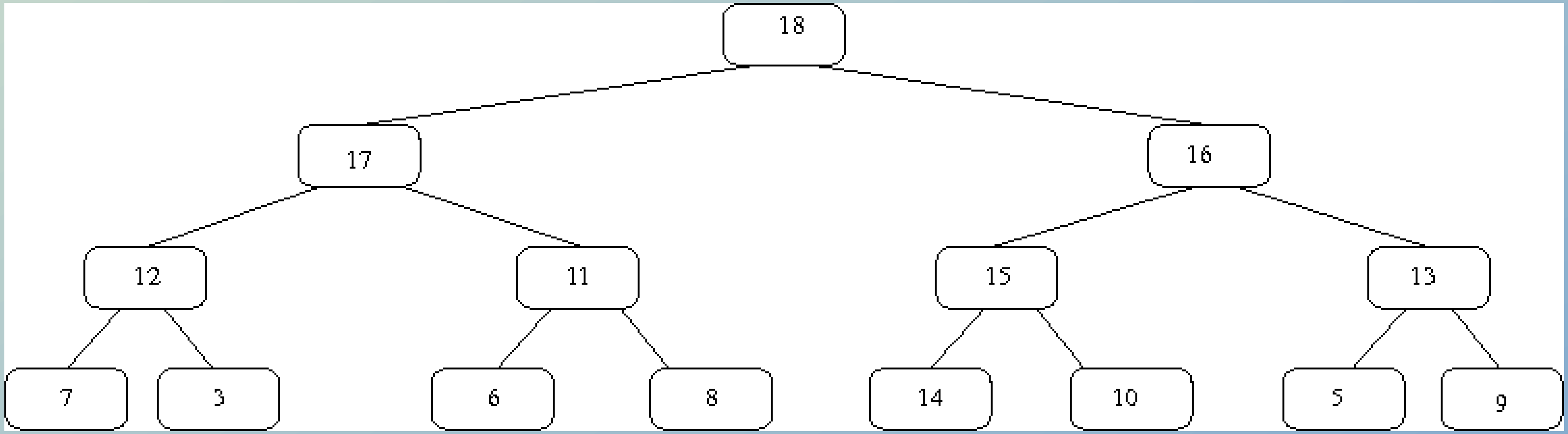
Outline: heaps

- heaps
- heap storage in arrays
- heap insertion and removal
- heap sort
- priority queues

Heaps

- a heap is a binary tree
  1. in which each node has a value greater than its children (max heap)
  2. or a value less than its children (min heap)
- this is the heap property
- unlike a binary search tree, nodes in a heap are not sorted overall
  1. instead, the heap property only insures that the largest (or smallest) value is at the top of the heap

heap example



- is this a min heap or a max heap?

heap requirements

- as well as the heap property of each node needing to be greater (less) than its children, a heap is a complete binary tree, meaning:
  - every level except for the lowest has the maximum possible number of nodes, and
  - at the lowest level, all the nodes are as far to the left as possible.
- a complete binary tree has two useful properties:
  - it is always balanced
  - its values can be stored breadth-first in an array

