CMSC 420: B-Trees

Justin Wyss-Gallifent

October 31, 2023

1 Introduction

B-trees are generalizations of 2-3 trees in which the number of keys and children is permitted to vary.

In addition in 2-3 trees overfull nodes were always split whereas in B-trees overfull nodes may be fixed by rotation.

Note 1.0.1. In theory we could have fixed overfull nodes in 2-3 trees by rotation but in practice that's not how the definition was formulated originally.

Note 1.0.2. There are several references to 2-3 trees in these notes. If you are not familiar with these it's fine; while knowing 2-3 tree might help a bit, none of those references are essential

2 Definition

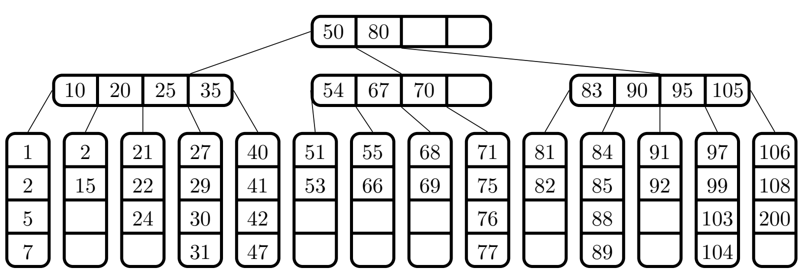
Definition 2.0.1. For an integer $m \geq 3$ a B-tree of order m is a multiway search tree with the following properties, assuming it's not empty:

- The root has between 1 and $m-1$ keys and if it has children then it must have one more children than keys, hence between 2 and m children.
- Each node except the root has between $\lceil m/2 \rceil - 1$ and $m-1$ keys and if it has children then it must have one more children than keys, hence between $\lceil m/2 \rceil$ and m children.
- All leaf nodes are at the same level.

By a *multiway search tree* we mean a generalization of BSTs and 2-3 trees. If a node has keys $a_1 < a_2 < ... < a_k$ then the children are roots of subtrees $A_1, ..., A_{k+1}$ with keys satisfying $A_1 < a_1 < A_2 < a_2 < ... < a_k < A_{k+1}$.

Note 2.0.1. Observe that by definition a B-tree is perfect, meaning it is full on every level.

Example 2.1. Here is an example of a B-tree of order $m = 5$. The root must have between 2 and $m = 5$ children (hence 1 and 4 keys) while the other nodes must have betwee $\lceil m/2 \rceil = \lceil 5/2 \rceil = 3$ and 5 children (hence 2 and 4 keys). The leaf nodes are shown as vertical just for spacing reasons.



3 Height

As with many of our other trees the height is logarithmic as a function of the number of nodes and keys. Here is the proof for keys, which is more relevant since typically we are interested in key counts:

Theorem 3.0.1. Suppose B-tree of order m has k keys and height h. Then:

(a) The sparsest possible such tree (minimum number of keys) has:

$$k = 2\lceil m/2\rceil^h - 1$$

(b) Consequently any such tree has:

$$k \geq 2\lceil m/2\rceil^h - 1$$

(c) And it then follows that:

$$h \leq \log_{\lceil m/2\rceil}\left(\frac{k+1}{2}\right)$$

(d) Thus:

$$h = \mathcal{O}(\lg k)$$

Proof. To obtain the minimum number of keys we need the fewest keys allowed per node. The following table illustrates the minimum number of nodes and keys for each level 0 (the root) through h (the leaves).

Level	Min Nodes	Min Keys
0	1	1
1	2	$2(\lceil m/2\rceil - 1)$
2	$2\lceil m/2\rceil$	$2\lceil m/2\rceil(\lceil m/2\rceil - 1)$
3	$2\lceil m/2\rceil^2$	$2\lceil m/2\rceil^2(\lceil m/2\rceil - 1)$
$\vdots$	$\vdots$	$\vdots$
h	$2\lceil m/2\rceil^{h-1}$	$2\lceil m/2\rceil^{h-1}(\lceil m/2\rceil - 1)$

