Priority Queues and Heaps

CS211
Fall 2000

ADT Priority Queue

- Operations:
  - boolean isEmpty();
  - void add (Object item);
  - Object removeFirst ();
- Other less-common operations:
  - update (an Item’s priority)
  - join two PQs to make one new PQ
  - delete (an Item)

- Uses
  - Job scheduler for OS
  - Can use to sort
  - Retain the best k items
  - Event-driven simulation
  - Wide use within other algorithms

Possible PQ Implementations

<table>
<thead>
<tr>
<th>Unordered List</th>
<th>Ordered List</th>
<th>Unordered Array</th>
<th>Ordered Array</th>
<th>BST</th>
<th>Balanced BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (item)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>removeFirst()</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

* BST becomes unbalanced as PQ is used

Can we do better than balanced trees? Well no, not in terms of big-O bounds, but...

Heaps

Definition: A min-heap is a complete binary tree in which the value at each node is ≤ the value of its children.

Definition: For a max-heap, each node is ≥ the value of its children.

Definition: Complete means that each level of the tree is filled except possibly the last, which is filled from left to right.

Add and RemoveFirst (for min-heap)

```
add (item):
    Place item in next empty position;
    while (item < parent) {
        Swap item with parent;
    }

removeFirst ():
    min = root.value;
    Swap root and last item in heap;
    Decrease heap size by 1;
    while (v > one of its children) {
        Swap v with its smallest child;
    }
    return min;
```

Heap Implementation (the Big Trick)

- Can avoid using pointers!
- Store the heap in an array
- For A[i]
  - left child = 2 × i
  - right child = 2 × i + 1
  - parent = ⌊i / 2⌋

A Min-Heap

```
40 51 6
```

To Build a Heap

- How long to construct a heap, given the items?
- Worst-case time for insert() is \( O(\log n) \)
- Total time to build heap using insert() is \( O(\log 1) + O(\log 2) + \ldots + O(\log n) \) or \( O(n \log n) \)

Can we do better?

We had two heap-fixing methods:
- bubbleUp: move up the tree as long as we're less than our parent
- bubbleDown: move down the tree as long as we're bigger than one of our children

If we build the heap from the bottom-up using bubbleDown then we can build it in time \( O(n) \) (Wow!)

Efficient Heap Building

- Build from the bottom-up
- If there are \( n \) items in the heap then...
  - There are about \( n/2 \) mini-heaps of height 1
  - There are about \( n/4 \) mini-heaps of height 2
  - There are about \( n/8 \) mini-heaps of height 3
  - and so on
  - The time to fix up a mini-heap is \( O(\text{its height}) \)

Total time spent fixing heaps is thus bounded by
\[
\frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \ldots
\]
This can be rewritten as
\[
n(1/2 + 2/4 + \ldots + i/2^i + \ldots) = n(2)
\]
Thus total heap-building time (using the bottom-up method) is \( O(n) \)

Other Heap Operations

- delete a particular item
- update an item (change its priority)
- join two priority queues

For delete and update, we need to be able to find the item
- One way to do this: Use a HashMap to keep track of the item's position in the heap
- Efficient joining of 2 Priority Queues requires another data structure
  - Skew Heaps or Pairing Heaps (Chapter 22 in text)

Another PQ Implementation

- If there are only a few possible priorities then can use an array of lists
  - Each array position represents a priority \((0..m-1\) where \( m \) is the array size)
  - Each list holds all items that have that priority (treated as a queue)
- One text [Skiena] calls this a bounded height priority queue

Time for add: \( O(1) \)
- Time for removeFirst:
  - \( O(m) \) in the worst-case
  - Generally, faster

PQ Application: Simulation

- Example: Given a probabilistic model of bank-customer arrival times and transaction times, how many tellers are needed
  - Assume we have a way to generate random inter-arrival times
  - Assume we have a way to generate transaction times
  - Can simulate the bank to get some idea of how long customers must wait

Time-Driven Simulation
- Check at each tick to see if any event occurs

Event-Driven Simulation
- Advance clock to next event, skipping intervening ticks
- This uses a PQ!