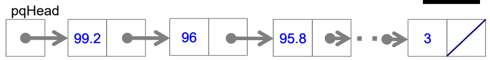
CS165: Priority Queues, Heaps

Prichard Ch. 12

Priority Queues

- Characteristics
  - Items are associated with a Comparable value: priority
  - Provide access to one element at a time - the one with the highest priority
- offer(E e) and add(E e) – inserts the element into the priority queue based on the priority order
- remove() and poll() – removes the head of the queue (which is the highest priority) and returns it

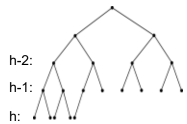
PQ – Reference-based Implementation



- Reference-based implementation
  - Sorted in descending order
    - Highest priority value is at the beginning of the linked list
    - `remove()` returns the item that `pqHead` references and changes `pqHead` to reference the next item.
    - `offer(E e)` must traverse the list to find the correct position for insertion.

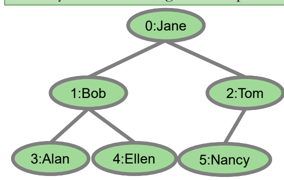
Complete tree definition

- Complete binary tree of height h
  - zero or more rightmost leaves not present at level h
- A binary tree T of height h is complete if
  - All nodes at level h – 2 and above have two children each, and
  - When a node at level h – 1 has children, all nodes to its left at the same level have two children each, and
  - When a node at level h - 1 has one child, it is a left child
  - So the leaves at level h go from left to right



Complete Binary Tree

Level-by-level numbering of a complete binary tree, NOTE 0 based!



What is the parent child index relationship?

*left child i: at 2*i+1*

right child i: at 2(i+1)*

parent i: at (i-1)/2

So we can store a complete binary tree in an array!!

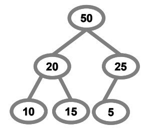
Heap - Definition

- A maximum heap (maxheap) is a ***complete* binary tree** that satisfies the following:
  - It is a leaf, or it has the heap property:
    - Its root contains a key greater or equal to the keys of its children
    - Its left and right sub-trees are also maxheaps
  - A minheap has the root less or equal children, and left and right sub trees are also minheaps

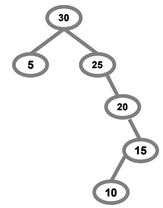
maxHeap Property Implications

- Implications of the heap property:
  - The root holds the maximum value (global property)
  - Values in descending order on every path from root to leaf
- A Heap is NOT a binary search tree, as in a BST the nodes in the right sub tree of the root are larger than the root

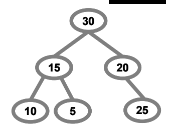
Examples



Satisfies heap property AND Complete

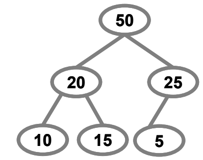


Satisfies heap property BUT Not complete



Does not satisfy heap property AND Not complete

Array(List) Implementation



0	**50**
1	**20**
2	**25**
3	**10**
4	**15**
5	**5**

Array(List) Implementation

- Traversal:
  - Root at position 0
  - Left child of position i at position 2*i+1
  - Right child of position i at position 2*(i+1)
  - Parent of position i at position (i-1)/2
    (int arithmetic **truncates**)

Heap Operations - `heapInsert`

- **Step 1**: put a new value into first open position (maintaining completeness), i.e. at the end
- but now we potentially violated the heap property, so:
- **Step 2**: bubble values up
  - **Re-enforcing the heap property**
  - Swap with parent, if new value > parent, until in the right place.
  - The heap property holds for the tree below the new value, when swapping up

Swapping up

- Swapping up enforces heap property for sub tree below the new, inserted value:

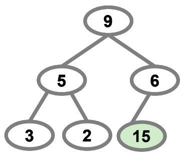
x
y new

- if (new > x) swap(x,new)

new
y x

x>y, therefore
new > y

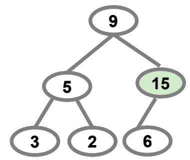
Insertion into a heap (Insert 15)



Insert 15

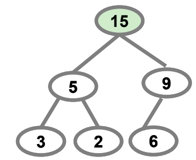
bubble up

Insertion into a heap (Insert 15)



bubble up

Insertion into a heap (Insert 15)



Heap operations – `heapDelete`

- **Step 1**: remove value at root (Why?)
- **Step 2**: substitute with rightmost leaf of bottom level (Why?)
- **Step 3**: bubble down
  - Swap with **maximum** child as necessary, until in place
  - each bubble down restores the heap property for the max child
  - this is called **HEAPIFY**

Swapping down

- Swapping down enforces heap property at the swap location:

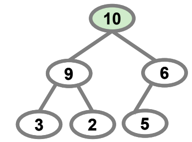
new
y x

- new<x and y<x:
  swap(x,new)

x
y new

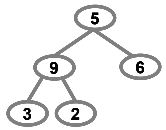
x>y and x>new

Deletion from a heap

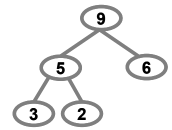


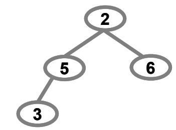
Delete 10
Place last node in root

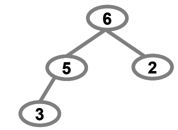
bubble down
heapify
draw the heap



delete again
draw the heap







HeapSort

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

HeapSort

- Alternative method (in-place):
  - **buildHeap:** 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
      - Leaves are already heaps
      - Start at last internal node
      - **Go backwards** calling `heapify` with each internal node
  - Iteratively swap the root item with last item in unsorted portion and rebuild

Building the heap

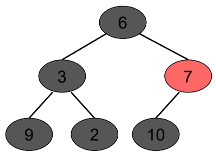
```
buildheap(n){
    for (i = (n-2)/2 down to 0)
    //pre: the tree rooted at index is a semiheap
    //i.e., the sub trees are heaps
    heapify(i); // bubble down
    //post: the tree rooted at index is a heap
}
```

- WHY start at (n-2)/2?
- WHY go backwards?

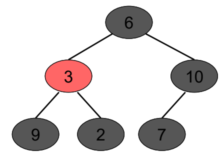
- The whole method is called **buildHeap**
- One bubble down is called **heapify**

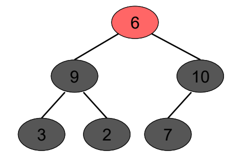
6	3	7	9	2	10

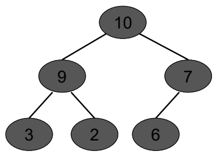
Draw as a Complete Binary Tree:



Repeatedly heapify, starting at last internal node, going backwards



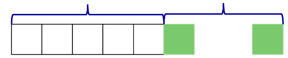




10	9	7	3	2	6

In place heapsort using an array

- First build a heap out of an input array using buildHeap(). See previous slides.
- Then partition the array into two regions; starting with the full heap and an empty sorted and stepwise growing sorted and shrinking heap.



Do it, do it

HEAP

10	9	6	3	2	5
9	5	6	3	2	10
6	5	2	3	9	10
5	3	2	6	9	10
3	2	5	6	9	10
2	3	5	6	9	10
2	3	5	6	9	10

SORTED