Today we are going to explore a class of data structures for performing *range queries* and see how they can be applied to speed up algorithms. Our focus is therefore twofold. First, we would like to design and analyze a specific data structure, that we will refer to as a SegTree, and explore its power. Then, we will see how such data structures can be useful to improve the performance of algorithms. This is a general skill that we would like to emphasize throughout the course – when designing algorithms, how can the use of appropriate data structures be used to obtain the best performance?

***Objectives of this lecture***

In this lecture, we will:

- Introduce the SegTree data structure
- See what kinds of problems SegTrees are capable of solving
- See how SegTrees and related data structures can be used to speed up algorithms

# 1 Range queries

Today we are interested in solving *range queries*. Range queries have a lot of applications, including in databases, computational geometry, geographic information systems, and computer-aided design.

***Definition: Range queries***

Given a sequence of data (often, but not strictly necessarily integers), a *range query* on that sequence asks about a property of some contiguous range of the data $i$ to $j$. For example, given a sequence of $n$ integers, we might want to ask what is the sum of the integers from position $i$ to $j$, or what is the maximum integer among those in positions $i$ to $j$.

For example, suppose we have a sequence of $n$ integers $a[0], a[1], \ldots, a[n-1]$, and we want to support querying for the sum of the integers in positions $i$ to $j$. We will use the convention that the left index is inclusive, and the right index is exclusive. Here are two approaches to get us warmed up.

**Approach #1** For each query, just loop over the integers in positions $i$ to $j-1$ and compute the sum. This takes $\Theta(j-i) = O(n)$ time. This is very simple, but not at all efficient if the sequence is long.

**Approach #2** Start by *precomputing* the prefix sums

$$p[j] = \sum_{0 \le i < j} a[i]$$

then answer a query for the sum between $i$ and $j$ by returning $p[j] - p[i]$. This takes $O(n)$ preprocessing time, which seems reasonable, then each query can be answered in $O(1)$ time! Amazing.

Approach #2 is basically optimal if we never plan to modify the elements of the array, but range queries become so much more useful if we allow modifications. So, our goal is to support an API that enables fast modifications *and* fast queries. Specifically, lets try to design a data structure that maintains an array of $n$ integers and implements:

- **Assign**$(i, x)$: Assign $a[i] \leftarrow x$,
- **RangeSum**$(i, j)$: Return $\sum_{i \le k < j} a[k]$.

How would approaches #1 and #2 above fare now that we want to support modifications?

1. Approach #1 can implement **Assign** in $O(1)$ time by just assigning $x$ to $a[i]$, then **RangeSum**s are still the same as before and take $O(n)$ time.
2. Approach #2 would require us to re-compute the prefix sums $p[j]$ for all $j < i$ whenever we perform **Assign**$(i, x)$, which requires $\Theta(n - i) = O(n)$ time. Queries however still take $O(1)$ time which is nice.

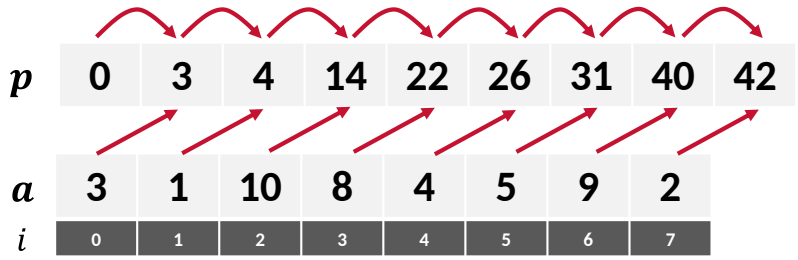
So, in both cases we have one operation that takes $O(1)$ time, and the other which takes $O(n)$ time. If in some particular application, one of the operations is extraordinarily rare, maybe this is a good solution, but if we perform roughly half and half updates and queries, then both solutions are taking $O(n)$ time on average for each operation. This is not great at all. Can we design a data structure that makes both operations fast?

You may have already seen a data structure that can do this. In fact, we've already seen it in this course! Augmented balanced binary search trees (e.g., Splay Trees) can be used to solve this problem in just $O(\log n)$ time per operation and $O(n)$ words of space. However, they are tricky to implement, and often in practice the constant factor is quite high, making them somewhat less practical. Splay Trees also give amortized bounds rather than worst case, though you can improve this by using AVL trees or Red-Black trees.

