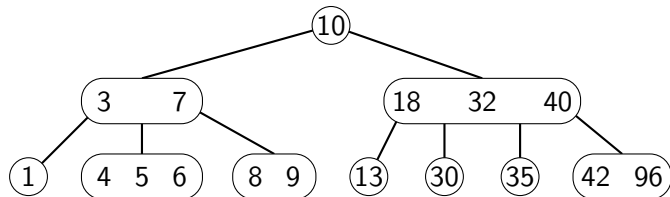


2-3-4 trees II: Deletion and alternative forms

COMS20017 (Algorithms and Data)

John Lapinskas, University of Bristol

Summary of a 2-3-4 tree with distinct values (so far)



Finding a value v : Let x be the root. If $v \in x$, return a pointer to x . Otherwise, if x is a leaf, return Not Found. Otherwise, let k be such that x is a k -node, let $x_1 \leq \dots \leq x_{k-1}$ be the values in x , let $x_0 = -\infty$, and let $x_k = \infty$; then $x_{i-1} < v < x_i$ for some i . Let c be the i 'th child of x . Then repeat the process from the start, taking $x = c$.

Inserting a value v : First attempt to find v as above, **splitting** any 4-nodes encountered (including the root). After reaching a leaf L , and splitting it if it is a 4-node, add v to L .

Deleting from a leaf: Dealing with 2-nodes

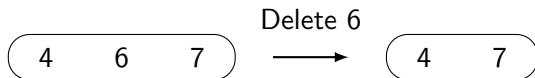
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node...

Deleting from a leaf: Dealing with 2-nodes

First suppose the value we're trying to delete is a leaf.

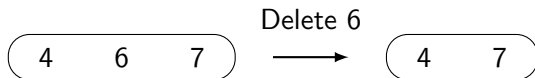
If it's in a 3-node or a 4-node... we just remove it:



Deleting from a leaf: Dealing with 2-nodes

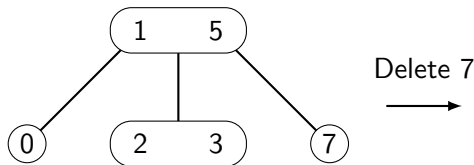
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

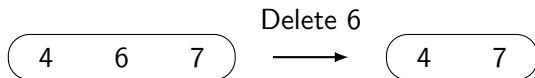
If v 's left (or right) sibling is **not** a 2-node, then we can **transfer** its rightmost (or leftmost) value to v :



Deleting from a leaf: Dealing with 2-nodes

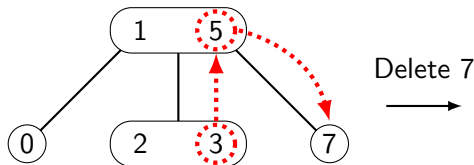
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

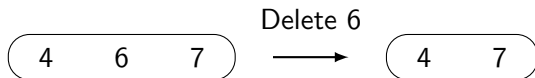
If v 's left (or right) sibling is **not** a 2-node, then we can **transfer** its rightmost (or leftmost) value to v :



Deleting from a leaf: Dealing with 2-nodes

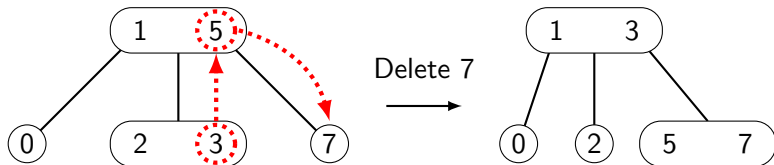
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

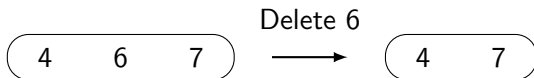
If v 's left (or right) sibling is **not** a 2-node, then we can **transfer** its rightmost (or leftmost) value to v :



Deleting from a leaf: Dealing with 2-nodes

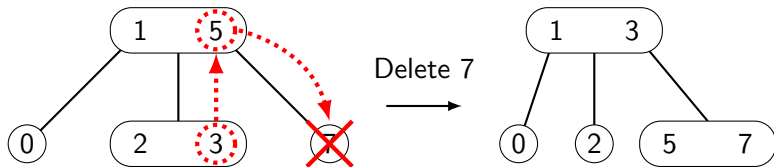
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

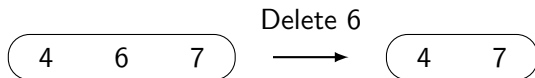
If v 's left (or right) sibling is **not** a 2-node, then we can **transfer** its rightmost (or leftmost) value to v :



Deleting from a leaf: Dealing with 2-nodes

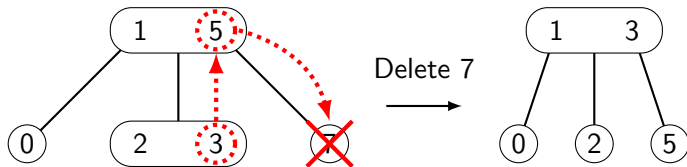
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

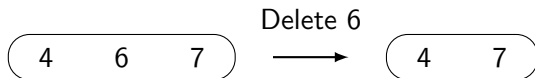
If v 's left (or right) sibling is **not** a 2-node, then we can **transfer** its rightmost (or leftmost) value to v :



Deleting from a leaf: Dealing with 2-nodes

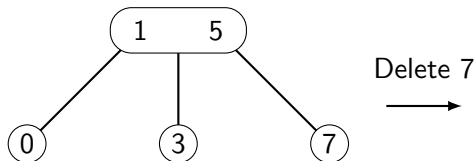
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

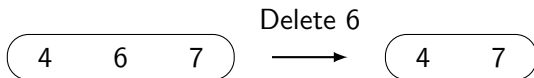
If v 's left (or right) sibling **is** a 2-node, and its parent is **not** a 2-node, then we can **fuse** the two siblings together in the opposite of a split:



