UNIVERSIDADE FEDERAL DE VIÇOSA
DEPARTAMENTO DE INFORMÁTICA
INF213 - Estrutura de Dados
Prof. Salles Magalhaes

Capicoin

Objetivos: praticar listas encadeadas, gerência de memória, criação de classes, hashing, etc.
Data limite para submissão: informada no Submitty

Imagem adaptada de http://wallyinuruguay.blogspot.com/2011/02/two-pesos.html

Observação: o conteúdo deste trabalho é relativamente simples, mas ele envolve várias técnicas e regras. Assim, é importante ler este texto várias vezes (e fazer desenhos/exemplos em papel) até se familiarizar com tudo. O texto ficou grande porque envolve alguns assuntos não vistos na disciplina (exemplo: funcionamento de criptomoedas), não porque o trabalho é difícil.

Sugiro que crie uma cópia deste doc, marque as partes que achar mais importante, adicione comentários, etc. Para o professor tirar dúvidas o aluno deverá mostrar que fez vários exemplos/rascunhos em papel.

Arquivos disponibilizados para o trabalho (note que há exemplos de entradas e das saídas correspondentes):
https://drive.google.com/file/d/1L0XNrtNVoLFh3WoizcuIP5kLAQUlbiaM/view?usp=sharing

A seguir teremos a descrição do problema e das classes que deverão ser criadas. O aluno NÃO poderá armazenar como membro de dados das classes (ou variáveis globais) nada além do especificado (exemplo: seu código não deverá armazenar em um bloco o seu hash, o número de transações nele, etc)! Funções adicionais, por outro lado, podem (e provavelmente devem!) ser criadas.

Parte 1

Introdução

Com o sucesso das criptomoedas, as capivaras da UFV decidiram lançar a sua própria (a Capicoin). Tal moeda será utilizada para várias transações na UFV. Como tais animais não são muito bons em programação, estão contando com a ajuda dos alunos de INF213 para implementar o código em C++ responsável por fazer várias operações na Capicoin.

O tema de criptomoedas/blockchains utiliza várias técnicas relacionadas a estruturas de dados: listas encadeadas (a blockchain é basicamente uma lista encadeada), hashing, árvores (bitcoin, por exemplo, armazena as transações de cada bloco dentro de uma árvore hash, ou *merkle tree*).

Na Capicoin todas transações são armazenadas em uma espécie de lista encadeada, chamada "blockchain". Cada pessoa/usuário possui um código (começando de 0) e, assim, transações são no formato 2→4 (pessoa com código 2 envia Capicoins para pessoa 4).

Obs: Como as capivaras normalmente são extremamente honestas, as várias falhas de segurança presentes na Capicoin não causam problemas. Uma criptomoeda real normalmente funcionaria de maneira um pouco diferente (mais sofisticada). Muitas simplificações foram feitas para que este trabalho não fique muito grande (e nem exija outros assuntos (que serão vistos em outras disciplinas), como assinaturas digitais). Porém, a ideia principal de uma blockchain é similar à descrita neste trabalho! Entendendo o funcionamento do Capicoin será fácil entender o funcionamento do Bitcoin, Ethereum, etc.

Hashing

Uma função hash mapeia uma chave (inteiro, struct, string, etc) em um inteiro. Por exemplo, um hash de string poderia mapear cada letra em um código (1 para "a", 2 para "b", ..) e depois somá-lo. Assim, hash("abc") seria 1+2+3 = 6.

Duas chaves diferentes podem ter hashes iguais (no exemplo acima, hash("abc")= hash("acb") = 6 -- em geral funções hashes são implementadas de modo a evitar colisões no caso de permutações da entrada, ou seja, em funções hash melhores hash("abc") não seria, necessariamente, igual a hash("acb")). Normalmente essas colisões ocorrem de forma rara (as funções são feitas tentando evitar essas colisões). Como o cálculo de um hash é determinístico → duas chaves iguais sempre possuem o mesmo hash.

Uma aplicação de hash é validar o conteúdo de dados. Por exemplo, ao transferir ou armazenar um texto podemos guardar juntamente dele o seu hash. Se você recalcular o hash desse texto e ele for diferente do originalmente armazenado → isso certamente significa que algum(s) bit do texto foi alterado! (se o hash se mantiver → isso significa que PROVAVELMENTE o texto continua original). Um algoritmo famoso para calcular hash de arquivos é o md5. Normalmente uma mudança mínima nos dados faz com que o resultado do hash fique MUITO diferente do original (exemplo: trocar um bit em um arquivo de vários GB poderia transformar o hash 123456789 em 6549876516).

Em criptomoedas, a blockchain utiliza hashing para garantir a segurança de operações. Por exemplo, suponha que haja uma transação assim:

- *(“Elon Musk transferiu 10 reais para Salles”, 12345)*, onde 12345 é o hash do texto entre aspas.

Suponha que eu falsifique a transação para:

- *(“Elon Musk transferiu 100000000000 reais para Salles”, 12345)*

É possível detectar isso recalculando o hash e vendo que agora ele é 33471 (por exemplo), ou seja, ele não irá bater com o 12345 que está armazenado junto com os dados. Porém, alguém poderia pensar: basta então falsificar a transação e trocar o hash junto, guardando:

- *(“Elon Musk transferiu 100000000000 reais para Salles”, 33471)*

Isso poderia ser feito, mas há técnicas na blockchain que impedem essa alteração de hash.

