

Data Structures G5029

Lecture 7

Kingsley Sage
Room 5C16, Pevensey III
khs20@sussex.ac.uk

© University of Sussex 2006

Lecture 7

- Heap sorting algorithm ...

Array sorting

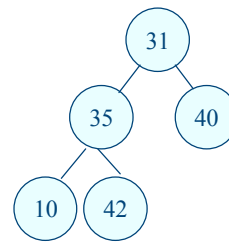
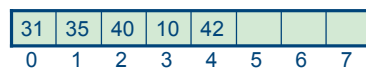
- Array sorting is a fundamental operation in many applications. The requirement is to sort the elements of an array $a[0], \dots, a[n-1]$ into increasing order.
- For this to make any sense in Java, the elements must be comparable with respect to order. So we assume the array elements implement the `Comparable` interface.
- The array itself may be of length greater than n . Our spec only requires that the first n elements be sorted.

Array sorting

- There are a number of efficient algorithms that run in $O(n \log n)$ time on average. Binary heaps provide such an algorithm, and has a number of attractive features.
- A simple implementation would be to insert the array elements one by one into a priority queue, implemented as a binary heap, and then to remove them one by one (although they would come out in reverse order, it would be trivial to deal with this).
- Two basic ways we can improve on this as an implementation:
 - Simple approach requires auxiliary storage for the heap. we can in fact use the original array itself as the heap, providing an "in place" sorting routine.
 - Rather than rebuild the heap every time we insert an item, we can insert all the items first, then restore the ordering.

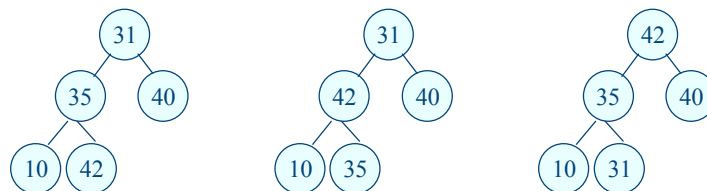
Array sorting

- Suppose we have array [31,35,40,10,42] to sort ...



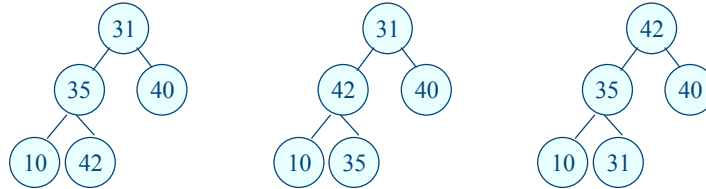
We now have to consider an efficient mechanism for restoring the heap ordering. There is an efficient mechanism we can for restoring heap order which we can apply to the original array. The original array will play the part of our working binary heap, without the need for auxiliary storage.

Building the heap



Apply the demote procedure repeatedly, beginning with parent as the last node in the tree. For the last three nodes 42,10 and 40, there is nothing to do, since a leaf node whose both left and right sub-trees are empty is already a tree. The last non-leaf node holds 35. Applying demote with this node as parent leads to the middle version (exchanging 35 and 42). Now apply demote with parent as the next node working backwards, which is the root node holding 31. Result is the tree shown on the right. The 31 has to sink two layers to the bottom and we're done.

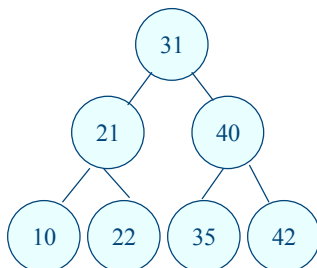
Building the heap



It's no accident that the resulting tree is ordered. You should have noticed that demote when applied to a parent, restores order beneath parent, provided that the left and right sub-trees of parent are already heap ordered. So if we work backwards from the end of the tree applying demote repeatedly, then every time we use demote both left and right sub-trees will already be properly heap ordered. Calling demote finally for the root node completes the "heapification" of the whole tree.

Binary search trees

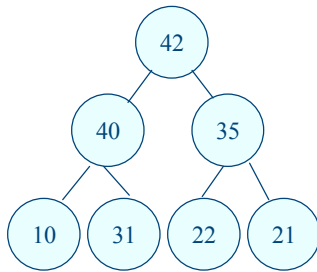
- Remember binary search trees?.



- The items are placed in the tree such that the value stored at each node is greater than the values stored in its left sub-tree, and less than the value stored in its right sub-tree.
- You can think of the items as being ordered from left to right.
- We obtain an interesting alternative structure if the items are ordered from bottom to top ...

Towards the binary heap ...

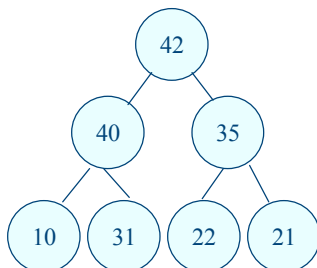
- Ordered from bottom to top ...



- The defining property is now that the value stored at each node of the tree is greater than or equal to the values stored in both sub-trees.
- A simple test for whether this property holds is that the value stored at every node should be no greater than the value stored at its parent (apart from the root which doesn't have a parent). It follows that every path from the root to a leaf forms a decreasing sequence.

Towards the binary heap ...

- The idea of being well-balanced ...



- Trees satisfying the defining property are particularly useful if they are well-balanced, in the sense of having short path lengths from root to leaves
- They don't have to be quite as well-balanced at this tree ...

Sorting by removal

- OK, so now we have converted the original array to a heap. It is now simple to order the the array by repeatedly applying the remove algorithm

42	35	40	10	31			
0	1	2	3	4	5	6	7

31	35	40	10	42			
0	1	2	3	4	5	6	7

40	35	31	10	42			
0	1	2	3	4	5	6	7

10	35	31	40	42			
0	1	2	3	4	5	6	7

- Start off with the heap array
- Max element is always first. Remove says we put last element in first place and then demote until heap order is established again. We can do both re-locations at once by swapping first and last elements.
- We then just focus on the first 4 elements. The last element is already sorted and is no longer in the heap – we are just using the array space as storage.
- Repeat until no heap left and we are just left with the elements stored in the array.

Web demo

- Useful visual demo of heap sorting ...

Next time ...

- Hash functions ...