

Order book algorithm types.

 chat.openai.com/chat/477fa3bf-d1ab-4ffc-a2ff-21207c3a3a0d

 saurav2fly@gmail.com

what are the different algorithm used to develop a order book for trading



There are several algorithms that can be used to develop an order book for trading, including:

1. **Priority Queue:** This algorithm organizes orders based on priority, such as price and time, and executes them in a specific order.
2. **Time-Priority:** This algorithm organizes orders based on the time they were placed, with priority given to the oldest orders.
3. **Price-Time Priority:** This algorithm organizes orders based on both price and time, with priority given to the orders with the best prices that were placed first.
4. **Volume-Weighted Average Price (VWAP):** This algorithm calculates the average price of a security based on the volume traded.
5. **Pro-Rata:** This algorithm allocates orders among multiple traders in proportion to their participation in the market.
6. **Implementation Shortfall:** This algorithm tries to minimize the difference between the execution price of an order and the order's benchmark price.
7. **TWAP (Time-Weighted Average Price):** This algorithm calculates the average price of a security based on the time it is traded.
8. **POV (Participation Rate):** This algorithm tries to execute a certain percentage of an order, rather than the entire order.

These are a few examples, there are many other algorithms that can be used to build an order book and the choice of algorithm depends on the specific use-case and the requirement of the trading.

saurav2fly@gmail.com

Lets code a Time-Priority order book in java script



Sure, here's an example of how you might implement a time-priority order book in JavaScript:

```

class Order {
  constructor(id, time, price, quantity) {
    this.id = id;
    this.time = time;
    this.price = price;
    this.quantity = quantity;
  }
}

class OrderBook {
  constructor() {
    this.buyOrders = [];
    this.sellOrders = [];
  }

  addOrder(order) {
    if (order.quantity > 0) {
      this.buyOrders.push(order);
    } else {
      this.sellOrders.push(order);
    }
  }

  sortOrders() {
    this.buyOrders.sort((a, b) => a.time - b.time);
    this.sellOrders.sort((a, b) => a.time - b.time);
  }

  executeOrders() {
    this.sortOrders();
    while (this.buyOrders.length > 0 &&
this.sellOrders.length > 0) {
      let buyOrder = this.buyOrders[0];
      let sellOrder = this.sellOrders[0];
      if (buyOrder.time < sellOrder.time) {
        console.log(`Executed trade: Buy
${buyOrder.quantity} at ${buyOrder.price} from ${buyOrder.id}
and Sell ${sellOrder.quantity} at ${sellOrder.price} from
${sellOrder.id}`);
        this.buyOrders.shift();
        this.sellOrders.shift();
      } else {
        break;
      }
    }
  }
}

```

The `OrderBook` class has two arrays to store the buy and sell orders respectively. The `addOrder` method adds orders to the corresponding array based on their quantity. The `sortOrders` method sorts the buy and sell orders by their time. The `executeOrders` method takes the oldest buy and sell orders and matches them, printing out the details of the trade.

Please note that this is a simple example just to give you an idea of how it can be implemented, this will not be a production ready solution, it's better to use some existing libraries for creating order book in javascript like:

`fast-order-book` , `orderbook` . Please let me know if you have any question.

 saurav2fly@gmail.com

let modify execute order function



Sure, here's an example of how you might modify the `executeOrders` function to match the highest buy and lowest sell orders:

```

executeOrders() {
  this.sortOrders();
  while (this.buyOrders.length > 0 && this.sellOrders.length
> 0) {
    let buyOrder = this.buyOrders[this.buyOrders.length-1];
    let sellOrder = this.sellOrders[0];
    if (buyOrder.price >= sellOrder.price) {
      console.log(`Executed trade: Buy
${buyOrder.quantity} at ${buyOrder.price} from ${buyOrder.id}
and Sell ${sellOrder.quantity} at ${sellOrder.price} from
${sellOrder.id}`);
      this.buyOrders.pop();
      this.sellOrders.shift();
    } else {
      break;
    }
  }
}

```

In this modified version, the `executeOrders` function takes the highest buy order and the lowest sell order, and matches them if the buy order's price is greater than or equal to the sell order's price. If a match is made, the details of the trade are printed and the orders are removed from the order book.

