6. Data structures

Computer Science Department,
College of Computer and Information Sciences,
King Saud University.

CSC 311: Design and Analysis of Algorithms\(^1\)
Dr. Waleed Alsalih

6.2 Heaps [Section 6.4]

A heap is a binary tree, with keys assigned to its nodes, that satisfies the following two conditions:

1. The binary tree is complete, i.e., all its levels are full except possibly the last level where some rightmost leaves may be missing.

2. For max-heap (min-heap), the key of a node is greater (smaller) than those of its children. We will assume max-heap unless otherwise stated.

Fig. 1 shows an example of a max-heap.

An array can be used to represent a heap as follows. The key of the root is stored in \(A[1]\) and \(A[0]\) is not used. The children of \(A[i]\) are \(A[2i]\) and \(A[2i+1]\). Leaf nodes will occupy the last \(\lceil n/2 \rceil\) positions. The heap in Fig. 1 can be represented as follows: [-, 100, 50, 20, 40, 2, 14].

Heap construction

One way to construct a heap of a set of keys is the bottom-up algorithm.

\(^1\)This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.
6. Data structures

Figure 1: An example of a max-heap.

Algorithm HeapBottomUp($H[1..n]$)

for $i = \lfloor n/2 \rfloor$ downto 1 do
    Sift($H[1..n], i$);
end

Worst-case time complexity

Assume, for simplicity, that $n = 2^k - 1$ (i.e., all levels are full). Then, the total number of key comparisons in the worst case is:

$$
\sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h - i) = \sum_{i=0}^{h-1} 2(h - i)2^i = 2(n - \log_2(n + 1)) \in O(n),
$$

where $h$ is the height of the tree, i.e., $h = \lfloor \log_2 n \rfloor$.

Exercise:

Prove that

$$
\sum_{i=0}^{h-1} 2(h - i)2^i = 2(n - \log_2(n + 1)).
$$

Hints:
6. Data structures

---

**Algorithm Sift**($H[1..n], k$)

\[
v := H[k];
heap := false;
while not heap and $2k \leq n$ do
    \( j := 2k; \)
    if $j < n$ then
        if $H[j] < H[j + 1]$ then
            \( j := j + 1; \)
        end
    end
    if $v \geq H[j]$ then
        heap := true;
    else
        \( H[k] := H[j]; \)
        \( k := j; \)
    end
end
\( H[k] := v; \)

\[
\sum_{i=0}^{n} i2^i = 2 + 2^{n+1}(n - 1), \text{ and}
\]

\[
\sum_{i=0}^{h} 2^i = 2^{h+1} - 1.
\]

**Root deletion**

Consider an algorithm that removes the maximum key (i.e., the root) from a heap.

**Heap sort**

Consider an algorithm that sorts an array in a non-decreasing order using heap operations.
6. Data structures

**Algorithm HeapRemoveRoot**($H[1..n]$)

\[
\begin{align*}
v & := H[1]; \\
H[1] & := H[n]; \\
n & := n - 1; \\
\text{if } n > 0 \text{ then} & \quad \text{Sift}(H[1..n], 1); \\
\text{end} & \\
\text{return } v;
\end{align*}
\]

**Algorithm HeapSort**($A[1..n]$)

\[
\begin{align*}
\text{HeapBottomUp}(A[1..n]); \\
\text{for } i = 1 \text{ to } n \text{ do} & \quad B[i] = \text{HeapRemoveRoot}(A[1..n]); \\
\text{end} & \\
\text{return } B[1..n];
\end{align*}
\]

**Time complexity of the heap sort algorithm**

This algorithm has two parts: heap construction and a sequence of remove operations. The first part runs in $O(n)$ time. The total number of key comparisons in the second part $C(n)$ meets the following inequality:

\[
C(n) \leq 2[\log(n - 1)] + 2[\log(n - 2)] + \cdots + 2[\log 1] \leq 2 \sum_{i=1}^{n-1} \log i \leq 2 \sum_{i=1}^{n-1} \log(n - 1) = 2(n - 1) \log(n - 1) \leq 2n \log n \in O(n \log n).
\]

So the overall complexity of the algorithm is $O(n \log n)$. 