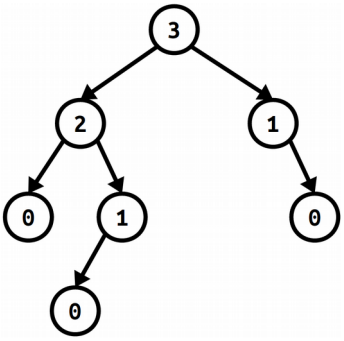
Problem Set 3: Balanced Trees

This problem set explores red/black trees, augmented search trees, data structure isometries, and some of the more advanced operations on binary search trees. By the time you've finished working through this problem set, you'll have a much richer feeling for just how powerful this particular family of techniques can be and how they can be used to design beautiful and fast data structures.

Due Thursday, May 3 at 2:30PM.

Problem One: Red/Black and AVL Trees (6 Points)



AVL trees, developed in 1962, were the first class of balanced binary search trees to be invented. They're named after their inventors Адельсо́н-Ве́льский and Ла́ндис (**A**delson-**V**elsky and **L**andis).

A binary tree is an AVL tree if either (1) the tree is empty or (2) the left and right children of the root are AVL trees and the heights of the left and right subtrees differ by at most one. As a reminder, the empty tree has height -1, and the height of a nonempty tree whose root's left child has height h_1 and root's right child has height h_2 is $1 + \max(h_1, h_2)$. A sample AVL tree is shown to the right; its nodes are tagged with their heights.

This problem explores red/black trees, AVL trees, and the connections between them. Download the starter files from

```
/usr/class/cs166/assignments/ps3
```

and implement the functions described in `trees.c`. To receive full credit, your code should compile with no warnings and should not have any memory errors. We'll test your code on the `myth` cluster. There's information about how to run the test driver in the `README` file.

i. Implement a function

```
bool is_avl_tree(const struct Node* root);
```

that takes as input a pointer to the root of a tree, then returns whether it's an AVL tree. Your function should run in time $O(n)$, where n is the number of nodes in the tree.

ii. Implement a function

```
bool is_red_black_tree(const struct Node* root);
```

that takes as input a pointer to the root of a tree, then returns whether it's a red/black tree. Your function should run in time $O(n)$, where n is the number of nodes in the tree.

It turns out that all AVL trees can be colored to meet the red/black tree requirements, though the reverse isn't true. This means that AVL trees have tighter structural properties than red/black trees. Lookups in AVL trees are faster than in red/black trees, though insertions and deletions are slower.

iii. Implement a function

```
void color_avl_tree(struct Node* root);
```

that takes as input a pointer to the root of an AVL tree, then colors the nodes in that tree so that it obeys all the red/black tree invariants. Your function should run in time $O(n)$, where n is the number of nodes in the tree.

The last part of this problem requires some creative insights on your part, but doesn't involve writing all that much code. Before tackling it, we recommend drawing out some sample AVL trees and trying to see if you can spot a pattern. You may even want to try proving, inductively, that all AVL trees are red/black trees; the same insights that would lead to a proof here also will give you a beautiful recursive algorithm.

Problem Two: Range Excision (2 Points)

Design and describe an algorithm that, given a red/black tree T and two values k_1 and k_2, deletes all keys between k_1 and k_2, inclusive, that are in T. Your algorithm should run in time $O(\log n + z)$, where n is the number of nodes in T and z is the number of elements deleted. You should assume that it's your responsibility to free the memory for the deleted elements and that deallocating a node takes time $O(1)$.

Problem Three: Dynamic Prefix Parities (8 Points)

Consider the following problem, called the ***dynamic prefix parity problem***. Your task is to design a data structure that logically represents an array of n bits, each initially zero, and supports the following operations as efficiently as possible:

- `initialize`(n), which creates a new data structure for an array of n bits, all initially 0;
- *ds.*`flip`(i), which flips the ith bit; and
- *ds.*`prefix-parity`(i), which returns the *parity* of the subarray from index 0 to index i, inclusive. (The parity of a subarray is zero if the subarray contains an even number of 1 bits and is one if it contains an odd number of 1 bits. Equivalently, the parity of a subarray is the logical XOR of all the bits in that array).

It's possible to solve this problem with `initialize` taking O(n) time such that `flip` runs in time O(1) and `prefix-parity` runs in time O(n) or vice-versa (do you see how?), but it's possible to get excellent runtimes for both `flip` and `prefix-parity` by being more creative with the approach.

i. Let $k \geq 2$ be an arbitrary natural number. Design a data structure that solves the prefix parity problem such that

   - `initialize`(n) takes time O(n),
   - *ds.*`flip`(i) takes time O($\log_k n$), and
   - *ds.*`prefix-parity`(i) takes time O($k \log_k n$).

