Data Stuctures

Textbook Reading

Data Structures Lecture Notes

Overview

"Data structuring":

Effectively use data structures to implement non-trivial steps in algorithms

Augmenting data structures:

Add information to existing data structures so they support additional queries

Data structures:

- (a, b)-trees
- Rank-select trees
- Priority search trees
- Range trees

Problems:

- (Orthogonal) line segment intersection reporting and counting
- Range reporting and counting

The Dictionary ADT

A data structure D that stores a set S of key-value pairs and supports three operations:

Insert(D, k, v)	Insert the key-value pair (k, v) into S
Delete(D, k)	Delete the key-value pair with key k from S
Find(D, k)	Report the key-value pair with key k or nil if there is none

Ordered Dictionaries

If the keys come from an ordered set, the following additional operations are often useful:

RangeFind(D, ℓ, r)	Report all key-value pairs in S with keys in the interval $[\ell, r]$
Predecessor(D, k)	Report the key-value pair in S with largest key no greater than k
Successor(D, k)	Report the key-value pair in S with smallest key no less than k
Minimum(D)	Report the key-value pair with minimum key in S
Maximum(D)	Report the key-value pair with maximum key in S

Examples of Dictionaries

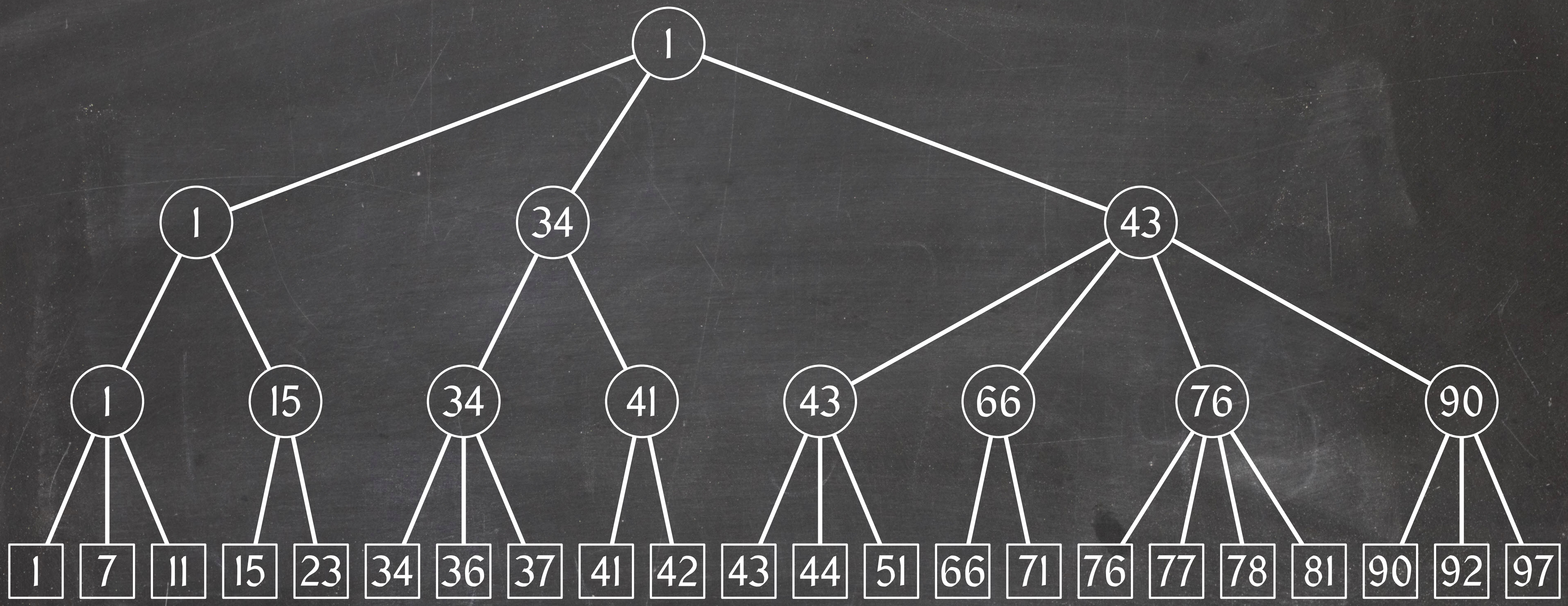
Simple dictionaries:

- (Sorted) arrays
- (Sorted) linked lists

Efficient dictionaries:

- Hash tables
- Balanced binary search trees (AVL, red-black trees, BB[α], AA, ...)
- (a, b)-Trees

(a, b)-Trees

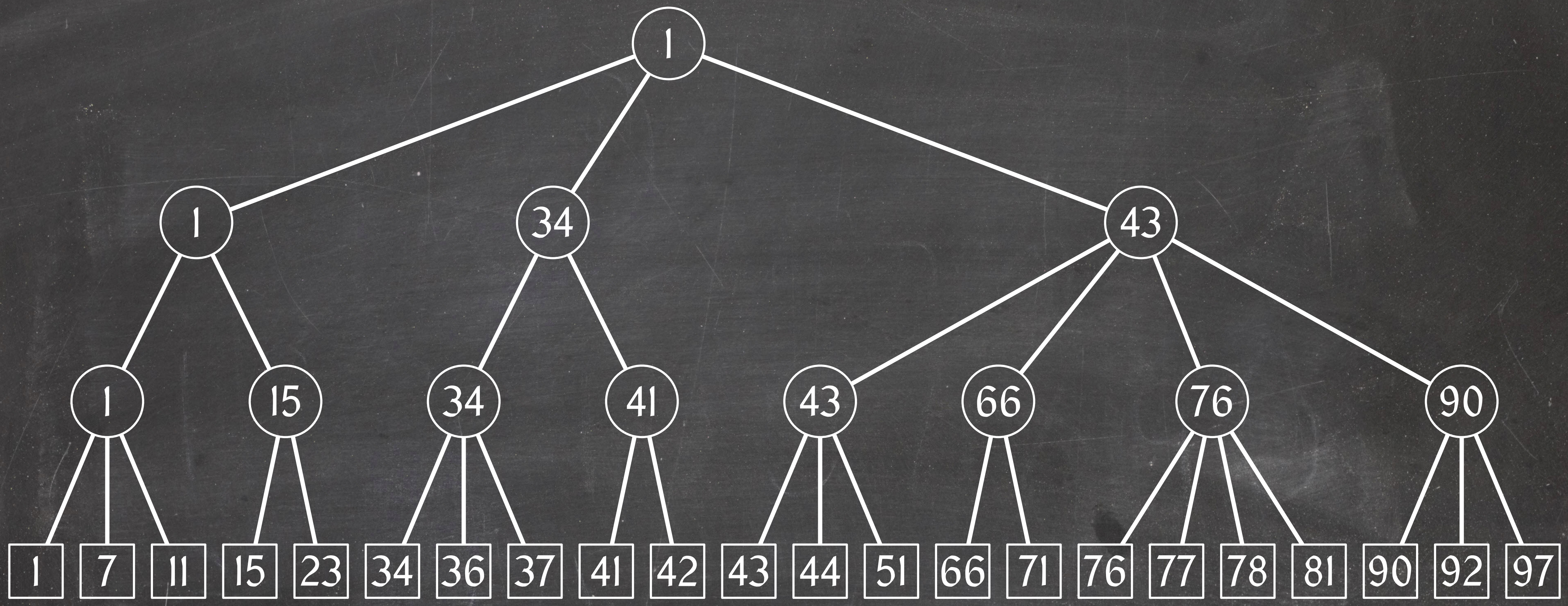


$$2 \leq a \text{ and } 2a - 1 \leq b$$

- All leaves are at the same depth.
- The root has between 2 and b children.
- Any other non-leaf node has between a and b children.

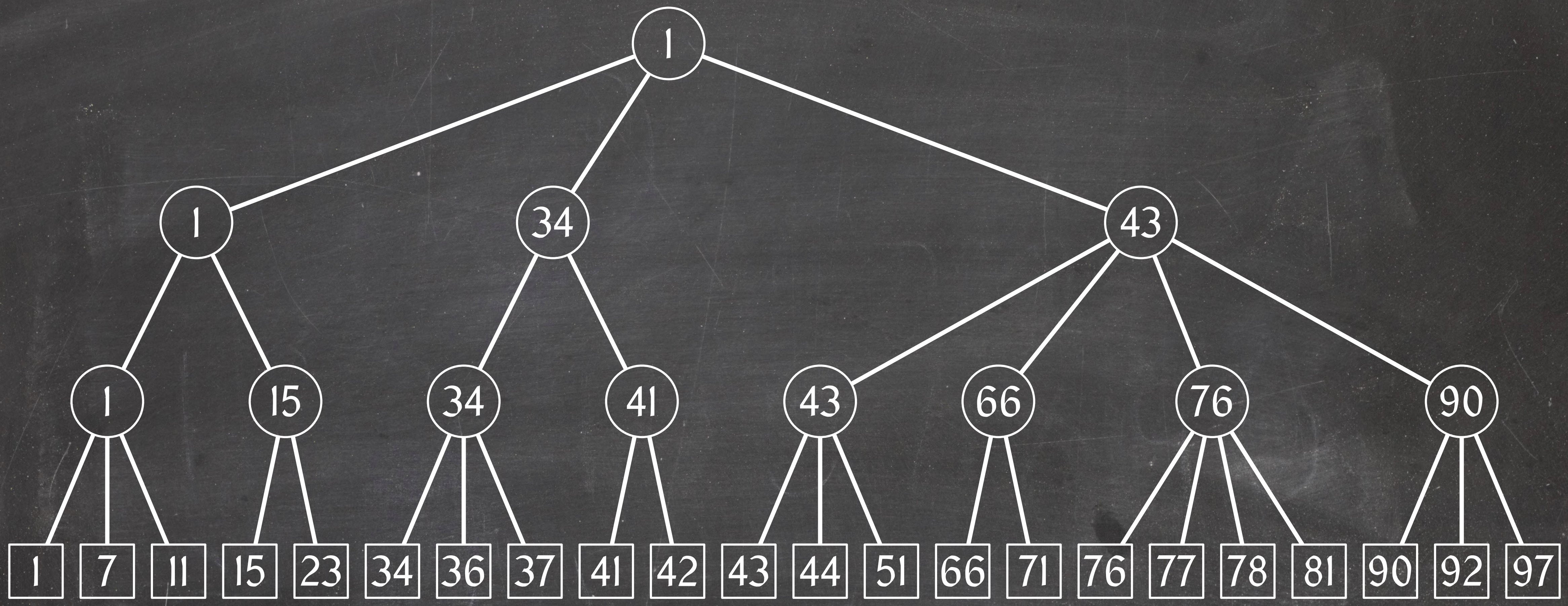
- Leaves store key-value pairs (data items) sorted by keys.
- Internal nodes store only keys.
- For a node v with children $w_1, w_2, \ldots, w_k$, $\text{key}(v) = \min_{1 \leq i \leq k} \text{key}(w_i)$.

Height of an (a, b)-Tree



Lemma: The height of an (a, b)-tree with n leaves is at most $1 + \log_a \frac{n}{2} \in O(\lg n)$.

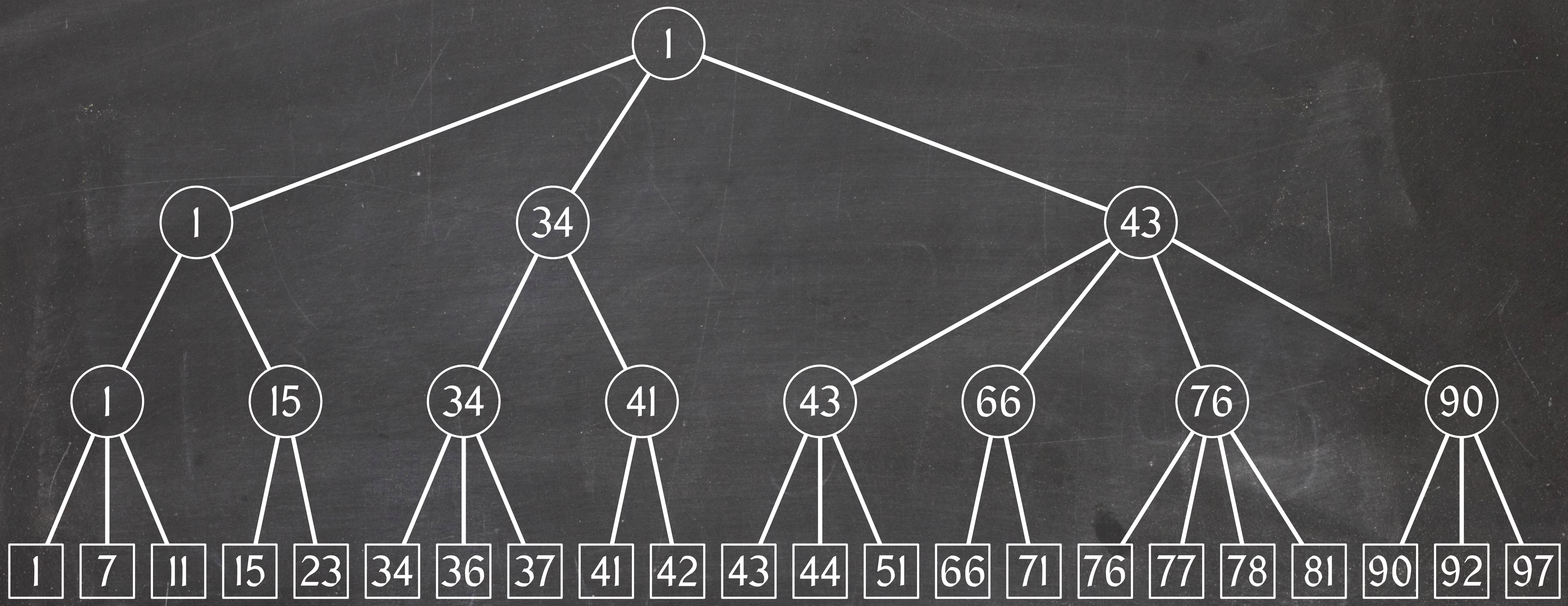
Height of an (a, b)-Tree



Lemma: The height of an (a, b)-tree with n leaves is at most $1 + \log_a \frac{n}{2} \in O(\lg n)$.

If the height is h, then the number of leaves is at least $2 \cdot a^{h-1}$.

Height of an (a, b)-Tree

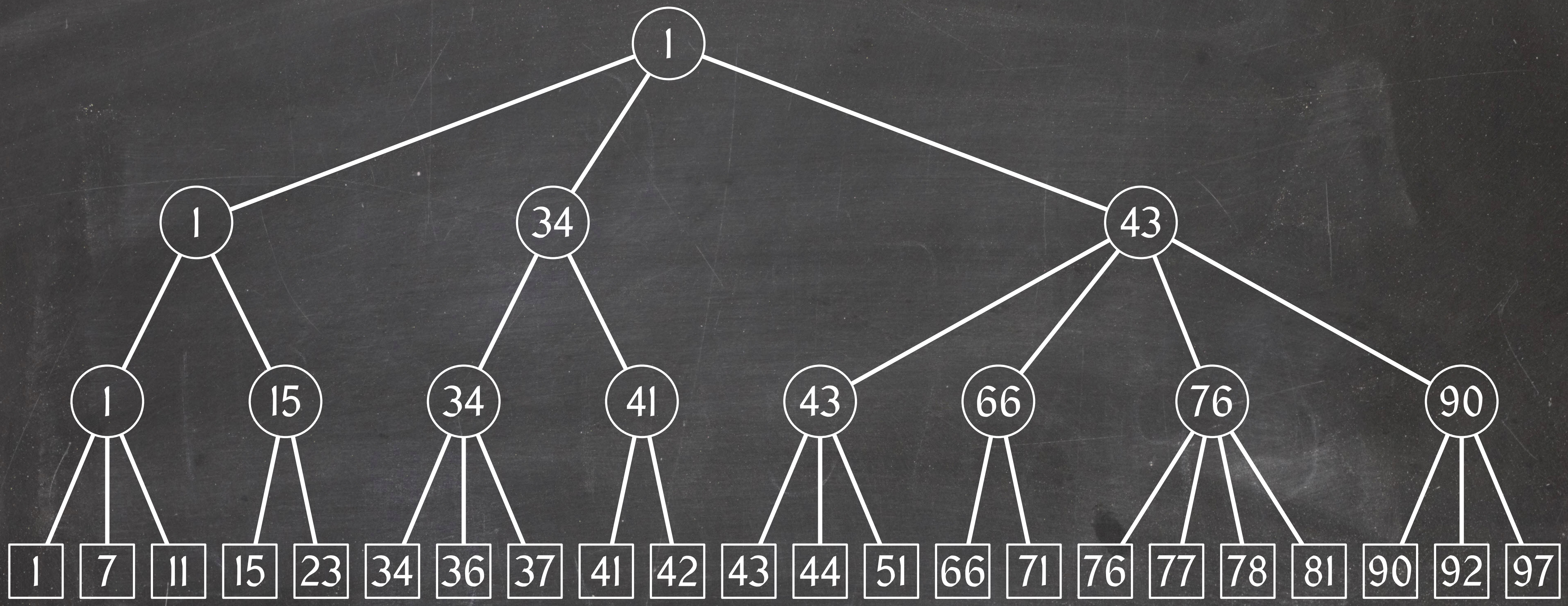


Lemma: The height of an (a, b)-tree with n leaves is at most $1 + \log_a \frac{n}{2} \in O(\lg n)$.

If the height is h, then the number of leaves is at least $2 \cdot a^{h-1}$.

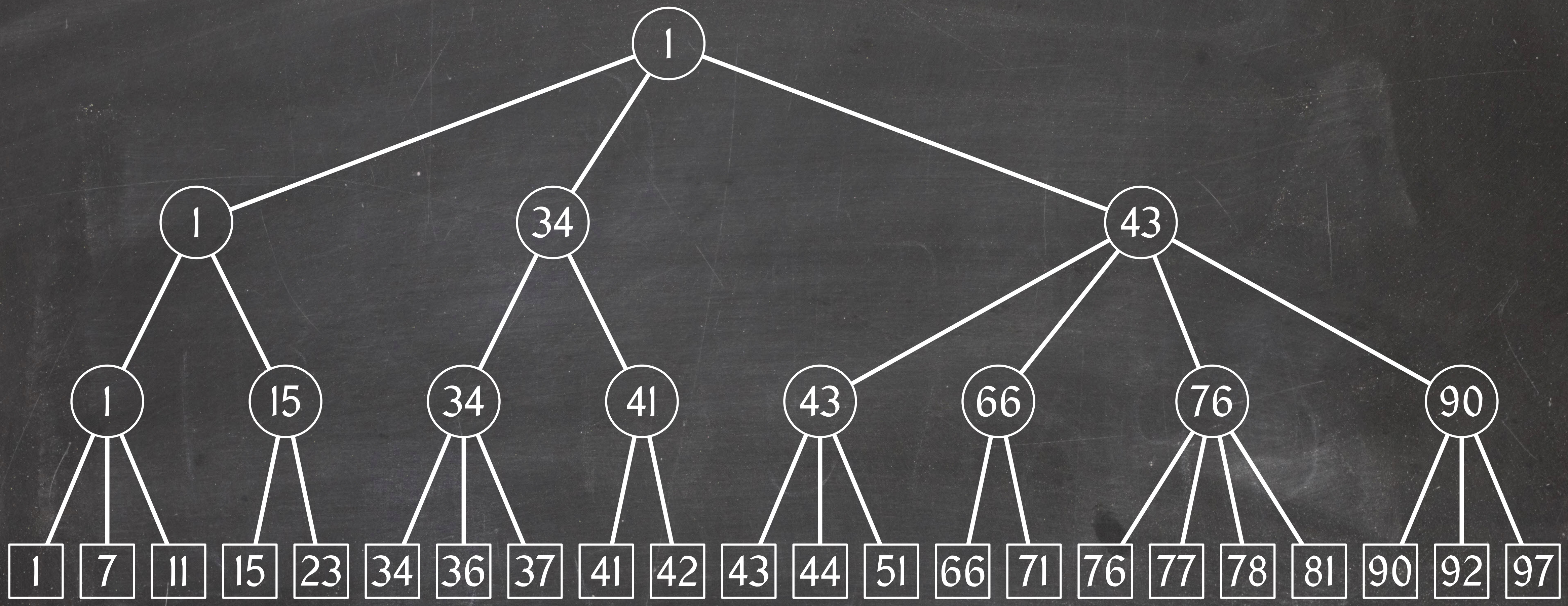
$$\Rightarrow \quad 2 \cdot a^{h-1} \le n \quad \Rightarrow \quad h \le 1 + \log_a \frac{n}{2}$$

Size of an (a, b)-Tree



Lemma: An (a, b)-tree with n leaves has less than 2n nodes.

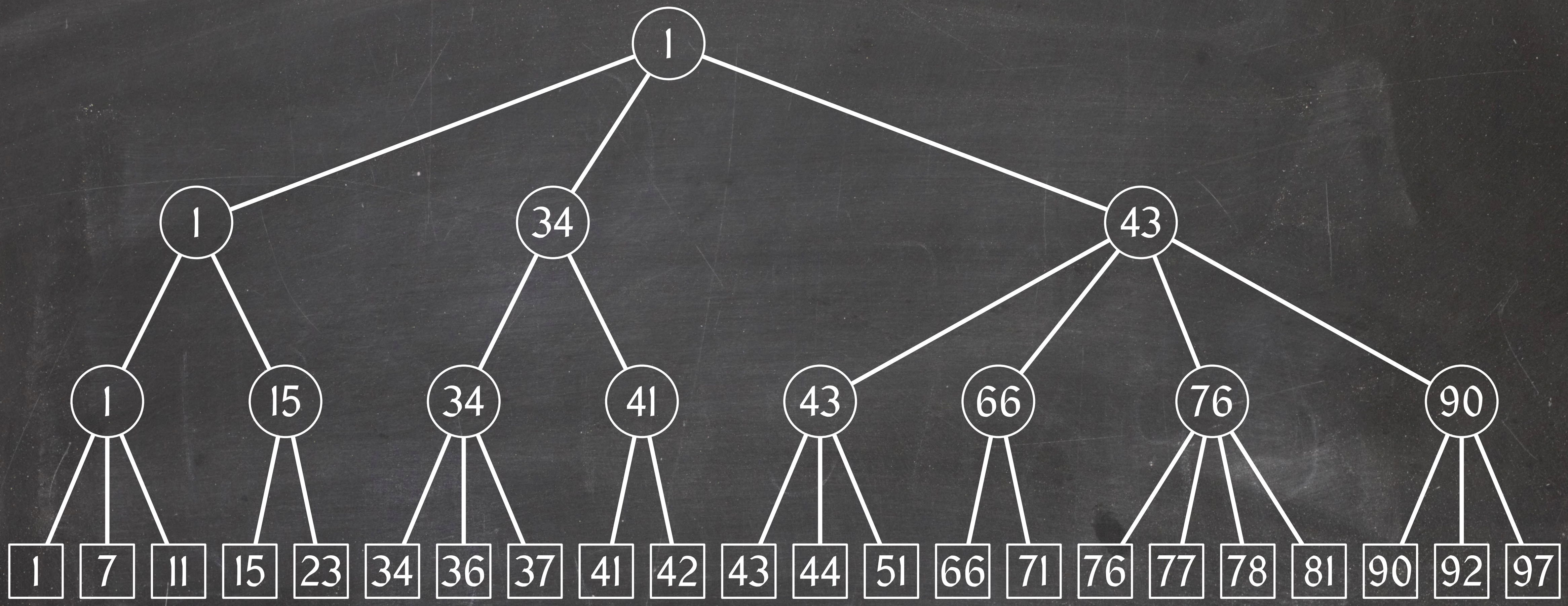
Size of an (a, b)-Tree



Lemma: An (a, b)-tree with n leaves has less than 2n nodes.

The number of nodes at height i above the leaves is at most $\frac{n}{a^i} \leq \frac{n}{2^i}$.

Size of an (a, b)-Tree



Lemma: An (a, b)-tree with n leaves has less than $2n$ nodes.

The number of nodes at height i above the leaves is at most $\frac{n}{a^i} \leq \frac{n}{2^i}$.

$$\sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

(a, b)-Tree Representation

Every node stores:

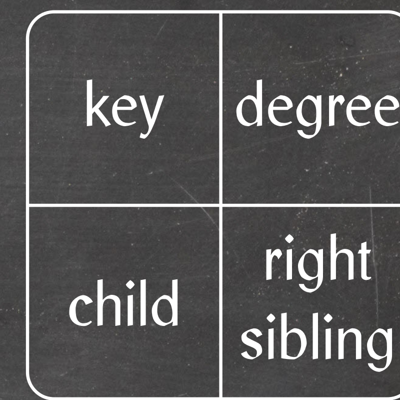
- Key-value pair (leaf) or key (internal node)
- Number of children
- Pointer to its leftmost child
- Pointer to its right sibling

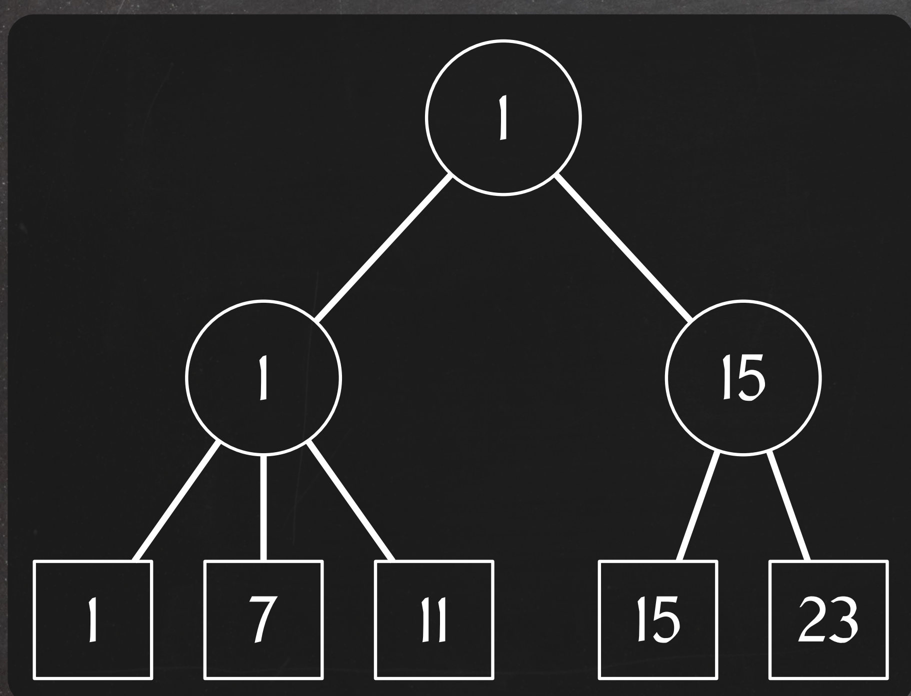
key	degree
child	right sibling

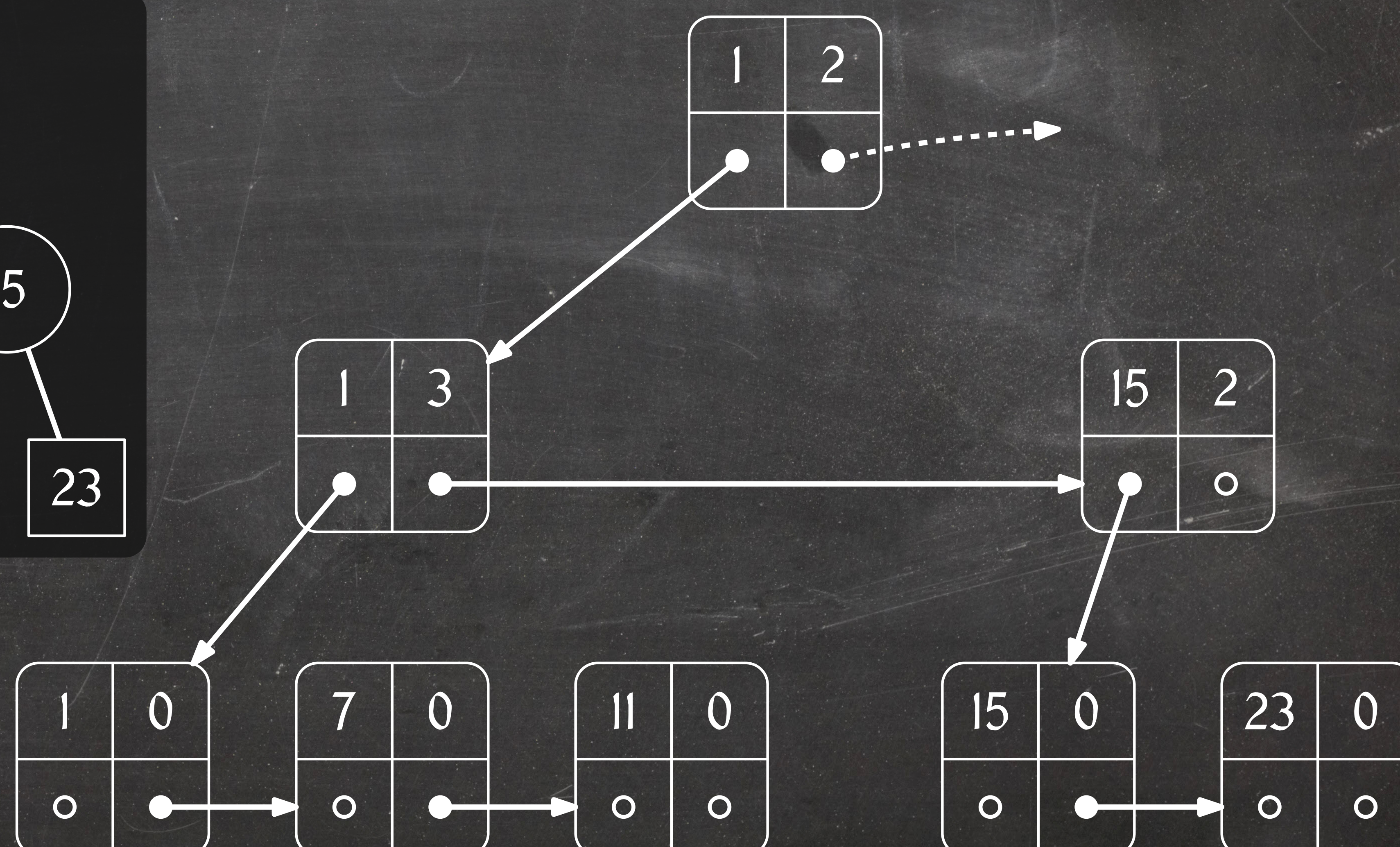
(a, b)-Tree Representation

Every node stores:

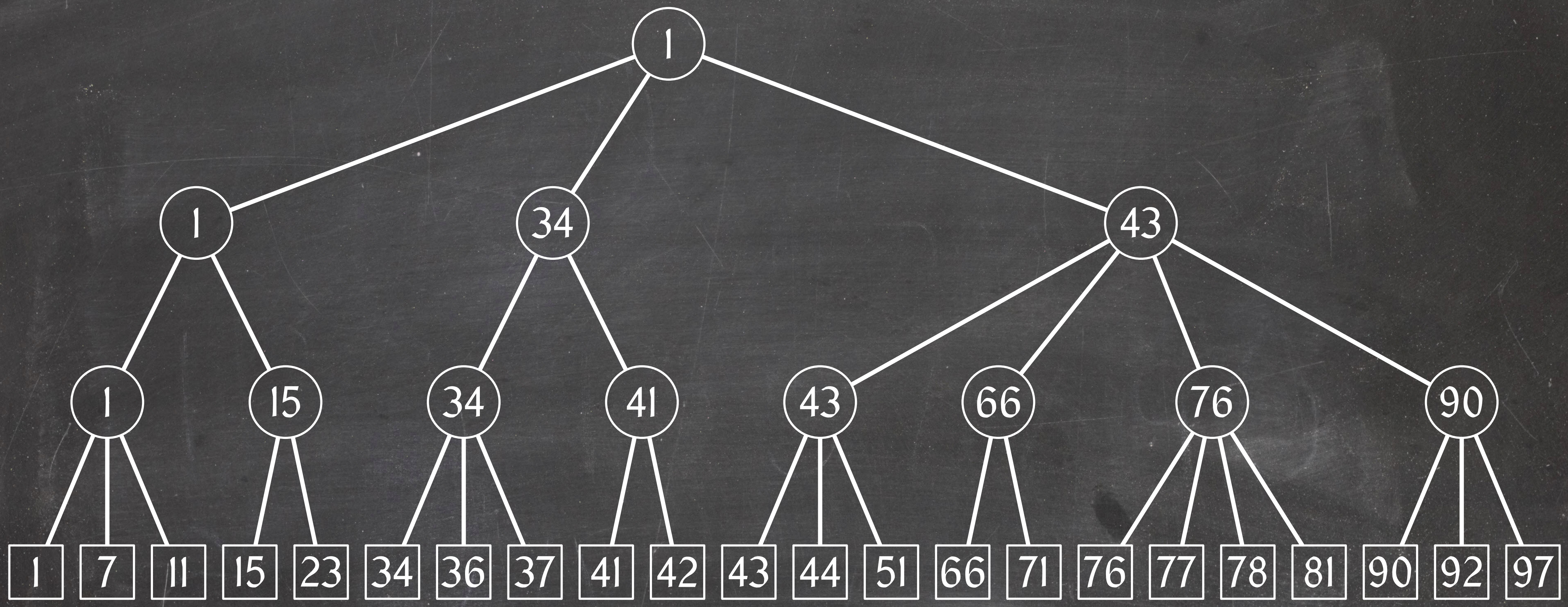
- Key-value pair (leaf) or key (internal node)
- Number of children
- Pointer to its leftmost child
- Pointer to its right sibling



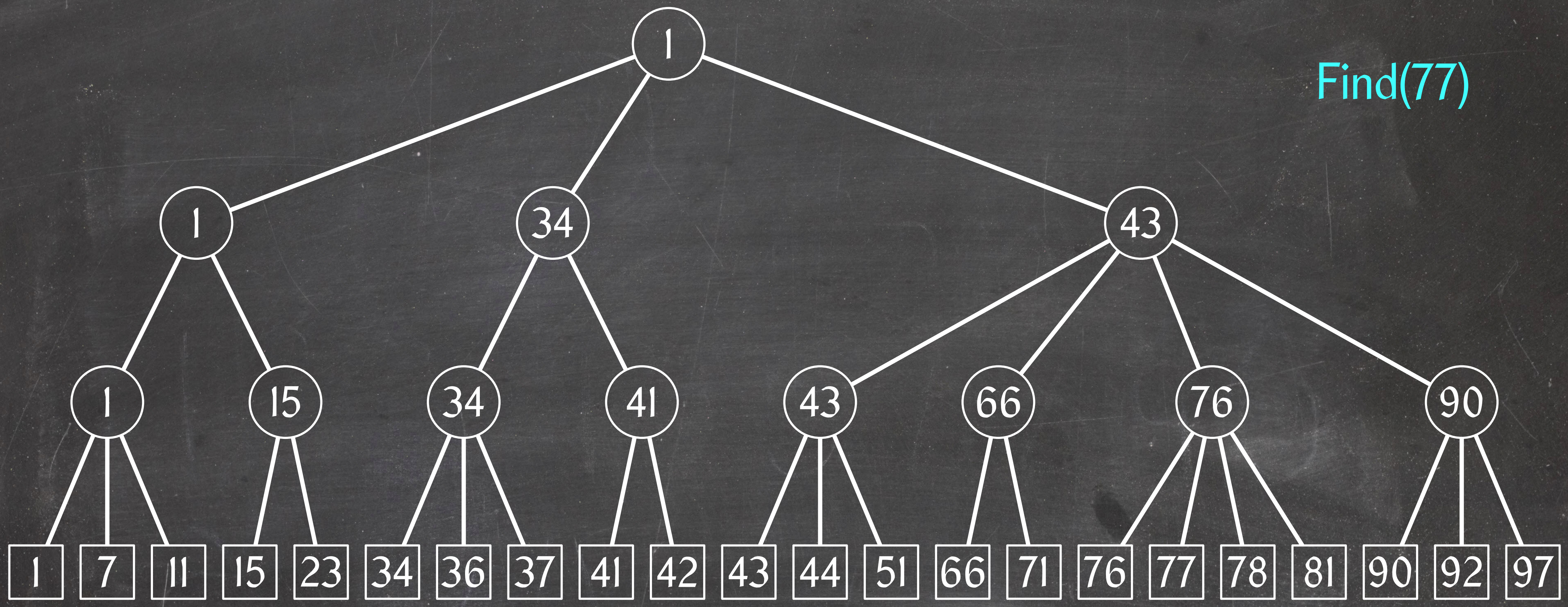




Find/Predecessor Operation

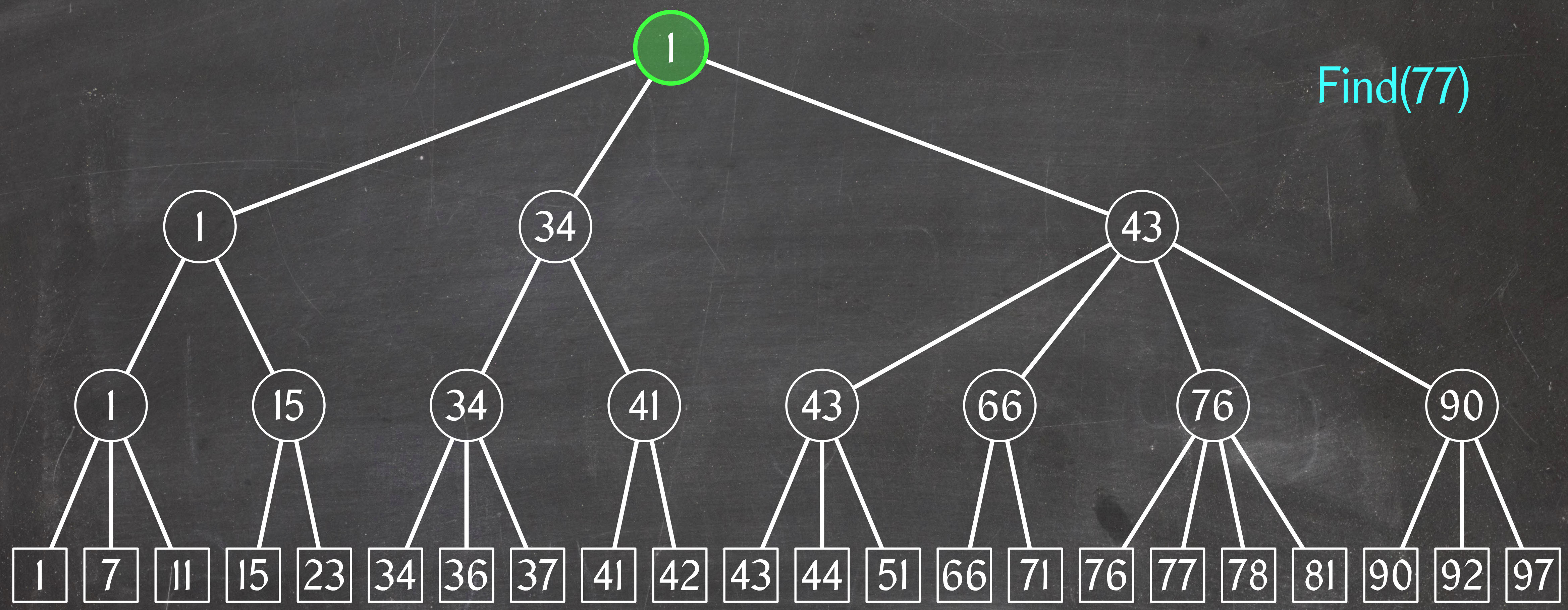


Find/Predecessor Operation



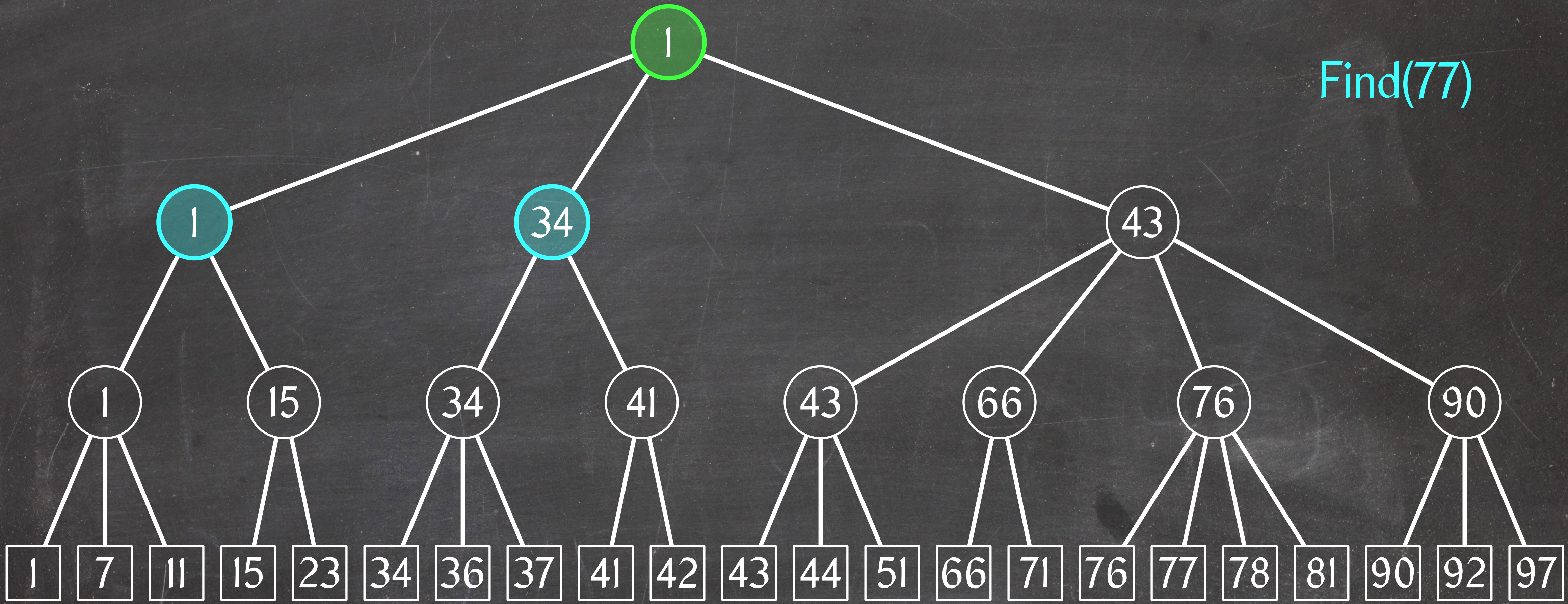
Find(77)

Find/Predecessor Operation



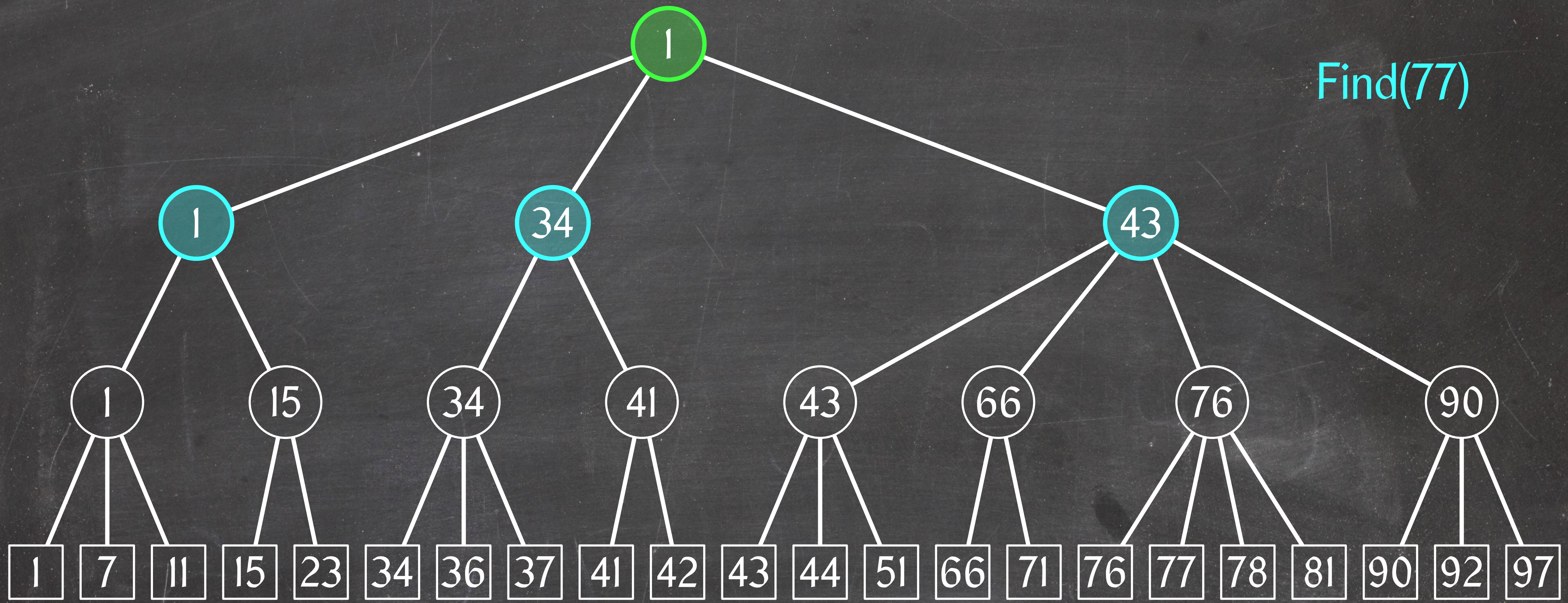
Find(77)

Find/Predecessor Operation



Find(77)

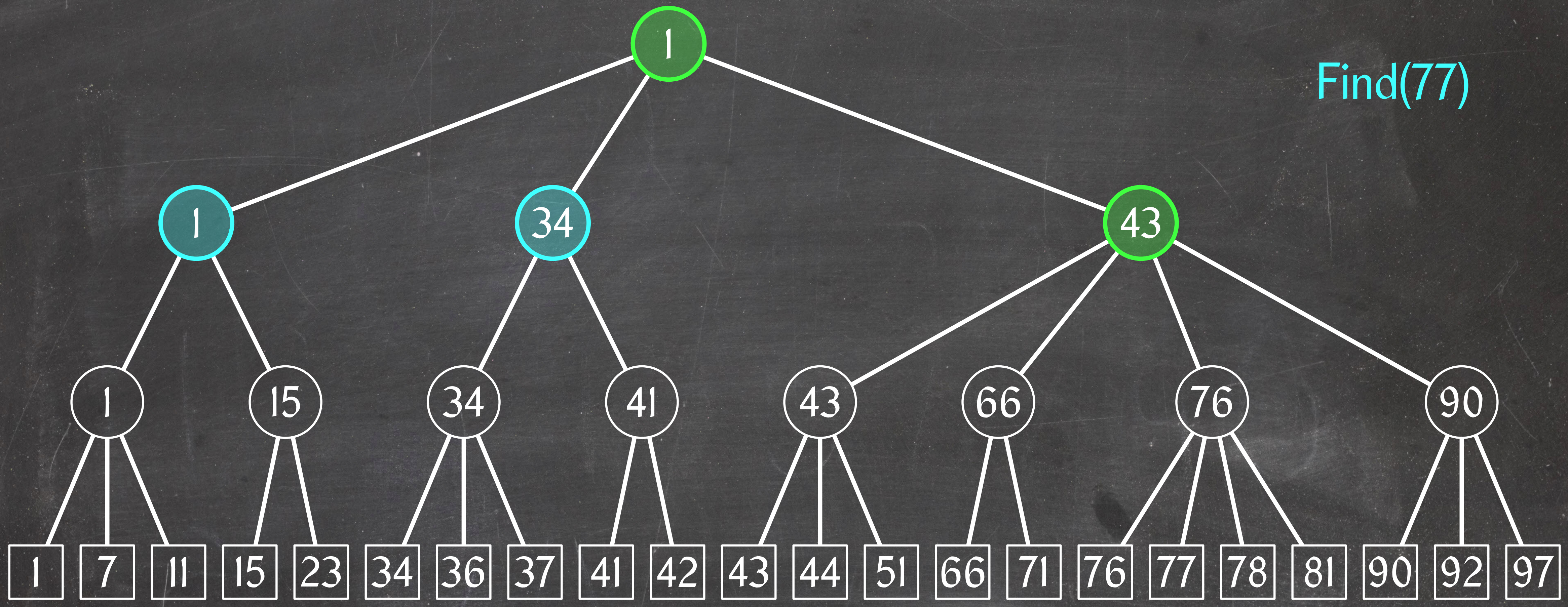
Find/Predecessor Operation



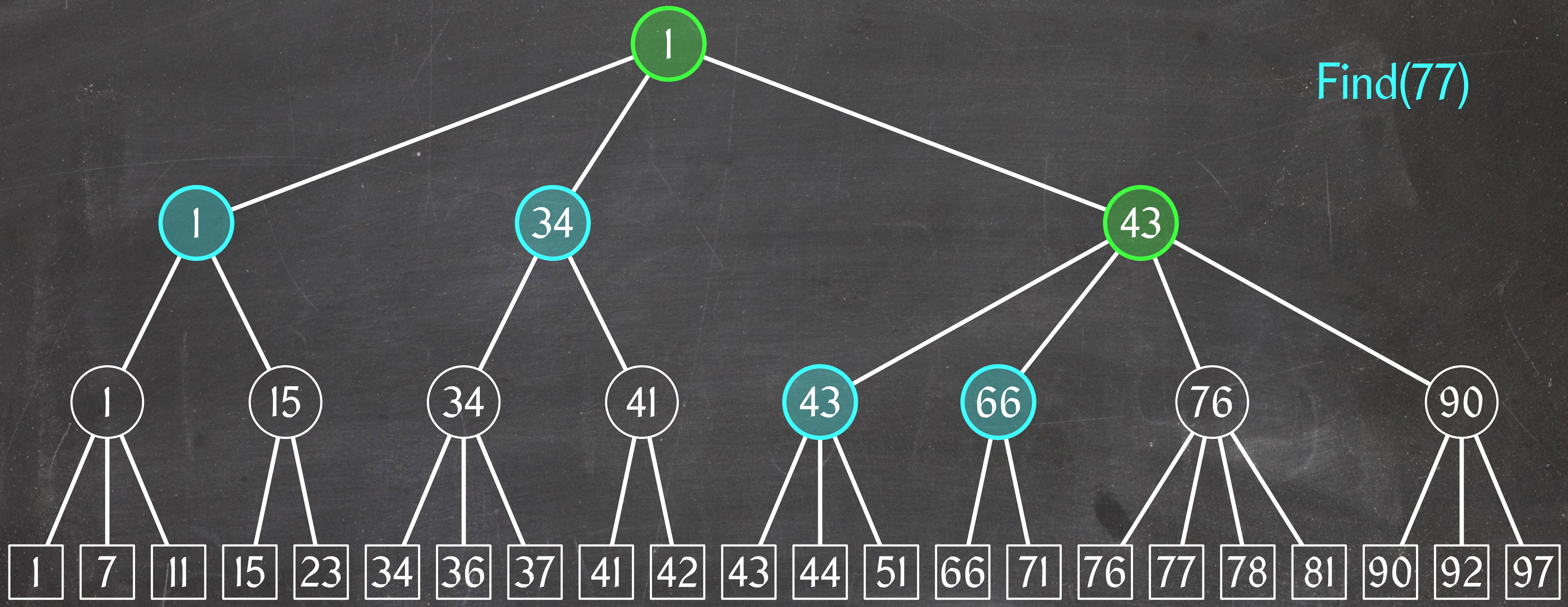
Find(77)

Find/Predecessor Operation

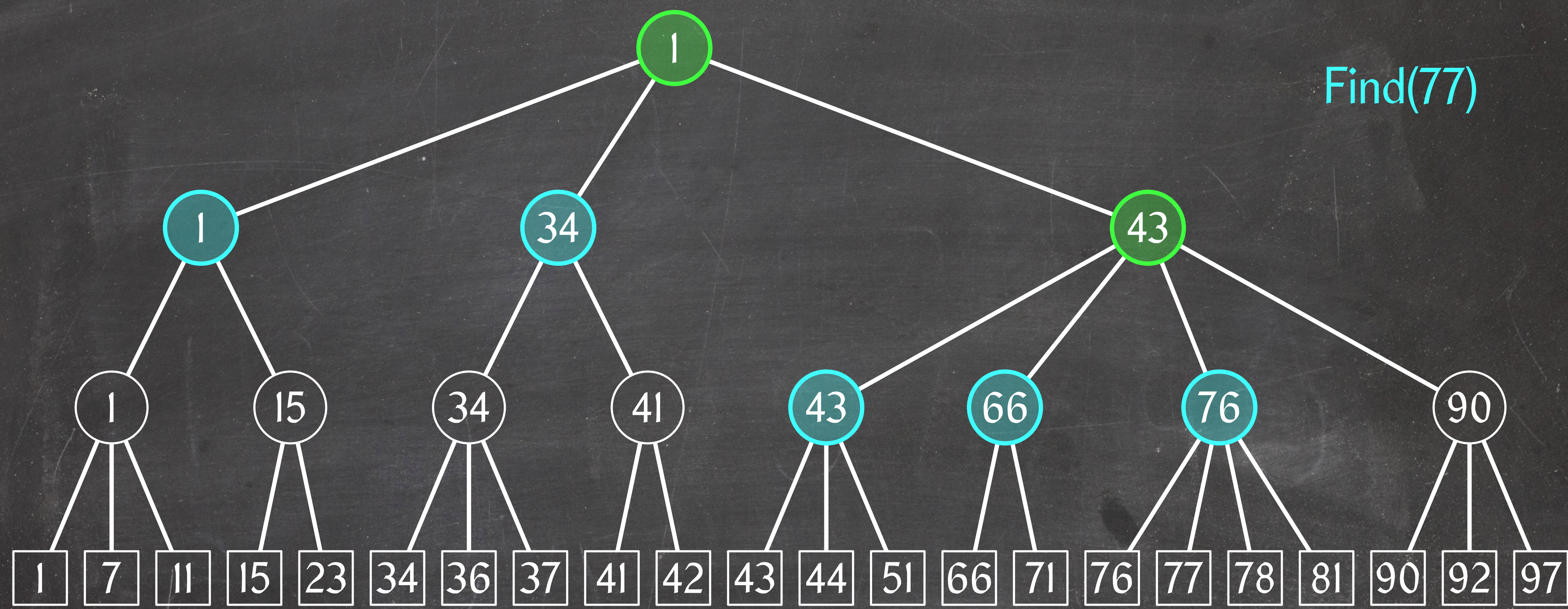
Find(77)



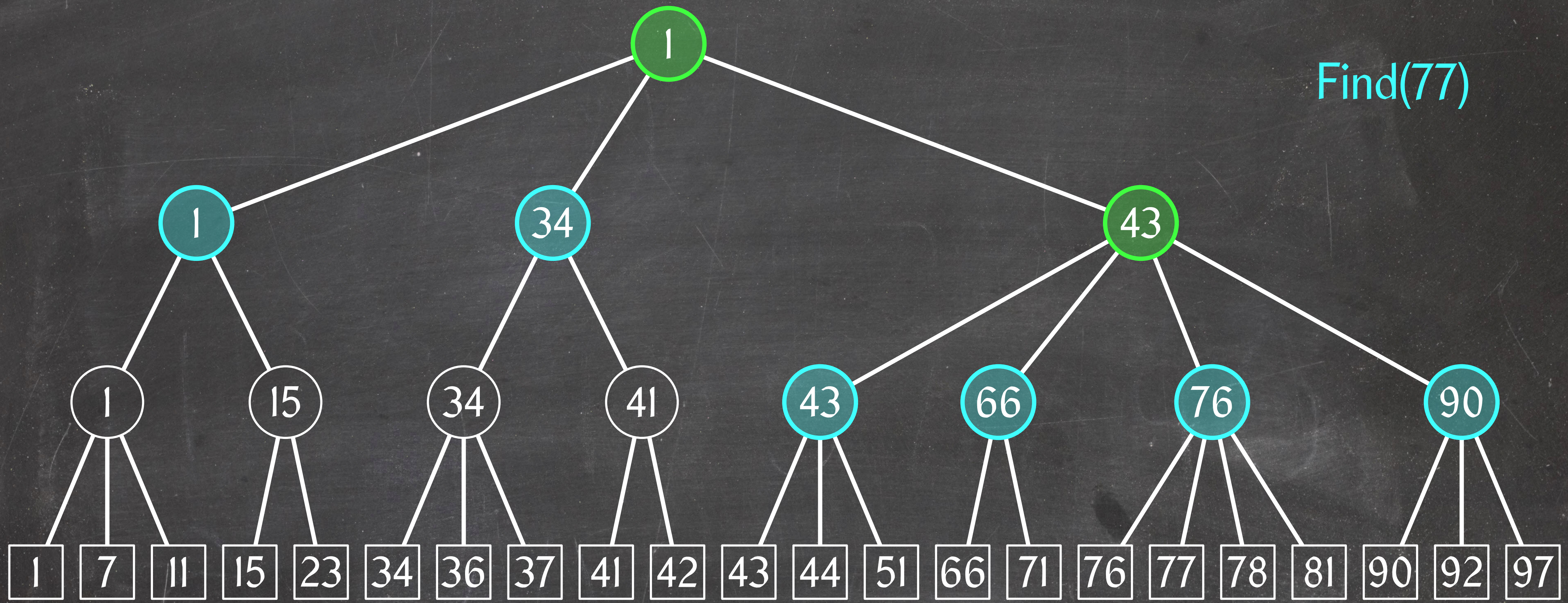
Find/Predecessor Operation



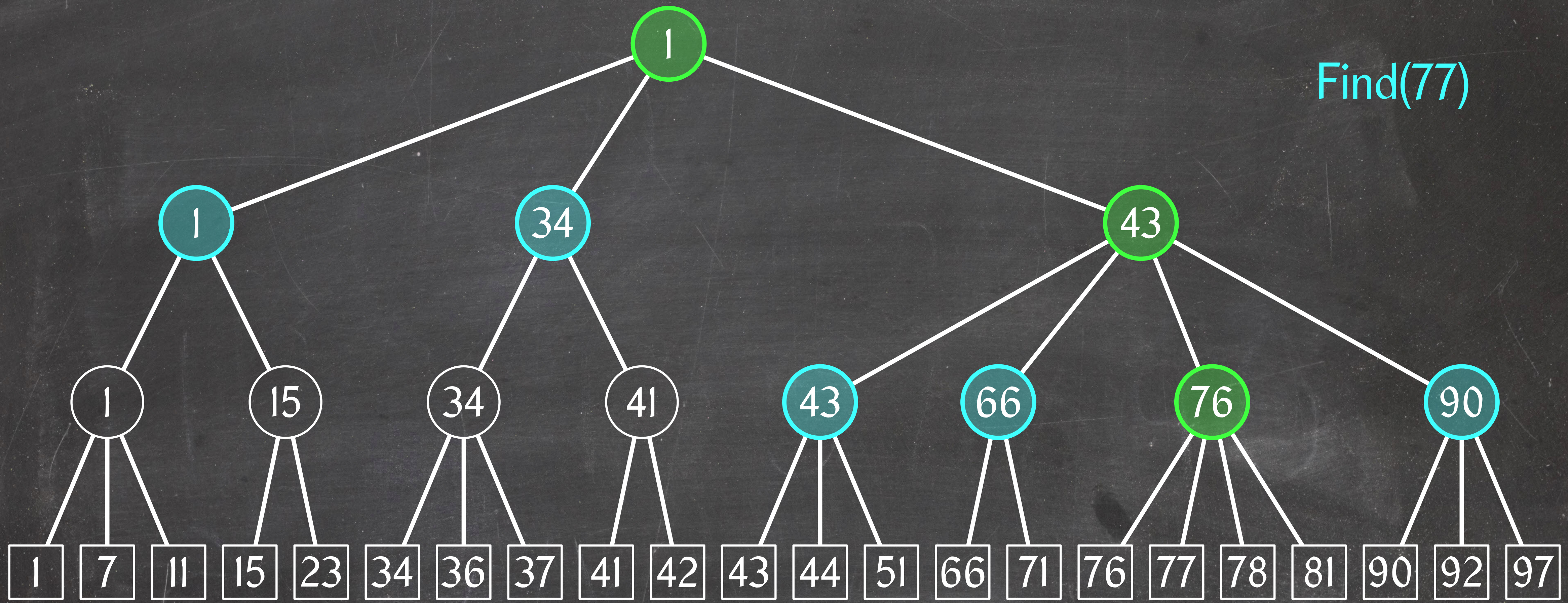
Find/Predecessor Operation



Find/Predecessor Operation

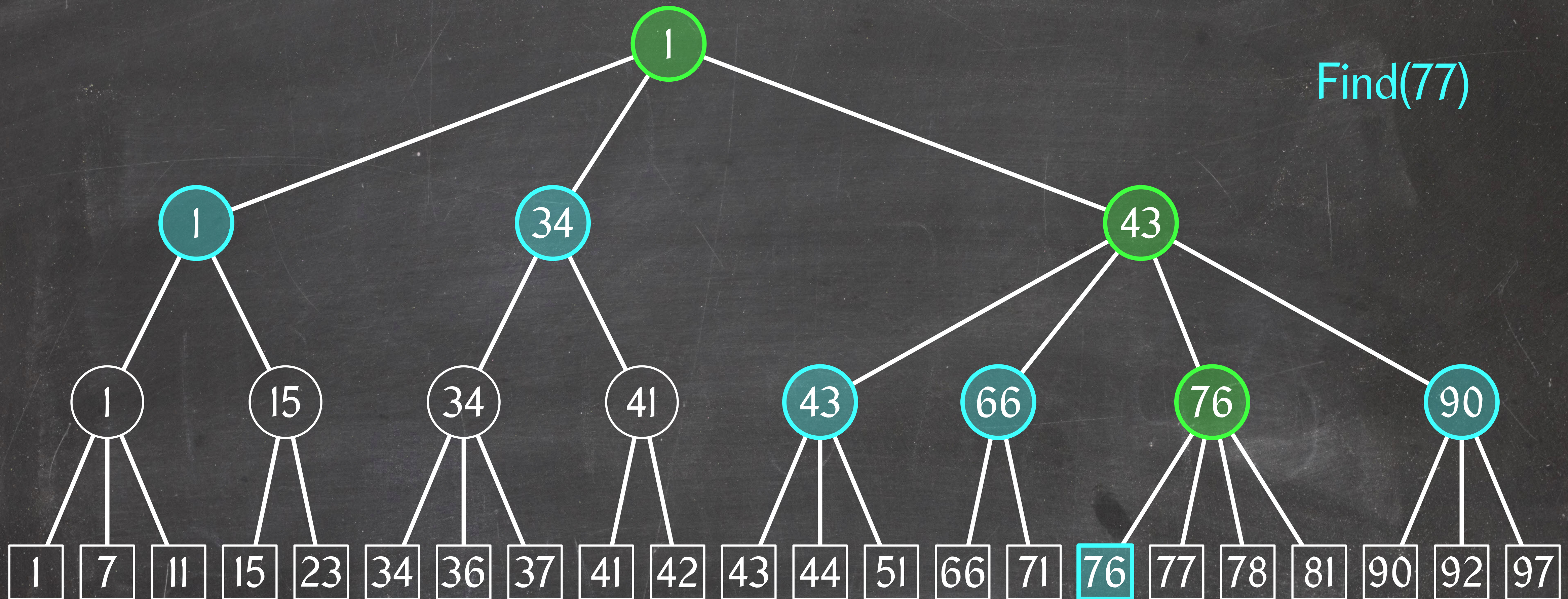


Find/Predecessor Operation



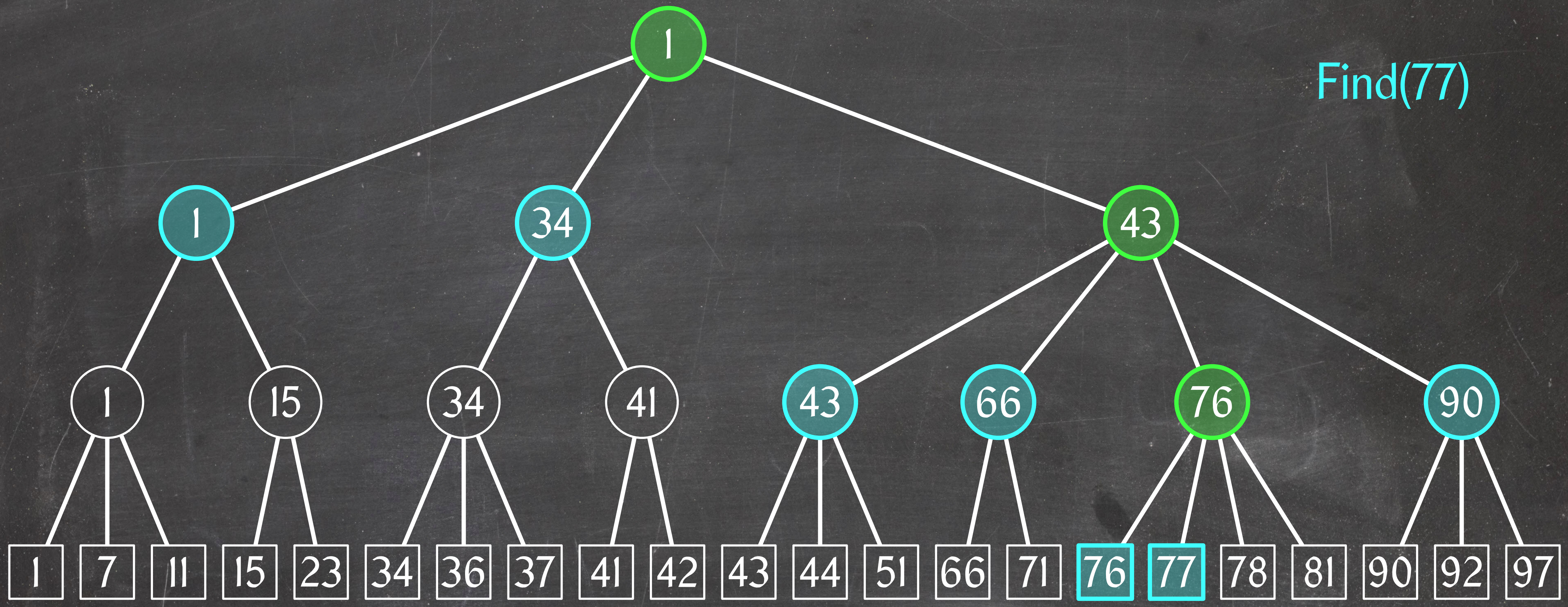
Find(77)

Find/Predecessor Operation



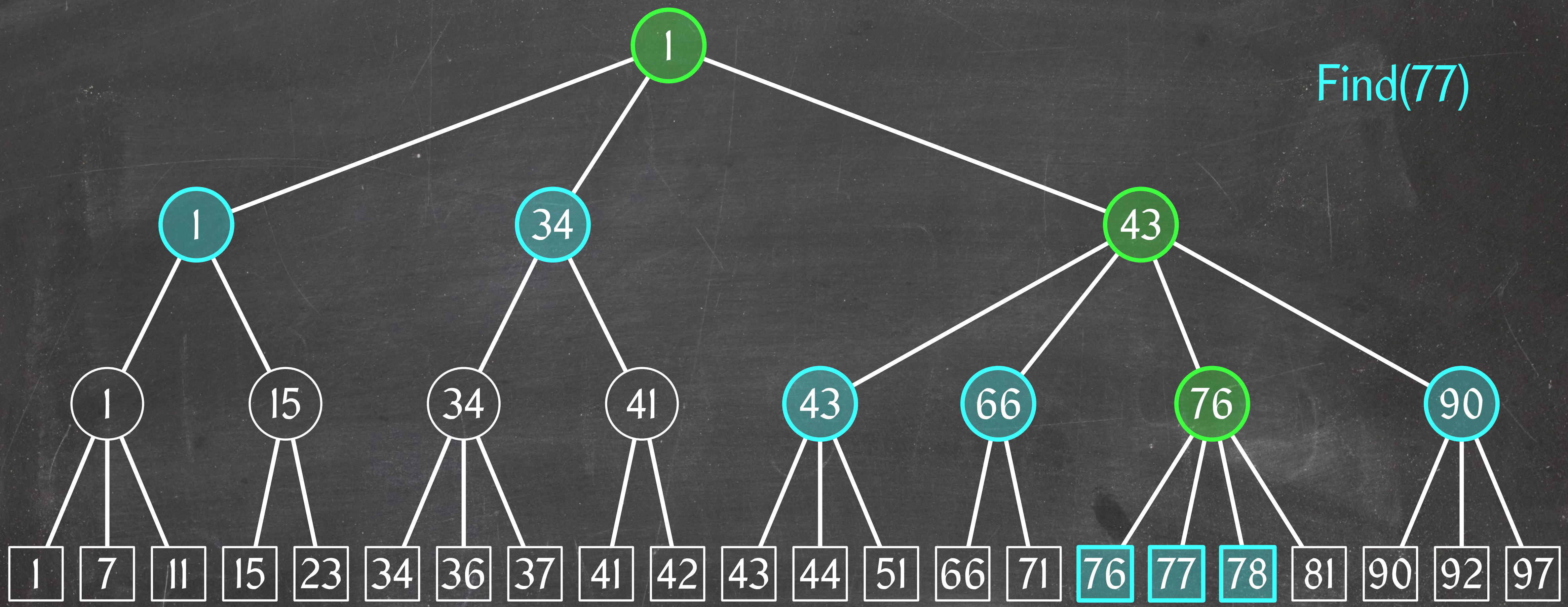
Find(77)

Find/Predecessor Operation



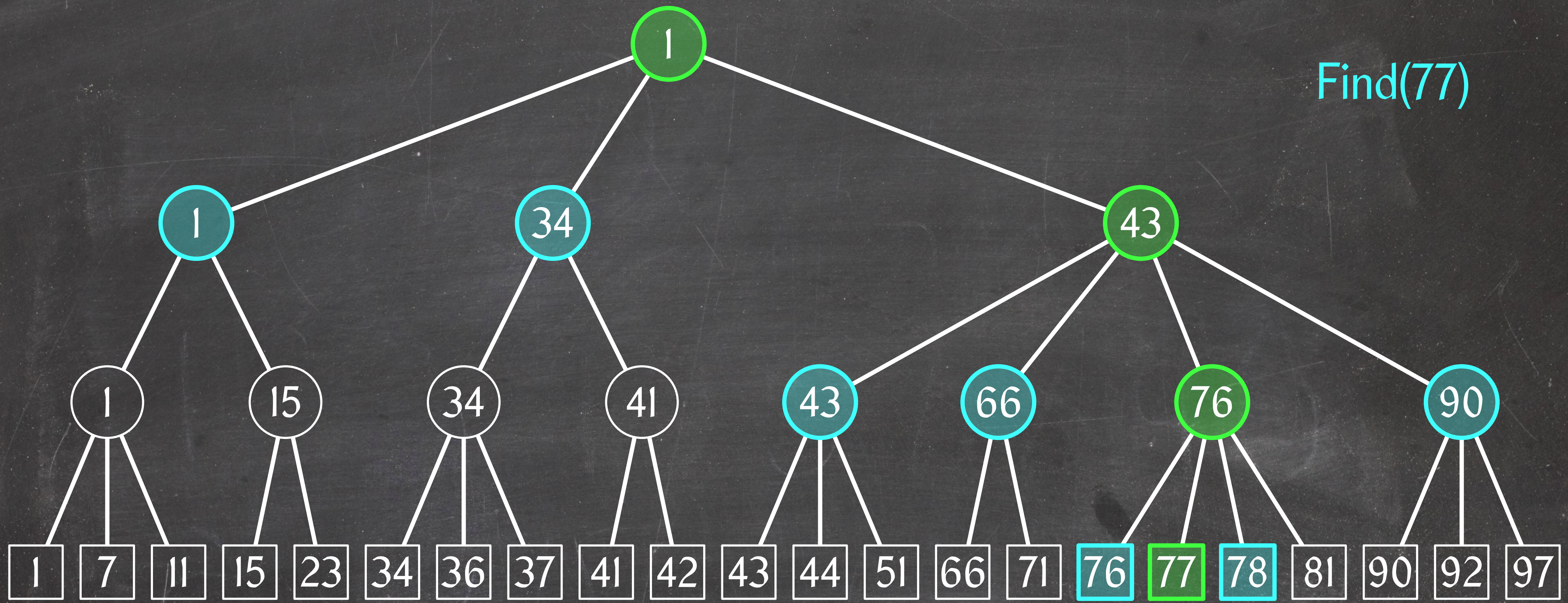
Find(77)

Find/Predecessor Operation



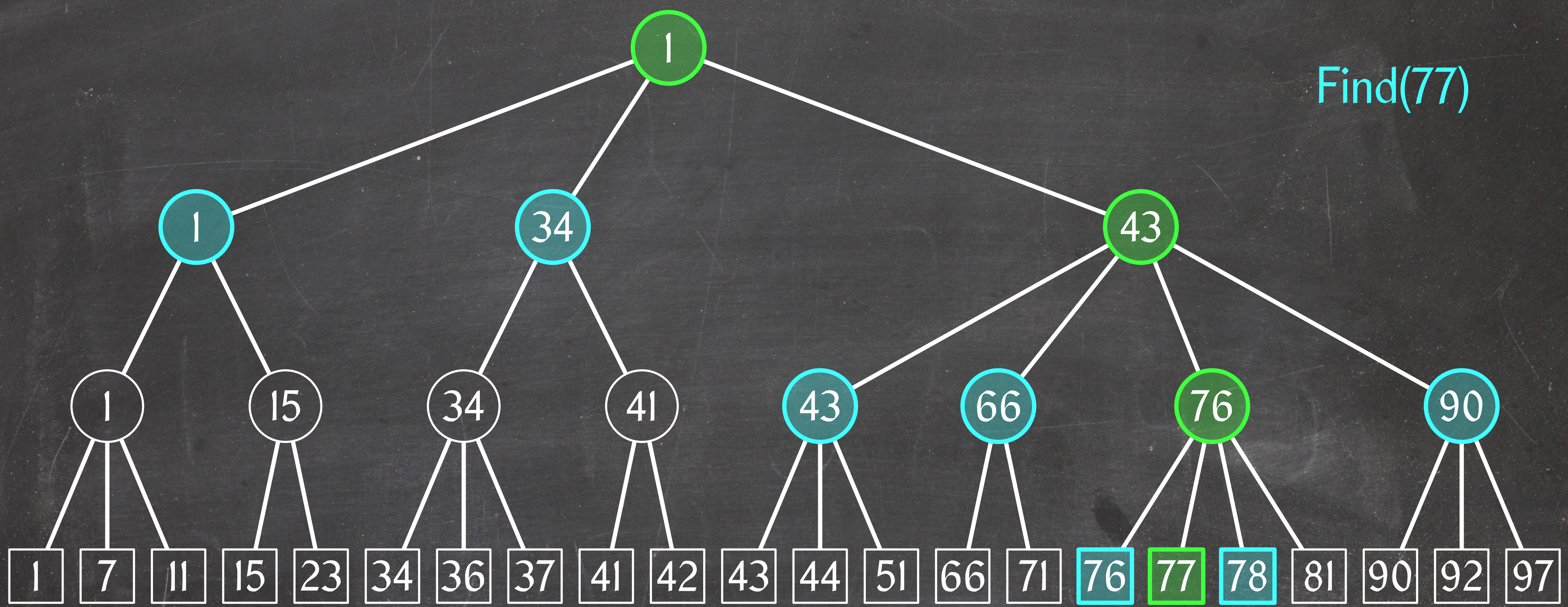
Find(77)

Find/Predecessor Operation



Find(77)

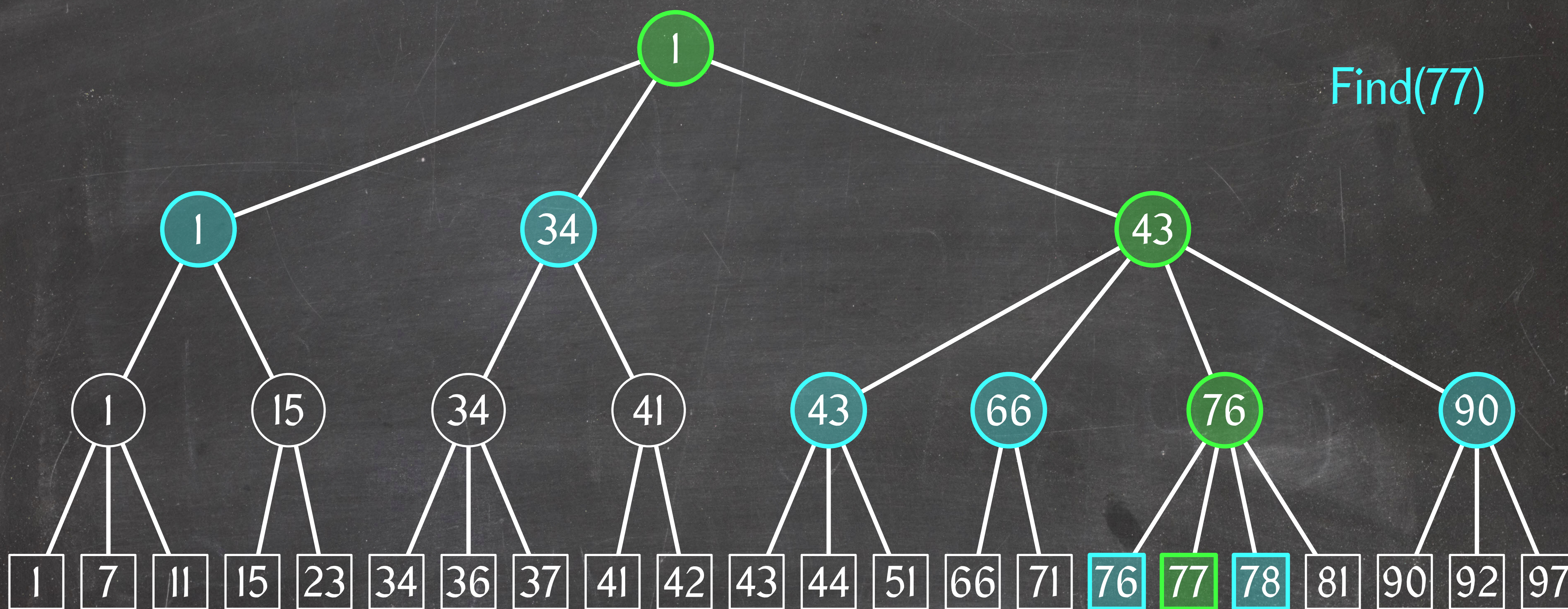
Find/Predecessor Operation



Find(v, x)/Predecessor(v, x):

- If v is not a leaf, then
  - Locate the child w such that
    - w has no right sibling or
    - w's right sibling has a key greater than x
  - Find(w, x)/Predecessor(w, x)
- If v is a leaf, then
  - Report v's key-value pair (Predecessor)
  - Report v's key-value pair if the key equals x, nil otherwise (Find)

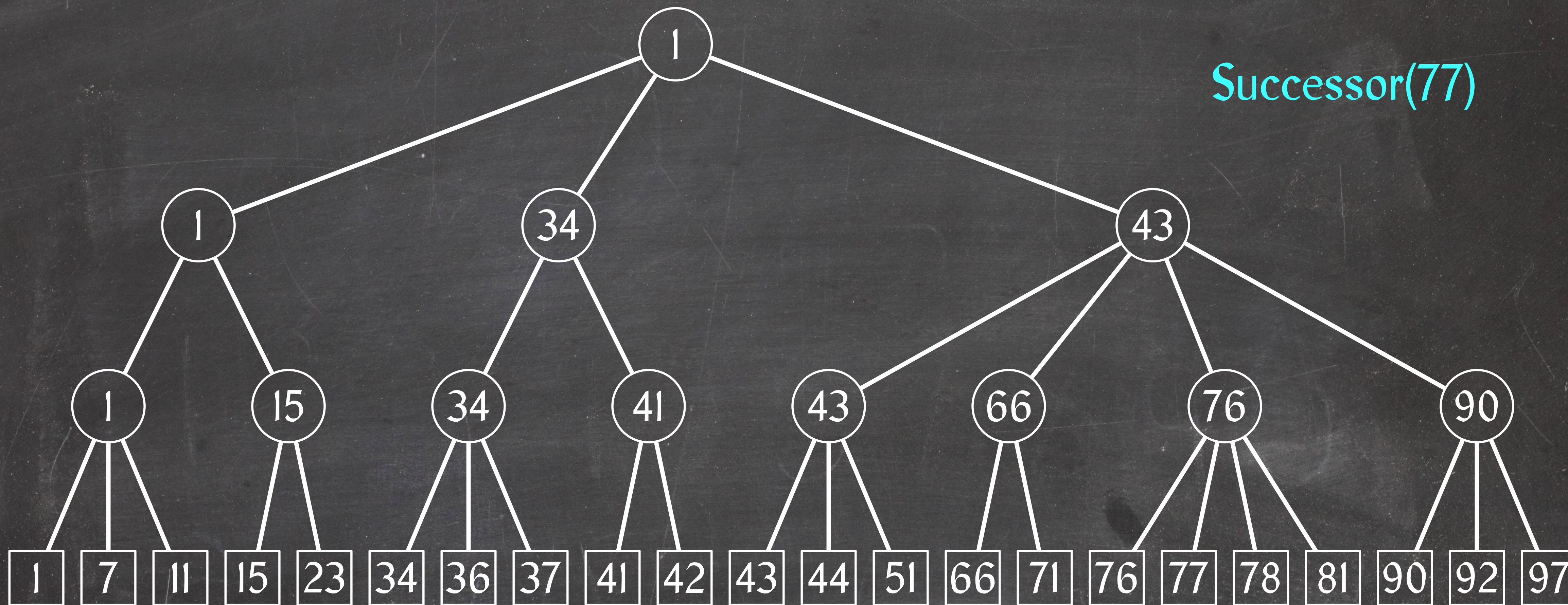
Find/Predecessor Operation



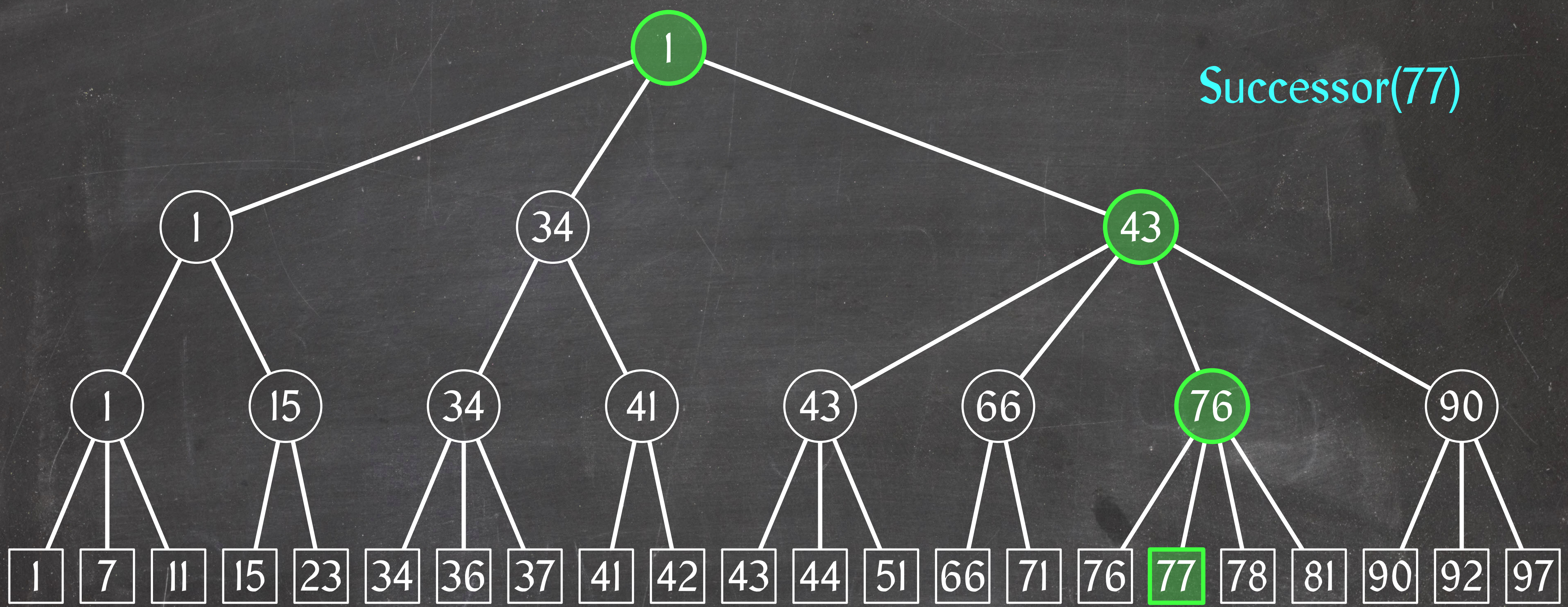
- We inspect at most b nodes per level.
- The cost per node is $O(1)$.

$\Rightarrow$ Cost of Find/Predecessor is in $O(b \log_a n) = O(\lg n)$.

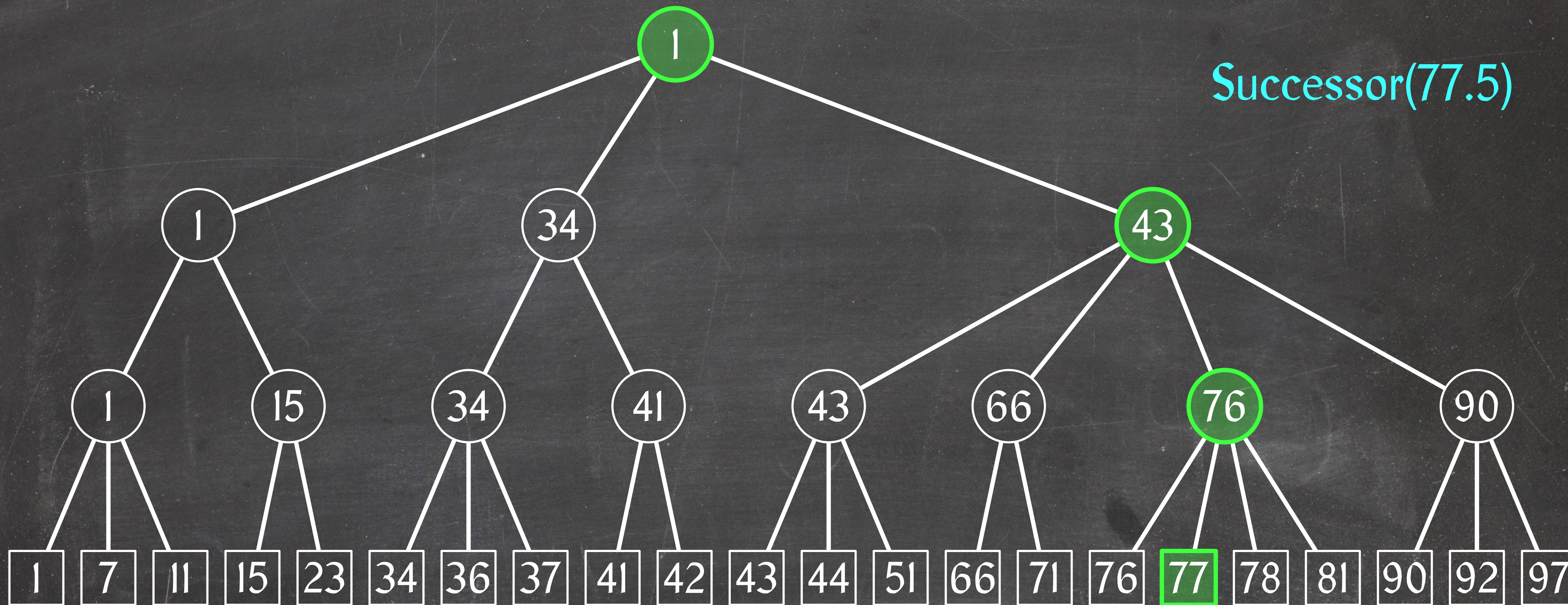
Successor Operation



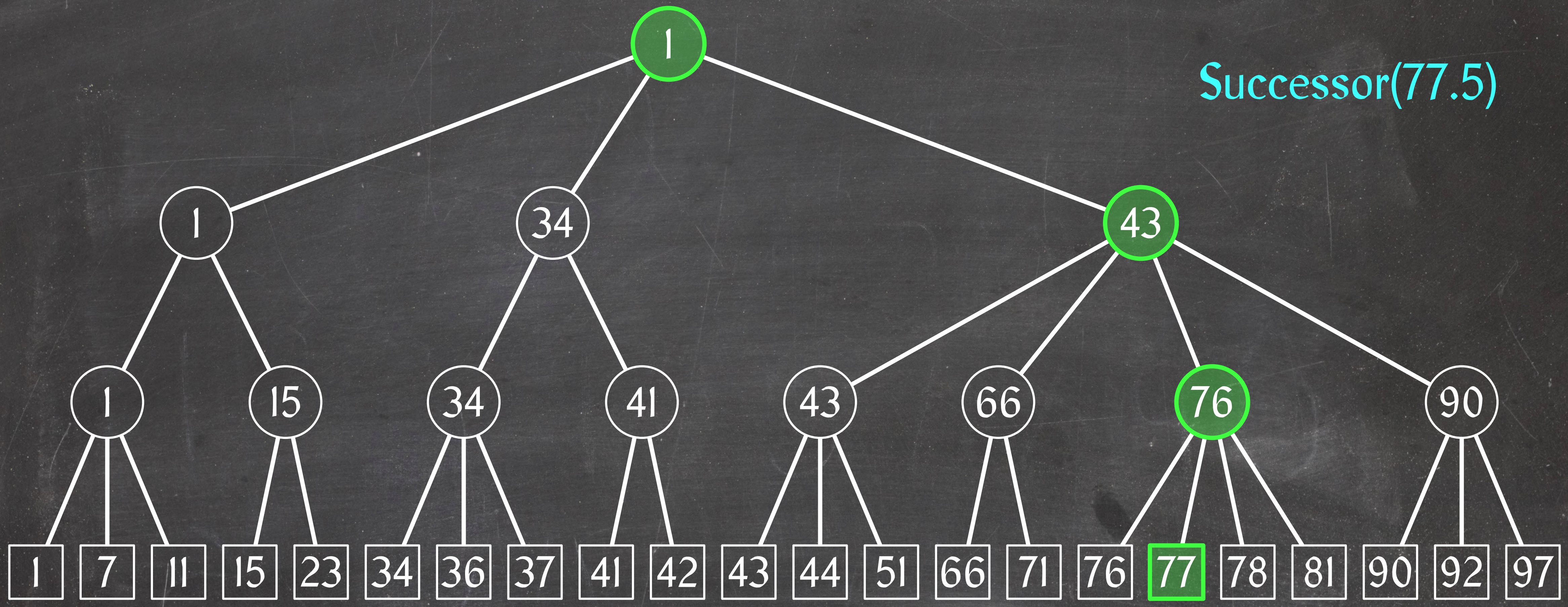
Successor Operation



Successor Operation

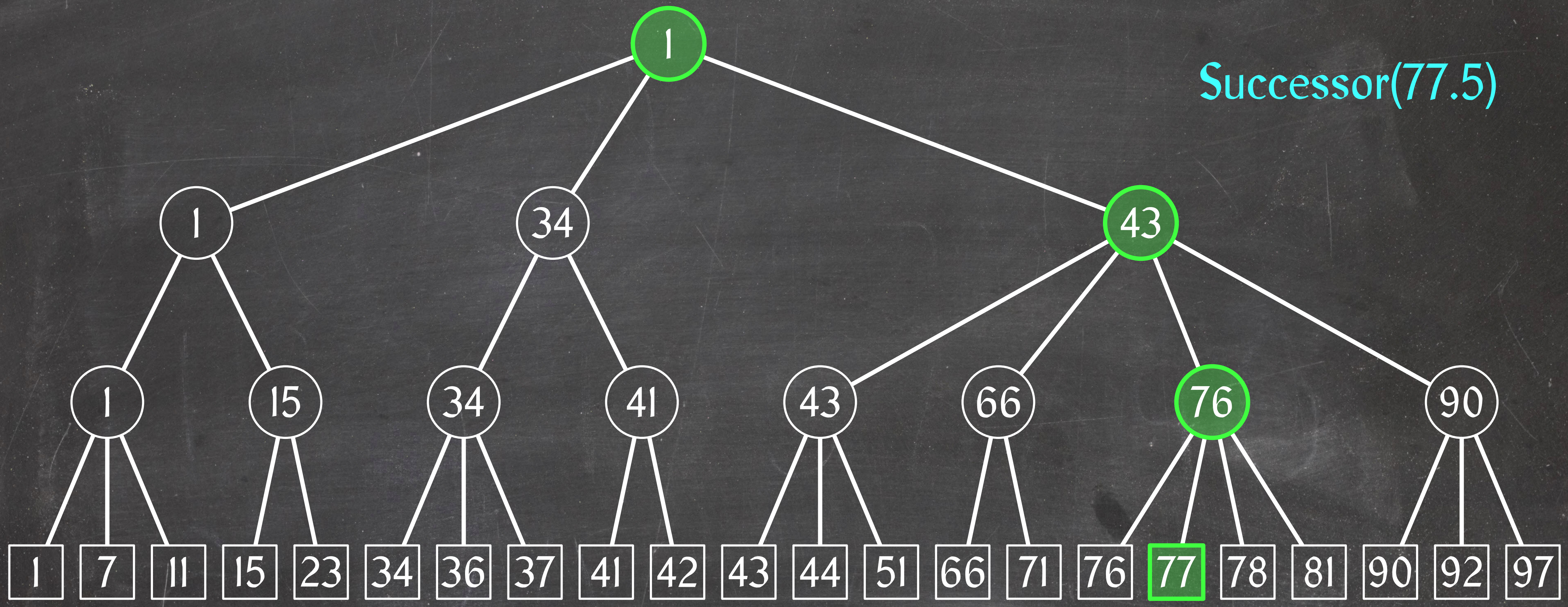


Successor Operation



Since x is possibly itself the answer to a Successor(x) query, we need to locate the node that may hold x first.

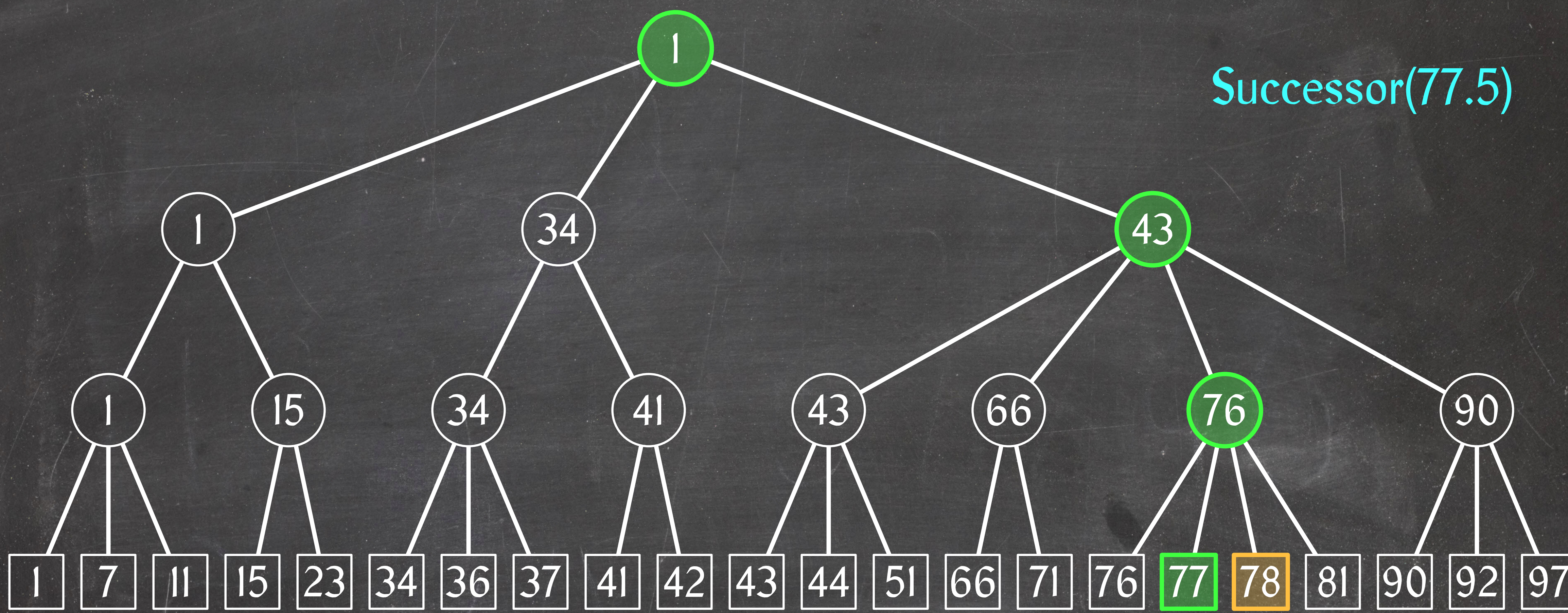
Successor Operation



Since x is possibly itself the answer to a Successor(x) query, we need to locate the node that may hold x first.

How do we find the successor if $x \notin T$?

Successor Operation

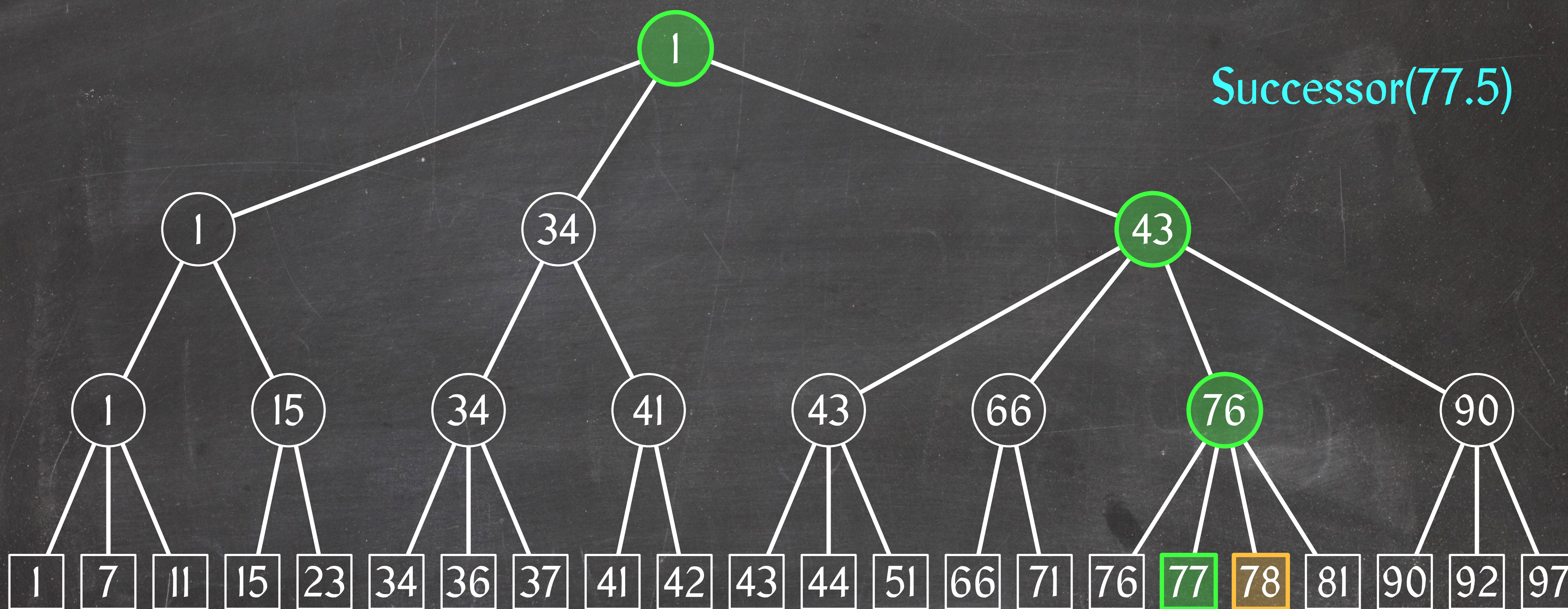


Since x is possibly itself the answer to a Successor(x) query, we need to locate the node that may hold x first.

How do we find the successor if $x \notin T$?

We walk up to x's closest ancestor that has a right sibling and locate the leftmost descendant leaf of this sibling.

Successor Operation



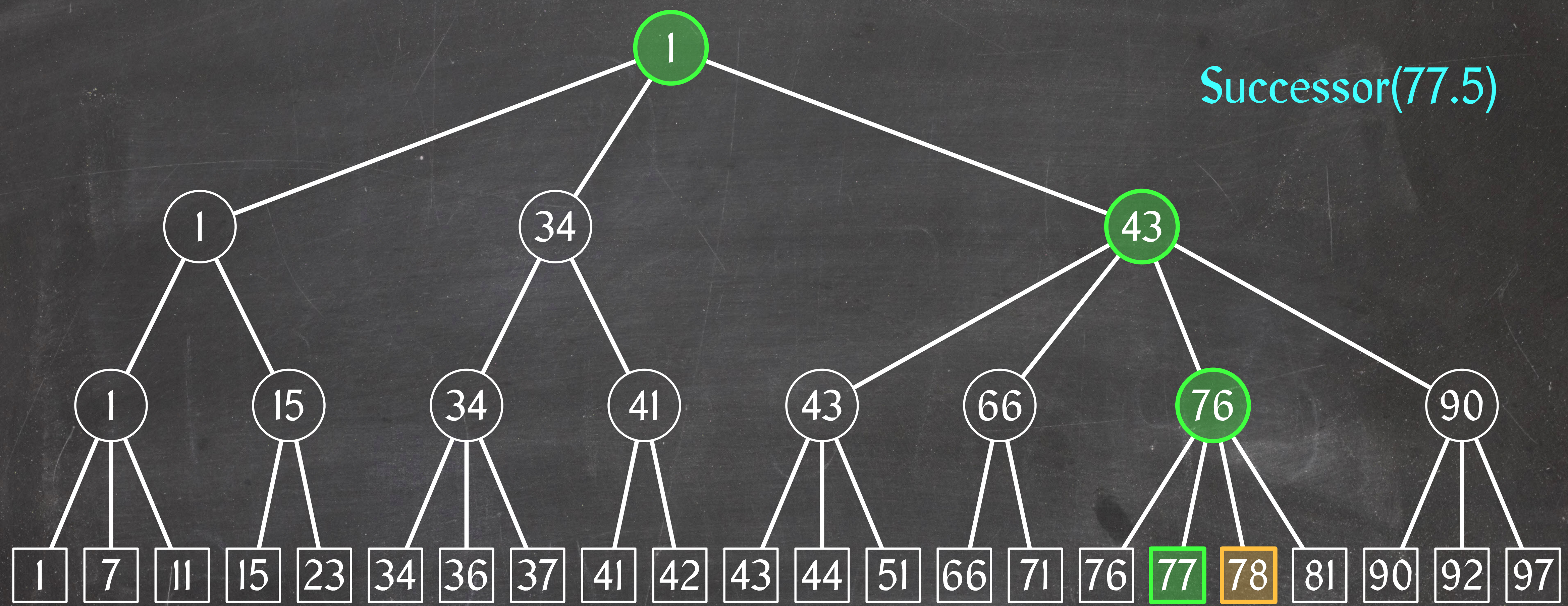
Since x is possibly itself the answer to a Successor(x) query, we need to locate the node that may hold x first.

How do we find the successor if $x \notin T$?

We walk up to x's closest ancestor that has a right sibling and locate the leftmost descendant leaf of this sibling.

How do we walk up?

Successor Operation



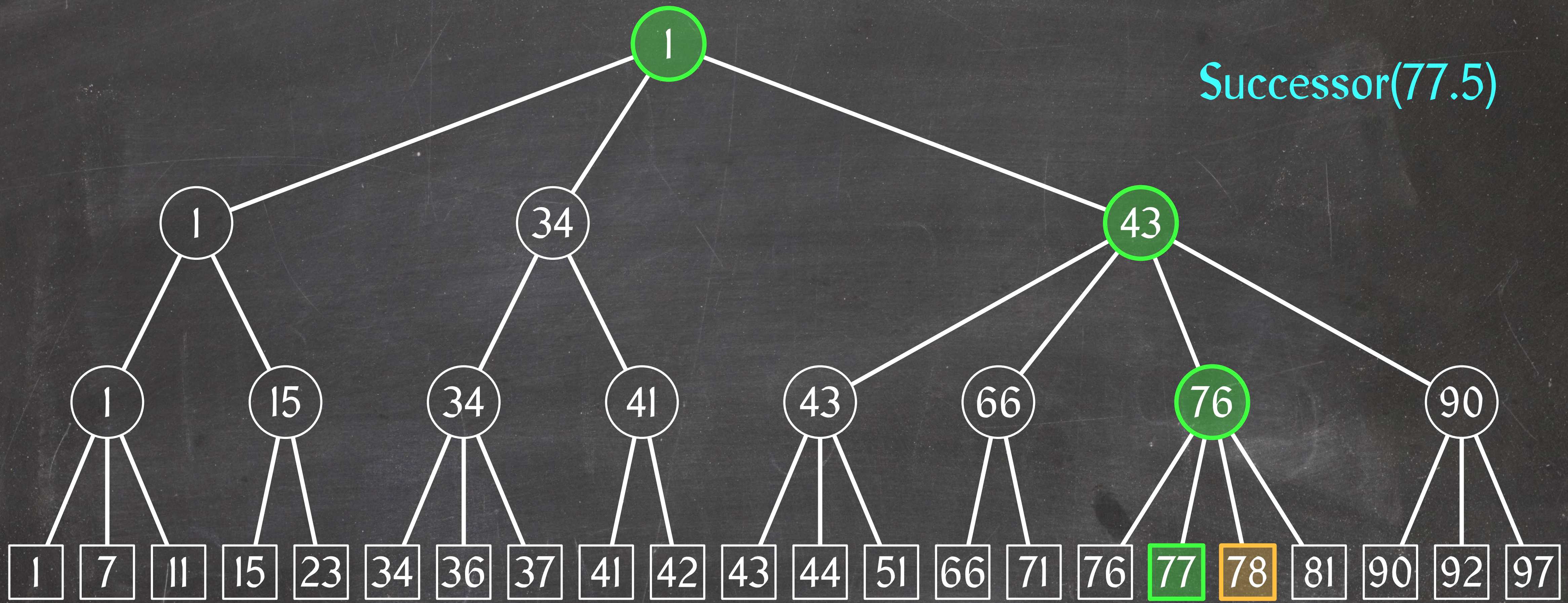
Since x is possibly itself the answer to a Successor(x) query, we need to locate the node that may hold x first.

How do we find the successor if $x \notin T$?

We walk up to x's closest ancestor that has a right sibling and locate the leftmost descendant leaf of this sibling.

How do we walk up? Using a stack.

Successor Operation



Successor(77.5)

Since x is possibly itself the answer to a Successor(x) query, we need to locate the node that may hold x first.

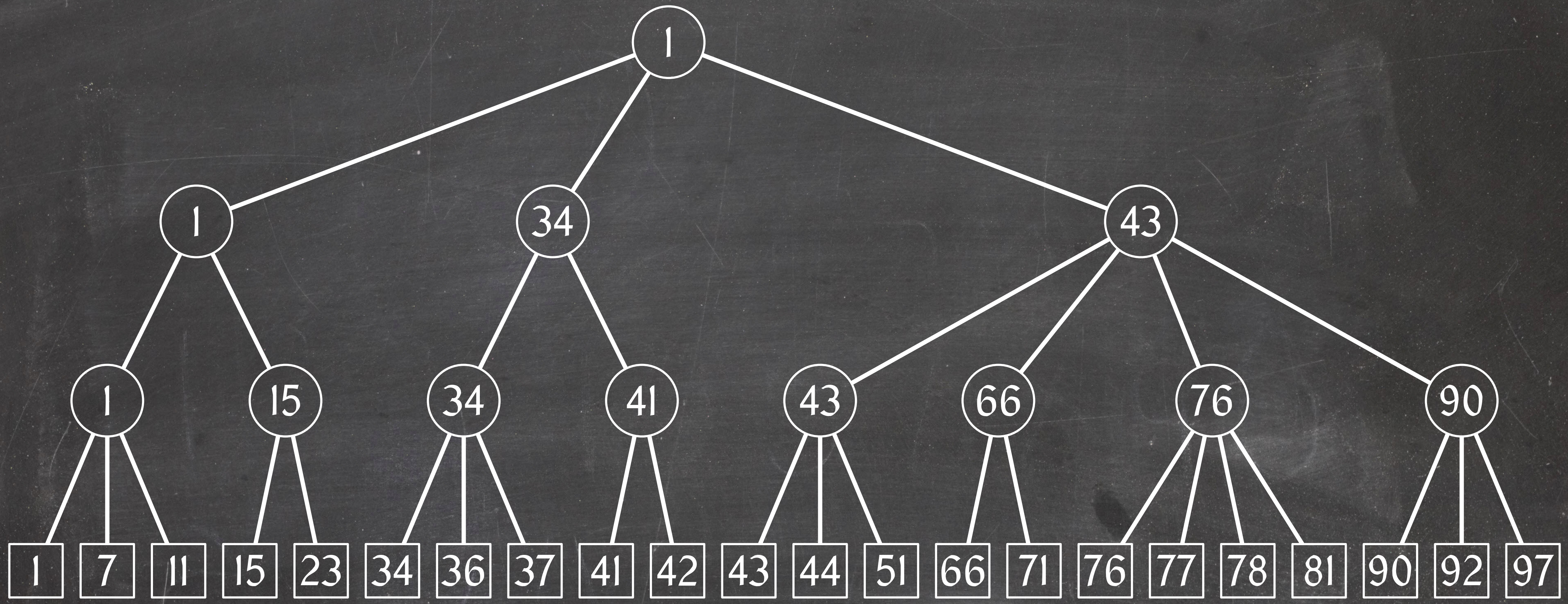
How do we find the successor if $x \notin T$?

We walk up to x's closest ancestor that has a right sibling and locate the leftmost descendant leaf of this sibling.

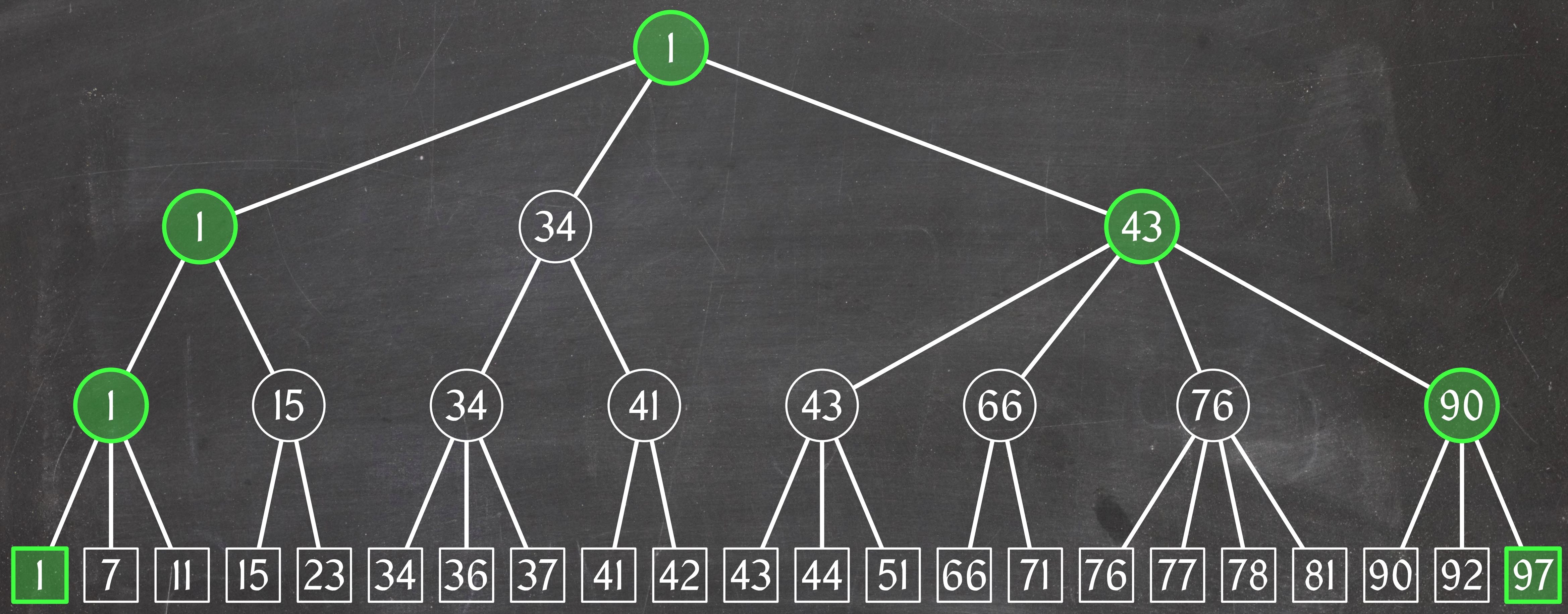
Cost: $O(\lg n)$

How do we walk up? Using a stack.

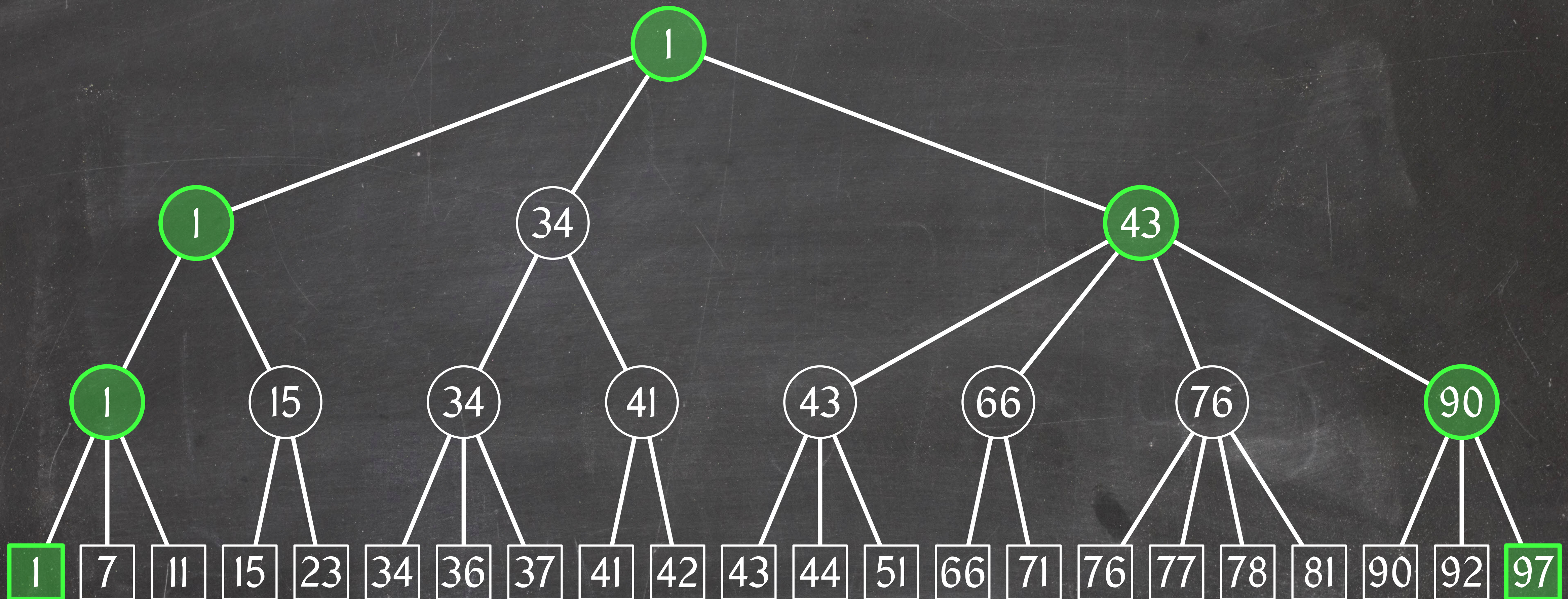
Minimum/Maximum Operation



Minimum/Maximum Operation

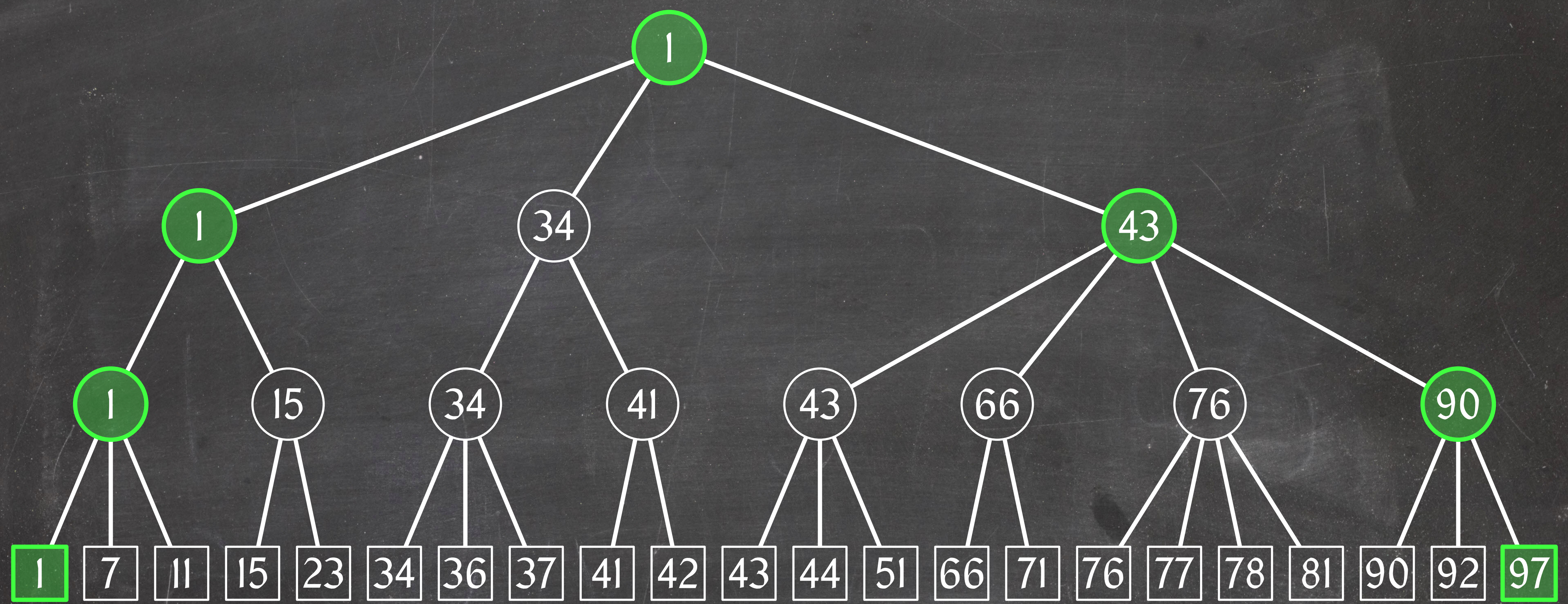


Minimum/Maximum Operation



Follow the path to the leftmost/rightmost leaf.

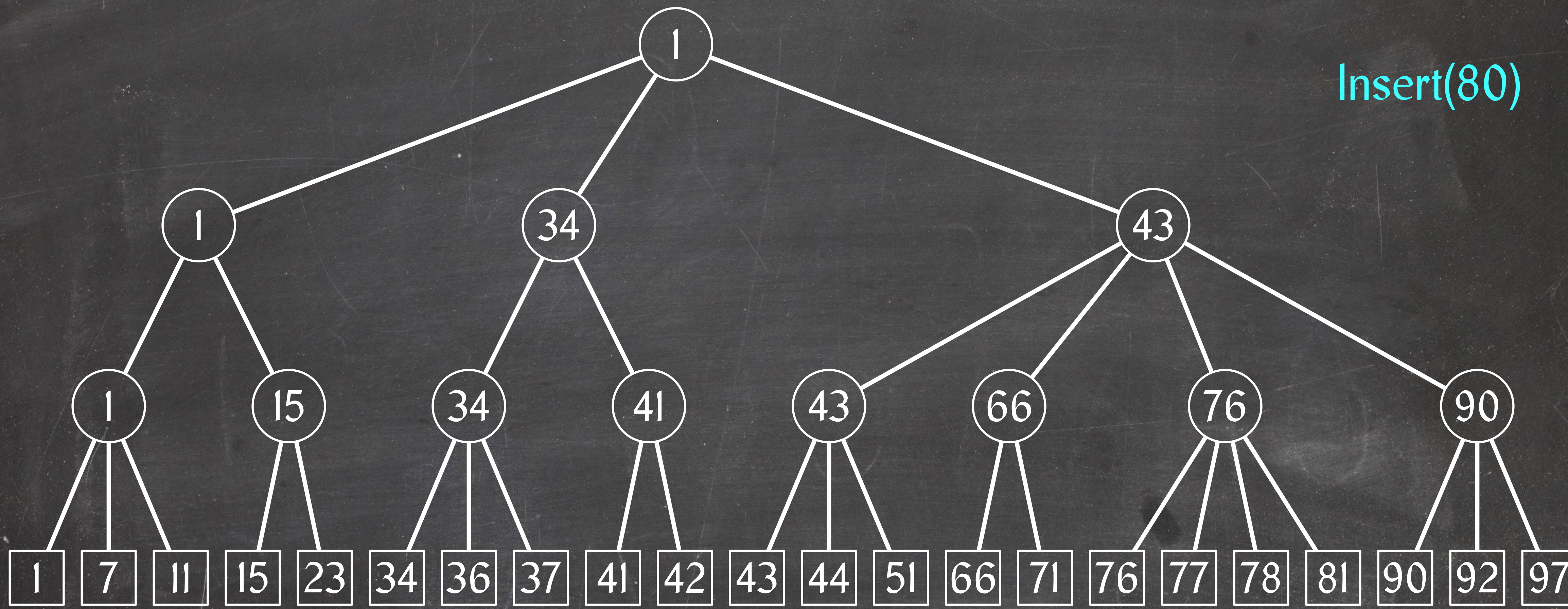
Minimum/Maximum Operation



Follow the path to the leftmost/rightmost leaf.

