Notes on 2-3 trees

(version 0.7)

Victor Shoup

1 Introduction

2-3 trees are a form of balanced tree that can be used to implement a dictionary, as well as other types of operations.

Recall that in its most basic form, a dictionary maintains a set of key/value pairs, and supports the following operations:

- *insert*: insert a given key/value pair into the data structure; if there is already a pair with same key component, then simply update the value component;
- *search*: given a key, determine if there is a key/value pair with a matching key component, and if so, return the corresponding value component;
- *delete*: given a key, delete the key/value pair, if any, with matching key component.

There are many data structures that may be used to implement a dictionary, such as binary search trees, hash tables, etc. 2-3 trees are one such data structure. As we shall see, if we implement a dictionary using 2-3 trees, then all three of the dictionary operations take time $O(\log n)$, where n is the number of key/value pairs in the dictionary. We will also see how to use 2-3 trees to implement other types of operations, beyond those of a simple dictionary.

2 2-3 trees: the basics

Before we start, a word of warning: the exposition here of 2-3 trees is a little bit different from what one usually sees in the literature. See Section 6 below for more on this.

Also, before we start, recall that the *depth* of a given node in a tree is the length of the path (i.e., the number of edges in the path) from the root to that node. Also recall that the *height* of a tree is the maximum depth of any node in the tree.

Now, at a high level, our notion of a 2-3 tree is that of a tree with the following properties:

- key/value pairs stored only at leaves (with no duplicate keys);
- all leaves are at the same depth;
- looking at the leaves from left to right, keys appear in sorted order;
- each internal node

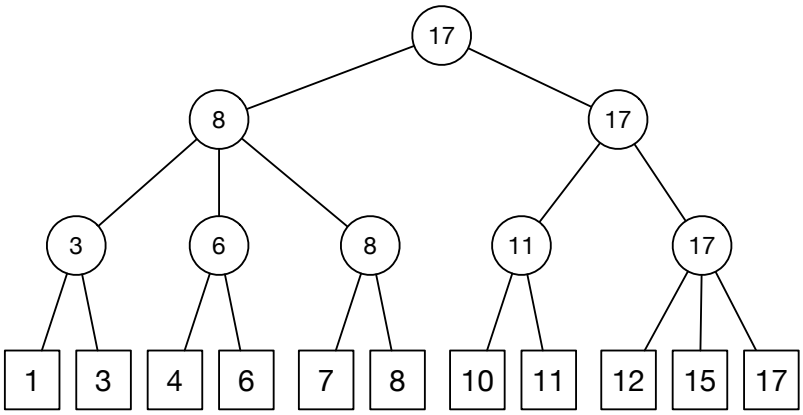


Figure 1: A 2-3 tree

- has either 2 or 3 children,
- and also has a "guide", which is the largest key that appears in its subtree.

See Figure 1 for an example of a 2-3 tree. In this tree, the keys are integer values. In this figure, each leaf is labeled with the key stored at that leaf — the corresponding value is not shown. Each internal node is labeled with the guide stored at that node. We see that the height of this tree is 3; moreover, every leaf is at the same depth, namely 3, as is required for any 2-3 tree.

Claim 1. *Consider a 2-3 tree with n leaves and height h. We have $n \geq 2^h$.*

Taking logs, the inequality in Claim 1 is equivalent to $h \leq \log_2 n$. As we shall see, the running times of the essential algorithms on a 2-3 tree are proportional to the height of the tree, and so this claim will give a very nice bound on the running times of these algorithms.

We prove Claim 1 by induction on the height of the tree. To be more precise, for $h \geq 0$, let P_h be the proposition:

For every 2-3 tree T of height h, the number of leaves in T is at least 2^h.

We prove that P_h holds for all $h \geq 0$ by induction on h.

Consider the base case $h = 0$. The only tree of height $h = 0$ has a single node, which is a leaf, so the number of leaves is 1, which is equal to 2^h. So that proves the claim for $h = 0$.

For the induction step, suppose $h > 0$ and assume that the induction hypothesis P_{h-1} holds. From this assumption, we want to show that P_h must also hold. To this end, let T be any tree of height h, and suppose that T has n leaves. It will suffice to show that $n \geq 2^h$. To do this, we observe that T consists of a root, which has either 2 or 3 children. Moreover, each of these children is itself the root of a 2-3 tree of height $h - 1$. Let us call these subtrees T_1, T_2, and (if it exists) T_3. See Figure 2. Let n_i be the number of leaves in

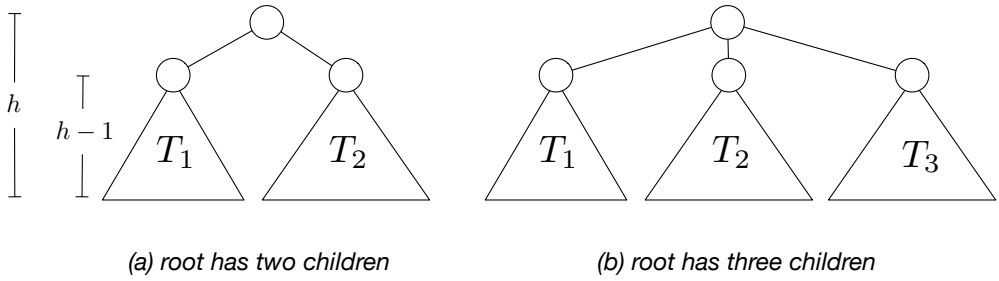


Figure 2: Proof of Claim 1

subtree T_i. We have

$$\begin{aligned} n &= \sum_i n_i \\ &\geq \sum_i 2^{h-1} \quad \text{(by the induction hypothesis, } P_{h-1}\text{, we have } n_i \geq 2^{h-1}\text{)} \\ &\geq 2 \cdot 2^{h-1} \quad \text{(since there are at least two subtrees)} \\ &= 2^h. \end{aligned}$$

That completes the proof of Claim 1.

