Announcements

- Project 1 – phase A due Wed Jan 12th 11pm via catalyst
- Homework 1 – due Friday Jan 14th at beginning of class
- Re-organization on course web page
- Info sheets?

Today

- Finish up Asymptotic Analysis
- New ADT: Priority Queues

A new ADT: Priority Queue

- Textbook Chapter 6
  - We will go back to binary search trees (ch4) and hash tables (ch5) later
  - Nice to see a new and surprising data structure first
- A priority queue holds comparable data
  - Unlike stacks and queues need to compare items
    - Given x and y, is x less than, equal to, or greater than y
    - What this means can depend on your data
    - Much of course will require comparable data: e.g. sorting
    - Integers are comparable, so will use them in examples
    - But the priority queue ADT is much more general

Priority Queue ADT

- Assume each item has a “priority”
  - The lesser item is the one with the greater priority
  - So “priority 1” is more important than “priority 4”
  - (Just a convention)
- Operations:
  - insert
  - deleteMin
  - create, is_empty, destroy
- Key property: deleteMin returns and deletes from the queue the item with greatest priority (lowest priority value)
  - Can resolve ties arbitrarily

Focusing on the numbers

- For simplicity in lecture, we’ll often suppose items are just int and the int is the priority
  - The same concepts without generic usefulness
  - So an operation sequence could be
    - insert 6
    - insert 5
    - x = deleteMin
  - Integers are common, but really just need comparable
  - Not having “other data” is very rare
  - Example: print job is a priority and the file
Example

- Insert x₁ with priority 5
- Insert x₂ with priority 3
- Insert x₃ with priority 4
- a = deleteMin
- b = deleteMin
- Insert x₄ with priority 2
- Insert x₅ with priority 6
- c = deleteMin
- d = deleteMin

- Analogy: insert is like enqueue, deleteMin is like dequeue
- But the whole point is to use priorities instead of FIFO

Applications

- Like all good ADTs, the priority queue arises often
  - Sometimes “directly”, sometimes less obvious
- Run multiple programs in the operating system
  - “Critical” before “interactive” before “compute-intensive”
  - Maybe let users set priority level
- Treat hospital patients in order of severity (or triage)
- Select print jobs in order of decreasing length?
- Forward network packets in order of urgency
- Select most frequent symbols for data compression (cf. CSE143)
- Sort insert all, then repeatedly deleteMin
  - Much like Project 1 uses a stack to implement reverse

More applications

- “Greedy” algorithms
  - Select the “best-looking” choice at the moment
  - Will see an example when we study graphs in a few weeks
- Discrete event simulation (system modeling, virtual worlds, …)
  - Simulate how state changes when events fire
  - Each event e happens at some time t and generates new
    events e₁, …, eₙ at times t₁, …, tₙ
  - Naive approach: advance “clock” by 1 unit at a time and
    process any events that happen then
  - Better:
    - Pending events in a priority queue (priority = time happens)
    - Repeatedly deleteMin and then insert new events
    - Effectively, “set clock ahead to next event”

Implementations of Priority Queue ADT

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Unsorted Linked-List</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted Circular Array</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Sorted Linked-List</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Binary Search Tree (BST)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

More on possibilities

- If priorities are random, binary search tree will likely do better
  - O(log n) insert and O(log n) deleteMin on average
- But we are about to see a data structure called a “binary heap”
  - O(log n) insert and O(log n) deleteMin worst-case
  - Very good constant factors
  - If items arrive in random order, then insert is O(1) on average
- One more idea: if priorities are 0, 1, …, k can use array of lists
  - insert: add to front of list at arr[priority]. O(1)
  - deleteMin: remove from lowest non-empty list O(k)

Need a good data structure!

