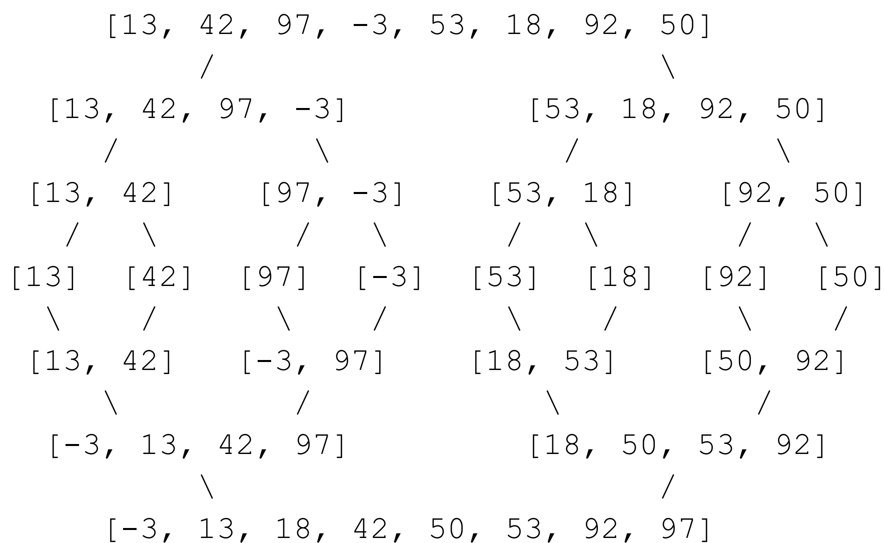
Lecture 23: Sorting

- Recently we've talked a bit about putting things in sorted order using Collections.sort
- But how exactly do they work?
  - That's what we'll talk about today
  - A lot of computer science students still don't understand sorting
  - Can often be an interview question
- Do you guys have any techniques for sorting?
  - How would you sort a deck of cards?
  - If you were a TA, how would you sort exams by last name?
- Two simple strategies
  - <u>*Insertion sort*</u>: choosing an item and *inserting* it into the proper place in a new list
  - <u>*Selection sort:*</u> look through the data and choose the smallest item. Then go back and choose the second-smallest item, etc.
  - Both of these are O(n^2) - they require approximately n passes over the n data
  - So these are slow sorts, and work fine on small amounts of data, but we can do better
- We're going to explore the idea of something called <u>*"merge sort"*</u>
  - The idea behind merge sort is simple
    - Divide the data into two halves
    - Sort each half
    - Merge the sorted halves together
  - This is going to end up being a recursive method - why?
    - Because we have to sort each half (using merge sort!)
  - A demonstration:



- We are going to write this method - (show the starter code)
  - The starter code builds up two random linked lists of strings
  - It "warms up" the Java VM by doing two trial runs
  - Then it compares our sorting routine with Collections.sort
  - We want to write the sort() method

- But first we're going to start with a helper method, one that we already know that we need
  - Merge two sorted Queues together (pre: queues are sorted)
  - Also accept a result queue in which to place the merged data
  - How did we do this in our demonstration?
    - We looked at the two values in the front and compared them, taking the smaller one
    - (we're taking advantage of the fact that the lists are already sorted, so the smallest value has to be at the front of one of the queues)
    - How do we look at the one in front without "picking" it?

```
if (list1.peek() <= list2.peek())
    move front of list1 to result
```

  - But in real life, we can't compare strings with <=
    - We have to use compareTo

```
if (list1.peek().compareTo(list2.peek()) <= 0)
    result.add(list1.remove());
else
    result.add(list2.remove());
```

  - And we want to keep doing this repeatedly, so wrap it in a loop
    - When do we want to stop?
    - Stop when one of the lists becomes empty, because then we can no longer "peek"

```
while (!list1.isEmpty() && !list2.isEmpty()) {
```

  - What happens when one of the lists becomes empty?
    - If list2 is empty, we want to add all remaining list1 elements to the result
    - If list1 is empty, we want to add all remaining list2 elements to the result
  - So either one of these two situations is needed:

```
while (!list1.isEmpty())
    result.add(list1.remove());
while (!list2.isEmpty())
    result.add(list2.remove());
```

  - It turns out that it doesn't hurt to do both of these - we don't have to wrap it in an if/else because it won't do anything if one of the lists is empty
  - And we're done!
- Now we can write the actual sorting function
  - We said this would be recursive, so the first thing to think about is a base case
    - What is a very easy list to sort?
    - An empty list
    - Or even a one-element list (there is nothing to order it with)
    - And what do we want to do in this case? Nothing!
    - So we'll have an "inverted" base case

```
public static void sort(Queue<String> list) {
    if (list.size() > 1) {
        ...
    }
}
```

  - The first step in the merge-sort process: split into two sublists

        - We'll have to create two helper queues
        - Then we want to put half of the values in the first helper, and half of the values in the second helper
        - How do we find half? size/2
        - But this doesn't work:

```
int half = list.size() / 2;
for (int i = 0; i < half; i++)
    half1.add(list.remove());
for (int i = 0; i < half; i++)
    half2.add(list.remove());
```

        - What about the odd-size queue? Then we want extra in one of the queues

```
int half = list.size() / 2;
for (int i = 0; i < half; i++)
    half1.add(list.remove());
while (!list.isEmpty())
    half2.add(list.remove());
```

    - Then we said we wanted to sort each half
        - How can we do this?
        - "If only we had a method that could sort a queue..."
        - So we can make a recursive call

```
sort(half1);
sort(half2);
```

    - The final step is merging the two sorted lists back together
        - But we already did this!

```
mergeInto(list, half1, half2);
```

- So now we can run our code and see how it compares to Collections.sort
    - What would you think is a good ratio of comparison, considering that Sun had a long time to get Java to work well?
    - A factor of 10? 50?
    - (run)
    - We actually match within a factor of about 2.5! Not bad for less than an hour of work
    - People also think that recursion and linked lists are slow - so 2.5 is still really good!
    - My output also mentions that this sort is ***<u>stable</u>***
        - What is a stable sort?
        - A stable sort preserves the relative order of things that are considered equal
        - Ex. sort a list of students by name. Then sort the list by class standing. What you would find is that all freshmen are grouped together, in alphabetical order. Same with sophomores, juniors, seniors
- Why is this code so fast?
    - How much work is done at each level? split is O(n), merge is O(n)
        - So O(n) at each level
    - And how many levels?
        - We keep splitting in half until we get to 1 --> how many times can we split in half?
        - log(n) --> so overall this sort is O(nlog(n))

- Other sorts:
    - Bogo sort: randomize the elements, then see if sorted, repeat until sorted (e.g. throw a deck of cards in the air and see if they land in sorted order)
        - (a joke sort)
    - Quick sort (this one is actually used a lot)
        - Pick a “pivot” value
        - Anything less than the pivot goes to the left side, anything greater than the pivot goes to the right side
        - Recursively sort the two sides
        - In practice, can actually be better than merge sort
        - But in worst case, is worse
    - Heap sort: essentially uses a priority queue
        - Put the elements into a priority queue, and then take them out
        - O(n log n)
    - Bubble sort
        - Repeatedly look through the list and swap any pairs of elements that are out of order
        - O(n^2)
- Can we do better than O(n logn)?
    - “Bucket sort” - if you know that you want to sort things into a variety of predetermined buckets, you can iterate over the data once and place each item in the proper bucket