Data Structures in Java

Lecture 13: Priority Queues (Heaps)

11/4/2015

Daniel Bauer
The Selection Problem

- Given an unordered sequence of $N$ numbers $S = (a_1, a_2, \ldots, a_N)$, select the $k$-th largest number.
Process Scheduling

CPU

Process 1 600ms

Process 2 200ms
Process Scheduling

• Assume a system with a single CPU core.

• Only one process can run at a time.

• Simple approach: Keep new processes on a Queue, schedule them in FIFO order. (Why is a Stack a terrible idea?)

```
CPU
```

```
Process 1  600ms
```

```
Process 2  200ms
```
Process Scheduling

• Assume a system with a single CPU core.

• Only one process can run at a time.

• Simple approach: Keep new processes on a Queue, schedule them in FIFO order. (Why is a Stack a terrible idea?)

• Problem: Long processes may block CPU (usually we do not even know how long).

• Observation: Processes may have different priority (CPU vs. I/O bound, critical real time systems)
Round Robin Scheduling

• Idea: processes take turn running for a certain time interval in round robin fashion.

Queue:

CPU

\[ t \]
Round Robin Scheduling

- Idea: processes take turn running for a certain time interval in round robin fashion.

Queue: 

CPU 

Process 1
Round Robin Scheduling

• Idea: processes take turn running for a certain time interval in round robin fashion.

Queue: 

CPU

Process 1  Process 2

Process 1  Process 3

front  back
Round Robin Scheduling

• Idea: processes take turn running for a certain time interval in round robin fashion.

Queue:  

CPU  

$\text{Process 1} \quad \text{Process 2} \quad \text{Process 1}$
Round Robin Scheduling

- Idea: processes take turn running for a certain time interval in round robin fashion.

Sometimes Process 3 is so crucial that we want to run it immediately when the CPU becomes available!
Priority Scheduling

• Idea: Keep processes ordered by priority. Run the process with the highest priority first.

• Usually lower number = higher priority.

Queued Processes

CPU

process 1
priority 10

process 2
priority 10
Priority Scheduling

• Idea: Keep processes ordered by priority. Run the process with the highest priority first.

• Usually lower number = higher priority.

Queued Processes

CPU

Process 1

Process 2

Priority 10

Priority 10
Priority Scheduling

• Idea: Keep processes ordered by priority. Run the process with the highest priority first.

• Usually lower number = higher priority.

Queued Processes

CPU
Priority Scheduling

• Idea: Keep processes ordered by priority. Run the process with the highest priority first.

• Usually lower number = higher priority.

Queued Processes

<table>
<thead>
<tr>
<th>Priority</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Process 1</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

CPU

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
</table>
The Priority Queue ADT

• A collection Q of comparable elements, that supports the following operations:

  • \textbf{insert}(x) - add an element to Q (compare to \textbf{enqueue}).

  • \textbf{deleteMin}() - return the minimum element in Q and delete it from Q (compare to \textbf{dequeue}).
Other Applications for Priority Queues

• Selection problem.

• Implementing sorting efficiently.

• Keep track of the $k$-best solutions of some dynamic programming algorithm.

• Implementing greedy algorithms (e.g. graph search).
Implementing Priority Queues
Implementing Priority Queues

• Idea 1: Use a Linked List.
  \[\text{insert}(x): O(1), \text{deleteMin}(): O(N)\]
Implementing Priority Queues

• Idea 1: Use a Linked List.
  \[\text{insert}(x): O(1), \text{deleteMin}(): O(N)\]

• Idea 2: Use a Binary Search Tree.
  \[\text{insert}(x): O(\log N), \text{deleteMin}(): O(\log N)\]
Implementing Priority Queues

• Idea 1: Use a Linked List.
  \texttt{insert(x): O(1), deleteMin(): O(N)}

• Idea 2: Use a Binary Search Tree.
  \texttt{insert(x): O(log N), deleteMin(): O(log N)}

