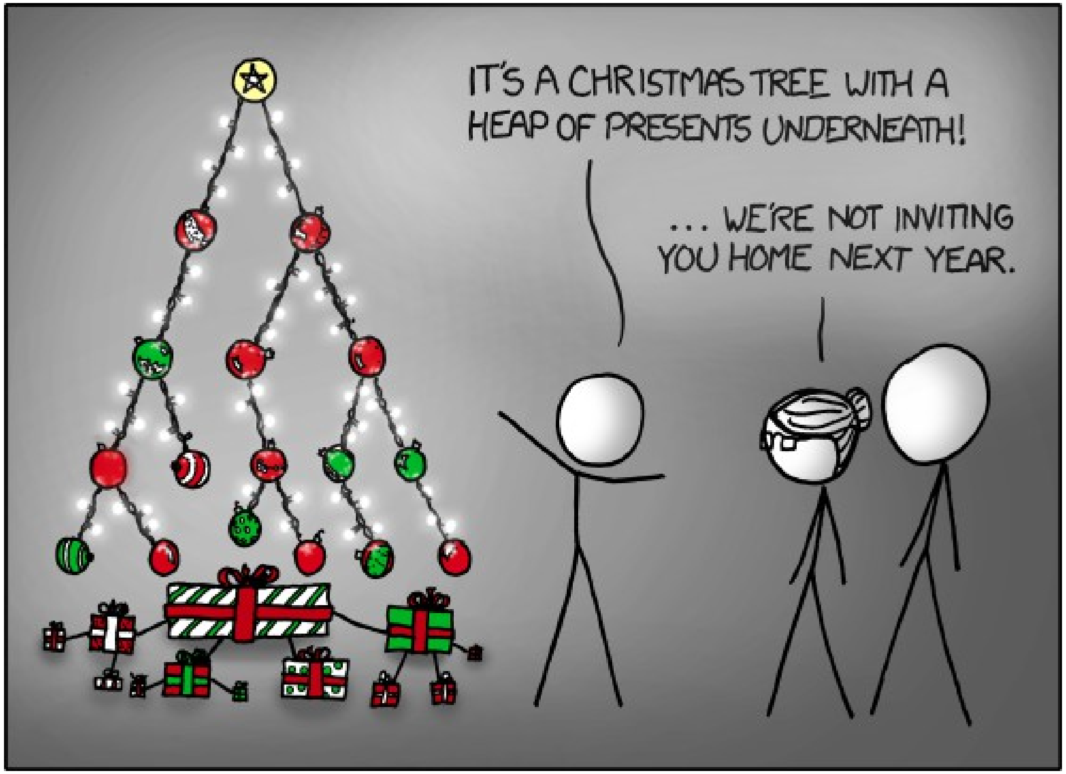
CS 240
Fall 2014

Mike Lam, Professor



Heap Sort

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
- Priority Queue ADT operations:
  - P.add(k,v)
  - P.remove_min()
  - P.min()
  - P.is_empty()
  - len(P)

Priority Queues

- Sorting using PQs
  - Add all the items to the PQ
  - Remove all the items
    - In sorted order
- Unsorted list implementation
  - Similar to selection sort; phase 1 is $O(n)$ and phase 2 is $O(n^2)$
- Sorted list implementation
  - Similar to insertion sort; phase 1 is $O(n^2)$ and phase 2 is $O(n)$
- Heap implementation
  - New sorting algorithm: "heap sort"
  - Not divide-and-conquer, but still $O(n \log n)$

Heaps for Sorting

- Linked-based heap implementation
  - Requires *O(n)* extra memory
  - Can use min or max heaps
  - Add operations: *O(n log n)*
  - Remove operations: *O(n log n)*
- In-place array heap implementation
  - No extra memory required
  - Need to use max heaps
  - Add operations: *O(n log n)* or *O(n)*
  - Remove operations: *O(n log n)*

Heap Implementation

- Because heaps are *complete* trees, there is a very convenient array-based representation
- Breadth-first traversal (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 $floor((k-1) / 2)$

Heap Sort

- Basic idea: build heap in-place then repeatedly remove max item
- Phase 1 ("heapification")
  - Start with single-item max heap w/ first item in list
  - Add each subsequent item to the heap
    - Up-heap or down-heap bubbling
- Phase 2 (sorting)
  - Repeatedly remove the maximum item and storing it at the end of the list in a down-ward growing sorted region
  - Down-heap bubbling to restore heap-order property

