CHAPTER 3

Heaps

3.1. Heaps and heap-ordered trees. Our use of trees as data structures in Chapter 2 was especially simple, since we needed only parent pointers to represent the trees, and the position of items within each tree was completely unspecified, depending only on the sequence of set operations. However, in many uses of trees, the items in the tree nodes have associated real-valued keys, and the position of items within the tree depends on key order. In this and the next chapter we shall study the two major examples of this phenomenon.

A *heap* is an abstract data structure consisting of a collection of items, each with an associated real-valued *key*. Two operations are possible on a heap:

insert (i, h): Insert item i into heap h, not previously containing i.
deletemin (h): Delete and return an item of minimum key from heap h; if h is empty return **null.**

The following operation creates a new heap:

makeheap (s): Construct and return a new heap whose items are the elements in set s.

In addition to these three heap operations, we sometimes allow several others:

findmin (h): Return but do not delete an item of minimum key from heap h; if h is empty return **null.**
delete (i, h): Delete item i from heap h.
meld (h_1, h_2): Return the heap formed by combining disjoint heaps h_1 and h_2. This operation destroys h_1 and h_2.

Williams [10], the originator of the term "heap," meant by it the specific concrete data structure that we call a d-heap in §2. Knuth [8] used the term "priority queue" to denote a heap. Aho, Hopcroft and Ullman [1] used "priority queue" for an unmeldable heap and "mergeable heap" for a meldable heap.

To implement a heap we can use a *heap-ordered tree*. (See Fig. 3.1.) Each tree node contains one item, with the items arranged in heap order: if x and $p(x)$ are a node and its parent, respectively, then the key of the item in $p(x)$ is no greater than the key of the item in x. Thus the root of the tree contains an item of minimum key, and we can carry out findmin in $O(1)$ time by accessing the root. The time bounds of the other operations depend on the tree structure.

We shall study two heap implementations that use this idea. The *d-heaps* described in §3.2 are appropriate when only one or a few unmeldable heaps are needed. The *leftist heaps* of §3.3 are appropriate when several meldable heaps are needed.

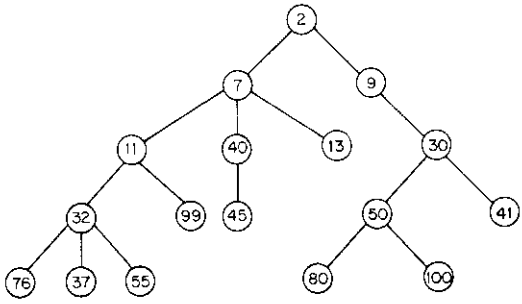

FIG. 3.1. *A heap-ordered tree. Numbers in nodes are keys. (To simplify this and subsequent figures, we treat the keys as if they themselves are the items.)*

3.2. *d*-heaps. Suppose the tree representing a heap is exogenous; that is, the heap items and the tree nodes are distinct. This allows us to restore heap order after an update by moving the items among the nodes. In particular, we can insert a new item i as follows. To make room for i, we add a new vacant node x to the tree; the parent of x can be arbitrary, but x must have no children. Storing i in x will violate heap order if the parent $p(x)$ of x contains an item whose key exceeds that of i, but we can remedy this by carrying out a *sift-up:* While $p(x)$ is defined and contains an item whose key exceeds $key(i)$, we store in x the item previously in $p(x)$, replace the