   As a hint, start by using augmented binary trees to solve this problem, then see if you can generalize your answer to use augmented *multiway* trees instead.

In the course of solving part (i) of this problem, you essentially reduced the large problem of "solve dynamic prefix parity for an array of size n" to "solve dynamic prefix parity for a bunch of smaller arrays of some smaller size."* That sounds a lot like what we did in the Fischer-Heun RMQ data structure, which turned the larger problem of "solve RMQ on an array of size n" into "solve RMQ on a bunch of arrays of a smaller size." And just as we used a Four Russians speedup to build a fast solution to RMQ, you can use a Four Russians speedup to boost the performance of your structure from part (i).

ii. Modify your data structure from part (i) of this problem so that

   - `initialize`(n) takes time O(n),
   - *ds.*`flip`(i) takes time O($\log n$ / $\log \log n$), and
   - *ds.*`prefix-parity`(i) takes time O($\log n$ / $\log \log n$).

Some hints on part (ii) of this problem:

- Remember that $\log_k b = \log b$ / $\log k$ thanks to the change-of-basis formula.
- Think about precomputing a lookup table of all possible array/query pairs for sufficiently small arrays. How might that change how you do `prefix-parity` queries?
- An array of bits can gives an integer that can be used as an index in an array-based lookup table.
- Be precise with your choice of block size. Constant factors matter!

* If you didn't notice that – or if you're having trouble with part (i) of this problem – consider this a free hint from your friendly course staff! 😀

Problem Four: Deterministic Skiplists (8 Points)

Although we've spent a lot of time talking about balanced trees, they are not the only data structure we can use to implement a sorted dictionary. Another popular option is the ***skiplist***, a data structure consisting of a collection of nodes with several different linked lists threaded through them.

Before attempting this problem, you'll need to familiarize yourself with how a skiplist operates. We recommend a combination of reading over the Wikipedia entry on skiplists and the original paper "Skip Lists: A Probabilistic Alternative to Balanced Trees" by William Pugh (available on the course website). You don't need to dive too deep into the runtime analysis of skiplists, but you do need to understand how to search a skiplist and the normal (randomized) algorithm for performing insertions.

The original version of the skiplist introduced in Pugh's paper, as suggested by the title, is probabilistic and gives *expected* O(log *n*) performance on each of the underlying operations. In this problem, you'll use an isometry between multiway trees and skiplists to develop a fully-deterministic skiplist data structure that supports all major operations in *worst-case* time O(log *n*).

i. There is a beautiful isometry between multiway trees and skiplists. Describe how to encode a skiplist as a multiway tree and a multiway tree as a skiplist. Include illustrations as appropriate.

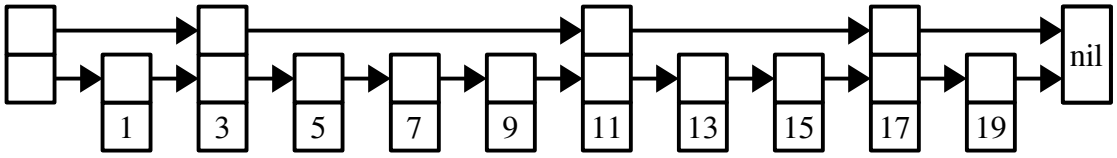
To design a deterministic skiplist supporting insertions, deletions, and lookups in time O(log *n*) each, we will enforce that the skiplist always is an isometry of a 2-3-4 tree.

ii. Using the structural rules for 2-3-4 trees and the isometry between multiway trees and skiplists you noted in part (i) of this problem, come up with a set of structural requirements that must hold for any skip list that happens to be the isometry of a 2-3-4 tree. To do so, go through each of the structural requirements required of a 2-3-4 tree and determine what effect they will have on the shape of a skiplist that's an isometry of a 2-3-4 tree.

Going forward, we'll call a skiplist that obeys the rules you came up with in part (ii) a ***1-2-3 skiplist***.

iii. Briefly explain why a lookup on a 1-2-3 skiplist takes worst-case O(log *n*) time.

iv. Based on the isometry you found in part (i) and the rules you developed in part (ii) of this problem, design a deterministic, (optionally amortized) O(log *n*)-time algorithm for inserting a new element into a 1-2-3 skiplist. Demonstrate your algorithm by showing the effect of inserting the value 8 into the skiplist given below:



Congrats! You've just used an isometry to design your own data structure! If you had fun with this, you're welcome to continue to use this isometry to figure out how to delete from a 1-2-3 skiplist or how to implement split or join on 1-2-3 skiplists as well.