Chapter 12 - Heaps

Introduction

- Heaps are largely about priority queues.
- They are an alternative data structure to implementing priority queues (we had arrays, linked lists…)
- Recall the advantages and disadvantages of queues implemented as arrays
- Priority queues are critical to many real-world applications.

Introduction

- First of all, a heap is a kind of tree that offers both insertion and deletion in O(log$_2$n) time.
- Fast for insertions; not so fast for deletions.

Introduction to Heaps

- Characteristics:
  - 1. A heap is ‘complete’ (save for the last row – going left to right – see figure 12.1, p. 580)
  - 2. usually implemented as an array
  - 3. Each node in a heap satisfies the ‘heap condition,’ which states that every node’s key is larger than or equal to the keys of its children.
  - ➔ The heap is thus an abstraction; we can draw it to look like a tree, but recognize that it is the array that implements this abstraction and that it is stored and processed in primary memory (RAM).
  - No ‘holes’ in the array.

Priority Queues, Heaps, and ADTs

- Heaps are mostly used to implement priority queues.
- Again, a heap is usually implemented as an array.

“Weakly Ordered”

- We know how a binary search tree is developed – with lesser keys to the left; greater keys to the right as we descend.
  - Because of this, we have nice, neat algorithms for binary search trees.

- Here: No Strong Ordering:
  - But for nodes in a heap, we don’t have this strong ordering - and this can cause us some difficulty.

- Cannot assert much:
  - We can only assert as one descends in the heap, nodes will be in descending order (see rule 3)
  - Equivalently, nodes below a node are <= the parent node.

- Weakly-ordered:
  - Heaps are thus said to be weakly ordered…

Weakly Ordered – More

- No Convenient Search Mechanism:
  - Because of weak ordering, there is no convenient searching for a specified key as we have in binary search trees.
  - Don’t have enough info at a node to decide whether to descend left or right.
- Delete:
  - So to delete a node with a specific key there are issues
  - No real slick way to find it.
- Randomness:
  - So, a heap’s organization approaches ‘randomness.’
- Sufficient Ordering: Yet, there is ‘sufficient ordering’ to allow
  - quick removal (yes, a delete) of the maximum node and
  - fast insertion of new nodes.
- ➔ As it turns out, these are the only operations one needs in using a heap as a priority queue.
- We will discuss the algorithms later…

Removal

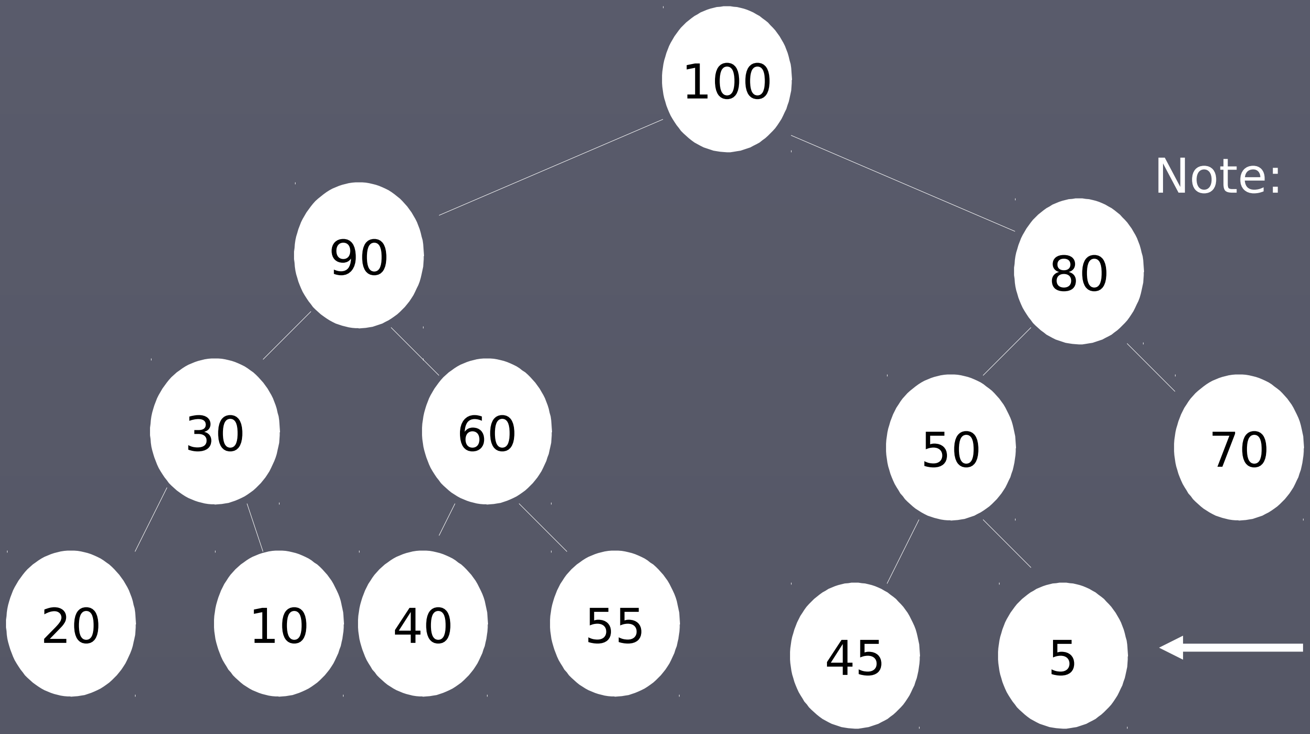
- Removal is easy:
  - When we remove from a heap, we always remove the node with the <u>largest</u> key.
  - Hence, removal is quite easy and has index 0 of the heap array.
  - maxNode = heapArray[0];
- But tree is then not complete:
  - But once root is gone, tree is not complete and we must fill this cell.
- Now this becomes interesting…