Deleting from a leaf: Dealing with 2-nodes

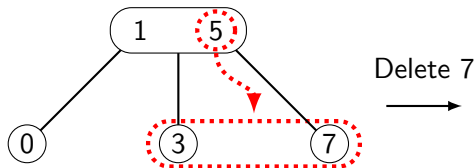
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

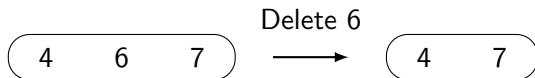
If v 's left (or right) sibling **is** a 2-node, and its parent is **not** a 2-node, then we can **fuse** the two siblings together in the opposite of a split:



Deleting from a leaf: Dealing with 2-nodes

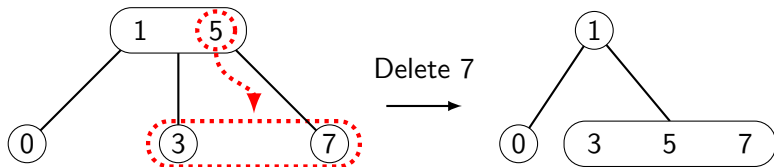
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

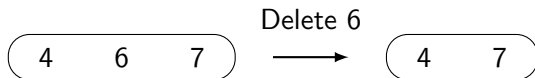
If v 's left (or right) sibling **is** a 2-node, and its parent is **not** a 2-node, then we can **fuse** the two siblings together in the opposite of a split:



Deleting from a leaf: Dealing with 2-nodes

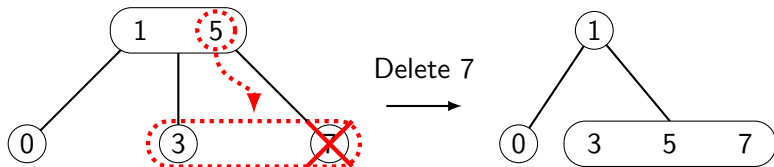
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

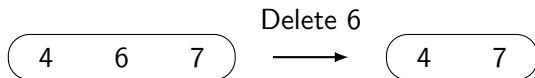
If v 's left (or right) sibling **is** a 2-node, and its parent is **not** a 2-node, then we can **fuse** the two siblings together in the opposite of a split:



Deleting from a leaf: Dealing with 2-nodes

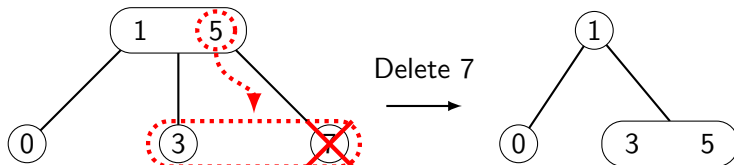
First suppose the value we're trying to delete is a leaf.

If it's in a 3-node or a 4-node... we just remove it:



If it's in a 2-node v , this would break perfect balance. So like with insertion, we need to first turn v into a 3-node or a 4-node.

If v 's left (or right) sibling **is** a 2-node, and its parent is **not** a 2-node, then we can **fuse** the two siblings together in the opposite of a split:



Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Deleting from a leaf: The full procedure

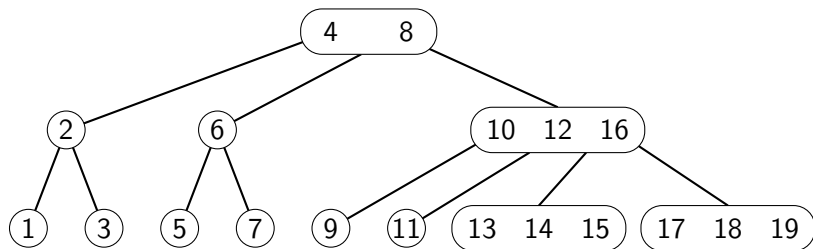
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

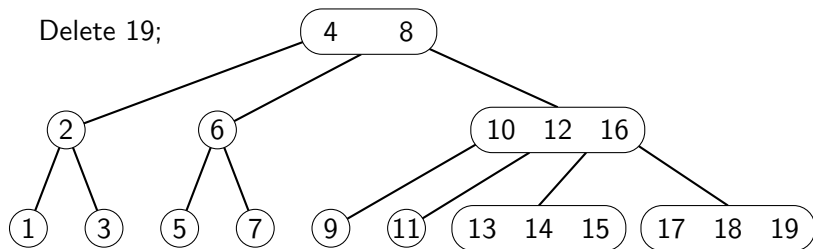
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

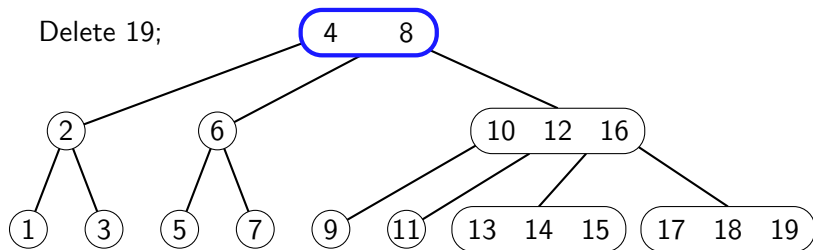
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

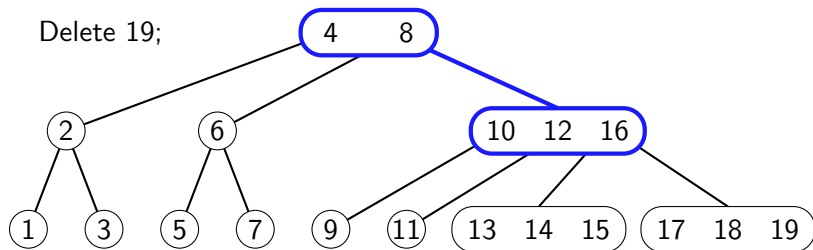
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

