Algorithms for Programming Contests - Week 11

Prof. Dr. Javier Esparza
Pranav Ashok, A. R. Balasubramanian,
Tobias Meggendorfer, Philipp Meyer,
Mikhail Raskin,
conpra@in.tum.de

30. Juni 2020

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.

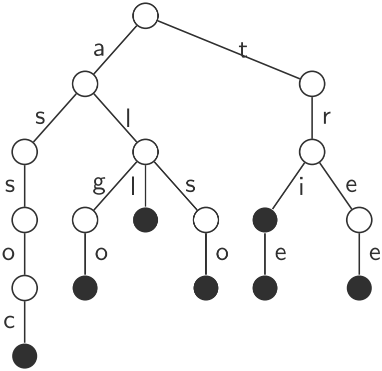
- Invented by René de la Briandais in 1959.
- Name originates from the term *retrieval*.

Trie - Applications

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

Trie

The next figure shows a trie with the words “tree”, “trie”, ”tri”, “algo”, “assoc”, “all”, and “also”.



Trie - Implementation details

As an example, an array of the following struct can be used:

```
struct node{
    int references[26]; // Size of Alphabet
    bool end;
};
```

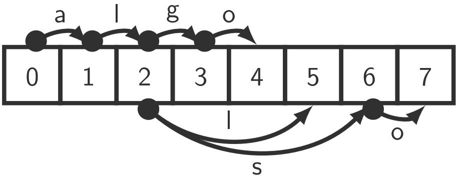
Or

```
int references[N][26]; // size of Alphabet
bool end[N];

int lastId;
//lastId shows the index of last nodes
```

Trie - Implementation details

You may have an array of nodes and references can be used as an index reference. Here is an example of a storage structure for the words inserted in this order: “algo”, “all”, “also”



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 as the end of a word.

Trie - Additional information

Complexity:

- Insertion in $\mathcal{O}(L)$ time,
- Look-up in $\mathcal{O}(L)$ time.

Want to know something more:

- Compressed Tries,
  useful for `switch()` on string values
- Adaptive Radix Tree - an interesting index structure for main-memory databases.

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

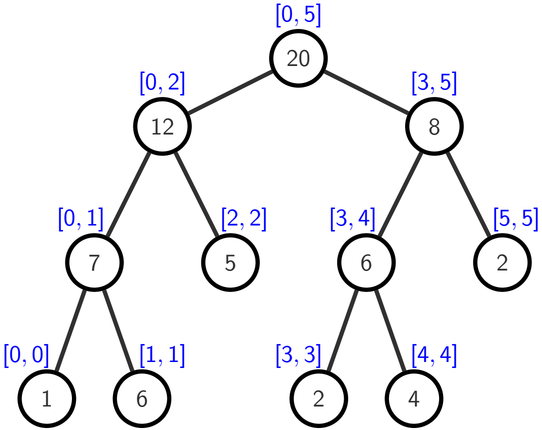
A *Segment Tree* is a tree data structure used for storing information about intervals.

