Problem Set 6

Due before midnight on Monday, December 3, 2018.

1 Introduction

This problem set continues the development begun in Problem Set 4 of a simulator for a population of simple agents attempting to reach consensus on a choice value. The long-rang goal of this and following assignment(s) is to simulate the blockchain consensus algorithm used in Bitcoin cryptocurrency, sometimes called *Nakamoto consensus*, to see how fast consensus is reached under various assumptions about the speed and reliability of the underlying network and the honesty of the agents participating in the protocol.

2 Blockchain Background

While not strictly needed for this assignment, a general understanding of blockchains and Nakamoto consensus will help motivate it.

A **blockchain** is a sequence of records or *blocks* that are cryptographically protected to preserve integrity and prevent various kinds of tampering. A blockchain can be extended only by someone who knows the solution to a difficult cryptographic puzzle that is derived from the current blockchain. An agent, called a *miner*, who wants to extend the blockchain first has to solve the puzzle for that chain. In general, many miners are working in parallel to solve the puzzle. Any one who succeeds is able to create a new block of *transactions* and append it to the end of the chain. The result is a new longer chain that is verifiably valid.

Once a new chain has been produced, the successful miner sends it around the network to other miners. When a longer (valid) chain is received from another miner, the recipient discards the old shorter chain and begins trying to solve the puzzle for the longer chain. Consensus on the new chain is reached when all of the miners have received the new chain and discarded their old, shorter, chains.

It is possible that two miners will solve the puzzle at nearly the same time and will each propose longer but different extensions of the current chain. These new chains will propagate around the network. Because they are equal length, neither will annihilate the other, so consensus cannot be reached until a yet longer chain is produced by one of the minors. As this new chain propagates through the network, miners will discard their old chains and adopt the new one.

Intuitively, consensus will eventually be reached if there is a unique longest chain in circulation, and no new chains are created before consensus is reached. The purpose of the puzzles (also called *proof of work* is to slow down the process of creating new chains, which in turn will decrease the likelihood of new chains interfering with the consensus process.

In practice, one does not require that consensus on what the current blockchain is. Rather, one is interested when consensus is reached on a particular block. For example, suppose a miner extends a length 10 blockchain c to create a new blockchain whose last (11^{th}) block is b. Intuitively, b is *committed* if b is the 11^{th} block of *every* longer blockchain

still in circulation. Of course, there is always the possibility that some miner still working on the old chain c will succeed in creating a new length-11 chain with a different 11^{th} block. However, the chance of that block not being annihilated as it attempts to infect other miners is vanishing small under suitable circumstances.

3 Problem

This assignment focuses on a particular space-efficient representation of blockchains. At an abstract level, a blockchain is just a sequence of blocks. A new blockchain is created by appending a new block to the end of an existing blockchain. The initial blockchain consists of a single **genesis block**. All subsequent blockchains begin with the genesis block.

We ignore the cryptography issues of real blockchains and assume that the only properties of interest in a blockchain are its length and the list of blocks that comprise the chain. Each block has an associated unique identifier, so there is never the possibility of two identical blocks being created in different parts of the system.

Figure 1 shows a situation with three active blockchains beginning with blocks Bk2, Bk3, and Bk4, respectively. Four agents ChA, ..., ChD have three different current choices for the blockchain. ChA prefers the chain beginning with Bk2, ChB and ChC both prefer the chain at Bk4, and ChD prefers the chain at Bk3.

