Project 1: Blockchain Storage Structure

DUE: Sunday, _February 13 at 11:59 PM

Basic Procedures

You must:

- Fill out a readme.txt file with your information (goes in your user folder, example readme.txt provided).
- Have a style and pass the automatic style checker (see P0).
- Comment your code well in JavaDoc style AND pass the automatic JavaDoc checker (see P0).
- Have code that compiles with the command: **javac *.java** in your user directory

You may:

- Add additional methods and variables, however these methods **must be private**.

You may NOT:

- Make your program part of a package.
- Add additional public methods or variables
- Use any built in Java Collections Framework classes anywhere in your program (e.g. no ArrayList, LinkedList, HashSet, etc.).
- Alter any method signatures defined in this document of the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Add any additional import statements (or use the "fully qualified name" to get around this requirement).
- Add any additional libraries/packages which require downloading from the internet.

Setup

- Download the `project1.zip` and unzip it. This will create a folder `section-yourGMUUserName-p1`;
- Rename the folder replacing `section` with the `001`, `002`, `005` etc. based on the lecture section you are in;
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address;
- After renaming, your folder should be named something like: `001-jsmith-p1`.
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`)

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip `section-username-p1.zip` (no other type of archive) following the same rules for `section` and `username` as described above.
    - The submitted file should look something like this:

```
001-jsmith-p1.zip --> 001-jsmith-p1 --> JavaFile1.java
                                          JavaFile2.java
```

- Submit to blackboard.

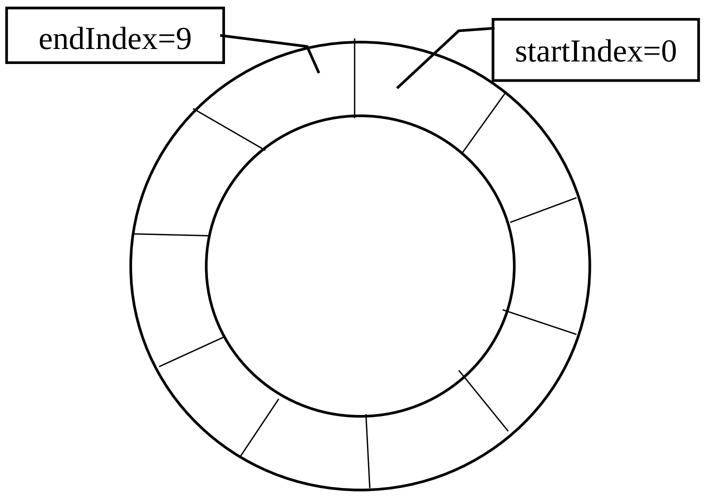
Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading..

Overview

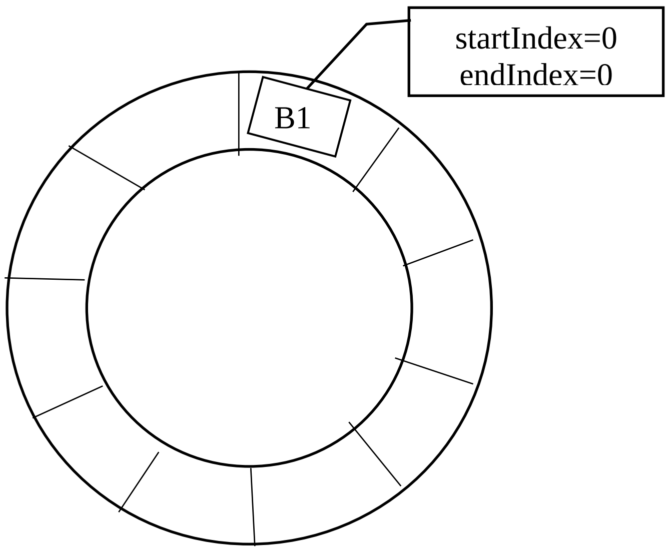
A Blockchain (BC) is a growing list of records, called blocks, that are linked together using cryptography (wiki). Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. It is tamper-proof, decentralized, and transparent. The timestamp proves that the transaction data existed when the block was published in order to get into its hash. As blocks each contain information about the block previous to it, they form a chain, with each additional block reinforcing the ones before it. Therefore, blockchains are resistant to modification of their data because once recorded, the data in any given block cannot be altered retroactively without altering all subsequent blocks. BC is completely decentralized across the network. This means that there is no master node, and every node in the network has the same copy. The transaction data can be currency, health data, identity information, etc. Bitcoin is by far the most successful implementation of blockchain technology.

