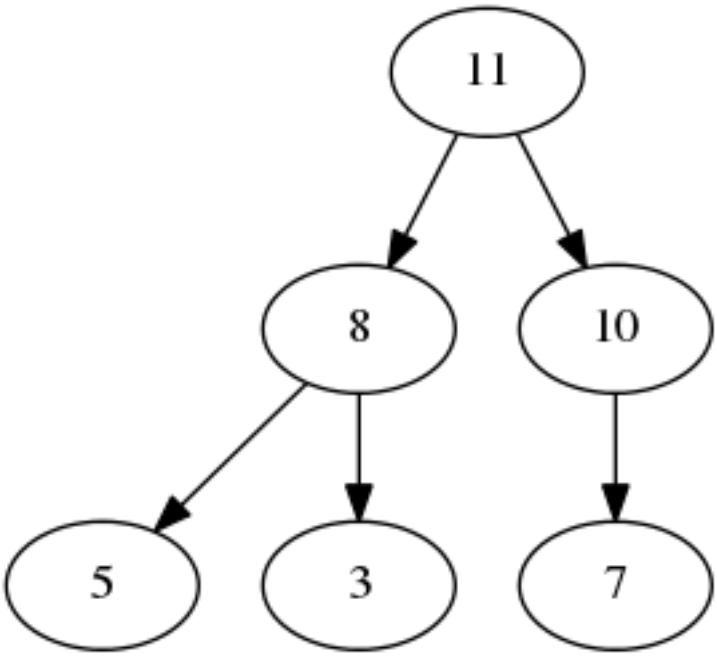
HEAPS

CS 200 RECITATION 11

Heap ADT

- A 'tree like' ADT
- Commonly binary, but does not have to be
- Always represents a complete tree
- I know this is out-of-order with the lecture, sorry.
- We are going to build one.

Example



All heaps have the heap property: Any given node's value is greater than all of it's children. The children don't have to be in any particular order, just less than the parent. This is called a Max Heap. If you invert the relationship (parent less that all children) it is a Min Heap.

Array Representation:

Heaps are commonly stored in an array. This is because they are complete trees. The tree is written into the array row by row starting with the root. The above heap can be stored in an array like so:

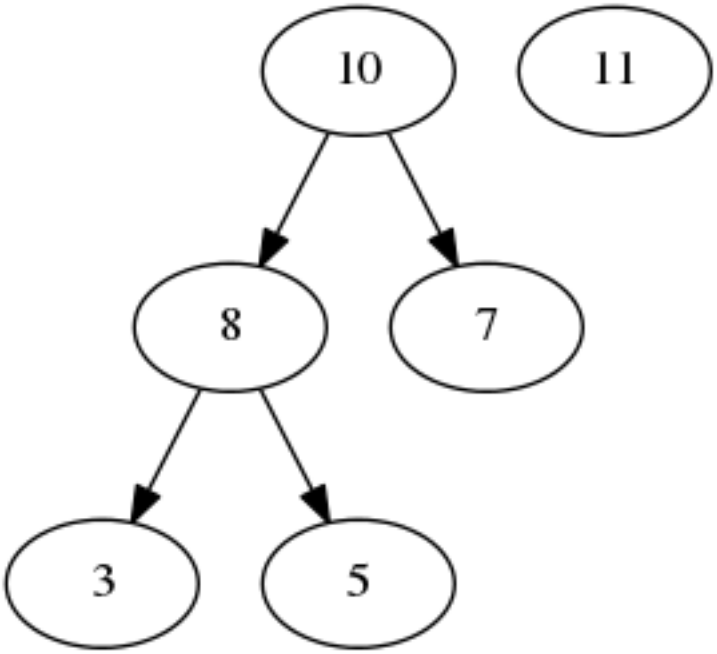
```
array = {11, 8, 10, 3, 5, 7}
```

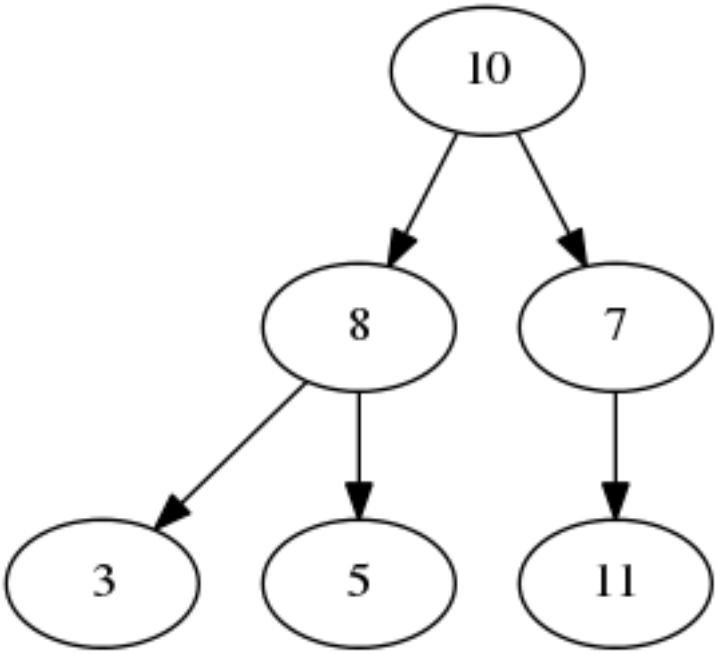
Because this representation of a complete tree (binary) has a rigid pattern, we can compute the index of the children of any particular node, or the index of a node's parent:

