Test 1 is Rescheduled for Monday, March 4.

Exercise 2 should ideally be done by the time of the test, but since I'll be frantically busy during that general time period (grading tests and attending a conference), let's make the official, drop-dead (must have a serious attempt by this time) due date as 1:00 p.m., Saturday, March 9.

No Class on Thursday, March 7. I'll be attending the SIGCSE conference March 6–March9, so my office hours on March 6 and March 7 are also canceled. In place of class on March 7, you are welcome to meet in the class room and talk to other students about any problems you are still having with Exercise 2 (or even Exercise 1).

Sample Questions for Test 1

In case it might be helpful, I'm giving some sample questions. The intent of this discussion is to help you know what to study, in case that helps your learning. Note that these questions are intended to be representative of the general kind of questions that will be on the test, but the actual questions will of course be different, and on the actual test the points may be distributed differently (i.e., not equally weighted among these 6 types of questions).

You might want to work these specific problems—we will go over them in detail on February 28.

1. **Working with Arrays:** given a description of a desired method or algorithm working on a one-dimensional array, be able to write code to implement it.

   Study: the insert and remove methods in our implementation of the list ADT, and all the other examples we have done with a single loop to process an array in some way.

   Example: Given the instance variables below, which follow our typical approach, write the method `reverse` that will rearrange the items in `a` to be in reversed order. For example, if the list starts out as `3, 7, 11, 15, 21, 35`, after this method runs, the list should be `35, 21, 15, 11, 7, 3`.

```
// instance variables:
private int[] a;
private int n;

public void reverse()
{
  // insert your code here (use your own paper)
}
```

---

2. **Implementing a specified ADT:** given an ADT (the method headers with English explanation) and a required approach, typically using an array to store data, implement the methods.

   Study: our version of the list ADT, and the two implementations (in-class for `Sack1` and Exercise 2) of the sack ADT.

   Example: In everyday life, and many programming situations, we want to view a collection of items, instances of a `Person` class, as being organized in a "line." Write the method bodies, following the approach that the people are arranged consecutively in the array, with position 0 holding the first person in line and position `n-1` holding the last person in line.

```
// instance variables:
private Person[] a;
int n;

// add the given person to the line at the end
public void add( Person p )
{
  // insert your code here (use your own paper)
}

// remove the person who is first in line and return
// a reference to that person
public Person serve()
{
  // insert your code here (use your own paper)
}
```

---

3. **Efficiency analysis:** given some description of an algorithm, state the $O()$ category for the number of representative operations for an input size n.

   Study: lots of in-class discussions—you should be able to state the $O()$ category for everything we have done.

   Examples:

   What is the $O()$ efficiency category for the worst-case number of array store operations for each of the algorithms from Problem 2?

   Suppose some algorithm takes $2 + 4 + 6 + \cdots + 2n$ representative operations for input size n. What is the $O()$ category for this algorithm?

   Suppose some algorithm takes $3n^2 + 7n + \log n + 1000$ representative operations for input size n. What is its $O()$ category?

---

4. **Using a given ADT implementation:** be able to create an application that uses some given ADT implementation to accomplish some goal.

   Study: Exercise 1 (works with a list of strings), the game application for Exercise 2 (uses a sack to store game blocks).

   Example: Write a method to count the number of food blocks in a given sack of blocks, using the sack ADT methods and the `getKind()` method in the `Block` class.

```
public int countFood( Sack1 sack )
{
  // insert your code here
}
```

---

5. **Famous algorithms:** be able to demonstrate the behavior of a famous algorithm.

   Study: the merge sort and binary search algorithms are the most famous, should also study how to find the minimum in a range of a list, and the insert and remove methods in our implementation of the list ADT

---

6. **Recursion:** given a description of a desired method or algorithm and the header, be able to write code to implement it recursively.

   Study: the merge sort and binary search algorithms, along with other simpler examples.

