Binary Heaps

CSE 373 - Data Structures
April 26, 2002
Readings and References

• Reading
  › Sections 6.1-6.4, *Data Structures and Algorithm Analysis in C*, Weiss

• Other References
A New Problem…

• Application: Find the smallest (or highest priority) item quickly
  › Operating system needs to schedule jobs according to priority
  › Doctors in ER take patients according to severity of injuries
  › Event simulation (bank customers arriving and departing, ordered according to when the event happened)
Use Lists or Binary Search Tree?

• We want an ADT that can efficiently perform:
  › FindMin (and DeleteMin)
  › Insert

• What if we use…
  › Lists: If sorted, what is the run time for Insert and FindMin? Unsorted?
  › Binary Search Trees: What is the run time for Insert and FindMin?
Less flexibility → More speed

- Lists
  - If sorted: FindMin is $O(1)$ but Insert is $O(N)$
  - If not sorted: Insert is $O(1)$ but FindMin is $O(N)$

- Binary Search Trees (BSTs)
  - Insert is $O(\log N)$ and FindMin is $O(\log N)$

- BSTs look good but…
  - BSTs are efficient for all Finds, not just FindMin
  - We only need FindMin
Better than a speeding BST

- We can do better than Binary Search Trees
  - Very limited requirements: Insert, FindMin, DeleteMin
  - FindMin is $O(1)$
  - Insert is $O(\log N)$
  - DeleteMin is $O(\log N)$
Binary Heaps

• A binary heap is a binary tree that is:
  › Complete: the tree is completely filled except possibly the bottom level, which is filled from left to right
  › Satisfies the heap order property
    • every node is less than or equal to its children
    • or every node is greater than or equal to its children

• The root node is always the smallest node
  › or the largest, depending on the heap order
Heap order property

- A heap provides limited ordering information
- Each *path* is sorted, but the subtrees are not sorted relative to each other
  - A binary heap is NOT a binary search tree

These are all valid binary heaps (minimum)
Binary Heap vs Binary Search Tree

**Binary Heap**

- Parent is greater than left child, less than right child
- min value

**Binary Search Tree**

- Parent is less than both left and right children
- min value

Parent is greater than left child, less than right child
Structure property

• A binary heap is a complete tree
  › All nodes are in use except for possibly the right end of the bottom row

• Pointers from node to node?
  › allow arbitrary connect and disconnect at any node
  › but we don't need this flexibility since the tree is always complete and we don't need to do a lot of reorganizing to meet a tree order property
Examples

- Complete tree, heap order is "max"

- Complete tree, heap order is "min"

- Not complete

- Complete tree, but min heap order is broken
Array Implementation of Heaps

- Root node = A[1]
- Keep track of current size N (number of nodes)
FindMin and DeleteMin

• **FindMin**: Easy!
  › Return root value A[1]
  › Run time = ?

• **DeleteMin**:
  › Delete (and return) value at root node
DeleteMin

- Delete (and return) value at root node
Maintain the Structure Property

- We now have a “Hole” at the root
  - Need to fill the hole with another value
- When we get done, the tree will have one less node and must still be complete
Maintain the Heap Property

• The last value has lost its node
  › we need to find a new place for it
• We can do a simple insertion sort operation to find the correct place for it in the tree
DeleteMin: Percolate Down

- Copy smaller child up and go down one level.
- Done if both children are $\geq$ item or reached a leaf node.
- What is the run time?
DeleteMin: Run Time Analysis

• Run time is $O(\text{depth of heap})$

• A heap is a complete binary tree

• Depth of a complete binary tree of $N$ nodes?
  
  depth = $\left\lfloor \log(N) \right\rfloor = \text{floor}(\log(N))$

• Run time of DeleteMin is $O(\log N)$
Insert

- Add a value to the tree
- Structure and heap order properties must still be correct when we are done
Maintain the Structure Property

- The only valid place for a new node in a complete tree is at the end of the array.
- We need to decide on the correct value for the new node, and adjust the heap accordingly.
Maintain the Heap Property

- The new value goes where?
- We can do a simple insertion sort operation to find the correct place for it in the tree.
Insert: Percolate Up

- Start at last node and keep comparing with parent \(A[i/2]\)
- If parent larger, copy parent down and go up one level
- Done if parent \(\leq\) item or reached top node \(A[1]\)
- Run time?
Insert: Done

- Run time?
Sentinel Values

- Every iteration of Insert needs to test:
  - if it has reached the top node $A[1]$
  - if parent $\leq$ item
- Can avoid first test if $A[0]$ contains a very large negative value
  - sentinel $-\infty <$ item, for all items
- Second test alone always stops at top

<table>
<thead>
<tr>
<th>value</th>
<th>$-\infty$</th>
<th>2</th>
<th>3</th>
<th>8</th>
<th>7</th>
<th>4</th>
<th>10</th>
<th>9</th>
<th>11</th>
<th>9</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

26-Apr-02  CSE 373 - Data Structures - 11 - Binary Heaps
Summary of Heap ADT Analysis

• Space needed for heap of N nodes: $O(\text{MaxN})$
  › An array of size MaxN, plus a variable to store the size $N$, plus an array slot to hold the sentinel

• Time
  › FindMin: $O(1)$
  › DeleteMin and Insert: $O(\log N)$
  › BuildHeap from $N$ inputs
    • $N$ Insert operations = $O(N \log N)$
    • Treat input array as a heap and fix it using percolate down = $O(N)$
Other Heap Operations

- Find(X, H): Find the element X in heap H of N elements
  - What is the running time? O(N)
- FindMax(H): Find the maximum element in H
  - What is the running time? O(N)
- We sacrificed performance of these operations in order to get O(1) performance for FindMin
Other Heap Operations

- **DecreaseKey(P, Δ, H):** Decrease the key value of node at position P by a positive amount Δ. eg, to increase priority
  - First, subtract Δ from current value at P
  - Heap order property may be violated
  - so percolate up to fix
  - Running Time: O(log N)
Other Heap Operations

- **IncreaseKey(P, Δ, H):** Increase the key value of node at position P by a positive amount Δ. e.g., to decrease priority
  - First, add Δ to current value at P
  - Heap order property may be violated
  - so percolate down to fix
  - Running Time: O(log N)
Other Heap Operations

- **Delete(P,H):** E.g. Delete a job waiting in queue that has been preemptively terminated by user
  - Use DecreaseKey(P,∞,H) followed by DeleteMin
  - Be careful about your sentinel value and overflow
  - Running Time: O(log N)
Other Heap Operations

- **Merge(H1,H2):** Merge two heaps H1 and H2 of size $O(N)$. H1 and H2 are stored in two arrays.
  - Can do $O(N)$ Insert operations: $O(N \log N)$ time
  - Better: Copy H2 at the end of H1 and use BuildHeap. Running Time: $O(N)$
- **Merges in $O(\log N)$ coming soon to a lecture hall near you ...**