Fall 2014

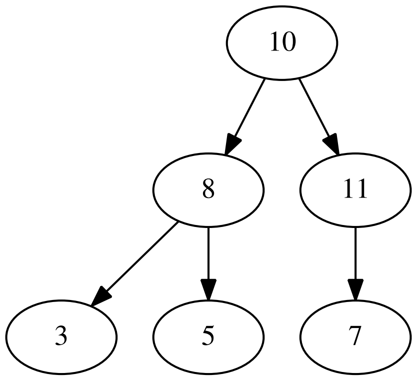
Heaps

Recitation 11

Heap ADT

- A 'tree like' ADT
- Commonly binary, but can be n-ary
- Always represents a complete tree
- We are going to build one

Example:



All heaps have the heap property: Any given node's value is greater than all of its 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 than 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 = {10, 8, 11, 3, 5, 7}
```

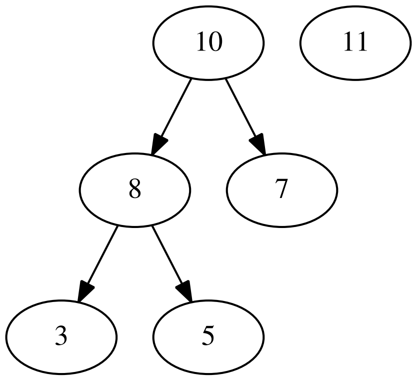
Because this representation of a complete tree (binary) has a rigid pattern, we can compute the index of the children of any particular node, or the index of a node's parent:

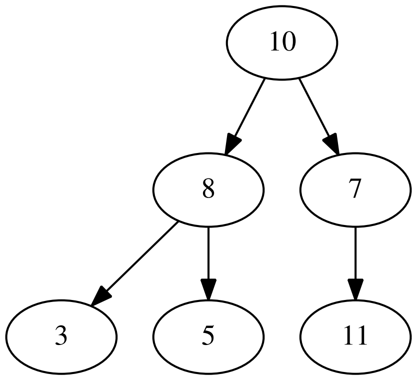
```
leftChild = (2 * parent) + 1
rightChild = leftChild + 1
parent = (self - 1) / 2) // integer math is essential for this operation to properly perform
```

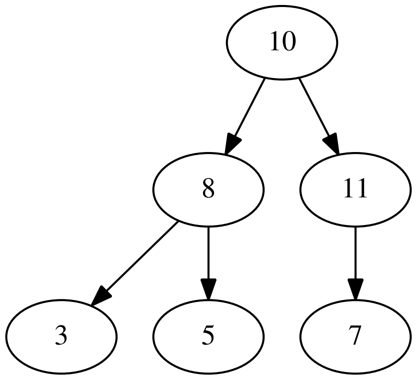
Heap Insert:

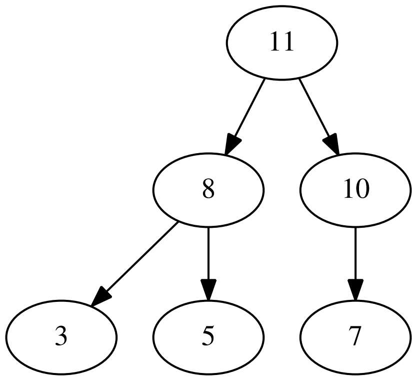
- Add node to next open space in array
- Heapify up the heap if necessary ($parent < node$)

Example:





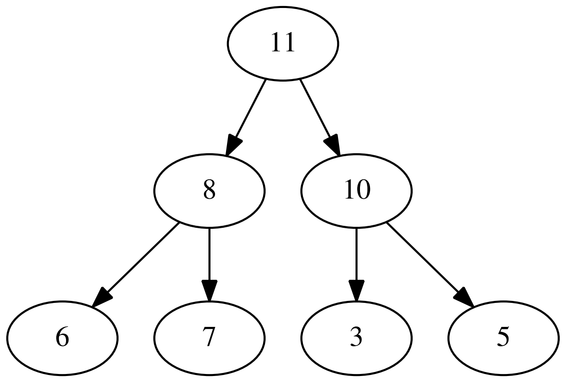


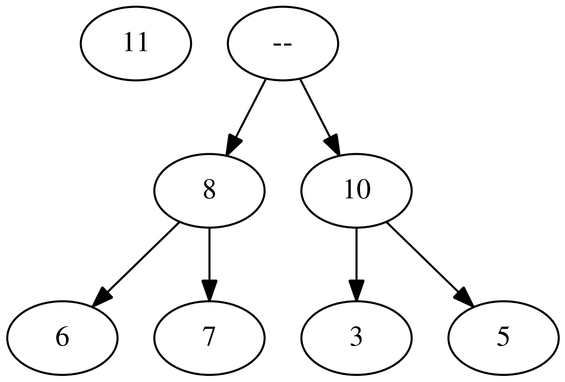


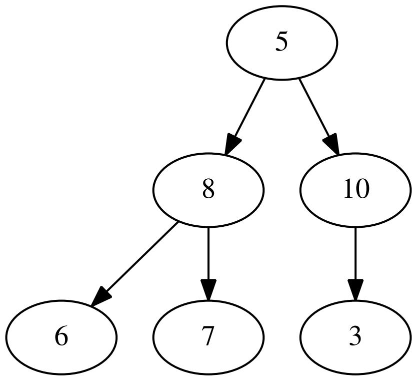
Heap Delete:

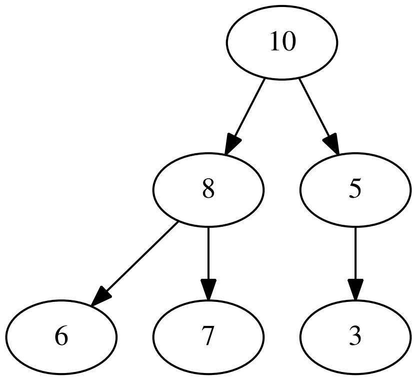
- Want to delete the root, but cannot do so directly (would break the tree).
- Swap with the right-most leaf (easy to find since it's last non-null element of the array).
- Heap is now incorrect because the root is not greater than its children.
- Run recursive method to move the root back down where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

Example:









Exercise:

Create a new project in eclipse for this recitation.
A copy of the skeleton code can be imported from ~`cs200/recit11/R11.jar`
The code provides a heap built 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.

A brief explanation of the code:

- `HeapADT.java`
  - This is the entire heap.
  - The ArrayList called mainArray is the storage. Useful ArrayList 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 works 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`
  - provides random numbers for testing

What you will be implementing:

In the class `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.
- `heapifyUp(int child)` - restores the heap to the proper form after insertion. Recursion is recommended.
  - If the parent is smaller than the child, swap the parent with the child
  - Recurse on the same with the new parent's index to work your way up the tree
- `heapifyDown(int root)` - restores the heap to the proper form after a delete. Recursion is recommended.
  - if any child is larger than root, swap root with the largest child
  - Recurse on same child (which now has the value of the root... we are following it down the tree) until no more swapping necessary.
- `heapIsEmpty()` - return true if the heap is empty, false otherwise.
- `swapItems()` - OPTIONAL. Swaps two heap items. useful in insert and delete operations.

No requirement to rigidly follow the code. Modify at will.