

2-3-4 trees I: Search and insertion

COMS20017 (Algorithms and Data)

John Lapinskas, University of Bristol

The limitations of binary search

From Algorithms I: If we have an n -element sorted array, we can search for a given value in $O(\log n)$ time with binary search.

Find 7:

↓ 7 < 10: Check left half subarray

4	5	8	10	13	18	21
---	---	---	----	----	----	----

The limitations of binary search

From Algorithms I: If we have an n -element sorted array, we can search for a given value in $O(\log n)$ time with binary search.

Find 7:

↓ 7 > 5: Check right quarter subarray

4	5	8	10	13	18	21
---	---	---	----	----	----	----

The limitations of binary search

From Algorithms I: If we have an n -element sorted array, we can search for a given value in $O(\log n)$ time with binary search.

Find 7:

↓ $7 \neq 8$: Return Not found

4	5	8	10	13	18	21
---	---	---	----	----	----	----

Why can't we use this to implement a dictionary, storing an array of key-value pairs sorted by key?

Because we can't easily insert or remove things from the middle of the array — this takes $\Omega(n)$ time! And if we used a linked list instead... it would take $\Omega(n)$ time to find the halfway point.

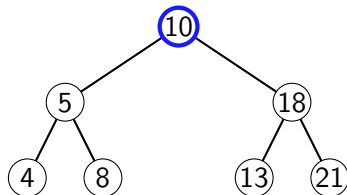
Instead, we can use a **binary search tree**.

Binary search trees

Idea: Each node has 0–2 children. If a node's value is x , then **all** its left descendants' values are $< x$, and **all** its right descendants' values are $> x$.

(For simplicity, we assume all values are distinct.)

Find 7;



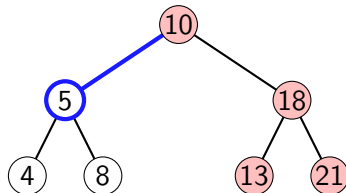
Then we can still find nodes by binary search, but we can also insert and delete them in $O(d)$ time where d is the depth of the tree.

Binary search trees

Idea: Each node has 0–2 children. If a node's value is x , then **all** its left descendants' values are $< x$, and **all** its right descendants' values are $> x$.

(For simplicity, we assume all values are distinct.)

Find 7;



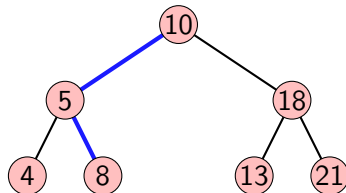
Then we can still find nodes by binary search, but we can also insert and delete them in $O(d)$ time where d is the depth of the tree.

Binary search trees

Idea: Each node has 0–2 children. If a node's value is x , then **all** its left descendants' values are $< x$, and **all** its right descendants' values are $> x$.

(For simplicity, we assume all values are distinct.)

Find 7;
Not found.



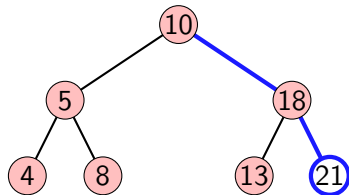
Then we can still find nodes by binary search, but we can also insert and delete them in $O(d)$ time where d is the depth of the tree.

Binary search trees

Idea: Each node has 0–2 children. If a node's value is x , then **all** its left descendants' values are $< x$, and **all** its right descendants' values are $> x$.

(For simplicity, we assume all values are distinct.)

Delete 21;



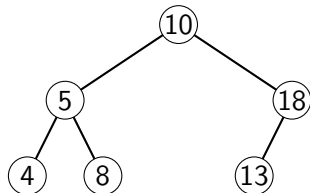
Then we can still find nodes by binary search, but we can also insert and delete them in $O(d)$ time where d is the depth of the tree.

Binary search trees

Idea: Each node has 0–2 children. If a node's value is x , then **all** its left descendants' values are $< x$, and **all** its right descendants' values are $> x$.

(For simplicity, we assume all values are distinct.)

