Heaps and Heapsort

October 21, 2013

A heap is a data structure that stores a set and allows fast access to the item with highest priority. It is the basis of a fast implementation of selection sort. On the average this algorithm is a little slower than quicksort but it is not sensitive to the input ordering or to random bits and runs about as fast in the worst case as on the average.

Priority queues. A data structure implements the priority queue abstract data type if it supports at least the following operations:

```c
void INSERT(item),
item FINDMIN(),
void DELETEMIN().
```

The operations are applied to a set of items with priorities. The priorities are totally ordered so any two can be compared. To avoid any confusion, we will usually refer to the priorities as ranks (and sometimes as keys). We will always use integers as priorities and follow the convention that smaller ranks represent higher priorities. In many applications, FINDMIN and DELETEMIN are combined:

```c
void EXTRACTMIN()
    r = FINDMIN(); DELETEMIN(); return r.
```

Function EXTRACTMIN removes and returns the item with smallest rank.

Heap. A heap is a particularly compact priority queue. We can think of it as a binary tree with items stored in the internal nodes, as in Figure 1. Each level is full except possibly the last, which is filled from left to right until we run out of items. The items are stored in heap-order: every node \( \mu \) has a rank larger than or equal to the rank of its parent. Symmetrically, \( \mu \) has a rank less than or equal to the ranks of both its children. As a consequence, the root contains the item with smallest rank.

We store the nodes of the tree in a linear array, level by level from top to bottom and each level from left to right, as shown in Figure 2. The embedding saves explicit pointers otherwise needed to establish parent-child relations. Specifically, we can find the children and parent of a node by index computation: the left child of \( A[i] \) is \( A[2i] \), the right child is \( A[2i+1] \), and the parent is \( A[(i/2)] \). The item with minimum rank is stored in the first element:

---

These notes are based on the lecture notes on “Design and Analysis of Algorithms”, Fall 2005 by Herbert Edelsbrunner.
item $\text{FINDMIN}()$

\begin{verbatim}
   assert n \geq 1; return A[1].
\end{verbatim}

Here and below we assume that array $A$ and its size $n$ are global variables.

Since the index along a path at least doubles each step, paths can have length at most $\log_2 n$.

**Inserting.** We first study the problem of repairing the heap-order if it is violated at the last position of the heap, as shown in Figure 3.

Let $n$ be the length of the array. We repair the heap-order by a sequence of swaps along a single path. Each swap is between an item and its parent. The item moves up the heap until it reaches a position where its rank is at least as large as that of its parent.
void SIFT-UP(int i)
  if \( i \geq 2 \) then \( k = \lfloor i/2 \rfloor \);
  if \( A[i] < A[k] \) then SWAP(i, k);
    SIFT-UP(k)
  endif
endif.

An item is added by first expanding the heap by one element, placing the new item in the position that just opened up, and repairing the heap-order.

void INSERT(item x)
  \( n++ \);  \( A[n] = x \); SIFT-UP(n).

Since a path has at most \( \log_2 n \) edges, the time to repair the heap-order takes time at most \( O(\log n) \).

**Deleting the minimum.** Now consider repairing the heap-order if it is violated at the root, as shown in Figure 4. We repair the heap-order by a sequence of swaps along a single path, only now going downwards. Each swap is between an item and the smaller of its children:

\[
\text{void SIFT-DN(int i)}
\]
\[
\text{if } 2i \leq n \text{ then}
\]
\[
\text{k = } 2i; \quad \text{ // left child}
\]
\[
\text{if } k + 1 \leq n \text{ and } A[k + 1] < A[k] \text{ then } k = k + 1; \quad \text{ // right child}
\]
\[
\text{if } A[k] < A[i] \text{ then SWAP(i, k);} \quad \text{SIFT-DN(k)}
\]
\[
\text{endif}
\]
\[
\text{endif.}
\]

Since a path has at most \( \log_2 n \) edges, the time to repair the heap-order takes time at most \( O(\log n) \). To delete the minimum we overwrite the root with the last element, shorten the heap, and repair the heap-order:

\[
\text{void DELETE-MIN()}
\]
\[
\text{A[1] = } A[n]; \quad \text{n}--; \quad \text{SIFT-DN(1)}.
\]

To summarize, a heap supports FIND-MIN in constant time and INSERT and DELETE-MIN in time \( O(\log n) \) each.
**Sorting.** Priority queues can be used for sorting. The first step throws all items into the priority queue (heap construction), and the second step takes them out in order. Let us look at these steps more closely.

- **Step 1: heap construction.** Assuming the items are already stored in the array, the first step can be done by repeated heap repair:

  \[
  \text{for } i = 1 \text{ to } n \text{ do } \text{SIFT-UP}(i) \text{ endfor.}
  \]

  In the worst case, the \( i \)-th item moves up all the way to the root. The number of exchanges is therefore at most \( \sum_{i=1}^{n} \log_2 i \leq n \log_2 n \). The upper bound is asymptotically tight because half the terms in the sum are at least \( \log_2 \frac{n}{2} = \log_2 n - 1 \). It is also possible to construct the initial heap in time \( O(n) \) by building it from bottom to top:

  \[
  \text{for } i = n \text{ downto } 1 \text{ do } \text{SIFT-DN}(i) \text{ endfor.}
  \]

  At each step of the for-loop, we consider the subtree with root \( A[i] \). At this moment the items in the left and right subtrees rooted at \( A[2i] \) and \( A[2i + 1] \) are already heaps. We can therefore use one call to function SIFT-DN to make the subtree with root \( A[i] \) a heap. We will prove shortly that this bottom-up construction of the heap takes time only \( O(n) \).