Today, we are going to design a data structure for this problem with the same bounds, $O(\log n)$ *worst-case* time per operation and $O(n)$ words of space, but that is much simpler to implement, and much faster in practice due to smaller constants hidden by the big-$O$. We will refer to this data structure as a SegTree[1]. We will also discuss how to generalize SegTrees to handle a much wider class of problems, where the $\sum$ operation is replaced by an arbitrary associative operator, enabling us to perform many different kinds of range queries with just one data structure!

## 2 Making range queries dynamic

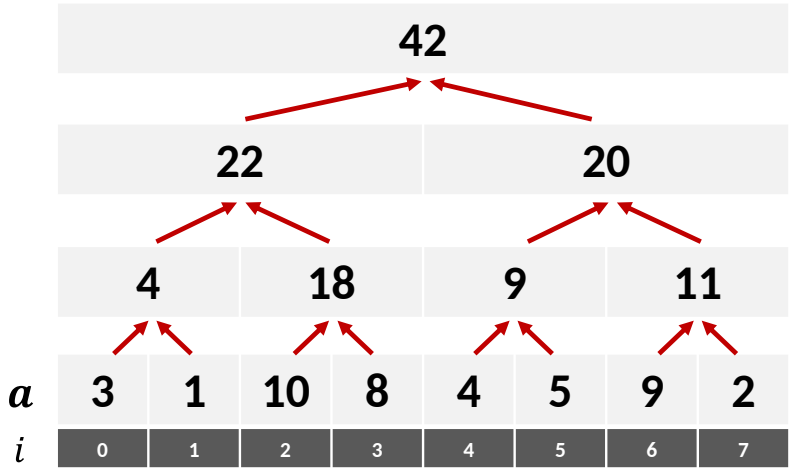
Let's take a step back and have a closer look at Approach #2 from earlier and see if we can find inspiration for a better algorithm. What the algorithm from Approach #2 is really doing is computing the sum from 1 to $n$ in a sequential loop and just saving the results along the way. The inefficiency of performing updates was due to the fact that if we edit element 0, there are $n$ values in $p$ that depend on it, and hence we do $O(n)$ work in updating everything.




The *dependencies* here are what make the update algorithm slow. Is there instead an alternative way to break up the computation such that most of the intermediate calculations depends on fewer of the numbers? You may or may not have seen this idea before when seeking a *parallel algorithm* for computing the sum (or

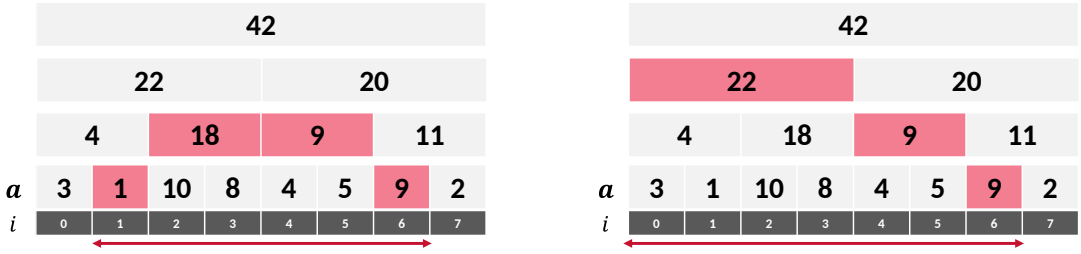
[1] "SegTree" has become the traditional name for this data structure in 15-451, though you might not find it called that in other places. In the competition programming literature they are called "segment trees". However, this name conflicts with another specifically augmented binary search tree that represents a set of line segments in the plane. To remove this ambiguity, Danny Sleator coined the name "SegTree" and it has stuck.

in general, any reduction) of a range of values. The left-to-right sequential sum is completely not parallel because of the dependencies, but a *divide-and-conquer* algorithm avoids this problem.




The divide-and-conquer sum has fewer dependencies now because each element of the input only affects $\log_2 n$ intermediate vales produced by the computation. This means that if we update an element of the input, the output could be updated efficiently[2]. Its not clear yet though how we can actually answer **RangeSum** queries using this information, so lets figure that out now.