Heap Sort

```python
def _up_heap(items, i):
    """ Perform up-heap bubbling, starting at index i."""
    if i > 0:
        p = (i-1)//2
        if items[i] > items[p]:
            items[i], items[p] = items[p], items[i]
            _up_heap(items, p)      # tail recursion

def _down_heap(items, i, n):
    """ Perform down-heap bubbling on an n-element heap, starting at index i."""
    lc = 2*i+1
    rc = 2*i+2
    max_idx = i
    if lc < n and items[lc] > items[max_idx]:  # check left child
        max_idx = lc
    if rc < n and items[rc] > items[max_idx]:  # check right child
        max_idx = rc
    if max_idx != i:                           # swap
        items[i], items[max_idx] = items[max_idx], items[i]
        if max_idx < n:
            _down_heap(items, max_idx, n)      # tail recursion
```

Heap Sort

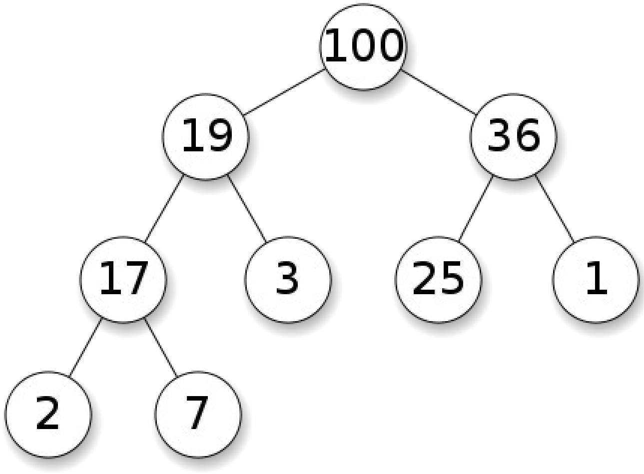
```
def heap_sort(items):
    """ Sort the provided Python list in-place using heap sort."""
    length = len(items)

    # build heap
    for j in range(1, length):
       _up_heap(items, j)

    # build sorted list
    for j in range(length-1, 0, -1):
        items[0], items[j] = items[j], items[0]    # extract max
        _down_heap(items, 0, j)
```

Example

- List: `[17, 25, 100, 2, 3, 36, 1, 7, 19]`

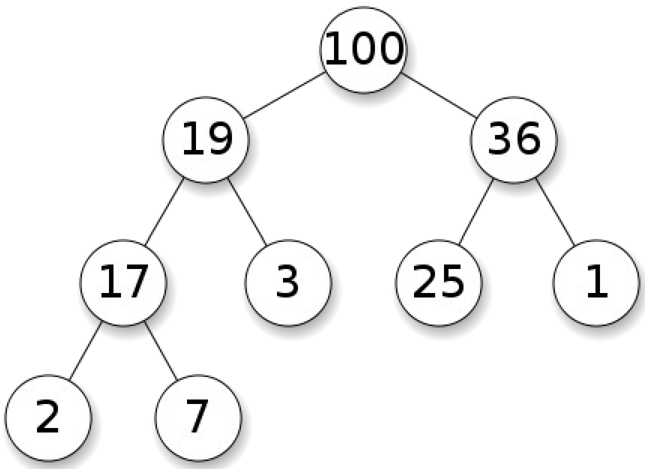
Example

- List: `[17, 25, 100, 2, 3, 36, 1, 7, 19]`
- Heap: `[100, 19, 36, 17, 3, 25, 1, 2, 7]`



Example

- List: `[17, 25, 100, 2, 3, 36, 1, 7, 19]`
- Heap: `[100, 19, 36, 17, 3, 25, 1, 2, 7]`
- Final: `[1, 2, 3, 7, 17, 19, 25, 36, 100]`



