CS104: Data Structures and Object-Oriented Design (Fall 2013)
November 14, 2013: 2-3 Trees: Inserting and Deleting
Scribes: CS 104 Teaching Team

Lecture Summary

In this class, we investigated 2-3 Trees in more depth. We saw how to traverse them, search in them, and mostly, how to insert and delete.

1 2-3 Trees: The Basics

The beginning of these notes is just a recap from the previous lecture's notes. 2-3 Trees differ from Binary Search trees in that each node contains 1 or 2 keys. Unless it is a leaf, a node with 1 key has exactly 2 children, and a node with 2 keys has exactly 3 children. The defining properties of 2-3 Trees are the following:

- All leaves must be at the same level.
- For nodes with 1 key k, the binary search tree property holds: all keys in the left subtree are smaller than k, and all keys in the right subtree are greater than k.
- For nodes with 2 keys $k_1 < k_2$, the three subtrees T_1, T_2, T_3 satisfy the following property:
  1. All keys in T_1 are smaller than k_1.
  2. All keys in T_2 are strictly between k_1 and k_2.
  3. All keys in T_3 are greater than k_2.

It will be our job in building these trees to ensure that the defining properties hold at all times.

To implement a 2-3 Tree, we will need a slightly revised version of a node, to accommodate multiple keys and values. The class would look as follows[1]:

```
template <class KeyType, class ValueType>
class Node {   // variables could be public, or could add getters/setters
  int numberKeys;
  KeyType key[2];      // space for up to 2 keys
  ValueType value[2];  // space for the corresponding values
  Node* subtree[3];    // pointers to the up to 3 subtrees
}
```

The first useful observation about 2-3 Trees is that — like other search trees — they make it really easy to output all keys in sorted order. We simply traverse the tree using depth-first in-order traversal. That is, at any node, we first recursively visit the entire left subtree, then the key (or left key, if there are two), then the middle tree (if it exists), the right key (if it exists), and finally the right subtree. This is the natural generalization of in-order traversal from binary trees to 2-3 Trees. If you do not remember how to traverse a tree, check back to the lecture notes from Lecture 17.

This describes how to implement an ordered iterator over search trees. This is one of the main advantages of search trees over Hashtables as data structures for Dictionaries. For Hashtables, it's much harder to implement an efficient Iterator. (Hashtables have other advantages to compensate.)

In implementing the operations on 2-3 Trees (in particular, insertions and deletions), we should keep in mind the following:

[1] This is slightly different from what I showed in lecture, where I used variables `leftKey` and `rightKey`. When you put them in an array, some of the operations we see later are actually easier to implement, hence the change.

- Our 2-3 Tree properties must always hold, so that the tree remains balanced. Thus, we must ensure that two-key nodes have three children, one-key nodes have two children, and we can't have 3 keys in one node.
- All insertions and removals in search trees typically (and definitely for 2-3 Trees) happen at a leaf; that's much easier to deal with than changing things at internal nodes.
- Typically, pre-processing (for removals) and/or post-processing (to restore various properties) is required, in addition to just adding or removing keys.

Internally, the best way to store a 2-3 Tree is probably with a dynamic memory structure and pointers, storing a pointer to the root somewhere. Initially, when the tree is empty, we can set the root pointer to `NULL`. The tree will grow both in number of nodes and height as insertions happen, so obviously, we do not want to fix its size ahead of time.

We could also try to store the tree in an array (like we did for Priority Queues), but lose all of the advantages of doing so, as index calculations are not possible any more. When the tree was complete and binary, it was very easy to figure out which were the children of a given node, but here, it would depend on the degrees of many other nodes, which would take way too long to look up.

