Handout 2
Interval Trees

Construction and Verification of Software
FCT-NOVA
Bernardo Toninho

19 April, 2022

Contents

This handout is due on Monday, May 9, at 23h59m. The exact details on how to turn in your solution will be made available at a later date.

Read the explanation *very* carefully and don't be scared about the page count. The first 6 pages are mostly pictures to help you understand how interval trees work. The next 2 pages describe in extensive detail how the interval tree operations can be implemented. Section 3 provides some starter code and details the task breakdown. Importantly, Section 4 provides **significant** assistance to the verification of each operation, providing needed lemmas and explaining how to approach the verification problem in the right way. All you need is really written in this document, so read carefully!

As a reference, my implementation of **Task 3** is less than 190 lines of Dafny code, with comments and generous use of linebreaks (e.g. multiple `requires` and `ensures` clauses in pre- and post-conditions and multiple `invariant` clauses in loops, rather than using conjunctions). My implementation of **Task 1** and **Task 2** are similar in length – around 150 lines of Dafny code, following the same style. Don't panic. Don't leave it for the last minute. Ask for help.

1 Interval Trees

In many real world (and imagined) programming scenarios, we are faced with a problem which can be encoded as follows: given a fixed-length sequence of (integer) values, whose contents can *change over time*, we want to compute some ranged queries over the elements of the sequence. For the purposes of this assignment, our ranged query of choice is a ranged sum (i.e., the sum of all elements within a given range in the sequence).

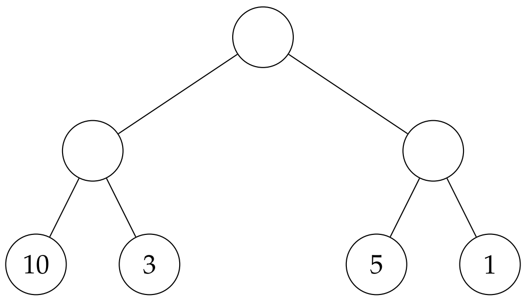
While this pattern may seem very abstract, it actually shows up in quite a few places: Arithmetic coding, for instance, is a form of encoding used in lossless data compression which critically relies on this update/ranged sum pattern. In quantum computing, specifically quantum computational chemistry, encoding fermions with qubits can be achieved using a similar update/ranged sum pattern (this is known as the Bravyi-Kitaev encoding [BK02]). Less exotic applications include computing various running statistics (e.g. running medians) in life sciences. In less real-world scenarios, this pattern is commonly found in competitive programming problems.

Now that we clearly understand that this is a relevant problem to solve, we can observe that the naive solutions are somewhat inadequate. Let us focus (for now) on a specific kind of ranged sum called the *prefix sum* of a sequence. As the name entails, a prefix sum is a ranged sum from the first element of the sequence up to a given element. For instance, given a sequence $\langle 2, 5, 3, 4\rangle$ its successive prefix sums are 2, $2 + 5 = 7$, $2 + 5 + 3 = 10$ and $2 + 5 + 3 + 4 = 14$.

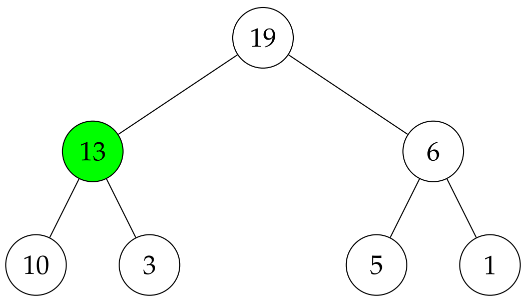
If we maintain the sequence as an array, we can update the values very efficiently ($O(1)$), but computing prefix sums requires $O(n)$ operations, where n is the length of the sequence. On the other hand, if we maintain the prefix sums rather than the sequence itself, we can very efficiently query a given prefix sum (in $O(1)$ time), but updates become $O(n)$ since we may need to update all the sums in the sequence. While this may seem like a reasonable trade off, for large enough n (or where computation is inherently very expensive such as in the quantum case cited above) and in scenarios where *both* the number of updates and prefix sum queries is high, the linear complexity might be too steep.

