Sorting

Dan Barrish-Flood
The Sorting Landscape

• $\Theta(n^2)$ sorts
  - insertion sort; insert each element into its proper place
  - selection sort; select the smallest element and remove it
  - bubble sort; swap adjacent out-of-order elements

• Key:
  - already discussed
  - won’t be discussed
  - will be discussed
The Sorting Landscape (cont’d)

• \( \Theta(n \lg n) \) sorts
  ➢ merge sort; fast, but not in-place
  ❖ heapsort
  ❖ quicksort

• **Key:**
  ➢ already discussed
    ▪ won’t be discussed
  ❖ will be discussed
The Sorting Landscape (cont’d)

- $\Theta(n)$ sorts
  - counting sort
  - radix sort
  - bucket sort

- **Key:**
  - already discussed
  - won’t be discussed
  - will be discussed
Binary Trees – a tree where every node has at most two children.
**Depth** of a node: length of path from root to that node. The numbers here show the depth.

**Height** of a tree: the largest depth of any node in the tree; equivalently, the longest path from the root to a leaf. The above tree has height 3.
Examples of binary trees

the complete binary tree of height two

the only binary tree of height 0.
Facts about Complete Binary Trees

\[ \text{DEPTH} \]

\[ \begin{array}{c}
0 \\
1 \\
2 \\
\vdots \\
h
\end{array} \]

\[ \text{# OF NODES} \]

\[ \begin{array}{c}
1 \\
2 \\
4 \\
\vdots \\
2^h \\
\sum_{i=0}^{h} 2^i = 2^{h+1} - 1
\end{array} \]
Facts about Complete Binary Trees (cont’d)

• In a complete binary tree, there are $2^d$ nodes at depth $d$ (thus, $2^h$ leaves).
• There are $2^{h+1} - 1$ nodes in a complete binary tree of height $h$.
• The height of a complete binary tree of $n$ nodes is $\text{floor}(\log n)$. 
Heaps

- a *heap* is an “almost” complete binary tree (extra nodes go from left to right at the lowest level), where

- the value at each node is $\geq$ the values at its children (if any) --- the *heap property*.
Heap Q & A

1. What is the height of a heap with n nodes?
2. How quickly can you find the maximum value in a heap?
3. How quickly can you extract (find and remove, maintaining the heap property) the maximum value from a heap?
HEAP_EXTRACT_MAX
HEAP_EXTRACT_MAX

1. remove and save the root (max) of the heap
2. replace it with the last leaf (the heap is one node smaller now)
3. “trickle down” the new root until the heap property is restored.
4. return the max
HEAP_EXTRACT_MAX code

HEAP-EXTRACT-MAX (A)

1     if heap-size[A] < 1
2         then error "heap underflow"
3     max ← A[1]
5     heap-size[A] ← heap-size[A] − 1
6     MAX-HEAPIFY (A, 1)
7     return max
What is “trickle down”?

- The “trickle down” procedure is called `MAX_HEAPIFY`
- Q. How long does it take?
- Q. Does `HEAP_EXTRACT_MAX` suggest a sorting algorithm?
HEAPSORT

1. Build a heap from the input list
2. Repeatedly perform EXTRACT_MAX until there remains only one item in the heap (or until the heap is empty, possibly).

• **Q.** How do you *build* a heap?
BUILD_MAX_HEAP

1. Put the items into an almost complete binary tree.
2. Call MAX_HEAPIFY on each node of the tree.
   • we must start at the bottom and go up
   • we can skip the leaves
BUILD_MAX_HEAP
BUILD_MAX_HEAP: running time:

• **Q.** What is an easy upper bound on the running time of BUILD_MAX_HEAP?

• **A.** Each call to MAX_HEAPIFY costs $O(lgn)$ time, and there are $O(n)$ such calls. So we arrive at a correct (but not asymptotically tight) upper bound of $O(nlgn)$.

• *But we can get a tighter bound...*
BUILD_MAX_HEAP: getting a tighter bound

These HEAPIFIES run on trees of height 1
BUILD_MAX_HEAP: getting a tighter bound

• How many nodes of height $h$ can there be in a heap of $n$ nodes?

