In-place Heap Sort

In-place

Implies that only additional O(1) extra data structure items are required to solve the problem. For example in the sorting algorithms we have presented we use three Vectors, inS, PQ, and outS. Do we need them all?

Advantages of in-place

- minimal space
- opportunity to for fine tuning optimization, i.e. combining operations and few methods call
- improved caching

Explain caching

- Memory hierarchy
- Speed of memory
- caching near by

In-place heap-sort

Instead of building a separate data structure for the heap, we could use the same array for the inS and the heap while building the heap. When inS is empty and we shrink the heap to build the output sequence, outS, the same array would contain both heap and outS. As example let consider that the heap is always on the left side (lower index) of the array while inS and outS are kept to right (higher index). We need to keep track of the location of the division between the heap and sequence. This is easy—it can be size of the heap.

To get a sequence in ascending order, we will use a Maximum heap instead of a Minimum heap: removeMin should removeMax. In other words we will build a heap with the maximum integer on top (at the root).

Example: Note | indicates the division between heap and in/outS

input: |4 7 2 1 3
Remove from sequence and place in heap:
heap | inS
4|7 2 1 3
7 4|2 1 3
7 4 2|1 3
7 4 2 1|3
7 4 2 1 3|

Remove min from heap and place in output sequence:
heap | outS
4 2 3 1|7
3 2 1|4 7
2 1|3 4 7
1|2 3 4 7
Output: 1 2 3 4 7|

Note some addition to detail is required to keep the sorting O($n\ lg\ n$). We must make sure that the method of inS and outS are O(1) and not O(n). If the sequence methods are O(n) then the sorting complexity is O(n^2). In the above example:

- "remove element from inS" should use `inS.removeFirst()`
- "append element to outS" should use `outS.insertFirst(integer)`

The number of steps are:
10 steps < 5 $lg5$ <15

Bottom-Up Heap Construction

We can make additional improvements? We can beat O($n\ lg\ n$) during the heap construction, if we know the size ahead of time. Assume size is $n = 2^h - 1$. n is an odd number. Height of the heap is $h = lg(n + 1)$.

Algorithm

Basic idea is to build the heap from the bottom up Although the algorithm is naturally recursive, I will describe it iteratively.
1. Make (n+1)/2 single item heap trees, Note (n-1)/2 items still in reserve.
2. Merge pairs of heaps single item heaps by adding an item from the reserve. Note (n+1)/4 heaps now.
3. Merge pairs of three item heaps by adding item from the reserve. Note (n+1)/8 heaps
:
:
Do this $h = lg(n + 1)$ times, the height of the tree.

Illustrate iteratively 15 keys page 317

Recursive Algorithm for **BottomUpHeap(S)**
input: Sequence S with $n = 2^h - 1$ keys
Output: Heap T storing the keys of S.

if S empty **then return** empty heap
Remove the first key, k, from S.
// Note k will become root of new Tree
Split S into S_1 and S_2 each of size (n-1)/2
// recursive calls
T_1 = BottomUpHeap(S_1)
T_2 = BottomUpHeap(S_2)
Create binary tree T with
    root storing k,
    T_1 left subtree and T_2 right subtree
Perform Bubble down from root of T
return T

Cost is O(n) is not immediately apparent form the algorithm.

The proof can be demonstrated visually. The worst case cost for each recursive call of the algorithm is height of the current tree, T, because of the bubbling. We will determine the total cost by delineating the cost of all the bubble down operations. We associate the path to the successor in-order node (first down right then down left)

with the root node. This a possible bubbling path. The total time, form all the recursive calls, is the total number of associated paths, which is O(n).

Although the heap is constructed in O(n), removing min from the heap and reconstructing the order sequence still takes O(n lg n).

Adaptable Priority Queues

Adaptable priority queues support access for quick retrieval and modification of all entries in the priority queue.

Real life Examples

Interface

```
interface AdaptablePriorityQueue extends PriorityQueue {
   public Entry remove(Entry e);

   /* returns old key */
   public Object replace(Entry e, Object key);

   /* returns the old value */
   public Object replaceValue(Entry e, Object value);
}
```

Implementation

We can extend entries to a “location aware” entry which contains a private field, Position: the position in the priority queue. Naturally the priority queue will have to update Position when ever changes are made to the priority queue.

Cost

Locating the position in the priority queue is O(1), but the updates will cost the same as updates in the underlying data structure for the priority queue.