- 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;}
```

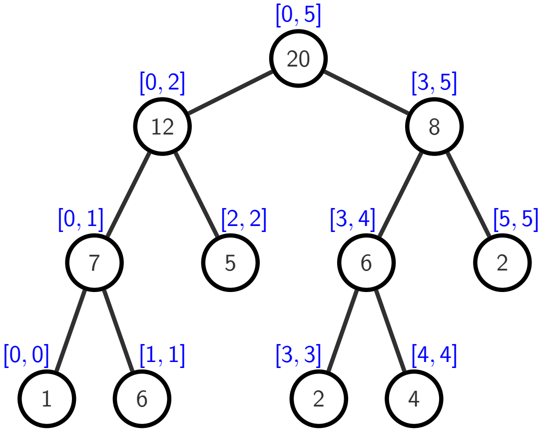
- Represent segment tree as an array of nodes.
- Root node is stored at index zero.
- Children of node i are stored at indices $2i+1$ and $2i+2$.
- Let n be the size of the input array:
  - Segment tree has height $h = \lceil \log n \rceil$.
  - Segment tree has $2^{h+1} - 1$ nodes.

Segment Tree - Implementation tricks

- Round n up to a power of 2, fill tail with 0
- Root node is stored at index 1 (index 0 unused)
- Children of node i are stored at indices $2i$ and $2i+1$.
- Node at level l out of h containing x:
  $(x/(2^{h-l}))+2^l=(x>>(h-l))+(1<<l)$
- Its endpoints: $(x>>(h-l))<<(h-l)$ and
  $((x>>(h-l))+1)<<(h-l)-1$
