Algorithms for Programming Contests - Week 10

Tobias Meggendorfer, Philipp Meyer,
Christian Müller, Gregor Schwarz
conpra@in.tum.de

09.01.2019

Trie

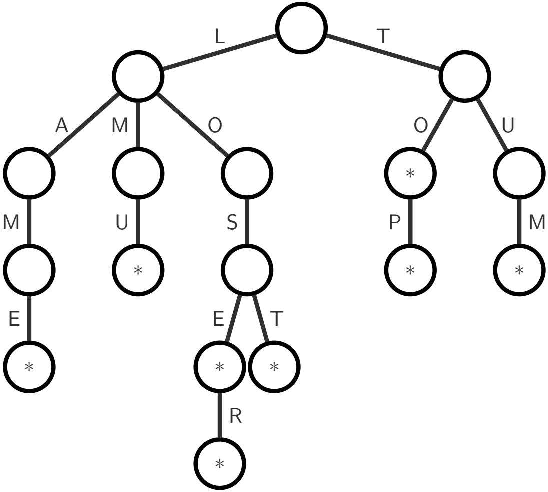
A *Trie* is a tree data structure that is used to store a set of strings:

- Root node represents empty string.
- Outgoing edges are associated with a label (e.g. a letter).
- Path from root to a node represents a prefix of a word/words.
- All descendants of a node have the same prefix.
- Position of a node in the trie defines the associated string.
- Nodes at which a word ends are tagged.

- Invented by René de la Briandais in 1959.
- Name originates from the term *retrieval*.

Trie - Example

Trie for words: LAME, LMU, LOSE, LOSER, LOST, TOP, TO, TUM



Trie - Implementation

A node of the trie contains:

- A label indicating whether a word ends at this node.
- A map/hashmap/array representing edges to children.

```
struct Node
{
    bool end;
    map<char,Node> next;
};
```

Trie - Operations

LOOKUP(S)

- Start at the root node and traverse edges w.r.t. letters of s.
- If no suitable edge exist, s is not contained in the trie.
- If the whole word was processed, check whether the current node marks the end of a word.

Trie - Operations

LOOKUP(S)

- Start at the root node and traverse edges w.r.t. letters of s.
- If no suitable edge exist, s is not contained in the trie.
- If the whole word was processed, check whether the current node marks the end of a word.

INSERT(S)

- Start at the root node and traverse edges w.r.t. letters of s.
- Add non-existing edges along the way.
- Tag last node.

Trie - Operations

LOOKUP(S)

- Start at the root node and traverse edges w.r.t. letters of s.
- If no suitable edge exist, s is not contained in the trie.
- If the whole word was processed, check whether the current node marks the end of a word.

INSERT(S)

- Start at the root node and traverse edges w.r.t. letters of s.
- Add non-existing edges along the way.
- Tag last node.

Complexity

- Lookup a string s of length k is in $\mathcal{O}(k)$.
- Inserting a string s of length k is in $\mathcal{O}(k)$.

Trie - Applications

- Storing a dynamic set of strings.
- Sorting strings lexicographically.
- Autocompletion
- Spell-checking

Segment Tree - Motivation

Given an integer array $a[n]$. Implement two operations:

- $\textsc{Add}(i, v)$: Add value v to i-th element.
- $\textsc{Sum}(\ell, r)$: Sum up all elements in interval $[\ell, r]$.

Segment Tree - Motivation

Given an integer array $a[n]$. Implement two operations:

- $\textsc{Add}(i, v)$: Add value v to i-th element.
- $\textsc{Sum}(\ell, r)$: Sum up all elements in interval $[\ell, r]$.

Alg 1:

- $\textsc{Add}(i, v)$: Simply add v to $a[i]$.
- $\textsc{Sum}(\ell, r)$: Loop from $a[\ell]$ to $a[r]$ and sum up values.
- Complexity: $\textsc{Add}(i, v) \in \mathcal{O}(1)$, $\textsc{Sum}(\ell, r) \in \mathcal{O}(n)$

Segment Tree - Motivation

Alg 2:

- Compute prefix sums and store them in $b[]$.
- $\textsc{Add}(i, v)$: Add v to $a[i]$ and recompute $b[]$.
- $\textsc{Sum}(\ell, r)$: Return $b[r] - b[\ell - 1]$.
- Complexity: $\textsc{Add}(i, v) \in \mathcal{O}(n)$, $\textsc{Sum}(\ell, r) \in \mathcal{O}(1)$

Segment Tree - Motivation

Alg 2:

- Compute prefix sums and store them in $b[]$.
- $\textsc{Add}(i, v)$: Add v to $a[i]$ and recompute $b[]$.
- $\textsc{Sum}(\ell, r)$: Return $b[r] - b[\ell - 1]$.
- Complexity: $\textsc{Add}(i, v) \in \mathcal{O}(n)$, $\textsc{Sum}(\ell, r) \in \mathcal{O}(1)$

Conclusion:

- Use Alg 1 if $\#\textsc{Add}$-Queries $\gg$ $\#\textsc{Sum}$-Queries.
- Use Alg 2 if $\#\textsc{Add}$-Queries $\ll$ $\#\textsc{Sum}$-Queries.

What if $\#\textsc{Add}$-Queries $\approx$ $\#\textsc{Sum}$-Queries?

Segment Tree

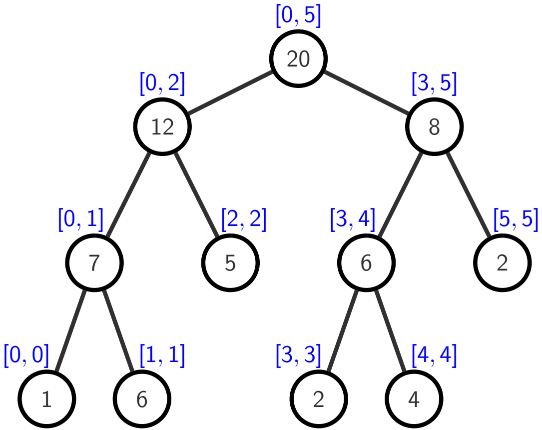
A *Segment Tree* is a tree data structure used for storing information about intervals.

