7
Heaps

7.1 Priority Queues

One of the first data structures we encountered in Chapter 2 was a queue. The idea of dealing with some records on a FIFO – first in, first out – basis seems to be a reasonable choice and enjoys widespread acceptance. Nonetheless, there are situations in which the order should be different. A case in point would be the emergency room of any hospital: there is a constant influx of new potential patients, all waiting to being taken care of. However, they will be treated by order of medical urgency (which some professional has to assess) rather than by arrival time. Similarly, we all have many items on our to-do lists, but some of these items may be given a higher priority than others.

Translating the problem into mathematical terms, we are looking for a data structure we shall call a *priority queue*, storing a collection of records, each of which has been assigned a value considered as its *priority*. We wish to be able to perform updates, as inserting new elements, deleting others, or changing the priority of some. In addition, there should be fast access to the element with highest priority.

A simple solution could be a list that has been sorted by nonincreasing priority. The element with highest priority would then be the first, but updates might require $\Omega(n)$ time for a list of n records. Another option would be to use a search tree, which takes $\Omega(n \log n)$ time to be built, as we shall see in Chapter 10, but then updating and extracting the maximum element can be done in time $O(\log n)$.

This chapter presents a better alternative. It uses a special form of a binary tree, called a *heap* and achieves $O(\log n)$ time updates and access to the maximum element in $O(1)$. The time to build a heap of n elements is only $O(n)$.

The name comes possibly from the fact that beside the rules given in the definition, there is little order between the elements of a heap, which conveys

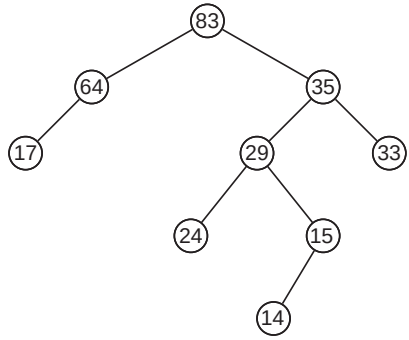


Figure 7.1. Example of a heap.

the impression of a disordered pile or elements that have been thrown one on another. It might remind many card games, in which the players throw cards in turn onto some pile, and only the top card is considered as being accessible.

7.2 Definition and Updates

Definition 7.1. A *heap* is a binary tree for which the value stored in any node is larger or equal to the values stored in its children, if they exist.

An immediate consequence of this definition is that the largest element of a heap is always stored in the root, as can also be seen in the sample heap shown in Figure 7.1.

The elements along any path from the root to a leaf appear in non-increasing order, but there is no specific order if one scans the elements by layers.

To remove the maximal element, or in fact any other node, one proceeds similarly to what we saw for binary search trees: only the value of the node is erased, and it has to be replaced. In the case of a heap, the value of a node v to be deleted should be replaced by the larger of the values of the children of v, if they exist. This process continues top-down until a leaf is reached, which can be removed without causing problems. For example, removing the root v_{83} of the heap in Figure 7.1 is done by replacing it with v_{64}, which in turn is replaced by v_{17}; the latter is then simply removed from the tree.

For the insertion of a new element, there is a certain degree of freedom: the new node should first be inserted in time $O(1)$ as a leaf, but there is no restriction on which of the potential leaves to choose. Obviously, this might violate the heap property, so a series of comparisons is performed, this time bottom-up, until the modified tree is a heap again. For example, one could choose to insert

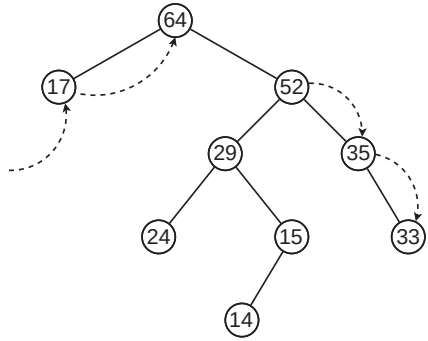


Figure 7.2. Inserting into and deleting from a heap.

first the node v_{52} as the right child of the leaf v_{33} in Figure 7.1. Since $52 > 33$, the values of the leaf v_{52} and its parent node v_{33} are swapped; but 52 is also larger than 35, so there is another exchange between v_{52} and v_{35}. Now $52 < 64$ and the update is done. Figure 7.2 shows the heap of Figure 7.1 after the deletion of the root and insertion of the value 52.