- Only store node values

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

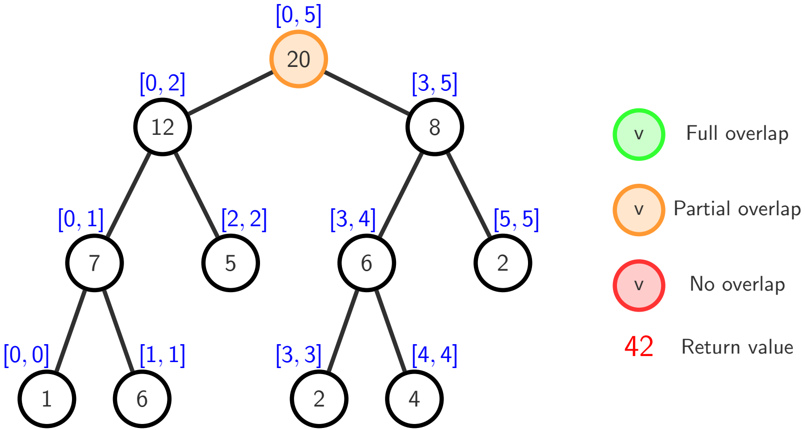
Algorithm 1 Segment Tree - Build

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

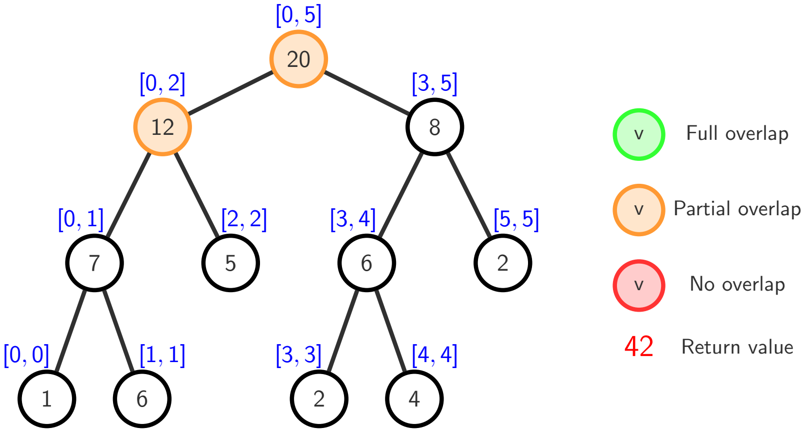
Segment Tree - Sum

SUM(1,5)



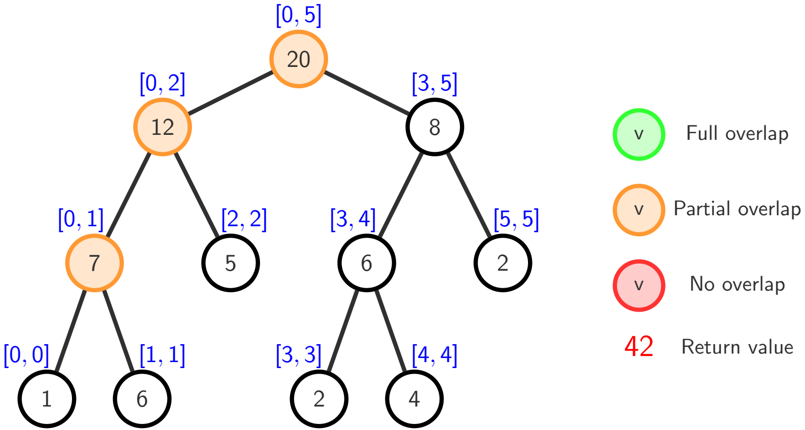
Segment Tree - Sum

SUM(1,5)



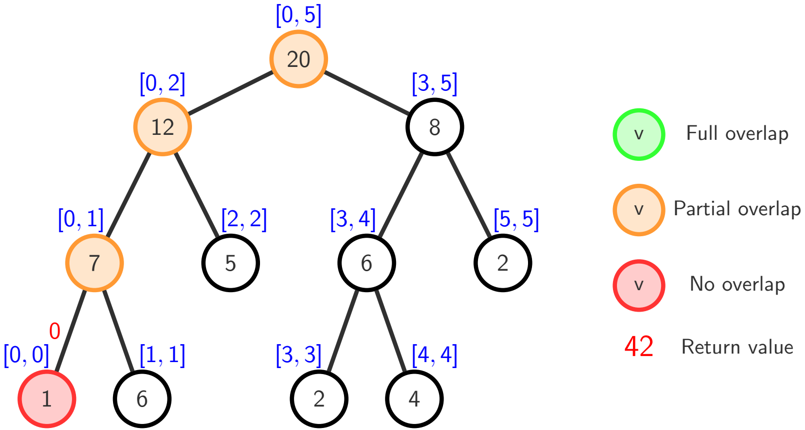
Segment Tree - Sum

SUM(1,5)



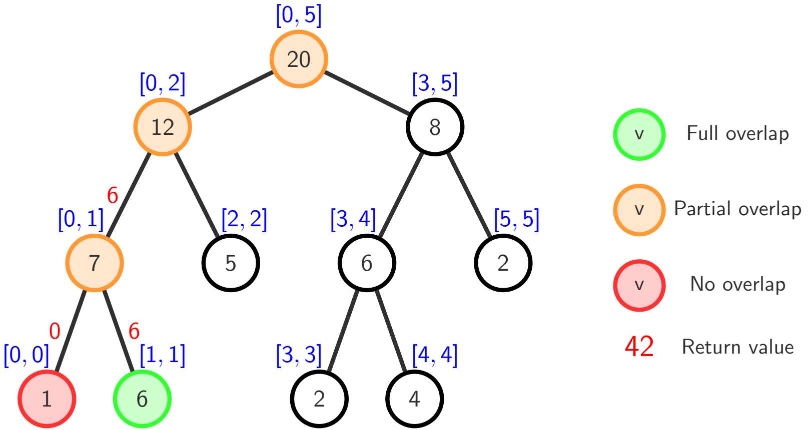
Segment Tree - Sum

SUM(1,5)



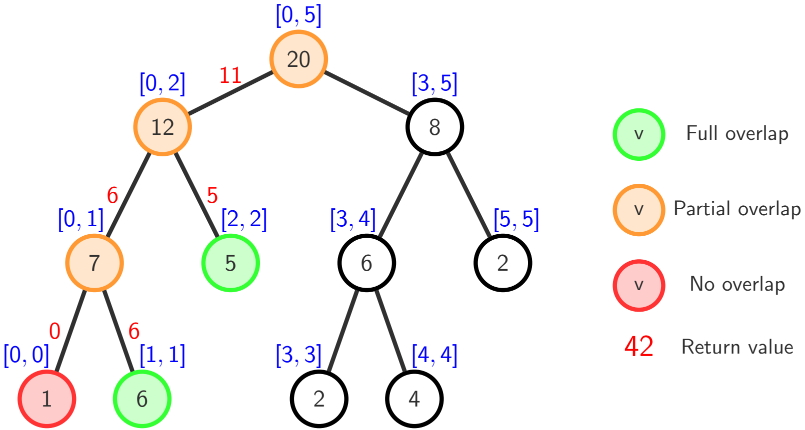
Segment Tree - Sum

SUM(1,5)



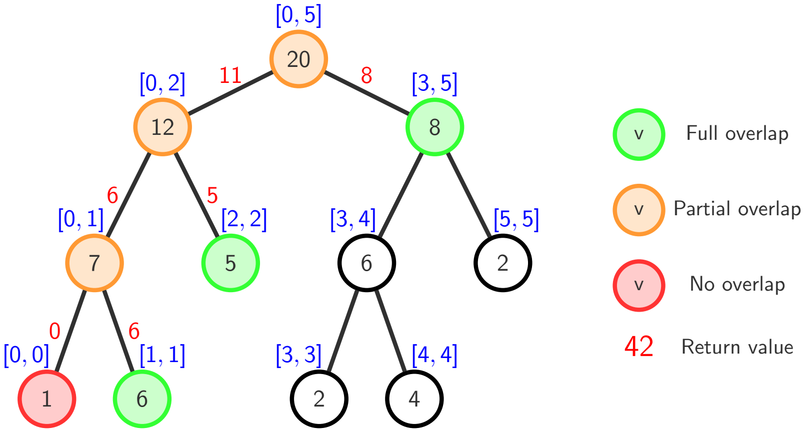
Segment Tree - Sum

SUM(1,5)



Segment Tree - Sum

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



Segment Tree - Sum