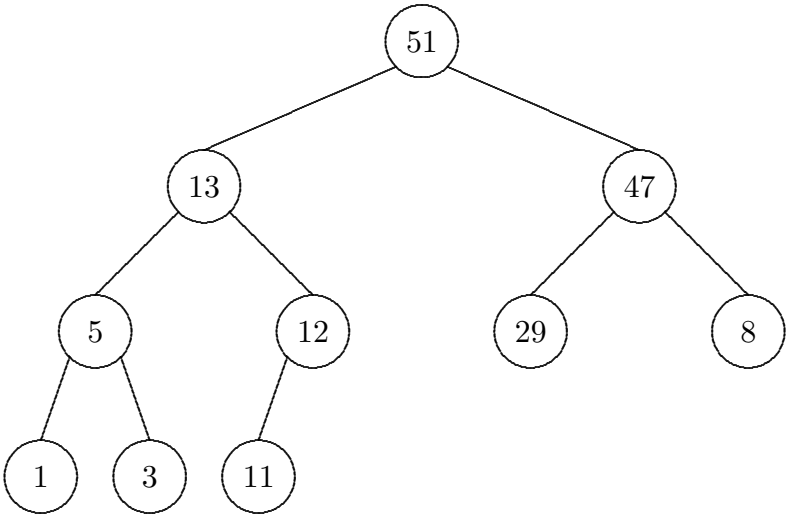
   Example: Write the body of the method below that takes an array and a range and computes the sum of the items in that range recursively—no credit if a loop is used. Here's the recursive idea: if you are given a pile of cards with numbers on them, you can ask your assistant to add up all the cards after the first one, and then when the assistant reports their answer, you can add your single card to that answer and return it.

```
public int sum( int[] a, int first, int last )
{
  // insert your code here (use your own paper)
}
```

The Heap Sort

As our last (for now) example of working with arrays, we want to develop the *heap sort algorithm*, which sorts an array in $O(n \log n)$ worst-case comparisons without using any extra storage.

A heap is a binary tree. We will work with binary trees a lot more later, but for now the picture below is enough, together with the terminology that each *node* (circle with an item in it) has a data item and 0, 1, or 2 links to other nodes. These other nodes are known as its left and right child nodes.



Later we will implement binary trees by using nodes with references to other nodes, but for now a tree is just a conceptual, pictorial thing. And, we will be able to store the items in a binary tree by using an array, following this simple idea: traverse the binary tree starting with the root (the only node that doesn't have any other node pointing to it) and go left to right, level by level, and as each item is encountered, store it in an array in the next available position. If we do this with our example, we get this array:

0	1	2	3	4	5	6	7	8	9
51	13	47	5	12	29	8	1	3	11

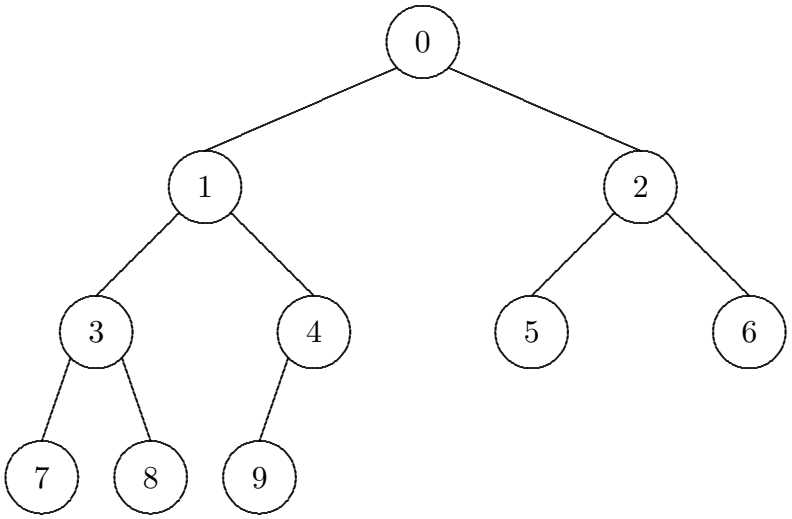
So, we will use an array to store the items in a heap, but we will constantly be picturing what the corresponding binary tree looks like.