Luckily, we don't need to adopt the trade off above because there is a simple data structure that allows us to efficiently perform *both* updates and compute prefix sums (in fact, general ranged queries). This data structure is called an Interval Tree.

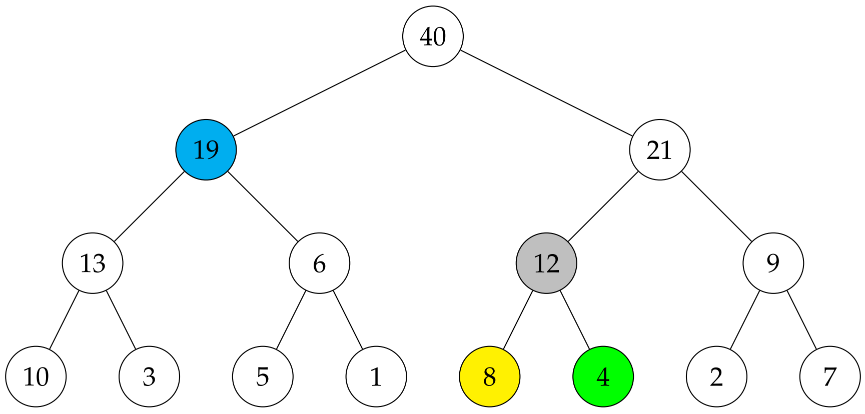
The interval tree encodes *both* the sequence and its prefix sums in such a way that both updates and ranged queries can be performed in $O(\log n)$ time. The concept is easy to visualize – consider the 4 element sequence $\langle 10, 3, 5, 1\rangle$, arranged in the following tree:



We now enforce the property that the value of every (non-leaf) node will be the sum of the value of its direct descendants:

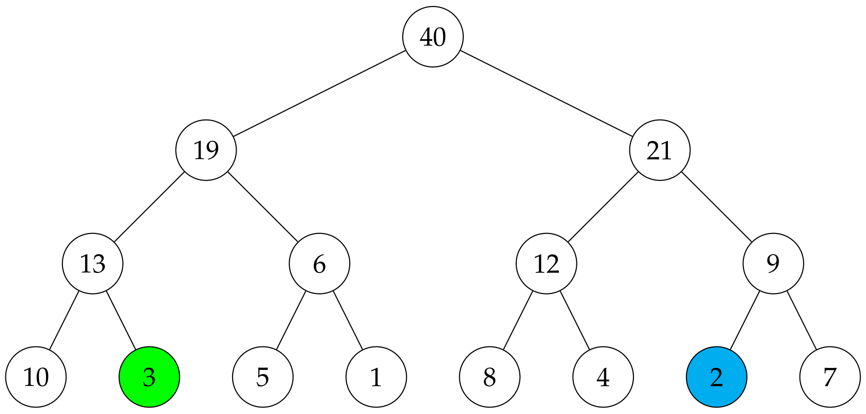


Observe that the prefix sum of the entire sequence is now stored in the root of the tree and we can easily compute all the prefix sums by querying the tree. For instance, to compute the prefix sum up to third element of the sequence, we need only add the value of the green node to the leaf containing 5. While it may not seem obvious that we are doing less work this way, it becomes clearer with larger trees:

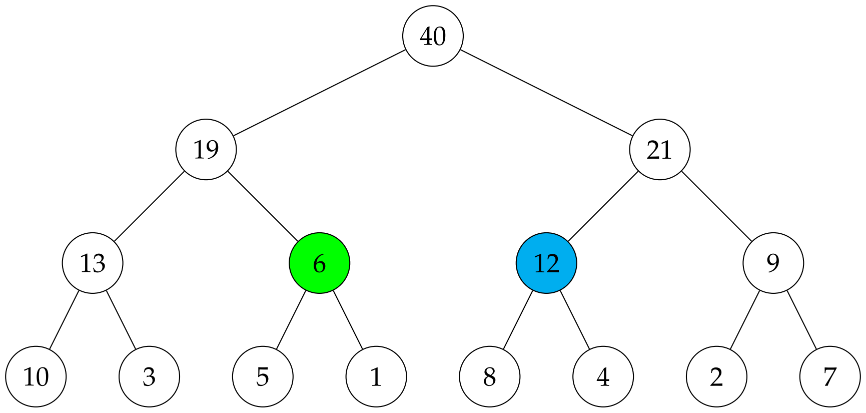


The tree above corresponds to the interval tree for the sequence $\langle 10, 3, 5, 1, 8, 4, 2, 7 \rangle$, with the same property as before, where each node contains the sum of its direct descendants. Now, if we want to calculate the prefix sum up to the fifth element (8) of the sequence, we need only add the blue and yellow colored nodes above! For a more general case, if we want the prefix sum up to the sixth element of the sequence we add the gray and blue nodes, and so on.