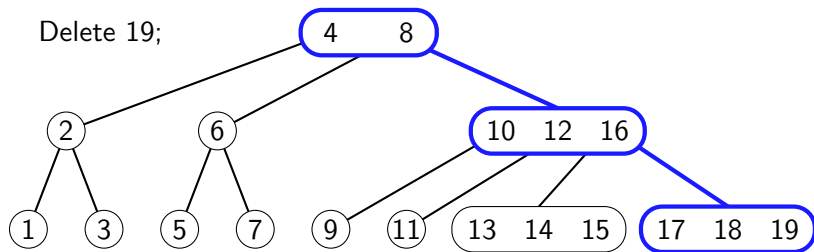
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

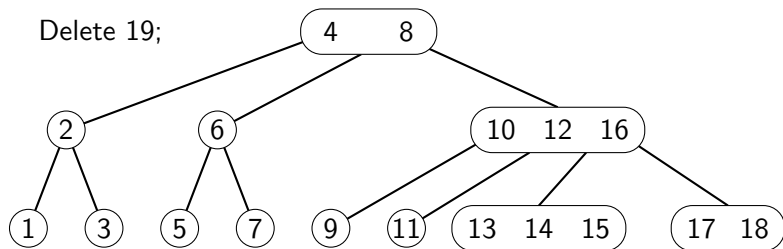
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

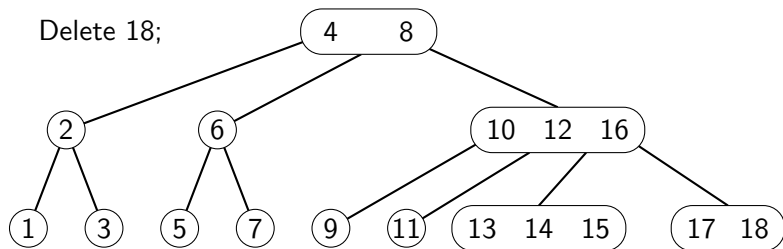
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

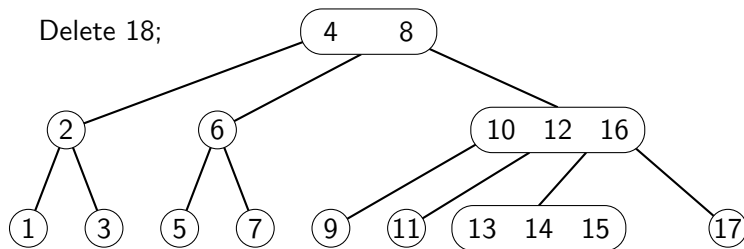
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

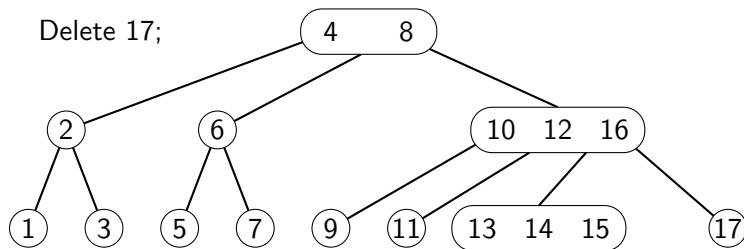
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

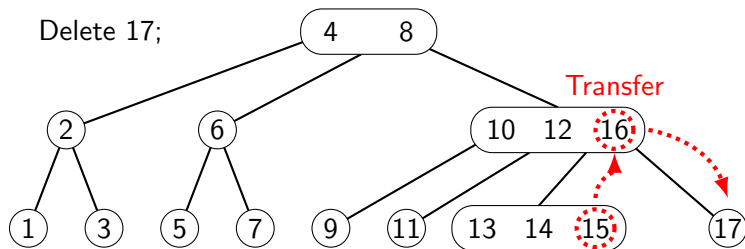
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

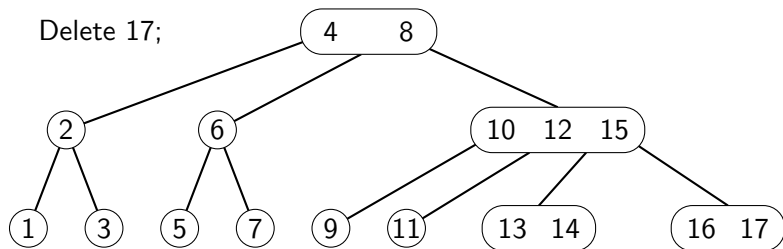
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

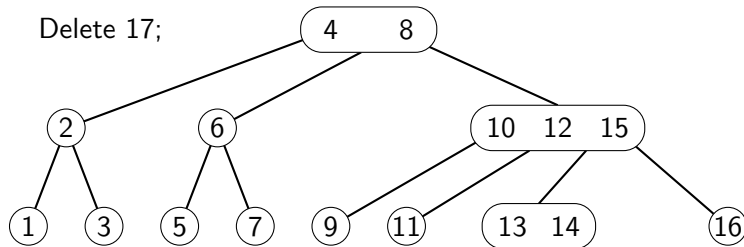
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

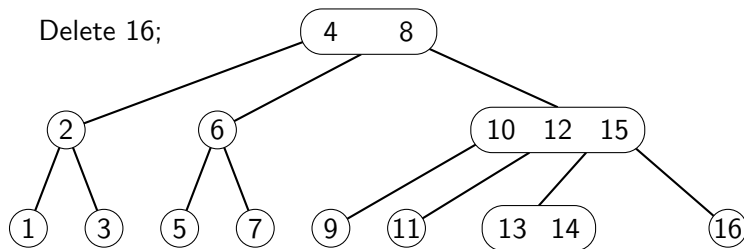
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

