Announcements

- Prelab 7 is due now.
- Lab 7 due on Sunday night.
- CS department winter term organizational meeting: Tuesday Nov 6, 4:30pm in King 237

Monday, November 5

Review of PQ's and Heaps

A priority queue stores a collection of comparable elements such that the highest priority element is "at the front". Operations are:

- insert(element) aka offer --- add element to the queue
- removeMin() aka poll --- remove the highest priority element
- findMin() aka peek --- return but do not remove the highest priority element
- size()
- isEmpty()
- traverse() / iterate()

We implement the priority queue using a binary heap:

- a complete binary tree (a binary tree where each level is completely filled except for possibly the last, which is filled from left-to-right.)
- with the heap order property:
  - for all nodes u, the value/key of u is <= value/key of its children
  - except for possibly the last, which is filled from left-to-right
  - a complete binary tree (a binary tree whose level is completely filled)

We implement the priority queue using a binary heap:

- insert()
- removeMin()
- findMin()
- remove()
- adjust()

We showed that size() and remove() run in O(1), and insert runs O(log n).

The nodes of a heap are stored in an array in level-order, starting at index 1.

- for all nodes u, value/key of u is <= value/key of its children
- except for possibly the last, which is filled from left-to-right

We implement the priority queue using a binary heap:

- insert()
- removeMin()
- findMin()
- remove()
- adjust()

RT: if percolate down to the bottom, get O(log n).

Note that percolate(index) is actually O(height of the tree rooted at index).

PercolateDown(index) -- move the elt at index down to the correct spot

Review of PercolateDown

- if index is a leaf, return
- set minChild = smaller of index's children (or, child)
- if data[minChild] < data[index] then we're out of heap order
- swap the two entries
- set index = minChild
- return

The nodes of a heap are stored in an array in level-order, starting at index 1.

- for all nodes u, value/key of u is <= value/key of its children
- except for possibly the last, which is filled from left-to-right

We implement the priority queue using a binary heap:

- insert()
- removeMin()
- findMin()
- remove()
- adjust()

RT: if percolate down to the bottom, get O(log n).

Note that percolate(index) is actually O(height of the tree rooted at index).

PercolateDown(index) -- move the elt at index down to the correct spot

Review of PercolateDown

- if index is a leaf, return
- set minChild = smaller of index's children (or, child)
- if data[minChild] < data[index] then we're out of heap order
- swap the two entries
- set index = minChild
- return

The nodes of a heap are stored in an array in level-order, starting at index 1.

- for all nodes u, value/key of u is <= value/key of its children
- except for possibly the last, which is filled from left-to-right

We implement the priority queue using a binary heap:

- insert()
- removeMin()
- findMin()
- remove()
- adjust()
Consider trying to construct a heap out of an unordered array.

heapify( array ) -- make array into a heap (enforce the heap order property)

How could you do this?

Idea #1:
- Construct a new empty temporary array, view it as a heap.
- For each element in the given array, add it to our new heap.
- At the end, our new heap contains all the elements, but with heap order.

Problem is... takes $O(n \log n)$ time. (Each of the $n$ inserts takes $O(\log n)$ time.)

Want a way to heapify in $O(n)$ time.

Idea #2:
- View the given array as a complete binary tree without the heap order property.
- Start at the leaves, move up the tree level-by-level.
- Call percolateDown(i) for each node in the tree.

Since percolateDown(i) takes $O(\text{height}(i))$, the runtime of heapify is:

\[
O(n \log n) = \sum_{\text{nodes } \in T} O(\text{height}(u)) = O(n)
\]

(we'll prove this by induction later)

Heapsort

Note that heapify gives us a new sorting algorithm, called heapsort:

heapsort(array):
- Call heapify(array).
- RemoveMin $n$ times to repeatedly get the next minimum element.

Heapsort is $O(n \log n)$, as expected. (Each of the $n$ removeMins are $O(\log n)$.)

