Chen describes a linear time in-place merging procedure, that we used to implement a merge sort algorithm.

The merge requires two adjacent sorted lists and a block size k as input. Both sublists are divided into a series of blocks, each of size k (with the possible exception of an undersized leftmost block of length f). The two rightmost blocks of the first sublist will initially be used as internal buffers. During the execution of the merge, the order of the left list’s blocks may be altered, but the elements within each block, again excluding the buffer blocks, remain sorted. The elements are inserted into a hole, which is created by moving one element to some temporary memory. As the algorithm progresses, the hole is always at the position where the next element will be inserted. While the position of the hole and the second list’s smallest element are only incremented by 1, each time they are updated, the buffers and the first list’s current block, containing its smallest elements require more attention, as we will explain in the following description of the corresponding procedures.

We do not provide a detailed description of all the procedures, as our implementation does not differ from the pseudo-code given by Chen in most parts, but we will provide a short overview.  
Some notations: n denotes the size of the list. As in the original paper we will use X to denote the first and Y to denote the second sublist.

mergesort\_chen: Divides the given list into two sublists of equal size. The first sublist is always sorted with another merge sort algorithm, using the second one as extra space, while the second sublist is handled by a recursive call to “mergesort\_chen”. The recursion stops when the list contains 50 or less elements and the remaining list is sorted by insertion sort. Both sublists are then merged using the “merge”-procedure with block size k=sqrt(n).

merge: This is the main procedure of Chen’s merging algorithm. First, there is some preparatory work, mostly initialising variables, like the size f of a possible undersized leftmost block, the position of the b1 and b2 buffer (more specific, the iterators to the respective first element), initially consisting of the 2 rightmost X-blocks. Also, the hole must be created and the corresponding element must be stored in temporary memory. Afterwards we begin to merge both sublists in a while-loop until there is no more room for the buffers in the remainder of X, i.e. all but 2k X-elements are in their final position. First, we check whether the current X-element or Y-Element is smaller or the second sublist is exhausted, in which case we always pick the X-element. In either case the following moves will be executed: hole 🡨 smaller 🡨 buffer element 🡨 element currently in next hole. While the hole position z and the current Y-element position y are only incremented by one in each iteration, keeping b1, b2 and the current X-element position x updated is a bit trickier. The latter one is updated each time it hits a block boundary by the “findNextXBlock” procedure. If the hole is not currently positioned in the block just left by x, then b2 is set to x-k before x is updated, because the block does now consist solely of buffer elements. The same buffer-update will be applied if y reaches a block boundary, but we do not need to check for the position of z in that case. Furthermore, b2 must be invalidated if the hole enters the current b2 block, which is done by setting b2 to the end-iterator of the current list. In each iteration we need to increment b1 and when it hits a block boundary, we set it to b2 if a b2 buffer is available, otherwise the b1 buffer can be reused and we set it to b1-k. After the while-loop terminates, we can put the element we used to create the initial hole back in the current hole, which is the element’s correct position.  
This process leaves us with a sorted output block, where every element is already in its final position, two unsorted buffer blocks and possibly a sorted remainder of Y.  
As suggested by Chen, we handled the remaining list by first sorting the buffer, merging the sorted buffer and the remaining Y-elements with block size sqrt(k), then using “mergeBandY” in the recursive invocation of “merge”. The proposed heapsort for sorting the buffer is replaced by our “mergesort\_chen” procedure, as our goal here is to compare different in-place merge sort algorithms without using other sorting algorithms for greater parts of the lists. Also, due to the O(k²+|remainder of Y|) time complexity of “mergeBandY” we do the recursive step two times instead of one before using “mergeBandY”, because the list can still be quite large in the first recursion step.

mergeBandY: Merges an unsorted block B with a sorted Block Y until B or Y is exhausted.  
If B is exhausted first, the merge is finished, otherwise the remaining B-Elements need to be sorted differently. Again, we chose “mergesort\_chen” over heapsort for this task.  
This procedure is not very efficient because in each step we determine the smallest remaining B-element and compare it with the current Y-element.

findNextXBlock: As the X-blocks are not always in their correct order during the merging process, we use this function to determine the next block, after we hit a block boundary.  
This is accomplished by comparing all X-blocks, excluding buffer blocks, by their respective first and last element and choosing the X-Block with the smallest elements.