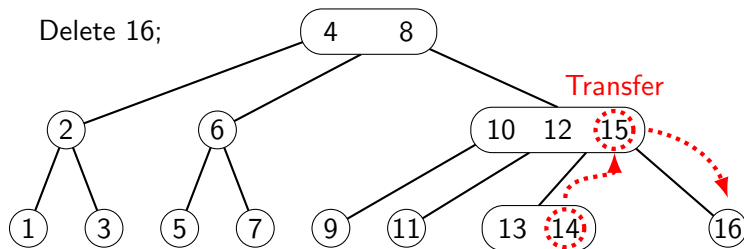
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

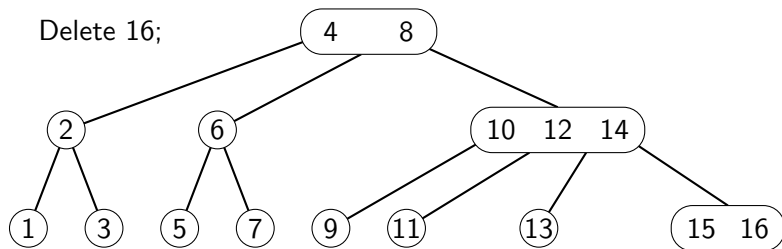
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

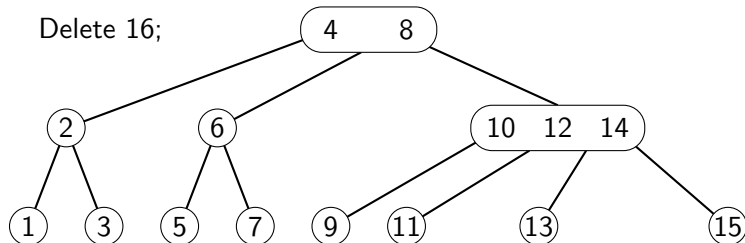
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

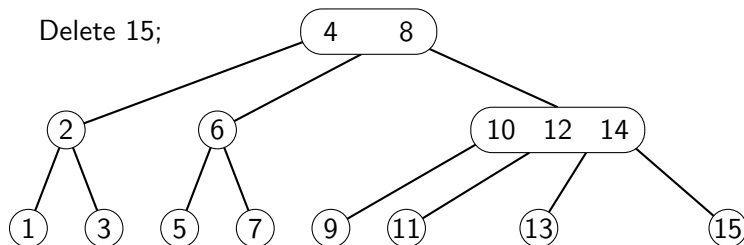
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

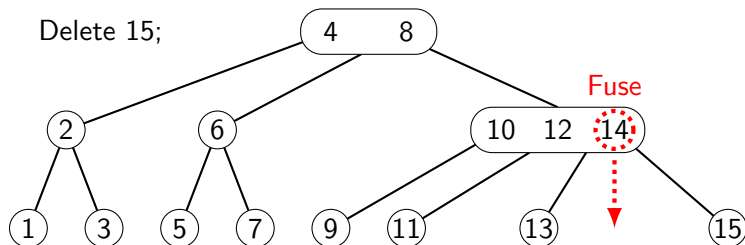
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

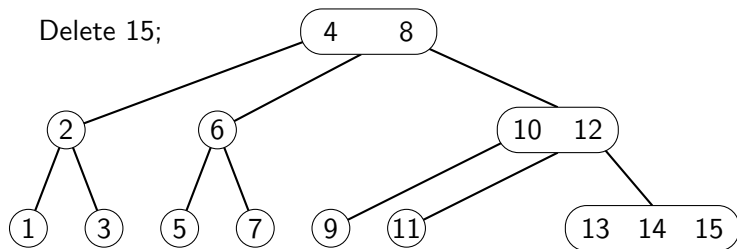
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

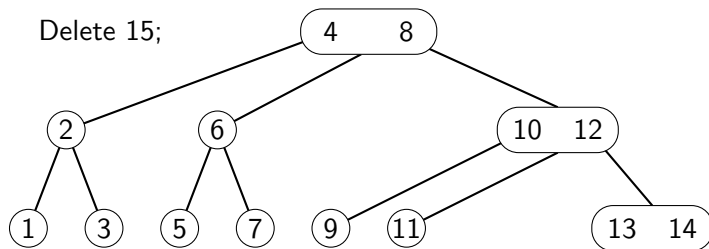
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

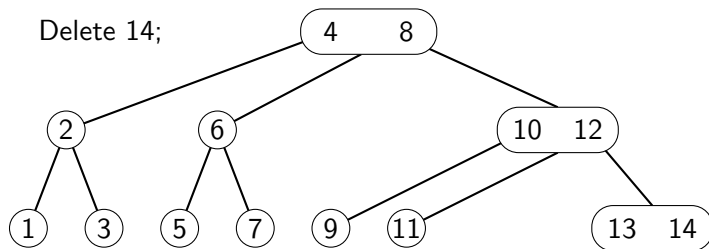
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

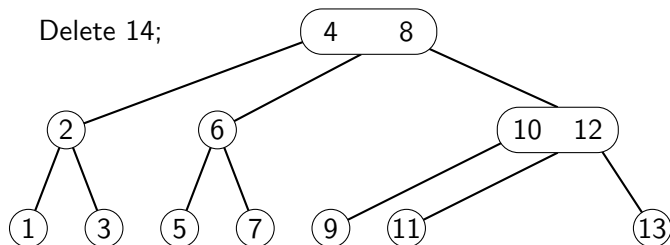
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

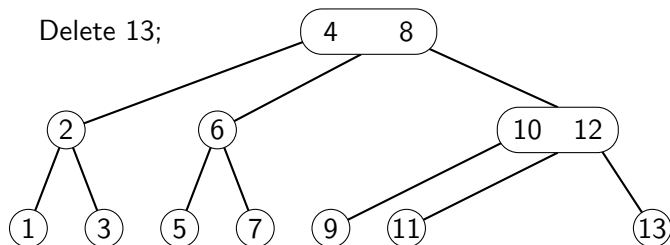
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