Neste trabalho vamos utilizar uma adaptação do algoritmo de hashing SHA256 (disponível nos arquivos SHA256.h e SHA256.cpp). Na classe disponibilizada, o aluno deverá passar um array de inteiros como argumento para a função “calcula” e o hash desse array será retornado como um inteiro (apenas a função “calcula” deverá ser utilizada). Originalmente, o algoritmo SHA256 calcula um hash de 256 bits. Porém, para facilitar o trabalho estamos utilizando apenas 32 desses bits, armazenando-os em um inteiro (como afirmado acima, as várias simplificações adotadas neste trabalho não deveriam ser feitas em uma aplicação real! Isso foi feito apenas por motivos didáticos).

Curiosidade: hashing é tão importante para criptomoedas que no Brasil o principal ETF (de forma simplória, um ETF é uma espécie de fundo que investe em uma variedade de criptomoedas/ações/etc e, assim, quando um investidor compra um ETF ele está comprando essa variedade de ativos indiretamente) de criptomoedas tem ticker (“nome na bolsa de valores”) hash11 (https://www.infomoney.com.br/cotacoes/b3/etf/etf-hash11/).

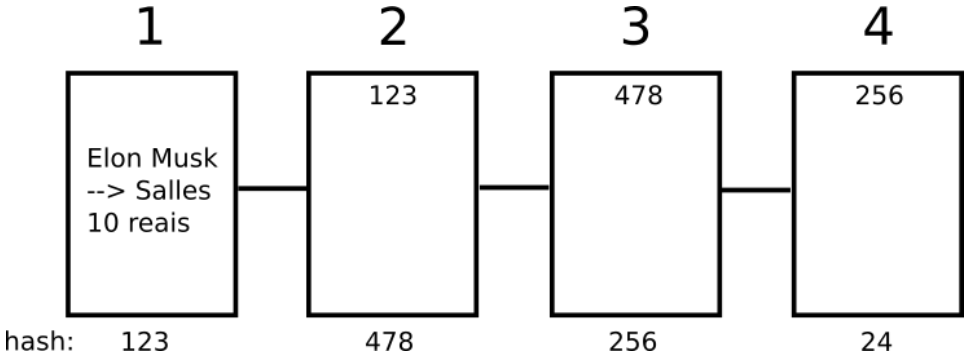
Blockchain

Uma blockchain é basicamente uma lista encadeada associada com hashing. No caso de criptomoedas, transações são armazenadas nos nodos (chamados de blocos). Cada bloco também armazena o hash do nodo anterior.

O hashing resolve um importante problema de segurança em criptomoedas: se alguém alterar algo na blockchain é possível detectar isso facilmente. Cada bloco pode ser rapidamente verificado (calculando o hashing dele e verificando se o próximo bloco contém o hashing que realmente pertence ao atual).

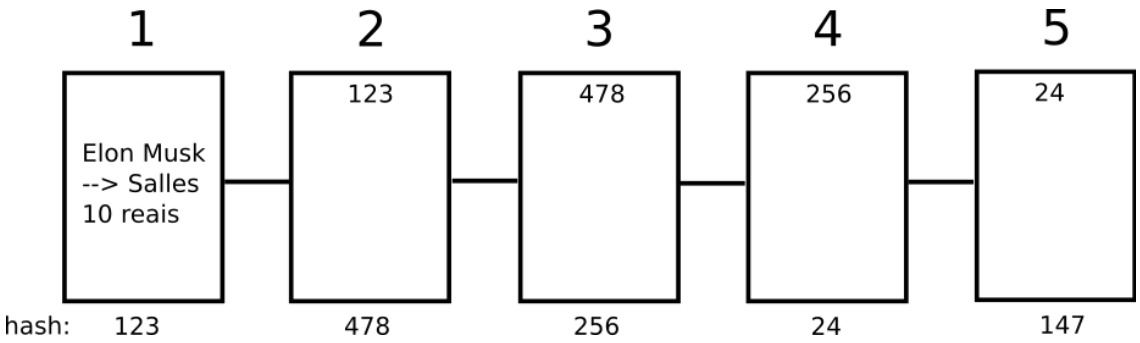
Adicionalmente, um bloco “aponta” apenas para o anterior (na nossa implementação haverá um ponteiro “next” para facilitar o percurso na lista, mas cada bloco possui o hash apenas do anterior). Com isso, ao adicionar um bloco à blockchain os dados dos anteriores não são alterados e, assim, o hash deles não muda. Dessa forma, nenhum hash da lista precisa ser recalculado para adicionar um bloco novo.

Veja o exemplo abaixo onde temos uma blockchain com 4 blocos:



Note que no topo de cada bloco há o hash do anterior. Imagine que o professor seja um hacker e troque “10 reais” por “100000000 reais” na transação do bloco 1. O hash do bloco 1 se transformará em 948 (por exemplo). Para deixar a modificação indetectável, esse hacker deveria modificar o bloco 2 trocando o 123 do topo dele para 948. Porém, há dois problemas: TUDO do bloco é utilizado no cálculo do seu hash (exceto os ponteiros) e, com isso, ao trocar 123 no bloco 2 para 948 → o hash do bloco 2 (478) será modificado. Isso exigirá modificar o bloco 3 também e assim sucessivamente! O segundo problema é que trocar o hash de um único bloco é um processo computacionalmente caro (você verá depois o motivo disso). Assim, a alteração em um nodo exige correções em vários nodos e cada uma dessas correções é MUITO cara computacionalmente. Isso (e outros recursos) garante a segurança da blockchain!

