COMP 2012 Object-Oriented Programming and Data Structures

Assignment 1 Bitcoin

Introduction

In this assignment, we will implement a simplified version of a blockchain-based cryptocurrency system that simulates the creation and transfer of bitcoins.

In short, you can treat Bitcoin system as a small market, where each user will get some money (bitcoins here) in their pocket, and make transactions in it. Unlike conventional trading markets that requires either person-to-person trust or a trusted third-party, Bitcoin system builds the trust of the transaction based on **the system algorithm itself**.

This validation relies on a record of all previous transactions, known as the **Blockchain**, which consists of a series of **Block**. To add a new Block, participants (called **Miners**) must solve a mathematical problem, and the first who succeeds will broadcast the new Block to all the other Miners. This process, known as **Mining a Block**, rewards Miners for their efforts. Trust in Bitcoin is maintained through consensus among all participants on the Blockchain.

For more information about the Bitcoin system, you may refer to the Bitcoin Wiki.

Additional remarks

Read the FAQ page for some common clarifications. You should check that a day before the deadline to ensure you do not miss any clarification, even if you have already submitted your work.

Additionally, it is crucial to continuously monitor the change log for any recent updates to the files. Staying updated with the change log will help you stay informed about any modifications or additions that may impact your work.

Submission details are in the Submission and Deadline section.

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment; three is ADDITIONAL PENALTY. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there may be additional disciplinary actions from the department and university, up to and including expulsion.
- The use of generative AI tools such as ChatGPT and CoPilot are not allowed for completing the exercises.

Main Program Functionalities

This section provides an overview of the example workflow of the bitcoin system and introduces the main functionalities that need to be implemented in the program.

Note that in our version of the Bitcoin system, we made **the following simplifications**:

- The basic trading unit of bitcoin transactions is called a `BitcoinBatch`, and each batch contains a number of bitcoins which, once specified, cannot be modified.
- The number of bitcoins in a `BitcoinBatch` is an **integer**.
- There will be only one Miner; the system is the only Miner which performs the block mining job and validates all transactions.

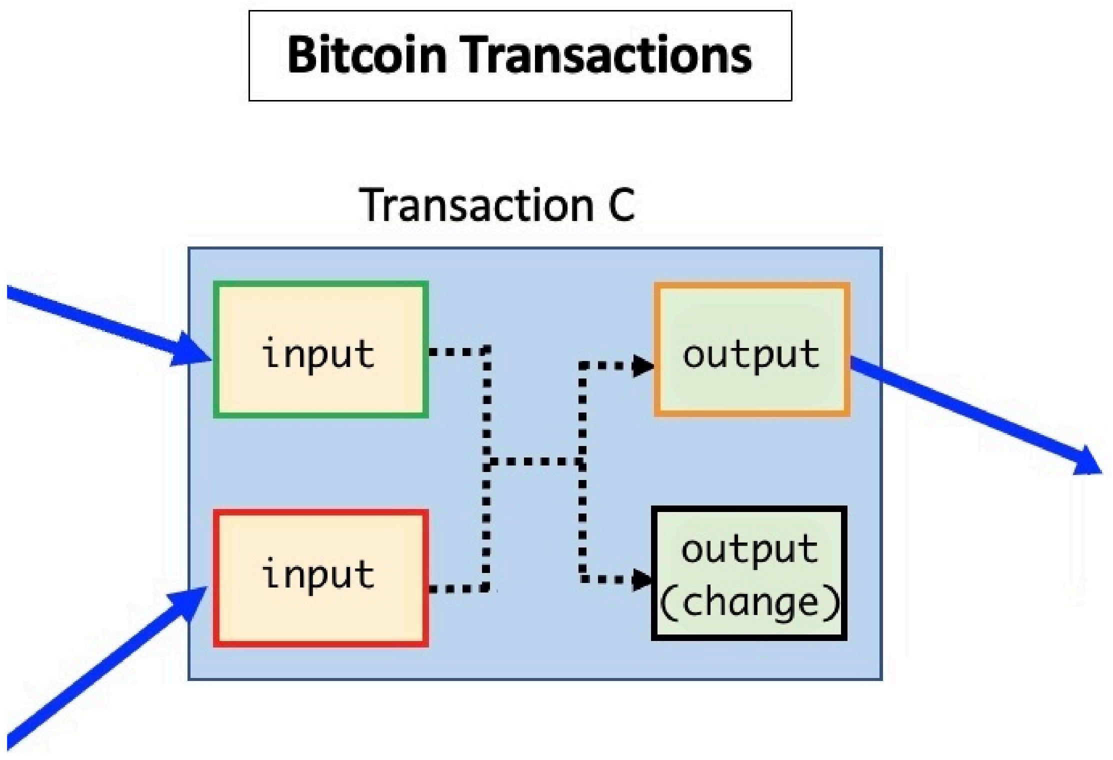
Next, three main functionalities of our Bitcoin system during transactions will be explained in detail.