• Can do even better with a \textbf{Heap} data structure:
  • Inserting N items in O(N).
  • This gives a sorting algorithm in O(N \log N).
Review: Complete Binary Trees

- All non-leaf nodes have exactly 2 children (full binary tree)
- All levels are completely full (except possibly the last)
Storing Complete Binary Trees in Arrays

- The shape of a complete binary tree with N nodes is unique.
- We can store such trees in an array in level-order.
- Traversal is easy:
  - $\text{leftChild}(i) = 2i$
  - $\text{rightChild}(i) = 2i + 1$
  - $\text{parent}(i) = \lfloor i/2 \rfloor$
Storing Incomplete Binary Trees in Arrays

- Assume the tree takes as much space as a complete binary tree, but only store the nodes that actually exist.
Heap

- A heap is a complete binary tree stored in an array, with the following **heap order property**:
  - For every node $n$ with value $x$:
    - the values of all nodes in the subtree rooted in $n$ are greater or equal than $x$. 
Max Heap

- A heap is a complete binary tree stored in an array, with the following **heap order property**:
- For every node $n$ with value $x$:
  - the values of all nodes in the subtree rooted in $n$ are **less or equal** than $x$. 

```
| 20 | 16 | 15 | 13 | 14 | 8 | 9 | 10 | 5 | 1 |
```
Min Heap - insert(x)

- Attempt to insert at last array position (next possible leaf in the last layer).
- If heap order property is violated, percolate the value up.
  - Swap that value (‘hole’) and value in the parent cell, then try the new cell.
  - If heap order is still violated, continue until correct position is found.
Min Heap - insert(x)

- Attempt to insert at last array position (next possible leaf in the last layer).
- If heap order property is violated, *percolate* the value *up*.
  - Swap that value (‘hole’) and value in the parent cell, then try the new cell.
- If heap order is still violated, continue until correct position is found.

![Min Heap - insert(3) Diagram](image)
Min Heap - insert(x)

- Attempt to insert at last array position (next possible leaf in the last layer).
- If heap order property is violated, *percolate* the value *up*.
  - Swap that value (‘hole’) and value in the parent cell, then try the new cell.
  - If heap order is still violated, continue until correct position is found.
Min Heap - deleteMin()

• The minimum is always at the root of the tree.
• Remove lowest item, creating an empty cell in the root.
• Try to place last item in the heap into the root.
  • If heap order is violated, *percolate* the value *down*:
    • Swap with the smaller child until correct position is found.
Min Heap - `deleteMin()`

- The minimum is always at the root of the tree.
- Remove lowest item, creating an empty cell in the root.
- Try to place last item in the heap into the root.
- If heap order is violated, *percolate* the value *down*:
  - Swap with the smaller child until correct position is found.

<table>
<thead>
<tr>
<th>15</th>
<th>3</th>
<th>10</th>
<th>8</th>
<th>5</th>
<th>14</th>
<th>13</th>
<th>9</th>
<th>20</th>
<th>16</th>
</tr>
</thead>
</table>

![Min Heap Diagram]

`deleteMin()` → 1
Min Heap - deleteMin()

- The minimum is always at the root of the tree.
- Remove lowest item, creating an empty cell in the root.
- Try to place last item in the heap into the root.
- If heap order is violated, **percolate** the value **down**:
  - Swap with the smaller child until correct position is found.
Min Heap - `deleteMin()`

- The minimum is always at the root of the tree.
- Remove lowest item, creating an empty cell in the root.
- Try to place last item in the heap into the root.
- If heap order is violated, **percolate** the value **down**:
  - Swap with the smaller child until correct position is found.
Running Time for Heap Operations

• Because a Heap is a complete binary tree, it’s height is about log N.

• Worst-case running time for \texttt{insert(x)} and \texttt{deleteMin()} is therefore $O(\log N)$.

• \texttt{getMin()} is $O(1)$. 
Building a Heap

• Want to convert an collection of N items into a heap.