Insert 30;



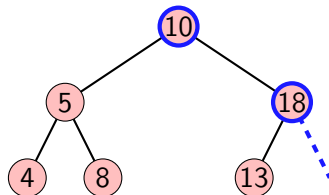
Then we can still find nodes by binary search, but we can also insert and delete them in $O(d)$ time where d is the depth of the tree.

Binary search trees

Idea: Each node has 0–2 children. If a node's value is x , then **all** its left descendants' values are $< x$, and **all** its right descendants' values are $> x$.

(For simplicity, we assume all values are distinct.)

Insert 30;



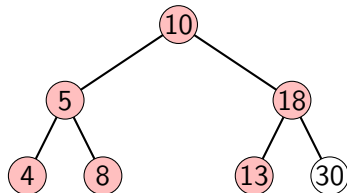
Then we can still find nodes by binary search, but we can also insert and delete them in $O(d)$ time where d is the depth of the tree.

Binary search trees

Idea: Each node has 0–2 children. If a node's value is x , then **all** its left descendants' values are $< x$, and **all** its right descendants' values are $> x$.

(For simplicity, we assume all values are distinct.)

Insert 30;



Then we can still find nodes by binary search, but we can also insert and delete them in $O(d)$ time where d is the depth of the tree.

Binary search trees

Idea: Each node has 0–2 children. If a node's value is x , then **all** its left descendants' values are $< x$, and **all** its right descendants' values are $> x$.

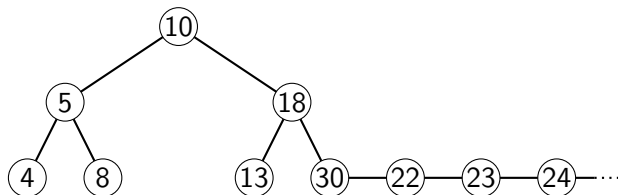
(For simplicity, we assume all values are distinct.)

Insert 22;

Insert 23;

Insert 24;

⋮



Then we can still find nodes by binary search, but we can also insert and delete them in $O(d)$ time where d is the depth of the tree.

Ideally, if the tree has n elements, then all but the bottom layer is full — the tree is **balanced**, as above. In that case,

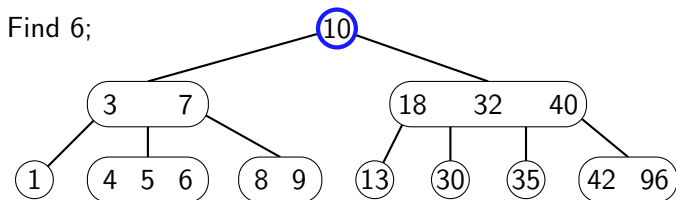
$$n \approx 2^d + 2^{d-1} + \dots + 1 = 2^{d+1} - 1 \Rightarrow d \in \Theta(\log n).$$

Problem: What if that doesn't happen? We could get $d \in \Omega(n)$...

2-3-4 trees

Idea: Force the tree to be **perfectly** balanced, with all levels full. To make this possible to maintain, allow nodes to contain more than one value.

A **k-node** can have up to k children and contain $k - 1$ values (so a binary search tree is made entirely of 2-nodes). We will allow $k \in \{2, 3, 4\}$.



Say a 3-node has values $x_1 \leq x_2$, and children c_1 , c_2 and c_3 .

Then all descendants of c_1 must have values at most x_1 ...

All descendants of c_2 must have values greater than x_1 and less than x_2 ...

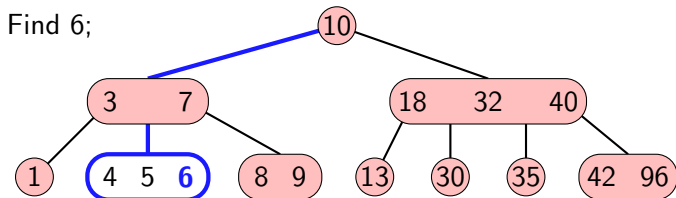
And all descendants of c_3 must have values greater than x_3 .

4-nodes work the same way. So we can still find a value in $O(d)$ time.