In this project we will develop a simple array-based data structure that can be used to store the blocks in the blockchain technology temporarily before they are added in the chain. The data structure will be generic in such a way that the transaction in a block can be any kind of data. This BC block storage structure has special properties. The data structure has one entry point (called **endIndex)** and one exit point (called **startIndex)**. New blocks can only be added on the storge structure through the entry (endIndex) and old blocks can only be archived (removed) from the exit point (startIndex). Individual blocks are added based on their timestamp; the first one to be created will be the first one to be archived (i.e., oldest blocks will be archived first). The storage structure has a circular shape with a mark on the **end** and **start**. Consider the array storage structure is initially (empty), has the last index (n-1) as endIndex and the first index (0) as a startIndex. Take a look at the following diagram for an empty BC storage with a capacity of 10.

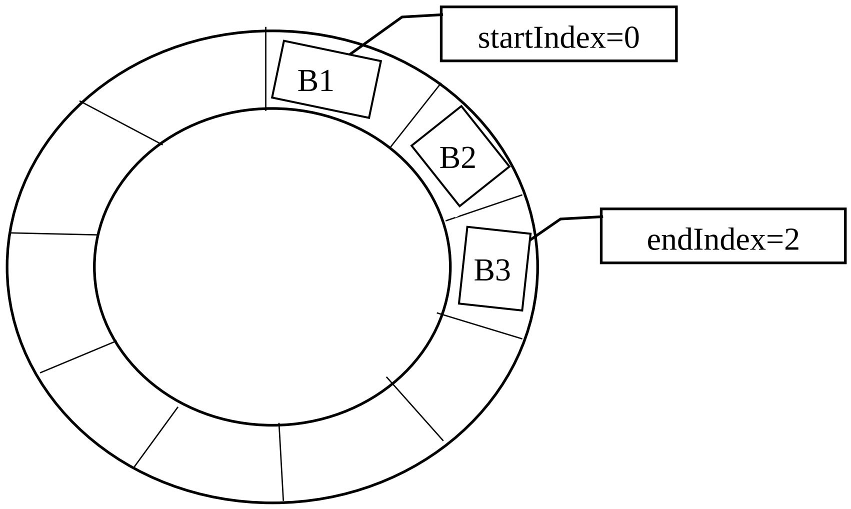


When the first block is created (we don’t care how it is created for this project) and ready to be stored, it will be stored at the end. For that to happen, first the **endIndex** will be incremented and point to the spot the new block will be added. Let see how new blocks are added and how old ones are archived.

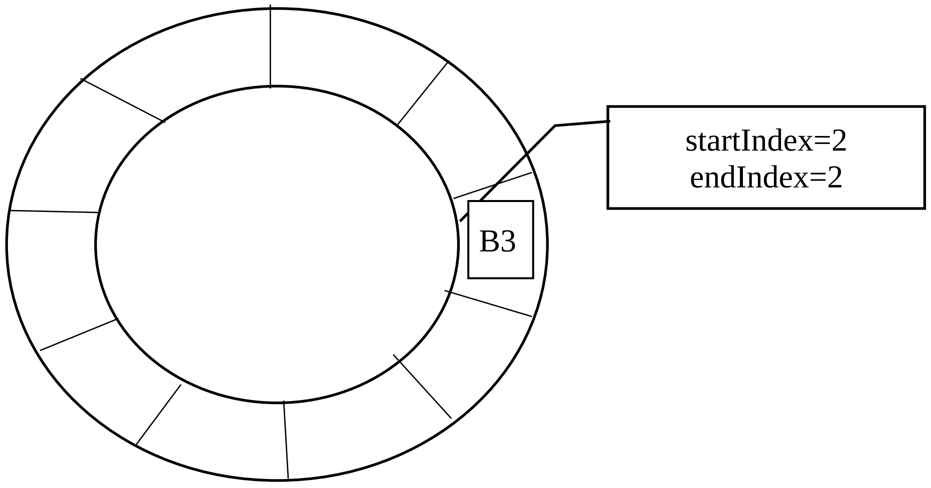
Take a look at the following diagram when the first block is added. Note the increment of **endIndex**.