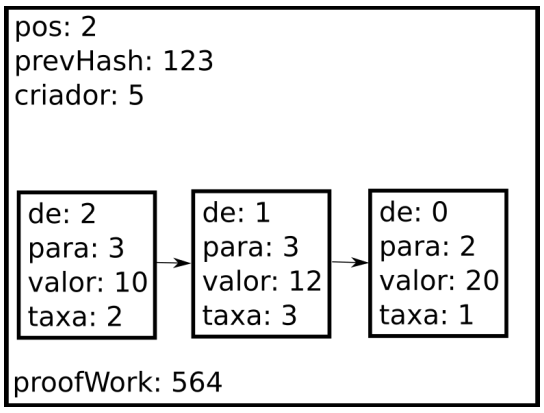
Vimos que modificar um bloco (e corrigir a validade da lista toda) é um processo caro. Por outro lado, como cada bloco armazena dados (hash) apenas do bloco anterior, adicionar um novo bloco ao final da blockchain é um processo relativamente barato (não é preciso recalcular nada nos blocos anteriores, já que eles não serão modificados por isso). Veja o exemplo abaixo, onde o bloco 5 foi adicionado (bastou armazenar no topo dele o hash do bloco 4).


1
2
3
4
5
Elon Musk
--> Salles
10 reais
123
478
256
24
hash:
123
478
256
24
147

Blocos

Vamos nos concentrar agora em um bloco da blockchain. Na Capicoin, cada bloco armazenará 0 ou mais transações utilizando uma lista simplesmente encadeada em seu interior (no caso do Bitcoin, por exemplo, cada bloco armazena 1MB de dados utilizando uma estrutura chamada *merkle tree*).

Veja, abaixo, um exemplo de bloco (armazenando 3 transações):



Para facilitar o trabalho, todos valores armazenados em um bloco serão inteiros do tipo “int”. Segue a descrição dos elementos que deverão ser armazenados em um bloco (seu programa deverá ter uma classe **Block** para isso):
- pos: posição do bloco na blockchain (o primeiro bloco deverá ser 1, o segundo 2, …)
- prevHash: hash do bloco anterior na blockchain (será 0 no caso do bloco 1).
- criador: código do “usuário” que criou o bloco (que o minerou)
- Lista encadeada de transacoes: cada bloco deverá armazenar uma lista simplesmente encadeada de transações (crie uma classe **Transaction**, que representará os nodos das transações e deverá armazenar os 4 inteiros e um ponteiro para a próxima transacao). Sendo assim, o bloco deverá possuir dois ponteiros: um para o início e outro para o fim dessa lista (tais ponteiros não aparecem na figura acima).
- proofWork: um inteiro, que será explicado posteriormente.

Adicionalmente, cada bloco precisa de dois ponteiros: um apontando para o bloco seguinte e outro para o anterior (pois eles formam uma lista duplamente encadeada).

O hash de um bloco poderá ser calculado passando para a função “calcula” da classe SHA256 um array com todos inteiros (na mesma ordem em que são descritos acima), incluindo os inteiros das transações (na mesma ordem em que as transações aparecem). Obviamente, os ponteiros não são utilizados no hash. No exemplo da figura acima, o hash de [2,123,5,2,3,10,2,1,3,12,3,0,2,20,1,564] é -232715264 (ou seja, se tal bloco estivesse em uma blockchain o próximo deveria armazenar em prevHash -232715264).

Note que um bloco não deverá armazenar seu hash (mas ele poderá ser facilmente recalculado, se necessário).

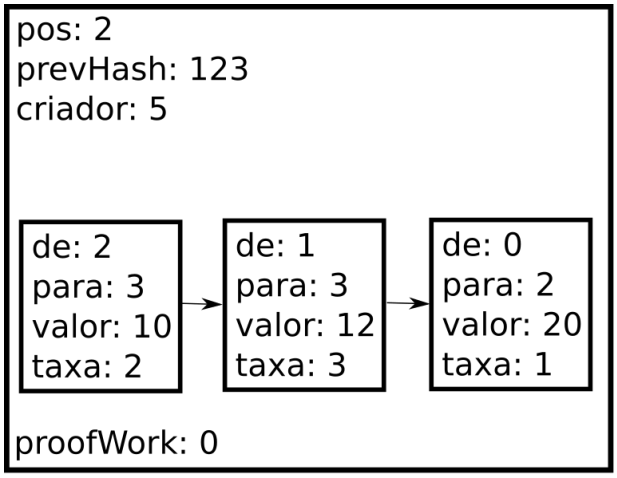
A classe Block deverá ter, pelo menos, os seguintes métodos públicos (pode, e provavelmente vai, ter outros):
- Construtor que espera como argumentos, respectivamente: posição, prevHash, criador
- Construtor que espera como argumentos o mesmo do anterior e proofWork (como último argumento).
- Método addTransaction(a,b,valor,taxa): adiciona uma transação ao fim da lista encadeada de transações. Essa operação representa o envio de "*valor*" Capicoins do usuário "*a"* para o "*b"*, pagando uma taxa de mineração de "*taxa*" (por enquanto não se preocupe com o significado dessa taxa). Nenhum valor/taxa/código de usuário poderá ser negativo. A complexidade de tal método deverá ser O(1). Esse método deverá ter exatamente o nome "addTransaction".
- Três métodos que serão explicados em "Mineração" (a seguir).