Removal of “maxNode”

- Move ‘last node’ to root.
  - Start by moving the ‘last node’ into the root.
  - The ‘last’ node is the rightmost node in the lowest occupied level of the tree.
  - This also corresponds to the last filled cell in the array (ahead).

- Trickle-down:
  - Then trickle this last node down until it is below a larger node and above a smaller one.

Removal (Delete)



Note: NOT binary search tree!

Last node added

Last node added is 5.

Delete Algorithm: heapArray[0] = heapArray [n-1]

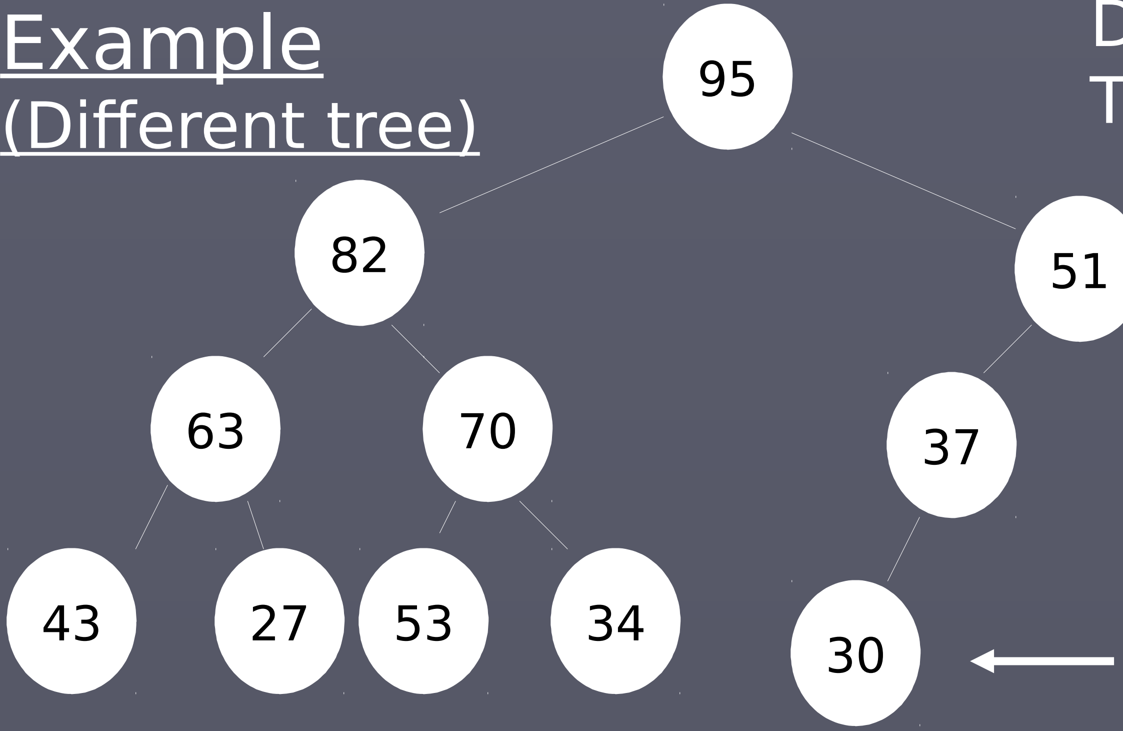
n--; // size of array is decreased by one.