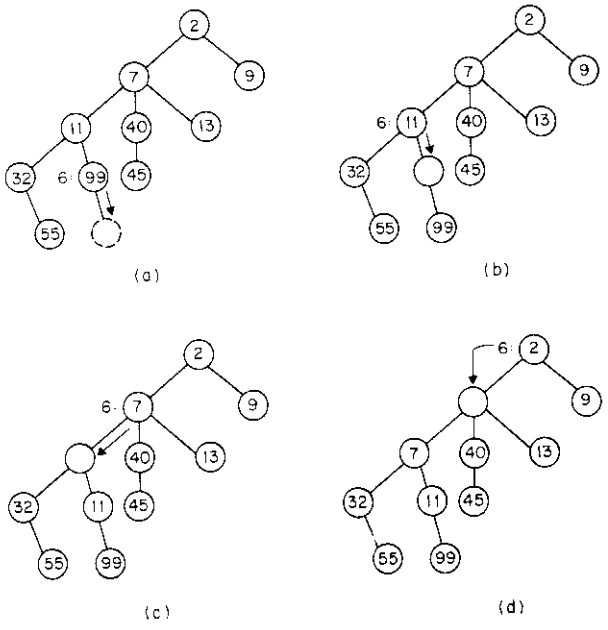

FIG. 3.2. *Insertion of key 6 using sift-up.* (a) *Creation of a new leaf. Since* $6 < 99$, 99 *moves into the new node.* (b) *Since* $6 < 11$, 11 *moves into the vacant node.* (c) *Since* $6 < 7$, 7 *moves into the vacant node.* (d) *Since* $6 \geq 2$, 6 *moves into the vacant node and the sift-up stops.*

vacant node x by $p(x)$, and repeat. When the sifting stops, we store i in x. (See Fig. 3.2.)

Deletion is a little more complicated than insertion. To delete an item i, we begin by finding a node y with no children. We remove the item, say j, from y and delete y from the tree. If $i = j$ we are done. Otherwise we remove i from the node, say x, containing it, and attempt to replace it by j. If $key\ (j) \leq key\ (i)$, we reinsert j by carrying out a sift-up starting from x. If $key\ (j) > key\ (i)$, we reinsert j by carrying out a *sift-down* starting from x: While $key\ (j)$ exceeds the key of some child of x, we choose a child c of x containing an item of minimum key, store in x the item in c, replace x by c, and repeat. (See Fig. 3.3.) When the sifting stops, we store j in x.

When deleting an item, the easiest way to obtain a node y with no children is to choose the most-recently added node not yet deleted. If we always use this rule, the tree nodes behave like a stack (last-in, first-out) with respect to addition and deletion.

The running times of the heap operations depend upon the structure of the tree, which we must still define. The time for a sift-up is proportional to the depth of the node at which the sift-up starts. The time for a sift-down is proportional to the total number of children of the nodes made vacant during the sift-down. A type of tree that has small depth, few children of nodes on a path and a uniform structure is the

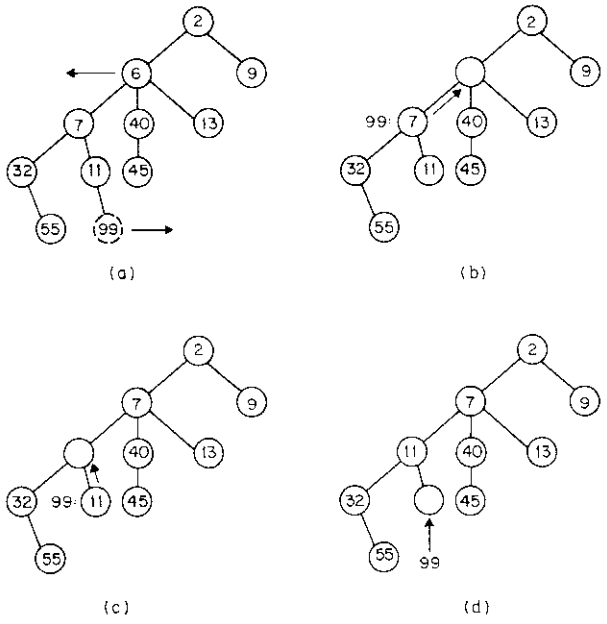

FIG. 3.3. *Deletion of key* 6 *using sift-down. The last-created node contains key* 99. (a) *Destruction of the last leaf. Key* 99 *must be reinserted.* (b) *The smallest key among the children of the vacant node is* 7. *Since* $99 > 7$, 7 *moves into the vacant node.* (c) *The smallest key among the children of the vacant node is* 11. *Since* $99 > 11$, 11 *moves into the vacant node.* (d) *Since the vacant node has no children,* 99 *moves in and the sift-down stops.*

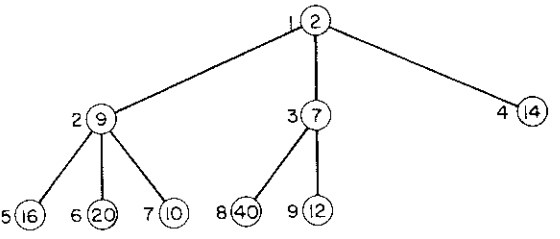

FIG. 3.4. *A 3-heap with nodes numbered in breadth-first order. Next node to be added is* 10, *a child of* 3.

complete d-ary tree: each node has at most d children and nodes are added in breadth-first order. Thus we define a *d-heap* to be a complete d-ary tree containing one item per node arranged in heap order. (See Fig. 3.4.)