Please note that this is just one way to match the orders, there are other ways as well, like matching the lowest buy order with the highest sell order, it depends on the requirement. Also, I would recommend using existing libraries for creating order book in javascript like: `fast-order-book` , `orderbook` for production use. Please let me know if you have any questions.

 saurav2fly@gmail.com

In OrderBook class add a function called `getBestBid` which will return the Order which has best price in the `buyOrders` array



Sure, here's an example of how you might implement a `getBestBid` function in the `OrderBook` class:

```
class OrderBook {  
  ...  
  getBestBid() {  
    if (this.buyOrders.length === 0) {  
      console.log("No buy orders in the book");  
      return;  
    }  
    let bestBid = this.buyOrders[0];  
    for (let i = 1; i < this.buyOrders.length; i++) {  
      if (this.buyOrders[i].price > bestBid.price) {  
        bestBid = this.buyOrders[i];  
      }  
    }  
    return bestBid;  
  }  
  ...  
}
```

This function will iterate over the `buyOrders` array and find the order with the highest price. It will return the `Order` object with the best bid price. If there is no buy orders in the book it will return a message "No buy orders in the book". Please let me know if you have any questions.

 saurav2fly@gmail.com

In `OrderBook` class add a overloaded function called `getBestBid` which accepts 1 argument as input which is of type number/interge. and this function should return second best highest price from `buyOrders` array if the passed parameter to this function is 1. and similarly function return third best order by price if we pass 2 as parameter.



Sure, here's an example of how you might implement an overloaded `getBestBid` function in the `OrderBook` class that accepts an integer parameter and returns the nth highest bid price:

```
class OrderBook {
  ...
  getBestBid(n = 0) {
    if (this.buyOrders.length === 0) {
      console.log("No buy orders in the book");
      return;
    }
    if (n >= this.buyOrders.length) {
      console.log(`There are only
${this.buyOrders.length} buy orders in the book. Please enter a
valid number`);
      return;
    }
    let bestBid = this.buyOrders[0];
    let bestBidIndex = 0;
    for (let i = 1; i < this.buyOrders.length; i++) {
      if (this.buyOrders[i].price > bestBid.price) {
        bestBid = this.buyOrders[i];
        bestBidIndex = i;
      }
    }
    for (let i = 0; i < n; i++) {
      this.buyOrders.splice(bestBidIndex, 1);
      bestBid = this.getBestBid();
    }
    return bestBid;
  }
  ...
}
```

The function first checks if the `buyOrders` array is empty and returns a message "No buy orders in the book" if it is. Then it checks if the passed parameter is greater than the number of buy orders in the book and returns a message "Please enter a valid number" if it is. Then it creates two variable `bestBid` and `bestBidIndex` to hold the best bid and its index respectively. It iterates over the `buyOrders` array, compares the price of each order with

the current best bid, and updates the best bid and its index if a higher price is found. Then it iterates over the passed parameter (n) and remove the best bid order from the buyOrders array, and call `getBestBid` function again to find the next best bid. Please let me know if you have any questions or need any further assistance.

 saurav2fly@gmail.com

write a function named `getBestAsk` in the class `OrderBook` which is similar to `getBestBid` function but it should work on `sellOrders` array and it should return the Order with the lowest price



Sure, here's an example of how you might implement a `getBestAsk` function in the `OrderBook` class that is similar to the `getBestBid` function, but works on the `sellOrders` array and returns the `Order` object with the lowest ask price:

```
class OrderBook {
  ...
  getBestAsk() {
    if (this.sellOrders.length === 0) {
      console.log("No sell orders in the book");
      return;
    }
    let bestAsk = this.sellOrders[0];
    for (let i = 1; i < this.sellOrders.length; i++) {
      if (this.sellOrders[i].price < bestAsk.price) {
        bestAsk = this.sellOrders[i];
      }
    }
    return bestAsk;
  }
  ...
}
```


This function will iterate over the `sellOrders` array and find the order with the lowest price. It will return the `Order` object with the best ask price. If there is no sell orders in the book it will return a message "No sell orders in the book". Please let me know if you have any questions or need any further assistance.

 saurav2fly@gmail.com

In `OrderBook` class add a overloaded function called `getBestAsk` which accepts 1 argument as input which is of type number/interge. and this function should return second best lowest price from `sellOrders` array if the passed parameter to this function is 1. and similarly function return third best order by price if we pass 2 as parameter.



