Karlsruhe Institute of Technology

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Competitive Programming

Winter Term 23/24

Segment Trees (no Treaps yet)

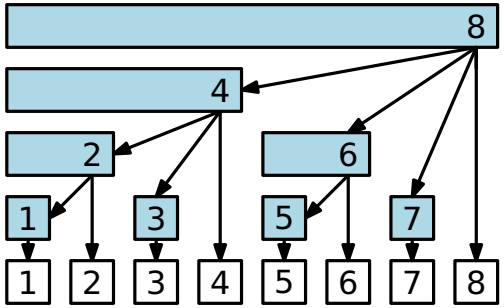
Kirill Simonov
Algorithm Engineering Group
HPI

Christopher Weyand
Scalable Algorithms Group
KIT

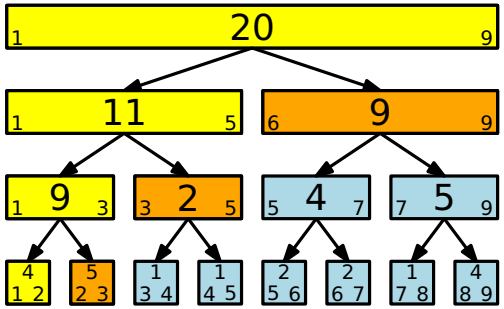
No one reads the title anyway

- today
  - reminder & lazy propagation
  - iterative segtree
  - binary search
  - nested data structures
  - implicit segtree
  - persistent segree
- odds and ends (not covered here)
  - O(n) construction
  - non-commutative combiner

Imposter!



Real Deal



https://cp-algorithms.com/data_structures/segment_tree.html

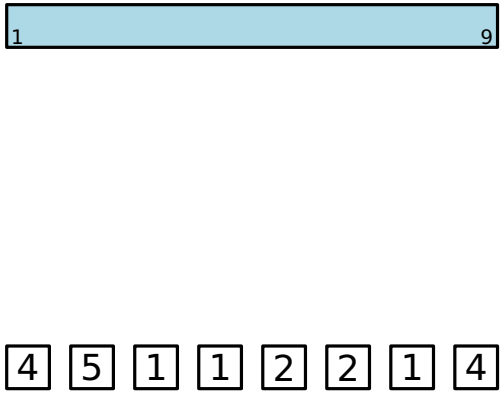
Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A

$\boxed{4}\ \boxed{5}\ \boxed{1}\ \boxed{1}\ \boxed{2}\ \boxed{2}\ \boxed{1}\ \boxed{4}$

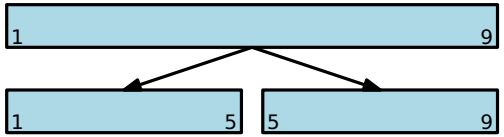
Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n + 1)$



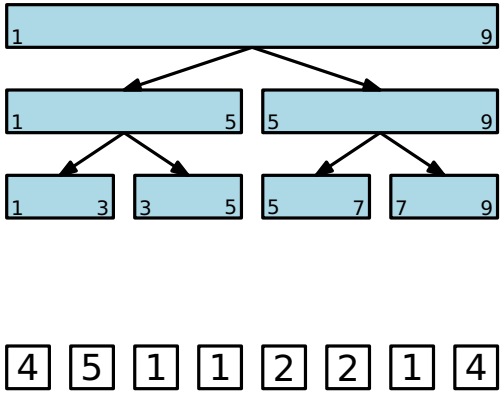
Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n + 1)$
- left child corresponds to left half of parent's range,
  right child to right half



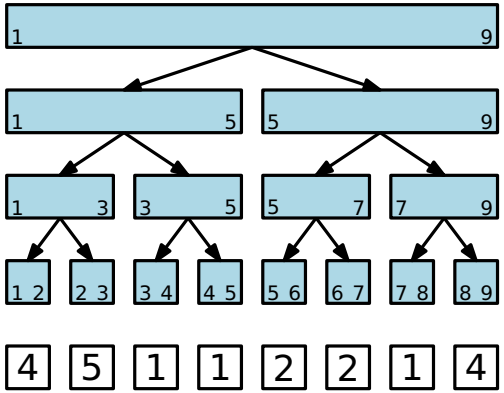
Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n + 1)$
- left child corresponds to left half of parent's range,
  right child to right half



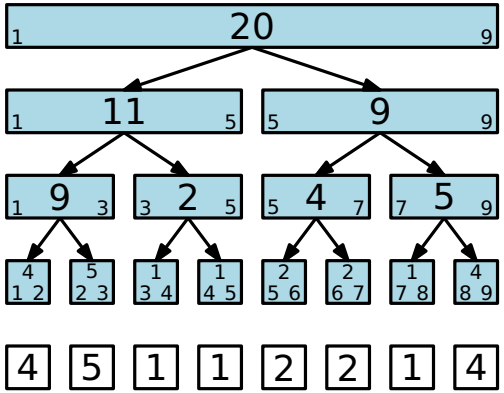
Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n + 1)$
- left child corresponds to left half of parent's range,
  right child to right half



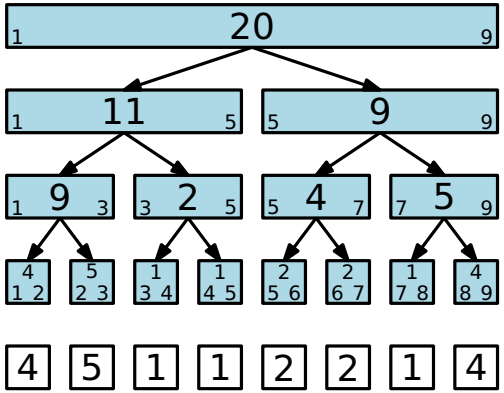
Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n + 1)$
- left child corresponds to left half of parent's range,
  right child to right half
- each node stores the aggregate over its range of values



Recap

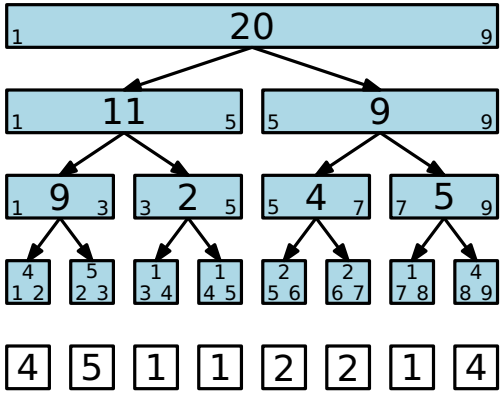
- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n + 1)$
- left child corresponds to left half of parent's range,
  right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?



Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n + 1)$
- left child corresponds to left half of parent's range,
  right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?

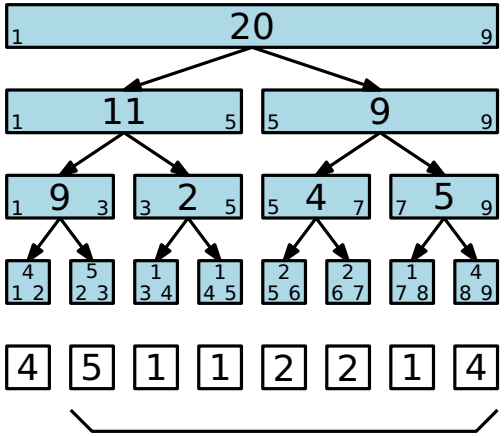
Range Queries



Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n + 1)$
- left child corresponds to left half of parent's range,
  right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?

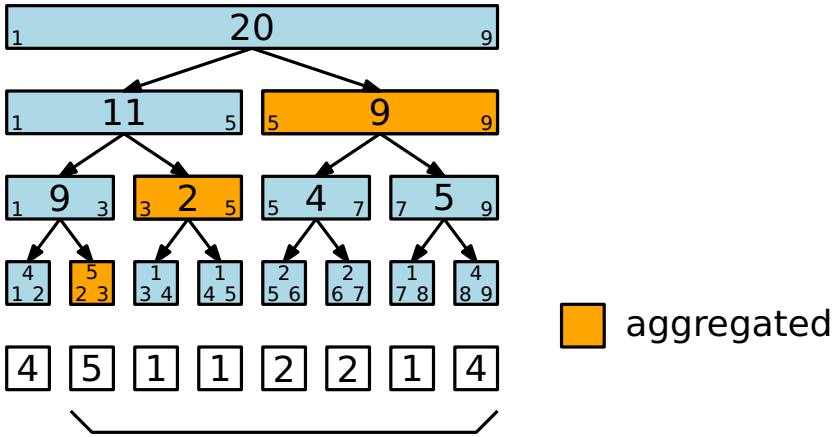
Range Queries



Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n + 1)$
- left child corresponds to left half of parent's range,
  right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?

Range Queries

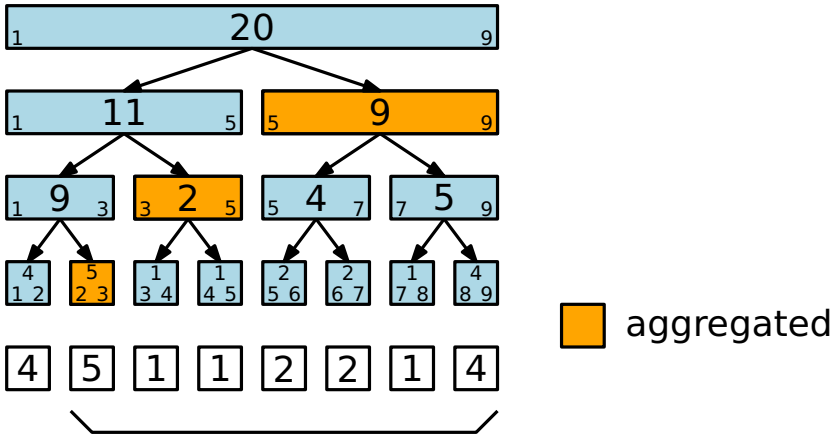


Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n+1)$
- left child corresponds to left half of parent's range,
  right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?

Range Queries

- segtree splits range into at most 2 nodes per layer $\Rightarrow \mathcal{O}(\log n)$

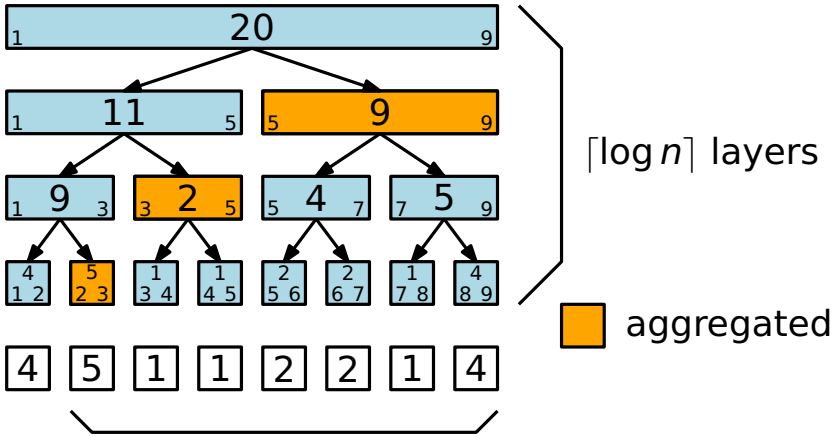


Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n+1)$
- left child corresponds to left half of parent's range,
  right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?

Range Queries

- segtree splits range into at most 2 nodes per layer $\Rightarrow \mathcal{O}(\log n)$



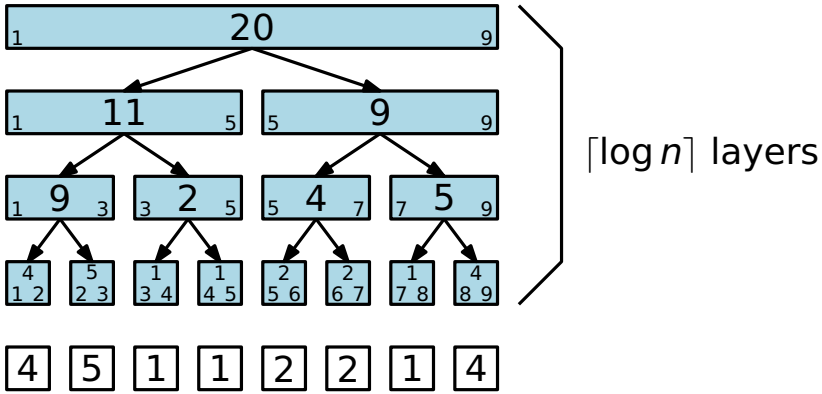
Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n+1)$
- left child corresponds to left half of parent's range,
  right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?

Range Queries

- segtree splits range into at most 2 nodes per layer $\Rightarrow \mathcal{O}(\log n)$

Point Updates



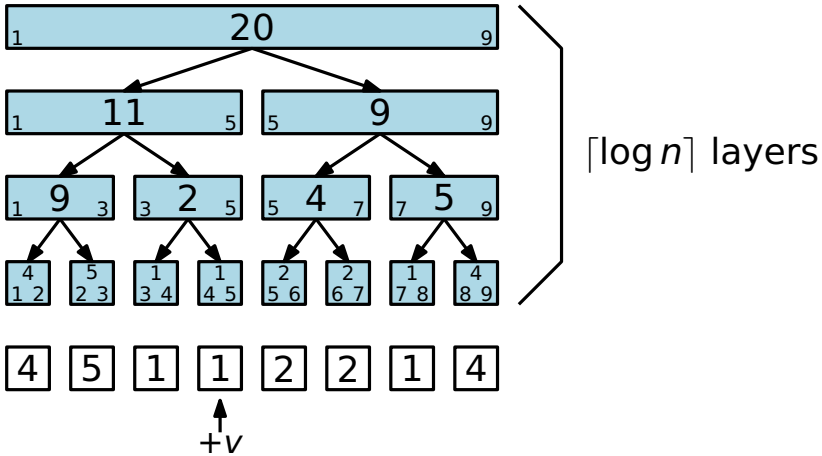
Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n+1)$
- left child corresponds to left half of parent's range, right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?

Range Queries

- segtree splits range into at most 2 nodes per layer $\Rightarrow \mathcal{O}(\log n)$

Point Updates



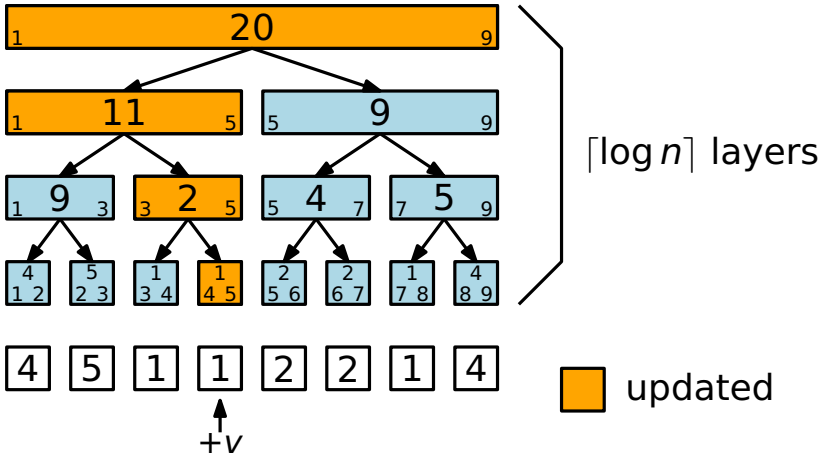
Recap

- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n+1)$
- left child corresponds to left half of parent's range, right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?

Range Queries

- segtree splits range into at most 2 nodes per layer $\Rightarrow \mathcal{O}(\log n)$

Point Updates



Recap

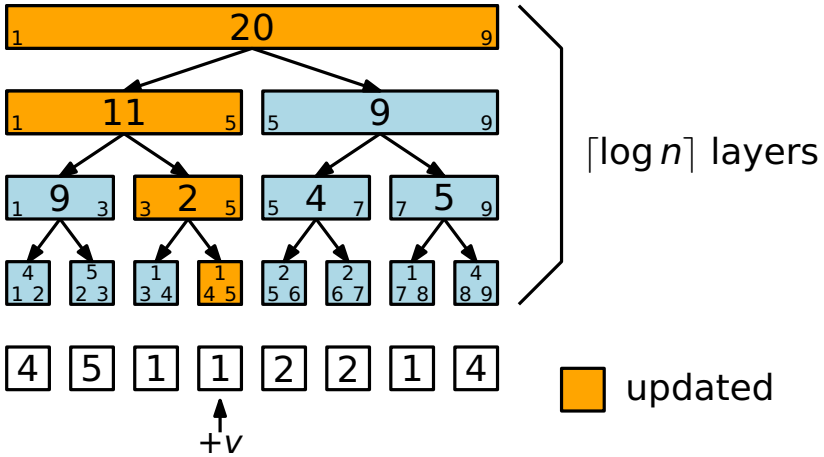
- binary tree where each node corresponds to a segment $[l, r)$ of some array A
- root corresponds to $[1, n+1)$
- left child corresponds to left half of parent's range, right child to right half
- each node stores the aggregate over its range of values
- how does this structure help us?