**Doing queries** The key idea is in figuring out how to build any interval $[i, j)$ that we might want to query out of some combination of the intervals represented by the divide-and-conquer tree. For example, if we wanted to query the interval $[1, 7) = [1, 6]$, we could add up the intervals $[1, 1], [2, 3], [4, 5], [6, 6]$ as shown below. Similarly, we can query $[0, 7)$ as shown on the right.




We need to somehow prove that we don't need too many intervals to make up any query interval. Because of course, we could always answer any query $[i, j)$ by summing up all of the intervals of size 1 from $i$ to $j$, but that's just the first algorithm which takes $O(n)$ time.

***Lemma 1: Few blocks per level***

Any interval $[i, j)$ can be made up of a set of disjoint intervals/blocks from the tree such that we use at most two intervals from any level.

[2]What we've actually made here is an extremely cool and powerful observation! It turns out that *parallel algorithms* are usually much easier to convert into dynamic algorithms / data structures than sequential ones, because they both share a common feature—both of them rely on having shallow dependence chains. An algorithm where all of the computations are dependent on the previous ones is hard to parallelize, and also hard to dynamize (make updatable) because changing a small amount of the input may change a large amount of the computation.

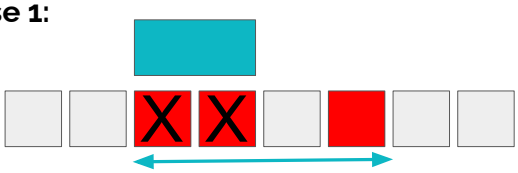
*Proof.* Suppose there is some configuration $C$ for the range $[i, j)$ that involves more than 2 intervals at some level(s). Let $h$ be the deepest level in $C$ that contains $s > 2$ configurations. Our idea is to construct a new configuration $D$ for the same range, such that level $h$ now uses at most $s - 1$ intervals, and any level deeper than $h$ uses at most 2 intervals. If we can succeed in doing so, we can just repeat the process until every level has at most 2 intervals.

More specifically, once we identify level $h$, we can look at the first 3 intervals in level $h$ and reduce them to 2 or fewer intervals. Let $I$ be the middle interval among the three. Starting from the current configuration $C$, we will do the following

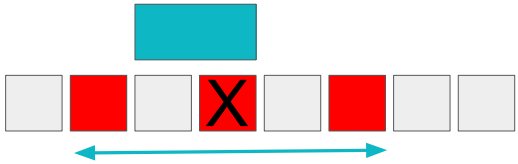
- Add $I$'s parent denoted $\mathsf{parent}(I)$ (if it is not already in $C$),
- Delete $I$ and all of $\mathsf{parent}(I)$'s descendants.

Without loss of generality, we may assume that $I$ is the right child of its parent — since the other case is symmetric/mirroring. Since each parent has two children, only the following cases are possible:

**Case 1:**



**Case 2:**



Case 1 is when the interval $I$ is paired with another interval in the configuration $C$. Case 2 is when the interval $I$ is not paired with another interval in $C$. In Case 1, the number of intervals at level $h$ is reduced by 2. In Case 2, the number of intervals at level $h$ is reduced by 1. No matter which case, in any level deeper than $h$, the number of intervals cannot increase. Also, because the range is contiguous, performing the above operation does not modify the range represented.

□

Since there is a construction with at most two blocks per level, we immediately get the following corollary.

***Corollary:*** $O(\log_2 n)$ ***blocks per query***

Any interval $[i, j)$ can be made up of a set of at most $2 \log_2 n$ disjoint blocks from the tree.

*Proof.* If the root is chosen, then no other nodes would be chosen. Henceforth, we assume that root is not chosen. In this case, the corollary follows from Lemma 1 and the fact that the tree has $\log_2 n$ non-root levels. □

# 3 The data structure

Now that we have the key ingredients, we can put together the data structure. For the moment let's assume that $n$ is a power of two. If it is not, we can always round it up to the next one by at most doubling it, so it won't affect our asymptotic bounds. We will talk about some fairly low level implementation details today, more than we might often do so in this course.