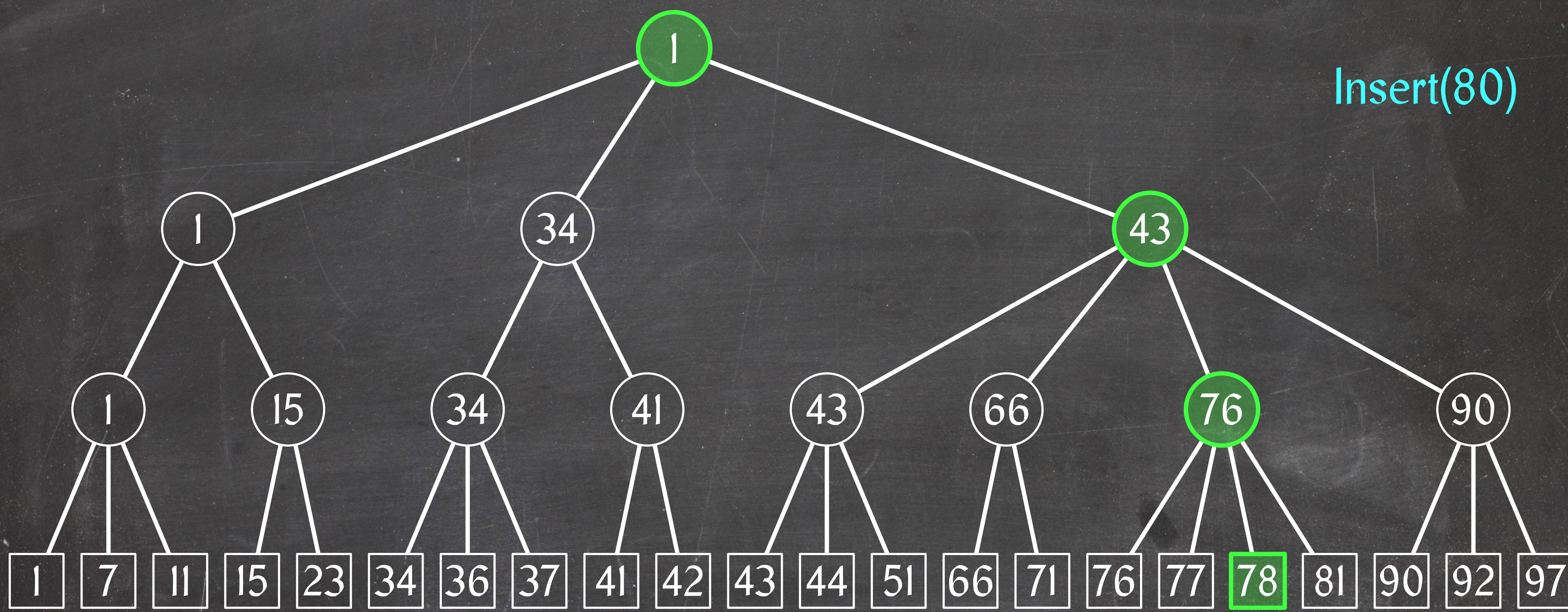
Cost: $O(b \log_a n) = O(\lg n)$

Insert Operation



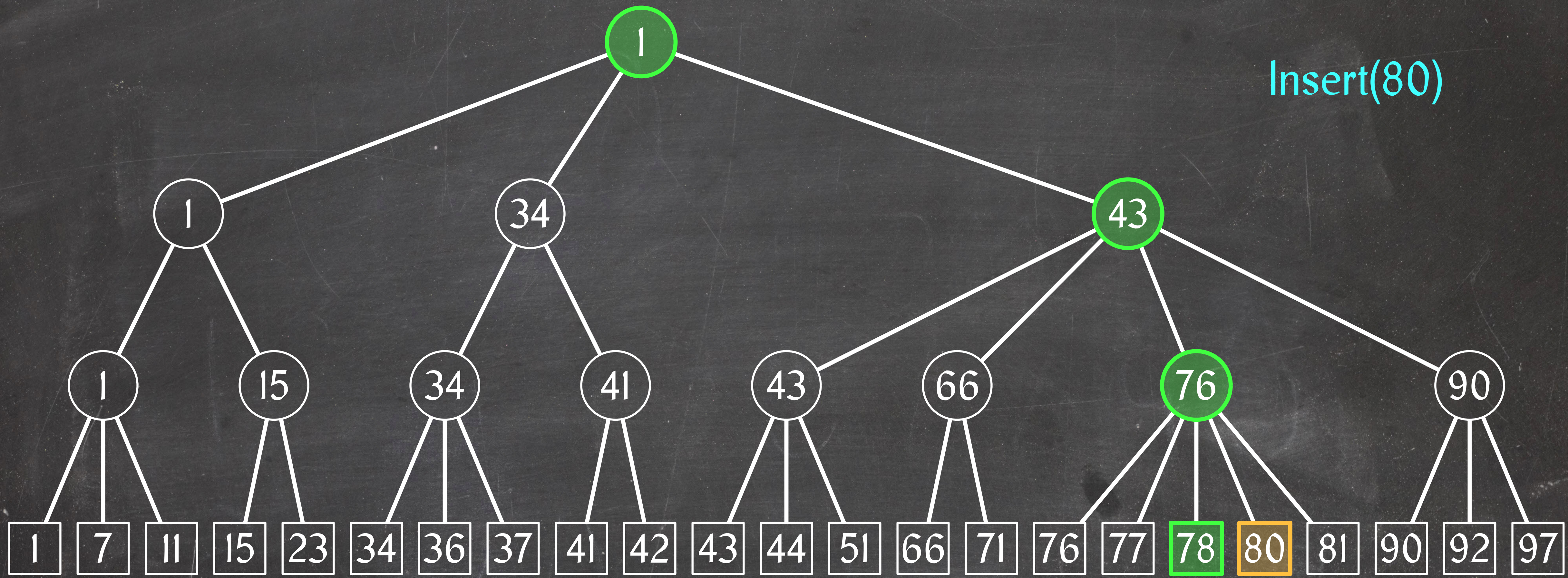
Insert(80)

Insert Operation



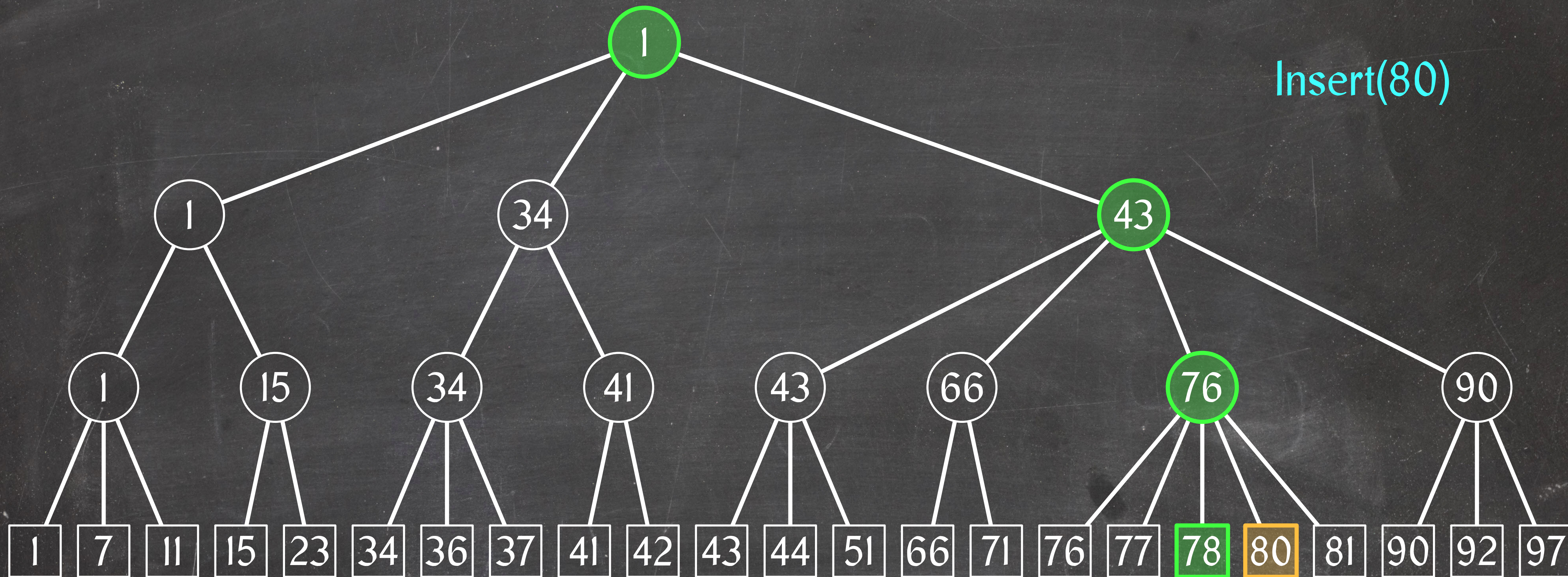
- Use a Predecessor(x) query to find the greatest leaf no greater than x.

Insert Operation



- Use a Predecessor(x) query to find the greatest leaf no greater than x.
- Make x a right sibling of this leaf.

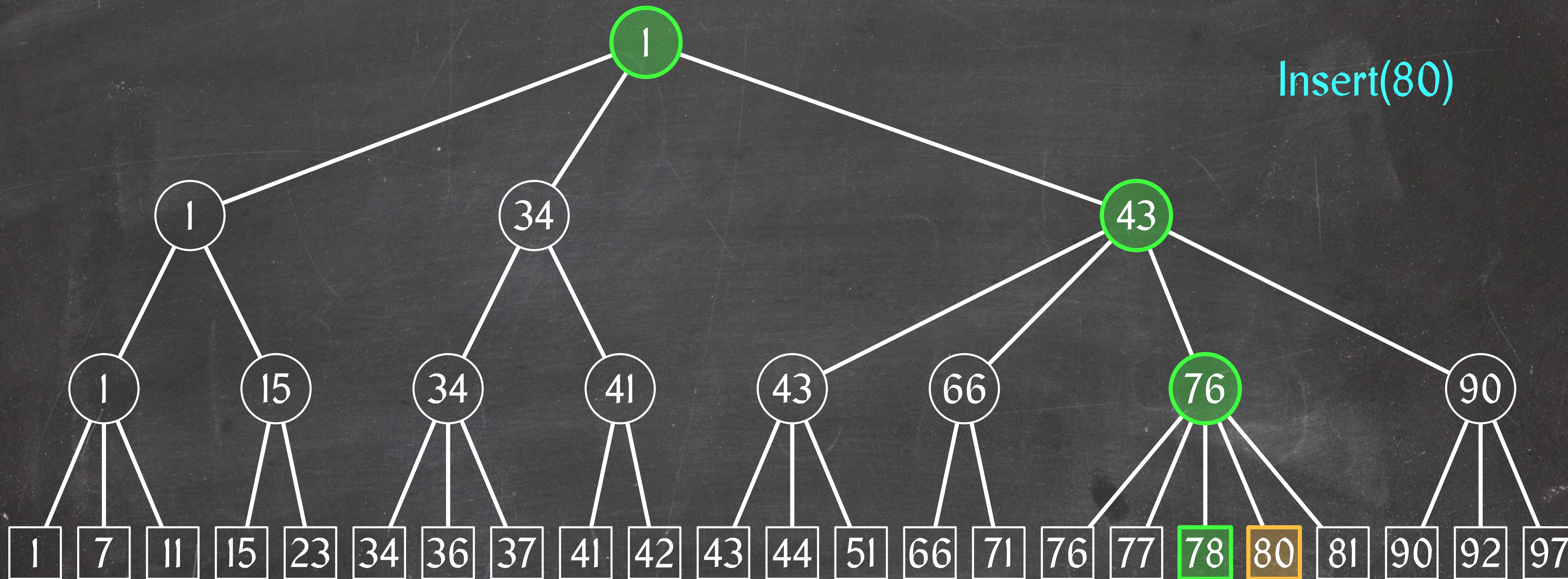
Insert Operation



- Use a Predecessor(x) query to find the greatest leaf no greater than x.
- Make x a right sibling of this leaf.

Is the result still an (a, b)-tree?

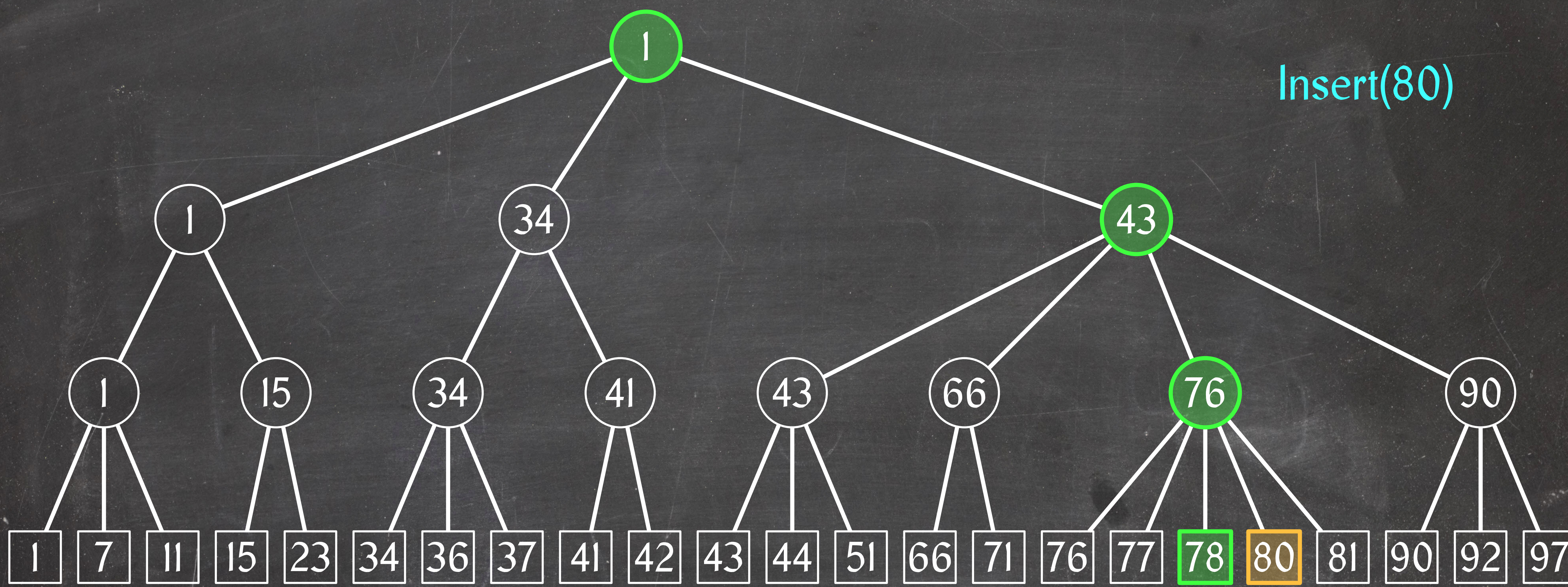
Insert Operation



- Use a Predecessor(x) query to find the greatest leaf no greater than x.
- Make x a right sibling of this leaf.

Is the result still an (a, b)-tree? Not necessarily!

Insert Operation

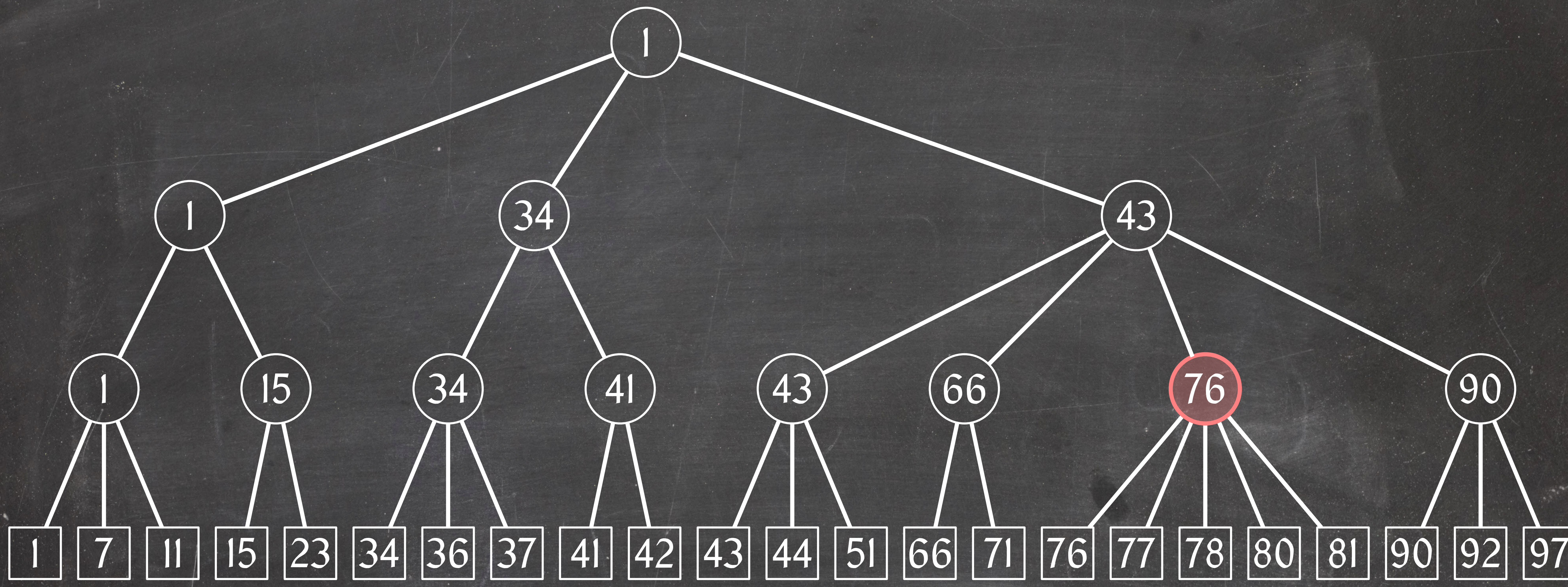


- Use a Predecessor(x) query to find the greatest leaf no greater than x.
- Make x a right sibling of this leaf.

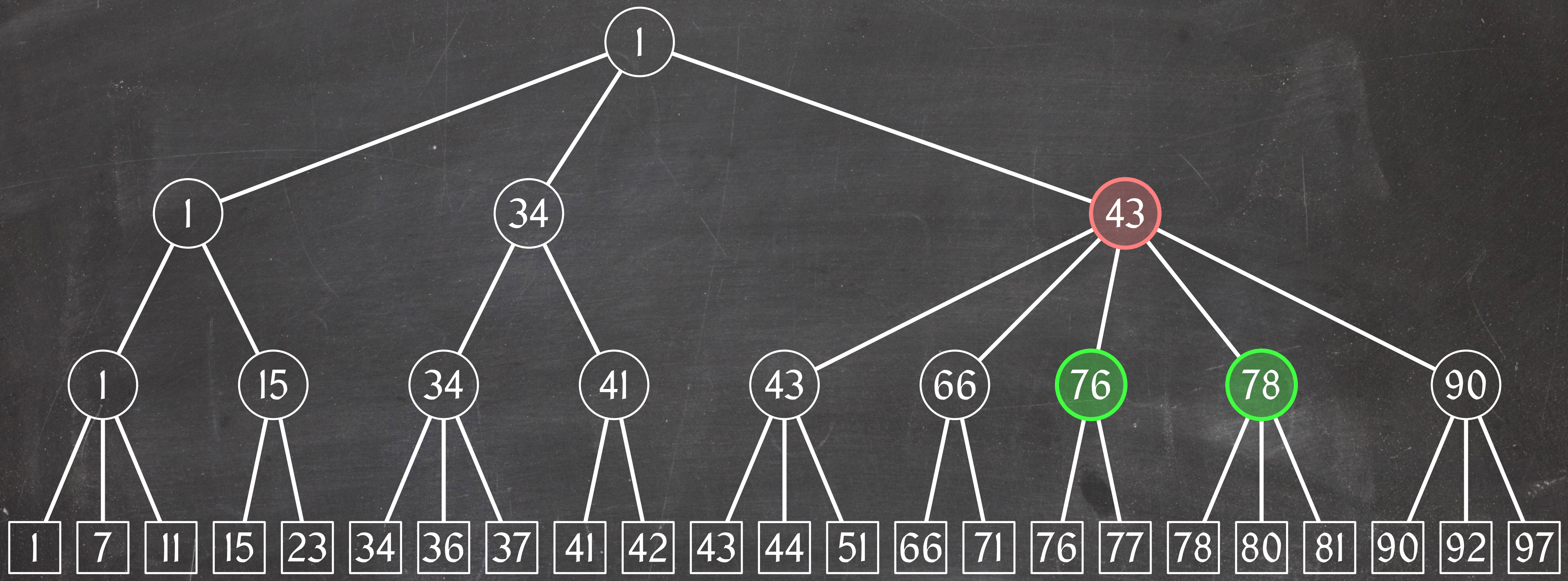
Is the result still an (a, b)-tree? Not necessarily!

How do we rebalance?

Node Splitting

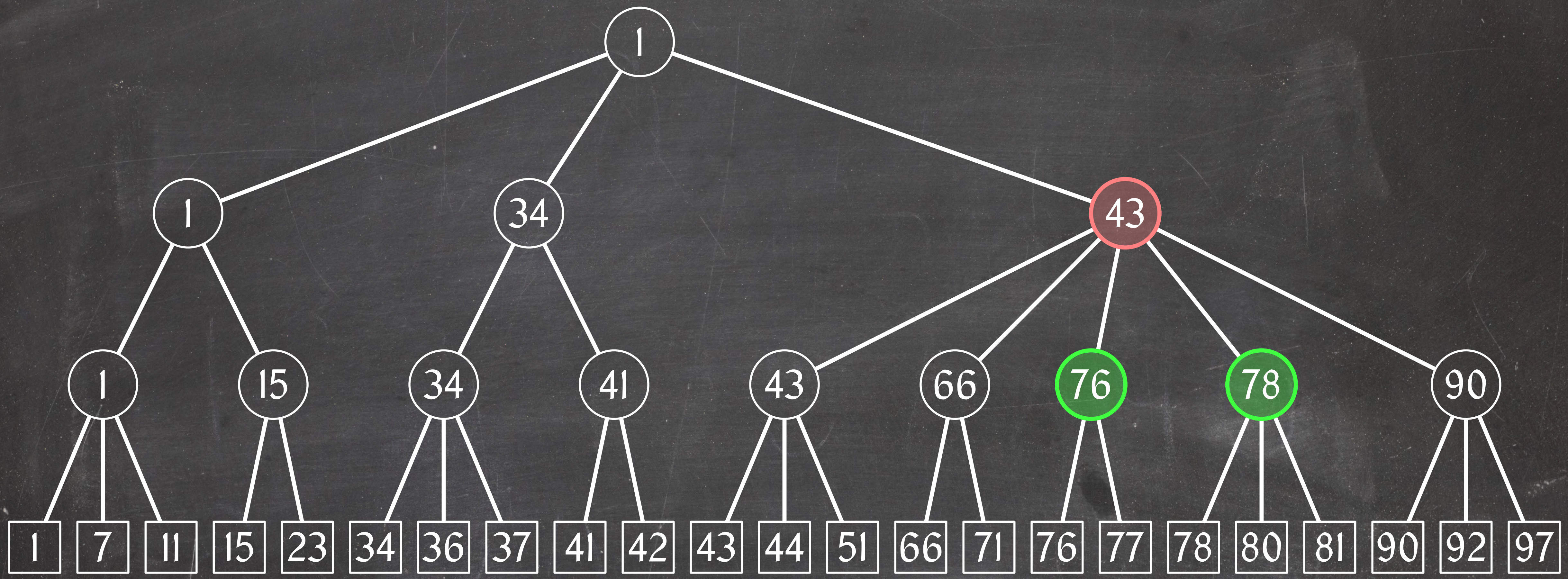


Node Splitting



Split a node of degree $b+1$ into two nodes of degrees $\left\lfloor \frac{b+1}{2} \right\rfloor$ and $\left\lceil \frac{b+1}{2} \right\rceil$.

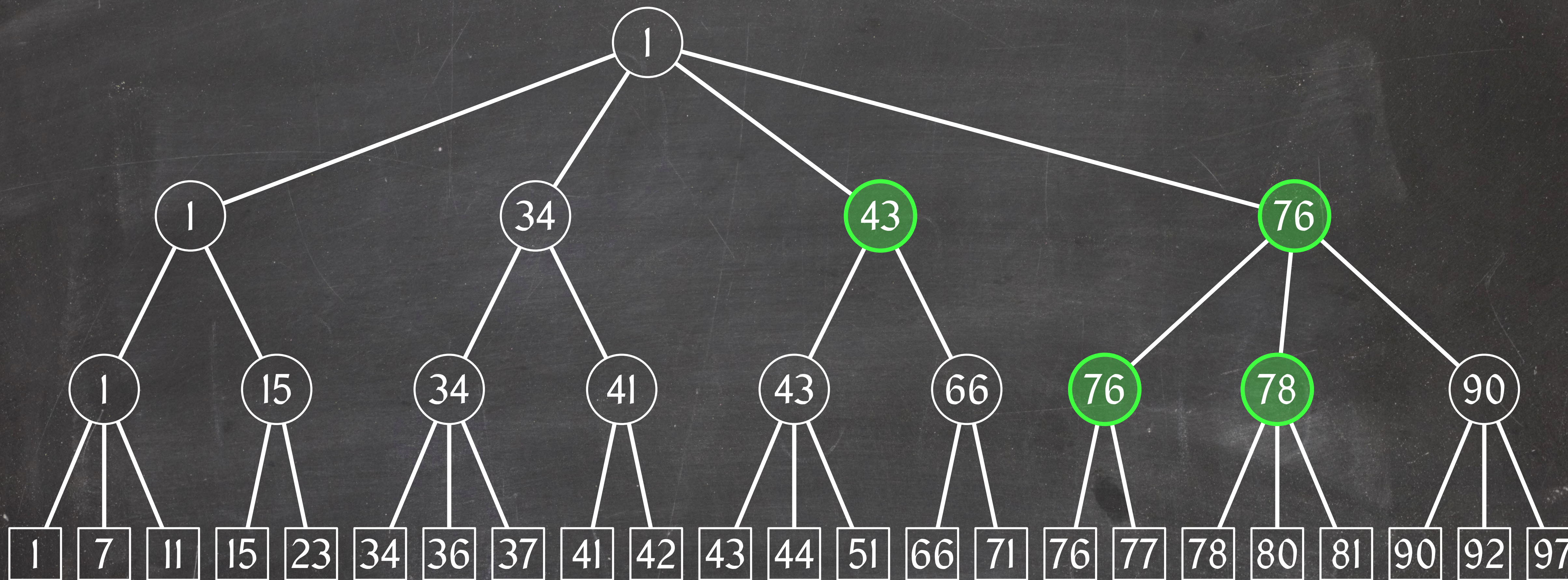
Node Splitting



Split a node of degree $b+1$ into two nodes of degrees $\left\lfloor \frac{b+1}{2} \right\rfloor$ and $\left\lceil \frac{b+1}{2} \right\rceil$.

- We have $a = \left\lfloor \frac{2a}{2} \right\rfloor \le \left\lfloor \frac{b+1}{2} \right\rfloor \le \left\lceil \frac{b+1}{2} \right\rceil \le \frac{b}{2} + 1 \le b$.

Node Splitting

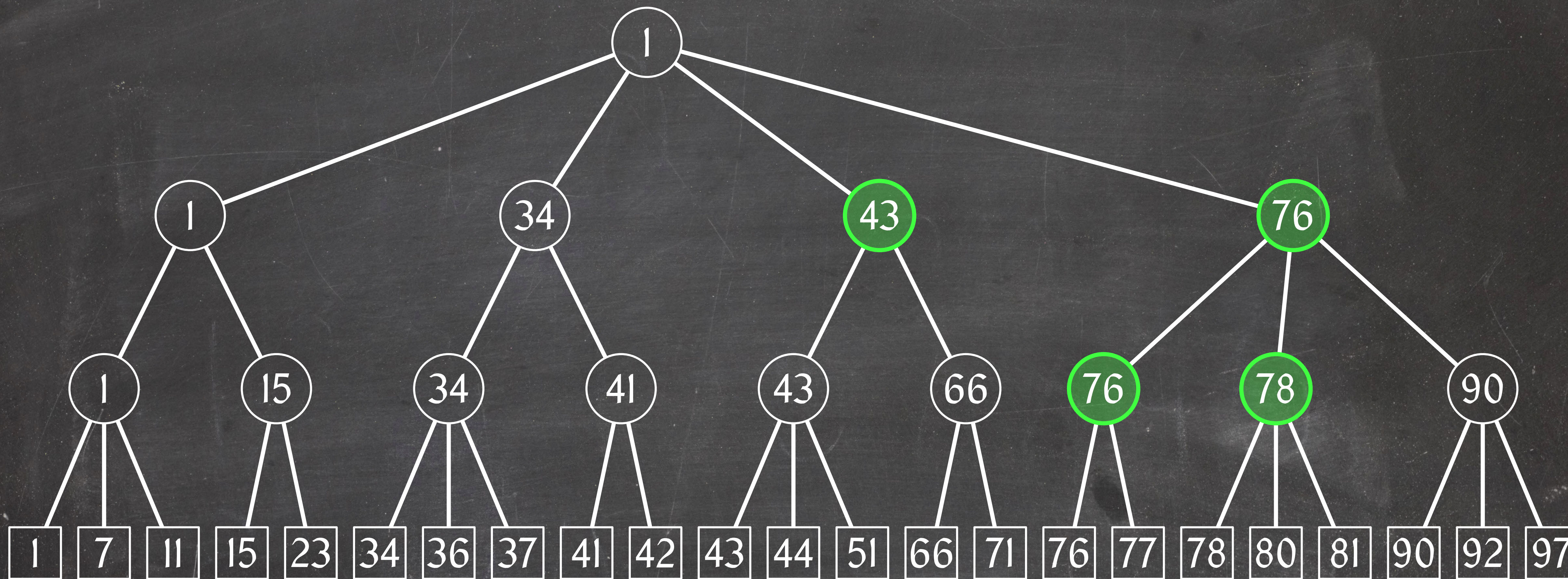


Split a node of degree $b+1$ into two nodes of degrees $\left\lfloor \frac{b+1}{2} \right\rfloor$ and $\left\lceil \frac{b+1}{2} \right\rceil$.

- We have $a = \left\lfloor \frac{2a}{2} \right\rfloor \leq \left\lfloor \frac{b+1}{2} \right\rfloor \leq \left\lceil \frac{b+1}{2} \right\rceil \leq \frac{b}{2} + 1 \leq b$.

If the parent now has degree $b+1$, split the parent recursively.

Node Splitting



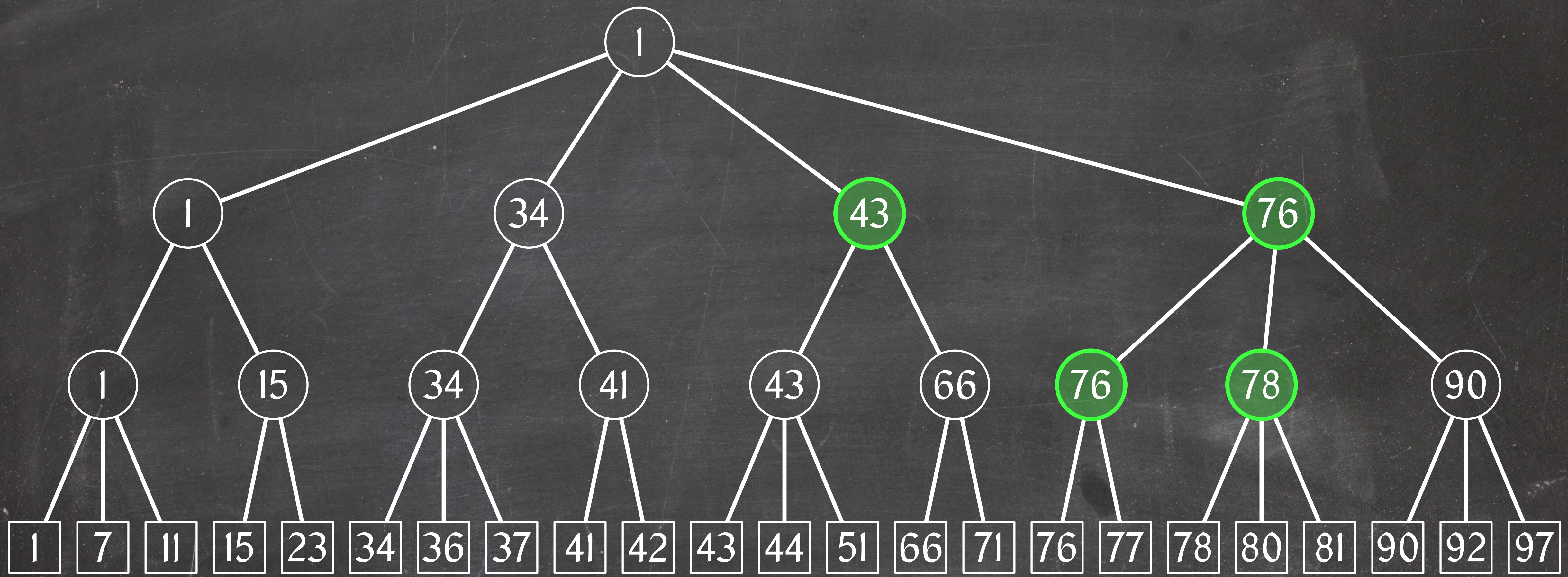
Split a node of degree $b+1$ into two nodes of degrees $\lfloor \frac{b+1}{2} \rfloor$ and $\lceil \frac{b+1}{2} \rceil$.

- We have $a = \lfloor \frac{2a}{2} \rfloor \le \lfloor \frac{b+1}{2} \rfloor \le \lceil \frac{b+1}{2} \rceil \le \frac{b}{2} + 1 \le b$.

If the parent now has degree $b+1$, split the parent recursively.

Cost per node split: $O(b) = O(1)$

Node Splitting



Split a node of degree $b+1$ into two nodes of degrees $\lfloor \frac{b+1}{2} \rfloor$ and $\lceil \frac{b+1}{2} \rceil$.

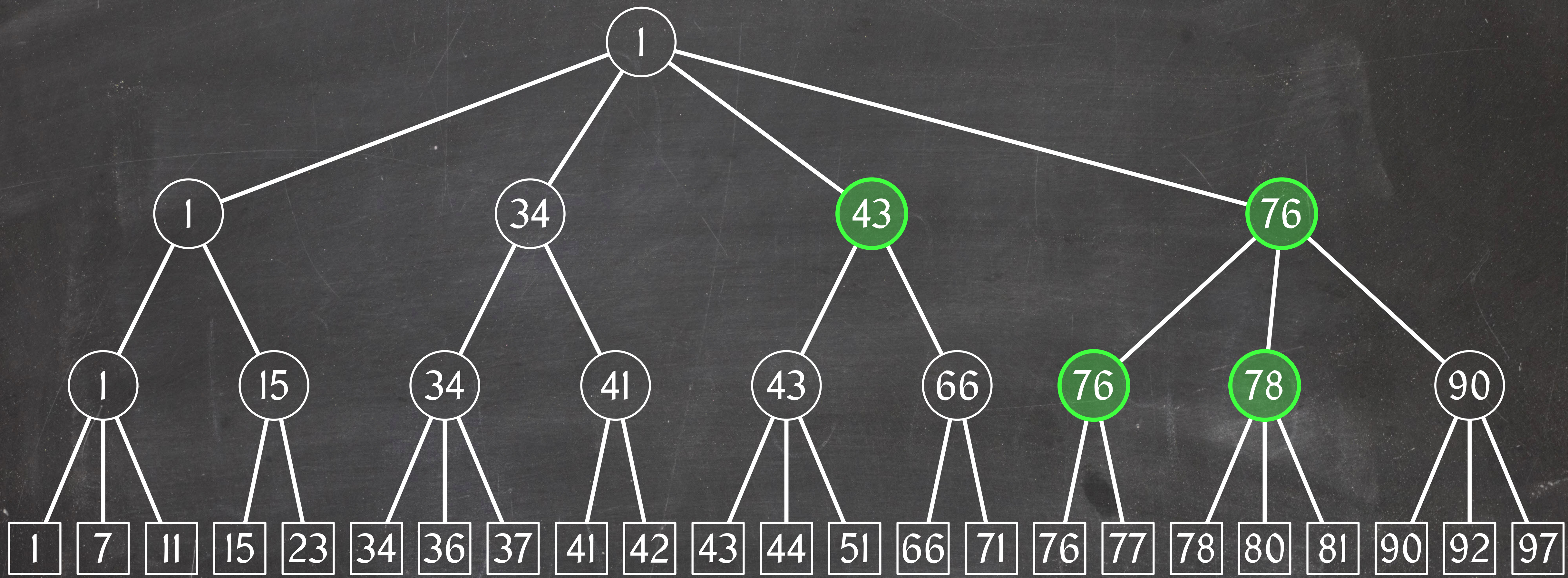
- We have $a = \lfloor \frac{2a}{2} \rfloor \leq \lfloor \frac{b+1}{2} \rfloor \leq \lceil \frac{b+1}{2} \rceil \leq \frac{b}{2} + 1 \leq b$.

If the parent now has degree $b+1$, split the parent recursively.

Cost per node split: $O(b) = O(1)$

At most one node split per level.

Node Splitting



Split a node of degree $b+1$ into two nodes of degrees $\lfloor \frac{b+1}{2} \rfloor$ and $\lceil \frac{b+1}{2} \rceil$.

- We have $a = \lfloor \frac{2a}{2} \rfloor \le \lfloor \frac{b+1}{2} \rfloor \le \lceil \frac{b+1}{2} \rceil \le \frac{b}{2} + 1 \le b$.

If the parent now has degree $b+1$, split the parent recursively.

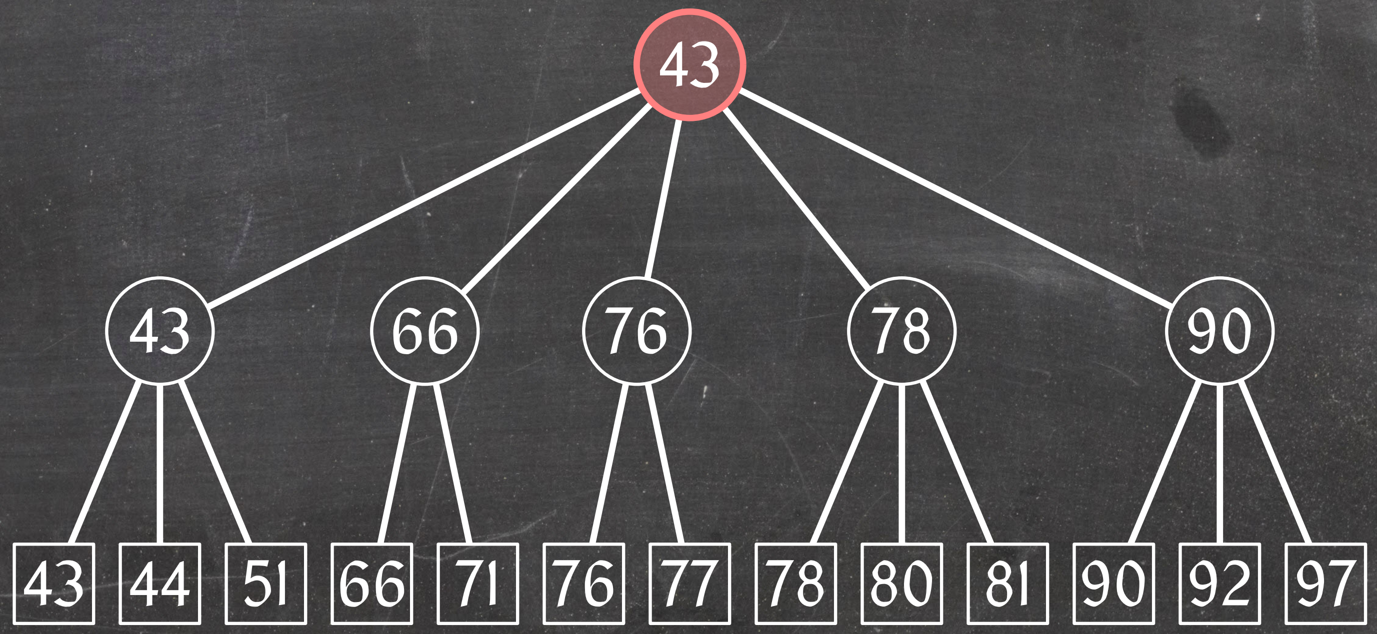
Cost per node split: $O(b) = O(1)$

At most one node split per level.

Insertion cost: $O(\lg n)$

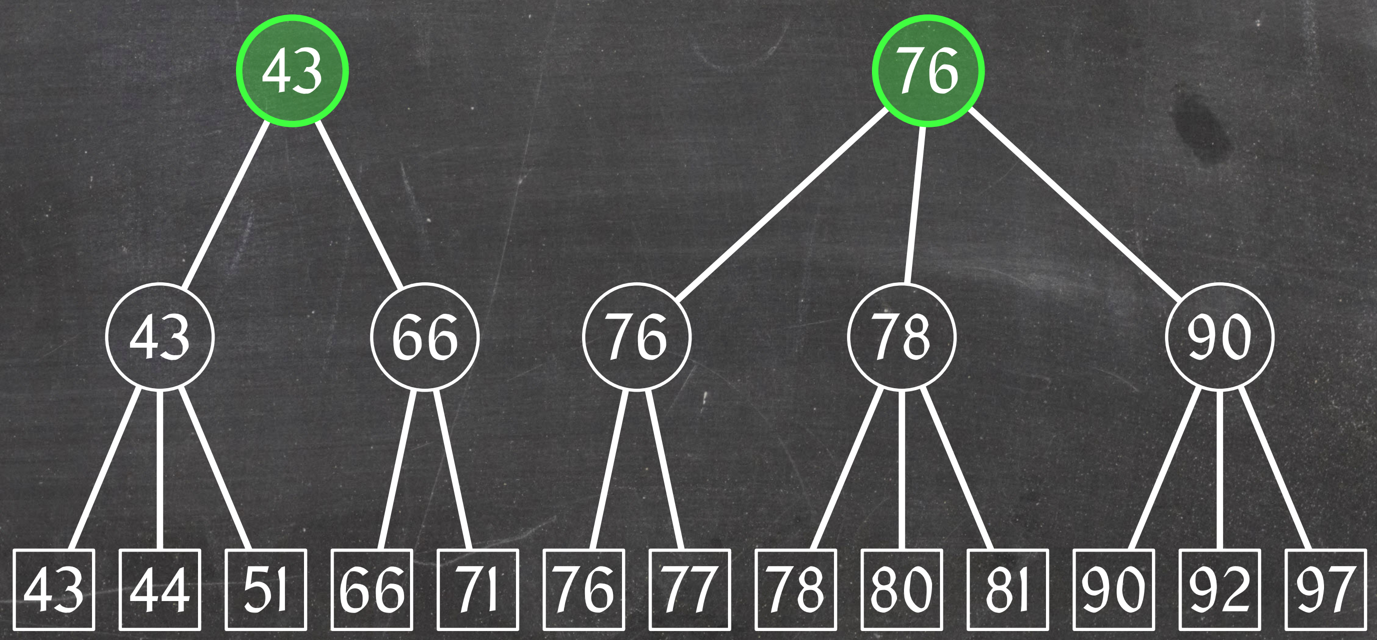
Splitting the Root

What do we do when we split the root?



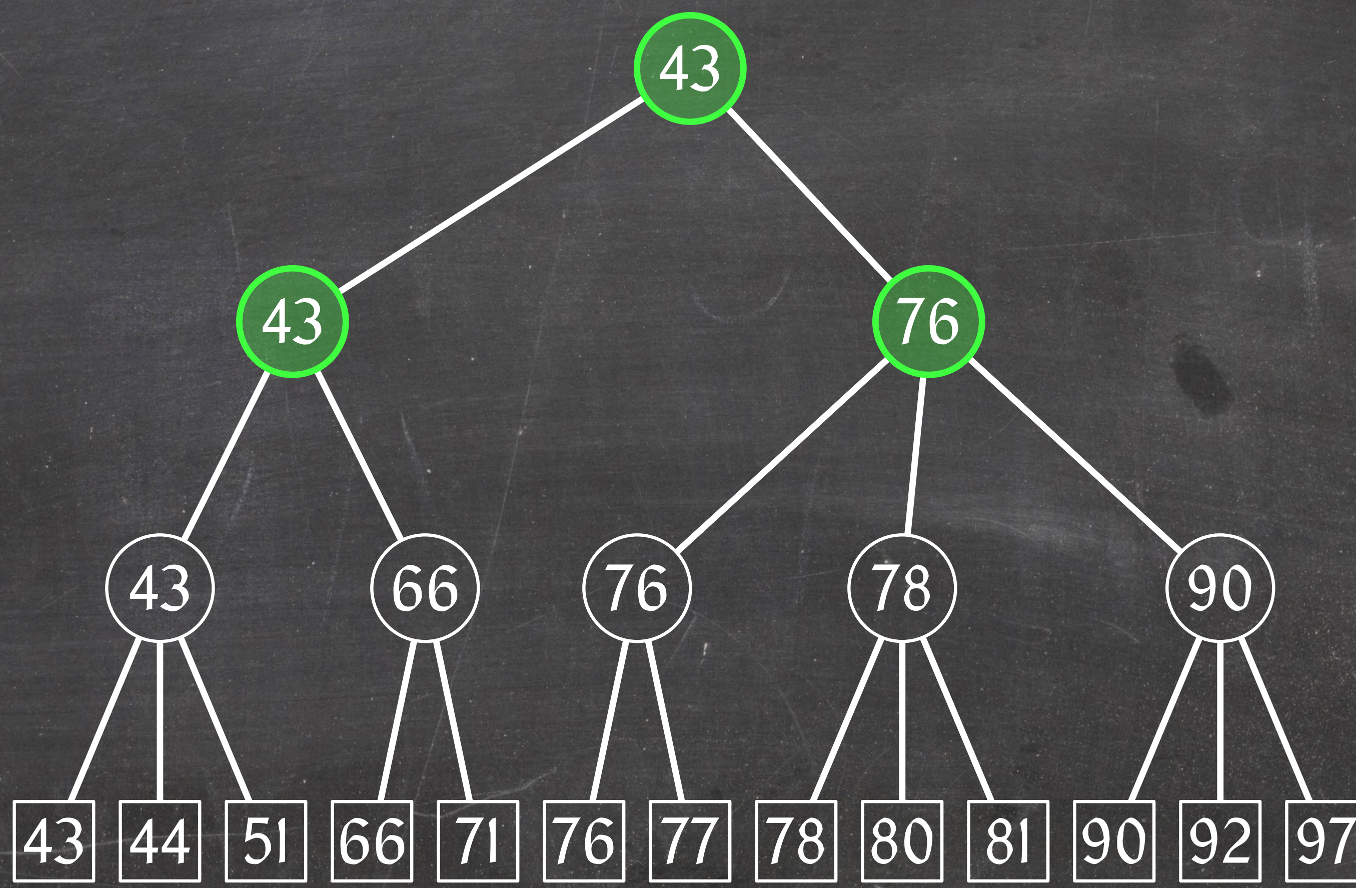
Splitting the Root

What do we do when we split the root?



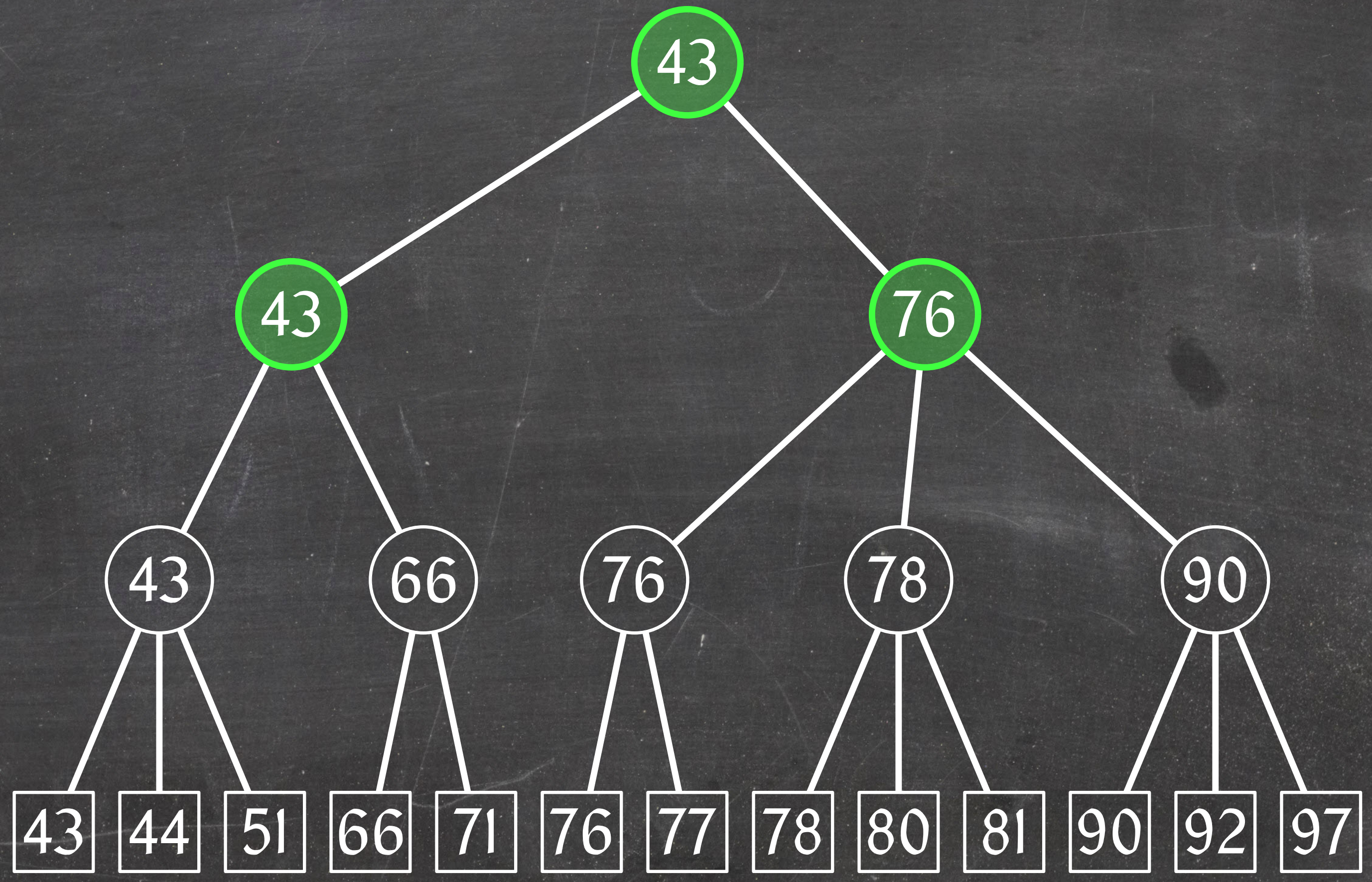
Splitting the Root

What do we do when we split the root?



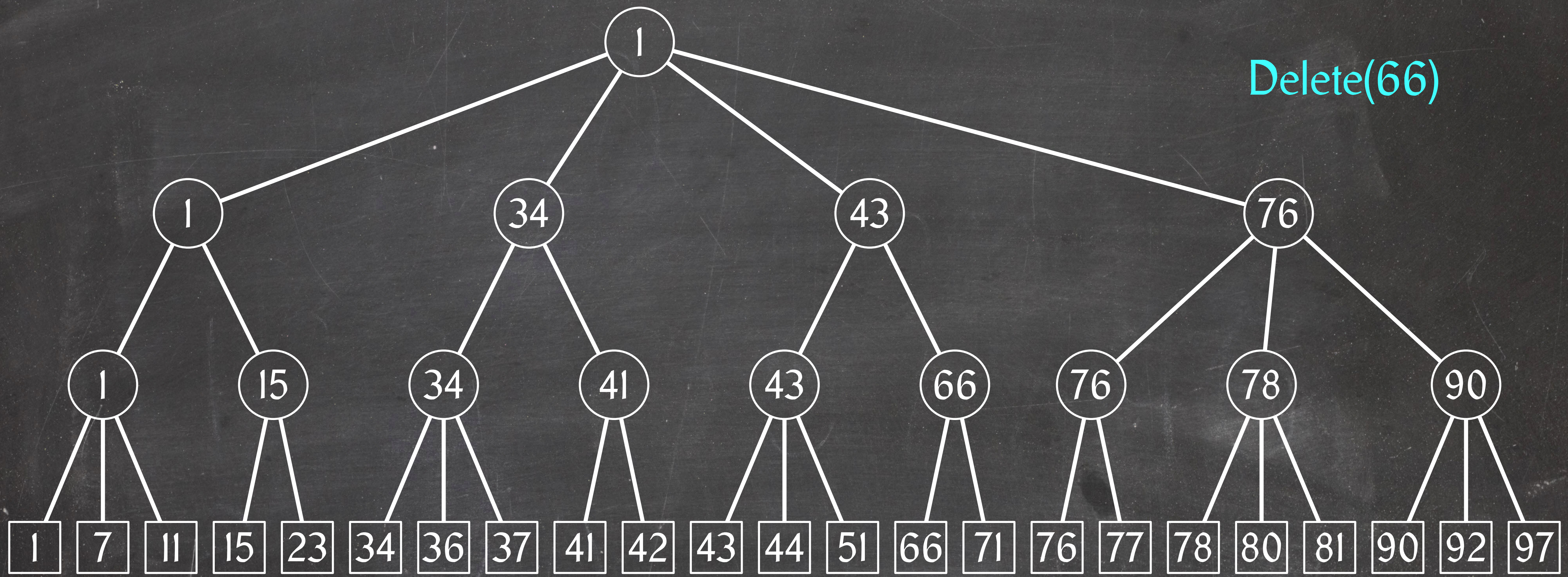
Splitting the Root

What do we do when we split the root?

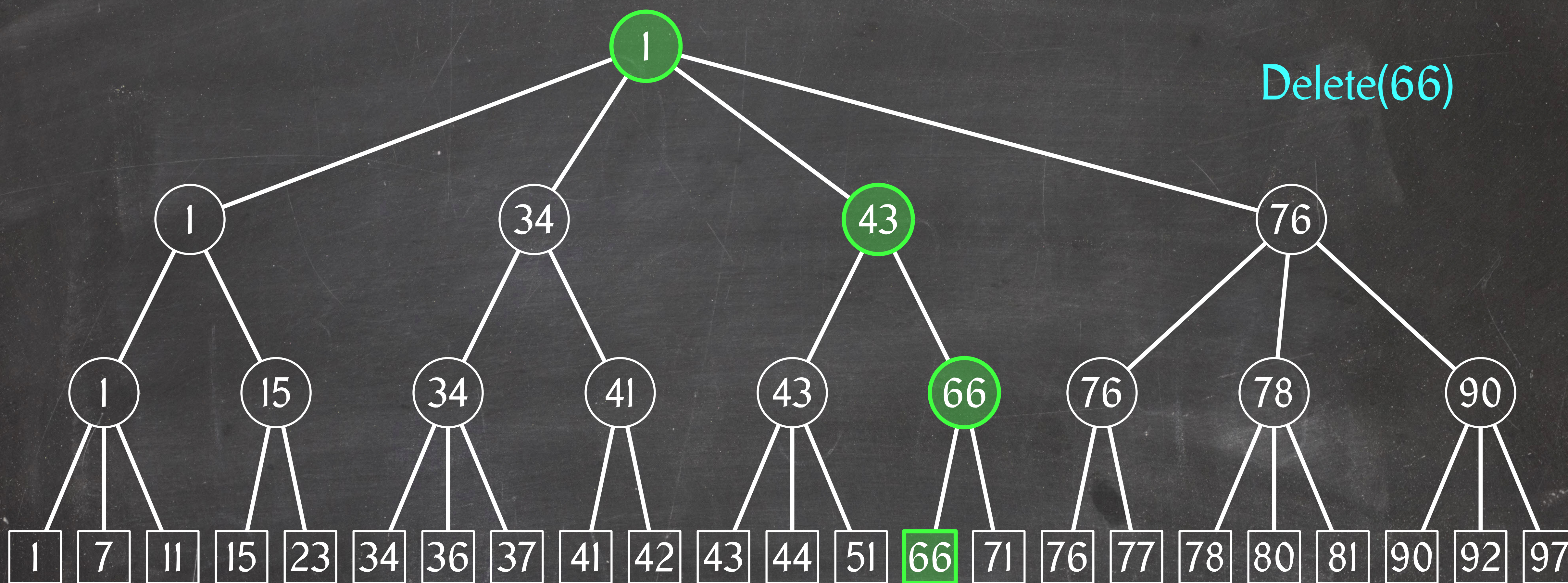


Note: This is exactly why we have to allow the root to have degree less than a.

Delete Operation

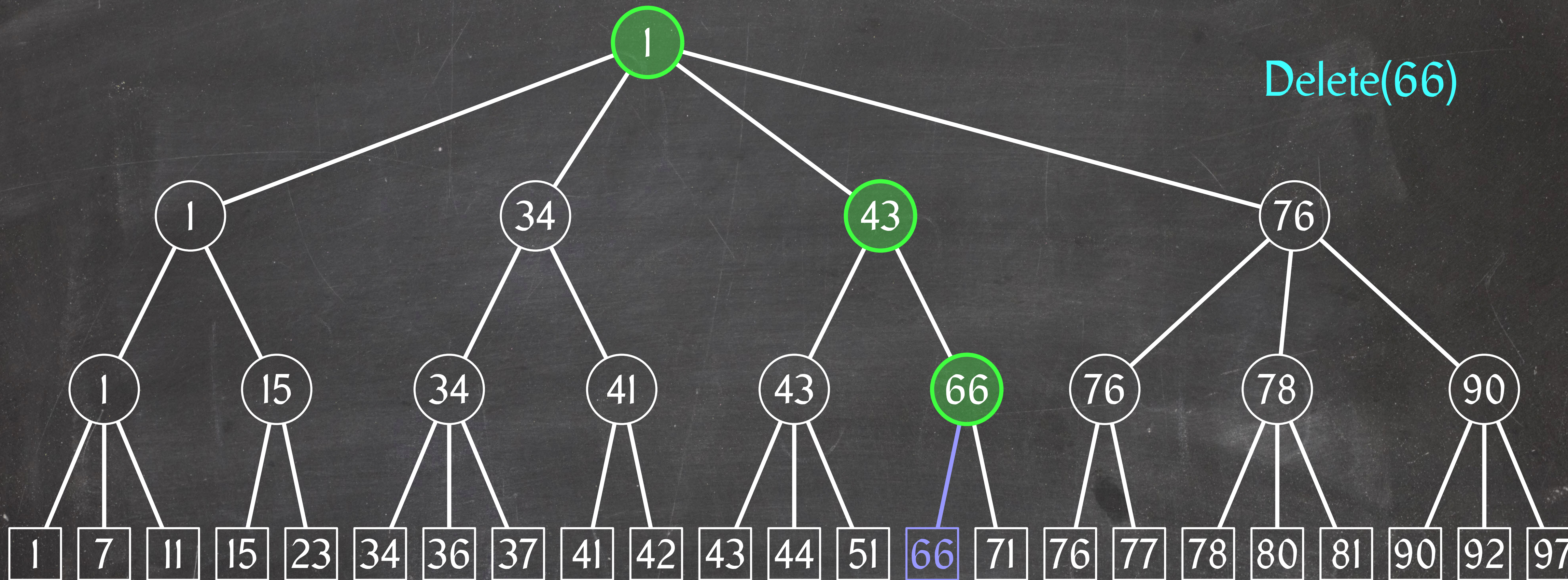


Delete Operation



- Find the leaf storing x.

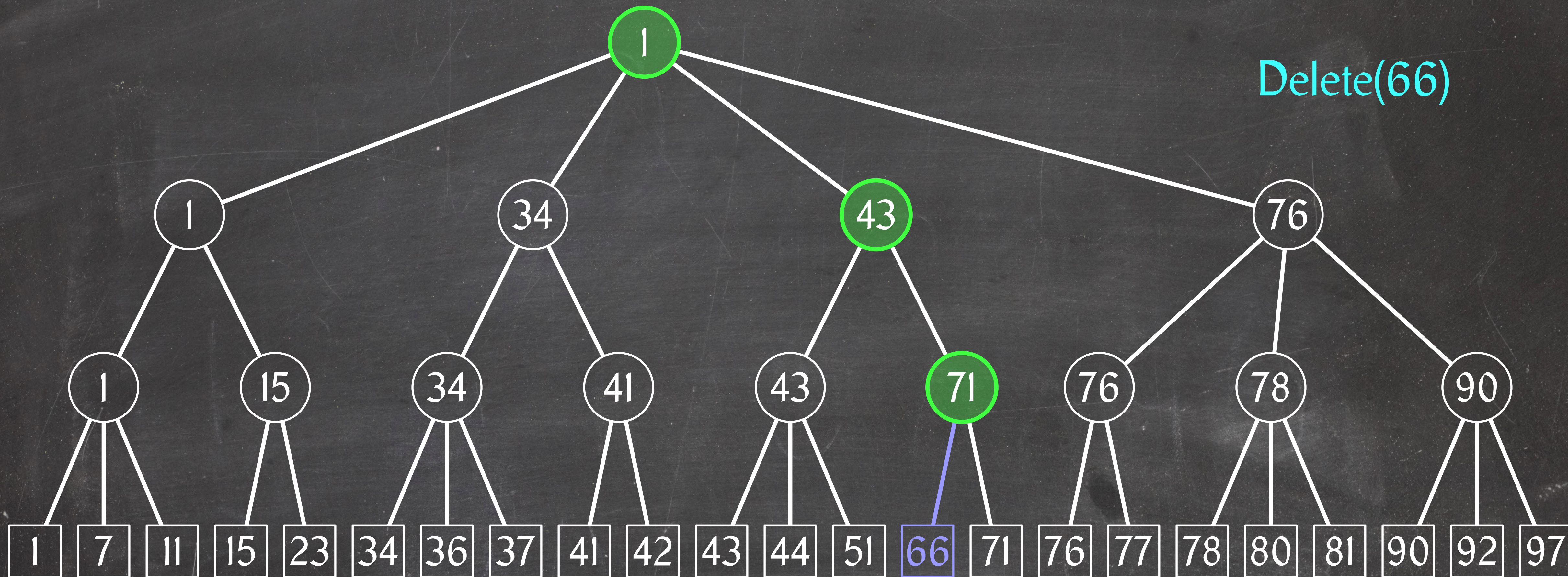
Delete Operation



Delete(66)

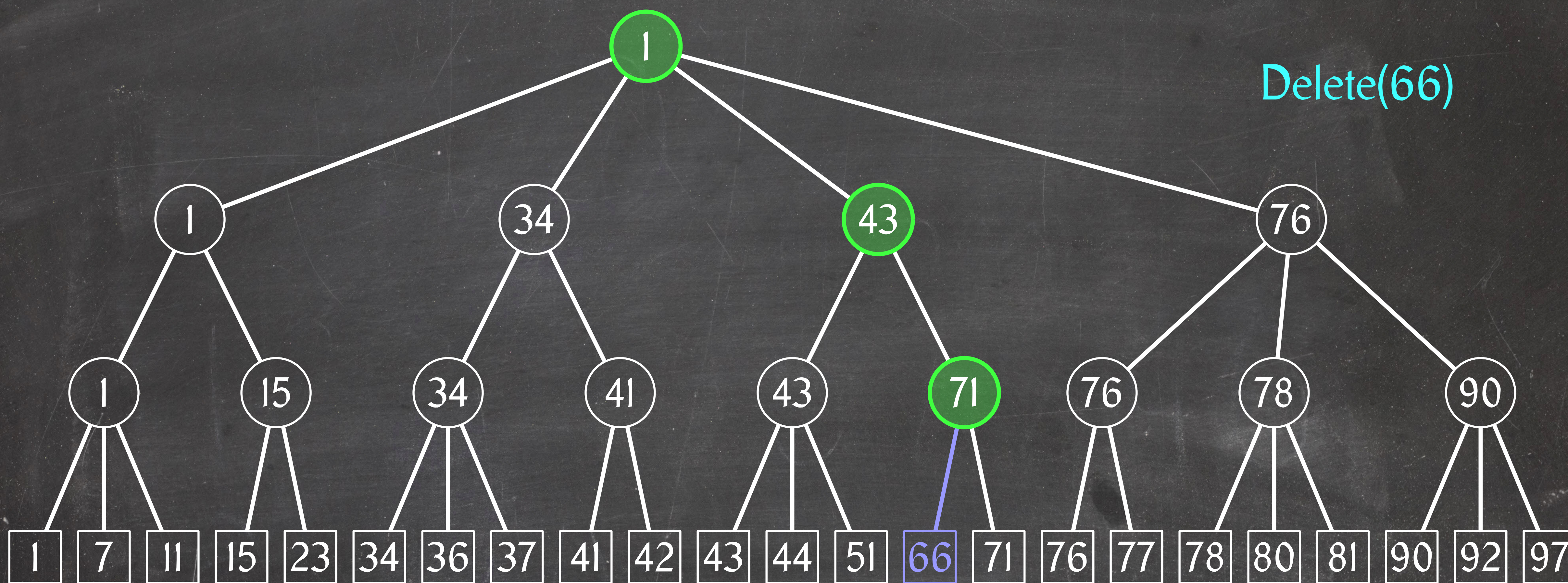
- Find the leaf storing x.
- Delete it.

Delete Operation



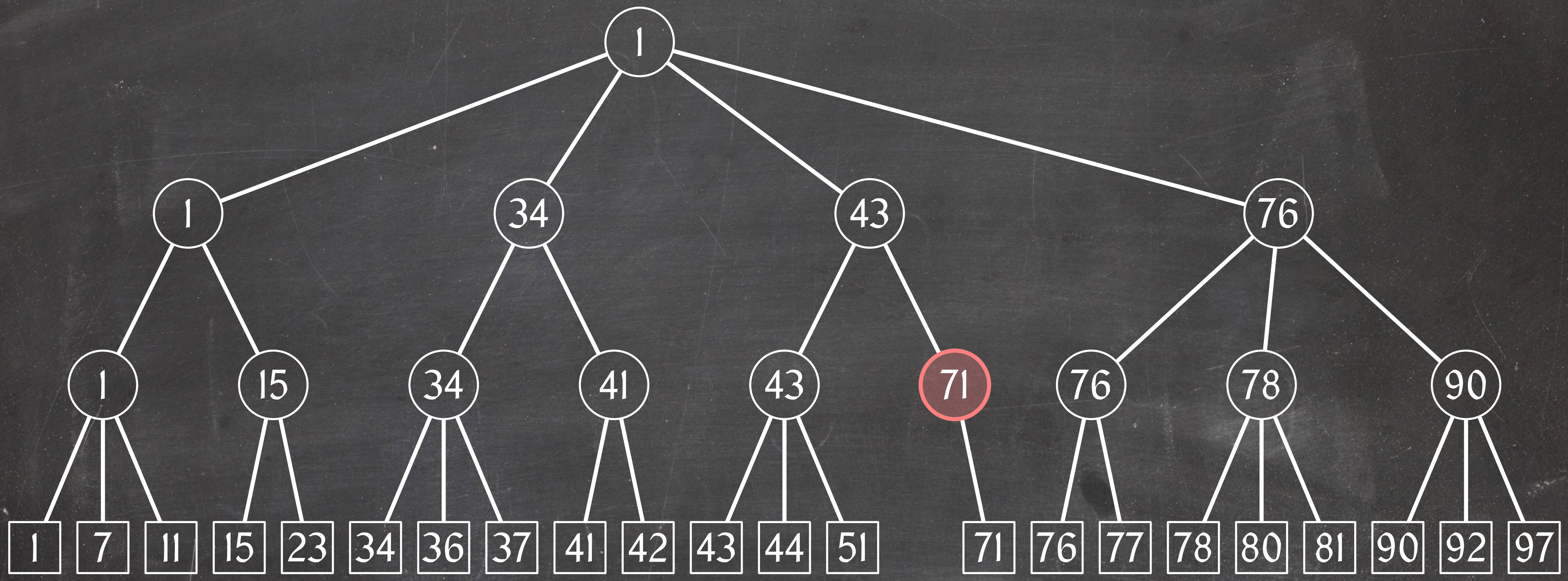
- Find the leaf storing x.
- Delete it.
- (Update the keys of its ancestors.)

Delete Operation

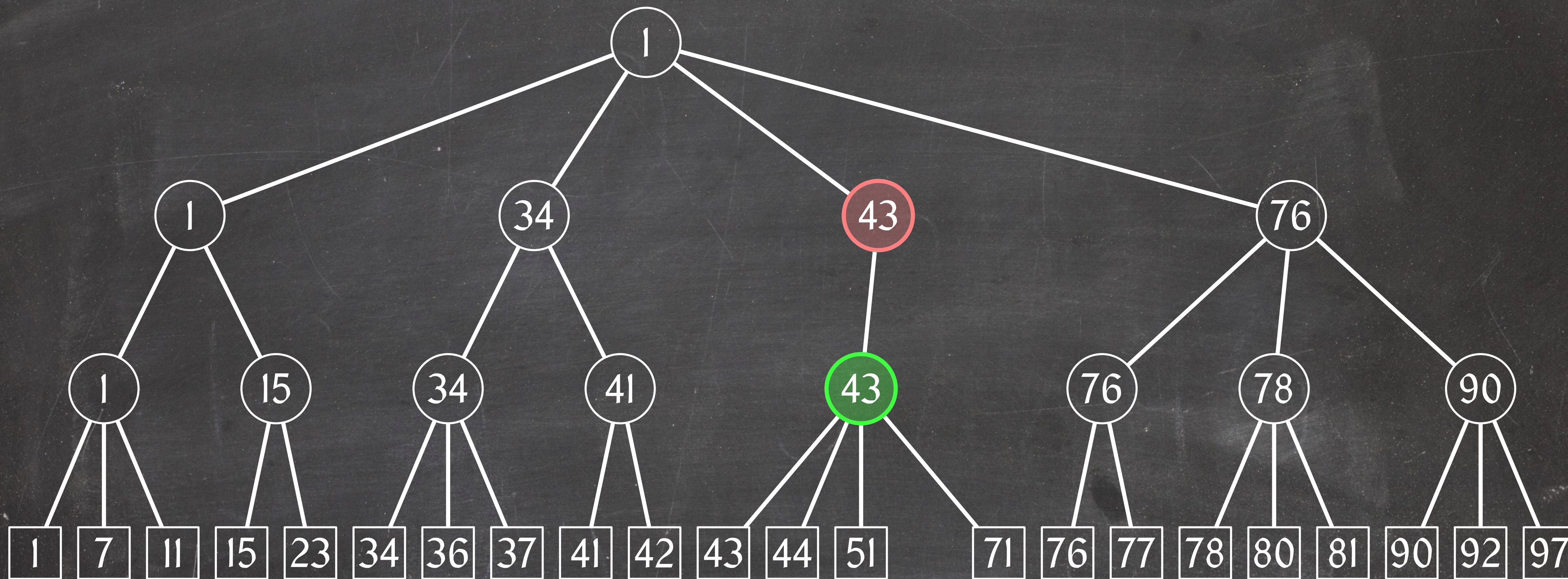


- Find the leaf storing x.
- Delete it.
- (Update the keys of its ancestors.)
- Rebalance. How?

Node Fusion

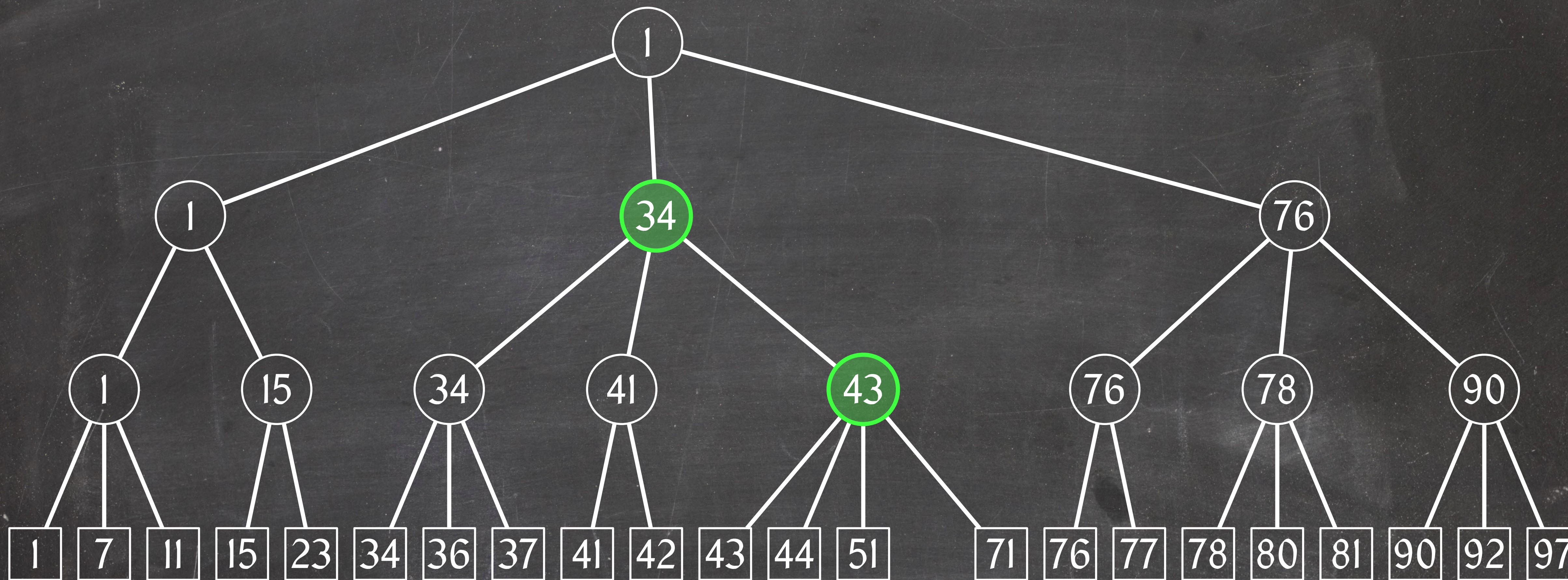

1
1
34
43
76
1
15
34
41
43
71
76
78
90
1
7
11
15
23
34
36
37
41
42
43
44
51
71
76
77
78
80
81
90
92
97

Node Fusion



Fuse a node of degree $a - 1$ with one of its neighbours.

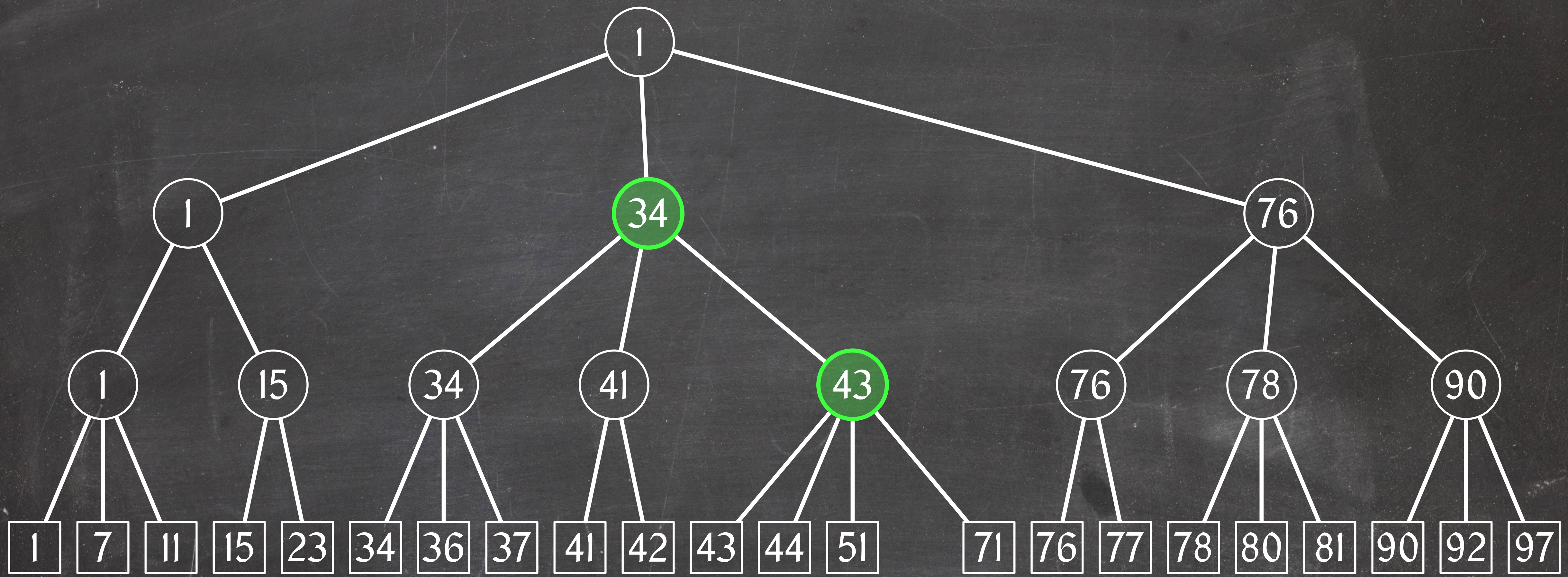
Node Fusion



Fuse a node of degree a – 1 with one of its neighbours.

If the parent now has degree a – 1, recurse.

Node Fusion

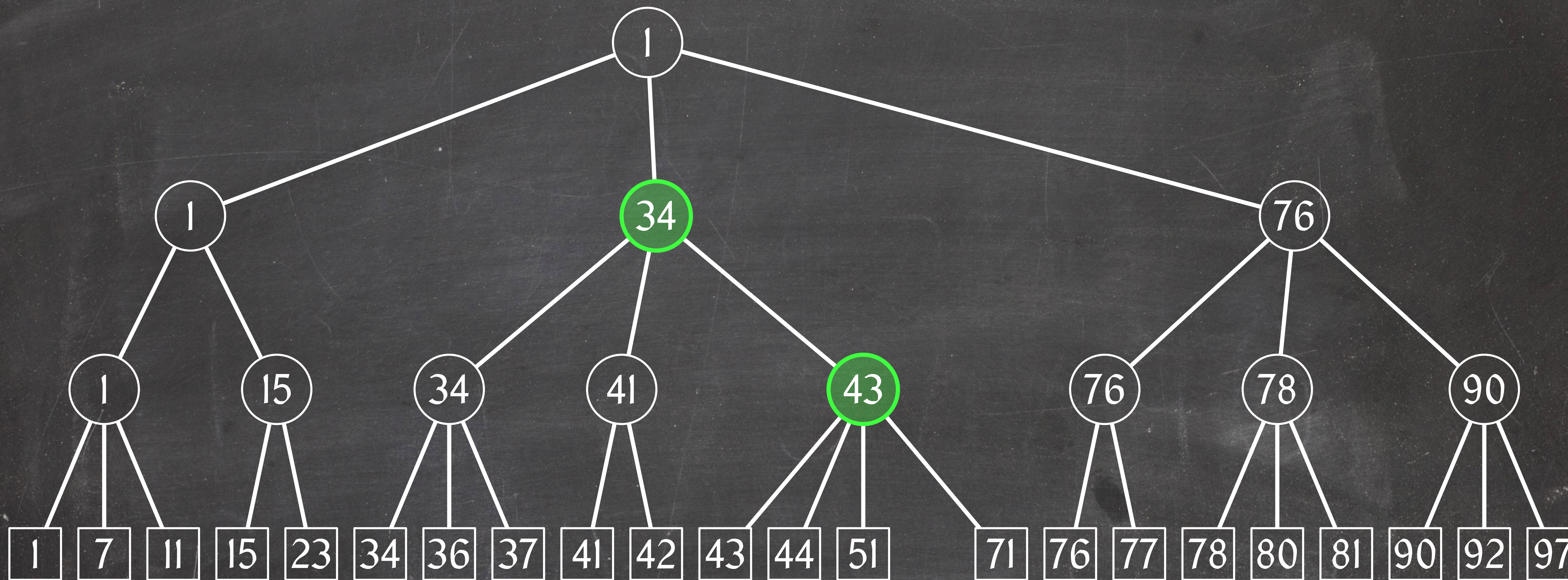


Fuse a node of degree $a - 1$ with one of its neighbours.

If the parent now has degree $a - 1$, recurse.

Cost per node fusion: $O(b) = O(1)$

Node Fusion



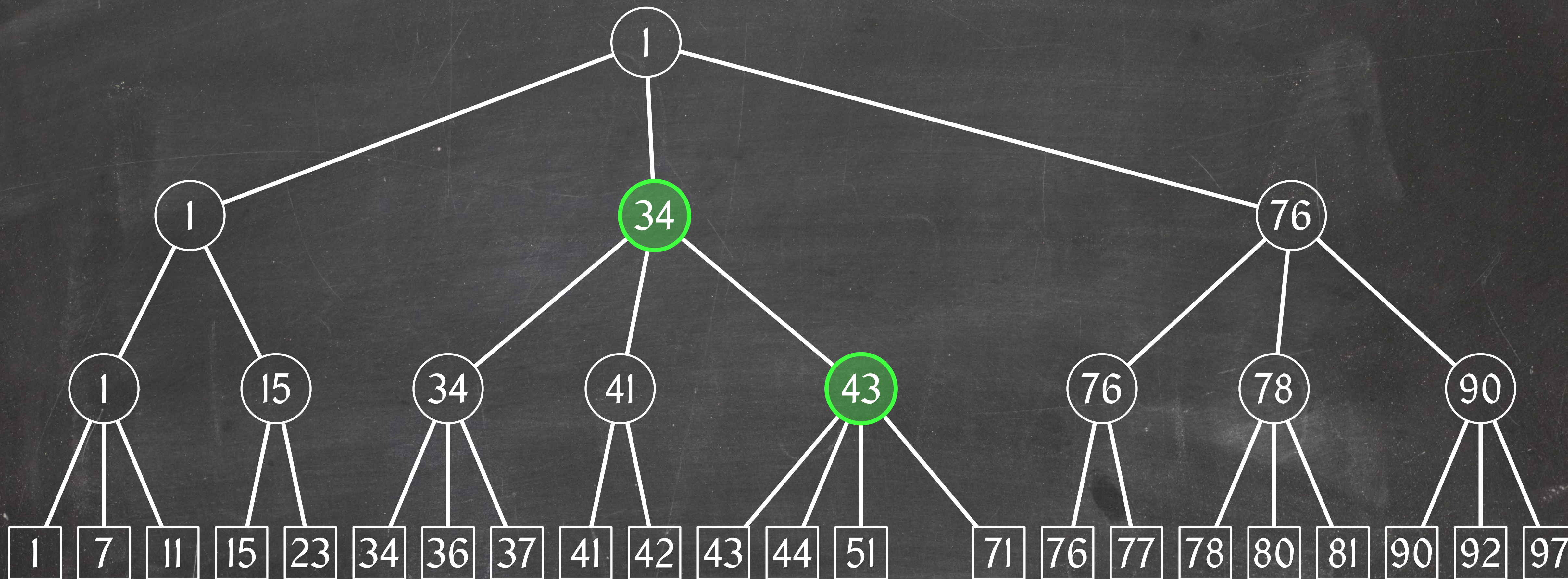
Fuse a node of degree $a - 1$ with one of its neighbours.

If the parent now has degree $a - 1$, recurse.

Cost per node fusion: $O(b) = O(1)$

At most one node fusion per level.

Node Fusion



Fuse a node of degree $a - 1$ with one of its neighbours.

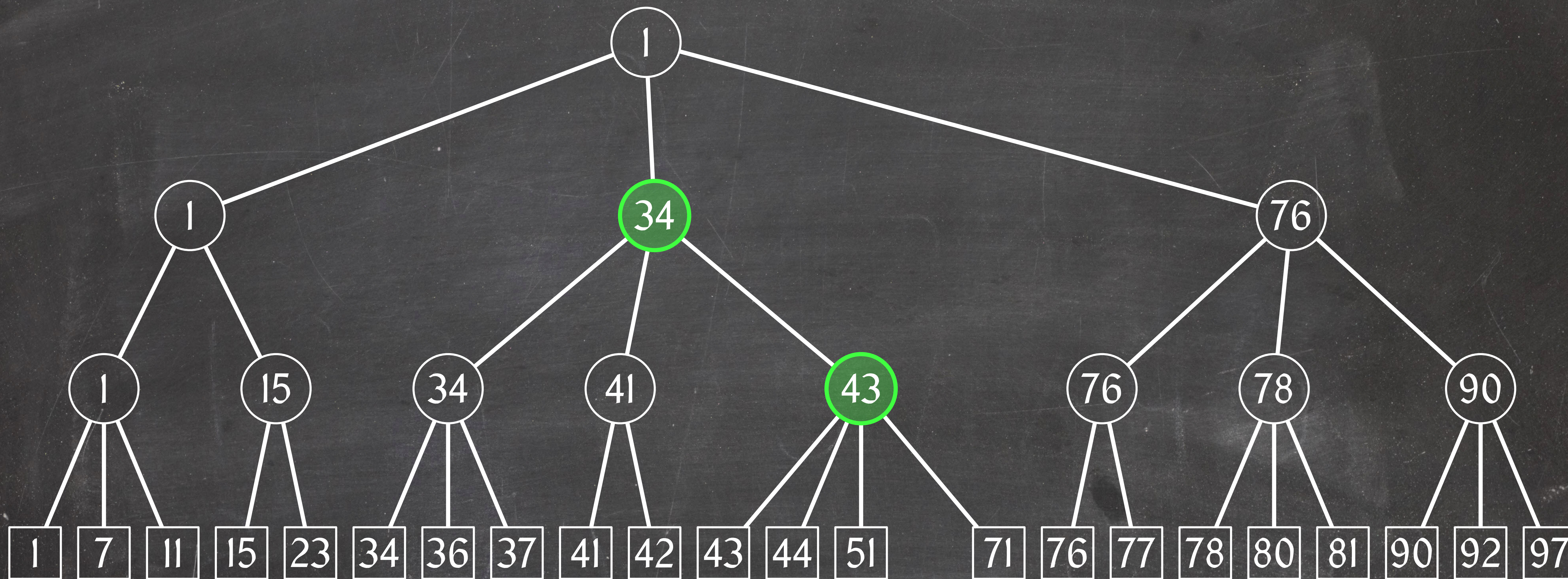
If the parent now has degree $a - 1$, recurse.

Cost per node fusion: $O(b) = O(1)$

At most one node fusion per level.

Deletion cost: $O(\lg n)$

Node Fusion



Fuse a node of degree $a - 1$ with one of its neighbours.

If the parent now has degree $a - 1$, recurse.

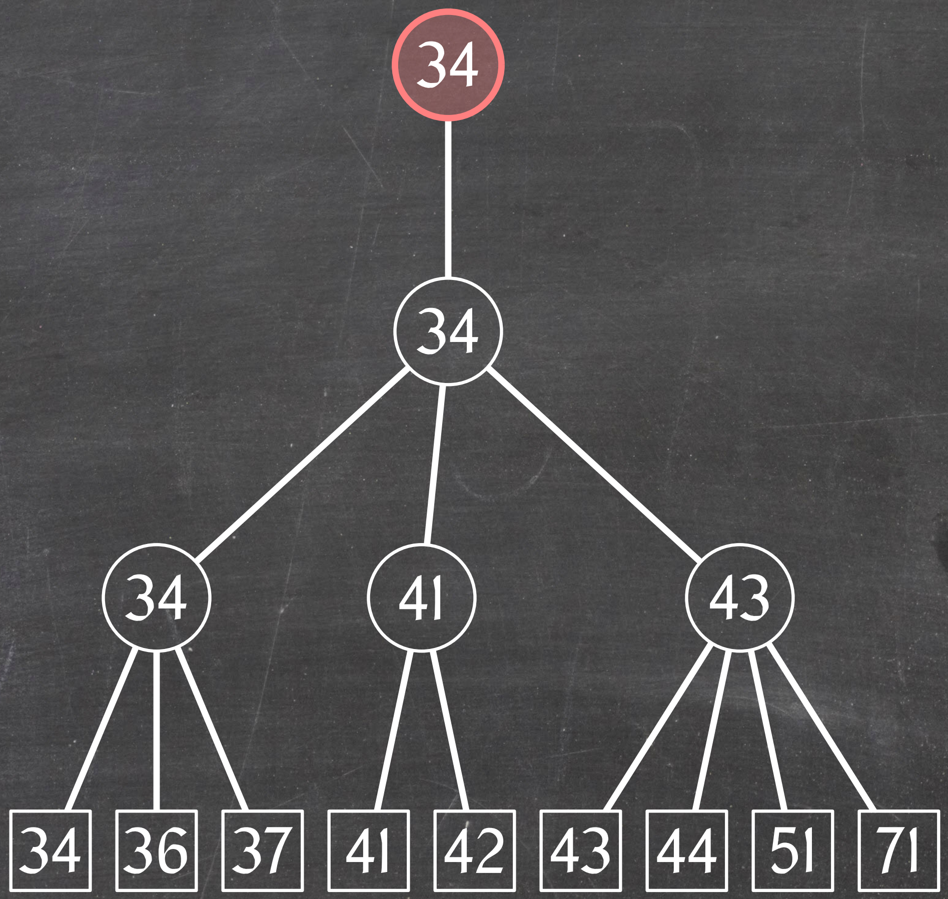
Can we always do this?

Cost per node fusion: $O(b) = O(1)$

At most one node fusion per level.

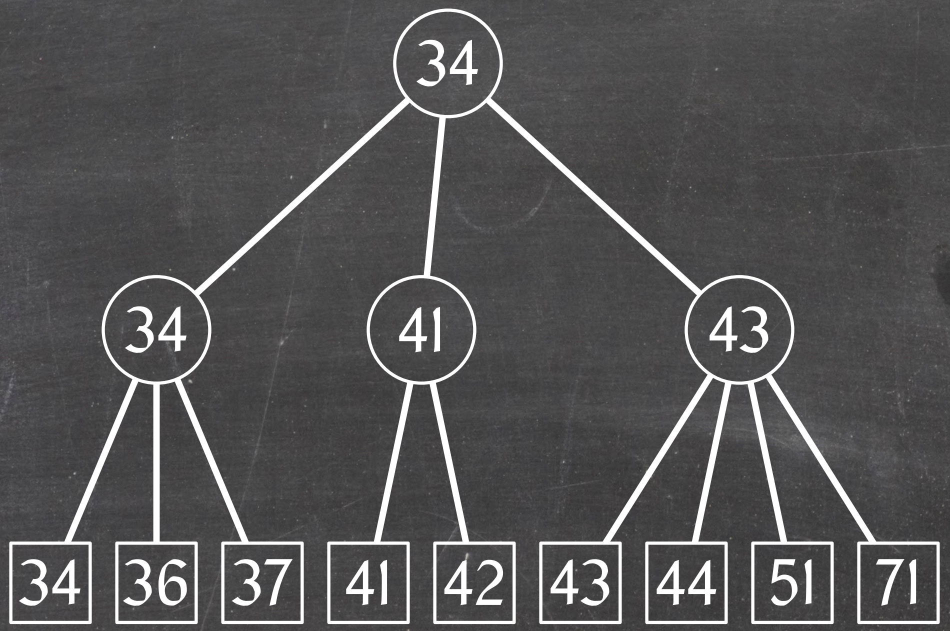
Deletion cost: $O(\lg n)$

Fusing Children of the Root



What do we do if the root's degree becomes 1?

Fusing Children of the Root

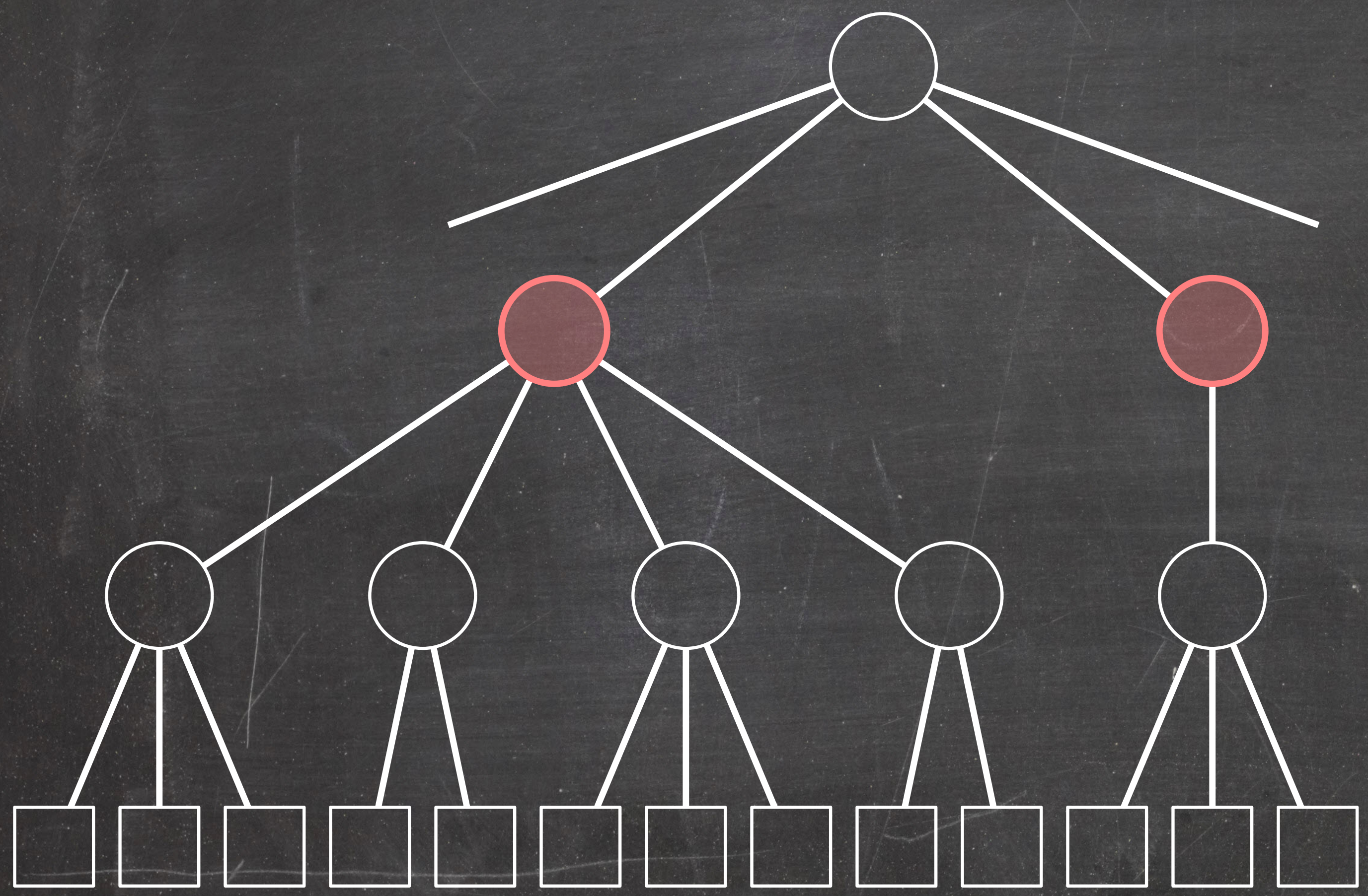


What do we do if the root's degree becomes 1?

We remove the root.

Node Sharing

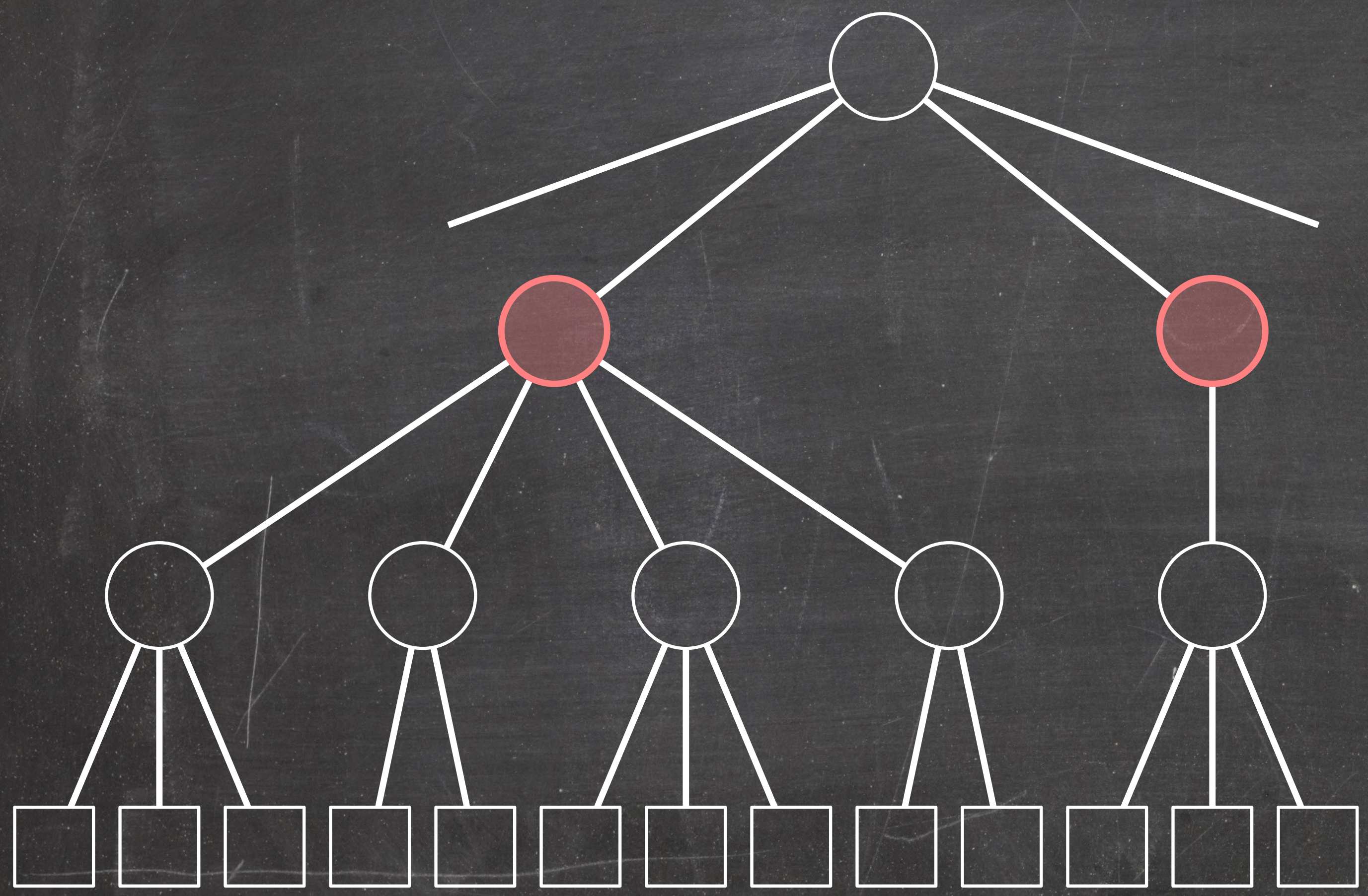
What if a node v and its sibling together have more than b children?



Node Sharing

What if a node v and its sibling together have more than b children?

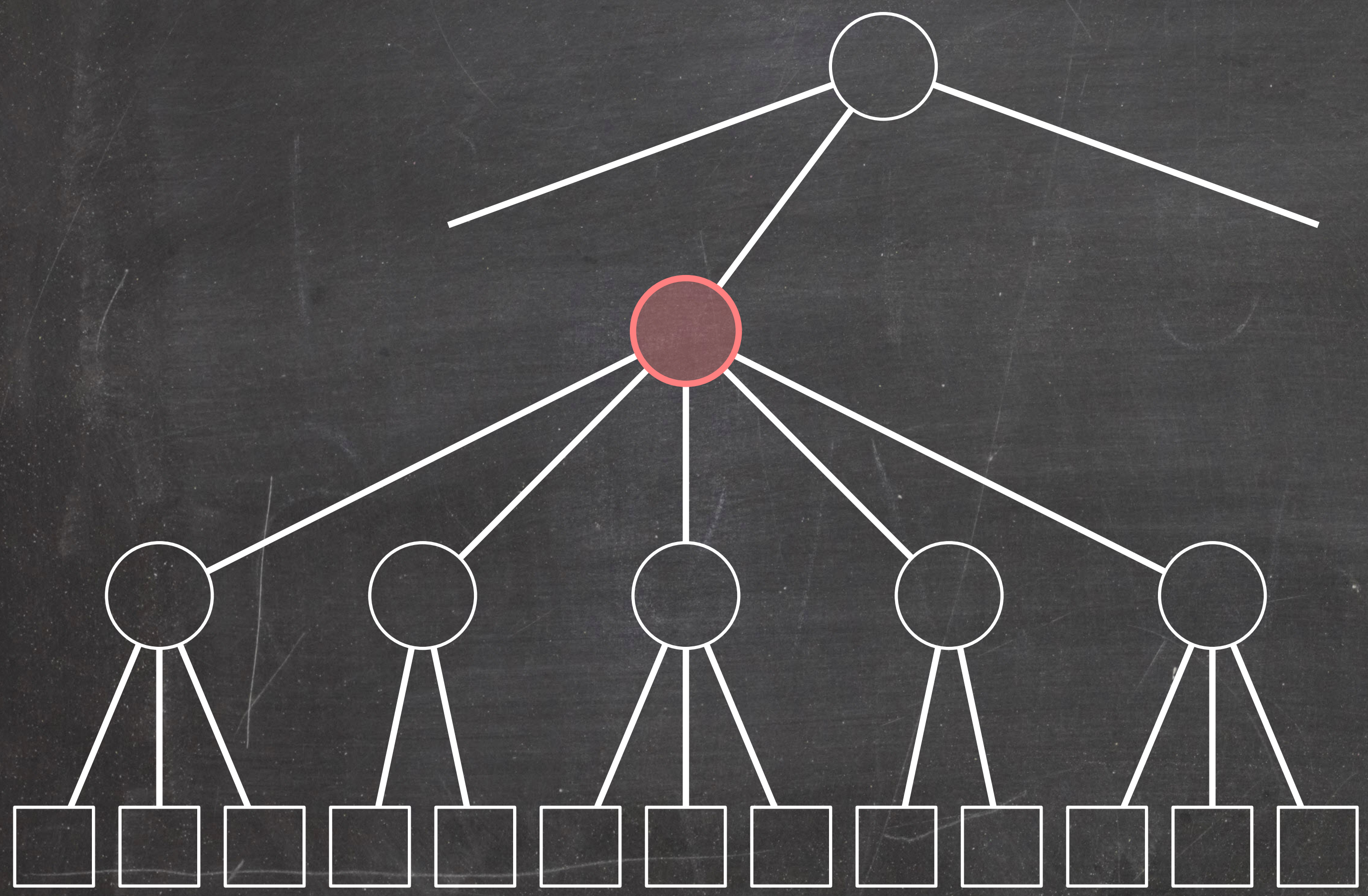
We fuse and then split (essentially borrowing children from v's sibling).



Node Sharing

What if a node v and its sibling together have more than b children?

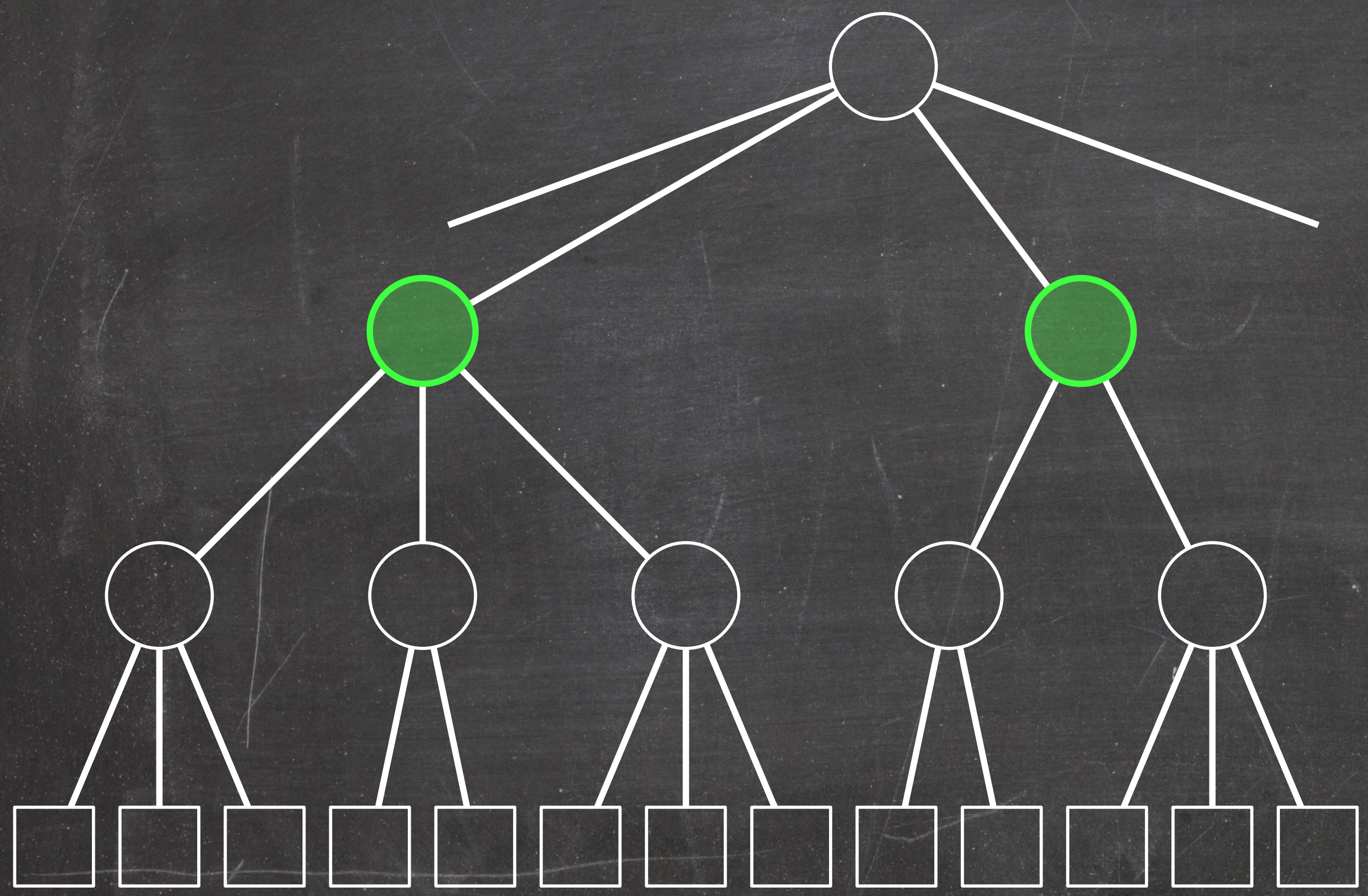
We fuse and then split (essentially borrowing children from v's sibling).



Node Sharing

What if a node v and its sibling together have more than b children?

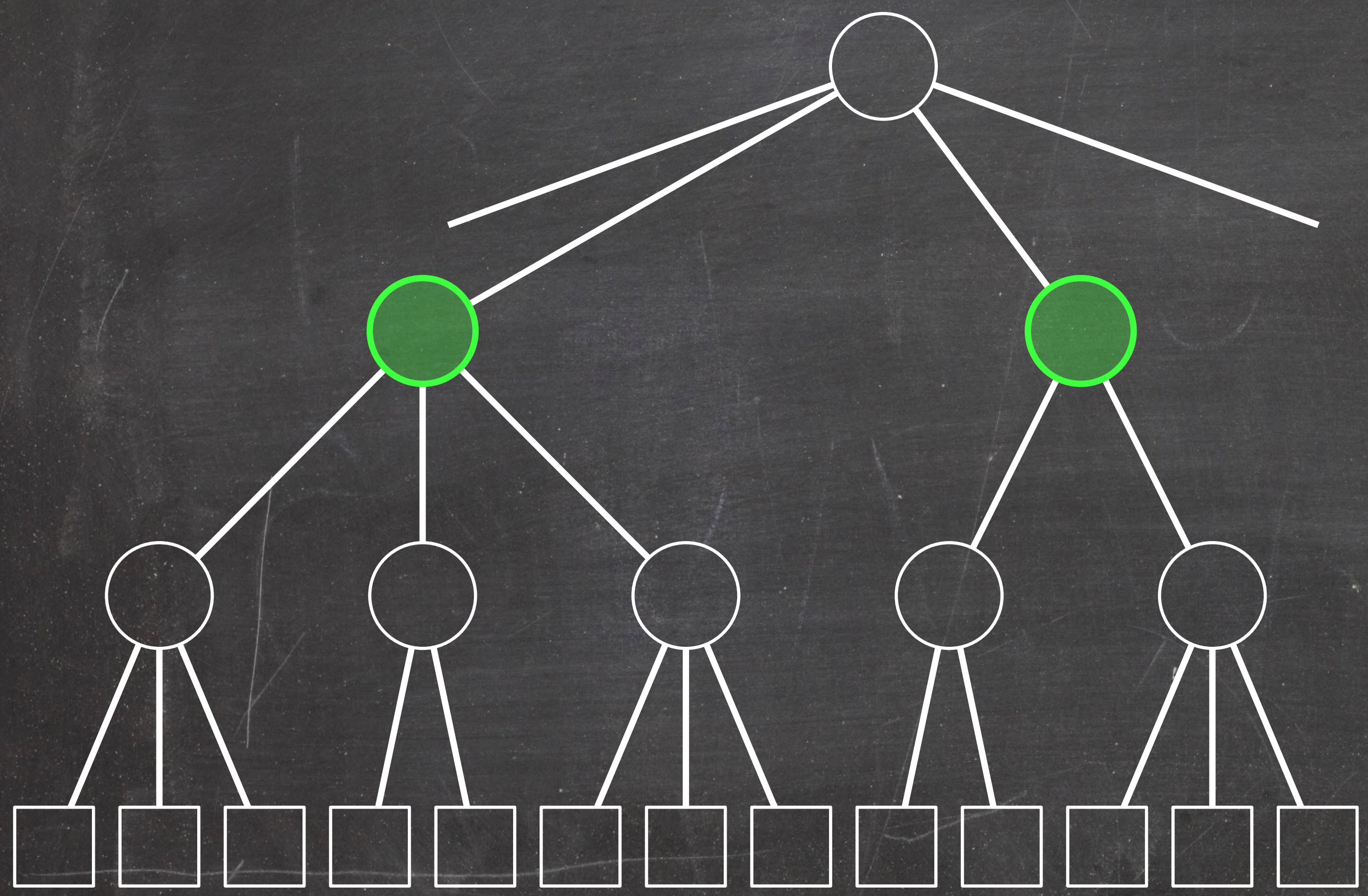
We fuse and then split (essentially borrowing children from v's sibling).



Node Sharing

What if a node v and its sibling together have more than b children?

We fuse and then split (essentially borrowing children from v's sibling).



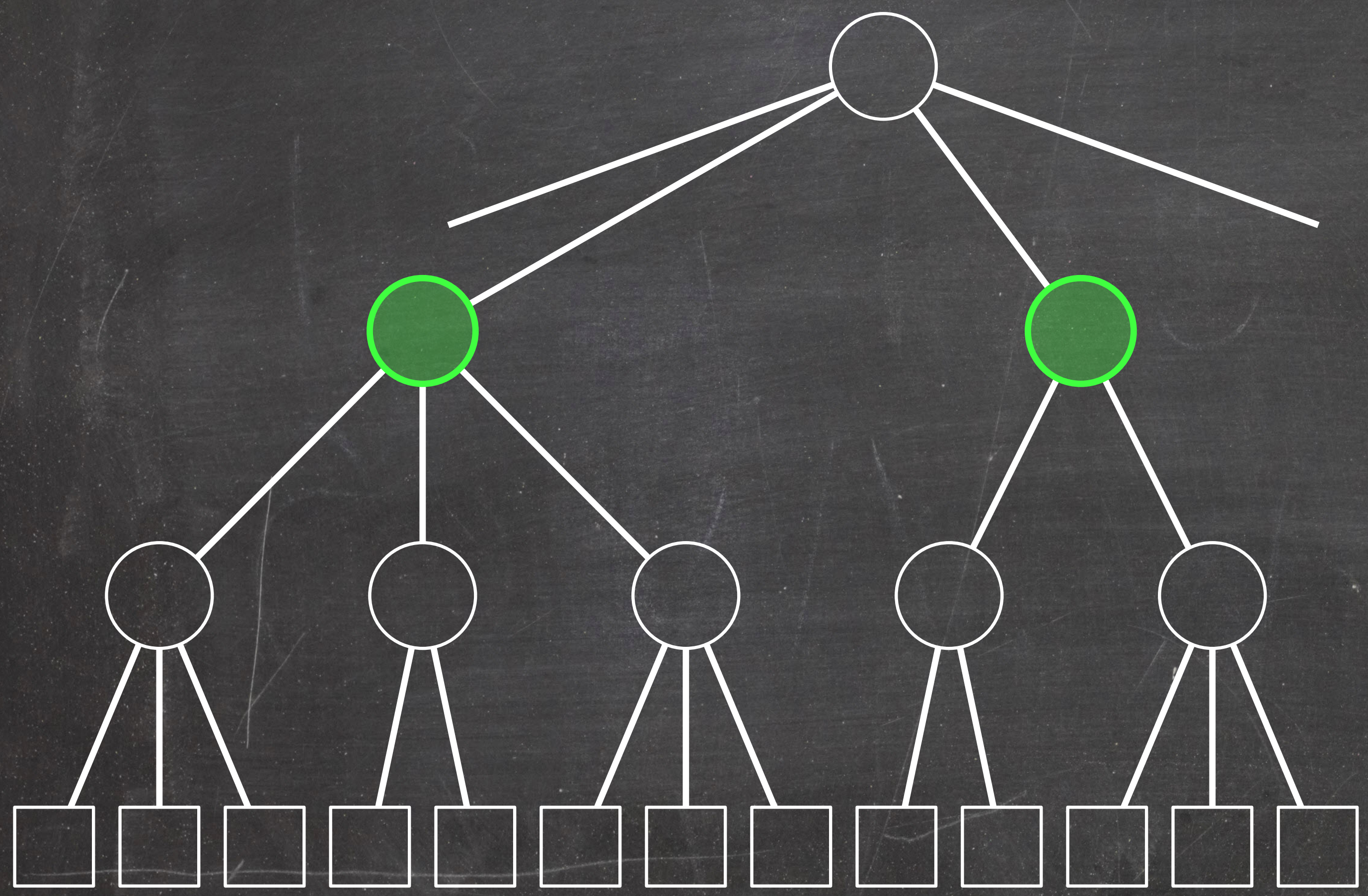
We have $\left\lfloor \frac{b+1}{2} \right\rfloor \geq \left\lfloor \frac{2a}{2} \right\rfloor = a$

and $\left\lceil \frac{b+a-1}{2} \right\rceil \leq \left\lceil \frac{2b}{2} \right\rceil = b.$

Node Sharing

What if a node v and its sibling together have more than b children?

We fuse and then split (essentially borrowing children from v's sibling).



We have $\left\lfloor \frac{b+1}{2} \right\rfloor \geq \left\lfloor \frac{2a}{2} \right\rfloor = a$

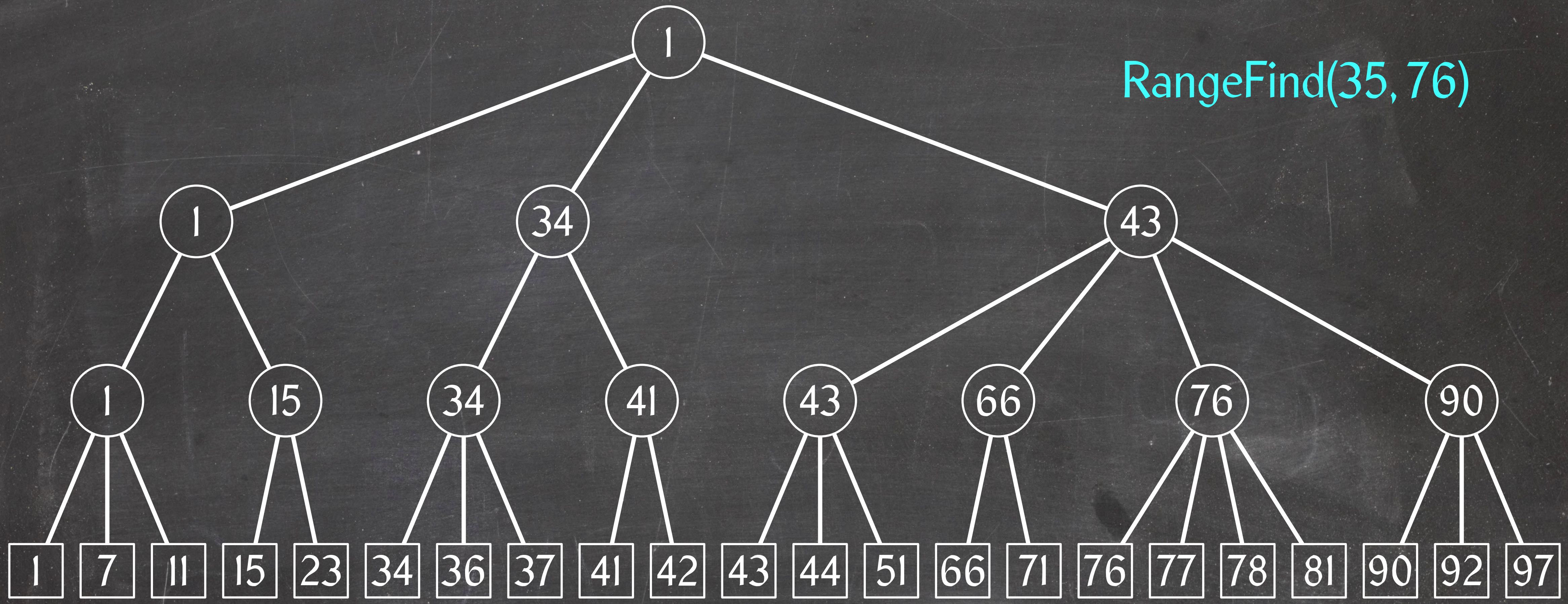
and $\left\lceil \frac{b+a-1}{2} \right\rceil \leq \left\lceil \frac{2b}{2} \right\rceil = b$.

After a fusion followed by a split, the tree is a valid (a, b)-tree again:

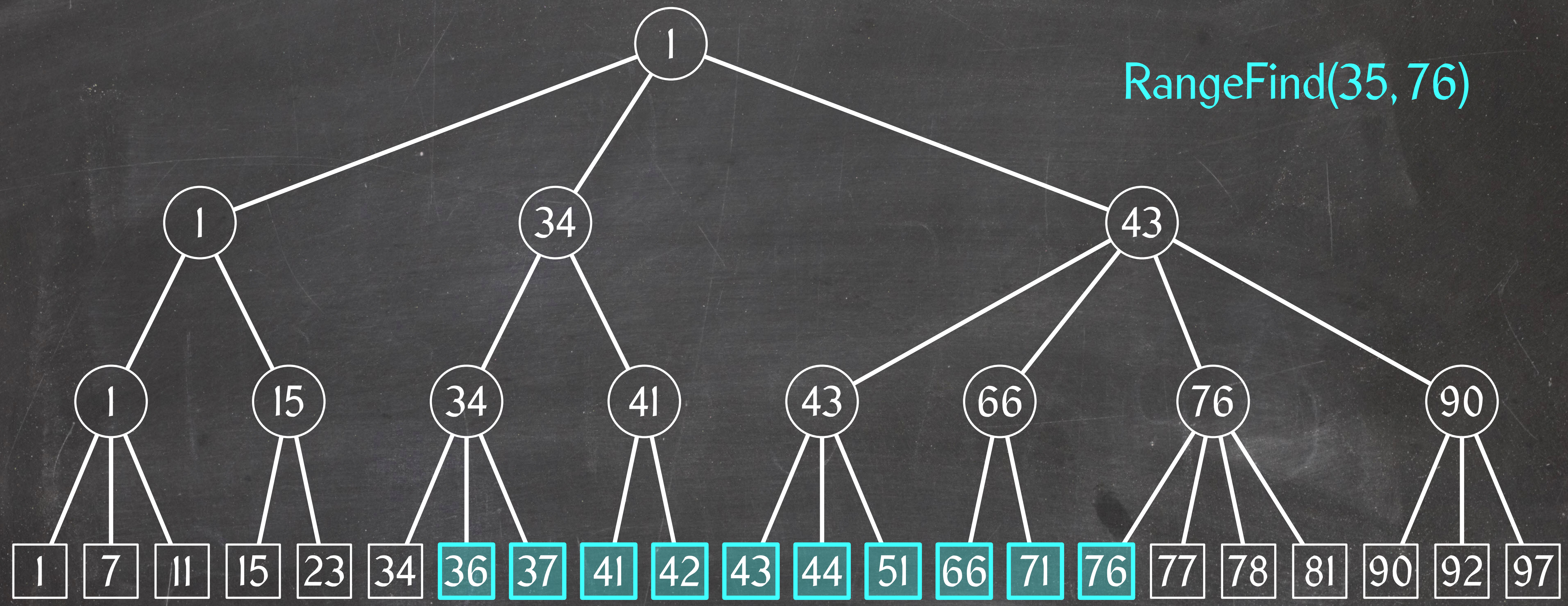
- We just argued that the two nodes we created have degrees between a and b.
- The degree of their parent has not changed.