• Each \texttt{insert(x)} takes $O(\log N)$ in the worst case, so the total time is $O(N \log N)$.

• Can show a better bound $O(N)$ for building a heap.
Building a Heap Bottom-Up

• Start with an unordered array.

• `percolateDown(i)` assumes that both subtrees under `i` are already heaps.

• Idea: restore heap property bottom-up.
  • Make sure all subtrees in the two last layers are heaps.
  • Then move up layer-by-layer.
Building a Heap Bottom-Up

- Start with an unordered array.

- `percolateDown(i)` assumes that both subtrees under `i` are already heaps.

- Idea: restore heap property bottom-up.
  - Make sure all subtrees in the two last layers are heaps.
  - Then move up layer-by-layer.
Building a Heap Bottom-Up

• Start with an unordered array.

• `percolateDown(i)` assumes that both subtrees under $i$ are already heaps.

• Idea: restore heap property bottom-up.
  • Make sure all subtrees in the two last layers are heaps.
  • Then move up layer-by-layer.
Building a Heap Bottom-Up

• Start with an unordered array.

• `percolateDown(i)` assumes that both subtrees under \( i \) are already heaps.

• Idea: restore heap property bottom-up.
  • Make sure all subtrees in the two last layers are heaps.
  • Then move up layer-by-layer.
Building a Heap Bottom-Up

• Start with an unordered array.

• `percolateDown(i)` assumes that both subtrees under `i` are already heaps.

• Idea: restore heap property bottom-up.

• Make sure all subtrees in the two last layers are heaps.

• Then move up layer-by-layer.

For \( i = \lfloor N/2 \rfloor \ldots 1 \)
`percolateDown(i)`
Building a Heap - Example

For \( i = \lfloor N/2 \rfloor \ldots 1 \) 
percolateDown\( (i) \)

\( i = 11/2 = 5 \)
Building a Heap - Example

For \( i = \lfloor N/2 \rfloor \ldots 1 \), percolateDown(i)

i=4

Diagram:

```
  5
 / \
4   6
 / \ / \n9   1  8
 / \ / \ / \n10 7 2 11
```

Array:

```
5  4  6  9  1  8  3  10  7  2  11
```
Building a Heap - Example

For $i = \lfloor N/2 \rfloor \ldots 1$

percolateDown($i$)

i=4

\[
\begin{array}{c}
5 & 4 & 6 & 7 & 1 & 8 & 3 & 10 & 9 & 2 & 11
\end{array}
\]
Building a Heap - Example

For $i = \lfloor N/2 \rfloor \ldots 1$

percolateDown($i$)

For $i = 3$

$i=3$
Building a Heap - Example

For $i = \lceil N/2 \rceil \ldots 1$
percolateDown($i$)

i=3

5 4 3 7 1 8 6 10 9 2 11
Building a Heap - Example

For $i = \lfloor N/2 \rfloor \ldots 1$

percolateDown(i)

For $i = 2$

5 4 3 7 1 8 6 10 9 2 11
Building a Heap - Example

For $i = \lfloor N/2 \rfloor \ldots 1$

percolateDown(i)

For $i = 2$

percolateDown(i)
Building a Heap - Example

For $i = \lceil N/2 \rceil \ldots 1$

percolateDown($i$)

For $i = 2$
Building a Heap - Example

For $i = \lfloor N/2 \rfloor \ldots 1$
percolateDown($i$)

For $i = 1$...
Building a Heap - Example

For \( i = \lceil N/2 \rceil \ldots 1 \)
\[ \text{percolateDown}(i) \]
Building a Heap - Example

For $i = \lfloor N/2 \rfloor \ldots 1$

percolateDown(i)
Building a Heap - Example

For $i = \lfloor N/2 \rfloor$ … 1
percolateDown(i)
BuildHeap - Running Time

• How many comparisons do we need in each of the \( N/2 \) \texttt{percolateDown} calls?

