Algorithms

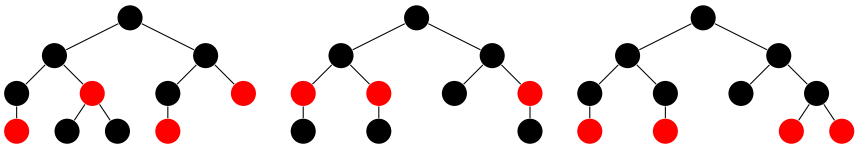
Example Sheet 3

Petar Veličković

Lent Term 2017

Warm-up

1. Propose an algorithm to convert an arithmetic expression (including integer constants, parentheses, addition and multiplication) from infix to postfix notation. For example: $(3 + 12) \times 4 + 2 \longrightarrow 3\ 12 + 4 \times 2+$
2. Prove that, if a binary search tree (BST) node n has two children, its successor has no left children.
3. Which of the following BSTs are also Red-Black trees (RBTs), and why?



4. For each of the possible 2–3–4-tree nodes (and their environments), sketch an equivalent cluster of RBT nodes (and their environment).
5. How would you optimally merge two binary heaps?
6. Assuming you have a BST and a binary heap containing a set of integer keys, which structure would you choose to produce the list of all the keys in sorted order? Why?
7. Sketch two binomial heaps, containing keys {`a, i, d`} and {`c, e, f, g, h, b, j`}, respectively. Illustrate the result of merging them, and then extracting the minimum.

Exercises

1. Solve *Exercise 5* from the pre-sheet, if you haven't done so optimally already. [*Hint:* The desirable complexity is $O(\log n)$ per operation.]

2. The AVL tree[1] represents the pioneering attempt at creating a self-balancing BST. It maintains a single invariant: for each node, it holds that the heights of its two child subtrees[2] may differ by at most 1. Provide pseudocode, with careful explanation, for its insertion operation.

3. Derive the number of red, black (non-leaf) and leaf nodes of:

   (a) A *complete* RBT of height h;

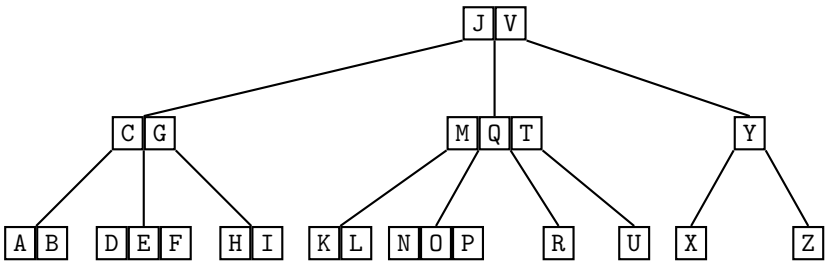
   (b) The *sparsest* possible RBT (smallest number of nodes) of height h;

   assuming the number of red nodes is as low as possible.

4. Considering B-trees whose nodes cannot hold more than three keys each:

   (a) Insert the following keys into an empty tree: `C A M B R I D G E X`.

   (b) Delete `M`, `Q`, and `Y`, in that order, from the following tree:



   Sketch the state of the tree after each significant step.

5. You are given a hash table with n keys and m slots, resolving collisions by chaining (with chains being no longer than L), and with each slot providing the length of its chain. Carefully explain an algorithm to return a *random* key from the table, with equal probabilities for all keys.

