merkle rs

```rust
pub struct MerkleTree<T: Hasher> {
    current_working_tree: PartialTree<T>,
    history: Vec<PartialTree<T>>,
    uncommitted_leaves: Vec<T::Hash>,
}

impl<T: Hasher> Default for MerkleTree<T> {
    fn default() -> Self {
        Self::new()
    }
}

impl<T: Hasher> MerkleTree<T> {
    pub fn new() -> Self {
        Self {
            current_working_tree: PartialTree::new(),
            history: Vec::new(),
            uncommitted_leaves: Vec::new(),
        }
    }

    pub fn from_leaves(leaves: &[T::Hash]) -> Self {
        let mut tree = Self::new();
        tree.append(leaves.to_vec().as_mut());
        tree.commit();
        tree
    }

    pub fn root(&self) -> Option<T::Hash> {
        Some(self.layer_tuples().last()?.first()?.1)
    }

    pub fn root_hex(&self) -> Option<String> {
        let root = self.root()?;
        Some(utils::collections::to_hex_string(&root))
    }


    fn helper_nodes(&self, leaf_indices: &[usize]) -> Vec<T::Hash> {
        let mut helper_nodes = Vec::<T::Hash>::new();

        for layer in self.helper_node_tuples(leaf_indices) {
            for (_index, hash) in layer {
                helper_nodes.push(hash)
            }
        }

        helper_nodes
```

```
    }

    fn helper_node_tuples(&self, leaf_indices: &[usize]) -> Vec<Vec<(usize,
T::Hash)>> {
        let mut current_layer_indices = leaf_indices.to_vec();
        let mut helper_nodes: Vec<Vec<(usize, T::Hash)>> = Vec::new();

        for tree_layer in self.layer_tuples() {
            let mut helpers_layer = Vec::new();
            let siblings =
utils::indices::sibling_indices(&current_layer_indices);
            // Filter all nodes that do not require an additional hash to be
calculated
            let helper_indices = utils::collections::difference(&siblings,
&current_layer_indices);

            for index in helper_indices {
                if let Some(tuple) = tree_layer.get(index) {
                    helpers_layer.push(*tuple);
                }
            }

            helper_nodes.push(helpers_layer);
            current_layer_indices =
indices::parent_indices(&current_layer_indices);
        }

        helper_nodes
    }

    pub fn proof(&self, leaf_indices: &[usize]) -> MerkleProof<T> {
        MerkleProof::<T>::new(self.helper_nodes(leaf_indices))
    }

    pub fn insert(&mut self, leaf: T::Hash) -> &mut Self {
        self.uncommitted_leaves.push(leaf);
        self
    }

    pub fn append(&mut self, leaves: &mut Vec<T::Hash>) -> &mut Self {
        self.uncommitted_leaves.append(leaves);
        self
    }

    pub fn commit(&mut self) {
        if let Some(diff) = self.uncommitted_diff() {
            self.history.push(diff.clone());
            self.current_working_tree.merge_unverified(diff);
            self.uncommitted_leaves.clear();
        }
    }

    pub fn rollback(&mut self) {
        // Remove the most recent commit
```

```
            self.history.pop();
            // Clear working tree
            self.current_working_tree.clear();
            // Applying all the commits up to the removed one. This is not an
            // efficient way of doing things, but the diff subtraction is not
    implemented yet on
            // PartialMerkleTree
            for commit in &self.history {
                self.current_working_tree.merge_unverified(commit.clone());
            }
        }

        pub fn uncommitted_root(&self) -> Option<T::Hash> {
            let shadow_tree = self.uncommitted_diff()?;
            shadow_tree.root().cloned()
        }

        pub fn uncommitted_root_hex(&self) -> Option<String> {
            let root = self.uncommitted_root()?;
            Some(utils::collections::to_hex_string(&root))
        }

        pub fn abort_uncommitted(&mut self) {
            self.uncommitted_leaves.clear()
        }

        pub fn depth(&self) -> usize {