RangeFind Operation

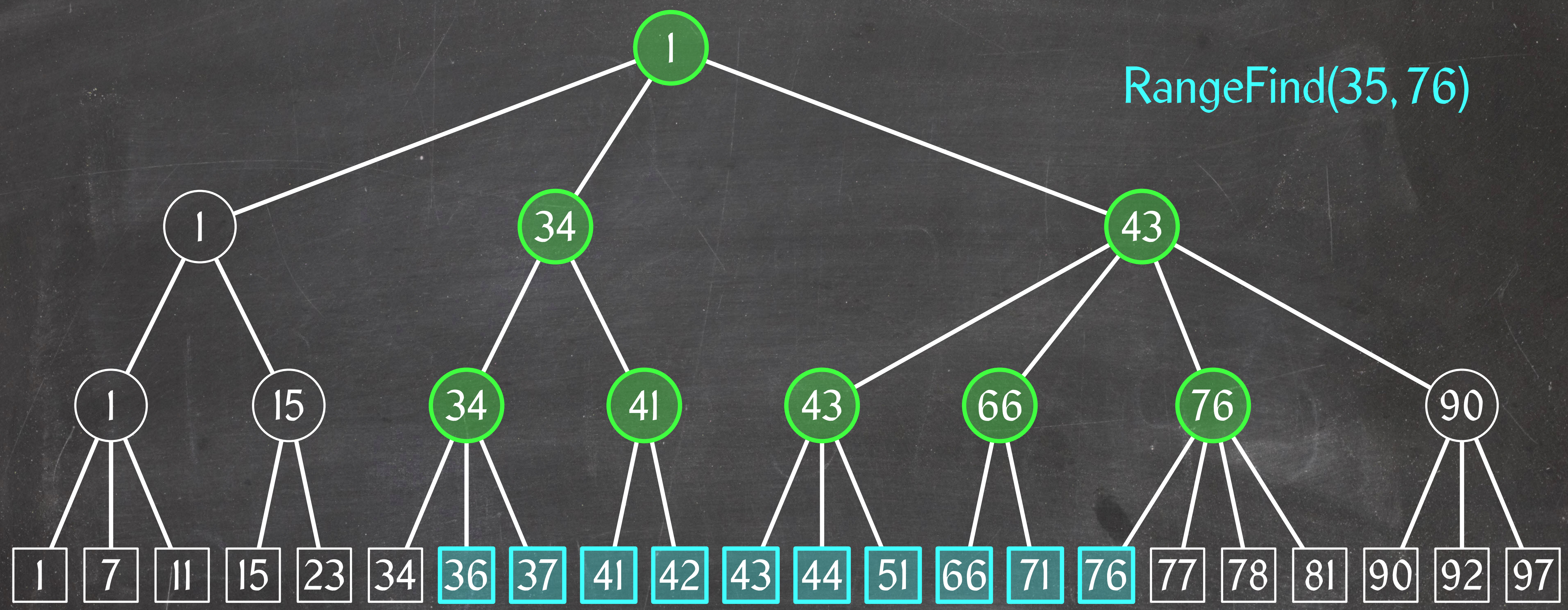
RangeFind(35, 76)



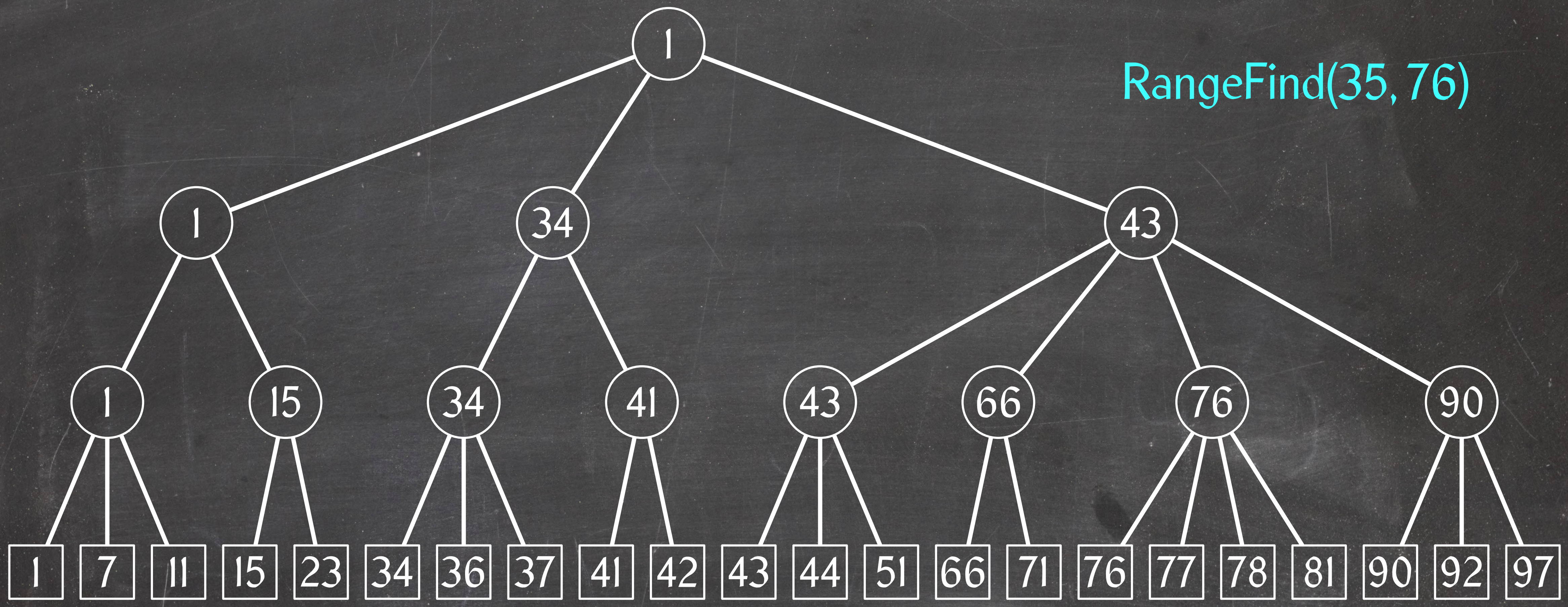
RangeFind Operation



RangeFind Operation



RangeFind Operation

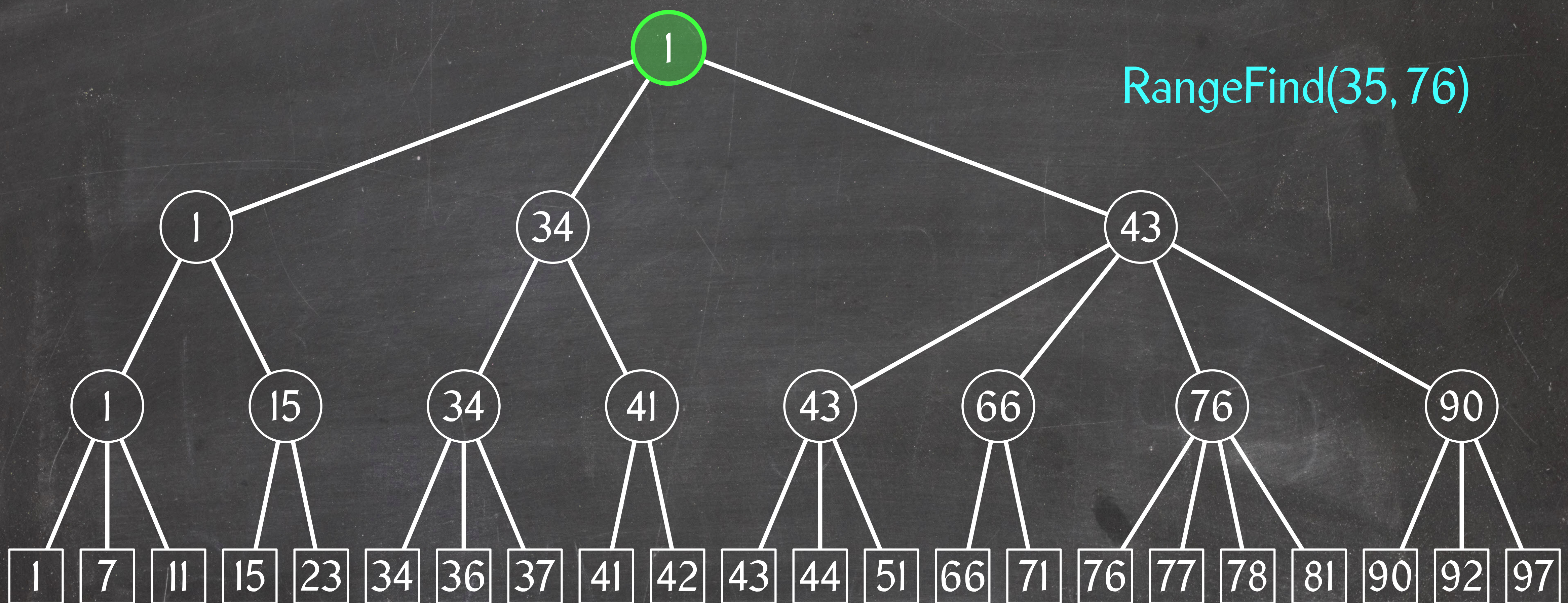


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

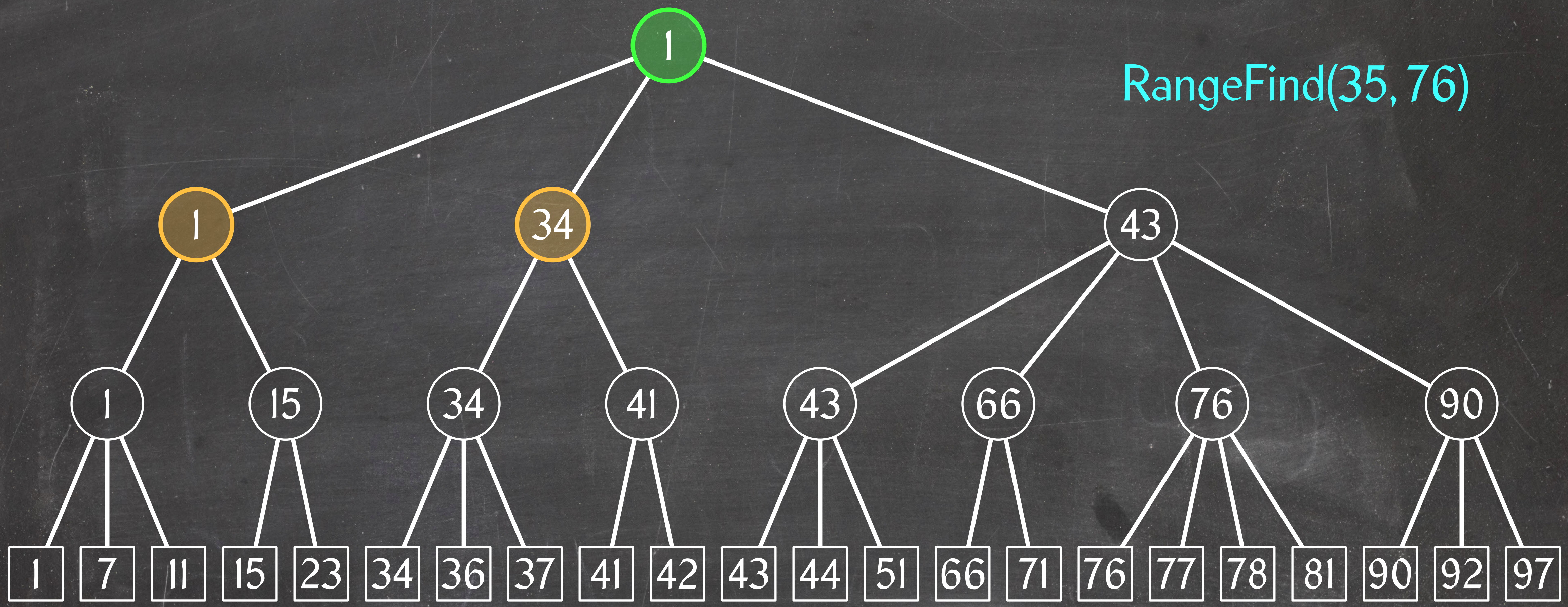


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \le x \le r$.

RangeFind Operation



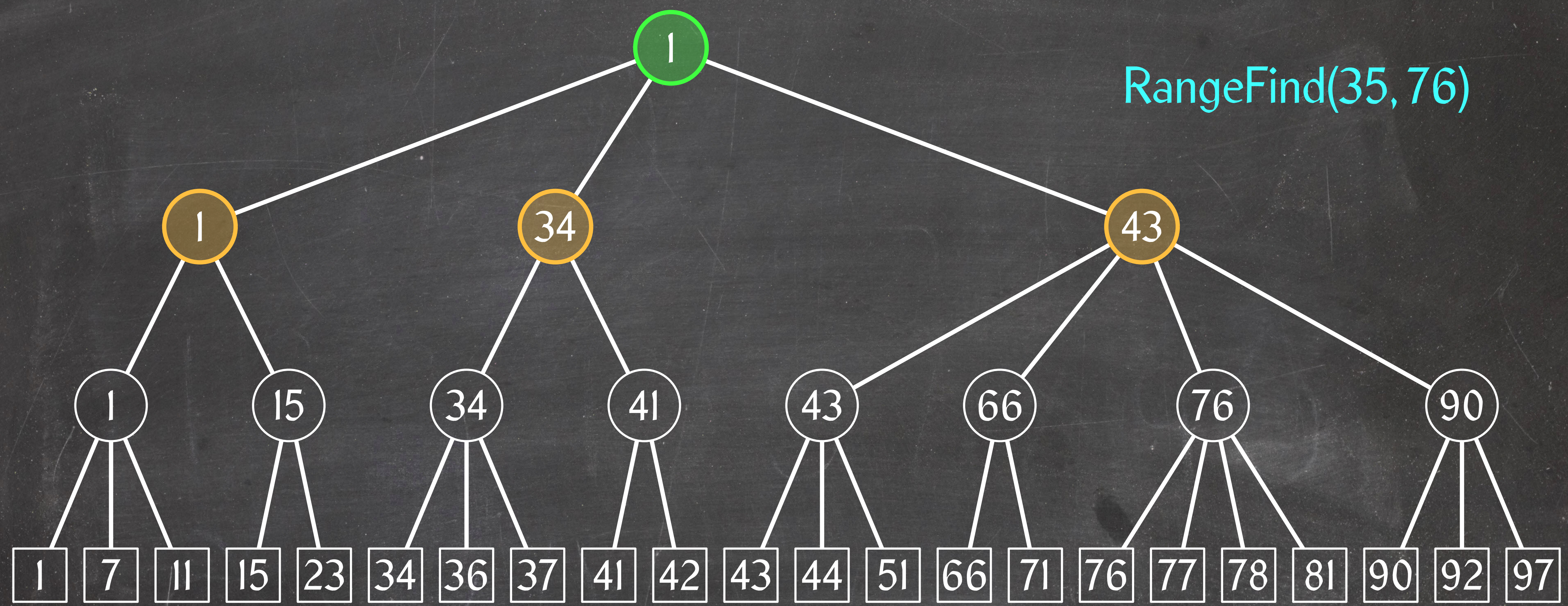
RangeFind(35, 76)

RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

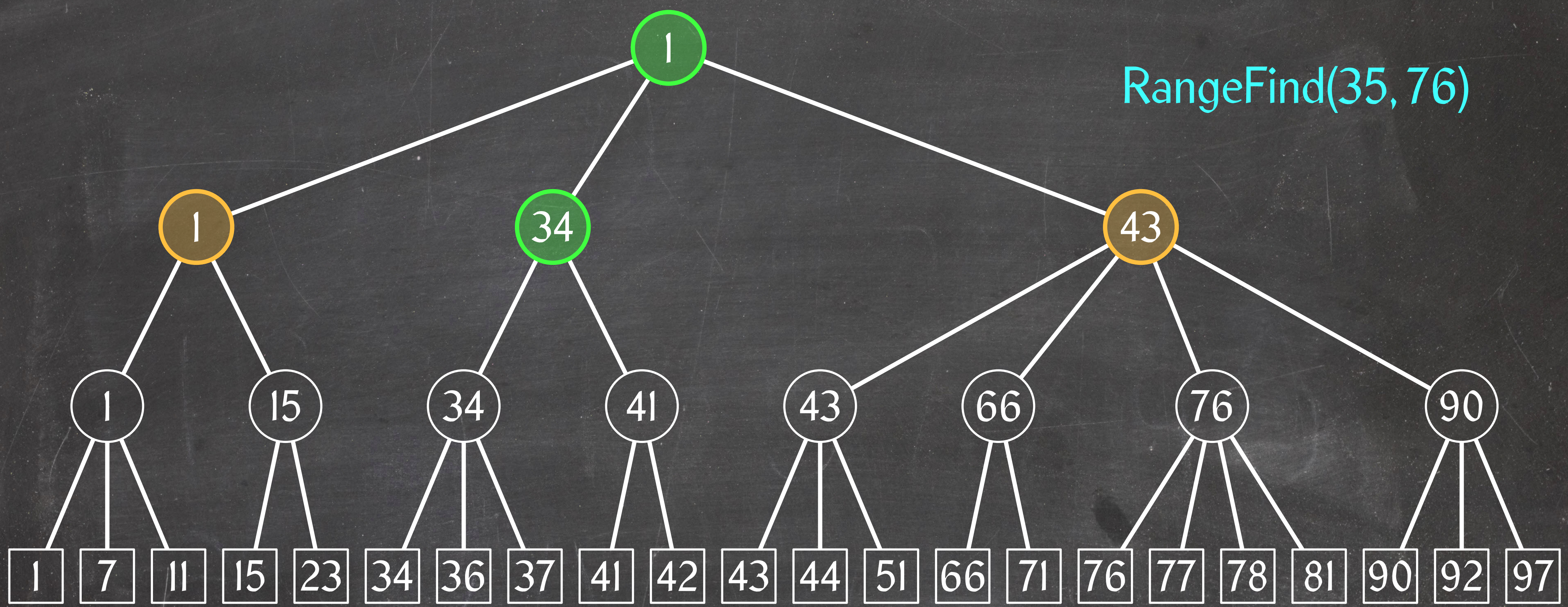


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

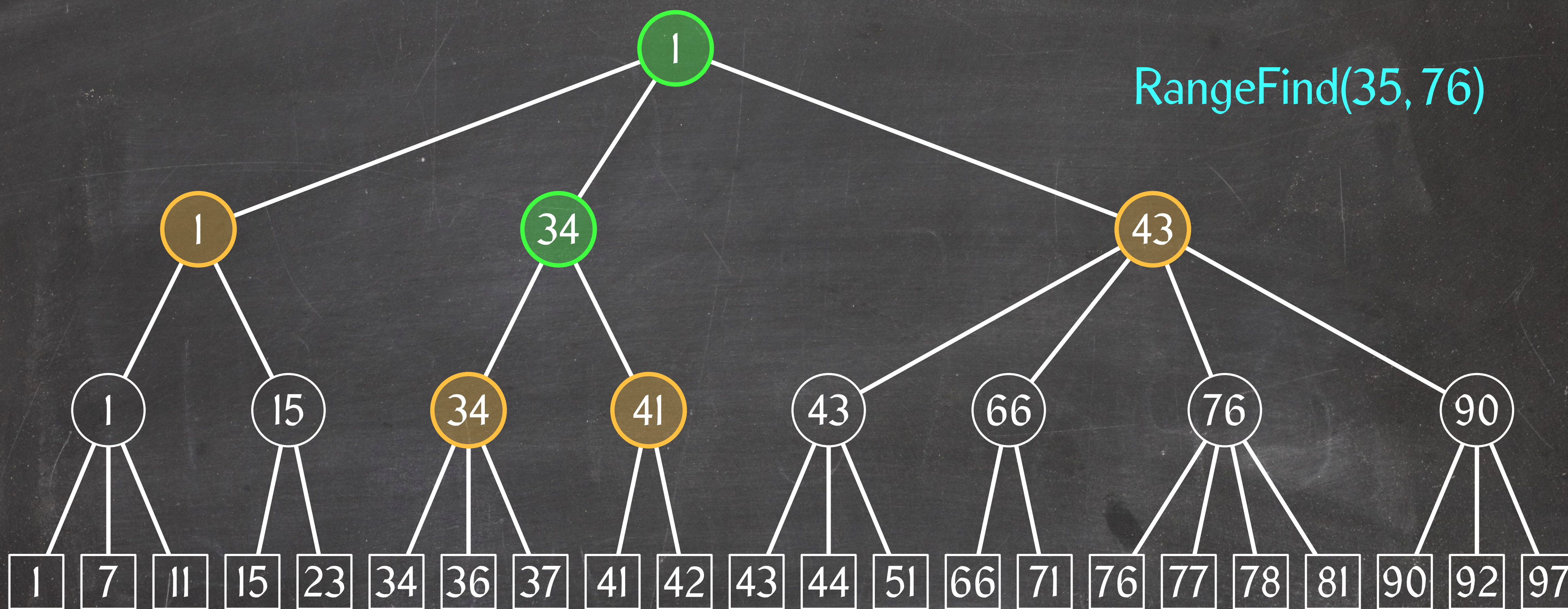


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

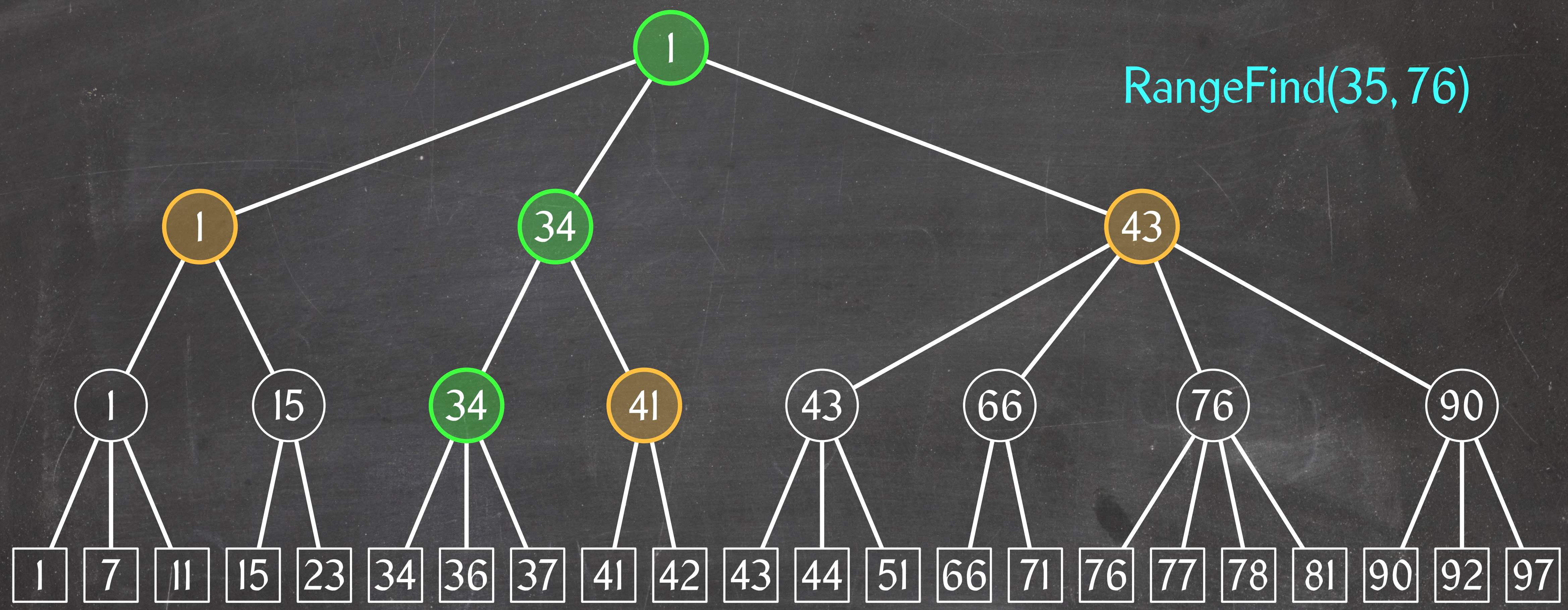


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

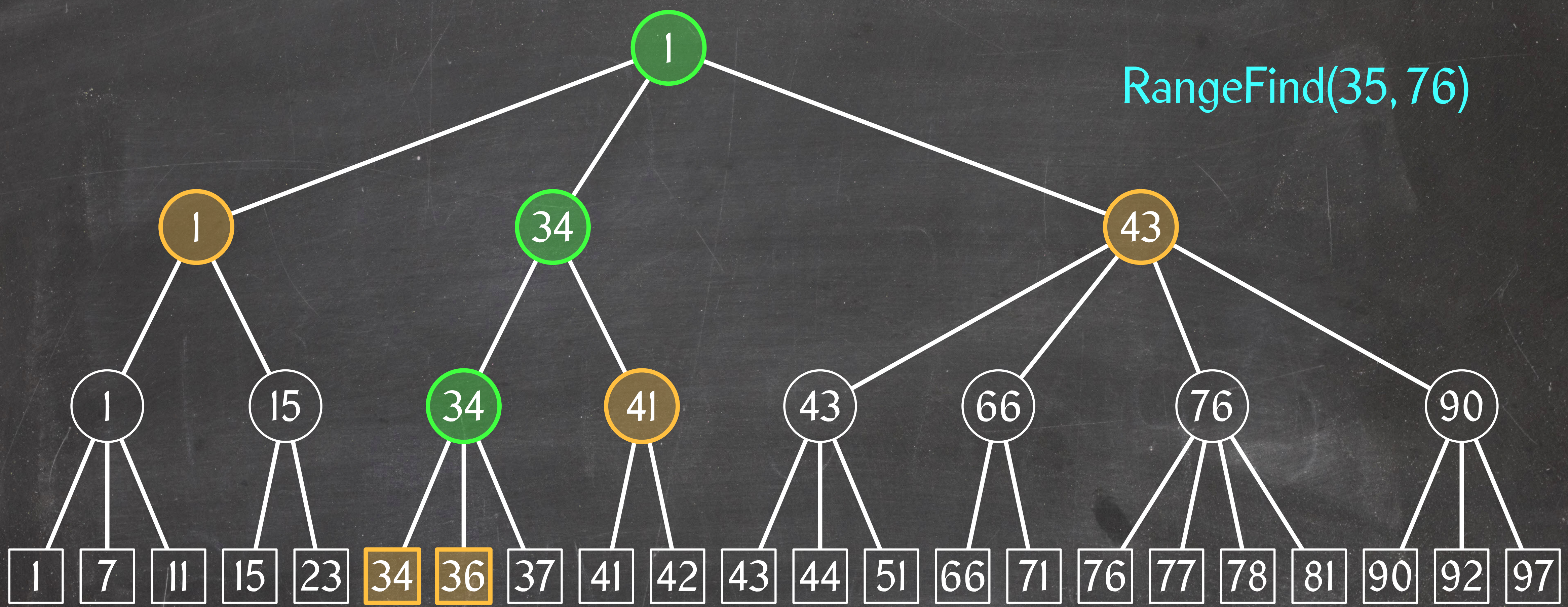


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

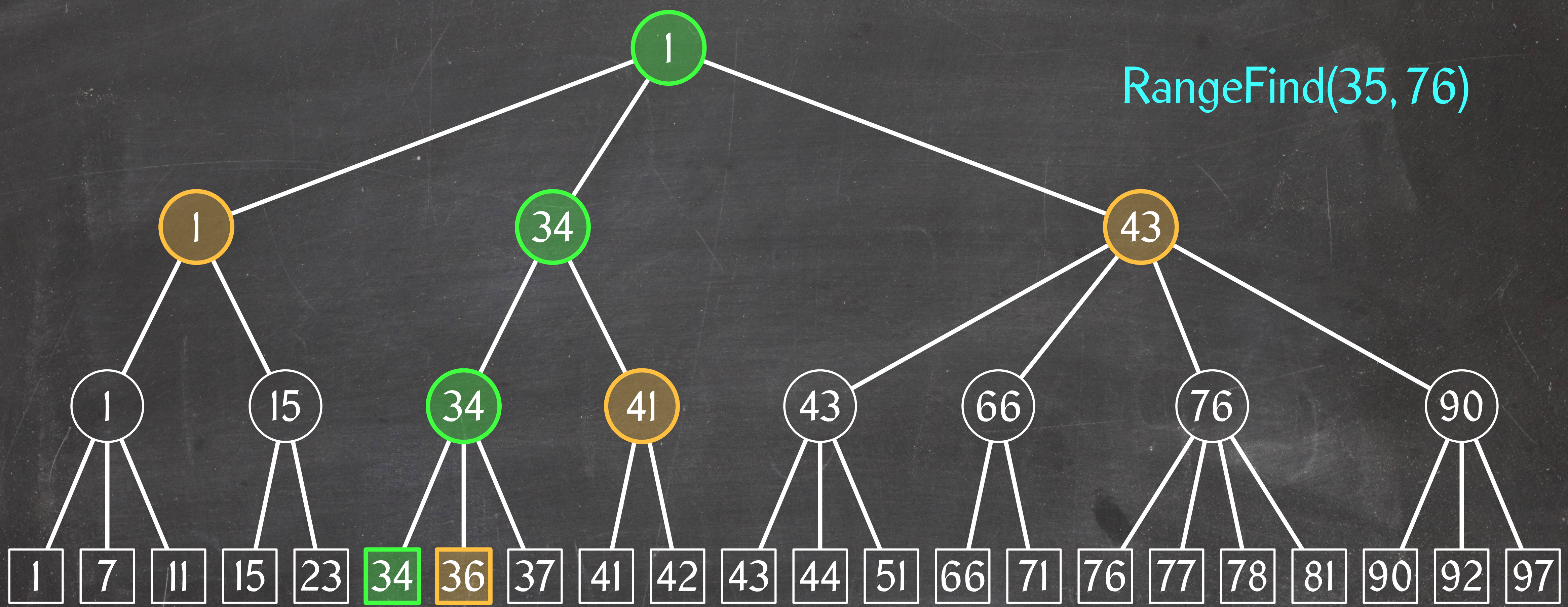


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

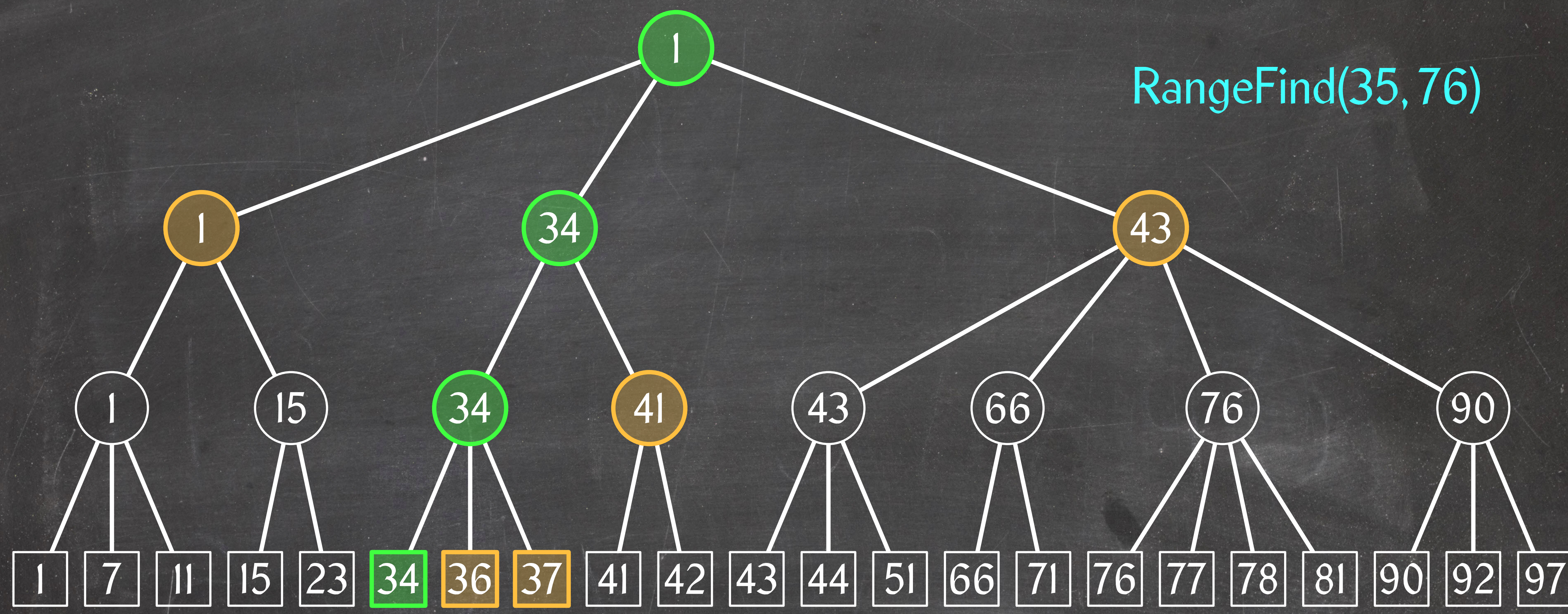


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \le x \le r$.

RangeFind Operation

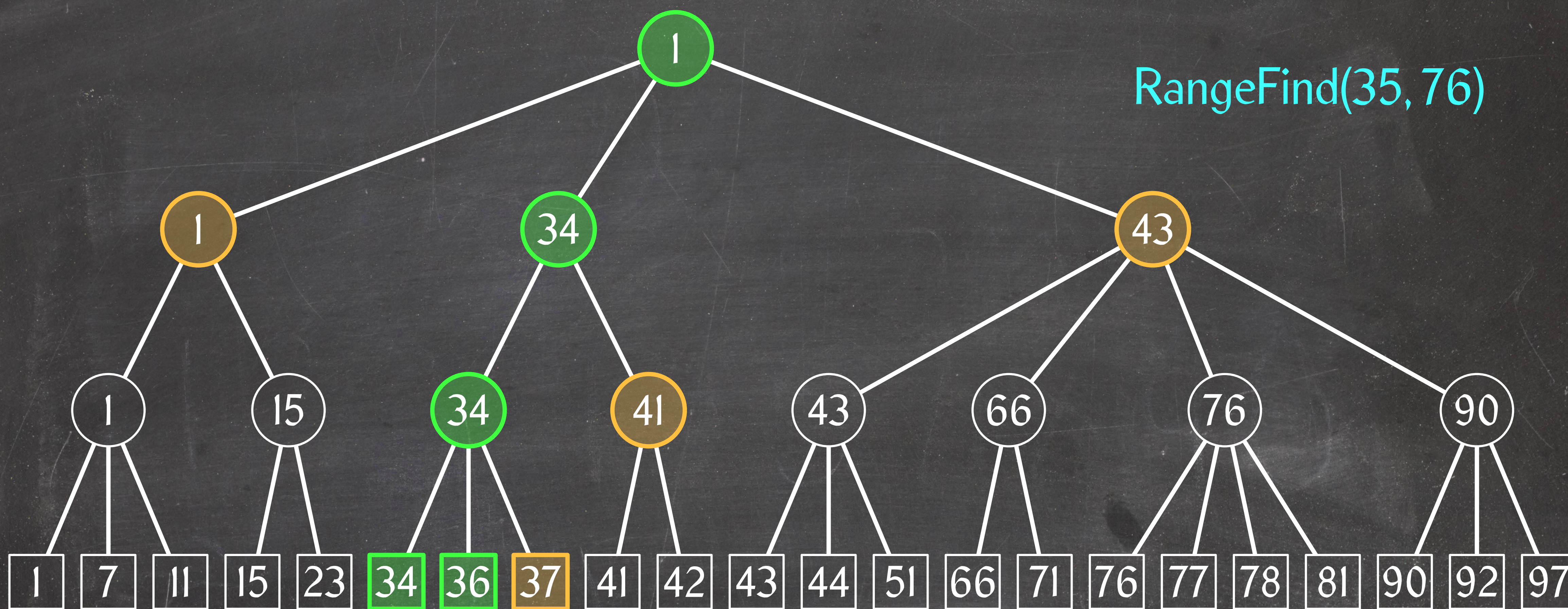


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

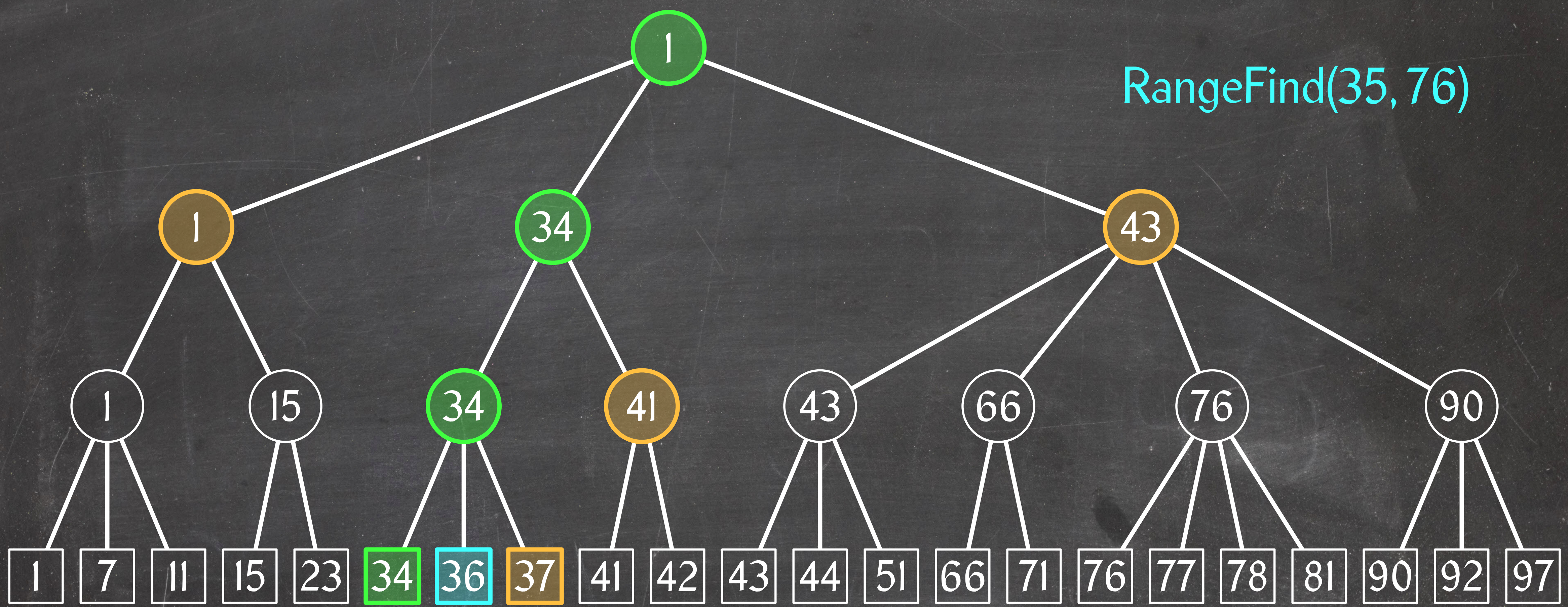


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \le x \le r$.

RangeFind Operation

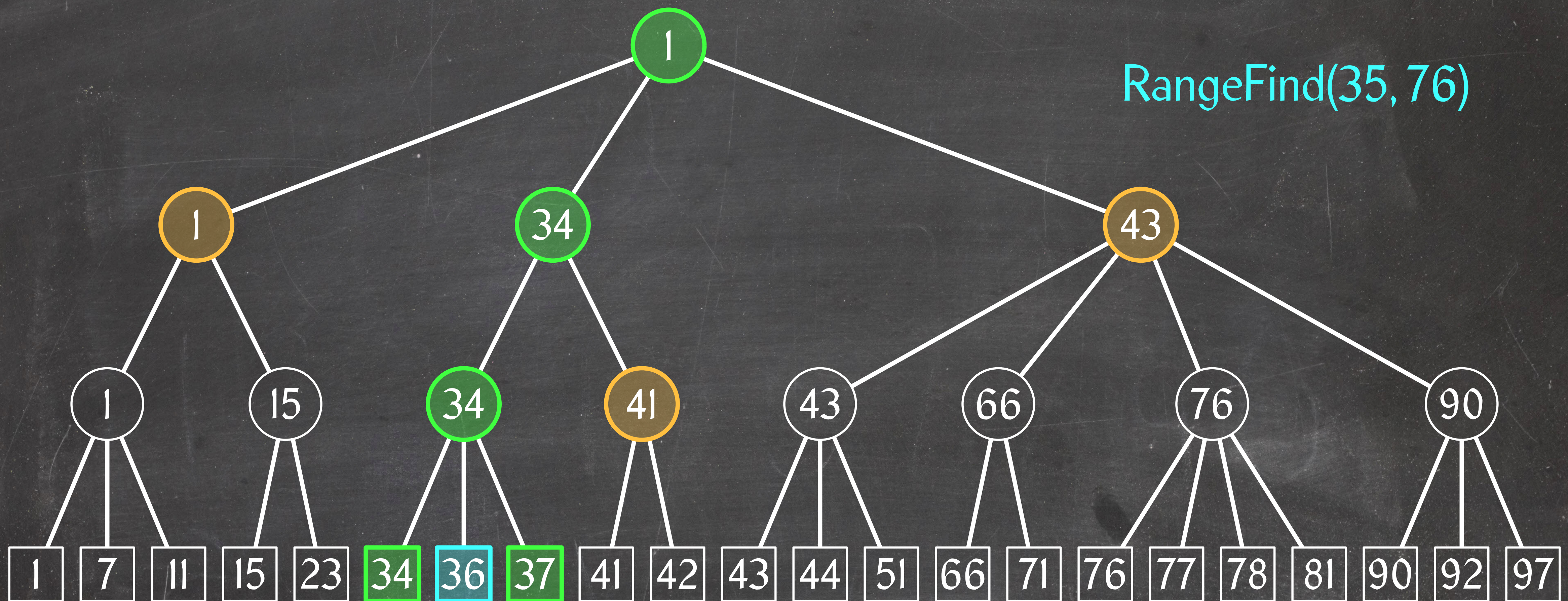


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

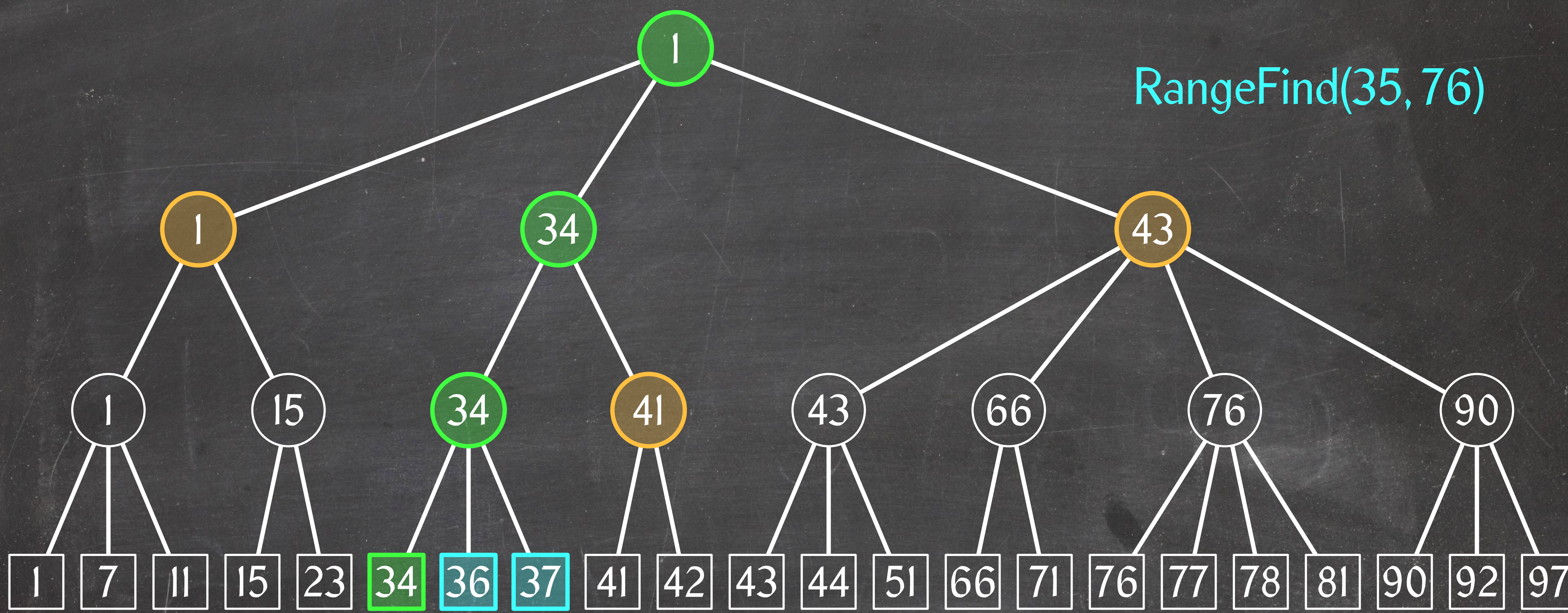


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

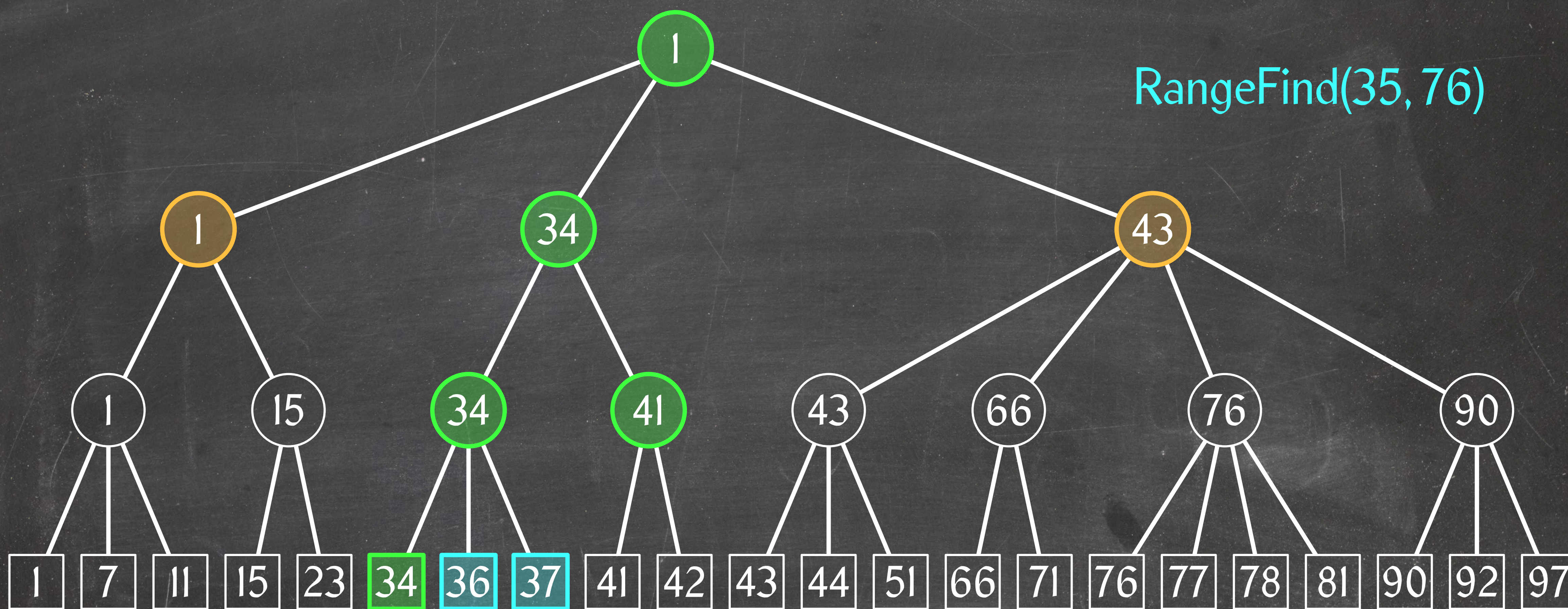


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

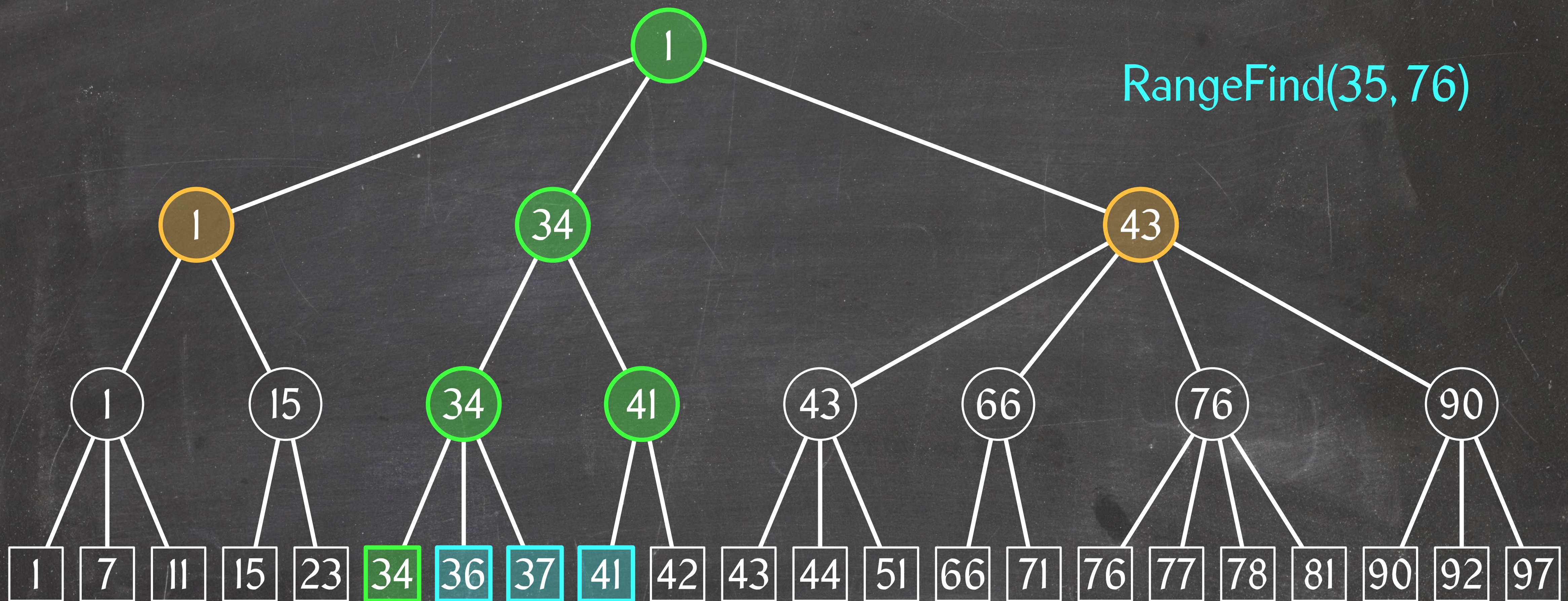


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

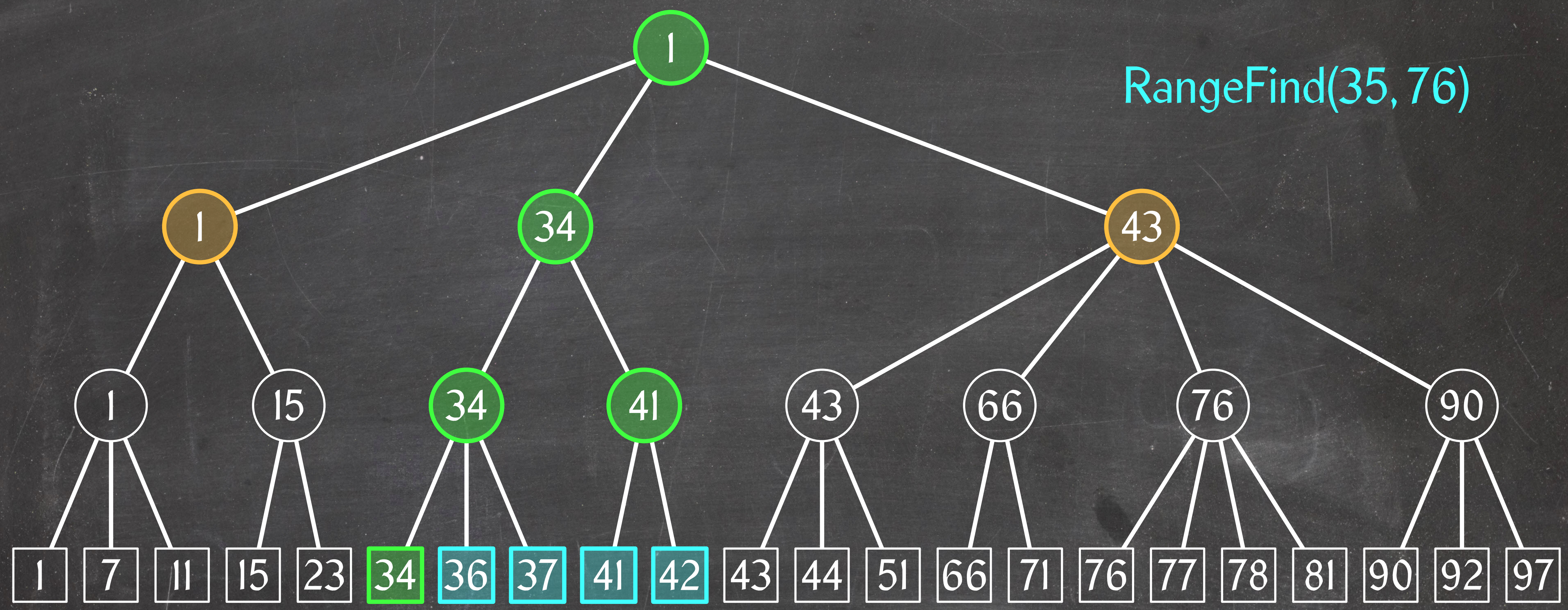


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

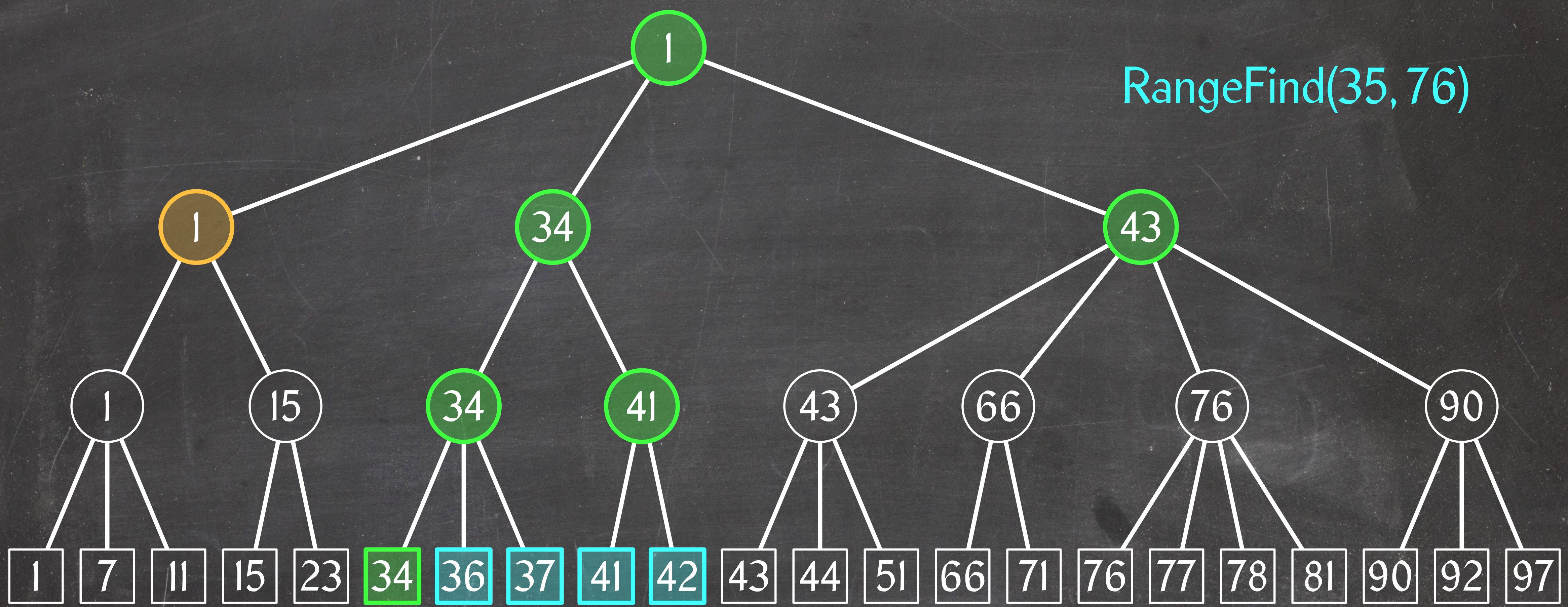


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \le x \le r$.

RangeFind Operation

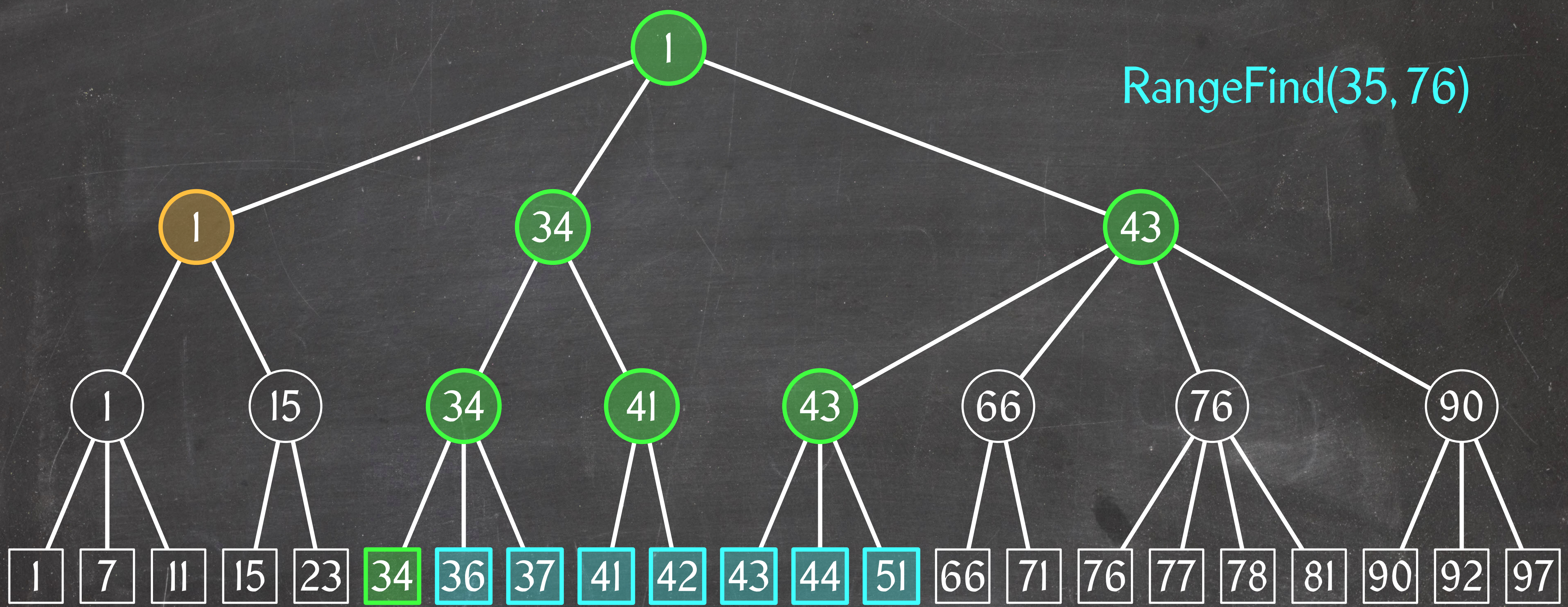


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

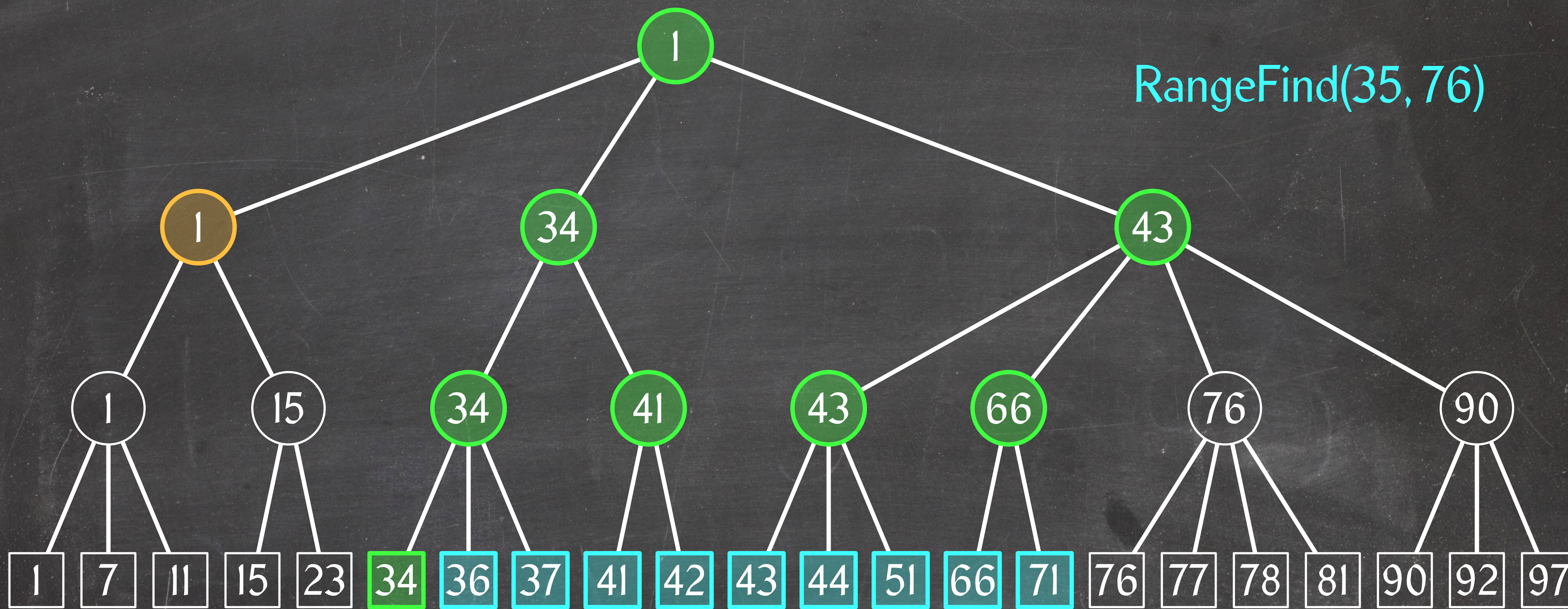


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

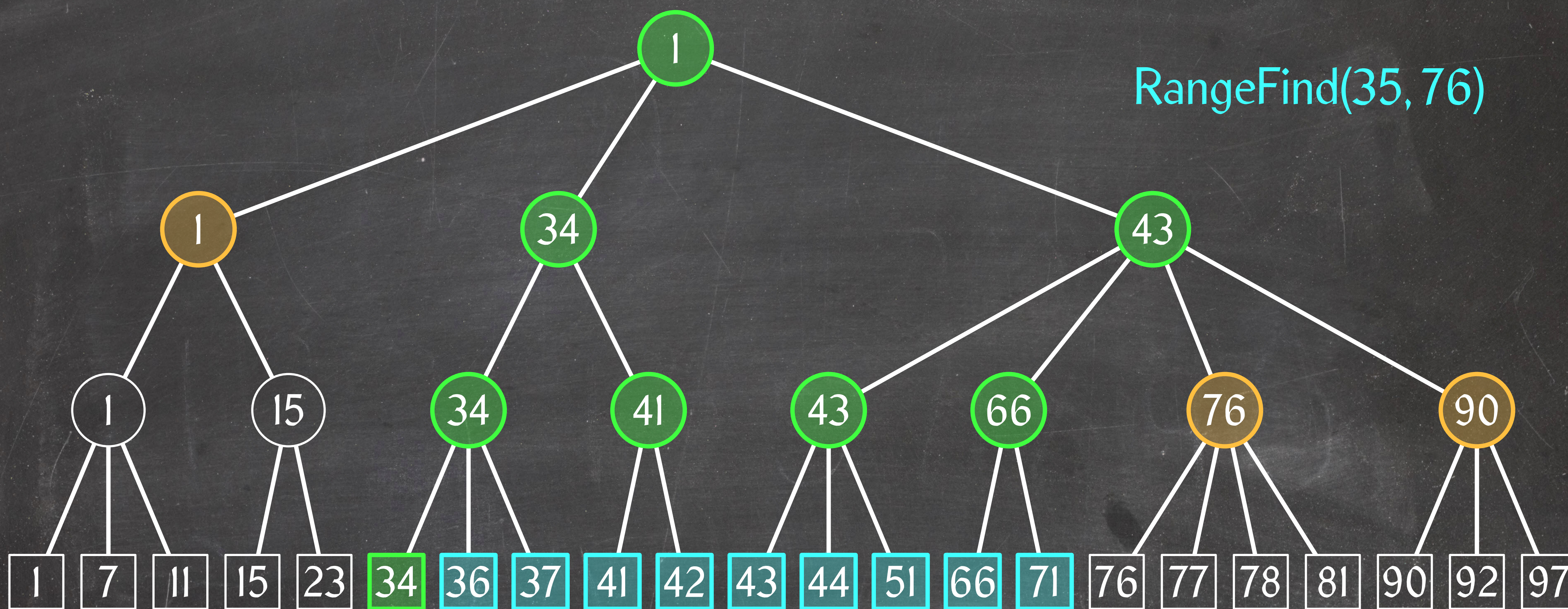


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

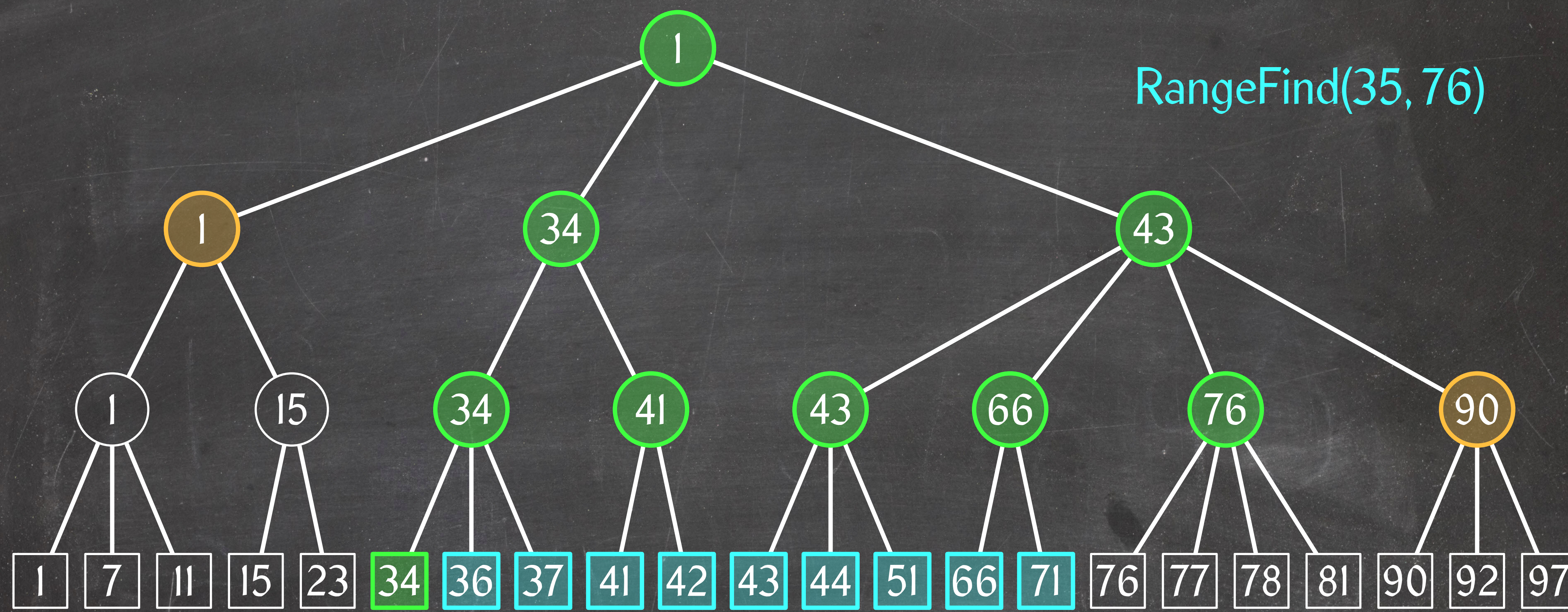


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation

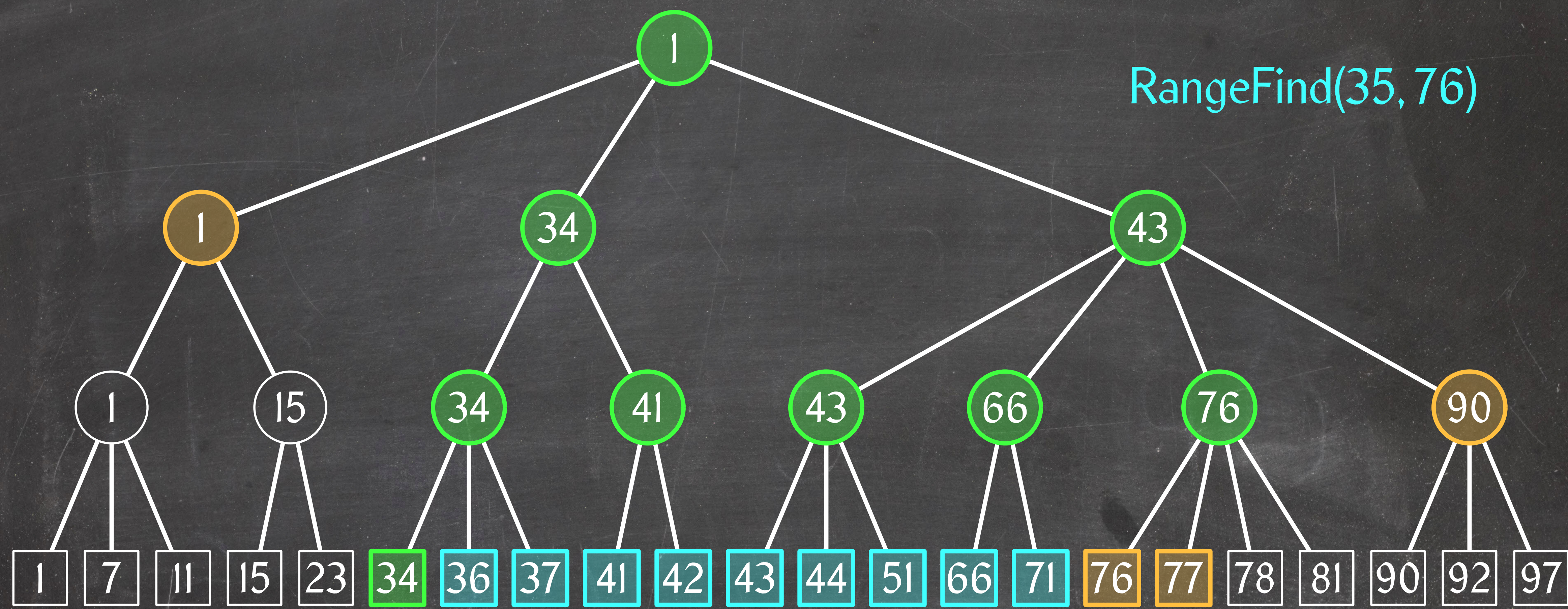


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \le x \le r$.

RangeFind Operation

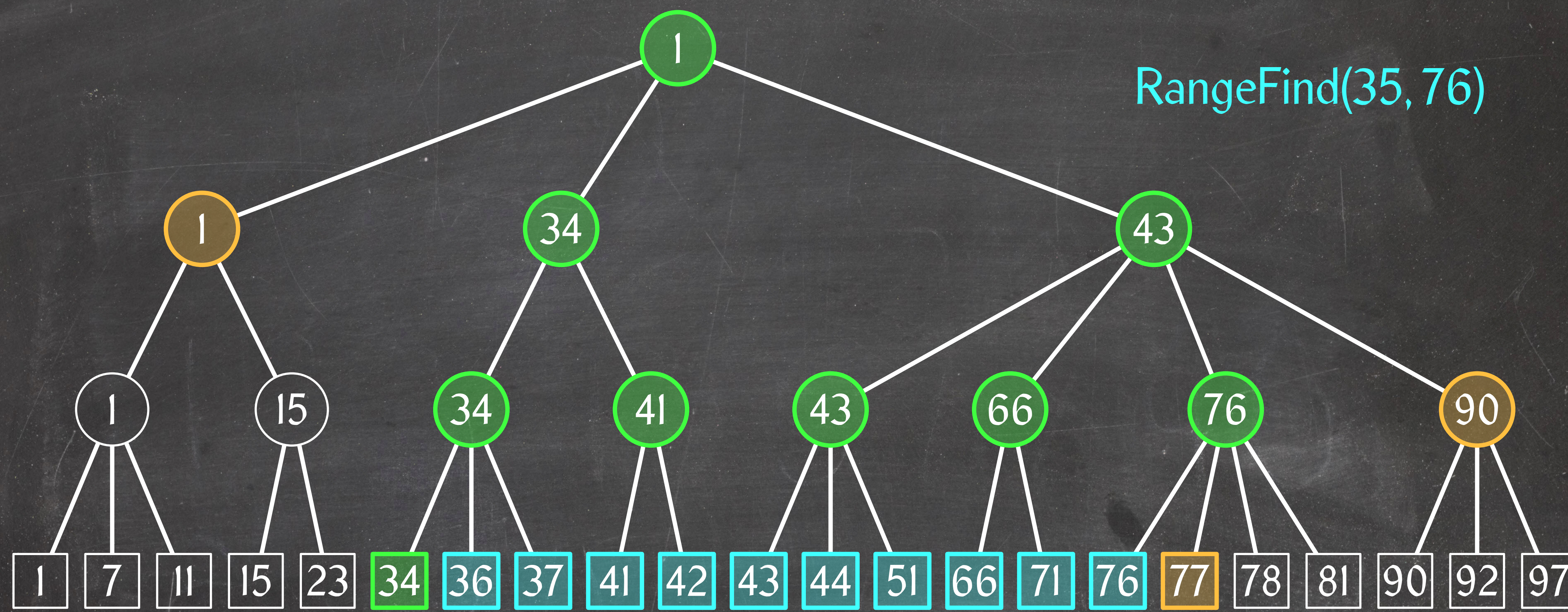


RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

RangeFind Operation



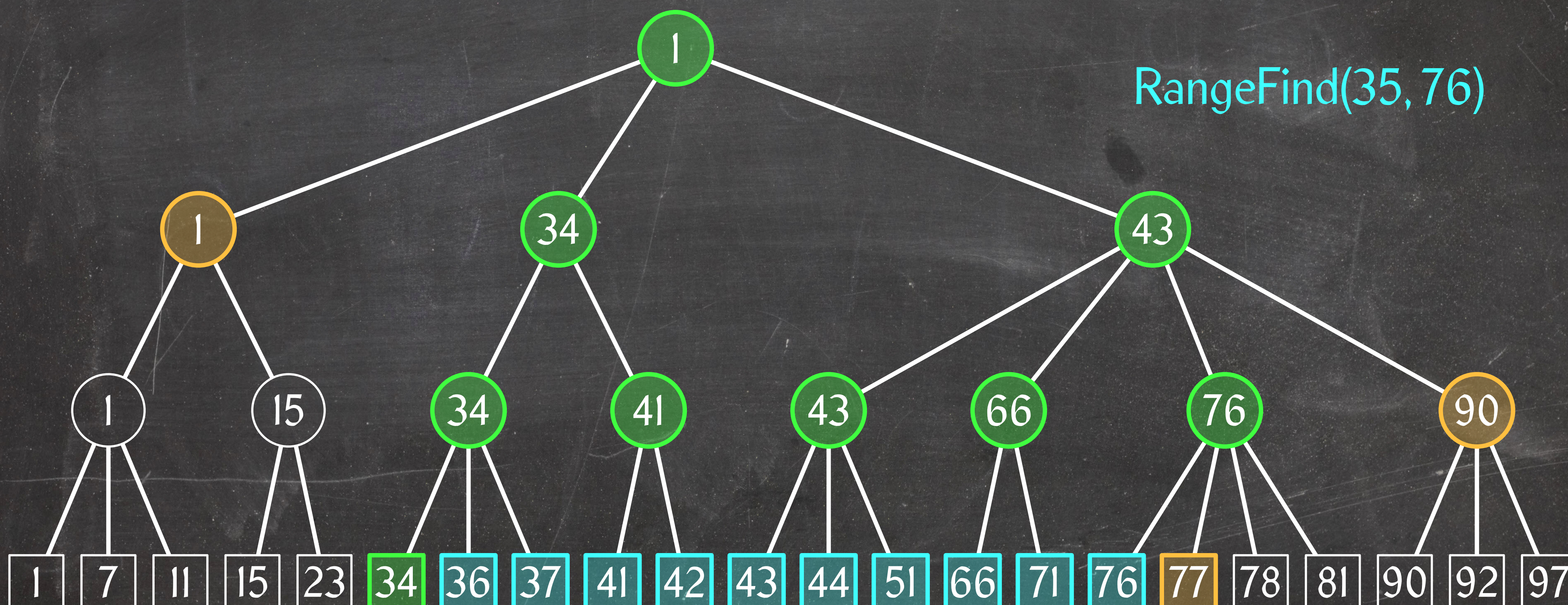
RangeFind(ℓ, r):

Perform a depth-first traversal of the tree:

- At every internal node, recursively visit every child
  - Whose key is no greater than r and
  - Whose right sibling does not exist or has a key no less than ℓ.
- At every leaf, report the element x it stores if $\ell \leq x \leq r$.

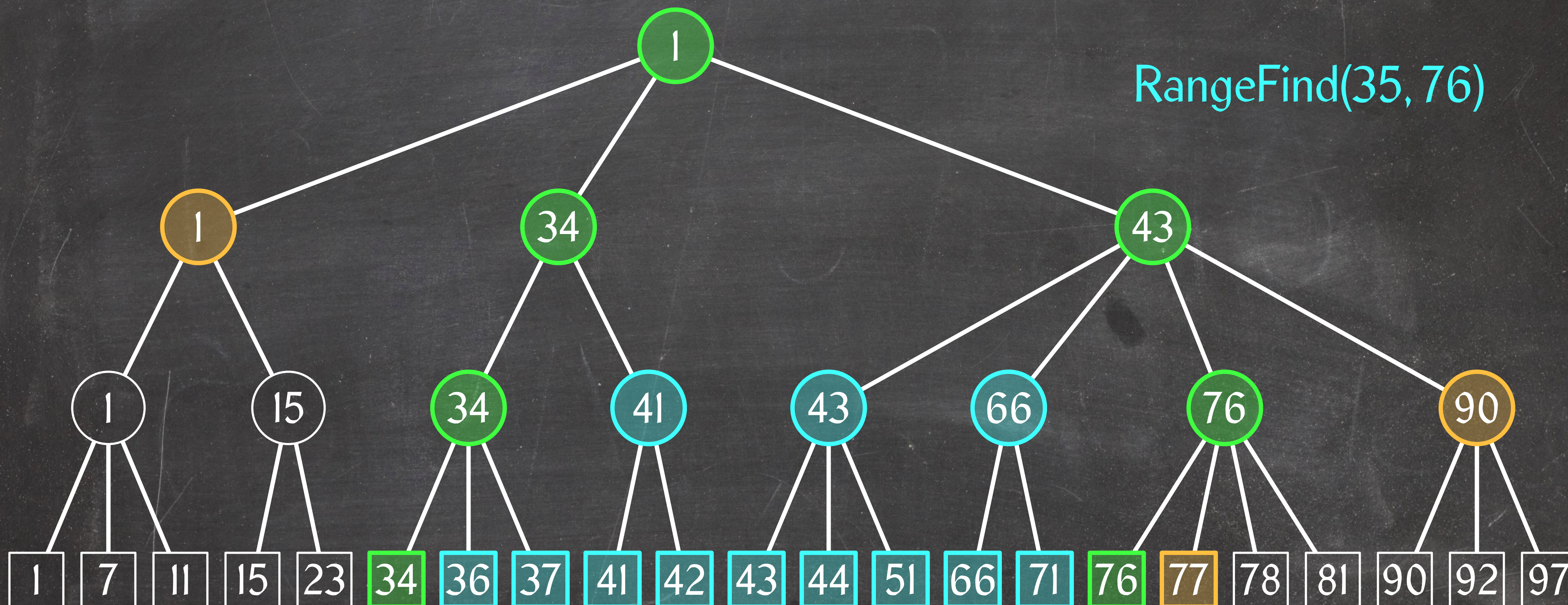
RangeFind Operation

Lemma: A RangeFind(ℓ, r) operation reports all elements between ℓ and r and only those.



RangeFind Operation

Lemma: A RangeFind(ℓ, r) operation takes $O(\lg n + k)$ time, where k is the number of elements reported.



- Every inspected node has a parent we visit $\Rightarrow$ we inspect at most b times as many nodes as we visit.
- We visit $O(\lg n)$ green nodes.
- The cyan nodes form (a, b)-trees with in total at most k leaves.

Putting Data Structures to Good Use

We have already seen examples where data structures help algorithms to maintain important state information efficiently:

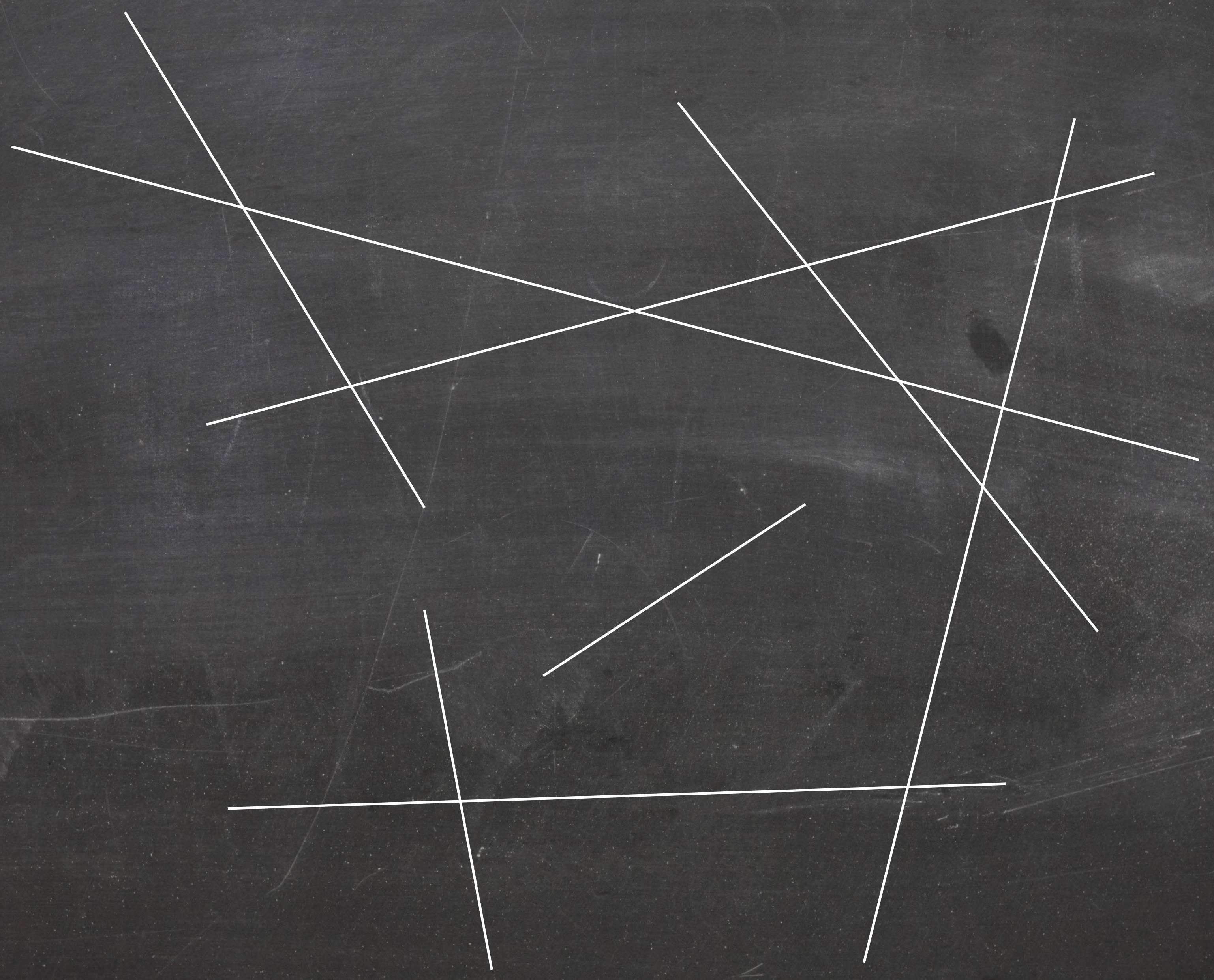
Graph exploration maintains the unexplored vertices adjacent to explored ones in a queue, stack or priority queue. The choice of structure influences the structure of the computed tree or forest.

Kruskal's algorithm uses a union-find data structure to maintain the set of trees in the current forest.

Huffman's algorithm uses a priority queue to decide which subtrees to merge in each step of building the tree.

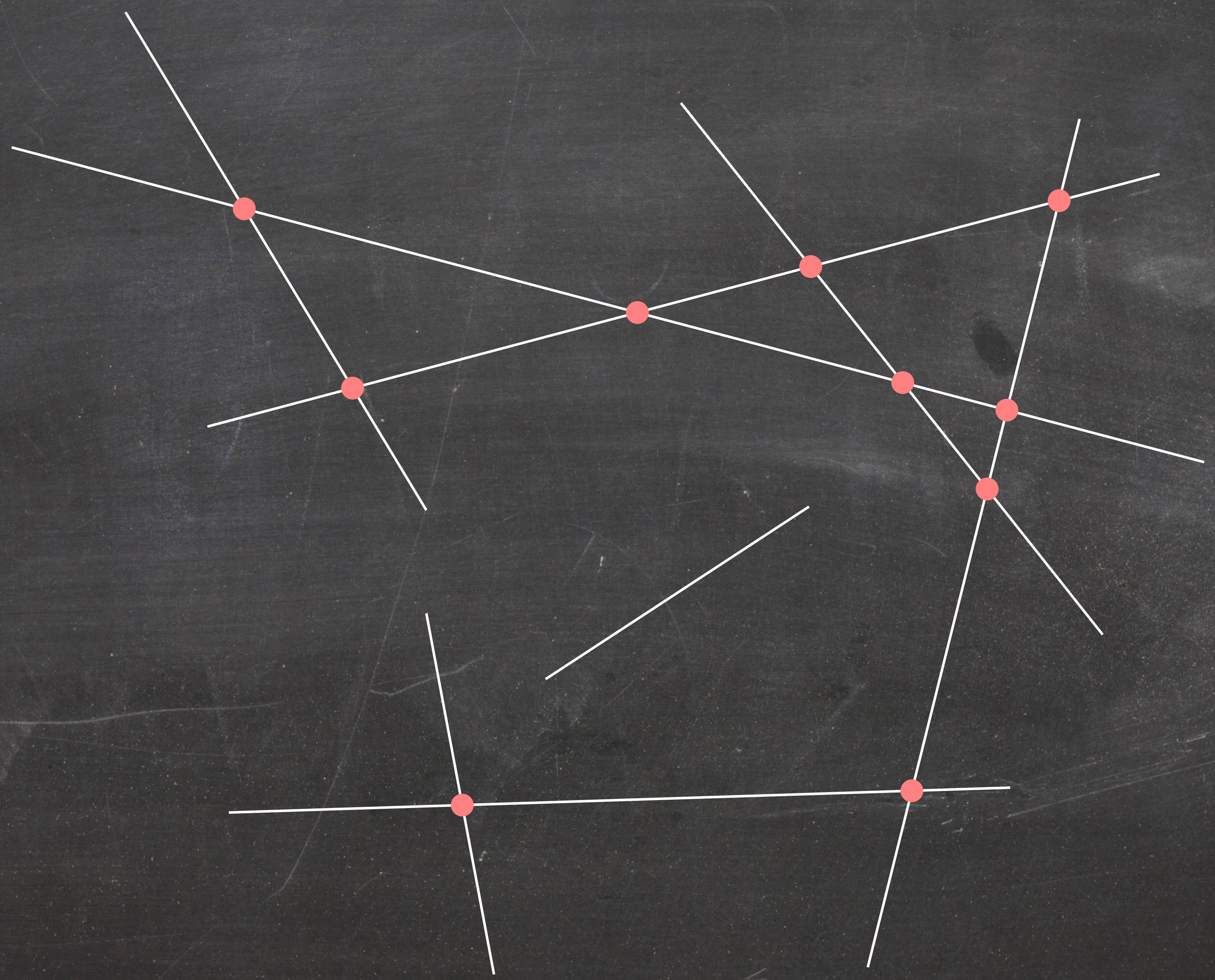
Line Segment Intersection

Given a set of line segments in the plane, report all their intersection points.



Line Segment Intersection

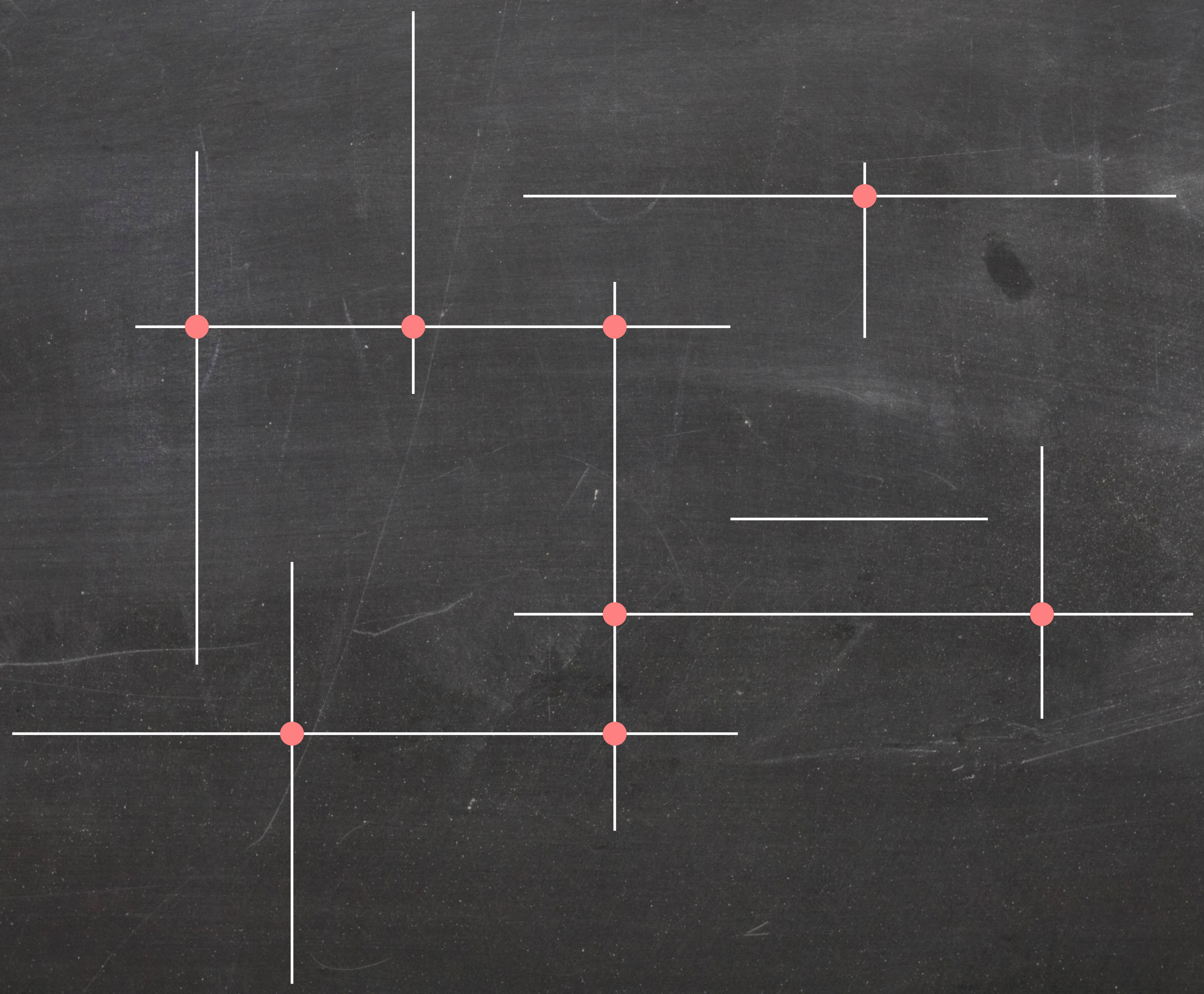
Given a set of line segments in the plane, report all their intersection points.



Orthogonal Line Segment Intersection

Special case: Find all intersections between

- n vertical segments $v_1, v_2, \ldots, v_n$ and
- n horizontal segments $h_1, h_2, \ldots, h_n$.

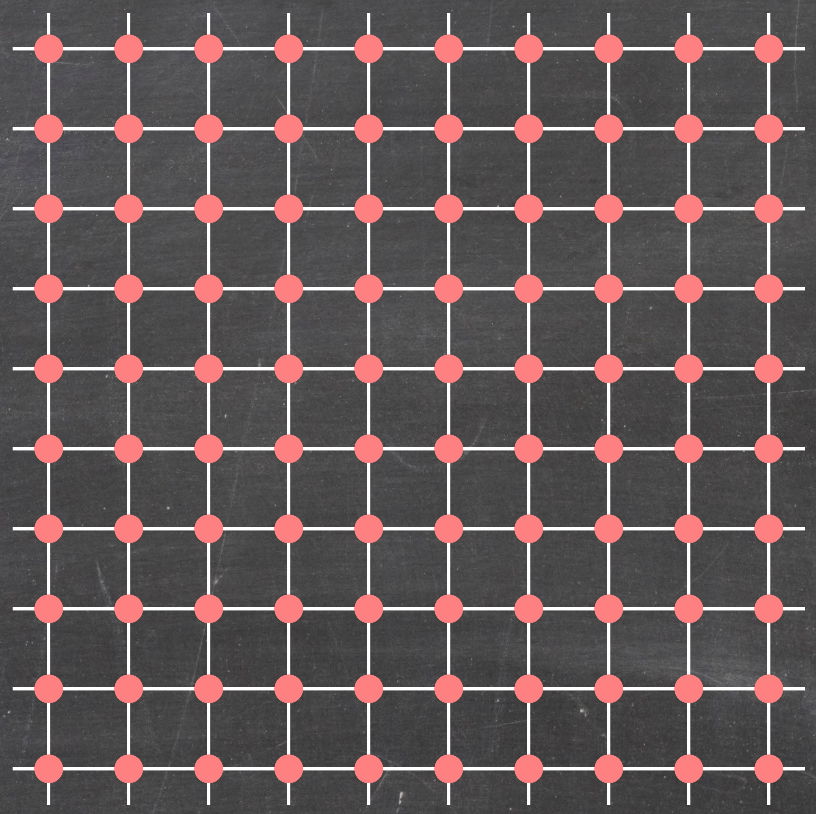


Output Sensitivity

How many intersections are there in the worst case?

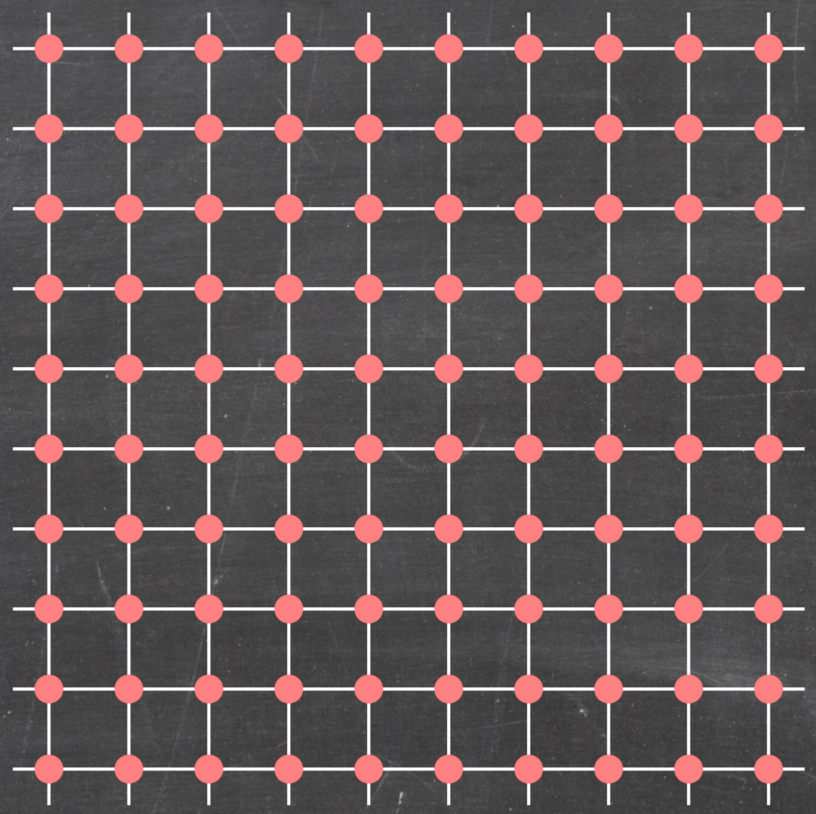
Output Sensitivity

How many intersections are there in the worst case?



Output Sensitivity

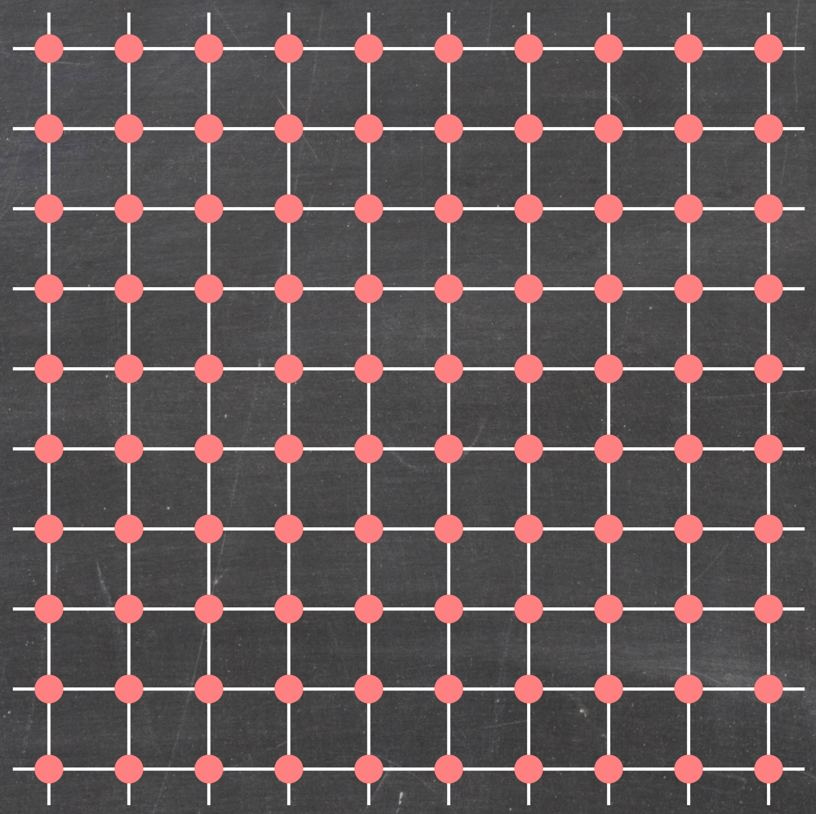
How many intersections are there in the worst case?



$\Rightarrow$ The trivial algorithm of testing every pair of segments is optimal in the worst case.

Output Sensitivity

How many intersections are there in the worst case?

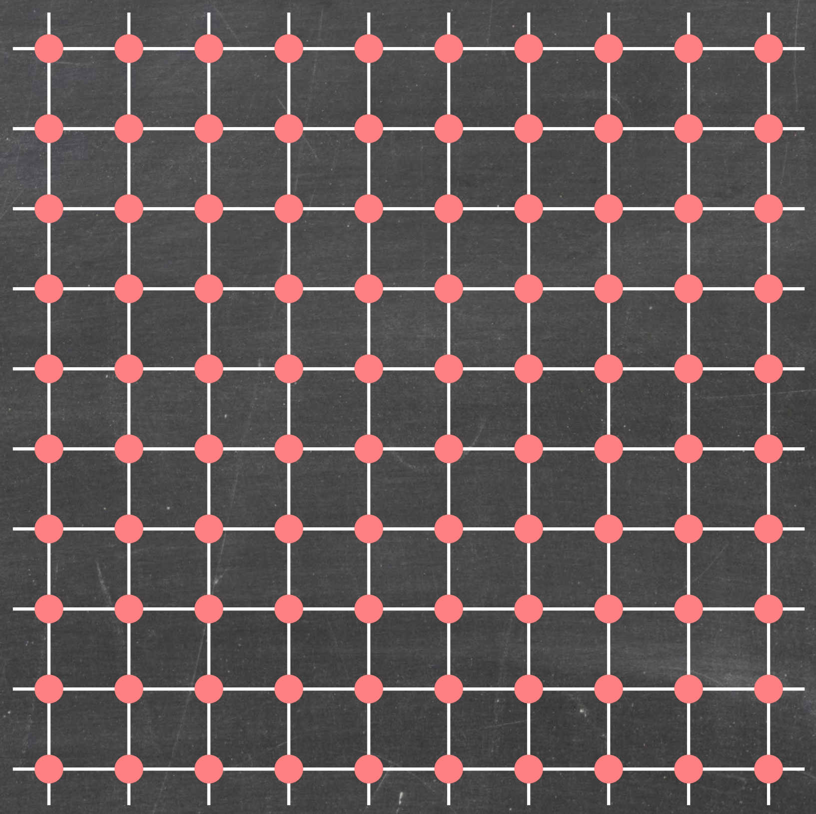


$\Rightarrow$ The trivial algorithm of testing every pair of segments is optimal in the worst case.

Can we still do better?

Output Sensitivity

How many intersections are there in the worst case?



$\Rightarrow$ The trivial algorithm of testing every pair of segments is optimal in the worst case.

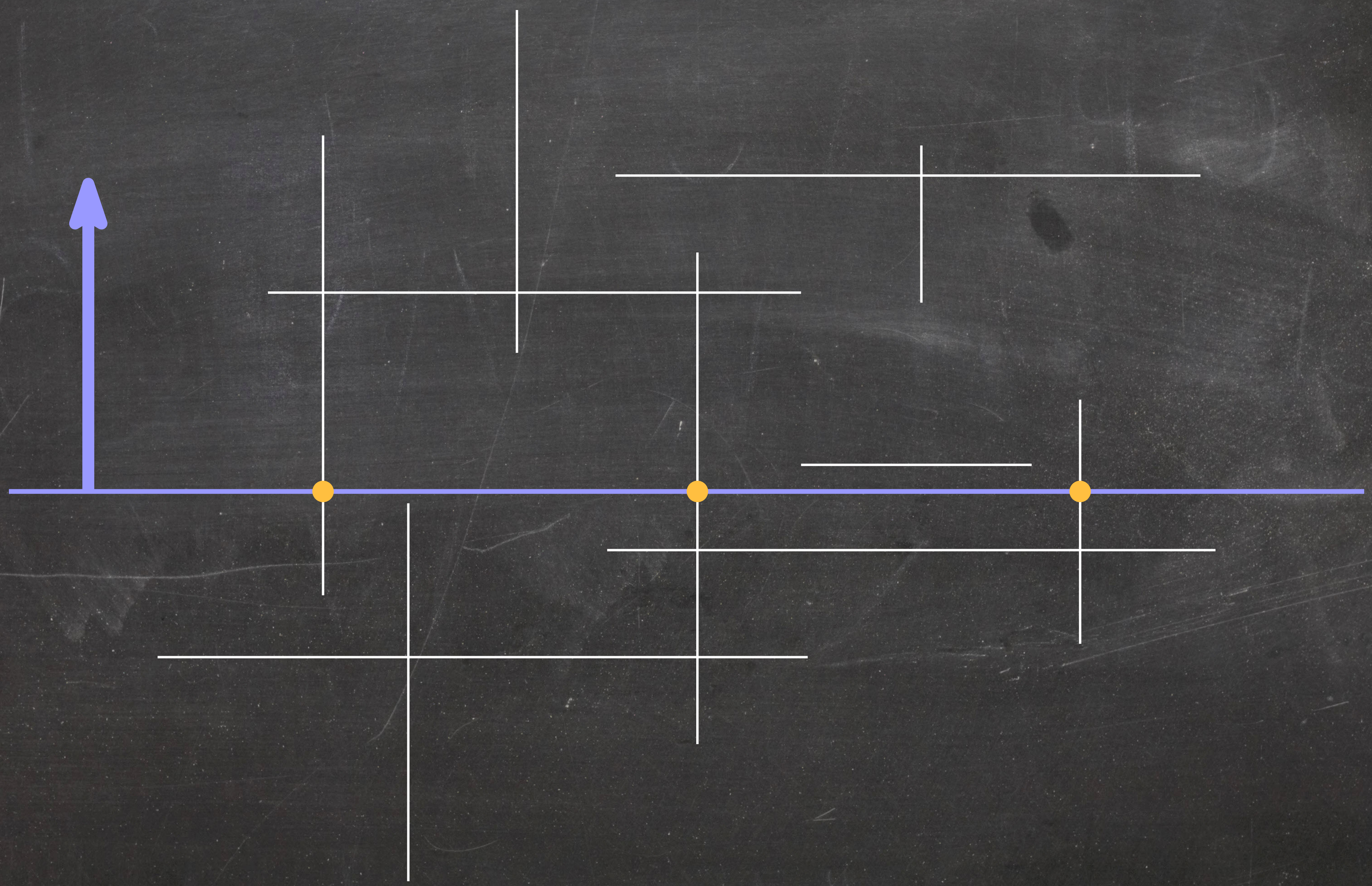
Can we still do better?

- Yes: We try to spend as little time as possible unless the output is big.
- This is called output sensitivity.

Plane Sweeping

Idea:

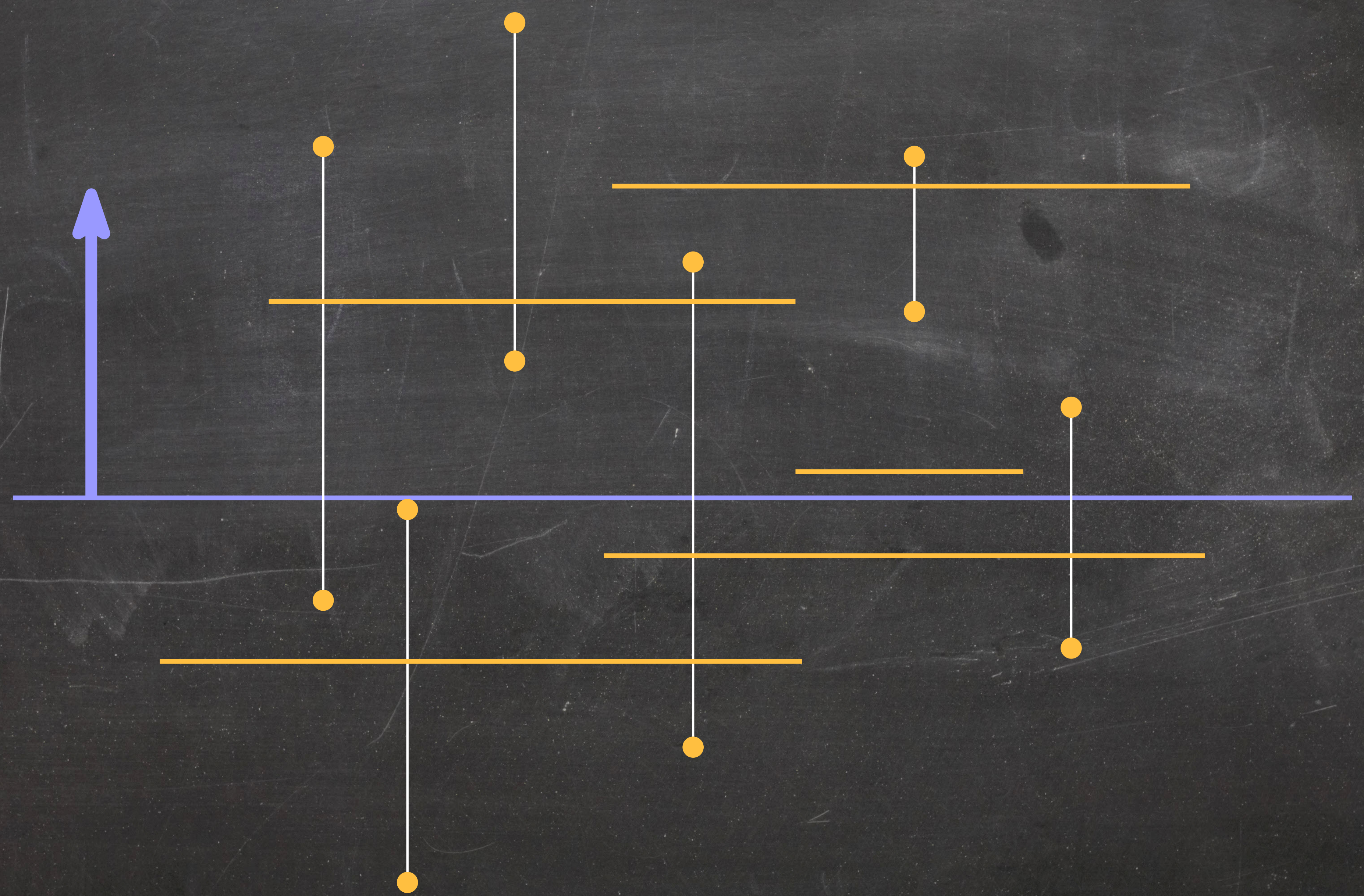
- Sweep a horizontal sweep line upward across the plane.
- Maintain a sweep line structure representing interactions between sweep line and geometric objects.



Event Points

Discretization of plane sweep technique:

- Update sweep line structure only at certain event points.
- Solve problem by asking queries on sweep line structure at other event points.



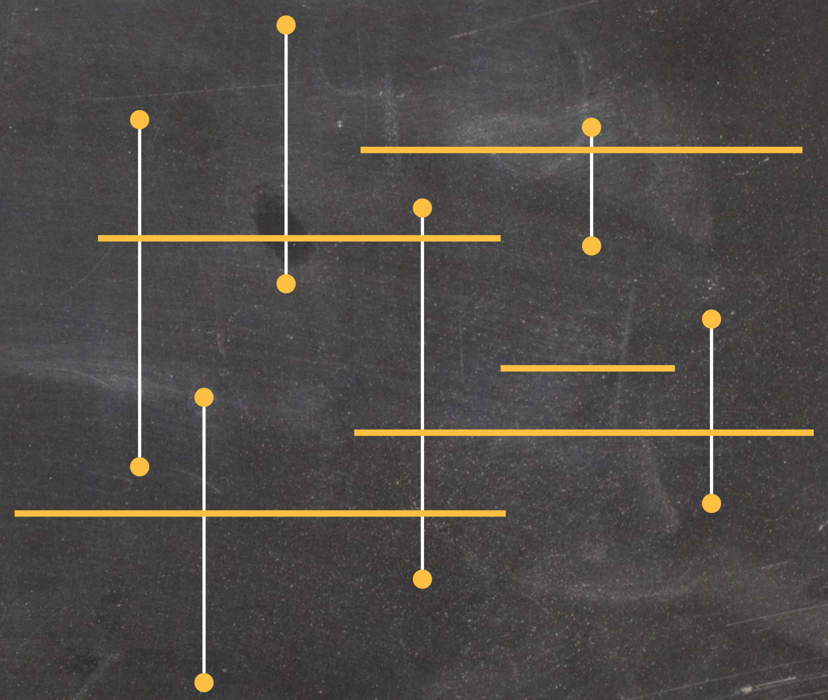
Orthogonal Line Segment Intersection: Final Algorithm

Sweep line structure: (a, b)-tree T storing all vertical segments intersecting the sweep line, sorted from left to right.

Orthogonal Line Segment Intersection: Final Algorithm

Sweep line structure: (a, b)-tree T storing all vertical segments intersecting the sweep line, sorted from left to right.

Event points:



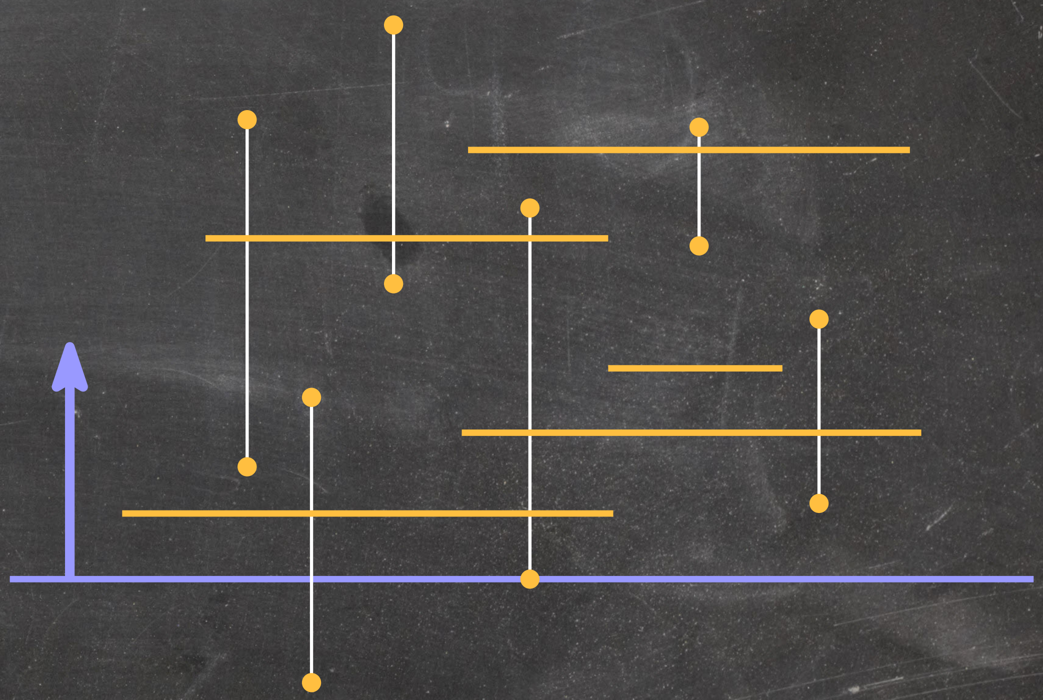
Orthogonal Line Segment Intersection: Final Algorithm

Sweep line structure: (a, b)-tree T storing all vertical segments intersecting the sweep line, sorted from left to right.

Event points:

- Bottom endpoint of vertical segment v_i:
  - Sweep line starts to intersect v_i.

  $\Rightarrow$ Insert v_i into T.



Orthogonal Line Segment Intersection: Final Algorithm

Sweep line structure: (a, b)-tree T storing all vertical segments intersecting the sweep line, sorted from left to right.

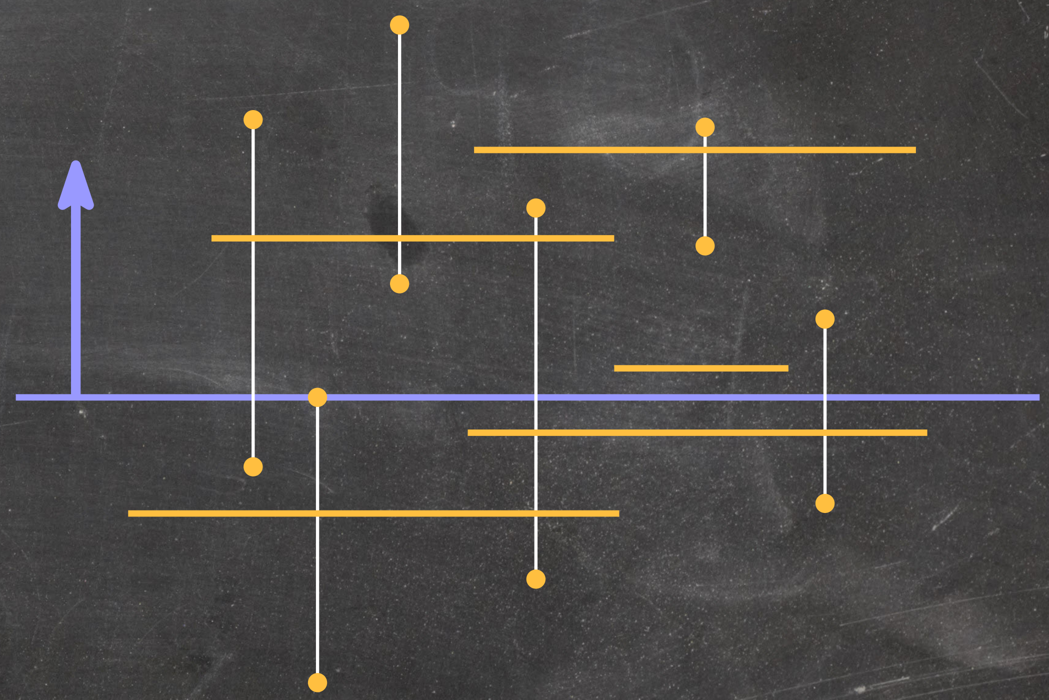
Event points:

- Bottom endpoint of vertical segment v_i:
  - Sweep line starts to intersect v_i.

  $\Rightarrow$ Insert v_i into T.

- Top endpoint of vertical segment v_i:
  - Sweep line stops intersecting v_i.

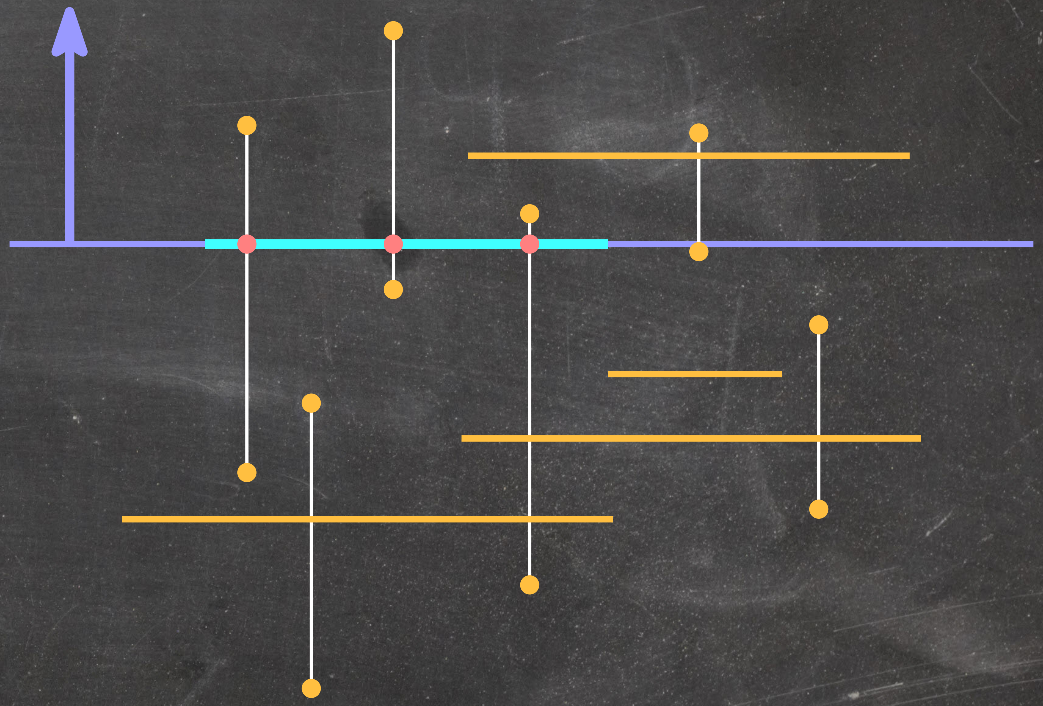
  $\Rightarrow$ Delete v_i from T.



Orthogonal Line Segment Intersection: Final Algorithm

Sweep line structure: (a, b)-tree T storing all vertical segments intersecting the sweep line, sorted from left to right.

Event points:



- Bottom endpoint of vertical segment v_i:
  - Sweep line starts to intersect v_i.

  $\Rightarrow$ Insert v_i into T.
- Top endpoint of vertical segment v_i:
  - Sweep line stops intersecting v_i.

  $\Rightarrow$ Delete v_i from T.
- Horizontal segment h_j:
  - T contains exactly the segments spanning the y-coordinate of h_j.

  $\Rightarrow$ Find all segments intersecting h_j using a RangeFind operation.

Orthogonal Line Segment Intersection: Analysis

Event points:

- n bottom endpoints of vertical segments $\Rightarrow$ n insertions into T
- n top endpoints of vertical segments $\Rightarrow$ n deletions from T
- n horizontal segments $\Rightarrow$ n RangeFind queries on T

Orthogonal Line Segment Intersection: Analysis

Event points:

- n bottom endpoints of vertical segments $\Rightarrow$ n insertions into T
- n top endpoints of vertical segments $\Rightarrow$ n deletions from T
- n horizontal segments $\Rightarrow$ n RangeFind queries on T

- Cost per insertion or deletion = $O(\lg n)$
- Cost per RangeFind operation = $O(\lg n + k_j)$, where k_j = number of segments intersecting h_j

Orthogonal Line Segment Intersection: Analysis

Event points:

- n bottom endpoints of vertical segments $\Rightarrow$ n insertions into T
- n top endpoints of vertical segments $\Rightarrow$ n deletions from T
- n horizontal segments $\Rightarrow$ n RangeFind queries on T

- Cost per insertion or deletion = $O(\lg n)$
- Cost per RangeFind operation = $O(\lg n + k_j)$, where k_j = number of segments intersecting h_j

Total cost:

$$O(n \lg n) + \sum_{j=1}^{n} O(k_j) = O(n \lg n + k)$$

Orthogonal Line Segment Intersection: Analysis

Event points:

- n bottom endpoints of vertical segments $\Rightarrow$ n insertions into T
- n top endpoints of vertical segments $\Rightarrow$ n deletions from T
- n horizontal segments $\Rightarrow$ n RangeFind queries on T

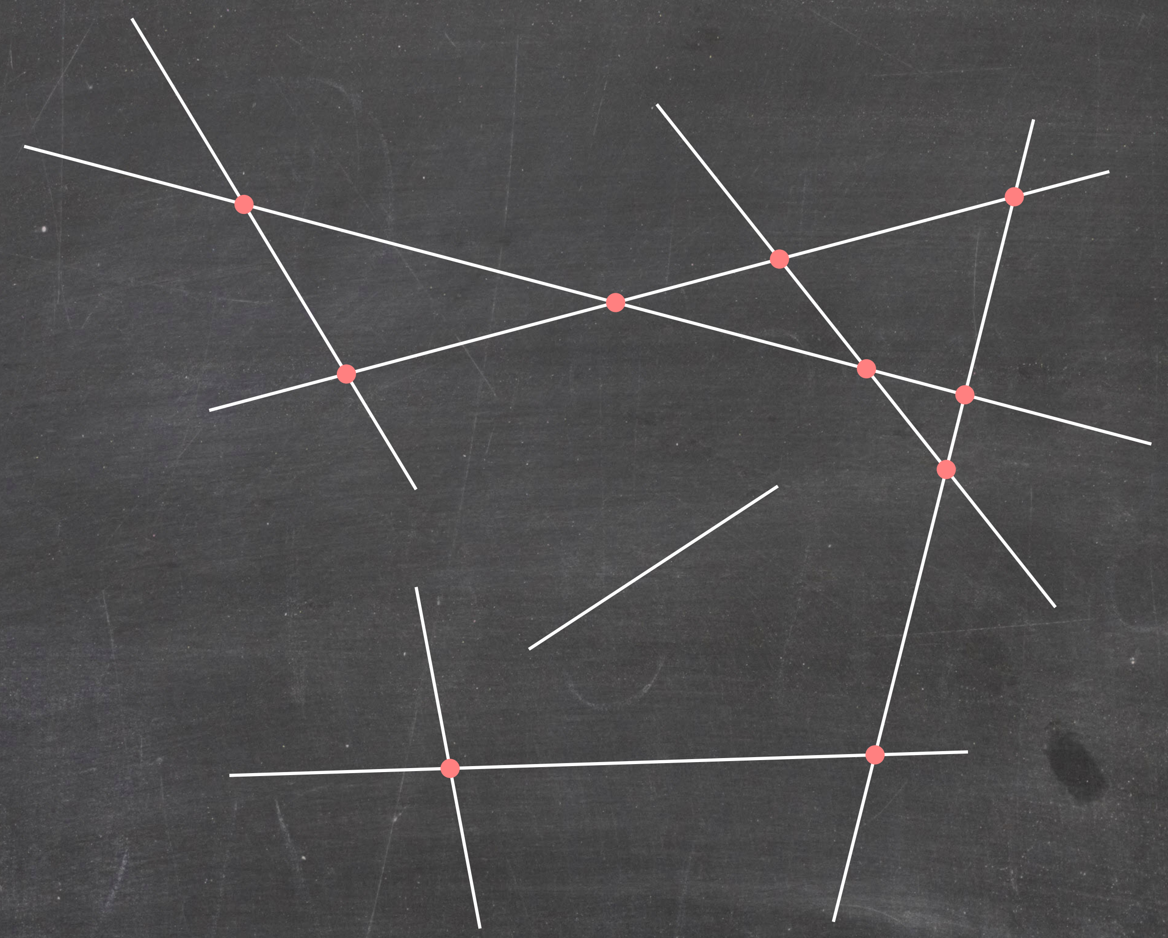
- Cost per insertion or deletion = $O(\lg n)$
- Cost per RangeFind operation = $O(\lg n + k_j)$, where k_j = number of segments intersecting h_j

Total cost:

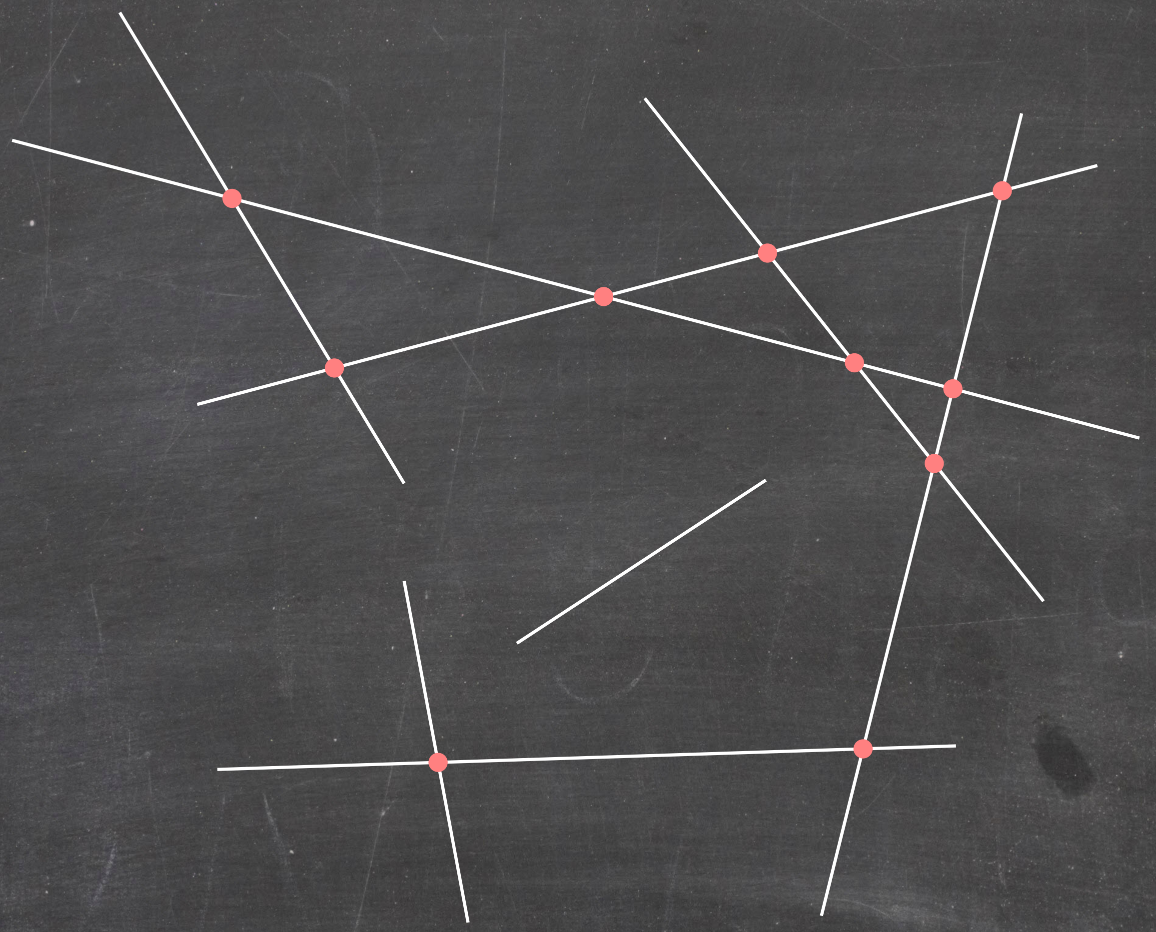
$$O(n \lg n) + \sum_{j=1}^{n} O(k_j) = O(n \lg n + k)$$

Theorem: The orthogonal line segment intersection problem can be solved in $O(n \lg n + k)$ time.

