Binance Book Implementation

Presented below is an implementation of the Binance Book required by the task. The container is built upon two underlying std::vectors, one for bids and one for asks. They are stored in reverse order to allow for fast insertion and deletion at the end (bids in increasing order, and asks in decreasing order). Detailed explanation is given below, next to the code blocks.

Starting with relevant includes and definitions of helper classes. Note that list is included for testing purposes only and does not contribute to the BinanceBook class definition.

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <cassert>

using Price = double;
using Quantity = double;

struct PriceQuantity
{
    Price price{};
    Quantity quantity{};

    auto operator<=>(const PriceQuantity&) const = default; // if C++20 available.
};

struct BookTicker
{
    Price bestBidPrice{};
    Quantity bestBidQty{};
    Price bestAskPrice{};
    Quantity bestAskQty{};
};
```

Concept definition for a PQContainer, which can be passed to the BinanceBook's replace() function, allowing any suitable PriceQuantity container.

```cpp
// PriceQuantity container concept
template <typename T>
concept PQContainer = requires(T cont){
    std::begin(cont);
    std::end(cont);
    requires std::is_same_v<typename T::value_type, PriceQuantity>;
};
```

The book is templated on a fixed initial capacity. If there is prior knowledge on how large the book will be, this will save some overhead on having to allocate memory on the fly.

```
// templated capacity for initialising the internal buffers
template<size_t capacity>
class BinanceBook
{
public:
    // preallocate memory for the bids and asks vectors
    BinanceBook() {
        bidsInternal.reserve(capacity);
        asksInternal.reserve(capacity);
    }

    void clear(){
        bidsInternal.clear();
        asksInternal.clear();
    }

    bool is_empty(){
        return bidsInternal.empty() && asksInternal.empty();
    }
```

There are three overloads for replace(). A const lvalue overload helps copy the input directly into the internal vectors' buffer if the size allows, avoiding heap allocation. The rvalue overload allows the external vectors' resources to transfer to the internal vectors, again avoiding unnecessary heap allocation.

```
    // const lvalue overload
    void replace(const std::vector<PriceQuantity>& bids, const
std::vector<PriceQuantity>& asks){
        // the internal buffer previously allocated for the internal vectors can be
reused.
        bidsInternal = bids;
        asksInternal = asks;
        std::reverse(bidsInternal.begin(), bidsInternal.end());
        std::reverse(asksInternal.begin(), asksInternal.end());
    }

    // rvalue overload
    void replace(std::vector<PriceQuantity>&& bids, std::vector<PriceQuantity>&&
asks){
        // use moved vectors' heap allocation directly
        bidsInternal = std::move(bids);
        asksInternal = std::move(asks);
        std::reverse(bidsInternal.begin(), bidsInternal.end());
        std::reverse(asksInternal.begin(), asksInternal.end());
    }
```

The third overload allows any combination of container types of PriceQuantity to be passed into replace(), using the PQContainer concept defined previously.

```
    // another overload to handle any container type other than vector
    void replace(const PQContainer auto& bids, const PQContainer auto& asks){
        // clear() does not get rid of vector's capacity, so the previously
        // allocated space can still be used.
        clear();
        for(auto& [price, quantity] : bids){
            bidsInternal.emplace_back(price, quantity);
        }
        for(auto& [price, quantity] : asks){
            asksInternal.emplace_back(price, quantity);
        }
        std::reverse(bidsInternal.begin(), bidsInternal.end());
        std::reverse(asksInternal.begin(), asksInternal.end());
    }
```

The update_bbo function takes a reference to a BookTicker, and is optimised for the assumption that most of the times, the updated prices will be well-behaved so no uncrossing of existing data is necessary. If we do need to uncross, the insertBid() and insertAsk() private methods are called (code below), which performs a binary search to find the place for insertion, removes the inferred values, and adds the new PriceQuantity to the vectors. During removal, due to the reversed ordering of the vector, the erase() method is called only on elements at the end, which does not involve shifting of elements. Also note that emplace_back() is used instead of push_back, so that elements can be constructed in-place for the vectors, avoiding unnecessary memory allocation.

