CS200: Priority Queues and Heaps

Walls & Mirrors Ch. 12.2
Quick Review of Binary Trees

- # of levels = 3
- Height = 3
- Full binary tree: All interior nodes have m children
- Perfect binary tree: Full binary tree where all leaves are at the same level
- Perfect binary tree
  - number of leaf nodes: $2^{h-1}$
  - total number of nodes: $2^h - 1$
Complete Binary Tree

Height of a complete tree: $\text{ceil}(\log(n+1))$
Array implementation of complete binary trees

Left child of node $I$ is $2I + 1$
Right child is $2I + 2$
Parent of a node is $(I - 1) / 2$
Review Cont’d

- Binary Search Trees

```
      5
     / \
    2   7
   / \ / \  
  1  4 6  9
   \  /  /   
    3     
```
Review Cont’d

- BST Insert

Add 6
Review Cont’d

- Delete (left child has no right child)
delete(5)

left child becomes root
(or vice-versa for right child with no left child)
Review Cont’d

- Delete

delete(5)
### Complexity of BST Operations

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>delete</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Compare with a sorted list
Tree Sort

- Uses the binary search tree ADT to sort an array of records according to search-key

- Efficiency: Use multiplication and addition rules
  - **Average case:** $O(n \times \log n)$
    - BST insertion is $O(\log n)$, # of insertions is $n$
    - BST traversal $O(n)$ (one copy operation for each element)
  - **Worst case:** $O(n^2)$
    - BST insertion is $O(\log n)$, # of insertions is $n$
Priority Queues

- Characteristics
  - items are associated with a priority
  - provide access to one element at a time - the one with the highest priority

- Uses
  - operating systems
  - network management (real time traffic usually gets highest priority when bandwidth is limited)
Priority Queue ADT Operations

1. Create an empty priority queue
   `createPQueue()`

2. Determine whether empty
   `pqIsEmpty():boolean`

3. Insert new item
   `pqInsert(in newItem:PQItemType) throws PQQueueException`

4. Retrieve and delete the item with the highest priority
   `pqDelete():PQItemType`
Priority Queue – Implementation

- ArrayList ordered by priority
  - Sorted ArrayList
    - location of maximum element?
    - add – find the insertion location
  - Unsorted ArrayList
    - add – at end of vector
    - find the maximum element (linear search)
- Binary search tree
Heap - Definition

- A heap is a complete binary tree that satisfies the following:
  - It is an empty tree
  - or it has the heap property:
    - Its root contains a key greater or equal to the keys of its children (for a maximum heap)
    - Its left and right subtrees are also heaps

- Implications of the heap property:
  - The root holds the maximum value (global property)
  - values in descending order on every path from root to leaf
Heap Examples - Validity

- Satisfies heap property
  - complete

- Satisfies heap property
  - Not complete

- Does not satisfy heap property
  - Not complete
Heap ADT

- `createHeap()`  // create empty heap
- `heapIsEmpty()`:boolean  
  // determines if empty
- `heapInsert(in newItem:HeapItemType) throws HeapException`  
  // inserts newItem based on its search key. Throws  
  // exception if heap is full
- `heapDelete()`:HeapItemType  
  // retrieves and then deletes heap’s root item, which has  
  // largest search key

This is essentially the queue ADT.
ArrayList-based Implementation

- Accessing items:
  - root at position 0
  - left child of position i at position 2i+1
  - right child of position i at position 2(i+1)
  - parent of position i at position ⌊(i-1)/2⌋
ArrayList-based Implementation

Why is there so much wasted space?
ArrayList-based Heap – Example
Heap Operations - heapInsert

- put a new value into first open position (maintaining completeness)
- percolate/sift/bubble values up
  - Re-enforcing the heap property
  - swap with parent until in the right place

Add 36 to the heap
Heap Operations - heapInsert

- put a new value into first open position (maintaining completeness)

- percolate values up
  - Re-enforcing the heap property
  - swap with parent until in the right place

Add 36 to the heap
Heap Operations - heapInsert

- put a new value into first open position (maintaining completeness)
- percolate values up
  - Re-enforcing the heap property
  - swap with parent until in the right place

Add 36 to the heap
Heap Operations - heapInsert

- put a new value into first open position (maintaining completeness)
- percolate values up
  - Re-enforcing the heap property
  - swap with parent until in the right place

Add 36 to the heap
Heap Insert Pseudocode

// insert newItem into bottom of tree
items[size] = newItem

// percolate new item up to appropriate spot
place = size-1
parent = (place-1)/2

while (parent >= 0 and items[place] > items[parent]) {
    swap items[place] and items[parent]
    place = parent
    parent = (place-1)/2
}

increment size

Part of the insert operation is often called siftUp
Heap operations – heapDelete

- always remove value at root (Why?)
- substitute with rightmost leaf of bottom level (Why?)
- percolate/sift down
  - swap with maximum child as necessary

Delete from heap
Heap operations – heapDelete

- always remove value at root (Why?)
- substitute with rightmost leaf of bottom level (Why?)
- percolate down
  - swap with maximum child as necessary

![Heap Diagram]