Claim 2. *The number of internal nodes in a 2-3 tree with n leaves is at most $n - 1$.*

This bound tells us that the overhead, in terms of space, of a 2-3 tree is linear in the number of keys. We leave the proof of Claim 2 as an exercise for the reader. One can prove it by induction on height, using a proof with essentially the same structure as the proof of Claim 1.

3 Data Layout and Search

Here is one possible way in which we could lay out the data in a 2-3 tree. In the following, we use *Java*-like syntax:

```
class Node {
    KeyType guide;
    // guide points to max key in subtree rooted at node
}
class InternalNode extends Node {
    Node child0, child1, child2;
    // child0 and child1 are always non-null
    // child2 is null iff node has only 2 children
}
class LeafNode extends Node {
    // guide points to the key
    ValueType value;
}
```

The idea is that every node, both internal node and leaf, contains a guide. At a leaf node, we have both a guide and a value, and the guide is really the key. For an internal node, the guide is always the largest key that appears in any leaf in the subtree rooted at that node.

With this data layout, here is how we can implement the *search* operation, which searches for a given key x. Again, using a *Java*-like syntax:

```
Search(x):   // use guides to search for the key x
p ← root of tree
h ← height of tree
repeat h times
    // p points to an internal node
    if  x ≤ p.child0.guide  then
        p ← p.child0
    else if  p.child2 = null or x ≤ p.child1.guide  then
        p ← p.child1
    else
        p ← p.child2

// p now points to a leaf node
if  x = p.guide  then
    return p.value
else
    return null   // or some default value
```

In the above pseudo-code, we write "$\leftarrow$" for assignment and we write "$a = b$" to test if keys a and b are equal. We also write "$a \leq b$" to test if key a comes before key b (or is equal to key b) in whatever natural order is used to compare keys. This may not be the exact notation that is used in a given programming language. For example, in *Java*, if keys are of type *String*, one would use the *compareTo* method to compare keys.

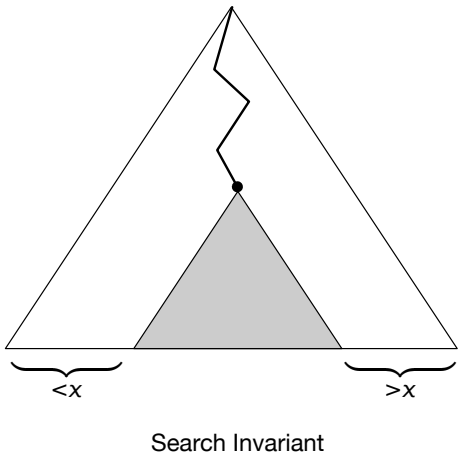


Figure 3: Search invariant

To understand why this search algorithm works, it is helpful to characterize its behavior in terms of a *loop invariant*, i.e., a condition that holds with each loop iteration. In general, to show that a loop invariant holds, one shows that it holds before the first loop iteration, and then one also shows that if it holds at the beginning of one loop iteration, it must also hold at the beginning of the next. Thus, showing that a loop invariant holds is basically the same as a proof by induction on the number of loop iterations.

For the above search algorithm, one considers the subtree rooted at p in any loop iteration, which partitions the keys at the leaves into three sets: those in the subtree, those to the left of the subtree, and those to the right of the subtree. The loop invariant is the following:

> *all the keys to the left of the subtree are less than x, and all the keys to the right of the subtree are greater than x.*

See Figure 3.

Now, the loop invariant holds trivially before the first iteration of the loop, as there are no keys to the left or the right of the subtree. Now suppose that the loop invariant holds at the beginning of one loop iteration. We want to show that it holds at the beginning of the next. So consider which way the if/then/else goes:

- If $x \leq p$.child0.guide, we update p to be p.child0. The keys to the left of the subtree stay the same, while we add to the keys to the right of the subtree those keys that were in the subtrees rooted at p.child1 and p.child2. But because of the way the guides are maintained and the keys are ordered at the leaves, all of these keys must be $> x$, and so the loop invariant is maintained.

- If $x > p$.child0.guide and p has only two children, then we will set p to p.child1. So we will add to the keys to the left of the subtree all those keys in the subtree rooted at p.child0, but by definition, all of those keys are $\leq p$.child0.guide $< x$. The keys to the right of the subtree will stay the same. So again, the loop invariant is maintained.

- Now suppose $x > p$.child0.guide and p has three children. There are two subcases:
  - $x \leq p$.child1.guide, or
  - $x > p$.child1.guide.

  We invite the reader to verify that the loop invariant is maintained in each of these two subcases.

With this loop invariant, we see that *with each loop iteration, if there is a leaf anywhere in the tree containing the key x, then it must be in the subtree rooted at p.* In particular, when the loop terminates, if there is a leaf anywhere in the tree containing the key x, then it must be at the leaf p. Of course, x may not appear in the tree at all, which is why we have to explicitly test if $x = p$.guide after the loop.

That completes the *correctness analysis* of the algorithm. As for the running time, we note that the amount of work done by the algorithm is proportional to the height of the tree, and so by Claim 1, this is $O(\log n)$, where n is the number of leaves in the tree.

A recursive search routine

One can also formulate the search routine in a recursive fashion, as follows. Here, one initially invokes $RecSearch(p, x, h)$, where $p =$ root of tree and $h =$ height of tree.

```
RecSearch(p, x, h):
if h = 0 then
        // p points to a leaf node
        if  x = p.guide  then
                return p.value
        else
                return null    // or some default value
else
        // p points to an internal node
        if  x ≤ p.child0.guide  then
                return RecSearch(p.child0, x, h − 1)
        else if  p.child2 = null or x ≤ p.child1.guide  then
                return RecSearch(p.child1, x, h − 1)
        else
                return RecSearch(p.child2, x, h − 1)
```

4 Insertion

The algorithm for insertion is conceptually quite simple, even if the detailed logic is perhaps a bit cumbersome.