- Will show an efficient, non-obvious data structure for this ADT
  - But first let’s analyze some “obvious” ideas for n data items
  - All times worst-case; assume arrays “have room”

<table>
<thead>
<tr>
<th>data</th>
<th>insert algorithm / time</th>
<th>deleteMin algorithm / time</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted array</td>
<td>add at end O(1)</td>
<td>search O(n)</td>
</tr>
<tr>
<td>unsorted linked list</td>
<td>add at front O(1)</td>
<td>search O(n)</td>
</tr>
<tr>
<td>sorted circular array</td>
<td>search / shift O(n)</td>
<td>move front O(1)</td>
</tr>
<tr>
<td>sorted linked list</td>
<td>put in right place O(n)</td>
<td>remove at front O(1)</td>
</tr>
<tr>
<td>binary search tree</td>
<td>put in right place O(n)</td>
<td>leftmost O(n)</td>
</tr>
</tbody>
</table>
Tree terms (review?)

The binary heap data structure implementing the priority queue ADT will be a tree, so worth establishing some terminology:

- root(tree)
- children(node)
- parent(node)
- leaves(tree)
- siblings(node)
- ancestors(node)
- descendants(node)
- subtree(node)
- depth(node)
- height(tree)
- degree(node)
- branching factor(tree)

Kinds of trees

Certain terms define trees with specific structure:

- Binary tree: Each node has at most 2 children
- N-ary tree: Each node has at most n children
- Complete tree: Each row is completely full except maybe the bottom row, which is filled from left to right

Teaser: Later we’ll learn a tree is a kind of directed graph with specific structure.

Our data structure

Finally, then, a binary min-heap (or just binary heap or just heap) is:

- A complete tree – the “structure property”
- For every (non-root) node the parent node’s value is less than the node’s value – the “heap order property” (not a binary search tree)

<table>
<thead>
<tr>
<th>Heap – Deletemin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Idea:</td>
</tr>
<tr>
<td>1. Remove root (that is always the min!)</td>
</tr>
<tr>
<td>2. Put “last” leaf node at root</td>
</tr>
<tr>
<td>3. Find smallest child of node</td>
</tr>
<tr>
<td>4. Swap node with its smallest child if needed</td>
</tr>
<tr>
<td>5. Repeat steps 3 &amp; 4 until no swaps needed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap – Insert(val)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Idea:</td>
</tr>
<tr>
<td>1. Put val at “next” leaf position</td>
</tr>
<tr>
<td>2. Repeatedly exchange node with its parent if needed</td>
</tr>
</tbody>
</table>

Heap Operations

- findMin:
- deleteMin: percolate down.
- insert(val): percolate up.
Operations: basic idea

- **findMin**: return root.data
- **deleteMin**:
  1. answer = root.data
  2. Move right-most node in last row to root to restore structure property
  3. “Percolate down” to restore heap property
- **insert**:
  1. Put new node in next position on bottom row to restore structure property
  2. “Percolate up” to restore heap property

DeleteMin

1. Delete (and return) value at root node

2. Restore the Structure Property

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

3. Restore the Heap Property

- Percolate down:
  - Keep comparing with both children
  - Move smaller child up and go down one level
  - Done if both children are ≥ item or reached a leaf node
  - What is the run time?

DeleteMin: Run Time Analysis

- Run time is $O$(height of heap)
- A heap is a complete binary tree
- Height of a complete binary tree of $n$ nodes?
  - height = $\lceil \log_2(n) \rceil$
- Run time of deleteMin is $O$(log $n$)

Insert

- Add a value to the tree
- Structure and heap order properties must still be correct afterwards
Insert: Maintain the Structure Property

- There is only one valid tree shape after we add one more node
- So put our new data there and then focus on restoring the heap property

Insert: Run Time Analysis

- Like deleteMin, worst-case time proportional to tree height – $O(\log n)$
- But... deleteMin needs the “last used” complete-tree position and insert needs the “next to use” complete-tree position
  - If “keep a reference to there” then insert and deleteMin have to adjust that reference: $O(\log n)$ in worst case
  - Could calculate how to find it in $O(\log n)$ from the root given the size of the heap
  - But it's not easy
  - And then insert is always $O(\log n)$, promised $O(1)$ on average (assuming random arrival of items)
- There’s a “trick”: don’t represent complete trees with explicit edges! (see in next lecture)