- A segment tree is a binary tree.
- Each node stores a value v for an interval $[\ell, r]$.
- Root represents the full interval $[0, n-1]$.
- If node v represents interval $[\ell, r]$, its left child represents $[\ell, m]$ and its right child $[m+1, r]$ where $m = (\ell + r)/2$.
- Leaves represent unit-intervals $[t, t]$.

- Segment trees were invented by Jon Louis Bentley in 1977.

Segment Tree - Example

Input array: `a[] = [1,6,5,2,4,2]`



Segment Tree - Implementation

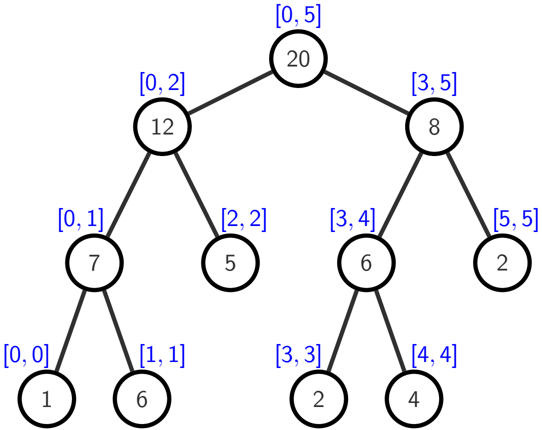
- Each node holds a value and the associated interval.

```
struct Node {int v,l,r;}
```

- Represent segment tree as an array of nodes.
- Root node is stored at index zero.
- Children of node i are stored at indices $2i+1$ and $2i+2$.
- Let n be the size of the input array:
  - Segment tree has height $h = \lceil \log n \rceil$.
  - Segment tree has $2^{h+1} - 1$ nodes.

Segment Tree - Example

Input array: `a[] = [1,6,5,2,4,2]`
Segment tree values: `v[] = [20,12,8,7,5,6,2,1,6,0,0,2,4,0,0]`



Segment Tree - Build

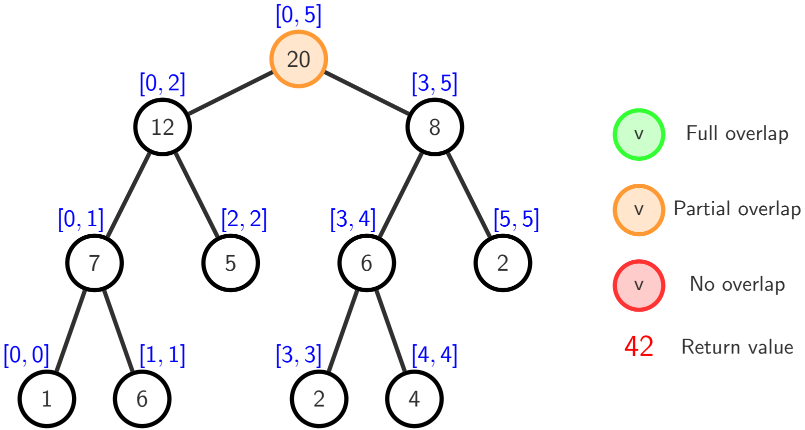
Algorithm 1 Segment Tree - Build

Input: input $a[]$, segment tree $t[]$, current index p, interval $[\ell, r]$.
Output: Segment tree rooted at p on interval $[\ell, r]$.

  procedure $\textsc{Build}(a[], t[], p, \ell, r)$
    $t[p].\ell \leftarrow \ell$
    $t[p].r \leftarrow r$
    if $\ell = r$ **then**
      $t[p].v \leftarrow a[\ell]$
      return $t[p].v$
    end if
    $m \leftarrow (l + r)/2$
    $t[p].v \leftarrow \textsc{Build}(a, t, 2p + 1, \ell, m) + \textsc{Build}(a, t, 2p + 2, m + 1, r)$
    return $t[p].v$
  end procedure

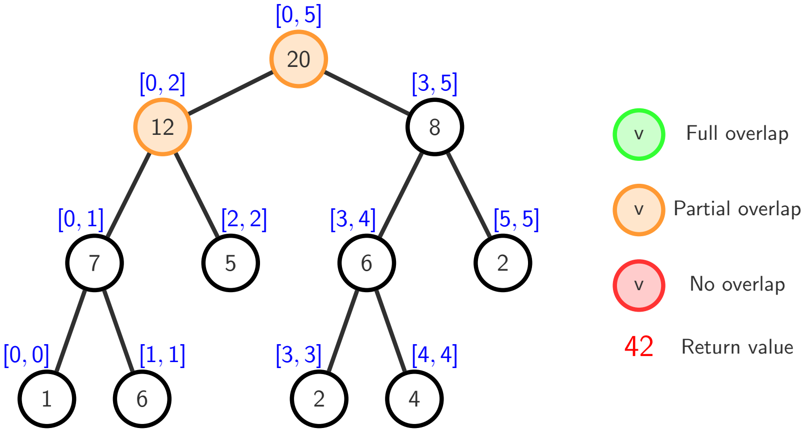
Segment Tree - Sum

SUM(1,5)