Range Queries

- segtree splits range into at most 2 nodes per layer $\Rightarrow \mathcal{O}(\log n)$

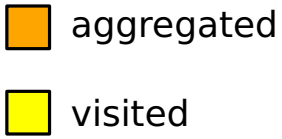
Point Updates

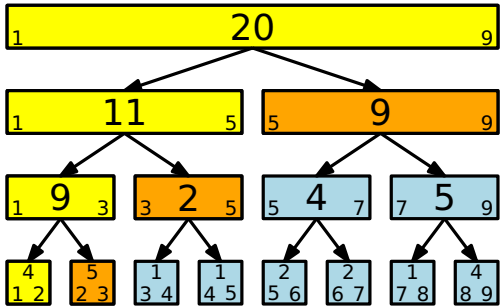
- 1 node per layer $\Rightarrow \mathcal{O}(\log n)$



Recap

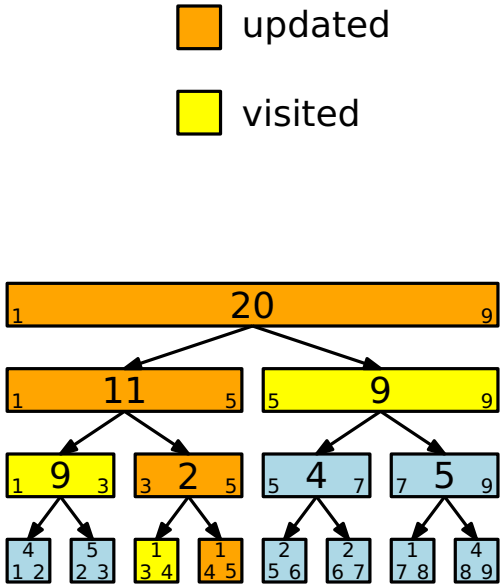
```
range query(node, l, r, q_l, q_r)
    if [l, r) ⊆ [q_l, q_r)
        return node value
    if [l, r) ∩ [q_l, q_r) = ∅
        return 0
    m = ⌊(l + r)/2⌋
    return range query(left child, l, m, q_l, q_r)+
                range query(right child, m, r, q_l, q_r)
```





Recap

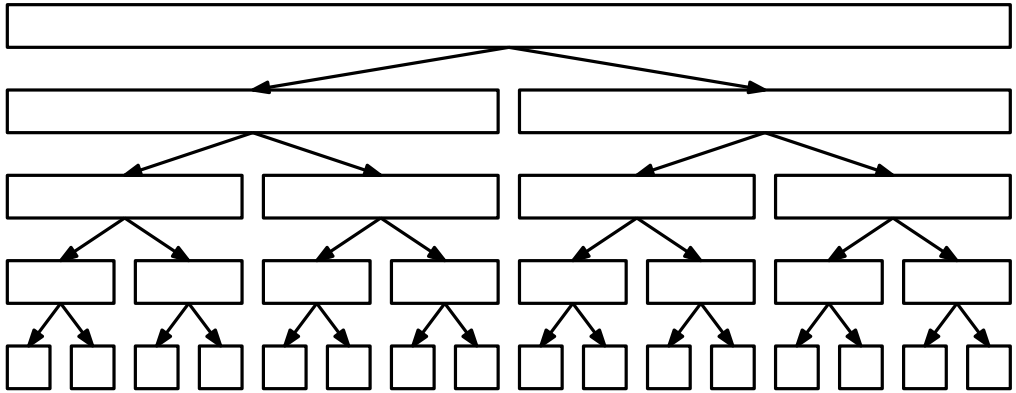
```
point update(node, l, r, i, v)
    if i ∉ [l, r)
        return
    if l = r − 1
        node value += v
        return
    m = ⌊(l + r)/2⌋
    point update(left child, l, m, i, v)
    point update(right child, m, r, i, v)
    node value = left value + right value
```



Lazy Propagation

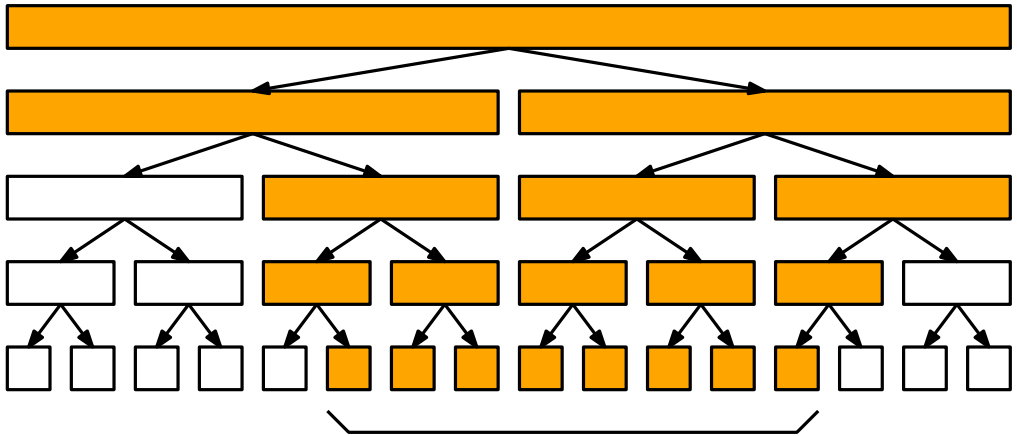
- how would we do range updates?

e.g. adding 1337 to all values in a range



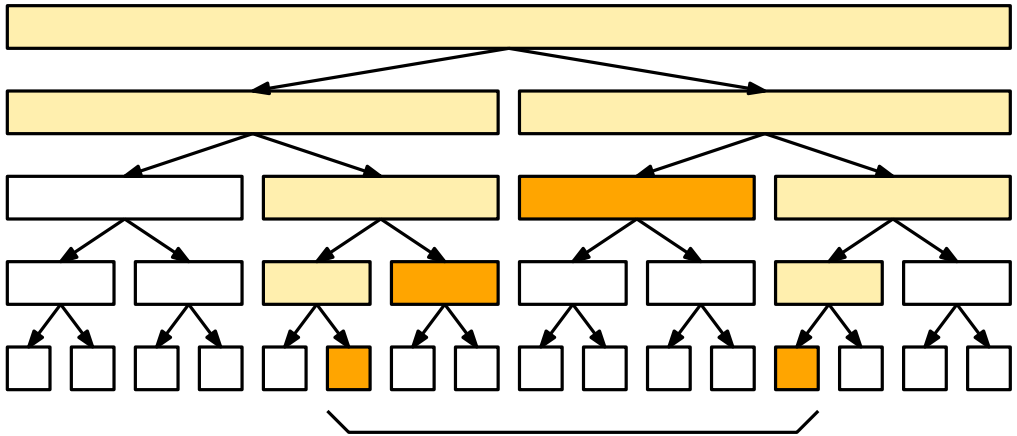
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$



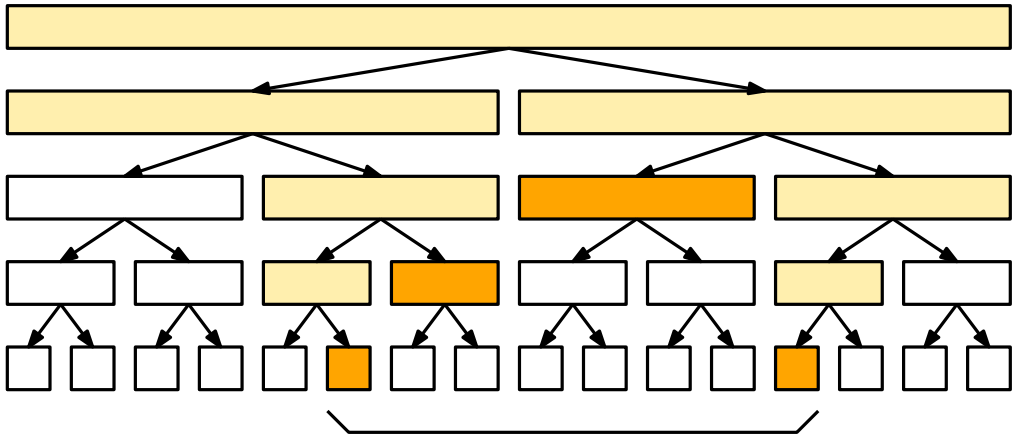
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?



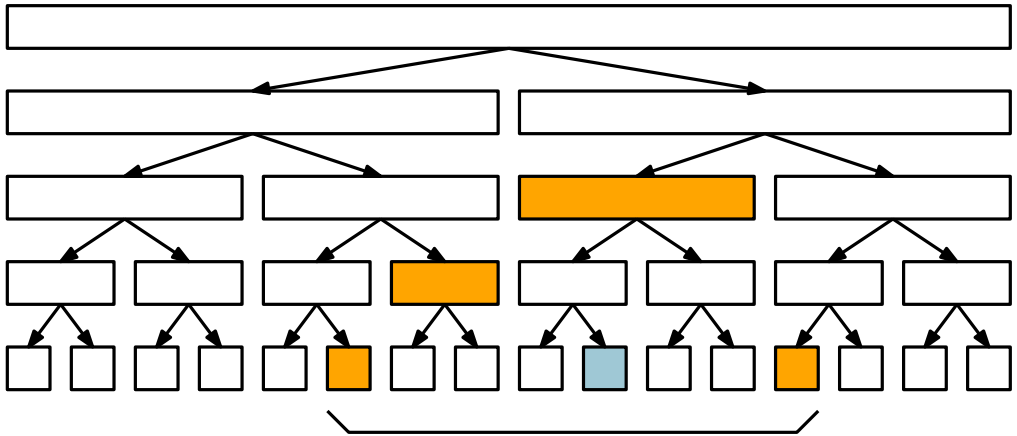
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated



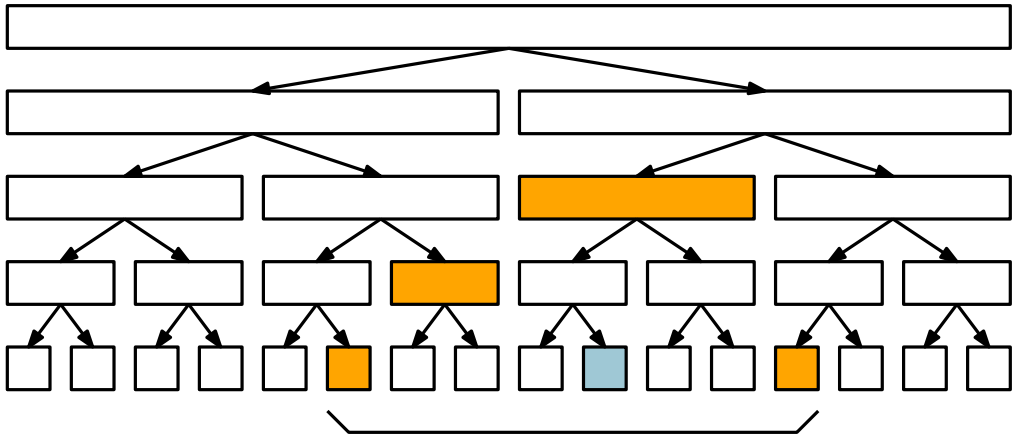
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated
- what happens when a later query accesses nodes below?



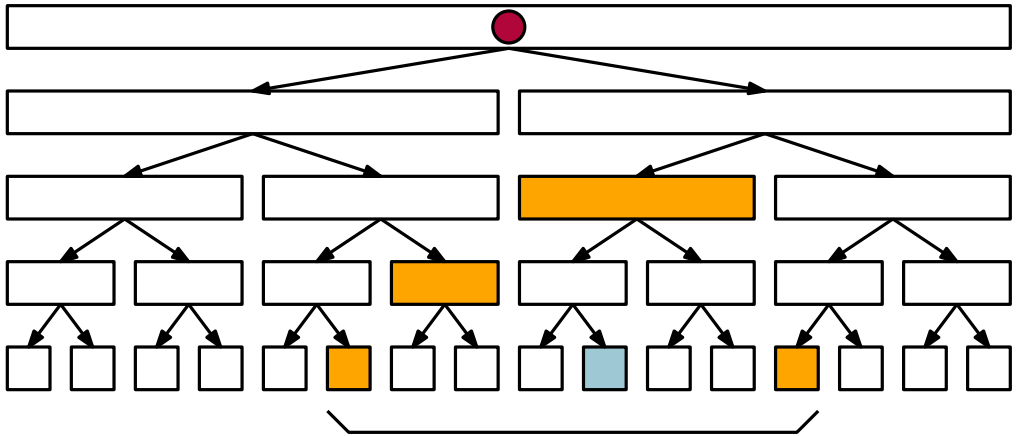
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated
- what happens when a later query accesses nodes below?
- we push the update reminder down



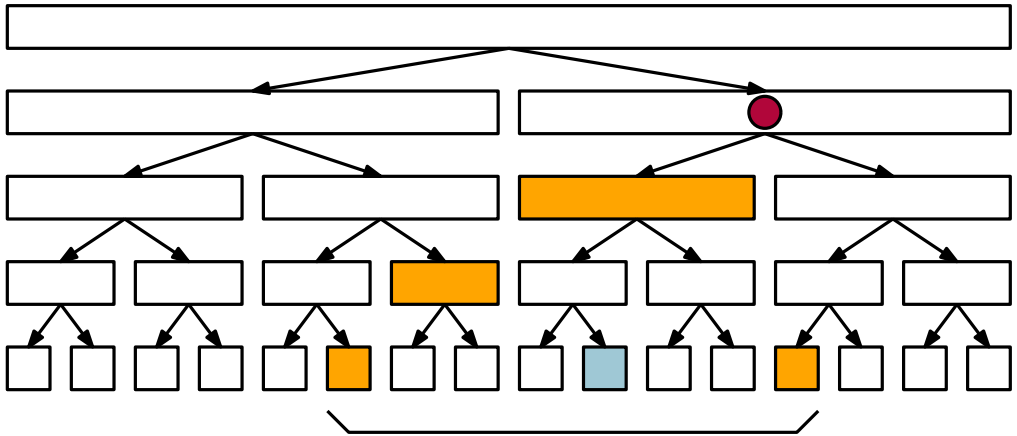
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated
- what happens when a later query accesses nodes below?
- we push the update reminder down



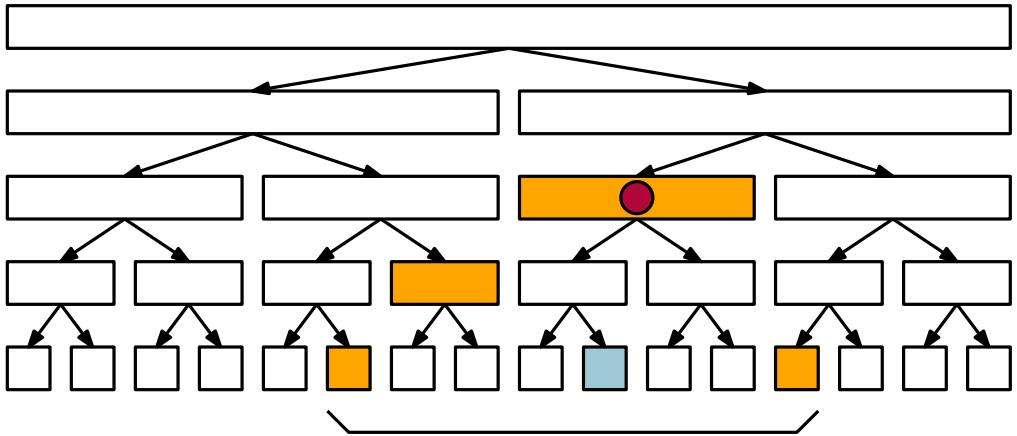
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated
- what happens when a later query accesses nodes below?
- we push the update reminder down



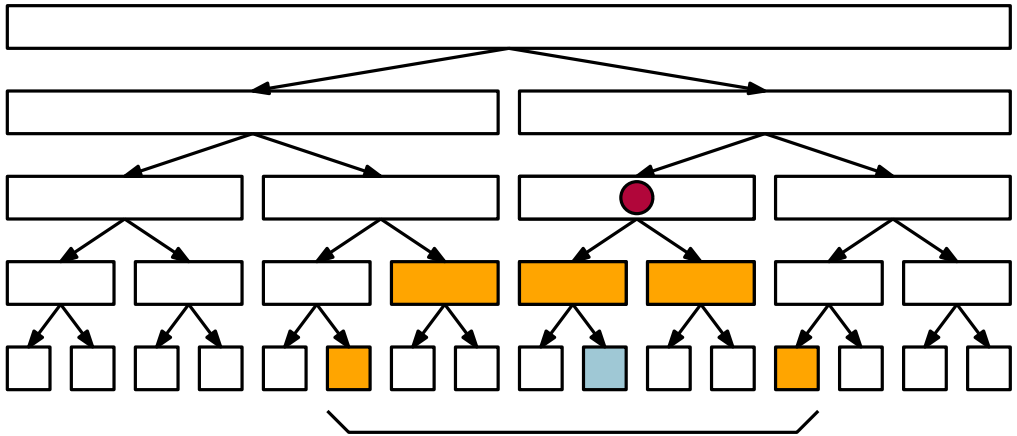
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated
- what happens when a later query accesses nodes below?
- we push the update reminder down



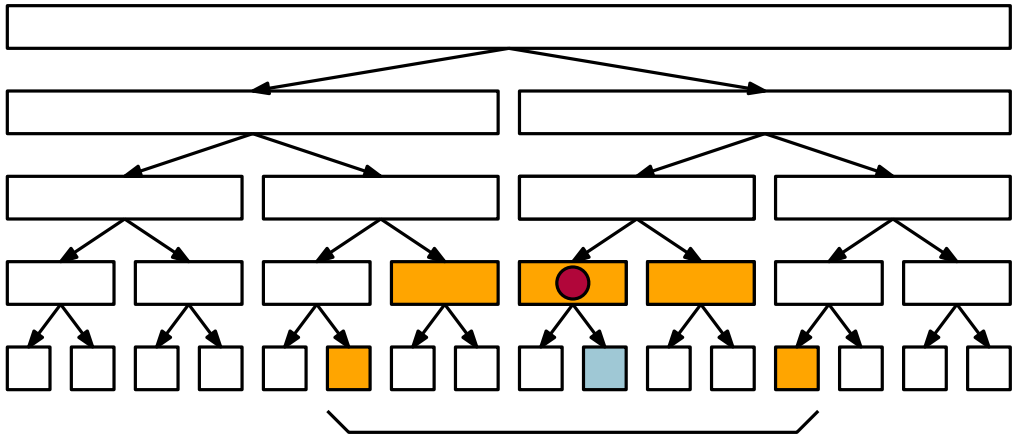
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated
- what happens when a later query accesses nodes below?
- we push the update reminder down