Suppose we want to insert a key x into the tree. The first thing we do is perform a search for x, as above. If we find that x is already in the tree, then we update its corresponding value, if necessary. Otherwise, suppose the search for x terminated at a some leaf, and let p be the parent node of this leaf. We add a new leaf with key x as a new child of p. If p

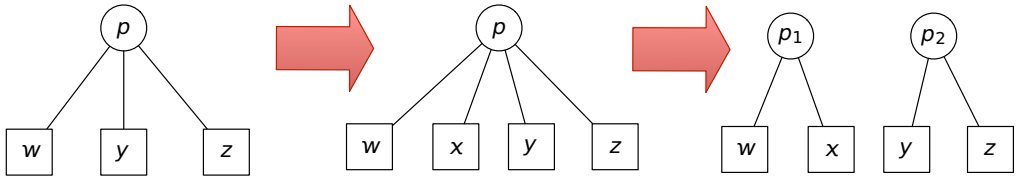


Figure 4: Splitting a node during insertion

originally had two children, then it now has three, and we are done. However, if p originally had three children, then it now has four, and the tree is no longer a 2-3 tree. So, we have to repair the tree.

To make this repair, we split p into two nodes, p_1 and p_2, where the first two children of p become children of p_1 and the last two children of p become children of p_2. This splitting procedure is illustrated in Figure 4.

But now observe that we replaced p by two nodes, p_1 and p_2, and so p's parent now has an additional child. If p's parent originally had two children, then it now has three, and we are done. However, if p's parent originally had three children, then it now has four, and again the tree is no longer a 2-3 tree. So, again, we have to repair the tree. We do this in the same way as above, splitting p's parent into two nodes, and dividing up its four children equally between these two nodes. This process may repeat, as now p's parent's parent may have four children. In the worst case, we will perform a node splitting operation on every node in the path from p to the root. If we have to split the root, then we create a new root with two children: this is the only time the height of a 2-3 tree increases.

The running time of the insert operation is proportional to the height of the tree, and so it is $O(\log n)$, where n is the number of leaves.

In the above discussion, we have completely ignored the extra bookkeeping that must be done to maintain the guides. However, it is not hard to see that the only guides that need adjustment are those on the path from p to the root, and so the cost of maintaining the guides is also just $O(\log n)$.

See Figure 5 for an example of how a tree grows during a sequence of insertions. Guides are not shown. We point out one technicality. Look at the point where we insert 10. We could have also made 10 a sibling of 8, and this would result in a valid 2-3 tree. However, this is *not* what the insertion algorithm would do. This insertion algorithm *must* perform the search for 10, which invariably takes it into the right subtree of the root. Indeed, if the tree already contained the key 9, then making 10 a sibling of 8 would result in an invalid 2-3 tree.

5 Deletion

Deletion is just slightly more complicated than insertion. Suppose we want to delete the leaf (if any) containing the key x. As with insertion, we begin by searching for x. If it is not found, there is nothing to do. So suppose it is found at some leaf, and let p be its parent. Now, we delete the leaf containing x. If p originally had three children, it now has two, and

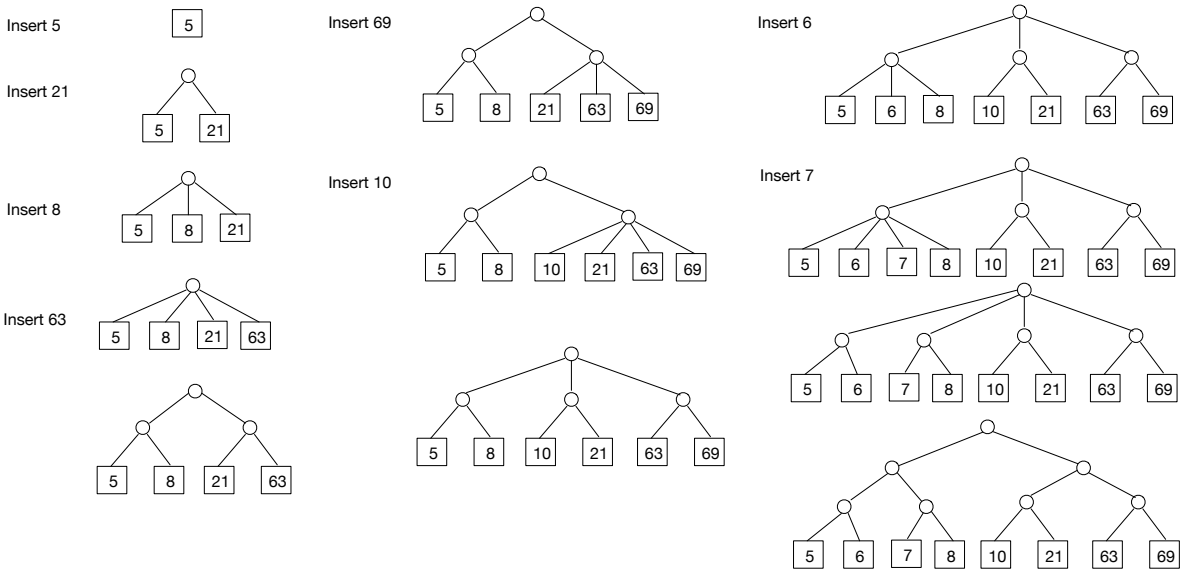


Figure 5: Insertion example

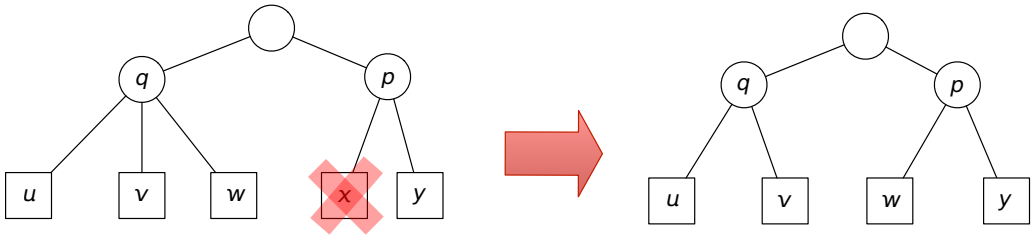


Figure 6: Deletion: adopting a child

we are done. However, if p originally had two children, it now has only one, and so we have to repair the tree.