The total number of keys then satisfies, at minimum:

$$\begin{aligned}
k &= 1 + \sum_{i=0}^{h-1} 2\left\lceil m/2 \right\rceil^i \left(\left\lceil m/2 \right\rceil - 1\right) \\
&= 1 + 2(\left\lceil m/2 \right\rceil - 1) \sum_{i=0}^{h-1} \left\lceil m/2 \right\rceil^i \\
&= 1 + 2(\left\lceil m/2 \right\rceil - 1) \left(\frac{\left\lceil m/2 \right\rceil^h - 1}{\left\lceil m/2 \right\rceil - 1} \right) \\
&= 1 + 2(\left\lceil m/2 \right\rceil^h - 1) \\
&= 2\left\lceil m/2 \right\rceil^h - 1
\end{aligned}$$

It follows that since this is a minimum that:

$$k \geq 2\left\lceil m/2 \right\rceil^h - 1$$

Then:

$$\begin{aligned}
2\left\lceil m/2 \right\rceil^h - 1 &\leq k \\
\left\lceil m/2 \right\rceil^h &\leq \frac{k+1}{2} \\
h &\leq \log_{\lceil m/2 \rceil} \left(\frac{k+1}{2} \right)
\end{aligned}$$

$\mathcal{QED}$

Example 3.1. For example if a B-tree of order $m = 20$ contains $k = 999$ keys then the maximum possible height can be calculated via:

$$h \leq \log_{\lceil 20/2 \rceil} \left(\frac{999+1}{2} \right) = \log_{10} 500 \approx 2.6987$$

Since height must be an integer the maximum height is 2.

As an aside, the sparsest possible B-tree of order $m = 20$ with height $h = 2$ has:

$$k = 2\left\lceil 20/2 \right\rceil^2 - 1 = 199$$

So our B-tree (with $m = 20$ and $h = 2$) with 999 keys is far from being the sparsest.

To reinforce why $h \leq 2$ observe that the sparsest possible B-tree of order $m = 20$ with larger height $h = 3$ has:

$$k = 2\lceil 20/2 \rceil^3 - 1 = 1999$$

Thus any B-tree of order $m = 20$ with larger height $h = 3$ has $k \geq 1999$ and so our 999 keys are not enough for a B-tree of order $m = 20$ with height $h = 3$.

Theorem 3.0.2. Suppose B-tree of order m has k keys and height h. Then:

(a) The densest possible such tree (maximum number of keys) has:

$$k = m^{h+1} - 1$$

(b) Consequently any such tree has:

$$k \leq m^{h+1} - 1$$

(c) And it then follows that:

$$h \geq \log_m(k+1) - 1$$

(d) Thus:

$$h = \Omega(\lg k)$$

Proof. Omitted. Try it! It's similar to but easier than the previous. $\mathcal{QED}$

Example 3.2. For example if a B-tree of order $m = 20$ contains $k = 999$ keys then the minimum possible height can be calculated via:

$$h \geq \log_{20}(999+1) - 1 \approx 1.3059$$

Since height must be an integer the minimum height is 2.

As an aside, the densest possible B-tree of order $m = 20$ with height $h = 2$ has:

$$k = 20^{2+1} - 1 = 7999$$

So our B-tree (with $m = 20$ and $h = 2$) with 999 keys is far from being the densest.

To reinforce why $h \geq 2$ observe that the densest possible B-tree of order $m = 20$ with smaller height $h = 1$ has:

$$k = 20^{1+1} - 1 = 399$$

Thus any B-tree of order $m = 20$ with smaller height $h = 1$ has $k \leq 399$ and so our 999 keys could not fit in a B-tree of order $m = 20$ with height $h = 1$.

Theorem 3.0.3. We have $h = \Theta(\lg k)$.

Proof. Follows immediately. $\mathcal{QED}$

4 Advantages

There are several advantages to using B-trees, including:

- Due to the number of keys that a node may contain there is consequently less tree balancing required when inserts and deletions occur.
- When a key is found in a node, a large collection of close keys are immediately accessible. This manifests in file storage where access (finding the node) is far slower than reading the data (once the node has been found).
- When doing range queries (find all values between x and y) it's easy to pluck grouped values out of a node.
- Although as we'll see the restructuring process must be managed carefully it turns out that it only happens infrequently because of the amount of empty key space permissible in a node.