Ranged Sums. Another interesting aspect is that the interval tree allows us to easily compute not just prefix sums but *any* ranged sum over the sequence. Lets now analyze a fully general case, focusing on how the algorithm works:

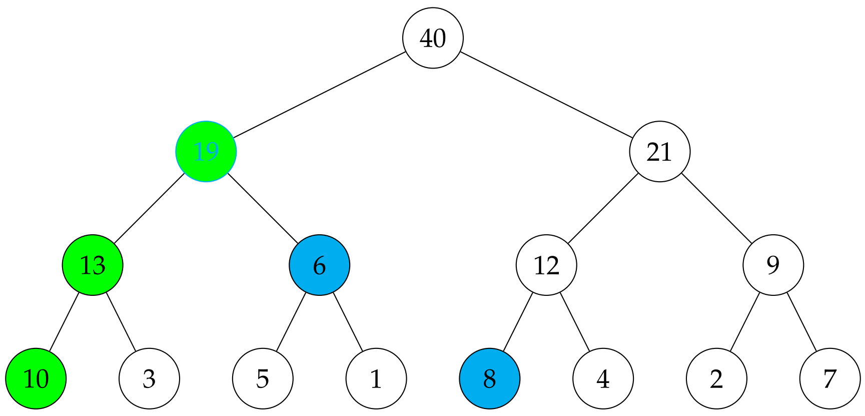


We want to compute the ranged sum of the interval $[1, 7[$ in the 0-indexed sequence consisting of $\langle 10, 3, 5, 1, 8, 4, 2, 7\rangle$. That means we want to sum the elements from 3 to 2, corresponding to the green and blue colored nodes above, respectively. Since the green node is a *right* child of its parent, we cannot use the value of its parent in the sum since it includes values outside of our desired range. Similarly, when thinking about the upper bound of the interval, since the blue node is a *left* child of its parent, we also cannot include the value of the parent in the sum since it includes values that are outside of the interval limit. In this case, we need to add the values 3 and 2 to our result and readjust our interval as follows:



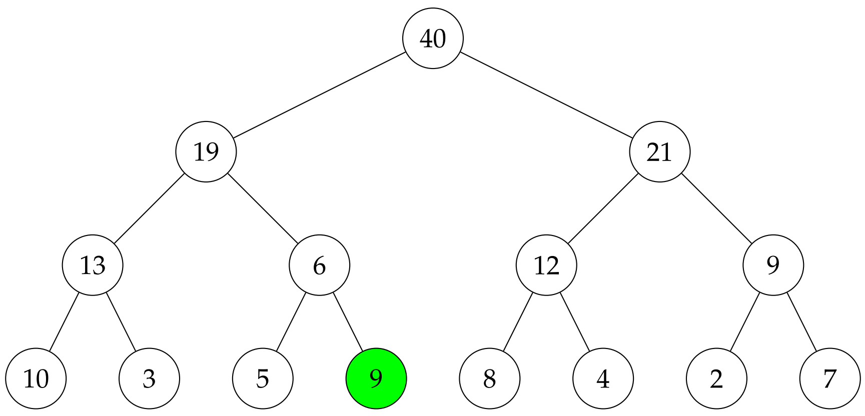
Because we included 3 and 2 in our result, we can save the most work by readjusting our lower and upper bounds to the green and blue nodes, respectively, and we repeat our analysis. Since the green node is a *right* child of its parent, we add it directly to our running total. The blue node is a *left* child of its parent, which for the upper limit means we must also add the value to our running total. If we readjust our range following the same principle, we should move the lower bound to the parent of its right sibling and the upper bound to the parent of its left sibling. Since this "crosses" the ranges we should stop – and rightly so, because we have calculated $3 + 6 + 12 + 2$ which corresponds exactly to the ranged sum we wanted.