Sure, here's an example of how you might implement an overloaded `getBestAsk` function in the `OrderBook` class that accepts an integer parameter and returns the nth lowest ask price:

```

class OrderBook {
  ...
  getBestAsk(n = 0) {
    if (this.sellOrders.length === 0) {
      console.log("No sell orders in the book");
      return;
    }
    if (n >= this.sellOrders.length) {
      console.log(`There are only
${this.sellOrders.length} sell orders in the book. Please enter
a valid number`);
      return;
    }
    let bestAsk = this.sellOrders[0];
    let bestAskIndex = 0;
    for (let i = 1; i < this.sellOrders.length; i++) {
      if (this.sellOrders[i].price < bestAsk.price) {
        bestAsk = this.sellOrders[i];
        bestAskIndex = i;
      }
    }
    for (let i = 0; i < n; i++) {
      this.sellOrders.splice(bestAskIndex, 1);
      bestAsk = this.getBestAsk();
    }
    return bestAsk;
  }
  ...
}

```

This function first checks if the `sellOrders` array is empty and returns a message "No sell orders in the book" if it is. Then it checks if the passed parameter is greater than the number of sell orders in the book and returns a message "Please enter a valid number" if it is. Then it creates two variable `bestAsk` and `bestAskIndex` to hold the best ask and its index respectively. It iterates over the `sellOrders` array, compares the price of each order with the current best ask, and updates the best ask and its index if a lower price is found. Then it iterates over the passed parameter (`n`) and remove the best ask order from the `sellOrders` array, and call `getBestAsk` function again to find the next best ask. Please let me know if you have any questions or need any further assistance.

 saurav2fly@gmail.com

show the whole code



Here's an example of the complete `OrderBook` class that includes the functions you've described:

```

class OrderBook {
  constructor() {
    this.buyOrders = [];
    this.sellOrders = [];
  }

  addOrder(order) {
    if (order.side === 'buy') {
      this.buyOrders.push(order);
    } else if (order.side === 'sell') {
      this.sellOrders.push(order);
    }
  }

  executeOrder(order) {
    if (order.side === 'buy') {
      let bestAsk = this.getBestAsk();
      if (!bestAsk) return;
      if (order.price >= bestAsk.price) {
        console.log(`Order executed: ${order.id} bought
at ${bestAsk.price}`);

        this.sellOrders.splice(this.sellOrders.indexOf(bestAsk), 1);
      }
    } else if (order.side === 'sell') {
      let bestBid = this.getBestBid();
      if (!bestBid) return;
      if (order.price <= bestBid.price) {
        console.log(`Order executed: ${order.id} sold
at ${bestBid.price}`);

        this.buyOrders.splice(this.buyOrders.indexOf(bestBid), 1);
      }
    }
  }

  getBestBid(n = 0) {
    if (this.buyOrders.length === 0) {
      console.log("No buy orders in the book");
      return;
    }
    if (n >= this.buyOrders.length) {
      console.log(`There are only
${this.buyOrders.length} buy orders in the book. Please enter a
valid number`);
      return;
    }
    let bestBid = this.buyOrders[0];
  }
}

```

```

    let bestBidIndex = 0;
    for (let i = 1; i < this.buyOrders.length; i++) {
        if (this.buyOrders[i].price > bestBid.price) {
            bestBid = this.buyOrders[i];
            bestBidIndex = i;
        }
    }
    for (let i = 0; i < n; i++) {
        this.buyOrders.splice(bestBidIndex, 1);
        bestBid = this.getBestBid();
    }
    return bestBid;
}

getBestAsk(n = 0) {
    if (this.sellOrders.length === 0) {
        console.log("No sell orders in the book");
        return;
    }
    if (n >= this.sellOrders.length) {
        console.log(`There are only
${this.sellOrders.length} sell orders in the book. Please enter
a valid number`);
        return;
    }
    let bestAsk = this.sellOrders[0];
    let bestAskIndex = 0;
    for (let i = 1; i < this.sellOrders.length; i++) {
        if (this.sellOrders[i].price < bestAsk.price) {
            best