The d-heap operations have running times of $O(1)$ for findmin, $O(\log_d n)$ for insert, and $O(d \log_d n)$ for delete and deletemin, where n is the number of items in the heap, since a complete d-ary tree has depth $\log_d n + O(1)$. The parameter d allows us to choose the data structure to fit the relative frequencies of the operations; as the proportion of deletions decreases, we can increase d, saving time on insertions. We shall use this capability in Chapter 7 to speed up a shortest-path algorithm.

The structure of a d-heap is so regular that we need no explicit pointers to represent it. If we number the nodes of a complete d-ary tree from one to n in breadth-first order and identify nodes by number, then the parent of node x is $\lceil (x - 1)/d \rceil$ and the children of node x are the integers in the interval $[d(x - 1) + 2 \,..\, \min\{dx + 1, n\}]$. (See Fig. 3.4.) Thus we can represent each node by an integer and the entire heap by a map h from $\{1 \,..\, n\}$ onto the items. The following programs implement all the d-heap operations except makeheap, which we shall consider later.

```
item function findmin (heap h)
   return if h = { } → null | h ≠ { } → h(1) fi
end findmin;

procedure insert (item i, modifies heap h);
1. siftup (i, |h| + 1, h)
end insert;

procedure delete (item i, modifies heap h);
   item j;
   j := h(|h|);
2. h(|h|) = null;
   if i ≠ j and key (j) ≦ key (i) → siftup (j, h⁻¹(i), h)
     | i ≠ j and key (j) > key (i) → siftdown (j, h⁻¹(i), h)
   fi
end delete;
```

```
item function deletemin (modifies heap h);
  if h = { } → return null
    | h ≠ { } →
    item i;
    i := h(1);
    delete (h(1), h);
    return i
  fi
end deletemin;

procedure siftup (item i, integer x, modifies heap h);
  integer p;
  p := ⌈(x − 1)/d⌉ ;
  do p ≠ 0 and key (h(p)) > key (i) →
    h(x), x, p := h(p), p, ⌈(p − 1)/d⌉ od;
3. h(x) := i
end siftup;

procedure siftdown (item i, integer x, modifies heap h);
  integer c;
  c := minchild (x, h);
  do c ≠ 0 and key (h(c)) < key (i) →
                    h(x), x, c := h(c), c, minchild (c, h) od;
    h(x) := i
end siftdown;

integer function minchild (integer x, heap h);
  return if d·(x − 1) + 2 > |h| → 0
4. |d·(x − 1) + 2 ≤ |h| → min {d · (x − 1) + 2 . . min {d · x + 1, |h|}} by key ∘ h
  fi
end minchild;
```

Notes. See §1.3 for a discussion of our notation for maps. The complicated expression in line 4 selects from among the set of integers from $d(x-1)+2$ to $\min\{dx+1, |h|\}$, i.e. the children of x, an integer c such that $key(h(c))$ is minimum, i.e. a node containing an item of minimum key. □

The best way to implement the map representing a d-heap is as an array of positions from one to the maximum possible heap size. We must also store an integer giving the size of the heap and, if arbitrary deletion is necessary, every item i must have a heap index giving its position $h^{-1}(i)$ in the heap. If arbitrary deletion is unnecessary we need not store heap indices, provided we customize deletemin to call siftdown directly. (A call to delete from deletemin will never result in a sift-up.)

We have written siftup and siftdown so that they can be used as heap operations to restore heap order after changing the key of an item. If the key of item i in heap h decreases, the call siftup $(i, h^{-1}(i), h)$ will restore heap order; if the key increases, siftdown $(i, h^{-1}(i), h)$ will restore heap order. We shall use this capability in Chapter 7.

The last heap operation we must implement is makeheap. We can initialize a d-heap by performing n insertions, but then the time bound is $O(n \log_d n)$. A better method is to build a complete d-ary tree of the items in arbitrary order and then execute siftdown $(h(x), x, h)$ for $x = n, n-1, \cdots, 1$, where h is the map defining the item positions. The running time of this method is bounded by a constant times the sum

$$\sum_{i=0}^{\infty} \frac{n(i+1)}{d^i} = O(n),$$

since there are at most n/d^i nodes of height i in a complete d-ary tree of n nodes.