The description above matches the algorithm that allows us to compute any ranged sum over an interval tree. We start from the leaves of the tree and with each step go up one level of the tree. Depending on whether the nodes under consideration are left or right children of their parents, we add them to the running total or skip them. Since in the $[1, 7[$ example we were always dealing with right-child lower bounds and left-child upper bounds, let us consider a mixed case. For instance, the sum of the interval $[0, 5[$ from before. Below we have colored at each level the lower and upper bounds of the intervals:

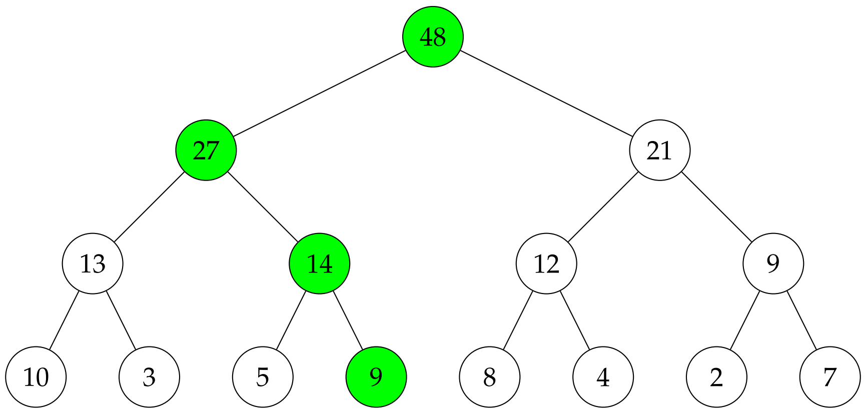


Starting from the leaves, the lower bound is a left-child of its parent, so we *don't* include it in our running total and adjust the lower bound directly to its parent node – the idea is that value will eventually be counted by one of the parent nodes, depending on the upper bound. The upper bound is a left-child of its parent, so we must include it and readjust to the parent of its left sibling. Now we are also in a different situation from before: the lower bound is a left child of its parent, so we also don't include it in our running total and move to its parent node. The upper bound is a *right* child of its parent so by similar reasoning we *don't* include it in our running total and move to the parent. We are now in a situation that is identical to our starting configuration (left-child lower and upper bounds): we skip the lower bound and add the upper bound to our running total. After this step our bounds have crossed and we stop. If we take stock, we notice that we computed $19 + 8$, which is exactly what we needed.

Updates. Now that we have a general understanding of how to perform ranged sums, lets think of how to update a value of the sequence while preserving the *invariant* that the value of each node is the sum of its children. Lets try to update the 4th element of the sequence from 1 to 9. First we update the corresponding leaf:



Now, to reestablish the invariant, we need to update *all* the nodes going up the tree to the root, starting from the green leaf node:



Note that we need not consider any other values in the tree. We simply track the change to the leaf node and perform the *same* update to the highlighted nodes. Since the update touches exactly one node per level in the tree, we perform an update in $\log(n)$ steps.

2 Implementing Interval Trees

By now you are probably worried that you will have to implement and verify a complicated pointer-based data structure. While you will have to implement interval trees in this assignment, we don't implement these trees using linked-list style cells but rather encode the flattened tree in an array, such that we can easily do the following operations:

(1) Find the array index that corresponds to an element of the sequence.

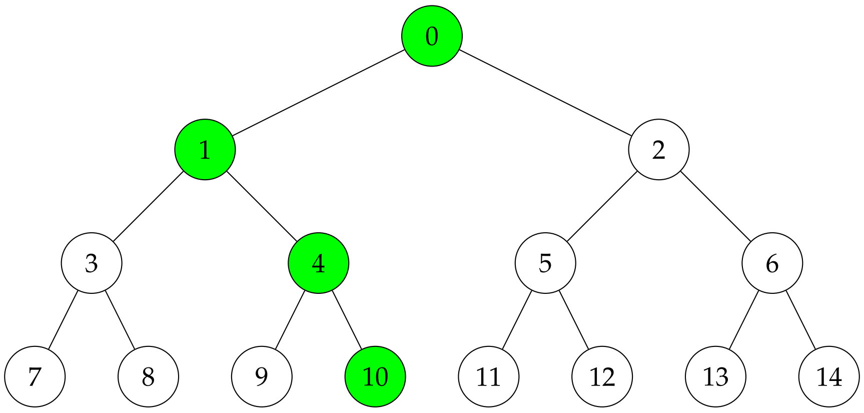
(2) Given an array index corresponding to a tree node, find the array indices of its children.