The number of operations for such updates is thus again dependent on the depth of the tree, and similarly to what we saw for binary search trees, this depth may be as large as $\Omega(n)$. The way to overcome such worst case behavior is by imposing some additional constraints, as done previously for AVL-trees and B-trees.

The best possible performance of updates in binary trees like heaps or search trees is attained by full binary trees with $n = 2^d - 1$ nodes, which have all their leaves on the same level of depth $d - 1$. For search trees, such a constraint was too restrictive and led to a more relaxed one, that of AVL-trees. For heaps, on the other hand, requiring the shape of a full tree may be efficiently supported, and we shall henceforth assume that a heap is a full binary tree. The definition of a full binary tree has to be extended for any number of nodes n, not just those of form $2^d - 1$ for some d:

Definition 7.2. A full binary tree with n nodes is a binary tree in which all the leaves are on the same level $d - 1$, if $n = 2^d - 1$, or, when $2^d \leq n < 2^{d+1}$, the leaves are on adjacent levels, with $n - 2^d + 1$ leaves left justified on level d, for some $d \geq 1$.

For example, the structure of the tree in Figure 7.3 is that of a full binary tree, with $n = 12$ nodes. Accordingly, there is one leaf on level $d - 1 = 2$, and there are $n - 2^d + 1 = 5$ leaves that are left justified on the lowest level $d = 3$. The values in the nodes show that the tree is a heap.

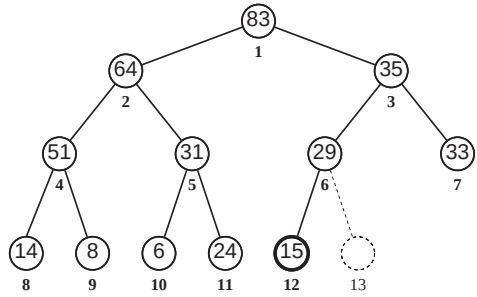


Figure 7.3. A heap as a full binary tree.

We saw earlier that to insert a new element, one has to create a new leaf, but that there is much freedom of where this leaf should be in the tree. Now the choice is more restricted. In fact, to maintain the shape of a full binary tree, a new element is first inserted as a leaf at the only location not violating the constraint: continuing the sequence of left justified leaves on the lowest level, or, if this level is already filled, as the leftmost leaf of the next level. For our example tree, this location is indicated in broken lines in Figure 7.3. In a second stage, the inserted element is then propagated bottom up if necessary, to its proper place in the order imposed by the heap.

Similarly, the only leaf that can be removed without losing the shape of a full binary tree is the rightmost one on the lowest level, which, in the example of Figure 7.3, is emphasized and contains the value 15. Denote this extreme element by v_e. Therefore, the procedure given earlier for the removal of the maximum element, or any other node w, has to be modified. We cannot simply erase the value of a node and move repeatedly the larger of its children's values to the higher level, because that could create a gap on the lowest level. So we start the deletion process by *replacing* the value of the node w to be removed by the value of v_e, and by removing the leaf v_e from the heap. Then the new value of w is percolated top down, if necessary, swapping its value with the larger of its children's values, until the heap property is restored. All the update operations can thus be performed in time $O(\log n)$.

The upper tree in Figure 7.4 is the result of deleting the largest value 83 from the heap of Figure 7.3. The root v_{83} is replaced by the leaf v_{15}, which is swapped twice with its left child. The lower tree in Figure 7.4 is obtained by inserting the value 42 into the heap of Figure 7.3. The new leaf v_{42} is inserted as the 13th node of the heap, and is then swapped twice with its parent node.

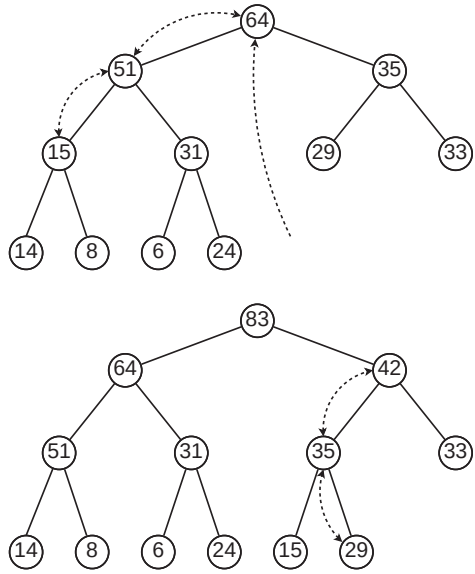


Figure 7.4. Deleting 83 from and inserting 42 into the tree of Figure 7.3.