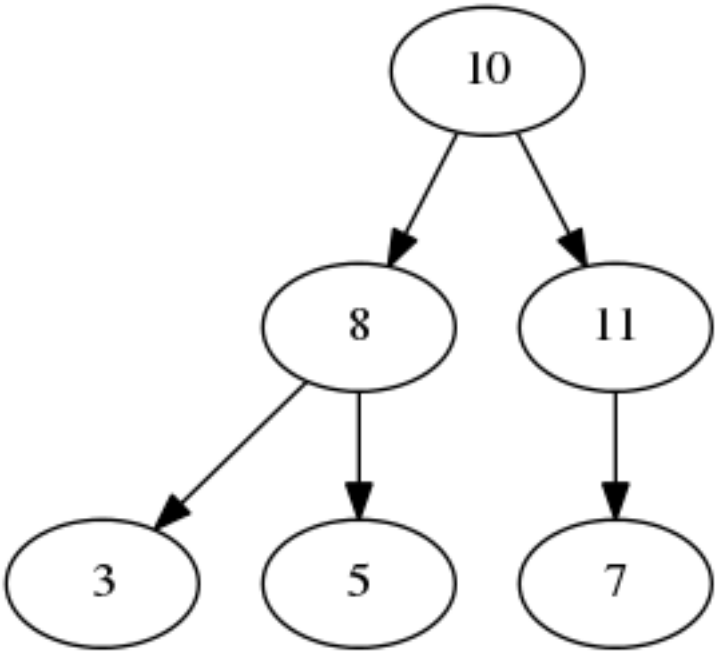
```
leftChild = (2 * parent) + 1
rightChild = leftChild + 1
parent = (self - 1) / 2 //integer math is a must here
```

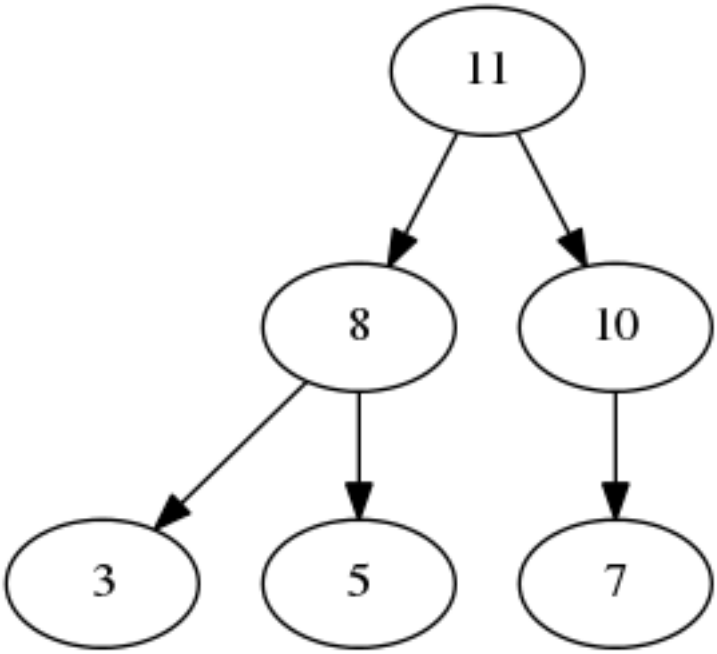
Heap Insert

- Add node to next open space in array
- Trickle up the heap if necessary (parent < node)
- example:



