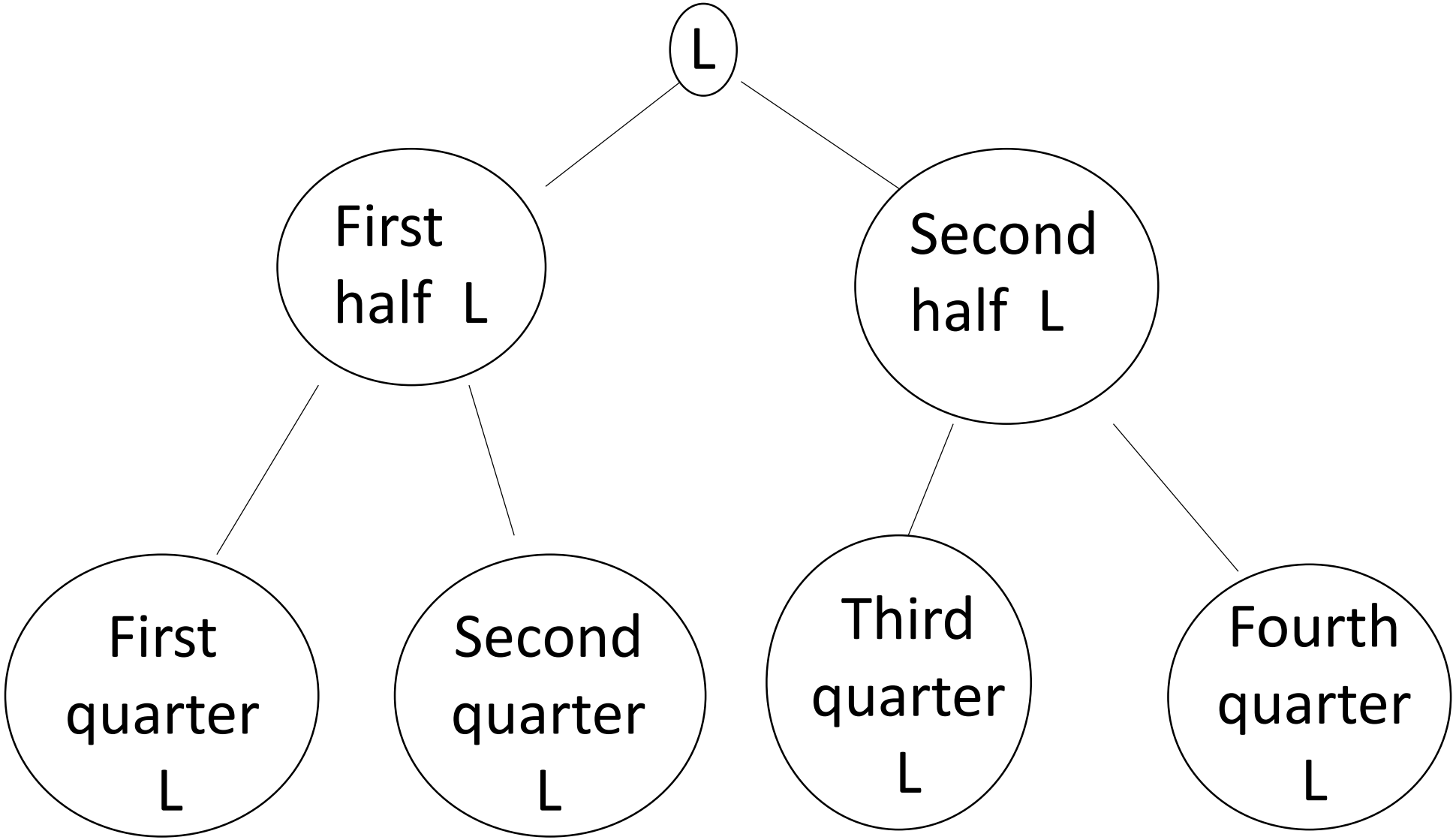
Sorting

We have seen several sorting algorithms, but they all take time $O(n^2)$ to sort a list of size n. A better algorithm is based on the simple fact that if we have two sorted lists A and B whose sizes together add up to n then we can merge them into a single list C of size n by only doing n comparisons: at each step we compare the next elements of A and B and put the smaller one into C.

