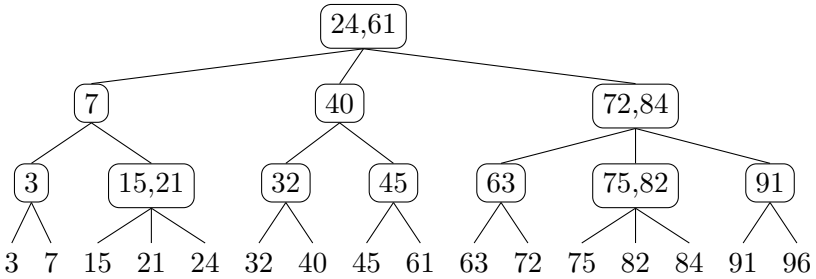
Fundamental Algorithms, Lecture Notes. 2–3 Trees

2–3 trees are one instance of a class of data structures called balanced trees. These data structures provide an efficient worst case instantiation for the Dictionary abstract data type. Recall that a dictionary supports the operations SEARCH, INSERT, DELETE on a set of items drawn from an ordered collection U, called the *universe*. The balanced trees support each operation in worst case time $O(\log n)$, where n is the number of items currently stored in the dictionary.

There are many different balanced search trees including red-black trees, AVL trees, splay trees (they have somewhat different performance guarantees), skip lists (here the performance guarantee is an expected time bound, and B-trees (they are widely used in databases for data stored on disk). What is special about 2-3 trees? First of all, it is easy to remember a few key facts from which one can deduce the details of the data structure procedures. Second, they are essentially B-trees, but adapted for main memory, and so more or less yield procedures for B-trees too.

A 2–3 tree is a tree in which all the leaves are at the same level and each internal node has either 2 or 3 children. All the items stored in the dictionary are held in the leaves. Each leaf stores one item. Recall that each item includes a key; the keys are used to order the items. Traversing the leaves of the 2–3 tree in left-to-right order yields the items in sorted order.

Each internal node of the 2–3 tree stores copies of one or two keys, called *guides*. The guides are copies of the largest key in each subtree apart from the rightmost subtree. We show an example below.



The minimum and maximum number of nodes at depth k are 2^k and 3^k, respectively. Thus, a 2–3 tree which stores n items has an edge height between $\lceil \log_3 n \rceil = \lceil \log n / \log_2 3 \rceil$ and $\lfloor \log n \rfloor$. So the path length from the root to any leaf is $O(\log n)$. Each operation requires the traversal of a path from the root to a leaf and possibly back to the root and hence requires $O(\log n)$ time.

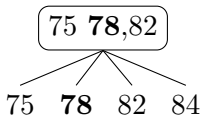
A search in a 2–3 tree proceeds in essentially the same way as in a binary search tree. To be precise, the search starts at the root and follows the path in the 2–3 tree that ends at the node storing the item being sought, or if the item is not present, at the node storing the item's immediate successor, the smallest item larger than the item being sought. There is a special case: if the item is larger than every item in the 2–3 tree, then this successor is **nil**.

For example, in the above tree, a search for item 45 proceeds as follows. At the root, the value 45 is compared to the middle guide of value 61, and as $45 \leq 61$ it is then compared to the left guide. As $45 > 24$, the search continues in the middle subtree. Now 45 is compared to 61 and to 40, yielding $40 < 45 \leq 61$, and the search goes to the right subtree. Here, the two comparisons yield $45 \leq 45$, causing the search to go to the left subtree, which is the leaf holding the item with key 45. Note that each comparison is of the form key $\leq$ guide. The reason for this is that the guide is the largest item in a subtree, and hence the search should proceed to the right of this subtree only if key $>$ guide.

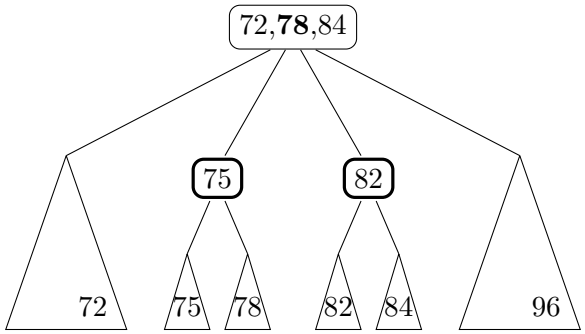
Next, we describe how to perform insertions and deletions.