Proposing Transactions in the Bitcoin System (`transact` function)

As a trading system, the first thing we need to care about is the proposing of a transaction. During the initialization of the system, we will give each user some batches of bitcoins to start with, and then let them make transactions.

There are three components of a transaction between a receiver and a sender: **inputs, outputs and transaction fee**. A transaction takes some batches of bitcoins (**inputs**) from a sender, and transfer some batches of bitcoins (**outputs**) to the receivers. These output batches are **new** batches of bitcoins, and they are regarded as the property of the receivers and can be used as inputs for future transactions.

Here is a brief overview of the transactions (from Bitcoin Wiki):



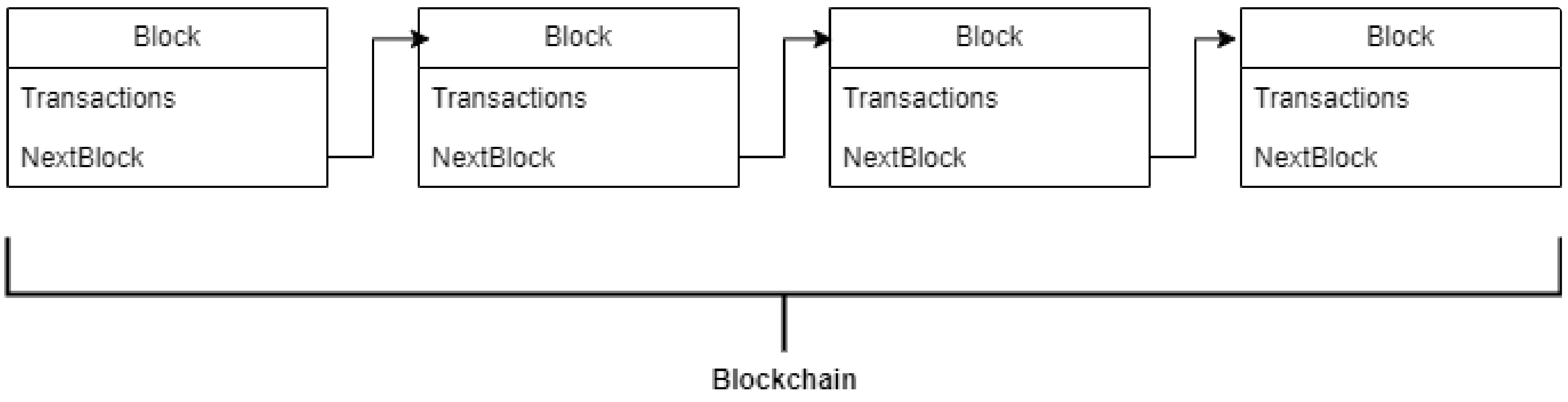
You may notice that in the above figure, there is an output bitcoin batch called "change". This is because a bitcoin batch cannot be modified. So if the total amount of bitcoins in the input batches is larger than or equal to (transaction amount + transaction fee), a new bitcoin batch, which holds the remaining amount, has to be created and will be sent back to the sender as a new batch of bitcoins. In our implementation, every transaction will have exactly **2 outputs**, one for **the receiver (with the transaction amount)**, and one for **the sender (with the non-negative change)**.

