Data Structures Toolbox – The Story So Far

Fundamental data structures and algorithms –

- collections – List, Stack, Queue
  - implementations: array (circular array), linked list (tail pointer)
  - can achieve O(1) operations for Stack, Queue
- searching and lookup
  - sequential search, binary search
  - Map/Dictionary
    - implementations: array, linked list (sorted, unsorted); balanced search tree (AVL, 2-4, …)
    - can achieve O(log n) operations using balanced search trees

    can we do better than O(log n)?
  - Set
    - can implement as Map/Dictionary where element is both key and value
- sorting
  - insertion sort, selection sort, mergesort, quicksort, ...
  - PriorityQueue
    - implementations: array, linked list (sorted, unsorted)
    - can have O(1) insert/remove but the other operation is O(n)

    can we do better than this?

ADTs – PriorityQueue

PriorityQueue	maintain removal order when there are out-of-order additions	• insert(x,p) – insert elt x with priority p • findMin() or findMax() – find elt with min/max priority • deleteMin() or deleteMax() – remove (and return) elt with min/max key note: a PQ is typically either a min-PQ or a max-PQ – it does not support both min and max operations simultaneously

Basic Implementation of PriorityQueue

operation	array - unsorted	array – reverse sorted	linked list - unsorted	linked list - sorted
find min	**O(1)** – store index of min	**O(1)** – in last slot	**O(1)** – store node with min	**O(1)** – at head
insert	**O(1)** – add at end	**O(n)** – binary search + shift	**O(1)** – add at head	**O(n)** – sequential search
remove min	**O(n)** – delete (swap) + update min index	**O(1)** – in last slot	**O(n)** – update min node	**O(1)** – at head

Tradeoff: fast insert or fast remove, but not both.

Can we do better?

Priority Queue Implementation

Can we do better?

Observations.

- either insert or remove takes O(n) time
  - would be nice to reduce this!
- there is an ordering of the elements (by priority)
  - sorted order is exploited in remove min but isn't helpful for insert (binary search in array is offset by having to shift to make room)

Recall: balanced search trees

- insert/remove is O(log n)

Priority Queue Implementation

operation	array – unsorted	array – reverse sorted	balanced search tree
find min	**O(1)** – store index of min	**O(1)** – in last slot	
insert	**O(1)** – add at end	**O(n)** – binary search + shift	
remove min	**O(n)** – shift + update min index	**O(1)** – in last slot	

Priority Queue Implementation

operation	array – unsorted	array – reverse sorted	balanced search tree
find min	**O(1)** – store index of min	**O(1)** – in last slot	**O(1)** – store min node
insert	**O(1)** – add at end	**O(n)** – binary search + shift	**O(log n)** – update tree structure
remove min	**O(n)** – shift + update min index	**O(1)** – in last slot	**O(log n)** – update tree structure + update min node

Tradeoff: worst-case time reduced from O(n) to O(log n), but have lost O(1) insert or remove.

Priority Queue Implementation

Can we do better?

Observation.

- O(log n) for insert, remove min is due to updating the tree structure

operation	balanced search tree
find min	**O(1)** – store min node
insert	**O(log n)** – update tree structure
remove min	**O(log n)** – update tree structure + update min node

Priority Queue Implementation

Can we do better?

Consider the essence of the problem –

- don't necessarily need a full sorted order at any one point, just the ability to get the min
- adding an element is only a small incremental change
- only a specific element is removed (the min)

Strategy –

- maintain a *partial order* of elements
  - stronger than unsorted (to improve on updating min) but not as strong as sorted (to improve on insert performance)
  -

Priority Queue Implementation

How to implement?

Observations.

- balanced search tree = binary (or multiway) tree +
  ordering constraint to aid in search +
  structural constraint to aid in efficiency

Can we do something along these lines for PQs?

- but with a weaker ordering constraint since search only needs to find the min

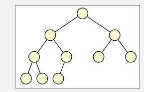
Heaps

The idea: a heap is a

- binary tree +
- an ordering property to aid in searching +
- a structural property to aid in efficiency of implementation

Heap ordering property: (min heap)

- every node's key is ≤ the keys of its children
  - smallest element is at the root



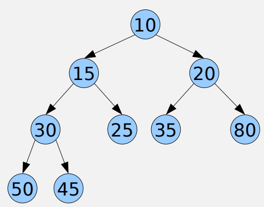
Structural constraint:

- the tree is a complete binary tree
  - the only empty spots are the rightmost elements in the last level

Note: the height of a complete binary tree is O(log n).

Heaps – FindMin

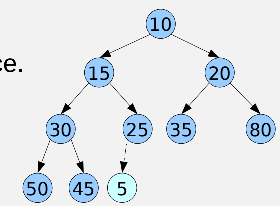
The ordering property means that the min element is at the root.



Heaps – Insertion

The structural property means insertion can only occur in one place.

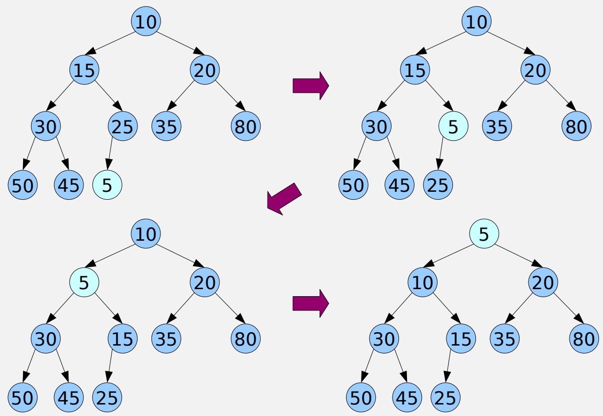
Strategy: insert in the only possible place, then fix up the ordering property if broken.



