Heaps. Heapsort.
(CLRS 6)

1 Introduction

• We have discussed several fundamental algorithms (e.g. sorting)
• We will now turn to data structures; Play an important role in algorithms design.
  – Today we discuss priority queues and next time structures for maintaining ordered sets.

2 Priority Queue

• A priority queue supports the following operations on a set \( S \) of \( n \) elements:
  – INSERT: Insert a new element \( e \) in \( S \)
  – FINDMIN: Return the minimal element in \( S \)
  – DELETEMIN: Delete the minimal element in \( S \)

• Sometimes we are also interested in supporting the following operations:
  – CHANGE: Change the key (priority) of an element in \( S \)
  – DELETE: Delete an element from \( S \)

• We can obviously sort using a priority queue:
  – Insert all elements using INSERT
  – Delete all elements in order using FINDMIN and DELETEMIN

• Priority queues have many applications, e.g. in discrete event simulation, graph algorithms

2.1 Array or List implementations

• The first implementation that comes to mind is ordered array:

\[
\begin{array}{cccccccccccc}
1 & 3 & 5 & 6 & 7 & 8 & 9 & 11 & 12 & 15 & 17 \\
\end{array}
\]

  – FINDMIN can be performed in \( O(1) \) time
  – DELETEMIN and INSERT takes \( O(n) \) time since we need to expand/compress the array after inserting or deleting element.
• If the array is unordered all operations take $O(n)$ time.

• We could use double linked sorted list instead of array to avoid the $O(n)$ expansion/compression cost
  – but INSERT can still take $O(n)$ time.

2.2 Heap implementation

• One way of implementing a priority queue is using a heap

• Heap definition:
  – Perfectly balanced binary tree
    * lowest level can be incomplete (but filled from left-to-right)
  – For all nodes $v$ we have $\text{key}(v) \geq \text{key}(\text{parent}(v))$

• Example:

\[
\begin{array}{c}
\text{2} \\
\text{5} & \text{3} \\
\text{9} & \text{19} & \text{11} & \text{4} \\
\text{15} & \text{14}
\end{array}
\]

• Heap can be implemented (stored) in two ways (at least)
  – Using pointers
  – In an array level-by-level, left-to-right
    Example:

\[
\begin{array}{cccccccc}
\text{2} & \text{5} & \text{3} & \text{9} & \text{19} & \text{11} & \text{4} & \text{15} & \text{14}
\end{array}
\]

  * the left and right children of node in entry $i$ are in entry $2i$ and $2i + 1$, respectively
  * the parent of node in entry $i$ is in entry $\left\lfloor \frac{i}{2} \right\rfloor$

• Properties of heap:
  – Height $\Theta(\log n)$
  – Minimum of $S$ is stored in root
• Operations:
  
  – **Insert**
    * Insert element in new leaf in leftmost possible position on lowest level
    * Repeatedly swap element with element in parent node until heap order is reestablished (**up-heapify**)
    Example: Insertion of 4

  – **FindMin**
    * Return root element
  
  – **DeleteMin**
    * Delete element in root
    * Move element from rightmost leaf on lowest level to the root (and delete leaf)
    * Repeatedly swap element with the smaller of the children elements until heap order is reestablished (**down-heapify**)
    Example:

  – **Change** and **Delete** can be handled similarly in \(O(\log n)\) time
    * Note: Assuming that we know the element to be changed/deleted (we cannot search in a heap!!)

• **Correctness**: Exercise.

• **Running time**: All operations traverse at most one root-leaf path \(\Rightarrow O(\log n)\) time.

  • Sorting using heap (**HeapSort**) takes \(\Theta(n \log n)\) time.
    – \(n \cdot O(\log n)\) time to insert all elements (build the heap)
    – \(n \cdot O(\log n)\) time to output sorted elements

• Sometimes we would like to build a heap faster than \(O(n \log n)\)
  
  – **BUILDHEAP**
    * Insert elements in any order in perfectly balanced tree
* DOWN-HEAPIFY all nodes level-by-level, bottom-up

- Correctness:
  * Induction on height of tree: When doing level $i$, all trees rooted at level $i - 1$ are heaps.

- Analysis:
  * The leaves are at height 0, the root is at height $\log n$
  * $n$ elements $\Rightarrow \leq \left\lceil \frac{n}{2} \right\rceil$ leaves $\Rightarrow \left\lceil \frac{n}{2^h} \right\rceil$ elements at height $h$
  * Cost of DOWN-HEAPIFY on a node at height $h$ is $h$
  * Total cost: $\sum_{i=1}^{\log n} h \cdot \left\lceil \frac{n}{2^h} \right\rceil = \Theta(n) \cdot \sum_{i=1}^{\log n} \frac{h}{2^h}$
  * It can be shown that $\sum_{i=1}^{\log n} \frac{h}{2^h} = O(1) \Rightarrow$ the total buildheap cost is $\Theta(n)$

* Computing $\sum_{i=1}^{n} \frac{h}{2^h}$ and $\sum_{i=1}^{\infty} \frac{h}{2^h}$
  - Differentiate $\sum_{h=0}^{n} x^h = \frac{1-x^{n+1}}{1-x}$, respectively $\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$ (assuming $|x| < 1$)
  - $\sum_{h=0}^{\infty} h x^{h-1} = \frac{1}{(x-1)^2} \Rightarrow \sum_{h=0}^{n} h x^h = \frac{x}{(x-1)^2} \Rightarrow \sum_{h=0}^{n} \frac{h}{2^h} = \frac{1/2}{(1/2-1)^2} = O(1)$