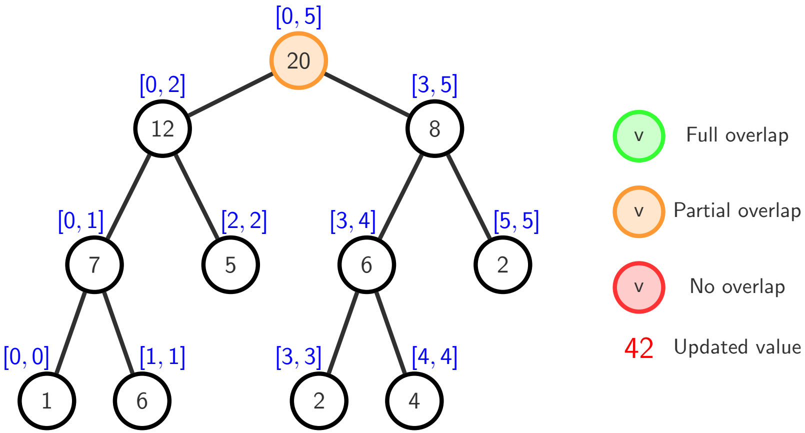
Algorithm 2 Segment Tree - Sum

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

procedure $\textsc{Sum}(t[], p, \ell, r)$
  if $\ell > t[p].r$ **or** $r < t[p].\ell$ **then**
    return 0
  if $\ell \leq t[p].\ell$ **and** $t[p].r \leq r$ **then**
    return $t[p].v$
  return $\textsc{Sum}(t, 2p+1, \ell, r) + \textsc{Sum}(t, 2p+2, \ell, r)$

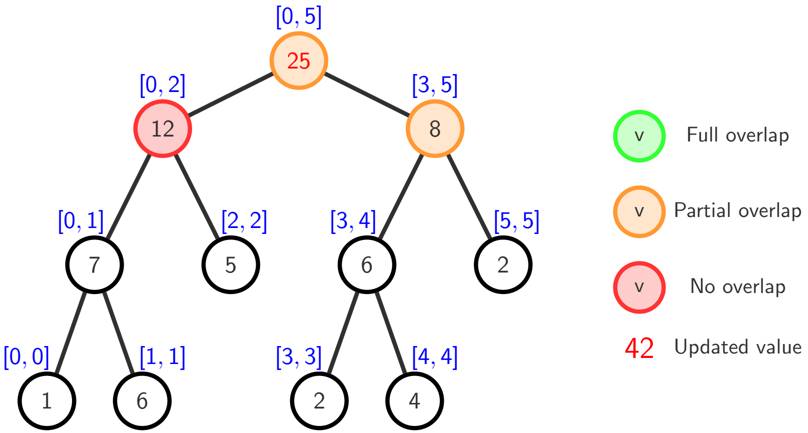
Segment Tree - Add

ADD 5 to index 3



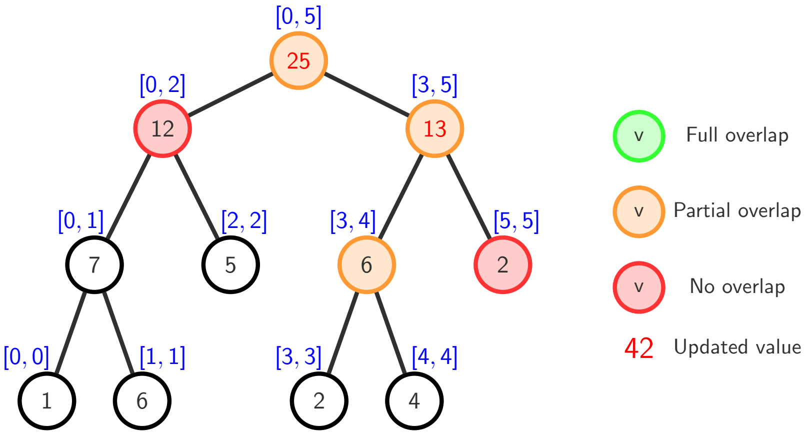
Segment Tree - Add

ADD 5 to index 3



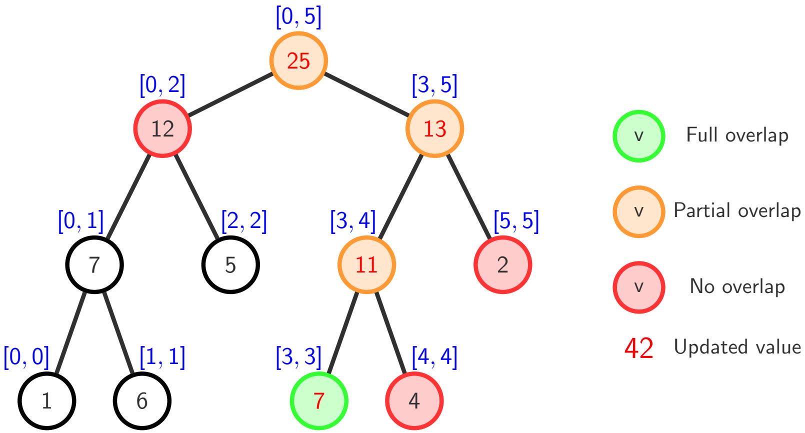
Segment Tree - Add

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
    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)
```