(3) Given an array index corresponding to a tree node, find the array index of its parent.

Here's how we represent the tree for a sequence of length n using an array. The array will contain the values of the tree nodes. The representation is such that the left child of the node in array index i will be stored in index $2i + 1$ and the right child in index $2i + 2$. The root will be stored in index 0, its left child in index 1, its right child in index 2; the children of index 1 in indices 3 and 4, and so on. This will hold for all indices that correspond to non-leaf nodes (and only for those). The leaves of the tree (i.e., the elements of the sequence) are stored in the last n indices of the array. This means that to represent the interval tree for a sequence of length n we need an array of length $2n - 1$.

Now, lets see how to perform the three operations above. First, **(1)** to obtain the i-th element of the sequence we need to go to index $i + (n - 1)$ of the array. This is because the first $n - 1$ elements of the array correspond to the non-leaf nodes of the interval tree. Finding the indices of children nodes is straightforward: **(2)** the children of the node in index i are stored in indices $2i+1$ and $2i+2$. Moreover, we know that this holds for all indices in the array from 0 (inclusive) to $n - 2$ (inclusive) or $n - 1$ (non-inclusive). Our representation will also have to enforce that if the tree is stored in an array called `tree`, it must be the case that the value `tree[i]` must be equal to the sum of values `tree[2*i+1]` and `tree[2*i+2]`.

Finally, **(3)** given an array index i that corresponds to any (non-root) tree node, we can find its parent in index $(i - 1)/2$ (integer division). Since this may not be obvious, lets revisit the tree from our running example, but where the numbers now denote the corresponding array indices of the nodes. The path up from the leaf to the root that we used to perform the update is highlighted in green:



Note that the parent of any non-root node i is in position $(i-1)/2$ (e.g. $(10-1)/2 = 4$, $(4-1)/2 = 1$, $(1-1)/2 = 0$). Moreover, this works for nodes in both left and right child positions (e.g., $(12-1)/2 = (11-1)/2 = 5$). Another very useful property that will be useful later on is to note that left-children fall on odd indices and right-children on even indices.

2.1 Basic Operations

Lets now see how the ideas for the update and range query operations map onto our array representation and the three index-based operations above. Below we will use s to refer to the sequence of elements for which we are constructing the interval tree and s_i to refer to the element in position i of the sequence. For convenience we will assume the sequence is 0 indexed (i.e. its elements are $s_0, s_1, \ldots, s_{|s|-1}$, where $|s|$ is the length of the sequence). We will use `tree` to refer to the array-based implementation of the tree and use `tree[i]` to refer to the element in position i, as expected.

2.1.1 Update

The update operation is straightforward enough. Without loss of generality we will consider the update operation as an increment operation by a certain value. Given index j of the *sequence* element we want to update, and the value v by which we want to increment s_j, we implement the update on the array representation of the tree by first calculating the index in the array that corresponds to element s_j. By **(1)** above, we know this must be position $j + |s| - 1$ of the *array*. The attentive reader may have noticed that since the length of `tree` is $2 \times |s| - 1$, then $|s| - 1$ is exactly half (integer division) of the length of the array.

We now update this position of the array, incrementing its value by v, which temporarily breaks our tree invariant that all array elements i, with $0 \leq i < |s| - 1$ must satisfy the equality `tree[i] = tree[2*i+1] + tree[2*i+2]`.

To restore the invariant, we increment the nodes in the path from the leaf in index $j + |s| - 1$ up to the root by amount v. Let m be $j + |s| - 1$. By **(3)**, we traverse the appropriate elements of the array by visiting position $m := (m-1)/2$, updating it by v and then repeating this procedure until we have updated all nodes up to the root (recall that the root is in index 0 and must also be updated).