            self.layer_tuples().len() - 1
        }

        pub fn leaves(&self) -> Option<Vec<T::Hash>> {
            Some(self.layers().first()?.to_vec())
        }

        pub fn leaves_len(&self) -> usize {
            if let Some(leaves) = self.leaves_tuples() {
                return leaves.len();
            }

            0
        }

        fn leaves_tuples(&self) -> Option<&[(usize, T::Hash)]> {
            Some(self.layer_tuples().first()?.as_slice())
        }

        fn layers(&self) -> Vec<Vec<T::Hash>> {
            self.current_working_tree.layer_nodes()
        }

        fn layer_tuples(&self) -> &[Vec<(usize, T::Hash)>] {
            self.current_working_tree.layers()
        }
```

```
    fn uncommitted_diff(&self) -> Option<PartialTree<T>> {
        if self.uncommitted_leaves.is_empty() {
            return None;
        }

        let committed_leaves_count = self.leaves_len();

        let shadow_indices: Vec<usize> = self
            .uncommitted_leaves
            .iter()
            .enumerate()
            .map(|(index, _)| committed_leaves_count + index)
            .collect();
        // Tuples (index, hash) needed to construct a partial tree, since partial
tree can't
        // maintain indices otherwise
        let mut shadow_node_tuples: Vec<(usize, T::Hash)> = shadow_indices
            .iter()
            .cloned()
            .zip(self.uncommitted_leaves.iter().cloned())
            .collect();
        let mut partial_tree_tuples = self.helper_node_tuples(&shadow_indices);

        // Figuring what tree height would be if we've committed the changes
        let leaves_in_new_tree = self.leaves_len() +
self.uncommitted_leaves.len();
        let uncommitted_tree_depth =
utils::indices::tree_depth(leaves_in_new_tree);

        match partial_tree_tuples.first_mut() {
            Some(first_layer) => {
                first_layer.append(&mut shadow_node_tuples);
                first_layer.sort_by(|(a, _), (b, _)| a.cmp(b));
            }
            None => partial_tree_tuples.push(shadow_node_tuples),
        }

        // Building a partial tree with the changes that would be needed to the
working tree
        PartialTree::<T>::build(partial_tree_tuples, uncommitted_tree_depth).ok()
    }
}
```

PartialTree

```
#[derive(Clone)]
pub struct PartialTree<T: Hasher> {
    layers: Vec<Vec<(usize, T::Hash)>>,
}

impl<T: Hasher> Default for PartialTree<T> {
```

```
    fn default() -> Self {
        Self::new()
    }
}

impl<T: Hasher> PartialTree<T> {

    pub fn new() -> Self {
        Self { layers: Vec::new() }
    }

    pub fn from_leaves(leaves: &[T::Hash]) -> Result<Self, Error> {
        let leaf_tuples: Vec<(usize, T::Hash)> =
leaves.iter().cloned().enumerate().collect();

        Self::build(vec![leaf_tuples], utils::indices::tree_depth(leaves.len()))
    }

    pub fn build(partial_layers: Vec<Vec<(usize, T::Hash)>>, depth: usize) ->
Result<Self, Error> {
        let layers = Self::build_tree(partial_layers, depth)?;
        Ok(Self { layers })
    }

    fn build_tree(
        mut partial_layers: Vec<Vec<(usize, T::Hash)>>,
        full_tree_depth: usize,
    ) -> Result<Vec<PartialTreeLayer<T::Hash>>, Error> {
        let mut partial_tree: Vec<Vec<(usize, T::Hash)>> = Vec::new();
        let mut current_layer = Vec::new();

        // Reversing helper nodes, so we can remove one layer starting from 0 each
iteration
        let mut reversed_layers: Vec<Vec<(usize, T::Hash)>> =
            partial_layers.drain(..).rev().collect();

        for _ in 0..full_tree_depth {
            // Appending helper nodes to the current known nodes
            if let Some(mut nodes) = reversed_layers.pop() {
                current_layer.append(&mut nodes);
            }
            current_layer.sort_by(|(a, _), (b, _)| a.cmp(b));

            // Adding partial layer to the tree