Heap is represented logically as a tree.

(Tree is organized to represent a priority queue. Easy to

Example
(Different tree)

Delete (Removal)
Trickle Down

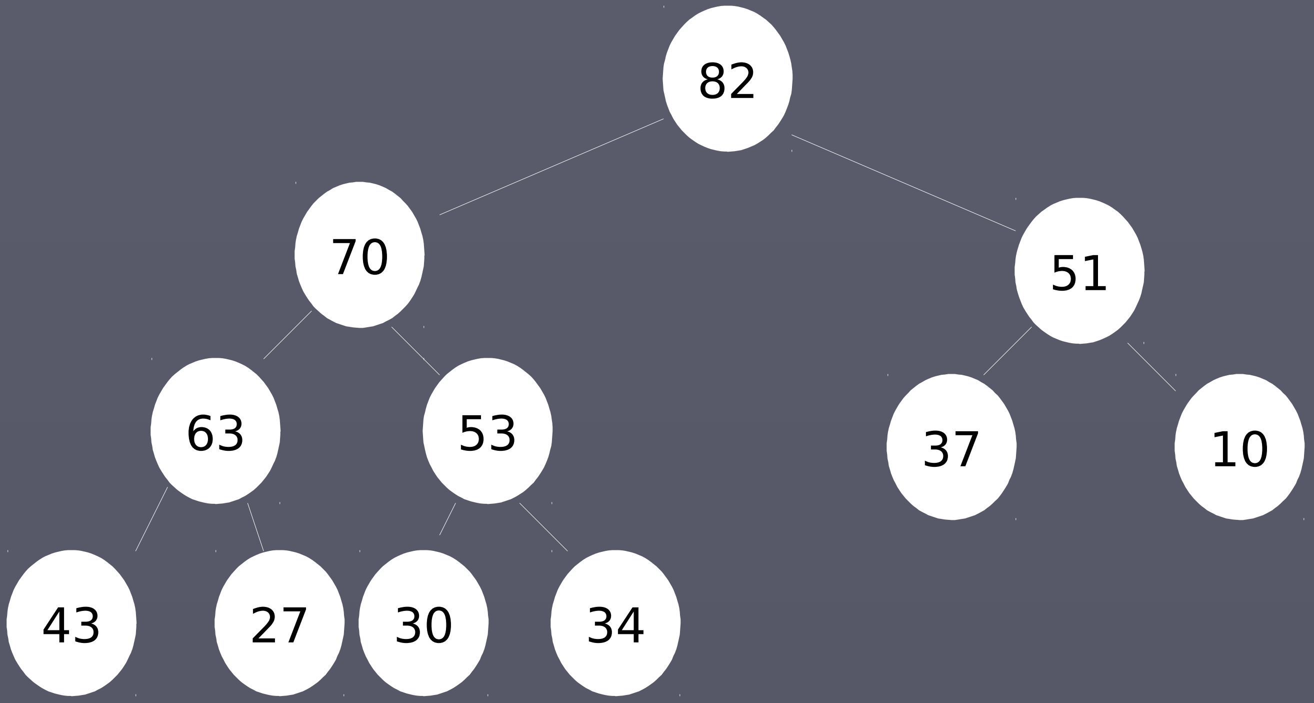


Last node added

le down swapping node w/larger child until node >= childre

e trickle down, the nodes swap out always swapping the larger node
the node we are trickling down (so as to maintain the larger nodes abov

ickle down selecting the largest child with which to swap.
ust compare, but always swap with the larger of the two.
we very well may NOT trickle down to a leaf node



he new arrangement via a delete would be as above.

Go through this…

Node 95 (the largest) is deleted.
Tree remains balanced after delete
and the rules for the heap are preserved.

(95 removed; compare 30 with 82; 82 selected and moved to root;
Compare 30 with 70. 70 moves up.
Compare 30 with 53; 53 moves up. 30 is a leaf node.)

Insertion – Trickle Up

- Pretty easy too.   Easier than Deletion.
- Insertions usually trickle up.