```

 saurav2fly@gmail.com

the code is incomplete please show the complete code



I apologize for the incomplete code. Here is the complete **OrderBook** class:

```

class OrderBook {
  constructor() {
    this.buyOrders = [];
    this.sellOrders = [];
  }

  addOrder(order) {
    if (order.side === 'buy') {
      this.buyOrders.push(order);
    } else if (order.side === 'sell') {
      this.sellOrders.push(order);
    }
  }

  executeOrder(order) {
    if (order.side === 'buy') {
      let bestAsk = this.getBestAsk();
      if (!bestAsk) return;
      if (order.price >= bestAsk.price) {
        console.log(`Order executed: ${order.id} bought
at ${bestAsk.price}`);

        this.sellOrders.splice(this.sellOrders.indexOf(bestAsk), 1);
      }
    } else if (order.side === 'sell') {
      let bestBid = this.getBestBid();
      if (!bestBid) return;
      if (order.price <= bestBid.price) {
        console.log(`Order executed: ${order.id} sold
at ${bestBid.price}`);

        this.buyOrders.splice(this.buyOrders.indexOf(bestBid), 1);
      }
    }
  }

  getBestBid(n = 0) {
    if (this.buyOrders.length === 0) {
      console.log("No buy orders in the book");
      return;
    }
    if (n >= this.buyOrders.length) {
      console.log(`There are only
${this.buyOrders.length} buy orders in the book. Please enter a
valid number`);
      return;
    }
    let bestBid = this.buyOrders[0];
  }
}

```

```

    let bestBidIndex = 0;
    for (let i = 1; i < this.buyOrders.length; i++) {
        if (this.buyOrders[i].price > bestBid.price) {
            bestBid = this.buyOrders[i];
            bestBidIndex = i;
        }
    }
    for (let i = 0; i < n; i++) {
        this.buyOrders.splice(bestBidIndex, 1);
        bestBid = this.getBestBid();
    }
    return bestBid;
}

getBestAsk(n = 0) {
    if (this.sellOrders.length === 0) {
        console.log("No sell orders in the book");
        return;
    }
    if (n >= this.sellOrders.length) {
        console.log(`There are only
${this.sellOrders.length} sell orders in the book. Please enter
a valid number`);
        return;
    }
    let bestAsk = this.sellOrders[0];
    let bestAskIndex = 0;
    for (let i = 1; i < this.sellOrders.length; i++) {
        if (this.sellOrders[i].price < bestAsk.price) {
            bestAsk = this

```

 saurav2fly@gmail.com

show `getBestAsk` function



Sure, here's the `getBestAsk` function in the `OrderBook` class:

```

getBestAsk(n = 0) {
  if (this.sellOrders.length === 0) {
    console.log("No sell orders in the book");
    return;
  }
  if (n >= this.sellOrders.length) {
    console.log(`There are only
${this.sellOrders.length} sell orders in the book. Please enter
a valid number`);
    return;
  }
  let bestAsk = this.sellOrders[0];
  let bestAskIndex = 0;
  for (let i = 1; i < this.sellOrders.length; i++) {
    if (this.sellOrders[i].price < bestAsk.price) {
      bestAsk = this.sellOrders[i];
      bestAskIndex = i;
    }
  }
  for (let i = 0; i < n; i++) {
    this.sellOrders.splice(bestAskIndex, 1);
    bestAsk = this.getBestAsk();
  }
  return bestAsk;
}

```

This function accepts an optional argument `n` which is a number. By default, it is set to 0. This function finds the best ask order in the `sellOrders` array and returns it. If `n` is passed as an argument, the function will return the `n`th best ask order. It finds the best ask order by looping through the `sellOrders` array and comparing the prices of each order. If the price of the current order is lower than the best ask order, it updates the best ask order. It then removes the best ask order from the array and calls the function again if `n` is greater than 0.

 saurav2fly@gmail.com

show Order class



Sure, here's an example of how the `Order` class can be implemented in JavaScript:

```
class Order {  
  constructor(id, side, price, quantity) {  
    this.id = id;  
    this.side = side;  
    this.price = price;  
    this.quantity = quantity;  
  }  
}
```

This class has 4 properties:

- `id` : a unique identifier for the order
- `side` : a string that represents whether the order is a buy or sell order ("buy" or "sell")
- `price` : a number that represents the price of the order
- `quantity` : a number that represents the quantity of the order

This class also has a constructor function that initializes the properties of an `Order` object when it is created. The constructor function takes 4 arguments: `id` , `side` , `price` , and `quantity` , and assigns them to the corresponding properties of the object.

