2-3 Trees

Robert Guderian

Comp 2140 Summer 2011
A 2-3 tree is not a binary tree: some nodes can have three children.

A 2-3 tree is useful for implementing tables because the trees are always short.

A 2-3 tree containing $n$ keys is guaranteed to have height $O(\log n)$, and its search, insert and delete algorithms all take $O(\log n)$ time.
A 2-3 tree of height $h$

- Is the empty tree, if $h = 0$, or
- If $h > 0$, the tree has one of two forms:

A root with two children:

- A 2-3 tree of height $h - 1$
- A 2-3 tree of height $h - 1$
A root with three children:

- 2-3 tree of height $h - 1$
- 2-3 tree of height $h - 1$
- 2-3 tree of height $h - 1$
The data are stored in the leaves of the tree — they are in sorted order if you look at the leaves from left to right.

The internal nodes of the tree are like the index in a book.

The internal nodes contain index keys to guide your searches to the leaf containing the data you want.
A node \( v \) with two children \( T_L \) and \( T_R \):

Node \( v \) contains only one key index \( k_1 \). Furthermore,
- the data and index keys in \( T_L \) are less than \( k_1 \), and
- the data and index keys in \( T_R \) are greater than or equal to \( k_1 \).
A node \( v \) with three children \( T_L, T_M \) and \( T_R \):

\[
\begin{align*}
\text{Node } v & \quad k_1, k_2 \\
< k_1 & \quad T_L \\
\geq k_1 < k_2 & \quad T_M \\
\geq k_2 & \quad T_R
\end{align*}
\]

Node \( v \) contains two values \( k_1 \) and \( k_2 \) (which are sorted: \( k_1 < k_2 \)). Furthermore,

- the data and index keys in \( T_L \) are less than \( k_1 \),
- the data and index keys in \( T_R \) are greater than or equal to \( k_2 \), and
- the data and index keys in \( T_M \) are greater than or equal to \( k_1 \) and less than \( k_2 \).
An Example 2-3 Tree

- 31
  - 15, 23
    - 9
    - 20
    - 27, 29
      - 2
      - 9
      - 15
      - 20
      - 23
      - 27
      - 29
    - 40, 47
      - 31
      - 40
      - 47
      - 60
      - 67
Searching in a 2-3 tree is very similar to searching in a binary tree, except that

- We may have to examine two index keys in a node to decide which child to move to next, and
- We always search down to a leaf — the interior nodes do not contain the data, so we keep going down to the leaves.
- Instead of using the binary search tree property to decide to which child to move, we use the rules for placing index keys in interior nodes to decide to which child to move.
A 2-3 tree with height $h$ will have
- At least $2^h - 1$ nodes (the number of nodes in a full binary tree of height $h$), and
- At most $(3^h - 1)/2$ nodes (the number of nodes of a full ternary tree of height $h$).

A 2-3 tree with $n$ nodes will therefore have height
- At least $\log_3(2n + 1)$, and
- At most $\log_2(n + 1)$. 
When we insert a new key $k$, we will insert a new leaf that contains $k$.

Which interior node should be the parent of the new leaf?

We search for the key $k$, and remember the last interior node we see on the search (on the level above the leaves).

That interior node should be the parent of the new leaf.
Inserting 18 causes the following search:

```
Start at the root.

15, 23

9

20

27, 29

2
9
15
20
23
27
29
```
Inserting 18 causes the following search:

\[ 15 < 18 \leq 23 \]

Go to the middle child.
Inserting 18 causes the following search:
Inserting 18 causes the following search:

18 < 20

15,23

Go to the left child.
Inserting 18 causes the following search:

Since we’re at the leaf level, the node we were at previously (parent 20) is the parent of the new node.
This case is a simple case: the chosen parent has only two children, and therefore has room for one more.

The leaves are kept in order, so 18 should become the middle child of its parent.
Now the parent has three children — it needs to contain two index keys $k_1$ and $k_2$, where $k_1 < k_2$.

The values in the parent’s middle and right children become $k_1$ and $k_2$, respectively.
Insert: Placing the New Node (Simple Case)
Insert 28. First, search to the leaves:

We end at leaf 27, so its parent (27,29) is the parent of the new leaf.
Insert 28. The parent of the new leaf already has three children:
To resolve the problem of a parent with three children and too full to accept another child:

- Split the four children into the two smaller children and the two larger children.
- The two smaller children stay with the parent, which becomes a node with just two children.
- Create a new node to be the parent of the two larger children.
Insert 28. Split the four children into two pairs:

The old parent should have just one index key, and the new parent needs an index key and to be connected to its parent.
At the level above the leaves, we can get the correct index keys by simply looking at the data values in the leaves.

