Stanford CS149: Parallel Computing
Written Assignment 4

Implementing CS149 Spark

Problem 1. (40 points):

In this problem we want you to implement a *very simple* version of Spark, called CS149Spark, that supports only a few operators. You will implement CS149Spark as a simple C++ library consisting of a base class `RDD` as well as subclasses for all CS149Spark transforms.

```
  class RDD {
    public:
    virtual bool hasMoreElements() = 0;   // all RDDs must implement this
    virtual string next() = 0;            // all RDDs must implement this

    int count() {                  // returns number of elements in the RDD
      int count = 0;
      while (hasMoreElements()) {
         string el = next();
         count++:
      }
      return count;
    }

    vector<string> collect() {     // returns STL vector representing RDD
      vector<string> data;
      while (hasMoreElements()) {
         data.append(next());
      }
      return data;
    }
};

class RDDFromTextFile : public RDD {
  ifstream inputFile;             // regular C++ file IO object
  public:
  RDDFromTextFile(string filename) {
    inputFile.open(filename);     // prepares file for reading
  }

  bool hasMoreElements() {
    return !inputFile.eof();      // .eof() returns true if no more data to read
  }

  string next() {
    return inputFile.readLine();  // reads next line from file
  }
};
```

For example, given the two definitions above, a simple program that counts the lines in a text file can be written as such.

```
  RDDFromTextFile r("myfile.txt'');   // creates an RDD where each element is a string
                                      // corresponding to a line from the text file

  printf("The RDD has length %d\n", r.count());
```

A. (8 pts) Now consider adding a `l33tify` RDD transform to CS149Spark, which returns a new RDD where all instances of the character 'e' in string elements of the source RDD are converted to the character '3'. For example, the following code sequence creates an RDD (`r1`) whose elements are lines from a text file. The RDD `r2` contains a l33tified version of these strings. This data is collected into a regular C++ vector at the end of the program using the call to `collect()`.

```
RDDFromTextFile r1("myfile.txt'');    // creates an RDD where each element is a string
                                      // corresponding to a line from the text file

RDDL33tify r2(r1);                    // l33tify all elements for r1
vector<string> lines = r2.collect();  // lines from the file, but in l33t form
```

Implement the functions `hasMoreElements()` and `next()` for the `l33tify` RDD transformation below. **A full credit solution will use minimal memory footprint and never recompute (compute more than once) any elements of any RDD.**

```
////////////////////////////////////////////////////////////
class RDDL33tify : public RDD {

  RDD parent;


  RDDL33tify(RDD parentRDD) {
    parent = parentRDD;



  }

  bool hasMoreElements() {









  }

  string next() {









  }
};
```

B. (10 pts) Now consider a transformation `FilterLongWords` that filters out all elements of the input RDD that are strings of greater than 32 characters.

Again, we want you to implement `hasMoreElements()` and `next()`.

You may declare any member variables you wish and assume `.length()` exists on strings. **Careful: `hasMoreElements()` is trickier now! Again a full credit solution will use minimal memory footprint and never recompute any elements of any RDD.**

A sample program using the `FilterLongWords` RDD transformation is below:

```
  RDDFromTextFile r1("myfile.txt'');   // creates an RDD where each element is a string
                                       // corresponding to a line from the text file

  RDDL33tify r2(r1);                   // converts elements to l33t form
  RDDFilterLongWords r3(r2);           // removes strings that are greater than 32 characters
  print("RDD r3 has length %d\n", r3.count());

////////////////////////////////////////////////////////////////////
class RDDFilterLongWords : public RDD {

  RDD parent;


  public:
  RDDFilterLongWords(RDD parentRDD) {
    parent = parentRDD;


  }

  bool hasMoreElements() {









  }

  string next() {









  }
};
```

C. (10 pts) Finally, implement a `groupByFirstWord` transformation which is like Spark's `groupByKey`, but instead (1) it uses the first word of the input string as a key, and (2) instead of building a list of all elements with the same key, concatenates all strings with the same key into a long string.

For example, `groupByFirstWord` on the RDD ["hello world", "hello cs149", "good luck", "parallelism is fun", "good afternoon"] would produce the RDD ["hello world hello cs149", "good luck good afternoon", "parallelism is fun"].

