Basic Algorithms — Fall 2020 — Problem Set 2
Due: Wed, Oct 7, 11am

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.

   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.

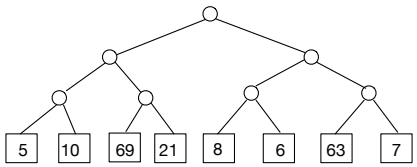
   (ii) Given two lists L_1 and L_2, form their concatenation L, i.e., the list consisting of all items in L_1 followed by all items in L_2 (destroying L_1 and L_2 in the process).

   (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 kth item in L.

   Describe a data structure 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, where the list elements are stored at the leaves in the same order as they appear in the list. For example, the following tree represents the list $(5, 10, 69, 21, 8, 6, 63, 7)$:

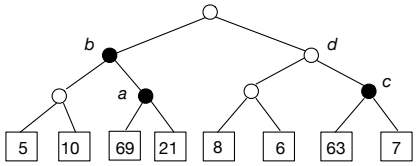


Be sure to specify what information is stored at each node. Just give high-level sketches the algorithms, emphasizing the similarity and differences with algorithms for ordinary 2-3 trees. You don't have to give detailed pseudocode.

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 modification to your solution to the previous exercise, making use of the "lazy evaluation" trick that was used in *AddRange* example from class. However, for this problem, you should have each internal node store a "reverse bit", which means that the list represented by the subtree rooted at each node is to be treated as the logical reverse of the list it would ordinarily represent. To reverse a list, all you have to do is flip the "reverse bit" of the root. However, all of the other operations become a bit more complicated. For example, after a sequence of concatenation, split, and reverse operations, you can easily end up with a tree with many internal nodes "reverse bits" set to 1.

   For example, consider the following tree, where the black nodes have their "reverse bits" set to 1:



   The subtree rooted at c represents the list $(7, 63)$, which is the reverse of $(63, 7)$. The subtree rooted at d represents the list $(8, 6, 7, 63)$. The subtree rooted at a represents the list $(21, 69)$, which is the reverse of $(69, 21)$. The subtree rooted at b represents $(69, 21, 10, 5)$, which is the reverse of $(5, 10, 21, 69)$. The entire tree represents the list $(69, 21, 10, 5, 8, 6, 7, 63)$.

   A useful subroutine you should design is the following: given a tree T representing a list L of length n and an index $k = 1, \ldots, n$, trace a path from the root to the leaf v in T representing kth element in L, so that the path from the root to the leaf gets cleared of any "'reverse bits", and moreover, T gets restructured so that all the leaves that are to the left of v in T represent elements whose position in L is in $\{1, \ldots, k-1\}$, and all the leaves that are to the right of v in T represent elements whose position in L is in $\{k+1, \ldots, n\}$.