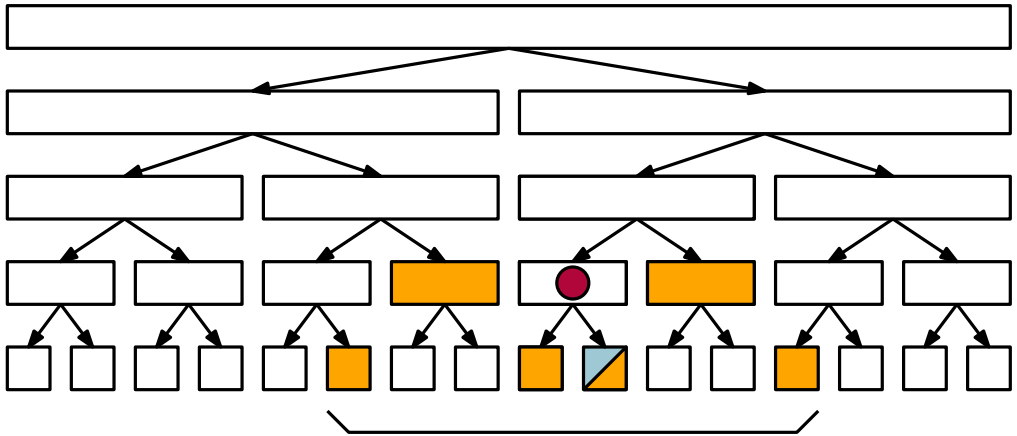
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated
- what happens when a later query accesses nodes below?
- we push the update reminder down



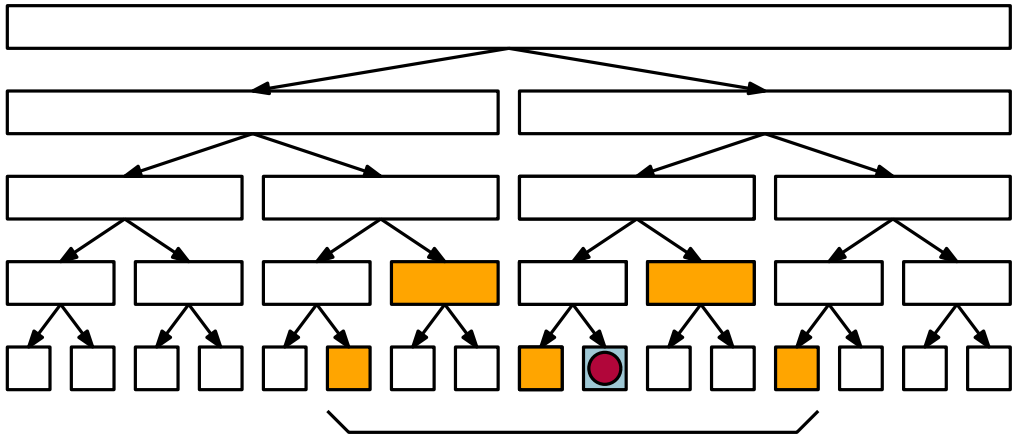
Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated
- what happens when a later query accesses nodes below?
- we push the update reminder down



Lazy Propagation

- how would we do range updates? e.g. adding 1337 to all values in a range
- we could update every node overlapping with the range $\Rightarrow \mathcal{O}(n)$
- for queries we split the range into $\log n$ nodes and sum up aggregates
  - can we do the same here (only update those and their parents)?
- in each of these nodes we remember that the subtree should be updated
- what happens when a later query accesses nodes below?
- we push the update reminder down



Lazy Propagation

- each node now contains two variables

Lazy Propagation

- each node now contains two variables

  `value`: aggregate over the range

Lazy Propagation

- each node now contains two variables

  `value`: aggregate over the range

  - here: sum of all values in range

Lazy Propagation

- each node now contains two variables

  `value`: aggregate over the range

  - here: sum of all values in range

  `lazy`: delayed update to apply to all child nodes

Lazy Propagation

- each node now contains two variables

  `value`: aggregate over the range

  - here: sum of all values in range

  `lazy`: delayed update to apply to all child nodes

  - here: value to add to every element in range

Lazy Propagation

- each node now contains two variables

  `value`: aggregate over the range

  - here: sum of all values in range

  `lazy`: delayed update to apply to all child nodes

  - here: value to add to every element in range
  - lazy update has already been applied to the node's `value`

Lazy Propagation

- each node now contains two variables

  `value`: aggregate over the range

  - here: sum of all values in range

  `lazy`: delayed update to apply to all child nodes

  - here: value to add to every element in range
  - lazy update has already been applied to the node's `value`

- before we recurse below a node, we **push** lazy updates down to its children

Lazy Propagation

- each node now contains two variables

  `value`: aggregate over the range

  - here: sum of all values in range

  `lazy`: delayed update to apply to all child nodes

  - here: value to add to every element in range
  - lazy update has already been applied to the node's `value`

- before we recurse below a node, we **push** lazy updates down to its children
  - thus, whenever we access a node, all lazy updates have been applied

Lazy Propagation

- each node now contains two variables

  `value`: aggregate over the range

  - here: sum of all values in range

  `lazy`: delayed update to apply to all child nodes

  - here: value to add to every element in range
  - lazy update has already been applied to the node's `value`

- before we recurse below a node, we **push** lazy updates down to its children
  - thus, whenever we access a node, all lazy updates have been applied

- requires to **apply** updates to aggregates

Lazy Propagation

- each node now contains two variables

  `value`: aggregate over the range

  - here: sum of all values in range

  `lazy`: delayed update to apply to all child nodes

  - here: value to add to every element in range
  - lazy update has already been applied to the node's `value`

- before we recurse below a node, we **push** lazy updates down to its children
  - thus, whenever we access a node, all lazy updates have been applied

- requires to **apply** updates to aggregates
- also, we must merge pending updates if a node gets more than one

Lazy Propagation

```
// add v to entire range of node
apply(node, l, r, v)
    value += (r − l) · v
    lazy += v
```

$[l, r)$: range of current node

Lazy Propagation

```
// add v to entire range of node
apply(node, l, r, v)
    value += (r − l) · v
    lazy += v
```

$[l, r)$: range of current node

```
// push lazy updates down to children
push(node, l, r)
    m = ⌊(l + r)/2⌋
    apply(left child, l, m, lazy)
    apply(right child, m, r, lazy)
    lazy = 0
```

Lazy Propagation

```
// add v to entire range of node
apply(node, l, r, v)
    value += (r − l) · v
    lazy += v
```

$[l, r)$: range of current node

```
// push lazy updates down to children
push(node, l, r)
    m = ⌊(l + r)/2⌋
    apply(left child, l, m, lazy)
    apply(right child, m, r, lazy)
    lazy = 0
```

```
// add v to every value in range [u_l, u_r)
update(node, l, r, u_l, u_r, v)
    if [l, r) ⊆ [u_l, u_r)
        apply(node, l, r, v)
        return
    if [l, r) ∩ [u_l, u_r) = ∅
        return
    m = ⌊(l + r)/2⌋
    push(node, l, r)
    update(left child, l, m, u_l, u_r, v)
    update(right child, m, r, u_l, u_r, v)
    value = left value + right value
```

Lazy Propagation

```
// add v to entire range of node
apply(node, l, r, v)
    value += (r − l) · v
    lazy += v
```

$[l, r)$: range of current node

```
// push lazy updates down to children
push(node, l, r)
    m = ⌊(l + r)/2⌋
    apply(left child, l, m, lazy)
    apply(right child, m, r, lazy)
    lazy = 0
```

```
// add v to every value in range [u_l, u_r)
update(node, l, r, u_l, u_r, v)
    if [l, r) ⊆ [u_l, u_r)
        apply(node, l, r, v)
        return
    if [l, r) ∩ [u_l, u_r) = ∅
        return
    m = ⌊(l + r)/2⌋
    push(node, l, r)
    update(left child, l, m, u_l, u_r, v)
    update(right child, m, r, u_l, u_r, v)
    value = left value + right value
```

```
// query sum of range [q_l, q_r)
query(node, l, r, q_l, q_r)
    if [l, r) ⊆ [q_l, q_r)
        return value
    if [l, r) ∩ [q_l, q_r) = ∅
        return 0
    m = ⌊(l + r)/2⌋
    push(node, l, r)
    return query(left child, l, m, q_l, q_r)+
           query(right child, m, r, q_l, q_r)
```

Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?

Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?
- let’s try a bottom up perspective on segment trees

Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?
- let’s try a bottom up perspective on segment trees

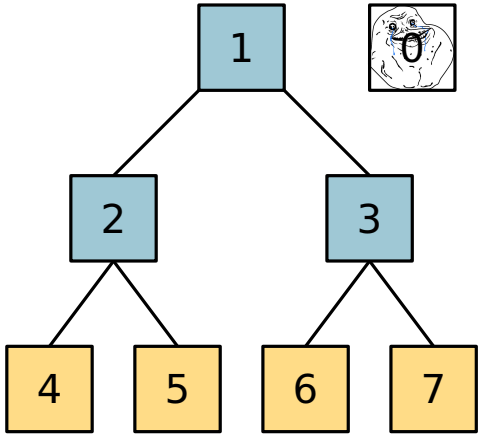
Some Insights into Heap Indexing:

Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?
- let's try a bottom up perspective on segment trees

Some Insights into Heap Indexing:

- use one-based indexing with n rounded up to power of two



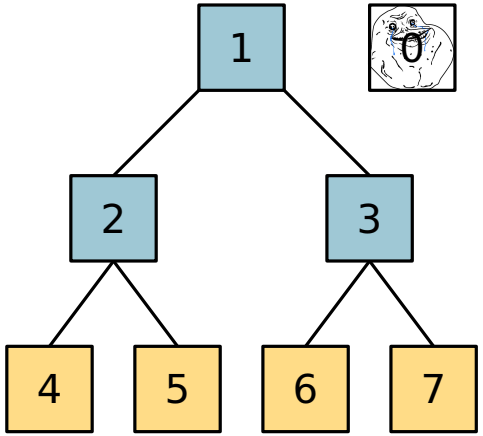
Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?
- let's try a bottom up perspective on segment trees

Some Insights into Heap Indexing:

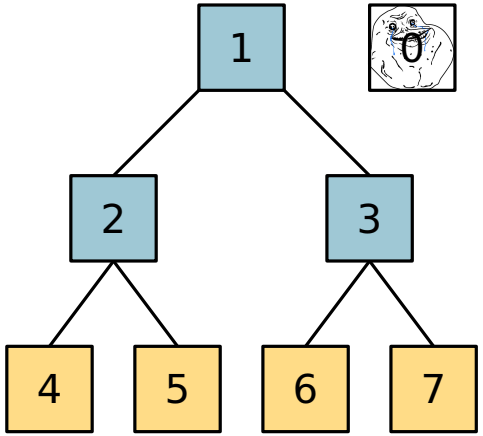
- use one-based indexing with n rounded up to power of two

```
n = 1 << (31 - __builtin_clz(n) + (__builtin_popcount(n)!=1));
```



Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?
- let's try a bottom up perspective on segment trees



Some Insights into Heap Indexing:

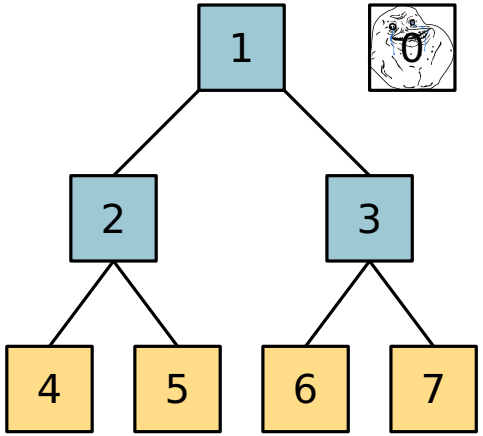
- use one-based indexing with n rounded up to power of two

```
n = 1 << (31 - __builtin_clz(n) + (__builtin_popcount(n)!=1));
```

- root is at index 1; leaves are at positions $[n, 2n)$

Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?
- let's try a bottom up perspective on segment trees



Some Insights into Heap Indexing:

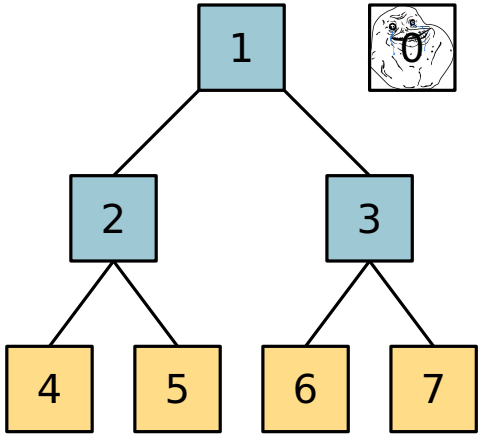
- use one-based indexing with n rounded up to power of two

```
n = 1 << (31 - __builtin_clz(n) + (__builtin_popcount(n)!=1));
```

- root is at index 1; leaves are at positions $[n, 2n)$
- parent of node i is node $i/2$

Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?
- let's try a bottom up perspective on segment trees



Some Insights into Heap Indexing:

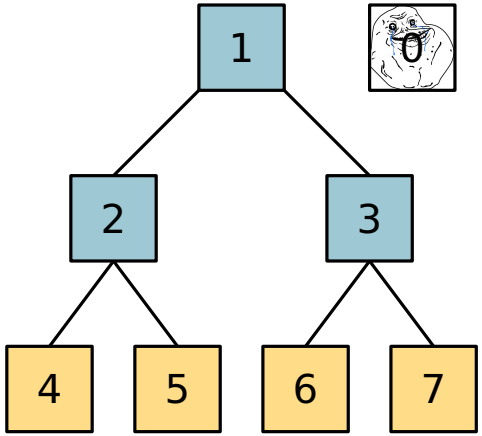
- use one-based indexing with n rounded up to power of two

```
n = 1 << (31 - __builtin_clz(n) + (__builtin_popcount(n)!=1));
```

- root is at index 1; leaves are at positions $[n, 2n)$
- parent of node i is node $i/2$
- even index is a left child; odd is right child

Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?
- let's try a bottom up perspective on segment trees

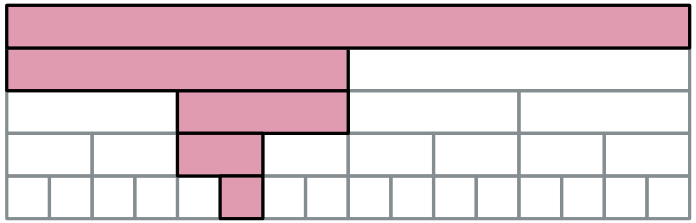


Some Insights into Heap Indexing:

- use one-based indexing with n rounded up to power of two

```
n = 1 << (31 - __builtin_clz(n) + (__builtin_popcount(n)!=1));
```

- root is at index 1; leaves are at positions $[n, 2n)$
- parent of node i is node $i/2$
- even index is a left child; odd is right child

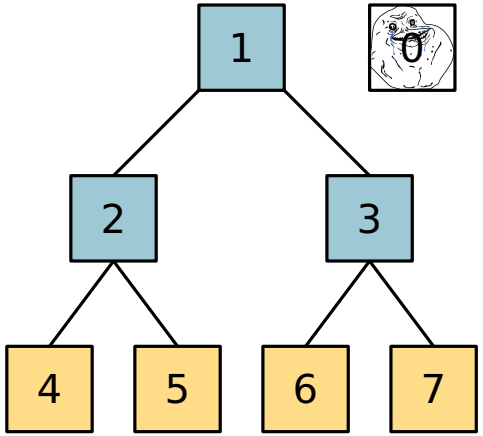


Update

- start at leaf and recompute all parents

Iterative Segment Tree

- are you sick of recursive functions with packs of parameters?
- let's try a bottom up perspective on segment trees

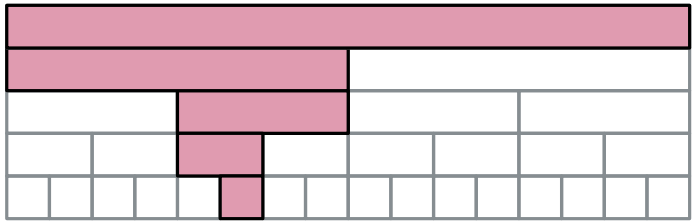


Some Insights into Heap Indexing:

- use one-based indexing with n rounded up to power of two

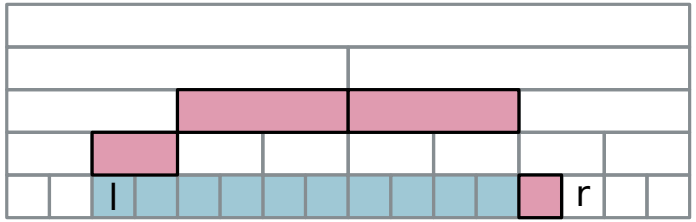
```
n = 1 << (31 - __builtin_clz(n) + (__builtin_popcount(n)!=1));
```

- root is at index 1; leaves are at positions $[n, 2n)$
- parent of node i is node $i/2$
- even index is a left child; odd is right child



Update

- start at leaf and recompute all parents



Query

- break down a consecutive range into $\log n$ nodes

Iterative Segment Tree

Update

- start at leaf and recompute all parents

Iterative Segment Tree

Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Iterative Segment Tree

Update

- start at leaf and recompute all parents

Query

- break down a consecutive range into $\log n$ nodes

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Iterative Segment Tree

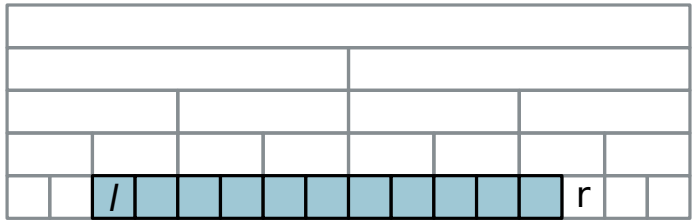
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level