One of the things that make common tree data structures inefficient is that traversing them requires chasing down pointers throughout the tree, each of which points to a node that might reside far away from the former in memory. To make SegTrees more efficient, we will apply the same ordering trick from the *binary heap* data structure that you may have seen before. It works because the SegTree data structure is a so-called *perfect binary tree* (every internal node has two children, and all leaves are on the same depth).

***Definition: The binary heap ordering trick***

Given a perfect binary tree on $N$ nodes, we can lay it out in memory using an array of size $N$ using the following scheme:

- Place the root node at position 0
- Given the node at position $i$, place its left child in position $2i + 1$
- Given the node at position $i$, place its right child in position $2i + 2$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | | | | | | | |
| | 1 | | | | 2 | | | |
| | 3 | | 4 | | 5 | | 6 | |
| *a* | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Notice that the number of nodes $N = 2n - 1$, and that when using this trick, the $n$ input elements of $a$ are all stored in the *last* $n$ elements of the array. We can use these ideas to implement the data structure. We will make use of the following convenience functions:

- Parent$(i) := \lfloor (i-1)/2 \rfloor$. Returns the parent of node $i$
- LeftChild$(i) := 2i + 1$. Returns the left child of $i$
- RightChild$(i) := 2i + 2$. Returns the right child of $i$

**Building the tree** To build the tree, we start with the input $a[0 \ldots (n-1)]$. We then perform the recursive divide-and-conquer to compute all of the block values for the intermediate sums.

***Algorithm: Building a SegTree***

```
n : int
A : array⟨int⟩

SegTree(a : list⟨int⟩) {
  n := size(a)
  A := array⟨int⟩(2*n-1)
  for i in 0 to (n-1) do
    A[n + i - 1] = a[i]
  build(0, 0, n)
}
```

```
build : (u : int, L : int, R : int) -> () = {
  mid := (L + R)/2
  if LeftChild(u) < n-1 then
    build(LeftChild(u), L, mid)
  if RightChild(u) < n-1 then
    build(RightChild(u), mid, R)
  A[u] = A[LeftChild(u)] + A[RightChild(u)] // reduce the value of the children
}
```

**Implementing updates** To implement **Assign**$(i, x)$, we just have to update the element at position $i$ and all of its ancestor blocks in the tree. This takes $O(\log n)$ time and looks like this.

*Algorithm: Updating a SegTree*

```
Assign : (i : int, x : int) -> () = {
  u := i + n - 1
  A[u] = x
  while u > 0 do {
    u = Parent(u)
    A[u] = A[LeftChild(u)] + A[RightChild(u)] // reduce the value of the children
  }
}
```

**Implementing queries** To implement **RangeSum**$(i, j)$, we need to figure out how to identify the set of $O(\log n)$ blocks that make up $[i, j)$. Fortunately, we can do this very naturally with recursion. What we will do is essentially the same as the original divide-and-conquer sum, except we only need to recurse when we are on a block that contains some elements from $[i, j)$ and some elements not from $[i, j)$.

*Algorithm: Querying a SegTree*

```
RangeSum : (i : int, j : int) -> int = {
  return compute_sum(0, i, j, 0, n)
}

compute_sum = (u : int, i : int, j : int, L : int, R : int) -> int {
  if (i <= L && R <= j) then
    return A[u]
  else {
    mid := (L + R)/2
    if i >= mid then
      return compute_sum(RightChild(u), i, j, mid, R)
    else if j <= mid then
      return compute_sum(LeftChild(u), i, j, L, mid)
    else {
      left_sum := compute_sum(LeftChild(u), i, j, L, mid)
      right_sum := compute_sum(RightChild(u), i, j, mid, R)
      return left_sum + right_sum // reduce the value of the children
    }
  }
}
```

# 4 Speeding up algorithms with range queries

Since this is a course on *design and analysis of algorithms* after all, one aspect that we want to focus on is choosing the right data structure for the job when designing an algorithm. We've seen in recent lectures that applying hashing can often drastically reduce the running time of algorithms. How can range queries help us design better algorithms? If we find ourselves designing an algorithm that requires summing over, or taking the minimum or maximum of a set of numbers in a loop, then we may be able to improve it by substituting that code with a range query. Lets see a great example.

***Problem: Inversion count***

Given a permutation $p$ of 0 through $n-1$, the number of *inversions* in the permutation is the number of pairs $i, j$ such that $i < j$ but $p[i] > p[j]$.