2.1.2 Ranged Query

The ranged query is slightly more subtle. First, make sure you understand the mechanics of how the ranged query works on the tree itself. You should be able to observe that the procedure that was described above leans heavily on the fact that the sum of *all* elements in a given level is exactly the same as the sum of *all* elements in the previous level. More generally, the sum of all nodes in the (array-indexed) range $[a, b[$ (note the open interval for b) is equal to the sum of all nodes in range $[2a + 1, 2b + 1[$. Convince yourself that this is the case and try to reimagine the explanation for the ranged query given above, but where the upper bound is non-inclusive – its the same calculation, except the "real" upper bound is off by one.

Lemma 2.1. *Given a valid array-based implementation of an interval tree encoding sequence s, we have that for all i and j such that $0 \leq i \leq j < |s| - 1$, the sum of values in array indices $[i, j[$ equals the sum of values in array indices $[2i + 1, 2j + 1[$.*

Now, lets try to see how to perform the ranged sum of the interval $[a, b[$ in the sequence s, again noting that the interval is open in the upper range. As you will see, the open interval makes the implementation more uniform (we assume below that $a \leq b \leq |s|$):

- First, since we start from the leaves of the tree, we calculate the array indices that correspond to elements a and b in the sequence s (i.e., $a + |s| - 1$ and $b + |s| - 1$, respectively), recalling that $|s| - 1$ is half (using integer division) of the length of `tree`. Let these two values be `ra` and `rb`.
- We now repeatedly inspect the nodes denoted by `ra` and `rb`, leveraging Lemma 2.1 above.
  - If `ra` is a right child, then its value *must* be included in the running sum. If you look at the tree that is labelled with the corresponding array indices, you will see that the nodes in right-child position are exactly the *even* numbered indices. Therefore, we test if `ra` is even, and if so, we add the array value at that position to our running total. Otherwise we can skip this value. Another interesting aspect is that the way we update `ra` is actually uniform: regardless of whether its a left or right child, we are always moving up the tree to position `ra/2` (check that this encodes the correct behavior!).
  - As for `rb`, if the value is even, this means that it is a right-child. Since the interval is open on its upper-bound, this actually means that we are in a situation where we *must* include the value at position `rb-1` in our sum because its node must be a *left-child*. Otherwise we are in a situation where we can skip this value. Now we need to understand how we update `rb` – by using the open interval we can be completely uniform since we *always* move up to the parent of `rb`, which means we always go to `(rb-1)/2`.
- When do we stop? When the interval is empty (i.e., `ra = rb`)!

You should now have enough information to implement an update and ranged query operations over an array-based representation of an interval tree.

3 Tasks

Your job is to produce a verified implementation of an interval tree in Dafny, using the array-based representation described above. The interval tree ADT should have the following operations (the pre- and post-conditions are incomplete):

```
class IntervalTree {

  //The actual tree
  var tree: array<int>

  /*The number of leaves in the tree (i.e. the number of elements in
   the sequence). */
  ghost var leaves: int

  /*Initializes an interval tree for a sequence of n elements whose
  values are 0. */
  constructor(n:int)
  requires n > 0
  ensures leaves = n

  //Updates the i-th sequence element (0-based) by v
  method update(i:int,v:int)
  requires 0 ≤ i < leaves

  //Ranged sum over interval [a,b[
  method query(a:int,b:int) returns (r:int)
  requires 0 ≤ a ≤ b ≤ leaves
  ensures r = rsum(a,b)

}
```

Where `rsum` stands for the following (the specification of these functions is complete, except for the ... in the **reads** clauses):

```
  //Sum of elements over range [a,b[
  function rsum(a:int,b:int) : int
  requires Valid()
  decreases b-a
  requires 0 ≤ a ≤ leaves ∧ 0 ≤ b ≤ leaves
  reads ...
  {
      if b ≤ a then 0 else get(b-1)+rsum(a,b-1)
  }

  predicate ValidSize()
  reads ...
  {
      tree.Length = 2*leaves-1
  }

  /*ith element of the sequence, through the array-based
  representation*/
  function get(i:int) : int
  requires 0 ≤ i < leaves ∧ ValidSize()
  reads ...
```

```
    {
        tree[i+leaves-1]
    }
```

Note that the `get` function above is actually only needed to assist Dafny's automation with quantifiers.