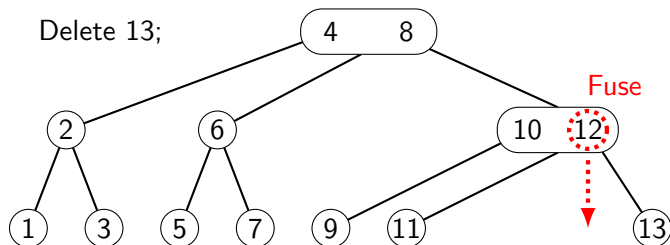
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

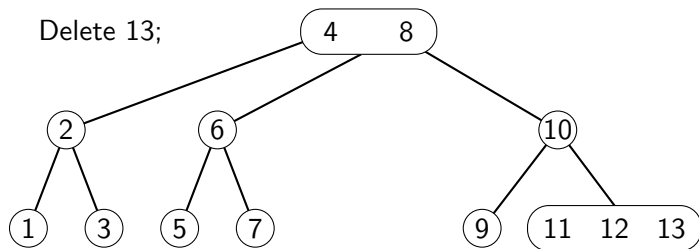
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

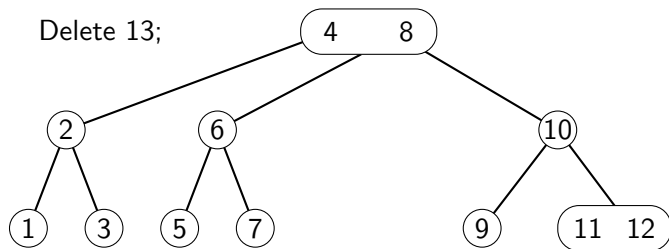
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

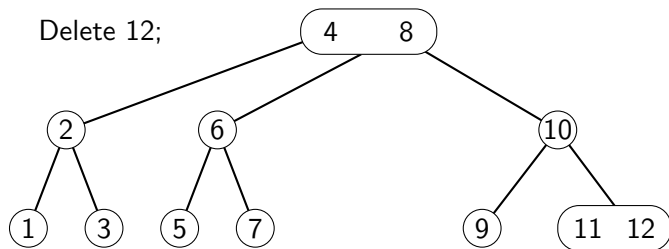
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

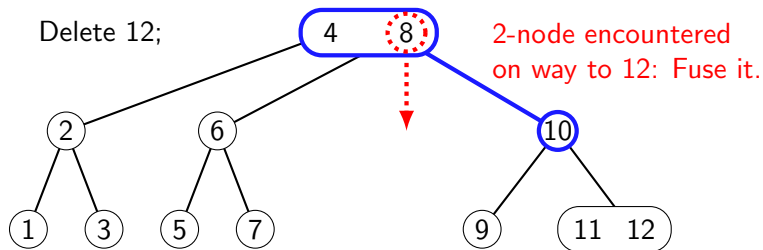
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

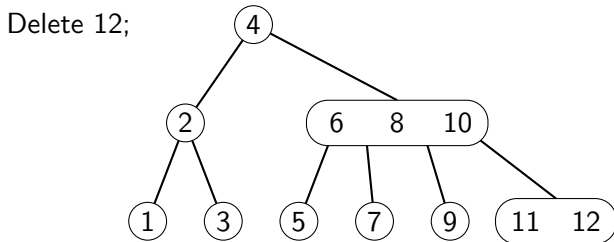
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

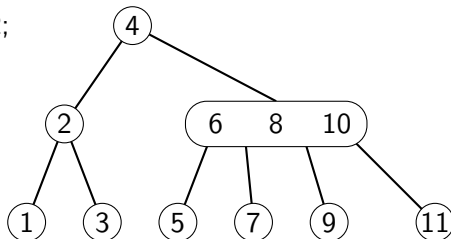
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 12;



Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

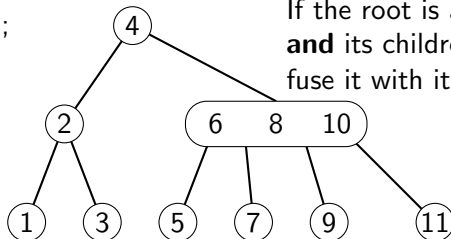
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 11;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

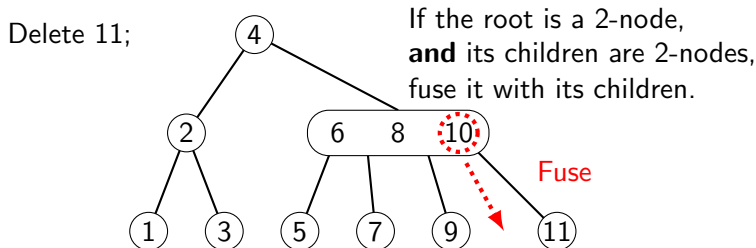
Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)



Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

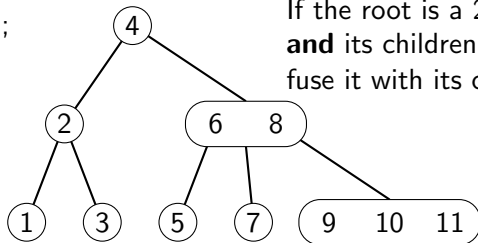
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 11;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

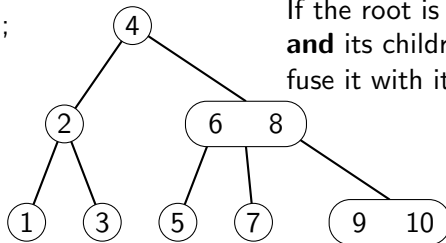
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 11;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

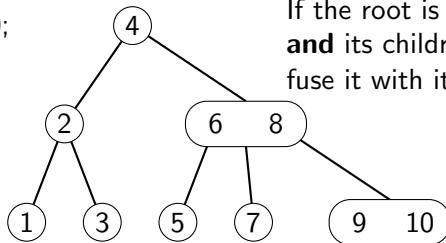
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 10;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