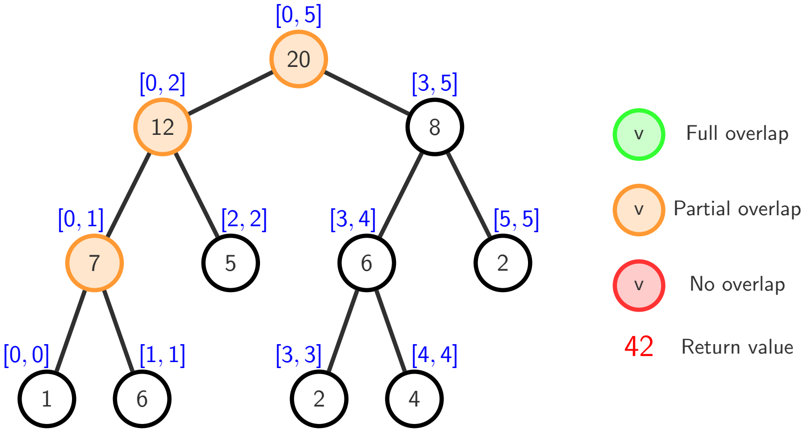
Segment Tree - Sum

SUM(1,5)



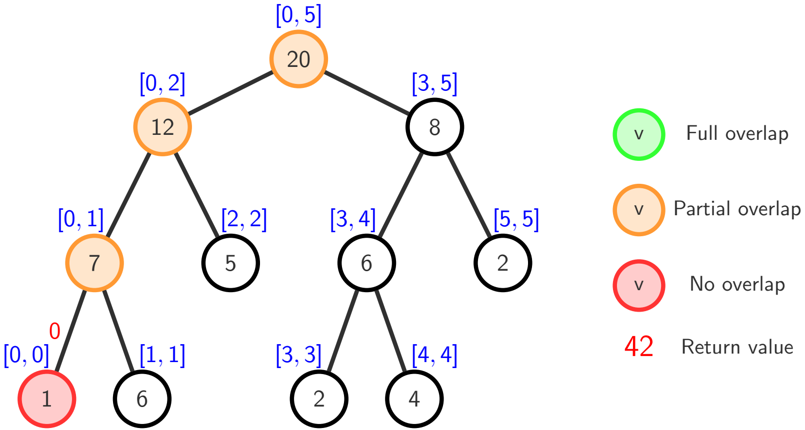
Segment Tree - Sum

SUM(1,5)



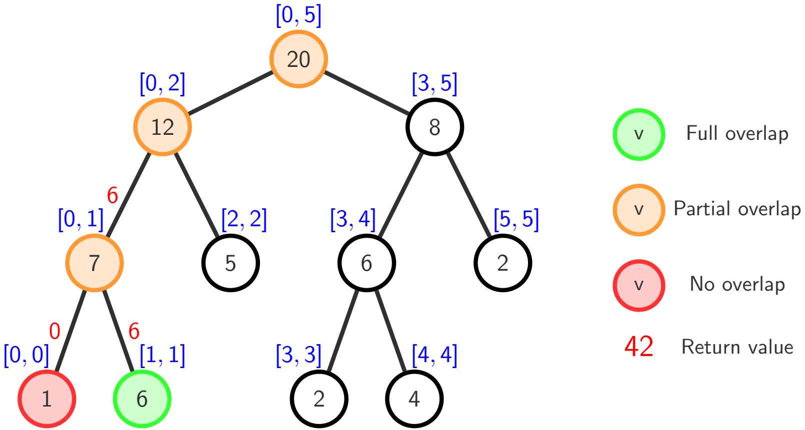
Segment Tree - Sum

SUM(1,5)



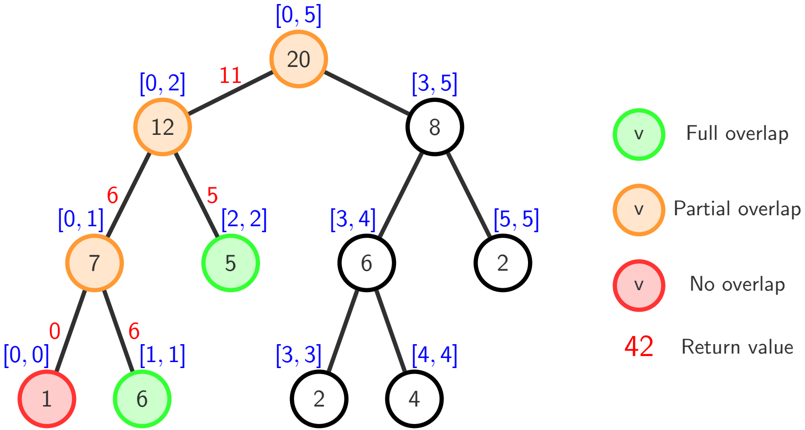
Segment Tree - Sum

SUM(1,5)



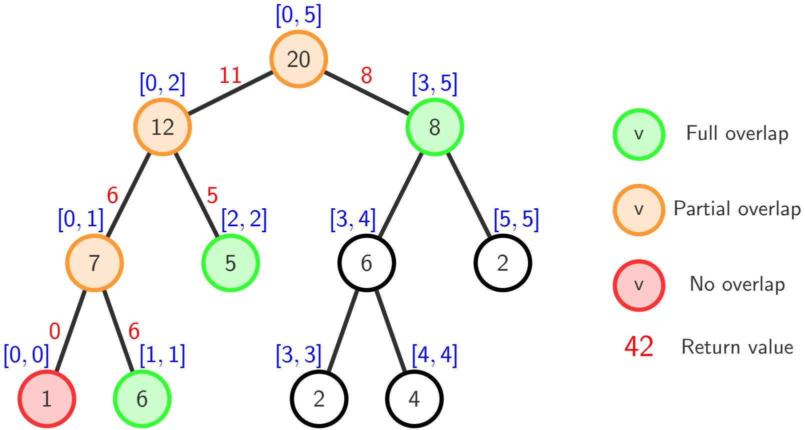
Segment Tree - Sum

SUM(1,5)