```
    void update_bbo(BookTicker& ticker){
        // check if new best bid is higher than current, if so directly push back
to bids vector
        if (bidsInternal.empty() || bidsInternal.back().price <
ticker.bestBidPrice){
            bidsInternal.emplace_back(ticker.bestBidPrice, ticker.bestBidQty);
        } else { // if not, find where to insert; erase the values until the end,
and pushback
            insertBid(ticker.bestBidPrice, ticker.bestBidQty);
        }

        if (asksInternal.empty() || asksInternal.back().price >
ticker.bestAskPrice){
            asksInternal.emplace_back(ticker.bestAskPrice, ticker.bestAskQty);
        } else {
            insertAsk(ticker.bestAskPrice, ticker.bestAskQty);
        }
    }
```

The extract() method returns a vector of vectors of PriceQuantity, constructed with initialiser lists on bidsInternal and asksInternal's reverse iterators. Copy elision and return value optimisation means we can efficiently return by rvalue. Here we assume that extract() does not alter the book's internal states, as it was not specified in the requirements.

```
    std::vector<std::vector<PriceQuantity>> extract() const{
        return {{bidsInternal.rbegin(), bidsInternal.rend()},
{asksInternal.rbegin(), asksInternal.rend()}};
    }
```

The to_string() method traverses the internal vectors using reverse iterators, and appends appropriate values to the string, with a result resembling the example given in the requirements.

```
    std::string to_string() {
        std:: string result;
        int level = 1;
        auto localBidIt = bidsInternal.rbegin();
        auto localAskIt = asksInternal.rbegin();

        while(localBidIt != bidsInternal.rend() && localAskIt !=
asksInternal.rend()){
            result += "[" + std::to_string(level) + "] [" +
std::to_string(localBidIt->quantity) + "] " + std::to_string(localBidIt->price) + "
| ";
            result += std::to_string(localAskIt->price) + " [" +
std::to_string(localAskIt->quantity) + "]" + "\n";
            localBidIt++;
            localAskIt++;
            level++;
        }
        if(localBidIt == bidsInternal.rend()){
            while(localAskIt != asksInternal.rend()){
                result += "[" + std::to_string(level) + "] [----] ---- | ";
                result += std::to_string(localAskIt->price) + " [" +
std::to_string(localAskIt->quantity) + "]" + "\n";
                localAskIt++;
                level++;
            }
        } else {
            while(localBidIt != bidsInternal.rend()){
                result += "[" + std::to_string(level) + "] [" +
std::to_string(localBidIt->quantity) + "] " + std::to_string(localBidIt->price) + "
| ";
                result += std::string("---- [----]") + "\n";
                localBidIt++;
                level++;
            }
        }
        return result;
```

```
    }
```

The private fields and methods of the class include the two internal vectors, along with the insertion methods used in update_bbo().

```
private:
    std::vector<PriceQuantity> bidsInternal; // ascending order, best bid at the
end
    std::vector<PriceQuantity> asksInternal; // descending order, best ask at the
end
    void insertBid(Price& price, Quantity& quantity){
        // binary search to find the index of the position for insertion
        // finds the index of the first price >= new price
        int lo = 0;
        int hi = bidsInternal.size()-1;
        int mid;
        while(lo < hi){
            mid = lo + (hi - lo)/2;
            if(bidsInternal[mid].price > price){
                hi = mid;
            } else if (bidsInternal[mid].price < price){
                lo = mid+1;
            } else {
                lo = mid;
                break;
            }
        }
        auto bidIt = bidsInternal.begin() + lo;
        bidsInternal.erase(bidIt, bidsInternal.end());
        bidsInternal.emplace_back(price, quantity);
    }
    void insertAsk(Price& price, Quantity& quantity){
        // binary search to find the index of the position for insertion
        // finds the index of the first price <= new price
        int lo = 0;
        int hi = asksInternal.size()-1;
        int mid;
        while(lo < hi){
            mid = lo + (hi - lo)/2;
            if(asksInternal[mid].price > price){
                lo = mid+1;
            } else if (asksInternal[mid].price < price){
                hi = mid;
            } else {
                lo = mid;
                break;
            }
        }
        auto askIt = asksInternal.begin() + lo;
        asksInternal.erase(askIt, asksInternal.end());
```

```
        asksInternal.emplace_back(price, quantity);
    }
};
```

The main function is written with generic test cases to test various functionalities of the class API.