2-3-4 trees

Idea: Force the tree to be **perfectly** balanced, with all levels full. To make this possible to maintain, allow nodes to contain more than one value.

A **k -node** can have up to k children and contain $k - 1$ values (so a binary search tree is made entirely of 2-nodes). We will allow $k \in \{2, 3, 4\}$.



Say a 3-node has values $x_1 \leq x_2$, and children c_1 , c_2 and c_3 .

Then all descendants of c_1 must have values at most x_1 ...

All descendants of c_2 must have values greater than x_1 and less than x_2 ...

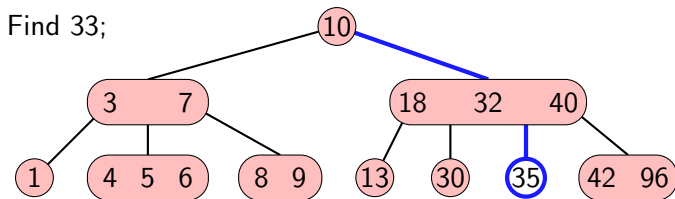
And all descendants of c_3 must have values greater than x_3 .

4-nodes work the same way. So we can still find a value in $O(d)$ time.

2-3-4 trees

Idea: Force the tree to be **perfectly** balanced, with all levels full. To make this possible to maintain, allow nodes to contain more than one value.

A **k -node** can have up to k children and contain $k - 1$ values (so a binary search tree is made entirely of 2-nodes). We will allow $k \in \{2, 3, 4\}$.



Say a 3-node has values $x_1 \leq x_2$, and children c_1 , c_2 and c_3 .

Then all descendants of c_1 must have values at most x_1 ...

All descendants of c_2 must have values greater than x_1 and less than x_2 ...

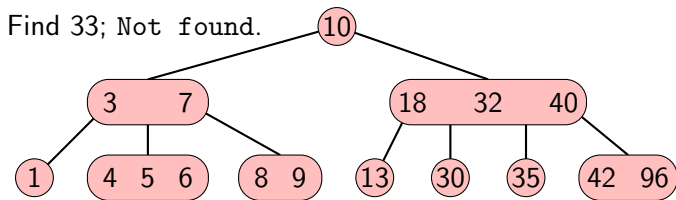
And all descendants of c_3 must have values greater than x_3 .

4-nodes work the same way. So we can still find a value in $O(d)$ time.

2-3-4 trees

Idea: Force the tree to be **perfectly** balanced, with all levels full. To make this possible to maintain, allow nodes to contain more than one value.

A **k-node** can have up to k children and contain $k - 1$ values (so a binary search tree is made entirely of 2-nodes). We will allow $k \in \{2, 3, 4\}$.



Say a 3-node has values $x_1 \leq x_2$, and children c_1 , c_2 and c_3 .

Then all descendants of c_1 must have values at most x_1 ...

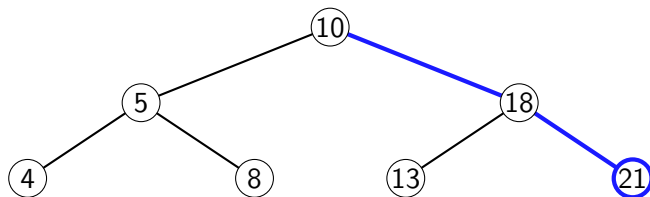
All descendants of c_2 must have values greater than x_1 and less than x_2 ...

And all descendants of c_3 must have values greater than x_2 .

4-nodes work the same way. So we can still find a value in $O(d)$ time.