Mineração

O processo de criar um bloco (*minerar*) normalmente é dificultado de forma artificial. Isso é feito por motivos de segurança (como já falado anteriormente, mudar o hash de um bloco (o que envolve recriá-lo) deve ser caro computacionalmente) e também para criar uma noção de "valor financeiro do trabalho" (como será falado posteriormente, as moedas são criadas na mineração e, assim, se fosse fácil criá-las elas não teriam valor).

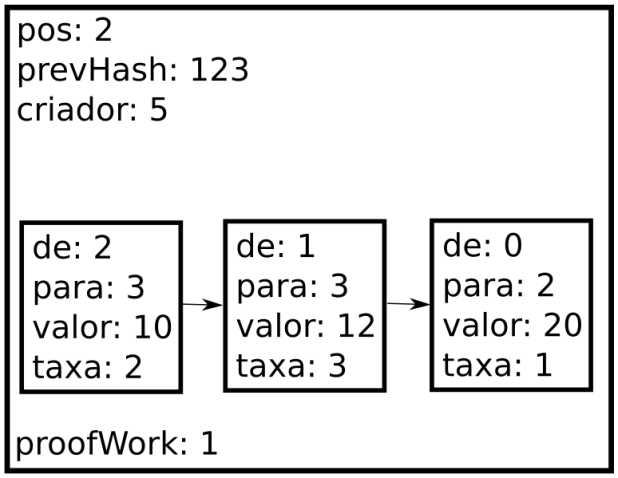
Em um bloco pos, prevHash, criador e as transações são valores fixos (precisamos que os valores deles sejam exatamente os informados pelo usuário). O valor de proofWork, por outro lado, é gerado pelo "minerador" no processo de criação de um bloco.

Considere o bloco do exemplo abaixo: o hash dele é -1012629999 caso proofWork seja 0 (canto superior esquerdo). Mudando o "proofWork" de 0 para 1 o hash muda para -1958065598. Como os hashes mudam de forma aproximadamente aleatória cada vez que algo é alterado (veja como a representação binária do hash muda com uma pequena mudança em proofWork), podemos observar o seguinte: é preciso testar aproximadamente 4 valores diferentes (exemplo: 0,1,2, …) de proofWork até obtermos um hash cujos 3 bits menos significativos sejam todos 0, pois a cada 8 (2^3) números consecutivos temos um satisfazendo essa característica.



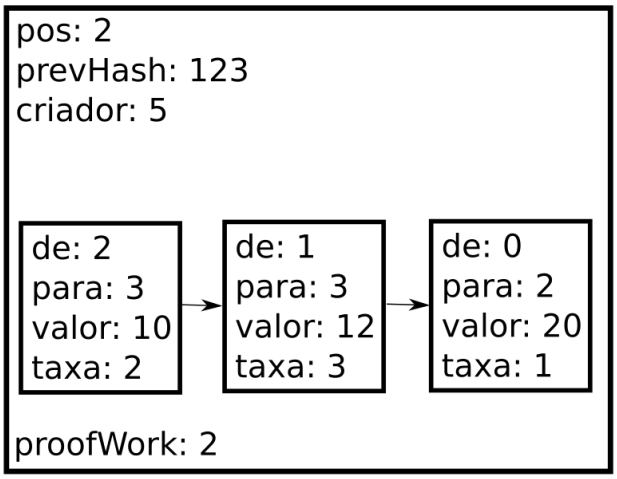
Hash: -1012629999

Hash(bin): 11000011101001000111111000010001



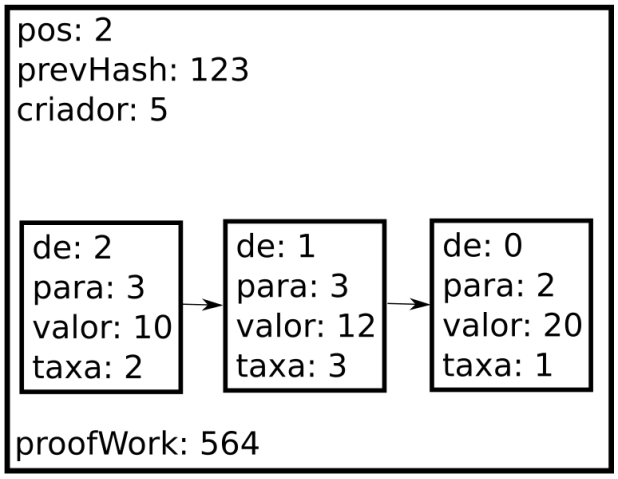
Hash: -1958065598

Hash(bin): 10001011010010100100101001000010



Hash: 1030190988

Hash(bin): 00111101011001110111011110001100



Hash: -232715264

Hash(bin): 11110010001000010000110000000000

Uma forma de dificultar o processo de criação de um bloco é adicionar uma regra como a seguinte: o criador do bloco deverá testar valores de proofWork até que o hash desse bloco obedeça a alguma restrição (exemplo: seja divisível por 10, tenha os 20 bits menos significativos iguais a 0, etc).