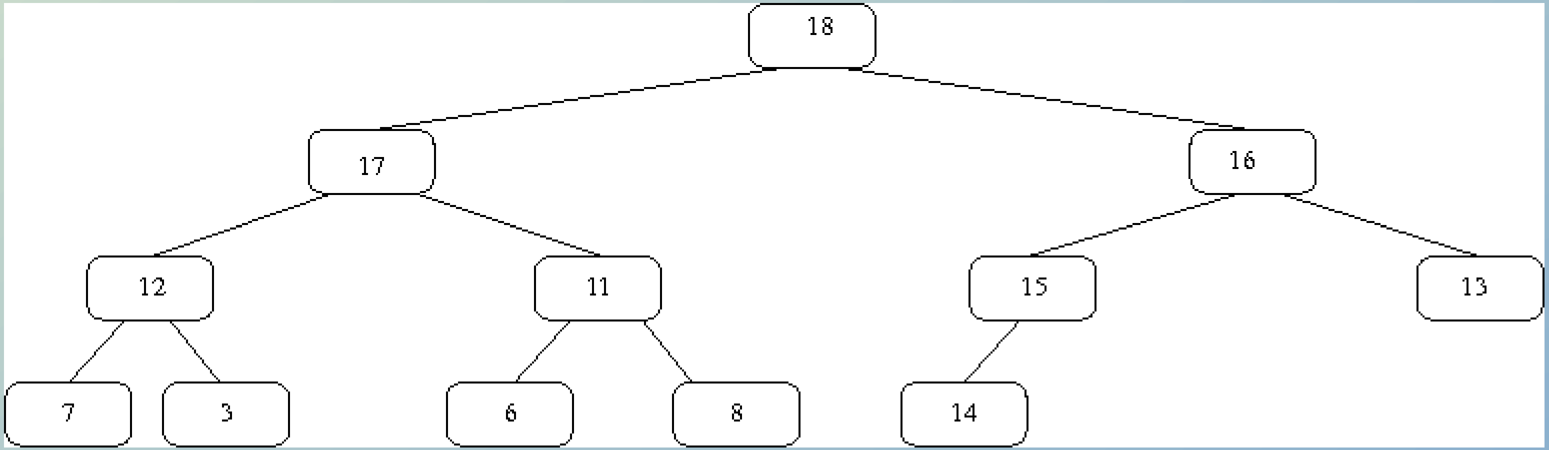
heap storage

- any complete binary tree, including any heap, can be conveniently stored in an array:
  - element 0 of the array stores the root
  - elements 1 and 2 of the array store the nodes at depth 2
  - elements 3, 4, 5, and 6 of the array store the nodes at depth 3
  - nodes at depth *d* are stored in array elements $2^{d-1}-1 \ldots 2^{d} - 2$
- conveniently, there is no need to store any references/pointers!
  - all the data is in the array
  - and we can move up to the parent or down to either child

heap properties: balanced binary tree

- a heap is always a complete binary tree
- a complete binary tree is always balanced, so that a complete binary tree of n nodes always has depth $O(\log n)$
- so for example, a heap with a million elements has depth 20

heap example



- stored in an array:

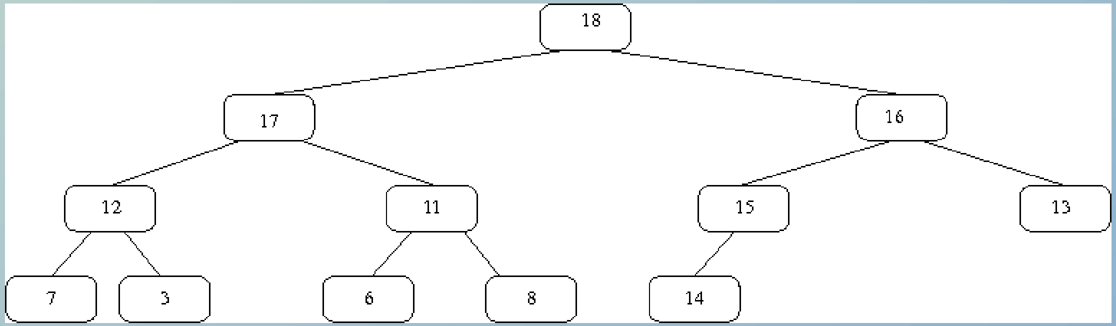
18	17	16	12	11	15	13	7	3	6	8	14
0	1	2	3	4	5	6	7	8	9	10	11

array storage of a complete binary tree: no references/pointers

- a node stored in array element *i* has:
  - its parent in array element (i - 1) / 2
  - its left child (if any) in array element 2i + 1
  - its right child (if any) in array element 2i + 2
- this means we can find (and move to) a node's parent as well as a node's children
- since a heap is a complete binary tree, the right child can only be present if the left child is also present
  - there may be one left child leaf node that does not have a right sibling

in-class exercises

- store this heap into an array



- which of the following arrays store max heaps, min heaps, or neither?

