```rust
use std::collections::HashSet;
use std::fmt::{Debug, Formatter, Result as FmtResult};
use std::hash::Hash;
use std::rc::Rc;

type NodeRef<T> = Rc<Node<T>>;

#[derive(Clone)]
struct Node<T: Eq + PartialEq + Hash> {
    value: T,
    adjacents: Vec<NodeRef<T>>,
}

impl<T: Eq + PartialEq + Hash + Debug> Node<T> {
    // Create a new Node
    fn new(value: T, adjacents: Vec<NodeRef<T>>) -> Self {
        Self { value, adjacents }
    }

    // Get a reference to the node's value
    fn get_value(&self) -> &T {
        &self.value
    }
}
impl<T: Eq + PartialEq + Hash + Debug> Debug for Node<T> {
    fn fmt(&self, f: &mut Formatter<'_>) -> FmtResult {
        let adj_values: Vec<String> = self
            .adjacents
            .iter()
            .map(|n| format!("{:?}", n.get_value()))
            .collect();
        write!(
            f,
            "[value: {:?}, adjacents: \"[{}]\"]",
            self.value,
            adj_values.join(", ")
        )

```

```
    }
}

struct Graph<T: Eq + PartialEq + Hash> {
    nodes: Vec<NodeRef<T>>,
}

impl<T: Eq + PartialEq + Hash + Debug + Clone> Graph<T> {
    fn new(nodes: Vec<NodeRef<T>>) -> Self {
        Self { nodes }
    }
    fn dfs(&self, start: NodeRef<T>) -> Vec<NodeRef<T>> {
        let mut visited = HashSet::new();
        let mut result = Vec::new();
        self.dfs_recursive(&start, &mut visited, &mut result);
        result
    }
    fn dfs_recursive(
        &self,
        node: &NodeRef<T>,
        visited: &mut HashSet<T>,
        result: &mut Vec<NodeRef<T>>,
    ) where
        T: Clone,
    {
        if visited.contains(node.get_value()) {
            return;
        }

        visited.insert(node.get_value().clone());
        result.push(node.clone());

        for adj in &node.adjacents {
            self.dfs_recursive(adj, visited, result);
        }
    }
}
```