For example, the inversion count of the sorted permutation is 0, because everything is in order. The inversion count of the reverse sorted permutation is $\binom{n}{2}$ since every pair is out of order. Lets start by designing an inefficient algorithm. Per the definition, we can just loop over all pairs $i < j$ and check whether $p[i] > p[j]$.

```
inversions : (p : list⟨int⟩) -> int {
  n := size(p)
  result : int = 0
  for j in 0 to n - 1 do {
    for i in 0 to j - 1 do {
      if p[i] > p[j] then
        result = result + 1
    }
  }
  return result
}
```

This will take $O(n^2)$ time, but can we improve it somehow using range queries? Lets try to decompose what the loops of the algorithm are doing. The first one is considering each index $j$ in order, straightforward enough. The second loop is considering all $i < j$ and counting the number of such $i$'s that have been seen previously such that $p[i]$ is larger than $p[j]$. Okay, this is sounding like some kind of range query now because we are counting the number of things in a range... How exactly do we express this using a SegTree?

We need a range of values to correspond to the **count** of the number of elements that we have seen that are greater than a certain value. So, lets just store an indicator variable for each element $x$ that contains a 1 if we have seen that element, or otherwise contains a 0. To count the number of elements that are greater than $p[j]$ will therefore correspond to a range query of **RangeSum(**$p[j]$**,** $n$**)**. The optimized code for inversion counting therefore looks something like this.

***Algorithm: Optimized inversion count using a SegTree***

```
inversions : (p : list⟨int⟩) -> int {
  n := size(p)
  counts : SegTree = (list⟨int⟩(n, 0)) // SegTree initially containing n zeros
  result : int = 0
  for j in 0 to n - 1 do {
    result = result + counts.RangeSum(p[j], n)
    counts.Assign(p[j], 1)
  }
  return result
}
```

Since each **Assign** and each **RangeSum** cost $O(\log n)$, the total cost of this algorithm is just $O(n \log n)$,

which is a great improvement over the earlier $O(n^2)$ one!

# 5 Extensions of SegTrees

## 5.1 Other range queries

We just figured out how to implement SegTrees that support an **Assign** and **RangeSum** API. What makes SegTrees so versatile, though, is that they are not limited to only performing sums of integers. There was nothing particularly special about summing integers, except that it made for a good motivating example. Note that nowhere in our algorithm did we ever need to perform subtraction, which means we never made the assumption that the operation was invertible, which actually makes it even more general than the original "Approach #2" at the beginning! The exact same algorithm that we have just discussed can therefore also be used to implement range queries over **any associative operation**. In the code presented above, there are three lines commented with ***reduce the value of the children***, which add the values computed at the child nodes. Replacing just these three lines with any other associative operation yields a correct data structure for performing range queries over that operator.

For example, we can also compute the maximum or minimum over a range by replacing $X + Y$ in the three commented lines above with $\max(X, Y)$ or $\min(X, Y)$. There's also no reason to restrict ourselves to integers. We can use any value type, such as floating-point values, or tuples of multiple values, as long as we are able to provide the corresponding associative operator.

***Key Idea: SegTrees with any associative operation***

SegTrees can be used with any associative operation, such as sum, min, max, but also even more complicated ones that you will see in recitation!

## 5.2 Other update operations

Our update operation for our vanilla SegTree is **Assign**$(i, x)$, which sets the value of $a[i]$ to $x$. In some applications, instead of directly setting the value, we might want something slightly different. For example, we might want to *add* to the value instead of overwriting it. Fortunately, this can be supported with a combination of **Assign** and **RangeSum**. Note that we can always get the current value of $a[i]$ by performing **RangeSum**$(i, i + 1)$, and then use that in an **Assign**. So to implement a new operation **Add**$(i, x)$, which adds $x$ to $a[i]$, we can just write

- **Add**$(i, x)$: **Assign**$(i, x +$ **RangeSum**$(i, i + 1))$

The **Add** operation calls a constant number of SegTree operations, and hence it also runs in $O(\log n)$ time. This operation will be convenient for the next extension.