5 Search

Since this is a multiway search tree, finding a key is easy, just like with binary search trees, AVL trees, and 2-3 trees.

6 Tree Restructuring

6.1 Introduction

Recall that in a 2-3 tree we had to manage the issues of an overfull node when a 3-node became a 4-node, and an underfull node when a 2-node became a 1-node. Similarly for B trees we must manage:

- Overfull Node: A node has $m + 1$ children (m keys).
- Underfull Node: A node has $\lceil m/2 \rceil - 1$ children ($\lceil m/2 \rceil - 2$ keys).

To deal with these issues we introduce three restructuring operations:

6.2 Rotation

The best possible situation arises when a node is underfull or overfull but there are extra keys in an adjacent sibling that we can use to restructure. This is called a *key rotation* and it's the best possible because it's not computationally intensive.

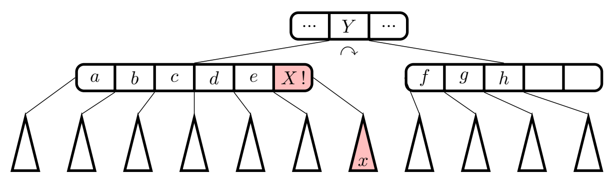
Suppose a key in node n_1 is overfull but the sibling n_2 directly to the right has key space. We take the largest key in n_1, promote it to and replace the next largest key in the parent which gets demoted to the n_2 sibling, putting it at the start of n_2's keys. We also move n_1's largest child to become n_2's smallest child. This is a right rotation.

A mirror argument works if a node n_1 is overfull but the sibling n_2 directly to the left has key space. We take the smallest key in n_1, promote it to and replace the next smallest key in the parent which gets demoted to the n_2 sibling, putting it at the end of n_2's keys. We also move n_1's smallest child to become n_2's largest child. This is a left rotation.

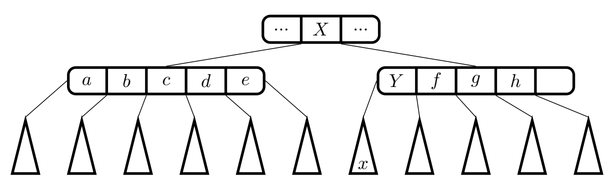
This approach will also work if a node is underfull and a sibling has an extra key it can donate.

Note that this will only work if there is an adjacent sibling in a position to help!

Here is an illustration of a right rotation in action for $m = 6$, The node on the left is overfull, it (temporarily) has seven children and six keys. Its sibling on the right has space so we rotate over.



$\Downarrow$ Right Rotate!



6.3 Splitting

Consider a B-tree of order m. A node can have at most have m children and $m-1$ keys but suppose it temporarily has $m+1$ children and m keys. This can arise from an insertion as we will see shortly.

In brief we take the median (or lower median) key from the overfull node and we promote it to the parent, inserting it at the location of the parent's branch which led to the overfull node. That branch is eliminated. The left and right keys and children from the overfull node break up to form two new nodes which are connected to the parent by two new branches which straddle the newly inserted node.

As a result of this the parent gains a key and a child, meaning it may also overflow and we then need to manage that problem one level up.

While this simplistic explanation works fine in practice it can be helpful to convince ourselves that the numbers work out nicely in terms of key and child counts.

If the number of keys m is odd:

(a) Take the median key and promote it to the parent, inserting it where the node's parent branch extended. This leaves an even $m-1$ keys with $(m-1)/2$ smaller and $(m-1)/2$ larger.

(b) Take the $(m-1)/2$ keys which are smaller than this median as well as the $(m-1)/2+1=(m+1)/2$ leftmost children and create a new node.

(c) Take the $(m-1)/2$ keys which are larger than this median as well as the $(m-1)/2+1=(m+1)/2$ rightmost children and create a new node.

To ensure this works we have to make sure that each of the new nodes is valid, meaning each has between $\lceil m/2 \rceil$ and m children. In other words we claim that:

$$\left\lceil \frac{m}{2} \right\rceil \leq \frac{m+1}{2} \leq m$$

But since m is odd we have $\lceil m/2 \rceil = (m+1)/2$ so the left inequality holds and $(m+1)/2 \leq m$ is equivalent to $m+1 \leq 2m$ and to $m \geq 1$ so the right inequality holds.

