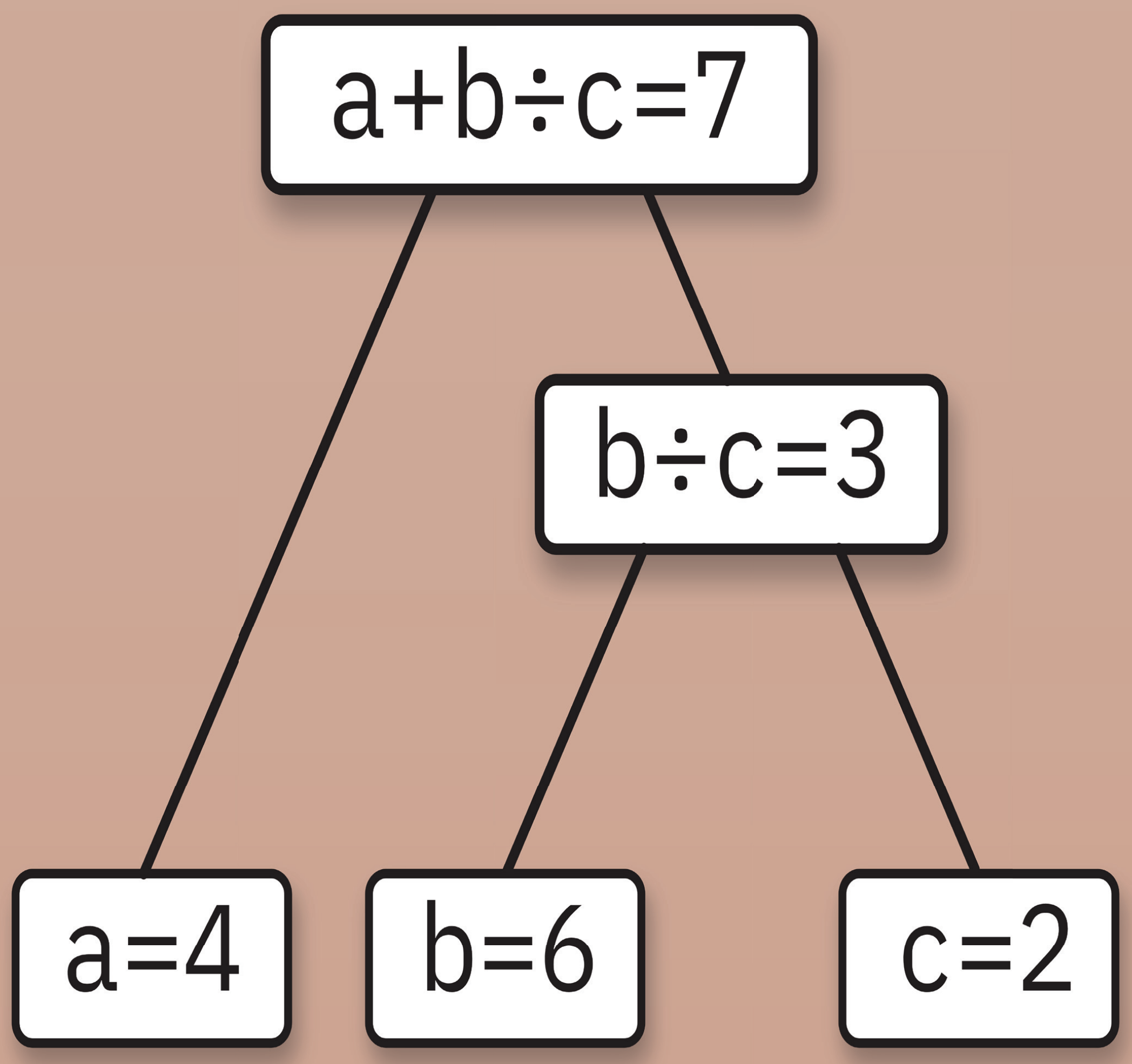