The following program implements makeheap:

```
heap function makeheap (set s);
    map h;
    h := { };
    for i ∈ s → h(|h| + 1) := i rof;
    for x := |s|, |s| − 1 .. 1 → siftdown (h(x), x, h) rof;
    return h
end makeheap;
```

We close this section with some history and a remark. Williams, building on the earlier TREESORT algorithm of Floyd, invented 2-heaps and discovered how to store them as arrays [6], [10]. Johnson [7] suggested the generalization to $d > 2$. An analysis of the constant factor involved in the timing of the heap operations suggests that the choice $d = 3$ or 4 dominates the choice $d = 2$ in all circumstances, although this requires experimental confirmation.

3.3. Leftist heaps. The d-heaps of §3.2 are not easy to meld. In this section we shall study leftist heaps, which provide an alternative to d-heaps when melding is necessary. Knuth [8] coined the term "leftist" for his version of a data structure invented by Crane [5].

If x is a node in a full binary tree, we define the *rank* of x to be the minimum length of a path from x to an external node. That is, *rank* $(x) = 0$ if x is an external node, *rank* $(x) = 1 + \min \{rank\ (left\ (x)), rank\ (right\ (x))\}$ if x is an internal node. A full binary tree is *leftist* if *rank* $(left\ (x)) \geq rank\ (right\ (x))$ for every internal

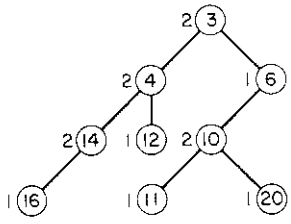

FIG. 3.5. *A leftist heap. External nodes are omitted. Numbers near nodes are ranks.*

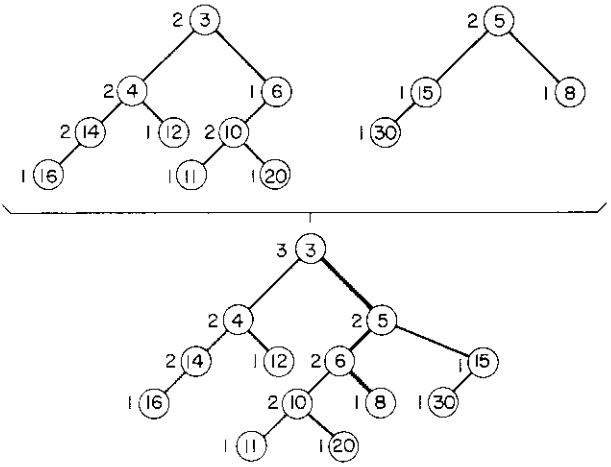

FIG. 3.6. *Merging two leftist heaps. Merged path is marked by heavy lines. To maintain the leftist property, children of* 5 *are swapped.*

node x. In a leftist tree the right path is a shortest path from the root to an external node. It is easy to prove by induction that this path has length at most $lg\ n$.

A *leftist heap* is a leftist tree containing one item per internal node, with items arranged in heap order. (See Fig. 3.5.) We shall treat leftist heaps as being endogenous; that is, the items themselves are the tree nodes. Since the external nodes contain no information, we shall assume that every external node is **null.** To save tests in our programs we assume that *rank* (**null**) is initialized to zero. To store a leftist heap, we need two pointers and one integer for each internal node, giving its left and right children and its rank. To identify the heap, we use a pointer to its root.

The fundamental operation on leftist heaps is melding. To meld two heaps, we merge their right paths, arranging nodes in nondecreasing order by key. Then we recompute the ranks of the nodes on the merged path and make the tree leftist by swapping left and right children as necessary. (See Fig. 3.6.) The entire meld takes $O(\log n)$ time, where n is the number of nodes in the two heaps.