Insertion An insertion begins by performing a search to determine where the item would be located (if it were present in the tree). The item is inserted by creating a leaf at this location and adding the item to the leaf. If this results in the parent having three children (because it had two children previously) we copy the new key value to the parent node to provide the guide corresponding to the new leaf and the insertion is complete. Note that the new item is never the rightmost child of its parent, unless it is the rightmost item in the whole tree; we will discuss later what needs to be done to the guides in this case. Otherwise, the parent now has four children (as it had three children previously); we again copy the new key value to the parent, so it has three guides, one for each child except the rightmost. Then the parent node is replaced by two nodes, each of which receives two of the four children at hand; this is called a *node partition*; the two new nodes become children of the grandparent of the new leaf node. Of course, the left to right order of the leaves is maintained. The second guide in left to right order is moved to the grandparent, providing it with the guide for its extra subtree. Moving up a level in the tree, if the grandparent now has three children the insertion is complete; if it has four children, the grandparent is split into two nodes, each of which receives two of the children. Again, the left to right order of the children is maintained. This process is then repeated at each successively higher level of the tree along the path from the inserted item to the root, as needed to remove all nodes with four children. A special case arises if the root is replaced by two nodes; then a new root node is created; its children are the two nodes newly formed from the old root.

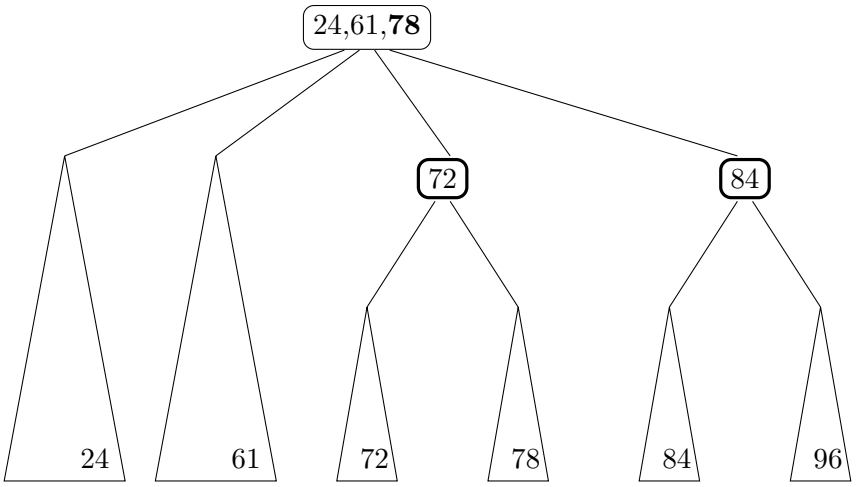
We illustrate by inserting item 78 into the 2–3 tree on the previous page. The first step creates a new leaf with item 78, changing its parent node to temporarily have 4 children, as follows. Changes are shown in bold. Also, in each subtree, the one value being shown is the largest value in the subtree.



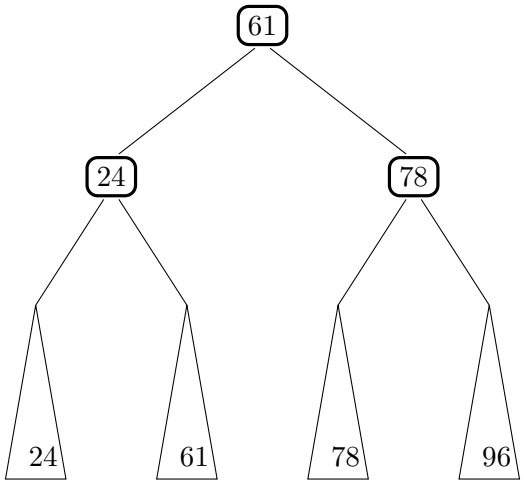
Next, we split the node with 4 children, causing it parent to acquire a 4-th child, as shown below:



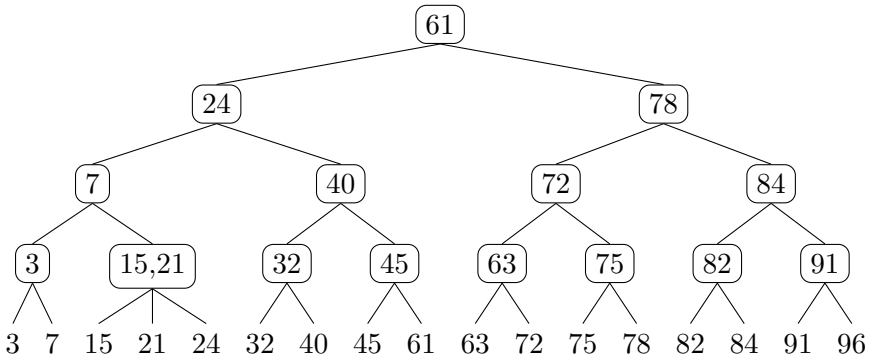
Next, we split the node which now has 4 children, leading to:



We continue by splitting the node which now has 4 children. This is the root node, and once it is split we need to create a parent for the two resulting nodes, i.e. a new root for the tree.