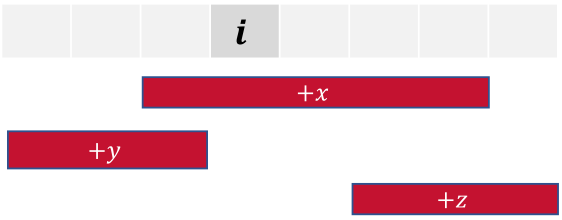
## 5.3 Flipping the operations

Our vanilla SegTree supports *point updates* and *range queries*, that is, we can edit the value of one element of the sequence and then query for properties (e.g., sum, min, max) of a range of values in $O(\log n)$ time. What if we want to do the opposite? Lets imagine that we want to support the following API over a sequence of $n$ integers:

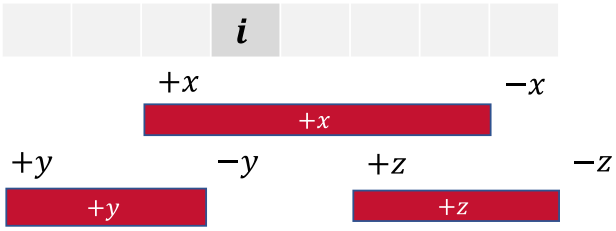
- **RangeAdd**$(i, j, x)$: Add $x$ to all elements $a[i], \ldots, a[j - 1]$
- **GetValue**$(i)$: Return the value of $a[i]$

Rather than come up with a brand new data structure, lets try to use our original SegTree as a black box and reduce this new problem to the old one. This should always be your first choice when designing a new data structure or algorithm (can I reduce to something that I already know how to solve? This is almost always easier than designing something new from scratch!)

**The idea** Somehow we need to convert range additions into just a single update, and range sums into the ability to get a specific value. How might we do that? Well, notice that at a particular location $i$, the value $a[i]$ is equal to the *sum* of all of the **RangeAdd**s that have touched location $i$. That sounds like **RangeSum** should be able to help us then...




Consider the diagram above. The value of **Get**$(i)$ is affected only by $+x$ since the ranges $+y$ and $+z$ do not touch $i$. Notice that more specifically, the value at $i$ is the sum of all of the ranges whose starting point is at most $i$, but whose ending point is at least $i$. We can represent this using *prefix sums*.




Notice that the value of $i$ is just the sum of the $+x$'s that occur at or before $i$, then subtract the $-x$'s that occur before at or before $i$ (since the interval ended before it reached $i$). This is exactly just the prefix sums of all of these $+x$'s and $-x$'s up to position $i$. Therefore, we can use the **RangeSum** method to compute this prefix sum and hence implement **Get**. Our algorithm therefore looks as follows.

***Algorithm: RangeAdd and Get***

We can implement the **RangeAdd** and **Get** API in terms of **Add** and **RangeSum** as follows.

- **RangeAdd**$(i, j, x)$: **Add**$(i, x)$; **Add**$(j, -x)$
- **Get**$(i)$: **return RangeSum**$(0, i + 1)$

Both **RangeAdd** and **Get** call a constant number of SegTree operations, and hence they both run in $O(\log n)$ time as well.

Note that in this algorithm we had to make use of *subtraction*, which means that it isn't applicable to any arbitrary associative operation anymore, since not every associative operator has an inverse. This algorithm is therefore only applicable to invertible associative operations.

## 5.4 Range queries *and* range updates

***Optional content — Will not appear on the homeworks or the exams***

We have now seen how to implement point updates with range queries (i.e., **Assign** and **RangeSum**) and the opposite set of operations, range updates with point queries (i.e., **RangeAdd** and **Get**). Wouldn't it be amazing if we could get the best of both worlds and have *both*? It would be amazing if there existed a data structure that supported the following API:

- **RangeAdd**$(i, j, x)$: Add $x$ to all elements $a[i], \ldots, a[j-1]$
- **RangeSum**$(i, j)$: Return $\sum_{i \le k < j} a[k]$.