- **Step 2: Extracting items from the heap.** This step can be implemented via \( n - 1 \) calls to EXTRACTMIN. Extracted items are moved to the freed space in the same array:

  \[
  \text{while } n > 1 \text{ do } \text{SWAP}(n, 1); n--; \text{ SIFT-DN}(1) \text{ endwhile.}
  \]

  Figure 5 shows the array after each iteration of the while-loop. Note how the heap gets smaller by one element each step. A single sift-down operation takes time \( O(\log n) \), and in total HEAPSORT takes time \( O(n \log n) \). In addition to the input array, HEAPSORT uses a constant number of variables and memory for the recursion stack used by SIFT-DN. We can save the memory for the stack by writing function SIFT-DN as an iteration. The sort can be changed to non-decreasing order by reversing the order of items in the heap.

To summarise, the overall algorithm is as follows:

```c
void HEAPSORT(int n)
    for i = n downto 1 do SIFT-DN(i, n) endfor;
    while n > 1 do
        SWAP(n, 1); n--; SIFT-DN(1);
    endwhile.
```
Figure 5: Each step moves the last heap element to the root and thus shrinks the heap. The circles mark the items involved in the sift-down operation.

**Analysis of heap construction.** Let us consider again the two techniques for constructing a heap, namely top-down:

```plaintext
for i = 1 to n do SIFT-UP(i) endfor.
```

and bottom-up:

```plaintext
for i = n downto 1 do SIFT-DN(i) endfor.
```

We claimed that the former approach takes $O(n \log n)$ time, while the latter takes only $O(n)$. Intuitively, this can be explained as follows. Suppose for simplicity that the last level is completely filled (i.e. $n = 2^k - 1$). Also assume that we have the worst case; in the top-down approach it means that SIFT-UP($i$) moves item $i$ to the first level, while in the bottom-up approach SIFT-DOWN($i$) moves $i$ to the last level. It can be seen that SIFT-DOWN would move most items by a distance close to $\log_2 n$, while SIFT-DOWN($i$) would move them by a very short (in fact, constant) distance. For example, consider the last level that stores roughly $n/2$ items: SIFT-DOWN would move them by $\log_2(n + 1) - 1$ steps, while SIFT-UP would not move them at all. $n/4$ items at the penultimate level would be moved by SIFT-UP only by 1 step, and so on.

To confirm this intuition, let us now prove that the bottom-up approach indeed takes $O(n)$ time. Assuming the worst case, in which every node sifts down all the way to the last level, we draw the swaps as edges in a tree; see Figure 6. To avoid drawing any edge twice, we always assume that the first swap is to the right and all subsequent swaps are to the left until we arrive at the last level. (This assumption does not affect the number
of swaps.) The paths cover each edge once, except for the edges on the leftmost path, which are not covered at all. The number of edges in the tree is \( n - 1 \), which implies that the total number of swaps is less than \( n \). Equivalently, the amortized number of swaps per item is less than 1. There is a striking difference in time-complexity to sorting, which takes an amortized number of about \( \log_2 n \) comparisons per item. The difference between 1 and \( \log_2 n \) may be interpreted as a measure of how far from sorted a heap-ordered array still is.

**Alternative implementations of priority queues.** There exist many other efficient implementations of priority queues. Similar to the binary heap, most of them maintain a tree that satisfies the “heap-order” property: the rank of each item is smaller or equal than the ranks of its children. But the structure of the tree can be much more flexible compared to binary heaps (specific rules vary depending on the implementation). Also, some implementations such as Fibonacci heaps maintain a collection of trees (i.e. a forest), each of which satisfies the heap-order property.

Two noteworthy examples are **Fibonacci heaps** and **pairing heaps**. Fibonacci heaps have the best known asymptotic complexity for a range of operations; in particular, **INSERT** and **DECREASEKEY** operations take constant (amortized) time, compared to \( O(\log n) \) for binary heaps.

Pairing heaps have a slightly worse complexity for the **DECREASEKEY** operation, but showed excellent performance in practice (see the wikipedia page on pairing heaps for references to experimental studies). Pairing heaps also require less memory (by a constant factor) compared to Fibonacci heaps.

The table below compares the complexity of different operations for these implementations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary heap</th>
<th>Fibonacci heap</th>
<th>Pairing heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial construction</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Insert</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>( O(\log n) )</td>
<td>( O(\log n)^* )</td>
<td>( O(\log n)^* )</td>
</tr>
<tr>
<td>DecreaseKey</td>
<td>( O(\log n) )</td>
<td>( O(1)^* )</td>
<td>unknown**</td>
</tr>
</tbody>
</table>

* Amortized time

**upper known bound: \( 2^{O(\sqrt{\log \log n})} \) - better than \( O(\log n) \) = \( 2^{O(\log \log n)} \)

lower known bound: \( \Omega(\log \log n) \) - worse than \( O(1) \)

---

1 Operation **DECREASEKEY(item, newRank)** decreases the rank of an item which is already in the priority queue. This operation is needed by several algorithms, e.g. the **Dijkstra algorithm** for computing shortest paths in a weighted graph.