To affect this repair, we first consider the trivial case where p is the root. In this case, we simply delete p, and the only child of p becomes the new root.

So assume that p is not the root. So p has a parent and also has one or two siblings. If one of p's *adjacent* siblings q has three children, then p "adopts" one of q's children, which restores the structure of the 2-3 tree. See Figure 6.

If none of the above holds, then p must have an adjacent sibling q with only two children. So the strategy now is to have p "give away" its only child to q. See Figure 7. But now p has no children, and so we delete p from the tree. However, this reduces the child count of p's parent, and so we have to repeat this whole process p's parent: it either adopts a child, or gives away a child and is itself deleted. This process may repeat for p's parent's parent, and so on. If we reach the root and the root is left with an only child, then we simply delete the root, and its only child becomes the new root. This is how the height of a 2-3 tree may

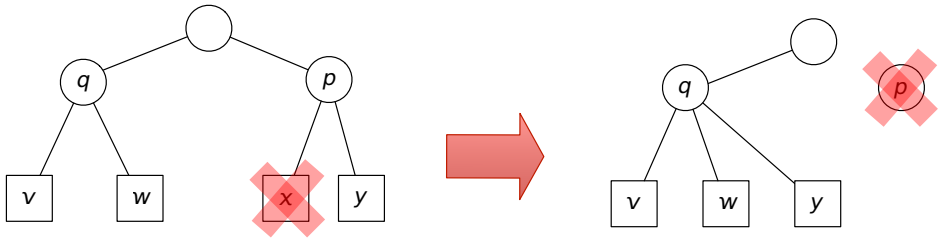


Figure 7: Deletion: giving away a child

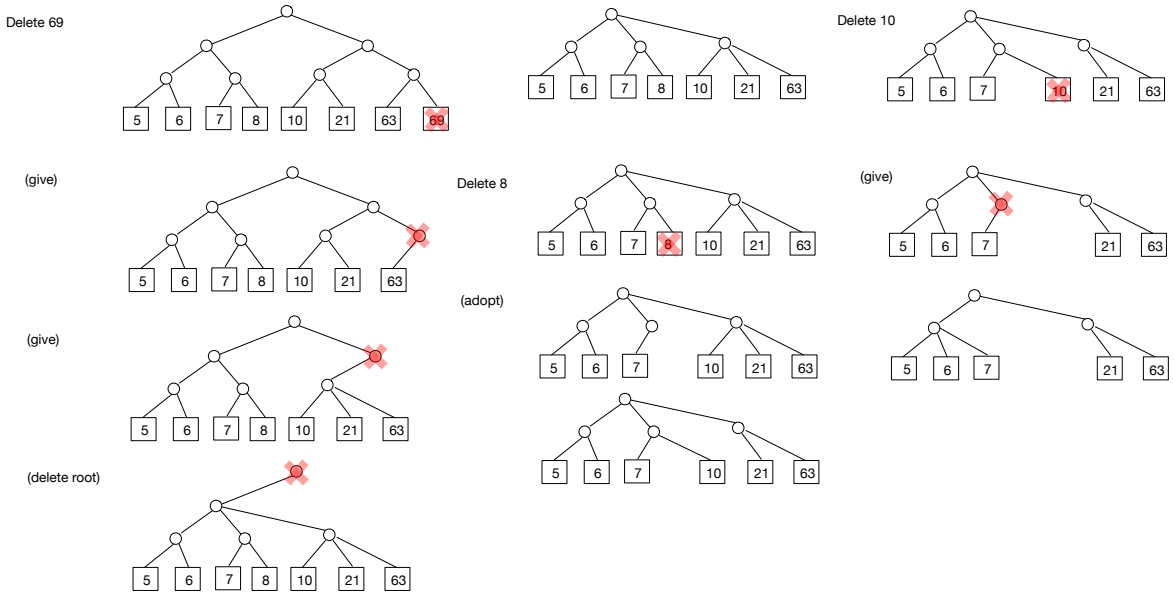


Figure 8: Deletion example

decrease.

Again, we have not discussed the extra steps needed to maintain the guides. However, the only guides that need updating are those along the path from p to the root. The total time needed to perform a deletion is proportional to the height of the tree, and so is $O(\log n)$, where n is the number of leaves in the tree.

See Figure 8 for an example of how a tree evolves during a sequence of deletions. Guides are not shown.

6 Traditional 2-3 trees and B-trees

Our exposition of 2-3 trees differs a bit from what is usually presented in the literature. In a "traditional" 2-3 tree, what we call "guides" are stored in the parent node, rather than the node itself. That is, each node has either one or two guides: the first guide is the largest

key that appears in its first subtree, and if there are three children, the second guide is the largest key that appears in its second subtree.

Here is one possible way in which we could lay out the data in such a traditional 2-3 tree. Again, we use *Java*-like syntax:

```
class Node { }
class InternalNode extends Node {
    KeyType guide0, guide1;
    Node child0, child1, child2;
}
class LeafNode extends Node {
    KeyType key;
    ValueType value;
}
```

Here is the revised logic of the search procedure, using this data layout:

```
Search(x):
p ← root of tree,  h ← height of tree
repeat h times
    // p points to an internal node
    if  x ≤ p.guide0  then  p ← p.child0
    else if  p.child2 = null or x ≤ p.guide1  then  p ← p.child1
    else  p ← p.child2

// p now points to a leaf node
if  x = p.key  then
    return p.value
else
    return null
```

Traditional 2-3 trees can be generalized to "B-trees". In a B-tree, each internal node contains between B and $2B$ guides. The number of children of a node is always equal to the number of guides plus 1. Thus, a traditional 2-3 tree is a B-tree with $B = 1$.