Inserting new values

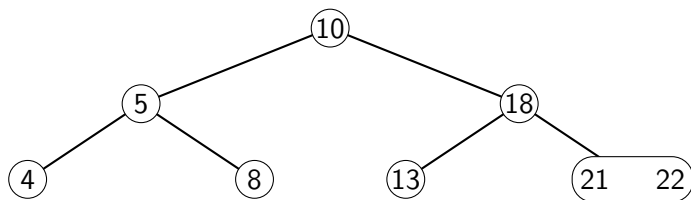
Insert 22;



To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

Inserting new values

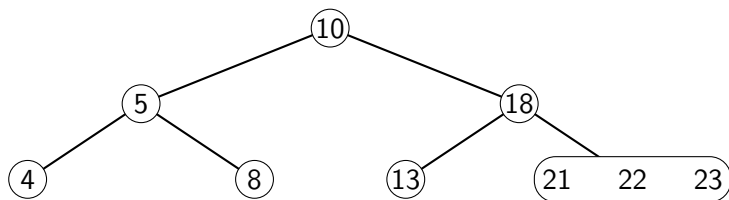
Insert 22;



To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

Inserting new values

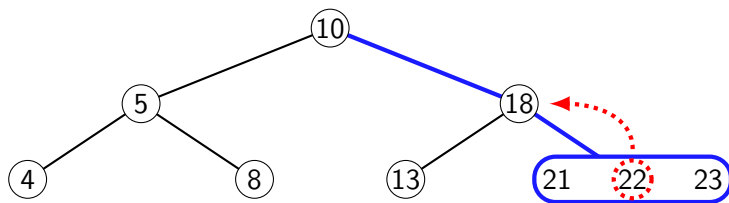
Insert 23;



To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

Inserting new values

Insert 24;

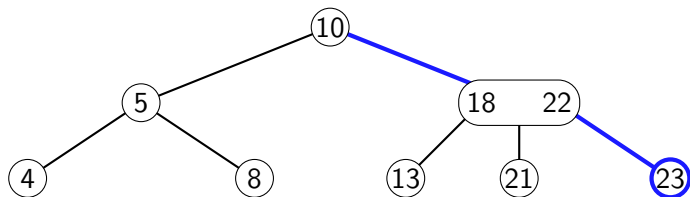


To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

Inserting new values

Insert 24;

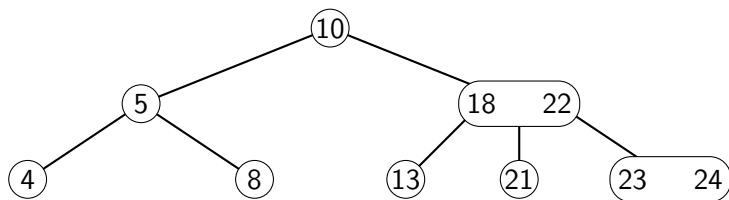


To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

Inserting new values

Insert 24;

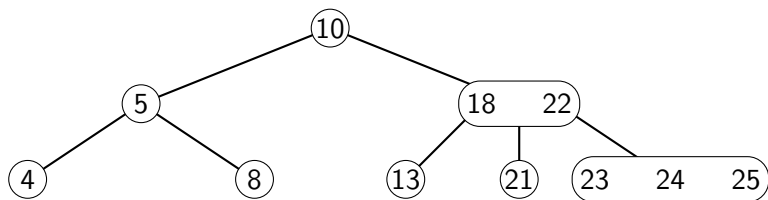


To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

Inserting new values

Insert 25;

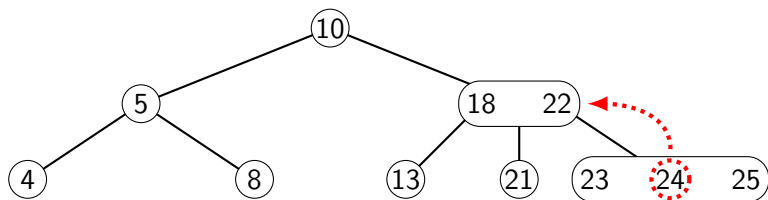


To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

Inserting new values

Insert 26;

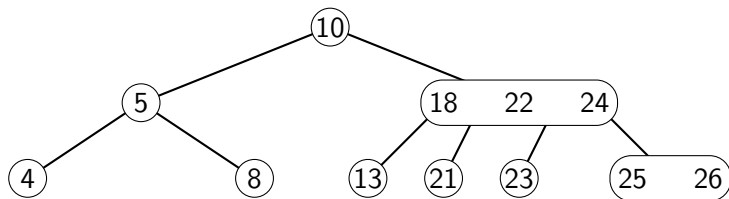


To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

Inserting new values

Insert 26;

