CS200: Priority Queues and Heaps

Walls & Mirrors Ch. 12.2

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)
  - Search problems in AI (A*)

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 PQueueException
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?
  - Unsorted ArrayList
    - add – find the insertion location
  - Binary search tree

Heap - Definition

- A heap is a complete binary tree that satisfies:
  - 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

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]

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

Heap Insert Pseudocode
```c
// 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

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

```plaintext
// 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

```plaintext
heapRebuild(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) / O(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

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?

Building a Heap

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

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
}