Neste trabalho, vamos usar a seguinte regra: você deverá testar todos valores possíveis para proofWork (começando do 0) até que isso gere um hash onde os 10 bits menos significativos sejam 0. Com isso, espera-se que seja necessário calcular em torno de 500 hashes até encontrar um proofWork que satisfaça essa regra. No exemplo acima, o primeiro valor de proofWork que satisfaz isso é 564 (veja a representação binária do hash dele). Apenas blocos satisfazendo essa restrição são considerados blocos válidos (blocos “minerados”).

Observe que qualquer proofWork que satisfaça a regra acima poderia gerar um bloco válido. Porém, neste trabalho você deverá utilizar o primeiro número (começando de 0) que satisfaz.

Isso será escolhido para que os hashes gerados pelo seu programa sejam os mesmos que eu gerei no meu (e, assim, o submitty conseguirá verificar mais facilmente a corretude do seu trabalho).

Para efeito de comparação, no Bitcoin atualmente (2022) os mineradores precisam calcular 30 trilhões de hashes (em média) até encontrar um "proofWork" que faça com que o hash satisfaça a regra da rede. Nesse processo de mineração muitas pessoas juntam poder computacional (em um *pool*) para tentar encontrar rapidamente o "proofWork". Atualmente gasta-se em torno de 10 minutos para encontrar um "proofWork" e, assim, criar/minerar um bloco (se fosse usado o poder computacional de apenas um computador isso seria MUITO mais demorado).

Com isso, quanto maior o poder computacional de uma pessoa → mais ela pode contribuir para o pool → mais dinheiro ela recebe pela sua contribuição. Uma GPU Geforce 1070, por exemplo, consegue testar em torno de 24 milhões de hash por segundo usando o algoritmo Ethereum e, com isso, gerar aproximadamente 2 dólares de moedas por dia (valores de maio/2022). (https://whattomine.com/)

Sua classe Block deverá ter 3 métodos relacionados à mineração:
- Um método para minerar o bloco: testa todos proofWork a partir de 0 até encontrar um que faça com que a hash do bloco tenha os 10 primeiros bits sendo 0. Após minerar, essa função deverá armazenar na variável "proofWork" do objeto para o valor encontrado.
- Um método para calcular (e retornar) o hash desse bloco. Tal método deverá obrigatoriamente se chamar *getHash*().
- Um método que verifica se proofWork está válido para esse bloco: isso acontecerá se o hash do bloco possuir os 10 primeiros bits sendo 0. Ou seja, esse método verificará se o bloco foi "minerado".

Parte 1 do trabalho

Neste trabalho, você deverá criar um programa (em main.cpp) que lê a partir da entrada padrão uma série de comandos. Na primeira etapa faremos dois tipos de operação: "validarBloco" e "minerarBloco". Sugiro que faça essa primeira parte sem se preocupar com a próxima (e teste-a bastante). Quando ela estiver pronta, mova para a parte 2. (quem enviar apenas a parte 1 poderá ganhar pontos parciais)

Esta primeira etapa poderia ser facilmente implementada sem a criação das classes aqui especificadas (o aluno poderia simplesmente guardar todos os inteiros em um array grande e fazer as operações nele). Porém, isso não será permitido no trabalho.

Por exemplo, se seu programa for executado assim "./a.out < exemplosEntrada/exemplo_valida_bloco_1.txt > saida.txt " o arquivo saida.txt deverá ser idêntico ao arquivo "exemplosSaida/exemplo_valida_bloco_1.txt "

validarBloco:

Caso a entrada comece com a string “validarBloco”, você deverá ler dados de um bloco, criar esse bloco e, então, verificar se ele é válido.

A segunda linha da entrada terá 5 inteiros: pos, prevHash, criador e proofWork. A seguir, há um número N. Por fim, há N linhas cada uma descrevendo uma transação (com 4 inteiros: origem, destino, valor e taxa).

Como saída, seu programa deverá imprimir três linhas contendo: o hash do bloco, o hash em binário e, por fim, uma string dizendo se o bloco foi minerado ou não.

Veja os exemplos de entrada e saída: exemplo_valida_bloco_1.txt, exemplo_valida_bloco_2.txt e exemplo_valida_bloco_3.txt

minerarBloco:
Caso a entrada comece com a string “minerarBloco”, você deverá ler uma string indicando se o processo de mineração deverá ser “verbose” (tal string será “quiet” ou “verbose”) e os dados de um bloco (exceto o proofWork). A seguir, você deverá minerá-lo para descobrir o proofWork.

A segunda linha da entrada terá 4 inteiros: pos, prevHash e criador. A seguir, há um número N. Por fim, há N linhas cada uma descrevendo uma transação (de forma similar a em validarBloco).

Como saída, seu programa deverá imprimir três linhas contendo: proofWork, o hash do bloco e o hash em binário.

Caso o modo seja “verbose”, antes de imprimir a resposta você deverá imprimir 3 linhas para cada proofWork testado: o array de inteiros passado para a função hash (primeira linha), o código hash em binário e uma linha em branco.

Veja os exemplos em exemplo_minera_bloco_1.txt e exemplo_minera_bloco_1_verbose.txt

Parte 2

Capicoin

A Blockchain armazena as transações da Capicoin. Porém, até agora não vimos sobre como as moedas são “criadas”.