The three largest values (27, 28 and 29) in the four children become index key values in the parents and the parent of the parents (the grandparent).

The original parent should have index key 27 (the value of its larger child) and the new parent should have index key 29, the value of its larger child.

We also pass the middle value (28) of the three values that become index keys up to the grandparent to be the index key separating the old parent from the new parent.
Insert 28. Set up index keys correctly:

Now the parent of the parents (the grandparent) is too full!
We have to split the grandparent in exactly the same way:

What index keys should be in the old grandparent and the new grandparent? Because the old grandparent was the root, there’s no parent to which to attach them. How can we attach the new grandparent into the tree?
Insert: A More Complex Case

- When we’re splitting an interior node that is not the parent of leaves, we will already have three index keys (one of which was passed up from a split at a child) that properly separate the four children.
- The minimum of those three index keys becomes the old parent’s index key.
- The maximum of those three index keys becomes the new parent’s index key.
- The middle of those three index keys goes up to the grandparent above to separate the old and new parents.

\[
\begin{align*}
T_1 & \quad T_2 & \quad T_3 & \quad T_4 \\
k_{\text{min}}, k_{\text{mid}}, k_{\text{max}} & \quad \Rightarrow \quad & k_{\text{min}} & \quad k_{\text{mid}} & \quad k_{\text{max}} \\
T_1 & \quad T_2 & \quad T_3 & \quad T_4 \\
k_{\text{min}} & \quad k_{\text{mid}} & \quad k_{\text{max}}
\end{align*}
\]
When the old parent is the root, there’s no grandparent to which to attach the new parent.  
We create a new grandparent, which becomes the new root.
The final result after creating a new root:
If we insert 34 into an empty tree, we get a tree with a leaf root.
Now insert 68. A leaf can have only one data value, so we split and make a new root:

\[ 34,68 \] becomes \[ 34 \] and \[ 68 \]
Now insert 1. There’s room in the parent for another child, so this is an easy case:
Next insert 2. There’s no room in the parent for another child, so we have to split and create a new root:
Next insert 88. There’s room in the parent for another child, so we have an easy case:
Next insert 56. There’s no room in the parent for another child, so we have to split the parent:

```
1
2
34
2
56, 68, 88
```
Next insert 56. There’s no room in the parent for another child, so we have to split the parent. Fortunately, there’s room in the grandparent for the new parent:
Next insert 94. There’s room in the parent for another child, so we have an easy case:
Next insert 72. There's no room in the parent for another child, so we have to split the parent:
Next insert 72. There’s no room in the parent for another child, so we have to split the parent. Unfortunately, there’s no room in the grandparent (the root), so we have to split the grandparent and create a new root:
Next insert 72. There’s no room in the parent for another child, so we have to split the parent. Unfortunately, there’s no room in the grandparent (the root), so we have to split the grandparent and create a new root:
Next insert 45. There’s room in the parent for another child, so we have an easy case:
Finally insert 39. There’s no room in the parent for another child, so we have to split the parent:
Finally insert 39. There’s no room in the parent for another child, so we have to split the parent. Fortunately, there’s room in the grandparent for the new parent:
Insertions are most easily performed recursively.

- We recursively search down the tree — one recursive call for each level of interior nodes in the tree.
- Then as we return back up the tree to the root, we perform whatever splits may be necessary.
Because data is stored only in the leaves, we must search to the leaf level for the data to be removed.

We therefore want to delete a leaf.
Delete 68. The parent of leaf 68 has three children, so it can lose one child and still be a valid node:

The parent will have only two children and should therefore only have one index key. What should the index key be?
Deletion: Easy Case

- The parent of two leaves should contain as its index key the key from the larger child.
- When one of three leaf children is deleted, the value from the largest remaining child becomes the parent’s index key.
The parent of deleted leaf 68 had three children, so it remains a valid node:
Delete 56:

When we delete 56, its parent will have only one child, which is not allowed.
After we delete 56, its parent has only one child:

The parent has a sibling next to it that has three children — we will “adopt” an extra child from the sibling with three children.
After we delete 56, we adopt the closest child from a neighbouring sibling with three children:

The index keys in the parent, the sibling, and their parent have to be adjusted somehow.
Here’s how the index keys get changed, if the sibling is to the right and $k_g$ is the key in the grandparent separating the two siblings:

$$
\begin{array}{c}
\begin{array}{c}
\text{Deletion: Adopting from a Sibling} \\
\text{Here’s how the index keys get changed, if the sibling is to the right} \\
\text{and } k_g \text{ is the key in the grandparent separating the two siblings:}
\end{array}
\end{array}
$$
After we delete 56, we adopt the closest child from a neighbouring sibling with three children, and we adjust the index keys as follows:
Delete 68:

