CS 240
Fall 2015

Mike Lam, Professor

Priority Queues and Heaps

Priority Queues

- FIFO abstract data structure w/ priorities
  - Always remove item with highest priority
- Store key (priority) with value
  - Store (key, value) tuples as items
  - Goal: retrieve/remove the lowest key value (highest priority)
  - Usually ignore the values during discussion for simplicity
- Priority Queue ADT operations:
  - `add(k,v)`
  - `remove_min()`
  - `min()`
  - `is_empty()`
  - `size()`

Priority Queues

- First idea: store them in an unordered list
- Second idea: store them in an ordered list

Priority Queues

- First idea: store them in an unordered list
  - add() is O(1) and remove_min() is O(n)
  - The latter has to find the minimum element
- Second idea: store them in an ordered list
  - add() is O(n) and remove_min() is O(1)
  - The former has to insert the element in order
- Can we do better?

Priority Queues

- First idea: store them in an unordered list
  - add() is O(1) and remove_min() is O(n)
  - The latter has to find the minimum element
- Second idea: store them in an ordered list
  - add() is O(n) and remove_min() is O(1)
  - The former has to insert the element in order
- Can we do better?
  - We could use skip lists or AVL trees
    - Skip lists require O(n log n) extra space
    - AVL trees require O(n) extra space
    - Plus, they're so complicated!

Priority Queues

- Basic idea: use binary trees
  - Dense binary trees are limited in height by *O(log n)*
  - If we can devise a scheme to add and remove elements from a binary tree in a way that is guaranteed to visit each level only once, we can insert and remove in *O(log n)* time
  - In order to do this, we have to impose some kind of order on the binary tree
  - But we don't need full BSTs

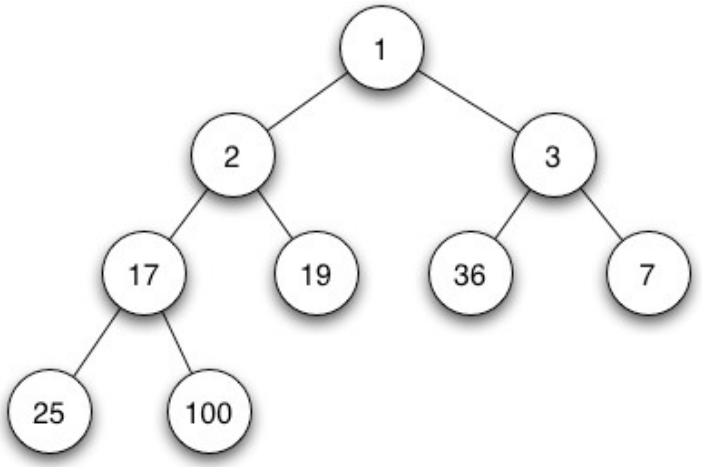


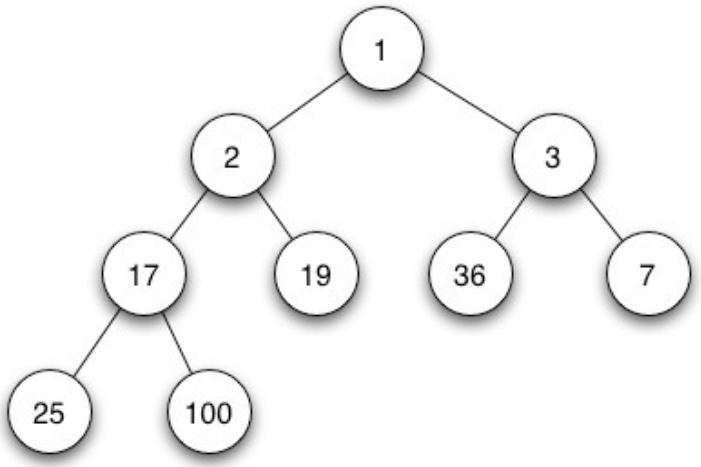
Image taken from
`http://en.wikipedia.org/wiki/Binary_heap`