> **Example 6.1.** For example if $m=7$ then we can have at most 6 keys. If there are 7 keys (hence 8 children) we take the median key and promote it to the parent. We take the 3 smaller keys as well as the 4 leftmost children and create a new node and we take the 3 larger keys as well as the 4 rightmost children and create a new node. Observe that the number of children of each of these new nodes satisfies $\lceil m/2 \rceil = 4 \leq 4 \leq 7 = m$.

If the number of keys m is even:

(a) Take the lower median key and promote it to the parent, inserting it where the node's parent branch extended. This leaves an odd $m-1$ keys with $(m-2)/2$ smaller and $m/2$ larger.

(b) Take the $(m-2)/2$ keys smaller than this lower median as well as the $(m-2)/2+1=m/2$ leftmost children and create a new node.

(c) Take the $m/2$ keys larger than this lower median as well as the $m/2+1$ right children and create a new node.

To ensure this works we have to make sure that each of the new nodes is valid, meaning each has between $\lceil m/2 \rceil$ and m children. In other words for the new leftmost node we claim that:

$$\left\lceil \frac{m}{2} \right\rceil \leq \frac{m}{2} \leq m$$

But since m is even we know $\lceil m/2 \rceil = m/2$ so the left inequality holds and $m/2 \leq m$ is equivalent to $m/2 \geq 0$ so the right inequality holds.

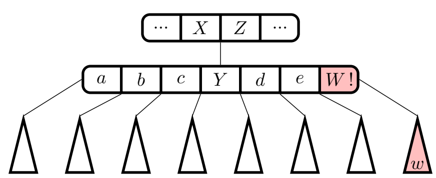
And for the new rightmost node we claim that:

$$\left\lceil \frac{m}{2} \right\rceil \leq \frac{m}{2} + 1 \leq m$$

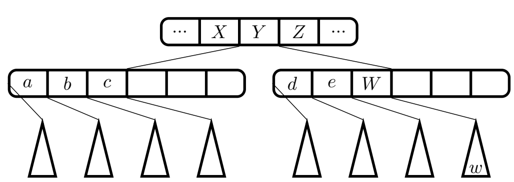
But since m is even we know $\lceil m/2 \rceil = m/2$ so the left inequality holds and $m/2 + 1 \leq m$ is equivalent to $m \geq 2$ so the right inequality holds.

> **Example 6.2.** For example if $m = 8$ then we can have at most 7 keys. If there are 8 keys (hence 9 children) we take the lower median key and promote it to the parent. We take the 3 smaller keys as well as the 4 leftmost children and create a new node and we take the 4 larger keys as well as the 5 rightmost children and create a new node. Observe that the number of children of the leftmost new node satisfies $\lceil m/2 \rceil = 4 \leq 4 \leq 8 = m$ and the rightmost new node satisfies $\lceil m/2 \rceil = 4 \leq 5 \leq 8 = m$.

Here is an illustration of a split in action for $m = 7$. In such a B-tree a non-leaf node may have between $\lceil m/2 \rceil = \lceil 7/2 \rceil = 4$ and $m = 7$ childrean and between 3 and 6 keys. The middle node is overfull, it (temporarily) has 8 children and 7 keys. We split it.



$\Downarrow$ Split!



6.4 Merging

Consider a B-tree of order m. A node can at least have $\lceil m/2 \rceil$ children and $\lceil m/2 \rceil - 1$ keys but suppose it temporarily has $\lceil m/2 \rceil - 1$ children and $\lceil m/2 \rceil - 2$

keys. This can arise directly from a deletion as we will see shortly.

First, if either of its adjacent siblings has more than $\lceil m/2 \rceil$ children then a key rotation does the job and we're fine.

If not, then both siblings have exactly $\lceil m/2 \rceil$ children. What we will do is pick one of them and merge our underfull node with that sibling.

Notice that when we do this the parent loses a child (because two siblings merge to one) which means it must lose a key. We take, from the parent, the key which separated the two siblings and we demote that key to the newly merged node, inserting it between the keys from the two merging nodes. It turns out that this works well except for the fact that then the parent may be underfull now and we then need to manage that problem one level up.