General Line Segment Intersection



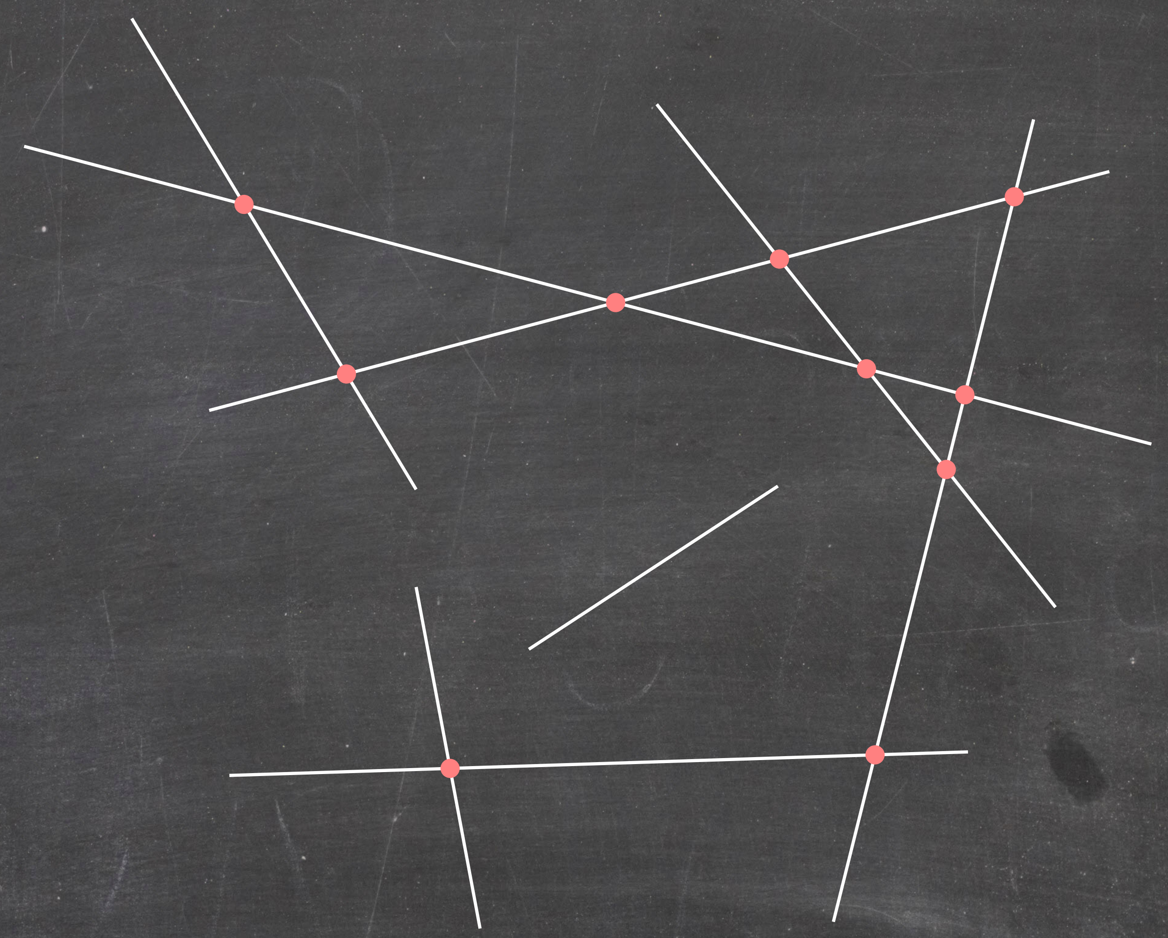
General Line Segment Intersection



Questions:

- Whats' the sweep line status?

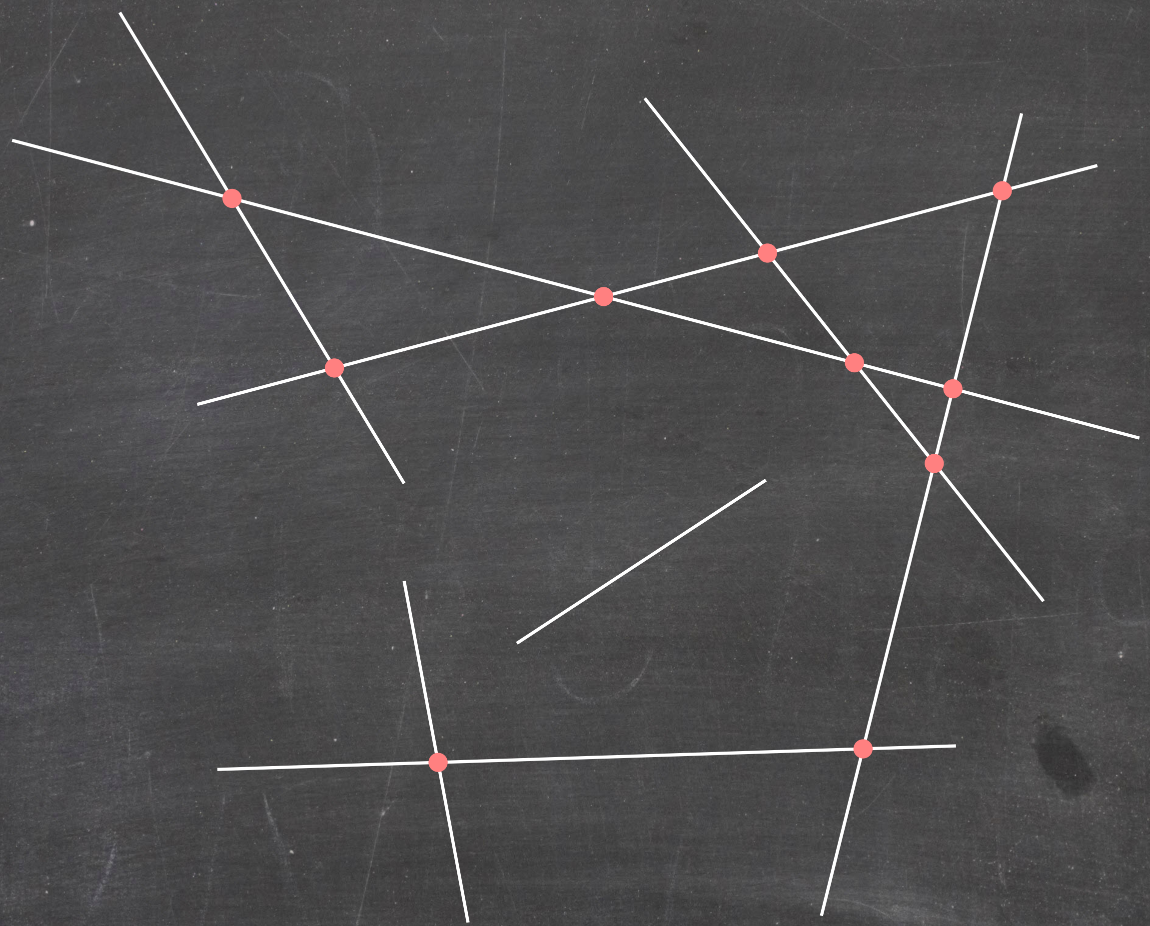
General Line Segment Intersection



Questions:

- Whats' the sweep line status?
  All segments intersecting the sweep line.

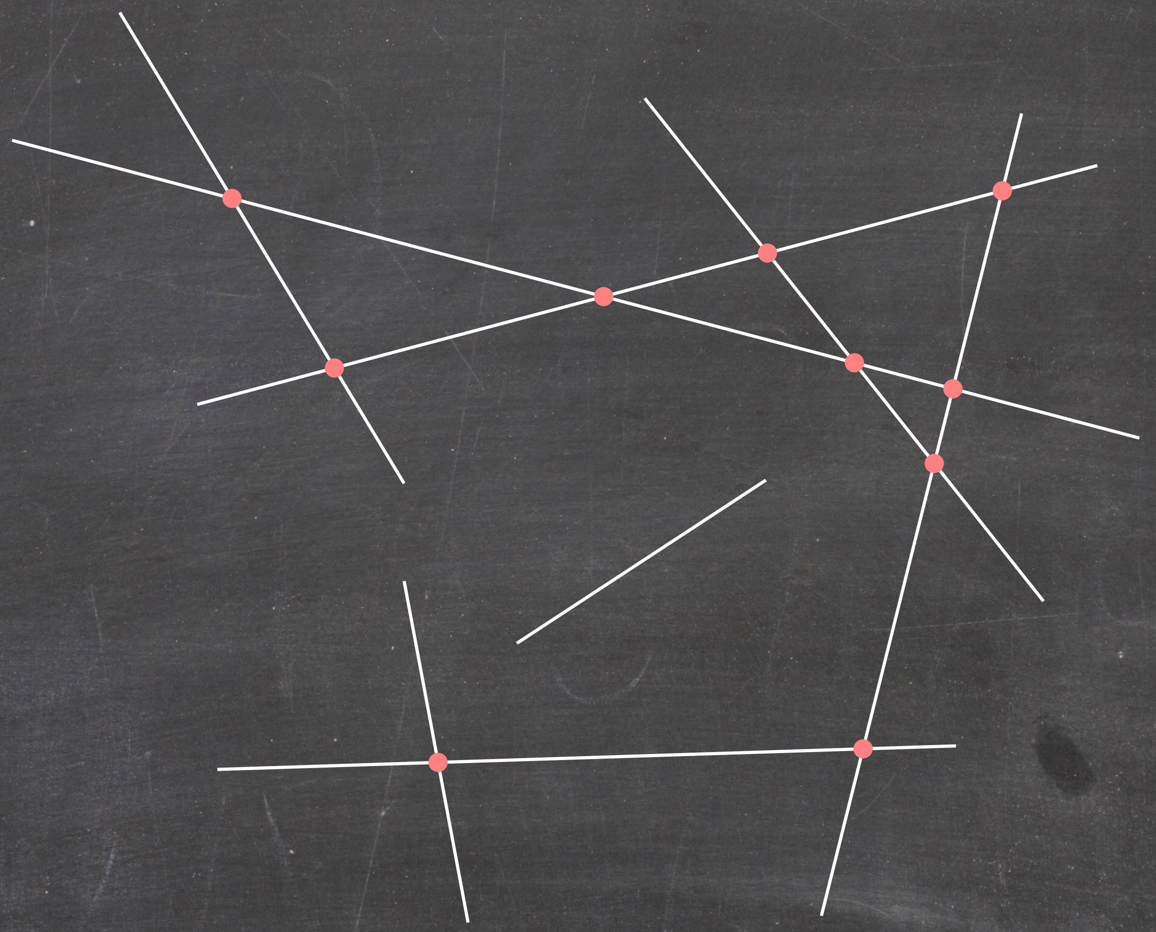
General Line Segment Intersection



Questions:

- Whats' the sweep line status?
  All segments intersecting the sweep line.
- How do we order the segments?

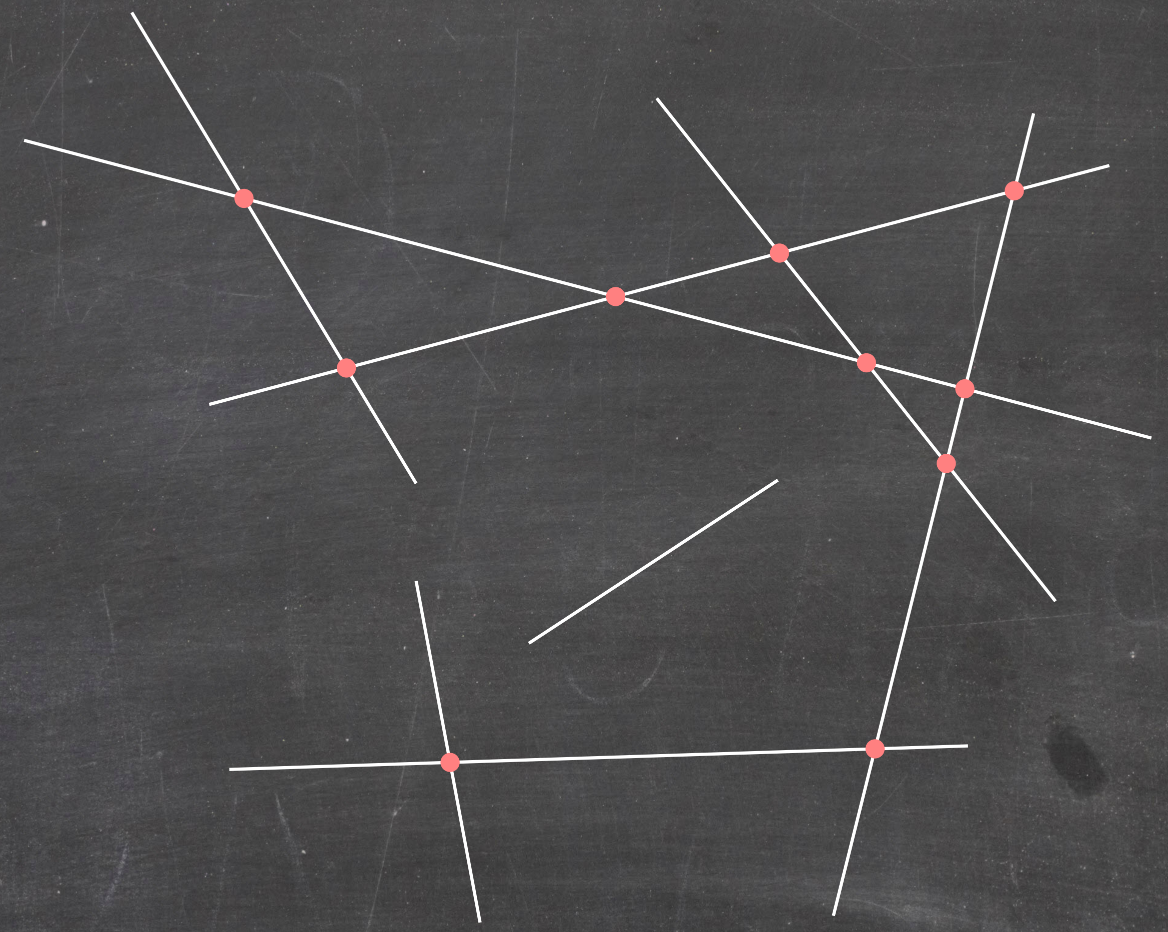
General Line Segment Intersection



Questions:

- Whats' the sweep line status?
  All segments intersecting the sweep line.
- How do we order the segments?
  By the x-coordinates of their intersections with the sweep line.

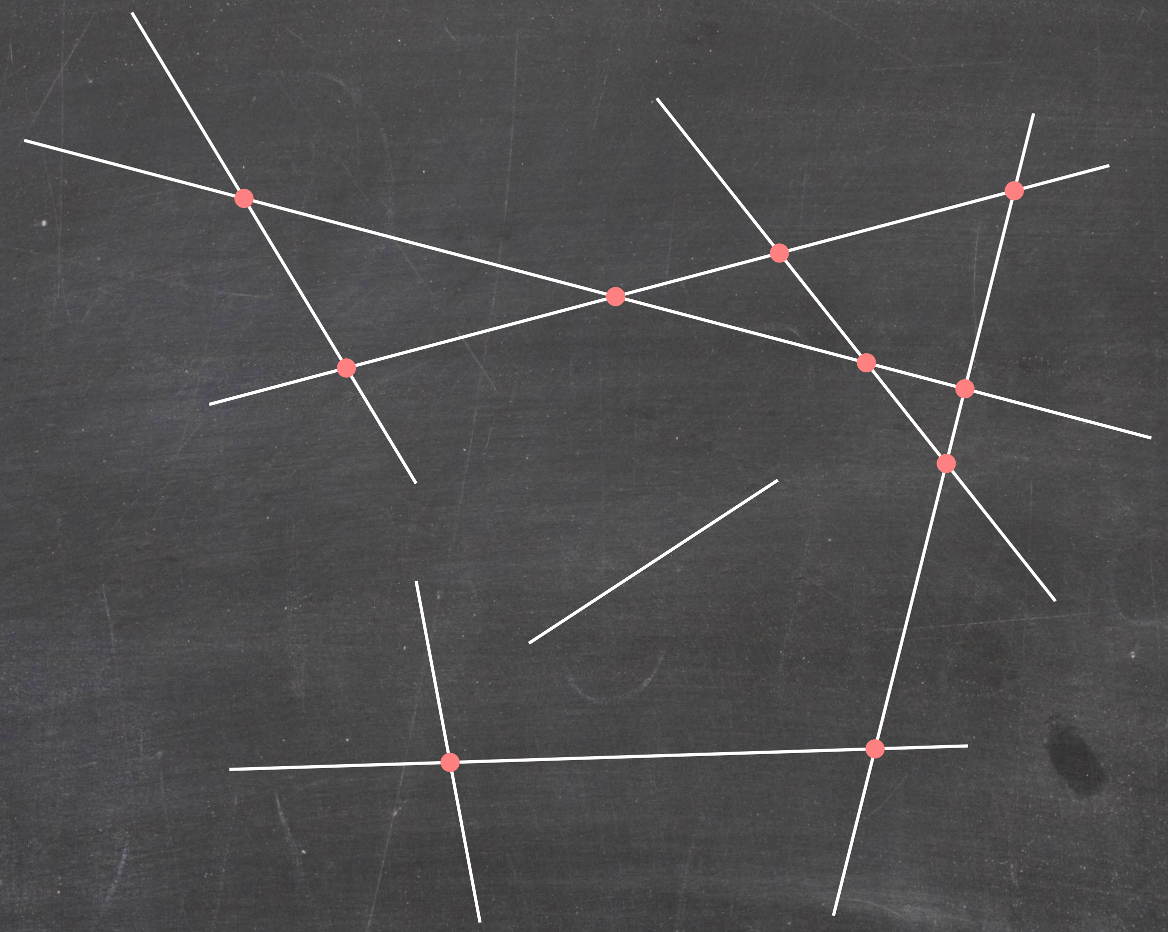
General Line Segment Intersection



Questions:

- Whats' the sweep line status?
  All segments intersecting the sweep line.
- How do we order the segments?
  By the x-coordinates of their intersections with the sweep line.
- Where does the sweep line status change?

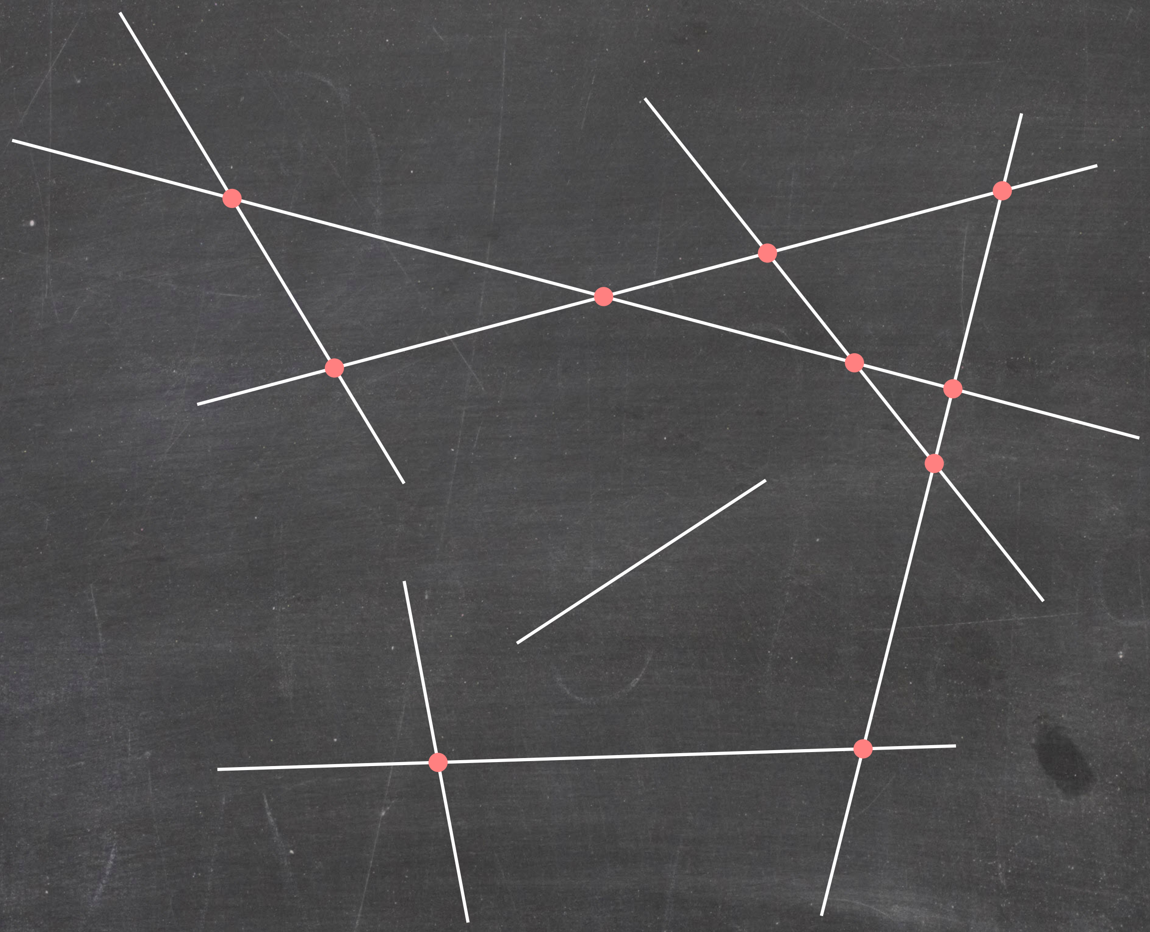
General Line Segment Intersection



Questions:

- Whats' the sweep line status?
  All segments intersecting the sweep line.
- How do we order the segments?
  By the x-coordinates of their intersections with the sweep line.
- Where does the sweep line status change?
  At segment endpoints

General Line Segment Intersection



Questions:

- Whats' the sweep line status?
  All segments intersecting the sweep line.
- How do we order the segments?
  By the x-coordinates of their intersections with the sweep line.
- Where does the sweep line status change?
  At segment endpoints and intersection points!

The Event Schedule

Apparent problem: We want to compute intersection points, but they are part of the event schedule.

The Event Schedule

Apparent problem: We want to compute intersection points, but they are part of the event schedule.

Consequence: We cannot generate all event points ahead of time.

The Event Schedule

Apparent problem: We want to compute intersection points, but they are part of the event schedule.

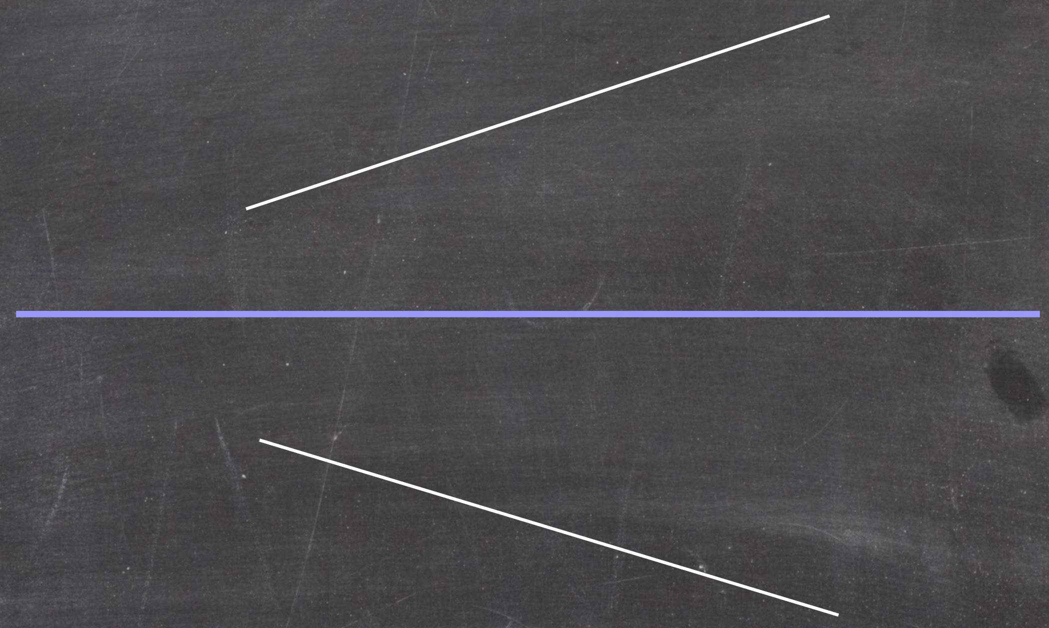
Consequence: We cannot generate all event points ahead of time.

Solution:

- Maintain set of event points sorted by y-coordinates in a priority queue Q (event schedule).
- Initially, Q contains all segment endpoints.
- As we detect intersections, we insert them into Q.

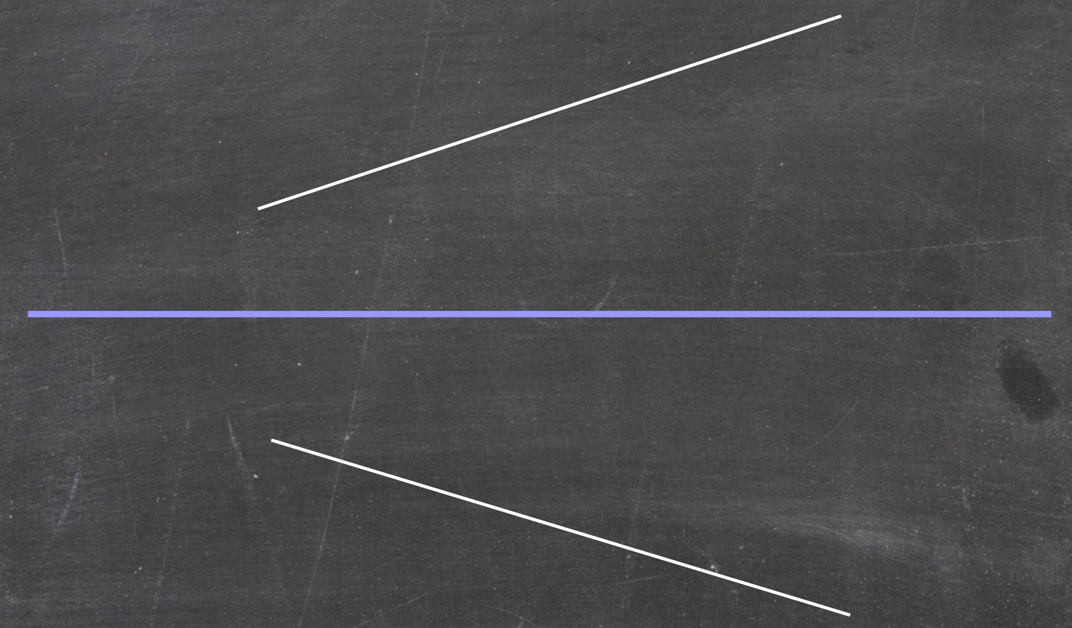
Detecting Intersections: First Attempt

Observation: If two segments s_1 and s_2 intersect, the sweep line must intersect them simultaneously.



Detecting Intersections: First Attempt

Observation: If two segments s_1 and s_2 intersect, the sweep line must intersect them simultaneously.

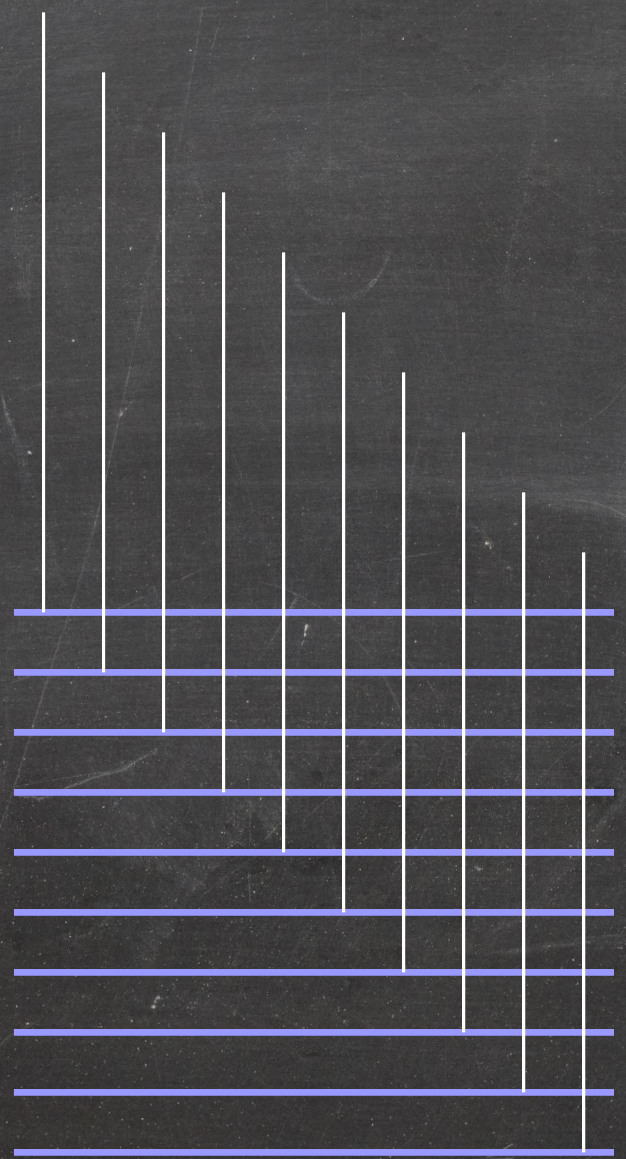


Idea:

- As in the orthogonal case, insert and delete segments into and from T when the sweep line passes their endpoints.
- When inserting a segment into T, test for intersections with all segments already in T.

Too Many Tests

Problem: We may still perform a quadratic number of intersection tests only to discover that there are no intersections.

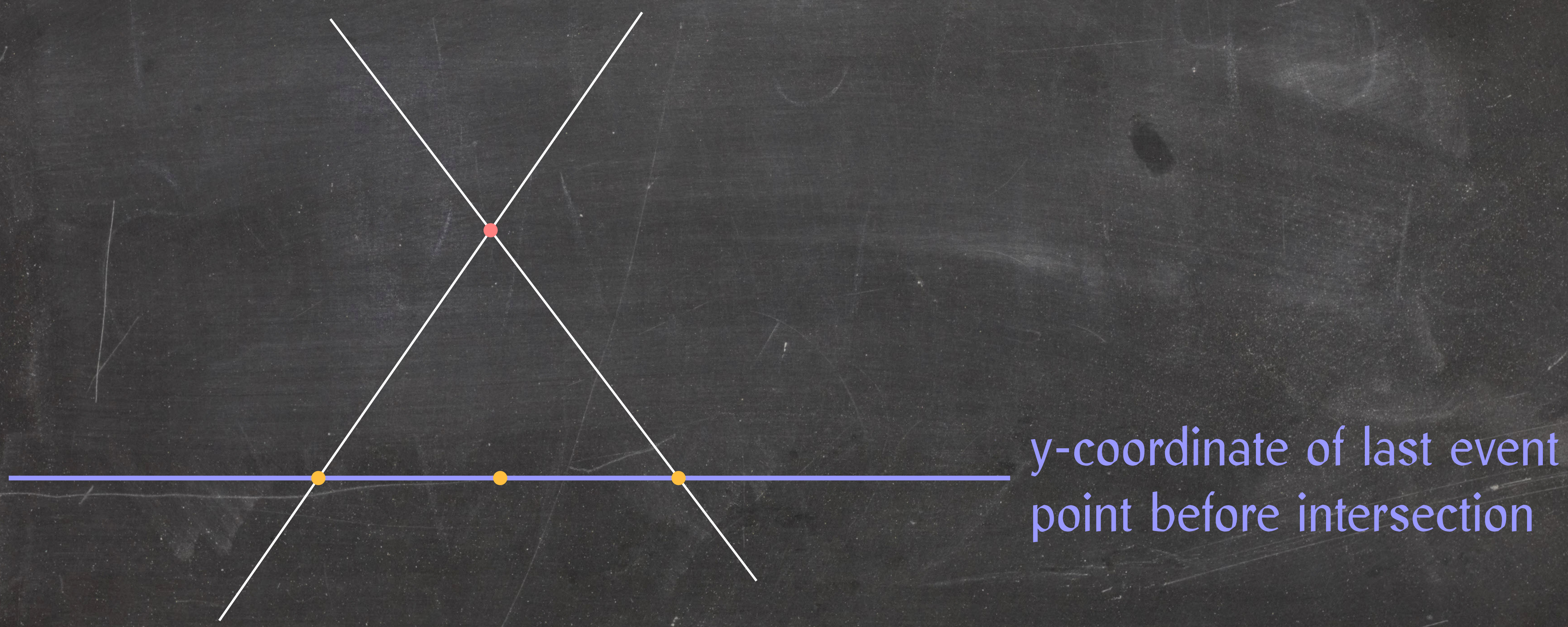


Detecting Intersection Points Lazily

Observation: Two segments s_1 and s_2 that intersect are adjacent in T immediately before they intersect.

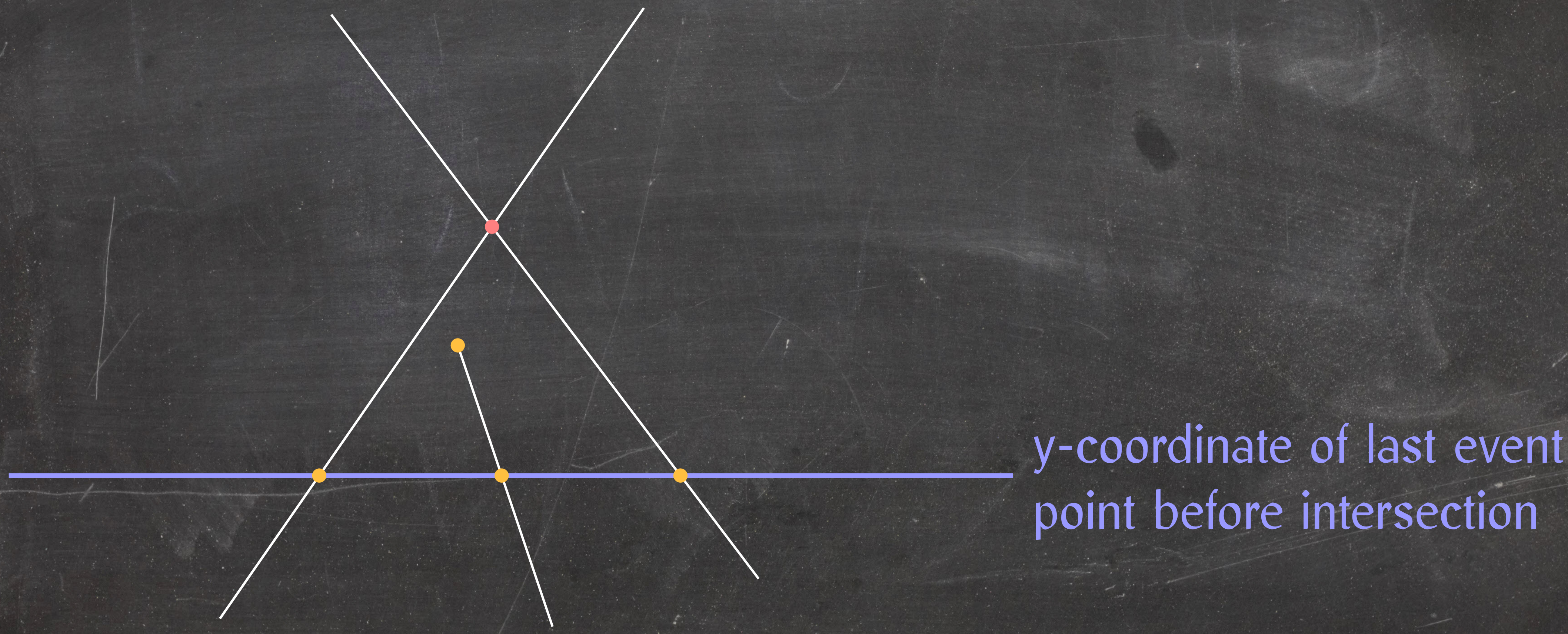
Detecting Intersection Points Lazily

Observation: Two segments s_1 and s_2 that intersect are adjacent in T immediately before they intersect.



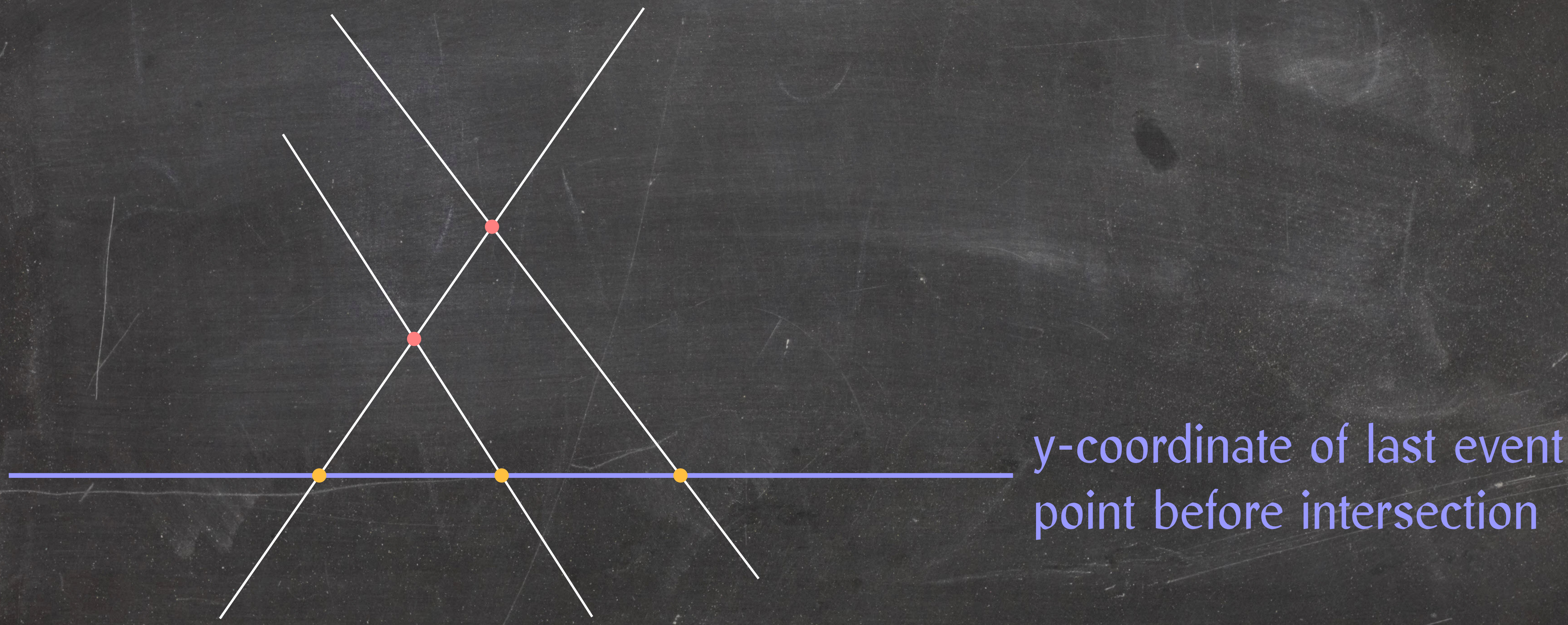
Detecting Intersection Points Lazily

Observation: Two segments s_1 and s_2 that intersect are adjacent in T immediately before they intersect.



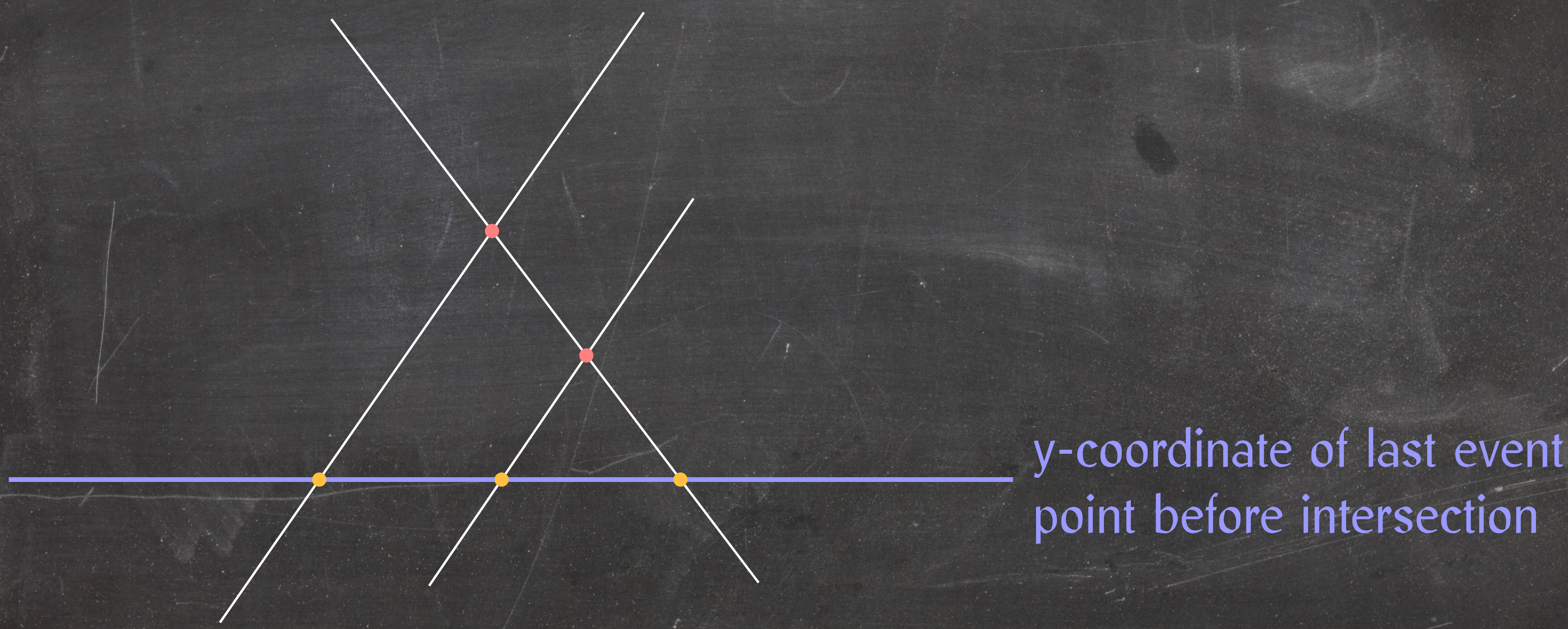
Detecting Intersection Points Lazily

Observation: Two segments s_1 and s_2 that intersect are adjacent in T immediately before they intersect.



Detecting Intersection Points Lazily

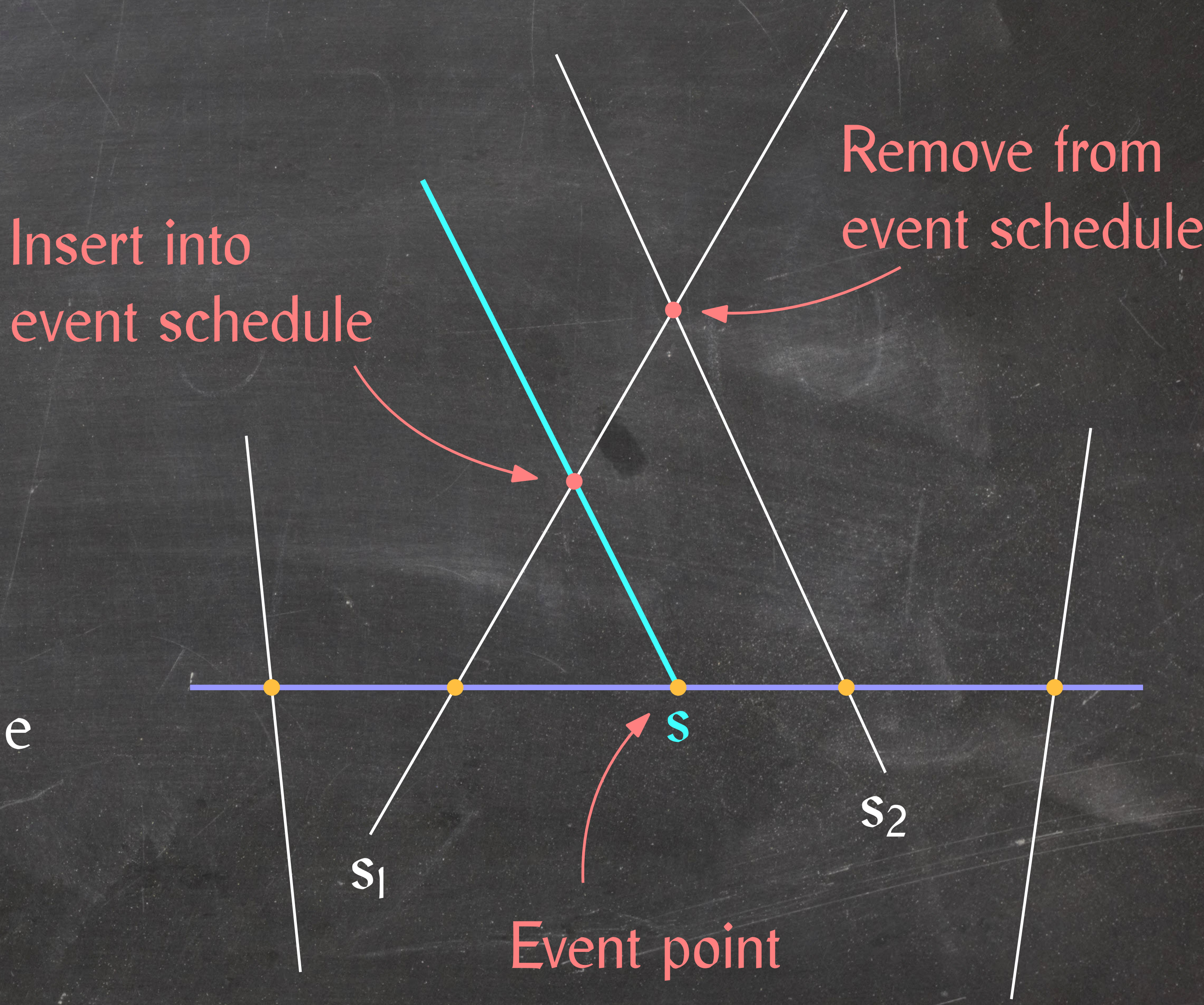
Observation: Two segments s_1 and s_2 that intersect are adjacent in T immediately before they intersect.



Event Points

Bottom endpoint:

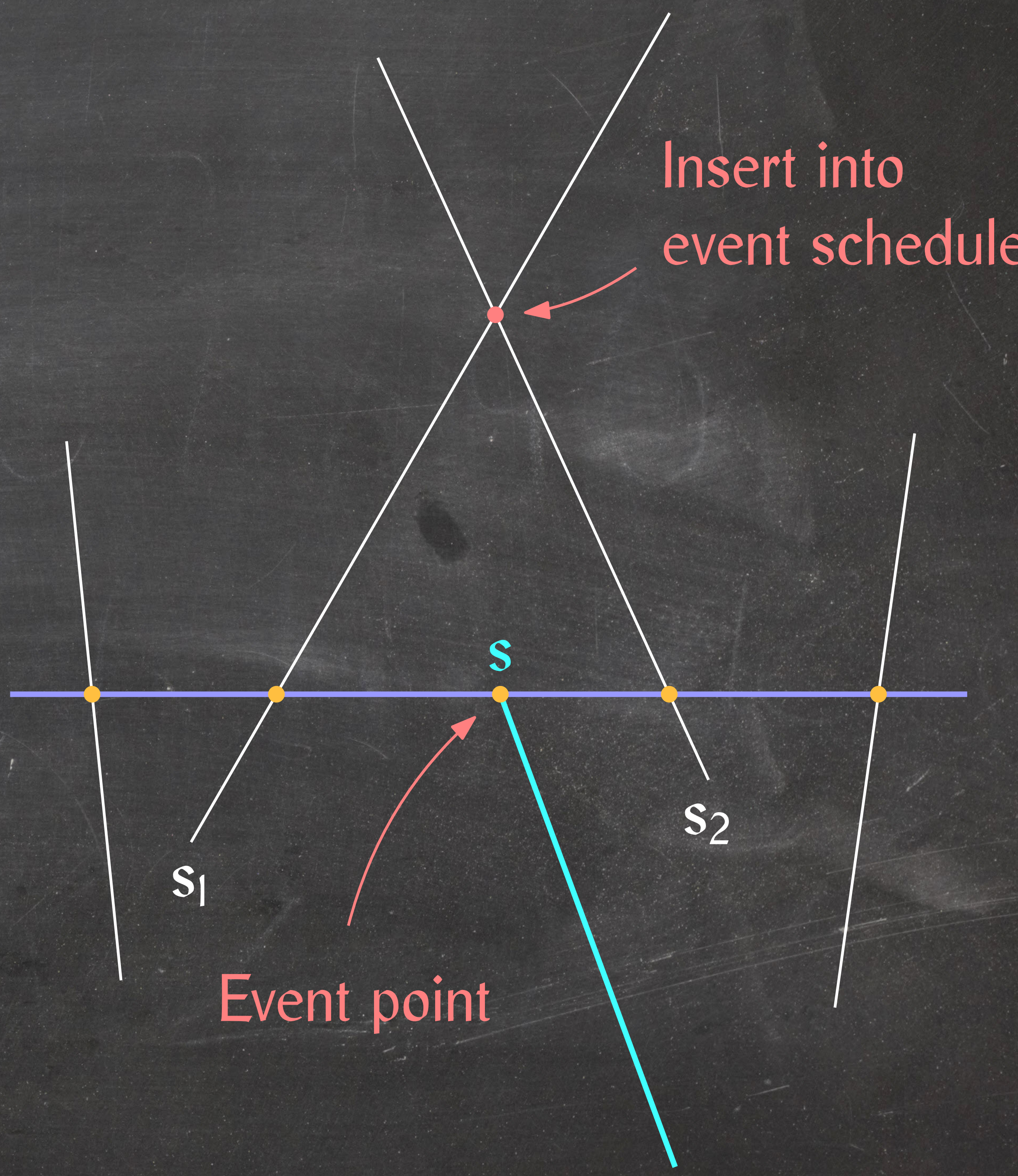
- Insert s into T and test for intersections with its two neighbours.
- If there are intersections, insert them into the event schedule.
- If s_1 and s_2 intersect after the current y-coordinate, remove the intersection from the event schedule.



Event Points

Top endpoint:

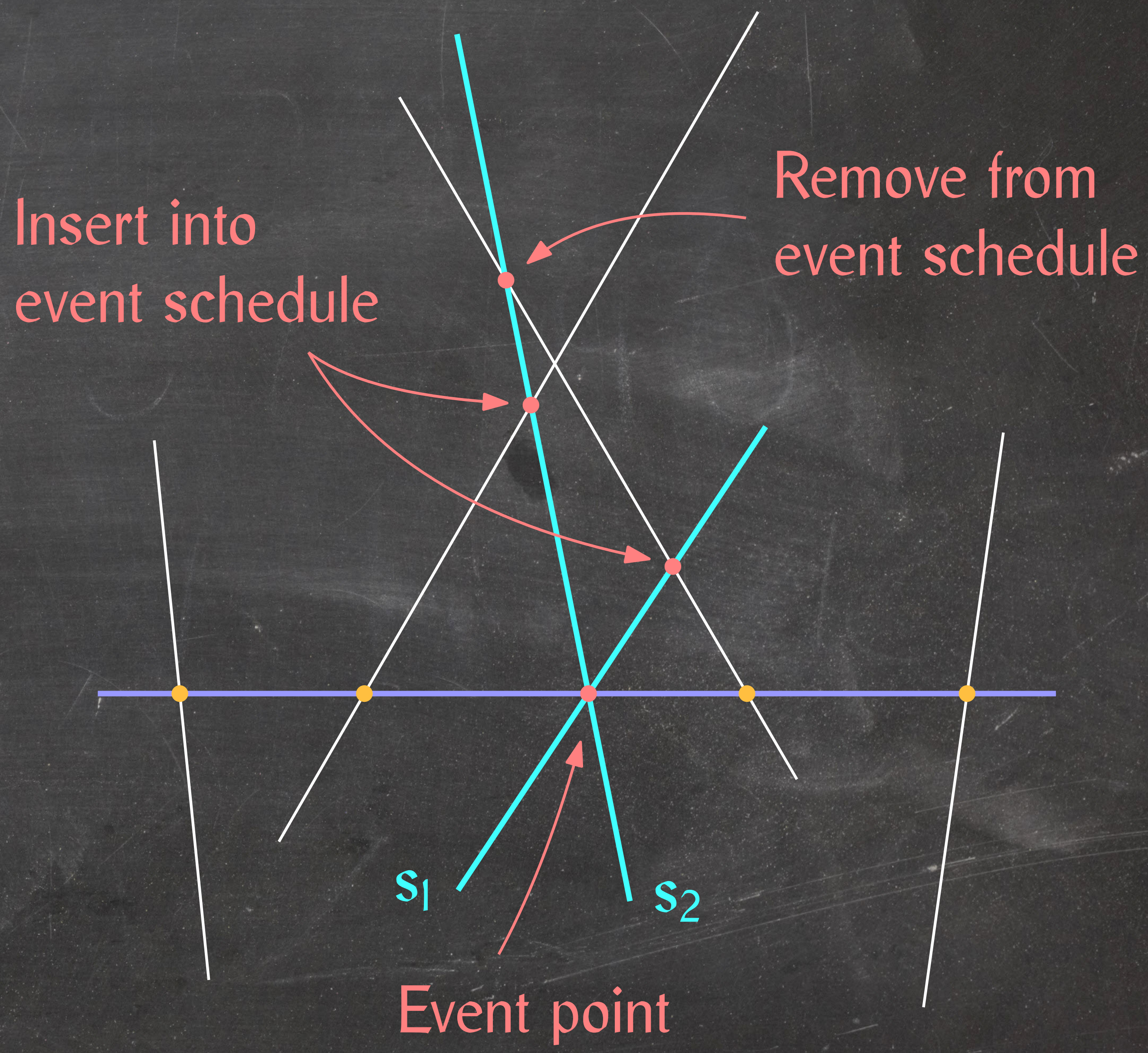
- Delete s from T.
- Test for intersections between the two segments that become adjacent.
- If they intersect after the current y-coordinate, insert the intersection into the event schedule.



Event Points

Intersection point:

- Report the intersection.
- Swap the order of the two intersecting segments.
- Remove intersections with their old neighbours from the event schedule.
- Test for intersections with their new neighbours and insert them into the event schedule if they are above the current y-coordinate.



General Line Segment Intersection: Analysis

$2n + k$ event points:

- n bottom endpoints
- n top endpoints
- k intersection points

- Each event point incurs $O(1)$ updates and queries of sweep line structure and event schedule.

$\Rightarrow$ Cost per event point $= O(\lg n)$

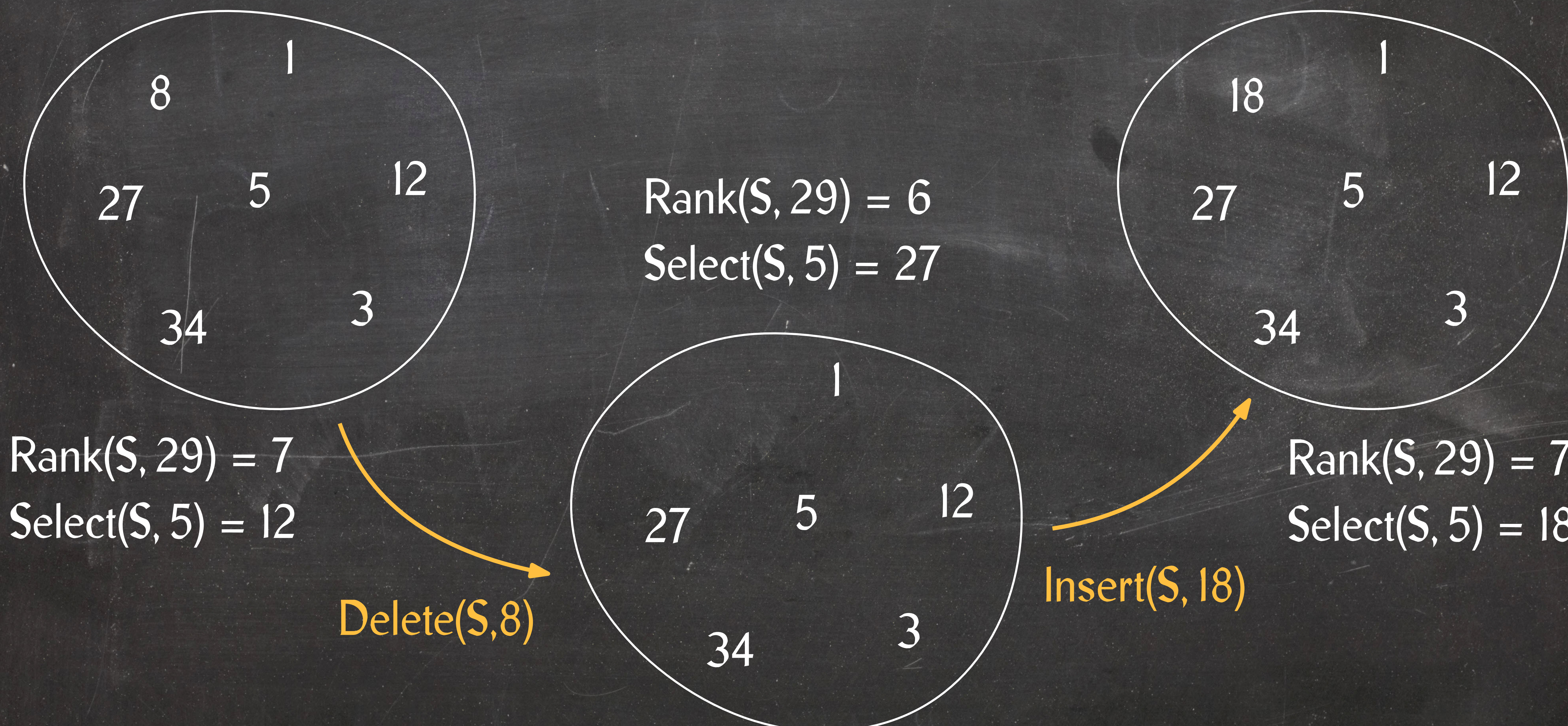
Theorem: The general line segment intersection problem can be solved in $O((n + k) \lg n)$.

Dynamic Rank and Select

Problem: Maintain a set S of numbers under insertions and deletions and support the following two types of queries:

Rank(S, x)   Count the number of elements in S less than x, plus 1.

Select(S, k)   Report the kth smallest element in S.



Orthogonal Line Segment Intersection Counting

Problem: Instead of reporting all intersections between horizontal and vertical segments, only count how many there are.

Orthogonal Line Segment Intersection Counting

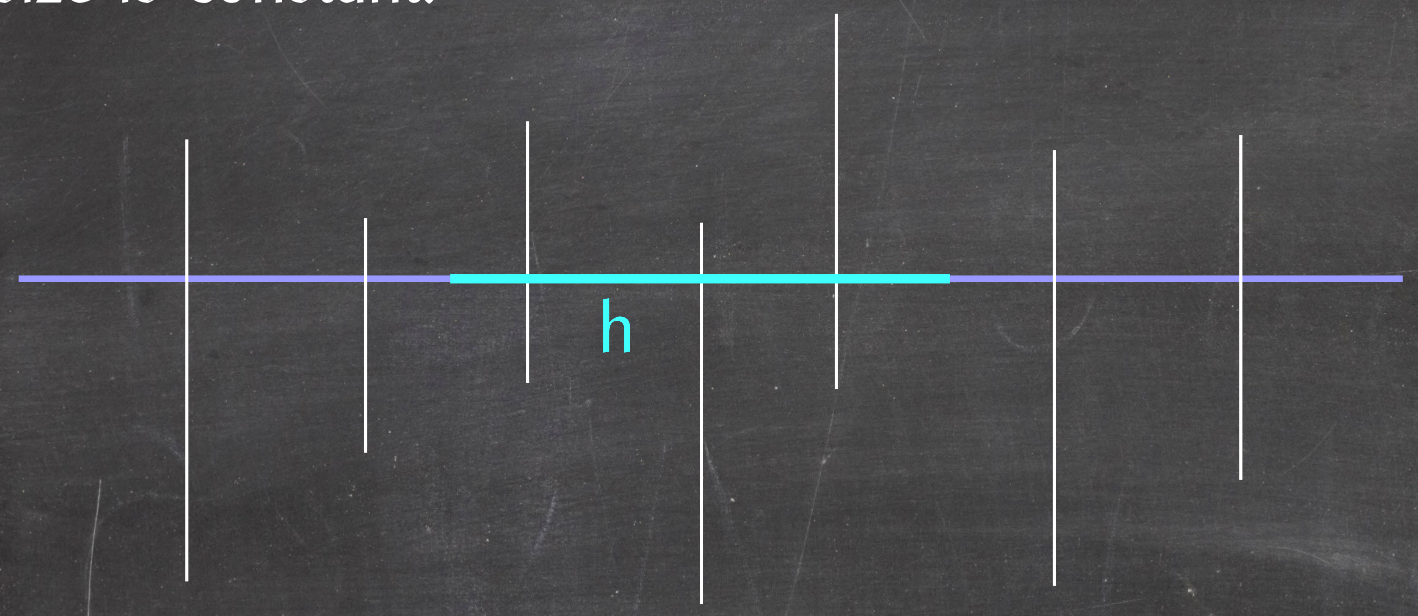
Problem: Instead of reporting all intersections between horizontal and vertical segments, only count how many there are.

We can do this in $O(n \lg n + k)$ time (how?), but the $O(k)$ is no longer justified: the output size is constant.

Orthogonal Line Segment Intersection Counting

Problem: Instead of reporting all intersections between horizontal and vertical segments, only count how many there are.

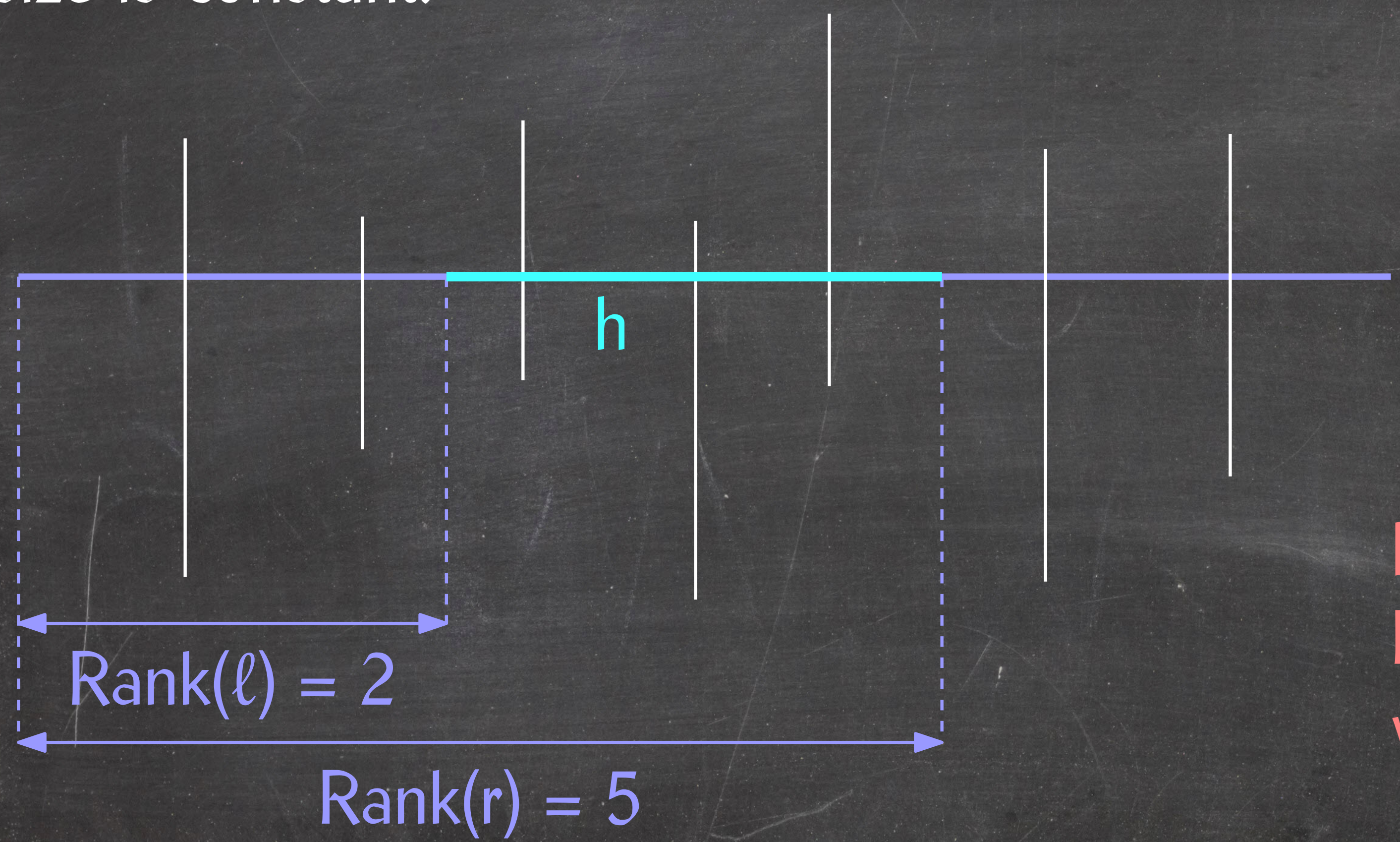
We can do this in $O(n \lg n + k)$ time (how?), but the $O(k)$ is no longer justified: the output size is constant.



Orthogonal Line Segment Intersection Counting

Problem: Instead of reporting all intersections between horizontal and vertical segments, only count how many there are.

We can do this in $O(n \lg n + k)$ time (how?), but the $O(k)$ is no longer justified: the output size is constant.

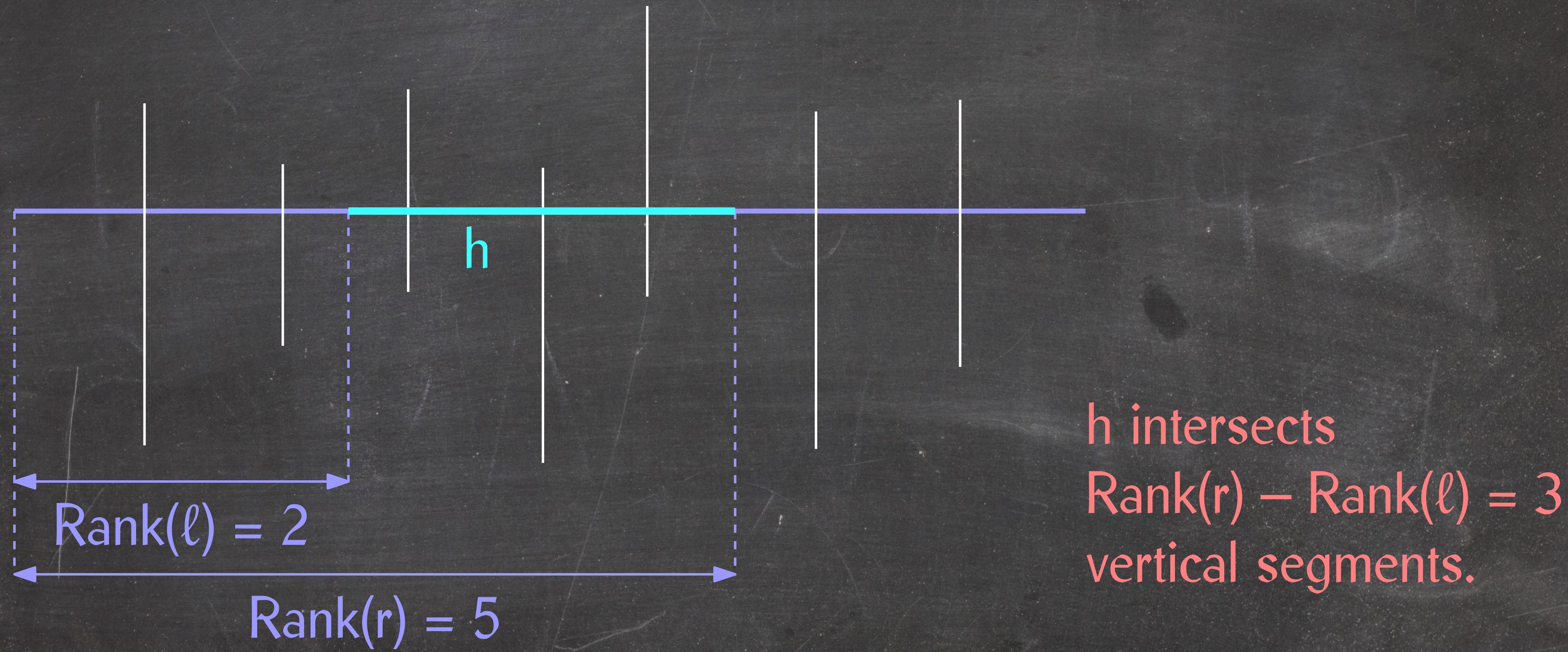


h intersects
Rank(r) − Rank(ℓ) = 3
vertical segments.

Orthogonal Line Segment Intersection Counting

Problem: Instead of reporting all intersections between horizontal and vertical segments, only count how many there are.

We can do this in $O(n \lg n + k)$ time (how?), but the $O(k)$ is no longer justified: the output size is constant.

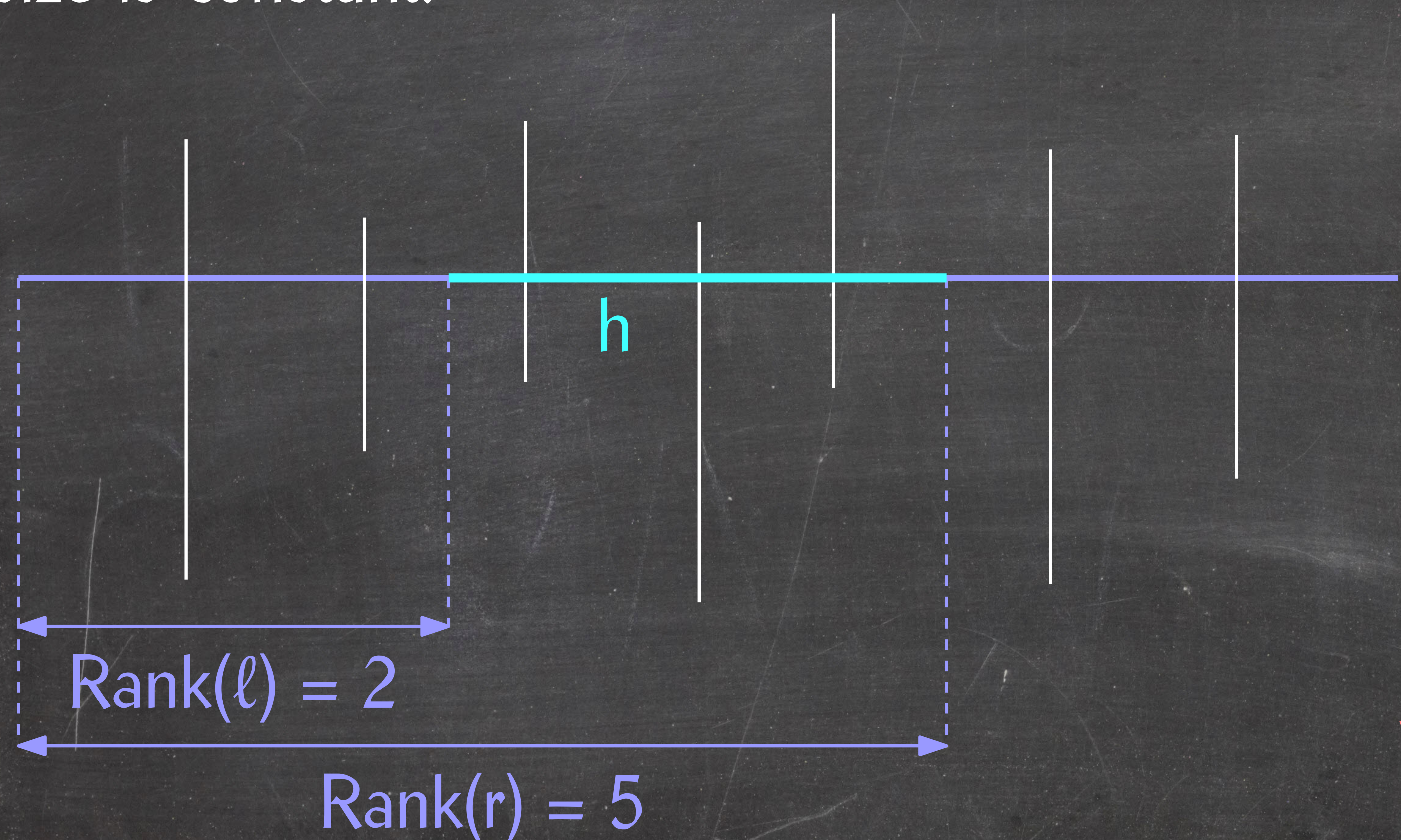


Instead of asking a RangeFind query for every horizontal segment, ask two Rank queries.

Orthogonal Line Segment Intersection Counting

Problem: Instead of reporting all intersections between horizontal and vertical segments, only count how many there are.

We can do this in $O(n \lg n + k)$ time (how?), but the $O(k)$ is no longer justified: the output size is constant.



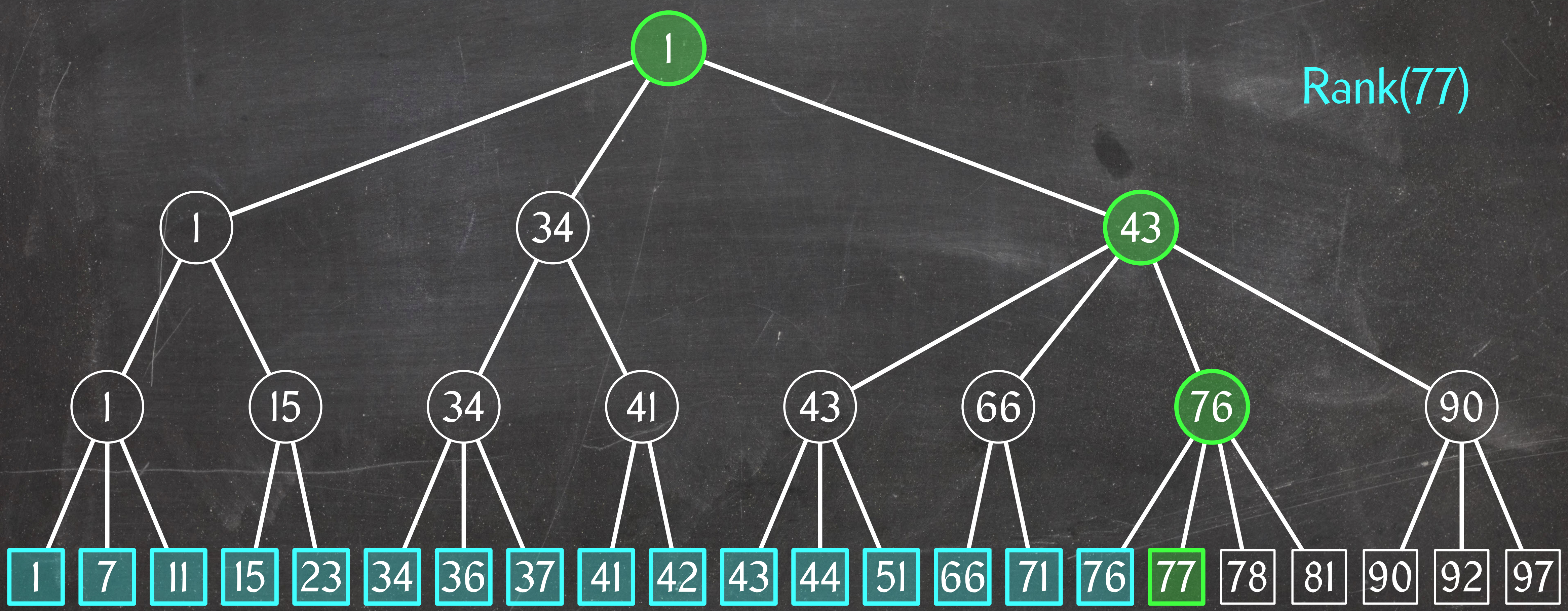
h intersects
Rank(r) − Rank(ℓ) = 3
vertical segments.

Instead of asking a RangeFind query for every horizontal segment, ask two Rank queries.

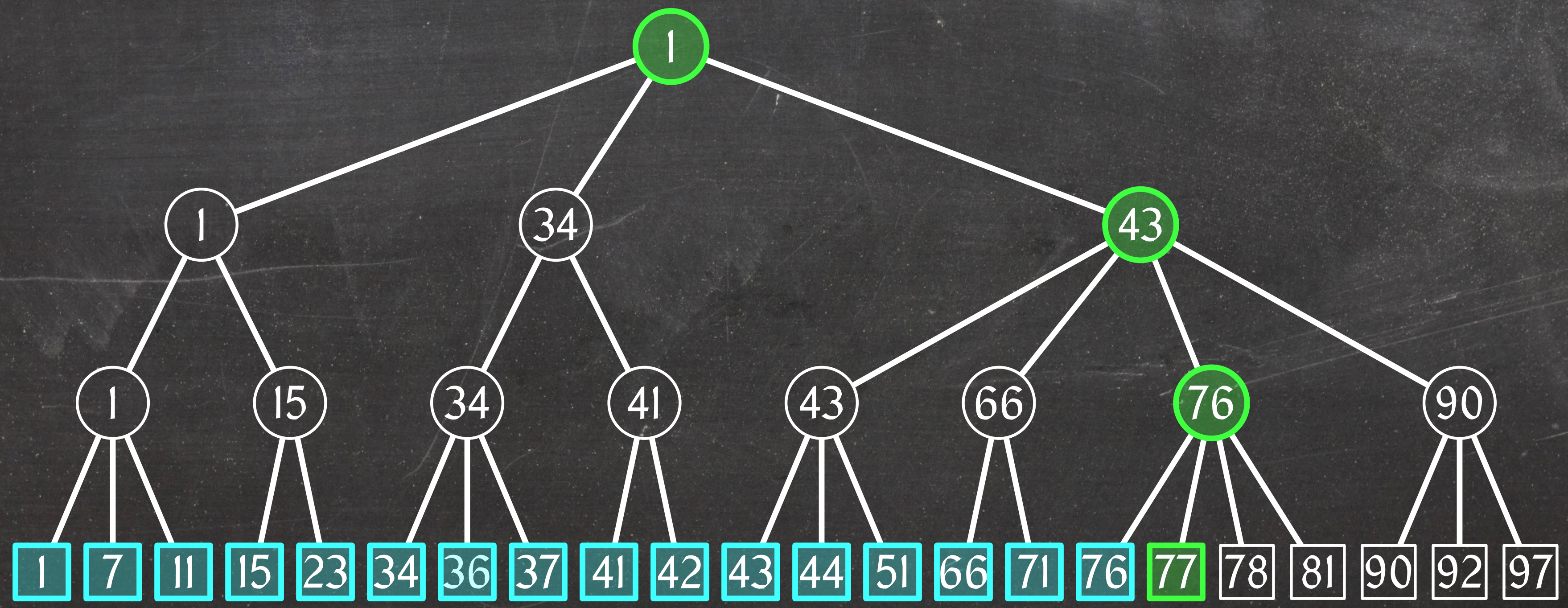
Lemma: If Insert, Delete, and Rank operations can be supported in $O(\lg n)$ time, the orthogonal line segment intersection counting problem can be solved in $O(n \lg n)$ time.

Rank and Select Queries on (a, b)-Trees

Observation: The rank of an element x is one more than the number of leaves to the left of the path to the leaf corresponding to x.



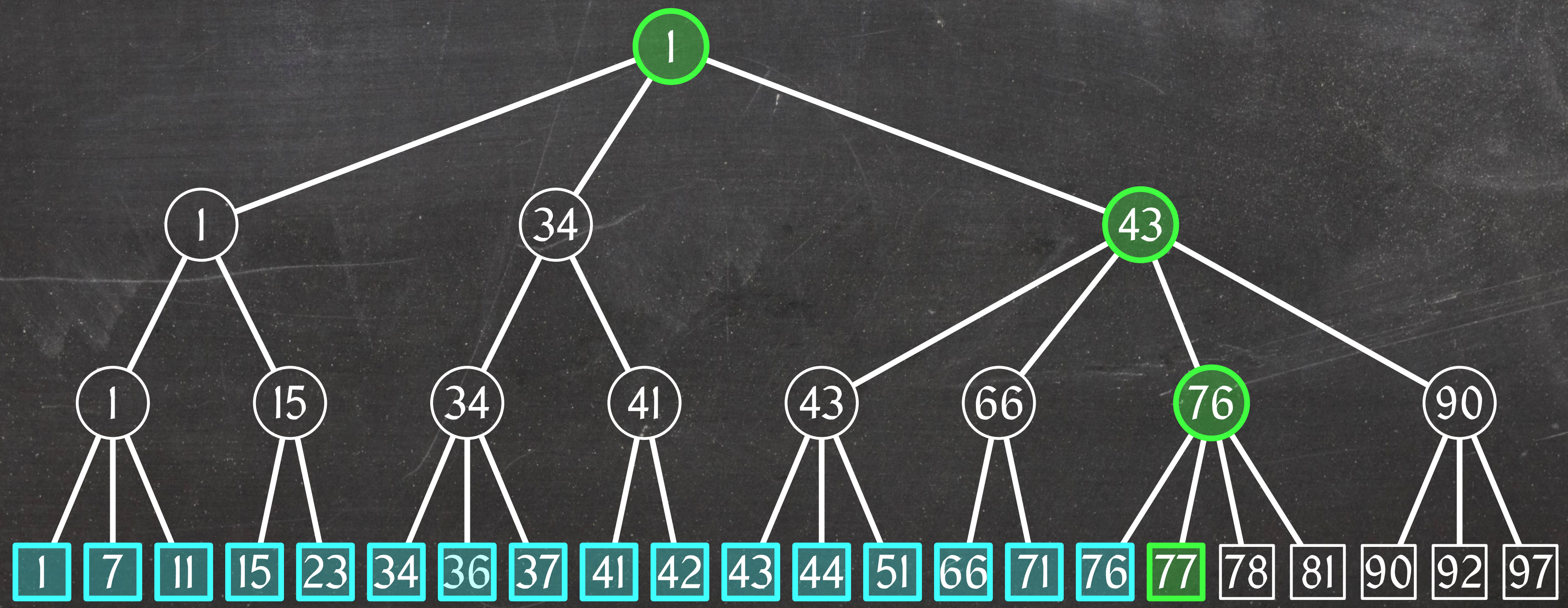
Augmenting Data Structures is a Balancing Act



Augmenting Data Structures is a Balancing Act

Option 1: Just use a plain (a, b)-tree

- Fast updates: $O(\lg n)$
- Slow queries: Potentially $O(n)$ using RangeFind



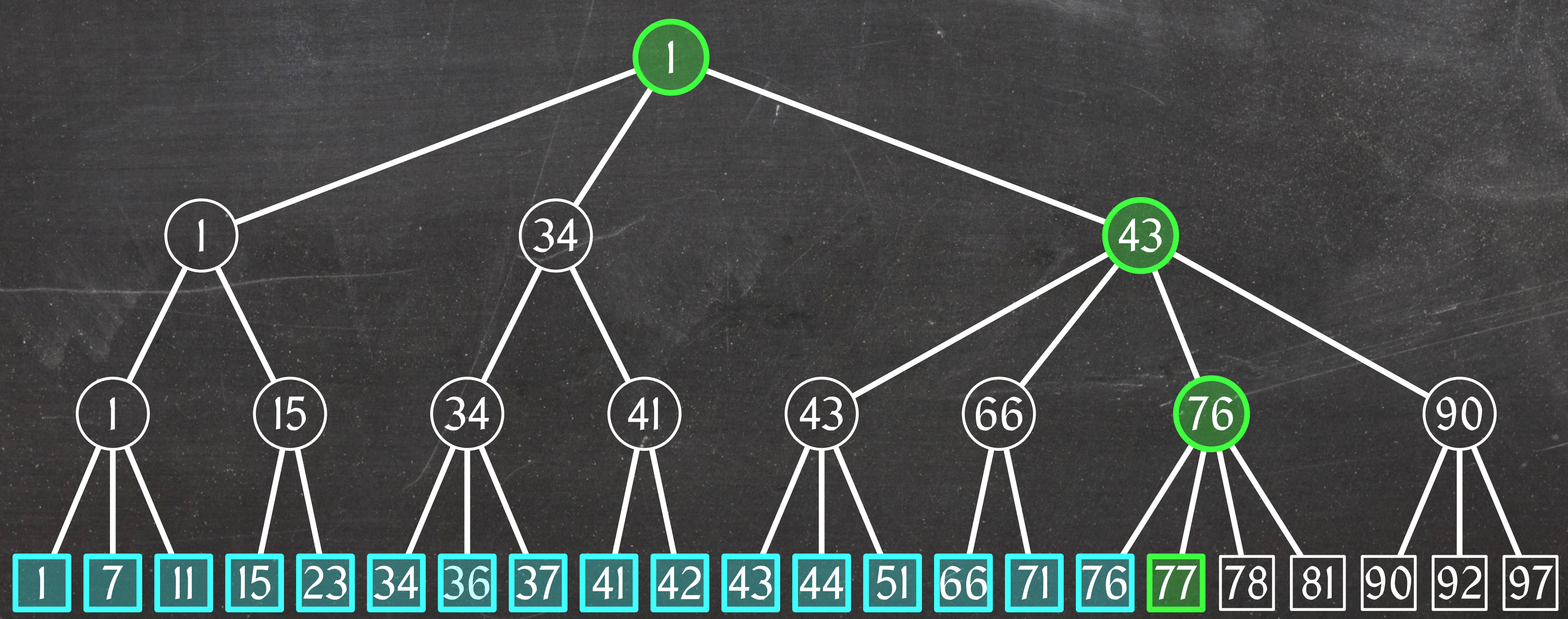
Augmenting Data Structures is a Balancing Act

Option 1: Just use a plain (a, b)-tree

- Fast updates: O(lg n)
- Slow queries: Potentially O(n) using RangeFind

Option 2: Store each leaf's rank explicitly

- Fast queries: O(lg n)
- Slow updates: Inserting a new minimum element causes all ranks to change.



Augmenting Data Structures is a Balancing Act

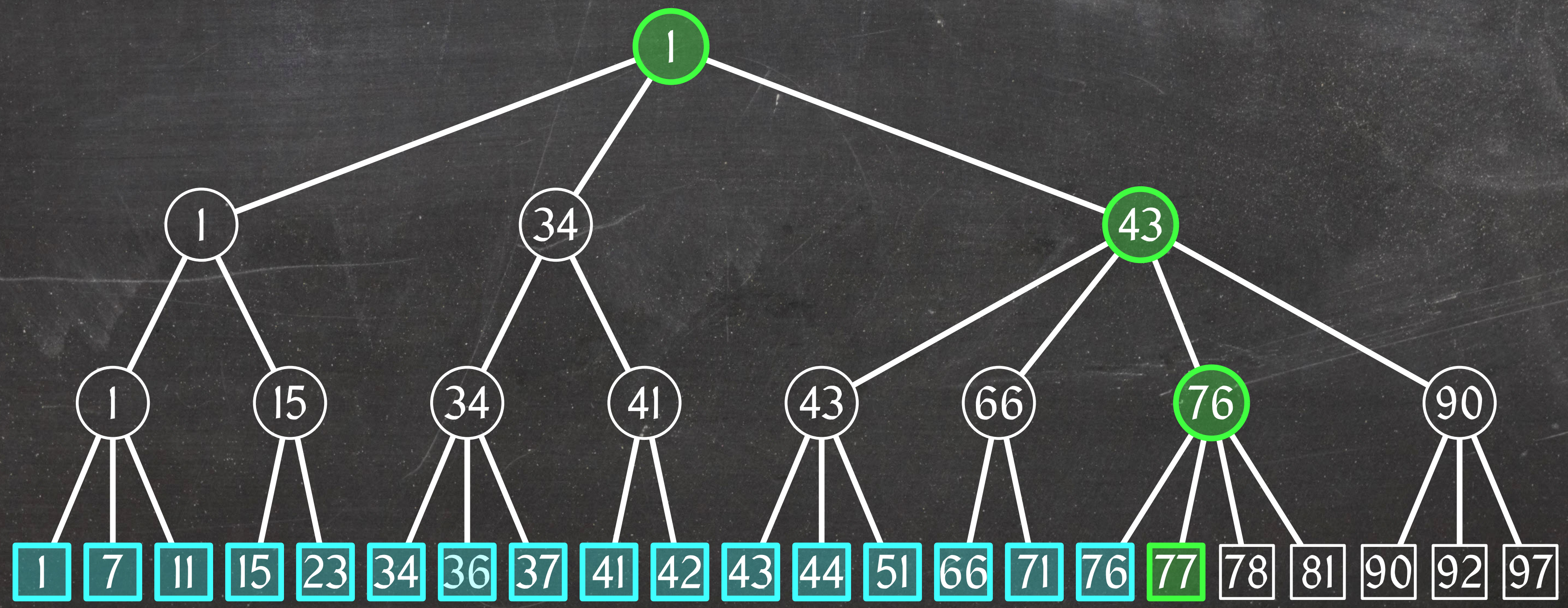
Option 1: Just use a plain (a, b)-tree

- Fast updates: O(lg n)
- Slow queries: Potentially O(n) using RangeFind

All the work happens during queries.

Option 2: Store each leaf's rank explicitly

- Fast queries: O(lg n)
- Slow updates: Inserting a new minimum element causes all ranks to change.



Augmenting Data Structures is a Balancing Act

Option 1: Just use a plain (a, b)-tree

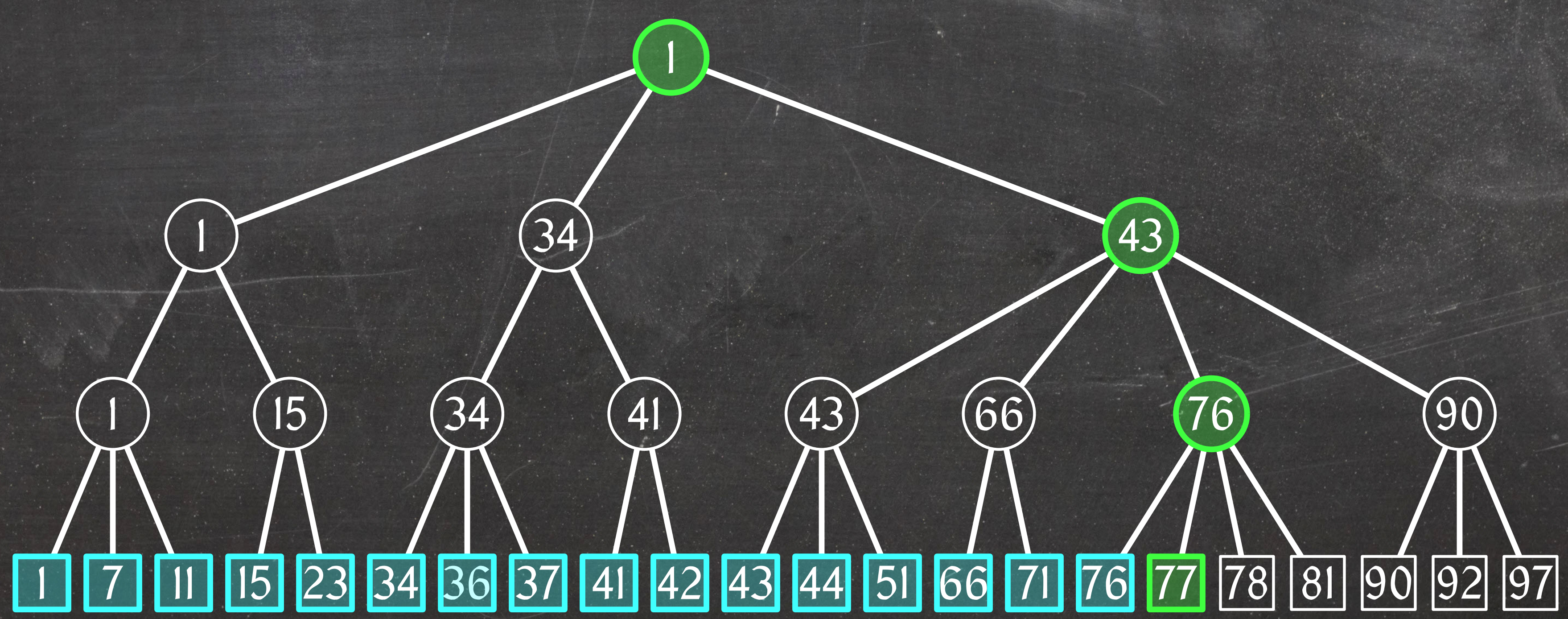
- Fast updates: O(lg n)
- Slow queries: Potentially O(n) using RangeFind

All the work happens during queries.

Option 2: Store each leaf's rank explicitly

- Fast queries: O(lg n)
- Slow updates: Inserting a new minimum element causes all ranks to change.

All the work happens during updates.



Augmenting Data Structures is a Balancing Act

Option 1: Just use a plain (a, b)-tree

- Fast updates: O(lg n)
- Slow queries: Potentially O(n) using RangeFind

All the work happens during queries.

Option 2: Store each leaf's rank explicitly

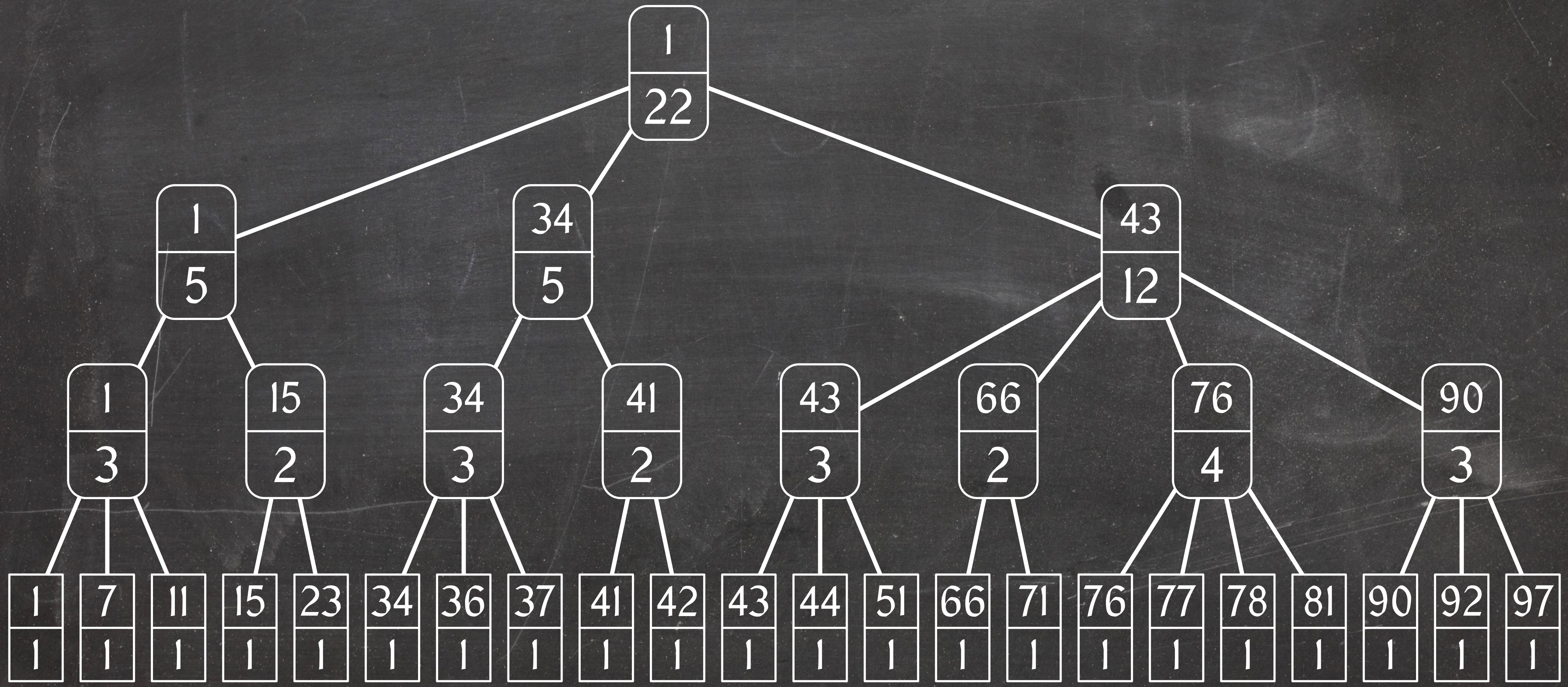
- Fast queries: O(lg n)
- Slow updates: Inserting a new minimum element causes all ranks to change.

All the work happens during updates.

Can we make updates compute some information that is cheap to compute and still helps speed up queries?

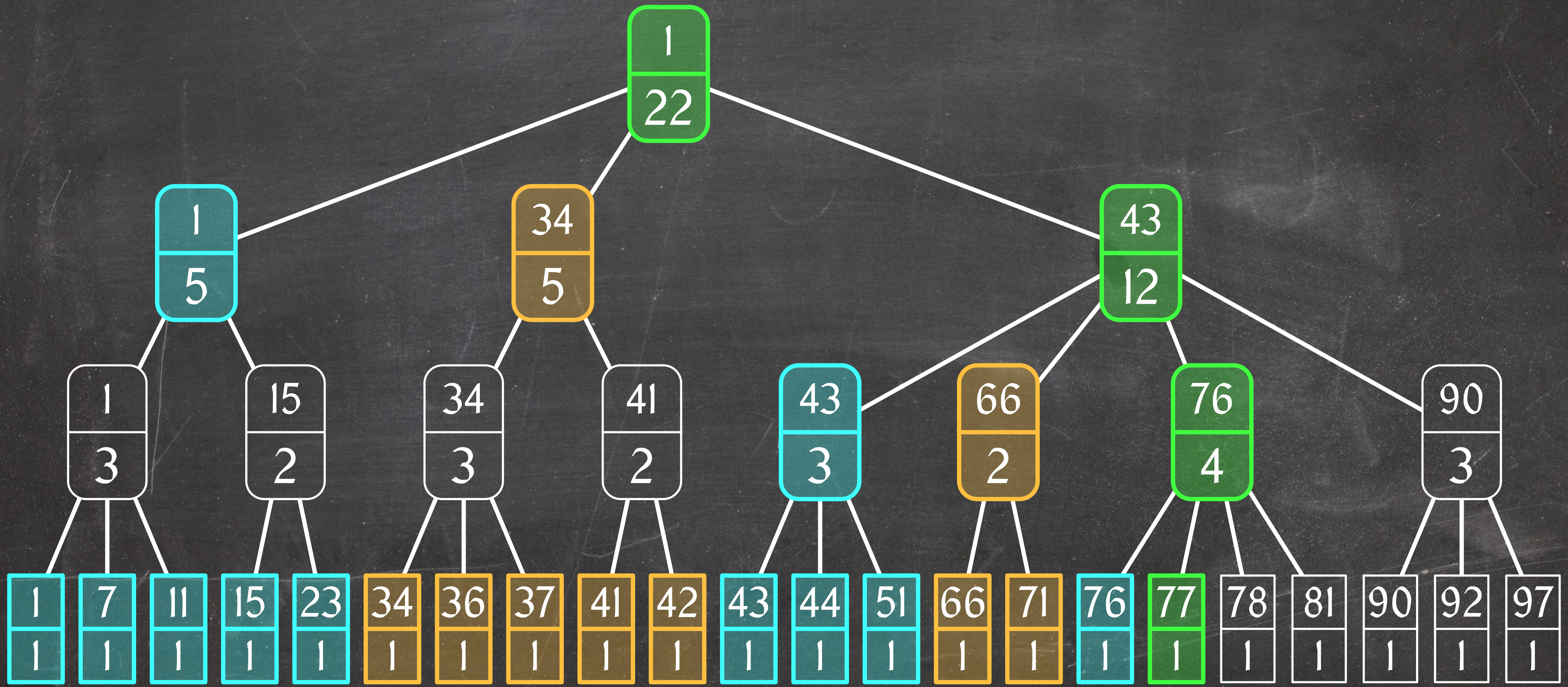
A Rank-Select Tree

In addition to the standard information, each node stores the number of leaves in its subtree.



Rank Queries

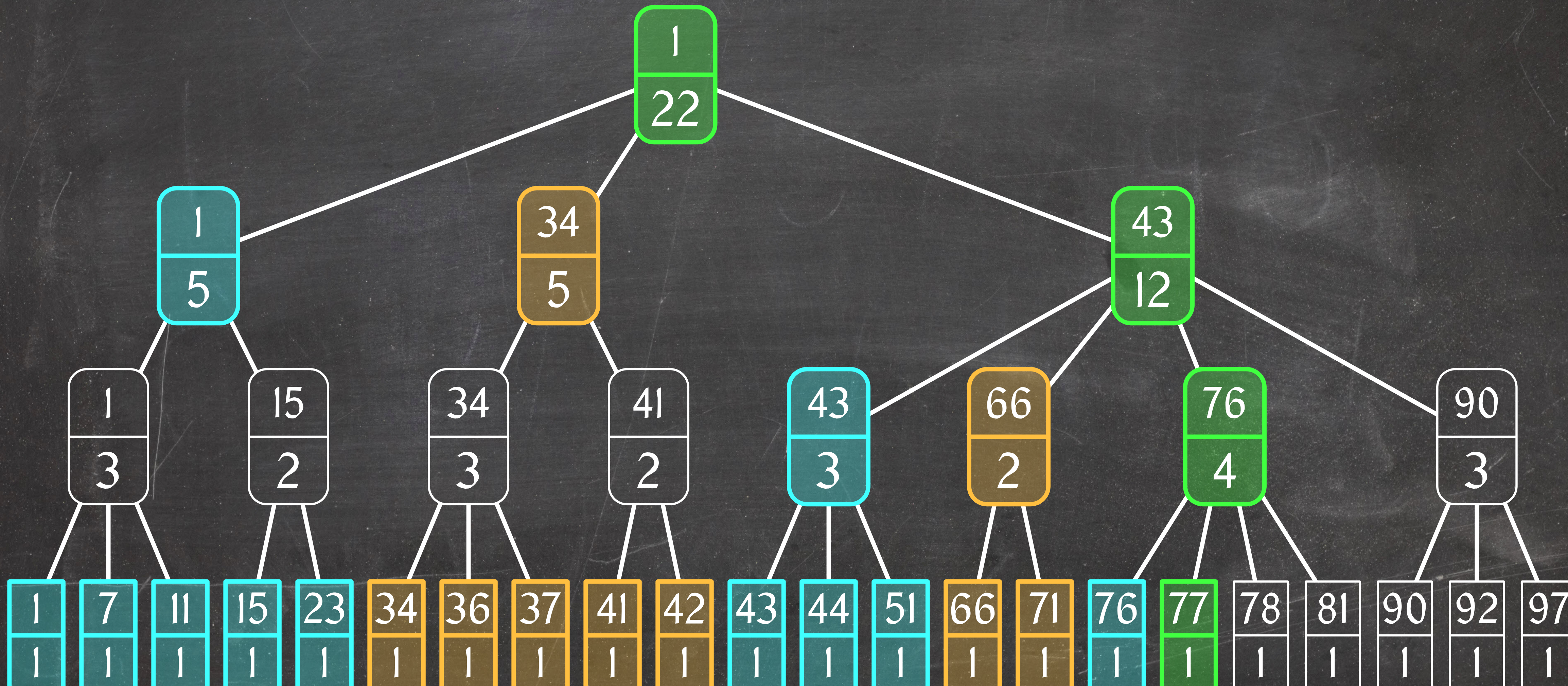
Lemma: Rank queries can be answered in $O(\lg n)$ time using a Rank-Select tree.



$\text{Rank}(77) = 5 + 5 + 3 + 2 + 1 + 1 = 17$

Select Queries

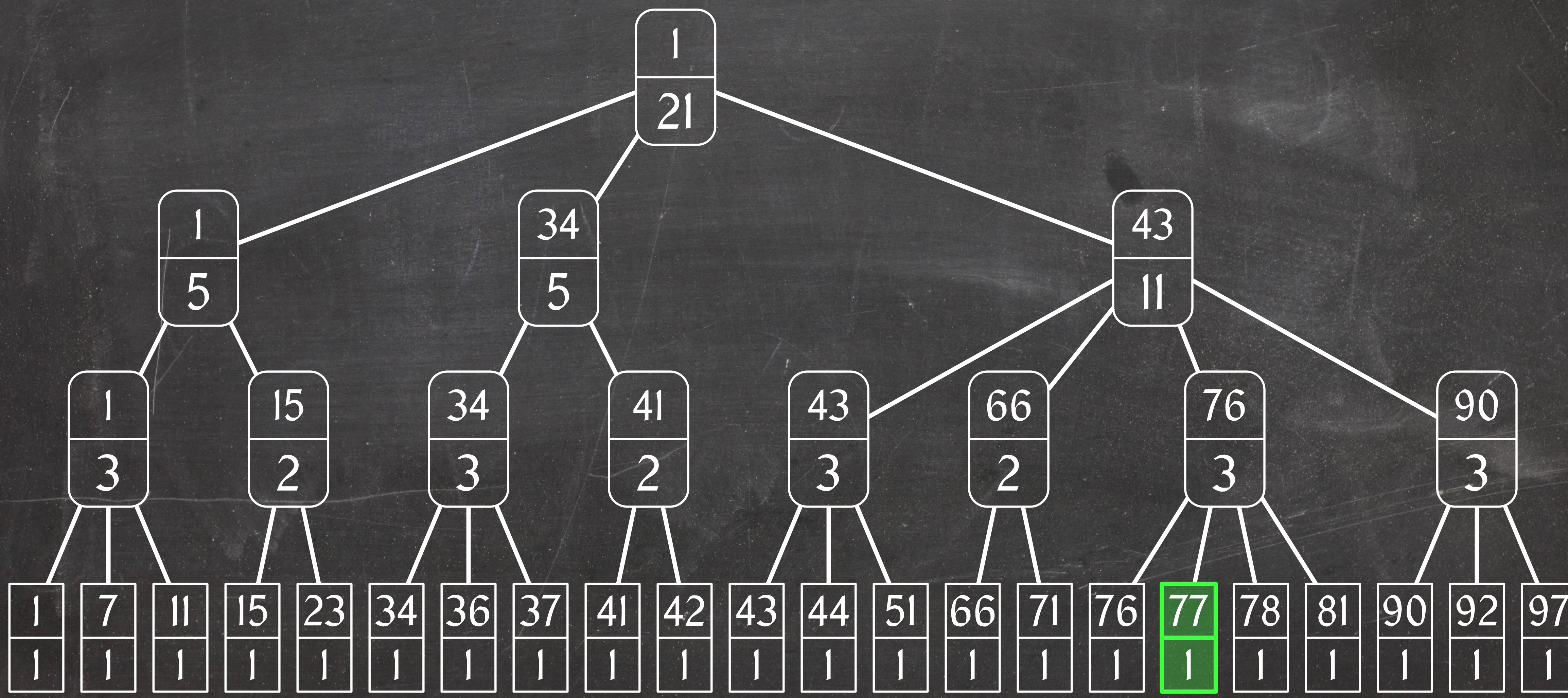
Lemma: Select queries can be answered in O(lg n) time using a Rank-Select tree.



Rank(77) = 5 + 5 + 3 + 2 + 1 + 1 = 17

Insertions

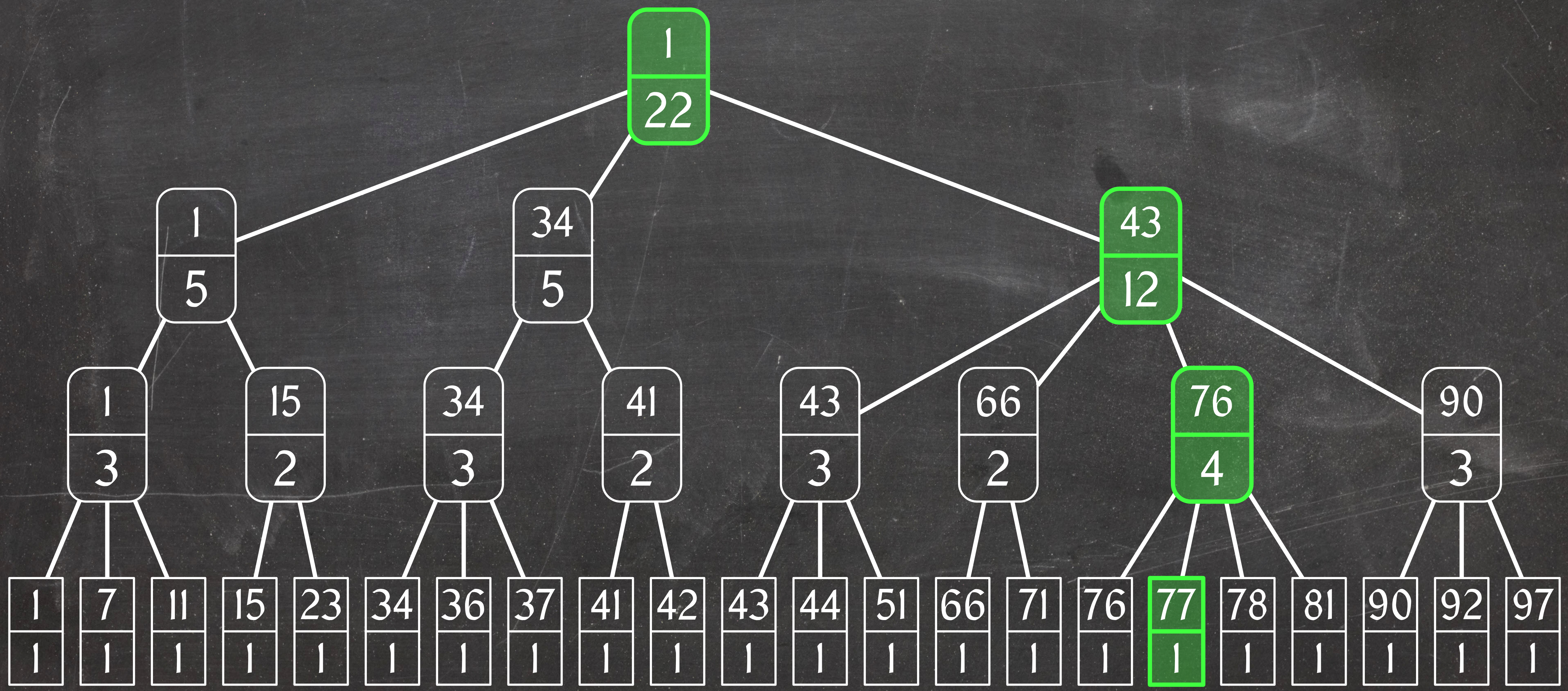
After the insertion of a new leaf v, which leaf counts need to be updated?



Insertions

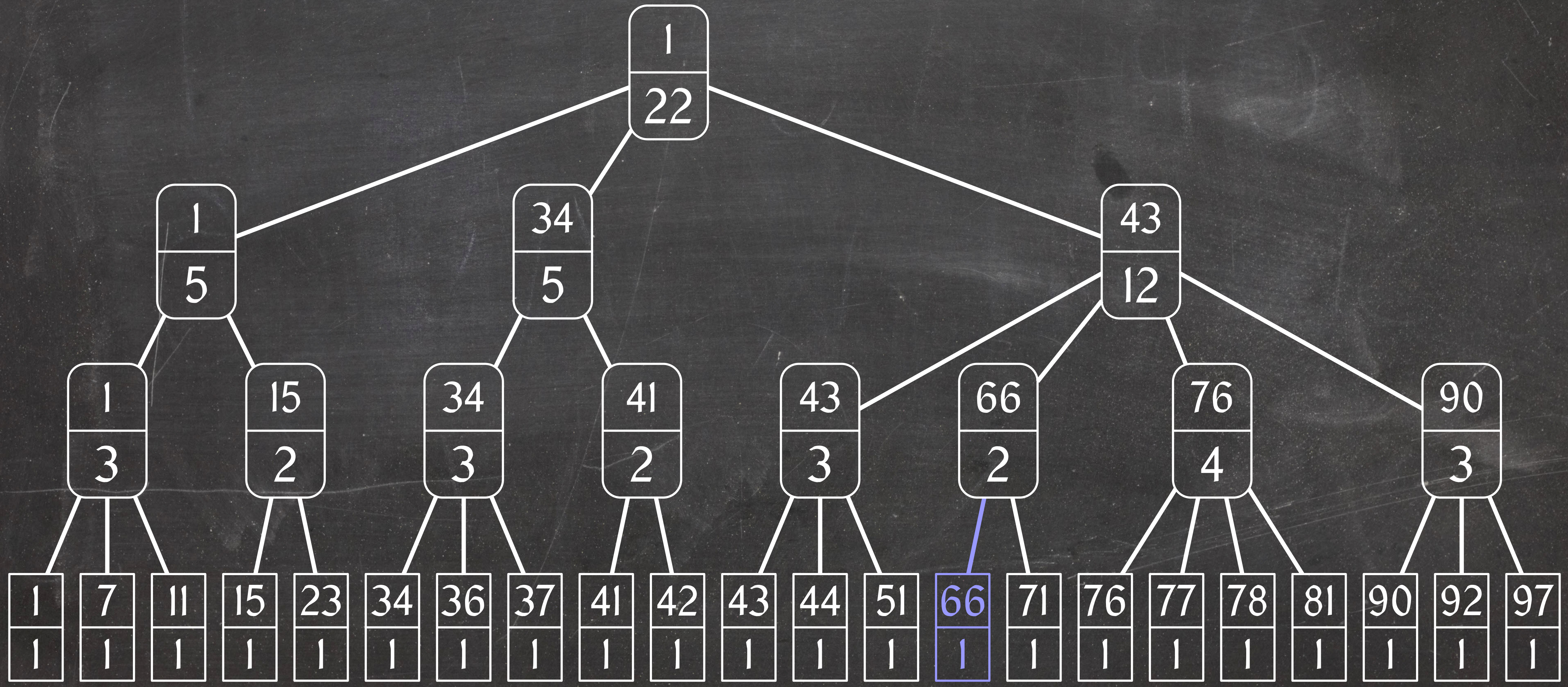
After the insertion of a new leaf v, which leaf counts need to be updated?

Those of of v's ancestors must be increased by one.



Deletions

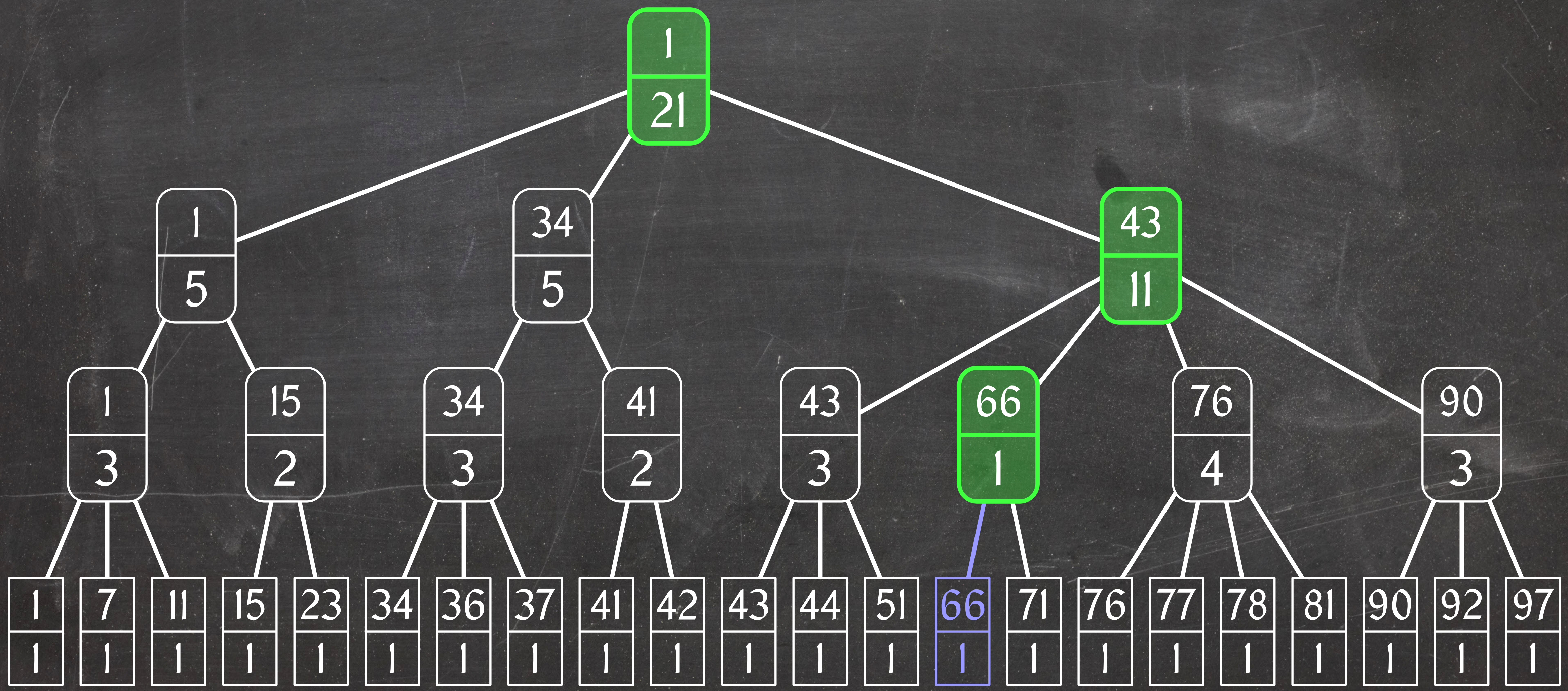
After the deletion of a leaf v, which leaf counts need to be updated?



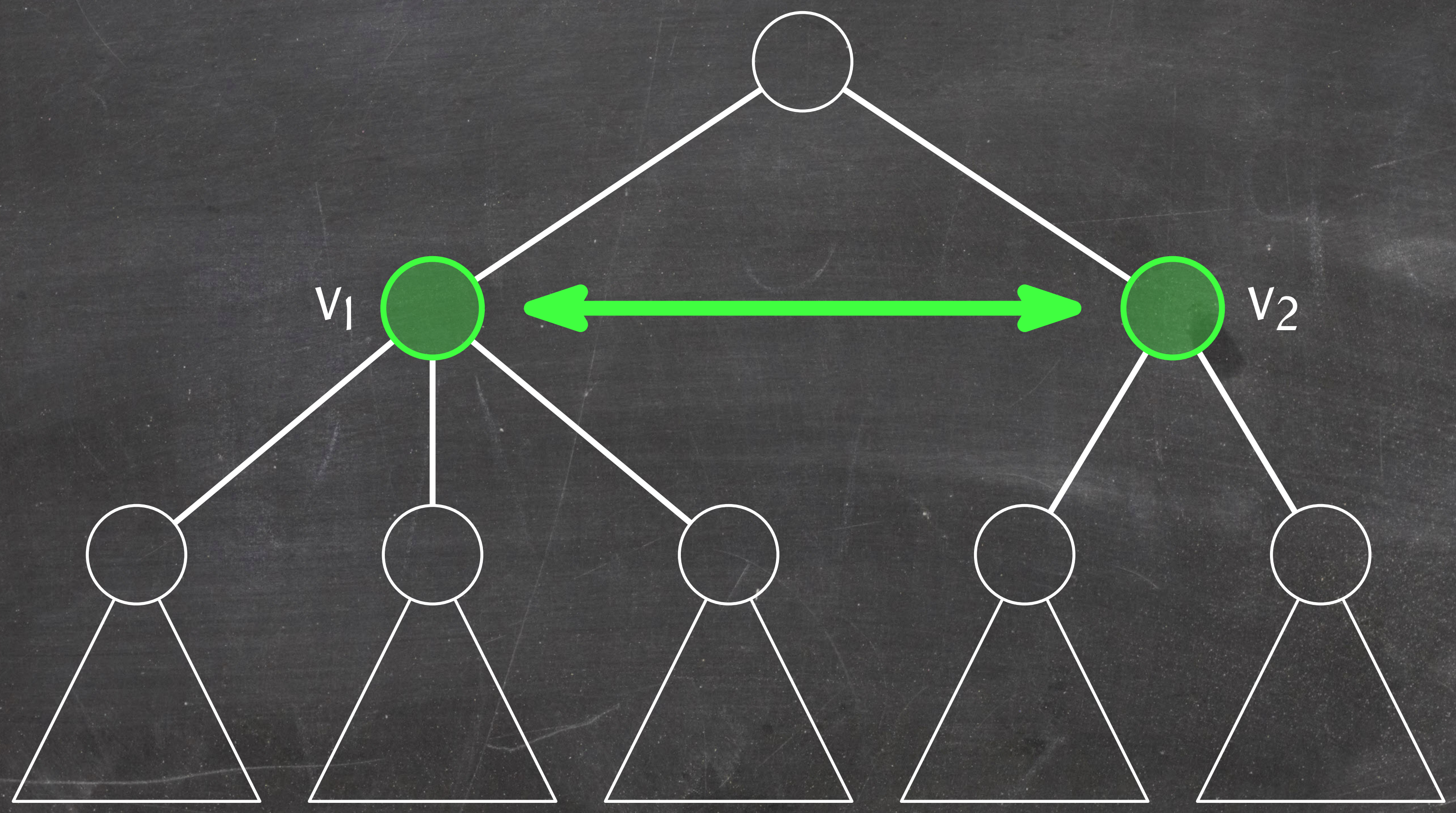
Deletions

After the deletion of a leaf v, which leaf counts need to be updated?

Those of of v's ancestors must be decreased by one.



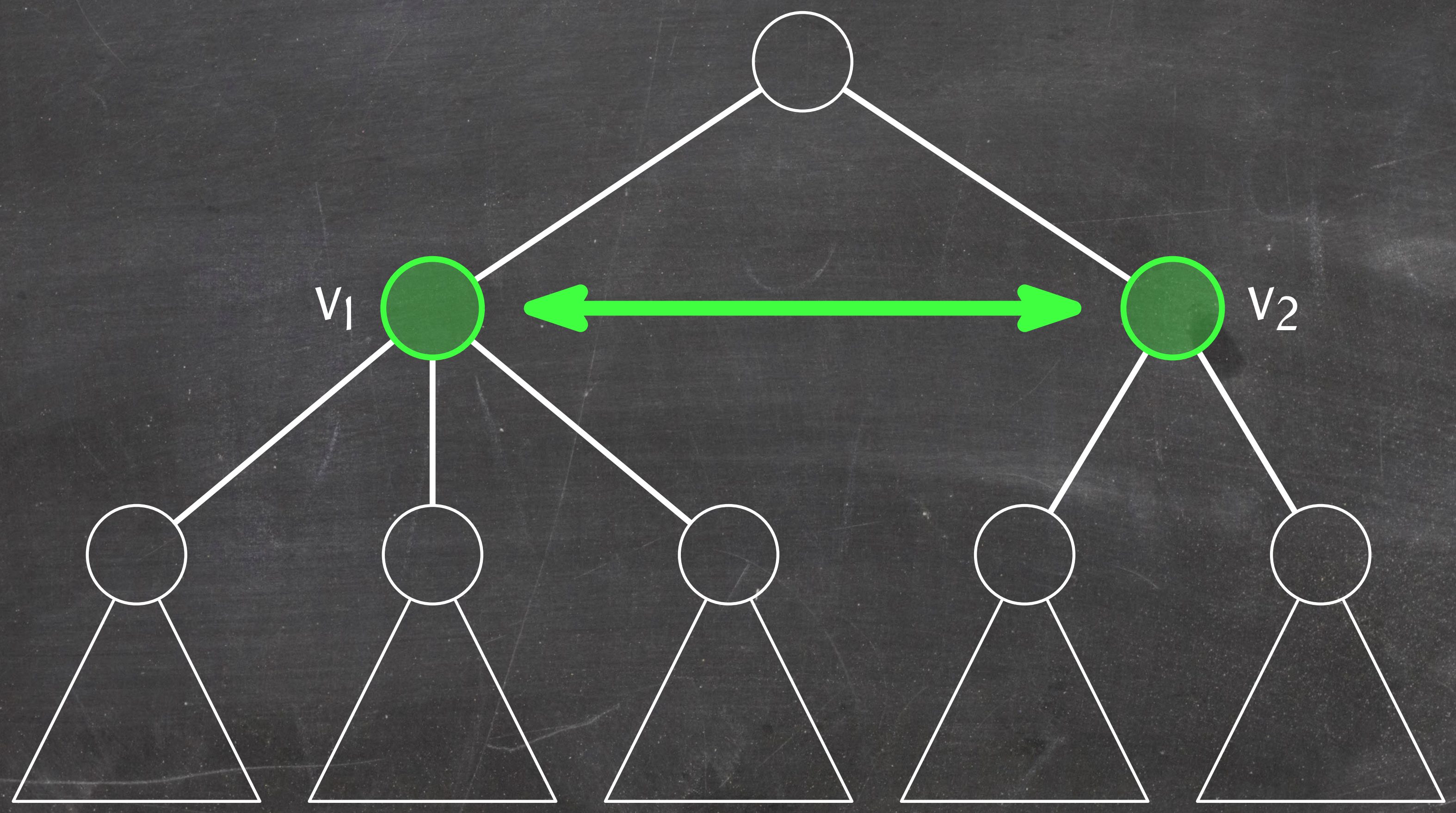
Node Splits



Node Splits

The leaf counts of v_1 and v_2 are the sums of the leaf counts of their children.