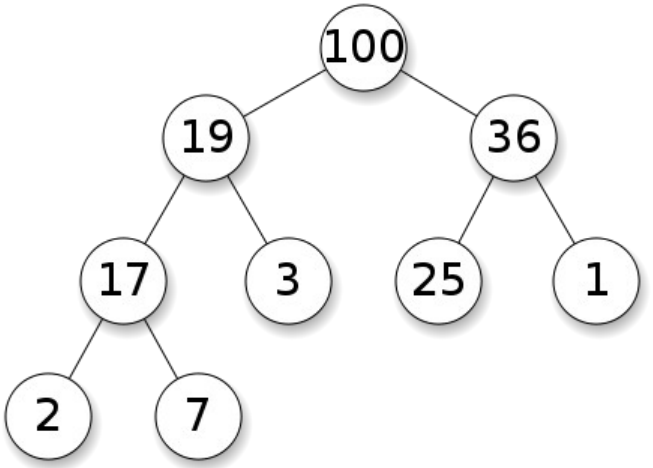
Heaps

- ***Heap***: binary tree structure with *heap-order property*
  - The key of every non-root node is **greater than** or equal to the key of that node's parent
  - i.e., the values increase as you move down the tree
- For the sake of efficiency, we also want the tree to be *complete*
  - All levels except the lowest have the maximum number of nodes possible
  - The last level's nodes are as far to the left as possible
  - The height of a complete tree with *n* items is *floor(log n)*
  - Careful! This is unrelated to being *full* or *proper*

Heaps

- Heaps can be minimum-based or maximum-based
- For the latter, the heap-order property is reversed:
  - The key of every non-root node is **less than** or equal to the key of that node's parent





Potential Confusion

- Priority queues vs. heaps
  - Priority queue is an ADT, heaps are an implementation
  - Similar to regular queues (ADT) vs. arrays (implementation)
  - Or to sets (ADT) vs. arrays or skip lists (implementations)
- Heap data structure vs. "the heap"
  - "The heap" is a common name for a pool of memory available for dynamic allocation
  - Usually managed by the operating system
  - Often grows upward in memory while the stack grows downward

Heap Operations

- Adding an item
  - Add at leftmost open position on the lowest level
    - Create a new level if the lowest level is full
  - Swap the new item upward to maintain the heap-order property
    - Start at the new node
    - If the current node's key is smaller than its parent's key, swap them and move up, repeating the process as necessary
    - Called *up-heap bubbling*
      - also: *sifting up*, or *percolating up*, etc.
      - not really related to bubble sort!

Heap Operations

- Removing minimal item
  - Remove root
  - Move the last item (rightmost item on the lowest level) into the root position
  - Swap the new root downward to maintain the heap-order property
    - Start at the new root
    - If the key of either child is smaller than the current node's key, swap the current node with the minimal child node and move down, repeating the process as necessary
    - Called down-*heap bubbling, sifting down*, or *trickling down*

Heap Operations

- Adding an item is *O(log n)*
  - May have to swap elements all the way up the heap
- Removing the minimum key is *O(log n)*
  - May have to swap elements all the way down the heap
- These are worst case bounds for a linked-tree implementation
  - Complication: keeping track of the last element
  - Also have to keep parent pointers
- These are amortized average case bounds for an array-based implementation
  - Most common method

Heap Implementation

- Because heaps are *complete* trees, there is a very convenient array-based representation (no extra space required!)
- Imagine flattening out the tree by doing a breadth-first traversal
- Concept: level numbering
  - Assign each node in the tree an index
  - The root is index 0
  - The left subchild of node *k* is index *2k+1*
  - The right subchild of node *k* is index *2k+2*
  - The parent of node *k* is at index *floor((k-1) / 2)*
  - The last element is at index *size-1*
  - The next open slot is at index *size*

Heap Implementation

- For complete trees, this scheme allows us to lay out the tree in a contiguous linear array
  - O(1) access to the root
  - We still have O(1) access to parents and children of a given index
  - We also have O(1) access to the last element in the heap as long as we track the size of the heap

Next Time

- Using heaps for sorting
  - “Heap sort”

http://visualgo.net/heap.html