> Actually, only a so-called "complete" binary tree can be stored in an array in this way.

To work interchangeably with the array and the corresponding binary tree we need to know these crucial facts:

- For a node in position k, its left child is in position $2k + 1$ and its right child is in position $2k + 2$.
- For a node in position j, its parent is in position $(j - 1)/2$, where the division is integer division.

⇐ Verify these claims by examining this tree, where the position of each node in the array is shown as the node's data:



The Heap Property

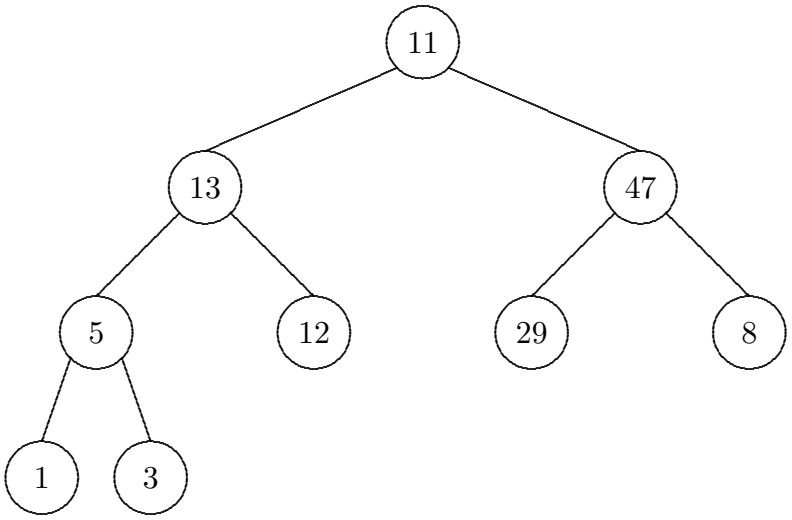
So far we have only been considering any complete binary tree. Now we want to define the "heap property" for a binary tree, which is that each node's data is bigger (whatever that means in a particular situation) than all of its 0, 1, or 2 children's data.

⇐ Verify that our example tree has the heap property. Note that if you start at any node and follow any downward path, you find a sequence of decreasing values.

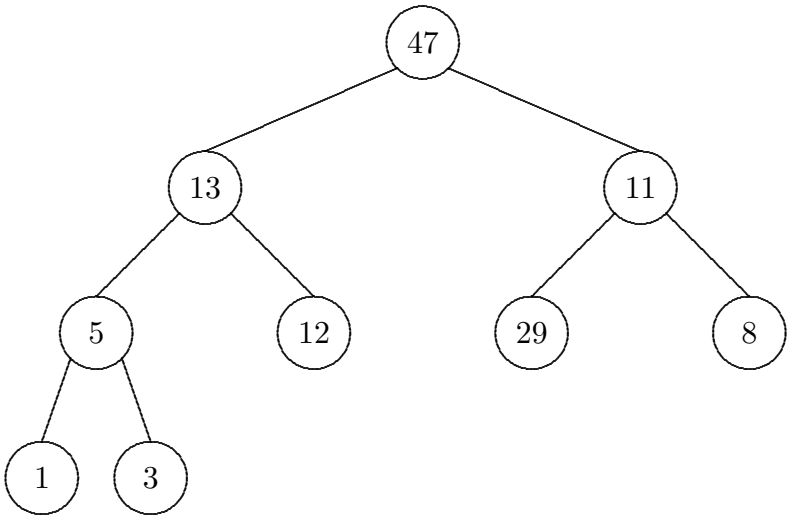
Removing from a Heap

To remove an item from the heap, we note the root data, swap the last item (bottom level, farthest right) up to the root position, decrement `number`, let the new root item (which is now horribly out of place) swap its way downward until it isn't out of place, and return the original root item. The idea of the downward swapping is to restore the heap property. We simply let the item drop by swapping with the larger of its one or two children, and repeating until it is larger than all its children.