To illustrate the concept of change, let's look into an example. Suppose there are three users in the system: Alice, Bob and Cindy. Suppose Cindy has no bitcoins initially, then Alice sends 100 bitcoins to Cindy and Bob sends 80 bitcoins to Cindy (let's ignore the fees now). Cindy will now have 180 bitcoins. Later, Cindy wants to give back Alice 140 bitcoins, but she would have to send 180 bitcoins instead of 140 since bitcoin batches are undividable. To avoid such awkward situation, change comes to help, and Cindy can setup a transaction with an input of a bitcon batch of 180 bitcoins and gets 40 bitcoins back as the change (in a new bitcoin batch). (We ignore the transaction fee here).

The `transact` function will do the job of proposing a transaction to the Bitcoin system. It determines the key components of the transaction and creates an instance (a Transaction object) of the `Transaction` class. Then the new instance will be included in the system's pending transactions for further processing and acceptance --- mining by the system.

Accepting Transactions in the Bitcoin System (`mineBlock` function)

In the context of Bitcoin, the system's acceptance/recognition record of transactions is called a **block**, which is stored with a data structure named **Blockchain**. Here's an overview of a Blockchain:



From the figure you can see that a blockchain is indeed **a linked list of blocks**, where each block contains another linked list of transactions and a pointer to the next block. In Bitcoin terms, accepting transactions (adding transaction records to the system) is named as **mining a block**. Such a procedure is achieved by `mineBlock` function in our system.

When a new block is mined, the transactions in the block will be considered as completed, and it will be added **at the end** of the blockchain. After mining a block, the input batches of bitcoins will be removed from the sender, and the output batches of bitcoins will be added to the receivers. The pending transactions will be updated accordingly.

To make the problem more interesting and closer to real life, each time when a block is mined, only the transactions with **the highest 3 transaction fees** will be included in the block. If there are **less than 3 pending transactions, all of them will be included. The transactions inside a block should be in a decreasing order with respect to the transaction fee.** In real world scenarios, these fees are paid to the miners. In our simplified case, since we don't have really have miners other than the system itself, we will just use them to determine the priority of the transactions for mining.

Managing Unspent Bitcoin Batches (`UnspentBitcoinBatches` class)

Another important functionality of the Bitcoin system is to manage the unspent bitcoin batches. Since we do not have a unique identifier for each bitcoin batch, we need to keep track of the unspent bitcoin batches to **avoid double spending**. By double spending, we mean the case that the same bitcoin batch is used as an input for more than one transaction.

The functionality is managed by `UnspentBitcoinBatches` class, which is only instantiated inside the `BitcoinSystem` once, and it operates like a global manager of the system. Every time one transaction is proposed, it will scan through all the sender's unspent bitcoin batches to find inputs for the new transaction. The transaction will only be considered for processing when the sender has enough unspent bitcoin batches to cover the transaction amount and fee. And when a block is mined, the system will also add the outputs of the transactions in the block to the `UnspentBitcoinBatches` set.

Skeleton Code

Please download the skeleton code HERE and read the code as you progress through the following sections. It should contain the following files:

- `user.h` and `user.cpp`: These files are the header and source files, respectively, for the `User` class.
- `transaction.h` and `transaction.cpp`: These files are the header and source files, respectively, for the `BitcoinBatch`, `Transaction` and `UnspentBitcoinBatches` classes.
- `blockchain.h` and `blockchain.cpp`: These files are the header and source files, respectively, for implementing the `Block` and `Blockchain` classes.
- `bitcoin.h` and `bitcoin.cpp`: These files are the header and source files, respectively, for the `Bitcoin` class.
- `main.cpp`: This file serves as the driver program for the assignment. You can use it to interact with the classes and test your implementation.
- `main_test.cpp`: This file is designed to facilitate the testing of functions on your local machine. It includes public test cases on ZINC but does not provide the expected output. Instead, it prints the output of your implementation. You should refer to ZINC or test case results for the expected output and perform necessary comparisons. You can find the test case ID either on ZINC or in the filenames of the test case results.
- `Makefile`: To compile the driver program `main.cpp`, navigate to the directory containing the makefile in the terminal, and run `make` or `make all`. If you want to compile with `main_test.cpp` instead, run `make test` in the terminal.