To insert an item into a leftist heap, we make the item into a one-node heap and meld it with the existing heap. To delete a minimum item, we remove the root and meld its left and right subtrees. Both these operations take $O(\log n)$ time. The following programs implement findmin, meld, insert, and deletemin on leftist heaps:

```
item function findmin (heap h);
   return h
end findmin;

heap function meld (heap h1, h2);
   return if h1 = null → h2 | h2 = null → h1
          | h1 ≠ null and h2 ≠ null → mesh (h1, h2) fi
end meld;
```

```
heap function mesh (heap h1, h2);
    if key (h1) > key (h2) → h1 ↔ h2 fi;
    right (h1) := if right (h1) = null → h2
                | right (h1) ≠ null → mesh (right (h1), h2) fi;
    if rank (left (h1)) < rank (right (h1)) → left (h1) ↔ right (h1) fi;
    rank (h1) := rank (right (h1)) + 1;
    return h1
end mesh;
```

Note. The meld program is merely a driver to deal efficiently with the special case of a **null** input; the mesh program performs the actual melding. □

```
procedure insert (item i, modifies heap h);
    left (i), right (i), rank (i) := null, null; 1;
    h := meld (i, h)
end insert;

item function deletemin (modifies heap h);
    item i;
    i := h;
    h := meld (left (h), right (h));
    return i;
end deletemin;
```

Before studying heap initialization and arbitrary deletion, let us consider two other useful heap operations:

listmin (x, h): Return a list containing all items in heap h with key not exceeding real number x.

heapify (q): Return a heap formed by melding all the heaps on the list q. This operation assumes that the heaps on q are disjoint and destroys both q and the heaps on it.

The heap order of leftist heaps allows us to carry out listmin in time proportional to the number of elements listed: we perform a preorder traversal starting from the root of the heap, listing every encountered item with key not exceeding x and immediately retreating from each item with key exceeding x. The same method works on d-heaps but takes $O(dk)$ rather than $O(k)$ time, where k is the size of the output list. The following program implements listmin on leftist heaps:

```
list function listmin (real x, heap h);
    return if h = null or key (h) > x → [ ]
            | h ≠ null and key (h) ≤ x → [h] & listmin (left (h))
                                            & listmin (right (h))
        fi
end listmin;
```

To carry out heapify, we treat the input list as a queue. We repeat the following step until only one heap remains, which we return: Remove the first two heaps from

the queue, meld them, and add the new heap to the end of the queue. The following program implements this method:

```
heap function heapify (list q);
  do |q| ≥ 2 → q := q[3 . .] & meld (q(1), q(2)) od;
  return if q = [ ] → null | q ≠ [ ] → q(1) fi
end heapify;
```

In order to analyze the running time of heapify, let us consider one pass through the queue. Let k be the number of heaps on the queue and n the total number of items they contain. After $\lceil k/2 \rceil$ melds, every heap has been melded with another, leaving at most $\lfloor k/2 \rfloor$ heaps. The total time for these melds is

$$O\left(\sum_{i=1}^{\lfloor k/2 \rfloor} \max\{1, \log n_i\}\right),$$

where n_i is the number of items in the ith heap remaining after the pass. We have $0 \le n_i \le n$ and $\sum_{i=1}^{\lfloor k/2 \rfloor} n_i = n$. These constraints imply that the time for one pass through the queue is $O(k \max\{1, \log (n/k)\})$. The time for the entire heapify is

$$O\left(\sum_{i=0}^{\lfloor \lg k \rfloor} \frac{k}{2^i} \max\left\{1, \log \frac{n2^i}{k}\right\}\right) = O\left(k \max\left\{1, \log \frac{n}{k}\right\}\right),$$

where k is the number of original heaps and n is the total number of items they contain.

We can make a leftist heap of n items in $O(n \log n)$ time by repeated insertion, but heapify gives a better method: We make each item into a one-item heap and apply heapify to a list of these heaps. In this case $k = n$ and the running time is $O(n)$. The following program implements makeheap using this method:

```
heap function makeheap (set s);
  list q;
  q := [ ];
  for i ∈ s → left (i), right (i), rank (i), q := null, null, 1, q & [i] rof;
  return heapify (q)
end makeheap;
```