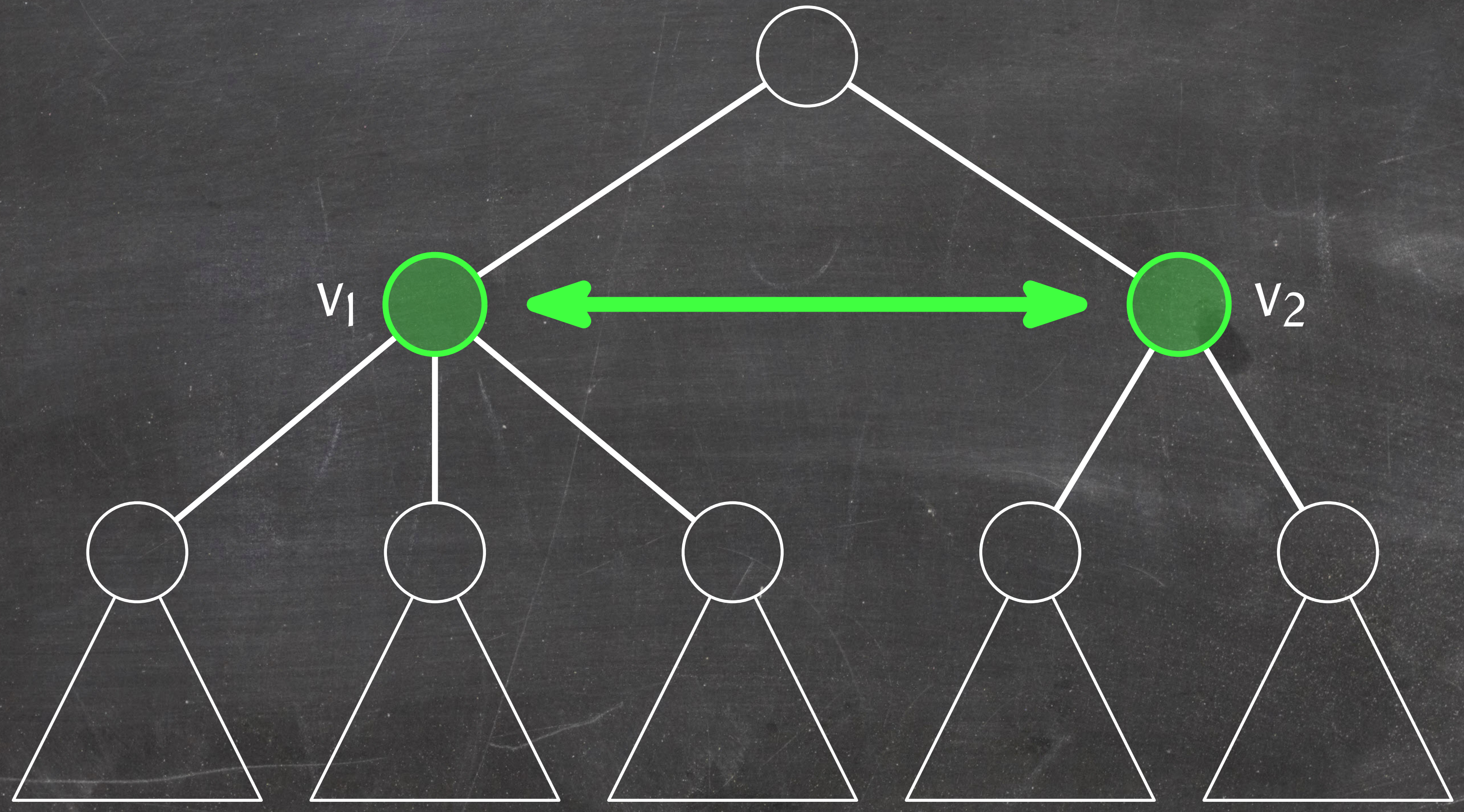
All other leaf counts remain unchanged.



Node Splits

The leaf counts of v_1 and v_2 are the sums of the leaf counts of their children.

All other leaf counts remain unchanged.

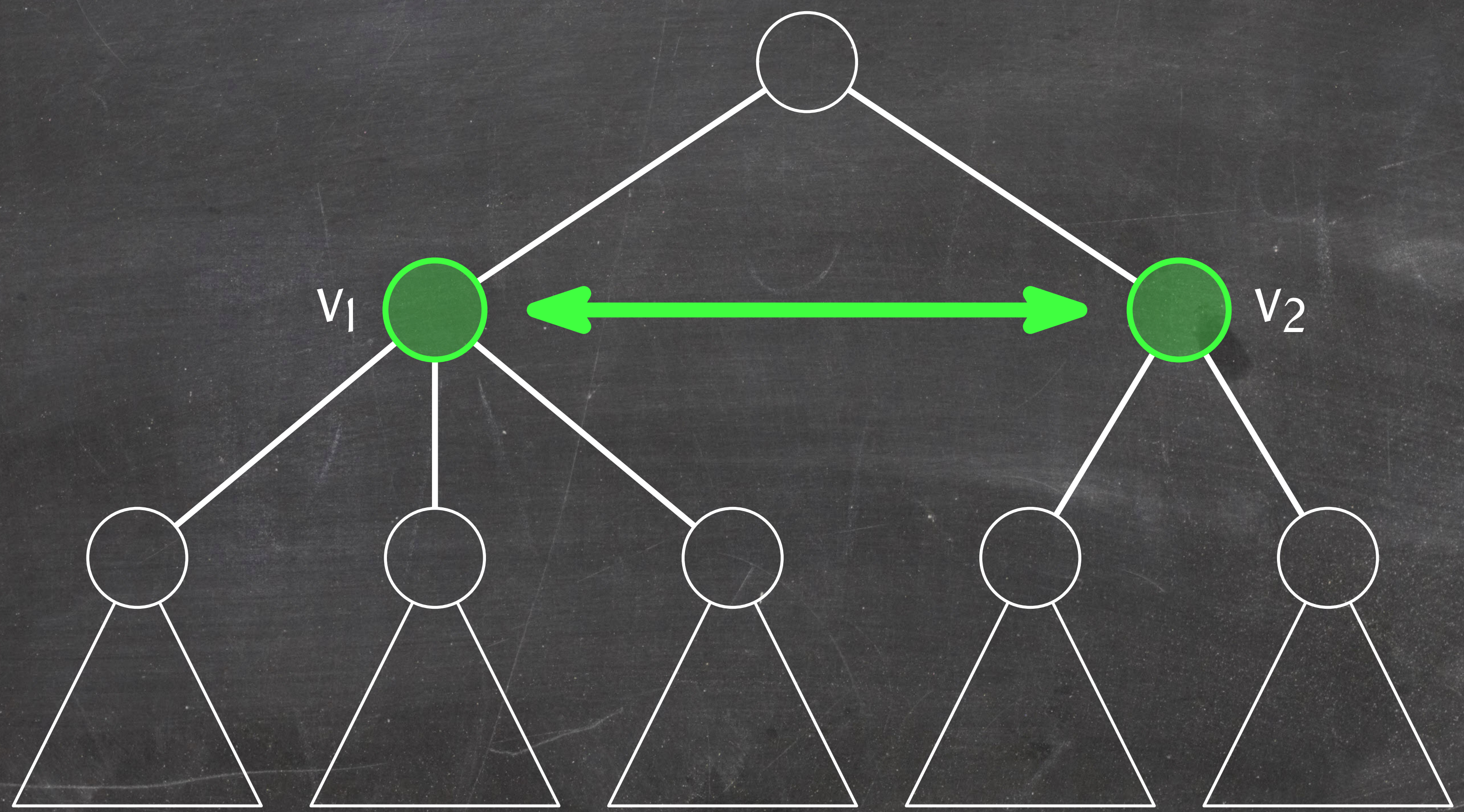


Lemma: A node split takes $O(1)$ time including the time to recompute leaf counts.

Node Splits

The leaf counts of v_1 and v_2 are the sums of the leaf counts of their children.

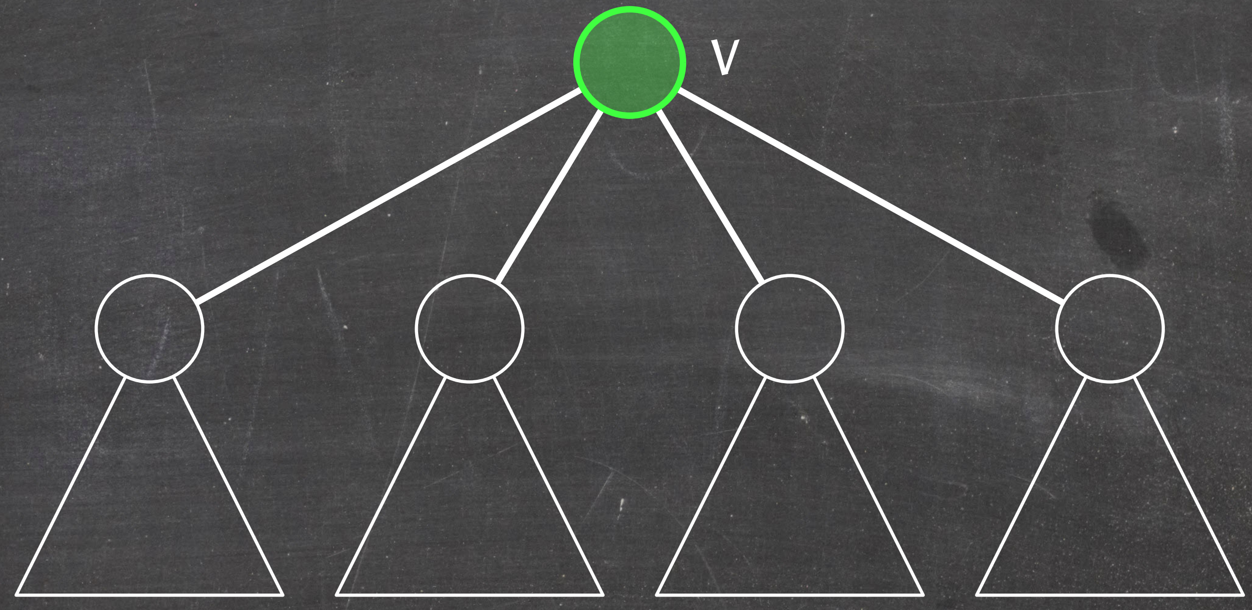
All other leaf counts remain unchanged.



Lemma: A node split takes $O(1)$ time including the time to recompute leaf counts.

Corollary: An insertion into a Rank-Select tree takes $O(\lg n)$ time.

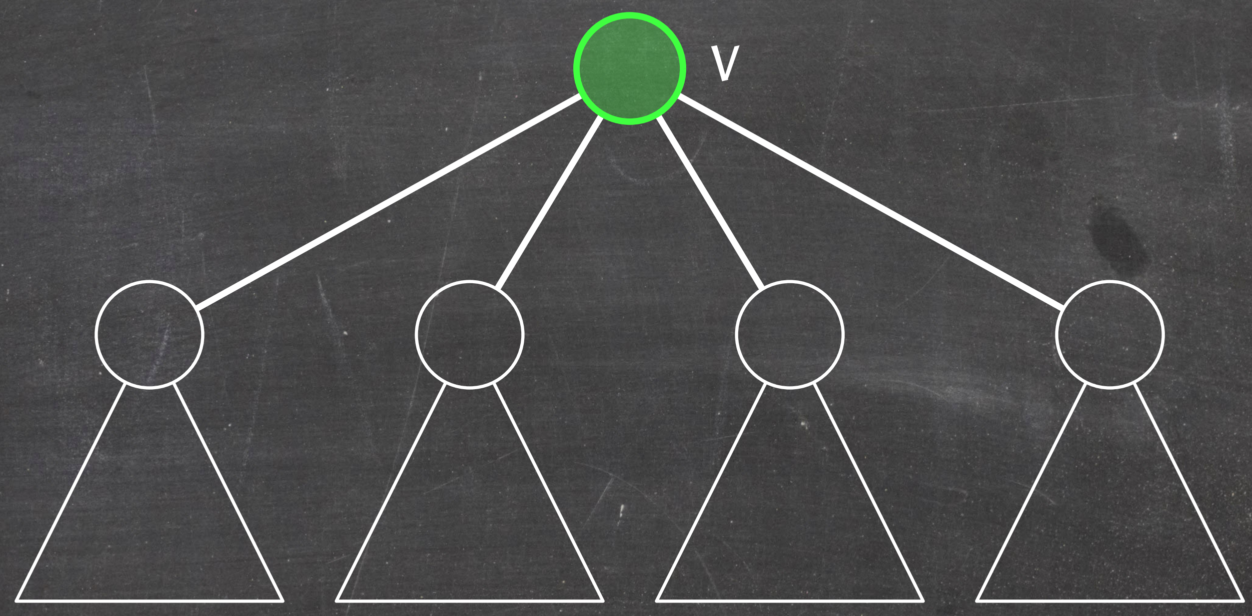
Node Fusions



Node Fusions

The leaf count of the fused node v is the sum of the leaf counts of its children.

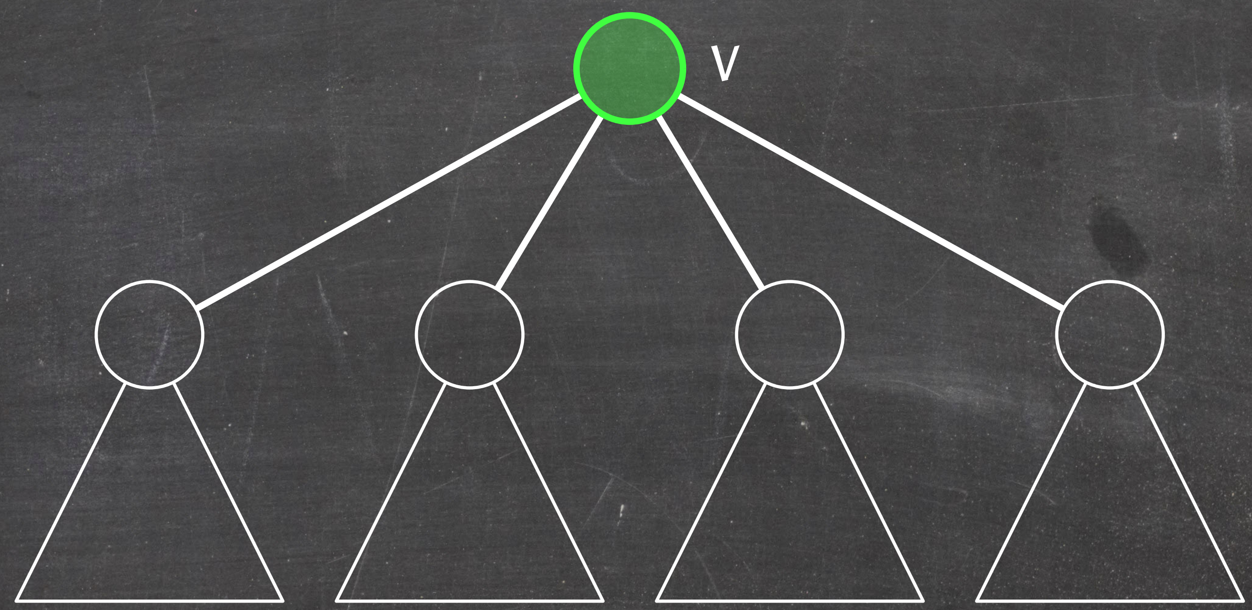
All other leaf counts remain unchanged.



Node Fusions

The leaf count of the fused node v is the sum of the leaf counts of its children.

All other leaf counts remain unchanged.

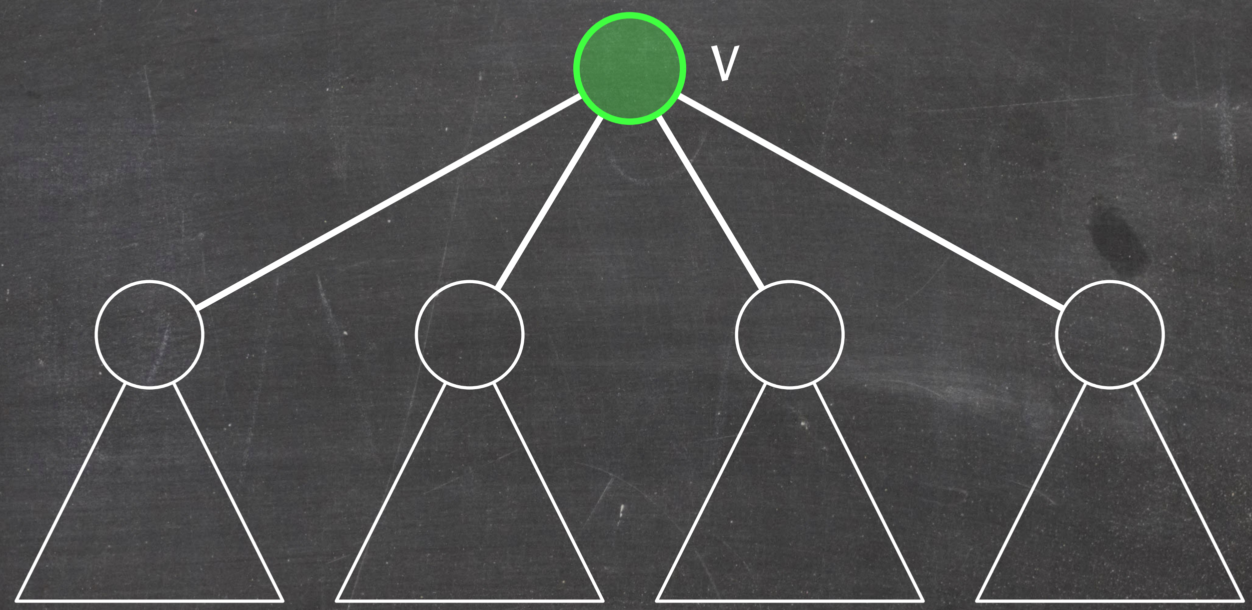


Lemma: A node fusion takes O(1) time including the time to recompute leaf counts.

Node Fusions

The leaf count of the fused node v is the sum of the leaf counts of its children.

All other leaf counts remain unchanged.



Lemma: A node fusion takes $O(1)$ time including the time to recompute leaf counts.

Corollary: A deletion from a Rank-Select tree takes $O(\lg n)$ time.

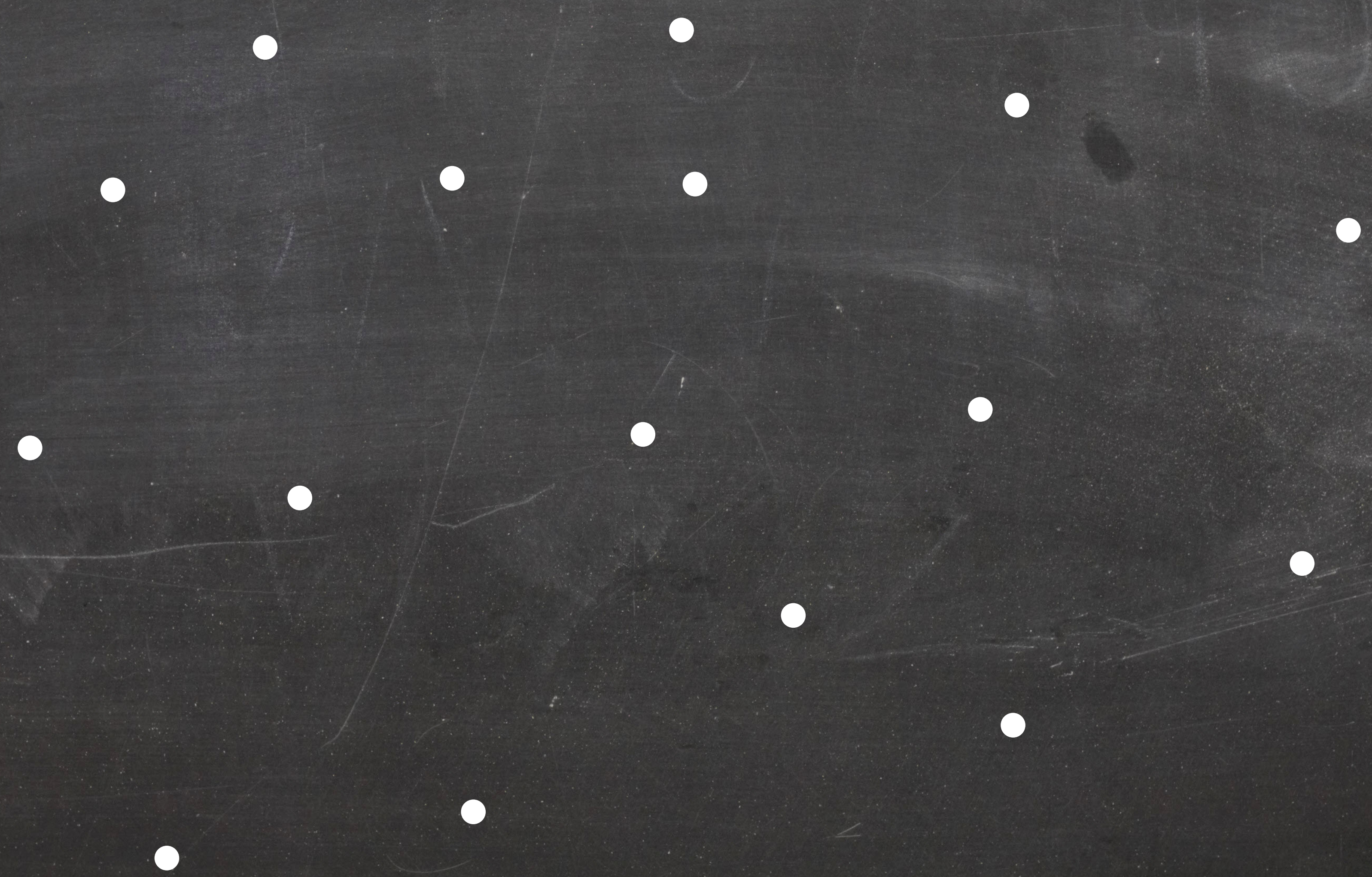
Rank-Select Tree: Summary

Theorem: A Rank-Select tree supports Insert, Delete, Rank, and Select operations in $O(\lg n)$ time.

Three-Sided Range Reporting

Problem: Maintain a set S of points in the plane under insertions and deletions and support three-sided range reporting queries:

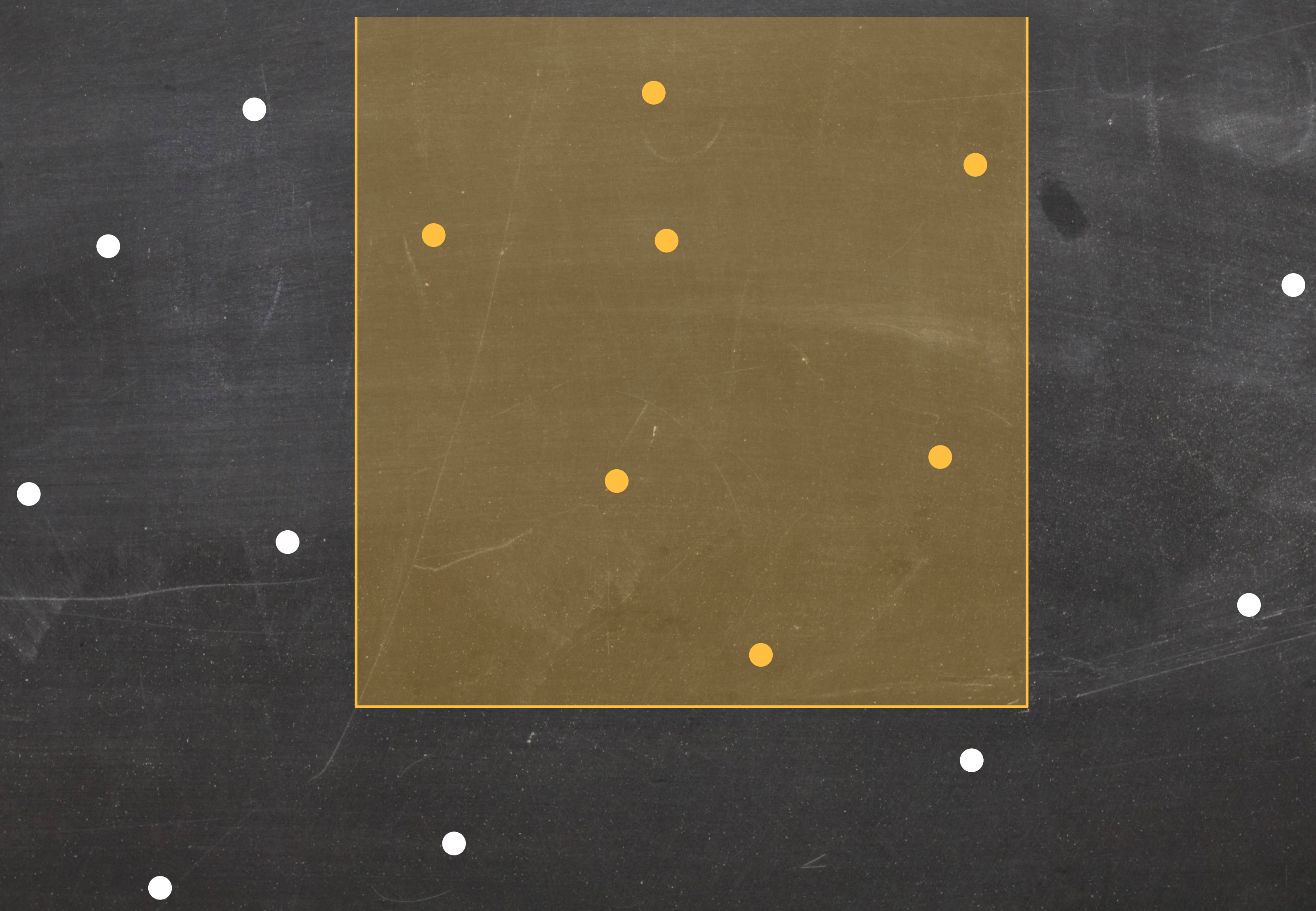
Given a query range $R = [\ell, r] \times [b, \infty)$, report all points in S that belong to R.



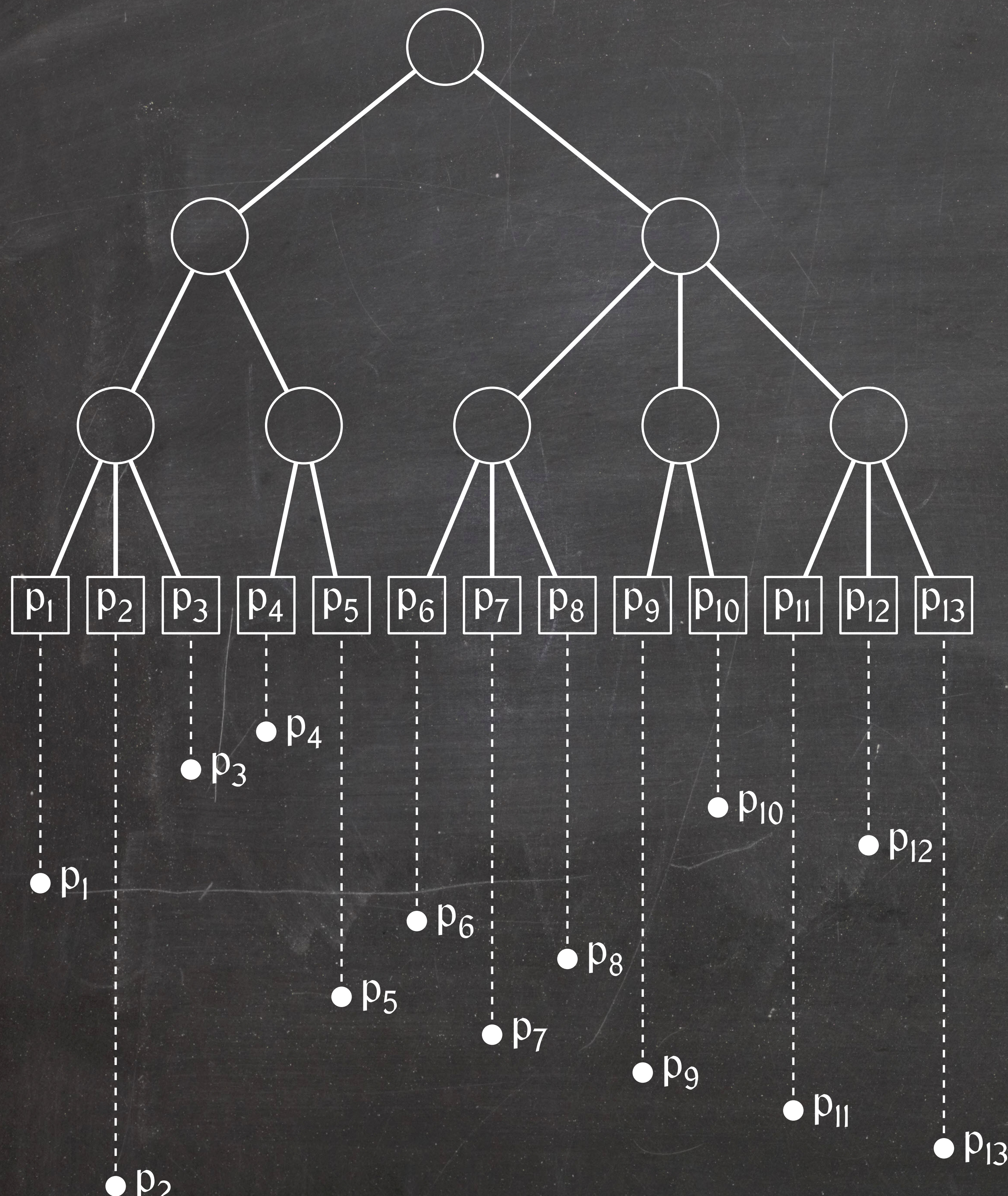
Three-Sided Range Reporting

Problem: Maintain a set S of points in the plane under insertions and deletions and support three-sided range reporting queries:

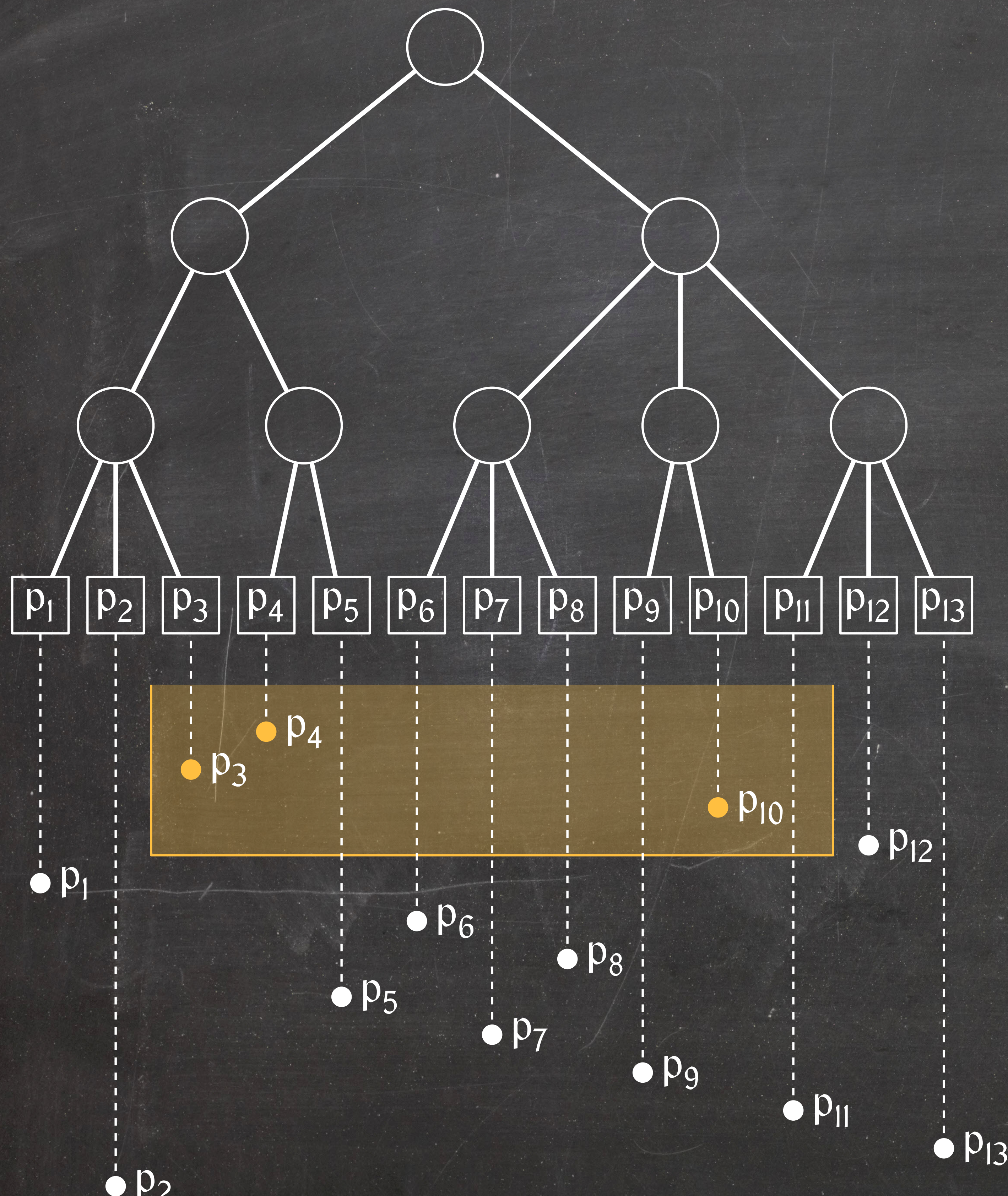
Given a query range $R = [\ell, r] \times [b, \infty)$, report all points in S that belong to R.



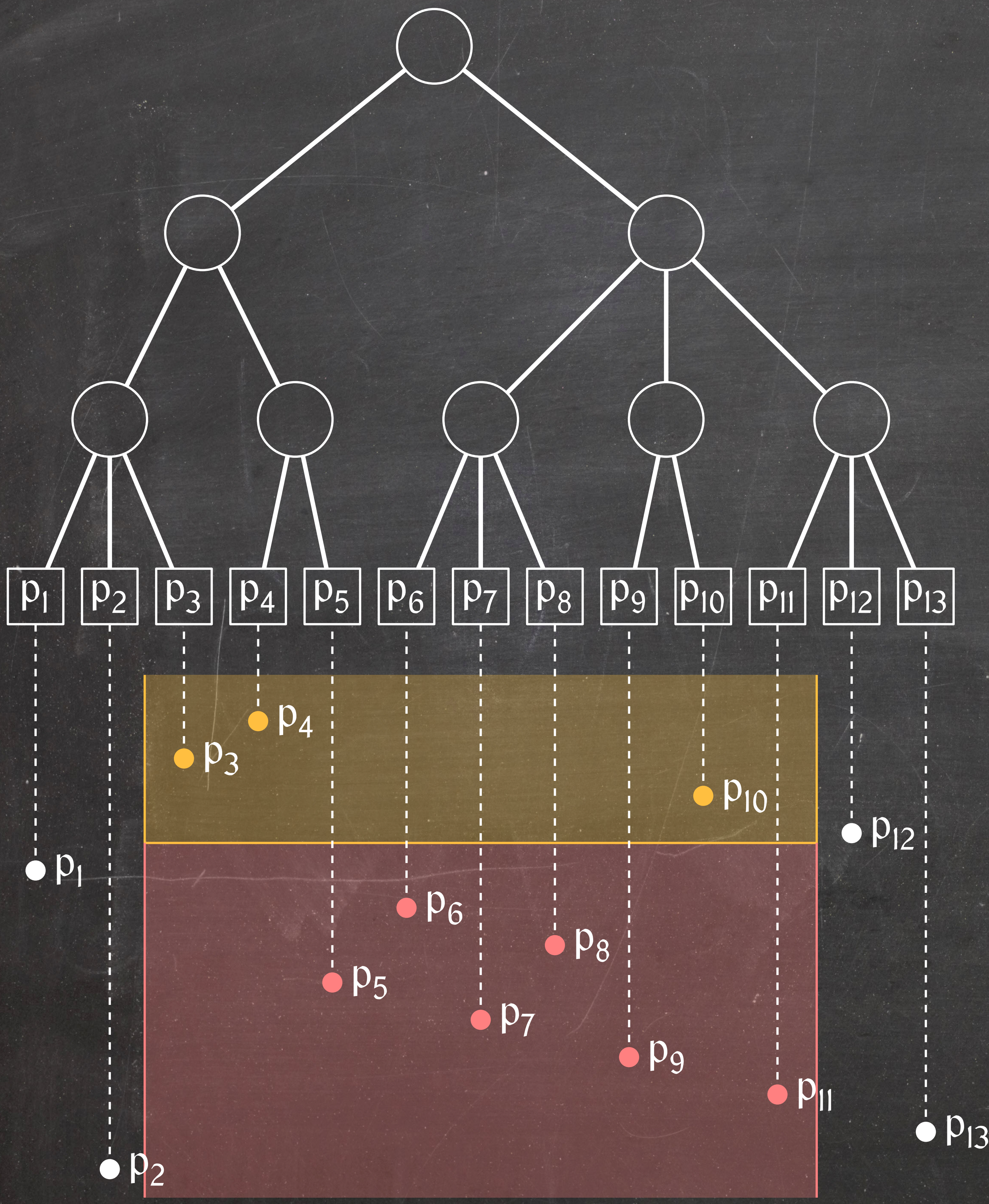
Three-Sided Range Reporting and (a, b)-Trees



Three-Sided Range Reporting and (a, b)-Trees

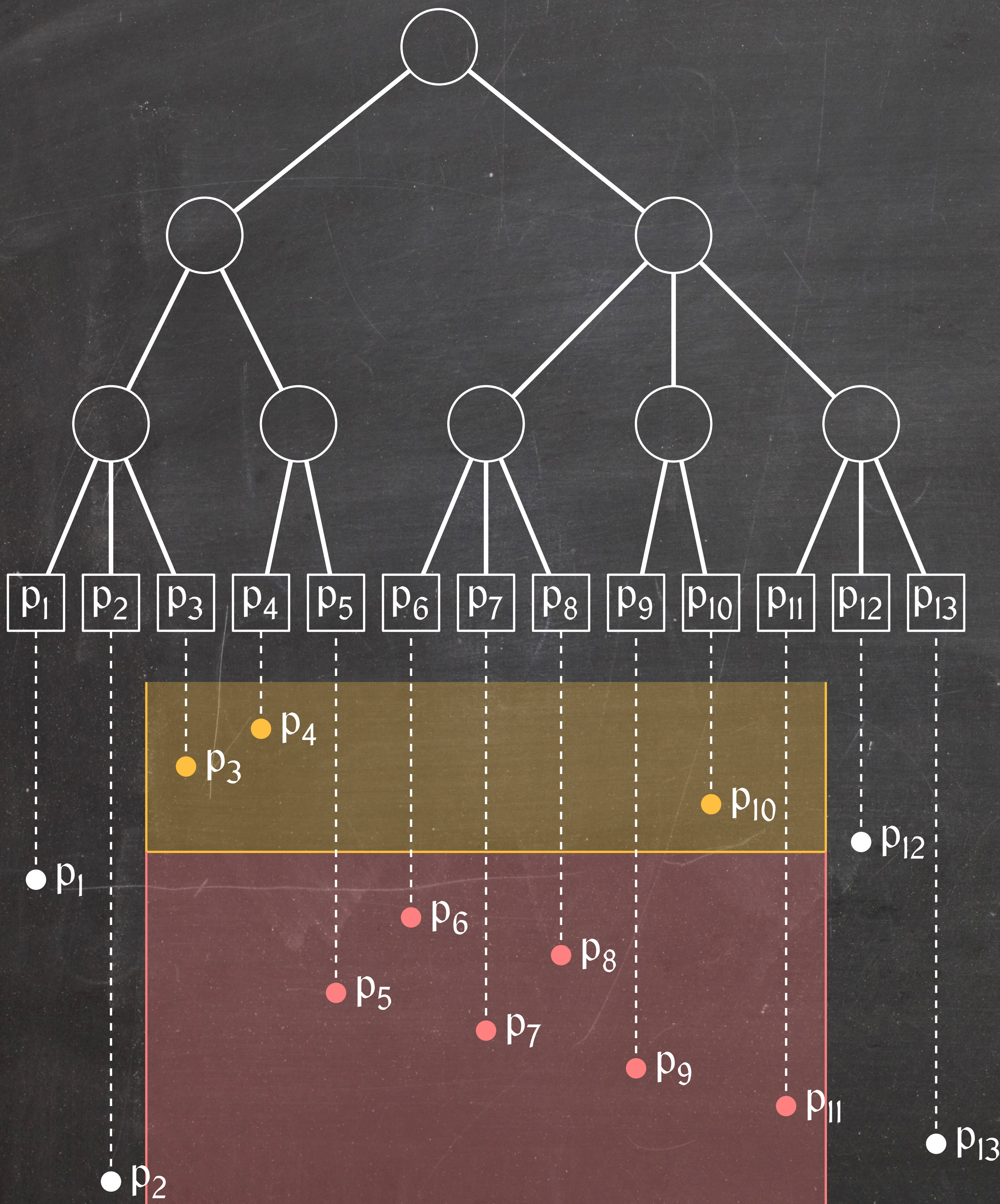


Three-Sided Range Reporting and (a, b)-Trees



A RangeFind operation allows us to find all the points in the x-range of the query in $O(\lg n + k)$ time.

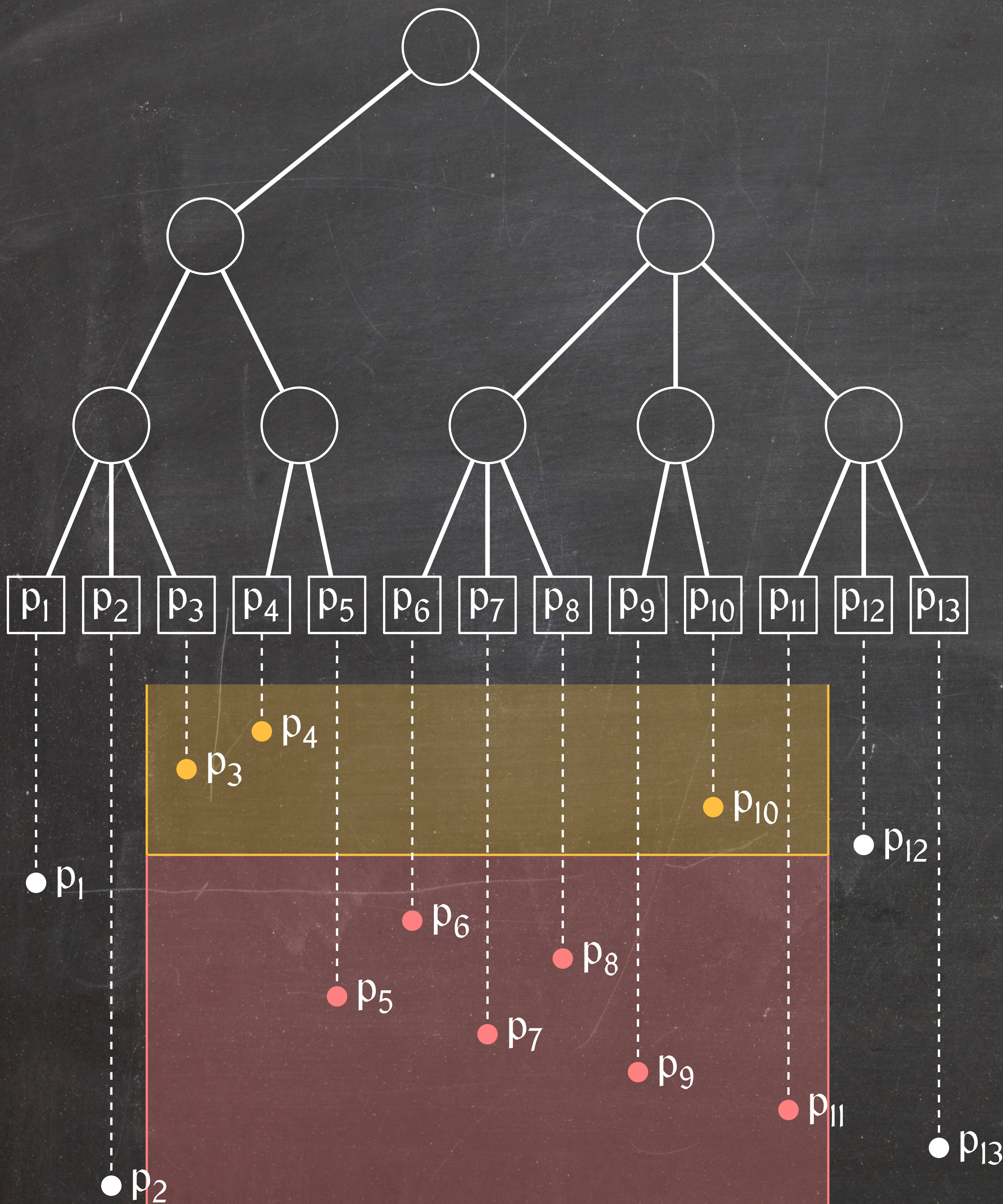
Three-Sided Range Reporting and (a, b)-Trees



A RangeFind operation allows us to find all the points in the x-range of the query in $O(\lg n + k)$ time.

Alas, only few of them may be part of the final output.

Three-Sided Range Reporting and (a, b)-Trees

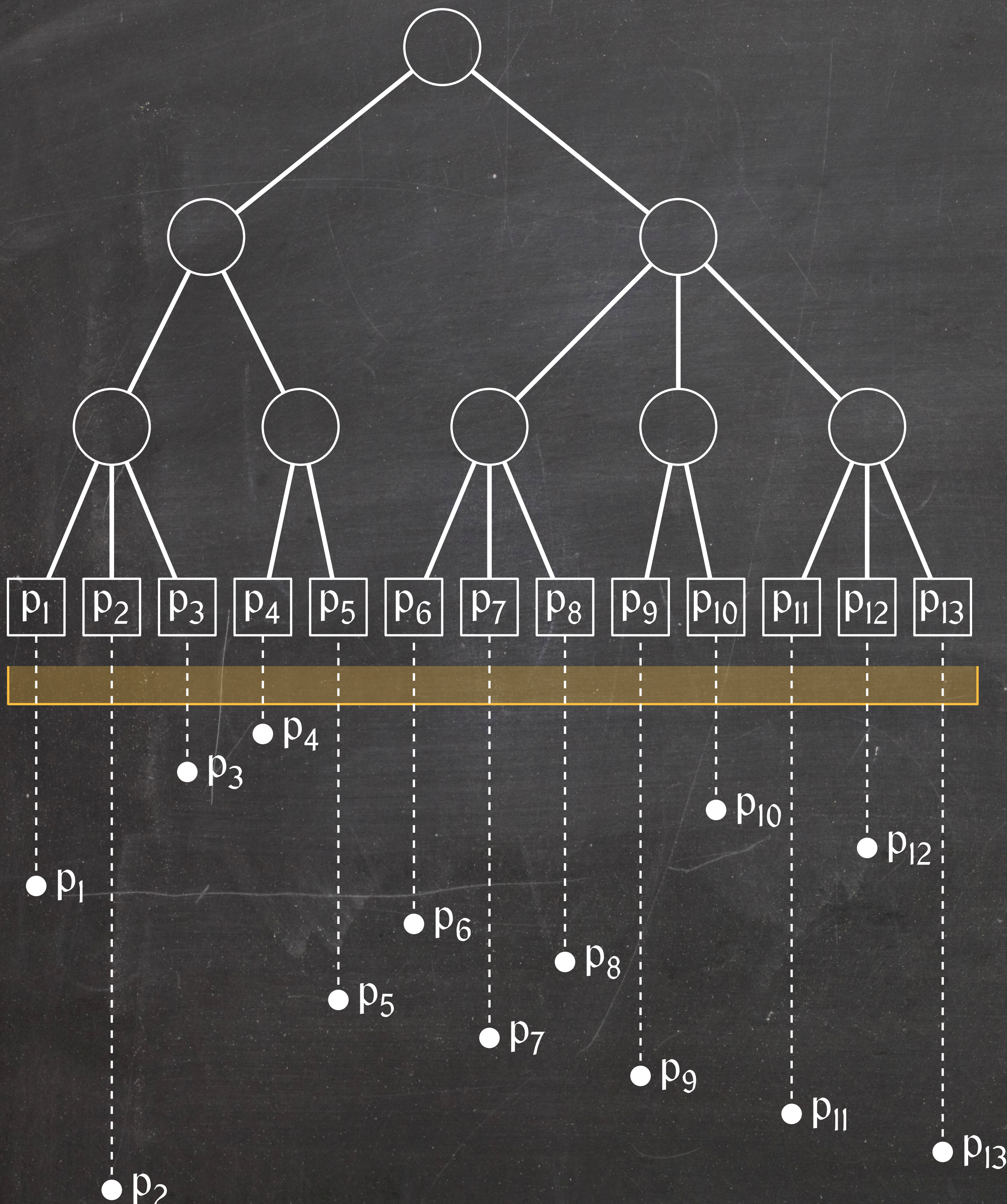


A RangeFind operation allows us to find all the points in the x-range of the query in $O(\lg n + k)$ time.

This is a different k!

Alas, only few of them may be part of the final output.

Three-Sided Range Reporting and (a, b)-Trees

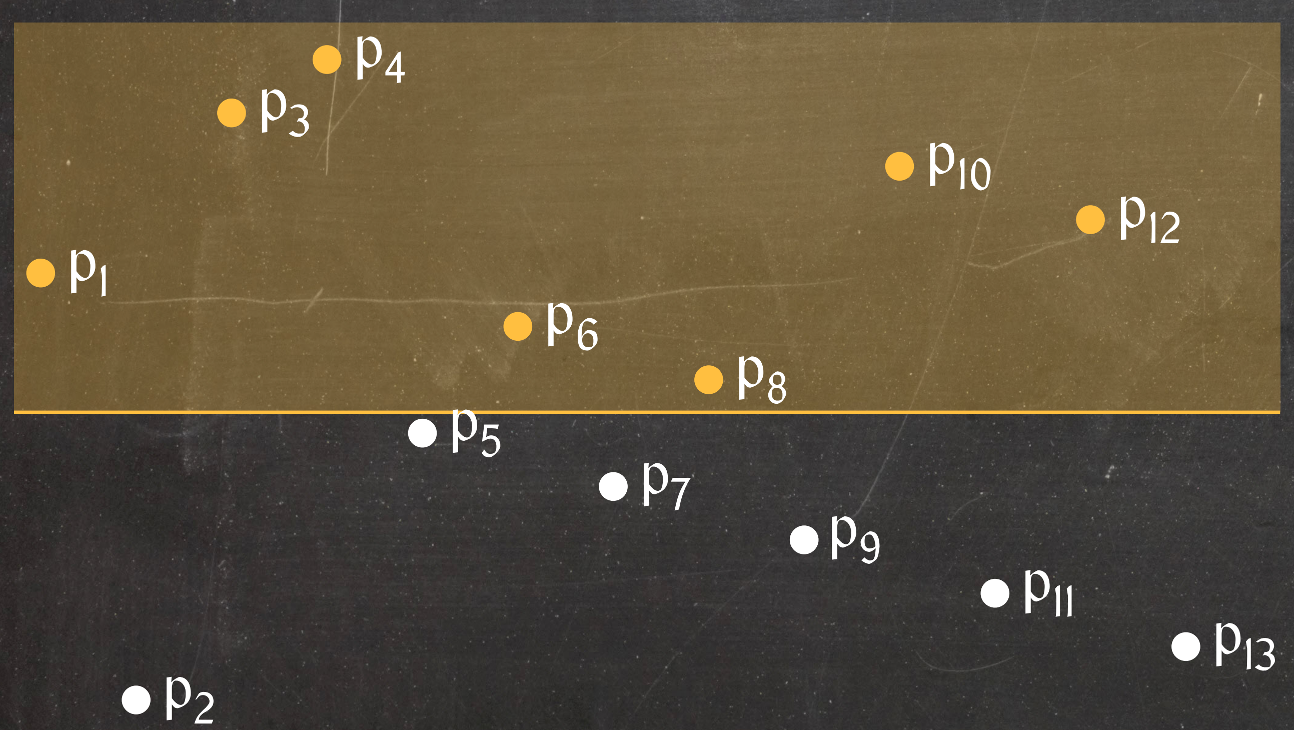


A RangeFind operation allows us to find all the points in the x-range of the query in $O(\lg n + k)$ time.

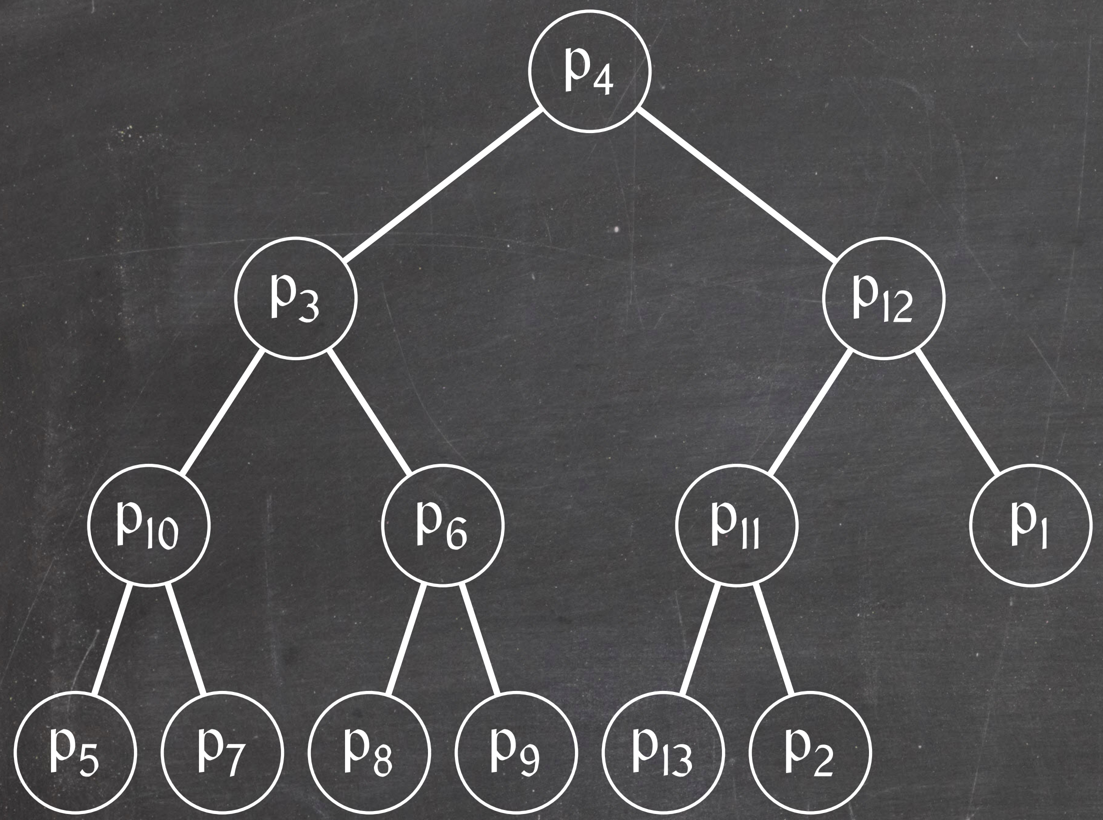
This is a different k!

Alas, only few of them may be part of the final output.

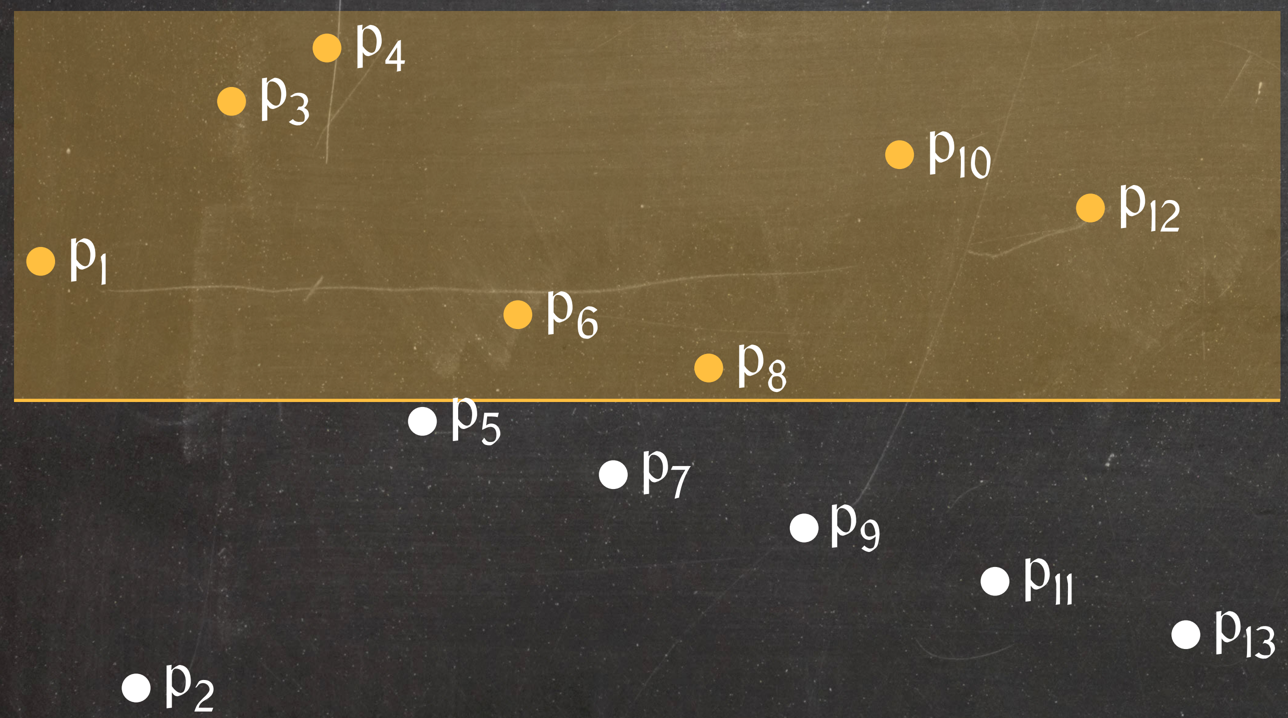
Heap Ordering and Searching With a Lower Bound



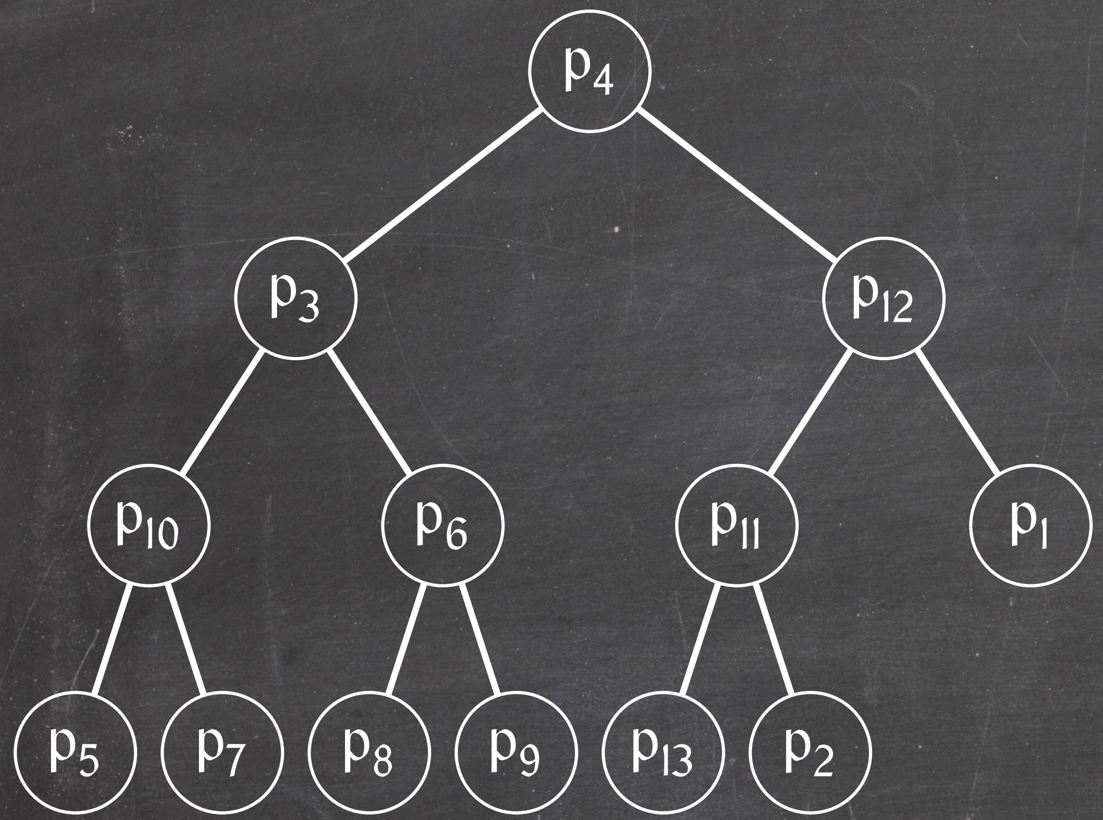
Heap Ordering and Searching With a Lower Bound



If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

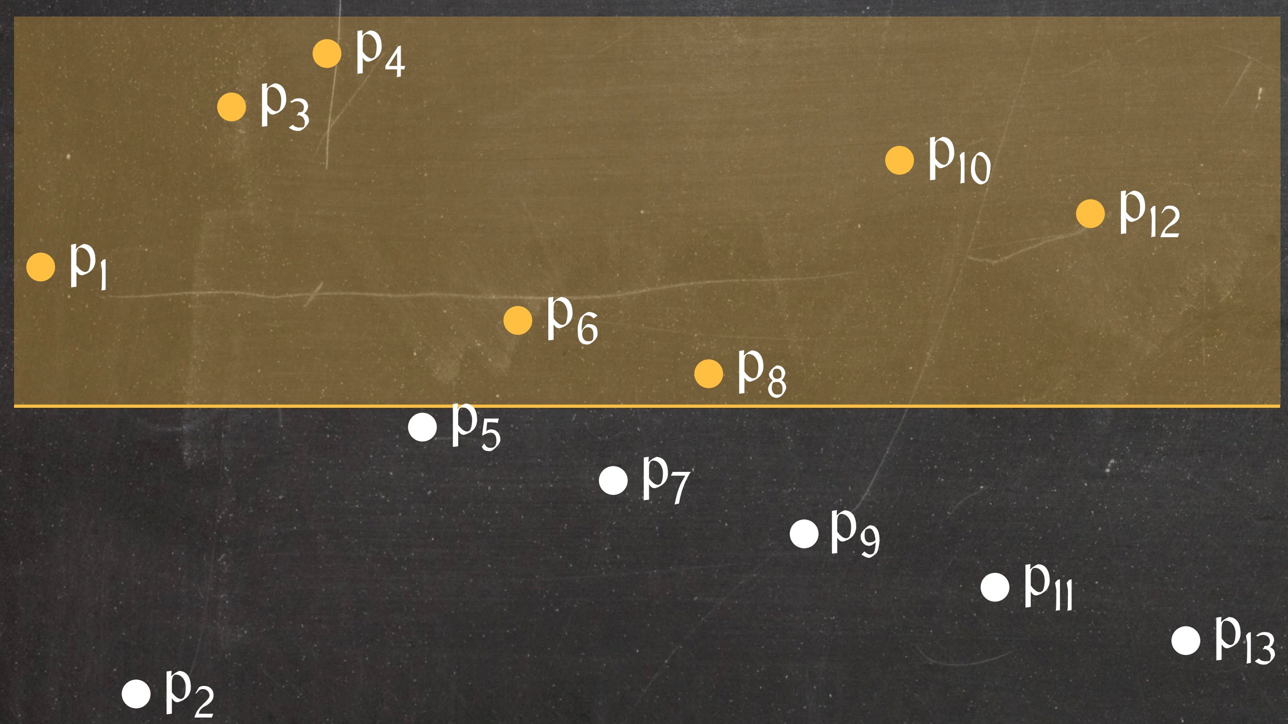


Heap Ordering and Searching With a Lower Bound

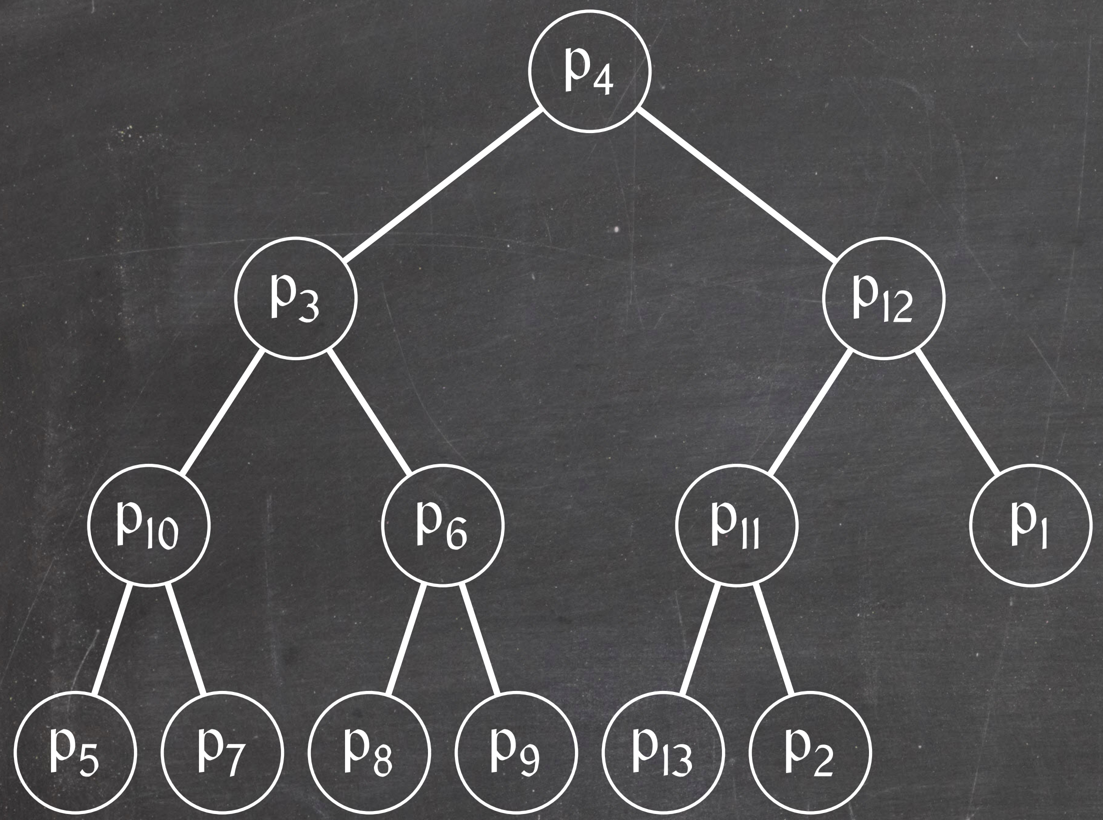


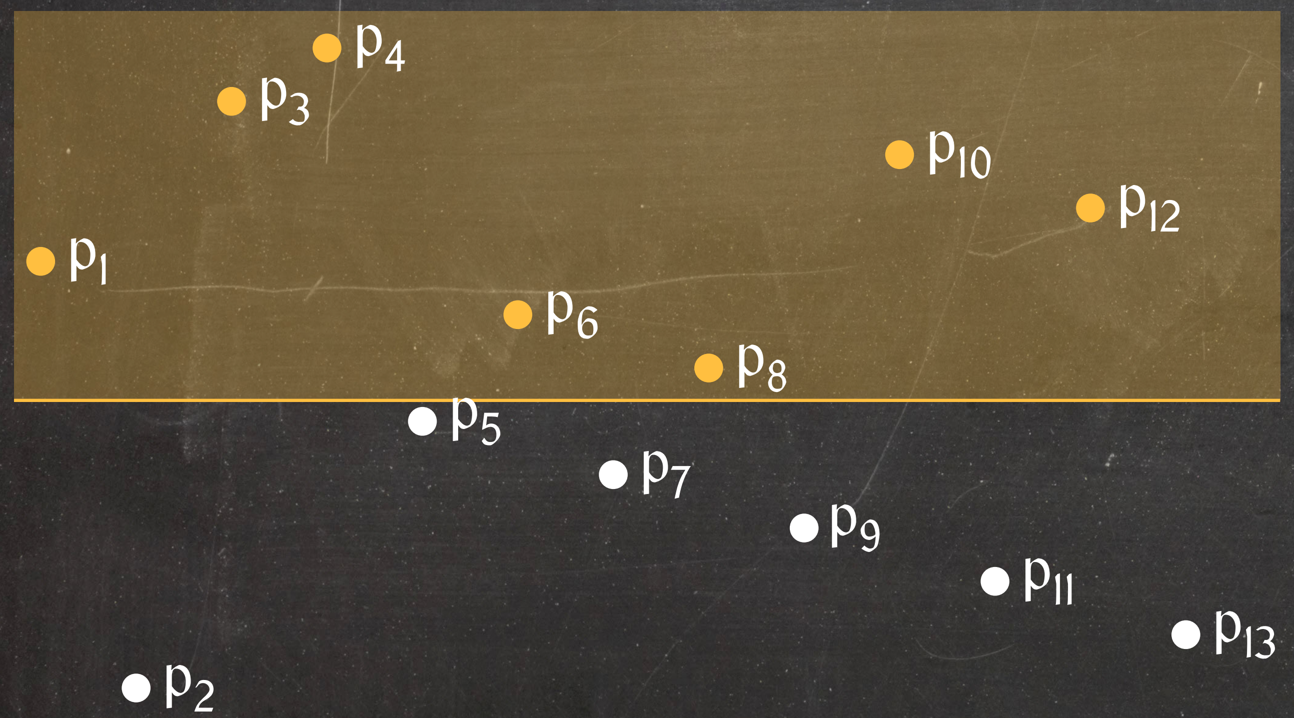
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.



Heap Ordering and Searching With a Lower Bound



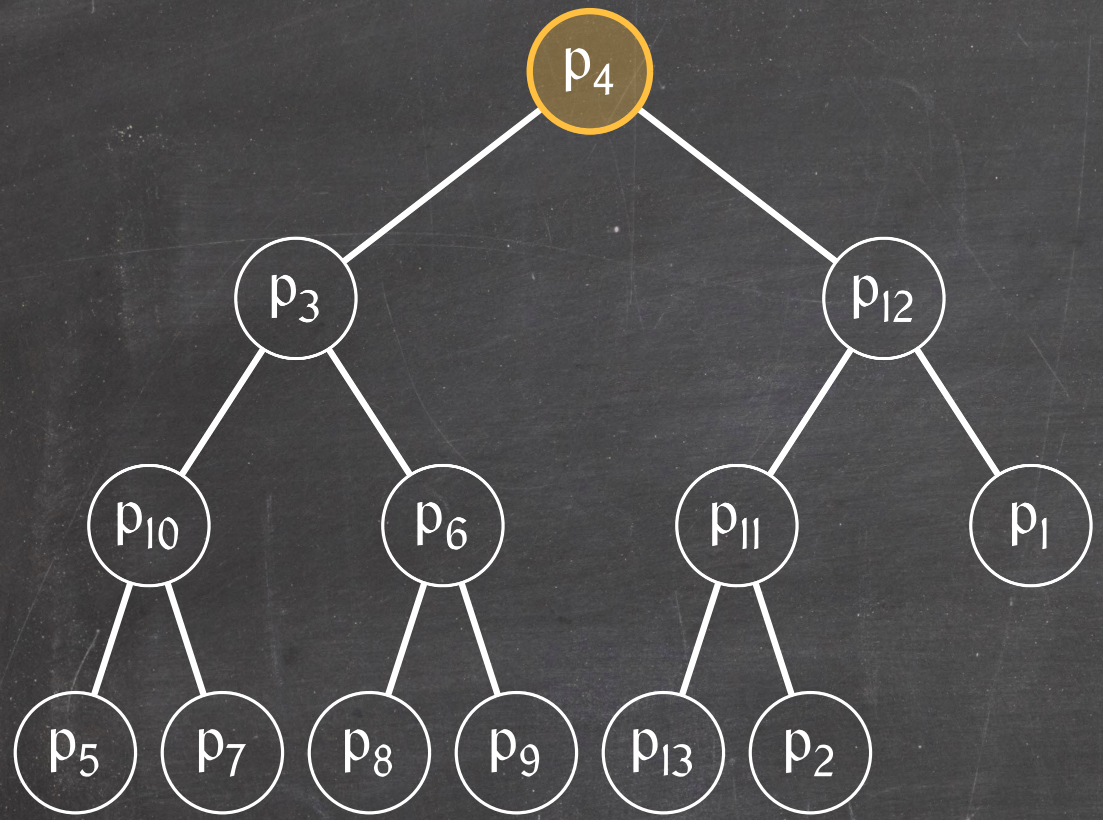


If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

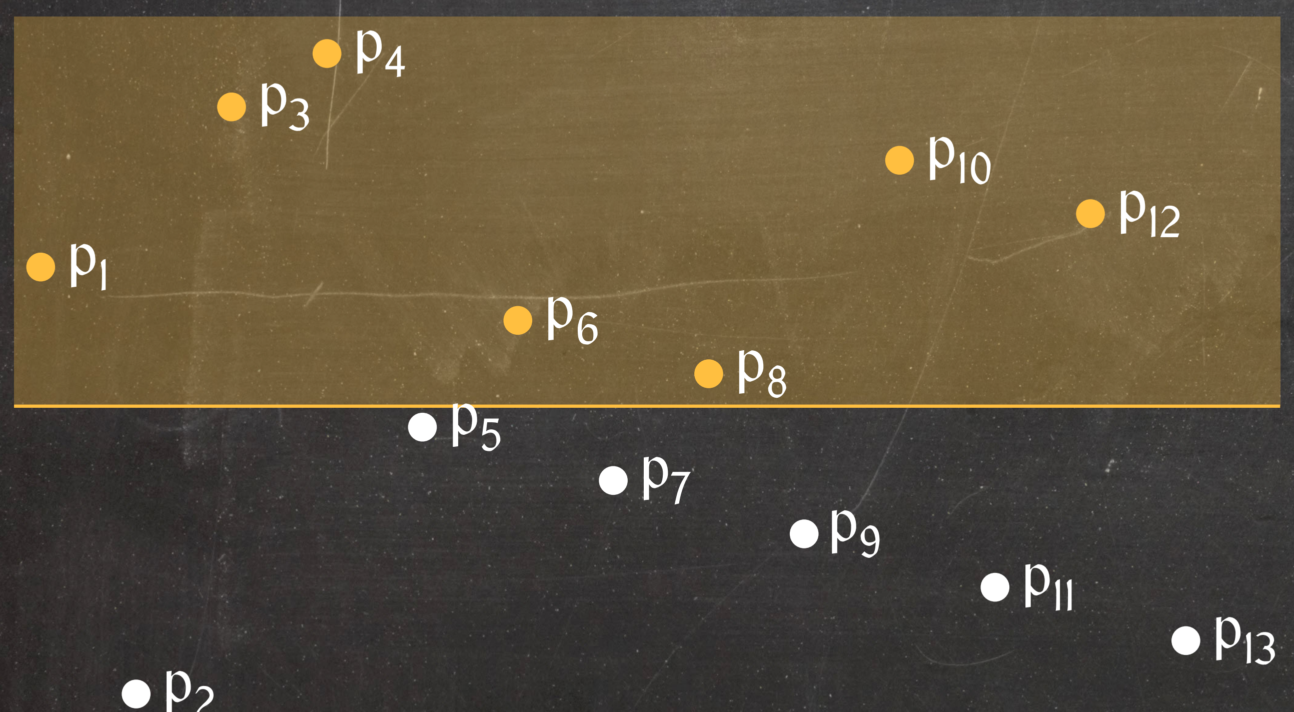
Heap Ordering and Searching With a Lower Bound



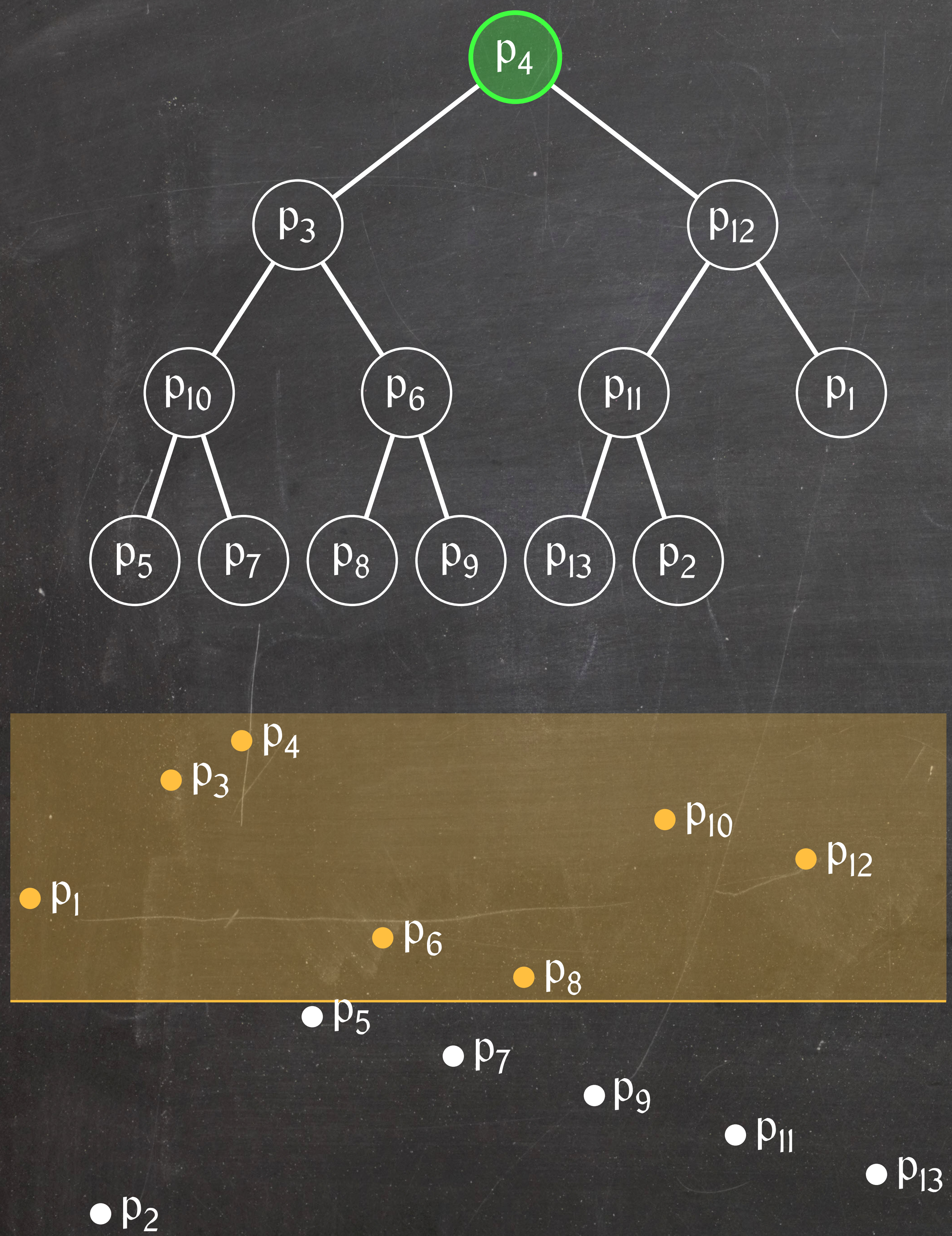
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.



Heap Ordering and Searching With a Lower Bound

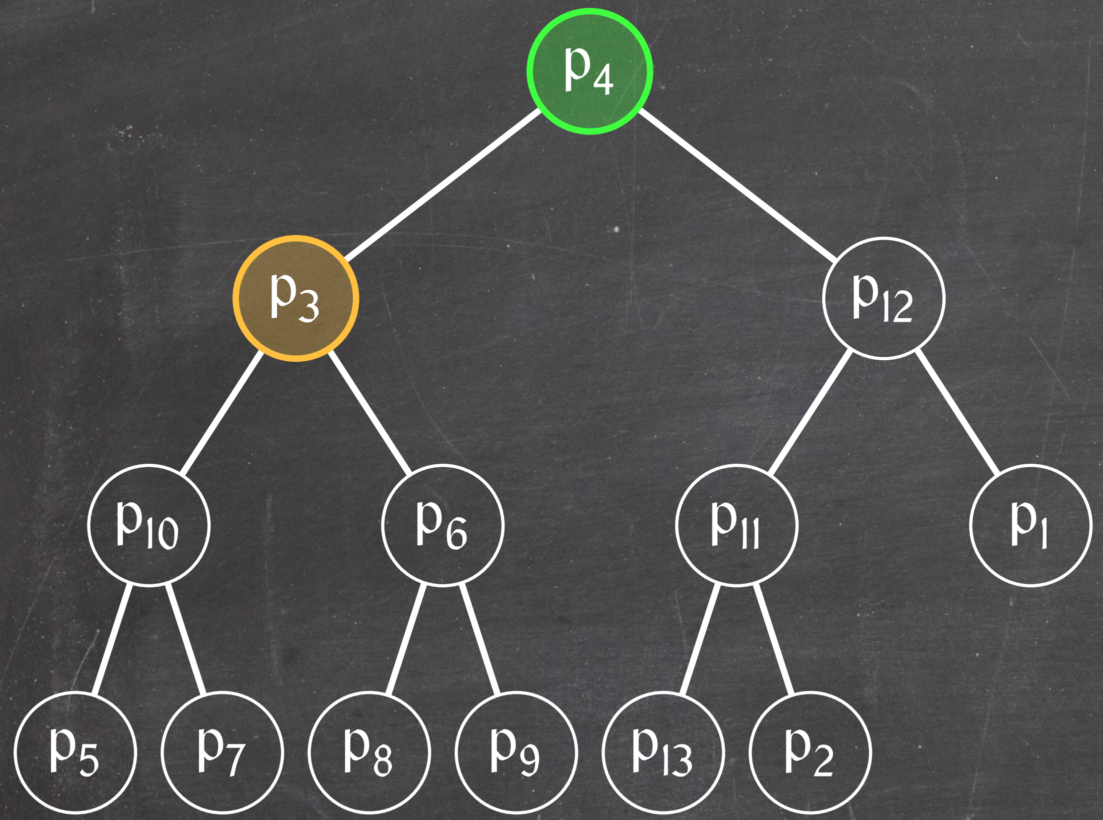


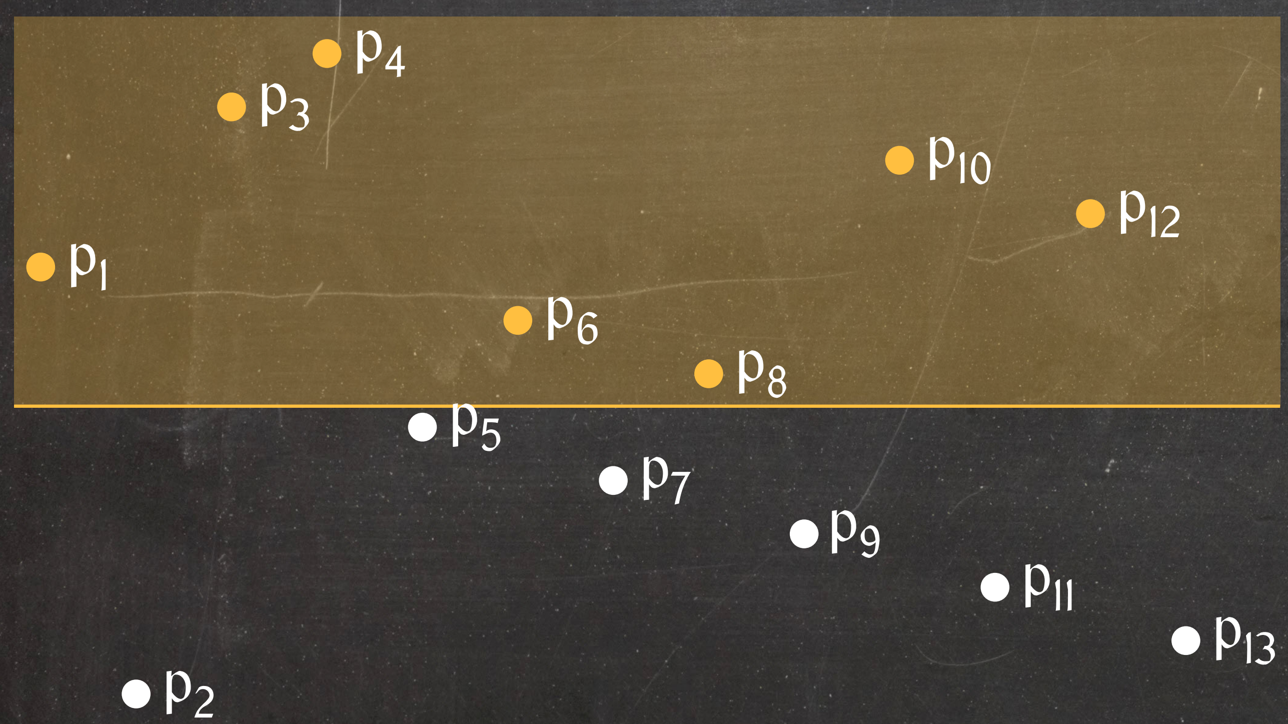
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound



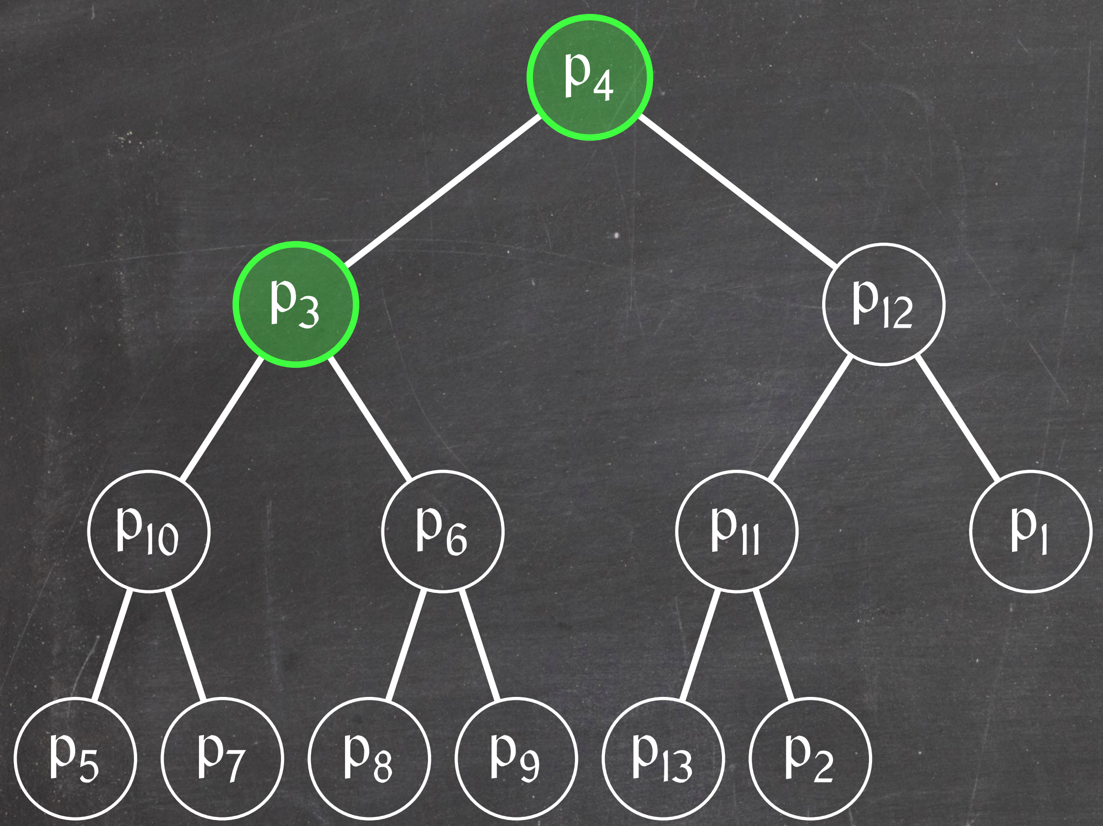


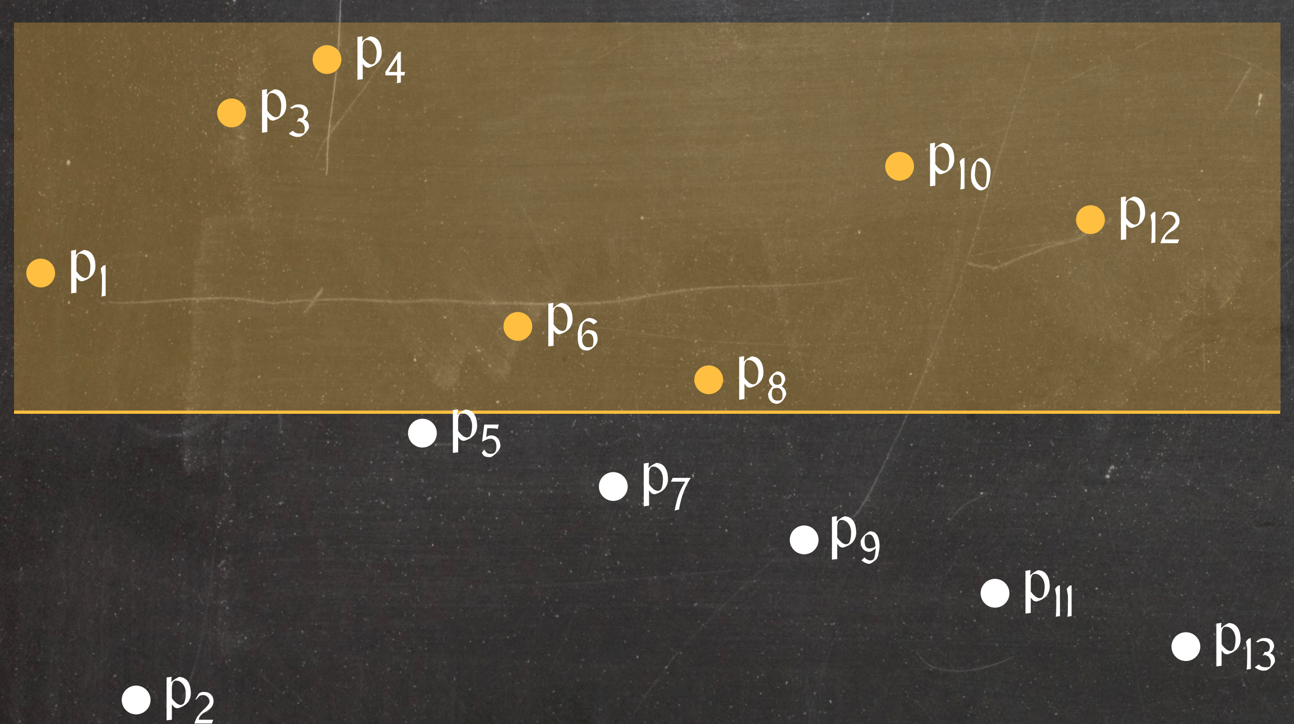
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound



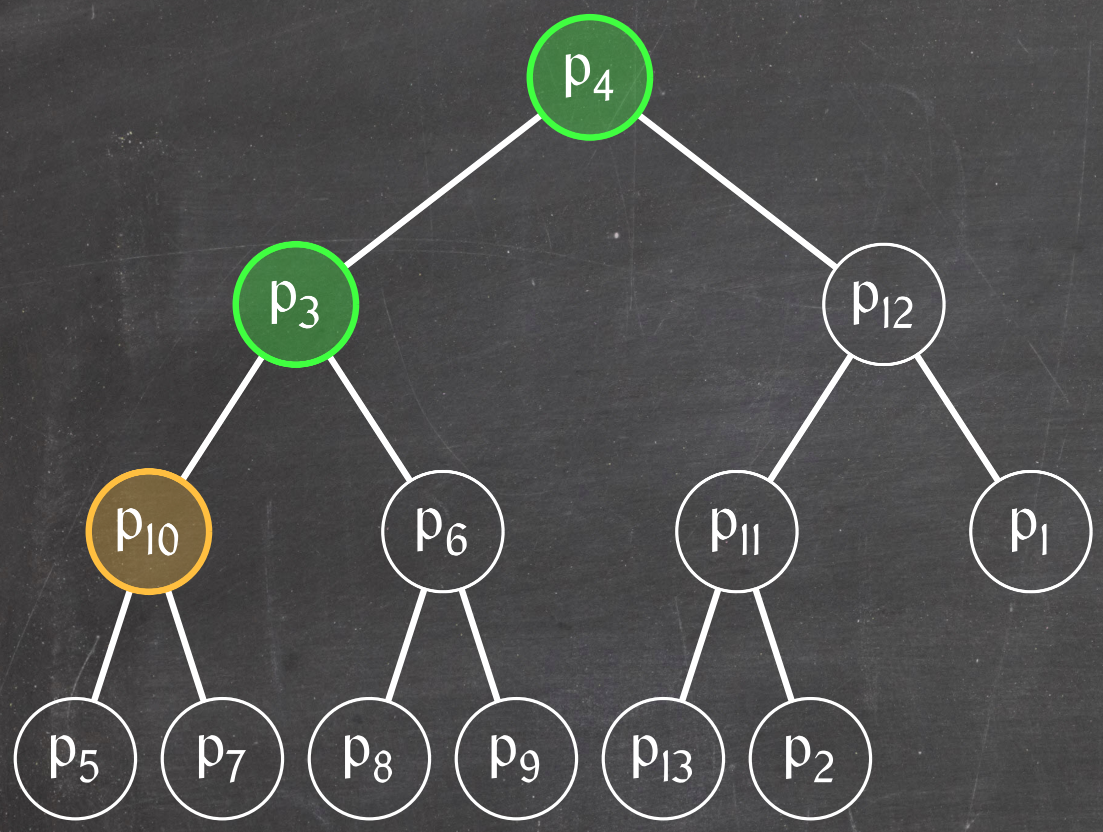


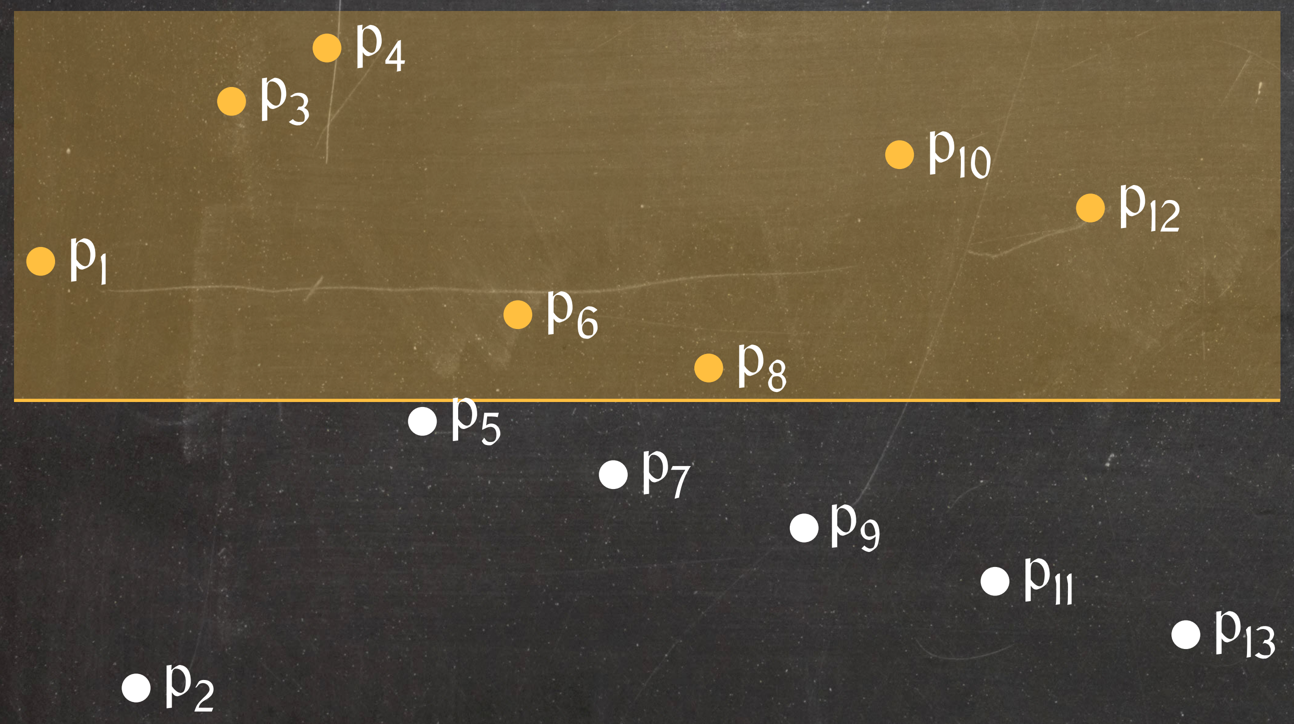
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound



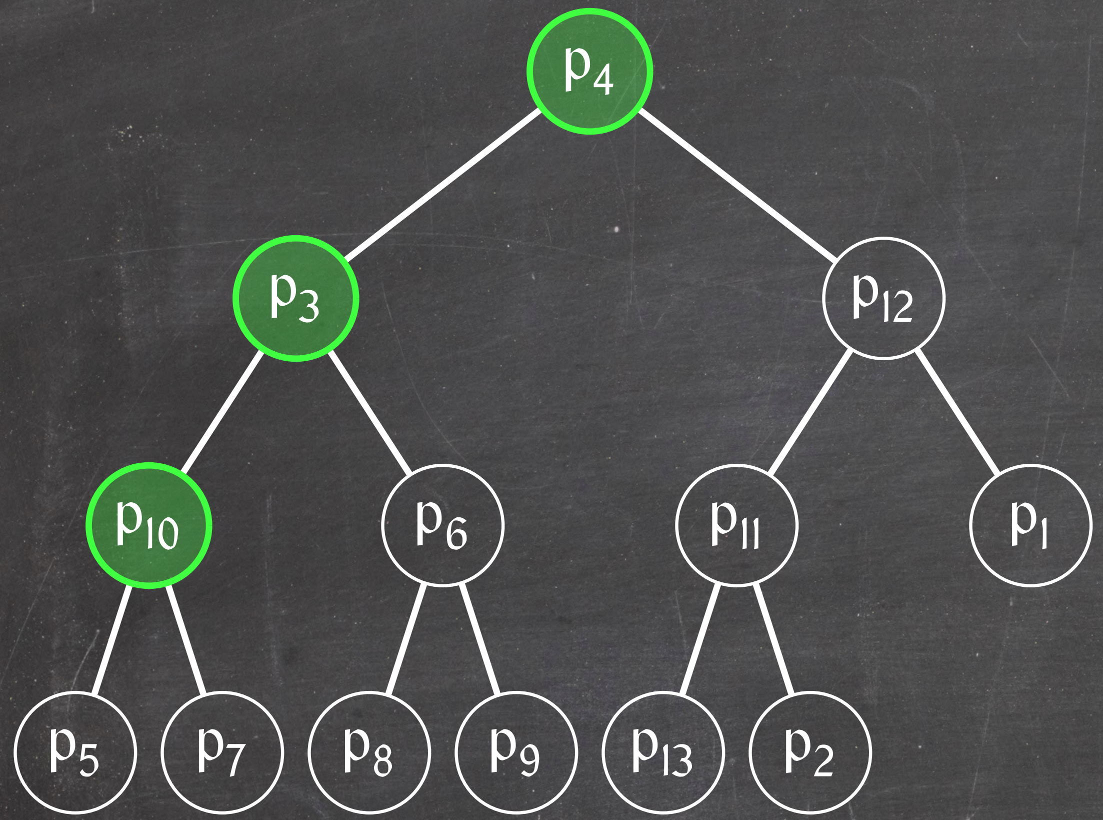


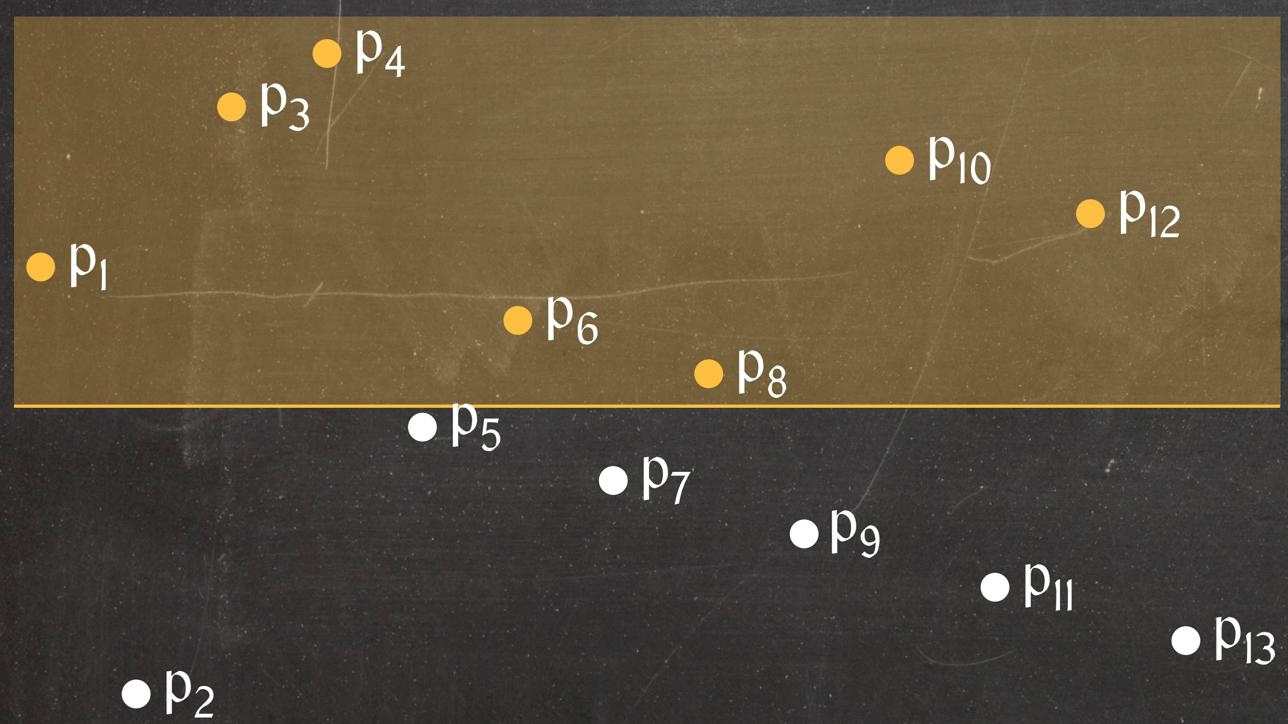
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound



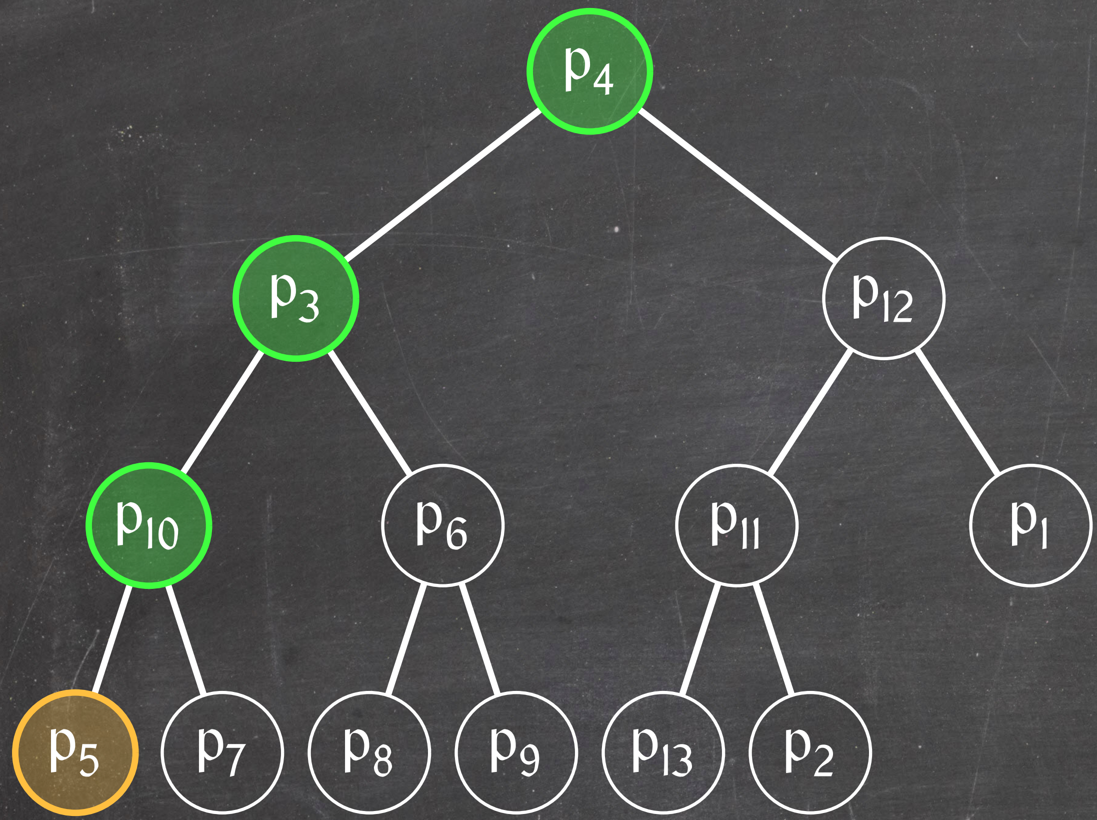


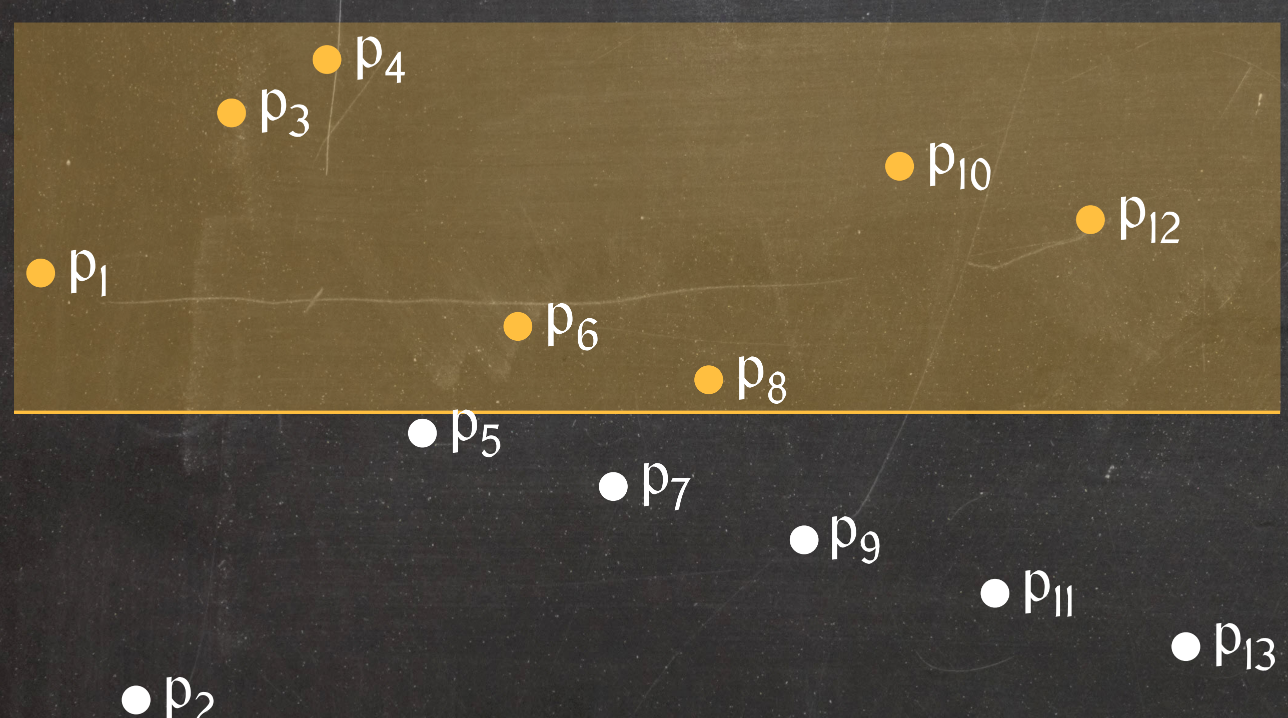
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound



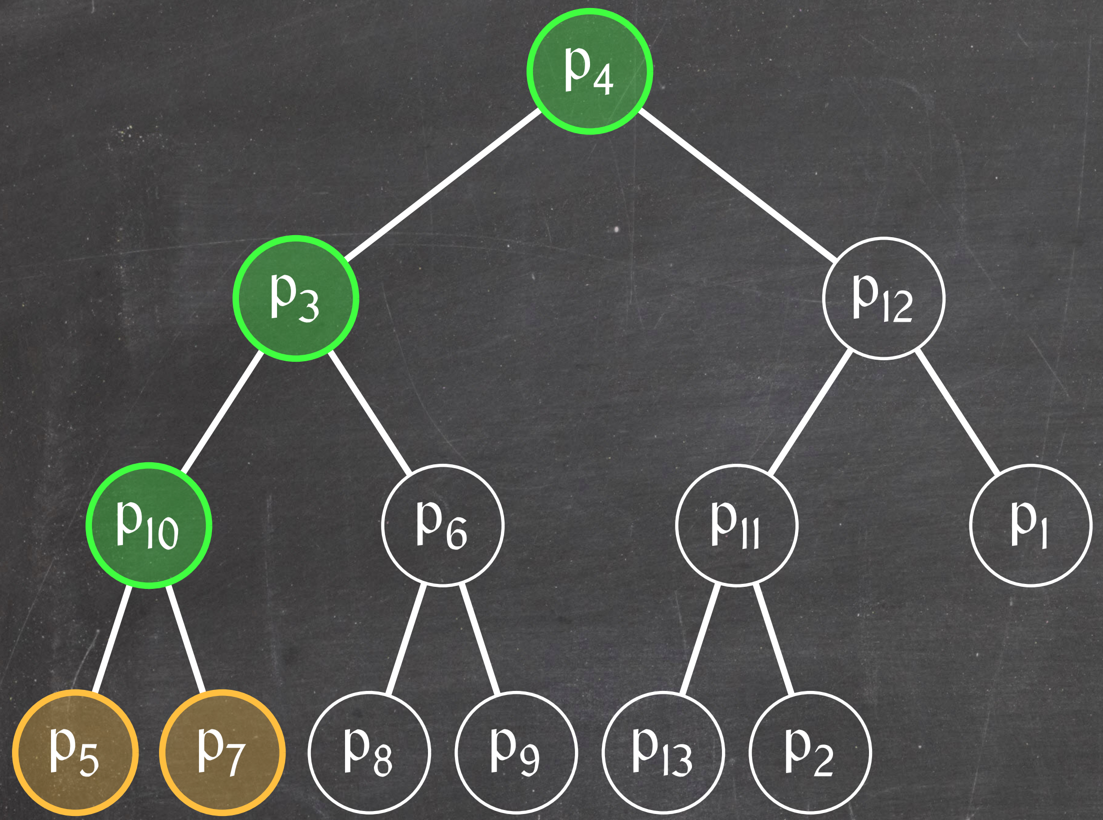


If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

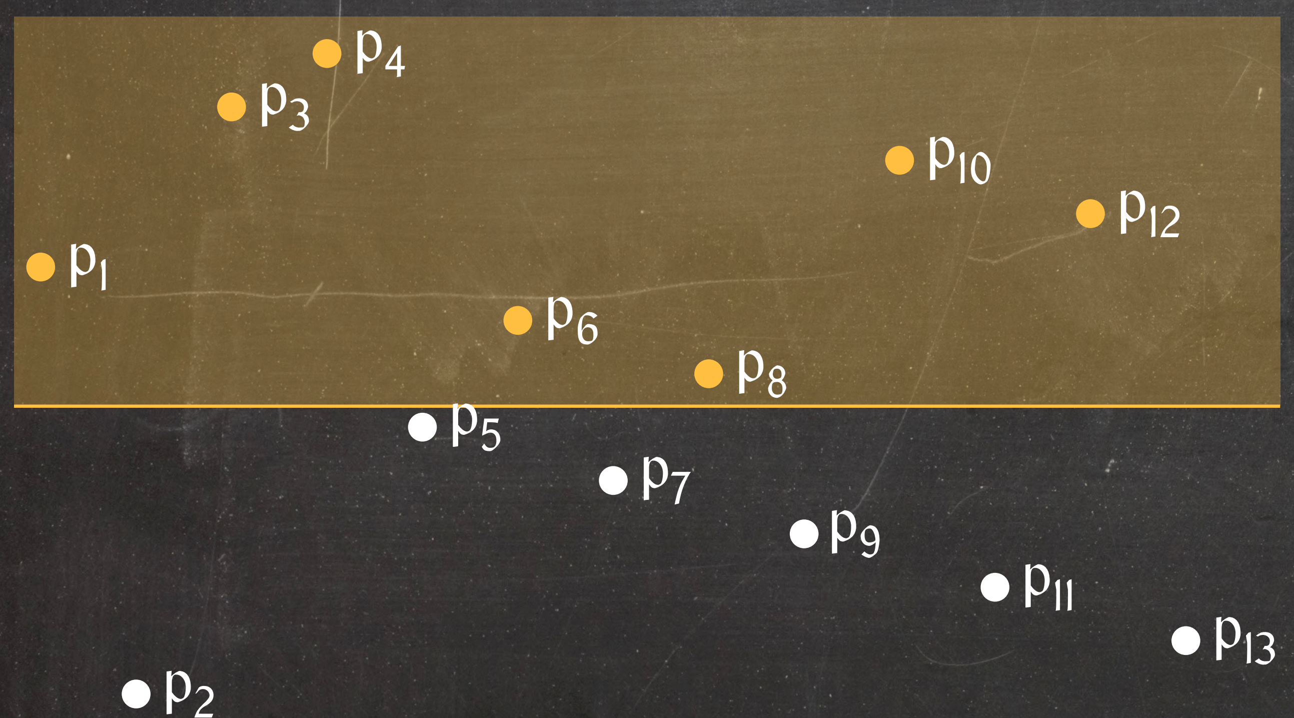
Heap Ordering and Searching With a Lower Bound



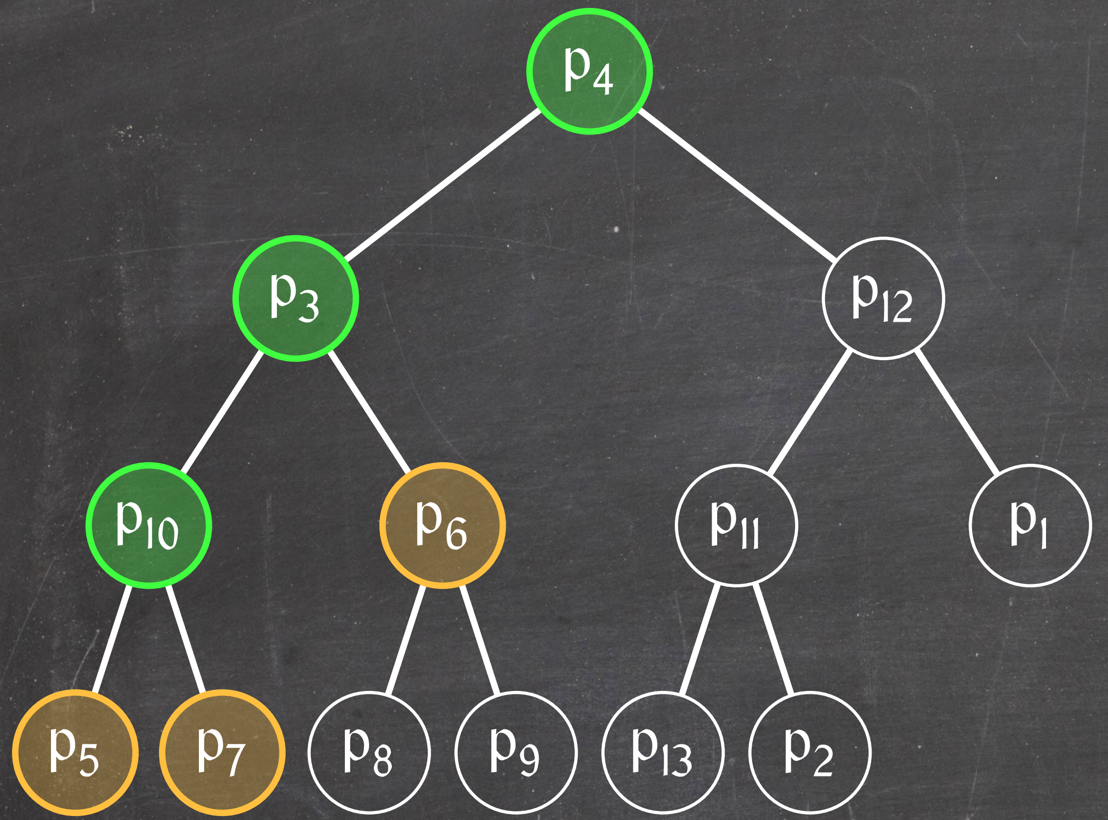
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

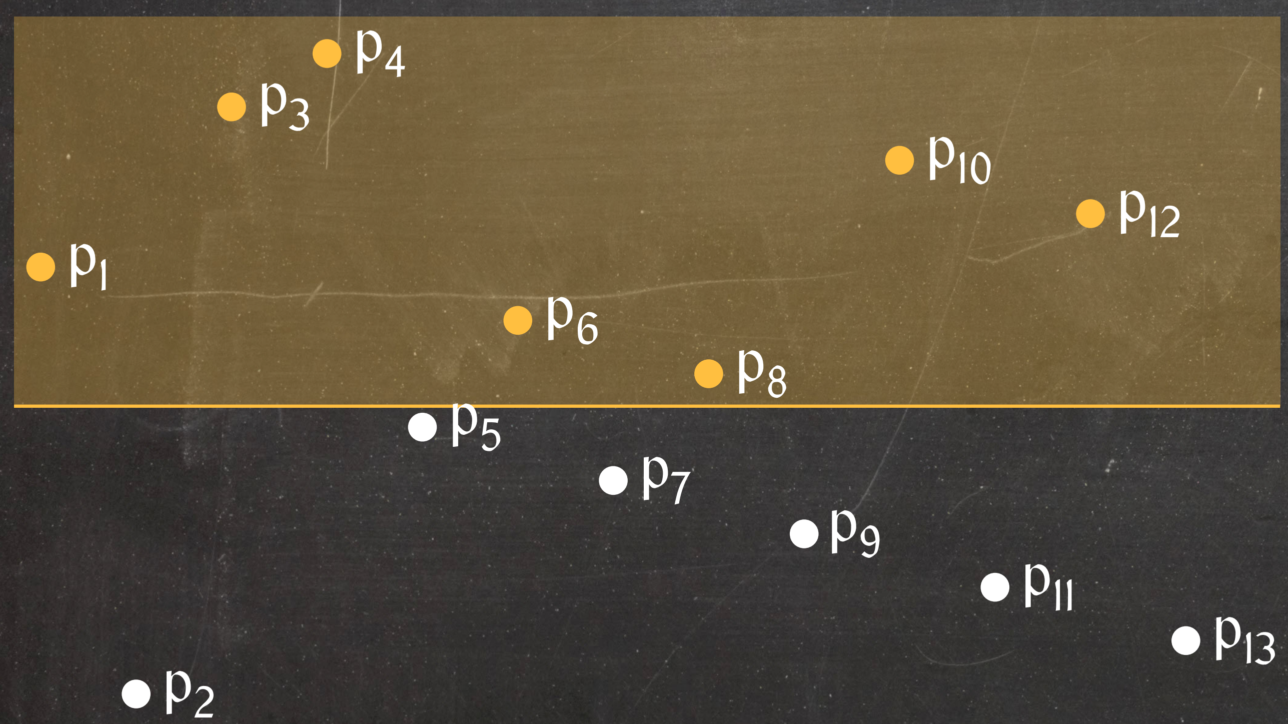
If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.