Here are some class and function illustrations that are included in the skeleton:

- The `User` class represents a user in the bitcoin system. The user has a name, a balance, and a list of bitcoin batches. The user can add a bitcoin batch or spend a bitcoin batch. The list of bitcoin batches store the "property" of the user. These bitcoin batches usually come from previous transactions, and the user can use them as inputs for future transactions. The `addBitcoinBatch` function is only used to initialize the user with some initial bitcoin batches. The `getBalance` function will return the sum of the amount of bitcoins in all of the user's bitcoin batches.
- The `Transaction` class stores the required information for a transaction, including both `inputs`, `outputForReceiver`, `outputForSender` and `fee`. It also records the corresponding `receiver` and `sender` for information. Note that here the `outputForReceiver` is the transaction amount and the `outputForSender` is the change.
- The `UnspentBitcoinBatches` class keeps track of all the unspent bitcoin batches to avoid double spending. The checking procedure during each transaction is the following: If the sender has enough unspent bitcoin batches to cover the transaction amount and fee, the transaction will be added to the pending transactions, and the input batches of this transaction will be removed from `UnspentBitcoinBatches`. Otherwise, the transaction will not be added to the pending transactions. When mining a block, the system should add the outputs of the transactions in the block to the `UnspentBitcoinBatches` set.
- The `BitcoinSystem` class represents a simple bitcoin system. The system has a blockchain, a list of users, a list of pending transactions, and an `UnspentBitcoinBatches` instance. The `addBitcoinToUser` function is used to give a user some initial bitcoin batches. The `transact` and `mineBlock` are the key functions for transaction proposing and accepting in the Bitcoin system, whose workflows will be illustrated later in Task 4.

Illustration Example

Here's an example of the main function to help you understand the flow:

```
int main(){
    BitcoinSystem system; // Initialize the blockchain system

    system.addUser("Alice"); // Add a user named Alice
    system.addUser("Bob"); // Add a user named Bob
    system.addBitcoinToUser("Alice", 100); // Give Alice a 100 bitcoin batch
    system.addBitcoinToUser("Bob", 50); // Give Bob a 50 bitcoin batch

    system.transact("Alice", "Bob", 30, 3); // Add a transaction from Alice to Bob, 30 bitcoir
    system.transact("Bob", "Alice", 20, 2); // Add a transaction from Bob to Alice, 20 bitcoir
    system.printUsers();
    system.printPendingTransactions();
    system.mineBlock(); // Mine a block
    system.printBlockchain();
    system.printUsers();

    return 0;
  }
```

The output of the above will be:

```
Transaction added to pending transactions
Transaction added to pending transactions
------------ Users ------------
User: Alice     Balance: 100
User: Bob       Balance: 50
-------------------------------
------------ Pending Transactions ------------
Transaction from Alice to Bob
Input 0: Alice 100
Output 0: Bob 30
Output 1: Alice 67
Transaction from Bob to Alice
Input 0: Bob 50
Output 0: Alice 20
Output 1: Bob 28
---------------------------------------------
Block mined
------------ Blockchain ------------
Blockchain length: 1
Block
Transaction from Alice to Bob
Input 0: Alice 100
Output 0: Bob 30
Output 1: Alice 67
Transaction from Bob to Alice
Input 0: Bob 50
Output 0: Alice 20
Output 1: Bob 28
-----------------------------------
------------ Users ------------
User: Alice     Balance: 87
User: Bob       Balance: 58
-------------------------------
```

From the main function, you can tell that we will first add some users in the system, and give them some initial output batches. Then we will add some transactions, and mine a block. After mining the block, the transactions will be completed, and the users' balance will be updated accordingly. From the output, you can see that the transaction is confirmed and the users' balance is updated only after the block that contains the transaction is mined.

Notice that the core functions of this assignment is the `BitcoinSystem::transact` and `BitcoinSystem::mineBlock` functions. There are also some other functions that you need to implement to support these two functions. All the print functions are provided; you can use them to help you debug your code.

Tasks

In this assignment, you will need to complete the following 5 tasks to construct a Bitcoin system. Each task will be briefly explain in below.