The class might be modified according to the requirement and business logic.

 saurav2fly@gmail.com

create a function ``getOrderBook`` which returns the complete data of `buyOrders` and `sellOrders`



Sure, here's an example of how the `getOrderBook` function can be added to the `OrderBook` class:

```
class OrderBook {  
  // other code  
  
  getOrderBook() {  
    return {  
      buyOrders: this.buyOrders,  
      sellOrders: this.sellOrders  
    }  
  }  
}
```

This function returns an object that contains the `buyOrders` and `sellOrders` arrays. This function can be used to get the current state of the order book. You can use it like this:

```
let myOrderBook = new OrderBook();  
let orderBookData = myOrderBook.getOrderBook();  
console.log(orderBookData);
```

It will log the complete data of buyOrders and sellOrders array. Note that this function simply returns the existing arrays, so any changes made to the returned object will also affect the original arrays. If you want to return a copy of the arrays instead, you can use the `slice()` method to create a new copy of the array.

 saurav2fly@gmail.com

write a function 'removeBuyOrder' which remove Order from `buyOrders` array. This function takes 1 argument as input which is `id` for the order, then remove that order from `buyOrders` array.



Sure, here's an example of how the `removeBuyOrder` function can be added to the `OrderBook` class:

```
class OrderBook {  
    // other code  
  
    removeBuyOrder(id) {  
        this.buyOrders = this.buyOrders.filter(order =>  
            order.id !== id);  
    }  
}
```

This function takes one argument `id` and it uses `filter()` method to filter out the order whose id is equal to passed id. It then re-assigns the filtered array to `buyOrders` property. You can use it like this:

```
let myOrderBook = new OrderBook();  
myOrderBook.removeBuyOrder(3);
```

This will remove the order with id 3 from `buyOrders` array. You can also add some error handling to check if the passed id does not exist in the array or not.

 saurav2fly@gmail.com

write a function 'removeSellOrder' which remove Order from `sellOrders` array. This function takes 1 argument as input which is `id` for the order, then remove that order from `sellOrders` array.



Sure, here's an example of how the `removeSellOrder` function can be added to the `OrderBook` class:

```
class OrderBook {  
  // other code  
  
  removeSellOrder(id) {  
    this.sellOrders = this.sellOrders.filter(order =>  
      order.id !== id);  
  }  
}
```

This function takes one argument `id` and it uses `filter()` method to filter out the order whose id is equal to passed id. It then re-assigns the filtered array to `sellOrders` property. You can use it like this:

```
let myOrderBook = new OrderBook();  
myOrderBook.removeSellOrder(5);
```

This will remove the order with id 5 from `sellOrders` array. You can also add some error handling to check if the passed id does not exist in the array or not.

 saurav2fly@gmail.com

Write a function `updateOrder` which takes all parameter of a Order and update the order based on order `id`



Sure, here's an example of how the `updateOrder` function can be added to the `OrderBook` class:

```
class OrderBook {
  // other code

  updateOrder(id, price, quantity, side) {
    let orders = side === 'buy' ? this.buyOrders :
this.sellOrders;
    let orderIndex = orders.findIndex(order => order.id ===
id);
    if(orderIndex !== -1){
      orders[orderIndex].price = price;
      orders[orderIndex].quantity = quantity;
    } else {
      console.log("Order not found to update");
    }
  }
}
```

This function takes 4 arguments `id`, `price`, `quantity`, `side` and it uses `findIndex()` method to find the index of the order whose id is equal to passed id. Then it updates the price and quantity of that order. You can use it like this:

```
let myOrderBook = new OrderBook();
myOrderBook.updateOrder(3, 100, 20, 'buy');
```