7.3 Array Implementation of Heaps

There remains a small technical problem. In the trees we have seen so far, there were only pointers from a node to its children, not to its parent, but here we need the ability to navigate both downward and upward in the tree. This was true also for the updates in the AVL trees of Chapter 5, but there no upward pointers were needed, because one could save all the necessary nodes, those on the path from the root to a leaf, in a stack.

Fortunately, the fact that heaps are restricted to have the form of a full binary tree enables a much simpler implementation, without any pointers at all. Any full binary tree with n nodes can be implemented as an array of size n, indexed by the numbers 1 to n. To build the correspondence between nodes of the tree and array cells, just number the nodes sequentially top down, and in each level from left to right. For example, this numbering appears in the heap of Figure 7.3, just underneath the nodes. The heap can therefore be represented in an array as

1	2	3	4	5	6	7	8	9	10	11	12
83	64	35	51	31	29	33	14	8	6	24	15

Denote by $v(i)$ the node indexed by i in the heap. The simple rules are that the left child of the node $v(i)$ is $v(2i)$ and the right child of $v(i)$ is $v(2i + 1)$, for

$1 \le i \le \lfloor n/2 \rfloor$, and hence the parent node of $v(j)$ is $v(\lfloor j/2 \rfloor)$, for $2 \le j \le n$. This explains why it is important to start the indexing with 1, and not with 0, as would be set by default in the C/C++ languages. There is thus no need for any pointers, and one can pass from one node to the next or previous level just by multiplying or dividing the index by 2. These operations are, moreover, easily implemented, since all one has to consider is the standard binary representation of the index. If i is represented by $b_k \cdots b_2 b_1 b_0$, with $b_j \in \{0, 1\}$, that is $i = \sum_{j=0}^{k} b_j 2^j$, then the parent of $v(i)$ is indexed by $b_k \cdots b_2 b_1$, the grand-parent by $b_k \cdots b_3 b_2$, and so on. The left child of $v(i)$ will be indexed by $b_k \cdots b_2 b_1 b_0 \mathbf{0}$, and its right child by $b_k \cdots b_2 b_1 b_0 \mathbf{1}$.

For example, the following is a list of indices of nodes, given in both decimal and binary, starting with the node $v(25)$, and passing from one node to its parent, up to the root $v(1)$:

25	11001
12	1100
6	110
3	11
1	1

Going from $v(25)$ to its two children yields

25	11001
50	110010
51	110011

Any element to be deleted will first be replaced by the one in the last entry of the array, indexed n, and any new element is first inserted in the next available spot, which should be indexed $n + 1$. The time complexity of all the updates clearly remains $O(\log n)$.

7.4 Construction of Heaps

A heap can obviously be constructed by inserting its elements in any order into an initially empty tree or array. This could take about $\sum_{i=1}^{n} \log i = \Omega(n \log n)$ time. The question is whether it can be done faster when the whole set of records is given at once. One could sort the array by non-increasing values, since a sorted array is a particular form of a heap, but sorting will also require at least $\Omega(n \log n)$ time, as we shall see in Chapter 10. The order imposed by a heap is less restrictive than strictly sorted order, so there is hope for the complexity of the construction of a heap to be less than the lower bound for sorting.

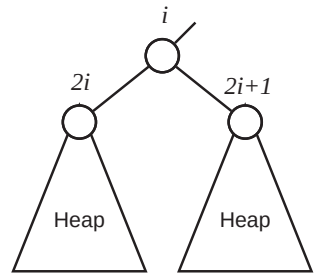
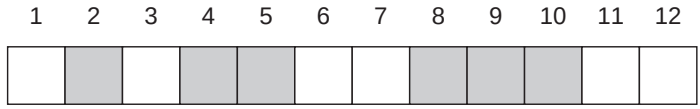


Figure 7.5. Schematic of the input of Heapify.