It is not hard to generalize Claim 1: *the height of any B-tree with n leaves is at most* $\log_{B+1}(n)$.

For example, suppose $B = 2^{10}$. Then a B-tree with at most billion leaves, so $n \leq 2^{30}$, has height at most 3:

$$\log_{B+1}(n) = \frac{\log_2 n}{\log_2(B+1)} \leq \frac{30}{10} = 3.$$

Compare this to the height of a 2-3 tree, or any binary search tree: the height of such a tree must be roughly 30. Of course, from the point of view of the number of comparisons performed in a search, the B-tree is no better: in the B-tree, it takes many comparisons to determine which subtree to explore (if the guides are sorted, this can be done using binary

search). However, from the point of view of memory access, the B-tree can be significantly better. This is especially true in storage systems, like a hard disk drive, which have high latency and high bandwidth. Let us assume that all the data stored in an internal node is stored in contiguous memory locations on a hard disk drive. After a request is made to read the data stored in a node, it may take quite a while to wait for the first byte of data to reach the CPU (high latency), but once it does, the remaining bytes of data will arrive at the CPU rather quickly (high bandwidth). So much better performance is obtained by reading a small number of large nodes rather than a large number of small nodes.

Because of their performance characteristics, B-trees are often used to organize the metadata in file systems. For example, Apple's HFS, HFS+, and APFS file systems all use B-trees.

7 Augmenting 2-3 trees

Many basic data structures can be augmented with additional data so as to support additional functionality. We give one example of how a 2-3 tree can be augmented.

Suppose that at each node, in addition to a "guide" field, we also maintain a "count" field, which keeps track of how many leaves are in the subtree rooted at that node.

Using this count field, we can efficiently implement queries of the following types:

- What is the kth smallest key?
- How many keys are $\leq x$?

For example, to find the kth smallest key, where $k \leq n$, we can do the following:

```
Find(k):   // find kth smallest key
p ← root of tree
h ← height of tree
repeat h times
    // p points to an internal node
    if  k ≤ p.child0.count  then
        p ← p.child0
    else if  p.child2 = null or k ≤ p.child0.count + p.child1.count  then
        p ← p.child1,  k ← k − p.child0.count
    else
        p ← p.child2,  k ← k − p.child0.count − p.child1.count

// p now points to a leaf node
return p.guide
```

We leave it to the reader to design the logic to determine how many keys are $\leq x$.

Of course, to maintain these count fields, one also has to modify the logic for the insertion and deletion routines. We also leave the details of this to the reader.

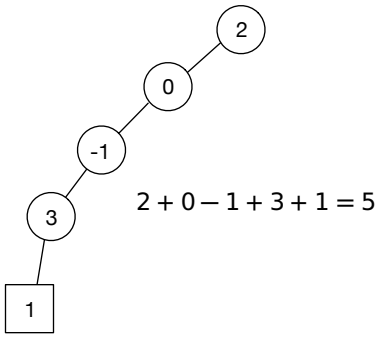


Figure 9: The "effective" value associated with a key

In summary, using a count field, we can implement the usual search, insertion, and deletion operations, as well as the operations of finding the kth smallest key, and determining how many keys are $\leq x$, so that each operation takes time $O(\log n)$, where n is the number of leaves in the tree.

8 Fast range operations and lazy updates

Suppose the values stored in a 2-3 tree are numbers. Moreover, suppose that in addition to the usual dictionary operations, we want to support the following operation:

- *AddRange*(x, y, Δ): add Δ to the value associated with each key in the range $[x, y]$.

An obvious implementation takes time $O(n_{x,y} + \log n)$, where $n_{x,y}$ is the number of keys in the range $[x, y]$. In the worst case, this is time $O(n)$, where n is the total number of keys in the tree. Perhaps surprisingly, we can actually implement this operation in time $O(\log n)$ using a kind of "lazy" update technique. This "lazy" update technique can be applied in a variety of other settings.

The idea is as follows:

- We store a value field at *every* node: both internal nodes and leaves.
- The "effective" value associated with a key is the sum all value fields on path from root to its leaf.

See Figure 9 for an example. In this figure, we have drawn the path from the root to some leaf, labeling each node with a value field. We do not show the guides in this figure. The effective value associated with the key stored at the leaf is the sum of all the value fields along this path. So we never explicitly store the "real" or "effective" value — it is represented implicitly.

To compute the effective value associated with a given key, we can easily modify the search algorithm so that we add up all of the value fields on the search path. The insertion and deletion routines can also be modified to maintain these value fields. We leave the details of that to the reader.