Thus the resulting 2–3 tree has the following form.



The following invariant (assertion) about the state of the 2–3 tree is maintained:
Invariant Right after a node is updated the guides are correct, meaning that each node stores a copy of the largest value in each of its subtrees apart from the rightmost.

The only violation of the 2–3 definition is that one node may have 4 childeren. When a 4-child node v is split into two nodes one of its guides is moved to its parent, thereby maintaining the invariant. The guide that needs to be moved is the second guide in left to right order in node v. See Figure 1. Note that there will be one or two guides at the parent node already; these are not shown in the figure. The one exception is when the parent node is a new root node (and therefore it had no guides previously).

There is one final detail. If the item being inserted has value w greater than the largest value v currently stored in the 2–3 tree, then, the new guide at the parent of the new leaf is the key value for its second rightmost item, namely v.

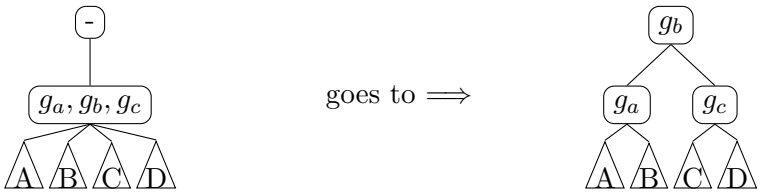


Figure 1: Updating guides when splitting a node. g_a is the largest key in subtree A, g_b the largest key in subtree B, g_c the largest key in subtree C.

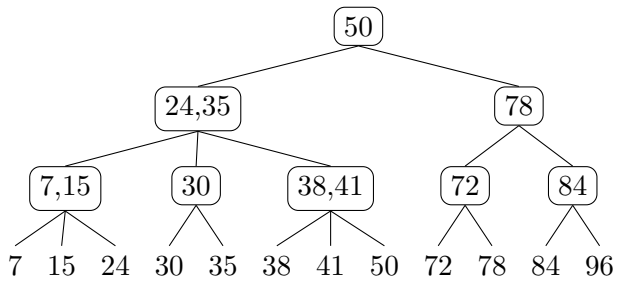
Deletion A deletion has a similar flavor. Again, a search is performed to find the item (stored at a leaf). The leaf is deleted. We will explain later how to manage the guides. If the parent now has two children the deletion is complete.

Otherwise, the parent now has one child. If one of the parent's adjacent siblings (it may have only one) has three children, the two nodes will repartition their four children so that they have two each (for example, if the sibling s is to the left of the one-child node v, v acquires the rightmost child of s). (To fully specify the algorithm, let us say that first the sibling on the left, if any, is checked, and then the sibling on the right, if any.) Otherwise, the parent and its sibling are combined into one node, which now has a total of three children.

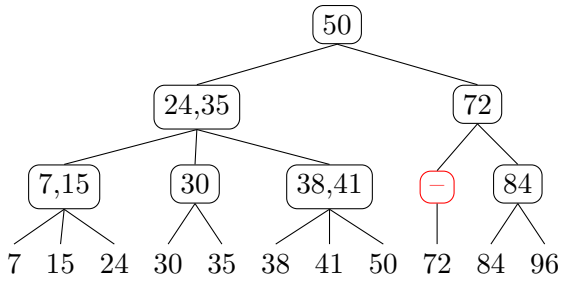
Now the grandparent g has one fewer children. If g has only one child, g's siblings are checked; if a sibling s has three children, s gives g one of its children; if not s and g are merged into a single node.

This process is repeated at each successively higher level of the tree along the path from the deleted item to the root, as needed to remove all nodes with one child. If the root ends up with one child, the root is simply removed and its sole child becomes the new root.

