1. **2-3 trees basics.** Suppose we insert the following numbers into a 2-3 tree (which is initially empty), in the given order:

   9, 15, 4, 3, 6, 13, 2, 11, 7, 1, 16, 12, 14, 8, 10, 5.

   (a) Draw a picture of the 2-3 tree after each insertion: you do not have to write down the “guides” at each node — just the tree structure and the values stored at the leaves.

   (b) Suppose that now a split operation is performed, which splits the tree into two trees: one containing those numbers ≤ 8, and the other containing those numbers > 8. Show the resulting trees (again, you can omit the “guides”).

2. **Heap basics.** Suppose we insert the following numbers into a min heap (which is initially empty), in the given order:

   9, 15, 4, 3, 6, 13, 2, 11, 7, 1, 16, 12, 14, 8, 10, 5.

   Recall that a min heap is a heap where the root contains the smallest item (as opposed to a max heap, where the root contains the largest item).

   (a) Draw a picture of the heap after each insertion. Draw the heap as a tree (ignoring the fact that this tree is usually implemented as an array).

   (b) Now suppose that three DeleteMin operations are performed. Draw a picture of the heap after each of these three operations.

3. **A heaping challenge.** You are given a min heap containing \( n \) data items, along with a data item \( x \) and a positive integer \( k \). Your task is to design an algorithm that runs in time \( O(k) \) and answers the following question: are there at least \( k \) items in the heap that are less than \( x \)? Of course, you could go through the entire heap and just count the number of items that are less than \( x \), but this would take time proportional to \( n \). The challenge is to design an algorithm whose running time is \( O(k) \) by somehow using the heap property.

4. **k-way merge.** Use a heap to design an \( O(n \log k) \)-time to merge \( k \) sorted lists into one sorted list, where \( n \) is the total number of elements in all the input lists.

   *Note:* Your algorithm should use space \( O(k) \) of internal memory, reading the input lists as streams, and writing the output list as a stream.

5. **List maintenance (I).** Consider the problem of maintaining a collection of lists of items on which the following operations can be performed:

   (i) Create a new list with one item.

   (ii) Given two lists \( L_1 \) and \( L_2 \), form their concatenation \( L \), i.e., the list consisting of all items in \( L_1 \) followed by all items in \( L_2 \) (destroying \( L_1 \) and \( L_2 \) in the process).

   (iii) Given a list \( L \) and a positive integer \( k \), split \( L \) into two lists \( L_1 \) and \( L_2 \), where \( L_1 \) consists of the first \( k \) items of \( L \), and \( L_2 \) the rest (\( L \) is destroyed in the process).

   (iv) Given a list \( L \) and a positive integer \( k \), report the \( k \)th item in \( L \).

   Describe data structures and algorithms supporting these operations so that operation (i) takes constant time, and operations (ii)–(iv) can be performed in time \( O(\log n) \) (where \( n \) is the length of \( L \)).

   *Hint:* Use a variation on 2-3 trees. Be sure to specify what information is stored at each node. Just sketch the algorithms, emphasizing the similarity and differences with algorithms for ordinary 2-3 trees.

6. **List maintenance (II).** Extending the previous exercise, suppose we also want to an operation that reverses a given list \( L \). Show how this operation can be implemented in constant time, while the other operations can still be performed within the time bounds of the previous exercise.

   *Hint:* this should be a small modification to the solution to the previous exercise, making use of the “lazy evaluation” trick that was used in FlipRange example from class.