Inicialmente (antes do primeiro bloco ser minerado), não existe moeda! A ideia é que os mineradores “acham” moedas no processo de mineração. Isso é simulado da seguinte forma: cada bloco possui uma recompensa em Capicoins e o criador do bloco recebe essa recompensa ao minerar (por isso armazenamos o inteiro “criador”).

Para criar uma “escassez”, moedas muitas vezes são projetadas para não haver inflação (causada por uma “impressão de dinheiro”). Assim, a ideia é que com o tempo a recompensa dos blocos caia. Na Capicoin, vamos supor que o primeiro bloco a ser minerado pagará 256 Capicoins para o criador. Cada bloco seguinte pagará metade do anterior (isso é conhecido como *halving*). No Bitcoin, por exemplo, o primeiro bloco minerado em 2009 pagava 50 BTC e o halving ocorre a cada 210000 blocos (ou seja, na Capicoin essa queda é muito acelerada!) e, assim, existirá no máximo 21 milhões de BTC (após um tempo a recompensa tenderá a zero). Como um bloco é minerado aproximadamente a cada 10 minutos, o halving do Bitcoin ocorre aproximadamente a cada 4 anos.

Pergunta: quantas Capicoins existirão no mundo? (supondo que aceitamos apenas valores inteiros, o primeiro bloco recompensa 256 Capicoins e esse valor cai pela metade a cada novo bloco). **Responda isso no seu README, justificando.**

Com recompensa cada vez menor, há uma tendência de minerar ser cada vez menos lucrativo. Por outro lado, pode ser que essa escassez faça com que a moeda passe a valer mais e, assim, o processo volte a ficar lucrativo (o minerador vai receber menos moedas, mas elas valerão mais). Várias características do mercado influenciam a lucratividade da mineração (recompensa, custo de hardware/energia, preço da moeda, confiança da população no mercado, etc).

Note que no primeiro bloco nenhum usuário recebeu moedas ainda (a primeira moeda será gerada apenas após o primeiro bloco ser minerado). Assim, na Capicoin o bloco 1 nunca terá transações (poderia até ter, mas todas com valores e taxas zerados!).

Há dois problemas:
1) por que alguém se interessaria em minerar quando a recompensa chegar a 0? (isso seria um problema, pois se não houver mineração não será possível fazer transações com a moeda)
2) A blockchain armazena transações nos blocos. Cada bloco possui um limite de transações e demora um certo tempo para ser minerado. O que acontece se tivermos mais transações sendo criadas do que esse limite? O que impede os usuários de criarem uma quantidade excessiva de transações?

Esses problemas são resolvidos com a taxa da transação. Ao criar uma transação, a pessoa enviando dinheiro deve informar quanto deseja pagar de taxa. Essa taxa será paga para o minerador (ou seja, ele vai receber a recompensa do bloco + a taxa de todas transações que ele adicionou ao novo bloco). Sendo assim, normalmente os mineradores escolhem as transações mais lucrativas.

Com isso, se você for enviar dinheiro para alguém e escolher uma taxa muito baixa → corre o risco de sua transação não entrar na blockchain (não ser efetivada). Se isso acontecer, você deverá: 1) desistir 2) tentar novamente (talvez na próxima tentativa a concorrência estará menor…) 3) tentar novamente, pagando uma taxa maior.

Operações da parte 2

Na segunda parte você deverá implementar uma classe **Blockchain** e suportar outras operações (além das da parte 1) relacionadas a transações com criptomoedas. A classe Blockchain deverá ter um construtor (sem argumentos) e armazenar apenas dois apontadores: um para o primeiro e um para seu último bloco (objetos do tipo “Block”).

Testes desta segunda parte terão na primeira linha a string “operacoes”. A seguir, haverá uma série de operações (terminando no final do arquivo). Se um caso de teste tiver a string “operacoes” não haverá nenhum teste referente `a parte 1 do trabalho nele.

criarBloco:
Caso uma linha comece com “criarBloco”, você deverá chamar um método da classe Blockchain para criar/minerar um novo bloco, passando para ele os detalhes das transações. A linha seguinte terá 3 inteiros: o número de transações que desejam entrar no bloco, número máximo (MX) de transações que caberão no bloco (em criptomoedas normalmente isso é fixo, mas aqui cada bloco poderá ter uma quantidade diferente de transações) e código da pessoa que irá minerar tal bloco (em criptomoedas reais normalmente várias pessoas tentam minerar simultaneamente e a primeira a resolver o “quebra-cabeça” fica sendo a criadora).

A seguir, haverá uma linha para cada transação (cada uma composta por 4 inteiros). Na main do seu programa você deverá criar um array de structs (cada struct com 4 inteiros) representando as transações lidas na entrada. A seguir, você deverá chamar um método da classe Blockchain para que um novo bloco seja criado. Esse método deverá:
1) escolher as MX transações mais lucrativas (se houver algum empate → a transação que apareceu primeiro na entrada tem preferência).
2) criar um objeto do tipo Bock com tais transações (varrendo o array e chamando a função addTransaction do bloco que, por sua vez, criará a lista encadeada de transações).
3) minerar o bloco e adicioná-lo ao final da blockchain.

imprimeBlockchain:
Caso uma linha comece com “imprimeBlockchain”, você deverá imprimir a blockchain inteira.