While this simplistic explanation works fine in practice it can be helpful to convince ourselves that the numbers work out nicely in terms of key and child counts.

Observe that our underfull node has $\lceil m/2 \rceil - 1$ children and $\lceil m/2 \rceil - 2$ keys and the sibling we chose has at $\lceil m/2 \rceil$ children and $\lceil m/2 \rceil - 1$ keys. When we merge them there are $2\lceil m/2 \rceil - 1$ children and $2\lceil m/2 \rceil - 3$ keys. Note that this is enough children but not enough keys; we'll cross that bridge in a minute.

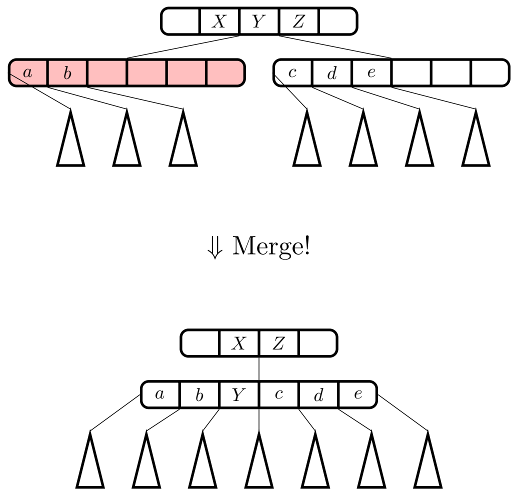
First we need to ascertain that:

$$\lceil m/2 \rceil \leq 2\lceil m/2 \rceil - 1 \leq m$$

This can be proved with an even-odd argument as with splitting. Try it!

Now for the key issue. We're short one key so we demote the key from the parent which was between the edges which connect the two adjacent nodes we are merging. The parent loses a key and a child, so its count is fine, but it might be underfull, and when this appears in the delete procedure it will just propagate.

Here is an illustration of a merge in action for $m = 7$, The left node is underfull but neither adjacent sibling can offer a key (left sibling not shown) so we merge with a sibling (the right in this case).

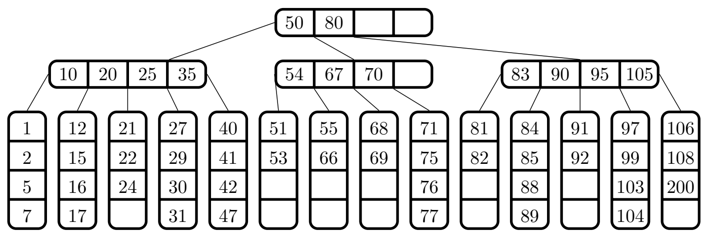


7 Insert

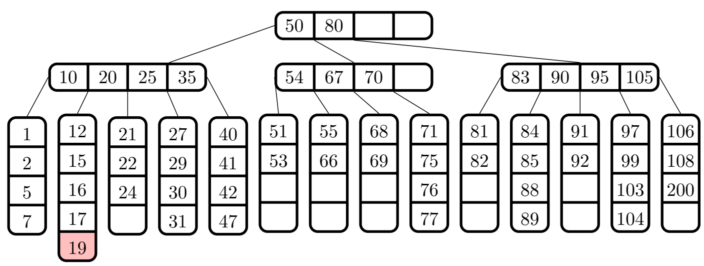
7.1 Algorithm

At this point insertion is easy. We find the correct leaf node and insert it and then we rotate and/or split up the tree until the restructuring is finished. Note that it's possible that no restructuring is required at all.