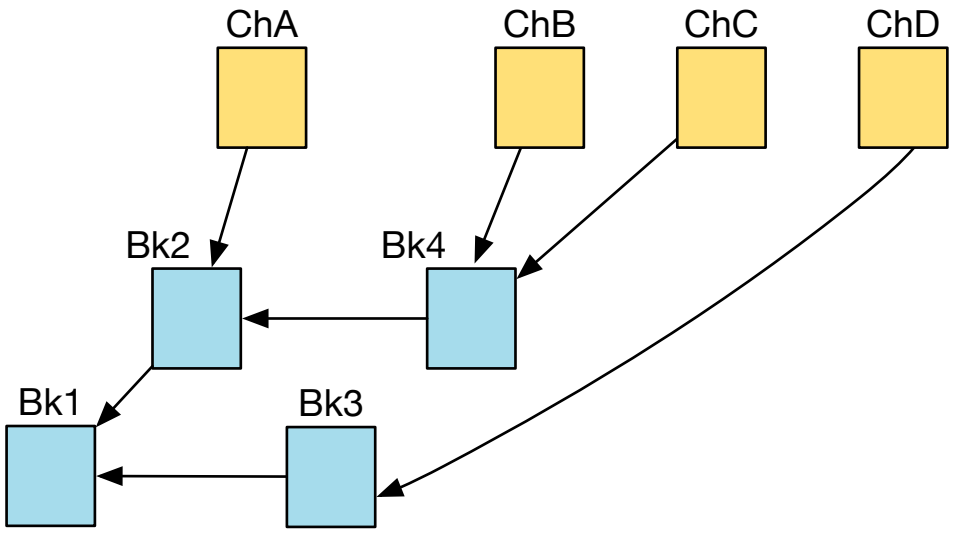

Figure 1: Three active blockchains.

Our blockchain representation has two goals:

1. No block is ever copied.
2. A block is automatically deleted as soon as it becomes inaccessible.

For goal 1, we delete the copy constructor and copy assignment (to prevent accidental copying), and we use pointers to represent the chain structure as a kind of linked list.

For goal 2, we replace the arrows of Figure 1 with a slight modification of the `SPtr` class presented in demo 21a-SmartPointer-v2. Figure 2 shows the smart pointers as white rectangles inside both the blockchain headers (possessed by the agents) and inside the blocks

themselves. The dashed white boxes represent the `count` dynamic extensions in the `SPtr` class. Recall that this is a count of the number of `SPtr` objects having the same `target` pointer.

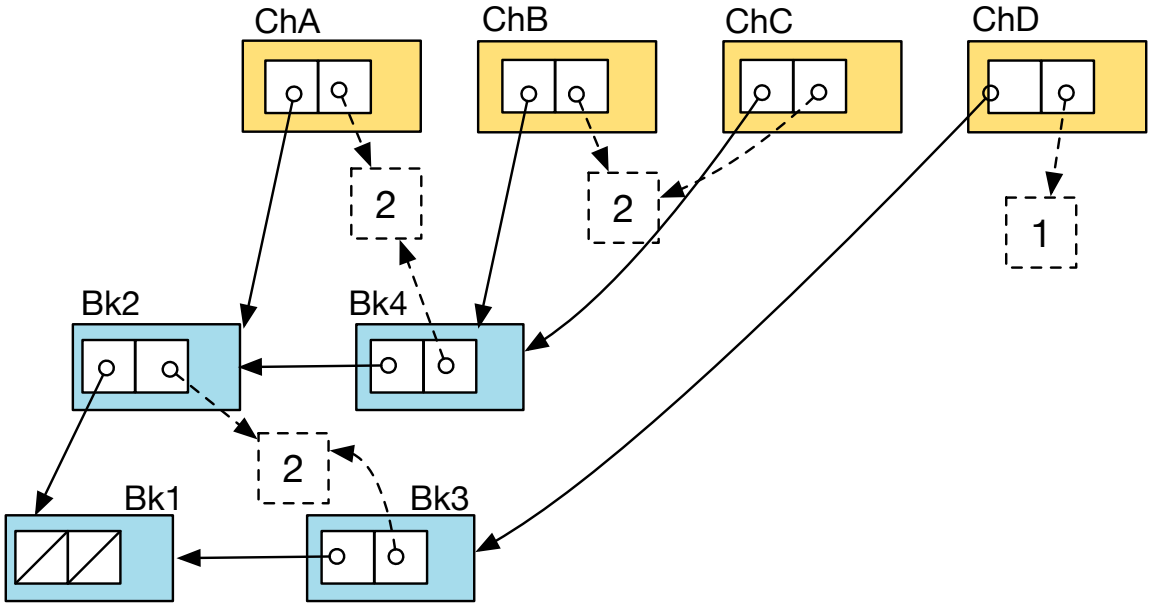

Figure 2: Three active blockchains.

