Priority Queues and Heaps

CMPT 225
Remember Queues?

- A queue should implement at least the first two of these operations:
  - `insert` – insert item at the back of the queue
  - `remove` – remove an item from the front
  - `peek` – return the item at the front of the queue without removing it
- It is assumed that these operations will be implemented efficiently
  - That is, in constant time
Queue Implementations

- Either with an array

- Or with a linked list
FIFO

- Queues are first-in first-out (FIFO)
- Priority queues are a fancier type of queue
  - Maintains an ordering of items in the queue, not necessarily first-in first-out
Priority Queues
Items in a priority queue are given a priority value
  - Which could be numerical or something else
The highest priority item is removed first
Uses include
  - System requests
  - Data structure to support Dijkstra’s Algorithm
Priority Queue Problem

- Can items be inserted and removed efficiently from a priority queue?
  - Using an array, or
  - Using a linked list?
- Note that items are not removed based on the order in which they are inserted

Now we’ll see how we can do these efficiently (using a different data structure)
Items in a priority queue have a *priority*
- Not necessarily numerical
- Could be lowest first or highest first
The highest priority item is removed first
Priority queue operations
- Insert
- Remove in priority queue order
  - Both operations should be performed in at most $O(\log n)$ time
Priority Queue Implementation
Items have to be removed in priority order
- This can only be done efficiently if the items are ordered in some way
- One option would be to use a balanced binary search tree
  - Binary search trees are fully ordered and insertion and removal can be implemented in $O(\log n)$ time
    - Some operations (e.g. removal) are complex
    - Although operations are $O(\log n)$ they require quite a lot of structural overhead
- There is a much simpler binary tree solution
Heaps

- A heap is binary tree with two properties
- Heaps are complete
  - All levels, except the bottom, are completely filled in
  - The leaves on the bottom level are as far to the left as possible.
- Heaps are partially ordered
  - The value of a node is at least as large as its children’s values, for a max heap or
  - The value of a node is no greater than its children’s values, for a min heap
Complete Binary Trees

- Complete binary trees

- Incomplete binary trees
Heaps are not fully ordered, an inorder traversal would result in:

9, 13, 10, 86, 44, 65, 23, 98, 21, 32, 17, 41, 29
Priority Queues and Heaps

- A heap can implement a priority queue
- The item at the top of the heap must always be the highest priority value
  - Because of the partial ordering property
- Implement priority queue operations:
  - Insertions – insert an item into a heap
  - Removal – remove and return the heap’s root
  - For both operations preserve the heap property
Heap Implementation
Using an Array
Heaps can be implemented using *arrays*

There is a natural method of indexing tree nodes
- Index nodes from top to bottom and left to right as shown on the right
- Because heaps are *complete* binary trees there can be no gaps in the array
Referencing Nodes

- It will be necessary to find the index of the parents of a node
  - Or the children of a node
- The array is indexed from 0 to $n - 1$
  - Each level's nodes are indexed from:
    - $2^{\text{level}} - 1$ to $2^{\text{level} + 1} - 2$ (where the root is level 0)
  - The children of a node $i$, are the array elements indexed at $2i + 1$ and $2i + 2$
  - The parent of a node $i$, is the array element indexed at $\text{floor}((i - 1) / 2)$
Heap Array Example

Heap

Underlying Array

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>98</td>
<td>86</td>
<td>41</td>
<td>13</td>
<td>65</td>
<td>32</td>
<td>29</td>
<td>9</td>
<td>10</td>
<td>44</td>
<td>23</td>
<td>21</td>
<td>17</td>
</tr>
</tbody>
</table>

October 2004  John Edgar
Heap Insertion
Heap Insertion

- On insertion the heap properties have to be maintained; remember that
  - A heap is a complete binary tree and
  - A partially ordered binary tree
- There are two general strategies that could be used to maintain the heap properties
  - Make sure that the tree is complete and then fix the ordering or
  - Make sure the ordering is correct first
- Which is better?
Heap Insertion Sketch

- The insertion algorithm first ensures that the tree is complete
  - Make the new item the first available (left-most) leaf on the bottom level
  - i.e. the first free element in the underlying array
- Fix the partial ordering
  - Compare the new value to its parent
  - Swap them if the new value is greater than the parent
  - Repeat until this is not the case
    - Referred to as *bubbling up*, or *trickling up*
Insert 81

Heap Insertion Example

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>98</td>
<td>86</td>
<td>41</td>
<td>13</td>
<td>65</td>
<td>32</td>
<td>29</td>
<td>9</td>
<td>10</td>
<td>44</td>
<td>23</td>
<td>21</td>
<td>17</td>
<td></td>
</tr>
</tbody>
</table>
Heap Insertion Example

Insert 81

81 is less than 98 so finished

(13-1)/2 = 6
Heap Removal
Heap Removal

- Make a temporary copy of the root’s data
- Similarly to the insertion algorithm, first ensure that the heap remains complete
  - Replace the root node with the right-most leaf
  - i.e. the highest (occupied) index in the array
- Swap the new root with its largest valued child until the partially ordered property holds
  - i.e. bubble down
- Return the root’s data
Heap Removal Example

Remove (root)

```
index  | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12
value  | 98 | 86 | 41 | 13 | 65 | 32 | 29 | 9  | 10 | 44 | 23 | 21 | 17
```
Heap Removal Example