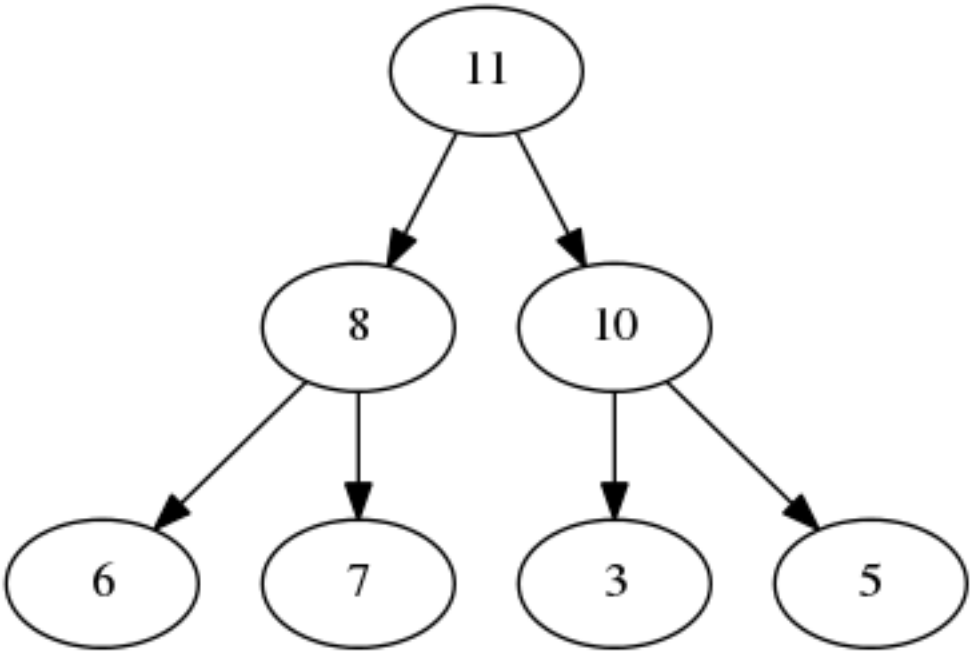

10
8
7
3
5
11

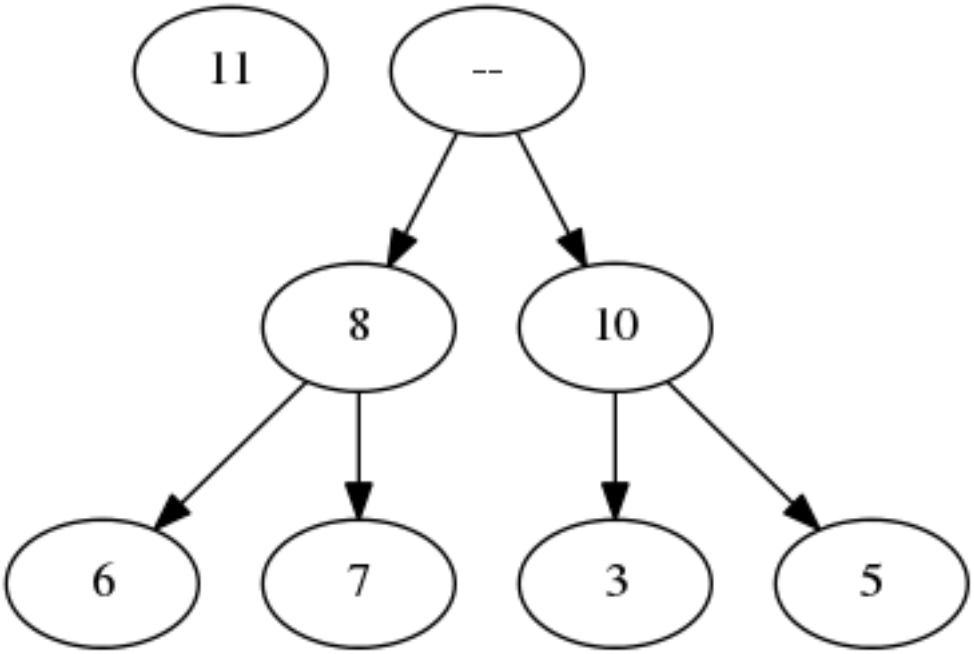

10
8
11
3
5
7

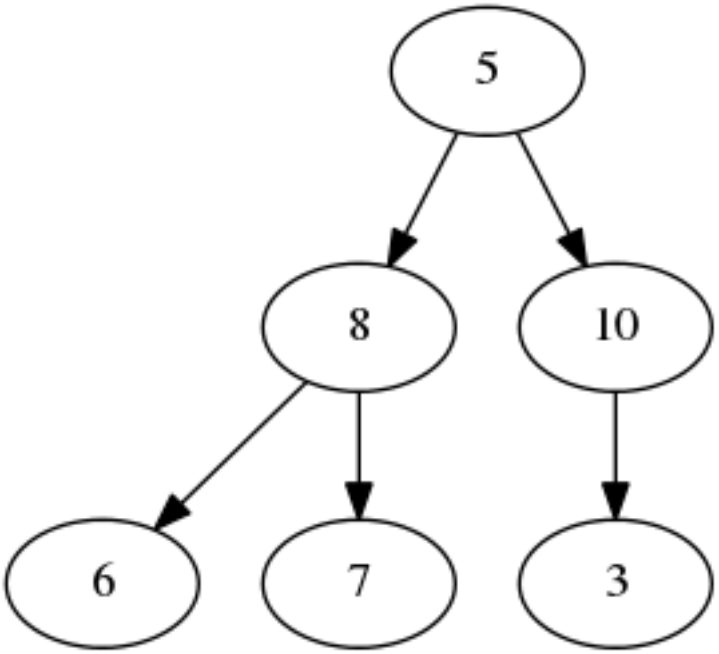


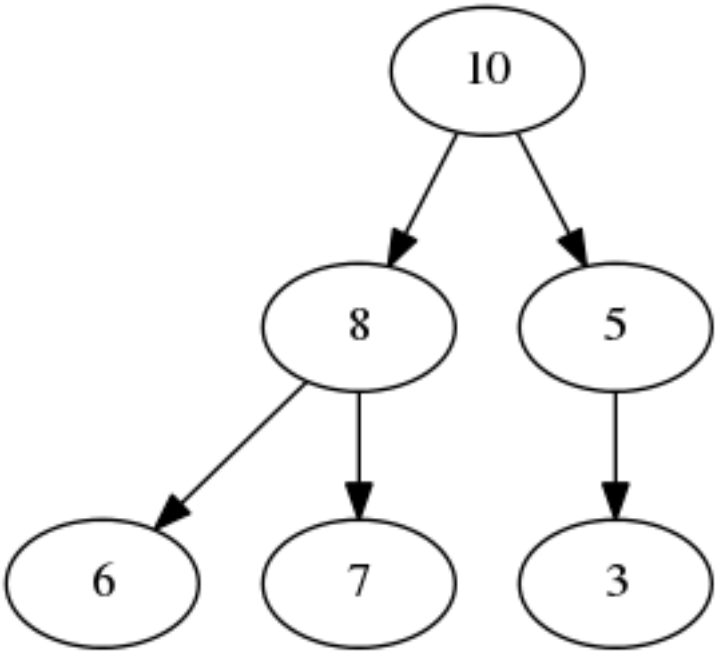
Heap Delete

- Want to delete the root, but cannot do so directly (would break tree)
- Swap with the right-most leaf (easy to find since it's last non-null element in the array)
- Heap is now incorrect because the root is not greater than it's children.
- run recursive method to move the root back down where it belongs, swapping as you go (heapRebuild())
- example:




11
--
8
10
6
7
3
5


5
8
10
6
7
3



Exercise

I've posted skeleton code for a heap to the course website. It's a heap build upon an ArrayList, and uses generic typing. Since we can calculate the index of any node we need, we don't have to store it. Thus a separate node object is unnecessary.

- HeapADT.java
  - Is the entire heap.
  - the ArrayList called mainArray is the storage. Useful array list methods
    - add(...) - adds something to the end of the list
    - get(...) - gets element at a specific index. Only works if something is there (must have been previously added)
    - set(...) - set the element at a specific index. Only work if something is there (must have been previously added)
  - The generic type T extends comparable, so use compareTo(...) to compare elements.
- MainClass.java