Thus:

- insert element in the first available slot
- “bubble up” until ordering property is restored
  - element is only out of order with respect to parent

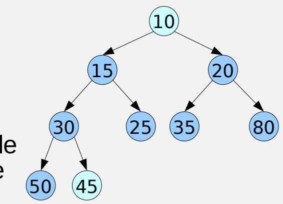
Heaps – Insertion



Heaps – RemoveMin

The structural property means removal can only occur from one place.

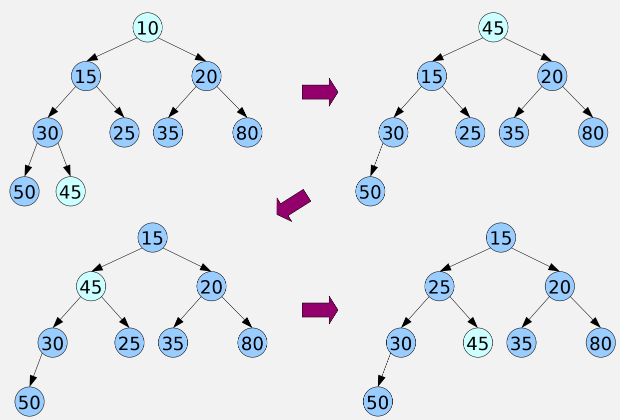
Strategy: swap element to delete with the element in the only possible position for removal, then fix up the ordering property if broken.

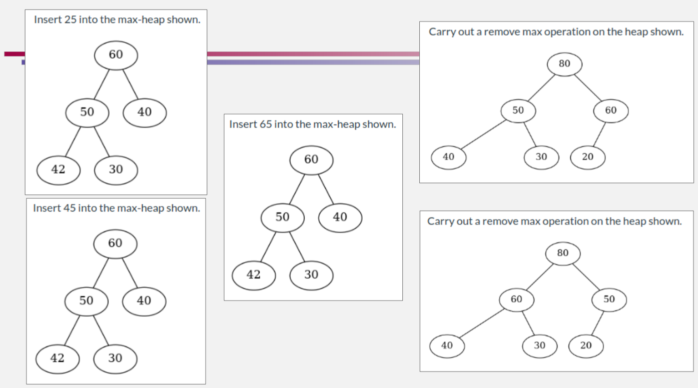


Thus:

- swap root with last slot
- remove element in last slot
- “bubble down” until ordering property is restored
  - swap with smaller child

Heaps – RemoveMin





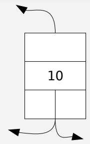
Strategy –

- insert/remove as dictated by the structural property
- fix up the ordering property if broken by bubbling element down or up

Heaps – Implementation

The standard implementation for binary trees is a linked structure.

- pointer to root node
- tree node stores element + pointers to parent, left child, right child



Running time and space –

- find min is O(1) – min element is at the root
- inserting and removing require knowing the location of the last element in the tree
  - O(n) to find – don't know which child will have the last leaf
  - solution – maintain a `last` pointer!
  - updating `last` after insertion/removal can require O(log n) time
    - bubbling is already O(log n) so this is just a constant factor
- space is O(n)
  - but there is overhead of three pointers per element (same as BST)

Heaps – Implementation

Assessment –

- same big-Oh running time as balanced search tree
- space is similar
  - need parent pointers, though many balanced search tree implementations have overhead beyond the binary tree structure
- implementation is simpler

Can we do better?

- reduce space overhead
  - array eliminates overhead of pointers...if structural information can be encoded in the indices
- reduce time to build heap from n elements
  - O(n log n) for n insert operations

Heaps – Implementation

The alternative to a linked structure is an array.

- calculate parent/child index instead of storing
  - root stored at slot 0
  - left child of node with index i is in slot 2i+1, right child in slot 2i+2
  - parent of node with index j is in slot (j-1)/2

Running time and space –

- find min is O(1) – min element is in slot 0
- inserting and removing require knowing the location of the last element in the tree
  - at `size-1` (i.e. maintain a `last` index)
  - updating this value after insertion/removal takes only O(1) time
    - just increment or decrement
- space is O(n)
  - only have to store elements (no additional pointers)
  - complete binary tree fills consecutive slots – no gaps

Heaps – Implementation

Arrays are the traditional implementation for heaps.

- same big-Oh as linked structure, but avoids space overhead of parent/child pointers

Running time:

- insert – O(log n)
  - O(1) to put element in array, update `last`
  - O(log n) to bubble up
- remove min – O(log n)
  - O(1) to swap with last, remove last, update `last`
  - O(log n) to bubble down
- find min – O(1)
  - min element is at root (index 0)

Heaps – Implementation

We didn't improve the big-Oh over the balanced search tree implementation for PQs.

But –

- reduced storage overhead (no parent, child pointers)
- reduced difficulty of implementation
  - array + bubble up, bubble down vs. linked structure + balanced search tree operations
  - traded maintaining 'min' reference for incrementing/decrementing 'last' index
- reduced constant factors
  - traded O(log n) maintenance of 'min' reference for O(1) maintenance of 'last' index

Building a Heap

How to build a heap?

- repeatedly insert each element $\sum_{i=0}^{n-1} \log(i)$ = Θ(n log n)

Building a Heap

Or...if you already have an array of elements...

- for any n elements in an array, the heap order property is at most broken only for the first n/2 elements

Heapify idea.

- for each index n/2 down to 0, bubble down that element

Running time.

- bubble down takes O(h) time
  - n/2 elements are leaves (already in place – no change)
  - n/4 elements are one level above leaf (at most 1 swap)
  - n/8 elements are two levels above leaf (at most 2 swaps)
  - …
- = $\sum_{i=1}^{\log n} (i-1)(\frac{n}{2^i})$ = n Θ(1) = Θ(n)