CSC 212
Data Structures and Abstractions
Spring 2016
Lecture 14: Priority Queues and Heaps

Administratvia

- Ranking (contest)
  - congrats!
- PA2
  - don’t have a group yet? we’ll assign you one tomorrow
  - radiusSearch on kd-trees on your own (ask questions!)
  - come prepared for the interview on April 6th
- 15 bonus points on final if you create a client-server application using
  sockets!
- Consider Linux

Balanced BSTs

<table>
<thead>
<tr>
<th></th>
<th>Sequential Search</th>
<th>Binary Search</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>insert</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>delete</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>min/max</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>floor/ceiling</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>rank</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n) **</td>
</tr>
</tbody>
</table>

** requires the use of 'size' at every node

Quiz

- How to remove data from an AVL tree?
- Can we sort using balanced trees? Cost?

Priority Queues

Applications

- Data Compression (huffman trees)
- Process Scheduling (CPUs)
- Graph Algorithms
- Network Routing
- Artificial Intelligence (search)
- Stream Data Algorithms
- HPC Task Scheduling

Priority Queues

Collections of <Key,Value> pairs

- keys are objects on which an order is defined
- Every pair of keys must be comparable according to a total order:
  - Reflexive Property: \( k \leq k \)
  - Antisymmetric Property: if \( k_1 \leq k_2 \) and \( k_2 \leq k_1 \), then \( k_1 = k_2 \)
  - Transitive Property: if \( k_1 \leq k_2 \) and \( k_2 \leq k_3 \), then \( k_1 \leq k_3 \)
Priority Queues

Queues
basic operations: enqueue, dequeue
always remove the item least recently added

Priority Queues (MaxPQ)
basic operations: insert, removeMax
always remove the item with highest (max) priority
Can also be implemented as a MinPQ

Priority Queues

Example (MinPQ)

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Value</th>
<th>Priority Queue Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(5,A)</td>
<td></td>
<td>{5,A}</td>
</tr>
<tr>
<td>insert(9,C)</td>
<td></td>
<td>{5,A, 9,C}</td>
</tr>
<tr>
<td>insert(3,B)</td>
<td></td>
<td>{3,B, 5,A, 9,C}</td>
</tr>
<tr>
<td>removeMin()</td>
<td></td>
<td>{3,B, 5,A, 9,C, 3,B}</td>
</tr>
<tr>
<td>removeMax()</td>
<td></td>
<td>{5,A, 9,C}</td>
</tr>
<tr>
<td>removeMin()</td>
<td></td>
<td>{9,C}</td>
</tr>
<tr>
<td>removeMin()</td>
<td></td>
<td>{}</td>
</tr>
<tr>
<td>min()</td>
<td></td>
<td>(3,B, 3,B, 9,C)</td>
</tr>
<tr>
<td>max()</td>
<td></td>
<td>(9,C)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td></td>
<td>true</td>
</tr>
</tbody>
</table>

Performance?

<table>
<thead>
<tr>
<th>Sorted Array/List</th>
<th>Unsorted Array/List</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>removeMax</td>
<td>O(1)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>max</td>
<td>O(1)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

Performance

// what does this function do?
//
void foo(int *array, int n) {
    std::priority_queue<int> pq;
    for (int i = 0 ; i < n ; i ++)
        pq.push(array[i]); // insert()
    while (! pq.empty()) {
        array[--n] = pq.top(); // max()
        pq.pop(); // removeMax()
    }
}

Heaps

Heaps

(max) Heap

Structure Property
a heap is a complete binary tree

Heap-Order Property
for every node x, key(parent(x)) >= key(x)
except the root, which has no parent

Height of a heap?

What is the minimum number of nodes in a complete binary tree of height $h$?

$n \geq \frac{2^h}{2} - 1$

$\log n \geq \log 2^h$

$\log n \geq h$
**Implementation**

- `parent(i)`
- `floor(i/2)`
- `left_child(i)`
- `i*2`
- `right_child(i)`
- `i*2 + 1`

**Complete tree...**

<table>
<thead>
<tr>
<th>50</th>
<th>30</th>
<th>45</th>
<th>25</th>
<th>15</th>
<th>49</th>
<th>42</th>
<th>7</th>
<th>18</th>
<th>5</th>
<th>9</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<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>

**Insert**

- Append new element to the end of array
- Check heap-order property
  - if violated, **Up-Heap** (swap with parent)
  - repeat until heap-order is restored
- if not, we are done

\(O(\log n)\)

**Insert**

1. **Insert**
   - Append new element to the end of array
   - Check heap-order property
     - if violated, **Up-Heap** (swap with parent)
     - repeat until heap-order is restored
   - if not, we are done

\(O(\log n)\)
removeMax

Max element is the first element of the array
the root of the heap

Copy last element of array to first position
then decrement array size by 1 (removes last element)

Check heap-order property
if violated, Down-Heap (swap with larger child)
repeat until heap-order is restored
if not, we are done

Performance

<table>
<thead>
<tr>
<th></th>
<th>Sorted Array/List</th>
<th>Unsorted Array/List</th>
<th>AVL</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>removeMax</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>max</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>insert N</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Performance

<table>
<thead>
<tr>
<th></th>
<th>Sorted Array/List</th>
<th>Unsorted Array/List</th>
<th>AVL</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>removeMax</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>max</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>insert N</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

buildHeap

(**) assuming we know the sequence in advance (buildHeap)
Problem
Build a heap by inserting a sequence of \( n \) elements
‘easy’: call insert \( n \) times \( O(n \log n) \)
Can we do it in linear time?

buildHeap
Place \( n \) items into the tree (array) in any order
keeps structure property
Perform Down-Heap on each internal node
from parent(n) to 1
keeps heap-order property

buildHeap example
input: 10 42 25 13 17 33 45 50 15 22 30 18

buildHeap example
input: 10 42 25 13 17 33 45 50 15 22 30 18

buildHeap example
input: 10 42 25 13 17 33 45 50 15 22 30 18

buildHeap example
input: 10 42 25 13 17 33 45 50 15 22 30 18
Analysis

Cost is **sum of the heights** of all internal nodes

Assume tree is full and complete, thus \( n = 2^{h+1} - 1 \)

\[
T(n) = h + 2(h - 1) + 4(h - 2) + 8(h - 3) + \cdots + 2^h (0)
\]

\[
= \sum_{i=0}^{h} 2^i (h - i) - 2 \sum_{i=0}^{h} 2^i + (h+1)
\]

\[
= h [2^{h+1} - 1] - [2 + (h+1 - 2)2^{h+1}]
\]

\[
= 2^{h+1} - h - 2 = n - (h + 1)
\]

\( = O(n) \)