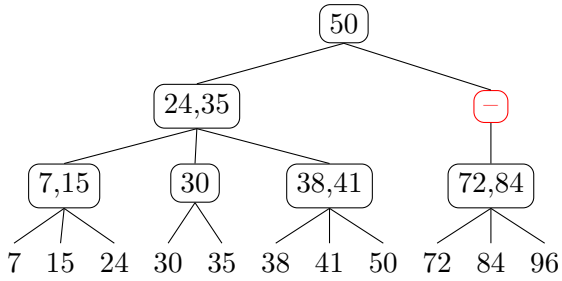
We illustrate the deletion of item 78 on the following tree.



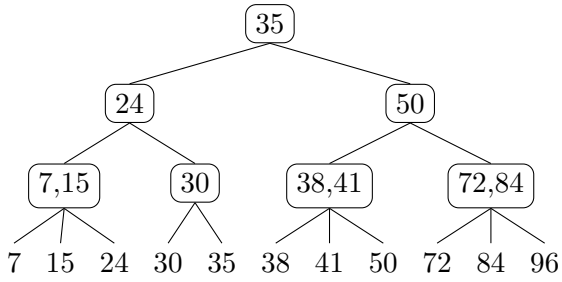
The removal of the leaf with value 78 results in its parent, shown in red, having only 1 child, as shown below. Also, we need to replace the guide with value 78, by the guide with value 72.



As its only sibling has just 2 children, the nodes are combined, resulting in the grandparent,shown in red, having only one child, as shown below.



Now, the grandparent needs to take a child from its left sibling. This yields the following tree.



The guides are managed as follows. If the deleted item is not the rightmost child of its parent, the guide with its value is simply removed from the parent. If it is the rightmost, the guide equal to the value of the second rightmost child needs to be removed from the parent. However, in this case, this removed key needs to replace the key of the deleted item where it occurred on the search path. One easy way to handle this is based on the following observation: unless the deleted item is the rightmost item in the tree, we will encounter a guide with its value on the access path. We keep a pointer to this guide allowing the replacement to be performed in $O(1)$ time.

There is one special case: if the item being deleted is the rightmost item in the tree, then a guide with its value is not present, and therefore the guide with value equal to the previously second rightmost item is simply removed.

We maintain essentially the same invariant as for the insertions, namely:

Invariant The guides are correct, meaning that after each node update, every node stores a copy of the largest value in each of its subtrees apart from the rightmost one.

Initially, when we remove the leaf node, if it is not the rightmost child of its parent, we remove the guide for this leaf from the parent. Otherwise, we remove the guide for its immediate predecessor from the parent, and copy this predecessor guide to the location of the guide for the deleted item.

When two nodes u and v are combined, creating a node with three subtrees, the node needs to end up with two guides, holding the rightmost (largest) values in its two leftmost subtrees. Before the join, these three subtrees were subtrees of u and v, and therefore one of these two guides had been stored at the parent w of u and v. So it is straightforward to transfer this guide in $O(1)$ time. It may be that there was a second guide stored in w which remains there. (See the illustrative figure below.)

Again, when two nodes u and v redistribute their subtrees (or equivalently, their children),u and v have four subtrees between them; two guides will be stored at one of u and v, zero at the other, and one guide will be stored at w, the parent of u and v. The final situation rearranges these guides, so that there is one guide each at u, v, and w. we need to move a guide from the parent to the child receiving a subtree, and we need to transfer a guide from the node losing a subtree to the parent. Again, it is straightforward to transfer this guide in $O(1)$ time. Also, once more, it may be that there was a second guide stored in w which remains there.

The invariant is maintained in both cases.

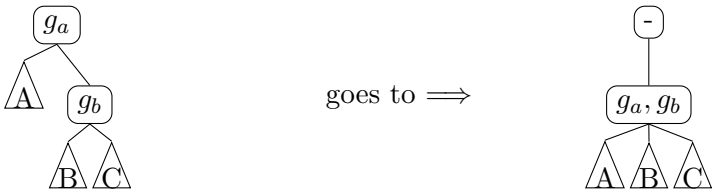


Figure 2: Updating guides when joining two nodes; g_a is the largest key in subtree A, g_b the largest key in subtree B

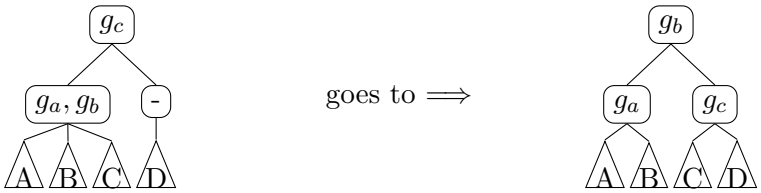


Figure 3: Updating guides when redistributing subtrees; g_a is the largest key in subtree A, g_b the largest key in subtree B, g_c the largest key in subtree C.