Heap Ordering and Searching With a Lower Bound



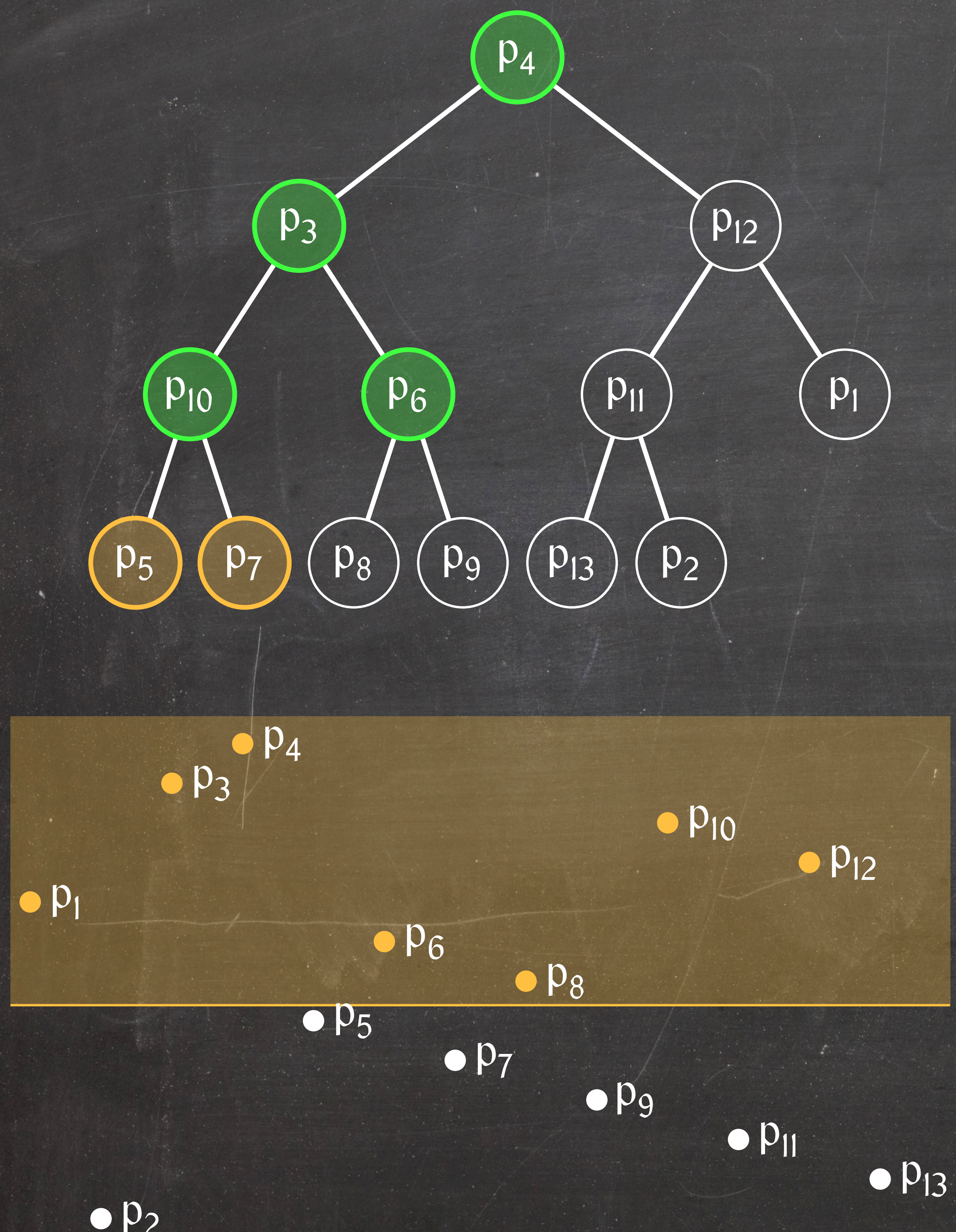


If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound

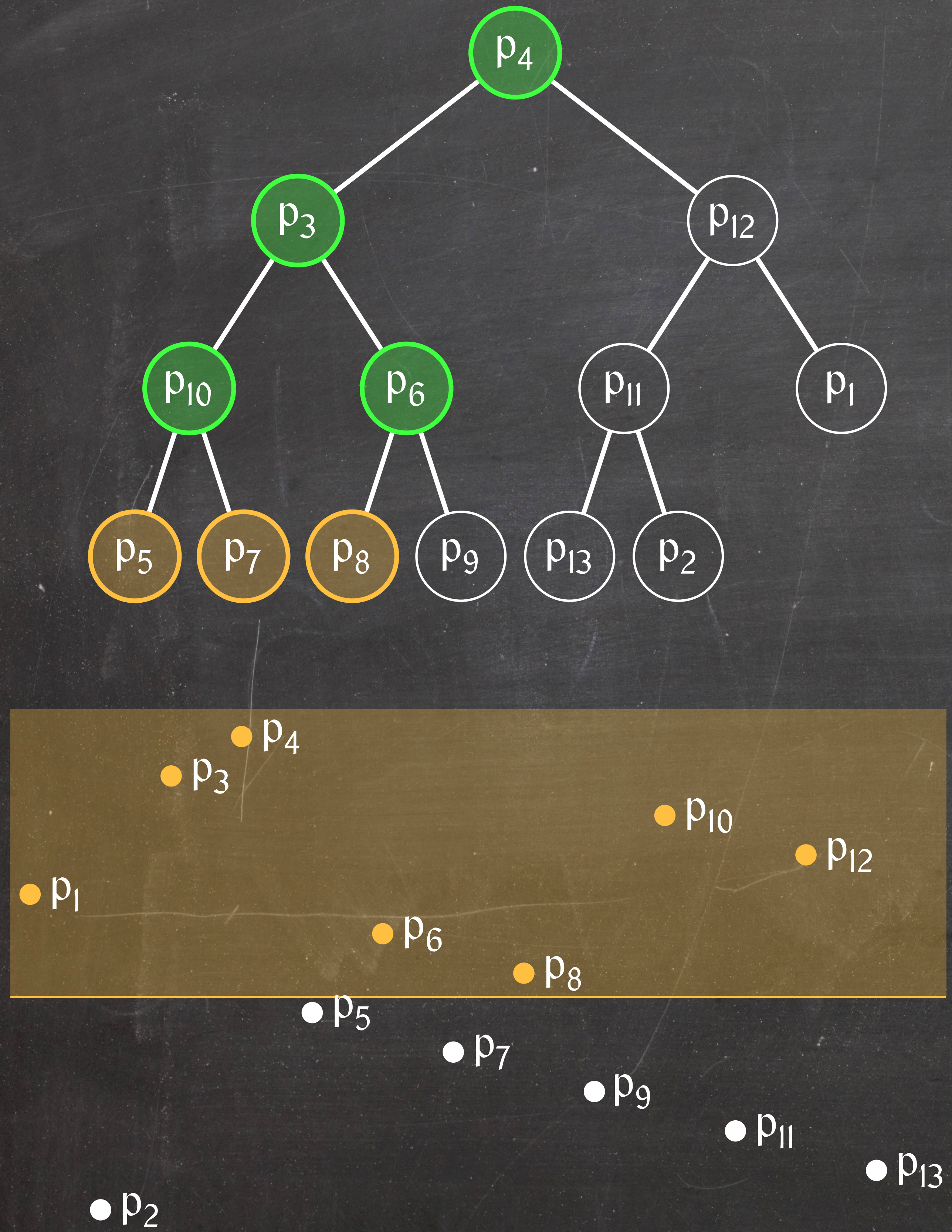


If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound

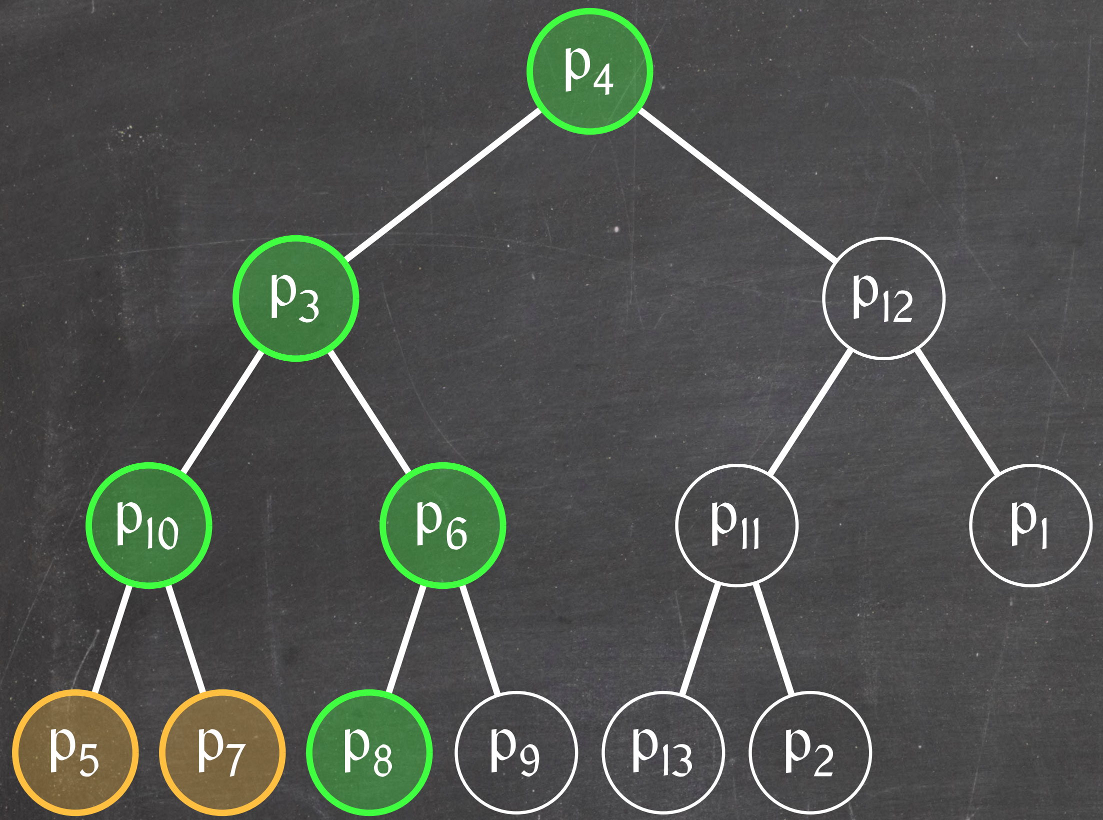


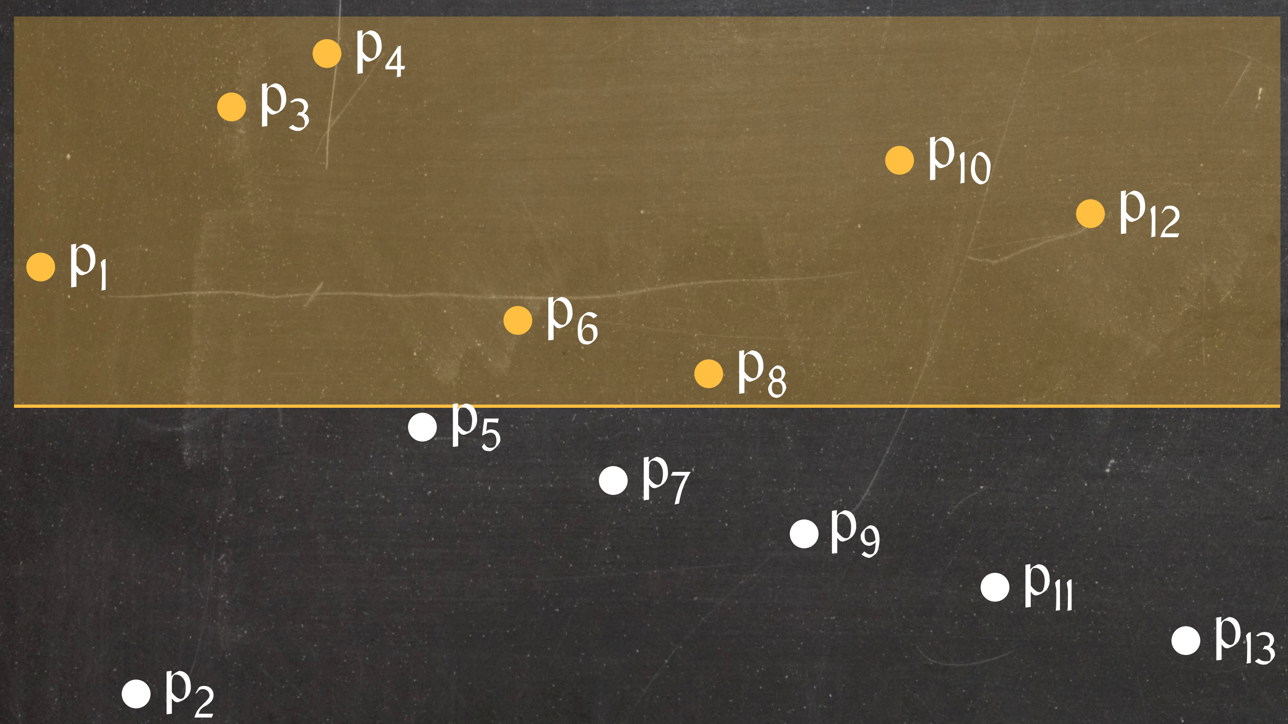
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in O(1 + k) time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound



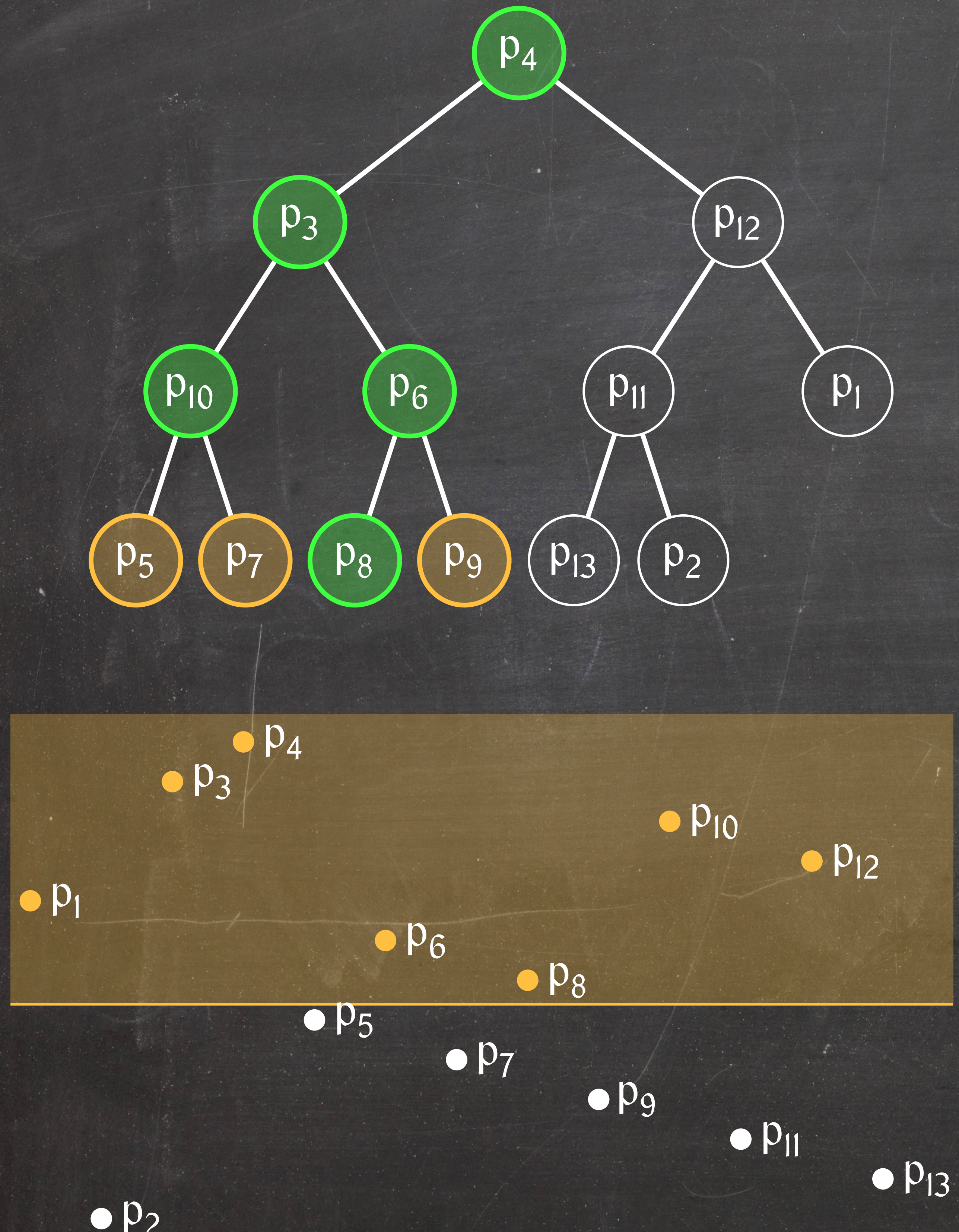


If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound

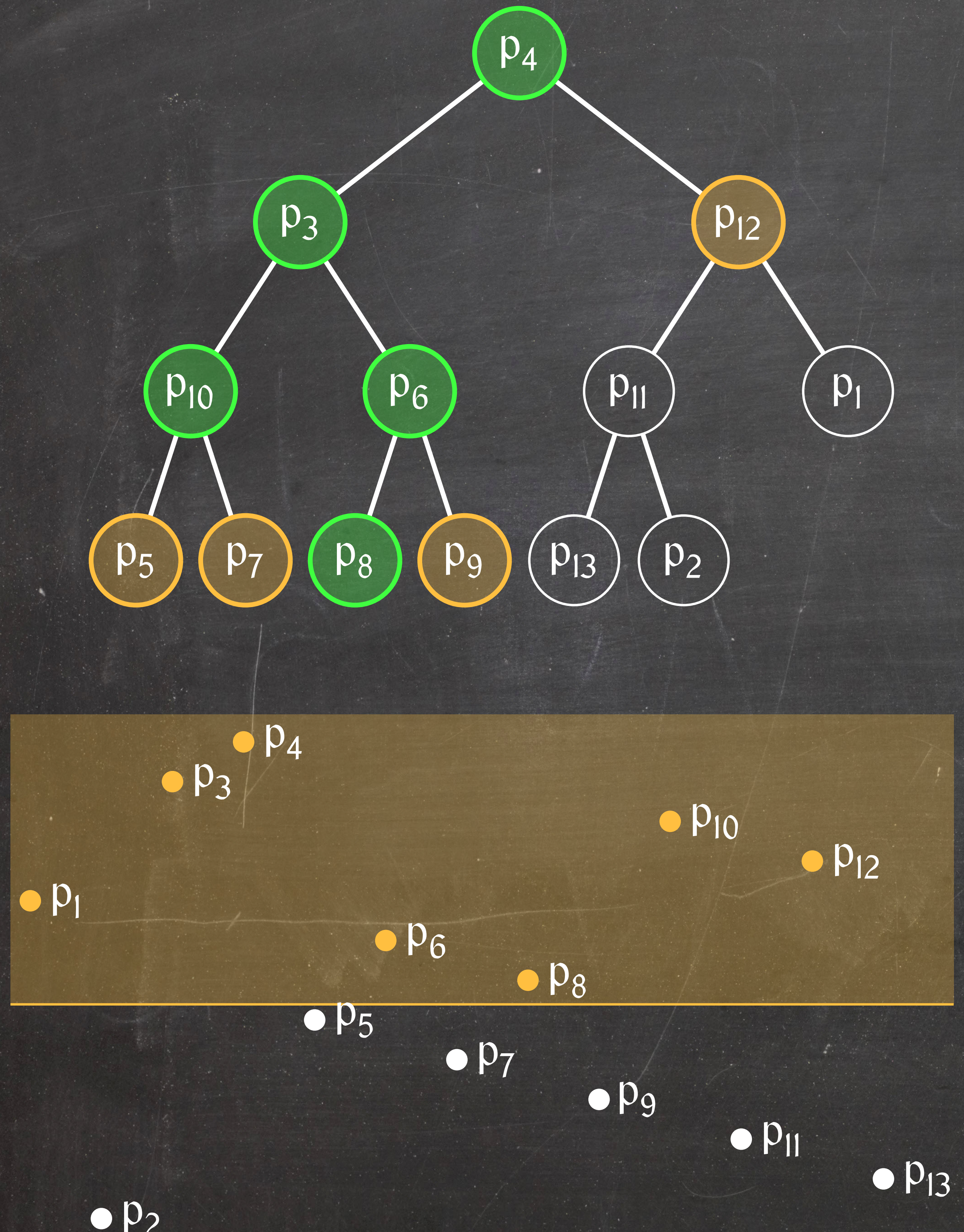


If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound

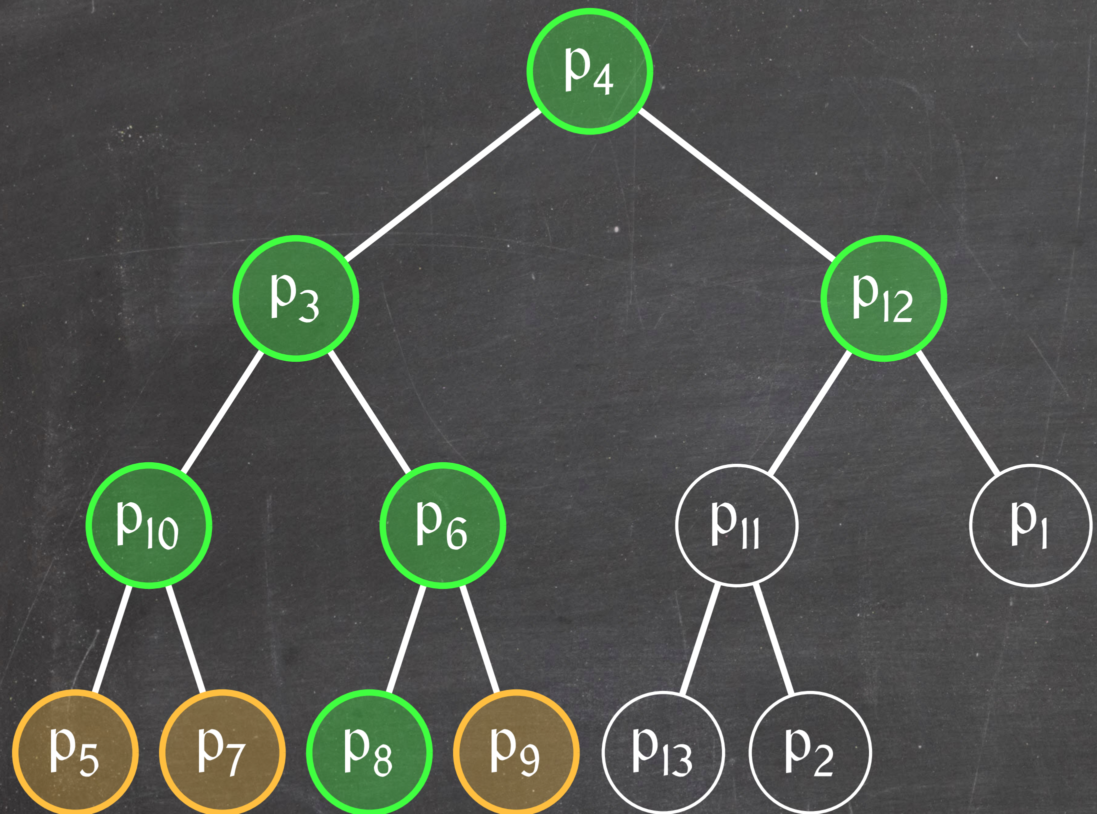


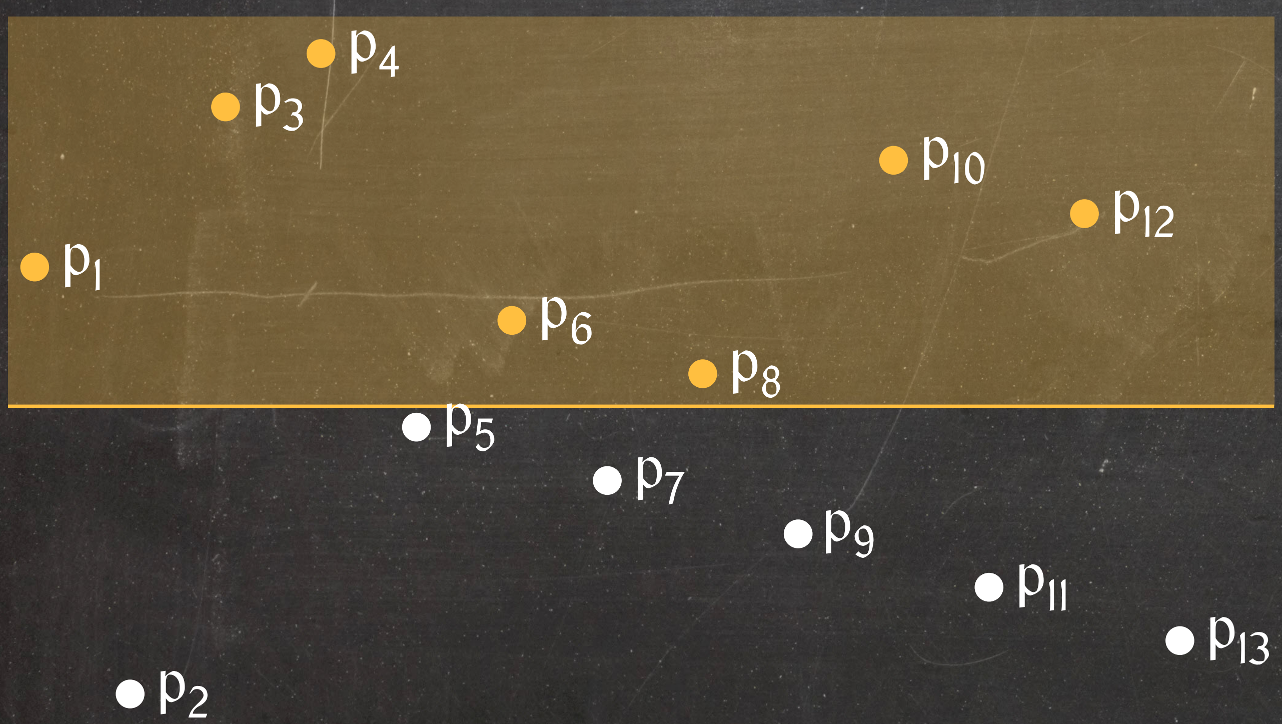
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound



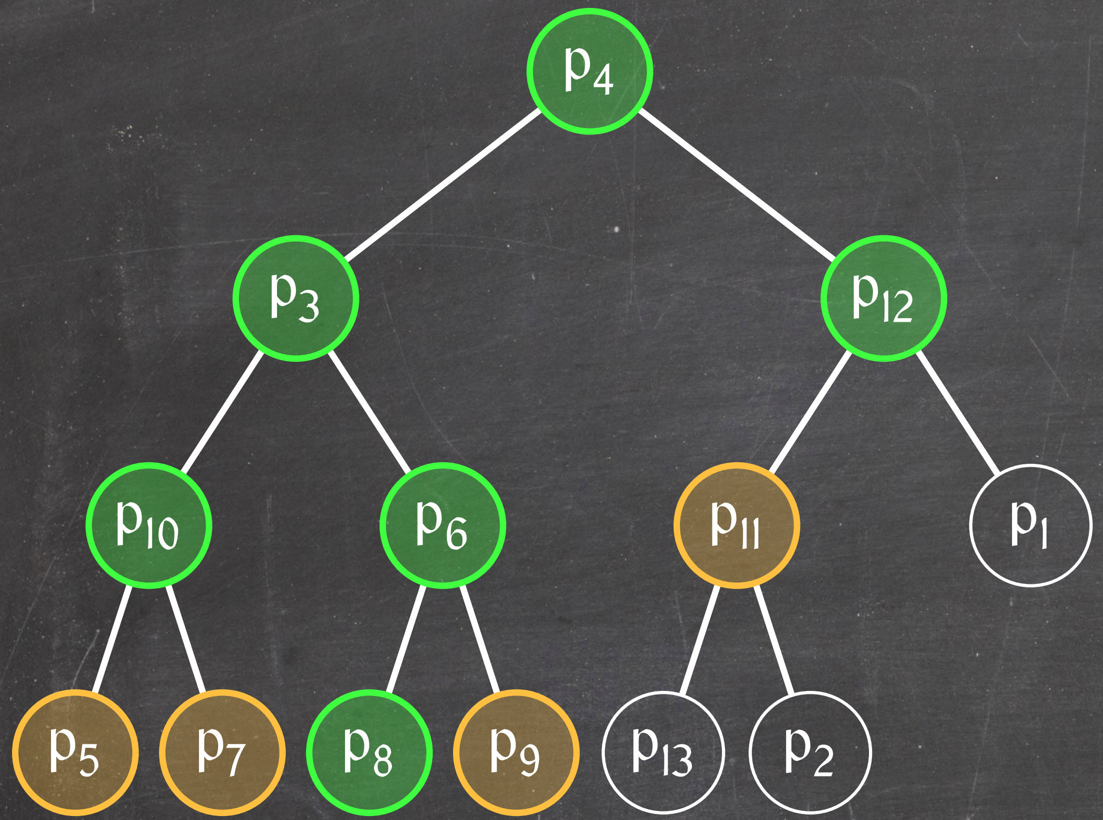


If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

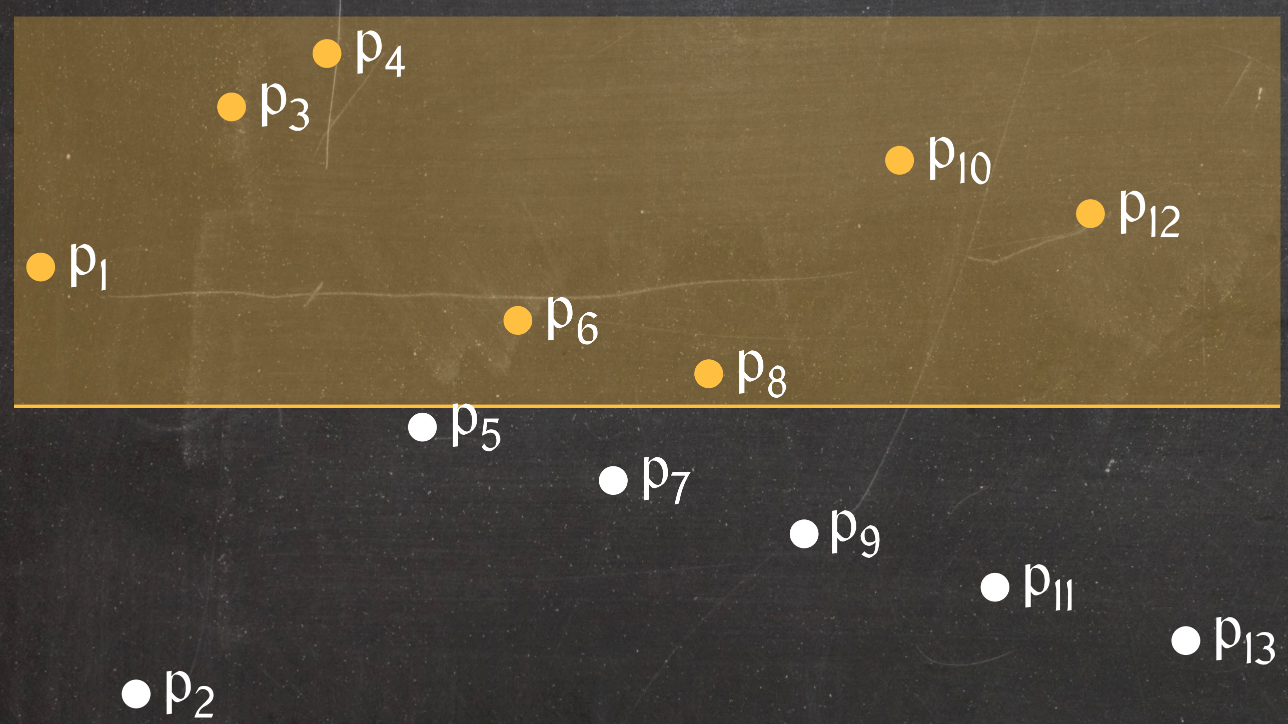
Heap Ordering and Searching With a Lower Bound



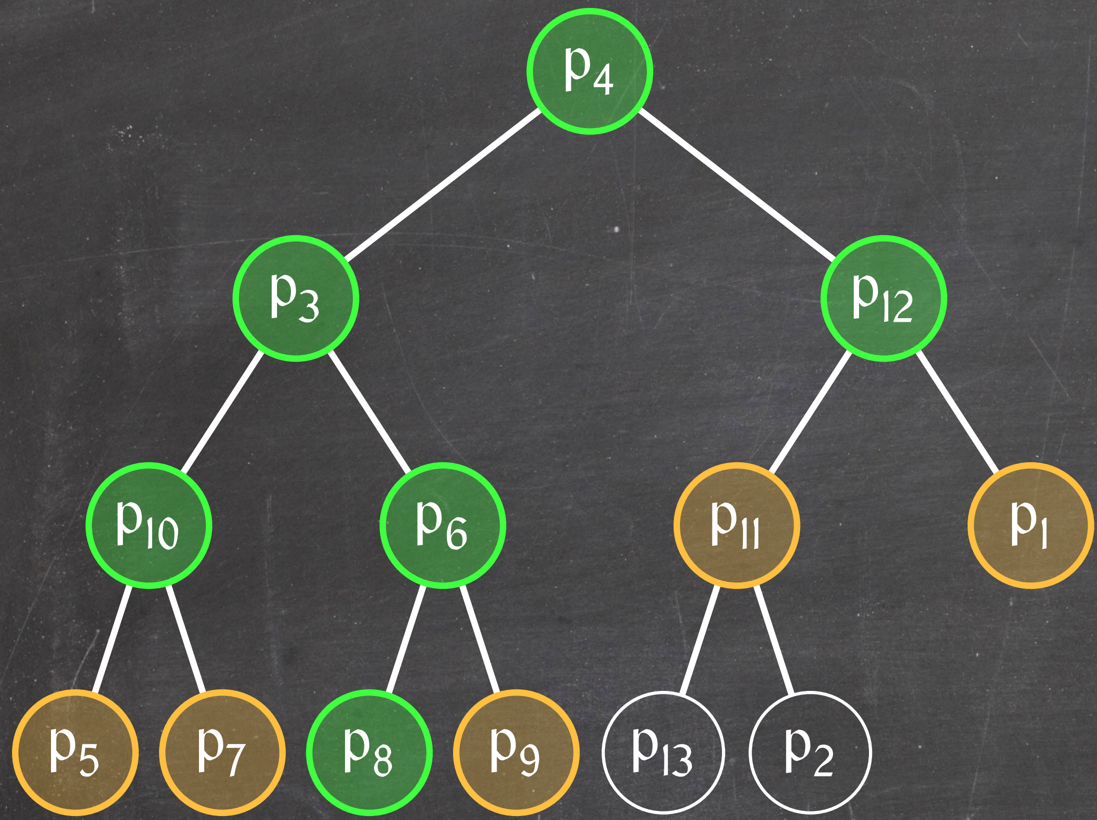
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.



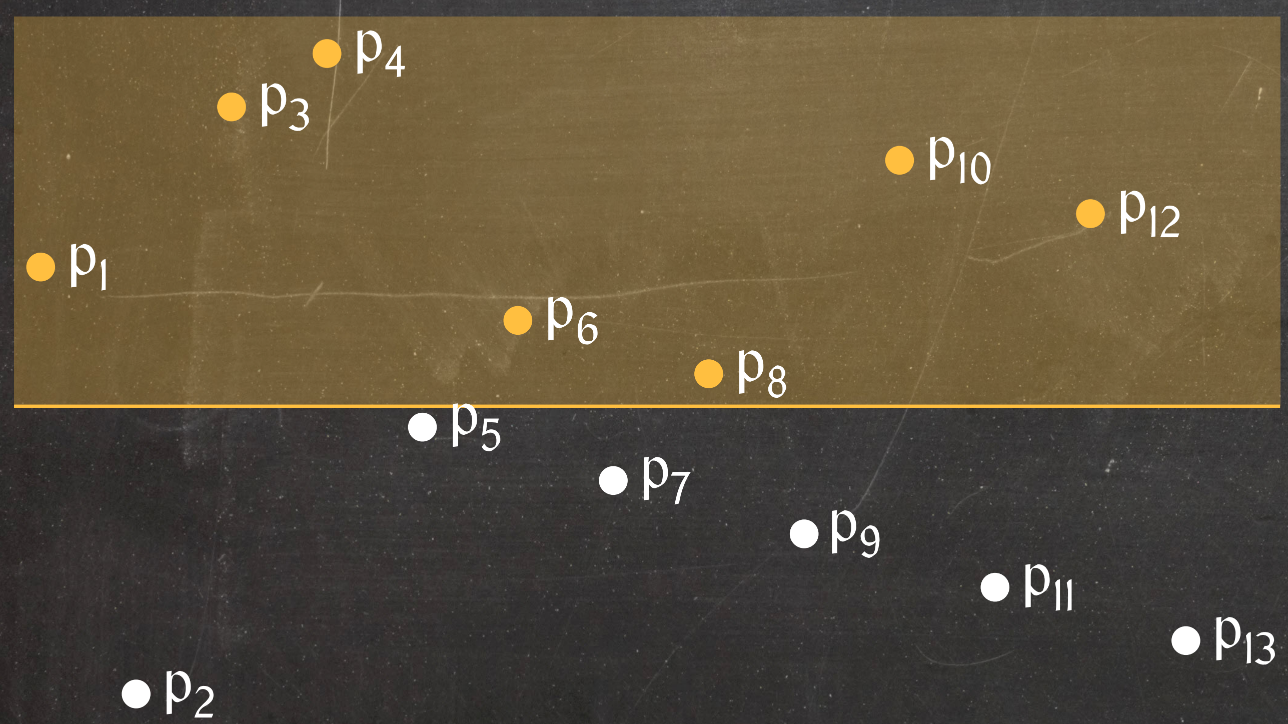
Heap Ordering and Searching With a Lower Bound



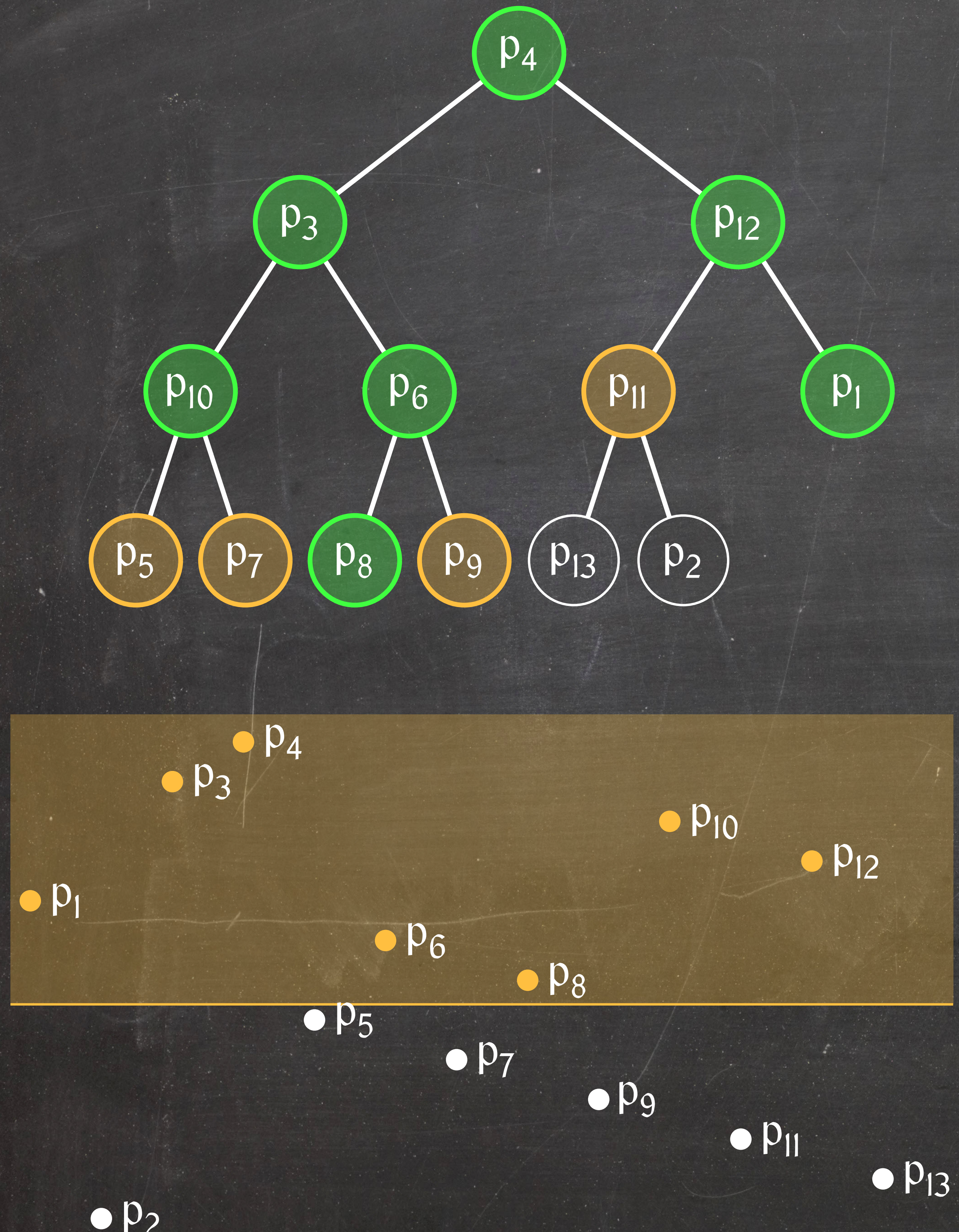
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.



Heap Ordering and Searching With a Lower Bound

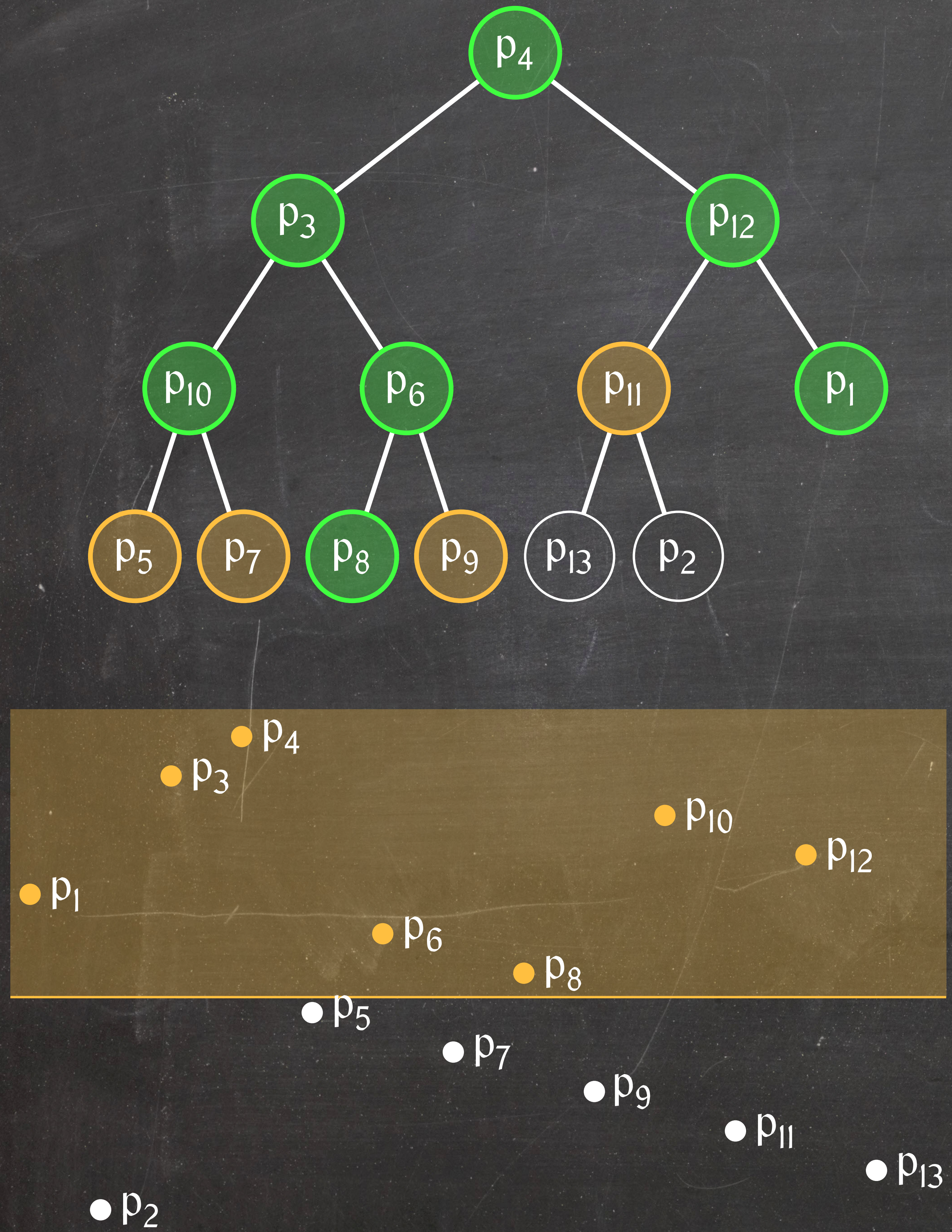


If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Heap Ordering and Searching With a Lower Bound



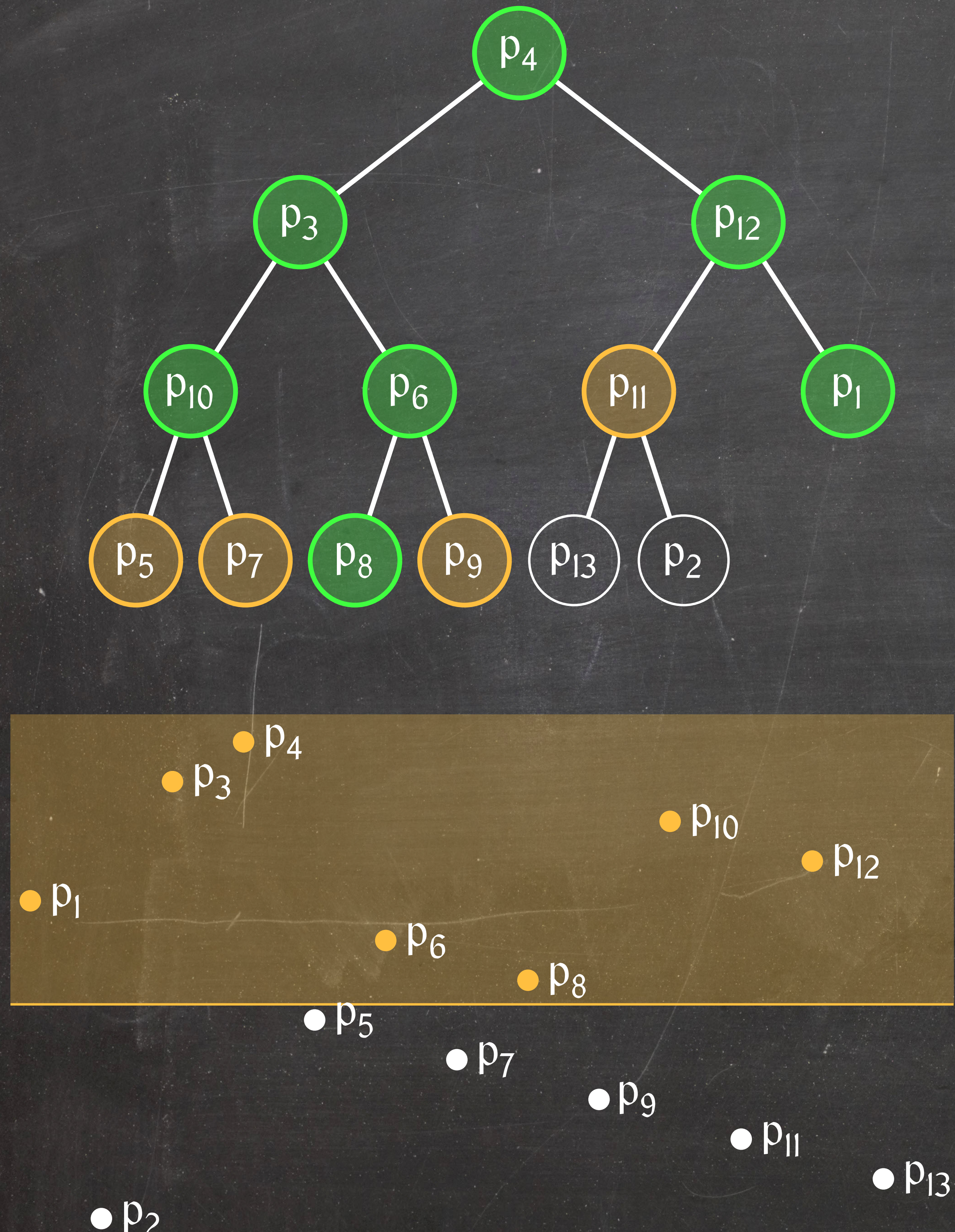
If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

Otherwise, output the point and inspect its children recursively.

Every node we visit, except the root, has a parent we output.

Heap Ordering and Searching With a Lower Bound



If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

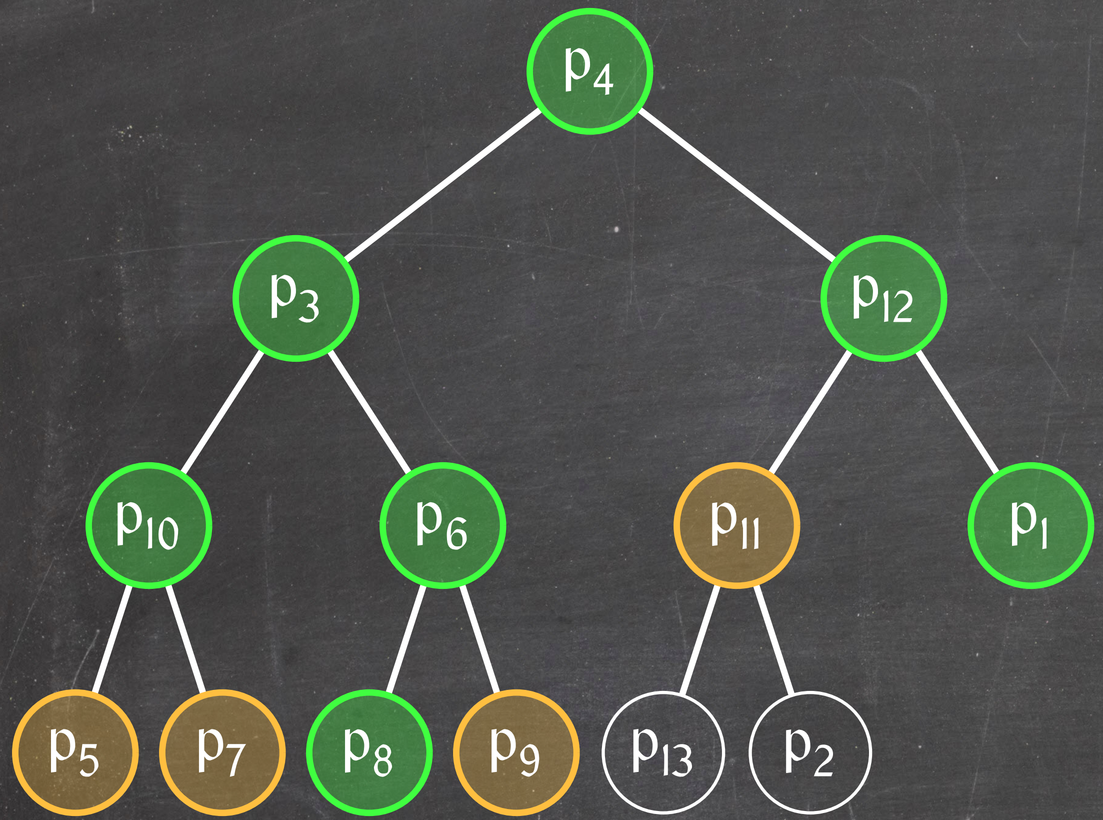
If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

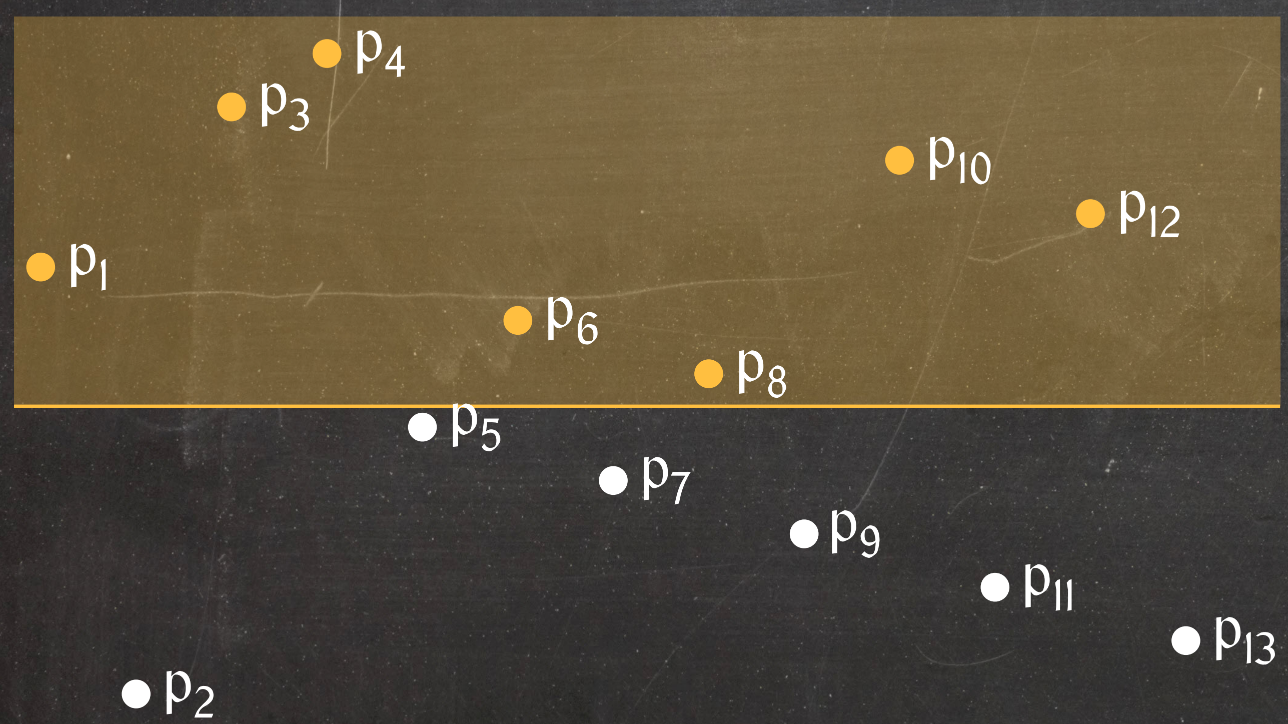
Otherwise, output the point and inspect its children recursively.

Every node we visit, except the root, has a parent we output.

At most two children per node.

Heap Ordering and Searching With a Lower Bound





If we store the points in a binary heap on the y-coordinates, can we report all the points above a query y-coordinate in $O(1 + k)$ time?

If the current point is below the query coordinate, none of its descendants can be above the query coordinate.

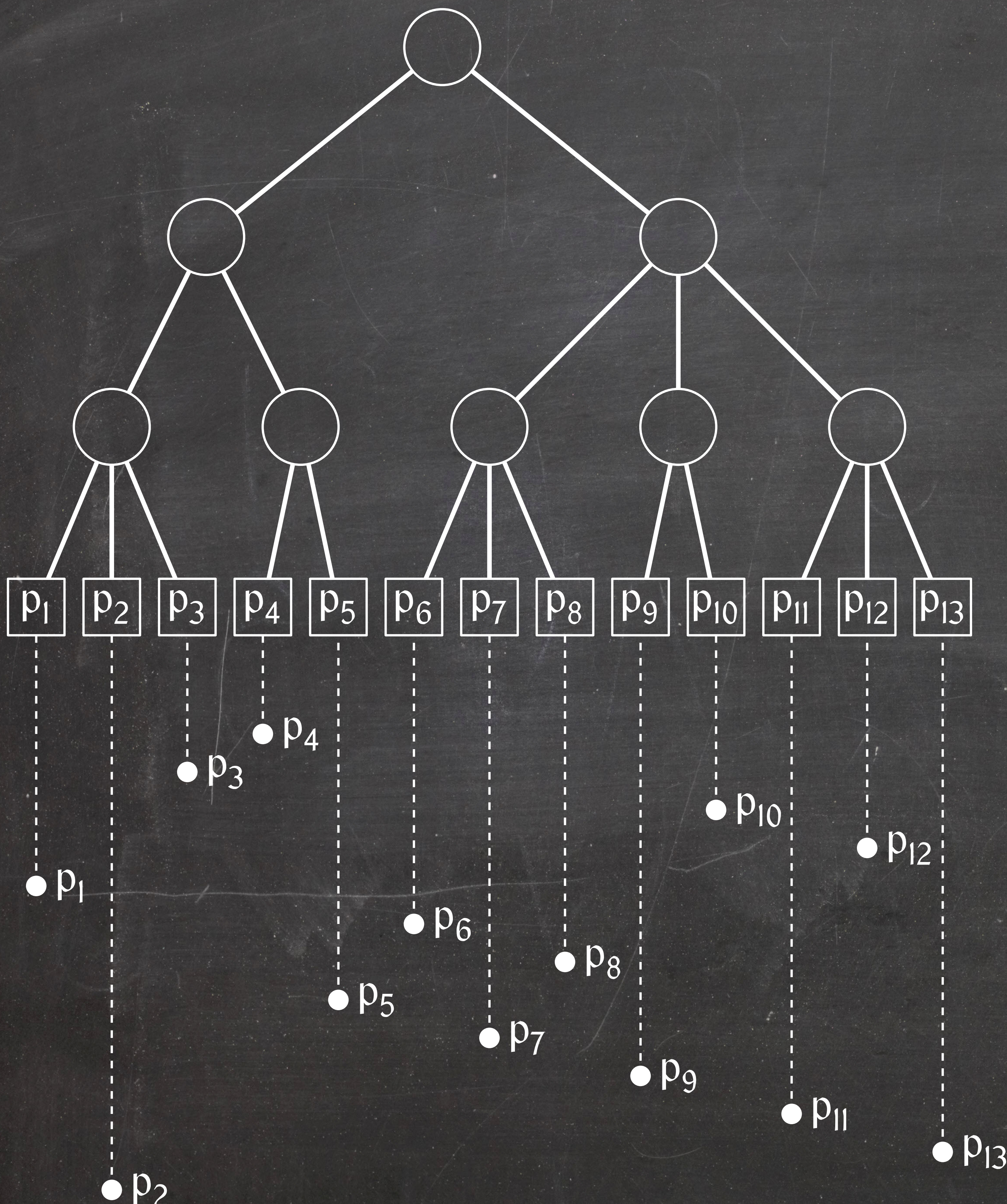
Otherwise, output the point and inspect its children recursively.

Every node we visit, except the root, has a parent we output.

At most two children per node.

$\Rightarrow$ We visit at most $1 + 2k$ nodes.

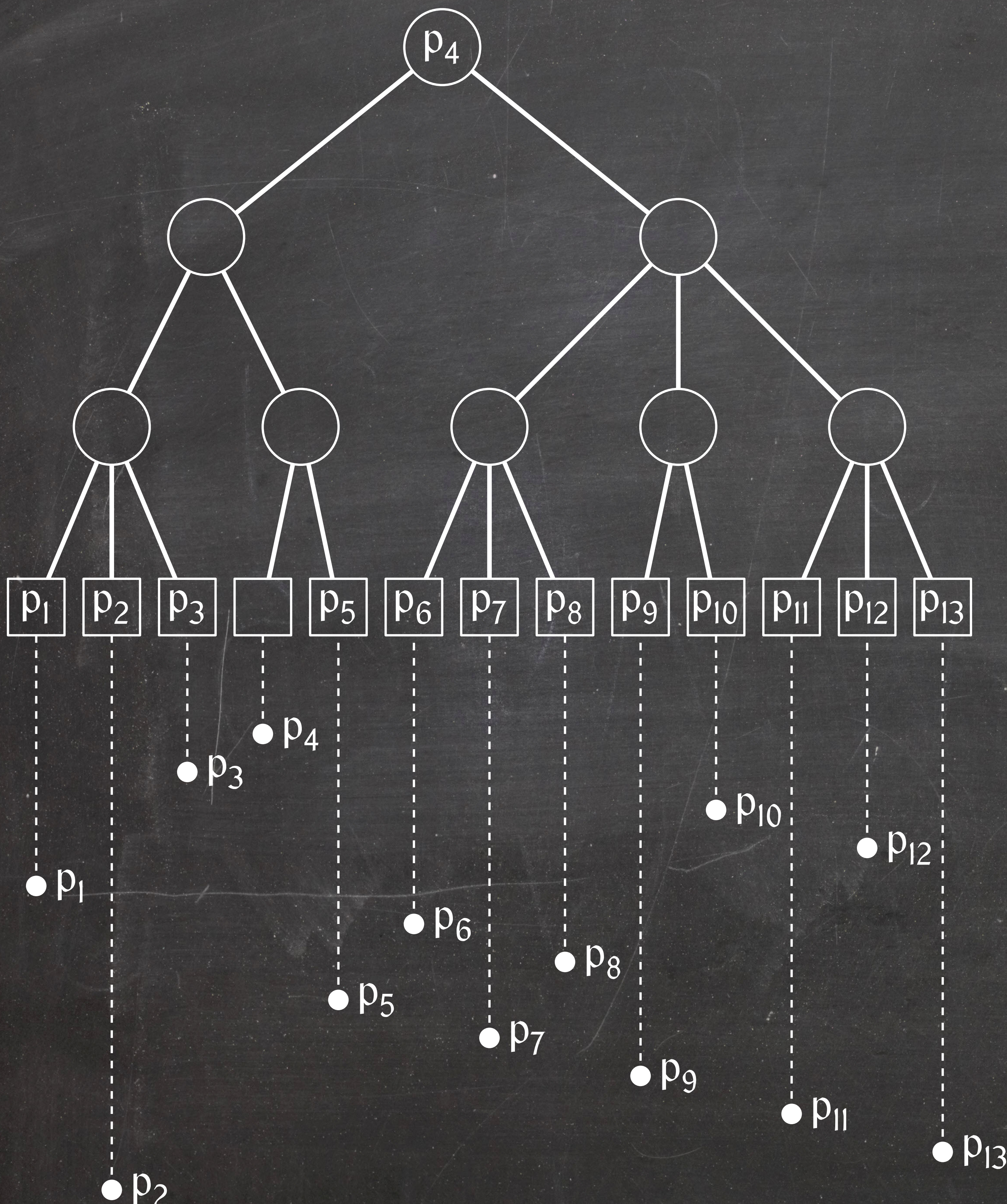
A Tree That's a Search Tree (on x) and a Heap (on y)



Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

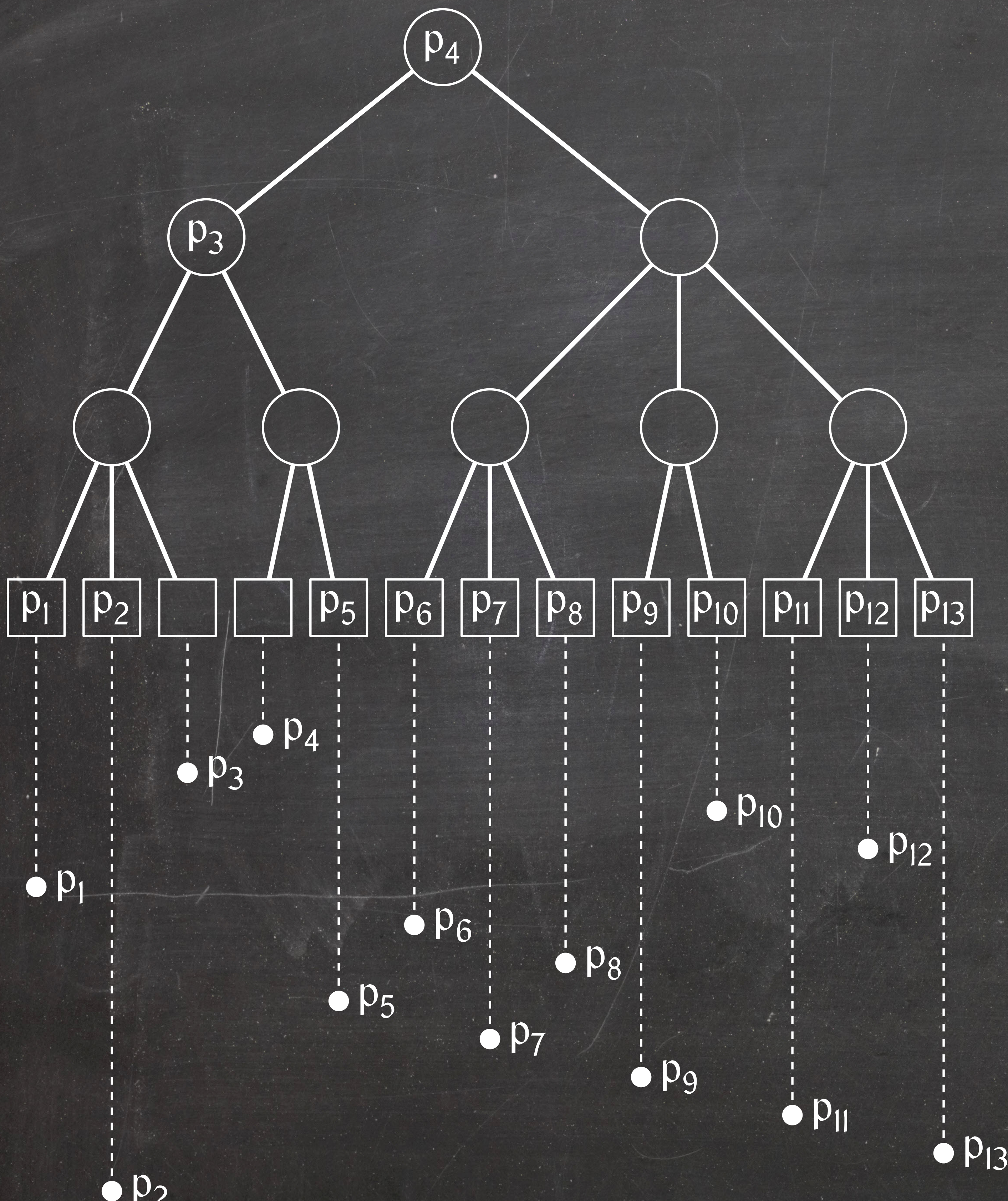
A Tree That's a Search Tree (on x) and a Heap (on y)



Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

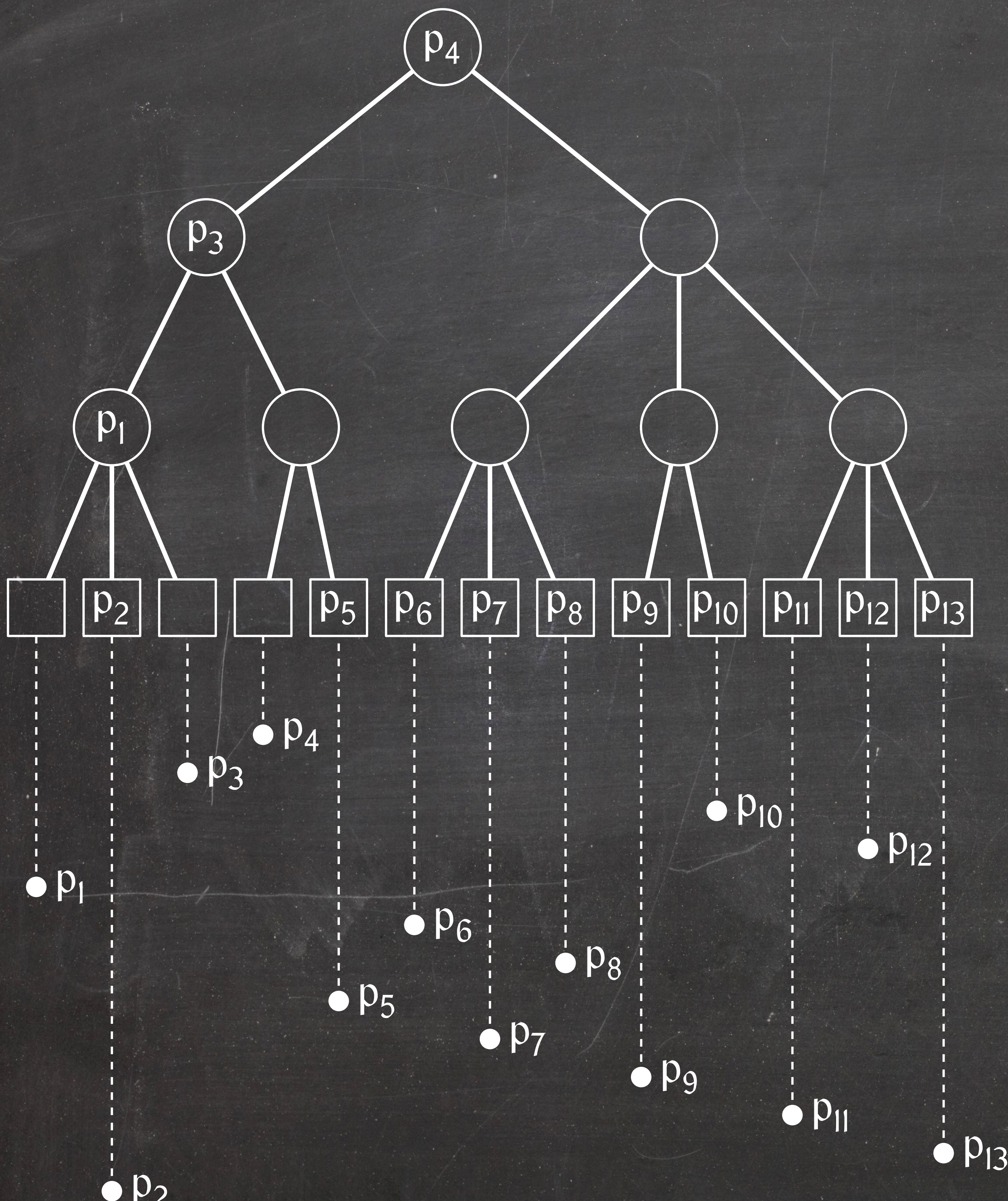
A Tree That's a Search Tree (on x) and a Heap (on y)



Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

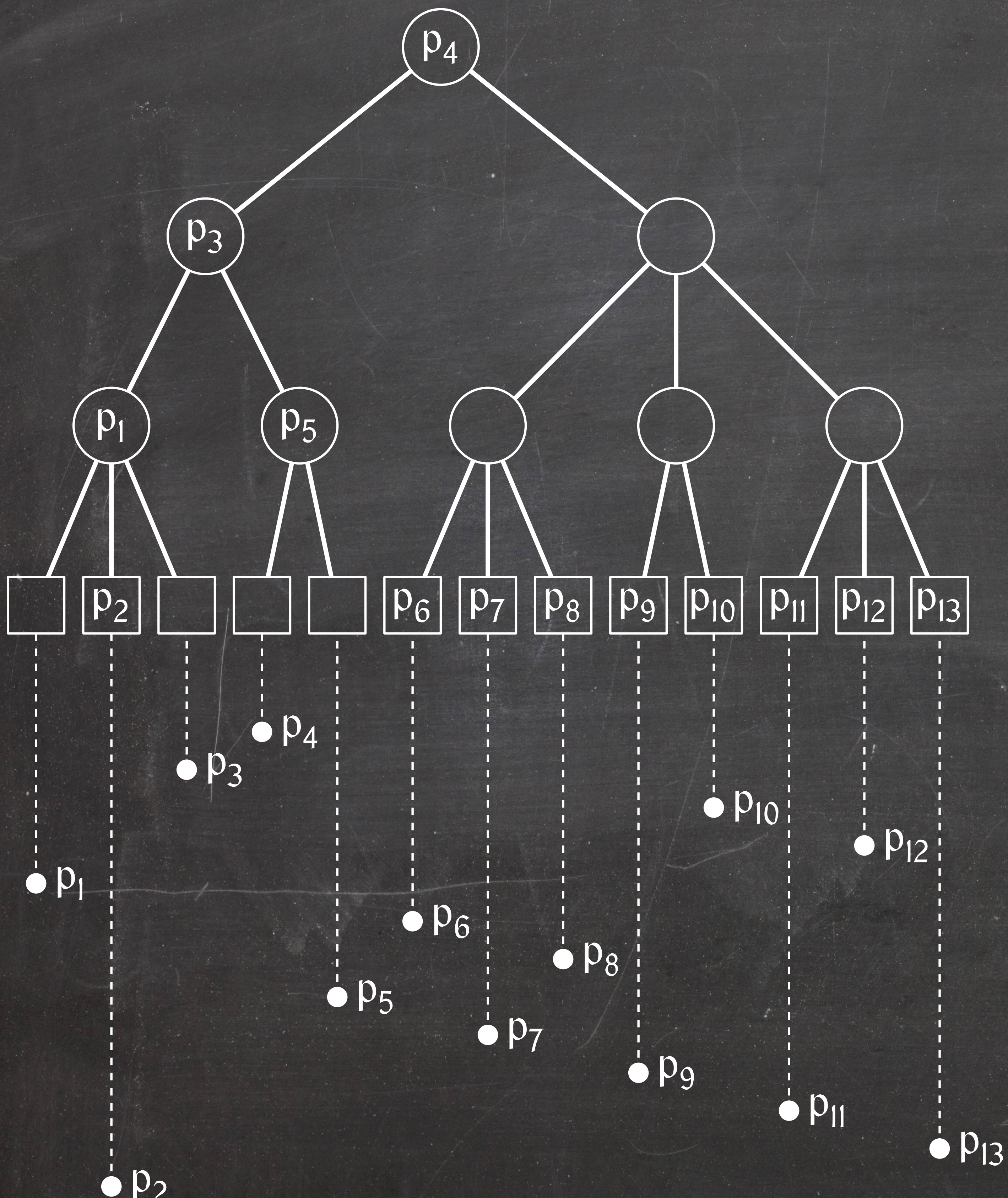
A Tree That's a Search Tree (on x) and a Heap (on y)



Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

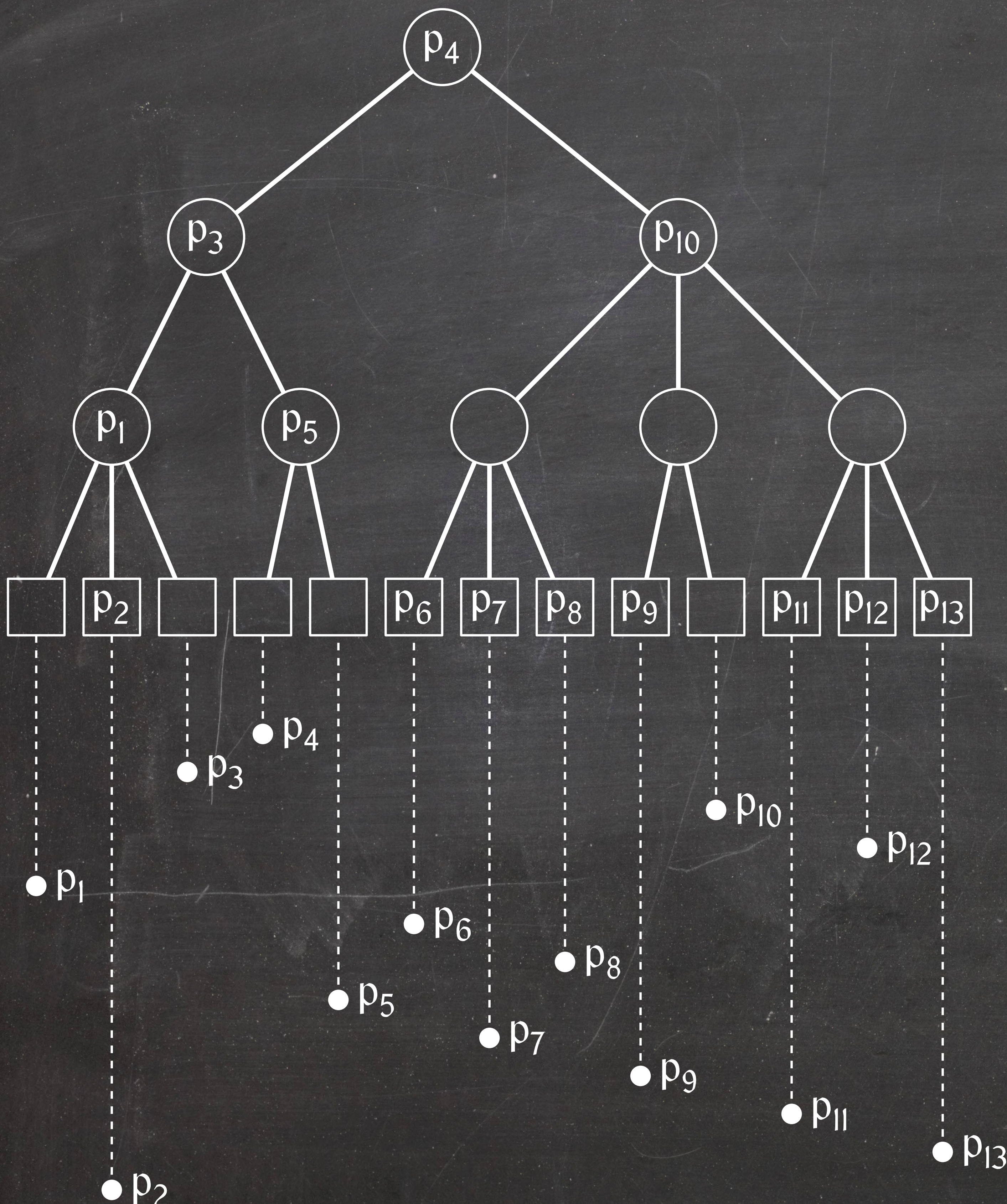
A Tree That's a Search Tree (on x) and a Heap (on y)



Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

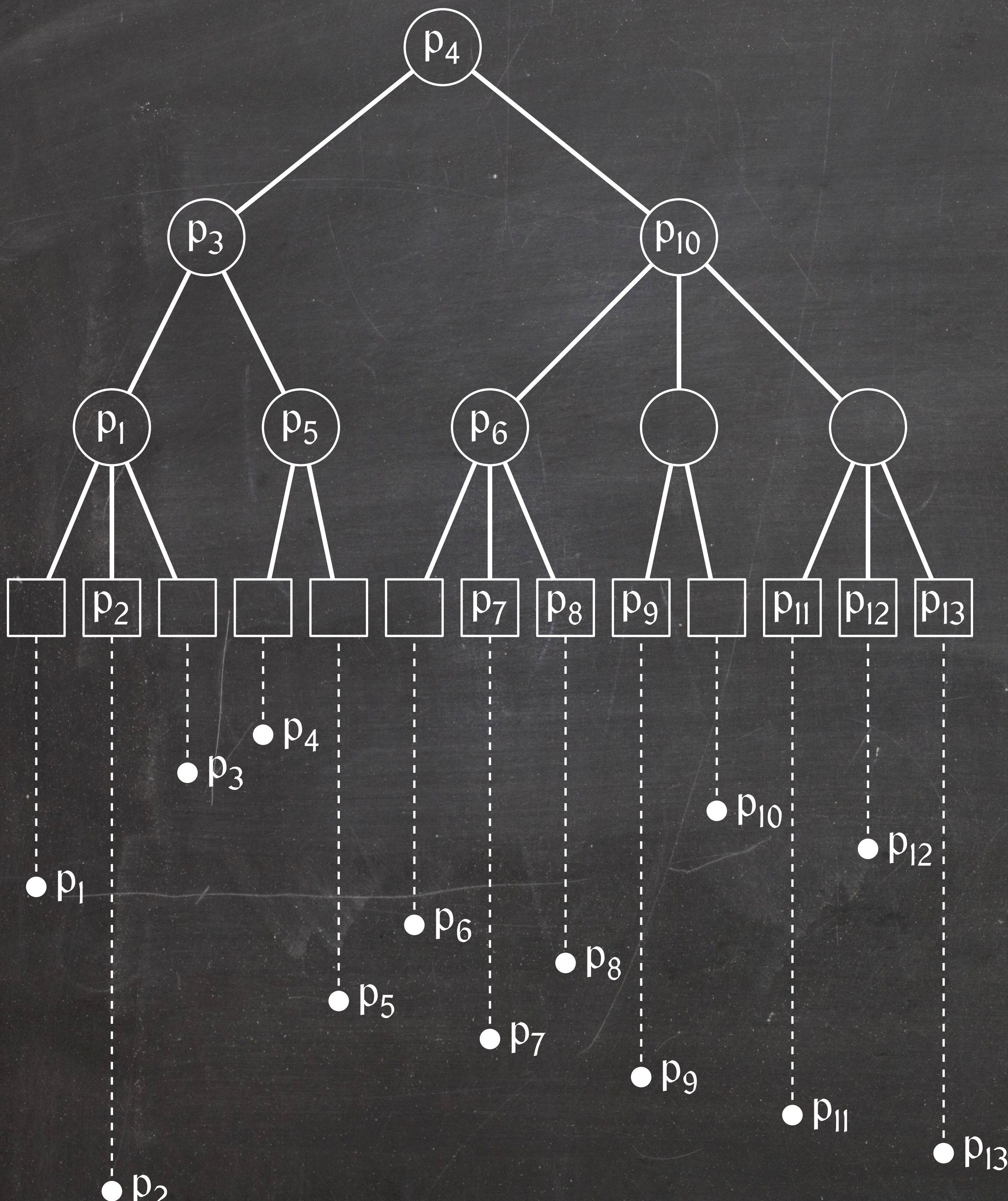
A Tree That's a Search Tree (on x) and a Heap (on y)



Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

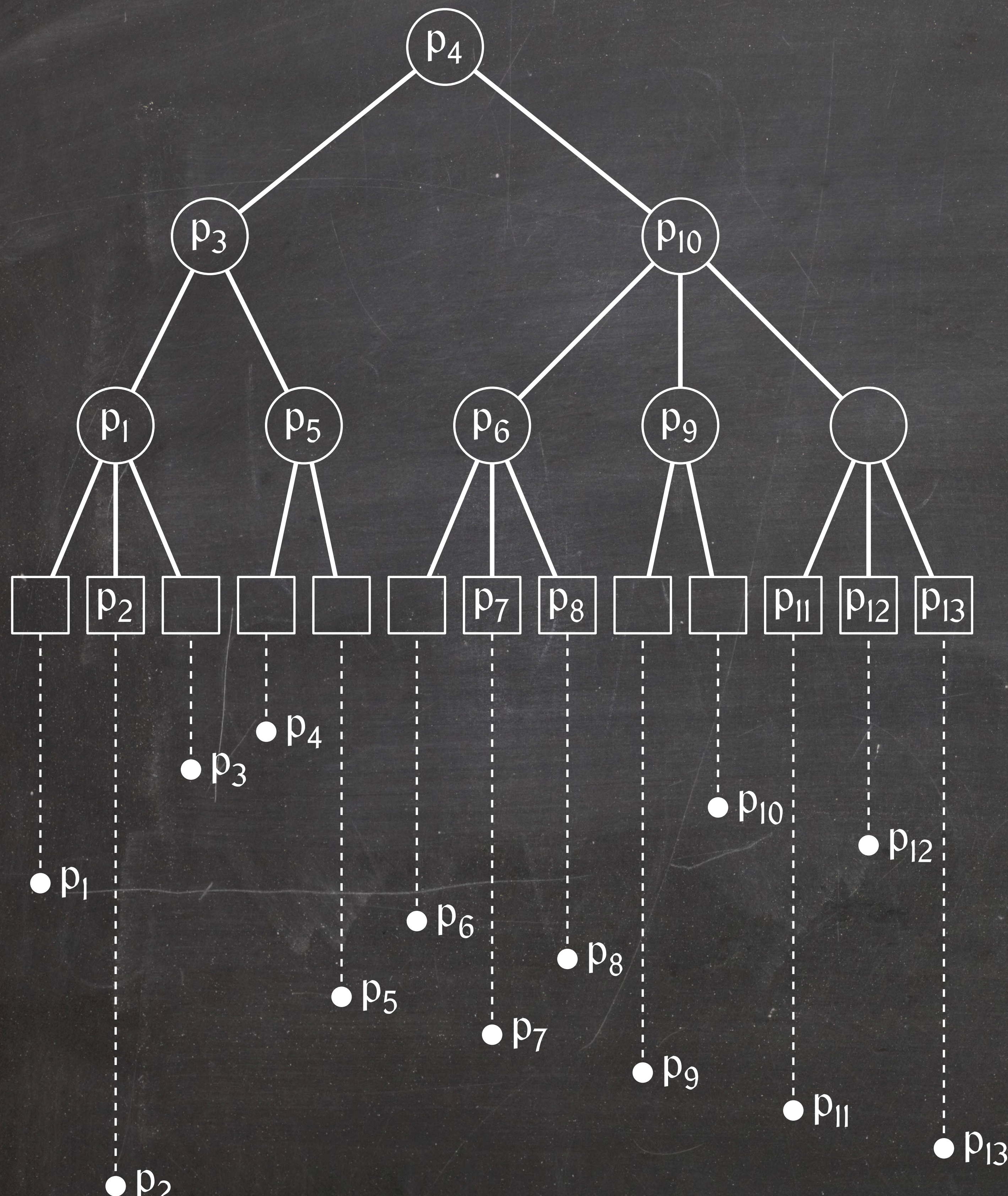
A Tree That's a Search Tree (on x) and a Heap (on y)



Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

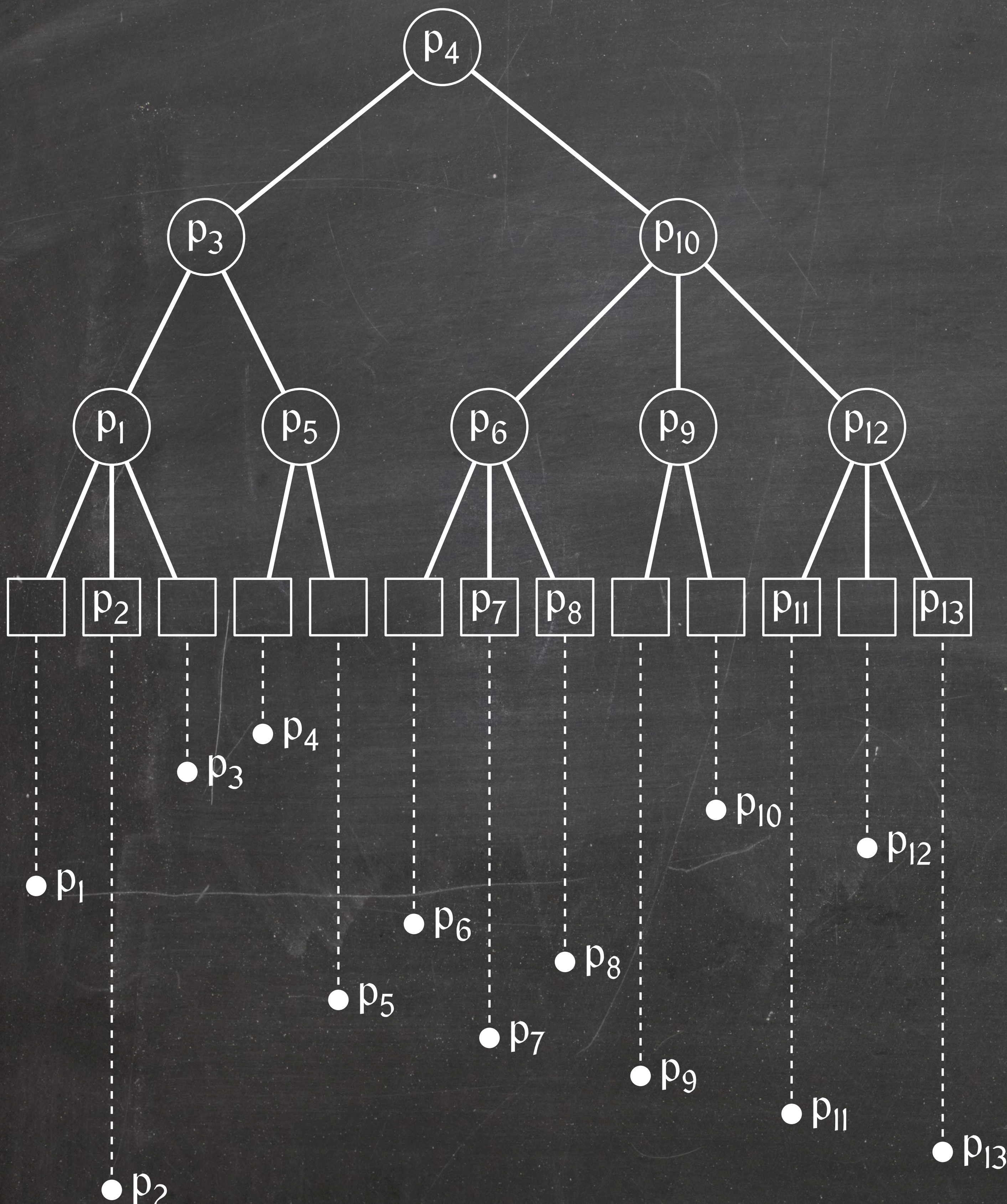
A Tree That's a Search Tree (on x) and a Heap (on y)



Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

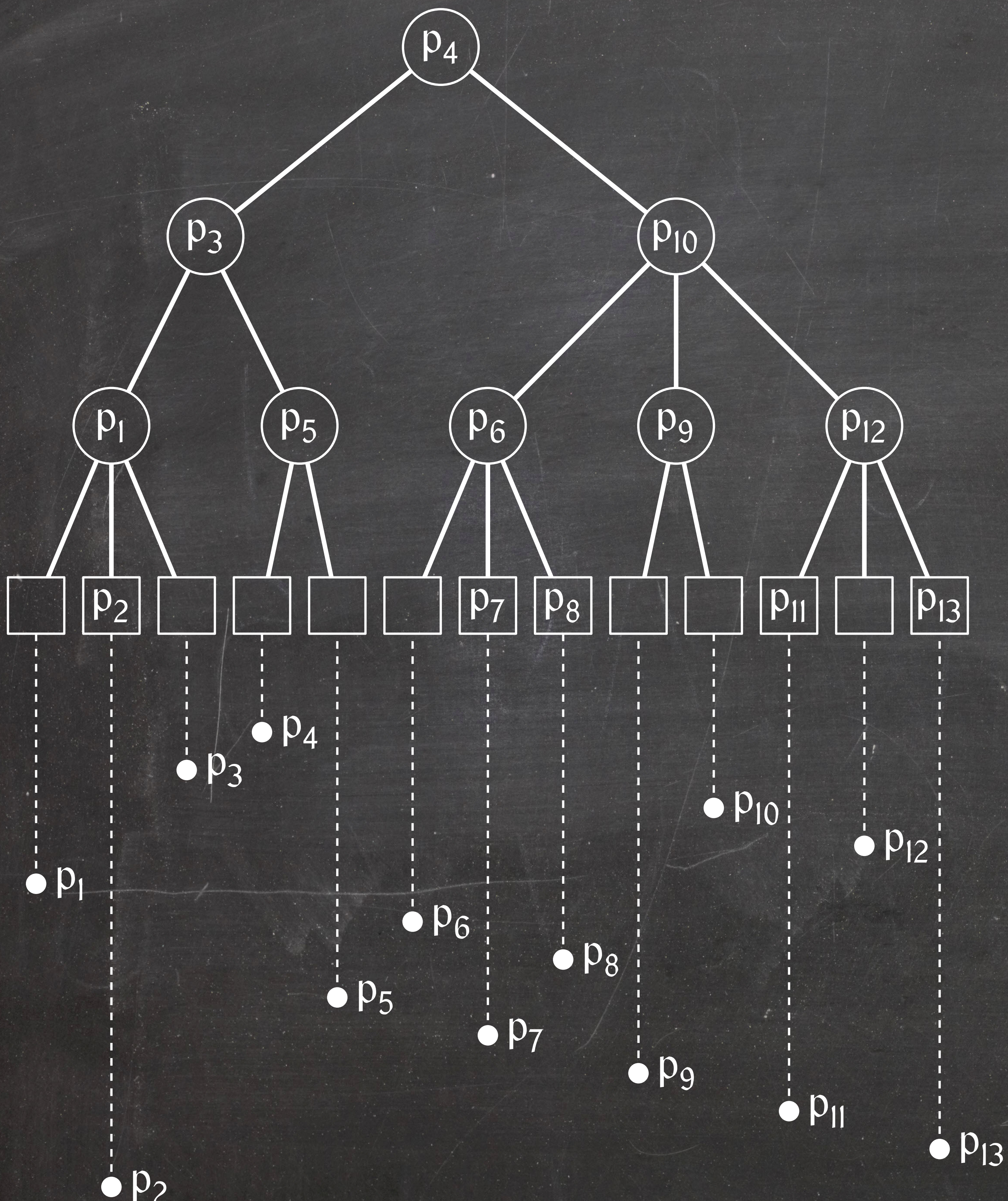
A Tree That's a Search Tree (on x) and a Heap (on y)



Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

A Tree That's a Search Tree (on x) and a Heap (on y)

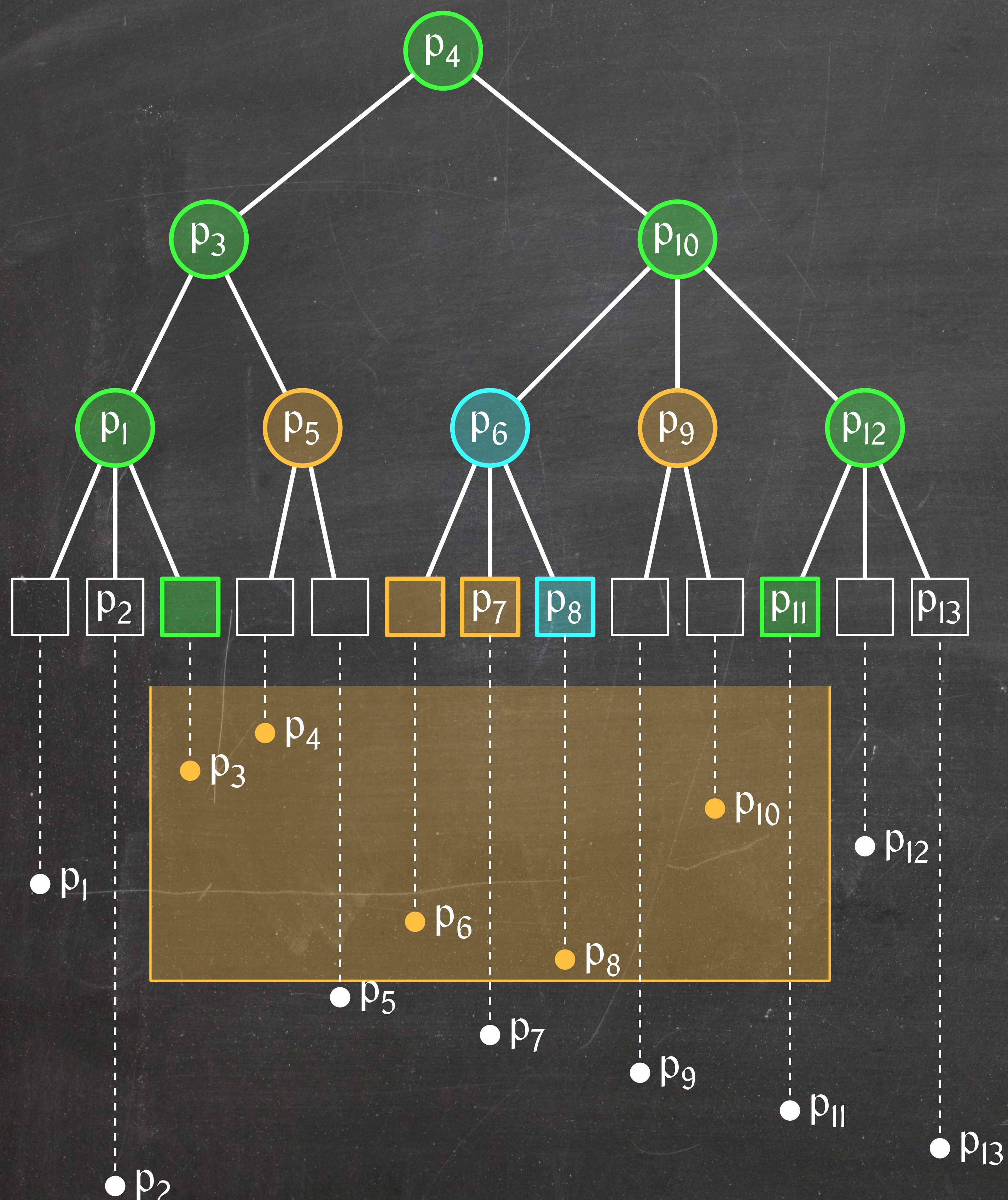


Priority search tree:

- Build a search tree on the x-coordinates.
- Propagate points up the tree to turn it into a max-heap.

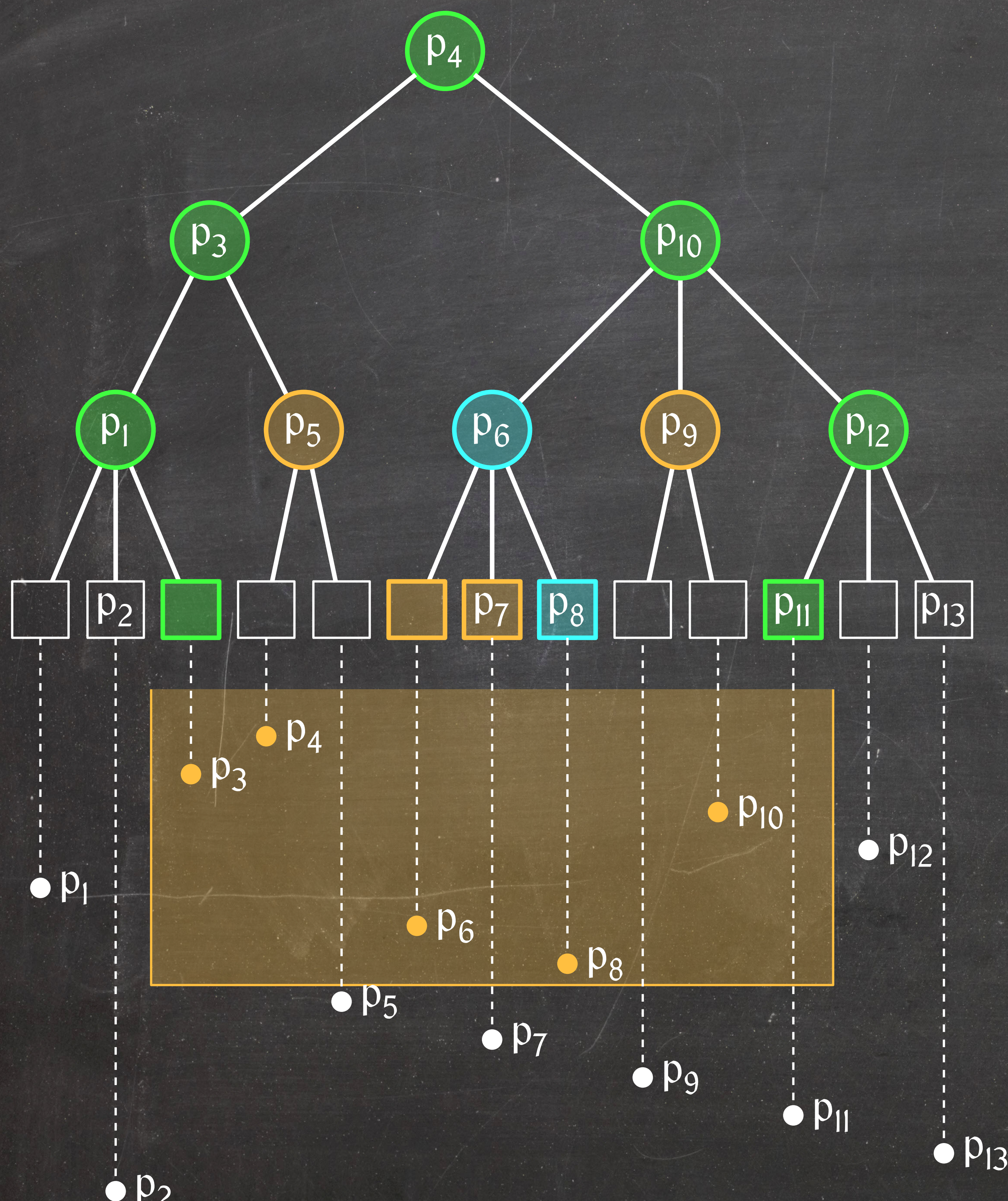
Note: We can still search for any point. It's now stored somewhere along the path to its corresponding leaf.

Three-Sided Range Reporting Queries



For every node on the two bounding paths (green), check whether its point needs to be reported.

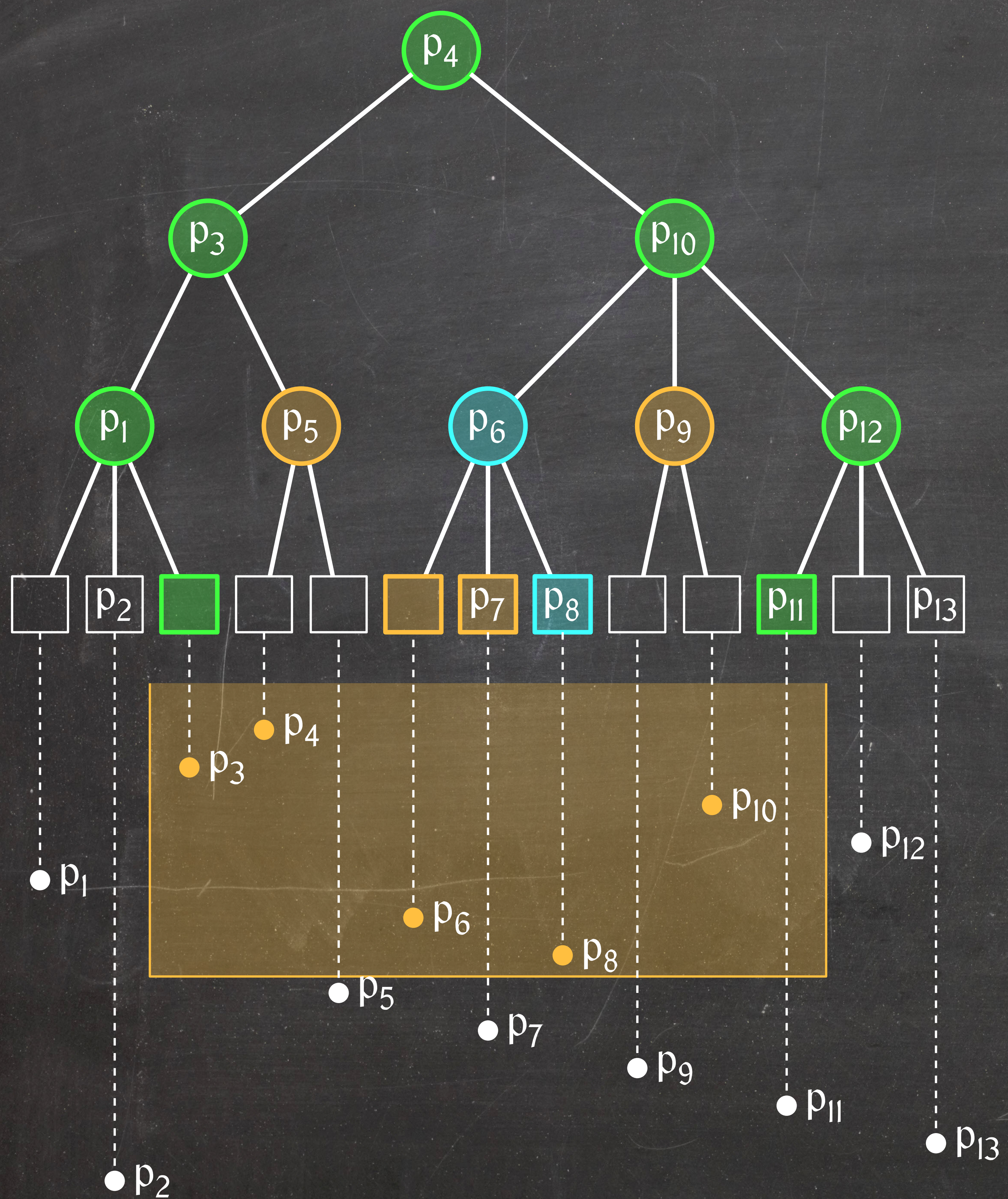
Three-Sided Range Reporting Queries



For every node on the two bounding paths (green), check whether its point needs to be reported.

These points may be in the range, outside the y-range or outside the x-range.

Three-Sided Range Reporting Queries

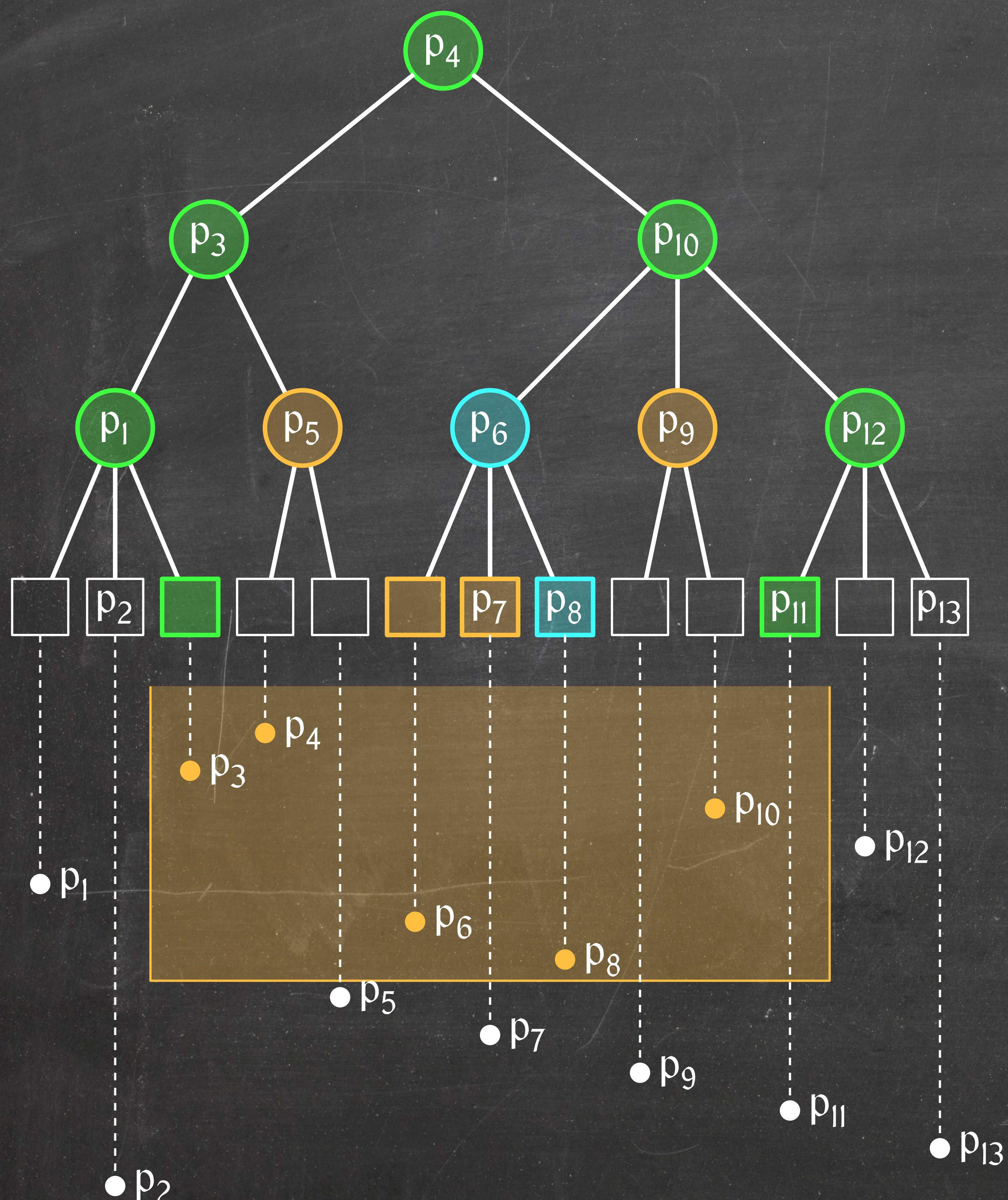


For every node on the two bounding paths (green), check whether its point needs to be reported.

These points may be in the range, outside the y-range or outside the x-range.

The points between the two bounding paths are all in the x-range.

Three-Sided Range Reporting Queries



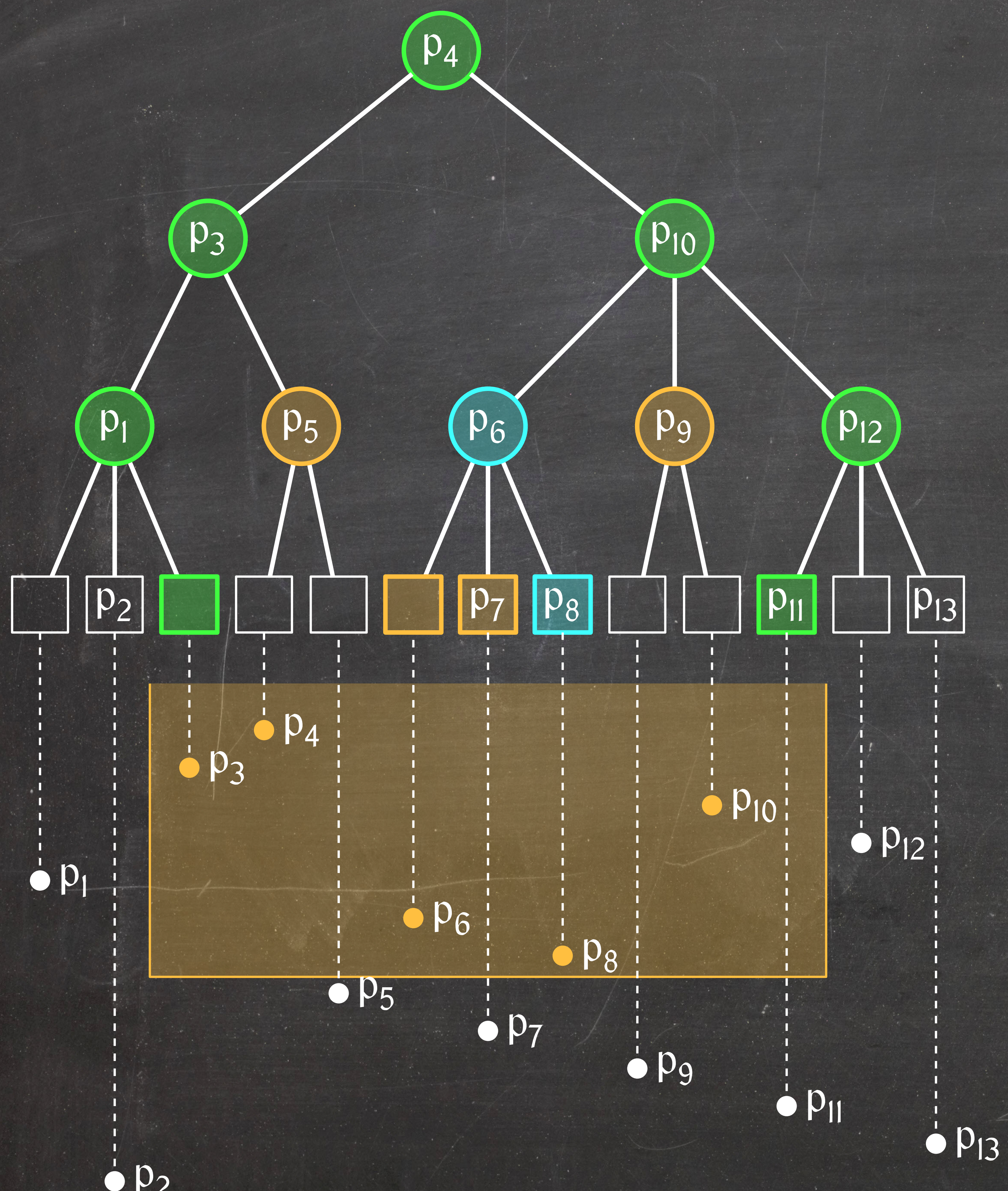
For every node on the two bounding paths (green), check whether its point needs to be reported.

These points may be in the range, outside the y-range or outside the x-range.

The points between the two bounding paths are all in the x-range.

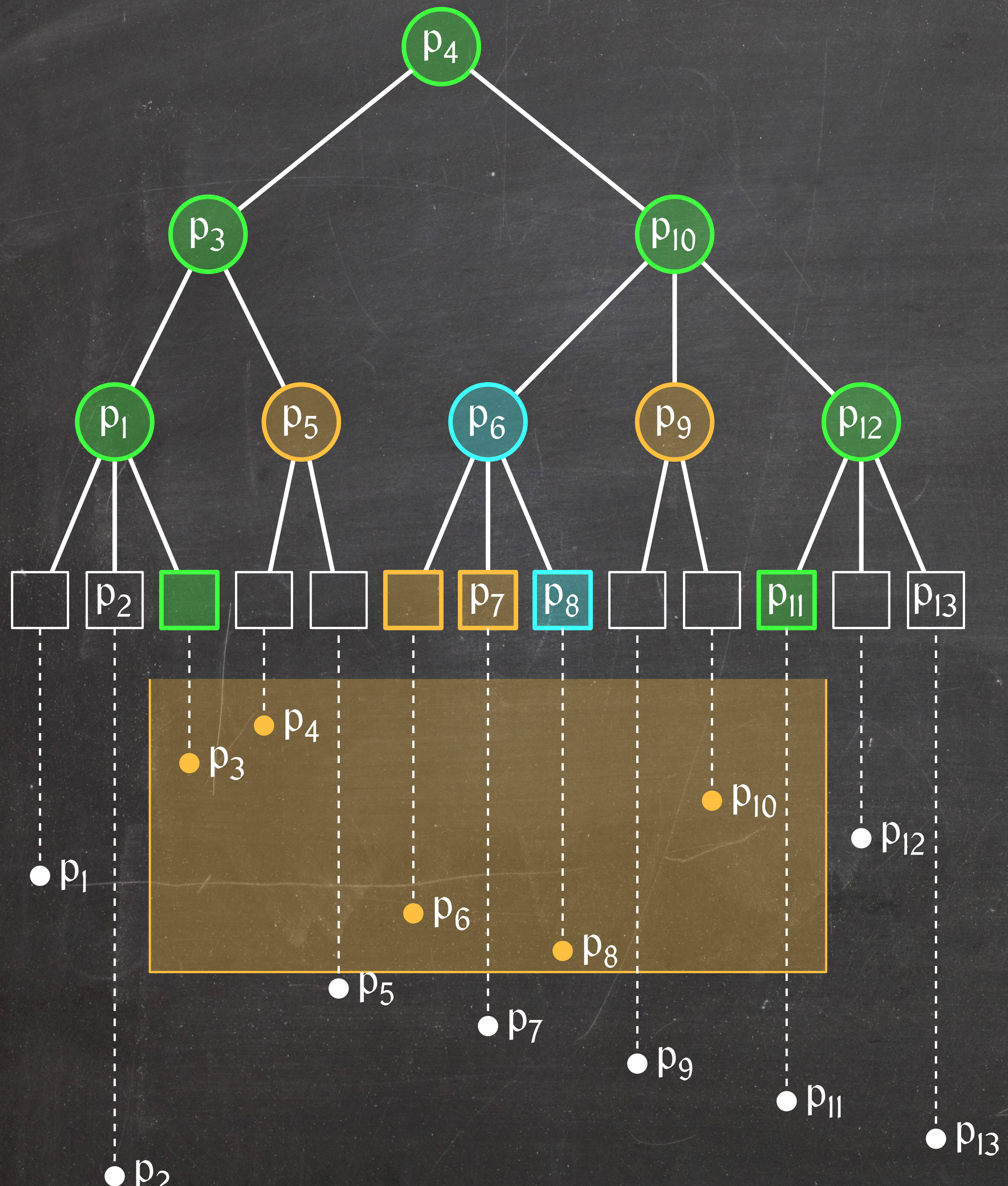
Use the $O(1 + k)$ procedure for heaps to report the points above the bottom y-coordinate.

Three-Sided Range Reporting Queries



O(lg n) green nodes

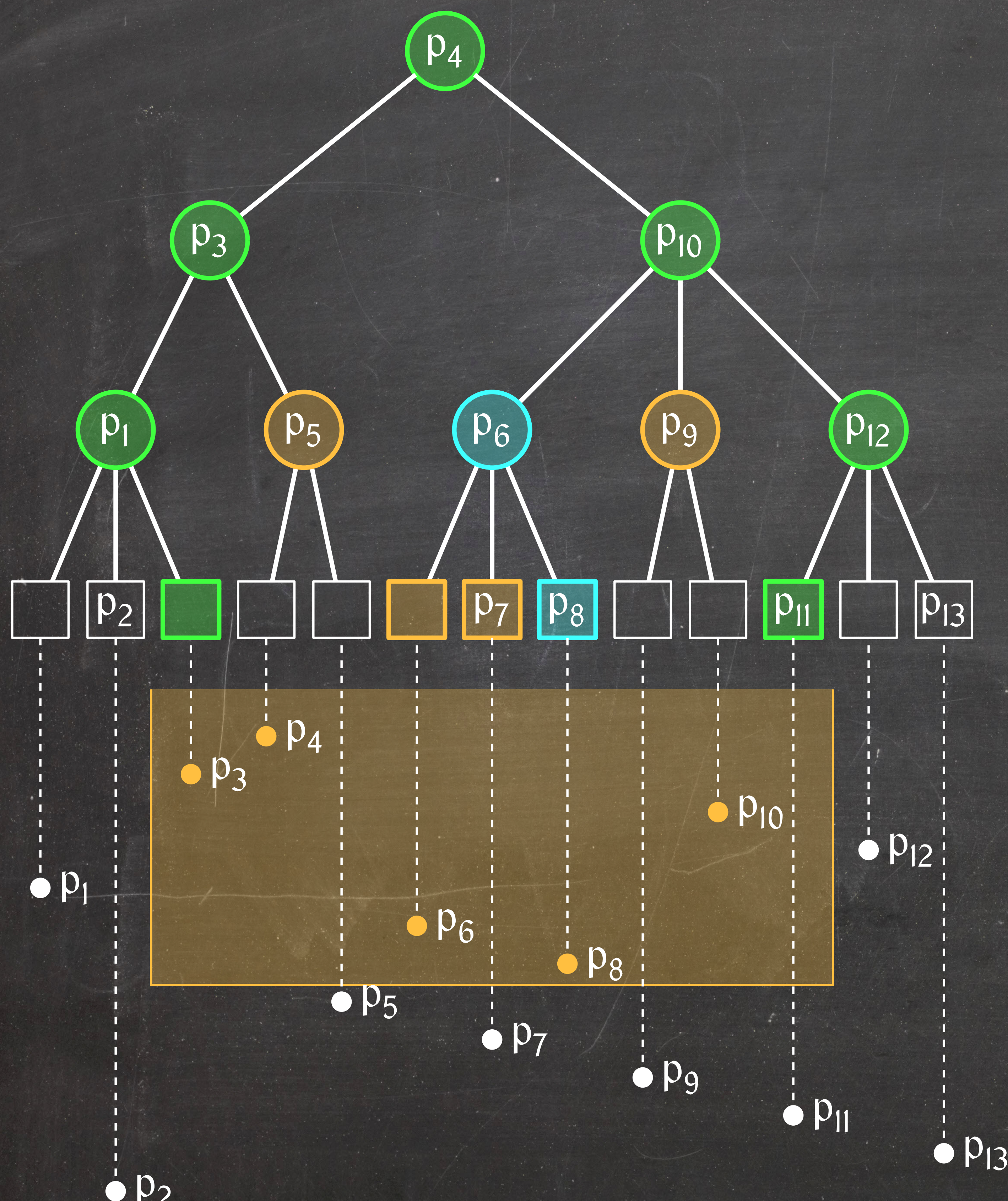
Three-Sided Range Reporting Queries



O(lg n) green nodes

O(b lg n) = O(lg n) children of green nodes

Three-Sided Range Reporting Queries

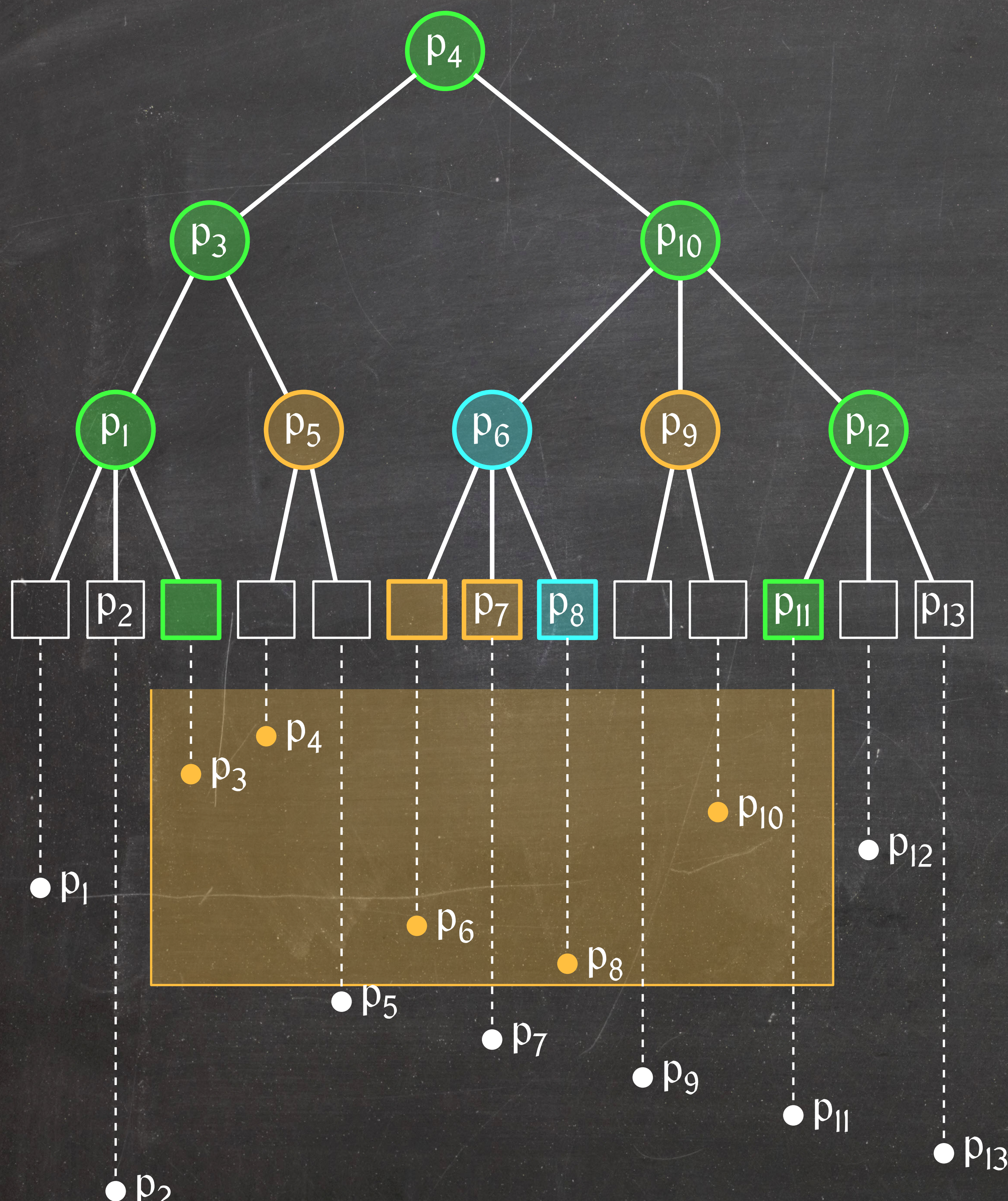


$O(\lg n)$ green nodes

$O(b \lg n) = O(\lg n)$ children of green nodes

For each child v between the two green paths, we spend $O(1 + k_v)$ time, where k_v is the number of points in its subtree we report.

Three-Sided Range Reporting Queries



$O(\lg n)$ green nodes

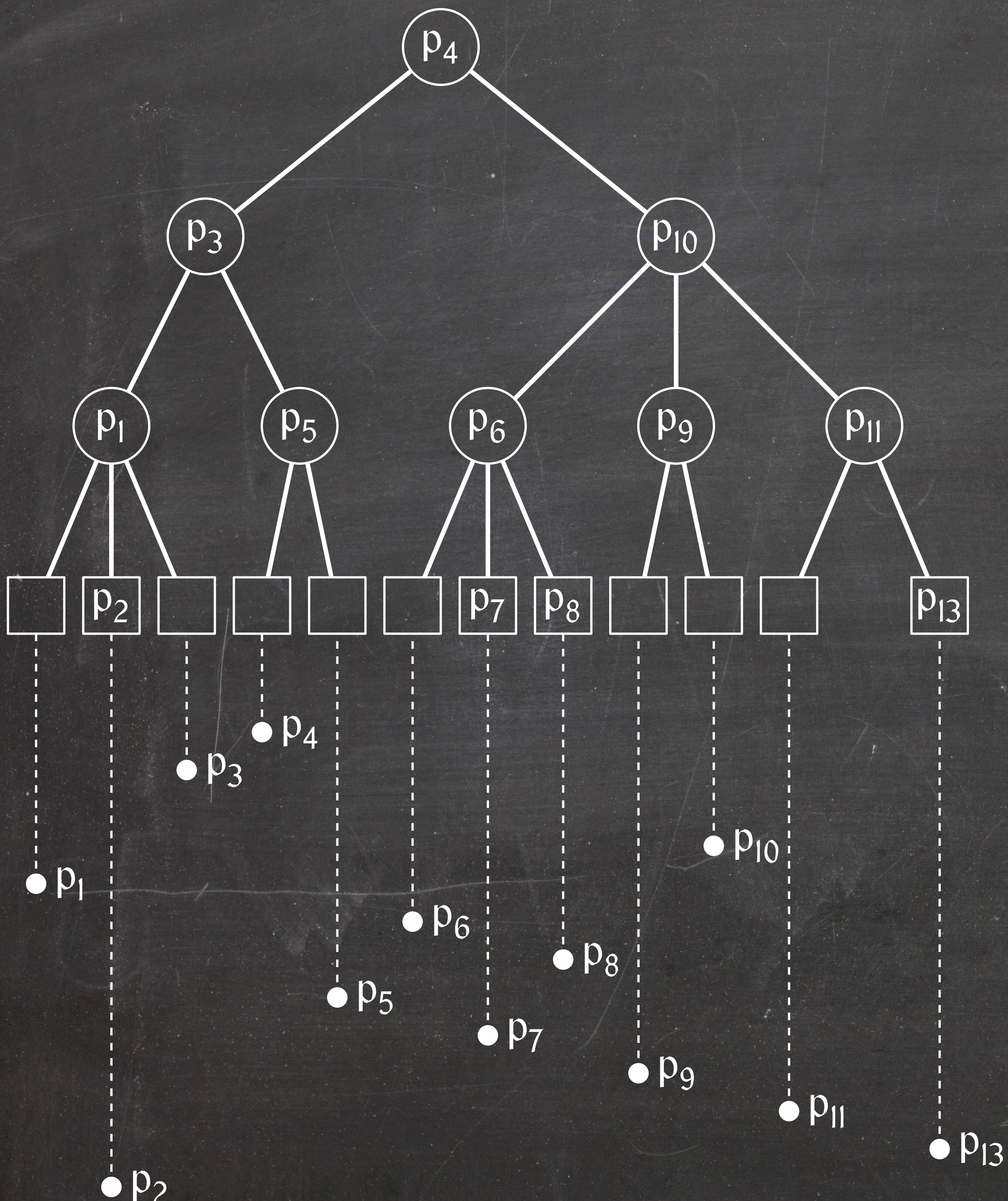
$O(b \lg n) = O(\lg n)$ children of green nodes

For each child v between the two green paths, we spend $O(1 + k_v)$ time, where k_v is the number of points in its subtree we report.

