COS 226 Bob's Notes: Shuffling, Sorting, and Heaps
Fall 2010
R. Tarjan, Preceptor

This edition of the notes contains miscellaneous topics related to sorting. First, though, a remark about representing a deque or related data structure by a circularly linked list, as discussed in the previous set of notes: in iterating over such a list, one needs to be careful about the test to detect the end of the list, since the last item does not contain a null pointer but instead a pointer to the first item. To check if an item is the last one, test whether the access pointer (the pointer that gives access to the whole structure) points to it.

Here is an alternative solution to the problem Kevin posed in lecture, of randomly shuffling a linked list. The method Kevin described is based on the idea of mergesort: split the list in half, randomly shuffle each half, and then merge the two halves. At each step of the merge, if the remaining unmerged parts of the two lists, say A and B, have sizes j and k, respectively, choose as the next element of the merged list the first element on A with probability $j/(j+k)$; otherwise, choose the first element on B. Continue in the way until the merge is complete. A drawback of this method is the need to flip a biased coin (a coin that comes up heads with probability other than ½). An alternative method that requires only unbiased coin flips is based on an idea of Candace Button: instead of imitating mergesort, we imitate quicksort, as follows: If the list contains only one item, there is nothing to do. Otherwise, for each item on the initial list, flip an unbiased coin (which has probability ½ of coming up heads). Move all the items with heads in front of all the items with tails. Randomly shuffle the items with heads by applying the method recursively; randomly shuffle the items with tails by applying the method recursively. This method shuffles in O(NlgN) expected time. Unlike the method based on mergesort, it can run forever (how?), but it runs for a long time with exceedingly small probability.

In precept we observed that quicksort needs extra space for the recursion stack, and that this space can be linear in the number of items to be sorted. Recursion stack overflow is thus a potential problem. One can prove that the expected stack size is O(lgN) (exercise: prove this), but by slightly modifying the quicksort algorithm we can guarantee that the stack size is O(lgN) in the worst case. The idea is simple: after dividing the set of items into two smaller sets, the small items and the large items, sort the smaller set first; save the larger subproblem for later. (In contrast, the standard version of quicksort solves the subproblem containing the small keys first, even if there are more small keys than large keys.) When solving the larger subproblem, do not make a recursive call; just repeat the splitting process. (The second recursive call is what is called a "tail recursion", which we replace by an iteration.) With this method, each time the recursion stack grows, the subproblem size at least halves. Thus space needed for the recursion stack is O(lgN). Exercise: implement this method.

Because it needs a recursion stack, it is questionable whether quicksort should be viewed as an in-place sorting method: a truly in-place sorting method would use only a constant number of extra storage locations in addition to the array occupied by the items to be sorted. This is true for example of heapsort. As I discussed in precept, the number of comparisons required by heapsort in the worst case can be reduced to $\sim N \lg N$, less than the number required by quicksort. Quicksort is generally favored over heapsort not because it does fewer comparisons but because it does less data movement, the data movement is more local (so the caching performance is better), and the algorithm is more parallelizable.

Here are some optimizations of heap insertion, heap deletion, and heapsort. To do an insertion, we insert the new item into the first empty position, say position k. We then find its correct location, by doing a search on its proper ancestors, which are in positions $k_j = \lfloor k / 2^j \rfloor$ for $j = 1, 2, 3, \ldots$ There are at most $\lg N$ such proper ancestors. (Why?) We could do a binary search among these ancestors to locate the proper position of the new item, as I suggested in precept, but an even better idea is to do an exponential/binary search: we compare the new item to those in positions k_1, k_2, k_4, k_8 until finding a bigger item; then we do binary search within the set of ancestors between the last two ancestors tested (one is smaller, the other is bigger). This takes at most $2 \lg \lg N$ comparisons in the worst case. (Why?) Furthermore, since most of the items are in the bottom levels of the tree, in some average sense the number of comparisons may well be O(1). (Problem: prove this formally.) In any case, having found the correct position of the next item, we sink by one level all its ancestors that are smaller and insert the new item in its correct position. (If we do the sinking by pairwise exchanges, we do about three times as many writes as are needed.) In conclusion, we can do an insertion in at most $2 \lg \lg N$ comparisons in the worst case (reducible to $\lg \lg N$ by doing standard binary search), $\lg N$ writes in the worst case, and perhaps O(1) comparisons and writes on average.

We can save a constant factor in deletions as well. To delete the maximum, we remove the item in position 1 and save it for return. This creates a hole, which we fill with the biggest of the two children of the hole. This creates a hole one level down in the tree, which we fill in the same way. We continue filling the hole until it reaches the bottom level of the tree. If the hole is in the last previously full position, we are done. If not, we move the item in the last full position into the hole, and move it to its correct position by comparing it against its new ancestors as described above. As a further optimization, we can avoid moving any items until we do all the comparisons and know exactly which position in the bottom level becomes the hole and where to put the item in the last full position. We can then move only the items that need to be moved, of which there are at most lgN. The worst-case number of comparisons is ~lgN, reduced from ~2lgN.

A third observation about heaps concerns heapsort. Heapsort consists of two passes: heap construction, in which the original array is turned into a heap, and sorting, in which the items are deleted from the heap in decreasing order and inserted at the end, resulting eventually in the entire array being in increasing order. The sorting pass consists of N deletions, which by the optimized method described above take a total of ~NlgN comparisons and ~NlgN writes, worst-case. The first pass is done bottom-up rather than top down: it consists of sinking each item to its correct level, doing a sink on positions N, N – 1, N – 2,…, 1. Each of these sinks can be optimized in the same way as above. Whether or not this optimization is done, the entire first pass takes O(N) time. We can see this as follows. Consider the last sink, that of the item in position 1. This sink takes $O(\lg N)$ time. Before this sink, the rest of the array consists of two interlaced heaps, one consisting of the even positions starting from 2, the other consisting of the odd positions starting from 3. Each of these heaps contains at most half the items. The total time for the first pass, T(N), satisfies the recurrence $T(N) \le 2T\lfloor (N/2) \rfloor + O(\lg N)$ (why?), which solves to T(N) = O(N). (Why?) Thus most of the time in heapsort is spent in the sorting pass.

In precept I was asked whether the A* search technique used in the 8-puzzle programming assignment is guaranteed to find a shortest path. The answer is yes, provided that the estimate of the distance from the current state to the goal state (in the assignment this is the number of out-of-place blocks or the sum of Manhattan distances of displacements) satisfies a certain condition. A necessary but not sufficient condition is that the estimate be a lower bound on the actual distance. The proper framework in which to discuss this issue is that of shortest path computation and in particular Dijkstra's algorithm, which we shall study later in the semester. See lecture 15. I plan to address A* in the precept notes after that lecture.