Task #1

You will need to finish the non-default constructors for class `User`, `Transaction` and `BitcoinBatch`. In addition, you will need to implement `BitcoinSystem::addUser` and `User::getBalance`. The default constructor and some easy accessors are already given.

Task #2

You will need to finish `BitcoinSystem::addBitcoinToUser`. In order to do this, you will also need to implement `BitcoinSystem::getUser`, `User::addBitcoinBatch` and `UnspentBitcoinBatches::addBatch` to support this functionality.

Task #3

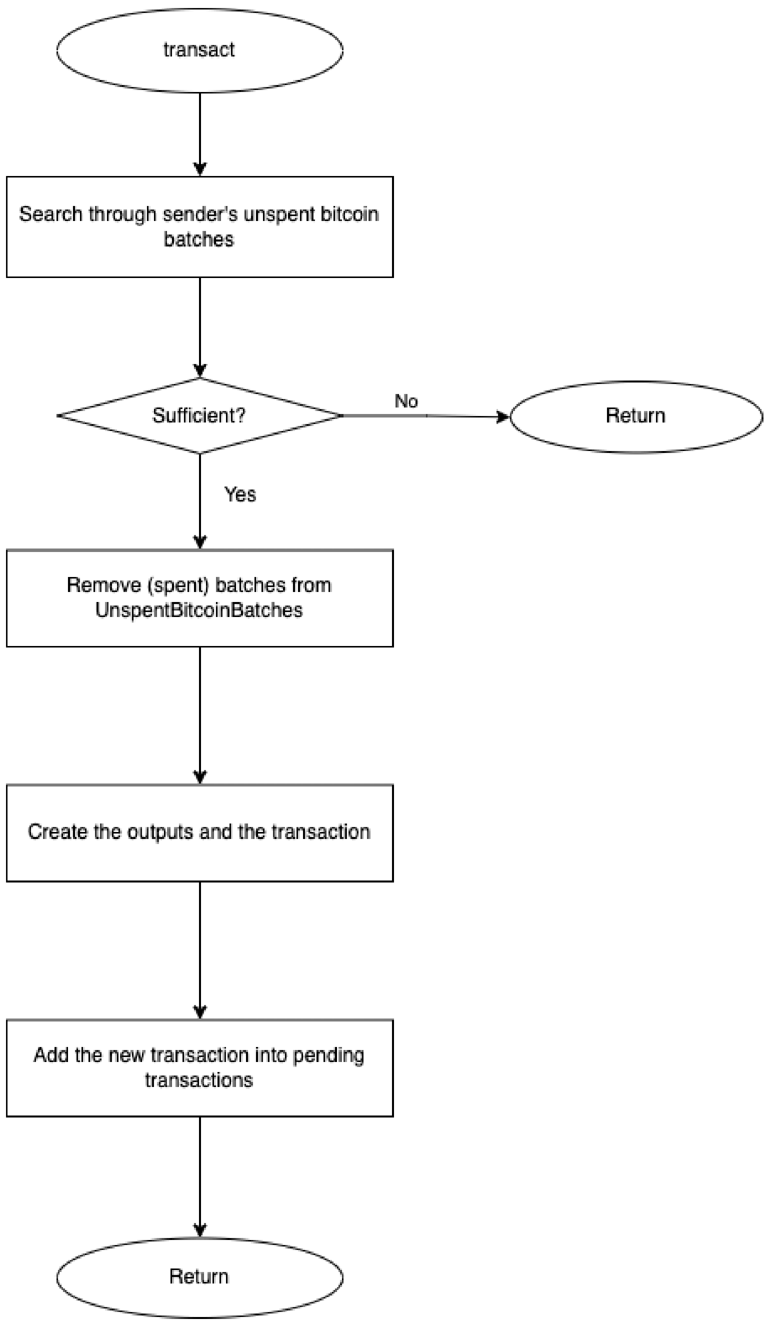
Implement `BitcoinSystem::transact`. You also need to implement `BitcoinSystem::addPendingTransaction`, `UnspentBitcoinBatches::isSpent`, `UnspentBitcoinBatches::spendBatch` and `UnspentBitcoinBatches::spend` to support this functionality.