Segment Tree - Sum

$\textsc{Sum}(1,5) = 19$



Segment Tree - Sum

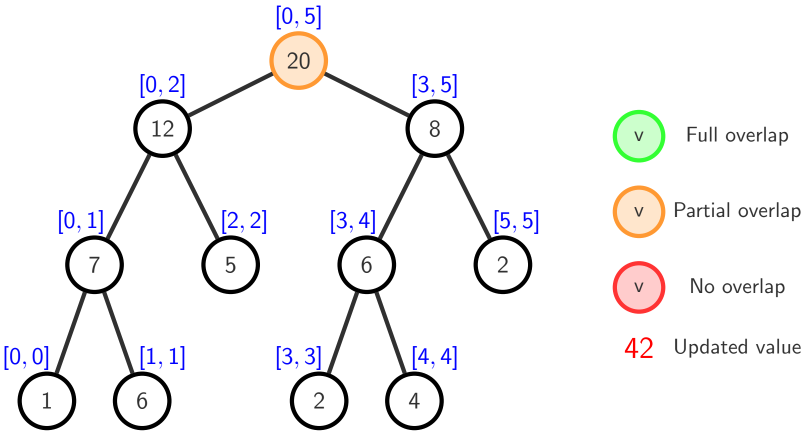
Algorithm 2 Segment Tree - Sum

Input: Segment tree $t[]$, current index p, interval $[\ell, r]$.
Output: Sum on interval $[\ell, r]$.

```
procedure SUM(t[], p, ℓ, r)
    if ℓ > t[p].r or r < t[p].ℓ then
        return 0
    end if
    if ℓ ≤ t[p].ℓ and t[p].r ≤ r then
        return t[p].v
    end if
    return SUM(t, 2p + 1, ℓ, r) + SUM(t, 2p + 2, ℓ, r)
end procedure
```

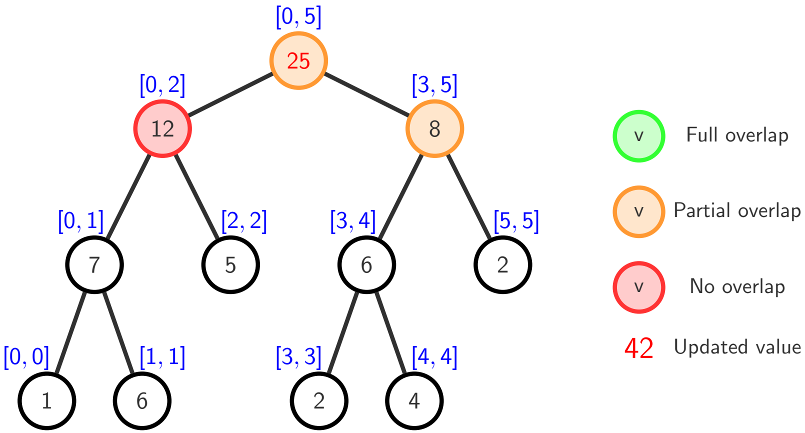
Segment Tree - Add

ADD 5 to index 3



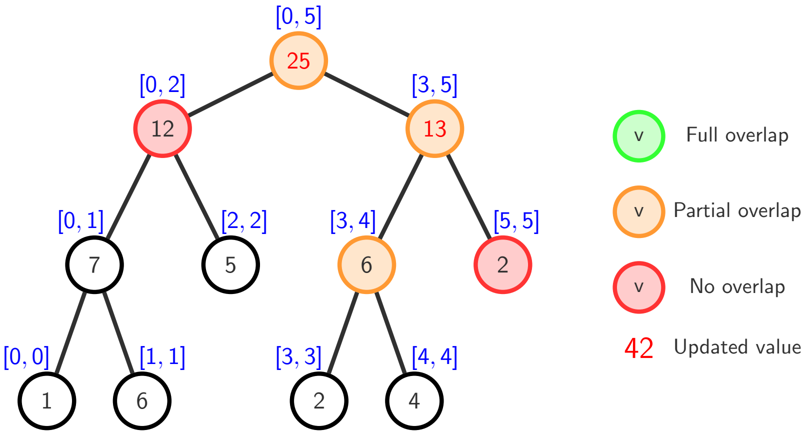
Segment Tree - Add

ADD 5 to index 3



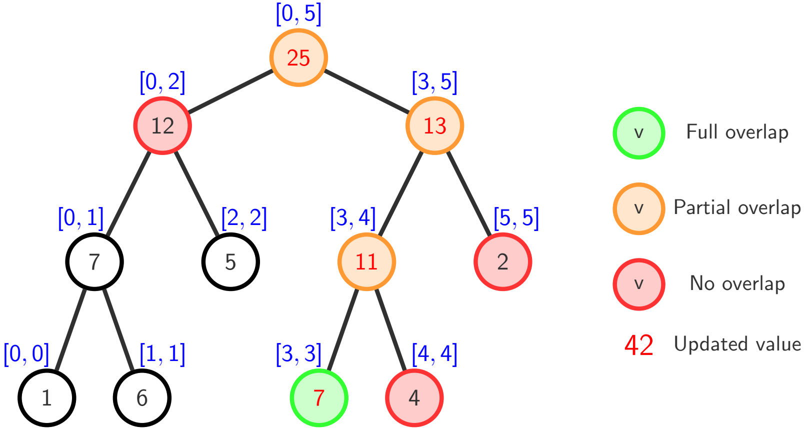
Segment Tree - Add

ADD 5 to index 3



Segment Tree - Add

ADD 5 to index 3



Segment Tree - Add

Algorithm 3 Segment Tree - Add

Input: Segment tree $t[]$, current index p, update index i, update value v.

```
procedure ADD(t[], p, i, v)
    if i < t[p].ℓ or i > t[p].r then
        return
    end if
    t[p] ← t[p] + v
    if t[p].ℓ ≠ t[p].r then
        ADD(t, 2p + 1, i, v)
        ADD(t, 2p + 2, i, v)
    end if
end procedure
```

Segment Tree - Complexity

Complexity

Let n be the size of the input array $a[]$.