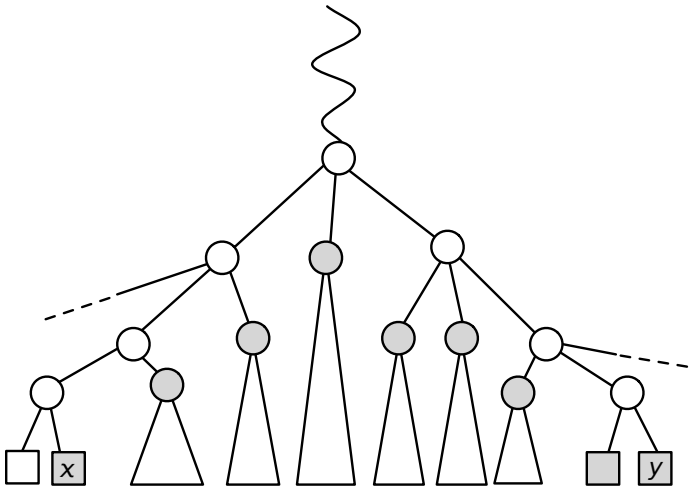


Figure 10: Implementing $AddRange(x, y, \Delta)$

We now describe how we can implement the $AddRange(x, y, \Delta)$ operation so that it takes time $O(\log n)$. Roughly speaking, we first perform a search for x and a search for y. Suppose the search path for x is e, and the search path for y is f. Both of these paths e and f start at the root, but eventually split. See Figure 10. We then add Δ to the value fields of all the nodes in Figure 10 that are shaded gray. Generally, these are the children of the nodes along the search paths that lie to the right of the path e and to the left of the path f. Now consider an arbitrary leaf in the tree. If the key stored in that leaf is in the range $[x, y]$, then there is exactly one gray node along the path from the root to the leaf; otherwise, there are no gray nodes on this path. It follows that the effective value associated with this key gets incremented by Δ iff the key is in the range $[x, y]$.

It is not hard to see that this implementation of $AddRange(x, y, \Delta)$ runs in time $O(\log n)$, since the number of gray nodes that need updating is $O(h)$, where h is the height of the tree. Moreover, the standard dictionary operations still run in time $O(\log n)$ as well.

A detailed example. Consider the 2-3 tree in Figure 11(a). The keys are the letters a–s, and just below each leaf we have indicated the key associated with that leaf. Each leaf is labeled with the value field stored at that leaf. In this particular tree, all of the internal nodes happen to have a value field of zero, and they are unlabeled in the figure. This will in fact be the state of any 2-3 tree before we perform any *AddRange* operations. Now suppose we do a search for the key e. The search path is highlighted in light green. The effective value associated with the key e is $0 + 0 + 0 + 0 + 4 = 4$. Here, we are adding up all of the value fields along the search path.

Figure 11(b) shows what happens when we perform the operation $AddRange(b, n, 1)$. The search paths for b and n are highlighted in light blue. These two paths already diverge at the root of the tree, but in general, they might diverge at some node below the root. The nodes shaded gray are the nodes whose value fields get incremented. So now we see that some internal nodes have nonzero value fields. As we perform more *AddRange* operations,

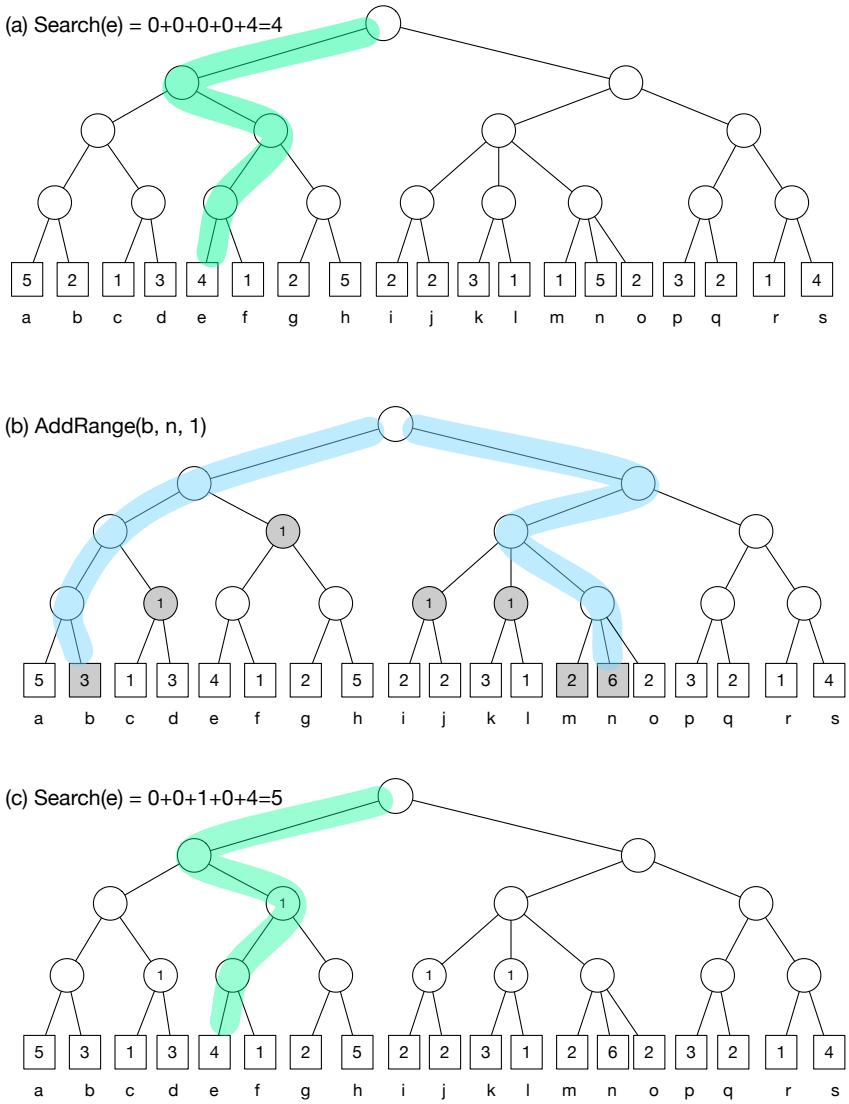


Figure 11: A detailed example of *AddRange*

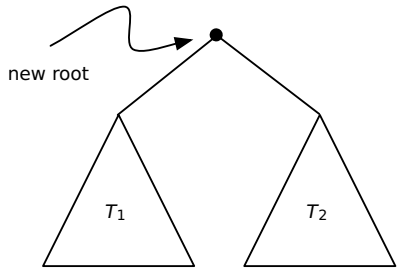


Figure 12: Join, Case 1

more internal nodes will have nonzero value fields.

