1.
   ```
   max_heap_delete(A, i) {
             A[i]=A[A.heap_size];
             A[A.heap_size] = null;
             A.heap_size--;
             if (i==1 || A[i/2] < A[i]) {
                       Max_Heapify(A, i);
             } else {
                       while(A[i/2]>=A[i] && i!=1){
                                 swap(A[i/2], A[i]);
                                 i=i/2;
                       }
             }
   }
   ```

   The time complexity should be the exact same as Max_Heapify() which is O(log n) because the deletion itself, along with a couple other lines of code, are just constant-time operations and so the fixing of the heap is what takes up the time cost. During the repairing process, the code will either perform the heapify method or traverse up the heap, making swaps as needed. Since this is the case, the time cost is bounded by the height of the tree which is log n.

2. An idea would be to create an output array that's size is k*# of transactions in each file so that the array is ready to be loaded and printed to an output file. Then, create a min-heap of a custom object type that holds data and the list # that the data came from (HeapNode). Then take the top element in each of the k lists and make HeapNodes for each one and insert them into the min-heap. Then take the root HeapNode of the min heap and insert the data into the output array and insert the next element from the list that that data belongs to into the min-heap. If any of the given lists are traced through completely, insert MAX_VALUE with that list into the heap. Repeat this process until all of the k lists have been traced through completely. Then finally, write the output array to an output file and the task is completed. This will take O(n log k) because there are n additions and removals to and from the heap and the heap contains at most k elements at every time step.