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. $\mathcal{O}(\log n)$.

Iterating with four variables for nodes often faster than recursion
Explicit stack instead of recursion: often faster, sometimes simpler than iteration

Segment Tree - Operations

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

A *semigroup* is a set S with some associative binary operator $\bullet : S \times S \rightarrow S$.
Associativity: For all $x, y, z \in S$ it holds that $(x \bullet y) \bullet z = x \bullet (y \bullet z)$.

In particular they work for:

- $(\mathbb{R}, +)$
- $(\mathbb{R}, \min)$
- $(\mathbb{R}, \max)$
- $(2^{\mathbb{N}}, XOR)$
- Matrices with matrix multiplication (take care of order!)

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: $\mathcal{O}(n \log n)$.
Update each node intersecting with the range. Complexity: $\mathcal{O}(n)$.

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: $\mathcal{O}(n \log n)$.
Update each node intersecting with the range. Complexity: $\mathcal{O}(n)$.

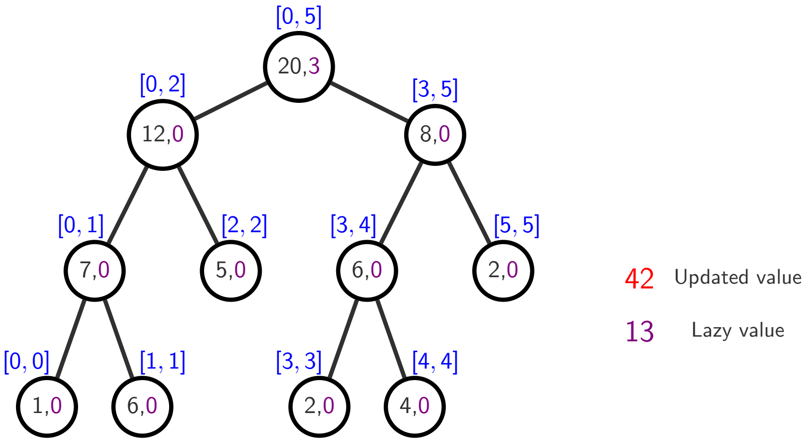
Can we do better? $\rightarrow$ Yes! Just be lazy...

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.