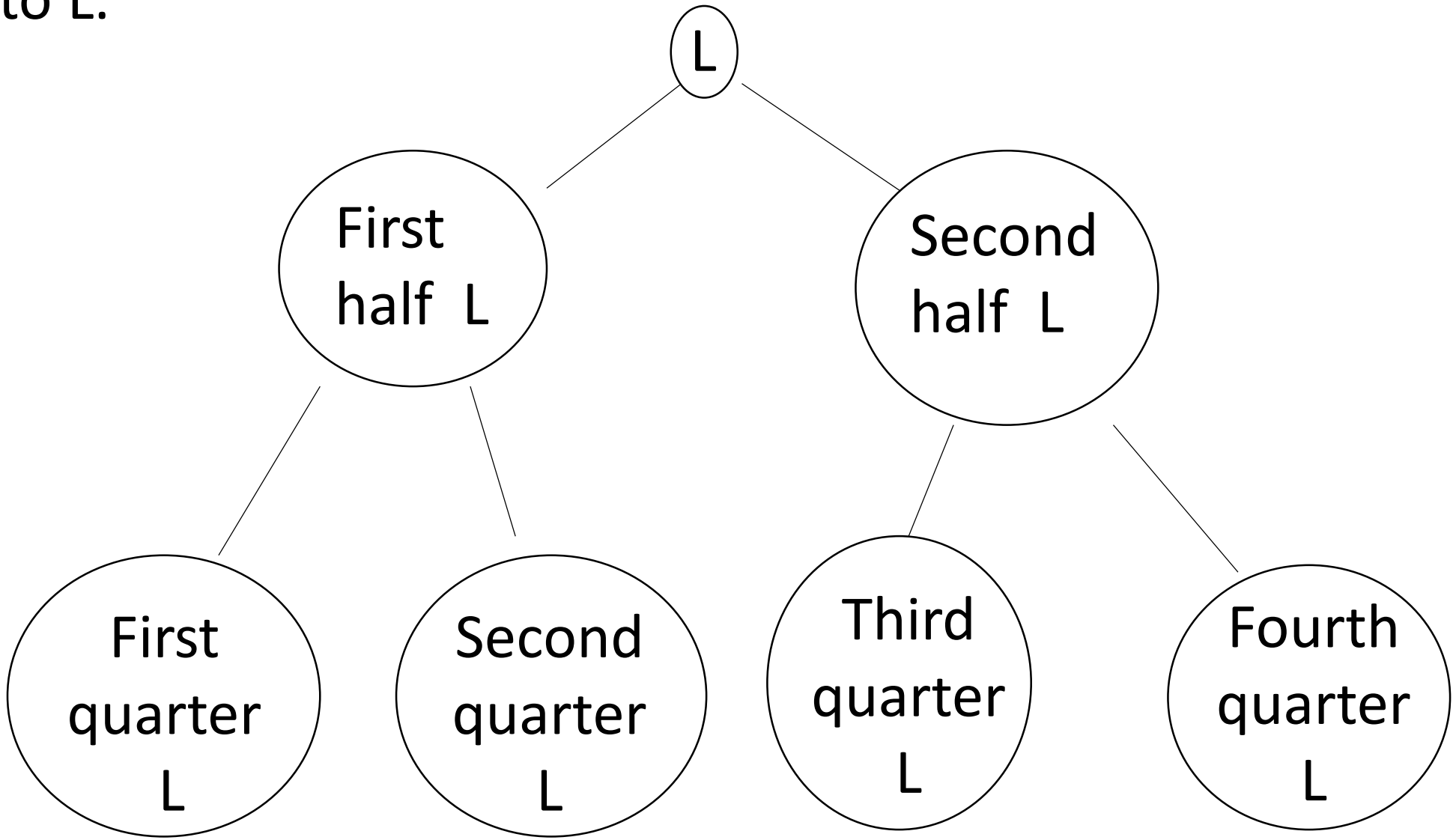
This gives us the *MergeSort* algorithm: at each step find the midpoint of the list. We recursively sort the first half of the list and recursively sort the second half of the list, then we merge these two into one sorted list.

The downside of MergeSort is that it takes extra memory: we can't merge the two halves in place without the danger of overwriting some elements that haven't yet been merged, so we merge the two halves into a new temporary list and then copy the temporary list over the original halves. This makes for a lot of extra copying, but it still makes fpr a much faster algorith,.

Consider the following picture, which shows the first two stages of breaking K into pieces:



If n is the length of L, it takes n comparisons to merge the bottom row into the middle row, and n comparisons to merge the middle row into L.



In fact, if we made the full diagram for MergeSort of L, it would have log(n) levels; each level would take n comparisons to merge into the level above. Altogether, MergeSort(L) does O(n*log(n)) comparisons, where n is the length of L. For large values of n that is a big improvement over SelectioSort(L), which does O(n^2) comparisons.

There are sorting algorithms that are similar to MergeSort only they avoid all of the extra copying that comes from merging into a temporary list. All of these run in time O(n*log(n)). In CS 151 we show that you can't do any better: any algorithm that sorts by comparing data must have a worst-case running time of at least O(n*log(n))