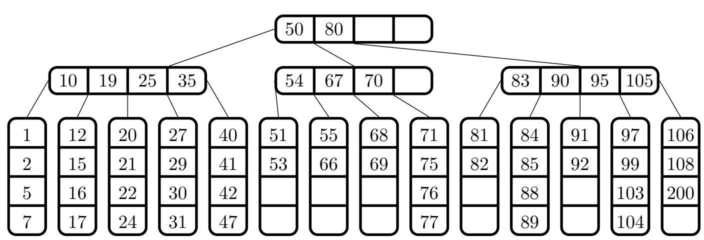
Example 7.1. Let's insert 19 into this B-tree:



⇓

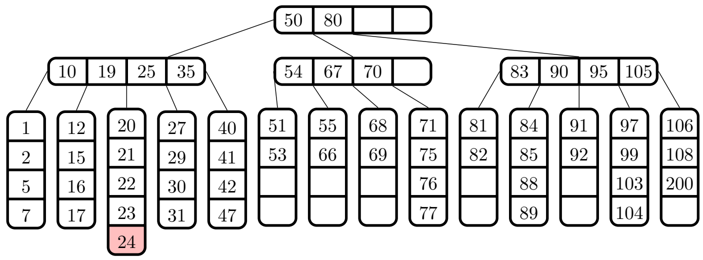


Luckily there is space in its right sibling and so we can do a rotation:

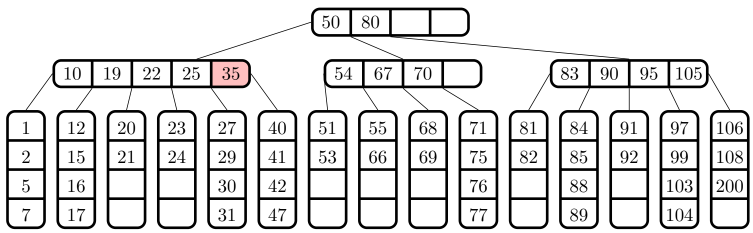


Now we are done.

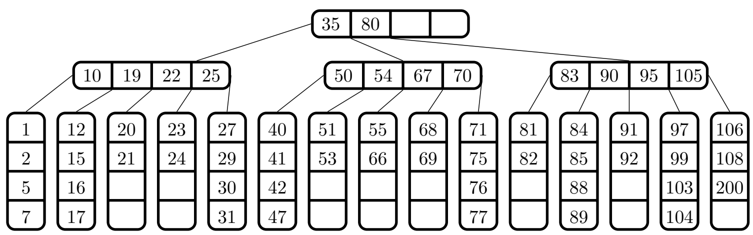
Example 7.2. Let's say we take the result of the previous exercise and insert 23. This overflows that leaf:



Neither adjacent sibling has space for a rotation so instead we split the overfull node in the middle, at the 22 and promote that middle 22. Unfortunately due to the promotion of the 22 the parent node is now overfull:



Luckly the parent has a sibling to the right which can accept a key, so we rotate it over:



Now we are done.

7.2 Time Complexity

Each rotation and split is $\Theta(1)$, occuring up to $\Theta(\lg n)$ or $\Theta(\lg k)$ times, for a total worst-case of $\Theta(\lg n)$ or $\Theta(\lg k)$.

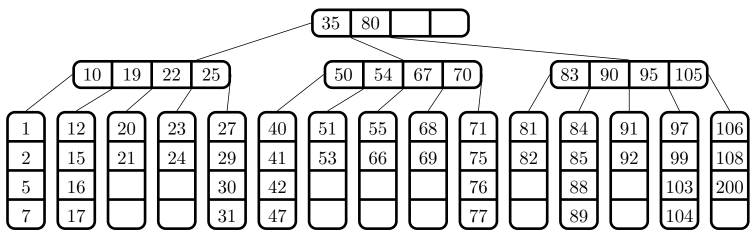
8 Delete

8.1 Algorithm

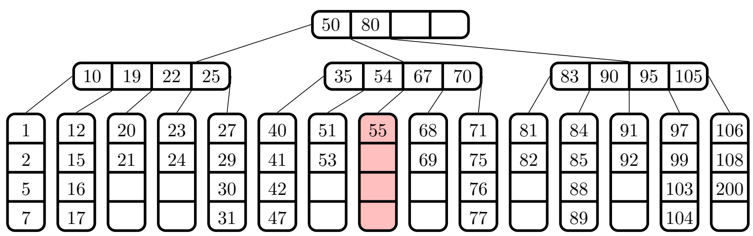
At this point deletion is easy. If the target key is not in a leaf then we find a replacement key (the inorder predecessor or successor). This replacement key will necessarily be in a leaf so effectively we are deleting a key from a leaf.