3.1 Task 1

Implement the interval tree according to the skeleton given above, completing it with the appropriate invariants and post-conditions. For this iteration, **do not** add any extra ghost fields to the code. This means that you **do not need** to use the dynamic frames pattern discussed in class, nor should your abstract state be **explicitly encoded** using ghost state. Be careful in ensuring that your specification does not reveal concrete state in post-conditions, although it will necessarily do so in **`reads`** and **`modifies`** clauses.

3.2 Task 2

Having completed **Task 1** successfully, adjust your specification to use the dynamic frames pattern discussed in class to preserve information hiding. Note that this will require adjusting loop invariants with extra seemingly "obvious" information about pertaining to pointers.
Caveat Emptor: If you are very confident, you may skip **Task 1** and develop your verified implementation only using dynamic frames (without losing any points). I strongly advise against this decision.

3.3 Task 3

Add a ghost field to your implementation that will represent the sequence that is backed by the interval tree:

```
ghost var s : seq<int>
```

The goal here is for the invariant of the ADT to enforce the following extra properties:

```
  predicate Valid()
  reads ...
  {
    // ...
    |s| = leaves
    ∧
    tree[0] = sum(s)
    ∧
    (∀ i • 0 ≤ i < leaves ⟹ s[i] = get(i))
  }

  function sum(s:seq<int>) : int
  {
    if |s| = 0 then 0 else s[0]+sum(s[1..])
  }
```

Doing so will inevitably require adjusting the loop invariant of the update operation.

4 Important Verification Hints

To be able to complete **Task 1** and **Task 2**, you will need to define a predicate that adequately captures the array-based representation of the tree in terms of the relationship between the values of non-leaf nodes and its children (`Valid()` in the sample code above). This invariant must be preserved by all tree operations.

The tree operations will likely ensure other properties, beyond just this invariant. The constructor should enforce that the the leaves of the tree have value zero. The update operation should enforce that only the correct leaf was updated. The query operation should enforce its result is correct (as given in the specification above).

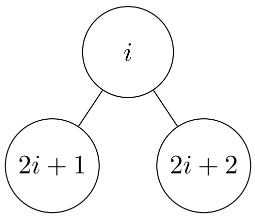
Constructor. The constructor should enforce that the leaves of the tree have the value zero. The constructor should be able to be verified directly by Dafny's automation. Recall the array initialization construct using a closure that we discussed in class. Moreover, note that the error message:

```
in the first division of the constructor body (before 'new;'),
'this' can only be used to assign to its fields.
```

is due to constructors in Dafny having two "regions": one where we only assign values to object fields and another where we can mutate fields. You can indicate the start of the second region with a line containing the command `new;`.

Update. The update operation should enforce that only the correct leaf was updated. For **Task 1** and **Task 2**, no lemmas should be necessary to verify the update operation. Think carefully of what the loop invariant should be. Specifically, the loop invariant should characterize the region of the array corresponding to non-leaf nodes that is still unchanged by the loop (e.g. using the `old` predicate); maintain the fact that only the appropriate leaf of the tree was modified; and, that an adjusted "sum of children" property holds, where one of the children may have been updated by v without the update having yet been reflected in its parent.

For this last point, its important to think of which indices this may happen to during the loop. Critically, this part of the loop invariant should be such that when the loop terminates it entails the overall "sum of children" property of the global ADT invariant. It is helpful if you think of an arbitrary part of the tree (where the labels are the node indices):



For any given part of the tree, we want the value at node i to equal the sum of the values at $2i+1$ and $2i+2$. **If** your "cursor" is either of the two children at the start or end of the loop body, then there is an "extra" v in the sum of the children that is not yet captured in the parent i. **Otherwise**, in all other parts of the tree the "sum of children" property should hold outright. Either we have already adjusted the tree or its a part of the tree that is many levels up and the property is still correct, although globally out of date. Recall that you can use **`if`** ...**`then`** ...**`else`** ... in specifications.

