SIAM J. COMPUT.
Vol. 16, No. 1, February 1987

009

A NOTE ON BOTTOM-UP SKEW HEAPS*

DOUGLAS W. JONES†

Abstract. In testing Sleator and Tarjan's skew heap implementations of priority queues, a problem with the bottom-up variant was found. The original definition of skew heaps includes the assumption that the keys of items in the heap are disjoint. When this is not true, the *delete min* operation on bottom-up skew heaps will occasionally discard items from the heap. Modified versions of the *delete min* algorithm are presented which do not require this assumption.

Key words. priority queue, heap, skew heap

AMS(MOS) subject classifications. 68P10, 68Q20, 05-04

1. Introduction. While performing the empirical tests reported in [1], a problem was encountered with the bottom-up variant of Sleator and Tarjan's skew heaps [2]. The following problem was observed: some of the items inserted in a bottom-up skew heap were never found by the *delete min* operation; that is, items were lost from the heap. This problem can be traced to the definition of a heap (or priority queue) used in § 2 of [2]: "a set of items selected from a totally ordered universe." This implies that no two items in the heap have the same key! In practical applications of heaps, for example, in finding shortest paths in a graph, or in discrete event simulation, the keys of items are occasionally equal.

The notation used in the original presentations of skew heaps identified each node by its key; in order to allow discussion of nodes with equal keys, it is necessary to distinguish between a node's identity and its key. Thus, in the following discussion, identifiers such as x and y will be used to refer to nodes, while their keys will be referenced as $key(x)$ and $key(y)$. The original code for the operations on skew heaps can be trivially converted to allow equal keys by preserving the old notation except when comparing the relative order of two nodes. Thus, comparisons for equality, such as $x = y$, remain unchanged, while comparisons such as $x > y$ are changed to $key(x) > key(y)$.

On inspection of the *insert* and *delete min* operations given in § 3 of [2], the problem was found to be in the latter operation. The original version of the bottom-up *delete min* operation is shown here with the notational changes discussed above (labels are referenced in the text; typographical errors in the original are corrected on lines 3 and 4):

```
function delete min(var h);
    var x, y1, y2, h3;
    if h = null → return null fi;
1:  y1, y2 := up(h), down(h);
2:  if key(y1) < key(y2) → y1 ↔ y2 fi;
3:  if y1 = h → h := null; return y1 fi;
    [initialize h3 to hold y1]
    h3 := y1; y1 := up(y1); up(h3) := h3;
    do true →
4:      if key(y1) < key(y2) → y1 ↔ y2 fi;
5:      if y1 = h → h := h3; return y1 fi;
        [remove x = y1 from its path]
```

* Received by the editors March 13, 1986; accepted for publication June 16, 1986.
† Department of Computer Science, University of Iowa, Iowa City, Iowa 52242.

```
        x := y; y1 := up(y1);
        [add x to the top of h3 and swap its children]
        up(x) := down(x); down(x) := up(h3); h3 := up(h3) := x
      od
end delete min;
```

The loss of data when equal keys are allowed happens on lines 3 and 5. In both cases, the variables y_1 and y_2 refer to the current nodes on the major and minor paths (the right branches of the two subtrees of the structure). In each case, the previous line (2 or 4) ensures that, in terms of the ordering of the key values, y_1 is farther from the root than y_2; thus, if y_1 refers to the root, y_2 must also refer to the root so the merge must be finished.

If items in the heap have equal keys, the above correctness argument does not hold. If y_1 is the same as h, $key(y_2)$ will equal $key(h)$, but that does not imply that y_2 refers to the same node as h. It is possible that y_2 refers to a descendant of h, so long as the keys of all nodes between y_2 and h are equal. In this case, y_2 and all nodes between it and h will be lost from the heap!

2. A solution. The simplest solution to this problem is to replace the comparisons on lines 2 and 4 in the original code with more complex comparisons which treat the root as having a lower key than any other item. The following code fragment will do this:

```
2:  4:  if (y1 = h) or (key(y1) < key(y2)) → y1 ↔ y2 fi;
```

Although this solution introduces a relatively small change in the code for *delete min*, it introduces an extra comparison into each iteration. This can be avoided if there is a key value, *min*, which is less than any key which will be encountered in practice. In this case, the key of the minimum item in the heap can be set to this value before starting the *delete min* operation, and reset to its original value when the work is done. This requires three assignments per visit to *delete min*, and it requires an additional local variable k to save minimum key. These are used by the following modifications to lines 1, 3 and 5 in the original code:

```
1:  y1, y2 := up(h), down(h); k := key(h); key(h) := min;
3:  if y1 = h → h := null; key(y1) := k; return y1 fi;
5:  if y1 = y → h := h3; key(y1) := k; return y1 fi;
```

3. A faster solution. Another solution to this problem is to replace lines 3 and 5 in the original code with a search for the top of the path headed by y_2. Thus, instead of adding code to avoid data loss in the first place, code is added to recover the lost data after the body of *delete min* has finished. The following code fragments do this:

```
3:  if y1 = h →
        if y2 = h → h := null; return y1 fi;
        x := y2;
        do up(y2) ≠ h → y2 := up(y2) od;
        up(y2) := x; h := y2; return y1;

    fi;
5:  if y1 = h →
```

```
    if y2 = h → h := h3; return y1 fi;
    x := y2;
    do up(y2) ≠ h → y2 := up(y2) od;
    up(y2) := up(h3); up(h3) := x; h := y2; return y1;
  fi;
```

This code simply patches the remainder of the path pointed to by y_2 into the structure, without attention to preserving the amortized bounds. The code used in the empirical measurements reported in [1] was a Pascal transliteration of this. In [1], keys were represented by randomly selected real numbers so equal keys were extremely unlikely. If equal keys are probable, a more complex solution will be required. The fundamental problem with the above code is that it will preserve long right paths in the tree because the search loops do not exchange the children of each node visited. The following code cures this problem:

```
3: if y1 = h →
     if y2 = h → h := null; return y1 fi;
     [initialize h3 to hold y2]
     h3 := y2; y2 := up(y2); up(h3) := h3;
     do true →
       if y2 = h → h := h3; return y2 fi;
       [remove x = y2 from its path]
       x := y2; y2 := up(y2);
       [add x to the top of h3 and swap its children]
       up(x) := down(x); down(x) := up(h3); h3 := up(h3) := x;
     od;
   fi;

5: if y1 = h →
     do true →
       if y2 = h → h := h3; return y2 fi;
       [remove x = y2 from its path]
       x := y2; y2 := up(y2);
       [add x to the top of h3 and swap its children]
       up(x) := down(x); down(x) := up(h3); h3 := up(h3) := x;
     od
   fi;
```

These changes result in an annoying lengthening of what was originally a rather elegant algorithm, but they result in a trivial increase in the algorithm's run-time. If keys are disjoint, a single additional comparison is introduced per *delete min* operation. If keys are not disjoint, an additional comparison is added in determining that the new code must be executed, but the traversal of the final path to the root in this code is actually faster than it would have been had the keys been disjoint!

REFERENCES

[1] D. W. JONES, *An empirical comparison of priority-queue and event-set implementations*, Comm. ACM, 29 (1986), pp. 300-311.

[2] D. D. SLEATOR AND R. E. TARJAN, *Self adjusting heaps*, this Journal, 15 (1986), pp. 52-69.