CSC3200 - FALL 2025 - ASSIGNMENT 4

DEADLINE: NOV 17TH 23:59

1. Hash Map (50%)

Please review the notes on hash maps from lecture 16. For this problem, your objective is to implement a templated hash map based datastructure for unordered_map using bucket hashing. You may use std::vector to holds the buckets and std::list as a bucket type from the STL in your implementation.

```
namespace student_std {
template <typename Key, typename T, typename Hash = std::hash<Key>>
class unordered_map {
public:
    using key_type = Key;
    using mapped_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using value_type = std::pair<Key, T>;
    using hasher = Hash;
    using reference = value_type&;
    using const_reference = const value_type&;

// ...
};
}
```

You may assume that `Key` has an `bool operator==(Key const&)`, that `T` is default-constructible and trivially copyable, that `Hash` is default-constructible and has a call operator `std::size_t operator()(Key const&) const`. Your datastructure must be default- and copy-constructible.

A constant instance of your datastructure must support the following operations: `std::size_t size()` returning the number of map entries, `bool contains(Key const&)`, `bool empty()`, `std::size_t bucket_count()` returning the current number of buckets.

A non-constant instance of your datastructure must additionally support `std::size_t erase(Key const&)` returning the number of deleted entries if any, `void clear()`, and `T& operator[](Key const&)` which returns a reference to the mapped value for the given key if any or a newly default constructed one if none previously existed.

Your unordered_map shall rehash to double the amount of buckets if the load factor reaches 2 or more.

Remark. Include `std::hash<Key>` from `<functional>`. You do not need to know anything about this class template except that it is default-constructible and has an

operator `std::size_t operator(Key const&)`. You may include the corresponding STL headers for std::pair, std::size_t, std::ptrdiff_t as in previous homework.

2. Binary Tree Iterator (50%)

Implement the following binary tree traversal inorder forward iterator from scratch:

```
namespace student_std {
template <typename BinaryTree>
class inorder_iterator {
        using value_type = typename BinaryTree::value_type;
    using difference_type = std::ptrdiff_t;
    using iterator_category = std::forward_iterator_tag;
    using reference = value_type const&;
    using pointer = value_type const*;

    reference operator*() const;
    pointer operator->() const;
    inorder_iterator& operator++();
    inorder_iterator operator++(int);
};
}
```

In addition to the prototypes above, constant instances must support equality and inequality comparison with themselves, they must be default-constructible, assignable and constructible from a const BinaryTree pointer. You may assume that every const BinaryTree type has the operations value_type const& value(), BinaryTree const* left(), BinaryTree const* right(), and BinaryTree const* parent(), which return the respective tree bunch or a nullpointer if no such child exists or in the case of parent() if the BinaryTree is itself the root node.

Submission Format

Two files: student_inorder_iterator.h, student_unordered_map.h.