When we delete 68, its parent will have only one child, which is not allowed.
After deleting 68, its parent has only one child, but it can adopt the nearest child from the sibling to the left, which has three children:

When the parent with only one child adopts its left sibling’s nearest child, we will have to adjust index keys in the parent, the sibling, and the grandparent.
Deletion: Adopting from a Sibling

Here’s how the index keys get changed, if the sibling is to the left and \( k_g \) is the key in the grandparent separating the two siblings:

\[
\begin{align*}
\text{A} & \quad \text{B} & \quad \text{C} & \quad \text{D} \\
\text{ki}, \text{kr} & \quad \text{kd} \\
\end{align*}
\]

becomes

\[
\begin{align*}
\text{A} & \quad \text{B} & \quad \text{C} & \quad \text{D} \\
\text{ki} & \quad \text{kg} \\
\end{align*}
\]
After deleting 68 and the parent adopts the nearest child from the sibling to the left, the index keys are as follows:

```
1
2
34
56
88
94
```

```
2
56
94
```

```
34, 88
```
Delete 56. The parent of 56 will have only one child:

No sibling has three children, so the parent with only one child cannot adopt a spare child. Instead, its sole child must be adopted by one of the parent’s siblings.
After deleting 56, its parent has only one child:

![Tree Diagram]

Let’s choose the parent’s sibling to the left to adopt its parent’s sole child.
After deleting 56, its parent’s sole remaining child (34) is adopted by the parent’s sibling to the left:

```
  34,88
 /    /
2    94
 / \
1  2  34  88  94
```

Now the sibling and its parent’s index keys must be fixed.
When the sibling to the left adopts the sole remaining child of a parent, the index keys are fixed as follows, where $k_g$ is the index key in the grandparent separating the sibling and the parent:

Note that the grandparent may now have only one child (if it had only two children before). This “one-child” problem may repeat all the way up to the root.
After deleting 56, its parent’s sole remaining child (34) was adopted by the parent’s sibling to the left. The grandparent originally had three children and now has two, so the deletion is complete:
Delete 25:

```
       25
     /   \
   15    35
 /   \  /   \n10   20 30  40
/ \ / \ / \ / \
5 10 15 20 25 30 35 40
```
Deletion: Another Hard Case

After deleting 25, its parent (30) has only one remaining child:

![Tree Diagram]

After deleting 25, its parent (30) has only one remaining child:
After deleting 25, its parent (30) with one remaining child had no sibling with an extra child, so the remaining child (leaf 30) is adopted by the parent’s sibling:

Now the grandparent (35) has only one remaining child.
The grandparent with one remaining child had no sibling with an extra child, so the remaining child (interior node 35,40) is adopted by the parent’s sibling:

Now the root has only one remaining child.
If the root has only one child (because one of its children had a sole remaining child that was adopted by a sibling), simply delete the root and make its child the new root.
The root had one remaining child, so we delete the root:

Now the tree is one level shorter.
Delete 56:

The parent will have only one remaining child.
After deleting 56, the parent has only one remaining child:

The parent’s adjacent sibling does not have an extra child, so the adjacent sibling must adopt the parent’s remaining child.
After deleting 56, the parent’s sibling adopts the parent’s only remaining child:

(The sibling’s extra index key came down from its parent.)
Delete 68:

The parent will have only one child (72).
After deleting 68, the parent’s sibling has adopted the parent’s remaining child (72):

Now the grandparent has only one child.
The grandparent’s sibling has adopted the grandparent’s remaining child:

Now the root has only one child.
The old root is deleted and its sole child is the new root:
Delete 1:

```
Deletion Examples

Delete 1:
```

```
[13x257]Deletion Examples
[28x185]Delete 1:
[78x104]1
[0x0]2
[0x0]34
[0x0]39
[0x0]45
[0x0]72
[0x0]88
[0x0]94
[93x132]2
[0x0]39,45
[0x0]88,94
[147x161]34,68
[129x3]Robert Guderian
[0x0]2-3 Trees
```
After deleting 1, the parent adopts an extra child from the adjacent sibling:
Now delete 88:

Robert Guderian 2-3 Trees
Neither the parent of 88 nor the parent’s sibling have to adopt:
Now delete 45:

Its parent will have only one child.
After deleting 45, its parent had only one child, and the parent’s sibling to the right adopted the parent’s remaining child:
Now delete 2:

Its parent will have only one child.
After deleting 2, its parent adopts an extra child from the parent's adjacent sibling:
Now delete 94:

Its parent will have only one child remaining.
Deletion Examples

After deleting 94, its parent’s sibling adopts the parent’s remaining child:

Now the root has only one child remaining.
The old root is deleted and its sole child becomes the new root: