CS 240E S25

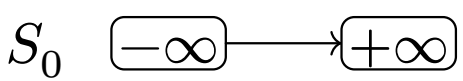
Tutorial 5

June 13

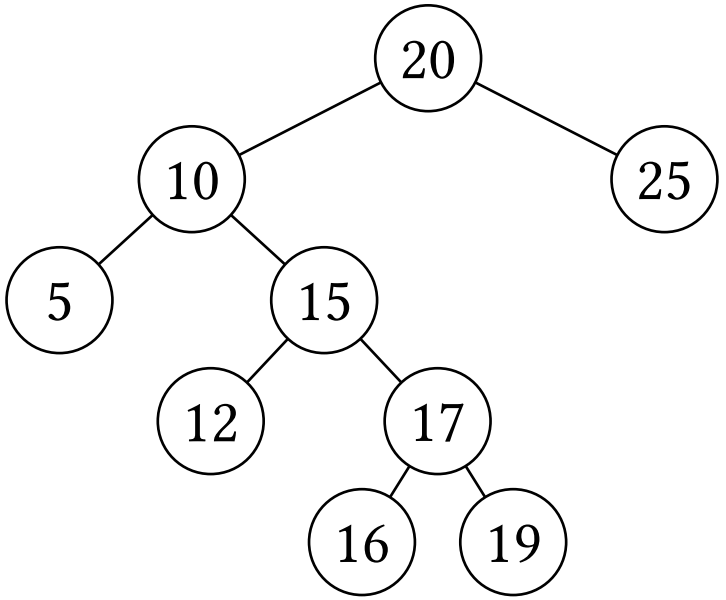
- Skip List
- Splay Trees
- Enrich: Segment Trees
- Optimal Static Ordering
- Counting Trees

Q1: Skip List

Insert the numbers 12, 11, 13, 10, and 20 into an empty skip-list using the sequence of coin flips `HHTHTHTTHHHT` (i.e., every time we go to do a coin flip we take the first item out of this list). **Then delete the keys 13 and 20.** Show the resulting skip-list.



Q2: Splay Trees



Given the following splay tree S and the potential function

$$\Phi(i) := \sum_{v \in S} \log n_v^{(i)}$$

where $n_v^{(i)}$ is the number of nodes in the subtree rooted at v after i operations, including v itself.

- Calculate its potential using the given potential function.
- Insert the key 18. Calculate the new potential.
- Verify that the potential difference is less than $4 \log n - 2R + 2$, where R is the number of rotations.

Q3: Static Ordering

Let A be an unordered array with n distinct items $k_0, \ldots, k_{n-1}$. Give an asymptotically tight (Θ) bound on the expected access cost if you put A in the optimal static order for the following probability distributions:

a) $p_i = \frac{1}{n}$ for $0 \leq i \leq n-1$
b) $p_i = \frac{1}{2^{i+1}}$, for $0 \leq i \leq n-2$. $p_{n-1} = 1 - \sum_{i=0}^{n-2} p_i = \frac{1}{2^{n-1}}$

Q4: Counting Trees

How many binary trees with n nodes are there, as a formula in terms of n? Find a recurrence relation.

(There is also a closed-form for this recurrence relation, but deriving it is outside the scope of this course.)

Enrich Content

Motivation

Question: You have an array of n integers, and you want to perform the following operations:

- Update the value of an element at index i.
- Add a value x to all elements in a index range $[l, r]$.
- Query the sum/min/max of elements in a index range $[l, r]$.

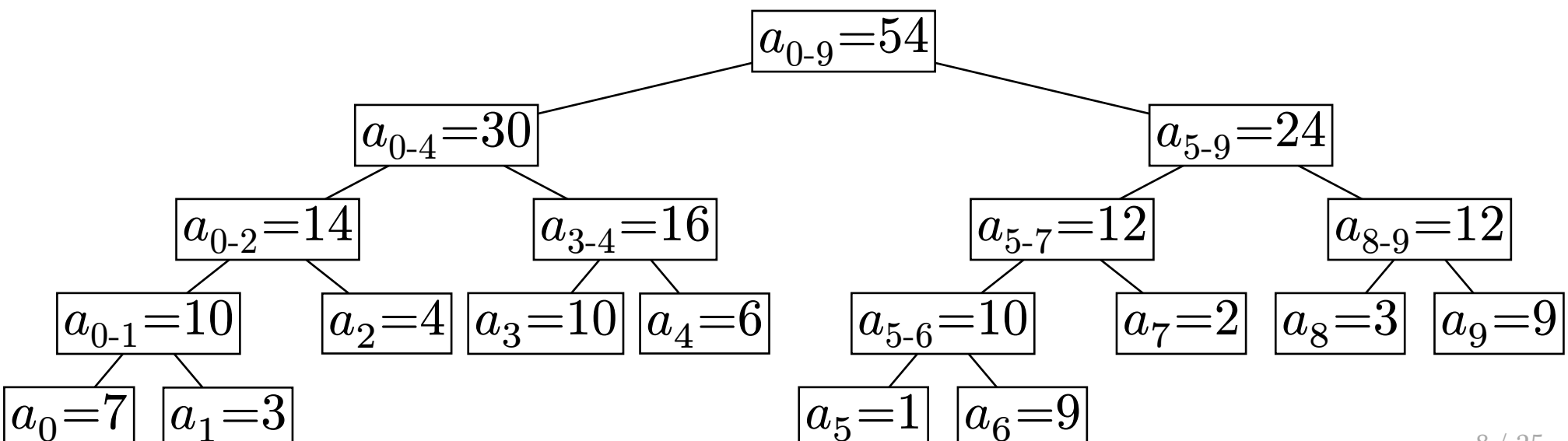
How would you implement this efficiently?

Introducing: Segment Tree!

Segment Tree

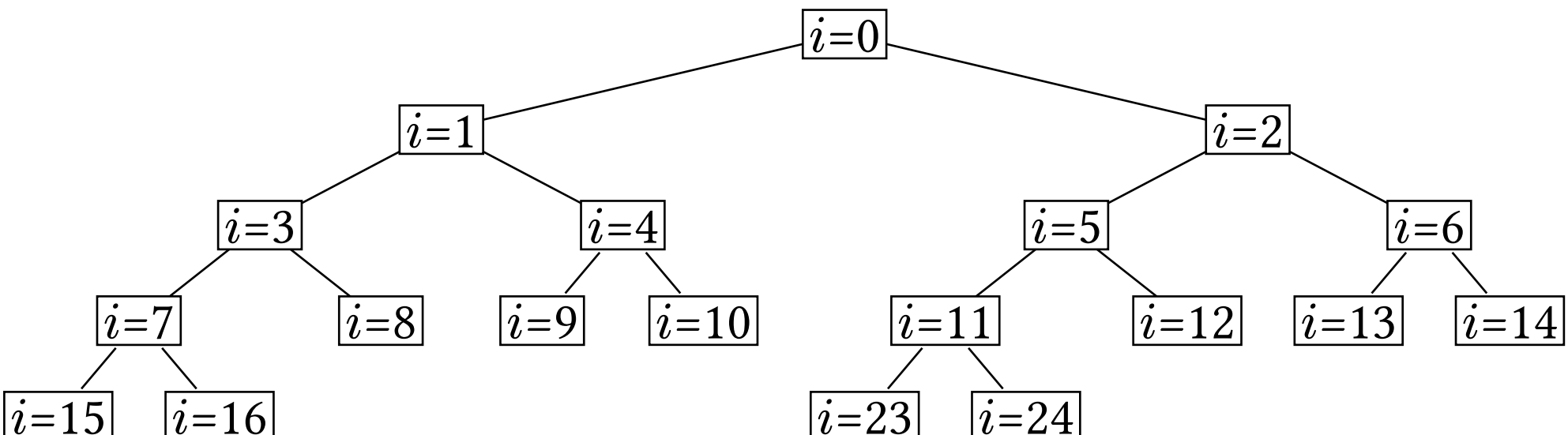
Segment tree is a powerful data structure that efficiently supports both point and range queries/updates on a sequence (an array).

It's a binary tree where each node represents a range of the array.



Store the tree

We can store the tree like a heap: for node with index i, the left child is at index $2i + 1$ and the right child is at index $2i + 2$. The root has index 0. This requires approximately $4n$ space for n elements, therefore, the space complexity is $O(n)$.



Build the tree

Given an initial array $a[n]$, we can build the tree recursively from top to bottom, and update the info of each node based on its children.

```
int n, a[MAXN], t[MAXN * 4]; // t is the segment tree
int build(int i, int l, int r) { // i - index, l/r - range
  if (l == r) t[i] = a[l], return t[i];
  int mid = (l + r) / 2; // divide the range into two halves
  t[i] = build(i * 2 + 1, l, mid) + build(i * 2 + 2, mid + 1,
r); // build chilren first then update current node
  // here we maintain the sum, but it can also be max/min/...
}
build(0, 0, n - 1); // build the tree for our array a
```

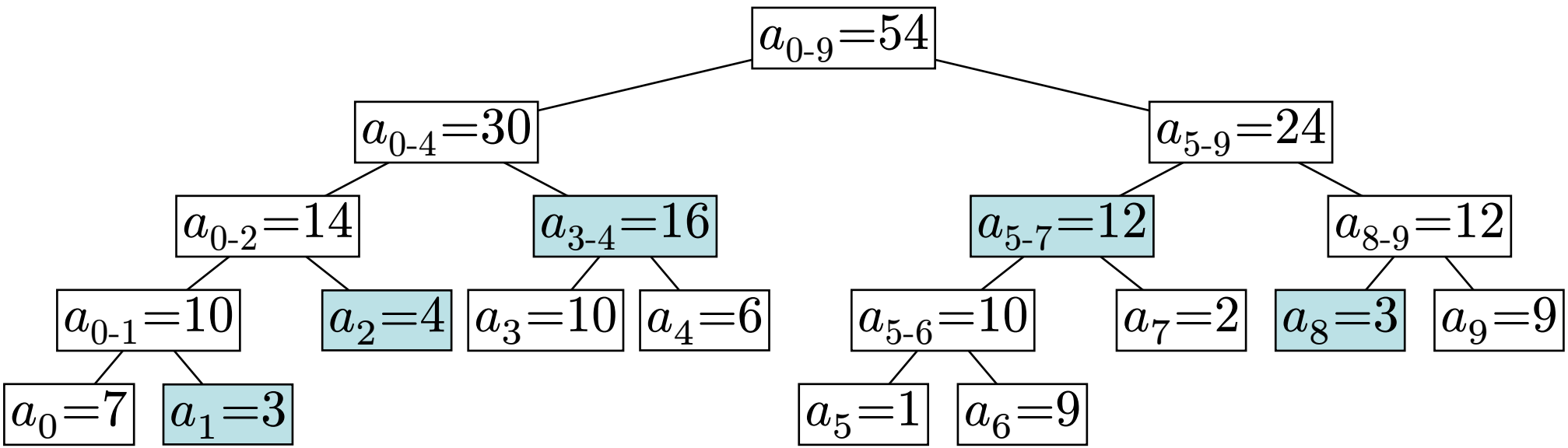
Query a Range

`Query` is very similar to `Build`: to query the sum of a range $[ql, qr]$, we still recursively traverse the tree, but only count the range that is within the query range. It takes $O(\log n)$ time.

```
int query(int i, int l, int r, int ql, int qr) {
  if (ql > r || qr < l) return 0; // no overlap
  if (ql <= l && qr >= r) return t[i]; // is a subrange
  int mid = (l + r) / 2;
  return query(i * 2 + 1, l, mid, ql, qr) + query(i * 2 + 2,
mid + 1, r, ql, qr); // partial overlap, we break it down to
two subranges, query each recursively then sum up
}
cout << query(0, 0, n - 1, 7, 27); // query the range [7, 27]
```

Query a Range: Example

Here is an example of querying the sum of range $[1, 8]$:



Therefore, the sum is $3 + 4 + 16 + 12 + 3 = 38$.

Update: Point Update

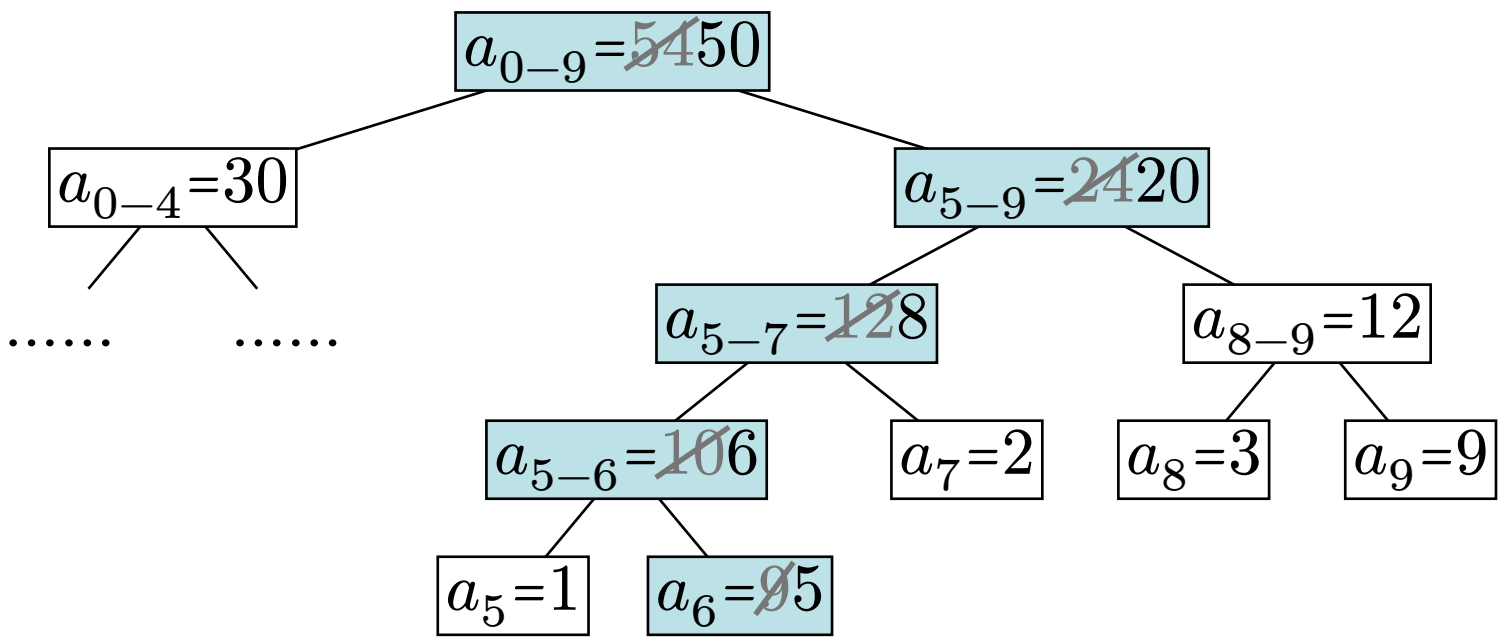
To update a value at index idx, we use similar recursive traversal, but only update the node at idx and its ancestors. It takes $O(\log n)$ time.

```
void update_point(int i, int l, int r, int idx, int val) {
  if (l == r) t[i] = val, return; // found the target, update
  int mid = (l + r) / 2;
  if (idx <= mid){ // target is in the left half
    update_point(i * 2 + 1, l, mid, idx, val);
  } else { // target is in the right half
    update_point(i * 2 + 2, mid + 1, r, idx, val);
  }
  t[i] = t[i * 2 + 1] + t[i * 2 + 2]; // update current node
}
```

Point Update: Example

Here is an example of updating the value at index 6, from 9 to 5:

```
update_point(0, 0, n - 1, 6, 5);
```



Update: Range Update

How to add a value x to all elements in a range $[l, r]$?

Idea: use the similar recursive traversal to update each index in the range. However, this is inefficient, as it takes $O(n)$ time for each update since we need to traverse every leaf in the range.

Better idea: use a **lazy tag** to mark the range that needs to be updated, say, the number to be added to this range, and only update the range when it is accessed later. This prevents unnecessary lower-level updates and perform the range update efficiently.

Segment Tree with Lazy Tags

Similar as the tree array t that stores the sum, we make another array `lazy` to store the lazy tags for each node:

```
int lazy[MAXN * 4];
```

Lazy tags are initialized to 0, meaning no lazy update. When we add a value x to a range $[l, r]$, we simply add x to the lazy tags of nodes that represents this range, which means there is a pending addition x for this range.

Pushdown for Lazy Tags

If a node we want to access has a lazy tag, we need to update the info on this node first, then propagate the tag down. This is called `pushdown`.

```
void pushdown(int i, int l, int r) {
  if (lazy[i] == 0) return; // no lazy tag, do nothing
  t[i] += lazy[i] * (r - l + 1); // update the current sum
  if (l != r) { // not a leaf node, push down to children
    lazy[i * 2 + 1] += lazy[i];
    lazy[i * 2 + 2] += lazy[i];
  }
  lazy[i] = 0; // after pushing down, clear the lazy tag
}
```

Update: Range Update

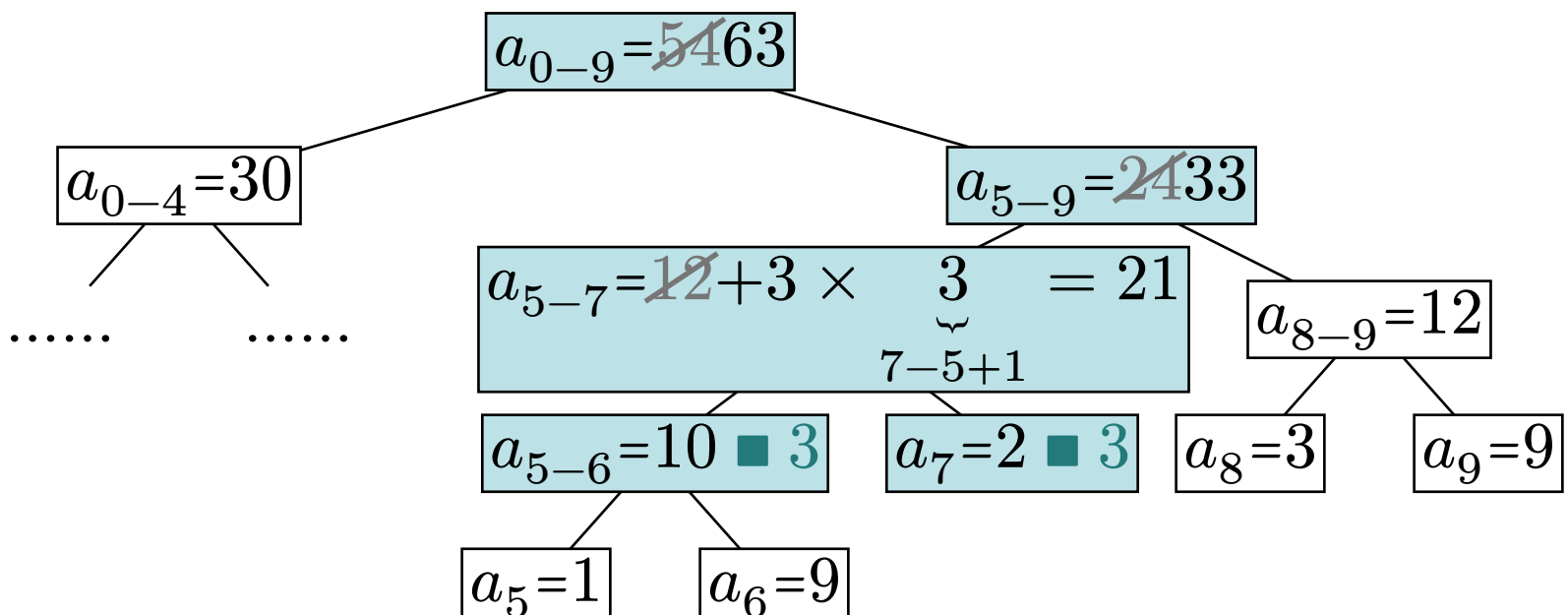
```
void update_range(int i, int l, int r, int tl, int tr, int val) {
  pushdown(i, l, r); // push down any pending updates
  if (tl > r || tr < l) return; // no overlap
  if (tl <= l && tr >= r) { // is a subrange
    lazy[i] += val; // add val to lazy tag
    pushdown(i, l, r); // update current node immediately
    return;
  }
  int mid = (l + r) / 2; // partial overlap, break it down
  update_range(i * 2 + 1, l, mid, tl, tr, val);
  update_range(i * 2 + 2, mid + 1, r, tl, tr, val);
  t[i] = t[i * 2 + 1] + t[i * 2 + 2]; // update current node
}
```

↑ Because when its parent is updating,
it needs to know the current sum of the node.

Range Update: Example

Here is an example of adding 3 to the range $[5, 8]$:

```
update_range(0, 0, n - 1, 2, 6, 3);
```



* ■ x means a lazy tag of x is pending for this node.

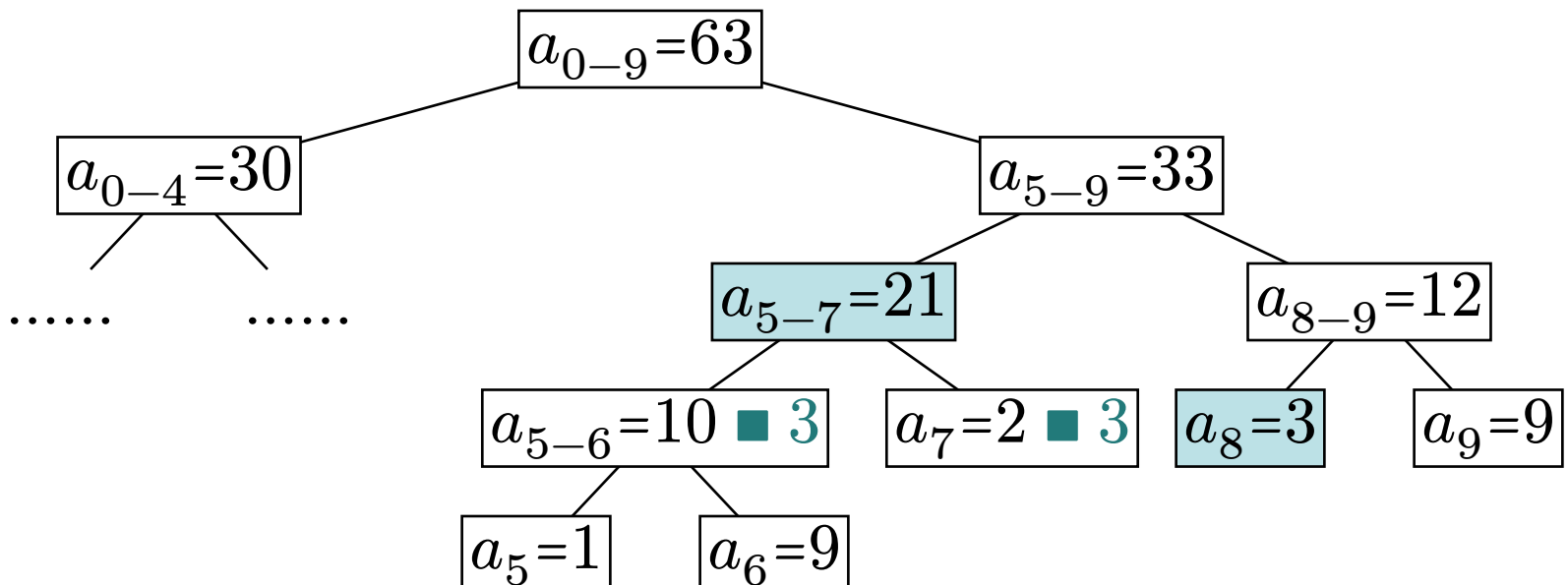
Query Revisited with Lazy Tags

When we do a `query` now, we need to `pushdown` the lazy tags first to ensure the sum at this node is up-to-date. Then we can query as before.

```
int query(int i, int l, int r, int ql, int qr) {
  pushdown(i, l, r)  // The only line we added!
  // other code same as before
  if (ql > r || qr < l) return 0;
  if (ql <= l && qr >= r) return t[i];
  int mid = (l + r) / 2;
  return query(i * 2 + 1, l, mid, ql, qr) + query(i * 2 + 2,
mid + 1, r, ql, qr);
}
```

Query with Lazy Tags: Example 1

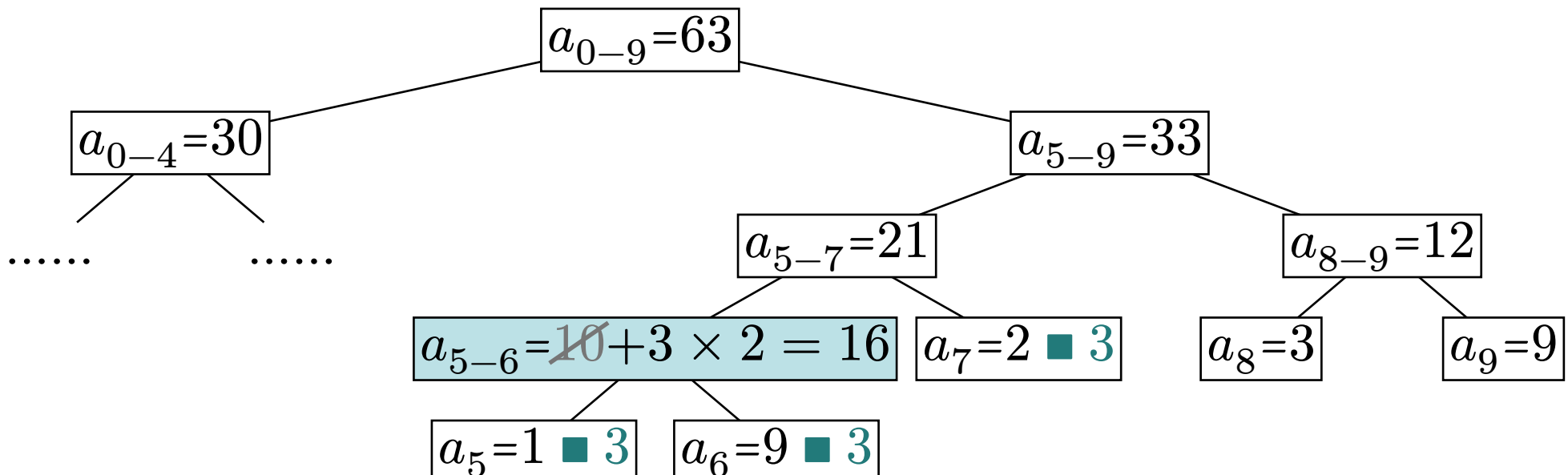
Here is an example of querying the range $[5, 8]$ on the previous tree:



Lazy tags are untouched because we only need their parents to calculate the sum: $21 + 3 = 24$.

Query with Lazy Tags: Example 2

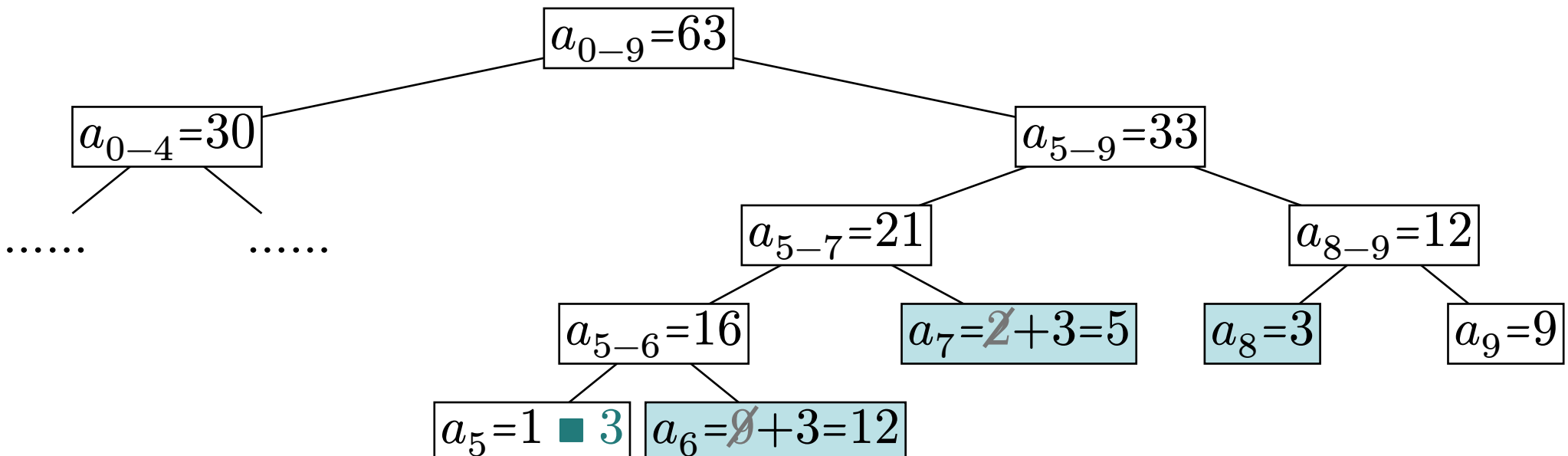
Here is an example of querying the range $[5, 6]$ on the previous tree:



We updated the value for a_{5-6} by performing `pushdown`, because we need the updated sum of a_{5-6}. Lazy tags are propagated down.

Query with Lazy Tags: Example 3

Here is an example of querying the range $[6, 8]$ on the previous tree:



We performed `pushdown` on a_6 and a_7 to update their values, then the lazy tags on them are cleared because they are leaves. $12 + 5 + 3 = 20$.

Segment Tree Summary

Segment tree takes $O(n)$ to build, and each query/update takes $O(\log n)$ time. It takes $O(n)$ space. Therefore, it's very efficient and useful for many problems. It's also the foundation of many other algorithms such as the Heavy-light Decomposition for trees.

Thinking

- How would you maintain the min/max of a range?
- How would you support `range-set` operation, i.e., set all elements in a range $[l, r]$ to a value x?
- If we want to add a new operation, multiply a value x to all elements in a range $[l, r]$, how would you implement it?
- How would you implement a segment tree for a 2D array?

Extension Reading

Segment tree uses approx. $4n$ space, but is it possible to achieve n space?

Fenwick Tree

Fenwick tree is an elegant data structure that supports point updates and range sum queries in $O(\log n)$ time, and uses only n (not $4n$) space. It utilizes the binary representation of indices to achieve this.

With some tricks, it is even possible to support range updates with two Fenwick trees.

It can only maintain invertible properties (sum), but cannot maintain min/max. However, with some tricks, it is possible to maintain min/max as well with $O(\log^2 n)$ time complexity.