Veja o modelo de entrada/saída em “exemplo_cria_bloco_1.txt”. Nesse modelo são criados 3 blocos e a blockchain é impressa 3 vezes. Note que os separadores “=” e “-” (usados para melhorar a visualização) têm sempre largura fixa. Por exemplo, antes de imprimir a blockchain você deverá imprimir 21 “=” .

imprimeSaldo:
Caso uma linha comece com “imprimeSaldo”, na frente dessa string haverá um inteiro B (maior ou igual a 1 e menor ou igual ao número de blocos). Você deverá, então, imprimir o saldo de cada um dos usuários após o bloco B. Seu programa deverá percorrer a blockchain e calcular tudo (“simular” todas transações), ou seja, os saldos não deverão estar previamente armazenados em nenhum lugar (exceto em variáveis locais temporárias da função de calcular saldo).

Note que os usuários são representados por inteiros (começando de 0) e não há limite para esses números. Se nas transações (armazenadas na blockchain) de todos os blocos até o B o usuário de maior código que já apareceu é o U, então você deverá imprimir o saldo de todos os usuários entre 0 e U (veja o exemplo), mesmo se acontecer do saldo de U eventualmente ficar zerado.

Para facilitar a implementação, nesta etapa os alunos poderão utilizar um objeto do tipo MyVec para calcular os saldos (os alunos estão autorizados a usarem MyVec apenas na função da classe Blockchain responsável por imprimir o saldo).

Veja o exemplo em “exemplo_imprime_saldo_1.txt”. Note que após o bloco 1 apenas o usuário 5 terá saldo (ele receberá 256 Capicoins por minerar o primeiro bloco). Após o bloco 2, o usuário 1 receberá 128 de recompensa e 5 de taxas de transação (ficando com 133 de saldo). Os usuários 3 e 4 receberão, respectivamente, 10 e 11 de 5. O usuário 5 perderá os 10 e 11 que foram transferidos e mais as taxas para as duas transações efetivadas. Observe que o usuário 7 aparece apenas no bloco 5 (dessa forma, apenas após esse bloco a impressão terminará nele).

imprimeTransacoes:
Se uma linha começar com “imprimeTransacoes”, você deverá imprimir todas as transações armazenadas na blockchain.

A impressão deverá ser feita utilizando um iterador da classe Blockchain que percorre todas as transações. Essa parte do trabalho será fornecida no arquivo main.cpp provido juntamente com o material deste trabalho. Sua tarefa será implementar os iteradores e o método “imprime” da sua classe Transaction.

Veja o exemplo em “exemplo_imprime_transacoes_1.txt”.

alteraTransacao:
Se uma linha começar com “alteraTransacao”, na frente dessa string haverá 6 inteiros representando, respectivamente, a posição B de um bloco, a posição T (contando a partir de 1) de uma transação dentro de um bloco, origem da transação, destino, valor e taxa.

Seu programa deverá, então, alterar a transação T do bloco B para os valores passados como argumento. A seguir, o bloco deverá ser reminerado (para atualizar seu proofWork). Observe que, com isso, a blockchain será invalidada (pois o bloco seguinte não terá o hash correto do bloco atual). O objetivo desta operação é justamente mostrar que esse problema ocorre.

Veja um exemplo em “exemplo_altera_transacao_1.txt”.

Mais informações sobre a blockchain

Dadas duas blockchains válidas (onde cada bloco contém o hash correto do bloco anterior), é possível ver se as duas são iguais de forma muito eficiente: basta comparar o hash do último bloco das duas. Por que isso “garante” a igualdade (responda no seu README)? (tecnicamente não garante 100%, mas é EXTREMAMENTE improvável que as duas sejam diferentes)

Há várias questões importantes de segurança não tratadas neste trabalho. Por exemplo, qualquer pessoa pode pedir para adicionar uma transação (o que me impede de adicionar uma transação do tipo “Elon Musk transfere 99999999 para Salles” ?). Isso é resolvido com assinaturas digitais (tema de outra disciplina), onde a pessoa dona dos fundos a serem transferidos “assina” uma transação para comprovar que ela é realmente a pessoa querendo criar a transação (ou seja, nesse exemplo apenas Elon Musk (ou alguém que tem as suas chaves privadas) conseguiria criar a transação assinada).

Outro problema (também resolvido facilmente com assinaturas digitais) é: se uma transação assinada é algo “digital”, por que eu não posso simplesmente copiá-la e duplicá-la? Exemplo: se tem uma transação assinada do tipo “Elon Musk transfere 1 para Salles”, o que me impede de fazer várias cópias dessa transação já assinada e pedir a um criador de bloco para adicioná-las no próximo bloco? (de modo a receber a transferência múltiplas vezes)

Uma blockchain real apresenta vários outros recursos/características. Por exemplo, normalmente a blockchain é armazenada de forma distribuída (todos usuários possuem uma cópia dela). Se mais de um usuário minerar um novo bloco (gerando blockchains diferentes), qual será o “correto” ? (há um “algoritmo de consenso” que resolve isso).

Dessa forma, conforme informado acima, a implementação deste trabalho ainda apresenta algumas falhas de segurança.