Iterative Segment Tree

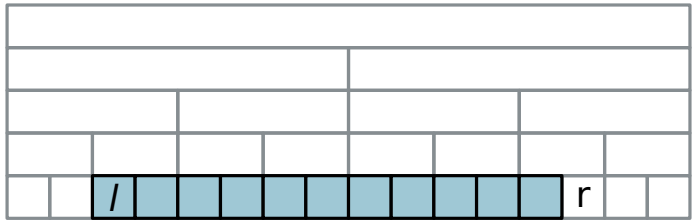
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked



Iterative Segment Tree

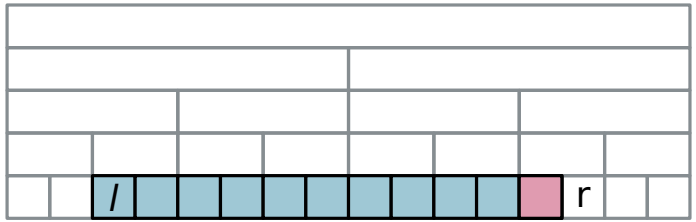
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked



Iterative Segment Tree

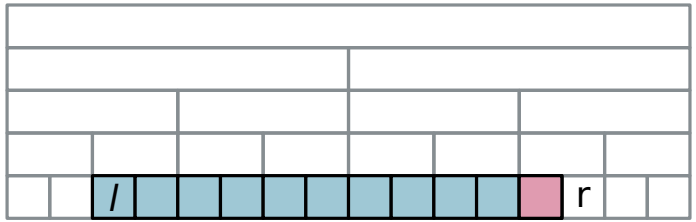
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked
- when r is a right child, then left sibling must be picked and parent will be new exclusive end



Iterative Segment Tree

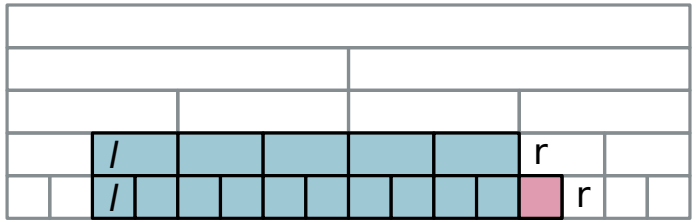
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked
- when r is a right child, then left sibling must be picked and parent will be new exclusive end



Iterative Segment Tree

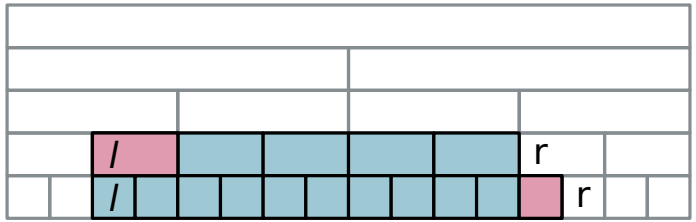
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked
- when r is a right child, then left sibling must be picked and parent will be new exclusive end



Iterative Segment Tree

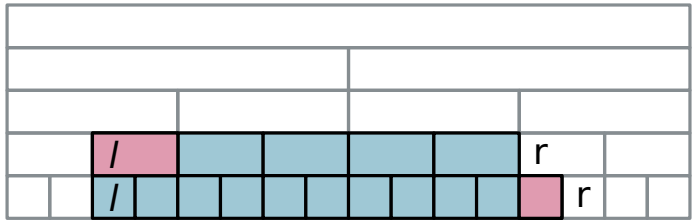
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked
- when r is a right child, then left sibling must be picked and parent will be new exclusive end
- when l is a right child, then parent is not fully included in range; we must pick l and move the start of the range to the right



Iterative Segment Tree

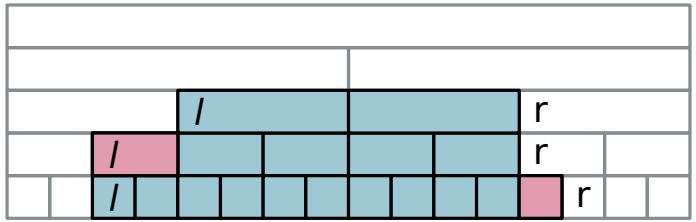
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked
- when r is a right child, then left sibling must be picked and parent will be new exclusive end
- when l is a right child, then parent is not fully included in range; we must pick l and move the start of the range to the right



Iterative Segment Tree

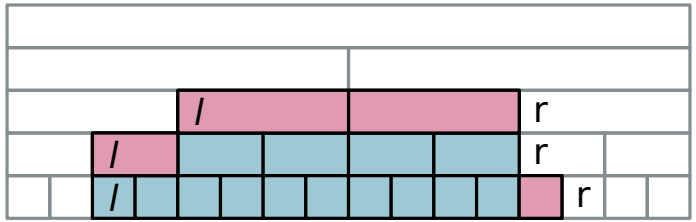
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked
- when r is a right child, then left sibling must be picked and parent will be new exclusive end
- when l is a right child, then parent is not fully included in range; we must pick l and move the start of the range to the right



Iterative Segment Tree

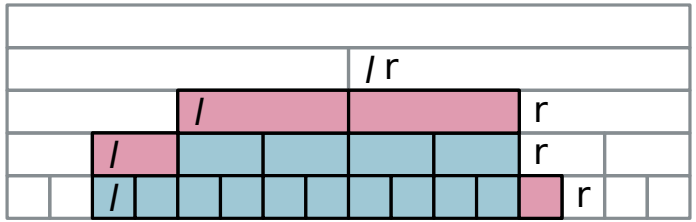
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked
- when r is a right child, then left sibling must be picked and parent will be new exclusive end
- when l is a right child, then parent is not fully included in range; we must pick l and move the start of the range to the right



Iterative Segment Tree

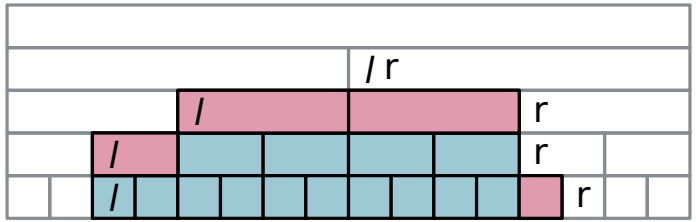
Update

- start at leaf and recompute all parents

```
void update(int i, int value) {
  d[i + n] = value;
  for (int v=n+i; v>1; v/=2)
    d[v/2] = d[v] + d[v^1];
}
```

Query

- break down a consecutive range into $\log n$ nodes
- nodes l, r mark the remaining range in the current level
- we go up level by level and choose a border node (l or $r-1$) if its parent cannot be picked
- when r is a right child, then left sibling must be picked and parent will be new exclusive end
- when l is a right child, then parent is not fully included in range; we must pick l and move the start of the range to the right



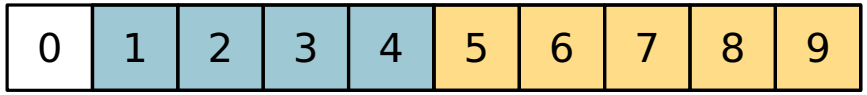

```
int query(int l, int r) {
  int res = 0;
  for (l += n, r += n; l < r;
       l /= 2, r /= 2) {
    if (l&1) res += d[l++];
    if (r&1) res += d[--r];
  }
  return res;
}
```

Iterative Segment Tree

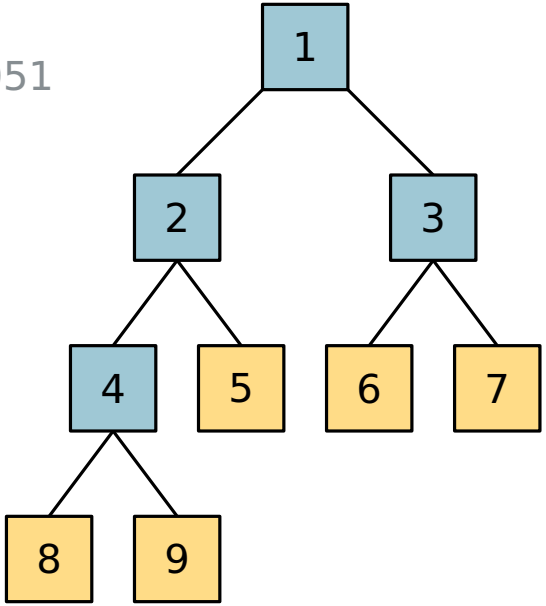
Can we use same indexing but don't round up to next power of 2?

YES

https://codeforces.com/blog/entry/18051



Example $n = 5$



- iterative code works as is
- here we get away with $2n$ memory
- **WARNING**: breaks some recusive top-down code
- usually you want padding to next power to use iterative and recursive code together

Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.

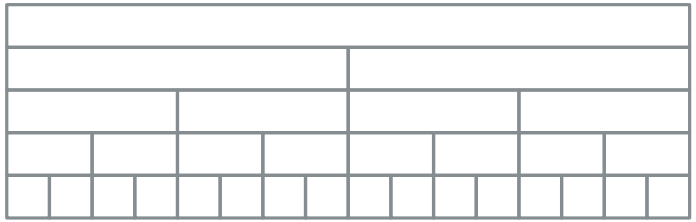
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.



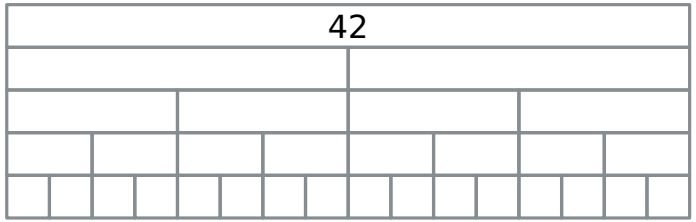
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.



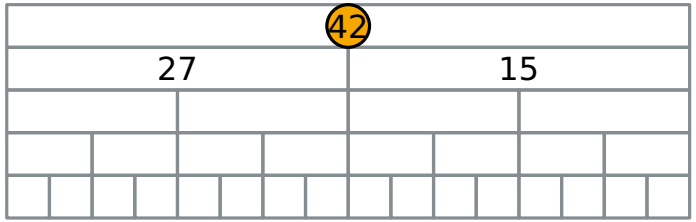
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.



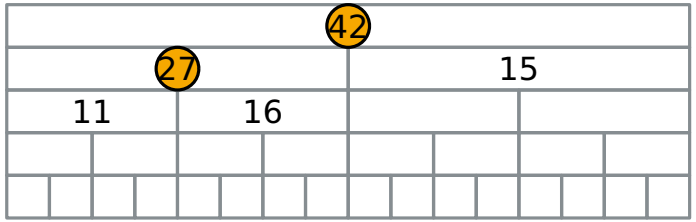
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.



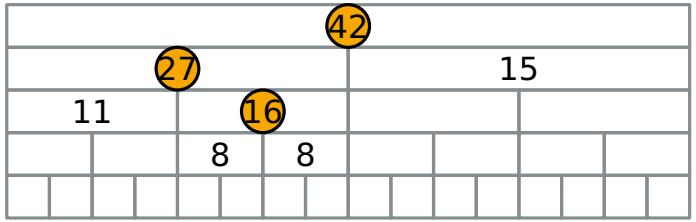
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.



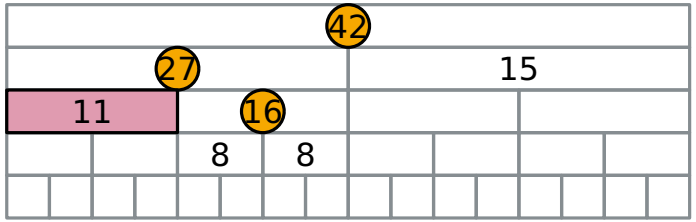
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.



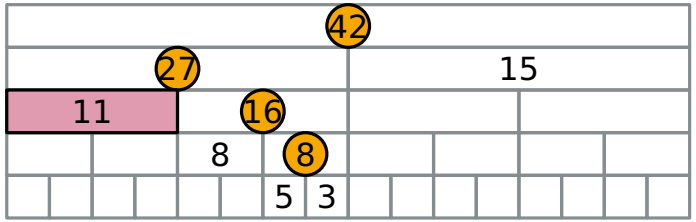
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.



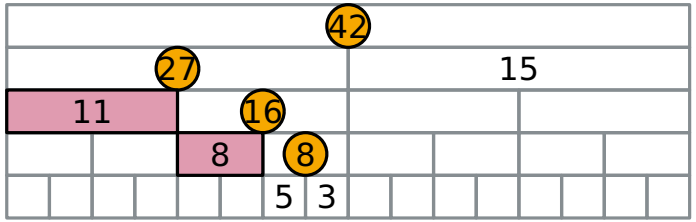
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.



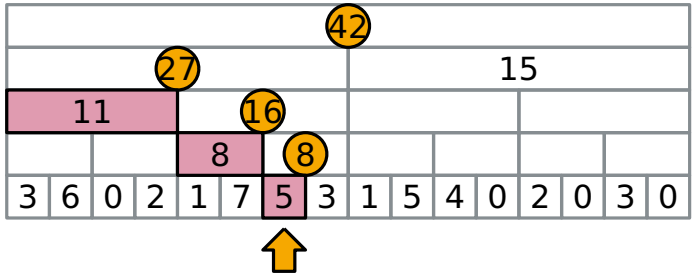
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.



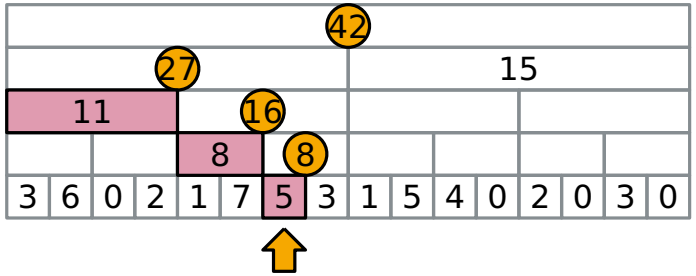
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.




```
int bin_search(int v, int k, int pref_sum) {
    if(v>=n) return v-n; // answer is here
    int with_left = pref_sum + d[2*v];
    if(with_left>k) // answer is left
        return bin_search(2*v, k, pref_sum);
    else  // answer is right
        return bin_search(2*v+1, k, with_left);
}
```

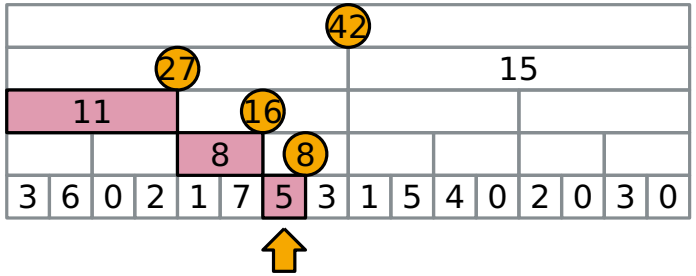
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.




```
int bin_search(int v, int k, int pref_sum) {
    if(v>=n) return v-n; // answer is here
    int with_left = pref_sum + d[2*v];
    if(with_left>k) // answer is left
        return bin_search(2*v, k, pref_sum);
    else  // answer is right
        return bin_search(2*v+1, k, with_left);
}
```

- answer is `bin_search`(1,k,0)

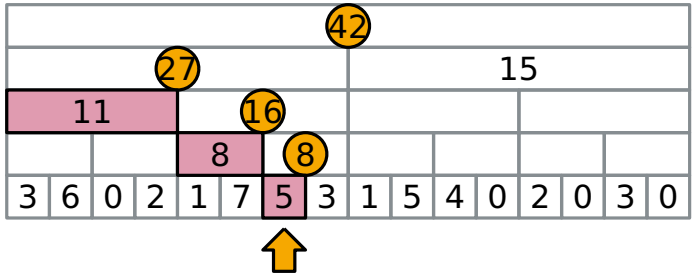
Binary Search over Prefix

Problem: Array of positive integers with point updates and queries of the form: "Given k, what is the shortest prefix, if any, with $sum(a_0, \ldots, a_i) > k$?".

- binary search the answer with a segment tree that answers range-sum queries

  $\rightarrow O(\log^2 n)$ per query

- can we do better? Example $k = 20$.




```
int bin_search(int v, int k, int pref_sum) {
    if(v>=n) return v-n; // answer is here
    int with_left = pref_sum + d[2*v];
    if(with_left>k) // answer is left
        return bin_search(2*v, k, pref_sum);
    else  // answer is right
        return bin_search(2*v+1, k, with_left);
}
```

- answer is `bin_search`(1,k,0)

  $\rightarrow O(\log n)$ per query

Binary Search over Prefix

Problem: Array of integers with point updates and queries of the form: "What is the shortest prefix of $A[l, r)$, if any, with $sum > k$?"

Binary Search over Prefix

Problem: Array of integers with point updates and queries of the form: "What is the shortest prefix of $A[l, r)$, if any, with $sum > k$?"

- same problem, but the query now has a range instead of being on prefix

Binary Search over Prefix

Problem: Array of integers with point updates and queries of the form: "What is the shortest prefix of $A[l, r)$, if any, with $sum > k$?"

- same problem, but the query now has a range instead of being on prefix

Stategy:

1. break range down into $\log n$ segtree nodes
2. find segtree node that contains answer in its subarray
3. do prefix bin-search inside this node

Binary Search over Prefix

Problem: Array of integers with point updates and queries of the form: "What is the shortest prefix of $A[l, r)$, if any, with $sum > k$?"