Remove (root)

replace root with right-most leaf

index

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>17</td>
<td>86</td>
<td>41</td>
<td>13</td>
<td>65</td>
<td>32</td>
<td>29</td>
<td>9</td>
<td>10</td>
<td>44</td>
<td>23</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>
Heap Removal Example

Remove (root)

? 86  ?

swap with larger child

children of root: 2*0+1, 2*0+2 = 1, 2

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>86</td>
<td>17</td>
<td>41</td>
<td>13</td>
<td>65</td>
<td>32</td>
<td>29</td>
<td>9</td>
<td>10</td>
<td>44</td>
<td>23</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>
Heap Removal Example

```
Remove (root)

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>86</td>
<td>65</td>
<td>41</td>
<td>13</td>
<td>17</td>
<td>32</td>
<td>29</td>
<td>9</td>
<td>10</td>
<td>44</td>
<td>23</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

children of 1: 2*1+1, 2*1+2 = 3, 4
```

swap with larger child

Remove (root)

? 65 ?

13 17

9 10 44 23

21

41

October 2004

John Edgar
Heap Removal Example

Remove (root)

swap with larger child

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>86</td>
<td>65</td>
<td>41</td>
<td>13</td>
<td>44</td>
<td>32</td>
<td>29</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>23</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

children of 4: 2*4+1, 2*4+2 = 9, 10

October 2004 John Edgar
Bubbles
Helper functions are usually written for preserving the heap property

- *bubbleUp* ensures that the heap property is preserved from the start node up to the root
- *bubbleDown* ensures that the heap property is preserved from the start node down to the leaves

These functions may be implemented recursively or iteratively
BubbleDown Algorithm

Go to terminal
Removal Algorithm
Insertion Algorithm

- Lab next week
BubbleUp Algorithm

- Lab next week
For both insertion and removal the heap performs at most \textit{height} swaps
\begin{itemize}
  \item For insertion at most \textit{height} comparisons
    \begin{itemize}
    \item To bubble up the array
    \end{itemize}
  \item For removal at most \textit{height} * 2 comparisons
    \begin{itemize}
    \item To bubble down the array (have to compare two children)
    \end{itemize}
\end{itemize}
\begin{itemize}
  \item Height of a complete binary tree is \lfloor \log_2(n) \rfloor
    \begin{itemize}
    \item Both insertion and removal are therefore $O(\log n)$
    \end{itemize}
\end{itemize}
Sorting with Heaps
Observation 1: Removal of a node from a heap can be performed in $O(\log n)$ time

Observation 2: Nodes are removed in order

Conclusion: Removing all of the nodes one by one would result in sorted output

Analysis: Removal of all the nodes from a heap is a $O(n \times \log n)$ operation
A heap can be used to return sorted data
- In $O(n \times \log n)$ time
However, we can’t assume that the data to be sorted just happens to be in a heap!
- Aha! But we can put it in a heap.
- Inserting an item into a heap is a $O(\log n)$ operation so inserting $n$ items is $O(n \times \log n)$
But we can do better than just repeatedly calling the insertion algorithm
To create a heap from an unordered array repeatedly call `bubbleDown`

- Any subtree in a heap is itself a heap
- Call `bubbleDown` on elements in the upper ½ of the array
- Start with index \( n/2 \) and work up to index 0
  - i.e. from the last non-leaf node to the root

`bubbleDown` does not need to be called on the lower half of the array (the leaves)

- Since `bubbleDown` restores the partial ordering from any given node down to the leaves
Heapify Example

Assume unsorted input is contained in an array as shown here (indexed from top to bottom and left to right)
Heapify Example

\[ n = 12, \frac{(n-1)}{2} = 5 \]

\( \text{bubbleDown}(5) \)
Heapify Example

n = 12, (n-1)/2 = 5

bubbleDown(5)

bubbleDown(4)
Heapify Example

n = 12, (n-1)/2 = 5

bubbleDown(5)
bubbleDown(4)
bubbleDown(3)
Heapify Example

n = 12, (n-1)/2 = 5
bubbleDown(5)
bubbleDown(4)
bubbleDown(3)
bubbleDown(2)
Heapify Example

\[ n = 12, \frac{n-1}{2} = 5 \]

bubbleDown(5)
bubbleDown(4)
bubbleDown(3)
bubbleDown(2)
bubbleDown(1)
Heapify Example

n = 12, (n-1)/2 = 5

bubbleDown(5)
bubbleDown(4)
bubbleDown(3)
bubbleDown(2)
bubbleDown(1)
bubbleDown(0)
Cost to Heapify an Array

- `bubbleDown` is called on half the array
  - The cost for `bubbleDown` is $O(\text{height})$
  - It would appear that heapify cost is $O(n \times \log n)$
- In fact the cost is $O(n)$
- The analysis is complex but
  - `bubbleDown` is only called on $n/2$ nodes
  - and mostly on sub-trees
HeapSort Algorithm Sketch

- Heapify the array
- Repeatedly remove the root
  - After each removal swap the root with the last element in the tree
  - The array is divided into a heap part and a sorted part
- At the end of the sort the array will be sorted in reverse order
HeapSort Notes

- The algorithm runs in $O(n \times \log n)$ time
  - Considerably more efficient than selection sort and insertion sort
  - The same average case complexity as MergeSort and QuickSort
- The sort can be carried out in-place
  - That is, it does not require that a copy of the array to be made
Summary
Objectives

- Define the ADT priority queue
- Define the partially ordered property
- Define a heap
- Implement a heap using an array
- Implement the heapSort algorithm
Readings

- Java Ch. 12
- C++ Ch. 11