Next, we will explore how the different operations (insertions and removals) work on trees. Our running example will be the tree T shown in Figure 1:

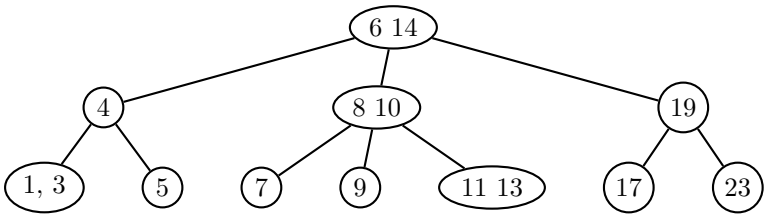

Figure 1: Our running example T of a 2-3 Tree.

2 Insertions

Whenever we insert a key (or, more precisely, a key-value pair) into a tree, the first thing we do is search for it, unsuccessfully, of course. We then insert it into the leaf node where the search terminated. If the leaf node only had one key before, it has two keys now, and everything is good. But if the leaf node already had two keys, then it now has three keys, so we'll need to fix it. We will see how to fix it by looking at the effects of inserting a number of keys into our example tree T.

First, let's look at the case of inserting 21 into T. To do so, we search for 21. When we do that, we will terminate at the node that already contains the key 23. This leaf node has only one key, so we can simply add 21. The result will only be a local change, and our tree now looks as in Figure 2:

This was easy. Of course, we won't always be so lucky as to insert into a node with only a single key. So next, let's look at what happens when we insert 26 into the tree we got after inserting 21. The key 26 also belongs in the node with 21 and 23, so when we add it to the node, that leaf now contains three keys, as shown in Figure 3:

To fix this, we can break up a node with 3 keys, by moving its middle key up to the parent node, and making the two other keys into separate nodes with one key each. In our example, 23 gets moved into the node that currently contains only the key 19, turning it into a node containing 19 and 23. The other two elements, 21 and 26, are split into two separate nodes (since the parent node is now a two-key node, it can

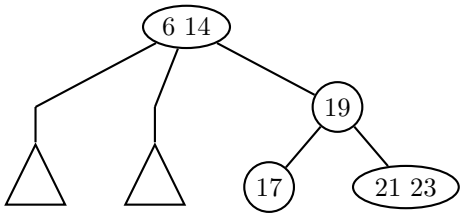

Figure 2: T after inserting the key 21. In this and subsequent figures, we'll just write triangles to denote parts that are unchanged or not particularly interesting.

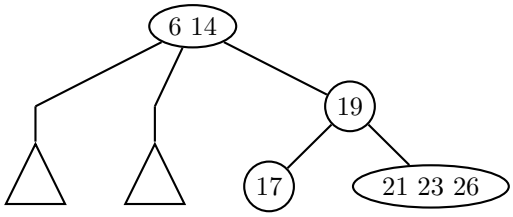

Figure 3: After inserting the key 26. Having three keys in a node is illegal, so we need to fix this.

support three children), and these nodes become the children nodes for the node containing 19 and 23. The resulting tree looks as shown in Figure 4.

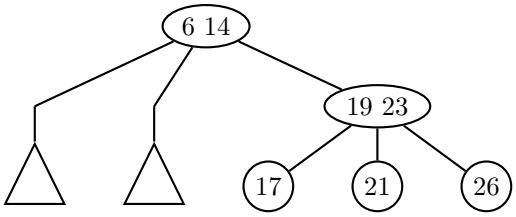

Figure 4: After fixing the problem from inserting the key 26.

So in this case, with just one split operation, the problem was fixed. Next, let's see what happens when we insert 12 into T. (So we're thinking of inserting it into the *original* tree, but it makes no difference in this particular case if you want to think of inserting it into the tree after we already inserted 21 and 26.)

Again, we first search for 12, ending up in the leaf that contains the keys 11 and 13 already. Again, we add it to the leaf node, resulting in 3 keys in that node. The resulting situation is shown in Figure 5.

As before, we take the middle key — here, 12 — and move it up to the parent's node, splitting the keys 11 and 13 into separate new nodes. But now, the parent has 3 keys, and 4 subtrees, as shown in Figure 6. So we'll have to fix that.