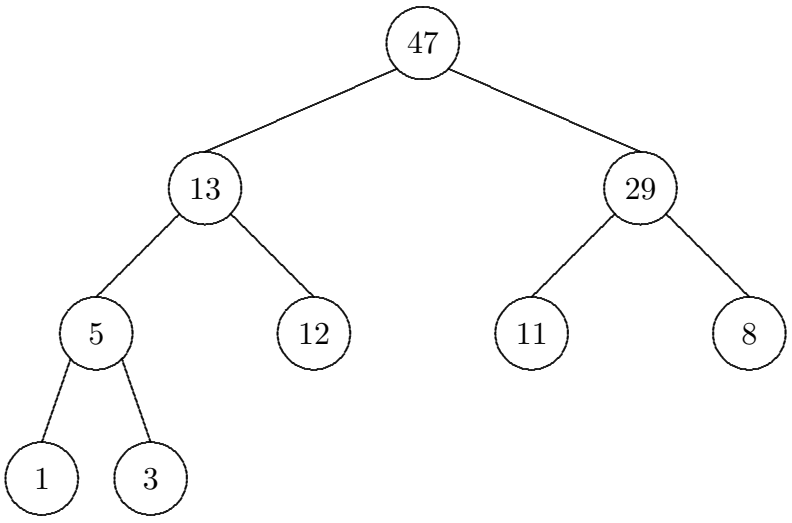
In our example, the item 51 is noted to be returned at the end. Then we swap the 11 into the root position and reduce `number`, obtaining this tree:



Next we let the 11 fall as far as it needs to. At this step, its children are 13 and 47, so we swap it with the 47 (because otherwise we'd be putting the 13 above the 47, which would violate the heap property), obtaining this tree:



Next the 11 has children 29 and 8, so we swap it with the 29, yielding:



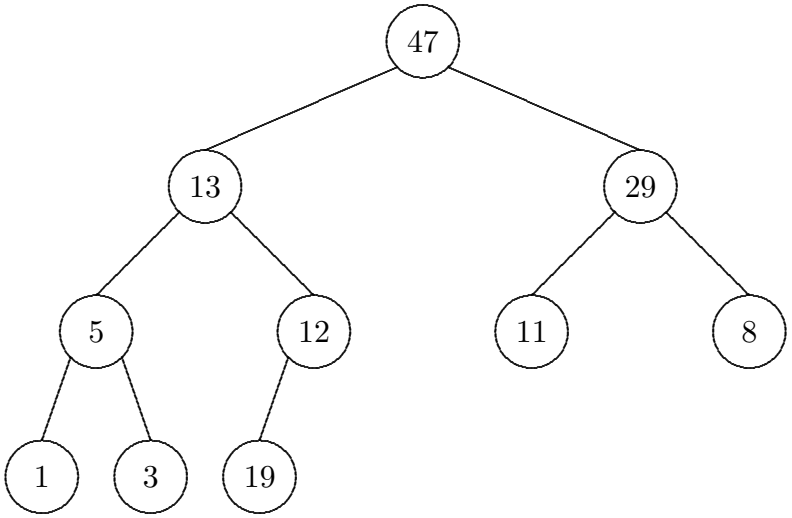
At this point the 11 has no children, so it stops falling. In general an item stops falling when it is bigger than all of its children.

⇐ Create an array representing a binary tree that has the heap property and demonstrate the removal operation directly in that array. Do this without drawing the binary tree that corresponds to the array.