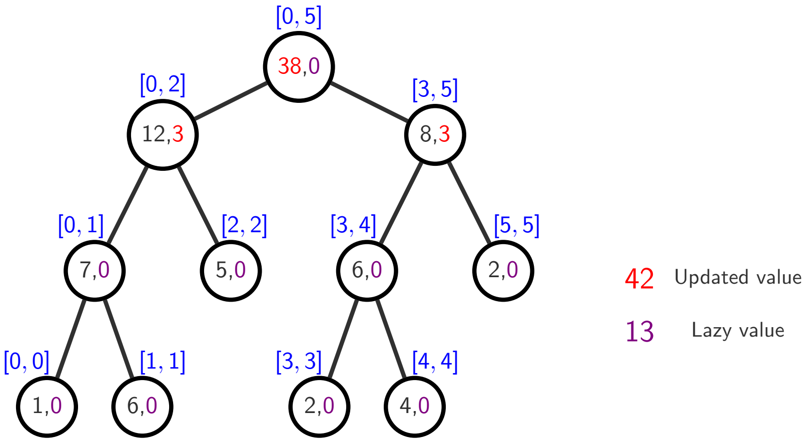
Segment Tree - Propagate

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**

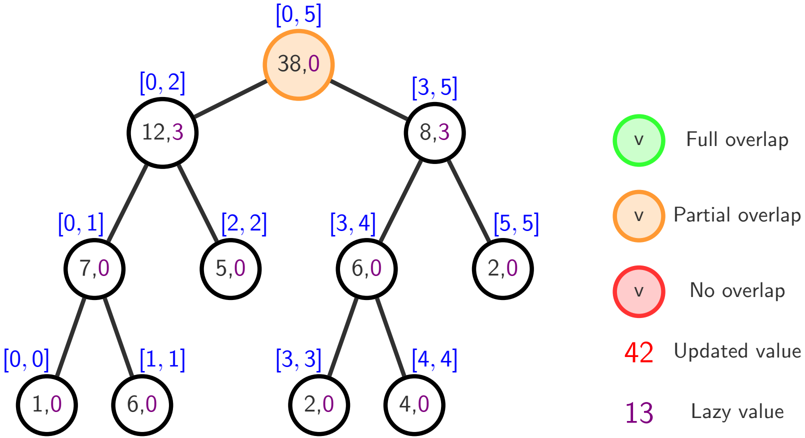
$t[2p+1].lazy \leftarrow t[2p+1].lazy + t[p].lazy$