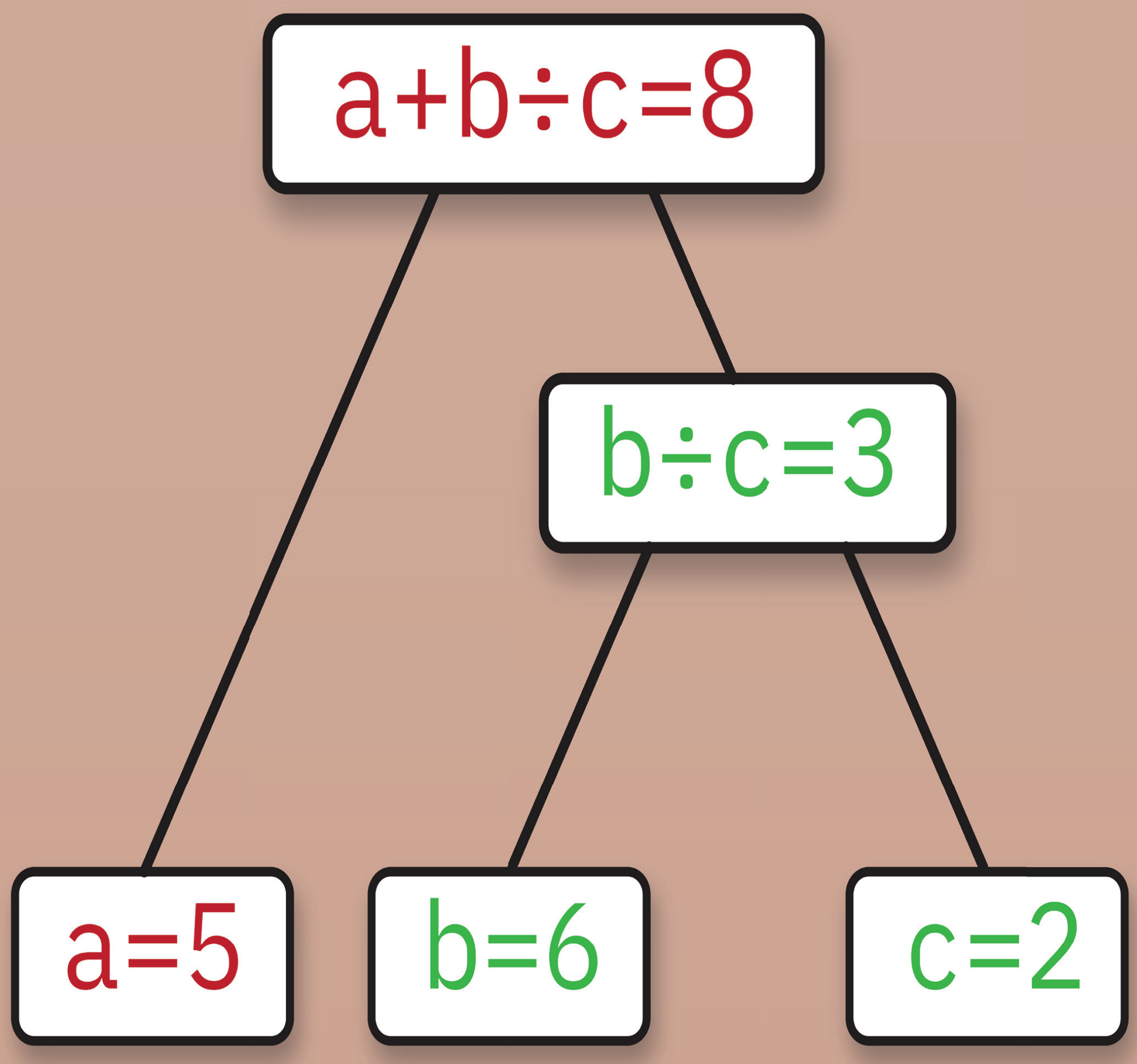
Nested Incremental Computation