- same problem, but the query now has a range instead of being on prefix

Stategy:

1. break range down into $\log n$ segtree nodes
2. find segtree node that contains answer in its subarray
3. do prefix bin-search inside this node

- can be done all at once: https://codeforces.com/blog/entry/83883?#comment-712628

Binary Search over Prefix

Problem: Array of integers with point updates and queries of the form: "What is the shortest prefix of $A[l, r)$, if any, with $sum > k$?"

- same problem, but the query now has a range instead of being on prefix

Stategy:

1. break range down into $\log n$ segtree nodes
2. find segtree node that contains answer in its subarray
3. do prefix bin-search inside this node

- can be done all at once: https://codeforces.com/blog/entry/83883?#comment-712628
- or by explicitly creating a vector of the segtree nodes that represent the range

Binary Search over Prefix

- why do I call it a *binary search*?
- the iterative code looks like this

```
int bin_search_gcd(int v, int g) {
    while(v<n) {
        if(gcd(d[2*v],g) == 1)
            v = 2*v;
        else
            g = gcd(d[2*v],g), v = 2*v+1;
    }
    return v-n;
}
```

- now lets track the l, r range of the current node (although we don't need it)

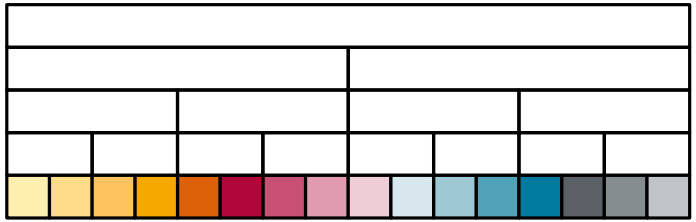
```
int bin_search_gcd(int v, int l, int r, int g) {
    while(r-l>1) {
        int mid = (r+l)/2;
        if(gcd(d[2*v],g) == 1)
            r = mid, v = 2*v;
        else
            l = mid, g = gcd(d[2*v],g), v = 2*v+1;
    }
    return l;
}
```

- this looks more like a bin-search!

Nested Data Structures

”Saving the entire subarrays in each vertex”

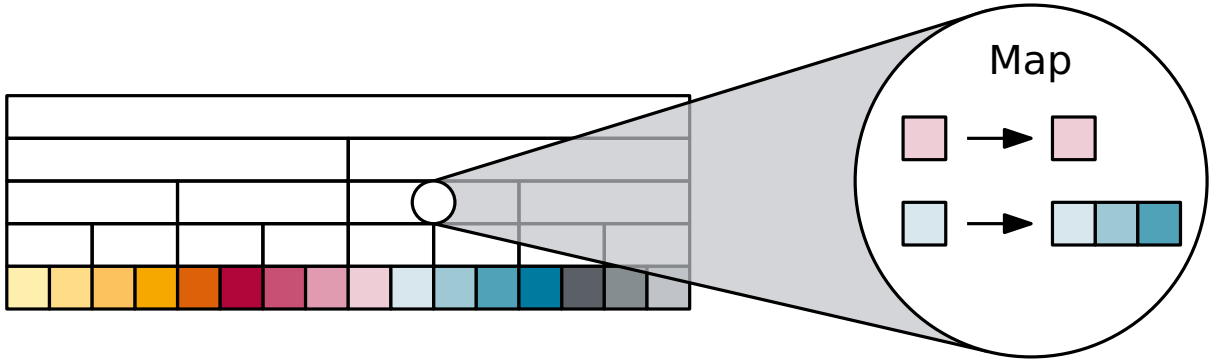
— cp-algorithms



Nested Data Structures

”Saving the entire subarrays in each vertex”

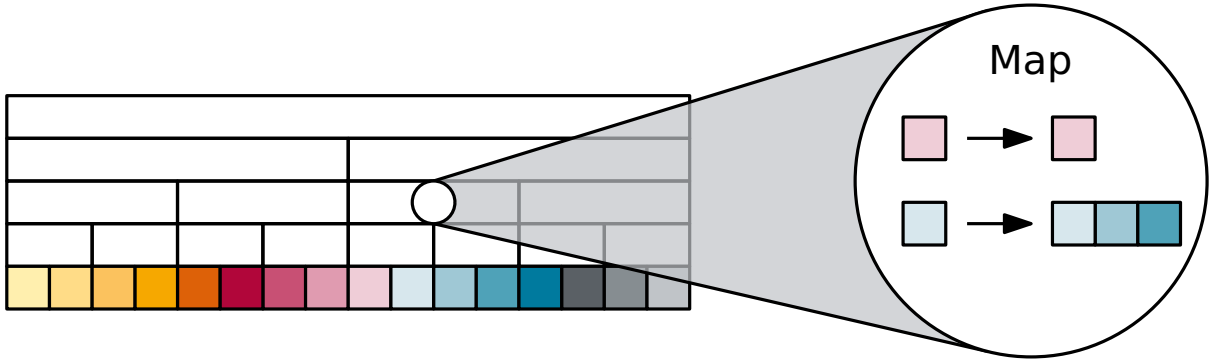
— cp-algorithms



Nested Data Structures

”Saving the entire subarrays in each vertex”

— cp-algorithms

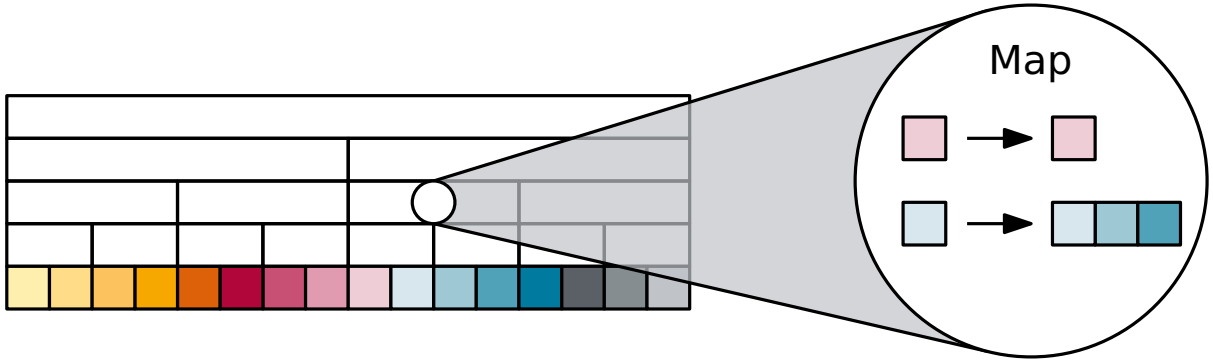


How much memory used per level?

Nested Data Structures

”Saving the entire subarrays in each vertex”

— cp-algorithms

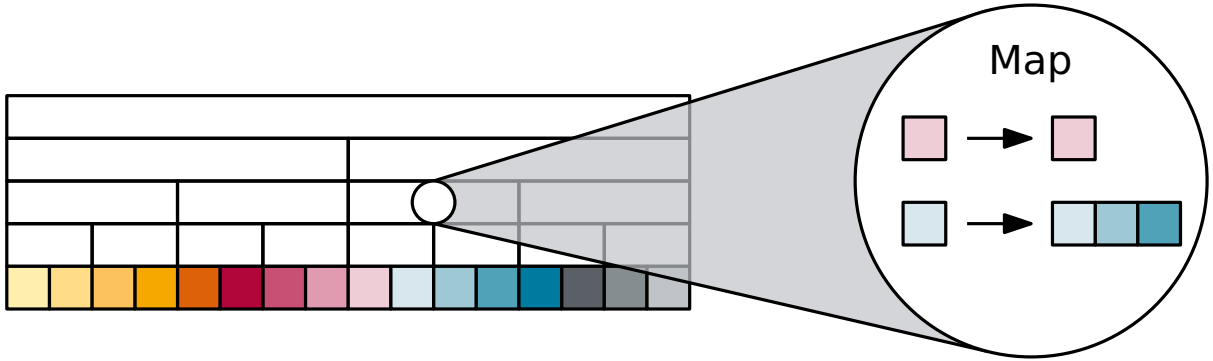


How much memory used per level? $\rightarrow n$

Nested Data Structures

”Saving the entire subarrays in each vertex”

— cp-algorithms



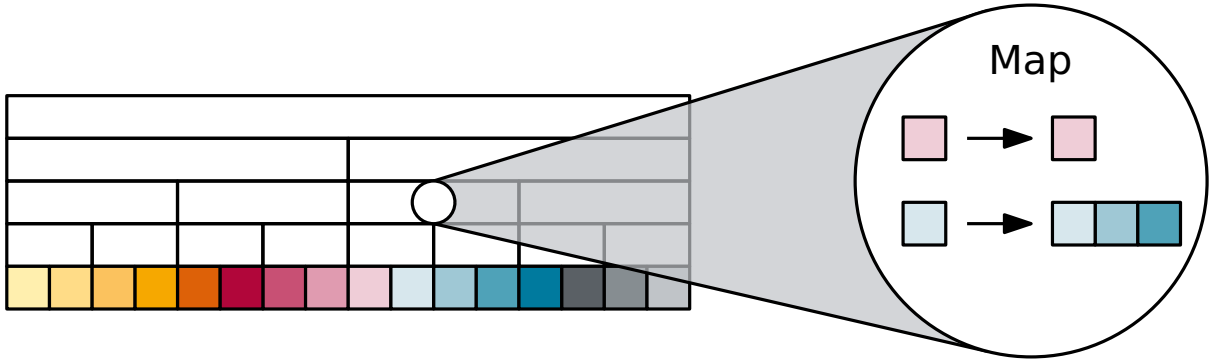
How much memory used per level? $\rightarrow n$

How much memory used?

Nested Data Structures

”Saving the entire subarrays in each vertex”

— cp-algorithms



How much memory used per level? $\rightarrow n$

How much memory used? $\rightarrow n \log n$

Nested Data Structures - Examples

Problem: We want to answer queries of the following form: for three given numbers (l, r, x) we have to find the minimal number in the segment $a[l \dots r]$ which is greater than or equal to x.

Nested Data Structures - Examples

Problem: We want to answer queries of the following form: for three given numbers (l, r, x) we have to find the minimal number in the segment $a[l \dots r]$ which is greater than or equal to x.

sorted subarray in each node

Nested Data Structures - Examples

Problem: We want to answer queries of the following form: for three given numbers (l, r, x) we have to find the minimal number in the segment $a[l \dots r]$ which is greater than or equal to x.

sorted subarray in each node

binary search solution in each node (`std::lower_bound`)

Nested Data Structures - Examples

Problem: We want to answer queries of the following form: for three given numbers (l, r, x) we have to find the minimal number in the segment $a[l \dots r]$ which is greater than or equal to x.

sorted subarray in each node

binary search solution in each node (`std::lower_bound`)

$O(\log^2 n)$ per query

Nested Data Structures - Examples

Problem: We want to answer queries of the following form: for three given numbers (l, r, x) we have to find the minimal number in the segment $a[l \dots r]$ which is greater than or equal to x.

sorted subarray in each node

binary search solution in each node (`std::lower_bound`)

$O(\log^2 n)$ per query

Problem: Same but with point updates.

Nested Data Structures - Examples

Problem: We want to answer queries of the following form: for three given numbers (l, r, x) we have to find the minimal number in the segment $a[l \dots r]$ which is greater than or equal to x.

sorted subarray in each node

binary search solution in each node (`std::lower_bound`)

$O(\log^2 n)$ per query

Problem: Same but with point updates.

use `multiset` instead of sorted `vector`

Nested Data Structures - Examples

Problem: We want to answer queries of the following form: for three given numbers (l, r, x) we have to find the minimal number in the segment $a[l \dots r]$ which is greater than or equal to x.

sorted subarray in each node

binary search solution in each node (`std::lower_bound`)

$O(\log^2 n)$ per query

Problem: Same but with point updates.

use `multiset` instead of sorted `vector`

binary search solution in each node (`std::multiset::lower_bound`)

Nested Data Structures - Examples

Problem: We want to answer queries of the following form: for three given numbers (l, r, x) we have to find the minimal number in the segment $a[l \dots r]$ which is greater than or equal to x.

sorted subarray in each node

binary search solution in each node (`std::lower_bound`)

$O(\log^2 n)$ per query

Problem: Same but with point updates.

use `multiset` instead of sorted `vector`

binary search solution in each node (`std::multiset::lower_bound`)

still $O(\log^2 n)$ per query or update

Nested Data Structures - Advanced Example

Problem: I have a very large (10^5) list of all first names of my students sorted by Matrikelnummer. Now Philipp asks me questions like: ”How many 3rd Semester (Mat.Nr. from 795000 to 799000) have names starting with ’Ch’?”. Also, people have a name change from time to time.

this should be done with a segment tree that has a trie in each node

needs trie deletion https://www.geeksforgeeks.org/trie-delete/

alternative w/o updates: persistent trie where I insert names in asc. matNR

persistency allows me to answer for prefix of matNR

answer queries as right border - left border

can be done offline to avoid persistency

Implicit Segment Tree

- what if $O(n)$ memory is too much?

Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty

Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero

Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand

Implicit Segment Tree

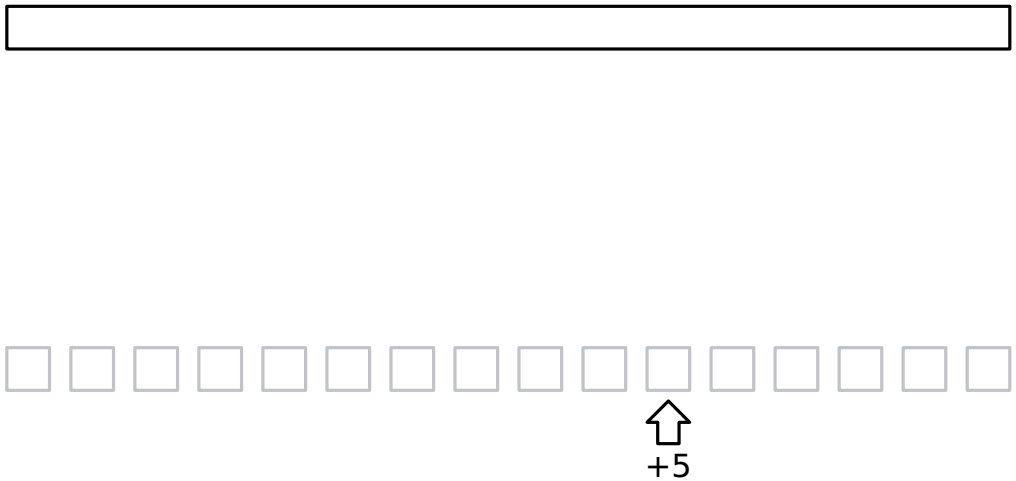
- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total

Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total

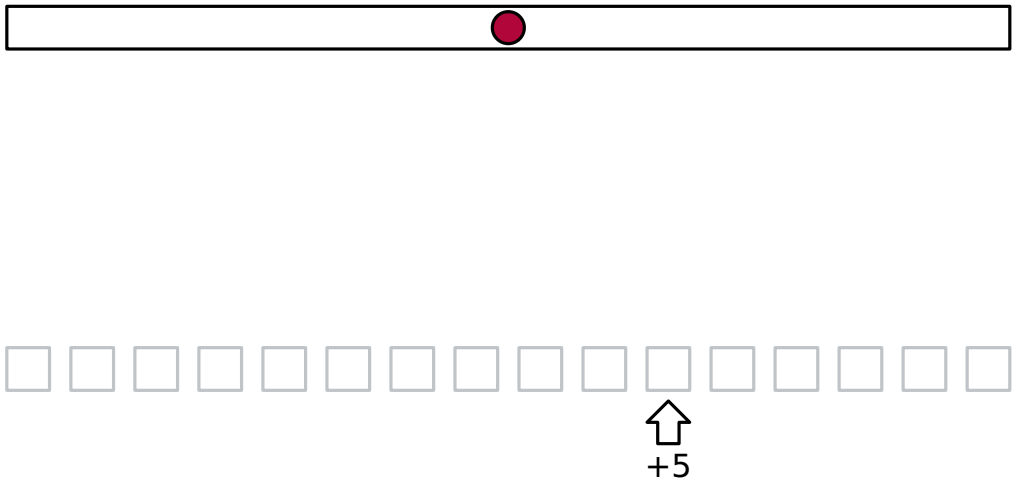
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



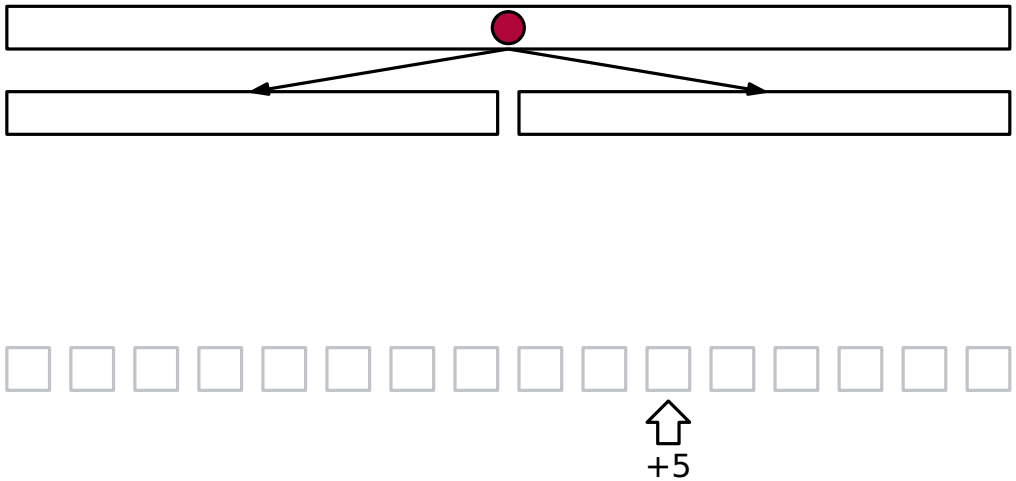
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



