Assignment 2: Balanced Trees

This problem set explores red/black trees and augmented balanced binary search trees. We hope that this solidifies your understanding of these types of trees!

Working in Pairs

You are welcome to work on the problem sets either individually or in pairs. If you work in pairs, you should jointly submit a single assignment, which will be graded out of 22 points. If you work individually, the problem set will be graded out of 18 points, but we will not award extra credit if you earn more than 18 points.

Due Wednesday, April 16 at 2:15PM at the start of lecture.

Problem One: Red/Black Trees (8 Points)

This problem is a combination of theory questions and programming questions. The source code for the programming questions is available at `/usr/class/cs166/assignments/ps2/`, but we have not included a Makefile or a testing harness – we're leaving that up to you this time!

The particular representation of red/black trees we've included here saves space by packing the bit indicating whether a node is red or black into the low-order bit of the left child pointer. This means that you'll have to do some masking before following the left pointer. Check the header file for more details.

We will test your code by compiling it in C99 mode (using the `-std=c99` option to `gcc`) on the `corn` machines. To receive full credit, your code must compile with no warnings (we'll use the `-Wall` flag when grading, and we recommend you use it during development) and your code should be written clearly (e.g. well-commented and with clear variable names).

i. **(4 Points)** Implement the `is_red_black_tree` function. This function accepts as input a pointer to the root of a binary tree annotated with color information and should return whether that tree is a red/black tree. You can assume that the input actually is a tree – that is, you can assume there aren't any directed or undirected cycles, and you can assume that (except for the low-order bit of the left child pointer) all internal pointers are valid – but cannot make any assumptions about the keys in the nodes or the node colors. To receive full credit, your implementation must run in time O(n). Please edit the comments before the `is_red_black_tree` function to include a brief writeup explaining why your code runs in time O(n).

ii. **(4 Points)** Implement the `to_red_black_tree` function. This function accepts as input a sorted array of integers and should return the root of a red/black tree containing precisely those integers. To receive full credit, your implementation must run in time O(n). Please edit the comments before the `to_red_black_tree` to include a brief writeup justifying why your code runs in time O(n). *(Hint: The naïve algorithm of inserting all the nodes into an empty red-black tree doesn't run in time O(n) on all inputs.)*

Problem Two: Flexible Sequences (6 Points)

There are many data structures that can be used to represent sequences of elements. Standard dynamic arrays allow for O(1) lookups, O(1) appends, but take time O(n) to insert elements at the front. Doubly-linked lists allow for O(1) insertions at the front and end and can be concatenated in time O(1), but require O(n) for random access. In this problem, you'll design a data structure for sequences that supports a huge number of operations in time O(log n) each.

Design a data structure that stores a sequence of elements and supports all of the following operations in time O(log n):

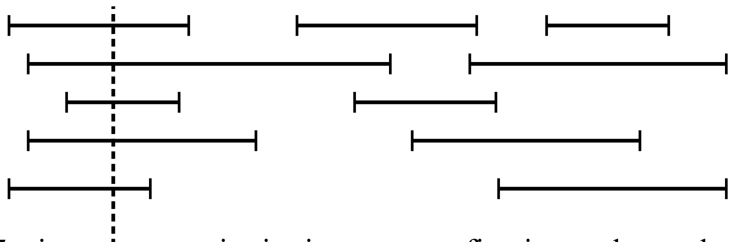
- *seq*.**insert**(i, x), which inserts value x at position i.
- *seq*.**delete**(i), which removes the element at position i.
- *seq*.**lookup**(i), which returns the value of the element at position i.
- *seq*.**set**(i, x), which replaces the element at position i with the new value x.
- *seq*.**size**(), which returns the number of elements in *seq*.
- **split**(*seq*, i), which destructively modifies *seq* by splitting it into two sequences S_1 and S_2 consisting of the elements before position i and the elements at or after position i, respectively.
- **concat**(seq_1, seq_2), which destructively modifies seq_1 and seq_2 by appending all the elements in seq_2 to the end of seq_1.

We recommend building your structure out of a balanced binary search tree. That way, you can augment your sequences to store extra information, just as you can augment a normal balanced BST. In fact, on Monday's lecture, we'll use these structures to build up Euler tour trees by adding in a few extra augmentations.

Problem Three: Dynamic Maximum Overlap (8 Points)

(Based on a chapter exercise from CLRS.)

Given a set of half-open intervals $I = \{ [s_1, t_1), [s_2, t_2), \ldots, [s_n, t_n) \}$, the *maximum overlap* problem is the following: what is the maximum number of intervals that mutually overlap one another? For example, given this set of intervals:



The answer would be 5, since at any point in time at most five intervals overlap one another (one such point in time is indicated above). The maximum overlap problem has applications to scheduling rooms for a conference or other large event – if the intervals represent the start and end times of all of the events, then the maximum overlap gives the minimum number of rooms necessary to schedule all of the events.

In the above picture, we've drawn a dotted line to indicate one of the "points of maximum overlap," a time at which the maximum number of intervals overlap. You might find the following fact useful: for any nonempty set I of intervals, there is always a point of maximum overlap that occurs at the very start of one of the intervals.

i. **(2 Points)** Design an $O(n \log n)$-time algorithm for the maximum overlap problem. *(Hint: Treat the interval boundaries as "events." Each interval start is a "+1" event, and each interval end is a "-1" event.)*

In the *dynamic maximum overlap* problem, the set I is not given in advance. Instead, the set I begins empty, and arbitrary intervals may be inserted and deleted from I at any time. At any time between insertions and deletions, we can query to determine the maximum overlap of the set I.

ii. **(6 Points)** Design a data structure that supports these operations in the indicated time bounds:

- *dynmax.***insert**(s, t), which inserts the half-open interval $[s, t)$ into the data structure. You can assume that the specific interval $[s, t)$ is not already in the data structure. This operation should run in time $O(\log n)$.
- *dynmax.***delete**(s, t), which deletes the half-open interval $[s, t)$ from the data structure. You can assume that $[s, t)$ is present in the data structure when this operation is performed. This operation should run in time $O(\log n)$.
- *dynmax.***max-overlap**(), which returns the maximum overlap of the intervals in the data structure. This operation should run in time $O(1)$.