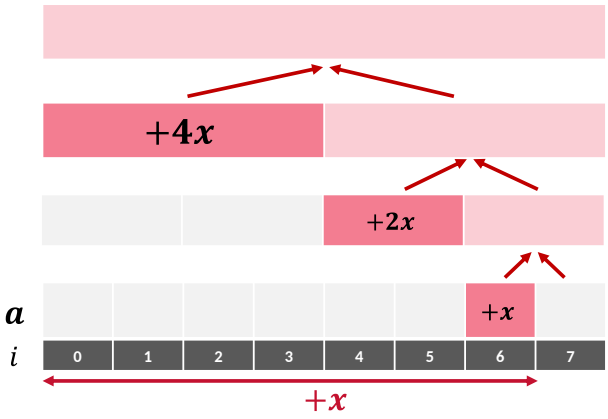
This almost sounds too good to be true, but it turns out that there is a very clever reduction that involves using *two SegTrees* as black boxes that implements both of these operations in $O(\log n)$ time.

***Exercise: RangeAdd and RangeSum***

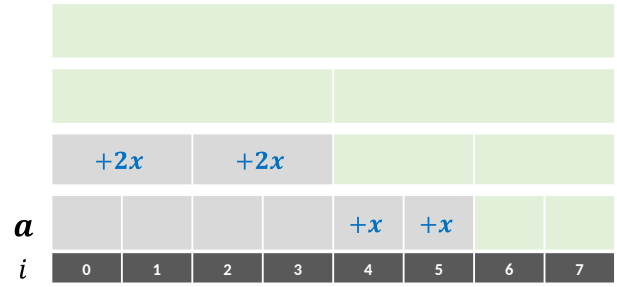
Figure out how to implement a data structure that supports both **RangeAdd** and **RangeSum**.

**Can we do even better?** At this point maybe we are starting to ask for too much, but would it be possible to support range queries for more general associative operations again, while also supporting range update operations too? Surely we are dreaming at this point, but it turns out to still be possible! We won't delve into the full implementation details, but it is super cool to know that this is possible, so lets just sketch a high-level overview of the how it works. The key technique is called **lazy propagation**.

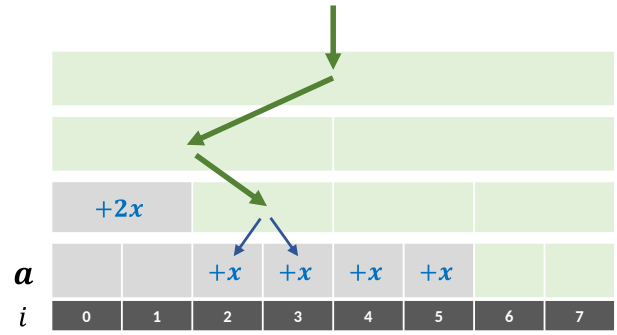
**Lazy propagation** Suppose we want to perform **RangeAdd**$(i, j, x)$. The naive way would be to manually add $x$ to all elements $a[i], \ldots, a[j-1]$ and then reduce the ancestors of these nodes in the tree, but this would take $O(n)$ time. Instead, lets use the same idea that we used for querying and start by identifying a set of at most $2 \log_2 n$ disjoint blocks that make up the interval $[i, j)$. What if we only performed the update operation $+x$ on these blocks? Well, for each block that was affected, if the block contained $s$ elements, then the sum would increase by $s \cdot x$. Then, we could reduce the ancestors of the affected blocks to update their values, too. This would leave us with a partially updated tree of blocks like so...




All of the red blocks have the most up-to-date value, but the descendants of the dark red blocks are all *out of date*, they are missing the update! To fix this, we remember for each child block of the updated blocks, *a "pending" value*, which indicates that an update is still waiting to be applied to that block and its descendants. In the following diagram, green blocks are up-to-date, and gray blocks are out of date. The blue values indicate the pending updates that are waiting to be applied.




So, when do we actually apply the pending updates? Well, if we never traverse to those blocks then we never need to, because we don't care about their values! Only when we actually traverse to that block in the future as part of a subsequent update or query do when then *apply* the update and then *propagate* the update to the block's children. Hence the name lazy propagation. We only propagate the update when we actually need it, instead of when we first perform the operation!




In the diagram above, a future query visits a block with a pending update $+2x$, so it applies it to the value of the block and then *propagates* the update, which creates pending updates of $+x$ on both of the children. That's it! With lazy propagation, you can have range updates and range queries, and you don't need your operation to be invertible since we didn't make use of subtraction this time.

***Exercise: Cost of lazy propagation***

Show that with lazy propagation, updates and queries still take $O(\log n)$ worst-case time.