**Data Structures**

Heap, Heap Sort & Priority Queue

Tzachi (Isaac) Rosen

---

**Heap**

- Is a nearly **complete** binary tree.
  - height is $\Theta(lg n)$.

- In general, heaps can be k-ary tree instead of binary.
- A heap can be stored as an **array** $A$.
Heap property

- **Max-Heap** property:
  - for all nodes $i$, excluding the root, $A[\text{parent}(i)] \geq A[i]$.
- **Min-Heap** property:
  - for all nodes $i$, excluding the root, $A[\text{parent}(i)] \leq A[i]$.

Max-Heapify

- **Before** max-heapify
  - $A[i]$ may be smaller than its children.
  - Assume left and right sub-trees of $i$ are max-heaps.
- **After** max-heapify
  - Sub-tree rooted at $i$ is a max-heap.
Max-Heapify

\[ \text{maxHeapify}(A, i, n) \]

\[ l = \text{left}(i), \ r = \text{right}(i) \]

largest = \( i \)

if \( (l \leq n \&\& A[l] > A[\text{largest}]) \) then largest = \( l \)

if \( (r \leq n \&\& A[r] > A[\text{largest}]) \) then largest = \( r \)

if \( (\text{largest} \neq i) \) then

\[ \text{exchange } A[i] \text{ with } A[\text{largest}] \]

\[ \text{maxHeapify}(A, \text{largest}, n) \]

- **Complexity**: \( O(\lg n) \)
Building a Max-Heap

\[ \text{buildMaxHeap}(A, n) \]
\[ \text{for } (i = \lfloor n/2 \rfloor \text{ downto } 1) \text{ do} \]
\[ \text{maxHeapify}(A, i, n) \]

\[ \begin{array}{cccccccc}
1 & 14 & 3 & 6 & 7 & 8 & 10 & 9 \\
4 & 2 & 5 & & & & & \\
8 & 9 & & & & & & \\
16 & & & & & & & \\
\end{array} \]

• **Simple bound:**
  – \( O(n) \) calls to MAX-HEAPIFY,
  – Each of which takes \( O(\lg n) \),
  – Complexity: \( O(n \lg n) \).
Building a Max-Heap

- **Tighter analysis:**
  - Number of nodes of height $h \leq \lceil n/2^{h+1} \rceil$
  - The height of heap is $\lg n$,
  - The time required by `maxHeapify` on a node of height $h$ is $O(h)$,
  - So the total cost of is: $\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{h+1}} O(h) = O\left( \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2} \right)$.
  - $\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2} = \frac{1/2}{(1-1/2)^2} = 2$ (substituting $x = 1/2$ in the formula $\sum_{h=0}^{\lfloor \lg n \rfloor} x^h = \frac{x}{1-x}$ for $|x| < 1$)
  - Thus, the running time of `BUILD-MAX-HEAP` is $O(n)$.

Heapsort

- $O(n \lg n)$ **worst case**.
  - Like merge sort.
- Sorts **in place**.
  - Like insertion sort.
- Combines the **best** of both algorithms.
Heapsort

\textbf{Heapsort} (A, n)

\textbf{buildMaxHeap}(A, n)

\textbf{for} (i = n \text{ downto } 2) \textbf{do}


\textbf{maxHeapify}(A, 1, i - 1)

\begin{itemize}
  \item Complexity:
    \begin{itemize}
      \item buildMaxHeap: \textbf{O}(n)
      \item for loop:
        \begin{itemize}
          \item n - 1 times
          \item exchange elements: \textbf{O}(1)
          \item maxHeapify: \textbf{O}(\text{lg } n)
        \end{itemize}
      \item Total time: \textbf{O}(n \text{ lg } n).
    \end{itemize}
\end{itemize}

Priority Queue

\begin{itemize}
  \item Each element has a \textbf{key}.
  \item \textbf{Max-priority queue} supports operations:
    \begin{itemize}
      \item \textbf{insert} (S, x): inserts element x into set S.
      \item \textbf{maximum} (S): returns largest key element of S.
      \item \textbf{extractMax} (S): removes and returns the largest key element of S.
      \item \textbf{increaseKey} (S, x, k): increases value of element x’s key to k. (Assume k \geq x’s current key value).
    \end{itemize}
  \item \textbf{Min-priority queue} supports similar operations.
\end{itemize}
Maximum

maximum (A)
return A[1]

• Complexity : O(1).

ejectMax (A, n)
max = A[1]
maxHeapify(A, 1, n - 1)
return max

• Complexity : O(lg n).

Increasing Key

increasingKey (A, i, key)
A[i] = key
while (i > 1 & A[parent(i)] < A[i]) do
exchange A[i] with A[parent(i)]
i ← parent(i)

• Complexity : O(lg n).
Insertion

\textbf{insert} \ (A, \ key, \ n)

\[ A[n + 1] = -\infty \]

\textbf{increasingKey}(A, \ n + 1, \ key)

- Complexity : \( O(\lg n) \).

Tzachi (Isaac) Rosen