Figure 11(c) shows what happens when we search again for the key e, after performing the operation $AddRange(b, n, 1)$. Again, the search path is highlighted in light green. The effective value associated with the key e is now $0+0+1+0+4=5$. Again, we are adding up all of the value fields along the search path, but now, one of the internal nodes has a value field of 1.

9 Join and split

Finally, we present two additional operations on 2-3 trees, *Join* and *Split*.

9.1 Operation Join

Operation $Join(T_1, T_2)$ joins two 2-3 trees, T_1 and T_2, assuming the largest key in T_1 is less than the smallest key in T_2. The original trees T_1 and T_2 are destroyed in the process.

Here is how we can implement $Join(T_1, T_2)$ in time $O(\log n)$, where n is the total number of leaves in T_1 and T_2. Assume T_i has height h_i for $i = 1, 2$.

Case 1: $h_1 = h_2$. In this case, we simply make the roots of T_1 and T_2 children of a new root. See Figure 12.

The time to perform the *Join* operation in this case is $O(1)$.

Case 2: $h_1 < h_2$. Let v be the root of T_1. Let p the left-most node in T_2 at depth $h_2 - h_1 - 1$: we can find p by starting at the root of T_2 following $h_2 - h_1 - 1$ left-most edges. We "implant" T_1 into T_2 by making v a left-most child of p. This makes v a node of depth $h_2 - h_1$ in the new tree, and so all of the leaves in the new tree are at depth h_2. See Figure 13. The result is will be a 2-3 tree, unless p now ends up with four children. If that happens, we split p just as in the *Insert* routine, proceeding up the path from p to the root of T_2, splitting nodes as necessary.

The time to perform the *Join* operation in this case is $O(h_2 - h_1 + 1)$.

Case 3: $h_1 > h_2$. This cases is the mirror image of Case 2. In this case, we "implant" T_2 into T_1. The time to perform the *Join* operation in this case is $O(h_1 - h_2 + 1)$.

In the above exposition, we have not discussed the bookkeeping necessary to to maintain the guide values. We leave those details to the reader.

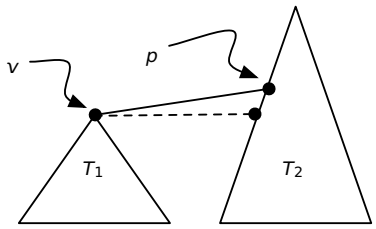


Figure 13: Join, Case 2

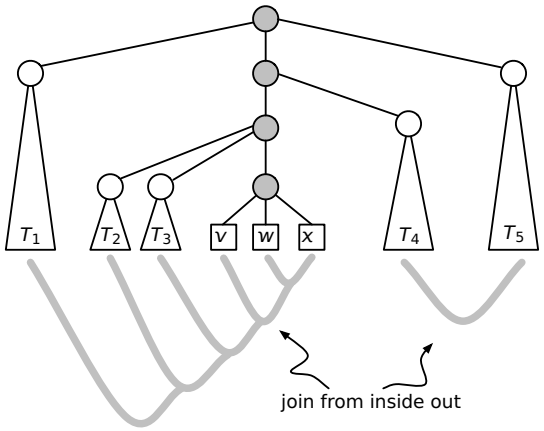


Figure 14: The $Split(T, x)$ operation

9.2 Operation Split

Operation $Split(T, x)$ takes a 2-3 tree T and a key x, and produces 2-3 trees T_1 and T_2, where T_1 contains all of the keys in T that are $\leq x$, and T_2 contains all of the keys in T that are $> x$. The original tree T gets destroyed in the process.

Here is how we can implement $Split(T, x)$ in time $O(\log n)$, where n is the number of leaves in T.

We begin by searching for x. We then delete all of the internal nodes on the search path. This results in a number of "free standing" trees. We then use a sequence of *Join* operation to merge the trees corresponding to keys $\leq x$ to build the tree T_1, and also to merge the trees corresponding to keys $> x$ to build the tree T_2. As depicted in Figure 14, these join operations are performed "from the inside out". This is necessary to ensure that the running time of all of these join operations is $O(\log n)$, rather than $O((\log n)^2)$, which is the bound that would follow naively from our analysis of the *Join* operation.

To analyze the running time of these merge operations, consider the following generalized setting. We want to merge 2-3 trees $X_1, \ldots, X_k$ of heights $h_1, \ldots, h_k$ where the following condition **C** holds:

(i) $h_i \leq h_{i+1}$ for $i = 1, \ldots, k-1$, and

(ii) there are at most 2 trees of any given height, except the first 3 may be of the same height.

The reader should verify that the two sequences of trees to be merged in the *Split* routine satisfy condition **C**.

We now analyze what happens when we merge these trees by "joining" them from left to right: we first join X_1 and X_2, obtaining Y_2; we then join Y_2 with X_3, obtaining Y_3; we then join Y_3 with X_4, and so on.

Claim 3. For $i = 2, \ldots, k$, the height of tree Y_i is either h_i of $h_i + 1$.

We prove this claim by induction on k. So we assume the claim is true for all sequences of trees of length less than k that satisfy condition **C**, and we prove that it holds for a given sequence $X_1, \ldots, X_k$ that satisfies condition **C**.

The claim is clearly true for $k \leq 2$, so we assume that $k > 2$.

Case 1: $h_1 \leq h_2 < h_3$. We have:

(a) Y_2 has height h_2 or $h_2 + 1$;

(b) Condition **C** holds for $Y_2, X_3, X_4, \ldots, X_k$.

Both of these properties are easily verified. The claim follows from the induction hypothesis applied to $Y_2, X_3, X_4, \ldots, X_k$.

