[2i+1]. (for Ternary we store at A[3i-1], A[3i] and A[3i+1] (see fig (*) above).

The advantage is that no pointer is needed which saves storage, however if the tree is unbalanced, i.e., some leaves are much farther away from the root than the others, the many non-existing nodes must be represented (array of size 30 is needed for 8 nodes, see)



Heaps can be represented using Explicit or Implicit tree representation, however since we can ensure that heaps will be balanced. We assume the array is A[1..k] where k is an upper bound on the number of elements the heap will ever contain.

First Remove: by the heap property, the node with the largest key in a heap is the root A[1]. we remove it, take the leaf A[n] and delete it and put it in place of the root (i.e. A[1]:=A[n] and n:=n-1). we have two separate heaps and x at the root. we now propagate x down the tree until it reaches a subtree for which it is a maximum. First we find the max of children and if it is larger than x, then swap it with x. Inductively continue until either when x becomes the maximal of a subtree or when it reaches a leaf. The maximum number of comparisons is $2\lceil \log_2 n \rceil$.

Insert operation is bottom up (vs. remove which is top-down): increment n by one and insert x as the new leaf A[n]. We then compare the new leaf with its parent, and exchange if the new leaf is larger than its parent. We continue inductively (for correctness) this process, promoting the new key up the tree until the new key is not larger than its parent (or it reaches the root). The maximum number is $\lceil \log_2 n \rceil$.
Overall we can insert and remove in time $O(\log n)$ per operation, however for some operations like search, heaps are not useful.

Heap sort: another fast sorting algorithm, but not as fast as quicksort in practice (though its worst case needs $\Theta(n \log n)$ comparisons). Unlike mergesort, heap sort is an in-place sort.

* First building a heap: given an array A[1..n] of elements in an arbitrary order, rearrange the elements so that the array satisfies the heap property. There are two ways top-down and bottom-up, but we say only top-down which is simpler and efficient. consider scanning the array from left to right.
  IH: The array A[1..i] is a heap.
  The base case is trivial (A[1] is heap). The main part is to incorporate A[i+1] into the heap A[1..i] (exactly inserting A[i+1] into the heap. The number of comparisons is the worst case is $\lfloor \log_2(i+1) \rfloor$. The total number of comparisons is $\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = \Theta(n \log n)$.
* The rest of sort: since A is a heap, then A[1] is the maximum element. Thus we remove from A and put it at A[n] and we continue with A[1..n-1]. The overall running time is $\sum_{i=1}^{n} 2\lceil \log_2(n-i+1) \rceil = \sum_{i=1}^{n} 2\lceil \log i \rceil = \Theta(n \log n)$. So the overall running time is $O(n \log n)$.