To fix this new problem of a node with 3 keys, we again take the middle key — in this case, 10 — and move it up one level, into the parent node with keys 6 and 14. Now, the keys 8 and 12 get split and become their own nodes; the new node containing 8 obtains the left two subtrees (here, nodes with keys 7 and 9),

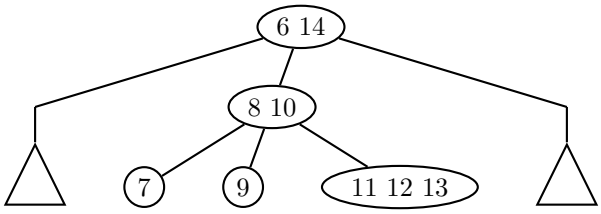

Figure 5: The result of inserting the key 12 into T.

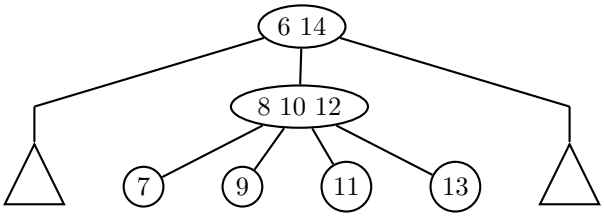

Figure 6: Moving the key 12 one level up.

while the node with key 12 obtains the right two subtrees, with keys 11 and 13. The resulting tree is shown in Figure 7.

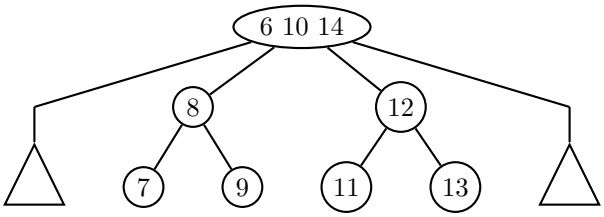

Figure 7: Moving the key 10 one level up.

Now, we have moved the problem yet another level up. The root node now contains three keys 6, 10, and 14. So we do the same thing: we push up the middle key (here, 10) one layer, and split the other two keys into separate nodes, each inheriting two of the subtrees. The result is shown in Figure 8

At this point, we finally have obtained a legal 2-3 tree again, and the process terminates. Notice that we have now increased the height of the tree by one. Some students were wondering in class how the tree can ever grow in height if we always insert at leaves. Now we know. It grows at the root, even though insertion happens at the leaves.

The important thing to keep in mind is that whenever a node has too many keys, you move up the *middle* key to the parent node, and split the remaining two keys into two separate new nodes. The left node inherits the two left subtrees, while the right node inherits the two right subtrees.

Below is pseudocode to help you understand how to insert into 2-3 Trees in general.

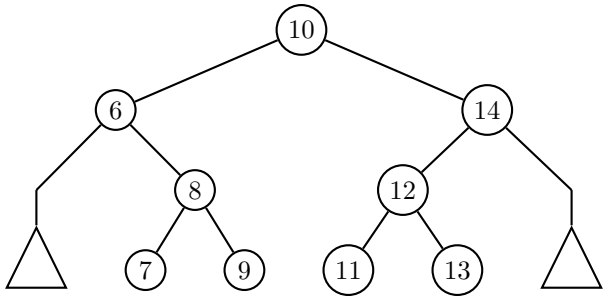

Figure 8: Moving the key 10 one more level up.

```
Inserting a new key

1. Search for the key. If found, deal with duplicate.
   Otherwise, we have found the leaf where the key belongs.
2. Insert the key into the leaf.
      If the leaf only had one key before, we are done now.
3. Otherwise, we must now fix the leaf.

To fix a node with 3 keys (a b c):
1. Move b up into the parent node if there is one.
   (Otherwise, create a new root for the tree, containing just b.)
2. Create a node with just a as the key, and make it the left child of the b node.
3. Create a node with just c as the key, and make it the right child of the b node.
```

Notice that insertion as we defined it maintains the properties of a 2-3 Tree: the number of levels on all subtrees is increased by one simultaneously when creating a new root, and not increased for any node otherwise.