  - main method and testing
  - basic tests in main(...)
  - heapSort(...) is a more advanced test, uncomment the call to it in main when ready.
- Randoms.java
  - same old randoms, provides random numbers for testing

Here is what you must implement in HeapADT.java:

- heapInsert(T data) - inserts a T into the heap
- heapDelete() - deletes and returns the root node in the heap. Then rebuilds the heap as necessary

- heapRebuild(int root) - restores the heap to proper form after a delete. Recursion is recommended:
  - if any children larger than root, swap root with largest child
  - recurse on same child (which now has the value of root... we are following it down the tree) until no more swapping is necessary
- heapIsEmpty() - return true if the heap is empty, false otherwise
- swapItems() - OPTIONAL. Swaps two heap items. useful in insert and delete operations.

As usual, there is no requirement that you rigidly follow the skeleton. Modify at will.

Grading

Same as before:

- Sign the attendance sheet
- turn in your code via checkin:
  - Monday: ~cs200/bin/checkin R11L01 R11L01.tar
  - Tuesday: ~cs200/bin/checkin R11L02 R11L02.tar
  - Wednesday: ~cs200/bin/checkin R11L03 R11L03.tar
  - Thursday: ~cs200/bin/checkin R11L04 R11L04.tar
- don't worry if your not finished, just turn in what you have and add a readme file saying that you ran out of time. This will not cost points.