WORKSHEET 07: AUGMENTED STRUCTURES

Binary trees are the easiest to reason about, but they are not always the most practical. In particular, if the data structure is very large, it may need to be read from the disk during the runtime. Reading from the disk happens in pages of fixed size (typically it is larger than the memory to store just one node or one database record). We need to store search trees as (m, n) trees, where m is the smallest number of successors and n is the largest number of successors.

7.1 Concepts and Facts

Definition: $(2, 4)$-trees are search trees where each node stores one/two/three keys and each non-leaf node has respectively two/three/four child nodes. For example, if $k_1 < k_2 < k_3$ are keys stored in some node, then one of the child pointers is to the left of k_1, another pointer is between k_1 and k_2, one more pointer is between k_2 and k_3, and the last pointer is to the right of k_3. (All the leaf nodes have null-children.)

Moreover, the nodes in $(2, 4)$-tree should be in searchable order (e.g. each subtree stores keys that fall in-between the two keys in the parent node), and all null-leaves are at the same depth. See 2-3-4 Tree in Wikipedia.

- In order to insert a key in such a tree, you find the last non-null node at the bottom where the key fits, and insert it among the existing keys. It might happen that a node overflows (has 4 keys and 5 null children). In this case it is split into two nodes - with 2 and 1 keys respectively (and one more key is promoted one level up). If ithe promoted key causes another overflow, you split again, and so on.
- In order to delete a key from such a tree, do the following steps:
  - If the key to be deleted is not a leaf, replace it by in-order successor (so that you only need to know how to delete leaves).
  - Otherwise proceed as in 2-3-4 Tree Deletion.

Definition: A tree is named a *Red-Black Tree*, if it is a Binary Search Tree, every node is either red or black (a flag stores its color) and it satisfies *red-black invariants*:

Root property: The root is black.

External property: Every leaf (a node with NULL key) is also black.

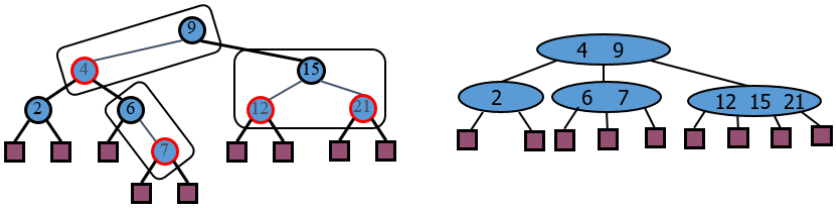
Internal property: If a node is red, then both its children are black.

Depth property: All simple paths from some node to its descendant leaves have the same number of black nodes.

Note: We can compute the black-height $h_{\text{black}}(v)$: it is 0 for all leaves, and for any other node v, it is the maximum of all $h_{\text{black}}(v_i)$ of its descendants v_i plus one.)

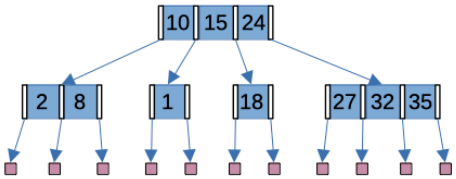
Statement: Each red-black tree can be uniquely represented as a (2,4)-tree. They are just differently drawn representations of the same concept.

Example: The picture below shows how a red-black tree can be converted into a (2,4) tree. Every black node (and optionally one or two of its red children) become keys in the new (2,4)-tree. The representation invariants of red-black trees ensures that the properties of (2,4)-tree are also satisfied.



7.2 Problems

Problem 1: Show how to insert a new node 31 into this $(2, 4)$-trees:



Problem 2: Build an example of (2,4)-tree, where the root has height equal to 3 and where deleting some key would cause the height to decrease.