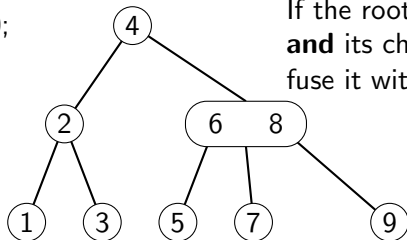
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 10;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

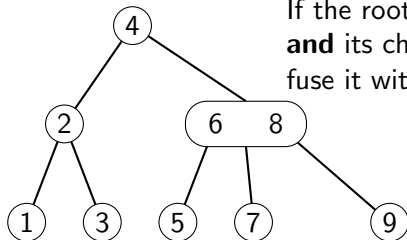
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 9;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

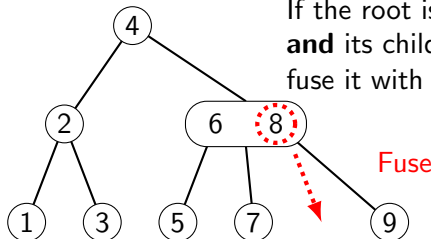
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 9;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

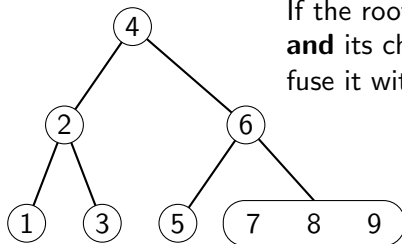
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 9;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

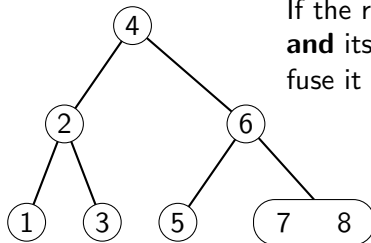
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 9;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

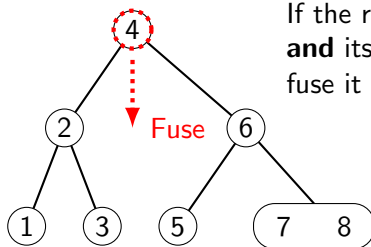
If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 8;



If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.

Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

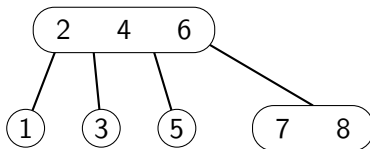
If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

Delete 8;

If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.



Deleting from a leaf: The full procedure

Say we are trying to delete a value from a **leaf** v . If v is a 3-node or 4-node, we just delete it.

If v is a 2-node with a 3-node or 4-node sibling w , we **transfer** a value from w to v , reducing to the 3-node case.

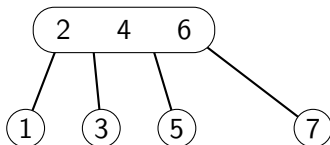
If v is a 2-node with a 2-node sibling w , and a 3-node or 4-node parent, we **fuse** v , w and a value from v 's parent, reducing to the 4-node case.

Like with insertion, we will make sure we never have a 2-node with a 2-node parent by fusing and transferring 2-nodes as we descend.

(These operations work on non-leaves as well!)

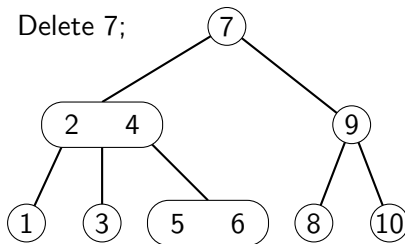
Delete 8;

If the root is a 2-node,
and its children are 2-nodes,
fuse it with its children.



Deleting from a non-leaf

When deleting from a non-leaf, preserving perfect balance is harder.
So let's **reduce** the problem to deleting from a leaf!

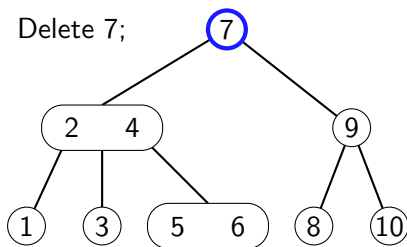


Exercise: If v is not stored in a leaf, then the **predecessor** w of v — the value just before v in sorted order — will always be in a leaf.

So we can overwrite v with w , and then delete w from its leaf — leaving the structure of the tree untouched!

Deleting from a non-leaf

When deleting from a non-leaf, preserving perfect balance is harder.
So let's **reduce** the problem to deleting from a leaf!

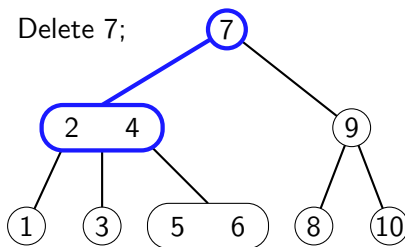


Exercise: If v is not stored in a leaf, then the **predecessor** w of v — the value just before v in sorted order — will always be in a leaf.

So we can overwrite v with w , and then delete w from its leaf — leaving the structure of the tree untouched!

Deleting from a non-leaf

When deleting from a non-leaf, preserving perfect balance is harder.
So let's **reduce** the problem to deleting from a leaf!

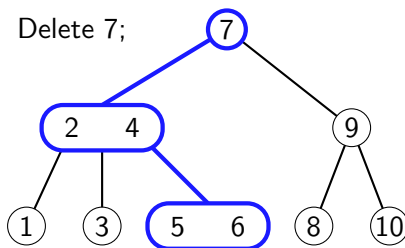


Exercise: If v is not stored in a leaf, then the **predecessor** w of v — the value just before v in sorted order — will always be in a leaf.

So we can overwrite v with w , and then delete w from its leaf — leaving the structure of the tree untouched!