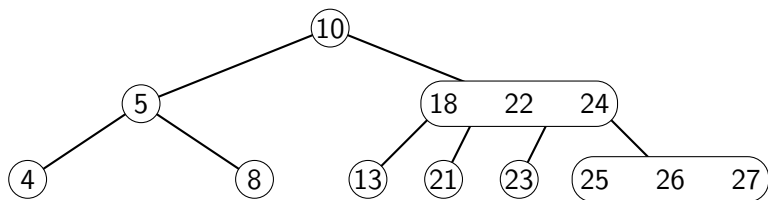


To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

Inserting new values

Insert 27;

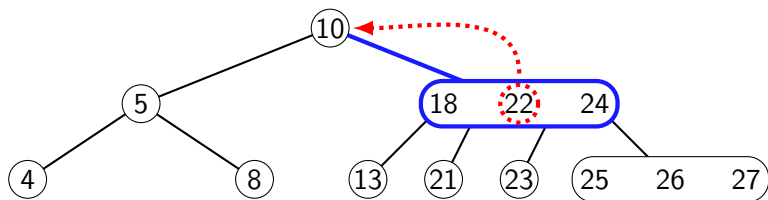


To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

Inserting new values

Insert 28;



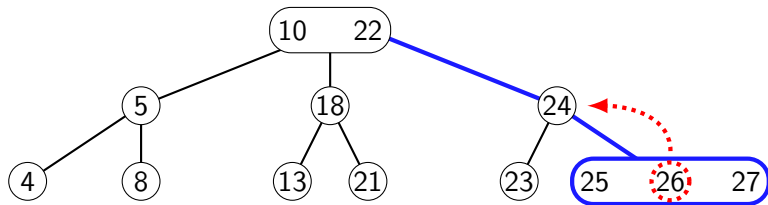
To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

Inserting new values

Insert 28;



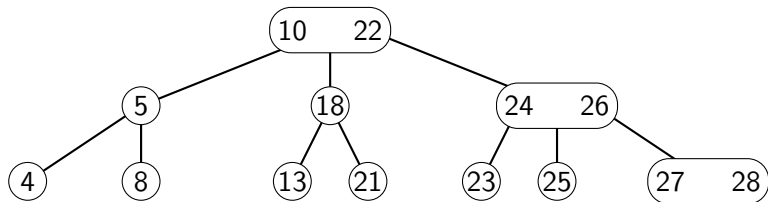
To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

Inserting new values

Insert 28;



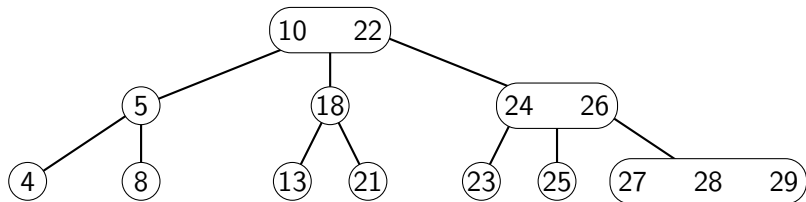
To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

Inserting new values

Insert 29;



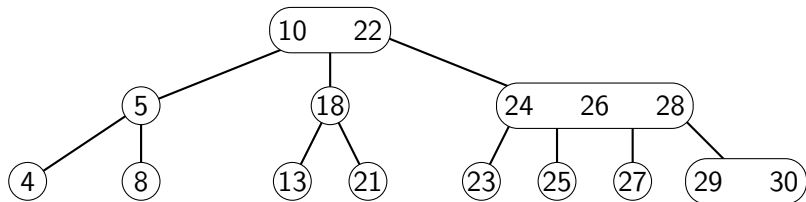
To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

Inserting new values

Insert 30;