Total cost:

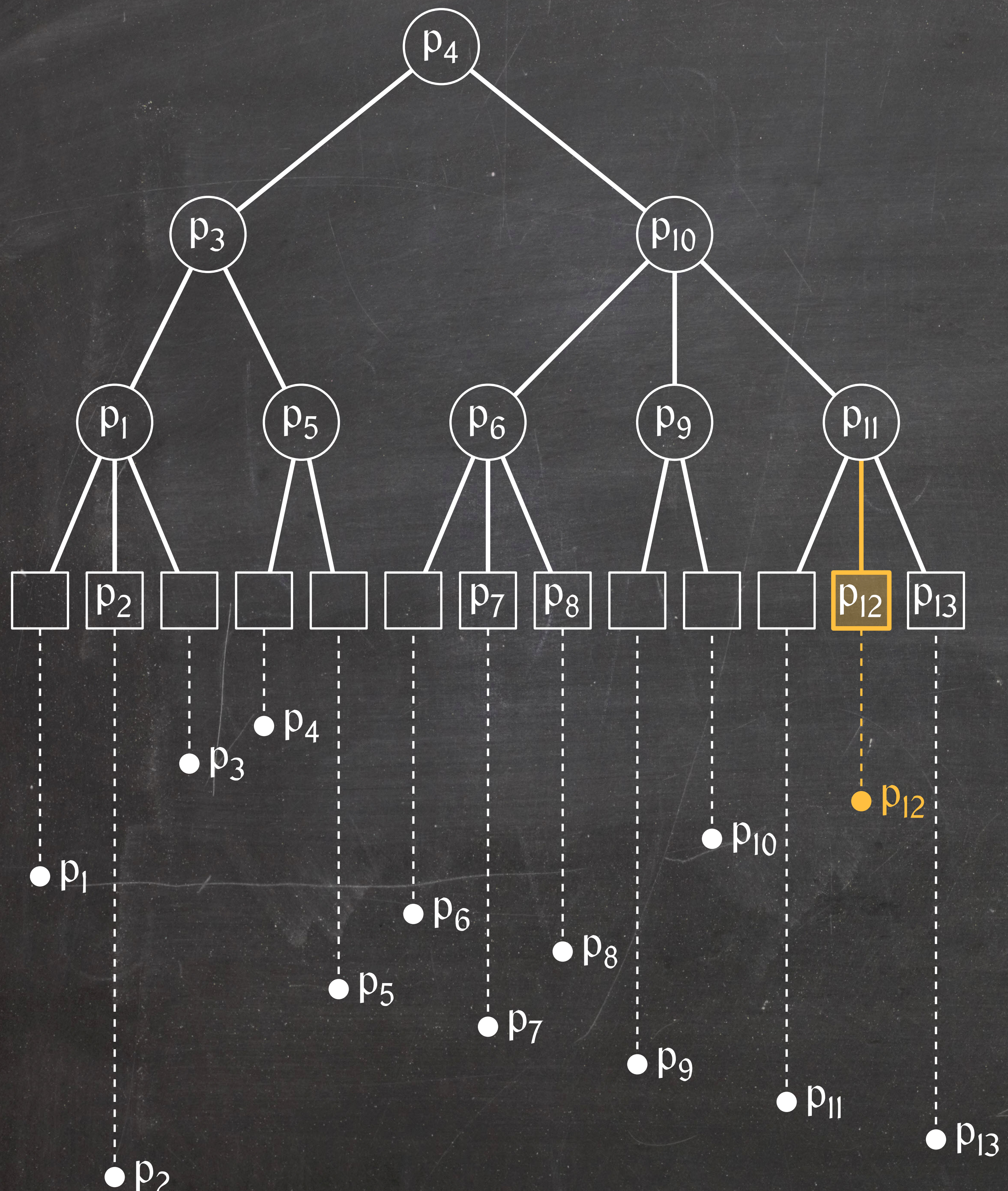
$O(\lg n) + \sum_v O(k_v) = O(\lg n + k)$

Insertions



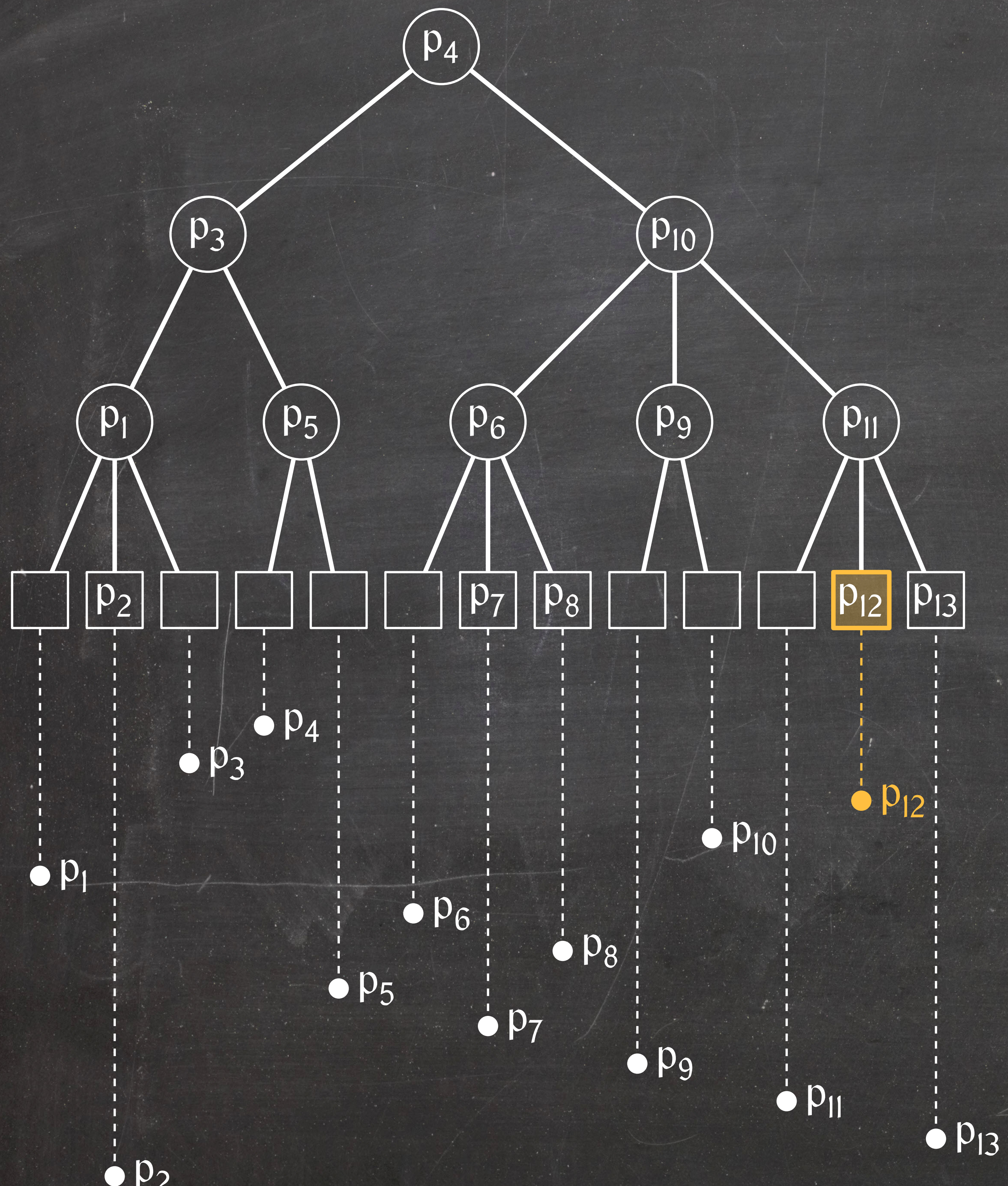
p4
p3
p10
p1
p5
p6
p9
p11
p2
p7
p8
p13
p4
p3
p1
p6
p5
p7
p8
p9
p10
p11
p13
p2

Insertions



Insert new point p as into a standard (a, b)-tree.

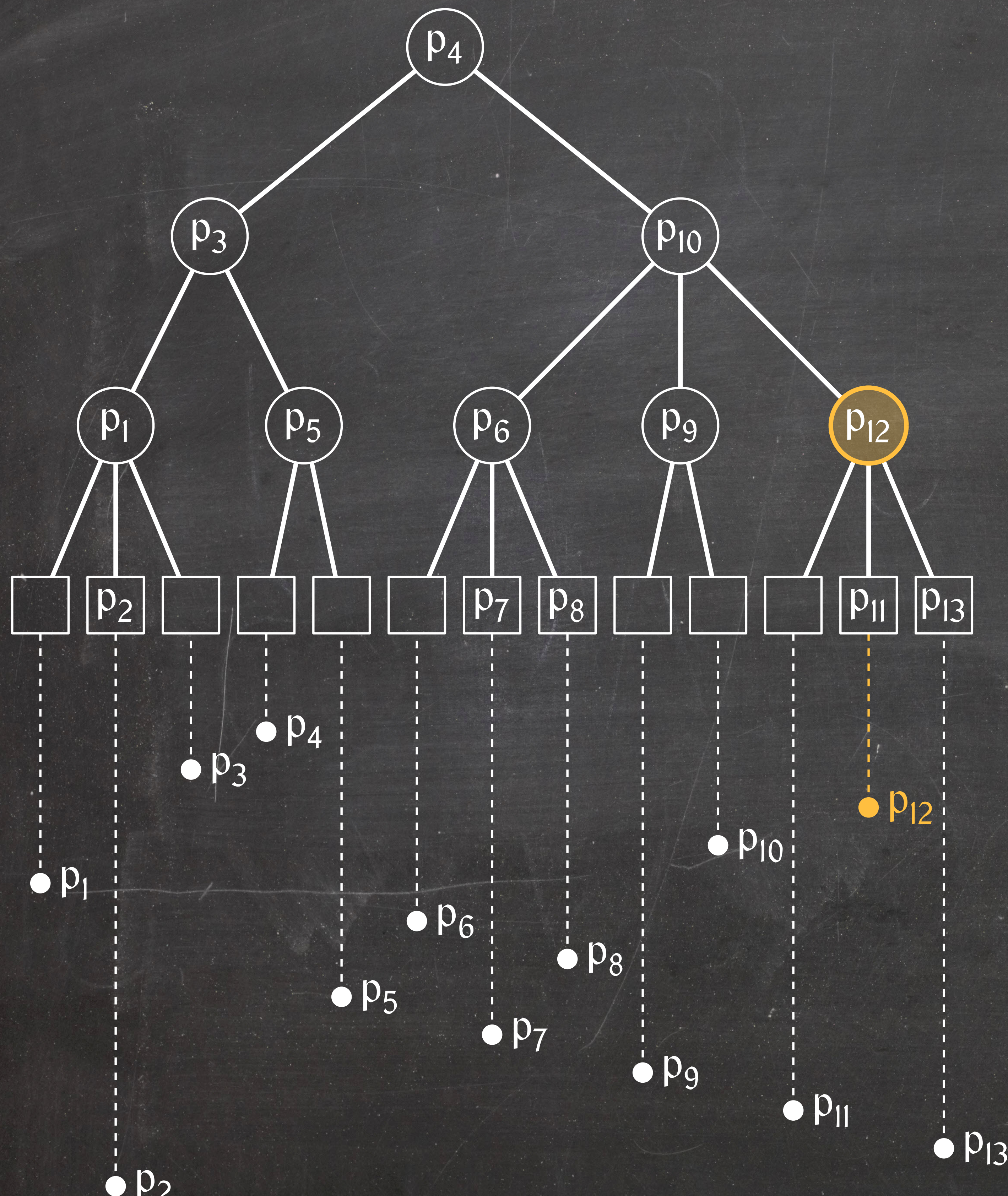
Insertions



Insert new point p as into a standard (a, b)-tree.

Heapify up as in a binary heap to restore heap order.

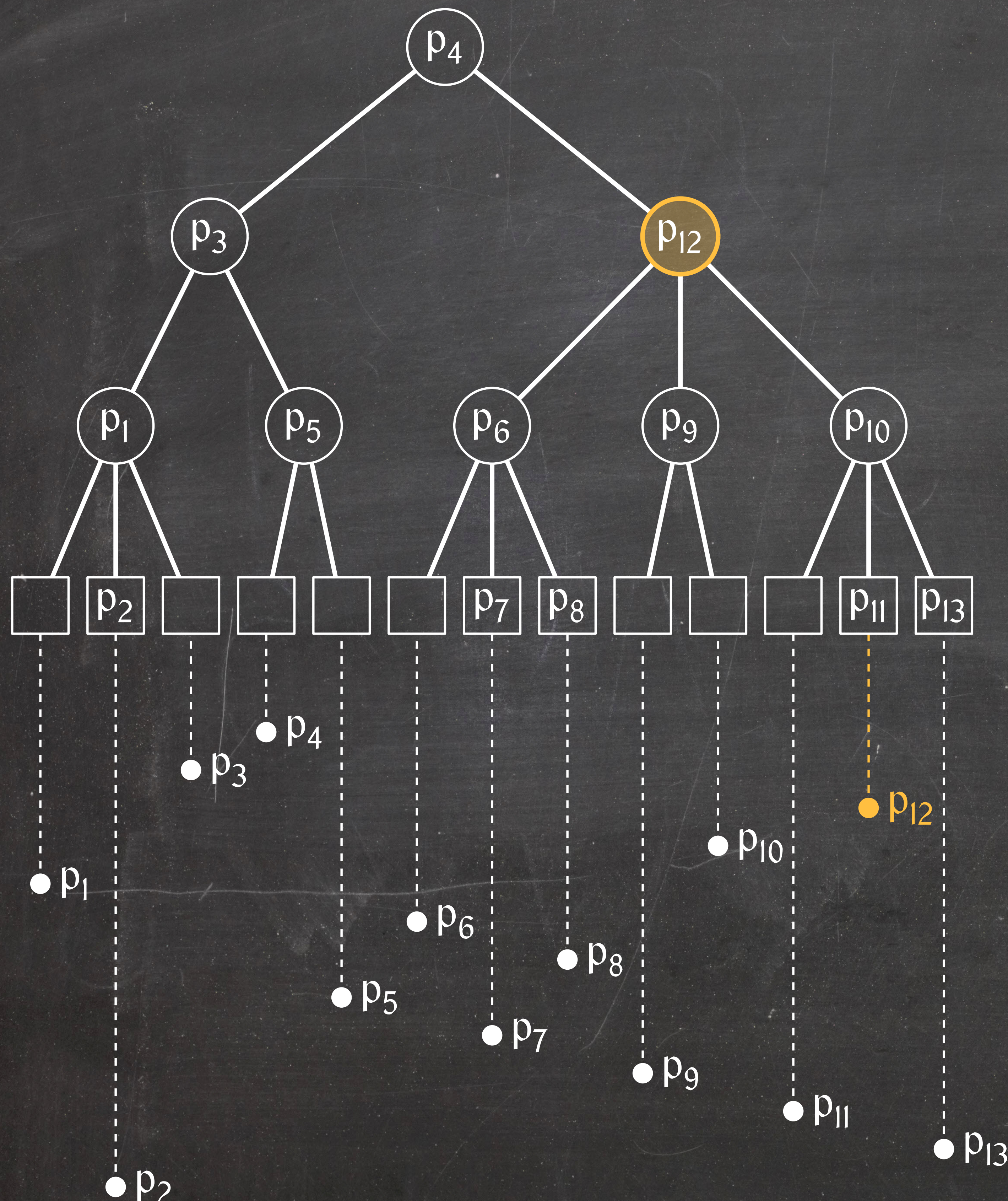
Insertions



Insert new point p as into a standard (a, b)-tree.

Heapify up as in a binary heap to restore heap order.

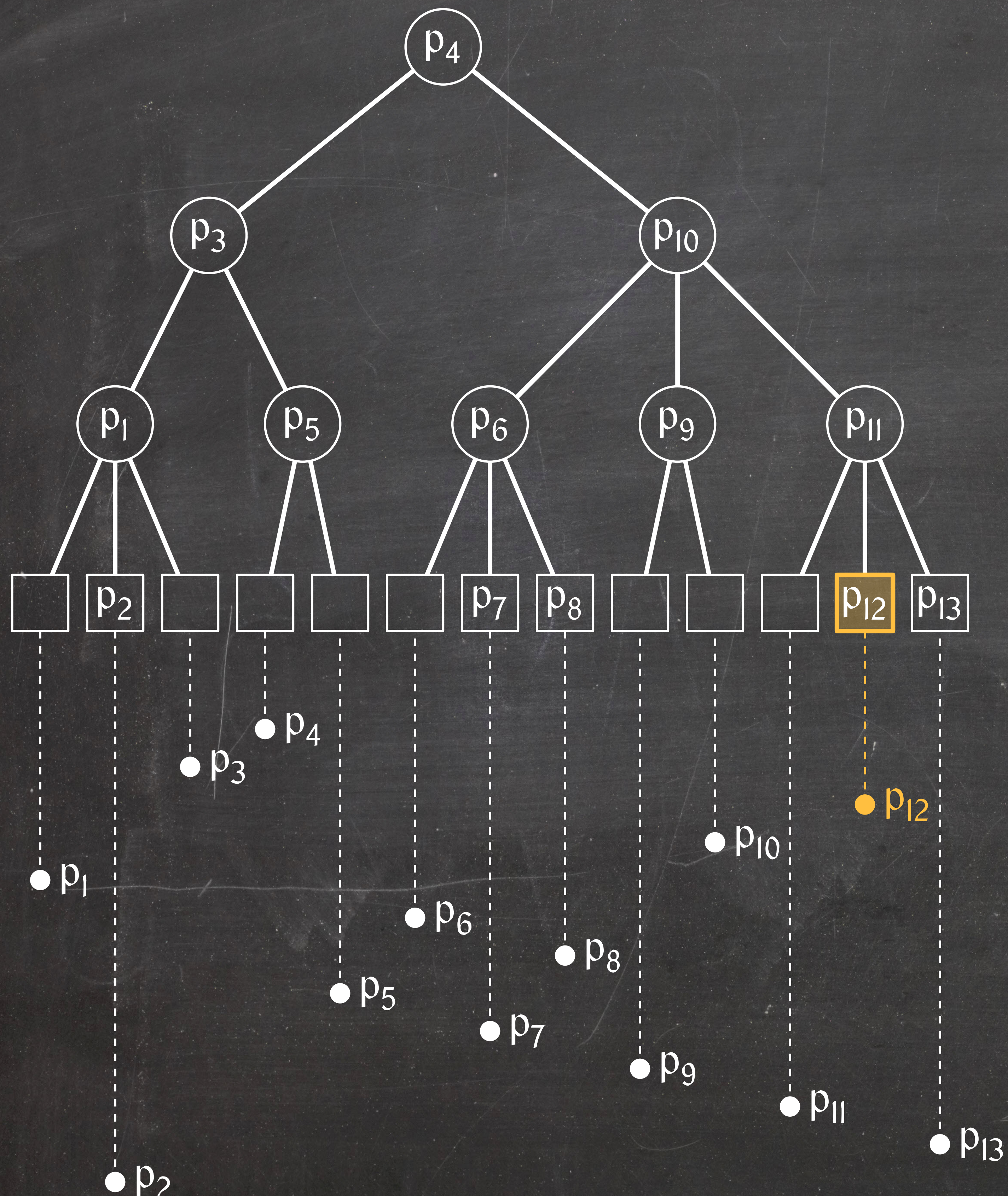
Insertions



Insert new point p as into a standard (a, b)-tree.

Heapify up as in a binary heap to restore heap order.

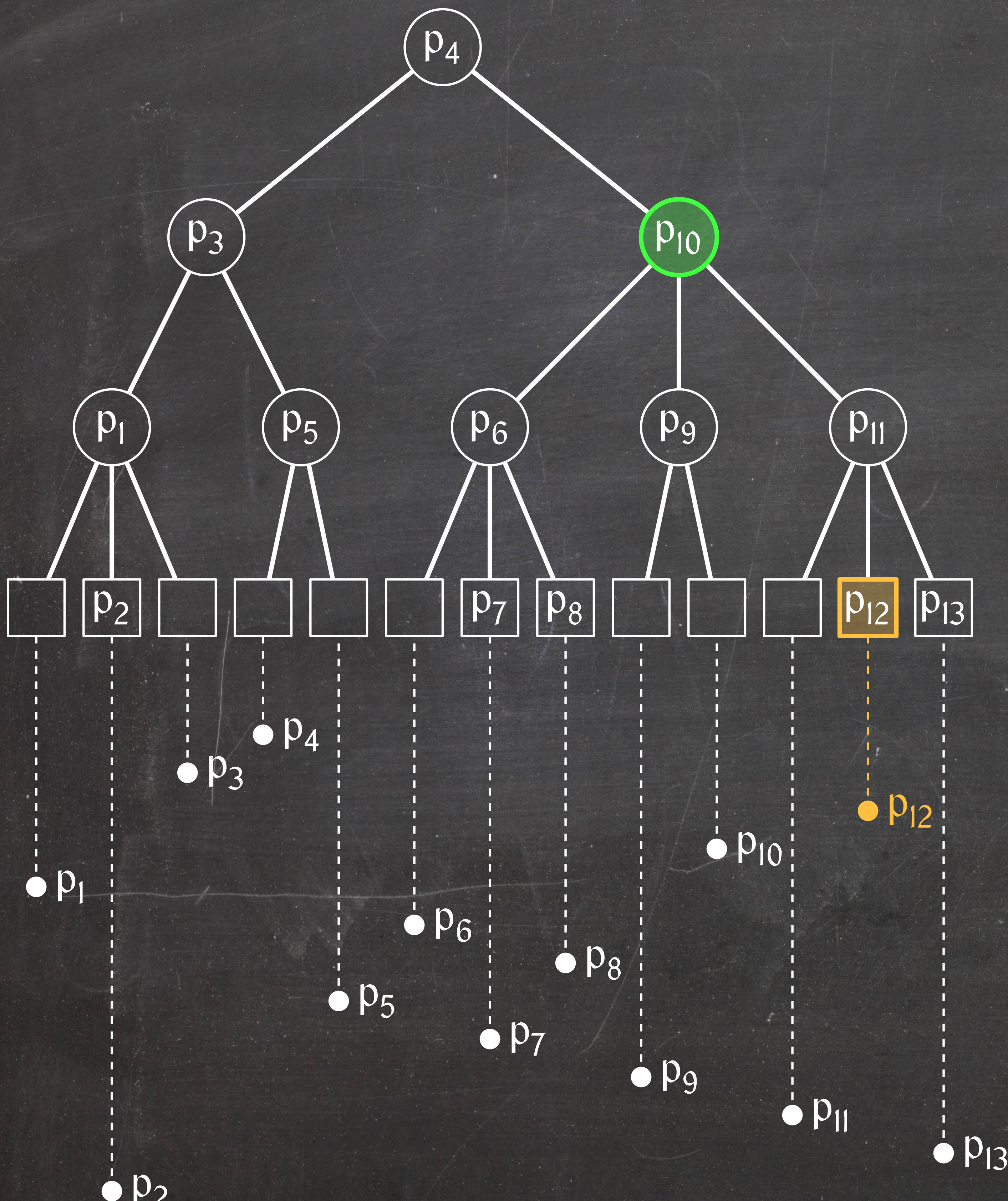
Insertions



Insert new point p as into a standard (a, b)-tree.

~~Heapify up as in a binary heap to restore heap order.~~

Insertions

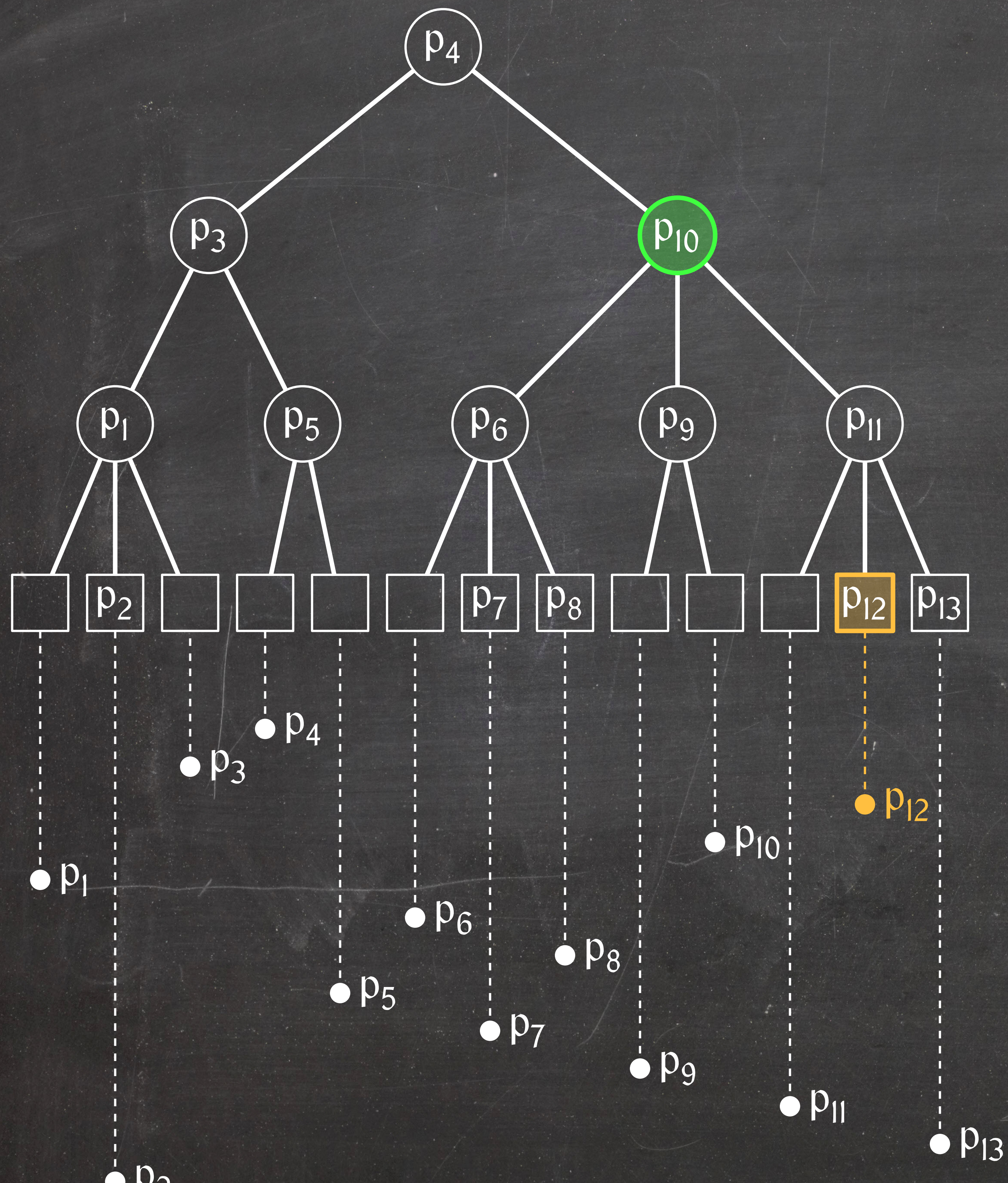


Insert new point p as into a standard (a, b)-tree.

~~Heapify up as in a binary heap to restore heap order.~~

Locate the lowest ancestor whose parent does not store a point lower than p.

Insertions



Insert new point p as into a standard (a, b)-tree.

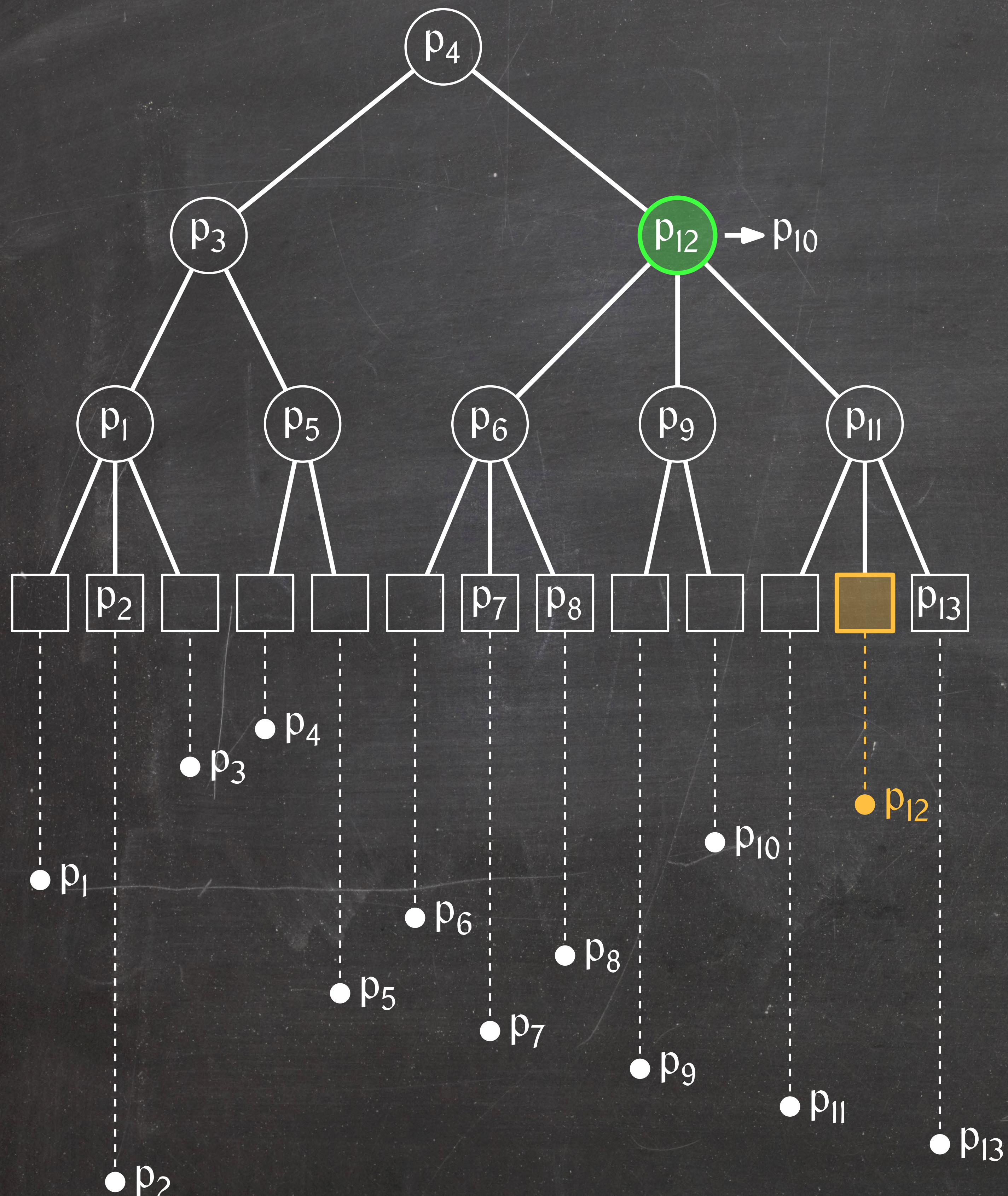
~~Heapify up as in a binary heap to restore heap order.~~

Locate the lowest ancestor whose parent does not store a point lower than p.

While p ≠ nil:

- Replace point q at current node with p.
- p = q
- Move to child of current node that is an ancestor of p's leaf.

Insertions



Insert new point p as into a standard (a, b)-tree.

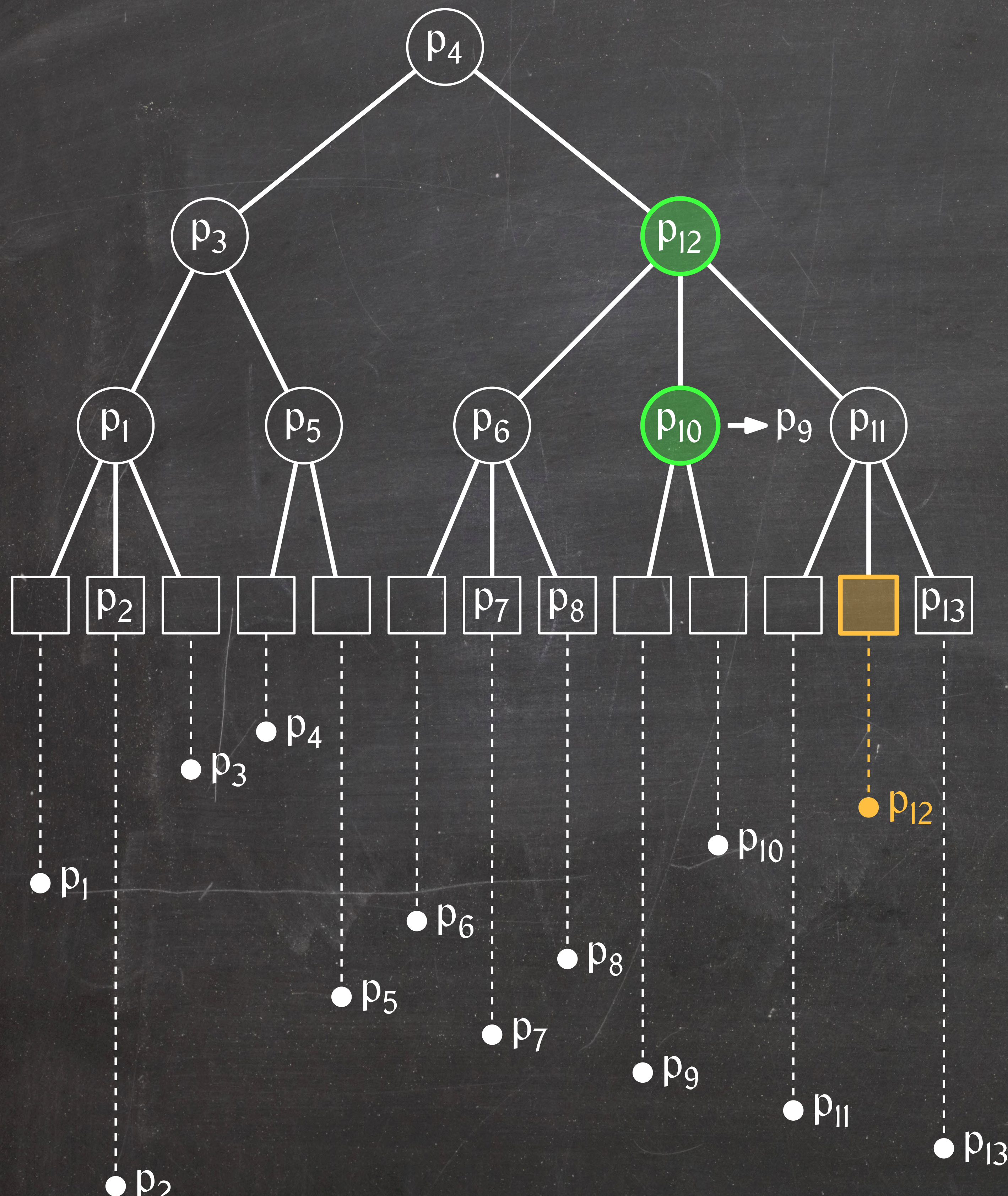
~~Heapify up as in a binary heap to restore heap order.~~

Locate the lowest ancestor whose parent does not store a point lower than p.

While p ≠ nil:

- Replace point q at current node with p.
- p = q
- Move to child of current node that is an ancestor of p's leaf.

Insertions



Insert new point p as into a standard (a, b)-tree.

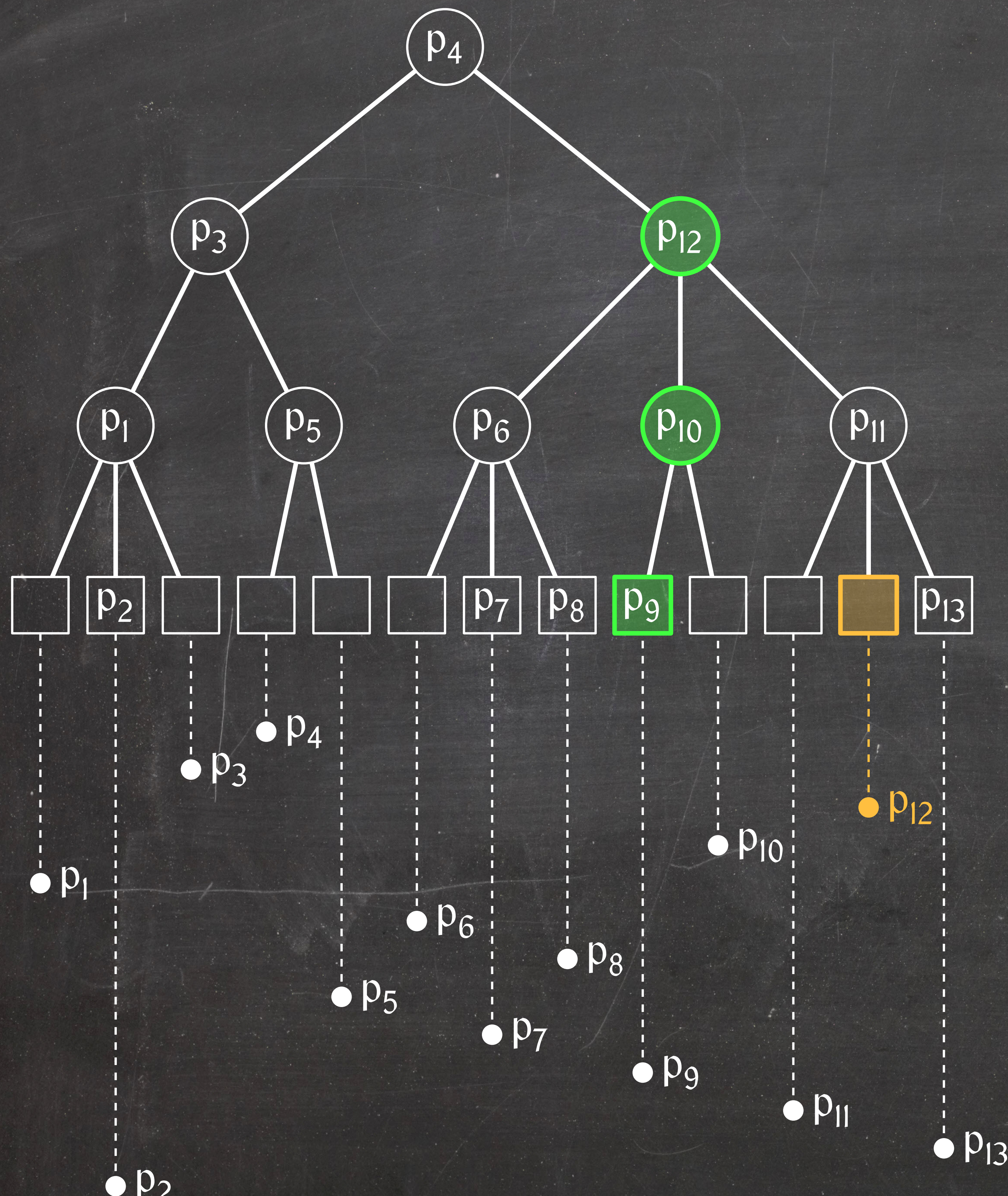
~~Heapify up as in a binary heap to restore heap order.~~

Locate the lowest ancestor whose parent does not store a point lower than p.

While p ≠ nil:

- Replace point q at current node with p.
- p = q
- Move to child of current node that is an ancestor of p's leaf.

Insertions



Insert new point p as into a standard (a, b)-tree.

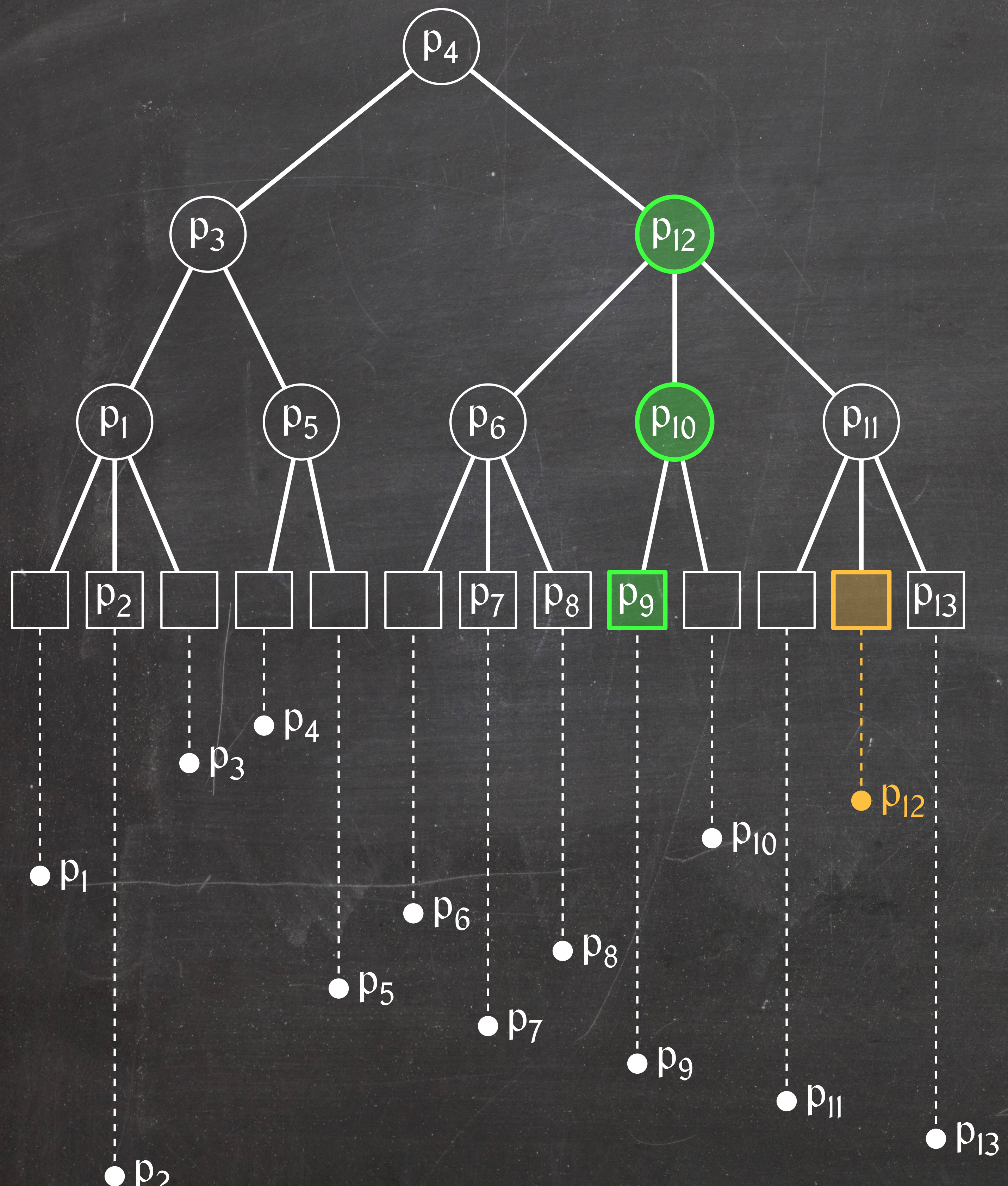
~~Heapify up as in a binary heap to restore heap order.~~

Locate the lowest ancestor whose parent does not store a point lower than p.

While p ≠ nil:

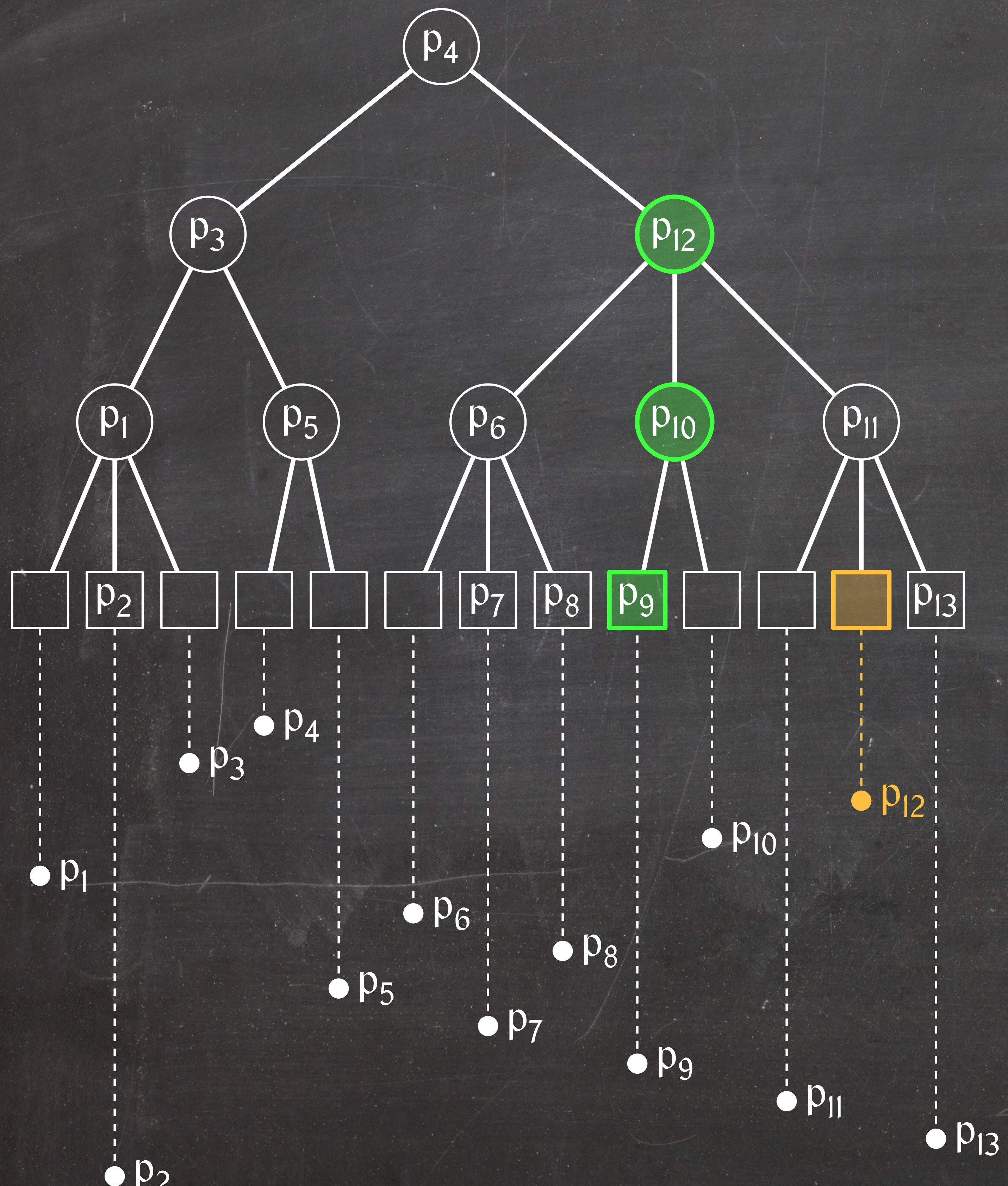
- Replace point q at current node with p.
- p = q
- Move to child of current node that is an ancestor of p's leaf.

Insertions



Inserting p takes $O(\lg n)$ time.

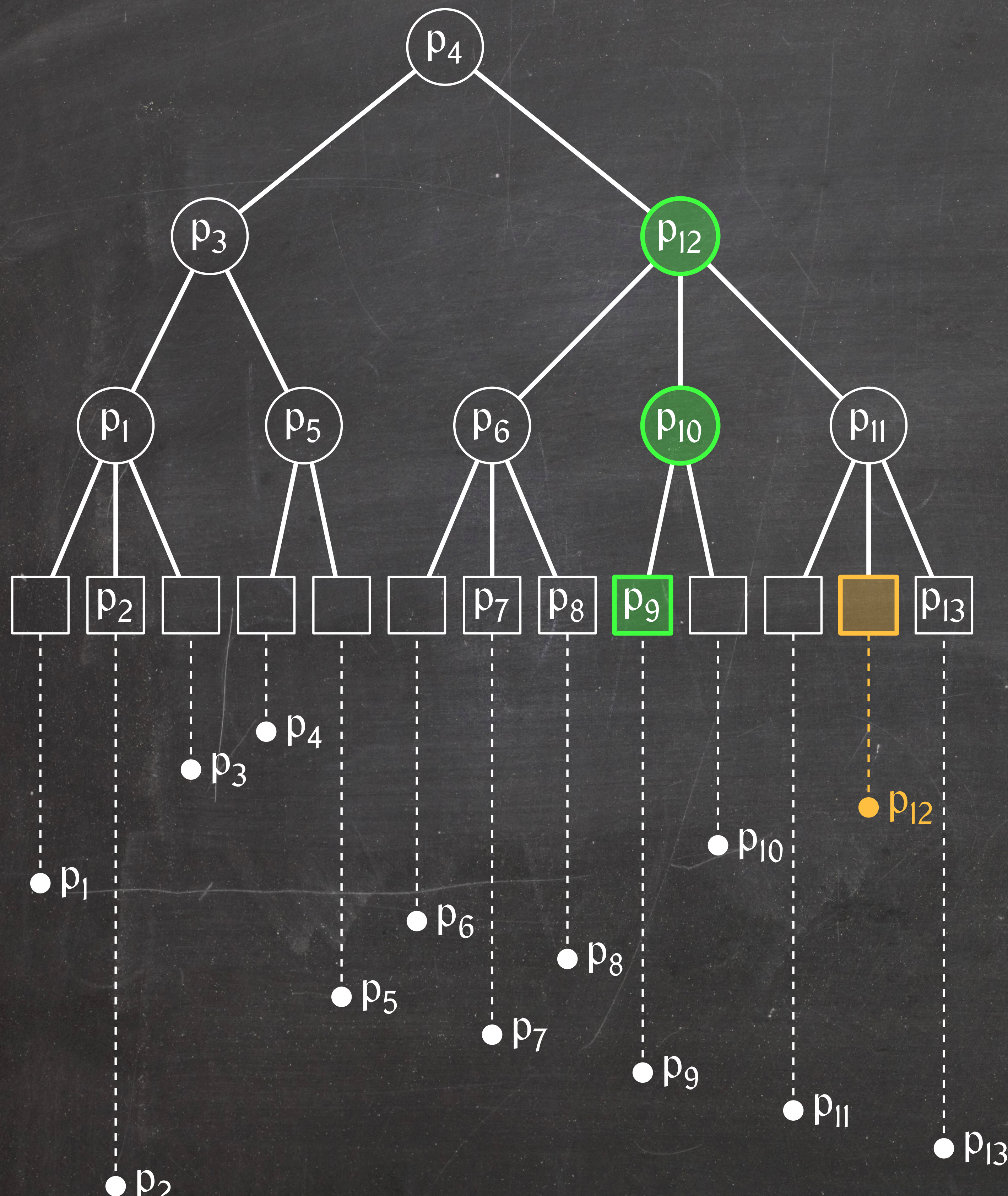
Insertions



Inserting p takes O(lg n) time.

Locating the ancestor where p is to be stored takes O(lg n) time.

Insertions

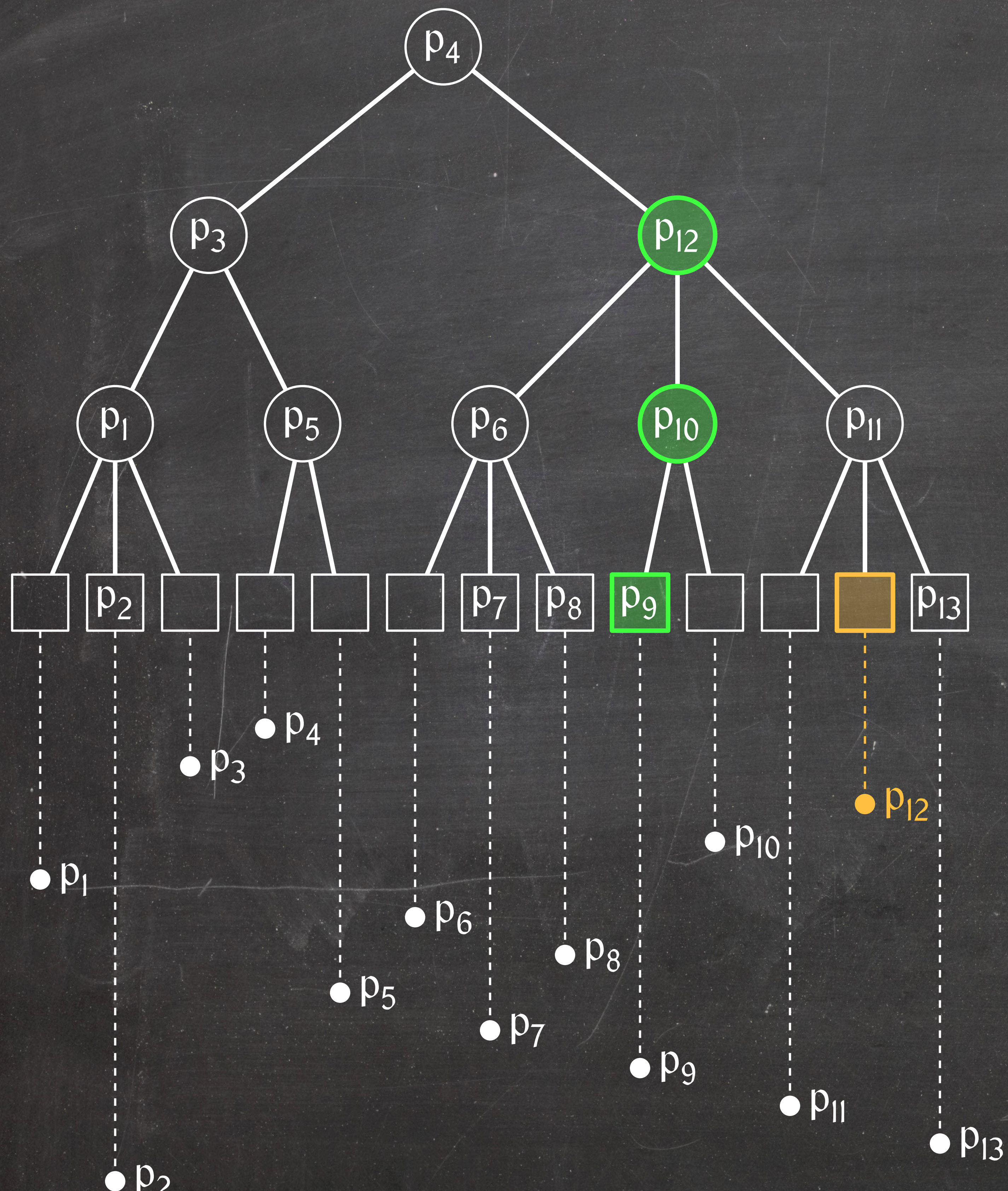


Inserting p takes O(lg n) time.

Locating the ancestor where p is to be stored takes O(lg n) time.

Evicting points and pushing them down the tree amounts to traversing a single top-down path. This also takes O(lg n) time.

Insertions



Inserting p takes O(lg n) time.

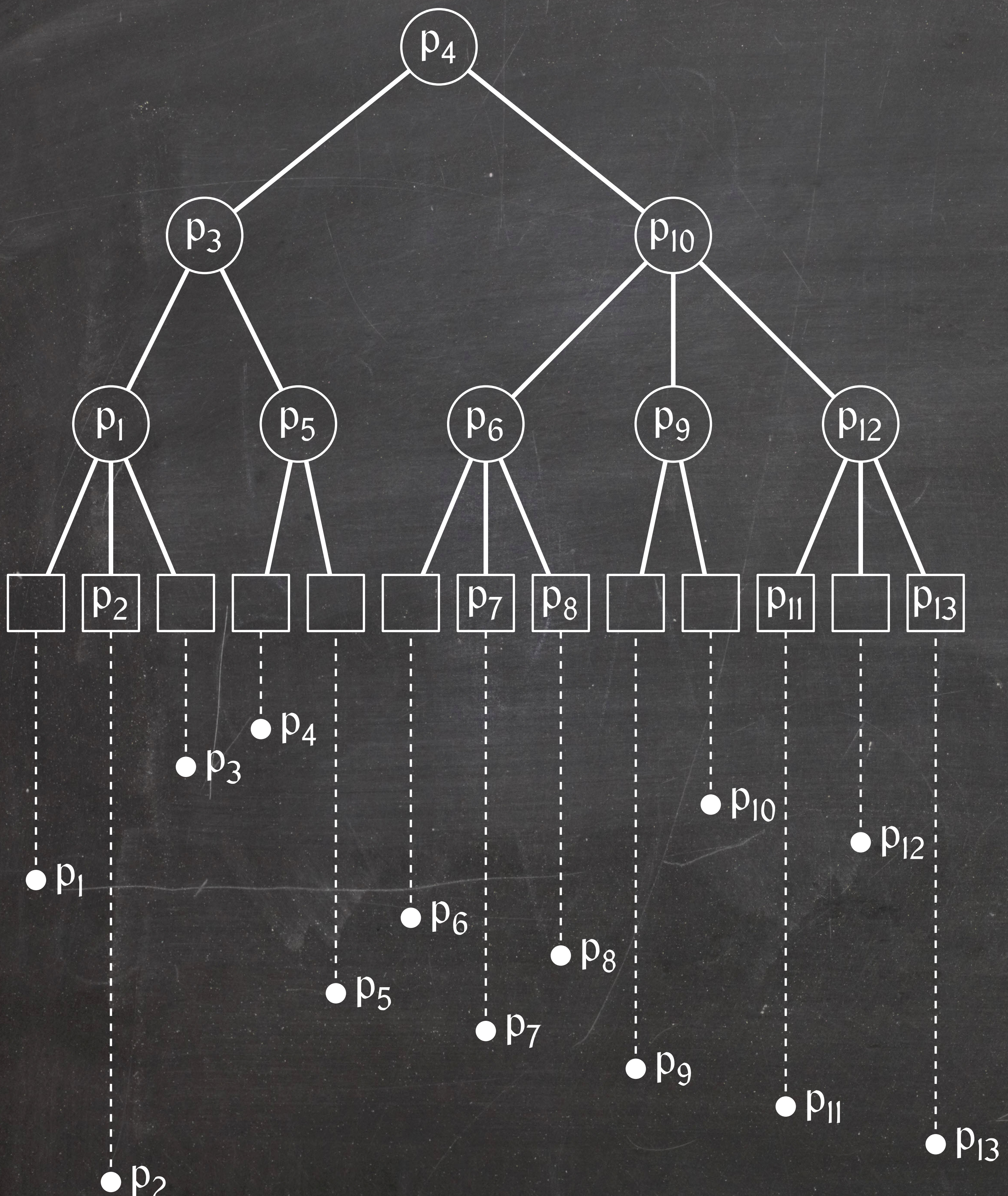
Locating the ancestor where p is to be stored takes O(lg n) time.

Evicting points and pushing them down the tree amounts to traversing a single top-down path. This also takes O(lg n) time.

Total cost:

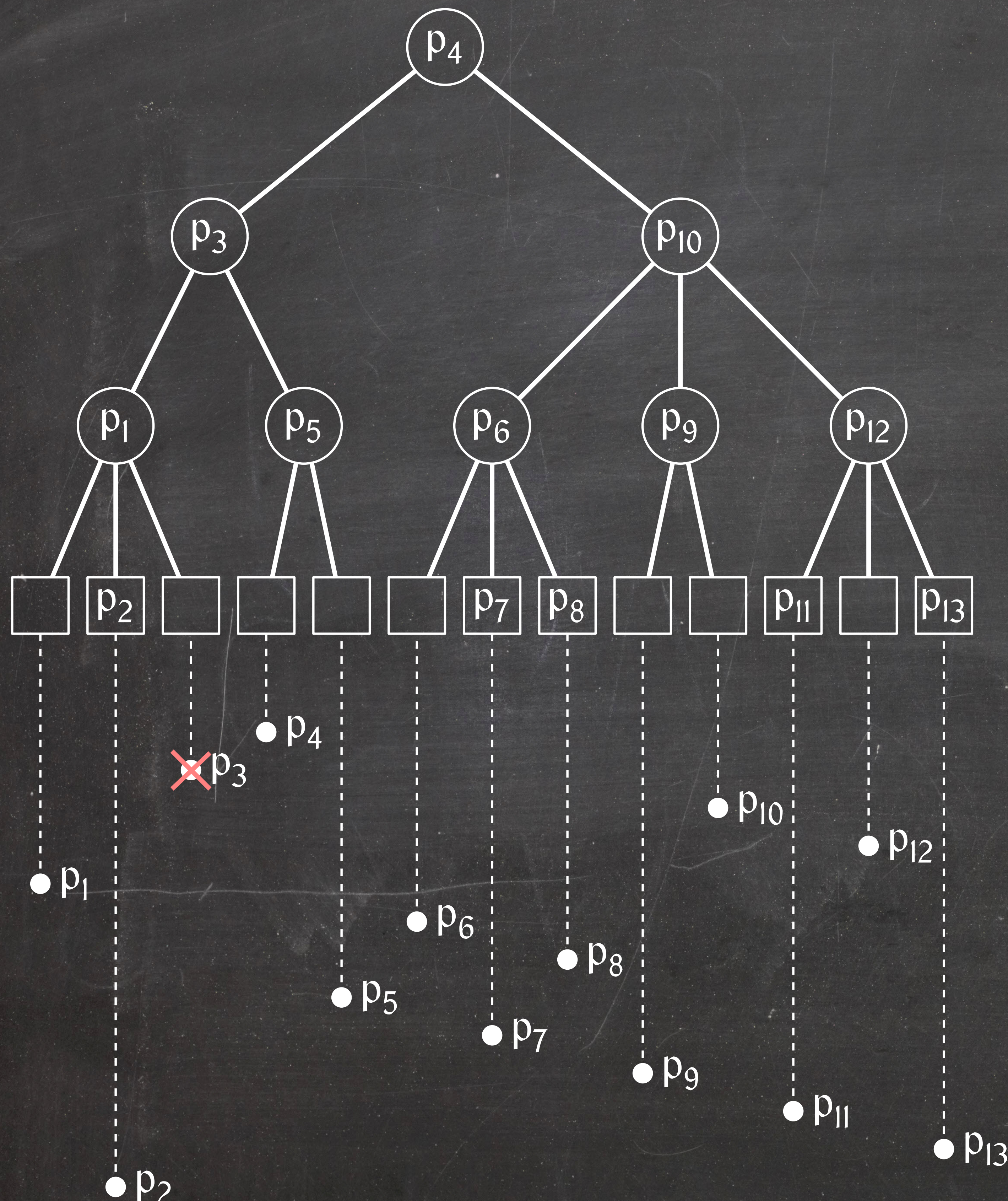
O(lg n) (excluding node splits)

Deletions



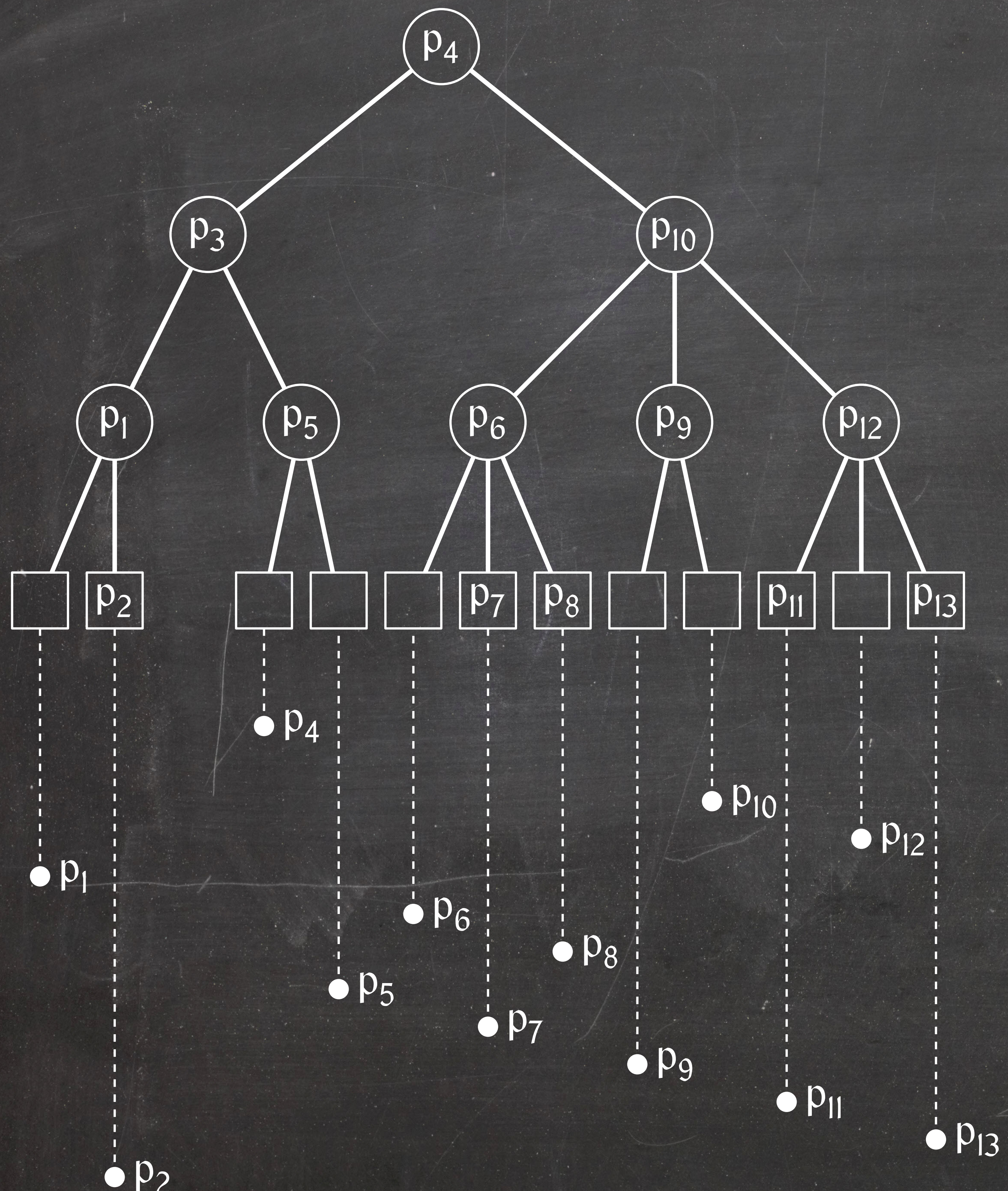
p_4
p_3
p_{10}
p_1
p_5
p_6
p_9
p_{12}
p_2
p_7
p_8
p_{11}
p_{13}
p_4
p_3
p_{10}
p_{12}
p_1
p_6
p_8
p_5
p_7
p_9
p_{11}
p_{13}
p_2

Deletions



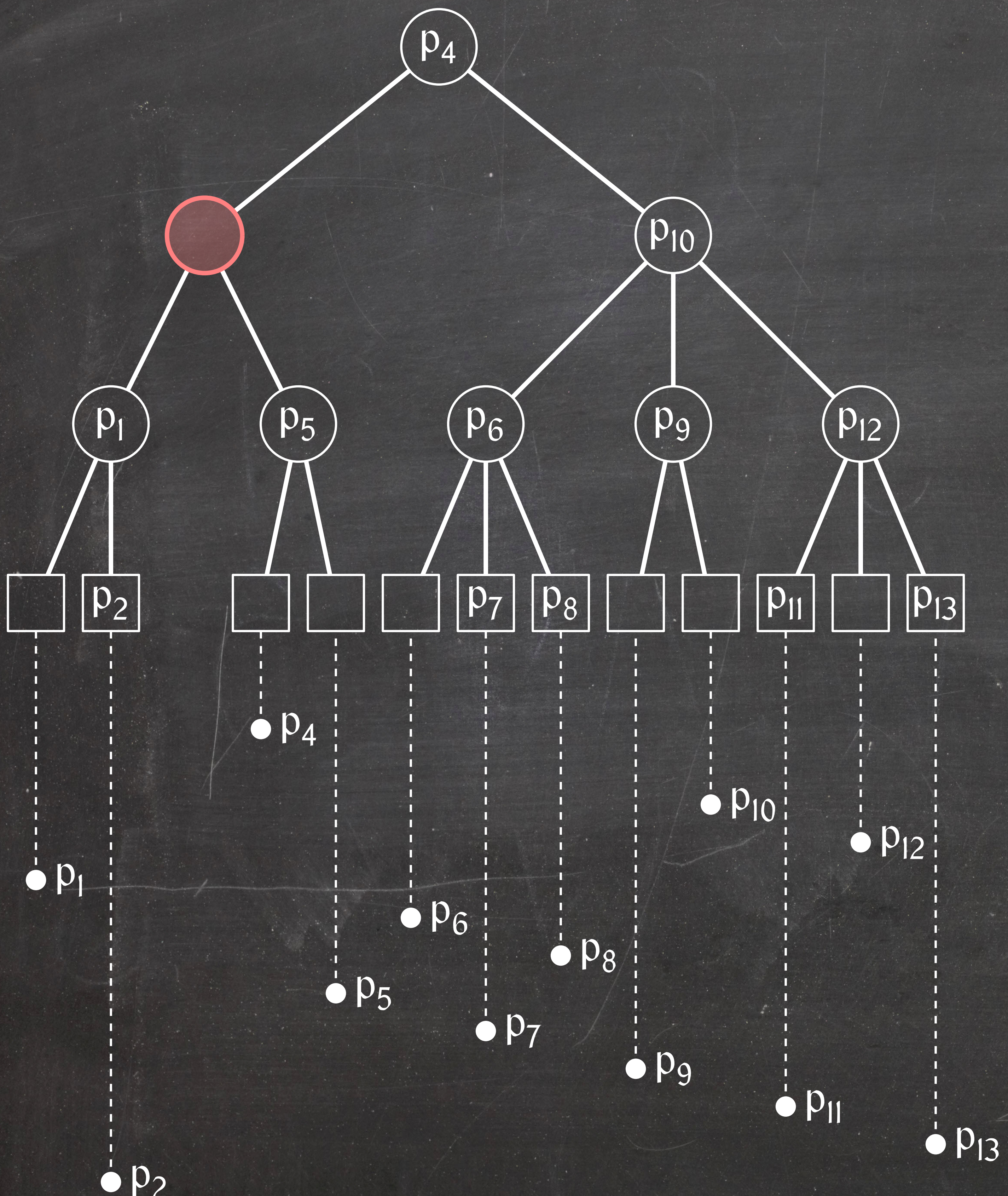
p4
p3
p10
p1
p5
p6
p9
p12
p2
p7
p8
p11
p13
p4
p3
p10
p12
p1
p6
p8
p5
p7
p9
p11
p13
p2

Deletions



Delete the leaf corresponding to p.

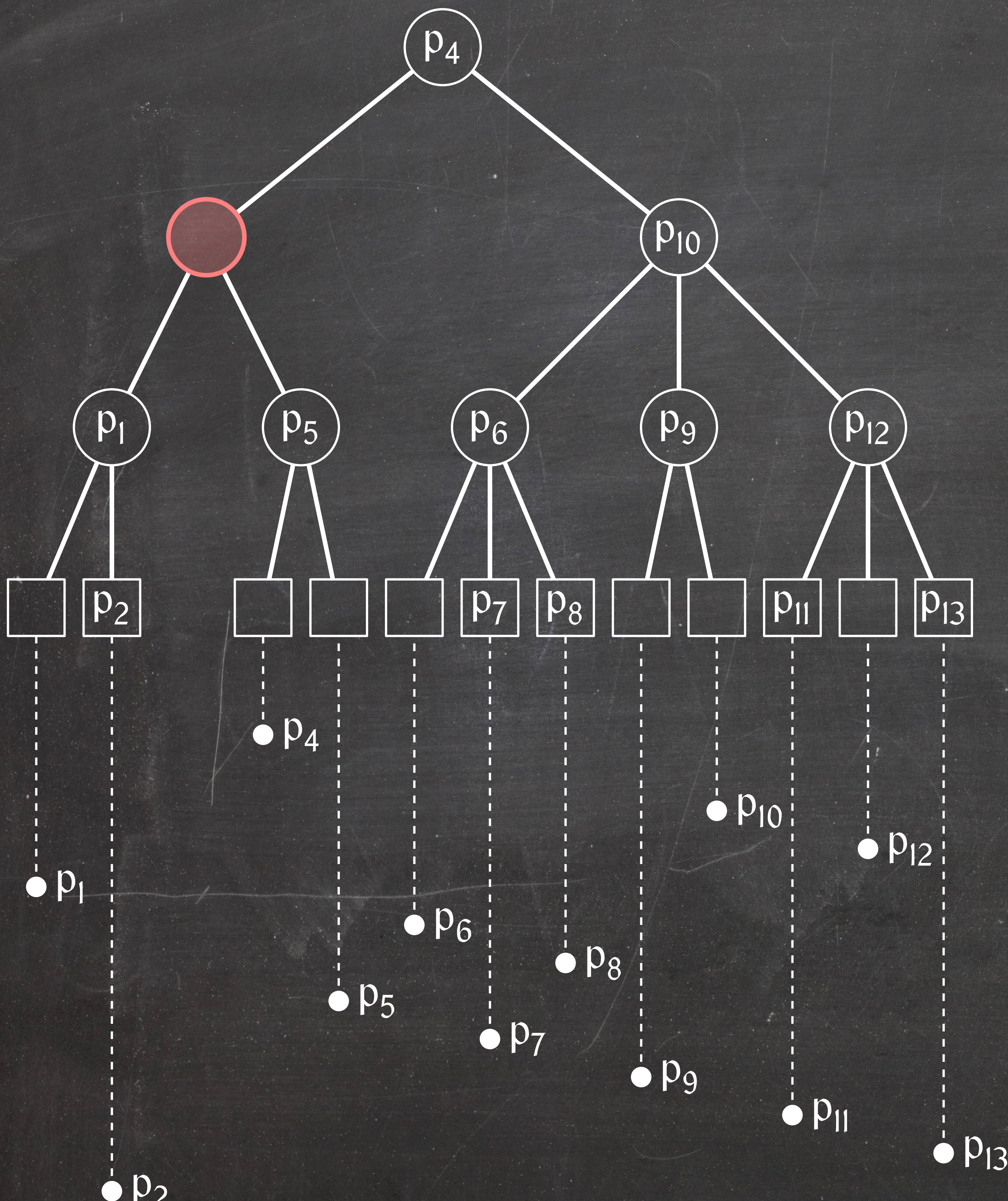
Deletions



Delete the leaf corresponding to p.

Delete p from the node where it is stored.

Deletions



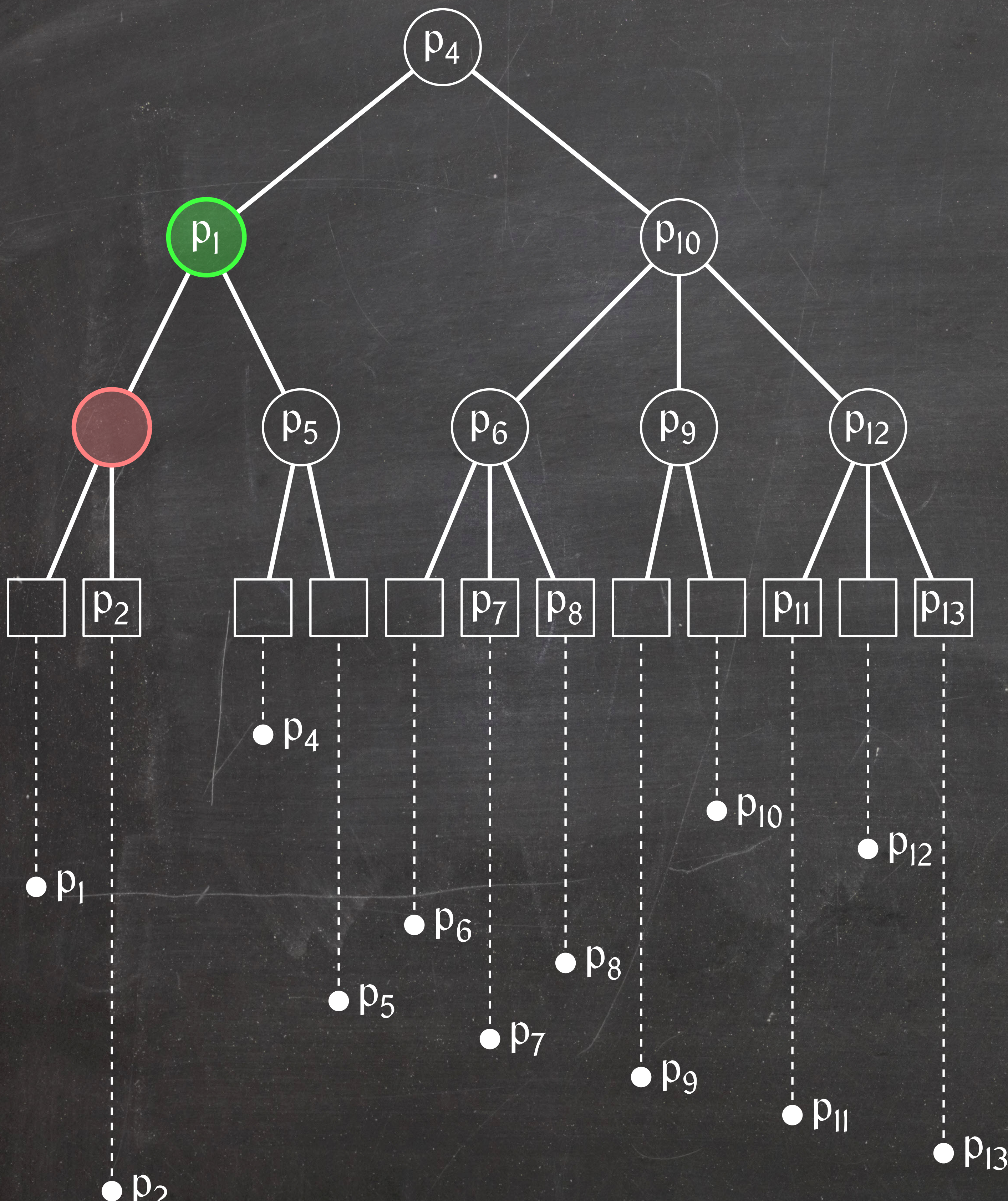
Delete the leaf corresponding to p.

Delete p from the node where it is stored.

While the current node v has a child that stores a point:

- Choose the child w whose point q has the highest y-coordinate.
- Store q at v.
- v = w

Deletions



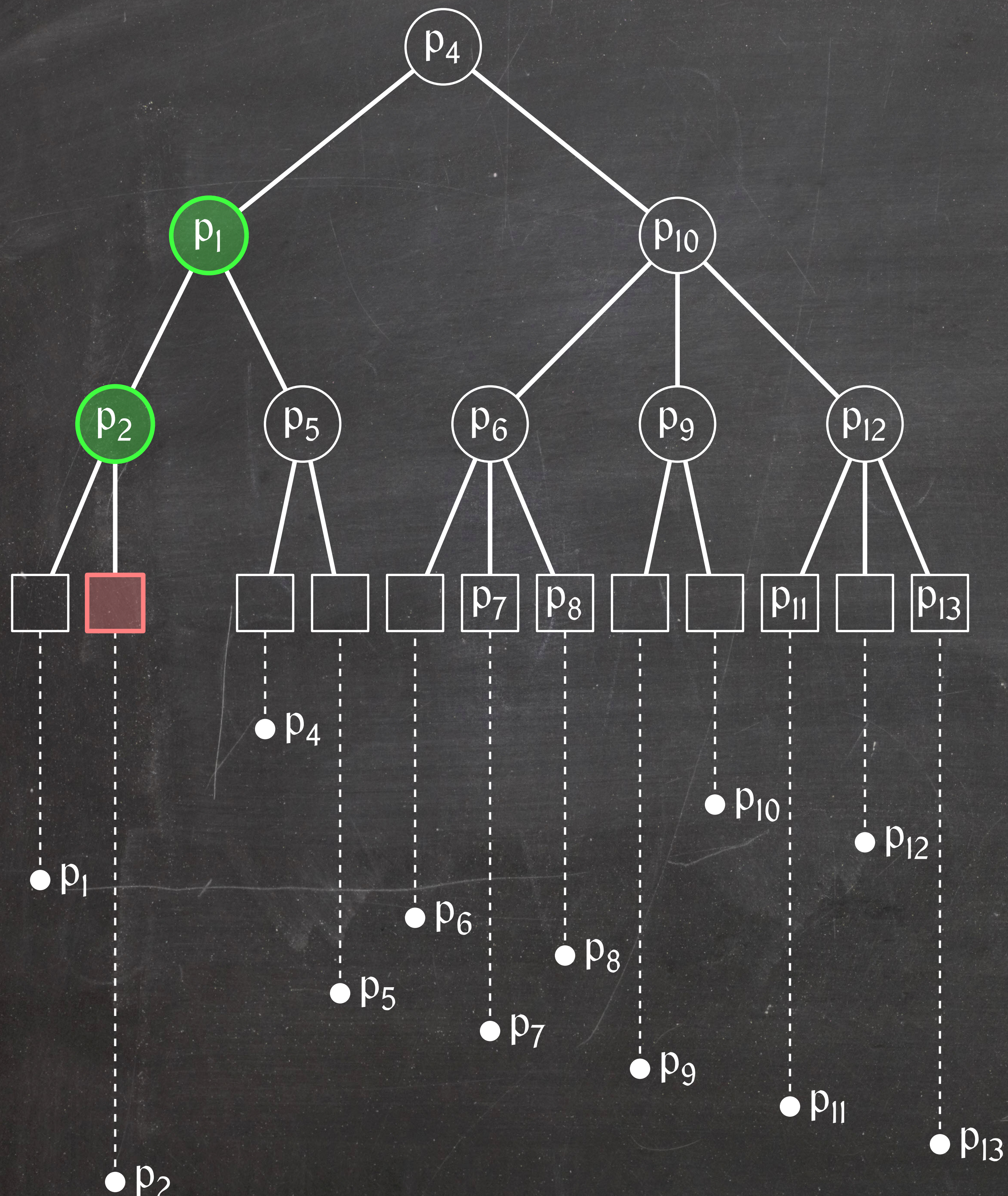
Delete the leaf corresponding to p.

Delete p from the node where it is stored.

While the current node v has a child that stores a point:

- Choose the child w whose point q has the highest y-coordinate.
- Store q at v.
- v = w

Deletions



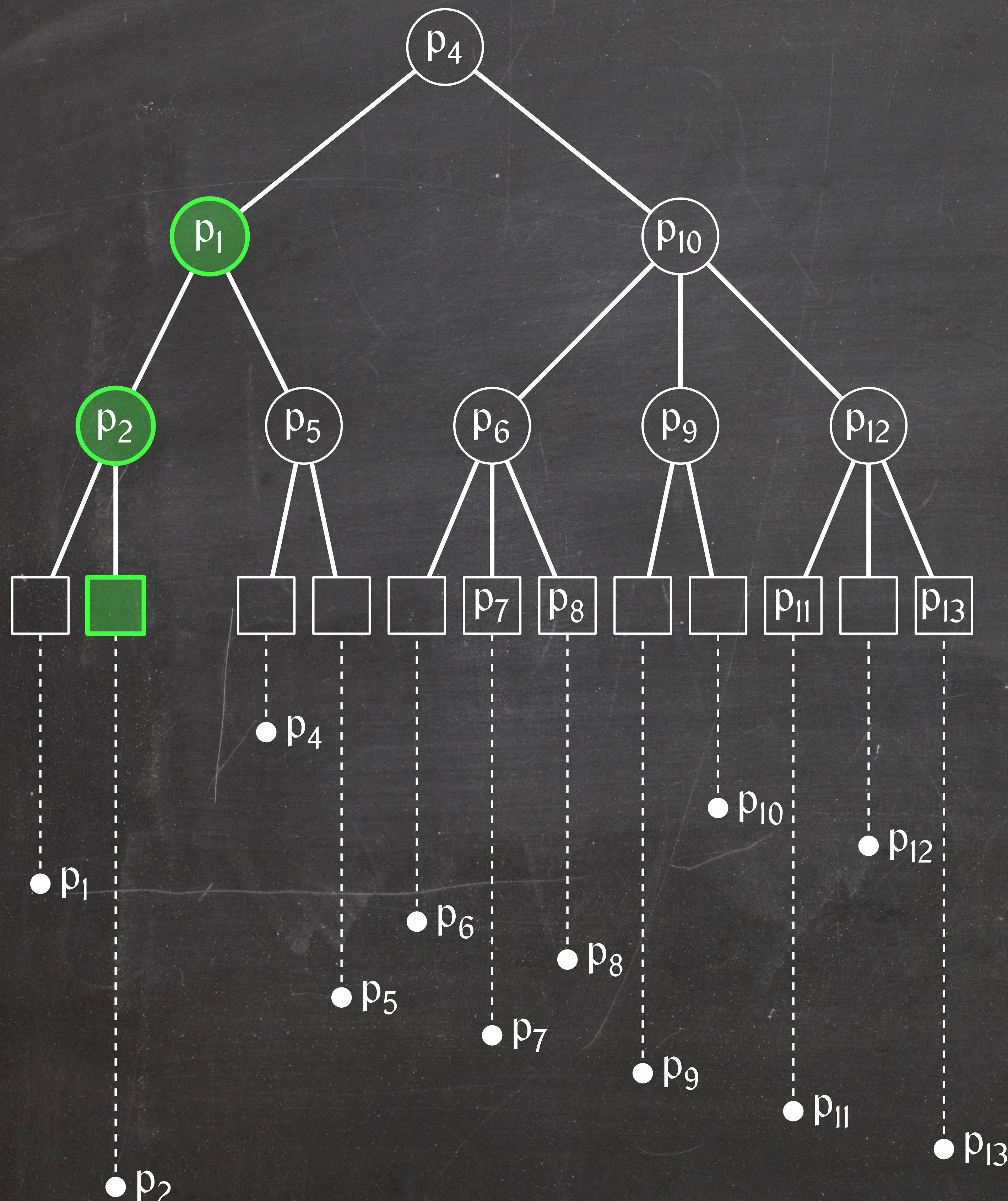
Delete the leaf corresponding to p.

Delete p from the node where it is stored.

While the current node v has a child that stores a point:

- Choose the child w whose point q has the highest y-coordinate.
- Store q at v.
- v = w

Deletions



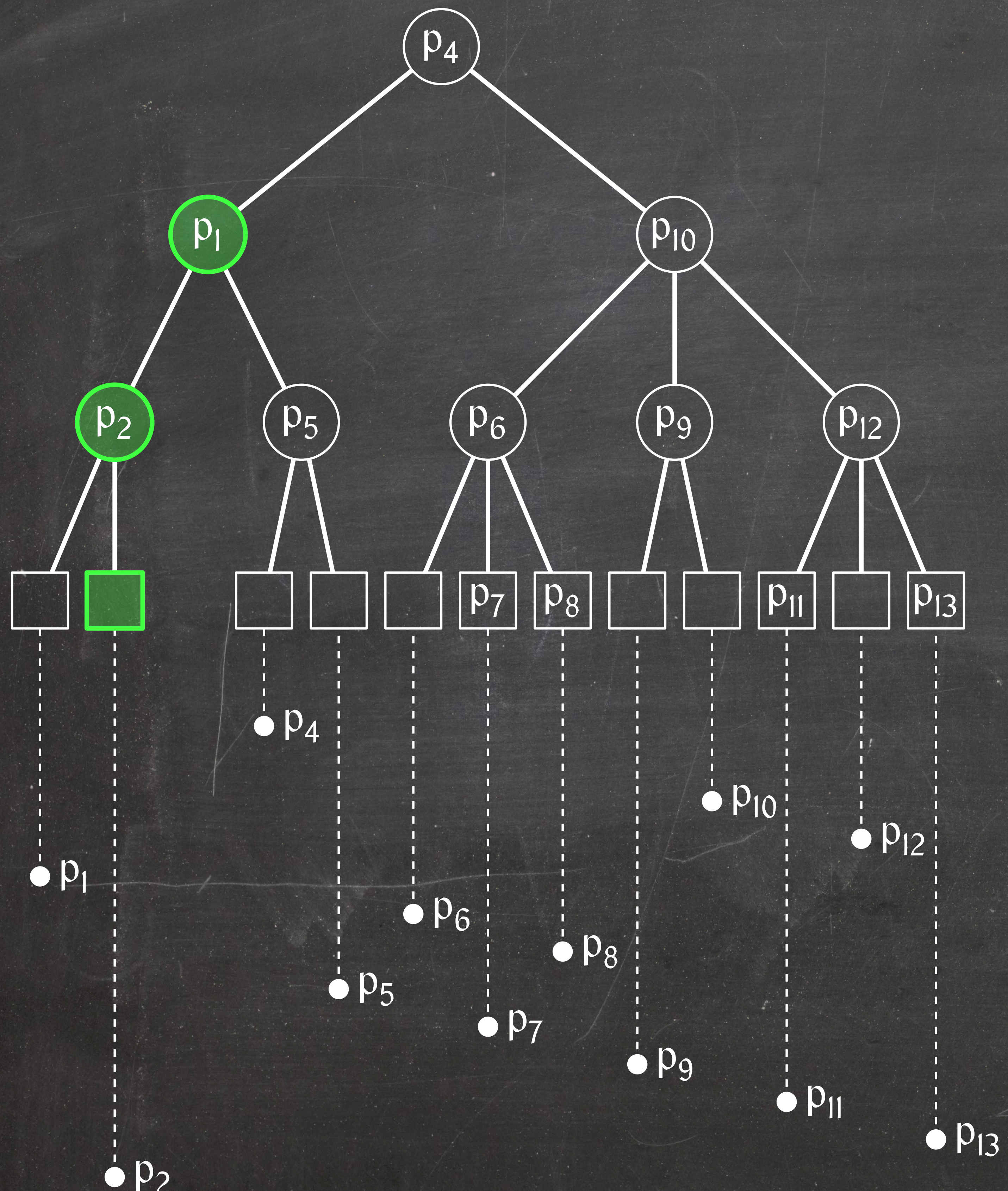
Delete the leaf corresponding to p.

Delete p from the node where it is stored.

While the current node v has a child that stores a point:

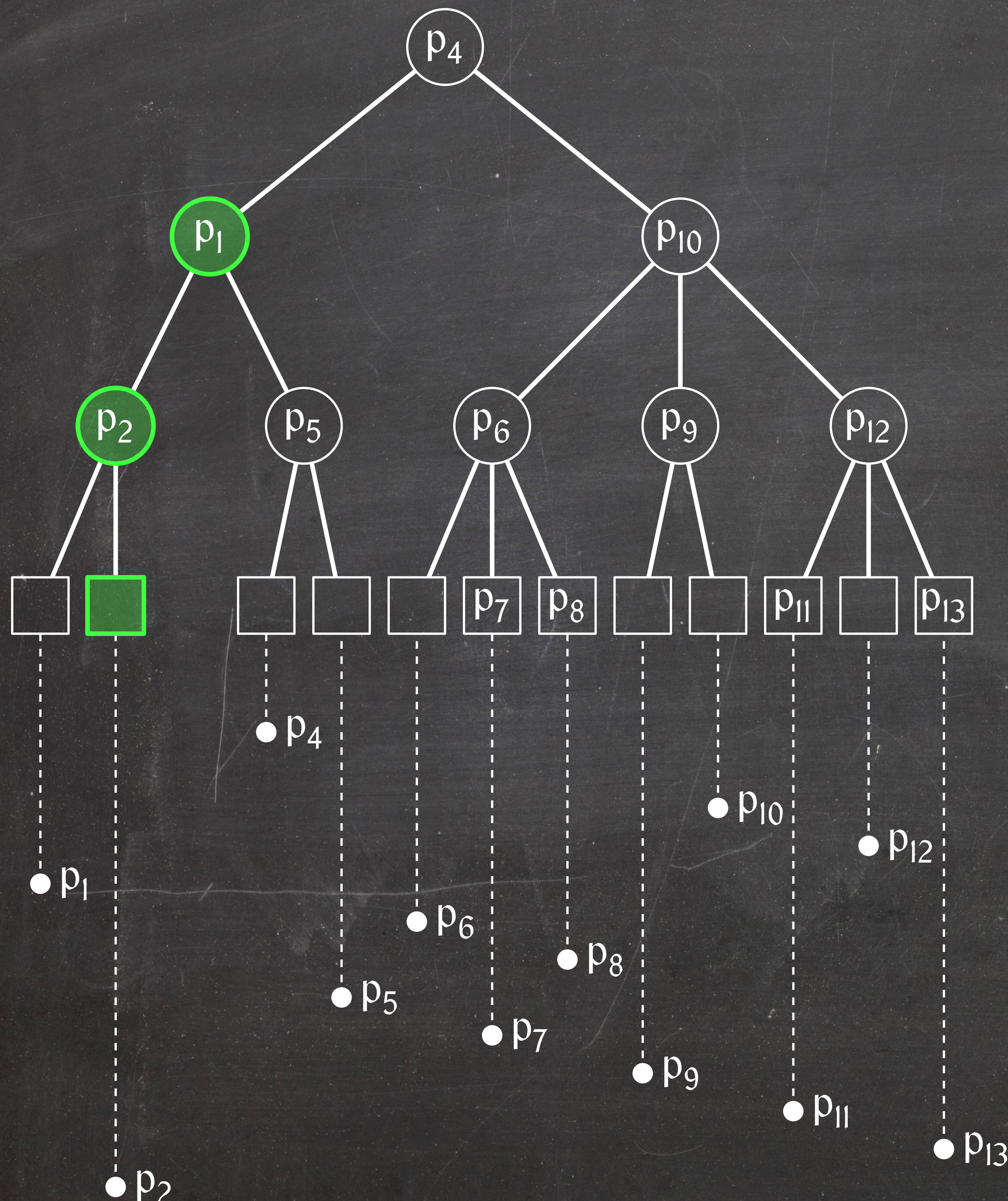
- Choose the child w whose point q has the highest y-coordinate.
- Store q at v.
- v = w

Deletions



Deleting p's leaf takes O(lg n) time.

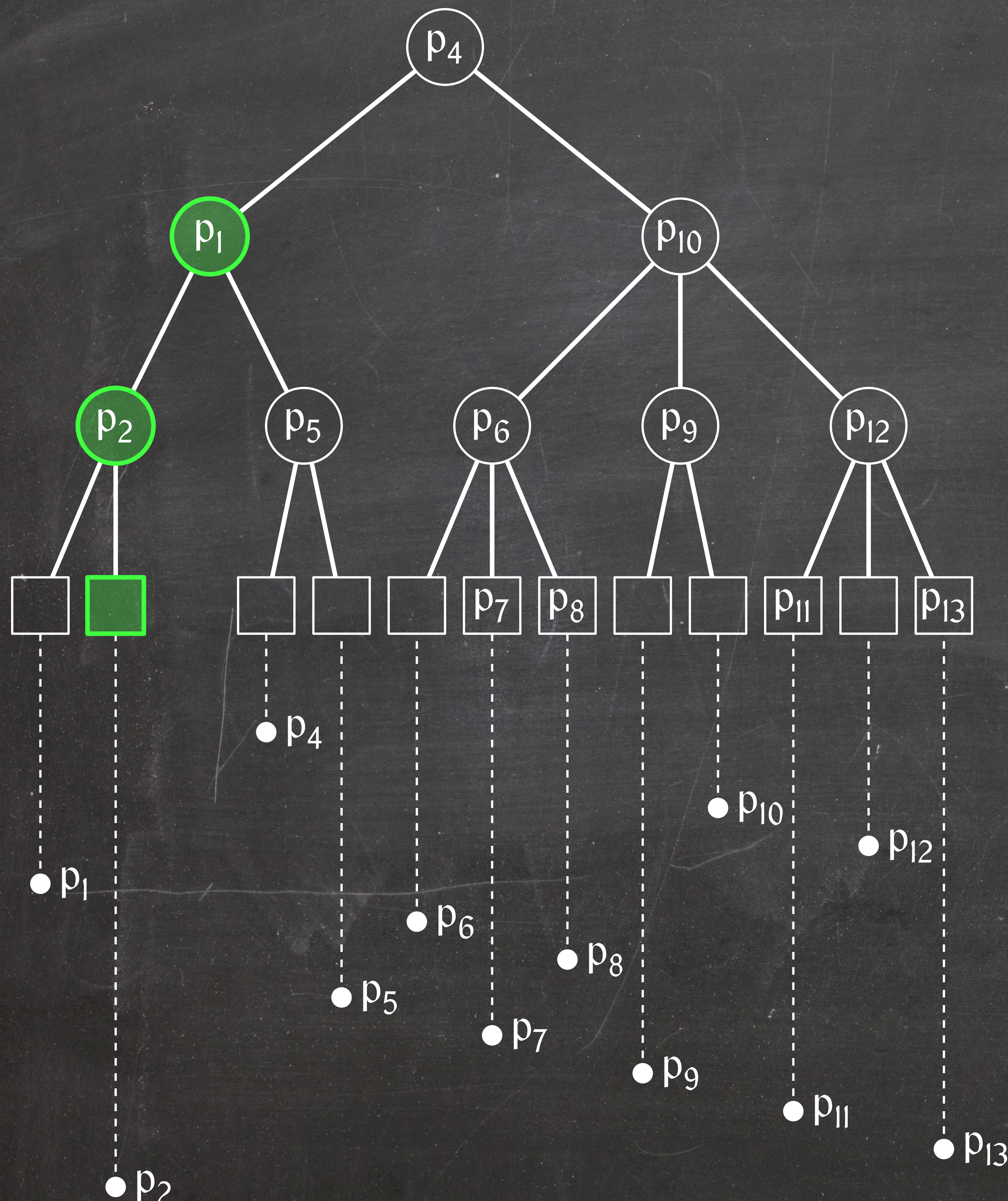
Deletions



Deleting p's leaf takes O(lg n) time.

So does locating the node storing p and deleting p from it.

Deletions

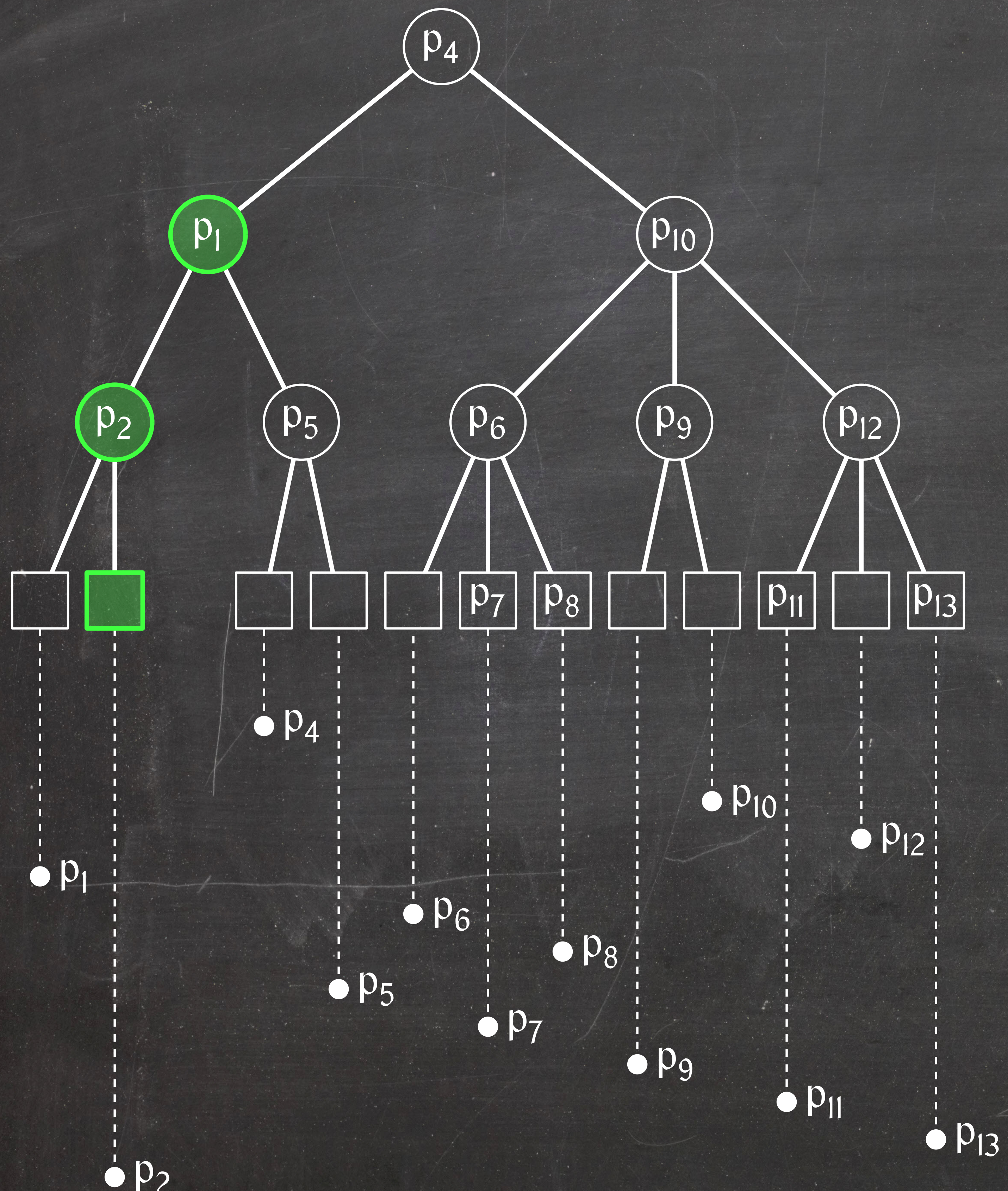


Deleting p's leaf takes O(lg n) time.

So does locating the node storing p and deleting p from it.

Backfilling the "hole" this creates amounts to traversing a single top-down path. This also takes O(lg n) time.

Deletions



Deleting p's leaf takes O(lg n) time.

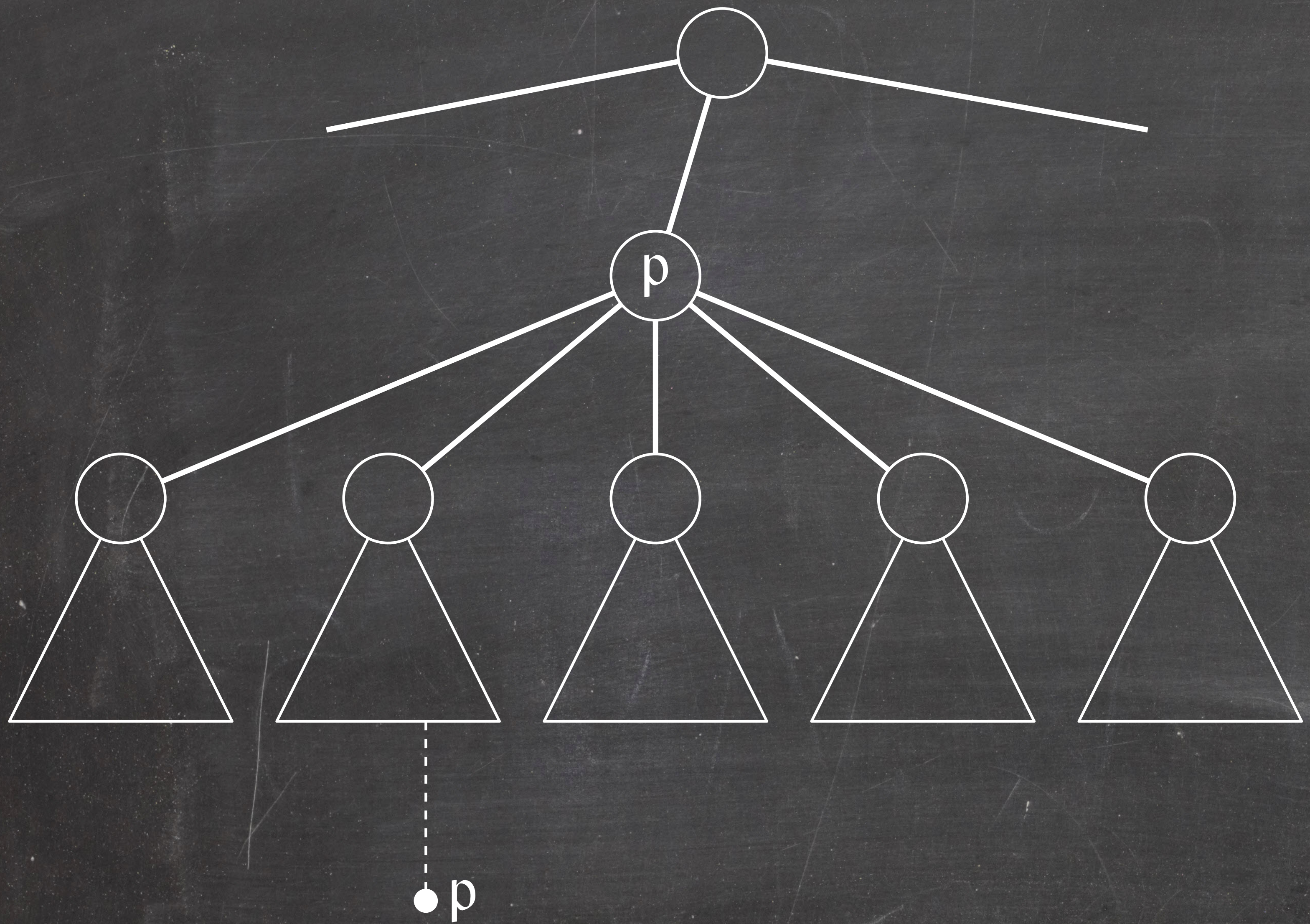
So does locating the node storing p and deleting p from it.

Backfilling the "hole" this creates amounts to traversing a single top-down path. This also takes O(lg n) time.

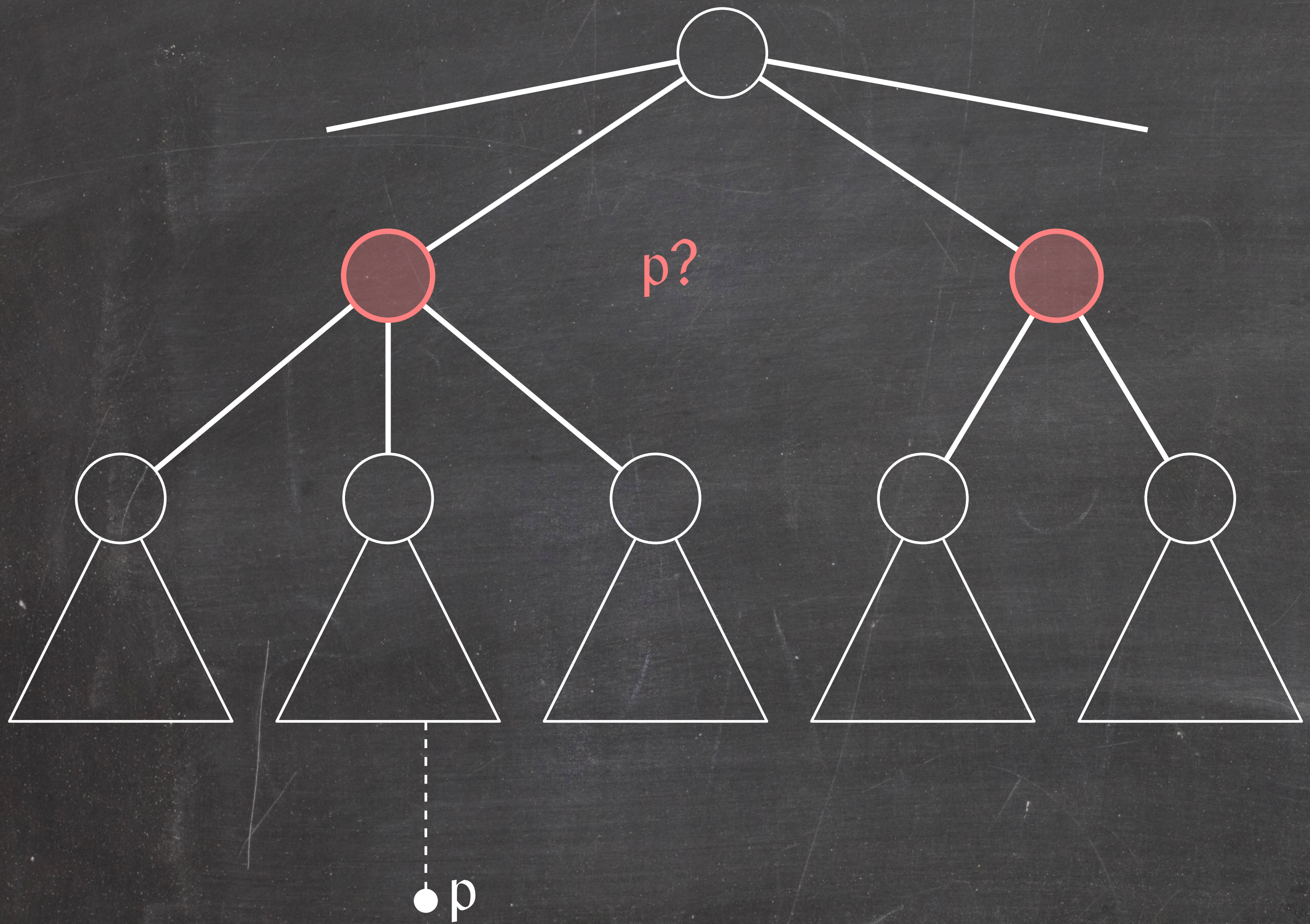
Total cost:

O(lg n) (excluding node fusions)

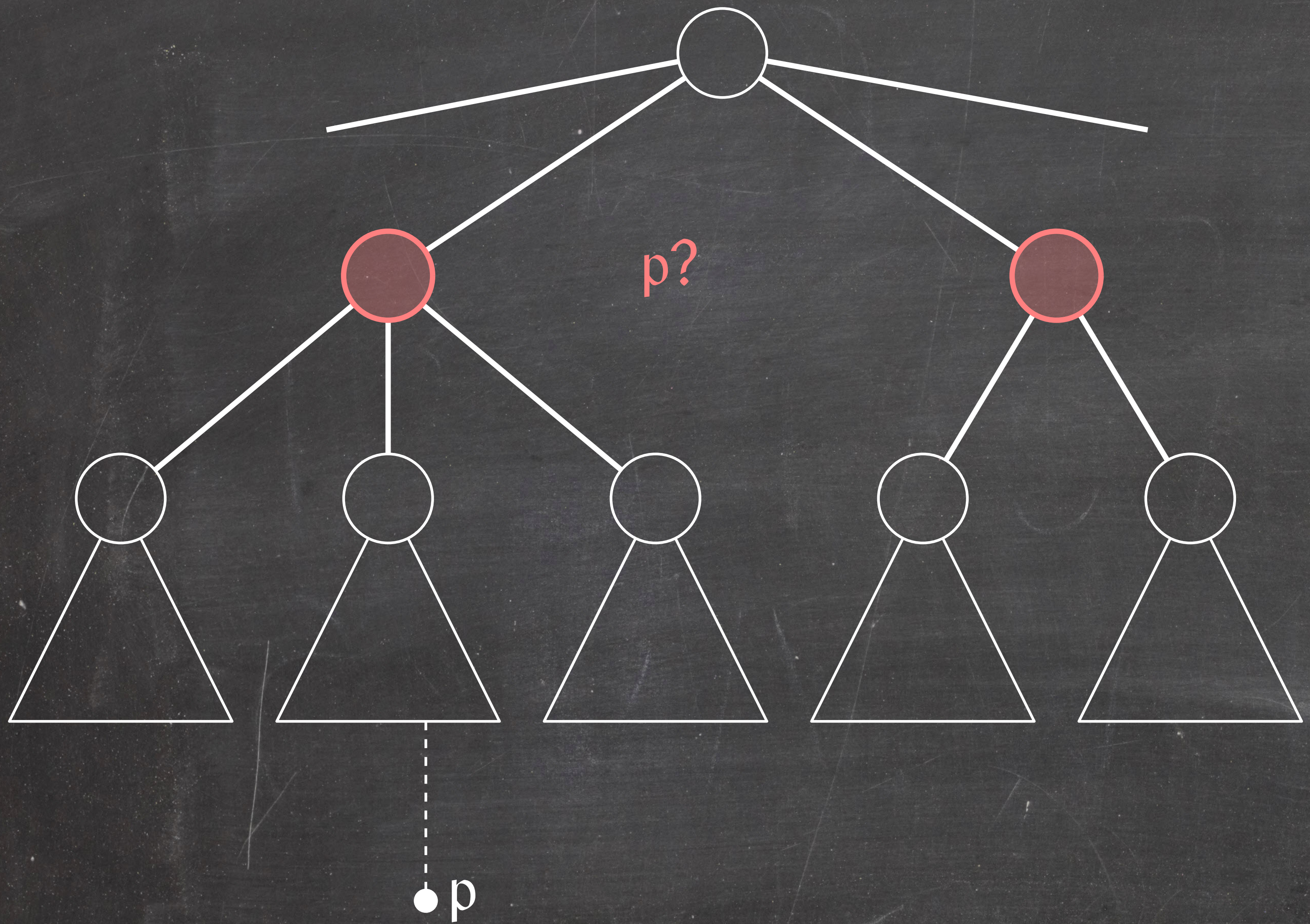
Node Splits



Node Splits

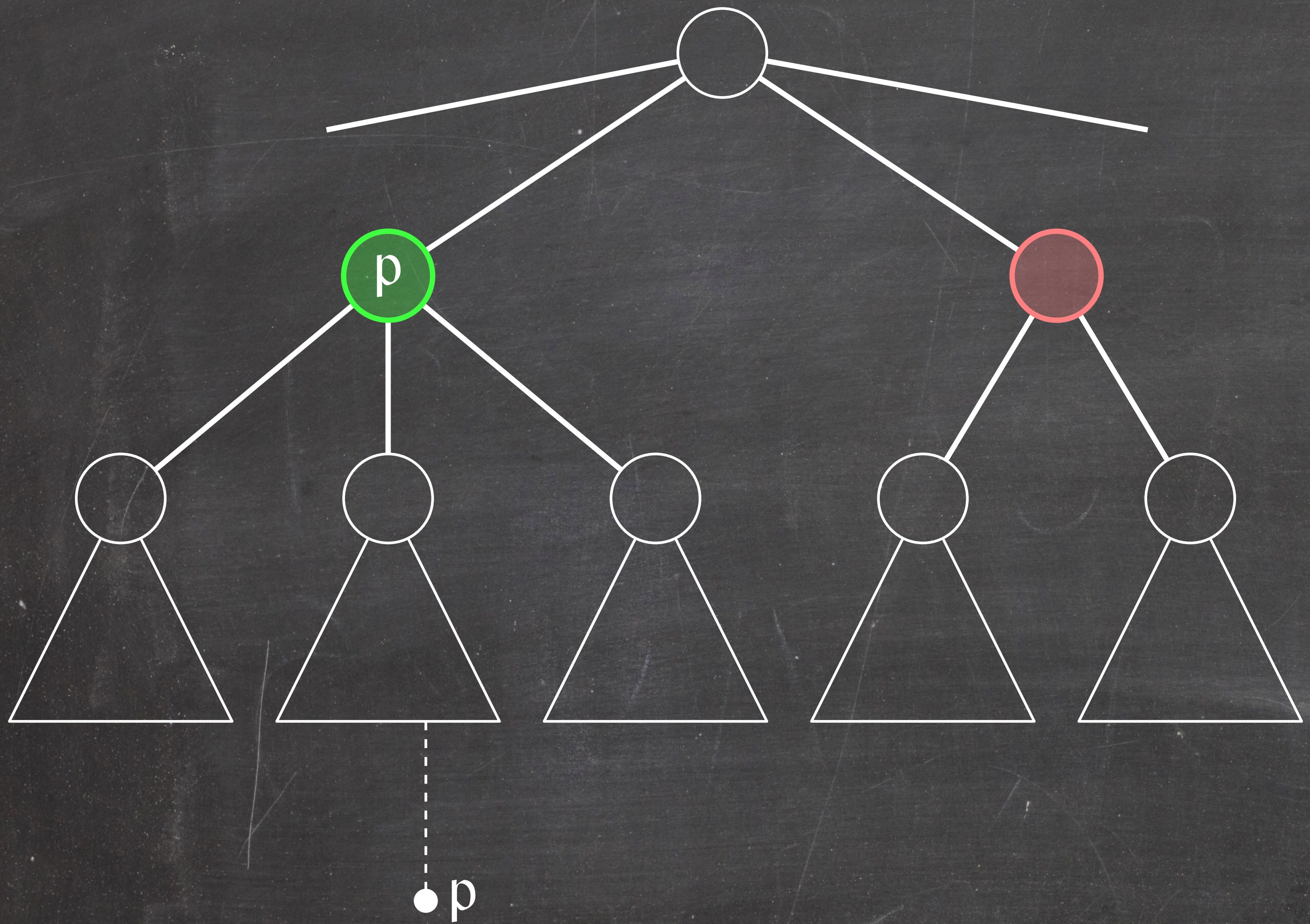


Node Splits



Where do we store p?

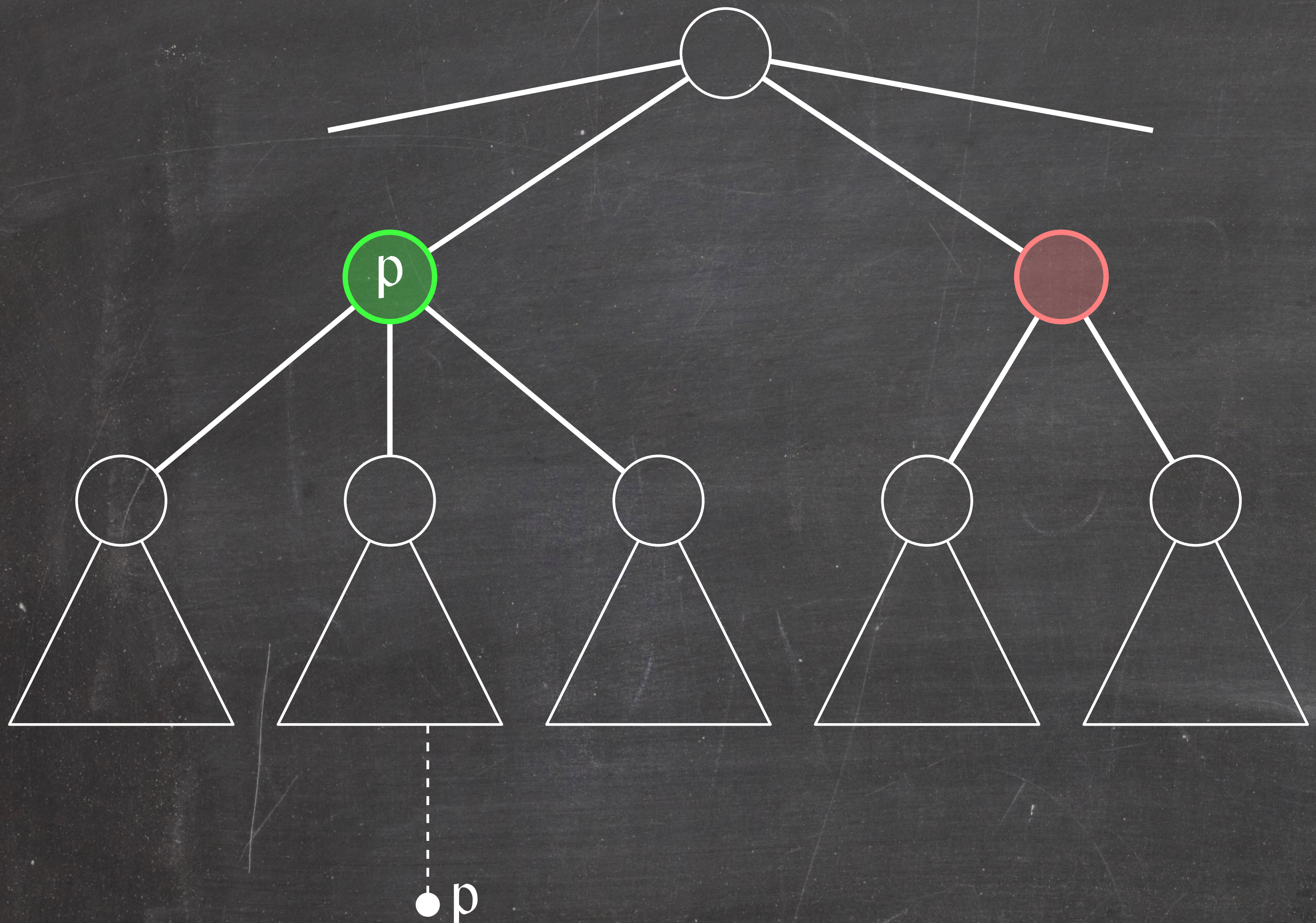
Node Splits



Where do we store p?

At the node that is the ancestor of p's leaf.

Node Splits

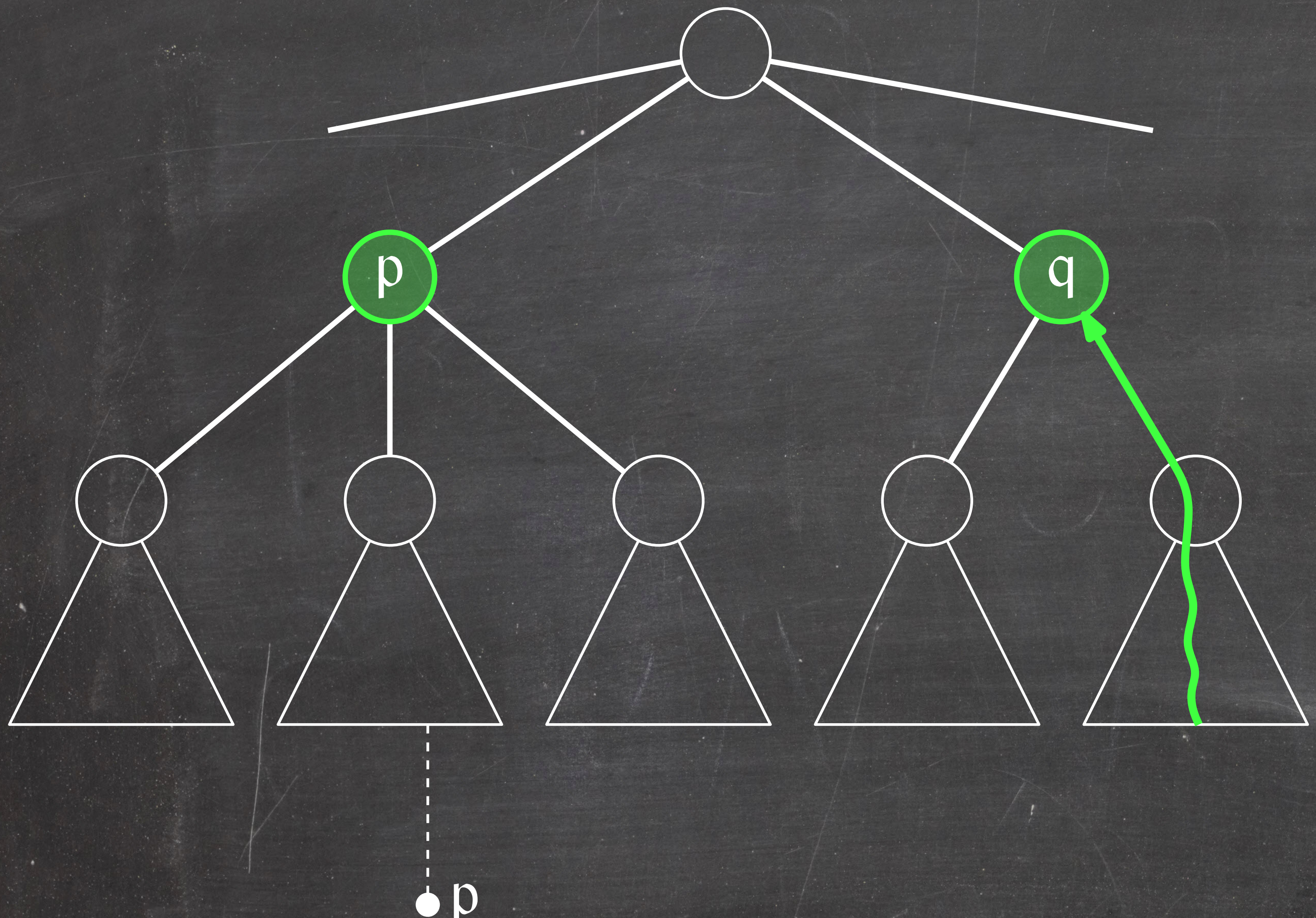


Where do we store p?

At the node that is the ancestor of p's leaf.

What do we store at the other node we created?