As a first step, consider a function called Heapify(i, j), working on the subarray between and including the indices i and j. More precisely, Heapify(i, j) works on the subtree $T_{v(i)}$ rooted at the node $v(i)$, but excluding all nodes with indices larger than j. The usefulness of the second argument j will become clear later. One has to remember that the array is only a convenient way to represent the heap, which is still defined as a tree. Therefore, the subarray $[i, j]$ processed by Heapify(i, j) does not really refer to all the elements in the range, but only to those belonging to the subtree rooted at $v(i)$. For example, Heapify$(2, 10)$ deals only with the grayed entries in



which is most easily understood when comparing the array with the corresponding tree form in Figure 7.3: the elements 3, 6, 7, and 12 remain white because they are not in the subtree rooted by 2, and element 11 is excluded because it is not in the given range [2, 10].

The purpose is to transform the subrange of the array given as parameter, or rather the corresponding subtree $T_{v(i)}$, into a heap. The definition of Heapify will be simplified by the additional assumption that the two subtrees of $T_{v(i)}$, which are $T_{v(2i)}$ and $T_{v(2i+1)}$ are already heaps, as given schematically in Figure 7.5.

Therefore, denoting the given array by A, if the value at the root $v(i)$ is not smaller than the values of its children, that is, if

$$A[i] \geq A[2i] \qquad \text{and} \qquad A[i] \geq A[2i+1],$$

then the entire subtree is already a heap, and no further action is needed. Otherwise, let us swap the values of $v(i)$ and the larger of its children, for example $v(2i)$. But then we are again in a situation in which a subtree is given that is *almost* a heap: its own subtrees, $T_{v(4i)}$ and $T_{v(4i+1)}$ are heaps, and only the root

```
Heapify(i, j)
    maxind ⟵ i
    if 2i ≤ j and A[2i] > A[maxind] then
            maxind ⟵ 2i
    if 2i + 1 ≤ j and A[2i + 1] > A[maxind] then
            maxind ⟵ 2i + 1
    if maxind ≠ i then
            A[i] ⟷ A[maxind]
            Heapify(maxind, j)
```

Figure 7.6. Heapify transforms a tree into a heap, assuming its subtrees are heaps.

$v(2i)$ may violate the heap condition. This may therefore be treated recursively. Figure 7.6 shows the formal definition of Heapify(i, j).

After the first two if statements, the variable *maxind* will be the index of the largest element among $\{A[i], A[2i], A[2i+1]\}$, which are the root and its two children. The fact that they are in the given range at all is checked by the conditions $2i \leq j$ and $2i + 1 \leq j$. If at this stage $maxind = i$, the tree is a heap and we are done; otherwise the value of the larger of the two children is swapped with the value of the root (this action is symbolized by the two-sided arrow $\longleftrightarrow$), and Heapify is invoked recursively for the subtree.

Background Concept: Swapping Two Elements

The swap of two elements used here is a frequent action that appears in many algorithms and therefore deserves some comments. The simplest and possibly also the best way to swap the values of two variables x and y is by means of a third, auxiliary, variable T:

```
swap1(x, y)
    T ⟵ x
    x ⟵ y
    y ⟵ T
```

An alternative, using also only three assignment, but no extra space for a temporary variable, exploits the properties of the Boolean bit-wise XOR operation, denoted by $x \oplus y$:

```
swap2(x, y)
    x ⟵ x ⊕ y
    y ⟵ x ⊕ y
    x ⟵ x ⊕ y
```

This elegant procedure, consisting of three copies of almost the same command, uses the fact that XOR is both associative and commutative. At the end of the execution of the second line, one gets

$$y = x \oplus y = (x \oplus y) \oplus y = x \oplus (y \oplus y) = x \oplus 0 = x.$$

And after the third line, one gets:

$$x = x \oplus y = (x \oplus y) \oplus x = (y \oplus x) \oplus x = y \oplus (x \oplus x) = y \oplus 0 = y.$$

To evaluate the time complexity of the procedure, note that invoking Heapify(i, j) generates a sequence of recursive calls to Heapify, and that the number of commands between consecutive calls is bounded by a constant. On the other hand, since $i \geq 1$ and $j \leq n$, the number of calls is bounded by the number of times i can be doubled until it exceeds j, which is at most $\log_2 n$.