Case 2: $h_1 \leq h_2 = h_3$. We have:

(a) if $k > 3$, then $h_3 < h_4$;

(b) Y_2 has height h_2 or $h_2 + 1$;

(c) Y_3 has height $h_2 + 1$;

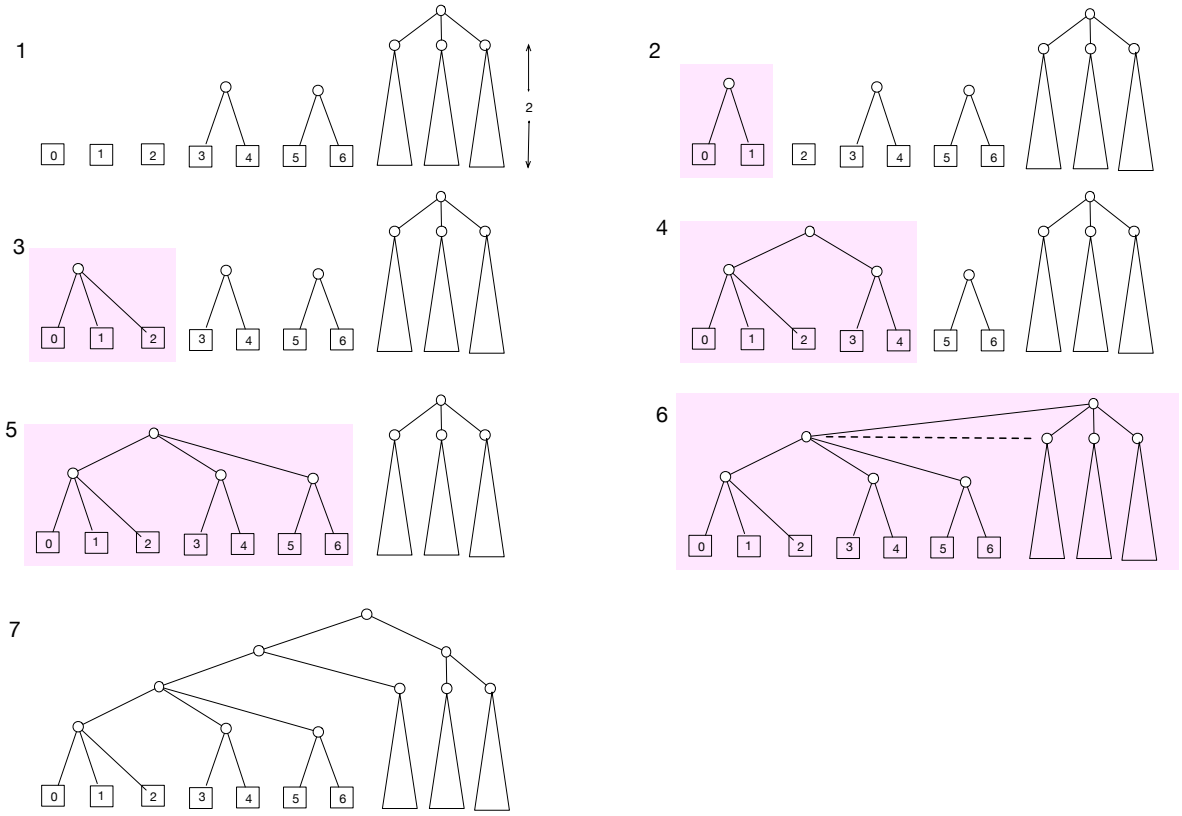
(d) Condition **C** holds for $Y_3, X_4, X_5, \ldots, X_k$.

Property (a) follows from condition **C**. Property (b) is clear. Property (d) easily follows from Properties (a) and (c).

It remains to prove Property (c). On the one hand, if Y_2 has height h_2, then the computation that builds Y_3 by joining Y_2 and X_3 (which have equal height) does so by adding a new root (as in Case 1 of the *Join* algorithm), and so Y_3 has height $h_2 + 1$. On the other hand, suppose Y_2 has height $h_2 + 1$. In this case, the root of Y_2 has only two children. This is because the root of Y_2 was freshly created when X_1 and X_2 were joined, and any freshly created root has only two children, regardless of which case of the *Join* algorithm was executed (the reader should examine the logic of the *Join* procedure to verify this). Therefore, the computation that builds Y_3 by joining Y_2 and X_3 does so by implanting X_3 into Y_2. Because the root of Y_2 only has two children, it will not need to be split, and so the resulting tree Y_3 also has height $h_2 + 1$.

The claim follows from the induction hypothesis applied to $Y_3, X_4, X_5, \ldots, X_k$.

Example. The following illustrates how the merge may proceed on a sequence of trees $X_1, X_2, X_3, X_4, X_5, X_6$ with heights $0, 0, 0, 1, 1, 3$.



We merge X_2 and X_2 to obtain Y_2 of height 1, and then we merge Y_2 and X_3 (implanting X_3 into Y_2) to obtain Y_3 also of height 1. These steps correspond to Case 2 above. Next, we merge Y_3 and X_4 to obtain Y_4 of height 2, and then we merge Y_4 and X_5 (implanting X_5 into Y_4) to obtain Y_5 also of height 2. These steps also correspond to Case 2 above. Finally, we merge Y_5 and X_6 (implanting Y_5 into X_6) to obtain Y_6 of height 4.

Based on Claim 3, the computation that builds Y_{i+1} by joining Y_i and X_{i+1} takes time $O(h_{i+1} - h_i + 1)$. Therefore, the overall time required to merge $X_1, \ldots, X_k$ is $O(t)$, where

$$\begin{aligned} t &\leq (h_2 - h_1 + 1) + (h_3 - h_2 + 1) + \cdots + (h_k - h_{k-1} + 1) \\ &= h_k - h_1 + k - 1. \end{aligned}$$

In the context of the *Split* operation, if h is the height of the original tree T, then $h_k \leq h$ and $k = O(h)$. It follows that the overall running time to merge all of the trees is $O(h)$, which is $O(\log n)$.