Node Splits



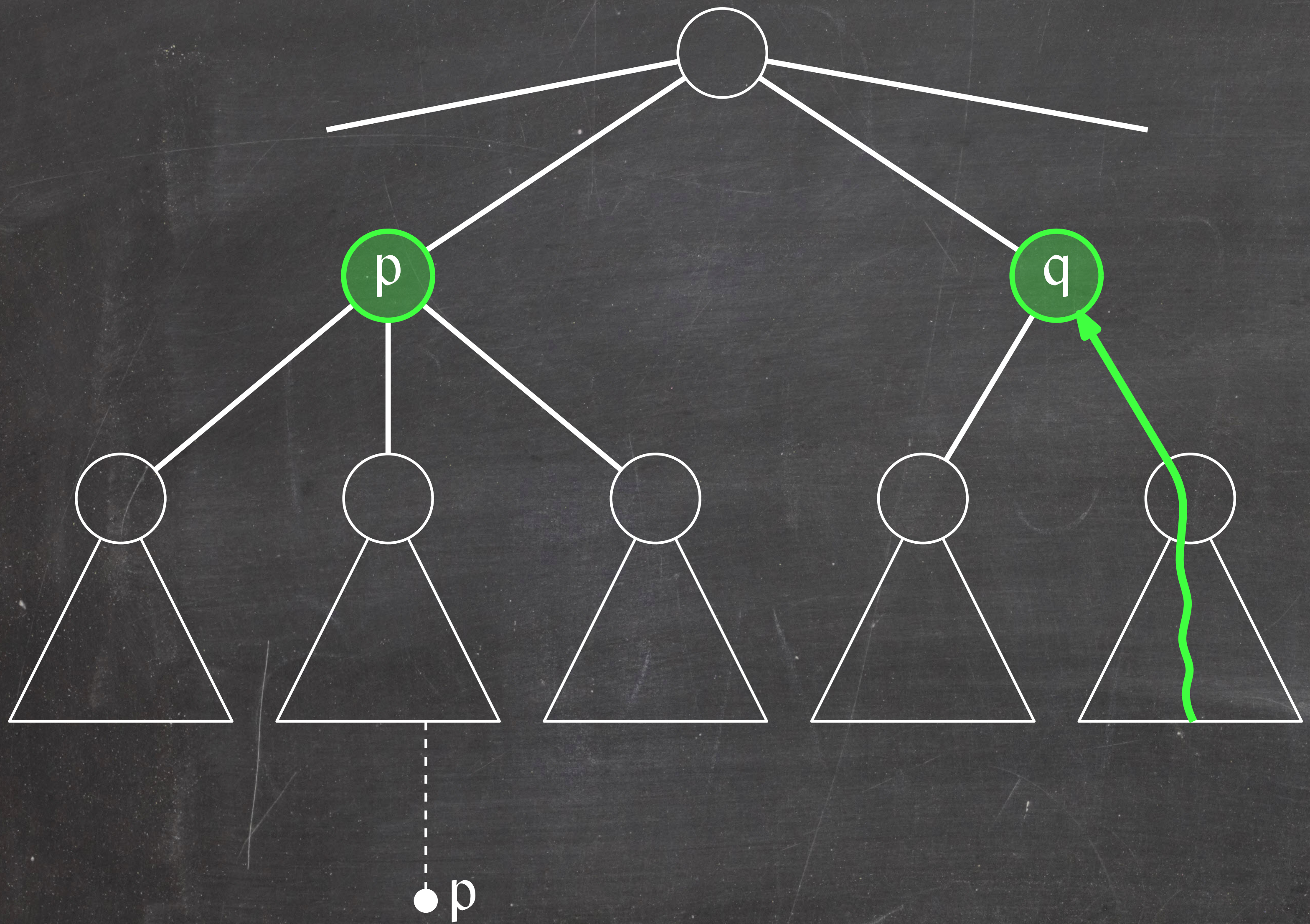
Where do we store p?

At the node that is the ancestor of p's leaf.

What do we store at the other node we created?

We backfill as after a deletion.

Node Splits



Where do we store p?

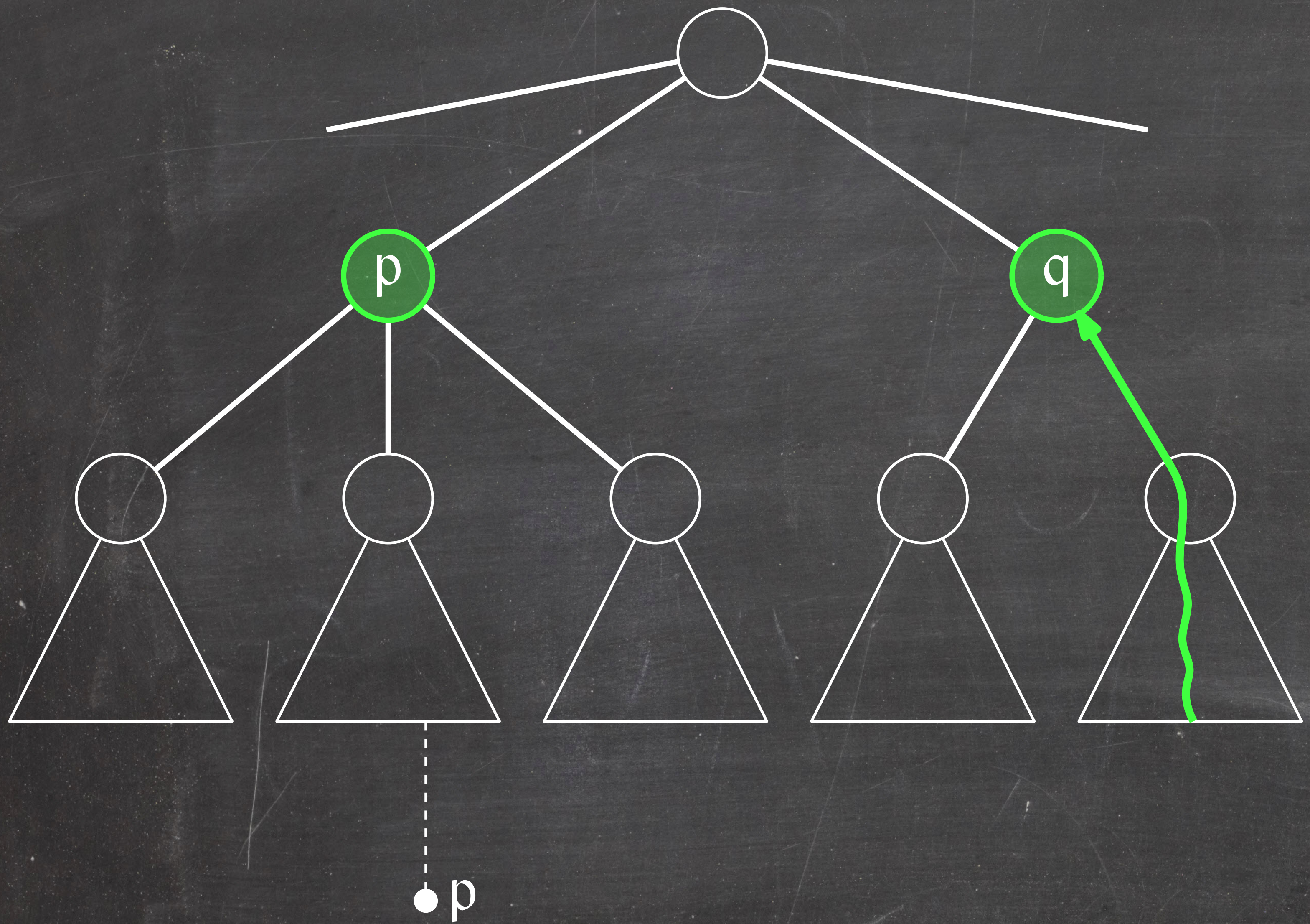
At the node that is the ancestor of p's leaf.

What do we store at the other node we created?

We backfill as after a deletion.

Lemma: A node split takes $O(\lg n)$ time.

Node Splits



Where do we store p?

At the node that is the ancestor of p's leaf.

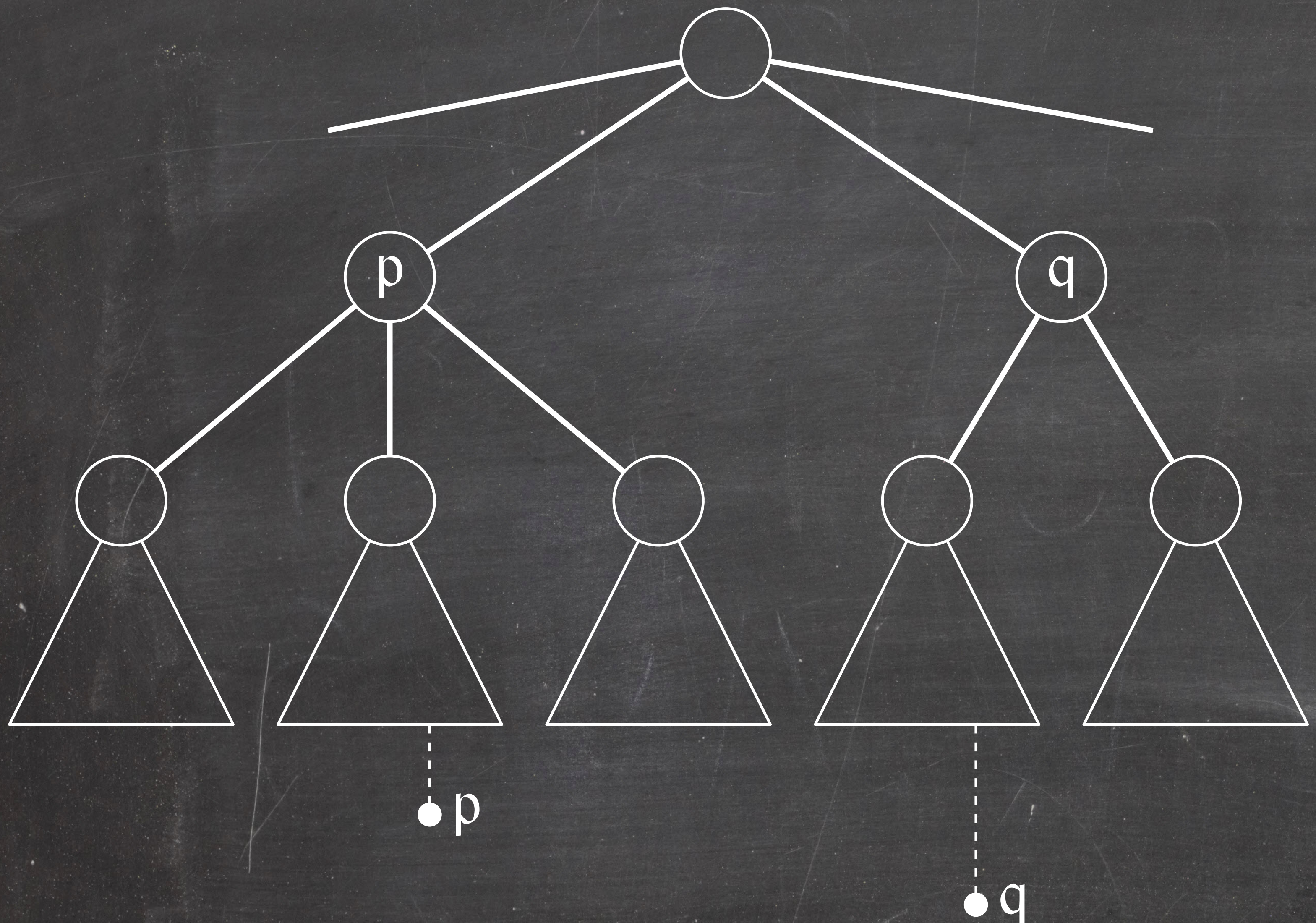
What do we store at the other node we created?

We backfill as after a deletion.

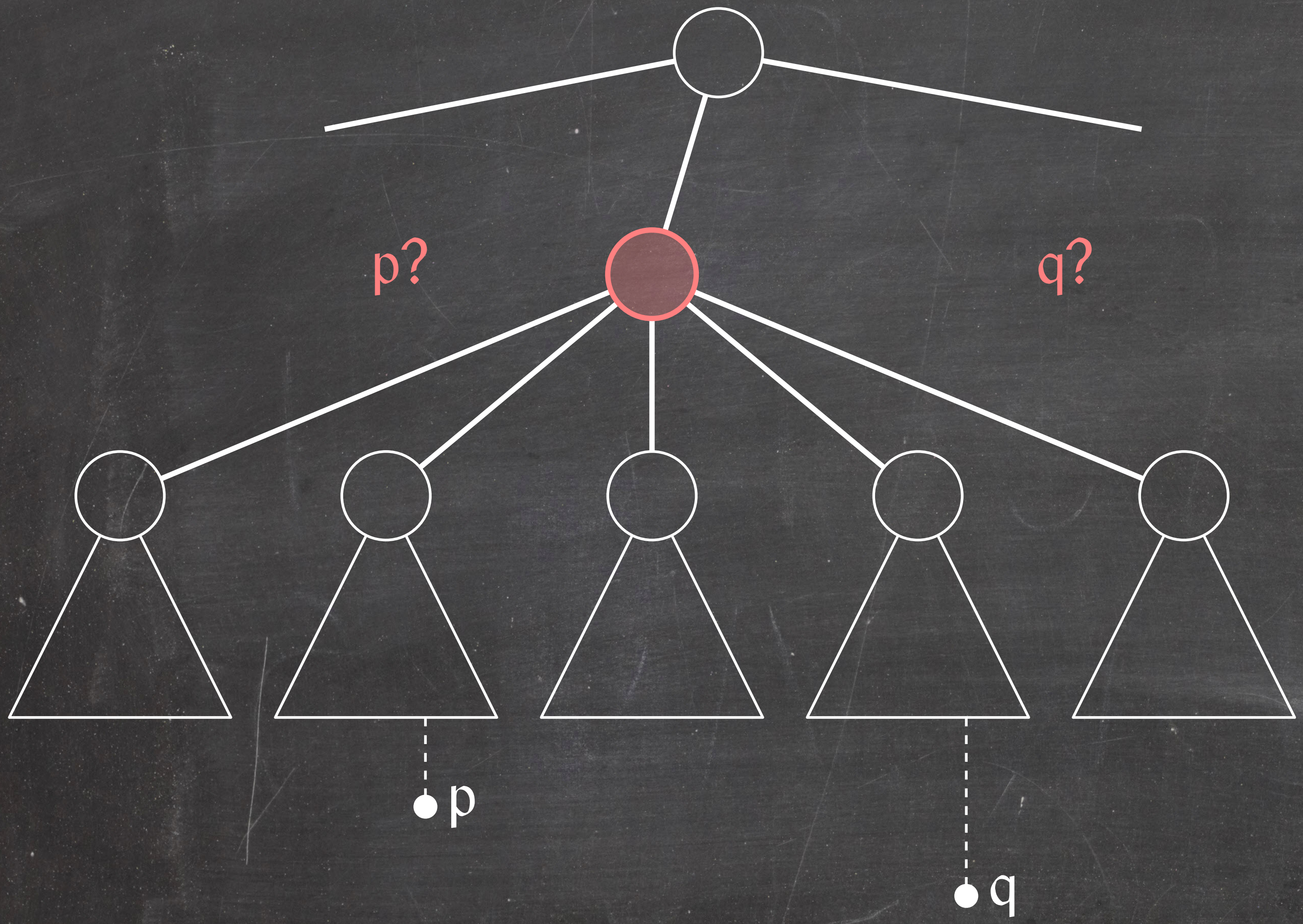
Lemma: A node split takes $O(\lg n)$ time.

Corollary: An insertion into a Priority Search Tree takes $O(\lg^2 n)$ time.

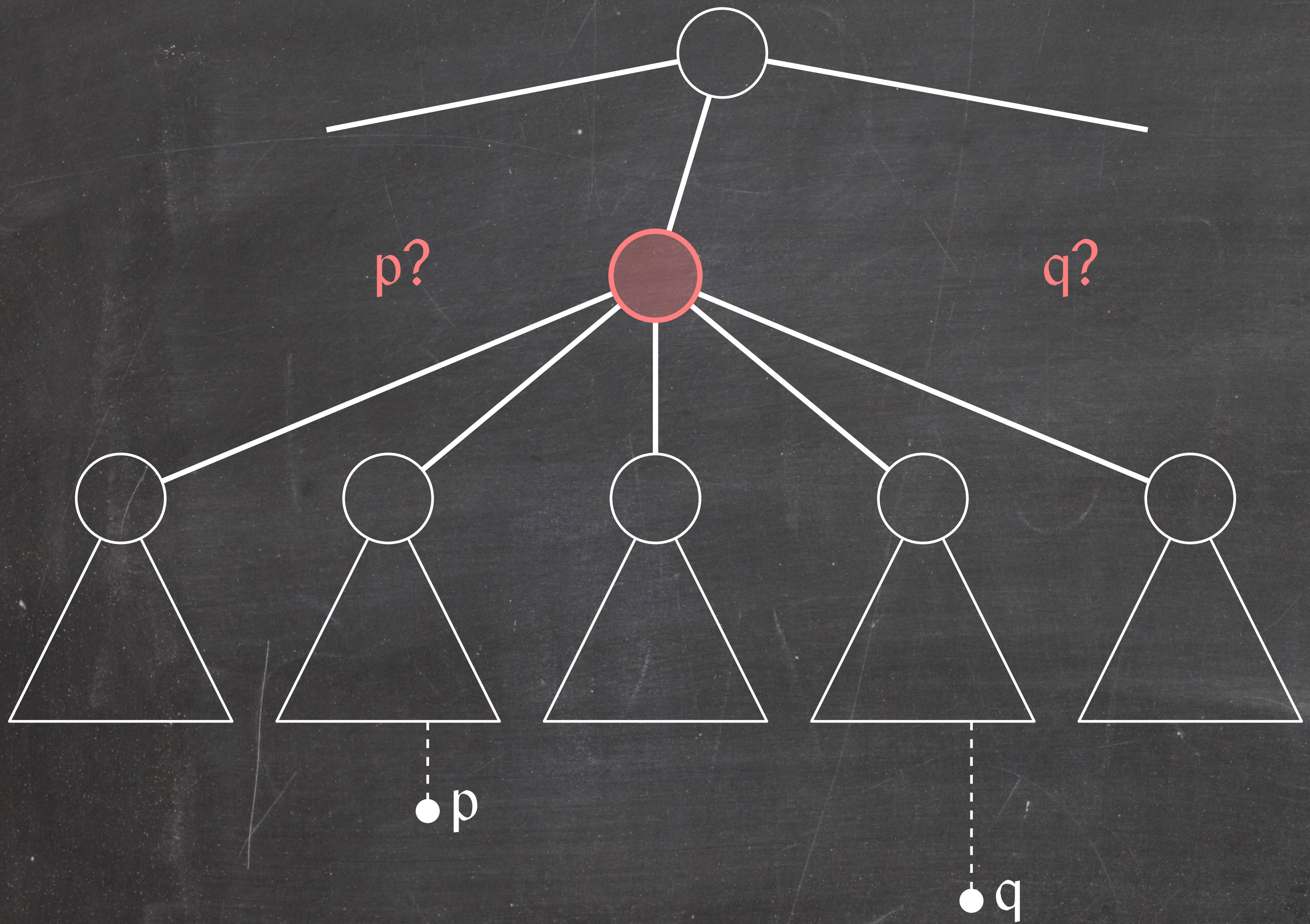
Node Fusions



Node Fusions

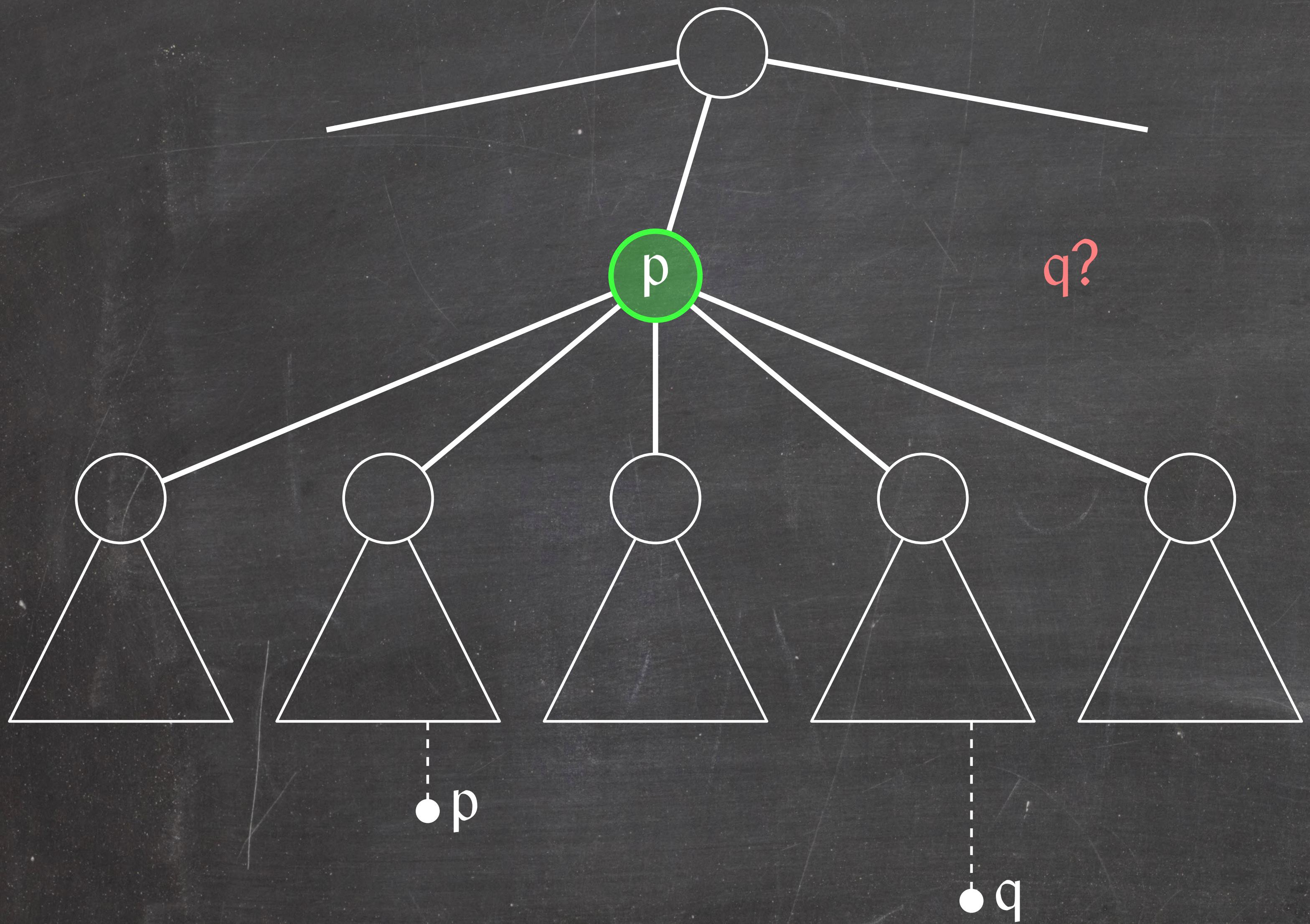


Node Fusions



Which of the two points do we store at the merged node?

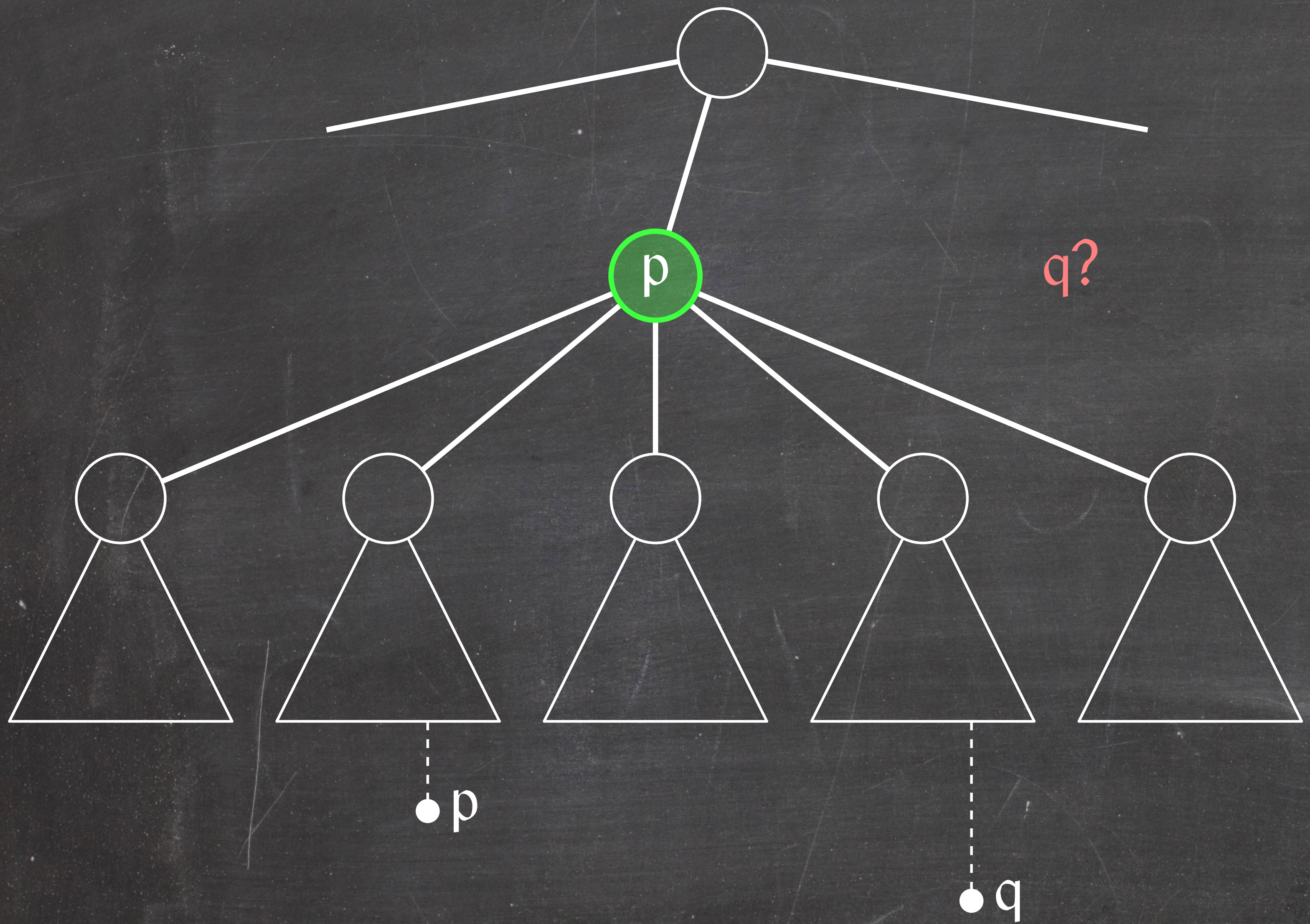
Node Fusions



Which of the two points do we store at the merged node?

The one with higher y-coordinate.

Node Fusions

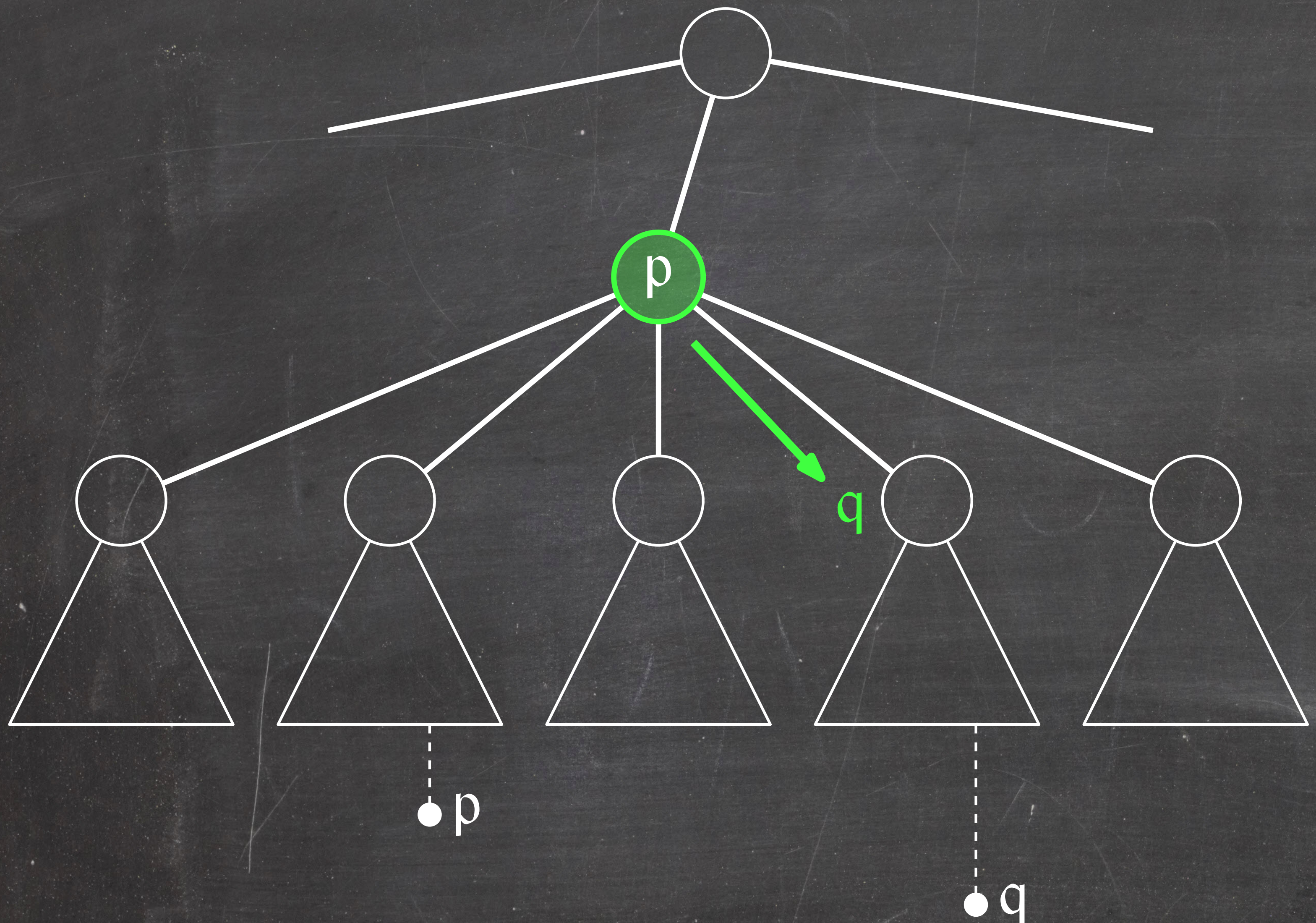


Which of the two points do we store at the merged node?

The one with higher y-coordinate.

Where do we store the other point?

Node Fusions



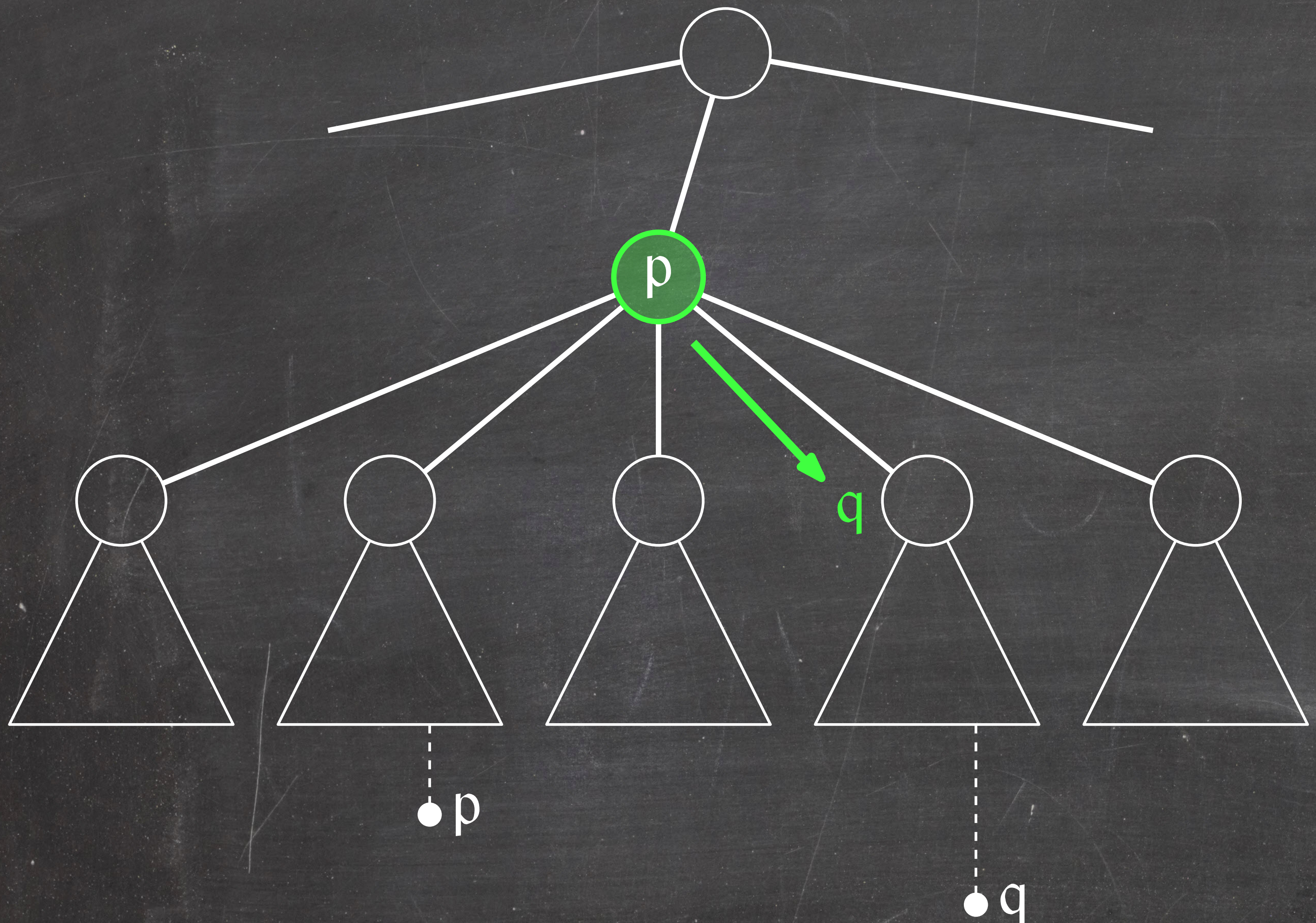
Which of the two points do we store at the merged node?

The one with higher y-coordinate.

Where do we store the other point?

We push it down the tree as after an insertion.

Node Fusions



Which of the two points do we store at the merged node?

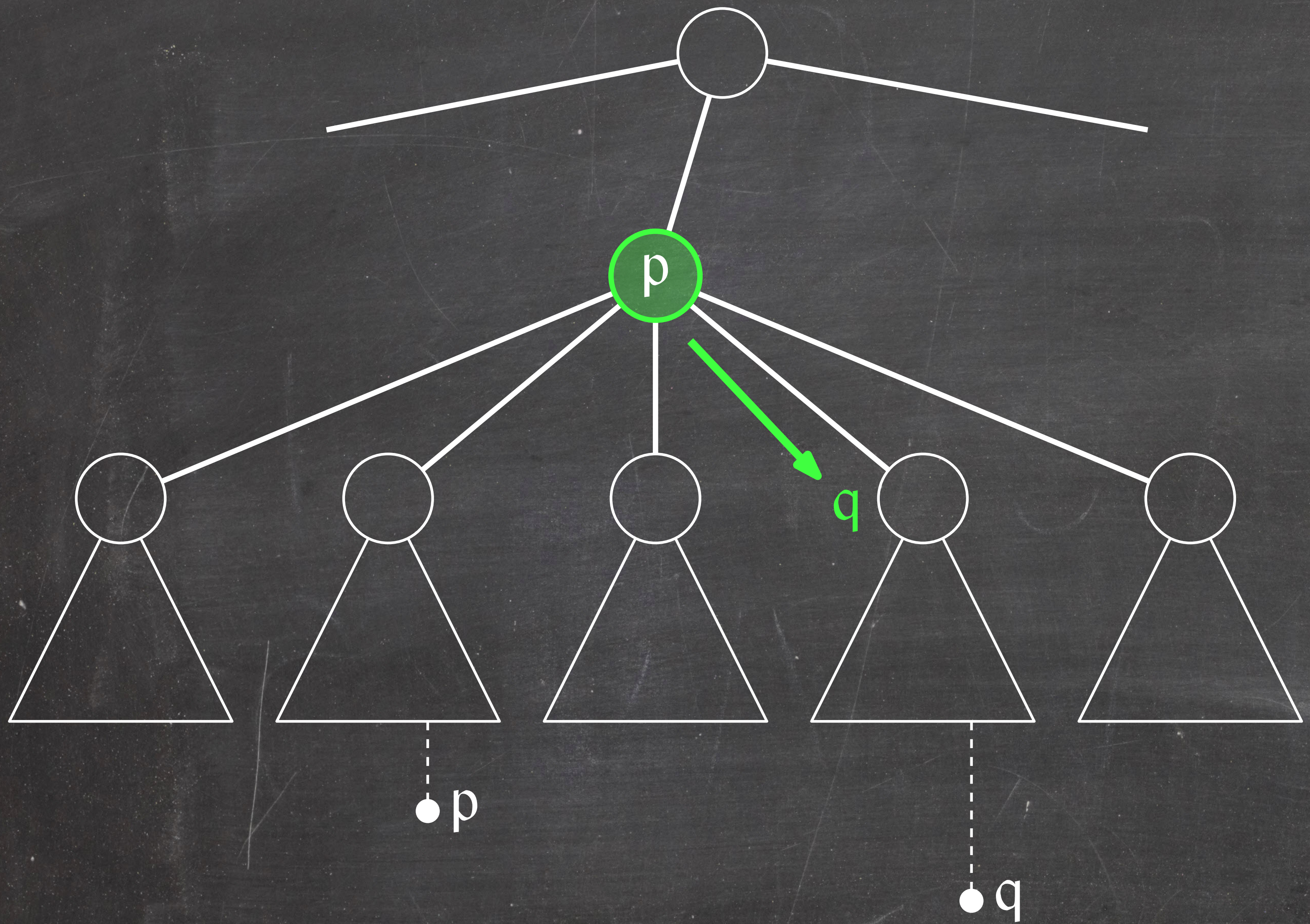
The one with higher y-coordinate.

Where do we store the other point?

We push it down the tree as after an insertion.

Lemma: A node fusion takes $O(\lg n)$ time.

Node Fusions



Which of the two points do we store at the merged node?

The one with higher y-coordinate.

Where do we store the other point?

We push it down the tree as after an insertion.

Lemma: A node fusion takes $O(\lg n)$ time.

Corollary: A deletion from a Priority Search Tree takes $O(\lg^2 n)$ time.

Priority Search Tree: Summary

Theorem: A Priority Search Tree supports Insert and Delete operations in $O(\lg^2 n)$ time and three-sided range queries in $O(\lg n + k)$ time.

Note: One can show that there are only $O(n/(b/2 - a))$ node splits and fusions over any sequence of n (a, b)-tree updates. Hence, the amortized cost per Insert and Delete operation is in $O(\lg n)$.

Note: In a red-black tree, every Insert and Delete operation causes only $O(1)$ rotations. Rotations are the equivalent of node splits and fusions. Hence, a priority search tree based on a red-black tree supports Insert and Delete operations in $O(\lg n)$ time in the worst case.

Augmenting (a, b)-Trees: The Template

What do we need to store to make queries fast?

Can we maintain this information efficiently under updates?

Augmenting (a, b)-Trees: The Template

What do we need to store to make queries fast?

Can we maintain this information efficiently under updates?

Insertions:

- Add a new leaf
- Up to $\lg n$ node splits

Augmenting (a, b)-Trees: The Template

What do we need to store to make queries fast?

Can we maintain this information efficiently under updates?

Insertions:

- Add a new leaf
- Up to $\lg n$ node splits

Deletions:

- Remove a leaf
- Up to $\lg n$ node splits and fusions

Augmenting (a, b)-Trees: The Template

What do we need to store to make queries fast?

Can we maintain this information efficiently under updates?

Insertions:

- Add a new leaf
- Up to $\lg n$ node splits

Deletions:

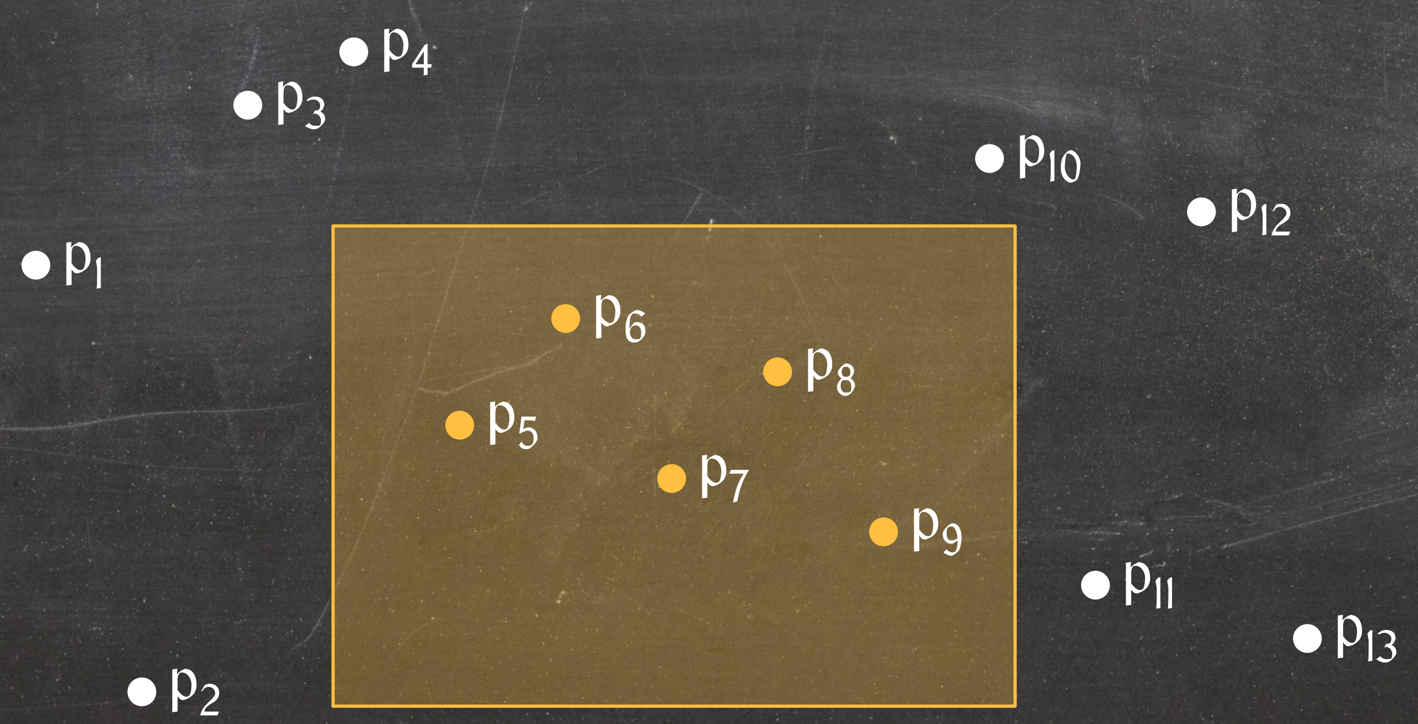
- Remove a leaf
- Up to $\lg n$ node splits and fusions

The only building blocks we need to worry about for updates:

- Fast leaf additions
- Fast leaf deletions
- (Very) fast node splits
- (Very) fast node fusions

d-Dimensional Range Reporting

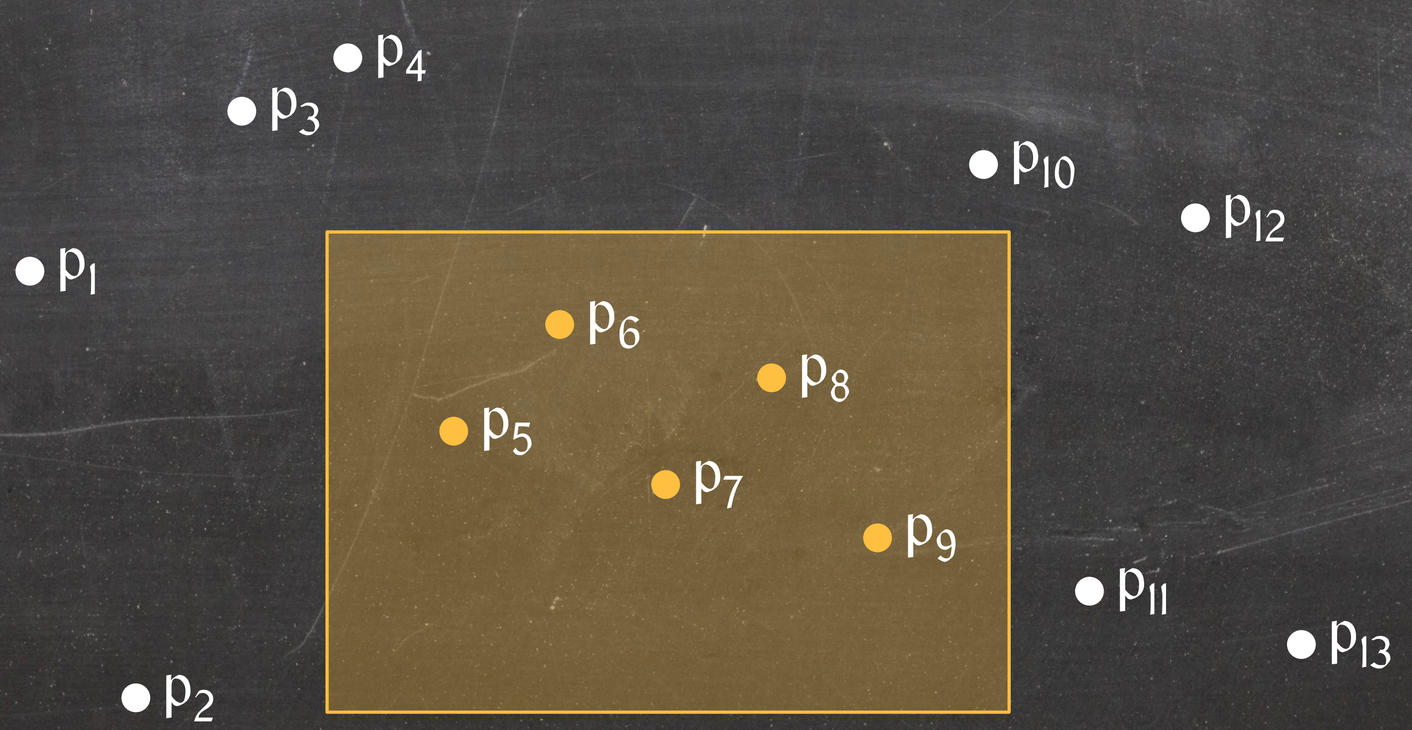
Goal: Build a static data structure over a point set S in $\mathbb{R}^d$ that allows us to report all the points in S that fall in a given (d-dimensional) query rectangle.



d-Dimensional Range Reporting

Goal: Build a static data structure over a point set S in $\mathbb{R}^d$ that allows us to report all the points in S that fall in a given (d-dimensional) query rectangle.

Queries should be fast.

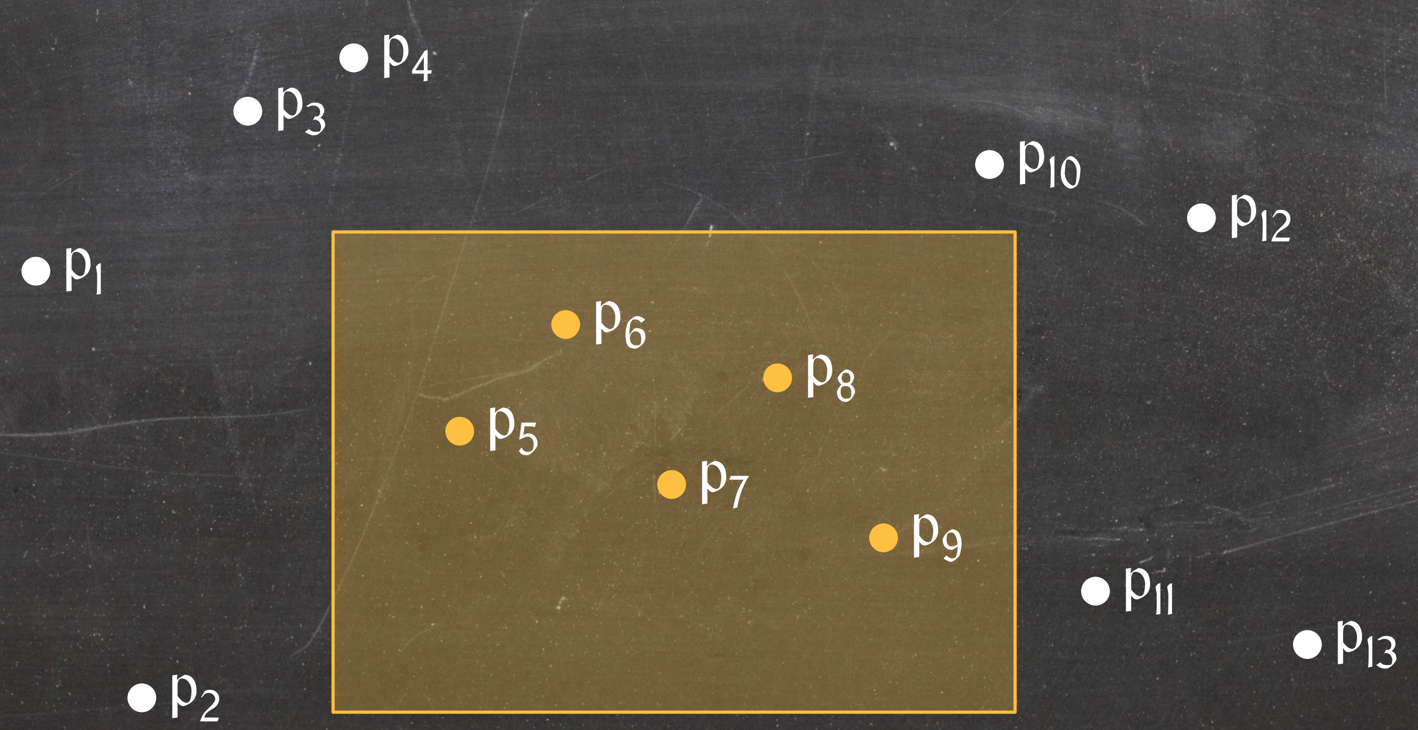


d-Dimensional Range Reporting

Goal: Build a static data structure over a point set S in $\mathbb{R}^d$ that allows us to report all the points in S that fall in a given (d-dimensional) query rectangle.

Queries should be fast.

The data structure should be small.



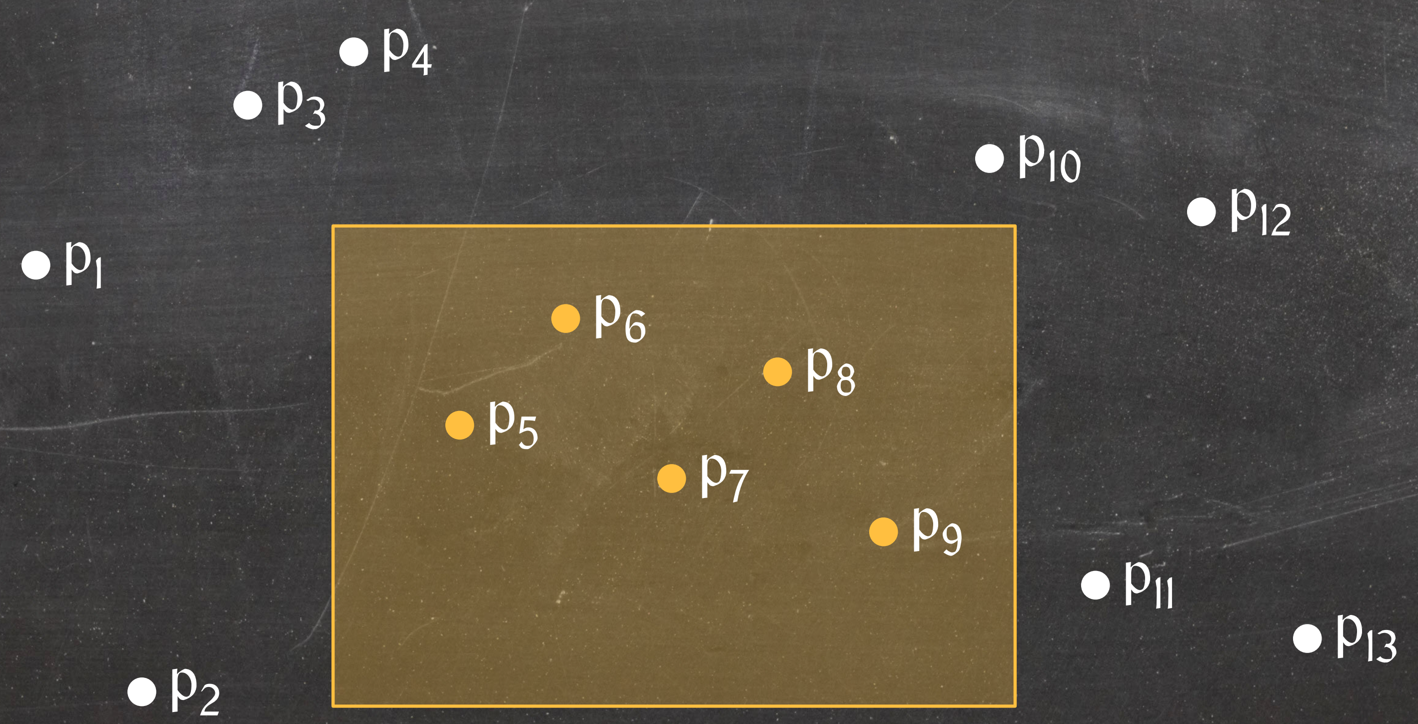
d-Dimensional Range Reporting

Goal: Build a static data structure over a point set S in $\mathbb{R}^d$ that allows us to report all the points in S that fall in a given (d-dimensional) query rectangle.

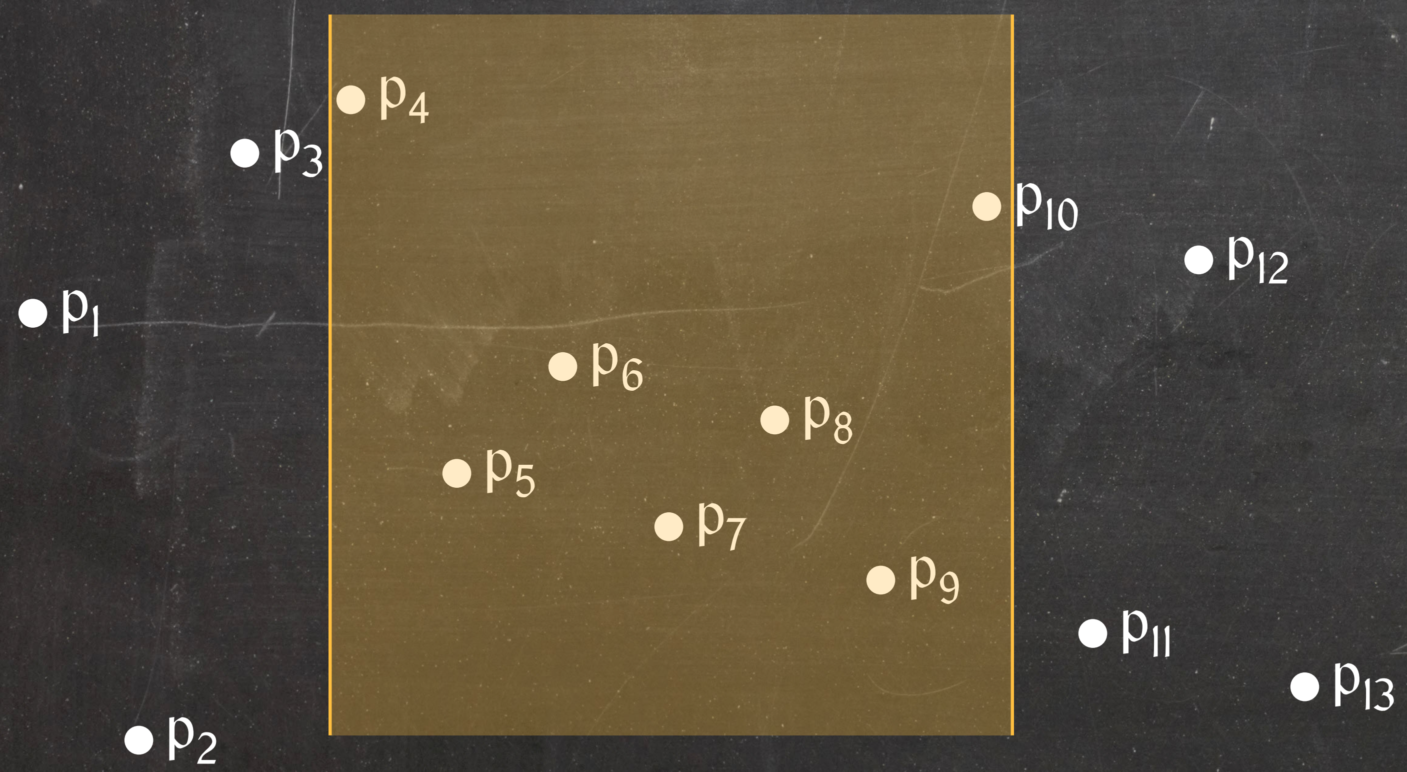
Queries should be fast.

The data structure should be small.

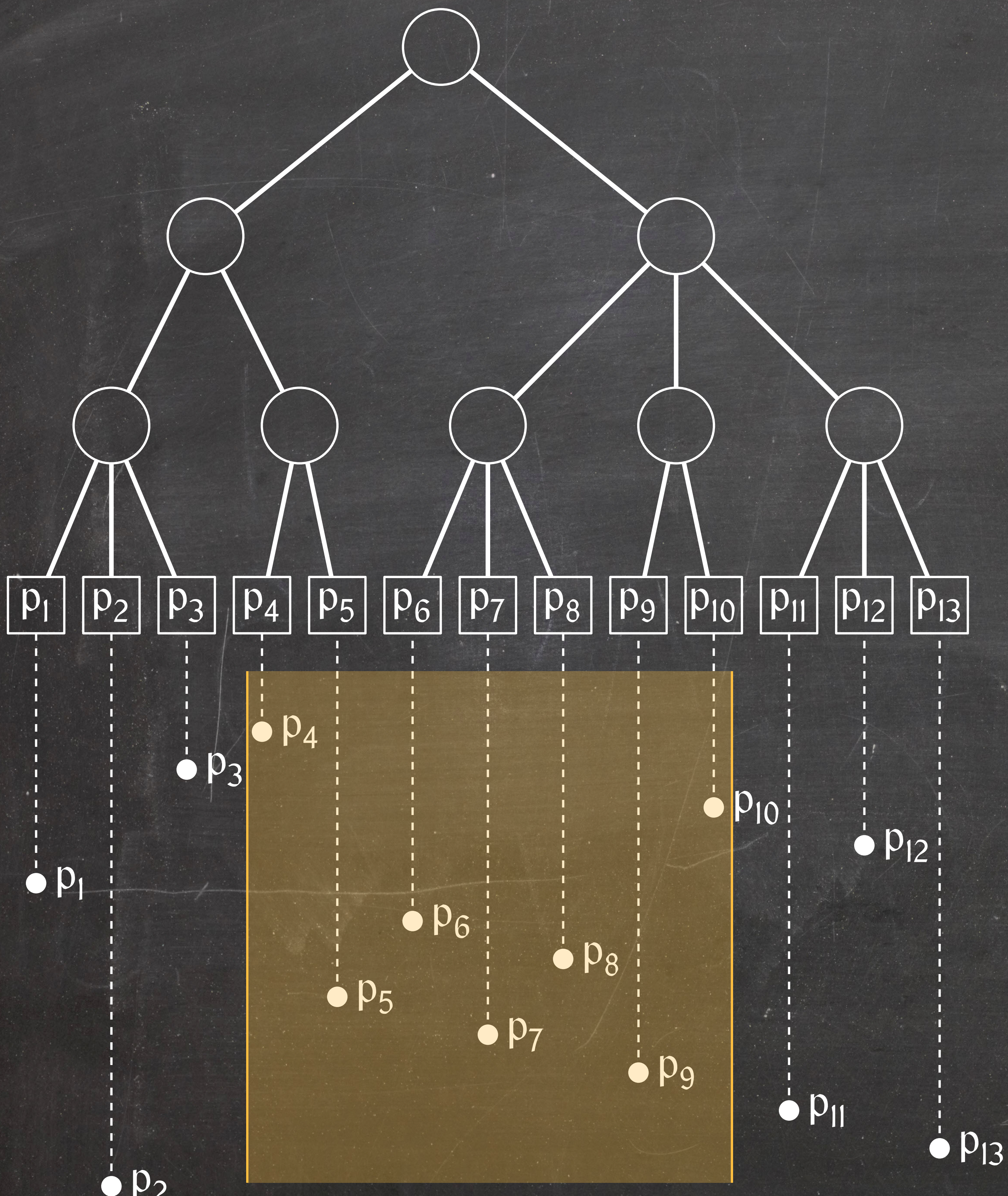
The data structure should be fast to build.



1-Dimensional Range Reporting ((a, b)-Tree)

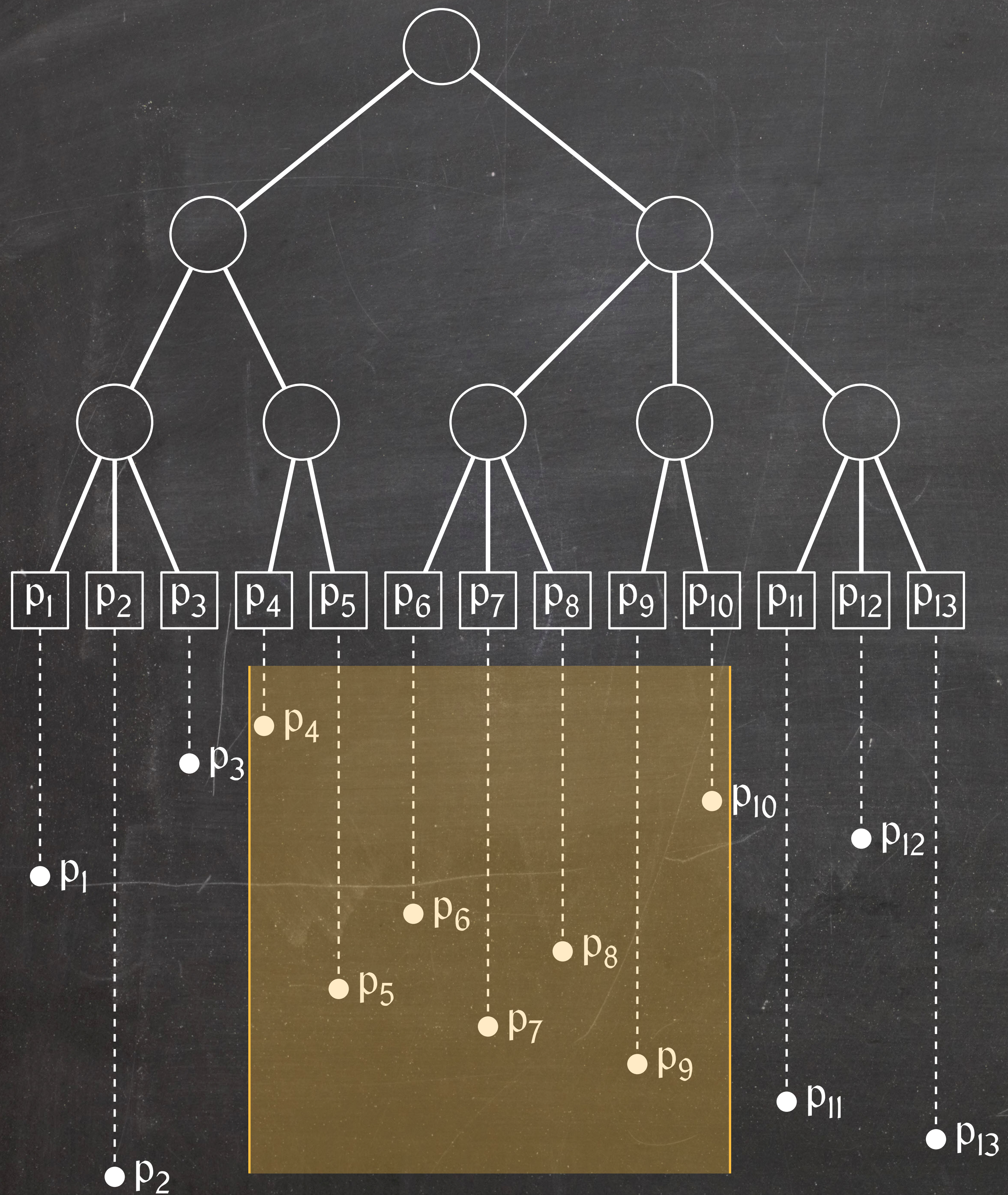


1-Dimensional Range Reporting ((a, b)-Tree)



1-Dimensional Range Reporting is just a standard RangeFind query.

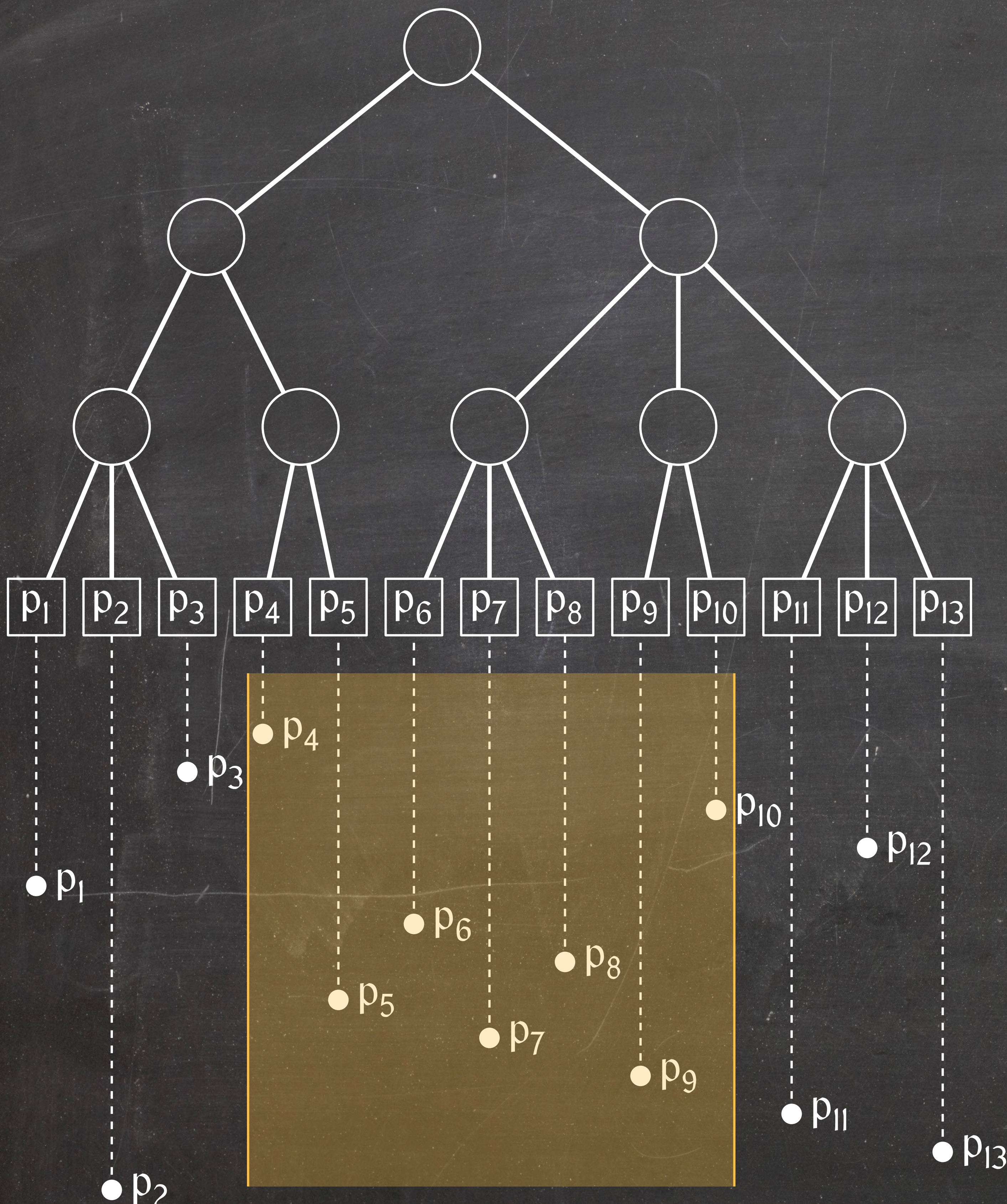
1-Dimensional Range Reporting ((a, b)-Tree)



1-Dimensional Range Reporting is just a standard RangeFind query.

Query cost: $O(\lg n + k)$

1-Dimensional Range Reporting ((a, b)-Tree)

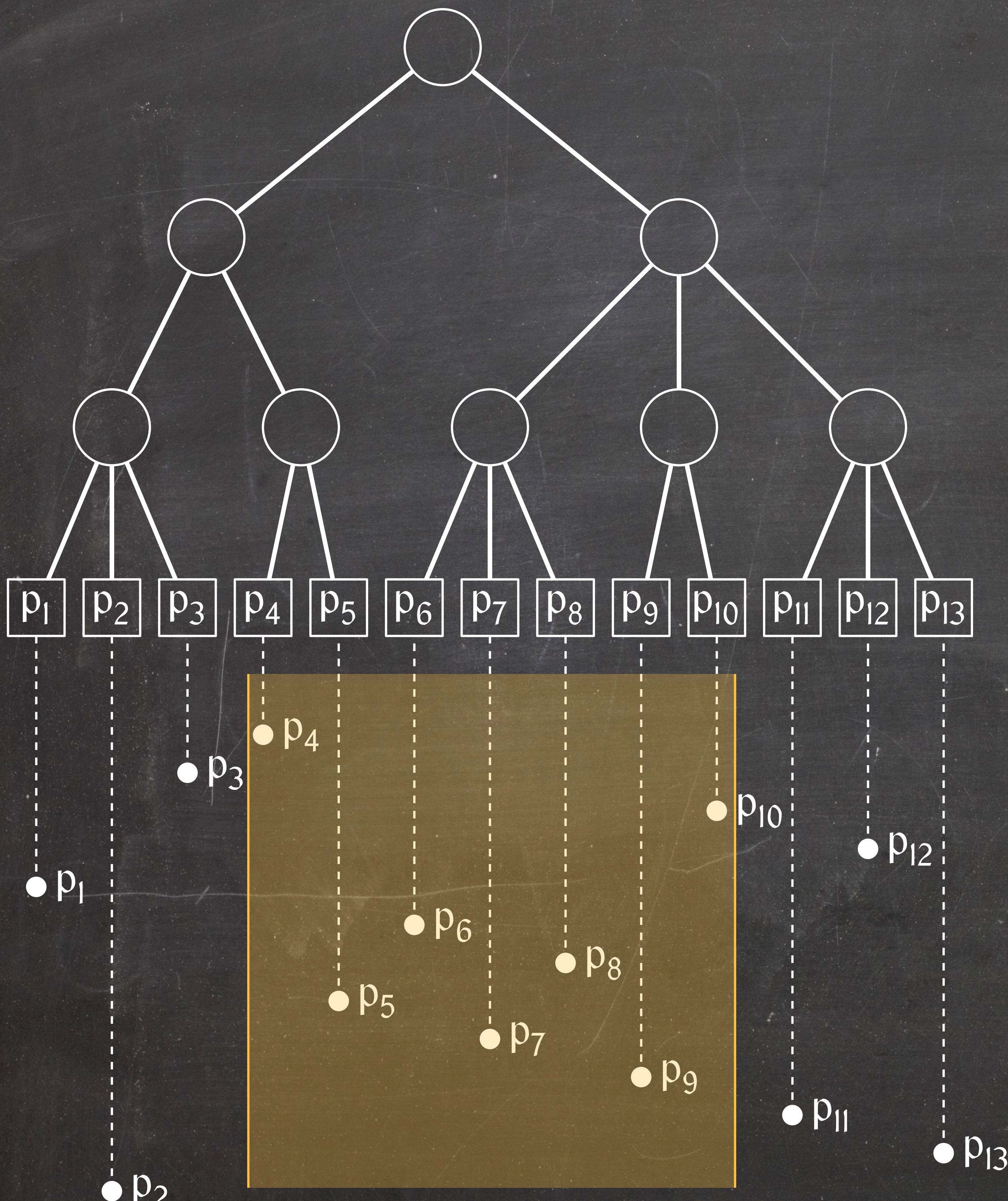


1-Dimensional Range Reporting is just a standard RangeFind query.

Query cost: $O(\lg n + k)$

Data structure size: $O(n)$

1-Dimensional Range Reporting ((a, b)-Tree)



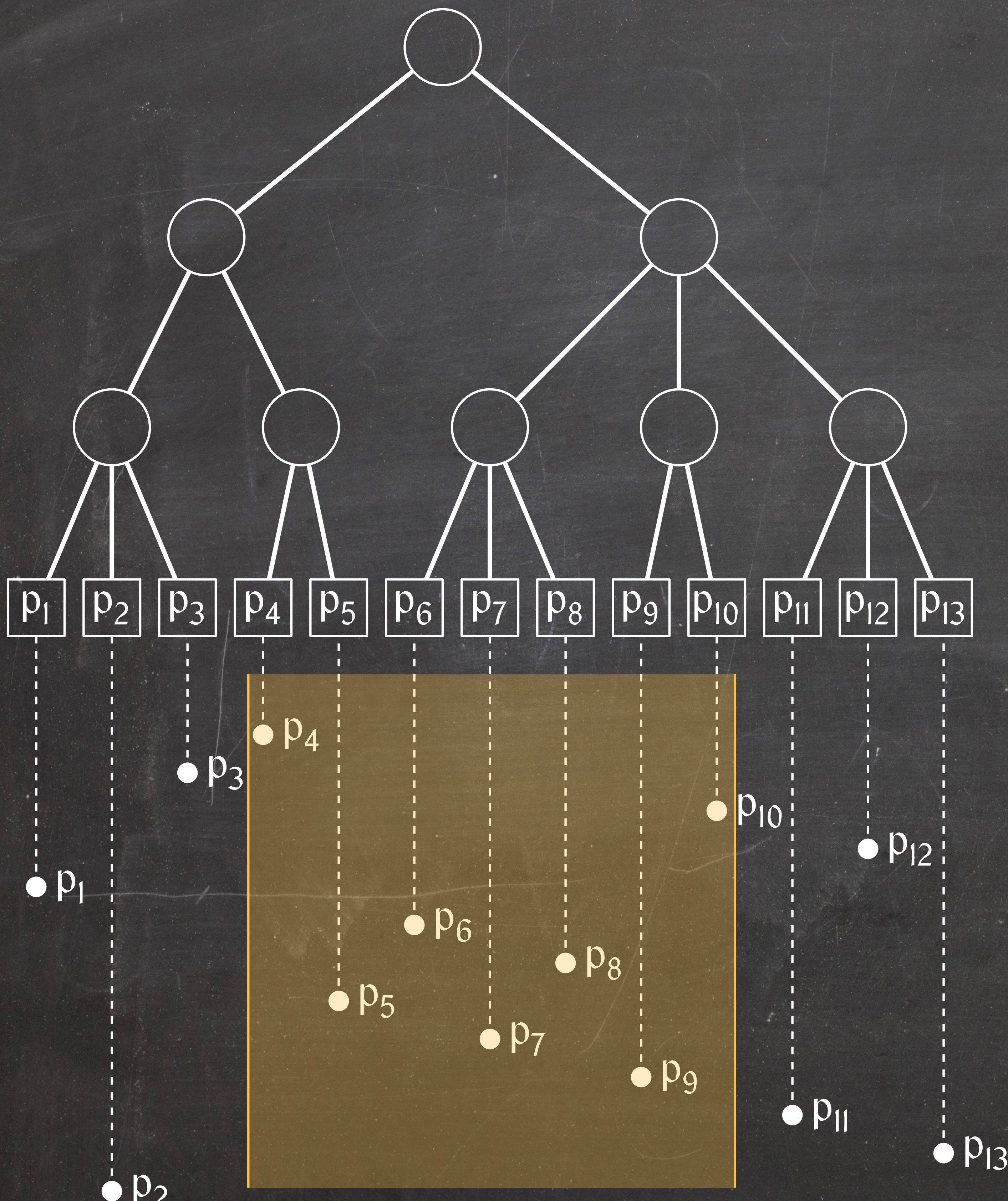
1-Dimensional Range Reporting is just a standard RangeFind query.

Query cost: $O(\lg n + k)$

Data structure size: $O(n)$

Construction cost: $O(n \lg n)$

1-Dimensional Range Reporting ((a, b)-Tree)



1-Dimensional Range Reporting is just a standard RangeFind query.

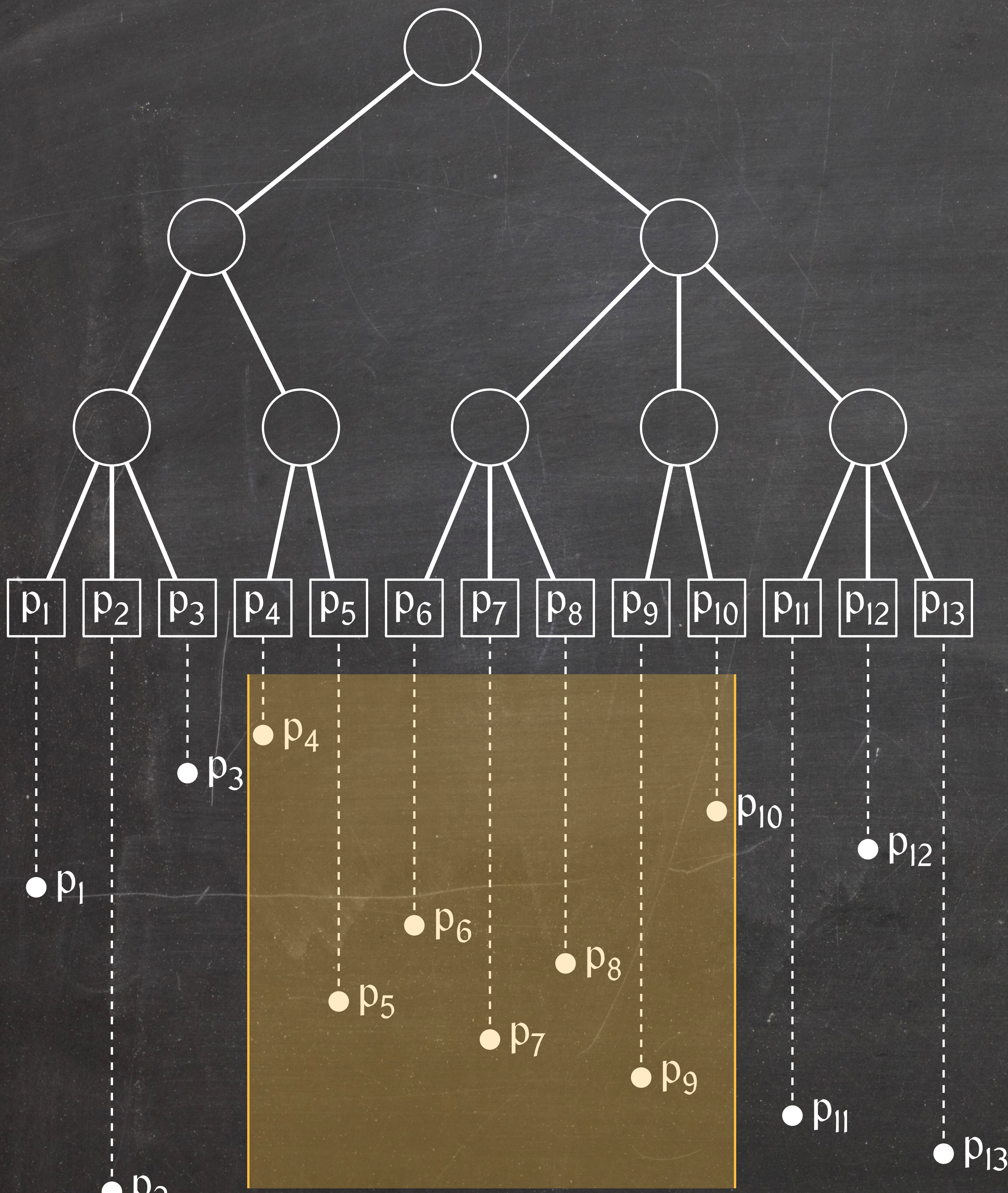
Query cost: $O(\lg n + k)$

Data structure size: $O(n)$

Construction cost: $O(n \lg n)$

- Using n Insert operations

1-Dimensional Range Reporting ((a, b)-Tree)



1-Dimensional Range Reporting is just a standard RangeFind query.

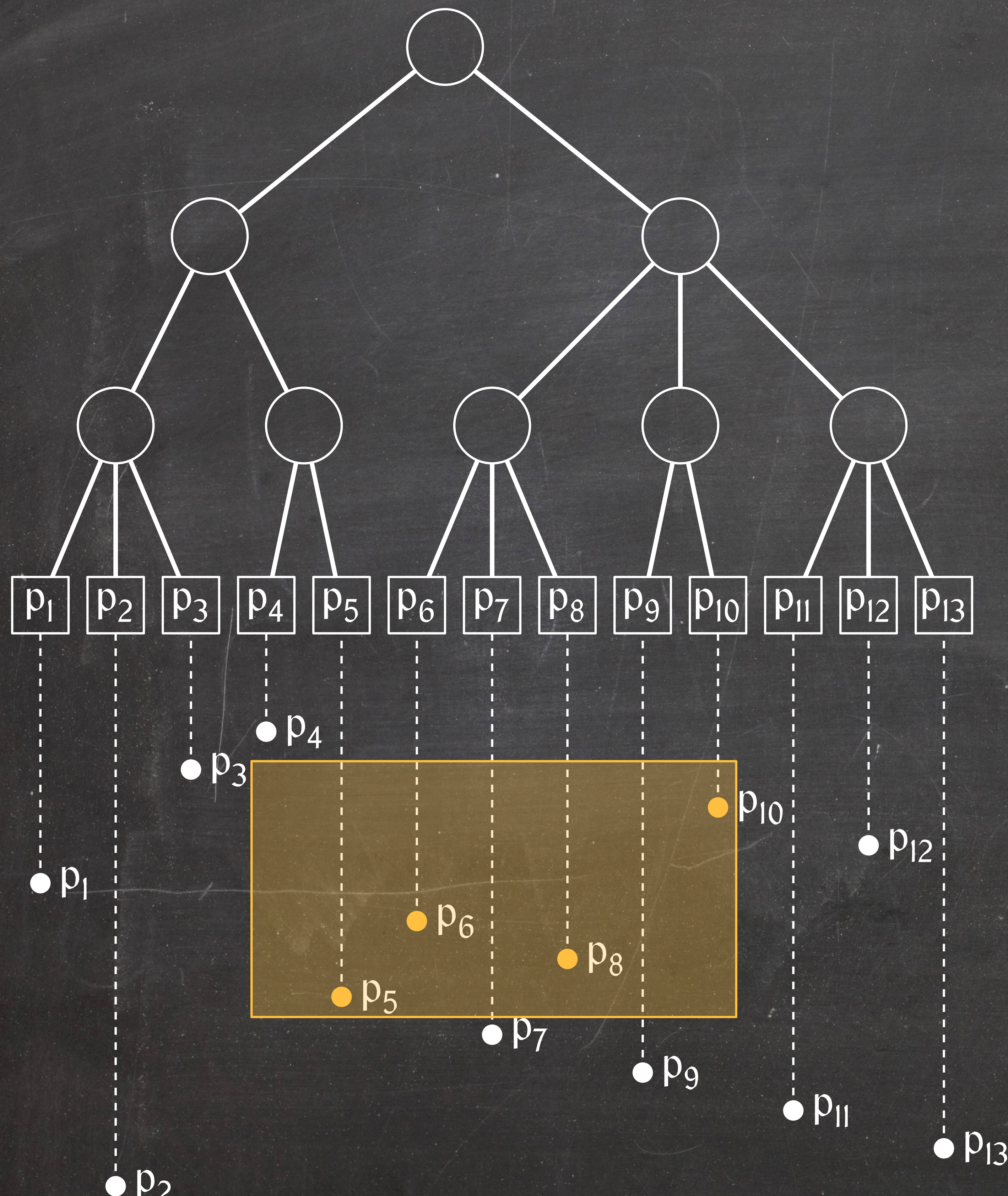
Query cost: $O(\lg n + k)$

Data structure size: $O(n)$

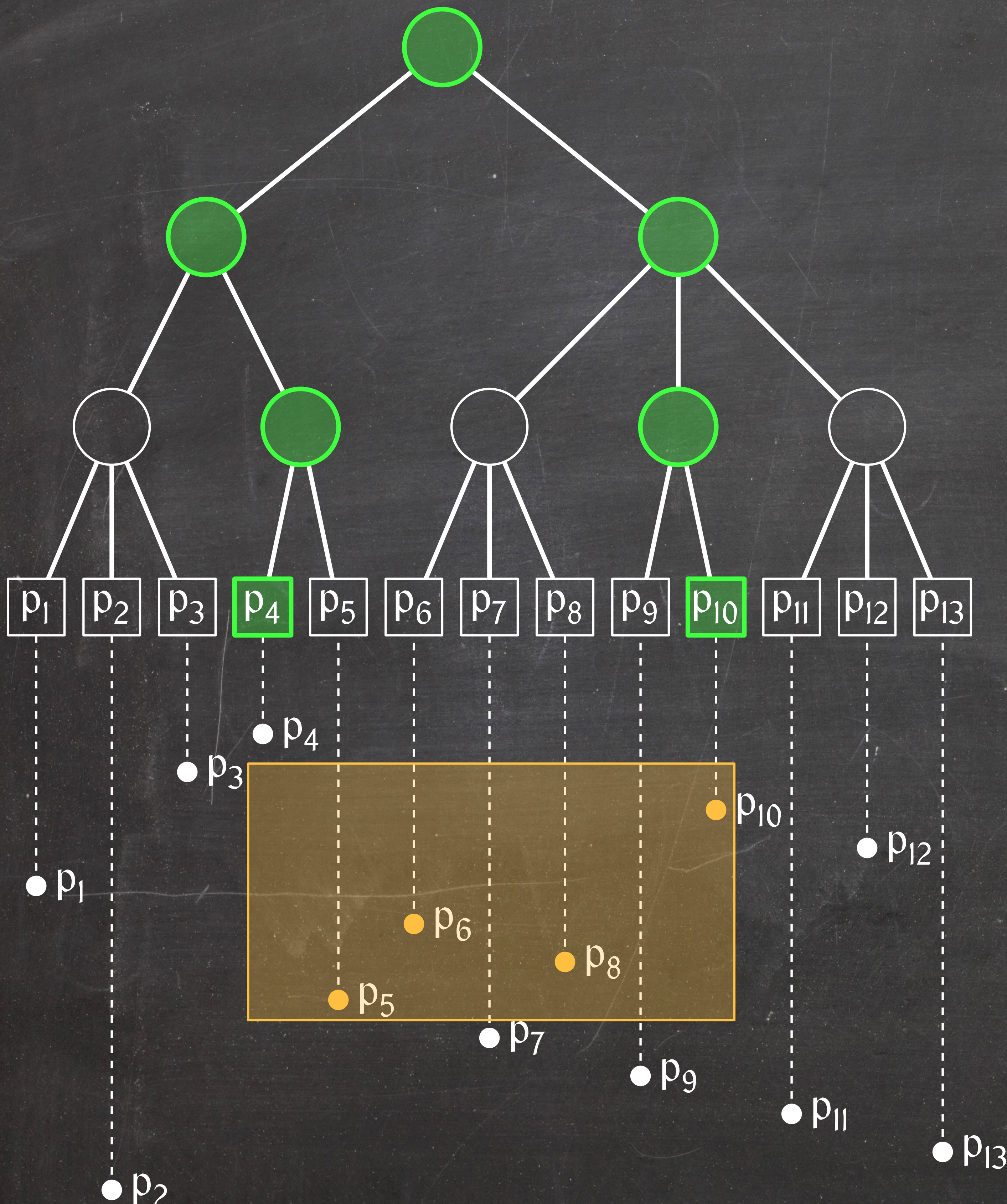
Construction cost: $O(n \lg n)$

- Using n Insert operations
- Sort the points and then build the tree bottom-up in $O(n)$ time!

2-Dimensional Range Reporting (2-d Range Tree)

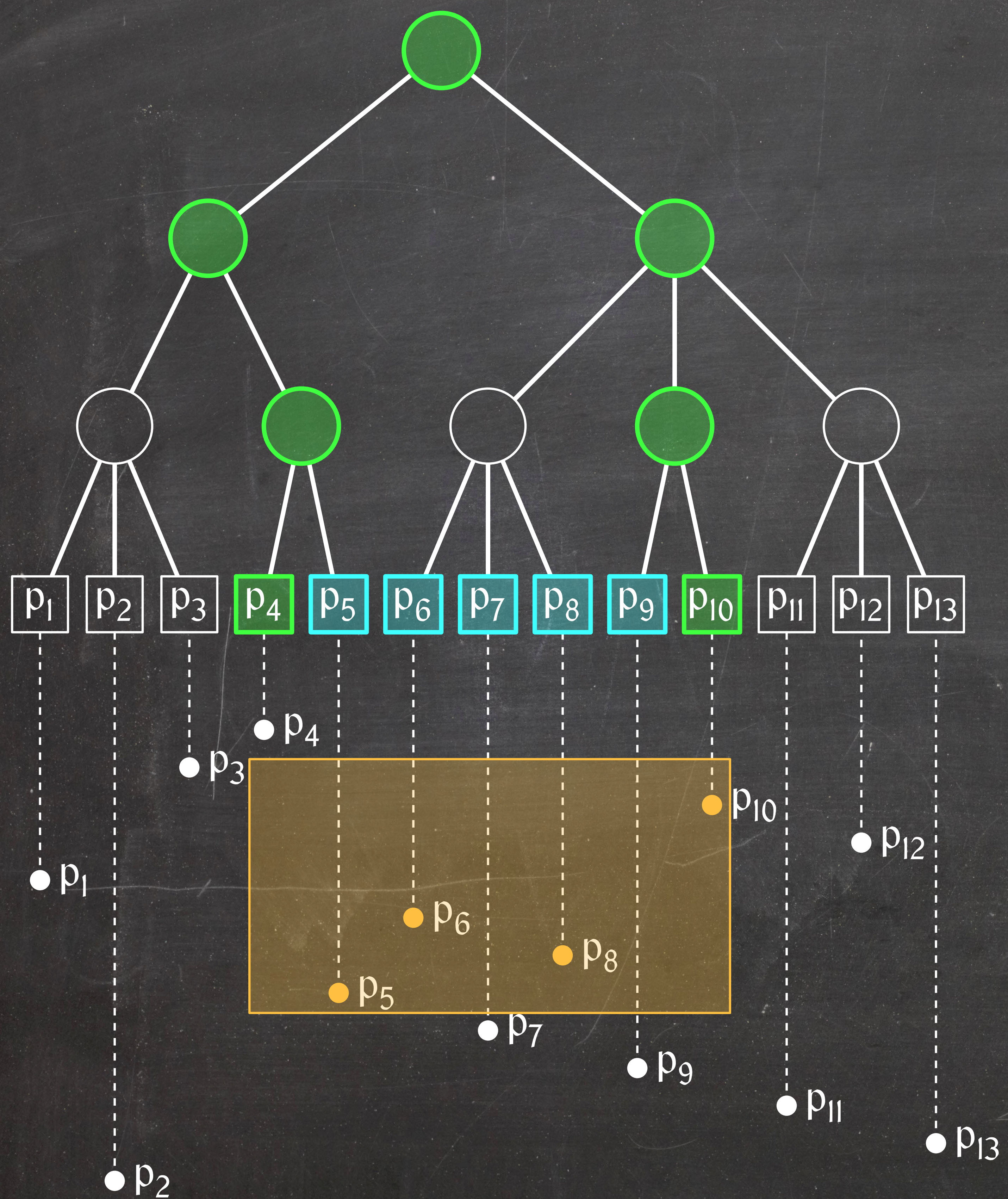


2-Dimensional Range Reporting (2-d Range Tree)



The leftmost and rightmost leaf in the x-range are easy to locate and check in O(lg n) time.

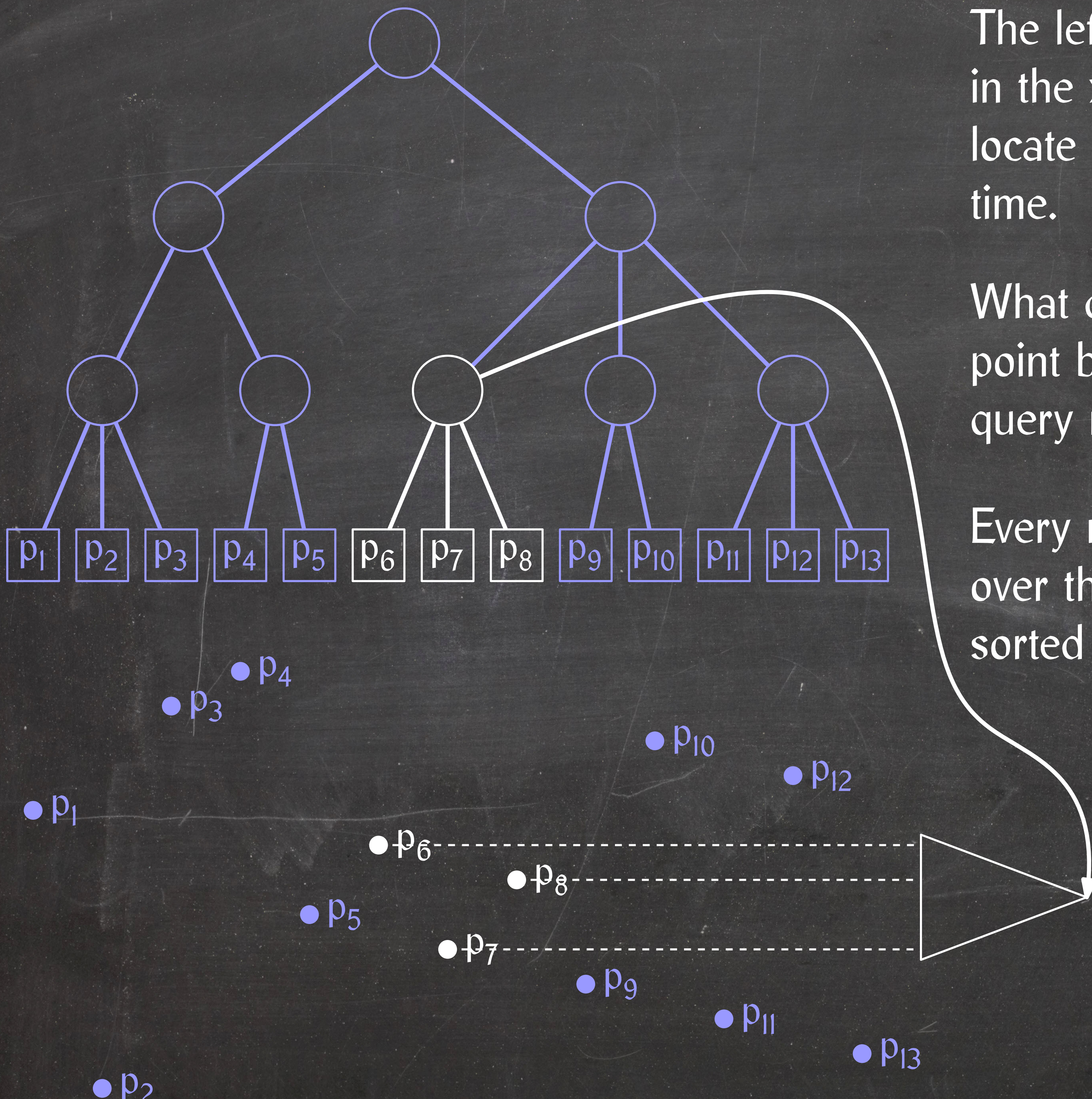
2-Dimensional Range Reporting (2-d Range Tree)



The leftmost and rightmost leaf in the x-range are easy to locate and check in O(lg n) time.

What determines whether any point between them is in the query range?

2-Dimensional Range Reporting (2-d Range Tree)

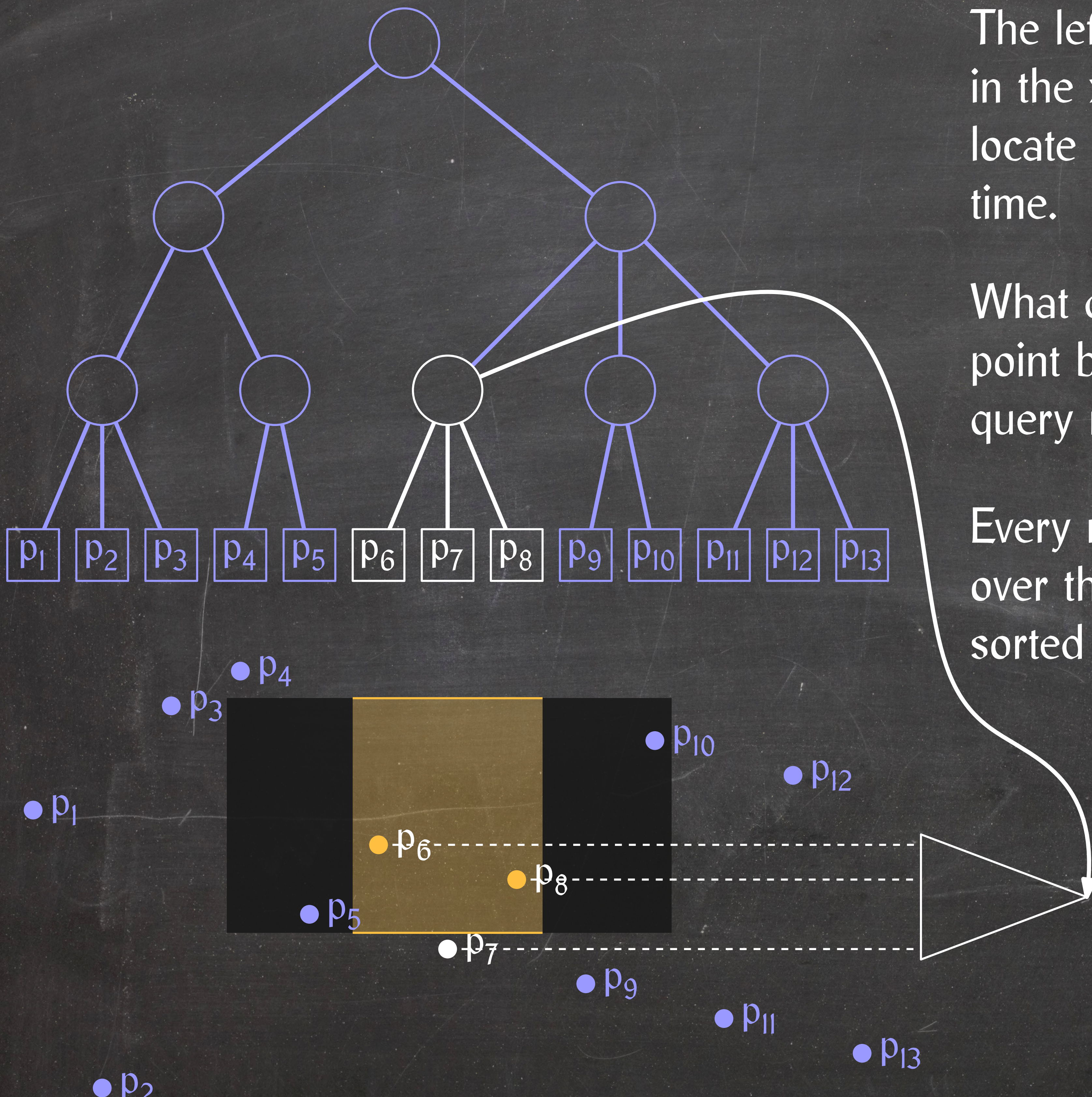


The leftmost and rightmost leaf in the x-range are easy to locate and check in $O(\lg n)$ time.

What determines whether any point between them is in the query range?

Every node stores an (a, b)-tree over the points in its subtree, sorted by their y-coordinates.

2-Dimensional Range Reporting (2-d Range Tree)

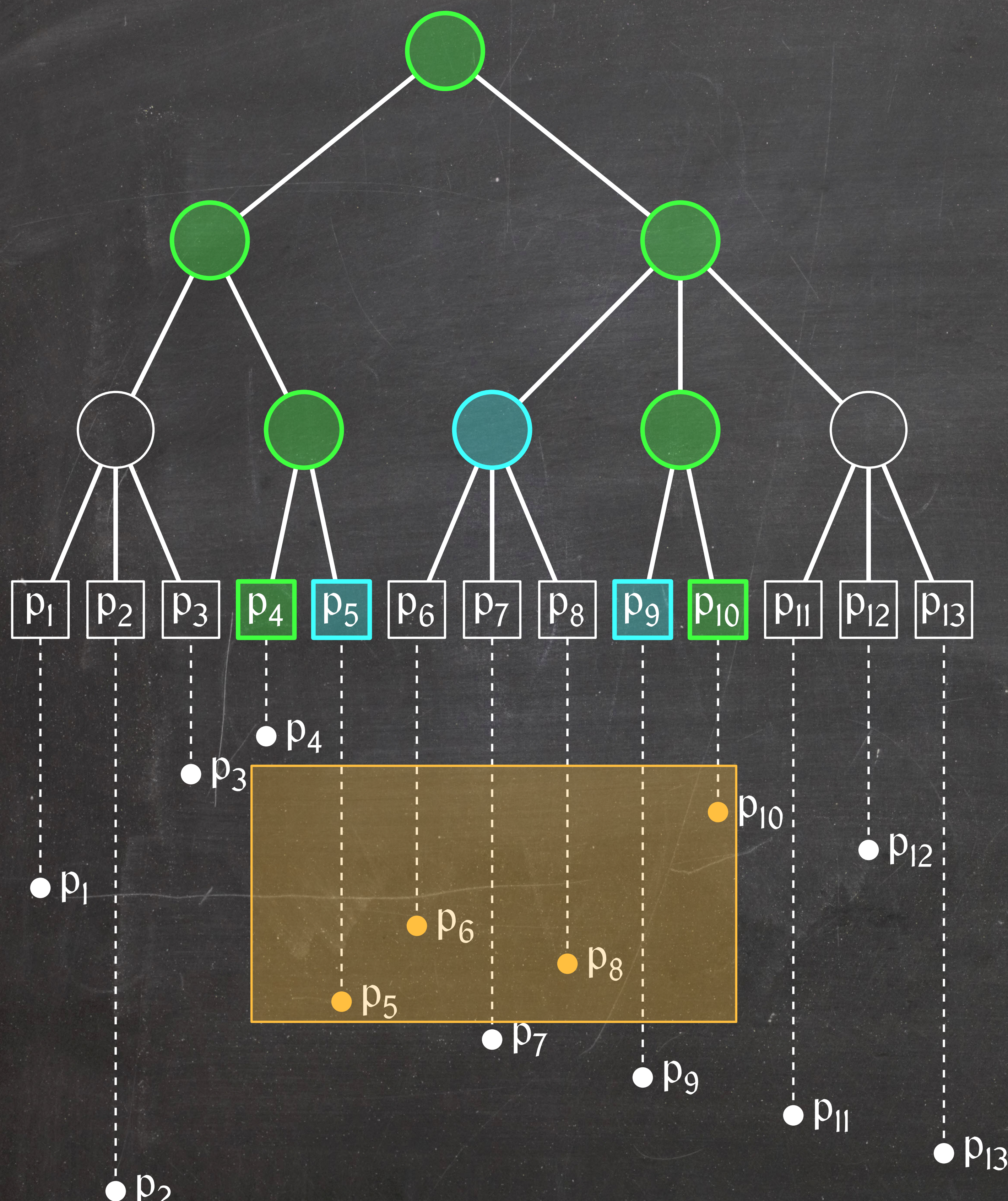


The leftmost and rightmost leaf in the x-range are easy to locate and check in O(lg n) time.

What determines whether any point between them is in the query range?

Every node stores an (a, b)-tree over the points in its subtree, sorted by their y-coordinates.

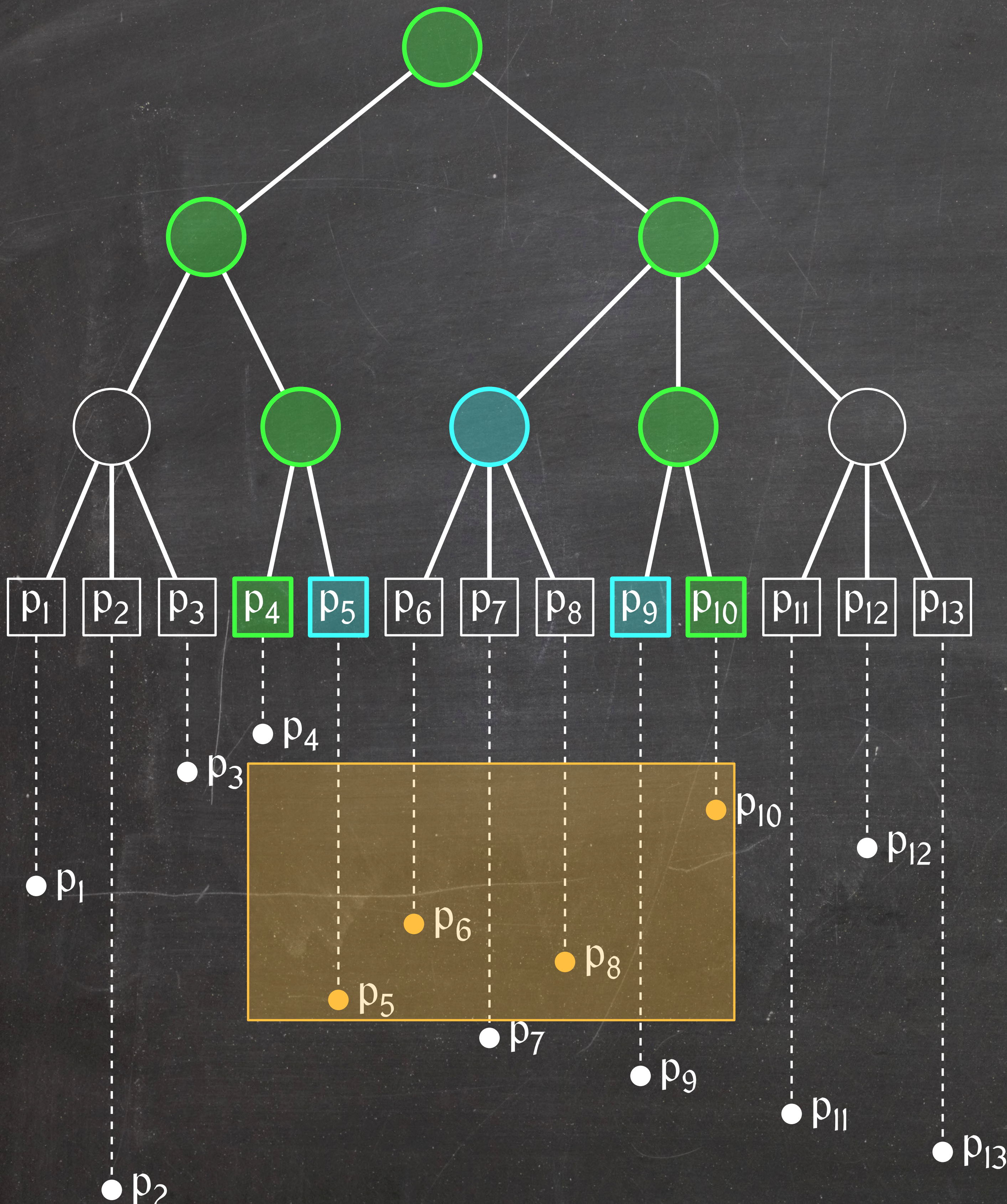
2-Dimensional Range Reporting (2-d Range Tree)



Query cost: $O(\lg^2 n + k)$

- $O(\lg n)$ RangeFind queries of cost $O(\lg n + k')$

2-Dimensional Range Reporting (2-d Range Tree)



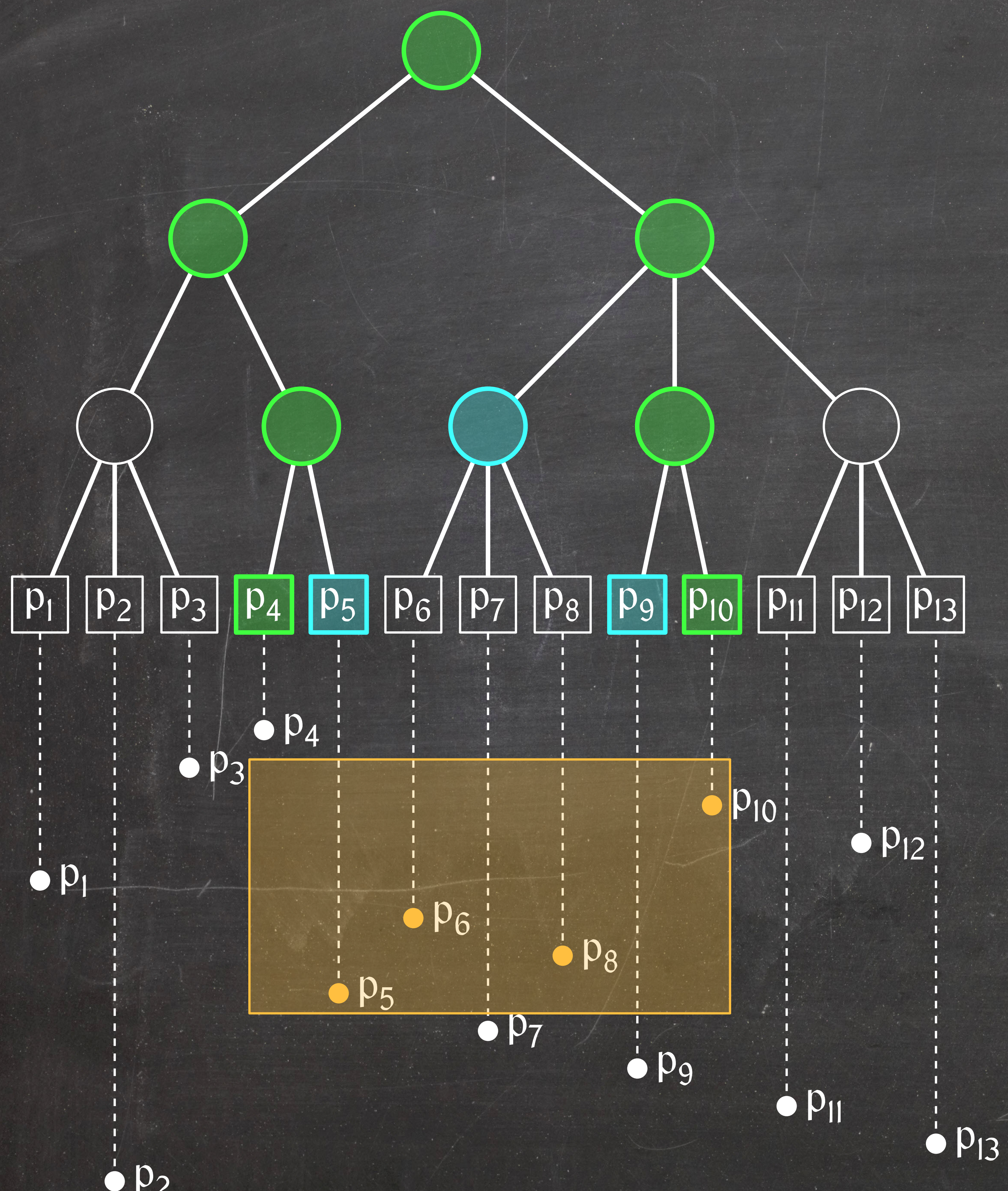
Query cost: $O(\lg^2 n + k)$

- $O(\lg n)$ RangeFind queries of cost $O(\lg n + k')$

Data structure size: $O(n \lg n)$

- Every point is stored in $O(\lg n)$ secondary trees

2-Dimensional Range Reporting (2-d Range Tree)



Query cost: $O(\lg^2 n + k)$

- $O(\lg n)$ RangeFind queries of cost $O(\lg n + k')$

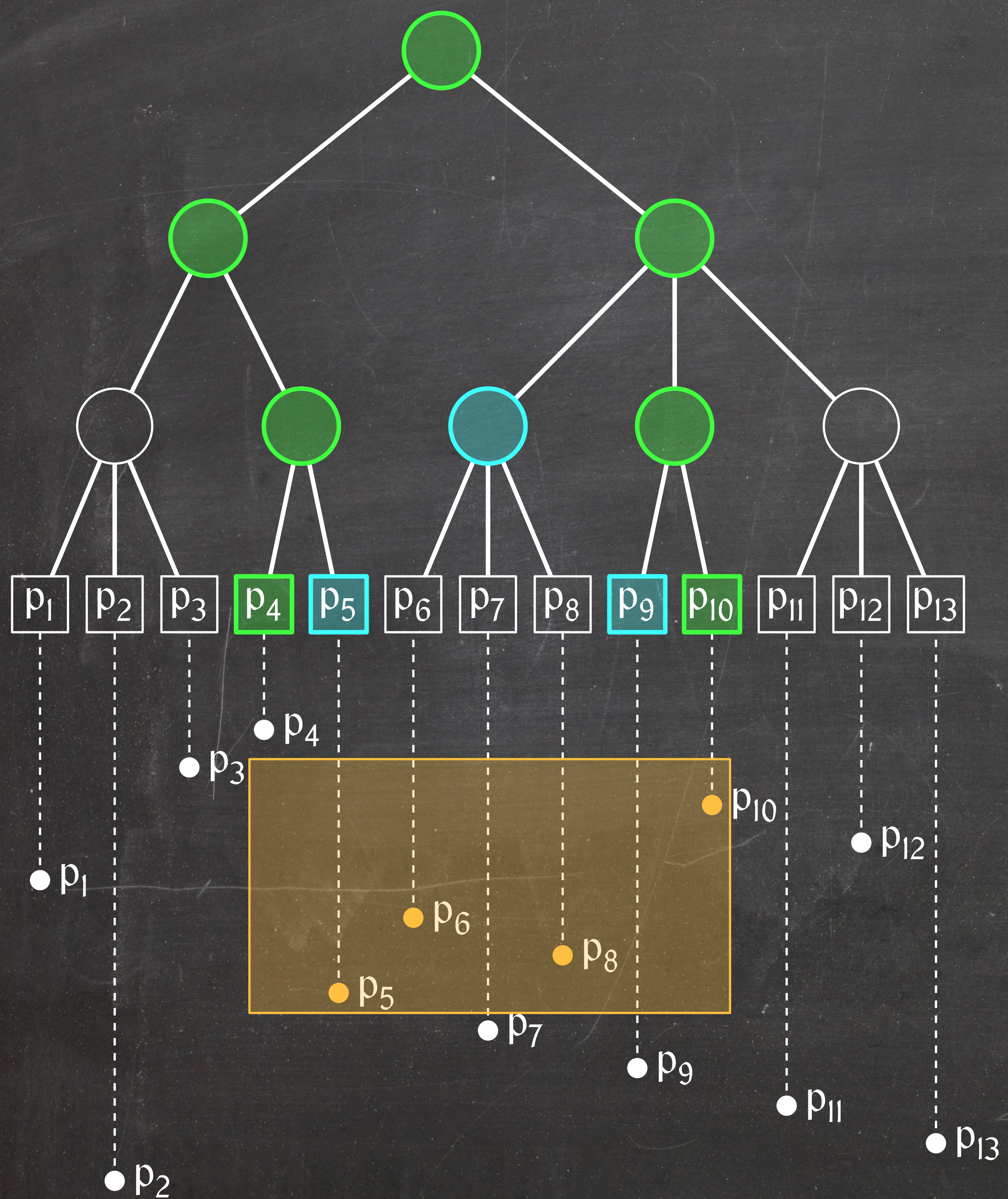
Data structure size: $O(n \lg n)$

- Every point is stored in $O(\lg n)$ secondary trees

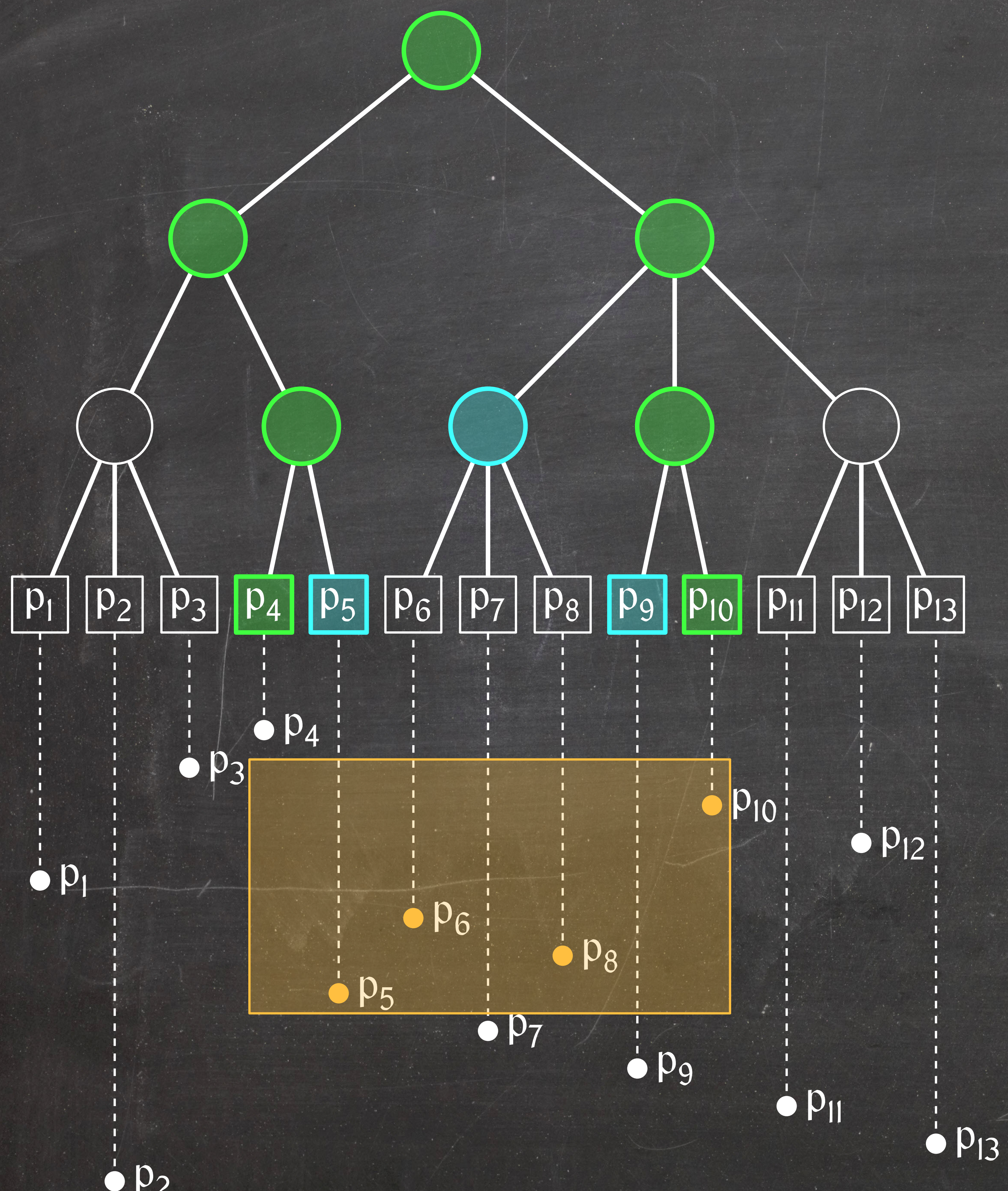
Construction cost: $O(n \lg n)$

- Sort points by x-coordinates.
- Build y-sorted point list for each node using bottom-up merging.
- Build each secondary tree in linear time.

d-Dimensional Range Reporting (d-d Range Tree)



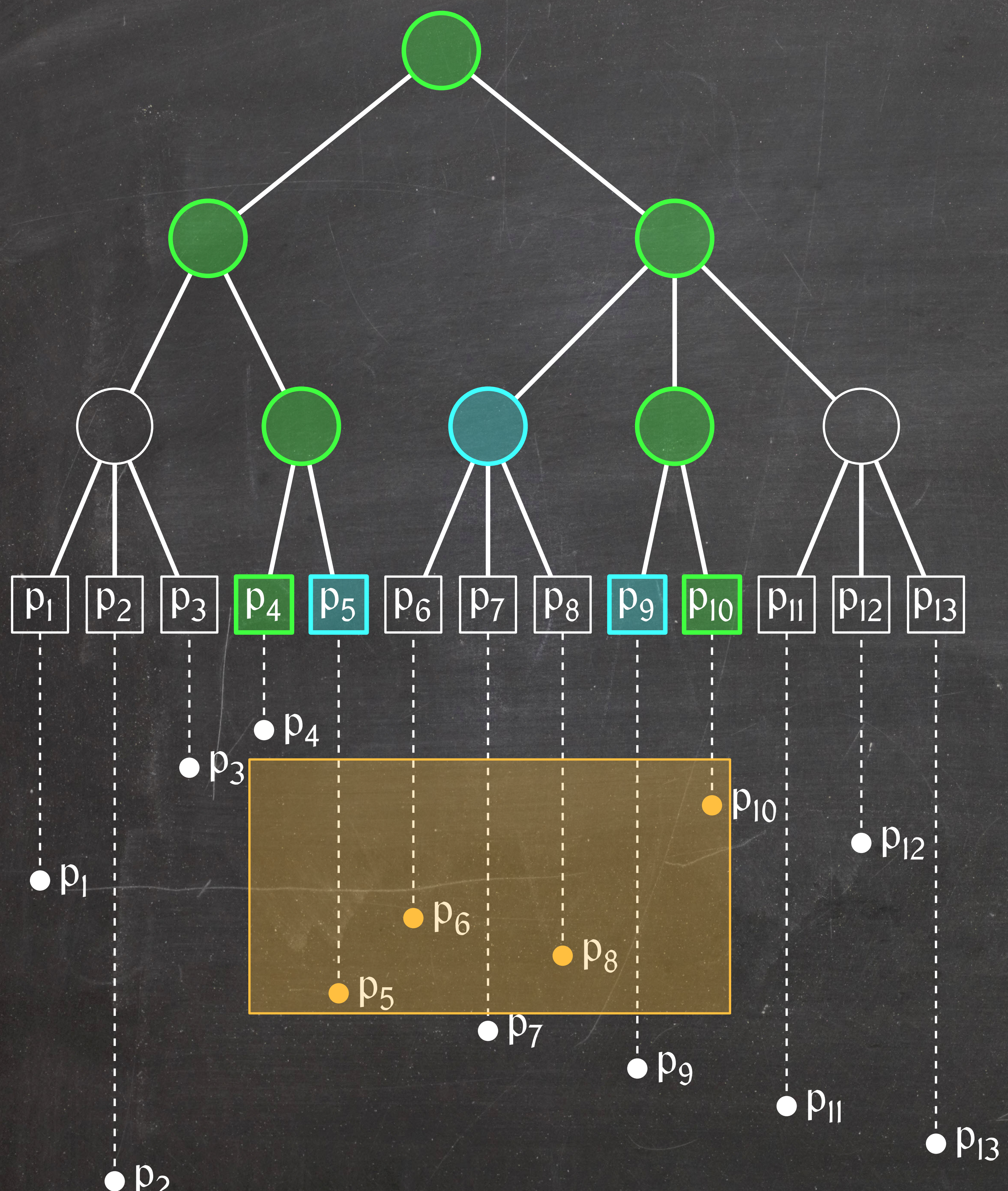
d-Dimensional Range Reporting (d-d Range Tree)



Query cost: $O(\lg^d n + k)$

- $O(\lg n)$ $(d - 1)$-dimensional range queries of cost $O(\lg^{d-1} n + k')$

d-Dimensional Range Reporting (d-d Range Tree)



Query cost: $O(\lg^d n + k)$

- $O(\lg n)$ $(d-1)$-dimensional range queries of cost $O(\lg^{d-1} n + k')$

Data structure size and construction cost: $O(n \lg^{d-1} n)$

- Secondary $(d-1)$-dimensional range trees store $O(n \lg n)$ points in total.
- A $(d-1)$-dimensional range tree storing m points has size $O(m \lg^{d-2} m)$ and takes $O(m \lg^{d-2} m)$ time to build.

Range Trees: Summary

Theorem: A d-dimensional range tree uses $O(n \lg^{d-1} n)$ space, can be constructed in $O(n \lg^{d-1} n)$ time, and supports d-dimensional range queries in $O(\lg^d n + k)$ time.

Notes:

- Using weight-balanced (a, b)-trees, updates can be supported in $O(\lg^d n)$ amortized time.
- Using a really cool technique called fractional cascading, the query cost can be reduced to $O(\lg^{d-1} n + k)$ time.

Summary

Data structures are very powerful tools for designing efficient algorithms.

To build a new data structure, we often don't have to start from scratch.

Augmenting data structures:

- Store additional information in the tree (Rank/Select)
- Change the rules where data items are stored (Priority Search Tree)
- Store entire data structures at the node of a tree (Range Tree)
- Build recursive data structures (Range Tree)