Deleting from a non-leaf

When deleting from a non-leaf, preserving perfect balance is harder.
So let's **reduce** the problem to deleting from a leaf!

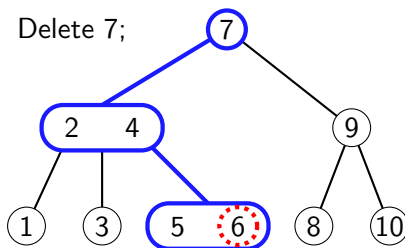


Exercise: If v is not stored in a leaf, then the **predecessor** w of v — the value just before v in sorted order — will always be in a leaf.

So we can overwrite v with w , and then delete w from its leaf — leaving the structure of the tree untouched!

Deleting from a non-leaf

When deleting from a non-leaf, preserving perfect balance is harder.
So let's **reduce** the problem to deleting from a leaf!

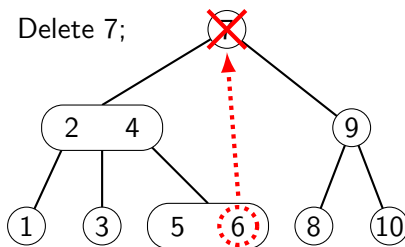


Exercise: If v is not stored in a leaf, then the **predecessor** w of v — the value just before v in sorted order — will always be in a leaf.

So we can overwrite v with w , and then delete w from its leaf — leaving the structure of the tree untouched!

Deleting from a non-leaf

When deleting from a non-leaf, preserving perfect balance is harder.
So let's **reduce** the problem to deleting from a leaf!

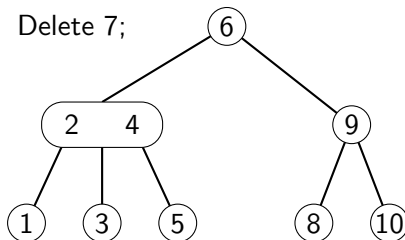


Exercise: If v is not stored in a leaf, then the **predecessor** w of v — the value just before v in sorted order — will always be in a leaf.

So we can overwrite v with w , and then delete w from its leaf — leaving the structure of the tree untouched!

Deleting from a non-leaf

When deleting from a non-leaf, preserving perfect balance is harder.
So let's **reduce** the problem to deleting from a leaf!

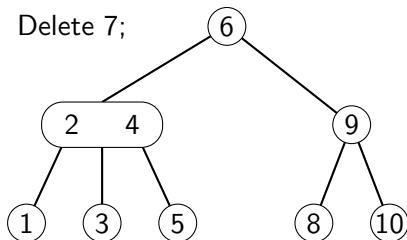


Exercise: If v is not stored in a leaf, then the **predecessor** w of v — the value just before v in sorted order — will always be in a leaf.

So we can overwrite v with w , and then delete w from its leaf — leaving the structure of the tree untouched!

Deleting from a non-leaf

When deleting from a non-leaf, preserving perfect balance is harder.
So let's **reduce** the problem to deleting from a leaf!



Exercise: If v is not stored in a leaf, then the **predecessor** w of v — the value just before v in sorted order — will always be in a leaf.

So we can overwrite v with w , and then delete w from its leaf — leaving the structure of the tree untouched!

In general, this will be its own delete operation, and might require fusing/transferring 2-nodes as normal.

Summary of 2-3-4 tree operations

- Find(v):
 - Descend the tree recursively, using the rule that all children between x and y have values between x and y . If you reach a leaf not containing v , return Not found.
- Insert(v):
 - Apply Find(v) until reaching the leaf ℓ where v would be if it was already in the tree.
 - If ℓ is a 2-node or a 3-node, add v to ℓ .
 - Otherwise, **split** ℓ into 2-nodes and add v to the appropriate new leaf.
 - To avoid ℓ being a 4-node with a 4-node parent, split 4-nodes on the way down (including the root).
- Delete(v):
 - Apply Find(v) to find the vertex ℓ containing v .
 - If ℓ is not a leaf, find v 's predecessor w , overwrite v with w , and Delete(w).
 - Otherwise, if ℓ is a 3-node or a 4-node, delete v from ℓ .
 - Otherwise, if ℓ 's left or right sibling is a 3-node or 4-node, **transfer** from it to make ℓ a 3-node, then delete v .
 - Otherwise, **fuse** ℓ with its 2-node sibling to make ℓ a 4-node, then delete v .
 - To avoid ℓ being a 2-node with a 2-node parent, fuse or transfer 2-nodes on the way down (including the root).

All operations take $O(d)$ time and maintain perfect balance. Perfect balance implies that in an n -element tree, $d \in O(\log n)$ (exercise!), so we're done.

Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement...

Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement... But they are secretly the same thing!

Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement... But they are secretly the same thing!

- Red-black trees are binary search trees where every non-leaf has exactly 2 children, and vertices are coloured red or black.

Exception: A black node with a single red leaf child is OK.

- Every root-leaf path has the same number of black vertices.
- No red vertex has a red child.

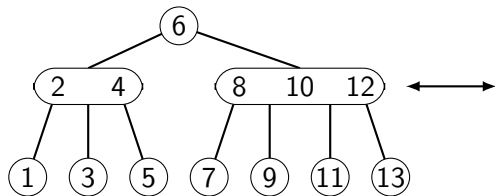
Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement... But they are secretly the same thing!

- Red-black trees are binary search trees where every non-leaf has exactly 2 children, and vertices are coloured red or black.

Exception: A black node with a single red leaf child is OK.

- Every root-leaf path has the same number of black vertices.
- No red vertex has a red child.



We can turn a 2-3-4 tree into a red-black tree (or vice versa) by:

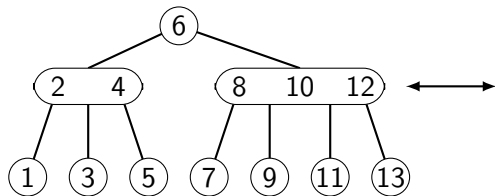
Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement... But they are secretly the same thing!