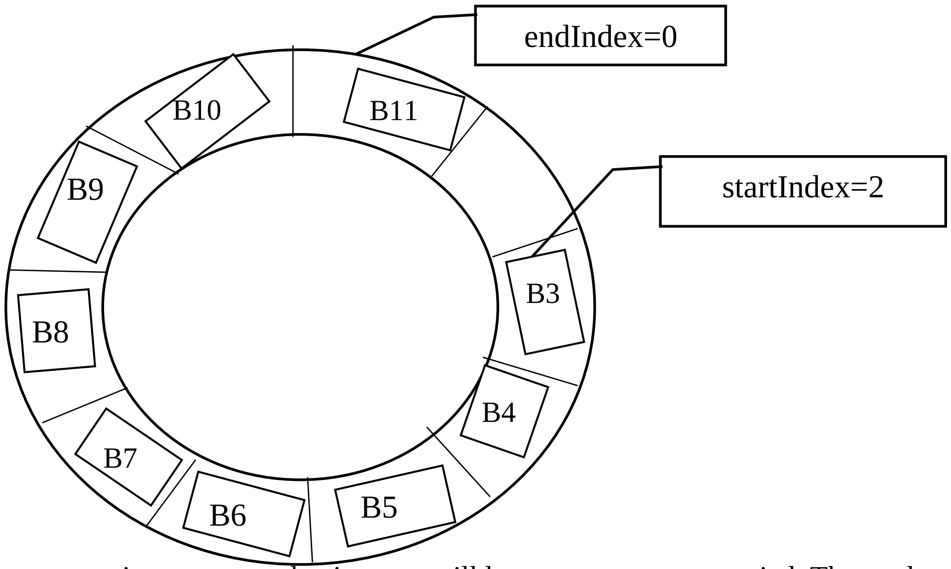
After two more additions:



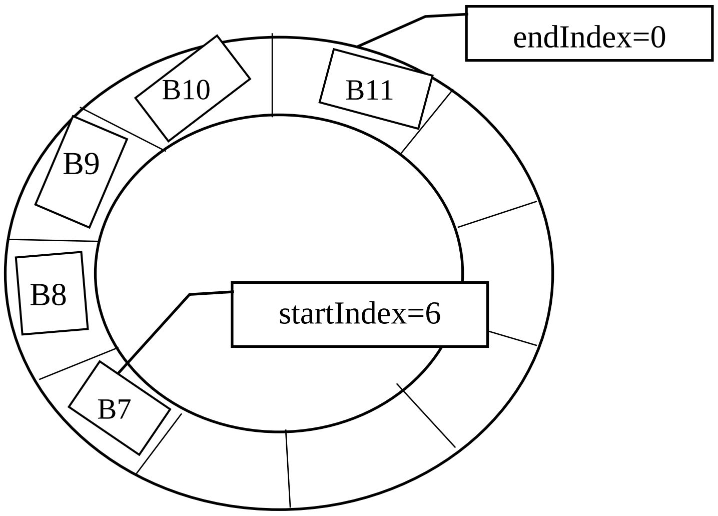
When it is time to archive an old block, the archiver will start from the start and move towards the end, removing one block at a time. The following diagram shows, after two blocks are removed.