The last heap operation is delete. It is possible to delete an arbitrary item from a leftist heap in $O(\log n)$ time if we add parent pointers to the tree representation. A better method for our purposes is to use *lazy deletion*, as proposed by Cheriton and Tarjan [4]: To delete an item, we merely mark it deleted; we carry out the actual deletion during a subsequent findmin or deletemin. To carry out findmin, we perform a preorder traversal of the tree, making a list of each nondeleted node all of whose proper ancestors are marked deleted; then we heapify the subtrees whose roots are on the list and return the root of the single tree resulting. Implementation of deletemin is similar. With this method we can also if we wish perform *lazy melding*: To meld two heaps we create a dummy node whose children are the roots of the two heaps to be melded. During findmin and deletemin we treat dummy nodes as if they were marked deleted. The following programs implement lazy melding

and findmin using this method. (We leave as an exercise implementing delete and deletemin and modifying insert and makeheap to mark newly inserted items nondeleted.) The program lazymeld marks dummy nodes by giving them a key of minus infinity.

```
heap function lazymeld (heap h1, h2);
  if h1 = null → return h2 | h2 = null → return h1
    | h1 ≠ null and h2 ≠ null →
      item i;
      i := create item;
      if rank (h1) < rank (h2) → h1 ↔ h2 fi;
      left (i), right (i), key (i), rank (i) := h1, h2, −∞, rank (h2) + 1;
      return i
  fi
end lazymeld;

item function findmin (modifies heap h);
  h := heapify (purge (h));
  return h
end findmin;

list function purge (heap h);
  return if h = null → [ ]
          | h ≠ null and key (h) > −∞ and not deleted (h) → [h]
          | h ≠ null and (key (h) = −∞ or deleted (h)) →
            purge (left (h)) & purge (right (h))
        fi
end purge;
```

Note. The purge program makes a list of all nondummy, nondeleted nodes in h all of whose proper ancestors are dummy or deleted. The heapify program must use the original version of meld rather than lazymeld. □

With lazy melding and lazy deletion, the time for a meld or deletion is $O(1)$; the time for a findmin is $O(k \max \{1, \log (n/(k+1))\})$, where k is the number of dummy and deleted items discarded from the heap. Lazy deletion is especially useful when there is a way to mark deleted items implicitly. We shall see an example of this in Chapter 6.

3.4. Remarks. There are several other heap operations that can be added to our repertoire with little loss in efficiency. One such operation is addtokeys (x, h), which adds real number x to the key of every item in heap h. To implement this operation we change our heap representation so that, instead of storing a key with each item, we store a key difference:

$$\Delta key\ (x) = \begin{cases} key\ (x) & \text{if } x \text{ is a tree root,} \\ key\ (x) - key\ (p(x)) & \text{if } p(x) \text{ is the parent of } x. \end{cases}$$

With this representation we can evaluate the key of an item x by summing key differences along the path from x to the tree root. The operation addtokeys (x, h) takes $O(1)$ time: we add x to the key difference of the tree root. The asymptotic running times of the other heap operations are unaffected by this change in the data structure. We shall see another use of storing differences in the next chapter. Cheriton and Tarjan [4] first proposed using key differences in heaps, borrowing the idea from an algorithm of Aho, Hopcroft and Ullman [2] for computing depths in trees.

Although leftist trees give a simple and efficient way to represent meldable heaps, almost any class of balanced trees will do almost as well; all we need is a definition of balance that allows rapid melding of two balanced trees containing items arranged in heap order. Empirical and theoretical evidence gathered by Brown [3] suggests that the class of "binomial trees" [9] gives the fastest implementation of meldable heaps when constant factors are taken into account. However, on binomial heaps the heap operations are harder to describe and implement than on leftist heaps.

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] ——, *On finding lowest common ancestors in trees*, SIAM J. Comput., 5 (1975), pp. 115–132.

[3] M. R. Brown, *Implementation and analysis of binomial queue algorithms*, SIAM J. Comput., 7 (1978), pp. 298–319.

[4] D. Cheriton and R. E. Tarjan, *Finding minimum spanning trees*, SIAM J. Comput., 5 (1976), pp. 724–742.

[5] C. A. Crane, *Linear lists and priority queues as balanced binary trees*, Tech. Rep. STAN-CS-72-259, Computer Science Dept., Stanford Univ., Stanford, CA, 1972.

[6] R. W. Floyd, *Algorithm* 245: *Treesort* 3, Comm. ACM, 7 (1964), p. 701.

[7] D. B. Johnson, *Priority queues with update and finding minimum spanning trees*, Inform. Process. Lett., 4 (1975), pp. 53–57.

[8] D. E. Knuth, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[9] J. Vuillemin, *A data structure for manipulating priority queues*, Comm. ACM, 21 (1978), pp. 309–314.

[10] J. W. J. Williams, *Algorithm* 232: *Heapsort*, Comm. ACM, 7 (1964), pp. 347–348.