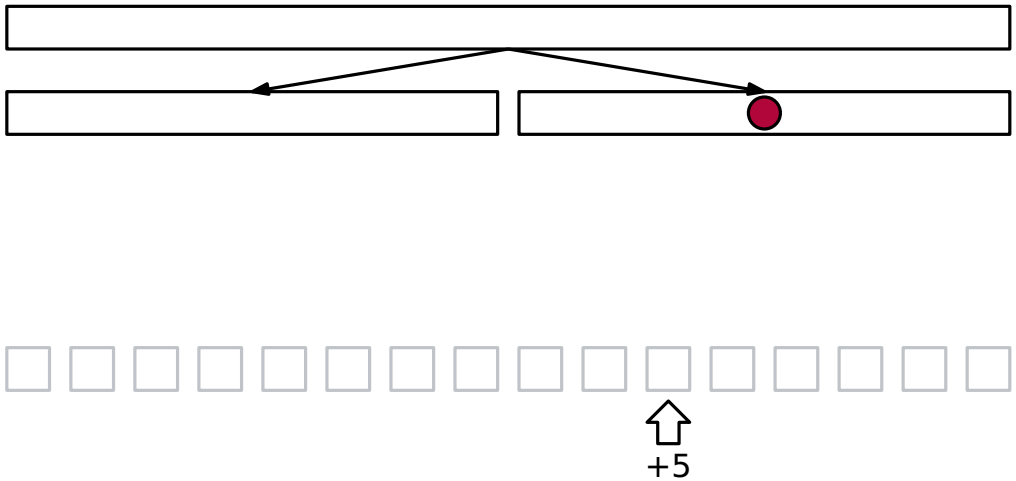
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



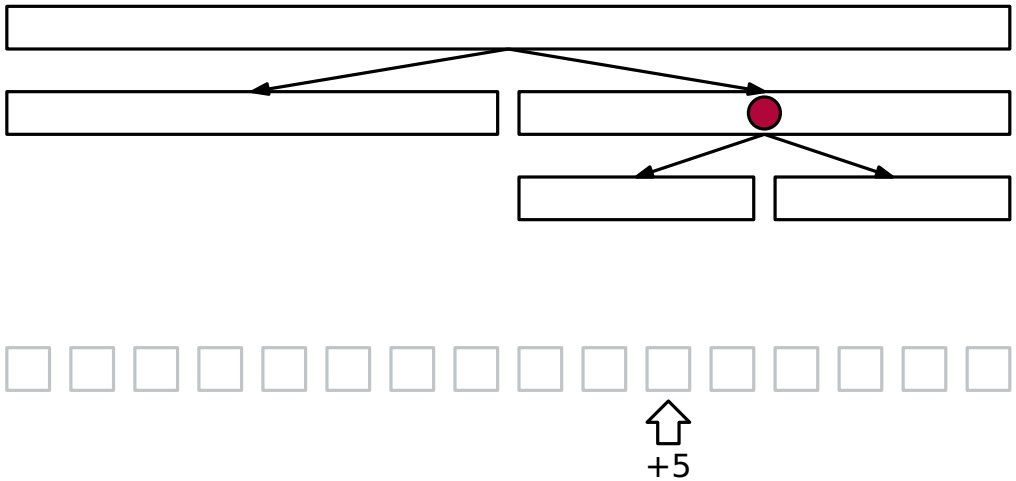
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



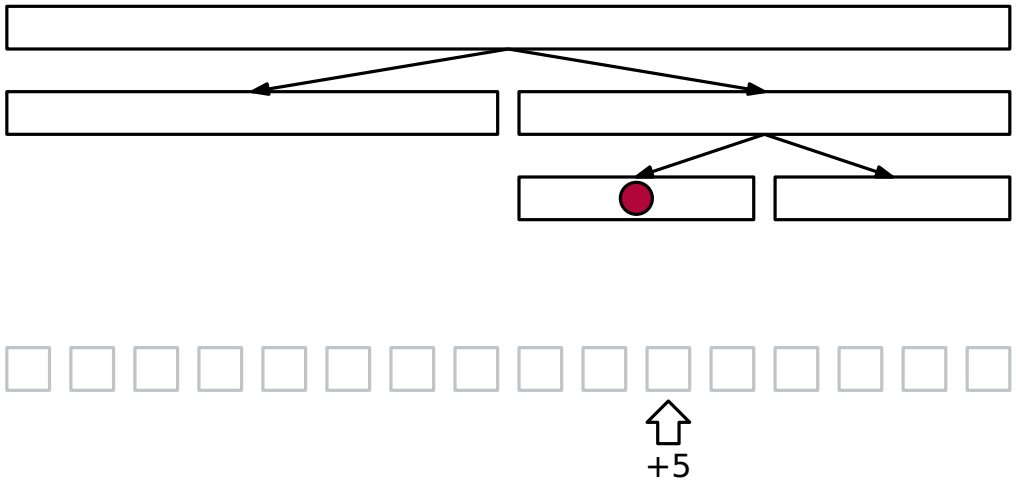
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



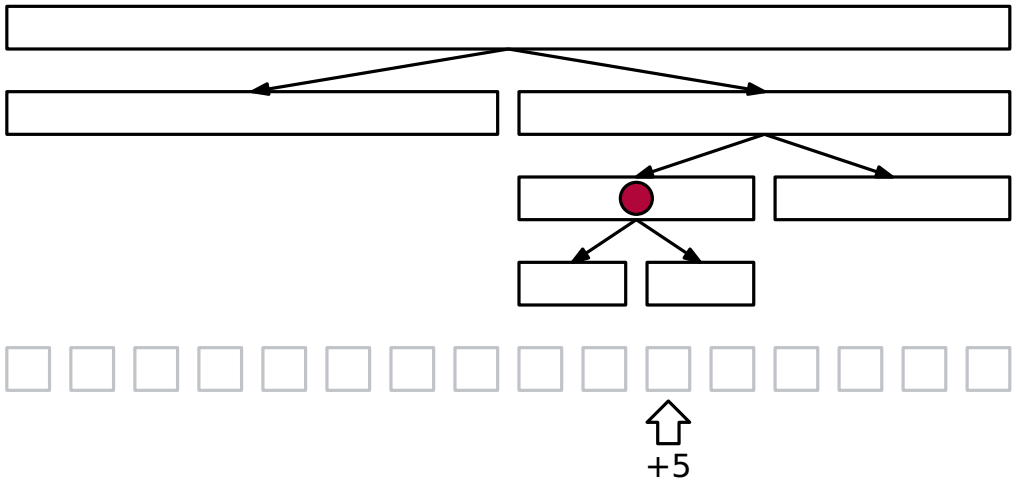
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



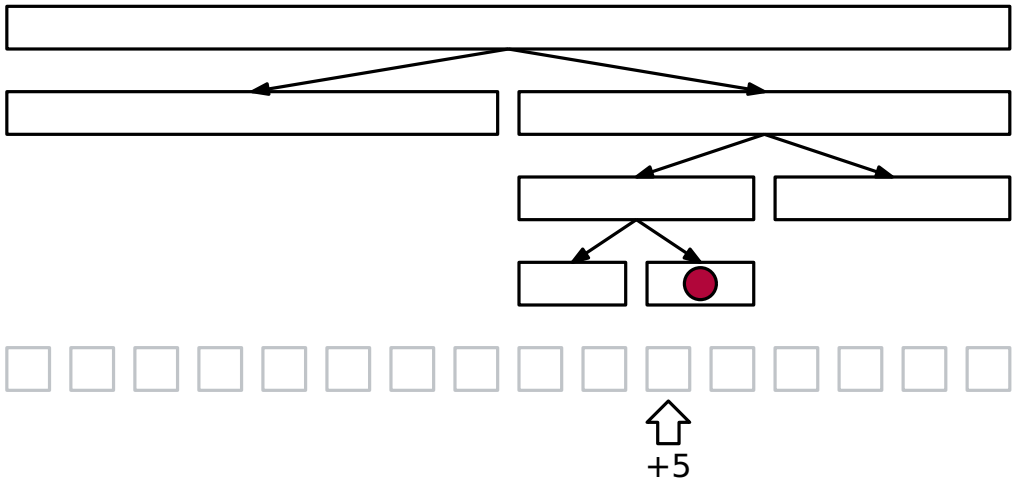
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



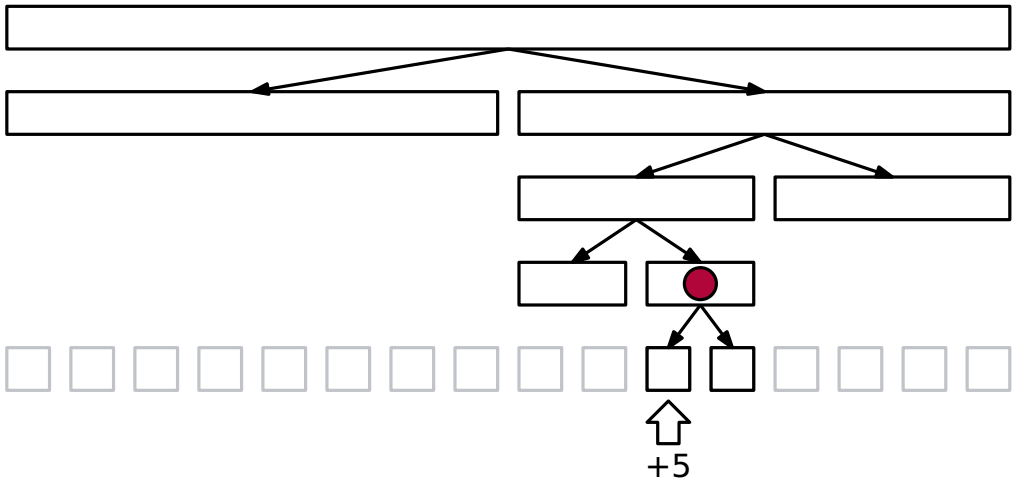
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



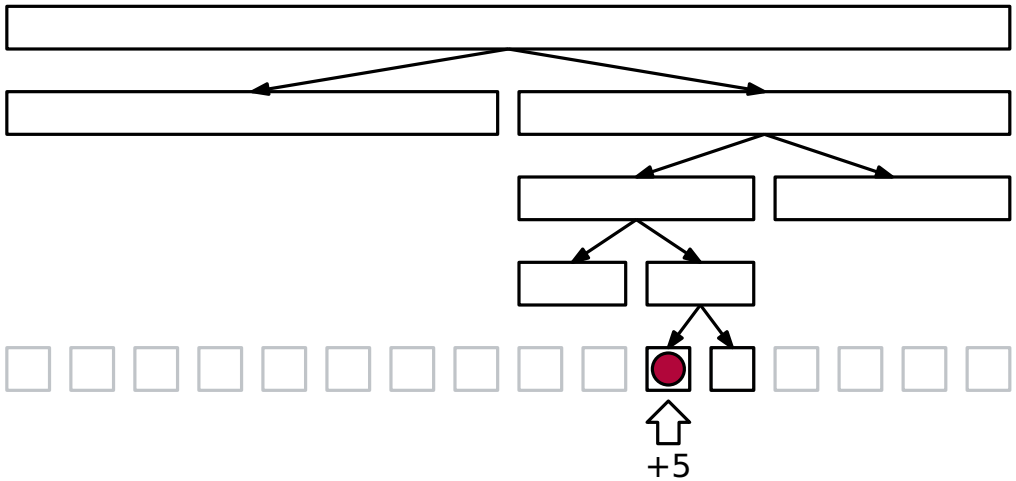
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



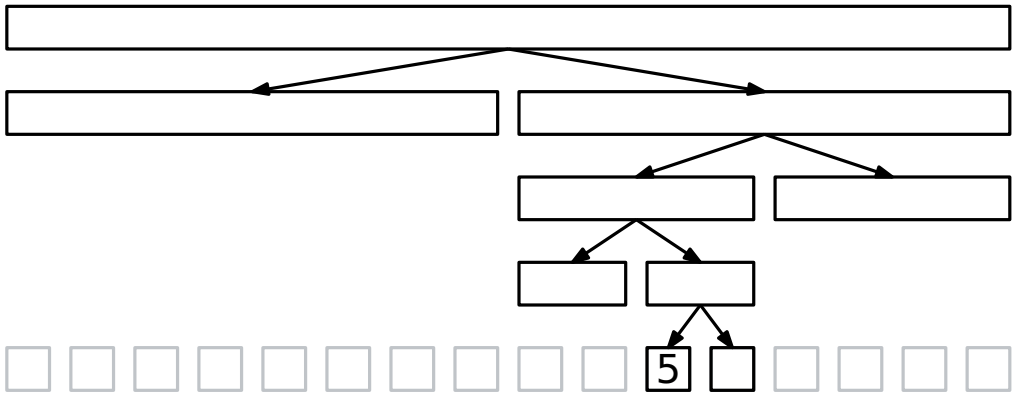
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



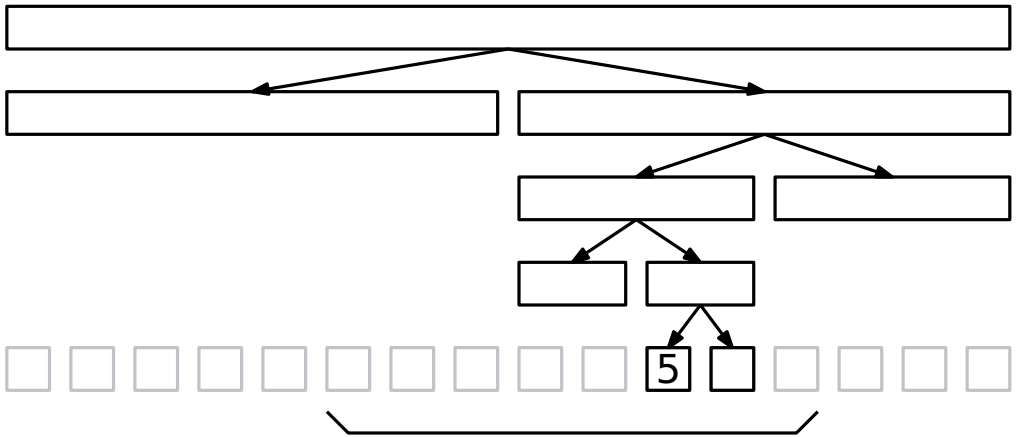
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



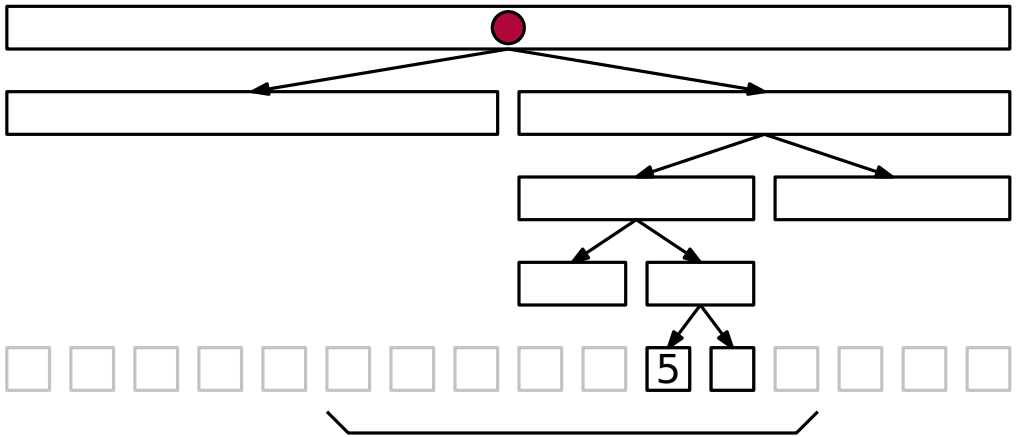
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



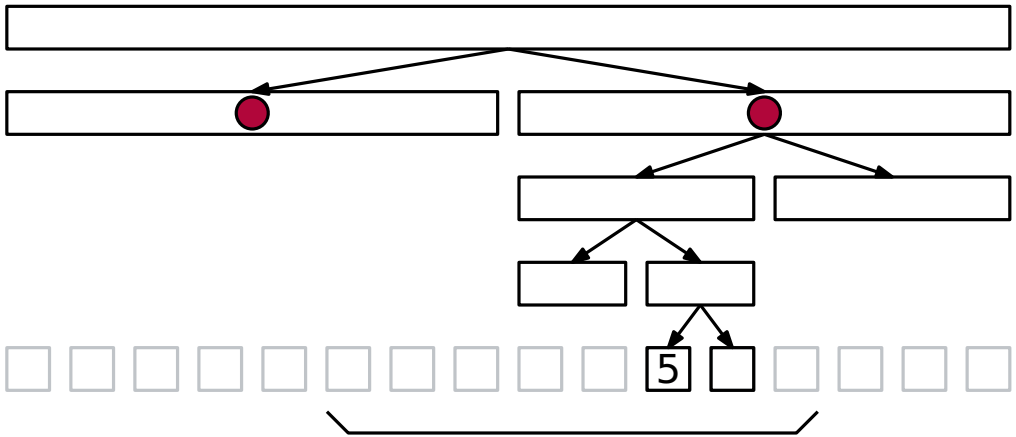
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