- Start at the bottom (first open position) via code:

  heapArray[n] = newNode;

  n++;

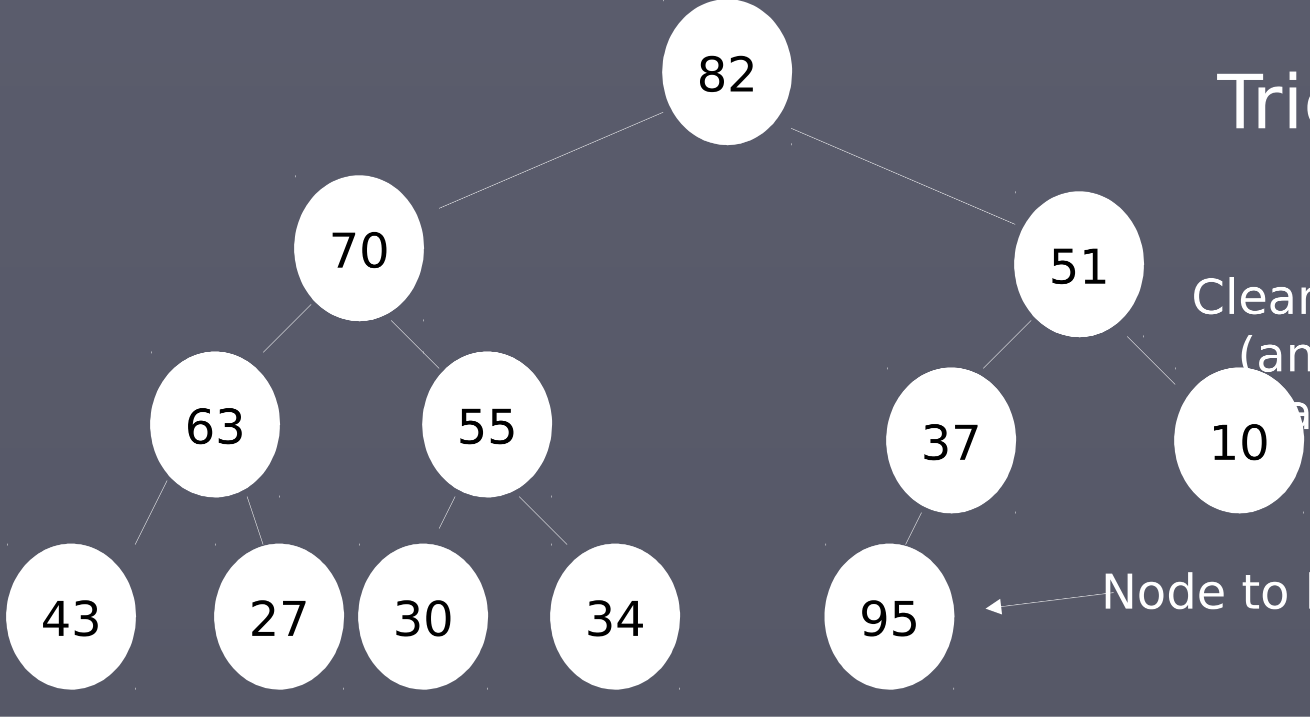
Inserting at bottom will likely destroy the heap condition.

This will happen when the new node is larger than its parent.

Trickle upwards until node is below a node larger than it and it is above a node smaller (or equal to) it.