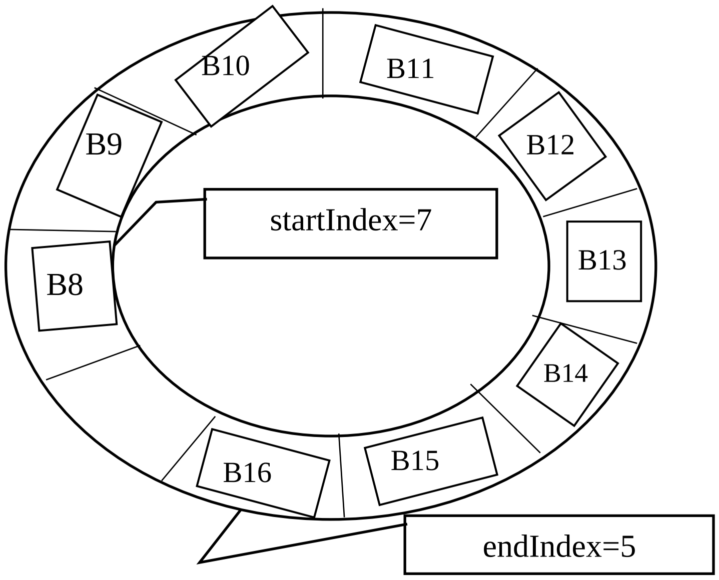
After 8 new blocks join the storage structure:



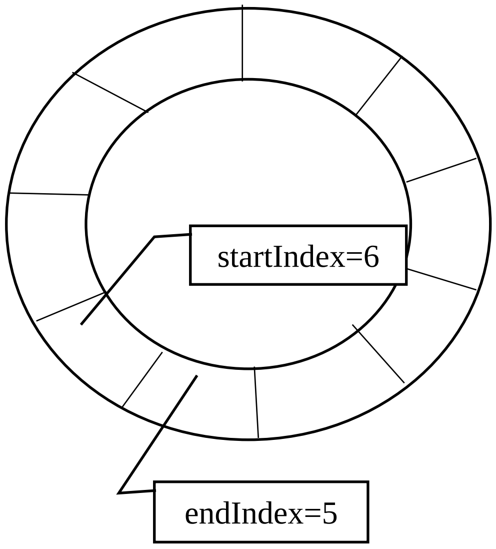
In order to enforce some maintenance mechanism, we will leave one spot unoccupied. That makes the above structure full. In order for a new block to be added, some old block should be archived(removed from the storage). The following is after 4 removals.



After 2 additions followed by 1 removal, followed by 3 additions, we will have a full storage structure again.



When we archive 9 blocks we will have:



In order to accommodate a surge in block storage, the structure can extend its capacity by doubling the number of slots. For example, in this specific example it can extend itself to accommodate up to 19 individuals (double the size and one empty slot requirement).

Implementation/Classes

This project will be built using a number of classes/interfaces representing the component pieces of the Block Store (BS) described in the previous section. Here we provide a description of these classes. Template files are provided for each class in the project package and these contain further comments and additional details.

INTERFACE (CircularBSInterface): **`CircularBSInterface.java`**

This interface represents all the functionalities of the array based storage structure. Since we want the storage structure to be used to store any kind of Block/Transaction, all the operations are generic. So, the return type and/or parameter types of some of the following methods are purposely omitted.

Operations

`+insert(newBlock): void` - This method adds a new block to the storage structure. If the structure is full, it will double the size to accommodate the new block. Note that the data type of the newBlock is not specified.

`+archive():` - this method archives/removes and returns the first block in the structure. For our specific example, it removes the block that is in structure the longest. If the structure is empty, it throws a NoBlockException.

`+archiveAll(): void` - removes all blocks in the structure . It throws NoBlockException, if there is no block to remove(i.e. it is empty).

+getFront() - returns the first block currently in the storage structure (without removing the block) . If nothing is in the storage structure it will throw NoBlockException.

+getBack() - returns the last block currently in the storage structure (without removing the block). If nothing is in the storage structure it will throw NoBlockException.

+getStorageCapacity():int – returns the maximum storage capacity. For example, it returns the maximum number of blocks that the storage structure can accommodate.

+size():int - returns the number of blocks in the storage structure at a time.

+isEmpty():boolean – checks if the storage structure empty.