- Build: Each of the $2n - 1$ nodes is visited once. $\mathcal{O}(n)$.
- Add: At most two nodes are visited on every level. $\mathcal{O}(\log n)$.
- Sum: At most four nodes are visited on every level. $\mathcal{O}(\log n)$.

Segment Tree - Operations

Segment trees do not only work for sums but for all monoids.

Segment Tree - Operations

Segment trees do not only work for sums but for all monoids.

A *monoid* is a set S with some binary operator $\bullet : S \times S \to S$ if it satisfies the following two axioms:

- Associativity: For all $x, y, z \in S$ it holds that $(x \bullet y) \bullet z = x \bullet (y \bullet z)$.
- Identity element: It exists an $e \in S$ such that for all $z \in S$ it holds that $e \bullet z = z \bullet e = z$.

Segment Tree - Operations

Segment trees do not only work for sums but for all monoids.

A *monoid* is a set S with some binary operator $\bullet : S \times S \to S$ if it satisfies the following two axioms:

- Associativity: For all $x, y, z \in S$ it holds that $(x \bullet y) \bullet z = x \bullet (y \bullet z)$.
- Identity element: It exists an $e \in S$ such that for all $z \in S$ it holds that $e \bullet z = z \bullet e = z$.

In particular they work for:

- $(\mathbb{R}, +)$
- $(\mathbb{R}, \min)$
- $(\mathbb{R}, \max)$
- $(2^{\mathbb{N}}, XOR)$

Segment Tree - RangeAdd

Implement a new operation:

- $\textsc{Add}(i, v)$: Add value v to i-th element.
- $\textsc{Sum}(\ell, r)$: Sum up all elements in interval $[\ell, r]$.
- $\textsc{RangeAdd}(\ell, r, v)$: Add value v to each element in range $[\ell, r]$.

Segment Tree - RangeAdd

Implement a new operation:

- $\textsc{Add}(i, v)$: Add value v to i-th element.
- $\textsc{Sum}(\ell, r)$: Sum up all elements in interval $[\ell, r]$.
- $\textsc{RangeAdd}(\ell, r, v)$: Add value v to each element in range $[\ell, r]$.

Naïve approach:

Call $\textsc{Add}(i, v)$ for all $i \in [\ell, r]$. Complexity: $\mathcal{O}(n \log n)$.

Can we do better?

Segment Tree - RangeAdd

Implement a new operation:

- $\textsc{Add}(i, v)$: Add value v to i-th element.
- $\textsc{Sum}(\ell, r)$: Sum up all elements in interval $[\ell, r]$.
- $\textsc{RangeAdd}(\ell, r, v)$: Add value v to each element in range $[\ell, r]$.

Naïve approach:

Call $\textsc{Add}(i, v)$ for all $i \in [\ell, r]$. Complexity: $\mathcal{O}(n \log n)$.

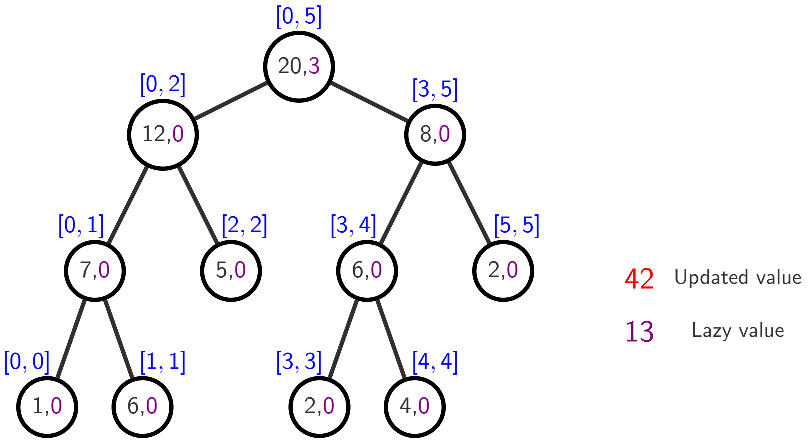
Can we do better? $\rightarrow$ Yes! Just be lazy...

Segment Tree - Lazy Propagation

- Store an additional integer value *lazy* in each node.
- Do not apply updates immediately but push them to the lazy variable.
- Only propagate lazy value to children when value of node is queried.

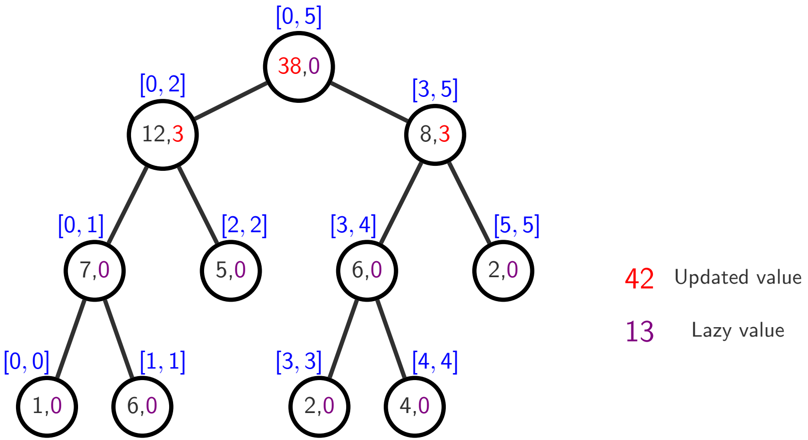
Segment Tree - Propagate

PROPAGATE lazy value of root node.



Segment Tree - Propagate

PROPAGATE lazy value of root node.