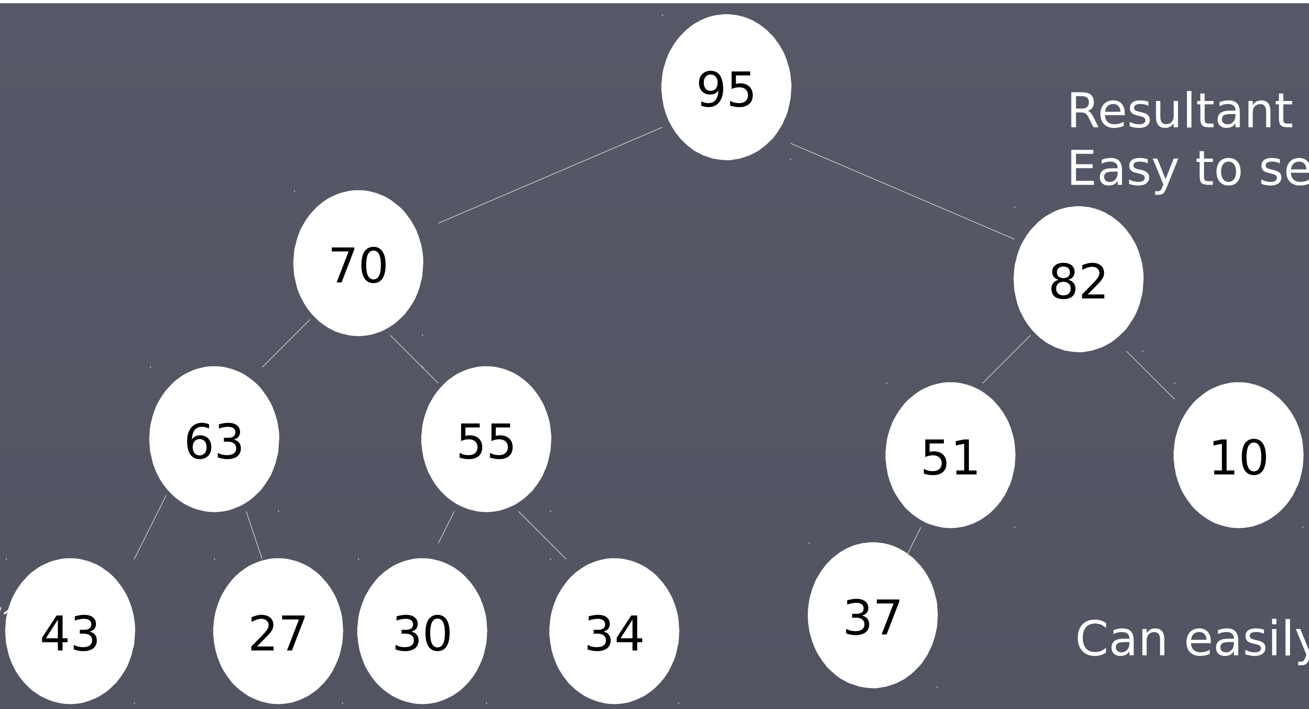
Consider the following slides:

Trickle Up



Clearly 95 must trickle up
(and appears it will end u
at the root!)

Node to be added…



Resultant heap:
Easy to see swaps…

Can easily see the swaps.

Insertion - more

- Note that this is easier because we don’t have to compare which node to swap (as we do in going down).
- We only have ONE parent, and it is equal to or larger!
- Progress until parent is larger, at whatever level that might occur.

- A side point: Clearly, the same nodes may constitute different heaps depending upon their arrival.

Insert / Delete – a ‘little bit’ of Implementation

- While some implementations actually do the swap, there’s a lot of overhead here for a large heap.
- A gain in efficiency is acquired if we substitute a ‘copy’ for a ‘swap.’ (like the selection sort…)
- Here, we copy the node to be trickled to a temp area and just do compares and copies (moves) until the right spot is found. Then we can copy (move) the node (temp) (to be trickled down) into that node location.
- More efficient than doing swaps repeatedly.

Java Code for Heaps

- Code is reasonably straightforward.
- We do, however, need to ensure that the array is not full.
- If full, what do we do?
- Likely if the array is an ArrayList or a Vector (one dimensional array), we are in good shape and can (for the first step) merely add the new entry to the end (bottom) of the Vector.
- No problems.

Heap Implementation

- Easy to discuss heaps as if heaps were implemented as trees even though the implementation has been based on an array or vector.

- As implemented with an array, we can use a binary tree, but not a binary search tree, because the ordering is not strong.
- Such a tree, complete at all times – with no missing nodes – is called a tree-heap.

Class Exercise

- Draw an appropriate array (or vector) to implement a tree-based heap whose ‘entries’ are sufficient to support
  - Inserting into the heap
  - Removal from the heap
  - Making the structure dynamic
- Pseudo-code your algorithms.

Class Exercise

- Implement a heap with a linked list.
- Draw the structure you design.
- Present pseudocode to
  - Insert into
  - Delete from
- the physical structure.

Sorting with a Heap: HeapSort: insert() and remove()

- Very easy to implement.
- We simply insert() all unordered items into the heap trickling down using an insert() routine….
- Then, repeatedly use the remove() items in sorted order….
- Both insert() and remove() operate in $O(\log_2 n)$ time and this must be applied n times, thus rendering an $O(n\log_2 n)$ time.
- This is the same as a QuickSort
- Not as fast because there are more operations in the trickledown than in the inner loop of a quicksort.
- Can be done both iteratively (using stacks) and recursively.
- Code is in your book.

Uses of Heaps

- Use of heap trees can be used to obtain improved running times for several network optimization algorithms.
- Can be used to assist in dynamically-allocating memory partitions.
- Lots of variants of heaps (see Google)

- A heapsort is considered to be one of the best sorting methods being in-place with no quadratic worst-case scenarios.
- Finding the min, max, both the min and max, median, or even the k-th largest element can be done in linear time using heaps.
- And more….