Lecture 9: Sorting II: Heaps

Lecture Overview

- Priority Queues
- Heaps
- Heapsort

Readings

CLRS 2.1, 2.2, 2.3, 6.1, 6.2, 6.3 and 6.4

Priority Queues

This is an abstract datatype implementing a set $S$ of elements, each associated with a key. Supports the following operations:

- $\text{insert}(S, x)$: insert element $x$ into set $S$
- $\text{max}(S)$: return element of $S$ with largest key
- $\text{extract}_\text{max}(S)$: return element of $S$ with largest key and remove it from $S$
- $\text{increase}_\text{key}(S, x, k)$: increases the value of element $x$’s key to new value $k$ (assumed to be as large as current value)

Heaps

An implementation of a priority queue. It is an array object, visualized as a nearly complete binary tree.

Heap Property: The key of a node is $\geq$ than the keys of its children; e.g., Figure 1.

![Binary Heap](image)

Figure 1: Binary Heap

NOTE: For convenience, the first index in the array is 1.
Visualizing an Array as a Tree

root of tree: first element in the array—corresponding index = 1,

node with index i:

parent(i) = ⌊i/2⌋; returns index of node’s parent, e.g. parent(5)=2

left(i) = 2i; returns index of node’s left child, e.g. left(4)=8

right(i) = 2i + 1; returns index of node’s right child, e.g. right(4)=9

Note: no pointers required! Height of a binary heap $O(\log_2 n)$.

Heap-Size Variable

For flexibility we may only need to consider the first few elements of an array as part of the heap. The variable heap-size denotes the number of items of the array that are part of the heap: $A[1], \ldots, A[\text{heap-size}]$;

Max-Heaps vs Min-Heaps

Max Heaps satisfy the Max-Heap Property: for all $i$, $A[i] \geq \max\{A[\text{left}(i)], A[\text{right}(i)]\}$. If left($i$) or right($i$) is undefined, replace $A[\text{left}(i)]$, respectively $A[\text{right}(i)]$, by $-\infty$. In particular, if node $i$ has no children, the property is trivially satisfied.

Everything we describe applies to the construction and operation of Min Heaps, satisfying the Min-Heap Property: for all $i$, $A[i] \leq \min\{A[\text{left}(i)], A[\text{right}(i)]\}$. If left($i$) or right($i$) is undefined, replace $A[\text{left}(i)]$, respectively $A[\text{right}(i)]$, by $+\infty$. In particular, if node $i$ has no children, the property is trivially satisfied.

Operations with Heaps

build_max_heap: produce a max-heap from unordered input array in $O(n)$;

max_heapify: correct a single violation of the heap property at the root of a subtree in $O(\log n)$;

heapsort: sort an array of size $n$ in $O(n \log n)$ via the use of heaps;

insert, extract_max: $O(\lg n)$

Max_Heapify(A,i)

Assume that the trees rooted at left($i$) and right($i$) are max-heaps. If element $A[i]$ violates the max-heap property, correct violation by trickling element $A[i]$ down the tree, making the subtree rooted at index $i$ a max-heap. See Figure 2 then read the pseudocode below.

\[
\begin{align*}
  l & \leftarrow \text{left}(i) \\
  r & \leftarrow \text{right}(i)
\end{align*}
\]
if \( l \leq \text{heap-size}(A) \) and \( A[l] > A[i] \)
    then largest \( \leftarrow l \)
    else largest \( \leftarrow i \)
if \( r \leq \text{heap-size}(A) \) and \( A[r] > A[\text{largest}] \)
    then largest \( \leftarrow r \)
if largest \( \neq i \)
    then exchange \( A[i] \) and \( A[\text{largest}] \)
MAX_HEAPIFY(A, largest)

Example

**Build_MAX_Heap(A)**

\( A[1 \cdots n] \) converted to a max_heap

*Observation:* Elements \( A[\lceil n/2 \rceil + 1 \cdots n] \) are all leaves of the tree (why? \( 2i > n \), for \( i > \lfloor n/2 \rfloor + 1 \)).

\[
\text{Build_MAX_Heap(A):} \\
\text{heap_size(A) = length(A)} \\
O(n) \text{ times for } i \leftarrow \lfloor \text{length}[A]/2 \rfloor \text{ downto 1} \\
O(\log n) \text{ time do } \text{Max_Heapify(A, i)} \\
O(n \log n) \text{ overall}
\]

See Figure \[3\] for an example.

**NOTE:** The trivial analysis of the algorithm noted above, shows that the running time is \( O(n \log n) \). Observe, however, that Max_Heapify only takes \( O(1) \) time for the nodes that are one level above the leaves, and in general \( O(\ell) \) for the nodes that are \( \ell \) levels above the leaves. Using this observation, it can be shown that the overall time for Build_MAX_Heap(A) is \( O(n) \).
MAX_HEAPIFY (A,2)
heap_size[A] = 10

Call MAX_HEAPIFY(A,4)
because max_heap property is violated

No more calls

Figure 2: MAX_HEAPIFY Example
MAX-HEAPIFY (A, 5)  
no change  
MAX-HEAPIFY (A, 4)  

MAX-HEAPIFY (A, 3)  

MAX-HEAPIFY (A, 2)  

MAX-HEAPIFY (A, 1)  

Figure 3: Example: Building Heaps
Sorting Strategy

• Build max_heap from unordered array

• Find maximum element (A[1])

• Swap elements A[n] and A[1]; now max element is at the right position;

• Discard node n from heap (decrement heap-size variable);

• New root could violate max_heap property, but children remain max_heaps. Run max_heapify to fix this;

Heap Sort Algorithm

\[
\begin{align*}
O(n) & \quad \text{Build}_\text{Max}_\text{Heap}(A): \\
n \text{times} & \quad \text{for } i = \text{length}[A] \text{ downto } 2 \\
 & \quad \text{do exchange } A[1] \leftrightarrow A[i] \\
 & \quad \text{heap_size}[A] = \text{heap_size}[A] - 1 \\
O(\log n) & \quad \text{MAX}_\text{HEAPIFY}(A, 1) \\
O(n \log n) & \quad \text{overall}
\end{align*}
\]

See Figure 4 for an illustration.
heap_size = 9
MAX.HEAPIFY (A,1)

Note: cannot run MAX.HEAPIFY with heapsize of 10

Figure 4: Illustration: Heap Sort Algorithm