Segment Tree - Propagate

Algorithm 4 Segment Tree - Propagate

Input: Segment tree $t[]$, current index p.

procedure $\textsc{Propagate}(t[], p)$

  $t[p].v \leftarrow t[p].v + (t[p].r - t[p].\ell + 1) * t[p].lazy$

  if $t[p].\ell \neq t[p].r$ **then**

    $t[2p+1].lazy \leftarrow t[2p+1].lazy + t[p].lazy$

    $t[2p+2].lazy \leftarrow t[2p+2].lazy + t[p].lazy$

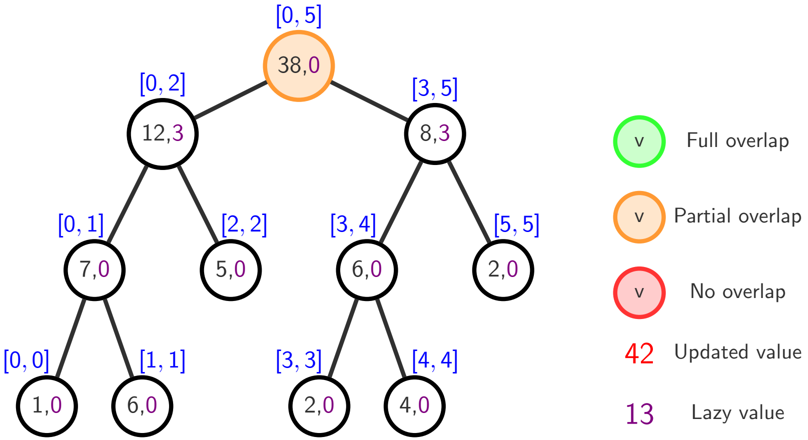
  end if

  $t[p].lazy \leftarrow 0$

end procedure

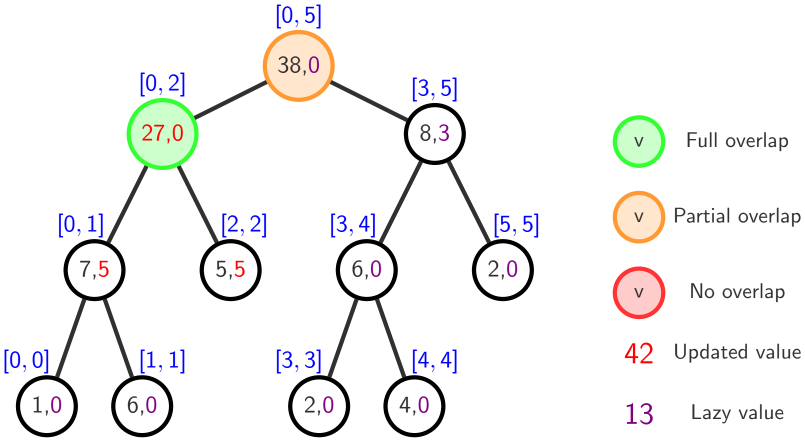
Segment Tree - Propagate

RANGEADD 2 on interval $[0, 4] \longrightarrow$ PROPAGATE root



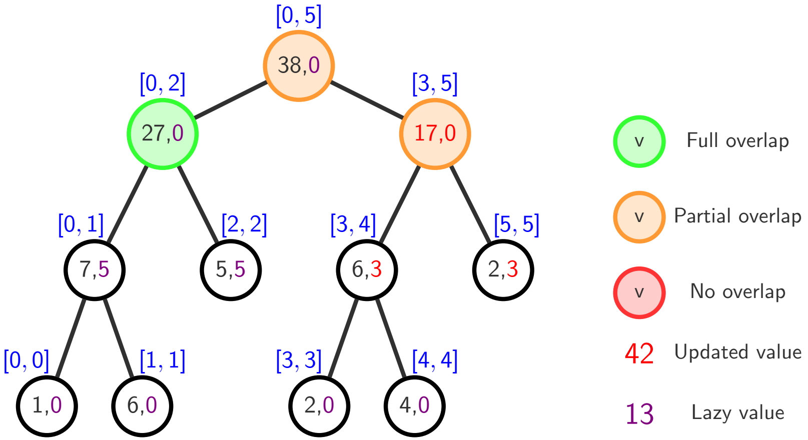
Segment Tree - Propagate

RANGEADD 2 on interval $[0,4] \longrightarrow$ PROPAGATE node 1



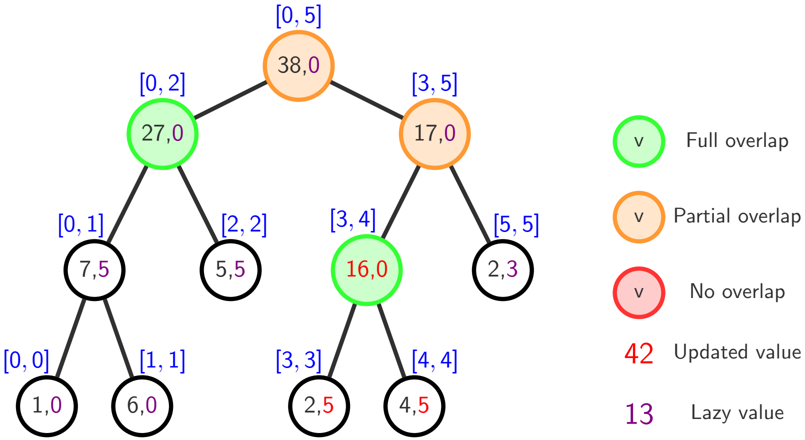
Segment Tree - Propagate

RangeAdd 2 on interval $[0, 4] \longrightarrow$ Propagate node 2



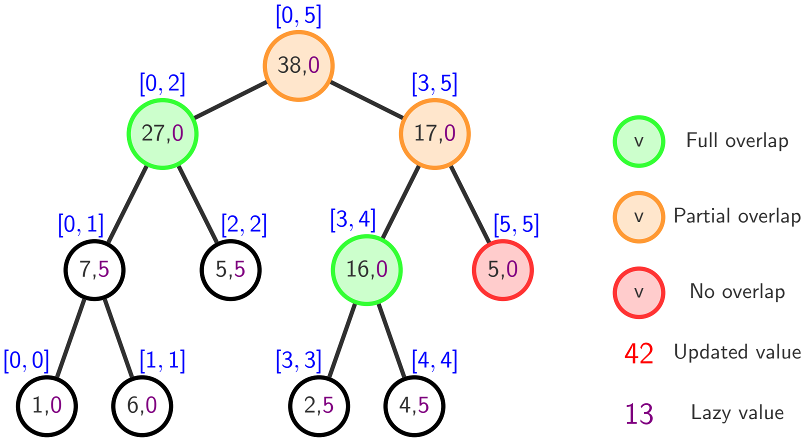
Segment Tree - Propagate

RANGEADD 2 on interval $[0, 4] \longrightarrow$ PROPAGATE node 5



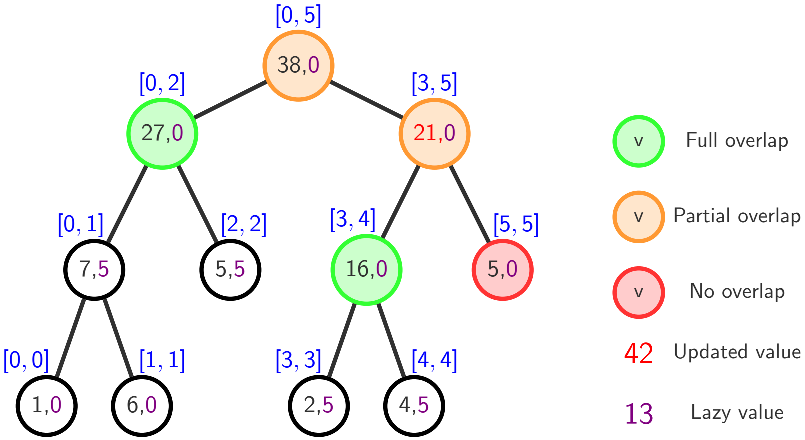
Segment Tree - Propagate

RANGEADD 2 on interval $[0,4] \longrightarrow$ PROPAGATE node 6



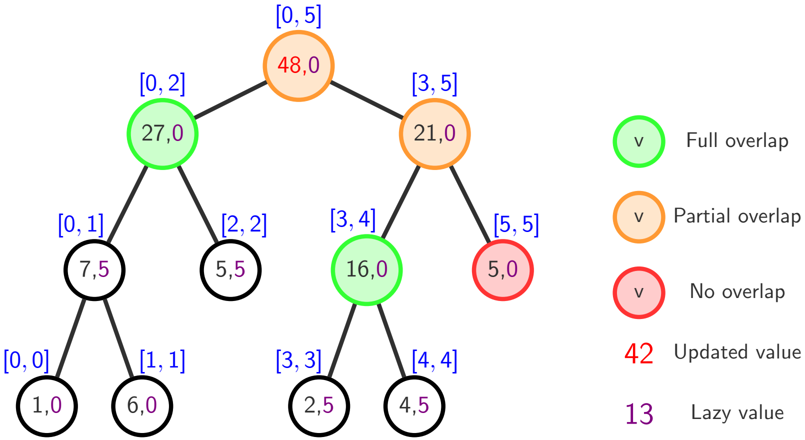
Segment Tree - Propagate