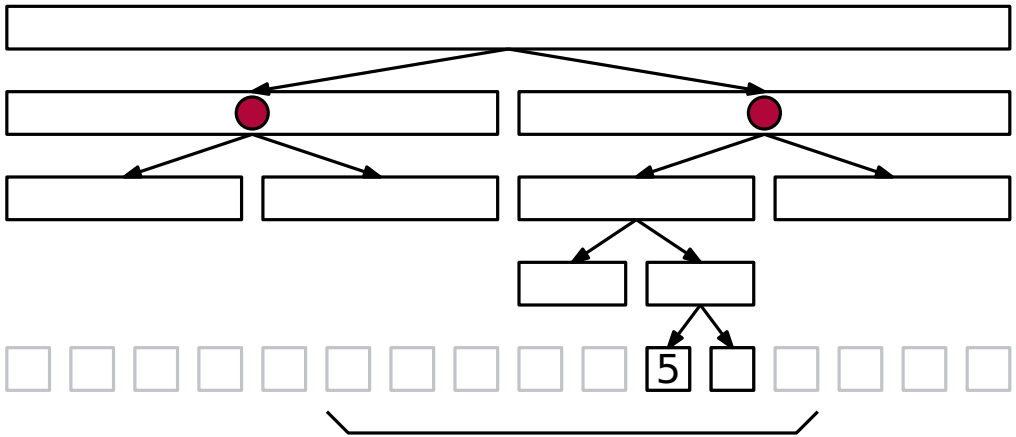
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



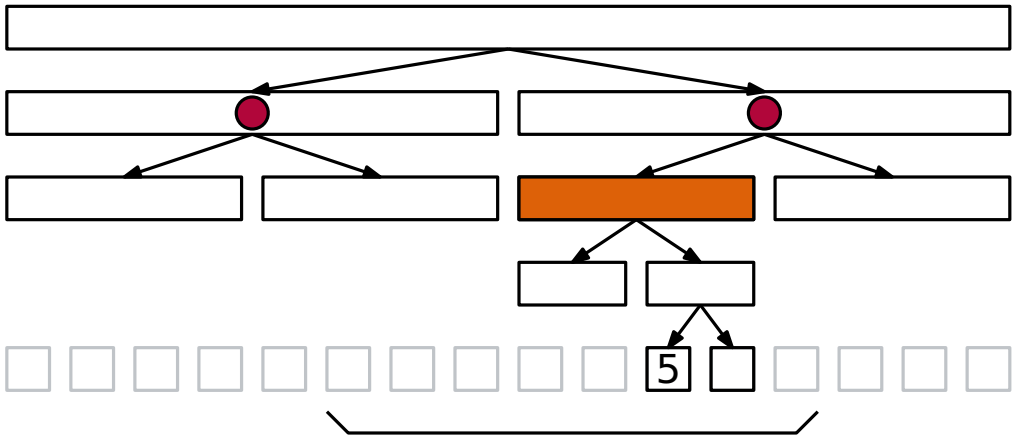
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



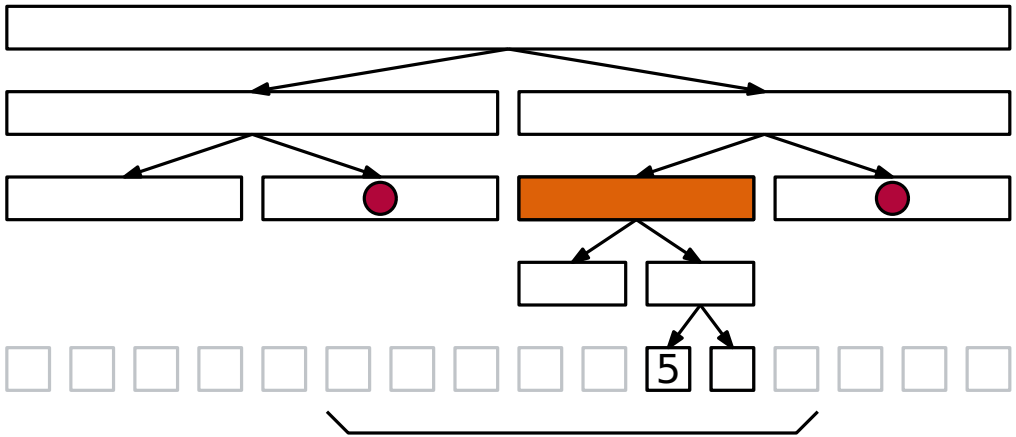
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



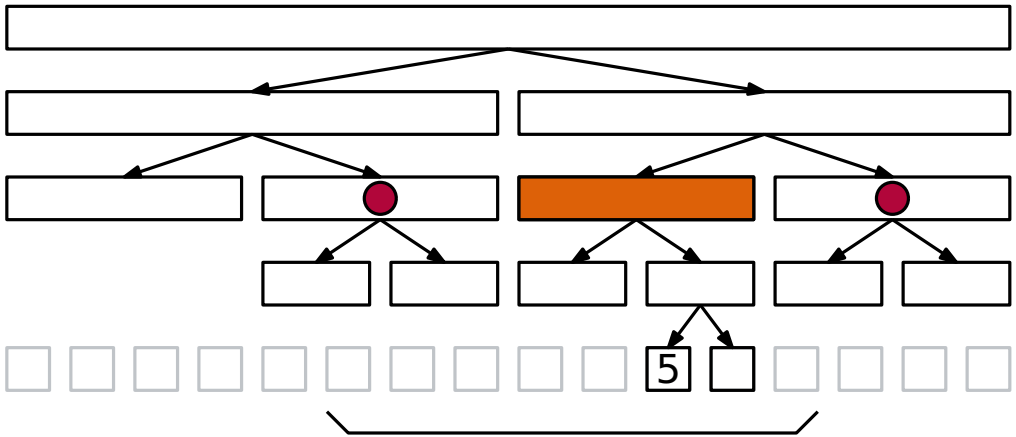
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



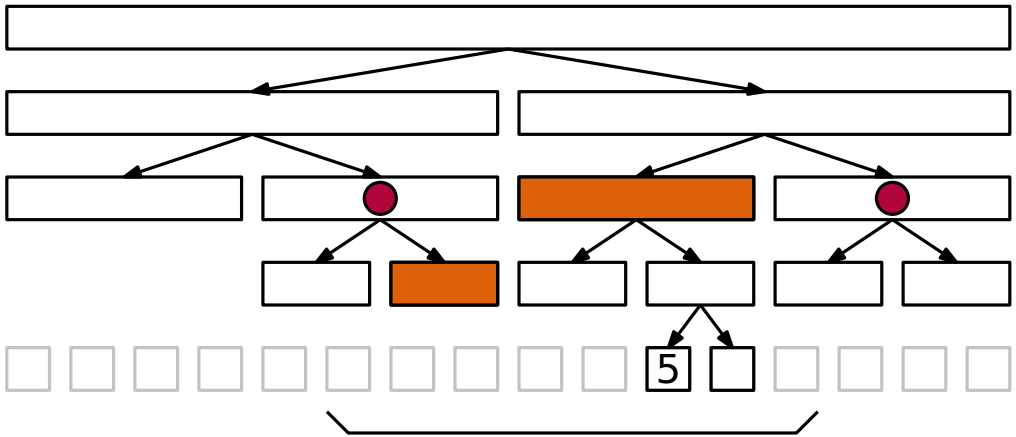
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



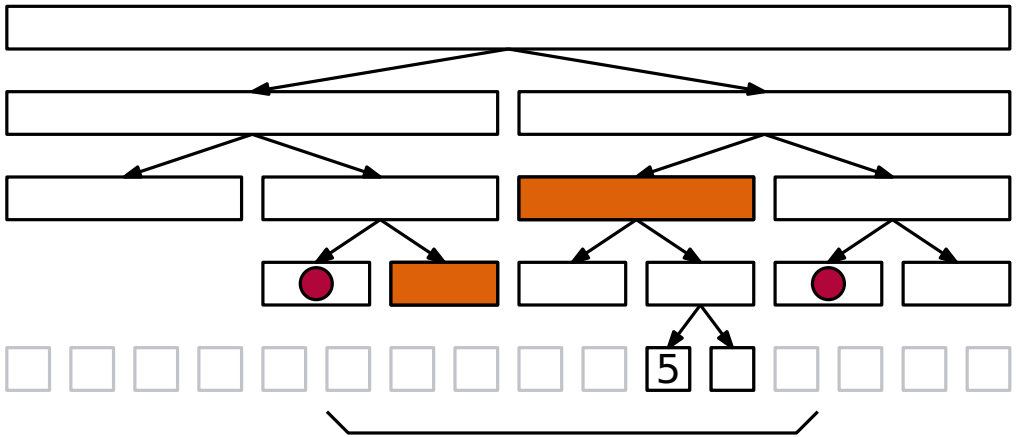
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



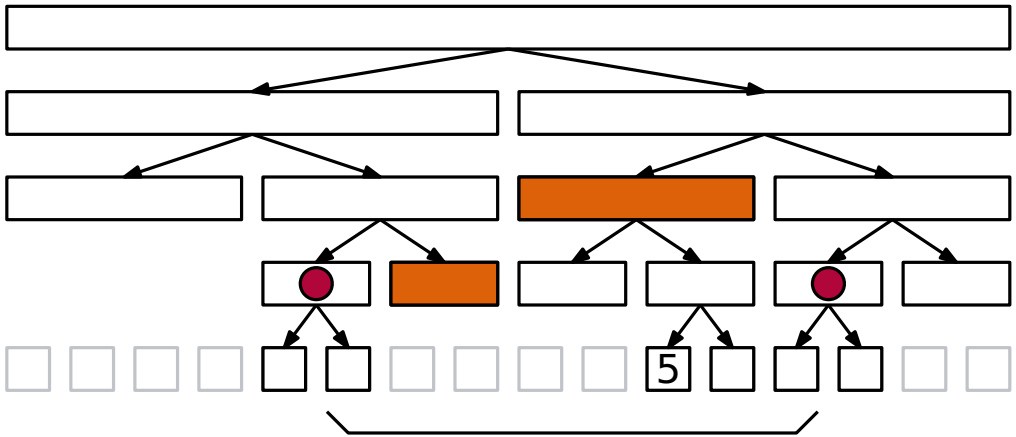
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



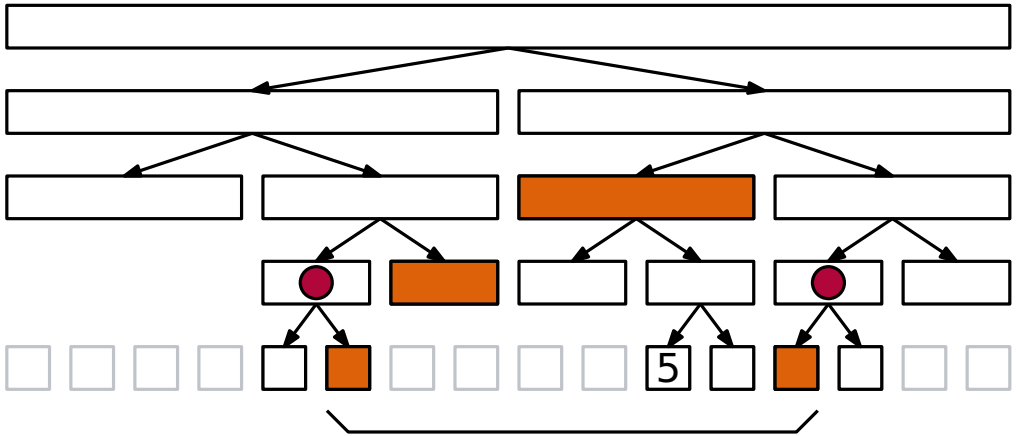
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



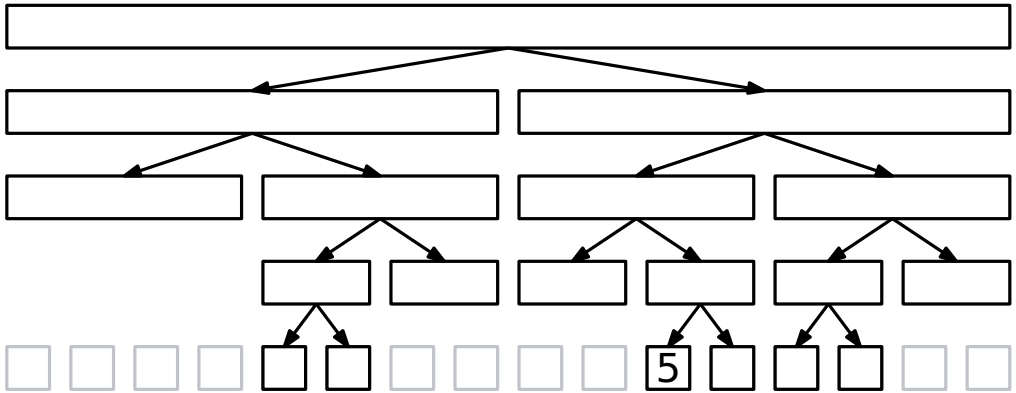
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



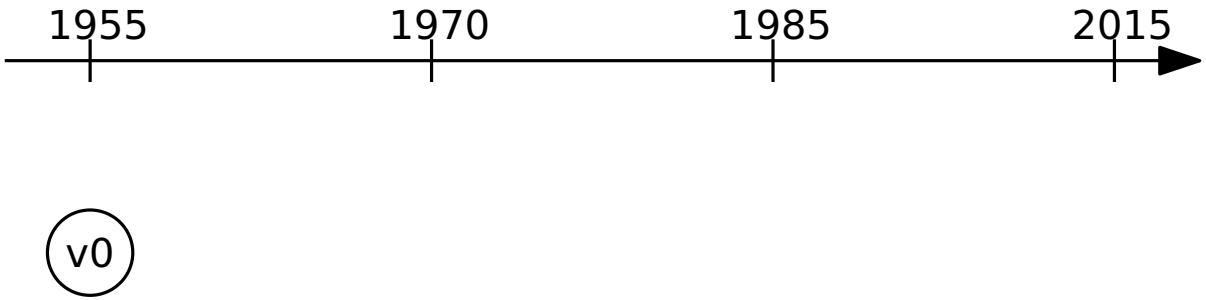
Implicit Segment Tree

- what if $O(n)$ memory is too much?
- e.g. range queries and point updates on an array of length 10^{12} that is initially empty
- **observation**: input is limited, so at most q positions in this array can be non-zero
- **idea**: implement segtree with pointers and create new nodes on demand
- each query needs $O(\log n)$ nodes $\rightarrow O(q \log n)$ in total



