10 Heaps and Heapsort

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:

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

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. 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:

```
void EXTRACTMIN(void)
  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 39. 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 μ has a rank larger than or equal to the rank of its parent. Symmetrically, μ 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.

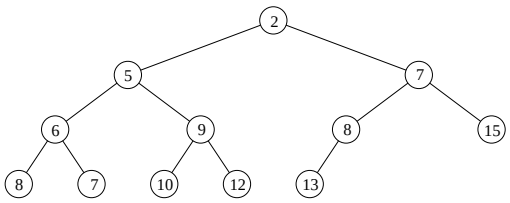

Figure 39: Ranks increase or, more precisely, do not decrease from top to bottom.

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 40. 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 $A[\lfloor i/2 \rfloor]$. The item with minimum rank is stored in the first element:

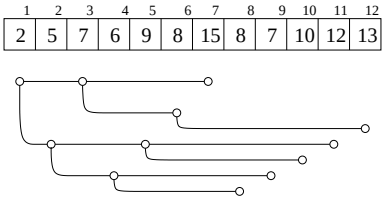

Figure 40: The binary tree is layed out in a linear array. The root is placed in $A[1]$, its children follow in $A[2]$ and $A[3]$, etc.

```
item FINDMIN(int n)
  assert n ≥ 1; return A[1].
```

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

Deleting the minimum. We first study the problem of repairing the heap-order if it is violated at the root, as shown in Figure 41. 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 the smaller of its children:

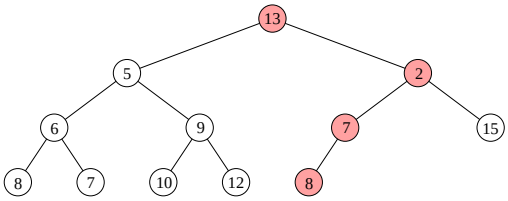

Figure 41: The root is exchanged with the smaller of its two children. The operation is repeated along a single path until the heap-order is repaired.

```
void SIFT-DN(int i, n)
  if 2i ≤ n then
    k = arg min{A[2i], A[2i+1]}
    if A[k] < A[i] then SWAP(i, k);
                        SIFT-DN(k, n)
    endif
  endif.
```

Here we assume that $A[n+1]$ is defined and larger than $A[n]$. Since a path has at most $\log_2 n$ edges, the time to repair the heap-order takes time at most $O(\log n)$. To delete the minimum we overwrite the root with the last element, shorten the heap, and repair the heap-order:

```
void DELETEMIN(int *n)
  A[1] = A[*n]; *n--; SIFT-DN(1, *n).
```

Instead of the variable that stores n, we pass a pointer to that variable, $*n$, in order to use it as input and output parameter.

Inserting. Consider repairing the heap-order if it is violated at the last position of the heap. In this case, 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 ≥ 2 then k = ⌊i/2⌋;
    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(int *n, item x)
  *n++; A[*n] = x; SIFT-UP(*n).
```

A heap supports FINDMIN in constant time and INSERT and DELETEMIN in time $O(\log n)$ each.

Sorting. Priority queues can be used for sorting. The first step throws all items into the priority queue, and the second step takes them out in order. Assuming the items are already stored in the array, the first step can be done by repeated heap repair:

```
for i = 1 to n do SIFT-UP(i) 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 $\sum_{i=1}^{n} \log_2 i \le n \log_2 n$. 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. We modify the first step accordingly, and we implement the second step to rearrange the items in sorted order:

```
void HEAPSORT(int n)
  for i = n downto 1 do SIFT-DN(i, n) endfor;
  for i = n downto 1 do
    SWAP(i, 1); SIFT-DN(1, i − 1)
  endfor.
```

At each step of the first `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 $O(n)$. Figure 42 shows the array after each iteration of the second `for`-loop. Note how the heap gets smaller by one element each step. A sin-

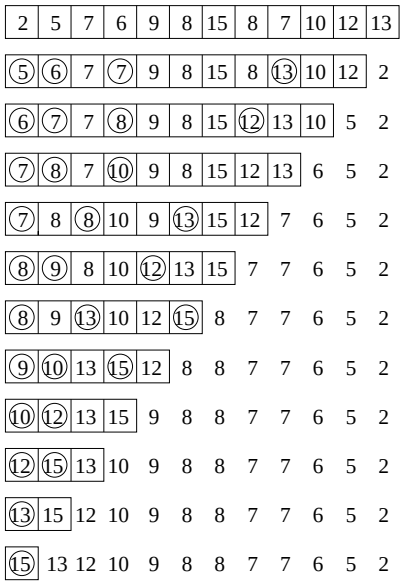

Figure 42: Each step moves the last heap element to the root and thus shrinks the heap. The circles mark the items involved in the sift-down operation.

gle 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.

Analysis of heap construction. We return to proving that the bottom-up approach to constructing a heap takes only $O(n)$ 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 43. To avoid

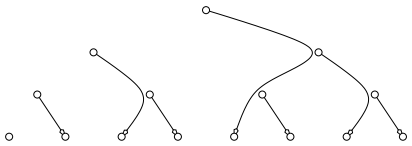

Figure 43: Each node generates a path that shares no edges with the paths of the other nodes.

drawing any edge twice, we always first swap to the right and then continue swapping to the left until we arrive at the last level. This introduces only a small inaccuracy in our estimate. 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.