Reduce, Reuse, Recycle

Incremental Computation is a software feature in which in a program tries to save time by only recomputing data that has been changed when the input changes or a structure has been modified.
Figure 1 represents a from scratch calculation where no previous information is known so the entire tree must be calculated.
In Figure 2 `[a]` has changed to 5 so anything connected to `[a]` must be recalculated (shown in red) but the data in green hasn't changed and there for can be reused saving the computer the work of the division.



1



2

Project

This project was to create an incremental program nested within another incremental one. What I did was create a tree like structure to incremntally calculate arthmatic — similar to as seen above — and keep a set of those trees in sorted order.

Process

In order to achieve this I used the rust programming language for its strong type guarantees and it's functional affordances. This allowed me to create a safe and fast program.

Arithmetic Tree

To implement the arithmetic tree each node is has an operation associated with an optional value and as well 2 children nodes if a node is a leaf it has no children and a non optional value associated with it.

To calculate the value of a node its defined operation is applied to its children, therefore to calculate the value of a tree the value of the root is calculated and the calculation recursively propagated down the tree. When a node values is calculated it stores it in its value parameter.

When the tree is modified the node that has been modified recalculates it self and propagates the change up the tree until a nodes value hasn't changed or it reaches the root

List of Trees

To implement the list of tree that maintain order a Btree is used and each tree is sorted based on the value of its root. By using a BTree a from scratch "sort" has a complexity of `O(nlogn×m)` where m is the complexity of calculating the root value and n is the length of the list.

Because we are using BTree maintaining order when a new element is added is just `O(logn)` and modifying is just a delete which is `O(logn)` the modify and insert which is `O(logn)`

Methods

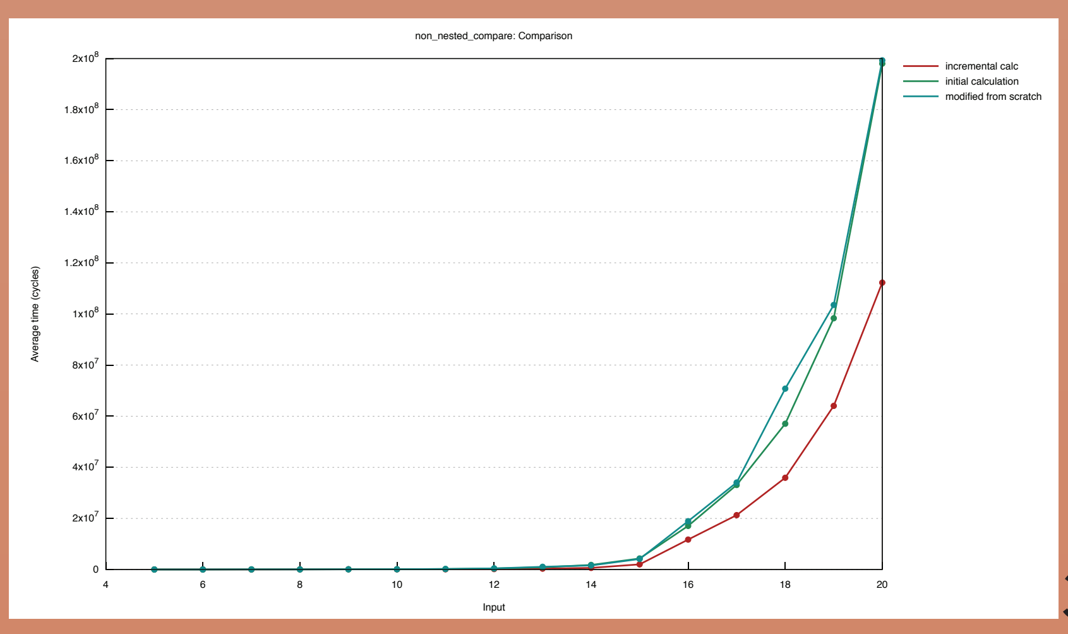
To generate my results I used the benchmarking framework for Rust called Criterion.rs I set up two different set up benchmarks one for just finding the speed-ups for the incremental calculation the arithmetic trees and for measuring the speed ups for the nested incremental program.

For the Non Nested Test I measured 3 things an incremental computation after a defined modification, an initial calculation and a from scratch calculation that included the same modification as the incremental test. I measured these with balanced tree with a depth of 5 to a depth of 20