            partial_tree.push(current_layer.clone());

            // This empties `current` layer and prepares it to be reused for the
next iteration
            let (indices, nodes): (Vec<usize>, Vec<T::Hash>) =
current_layer.drain(..).unzip();
            let parent_layer_indices = utils::indices::parent_indices(&indices);

            for (i, parent_node_index) in parent_layer_indices.iter().enumerate()
{
```

```
                match nodes.get(i * 2) {
                    // Populate `current_layer` back for the next iteration
                    Some(left_node) => current_layer.push((
                        *parent_node_index,
                        T::concat_and_hash(left_node, nodes.get(i * 2 + 1)),
                    )),
                    None => return Err(Error::not_enough_helper_nodes()),
                }
            }
        }

        partial_tree.push(current_layer.clone());

        Ok(partial_tree)
    }

    /// Returns how many layers there is between leaves and the root
    pub fn depth(&self) -> usize {
        self.layers.len() - 1
    }

    /// Return the root of the tree
    pub fn root(&self) -> Option<&T::Hash> {
        Some(&self.layers.last()?.first()?.1)
    }

    pub fn contains(&self, layer_index: usize, node_index: usize) -> bool {
        match self.layers().get(layer_index) {
            Some(layer) => layer.iter().any(|(index, _)| *index == node_index),
            None => false,
        }
    }

    pub fn merge_unverified(&mut self, other: Self) {
        // Figure out new tree depth after merge
        let depth_difference = other.layers().len() - self.layers().len();
        let combined_tree_size = if depth_difference > 0 {
            other.layers().len()
        } else {
            self.layers().len()
        };

        for layer_index in 0..combined_tree_size {
            let mut combined_layer: Vec<(usize, T::Hash)> = Vec::new();

            if let Some(self_layer) = self.layers().get(layer_index) {
                let mut filtered_layer: Vec<(usize, T::Hash)> = self_layer
                    .iter()
                    .filter(|(node_index, _)| !other.contains(layer_index,
*node_index))
                    .cloned()
                    .collect();

                combined_layer.append(&mut filtered_layer);
```

```
            }

            if let Some(other_layer) = other.layers().get(layer_index) {
                let mut cloned_other_layer = other_layer.clone();
                combined_layer.append(&mut cloned_other_layer);
            }

            combined_layer.sort_by(|(a, _), (b, _)| a.cmp(b));
            self.upsert_layer(layer_index, combined_layer);
        }
    }

    /// Replace layer at a given index with a new layer. Used during tree merge
    fn upsert_layer(&mut self, layer_index: usize, mut new_layer: Vec<(usize,
T::Hash)>) {
        match self.layers.get_mut(layer_index) {
            Some(layer) => {
                layer.clear();
                layer.append(new_layer.as_mut())
            }
            None => self.layers.push(new_layer),
        }
    }

    pub fn layer_nodes(&self) -> Vec<Vec<T::Hash>> {
        let hashes: Vec<Vec<T::Hash>> = self
            .layers()
            .iter()
            .map(|layer| layer.iter().cloned().map(|(_, hash)| hash).collect())
            .collect();

        hashes
    }

    /// Returns partial tree layers
    pub fn layers(&self) -> &[Vec<(usize, T::Hash)>] {
        &self.layers
    }

    /// Clears all elements in the ree
    pub fn clear(&mut self) {
        self.layers.clear();
    }
}
```

proof

```
pub struct MerkleProof<T: Hasher> {
    proof_hashes: Vec<T::Hash>,
}
```

```
impl<T: Hasher> MerkleProof<T> {
    pub fn new(proof_hashes: Vec<T::Hash>) -> Self {
        MerkleProof { proof_hashes }
    }

    pub fn from_bytes(bytes: &[u8]) -> Result<Self, Error> {
        Self::deserialize::<DirectHashesOrder>(bytes)