+isFull():boolean – checks if the storage structure is full.

Note that this interface should be generic, i.e., it should be able to accommodate any type of block. The blockchain is just an example.

CLASSNAME (CircularBS): CircularBS.java

This class implements the CircularBSInterface using the storage structure that behaves as described in the previous section. The storage structure uses an array as an implementation data structure. Note that the underline structure uses **startIndex** and **endIndex** to keep track of the first and last elements in the storage structure. The class has the following additional behaviors:

Constructors

+CircularBS(): – creates an empty circularBS with a default capacity of 50.

+CircularBS(capacity:int): - creates an empty circularBS with a capacity of capacity.

+doubleCapacity():void – This method doubles the size (the array length) of the CircularBS, if it is full. The already existing blocks should not be lost during the resizing.

+getStart():int – returns the position of **startIndex.**

+getEnd():int – returns the position of **endIndex.**

The big-O of your methods' implementations can be as follows:

Method Name	Big-O	Remark
insert()	O(n)	O(1) amortized complexity
archive()	O(1)	
archiSveAll	O(n)	
getFront()	O(1)	
getBack()	O(1)	
getStorageCapacity()	O(1)	
size()	O(1)	
isEmpty()	O(1)	
isFull()	O(1)	
doubleCapacity()	O(n)	
getStart()	O(1)	
getEnd()	O(1)	

CLASSNAME (PriorityBS): PriorityBS.java

This class implements the CircularBSInterface. The class uses some priority for its operations. For instance, for the blockchain, certain transactions with higher incentives will have priorities to get processed, irrespective of their arrival time. In general, the storage structure will serve/process a high priority element before it serves a low priority element. Other than this priority, the PriorityBS **is a** CircularBS. Note that to compare two objects of a class in Java, the class need to implement a specific built-in Java interface.

Constructors

+PriorityBS(): - creates an empty PriorityBS with a default capacity of 50.

+PriorityBS(capacity:int): - creates an empty PriorityBS with a capacity of capacity.

+doubleCapacity():void - This method doubles the size(the array length) of the PriorityBC, if it is full. The already existing blocks should not be lost during the resizing.
+getStart():int - returns the position of **startIndex.**

+getEnd():int - returns the position of **endIndex.**

+toString():String - returns the string representation of the data in the storage structure separated by comma. For example, if the data in the storage structure are Brown, Minilik, Adams, Lawson, the method returns [**Adams,Brown,Lawson,Minilik] (Note the ordering).** If the storage structure is empty, it will return **[].**

The big-O of your method implementation can be as follows:

Method Name	Big-O	Remark
insert()	O(n)	

archive()	O(1)	
archiveAll()	O(n)	
getFront()	O(1)	
getBack()	O(1)	
getStorageCapacity()	O(1)	
size()	O(1)	
isEmpty()	O(1)	
isFull()	O(1)	
doubleCapacity()	O(n)	
toString()	O(n)	
getStart()	O(1)	
getEnd()	O(1)	

CLASSNAME (NoBlockException): NoBlockException.java

Some of the operations in the classes, throw exception. For example, we cannot test an empty storage structure (remove a block from an empty structure). NoBlockException handles this error by returning “No Block to process” string. NoBlockException is **unchecked** exception.

Requirements

An overview of the requirements are listed below, please see the grading rubric for more details.

- **Implementing the classes** - You will need to implement required classes and interface
- **JavaDocs** - You are required to write JavaDoc comments for all the required classes and methods. Check provided classes for example JavaDoc comments.
- **Big-O** – The method in each class should be implements according the REQUIRED Big-O runtime shown in the tables above. Your implementation of those methods should NOT have a higher Big-O.

Testing

The main methods provided in the template files contain useful code to test your project as you work. You can use command like "`java PriorityBSMainTester/ java CircularBSMainTester`" to run the testing defined in `main()`. You could also edit `main()` to perform additional testing. JUnit test cases will not be provided for this project, but feel free to create JUnit tests for yourself. A part of your grade *will* be based on automatic grading using test cases made from the specifications provided.