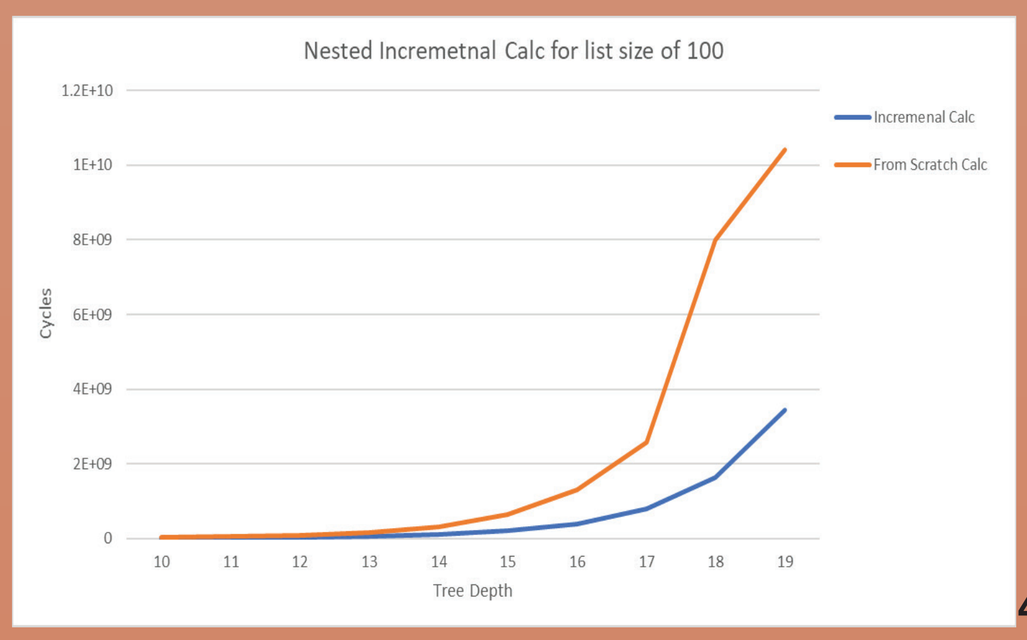
For the Nested Test I only measured two things a from scratch sort and incremental insertion as the previous test should that two from scratch tests were very similar. I also started my tree depth at 10 rather than 5 as any where from 5 to 12 showed a nominal difference in speed as seen in Figure 3. To test the affect of the size of list had on speed I measured using a list of 100, 900, 1600, and 2000.

Results

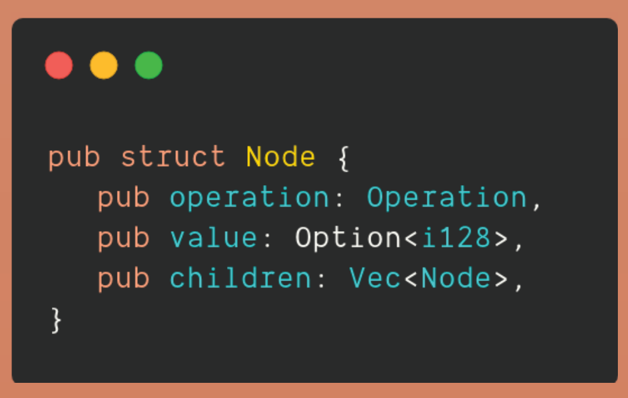
The results generated were mostly as expected. For the non nested calculations doing a incremental calculation started being faster after the trees got large and the overhead of keeping the intermediate results was out weighted by the massive size of the trees this was around a tree depth of 12 as seen in Figure 3. For the nested results it was similar however the incremental one was consistently better which makes sense as a insert should be much faster than a full sort.



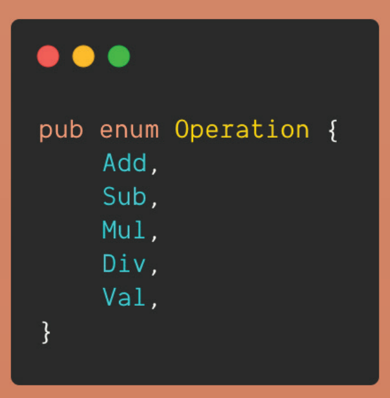
3



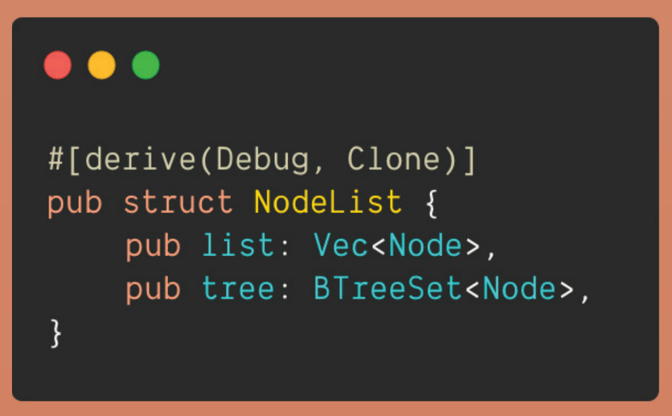
4



```
pub struct Node {
   pub operation: Operation,
   pub value: Option<i128>,
   pub children: Vec<Node>,
}
```



```
pub enum Operation {
    Add,
    Sub,
    Mul,
    Div,
    Val,
}
```



```
#[derive(Debug, Clone)]
pub struct NodeList {
    pub list: Vec<Node>,
    pub tree: BTreeSet<Node>,
}
```

Quinn Pollock
CISC 499
Supervised by Professor Joshua Dunfield