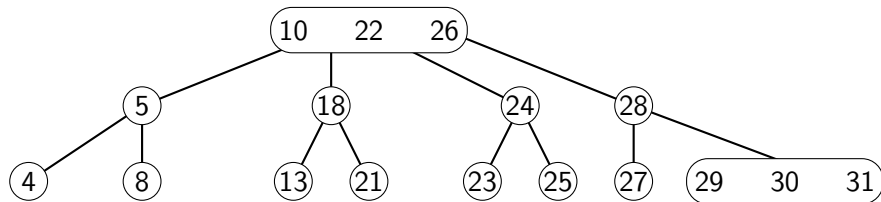
To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

Inserting new values

Insert 31;



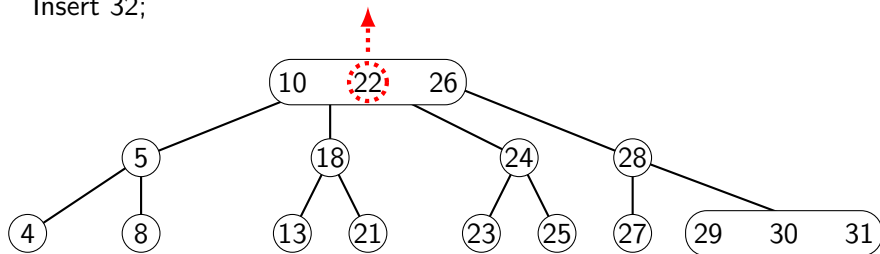
To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

Inserting new values

Insert 32;



To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

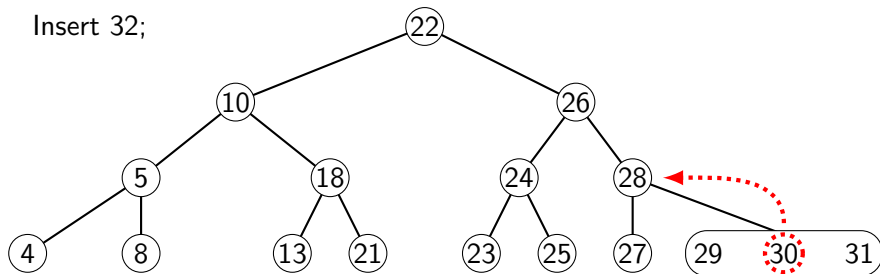
If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

If we have to split the root, d increases by 1.

Inserting new values

Insert 32;



To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

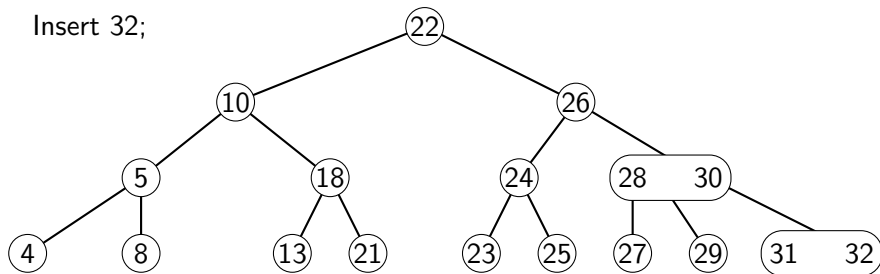
If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

If we have to split the root, d increases by 1.

Inserting new values

Insert 32;



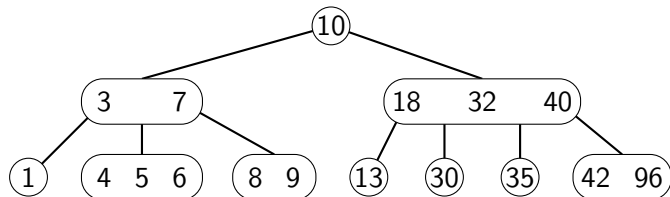
To insert a value k , first we find the leaf that would contain it if it was there. If it's a 2-node or a 3-node, we can just add the new value.

If it's a 4-node, we first **split** it, sending one value up to its parent and keeping the others as 2-nodes.

If its parent is a 4-node as well, we're in trouble... so we split all 4-nodes we find on the way down. Still only takes $O(d)$ time.

If we have to split the root, d increases by 1. But balance is maintained!

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 a value v : Next time!