• In the worst case, each call to \texttt{percolateDown} needs to move the value all the way down to the leaf level.

• We need to sum the possible steps for each level of the tree.
BuildHeap - Running Time

- Upper bound for nodes in a complete binary tree (if all levels are full): \(2^{h+1} - 1\)

- A complete binary tree with \(N\) nodes has height: \(h = \lfloor \log(N + 1) \rfloor\)
BuildHeap - Running Time

$2^h \cdot 0$ nodes · $0$ steps

$8 \cdot 0 \rightarrow$
BuildHeap - Running Time

$2^{h-1}$ nodes · 1 steps

$2^h$ nodes · 0 steps
BuildHeap - Running Time

$2^{h-2}$ nodes $\cdot$ 2 steps

$2^{h-1}$ nodes $\cdot$ 1 steps

$2^h$ nodes $\cdot$ 0 steps
BuildHeap - Running Time

$2^{h-3}$ nodes · 3 steps

$2^{h-2}$ nodes · 2 steps

$2^{h-1}$ nodes · 1 steps

$2^h$ nodes · 0 steps
BuildHeap - Running Time

\[ T(N) = 2^{h-1} \cdot 1 + \cdots + 4 \cdot (h - 2) + 2 \cdot (h - 1) + h \cdot 1 \]
BuildHeap - Running Time

\[ T(N) = 2^{h-1} \cdot 1 + \cdots + 4 \cdot (h - 2) + 2 \cdot (h - 1) + h \cdot 1 = \sum_{j=0}^{h-1} j \cdot 2^{h-j} \]
BuildHeap - Running Time

\[ 2T(N) = 2^h \cdot 1 + \cdots + 8 \cdot (h - 2) + 4 \cdot (h - 1) + h \cdot 2 \]

\[ T(N) = 2^{h-1} \cdot 1 + \cdots + 4 \cdot (h - 2) + 2 \cdot (h - 1) + h \cdot 1 \]

\[
2T(N) - T(N) = 2^h + 2^{h+1} + \cdots + 8 + 4 + 2 + h
\]

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

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

\[ T(N) = (2^{h+1} - 1) - (\log(N + 1) + 1) = O(N) \]
The Selection Problem

• Given an unordered sequence of \( N \) numbers \( S = (a_1, a_2, \ldots , a_N) \), select the \( k \)-th largest number.

• Approach 1: Sort the numbers in decreasing order. Then pick the number at \( k \)-th position. \( \Rightarrow O(N \log N + k) \)

• Approach 2: Initialize array of size \( k \) with the first \( k \) numbers. Sort the array in decreasing order. For every element in the sequence, if it is larger than the \( k \)-th entry in the array, replace the appropriate entry in the array with the new number.
\( \Rightarrow O(k \log k) + O(N \cdot k) \)
The Selection Problem

- Given an unordered sequence of \( N \) numbers \( S = (a_1, a_2, \ldots, a_N) \), select the \( k \)-th largest number.

- Using a Heap (Option 1):
  - First build a Max-Heap in \( O(N) \).
  - Then call `deleteMax()` \( k \) times \( O(k \log N) \).
  - Total: \( O(N + k \log N) \)
  - If \( k \) has a linear dependence on \( N \) (e.g. \( k=N/2 \)), then the total is \( O(N \log N) \).
The Selection Problem

• Given an unordered sequence of $N$ numbers $S = (a_1, a_2, \ldots, a_N)$, select the $k$-th largest number.

• Using a Heap (Option 2):
  
  • Build a Min-Heap $S$ from the first $k$ unordered elements in $O(k)$.
  
  • The root of $S$ now contains the $k$-th largest element.
  
  • Iterate through the remaining $N-k = O(N)$ numbers:
    
    • If a number is larger than the root of $S$, remove the root of $S$ and insert the new number into $S$. This takes $O(\log k)$ time.
    
    • Total: $O(k + N \cdot \log k) = O(N \log k)$