Let's try to analyze the running time. To find the right leaf node takes $\Theta(h)$, where h is the height of the tree. To actually insert the key into the tree takes $\Theta(1)$. Breaking any three-key node into new smaller nodes takes $\Theta(1)$, and we have to do this at most once for each level of the tree, so it takes $\Theta(h)$.

So next, we want to calculate the height of a 2-3 Tree. We do this the way we have calculated tree heights before. If the tree has height h, because all leaves are at the same level, and each internal node has at least 2 children, it has at least as many nodes as a complete binary tree of h levels (plus one more), giving us an upper bound of $\log_2(n)$ on the height, as we calculated before in the context of heaps. On the other hand, a tree of h levels with maximum degree 3 has at most 1 node on level 0, 3 nodes on level 1, 9 nodes on level 2, and generally 3^k nodes on level k, for a total of at most $\sum_{k=0}^{h-1} 3^k = \frac{3^h-1}{3-1} \leq \frac{3^h}{2}$ nodes. Thus, $n \leq 3^h/2$, giving us that $h \geq \log_3(2n) \geq \log_3(n)$. Thus, h is between $\log_3(n)$ and $\log_2(n)$, and those two are within small constant factors of each other. Thus, the runtime for insertion is $\Theta(\log(n))$. (We can ignore the base of the logarithm, as they are off only by constant factors.)

3 Removals

To understand how removing a key from a 2-3 Tree works, let's return to our original tree T, but also add another key 16 to it. The resulting tree is shown in Figure 9.

Now, let's see what happens when we want to remove the key 14. As we mentioned earlier, removing keys from internal nodes is a glorious mess, since we wouldn't have the right number of keys for the corresponding

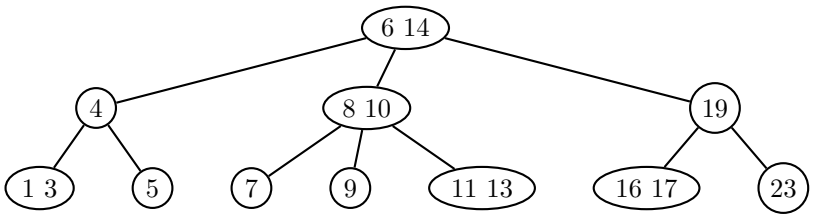

Figure 9: The tree T with the key 16 added.

number of subtrees any more. Thus, removals should always happen at leaves. In order to accomplish this, we need to first do some swapping.

The swap will always be with the immediate predecessor (next smallest key in the tree) or immediate successor (next largest key in the tree). For concreteness in this class, we will use the immediate successor. To find it, take the next edge right from the key to delete, and then the leftmost edge for all subsequent levels. So the pre-processing step is to find the successor whenever the key is not already in a leaf, and swap the key with the successor key. The result of swapping 14 (key to delete) with its successor (16) is shown in Figure 10.

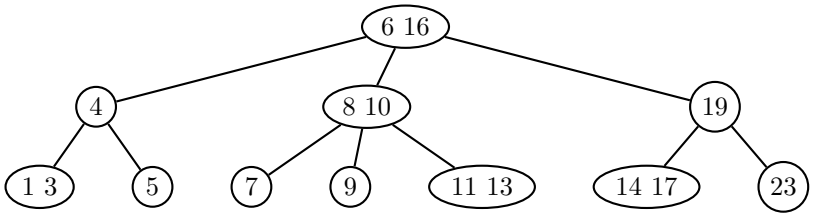

Figure 10: After 16 and 14 have been swapped.

Now, the key can be safely deleted from the leaf. Notice that this maintains the search tree property, as the successor was safe to take the place of the key we want to delete. The result of deleting 14 now is shown in Figure 11.