Your implementation can be rough pseudocode, and may assume the existence of a dictionary data structure (mapping strings to strings) to actually perform the grouping, an iterator over the dictionaries keys, and useful string functions like: `.first()` to get the first word of a string, and `.append(string)` to append one string to another.

Rough pseudocode is fine, but your solution should make it clear how you are tracking the next element to return in `next()`. A full credit solution will use minimal memory footprint and never recompute any elements of any RDD.

```
class RDDGroupByFirstWord : public RDD {
  RDD parent;
  Dictionary<string, string> dict;       // assume dict["hello''] returns the string
                                         // associated with key "hello''



  public:
  RDDGroupByFirstWord(RDD parentRDD) {
    parent = parentRDD;


  }

  bool hasMoreElements() {









  }

  string next() {









  }
};
```

D. (6 pts) Describe why the RDD transformations `L33tify`, `FilterLongWords`, and RDD construction from a file, as well as the action `count()` can all execute efficiently on very large files (consider TB-sized files) on a machine with a small amount of memory (1 GB of RAM).

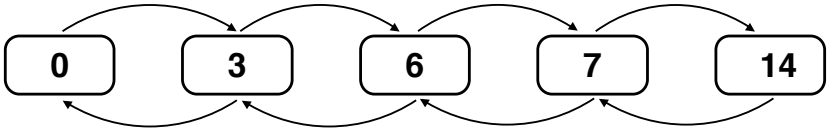
E. (6 pts) Describe why the transformation `GroupByFirstWord` differs from the other transformations in terms of how much memory footprint it requires to implement.

Concurrent Linked Lists

Problem 2. (30 points):

Consider a **SORTED doubly-linked list** that supports the following operations.

- `insert_head`, which traverses the list from the head. The implementation uses hand-over-hand locking just like in class.
- `delete_head`, which deletes a node by traversing from the head, using hand-over-hand locking just like in class.
- `insert_tail`, which traverses the list **backwards from the tail** to insert a node using hand-over-hand locking in the opposite order as `insert_head`.



A. (15 pts) Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

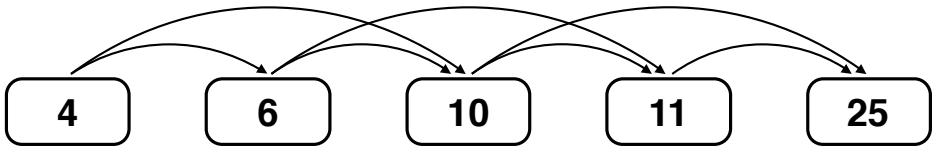
   - Test 1: `insert_head(2)`, `delete_head(14)`
   - Test 2: `insert_head(12)`, `delete_head(6)`
   - Test 3: `insert_head(13)`, `insert_tail(4)`

   The first two unit tests complete without error, but the third test goes badly and it does not terminate with the right answer. Describe what behavior is observed and why the problem occurs. (All unit tests start with the list in the state shown above.)

B. (15 pts) Imagine that locks in this system supported not only `lock()` and `unlock()`, but the ability to query the state of the lock via the call `trylock()` (this call takes the lock if the lock is free, but immediately returns false if the lock is currently locked – it does not block). Given this functionality, describe a fix to the problem you identified in part A? **Your answer should avoid livelock, but it is acceptable to allow for the possibility of starvation.**

More Pointers, More Problems

Problem 3. (30 points):



Consider the semi-skip list structure pictured above. Each node maintains a pointer to the next node and the next-next node in the list. **The list must be kept in sorted order.** A node struct is given below.

```
struct Node {
   int value;
   Node* next;
   Node* skip;   // note that skip == next->next
};
```

Please describe a thread-safe implementation of **node deletion** from this data structure. You may assume that deletion is the only operation the data structure supports. Please write C-like pseudocode.

To keep things simple, you can ignore edge-cases near the front and end of the list (assume that you're not deleting the first two or last two nodes in the list, and the node to delete is in the list). If you define local variables like `curNode`, `prevNode`, etc. just state your assumptions about them. **However, please clearly state what per-node locks are held at the start of your process. E.g., "I start by holding locks on the first two nodes.".** Full credit will only be given for solutions that maximize concurrency.

```
// delete node containing value
  void delete_node(Node* head, int value) {
```