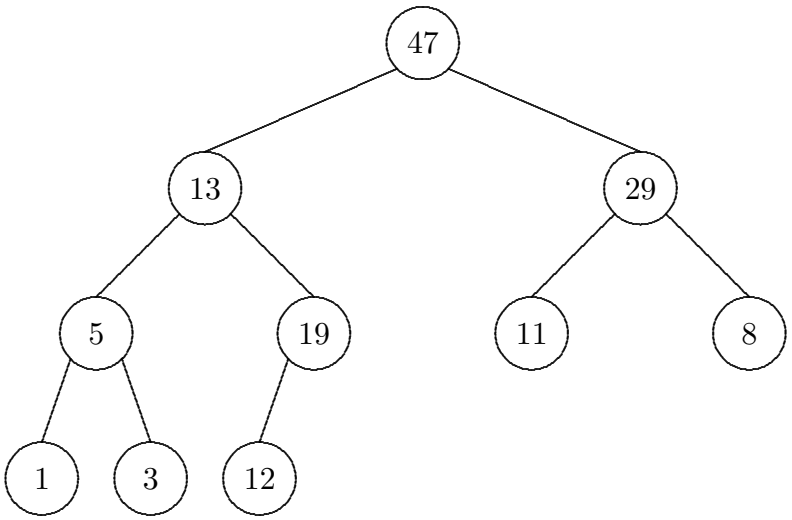
Adding to a Heap

To add an item to a heap, we place it in position `number` in the array, which corresponds to the next available spot on the bottom level (or even as the first item in the next level). We increment `number`, and then let this new item swap its way upward until it reaches the root or a situation where it is smaller than its parent.

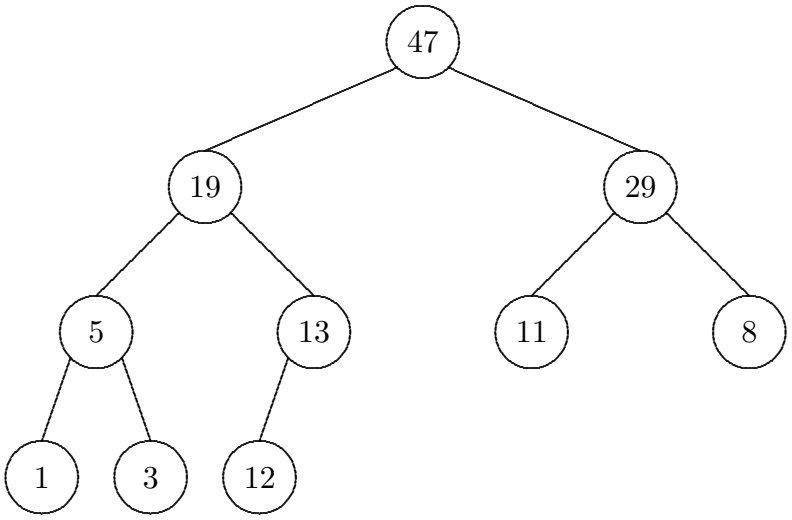
In our example, if we add 19, we first add the item to the end of the array, corresponding to this tree:



In this tree the heap property is violated because 19 is a child of 12, so we swap the 19 with its parent, obtaining this tree:



Now the heap property is violated because 19 is below 13, so we swap those two, obtaining this tree:



At this point the 19 stops rising because it is less than its parent, 47, and the tree satisfies the heap property.

⇐ Create an array representing a binary tree that has the heap property and demonstrate the add operation directly in that array. Do this without drawing the binary tree that corresponds to the array.

⇐ We should make sure that we believe that these two algorithms will always work. Maybe there's something special about our examples that lets them work when they won't always work?! To prove that the "dropping" algorithm works, we can note that wherever the falling item, its left and right subtrees satisfy the heap property, and we can prove that doing the correct swap will make the whole tree currently rooted at the dropping item satisfy the heap property. Then we should try to convince ourselves that the "rising" algorithm works.

⇐ What is the $O()$ category for the worst-case number of comparisons these remove and add operations?

⇐ Using the index cards, develop the heap sort algorithm in the real world.

⇐ Now let's write the heap sort algorithm in Java.