This will update the order with id 3 from `buyOrders` array with new price 100 and quantity 20. You can also add some error handling to check if the passed id does not exist in the array or not. Also you can validate the input arguments before updating the order.

saurav2fly@gmail.com

```
``` function matchOrders(orders) { let matches = []; for (let i = 0; i <
orders.length; i++) { for (let j = i + 1; j < orders.length; j++) { if (
(orders[i].type === "buy" && orders[j].type === "sell") || (orders[i].type
=== "sell" && orders[j].type === "buy")) { if (orders[i].price >=
orders[j].price) { matches.push({ buy: orders[i].type === "buy" ? orders[i] :
orders[j], sell: orders[i].type === "sell" ? orders[i] : orders[j] });
orders.splice(j, 1); j--; } } } } return matches; } ``` form similar logic in
`executeOrder` function
```



Sure, here's an example of how the `executeOrder` function can be modified to include the matching logic:

```

class OrderBook {
 // other code

 executeOrder(order) {
 if (order.type === 'buy') {
 let sellOrders = this.getSellOrders();
 let matches = [];
 for (let i = 0; i < sellOrders.length; i++) {
 if (order.price >= sellOrders[i].price) {
 matches.push({
 buy: order,
 sell: sellOrders[i]
 });
 this.removeSellOrder(sellOrders[i].id);
 i--;
 }
 }
 if (matches.length > 0) {
 this.matches.push(...matches);
 } else {
 this.buyOrders.push(order);
 }
 } else {
 let buyOrders = this.getBuyOrders();
 let matches = [];
 for (let i = 0; i < buyOrders.length; i++) {
 if (order.price <= buyOrders[i].price) {
 matches.push({
 buy: buyOrders[i],
 sell: order
 });
 this.removeBuyOrder(buyOrders[i].id);
 i--;
 }
 }
 if (matches.length > 0) {
 this.matches.push(...matches);
 } else {
 this.sellOrders.push(order);
 }
 }
 }
}

```

In this modified version of `executeOrder` function, when a new order is placed, it checks for any matching orders in the opposite side(buy or sell) and if any matching order is found, it pushes the matched orders to the `matches` array and removes them from the respective `buyOrders` or `sellOrders` array. If no matching orders are found then it simply pushes the new order to the respective `buyOrders` or `sellOrders` array.

You can use it like this:

```
let myOrderBook = new OrderBook();
myOrderBook.executeOrder({id: 1, price: 100, quantity: 20,
type: 'buy'});
```

This will check for any matching sell orders with price less than or equal to 100 and if found, will add those matched orders in `matches` array and remove them from `sellOrders` array. If no matching sell orders are found, it will add this order to `buyOrders` array. It also assumes that you have implemented the `removeBuyOrder` and `removeSellOrder` function as discussed before.

Please note that this is just one way to implement matching logic and there could be other ways as well to achieve this.

saurav2fly@gmail.com

in `executeOrder` function all `Order` of `buyOrders` array will iterate over all `Order` of `sellOrders` array and if price of both the orders are same and quantity is also same remove those orders from their respective arrays



Here is an example of how you could implement a modified version of the `executeOrder` function in the `OrderBook` class that matches orders based on price and quantity:



```

class OrderBook {
 // ... existing code ...

 executeOrder(order) {
 let matches = [];
 for (let i = 0; i < this.buyOrders.length; i++) {
 for (let j = 0; j < this.sellOrders.length; j++) {
 if (
 (this.buyOrders[i].type === "buy" &&
this.sellOrders[j].type === "sell")
 && this.buyOrders[i].price >=
this.sellOrders[j].price
 && this.buyOrders[i].quantity ===
this.sellOrders[j].quantity
) {
 matches.push({
 buy: this.buyOrders[i],
 sell: this.sellOrders[j]
 });
 this.buyOrders.splice(i, 1);
 this.sellOrders.splice(j, 1);
 i--;
 break;
 }
 }
 }
 return matches;
 }
 // ... existing code ...
}

```

This function iterates over the `buyOrders` array and the `sellOrders` array and compares the price and quantity of each order. If the price and quantity of a buy order and sell order match, then the orders are removed from their respective arrays and added to the `matches` array. The `matches` array is then returned by the function.

 saurav2fly@gmail.com

show Order class