Heapsort is easier to implement than quicksort and mergesort.

Structural Induction Example

A complete binary tree $T$ is either:
- An empty tree, or
- A root node $r$ with left and right complete binary trees $L$ and $R$ such that:
  - If $L$ is not perfect (it's "missing" some leaves), then $\text{height}(L) = \text{height}(R) - 1$.
  - $r$ is a root node with both left and right complete binary trees $L$ and $R$.

A complete binary tree $T$ is either:

Some properties:
- A complete binary tree's height is $O(\log n)$.
- The number of nodes in a complete binary tree is $O(n)$.
- Each node in a complete binary tree has at most $2$ children.

We will show that $P(T)$ is true for all complete binary trees $T$. By structural induction.

For any complete binary tree $T$, $P(T)$ is the boolean property that:

\[
\sum_{\text{nodes } \in T} \text{height}(u) \in O(n)
\]

(we'll prove this by induction later)

We will show that $P(T)$ is true for all complete binary trees $T$. By structural induction.

For any complete binary tree $T$, $P(T)$ is the boolean property that:

\[
\sum_{\text{nodes } \in T} \text{height}(u) \in O(n)
\]

(we'll prove this by induction later)

For any complete binary tree $T$, $P(T)$ is the boolean property that:

\[
\sum_{\text{nodes } \in T} \text{height}(u) \in O(n)
\]

(we'll prove this by induction later)

For any complete binary tree $T$, $P(T)$ is the boolean property that:

\[
\sum_{\text{nodes } \in T} \text{height}(u) \in O(n)
\]

(we'll prove this by induction later)

For any complete binary tree $T$, $P(T)$ is the boolean property that:

\[
\sum_{\text{nodes } \in T} \text{height}(u) \in O(n)
\]

(we'll prove this by induction later)
A complete binary tree T is either
• an empty tree, or
• a root node r with left and right complete binary trees L and R such that if L is not perfect (it’s “missing” some leaves) then \( h(R) = h(L) - 1 \) (o.w. \( h(R) = h(L) \)).

For any complete binary tree T, let \( P(T) \) be the boolean property
\[
\sum_{v \in T} h(v) \in O(n(T))
\]
where \( h \) is the height of node v.

Step 1: Show that \( P(T) \) is true when T is your base case.

Base case: T is an empty tree
We want to show (WTS) that \( P(T) \) is true when T is an empty tree, i.e. that
\[
\sum_{v \in T} h(v) \in O(n(T))
\]
when T is empty, there are no nodes in T so \( n(T) = 0 \) and the sum is empty:

\[
\sum_{\text{nodes} \in T} h(v) = 0
\]
because \( n(T) = 0 \).

Step 2: Show that \( P(T) \) is true when T is your recursive case.

This is our induction hypothesis
We want to use these two hypothetical assumptions and the definition of T to show \( P(T) \) is true, i.e. that
\[
\sum_{v \in T} h(v) \in O(n(T))
\]
recursive case: T is a root node r with L and R complete binary trees. Suppose hypothetically, that \( P(L) \) and \( P(R) \) are true, i.e. that
\[
\sum_{v \in L} h(v) \in O(n(L)) \quad \text{and} \quad \sum_{v \in R} h(v) \in O(n(R))
\]
We want to show (WTS) that \( P(T) \) is true when T is a complete binary tree, i.e. that
\[
\sum_{v \in T} h(v) \in O(n(T))
\]
because T's nodes are r, L, & R

Use these facts along with our IH
\[
\sum_{v \in T} h(v) = h(r) + \sum_{v \in L} h(v) + \sum_{v \in R} h(v)
\]
b/c T's nodes are r, L, & R

\[
= h(r) + O(n(L)) + O(n(R))
\]
by the induction hypothesis

\[
= h(r) + O(n(T))
\]
b/c \( n(L) + n(R) = n(T) \)

Thus \( P(T) \) is true for all T by structural induction on T.