    }

    pub fn deserialize<S: MerkleProofSerializer>(bytes: &[u8]) -> Result<Self,
Error> {
        S::deserialize(bytes)
    }

    pub fn verify(
        &self,
        root: T::Hash,
        leaf_indices: &[usize],
        leaf_hashes: &[T::Hash],
        total_leaves_count: usize,
    ) -> bool {
        match self.root(leaf_indices, leaf_hashes, total_leaves_count) {
            Ok(extracted_root) => extracted_root == root,
            Err(_) => false,
        }
    }

    pub fn root(
        &self,
        leaf_indices: &[usize],
        leaf_hashes: &[T::Hash],
        total_leaves_count: usize,
    ) -> Result<T::Hash, Error> {
        if leaf_indices.len() != leaf_hashes.len() {
            return Err(Error::leaves_indices_count_mismatch(
                leaf_indices.len(),
                leaf_hashes.len(),
            ));
        }
        let tree_depth = utils::indices::tree_depth(total_leaves_count);

        // Zipping indices and hashes into a vector of (original_index_in_tree,
leaf_hash)
        let mut leaf_tuples: Vec<(usize, T::Hash)> = leaf_indices
            .iter()
            .cloned()
            .zip(leaf_hashes.iter().cloned())
            .collect();
        // Sorting leaves by indexes in case they weren't sorted already
        leaf_tuples.sort_by(|(a, _), (b, _)| a.cmp(b));
        // Getting back _sorted_ indices
        let (sorted_indices, _): (Vec<_>, Vec<_>) =
leaf_tuples.iter().cloned().unzip();
```

```
        let proof_indices_by_layers =
            utils::indices::proof_indices_by_layers(&sorted_indices,
total_leaves_count);

        // The next lines copy hashes from proof hashes and group them by layer
index
        let mut proof_layers: Vec<Vec<(usize, T::Hash)>> =
Vec::with_capacity(tree_depth + 1);
        let mut proof_copy = self.proof_hashes.clone();

        for proof_indices in proof_indices_by_layers {
            if proof_copy.len() < proof_indices.len() {
                return Err(Error::not_enough_hashes_to_calculate_root());
            }
            let proof_hashes = proof_copy.splice(0..proof_indices.len(), []);

proof_layers.push(proof_indices.iter().cloned().zip(proof_hashes).collect());
        }

        match proof_layers.first_mut() {
            Some(first_layer) => {
                first_layer.append(&mut leaf_tuples);
                first_layer.sort_by(|(a, _), (b, _)| a.cmp(b));
            }
            None => proof_layers.push(leaf_tuples),
        }

        let partial_tree = PartialTree::<T>::build(proof_layers, tree_depth)?;

        match partial_tree.root() {
            Some(root) => Ok(*root),
            None => Err(Error::not_enough_hashes_to_calculate_root()),
        }
    }

    pub fn root_hex(
        &self,
        leaf_indices: &[usize],
        leaf_hashes: &[T::Hash],
        total_leaves_count: usize,
    ) -> Result<String, Error> {
        let root = self.root(leaf_indices, leaf_hashes, total_leaves_count)?;
        Ok(utils::collections::to_hex_string(&root))
    }


    pub fn proof_hashes(&self) -> &[T::Hash] {
        &self.proof_hashes
    }

    pub fn proof_hashes_hex(&self) -> Vec<String> {
        self.proof_hashes
            .iter()
            .map(utils::collections::to_hex_string)
```

```
            .collect()
    }

    pub fn to_bytes(&self) -> Vec<u8> {
        self.serialize::<DirectHashesOrder>()
    }

    pub fn serialize<S: MerkleProofSerializer>(&self) -> Vec<u8> {
        S::serialize(self)
    }
}

impl<T: Hasher> TryFrom<Vec<u8>> for MerkleProof<T> {
    type Error = Error;

    fn try_from(bytes: Vec<u8>) -> Result<Self, Self::Error> {
        MerkleProof::from_bytes(&bytes)
    }
}

impl<T: Hasher> TryFrom<&[u8]> for MerkleProof<T> {
    type Error = Error;

    fn try_from(bytes: &[u8]) -> Result<Self, Self::Error> {
        DirectHashesOrder::deserialize(bytes)
    }
}
```