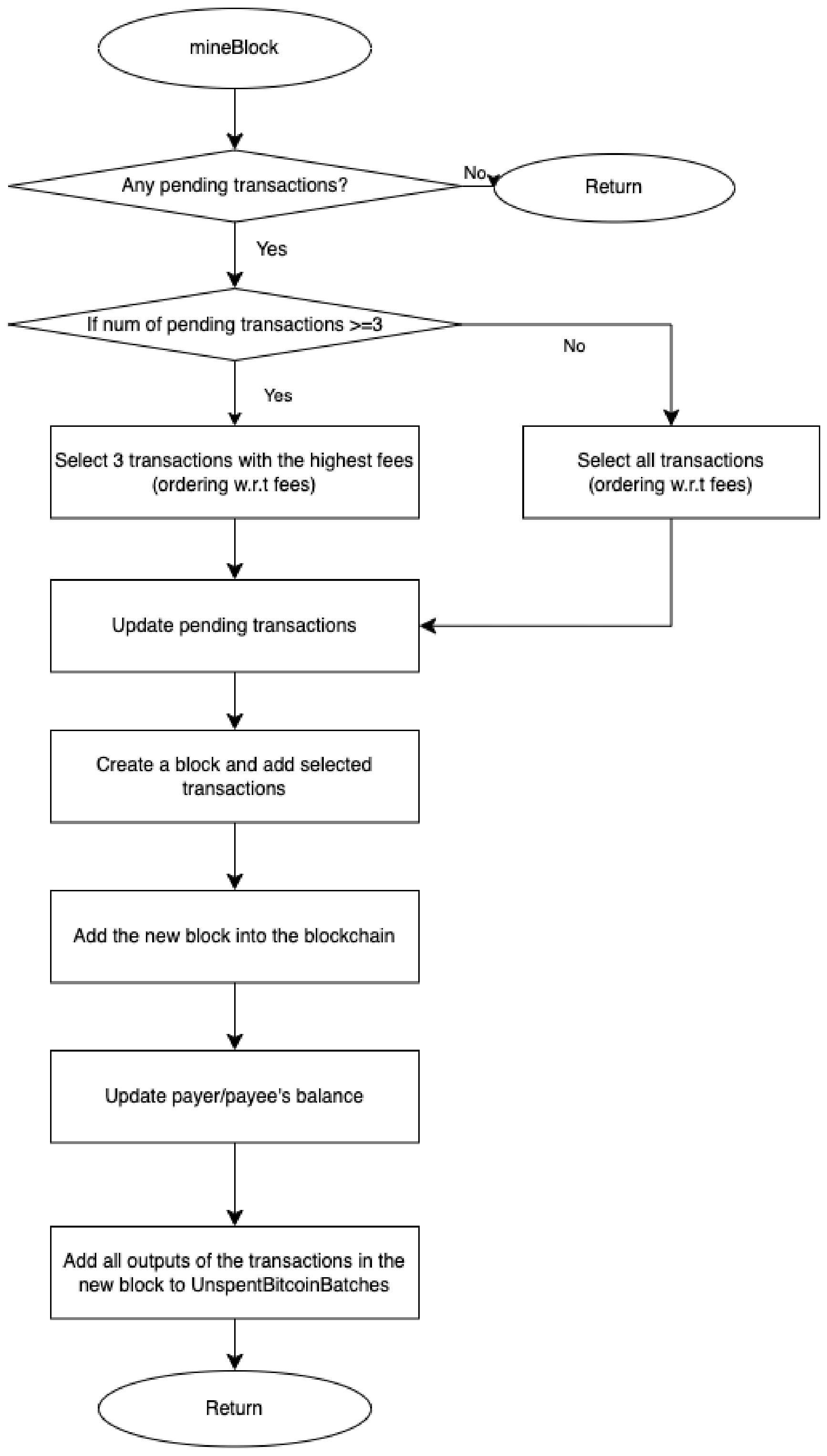
Hints: Implementing `UnspentBitcoinBatches::isSpent`, `UnspentBitcoinBatches::spendBatch` may be useful for the `UnspentBitcoinBatches::spend` function. Please refer to the flow chart for the detailed logic.