If the leaf is not underfull we are done, otherwise we rotate and/or merge up the tree until the restructuring is finished.

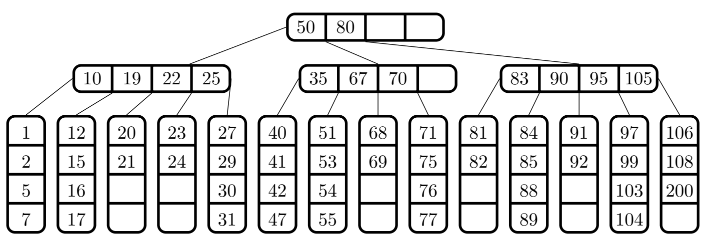
Example 8.1. Let's delete 66 from this tree:



The result yields an underfull node:



Neither adjacent sibling can offer a key via rotation so the only choice is to merge with a sibling. Let's merge with the left sibling, which means that the sandwiched key 54 in the parent is pulled down:



Luckily the parent could give up a key with no issue and we are done.

8.2 Time Complexity

Each rotation and merge is $\Theta(1)$, occuring up to $\Theta(\lg n)$ or $\Theta(\lg k)$ times, for a total worst-case of $\Theta(\lg n)$ or $\Theta(\lg k)$.

9 B+ Trees

9.1 Structure

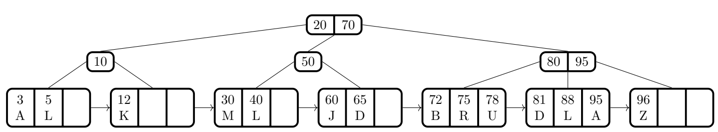
A B+ tree is a variation on a B-tree whereby:

- Internal nodes do not store values (the actual data) but rather just the keys. In our representations we haven't actually shown values, just keys, so the image below has some to help clarify.
- All the key-value pairs are stored in the leaf nodes.
- Each leaf node has a pointer which points to the leaf node to the right.

Essentially the keys in the internal node are guideposts to the leaf nodes and the leaf nodes contain the key-value pairs which really constitute the data.

Note 9.1.1. Insertion and deletion in B+ trees are more complicated than in B trees because:

- Insertion always inserts in a leaf and internal nodes are only created as part of the splitting process. However we can't simply promote a key from a leaf to its parent because the key needs to remain in the leaf to be associated to its data. Consequently a key may appear twice in the tree - in a leaf and in an internal node.
- Since keys may appear both in leaf as well as internal nodes, deletion of a key must take care of both.

> **Example 9.1.** Here is a B+ tree with $m = 4$. As with a B-tree the root node may have between 2 and $m = 4$ children and hence between 1 and 3 keys and every other node may have between $\lceil m/2 \rceil = 2$ and $m = 4$ children and hence between 1 and 3 keys: Each leaf node also has a value (some data) associated to its key.
>
> 
>

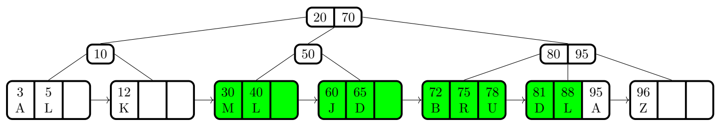
B+ trees have a few benefits including:

- Since the values (data) are saved only in the leaves this saves space in internal nodes.
- All queries (looking for keys with associated values) will reliably travel to the bottom of the tree.
- Range queries are especially nice. For example in the above tree if we're looking for all keys (with values) in the range $[30, 88]$ we simply find the 30 and then follow the leaf nodes across.

9.2 Range Queries

Here is a range query example.

Example 9.2. Same example with the above range query $[30, 88]$:



9.3 Application: Databases

Imagine a sequential database in which each row has a non-unique ID and the rows are in no particular order.

If we wish to query for all rows with a particular ID we would need to go through all the rows sequentially and pick out those with that ID. This can take a long time.

If we create an index on the ID the result is typically a B+ tree in which the IDs are the keys and the values are the row numbers. Then when we search for a particular ID, or range of IDs, we quickly get back a list of rows we ought to look at and we can go directly those rows in the database to get the actual data.