```
int main() {
    // Create a BinanceBook instance
    BinanceBook<30> book;

    // Test `is_empty` function
    assert(book.is_empty());

    // Test `replace` function: both a vector and a list are used to demonstrate
container flexibility
    std::vector<PriceQuantity> bids = { {100.0, 1.0}, {99.0, 2.0}, {98.0, 3.0} };
    std::list<PriceQuantity> asks = { {101.0, 1.0}, {102.0, 2.0}, {103.0, 3.0} };
    book.replace(bids, std::move(asks));
    std::cout << "Replaced with well-formed data\n";
    std::cout << book.to_string() << "\n";

    // Test `extract` function
    auto extractedData = book.extract();
    assert(extractedData.size() == 2);
    assert(extractedData[0].size() == 3);
    assert(extractedData[1].size() == 3);
    assert(extractedData[0][0].price == 100.0);
    assert(extractedData[0][1].price == 99.0);
    assert(extractedData[0][2].price == 98.0);
    assert(extractedData[1][0].price == 101.0);
    assert(extractedData[1][1].price == 102.0);
    assert(extractedData[1][2].price == 103.0);

    // Test `update_bbo` function
    BookTicker ticker1 = { 99.0, 3.0, 102.0, 3.0 };
    book.update_bbo(ticker1);
    extractedData = book.extract();
    assert(extractedData[0].size() == 2);
    assert(extractedData[1].size() == 2);
    assert(extractedData[0][0].price == 99.0);
    assert(extractedData[0][0].quantity == 3.0);
    assert(extractedData[1][0].price == 102.0);
    assert(extractedData[1][0].quantity == 3.0);
    std::cout << "Updated with new ticker, existing values inferred, duplicate
value changed\n";
    std::cout << book.to_string() << "\n";
```

```
    BookTicker ticker2 = { 105.0, 5.0, 120.0, 5.0 };
    book.update_bbo(ticker2);
    extractedData = book.extract();
    assert(extractedData[0][1].price == 99.0);
    assert(extractedData[0][1].quantity == 3.0);
    assert(extractedData[0][0].price == 105.0);
    assert(extractedData[0][0].quantity == 5.0);
    std::cout << "Updated with new ticker, existing values inferred\n";
    std::cout << book.to_string() << "\n";

    // Test `clear` function
    book.clear();
    assert(book.is_empty());

    std::cout << "All tests passed!\n";
    return 0;
}
```

Thereby concluding the implementation of the BinanceBook class.

Discussion:

Several alternative data structures were considered for this challenge. Due to the need for fast insertion and removal of elements that occur in update_bbo(), the std::list was considered. However, it is cache-unfriendly due to scattered memory usage, and binary search cannot be used to find the insertion point, meaning the linear search would dominate the complexity regardless of efficient insertion/deletion. In a simple profiling attempt using std::chrono, the list turns out to be much slower than using a vector. Similar observations were made on std::map: although insertion is O(logN) due to its underlying balanced binary tree, the erasing of elements requires traversing linearly through the data structure which is scattered in memory, increasing cache misses.

```
    BinanceBook<0> book;
    auto start_time = std::chrono::high_resolution_clock::now();

    const int cycles = 2000;
    const int end = 250;
    for(int k = 0; k < cycles; k++){
        book.replace(bids, asks);
        for(int i = 0; i < end; i++){
            BookTicker tick{i, 0, 2 * end - i, 0};
            book.update_bbo(tick);
        }
        BookTicker tick{end/2, 0, 2 * end - end/2, 0};
        book.update_bbo(tick);
        book.clear();
    }
    auto end_time = std::chrono::high_resolution_clock::now();
```

```
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end_time
- start_time).count();
    std::cout << book.to_string() << "\n";
    std::cout << "Execution time vector optimised: " << duration << " milliseconds"
<< std::endl;
```

The same test above took the list implementation > 3s to execute, while the vector implementation took 29ms.