1 Heaps

Heaps are data structures that make it easy to find the element with the most extreme value in a collection of elements. A Min-Heap prioritizes the element with the smallest value, while a Max-Heap prioritizes the element with the largest value. Because of this property, heaps are often used to implement priority queues.

You can find more about heaps by reading pages 151–169 in CLRS.

1.1 Representing a Heap

While a heap can be represented as a regular tree, it is often more efficient to store a binary heap as an array. We call the first element in the heap element 1. Now, given an element $i$, we can find its left and right children with a little arithmetic:

Exercise.
- $\text{Parent}(i) =$
- $\text{Left}(i) =$
- $\text{Right}(i) =$

Solution.
The array representation is simplified by calling the first element in the heap 1:
- $\text{Parent}(i) = \lfloor \frac{i}{2} \rfloor$
- $\text{Left}(i) = 2i$
- $\text{Right}(i) = 2i + 1$

The completeness requirement makes sure this representation of heaps is compact.
1.2 Heap operations

1.2.1 Max-Heapify

Max-Heapify($H, N$): Given that the children of the node $N$ in the Max-Heap $H$ are each the root of a Max-Heap, rearranges the tree rooted at $N$ to be a Max-Heap.

Max-Heapify($H, N$):

Require: Left($N$), Right($N$) are each the root of a Max-Heap

($l, r) \leftarrow (\text{Left}(N), \text{Right}(N))$

if Exists($l$) and $H[l] > H[N]$ then

largest $\leftarrow l$

else

largest $\leftarrow N$

end if

if Exists($r$) and $H[r] > H[\text{largest}]$ then

largest $\leftarrow r$

end if

if largest $\neq N$ then

SWAP($H[N], H[\text{largest}]$)

Max-Heapify($H, \text{largest}$)

end if

Ensure: $N$ is the root of a Max-Heap

Exercise.

- Run Max-Heapify with $N = 1$ on $H = [14, 16, 10, 8, 7, 9, 6, 2, 4, 1]$

- What is Max-Heapify’s run-time?

Solution.

- Max-Heapify($H, N$) returns a max-heap $[16, 14, 10, 8, 7, 9, 6, 2, 4, 1]$

- Runs in $O(\log n)$ time because node $N$ is moved down at most $\log n$ times.
1.2.2 Build-Heap

**Build-Heap**(*A*): Given an unordered array, makes it into a max-heap.

**Build-Heap**(*A*):

**Require:** *A* is an array.

for *i* = ⌊length(*A*)/2⌋ downto 1 do  
    Max-Heapify(*A*, *i*)
end for

**Exercise.**

- *Run Build-Heap on* *A* = [2, 1, 4, 3, 6, 5]
- *Running time (loose upper bound):*
- *Running time (tight upper bound):*

**Solution.**

- [2, 1, 4, 3, 6, 5] → [2, 1, 5, 3, 6, 4] → [2, 6, 5, 3, 1, 4] → [6, 3, 5, 2, 1, 4]
- *O*(n log n)
- *O*(n). Intuitively, we might be able to get a tighter bound because Max-Heapify is run more often at lower points on the tree, when subtrees are shallow. Making use of the fact that an *n*-element heap has height ⌊log *n*⌋ and at most ⌈*n*/2^h+1⌉ nodes of any height *h*, the total number of comparisons that will have to be made is

\[
\sum_{h=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor O(h) = O \left( n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^{h+1}} \right)
\]

It is not hard to see that \(\sum_{h=0}^{\infty} \frac{h}{2^h} = 2\). So the runtime is *O*(n).
1.2.3 Extract-Max

**Extract-Max**\( (H) \): Remove the element with the largest value from the heap.

\[
\text{Extract-Max}(H):
\]

\textbf{Require:} \( H \) is a non-empty \textsc{Max-Heap}

\[
\begin{align*}
\text{max} & \leftarrow H[\text{root}] \\
H[\text{root}] & \leftarrow H[\text{Size}(H)] \quad \{\text{last element of the heap.}\}
\text{Size}(H) & = 1 \\
\text{Max-Heapify}(H, \text{root}) \\
\text{return} & \quad \text{max}
\end{align*}
\]