```
class Order {
 constructor(id, type, price, quantity) {
 this.id = id;
 this.type = type;
 this.price = price;
 this.quantity = quantity;
 }
}
```

 saurav2fly@gmail.com

in `executeOrder` function remove `buyOrders[i].type` check



```
executeOrder(order) {
 for (let i = 0; i < this.buyOrders.length; i++) {
 for (let j = 0; j < this.sellOrders.length; j++) {
 if (this.buyOrders[i].price >= this.sellOrders[j].price
&& this.buyOrders[i].quantity === this.sellOrders[j].quantity)
 {
 this.buyOrders.splice(i, 1);
 this.sellOrders.splice(j, 1);
 i--;
 break;
 }
 }
 }
}
```

This function will iterate over each order in the `buyOrders` array and the `sellOrders` array. If it finds an order with the same price and quantity, it will remove both of those orders from the `buyOrders` and `sellOrders` arrays using the `Array.prototype.splice()` method. And also decrementing `i` variable so that the next iteration starts from the same index.

 saurav2fly@gmail.com

In `executeOrder` function check if quantity of buy and sell order are same then remove orders from both the array `buyOrders` and `sellOrders`. otherwise just subtract and remove the other order from their respective arrays



```

class OrderBook {
 //...
 executeOrder(order) {
 let matches = [];
 for (let i = 0; i < this.buyOrders.length; i++) {
 for (let j = 0; j < this.sellOrders.length; j++) {
 if (this.buyOrders[i].price >=
this.sellOrders[j].price) {
 if (this.buyOrders[i].quantity ==
this.sellOrders[j].quantity) {
 matches.push({
 buy: this.buyOrders[i],
 sell: this.sellOrders[j]
 });
 this.buyOrders.splice(i, 1);
 this.sellOrders.splice(j, 1);
 i--;
 break;
 } else if (this.buyOrders[i].quantity <
this.sellOrders[j].quantity) {
 matches.push({
 buy: this.buyOrders[i],
 sell: this.sellOrders[j]
 });
 this.sellOrders[j].quantity -=
this.buyOrders[i].quantity;
 this.buyOrders.splice(i, 1);
 i--;
 break;
 } else {
 matches.push({
 buy: this.buyOrders[i],
 sell: this.sellOrders[j]
 });
 this.buyOrders[i].quantity -=
this.sellOrders[j].quantity;
 this.sellOrders.splice(j, 1);
 }
 }
 }
 }
 return matches;
 }
 //...
}

```

 saurav2fly@gmail.com

generate test data to test all functions in `OrderBook`



Here is an example of test data that can be used to test all functions in the `OrderBook` class:

```

let order1 = new Order(1, "buy", 100, 10);
let order2 = new Order(2, "buy", 90, 15);
let order3 = new Order(3, "buy", 80, 20);
let order4 = new Order(4, "sell", 110, 5);
let order5 = new Order(5, "sell", 120, 10);
let order6 = new Order(6, "sell", 130, 15);

let orderBook = new OrderBook();
orderBook.addOrder(order1);
orderBook.addOrder(order2);
orderBook.addOrder(order3);
orderBook.addOrder(order4);
orderBook.addOrder(order5);
orderBook.addOrder(order6);

console.log(orderBook.getBestBid()); // should return order3
console.log(orderBook.getBestBid(1)); // should return order2
console.log(orderBook.getBestAsk()); // should return order4
console.log(orderBook.getBestAsk(1)); // should return order5
console.log(orderBook.getOrderBook()); // should return {
buyOrders: [order3, order2, order1], sellOrders: [order4,
order5, order6] }

orderBook.executeOrder();
console.log(orderBook.getOrderBook()); // should return {
buyOrders: [order2, order1], sellOrders: [order5, order6] }

orderBook.removeBuyOrder(1);
console.log(orderBook.getOrderBook()); // should return {
buyOrders: [order2], sellOrders: [order5, order6] }

orderBook.removeSellOrder(4);
console.log(orderBook.getOrderBook()); // should return {
buyOrders: [order2], sellOrders: [order5, order6] }

orderBook.updateOrder(2, "sell", 110, 20);
console.log(orderBook.getOrderBook()); // should return {
buyOrders: [], sellOrders: [order2, order5, order6] }

```

It's important to note that the test data above is a simplified example, and it's not a complete or production ready code. It's also important to test all the edge cases, and add more test cases to validate all the functionalities of the code.

add testcase for `executeOrder` in the above test case



Here is an example test case for the `executeOrder` function in the `OrderBook` class:

