CS 209
Data Structures and Mathematical Foundations

04 / 03 / 2024

Instructor: Michael Eckmann

Today’s Topics

- Questions?/Comments?
- Priority Queues
  - Binary Heaps

Next data structures

- Priority Queues
- Heap

Queues and Stacks

- Queue
  - FIFO
  - Enqueue (to one side)
  - Dequeue (from other side)
  - Empty?
- Stack
  - LIFO
  - Push (to top)
  - Pop (from top)
  - Peek
  - Empty?

Priority Queues

- The main difference between a Queue and a Priority Queue is that
  - In a Queue, an item will be dequeued based on the order in which it was enqueued
  - In a Priority Queue
    - Every item is enqueued with a priority value
    - Higher priority items have preference when dequeuing
    - Items with the same priority can be enqueued / dequeued in any order

Priority Queues

- Priority queues have the following characteristics
  - Each item placed into a priority queue has a priority value associated with it
  - When a dequeue is requested from a priority queue, we dequeue the highest priority item (if there are multiple items with this same highest priority value, then any one of those can be dequeued)
  - We need a way to determine if the priority queue is empty
  - We can also have a peek to see what item will be dequeued next but without removing it

Priority Queues

- One common implementation of a priority queue is with a binary heap.
- We will discuss binary heaps now.

Heaps

- A binary heap is a binary tree in which the arrangement of the values in the heap adhere to the following rules.
  - 1) the element in each node is >= the elements of that node's children
  - 2) the tree is complete -
    - every level except the deepest must contain as many nodes as possible
    - and at the deepest level all the nodes are as far left (in that level) as possible
- Rule 1 above is for maxHeaps, but if change that rule such that a node is <= children, then it is called a minHeap
- Examples of heaps and non-heaps on the board.
- Differentiate heaps from BSTs.

Heaps

- Besides the node style implementation of binary trees (look back at BSTNode and BST classes), a binary tree can be implemented with a list.
- The list implementation works well for complete binary trees. Example on the board.
- Node [i] has its children (if they exist) at
    - left child: [2i+1]
    - right child: [2i+2]
- If Node [i] is not the root, then Node [i]'s parent is at
    - [(i-1)//2]

Heaps

- To insert a new node into a heap
  - 1) after the node is inserted the structure still must be a heap
    - that is it must still be a complete binary tree
    - with the element in each node >= the elements of that node's children
  - Any ideas on how to do this, efficiently?

Heaps

- To insert a new node into a heap
  - Place the node in the next open slot (deepest level and as far to the left as possible) ---- when this is done the tree is complete
  - Then, check that node against its parent and swap if necessary (when a node's parent is < the node). Continue to do this until we don't need to swap or we reach the root.
  - When this process is done, we will have a heap.

- The process of rising a node into its proper place is (upward) reheapification. To heapify is to start with a complete binary tree (the result of the first step above) that may not be a heap and make it a heap.

Heaps

- To delete the root from the heap
  - 1) after the root is removed from the heap, the structure still must be a heap
    - that is it must still be a complete binary tree
    - with the element in each node >= the elements of that node's children
  - Any ideas on how to do this, efficiently?

Heaps

- To remove the root from the heap
  - if the tree consists of only one node, the result is an empty tree which is a heap and we are done
  - if the tree consists of more than one node, then move the last element (the one furthest to the right in the deepest level) to the root
    - now we have a complete binary tree with one fewer nodes
    - but it may not be a heap, so we need to heapify, i.e. do (downward) reheapification.
    - Let's name the element that moved to the root the out-of-place node
    - while (out-of-place node is < one of its children)
      - swap the out-of-place node with its larger child
      - out-of-place node is now where the larger child was
      - Do this until no swap occurs or when reach a leaf

Heaps

- Heaps can be used to implement Priority Queues
  - Main operations of a priority queue
    - remove (highest priority item)
      - dequeue
    - add
      - enqueue
- For a Heap implementation of a priority queue, we would remove from the root (and then make sure the heap remains a heap by the process we described earlier.)
- For add, we would place at last slot and upward reheapify.

Heaps

- Heaps are not used for searching (inefficient), nor are they used for anything other than what they are designed for.
- Let's analyze runtime for remove largest (in a max heap) and add for heaps.

Heaps

- Let's create a class (named ItemAndPriority) that holds a data item of some type, and a priority value (int).
- Then let's create a class Heap that stores objects of them in a heap (implemented as a list).
- This Heap then can be used as a priority queue based on the priority value in the ItemAndPriority class.