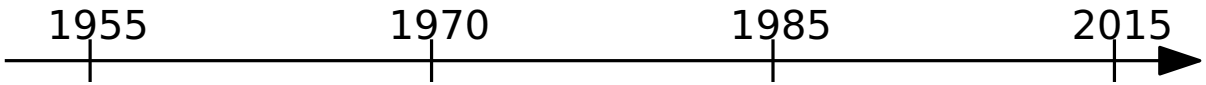
Persistent Segment Tree

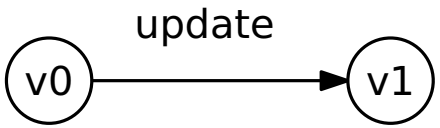
Persistent Segment Tree



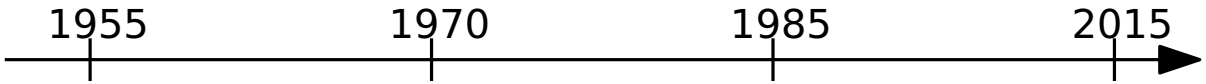
Persistent Segment Tree

update $v0 \rightarrow v1$



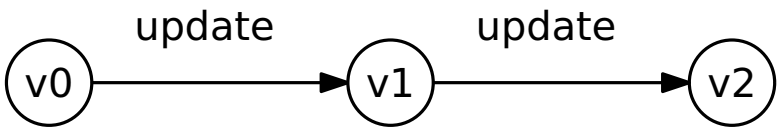


Persistent Segment Tree

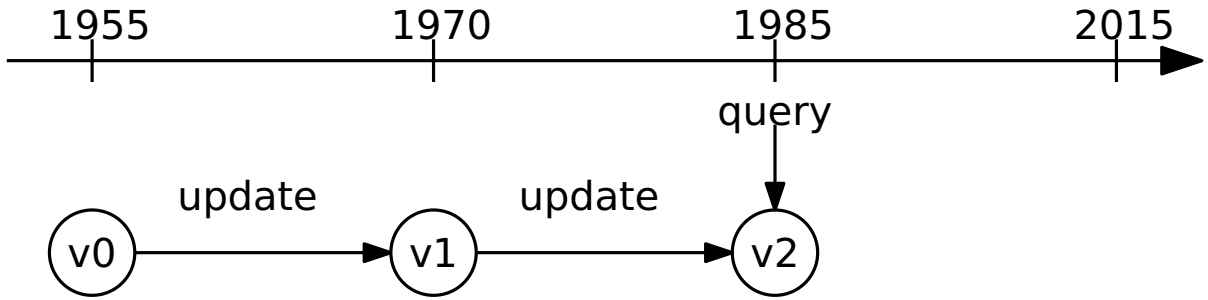


update $v0 \rightarrow v1$

update $v1 \rightarrow v2$



Persistent Segment Tree

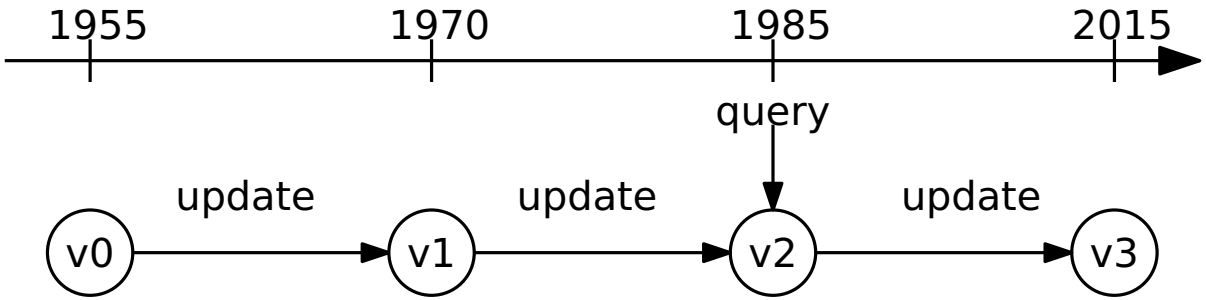


update $v0 \rightarrow v1$

update $v1 \rightarrow v2$

query $v2$

Persistent Segment Tree



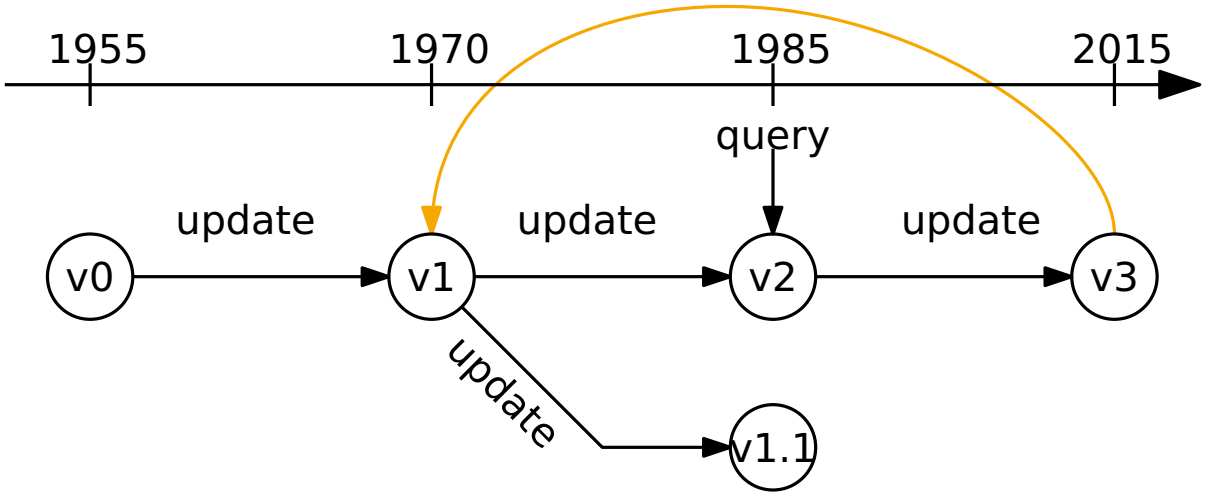
update $v0 \to v1$

update $v1 \to v2$

query $v2$

update $v2 \to v3$

Persistent Segment Tree



update $v0 \to v1$

update $v1 \to v2$

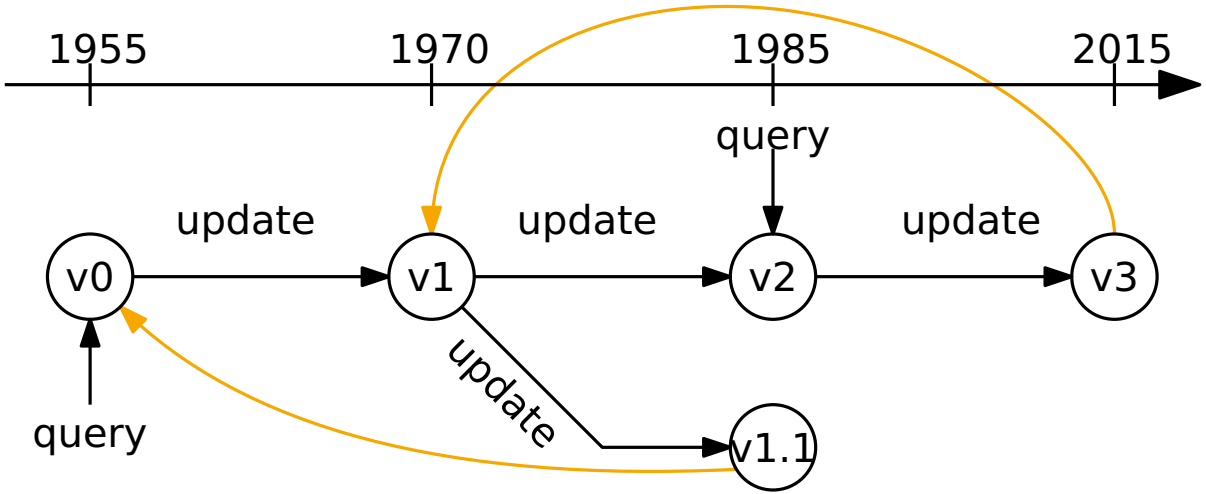
query $v2$

update $v2 \to v3$

time travel

update $v1 \to v1.1$

Persistent Segment Tree



update $v0 \to v1$

update $v1 \to v2$

query $v2$

update $v2 \to v3$

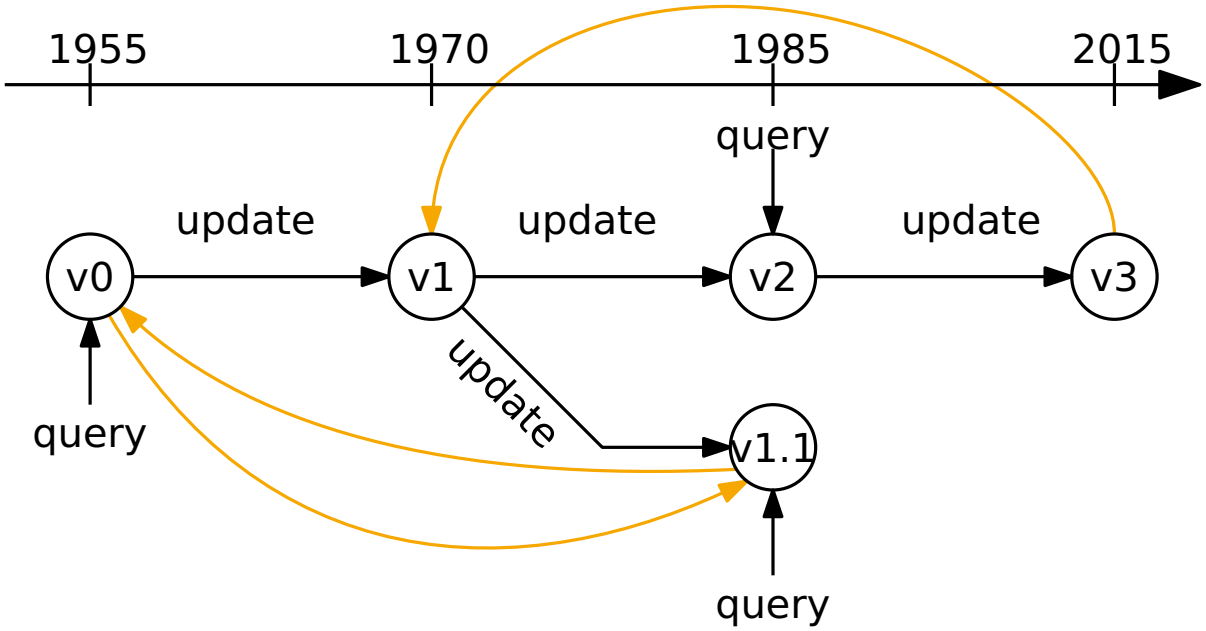
time travel

update $v1 \to v1.1$

time travel

query $v0$

Persistent Segment Tree



update $v0 \to v1$

update $v1 \to v2$

query $v2$

update $v2 \to v3$

time travel

update $v1 \to v1.1$

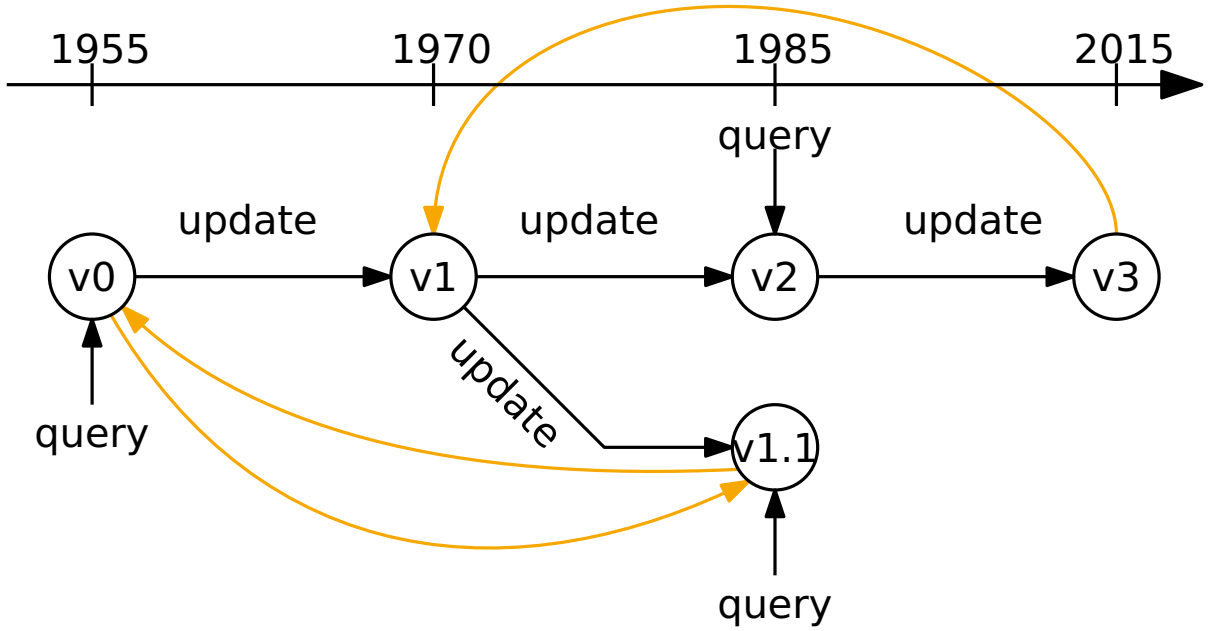
time travel

query $v0$

time travel

query $v1.1$

Persistent Segment Tree



update $v0 \to v1$

update $v1 \to v2$

query $v2$

update $v2 \to v3$

time travel

update $v1 \to v1.1$

time travel

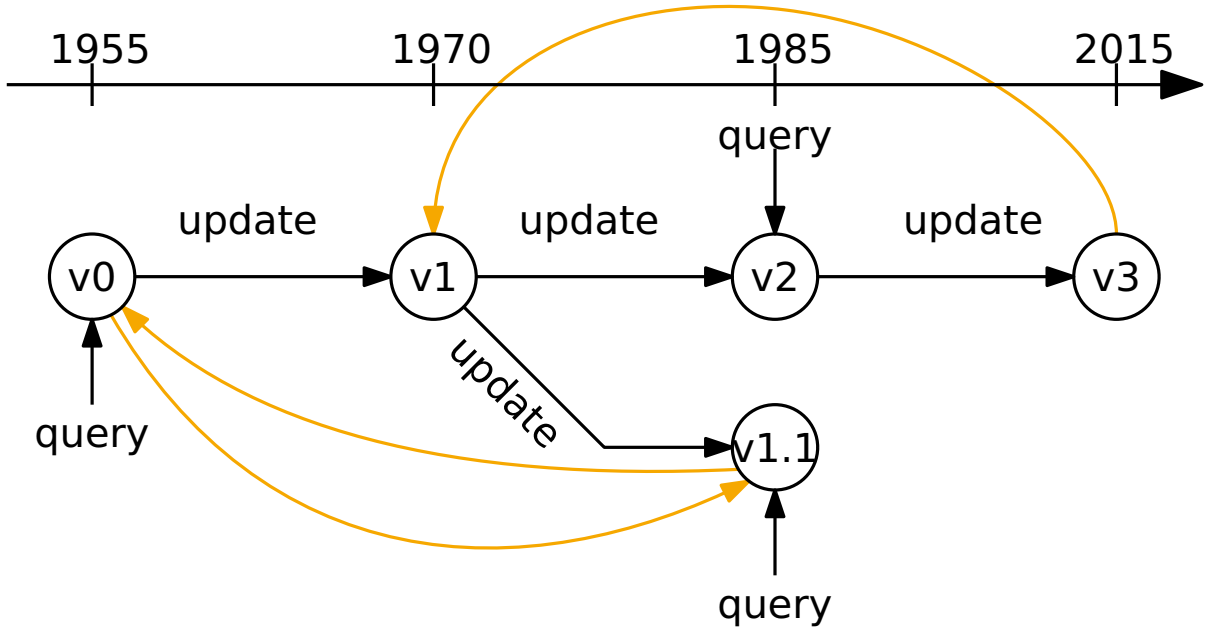
query $v0$

time travel

query $v1.1$

- a persistent data structure remembers its previous state for each modification

Persistent Segment Tree



update $v0 \to v1$

update $v1 \to v2$

query $v2$

update $v2 \to v3$

time travel

update $v1 \to v1.1$

time travel

query $v0$

time travel

query $v1.1$

- a persistent data structure remembers its previous state for each modification
- allows access to any version of this data structure and execute a query on it or create a new revision from there

Persistent Segment Tree

- we use pointer like implicit segtree

Persistent Segment Tree

- we use pointer like implicit segtree
- every time we modify a vertex, we make a copy

Persistent Segment Tree

- we use pointer like implicit segtree
- every time we modify a vertex, we make a copy
- the old revision still points to the original

Persistent Segment Tree

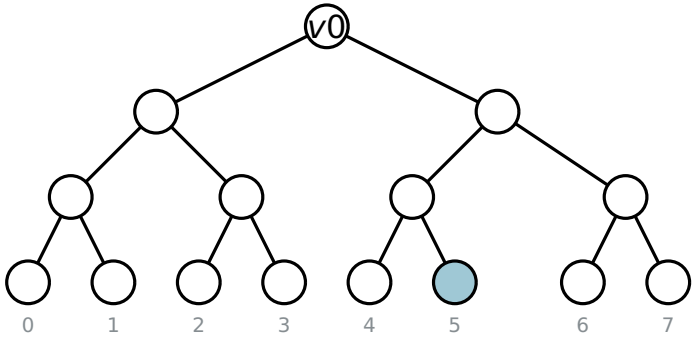
- we use pointer like implicit segtree
- every time we modify a vertex, we make a copy
- the old revision still points to the original
- the new revision points to the copy

Persistent Segment Tree

- we use pointer like implicit segtree
- every time we modify a vertex, we make a copy
- the old revision still points to the original
- the new revision points to the copy
- update returns the root of the new revision

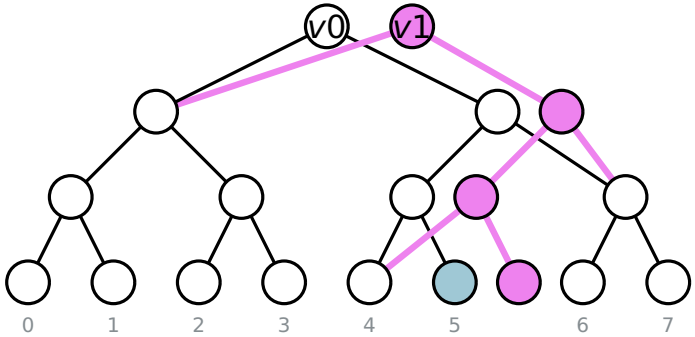
Persistent Segment Tree

- we use pointer like implicit segtree
- every time we modify a vertex, we make a copy
- the old revision still points to the original
- the new revision points to the copy
- update returns the root of the new revision



Persistent Segment Tree

- we use pointer like implicit segtree
- every time we modify a vertex, we make a copy
- the old revision still points to the original
- the new revision points to the copy
- update returns the root of the new revision

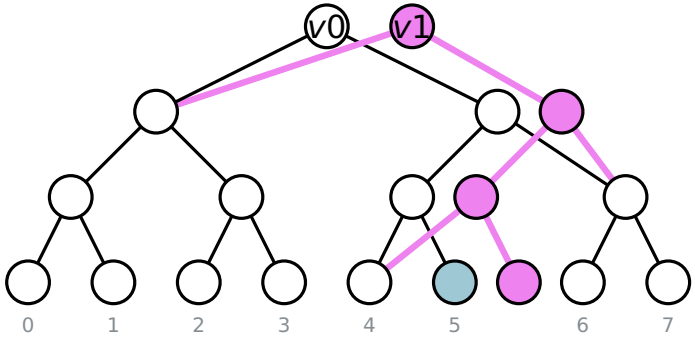


Example after updating position 5

Persistent Segment Tree

- we use pointer like implicit segtree
- every time we modify a vertex, we make a copy
- the old revision still points to the original
- the new revision points to the copy
- update returns the root of the new revision

```
struct Vertex {
    Vertex *l, *r;
    int sum;
    // c-tors
};
```



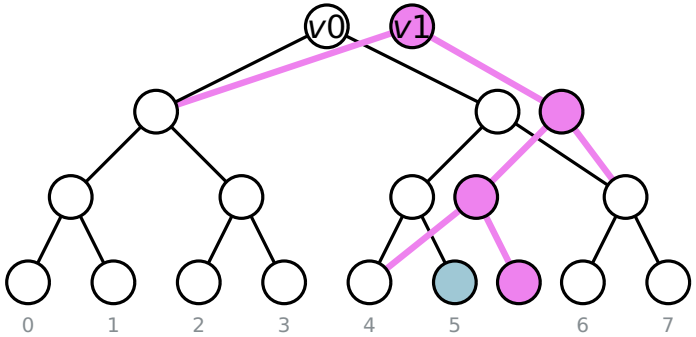
Example after updating position 5

Persistent Segment Tree

- we use pointer like implicit segtree
- every time we modify a vertex, we make a copy
- the old revision still points to the original
- the new revision points to the copy
- update returns the root of the new revision

```
struct Vertex {
    Vertex *l, *r;
    int sum;
    // c-tors
};
```

```
Vertex* update(Vertex* v, int l, int r,
               int pos, int new_val) {
    if (r-l == 1)
        return new Vertex(new_val);
    int m = (l + r) / 2;
    if (pos < m)
        return new Vertex(
            update(v->l, l, m, pos, new_val),
            v->r);
    else
        return new Vertex(
            v->l,
            update(v->r, m, r, pos, new_val));
}
```



Example after updating position 5