Heap Sort
Properties

• Running time = $O(n \ lg \ n)$
• In place sorting
• Uses a data structure called a “heap”
Heaps

• Nearly complete binary tree
• Can be represented as a tree/array
• Two-types of heaps
  – MAX-HEAPS
  – MIN-HEAPS
Heap Property

• **Max-Heap Property**
  For every node other than the root
  \[ A[\text{PARENT}(i)] \geq A[i] \]
  Largest element in the heap is the root

• **Min-Heap Property**
  \[ A[\text{PARENT}(i)] \leq A[i] \]
  Smallest element in the heap is the root
Figure 6.1  A max-heap viewed as (a) a binary tree and (b) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.
Representation

• In an array representation, computing child and parent locations are very efficient

PARENT(i)
    return \lfloor \frac{i}{2} \rfloor

LEFT(i)          RIGHT(i)
    return 2i       return 2i + 1

• **Height** – longest path from root to leaf
Use

• Max Heaps – Heap Sort
• Min Heaps – Priority Queue’s

• Before looking at Heap Sort we will look at some operations on a heap
Maintaining the heap property

• MAX-HEAPIFY(A, i)
  – A is the array
  – i is an index in the array

• Assumption is that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps but A[i] may be smaller than its children

• This violates the heap property and we want to rearrange the elements to reinforce a heap
**MAX-HEAPIFY** \((A, i, n)\)

\[ l = \text{LEFT}(i) \]

\[ r = \text{RIGHT}(i) \]

**if** \( l \leq n \) and \( A[l] > A[i] \)

\[ \text{largest} = l \]

**else** \( \text{largest} = i \)

**if** \( r \leq n \) and \( A[r] > A[\text{largest}] \)

\[ \text{largest} = r \]

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

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

**MAX-HEAPIFY** \((A, \text{largest}, n)\)
Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $heap-size[A] = 10$. (a) The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in (b) by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in (c), node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.
Complexity of MAX-HEAPIFY

- As an index $i$, MAX-HEAPIFY can take as long as the largest subtree of $i$
- Depends on height $O(h)$
- W.C. Root is wrong, $O(lg n)$
Building a Heap

- Given an arbitrary array $A[1..n]$ create a heap
- Run MAX-HEAPIFY in a bottom up manner
- Leaves are already 1-element heaps
- Leaves – $A[(\lceil n/2 \rceil +1) \ldots n]$
**Build-Max-Heap** \((A, n)\)

for \(i = \lfloor n/2 \rfloor\) downto 1

**Max-Heapify** \((A, i, n)\)
Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. (a) A 10-element input array $A$ and the binary tree it represents. The figure shows that the loop index $i$ refers to node 5 before the call MAX-HEAPIFY($A, i$). (b) The data structure that results. The loop index $i$ for the next iteration refers to node 4. (c)-(e) Subsequent iterations of the for loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. (f) The max-heap after BUILD-MAX-HEAP finishes.
Complexity of Building a Heap

- Intuition
- Each call to MAX-HEAPIFY cost $O(lg \ n)$
- $O(n)$ such calls, running time $O(n \ lg \ n)$
- But this is not a tight bound!
Heap Sort

- Root (A[1]) of a Max Heap is largest element
- We can put it in its correct position by swapping it with A[n]
- But this destroys the Heap property
- MAX-HEAPIFY(A,1) restores this
- Repeat from n-1 to a heap size of 1
HEAPSORT\((A, n)\)

BUILD-MAX-HEAP\((A, n)\)

for \(i = n\) downto 2

exchange \(A[1]\) with \(A[i]\)

MAX-HEAPIFY\((A, 1, i - 1)\)
Figure 6.4  The operation of HEAPSORT. (a) The max-heap data structure just after it has been built by BUILD-MAX-HEAP. (b)–(j) The max-heap just after each call of MAX-HEAPIFY in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. (k) The resulting sorted array A.
Complexity of Heap Sort

- BUILD-HEAP – $O(n)$
- $n-1$ calls to MAX-HEAPIFY each taking $O(lg n)$
- Total $O(n \ lg n)$
Priority Queues

- Quick Sort usually defeats Heap Sort
- But a Heap is a very useful data structure
- A big use of Heaps – Efficient Priority Queues
- 2 types
  - Max-priority Queue
  - Min-priority Queue
Max-Priority Queues

- INSERT(S, x) inserts the element x into set S.
  \[ S \leftarrow S \cup \{x\} \]
- MAXIMUM(S) returns max key of S
- EXTRACT-MAX(S) removes and returns max key of S
- INCREASE-KEY(S, x, k) increases element x’s key to k where k >= x
- Similar operations for Min-Priority Queues
HEAP-MAXIMUM (A)

- $\Theta(1)$

```plaintext
HEAP-MAXIMUM(A)
1  return A[1]
```
HEAP-EXTRACT-MAX(A)

- \(O(lg \ n)\)

HEAP-EXTRACT-MAX(A, n)

\[
\text{if } n < 1 \\
\quad \text{error "heap underflow"}
\]

\[
max = A[1] \\
\text{MAX-HEAPIFY}(A, 1, n - 1) \quad \text{// remakes heap}
\]

return \(max\)
HEAP-INCREASE-KEY(A, i, key)

- Traverse from node to root to find the right place – O(lg n)

**HEAP-INCREASE-KEY (A, i, key)**

if \( key < A[i] \)

\[ \text{error "new key is smaller than current key"} \]

\( A[i] = key \)

while \( i > 1 \) and \( A[\text{PARENT}(i)] < A[i] \)

exchange \( A[i] \) with \( A[\text{PARENT}(i)] \)

\( i = \text{PARENT}(i) \)
Figure 6.5  The operation of \textsc{Heap-Increase-Key}. (a) The max-heap of Figure 6.4(a) with a node whose index is $i$ heavily shaded.  (b) This node has its key increased to 15.  (c) After one iteration of the \textbf{while} loop of lines 4–6, the node and its parent have exchanged keys, and the index $i$ moves up to the parent.  (d) The max-heap after one more iteration of the \textbf{while} loop.  At this point, $A[ \text{PARENT}(i)] \geq A[i]$.  The max-heap property now holds and the procedure terminates.
MAX-HEAP-INSERT(A, key)

• Expand max heap by adding a leaf (-infinity)
• Then call HEAP-INCREASE-KEY
• O(lg n)

MAX-HEAP-INSERT(A, key, n)

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

HEAP-INCREASE-KEY(A, n + 1, key)