Heap Sort Analysis

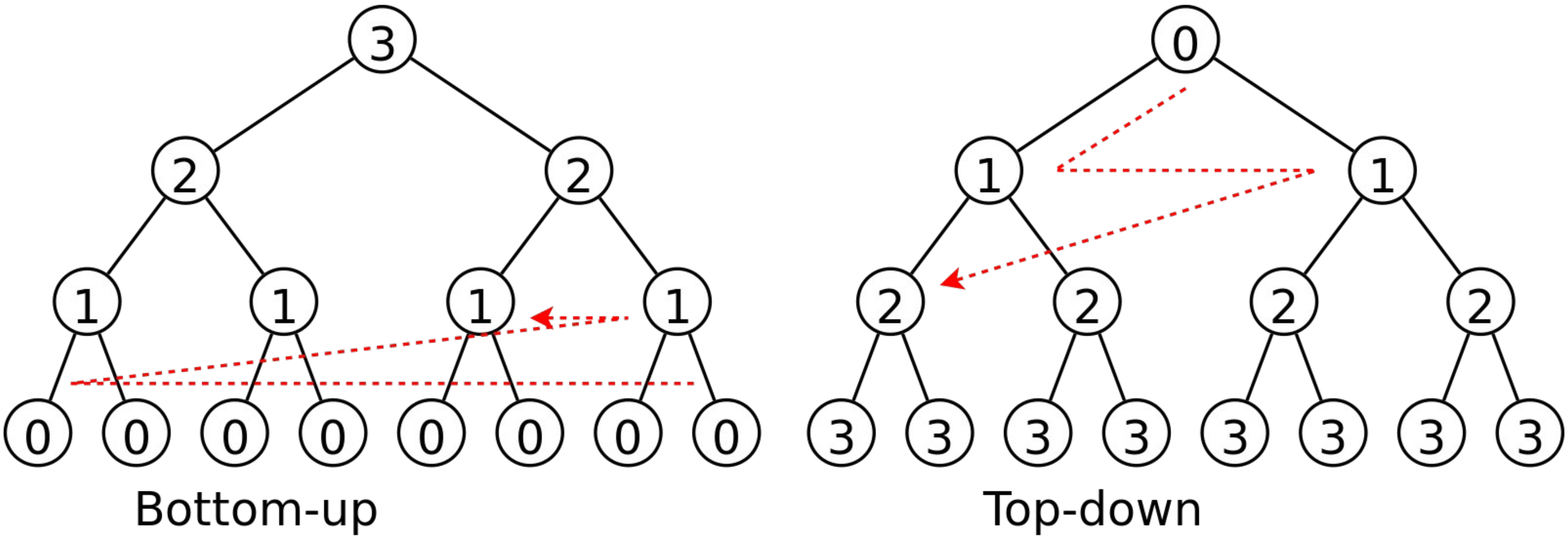
- Phase 1 (heap grows):
  - If each add() operation requires *O(log n)* time, this phase will require *O(n log n)* time
  - If we can argue that each add() operation requires only *O(1)* time on average, this phase will require *O(n)* time
- Phase 2 (heap shrinks):
  - Each remove_max() operation requires *O(log n)* time, so this phase will require *O(n log n)* time

Heapification

- One option: up-heap bubbling
  - Bubble up each newly added item to preserve heap-order property
  - Worst-case running time: *O(n log n)*
- Another option: down-heap bubbling
  - Possible when we have all elements in advance
    - Bottom-up heap construction
  - Bubble down from each non-leaf node
    - More nodes belong near the bottom of the tree, so this is better in the long run (formal argument in 9.3.6)
  - Worst-case running time: *O(n)*

Heapification

- Benefit of bottom-up construction



The number in the circle indicates the maximum times of swapping required when adding the node to the heap.

Image taken from: `https://en.wikipedia.org/wiki/Heapsort`

Heap Sort

- Worst case: O(n log n)
- In-place
- Not stable
  - Up-heap and down-heap bubbling does not preserve ordering of equal elements
- No improvement for nearly-ordered lists
  - Still builds heap, re-ordering elements twice
- However, no pathological cases
- Good alternative to quick sort in certain cases
  - Example: intro sort

Heap Sort

- Good example of CS 240 cross-cutting
- Abstract data type (priority queue) to solve problem
  - Sorting data
- Concrete data structure (heap) to implement ADT w/ certain properties
  - No additional memory
  - *O(1)* access to parents and children
  - *O(log n)* additions and removals
- Big picture: clever data structure enabling an efficient algorithm