Delete from heap

Save 36
Heap operations – heapDelete

- always remove value at root (Why?)
- substitute with rightmost leaf of bottom level (Why?)
- percolate down
  - swap with maximum child as necessary

Return 36

Delete from heap
heapDelete Pseudocode

// return the item in root
rootItem = items[0]

// copy item from last node into root
items[0] = items[size-1]
size--

// restore the heap property
heapRebuild(items, 0, size)

return rootItem
heapRebuild Pseudocode

heapRebuild(inout items:ArrayType, in root:integer, in size:integer)
    if (root is not a leaf) {
        child = 2 * root + 1  // left child
        if (root has right child) {
            rightChild = child + 1
            if (items[rightChild].getKey() > items[child].getKey()) {
                child = rightChild  } // larger child
        }
        if (items[root].getKey() < items[child].getKey()) {
            swap items[root] and items[child]
            heapRebuild(items, child, size)
        }
    }

heapRebuild is also called siftDown
Array-based Heaps: Complexity

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>delete</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Heap versus BST for PriorityQueue

- BST can also be used to implement a priority queue
- How does worst case complexity compare?
- How does average case complexity compare?
- What if you know maximum needed size for the PriorityQueue?
Small number of priorities

- A heap of queues with a queue for each priority value.
HeapSort

Algorithm
- Insert all elements (one at a time) to a heap
- Iteratively delete them
  - removes minimum/maximum value at each step

Computational complexity?
HeapSort

- Alternative method (in-place):
  - Create a heap out of the input array:
    - Consider the input array as a complete binary tree
    - Create a heap by iteratively expanding the portion of the tree that is a heap
      - Start from leaves, which are semi-heaps
      - Move up to next level calling heapRebuild with each parent
  - Iteratively swap the root item with last item in unsorted portion and rebuild
Example

(a) \( \text{anArray} \)

\[
\begin{array}{cccccc}
6 & 3 & 5 & 9 & 2 & 10 \\
0 & 1 & 2 & 3 & 4 & 5
\end{array}
\]

After making \( \text{anArray} \) a heap

Array \( \text{anArray} \)

\[
\begin{array}{cccccc}
10 & 9 & 6 & 3 & 2 & 5 \\
\text{last}
\end{array}
\]

Tree representation of Heap region

(b)

Heap

\[
\begin{array}{cccccc}
5 & 9 & 6 & 3 & 2 & 10 \\
\text{last}
\end{array}
\]

Sorted

After swapping \( \text{anArray}[0] \) with \( \text{anArray}[\text{last}] \) and decrementing \( \text{last} \)

Heap

\[
\begin{array}{cccccc}
5 & 9 & 6 & 3 & 2 & 10 \\
\text{last}
\end{array}
\]

Sorted

After heapRebuild(\( \text{anArray}, 0, 4 \))

Heap

\[
\begin{array}{cccccc}
9 & 5 & 6 & 3 & 2 & 10 \\
\text{last}
\end{array}
\]

Sorted

\[
\begin{array}{cccccc}
9 & 5 & 6 & 3 & 2 \\
\text{last}
\end{array}
\]

CS200
HeapSort Pseudocode

heapSort(ourItems:ArrayList, n:integer)
   // First step: build heap out of the input array
   for (index = n - 1 down to 0) {
      // Invariant: the tree rooted at index is a semiheap
      // semiheap: tree where the subtrees of the root
      // are heaps
      heapRebuild(ourItems, index, n)
      // The tree rooted at index is a heap
   }

Can replace n – 1 with n / 2 in the loop. Why?
Example

<table>
<thead>
<tr>
<th>Original anArray</th>
<th>Array anArray</th>
<th>Tree representation of anArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 3 5 9 2 10</td>
<td>6 3 5 9 2 10</td>
<td></td>
</tr>
<tr>
<td>After heapRebuild(anArray, 2, 6)</td>
<td>6 3 10 9 2 5</td>
<td></td>
</tr>
<tr>
<td>After heapRebuild(anArray, 1, 6)</td>
<td>6 9 10 3 2 5</td>
<td></td>
</tr>
<tr>
<td>After heapRebuild(anArray, 0, 6)</td>
<td>10 9 6 3 2 5</td>
<td></td>
</tr>
</tbody>
</table>
HeapSort Pseudocode

heapSort(ourItems:ArrayList, n:integer)
    for (index = n/2 down to 0) {
        heapRebuild(ourItems, index, n)
    }

    last = n -1
    for (step = 1 through n) {
        swap ourItems[0] and ourItems[last]
        decrement last
        heapRebuild(ourItems, 0, last)  
    }

Heap

Sorted (largest elements in array)

0 1  ****  last last+1  ****  n-1
Building a Heap

- What is the complexity of building a heap?
- An alternative way to express the heap-building part of the code:

```java
heapBuild(ourItems:ArrayList, index:integer, n:integer)
    if index is not a leaf{
        heapBuild(ourItems, 2*index + 1, n)  // left subtree
        heapBuild(ourItems, 2*index + 2, n)  // right subtree
        heapRebuild(ourItems, index, n)
    }
```