Augmenting 2–3 Trees

By storing additional information at their internal nodes, 2–3 trees can support additional operations in $O(\log n)$ time.

Example 1. Consider the query "report the kth smallest item in rank order". To answer this query it suffices to store at each internal node the number of items in each of its subtrees. Suppose the root has three subtrees holding n_1, n_2, and $n - (n_1 + n_2)$ items, respectively. If $n_1 < k \leq n_2$, the search continues in the middle subtree, seeking the $(k - n_1)$ the smallest item in that subtree. The other cases are similar. To complete the solution, one has to show how to maintain the size information at internal nodes when items are inserted or deleted. If the only change to the structure of the

2–3 tree is to add (or delete) a leaf, the only counts that need to change are those for the subtrees into which the access descends, and these counts are incremented (or respectively decremented) by 1. If some of the internal nodes change (due to being split or combined) the counts need to be appropriately updated; this can be done for each changed or new node, for its size is simply the size of its subtrees, and can be obtained by summing the subtree sizes.

As with the guides, it actually suffices to keep counts at each node for all but its rightmost subtree. However, describing the needed updates to the counts is a bit more involved, so we leave this to the interested reader.

Example 2. Suppose that in addition to their key field, items have a second attribute, color say, which can take on the values red or green. Suppose that a query asks for the number of green items less than a key value k. The following additional information is stored at each internal node: for each of its subtrees it keeps the count of the number of green items stored in the subtree. This is maintained in essentially the same way as the counts of the number of items in Example 1. To answer a query with respect to key value k, the search path from the root to the leaf location where k is stored (or if not in the 2–3 tree, where k ought to be stored) is traced. Then, the counts of green nodes for each subtree hanging to the left from this path are summed (these counts are stored in the internal nodes on the search path). This computation entails summing at most $2 \log n$ counts and consequently the search takes $O(\log n)$ time.

Example 3. Suppose that the query asks for the fraction of green items among items with key value less than k, that is (number of green items with key value less than k)/(number of items with key value less than k). Clearly, this can be easily done by maintaining two counts: the number of items, as in Example 1, and the number of green items, as in Example 2.

Example 4. Again, suppose that the items have a key value and a color attribute as in Examples 2 and 3. But now suppose that there is a new operation FLIP(k_1, k_2) which switches the color of all items with key value betwen k_1 and k_2 inclusive. Also suppose that the query operation is called Color(k); it asks what is the color of the item with key value k, if such an item is present. Of course, insertions and deletions continue to be supported.

Here, for each subtree, we store an additional *flip* bit. If the bit value is "1" this means that the recorded color of all items in the subtree needs to be flipped to obtain the actual color. More formally, for an item e with recorded color c, its actual color is c if the logical "or" of the following flip bits has value 0: all the flip bits for the subtrees containing e on the path from the 2–3 tree root to e; and its actual value is $\bar{c}$, the other color, if the logical "or" of these bits has value 1. As usual, the flip bit for a subtree is stored at the parent of the subtree.

Given the flip bits, it is straightforward to answer a color query in $O(\log n)$ time. Maintaining the flip bits when an insertion or deletion is performed is also straightforward. To facilitate the description that follows, we suppose that a flip bit for a subtree is associated with the edge from its parent. When a node v is split, the flip bit for the edge from v's parent is duplicated: both of the new edges get this value for their flip bit. More care is needed in the case of deletions. This is left as an exercise for the reader.

It remains to explain how to perform the operation FLIP(k_1, k_2). Paths to the items e_1 and e_2 with key values k_1 and k_2 are traced by means of two searches (or if either of these items is not present, to the locations where these items would be found). Then the colors of these two items are flipped as are the flip bits for all the subtrees hanging on the inside from the two paths leading to items e_1 and e_2. By the "inside" we mean subtrees to the right of the path leading to e_1 and to the left of the path leading to e_2. As this entails updating $O(\log n)$ flip bits, the FLIP operation takes $O(\log n)$ time.