RangeAdd 2 on interval $[0, 4] \longrightarrow$ Update node 1



Segment Tree - Propagate

RangeAdd 2 on interval $[0, 4] \longrightarrow$ Update root



Segment Tree - RangeAdd

Algorithm 5 Segment Tree - RangeAdd

Input: Segment tree $t[]$, current index p, interval $[\ell, r]$, value v.

```
procedure RangeAdd(t[], p, ℓ, r, v)
    Propagate(t, p)
    if ℓ > t[p].r or r < t[p].ℓ then
        return
    end if
    if ℓ ≤ t[p].ℓ and t[p].r ≤ r then
        t[p].lazy ← t[p].lazy + v
        Propagate(t, p)
    else if t[p].ℓ ≠ t[p].r then
        RangeAdd(t, 2p + 1, ℓ, r, v)
        RangeAdd(t, 2p + 2, ℓ, r, v)
        t[p].v ← t[2p + 1].v + t[2p + 2].v
    end if
end procedure
```

Segment Tree - Sum

Algorithm 6 Segment Tree - Sum

Input: Segment tree $t[]$, current index p, interval $[\ell, r]$.
Output: Sum on interval $[\ell, r]$.

```
procedure Sum(t[], p, ℓ, r)
    if ℓ > t[p].r or r < t[p].ℓ then
        return 0
    end if
    Propagate(t, p)
    if ℓ ≤ t[p].ℓ and t[p].r ≤ r then
        return t[p].v
    end if
    return Sum(t, 2p + 1, ℓ, r) + Sum(t, 2p + 2, ℓ, r)
end procedure
```

Segment Tree - Complexity

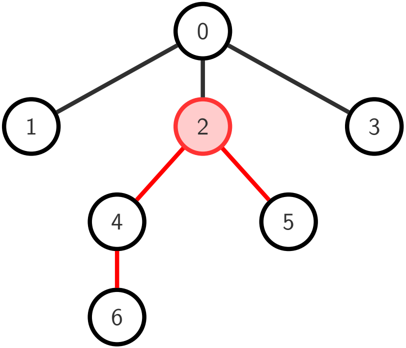
Complexity

Let n be the size of the input array $a[]$.