Task #4

Implement `BitcoinSystem::mineBlock`. You also need to implement `BitcoinSystem::selectPendingTransaction`, `BitcoinSystem::removePendingTransaction`, `Transaction::getFee`, `Blockchain::addBlock`, `Block::addTransaction` and `User::spend` to support this functionality.

You may refer to the below flow chart for the detailed logic. You don't need to consider the case when the fees of the transactions are the same.


transact
Search through sender's unspent bitcoin batches
Sufficient?
No
Return
Yes
Remove (spent) batches from UnspentBitcoinBatches
Create the outputs and the transaction
Add the new transaction into pending transactions
Return


mineBlock
Any pending transactions?
No
Return
Yes
If num of pending transactions >=3
No
Yes
Select 3 transactions with the highest fees (ordering w.r.t fees)
Select all transactions (ordering w.r.t fees)
Update pending transactions
Create a block and add selected transactions
Add the new block into the blockchain
Update payer/payee's balance
Add all outputs of the transactions in the new block to UnspentBitcoinBatches
Return

Task #5

Implement the destructors for all the classes. You should free all the memory properly and also avoid double free. For some of the `BitcoinBatches`, they may be included in some different classes, you should be careful about this.

Important Notes

- **Order matters:** For those functions that add new elements to the system, you should always append the new element to **the end** of the array or linked list. Also, when removing elements from the system, you should **preserve the order** of the elements. You should be careful about this to make sure your printed results are exactly the same as the expected outputs.
- **Ownership matters:** During the process of proposing and accepting transactions, you should always pay attention to the ownership transfer of the bitcoin batches. It is rather essential when you are dealing with the destructors.

Special Note

Please pay special attention to that the transactions within a block should be in a DECREASING order with respect to the transaction fees.

Resources & Grading Scheme

- Skeleton code: pa1_skeleton.zip
- Demo program: pa1

The demo program is compiled in Linux (Lab 2 machines) using `main.cpp` instead of `main_test.cpp`. If all tasks have been implemented correctly, your generated program `pa1` should produce the same output as the demo program when provided with the same input.

Test cases & Grading Scheme

There are 30 given test cases available on ZINC, and can be also found in the `main_test.cpp` file. Initially, these 30 test cases are executed without any memory leak checking. Then, the same 30 test cases are run again with their original IDs incremented by 30. For example, test case #13 is repeated as test case #43.

You can find the inputs and expected outputs of the 30 given test cases in the above skeleton code zip. Your program should aim to produce the same output as the provided sample output for all given test cases above.

It's important to note that the sample output cannot cover all possible cases, so it is part of the assessment for you to design your own test cases to thoroughly test your program's functionality.

In addition to the give test cases, there will be 20 additional test cases which won't be revealed to you before the deadline. They are referred as "hidden test cases" below.

Submission & Deadline

Deadline: 12/10/2024 23:59:00 (HKT)

Please submit the following files to ZINC by **zipping** the following 4 files. ZINC usage instructions can be found here.

```
transaction.cpp
blockchain.cpp
user.cpp
bitcoin.cpp
```

Special submission notes regarding this assignment:

- When submitting your assignment to ZINC, please remember to include **only** the `transaction.cpp`, `blockchain.cpp`, `user.cpp`, and `bitcoin.cpp` files.
- While you have the freedom to modify other files to add your own test cases, it is crucial to ensure that your submitted files can successfully compile with the original versions of `transaction.h`, `blockchain.h`, `user.h`, and `bitcoin.h` on ZINC.
- It is essential **not to modify** the provided print functions. Modifying these functions may result in a zero mark for the assignment, so it is advised to refrain from making any changes to them.