index	0	1	2	3	4	5	6	7	8	9
array 1	0	1	2	0	4	5	6	7	8	9
array 2	9	8	7	6	5	4	3	2	1	0
array 3	5	5	5	6	6	6	6	7	7	1
array 4	9	3	9	2	1	6	7	1	2	1
array 5	8	7	6	1	2	3	4	2	1	2

heap insertion

- the two heap requirements must be maintained when adding to a heap
- to maintain the complete binary tree property, the new node must be added to the right of all nodes at depth d^{max}
- or, if there already are $2^{(dmax)}$ nodes at that level, the new node should be inserted all the way to the left, making the tree deeper by one level
- either way, the new value is inserted in the array just after all elements already in the array, which takes O(1) time as long as the array is not resized
- now the tree is complete, but may not have the heap property
- to check for the heap property, compare the value in the new node with the value in the parent, and swap the two if needed to maintain the heap property
- continue with the parent's parent, all the way to the root if necessary
- now the complete binary tree also has the heap property

heap deletion

- in a max heap, the largest node is at the root of the heap
  - in a min heap, the smallest node is at the root of the heap
- the root node is removed, and replaced with the bottommost, rightmost node: the node at the end of the array – this swap takes O(1) time
- the remaining tree is complete, but may not have the heap property
- if the root node is less than either of its children, it is swapped with the largest of its children
  - in a min heap, the root node is swapped with the smallest of its children
- the operation continues with the new node
- now the complete binary tree also has the heap property

heap sort

- start at the front of an unsorted array, and build a max heap with each element in turn
  - the heap expands exactly as the unsorted part of the array shrinks
- once the max heap is built, repeatedly delete the largest element from the heap and put it in its sorted place
  - the heap shrinks exactly as the sorted part of the array grows.

partial example of heap sort: building the heap

11	18	16	12	15	17	subarray of size 1 is always a heap
18	11	16	12	15	17	swap new value with parent -- done!
18	11	16	12	15	17	new value is less than parent -- done!
18	12	16	11	15	17	swap new value with parent -- done!

partial example of heap sort: removing from the heap

18	17	16	12	11	15	
15	17	16	12	11	18	swap largest with last value
17	15	16	12	11	18	swap root with largest child -- done!
11	15	16	12	17	18	swap largest with last value
16	15	11	12	17	18	swap root with largest child
16	15	12	11	17	18	swap with only child -- done!

priority queues

- the queues studied so far were strictly FIFO
- that means objects were returned in the order inserted
- in the real world, queues may need priorities, e.g.:
    - at airport check-in, there is a special line for first-class passengers
    - if any first-class passengers are waiting, they are handled first
    - if no first-class passengers are waiting, only then will the check-in agent check in the other passengers
- likewise some kinds of traffic in a network get priority
    - e.g. identified real-time traffic on WiFi

priority queue implementations

- linked list: objects are inserted in the proper place in the list in O(n), objects are returned from the front of the list in O(1)
  - nice because insertion of high priority items is fast
- array: objects are inserted in the proper place in the array, with all other objects shifted to make room in O(n), objects are returned from the back of the array in O(1)
- binary search tree: objects are inserted into the binary search tree, using the priority as the key in O(n), objects are removed from the leftmost or rightmost node of the tree in O(n)
  - with a balanced tree algorithm is used, the times become O(log n)
- heap: objects are inserted into the heap using the priority as the key in O(log n), and removed from the top of the heap in O(log n)
  - this is the simplest algorithm with guaranteed log time operations!

priority queue performance

- if the priority queue only has a few elements, any of these implementations is fine
- however, if the priority queue might grow long, then frequent operations should be fast
- the performance depends on the algorithm, on the operation, and on the priority
- e.g., always adding something with highest priority is fast if using a linked list

priority queue in a heap

- O(log n) insertion and removal
- the priority is used as the key that determines the sorting order
- the value to be returned also has to be stored in the heap
- values with the same priority are returned in random order
  - not necessarily in FIFO order
- example: adding to a priority queue implemented as a heap the values a, b, c with priority 100 and x, y, z with priority 5
- first 3 removals give cab or acb or bca or abc or cba or bac