- BUILD: Each of the $2n - 1$ nodes is visited once. $\mathcal{O}(n)$.
- ADD: At most two nodes are visited on every level. $\mathcal{O}(\log n)$.
- SUM: At most four nodes are visited on every level. $\mathcal{O}(\log n)$.
- RANGEADD: At most four nodes are visited on every level. $\mathcal{O}(\log n)$.

LCA - Example

The *Lowest Common Ancestor (LCA)* of two nodes *u* and *v* in a tree is the deepest node that has both *u* and *v* as descendants.
(A node is assumed to be a descendant of itself.)



$\mathrm{LCA}(5, 6) = 2$

LCA - Computation

Naïve approach to compute $\mathrm{LCA}(u, v)$:

- Compute the path from root to u and v.
- Find the first entry at which both paths differ.
- The LCA is the node right before this mismatch.

LCA - Computation

Naïve approach to compute $\mathrm{LCA}(u, v)$:

- Compute the path from root to u and v.
- Find the first entry at which both paths differ.
- The LCA is the node right before this mismatch.
- Example above:
  - Path from root to node 5: $(0, 2, 5)$
  - Path from root to node 6: $(0, 2, 4, 6)$
  - LCA is 2.
- Complexity: $\mathcal{O}(n)$, where n is the number of nodes in the tree.

LCA - Computation

Naïve approach to compute $\mathrm{LCA}(u, v)$:

- Compute the path from root to u and v.
- Find the first entry at which both paths differ.
- The LCA is the node right before this mismatch.
- Example above:
  - Path from root to node 5: $(0, 2, 5)$
  - Path from root to node 6: $(0, 2, 4, 6)$
  - LCA is 2.
- Complexity: $\mathcal{O}(n)$, where n is the number of nodes in the tree.

We can do better using segment trees!

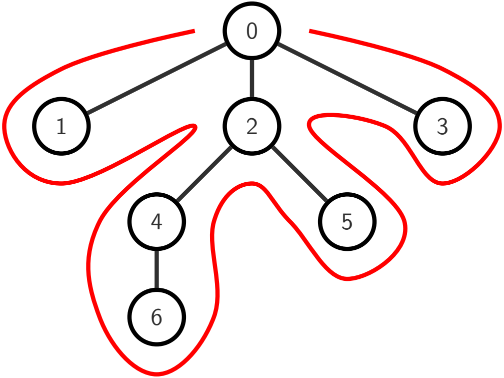
LCA - Eulerian Tour Technique

The *Eulerian Tour Technique* is a special representation of trees:

- Replace every undirected edge $\{u, v\}$ by two directed edges (u, v) and (v, u).
- Compute an Eulerian cycle starting from the root.

The *Euler Tour Representation* (ETR) of a tree is the traversal order of nodes in the Eulerian cycle.

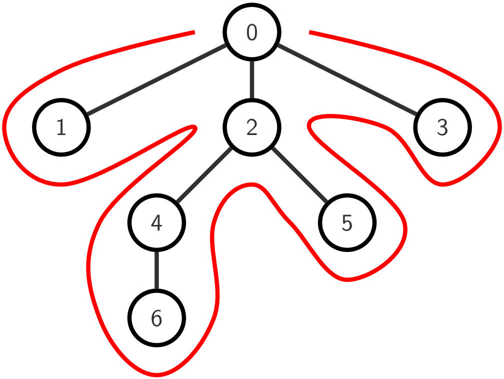
LCA - Eulerian Tour Representation



idx	0	1	2	3	4	5	6	7	8	9	10	11	12
ETR	0	1	0	2	4	6	4	2	5	2	0	3	0
depth	0	1	0	1	2	3	2	1	2	1	0	1	0

first visit

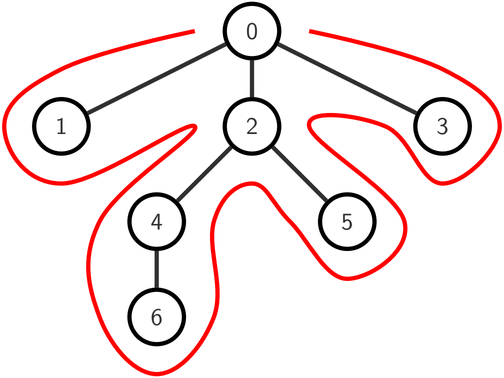
LCA - Eulerian Tour Representation



idx	0	1	2	3	4	5	6	7	8	9	10	11	12
ETR	0	1	0	2	4	6	4	2	5	2	0	3	0
depth	0	1	0	1	2	3	2	1	2	1	0	1	0

Depth of $\text{LCA}(5,6)$ is 1, and $\text{LCA}(5,6) = 2$.

LCA - Eulerian Tour Representation



idx	0	1	2	3	4	5	6	7	8	9	10	11	12
ETR	0	1	0	2	4	6	4	2	5	2	0	3	0
depth	0	1	0	1	2	3	2	1	2	1	0	1	0

Depth of $\mathrm{LCA}(1,3)$ is 0, and $\mathrm{LCA}(5,6) = 0$.

LCA - Computation

How to compute LCA(u, v)?

- Compute the ETR of the tree.
- Compute the depths corresponding to the nodes in the ETR.
- Store at which index a node is first visited in the ETR.
- Build a segment tree on the depth array using the mininum operator.
- LCA(u, v) is the node associated to the minimum in the interval $[x, y]$ of the depth array, where x and y are the indices of the first occurrences of u and v in the ETR.

LCA - Complexity

Complexity

- Computing the ETR requires a tree traversal. $\mathcal{O}(n)$.
- Building the segment tree on the depth array. $\mathcal{O}(n)$.
- Any further computation of $\text{LCA}(u, v)$ requires one minimum query in the segment tree. $\mathcal{O}(\log n)$.