• Consider a complete binary tree:
  – There are $\text{ceiling}(n/2)$ leaves (height $= 0$)
  – There $\frac{1}{2}$ that many nodes at height $= 1$
  – In general, there are this many nodes of height $h$:

$$
\left\lfloor \frac{n}{2^{h+1}} \right\rfloor
$$

adding more leaves doesn’t increase that.
**BUILD_MAX_HEAP running time**

\[
\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{h+1}} \Theta(h) = \Theta(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h})
\]

Recall:

\[
\sum_{i=0}^{\infty} ir^i = \frac{r}{(1-r)^2}, \quad |r| < 1
\]

Here, \( r = 1/2 \), so

\[
\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} < \left( \frac{1/2}{1-1/2} \right)^2 = 2
\]

Thus the running time is \( \Theta(n) \)
HEAPSORT: running time

1. BUILD_MAX_HEAP = \( \Theta(n) \)
2. \( n \times \text{HEAP_EXTRACT_MAX} \)
   = \( n \times \Theta(\lg n) \)
   = \( \Theta(n \lg n) \)
\( \Theta(n) + \Theta(n \lg n) = \Theta(n \lg n) \)

No better than mergesort – but heapsort is done in-place.

It is not necessary to build a binary tree data structure (cool)!
a HEAP is an ARRAY

Observe:
- children of $i$ are $2i$ and $2i+1$
- parent of $i$ is $\text{floor}(i/2)$
- so pointers are not needed!
code for HEAPSORT

**BUILD-MAX-HEAP(A)**

1. \(heap-size[A] \leftarrow \text{length}[A]\)
2. for \(i \leftarrow \left\lfloor \text{length}[A]/2 \right\rfloor \) downto 1
3. do MAX-HEAPIFY\((A, i)\)

**HEAPSORT(A)**

1. BUILD-MAX-HEAP\((A)\)
2. for \(i \leftarrow \text{length}[A] \) downto 2
3. do exchange \(A[1] \leftrightarrow A[i]\)
4. \(heap-size[A] \leftarrow heap-size[A] - 1\)
5. MAX-HEAPIFY\((A, 1)\)

**MAX-HEAPIFY\((A, i)\)**

1. \(l \leftarrow \text{LEFT}(i)\)
2. \(r \leftarrow \text{RIGHT}(i)\)
3. if \(l \leq heap-size[A] \) and \(A[l] > A[i]\) then \(largest \leftarrow l\) else \(largest \leftarrow i\)
4. if \(r \leq heap-size[A] \) and \(A[r] > A[largest]\) then \(largest \leftarrow r\)
5. if \(largest \neq i\) then exchange \(A[i] \leftrightarrow A[largest]\)
6. MAX-HEAPIFY\((A, largest)\)

**PARENT\((i)\)**

7. return \(\left\lfloor i/2 \right\rfloor\)

**LEFT\((i)\)**

8. return \(2i\)

**RIGHT\((i)\)**

9. return \(2i + 1\)
Using a heap as a *priority queue*

- Another application of heaps, as long as we’re discussing heaps...
- A *priority queue* is a data structure supporting the following operations:
  - **INSERT**(x) : add x to the queue
  - **MAXIMUM** : return the largest item (i.e. item w/largest key)
  - **EXTRACT_MAX** : remove and return the largest item
Priority Queue: typical applications

• **Process scheduling**: some processes are of higher priority than others.

• **Event-driven simulation**: select the event that will happen next; need to invert the heap to support MINIMUM and EXTRACT_MIN (i.e. a node’s key \( \leq \) its children’s keys, instead of \( \geq \)).

• **Best-first search**: explore the best alternative found so far.
About priority queues

• We know how to do \texttt{MAXIMUM} and \texttt{EXTRACT\_MAX}. What about \texttt{INSERT}?

• \texttt{INSERT} Idea: put the new element at bottom of the heap (i.e. the new last leaf), and “\textit{trickle it up}”, til the heap property is restored.
priority queue: **INSERT** running time = $O(\log n)$

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

1. if $key < A[i]$
2. then error “new key is smaller than current key”
3. $A[i] \leftarrow key$
4. while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5. do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
6. $i \leftarrow \text{PARENT}(i)$

**MAX-HEAP-INSERT** $(A, key)$

1. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] + 1$
2. $A[\text{heap-size}[A]] \leftarrow -\infty$
3. **HEAP-INCREASE-KEY** $(A, \text{heap-size}[A], key)$
Two fine points about sorting

1. The **keys** we are sorting may have arbitrarily large **records** associated with them.
   - Use pointers (when possible) to avoid copying this satellite data.

2. Our running times have been in terms of **number of comparisons**.
   - We assume the time for comparison is constant. When (i.e. *for what type of key*) is this false?