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

$t[p].lazy \leftarrow 0$

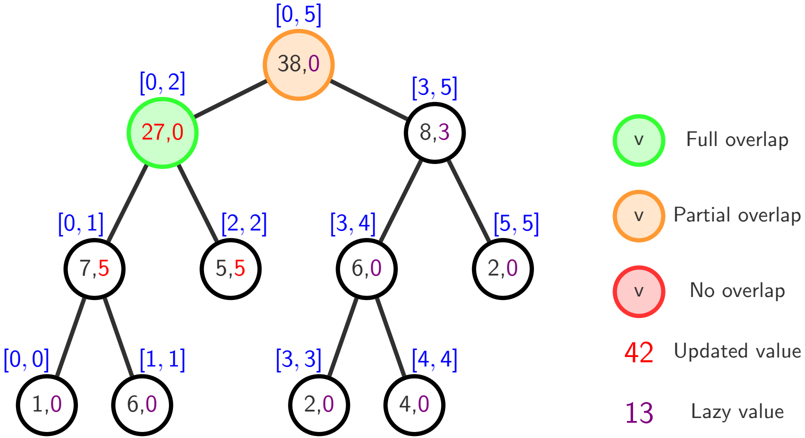
Segment Tree - Propagate

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



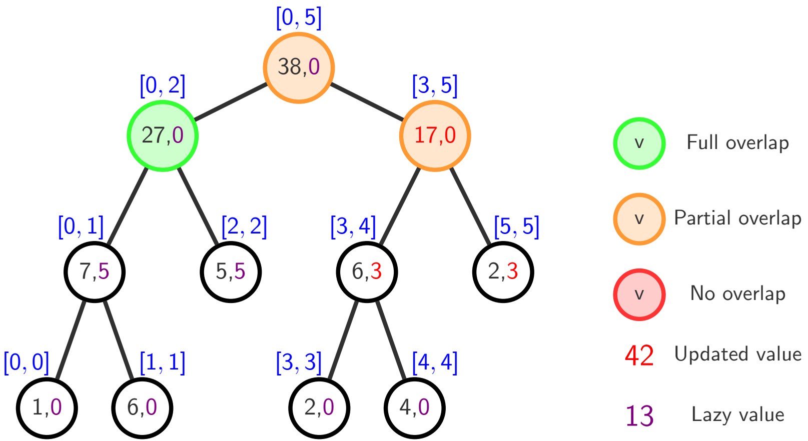
Segment Tree - Propagate

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



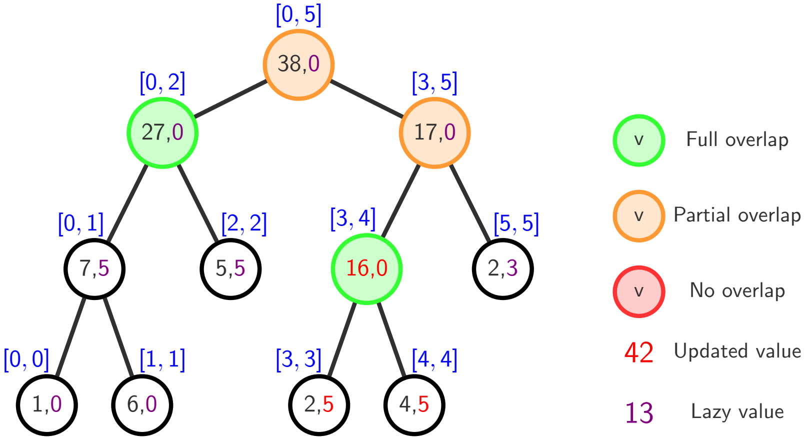
Segment Tree - Propagate

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



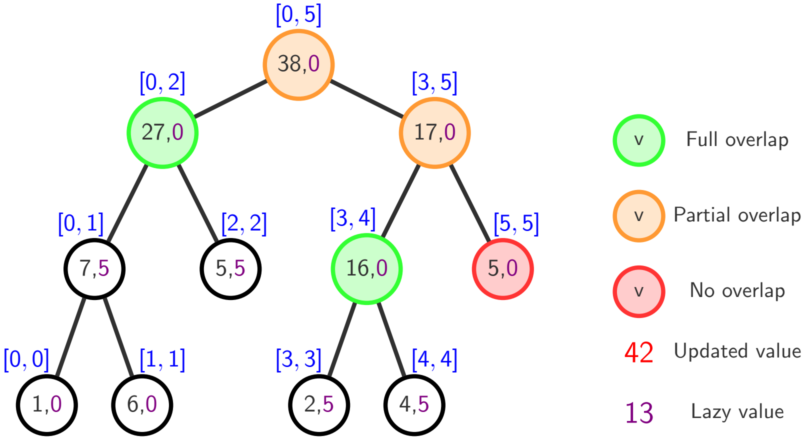
Segment Tree - Propagate

RangeAdd 2 on interval $[0, 4] \longrightarrow$ Propagate node 5



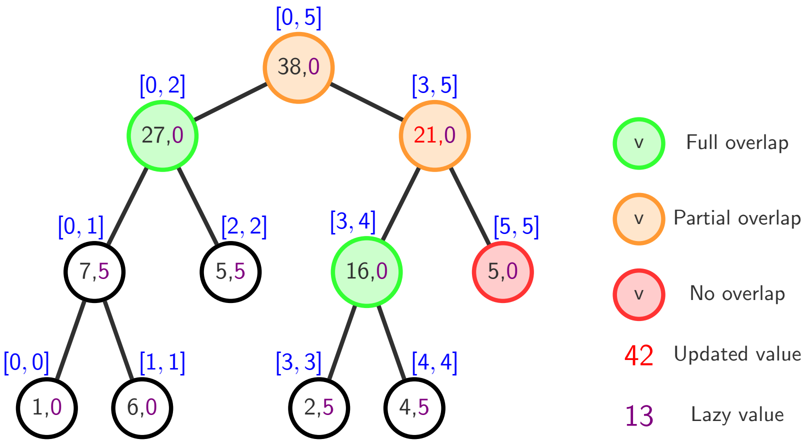
Segment Tree - Propagate

RangeAdd 2 on interval $[0, 4] \longrightarrow$ Propagate node 6



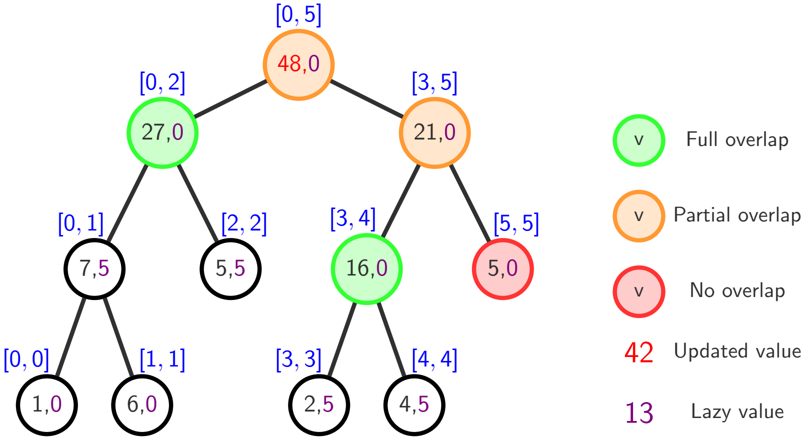
Segment Tree - Propagate

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 $\textsc{RangeAdd}(t[], p, \ell, r, v)$
  $\textsc{Propagate}(t, p)$
  if $\ell > t[p].r$ **or** $r < t[p].\ell$ **then**
    return
  if $\ell \leq t[p].\ell$ **and** $t[p].r \leq r$ **then**
    $t[p].lazy \leftarrow t[p].lazy + v$
    $\textsc{Propagate}(t, p)$
  else if $t[p].\ell \neq t[p].r$ **then**
    $\textsc{RangeAdd}(t, 2p+1, \ell, r, v)$