Additional Operations

In fact, 2–3 trees support two other operations: SPLIT and JOIN, both also running in $O(\log n)$ time. SPLIT(T, k) produces two 2–3 trees, T_1 and T_2, respectively comprising the items smaller than k, and greater than or equal to k in tree T (we assume all items have distinct keys); T is not preserved by this operation. JOIN(T_1, T_2) produces a 2–3 tree T comprising the items in T_1 followed by the items in T_2 (this operation is defined only if the smallest item in T_2 is larger than the largest item in T_1); again, T_1 and T_2 are not preserved by the operation.

JOIN(T_1, T_2) is implemented as follows. Let h_i be the height of T_i, $i = 1, 2$. If $h_1 = h_2$, then the new tree is formed by creating a root whose two children are the roots of T_1 and T_2. Otherwise, without loss of generality, suppose that $h_1 < h_2$. Consider the path in T_2 to the leftmost leaf; descend to the node, v, at height $h_1 + 1$; node v is given a new child, namely the root of T_1. The JOIN then proceeds in the same way as an insertion (namely, if a node has four children, the node is split).

SPLIT(T, k) is implemented as follows. The path to the item with key k is traversed. This path is removed from the 2–3 tree. Each subtree whose parent lies on this path is placed on one of two stacks: a stack of subtrees to the left of k, and a stack of subtrees to the right of k, plus a one-node subtree for item k if present in the tree. The subtrees in each stack are joined together. It simplifies our analysis if the heights of the trees on the stack are strictly decreasing. But this need not be true. So we modify the process by which subtrees are added to a stack: if the subtree T'' we want to add to a stack has the same height as the subtree T' currently at the top of the stack, then T' is popped from the stack and T' and T'' are joined as before, forming T'''. We then iteratively seek to add T''' to the stack.

Now, we join the trees on the stack in stack order (so the subtrees are joined in reverse height order). In general, we will be joining a tree T'' resulting from the joins performed so far, with the tree T' on the top of the stack. We will maintain the invariant that $ht(T') \geq ht(T'')$. If this is true, then the height of the tree resulting from the join is at most $ht(T') + 1$. Note that the height of the next tree on the stack, now at the top of the stack, is at least $ht(T') + 1$, since the heights on the stack, from top to bottom, were strictly increasing; this shows the invariant is maintained. Each join takes time proportional to the height difference between the two subtrees, plus $O(1)$ to account for the case that the two subtrees are of equal height. The sum of these differences is at most the height of the tree at the bottom of the stack, and this height is $O(\log n)$. As there are $O(\log n)$ trees being joined in total, the time to join all the subtrees to the left of k is $O(\log n)$. The same is true for the cost of joining the subtrees to the right of k, plus the subtree for k, if present.

Exercises

1. Show how the counts in Example 1 are updated when the internal nodes change (by being combined or split).

2. Show how to support the query "how many items have key values larger than k" in $O(\log n)$ time in Example 2.

3. Consider maintaining a collection of linked lists on which the following operations can be performed, where each list contains at most n items.

   1. List L is the concatenation of lists L_1 and L_2; this operation destroys L_1 and L_2.

   2. List L is split into two lists, L_1 comprising the first k items in L, and L_2 comprising the remainder. This operation destroys L.

3. Create a new one item list.

4. Report the first item in list L.

(a) Give a data structure implementing linked lists so that each of the operations $(1), (2), (4)$ can be performed in $O(\log n)$ time and operation (3) in constant time, where n is the size of the linked list L.
(b) Suppose we also allow the operation Reverse(L) which reverses the links in list L. Show how to support this operation in addition, also in $O(\log n)$ time.

4. Consider maintaining a database for a biologist storing results concerning pea plant experiments. Each experiment records a distinct (integer) ID, a plant weight and a plant height. We would like to support the following query:

for plants with heights in the range $[h_1, h_2]$, what is the average weight?

Also insertions and deletions of experiments will be allowed.

a. Show how to support these operations in time $O(\log n)$, where n is the database size.

b. Suppose, in addition, we wish to support the following query:

for plants with weights in the range $[w_1, w_2]$, what is the average height?

Show how to support this query in addition, also in time $O(\log n)$.
Hint: Use multiple 2–3 trees.

5. Suppose a bank has a collection of accounts, each identified with a (distinct) owner name, and each having a value, namely the balance. The bank wishes to support the following operations:

1. Create an account.
2. Close an account.
3. Add (or subtract) a given sum from the balance for an account identified by the owner's name.
4. Report the account with maximum balance.
5. Report the balance to an owner on request.

Show how to support all these operations in $O(\log n)$ time.