General notes:

- **The submission quota for this assignment on ZINC is set to 150.**
- The compiler used on ZINC is `g++11`. However, there should not be any noticeable differences if you use other versions of the compiler to test on your local machine.
- You may submit your file multiple times, but only the last submission will be graded. **You do NOT get to choose which version we grade.** If you submit after the deadline, late penalty will be applied according to the submission time of your last submission.
- Submit early to avoid any last-minute problem. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we would grade your latest submission with all test cases after the deadline.
- In the grading report, pay attention to various errors reported. For example, **under the "make" section, if you see a red cross, click on the STDERR tab to see the compilation errors.** You must fix those before you can see any program output for the test cases below.
- Make sure you submit the correct file yourself. You can download your own file back from ZINC to verify. Again, **we only grade what you uploaded last to ZINC**.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online auto-grader ZINC. If we cannot even compile your work, it will not be graded. Therefore, for parts that you cannot finish, just put in dummy implementation so that your whole program can be compiled for ZINC to grade the other parts that you have done. Empty implementations can be like:

```
int SomeClass::SomeFunctionICannotFinishRightNow()
{
    return 0;
}

void SomeClass::SomeFunctionICannotFinishRightNowButIWantOtherPartsGraded()
{
}
```

Late Submission Policy

There will be a penalty of -1 point (out of a maximum of 100 points) for every minute you are late. For instance, since the deadline for assignment 1 is 23:59:00 on **Oct 12**, if you submit your solution at 1:00:00 on **Oct 13**, there will be a penalty of -61 points for your assignment. However, the lowest grade you may get from an assignment is zero: any negative score after the deduction due to a late penalty (and any other penalties) will be reset to zero.

FAQ

Frequently Asked Questions

Q: What are the differences between User::bitcoinBatches and UnspentBitcoinBatches::unspentBitcoinBatches?

A: They are different. User::bitcoinBatches stores the batches that the user owns; while UnspentBitcoinBatches::unspentBitcoinBatches stores the batches that the users haven't yet spent (unspent). A bitcoin batch, once added to a user by either BitcoinSystem::addBitcoinToUser or BitcoinSystem::mineBlock, becomes unspent and is added to both User::bitcoinBatches and UnspentBitcoinBatches::unspentBitcoinBatches. When a transaction is proposed by BitcoinSystem::transact, the bitcoin batches involved are considered spent and thus removed from UnspentBitcoinBatches::unspentBitcoinBatches. However, the user still owns the bitcoin batches since the transaction is pending (i.e., is proposed and yet to be confirmed), so it won't be removed from User::bitcoinBatches. By doing so, we prevent the same bitcoin batches from spending multiple times by BitcoinSystem::transact. When a transaction is confirmed by BitcoinSystem::mineBlock, the ownership of the bitcoin batches involved are transferred. Thus, they will be removed from User::bitcoinBatches. We have also updated the comments in user.h that cause ambiguity.

Q: My code doesn't work / there is an error, here is the code, can you help me fix it?

A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own and we should not finish the tasks for you. We might provide some very general hints to you, but we shall not fix the problem or debug for you.

Q: Can I add extra helper functions?

A: You may do so in the files that you are allowed to modify and submit. That implies you cannot add new member functions to any given class.

Q: Can I include additional libraries?

A: No. Everything you need is already included - there is no need for you to add any include statement (under our official environment).

Q: Can I use global variable or static variable such as "static int x"?

A: No.

Q: Can I use "auto"?

A: No.

Q: Can I use function X or class Y in this assignment?

A: In general if it is not forbidden in the description and the previous FAQs, and you can use it without including any additional library on ZINC, then you can use it. We suggest quickly testing it on ZINC (to see if a basic usage of it compiles there) before committing to using it as library inclusion requirement may differ on different environments.

Q: My program gives the correct output on my computer, but it gives a different one on ZINC. What may be the cause?

A: Usually inconsistent strange result (on different machines/platforms, or even different runs on the same machine) is due to relying on uninitialized hence garbage values, missing return statements, accessing out-of-bound array elements, improper use of dynamic memory, or relying on library functions that might be implemented differently on different platforms (such as pow() in cmath).
In this particular PA, it is probably related to misuse of dynamic memory. Good luck with bug hunting!

Change Log

- 01/10/2024: Number of test cases in the skeleton has updated from 16 to 30.
- 10/10/2024: Special Clarification: the transactions within a block should be in a DECREASING order with respect to the transaction fees.

Menu

Page maintained by

YANG, Baichen
Email: byangak@connect.ust.hk
Last Modified: 10/10/2024 13:43:47

Code contributed by

LI, Yu Hong Harry
WU, Qi

Homepage

Course Homepage