Your job is to implement two new classes, `Blockchain` and `Block`, to represent blockchains. In addition to the `SPtr` data member, each `Block` will also have two `const` fields, a unique identifier and it's level in the blockchain tree, where the genesis block is considered to be at level 0. (The genesis block in Figures 1 and 2 is `Bk1`.)

Class `Blockchain` contains a single private data member of type `SPtr`, which implements the smart pointer to the most recent block in the chain. It should have several public functions:

1. `Blockchain extend()` returns a new blockchain created by extending the current blockchain. The new chain should be stack-allocated and returned by value.
2. `print()` prints the blocks that comprise a blockchain in order of increasing level. For example, the output from printing the blockchain ChC in Figure 2 might look like `[0,1] [1,2] [2,4]`. Here, the first number of each pair is the block's level in the tree and the second number is its UID.
3. `operator<<()` is the usual inline extension of `print()`.

Other functions that might prove useful include `unsigned length()`, `bool operator==()`, and `block* tail()` (which returns a regular pointer to the last (most recent) block).

Done with tail

Class `Block` should have three private data members: `serialNo`, `level`, and `SPtr`. All three should have the `const` type qualifier, meaning that they cannot be changed after initialization. The copy constructor and copy assignment should also be deleted since blocks are never supposed to be copied. It should also have a public function `blkLevel()` that returns the level of the current block.

Finally, a driver program will need to be written along the lines of class `Game` in problem set 3 (Think-a-Dot). The driver should create an array `bc` of 10 blockchains. Each blockchain should be initialized with a copy of the length-0 blockchain that contains only the level-0 genesis block.

The driver should support four one-letter commands, some of which take single-digit arguments.

`Ajk` does the assignment `bc[j] = bc[k]`.

`Ej` extends blockchain `bc[j]`.

`P` prints the blockchains in array `bc[]`, one per line.

`Q` quits.

4 Teaching Objectives

- Make use of a slightly expanded version of the smart pointer class `SPtr` given in class demo 21a-SmartPointer-v2.
- Make use of class `Serial` from the same demo to assign unique ID's to new blocks.
- Learn how to use smart pointers of class `SPtr` to manage the job of deleting blocks when they no longer needed.

5 Programming Notes

1. When running your code under valgrind, expect an output line

   ```
   in use at exit:  4 bytes in 1 blocks
   ```

   These four bytes are the ones containing the single instance of class `Serial`. Since that instance is stored in a static variable, it is never deleted, and that's okay.
2. Use `const` wherever possible. In particular, the `print()` functions are generally `const` since they are not supposed to change the data that they are printing.
3. Minor modifications to class `SPtr` are permissible and necessary. For example, the `typedef` for `T` will need to be changed. Also, you might find it helpful to define `operator->()` to behave like the standard `->` operator behaves on ordinary C pointers. You might also want to define a function to return the C pointer `target` that the `SPtr` object manages.

I think I've tacked all three of these

6 Grading Rubric

Your assignment will be graded according to the scale given in Figure 3 (see below).

#	Pts.	Item
1.	4	All relevant standards from previous problem sets are followed regarding submission, identification of authorship on all files, and so forth. A well-formed `Makefile` or `makefile` is submitted that specifies compiler options `-O1 -g -Wall -std=c++17`. Running `make` successfully compiles and links the project and results in an executable file `blockchain`. Each function definition is preceded by a comment that describes clearly what it does.
2.	4	Sample input and output files are submitted that show the program behaves as expected. In particular, you should create a test file that grows a blockchain structure like the one in Figure 1 and then proceeds to replace some of the blockchains with others. An inaccessible block should be deleted automatically when the last smart pointer to it is deallocated. It may be useful to leave the `SPtr` debugging printout in place that shows when a block is deleted.
3.	4	The program shows good style. All functions are clean and concise. Inline initializations, inline functions, and `const` are used where appropriate. Variable names are appropriate to the context. Programs are consistently indented according to the course indenting style. Each class has a separate `.hpp` file and, if needed, a separate `.cpp` file.
4.	4	All of the functionality in section 3 is correctly implemented.
5.	4	Valgrind gives clean output with all storage blocks freed except for the instantiation of `Serial`.
	20	Total points.

Figure 3: Grading rubric.