Problem 3:

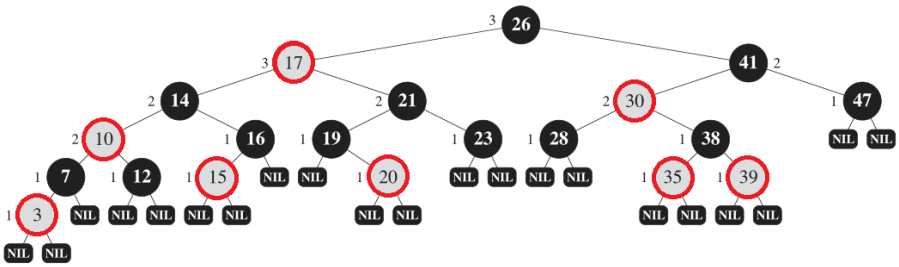


Fig. 1: Sample Red-Black Tree

(A) Compute the following three key values (u, v, and w):

$$\begin{cases} u = 3(a+b)+2 \\ v = 3(b+c)+1 \\ w = 3(c+a) \end{cases}$$

Here a, b, c are the last 3 digits of your Student ID.

Verify the black height of every node in the graph – all NULL leaves have black height equal to zero. Any other node has black height equal to the number of black nodes that are on some descendant path. (According to the depth property – the black height of any node should not depend on the path to the leaf we chose.)

(B) Show how the tree looks after the nodes u, v and w (in this order) are inserted in the Red-Black Tree shown in Figure *Sample Red-Black Tree*.

If any of the values u, v, w coincide with existing nodes, they should not be inserted. (Red-Black trees and BSTs in general can handle duplicates; but here we assume that it stores a map/set with unique keys.)

Show the intermediate steps – the tree after each successive inserted node. Clearly show, which are the red/black vertices in the submitted answers.

Note: Check that your inserts preserve the BST order invariant (along with all the Red-Black tree invariants). Secondly, try to follow the standard algorithm when inserting new nodes (still, preserving the invariants is more important).

Problem 4: Compare the following two implementations: (1) heaps, (2) AVL trees to implement the following ADTs. For each method find the worst-case (or amortized) time complexity $\Theta(g(n))$.

(A) Priority Queue ADT:

$Q =$ newEmptyQueue()
Q.INSERT(x)
$x = Q$.DELETEMIN()
$x = Q$.FINDMIN()

(B) Predecessor/Successor ADT:

$S =$ NEWEMPTYCONTAINER()
S.INSERT(x)
S.DELETE(x)
$y = S$.PREDECESSOR(x) – return the reference to the next-smaller than x
$y = S$.SUCCESSOR(x) – return the reference to the next-larger than x

Problem 5: Assume that you need to build a *range index* data structure R – this data structure is a database-like container where we can insert items x_i (such as page requests for Google Analytics). Each item x has some numeric key $x.k$ (the timestamp of the request or, perhaps, the milliseconds it took to compute the HTTP response). We need to query this data to draw nice graphs – e.g. display barcharts counting the number of requests in each value range or to find the total time spent for requests received during some time period or anything else.

(A) Pick some data-structure to support *range index* described above. Give the time estimate for R.INSERT(x), R.DELETE(x), $x = R$.FINDBYKEY(k).

(B) Give the time estimate for R.FINDMIN() and R.FINDMAX() to find the minimum and the maximum key in the whole data structure R.

(C) Assume that the data structure also supports operation $R.rank(k)$ defined as the number of keys in the index that are smaller or equal to the given value k. Write a pseudocode to compute another operation – R.COUNT(k_1, k_2) that returns the number of keys k in-between, i.e. satisfying $k_1 \leq k \leq k_2$. (The operation R.COUNT(k_1, k_2) would be very useful to display barcharts for Google Analytics or similar aggregated data.)

(D) In order to implement $R.rank(k)$ from the previous task, you can augment the items x stored in R by storing additional numerical information (denoted by $x.\gamma$). Which kind of augmented information would you use? Consider the following options (plus any others you might need):

- the minimum key in the subtree rooted at node
- the maximum key in the subtree rooted at node
- the height of the subtree rooted at node
- the number of nodes in the subtree rooted at node
- the rank of node
- the sum of keys in the subtree roted at node

(E) Provide a way to compute $R.rank(k)$ from the values $x.\gamma$ you selected in the previous item.