Implementation

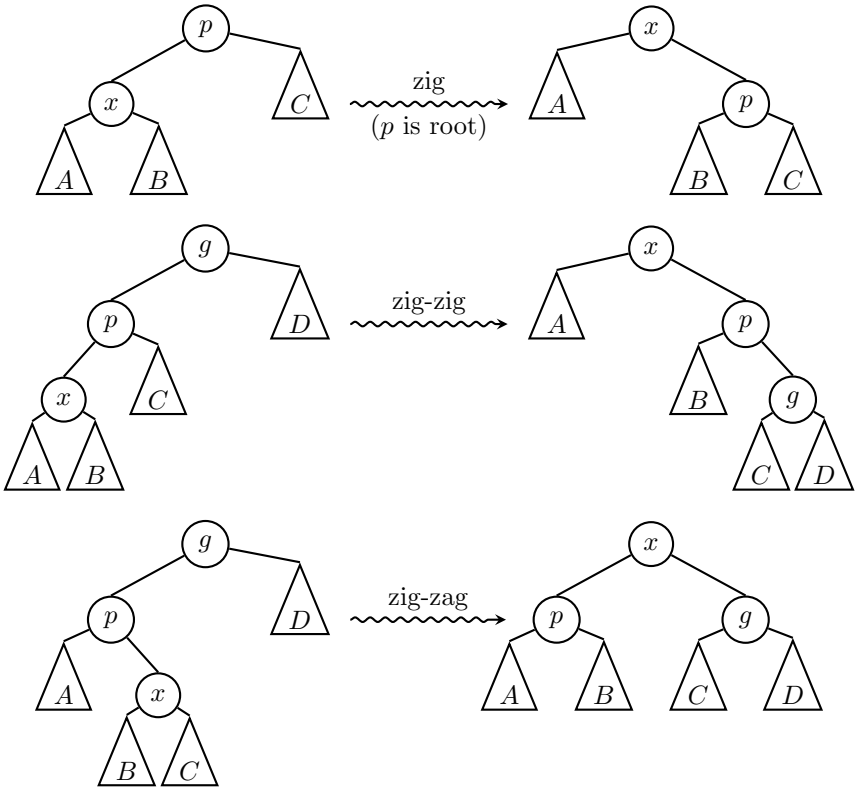
As illustrated by the RBT and AVL trees, a very important operation to master in order to be able to properly implement self-balancing BSTs is *tree rotation*. While easy to sketch, this operation can be deceptively difficult to implement properly, requiring one to keep track of dozens of pointers simultaneously.

In order to "break the ice", for this implementation exercise, we will focus on the *splay tree*[3] data structure. This is a very elegant self-balancing BST, primarily because it is guided by an intuitive line of thought, and requires no additional

[1] `https://en.wikipedia.org/wiki/AVL_tree` (read only *after* attempting!)
[2] Here we consider an empty subtree to have a height of zero.
[3] `https://en.wikipedia.org/wiki/Splay_tree`

bookkeeping (unlike AVL/RBT). The primary idea is *temporal locality of reference—"if I have recently accessed a key, there is good chance I will want to access it again!"*. This is the basis behind the `splay(x)` operation, which will perform a series of rotations on x in order to make it the new root of the tree (and therefore accessible in $O(1)$!). Even though it might be counterintuitive, if we do these rotations in a specific fashion, it is possible to guarantee $O(\log n)$ complexity[4] per operation.

The three cases for performing the rotations, each making x at most two levels higher, are as follows:



`splay(x)` repeatedly performs the appropriate rotation until x becomes root. For this exercise, you should implement the `insert(v)` and `find(v)` operations on the splay tree—these operations trivially apply `splay` on the node they create/locate—in the language of your choice.

[4] **N.B.** This is *amortized* complexity—individual operations may require linear time, but over the lifetime of the tree, m operations will always require $O(m \log n)$ time. You still don't have the required mathematical toolkit to actually do the analysis—but you will in a few lectures' time!

Job interviews

1. How would you augment the stack data structure with a `GetMaximum()` operation, which would return the largest item on the stack without modifying it?

2. Consider a binary tree in which all nodes are either leaves or have exactly two children. Given its pre-order and post-order traversal list, how would you reconstruct it?

3. Implement the *least-recently used cache* (LRU) data structure, representing a dictionary with the following operations:

   - `mk_cache(n)`: create a new LRU cache of capacity for n keys;
   - `get(k)`: obtain the value associated with the key k, if it exists;
   - `insert(k, v)`: inserts the mapping $k \to v$ into the cache (potentially overwriting, if k is already in the cache). If this operation would exceed the capacity of the cache, first remove the key that was least-recently accessed (via either `get` or `insert`).