Ranged Query. The ranged query invariant will turn out to be surprisingly succinct, relating the result `r` with the ranged sum `rsum(a,b)` and the sum of the "current interval" (between `ra` and `rb` in the explanation above). However, a and b stand for indices in the *sequence* whereas `ra` and `rb` stand for indices in the *array*. This means we cannot use `rsum(ra,rb)` in our invariant. I strongly recommend you use the following auxiliary definition:

```
//sum of array elements in [a,b[
function sumArr(a:int,b:int) : int
requires Valid()
requires 0 ≤ a ≤ tree.Length ∧ 0 ≤ b ≤ tree.Length
reads ...
{
if b ≤ a then 0 else tree[b-1]+sumArr(a,b-1)
}
```

To actually verify the ranged query, Dafny will require some assistance. First, Dafny needs a way to relate the outcome of `rsum` and `sumArr`:

```
lemma shift(a:int,b:int,c:int)
requires Valid() ∧ 0 ≤ c ≤ leaves-1
requires 0 ≤ a ≤ leaves ∧ 0 ≤ b ≤ leaves
requires ∀ i • a ≤ i < b ⟹ get(i) = tree[i+c]
ensures rsum(a,b) = sumArr(a+c,b+c)
decreases b-a
{ ... }
```

The lemma above looks complicated, but all its saying is that `rsum` sums the same elements as `sumArr`, shifted by some amount `c`, provided the result of `get` behaves accordingly. This lemma will be necessary to "get the ball rolling", so to speak, when the result is still 0.

For the correctness of the procedure throughout the tree, we will need to encode the analogue of Lemma 2.1 in Dafny and will require an "obvious" property of `sumArr`:

```
lemma crucial(ra:int,rb:int)
requires 0 ≤ ra ≤ rb ∧ 2*rb < tree.Length ∧ Valid()
ensures sumArr(ra,rb) = sumArr(2*ra+1,2*rb+1)

lemma sumArrSwap(ra:int,rb:int)
requires Valid()
requires 0 ≤ ra < rb ∧ 0 ≤ rb ≤ tree.Length
ensures sumArr(ra,rb) = tree[ra]+sumArr(ra+1,rb)
```

You will need *both* of these to reestablish the loop invariant. Think *very* carefully of how to instantiate the lemmas. My recommendation is to start from the loop invariant (which we assume holds at the start of the loop) and then incrementally update it with each operation performed in the loop body. At the end of the loop you should end up with several possibilities, depending on the parity of `ra` and `rb`. With the goal of reestablishing the loop invariant in mind, see how `crucial` could help simplify this expression and how `sumArrSwap` relates to your running values.

Task 2. For this task, you should not need to change your verification efforts in a significant way. Beyond the dynamic frames patterns in **reads** and **modifies** clauses (and the corresponding adjustments to the ADT invariant), you should only need to adjust the specification of the update method. Specifically, you will have to include various “obvious” (emphasis on the obvious aspect) properties as part of your loop invariant that ensure dynamically allocated memory has not suffered any “dramatic” changes. The post-condition will also require a very minor adjustment to guarantee that the appropriate auxiliary functions may be used.

Task 3. For this task, you will very likely need the following lemmas:

```
lemma sum_zero(s:seq<int>)
  requires ∀ i • 0 ≤ i < |s| ⟹ s[i] = 0
  ensures sum(s) = 0

lemma sum_elem(s:seq<int>,x:int,p:int)
  requires 0 ≤ p < |s|
  ensures x+sum(s) = sum(s[p := s[p]+x])
```

The challenge will be in enriching the loop invariant of the update operation. Try to express the relationship between the value of `sum(s)` before and after the update and the relationship between the value “currently” being touched by your loop cursor and the one before the update. This task will also require a few “obvious” properties to be stated as part of the invariant to satisfy pre-conditions of auxiliary functions.

5 Grading

You are expected to use the techniques covered in lecture to ensure the ADT interface does not break the abstraction. The handout is divided in three tasks which will be graded independently. As a baseline, if you are able to complete **Task 1** successfully (meaning with an adequately abstract specification that captures the properties mentioned in this document), with some lemmas assumed but not proved, you should expect a passing grade (but no more).

References

[BK02] Sergey B. Bravyi and Alexei Yu. Kitaev. Fermionic quantum computation. *Annals of Physics*, 298(1):210–226, 2002.