\textbf{Exercise.}

- \textit{Run Extract-Max on} \( H = [6, 3, 5, 2, 1, 4] \).
- \textit{What is Extract-Max’s run time?}

\textbf{Solution.}

- \textbf{Extract-Max} first returns 6. It then moves the last element, 4, to the head (why the last element?) and \textsc{Max-Heapify} is used to maintain the heap structure:

\[
[4, 3, 5, 2, 1] \rightarrow [5, 3, 4, 2, 1]
\]

- \( O(\log n) \)
1.2.4 Insert

**Insert**($H, v$): Add the value $v$ to the heap $H$.

**Insert**($H, v$):

**Require**: $H$ is a **Max-Heap**, $v$ is a new value.

- $\text{Size}(H) += 1$
- $H[\text{Size}(H)] \leftarrow v$ \{Set $v$ to be in the next empty slot.\}
- $N \leftarrow \text{Size}(H)$ \{Keep track of the node currently containing $v$.\}

**while** $N$ is not the root and $H[\text{Parent}(N)] < H[N]$ **do**

- $\text{Swap}(H[\text{Parent}(N)], H[N])$
- $N \leftarrow \text{Parent}(N)$

**end while**

**Exercise.**

- Run **Insert**($H, v$) with $v = 8$ and $H = [6, 3, 5, 2, 1, 4]$

- **What is Insert’s runtime?**

**Solution.**

- Insert $v$ into $H$ as follows:
  - $[6, 3, 5, 2, 1, 4, 8] \rightarrow [6, 3, 8, 2, 1, 4, 5] \rightarrow [8, 3, 6, 2, 1, 4, 5]$

Here, the while loop is halted by the condition that $N$ is the root.

- **Insert’s runtime is $O(\log n)$**.
2 Graph Problems

Exercise. Answer T/F for the following problems:

(a) We know that the node with the highest post-order belongs to a source SCC. Then the node with the lowest post-order always belongs to a sink SCC.

(b) Suppose two vertices $u$ and $v$ in a directed graph satisfy $\text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v)$, then there can be no edge in either direction between $u$ and $v$.

(c) In a DFS of a directed graph $G$, the set of vertices reachable from the vertex with lowest post-order is a strongly-connected component of $G$.

(d) In a DFS of a directed graph $G$, the set of vertices reachable from the vertex with highest post-order is a strongly-connected component of $G$.

(e) If a DFS has a cross edge, the graph is not strongly connected.

Solution.

(a) **False.** Consider $G$ with vertices $a, b, c$ and edges $(a, b), (b, a), (a, c)$. DFS with $a : [1, 6], b : [2, 3], c : [4, 5]$. Although $b$ has the lowest post-order, it’s not part of a sink SCC.

(b) **False.** Consider $G$ with vertices $a, b$ and edge $(b, a)$. DFS with $a : [1, 2], b : [3, 4]$.

(c) **False.** Consider $G$ with vertices $a, b, c$ and edges $(a, b), (b, a), (a, c)$. DFS with $a : [1, 6], b : [2, 3], c : [4, 5]$. $a$ and $c$ are reachable from $b$, but $\{a, b, c\}$ are not an SCC.

(d) **False.** Consider $G$ with vertices $a, b, c$ and edges $(a, b), (c, a), (c, b)$. DFS with $a : [1, 4], b : [2, 3], c : [5, 6]$. $c$ has the highest postorder and both $a, b$ are reachable from $c$. But $G$ is not strongly connected.

(e) **False.** Consider $G$ with vertices $a, b, c$ and edges $(a, b), (b, a), (a, c), (c, b)$. $G$ is strongly connected. DFS with $a : [1, 6], b : [2, 3], c : [4, 5]$. Then $(c, b)$ is a cross-edge.