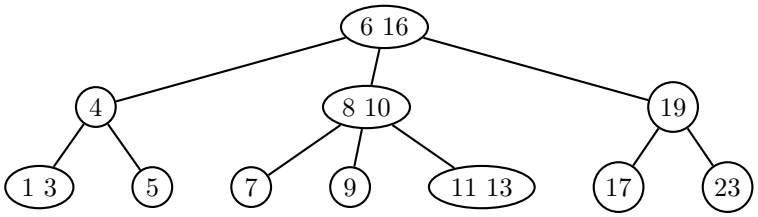

Figure 11: After the deletion of the key 14.

Deleting the key 14 was pretty easy: after the swap, its leaf node contained two keys, which made it safe to just delete a key. Next, let's see what happens if we remove the key 5. That key is in a node all by itself,

so after we remove it, we have an empty leaf node. This situation is shown in Figure 12.

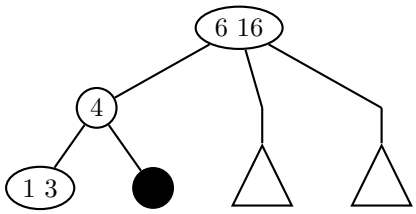

Figure 12: After the deletion of the key 5. The now-empty node is shown in black.

Here, the empty node has a sibling with two keys 1 and 3. In this case, we can move the key from the parent down to fill the node, and move one of the keys from the sibling up to the parent. The result is the tree in Figure 13:

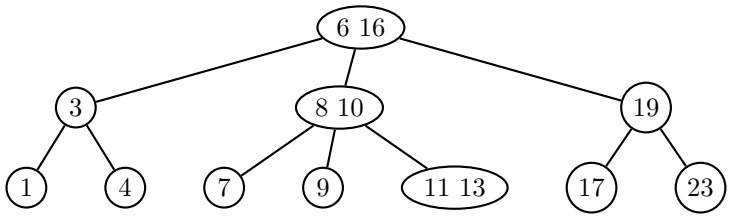

Figure 13: Borrowing a key from the sibling via the parent.

Next, let's see how borrowing from a sibling works when a node has two siblings, and the one with two keys is the furthest. To see this, let's look at what happens if we next remove the key 7. The intermediate stage will be the following:

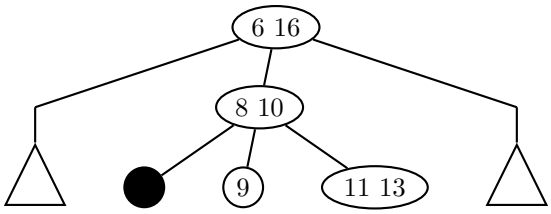

Figure 14: Deleting the key 7. The now-empty node is shown in black.

This time, borrowing a key from the sibling still works, but takes a few more intermediate steps. The key 11 from the sibling gets moved up to the parent node; the key 10 from the parent gets moved down to the other sibling, while the key 9 from the other sibling gets moved up to the parent. Finally, we can move the key 8 down to the empty node. The result is only locally rearranged, and shown in Figure 15.

Next, let's see what happens when there's no sibling to borrow from. To see this, we delete the key 8 next. The resulting tree is shown in Figure 16.

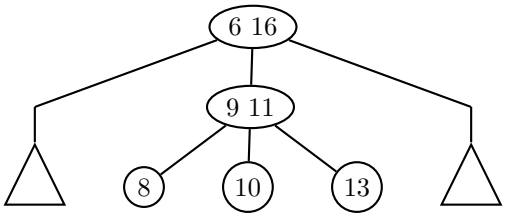

Figure 15: Borrowing from a 2-step removed sibling via the parent. This requires some moving up and down of keys, but only between the siblings and the parent.

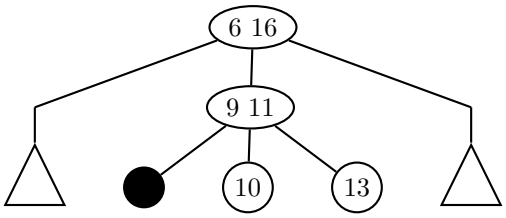

Figure 16: Deleting the key 8. The now-empty node is shown in black.