    $\textsc{RangeAdd}(t, 2p+2, \ell, r, v)$
    $t[p].v \leftarrow t[2p+1].v + t[2p+2].v$

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 $\textsc{Sum}(t[], p, \ell, r)$
  if $\ell > t[p].r$ **or** $r < t[p].\ell$ **then**
    return 0
  $\textsc{Propagate}(t, p)$
  if $\ell \leq t[p].\ell$ **and** $t[p].r \leq r$ **then**
    return $t[p].v$
  return $\textsc{Sum}(t, 2p+1, \ell, r) + \textsc{Sum}(t, 2p+2, \ell, r)$

Segment Tree - Complexity

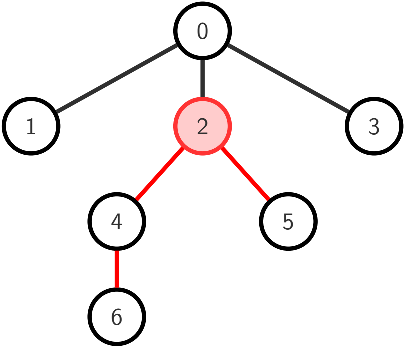
Complexity

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!

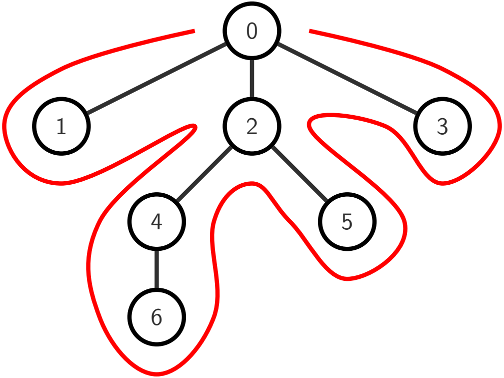
LCA - Eulerian Tour Technique

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.

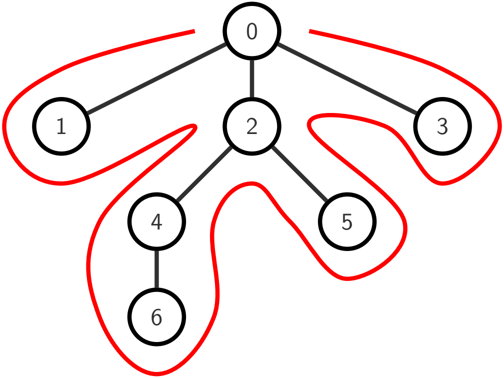
LCA - Eulerian Tour Representation



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

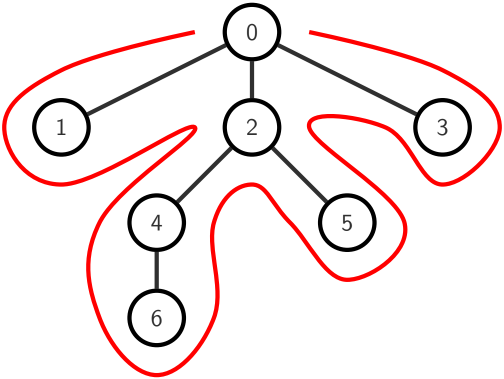
LCA - Eulerian Tour Representation



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

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 $\mathrm{LCA}(1,3) = 0$.

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 $\mathrm{LCA}(u, v)$ requires one minimum query in the segment tree. $\mathcal{O}(\log n)$.