O que poderá ser utilizado

A princípio, você poderá utilizar apenas as bibliotecas string, iostream, cassert (recomendo que use asserts para detectar erros) e algorithm (o aluno poderá utilizar algum método de ordenação pronto do C++) . Se achar que outra biblioteca é realmente necessária, pergunte ao professor se o uso será permitido.

Vectors, listas, maps (e similares -- mesmo o que foi implementado pelo professor em sala) não deverão ser usados, ou seja, tudo deverá ser implementado pelo aluno (a exceção é o uso do MyVec na função de calcular saldo). A ideia será praticar alocação dinâmica de memória (para os arrays), gerência de lista encadeada, etc.

Dicas

→ Sempre faça vários exemplos e diagramas em papel. ←

- Crie casos de teste simples para validar sua implementação e entender melhor o problema. Isso é uma importante habilidade de computação. Seu programa será testado com casos mais desafiadores (escondidos no submitty) -- é sua tarefa pensar em mais casos e garantir que seu programa funcione (lembre-se que um cliente provavelmente não lhe dará casos de teste prontos em um projeto). Exemplos incluem casos de teste verificando a gerência de memória (sugiro que use o Valgrind para ajudá-lo a detectar erros).
- Algumas dessas classes (exemplo: Transaction) não deveriam ter setters, principalmente públicos (deveríamos criá-las e nunca mais permitimos modificações!). Porém, como vamos testar o que acontece caso alguém tente modificar a blockchain (por motivos didáticos), pode ser necessário criar alguns métodos “set”.
- Leia novamente o que está escrito em vermelho acima.

Organização do código

Seu código deverá ser bem organizado, com comentários e seguindo boas práticas de programação.

Cada classe deverá estar definida em arquivos separados (a definição em um arquivo .h e a implementação em um .cpp).

Arquivo README

Seu trabalho deverá incluir um arquivo README.

Tal arquivo conterá (nessa ordem):

- Seu nome/matricula
- Informações sobre todas fontes de consulta utilizadas no trabalho.
- Respostas para a(s) pergunta(s) feita(s) neste documento.

Submissao

Submeta seu trabalho (main.cpp, Block.h, Block.cpp, Transaction.h, Transaction.cpp, Blockchain.cpp, Blockchain.h, README.txt) utilizando o sistema Submitty até a data limite. Seu

programa será avaliado de forma automática (os resultados precisam estar corretos, o programa não pode ter erros de memória, etc), passará por testes automáticos “escondidos” e a qualidade do seu código será avaliada de forma manual.

A avaliação automática será feita em partes. Assim, um programa incompleto ou que não segue completamente a especificação poderá obter uma nota parcial.

Seu programa será compilado com o comando “g++ main.cpp SHA256.cpp Block.cpp Blockchain.cpp Transaction.cpp -std=c++17 -O3” (o std=c++17 é necessário devido ao código do SHA256) e testado no Linux. Como na primeira etapa a classe Blockchain não precisará ser implementada, então nela o aluno poderá deixar o arquivo Blockchain.cpp/h vazio (mas ele deverá ser enviado)

Duvidas

Dúvidas sobre este trabalho deverão ser postadas no PVANET Moodle. Se esforce para implementá-lo e não hesite em postar suas dúvidas!

Testes automáticos

O trabalho será avaliado automaticamente. Na etapa 1 os alunos poderão ver as notas de todos os testes.

Porém, na etapa 2 alguns testes estarão escondidos (o aluno só saberá a nota após o deadline). Isso é um incentivo para cada aluno criar seus próprios casos de teste! Na vida real é extremamente importante pensar nas situações onde seu código pode falhar (e criar testes).

A nota do trabalho será dividida igualmente em 3 partes:
1) Nota dos testes automáticos da etapa 1
2) Nota dos testes automáticos da etapa 2
3) Nota da avaliação manual das duas etapas

Com isso, se o submitty indicar que os testes de uma das etapas valem 16 pontos e um aluno receber 15 pontos nesses testes → esse aluno terá obtido 93.8% de 1/3 da nota do trabalho (os outros 2/3 virão da próxima etapa e da avaliação manual)

Obs: mesmo na etapa 1 o Submitty não deve ser utilizado para “debugar” seu programa! (há inclusive uma dedução de pontos (feita de forma automática) caso o aluno faça um número excessivo de envios). Sempre crie seus próprios testes e avalie seu programa no seu computador.

Avaliacao

Principais itens que serão avaliados (além dos avaliados nos testes automáticos):
- Comentarios
- Aderência à especificação
- Indentacao

- Eficiência
- Nomes adequados para variáveis (exemplo: a variável “numPessoas” tem o nome muito mais mnemônico que a variável “x”)
- Separação do código em funções lógicas
- Uso correto de const/referência
- Gerencia correta de memoria.
- Uso de variáveis globais apenas quando absolutamente necessário e justificável (uso de variáveis globais, em geral, é uma má prática de programação).
- Organização do código em classes
- etc

Os pontos dados automaticamente pelo Submitty poderão ser zerados posteriormente caso o aluno não implemente o código conforme especificado (exemplo: não crie a lista encadeada e as classes). Ou seja, a nota dos testes automáticos é garantida apenas se o aluno seguir esta especificação.

Regras sobre plágio e trabalho em equipe

Leia as regras gerais aqui:
https://docs.google.com/document/d/1qwuZtdioZO-QiDsq6SAm7m-DU6bLrid7_7nEL4g9HOk/edit?usp=sharing