```
// Test executeOrder function
let buyOrder1 = new Order(1, "buy", 100, 10);
let buyOrder2 = new Order(2, "buy", 90, 15);
let sellOrder1 = new Order(3, "sell", 110, 5);
let sellOrder2 = new Order(4, "sell", 100, 10);

orderBook.addOrder(buyOrder1);
orderBook.addOrder(buyOrder2);
orderBook.addOrder(sellOrder1);
orderBook.addOrder(sellOrder2);

let matchedOrders = orderBook.executeOrder();
console.log(matchedOrders);
// expected output: [{buy: Order {id: 2, type: "buy", price:
90, qty: 15}, sell: Order {id: 4, type: "sell", price: 100,
qty: 10}}]
console.log(orderBook.getOrderBook());
// expected output: {buyOrders: [Order {id: 1, type: "buy",
price: 100, qty: 10}], sellOrders: [Order {id: 3, type: "sell",
price: 110, qty: 5}]}
```

This test case creates 4 test orders, 2 buy and 2 sell, and adds them to the `OrderBook`. It then calls the `executeOrder` function and checks that the matched orders returned are as expected and that the remaining buy and sell orders in the `OrderBook` are also as expected.

In `getBestBid` function before recursion we are slicing the `buyOrders` array which is remove those element permanently



Yes, that is correct. The `Array.prototype.slice()` method creates a new array with the specified elements, and it does not modify the original array. However, if you want to avoid modifying the original array and still get the desired result, you can use the `Array.prototype.slice()` method in a way that only returns a copy of the desired elements without modifying the original array. For example:

```
let copyOfBuyOrders = buyOrders.slice();
let bestBid = getBestBid(n, copyOfBuyOrders);
```

This will create a copy of the `buyOrders` array and pass it to the `getBestBid` function, so the original array will remain unchanged.

 saurav2fly@gmail.com

can you please rewrite the `getBestBid` function in a way where we get the same result but without altering the real state `buyOrders`



Sure, here's one way to rewrite the `getBestBid` function to achieve the same result without altering the state of the `buyOrders` array:



```

getBestBid(n) {
 let tempBuyOrders = [...this.buyOrders];
 tempBuyOrders.sort((a,b) => b.price - a.price);
 return tempBuyOrders[n];
}

```

In this version, we create a new variable called `tempBuyOrders` which is a copy of the original `buyOrders` array. We then sort the `tempBuyOrders` array by price in descending order, and return the nth element of this sorted array. This way, the original `buyOrders` array is not modified. }

Another way to do this is to use the sort method and slice method together.

```

getBestBid(n) { this.buyOrders.sort((a,b) => b.price - a.price); return
this.buyOrders.slice(0, n+1)[n]; }

```

This way we use the sort method on the original array but only select the first n orders, and return the nth order.

Note: This modification will also work for `getBestAsk` function.

 saurav2fly@gmail.com

show `getBestBid` function



```
getBestBid(n) {
 let bestBid;
 let sortedBuyOrders = [...this.buyOrders].sort((a, b) =>
b.price - a.price);
 if (n < sortedBuyOrders.length) {
 bestBid = sortedBuyOrders[n];
 }
 return bestBid;
}
```

This function creates a copy of the `buyOrders` array and sorts it in descending order by price. It then checks if the passed parameter `n` is less than the length of the sorted array, and if so it returns the `n`th element of the array as the best bid. This way the original `buyOrders` array is not modified.

 saurav2fly@gmail.com

show `getBestAsk`` function



```
getBestAsk(n) {
 let tempSellOrders = [...this.sellOrders];
 tempSellOrders.sort((a, b) => a.price - b.price);
 return tempSellOrders[n];
}
```

This function creates a temporary copy of the `sellOrders` array, sorts it by price in ascending order, and returns the nth element of the array. This way, the original `sellOrders` array is not altered.