The question is now how to employ the Heapify procedure to build a heap from scratch. The answer lies in the simple idea of processing a given input array *backward*, or in terms of the corresponding binary tree, building the heap *bottom-up*. A full binary tree with n nodes has $\lceil \frac{n}{2} \rceil$ leaves, each of them being a heap, as the heap condition is trivially fulfilled for nodes without children. The trees rooted by the parent nodes of these leaves, in our example of Figure 7.3, the nodes indexed 6, 5, 4 and 3, satisfy then the condition that their own subtrees are heaps, so Heapify can be applied on them. This paves the way to apply subsequently Heapify to the trees rooted by the grand-parents (2 and 1 in the example), etc. This yields the simple construction routine:

```
Buildheap(n)
    for i ⟵ ⌊n/2⌋ by step −1 to 1
        Heapify(i, n)
```

An immediate upper bound on the complexity, given that Heapify takes $O(\log n)$ time and that there are about $n/2$ iterations, is $O(n \log n)$. This turns out to be overly pessimistic. A tighter bound can be achieved if one realizes that only the entire tree has depth $\log_2 n - 1$, and that the subtrees on which the recursive calls of Heapify act are shallower.

To simplify the notation, assume that $n = 2^{k+1} - 1$, that is, we consider a full tree of depth k having all its $\frac{n+1}{2}$ leaves on the same level k. In Buildheap, there is a single call to Heapify with a tree of depth k, there are two such calls with trees of depth $k - 1$, and more generally, there are 2^i calls to Heapify with

trees of depth $k-i$, for $0 \le i < k$. The total number N of calls is therefore

$$N = 1 \cdot k + 2(k-1) + 2^2(k-2) + \cdots + 2^{k-1} \cdot 1 = \sum_{i=0}^{k-1} 2^i(k-i).$$

Such a summation can be dealt with by writing it out explicitly line by line, starting with the last index $k-1$ and going backward, and then summing by columns:

$$\begin{aligned} N = \quad & 2^{k-1} \\ & + 2^{k-2} + 2^{k-2} \\ & + 2^{k-3} + 2^{k-3} + 2^{k-3} \\ & + \qquad \cdots \\ & + 2^{k-k} + 2^{k-k} + 2^{k-k} + \cdots + 2^{k-k}. \end{aligned}$$

The first column is the geometric series $\sum_{i=0}^{k-1} 2^i$, well known to sum up to $2^k - 1$. The second column is similar, but the last index is $k-2$, so the sum is $2^{k-1} - 1$, and in general, we get that the jth column is $\sum_{i=0}^{k-j} 2^i$ summing up to $S(j) = 2^{k-j+1} - 1$, for $1 \le j \le k$. Summing the summations, we finally get

$$N = \sum_{j=1}^{k} S(j) = \sum_{j=1}^{k} (2^{k-j+1} - 1) = \sum_{i=1}^{k} 2^i - k = 2^{k+1} - 2 - k < n.$$

The overall complexity of building a heap of n elements is thus only $O(n)$.

7.5 Heapsort

We conclude this chapter on heaps by preponing the description of a sorting method that should in fact be dealt with only in Chapter 10. It is, however, strongly connected to the current subject and a good example of an important application of heaps.

Suppose that an array of n numbers is given and that it has already been transformed into a heap. The array can then be sorted by repeatedly extracting the largest number. Since on one hand the extracted numbers have to be stored somewhere, and on the other hand, the shrinking heap should form a contiguous block in the array including the first element, it is natural to store the extracted elements at the end of the array. The sorted sequence is thus constructed backward using the cells that are vacated, so that no additional space is needed for the sort.

1	2	3	4	5	6	7	8	9	10	11	12
35	31	33	15	24	29	6	14	8	51	64	83
33	31	29	15	24	8	6	14	35	51	64	83
31	24	29	15	14	8	6	33	35	51	64	83

Figure 7.7. Partially sorted array during Heapsort.

More precisely, at the end of iteration j, $1 \leq j \leq n$, the elements $A[1] \cdots A[n-j]$ form a heap and the rest of the elements, those in $A[n-j+1] \cdots A[n]$, store the j largest numbers of the original array in increasing order. The initial stage is considered here as the end of iteration 0. Figure 7.7 shows the array corresponding to the heap of Figure 7.3 at the end of iterations 3, 4, and 5. The heap is to the left of the black bar, the partially sorted array to its right.