Now, we cannot borrow from a sibling directly. But here, fortunately, the empty node has a parent with two keys, so it's possible to borrow a key from the parent, and another one from the immediately neighboring sibling. The resulting tree is shown in Figure 17.

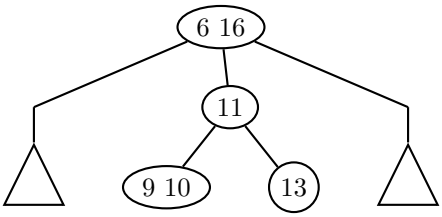

Figure 17: Borrowing a key from the parent.

So far, we have been able to handle every deletion with just one constant-time fix. But you will have noticed that there is one case we have not yet considered: that the node's parent has one key (so one cannot borrow from the parent), and its only sibling has only one key (so one cannot borrow from the sibling).

To explore this case, let's look at the effect of deleting the key 17 from the tree next. The first step of deletion is shown in Figure 18.

Now, there is no easy way to fix the problem at this level. So we need to escalate the problem one level up. To do so, we combine the one key in the parent node with the one key in the sibling node to form a node with two keys, and make the parent empty. In other words, we produce the tree shown in Figure 19:

At the next higher level, we now have a parent with two keys, so we can borrow from the parent to fix the problem, giving us the tree shown in Figure 20.

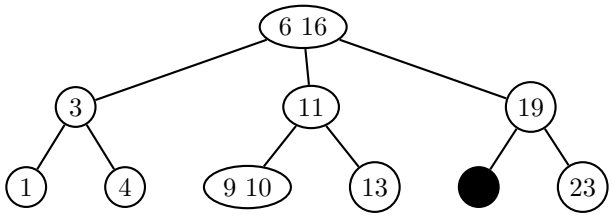

Figure 18: Deleting the key 17. The now-empty node is shown in black.

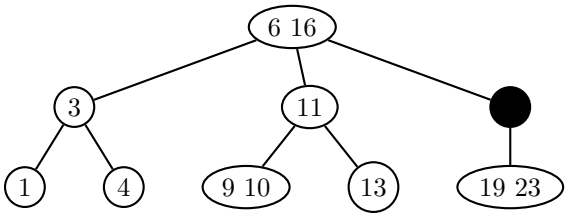

Figure 19: Combining a parent key with a sibling key, and passing the problem of the empty node up one level.

Of course, we were a bit lucky — if the parent had had only one key, we may have had to pass the problem up yet another level, possibly all the way up to the root. Once the root is empty, though, it is safe to simply delete it, thereby reducing the height of the tree by 1.

In summary, we get the following cases for deletion of a key:

- If the leaf containing the key had 2 keys, we can just remove it and are done.
- Otherwise, we will now describe different cases for dealing with a node (leaf or internal) that has 0 keys left, and possibly one subtree hanging off:
  1. If the node has a sibling with 2 keys, then rearrange one key from the sibling and keys from the parent to fix the problem completely.
  2. If all siblings have only one key, but the parent has 2 keys, then use a key from the parent and one from the sibling to fix the problem completely.

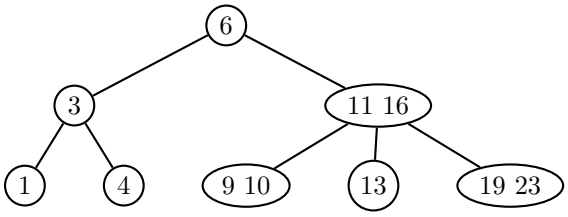

Figure 20: Borrowing from the parent to resolve the empty node in the middle of the tree.

3. If the parent has only one key and the only sibling also has only one key, then merge the parent's key and the sibling's key, and shift the empty node up one level. Recurse (or iterate) to solve the problem there.

We will see these removal rules again in a little more detail in the next lecture (and its notes).