- Red-black trees are binary search trees where every non-leaf has exactly 2 children, and vertices are coloured red or black.

Exception: A black node with a single red leaf child is OK.

- Every root-leaf path has the same number of black vertices.
- No red vertex has a red child.



We can turn a 2-3-4 tree into a red-black tree (or vice versa) by: replacing each 2-node by a black node;

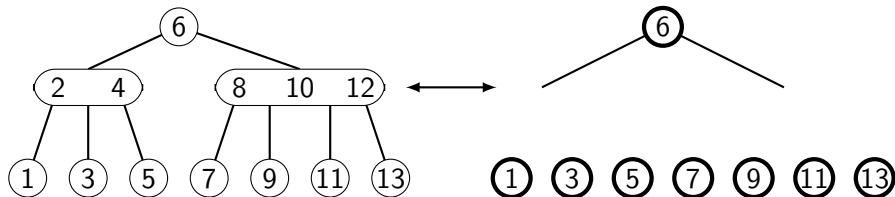
Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement... But they are secretly the same thing!

- Red-black trees are binary search trees where every non-leaf has exactly 2 children, and vertices are coloured red or black.

Exception: A black node with a single red leaf child is OK.

- Every root-leaf path has the same number of black vertices.
- No red vertex has a red child.



We can turn a 2-3-4 tree into a red-black tree (or vice versa) by: replacing each 2-node by a black node;

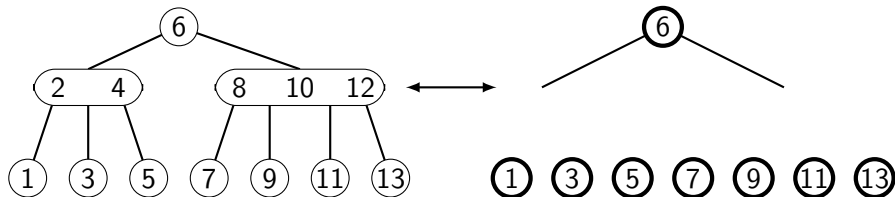
Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement... But they are secretly the same thing!

- Red-black trees are binary search trees where every non-leaf has exactly 2 children, and vertices are coloured red or black.

Exception: A black node with a single red leaf child is OK.

- Every root-leaf path has the same number of black vertices.
- No red vertex has a red child.



We can turn a 2-3-4 tree into a red-black tree (or vice versa) by:
replacing each 2-node by a black node; each 4-node by a black node with two red children;

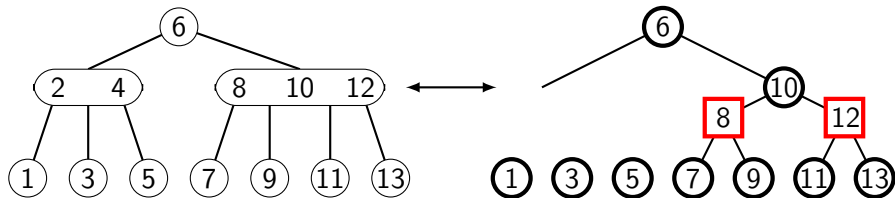
Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement... But they are secretly the same thing!

- Red-black trees are binary search trees where every non-leaf has exactly 2 children, and vertices are coloured red or black.

Exception: A black node with a single red leaf child is OK.

- Every root-leaf path has the same number of black vertices.
- No red vertex has a red child.



We can turn a 2-3-4 tree into a red-black tree (or vice versa) by:
replacing each 2-node by a black node; each 4-node by a black node with two red children;

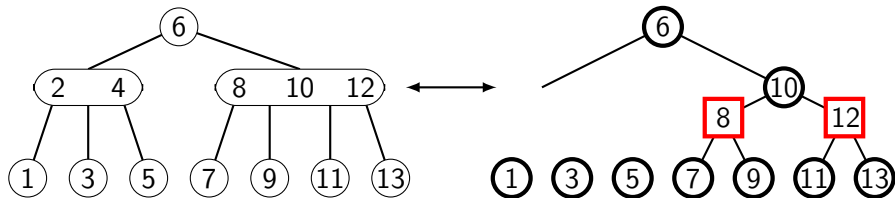
Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement... But they are secretly the same thing!

- Red-black trees are binary search trees where every non-leaf has exactly 2 children, and vertices are coloured red or black.

Exception: A black node with a single red leaf child is OK.

- Every root-leaf path has the same number of black vertices.
- No red vertex has a red child.



We can turn a 2-3-4 tree into a red-black tree (or vice versa) by:
replacing each 2-node by a black node; each 4-node by a black node with two red children; and each 3-node by a black node with one red child.

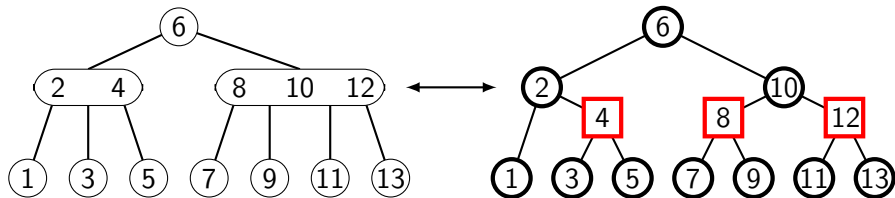
Red-black trees

Red-black trees are often used over 2-3-4 trees in practice, because they are slightly faster to implement... But they are secretly the same thing!

- Red-black trees are binary search trees where every non-leaf has exactly 2 children, and vertices are coloured red or black.

Exception: A black node with a single red leaf child is OK.

- Every root-leaf path has the same number of black vertices.
- No red vertex has a red child.



We can turn a 2-3-4 tree into a red-black tree (or vice versa) by:
replacing each 2-node by a black node; each 4-node by a black node with two red children; and each 3-node by a black node with one red child.