In iteration j, the largest element of the heap, the one stored in position 1, is swapped with the last one in the current heap. This places the extracted element into its final position in the sorted array. Then the heap has to be updated, but we are in the special case in which all the elements satisfy the heap property, except possibly the root. The heap can thus be repaired, in time $O(\log j)$, by a single application of Heapify, in which the second argument is used to limit the scope of the current heap. Overall, the updates will take

$$\sum_{j=1}^{n} \log j < \sum_{j=1}^{n} \log n = O(n \log n),$$

and the formal algorithm is given by

```
Heapsort(n)
    Buildheap(n)
    for j ⟵ n by step −1 to 2
            A[1] ⟷ A[j]
            Heapify(1, j − 1)
```

In fact, Heapsort consists of two quite similar loops. The first builds the heap and the second retrieves repeatedly the largest element remaining in the heap. An easy way to remember the varying parameters of Heapify is the rhombus shaped diagram in Figure 7.8. It shows in its upper part, corresponding to

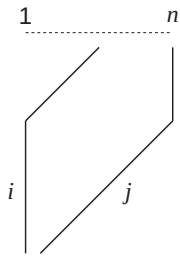


Figure 7.8. Schematic of the parameters of Heapsort.

Buildheap, that i passes from $n/2$ to 1 while j stays fixed at n; in the lower part, i remains equal to 1, but j decreases from n to 2.

As a final note it should be mentioned that though the definition of a heap given herein places the largest element at the root, one could clearly also define a *min-heap* with the smallest element on top and with the obvious adaptations to the earlier procedures. What we have defined as a heap is therefore often referred to in the literature as a *max-heap*.

Exercises

7.1 Given are two heaps A of size m and B of size n. The heaps are given explicitly as binary trees, not as arrays. Assume that all $m+n$ elements are different, that $m = 2^k - 1$ for some integer k and that $m/2 < n \leq m$. The task is to build a heap C consisting of the elements of $A \cup B$.

(a) What is the depth of the tree A, and what is the depth of the tree B?
(b) What will be the depth of the tree C?
(c) Suggest an efficient algorithm for the construction of C. What is its complexity?

7.2 What happens to the correctness and efficiency of the Buildheap procedure, if one changes the control of the main loop from

for $j \longleftarrow \lfloor \frac{n}{2} \rfloor$ by step -1 to 1 to

(a) for $j \longleftarrow n$ by step -1 to 1 ?
(b) for $j \longleftarrow 1$ to n ?

If the procedure does not work, show a counterexample with smallest possible n.

7.3 A possible generalization of a heap is a *ternary heap*, which is a full ternary tree (each internal node has 3 children). The procedure Heapify is generalized accordingly to Heapify3.

(a) A full ternary tree can also be represented as an array. How does one pass from a node to its three children? How from a node to its parent?
(b) To remove the maximum element and then restore the heap, how many calls to Heapify3 and how many comparisons on each level of the tree are needed?
(c) What may be concluded in comparison with binary heaps?

7.4 Let T be a binary tree in which each node is annotated by a pair of integers (s, h). T is a *search-heap* if it is a search tree according to s and a heap according to h.

(a) Given is the following set of (s, h) pairs: $\{(26, 16), (17, 8), (5, 1), (24, 34), (12, 5), (41, 12), (49, 3), (52, 9), (54, 29), (34, 2), (38, 14), (10, 6)\}$. Build the search-min-heap and the search-max-heap for this set.
(b) Assuming that all s-values are different, and all h-values are different, show that there is exactly one search-heap (more precisely, one for *min*, one for *max*) for any set of pairs.
(c) For a given such set of (s, h) pairs, write an algorithm for the construction of the corresponding search-heap.

7.5 Suggest a data structure supporting the following operations in the given time complexities:

(a) Build in time $O(n)$ for a set of n elements.
(b) Insert(x), inserting a new element x into the structure in time $O(\log n)$.
(c) min and max, finding minimum and maximum elements in time $O(1)$.
(d) delete-min and delete-max. removing the minimum and maximum elements in time $O(\log n)$.

7.6 Apply Heapsort with a max-heap to sort an array of length n into increasing order. How many comparisons and swaps are executed if the array

(a) is already given in increasing order?
(b) is given in decreasing order?